В. В. Амосов

СХЕМОТЕХНИКА и СРЕДСТВА ПРОЕКТИРОВАНИЯ ЦИФРОВЫХ УСТРОЙСТВ

- *Схемотехника и проектирование сложных цифровых устройств, содержащих процессор*
- *Языки проектирования VHDL и Verilog HDL*
- *Средства проектирования фирм Actel, Altera, Cadence D. S., Mentor Graphics и применение этих средств*
- *Конфигурирование проектов разной сложности на ПЛИС*

CLR

УЧЕБНОЕ ПОСОБИЕ

bhv®

В. В. Амосов

СХЕМОТЕХНИКА
и СРЕДСТВА ПРОЕКТИРОВАНИЯ ЦИФРОВЫХ УСТРОЙСТВ

Рекомендовано Учебно-методическим объединением
по университетскому политехническому образованию
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлениям подготовки 220100 "Системный анализ
и управление" и 230100 "Информатика и вычислительная техника".

Санкт-Петербург

«БХВ-Петербург»

2007

УДК 681.3.06(075.8)
ББК 32.973-02я73
А61

Амосов В. В.

А61 Схемотехника и средства проектирования цифровых устройств. — СПб.: БХВ-Петербург, 2007. — 560 с.: ил. — (Учебное пособие)

ISBN 978-5-9775-0018-0

Приводится описание схемотехники цифровых устройств. Основное внимание уделяется обучению разработке программно-аппаратных комплексов, содержащих процессор: написание поведенческих и структурных VHDL и Verilog HDL-моделей, их тестирование и функциональное тестирование выполнения программ. Описывается современный инструментарий разработчика. На примерах дается описание использования этого инструментария. Каждая глава содержит упражнения или лабораторные работы, позволяющие закрепить теоретический материал. Достоинством книги является сочетание теории и практики, что позволяет легко освоить этапы разработки программно-аппаратных комплексов, включая тестирование как аппаратной, так и программной составляющих.

Для студентов вузов и специалистов-схемотехников

УДК 681.3.06(075.8)
ББК 32.973-02я73

Рецензенты:

Черноруцкий И. Г., д. т. н., профессор СПбГПУ
Александров А. М., д. т. н., профессор НПО «Импульс»

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Татьяна Лапина*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.08.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,15.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ОАО "Техническая книга"
190005, Санкт-Петербург, Измайловский пр., 29.

ISBN 978-5-9775-0018-0

© Амосов В. В., 2007
© Оформление, издательство "БХВ-Петербург", 2007

Оглавление

Введение **1**

ЧАСТЬ I. ТРАНЗИСТОРНАЯ СХЕМОТЕХНИКА. БАЗОВЫЕ СХЕМЫ СЕРИЙ ЭЛЕМЕНТОВ ЦИФРОВОЙ ТЕХНИКИ **3**

Глава 1. Основные понятия и методы анализа устройств транзисторной схемотехники **5**

1.1. Стрелки в схемах электронных цепей 5
1.2. Анализ цепи на основе системных и элементных законов 6
1.2.1. Системные законы или законы Кирхгофа (операциональная формулировка), границы применимости 6
1.2.2. Элементные законы — графические, математические модели компонентов цепи и модели в виде их схем замещения 7
1.2.3. Усилители 11
1.2.4. Анализ схем инвертора и усилительного каскада (УК) 12

Упражнения **16**

Упражнение 1.1. Понятия и теоремы для анализа (расчета) транзисторных схем 16
Определение идеальных элементов схем замещения 16
Теоремы эквивалентных преобразований 19
Задание 20
Упражнение 1.2. Анализ транзисторных схем в квазистатике 20
О коэффициентах усиления транзисторных схем по току, по напряжению, по мощности (K_I, K_U, K_P) 20
Формулы зависимостей между токами базы ($I_б$), коллектора ($I_к$) и эмиттера ($I_э$) 21
Задание 25

Глава 2. Простейшие схемы аналоговой техники, элементы цифровой техники **26**

2.1. Схемы аналоговой техники (УПТ и УПерТ) 26
2.2. Схемы элементов цифровой техники (ЦТ) 28
2.2.1. Квазистатический и динамический режимы работы 28
2.2.2. Классификация элементов цифровой техники (ЦТ) 31
2.2.3. Языки описания логических элементов 33

Лабораторная работа. Исследование инвертора, усилителей постоянного и переменного токов на биполярном транзисторе. Работа с системой Design Center ... 34

Работа с системой Design Center ... 36
Программа работы ... 38

Глава 3. Логики серий микросхем. Транзисторно-транзисторная логика (ТТЛ) ... 43

3.1. Логики серий микросхем ... 43
3.2. Транзисторно-транзисторная логика ... 44
3.2.1. Двухступенчатая логика ЛЭ ТТЛ ... 47
3.2.2. Третье состояние ЛЭ ТТЛ ... 47
3.2.3. ЛЭ ТТЛ с открытым коллектором ... 48

Лабораторная работа. Исследование базовой схемы логического элемента ТТЛ ... 49

Программа работы ... 50

Глава 4. Эмиттерно-связанная логика (ЭСЛ) ... 51

4.1. Базовая схема серий ЭСЛ ... 51

Лабораторная работа. Исследование базовой схемы логического элемента ЭСЛ ... 55

Программа работы ... 56

Глава 5. Логические элементы на МДП-транзисторах ... 59

5.1. МДП-транзисторы ... 59
5.1.1. Условное графическое изображение МДП-транзисторов ... 60
5.2. Комплементарные ЛЭ на МДП-транзисторах ... 61
5.2.1. О нагрузочной способности ЛЭ на КМДП-схемах ... 62
5.2.2. Базовые схемы серий логики на КМДП-схемах ... 63

Лабораторная работа. Исследование базовой схемы логического элемента на КМДП-транзисторах ... 64

Программа работы ... 64

ЧАСТЬ II. КОМБИНАЦИОННЫЕ И ПОСЛЕДОВАТЕЛЬНОСТНЫЕ ЛОГИЧЕСКИЕ УСТРОЙСТВА, ИХ ОПИСАНИЕ НА ЯЗЫКЕ VHDL. СРЕДСТВО ПРОЕКТИРОВАНИЯ ФИРМЫ ALTERA — MAX+PLUS II ... 67

Глава 6. Комбинационные и последовательностные логические устройства ... 69

6.1. Проектирование КЛУ на примере разработки логической схемы комбинационного полусумматора ... 69
6.1.1. Проектирование логической схемы комбинационного одноразрядного двоичного сумматора (См) ... 72
6.2. Вентили. Вентильное проектирование ... 74
6.3. Минимизация переключательных функций логических устройств ... 75

Лабораторная работа. Система автоматизированного проектирования фирмы Альтера Max+Plus II. Схемы одноразрядных двоичных полусумматора и сумматора ... 76

Цель работы ... 77
Программа работы ... 77

Глава 7. Переходные процессы в комбинационных логических устройствах ... 82

7.1. Статические и динамические риски ... 82
7.2. Формирователи длительности импульсов ... 83
7.3. Генераторы симметричных и несимметричных импульсов ... 85

Лабораторная работа. Исследование появления рисков. Генераторы и формирователи импульсов ... 86

Цель работы ... 86
Программа работы ... 86

Глава 8. Запоминающие элементы логических устройств ... 89

8.1. Описание функционирования ЗЭ ЛУ (спецификация простейшего ЗЭ ЛУ) ... 89
8.2. Внутренняя структура ЗЭ ... 90
8.2.1. RS-триггер на "ИЛИ-НЕ" ... 91
8.2.2. RS-триггер на "И-НЕ" ... 93
8.3. D-триггеры ... 93
8.3.1. DV-триггер ... 95
8.3.2. Двухтактный (двухфазный) D-триггер ... 95
8.4. Начальные сведения об описании ЛУ на языке VHDL ... 96
8.4.1. Синтезируемость кода на языке VHDL ... 97

Лабораторная работа. Программирование на VHDL в среде Max+Plus II RS- и D-триггеров ... 100

Цель работы ... 100
Программа работы ... 100

Глава 9. О необходимости тактирования схем ЦУ. Последовательные и параллельные регистры ... 104

9.1. Переключение RS-триггера на "ИЛИ-НЕ" во времени ... 104
9.2. Системы синхронизации на примере работы регистров сдвига (последовательных регистров) ... 105
9.2.1. Схема однотактного последовательного регистра ... 105
9.2.2. Схема двухтактного последовательного регистра ... 107
9.3. Параллельные регистры (регистры памяти) ... 109
9.4. Универсальные (параллельные, последовательные и реверсивные) регистры ... 109

Лабораторная работа. Исследование схем сдвигающего регистра, регистра памяти и универсального регистра ... 110

Цель работы ... 110
Программа работы ... 110

Глава 10. Счетные элементы (СЭ) или элементы счетчиков 118

10.1. Описание функционирования двоичного СЭ (спецификация простейшего СЭ) 118
10.2. Внутренняя структура СЭ 119
10.2.1. Т-триггер с задержкой 120
10.2.2. JK-триггер с задержкой 120
10.2.3. Двухтактный Т-триггер 121
10.3. RST-триггер 124
10.4. Современные JK-триггеры 124
10.5. Двоичные счетчики (делители) 125
10.5.1. Схема выделения считаемых импульсов 126

Лабораторная работа. Исследование схем Т- и JK-триггеров и схемы счетчика 126

Цель работы 126
Программа работы 127

ЧАСТЬ III. ФУНКЦИОНАЛЬНЫЕ УСТРОЙСТВА ЦТ СРЕДНЕЙ ИНТЕГРАЦИИ, ИХ ОПИСАНИЕ НА ЯЗЫКЕ VHDL 131

Глава 11. Дешифраторы (DC), особенности языка VHDL 133

11.1. Описание функционирования DC (спецификация простейшего DC) 133
11.2. Проектирование DC 134
11.3. Увеличение разрядности DC 135
11.4. Получение произвольных логических функций (ЛФ) в СДНФ с использованием DC и ЛЭ "ИЛИ" 136
11.5. Особенности языка VHDL 137

Лабораторная работа. Исследование функционирования схем дешифраторов 139

Цель работы 139
Программа работы 139

Глава 12. Шифраторы 142

12.1. Описание функционирования приоритетного шифратора (HPRI) 142
12.2. Проектирование HPRI 143
12.3. Увеличение разрядности HPRI 145
12.4. Указатель наиболее приоритетного входа из всех тех, на которые пришли входные сигналы 145

Лабораторная работа. Исследование функционирования схем шифраторов и указателей 146

Цель работы 146
Программа работы 146

Глава 13. Мультиплексоры 148

13.1. Мультиплексоры (MUltipleXer — MUX) 148
13.1.1. Описание функционирования мультиплексора (спецификация простейшего MUX) 148

13.1.2. Проектирование MUX ... 149
13.1.3. Увеличение разрядности MUX ... 150
13.2. Демультиплексоры (DMX) ... 151
13.2.1. Проектирование DMX ... 152
13.3. Получение произвольных логических функций (ЛФ) с помощью MUX ... 152
13.3.1. Первый вариант реализации произвольных ЛФ на MUX ... 153
13.3.2. Второй вариант реализации произвольных ЛФ на MUX ... 154

Лабораторная работа. Исследование функционирования схем мультиплексоров и демультиплексоров ... 156

Цель работы ... 156
Программа работы ... 156

Глава 14. Компараторы (CMP) ... 158

14.1. Описание функционирования компаратора (CMP) (спецификация простейшего CMP) ... 158
14.2. Проектирование CMP ... 159
14.2.1. Реализация отношения "равенство" ... 159
14.2.2. Реализация отношения "больше" ... 160

Лабораторная работа. Исследование функционирования логической схемы компаратора для сравнения двухбитных слов ... 162

Цель работы ... 162
Программа работы ... 162

Глава 15. Устройства недопущения, обнаружения и исправления ошибок ... 163

15.1. Схемы контроля по модулю 2 (схемы свертки по модулю 2) ... 163
15.1.1. Проектирование схем контроля по модулю 2 ... 164
15.1.2. ИС КР1533ИП5 — схема контроля по модулю 2 ... 165
15.2. Схемы контроля, использующие мажоритарные элементы или элементы "голосования" (спецификация простейшего мажоритарного элемента) ... 166
15.2.1. Проектирование мажоритарного элемента ... 167

Лабораторная работа. Исследование функционирования схем контроля по модулю 2 и схемы мажоритарного элемента ... 167

Цель работы ... 167
Программа работы ... 168

Глава 16. Кодер и декодер Хемминга ... 170

16.1. Код Хемминга ... 170
16.1.1. Пример четырехразрядного кодирования Хемминга ... 171
16.1.2. Пример четырехразрядного декодирования Хемминга ... 172
16.2. Техническая реализация кодера и декодера Хемминга ... 172

Лабораторная работа. Исследование функционирования логической схемы кодера и декодера Хемминга ... 173

Цель работы ... 173
Программа работы ... 173

Глава 17. Многоразрядные сумматоры, арифметико-логические устройства (АЛУ) и умножители ... 175

17.1. Многоразрядные сумматоры ... 175
17.1.1. Комбинационный сумматор параллельного действия ... 175
17.1.2. Комбинационный сумматор последовательного действия ... 176
17.1.3. Накапливающий сумматор параллельного действия ... 176
17.1.4. Накапливающий сумматор последовательного действия ... 177
17.1.5. Схемные методы ускорения распространения переноса в многоразрядных параллельных сумматорах ... 178
17.2. Арифметико-логические устройства и блоки ускоренного переноса ... 180
17.3. Умножители параллельного действия (матричные умножители) ... 182
17.3.1. Увеличение разрядности матричных умножителей ... 184

Лабораторная работа. Исследование работы сумматоров, арифметико-логических устройств и умножителей ... 185

Цель работы ... 185
Программа работы ... 185

Глава 18. Схемы памяти ... 187

18.1. Иерархия ЗУ ... 187
18.2. Функциональная классификация ЗУ ... 188
18.3. Способы создания ЗУ ... 190
18.3.1. Простейший вариант структуры ЗУ с адресацией или с ПВ (статических ОЗУ (SRAM), ROM ЗУ) ... 191
18.3.2. Запоминающие элементы (ЗЭ) статической памяти (SRAM) ... 192
18.3.3. Запоминающие элементы динамической памяти (DRAM) ... 194

Лабораторная работа. Исследование функционирования схем памяти ... 195

Цель работы ... 195
Программа работы ... 195

Глава 19. Структуры построения специальных схем памяти, RAM и ROM ... 199

19.1. Структура кэшированной (CACHE) памяти ... 199
19.1.1. Структура полностью ассоциативной кэш-памяти ... 200
19.2. Структура схем памяти с последовательной выборкой ... 201
19.2.1. Структура циклических схем памяти (видеопамять) ... 201
19.2.2. Структура схем памяти, аналогичных регистрам сдвига (буферы FIFO и LIFO) ... 203
19.3. Структура схемы ROM на примере схемы ПЗУ ... 205
19.3.1. Проектирование с помощью схем ПЗУ ... 206

Лабораторная работа. Исследование функционирования схемы видеопамяти 207

Цель работы 207
Программа работы 208

Глава 20. Классификация и этапы разработки специализированных БИС 209

20.1 Базовые кристаллы (БК) 210
20.1.1. Конструкции БК 210
20.1.2. Терминология БК 211
20.2. Программируемые логические интегральные схемы (ПЛИС) 212
20.2.1. Устройства на программируемых логических матрицах (ПЛМ) 213
20.2.2. Устройства на программируемой матричной логике (ПМЛ) 215
20.3. Классификация ПЛСБИС 215

Лабораторная работа. Исследование функционирования схем регистров FIFO, LIFO и кэш-памяти 216

Цель работы 216
Программа работы 217

ЧАСТЬ IV. ПОВЕДЕНЧЕСКИЕ И СТРУКТУРНЫЕ VHDL-МОДЕЛИ СЛОЖНЫХ ЦИФРОВЫХ УСТРОЙСТВ НА ПРИМЕРЕ VHDL-МОДЕЛЕЙ ПРОЦЕССОРА DP-32, ИХ ТЕСТИРОВАНИЕ 219

Глава 21. Типы и уровни проектирования сложных цифровых устройств, концепции языка VHDL 221

21.1. Традиционные методы описания проектов 221
21.2. Типы и уровни описания сложных проектов 221
21.2.1. Области применения методов проектирования 223
21.3. Основные концепции языка VHDL 223
21.3.1. Тестирование 224
21.3.2. Анализ 226
21.3.3. Детализация 226
21.3.4. Симуляция 226
21.3.5. Синтез 228

Лабораторная работа. Приобретение навыков работы с системой Active-CAD и знакомство с программой VHDL Test Bench на примере VHDL-проекта АЛУ 229

Цель работы 229
Задачи 229
Программа работы 230
Active VHDL 230
АЛУ 230
Создание проекта 230
Симуляция с использованием механизма стимуляторов (stimulator) 239
Отслеживание версий 241
Генерация программы Test Bench 241

Глава 22. Поведенческая VHDL-модель процессора DP32. Ко-симуляция 244

22.1. Описание команд процессора DP32 244
22.1.1. Арифметические и логические команды DP32 245
22.1.2. Команды "чтение из памяти" и "запись в память" DP32 246
22.1.3. Команды "ветвления" DP32 247
22.1.4. Описание пакетов VHDL-модели процессора DP32, коды команд 248
22.2. Ко-симуляция и тестирование процессора DP32 250
22.2.1. VHDL-модель теста 251
22.2.2. Описание выполнения ко-симуляции на поведенческой модели DP32 252
22.2.3. Конфигурация для VHDL-модели теста поведенческой модели DP32 254
22.2.4. VHDL-модель генератора 255
22.2.5. VHDL-модель памяти 256
22.2.6. Описание тестовой программы процессора DP32 257
22.2.7. Описание DP32 поведенческой моделью 259

Лабораторная работа. Исследование с помощью системы Active-CAD поведенческой VHDL-модели процессора DP32. Ко-симуляция. Тестирование 269

Цель работы 269
Задачи 269
Программа работы 269

Глава 23. Архитектура и структурная VHDL-модель процессора DP32. Ко-симуляция 271

23.1. Архитектура процессора DP32 271
23.2. Описание выполнения ко-симуляции на структурной модели DP32 273
23.3. Конфигурация для VHDL-модели теста структурной модели DP32 275
23.4. Структурная VHDL-модель процессора DP32 277
23.4.1. Мультиплексор (MUX) 291
23.4.2. Защелка (Transparent Latch) 292
23.4.3. Буфер (Buffer) 293
23.4.4. Защелкивающий буфер (Latching Buffer) 293
23.4.5. Регистр *PC* (Счетчик команд) 295
23.4.6. Регистры общего назначения (Register File — массив регистров) 296
23.4.7. Компаратор (Condition Code Comparator) 298
23.4.8. Защелка (*Immed_signext*) 298
23.4.9. АЛУ 299

Лабораторная работа. Исследование с помощью системы Active-CAD структурной VHDL-модели процессора DP32. Ко-симуляция. Тестирование 300

Цель работы 300
Задачи 300
Программа работы 300

Глава 24. Обнаружение и исправление ошибок VHDL-моделей цифровых устройств. VHDL-модели современных процессоров ... 303

24.1. Обнаружение и пути исправления ошибок VHDL-моделей процессора DP32 ... 303
24.1.1. Обнаружение и пути исправления ошибок в поведенческой VHDL-модели процессора DP32 ... 303
24.1.2. Обнаружение и пути исправления ошибок в структурной VHDL-модели процессора DP32 ... 304
24.1.3. Обнаружение и пути исправления ошибок в VHDL-модели памяти ... 305
24.2. VHDL-модели современных процессоров ... 306
24.2.1. Реализация проекта конвейера команд на основе поведенческой модели процессора DP32 ... 307
24.2.2. Реализация защищенного режима ... 308

Лабораторная работа. Усовершенствование (развитие) VHDL-модели процессора DP32. Ко-симуляция. Тестирование ... 316

Цель работы ... 316
Программа работы ... 317

ЧАСТЬ V. ЯЗЫК ПРОЕКТИРОВАНИЯ VERILOG HDL. ПРИМЕРЫ, ИНСТРУМЕНТАРИЙ ... 319

Глава 25. Язык проектирования Verilog HDL ... 321

25.1. Структурное описание ... 321
25.1.1. Модули (Modules) ... 321
25.1.2. Макромодули (Macromodules) ... 322
25.1.3. Объявление портов (Port Definition) ... 323
25.1.4. Структура модуля ... 324
25.2. Функциональное описание ... 329
25.2.1. Последовательные операторы ... 329
25.2.2. Объявление функций ... 329
25.2.3. Описание функций и операторы языка Verilog HDL ... 332
25.2.4. Использование оператора задачи *task* ... 339
25.2.5. Оператор *always* ... 340

Лабораторная работа. Исследование Verilog HDL-проектов импульсного фильтра, параллельного регистра и АЛУ с помощью системы VeriLogger Pro/ TestBencher Pro ... 342

Цель работы ... 342
Описание работы с системой (пакетом) VeriLogger Pro / Testbencher Pro ... 342
Установка пакета ... 342
О пакете VeriLogger Pro / Testbencher Pro ... 342
Программа работы ... 346

Глава 26. Verilog HDL-проекты импульсного фильтра и параллельного регистра ... 347

26.1. Импульсный фильтр (спецификация проекта) ... 347
Текст Verilog HDL-проекта импульсного фильтра ... 349
26.2. Параллельный регистр (спецификация проекта) ... 351
Текст Verilog HDL-проекта параллельного регистра ... 352

Лабораторная работа. Исследование Verilog-проектов импульсного фильтра, параллельного регистра и АЛУ с помощью системы QUARTUS II ... 355

Цель работы ... 355
Описание работы с системой (пакетом) QUARTUS II ... 356
Установка пакета ... 356
О пакете QUARTUS II ... 356
Программа работы ... 359

Глава 27. Verilog HDL-проект арифметико-логического устройства (спецификация проекта) ... 361

27.1. Текст Verilog HDL-проекта АЛУ ... 363

Лабораторная работа. Исследование с помощью систем VeriLogger Pro/ TestBencher Pro и QUARTUS II Verilog-проекта, написанного по индивидуальному заданию ... 371

Цель работы ... 371
Примерные варианты индивидуальных заданий ... 371
Программа работы ... 372

ЧАСТЬ VI. СРЕДСТВА ПРОЕКТИРОВАНИЯ ФИРМЫ MENTOR GRAPHICS ... 373

Глава 28. Редакторы системного и архитектурного уровней (HDL Designer). Примеры использования ... 375

28.1. Оболочка Design Browser ... 376
28.2. Редактор Block Diagram ... 376
28.3. Редактор State Diagram ... 377
28.4. Редактор Flow Chart ... 378
28.5. Редактор Truth Table ... 379

Лабораторная работа. Знакомство с HDL-дизайнером на примере проекта "Таймер" ... 379

Цель работы ... 379
Спецификация проекта "Таймер" ... 379
Программа работы ... 384

Глава 29. Симулятор (Model Sim). Пример использования ... 396

29.1. Графический интерфейс пользователя Model Sim ... 396
29.1.1. Окно *Main window* ... 397
29.1.2. Окно *Dataflow* ... 398

Лабораторная работа. Симуляция проекта "Таймер" и устройств ЦТ с помощью Model Sim ... 398

Цель работы ... 398
Программа работы ... 398
Создание Test Bench ... 398
Вызов симулятора Model Sim ... 403

Глава 30. Синтез логических схем. Получение файлов для конфигурирования ПЛИС ... 405

30.1. Интерфейс системы Leonardo Spectrum. Элемент управления FlowTabs ... 406
30.1.1. Загрузка библиотеки технологий (вкладка *Technology*) ... 406
30.1.2. Чтение проекта (вкладка *Input*) ... 406
30.1.3. Установки синхронизации (вкладка *Constraints*) ... 407
30.1.4. Оптимизация проекта (вкладка *Optimize*) ... 408
30.1.5. Сохранение полученных результатов ... 408
30.2. Средство анализа схемы проекта Leonardo Insight ... 409

Лабораторная работа. Синтез проектов "Таймер" и устройств ЦТ ... 410

Цель работы ... 410
Программа работы ... 410

ЧАСТЬ VII. СРЕДСТВА ПРОЕКТИРОВАНИЯ ФИРМЫ ACTEL ... 413

Глава 31. Libero IDE. Возможные методы проектирования ... 415

31.1. Инструменты Libero ... 415
31.2. Управление проектом в Libero ... 417

Лабораторная работа. Знакомство на примерах со средством проектирования Libero IDE фирмы Actel ... 419

Цель работы ... 419
Программа работы ... 419
Шаг 1 — создание нового проекта ... 420
Шаг 2 — выполнение симуляции перед синтезом ... 420
Шаг 3 — синтез проекта в *Synplify* ... 422
Шаг 4 — размещение и трассировка ... 423
Шаг 5 — временная симуляция ... 424

Глава 32. Стартовый комплект для начала работы с ПЛИС Actel 425

32.1. Макетная плата ProASIC^{PLUS} Evaluation Board 425
32.1.1. Источник питания 426
32.1.2. Контакты программатора 427
32.1.3. Схема синхронизации 427
32.1.4. Подключение светодиодов 427
32.1.5. Подключение кнопок 428

Лабораторная работа. Пример проекта для ProASIC^{PLUS} Evaluation Board 428

Цель работы 428
Спецификация представленного проекта 428
Описание проекта блок-схемой 429
Задания для самостоятельной работы 440
Программа работы 441

Глава 33. Описание программируемых логических ИС (ПЛИС) фирмы Actel 442

33.1. Семейство ProASIC^{PLUS} 442
33.1.1. Характеристики микросхем серии ProASIC^{PLUS} 442
33.2. Архитектура ProASIC^{PLUS} 444
33.2.1. Ресурсы маршрутизации 445
33.2.2. Ресурсы синхронизации 448
32.2.3. Блоки ввода/вывода 449
33.2.4. Управление таймером и его характеристики 450
33.2.5. Защита проекта пользователя 451
33.2.6. Встроенная память 451
33.4. Семейство FPGA eX 451
33.4.1. Технология Antifuse 452
33.4.2. Описание семейства FPGA eX 453

Лабораторная работа. Программирование проектов в ПЛИС фирмы Actel 456

Цель работы 456
Программа работы 456
Описание программирования проектов в ПЛИС фирмы Actel 456

ЧАСТЬ VIII. СРЕДСТВА ПРОЕКТИРОВАНИЯ ФИРМЫ ALTERA ДЛЯ ЦИФРОВЫХ УСТРОЙСТВ СРЕДНЕЙ ИНТЕГРАЦИИ 459

Глава 34. Стартовый комплект для работы с ПЛИС Altera 461

34.1. Макетная плата UP2 Education Board 461
34.1.1. Состав платы 462

Лабораторная работа. Программирование и конфигурирование ПЛБИС в системе MAX+PLUS II фирмы Altera ... 467

Цель работы ... 467
Методические указания ... 467
Программа работы ... 469

Глава 35. Примеры проектирования и программирования в ПЛИС фирмы Altera ... 470

35.1. Пример реализации на ПЛИС фирмы Altera работы VGA-монитора ... 470
35.1.1. VGA-синхронизация ... 470
35.1.2. Использование ПЛИС для генерации видеосигнала VGA ... 472
35.1.3. Работа VGA в текстовом режиме ... 475
35.2. Пример реализации на ПЛИС фирмы Altera работы клавиатуры PS/2 ... 477
35.2.1. Протокол последовательной передачи данных PS/2 ... 477
35.3. Пример реализации на ПЛИС фирмы Altera работы мыши PS/2 ... 482
35.3.1. Модуль *Mouse* библиотеки *UPcore* ... 482

Лабораторная работа. Конфигурирование предложенных проектов в ПЛБИС фирмы Altera ... 483

Цель работы ... 483
Программа работы ... 483
Работа с монитором ... 483
Работа с клавиатурой ... 484
Работа с манипулятором "мышь" ... 484
Задания на следующую лабораторную работу (самостоятельное выполнение индивидуального проекта) ... 484

Глава 36. Описание ПЛИС FLEX 10K фирмы Altera ... 485

36.1. Общая характеристика семейства FLEX 10K ... 486
36.1.1. Основные компоненты структуры FLEX 10K ... 487
36.2. Конфигурирование и реконфигурирование ПЛИС семейства FLEX 10K ... 493

Лабораторная работа. Программирование (конфигурирование) индивидуального проекта в ПЛБИС фирмы Altera ... 494

Цель работы ... 494
Программа работы ... 494

ЧАСТЬ IX. СРЕДСТВА ПРОЕКТИРОВАНИЯ ФИРМЫ ALTERA ДЛЯ ЦИФРОВЫХ УСТРОЙСТВ БОЛЬШОЙ ИНТЕГРАЦИИ ... 495

Глава 37. Описание программируемых логических больших ИС (ПЛБИС) APEX фирмы Altera ... 497

37.1. Архитектура ПЛБИС семейства APEX20K ... 498

Лабораторная работа. Создание модели процессора Nios в среде QUARTUS 500

Цель работы 500
Программа работы 500

Глава 38. Excalibur — набор разработчика фирмы Altera 504

38.1. Описание отладочной платы 504
38.1.1. Цепь JTAG 505
38.1.2. Контроллер конфигурации 506
38.1.3. Перемычка JP2 507
38.1.4. Кнопка SW2: Reset 507
38.1.5. Кнопка SW3: Clear 507
38.1.6. Источник питания 507

Лабораторная работа. Конфигурирование на ПЛСБИС APEX проекта, содержащего процессор Nios, и программирование процессора 508

Цель работы 508
Программа работы 508
Конфигурирование проекта 508
Выполнение пользовательских программ 508
Пример запуска пользовательской программы 509

Глава 39. Встраиваемый процессор Nios 510

39.1. Архитектура процессора Nios 510
39.1.1. Регистровый файл 512
39.1.2. Арифметико-логическое устройство 512
39.1.3. Контроллеры 513
39.1.4. Память и организация ввода/вывода 513
39.1.5. Шины данных и команд 514
39.1.6. Кэш-память 514
39.2. Шина Avalon 514

Лабораторная работа. Создание индивидуальных программ для процессора Nios. Тестирование их на макетной плате 515

Программа работы 515

ПРИЛОЖЕНИЯ 517

Приложение 1. Основные элементы языка VHDL 519

Алфавит языка 519
Комментарии 520
Числа 520
Символы 520
Строки 520

Типы данных ... 520
Простые типы ... 521
Сложные типы ... 521
Основные элементы VHDL ... 522
Синтаксис ... 522
Характеристика объектов VHDL ... 522
Атрибуты ... 523
Компоненты ... 524
Операторы и выражения ... 524
Описание на VHDL объектов проекта: интерфейс, тело объекта и конфигурация ... 526
Описание задержек сигналов ... 528
Описание пакета в VHDL ... 528

Приложение 2. Операции языка Verilog HDL и примеры их применения ... 531

Основные операции Verilog HDL ... 531
Примеры применения операций Verilog HDL ... 532
Арифметические операции ... 532
Операции отношений ... 532
Операции совпадения, равенства ... 533
Операция сравнения (Handling Comparisons to X or Z) ... 533
Логические операции ... 533
Поразрядные операции ... 534
Операции сведения вектора к элементу поразрядными операциями ... 534
Операции сдвига ... 534
Операции условия ... 534
Операция конкатенации ... 535

Литература ... 537

Предметный указатель ... 538

Введение

Все современные сложные вычислительные системы — это программно-аппаратные комплексы. Для их разработки требуется слияние работы программиста и специалиста в области создания цифровой аппаратуры: программист должен писать программный продукт, наилучшим образом работающий на аппаратуре (тестирование на HDL-моделях аппаратуры), а специалист разрабатывать аппаратуру, наилучшим образом удовлетворяющую программному продукту: операционной системе, прикладным программам, пользовательским программам.

В предлагаемой книге делается акцент на обучение разработке программно-аппаратных комплексов, содержащих процессор: написание VHDL и Verilog HDL — моделей (поведенческих и структурных (RTL)) отдельных аппаратных частей этих комплексов; их тестирование; функциональное тестирование выполнения программ; на примерах рассматриваются особенности написания синтезируемого HDL-кода.

В книге каждая глава содержит подробное описание упражнений или лабораторных работ, закрепляющих теоретический материал по схемотехнике цифровых устройств. Описывается современный инструментарий разработчика. На примерах дается описание использования этого инструментария.

Вопросы схемотехники и средств проектирования цифровых устройств рассматриваются с азов до тонкостей, которые могут быть полезны профессиональным проектировщикам программно-аппаратных комплексов.

Особенность книги состоит в строгом соответствии теоретического материала и описания практикума, то есть теоретический материал — это подготовка к выполнению соответствующего практикума.

Книга состоит из девяти разделов и охватывает четыре направления.

- Схемотехника и проектирование сложных цифровых устройств, включающих процессор.
- Языки проектирования (VHDL и Verilog HDL), синтезируемость HDL-кода.

- Средства проектирования (инструментарий, платформы) фирм Actel, Altera, Cadence D. S., Mentor Graphics и методы (траектории) применения этих средств.
- Конфигурирование (программирование) проектов разной сложности на ПЛИС фирм Actel и Altera.

Описываются языки проектирования VHDL и Verilog HDL, и, начиная со второго раздела, в рамках практикума ведется обучение их использованию (от простого к сложному), особое внимание уделяется написанию синтезируемого VHDL- и Verilog HDL-кода.

После обучения созданию HDL-проектов, содержащих процессоры, с помощью командной симуляции описывается тестирование HDL-проектов, а также тестирование выполнения программ, написанных для этих проектов.

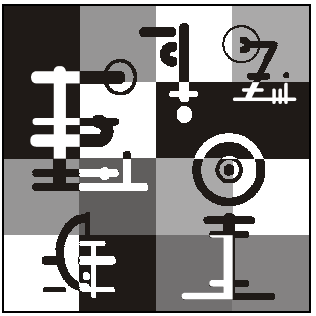
С помощью средств проектирования разных фирм описывается программирование HDL-проектов цифровых устройств в ПЛИС.

Содержание книги основано на материале курса лекций, читаемого автором на кафедре информационных и управляющих систем Санкт-Петербургского государственного политехнического университета.

Лабораторные работы к главам со 2 по 10 написаны автором совместно с Т. А. Вишневской и Е. Г. Локшиной, к главам с 31 по 39 совместно с А. С. Огневым.

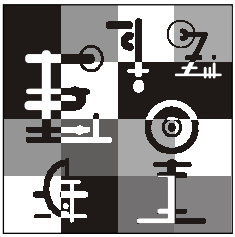
Автор признателен своим учителям, профессорам кафедры Т. К. Кракау, Б. Е. Аксенову, К. К. Гомоюнову за поддержку и помощь в выборе жизненного пути, за заложенный базис знаний.

Автор благодарит рецензентов за ряд ценных замечаний, способствовавших улучшению книги.



Часть I

Транзисторная схемотехника. Базовые схемы серий элементов цифровой техники



Глава 1

Основные понятия и методы анализа устройств транзисторной схемотехники

1.1. Стрелки в схемах электронных цепей

Принципиальная электрическая схема — это схематическое изображение реальной электрической цепи (реального устройства) (рис. 1.1).

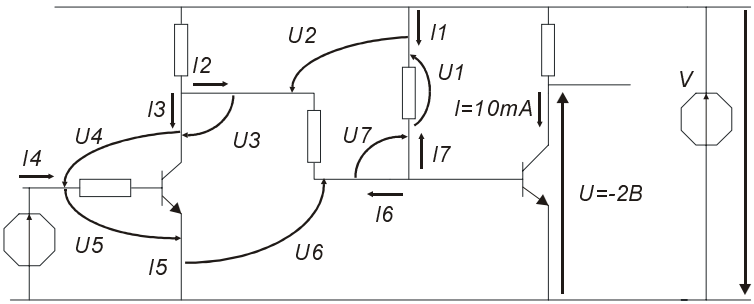


Рис. 1.1. Принципиальная электрическая схема

Стрелки замещают на схемах реальные измерительные приборы или, иначе, задают направление отсчета токов и напряжений.

Рассмотрим соответствие между направленностью стрелки и подключением измерительного прибора: A — амперметра и V — вольтметра (рис. 1.2).

Ток в цепи направлен от положительной обкладки источника к отрицательной. Его направление в ветвях цепи можно определить только после подключения измерительных приборов и расчета цепи.

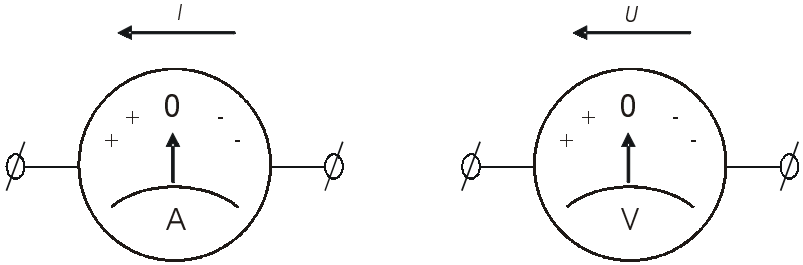


Рис. 1.2. Измерительные приборы

1.2. Анализ цепи на основе системных и элементных законов

Для анализа необходимо иметь информацию:

- о компонентах цепи, описываемых характеристиками или связями между основными величинами, показаниями подключенных к ним амперметров и вольтметров (это *элементные законы*);
- о законах, управляющих поведением устройства, объединяющего эти компоненты (это *системные законы*, в данном случае *законы Кирхгофа*).

1.2.1. Системные законы или законы Кирхгофа (операциональная формулировка), границы применимости

1 закон Кирхгофа:

$$\sum_{k=1}^{n} I_k(t) = 0$$

$$I_1 + I_2 + I_3 \ldots + I_n = 0$$

$$I_1 + I_2 - I_3' \ldots + I_n = 0$$

Подключение амперметров (I) представлено на рис. 1.3.

2 закон Кирхгофа:

$$\sum_{k=1}^{n} U_k(t) = 0$$

$$U_1 + U_2 + U_3 \ldots + U_n = 0$$

$$U_1 + U_2 - U_3' \ldots + U_n = 0$$

Подключение вольтметров (U) представлено на рис. 1.3.

Если на вход цепи подать переменный во времени сигнал, то для мгновенных показаний приборов в один и тот же момент времени законы Кирхгофа будут оставаться справедливыми, если продолжать увеличивать частоту переменного сигнала (то есть увеличивать скорости изменения dU_k/dt и dI_k/dt), то с некоторых частот мы обнаружим существенные отклонения от законов Кирхгофа.

Этими частотами определяется частотная граница представления о рассматриваемом электромагнитном устройстве как об электрической цепи.

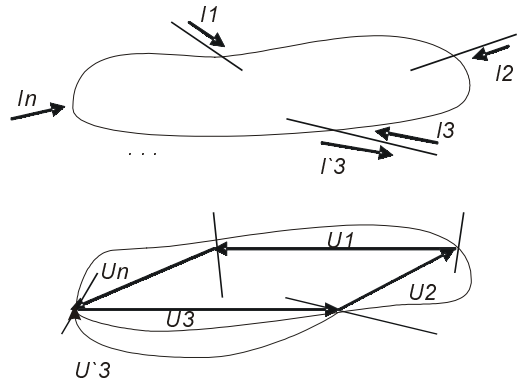


Рис. 1.3. Подключение измерительных приборов

1.2.2. Элементные законы — графические, математические модели компонентов цепи и модели в виде их схем замещения

Основные компоненты электронных цепей это — резисторы, конденсаторы, катушки индуктивности, трансформаторы, а также диоды и транзисторы разных типов. Компоненты классифицируют по числу выводов.

Резисторы, конденсаторы, катушки индуктивности, диоды — двухполюсники, транзисторы — трехполюсники, трансформаторы — четырехполюсники.

Основными характеристиками, определяющими свойства компонентов, являются связи между током и напряжением или их производными и интегралами по времени.

Вольт-амперная характеристика (ВАХ) компонента цепи

Рассмотрим цепь постоянного тока и подключим к компоненту цепи (рис. 1.4) измерительные приборы.

При этом зависимость $I(U)$ для этого компонента называют его ВАХ. Эту зависимость можно получить, подключая элемент в разные схемы и фиксируя показания I и U, или измерить, используя следующую цепь (рис. 1.5).

Рис. 1.4. Компонент цепи постоянного тока

В результате измерения получаем следующую ВАХ (рис. 1.6).

Расположение ВАХ относительно осей тока и напряжения зависит от того, как были подключены измерительные приборы во время съема ВАХ.

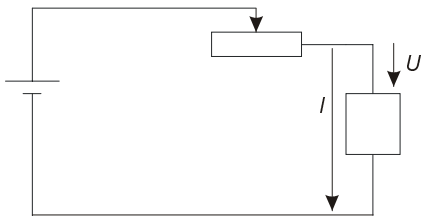


Рис. 1.5. Цепь определения ВАХ

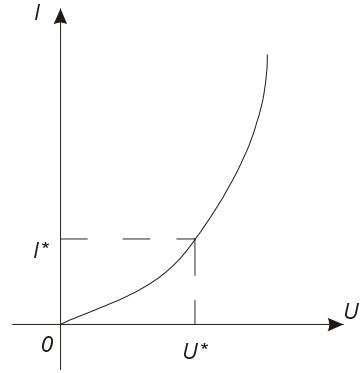


Рис. 1.6. ВАХ компонента цепи постоянного тока

Для расчета схемы необходимо перейти от компонентов цепи с реальными ВАХ к их графическим, математическим моделям и моделям в виде схем замещения. Чтобы предсказать поведение устройства, построенного из компонентов, необходимо перейти к какому-либо способу описания их поведения. Любой из этих способов есть модель данного компонента. Любая модель учитывает определенное, ограниченное число его свойств. Наилучшая модель — самая простая из тех, которые дают приемлемый по точности результат.

Соответствие реальных компонентов цепи (принципиальной электрической схемы) идеальным элементам схемы замещения:

- резистор (реальный компонент) соответствует сопротивлению (идеальному элементу);
- конденсатор — емкости;
- катушка индуктивности соответствует индуктивности;
- батарейка или сетевой источник соответствует идеальным источникам тока или напряжения;
- диод — идеальному вентилю;
- переключатель соответствует идеальному ключу (ВАХ короткого замыкания и ВАХ разрыва цепи).

Схема замещения — схема, состоящая из идеальных элементов. *Смешанная схема* — схема, состоящая как из идеальных элементов, так и из реальных компонентов.

Трехполюсники — компоненты цепи с тремя выводами (типы включения, семейства ВАХ)

Из законов Кирхгофа имеем: $I1 + I2 + I3 = 0$; $U13 + U32 + U21 = 0$.

Всего 6 величин, но независимых — 4 величины, так как две можно определить из законов Кирхгофа.

Обычно трехполюсник включают так, что один из его выводов является *общим* для выхода и входа (рис. 1.7). Тогда различают входные и выходные семейства ВАХ, описывающие поведение трехполюсника, каждое со своим параметром:

$I_{вх} = I_{вх}(U_{вх}, U_{вых})$, где $U_{вых}$ — параметр;

$I_{вых} = I_{вых}(U_{вых}, I_{вх})$, где $I_{вх}$ — параметр.

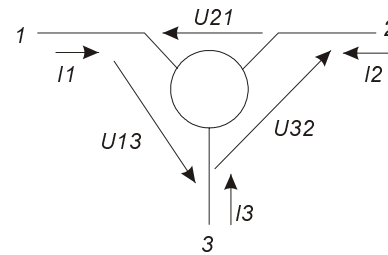


Рис. 1.7. Гипотетический трехполюсник

В качестве трехполюсников используются *транзисторы — биполярные* и *МДП* (металл-диэлектрик-п/проводник).

На рис. 1.8 показаны биполярные транзисторы различной проводимости (n-p-n и p-n-p). Направления стрелок на эмиттерах указывают нормальное протекание тока, например: в n-p-n — из базы в эмиттер и из коллектора в эмиттер; в p-n-p — наоборот.

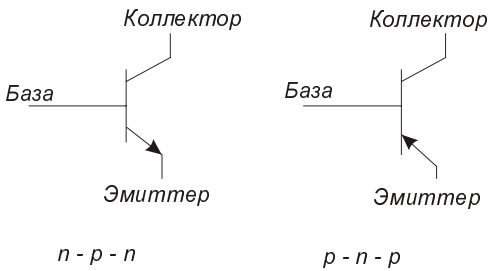


Рис. 1.8. Графическое обозначение биполярных танзисторов

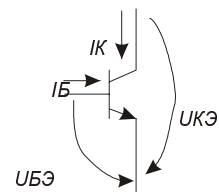


Рис. 1.9. Подключение измерительных приборов к n-p-n-транзистору

N-p-n-транзистор, для заданного подключения измерительных приборов (рис. 1.9), определяется входным (рис. 1.10) и выходным (рис. 1.11) семействами ВАХ.

Для транзистора, используя выходное семейство, можно построить еще одну зависимость (рис. 1.12), справедливую для всех $UKK \geq 0.1B : IK = IK(IБ)$.

Аппроксимирующая формула: $IK \cong B * IБ$.

Для каждого транзистора наклон, определяемый B, индивидуален.

Здесь B — коэффициент связи между выходным и входным токами в схеме с общим эмиттером (ОЭ) или иначе коэффициент передачи тока базы. Коэффициент B

велик (примерно равен нескольким десяткам), поэтому имеем усиление по току, но при этом нестабилен (может при нагреве транзистора измениться от 10 до 100), что приводит к искажению сигнала и потере информации.

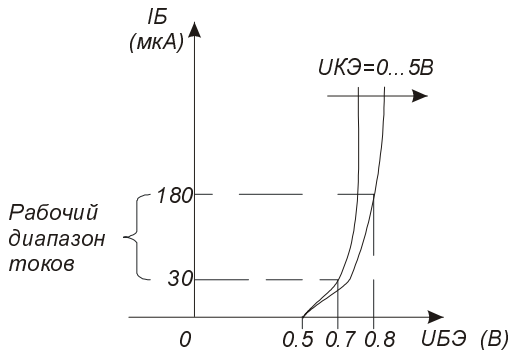


Рис. 1.10. Входное семейство ВАХ IБ (UБЭ, UКЭ)

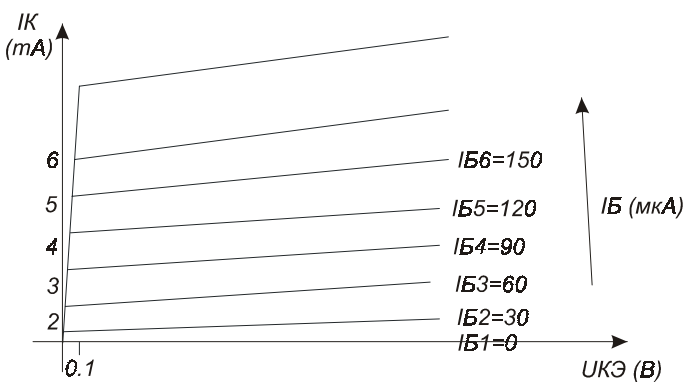


Рис. 1.11. Выходное семейство ВАХ IК (UКЭ, IБ)

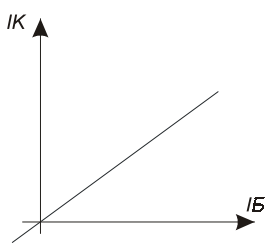


Рис. 1.12. Зависимость тока коллектора от тока базы

Графическая и аналитическая модели открытого и ненасыщенного биполярного транзистора

Транзистор может быть заперт ($IБ = IК = 0B$, $UБЭ < 0.7B$) или открыт, ненасыщен ($UБЭ = 0.7B$, $UКЭ \geq 0.1B$) или насыщен ($UБЭ = 0.7B$, $UКЭ = 0B$). Пусть транзистор открыт и ненасыщен: $UБЭ = 0.7B$, $UКЭ \geq 0.1B$.

Графическая модель открытого и ненасыщенного транзистора показана на рис. 1.13.

Его аналитическая модель:

Вход $UБЭа = 0.7B$

Выход $IKa \cong B * IББ$

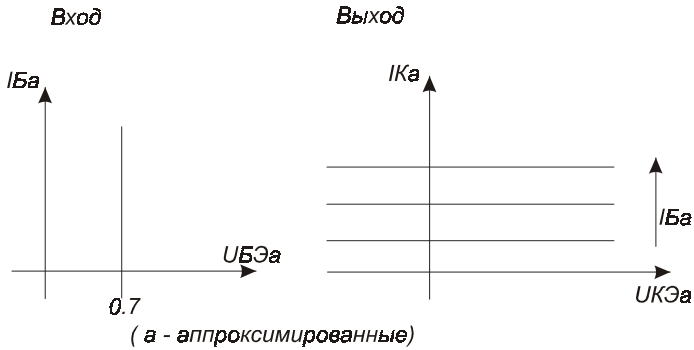


Рис. 1.13. Графическая модель открытого и ненасыщенного транзистора

1.2.3. Усилители

Рассмотрим ряд понятий:

- большинство устройств ВТ и АСУ предназначены для обработки поступающей на их вход информации;
- информация — это сведения или сообщения, но не что-то вещественное;
- материальный носитель информации — сигнал;
- сигнал — процесс, доступный для восприятия человеком;
- в электронной технике сигналом является изменение тока и напряжения во времени.

В любых электронных устройствах всегда имеет место усиление электрических сигналов. Под *усилением* понимается увеличение мощности сигнала: $P(t) = U(t) * I(t)$.

Каждый элемент вычислительного устройства должен быть, одновременно, и усилителем. Иначе сигнал, проходящий через элемент, затухнет, что приведет к потере информации.

Структура усилителя

На рис. 1.14 изображена структурная схема усилителя, в которой ИЭ — источник энергии, УЭ — управляющий элемент, стрелки — потоки энергии, определяемые мощностью источников (P).

Сущность усиления состоит в том, что: с помощью УЭ (управляющего элемента) происходит автоматическое управление отбором энергии от ИЭ (источника энергии) в нагрузку под управляющим воздействием энергии источника сигнала, причем энергия, поступающая в нагрузку, превосходит энергию от источника сигнала.

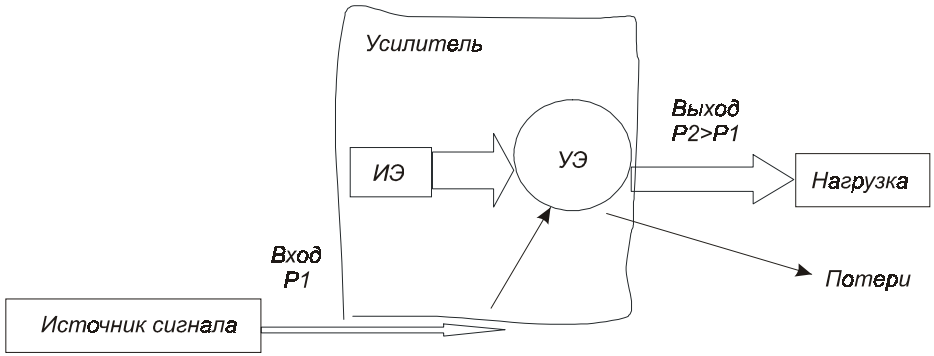


Рис. 1.14. Структурная схема усилителя

В аналоговой технике информация (звук, ТВ-сигнал) заложена в частоте изменения передаваемого сигнала (изменения тока и напряжения), и для того, чтобы услышать в приемнике звук без искажений, нужно пропорционально увеличить (усилить) амплитуды изменения тока и напряжения (то есть увеличить мощность сигнала), оставив частоту изменения сигнала (или информацию) без изменений.

1.2.4. Анализ схем инвертора и усилительного каскада (УК)

Инвертор

Инвертор — элемент ВТ, выполняющий логическую операцию отрицания.

Аналитическая модель работы инвертора

Транзистор в любой схеме (рис. 1.15) может находиться в трех состояниях.

1. Заперт:

$$U_{вх} < 0.7B;\ I_б = 0;\ U_{Rб} = I_б * R_б = 0;$$

$$U_{бэ} = U_{вх} < 0.7B;\ I_к = B * I_б = 0;$$

$$U_{вых} - ? \quad \left(U_{вых} - U_V + U_{Rк} = 0\right);$$

$$U_{вых} = V - I_к * R_к = V - B * I_б * R_к \Rightarrow U_{вых} = V.$$

2. Открыт и ненасыщен:

$$U_{вх} < 0.7B;\ I_б > 0;\ U_{бэ} = 0.7B = const;$$

$$U_{Rб} = U_{вх} - U_{бэ} = U_{вх} - 0.7B;$$

$$I_э = I_б + I_к;$$

$$I_б = U_{Rб}/R_б = \left(U_{вх} - 0.7\right)/R_б;$$

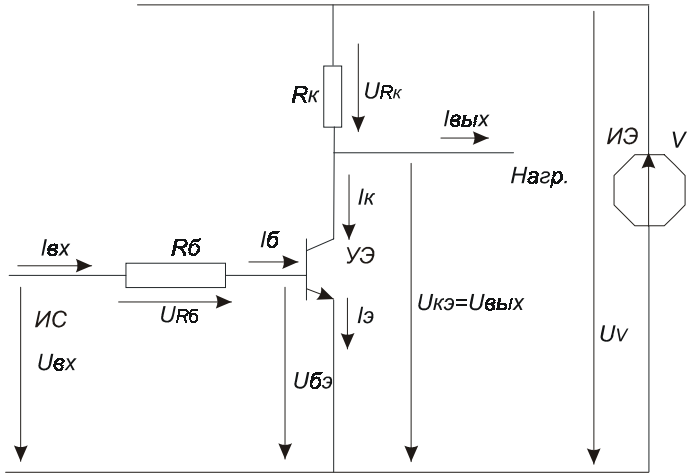


Рис. 1.15. Схема инвертора

$I_к = B * I_б = B * (U_{вх} - 0.7)/R_б$ — эта формула работает в диапазоне:

$0 \le I_к \le I_{кн}$ и $0 \le I_б \le I_{бн}$.

$I_{кн}$ — максимальный ток, который может протекать через коллектор-эмиттер транзистора в данной схеме. Этот случай соответствует закоротке между К и Э, то есть:

$I_{кн} = V/R_к \Rightarrow I_{бн} = I_{кн}/B = V/(R_к * B)$

$\Rightarrow U_{вых} = V - I_к * R_к = V - I_б * B * R_к = V - (U_{вх} - 0.7)/R_б * B * R_к$

$0 \le U_{вых} \le V$; $0.7 \le U_{вх} \le (I_{кн} * R_б/B + 0.7) = U_{вхн}$ получили из

$I_{кн} = B * I_{бн} = B * (U_{вхн} - 0.7)/R_б$ и $I_{кн} = V/R_к$.

3. Насыщен:

$U_{вх} > U_{вхн}$; $I_б > I_{бн}$; $U_{бэ} = 0.7B$;

$U_{Rб} = U_{вх} - U_{бэ} = U_{вх} - 0.7B$;

$I_б = U_{Rб}/R_б = (U_{вх} - 0.7)/R_б$;

$I_к = I_{кн} = V/R_к = const$;

$U_{вых} = V - I_к * R_к = V - V/R_к * R_к = 0$;

$I_э = I_б + I_к$.

Используя аналитическую модель поведения транзистора в данной схеме, изобразим зависимость $U_{вых}(U_{вх})$ или ХПН — характеристику передачи напряжения (рис. 1.16).

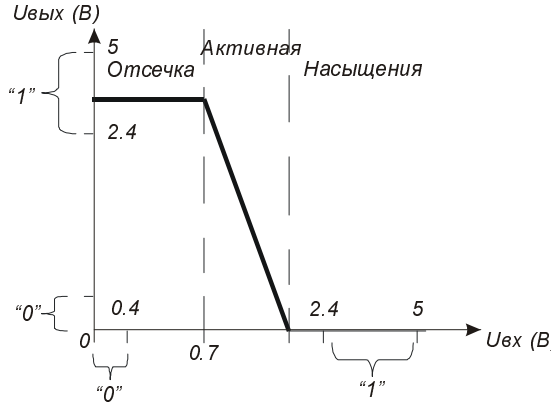


Рис. 1.16. Характеристика передачи напряжения

Усилительный каскад (УК)

УК — элемент аналоговой техники (одна ступень многокаскадного усилителя).

Если подать переменный во времени сигнал (синусоиду) на вход схемы инвертора (рис. 1.17), то на выходе получим постоянный уровень напряжения V, синусоида исчезнет, информационный сигнал пропадет, что приведет к потере информации, а это плохо.

Что делать?

Хорошо было бы сдвинуть ХПН влево (рис. 1.18), тогда синусоида на выходе не исчезнет, а ее амплитуда увеличится (эффект усиления), информация сохранится, сигнал увеличится, и это хорошо.

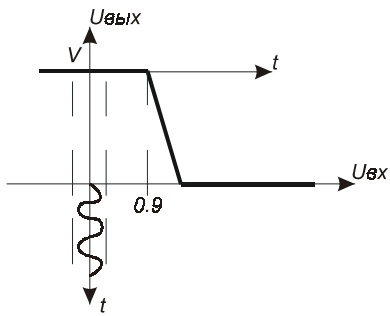


Рис. 1.17. Изображение сигнала на ХПН инвертора

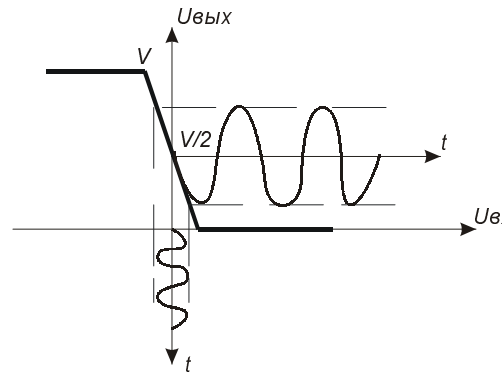


Рис. 1.18. ХПН инвертора с сопротивлением Rсм

Сдвиг ХПН достигается подключением сопротивления $R_{см}$ (см. рис. 1.19). Значение $R_{см}$ должно быть таким, чтобы при $U_{вх} = 0 \Rightarrow U_{вых} = V/2$.

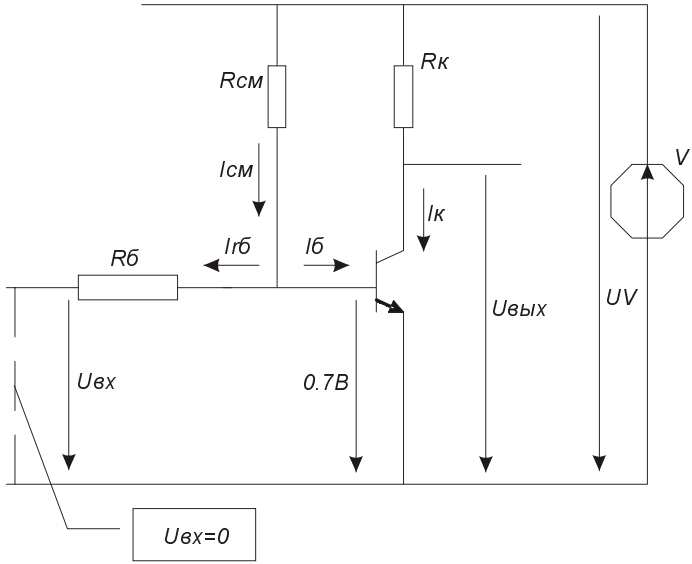


Рис. 1.19. Схема усилительного каскада

Схема УК

Рассмотрим схему усилительного каскада (рис. 1.19). Определим значение $R_{см}$:

$U_{вых_{0(U_{вх}=0)}} = V/2 = V - R_к * I_{к0}$, поэтому $I_{к0} = V/(2 * R_к)$. С другой стороны $I_{к0} = B * I_{б0}$,

следовательно, из двух последних получаем $V/(2 * R_к) = B * I_{б0}$.

Таким образом, из требования $U_{вых} = V/2$ следует, что $I_{б0} = V/(2 * R_к * B)$.

С другой стороны $I_{б0} = I_{см0} - I_{Rб0} = (V - 0.7)/R_{см} - 0.7/R_б$, следовательно, из двух последних получаем:

$(V - 0.7)/R_{см} - 0.7/R_б = V/(2 * R_к * B)$.

Отсюда $R_{см} = (V - 0.7)/(V/(2 * R_к * B) + 0.7/R_б)$.

Определение

Выбором рабочей точки называют поиск такого значения $R_{см}$, что при $U_{вх} = 0$ выходное напряжение ($U_{вых}$) будет равно $V/2$.

Упражнения

Упражнение 1.1. Понятия и теоремы для анализа (расчета) транзисторных схем

Рассматриваются основные понятия и теоремы, необходимые для анализа транзисторных схем.

Определение идеальных элементов схем замещения

Источник напряжения

Пусть ВАХ реального источника энергии (сетевого источника, батарейки) выглядит так (рис. 1.20).

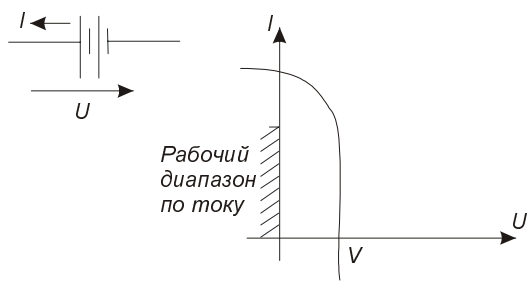


Рис. 1.20. ВАХ реального источника энергии

Тогда при расчете схемы нелинейный участок этой ВАХ можно не учитывать и реальный источник заменить на источник напряжения, имеющий следующие обозначения и ВАХ (рис. 1.21).

Здесь при любом токе: $U = V$.

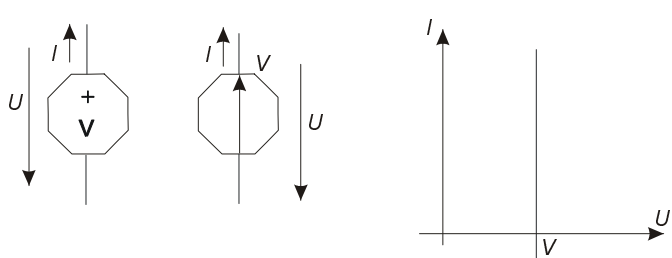


Рис. 1.21. Обозначения и ВАХ источника напряжения

Источник тока

Пусть ВАХ реального источника энергии (сетевого источника, батарейки) выглядит так (рис. 1.22).

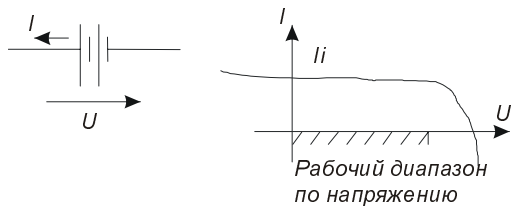


Рис. 1.22. ВАХ реального источника энергии

Тогда при расчете схемы нелинейный участок этой ВАХ можно не учитывать и реальный источник заменить на источник тока, имеющий следующие обозначения и ВАХ (рис. 1.23).

Здесь при любом напряжении $I = Ii$.

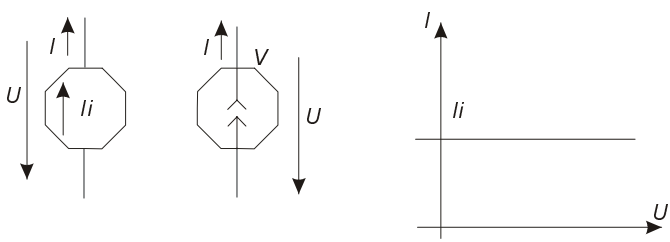


Рис. 1.23. Обозначения и ВАХ источника тока

Резистивный элемент (сопротивление)

До появления закона Ома каждый резистор описывался сложной нелинейной ВАХ, ее аппроксимировали (идеализировали, сделали линейной). После резистор описали линейной зависимостью $U = \kappa * I$ (закон Ома) и коэффициент пропорциональности назвали сопротивлением R (рис. 1.24).

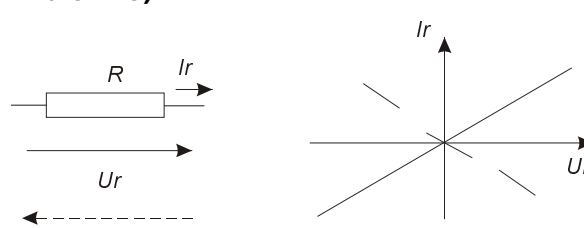


Рис. 1.24. Обозначение и ВАХ резистивного элемента

$Ur = +R * Ir$, при согласном включении измерительных приборов.

$Ur = -R * Ir$, при встречном включении измерительных приборов.

Емкостный элемент (емкость)

До появления закона Фарадея каждый конденсатор описывался сложной нелинейной вольт-кулонной характеристикой, ее аппроксимировали (идеализировали, сделали линейной). После конденсатор описали линейной зависимостью $Q = k * U$ (закон Фарадея) и коэффициент пропорциональности назвали емкостью C (рис. 1.25).

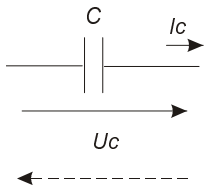


Рис. 1.25. Обозначение емкостного элемента

$Ic = +C * dUc/dt$; $Q = +C * Uc$, при согласном подключении измерительных приборов.

$Ic = -C * dUc/dt$; $Q = -C * Uc$, при встречном подключении измерительных приборов.

Индуктивный элемент (индуктивность)

До появления закона Генри каждая индуктивная катушка описывалась сложной нелинейной вольтсекунд-амперной характеристикой, ее аппроксимировали (идеализировали, сделали линейной). После индуктивную катушку описали линейной зависимостью $\sigma = k * I_L$ (закон Генри) и коэффициент пропорциональности назвали индуктивностью L (рис. 1.26).

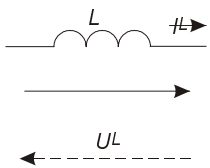


Рис. 1.26. Обозначение индуктивного элемента

$U_L = +L * dI_L/dt$; $\sigma = +L * I_L$ при согласном подключении измерительных приборов.

$U_L = -L * dI_L/dt$; $\sigma = -L * I_L$ при встречном подключении измерительных приборов.

Идеальный вентиль

Диод описывался сложной нелинейной ВАХ. Для упрощения описания его ВАХ кусочноаппроксимировали. Полученный элемент назвали идеальным вентилем (рис. 1.27).

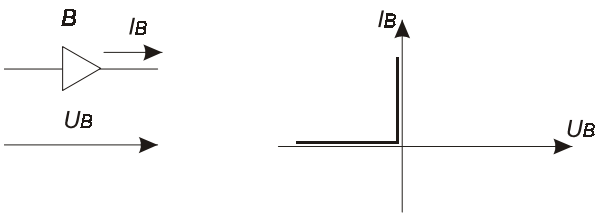


Рис. 1.27. Обозначение и ВАХ идеального вентиля

Идеальный ключ

Переключатель описывался сложной нелинейной ВАХ. Для упрощения описания его ВАХ кусочноаппроксимировали. Полученный элемент назвали идеальным ключом (рис. 1.28).

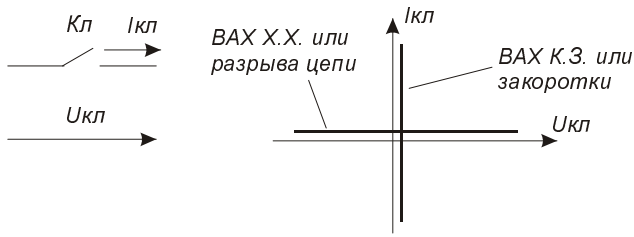


Рис. 1.28. Обозначение и ВАХ идеального ключа

Теоремы эквивалентных преобразований

Теорема размножения источников напряжения

При параллельном подключении к идеальному источнику напряжения нескольких ветвей возможно эквивалентное преобразование схемы путем размножения источников для каждой ветви и разрыва связи между ветвями (рис. 1.29).

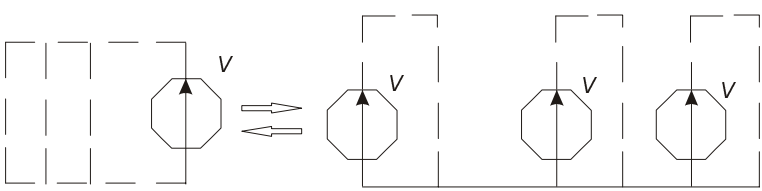


Рис. 1.29. Размножение источников напряжения

Теорема об эквивалентности двух схем замещения источников электрической энергии

Даны две схемы замещения реального источника электрической энергии (рис. 1.30): первая описывается последовательным соединением источника напряжения (V) и сопротивления (R), вторая — параллельным соединением источника тока (Ii) и сопротивления (R). Значения источников и сопротивления связаны зависимостью $Ii = V/R$.

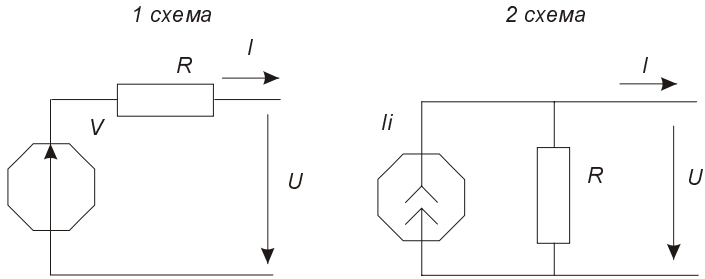


Рис. 1.30. Схемы замещения источников электрической энергии

Утверждается, что ВАХ 1 схемы = ВАХ 2 схемы при заданном подключении измерительных приборов.

Задание

Предлагается эквивалентно преобразовать сложную схему замещения, состоящую из резистивных элементов и идеальных источников, и определить показания амперметра и вольтметра, подключенных к одной из ветвей схемы.

Упражнение 1.2. Анализ транзисторных схем в квазистатике

Рассматриваются алгоритмы анализа схем с транзисторами, включенными с общими эмиттерами, базами и коллекторами, в квазистатическом режиме работы схем (то есть без учета факторов, влияющих на быстродействие).

О коэффициентах усиления транзисторных схем по току, по напряжению, по мощности (K_I, K_U, K_P)

Рассуждения о коэффициентах усиления правомерны при работе УК в активной области (то есть там, где справедлива формула $I_к = B * I_б$).

Включение транзистора с общим эмиттером (ОЭ)

$K_I = I_{вых} / I_{вх} = I_к / I_б = B > 1$.

$K_U = U_{вых} / U_{вх} = U_{кэ} / 0.7 => 1$.

$K_P = Ki * Ku >> 1$.

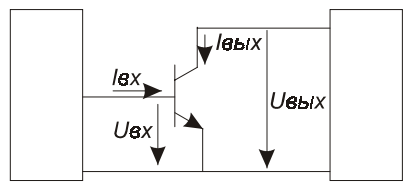


Рис. 1.31. Схема включения транзистора с общим эмиттером

Включение транзистора с общим коллектором (ОК)

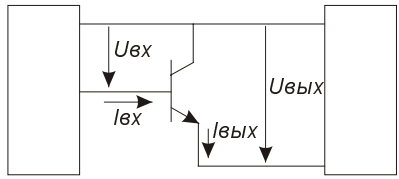


Рис. 1.32. Схема включения транзистора с общим коллектором

$K_I = I_{вых}/I_{вх} = I_э/I_б = (I_к + I_б)/I_б = B + 1 > 1$.

$K_U = U_{вых}/U_{вх} = U_{кэ}/U_{кб} = U_{кэ}/(U_{кэ} - U_{бэ}) \cong 1$.

$K_P = Ki * Ku \geq 1$ (не велико в сравнении с ОЭ).

Включение транзистора с общей базой (ОБ)

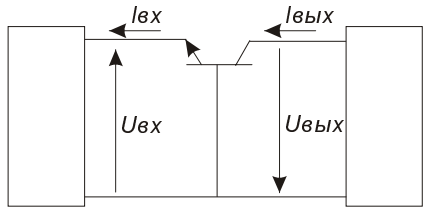


Рис. 1.33. Схема включения транзистора с общей базой

$K_I = I_{вых}/I_{вх} = I_к/I_э = I_к/(I_к + I_б) = \alpha < 1$.

$K_U = U_{вых}/U_{вх} = U_{кб}/U_{бэ} = (U_{кэ} - U_{бэ})/U_{бэ} \geq 1$.

$K_P = Ki * Ku \geq 1$, но мал и может быть < 1 при приближении к насыщению.

Формулы зависимостей между токами базы ($I_б$), коллектора ($I_к$) и эмиттера ($I_э$)

Эти формулы справедливы только для активной области работы инвертора и усилительного каскада.

I закон Кирхгофа: $I_б + I_к + I_э = 0$; $I_к = B * I_б$.

$I_б = I_к/B$; $I_к = B/(B+1) * I_э$; $I_к = \alpha * I_э$;

$\alpha = B/(B+1)$ — коэффициент передачи тока эмиттера (используется при включении с ОБ).

$I_э = I_б * (B+1)$.

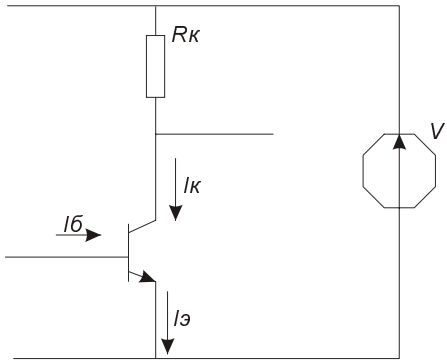


Рис. 1.34. Схема подключения амперметров для токов в транзисторе

Алгоритмы анализа транзисторных схем при включении транзисторов с ОЭ, ОБ и ОК

Общий эмиттер (ОЭ)

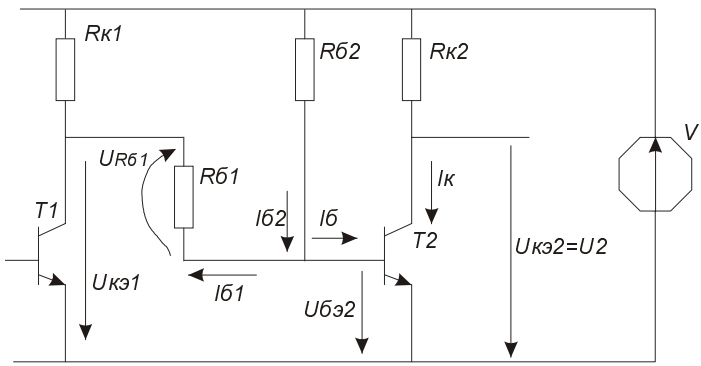


Рис. 1.35. Схема с транзисторами с общим эмиттером

Транзистор $T1$ находится в состоянии насыщения.

Найдите значения $U2$, соответствующие этому состоянию (для первого и второго вариантов данных).

ДАНО:

1 вариант

($T1$-насыщен; $U_{кэ1} = 0\ B$)

$V = 5B$

$B2 = 25$

$R_{Б1} = R_{Б2} = 33\ кОм$

$R_{к1} = R_{к2} = 1\ кОм$

2 вариант

($T1$-насыщен; $U_{кэ1} = 0\ B$)

$V = 5B$

$B2 = 25$

$R_{к1} = R_{к2} = 1\ кОм$

$R_{Б1} = 1\ кОм$

$R_{Б2} = 33\ кОм$

3 вариант

($T1$-насыщен; $U_{кэ1} = 0\ B$)

$V = 5B$

$B2 = 25$

$R_{к1} = R_{к2} = 1\ кОм$

$R_{Б2} = 10\ кОм$

НАЙТИ такое $R_{Б1}$, чтобы $T2$ был насыщен (для третьего варианта данных).

РЕШЕНИЕ:

1 вариант

Предположим, $T2$ открыт и ненасыщен ($U_{бэ2} = 0.7B$; $I_к = B * I_б$).

$$UR_{б1} + U_{кэ1} - U_{бэ2} = 0;\quad I_{б1} * R_{б1} = 0.7B;\quad I_{б1} = 0.7B/33кОм = 21мкА;$$

$$V - U_{бэ2} - I_{б2} * R_{б2} = 0;\quad 5 - 0.7 - I_{б2} * R_{б2} = 0;\quad I_{б2} = 4.3B/33кОм = 130мкА;$$

$$I_б + I_{б1} - I_{б2} = 0;\quad I_б = I_{б2} - I_{б1} = 109мкА;\quad I_{кн} = V/R_{к2} = 5мА;$$

$$I_к = B * I_б = 2.5мА < I_{кн},\ поэтому\ T2\ открыт\ и\ ненасыщен;$$

$$U_{кэ2} = U2 = V - I_к * R_{к2} = 2.5B.$$

2 вариант

Предположим, $T2$ открыт и ненасыщен ($U_{бэ2} = 0.7B$; $I_к = B * I_б$).

$$UR_{б1} + U_{кэ1} - U_{бэ2} = 0;\quad I_{б1} * R_{б1} = 0.7B;\quad I_{б1} = 0.7B/1кОм = 700мкА;$$

$$V - U_{бэ2} - I_{б2} * R_{б2} = 0;\quad 5 - 0.7 - I_{б2} * R_{б2} = 0;\quad I_{б2} = 4.3B/33кОм = 130мкА;$$

$$I_б + I_{б1} - I_{б2} = 0;\quad I_б = I_{б2} - I_{б1} = -570мкА;$$

Поэтому T2 — заперт и $U_{кэ2} = U2 = 5B$.

3 вариант

$T2$ насыщен, поэтому $I_к = I_{кн} = V/R_{к2} = 5мА$; $I_б = I_к/B = 0.2мА = 200мкА$.

$$V - U_{бэ2} - I_{б2} * R_{б2} = 0;\quad 5 - 0.7 - I_{б2} * R_{б2} = 0;\quad I_{б2} = 4.3B/10кОм = 430мкА;$$

$$I_б + I_{б1} - I_{б2} = 0;\quad I_{б1} = I_{б2} - I_б = 230мкА;\quad UR_{б1} + U_{кэ1} - U_{бэ2} = 0;$$

$$I_{б1} * R_{б1} = 0.7B;\quad R_{б1} = 0.7B/230мкА = 3.04кОм$$ и больше.

Общая база (ОБ)

ДАНО:

1. Схема.
2. Коэффициент B транзистора.

НАЙТИ все токи и напряжения в схеме.

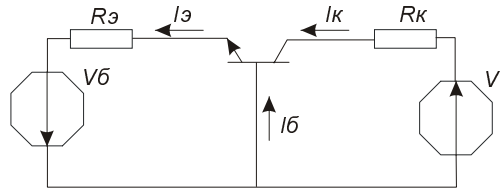


Рис. 1.36. Схема с транзисторами с общей базой

РЕШЕНИЕ:

Предположим, транзистор открыт и ненасыщен ($U_{бэ} = 0.7B$; $I_к = B * I_б$).

Тогда $I_{эр(реальное)} = (Vб - 0.7)/R_э$.

Если $I_{эр} < 0$, то предположение неверно и транзистор заперт.

Если нет, то определяем $I_{кн(насыщения)}$ и $I_{эн}$: при насыщении транзистора $U_{бэ} = 0.7B$, $U_{кэ} = 0$, поэтому $U_{бк} = 0.7B$, поэтому $I_{кн} = (V + U_{бк})/R_к = (V + 0.7)/R_к$, поэтому $I_{эн} = I_{кн}/\alpha = I_{кн} * (B+1)/B$.

После сравниваем $I_{эр}$ и $I_{эн}$:

1. Если $I_{эр} > I_{кн}$, то транзистор насыщен, поэтому $I_{кр} = I_{кн}$, $I_{бр} = I_{эр} - I_{кн}$, $I_э = I_{эр}$.
2. Если $I_{эр} < I_{эн}$, то транзистор открыт и ненасыщен, поэтому $I_{кр} = \alpha * I_{эр}$, $I_{бр} = I_{эр} - I_{кр}$, $I_э = I_{эр}$ и т. д.

Общий коллектор (ОК)

ДАНО:

1. Схема
2. Коэффициент "B" транзистора

НАЙТИ все токи и напряжения в схеме.

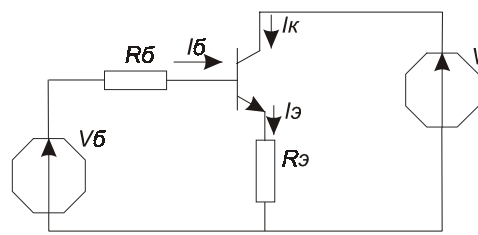


Рис. 1.37. Схема с транзисторами с общим коллектором

РЕШЕНИЕ:

Определим $I_{эн} = V/R_э$, здесь $U_{кэн} = 0B$.

Предположим, транзистор открыт и ненасыщен ($U_{бэ} = 0.7B$; $I_к = B * I_б$).

Тогда $I_э = I_б * (B+1)$.

Из 2 закона Кирхгофа: $V_б - I_э * R_э - U_{бэ} - I_б * R_б = 0$, подставляя первое во второе, имеем $V_б - I_б * (B+1) * R_э - 0.7 - I_б * R_б = 0$. Отсюда $I_б = (V_б - 0.7)/(R_э * (B+1) + R_б)$, определяем $I_э = I_б * (B+1)$.

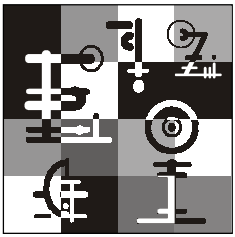
Если $I_б < 0$, то транзистор заперт.

Если $I_б > 0$, то:

- если $I_э < I_{эн}$, то транзистор открыт и ненасыщен, поэтому $I_э = I_б * (B+1)$ и $I_к = I_э - I_б$ ...;
- если $I_э > I_{эн}$, то транзистор насыщен, поэтому $I_э = I_{эн}$, $U_{кэ} = 0B$, а $I_б$ ищем из уравнения $V_б - I_э * R_э - U_{бэ} - I_б * R_б = 0$, подставляя $I_э = I_{эн}$. $I_к = I_э - I_б$ и так далее...

Задание

Предлагается, используя вышеизложенные алгоритмы, определить токи и напряжения в индивидуальной транзисторной схеме, при включении транзисторов с ОЭ, ОК и ОБ.



Глава 2

Простейшие схемы аналоговой техники, элементы цифровой техники

Перейдем к рассмотрению простейших схем аналоговой техники: схема усилительного каскада (УК) постоянного тока (УПТ) (рис. 2.1) и схема УК переменного тока (УПерТ) (рис. 2.2). Более подробно эта тематика описана в соответствующей лабораторной работе.

С этой главы начинается анализ схем элементов цифровой техники (ЦТ): режимы работы, факторы, влияющие на их быстродействие, и классификация элементов ЦТ.

2.1. Схемы аналоговой техники (УПТ и УПерТ)

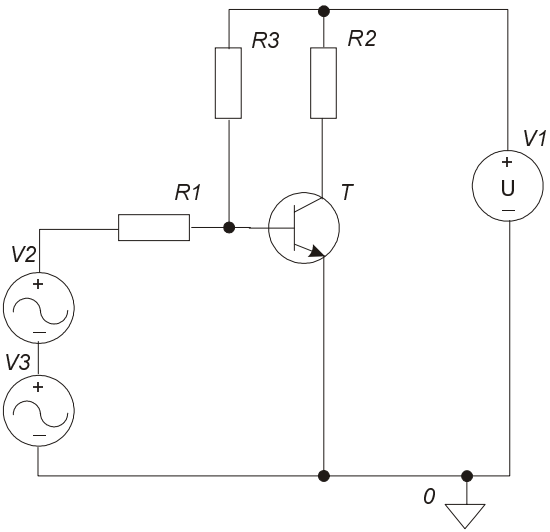


Рис. 2.1. Принципиальная схема усилителя постоянного тока

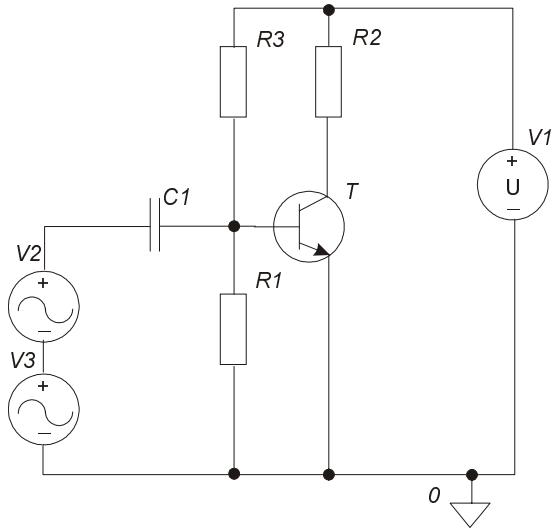


Рис. 2.2. Принципиальная схема усилителя переменного тока

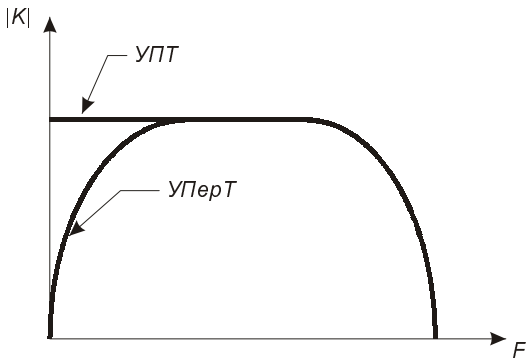


Рис. 2.3. Амплитудно-частотная характеристика УПТ и УПерТ

$|K| = \dfrac{|U_{вых}|}{|U_{вх}|}$ — коэффициент усиления по напряжению.

Завал по нижним частотам (рис. 2.3) объясняется наличием у УПерТ емкости $C1$.

Завал по верхним частотам и у УПТ и у УПерТ объясняется инерционными свойствами транзистора: чтобы его открыть, нужно внести заряд в базу, и наоборот.

2.2. Схемы элементов цифровой техники (ЦТ)

В дальнейшем будут рассматриваться устройства ЦТ малой интеграции (элементы), средней, большой и сверхбольшой интеграции.

2.2.1. Квазистатический и динамический режимы работы

Далее представлен анализ работы цепочки инверторов некоторого гипотетического устройства ЦТ в динамическом режиме.

Из временных диаграмм (рис. 2.4) видно, что сигнал по нижней цепочке (с инверторами) из-за задержек инверторов придет на нижний вход правого элемента тогда, когда сигнал на верхний вход уже изменит свое значение, хотя на входы устройства они пришли одновременно. Поэтому в работе устройства произойдет сбой.

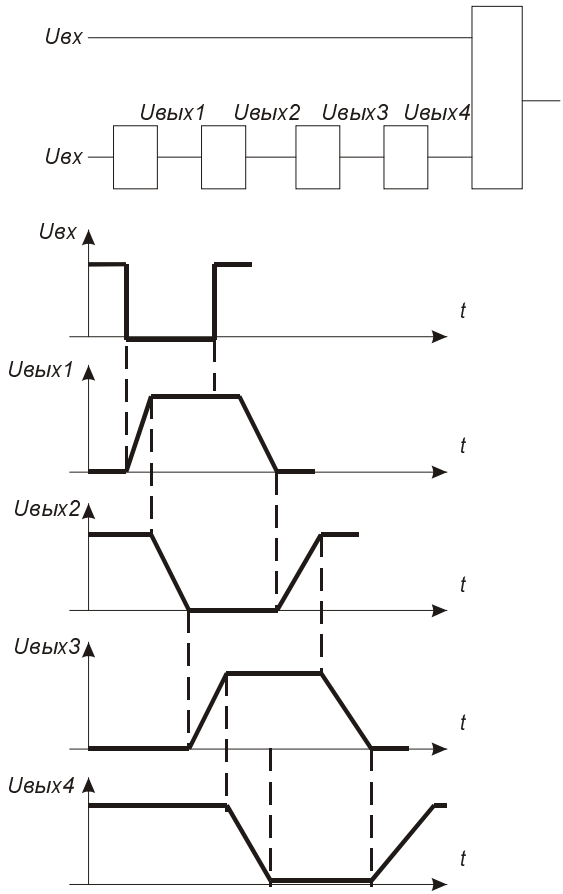


Рис. 2.4. Временные диаграммы цепочки инверторов

В динамическом режиме работы скорости изменения силы тока и напряжения столь велики, что при расчете схемы в этом режиме нельзя пренебрегать накоплением энергии и связанным с этим запаздыванием выходного сигнала относительно входного (этим динамический режим отличается от квазистатического режима).

Для анализа динамического режима работы электронной цепи надо составить такую ее схему замещения (или мысленную модель), в которой были бы отражены факторы, влияющие на быстродействие работы этой цепи или на переходные процессы в ней.

Факторы, влияющие на быстродействие работы элементов вычислительной техники

Одна из основных задач разработчиков элементной базы ВТ — увеличение быстродействия ВТ за счет уменьшения времени переключения элементов ВТ из диапазона "0" в диапазон "1" и обратно (рис. 2.5). Это время определяется временем переходного процесса в транзисторных цепях.

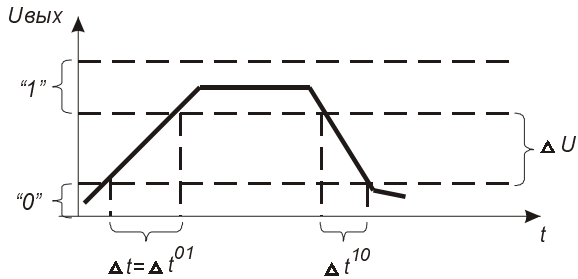


Рис. 2.5. Временная диаграмма

Для изменения состояния транзистора в его базу должен быть внесен или изъят заряд: $\Delta Q = C * \Delta U$, предположим для простоты, что заряд вносится постоянным током I. Тогда:

$$\Delta Q = \int I dt = I * \Delta t \Rightarrow \Delta t = \frac{C * \Delta U}{I}.$$

Полученная формула приближенная, так как и ток (I) и емкость (C) не являются неизменными, но оценить влияние различных факторов на быстродействие можно.

Для уменьшения времени Δt (наносекунды) надо:

1. Уменьшить расстояние ΔU между диапазонами напряжений, соответствующими "0" и "1" (от 100 до 1 В).
2. Уменьшить значение паразитных емкостей C (пикофарады) (значение паразитных емкостей связано с площадью расположения элементов в чипе, то есть с уровнем интеграции; значение C уменьшено на 4 порядка).
3. Увеличить переключающий ток I (миллиамперы) (увеличивать нецелесообразно, так как это ведет к увеличению выделяемой энергии и усложняется проблема теплоотвода).
4. Уменьшить глубину насыщения транзистора.

О необходимости введения открытого транзистора в насыщение в схемах ЛЭ с ОЭ

Известно, что параметр B транзистора нестабилен, следовательно, при постоянном токе базы пологий участок выходной ВАХ транзистора может располагаться на разной высоте.

$$Ic_{пар.вых.} = C_{пар.вых.} \frac{dUкэ}{dt}.$$

Из анализа ВАХ $Iк(Uкэ)$ (рис. 2.7) видно, что если транзистор не насыщен (находится на границе насыщенного и открытого состояний), то при наименьшем значении "B" напряжение на выходе $Uкэ = \Delta U1$ будет превышать нулевой диапазон, а значит не будет входить ни в нулевой, ни в единичный диапазоны. Это приведет к сбою в работе устройства ЦТ.

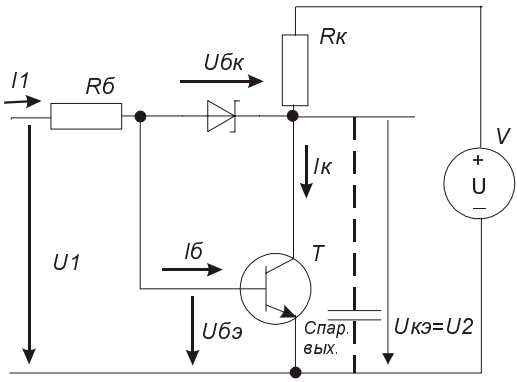


Рис. 2.6. Схема каскада с диодом Шотки

Таким образом, транзистор в схемах элементов ЦТ, включенный с общим эмиттером, должен быть насыщен ($Rк = Rк1$), но не глубоко.

Известны два способа устранения глубокого насыщения транзисторов в ЛЭ:

- первый основан на использовании включения транзистора с ОБ *(см. главу 4)*;
- второй основан на применении нелинейной отрицательной обратной связи через диод Шотки (рис. 2.6), включенный между базой и коллектором транзистора.

Идея второго способа — на вход ЛЭ подают большой ток $I1$, такой, чтобы даже транзистор с наименьшим значением B был насыщен. Обратная связь работает так, что при приближении транзистора к области насыщения лишний входной ток $I1$ отводится через диод Шотки и коллектор, не давая транзистору войти в глубокое насыщение.

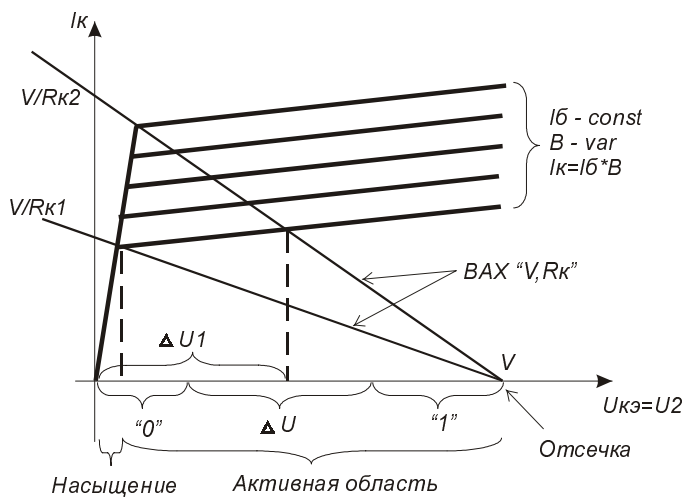


Рис. 2.7. Вольт-амперная характеристика $Iк(Uкэ)$

Вывод: у ЛЭ с диодом Шотки самый выгодный режим работы, так как отпирающий ток $I1$ велик, следовательно, заряд вносится в базу с большой скоростью, следовательно, $Iкн$ достигается относительно быстро.

Затем $Iб$ уменьшается до $Iбб$, поэтому избыточный заряд в базу не вносится.

2.2.2. Классификация элементов цифровой техники (ЦТ)

Элемент ЦТ — электронная схема, выполняющая определенную функцию: логическую, хранения информации, вспомогательную или специальную.

В настоящее время элементы ЦТ выпускаются сериями: микропроцессорные системы (МПС), микроконтроллерные наборы.

Серия элементов ЦТ называется полной, если она обладает функциональной и технической полнотой.

Функциональная полнота — свойство серии выполнять любую булеву (переключательную, логическую) функцию. Техническая полнота — свойство серии реализовать как логическую, так и другие функции: хранения, вспомогательные, специальные.

Логическая переменная (ЛП) — переменная, имеющая два или более состояний (0, 1, Z, " — ", ...), определяемых словарем логической переменной.

Классификация элементов ЦТ.

❒ По функциональному назначению.

- *Логические элементы* — предназначены для логического преобразования информации, то есть для реализации логических функций (операций):
  - ◊ операцию логического умножения, конъюнкции, реализует *конъюнктор*, элемент "И" (рис. 2.8), схема совпадения. Символичная запись: $Y = X1 \wedge X2 \wedge \ldots = X1 * X2 \ldots$

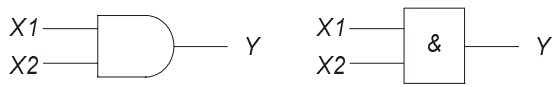


Рис. 2.8. Условное графическое обозначение (УГО) конъюнктора

  - ◊ операцию логического сложения, дизъюнкции, реализует *дизъюнктор*, элемент "ИЛИ" (рис. 2.9), схема собирания. Символичная запись: $Y = X1 \vee X2 \vee \ldots = X1 + X2 + \ldots$

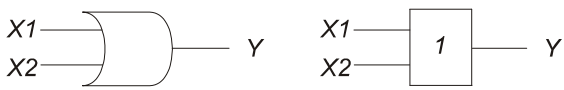


Рис. 2.9. УГО дизъюнктора

◊ операцию логического отрицания, инверсии, реализует *инвертор*, элемент "НЕ" (рис. 2.10). Символичная запись: $Y = X1!$;

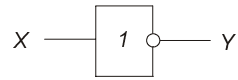


Рис. 2.10. УГО инвертора

◊ универсальные функции: операцию "И-НЕ" реализует элемент "И-НЕ" или элемент Шеффера $Y = (X1 * X2)!$; операцию "ИЛИ-НЕ" — элемент "ИЛИ-НЕ" или элемент Пирса $Y = (X1 + X2)!$. Универсальные элементы обладают функциональной полнотой, так как, используя каждый из них и правило Де'Моргана, можно реализовать любую логическую функцию. Эти элементы являются логическими базовыми элементами. На их основе реализованы процессоры, каналы, накопители и так далее.

Для логических преобразований используются правила Де'Моргана: $(X1 + X2)! = X1! * X2!$ $\qquad$ $(X1 * X2)! = X1! + X2!$;

◊ несколько функций реализуют *функциональные элементы*, например, одновременная реализация одним элементом функций "И", "ИЛИ" "И-НЕ", "ИЛИ-НЕ";

◊ желаемые функции реализуют *адаптивные или программируемые элементы* (программируемые логические интегральные схемы — ПЛИС).

- *Элементы хранения информации* (элементы памяти) предназначены для запоминания и временного хранения двоичной информации (триггеры, регистры, буферы-защелки и т. д.).
- *Вспомогательные элементы* предназначены для формирования, задержки, генерирования и т. п. электрических сигналов в схемах.
- *Специальные элементы* — предназначены для физического преобразования электрических сигналов (индикаторы и пр.).

❒ По типу связей элементы ЦУ делятся на: *элементы с потенциальной связью* (по постоянному току) соединяются непосредственно или через резистор, диод, транзистор, здесь значение ЛП задается ее словарем; *элементы с импульсной связью* соединяются через конденсаторы.

❒ По типу логики элементы делятся: для биполярных транзисторов на *элементы резистивно-транзисторной логики*, *диодно-транзисторной логики*, *транзисторно-транзисторной логики (ТТЛ)*, *эмиттерносвязанной логики (ЭСЛ)*, *интегральноинжекционной логики (ИИЛ)*; для МДП-транзисторов — *МДП-логика*: статическая на n-МОП-транзисторах и на К-МОП-схемах, и динамическая.

❒ По полярности логики (полярность связана с изображением 0 и 1 соответствующими диапазонами напряжений), если на оси значений напряжений значе-

ния нулевого диапазона находятся левее значений единичного диапазона, то это *"положительная логика"*, если правее — "отрицательная логика".

- По технологии изготовления могут быть: *полупроводниковые элементы* (все компоненты и межкомпонентные соединения выполнены в объеме и на поверхности полупроводника); *пленочные* (все выполнено в виде пленок на поверхности); *гибридные* (имеют место и полупроводниковые и пленочные элементы).
- По конструктивному оформлению элементы делятся на *корпусные* и *бескорпусные*.
- По способу питания: элементы с *потенциальным* и *импульсным питанием*.
- По электрическим и эксплуатационным параметрам характеризуются: быстродействием, определяемым временем задержки распространения сигнала (меньше 5 нс — *сверхбыстродействующие*, от 5 до 10 нс — *высокое быстродействие*, от 10 до 50 нс — *среднее*, больше 50 нс — *низкое*); мощностью рассеивания (большая, средняя, малая); эксплуатационными параметрами (надежность, устойчивость к механическим, климатическим, температурным воздействиям).
- По экономическим параметрам характеризуются стоимостью.
- По степени интеграции элементов. Определяется коэффициентами функциональной $Кф$ и компонентной $Кк$ интеграции: $Кф = \lg Nэ$, где $Nэ$ — количество элементов "И-НЕ" либо "ИЛИ-НЕ", расположенных на кристалле.
  - $Кф \le 1$ — *малая интегральная схема (ИС)* (триггеры и пр.).
  - $Кф \le 2$ — *средняя ИС (СИС)* (счетчики, регистры и пр.).
  - $Кф \le 3$ — *большая ИС (БИС)* (АЛУ, ОЗУ и пр.).
  - $Кф > 3$ — *сверхбольшая ИС (СБИС)* (микроконтроллеры, таймеры и пр.).

  $Кк = \lg Nк$, где $Nк$ — общее число транзисторов, расположенных на кристалле (чипе). Для оценки сложности ИС вводят параметр "плотность упаковки" $\gamma = \dfrac{Nк}{V}$.

2.2.3. Языки описания логических элементов

На примере элемента "И-НЕ" (ТТЛ) покажем описание логических элементов.

- Естественный (словесный) язык: функция ложна в том и только том случае, когда истинны оба аргумента, и истинна во всех остальных случаях.
- Символическая запись (логическая формула): $Y = \overline{(X1 * X2)}$.
- Логическая схема показана на рис. 2.8—2.10.

- Временные диаграммы: а) без учета задержки распространения сигнала; б) с учетом задержки распространения (рис. 2.11).
- Электрическая принципиальная схема приведена на рис. 3.1.

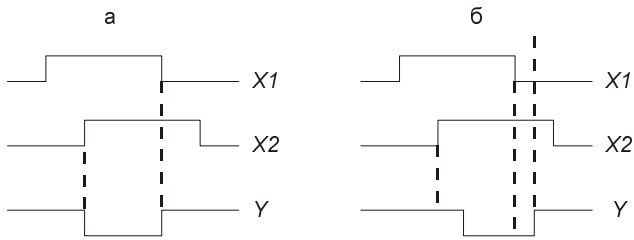


Рис. 2.11. Временные диаграммы

Лабораторная работа. Исследование инвертора, усилителей постоянного и переменного токов на биполярном транзисторе. Работа с системой Design Center

Настоящие методические указания помогут читателям при выполнении лабораторных работ, объектом изучения которых являются электронные логические элементы (ЛЭ).

Термин "анализ" при изучении различных объектов означает определение связи между входными и выходными сигналами при заданной структуре и параметрах объекта. Под "синтезом" (или проектированием) понимают построение структуры и (или) нахождение параметров объекта, обеспечивающих необходимый или оптимальный по каким-либо признакам характер этой связи. Такие варианты синтеза называют соответственно структурным или параметрическим синтезом.

Анализ и синтез могут проводиться при изучении двух возможных режимов работы объектов: квазистатического и динамического. В дальнейшем для простоты квазистатический режим будем именовать статическим.

Динамические режимы в свою очередь делятся на два типа:

- изучение реакции устройства на импульсное входное воздействие;
- изучение реакции устройства на гармоническое входное воздействие.

Реакцию объекта на первый тип воздействия обычно называют переходным процессом.

Инструментом исследования в этих лабораторных работах служит система схемотехнического моделирования и проектирования печатных плат Design Center. Используется только та часть системы, которая предназначена для моделирования.

Под исследованием ЛЭ понимается их анализ, т. е. получение характеристик вход-выход (ХВВ) в статическом и динамическом режимах.

Из числа имеющихся в Design Center программ используются следующие:

- Schematics — графический редактор принципиальных схем, который одновременно является управляющей оболочкой для запуска основных модулей системы на всех стадиях работы;
- PSpice — моделирование аналоговых устройств;
- PLogic — моделирование цифровых устройств;
- StmEd — редактор входных сигналов;
- Probe — графическое отображение результатов моделирования.

В графическом редакторе схему создают на основе имеющихся в библиотеках графических изображений элементов. Каждому графическому элементу соответствует определенная математическая модель, параметры которой могут быть изменены. Модели компонентов созданы на основе математических выражений, описывающих их работу. Как в любых моделях в них учтены не все (что невозможно), а основные особенности работы компонентов. Справедливость (адекватность) модели может быть проверена только сравнением с экспериментом, при этом всегда существуют ограничения, в рамках которых модель дает результаты, совпадающие с экспериментом в пределах требуемой точности. На основе моделей компонентов может быть промоделирована работа различных аналоговых, цифровых и цифроаналоговых устройств.

В лабораторных работах используются в основном следующие компоненты:

- Резисторы (RN);
- Конденсаторы (CN);
- Диоды (DN);
- Биполярные (QN) и МОП-транзисторы (TN);
- Источники питания (тока (IN) или напряжения (VN));
- Цифровые устройства.

Примечание

Здесь N — порядковый номер элемента данного типа, формируемый автоматически.

Выполнение лабораторных работ состоит из следующих этапов:

1. Ввод принципиальной схемы исследуемого ЛЭ с помощью графического редактора.
2. Моделирование работы схемы, т. е. подача различных входных воздействий и фиксация результатов в виде временных диаграмм в выбранных точках схемы.
3. Исследование работы объекта, т. е. получение выходных данных при варьировании входных воздействий и (или) параметров компонентов.

В отчете по работе необходимо отразить схему исследуемого объекта, временные диаграммы, полученные в ходе работы, и характеристики, полученные на их основе.

Работа с системой Design Center

1. Создание нового проекта или открытие существующего.
   - Запустить Design Center (**Start** | **Programs** | **DesignLab Eval 8** | **Design Manager** (Старт | Программы | DesignLab Eval 8 | Design Manager)).
   - Установить имя проекта, выбрав команду меню **File** | **New Workspace** (Файл | Новая рабочая область). В открывшемся окне в верхней строке ввести имя создаваемого проекта, а в нижней — путь к нему (желательно создавать свои проекты в папке Projects) и нажать кнопку **Create**. В результате будет создана новая папка с именем рабочего проекта для хранения всех его файлов.
   - Для открытия уже существующего проекта выбрать команду меню **File** | **Open Workspace** (Файл | Открыть рабочую область). Указать полный путь к проекту и нажать кнопку **Open**.
   - Для создания нового файла схемы разрабатываемого устройства щелкнуть на значке **Run Schematics** на панели инструментов или использовать команду меню **Tools** | **Schematics** (Инструменты | Схематическое представление). (Для открытия существующего файла дважды щелкнуть по имени файла схемы с расширением sch.)
   - В окне графического редактора отображается рабочее поле схемы с нанесенной на нем координационной сеткой, конфигурацию которой можно настроить в меню **Options** | **Display Options** (Опции | Опции дисплея).
   - Сохранить файл с нужным именем (**File** | **Save as** (Файл | Сохранить как)).
2. Ввод функциональной схемы с помощью графического редактора.
   - Для ввода элементов схемы выбрать команду меню **Draw** | **Get New Part** (Чертить | Получить новую часть). В диалоговом окне в полном списке элементов **Full List** выбрать нужный (ориентируясь на его изображение)

и нажать кнопку **Place** (Поместить). В результате появится изображение компонента, "привязанное" к курсору. Нажатие левой кнопки мыши фиксирует расположение компонента, после чего его можно поместить еще и в другом месте. Нажатие правой кнопки завершает процесс размещения этого элемента. Повторить процедуру для всех элементов схемы.

- Перемещение элемента по полю проекта выполняется в режиме Drag & Drop.
- Чтобы получить копию любого элемента схемы, надо использовать операции **Cut** (Вырезать), **Copy** (Копировать), **Paste** (Вставить) буфера приложений Windows. В процессе выбора местоположения элемента на схеме его можно повернуть **Edit** | **Rotate** (Редактировать | Вращать) или получить зеркальное изображение **Edit** | **Flip** (Редактировать | Отобразить зеркально).
- Возможна установка режима "резиновой нити" **Options** | **Display Options** | **Rubberband** (Опции | Опции дисплея | Резиновая лента).
- Для соединения элементов установить режим **Draw** | **Wire** (Чертить | Провод). После выбора этой команды изображение курсора принимает форму карандаша. Щелчок левой кнопкой фиксирует начало проводника, а повторное нажатие — конец проводника. При перемещении курсора прокладывается проводник. Для задания шины нужно выбрать **Draw** | **Bus** (Чертить | Шина). Шины отображаются жирной линией.
- Для удаления соединения или элемента надо выделить его и нажать клавишу **Delete** (Удалить).
- Для правильного функционирования устройства необходимо наличие узла "земли" (компонент **GND_ANALOG** в библиотеке элементов схемы).
- Компоненты характеризуются набором параметров — атрибутов. Выделить редактируемый элемент и установить необходимые атрибуты можно, используя команду меню **Edit** | **Attribute** (Редактировать | Атрибуты). Открывается диалоговое окно со списком всех параметров элемента. Значком звездочка (*) помечены нередактируемые параметры. После изменения значения атрибута следует нажать кнопку **Save Attr** (Сохранить атрибут).
- Узлы, в которых нас интересуют результаты моделирования, должны быть поименованы. Двойной щелчок по изображению проводника позволяет задать ему новое имя. Кроме того, в этих узлах должны быть расставлены маркеры для построения графиков напряжений (пиктограмма **Voltage** | **Level Marker** (Напряжение | Маркер уровня) на панели инструментов) или токов (пиктограмма **Current** | **Level Marker** (Электрический ток | Маркер уровня) на панели инструментов).
- Завершив построение схемы, сохранить ее (**File** | **Save** (Файл | Сохранить)).

3. Моделирование работы схемы.
   - Для задания директив моделирования выполнить команду меню **Analysis** | **Setup** (Анализ | Установка). Для расчета переходных процессов в схеме отметить раздел **Transient** (Переходный процесс). Установить параметры моделирования можно, нажав кнопку **Transient**: шаг моделирования (**Print Step**) и время моделирования (**Final Time**).
   - Для вывода амплитудно-частотной характеристики установите в меню моделирования **Analysis** | **Setup** (Анализ | Установка) режим **AC Sweep**.
   - Процесс моделирования запускается командой **Analysis** | **Simulate** (Анализ | Моделирование).
   - Результатом моделирования будут временные диаграммы во всех тех узлах схемы, на которые установлены маркеры. Все диаграммы располагаются на одном графике. Результаты записываются в файл с расширением dat.
   - Для вывода на экран временных диаграмм на отдельные графики необходимо в схеме установить только один маркер (например, на выходе) и запустить процесс моделирования. Затем выполнить команду меню **Plot** | **Add** (График | Добавить) и с помощью команды **Trace** | **Add** (Проследить | Добавить) выбрать из списка нужные зависимости для конкретного узла схемы.
   - Определение числовых характеристик в различных точках графика выполняется с использованием режима **Tools** | **Cursor** | **Display** (Инструменты | Курсор | Экран). Если на экране несколько графиков, то надо отметить нужный, а затем включить указанный режим и подвести курсор к интересующей точке графика, нажать левую кнопку мыши.
   - В случае неудачного завершения моделирования список обнаруженных ошибок и сообщений можно посмотреть в окне сообщений.

Программа работы

1. Создайте проект инвертора согласно рис. 2.12.

   Используемые элементы схемы: R — резистор, Q_KT316D — транзистор, $VPULSE$ — импульсный источник, $VSRC$ — источник постоянного напряжения, GND_ANALOG — "земля".

2. Установите следующие параметры элементов:
   - для резистора $R1$ — параметр $VALUE = 5\ кОм$;
   - для резистора $R2$ — параметр $VALUE = 320\ Ом$;
   - для источника постоянного напряжения $V2$ — параметр $DC = 5\ В$;

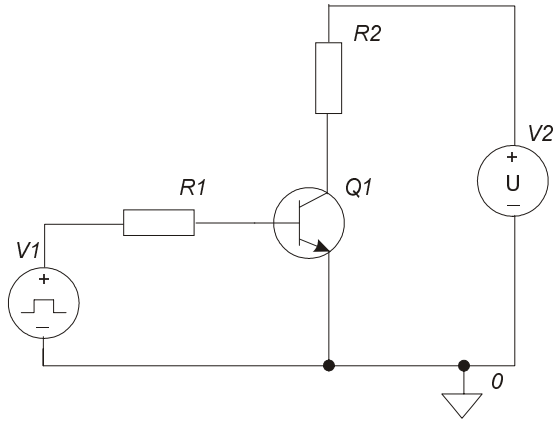


Рис. 2.12. Схема инвертора

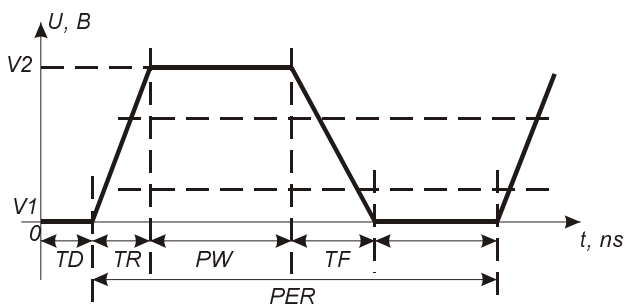


Рис. 2.13. Параметры импульса

- для генератора импульсов $V1$, который формирует импульсы произвольной формы, необходимо задать нужный импульс (треугольный, прямоугольный, нестандартный). Параметры импульса представлены на рис. 2.13. Для задания прямоугольного импульса установите следующие параметры для генератора $V1$:
  - ◊ $DC = 0$ — постоянная составляющая напряжения;
  - ◊ $AC = 0$ — амплитуда напряжения при анализе в частотной области;
  - ◊ $V1 = 0\ B$ — минимальное напряжение;
  - ◊ $V2 = 5\ B$ — максимальное напряжение;
  - ◊ $TD = 0$ — задержка;
  - ◊ $TR = 0.01\,ns$ — длительность переднего фронта;
  - ◊ $TF = 0.01\,ns$ — длительность заднего фронта;

◊ $PW = 300\,ns$ — длительность импульса;

◊ $PER = 600\,ns$ — период повторения.

Примечание

В системе Design Center применяются только английские эквиваленты единиц измерения. Например: ns ($нс$).

3. Исследование работы схемы.
   - Определите, где входное и выходное напряжение в схеме инвертора. Подключите маркеры напряжения на вход и выход для последующего сравнения диаграмм.
   - Промоделируйте работу схемы, зарисуйте временные диаграммы (время моделирования — $800\,ns$, шаг — $20\,ns$).
   - Измерьте время переключения выходного сигнала из "0" в "1" и обратно.
   - Временем переключения считается время от поступления команды на переключение в "0" или "1" (входной сигнал) до выполнения команды — достижения выходным сигналом диапазона "0" или "1".
   - Параметр $V2$ генератора импульсов установите равным $2.5\,B$. Повторите моделирование при различных значениях резисторов $R1$ и $R2$: $R1 = R1 \pm 50\%$; $R2 = R2 \pm 50\%$. Составьте таблицу зависимости быстродействия устройства от значений резисторов.
   - Попробуйте провести теоретический анализ влияния значений резисторов на работу устройства и на его быстродействие.
4. Получение квазистатической передаточной характеристики инвертора.
   - Промоделируйте работу схемы, установив режим моделирования **DC SWEEP**. Установите имя входного источника напряжения (на рис. 2.12 — $V2$) и задайте для него значения от 0 до $5\,B$ с шагом $0.1\,B$. Маркер напряжения должен быть установлен на выходе.
   - Получите диаграмму квазистатической передаточной характеристики инвертора. Зафиксируйте значения напряжений. В отчете укажите три диапазона состояния транзистора.
5. Рассчитайте выходную мощность для $U_{вых} \approx \frac{U}{2}$. Для этого установите параметр DC источника постоянного напряжения таким, чтобы на выходе схемы было напряжение $\frac{U}{2}$. Затем с помощью кнопки I на панели инструментов зафиксируйте значение тока в коллекторной цепи транзистора $P = U_{вых} * I_к$.

6. Для получения схемы простейшего усилителя постоянного тока измените схему инвертора, добавив резистор $R3$ для смещения передаточной характеристики (рис. 2.14).
7. Рассчитайте значение $R3$ для получения $U_{вых} \approx \frac{U}{2}$ при $U_{вх} = 0$:

$$R3 = \frac{u3}{i3}; \quad u3 = U - u0; \quad i3 = i0 - i1;$$

$$i1 = -\frac{u0}{R1}; \quad i0 = \frac{i2}{B};$$

$$i2 = \frac{U}{2R2}; \quad B = 126.$$

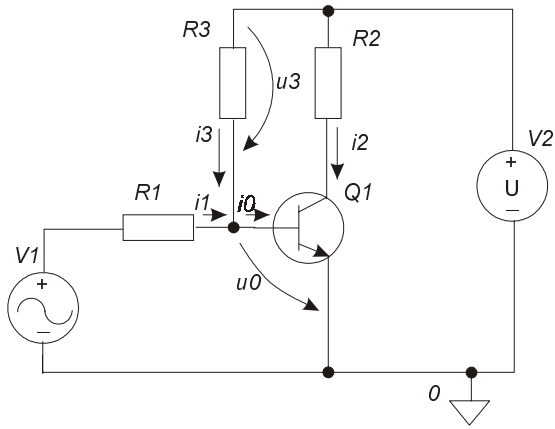


Рис. 2.14. Схема усилителя постоянного тока

8. Построение передаточной характеристики усилителя.
   - Установите расчетное значение $R3$.
   - Выполните п. 4 для усилителя (диапазон входного напряжения: $-2.5B \div 2.5B$).
9. Получение реакции усилителя на двухполярный сигнал.
   - Установите на входе источник синусоидального напряжения $VSIN$ с параметрами: $DC = 0$; $AC = 0.7\ B$; $VAMPL = 0.1\ B$; $VOFF = 0$; $FREQ = 20\ kHz$.
   - Промоделируйте схему, установив **Final Time** равным **200us** (мкс), шаг — **200ns**.
   - Зарисуйте диаграммы.
10. Получение амплитудно-частотной характеристики (АЧХ) усилителя (режим **AC SWEEP**).
    - Установите следующие параметры моделирования: характер частотной шкалы на диаграмме — **Octave**, начальная частота — **1Hz**, конечная частота — **1GHz**, количество рассчитываемых точек в октаве — **20**. Промоделируйте схему.
    - Определите максимальную частоту нормальной работы усилителя $F_{\max}$. Для этого необходимо измерить максимальную амплитуду $A_{\max}$ по АЧХ, вычислить величину, равную $0.707 * A_{\max}$, и определить соответствующую ей $F_{\max}$.
11. Исследование **реакции** усилителя на шум.
    - Добавьте в схему дополнительный генератор синусоидальных импульсов низкой частоты (источник шума) (рис. 2.15).

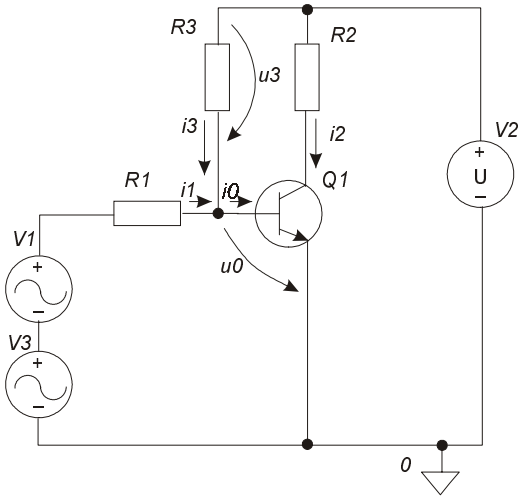


Рис. 2.15. Схема усилителя с источником шума на входе

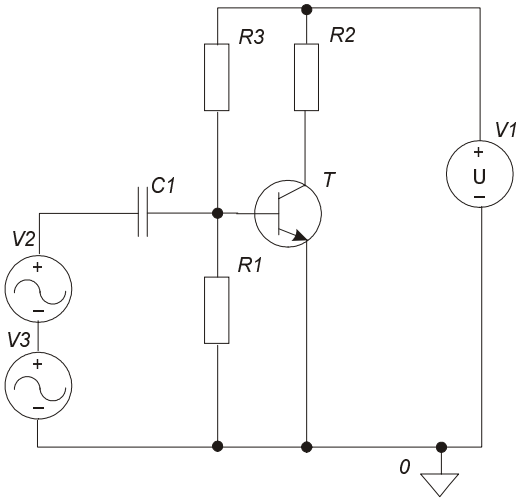


Рис. 2.16. Схема усилителя переменного тока

- Установите атрибуты источника шума $V3$ — $DC = 0$; $AC = 0.1\,B$; $VAMPL = 0.1\,B$; $VOFF = 0$; $FREQ = 50\,kHz$. Получите и зарисуйте временные диаграммы (время моделирования — **30ms**, шаг — **200ns**).

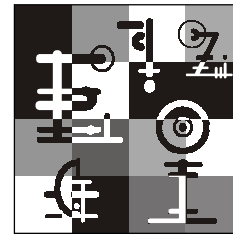
12. Измените схему в соответствии с рис. 2.16, предварительно рассчитав значение емкости, используя формулу $R_C = \dfrac{1}{2\pi fC}$, где f — частота пропускания. Установите значение емкости $C1$ из диапазона:

$$\frac{1}{2\pi f_{проп} R_C} < C1 < \frac{1}{2\pi f_{шума} R_C}; \quad R_C = R1.$$

13. Промоделируйте работу схемы, получите временные диаграммы, используя старые установки. Объясните наблюдаемый эффект. Получите амплитудно-частотную характеристику усилителя.

- Определите полосу пропускания усилителя. Для этого найдите максимальную амплитуду $A_{\max}$ по графику АЧХ. Найдите величину, равную $0.707 * A_{\max}$, и определите ω_H и ω_B.
- Полоса пропускания усилителя рассчитывается как: $\Delta\omega = \omega_B - \omega_H$.

14. В отчет включите все произведенные вычисления и полученные диаграммы с пояснениями к ним.



Глава 3

Логики серий микросхем. Транзисторно-транзисторная логика (ТТЛ)

Перечисляются основные логики (классы) серий микросхем и подробно рассматривается базовая схема серий ТТЛ.

Базовые схемы серий микросхем различных логик — это схемы, выполняющие одну из двух логических операций: либо "И-НЕ", либо "ИЛИ-НЕ", и отличающиеся друг от друга конструктивно, а также типами используемых транзисторов.

3.1. Логики серий микросхем

Логика (класс) определяется базовой схемой, простейшей схемой данной логики, реализующей либо операцию "И-НЕ", либо "ИЛИ-НЕ". Серии микросхем одной логики, имея общую базовую схему, отличаются конструктивно, назначением, например: малой и средней мощности, с открытым и без открытого коллектора, с диодом и без диода Шотки.

Перечислим основные логики (классы) серий микросхем.

- Транзисторно-транзисторная логика (ТТЛ) — на сериях этой логики были реализованы ЭВМ типа СМ-4, СМ-1420, СМ-1600, ЕС-1020 и пр.
- Эмиттерно-связанная логика (ЭСЛ) — на сериях этой логики были реализованы ЭВМ типа БЭСМ-6, ЕС-1045, ЕС-1050, ЕС-1060 и пр.
- Логика на n-МОП-схемах — на сериях этой логики были реализованы процессоры Intel 186, Intel 286.
- Логика на К-МОП-схемах — на сериях этой логики были реализованы процессоры Intel 386, Intel 486, Pent. II, III, IV и пр.
- Логика на схемах на арсениде галлия — на сериях этой логики был реализован α -процессор.

3.2. Транзисторно-транзисторная логика

Вид базовой схемы ТТЛ представлен на рис. 3.1.

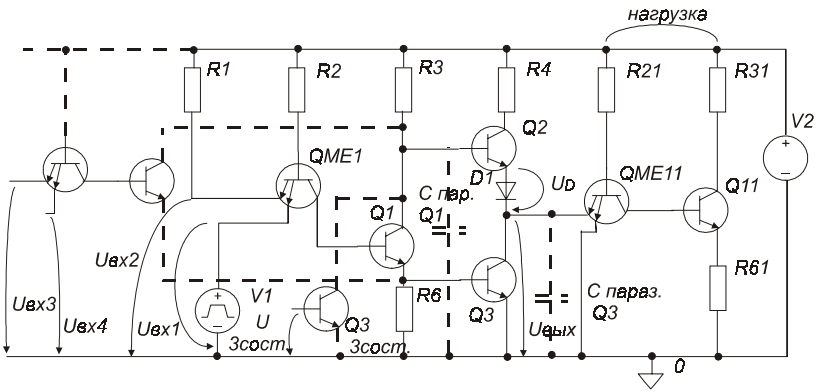


Рис. 3.1. Базовая схема ТТЛ

Словарь логической переменной (ЛП) для ТТЛ:

- **"0"** — значение логической переменной ТТЛ лежит в диапазоне от 0 до $0.4\ B$;
- **"1"** — значение логической переменной ТТЛ лежит в диапазоне от 2.4 до $5.0\ B$.

Если рассматривать схему, изображенную сплошной линией, то этой схеме будет соответствовать логическая операция (ЛО) "И-НЕ" ($y = \overline{x_1 x_2}$).

Если рассматривать схему, изображенную сплошной и штриховой линиями, то этой схеме будет соответствовать логическая операция "И-ИЛИ-НЕ" ($y = \overline{x_1 x_2 + x_3 x_4}$). Такая базовая схема определяет двухступенчатую ТТ-логику.

Докажем, что схема, изображенная сплошной линией, выполняет логическую операцию "И-НЕ".

Для этого рассмотрим все возможные комбинации сигналов на входах, состояния транзисторов при каждой комбинации и соответствующие значения сигнала на выходе:

1. Пусть хотя бы на одном входе — "0" или (закоротка).

   $Q1$ — заперт, $Q3$ — заперт (нет базового тока), $Q2$ — насыщен (ток $I_{вх}Q2 = IR3 = IББ2$ течет: "+", $R3$, Б-Э $Q2$, D, Э-К $QME11H$, Б-Э $Q11H$, $R61H$, "–"; ток $I_{вых}Q2$ течет: "+", $R4$, К-Э $Q2$, D и т. д.) $\Rightarrow$ на выходе имеем "1" ($2.4 \ldots 5B$).

2. Пусть на всех входах "1".

   Весь ток $IБQME1$ через коллектор $QME1$ течет в базу $Q1$, $Q1$ — насыщен $\Rightarrow$ $Q3$ — насыщен, а $Q2$ должен быть заперт (необходимо для увеличения быстродействия). Для этого подключается диод D, если бы D не было, то так как $Q1$ и $Q3$ — насыщены $UКЭQ1 = 0$, $UБКQ3 = 0.7B \Rightarrow UБЭQ2 = 0.7B$, и $Q2$ был бы открыт. Диод делит эти $0.7B$ пополам $\Rightarrow$ имеем $UБЭQ2 = 0.35B$ и $UD = 0.35B$. При этом переход $КБQ2$ — закрыт и $Q2$ — заперт $\Rightarrow$ на выходе имеем "0" ($0 \ldots 0.4B$).

   О токах $Q3$:

   $I_{вх}Q3 = IБQ3$ ("+", $R3$, $КЭQ1$, $БЭQ3$, "–"). Сопротивление $R6$ делит $IЭQ1$, забирая часть этого тока, так как для $IБQ3$ тока $IЭQ1$ слишком много, иначе транзистор $Q3$ будет работать не в режиме и скоро выйдет из строя.

   $I_{вых}Q3 = IКQ3$ ("+", $R21H$, $БЭQME11H$, $КЭQ3$, "–").

Это рассмотрение позволяет описать таблицей истинности функционирование базовой схемы ЛЭ ТТЛ (сплошная линия) (табл. 3.1).

Таблица 3.1. *Таблица истинности базовой схемы ЛЭ ТТЛ*

$U_{вх}1(X1)$	$U_{вх}2(X2)$	$U_{вых}(Y)$
0	0	1
0	1	1
1	0	1
1	1	0

Из таблицы следует, что базовый ЛЭ ТТЛ выполняет операцию "И-НЕ": $Y = (X1 * X2)!$.

Для дальнейшего понимания работы базовой схемы ЛЭ ТТЛ в квазистатическом режиме рассмотрим его характеристику передачи напряжения (ХПН) (рис. 3.2) или зависимость $U_{вых}(U_{вх}1)$ при подаче на $U_{вх}2$ напряжения из "1" диапазона (на схеме это подключение к "+" $V2$ через сопротивление $R1$).

Опишем, что происходит в схеме на каждом участке ХПН при постепенном увеличении $U_{вх}1$ от $0B$ и выше:

1. $U_{вх}1$ меняется от 0 до $0.4 \ldots 0.6B$.

Большая часть тока $IБQME1$ течет в эмиттер и Вход 1, меньшая начинает оттекать в коллектор $QME1$ и базу-эмиттер $Q1$, но транзистор $Q1$ еще заперт (переход К-Б закрыт, так как $IБQ1$ мал).

Таким образом, имеем: $Q2$ насыщен, $Q3$ заперт (т. к. $IБQ3$ мал).

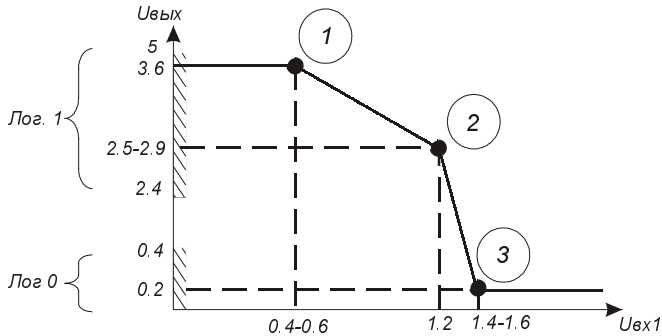


Рис. 3.2. ХПН базовой схемы ЛЭ ТТЛ (на $U_{вх}2$ — 0)

2. $U_{вх}1$ меняется от $0.4\ldots0.6B$ до $1.2B$.

   Транзистор $Q1$ начинает открываться $\Rightarrow IKQ1>0 \Rightarrow IБQ2$ уменьшается $\Rightarrow IKQ2$ и $IЭQ2$ уменьшаются, а транзистор $Q3$ еще не открылся ($IБQ3$ мал и К-Б $Q3$ закрыт).

3. $U_{вх}1$ меняется от $1.2B$ до $1.4\ldots1.6B$.

   Продолжает открываться $Q1$, начинает открываться $Q3$ и запираться $Q2$, здесь $Q2$ и $Q3$ открыты одновременно, они закорачивают источник. Чтобы они не сгорели, включают токоограничивающее сопротивление (ТОС) — $R4$.

   Так как переключение $Q3$ и $Q2$ происходит быстро (~10 нс), а для нагрева транзисторов требуется время, то через $Q2$ и $Q3$ допустимы большие токи. В этой связи $R4$ мало, по сравнению с $R3$.

   $R3$ — ТОС для транзистора $Q1$. $R3>>R4$, так как транзистор $Q1$ может быть насыщен продолжительное время.

4. $U_{вх}1$ меняется от $1.4\ldots1.6B$.

   $Q1$ насыщен, $Q3$ насыщен, $Q2$ заперт.

Для понимания работы базовой схемы ЛЭ ТТЛ в динамическом режиме ответим на **вопрос**: за счет какого схемного решения удалось значительно увеличить скорость переключения ЛЭ ТТЛ из "0"-диапазона в "1"-диапазон и обратно?

Ответ: подключение дополнительных транзисторов $Q2$ и $Q3$, диода D и сопротивлений $R6$, $R4$ позволило значительно (~ в 10 раз) увеличить быстродействие ЛЭ ТТЛ.

Доказательство

Увеличение быстродействия ($\frac{dU_{вых}}{dt}\uparrow$) в значительной мере зависит от перезарядки $C_{вых.параз}$, описываемой формулой $I_C = C\frac{dU_C}{dt}$.

Сравним перезарядку $C_{параз}Q1$ в отсутствии дополнительных элементов с перезарядкой $C_{параз}Q3$.

Зарядка:

1. $C_{параз}Q1$ ($Q1$ — запирается), путь зарядки "+", $R3$, $C_{параз}Q1$, "–".
2. $C_{параз}Q3$ ($Q1, Q3$ — запираются, $Q2$ — насыщается), путь зарядки "+", $R4$, К-Э $Q2$, D, $C_{параз}Q3$, "–". ($R4 \sim 100Ом$, $R3 \sim 1кОм \Rightarrow I_C Q3 > I_C Q1 \Rightarrow$ $\Rightarrow \frac{dU_C Q3}{dt} > \frac{dU_C Q1}{dt}$).

Разрядка:

1. $C_{параз}Q1$ ($Q1$ — насыщается) $I_C Q1 = I_K Q1 - IR3$.
2. $C_{параз}Q3$ ($Q1, Q3$ — насыщаются, $Q2$ — запирается) $I_C Q3 = I_K Q3$.

Из первого и второго пунктов следует, что $I_C Q3 > I_C Q1 \Rightarrow \frac{dU_C Q3}{dt} > \frac{dU_C Q1}{dt}$.

3.2.1. Двухступенчатая логика ЛЭ ТТЛ

Если к схеме подключить транзисторы, отмеченные штриховой линией, то есть организовать еще два входа $U_{вх3}$ и $U_{вх4}$ для двух входных переменных $X3$ и $X4$, то полученная схема будет выполнять логическую операцию: $Y = (X1 * X2 + X3 * X4)!$.

В данном рассмотрении операция "И" — первая ступень, операция "ИЛИ" — вторая ступень, операцию "НЕ" за ступень не считают.

3.2.2. Третье состояние ЛЭ ТТЛ

Некоторые ЛЭ имеют выходной каскад с тремя состояниями выхода: логический "0", логическая "1" и третье состояние (Z-состояние).

Z-состояние — это состояние, когда выход не отдает ток в нагрузку и не потребляет тока из нее (как бы "висит в воздухе"), но может быть легко переведен в другие состояния ("0" или "1").

Такой выход необходим для организации двунаправленных магистралей.

Z-состояние на схеме ЛЭ ТТЛ задают напряжение на входе $U_{3сост.}$ и транзистор $Q3_{3сост.}$.

При подаче напряжения из единичного диапазона на вход ($U_{3сост.}$) транзистор $Q3_{3сост.}$ насыщается и отбирает базовый ток у транзистора $Q2$ и коллекторный ток у транзистора $Q1$. В результате транзисторы $Q2$ и $Q3$ запираются, точка между диодом $D1$ и коллектором $Q3$ изолируется от полюсов источника питания $V2$, и выход ЛЭ ТТЛ переходит в Z-состояние.

3.2.3. ЛЭ ТТЛ с открытым коллектором

Для понимания необходимости открытого коллектора у ЛЭ ТТЛ (рис. 3.3) ответим на **вопрос**: зачем подключать к выходу открытый коллектор?

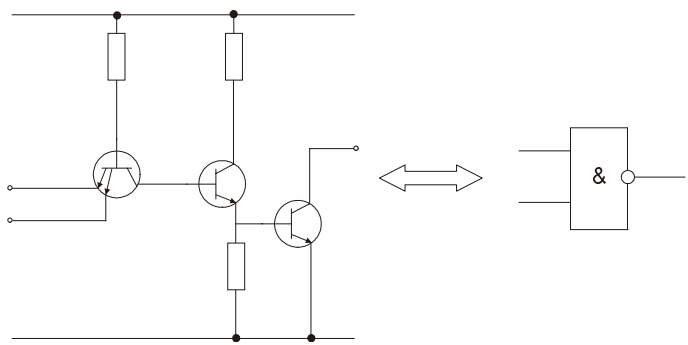


Рис. 3.3. ЛЭ ТТЛ с открытым коллектором

Ответ 1: ЛЭ можно объединить по схеме "монтажное И", тогда получаем двухступенчатую логику (рис. 3.4).

$$(X1 * X2 + X3 * X4)! = (X1 * X2)! * (X3 * X4)!.$$

Ответ 2: открытый коллектор необходим для подключения нагрузок, рассчитанных на напряжение больше чем 5 В (тиристоров, реле, индикаторов и пр.), но при токе в нагрузке не более допустимого выходного коллекторного тока интегральной схемы (ИС) ТТЛ с открытым коллектором.

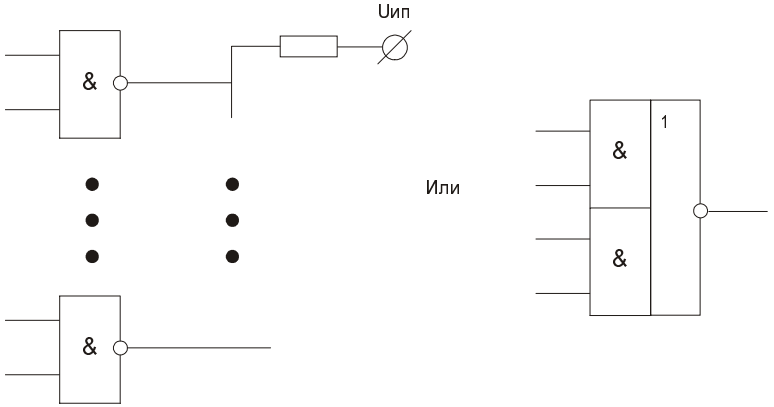


Рис. 3.4. Объединение ЛЭ по схеме "монтажное И"

Лабораторная работа. Исследование базовой схемы логического элемента ТТЛ

Базовая схема элемента ТТЛ (рис. 3.5) состоит из двух частей. Первая часть входная, она реализует функцию "И", содержит резистор $R2$ и многоэмиттерный транзистор $QME1$. Вторая — выходная, реализует функцию инверсии, содержит сложный инвертор на транзисторах $Q1$, $Q2$, $Q3$. Этот инвертор состоит из фазораспределяющего каскада ($Q1, R3, R6$), предназначенного для противофазного переключения транзисторов $Q2$ и $Q3$, и выходного усилителя ($Q2$, $Q3$, $R4$, $D1$).

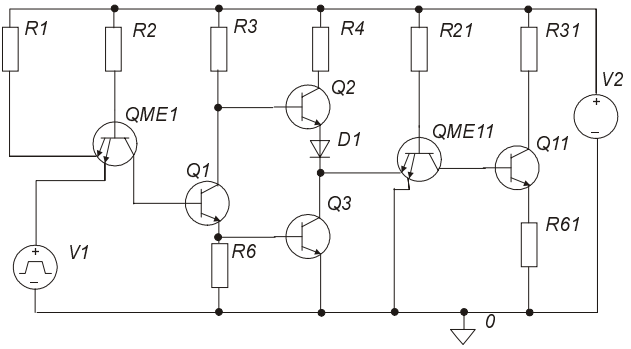


Рис. 3.5. Базовая схема логического элемента ТТЛ ("И-НЕ")

Программа работы

1. Создайте с помощью системы Design Center проект элемента ТТЛ согласно рис. 3.5.
2. Установите следующие значения элементов схемы:
   - значения резисторов — $R1 = 5\ Ом$, $R2 = 10\ кОм$, $R3 = 1\ кОм$, $R4 = 50\ Ом$, $R6 = 2\ кОм$;
   - для источника постоянного напряжения $DC = 5\ В$.
3. Получите характеристику передачи напряжения (ХПН) элемента ТТЛ, изображенную на рис. 3.6. (ХПН строится по трем точкам, которые на рисунке обведены кружком.)

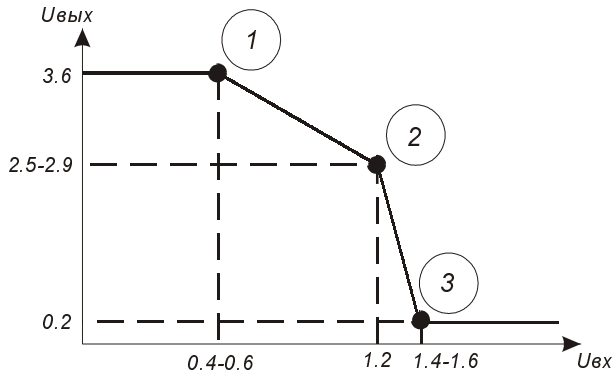
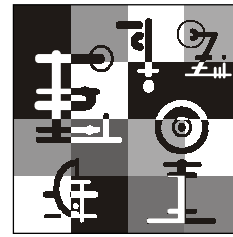


Рис. 3.6. Характеристика передачи напряжения элемента ТТЛ

   - Установите на входе схемы напряжение $U_{вх}$, соответствующее точке 1.
   - Используя кнопку **V**, измерьте напряжение на выходе схемы. Изменяя значения резисторов, добейтесь, чтобы на выходе схемы было напряжение $U_{вых}$, соответствующее этой точке на графике.
   - Повторите п. 3 для точек 2 и 3.
   - Проверьте все три точки ХПН и при необходимости откорректируйте полученные значения резисторов.
4. Получение временных характеристик элемента ТТЛ.
   - Подключите на вход схемы генератор прямоугольных импульсов и установите его параметры: $DC = 0\ В$; $V1 = 0\ В$; $V2 = 0\ В$; $TD = 0$; $TR = TF = 0.01\ нс$; $PW = 300\ нс$; $PER = 600\ нс$.
   - Получите временные характеристики, задав шаг моделирования равным $20\ нс$, а время моделирования — $2\ мкс$.
   - Измерьте время переключения выходного сигнала из "0" в "1" и обратно.
   - Изменяя значения резисторов, добейтесь уменьшения времени переключения.
5. В отчете представьте все полученные результаты и графики.



Глава 4

Эмиттерно-связанная логика (ЭСЛ)

Рассмотрим коэффициенты передачи в схемах с разными включениями транзисторов (табл. 4.1)

Таблица 4.1. *Коэффициенты передачи*

	ОЭ	**ОК**	**ОБ**	**ОК-ОБ** $\left(K_{1,2} = K_1 * K_2\right)$
K_I	$\sim B$	$\sim B$	$\alpha(\sim 1)$	$\sim B$
K_U	$\sim B$	~ 1	$>>1$	$>>1$

Для обеспечения незатухающего сигнала оба коэффициента передачи должны быть >1, поэтому для построения элементов вычислительной техники годятся схемы с ОЭ и ОК-ОБ.

В двухкаскадном усилителе ОК-ОБ, в отличие от элементов ТТЛ (ОЭ), выходное напряжение определяется включением транзистора с ОБ, то есть зависит от стабильного коэффициента α. Использование этого факта позволяет делать ЛЭ, в которых транзисторы не насыщены, а это исключает время рассасывания неравновесных носителей в базе при запирании транзистора и повышает быстродействие ЛЭ.

Но, по сравнению с другими ЛЭ, ЛЭ ЭСЛ при работе потребляют наибольшую энергию.

4.1. Базовая схема серий ЭСЛ

Ее вид представлен на рис. 4.1.

Словарь логической переменной (ЛП) для ЭСЛ:

- **"0"** — значение логической переменной ЭСЛ лежит в диапазоне от -1.9 до $-1.45\ B$;
- **"1"** — значение логической переменной ЭСЛ лежит в диапазоне от -0.95 до $-0.7\ B$.

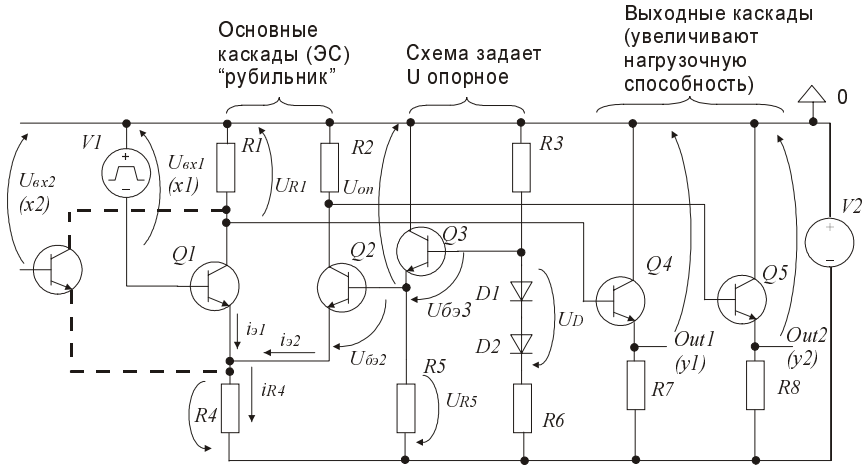


Рис. 4.1. Базовая схема серий ЭСЛ

Механическим аналогом данной схемы может служить "рубильник". Одним концом он прикреплен к стене, к другому прикладывается входное воздействие.

По аналогии, в схеме на базу транзистора $Q2$ относительно "+" источника $V2$ прикладываем приблизительно постоянное напряжение U_{on}, а к базе транзистора $Q1$ и "+" источника $V2$ прикладываем либо напряжение "0", либо напряжение "1", определяемые словарем ЛП.

Для дальнейшего понимания работы базовой схемы ЭСЛ ответим на вопрос: почему, когда транзисторы $Q1$ и $Q2$ открыты и не насыщены, они ведут себя в противофазе?

Для ответа проследим причинно-следственную связь: при больше или меньше открытых друг относительно друга транзисторах $Q1$ и $Q2$ напряжения U_{on} и U_{R5} всегда остаются постоянными ($U_{R5} - U_{on} = V2$).

При больше или меньше открытом $Q2$: $U_{БЭ2} = 0.7B$, будет постоянным $\Rightarrow U_{R4} = U_{R5} - U_{БЭ2}$ тоже постоянно $\Rightarrow$ ток $I_{R4} = \dfrac{U_{R4}}{R4}$ будет постоянным при любых состояниях открытых $Q1$ и $Q2$, но $I_{R4} = I_{Э1} + I_{Э2}$ есть сумма ($I_{Э1} + I_{Э2}$), и она тоже постоянна $\Rightarrow$ если $Q1$ открывается больше и $I_{Э1}\uparrow$, то $Q2$ на столько же "призакрывается" и $I_{Э2}\downarrow$ (это следует из того, что сумма ($I_{Э1} + I_{Э2}$) постоянна).

Приведенная причинно-следственная связь доказывает работу $Q1$ и $Q2$ в противофазе.

Если рассматривать схему, изображенную сплошной и штриховой линиями, то этой схеме будут соответствовать логические операции: "ИЛИ-НЕ" и "ИЛИ" ($y_1 = \overline{x_1 + x_2}$; $y_2 = x_1 + x_2$).

Докажем это.

По аналогии с ТТЛ нужно рассмотреть все возможные комбинации сигналов на входах, состояния транзисторов при каждой комбинации и соответствующие значения сигнала на выходе:

1. Пусть на обоих входах напряжение из "0"-диапазона ("большой минус"), тогда транзистор $Q1$ близок к запиранию, а транзистор $Q2$ — к насыщению, поэтому в базу $Q4$ оттекает больший ток, чем в базу $Q5$, поэтому $Q4$ близок к насыщению и на выходе "$Y1$" напряжение из "1"-диапазона, а $Q5$ близок к запиранию и на выходе "$Y2$" напряжение из "0"-диапазона;
2. Если хотя бы на одном входе напряжение из "1"-диапазона ("малый минус"), то транзистор $Q1$ или параллельно ему включенный транзистор близок к насыщению, а транзистор $Q2$ — к запиранию, поэтому в базу транзистора $Q5$ оттекает больший ток, чем в базу $Q4$, поэтому $Q5$ близок к насыщению и на выходе "$Y2$" напряжение из "1"-диапазона, а $Q4$ близок к запиранию и на выходе "$Y1$" напряжение из "0"-диапазона.

Это рассмотрение позволяет описать таблицей истинности функционирование базовой схемы ЛЭ ЭСЛ (сплошная и штриховая линии) (табл. 4.2).

Таблица 4.2. *Таблица истинности базовой схемы ЛЭ ЭСЛ*

$X1$	$X2$	$Y1$	$Y2$
0	0	1	0
1	0	0	1
0	1	0	1
1	1	0	1

Из таблицы следует, что базовый ЛЭ ЭСЛ выполняет следующие операции:

1. "ИЛИ-НЕ" — $Y1 = \overline{(X1 + X2)}$.
2. "ИЛИ" — $Y2 = (X1 + X2)$.

Для понимания работы базовой схемы ЛЭ ЭСЛ в квазистатическом режиме рассмотрим его характеристики передачи напряжения (ХПН) или зависимости $U_{вых1}(U_{вх1})$ и $U_{вых2}(U_{вх1})$ при подаче на $U_{вх2}$ напряжения из "0"-диапазона (рис. 4.2).

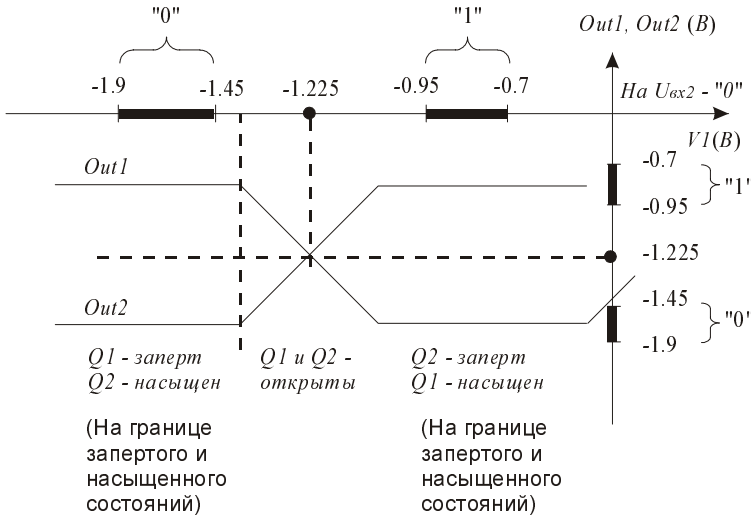


Рис. 4.2. Характеристика передачи напряжения

В дальнейшем, пояснение особенностей работы схемы базового элемента ЭСЛ будет состоять из ответов на ряд вопросов.

Вопрос: в чем причина появления "хвоста" на ХПН $Out1(U_{вх1})$?

Ответ: этот фрагмент ХПН соответствует запертому $Q2$ и насыщенному $Q1 \Rightarrow U_{бэ1} = 0.7B$, $U_{кэ1} = 0B$ и (из $U_{кэ1} = U_{бэ1} - U_{бк1}$) $U_{бк1} = 0.7B$, то есть как бы мы ни увеличивали $U_{вх1}$, $U_{бк1}$ будет всегда $= 0.7B$. Из 2 закона Кирхгофа имеем: $U_{вх1} = U_{R1} + U_{бк1} \Rightarrow$ при увеличении $U_{вх1}$ на столько же увеличится U_{R1}. Далее, из 2 закона Кирхгофа: $U_{Y1} = U_{R1} - U_{бэ}Q4$, транзистор $Q4$ все время открыт, то есть $U_{бэ}Q4 = 0.7B = const$. Из этого следует, что при изменении $U_{вх1}$ ровно на столько же изменятся U_{R1} и U_{Y1}. Этим объясняется наличие "хвоста" на ХПН.

Вопрос: для чего нужны два диода ($D1$ и $D2$)?

Ответ: $D1$ и $D2$ — термокомпенсирующие диоды. Они необходимы для обеспечения постоянства напряжений $U_{оп}$ и U_{R5}.

Из 2 закона Кирхгофа имеем: $U_{R5} = U_D - U_{бэ}Q3 + U_{R6}$. При повышении температуры на одинаковые величины увеличатся значения напряжений $U_{бэ}Q3$ и U_D при $U_{R6} \cong const$, и так как значения напряжений $U_{бэ}Q3$ и U_D в выражении, определяющем напряжение U_{R5}, имеют разные знаки, то суммарное изменение $U_{бэ}Q3$ и U_D будет равно нулю, напряжение U_{R5}, а следовательно, и напряжение $U_{оп}$ останутся постоянными.

Вопрос: для чего нужны выходные каскады на транзисторах $Q4$ и $Q5$?

Ответ: каскады нужны для увеличения нагрузочной способности схемы базового элемента ЭСЛ.

На транзисторах $Q4$ и $Q5$ собраны каскады с ОК или иначе эмиттерные повторители (ЭП) напряжения. Выходные напряжения можно было бы снимать с сопротивлений $R1(U_{R1})$ и $R2(U_{R2})$, но для подачи с выхода на несколько входов следующих ЛЭ необходим большой ток (чтобы хватило каждому входу), при этом необходимо сохранить значения выходных напряжений, так как в них заложена информация — логические "0" и "1". ЭП на $Q4$ и $Q5$ увеличивают выходные сигналы по току (в "В+1" раз) и приблизительно сохраняют по напряжению (отличие на $0.7B$: $U_{Y1} = U_{R1} - U_{БЭ}Q4 = U_{R4} - 0.7B$).

Вопрос: что понимается под параметрическим синтезом базовой схемы ЭСЛ?

Ответ: параметрический синтез схемы может быть направлен на обеспечение функциональной устойчивости работы схемы ЭСЛ. При параметрическом синтезе значения $R1$, $R2$ и $R4$ подбираются так, чтобы выходной язык логической переменной соответствовал входному языку, то есть чтобы диапазоны "0" и "1" на входе совпадали с диапазонами "0" и "1" на выходе.

Лабораторная работа. Исследование базовой схемы логического элемента ЭСЛ

Базовая схема элемента ЭСЛ (рис. 4.3) состоит из входного дифференциального каскада на транзисторах $Q1$ и $Q2$, источника опорного напряжения на транзисторе $Q3$, резисторов $R3$ и $R6$, задающих режим работы транзистора $Q3$ и эмиттерных повторителей на транзисторах $Q4$ и $Q5$.

Если на входе элемента напряжение, соответствующее логическому "0", то транзистор $Q1$ закрыт, ток протекает только через транзистор $Q2$. На коллекторе $Q1$ присутствует напряжение, соответствующее логической "1", а на коллекторе $Q2$ — логическому "0", эти напряжения через эмиттерные повторители поступают на выходы $Out1$ и $Out2$ соответственно.

Если на вход элемента поступает напряжение логической "1", то транзистор $Q1$ открывается, а транзистор $Q2$ закрывается. Теперь напряжение на коллекторе $Q1$ соответствует логическому "0", а на коллекторе $Q2$ — логической "1". Эти уровни напряжений также поступают на выходы $Out1$ и $Out2$.

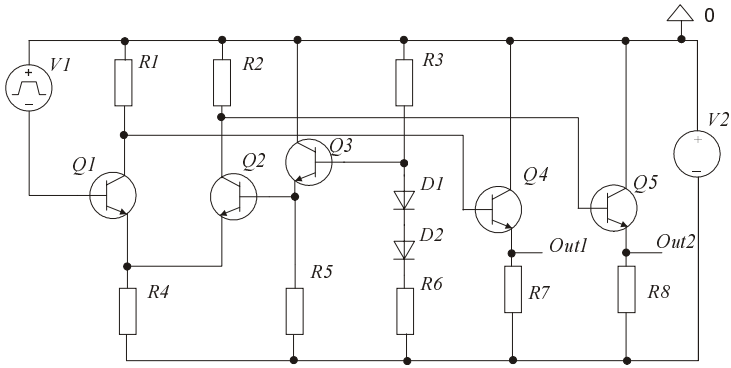


Рис. 4.3. Базовая схема элемента ЭСЛ

Программа работы

1. Создайте проект элемента ЭСЛ согласно рис. 4.3.
2. Установите следующие значения элементов схемы:
   - значения резисторов — $R1 = 250\ Ом$, $R2 = 300\ Ом$, $R3 = 270\ Ом$, $R4 = 1.1\ кОм$, $R5 = 1\ кОм$; $R6 = 2.1\ кОм$; $R7 = R8 = 1.1\ кОм$;
   - для источника постоянного напряжения $DC = 5\ B$.
3. Установите напряжение $U_{on} = -1.225\ B$ на базе транзистора $Q2$ (эмиттере транзистора $Q3$).
4. На вход транзистора $Q1$ подключите источник постоянного напряжения и установите параметры $DC = -1.225\ B$; $AC = 0$.
5. Используя кнопку **V**, измерьте напряжение на базе транзистора $Q2$. Для получения нужного значения U_{on} необходимо изменять значения резисторов $R3$ и $R6$. Подобранные значения резисторов $R3$ и $R6$ должны удовлетворять следующим ограничениям:

$$100\ Ом < R3 < 1\ кОм \qquad\qquad 1\ кОм < R6 < 5\ кОм.$$

Процесс подбора номиналов $R3$ и $R6$ отразите в табл. 4.3.

Таблица 4.3. *Таблица подбора номиналов резисторов*

$R3$	$R6$	U_{on}
Ом	Ом	В
270	2100	
…	…	…

6. Подбором значений сопротивлений добейтесь того, чтобы характеристика передачи напряжения (ХПН) элемента ЭСЛ находилась в стандартных границах. Для положительной логики за логическую "1" принят диапазон от -0.95 до $-0.7\ B$ ($U_{1ср} = -0.825\ B$), а за логический "0" — диапазон от -1.9 до $-1.45\ B$ ($U_{0ср} = -1.675\ B$).

   ХПН должна выглядеть в соответствии с рис. 4.2.

7. Установите на входе схемы напряжение логического "0" $U_{0ср}$. Используя кнопку **V**, измерьте напряжение на выходах схемы. Подбирая величины резисторов $R1$, $R2$, $R4$, получите на выходах схемы напряжения, близкие к $U_{1ср}$ и $U_{0ср}$.

8. То же самое проделайте для входного напряжения $U_{1ср}$. Полученные значения резисторов $R1$, $R2$, $R4$ должны удовлетворять следующим ограничениям:

   $200\ Ом < R1,\ R2 < 5\ кОм$;

   $1\ кОм < R4 < 100\ кОм$.

9. По результатам выполнения предыдущих пунктов в отчете представьте 2 таблицы (табл. 4.4 и 4.5).

Таблица 4.4. $U_{вх} = U_{0ср}$

$R1$	$R2$	$R4$	**Выходы схемы**	
Ом	*Ом*	*Ом*	**прямой**	**инверсный**
			В	*В*
…	…	…	…	…

Таблица 4.5. $U_{вх} = U_{1ср}$

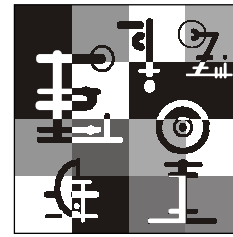
$R1$	$R2$	$R4$	**Выходы схемы**	
Ом	*Ом*	*Ом*	**прямой**	**инверсный**
			В	*В*
…	…	…	…	…

В последней строке таблицы должны быть приведены итоговые величины резисторов и значения напряжений на выходах схемы.

10. Получение временных характеристик элемента ЭСЛ.

- Подключите на вход схемы генератор прямоугольных импульсов и установите его параметры: $DC = 0\ B$; $V1 = U_{0ср}$; $V2 = U_{1ср}$; $TD = 0\ нс$; $TR = TF = 0.01\ нс$; $PW = 300\ нс$; $PER = 600\ нс$.
- Получите временные характеристики, задав шаг моделирования равным $20\ нс$, время моделирования $2\ мкс$.
- Получите на экране монитора одновременно графики входных и выходных напряжений. Определите, какой выход элемента является прямым, а какой — инверсным.
- Измерьте время переключения выходного сигнала из "0" в "1" и обратно для каждого из выходов схемы.

11. В отчете представьте все полученные результаты и графики.



Глава 5

Логические элементы на МДП-транзисторах

Будем считать, что МДП-транзисторы и МОП-транзисторы — одно и то же (МДП — металл-диэлектрик-полупроводник, МОП — металл-окисел-полупроводник).

Логические элементы (ЛЭ) на МДП-транзисторах условно подразделяются на статические ЛЭ и динамические ЛЭ. Статические ЛЭ, в свою очередь, подразделяются на ЛЭ на "n"-МДП-транзисторах и на ЛЭ на КМДП-схемах. Динамические ЛЭ подразделяются на ЛЭ на двухфазных динамических МДП-схемах и ЛЭ на четырехфазных динамических МДП-схемах.

В дальнейшем рассмотрим ЛЭ на КМДП-схемах, как получившие наибольшее развитие и распространение.

5.1. МДП-транзисторы

Рассмотрим схематическое изображение в разрезе МДП-транзистора (рис. 5.1) с индуцированным "n"-каналом (следовательно, "p"-подложкой).

Здесь, в пластине "p"-типа, называемой подложкой, имеются две диффузионные области "n"-типа, которые могут соединяться наводимым под действием напряжения $U_{зи}$ (затвор-исток) каналом "n"-типа.

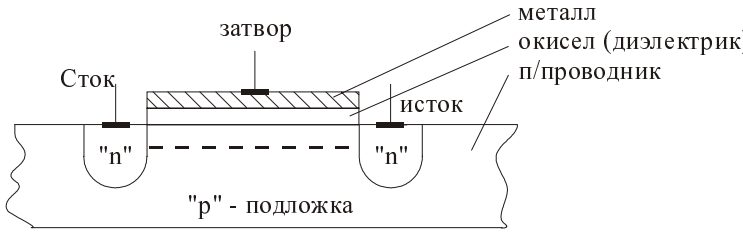


Рис. 5.1. Схематическое изображение МДП-транзистора

Различают два вида МДП-транзисторов:

- *обедненный* МДП-транзистор или транзистор *со встроенным каналом*, у него канал сформирован при изготовлении;
- *обогащенный* МДП-транзистор или транзистор *с индуцированным или наведенным каналом*, здесь канал появляется только тогда, когда между двумя выводами (затвор и исток) приложено напряжение, превышающее определенное значение ($U_{пороговое}$).

Канал также может быть двух видов: "p"-канал (следовательно "n"-подложка) и "n"-канал (следовательно "p"-подложка).

Управляющим выводом в МДП-транзисторе является *затвор (З)*.

Область, из которой в результате подключения питания **основные** носители (в данном случае — электроны) входят в канал, называют *истоком (И)* (к нему подключен "–" источника), другую область называют *стоком (С)*.

5.1.1. Условное графическое изображение МДП-транзисторов

На рис. 5.2 представлено условное графическое обозначение МДП-транзисторов: слева — подложка "p"-типа с "n"-индуцированным каналом; справа — подложка "n"-типа с "p"-индуцированным каналом.

На рис. 5.3 представлено условное графическое обозначение МДП-транзисторов: слева — подложка "p"-типа с "n"-встроенным каналом; справа — подложка "n"-типа с "p"-встроенным каналом.

Рис. 5.2. Условное графическое обозначение МДП-транзисторов с "n"- и "p"-индуцированными каналами

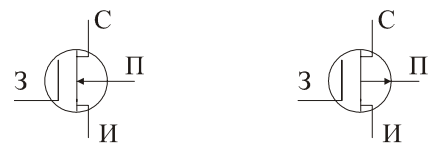


Рис. 5.3. Условное графическое обозначение МДП-транзисторов с "n"- и "p"-встроенными каналами

В интегральных логических схемах *подложка* как управляющий вывод не используется и ее потенциал относительно *истока* постоянен и равен нулю (она накоротко соединена с *истоком*). Поэтому МДП-транзисторы являются трехполюсниками.

Рассмотрим проходные характеристики $I_c = I_c(U_{зи})$ разных МДП-транзисторов (I_c — ток стока, $U_{зи}$ — напряжение затвор-исток) (рис. 5.4 и 5.5).

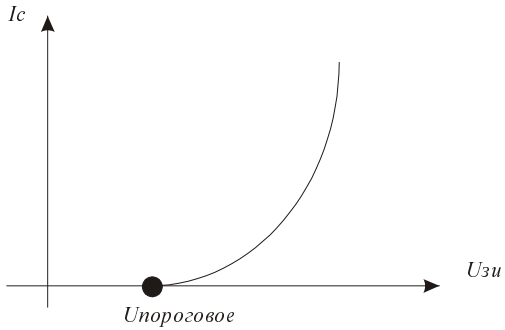


Рис. 5.4. Проходная характеристика транзистора с "n"-индуцированным каналом

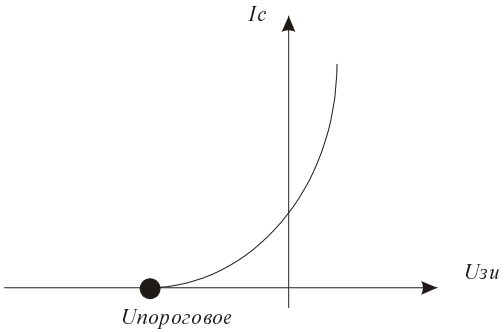


Рис. 5.5. Проходная характеристика транзистора с "n"-встроенным каналом

Напряжение $U_{пороговое}$ (или $U_{отсечки}$) — напряжение, при котором начинает образовываться канал в МДП-транзисторе.

Характеристики "p"-канальных МДП-транзисторов отличаются противоположными знаками напряжений и токов.

5.2. Комплементарные ЛЭ на МДП-транзисторах

Иначе эти ЛЭ называют *ЛЭ на КМДП-схемах*.

Комплементарные схемы — схемы, содержащие одинаково используемые транзисторы разных типов проводимости (complement — дополнять).

Рассмотрим работу инвертора на КМДП-схемах (рис. 5.6).

Эта схема симметрична относительно горизонтальной линии (у верхнего транзистора *исток* вверху, у нижнего транзистора *исток* внизу). Транзисторы работают в одинаковых условиях.

Отпирающим напряжением для нижнего транзистора является положительное напряжение затвор-исток: $U_{зи}^{n} > U_{пороговое}$.

Отпирающим напряжением для верхнего транзистора является отрицательное напряжение затвор-исток: $U_{зи}^{p} < -U_{пороговое}$.

Зная работу МДП-транзисторов, докажем, что представленная схема — схема инвертора:

1. За логический ноль примем напряжение, равное $0\,B$, за логическую единицу напряжение, равное $V = V1$ (приблизительно $5\,B$).
2. Опишем работу схемы в табличном виде (табл. 5.1).

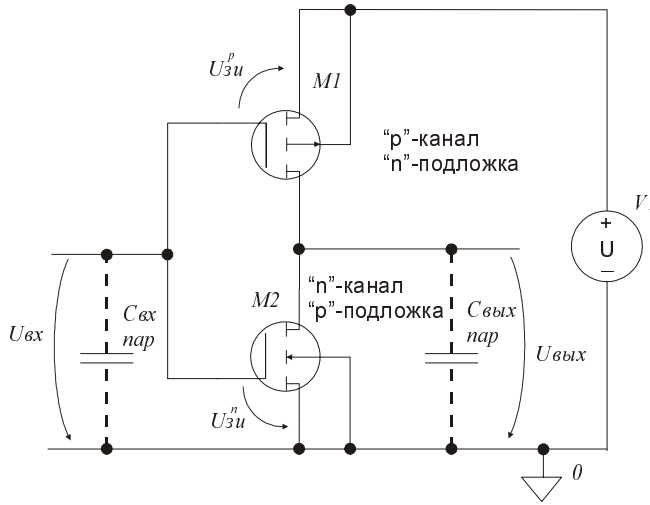


Рис. 5.6. Схема инвертора

Таблица 5.1. *Работа схемы инвертора*

$U_{вх}$ =	U_x	0	V
M1:	$U_{зи}^{p}$	$-V$ — открыт	0 — заперт
M2:	$U_{зи}^{n}$	0 — заперт	V — открыт
$U_{вых}$ =	U_Y	V	0

3. Из таблицы следует, что, подавая на вход "0", получаем "1", и наоборот, а это значит, что перед нами схема инвертора.

Основным преимуществом ЛЭ на КМДП-схемах является малое энергопотребление, так как здесь отсутствует протекание сквозного тока за исключением момента смены значения логической переменной.

К недостаткам можно отнести значительное усложнение технологического процесса изготовления.

5.2.1. О нагрузочной способности ЛЭ на КМДП-схемах

При параллельном подключении входов нагрузок к выходу ЛЭ на КМДП-схеме суммарная паразитная емкость ($C_{\sum паразит.}$) увеличивается.

Перезарядкой этой емкости определяется быстродействие работы ЛЭ, а именно величина скорости изменения напряжения на емкости: $\frac{dU_c}{dt}$. Перезарядка описывается следующей формулой:

$$i_c = \pm C_{\sum параз.} \frac{dU_c}{dt}.$$

Если предположить, что i_c постоянен и $C_{\sum параз.}$ увеличивается, то из формулы видно, что $\frac{dU_c}{dt}$ будет уменьшаться, поэтому быстродействие переключения из "0" в "1" и обратно тоже будет уменьшаться.

Таким образом, из приведенного рассмотрения следует, что нагрузочная способность ЛЭ на КМДП-схемах ограничивается требованием к быстродействию работы этих ЛЭ.

5.2.2. Базовые схемы серий логики на КМДП-схемах

На КМДП-схемах необычно реализуются ЛЭ для операций "И-НЕ" и "ИЛИ-НЕ".

Рассмотрим логическую операцию (ЛО) "ИЛИ-НЕ": $y = (x1 + x2)!$.

Обычно эту операцию получали параллельным соединением выходов, здесь такая схема не работает, имеем схему, представленную на рис. 5.7.

Логическую операцию "И-НЕ" ($y = (x1 * x2 * x3)!$) реализует схема, представленная на рис. 5.8.

Доказательство того, что представленные схемы реализуют именно те ЛО, а не другие, предлагается провести самостоятельно по аналогии с ЛЭ ТТЛ и ЭСЛ.

Таким образом, ЛЭ на КМДП-схемах имеет две базовые схемы, выполняющие операции "И-НЕ" и "ИЛИ-НЕ", соответственно.

К преимуществу ЛЭ на КМДП-схемах можно отнести то, что у этих элементов можно использовать способность сохранять входной заряд на паразитной емко-

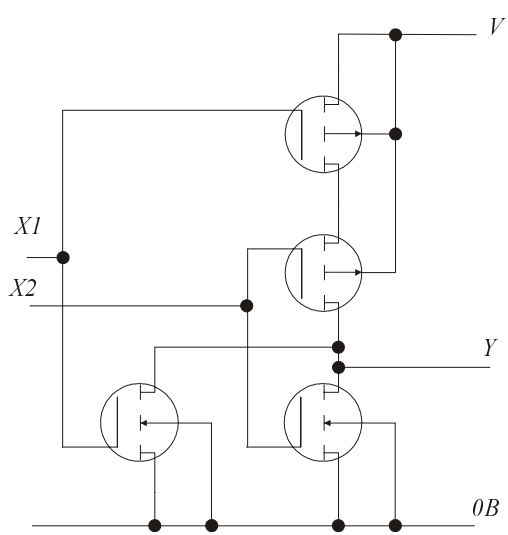


Рис. 5.7. Схема, реализующая операцию "ИЛИ-НЕ"

сти $C_{вх} = C_{зи}$. Это позволяет выполнять на рассматриваемых элементах тактированные или динамические схемы с внутренним запоминанием.

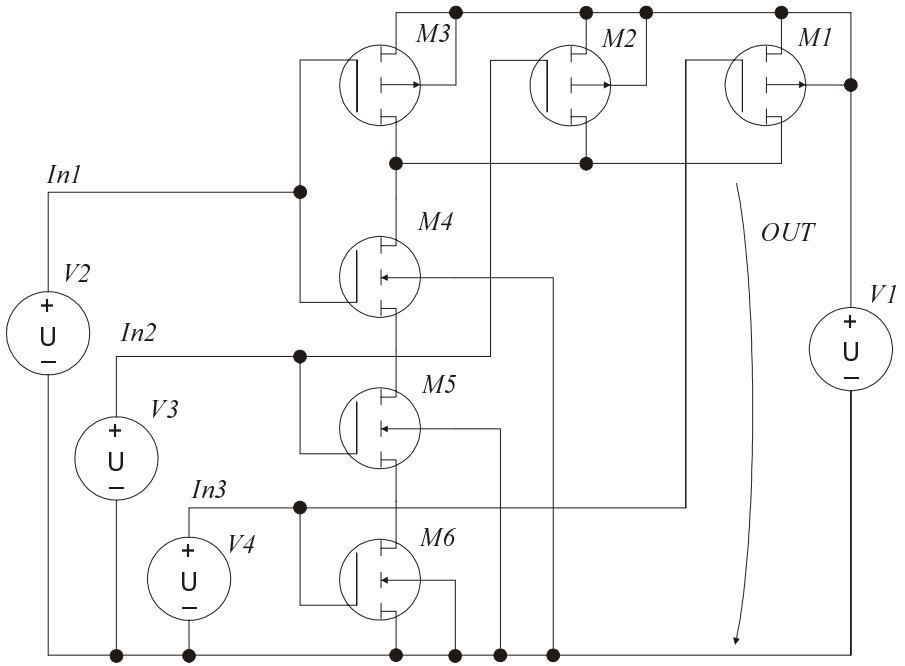


Рис. 5.8. Схема, реализующая операцию "И-НЕ"

Лабораторная работа. Исследование базовой схемы логического элемента на КМДП-транзисторах

Программа работы

1. Создайте проект логического элемента согласно рис. 5.8. При создании проекта используйте модель транзистора типа MbreakN4 и MbreakP4.
2. Получите временные диаграммы на выходе элемента.
   - Установите на входах элемента генераторы импульсов VPULSE. Значения параметров ($DC = 0$, $V1 = 0$, $V2 = 5\ В$, $TD = 0$, $TR = TF = 1\ нс$) являются

одинаковыми для всех трех генераторов импульсов. Параметры PW и PER устанавливаются следующими:

◊ для 1-го генератора — $PW = 100$ *нс*, $PER = 200$ *нс*;

◊ для 2-го генератора — $PW = 200$ *нс*, $PER = 400$ *нс*;

◊ для 3-го генератора — $PW = 400$ *нс*, $PER = 800$ *нс*.

- Установите для этапа моделирования шаг моделирования (**Print Step**) равным 20 *нс* и время моделирования — 2 *мкс*.
- Получите на экране монитора одновременно временные диаграммы входных и выходного напряжений.

3. Определите логическую функцию, выполняемую элементом. Результаты исследования представьте в табл. 5.2.

Таблица 5.2. *Логическая функция, выполняемая элементом*

In1	***In2***	***In3***	***Out***

4. Определите, при каких комбинациях входных сигналов возникают искажения выходного сигнала. Объясните причину этих искажений.

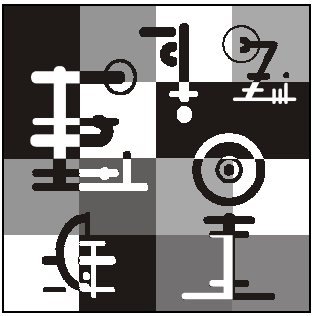
   Добейтесь исчезновения искажений выходного сигнала, используя задержку между входными импульсами (параметр TD).

5. Исследуйте влияние нагрузки на выходной сигнал. Нагрузка (вход следующей микросхемы) моделируется подключением на выход элемента конденсатора емкостью 1 *пф*. Подключение нескольких нагрузок моделируется увеличением емкости в нужное число раз. Составьте таблицу зависимости длительности положительного фронта выходного сигнала от величины емкости конденсатора по типу табл. 5.3.

Таблица 5.3. *Зависимость длительности сигнала от емкости конденсатора*

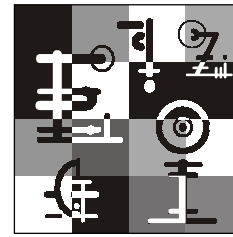
Величина емкости конденсатора, пф	**Длительность положительного фронта, нс**

6. В отчете представьте все полученные результаты и графики.



Часть II

Комбинационные и последовательностные логические устройства, их описание на языке VHDL. Средство проектирования фирмы Altera — MAX+PLUS II



Глава 6

Комбинационные и последовательностные логические устройства

С точки зрения учета предыстории, логические устройства (ЛУ) делятся на комбинационные (КЛУ) и последовательностные (ПЛУ). ПЛУ иначе называют автоматы с памятью.

В КЛУ значения сигналов на выходе в текущий момент времени зависят от значений входных сигналов в этот же момент времени. То есть то, что было на входах до этого момента, никак не влияет на значения сигналов на выходе в этот момент времени. В ПЛУ предыстория влияет на значения сигналов на выходе в текущий момент времени.

6.1. Проектирование КЛУ на примере разработки логической схемы комбинационного полусумматора

Полусумматором (ПС) назовем одноразрядный двоичный сумматор на два входа. Необходимо спроектировать его логическую схему.

Этапы проектирования КЛУ.

1. КЛУ представляют "черным" ящиком с описанием входов и выходов.
2. КЛУ представляют переключательной или логической функцией, описанной таблицей истинности.
3. Переводят заданную переключательную функцию с языка табличного описания на символический язык с одновременным представлением переключательной функции через функции выбранной функционально полной системы (через "ИЛИ-НЕ" или "И-НЕ").

4. Полученную функцию записывают в канонической форме представления: СДНФ или СКНФ (совершенные дизъюнктивная или конъюнктивная нормальные формы).
5. Минимизируют или упрощают выражения переключательной функции (карта Карно), т. к. СДНФ или СКНФ неудобны для построения логических схем, они избыточны и схемы оказываются сложными.

Рассмотрим по шагам алгоритм проектирования КЛУ на примере комбинационного полусумматора:

1. Представление проектируемого устройства "черным" ящиком с описанием входов и выходов (рис. 6.1).

   Здесь X и Y — слагаемые, C — сумма, $П$ — перенос.
2. Описание таблицей истинности закона функционирования ПС или его переключательных функций (табл. 6.1).

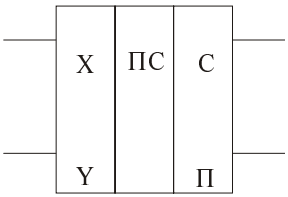


Рис. 6.1. УГО полусумматора

Таблица 6.1. *Таблица истинности полусумматора*

X	**У**	**C**	**П**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3. Запись переключательных функций для C (суммы) и $П$ (переноса) в СДНФ и СКНФ.

Об алгоритме записи переключательной функции в СДНФ

При записи перебирают все возможные наборы входных переменных, которые соответствуют *C* = 1 и *П* = 1, т. е. "ловим" 1.

В СДНФ:

$$C = (X!*Y) + (X*Y!) = X \bmod 2\, Y = X \text{ XOR } Y = X \text{ ЛИБО } Y$$

$$П = X*Y$$

Здесь ! обозначает операцию инверсии, т. е. $X!$ — инверсия X.

Об алгоритме записи переключательной функции в СКНФ

При записи перебирают все возможные наборы входных переменных, которые соответствуют С = 0 и П = 0, т. е. "ловим" 0.

В СКНФ:

$C = (X + Y) * (X! + Y!)$

$П = (X + Y) * (X + Y!) * (X! + Y)$

С точки зрения наименьшего числа операций выбирается запись переключательных функций для C (суммы) и $П$ (переноса) в СДНФ.

4. Соответствующая логическая схема выглядит так, как показано на рис. 6.2.

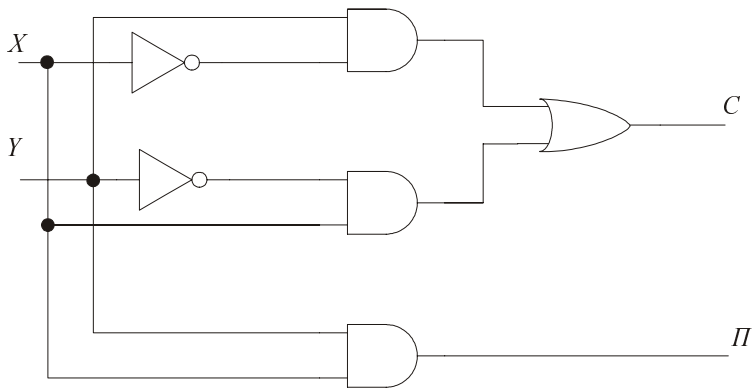


Рис. 6.2. Логическая схема для СДНФ

5. При проектировании необходимо, как минимум, выполнение двух требований:
   - обеспечение минимального количества по возможности однотипных ЛЭ;
   - получение максимального быстродействия, для чего строят схему с минимальным числом ступеней переработки информации.

   Для уменьшения количества ЛЭ исходные переключательные функции представим в виде:

   $C = (X + Y) * П!$

   $П = X * Y$.

   (для этого нужно доказать, что $(X! * Y) + (X * Y!) = (X + Y) * (X + Y)!$, используя правила Де'Моргана: $(X + Y)! = X! * Y!$ и $(X * Y)! = X! + Y!$).

   Соответствующая логическая схема выглядит так, как показано на рис. 6.3.

   Если в первую очередь при проектировании нужно учесть однотипность ЛЭ, например на "ИЛИ-НЕ", то переключательные функции записывают в виде:

   $C = ((X + Y)! + (X! + Y!)!)!$

   $П = (X! + Y!)!$

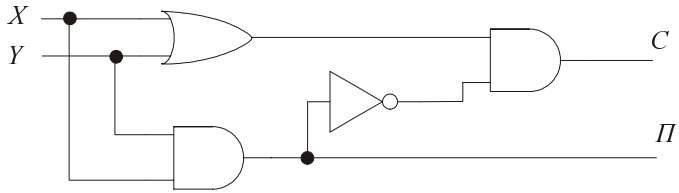


Рис. 6.3. Логическая схема полусумматора после минимизации

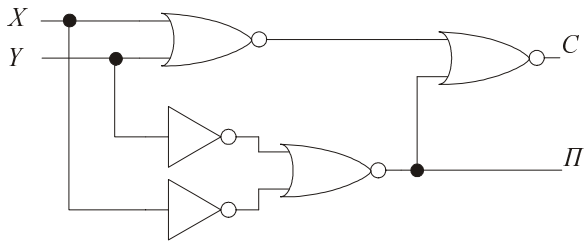


Рис. 6.4. Логическая схема полусумматора в базисе "ИЛИ-НЕ"

Соответствующая логическая схема выглядит так, как показано на рис. 6.4.

Если разряды слагаемых подаются от триггерного регистра сдвига, то используют как X и Y, так и их инверсные значения $X!$ и $Y!$, при этом исчезает одна ступень переработки информации (инвертирование).

6.1.1. Проектирование логической схемы комбинационного одноразрядного двоичного сумматора (См)

1. Представим проектируемое устройство как "черный" ящик с описанием входов и выходов (рис. 6.5).

   Здесь X и Y — разряды слагаемых, Z — перенос из предыдущего разряда, C — сумма, $П$ — перенос в следующий разряд.

2. Опишем таблицей истинности закон функционирования См или его переключательную функцию (табл. 6.2).

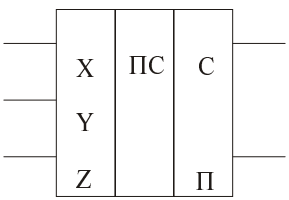


Рис. 6.5. УГО полусумматора

Таблица 6.2. *Таблица истинности сумматора*

X	У	Z	С	П
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3. Запись переключательных функций для C (суммы) и Π (переноса):

- Первый вариант (в СДНФ и в СКНФ).

  В СДНФ:

  $C = X!*Y!*Z + X!*Y*Z! + X*Y!*Z! + X*Y*Z$

  $\Pi = X!*Y*Z + X*Y!*Z + X*Y*Z! + X*Y*Z = X*Y + X*Z + Y*Z$

  В СКНФ:

  $C = (X+Y+Z)*(X+Y!+Z!)*(X!+Y+Z!)*(X!+Y!+Z)$

  $\Pi = (X+Y+Z)*(X+Y+Z!)*(X+Y!+Z)*(X!+Y+Z) = (X+Y)*(X+Z)*(Y+Z)$

- Второй вариант соответствует случаю, когда при образовании суммы учитывается результат переноса.

  $C = \Pi!*((X+Y)+Z)+(X*Y)*Z$

  $\Pi = X*Y+(X+Y)*Z$

  Логическая схема выглядит так, как показано на рис. 6.6.

- Третий вариант. Одноразрядный сумматор можно спроектировать из двух полусумматоров и одного дизъюнктора. В этом случае переключательная функция:

  $C = C_{ПС} * Z! + C_{ПС}!*Z$

  $\Pi = \Pi_{ПС} + C_{ПС} * Z$, где $C_{ПС} = X*Y! + X!*Y \quad \Pi_{ПС} = X*Y$

  Соответствующая логическая схема выглядит так, как показано на рис. 6.7.

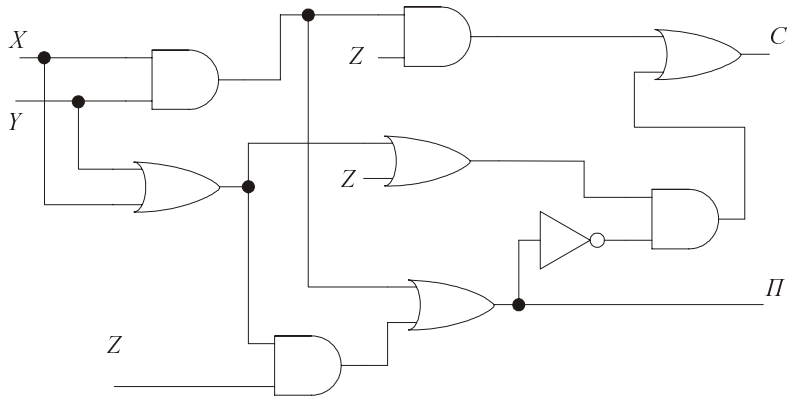


Рис. 6.6. Логическая схема одноразрядного сумматора

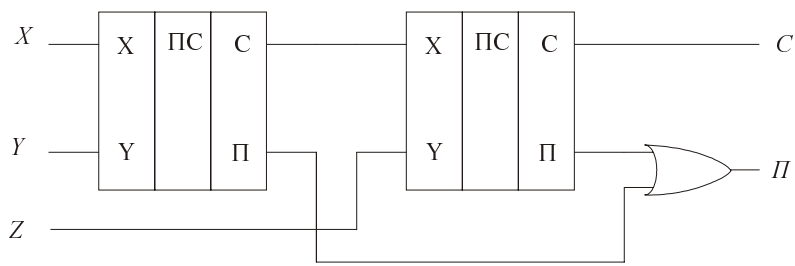


Рис. 6.7. Схема сумматора из двух полусумматоров

6.2. Вентили. Вентильное проектирование

ЛЭ, реализующие операцию "ИЛИ-НЕ" и "И-НЕ", иначе называют вентиль на "ИЛИ-НЕ" и вентиль на "И-НЕ".

При подаче на один из входов (управляющий) вентиля "ИЛИ-НЕ" логической "1" на его выходе всегда будет логический "0", что бы ни подавалось на его второй вход, поэтому вентиль закрыт.

Если на управляющий вход подать "0", то на выход будет поступать инвертированный сигнал со второго входа и вентиль будет открыт.

Для вентиля на "И-НЕ" закрытое состояние задается подачей на управляющий вход "0", при этом на выходе вентиля всегда будет "1", а открытое состояние — подачей на управляющий вход "1".

Если проектирование ведется с использованием ЛЭ "И-НЕ" и "ИЛИ-НЕ", то оно называется *вентильным*.

6.3. Минимизация переключательных функций логических устройств

Наилучшим методом минимизации при числе входов, не превышающим четырех, является составление карты Карно. Рассмотрим составление карты и минимизацию на примере.

Задача

Синтезировать мажоритарный элемент на три входа (A, B, C):

- в базисе "И-НЕ";
- в базисе "ИЛИ-НЕ".

> ***Примечание***
>
> Мажоритарный элемент — это элемент, значение выходного сигнала которого совпадает со значением большинства входных сигналов.

Решение

1. Составляем таблицу истинности (задаем переключательную функцию табличным видом — табл. 6.3)

Таблица 6.3. *Таблица истинности мажоритарного элемента*

C	**B**	**A**	**У**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

2. Наносим функцию на карту Карно. Для этого выполняем следующие действия.
   - Разбиваем входные переменные на комбинации AB и C, они должны быть расположены таким образом, чтобы при переходе от одной комбинации к соседней менялось бы состояние только одной переменной.

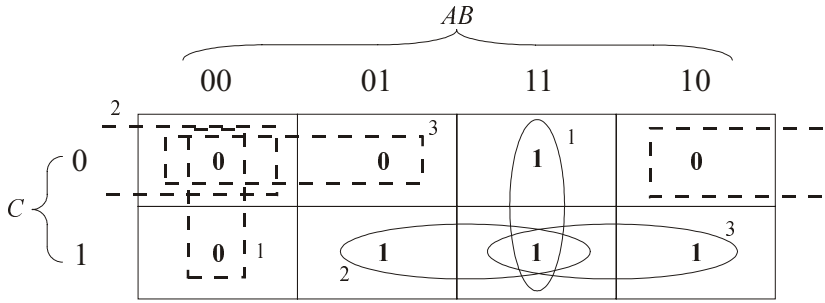


Рис. 6.8. Карта Карно

- На тех пересечениях, где комбинации входных переменных дают "1" и "0" рассматриваемой функции, пишем эти значения, если значения не определены, пишем (Н).
- Составляем минимизирующие контуры, исключаем переменные и пишем переключательную функцию в соответствии с необходимым базисом:
  - ◊ в базисе "И-НЕ" по "1" (единичным) контурам имеем:

$$Y = \overset{1}{A} * B + \overset{2}{B} * C + \overset{3}{A} * C = (A * B + B * C + A * C)!! = ((A * B)! * (B * C)! * (A * C)!)! ;$$

  - ◊ в базисе "ИЛИ-НЕ" по "0" (нулевым) контурам имеем:

$$Y = (\overset{1}{A} + B) * (\overset{2}{B} + C) * (\overset{3}{A} + C) = ((A + B)! + (B + C)! + (A + C)!)! .$$

Лабораторная работа. Система автоматизированного проектирования фирмы Альтера Max+Plus II. Схемы одноразрядных двоичных полусумматора и сумматора

Упрощенно работа с САПР фирмы Альтера Max+Plus II (как и с другими САПР, позволяющими на выходе получить готовый кристалл (чип) проекта) состоит из выполнения ряда этапов:

1. Создание проекта с помощью редакторов проектов (графического и текстового).

2. Компиляция проекта, в которую входят его анализ, детализация и синтез.
3. Работа в редакторе временных диаграмм (вэформере), в котором задаются входные воздействия.
4. Симуляция или моделирование работы проекта во времени, которое необходимо для тестирования проекта на соответствие спецификации (техническому заданию).
5. Размещение и трассировка схемы проекта на кристалле.
6. Программирование (конфигурирование) проекта в программируемую логическую интегральную схему (ПЛИС).

В этой и ряде следующих лабораторных работ ограничимся первыми четырьмя этапами работы с САПР.

Цель работы

Данная работа предназначена для ознакомления:

- с системой автоматизированного проектирования (САПР) электронных устройств Max+Plus II;
- с работой в графическом редакторе этой системы.

В качестве примера выбрана схема одноразрядного сумматора, выполненного на полусумматорах.

Программа работы

1. Создание нового проекта или открытие существующего.
   - Запустить Max+Plus II.
   - Установить имя проекта, выбрав команду меню **File** | **Name** (Файл | Имя). Удобно, чтобы все файлы одного проекта находились в отдельной папке. Если она не была создана, выбрать папку Max2work, в поле имени проекта ввести *папка\имя проекта* и подтвердить создание директории.
   - Для создания нового файла схемы разрабатываемого устройства выбрать кнопку **New** на панели инструментов или использовать команду меню **File** | **New** (Файл | Новый). (Для открытия существующего файла щелкнуть по кнопке **Open** или выполнить команду меню **File** | **Open** (Файл | Открыть)).
   - В диалоговом окне выбрать один из редакторов для создания нового файла изображения схемы. В нашем примере выбираем графический редактор Graphic Editor. Файл изображения схемы будет иметь расширение gdf.

- В окне выбранного редактора сохранить файл с нужным именем (**File** | **Save as** (Файл | Сохранить как)).

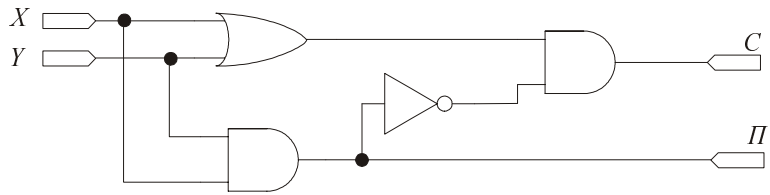


Рис. 6.9. Схема одноразрядного полусумматора

2. Ввод функциональной схемы полусумматора с помощью Graphic Editor (рис. 6.9).
   - Для ввода элементов схемы дважды щелкнуть левой кнопкой мыши в выбранной позиции экрана. В диалоговом окне **Enter Symbol** (Ввод | Символ), открытом через контекстное меню, выбрать нужную библиотеку элементов (для данного примера библиотеку примитивов prim) и имя элемента. Повторить процедуру для всех элементов схемы (имена символов в библиотеке: `OR2` — "ИЛИ", `AND2` — "И", `NOT` — "Инвертор").
   - Для удобства работы в рабочем поле проекта используется сетка для привязки элементов к координатам экрана. Чтобы показать сетку, нужно выполнить команду меню **Options** | **Show Gridelines** (Опции | Показать сетку). Для изменения размеров сетки используется команда меню **Options** | **Grideline Spacing** (Опции | Интервал сетки).
   - Перемещение элемента по полю проекта выполняется в режиме Drag&Drop.
   - Чтобы получить копию любого элемента схемы, надо выделить его и использовать режим Drag&Drop с нажатой клавишей <Ctrl>.
   - Установить режим "резиновой нити" **Options** | **Rubberbanding** (Опции | Резиновая нить).
   - Для соединения элементов выделить соединяемый контакт и, удерживая левую кнопку мыши, протащить курсор до следующего присоединяемого контакта. Для соединения элементов через шину нужно выбрать жирную линию в контекстном меню, отметив пункт меню **Line Style** (Стиль линии), и повторить ранее описанные действия. Для удаления соединения надо выделить его и нажать клавишу <Delete>.
   - Для правильного функционирования устройства необходимо наличие входных и выходных контактов. Чтобы их внести в схему, нужно в окне **Enter Symbol** в поле имени элемента ввести `input(output)`.

- Каждый вход и выход схемы должен быть поименован. Двойной щелчок в области имени контакта позволяет задать ему новое имя.
- Сохранить файл схемы (**File** | **Save as** (Файл | Сохранить как)).
- В дальнейшем понадобится использовать схему полусумматора как символ. Чтобы создать символ, надо выбрать команду меню **File** | **Create Default Symbol** (Файл | Создать символ, заданный по умолчанию). В результате будет автоматически создан файл символа с расширением sym.
- Закрыть файл **File** | **Close**.

3. Компиляция проекта.
   - Открыть окно компилятора командой **Max+Plus II** | **Compiler**.
   - Выбрать семейство **PLD** (по умолчанию **Max+Plus II**) и конкретное устройство в этом семействе для данного проекта, используя команду меню **Assign** | **Device** (Назначить | Устройство). Можно указать режим **AUTO** для выбора конкретного устройства, заданного по умолчанию.
   - Нажать кнопку **Start** для запуска компилятора. В процессе работы компилятор открывает окно процессора сообщений **Message Processor**, где можно просмотреть обнаруженные ошибки. Для получения информации об ошибках следует выбрать кнопку **Help on Message** в окне процессора сообщений.
   - В случае обнаружения ошибок вернитесь в Graphic Editor для внесения изменений в схему и проведите компиляцию снова.
   - Окно компилятора закрывается как стандартное окно Windows.
4. Задание входных воздействий.
   - Выбрать команду меню **File** | **New** и **Waveform Editor** для создания файла входных временных диаграмм (scf).
   - Задать время моделирования в меню **File** | **End Time** (Файл | Время завершения).
   - Выбрать имена входов, на которые должны быть поданы сигналы. Для этого в окне, открытом командой **Node** | **Enter Nodes from SNF** (Узел | Ввод узлов), нажать кнопку **List** и выбрать из списка входов (**I**) и выходов (**O**) нужные. Они будут перенесены в создаваемый файл Waveform-редактора, при этом на входах по умолчанию будет установлен "0", а на выходах "X" — неопределенный уровень сигнала. После закрытия окна **Enter Nodes from SNF** в области **Name** (Имя) редактора Waveform появятся имена выбранных входов и выходов, в области **Value** (Величина) — установленные по умолчанию уровни сигналов.
   - Задать размер сетки для удобства просмотра диаграмм в меню **Options** | **Grid Size** (Опции | Размер сетки).

- Задать форму сигнала для выбранного узла. Это можно сделать различными способами:
  - ◊ постоянный уровень сигнала задается нажатием на соответствующую кнопку (**0** или **1**) на панели инструментов приложения;
  - ◊ периодический сигнал задается нажатием на клавишу с часами на панели инструментов. В открывшемся диалоговом окне устанавливается начальный уровень сигнала и длительность импульса (коэффициент **Multiplied by** умножается на установленный размер сетки);
  - ◊ сигнал произвольной формы задается вручную. Для этого мышью выделяется временной интервал и для него устанавливается необходимый уровень сигнала;
  - ◊ сигнал может быть инвертирован с помощью кнопки на панели инструментов.
- Задать сигналы на входах X и Y в соответствии с рис. 6.10. Сохранить файл с временными диаграммами с именем текущего проекта. Сравнить полученные сигналы на выходах с рисунком.

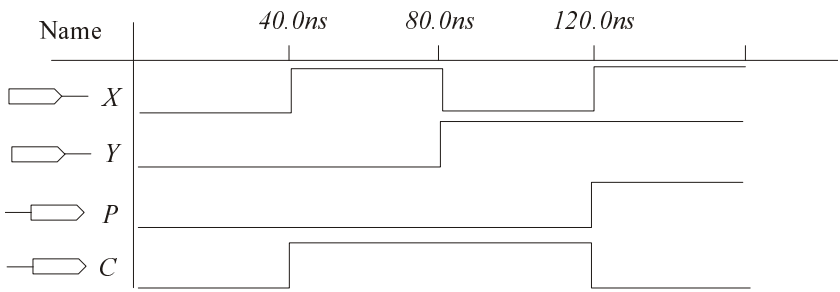


Рис. 6.10. Временные диаграммы работы одноразрядного полусумматора

5. Моделирование работы схемы.
   - Открыть окно моделирования командой **Max+Plus II | Simulator**. При открытии автоматически создается файл с расширением snt.
   - Нажать кнопку **Start** для начала моделирования. Результатом моделирования будут временные диаграммы, записанные в ранее созданный редактором Waveform файл с расширением scf.
   - В случае неудачного завершения моделирования список обнаруженных ошибок и сообщений можно посмотреть в окне процессора сообщений **Message Processor**.

- Для просмотра полученных временных диаграмм нажать кнопку **Open SCF** (Открыть файл временных диаграмм) в окне моделирования.

6. Создание схемы сумматора на основе полусумматора (рис. 6.11).
   - Повторить алгоритм создания схемы, используя созданный символ полусумматора, присвоив ей имя проекта.
   - Проверить правильность работы устройства по описанному ранее алгоритму, используя временные диаграммы.

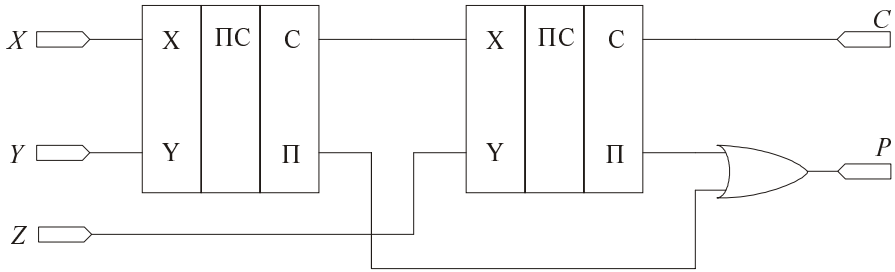
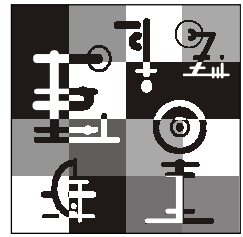


Рис. 6.11. Схема одноразрядного сумматора с использованием полусумматоров



Глава 7

Переходные процессы в комбинационных логических устройствах

В цифровой системе, как и в любой другой (социальной, физической, химической), при каждом переключении (воздействии на нее) из-за наличия задержек ЛЭ на выходе возникают неуправляемые изменения, которые в конечном счете затухают, но для прихода в предсказуемое состояние требуется некоторое время. Такие изменения называют рисками. Их надо учитывать при проектировании устройств ЦТ. Факт наличия рисков используют для создания устройств ЦТ: формирователей длительности импульсов и генераторов импульсов.

7.1. Статические и динамические риски

Рассмотрим схему КЛУ, на выходе " Y " которой будем иметь постоянный логический ноль "0" при любом значении входного сигнала " X " ("0" или "1").

Схема описывается следующей логической функцией: $Y = X * X!$ и выглядит так, как показано на рис. 7.1.

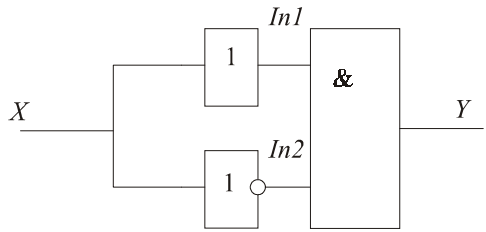


Рис. 7.1. Логическая схема

Опишем работу схемы временными диаграммами без учета задержек повторителя и инвертора (рис. 7.2) и временными диаграммами с учетом не равных друг другу задержек повторителя (*Тзг*) и инвертора (*Тзi*), при условии, что *Тзi* > *Тзг* (рис. 7.3).

Получившийся во втором случае единичный импульс называют *статическим "0"-риском*. Он возникает на фронте импульса " X ".

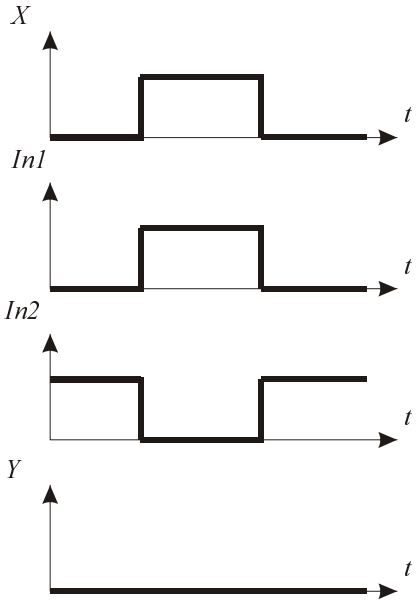


Рис. 7.2. Временные диаграммы без учета задержек

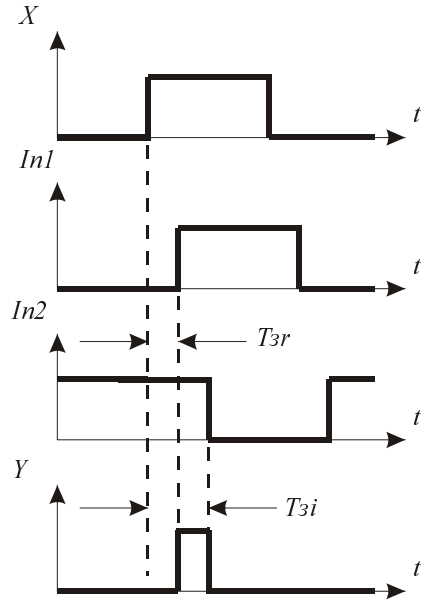


Рис. 7.3. Временные диаграммы с учетом не равных друг другу задержек

При $Tзi < Tзr$ статический "0"-риск появится на спаде импульса " X ".

Статический "1"-риск возникает на выходе в схеме с неизменным единичным сигналом ($Y = (X * X!)!$).

После переходного процесса риски исчезают, но при работе совместно с ПЛУ риски воспримутся элементами памяти ПЛУ, что изменит алгоритм работы всего ЛУ в целом.

Если вместо одноактного переключения сигнала на выходе КЛУ имеем несколько переключений, и в результате на выходе устанавливается уровень сигнала ("0" или "1"), соответствующий логической функции КЛУ, то такую ситуацию называют *динамическим риском*. Эти переключения (почти "пила") не воспринимаются элементами памяти ПЛУ, они не так страшны, как статические риски.

7.2. Формирователи длительности импульсов

Для увеличения длительности импульса " X " на величину задержки (delay) $Tз = Td$ используют схему, изображенную на рис. 7.4. Ее временная диаграмма показана на рис. 7.5.

Для уменьшения длительности импульса применяют схему, представленную на рис. 7.6. Получившаяся временная диаграмма показана на рис. 7.7.

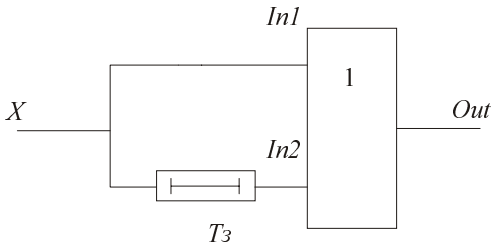


Рис. 7.4. Схема увеличения длительности импульсов

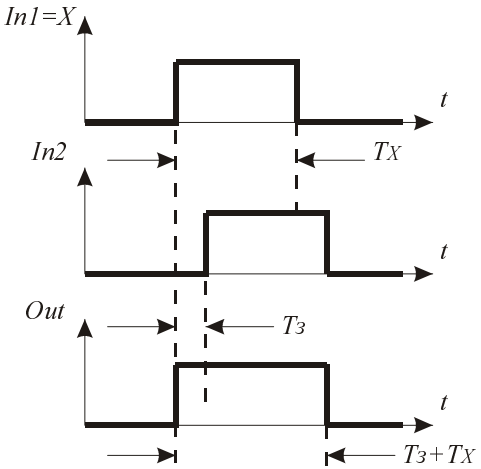


Рис. 7.5. Временная диаграмма схемы увеличения длительности импульсов

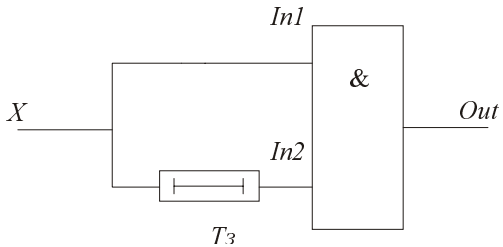


Рис. 7.6. Схема уменьшения длительности импульсов

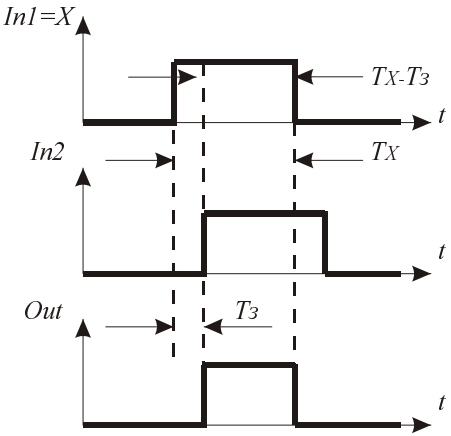


Рис. 7.7. Временная диаграмма схемы уменьшения длительности импульсов

В качестве элементов задержки используют цепочку инверторов или повторителей. Величина задержки регулируется изменением числа элементов цепочки.

7.3. Генераторы симметричных и несимметричных импульсов

Рассмотрим схему генератора симметричных импульсов (рис. 7.8 и 7.9).

Здесь *Тз* — время задержки двух инверторов.

Схема генератора несимметричных импульсов выглядит так, как показано на рис. 7.10.

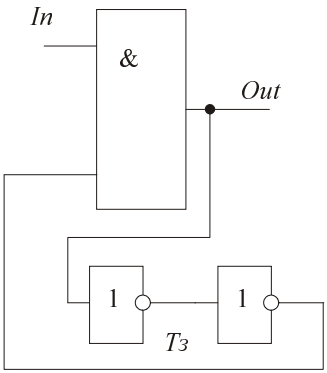


Рис. 7.8. Схема генератора симметричных импульсов

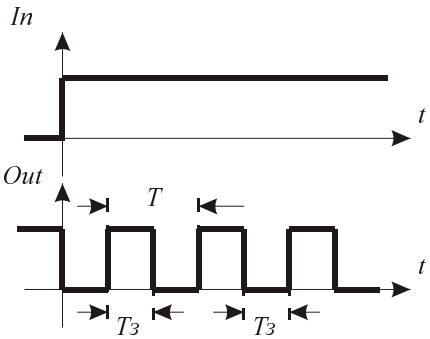


Рис. 7.9. Временная диаграмма схемы генератора симметричных импульсов

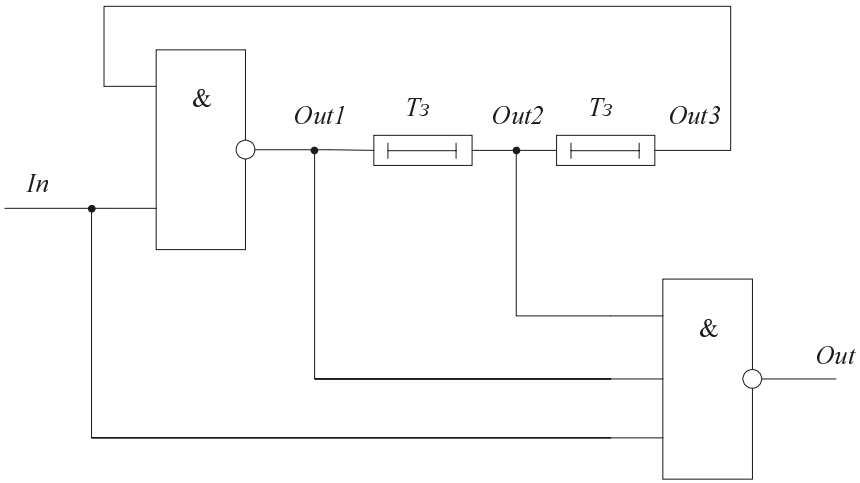


Рис. 7.10. Схема генератора несимметричных импульсов

Здесь нулевой уровень импульса на выходе *Out* будет меньшей длительности, чем единичный уровень, так как для нуля на выходе требуется наличие сразу трех "1" на входах элемента "И" с тремя входами, а это возможно только после срабатывания всех задержек.

Единичный же уровень на выходе появится сразу после прихода на любой из входов "0" (необязательно ждать срабатывания всех задержек).

В таких генераторах допустимы отклонения в частоте около нескольких процентов.

Для получения стабильной частоты с отклонениями 10^{-5} и меньше используют кварцевые генераторы.

Лабораторная работа. Исследование появления рисков. Генераторы и формирователи импульсов

Цель работы

В лабораторной работе исследуются схемы, основанные на использовании задержек. В приведенных примерах в качестве задержек выступают такие элементы, как инвертор и повторитель сигнала.

Основные задачи данной работы — это изучение функционирования приведенных схем генераторов и формирователей импульсов. Необходимо создать проекты для соответствующих схем, получить временные диаграммы выходных сигналов, а также пояснить работу каждой исследуемой схемы.

Программа работы

1. Исследовать работу элемента LCELL библиотеки примитивов САПР фирмы Альтера Max+Plus II.
2. Исследовать процесс появления рисков в схеме устройства. Создать в графическом редакторе САПР фирмы Альтера Max+Plus II проект схемы, приведенной на рис. 7.11, откомпилировать и промоделировать его работу. Зарисовать временные диаграммы. Пояснить полученные результаты.
3. Создать в графическом редакторе проект схемы простейшего генератора импульсов, откомпилировать и промоделировать его работу. Зарисовать временные диаграммы. Схема генератора приведена на рис. 7.12.

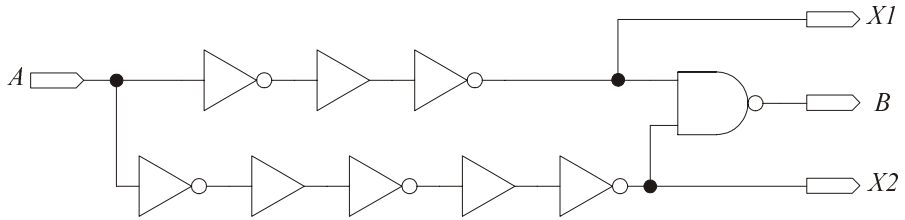


Рис. 7.11. Схема, демонстрирующая появление рисков

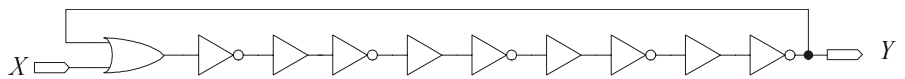


Рис. 7.12. Схема генератора импульсов

4. Создать в графическом редакторе проект схемы простейшего формирователя импульсов (рис. 7.13), откомпилировать и промоделировать работу этой схемы. Зарисовать временные диаграммы.

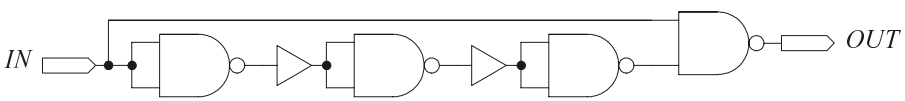


Рис. 7.13. Схема простейшего формирователя импульсов

5. Создать в графическом редакторе проект схемы формирователя импульсов с парафазным выходом (рис. 7.14), откомпилировать и промоделировать его работу. Зарисовать временные диаграммы.

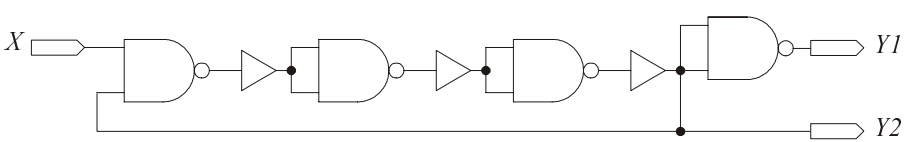


Рис. 7.14. Схема формирователя импульсов с парафазным выходом

6. Создать в графическом редакторе проекты схем формирователей импульсов с выделением фронта и среза входного импульса, откомпилировать и промоделировать их работу. Зарисовать временные диаграммы. Схемы формирователей приведены на рис. 7.15 и 7.16.

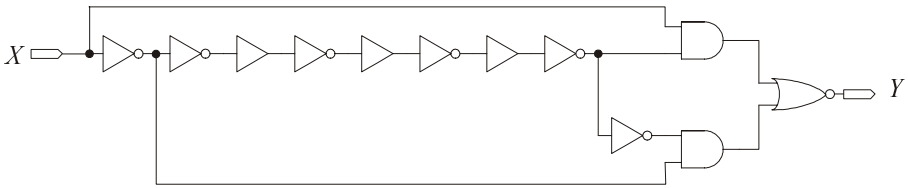


Рис. 7.15. Схема формирователя импульсов с выделением фронта и среза

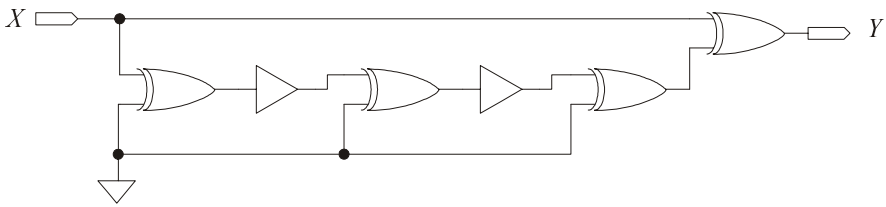
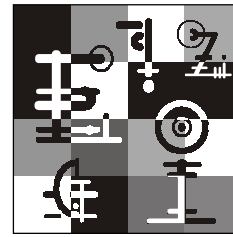


Рис. 7.16. Схема формирователя импульсов на элементах XOR с выделением фронта и среза

7. В отчете привести все исследуемые схемы с пояснением работы каждого устройства и временные диаграммы, полученные в результате моделирования.



Глава 8

Запоминающие элементы логических устройств

ЗЭ ЛУ могут быть построены на основе любой из ранее описанных логик: ТТЛ, ЭСЛ или К-МОП. Они принадлежат к последовательностным логическим устройствам (ПЛУ) или автоматам с памятью.

8.1. Описание функционирования ЗЭ ЛУ (спецификация простейшего ЗЭ ЛУ)

В логическом элементе (ЛЭ) "инвертор" выход сразу же реагирует на изменение сигнала на входе. ЗЭ должен вести себя не так. Если на вход ЗЭ подали сигнал установки "1", а затем перешли к стадии хранения информации в ЗЭ (то есть туда же подали сигнал перехода к стадии хранения), то на выходе единичная информация должна поддерживаться сколь угодно долго, несмотря на то, что на вход подается комбинация хранения, в отличие от ЛЭ ЛУ.

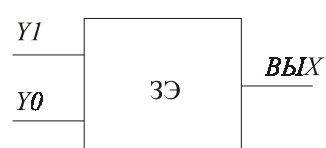


Рис. 8.1. Запоминающий элемент

ЗЭ изображен на рис. 8.1 ($Y1$ — вход установки или запоминания на выходе "1", $Y0$ — вход запоминания "0").

Временные диаграммы функционирования стандартного ЗЭ ЛУ представлены на рис. 8.2.

Здесь "*Хр*" — стадия хранения, "*Ус*0" — стадия установки "0", "*Ус*1" — стадия установки "1".

При неопределенности ЗЭ может растеряться, как и мы, когда нас дергают в разные стороны.

Электромеханическим аналогом работы ЗЭ является работа выключателя.

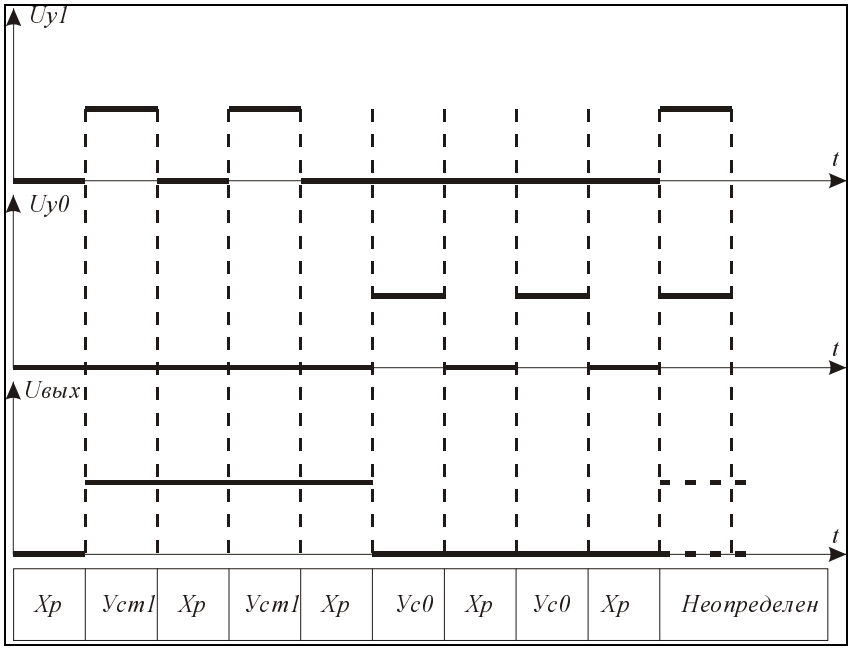


Рис. 8.2. Временные диаграммы ЗЭ

8.2. Внутренняя структура ЗЭ

Ответим на **вопрос**: за счет какого схемного усовершенствования возникает стабильное хранение "1" или "0"?

Для выключателя стабильность состояний задается внутренней связью ("пружинкой").

Для ЗЭ ЛУ стабильность состояний хранения "1" или "0" задается обратной связью (ОС).

Рассмотрим следующую схему с ОС (рис. 8.3).

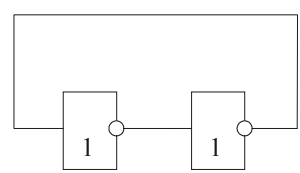


Рис. 8.3. Бистабильная ячейка из двух инверторов

Бистабильная ячейка имеет два устойчивых состояния.

Докажем это. Соответствующая электрическая схема выглядит так, как показано на рис. 8.4.

Здесь $U2 = U3$ и $U4 = U1$, этим равенствам соответствуют следующие точки пересечения ХПН инверторов — рис. 8.5.

Получили три точки пересечения или равновесия (средняя — неустойчивая, крайние — устойчивые), *но* пока имеем входы только для сигналов ОС, поэтому это еще не ЗЭ, так как нет входов установки "0" и "1".

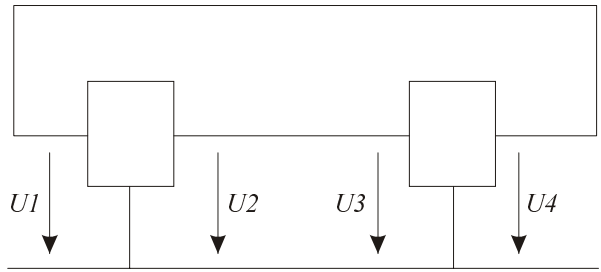


Рис. 8.4. Электрическая схема бистабильной ячейки

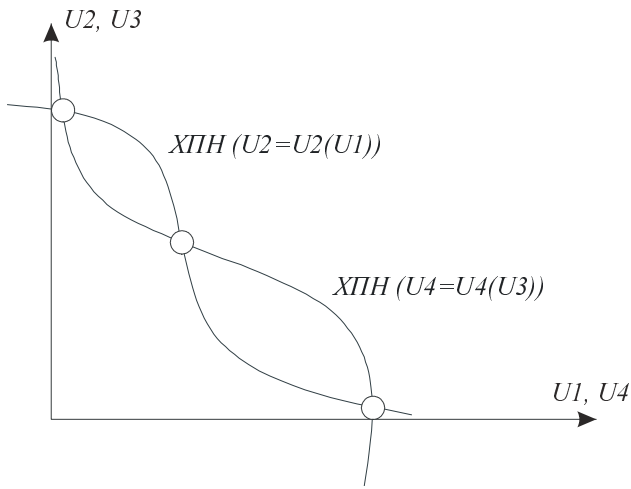


Рис. 8.5. Характеристика передачи напряжения бистабильной ячейки

8.2.1. RS-триггер на "ИЛИ-НЕ"

Объединим в едином ЗЭ входы установки "0" и "1" и его внутренние сигналы (сигналы ОС).

Если использовать элементы "ИЛИ-НЕ", то имеем рис. 8.6.

Для превращения этой схемы в схему бистабильной ячейки, хранящей информацию, на дополнительные входы нужно подать "нули", так как при этом из элементов "ИЛИ-НЕ" получаются инверторы. Поэтому комбинацией хранения в данном случае (случае с "ИЛИ-НЕ") являются два "нуля".

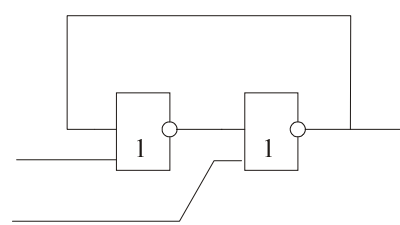


Рис. 8.6. RS-триггер

Обычно эту схему рисуют симметрично (на рис. 8.7 представлен временной, событийный процесс переключения триггера из состояния "0" в состояние "1" за время от $t0$ до $t4$, он связан с наличием задержек "ИЛИ-НЕ").

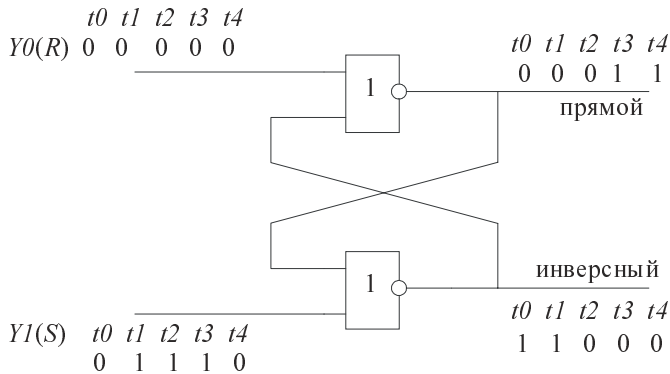


Рис. 8.7. RS-триггер с временным событийным процессом переключения

Сначала выберем прямой и инверсный выходы ЗЭ, так как по сигналу на прямом выходе судят о состоянии ЗЭ: единичное оно или нулевое. Пусть верхний выход схемы будет прямым выходом, а нижний — инверсным.

ЗЭ имеет две стадии работы:

- ❒ спокойная стадия или стадия хранения на выходе 0 или 1;
- ❒ стадия переключения:
  - из состояния хранения "0" в состояние хранения "1";
  - из состояния хранения "1" в состояние хранения "0" .

Рассмотрим комбинации сигналов, которые необходимо подавать на входы ЗЭ (S и R) для обеспечения работы на этих двух стадиях.

Для ЗЭ на "ИЛИ-НЕ" эти комбинации выглядят следующим образом:

- ❒ хранения: R — 0, S — 0;
- ❒ переключения из состояния хранения "0" в состояние хранения "1": R — 0, S — 1;
- ❒ переключения из состояния хранения "1" в состояние хранения "0": R — 1, S — 0.

Определим входы ЗЭ на "ИЛИ-НЕ": один вход назовем входом установки "1" ($Y1$), при подаче на него единичного сигнала, ЗЭ переходит в единичное состояние;

другой вход назовем входом установки "0" ($Y0$), при подаче на него единичного сигнала, ЗЭ переходит в нулевое состояние.

Обозначение ЗЭ по ГОСТ представлено на рис. 8.8, здесь S (set) — установка "1", R (reset) — сброс "1", T — триггер.

Этот ЗЭ называют RS-триггером.

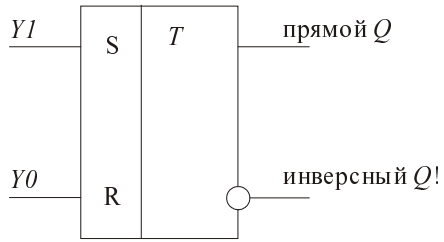


Рис. 8.8. УГО запоминающего элемента.

8.2.2. RS-триггер на "И-НЕ"

RS-триггер может быть построен на элементах "И-НЕ", тогда логическая схема выглядит так, как показано на рис. 8.9.

Комбинации сигналов, необходимые для обеспечения работы:

- хранения: R — 1, S — 1;
- переключения из состояния хранения "0" в состояние хранения "1": R — 1, S — 0;
- переключения из состояния хранения "1" в состояние хранения "0": R — 0, S — 1.

Тот факт, что на установку и сброс подается "0" (нулевой) сигнал (а не единичный), в ГОСТ отображается инверсией (кружком) на входе.

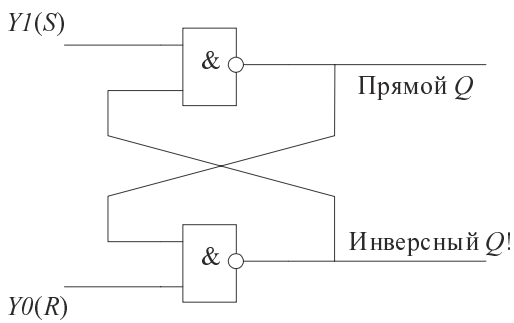


Рис. 8.9. Триггер на элементах "И-НЕ"

8.3. D-триггеры

Для понимания использования RS-триггера рассмотрим часть схемы накапливающего или комбинационного сумматора последовательного действия (рис. 8.10). Здесь нужно помнить предыдущий перенос $Pк-1$ при присутствующем на выходе сумматора следующем переносе $Pк$.

Вопрос: как подключить вместо ЗЭ RS-триггер на "ИЛИ-НЕ"?

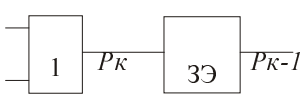


Рис. 8.10. Часть схемы накапливающего сумматора

Решение

Заменим ЗЭ RS-триггером на "ИЛИ-НЕ" с инвертором на входе R (рис. 8.11). Входом такого ЗЭ будут объединенные входы инвертора и входа S триггера. Если на вход ($Pк$) такого ЗЭ придет "1", то на выходе ($Pк-1$) будем иметь "1", если на вход ($Pк$) такого ЗЭ придет "0", то на выходе ($Pк-1$) будем иметь "0".

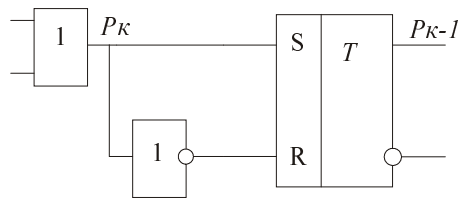


Рис. 8.11. Подключение RS-триггера в качестве ЗЭ

В данном случае триггер не работает как ЗЭ. Он работает как повторитель.

Для одновременного хранения предыдущего и следующего переносов к схеме через пару элементов "И" подключают синхронизирующие импульсы (СИ), которые разрешают считывание с элемента "ИЛИ" только в определенное время.

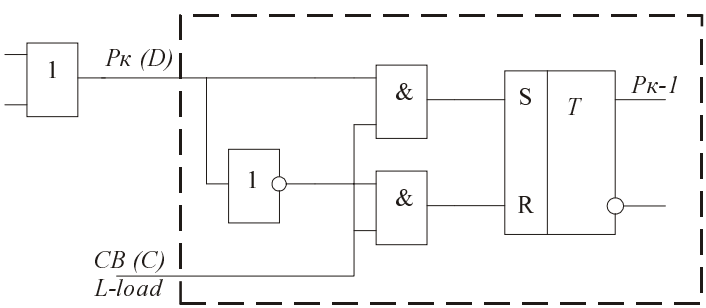


Рис. 8.12. Схема с защелкой

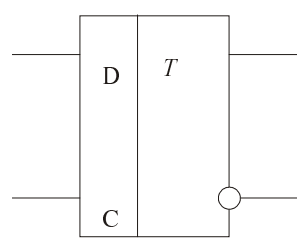


Рис. 8.13. УГО D-триггера

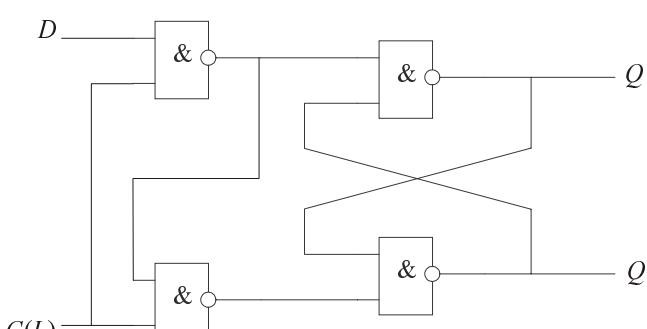


Рис. 8.14. Схема D-триггера в базисе "И-НЕ"

Эту пару ЛЭ "И" называют: семафором или защелкой.

На рис. 8.12 представлена схема, в состав которой входят инвертор, пара ЛЭ "И" и RS-триггер на "ИЛИ-НЕ".

Обведенный участок схемы называется D-триггером (рис. 8.13).

Схема D-триггера в базисе "И-НЕ" (реализованная на элементах "И-НЕ") показана на рис. 8.14.

8.3.1. DV-триггер

Рассмотрим DV-триггер (рис. 8.15 и 8.16). Здесь V-вход дополнительного управления (если V = 0, то входы отключены; если V = 1, то DV-триггер ведет себя как D-триггер).

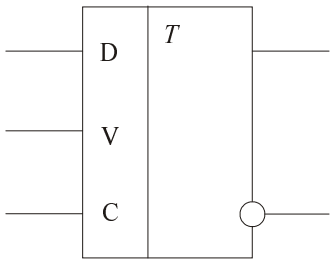


Рис. 8.15. УГО DV-триггера

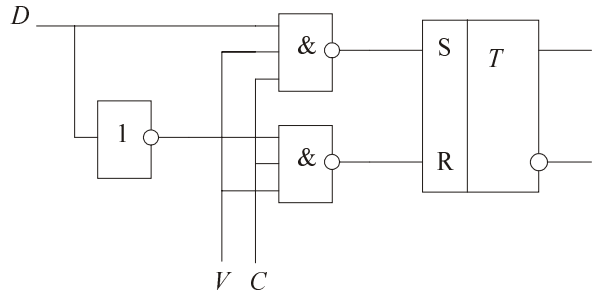


Рис. 8.16. Схема DV-триггера

8.3.2. Двухтактный (двухфазный) D-триггер

Если необходимо одновременно хранить два вида информации (например: предыдущий перенос и новое значение переноса при сложении), то используют двухтактный или двухфазный D-триггер, состоящий из двух D-триггеров (рис. 8.17 и 8.18).

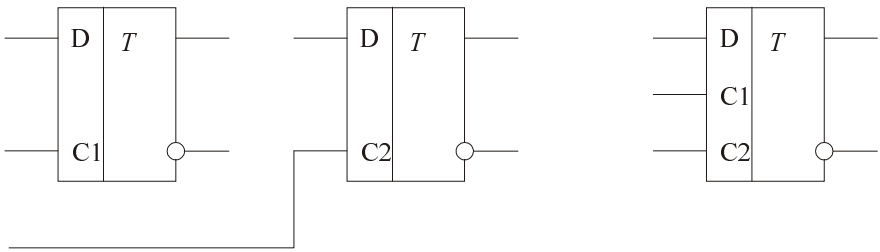


Рис. 8.17. Двухтактный (двухфазный) D-триггер

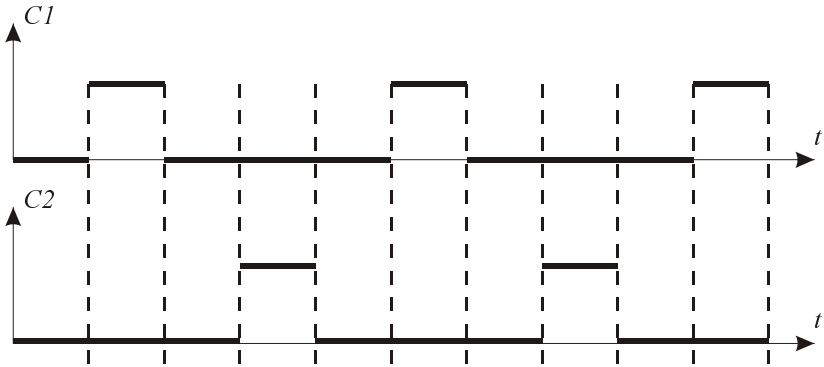


Рис. 8.18. Временная диаграмма, тактовые импульсы *C1* и *C2* сдвинуты друг относительно друга во времени

8.4. Начальные сведения об описании ЛУ на языке VHDL

Название языка VHDL расшифровывается как: VHSIC (Very High-Speed Integrated Circuits) HDL (Hardware Description Languages) (Язык описания аппаратуры очень высокоскоростных интегральных схем).

В данной главе дается пояснение к проектам RS-триггера на VHDL, представленным далее в лабораторной работе. В этой работе рассматриваются два варианта описания простейших логических устройств с использованием языков программирования (в нашем случае VHDL).

Первый вариант — создание схемы, когда разработка ведется, начиная с мельчайших элементов схемы (компонентов), которые в дальнейшем соединяются в схему более высокого уровня иерархии (*структурный подход*).

Второй вариант — описание поведения устройства в целом в зависимости от значений сигналов на его входах (*поведенческий подход*).

Программа (проект) на языке VHDL состоит из нескольких блоков.

- Для подключения библиотечных функций и описания типов переменных языка используются операторы:

  `LIBRARY <имя библиотеки>;`

  `USE <имя модуля в библиотеке>;`

- Для описания входов и выходов устройства применяется блок:

  `ENTITY <имя проекта> … END <имя проекта>;`

- Для описания модели функционирования устройства во времени для различных комбинаций значений входных сигналов используют блок:

  `ARCHITECTURE <имя модели поведения> OF <имя проекта> IS BEGIN … END <имя модели поведения>;`

  Модели могут быть двух типов: поведенческие и структурные, что соответствует двум подходам к описанию схем, описанным ранее.

- Следующий блок используется в случае структурного подхода и устанавливает соответствие компонента и реального проекта, полностью описывающего этот компонент:

  `CONFIGURATION <имя конфигурации> OF <имя проекта> IS … END <имя конфигурации>;`

Основными операторами языка являются:

- оператор присваивания:

  `c <= a;`

- оператор сравнения:

  `IF … THEN … ELSE …END;`

- оператор цикла:

  `Метка: FOR <переменная цикла> IN <порядок> LOOP … END LOOP метка.`

Могут быть использованы следующие логические операции: `AND`, `OR`, `NOT`.

Операторы языка VHDL отделяются друг от друга точкой с запятой.

Кроме константы и переменной в VHDL введено понятие *сигнал*, ему соответствует соединительный провод в реальном устройстве. Сигнал в VHDL можно задерживать на заданное время так же, как реальный сигнал.

Для возможности описания параллельной во времени работы отдельных узлов сложного ЛУ в VHDL введен оператор `process`: внутри него команды на VHDL выполняются последовательно, если его нет, то все команды выполнятся параллельно (одновременно).

Более полное описание языка VHDL представлено в *приложении 1*.

В дальнейшем, при рассмотрении сложных устройств ЦТ (например, процессоров), изучение VHDL будет продолжено.

8.4.1. Синтезируемость кода на языке VHDL

Под синтезируемостью VHDL-кода понимается возможность перевода VHDL-проекта в схему, состоящую из логических элементов.

Для перехода к синтезируемости кода на VHDL рассмотрим основные этапы, которые включает проектирование с использованием языков описания аппаратуры и реализации их на ПЛИС:

1. Создание алгоритмического HDL-описания проекта.
2. Моделирование проекта (функциональное тестирование).
3. Автоматический синтез логической схемы.
4. Физическое проектирование.
5. Моделирование логической схемы вентильного уровня с учетом задержек сигналов в проводах (временная симуляция).

Современные системы автоматического проектирования состоят из нескольких подсистем, автоматизирующих выполнение этапов проектирования цифровых систем.

- Редакторы ввода проектов.
  - Текстовые редакторы — предназначены для создания поведенческого или иерархического функционально-структурного описания проекта на HDL.
  - Графические редакторы — для ввода проектов в виде иерархических блок-схем, логических схем, автоматов состояний. Некоторые САПР автоматически генерируют HDL-описание по графическому представлению.
- Симуляторы — служат для тестирования проектов на соответствие спецификации.
- Средства автоматического синтеза (синтезаторы) — переводят поведенческое описание проекта в логическую схему, которая может быть представлена RTL или вентильным уровнем в технологическом базисе выбранной микросхемы.
  - Register transfer level (RTL) — уровень описания, в котором поведение проекта явно описано в терминах передачи данных между запоминающими элементами и комбинационной логикой, которая может представлять любую вычислительную или арифметико-логическую схему.
  - Вентильный уровень (Gate level) — логическая ячейка (из них состоит ПЛИС) может конфигурироваться в набор логических элементов (вентили и более сложные по функциональности ЛЭ), данный набор определяет базис проекта, или библиотеку, которую предоставляет производитель данной микросхемы. В соответствии с этой библиотекой синтезатор строит логическую схему вентильного уровня по поведенческому описанию проекта.
- Средства размещения и трассировки — с их помощью выполняется размещение логической схемы проекта на кристалле, назначение выводов микросхемы входам-выходам, а также создается файл временных задержек для тестирования.
- Программаторы — средства программирования или конфигурирования ПЛИС.

Синтезируемое подмножество языков HDL

На этапе алгоритмического описания проект можно создавать не только на HDL, но и на других языках программирования (например, C, C++, Паскаль), а затем переводить это описание на язык HDL. Необходимо выполнять перевод, так как синтезаторы принимают на вход только HDL-описания, и не все конструкции, поддерживаемые данными языками, могут быть представлены синтезируемым HDL-кодом.

Существуют стандартные правила поддерживаемых HDL-конструкций для всех синтезаторов. В свою очередь производители программ синтеза могут включать свои правила.

Основные синтезируемые конструкции языка VHDL

Поддерживаемые (синтезируемые) конструкции

- Описание объектов: `entity`, `architecture`, `package`, `function` и `procedure`.
- Все библиотеки IEEE, включая:

  `std_logic_1164`
  `std_logic_unsigned`
  `std_logic_signed`
  `std_logic_arith`
  `numeric_std and numeric_bit`
  `standard library package (std)`.
- Описание портов: `in`, `out`, `inout`, `buffer`.
- Описание связей, констант, переменных: `signal`, `constant`, `variable`.
- Типы сигналов и переменных: `integer`, `bit`, `Boolean`, `std_logic`, `std_ulogic`.
- Все операции (`-`, `-`, `*`, `/`, `**`, `mod`, `rem`, `abs`, `not`, `=`, `/=`, `<`, `<=`, `>`, `>=`, `and`, `or`, `nand`, `nor`, `xor`, `xnor`, `sll`, `srl`, `sla`, `sra`, `rol`, `ror`, `&`).
- Последовательные операторы: назначение сигналов и переменных, `wait`, `if`, `case`, `loop`, `for`, `while`, `return`, `null`, `function`.
- Параллельные операторы: назначение сигнала, `process`, `block`, `generate` (`for` и `if`), создание экземпляра компонента, `function`, и вызов процедуры.
- Предопределенные атрибуты: `t'base`, `t'left`, `t'right`, `t'high`, `t'low`, `t'succ`, `t'pred`, `t'val`, `t'pos`, `t'leftof`, `t'rightof`, `integer'image`, `a'left`, `a'right`, `a'high`, `a'low`, `a'range`, `a'reverse_range`, `a'length`, `a'ascending`, `s'stable`, `s'event`.

Неподдерживаемые конструкции

- Типы доступа `access` и файлы `file`.
- Инициализация значений сигналов и портов.
- Тип переменных `real`.

- Нельзя изменять значение переменной в разных процессах `process`.
- Запрещается использовать значения `X`, `Z` в выражениях.

Игнорируемые конструкции

Игнорируются задержки `after`.

Лабораторная работа. Программирование на VHDL в среде Max+Plus II RS- и D-триггеров

Цель работы

Освоение работы в текстовом редакторе и получение начальных навыков программирования на языке VHDL. Одновременно с этим исследуется работа таких устройств, как RS- и D-триггеры. Изучение основ языка проводится на примерах программных моделей RS-триггера. Закреплением полученного учебного материала является самостоятельное написание двух типов моделей для D-триггера.

Программа работы

1. Создать в графическом редакторе проект схемы RS-триггера на элементах "И-НЕ" (элемент `nand2`), откомпилировать и промоделировать его работу. Зарисовать временные диаграммы. Схема RS-триггера приведена на рис. 8.19.

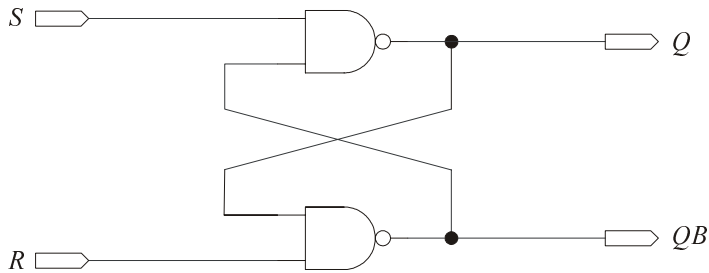


Рис. 8.19. Схема RS-триггера

2. Создать проект схемы RS-триггера в текстовом редакторе, используя язык проектирования схем VHDL.
   - Для этого после выполнения команды меню **File | New** в открывшемся диалоговом окне выбрать текстовый редактор **Text Editor**.

- Создать проект схемы логического элемента "И-НЕ" на языке VHDL. Сохранить файл схемы с расширением vhd. Имя проекта схемы должно совпадать с именем, указанным в блоке `ENTITY`. Текст программы приведен далее:

```
LIBRARY ieee;                   -- подключение библиотеки ieee
USE ieee.std_logic_1164.ALL;    -- использование библиотечного
                                -- модуля, содержащего
                                -- дополнительные типы переменных.
ENTITY notand IS
PORT( a : IN std_logic;         -- описание входов и выходов
      b : IN std_logic;         -- устройства (IN — вход, OUT —
      c : OUT std_logic );      -- выход, INOUT — двунаправленный
END notand;                     -- сигнал)
ARCHITECTURE behavior OF notand IS
BEGIN
    C <= NOT ( a AND b );
END behavior;
```

- Откомпилировать проект и получить временные диаграммы. Сравнить с таблицей истинности элемента "И-НЕ".
- Создать проект схемы RS-триггера на основе логических элементов "И-НЕ", используя язык VHDL. Пример программы, описывающей работу этого триггера, приведен ниже (структурная модель):

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY rstr IS
PORT( s : IN std_logic;
      r : IN std_logic;
      q : INOUT std_logic;
     qb : INOUT std_logic );
END rstr;
ARCHITECTURE behav OF rstr IS
COMPONENT notand    -- описание используемого компонента
PORT( a : IN std_logic;
      b : IN std_logic;
      c : INOUT std_logic);
END COMPONENT;
BEGIN
u1: notand                  -- указание u1, как компонента notand
    PORT MAP ( s, qb, q);   -- указание входов и выхода для u1
```

```
u2: notand
    PORT MAP (q, r, qb);
END behav;
CONFIGURATION con OF rstr IS
    FOR behav
       FOR u1, u2: notand
            USE ENTITY work.notand (behavior); -- определяет интерфейс
END FOR;    -- и модель компонента notand
 END FOR;
END con;
```

Примечание

Обратите внимание, что соединение двух одинаковых частей RS-триггера (`u1` и `u2`), описываемых одним компонентом `notand`, задано в описании карт портов (`Port map`) этих частей `u1` и `u2`.

- Откомпилировать проект и получить временные диаграммы. Сравнить с результатами моделирования, полученными в п. 1.
- Реализовать проект RS-триггера, используя поведенческую модель. Для этого в текстовом редакторе можно выбрать шаблон программной модели триггера (**Full design: Flipflop**) в меню **Templates | VHDL template** и внести соответствующие изменения или набрать текст следующей программы:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY rstr1 IS
PORT( s : IN std_logic;
      r : IN std_logic;
      q : OUT std_logic );
END rstr1;
ARCHITECTURE behav OF rstr1 IS
    SIGNAL qs:std_logic;
BEGIN
PROCESS (s,r)
    BEGIN
        IF s='1' THEN
            IF r='1' THEN qs<=qs;
            ELSE qs<='0';
            END IF;
        ELSE qs<='1';
        END IF;
```

```
END PROCESS;
q<=qs;
END behav;
```

- Откомпилировать проект и получить временные диаграммы. Сравнить с результатами моделирования, полученными в п. 1.

3. Создать в графическом редакторе проект схемы D-триггера на элементах "И-НЕ" (элемент `nand2`), откомпилировать и промоделировать его работу. Зарисовать временные диаграммы. Схема D-триггера приведена на рис. 8.20. В качестве элемента RS-триггера использовать схему, приведенную в п. 1.

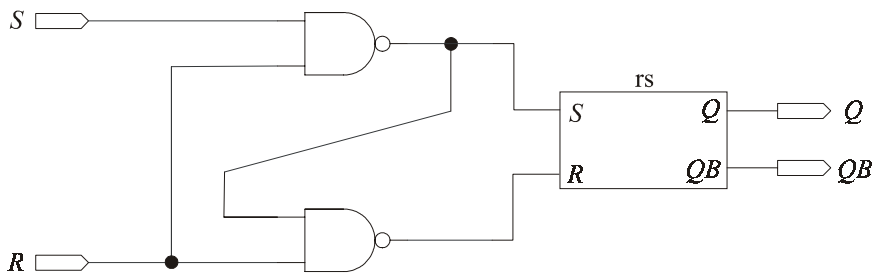
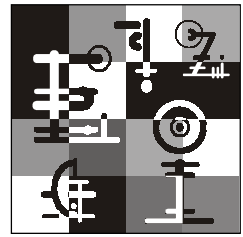


Рис. 8.20. Схема D-триггера

4. Реализовать проект D-триггера на языке программирования VHDL, используя структурную и поведенческую модели функционирования устройства. Откомпилировать проекты и получить временные диаграммы. Сравнить их с диаграммами, полученными в п. 3.



Глава 9

О необходимости тактирования схем ЦУ. Последовательные и параллельные регистры

Рассматриваются риски при работе RS-триггера, а также анализируется во времени работа регистров сдвига, регистра памяти и универсального регистра.

9.1. Переключение RS-триггера на "ИЛИ-НЕ" во времени

Рассмотрим переключение RS-триггера на "ИЛИ-НЕ" (рис. 9.1) из состояния "0" в состояние "1", приняв во внимание, что любой ЛЭ (в том числе и "ИЛИ-НЕ") имеет время задержки распространения сигнала (Tp)

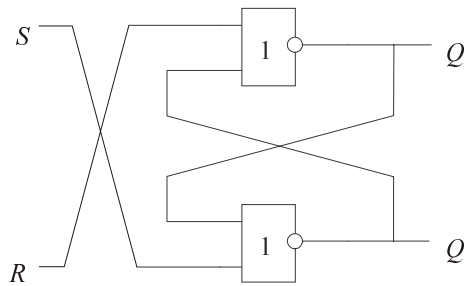


Рис. 9.1. Схема RS-триггера на элементах "ИЛИ-НЕ"

Из временного анализа (рис. 9.2) видно, что в ответ на входной сигнал триггер переключается с задержкой $2*Tp$, поэтому $Tвх$ должно быть больше $(3...4)Tp$.

На отрезке "**!!!???**": $Q = Q!$ — такого быть не должно, и если в этот момент прервать "1"-сигнал на Set, то триггер "растеряется". Ситуация, когда $Q = Q!$, называется *риском*.

Из временного анализа вытекает, что необходимо добиться такой работы триггера, при которой на отрезке времени "**!!!???**" информация последующими элементами не воспринималась бы, так как она *ложная*.

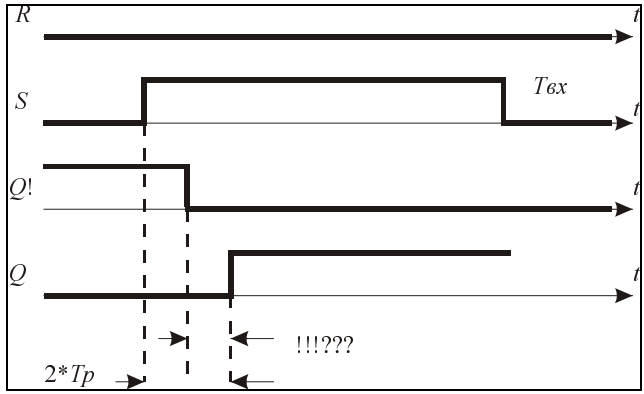


Рис. 9.2. Временная диаграмма RS-триггера

Для устранения возникающих трудностей подачу информации осуществляют от элемента к элементу с помощью синхронизирующих, тактирующих или стробирующих импульсов. В другие отрезки времени считывание информации запрещено.

Таким образом, продвижение информации по регистрам сдвига и все действия в вычислительных устройствах строго тактированы или синхронизированы.

9.2. Системы синхронизации на примере работы регистров сдвига (последовательных регистров)

Каждая из систем синхронизации соответствует различному воздействию импульсов синхронизации на логическое устройство (ЛУ). Различают одно-, двух- и более тактные (фазные) системы синхронизации.

Рассмотрим одно- и двухтактные системы синхронизации на примере соответствующих регистров сдвига. *Регистр сдвига* — это устройство, состоящее из нескольких разрядов, каждый из которых может находиться в одном из двух устойчивых состояний, отвечающих двум значениям логической переменной. Разряд регистра может хранить один бит информации.

Регистры сдвига строятся на одно- и двухтактных D-триггерах.

9.2.1. Схема однотактного последовательного регистра

При передаче чего-либо, не обязательно информации в регистрах, на какое-то короткое время передающее звено (разряд регистра) должно хранить одновременно то, что у него было, и уже вновь поступившее. Поэтому разряд регистра должен содержать минимум два запоминающих элемента (ЗЭ). На рис. 9.3 и 9.4 представлены схемы однотактных последовательных регистров сдвига на однотактных и двухтактных D-триггерах, соответственно.

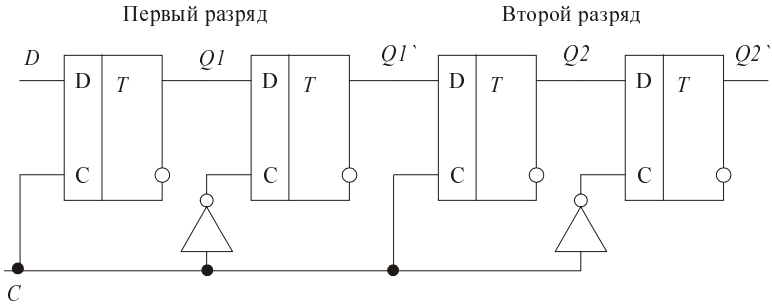


Рис. 9.3. Схема однотактного последовательного регистра на однотактных D-триггерах

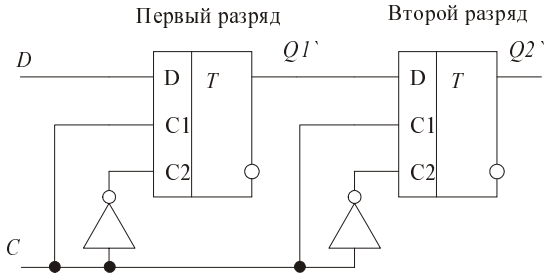


Рис. 9.4. Схема однотактного последовательного регистра на двухтактных D-триггерах

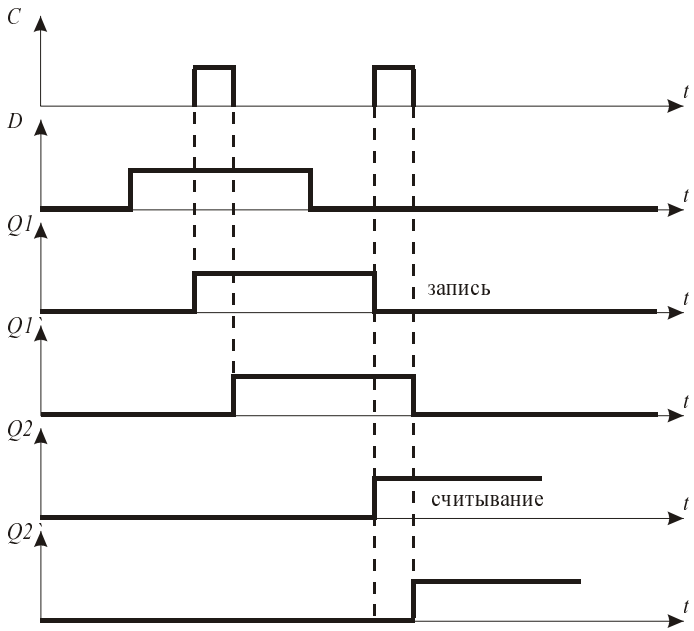


Рис. 9.5. Временная диаграмма работы однотактного последовательного регистра

Рассмотрим работу этого регистра во времени (рис. 9.5).

Главное в регистрах сдвига: обеспечить *одновременность* операций записи и считывания информации в каждый разряд регистра. В однотактных последовательных регистрах сдвига эта *одновременность* обеспечивается за время тактового импульса.

9.2.2. Схема двухтактного последовательного регистра

В двухтактном последовательном регистре сдвига каждый разряд также содержит два ЗЭ, а передача информации от разряда к разряду происходит за два такта. На рис. 9.6 и 9.7 представлены схемы двухтактных последовательных регистров сдвига на однотактных и двухтактных D-триггерах, соответственно.

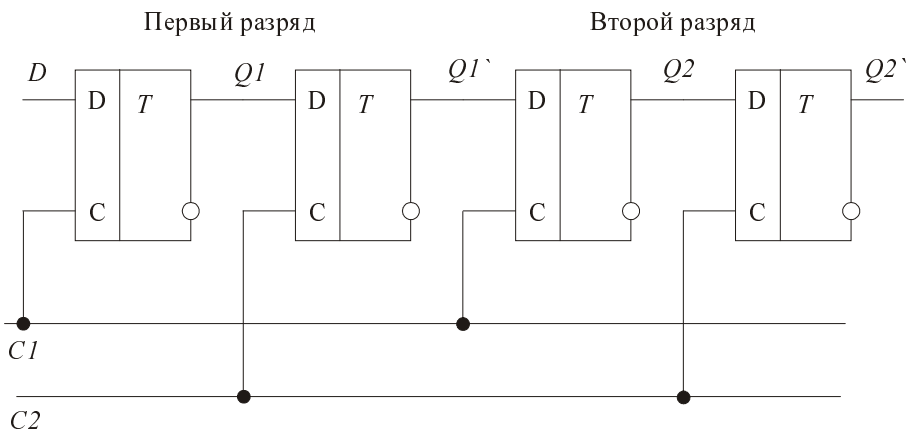


Рис. 9.6. Схема двухтактного последовательного регистра на однотактных D-триггерах

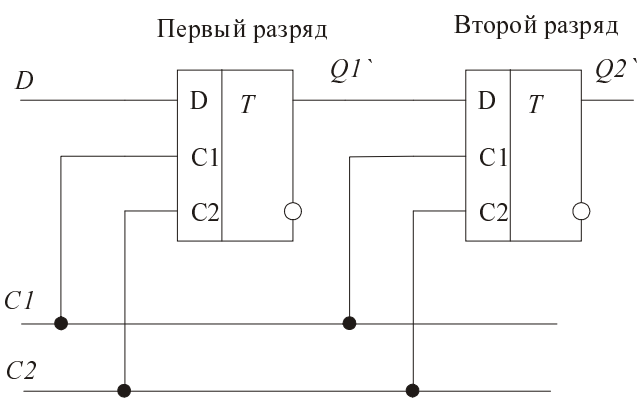


Рис. 9.7. Схема двухтактного последовательного регистра на двухтактных D-триггерах

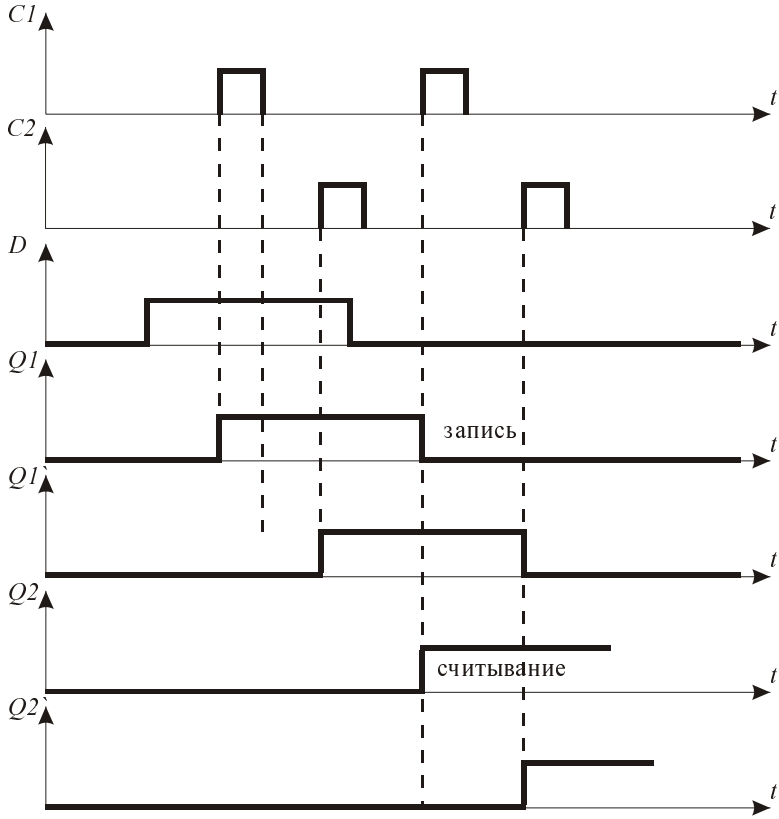


Рис. 9.8. Временная диаграмма работы двухтактного последовательного регистра

Рассмотрим работу этого регистра во времени (рис. 9.8).

Здесь одновременные запись в первый D-триггер и считывание из второго D-триггера для одного и того же разряда разнесены между импульсами $C1$ и $C2$.

В однотактном регистре сдвига, в отличие от двухтактного, одновременные запись и считывание в отдельный разряд регистра происходят за время тактового импульса, поэтому однотактный регистр менее устойчив к помехам (в нем больше вероятности потерять информацию из-за уменьшения длительности тактового импульса под действием помех).

Каждый разряд *многотактного* регистра содержит m последовательно включенных D-триггеров. Управление здесь производится пачками, каждая содержит m импульсов.

Последовательный регистр позволяет осуществлять преобразование последовательного кода в параллельный.

9.3. Параллельные регистры (регистры памяти)

Упрощенная (приблизительная) схема регистра памяти приведена на рис. 9.9.

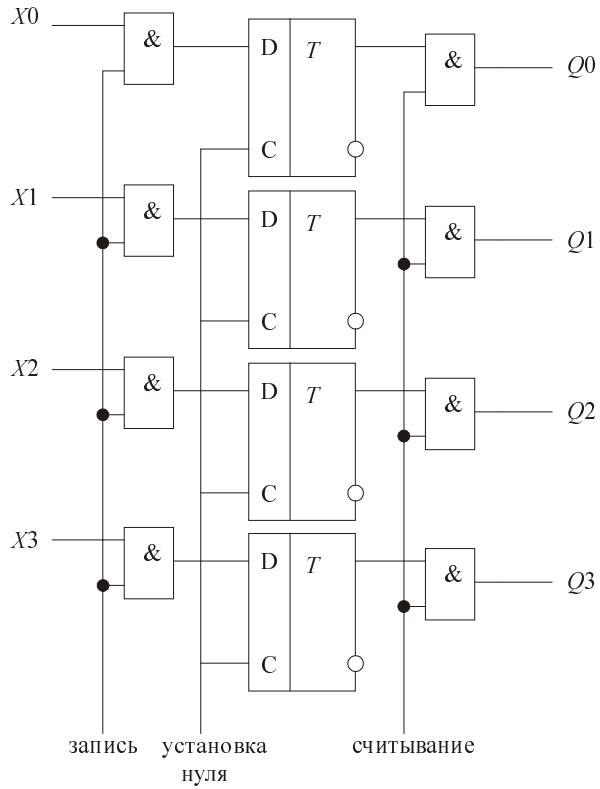


Рис. 9.9. Упрощенная схема регистра памяти

Функциональные возможности: прием числа в параллельном коде (через защелку), хранение и передача числа через защелку на выход.

9.4. Универсальные (параллельные, последовательные и реверсивные) регистры

Универсальные регистры (рис. 9.10) объединяют функциональные возможности последовательных и параллельных регистров. Они также могут получать информацию в параллельном коде и выдавать ее на выход в последовательном коде (преобразовывать параллельный код в последовательный и наоборот).

В этих регистрах возможен реверс (обратный ход) информации с выхода на вход. Реверсивные регистры сдвигают код числа в сторону старшего или младшего разрядов.

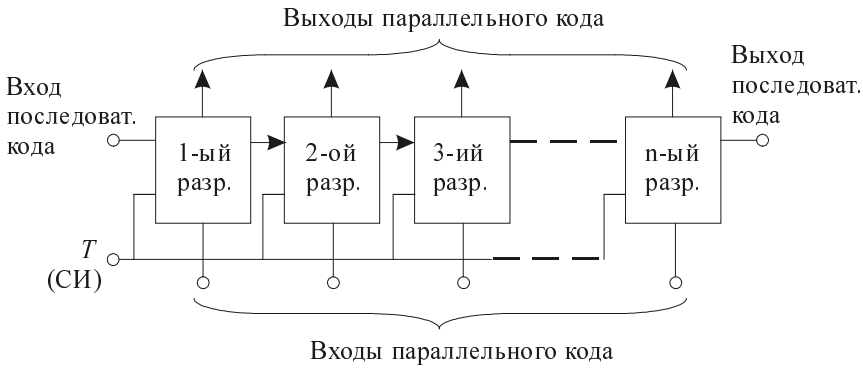


Рис. 9.10. Функциональная схема универсального регистра

Лабораторная работа. Исследование схем сдвигающего регистра, регистра памяти и универсального регистра

Цель работы

В данной лабораторной работе исследуется функционирование различных схем сдвигающих регистров и регистров памяти. Сдвигающие регистры осуществляют прием, хранение и передачу информации, а также сдвиг информации на требуемое число разрядов влево или вправо, преобразование последовательного кода в параллельный (и наоборот). Регистры памяти предназначены для хранения информации. Основу реализации сдвигающих и параллельных регистров составляют D-триггеры (элемент DFF из библиотеки примитивов).

Программа работы

1. Исследовать работу элемента библиотеки примитивов DFF.
2. Создать в графическом редакторе проект схемы двухразрядного сдвигающего регистра на D-триггерах (элемент DFF) с одним тактовым входом, откомпили-

ровать и промоделировать его работу. Зарисовать временные диаграммы. Схема сдвигающего регистра приведена на рис. 9.11.

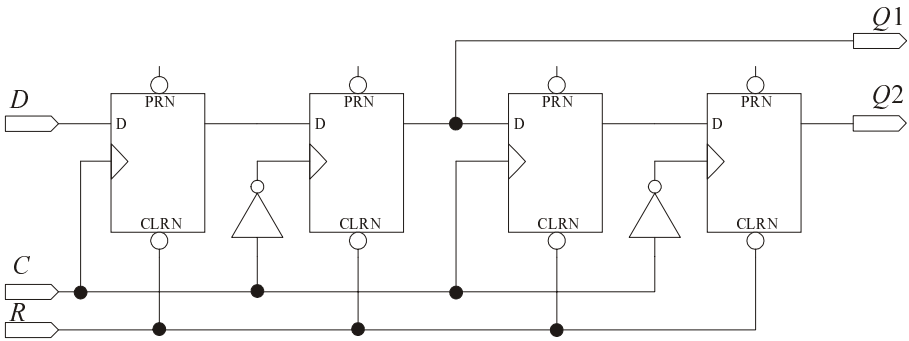


Рис. 9.11. Схема двухразрядного однотактного сдвигающего регистра

3. Создать в графическом редакторе проект схемы D-триггера с двумя тактовыми входами на элементах DFF из библиотеки примитивов, откомпилировать и промоделировать его работу. Зарисовать временные диаграммы. Схема D-триггера приведена на рис. 9.12. Сохранить схему в виде символа.

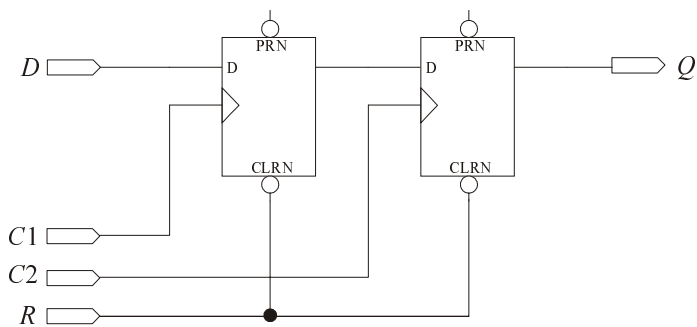


Рис. 9.12. Схема двухтактного D-триггера

4. Создать в графическом редакторе проекты схем двухразрядного сдвигающего регистра на D-триггерах с двумя тактовыми входами, откомпилировать и промоделировать работу этих схем. Зарисовать временные диаграммы. Схема сдвигающего регистра приведена на рис. 9.13.
5. Создать в графическом редакторе проект схемы четырехразрядного регистра памяти на D-триггерах (элемент DFF), откомпилировать и промоделировать его работу. Зарисовать временные диаграммы. Схема регистра памяти приведена на рис. 9.14.

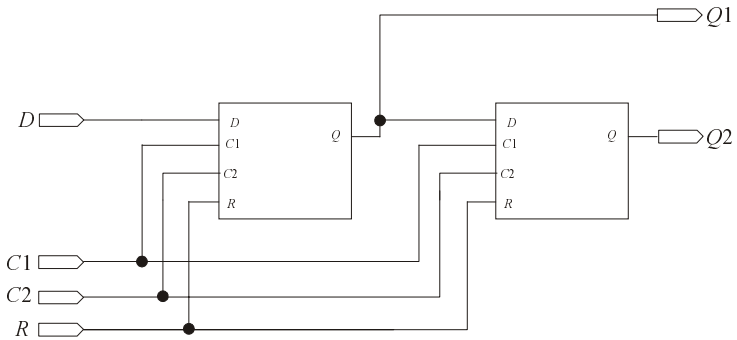


Рис. 9.13. Схема двухтактного двухразрядного сдвигающего регистра

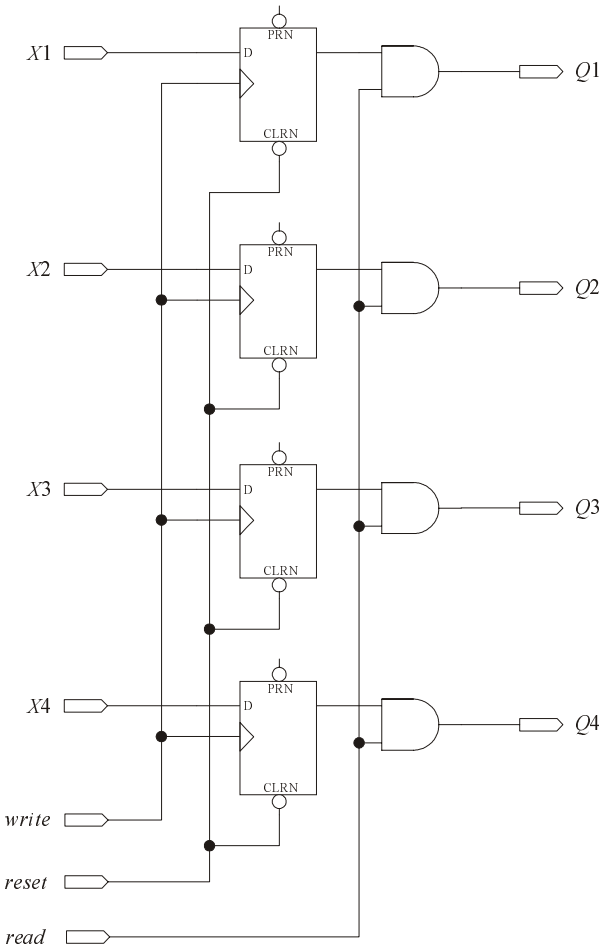


Рис. 9.14. Схема простейшего регистра памяти

6. Исследовать работу элемента 74395 библиотеки макрофункций mf САПР фирмы Альтера Max+Plus II (рис. 9.15).

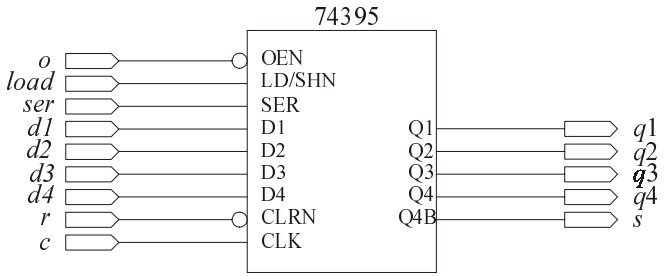


Рис. 9.15. Элемент 74395 (универсальный регистр)

Этот элемент представляет собой универсальный регистр, работающий и в режиме параллельной загрузки данных, и в режиме последовательного ввода и сдвига данных. Входы **d1**, **d2**, **d3**, **d4** используются для параллельного ввода информации, а вход **SER** — для последовательного. Вход **LD/SHN** управляет режимом работы регистра ("0" на этом входе означает, что разрешена последовательная загрузка данных, "1" — параллельная). Вход **CLRN** — установка нуля, вход **CLK** — разрешение записи (запись производится по спаду единичного сигнала).

Откомпилировать проект и получить временные диаграммы работы регистра в разных режимах.

7. Реализовать проект двухразрядного сдвигающего регистра с одним тактовым входом на языке программирования VHDL, используя поведенческую модель функционирования устройства. Откомпилировать проект и получить временные диаграммы. Сравнить их с диаграммами, полученными в п. 2.

8. Реализовать проект двухразрядного сдвигающего регистра с двумя тактовыми входами, основанного на двухтактных D-триггерах (см. рис. 9.13), на языке программирования VHDL, используя поведенческую модель функционирования устройства. Откомпилировать проект и получить временные диаграммы. Сравнить их с диаграммами, полученными в п. 4.

9. Реализовать проект регистра памяти на языке программирования VHDL, используя поведенческую модель функционирования устройства. Откомпилировать проект и получить временные диаграммы. Сравнить их с диаграммами, полученными в п. 5.

10. Для проверки на синтезируемость написанных ранее VHDL-кодов рассмотрим синтезируемые VHDL-коды D-триггера с асинхронным сбросом и установкой

(D-триггер срабатывает только при наличии соответствующего синхронизирующего импульса) и четырехразрядного регистра сдвига с асинхронным сбросом. При рассмотрении будем использовать встроенные средства САПР Quartus II фирмы Altera. Для описания триггеров, или регистров, обычно используются условия наступления события (event) прохождения во времени фронта или среза (спада) синхронизирующего импульса.

Примеры таких условий для VHDL:

- (`clk'event and clk = '1'`) — условие фронта синхроимпульса;
- (`clk'event and clk = '0'`) — условие среза синхроимпульса.

Синтезируемый VHDL-код D-триггера с асинхронным сбросом и установкой будет выглядеть так:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_async is
port (data, clk, reset, preset : in std_logic;
      q : out std_logic);
end dff_async;
architecture behav of dff_async is
begin
process (clk, reset, preset) begin
  if (reset = '0') then
    q <= '0';
  elsif (preset = '0') then
    q <= '1';
  elsif (clk'event and clk = '1') then
    q <= data;
  end if;
end process;
end behav;
```

Для синтеза этого VHDL-кода необходимо выполнить следующие действия.

- Запустить САПР Quartus II фирмы Altera.
- Создать новый проект, выбрав в меню **File** пункт **New Project Wizard**, и нажать кнопку **Next** в открывшемся окне. Ввести имя проекта в поле **What is the name of this project?** и нажать кнопку **Finish**.
- Создать новый файл, выбрав в меню **File** | **New**. В открывшемся окне на закладке **Device Design Files** выберите файл типа VHDL. В результате создастся файл для написания кода на соответствующем языке.

- Набрать или скопировать вышеприведенный VHDL-код D-триггера.
- Сохранить файл с именем dff_async.
- Откомпилировать проект, выбрав в меню **Processing** | **Start Compilation**. В процессе компиляции, в окне **Messages** отображаются текущие сообщения компилятора, если в коде присутствуют ошибки или несинтезируемые конструкции, компилятор выдаст ошибку.
- При успешном завершении компиляции откроется закладка **Compilation Report**, где в разделе **Analysis & Synthesis** можно посмотреть отчет синтезатора, например отчет об использовании ресурсов микросхемы (раздел **Resource Usage Summary**) (рис. 9.16).

	Resource	Usage
1	Total logic elements	1
2	Total combinational functions	0
3	-- Total 4-input functions	0
4	-- Total 3-input functions	0
5	-- Total 2-input functions	0
6	-- Total 1-input functions	0
7	-- Total 0-input functions	0
8	Combinational cells for routing	0
9	Total registers	1
10	I/O pins	5
11	Maximum fan-out node	q~reg0
12	Maximum fan-out	1
13	Total fan-out	5
14	Average fan-out	0.83

Рис. 9.16. Использование ресурсов микросхемы

Как видно из таблицы на рис. 9.16, для построения схемы синтезатор использовал 1 логический элемент (**Total logic elements — 1**), сконфигурированный как регистр (**Total registers —1**) и 5 выводов микросхемы (**I/O pins — 5**).

- Для того чтобы открыть RTL-представление схемы, нужно выбрать пункт меню **Tools** | **RTL Viewer**. Откроется искомая схема (рис. 9.17).
- Для того чтобы открыть представление схемы в технологическом базисе, нужно выбрать пункт меню **Tools** | **Technology Map Viewer**. Откроется искомая схема (рис. 9.18).

Повторите описанные действия, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы для D-триггера с асинхронным сбросом и установкой.

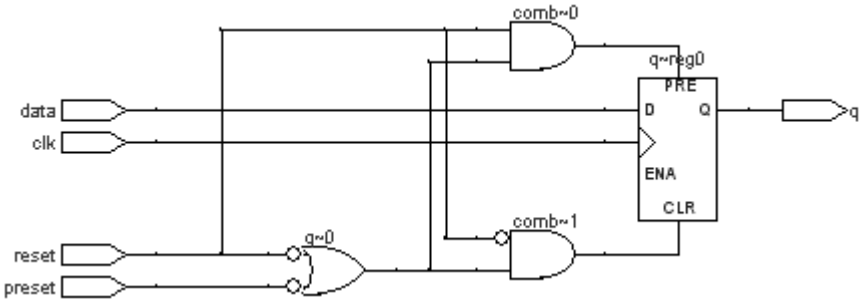


Рис. 9.17. RTL-представление

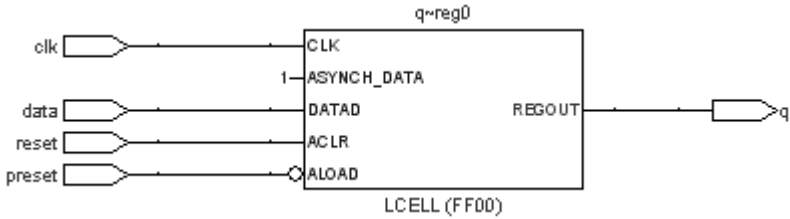


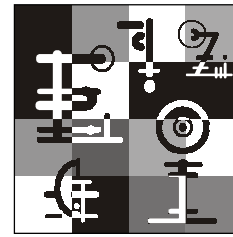
Рис. 9.18. Представление схемы в технологическом базисе

Синтезируемый VHDL-код четырехразрядного регистра сдвига с асинхронным сбросом:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY shift_reg IS
       PORT
       (clk    : IN STD_LOGIC ;
        data   : IN STD_LOGIC ;
        reset  : IN STD_LOGIC ;
        q      : OUT STD_LOGIC
       );
END shift_reg;

ARCHITECTURE behav OF shift_reg IS
       SIGNAL rs: STD_LOGIC_VECTOR (3 downto 0) ;
BEGIN
process (clk, reset) begin
  if (reset = '0') then
```

```
    rs <= "0000";
  elsif (clk'event and clk = '1') then
    rs <= data & rs(3 downto 1);
  end if;
end process;
q <= rs(0);
END behav;
```

Для четырехразрядного регистра сдвига с асинхронным сбросом повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Глава 10

Счетные элементы (СЭ) или элементы счетчиков

СЭ могут быть построены на основе любой из изученных логик: ТТЛ, ЭСЛ или К-МОП. Они принадлежат к последовательностным логическим устройствам.

10.1. Описание функционирования двоичного СЭ (спецификация простейшего СЭ)

Двоичный СЭ должен считать входные (счетные) импульсы до двух, поэтому у него, как минимум, должен быть один (счетный) вход и два выхода. Один из выходов назовем "Состоянием" другой "Переносом" (рис. 10.1).

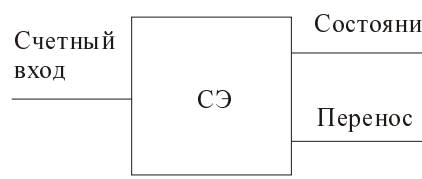


Рис. 10.1. УГО счетного элемента

Опишем таблицей функционирование двоичного СЭ (табл. 10.1).

Таблица 10.1. *Функционирование двоичного СЭ*

Счетный импульс	0	1	1	1	1	…
Состояние	0 →	1 →	0 →	1 →	0 →	…
Перенос	0 →	0 →	1 →	0 →	1 →	…

Временные диаграммы функционирования стандартного СЭ должны быть следующими — рис. 10.2.

Здесь для многоразрядного счетчика, состоящего из нескольких СЭ, импульс переноса каждого СЭ, в свою очередь, будет являться счетным импульсом для следующего разряда (следующего СЭ).

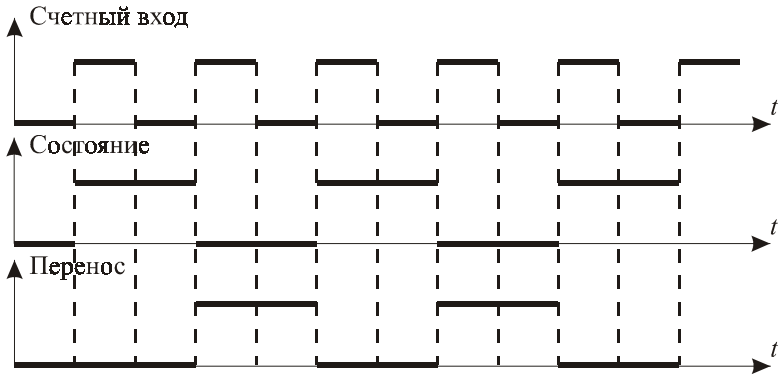


Рис. 10.2. Временная диаграмма стандартного СЭ

10.2. Внутренняя структура СЭ

В качестве СЭ можно использовать RS-триггер, но в соответствии с функционированием СЭ счетный импульс в этом случае должен поочередно поступать то на вход Set, то на вход Reset, поэтому нужно иметь переключатель (рис. 10.3). Если $Q = 0$, то переключатель вверх.

Здесь сигналы, Q и $Q!$ — управляющие, а сигналы на Счетном входе — информационные.

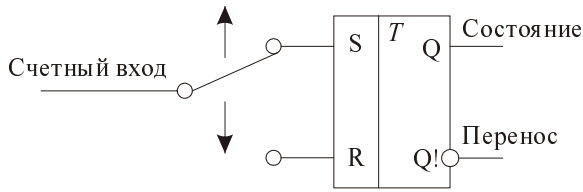


Рис. 10.3. Счетный элемент с переключателем

Такой переключатель реализуется парой элементов "И" (защелкой) (рис. 10.4). Если $Q = 1$, то переключатель вниз.

Рассмотрим развитие процесса переключения в счетном элементе (СЭ) с учетом времени задержки сигнала (Tp) элементами "И" и RS-триггером на "ИЛИ-НЕ". Из-за того что Tp элементов не определены, становится неясным время окончания счетного импульса, и схема оказывается неработоспособной, то есть такой счетный триггер за время счетного импульса может переключиться несколько раз вместо одного. Преодолевают описанную ранее неопределенность двумя способами:

- введением гарантированной задержки;
- избеганием преждевременного переключения.

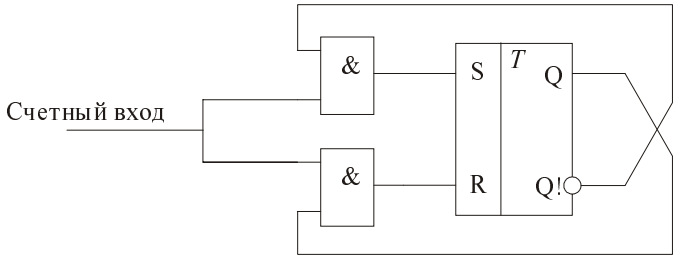


Рис. 10.4. Счетный элемент с защелкой

10.2.1. Т-триггер с задержкой

Рассмотрим первый способ — введем гарантированную задержку "Н" (рис. 10.5).

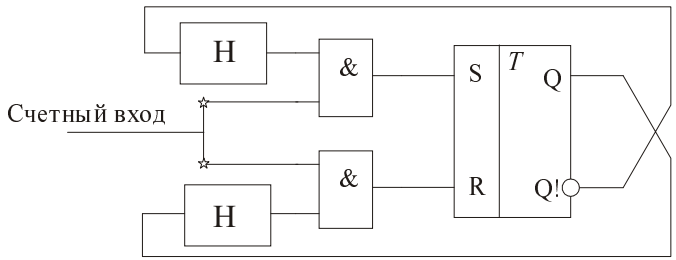


Рис. 10.5. Т-триггер с задержкой

Здесь расстояние во времени между событием (переходом из "0" в "1" сигнала на счетном входе) и следующим за ним событием (появлением отклика на входах элементов "И") — строго гарантированы временем задержки "Н".

Поэтому во время этой задержки нужно закончить счетный импульс, защелка из элементов "И" закроется, и новое состояние RS-триггера не поступит на его входы, тем самым СЭ переключится только один раз, что и требуется.

Такой СЭ называется Т-триггером с задержкой, на его примере рассмотрим работу JK-триггера.

10.2.2. JK-триггер с задержкой

Данный элемент отличается тем, что два входа "звездочки" не соединяют, а выводят отдельно (рис. 10.6).

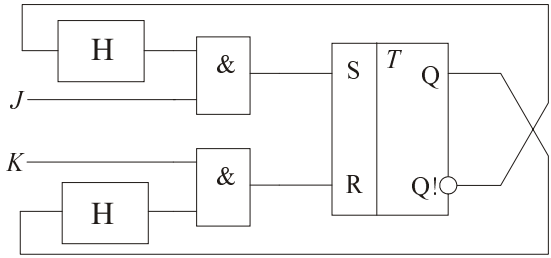


Рис. 10.6. JK-триггер с задержкой

JK-триггер имеет большое универсальное применение:

- если входы J и K соединить, то получим СЭ (Т-триггер);
- если со входами J и K работать отдельно, то JK-триггер ведет себя аналогично RS-триггеру (J — это S, а K — это R).

Рассмотрим временные диаграммы, описывающие работу JK-триггера (рис. 10.7).

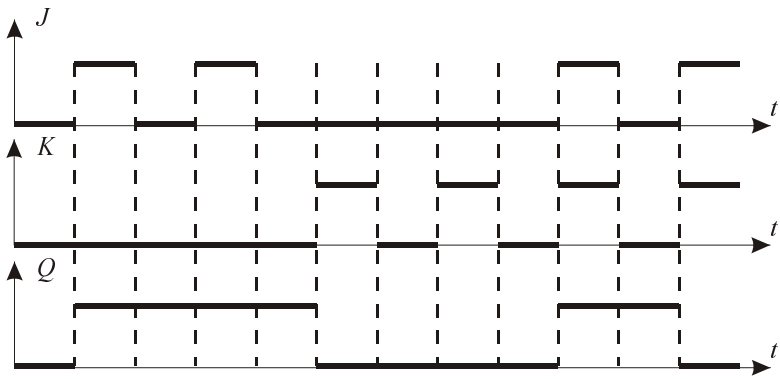


Рис. 10.7. Временные диаграммы работы JK-триггера

10.2.3. Двухтактный Т-триггер

Теперь по второму способу попытаемся избежать преждевременного переключения. В основе данного метода лежит использование двухтактного Т-триггера. Рассмотрим схему на "И", "ИЛИ-НЕ" и инверторах (рис. 10.8).

Этот двухтактный Т-триггер (СЭ) состоит из двух инверторов, двух защелок на элементах "И" и двух RS-триггеров на элементах "ИЛИ-НЕ".

Для СЭ нужно, чтобы под действием каждого счетного импульса он переходил в состояние противоположное предыдущему.

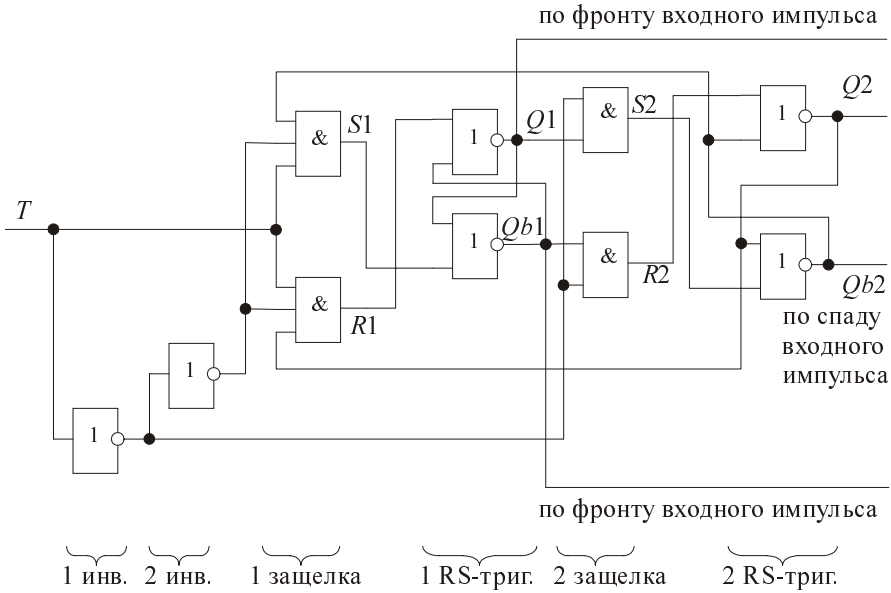


Рис. 10.8. Двухтактный Т-триггер

В этом Т-триггере связь выхода второго RS-триггера со входом первого организована так, что первый RS-триггер (при наличии счетного импульса и с помощью первой защелки) воспринимает состояние противоположное второму.

В момент прихода счетного импульса на вход первого RS-триггера с помощью первого инвертора и второй защелки удается разорвать связь выхода первого RS-триггера со входом второго RS-триггера. Этим схемотехническим решением с большой долей вероятности устраняется преждевременное переключение СЭ.

При неблагоприятном расположении задержек элементов (когда суммарная задержка элементов первой защелки и первого RS-триггера меньше, чем задержка первого инвертора) СЭ будет преждевременно переключаться (вторая защелка не успеет закрыться и СЭ переключится несколько раз). Для устранения этой ситуации подключают второй инвертор. Здесь только тогда, когда вторая защелка закроется, откроется первая защелка.

Из анализа работы Т-триггера следует, что если выходной сигнал снимать с первого RS-триггера, то переключение Т-триггера произойдет по фронту входного (счетного) импульса; если снимать со второго RS-триггера, то переключение произойдет по спаду входного импульса.

Если выходной сигнал снимать с выхода первого RS-триггера, то он будет изменяться по фронту входного (счетного) импульса, если с выхода второго RS-триггера, то он будет изменяться по спаду входного (счетного) импульса.

Аналогичная по функциональности схема двухтактного Т-триггера на элементах "И-НЕ" с инвертором представлена на рис.10.9.

На рис. 10.10 представлена схема того же двухтактного Т-триггера на элементах "И-НЕ".

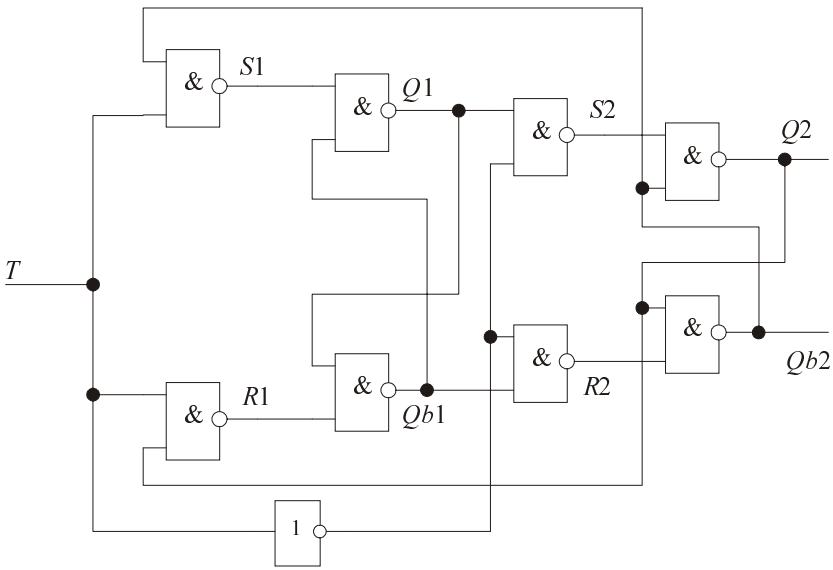


Рис. 10.9. Схема Т-триггера на "И-НЕ" с инвертором

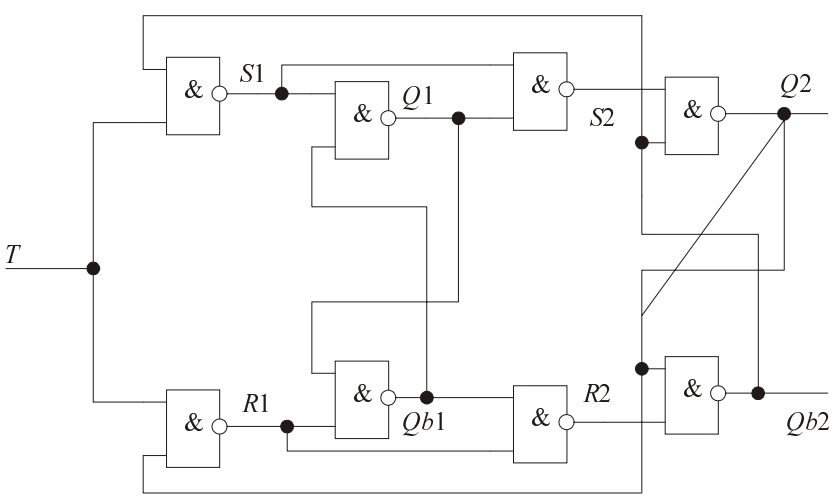


Рис. 10.10. Схема Т-триггера на "И-НЕ"

10.3. RST-триггер

В большинстве устройств счетчик нужно устанавливать в определенное начальное состояние. Для этого нужны отдельные входы S и R. Устройство, удовлетворяющее этим критериям, называется RST-триггером.

На рис. 10.11 представлен один из вариантов схемы RST-триггера.

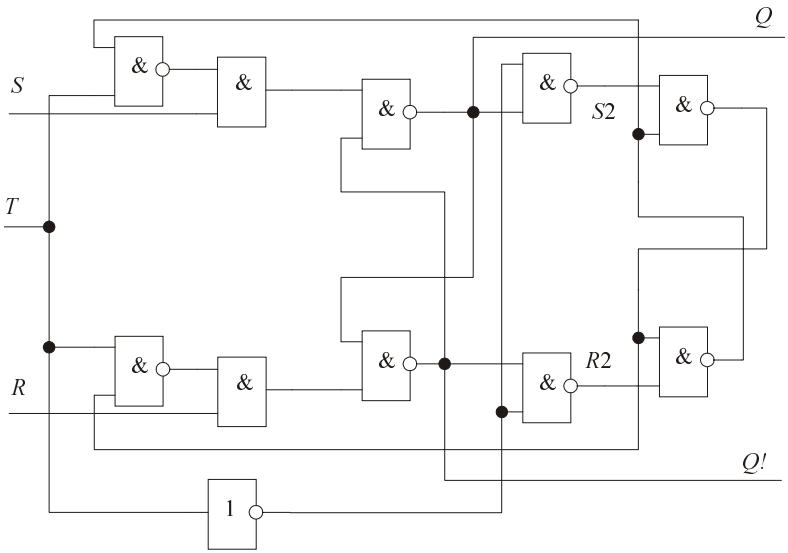


Рис. 10.11. RST-триггер

10.4. Современные JK-триггеры

Современные JK-триггеры строят на основе двухтактных T-триггеров. Рассмотрим пример изображения по ГОСТу одного из JK-триггеров (рис. 10.12).

Это двухступенчатый (двухтактный), синхронный (тактированный) JK-триггер с элементами "И" на входах J и K и инверсными входами S и R.

Если на входе тактовых импульсов стоит:

- "/" или "▶", то все остальные входы открываются по фронту тактового импульса;
- "\" или "◀", то все остальные входы открываются по спаду тактового импульса.

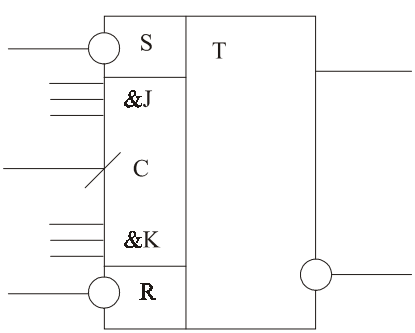


Рис. 10.12. УГО JK-триггера

10.5. Двоичные счетчики (делители)

Рассмотрим четырехразрядный двоичный счетчик (делитель) на JK-триггерах (рис. 10.13).

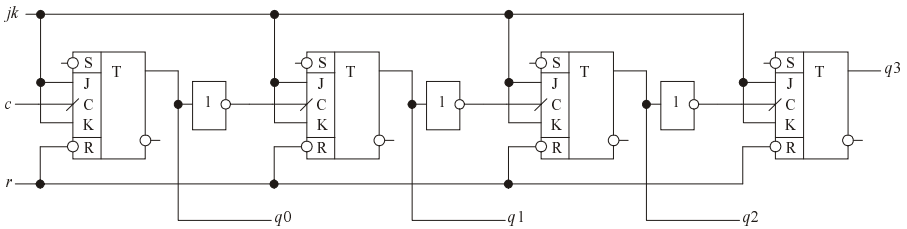


Рис. 10.13. Четырехразрядный двоичный счетчик

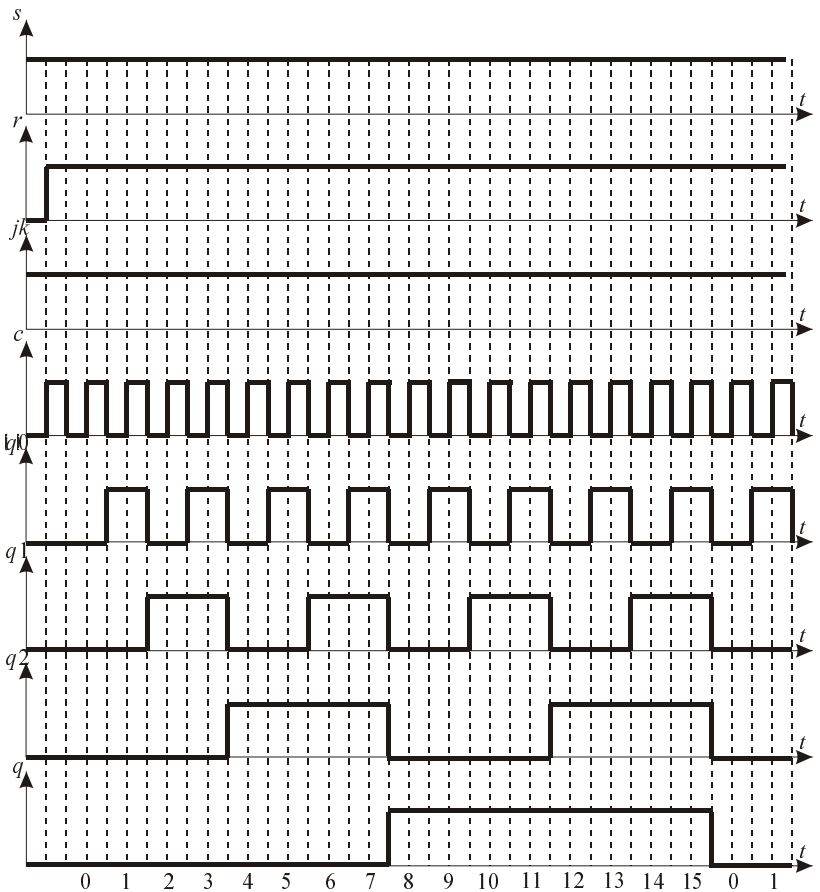


Рис. 10.14. Временные диаграммы четырехразрядного двоичного счетчика

Вход $CLRN$ — инверсный вход сброса выхода триггера в ноль (сбрасывается подачей на этот вход логического нуля, вход $CLRN$ не будет влиять на выход (будет отключен) при поступлении на него единицы).

Вход PRN — инверсный вход установки выхода триггера в единицу.

Вход C — вход синхронизирующих импульсов, причем информация будет появляться на выходе триггера при прохождении фронта этих импульсов.

Инверторы необходимы для нормальной работы счетчика, так как ненулевые разряды ($q1, q2, \ldots$) считаемого числа должны появляться по спаду импульса на выходе предыдущего разряда, тогда как триггер переключается по фронту синхроимпульса (см. рис. 10.14).

10.5.1. Схема выделения считаемых импульсов

Рассмотрим схему выделения шестого импульса (рис. 10.15).

Двоичный код шестого импульса 0110_2 (= 6_{10}), элемент "И-НЕ" будет иметь на выходе "1", только при наличии всех "1" на входах, поэтому ко входам, на которые поступает "0" (это нулевой и последний разряды), подключаем инверторы.

Таким образом, все "1" на входах "И-НЕ" получим только при комбинации "0110", которая соответствует шестому считаемому импульсу. По аналогии можно создать схему выделения любого по счету импульса.

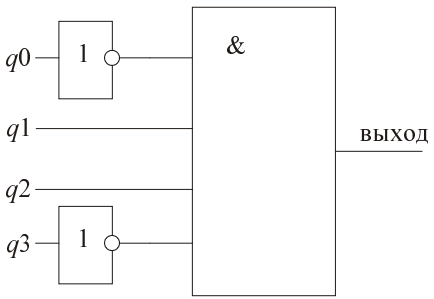


Рис. 10.15. Схема выделения шестого импульса

Лабораторная работа. Исследование схем T- и JK-триггеров и схемы счетчика

Цель работы

Лабораторная работа предполагает изучение работы схем триггеров, используемых в счетных устройствах (счетчиках).

Счетчик подсчитывает число импульсов, поступающих на его вход за некоторое время, формирует и запоминает код этого числа.

Схемотехническая реализация счетчика зависит от решаемых задач:

- увеличивается или уменьшается код с поступлением счетных импульсов (счетчики на сложение и счетчики на вычитание);
- в каком коде отображается результат счета (двоичные, двоично-десятичные и т. д.);
- какие триггеры используются для реализации счетчика и каким образом реализованы связи между его отдельными триггерами (счетчики на T-триггерах, JK-триггерах, D-триггерах с последовательным, сквозным или параллельным переносом);
- какие сервисные функции имеет счетчик (синхронные или асинхронные загрузки, сброс, разрешение счета, управление направлением счета и т. д.).

В данной работе изучается схема счетчика на сложение с последовательным переносом, построенная на JK-триггерах.

Программа работы

1. Создайте проект триггера типа JK-RS (рис. 10.16). Откомпилируйте и промоделируйте работу этой схемы. Зарисуйте временные диаграммы.

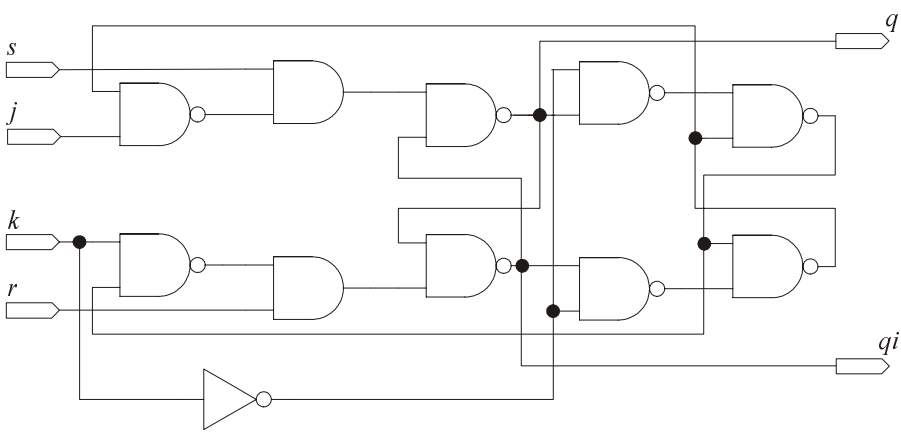


Рис. 10.16. Схема JK-RS-триггера

2. Создайте проект триггера типа TRS (рис. 10.17). Откомпилируйте и промоделируйте работу этой схемы. Зарисуйте временные диаграммы.
3. Исследуйте четырехразрядный двоичный счетчик на сложение с последовательным переносом, изображенный на рис. 10.18. Здесь в качестве базового элемента используется JK-триггер (элемент библиотеки примитивов JKFF).

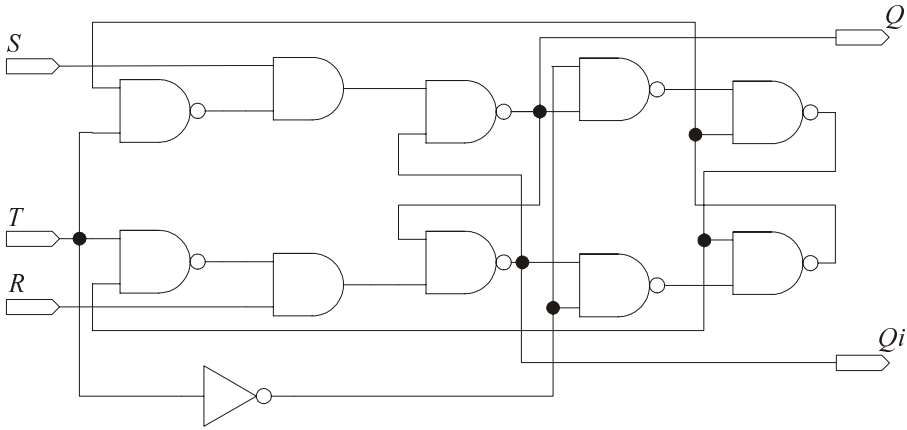


Рис. 10.17. Схема TRS-триггера

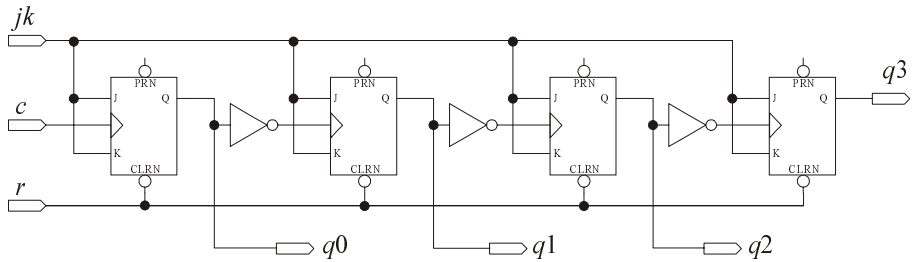


Рис. 10.18. Схема четырехразрядного счетчика

4. Исследуйте элемент JKFF библиотеки примитивов. Получите временные диаграммы работы этого элемента.
5. Исследуйте элемент 7493 библиотеки макрофункций mf (рис. 10.19). Этот элемент представляет собой одновременно счетчик по mod 2 и по mod 8 с возможностью установки в режим mod 16. Входы $RO1$ и $RO2$ сбрасывают счетчик в ноль (высокий уровень сигнала). Входы $CLKA$ и $CLKB$ счетные. $CLKA$ для счетчика по mod 2, а $CLKB$ для счетчика по mod 8. Выход QA — выход счетчика по mod 2, выходы QB, QC, QD — выходы счетчика по mod 8. Чтобы сделать четырехразрядный двоичный счетчик, нужно соединить выход $q0$ с входом $CLKB$. Проверьте все режимы работы элемента и получите для них временные диаграммы.
6. Создайте проекты JK- и T-триггеров на языке VHDL, используя поведенческий подход. Проверьте правильность работы программ, получите временные диаграммы.

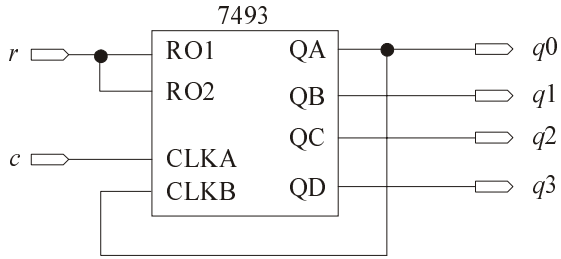
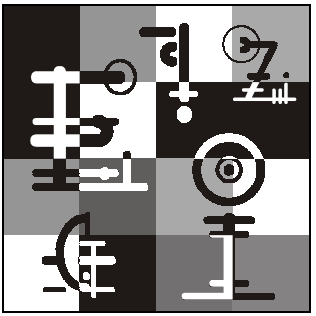


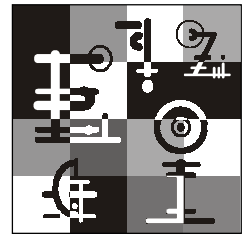
Рис. 10.19. Элемент 7493

7. Создайте проект четырехразрядного двоичного счетчика на языке VHDL (поведенческая модель). Проверьте правильность работы программы, получите временные диаграммы.
8. По аналогии с синтезируемым VHDL-кодом четырехразрядного регистра сдвига (см. лабораторную работу из *главы 9*) напишите синтезируемый VHDL-код четырехразрядного счетчика с асинхронным сбросом. Проверьте его на синтезируемость — для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Часть III

Функциональные устройства ЦТ средней интеграции, их описание на языке VHDL



Глава 11

Дешифраторы (DC), особенности языка VHDL

Дешифраторы (Decoder, DC) могут быть построены на основе любой из изученных логик: ТТЛ-, ЭСЛ- или К-МОП-логики. Они принадлежат к комбинационным логическим устройствам (КЛУ) средней интеграции.

11.1. Описание функционирования DC (спецификация простейшего DC)

Дешифратором (рис. 11.1) называют КЛУ, в котором каждой из комбинаций сигналов на входах соответствует сигнал только на одном из его выходов. (В зависимости от комбинаций на входах DC может выдавать управляющее воздействие в различные цепи, подключенные к выходам.)

Здесь $X0$ и $X1$ — входы, они обозначаются двоичными весами ($2^0 = 1$ и $2^1 = 2$).

EN (Enable) — вход, разрешающий работу.

$F0$, $F1$, $F2$, $F3$ — выходы, их четыре ($2^2 = 4$).

Двоичные DC на n-входов используются для преобразования двоичного кода в десятичный или в код "1 из N", где $N = 2^n$.

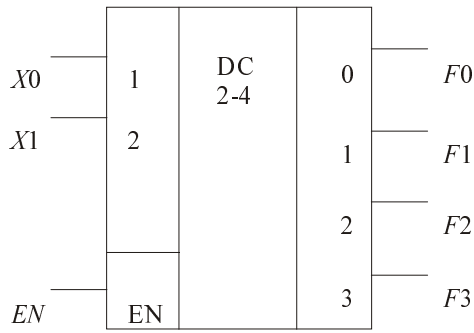


Рис. 11.1. Условное обозначение DC

11.2. Проектирование DC

В *главе 6* на примерах полусумматора и одноразрядного сумматора был подробно рассмотрен алгоритм проектирования комбинационных логических устройств (КЛУ), поэтому при рассмотрении DC и последующем рассмотрении КЛУ средней интеграции их проектирование будет проведено кратко.

1. Таблица истинности (табл. 11.1).

Таблица 11.1. *Описание таблицей истинности закона функционирования DC*

Входы			**Выходы**			
EN	**X0**	**X1**	**F0**	**F1**	**F2**	**F3**
1	0	0	1	0	0	0
1	1	0	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

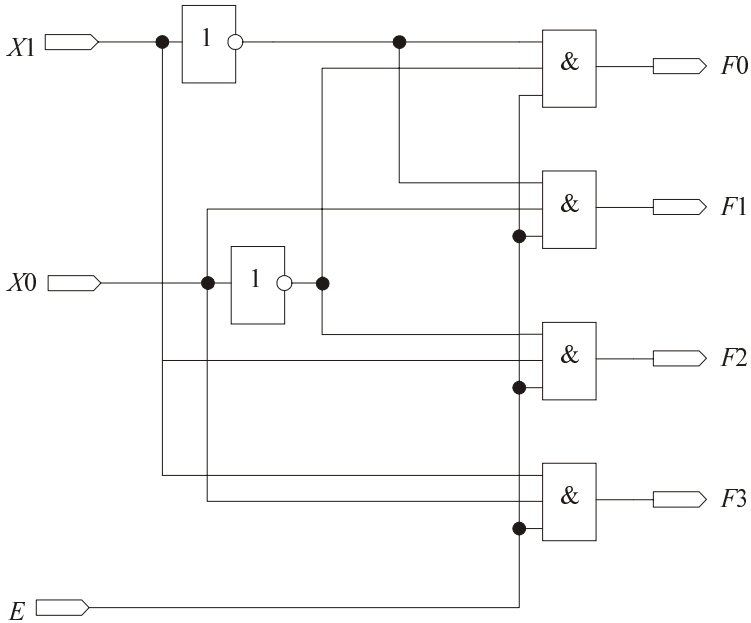


Рис. 11.2. Логическая схема DC

2. Описание закона функционирования DC переключательными функциями в СДНФ.
   $F0 = X0! * X1! * EN$; $F1 = X0 * X1! * EN$; $F2 = X0! * X1 * EN$; $F3 = X0 * X1 * EN$.
3. Логическая схема.
4. Проектирование в базисе "И-НЕ": если DC спроектирован на элементах "И-НЕ", то это *DC с инверсными выходами*.

Из логической схемы DC видно (см. рис. 11.2), что каждый его прямой вход подключен к $N = 2^{n-1}$ входам элементов "И", каждый из которых требует свой ток.

Для разгрузки внешних входов используют следующую схему — рис. 11.3.

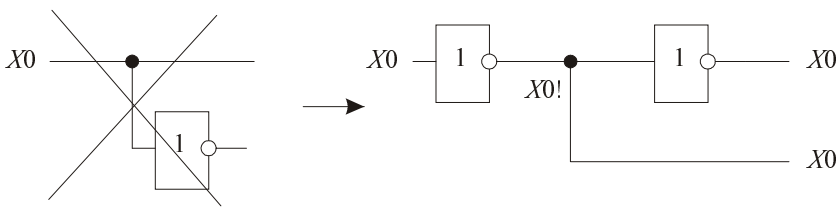


Рис. 11.3. Схемное решение разгрузки внешних входов

Время задержки сигнала дешифратором ($Тз_{DC}$) будет таким:

$Тз_{DC} = 2 * Тз\ инвертора + Тз"И"$, где $Тз"И" = \max\left(Тз^{01}, Тз^{10}\right)$.

11.3. Увеличение разрядности DC

Так как схема DC проста (в ней мало разных элементов), то размещать ее в дорогом многовыводном корпусе — нерентабельно, поэтому в сериях ИС используют DC с максимальным числом информационных входов, равным 4. DC большей разрядности строят из DC разрядности от 4 и ниже, по следующему алгоритму:

1. Входное слово делится на группы, причем разрядность групп, начиная с младшей, соответствует числу входов стандартных DC, имеющихся в серии ИС.
2. Группа оставшихся старших разрядов с помощью соответствующего ей DC выбирает один из дешифраторов, относящихся к более младшим разрядам.

Рассмотрим пример увеличения разрядности дешифраторов (DC): из DC с 2 входами и 4 выходами (2-4) и DC с 3 входами и 8 выходами (3-8) спроектирован DC с 5 входами и 32 выходами (5-32) (рис. 11.4).

Тестирование проекта: пусть на входы DC 5-32 $X4$, $X3$, $X2$, $X1, X0$ подано двоичное слово $10001_2 = 17_{10}$, следовательно, единица должна появиться на 17 выходе ($F17$). Проверьте это!

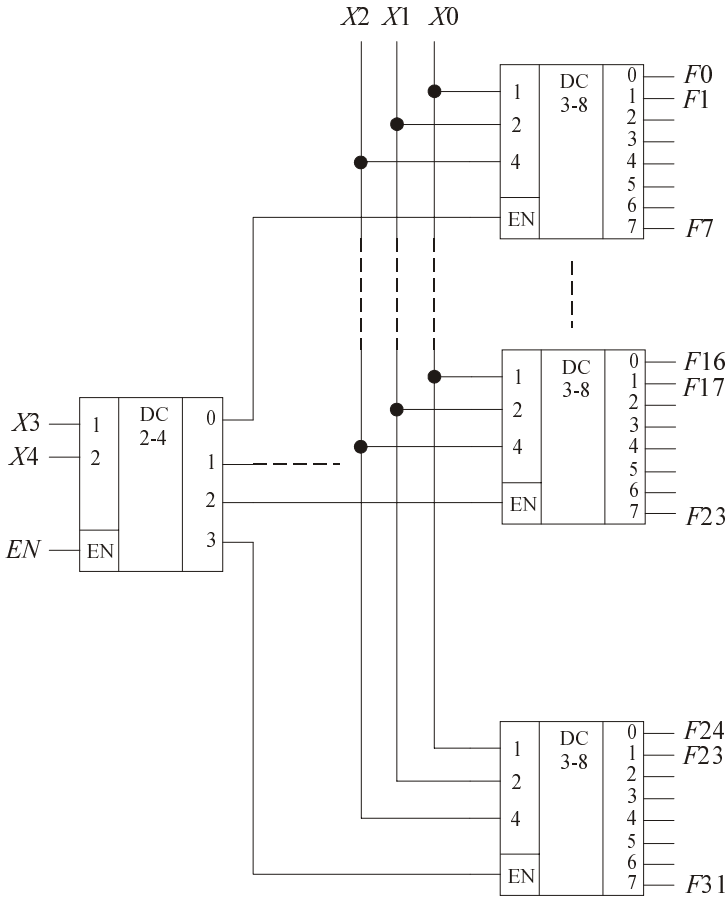


Рис. 11.4. Логическая схема DC с 5 входами и 32 выходами

11.4. Получение произвольных логических функций (ЛФ) в СДНФ с использованием DC и ЛЭ "ИЛИ"

Рассмотрим логические функции выходов DC с 3 входами (рис. 11.5) (пока, без элементов "ИЛИ" и подключений к ним).

На выходах DC имеем полный набор двоичных произведений (конъюнктивных термов), который можно составить из трех двоичных входных переменных: $X1$, $X2$, $X3$.

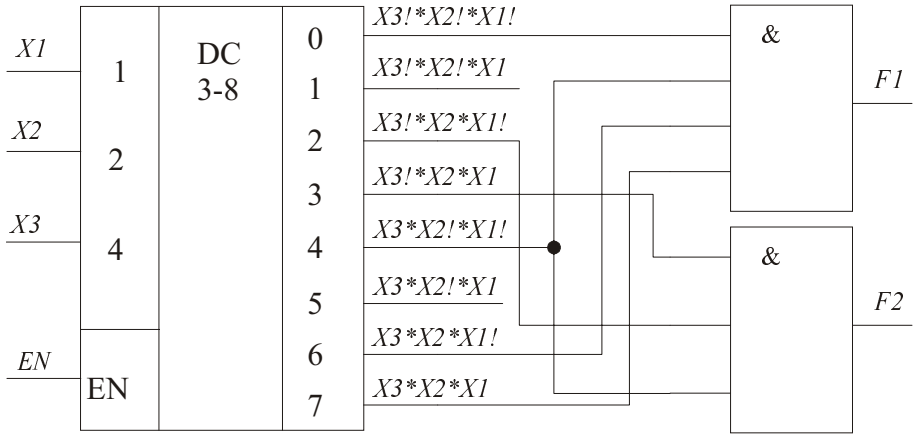


Рис. 11.5. Реализация ЛФ на DC и ЛЭ "ИЛИ"

Любая ЛФ в СДНФ — это дизъюнкция соответствующих термов! Следовательно, подключая на входы схемы "ИЛИ" эти термы, можно реализовать любую переключательную функцию трех переменных.

Рассмотрим пример.

Дано:

1. DC 3-8;
2. ЛЭ "ИЛИ" на 2, 3, … входов;
3. Логическое соотношение: $F*X \vee F*X! = F*(X \vee X!) = F$.

Реализовать функции $F1 = X2!*X1! \vee X3*X2$ и $F2 = X3!*X2 \vee X3*X2!*X1!$.

Реализация проводится подключением соответствующих термов ко входам элементов "ИЛИ" и представлена на рис. 11.5.

11.5. Особенности языка VHDL

С точки зрения программиста язык VHDL состоит из двух компонентов — общеалгоритмического и проблемно-ориентированного.

Общеалгоритмический компонент VHDL — это язык, близкий по синтаксису и семантике к современным языкам программирования типа Паскаль, С и др. Язык относится к классу строго типизированных. Помимо встроенных простых или скалярных типов данных (целый, вещественный, булев, битовый, данных типа время, данных типа ссылка) пользователь может вводить свои типы данных (перечисление, диапазон и др.).

Помимо скалярных данных можно использовать агрегаты: массивы `array`, в том числе и битовые векторы `bit_vector`, символьные строки `string`, записи `record`, файлы `file`.

Последовательно выполняемые (последовательные) операторы VHDL могут использоваться в описании процессов, процедур и функций. Их существует нескольких видов:

- оператор присваивания переменной (`:=`);
- последовательный оператор назначения сигналу (`<=`);
- последовательный оператор утверждения (`assert`);
- условный оператор (`if`);
- оператор выбора (`case`);
- оператор цикла (`loop`);
- пустой оператор (`null`);
- оператор возврата процедуры — функции (`return`);
- оператор последовательного вызова процедуры.

Язык поддерживает концепции пакетного и структурного программирования. Различаются локальные и глобальные переменные.

Проблемно-ориентированный компонент позволяет описывать цифровые системы в привычных разработчику понятиях и терминах. Сюда можно отнести:

- понятие модельного времени `now`;
- данные типа `time`, позволяющие указывать время задержки в физических единицах;
- данные вида сигнал `signal`, значение которых изменяется не мгновенно, как у обычных переменных, а с указанной задержкой, а также специальные операции и функции над ними;
- средства объявления объектов `entity` и их архитектур `architecture`.

Если говорить про операторную часть проблемно-ориентированного компонента, то условно ее можно разделить на средства:

- поведенческого описания аппаратуры (параллельные процессы и средства их взаимодействия);
- потокового описания (описание на уровне межрегистровых передач) — параллельные операторы назначения сигнала (`<=`) с задержкой передачи сигналов;
- структурного описания объектов проекта (операторы конкретизации компонентов с заданием карт портов (`port map`) и карт настройки (`generic map`), объявление конфигурации и т. д.).

Параллельно выполняемые (параллельные) операторы VHDL включают:

- оператор процесса `process`;
- оператор блока `block`;
- параллельный оператор назначения сигналу `<=`;
- оператор условного назначения сигналу `when`;
- оператор селективного назначения сигналу `select`;
- параллельный оператор утверждения `assert`;
- параллельный оператор вызова процедуры;
- оператор конкретизации компонента `port map`;
- оператор генерации конкретизации `generate`.

Примечание

Последовательные и параллельные операции назначения, вызова процедуры и утверждения различаются контекстно, то есть внутри процессов и процедур они последовательные, вне — параллельные.

Лабораторная работа. Исследование функционирования схем дешифраторов

Цель работы

Работа предполагает изучение функционирования схем дешифраторов. Дешифраторы и шифраторы по существу принадлежат к числу преобразователей кодов. Двоичные дешифраторы преобразуют двоичный код в код "1 из N". Иными словами, в зависимости от входного кода на выходе возбуждается одна из цепей. Число выходов полного дешифратора равно 2^n. Если часть входных наборов не используется, то дешифратор называют неполным.

В данной работе изучается схема дешифратора для преобразования 3-битного двоичного кода в код "1 из 8".

Программа работы

1. Создайте проект дешифратора (рис. 11.6). Откомпилируйте и промоделируйте работу этой схемы, предварительно установив размер сетки (**Grid Size**) $20.0\,ns$. Зарисуйте временные диаграммы. По полученным выходным диаграммам оцените время задержки сигнала при прохождении его через дешифратор.

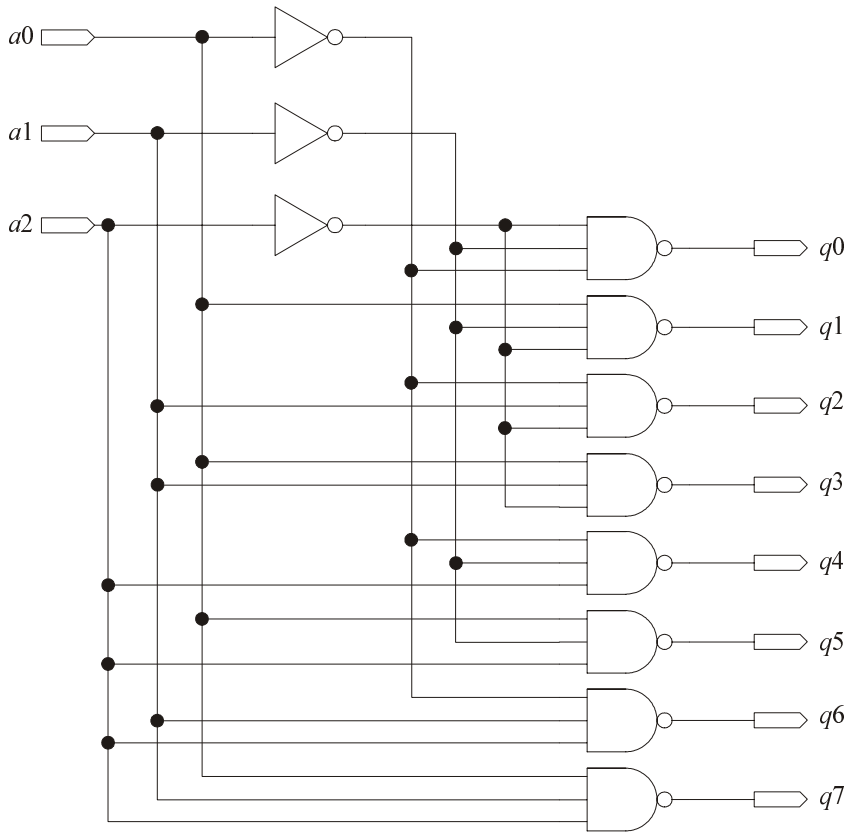


Рис. 11.6. Схема дешифратора
для преобразования 3-битного двоичного кода в код "1 из 8"

2. Создайте проект схемы дешифратора, увеличив его разрядность вдвое. Используйте для проекта компонент (символ), созданный в п. 1. Откомпилируйте и промоделируйте работу этой схемы. Зарисуйте временные диаграммы.
3. Спроектируйте дешифратор, показанный на рис. 11.6, с использованием поведенческой модели на языке VHDL. Сравните полученные временные диаграммы с диаграммами из п. 1.
4. Напишите структурную модель на языке VHDL для увеличения разрядности дешифратора вдвое. Полученные временные диаграммы сравните с диаграммами из п. 2.
5. Исследуйте работу элемента 74138 библиотеки mf (рис. 11.7). Описание работы указанного устройства можно посмотреть, используя HELP.

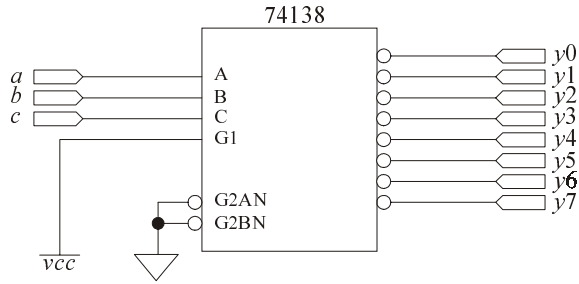
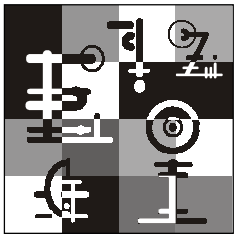


Рис. 11.7. Элемент 74138

6. Для проверки на синтезируемость написанного ранее VHDL-кода рассмотрим синтезируемый VHDL-код DC 3-8:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity decoder is
port ( a  : in std_logic_vector (2 downto 0);
       q  : out std_logic_vector (7 downto 0));
end decoder;
architecture behav of decoder is
begin
process (a)
begin
case a is
  when "000" =>  q <= "00000001";
  when "001" =>  q <= "00000010";
  when "010" =>  q <= "00000100";
  when "011" =>  q <= "00001000";
  when "100" =>  q <= "00010000";
  when "101" =>  q <= "00100000";
  when "110" =>  q <= "01000000";
  when "111" =>  q <= "10000000";
  when others =>  q <= "00000000";
end case;
end process;
end behav;
```

По аналогии с синтезируемым VHDL-кодом четырехразрядного регистра сдвига (лабораторная работа из *главы 9*) проверьте VHDL-код DC 3-8 на синтезируемость. Для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Глава 12

Шифраторы

Шифраторы преобразуют код "1 из N" в двоичный. Это обратная операция дешифрированию. Шифраторы принадлежат к комбинационным логическим устройствам (КЛУ) средней интеграции.

12.1. Описание функционирования приоритетного шифратора (HPRI)

В сериях ИС используется приоритетный шифратор (HPRI) (рис. 12.1), который в случае прихода не одного, а нескольких сигналов на входы, выдаст двоичный код наиболее приоритетного входа, на который пришел сигнал. Если сигнал приходит на единственный вход, то на выходе имеем двоичный код номера этого входа и HPRI работает как обычный шифратор.

Здесь $r0$ … $r7$ (request — запрос) — сигналы на входах HPRI.

EI (Enable In) — вход, разрешающий работу.

$a2$, $a1$, $a0$ — двоичный код наиболее приоритетного входа, на который пришел сигнал.

EO (Enable Out) — выходной сигнал, появляющийся при отсутствии сигналов на входах HPRI. Разрешает работу более младшего HPRI при увеличении разрядности.

G — выходной сигнал, появляющийся при наличии входных сигналов.

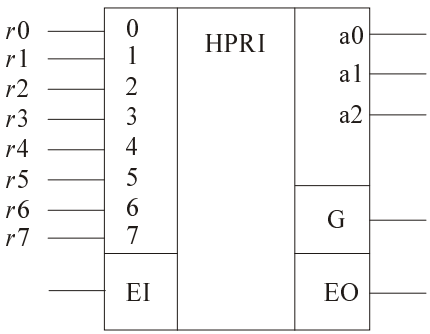


Рис. 12.1. УГО шифратора

12.2. Проектирование HPRI

1. Таблица истинности HPRI (табл. 12.1).

Таблица 12.1. *Таблица истинности шифратора*

EI	r7	r6	r5	r4	r3	r2	r1	r0	a2	a1	a0	G	EO
1	1	z	z	z	z	z	z	z	1	1	1	1	0
1	0	1	z	z	z	z	z	z	1	1	0	1	0
1	0	0	1	z	z	z	z	z	1	0	1	1	0
1	0	0	0	1	z	z	z	z	1	0	0	1	0
1	0	0	0	0	1	z	z	z	0	1	1	1	0
1	0	0	0	0	0	1	z	z	0	1	0	1	0
1	0	0	0	0	0	0	1	z	0	0	1	1	0
1	0	0	0	0	0	0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	1
0	z	z	z	z	z	z	z	z	0	0	0	0	0

2. ЛФ в СДНФ:

$$a2 = (r7 + r7!*r6 + r7!*r6!*r5 + r7!*r6!*r5!*r4)*EI.$$

$$a1 = (r7 + r7!*r6 + r7!*r6!*r5!*r4!*r3 + r7!*r6!*r5!*r4!*r3!*r2)*EI.$$

$$a0 = (r7 + r7!*r6!*r5 + r7!*r6!*r5!*r4!*r3 + r7!*r6!*r5!*r4!*r3!*r2!*r1)*EI.$$

$$EO = r7!*r6!*r5!*r4!*r3!*r2!*r1!*r0!*EI\ .$$

$$G = (r7 + r6 + r5 + r4 + r3 + r2 + r1 + r0)*EI.$$

3. Минимизация ЛФ с использованием логического соотношения: $X + F*X! = X + F$.

$$a2 = (r7 + r6 + r5 + r4)*EI.$$

$$a1 = (r7 + r6 + r5!*r4!*r3 + r5!*r4!*r2)*EI.$$

$$a0 = (r7 + r6!*r5 + r6!*r4!*r3 + r6!*r4!*r2!*r1)*EI.$$

4. Логическая схема (рис. 12.2).

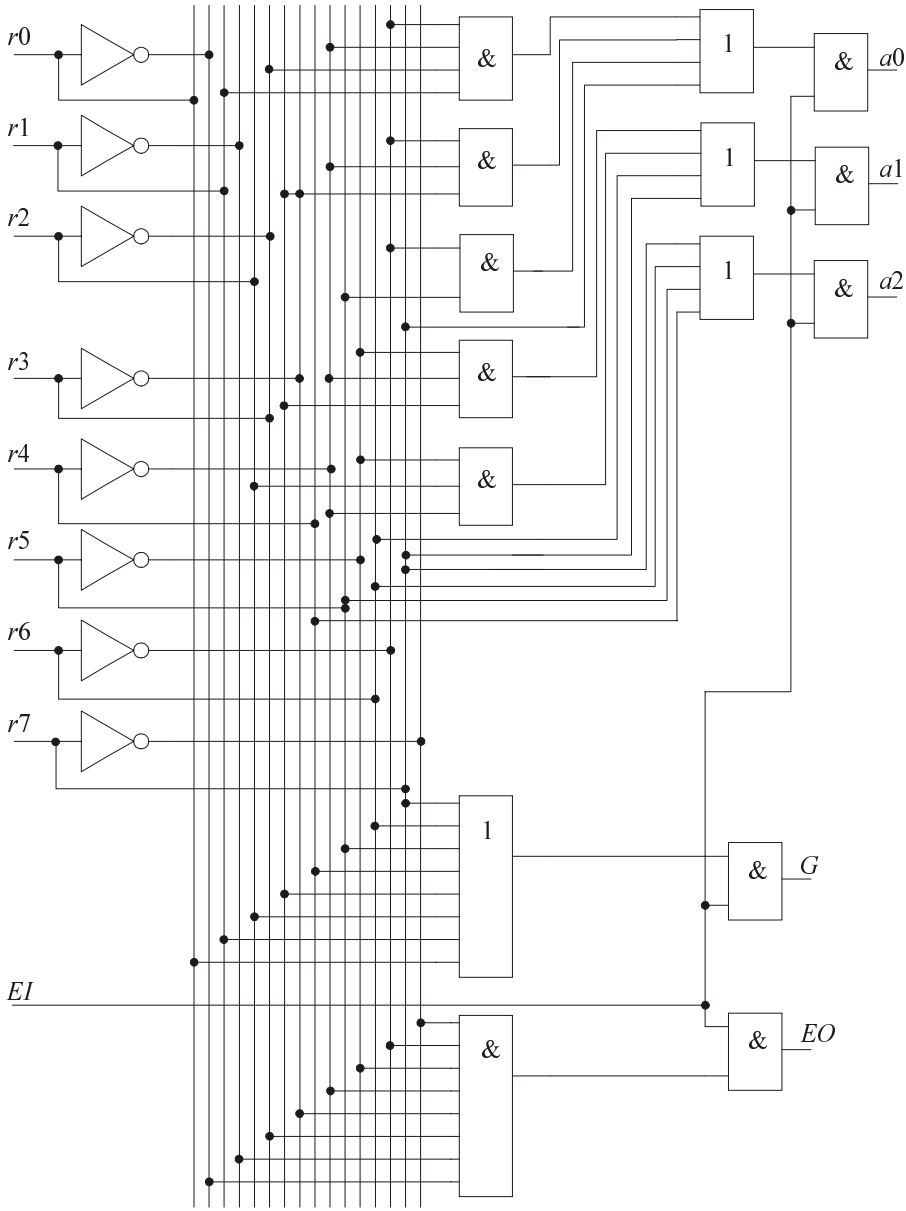


Рис. 12.2. Логическая схема шифратора

12.3. Увеличение разрядности HPRI

Рассмотрим пример увеличения разрядности HPRI с 8 входов до 16 (рис. 12.3).

Дано: HPRI 8-3.

Спроектировать: HPRI 16-4.

Тестирование проекта: пусть на вход $r9$ подана "1", тогда на выходе HPRI 16-4 $a3a2a1a0$ должно быть двоичное слово $1001_2 = 9_{10}$. Пусть на вход $r7$ подана "1", тогда на выходе HPRI 16-4 $a3a2a1a0$ должно быть двоичное слово $0111_2 = 7_{10}$. Пусть и на вход $r9$ подана "1" и на вход $r7$ подана "1", тогда на выходе HPRI 16-4 $a3a2a1a0$ должно быть двоичное слово $1001_2 = 9_{10}$. Проверьте это!

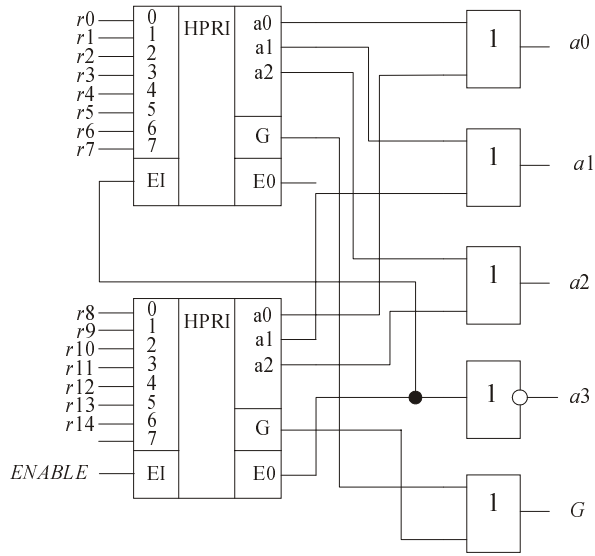


Рис. 12.3. Схема шифратора увеличенной разрядности

Примечание

При необходимости проектирования HPRI в базисе "И-НЕ" применяют формулу $(a1!*a2!)! = a1 + a2$.

12.4. Указатель наиболее приоритетного входа из всех тех, на которые пришли входные сигналы

В этом указателе число входов и выходов одинаково. В качестве результата на выходе имеем код "1 из N". Из многих входных сигналов на один из выходов проходит сигнал, поступивший на наиболее приоритетный, соответствующий этому выходу, вход. Указатель используется при нормализации чисел с плавающей запятой.

Лабораторная работа. Исследование функционирования схем шифраторов и указателей

Цель работы

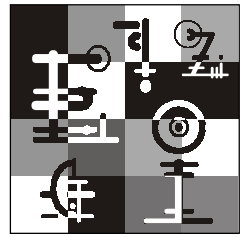
Лабораторная работа предполагает изучение работы схем шифраторов и указателей наиболее приоритетного входа из всех тех, на которые пришли входные сигналы, или указателей старшей единицы. С понятием шифрации связано представление о сжатии данных, с понятием дешифрации — обратное преобразование. Двоичные шифраторы выполняют операцию, обратную по отношению к операции дешифратора. Они преобразуют двоичный код в код "1 из N". Иными словами, в зависимости от входного кода на выходе возбуждается одна из цепей. Одно из основных применений шифратора — ввод данных с клавиатуры, при котором нажатие клавиши с десятичной цифрой должно приводить к передаче в устройство двоичного кода данной цифры. Указатели старшей единицы решают ту же задачу, что и приоритетные шифраторы, но вырабатывают результат в иной форме — в виде кода "1 из N". Число входов в этом случае равно числу выходов схемы. Указатели старшей единицы применяются в устройствах нормализации чисел с плавающей точкой и т. д.

В данной работе изучается схема приоритетного шифратора, преобразующего код "1 из 8" в 3-битный двоичный код.

Программа работы

1. Спроектируйте схему приоритетного шифратора, представленного *в главе 12* и преобразующего код "1 из 8" в 3-битный двоичный код. Получите и зарисуйте временные диаграммы. Создайте новый компонент для этого проекта.
2. Увеличьте разрядность шифратора вдвое, используя для проекта компонент, полученный в п. 1.
3. Спроектируйте шифратор с использованием поведенческой модели на языке VHDL. Сравните полученные временные диаграммы с диаграммами из п. 1.
4. Напишите структурную модель на языке VHDL для увеличения разрядности шифратора в два раза. Полученные временные диаграммы сравните с диаграммами из п. 2.
5. Создайте проект схемы указателя старшей единицы "8-8" или "16-16". Протестируйте его (получите и зарисуйте временные диаграммы).

6. По аналогии с синтезируемым VHDL-кодом DC 3-8 (лабораторная работа в *главе 11*) напишите синтезируемый VHDL-код приоритетного шифратора, преобразующего код "1 из 8" в 3-битный двоичный код. Проверьте его на синтезируемость, для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.
7. По аналогии с синтезируемым VHDL-кодом DC 3-8 (лабораторная работа в *главе 11*) напишите синтезируемый VHDL-код схемы указателя старшей единицы "8-8" или "16-16". Проверьте его на синтезируемость, для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Глава 13

Мультиплексоры

С помощью мультиплексоров выполняют важную в вычислительной технике функцию маршрутизации (функцию определения пути передачи информации). Достоинством мультиплексоров является возможность реализации на них произвольных логических функций.

13.1. Мультиплексоры (MUltipleXer — MUX)

Мультиплексором называют комбинационную логическую схему, позволяющую выбирать с помощью управляющего (адресующего) слова один из нескольких входов и подавать сигнал с этого входа на общий выход.

13.1.1. Описание функционирования мультиплексора (спецификация простейшего MUX)

Условное обозначение MUX приведено на рис. 13.1.

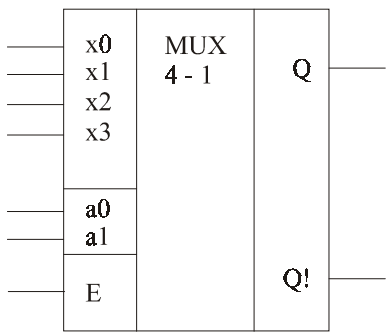


Рис. 13.1. УГО мультиплексора

Здесь $x0 \ldots x3$ — входные сигналы MUX.

E (Enable) — вход, разрешающий работу.

$a1$, $a0$ — адресные входы MUX.

Q — прямой выход.

$Q!$ — инверсный выход.

MUX 4-1 — мультиплексор с 4 входами в 1 выход.

Некоторые мультиплексоры можно использовать в качестве "демультиплексоров" или дешифраторов, поменяв назначение выводов.

13.1.2. Проектирование MUX

1. Таблица истинности MUX (табл. 13.1).

Таблица 13.1. *Таблица истинности мультиплексора*

E	**a0**	**a1**	**F**
1	0	0	x0
1	1	0	x1
1	0	1	x2
1	1	1	x3
0	z	z	0

2. ЛФ в СДНФ:

$$F = (x0 * a0!*a1!+x1 * a0 * a1!+x2 * a0!*a1 + x3 * a0 * a1) * E.$$

Если необходимо проектировать в базисе "И-НЕ", то, используя правило Де'Моргана: $a + b = (a!*b!)!$, получаем:

$$F = (((x0 * a0!*a1!)!*(x1 * a0 * a1!)!*(x2 * a0!*a1)!*(x3 * a0 * a1)!)!) * E.$$

Введем понятие мультиплексной формулы:

$$F = x_0 * a_{n-1}!*a_{n-2}!*\ldots * a_1!*a_0!+x_1 * a_{n-1}!*a_{n-2}!*\ldots * a_1!*a_0 +$$

$$+ \ldots + x_{2^n-1} * a_{n-1} * a_{n-2} * \ldots * a_1 * a_0.$$

Здесь слагаемые называются конъюнктивными термами. В мультиплексной формуле имеем полное возможное количество термов для n-адресных входов.

3. Логическая схема (рис. 13.2).

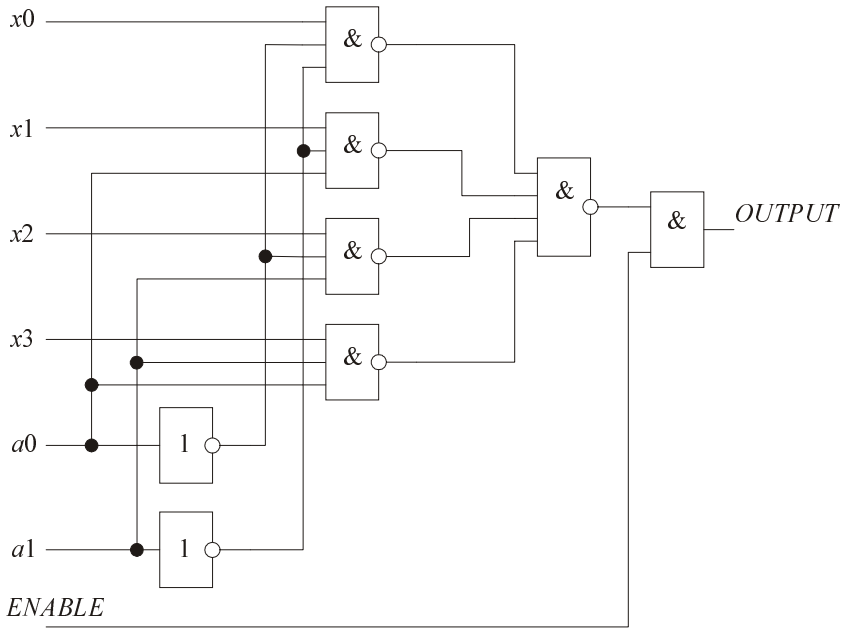


Рис. 13.2. Логическая схема мультиплексора

13.1.3. Увеличение разрядности MUX

Стандартные MUX имеют разрядность меньшую или равную 16 входам в 1 выход (16-1). Для ее увеличения используется пирамидальная структура.

Рассмотрим пример (рис. 13.3).

Дано: MUX 4-1.

Спроектировать: MUX 16-1.

Проектирование проводится по аналогии с дешифратором (DC), но здесь имеем собирающую пирамиду.

Тестирование проекта: пусть на входы $a3a2a1a0$ сначала подано двоичное слово $1011_2 = 11_{10}$, а затем подано двоичное слово $1110_2 = 14_{10}$, тогда на выход, соответственно, должны поступать сигналы с 11 и 14 входов. Проверьте это!

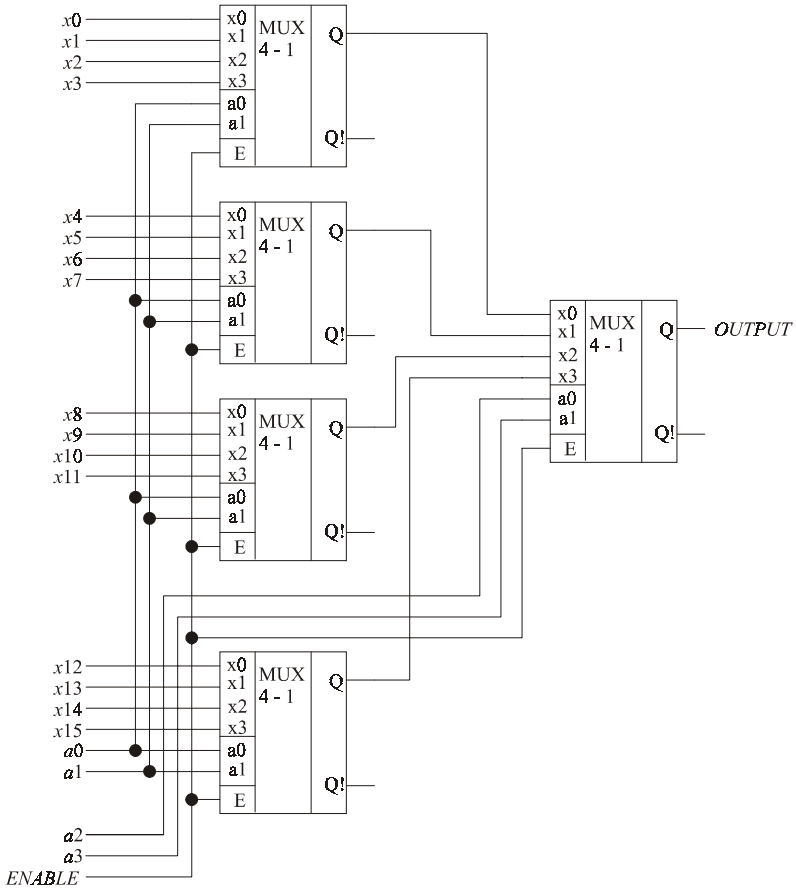


Рис. 13.3. Логическая схема мультиплексора 16-1

13.2. Демультиплексоры (DMX)

В DMX совокупность значений сигналов ($x_0 \ldots x_{n-1}$) определяет номер (адрес) выходного канала, к которому подключается сигнал E.

Условное обозначение DMX приведено на рис. 13.4.

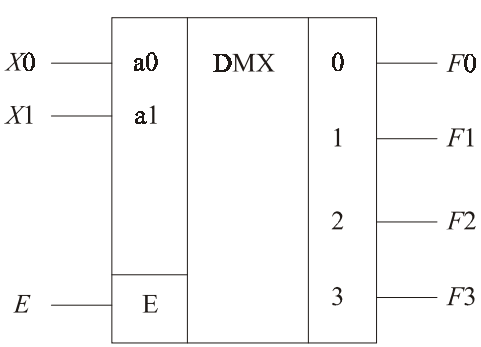


Рис. 13.4. УГО демультиплексора

13.2.1. Проектирование DMX

Проектируем по аналогии с DC:

1. Переключательные функции в СДНФ:

$$F0 = X0!*X1!*E;\ F1 = X0*X1!*E;\ F2 = X0!*X1*E;\ F3 = X0*X1*E;$$

2. Логическая схема (рис. 13.5).

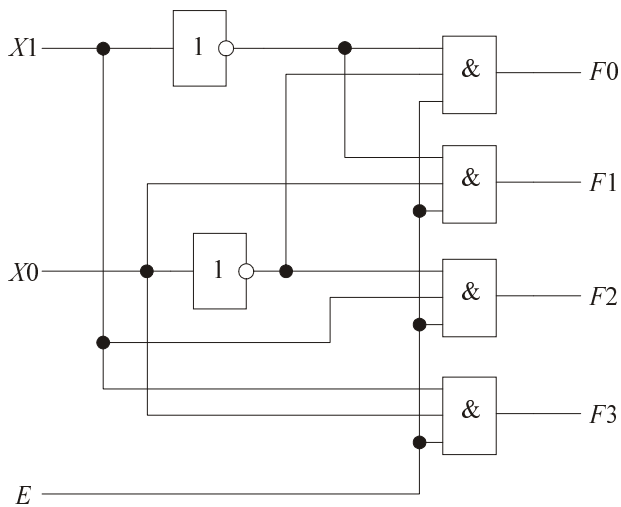


Рис. 13.5. Логическая схема демультиплексора

Примечание

Если в схеме DC на вход *EN* подавать информационный сигнал (*E* в DMX), то DC будет работать как DMX, поэтому DC со входом разрешения *EN* называют DC-DMX. Разницы между ними нет.

13.3. Получение произвольных логических функций (ЛФ) с помощью MUX

По аналогии с DC, используя MUX, можно получить произвольную логическую функцию, но с количеством аргументов, равным числу адресных входов MUX.

Назовем сигнальные входы MUX ($x0$ … $x3$) настроечными. Аргументы ЛФ будем подавать на адресные входы.

13.3.1. Первый вариант реализации произвольных ЛФ на MUX

Дано: MUX 4-1.

Спроектировать (реализовать) на MUX логическую функцию:

$F1 = Y0!*Y1!+Y0*Y1$.

Проектирование:

1. По логической функции пишем таблицу истинности (табл. 13.2).

Таблица 13.2. *Таблица истинности рассматриваемой ЛФ*

Y1 (a1)	**Y0 (a0)**	**F1**
0	0	1
0	1	0
1	0	0
1	1	1

2. Рассмотрим первую строку: аргументами $Y0$ и $Y1$ выбирается настроечный вход $x0$, сигнал с него будет подан на выход Q, этот сигнал по таблице истинности должен быть равен "1", поэтому на настроечный вход $x0$ подаем "1".

   На остальные настроечные входы подаем сигналы, рассуждая аналогично.

   В результате имеем — рис. 13.6.

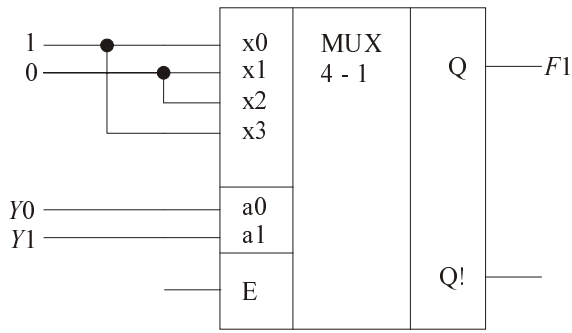


Рис. 13.6. Реализация на MUX рассматриваемой ЛФ

Первый вариант реализации произвольных ЛФ на MUX требует много настроечных входов, а это связано с дорогим многовыводным корпусом, что нерентабельно. В программируемых логических интегральных схемах (ПЛИС) эту ситуацию устраняют, последовательно вводя настроечную комбинацию в регистр сдвига, к разрядам которого подключены настроечные входы. Здесь жертвуем скоростью работы: результат получаем не за один такт, а за 2^n тактов, где n — число аргументов.

13.3.2. Второй вариант реализации произвольных ЛФ на MUX

В его основе лежит уменьшение числа настроечных входов за счет подачи одного из аргументов на один из настроечных входов. Этот аргумент меньше всего входит в термы логической функции, описывающей проектируемое ЛУ.

Рассмотрим на примере.

Дано: MUX 4-1.

Спроектировать (реализовать) на MUX логическую функцию:

$F1 = Y1! * Y2! + Y1 * Y2 * Y3$.

Проектирование (на настроечный вход будем подавать аргумент $Y3$):

1. По логической функции пишем таблицу истинности (табл. 13.3).

Таблица 13.3. *Таблица истинности рассматриваемой ЛФ*

Y1 (a1)	Y0 (a0)	F1
0	0	1
0	1	0
1	0	0
1	1	Y3

2. Рассуждая аналогично первому варианту, имеем элемент, показанный на рис. 13.7.

Здесь для трех аргументов при использовании первого варианта потребовалось бы 8 настроечных входов, а при использовании второго варианта всего четыре.

Рассмотрим на примере подачу сразу двух аргументов на один настроечный вход. Для этого необходимо вводить дополнительные логические элементы. В нашем примере это ЛЭ "И".

Дано: MUX 4-1.

Спроектировать (реализовать) на MUX логическую функцию:

$F = X1 * X2! + X3 * X4$.

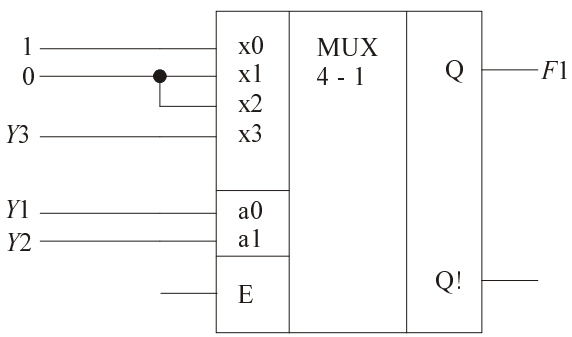


Рис. 13.7. Реализация на MUX рассматриваемой ЛФ

Проектирование (на настроечный вход будем подавать произведение аргументов $X3 * X4$):

1. По логической функции пишем таблицу истинности (табл. 13.4).

Таблица 13.4. *Таблица истинности рассматриваемой ЛФ*

X2 (a0)	X1 (a1)	F
0	0	X3*X4
0	1	1
1	0	X3*X4
1	1	X3*X4

2. Рассуждая аналогично первому варианту, имеем — рис. 13.8.

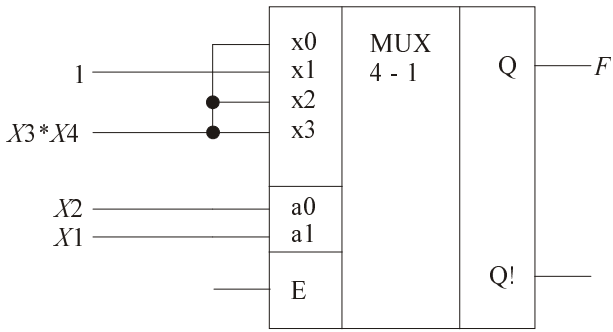


Рис. 13.8. Реализация на MUX рассматриваемой ЛФ

В случае подачи двух аргументов на настроечный вход для обеспечения универсальности, кроме MUX необходимо иметь логический блок, на входы которого подаются эти аргументы, а на выходах снимаются результаты от всех известных логических операций с двумя аргументами.

При подаче трех и более аргументов на настроечные входы используют собирающую пирамидальную структуру, состоящую из двух ярусов. Более подробно об этом можно прочитать в соответствующей литературе.

Лабораторная работа. Исследование функционирования схем мультиплексоров и демультиплексоров

Цель работы

Лабораторная работа предполагает изучение работы схем мультиплексоров и демультиплексоров. Традиционное использование мультиплексоров состоит в управляемой передаче данных от нескольких входных каналов в один выходной канал. Каждый из входных каналов поочередно подключается к выходному под управлением адресующего сигнала. Входы мультиплексора делятся на две группы: информационные и управляющие (адресующие). Двоичный управляющий код задает номер информационного входа, сигнал с которого поступает на выход.

Демультиплексор имеет обратное назначение: распределяет данные из одного канала между несколькими приемниками информации.

Программа работы

1. Создайте графический проект мультиплексора MUX 4-1, представленного в данной главе. Откомпилируйте и промоделируйте работу этой схемы, предварительно установив размер сетки (**Grid Size**) 20.0 *ns* . По полученным выходным диаграммам оцените время задержки сигнала при прохождении его через мультиплексор.
2. Спроектируйте этот мультиплексор с использованием поведенческой модели на языке VHDL. Создайте новый компонент для этого проекта.
3. Спроектируйте графический проект MUX 64-1 из MUX 4-1 и оттестируйте его (получите временные диаграммы).
4. Спроектируйте и оттестируйте проект MUX 64-1 из MUX 4-1 с использованием структурной модели на языке VHDL.
5. Исследуйте работу элемента 74153 библиотеки mf (рис. 13.9). Описание работы указанного устройства можно посмотреть в разделе **Помощь** в САПР Max+Plus II.
6. Спроектируйте и протестируйте схему демультиплексора (DMX), представленного в данной главе.

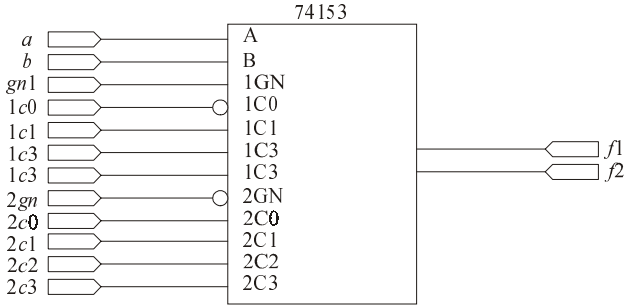
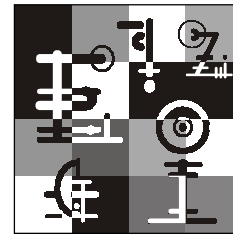


Рис. 13.9. Элемент 74153

7. Спроектируйте и оттестируйте этот демультиплексор с использованием поведенческой модели на языке VHDL.
8. Создайте проекты устройств, построенных с использованием мультиплексора (п.1) и выполняющих заданные вами самостоятельно логические функции, протестируйте эти проекты (получите временные диаграммы).
9. По аналогии с синтезируемым VHDL-кодом DC 3-8 (лабораторная работа в *главе 11*) напишите синтезируемый VHDL-код мультиплексора MUX 4-1. Проверьте его на синтезируемость, для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Глава 14

Компараторы (СМР)

Компаратор — это устройство сравнения двух двоичных слов, определяющее, в каком отношении находятся между собой эти слова.

14.1. Описание функционирования компаратора (СМР) (спецификация простейшего СМР)

Будем считать базовыми отношениями "равенство" (при этом сигнал, соответствующий равенству двоичных слов "X" и "Y", равняется "1" ($F_{X=Y}=1$)) и "больше" ($F_{X>Y}=1$).

Тогда из алгебры логики получаем: $F_{X\neq Y}=\left(F_{X=Y}\right)!$; $F_{X<Y}=F_{Y>X}$; $F_{X\leq Y}=\left(F_{Y>X}\right)!$; $F_{X\leq Y}=\left(F_{X>Y}\right)!$.

Стандартные СМР имеют три выхода: "равно", "больше" и "меньше" (рис. 14.1).

Здесь $X0\ \ldots X3$ — двоичное слово A.

$Y0\ \ldots Y3$ — двоичное слово B.

$A_<$, $A_=$, $A_>$ — признаки отношений, расположенные слева внизу в УГО компаратора, необходимы для увеличения его разрядности. Они показывают, в каком отношении находятся более младшие разряды (в нашем случае 4 разряда) сравниваемых многоразрядных двоичных слов (чисел) A и B. Для примера, если более младшие разряды слова A меньше более младших разрядов слова B, то на вход $A_<$ подается единица, в противном случае — ноль. Аналогично с оставшимися признаками отношений.

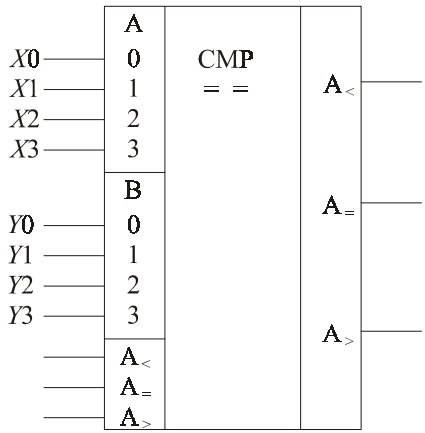


Рис. 14.1. УГО компаратора

$A_<$, $A_=$, $A_>$ — признаки отношений, расположенные справа в УГО компаратора, определяют отношение четырех разрядов двоичных слов A и B, подаваемых на этот СМР: если все эти разряды равны, то на выходе $A_=$ появится единица, на остальных выходах — нули (аналогичные рассуждения проводятся для оставшихся выходов, соответствующих своим признакам отношений).

14.2. Проектирование СМР

Проектирование СМР начнем с этапа написания логических функций.

14.2.1. Реализация отношения "равенство"

Условие *равенства* одноименных разрядов будет выглядеть следующим образом:

$$Z_i = X_i * Y_i + X_i!*Y_i! = ((X_i * Y_i)!*(X_i!*Y_i!)!)! = (X_i * Y_i! + X_i!*Y_i)! = (X_i \; xor \; Y_i)!.$$

Условие *неравенства* (отрицание условия равенства) одноименных разрядов будет выглядеть так:

$$Z_i! = X_i * Y_i! + X_i!*Y_i = ((X_i * Y_i)!*(X_i!*Y_i)!)! = (X_i * Y_i + X_i!*Y_i!)! = X_i \; xor \; Y_i.$$

Условие *равенства многоразрядных* слов X и Y:

$$Z = Z_{n-1} * Z_{n-2} * \ldots * Z_0.$$

Схемная реализация отношения "равенство" в базисе "И-НЕ" показана на рис. 14.2.

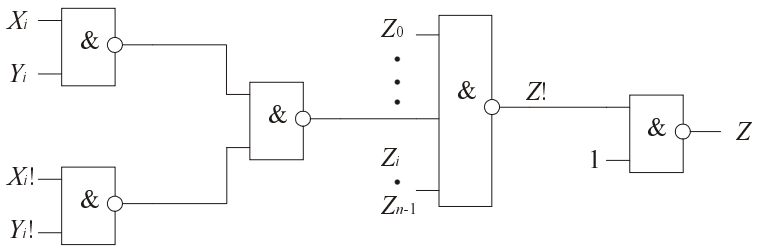


Рис. 14.2. Реализация отношения "равенство" в базисе "И-НЕ"

Если применить преобразование

$$Z_i = (X_i * Y_i! + X_i!*Y_i)! = (X_i * (X_i! + Y_i!) + (X_i! + Y_i!) * Y_i)! = (X_i * (X_i * Y_i)! + Y_i * (X_i * Y_i)!)!,$$

то возможна реализация без парафазных (прямого и инверсного сигналов вместе) входов (рис. 14.3).

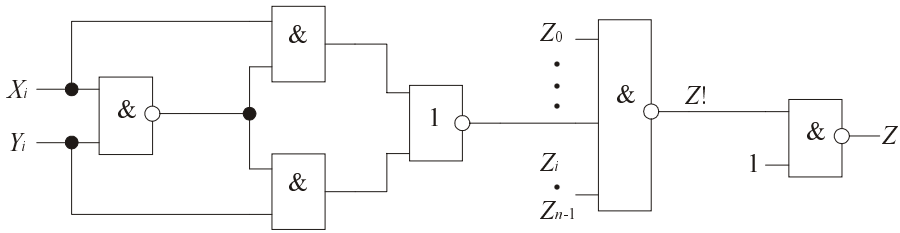


Рис. 14.3. Реализация отношения "равенство" без инверсных входных сигналов

14.2.2. Реализация отношения "больше"

Условие отношения "больше" для одноименных разрядов:

$$F_{x>y} = X_i * Y_i!.$$

Для двухразрядных слов $X = X1\,X0$ и $Y = Y1\,Y0$ условие отношения "больше" выглядит так:

$$F_{x>y} = X1 * Y1! + Z1 * X0 * Y0!,$$

где $Z1$ — условие равенства старших разрядов двухразрядных слов.

Для многоразрядных слов условие отношения "больше", по аналогии, выглядит следующим образом:

$$F_{x>y} = X_{n-1} * Y_{n-1}! + Z_{n-1} * X_{n-2} * Y_{n-2}! +$$

$$+ \ldots + Z_{n-1} * Z_{n-2} * \ldots * Z_1 * X0 * Y0!.$$

Рассмотрим пример: нужно спроектировать СМР для двоичных слов из двух разрядов со входами наращивания разрядности (рис. 14.4).

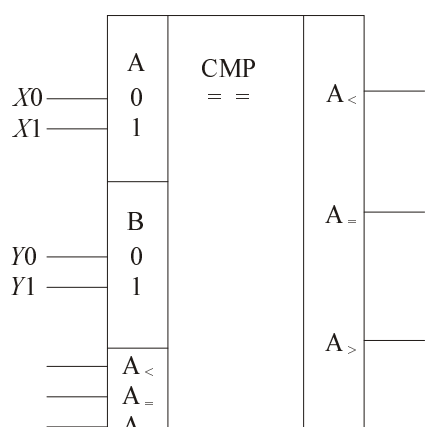


Рис. 14.4. УГО СМР для двоичных слов из двух разрядов с входами наращивания разрядности

Для реализации используем следующие переключательные формулы:

- условие равенства: $Z_i = (X_i * Y_i! + X_i! * Y_i)! = (X_i \; xor \; Y_i)!$;
- условие $X > Y$ с учетом увеличения разрядности:

$$F_{i\,X>Y} = X_i * Y_i! + Z_i * X_{i-1} * Y_{i-1}! + F_{i-1\,X>Y} * Z_i * Z_{i-1} =$$

$$= ((X_i * Y_i)! * (Z_i * X_{i-1} * Y_{i-1}!)! * (F_{i-1\,X>Y} * Z_i * Z_{i-1})!)!$$

В соответствии с логическими функциями изобразим схему (рис. 14.5).

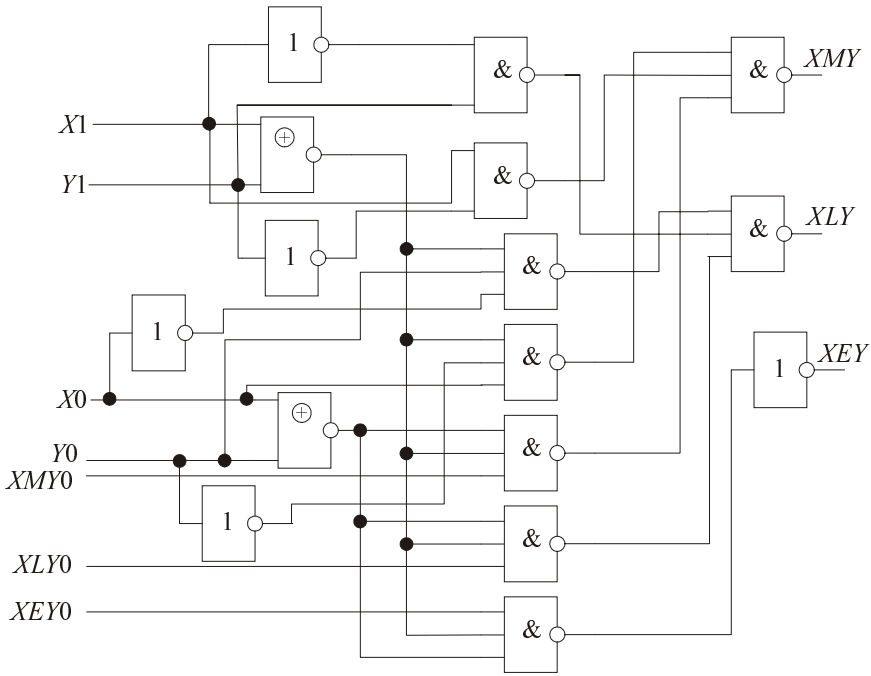


Рис. 14.5. Схема СМР для двоичных слов из двух разрядов с входами наращивания разрядности

Здесь: M (More) — больше, L (Less) — меньше, E (Equal) — равный, то есть выход XMY — $X > Y$, выход XLY — $X < Y$ и выход XEY — $X = Y$.

Входы $XMY0$, $XLY0$ и $XEY0$ необходимы для увеличения разрядности, то есть если сравниваются не двухбитные, а четырехбитные слова, то на эти входы подаются сигналы с соответствующих выходов такого же двухбитного компаратора, который сравнивает два младших разряда, тогда как рассматриваемый компаратор сравнивает два старших разряда. По аналогии можно сравнивать, например, 16- и 32-разрядные слова и т. д.

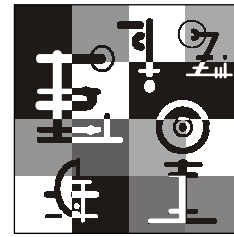
Лабораторная работа. Исследование функционирования логической схемы компаратора для сравнения двухбитных слов

Цель работы

Компараторы определяют отношения между двумя словами ("равно", "больше", "меньше"). Эти отношения используются как логические условия в микропрограммах, в устройствах контроля и диагностики ЭВМ и т. п. В данной лабораторной работе исследуется функционирование схемы компаратора с тремя выходами: "равно", "больше" и "меньше".

Программа работы

1. Исследуйте работу рассмотренного в данной главе компаратора для двоичных слов из двух разрядов со входами наращивания разрядности. Создайте проект, используя графический редактор, и откомпилируйте его. С помощью редактора временных диаграмм задайте различные комбинации двухбитных слов, чтобы можно было проверить правильность функционирования схемы. Получите временные диаграммы.
2. Создайте символ для компаратора, чтобы использовать его в следующем проекте.
3. Создайте новый проект и увеличьте разрядность компаратора в два раза, используя входы $XMY0$, $XLY0$ и $XEY0$. Проверьте правильность работы схемы, задав различные комбинации сравниваемых слов.
4. Разработайте поведенческую модель компаратора для сравнения двухбитных слов на VHDL.
5. Используйте ее в структурной модели компаратора с увеличенной разрядностью.
6. Получите временные диаграммы и сравните их с диаграммами, полученными в п. 3.
7. По аналогии с синтезируемым VHDL-кодом DC 3-8 (лабораторная работа из *главы 11*) напишите синтезируемый VHDL-код компаратора для двоичных слов из двух разрядов со входами наращивания разрядности. Проверьте его на синтезируемость, для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Глава 15

Устройства недопущения, обнаружения и исправления ошибок

Рассмотрим ряд схем, применяемых в устройствах недопущения, обнаружения и исправления ошибок.

15.1. Схемы контроля по модулю 2 (схемы свертки по модулю 2)

Приведем определения из теории передачи информации (ТПИ).

- *Кодовой комбинацией* будем называть ряд из символов словаря логической переменной (ЛП): "0", "1" и т. д. Это буква, цифра, знаки препинаний, отношений, знаки арифметических, алгебраических, логических действий.
- *Код* — совокупность кодовых комбинаций для представления информации. Это слова, выражения, программы, подпрограммы, программные пакеты, описание объектов, их поведения, их конфигурации.
- *Кодовое расстояние* S между двумя кодовыми комбинациями — это число разрядов, в которых эти комбинации отличаются друг от друга.
- *Минимальное кодовое расстояние* $S_{\min}$ — минимальное расстояние для любой пары кодовых комбинаций.
- *Кратность ошибки* E — это число неверных разрядов.
- *Вес кодовой комбинации* — это число единиц данной комбинации.

Рассмотрим ряд условий, без которых обнаружение и исправление ошибок при кодировании невозможно:

1. Условие для обнаружения ошибки: $S_{\min} = E_{обн} + 1$;
2. Условие для исправления ошибки: $S_{\min} = 2 * E_{испр} + 1$, где $E_{обн}$ и $E_{испр}$ — кратность обнаруженных и исправленных ошибок.

Применяемый нами ранее обычный двоичный код имеет $S_{\min} = 1$, поэтому обнаружение и исправление ошибок невозможно, для этого $S_{\min}$ должно быть больше или равно 2.

Для контроля по модулю 2 $S_{\min}$ должно быть равно 2, то есть к информационным разрядам добавляется дополнительный (контрольный) разряд и при ошибочном изменении одного информационного разряда меняется и дополнительный, поэтому $S_{\min} = 2$.

При контроле по модулю 2 все информационные символы (буквы, цифры и т. д.) дополняются контрольным разрядом ($E_ч$ или $E_н$), значение которого выбирается так, чтобы сделать вес каждой кодовой комбинации четным (при контроле по четности — $E_ч$) или нечетным (при контроле по нечетности — $E_н$).

15.1.1. Проектирование схем контроля по модулю 2

На первом этапе спроектируем схему контроля по четности, на выходе которой вырабатывается значение сигнала $E_ч$, и схему контроля по нечетности с выходным сигналом $E_н$ для двухразрядного слова: $X = x1x0$.

1. Таблица истинности (табл. 15.1).

Таблица 15.1. *Таблица истинности схемы контроля по четности и нечетности для двухразрядного слова*

x1	x0	Еч	Ен
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

ЛФ в СДНФ:

$$Eч = x1!*x0 + x1*x0! = x0\ XOR\ x1$$

$$Eн = x1!*x0! + x1*x0 = (x0\ XOR\ x1)!$$

Логическая схема (рис. 15.1).

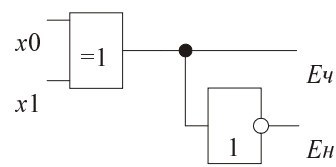


Рис. 15.1. Схема контроля по четности и нечетности для двухразрядного слова

На втором этапе, по аналогии, напишем ЛФ для схемы контроля по четности ($E_ч$) и нечетности ($E_н$) четырехразрядного информационного символа $X = x3x2x1x0$:

$$Eч = x3XOR\ x2XOR\ x1XOR\ x0$$

$$Eн = (x3XOR\ x2XOR\ x1XOR\ x0)! .$$

Таким образом, при считывании символа с контрольным разрядом ($E_ч$ или $E_н$) из памяти или после его передачи по магистралям производится контроль на четность (или нечетность) соответствующей кодовой комбинации. При изменении четности (или нечетности) веса кодовой комбинации операция считается ошибочной.

Логическая схема контроля по модулю 2 (или свертки по модулю 2) может быть пирамидального (рис. 15.2) и последовательного (рис. 15.3) типов.

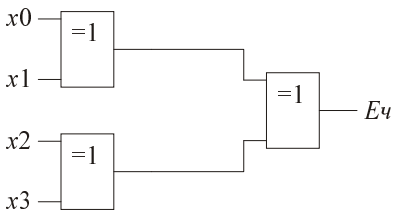


Рис. 15.2. Схема контроля по модулю 2 пирамидального типа

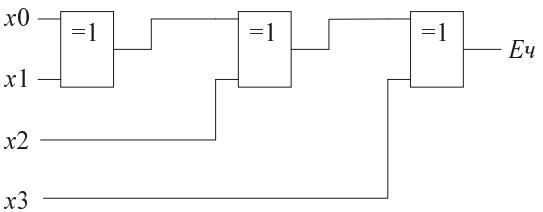


Рис. 15.3. Схема контроля по модулю 2 последовательного типа

15.1.2. ИС КР1533ИП5 — схема контроля по модулю 2

Рассмотрим ИС КР1533ИП5 (рис. 15.4). Это пирамидальная структура из трехвходовых элементов типа четность/нечетность. Она допускает свертку байта с девятым контрольным разрядом и имеет два выхода: E (Even — четный) и O (Odd — нечетный).

Если на выходе E — "1", а на O — "0", то вес четный, если наоборот, то вес нечетный.

Значения E и O — это признаки четности и нечетности, соответственно. Их нельзя отождествлять со значениями $E_н$ и $E_ч$.

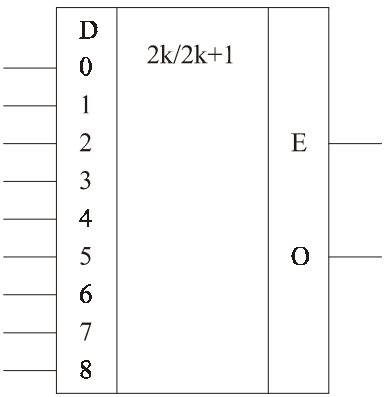


Рис. 15.4. ИС КР1533ИП5

Пример использования ИС КР1533ИП5

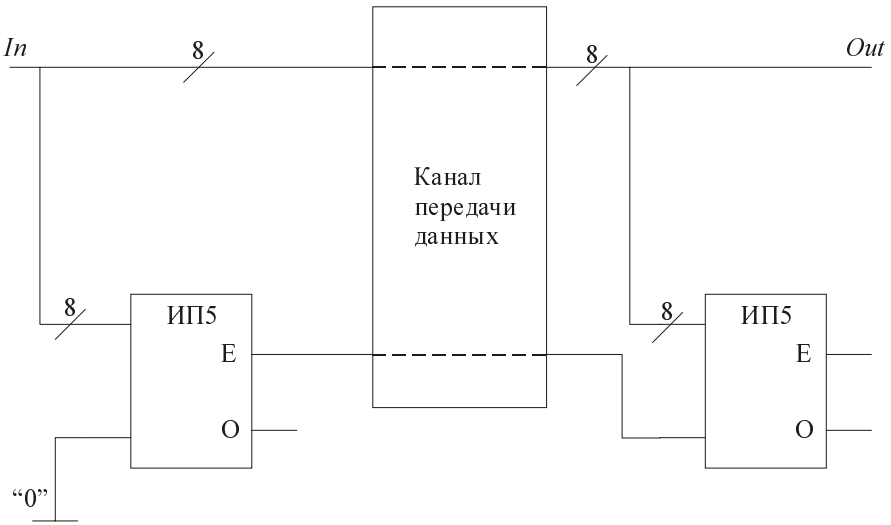


Рис. 15.5. Схема использования ИС КР1533ИП5

На вход (In) и в канал передачи поступает байт (8 бит) (рис. 15.5), он же поступает на первый ИП5, который работает следующим образом:

- если вес байта четный, то на выходе Even — "1",
- если вес байта нечетный, то на выходе Even — "0".

Таким образом, поступающие в канал передачи 9 бит всегда будут иметь нечетный вес и, если канал работает без ошибок, то на выходе E второго ИП5 всегда будет "0".

Появление "1" на выходе E второго ИП5 означает, что в канале передачи произошла ошибка и передачу нужно повторить.

15.2. Схемы контроля, использующие мажоритарные элементы или элементы "голосования" (спецификация простейшего мажоритарного элемента)

На одном из выходов мажоритарного элемента (МЭ) имеем такое значение сигнала, которое соответствует большинству входных сигналов, а их, как правило, три. На других выходах расположен двоичный адрес входа, на который поступил сигнал, отличающийся от большинства.

15.2.1. Проектирование мажоритарного элемента

1. Таблица истинности (табл. 15.2).

Таблица 15.2. *Таблица истинности мажоритарного элемента*

Входы			**Выходы**		
X1	**X2**	**X3**	**X**	**a1**	**a0**
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	1	0	0

2. Переключательные функции в СДНФ:

$X = X1 * X2 + X1 * X3 + X2 * X3$;

$a1 = X2\ XOR\ X3$;

$a0 = X1\ XOR\ X3$.

Логическую схему МЭ изобразите и протестируйте в соответствующей лабораторной работе.

Лабораторная работа. Исследование функционирования схем контроля по модулю 2 и схемы мажоритарного элемента

Цель работы

В данной лабораторной работе исследуется функционирование схемы контроля четности/нечетности на 9 входов (аналог КР1533ИП5). Под четностью/нечетностью понимается четность/нечетность количества единиц в двоичном слове из девяти разрядов. Такие схемы применяются в аппаратуре обнаружения ошибок при передаче данных.

Программа работы

1. Исследуйте работу схемы контроля на четность/нечетность для двухразрядного слова с третьим контрольным входом, допускающую свертку по модулю 2. Создайте проект, используя графический редактор, и откомпилируйте его. С помощью редактора временных диаграмм задайте различные комбинации двухбитного слова, чтобы можно было проверить правильность функционирования схемы. Получите временные диаграммы.
2. Создайте символ для этой схемы, чтобы использовать его в следующем проекте.
3. Используйте этот символ и пирамидальную структуру для получения схемы контроля четности/нечетности восьмибитного слова с девятым контрольным входом, рассмотренную в данной главе. Создайте символ для этой схемы, аналогичный изображенному на рис. 15.6. Проверьте правильность работы схемы, задав различные комбинации входных битов.

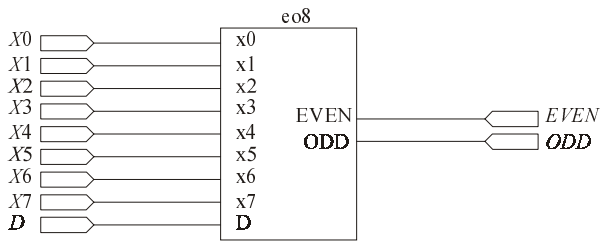


Рис. 15.6. Элемент контроля четности/нечетности (аналог КР1533ИП5)

Исследуйте применение элемента контроля четности/нечетности в схеме обнаружения ошибок при передаче данных. Примерный вариант схемы приведен на рис. 15.7. Схема должна быть протестирована как на данных, содержащих ошибки, так и без них. Для этого нужно придумать и реализовать элемент Data Transfer Module. Приведите полученные временные диаграммы и поясните принцип работы схемы.

Напишите поведенческую модель схемы контроля четности/нечетности на VHDL.

Используйте ее в структурной модели примера использования схем контроля при обнаружении ошибок. Получите временные диаграммы и сравните их с диаграммами, полученными в п. 4.

Исследуйте работу мажоритарного элемента на три входа, рассмотренного в данной главе. Создайте графический проект, протестируйте (получите временные диаграммы) и поясните принцип работы схемы.

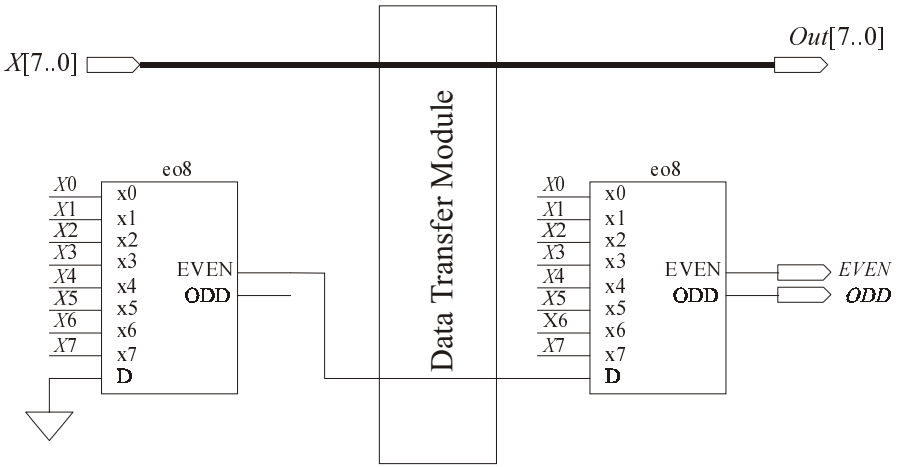
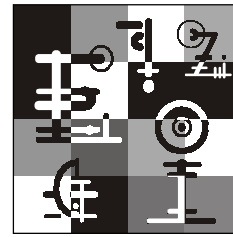


Рис. 15.7. Пример использования схем контроля четности/нечетности при обнаружении ошибок

Напишите и протестируйте поведенческую модель работы схемы мажоритарного элемента на VHDL. Полученные временные диаграммы сравните с диаграммами из п. 7.

По аналогии с синтезируемым VHDL-кодом DC 3-8 (лабораторная работа в *главе 11*) напишите синтезируемый VHDL-код схемы контроля четности/нечетности восьмибитного слова с девятым контрольным входом. Проверьте его на синтезируемость, для этого повторите действия, ранее описанные для D-триггера, получите самостоятельно RTL-представление, представление в технологическом базисе и таблицу использования ресурсов микросхемы.



Глава 16

Кодер и декодер Хемминга

Устройство, состоящее из кодера и декодера Хемминга, относится к устройствам недопущения, обнаружения и исправления ошибок.

Рассмотрим спецификацию на проектирование этого устройства. Более подробно основа кодера и декодера Хемминга освещается в теории передачи информации.

16.1. Код Хемминга

Код Хэмминга позволяет обнаружить и исправить возможные одиночные ошибки.

В рассматриваемом коде из N позиций M используются для информации, а K в качестве контроля.

Все информационные разряды разбиваются на контрольные группы, при этом за каждой группой закрепляется контрольный разряд.

Контрольные разряды в коде Хэмминга формируются как "исключающее или" или "XOR" информационных разрядов, т. е. дополнение до четного числа единиц.

Код Хемминга создается в кодере Хемминга.

В приемном устройстве или декодере Хемминга производится K проверок на четность всех контрольных групп.

После каждой проверки в регистр ошибок записывается:

- 0 — если результат проверки четный, нет ошибок;
- 1 — если результат проверки нечетный, наличие ошибки в группе.

Полученная последовательность единиц и нулей в регистре ошибок образует двоичное слово, которое называют *синдромом ошибки*; оно указывает номер позиции в слове, в котором искажен символ.

Определение числа контрольных разрядов, исходя из числа информационных, осуществляется в соответствии с табл. 16.1. Здесь **M** — число информационных разрядов; **N** — общее число разрядов; **K** — число контрольных разрядов.

Таблица 16.1. *Таблица соответствия числа контрольных разрядов числу информационных разрядов*

M	1	2	3	4	5	6	7	8	9	10	11	12	13
N	3	5	6	7	9	10	11	12	13	14	15	17	18
K	2	3	3	3	4	4	4	4	4	4	4	5	5

16.1.1. Пример четырехразрядного кодирования Хемминга

Методика разбиения числа на контрольные группы:

Первая контрольная группа охватывает все разряды кода, номера которых в двоичной системе имеют единицу в первом разряде, т. е. 1, 3, 5, 7;

Вторая группа — с единицей во втором разряде, т. е. 2, 3, 6, 7;

Третья группа — с единицей в третьем разряде, т. е. 4, 5, 6, 7;

Разряды, имеющие степень двойки, — контрольные, т. е. 1, 2, 4.

Пусть $D0$, $D1$, $D2$, $D3$ — информационные разряды кода Хемминга, а $K1$, $K2$, $K3$, K — контрольные разряды кода Хемминга.

Контрольный разряд K проверяет четность всего закодированного слова и позволяет определить наличие двойной ошибки совместно с контрольными разрядами кода Хэмминга.

Код с таким контрольным разрядом называют *модифицированный код Хэмминга.*

Кодирование:

$$K1 = D0\ XOR\ D1\ XOR\ D3$$

$$K2 = D0\ XOR\ D2\ XOR\ D3$$

$$K3 = D1\ XOR\ D2\ XOR\ D3$$

$$K = K1\ XOR\ K2\ XOR\ D0\ XOR\ K3\ XOR\ D1\ XOR\ D2\ XOR\ D3$$

Результаты кодирования информационного слова **1101** представлены в табл. 16.2.

Таблица 16.2. *Кодирование информационного слова* ***1101***

Номер разряда	8	7	6	5	4	3	2	1
Номер разряда в двоичной форме	1000	111	110	101	100	011	010	001
Обозначение разрядов	K	D3	D2	D1	K3	D0	K2	K1
Пример кодирования	0	1	1	0	0	1	1	0

16.1.2. Пример четырехразрядного декодирования Хемминга

При декодировании определяется синдром ошибки или двоичное слово: KP, $P3$, $P2$, $P1$.

Декодирование:

$$P1 = K1\ XOR\ D0\ XOR\ D1\ XOR\ D3$$

$$P2 = K2\ XOR\ D0\ XOR\ D2\ XOR\ D3$$

$$P3 = K3\ XOR\ D1\ XOR\ D2\ XOR\ D3$$

$$KP = K1\ XOR\ K2\ XOR\ D0\ XOR\ K3\ XOR\ D1\ XOR\ D2\ XOR\ D3\ XOR\ K$$

Рассмотрим некоторые возможные состояния после декодирования и как их необходимо понимать (табл. 16.3).

Таблица 16.3. *Ряд значений синдрома ошибки*

KP	P3	P2	P1	Пояснение
0	0	0	0	Нет ошибок
0	0	1	1	Двойная ошибка, разряды неизвестны
1	0	0	0	Ошибка в главном контрольном разряде
1	0	1	1	Одиночная ошибка в 3 разряде, исправляется

16.2. Техническая реализация кодера и декодера Хемминга

В регистре ошибок хранится номер ошибочного разряда или синдром ошибки. Это позволяет реализовать аппаратную коррекцию одиночной ошибки.

При проектировании схему можно разбить на несколько законченных модулей.

- **Модуль кодирования** осуществляет формирование четырех контрольных разрядов. Модуль является полностью комбинационным, т. е. используются только логические элементы (без триггеров и регистров), и к тому же однонаправленным.
- **Модуль декодирования** выполняет более сложную задачу: формирует четыре разряда регистра ошибок, сигналы которого поступают на дешифратор, соответствующий выход которого показывает ошибочный разряд.

Автоматической коррекции подвергаются только разряды данных; при этом, если ошибка появляется в контрольных разрядах, то она просто игнорируется.

Модуль имеет два выхода для обозначения ошибок:

- *ONE_ERR* — если на этом выходе "1", то была одиночная ошибка, и она исправлена;
- *DBL_ERR* — если на этом выходе "1", то была двойная ошибка. В результате имеем сбой в работе модуля декодирования.

Рассмотренные два главных модуля объединяются в итоговой схеме с помощью буферов "с тремя состояниями на выходе" для обеспечения двунаправленного обмена.

Лабораторная работа. Исследование функционирования логической схемы кодера и декодера Хемминга

Цель работы

Основная цель данной работы спроектировать и исследовать работу схемы четырехразрядного кодировщика двоичных данных в код Хэмминга и устройства, выполняющего обратное преобразование.

Модуль кодирования осуществляет формирование четырех контрольных разрядов.

Модуль декодирования формирует три разряда регистра ошибок, сигналы которого поступают на дешифратор, соответствующий выход которого показывает ошибочный разряд.

Одиночные ошибки, возникающие при передаче данных, исправляются.

Автоматической коррекции подвергаются только разряды данных (если ошибка появляется в контрольных разрядах, то она игнорируются).

Программа работы

1. Используя материал данной главы и приведенные в нем логические функции для контрольных разрядов, спроектируйте схему кодера Хемминга. Создайте проект, используя графический редактор, откомпилируйте и протестируйте его. Получите временные диаграммы.

2. Создайте символ для кодера Хемминга, чтобы использовать его в следующем проекте.
3. Спроектируйте декодер Хемминга, используя приведенные в лекции логические функции для синдрома ошибки.

   Модуль должен иметь два выхода для обозначения ошибок:

   - *ONE _ ERR* — одиночная ошибка, исправлена;
   - *DBL _ ERR* — двойная ошибка, сбой в декодировании.

   Проверьте правильность работы схемы, задав различные комбинации кодированных слов.
4. Создайте проект примера, реализующего функции кодирования и декодирования информации с использованием модифицированного кода Хемминга.

   Ориентировочная схема приведена на рис. 16.1.

   В примере должна быть предусмотрена возможность внесения ошибок в передаваемый код. Для этого необходимо разработать специальный модуль Data Transformer Module.

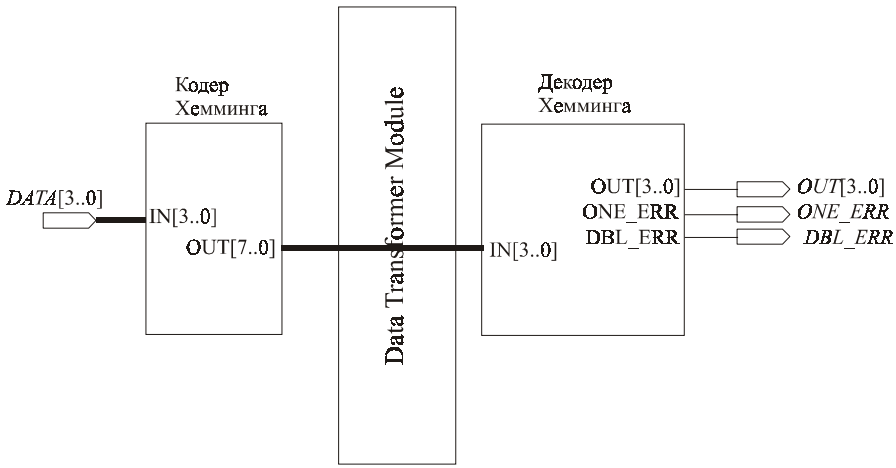
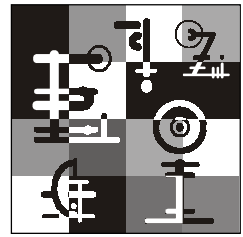


Рис. 16.1. Схема тестового примера работы кодера и декодера Хемминга

5. Разработайте поведенческие модели на VHDL для каждого из модулей, рассмотренных в пп. 1 и 3.
6. Используйте их в структурной модели примера из п. 4.
7. Получите временные диаграммы и сравните их с диаграммами, полученными в п. 4.



Глава 17

Многоразрядные сумматоры, арифметико-логические устройства (АЛУ) и умножители

Рассмотрим устройства, реализующие арифметические и логические операции над многоразрядными двоичными словами. Будем считать операции вычитания и деления производными от рассматриваемых операций.

17.1. Многоразрядные сумматоры

На основе одноразрядного сумматора (СМ) могут строиться четыре типа многоразрядных сумматоров.

С точки зрения учета предыстории СМ, как и все ЛУ, делятся на *комбинационные* и *последовательностные* (они же накапливающие или автоматы с памятью).

СМ, как и все ЛУ, строятся либо по *параллельному*, либо по *последовательному* способам обработки многоразрядных чисел.

17.1.1. Комбинационный сумматор параллельного действия

Здесь (рис. 17.1) столько СМ, сколько разрядов в двоичных числах X и Y. Все разряды этих чисел суммируются одновременно, то есть разряды чисел X и Y подаются на соответствующие входы одновременно и держатся до окончания распространения переноса.

Переносы обрабатываются во времени последовательно, поэтому такое распространение переноса называют последовательным.

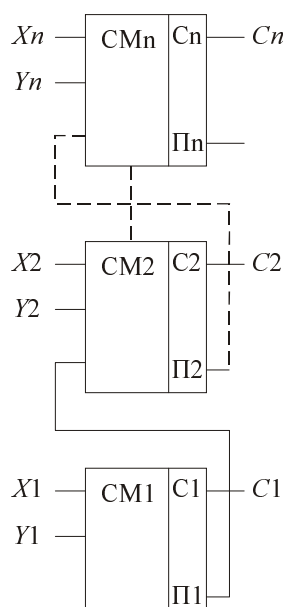


Рис. 17.1. Схема комбинационного сумматора параллельного действия

Рассмотрим пример, при котором потребуется максимальное время сложения:

$$\begin{array}{r} 11111111 \\ + \\ 00000001 \\ \hline 11111110 \\ + \\ 1 \\ \hline 11111100 \\ + \\ 1 \\ \hline \end{array}$$ и так далее.

Максимальное время сложения такого сумматора будет:

$T_{см} = n * T_n$,

где n — количество разрядов, T_n — время образования и распространения переноса между двумя соседними разрядами СМ.

17.1.2. Комбинационный сумматор последовательного действия

В данном случае имеем один одноразрядный сумматор (рис. 17.2). На его вход поочередно поступают разряды слагаемых, начиная с первого разряда. Перенос, образовавшийся при сложении какого-либо разряда, задерживается (H) на такой же промежуток времени (T_n), как и промежуток между подачей разрядов числа, и таким образом суммируется со следующим разрядом. Получаемая сумма поступает в регистр хранения.

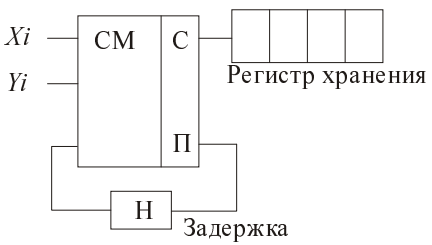


Рис. 17.2. Схема комбинационного сумматора последовательного действия

Время сложения такого СМ всегда одинаково и равно $T_{см} = n * T_н$. Оно больше, чем у комбинационного сумматора параллельного действия.

17.1.3. Накапливающий сумматор параллельного действия

Здесь на вход подается одно число, а второе подается с триггеров, в которые оно заранее было записано (рис. 17.3).

Максимальное время сложения такого сумматора аналогично комбинационному сумматору параллельного действия и будет равно $T_{см} = n * T_n$.

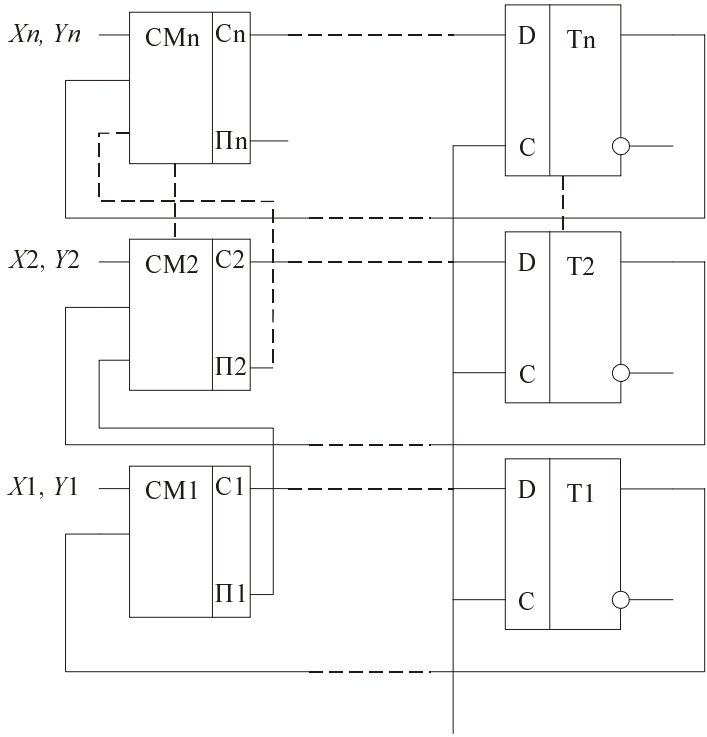


Рис. 17.3. Схема накапливающего сумматора параллельного действия

17.1.4. Накапливающий сумматор последовательного действия

Максимальное время сложения такого сумматора (рис. 17.4) будет равно $T_{см} = n * T_н$ (аналогично комбинационному сумматору последовательного действия).

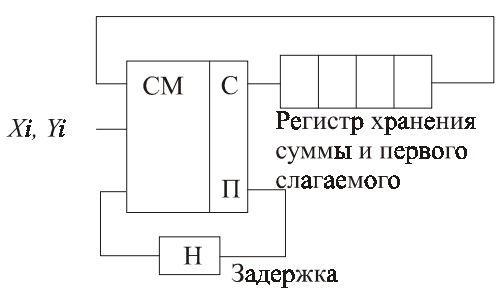


Рис. 17.4. Схема накапливающего сумматора последовательного действия

17.1.5. Схемные методы ускорения распространения переноса в многоразрядных параллельных сумматорах

Операция суммирования (сложения) является базовой, ее используют все остальные арифметические операции, поэтому важно ускорить ее выполнение. Рассмотрим три метода ускорения распространения переноса: сквозной последовательный, параллельный и групповой. Последовательный перенос был описан на примере работы сумматоров параллельного действия.

Сквозной последовательный перенос

Рассмотрим одноразрядный СМ из двух полусумматоров (ПС) — рис. 17.5.

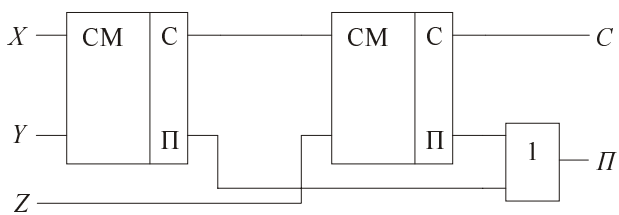


Рис. 17.5. Одноразрядный СМ из двух полусумматоров

В этом случае логическая функция (ЛФ), описывающая СМ, выглядит так:

$C = C_{ПС} * Z! + C_{ПС}! * Z$,

$П = П_{ПС} + C_{ПС} * Z$,

где ЛФ для первого ПС:

$C_{ПС} = X * Y! + X! * Y$,

$П_{ПС} = X * Y$.

Из анализа ЛФ, описывающих одноразрядный СМ, видно, что перенос получается на одну операцию раньше, чем сумма.

Назовем операцию "И" в ЛФ для переноса ($П$) в СМ ($C_{ПС} * Z$) *функцией генерации (GN)* для одного разряда, а операцию "ИЛИ" ($П_{ПС} + C_{ПС} * Z$) *функцией прозрачности (PN)* для одного разряда.

Здесь, если предыдущий перенос $Z = 0$, то из ЛФ для СМ имеем: $C = C_{ПС}$ и $П = П_{ПС}$, то есть результат можно снимать с первого ПС, не дожидаясь срабаты-

вания второго ПС. Если же $Z = 1$, то нужно ждать окончания работы второго ПС и результат снимать с него.

Идея сквозного переноса заключается в том, чтобы выявить, где произойдет перенос (где он будет равен "1"), и только там ждать срабатывания второго ПС для получения окончательной суммы.

Если переносов нет, то получение суммы определяется окончанием распространения переноса в дополнительной схеме и снятием суммы с первых ПС.

В дополнительной схеме вырабатываются переносы с использованием функций генерации (GN) и прозрачности (PN).

Эта дополнительная схема получила название *блок ускоренного переноса* (Carry Unit — CRU).

Параллельный перенос

Если сделать многоразрядный СМ, где сложение выполняется поразрядно и без распространения переноса, то быстродействие увеличится еще больше.

Реализация такого СМ возможна, если на входы каждого разряда СМ для образования и учета предыдущего переноса будут подаваться все цифры слагаемых предыдущих разрядов.

Например, перенос в третьем разряде будет определяться следующей формулой:

$$\Pi 3 = X3 * Y3 + \left(X3 * Y3! + X3! * Y3\right) * \left(X2 * Y2 + \left(X2 * Y2! + X2! * Y2\right) * X1 * Y1\right) =$$

$$= PN3 = X3 * Y3 + GN3.$$

Здесь:

$$X1 * Y1 = GN1 = PN1;\ \left(X2 * Y2! + X2! * Y2\right) * PN1 = GN2;\ \left(X2 * Y2 + GN2\right) = PN2;$$

$$\left(X3 * Y3! + X3! * Y3\right) * PN2 = GN3;\ \left(X3 * Y3 + GN3\right) = PN3 = \Pi 3.$$

При организации параллельного переноса используется блок ускоренного переноса (CRU).

Групповой перенос

При групповом переносе разряды СМ разбиваются на группы: внутри этих групп применяется последовательный перенос, а между группами параллельный (или наоборот).

СМ с групповым переносом — это компромисс между сумматорами, использующими сквозной последовательный и параллельный переносы.

17.2. Арифметико-логические устройства и блоки ускоренного переноса

В основе арифметико-логического устройства (ALU) лежит схема многоразрядного сумматора с дополнительной логикой.

Рассмотрим четырехразрядное ALU (74181) — рис. 17.6.

Здесь A и B — два четырехразрядных двоичных числа;

S — код операции, выполняемой ALU;

CN — инверсный входной перенос;

M (Mode) — задает тип операции ($M = 1$ — логическая операция, $M = 0$ — арифметико-логическая операция);

F — результат операции ALU;

GN и PN — функции генерации и прозрачности, используемые при организации сквозного последовательного и параллельного переносов при наращивании размерности ALU;

$CN4$ — инверсный выходной перенос.

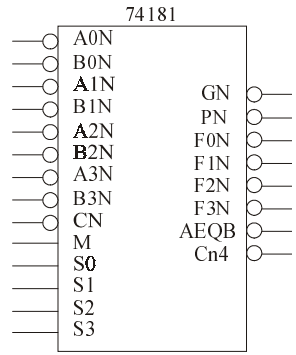


Рис. 17.6. Схема ALU (74181)

Выход $AEQB$ — выход сравнения на равенство чисел A и B.

Операции, выполняемые ALU, сведены в табл. 17.1.

Таблица 17.1. *Таблица операций ALU (74181)*

				M=H	**M=L. Арифметические операции**	
S3	**S2**	**S1**	**S0**	**Логические функции**	**/Cn=H (без переноса)**	**/Cn=L (с переносом)**
L	L	L	L	F=/A	F=A	F=A plus 1
L	L	L	H	F=/(A+B)	F=A+B	F=(A+B) plus 1
L	L	H	L	F=(/A)B	F=A+/B	F=(A+/B) plus 1
L	L	H	H	F=0	F= minus 1 (2s Comp)	F= ZERO
L	H	L	L	F=/(AB)	F=A plus A(/B)	F=A plus A(/B) plus 1
L	H	L	H	F=/B	F=(A+B) plus A(/B)	F=(A+B) plus A(/B) plus 1
L	H	H	L	F=A $ B	F=A minus B minus 1	F=A minus B

Таблица 17.1 *(окончание)*

				M=H	M=L. Арифметические операции	
S3	**S2**	**S1**	**S0**	**Логические функции**	**/Cn=H (без переноса)**	**/Cn=L (с переносом)**
L	H	H	H	F=A(/B)	F=A(/B) minus 1	F=A(/B)
H	L	L	L	F=/A+B	F=A plus AB	F=A plus AB plus 1
H	L	L	H	F=/(A $ B)	F=A plus B	F=A plus B plus 1
H	L	H	L	F=B	F=(A+/B) plus AB	F=(A+/B) plus AB plus 1
H	L	H	H	F=AB	F=AB minus 1	F=AB
H	H	L	L	F=1	F=A plus A	F=A plus A plus 1
H	H	L	H	F=A+/B	F=(A+B) plus A	F=(A+B) plus A plus 1
H	H	H	L	F=A+B	F=(A+/B) plus A	F=(A+/B) plus A plus 1
H	H	H	H	F=A	F=A minus 1	F=A

Здесь:

"1" — "1111";

"0" — "0000";

"CN" — "000CN".

Логические операции ("+", "/", "$" и др.) выполняются поразрядно, а арифметические (plus, minus и др.) — с учетом межразрядных переносов.

В случае одновременного выполнения логических и арифметических операций сначала поразрядно выполняются инвертирование, логические умножение и сложение, а затем арифметические операции с переносом.

При увеличении разрядности ALU можно соединять:

- с организацией последовательного переноса (рис. 17.7);
- с организацией параллельного переноса (рис. 17.8);
- ALU можно использовать для выработки сигналов сравнения слов A и B: $F_{A=B}$; $F_{A<B}$; $F_{A>B}$; с помощью выхода $AEQB$ и дополнительной, уже изученной, логики.

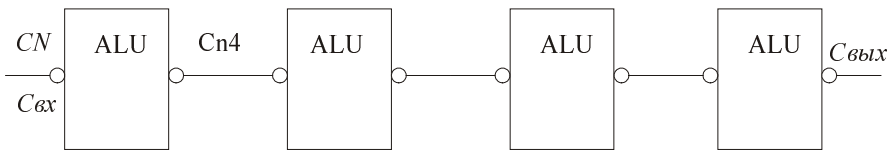


Рис. 17.7. Пример увеличения разрядности ALU с последовательным переносом

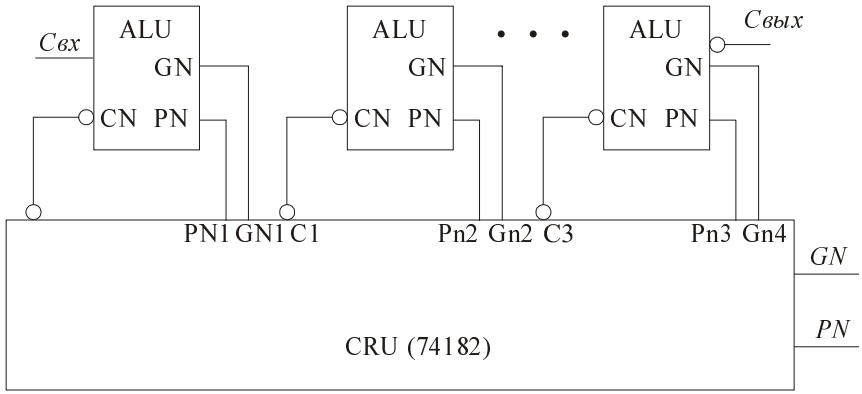


Рис. 17.8. Пример увеличения разрядности ALU с параллельным переносом

17.3. Умножители параллельного действия (матричные умножители)

Рассмотрим умножение двух четырехразрядных двоичных чисел:

$A_m = a_{m-1} \ldots a_0$ и $B_n = b_{n-1} \ldots b_0$, где $m = n = 4$:

$$
\begin{array}{cccccccc}
 & & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & * & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
+ & & & & & & & & \\
 & & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 &
\end{array}
$$

Здесь значения $a_i * b_j$ определяются конъюнкторами одновременно (параллельно во времени).

Операция сложения значений $a_i * b_j$ в столбцах и распространение переноса занимает основное время перемножения.

Описанный матричный умножитель называют *множительным блоком (МБ)* — рис. 17.9.

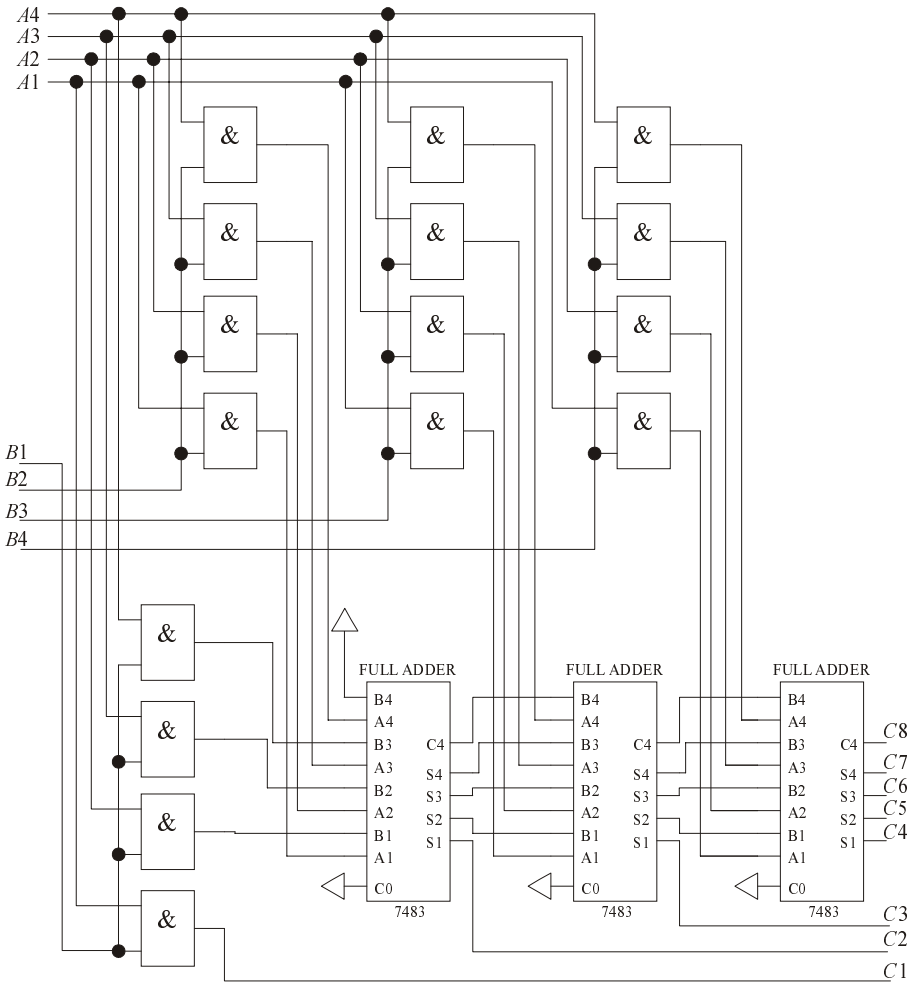


Рис. 17.9. Схема матричного умножителя четырехразрядных чисел

Если к произведению добавить два слагаемых, то есть реализовать функцию $P = A_m * B_n + C_m + D_n$, то это устройство называют *множительно-суммирующим блоком (МСБ)*.

МСБ применяется для увеличения разрядности.

17.3.1. Увеличение разрядности матричных умножителей

Рассмотрим пример увеличения разрядности с помощью МСБ.

Дано: МСБ 4*2 (рис. 17.10).

Спроектировать: МСБ 4*4.

Проектирование производится в соответствии с рассмотренным ранее умножением двух четырехразрядных двоичных чисел A и B. Разряды этих чисел подаются на соответствующие входы двух МСБ. Сдвиг с суммированием осуществляется подачей разрядов результата верхнего МСБ на соответствующие входы чисел D и C нижнего МСБ. Схема МСБ 4*4 из МСБ 4*2 представлена на рис. 17.11.

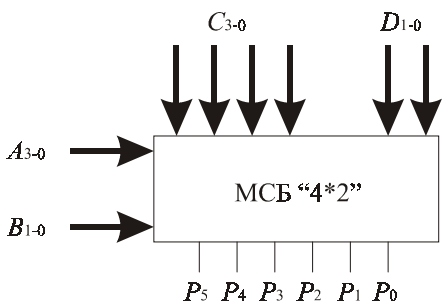


Рис. 17.10. УГО МСБ 4*2

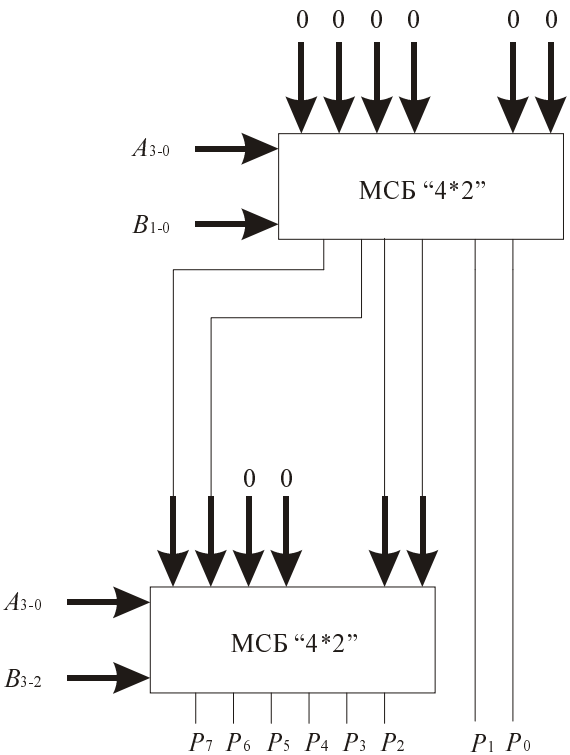


Рис. 17.11. Пример увеличения разрядности матричных умножителей

Лабораторная работа. Исследование работы сумматоров, арифметико-логических устройств и умножителей

Цель работы

Данная лабораторная работа предполагает изучение функционирования схемы полного сумматора на основе стандартного элемента библиотеки макрофункций 7483. Также ставится задача рассмотрения работы арифметико-логического устройства и умножителя.

Программа работы

1. Создав в графическом редакторе новый проект для полного четырехразрядного сумматора, изучите работу элемента 7483 САПР MAX+Plus II. Для этого используйте Help MAX+Plus II и схему этого элемента. Схему можно увидеть, если дважды щелкнуть левой кнопкой мыши в области символа элемента. Опишите работу полного сумматора и приведите временные диаграммы его работы для разных комбинаций входных слов.
2. Реализуйте схему восьмиразрядного сумматора, используя элемент 7483.
3. Исследуйте работу четырехразрядного арифметико-логического устройства (элемент библиотеки макрофункций 74181, описанный в этой главе), задав различные режимы и комбинации входных слов. На временных диаграммах должны быть представлены как арифметические, так и логические типы операций. Эти режимы задаются с помощью входа M. Код операции подается на четырехразрядный вход S ($S0$—$S3$). Проверьте работу элемента с использованием бита переноса из предыдущего разряда и без него. Более подробную информацию о работе этого элемента вы можете получить, используя Help MAX+Plus II или таблицу команд, приведенную в этой главе (табл. 17.1).
4. Спроектируйте схему двукратного увеличения разрядности АЛУ, применяя последовательный перенос. Для реализации схемы используйте элемент 74181.
5. Спроектируйте схему двукратного увеличения разрядности АЛУ, применяя параллельный перенос. Для реализации схемы используйте элементы 74181 и 74182 (Carry Unit). Сравните полученные временные диаграммы с диаграммами из п. 4.
6. Разработайте схему компаратора для сравнения четырехразрядных слов, используя элемент 74181.

7. Исследуйте работу схемы быстродействующего параллельного умножителя, рассмотренную в этой главе. Получите временные диаграммы и поясните принцип действия схемы.

8. Исследуйте в графическом редакторе работу универсального настраиваемого умножителя, изображенного на рис. 17.12. Для этого в библиотеке mega-lpm выберите элемент LPM_MULT. Настройте элемент, оставив только входы `DATAA` и `DATAB` и выход `RESULT` и установив следующие параметры:

```
INPUT_A_IS_CONSTANT          "NO"
INPUT_B_IS_CONSTANT          "NO"
LPM_PIPELINE                 <none>
LPM_REPRESENTATION           "UNSIGNED"
LPM_WIDTHA                   4
LPM_WIDTHB                   4
LPM_WIDTHP                   (LPM_WIDTHA+LPM_WIDTHB)
LPM_WIDTHS                   LPM_WIDTHA
USE_EAB                      <none>
```

   Настройку параметров можно выполнять, используя контекстное меню (**EDIT PORT | PARAMETERS**). Чтобы не засорять схему ненужной информацией, в меню **Options** отключите пункт **Show Parameters**.

Используя материал этой главы, постройте схемы множительно-суммирующего блока (МСБ) для четырехразрядных двоичных чисел (МСБ 4*2 и МСБ 4*4). Для построения этих схем создайте элемент "одноразрядный сумматор". Проверьте правильность работы схемы МСБ. Получите временные диаграммы. Создайте символ для МСБ.

Спроектируйте схему множительно-суммирующего блока, расширив разрядность одного из сомножителей до 8, то есть из МСБ 4*2 или 4*4, используя увеличение разрядности, спроектируйте МСБ 4*8. Применяйте для этого элементы, созданные в предыдущем пункте. Проверьте правильность работы схемы, получив временные диаграммы.

Проекты АЛУ и умножителя на языке VHDL будут представлены при рассмотрении VHDL-проекта процессора DP-32.

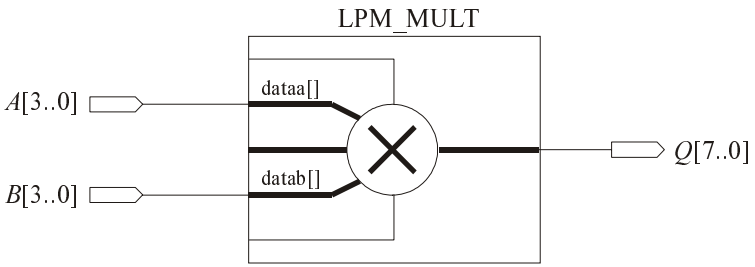
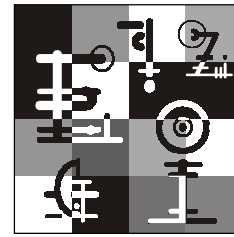


Рис. 17.12. Схема использования элемента LPM_MULT



Глава 18

Схемы памяти

Основные показатели схем памяти (микросборок или плат памяти) — это емкость (С) и быстродействие (Б). Из схем памяти строится запоминающее устройство (ЗУ).

Емкость (С) измеряется в байтах (1 байт = 8 бит). Схема памяти, как правило, состоит из 8 чипов памяти (бывает 16 или 32 чипа). Емкость чипов (или кристаллов) измеряется в битах, а микросборок и ЗУ, состоящих из микросборок, в байтах.

Быстродействие (Б) определяется временами считывания, записи и длительностями циклов (минимальный интервал между последовательными чтениями или записями).

Пусть М — эмпирическое соотношение, характеризующее производительность ЭВМ и определяющее показатель работы ЭВМ. М определяется количеством операций за секунду (оп/сек).

Как правило: М (оп/сек) ЭВМ = С (байт) ЗУ.

Из личных наблюдений автора (грубо):

- для Минск-32: Б — 40 Коп/сек, С — 40 Кбайт;
- для Р120: Б — 120 МГц (30 Моп/сек), С — 32 Мбайт;
- для PII 300: Б — 300 МГц (75 Моп/сек), С — 64—128 Мбайт;
- для Р1200: Б — 1200 МГц (300 Моп/сек), С — 256 Мбайт и т. д.

18.1. Иерархия ЗУ

Единого, общего ЗУ, одновременно удовлетворяющего максимальным требованиям по С и Б, нет. Это дорого и неэффективно, поэтому существует иерархия ЗУ (рис. 18.1).

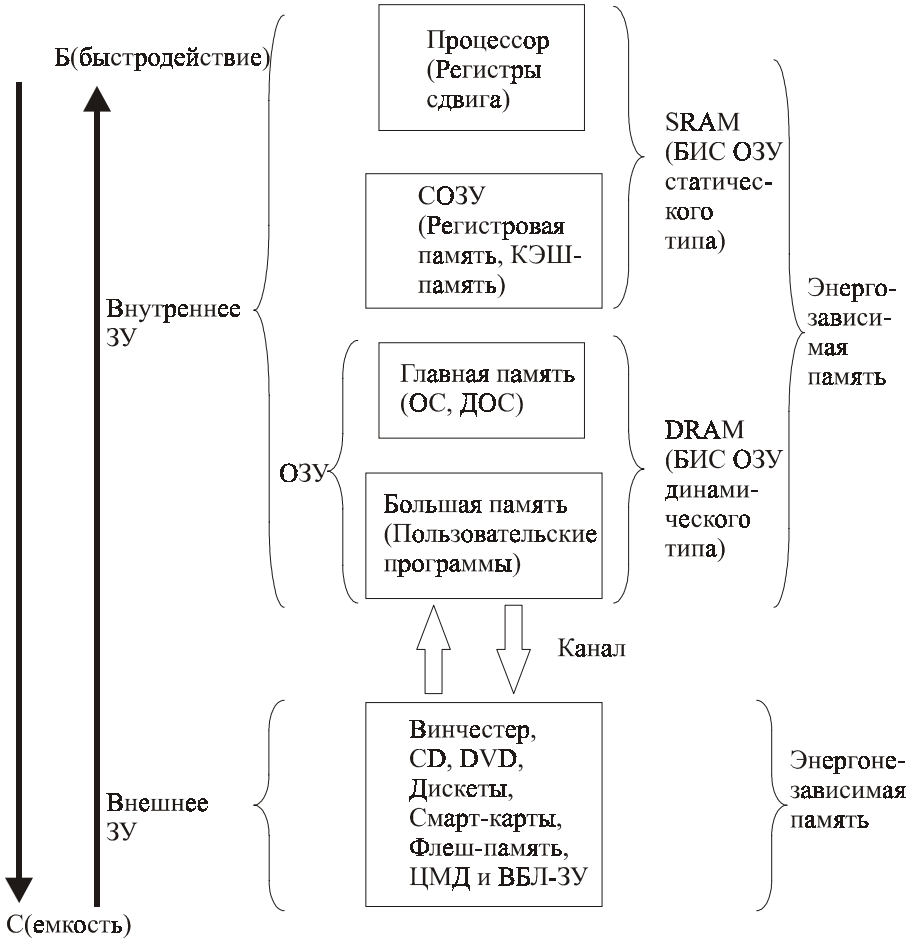


Рис. 18.1. Приблизительная структура иерархии ЗУ

18.2. Функциональная классификация ЗУ

Далее приведена расшифровка обозначений с рис. 18.2.

ROM — Read-Only Memory (ЗУ только для чтения или постоянное запоминающее устройство, ПЗУ).

PROM — Programmable ROM (программируемое ПЗУ, ППЗУ).

EPROM — Electrically PROM (стираемое программируемое ПЗУ, СППЗУ).

EEPROM — Electrically Erasable PROM (электрически стираемое программируемое ПЗУ).

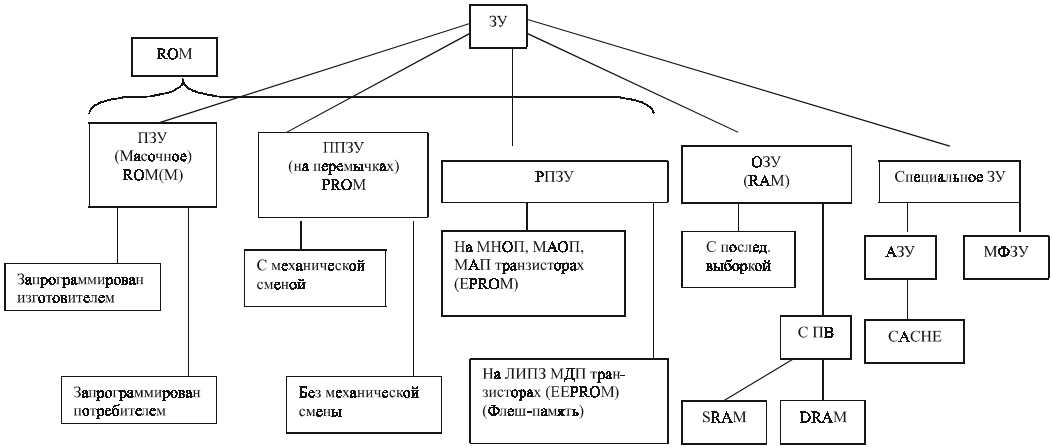


Рис. 18.2. Пример функциональной классификации ЗУ

RAM — Random Access Memory (оперативное запоминающее устройство, ОЗУ).

SRAM — Static RAM (статическое ОЗУ).

DRAM — Dynamic RAM (динамическое ОЗУ).

CACHE Memory (кэш, быстродействующая буферная память небольшой емкости).

ПЗУ — постоянное ЗУ (информация записывается один раз в заводских условиях).

ППЗУ — полупостоянное или программируемое постоянное ЗУ (информация записывается один раз в домашних условиях).

РПЗУ — репрограммируемое ЗУ (здесь запись требует гораздо больше времени и энергии, чем считывание).

ОЗУ — оперативное ЗУ (здесь информацию одинаково легко записывать и считывать).

ПВ — произвольная выборка. Если иначе, то последовательные запись и считывание. ЗУ с ПВ разбивают на участки, в каждом из которых может быть записано определенное число бит (машинное слово или ячейка памяти). Эти участки пронумерованы, а номера называют адресами. Таким образом, записываемую или считываемую информацию направляют по адресам. При ПВ каждый бит может иметь свой адрес.

АЗУ — ассоциативные ЗУ. В АЗУ поиск ячейки ЗУ ведется не по адресам, а по специальному признаку, для которого в начале слова отводится несколько разрядов.

МФЗУ — многофункциональное ЗУ. Под многофункциональностью понимают логическую обработку информации перед запоминанием.

18.3. Способы создания ЗУ

Существует несколько способов создания ЗУ.

- Создание различных участков, соответствующих информационным "0" и "1", в общей среде (магнитные ленты, диски и др.).
- Создание отдельных ЗЭ, хранящих "0" или "1" (триггеров, ЗЭ на МНОП-транзисторах и др.), объединенных в матрицу ЗЭ. Это ЗУ с ПВ.
- Устройства с непрерывным кольцевым движением информации (динамические МДП-регистры, ПЗС (приборы с зарядовой связью)).

Рассмотрим пример проектирования ЗУ.

Техническое задание: спроектировать ЗУ из 256 ЗЭ с минимальным числом выводов.

Проектирование:

1. Если ЗЭ расположить в линию, то понадобится 256 выводов (дорогой корпус, не рентабельно).
2. При матричной организации имеем 16 выводов по вертикали и 16 по горизонтали. Каждый вывод подключен к своей адресной шине. До ЗЭ добираемся, подавая сигнал на два соответствующих вывода: один по горизонтали, другой по вертикали, ЗЭ находится на их пересечении. Здесь число выводов меньше: $2*\sqrt{256}=32$.
3. И, наконец, если для определения горизонтальной и вертикальной адресных шин использовать два дешифратора (DC 4-16), то число выводов уменьшится до 8 (4 вывода, для определения горизонтальной адресной шины, и 4 вывода для определения вертикальной адресной шины). Это и есть ЗУ с ПВ.

Примечание

Емкость ЗУ напрямую связана с количеством разрядов в адресном слове, например, если адресное слово содержит 8 разрядов, то 512 ЗЭ в ЗУ уже не разместить, так как свой адрес могут иметь только 256 ЗЭ, и обратиться по адресу к остальным 256 ЗЭ будет невозможно.

То есть для увеличения емкости ЗУ необходимо увеличивать разрядность машинного или адресного слова.

Более подробно о современных методах построения ЗУ можно прочитать в соответствующей литературе.

Далее рассмотрим простейшую структуру схем памяти и соответствующие ей ЗЭ.

18.3.1. Простейший вариант структуры ЗУ с адресацией или с ПВ (статических ОЗУ (SRAM), ROM ЗУ)

В качестве ЗЭ в схеме на рис. 18.3 используется DV-триггер. Это D-триггер с тактовым входом (срабатывание по фронту), с входом разрешения (*ENA*), с двумя инверсными входами установки "1" — *PRN* и установки "0" — *CLRN* (если они "висят" в воздухе, то на них подается "1").

Если на вход "запись" подается "1", то имеем режим записи по выставленному на дешифраторы адресу, если на вход "запись" подается "0" , то имеем режим считывания.

Информация (бит) подается на вход "число".

Предлагается самостоятельно разобраться, как в ЗУ записывается и считывается информация.

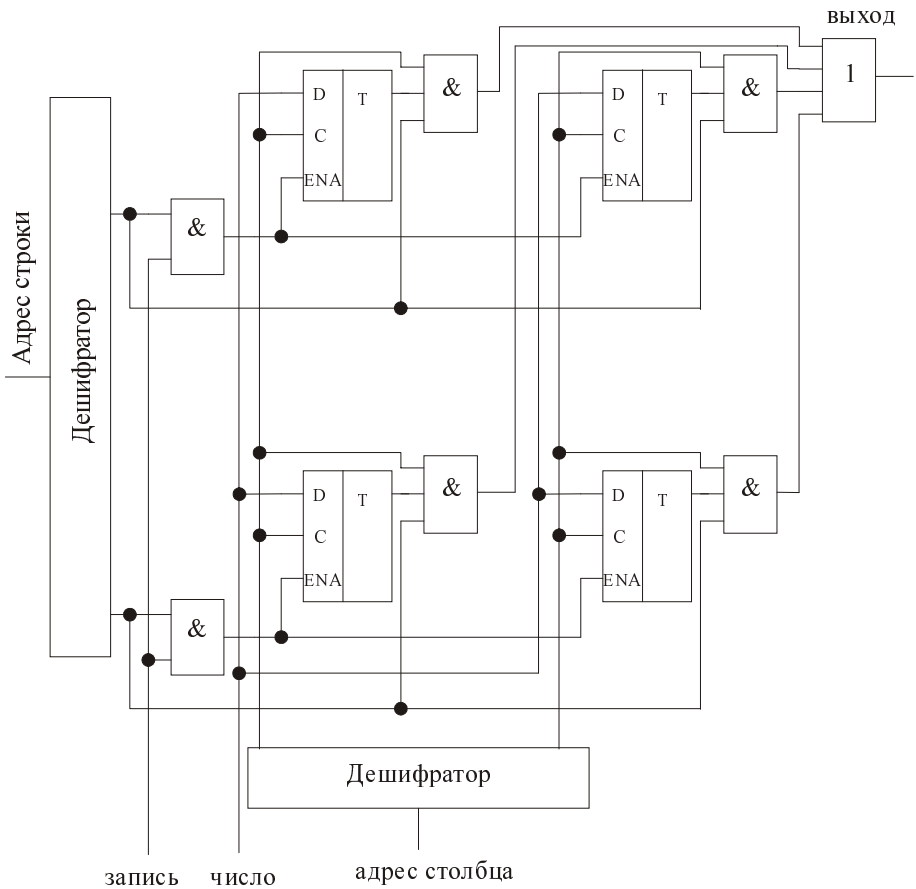


Рис. 18.3. ЗУ с ПВ из четырех ЗЭ

18.3.2. Запоминающие элементы (ЗЭ) статической памяти (SRAM)

Рассмотрим элементы БИС ОЗУ статического типа на МДП-транзисторах.

ЗЭ SRAM на n-МОП-транзисторах

На рис. 18.4 изображен простейший вариант ЗЭ на n-МОП-транзисторах или RS-триггер на конъюнкторах с отрицанием.

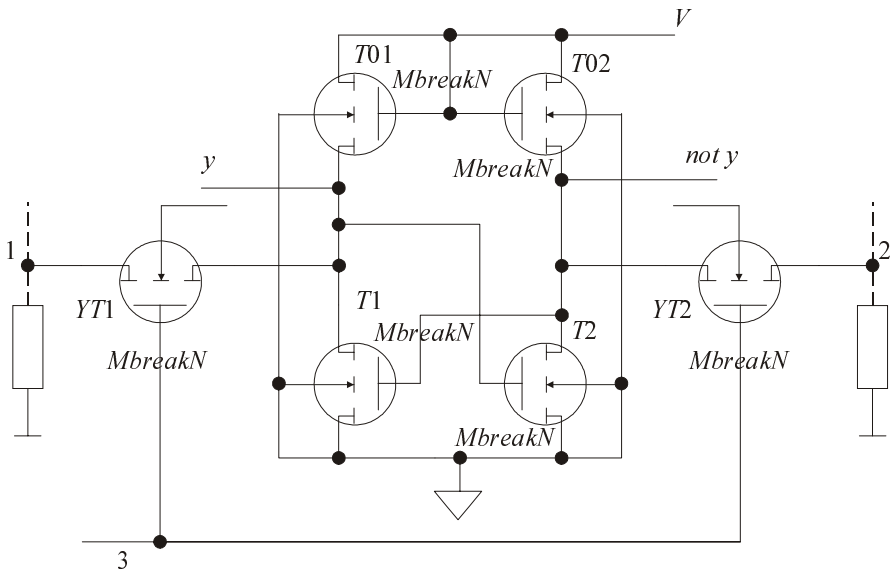


Рис. 18.4. ЗЭ SRAM на n-МОП-транзисторах

По входам 1 и 2 управляющих транзисторов (УТ) триггеры или ЗЭ объединены в столбцы. По входу 3 в строки.

О режиме хранения: при $y = 0$ транзисторы $T01$ и $T1$ открыты, $T02$ и $T2$ закрыты, но напряжение $U_{зи}T1$ уменьшается (токи утечки), поэтому в состоянии хранения $T02$ приоткрывается и дозаряжает $U_{зи}T1$.

Для записи надо подать воздействие по всем трем входам:

- для записи "1" подать напряжение U (единичный уровень) на входы 1 и 3 и "0" (нулевой уровень) на вход 2.

  Рассмотрим процесс во времени: был "0", записываем "1", то есть $T01$ и $T1$ были открыты и по ним протекал ток, а $T02$ и $T2$ были закрыты. После подачи

импульса сначала $C_{зи}T1 = C_{вх}T1$ разряжается через $YT2$, затем $C_{вх}T2$ заряжается через $YT1$;

- для записи "0" подать напряжение U на входы 2 и 3 и "0" на вход 1.

Для считывания воздействуем U (единичным уровнем) только на 3 вход, а к выходам 1 и 2 подаем "0" через нагрузку.

Транзисторы $YT1$ и $YT2$ используются в режиме двусторонней проводимости, в зависимости от того, записываем мы или считываем, "Сток" и "Исток" меняются местами. Эти транзисторы называются *двунаправленными ключами ввода/вывода*.

ЗЭ SRAM на К-МОП-схемах

Рассмотрим простейший вариант ЗЭ на К-МОП-схемах или RS-триггер на конъюнкторах с отрицанием (рис. 18.5).

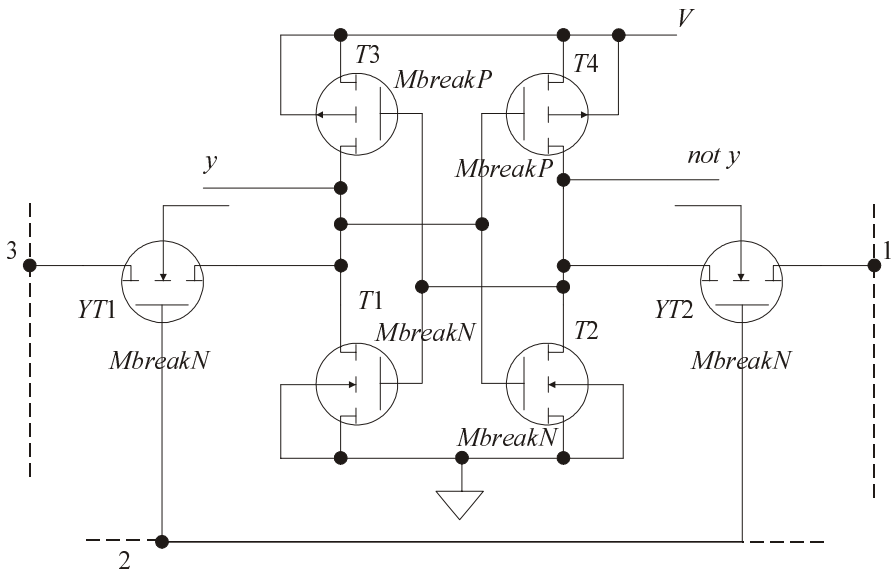


Рис. 18.5. ЗЭ SRAM на К-МОП-схемах

По входам 1 и 3 ЗЭ объединены в столбцы, а по входам 2 — в строки.

Транзисторы УТ по аналогии используются в режиме двусторонней проводимости.

О режиме хранения: здесь транзисторы у конъюнкторов разной проводимости, поэтому когда один транзистор конъюнктора открыт, другой заперт, и емкость $C_{зи}$ подзаряжается от источника через открытый транзистор соседнего конъюнктора.

Для записи "1" на вход 3 подается "0", а на вход 1 и 2 подается "1".

Для записи "0" на вход 1 подается "0", а на вход 3 и 2 подается "1".

Для считывания воздействуем "1" только на 2 вход, а к 1 и 3 входам подключаем "0" через нагрузку.

18.3.3. Запоминающие элементы динамической памяти (DRAM)

Эти ЗЭ относятся к элементам БИС ОЗУ динамического типа на МДП-транзисторах. Схема приведена на рис. 18.6.

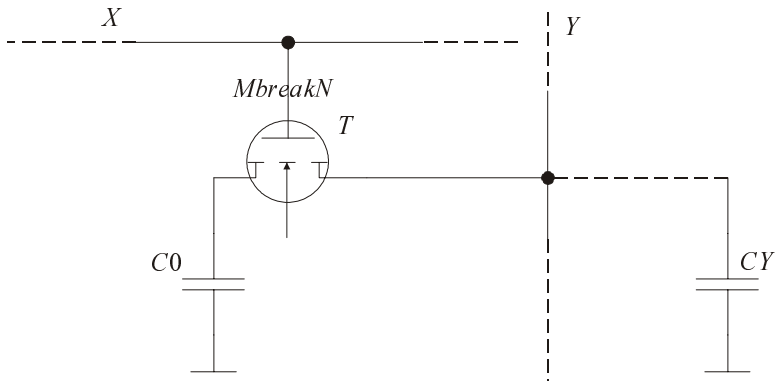


Рис. 18.6. ЗЭ DRAM на МДП-транзисторах

Принцип действия основан на хранении информации на конденсаторах.

Режим хранения: напряжение U на шине строки $X = 0$, поэтому транзистор T закрыт. При закрытом транзисторе T конденсатор $C0$ будет отключен от шины Y, поэтому на $C0$ будет храниться либо напряжение, соответствующее единице $U1$, либо напряжение, соответствующее нулю $U0$. В режиме хранения из-за токов утечки напряжение $UC0$ будет относительно медленно (по сравнению с временами записи и чтения) меняться, поэтому необходима регенерация (периодическое восстановление) напряжений $U1$ или $U0$ на конденсаторе $C0$.

Режим записи: на соответствующую адресу шину Y подается напряжение $U1$, либо $U0$, затем подается положительный импульс выборки на шину X. При этом в остальных элементах выбранной строки X идет регенерация.

Режим считывания: на шину столбца Y подаем напряжение $U_{on}(U0 < U_{on} < U1)$, оно поддерживается емкостью C_y, затем подается импульс выборки на шину X, поэтому транзистор отпирается, и $C0$ и C_y оказываются параллельно включенными и взаимно перезаряжаются, поэтому на шине Y либо ($U_{on} + dU1$), если считываем "1", либо ($U_{on} + dU0$), если считываем "0". Затем напряжение на шине Y подаем на усилитель-считыватель (например, четное число инверторов), который формирует либо напряжение $U1$, либо напряжение $U0$.

Регенерация — это считывание, преобразование в напряжение $U1$ или $U0$ усилителем-считывателем и последующая перезапись. Регенерация проходит одновременно для всех элементов строки.

Таким образом, динамические элементы памяти маломощны (потребляют энергию во время регенерации, записи и считывания). Динамическая память (DRAM) имеет высокую информационную емкость БИС в 4—16 раз больше, чем у БИС памяти статического типа (SRAM), но быстродействие у динамической памяти ниже (большое время выборки).

Лабораторная работа. Исследование функционирования схем памяти

Цель работы

Основная задача данной лабораторной работы — изучение различных схем памяти. На примере простейшей памяти на примитивах рассматриваются основные этапы работы запоминающего устройства. Далее исследуются более сложные схемы регистровой памяти.

Программа работы

1. Исследуйте работу схемы простейшей памяти емкостью 4 бита, приведенной на рис. 18.7. Создайте проект, используя графический редактор, и откомпилируйте его. С помощью редактора временных диаграмм задайте четыре различных значения информационного сигнала и проследите, чтобы эти значения были сохранены в разных ячейках памяти. Получите временные диаграммы.

2. Разработайте поведенческую модель работы схемы простейшей памяти на VHDL.
3. Реализуйте в графическом редакторе проект памяти для хранения четырех восьмибитных слов (схема представлена на рис. 18.8). Поясните работу этой схемы и протестируйте ее (приведите временные диаграммы).

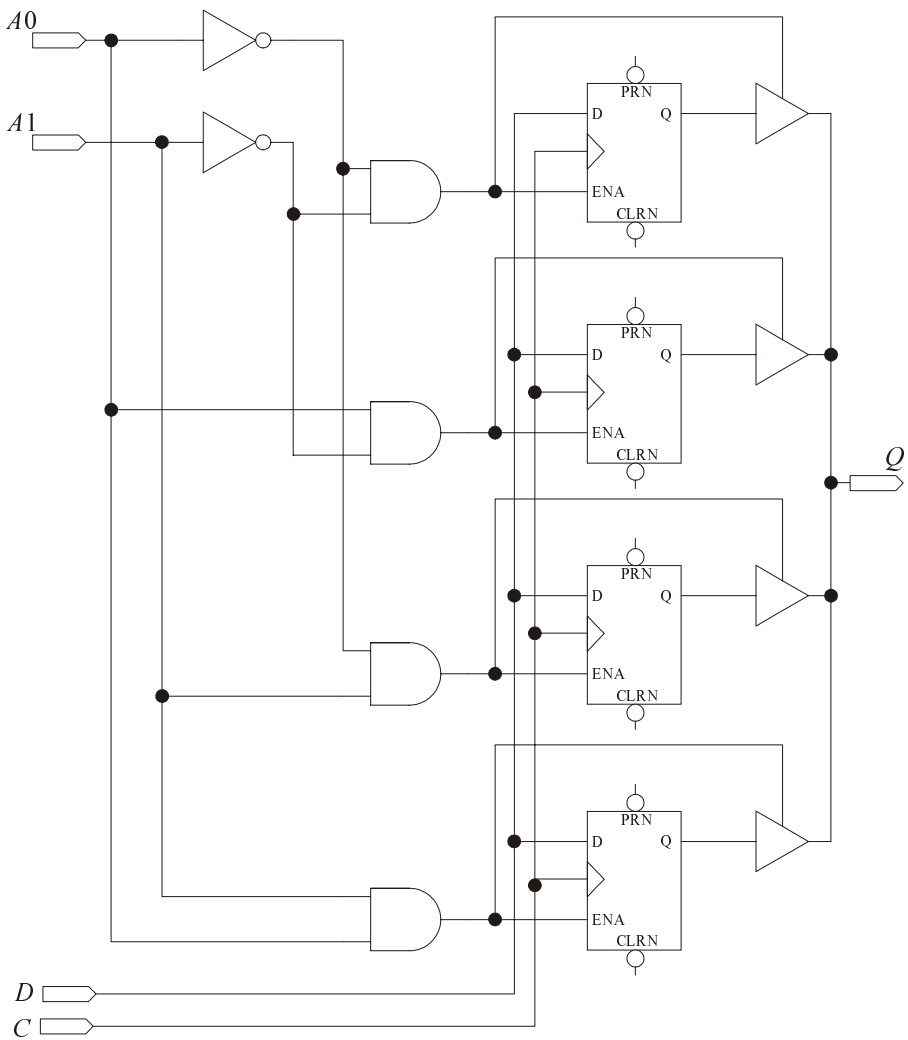


Рис. 18.7. Схема памяти на примитивах

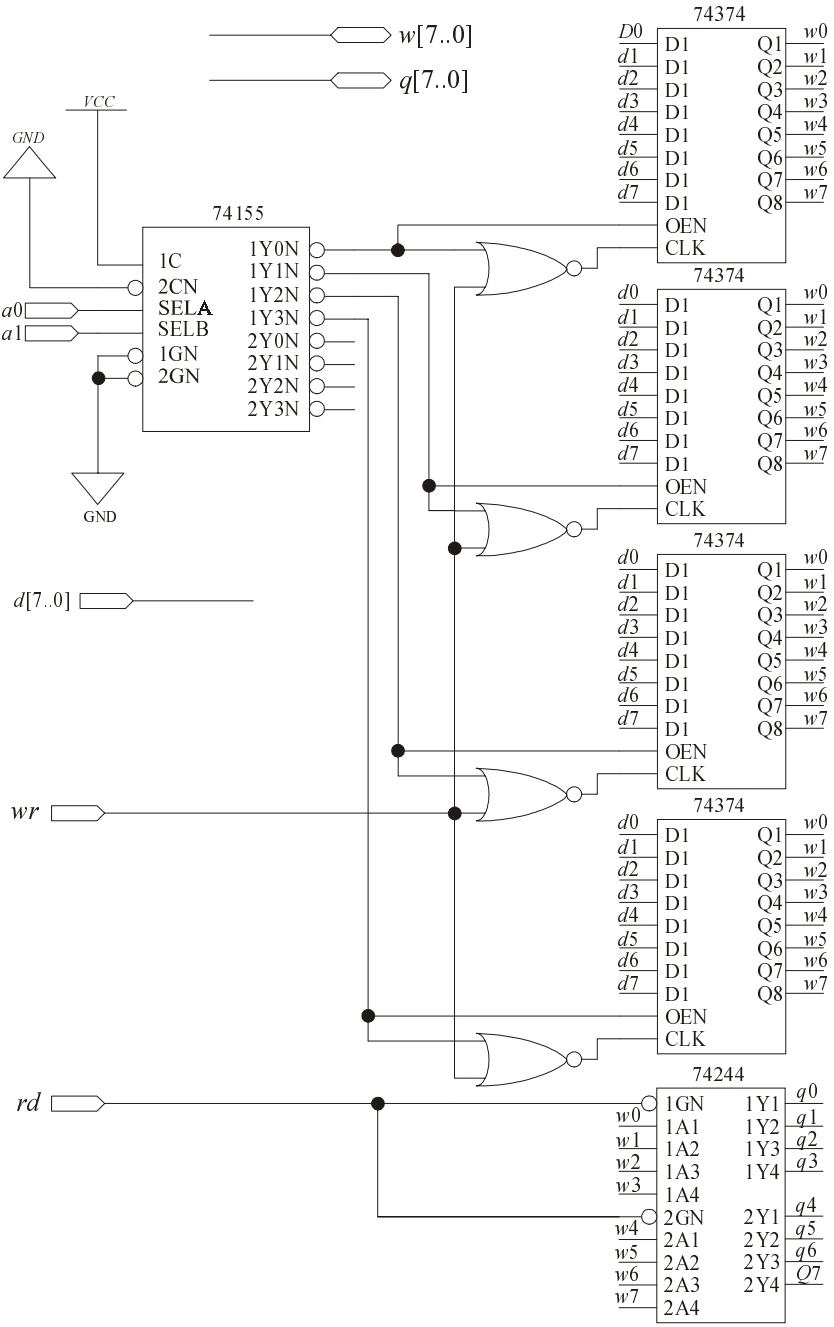


Рис. 18.8. Схема памяти, реализованная на регистрах

4. Исследуйте работу универсальной настраиваемой памяти (рис. 18.9) в графическом редакторе. Для этого в библиотеке mega-lpm выберите элемент lpm_ram_dq. Перед настройкой элемента внимательно ознакомьтесь с его описанием, используя Help Max+plusII. Настройте элемент, оставив только входы WE, $ADDRESS$, $DATA$ и выход Q, установив следующие параметры:

```
lpm_address_control          "unregistered"
lpm_file                     <none>
lpm_indata                   "unregistered"
lpm_numwords                 <none>
lpm_outdata                  "unregistered"
lpm_width                    8
lpm_widthad                  10
```

Настройку параметров можно производить, используя контекстное меню (**EDIT PORT** | **PARAMETERS**).

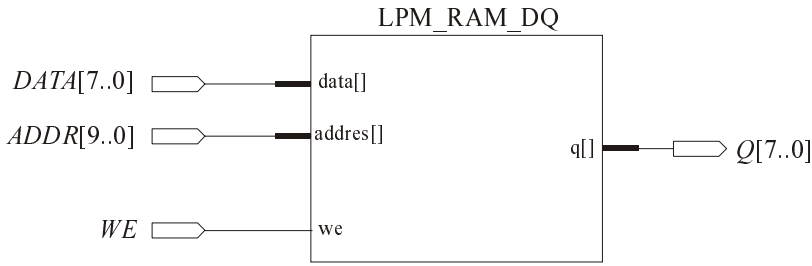
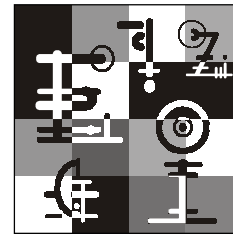


Рис. 18.9. Элемент lpm_ram_dq



Глава 19

Структуры построения специальных схем памяти, RAM и ROM

Из специальных схем памяти рассмотрим КЭШ-память, из RAM рассмотрим схемы памяти с последовательной выборкой (пример построения RAM с произвольной выборкой (ПВ) был приведен ранее), из ROM рассмотрим функциональную схему ПЗУ и возможность проектирования с помощью схемы ПЗУ.

В данном рассмотрении термины схема памяти и ЗУ будут обозначать одно и то же.

19.1. Структура кэшированной (CACHE) памяти

Кэш-память (рис. 19.1) служит для запоминания копий данных, передаваемых между устройствами (в частности: между процессором (CPU) и RAM). Отличается более высоким быстродействием (SRAM) и небольшой емкостью.

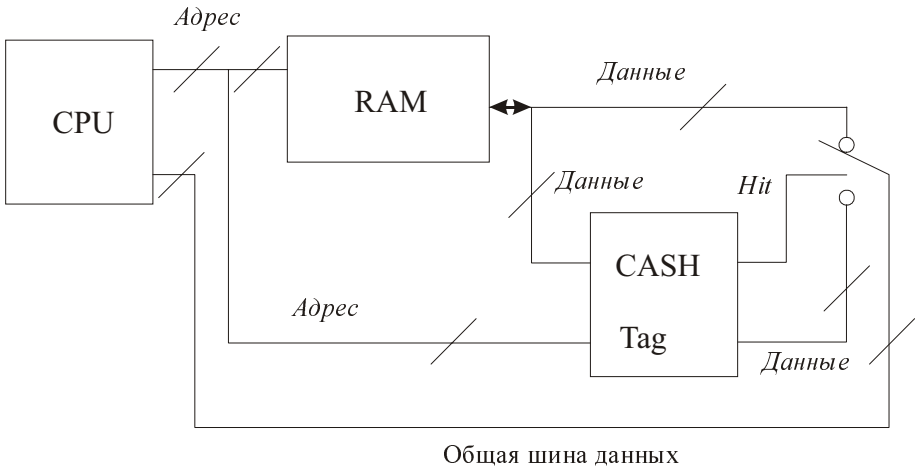


Рис. 19.1. Схема кэшированной памяти

Функционирование кэшированной памяти: при чтении сначала обращаются к кэш-памяти. Сигнал *"Hit"* (попадание) вырабатывается, если в кэш-памяти имеется копия данных адресованной ячейки RAM.

Здесь Tag — данные, определяющие ячейку RAM, копия которой находится в кэш-памяти.

В результате, данные из кэш-памяти поступают на общую шину данных.

При использовании кэш-памяти RAM освобождается и может использоваться другими устройствами.

19.1.1. Структура полностью ассоциативной кэш-памяти

Полностью ассоциативная кэш-память переводится на английский как Fully Associated Cache Memory (FACM). Рассмотрим структуру FACM (рис. 19.2).

В поле "Тег" записан полный физический адрес данных из RAM, копия которых находится в кэш-памяти.

Рассмотрим работу FACM.

Чтение:

- если сигнал $Hit = 1$, то данные из кэш-памяти поступают на общую шину данных;
- если сигнал $Hit = 0$, то данные читаются из RAM и одновременно вместе с адресом помещаются в свободную или давно неиспользуемую ячейку кэш-памяти.

При *записи* данные с адресом сначала размещаются в кэш-памяти (при $Hit = 1$ в обнаруженную ячейку, при $Hit = 0$ в свободную ячейку).

Копирование данных из RAM выполняется под управлением контроллера, когда нет обращения к RAM.

Более экономичными по аппаратным затратам являются кэш-память со структурой прямого размещения (L2) и кэш-память с наборно-ассоциативной архитектурой (L1). Здесь L — level (внутрипроцессорный уровень).

В современных микропроцессорных структурах кэш первого уровня — L1, кэш второго уровня — L2.

Уровень L1 — промежуточный по эффективности вариант кэш-памяти между FACM и уровнем L2.

Более подробно о кэш-памяти первого и второго уровней можно прочитать в соответствующей литературе.

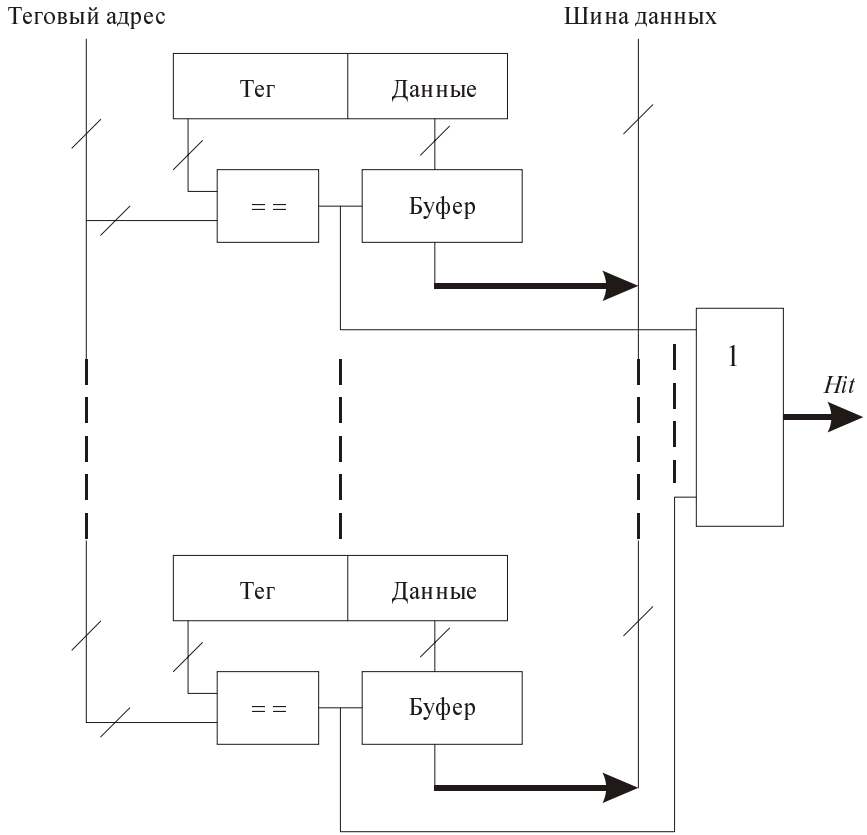


Рис. 19.2. Схема полностью ассоциативной кэш-памяти

19.2. Структура схем памяти с последовательной выборкой

Структура схем памяти с последовательной выборкой может строиться:

- либо по аналогии с регистрами сдвига;
- либо с хранением данных в RAM и с последовательным адресным доступом.

19.2.1. Структура циклических схем памяти (видеопамять)

Видеопамять относится к памяти с последовательным адресным доступом к RAM и содержит последовательность кадров. Кадр на мониторе — это последовательность слов. Ее длина равна числу пикселов экрана.

Пиксел — одна точка экрана, задаваемая параметрами (цвет, яркость).

Разрядность слов, описывающих пиксел:

- 8 — для черно-белого монитора;
- 24 — для полноцветного режима.

Функциональная схема видеопамяти показана на рис. 19.3.

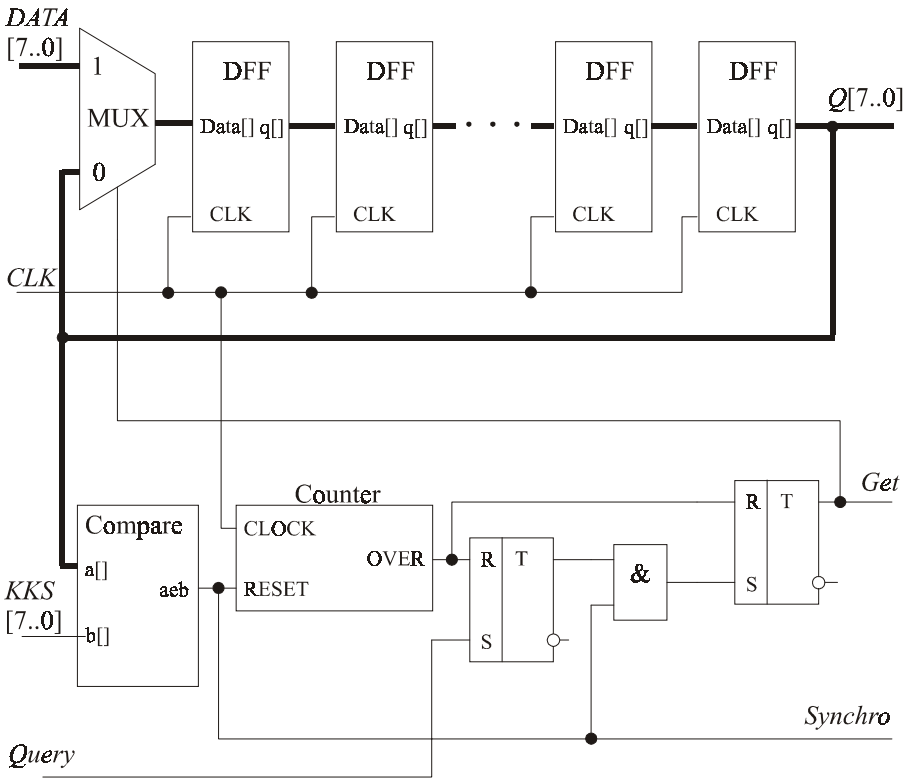


Рис. 19.3. Функциональная схема видеопамяти

Здесь на вход *DATA* последовательно поступают данные из RAM (побайтно), на вход *CLK* — синхроимпульсы, на вход *KKS* — код кадрового синхросигнала, на вход *Query* — запрос передачи.

С выхода *Q* данные поступают на монитор, на выходе *Get* вырабатывается сигнал разрешения передачи данных из RAM, на выходе *Synchro* вырабатывается сигнал запуска развертки монитора (сигнал кадровой синхронизации).

Рассмотрим работу видеопамяти.

Считывание из видеопамяти (циклическая перезапись), при этом изображение на мониторе неподвижно:

1. Выбран нижний вход MUX .
2. Хранимые данные поступают на выход Q и на монитор, переписываются с выхода Q на нижний вход MUX .
3. В последовательность данных (слов) кадра введен код кадрового синхросигнала ($"KKS"$), его появление на выходе Q (в начале каждого кадра) фиксируется компаратором, который запускает развертку монитора (сигнал $"Synchro"$), при этом начало кадра точно соответствует положению луча монитора вверху и слева.

Пакетная запись, при этом изображение на мониторе меняется:

1. Начинается при наличии запроса передачи (сигнала $"Query"$) во время появления среди данных кода кадрового синхросигнала ($"KKS"$).
2. После этого вырабатывается сигнал разрешения передачи кадра из RAM на вход $"DATA"$ (сигнал $"Get"$), одновременно переключающий MUX на верхний канал.
3. Запись кадра завершается сигналом переполнения счетчика ($"CTR"$), модуль которого равен длине кадра.
4. После окончания записи видеопамять возвращается в режим циклической перезаписи.

19.2.2. Структура схем памяти, аналогичных регистрам сдвига (буферы FIFO и LIFO)

В буфере FIFO (First In — First Out) порядок выборки информации (машинных слов) аналогичен порядку поступления, то есть новое слово поступает в конец очереди, считывание происходит с начала очереди.

Поступление в буфер возможно нерегулярно и с большей частотой, считывание — регулярно и с необходимой (меньшей) частотой.

Буфер FIFO может быть шлюзом между устройствами, работающими с большими и малыми частотами.

Функциональная схема буфера FIFO показана на рис. 19.4.

Здесь импульс WR — импульс, по которому происходит запись в буфер.

RD — импульс, по которому происходит чтение из буфера.

DI — входной порт буфера данных, представляющих последовательность машинных слов.

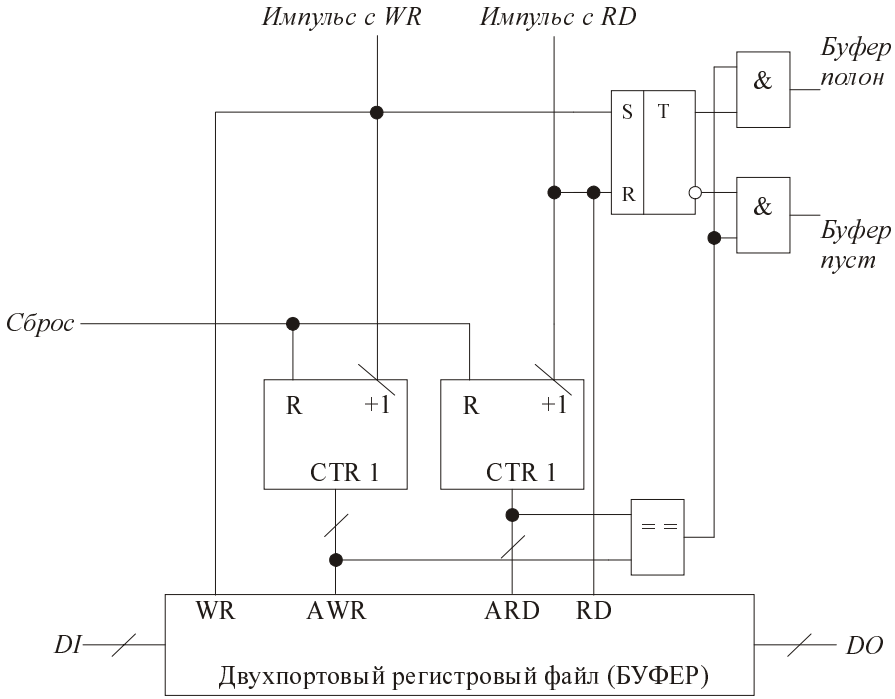


Рис. 19.4. Функциональная схема буфера FIFO

DO — выходной порт буфера данных, представляющих последовательность машинных слов.

AWR — адрес в буфере записанного машинного слова.

ARD — адрес в буфере прочитанного машинного слова.

$CTR\,1$ и $CTR\,2$ — счетчики адресов записанных и прочитанных машинных слов, соответственно.

Рассмотрим работу буфера FIFO.

1. Сначала счетчики адресов $CTR\,1$ и $CTR\,2$ обнуляются.
2. При записи в буфер $CTR\,1$ считает и AWR увеличивается.
3. При чтении из буфера считает $CTR\,2$, а увеличивается ARD.
4. Частота импульсов RD и WR может быть различна.
5. Если в процессе работы $ARD = AWR$ *при чтении* из буфера, значит буфер пуст. При этом вырабатывается соответствующий сигнал и чтение прекращается.

6. Если в процессе работы $ARD = AWR$ *при записи* в буфер, значит буфер полон. При этом вырабатывается соответствующий сигнал и запись прекращается (здесь имеем переход счетчика через ноль).

По аналогии строится буфер LIFO (Last In — First Out). В нем порядок выборки информации (машинных слов) противоположен порядку поступления, то есть новое слово поступает в начало очереди, считывание происходит с начала очереди.

Предлагается самостоятельно спроектировать буфер LIFO.

19.3. Структура схемы ROM на примере схемы ПЗУ

Существует два подхода к определению понятия ПЗУ (постоянное ЗУ):

- первый изложен при описании функциональной классификации ЗУ;
- второй подход состоит в том, что под ПЗУ понимают и МПЗУ (масочное ПЗУ), и ППЗУ (программируемое ПЗУ), и РПЗУ (репрограммируемое ПЗУ).

Условное обозначение схемы ПЗУ показано на рис. 19.5.

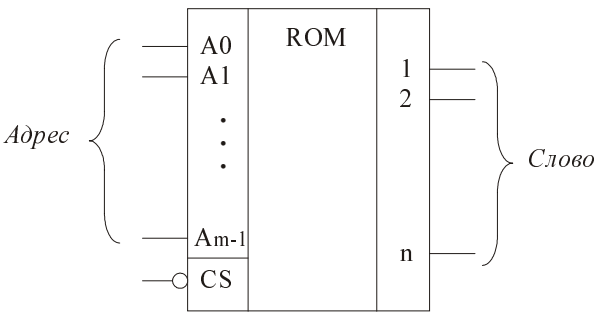


Рис. 19.5. Условное обозначение схемы ПЗУ

Это ПЗУ хранит $N = 2^m$ n-разрядных слов.

Здесь CS — chip select, этот вход может обозначаться RD — read или BM — выбор микросхемы, или BK — выбор кристалла, или PB — разрешение выбора.

Рассмотрим функциональную схему ПЗУ — рис. 19.6.

Запись информации в ПЗУ осуществляется путем разрушения связей (перемычек) между выходами дешифратора DC и входами элементов "ИЛИ".

Рассмотренные МПЗУ, ППЗУ и РПЗУ имеют, приблизительно, одинаковую функциональную схему. Их основное отличие состоит в разной реализации связей (перемычек):

- у МПЗУ это металлизация;
- у ППЗУ это перемычки из поликремния или нихрома (при однократном программировании их пережигают);

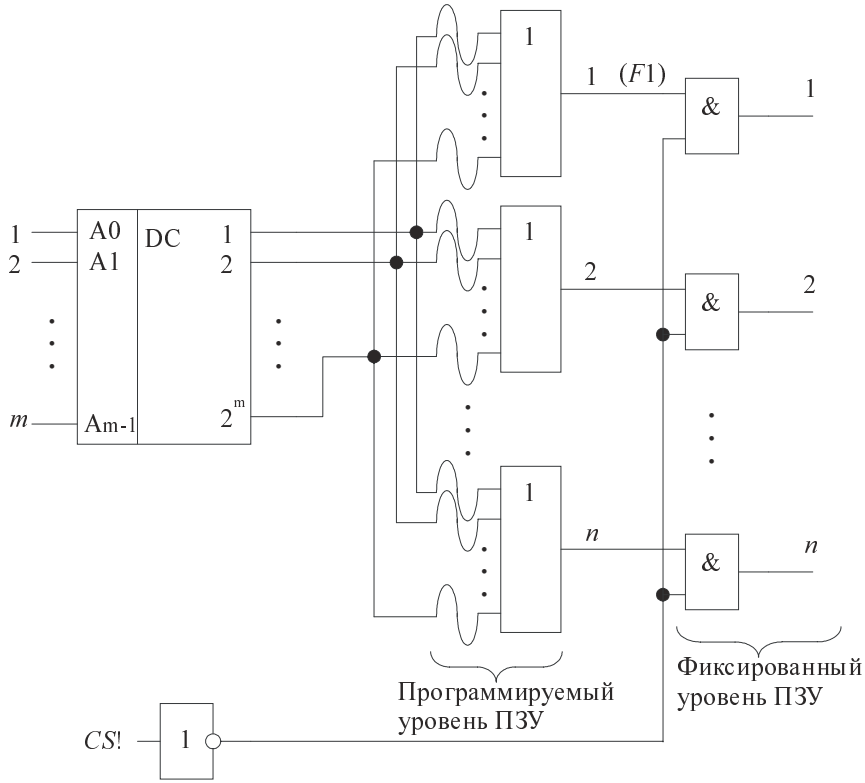


Рис. 19.6. Функциональная схема ПЗУ

- у РПЗУ это бистабильные МДП-структуры или транзисторы МНОП, МАОП, МАП, а также МДП-транзисторы с лавинной инжекцией и плавающим затвором (ЛИПЗ) и ЛИПЗ МДП-транзисторы с дополнительным управляющим затвором.

Здесь Н — нитрид кремния, А — алюминий, О — окисел кремния.

Более подробно о РПЗУ с разными бистабильными МДП-структурами можно прочитать в соответствующей литературе.

19.3.1. Проектирование с помощью схем ПЗУ

Проиллюстрируем проектирование с помощью схем ПЗУ на примере.

Дано: трехадресное ПЗУ n-разрядных слов.

Реализовать: логические функции $F1 = X1 + X2 * X3$, $F2 = \ldots$, $\ldots$, $Fn = \ldots$.

Проектирование: из описания DC его выходы соответствуют полному набору конъюнктивных термов, на первом выходе ($X1!*X2!*\ldots*Xm!$), на втором ($X1!*X2!*\ldots*Xm!$) и так далее.

Для проектирования развернем функцию $F1$ в СДНФ:

$$F1 = X1 + X2*X3 = X1*(X2 + X2!)*(X3 + X3!) + (X1 + X1!)*X2*X3 =$$
$$= X1*X2*X3 + X1*X2!*X3 + X1*X2*X3! + X1*X2!*X3! + X1!*X2*X3.$$

Все представленные пять термов присутствуют на выходах DC 3-8, поэтому для получения функции $F1$ на выходе первого (верхнего) "ИЛИ" необходимо оставить перемычки на соответствующих этим термам входах первого "ИЛИ", а все остальные перемычки этого "ИЛИ" разорвать.

Таким образом, с помощью трехадресного ПЗУ n-разрядных слов можно получить n логических функций трех аргументов.

Лабораторная работа. Исследование функционирования схемы видеопамяти

Цель работы

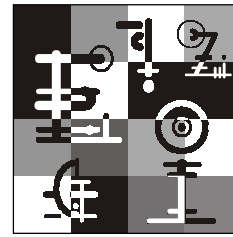
Основной задачей лабораторной работы является проектирование и исследование работы схемы видеопамяти.

В настоящей главе представлена функциональная схема видеопамяти. На выходе Q последовательно в порядке сканирования экрана монитора лучом появляются коды, задающие параметры светимости (цвет, яркость) элементарных точек экрана — пикселов. Текущее изображение на мониторе — кадр — представлено последовательностью слов, длина которой равна числу пикселов экрана. Для хранения каждого слова используется статический регистр. В последовательность данных вводятся специальные коды синхросигналов. Появление кода синхросигнала на выходе обнаруживается компаратором и синхронизирует запуск развертки монитора (выход $Synchro$). Данные поступают на вход $DATA$, вход KKS — эталонное значение кода кадрового синхросигнала. Пакетная запись данных может начинаться после появления запроса передачи в момент прохождения кода кадрового син-

хросигнала (вход *Query*). При этом вырабатывается сигнал разрешения передачи кадра из памяти ЭВМ на вход *DATA* (выход *Get*).

Программа работы

1. Используя материал данной главы, разработайте схему видеопамяти. Функциональная схема этого устройства приведена на рис. 19.3. Проверьте правильность работы схемы, задав все возможные режимы ее работы. Поясните принцип работы устройства и приведите временные диаграммы.
2. Напишите программную модель видеопамяти на VHDL. Сравните результаты работы модели с результатами, полученными в предыдущем пункте.



Глава 20

Классификация и этапы разработки специализированных БИС

Есть стандартные БИС микропроцессоров и ЗУ, они позволяют создать изделия от универсальной ЭВМ до бытовой техники, но только специализированные БИС придают этим изделиям уникальные свойства: по надежности, микроразмещению, быстродействию и т. д.

Классифицируют специализированные БИС по способам их проектирования и изготовления, направленным на реализацию конкретной функции заказчика (рис. 20.1).

Рис. 20.1. Классификация специализированных БИС

Где ПЛИС — это программируемые логические интегральные схемы.

БК — это базовые кристаллы.

ПЛСБИС — это программируемые логические сверхбольшие интегральные схемы, которые в своей основе могут иметь как архитектуру ПЛИС, так и архитектуру БК.

СЭ — это заказные специализированные БИС на стандартных элементах.

ПЗ — это полностью заказные специализированные БИС.

Все топологические слои заказной БИС — переменные и изготовляются по индивидуальным фотошаблонам.

Полузаказные БИС — это совокупность заранее спроектированной постоянной части и переменной, заказной части, определяемой заказчиком.

Для БК специализация полузаказных БИС осуществляется на заключительном этапе производства за счет нанесения переменных слоев межсоединений с помощью дополнительных фотошаблонов.

Для ПЛИС и ПЛСБИС дополнительных фотошаблонов не требуется, их программирование (специализация, конфигурирование) осуществляется электрическим способом: за счет изменения физического состояния элементов программирования (ПЛИС фирмы Альтера (MAX)) или за счет программы управления коммутацией логических элементов, хранящейся во внутренних (или внешних) элементах энергонезависимой памяти (ПЛИС фирмы Альтера (FLEX)).

Этапы разработки специализированных БИС упрощенно приведены в табл. 20.1. Здесь Н — необходим, О — отсутствует, В — возможен, но не всегда.

Таблица 20.1. *Этапы разработки специализированных БИС*

Этапы проектирования	**ПЛИС**	**БК**	**СЭ**	**ПЗ**
Архитектурный	Н	Н	Н	Н
Логический	Н	Н	Н	Н
Схемотехнический	О	О	О	Н
Изготовление шаблонов	О	В	Н	Н
Изготовление кристаллов (чипов)	О	Н	Н	Н
Проверка готовой схемы	Н	Н	Н	Н

20.1 Базовые кристаллы (БК)

Рассмотрим особенности конструкций и терминологию, относящиеся к базовым кристаллам.

20.1.1. Конструкции БК

(Конструкции ПЛСБИС — аналогичны, но в них есть особенности, которые будут рассмотрены далее.)

1. Классическая конструкция с канальной архитектурой (рис. 20.2).
2. Бесканальная архитектура (повышаются плотность упаковки и коэффициент использования логических элементов (ЛЭ) БК).

   Здесь центральная часть БК состоит из плотноупакованных рядов нескоммутированных элементов (базовых ячеек (БЯ)) и не содержит в первом уровне межсоединений фиксированных каналов. То есть любая область расположе-

ния транзисторов может быть использована для создания как ЛЭ, так и межсоединений.

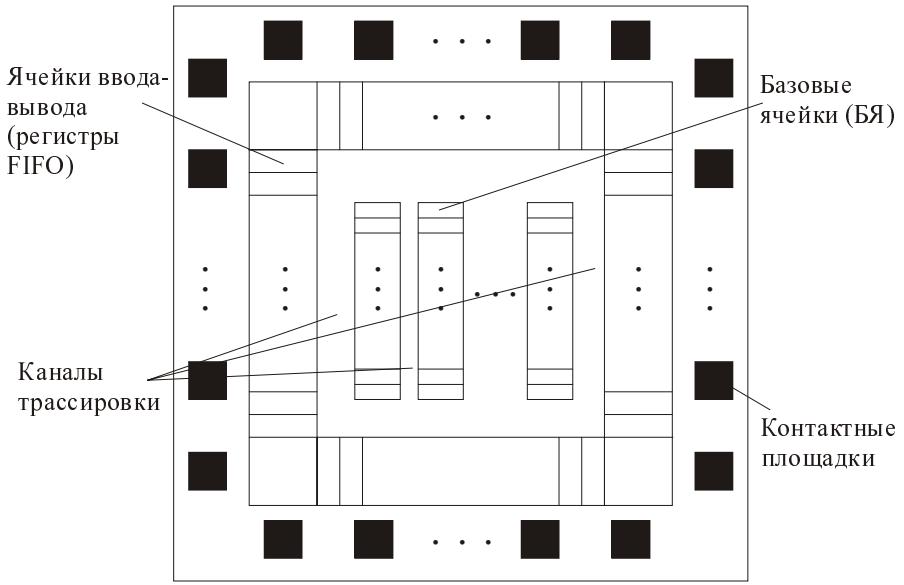


Рис. 20.2. Конструкция БК с канальной архитектурой

3. Архитектура, при которой БК содержит матрицу нескоммутированных элементов и функционально законченные узлы: ОЗУ, ПЗУ, ПЛМ, регистры и т. д. Такая архитектура ограничивает круг решаемых задач, но позволяет уменьшить площадь кристалла и повысить быстродействие.

20.1.2. Терминология БК

БК ИС — часть полупроводниковой пластины (интегральной схемы (ИС)) с определенным набором сформированных элементов, используемая для создания ИС путем изготовления межэлементных соединений.

БМК ИС — БК ИС с регулярным (матричным) расположением сформированных в нем элементов.

Базовая ячейка (БЯ) БМК — совокупность несоединенных и соединенных элементов, регулярно повторяющаяся в пределах БМК.

Две организации БЯ:

- БЯ состоит из набора нескоммутированных элементов — транзисторов, резисторов и так далее, на основе которых формируются библиотечные элементы или функциональные ячейки (ФЯ);

- БЯ являются функционально законченным узлом, выполняющим элементарную функцию: "И-НЕ", "ИЛИ-НЕ", конечный автомат на 3 или 4 входа и т. д.

ФЯ БМК — функционально законченная электрическая схема, реализуемая соединением элементов БМК в пределах одной или нескольких БЯ.

Библиотека ФЯ БМК — совокупность ФЯ БМК, используемая при проектировании полузаказных матричных ИС.

Полузаказная матричная ИС — ИС, разработанная на основе БМК.

Эквивалентный ЛЭ БМК — совокупность элементов БМК, эквивалентных по логической функции ЛЭ "И-НЕ" ("ИЛИ-НЕ") (вентилям), и предназначенная для характеристики его логической сложности.

БМК, иначе, называют вентильными матрицами (ВМ). За рубежом подобные структуры, в которых основную роль играют регулярно расположенные элементы, получили название Gate Array (GA).

20.2. Программируемые логические интегральные схемы (ПЛИС)

Можно проектировать цифровое устройство, используя ИС разных ЛЭ (многовходовых "И-ИЛИ-НЕ", дешифраторов, мультиплексоров и т. д.), а можно проектировать с помощью ПЛИС.

Проектирование на ПЛИС эффективно, когда полностью учитывается специфика ее логической структуры и специфика ее функциональных возможностей; то есть методика проектирования на ПЛИС основана на согласовании математического описания разрабатываемого устройства и базовой логической структуры ПЛИС.

Существуют три типа двухуровневых (программируемый и фиксированный уровни) ПЛИС.

Первый тип — с двумя программируемыми матрицами "И" и "ИЛИ" (это программируемая логическая матрица (ПЛМ)).

Второй и третий типы — с одной программируемой матрицей:

- 2 тип — если программируемая матрица — матрица "И", то это программируемая матричная логика (ПМЛ);
- 3 тип — если программируемая матрица — матрица "ИЛИ", то это программируемое постоянное ЗУ (ППЗУ).

В табл. 20.2 рассмотрены три типа ПЛИС: ПЛМ, ПМЛ и ППЗУ.

Таблица 20.2. *Три типа ПЛИС*

Тип ПЛИС	**К556РТ1**	**КМ1556ХП4**	**КМ556РТ7**
Программируемый уровень	Два: И, ИЛИ	И	ИЛИ
Фиксированный уровень	ИЛИ	ИЛИ	И
Условное обозначение логической ИС	ПЛМ	ПМЛ	ППЗУ

На ПЛИС разработаны: контроллеры клавиатуры, восьмиразрядные сумматоры, цифровые фильтры, таймеры и т. д.

20.2.1. Устройства на программируемых логических матрицах (ПЛМ)

Рассмотрим на примере два программируемых уровня ПЛМ (фиксированный уровень не рассматриваем):

- на первом уровне выполняется операция "И" над переменными X и их инверсиями;
- на втором уровне выполняется операция "ИЛИ" над переменными P, являющимися выходными сигналами первого уровня матрицы.

Инверторы будем обозначать кружками.

Крестиками обозначим элементы связи, которые включаются:

- либо после запуска заранее написанной программы управления коммутацией ЛЭ, хранящейся во внутренних элементах энергонезависимой памяти ПЛМ или во внешних элементах, на отдельной ИС ПЗУ. Для такого включения необходимо время. (Это относится к ПЛИС (FLEX).);
- либо сразу после подачи питания. (Это относится к ПЛИС (MAX).)

Спроектировать на ПЛМ (реализовать): логическую функцию (ЛФ) $Y1 = X1 * X2! * X3! + X1 * X2 * X3$.

Проектирование:

1. Представим ЛФ $Y1 = P1 + P2$, где $P1 = X1 * X2! * X3!$, $P2 = X1 * X2 * X3$.
2. Проектируем с помощью функциональной схемы ПЛМ (рис. 20.3).

Фрагмент логической схемы ПЛМ, относящийся к функции $Y1$, выглядит следующим образом — рис. 20.4.

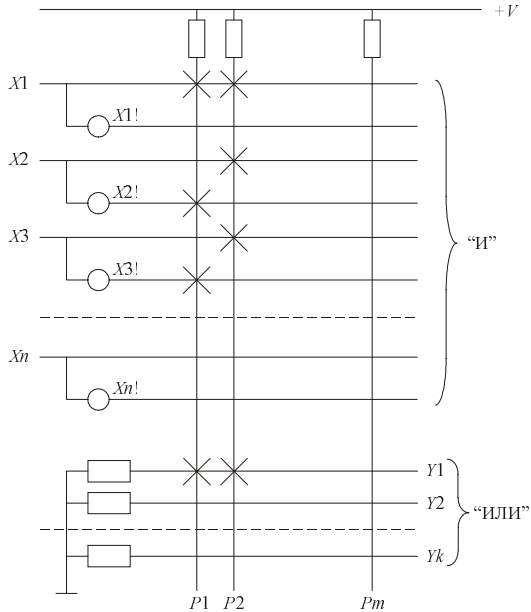


Рис. 20.3. Функциональная схема ПЛМ

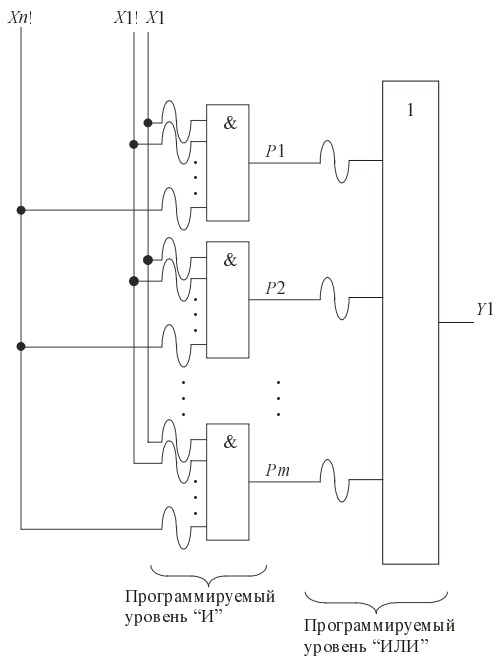


Рис. 20.4. Фрагмент логической схемы ПЛМ

20.2.2. Устройства на программируемой матричной логике (ПМЛ)

Пример проектирования на ПМЛ.

Спроектировать на ПМЛ (реализовать): логическую функцию (ЛФ) $Y3 = (X1 * X2! * X3!)!$, $Y4 = X1! * X2 * X3$.

Проектирование: проектируем с помощью функциональной схемы ПМЛ (рис. 20.5).

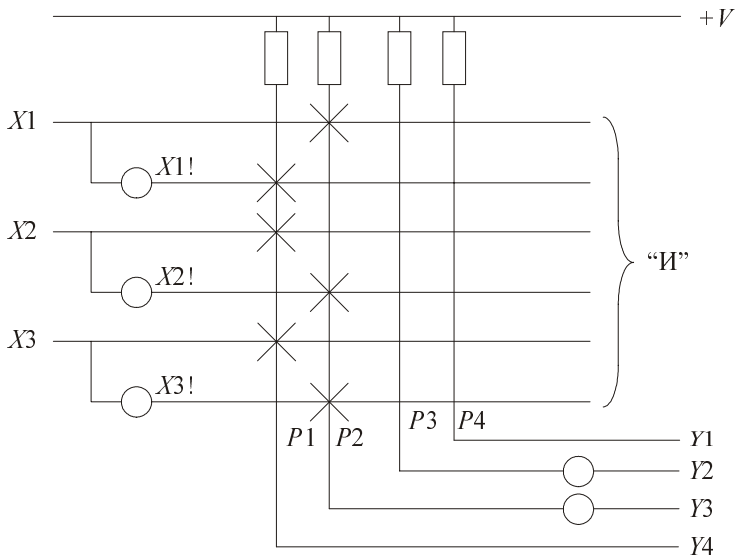


Рис. 20.5. Функциональная схема ПМЛ

За рубежом ПЛМ и ПМЛ обозначаются, соответственно, Programmable Logic Array (PLA) и Programmable Array Logic (PAL).

20.3. Классификация ПЛСБИС

Упрощенно ПЛСБИС классифицируются:

- по степени интеграции или логической емкости, измеряемой числом логических вентилей, по ней судят о возможности создания проекта с помощью данной ПЛСБИС;
- по архитектуре простейшего функционально законченного узла (универсального логического преобразователя). Она может быть реализована:
  - на основе ПЗУ;

- на основе ПЛМ;
- на основе таблицы перекодировок (Look-up-table (LUT));

❒ по организации внутренней структуры ПЛСБИС и структуры матрицы соединений (одноуровневые и многоуровневые (иерархические) структуры);

❒ по наличию внутри ПЛСБИС RAM-памяти.

ПЛСБИС, реализованные на матрицах элементов "И" и "ИЛИ" (на ПЛМ или PLA) или на матрице элемента "И" (на ПМЛ или PAL), получили название Programmable Logic Device (PLD).

ПЛСБИС, в основу которых заложена перепрограммируемая пользователем ВМ или перепрограммируемый БМК, получили название Field Programmable Gate Array (FPGA).

ПЛСБИС, реализованные одновременно на ПЛМ (PLA) или ПМЛ (PAL) и на вентильной матрице (ВМ или GAL), которые объединены с помощью матрицы программируемых соединений, получили название High-complexity PLD (CPLD).

О ПЗУ и ПЛМ как универсальных логических преобразователях говорилось в предыдущих главах.

Таблица перекодировок (LUT) с n-входами — это одноразрядное ЗУ объемом 2^n бит, состоящее из n-входового декодера DC, соединенного через программируемые элементы (перемычки) с одним многовходовым элементом "ИЛИ", к которому подключен один элемент "И" (см. рис. 19.6).

Число входов в LUT небольшое, тогда как число универсальных логических преобразователей, реализованных на основе LUT, достигает в ПЛСБИС нескольких тысяч.

За счет этого возможна иерархическая реализация сложных логических функций с использованием нескольких LUT, что позволило уменьшить аппаратную избыточность по сравнению с реализацией на ПЗУ.

Лабораторная работа. Исследование функционирования схем регистров FIFO, LIFO и кэш-памяти

Цель работы

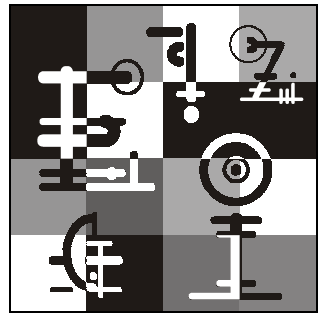
Основной задачей лабораторной работы является проектирование и исследование работы схем буферов FIFO и LIFO, которые можно отнести к запоминающим устройствам с последовательным доступом.

Функциональная схема буфера FIFO представлена в предыдущей главе. Буфер FIFO — это ЗУ для хранения очередей данных (списков) с порядком выборки слов таким же, что и порядок их поступления. Новое слово ставится в конец очереди, считывание осуществляется с начала очереди. В схеме перед началом работы оба счетчика адресов $CTR1$ и $CTR2$ сбрасываются. При записи адреса увеличиваются на единицу при каждом обращении, начиная с нулевого. То же происходит при чтении слов. Если адреса сравняются при чтении, то буфер пуст. Если адреса сравняются при записи, то буфер полон. Если буфер полон, то нужно прекратить прием данных, а если пуст, то нужно прекратить чтение. Аналогично решается задача построения буфера LIFO.

Кэш-память запоминает копии информации, передаваемой между устройствами. Структурная схема кэш-памяти приведена в предыдущей главе. Каждая ячейка памяти хранит данные, а поле Tag — полный физический адрес информации, хранящейся в основной памяти, копия которой записана в кэш-память. При любых обменах физический адрес запрашиваемой информации сравнивается с полями Tag всех ячеек и при совпадении их в любой ячейке сигнал $"Hit"$ устанавливается в "1". При чтении и значении сигнала $Hit = 1$ данные выдаются на шину данных, если же совпадений нет ($Hit = 0$), то при чтении из основной памяти данные вместе с адресом помещаются в свободную ячейку кэш-памяти. При записи данные вместе с адресом размещаются в кэш-памяти (в обнаруженную ячейку при $Hit = 1$ и свободную при $Hit = 0$).

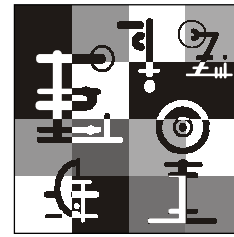
Программа работы

1. Используя материал предыдущей главы, разработайте схему буфера FIFO. Функциональная схема этого устройства приведена в той же главе. Проверьте правильность работы схемы, задав все возможные режимы ее работы. Поясните принцип работы устройства и приведите временные диаграммы.
2. Спроектируйте программную модель буфера FIFO на VHDL. Сравните результаты работы модели с результатами, полученными в предыдущем пункте.
3. Разработайте в графическом редакторе схему буфера LIFO. Получите временные диаграммы. Используя диаграммы, поясните принцип работы этого устройства.
4. Создайте программную модель буфера LIFO на VHDL. Сравните результаты работы модели с результатами, полученными в п. 3.
5. Используя материал предыдущей главы, разработайте схему кэш-памяти. Функциональная схема этого устройства приведена в той же главе. Проверьте правильность работы схемы, задав все возможные режимы ее работы. Поясните принцип работы устройства и приведите временные диаграммы.



Часть IV

Поведенческие и структурные VHDL-модели сложных цифровых устройств на примере VHDL-моделей процессора DP-32, их тестирование



Глава 21

Типы и уровни проектирования сложных цифровых устройств, концепции языка VHDL

Рассмотрим современный подход к проектированию сложных устройств вычислительной техники (ВТ): процессоров, контроллеров, систем на кристалле.

Средства проектирования можно подразделить на средства описания сложных проектов и инструментарий, с помощью которого осуществляется проектирование.

21.1. Традиционные методы описания проектов

- Описание логическими (булевыми) уравнениями.
- Описание схемами.

При проектировании сложных устройств использование этих методов невозможно из-за трудоемкости, поэтому современные проекты разрабатываются на более высоком уровне абстракции: описание с помощью редакторов архитектурного уровня (HDL-дизайнеры) и языков проектирования.

Таким образом, проектирование современных устройств ВТ традиционными методами *автоматизировано* (HDL-дизайнеры, текстовые редакторы, компиляторы, редакторы временных диаграмм (вэформеры), симуляторы, тест-бенчеры, синтезаторы, оптимизаторы конфигурирования (программирования) проекта в чип и т. д.).

21.2. Типы и уровни описания сложных проектов

Типы (рис. 21.1): структурный (structure) и поведенческий или функциональный (behavior).

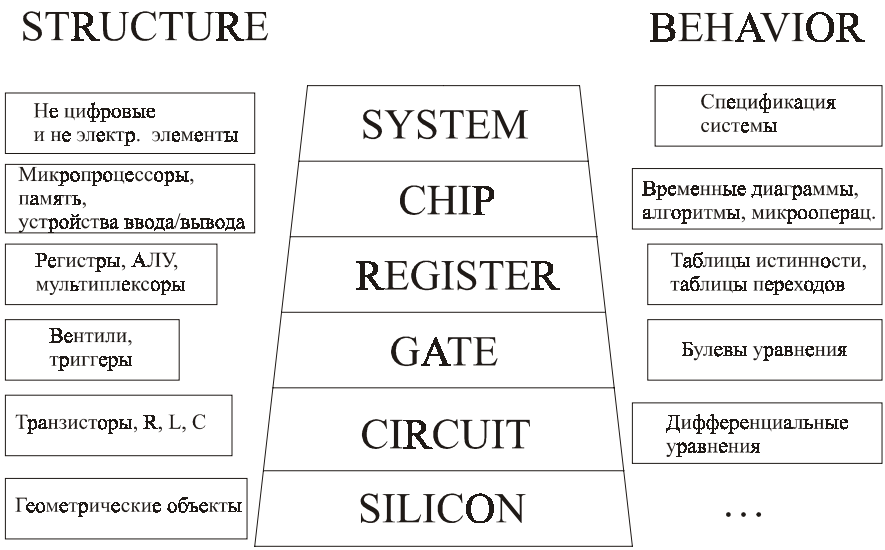


Рис. 21.1. Типы и уровни описания проектов

Уровни:

1. Системный уровень (system) — формируется спецификация системы в целом (требования к системе), здесь используется язык UML (Unified Modeling Language).

   При структурном типе описания на этом уровне задаются функциональные блоки, связывающие их сигналы и драйверы этих сигналов.

   При поведенческом типе описания задаются команды и их формат.

2. Кристаллический уровень (архитектурный, chip) — создаются спецификации для отдельных элементов проекта.

3. Уровень RTL (Register Transfer Level) — здесь базовыми элементами описания являются регистры, защелки, ALU, компараторы, схемы памяти, MUX, DC, DMX и т. д.

4. Уровень Gate — базовые элементы схемы малой интеграции (вентили ("И-НЕ", "ИЛИ-НЕ", "2И", "2 И-НЕ" и т. д.); триггеры, повторители, инверторы и т. д.).

5. Уровень Circuit — базовыми являются электрические элементы.

6. На уровне Silicon определяется топология схемы.

21.2.1. Области применения методов проектирования

Рассмотрим область применения языков описания устройств ВТ (HDL — Hardware Description Languages) при их проектировании, а также области применения других методов проектирования.

Здесь HDL могут одновременно оперировать и структурным и функциональным типами описания устройства (рис. 21.2).

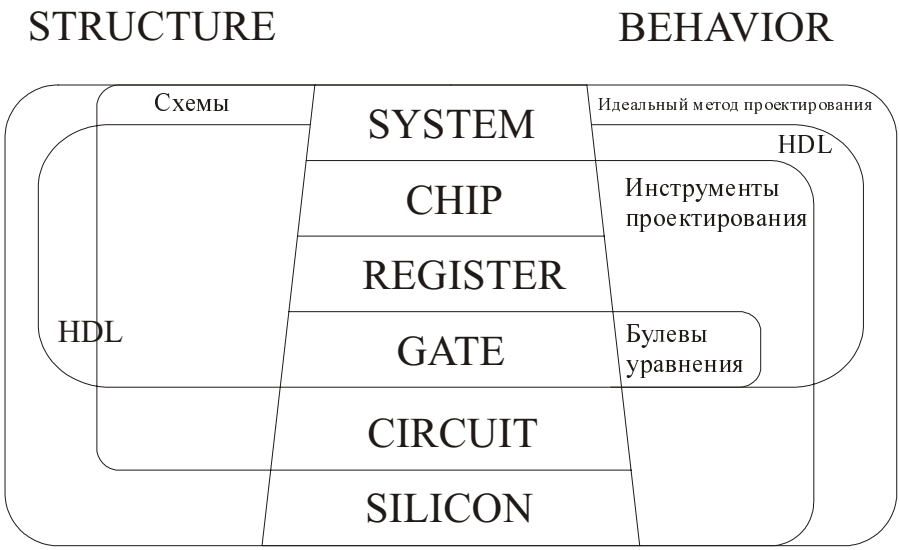


Рис. 21.2. Области применения методов проектирования

21.3. Основные концепции языка VHDL

Название языка VHDL расшифровывается как VHSIC (Very High-Speed Integrated Circuits, язык описания аппаратуры на очень высокоскоростных интегральных схемах) HDL.

Первая концепция — использование VHDL направлено на:

1. Создание описания интерфейсов устройств.
2. Создание описания функционирования устройств.
3. Создание описания структуры устройств.
4. Тестирование.

Замечание

Предполагается, что первые четыре пункта выполняет разработчик, следующие пункты автоматизированы и их выполняет система автоматизированного проектирования (САПР).

5. Анализ проекта.
6. Детализация.
7. Симуляция.
8. Синтез.

Вторая концепция — VHDL поддерживает три стиля описания аппаратных архитектур:

1. Structural description (структурный стиль — иерархия связных компонентов).
2. Data-flow description (потоковый стиль — множество параллельных операций, управляемых сигналами).
3. Behavioral description (поведенческий стиль — последовательные программные предложения (процессы, процедуры)).

Первые три пункта первой концепции были рассмотрены ранее, поэтому рассмотрим подробно пункты с четвертого по восьмой.

21.3.1. Тестирование

Для автоматизации тестирования предусмотрена возможность написания модели теста (VHDL Test Bench), включающей тестируемое устройство как компонент (UUT — Unit Under Test) и сам тест (Test), назначающий сигналы на входы этого компонента.

В дальнейшем сигналы с выходов устройства, полученные в результате симуляции, могут сравниваться с ожидаемыми выходными сигналами.

Рассмотрим пример регрессионного тестирования.

Регрессионное тестирование — это проверка корректности усовершенствованной, измененной или уточненной модели.

Test Bench может задавать сигналы на входах и проверять сигналы на выходах на любой стадии создания проекта, то есть можно уточнять описание устройства, менять описание на более подробное, а функциональная проверка остается прежней. Например, необходимо убедиться, что структурная модель работает так же, как и поведенческая более высокого уровня, для этого:

- в модель теста включают две версии проекта: поведенческую и структурную более низкого уровня;

- на входы обеих моделей подаются одинаковые значения;
- проверяется соответствие выходных сигналов поведенческой и структурной версий проекта.

Для проверки соответствия используется специальный оператор проверки условия: **assert**.

Пример регрессионного теста:

```
architecture regression of test_bench is
  signal d0, d1, d2, d3, en, clk : bit;
  signal q0a, q1a, q2a, q3a, q0b, q1b, q2b, q3b : bit;
begin
  dut_a : entity work.reg4(struct)
    port map (d0, d1, d2, d3, en, clk, q0a, q1a, q2a, q3a);
  dut_b : entity work.reg4(behav)
    port map (d0, d1, d2, d3, en, clk, q0b, q1b, q2b, q3b);
stimulus: process is
begin
  d0 <='1'; d1 <='1'; d2 <='1'; d3 <='1'; wait for 20 ns;
  en <='0'; clk <='0'; wait for 20 ns;
  en <='1'; wait for 20 ns;
  clk <='1'; wait for 20 ns;
  . . .
  wait;
end process stimulus;
. . .
verify: process is
begin
  wait for 10 ns;
  assert q0a = q0b and q1a = q1b and q2a = q2b and q3a = q3b
    report "implementations have different outputs"
    severity error,
  wait on d0, d1, d2, d3 en, clk;
end process verify;
end architecture regression;
```

Оператор `assert` используется для проверки выполнения определенного условия и выдачи сообщения, если условие нарушено.

Синтаксис этого оператора выглядит следующим образом:

```
assert condition
    [report report_string]
    [severity severity_level];
```

Строка `report_string` должна быть строковой переменной или выражением. Это сообщение будет выдано, если условие нарушено. Если часть [**report** `report_string`] опущена, то по умолчанию будет выдано сообщение `Assertion violation` (наличие нарушения). Симулятор может остановить выполнение программы, если условие нарушено и `severity_level` (уровень серьезности) выше определенного порогового значения. Обычно порог может быть задан пользователем. Если в операторе присутствует ключевое слово **report**, то `report_string` содержит текст выдаваемого сообщения.

21.3.2. Анализ

Включает проверку синтаксиса и семантики VHDL-программы, при этом каждый элемент анализируется отдельно, а затем они помещаются в библиотеку проекта.

21.3.3. Детализация

Происходит "развертка" иерархии модели. Процесс развертывания повторяется рекурсивно и заканчивается на полностью поведенческих моделях.

В результате проект представляется набором процессов и взаимосвязанных сигналов.

21.3.4. Симуляция

Симуляция представляет собой выполнение процессов в детализированной модели.

Время симуляции отсчитывается по дискретным событиям (event). Это *модельное время*.

Событием называется изменение значение сигнала. Каждый сигнал имеет свой драйвер, состоящий из транзакций.

Транзакция — это пара: время-событие (здесь время определяется двумя соседними транзакциями).

Для понимания введенных терминов, подумайте над фразой: "Расскажи о своем драйвере на предстоящую неделю и особенно остановись на транзакциях в четверг". Она странная, но не лишена смысла.

Рассмотрим симуляцию процессов, а также *алгоритм симуляции*.

Первый раз процесс активизируется во время инициализации симуляции. Он выполняет все последовательные операторы, а затем повторяет выполнение сначала. Процесс может остановить выполнение (зависнуть, заснуть, заблокироваться и т. д.) при выполнении оператора **wait**.

Синтаксис этого оператора выглядит следующим образом:

```
wait [on sensitivity_list] [until condition]
     [for time_expression]
```

В операторе **wait** определяется список сигналов, к которым процесс будет чувствителен (sensitivity_list) в заблокированном состоянии. Когда значение любого из этих сигналов изменяется, процесс возобновляет выполнение операторов ("развисает", оживает, просыпается и т. д.) и проверяется условие. Если условие выполнилось (или вообще отсутствует), то продолжается выполнение следующих операторов. В противном случае процесс снова зависает. Если в операторе **wait** опущен список сигналов, то процесс чувствителен ко всем сигналам, упомянутым в условии.

Если время ожидания не ограничено (time_exprerssion опущено), то процесс может ждать бесконечно. Если в операторе **wait** указана положительная продолжительность в time_exprerssion, то процесс не будет ждать дольше указанного времени.

Если список сигналов (sensitivity_list) включен в заголовок процесса, то подразумевается, что процесс имеет неявный оператор **wait** последним оператором тела процесса. В этом случае процесс может не содержать никаких явных операторов **wait**.

Процесс возобновляется при изменении входных сигналов, указанных явно в заголовке или неявно в операторе **wait**. Он назначает сигналы на выходы и намечает новые транзакции. Если значения на выходах изменились, возникает новое событие.

Далее приведен краткий алгоритм симуляции.

1. Фаза инициализации.
   - Сигналы получают начальные значения.
   - Время симуляции устанавливается в ноль (0).
   - Каждый процесс:
     - ◊ активизируется;
     - ◊ выполняется до оператора **wait** и блокируется;
     - ◊ обычно назначаются транзакции на будущее время.
2. Цикл симуляции.
   - Время продвигается до времени следующей транзакции.
     - ◊ Значение сигнала обновляется.
     - ◊ Если значение изменилось — событие.
     - ◊ Все процессы, чувствительные к любому из этих событий, или у которых истекло время ожидания в операторе wait for…:

- ◊ возобновляются;
- ◊ выполняются до оператора `wait` и блокируются.

3. Симуляция заканчивается, когда заканчиваются все назначенные транзакции.

21.3.5. Синтез

В процессе синтеза регистровая модель устройства (Register Transfer Level — RTL) переводится в модель на уровне вентилей и триггеров (GATE).

Не все конструкции языка VHDL могут быть реализованы в логике, поэтому программа (VHDL-модель устройства) на входе синтезатора должна состоять из ограниченного подмножества языка. Кроме того, интерпретация VHDL-кода зависит от конкретного синтезатора.

Логический синтез дает следующие преимущества:

- ❒ уменьшается время проектирования;

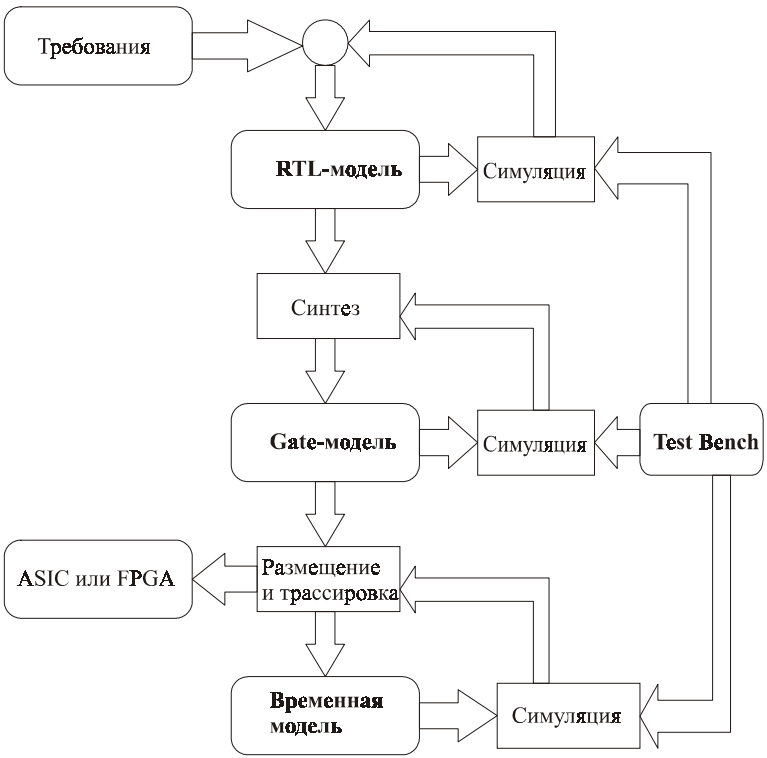


Рис. 21.3. Методология проектирования

- легче вносить изменения в проект (они происходят на более высоком уровне);
- документация в виде кода программы для синтезатора легче воспринимается, чем схема;
- можно выбрать критерии оптимизации (площадь кристалла, скорость);
- отсутствуют ошибки, связанные с переводом на следующий уровень.

Рассмотрим методологию проектирования устройств.

На рис. 21.3 изображена схема, объясняющая этапы проектирования, начиная с RTL-уровня, в случае реализации устройства ВТ на специализированной ИС (ASIC или FPGA).

Здесь ASIC (Application Specific Integrated Circuits) — специализированная ИС, разрабатываемая по индивидуальному заданию.

Лабораторная работа. Приобретение навыков работы с системой Active-CAD и знакомство с программой VHDL Test Bench на примере VHDL-проекта АЛУ

Цель работы

1. Ознакомиться со средством для создания проектов цифровых устройств Active-CAD (иначе Active-VHDL).
2. Применить полученные ранее знания о VHDL для работы в Active VHDL с проектом АЛУ (ALU).
3. Расширить знания о VHDL в области тестирования VHDL-моделей.

Задачи

1. На основе исходного описания АЛУ на языке VHDL создать проект в Active VHDL.
2. Проверить правильность функционирования АЛУ с помощью инструмента симуляторов Active VHDL.
3. Сгенерировать тест для АЛУ средствами Active VHDL.
4. Проверить правильность функционирования АЛУ симуляцией кода VHDL Test Bench.

Программа работы

Active VHDL

Active VHDL — интегрированная среда, предназначенная для создания проектов на VHDL. Ядром системы является VHDL-симулятор. Active VHDL составляет самостоятельную систему, которая позволяет писать, отлаживать и симулировать исполнение VHDL-кода. Основываясь на понятии проекта, Active VHDL позволяет организовать ресурсы проекта в удобную и понятную систему. Знакомство с этим инструментом предлагается на примере проекта АЛУ.

АЛУ

Работа АЛУ описывалась ранее, схему АЛУ смотрите на рис. 21.4.

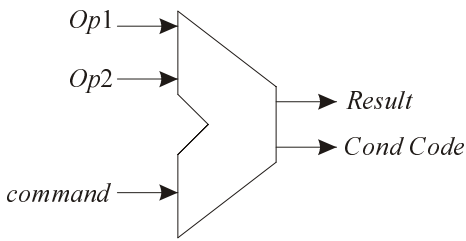


Рис. 21.4. Функциональная схема АЛУ

Создание проекта

1. Запустите Active VHDL.
2. В окне **Getting Started** выберите пункт **Create New Design**.
3. Впишите имя создаваемого проекта.
4. Если расположение проекта вас не устраивает, выберите нужный каталог, нажав кнопку **Browse**.
5. Следующий диалог предлагает выбрать один из вариантов:
   - **Создать новые файлы проекта** — выбрав этот пункт, вы сможете в диалоге включить в проект новые компоненты, задать их интерфейсную часть (названия портов, их типы и пр.), при этом не написав ни одной строки VHDL-кода;
   - **Создать пустой проект** — не содержит никаких файлов;
   - **Импортировать проект** — импорт из Active CAD;
   - **Добавить существующие VHDL-файлы в проект**.

Выберите последний вариант, чтобы добавить в проект готовый VHDL-файл, содержащий описание работы АЛУ.

6. Нажмите кнопку **Add Files** и найдите нужный файл (в данном случае файл с описанием АЛУ) на диске (винчестере). Обратите внимание, что в диалоге **Add Files to Design** есть опция **Make local copy**. По умолчанию в каталоге вашего проекта дублируется выбранный файл, и дальнейшие изменения в файле происходят независимо. Если вы экономите место на диске или хотите, чтобы разные проекты, включающие данный файл, изменялись синхронно, то не устанавливайте эту опцию. В обоих случаях Active VHDL гарантирует корректное управление проектом.
7. Ознакомьтесь с интерфейсом Active VHDL.
   - Окно **Console** отображает сообщения системы (от компилятора, симулятора и т. п.) и содержит несколько вкладок.
   - **Console** — содержит сообщения от всех элементов Active VHDL. В этом окне можно вводить макрокоманды, которые позволяют выполнять все операции, предусмотренные в Active VHDL, не используя графические средства (меню, кнопки и пр.). Например макрокоманды:
     - ◊ создание временной диаграммы — `wave`;
     - ◊ назначение сигналов;
     - ◊ компиляция — `Vcom`;
     - ◊ симуляция и т. п.

     Это позволяет автоматизировать процесс моделирования. Макрокоманды могут выполняться из файла — можно написать командный файл, содержащий сценарий, по которому будет происходить симуляция, например, поочередно с разными тестовыми векторами из разных файлов.
   - **Simulation** — содержит сообщения от симулятора и сообщения, выдаваемые моделью в процессе симуляции.
   - **Compilation** — содержит только сообщения от компилятора. Если вы быстро хотите найти ошибку, которую обнаружил компилятор, щелкните мышью на нужном сообщении об ошибке. Active VHDL откроет файл в текстовом редакторе, в котором обнаружена ошибка, и подчеркнет строку с ошибкой.
   - Окно **Design Browser** показывает содержание всего проекта. Здесь можно добавить или удалить файлы и каталоги, например, с помощью контекстного меню, вызванного нажатием правой кнопки мыши. **Design Browser** позволяет выбрать файл для редактирования в соответствующем окне. Для этого доста-

точно щелкнуть мышью на имени файла. Окно **Design Browser** состоит из трех вкладок.

◊ **Resource Tab** — показывает все файлы ресурсов, находящиеся в каталоге проекта. Файлы разных типов хранятся в разных каталогах. Файлы с расширением log содержат сообщения от соответствующих страничек окна **Console**.

◊ **Files Tab** — показывает содержание рабочей библиотеки и только те файлы ресурсов, которые в данный момент присоединены к проекту. Каждый файл представлен отдельной иконкой. Для текстовых VHDL-файлов цвет иконки несет дополнительную информацию:

- зеленый — файл успешно откомпилирован;
- голубой — файл не откомпилирован или изменен после последней компиляции;
- желтый — ошибки компиляции.

◊ **Structure Tab** — разделена на две части.

- Верхняя часть показывает структуру описываемой модели — все компоненты, все процессы, которыми они описываются, пакеты, подключенные к проекту.
- Нижняя часть показывает сигналы и переменные выбранного в верхней части элемента.

- Окно **HDL Editor** представляет собой обычный текстовый редактор. Ключевые слова, комментарии и прочее выделяются разными цветами.

  Кнопки позволяют, соответственно, закомментировать и раскомментировать целый кусок кода.

8. Для выполнения лабораторной работы необходимо ознакомиться с кодом на VHDL, описывающим АЛУ.

   Пакет `package ALU_32_types` определяет перечислимый тип команд АЛУ: `alu_command`. Данное описание должно быть помещено в отдельный пакет, так как эти команды в дальнейшем будут использоваться не только в описании АЛУ, но и в элементах, обращающихся к АЛУ. Тела пакета (package body) в данном случае не требуется, потому что тип команд полностью определен.

```
---------PACKAGE DECLARATION ALU_32_types ---------------------------
PACKAGE alu_32_types IS
TYPE ALU_command IS (disable,pass1,log_and,log_or,log_xor,log_mask,
                     incr1,add,subtract,multiply,divide);
END alu_32_types;
```

Для описания АЛУ, наряду с пакетом `package ALU_32_types`, используется пакет `package DP32_types` процессора DP32. АЛУ входит в структурную модель процессора DP32 как компонент.

В объявлении пакета `package DP32_types` 8-битными константами задаются коды команд процессора DP32, декларируются вектора, массивы, задаются их типы, объявляются глобальные процедуры и функции.

```
---------PACKAGE DECLARATION dp32_types ------------------------------
PACKAGE dp32_types IS
CONSTANT unit_delay : Time := 1 ns;
--  Subtype Declaration
SUBTYPE bit_32 IS std_logic_vector(31 downto 0);
SUBTYPE CC_bits IS std_logic_vector(2 downto 0);
SUBTYPE cm_bits IS std_logic_vector(3 downto 0);
SUBTYPE bus_bit_32 IS  std_logic_vector(31 downto 0);--bit_32;
SUBTYPE bit_8 IS std_logic_vector(7 downto 0);
--  Type Declaration
TYPE bool_to_bit_table IS ARRAY (boolean) OF std_logic;
TYPE bit_32_array IS ARRAY (integer RANGE <>) OF bit_32;
--  Constant Declaration
CONSTANT bool_to_bit :bool_to_bit_table;
CONSTANT op_add : bit_8 := X"00";
CONSTANT op_sub : bit_8 := X"01";
CONSTANT op_mul : bit_8 := X"02";
CONSTANT op_div : bit_8 := X"03";
CONSTANT op_addq : bit_8 := X"10";
CONSTANT op_subq : bit_8 := X"11";
CONSTANT op_mulq : bit_8 := X"12";
CONSTANT op_divq : bit_8 := X"13";
CONSTANT op_land : bit_8 := X"04";
CONSTANT op_lor : bit_8 := X"05";
CONSTANT op_lxor : bit_8 := X"06";
CONSTANT op_lmask : bit_8 := X"07";
CONSTANT op_ld : bit_8 := X"20";
CONSTANT op_st : bit_8 := X"21";
CONSTANT op_ldq : bit_8 := X"30";
CONSTANT op_stq : bit_8 := X"31";
CONSTANT op_br : bit_8 := X"40";
CONSTANT op_brq : bit_8 := X"50";
```

```
CONSTANT op_bi : bit_8 := X"41";
CONSTANT op_biq : bit_8 := X"51";
--  Function Declaration
function bits_to_int(bits: in std_logic_vector) return integer;
function bits_to_natural(bits: in std_logic_vector) return natural;
--  Procedure Declaration
procedure int_to_bits(int: in integer; bits:out std_logic_vector);
END dp32_types;
```

В теле пакета отдельным константам присваиваются значения и описываются объявленные глобальные процедуры и функции.

```
-------      PACKAGE BODY dp32_types --------------------
PACKAGE BODY dp32_types IS
CONSTANT bool_to_bit :bool_to_bit_table :=
       (false =>'0',true =>'1');
-----------------------------------------------------------------------
function bits_to_int(bits: in std_logic_vector) return integer IS
       variable temp: std_logic_vector(bits'range);
       variable result: integer:=0;
       variable tmp:        integer:=0;
  begin
       if bits(bits'left)='1' then temp:= not bits;
       else temp:=bits;
       end if;
       for index in bits'RANGE loop --(31 downto 0)
              if temp(index)='0' then tmp:=0;
              else  tmp:=1;
              end if;
              result:=result*2 +tmp;
       end loop;
       if bits(bits'left)= '1' then
              result:= (-result)-1;
       end if;
       return result;
  end bits_to_int;
-----------------------------------------------------------------------
function bits_to_natural(bits: in std_logic_vector) return natural is
       variable result: natural:=0;
       variable tmp:        integer:=0;
  begin
```

```
        for index in bits'RANGE loop --(31 downto 0)
                if bits(index)='0' then tmp:=0;
                else  tmp:=1;
                end if;
                result:=result*2 +tmp;
        end loop;
        return result;
  end bits_to_natural;
-----------------------------------------------------------------------
procedure int_to_bits(int: in integer; bits:out std_logic_vector) is
        variable result: std_logic_vector(31 downto 0);
        variable temp: integer;
begin
        if int < 0 then temp:=-(int+1);
        else temp:=int;
        end if;
        for index in bits'reverse_range loop
                if (temp rem 2)=0 then result(index):='0';
                else  result(index):='1';
                end if;
                temp:=temp/2;
        end loop;
        if int<0 then result:= not result;
                result(bits'left):='1';
        end if;
        bits:=result;
end int_to_bits;
-----------------------------------------------------------------------
END dp32_types;
```

Элемент описания `ENTITY ALU_32` обращается к элементам, описанным в пакетах `package ALU_32_types` и `package DP32_types`, поэтому в начале `ENTITY` присутствует строка, подключающая эти пакеты.

```
USE work.dp32_types.all,work.alu_32_types.all;
```

Модуль АЛУ имеет два входных порта операндов, входной порт для команды, выходной порт результата и выходной порт кода условия (`CC`), соответствующего признакам (флагам) нулевого, отрицательного результата и переполнения (`Z`, `N`, `V`). Порт, соответствующий команде, указан как порт перечислимого типа,

то есть на этой стадии проекта кодирование для функций АЛУ не определено. Коды команд задаются глобально при объявлении пакета `package DP32`.

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------
---------------------ENTITY ALU_32 ----------------------
-------------------------------------------------------------------
ENTITY ALU_32 IS
       generic (Tpd:Time:=unit_delay);
       PORT(   operand1 : IN  bit_32;
              operand2  : IN  bit_32;
              result    : OUT bus_bit_32 bus;
              cond_code : OUT CC_bits;
              command   : IN  ALU_command);
END ALU_32;
```

Поведение АЛУ характеризуется в элементе описания `ARCHITECTURE OF ALU_32` процессом `ALU_function`, чувствительным к изменениям в портах операндов и команды.

```
ALU_function: process (operand1, operand2, command)
```

Если команда, которая должна быть выполнена, — арифметическая операция, то операнды вначале конвертируются в тип `integer`.

```
     case command is
     when add|subtract|multiply|divide =>
            a:=bits_to_int(operand1);
            b:=bits_to_int(operand2);
            when incr1 =>
            a:=bits_to_int(operand1);
            b:=1;
            when others =>
                   null;
```

Затем следует оператор выбора команды `case`. Для команды `disable` не выполняется никакое действие, результатом команды `pass1` является `operand1` без изменений. Результат для логических команд получается использованием логических операций языка VHDL над битовыми векторами. Для арифметических команд определен флаг переполнения `cc_V`.

Затем назначаются соответствующий сигнал на шину `result` и оставшиеся флаги. Сигнал подается на `result` только в том случае, если нет команды `disable`.

Элемент описания `ARCHITECTURE OF ALU_32` выглядит следующим образом:

```
------------------------ARCHITECTURE  OF ALU_32 --------------------
ARCHITECTURE behaviour OF ALU_32 IS
       alias cc_V: std_logic is cond_code(2);
       alias cc_N: std_logic is cond_code(1);
       alias cc_Z: std_logic is cond_code(0);
BEGIN
       ALU_function: process (operand1,operand2,command)
              VARIABLE a,b : integer;
              VARIABLE temp_result : bit_32:=(others=>'0');
       begin
              case command is
                     when add|subtract|multiply|divide =>
                            a:=bits_to_int(operand1);
                            b:=bits_to_int(operand2);
                     when incr1 =>
                            a:=bits_to_int(operand1);
                            b:=1;
                     when others =>
                            null;
              end case;
              case command is
                     when disable =>
                                   null;
                     when pass1 =>
              temp_result:= operand1;
                     when log_and=>
              temp_result:= operand1 and operand2;
                     when log_or =>
              temp_result:= operand1 or operand2;
                     when log_xor =>
              temp_result:= operand1 xor operand2;
                     when log_mask =>
              temp_result:= operand1 and not operand2;
                     when add|incr1 =>
              if b>0 and a> integer'high – b then   --positive overflow
              int_to_bits(((integer'low + a)+b)- integer'high –1,temp_result);
                                   cc_V <= '1' after Tpd;
```

```
elsif b<0 and a<integer'low-b then  --negative overflow
int_to_bits(((integer'high + a)+b)- integer'low +1,temp_result);
                    cc_V <= '1'after Tpd;
else int_to_bits(a+b,temp_result);
cc_V <= '0'after Tpd;
end if;
       when subtract =>
if b<0 and a> integer'high + b then  --positive overflow
int_to_bits(((integer'low + a)-b)- integer'high –1,temp_result);
                    cc_V <= '1'after Tpd;
elsif b>0 and a<integer'low+b then  --negative overflow
int_to_bits(((integer'high + a)-b)- integer'low +1,temp_result);
                    cc_V <= '1'after Tpd;
else int_to_bits(a-b,temp_result);
                    cc_V <= '0'after Tpd;
end if;
                    when multiply =>
if ((a>0 and b>0)or (a<0 and b<0))  --result positive
and(abs a > integer'high / abs b) then --positive overflow
int_to_bits(integer'high,temp_result);
                    cc_V <= '1'after Tpd;
elsif ((a>0 and b<0)or(a<0 and b>0))  --result negative
and((- abs a) < integer'low / abs b) then --negative overflow
int_to_bits(integer'low,temp_result);
                    cc_V <= '1'after Tpd;
else int_to_bits(a*b,temp_result);
                    cc_V <= '0'after Tpd;
end if;
              when divide =>
if b=0 then if a>=0 then --positive overflow
int_to_bits(integer'high,temp_result);
else int_to_bits(integer'low,temp_result);
       end if;
                    cc_V <= '1'after Tpd;
else int_to_bits(a/b,temp_result);
                          cc_V <= '0'after Tpd;
end if;
end case;
if command /= disable then
```

```
                        result <=temp_result after Tpd;
                else
                        result <= null after Tpd;
                end if;
--              if (temp_result = X"00000000")then
                cc_Z <=bool_to_bit(temp_result = X"00000000")after Tpd;
                cc_N <=bool_to_bit(temp_result(31) = '1')after Tpd;
--                      cc_Z <='0';
--              else
--                      cc_Z <='1';
--              end if;
--              if (temp_result(31) = '1')then
--                      cc_N <='0';
--              else
--                      cc_N <='1';
--              end if;
        end process ALU_function;
END behaviour;
```

9. Если в программе есть ошибки, то исправьте их.
10. Откомпилируйте проект.

Симуляция с использованием механизма стимуляторов (stimulator)

Функциональная симуляция — первый шаг в процессе тестирования устройства.

Достоинства использования стимуляторов.

- Это самый быстрый, легкий и наглядный способ симуляции сигналов.
- Позволяет мгновенно увидеть реакцию на сигналы.
- Могут быть назначены любому порту или сигналу устройства, в отличие от Test Bench, которые управляют сигналами модуля только верхнего уровня иерархии.
- Используя Stimulator, удобно отлаживать программы, описывающие процессы и архитектуры на нижних уровнях иерархии.

Недостатки использования стимуляторов.

- Stimulator хранятся в специальных файлах awf, предназначены только для Active VHDL и не работают в других VHDL-симуляторах.
- Stimulator не могут выполнять такие сложные задачи, как, например, чтение из файла и т. д.

Выполнение симуляции.

1. До инициализации симулятора надо выбрать **Top-level design entity** (верхний уровень). Для этого в **Files Tab** в библиотеке проекта выберите нужный модуль, щелкните по нему правой кнопкой мыши и выполните команду **Set as Top-level**.
2. Чтобы начать симуляцию, надо инициализировать симулятор, выбрав команду **Initialize Simulation** в меню **Simulation**.
3. После того как симулятор был инициализирован, надо открыть окно редактора временных диаграмм **Waveform Editor** и создать новый файл. **Waveform Editor** показывает результаты симуляции в виде временных диаграмм, а также позволяет создавать тестовые вектора графически, то есть не создавая VHDL-код.
4. Чтобы добавить сигналы в диаграмму, методом drag-and-drop перетащите компонент **Root** из окна **Structure Tab** в окно редактора диаграмм.
5. Теперь следует задать входные сигналы.
   - В окне **Waveform Editor** выберите входные сигналы и нажмите правую кнопку мыши.
   - В контекстном меню выберите команду **Stimulators**. Это простейший способ назначения сигналов.
   - Выберите тип стимулятора **Stimulator Type**:
     - ◊ **Value Stimulators** — позволяют присвоить значение выбранному сигналу. Значение может быть присвоено любому сигналу или шине. Автоматически производится проверка соответствия значения типу выбранного сигнала или порта;
     - ◊ **Formula Stimulators** — позволяют вводить формулы, используя специальные выражения. Например, формула:

       ```
       0 0, 1 10 ns
       ```

       описывает сигнал, который начинается с 0 в момент времени 0, а затем изменяет значение на 1 через 10 нс;
     - ◊ **Clock Stimulators** — задают прямоугольные импульсы;
     - ◊ **Predefined Stimulators** — формулы и периодические сигналы, созданные пользователем для частого использования;
     - ◊ **Custom Stimulators** — создаются по графическому представлению временных диаграмм. В редакторе временных диаграмм их можно копировать, вырезать, перетаскивать и пр.;
     - ◊ **HotKey Stimulators** — по сути, аналогичны **Value Stimulators**, но они позволяют использовать удобный механизм для назначения сигналов. Чтобы изменить значение сигнала, необходимо просто нажать специальную клавишу.

Например, по выбранной вами клавише сигнал будет переключаться между двумя значениями , '0' и '1'.

6. Назначьте произвольное значение `value` для операндов `operand1`, `operand2` (например, `16#3` и т. п.) и для команды `command`.
7. Чтобы заполнить диаграмму, выберите время симуляции в окне **Time to Run** и нажмите кнопку **Run For** на панели инструментов.
8. Завершите симуляцию командой **End Simulation** в меню **Simulation**.

Отслеживание версий

В процессе работы всегда полезно запоминать текущее состояние проекта, особенно перед внесением глобальных изменений. Проектировщик может случайно стереть нужный файл, пойти по неправильно выбранному пути и захотеть вернуться и т. п., поэтому в Active VHDL есть возможность сохранять текущее состояние, создавать версии проекта, используя команду **Backup Revision** в меню **Design**. Версий может быть неограниченное число, каждой версии автоматически при создании присваивается номер. Для того чтобы удобнее было следить за развитием проекта, предоставлена возможность добавлять комментарии к каждой версии. Если возникла необходимость вернуться к сохраненной версии проекта, то в меню **Design** выбирается команда **Restore Revision**.

Сохраните текущую версию проекта.

Генерация программы Test Bench

VHDL Test Bench — это программа, которая описывает изменение входных сигналов на стандартном языке VHDL. Для назначения сигналов на входы существует множество специальных VHDL-функций и языковых конструкций. Данные можно брать из текстового файла, можно создать отдельный процесс, управляющий входами, и т. д.

Обычно создание Test Bench — это создание дополнительного VHDL-файла, который включает тестируемое устройство как компонент (UUT — Unit Under Test) и назначает определенные сигналы на входы этого компонента. На рис. 21.5 представлена укрупненная функциональная модель программы Test Bench.

VHDL Test Bench симулирует входные сигналы и тестирует выходные сигналы устройства. Например, можно создать VHDL-программу, которая пишет сигналы с выходов устройства в текстовый файл и сравнивает их с ожидаемыми выходными сигналами, находящимися в отдельном файле. Такой метод обеспечивает наиболее надежную проверку с минимальным вмешательством пользователя.

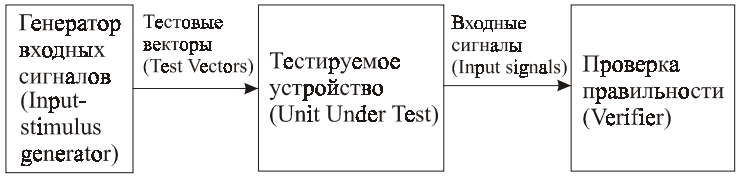


Рис. 21.5. Функциональная модель программы Test Bench

Достоинства VHDL-программы Test Bench.

- Test Bench обладает мощными возможностями по симуляции и верификации.
- Test Bench написана на стандартном языке VHDL, следовательно, можно запустить симуляцию на любом VHDL-симуляторе.
- Test Bench может симулировать сигналы на входах и проверять сигналы на выходах на любой стадии создания проекта, то есть можно уточнять описание устройства, менять описание на более подробное, а функциональная проверка остается прежней.
- Одна и та же программа Test Bench может быть многократно использована с различными тестовыми векторами.

Недостатки программы Test Bench.

- Написание Test Bench занимает много времени, особенно на первых стадиях создания проекта.
- Чтобы написать Test Bench, требуется хорошее знание языка VHDL.

Active VHDL позволяет автоматически генерировать тесты на основе графического представления сигналов (временных диаграмм).

1. Создайте новый пустой файл временных диаграмм.
2. Добавьте в него входные и выходные сигналы АЛУ.
3. Задайте произвольные значения операндов.
4. Задайте временную диаграмму для порта `command`, в которой перебирались бы все значения команд, изменяясь через каждые 100 нс. (Используйте **Formula Stimulator** и описание команд АЛУ в тексте программы: `disable 0 ns`, `pass1 100 ns` и т. д.).
5. Просимулируйте работу программы АЛУ на заданный промежуток времени.
6. Завершите симуляцию командой **End Simulation**.
7. Сохраните на диске получившуюся временную диаграмму, оставив в ней только входные сигналы.

8. Инициализируйте процесс симуляции.
9. В окне Design Browser в библиотеке проекта найдите элемент `ALU_32(begaviour)` и в контекстном меню выберите команду генерации теста (**Generate Test Bench**).

После диалога выбора элемента, для которого будет генерироваться тест, и выбора типа теста, необходимо задать тестовый вектор. Установите опцию **Test Vectors from file**, с помощью команды **Browse** выберите файл временных диаграмм с входными сигналами АЛУ.

1. По завершении генерации теста в вашем проекте появятся новые файлы.

   Автоматически сгенерируется файл с расширением vhd, в котором содержится программа на языке VHDL, реализующая тест АЛУ на основе тестовых векторов, сформированных вами в графическом редакторе временных диаграмм.

2. Ознакомьтесь с текстом программы.

   Test Bench — это обычный объект языка VHDL, не имеющий портов, описанный следующим образом:

```
entity alu_32_tb is
end alu_32_tb;
```

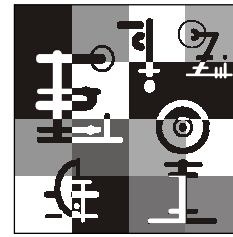
   В его архитектуру тестируемый элемент (АЛУ) включается как компонент UUT (Unit Under Test).

```
component ALU_32
port(
    operand1 : in bit_32;
    operand2 : in bit_32;
    result : out bus_bit_32;
    cond_code : out CC_bits;
    command : in ALU_command
    );
end component;
```

   В программе описывается процесс, чувствительный к изменениям входных сигналов АЛУ.

   Файл с расширением do содержит программу на языке макрокоманд Active VHDL для запуска теста. Этот командный файл компилирует исходные тексты программ, инициализирует симуляцию, создает пустой файл временных диаграмм и добавляет в него временные диаграммы сигналов тестируемого АЛУ.

3. Запустите этот командный файл с помощью команды **Execute** из контекстного меню.
4. Проверьте правильность выполнения команд АЛУ.



Глава 22

Поведенческая VHDL-модель процессора DP32. Ко-симуляция

Рассмотрим гипотетический 32-разрядный RISC-процессор с Гарвардской архитектурой: DP32 (Digital Processor 32-разрядный), а также процесс симуляции команд программы, написанной с помощью команд данного процессора, при этом будем использовать поведенческую модель DP-32.

Командная симуляция (Command simulation) обозначается сокращенно *ко-симуляция* (Co-simulation).

22.1. Описание команд процессора DP32

Процессор DP32 содержит 256 регистров общего назначения (РОН, R), счетчик команд (PC), регистр кода условия (Condition Code register, CC).

РОН программно доступны, а регистры PC и CC недоступны.

R0

. . .

R255
PC

CC

V	N	Z

При включении/перезагрузке PC устанавливается в 0, а остальные регистры не определены.

Память, доступная процессору, состоит из 32-битных слов и адресуется 32-битным адресом.

Команды хранятся в памяти, длина команды кратна 32 битам.

Регистр *PC* содержит адрес следующей команды, которая должна быть выполнена. После выполнения каждой команды *PC* увеличивается на единицу и указывает на следующее слово.

Три бита регистра *CC* обновляются после каждой логической или арифметической операции:

- **Z** (Zero) — бит устанавливается в "1", если результат операции — ноль;
- **N** (Negative) — бит устанавливается в "1", если результат арифметической операции — отрицательный, но не определен для логических операций;
- **V** (oVerflow) — устанавливается в "1", если результат арифметической операции выходит за границы чисел, представимых целым типом `integer`, и не определен для логических операций.

Описание команд процессора DP32 можно условно назвать спецификацией для поведенческой модели DP32. Команды процессора могут быть разного формата.

22.1.1. Арифметические и логические команды DP32

Все арифметические и логические команды имеют следующий формат:

31 24	23 16	15 8	7 0
op	r3	r1	r2/i8

Битовые поля команд:

- `op` (с 24 по 31 бит) — код команды;
- `r3` — адрес регистра-приемника;
- `r1` и `r2` — адреса регистров-источников;
- `i8` — непосредственный операнд типа `integer`.

Таблица 22.1. *Арифметические и логические операции*

Название		Действие
add	сложение	r3←r1+r2
subtract	вычитание	r3←r1-r2
multiply	умножение	r3←r1*r2
divide	деление (нацело)	r3←r1/r2
add quick	быстрое сложение	r3←r1+i8

Таблица 22.1 *(окончание)*

Название		Действие
subtract quick	быстрое вычитание	r3←r1-i8
multiply quick	быстрое умножение	r3←r1*i8
divide quick	быстрое деление	r3←r1/i8
logical and	логическое И	r3←r1& r2
logical or	логическое ИЛИ	r3←r1 | r2
logical exclusive or	исключающее ИЛИ	r3←r1⊕ r2
logical mask	логическая маска	r3←r1& ¬r2
		(¬r2 — инверсия r2)

22.1.2. Команды "чтение из памяти" и "запись в память" DP32

Команды "чтение из памяти" и "запись в память" имеют два разных формата в зависимости от длины смещения.

Формат для длинного смещения:

31 24	23 16	15 8	7 0
op	r3	r1	игнорир.

Смещение (displacement)

Формат для короткого смещения:

31 24	23 16	15 8	7 0
op	r3	r1	i8

Поля:

- op — код команды;
- r3 — регистр, в который надо записать или из которого надо прочитать;
- r1 — индексный регистр;

- `смещение` — непосредственное длинное смещение 32 бита;
- `i8` — непосредственное короткое смещение 8 бит.

Таблица 22.2. *Операции чтения и записи*

Название		Действие
`load`	чтение	`r3←M[r1+disp32]`
`store`	запись	`M[r1+disp32]←r3`
`load quick`	"быстрое" чтение	`r3←M[r1+i8]`
`store quick`	"быстрая" запись	`M[r1+i8]←r3`

22.1.3. Команды "ветвления" DP32

Каждая из четырех команд "ветвления" имеет собственный формат (табл. 22.3).

Таблица 22.3. *Четыре операции ветвления*

Название		Действие
`branch`	ветвление	`if <условие> then` `PC←PC+disp32`
`branch quick`	быстрое ветвление	`if <условие> then` `PC←PC+i8`
`branch indexed`	ветвление с индексированием	`if <условие> then` `PC←r1+disp32`
`branch indexed quick`	быстрое ветвление с индексированием	`if <условие> then` `PC←r1+i8`

Формат обычной команды ветвления (`branch`):

31 24	23 20	19 16	15 8	7 0
`op`	`xxxx`	`ivnz`	`XXXX`	`XXXX`

`Смещение` (displacement)

Формат команды ветвления `branch quick`:

31 24	23 20	19 16	15 8	7 0
op	xxxx	ivnz	XXXX	i8

Формат команды ветвления `indexed branch`:

31 24	23 20	19 16	15 8	7 0
op	xxxx	ivnz	r1	XXXX

Смещение (displacement)

Формат команды ветвления `quick indexed branch`:

31 24	23 20	19 16	15 8	7 0
op	xxxx	ivnz	r1	i8

Поля:

- `op` — код команды;
- `r3` — регистр, в который надо записать или из которого надо прочитать;
- `r1` — индексный регистр;
- `смещение` — непосредственное длинное смещение 32 бита;
- `ivnz` — маска условия (condition mask);
- `i8` — непосредственное короткое смещение (8 бит).

Ветвление происходит, если выполняется:

```
<условие>≡((V&v)|(N&n)|(Z&z))=i
```

22.1.4. Описание пакетов VHDL-модели процессора DP32, коды команд

Пакет `package ALU_32` определяет перечислимый тип команд АЛУ: `alu_command`.

Данное описание должно быть помещено в отдельный пакет, так как эти команды в дальнейшем будут использоваться не только в описании АЛУ, но и в элементах, обращающихся к АЛУ.

Тела пакета (package body) в данном случае не требуется, потому что тип команд полностью определен.

VHDL-код пакета `package ALU_32` помещен в *главе 21*, в указаниях к лабораторной работе.

Пакет `package DP32` необходим для описания работы процессора DP32. В объявлении пакета 8-битными константами задаются коды команд процессора DP32, декларируются векторы, массивы, задаются их типы, объявляются глобальные процедуры и функции. В теле пакета отдельным константам присваиваются значения и описываются объявленные глобальные процедуры и функции (`procedure`, `function`). VHDL-код пакета `package DP32` помещен в *главе 21*, в указаниях к лабораторной работе.

Коды команд DP32, заданные 8-битными константами в описании пакета DP32, выглядят следующим образом — табл. 22.4.

Таблица 22.4. *Таблица команд*

Название		**Действие**	**Код операции**
`add`	сложение	`r3←r1+r2`	**00**
`subtract`	вычитание	`r3←r1-r2`	**01**
`multiply`	умножение	`r3←r1*r2`	**02**
`divide`	деление (нацело)	`r3←r1/r2`	**03**
`add quick`	быстрое сложение	`r3←r1+i8`	**10**
`subtract quick`	быстрое вычитание	`r3←r1-i8`	**11**
`multiply quick`	быстрое умножение	`r3←r1*i8`	**12**
`divide quick`	быстрое деление	`r3←r1/i8`	**13**
`logical and`	логическое И	`r3←r1& r2`	**04**
`logical or`	логическое ИЛИ	`r3←r1 | r2`	**05**
`logical exclusive or`	исключающее ИЛИ	`r3←r1⊕ r2`	**06**
`logical mask`	логическая маска	`r3←r1& ¬r2` (`¬r2` — инверсия `r2`)	**07**

Таблица 22.4 *(окончание)*

Название		Действие	Код операции
load	чтение	r3←M[r1+disp32]	**20**
store	запись	M[r1+disp32]←r3	**21**
load quick	"быстрое" чтение	r3←M[r1+i8]	**30**
store quick	"быстрая" запись	M[r1+i8]←r3	**31**
branch	ветвление	if <условие> then PC←PC+disp32	**40**
branch quick	быстрое ветвление	if <условие> then PC←PC+i8	**50**
branch indexed	ветвление с индексированием	if <условие> then PC←r1+disp32	**41**

Коды команд понадобятся для представления программы в машинных кодах DP32, размещения ее в памяти и дальнейшей ко-симуляции.

По описанию команд процессора DP32 и их форматов можно создать его поведенческую модель на языке VHDL, абстрагируясь от внутренней структуры процессора, и построить для нее Test Bench.

22.2. Ко-симуляция и тестирование процессора DP32

Для написания тестирующей процессор программы (Test Bench) надо дополнительно написать VHDL-модель генератора, который будет управлять входами процессора (сигналы `phi1`, `phi2`, `reset`) и VHDL-модель памяти. В памяти будут храниться данные и тестовая программа, написанная в командах DP32.

Такое написание Test Bench позволяет с помощью ко-симуляции определить набор команд процессора и сравнить возможные альтернативы до выполнения более детального проектирования.

Когда выбрана необходимая поведенческая модель, можно проектировать архитектуру на следующем (более детальном) уровне.

22.2.1. VHDL-модель теста

Чтобы протестировать работу VHDL-модели процессора, нужно на языке VHDL написать программу (листинг 22.1), соответствующую следующей схеме тестирования — рис. 22.1.

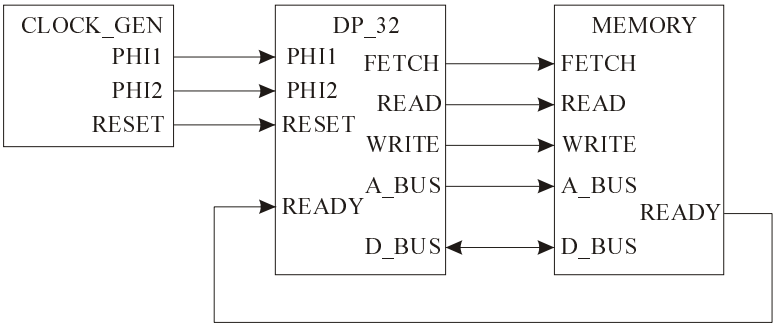


Рис. 22.1. Функциональная схема тестирования DP32

Листинг 22.1. dp32_test

```
USE work.dp32_types.all, work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity dp32_test is
end dp32_test;
architecture structure of dp32_test is
    COMPONENT clock_gen
                 port(phi1,phi2:out std_logic;
                         reset :out std_logic);
    END COMPONENT;
    COMPONENT dp32
                 port (d_bus       :inout bus_bit_32 bus;
                       a_bus       :out bit_32;
                       read,write  :out std_logic;
                       fetch       :out std_logic;
                       ready       :in std_logic;
                       phi1,phi2   :in std_logic;
                       reset       :in std_logic);
    END COMPONENT;
```

```
    COMPONENT memory
                 port(d_bus        :inout bus_bit_32 bus;
                 a_bus             :in bit_32;
                 read,write              :in std_logic;
                 ready                            :out std_logic);
    END COMPONENT;
    SIGNAL  d_bus         : bus_bit_32 bus;
    SIGNAL  a_bus         : bit_32;
    SIGNAL  read,write    : std_logic;
    SIGNAL  fetch         : std_logic;
    SIGNAL  ready         : std_logic;
    SIGNAL  phi1,phi2     : std_logic;
    SIGNAL  reset         : std_logic;

begin
    cg  :  clock_gen
    port map (phi1 =>phi1,phi2 =>phi2,reset =>reset);
    proc:  dp32
    port map (d_bus =>d_bus,a_bus =>a_bus,
              read =>read,write =>write,fetch =>fetch,
              ready =>ready,
              phi1 =>phi1,phi2 =>phi2,reset =>reset);
    mem  :  memory
    port map (d_bus =>d_bus,a_bus =>a_bus,
              read =>read,write =>write,ready =>ready);

end structure;
```

В данном случае под `dp32` может пониматься как поведенческая, так и структурная модели процессора. Какая модель используется для тестирования, будет объявлено в соответствующей конфигурации.

Для тестирования необходимо написать VHDL-модели генератора и памяти (представлены далее), а также написать в кодах DP32 тестовую программу и разместить ее в VHDL-модели памяти.

22.2.2. Описание выполнения ко-симуляции на поведенческой модели DP32

Начинается симуляция, ко-симуляция и тестирование DP32 с выполнения рассмотренной только что подпрограммы "VHDL-модель теста" или `dp32_test` (листинг 22.1).

Верхним уровнем рассматриваемого проекта процессора DP32 будет являться конфигурация `Configuration dp32_behaviour_test` (листинг 22.2), соответствующая поведенческой модели и ссылающаяся на `dp32_test`.

Здесь уместна аналогия с включением и работой компьютера. Сначала начинают выполняться команды BIOS. Для их выполнения одновременно, параллельно во времени начинают работать и генератор тактовых импульсов, и память, и процессор и многое другое, потом кто-то из них переходит в режим зависания, ожидая разрешающего сигнала по шине от процессора или "соседа".

В нашем случае в качестве BIOS выступает программа `dp32_test`. При выполнении этой программы в первую очередь подключаются общие области, описываемые в пакетах (package) `dp32_types` и `alu_32_types`, затем подключается стандартная библиотека IEEE. После начинается выполнение (симуляция) структурной VHDL-программы `dp32_test`: задаются три компонента (процессор (`dp32`), память (`memory`) и генератор (`clock_gen`)) и, так как в тексте отсутствует оператор `process`, начинается одновременная симуляция, или параллельное выполнение во времени, команд сразу трех VHDL-программ (`dp32 (behaviour)`, `memory`, `clock_gen`).

VHDL-программа `clock_gen` (см. листинг 22.3) выдает три сигнала: `reset` — сначала "1", а после 48 наносекунд "0"; `phi1` и `phi2` — каждый по 8 наносекунд, они будут повторяться, отстоя друг от друга на 2 наносекунды, начиная с единичного сигнала `phi1`.

VHDL-программа `memory` (см. листинг 22.4) обнуляет шину данных `d_bus` и сигнал готовности `ready` (то есть память не готова к работе), а затем программа зависает (команда `wait`) и ждет сигналов от процессора либо на чтение (`read=1`), либо на запись (`write=1`).

VHDL-программа `dp32 (behaviour)` (см. листинг 22.5) — поведенческая и описывается с помощью оператора `process`. Вначале задаются необходимые переменные и подпрограммы `procedure`, выполнение (симуляция) `dp32` начинается с реакции на сигнал `reset`, если он равен "1", то соответствующие сигналы, шины и регистр счетчика команд `PC` (см. `dp32 (behaviour)`) обнуляются. Ждем, пока `reset` не станет равным "0", после этого идем в подпрограмму чтения из памяти `procedure memory_rea` по адресу (`a_bus`), соответствующему `PC`, и здесь процессор выставляет для памяти единичный сигнал `read=1`.

VHDL-программа `memory` "развисает" (выходит из ожидания), адрес проверяется на вхождение в адресные границы, и на шину `d_bus` выкладывается содержимое ячейки памяти по адресу `a_bus`, выставляется сигнал `ready=1`, что память готова выдать содержимое процессору. Программа зависает (команда `wait`) и ждет сигнала от процессора, что чтение прошло успешно (`read=0`).

Далее выполняется VHDL-программа `dp32 (behaviour)`, а именно подпрограмма чтения из памяти `procedure memory_read`, и если сигнал `ready` равен "1", то счи-

танная из памяти команда записывается в ячейку текущей команды (инструкции) current_instr, и после того как сигнал phi1 становится равным "1", сигналу read присваивается "0".

VHDL-программа memory "развисает", оператор process выходит на конец и начинает выполняться снова, то есть обнуляет шину данных d_bus и сигнал готовности ready (память не готова к работе), а затем программа зависает (команда wait) и ждет сигналов от процессора либо на чтение (read=1), либо на запись (write=1).

Далее выполняется VHDL-программа dp32 (behaviour) с места, откуда ушли в procedure memory_read, если reset не единица, то уходим в procedure add и увеличиваем значение счетчика команд на "1". То есть в дальнейшем при обращении к памяти будем считывать или выполнять другую операцию уже со следующей ячейкой памяти. Далее следуют стадии декодирования и выполнения текущей команды тестовой программы, а это и есть ко-симуляция (моделирование выполнения команд программы для процессора DP32).

И так далее…

Представленное описание позволит более детально разобраться в поведенческой VHDL-модели процессора DP32. Оно привьет навык написания поведенческих VHDL-моделей других сложных вычислительных устройств.

22.2.3. Конфигурация для VHDL-модели теста поведенческой модели DP32

Листинг 22.2. Configuration dp32_behaviour_test

```
Configuration dp32_behaviour_test of dp32_test is
       for structure
              for cg  : clock_gen
               use entity work.clock_gen (behaviour)
                 generic map (Tpw => 8 ns, Tps => 2 ns);
              end for;
              for mem  : memory
                 use entity work.memory (behaviour);
              end for;
              for proc: dp32
                use entity work.dp32 (behaviour);
              end for;
       end for;
end dp32_behaviour_test;
```

22.2.4. VHDL-модель генератора

Архитектура генератора состоит из двух параллельно выполняемых операторов (листинг 22.3). Один — драйвер сигнала `reset`, а другой — драйвер clock-сигналов, он описывается оператором `process`.

Сигнал `reset` устанавливается в ноль через некоторое время (48 наносекунд) после начала симуляции.

Листинг 22.3. clock_gen

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity clock_gen is
       generic (Tpw  : Time:= 8 ns;
                Tps  : Time:= 2 ns);
       port (phi1,phi2 :out std_logic;
             reset :out std_logic);
end  clock_gen;

ARCHITECTURE behaviour OF clock_gen IS
       Constant clock_period:Time :=2*(Tpw+Tps);
begin
       reset_driver:
              reset <='1','0' after 2*clock_period+Tpw;
       clock_driver:process
       begin
              phi1 <= '1','0' after Tpw;
              phi2 <= '1'after Tpw+Tps,'0' after Tpw+Tps+Tpw;
              wait for clock_period;
       end process clock_driver;
end behaviour;
```

Следует напомнить, что выполнение оператора `process`, описывающего драйвер clock-сигналов, будет каждый раз с приходом к концу повторяться, то есть clock-сигналы (`phi1` и `phi2`) будут повторяться по 8 наносекунд каждый, отстоя друг от друга на 2 наносекунды.

22.2.5. VHDL-модель памяти

Память описывается оператором `process`. Содержимое памяти хранится в массиве. Когда процесс становится активным, шина данных (`d_bus`) отсоединена и сигнал `ready` — "0". Процесс ждет сигналов чтения (`read`) или записи (`write`). Когда приходит один из этих сигналов, выполняется соответствующая команда, но если адреса чтения или записи лежат в пределах адресного пространства (листинг 22.4).

Листинг 22.4. memory

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity memory is
       generic (Tpd  :Time:= unit_delay);
       port (d_bus  :inout bus_bit_32 bus;
             a_bus  :in  bit_32;
             read   :in  std_logic;
             write  :in  std_logic;
             ready  :out std_logic);
end  memory;

ARCHITECTURE behaviour OF memory IS
begin
       process
            constant  low_address :integer :=0;
            constant high_address :integer :=65535;
            type memory_array is
              array(integer range low_address to high_address)of bit_32;
            variable address : integer;
            variable mem :memory_array:=
                     (X"0700_0000",
                      X"1002_0000",
                      X"2102_0000",
                      X"0000_0008",
                      X"1002_0201",
                      X"1101_020A",
                      X"5009_00FA",
                      X"5000_00FA",
                      X"0000_0000",
```

```
                    others =>X"0000_0000");
       begin
              --
              -- put d_bus and reply into initial state
              --
              d_bus <=null after Tpd;
              ready <='0' after Tpd;
              --
              -- wait for a command
              --
              wait until (read='1')or(write='1');
              --
              -- dispatch read or write cycle
              --
              address := bits_to_int(a_bus);
              if address >=low_address and address <=high_address then
address match for this memory
                     if write ='1' then
                            ready<='1' after Tpd;
                            wait until write'0';-- wait until end of write cycle
                            mem(address):=d_bus'delayed(Tpd);-- sample data
from Tpd ago
                     else -- read='1'
                            d_bus <= mem(address) after Tpd;-- fetch data
                            ready <='1' after Tpd;
                            wait until read='0';-- hold for read cycle
                     end if;
              end if;
       end process;
end behaviour;
```

Программа в машинных кодах, предназначенная для ко-симуляции и тестирования, располагается, начиная с первой ячейки памяти. Содержимое остальных обнуляется.

22.2.6. Описание тестовой программы процессора DP32

В VHDL-модель памяти процессора изначально была помещена тестовая программа в машинных кодах:

```
0700_0000,
1002_0000,
2102_0000,
```

```
0000_0008,
1002_0201,
1101_020A,
5009_00FA,
5000_00FA,
0000_0000,
```

Переведем ее на язык символического кодирования:

```
(0) r0←r0& ¬r0
(1) r2←r0+0
(2) M[r0+disp32]←r2
(3) disp32=8
(4) r2←r2+1
(5) r1←r2-10
(6) if (1=V&0|N&0|Z&1) then PC←PC+(-6)
(7) if (0=V&0|N&0|Z&0) then PC←PC+(-6)
(8) r0←r0+r0
```

Словесное описание:

(0) Логическое маскирование, при этом обнуляем содержимое регистра `r0` (`r0=0` независимо от начального значения).

(1) Быстрое сложение: `r2=r0+0`.

(2) Запись в память содержимого `r2` по адресу "содержимое `r0` + длинное смещение, которое находится в следующей команде".

(3) Длинное смещение для предыдущей команды записи в память.

(4) Быстрое сложение: `r2=r2+1`.

(5) Быстрое вычитание: `r1=r2-10`.

(6) Команда ветвления (условный переход): если условие выполняется, то счетчик команд уменьшается на 6.

(7) Команда ветвления (безусловный переход): приведенное условие выполняется всегда независимо от значения регистра кода условия `CC`, т. е. имеем бесконечный цикл.

(8) Сложение: `r0=r0+r0` (невыполняемая ненужная команда).

Описание схемой приведено на рис. 22.2.

Из описания видно, что тестовая программа последовательно пишет в ячейку памяти с восьмым адресом числа от 0 до 9. Запись в ячейку памяти с восьмым адресом

связана с тем, что в первых восьми ячейках памяти находится сама программа (по адресам с 0 до 7), и, чтобы ее не затереть, была выбрана ячейка памяти с восьмым адресом, в которой находится невыполняемая ненужная команда `r0=r0+r0`.

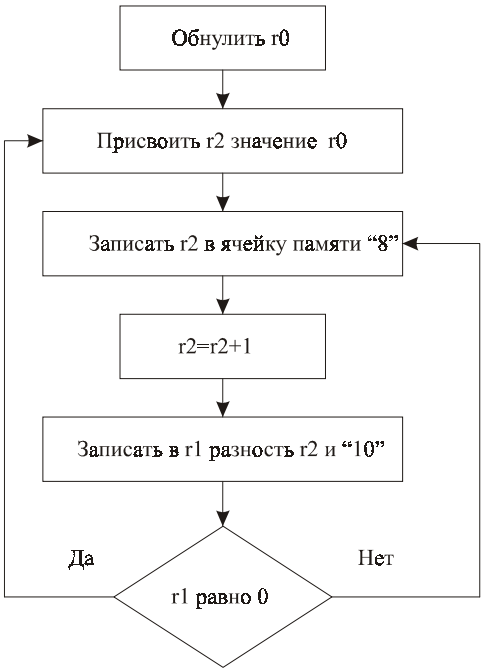


Рис. 22.2. Схема тестовой программы

22.2.7. Описание DP32 поведенческой моделью

Листинг 22.5. dp32 (behaviour)

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity dp32 is

    generic ( Tpd    : Time:= unit_delay);
    port (d_bus    :inout bus_bit_32;
          a_bus    :out bit_32;
          read     :out std_logic;
```

```
          write    :out std_logic;

          fetch    :out std_logic;
          ready    :in std_logic;
          phi1     :in std_logic;
          phi2     :in std_logic;
          reset    :in std_logic);
end dp32;

architecture behaviour of dp32 is
    SUBTYPE reg_addr is natural range 0 to 255;
    TYPE reg_array is array (reg_addr) of bit_32;
-- Здесь объявляется тип для файла регистров
begin
    process
-- Архитектура содержит только один оператор — процесс, который
-- описывает действие процессора  как последовательный алгоритм. В этом
-- процессе вводятся несколько переменных, описывающих внутреннее
-- состояние процессора — файл регистров, счетчик команд, регистр текущей
-- команды и еще несколько рабочих переменных
        VARIABLE reg :reg_array:=(others=>X"00000000");
        VARIABLE PC  :bit_32;
        VARIABLE current_instr :bit_32:=X"00000000";
        VARIABLE op: bit_8;
        VARIABLE r3,r1,r2:reg_addr;
        VARIABLE i8:integer;
        ALIAS cm_i: std_logic is current_instr(19);
        ALIAS cm_V: std_logic is current_instr(18);
        ALIAS cm_N: std_logic is current_instr(17);
        ALIAS cm_Z: std_logic is current_instr(16);
        VARIABLE cc_V,cc_N,cc_Z : std_logic:='0';
        VARIABLE temp_V,temp_N,temp_Z : std_logic:='0';
        VARIABLE displacement : bit_32:=X"00000000";
        VARIABLE effective_addr : bit_32:=X"00000000";

        procedure memory_read (addr         : in bit_32;
                               fetch_cycle  : in boolean;
                               result       :out bit_32) IS
-- Процедура чтения из памяти по определенному адресу
```

```
begin
    --start bus cycle with address output
    a_bus <=addr after Tpd;
    fetch <= bool_to_bit(fetch_cycle)after Tpd;
    wait until phi1 ='1';
    if reset ='1' then
        return;
    end if;
    --
    -- T1 phase
    --
    read <= '1'after Tpd;
    wait until phi1 ='1';
    if reset ='1' then
        return;
    end if;
    --
    -- T2 phase
    --
    loop
        wait until phi2 ='0';
        if reset ='1' then
            return;
        end if;
        -- END of T2
        if ready ='1' then
            result:=d_bus;
            exit;
        end if;
    end loop;
    wait until phi1 ='1';
    if reset ='1' then
        return;
    end if;
    --
    -- Ti phase at end of cycle
    --
    read <='0' after Tpd;
```

```
        end memory_read;

        procedure memory_write (addr    : in bit_32;
                                data    : in bit_32) IS
-- Процедура записи в память по определенному адресу
        begin
            --start bus cycle with address output
            a_bus <=addr after Tpd;
            fetch <= '0'after Tpd;
            wait until phi1 ='1';
            if reset ='1' then
                return;
            end if;
            --
            -- T1 phase
            --
            write <='1'after Tpd;
            wait until phi2 ='1';
            d_bus <=data after Tpd;
            wait until phi1 ='1';
            if reset ='1' then
                return;
            end if;
            --
            -- T2 phase
            --
            loop
                wait until phi2 ='0';
                if reset ='1' then
                    return;
                end if;
                -- END of T2
                exit when ready ='1';
            end loop;
            wait until phi1 ='1';
            if reset ='1' then
                return;
            end if;
            --
```

```
            -- Ti phase at end of cycle
            --
            write <='0' after Tpd;
            d_bus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"  after Tpd;
        end memory_write;

-- Следующие четыре процедуры выполняют арифметические операции над 32
-- битными векторами и выдают результат операции и флаги
        procedure add (result        : inout bit_32;
                       op1,op2       : in integer;
                       V,N,Z         : out std_logic) IS
        begin

            if op2>0 and op1> integer'HIGH –op2 then --positive overflow
                int_to_bits(((integer'LOW+op1)+op2)-integer'HIGH-1,result);
                V:='1';
            elsif op2<0 and op1< integer'LOW –op2 then --negative overflow
                int_to_bits(((integer'HIGH+op1)+op2)-integer'LOW+1,result);
                V:='1';
            else
                 int_to_bits(op1+op2,result);
                 V:='0';
            end if;
            N:=result(31);
            Z:=bool_to_bit(result=X"0000_0000");
        end add;

        procedure subtract (result       : inout bit_32;
                            op1,op2      : in integer;
                            V,N,Z        : out std_logic) IS
        begin
            if op2<0 and op1> integer'HIGH +op2 then --positive overflow
                int_to_bits(((integer'LOW+op1)-op2)-integer'HIGH-1,result);
                V:='1';
            elsif op2>0 and op1< integer'LOW +op2 then --negative overflow
                int_to_bits(((integer'HIGH+op1)-op2)-integer'LOW+1,result);
                V:='1';
            else
                 int_to_bits(op1-op2,result);
```

```
            V:='0';
        end if;
        N:=result(31);
        Z:=bool_to_bit(result=X"0000_0000");
    end subtract;

    procedure divide (result      : inout bit_32;
                      op1,op2     : in integer;
                      V,N,Z       : out std_logic) IS
    begin
        if op2=0 then
            if op1>0 then    --positive overflow
                 int_to_bits(integer'HIGH,result);
            else int_to_bits(integer'LOW,result);
            end if;
            V:='1';
        else
             int_to_bits(op1/op2,result);
             V:='0';
        end if;
        N:=result(31);
        Z:=bool_to_bit(result=X"0000_0000");
    end divide;

    procedure multiply (result    : inout bit_32;
                        op1,op2   : in integer;
                        V,N,Z     : out std_logic) IS
    begin

        if ((op1>0 and op2>0)or(op1<0 and op2<0)
       --result positive
                and (abs op1>integer'HIGH/abs op2)) then
                --positive overflow
            int_to_bits(integer'HIGH,result);
            V:='1';
        elsif ((op1>0 and op2<0)or(op1<0 and op2>0))
       --result negative
                and ((-abs op1)<integer'LOW/abs op2) then
                --negative overflow
```

```
                int_to_bits(integer'LOW,result);
                V:='1';
            else
                 int_to_bits(op1*op2,result);
                 V:='0';
            end if;
            N:=result(31);
            Z:=bool_to_bit(result=X"0000_0000");
        end multiply;

-- Тело процесса: процесс состоит из трех частей, которые повторяются по
-- очереди (начало процесса, взять инструкцию, выполнить инструкцию)
    begin
    --
    -- check for reset active
    --
    if reset ='1' then
        read     <='0' after Tpd;
        write     <='0' after Tpd;
        fetch     <='0' after Tpd;
--        d_bus     <= null after Tpd;
        d_bus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"  after Tpd;
        PC:= X"0000_0000";
-- Все порты в исходном состоянии, шина данных отсоединена
        wait until reset = '0';
    end if;
    --
    --Взять следующую инструкцию
    memory_read(PC,true,current_instr);
    if reset /='1' then
        add (PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
        --Увеличивается счетчик команд
        -- decode & execute
        --Выполнение команды
        op:=current_instr (31 downto 24);
        r3:=bits_to_natural(current_instr (23 downto 16));
        r1:=bits_to_natural(current_instr (15 downto 8));
        r2:=bits_to_natural(current_instr (7 downto 0));
```

```
        i8:=bits_to_int(current_instr (7 downto 0));
        case op is
            when op_add =>
                add(reg(r3),bits_to_int(reg(r1)),bits_to_int(reg(r2)),
                        cc_V,cc_N,cc_Z);
            when op_addq =>
                add(reg(r3),bits_to_int(reg(r1)),i8,cc_V,cc_N,cc_Z);
            when op_sub =>
                subtract(reg(r3),bits_to_int(reg(r1)),bits_to_int(reg(r2)),
                        cc_V,cc_N,cc_Z);
            when op_subq =>
                subtract(reg(r3),bits_to_int(reg(r1)),i8,cc_V,cc_N,cc_Z);
            when op_mul =>
                multiply(reg(r3),bits_to_int(reg(r1)),bits_to_int(reg(r2)),
                        cc_V,cc_N,cc_Z);
            when op_mulq =>
                multiply(reg(r3),bits_to_int(reg(r1)),i8,cc_V,cc_N,cc_Z);

            when op_div =>
                divide(reg(r3),bits_to_int(reg(r1)),bits_to_int(reg(r2)),
                        cc_V,cc_N,cc_Z);
            when op_divq =>
                divide(reg(r3),bits_to_int(reg(r1)),i8,cc_V,cc_N,cc_Z);
            when op_land =>
                reg(r3)    := reg(r1)and reg(r2);
                cc_Z :=bool_to_bit(reg(r3)=X"0000_0000");
            when op_lor =>
                reg(r3)    := reg(r1)or reg(r2);
                cc_Z :=bool_to_bit(reg(r3)=X"0000_0000");
            when op_lxor =>
                reg(r3)    := reg(r1)xor reg(r2);
                cc_Z :=bool_to_bit(reg(r3)=X"0000_0000");
            when op_lmask =>
                reg(r3)    := reg(r1)and not reg(r2);
                cc_Z :=bool_to_bit(reg(r3)=X"0000_0000");
            when op_ld =>
-- При операции чтения вначале читается смещение из памяти, затем
-- счетчик команд увеличивается, чтобы указывать на следующую команду в
-- памяти
```

```
                memory_read(PC,true,displacement);
                if reset /='1' then
                    add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                    add(effective_addr,
                            bits_to_int(reg(r1)),bits_to_int(displacement),
                            temp_V,temp_N,temp_Z);
                    memory_read(effective_addr,false,reg(r3));
                end if;
            when op_ldq =>
                add(effective_addr,
                    bits_to_int(reg(r1)),i8,
                    temp_V,temp_N,temp_Z);
                    memory_read(effective_addr,false,reg(r3));
            when op_st =>
                memory_read(PC,true,displacement);
                if reset /='1' then
                    add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                    add(effective_addr,
                            bits_to_int(reg(r1)),bits_to_int(displacement),
                            temp_V,temp_N,temp_Z);
                    memory_write(effective_addr,reg(r3));
                end if;
            when op_stq =>
                    add(effective_addr,
                            bits_to_int(reg(r1)),i8,
                            temp_V,temp_N,temp_Z);
                    memory_write(effective_addr,reg(r3));
            when op_br =>
-- Команда ветвления вначале читает смещение, увеличивает счетчик
-- команд, затем, если условие ветвления выполнено, счетчику команд
-- присваивается новое значение
                memory_read(PC,true,displacement);
                if reset /='1' then
                    add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                    add(effective_addr,
                        bits_to_int(PC),bits_to_int(displacement),
                        temp_V,temp_N,temp_Z);
                    if ((cm_V and cc_V)or(cm_N and cc_N)or(cm_Z and cc_Z))
```

```
                        =cm_i then
                    PC:= effective_addr;
                end if;
            end if;
        when op_bi =>
            memory_read(PC,true,displacement);
            if reset /='1' then
                add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                add(effective_addr,
                    bits_to_int(reg(r1)),bits_to_int(displacement),
                    temp_V,temp_N,temp_Z);
                if ((cm_V and cc_V)or(cm_N and cc_N)or(cm_Z and cc_Z))
                        =cm_i then
                    PC:= effective_addr;
                end if;
            end if;
        when op_brq =>
                add(effective_addr,
                    bits_to_int(PC),i8,
                    temp_V,temp_N,temp_Z);
                if ((cm_V and cc_V)or(cm_N and cc_N)or(cm_Z and cc_Z))
                        =cm_i then
                    PC:= effective_addr;
                end if;
        when op_biq =>
                add(effective_addr,
                    bits_to_int(reg(r1)),i8,
                    temp_V,temp_N,temp_Z);
                if ((cm_V and cc_V)or(cm_N and cc_N)or(cm_Z and cc_Z))
                        =cm_i then
                    PC:= effective_addr;
                end if;

        when others =>
            assert false report " Hi, illegal instruction" severity warning;
        end case;
    end if; -- reset /='1'
    end process;
end behaviour;
```

Лабораторная работа. Исследование с помощью системы Active-CAD поведенческой VHDL-модели процессора DP32. Ко-симуляция. Тестирование

Цель работы

1. Ознакомиться с поведенческой (функциональной) VHDL-моделью простейшего процессора.
2. Применить полученные ранее знания о работе с системой Active-CAD (Active VHDL) для создания проекта процессора.
3. Наблюдать работу модели средствами Active VHDL.

Задачи

1. На основе исходного описания процессора на языке VHDL создать проект в Active VHDL для поведенческой модели процессора.
2. Разобраться с программой, представленной в модели памяти *(см. главу 22)*.
3. Написать собственную программу, использующую команды DP32.
4. Ко-симулировать эти программы.
5. Проверить правильность функционирования процессора, наблюдая переменные, значения сигналов и прочее в процессе симуляции в среде Active VHDL.

Программа работы

1. Создайте в системе Active VHDL новый проект.
2. Создайте отдельный каталог для файлов поведенческой модели процессора.
3. Добавьте в него файлы:
   - dp32_behav.vhd — поведенческая модель процессора;
   - memory.vhd — модель памяти;
   - clk_gen.vhd — модель генератора тактовой частоты;

- dp32_test.vhd — модель теста;
- dp32_behaviour_test — конфигурация для VHDL-модели теста поведенческой модели DP32;
- alu.vhd — модель АЛУ (там находятся пакеты `package ALU_32` и `package DP32`, необходимые для работы поведенческой модели).

4. Убедитесь в том, что вы правильно поняли описание структуры команд.

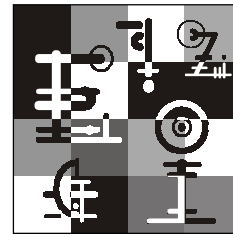
   Для этого выпишите программу, занесенную в память процессора, затем, пользуясь описанием структуры команд, таблицей констант в пакете `dp32_types` и кодом программы, переведите программу в символическое представление.

   Например, код команды `10020000` можно записать в виде `r2←r0+0`.

5. Составьте блок-схему алгоритма программы.
6. Заполните таблицу:

Адрес в памяти	**Код команды**	**Расшифровка команды**
`0000_0001`	`1002_0000`	`r2←r0+0`
. . .		

7. Запустите симуляцию модели (ко-симуляцию программы для DP32).
8. Откройте окно **Watch** (меню **View**), поместите туда ячейку памяти, в которой находится счетчик.
9. В этом окне наблюдайте за изменениями счетчика команд PC.
10. Последовательно выполняя симуляцию по 40 нс, выясните примерное время выполнения одного цикла предложенной программы (до возвращения в начало).
11. В случае появления ошибок во время симуляции, определите и исправьте код VHDL-моделей процессора, памяти или генератора.
12. Напишите собственную программу, использующую команды DP32 (таблица, схема алгоритма).
13. Проведите ко-симуляцию собственной программы.
14. Убедитесь в правильности ее выполнения. При этом проводится функциональное тестирование VHDL-моделей процессора, памяти и генератора.



Глава 23

Архитектура и структурная VHDL-модель процессора DP32. Ко-симуляция

Рассмотрим архитектуру процессора DP32, структурную или RTL (Register Transfer Level) VHDL-модель, написанную по этой архитектуре; а также процесс ко-симуляции, при этом будем использовать структурную модель DP-32.

23.1. Архитектура процессора DP32

Архитектуру процессора DP32 на уровне регистров (register) с описанным набором команд можно представить на рис. 23.1.

DP32 состоит из набора регистров и АЛУ, соединенных шинами $Op1\ Bus$, $Op2\ Bus$ и $R\ Bus$.

Есть также буферы-защелки для интерфейса с шинами $A\ Bus$ (адресной шины) и $D\ Bus$ (шины данных), связывающими процессор и память, а также есть устройство управления (УУ, Control Unit), необходимое для согласования команд процессора.

Программно доступные регистры реализованы с помощью массива регистров (Register file). Его порты $Q1$ и $Q2$ соединены соответственно с шинами $Op1\ Bus$ и $Op2\ Bus$. Адрес ($A2$) для второго порта ($Q2$) обычно берется из поля $r2$ регистра текущей команды, но когда выполняется команда `store` (запись), то используется поле $r3$ с адресом ($A3$). Чтобы разрешить ситуацию с выполнением команд, используется мультиплексор, на который поступают адреса массива регистров $A2$ и $A3$.

Шины $Op1\ Bus$ и $Op2\ Bus$ соединены со входами АЛУ, а выход АЛУ управляет шиной $R\ Bus$ (Result Bus). Результат может быть запомнен в буфере-защелке $Rås$ (Result Latch) и занесен в регистр (то есть в Register file через порт $D3$).

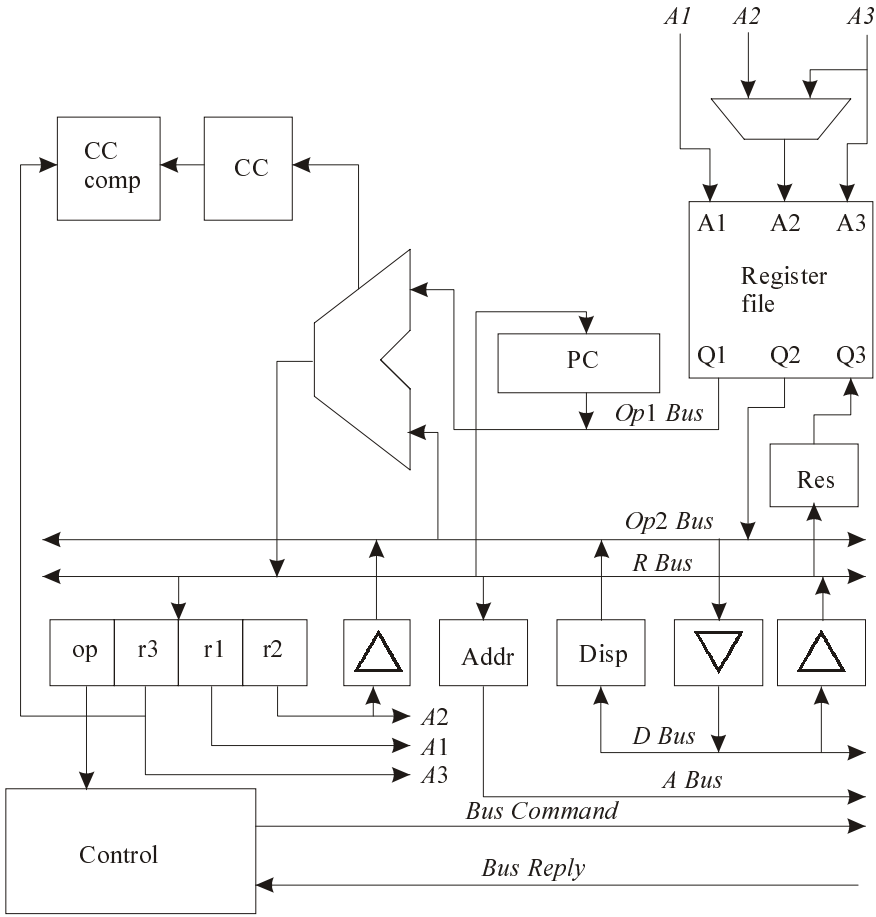


Рис. 23.1. Архитектура процессора DP32 на уровне регистровых передач

Счетчик команд *PC* тоже может управлять шиной *Op*1 *Bus*, а в него могут быть записаны данные с шины *R Bus* (Result Bus).

Флаги, полученные в АЛУ в результате выполнения операций, сохраняются в регистре Кода условия (*CC*) и могут сравниваться в *CC comp* с полем *r*3 (Condition Mask) регистра текущей команды.

Интерфейс шин памяти (*A Bus* и *D Bus*) включает в себя (слева направо):

- защелку *Addr* (Address Latch) для управления шиной адреса *A Bus*;
- защелку для смещения *Disp* (Displacement Latch), управляющую шиной *Op*2 *Bus*;

- буфер для выходных данных (Data Output Buffer), управляемый шиной $Op2\ Bus$;
- буфер для входных данных (Data Input Buffer), управляющий шиной $R\ Bus$ (Result Bus).

Слева от защелки $Addr$ находится защелка $\mathrm{Im}\,med_signext$, управляющая шиной $Op2\ Bus$.

Команда, считанная из памяти, хранится в регистре текущей команды (`instr_latch`). Поля $r1$, $r2$ и $r3$ используются как адреса массива регистров (Register file).

Четыре бита из поля $r3$ используются как маска условия (Condition Mask), а поле op (Opcode Field) используется устройством управления (Control Unit).

23.2. Описание выполнения ко-симуляции на структурной модели DP32

Начинается симуляция, ко-симуляция и тестирование DP32 с выполнения подпрограммы "VHDL-модель теста" или `dp32_test` (листинг 22.1).

Верхним уровнем рассматриваемого проекта процессора DP32 будет являться конфигурация `Configuration dp32_rtl_test` (листинг 23.1), соответствующая структурной модели и ссылающаяся на `dp32_test`.

Здесь так же, как и в случае с поведенческой моделью, уместна аналогия с включением и работой компьютера.

BIOS здесь является программа `dp32_test`, при ее выполнении в первую очередь подключаются общие области, описываемые в пакетах (package) `dp32_types` и `alu_32_types`, затем подключается стандартная библиотека `IEEE`. После начинается выполнение (симуляция) структурной VHDL-программы `dp32_test`: задаются три компонента (процессор (`dp32`), память (`memory`) и генератор (`clock_gen`)) и, так как в тексте отсутствует оператор `process`, начинается одновременная симуляция или параллельное выполнение во времени команд сразу трех VHDL-программ (`dp32(rtl)`, `memory`, `clock_gen`):

1. VHDL-программа `clock_gen` выдает три сигнала: `reset` — сначала "1", а после 48 наносекунд — "0"; `phi1` и `phi2` — каждый по 8 наносекунд, они будут повторяться, отстоя друг от друга на 2 наносекунды, начиная с единичного сигнала `phi1`;
2. VHDL-программа `memory` обнуляет шину данных `d_bus` и сигнал готовности `ready` (то есть память не готова к работе), а затем программа зависает (команда `wait`) и ждет сигналов от процессора либо на чтение (`read=1`), либо на запись (`write=1`);

3. VHDL-программа `dp32(rtl)` — структурная (см. листинг 23.2), в начале задаются все компоненты, представленные в описании архитектуры DP32; затем описываются сигналы, которые соединяют эти компоненты; после начинается описание на VHDL архитектуры процессора, то есть того, как (какими сигналами) соединяются компоненты между собой; далее с помощью оператора `process` описывается то, как должны изменяться друг относительно друга эти сигналы для реализации всех команд процессора.

   При этом декларируется, что мы имеем дело с конечным автоматом или машиной состояний (`state_machine`), и задаются эти состояния. Их одиннадцать:

   - `resetting` — это состояние, связанное со сбросом процессора;
   - `fetch_0`, `fetch_1`, `fetch_2` — состояния, связанные с обращением к памяти;
   - `decode` — состояние расшифровки команды;
   - `disp_fetch_0`, `disp_fetch_1`, `disp_fetch_2` — состояния, связанные с обращением к памяти со смещением;
   - `execute_0`, `execute_1`, `execute_2` — состояния выполнения команд процессора.

4. Далее задаются необходимые переменные и константы и начинается выполнение команд языка VHDL внутри оператора `process`, изменяющих значения сигналов процессора с жесткой временной привязкой к тактовым сигналам `phi1` и `phi2`.

5. В начале описывается состояние `resetting` (оно соответствует единичному значению сигнала `reset`), в этом состоянии все сигналы, разрешающие открытие защелок буферов, обнуляются, это означает, что все компоненты процессора разъединены.

6. Если сигнал `reset` становится равным нулю, то переходим в следующее состояние `fetch_0`, в котором нулевое значение счетчика команд *PC* через шину *Op*1 *Bus*, АЛУ, шину *R Bus* и защелку *Addr* попадает на шину *A Bus*.

7. Далее наступает следующее состояние `fetch_1`, в котором значение *PC* увеличивается на "1" и сигналам `read` и `fetch` присваивается "1".

8. VHDL-программа `memory` "развисает" (выходит из ожидания), адрес проверяется на вхождение в адресные границы, и на шину `d_bus` выкладывается содержимое ячейки памяти по адресу `a_bus`, выставляется сигнал `ready=1`, что память готова выдать содержимое процессору. Программа зависает (команда `wait`) и ждет сигнала от процессора, что чтение прошло успешно (`read=0`).

9. Параллельно во времени выполняется VHDL-программа `dp32(rtl)`: переходим к состоянию `fetch_2`, в котором шина *R Bus* освобождается от ALU, через буфер для входных данных (data input buffer) данные из шины *D Bus* попадают на шину *R Bus*, а считанная из памяти команда записывается в регистр текущей команды (`instr_latch`). Если сигнал `ready` равен "1", то переходим в состояние `decode`, в котором сигналам `fetch` и `read` присваивается "0".
10. VHDL-программа `memory` "развисает", оператор `process` выходит на конец и начинает выполняться снова, то есть обнуляет шину данных `d_bus` и сигнал готовности `ready` (память не готова к работе), а затем программа зависает (команда `wait`) и ждет сигналов от процессора либо на чтение (`read=1`), либо на запись (`write=1`).
11. Параллельно во времени выполняется VHDL-программа `dp32(rtl)`, в которой шины *R Bus* и *D Bus* освобождаются друг от друга и далее следуют стадии декодирования и выполнения (`execute_0`, `execute_1`, `execute_2`) текущей команды тестовой программы, а это и есть ко-симуляция (моделирование выполнения команд программы для процессора DP32).
12. И так далее…

Представленное описание позволит более детально разобраться в структурной (RTL) VHDL-модели процессора DP32. Это описание привьет навык написания RTL VHDL-моделей других сложных вычислительных устройств.

23.3. Конфигурация для VHDL-модели теста структурной модели DP32

Сама VHDL-модель теста для структурной модели будет такой же, что и для поведенческой модели. Отличаются модели тестов конфигурациями, для структурной модели конфигурация выглядит следующим образом — листинг 23.1.

Листинг 23.1. Configuration dp32_rtl_test

```
use work.dp32_types.all;

configuration dp32_rtl_test of dp32_test is
    for structure
        for cg : clock_gen
            use entity work.clock_gen(behaviour)
                generic map (Tpw => 8 ns, Tps => 2 ns);
        end for;
```

```
        for mem : memory
            use entity work.memory(behaviour);
        end for;
        for proc : dp32
            use entity work.dp32(rtl);
            for rtl
                for all : reg_file_32_rrw
                    use entity work.reg_file_32_rrw(behaviour);
                end for;
                for all : mux2
                    use entity work.mux2(behaviour);
                end for;
                for all : latch
                    use entity work.latch(behaviour);
                end for;
                for all : PC_reg
                    use entity work.PC_reg(behaviour);
                end for;
                for all : ALU_32
                    use entity work.ALU_32(behaviour);
                end for;
                for all : cond_code_comparator
                    use entity work.cond_code_comparator(behaviour);
                end for;
                for all : buffer_32
                    use entity work.buffer_32(behaviour);
                end for;
                for all : latch_buffer_32
                    use entity work.latch_buffer_32(behaviour);
                end for;
                for all : signext_8_32
                    use entity work.signext_8_32(behaviour);
                end for;
            end for;
        end for;
    end for;
end dp32_rtl_test;
```

23.4. Структурная VHDL-модель процессора DP32

Представим структурную или RTL-модель процессора DP32, а также VHDL-модели ее компонентов (листинг 23.2). Структурная модель полностью соответствует архитектуре процессора DP32 на уровне регистров.

Рассмотрение работы этой модели было проведено при описании ко-симуляции на структурной модели.

Листинг 23.2. dp32(rtl)

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;
use work.dp32_types.all, work.ALU_32_types.all;


entity dp32 is
    generic ( Tpd    : Time:= unit_delay);
    port (d_bus    :inout bus_bit_32 bus;
          a_bus    :out bit_32;
          read    :out std_logic;
          write    :out std_logic;
          fetch    :out std_logic;
          ready    :in std_logic;
          phi1    :in std_logic;
          phi2    :in std_logic;
          reset    :in std_logic);
end dp32;


architecture RTL of dp32 is
component reg_file_32_rrw
generic (depth : positive);
port (a1 : in std_logic_vector(depth-1 downto 0);
    q1 : out bus_bit_32 bus;
    en1 : in std_logic;
    a2 : in std_logic_vector(depth-1 downto 0);
    q2 : out bus_bit_32 bus;
```

```
    en2 : in std_logic;
    a3 : in std_logic_vector(depth-1 downto 0);
    d3 : in bit_32;
    en3 : in std_logic);
end component;
component mux2
generic (width : positive);
port (i0, i1 : in std_logic_vector(width-1 downto 0);
y : out std_logic_vector(width-1 downto 0);
sel : in std_logic);
end component;
component PC_reg
port (d : in bit_32;
q : out bus_bit_32 bus;
latch_en : in std_logic;
out_en : in std_logic;
reset : in std_logic);
end component;
component ALU_32
port (operand1 : in bit_32;
operand2 : in bit_32;
result : out bus_bit_32 bus;
cond_code : out CC_bits;
command : in ALU_command);
end component;
component cond_code_comparator
port (cc : in CC_bits;
cm : in cm_bits;
result : out std_logic);
end component;
component buffer_32
port (a : in bit_32;
b : out bus_bit_32 bus;
en : in std_logic);
end component;
component latch
generic (width : positive);
port (d : in std_logic_vector(width-1 downto 0);
q : out std_logic_vector(width-1 downto 0);
```

```
en : in std_logic);
end component;
component latch_buffer_32
port (d : in bit_32;
q : out bus_bit_32 bus;
latch_en : in std_logic;
out_en : in std_logic);
end component;
component signext_8_32
port (a : in bit_8;
b : out bus_bit_32 bus;
en : in std_logic);
end component;


signal op1_bus : bus_bit_32;
signal op2_bus : bus_bit_32;
signal r_bus : bus_bit_32;
signal ALU_CC : CC_bits;
signal CC : CC_bits;
signal current_instr : bit_32;
alias instr_a1 : bit_8 is current_instr(15 downto 8);
alias instr_a2 : bit_8 is current_instr(7 downto 0);
alias instr_a3 : bit_8 is current_instr(23 downto 16);
alias instr_op : bit_8 is current_instr(31 downto 24);
alias instr_cm : cm_bits is current_instr(19 downto 16);
signal reg_a2 : bit_8:=(others=>'0');
signal reg_result : bit_32;
signal addr_latch_en : std_logic;
signal disp_latch_en : std_logic;
signal disp_out_en : std_logic;
signal d2_en : std_logic;
signal dr_en : std_logic;
signal instr_latch_en : std_logic;
signal immed_signext_en : std_logic;
signal ALU_op : ALU_command;
signal CC_latch_en : std_logic;
signal CC_comp_result : std_logic;
signal PC_latch_en : std_logic;
```

```
signal PC_out_en : std_logic;
signal reg_port1_en : std_logic;
signal reg_port2_en : std_logic;
signal reg_port3_en : std_logic;
signal reg_port2_mux_sel : std_logic;
signal reg_res_latch_en : std_logic;


begin -- architecture RTL of dp32
reg_file : reg_file_32_RRW
    generic map (depth => 8)
    port map (a1 => instr_a1, q1 => op1_bus, en1 => reg_port1_en,
            a2 => reg_a2, q2 => op2_bus, en2 => reg_port2_en,
            a3 => instr_a3, d3 => reg_result, en3 => reg_port3_en);
reg_port2_mux : mux2
    generic map (width => 8)
    port map (i0 => instr_a2, i1 => instr_a3, y => reg_a2,
                sel => reg_port2_mux_sel);
reg_res_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => reg_result, en => reg_res_latch_en);
PC : PC_reg
    port map (d => r_bus, q => op1_bus,
                latch_en => PC_latch_en, out_en => PC_out_en,
                reset => reset);
ALU : ALU_32
    port map (operand1 => op1_bus, operand2 => op2_bus,
            result => r_bus, cond_code => ALU_CC,
            command => ALU_op);
CC_reg : latch
    generic map (width => 3)
    port map (d => ALU_CC, q => CC, en => CC_latch_en);
CC_comp : cond_code_comparator
    port map (cc => CC, cm => instr_cm, result => CC_comp_result);
dr_buffer : buffer_32
    port map (a => d_bus, b => r_bus, en => dr_en);
d2_buffer : buffer_32
    port map (a => op2_bus, b => d_bus, en => d2_en);
disp_latch : latch_buffer_32
```

```
    port map (d => d_bus, q => op2_bus,
                latch_en => disp_latch_en, out_en => disp_out_en);
addr_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => a_bus, en => addr_latch_en);
instr_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => current_instr, en => instr_latch_en);
immed_signext : signext_8_32
    port map (a => instr_a2, b => op2_bus, en => immed_signext_en);


state_machine: process
    type controller_state is
        (resetting, fetch_0, fetch_1, fetch_2, decode,
        disp_fetch_0, disp_fetch_1, disp_fetch_2,
        execute_0, execute_1, execute_2);
    variable state, next_state : controller_state;
    variable write_back_pending : boolean;
    type ALU_op_select_table is
        array (natural range 0 to 255) of ALU_command;
    constant ALU_op_select : ALU_op_select_table :=
        (16#00# => add,
        16#01# => subtract,
        16#02# => multiply,
        16#03# => divide,
        16#10# => add,
        16#11# => subtract,
        16#12# => multiply,
        16#13# => divide,
        16#04# => log_and,
        16#05# => log_or,
        16#06# => log_xor,
        16#07# => log_mask,
        others => disable);
begin -- process state_machine
    --
    -- start of clock cycle
    --
```

```
wait until phi1 = '1';
--
-- check for reset
--
if reset = '1' then
    state := resetting;
--
-- reset external bus signals
--
read <= '0' after Tpd;
fetch <= '0' after Tpd;
write <= '0' after Tpd;
--
-- reset dp32 internal control signals
--
addr_latch_en <= '0' after Tpd;
disp_latch_en <= '0' after Tpd;
disp_out_en <= '0' after Tpd;
d2_en <= '0' after Tpd;
dr_en <= '0' after Tpd;
instr_latch_en <= '0' after Tpd;
immed_signext_en <= '0' after Tpd;
ALU_op <= disable after Tpd;
CC_latch_en <= '0' after Tpd;
PC_latch_en <= '0' after Tpd;
PC_out_en <= '0' after Tpd;
reg_port1_en <= '0' after Tpd;
reg_port2_en <= '0' after Tpd;
reg_port3_en <= '0' after Tpd;
reg_port2_mux_sel <= '0' after Tpd;
reg_res_latch_en <= '0' after Tpd;
--
-- clear write-back flag
--
write_back_pending := false;
--
else -- reset = '0'
state := next_state;
end if;
```

```
--
-- dispatch action for current state
--
case state is
when resetting =>
--
-- check for reset going inactive at end of clock cycle
--
wait until phi2 = '0';
if reset = '0' then
next_state := fetch_0;
else
next_state := resetting;
end if;
--
when fetch_0 =>
--
-- clean up after previous execute cycles
--
reg_port1_en <= '0' after Tpd;
reg_port2_mux_sel <= '0' after Tpd;
reg_port2_en <= '0' after Tpd;
immed_signext_en <= '0' after Tpd;
disp_out_en <= '0' after Tpd;
dr_en <= '0' after Tpd;
read <= '0' after Tpd;
d2_en <= '0' after Tpd;
write <= '0' after Tpd;
--
-- handle pending register write-back
--
if write_back_pending then
reg_port3_en <= '1' after Tpd;
end if;
--
-- enable PC via ALU to address latch
--
PC_out_en <= '1' after Tpd; -- enable PC onto op1_bus
ALU_op <= pass1 after Tpd; -- pass PC to r_bus
```

```
--
wait until phi2 = '1';
addr_latch_en <= '1' after Tpd; -- latch instr address
wait until phi2 = '0';
addr_latch_en <= '0' after Tpd;
--
next_state := fetch_1;
--
when fetch_1 =>
--
-- clear pending register write-back
--
if write_back_pending then
    reg_port3_en <= '0' after Tpd;
    write_back_pending := false;
end if;
--
-- increment PC & start bus read
--
ALU_op <= incr1 after Tpd; -- increment PC onto r_bus
fetch <= '1' after Tpd;
read <= '1' after Tpd;
--
wait until phi2 = '1';
PC_latch_en <= '1' after Tpd; -- latch incremented PC
wait until phi2 = '0';
PC_latch_en <= '0' after Tpd;
--
next_state := fetch_2;
--
when fetch_2 =>
--
-- cleanup after previous fetch_1
--
PC_out_en <= '0' after Tpd; -- disable PC from op1_bus
ALU_op <= disable after Tpd; -- disable ALU from r_bus
--
-- latch current instruction
--
```

```
dr_en <= '1' after Tpd; -- enable fetched instr onto r_bus
--
wait until phi2 = '1';
instr_latch_en <= '1' after Tpd; -- latch fetched instr from r_bus
wait until phi2 = '0';
instr_latch_en <= '0' after Tpd;
--
if ready = '1' then
    next_state := decode;
else
    next_state := fetch_2; -- extend bus read
end if;
when decode =>
--
-- terminate bus read from previous fetch_2
--
fetch <= '0' after Tpd;
read <= '0' after Tpd;
dr_en <= '0' after Tpd; -- disable fetched instr from r_bus
--
-- delay to allow decode logic to settle
--
wait until phi2 = '0';
--
-- next state based on instr_op of currect instruction
--
case instr_op is
    when op_add | op_sub | op_mul | op_div
        | op_addq | op_subq | op_mulq | op_divq
        | op_land | op_lor | op_lxor | op_lmask
        | op_ldq | op_stq =>
        next_state := execute_0;
    when op_ld | op_st =>
        next_state := disp_fetch_0; -- fetch offset
    when op_br | op_bi =>
        if CC_comp_result = '1' then -- if branch taken
            next_state := disp_fetch_0; -- fetch displacement
        else -- else
            next_state := execute_0; -- increment PC
```

```
        -- past displacement
        end if;
    when op_brq | op_biq =>
        if CC_comp_result = '1' then -- if branch taken
            next_state := execute_0; -- add immed
            -- displacement to PC
        else -- else
            next_state := fetch_0; -- no action needed
        end if;
    when others =>
        assert false report "illegal instruction" severity warning;
        next_state := fetch_0; -- ignore and carry on
end case; -- op
--
when disp_fetch_0 =>
--
-- enable PC via ALU to address latch
--
PC_out_en <= '1' after Tpd; -- enable PC onto op1_bus
ALU_op <= pass1 after Tpd; -- pass PC to r_bus
--
wait until phi2 = '1';
addr_latch_en <= '1' after Tpd; -- latch displacement address
wait until phi2 = '0';
addr_latch_en <= '0' after Tpd;
--
next_state := disp_fetch_1;
--
when disp_fetch_1 =>
--
-- increment PC & start bus read
--
ALU_op <= incr1 after Tpd; -- increment PC onto r_bus
fetch <= '1' after Tpd;
read <= '1' after Tpd;
--
wait until phi2 = '1';
PC_latch_en <= '1' after Tpd; -- latch incremented PC
wait until phi2 = '0';
```

```
PC_latch_en <= '0' after Tpd;
--
next_state := disp_fetch_2;
--
when disp_fetch_2 =>
--
-- cleanup after previous disp_fetch_1
--
PC_out_en <= '0' after Tpd; -- disable PC from op1_bus
ALU_op <= disable after Tpd; -- disable ALU from r_bus
--
-- latch displacement
--
wait until phi2 = '1';
disp_latch_en <= '1' after Tpd; -- latch fetched disp from r_bus
wait until phi2 = '0';
disp_latch_en <= '0' after Tpd;
--
if ready = '1' then
    next_state := execute_0;
else
    next_state := disp_fetch_2; -- extend bus read
end if;
when execute_0 =>
--
-- terminate bus read from previous disp_fetch_2
--
fetch <= '0' after Tpd;
read <= '0' after Tpd;
--
case instr_op is
    when op_add | op_sub | op_mul | op_div
        | op_addq | op_subq | op_mulq | op_divq
        | op_land | op_lor | op_lxor | op_lmask =>
    -- enable r1 onto op1_bus
            reg_port1_en <= '1' after Tpd;
            if instr_op = op_addq or instr_op = op_subq
                or instr_op = op_mulq or instr_op = op_divq then
    -- enable i8 onto op2_bus
```

```
            immed_signext_en <= '1' after Tpd;
        else
-- select a2 as port2 address
            reg_port2_mux_sel <= '0' after Tpd;
-- enable r2 onto op2_bus
            reg_port2_en <= '1' after Tpd;
        end if;
-- select ALU operation
        ALU_op <= ALU_op_select(bits_to_int(instr_op)) after Tpd;
        --
        wait until phi2 = '1';
-- latch cond codes from ALU
        CC_latch_en <= '1' after Tpd;
-- latch result for reg write
        reg_res_latch_en <= '1' after Tpd;
        wait until phi2 = '0';
        CC_latch_en <= '0' after Tpd;
        reg_res_latch_en <= '0' after Tpd;
--
        next_state := fetch_0; -- execution complete
        write_back_pending := true; -- register write_back required
--
when op_ld | op_st | op_ldq | op_stq =>
-- enable r1 to op1_bus
    reg_port1_en <= '1' after Tpd;
    if instr_op = op_ld or instr_op = op_st then
-- enable displacement to op2_bus
        disp_out_en <= '1' after Tpd;
    else
-- enable i8 to op2_bus
        immed_signext_en <= '1' after Tpd;
    end if;
    ALU_op <= add after Tpd; -- effective address to r_bus
--
    wait until phi2 = '1';
    addr_latch_en <= '1' after Tpd; -- latch effective address
    wait until phi2 = '0';
    addr_latch_en <= '0' after Tpd;
--
```

```
            next_state := execute_1;
        --
        when op_br | op_bi | op_brq | op_biq =>
            if CC_comp_result = '1' then
                if instr_op = op_br then
                    PC_out_en <= '1' after Tpd;
                    disp_out_en <= '1' after Tpd;
                elsif instr_op = op_bi then
                    reg_port1_en <= '1' after Tpd;
                    disp_out_en <= '1' after Tpd;
                elsif instr_op = op_brq then
                    PC_out_en <= '1' after Tpd;
                    immed_signext_en <= '1' after Tpd;
                else -- instr_op = op_biq
                    reg_port1_en <= '1' after Tpd;
                    immed_signext_en <= '1' after Tpd;
                end if;
                ALU_op <= add after Tpd;
            else
                assert instr_op = op_br or instr_op = op_bi
                report "reached state execute_0 "
                        & "when brq or biq not taken"
                severity error;
                PC_out_en <= '1' after Tpd;
                ALU_op <= incr1 after Tpd;
            end if;
    --
            wait until phi2 = '1';
            PC_latch_en <= '1' after Tpd; -- latch incremented PC
            wait until phi2 = '0';
            PC_latch_en <= '0' after Tpd;
    --
            next_state := fetch_0;
    --
        when others =>
            null;
    end case; -- op
--
    when execute_1 =>
```

```
    --
    -- instr_op is load or store instruction.
    -- cleanup after previous execute_0
    --
        reg_port1_en <= '0' after Tpd;
        if instr_op = op_ld or instr_op = op_st then
    -- disable displacement from op2_bus
            disp_out_en <= '0' after Tpd;
        else
    -- disable i8 from op2_bus
            immed_signext_en <= '0' after Tpd;
        end if;
        ALU_op <= add after Tpd; -- disable ALU from r_bus
--        ALU_op <= disable after Tpd; -- disable ALU from r_bus    --
    -- start bus cycle
    --
        if instr_op = op_ld or instr_op = op_ldq then
            fetch <= '0' after Tpd; -- start bus read
            read <= '1' after Tpd;
        else -- instr_op = op_st or instr_op = op_stq
            reg_port2_mux_sel <= '1' after Tpd; -- address a3 to port2
            reg_port2_en <= '1' after Tpd; -- reg port2 to op2_bus
            d2_en <= '1' after Tpd; -- enable op2_bus to d_bus buffer
            write <= '1' after Tpd; -- start bus write
        end if;
    --
        next_state := execute_2;
    --
    when execute_2 =>
        --
        -- instr_op is load or store instruction.
        -- for load, enable read data onto r_bus
        --
        if instr_op = op_ld or instr_op = op_ldq then
            dr_en <= '1' after Tpd; -- enable data to r_bus
            wait until phi2 = '1';
        -- latch data in reg result latch
            reg_res_latch_en <= '1' after Tpd;
            wait until phi2 = '0';
```

```
            reg_res_latch_en <= '0' after Tpd;
            write_back_pending := true; -- write-back pending
        end if;
    --
        next_state := fetch_0;
    --
    end case; -- state
end process state_machine;
end RTL;
```

23.4.1. Мультиплексор (MUX)

Входы (рис. 23.2) — два битовых вектора $i0$ и $i1$, вход $select$ типа бит. Выходной вектор — y.

Размерность векторов определяется константой `width`, значение которой определяется, когда мультиплексор используется в структурном описании процессора. Архитектура содержит назначение сигнала, при этом значение входа $select$ используется для того, чтобы определить, какой из двух входов соединен с выходом. Такое назначение сигнала чувствительно ко всем входным сигналам, так что при изменении любого из них этот оператор будет выполняться (листинг 23.3).

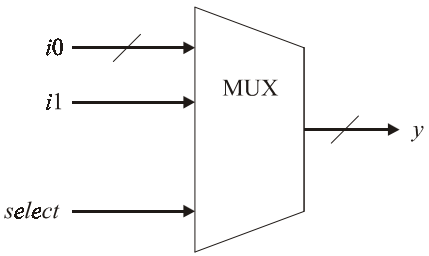


Рис. 23.2. Схема мультиплексора

Листинг 23.3. mux2

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity mux2 is
    generic (width: positive;
            Tpd: Time:= unit_delay);
    port (i0,i1 :in std_logic_vector(width-1 downto 0);
            y    :out std_logic_vector(width-1 downto 0);
            sel    :in std_logic);
end mux2;
architecture behaviour of mux2 is
begin
    with sel select
```

```
        y <=    i0 after Tpd when '0',
                i1 after Tpd when '1',
                "XXXXXXXX" after Tpd when others;
end behaviour;
```

23.4.2. Защелка (Transparent Latch)

Входы (рис. 23.3) — *enable* типа бит, битовый вектор d, выход — битовый вектор q.

Размерность векторов определяется константой width, значение которой определяется, когда мультиплексор используется в структурном описании процессора. Архитектура содержит процесс, чувствительный к изменению входов d и *enable*. Когда вход *enable* равен '1', изменения на входе передаются на выход. Однако когда вход *enable* меняется на '0', то все изменения входа игнорируются и на выход подается прежнее значение (листинг 23.4).

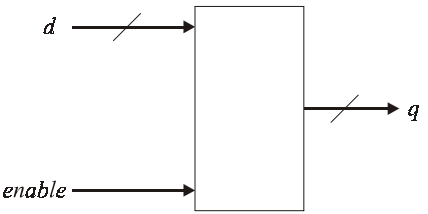


Рис. 23.3. Схема регистра-защелки

Листинг 23.4. latch

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity latch is
    generic (width: positive;
            Tpd: Time:= unit_delay);
    port (d :in std_logic_vector(width-1 downto 0);
          q :out std_logic_vector(width-1 downto 0);
         en    :in std_logic);
end latch;
architecture behaviour of latch is
begin
    process(d,en)
    begin
        if en='1' then
            q <= d after Tpd;
        end if;
    end process;
end behaviour;
```

23.4.3. Буфер (Buffer)

Имеет вход *enable*, вход *a* — бит-вектор, выход *b* — бит-вектор (рис. 23.4).

Поведение буфера описывается процессом, реагирующим на изменение сигналов *enable* и *a*. Если вход *enable* равен '1', сигнал со входа передается на выход, если *enable* равен '0', то выход отсоединен и значение входа *a* игнорируется (листинг 23.5).

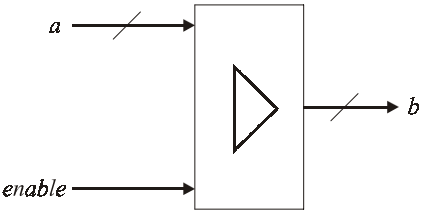


Рис. 23.4. Схема буфера

Листинг 23.5. buffer_32

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity buffer_32 is
    generic (Tpd: Time:= unit_delay);
    port (a :in bit_32;
          b :out bus_bit_32 bus;
         en    :in std_logic);
end buffer_32;
architecture behaviour of buffer_32 is
begin
    b_diver:process(en,a)
    begin
        if en='1' then
            b <= a after Tpd;
        else
            b <= null after Tpd;
        end if;
    end process b_diver;
end behaviour;
```

23.4.4. Защелкивающий буфер (Latching Buffer)

Комбинация простой защелки и буфера (рис. 23.5). Когда вход *Latch_enable* равен '1', изменения входа *d* хранятся в защелке и могут быть переданы на выход. Однако, когда *Latch_enable* изменяется в '0', всякое новое значение *d* игнорируется

и на выход подается хранящееся значение. Вход *Out_enable* управляет передачей хранящегося значения на выход. Когда должно быть запомнено новое значение, вход *Out_enable* равен '0', и на выход ничего не подается (`null`). В отличие от простой защелки здесь должна быть введена переменная для хранения значения, так как выход может быть отсоединен, когда приходит новое значение (листинг 23.6).

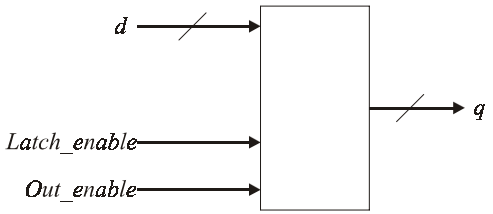


Рис. 23.5. Схема защелкивающего буфера

Листинг 23.6. latch_buffer_32

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity latch_buffer_32 is
    generic (Tpd: Time:= unit_delay);
    port (d :in bit_32;
          q :out bus_bit_32 bus;
    latch_en: in std_logic;
    out_en    : in std_logic);
end latch_buffer_32;
architecture behaviour of latch_buffer_32 is
begin
    process(d,latch_en,out_en)
        variable latched_value :bit_32;
    begin
        if latch_en='1' then
            latched_value:= d;
        end if;
        if out_en='1' then
            q<= latched_value after Tpd;
```

```
        else
            q <= null after Tpd;
        end if;
    end process;
end behaviour;
```

23.4.5. Регистр *PC* (Счетчик команд)

Регистр *PC* (рис. 23.6) — master/slave регистр, он может быть обнулен воздействием на вход *reset*. Если на *reset* ничего не подается, то *PC* действует как защелка. Когда подается '1' на вход *Latch_enable*, значение входа *d* сохраняется (в переменной `master_PC`), но на выход (если *Out_enable* равен '1') подается ранее запомненное значение `slave_PC`. Когда *Latch_enable* изменяется с '1' на '0', значение `slave_PC` обновляется (принимает значение `master_PC`), а любые изменения входа *d* игнорируются (листинг 23.7). Если бы для *PC* использовалась обычная защелка (Transparent Latch), то происходила бы "гонка", так как новое значение сразу передавалось бы на выход взамен старого, влияя на вычисление нового значения.

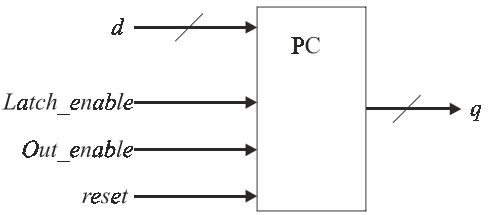


Рис. 23.6. Схема счетчика команд

Листинг 23.7. PC_reg

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity PC_reg is
    generic (Tpd    : Time:= unit_delay);
    port (d        :in bit_32;
          q        :out bus_bit_32 bus;
          latch_en    :in std_logic;
          out_en    :in std_logic;
          reset        :in std_logic);
end PC_reg;
architecture behaviour of PC_reg is
begin
    process(d,latch_en, out_en,reset)
        variable master_PC: bit_32:=X"0000_0000";
```

```
        variable slave_PC: bit_32:=X"0000_0000";
    begin
        if reset='1' then
            slave_PC:=X"0000_0000";
        elsif latch_en = '1' then
            master_PC:=d;
        else
            slave_PC:=master_PC;
        end if;
        if out_en='1' then
            q <= slave_PC after Tpd;
        else
            q <= null after Tpd;
        end if;
    end process;
end behaviour;
```

23.4.6. Регистры общего назначения (Register File — массив регистров)

Значение рабочих регистров может быть изменено только программным путем. Для этого существует 3 порта: 2 для чтения и 1 для записи (рис. 23.7). Каждый порт имеет адресный вход ($a1$, $a2$, $a3$) и разрешающий вход ($en1$, $en2$, $en3$). Порты для чтения соответствуют выходным шинам $q1$ & $q2$, порт для записи соответствует входной шине $d3$.

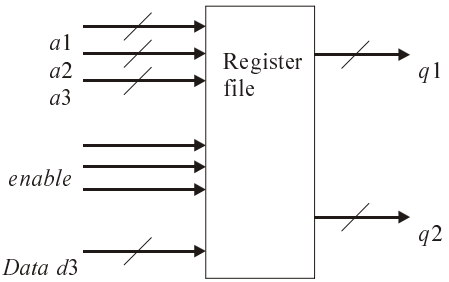


Рис. 23.7. Схема регистров общего назначения

Когда изменяется какой-либо из входов, то вначале проверяется вход *enable* пишущего порта, и, если есть сигнал, адресуемый регистр обновляется. Затем проверяются входы *enable* читающих портов, и, если надо, адресуемые данные передаются на соответствующую выходную шину. В противном случае на выход ничего не подается (листинг 23.8).

Листинг 23.8. reg_file_32_rrw

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
entity reg_file_32_rrw is
    generic (depth    : positive;
             Tpd    : Time:= unit_delay;
             Tac    : Time:= unit_delay);
    port (a1    :in std_logic_vector(depth-1 downto 0);
          q1    :out bus_bit_32 bus;
          en1    :in std_logic;
          a2    :in std_logic_vector(depth-1 downto 0);
          q2    :out bus_bit_32 bus;
          en2    :in std_logic;
          a3    :in std_logic_vector(depth-1 downto 0);
          d3    :in bit_32;
           en3    :in std_logic);
end reg_file_32_rrw;
architecture behaviour of reg_file_32_rrw is
begin
    reg_file:process(a1,en1,a2,en2,a3,d3,en3)
        subtype reg_addr is natural range 0 to depth-1;
        type register_array is array (reg_addr) of bit_32;

        variable registers: register_array:=(others=>X"00000000");

    begin
        if en3='1' then
            registers(bits_to_natural(a3)):=d3;
        end if;
        if en1='1' then
            q1 <= registers(bits_to_natural(a1))after Tac;
        else
            q1 <= null after Tpd;
        end if;
        if en2='1' then
            q2 <= registers(bits_to_natural(a2))after Tac;
        else
            q2 <= null after Tpd;
        end if;
    end process reg_file;
end behaviour;
```

23.4.7. Компаратор (Condition Code Comparator)

Компаратор получает на вход Condition Code с выхода АЛУ и Condition Mask из команды процессора и выдает на выход результат, который будет использоваться командами ветвления.

Действие этого устройства описывается оператором назначения сигнала, чувствительным ко всем входам. При изменении значения на любом из них будет подсчитан новый результат (листинг 23.9).

Листинг 23.9. cond_code_comparator

```
USE work.dp32_types.all,work.alu_32_types.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity cond_code_comparator is
    generic (Tpd    : Time:= unit_delay);
    port (cc    :in CC_bits;
          cm    :in cm_bits;
          result:out std_logic);
end  cond_code_comparator;
ARCHITECTURE behaviour OF cond_code_comparator IS
    alias cc_V: std_logic is cc(2);
    alias cc_N: std_logic is cc(1);
    alias cc_Z: std_logic is cc(0);
    alias cm_i: std_logic is cm(3);
    alias cm_V: std_logic is cm(2);
    alias cm_N: std_logic is cm(1);
    alias cm_Z: std_logic is cm(0);
begin
    result <= bool_to_bit(((cm_V and cc_V)
                        or (cm_N and cc_N)
                        or (cm_Z and cc_Z))=cm_i) after Tpd;
end behaviour;
```

23.4.8. Защелка (*Immed_signext*)

Используется для реализации быстрых операций процессора DP32 (листинг 23.10).

Листинг 23.10. signext_8_32

```
USE work.dp32_types.all,work.alu_32_types.all;

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
entity signext_8_32 is
    generic (Tpd    : Time:= unit_delay);
    port (a        :in bit_8;
          b        :out bus_bit_32 bus;
          en    :in std_logic);
end  signext_8_32;
architecture behaviour of  signext_8_32 is
begin
    b_driver:process(en,a)
    begin
        if en='1' then
            b(7 downto 0)<= a after Tpd;
            if a(7) = '1' then
                b(31 downto 8)<= X"FFFF_FF" after Tpd;
            else
                b(31 downto 8)<= X"0000_00" after Tpd;
            end if;
        else
            b <= null after Tpd;
        end if;
    end process b_driver;
end behaviour;
```

23.4.9. АЛУ

АЛУ VHDL-программа представлена в *главе 21* в описании методических указаний к лабораторной работе, схема АЛУ — на рис. 23.8.

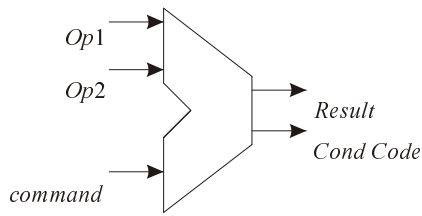


Рис. 23.8. Схема арифметико-логического устройства (АЛУ)

Лабораторная работа. Исследование с помощью системы Active-CAD структурной VHDL-модели процессора DP32. Ко-симуляция. Тестирование

Цель работы

1. Знакомство со структурной (регистровой, RTL) VHDL-моделью простейшего процессора.
2. Применение полученных ранее знаний о работе с системой Active-CAD (Active VHDL) для создания проекта процессора.
3. Наблюдение работы модели средствами Active VHDL.

Задачи

1. На основе исходного описания процессора на языке VHDL создать проект в Active VHDL для структурной модели процессора.
2. Разобраться с программой, представленной в модели памяти *(см. главу 22)*.
3. Написать собственную программу, использующую команды DP32.
4. Ко-симулировать эти программы.
5. Проверить правильность функционирования процессора (функциональное тестирование), наблюдая переменные, значения сигналов и прочее в процессе ко-симуляции в среде Active VHDL.

Программа работы

1. Создайте в системе Active VHDL новый проект.
2. Создайте отдельный каталог для файлов структурной модели процессора.
3. Добавьте в него файлы:
   - dp32_rtl.vhd — структурная модель процессора;

- rtl_test_cfg.vhd — конфигурация для VHDL-модели теста структурной модели DP32;
- dp32_test.vhd — модель теста;
- memory.vhd — модель памяти;
- clk_gen.vhd — модель генератора тактовой частоты;
- alu_32.vhd — модель АЛУ;
- mux2.vhd — модель мультиплексора;
- latch.vhd — модель защелки Transparent Latch;
- buffer32.vhd — модель 32-разрядного буфера;
- ltch_buf.vhd — модель защелкивающего буфера (Latching buffer);
- pc_reg.vhd — модель регистра *PC* (Счетчика Команд);
- reg_file.vhd — модель Регистров общего назначения (Register File — массив регистров);
- cc_comp.vhd — модель Компаратора (Condition Code Comparator);
- signext.vhd — модель Защелки (Immed_signext).

4. Убедитесь в том, что вы правильно поняли описание структуры команд.

   Для этого выпишите программу, занесенную в память процессора, затем, пользуясь описанием структуры команд, таблицей констант в пакете `dp32_types` и кодом программы, переведите программу в символическое представление.

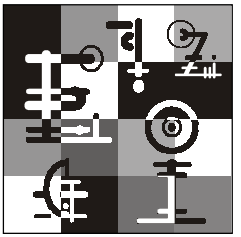
   Например, код команды `10020000` можно записать в виде: `r2←r0+0`.

5. Составьте блок-схему алгоритма программы.
6. Заполните таблицу:

Адрес в памяти	**Код команды**	**Расшифровка команды**
0000_0001	1002_0000	r2←r0+0
. . .		

7. Запустите симуляцию модели (ко-симуляцию программы для DP32).
8. Откройте окно **Watch** (меню **View**), поместите туда ячейку памяти, в которой находится счетчик.
9. В этом окне наблюдайте за изменениями счетчика команд *PC*.

10. Последовательно выполняя симуляцию по 40 нс, выясните примерное время выполнения одного цикла предложенной программы (до возвращения в начало).
11. В случае появления ошибок во время ко-симуляции, определите и исправьте код VHDL-моделей процессора, памяти или генератора.
12. Напишите собственную программу, использующую команды DP32 (блок-схема алгоритма, таблица).
13. Проведите ко-симуляцию собственной программы.
14. Убедитесь в правильности ее выполнения, проведите функциональное тестирование.



Глава 24

Обнаружение и исправление ошибок VHDL-моделей цифровых устройств. VHDL-модели современных процессоров

Рассмотрим ошибки в VHDL-коде моделей DP32 и памяти, которые были сделаны специально для их последующего обнаружения и исправления, также рассмотрим отдельные компоненты современных процессоров и способы их описания и подключения на примере процессора DP32.

24.1. Обнаружение и пути исправления ошибок VHDL-моделей процессора DP32

Проведем отдельный анализ основных (смысловых) ошибок в поведенческой, структурной моделях DP32 и модели памяти, представленных в *главах 22 и 23*.

24.1.1. Обнаружение и пути исправления ошибок в поведенческой VHDL-модели процессора DP32

Рассмотрим фрагмент процедуры поведенческой модели `memory_write`:

```
-- Ti phase at end of cycle
--
write <='0' after Tpd;
d_bus <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"  after Tpd;
```

Здесь, не привязываясь по времени к тактам `ph1` и `ph2`, почти сразу обнуляется сигнал `write` и шина данных переводится в третье состояние.

С другой стороны, рассмотрим фрагмент модели памяти (`memory`):

```
if write ='1' then
ready<='1' after Tpd;
```

```
wait until write='0'; -- wait until end of write cycle
mem(address):=d_bus'delayed(Tpd); -- sample data from Tpd ago
```

Здесь при `write='0'` происходит запись с шины данных в память, но она процессором уже переведена в третье состояние.

В результате то, что было на шине данных, то есть что хотели записать в память, теряется.

Для исправления этой ошибки необходимо между назначением сигналу `write` нулевого значения в поведенческой модели и переводом шины данных в третье состояние (с последующим выходом из процедуры `memory_write`) организовать задержку на один такт, за который память успеет получить управление и записать данные.

24.1.2. Обнаружение и пути исправления ошибок в структурной VHDL-модели процессора DP32

В компоненте `reg_file_32` неверно описан подтип:

```
subtype reg_addr is natural range 0 to depth-1;
type register_array is array (reg_addr) of bit_32;
```

Здесь константа `depth` используется для задания разрядности адресных шин и при таком описании подтипа число регистров будет равно числу разрядов в адресной шине (`depth`), а должно быть два в степени `depth` (в DP32: 256 регистров с адресами от 0 до 255).

В структурной модели `p32_rtl` были обнаружены следующие ошибки.

Первая ошибка:

```
              . . .
case state is
       when resetting =>
              . . .
       when fetch_0 =>
       reg_port1_en <= '0' after Tpd;
       reg_port2_mux_sel <= '0' after Tpd;
       reg_port2_en <= '0' after Tpd;
       immed_signext_en <= '0' after Tpd;
       disp_out_en <= '0' after Tpd;
       dr_en <= '0' after Tpd;
       read <= '0' after Tpd;
       d2_en <= '0' after Tpd;
       write <= '0' after Tpd;
```

Здесь ошибка в очередности последних двух VHDL-команд: получается, что буфер d2_buffer, выдающий данные на шину данных (d_bus), отключается прежде, чем начинается запись данных в память с шины d_bus.

Вторая ошибка:

```
    . . .
when execute_1 =>
    . . .
    ALU_op <= add after Tpd; -- disable ALU from r_bus
```

Состояние (execute_1) предназначено для операций записи в память и чтения из памяти, поэтому АЛУ должно быть отключено от шины r_bus командой disable, команда add здесь ошибочна.

Третья ошибка:

```
-- start bus cycle
        if instr_op = op_ld or instr_op = op_ldq then
            fetch <= '0' after Tpd; -- start bus read
            read <= '1' after Tpd;
        else
            reg_port2_mux_sel <= '1' after Tpd; -- address a3 to port2
            reg_port2_en <= '1' after Tpd; -- reg port2 to op2_bus
```

Здесь ошибка в том, что, благодаря выполнению верхней (над текстом) и нижней (под текстом) команд, преждевременно открывается буфер d2 , передающий данные с шины op2_bus на шину d_bus, в то время как данные на шину op2_bus из файлового регистра еще не установились.

Исправляет ситуацию добавления между этими командами задержки на один такт.

```
            d2_en <= '1' after Tpd; -- enable op2_bus to d_bus buffer
            write <= '1' after Tpd; -- start bus write
        end if;
```

24.1.3. Обнаружение и пути исправления ошибок в VHDL-модели памяти

Первая ошибка:

```
        . . .
begin
        --
        -- put d_bus and reply into initial state
        --
        d_bus <=null after Tpd;
```

В результате выполнения последней команды (команды над текстом) два устройства (memory и d2_buffer), описываемые разными параллельно идущими во времени процессами, будут конфликтовать, одновременно посылая сигналы на одну и ту же шину (d_bus).

Для исправления можно вдвое увеличить задержку в последней команде: вместо Tpd поставить (2* Tpd). В результате шина d_bus обнулится через время (2* Tpd), а не через Tpd, что устранит конфликт с устройством d2_buffer.

Вторая ошибка:

```
        ready <='0' after Tpd;
        --
        -- wait for a command
        --
        wait until (read='1')or(write='1');
        --
        -- dispatch read or write cycle
        --
        address := bits_to_int(a_bus);
        if address >=low_address and address <=high_address then
            -- address match for this memory
            if write ='1' then
                ready<='1' after Tpd;
                wait until write='0'; -- wait until end of write cycle
                mem(address):=d_bus'delayed(Tpd); -- sample data from Tpd ago
            else  -- read='1'
                d_bus <= mem(address) after Tpd;    -- fetch data
```

Ошибка в последней команде (над текстом) аналогична первой и устраняется так же:

```
                ready <='1' after Tpd;
                wait until read='0'; -- hold for read cycle
            end if;
        end if;
    end process;
end behaviour;
```

24.2. VHDL-модели современных процессоров

Рассмотрим отдельные компоненты современных процессоров и способы их описания и подключения на примере процессора DP32.

24.2.1. Реализация проекта конвейера команд на основе поведенческой модели процессора DP32

Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению команды на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры.

Так обработку любой машинной команды можно разделить на несколько этапов (несколько ступеней), организовав передачу данных от одного этапа к следующему.

При этом конвейерную обработку можно использовать для совмещения этапов выполнения разных команд. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах.

Этапы исполнения команды на конвейере

Выполнение типичной команды можно разделить на следующие этапы:

1. Выборка команды — `IF` (по адресу, заданному счетчиком команд, из памяти извлекается команда) и ее декодирование/ выборка операндов из регистров — `ID`.
2. Вычисление эффективного адреса памяти — `EF`.
3. Выполнение операции — `EX`.
4. Обращение к памяти и запоминание результата — `MEM (WB)`.

Приведенное разбиение команды на этапы может быть сделано различными способами, указанный способ не является ни единственным, ни наилучшим, но используется в данном проекте.

Конвейеризация поведенческой модели DP32

Чтобы конвейеризовать DP32, разбиваем выполнение его команд на указанные ранее этапы, отведя для выполнения каждого этапа один такт синхронизации, и начинаем в каждом такте выполнение новой команды.

Естественно, для хранения промежуточных результатов каждого этапа необходимо использовать промежуточные переменные.

Хотя общее время выполнения одной команды в таком конвейере будет составлять четыре такта, в каждом такте в совмещенном режиме будет выполняться четыре различных команды.

Каждый этап представим в виде процесса. Все процессы, работая параллельно, будут реализовывать механизм конвейеризации. Подсчитаем общее необходимое количество процессов: 4 — для каждого этапа выполнения команды, 1 — для обеспе-

чения механизма перезапуска процессора (`Reset`) и 2 — для разрешения конфликтов (по памяти и по регистрам).

Все процессы будут запускаться по приходу определенных сигналов, синхронизирующих их работу. Таким образом, информация между процессами будет передаваться в виде сигналов, а внутри самих процессов будут определяться переменные, соответствующие данным сигналам.

24.2.2. Реализация защищенного режима

Рассмотрим защищенный режим на примере процессора i80286, а затем опишем его реализацию в процессоре DP32.

Защищенный режим на примере процессора i80286

Процессор i80286 и более поздние процессоры поддерживают два режима операций: защищенный режим и реальный режим.

Реальный режим совместим с работой процессора i8086 и позволяет прикладной программе адресоваться к памяти объемом до 1 Мбайт.

Защищенный режим расширяет диапазон адресации до 16 Мбайт.

Основное отличие между реальным и защищенным режимом заключается в способе преобразования процессором логических адресов в физические.

Логические адреса — это адреса, используемые в прикладной программе. Как в реальном, также и в защищенном режиме, логический адрес — это 32-разрядное значение, состоящее из 16-битового селектора (адреса сегмента) и 16-битового смещения.

Физические адреса — это адреса, которые процессор использует для обмена данными с компонентами системной памяти. В реальном режиме физический адрес представляет собой 20-битовое значение, а в защищенном режиме — 24-битовое.

Когда процессор обращается к памяти (для выборки инструкции или записи переменной), он генерирует из логического адреса физический адрес.

В реальном режиме генерация физического адреса состоит из сдвига селектора (адреса сегмента) на 4 бита влево (это означает умножение на 16) и прибавления смещения. Полученный в результате 20-разрядный адрес используется затем для доступа к памяти (рис. 24.1).

Чтобы получить физический адрес в защищенном режиме, селекторная часть логического адреса используется в качестве индекса таблицы дескрипторов. Запись в таблице дескрипторов содержит 24-битовый базовый адрес, к которому затем для образования физического адреса прибавляется смещение логического адреса (рис. 24.2).

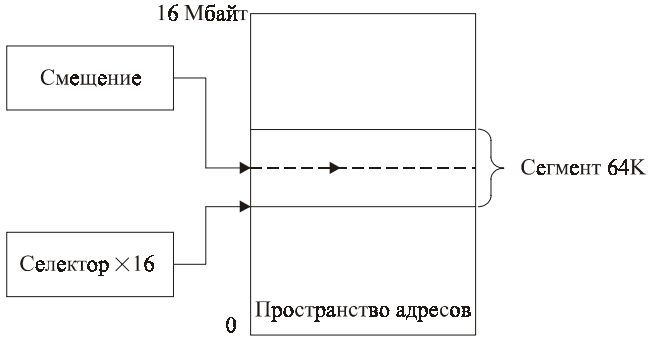


Рис. 24.1. Генерация физического адреса в реальном режиме

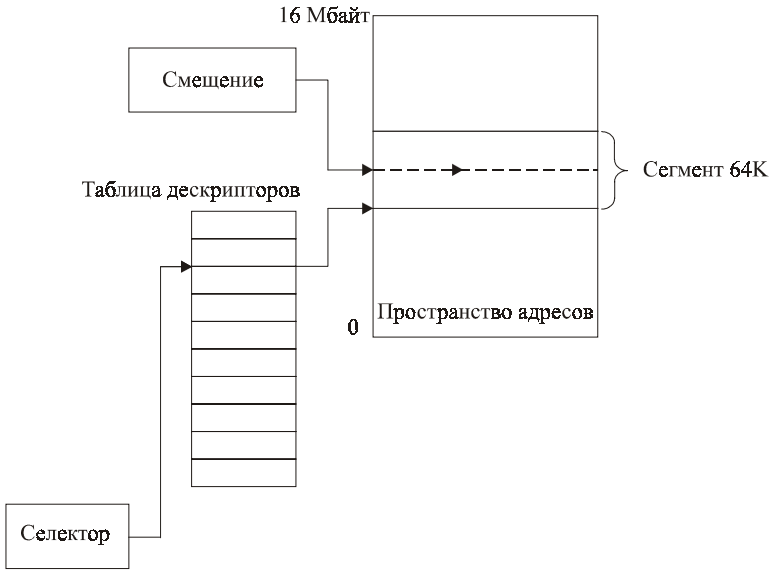


Рис. 24.2. Генерация физического адреса в защищенном режиме

Каждая запись в таблице дескрипторов называется *дескриптором* и определяет сегмент в памяти.

Запись таблицы дескрипторов занимает 8 байт, а записанная в дескрипторе информация включает в себя базовый адрес, предельное значение и флаги полномочий доступа к сегменту.

Записи предельного значения сегмента и полномочий доступа в дескрипторе определяют размер и тип сегмента.

Сегменты могут иметь размер от 1 до 65 536 байт и могут быть сегментами кода или сегментами данных. Сегменты кода могут содержать выполняемые машинные инструкции и доступные только по чтению данные. Сегменты данных могут содержать данные, доступные по чтению и записи.

Записывать данные в сегменты кода или выполнять инструкции в сегментах данных невозможно. Любая попытка сделать это или попытка доступа к данным вне границ сегмента вызывает общий сбой по нарушению защиты (сокращенно сбой GP). Поэтому режим и называется защищенным.

По данному адресу в реальном режиме прикладная программа может определить физический адрес.

В защищенном режиме это обычно не так, поскольку селекторная часть логического адреса является индексом в таблице дескрипторов, и сам селектор не имеет прямого отношения к вычислению физического адреса. Это дает то преимущество, что управление виртуальной памятью можно реализовать, не влияя на прикладную программу. Например, путем простого обновления поля базового адреса дескриптора сегмента, операционная система может перемещать сегмент в физической памяти без влияния на использующую сегмент прикладную программу. Прикладная программа ссылается только на селектор сегмента, и на селектор не влияют изменения в дескрипторе.

Прикладная программа редко имеет дело с дескрипторами. При необходимости дескрипторы создаются и уничтожаются операционной системой и администратором памяти, а прикладная программа знает о соответствующих селекторах.

Селекторы аналогичны описателям файлов — с точки зрения прикладной программы это то, что обслуживается операционной системой, но в операционной системе они работают как индексы содержащих дополнительную информацию таблиц.

Особенности защищенного режима процессора i80286

В реальном режиме процессор i80286 является практически полным аналогом i8086, но имеет большее быстродействие. В реальный режим процессор переключается после аппаратного сброса или после включения питания компьютера.

В защищенном режиме процессор i80286 полностью преображается. Используя совершенно иной метод адресации памяти, процессор i80286 расширяет адресное пространство до 16 Мбайт.

Процессор i80286 в защищенном режиме имеет встроенную поддержку мультизадачных операционных систем, значительно ускоряющую и упрощающую процесс переключения задач. Эта поддержка активно используется всеми мультизадачными операционными системами и оболочками, разработанными для компьютера IBM PC/AT.

Кроме расширения адресного пространства, новый метод адресации памяти позволяет изолировать адресные пространства отдельных задач друг от друга. При этом прикладная программа, работающая в среде операционной системы, использующей защищенный режим, не может случайно или намеренно разрушить целостность самой операционной системы.

В защищенном режиме программа может записывать данные только в те области памяти, которые выделены ей операционной системой. Это сильно повышает надежность работы мультизадачных и, в частности, мультипользовательских операционных систем. В последнем случае изолирование адресных пространств задач, принадлежащих отдельным пользователям, в хорошо спроектированной мультипользовательской операционной системе полностью исключает такую ситуацию, когда после запуска одним пользователем недостаточно отлаженной программы приходится перезапускать всю систему.

В следующих моделях процессоров фирмы Intel — i80386 и i80486 — помимо расширения адресного пространства до умопомрачительной величины в 4 Гбайт реализована концепция страничной виртуальной памяти. Все это возможно только в защищенном режиме.

Не вдаваясь в детали, можно сказать, что механизм страничной виртуальной памяти позволяет разместить часть оперативной памяти на диске. Если ваш компьютер имеет диск большого размера, вы можете задействовать часть диска для создания *виртуальной оперативной памяти*. При этом размер оперативной памяти (виртуальной), предоставляемый программам, ограничен лишь размером свободного пространства на диске.

Кроме того, только механизм страничной виртуальной памяти может обеспечить прикладные программы относительно быстрой оперативной памятью, размер которой больше размера физической памяти, установленной в компьютере.

Помимо страничной виртуальной памяти в процессорах i80386 и i80486 реализован так называемый режим виртуального процессора i8086 или просто *виртуальный режим*. Этот режим реализуется в рамках защищенного режима (процессор может переключиться в виртуальный режим только из защищенного режима). В виртуальном режиме процессор способен выполнять программы, составленные для процессора i8086, находясь в защищенном режиме и используя аппаратные средства защищенного режима: мультизадачность, изолирование адресных пространств отдельных задач друг от друга, страничная виртуальная память.

Перечислим кратко основные преимущества, которые получает программа, работающая в защищенном режиме процессора:

- возможность непосредственной адресации памяти за пределами первого мегабайта;

- для процессоров i80386 и i80486 реализован механизм страничной виртуальной памяти, позволяющий программам работать с памятью, размер которой может быть много больше физической оперативной памяти, установленной в компьютере;
- аппаратная поддержка мультизадачности позволяет создавать на основе процессоров, работающих в защищенном режиме, высокопроизводительные мультизадачные и мультипользовательские системы;
- эффективная работа нескольких программ, составленных для MS-DOS, основанная на использовании виртуального режима работы процессора.

Реализация защищенного режима в процессоре DP32

Для поддержки защищенного режима в DP32 необходимо расширить стандартный набор команд процессора. В результате были добавлены следующие команды и описания.

- Команда загрузки регистра глобальной таблицы дескрипторов.

  Формат команды `lgdtr`:

31 24	23 20	19 16	15 8	7 0
Op	xxxx	xxxx	xxxx	N

Смещение (displacement)

  `N` — число задач.

  `Смещение` (displacement) указывает на расположение в памяти глобальной таблицы дескрипторов.

  `Op = "0A"`

- Команда перехода в защищенный режим.

  Формат команды `protect`:

31 24	23 20	19 16	15 8	7 0
Op	xxxx	xxxx	xxxx	xxxx

  `Op = "0B"`

  Команда выполняет перевод процессора в защищенный режим со следующего такта.

- Описание формата глобальной таблицы дескрипторов.
- Блок глобальной таблицы дескрипторов:
  - 1 слово — адрес сегмента кода;
  - 2 слово — размер сегмента кода;

- 3 слово — адрес сегмента данных;
- 4 слово — размер сегмента данных.

Для каждой задачи отводится 4 слова в таблице дескрипторов.

Также для каждой задачи необходимо хранить значения счетчика команд (PC), регистра кода условия (CC) и регистров общего назначения.

При переключении задач, процессором автоматически будет производиться сохранение и восстановление PC , CC и набора регистров, конкретной задачи.

Дополнение поведенческой модели DP32

Для добавления новых команд внесем следующие изменения в файл `Package dp32_`:

```
CONSTANT op_mov     : bit_8 := X"08";
CONSTANT op_lgdtr   : bit_8 := X"0A";
CONSTANT op_protect : bit_8 := X"0B";
```

> ***Замечание***
>
> Команда с операцией `op_mov` — стандартная команда сдвига.

В файл dp32_behav.vhd добавляем процедуры загрузки и сохранения контекста. Эти процедуры используются при переключении между задачами. Каждая задача работает в своем контексте.

```
-------------------------------------------------------------
                PROCEDURE Load_Context(task :in integer) IS
        Begin
            CS:=reg(task*32);
            add(CS_end,bits_to_int(reg(task*32+1)),bits_to_int(CS),
                temp_V,temp_N,temp_Z);

            DS:=reg(task*32+2);
            add(DS_end,bits_to_int(reg(task*32+3)),bits_to_int(DS),
                temp_V,temp_N,temp_Z);

            cc_V:=reg(task*32+4)(0);
            cc_N:=reg(task*32+5)(0);
            cc_Z:=reg(task*32+6)(0);

            PC:=reg(task*32+7);

            reg(0 to 15):=reg((task*32+16) to (task*32+31));
```

```
        End Load_Context;
-------------------------------------------------------------
        PROCEDURE Save_Context(task :in integer) IS
        variable iter_task:integer;
        Begin

            if PC=CS_end then

                curr_task:=curr_task-1;
                num_task:=num_task-1;

                assert (num_task>0)
                report "All task was finished !" severity FAILURE;

                for iter_task in task to num_task loop
                    reg((iter_task*32) to
(iter_task*32+31)):=reg(((iter_task+1)*32) to ((iter_task+1)*32+31));
                end loop;

            else
                reg(task*32+7):=PC;

                reg(task*32+4)(0):=cc_V;
                reg(task*32+5)(0):=cc_N;
                reg(task*32+6)(0):=cc_Z;

                reg((task*32+16) to (task*32+31)):=reg(0 to 15);
            end if;
        End Save_Context;
-------------------------------------------------------------
```

Функциональность новых команд, добавленных в файл `Package dp32_`, в поведенческой модели dp32_behav.vhd описывается следующим образом:

```
            when op_lgdtr =>
                    num_task:=i8;
                    memory_read(PC,true,gdtr);
                    if reset /='1' then
                        add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                    end if;
```

```
            when op_protect =>
                    PC:=gdtr;
                    for curr_task in 1 to num_task loop
                        memory_read(PC,true,reg(curr_task*32));
                        add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                        memory_read(PC,true,reg(curr_task*32+1));
                        add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                        memory_read(PC,true,reg(curr_task*32+2));
                        add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);
                        memory_read(PC,true,reg(curr_task*32+3));
                        add(PC,bits_to_int(PC),1,temp_V,temp_N,temp_Z);

                        reg(curr_task*32+7):=reg(curr_task*32);
                    end loop;

                    cc_V:='0';
                    cc_N:='0';
                    cc_Z:='0';

                    reg(0 to 15):=(X"0000_0000",X"0000_0000",X"0000_0000",

X"0000_0000",X"0000_0000",X"0000_0000",

X"0000_0000",X"0000_0000",X"0000_0000",

X"0000_0000",X"0000_0000",X"0000_0000",
                                        X"0000_0000",X"0000_0000",X"0000_0000",
                    X"0000_0000");

                    protect_mode:=true;
                    curr_task:=1;
                    PC:=reg(curr_task*32);
```

В процессор (файл dp32_behav.vhd) добавляем код, управляющий переключением между задачами:

```
    if (protect_mode) then
        Save_Context(curr_task);

        if (curr_task=num_task) then
            curr_task:=1;
```

```
        else
            curr_task:=curr_task+1;
        end if;
    end if;
```

В процессор (файл dp32_behav.vhd) добавляем переменные, ответственные за хранение физических адресов начала и конца сегментов кода и данных, а также за число задач (номер текущей задачи и индикатор защищенного режима — `protected_mode`). Здесь если `protected_mode` равняется 1, то процессор в данный момент времени находится в защищенном режиме, иначе процессор находится в реальном режиме.

```
        variable CS :bit_32:=X"0000_0000";
        variable DS :bit_32:=X"0000_0000";
        variable CS_end :bit_32:= X"0000_FFFF";
        variable DS_end: bit_32:= X"0000_FFFF";
        variable gdtr :bit_32:=X"0000_0000";
        variable num_task: integer;
        variable curr_task: integer;
        variable protect_mode: boolean:=false;
```

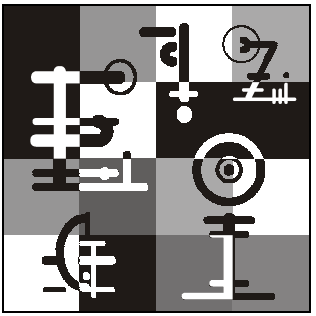
Лабораторная работа. Усовершенствование (развитие) VHDL-модели процессора DP32. Ко-симуляция. Тестирование

Цель работы

1. Ознакомление с проектом конвейера команд и проектом, реализующим защищенный режим, современных процессоров.
2. Понимание работы и реализации проектов конвейера команд и защищенного режима в поведенческой и структурной VHDL-моделях процессора DP32.
3. Наблюдение работы проектов конвейера команд и защищенного режима на примере моделей DP32 в среде FPGA Advantage 7.0 (рассмотрена в *главах 28—30*).

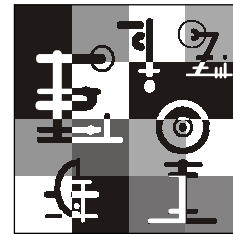
Программа работы

1. Представьте поведенческую модель DP32 с проектом, реализующим защищенный режим, в среде FPGA Advantage 7.0.
2. Продемонстрируйте преимущества работы поведенческой модели DP32 с проектом, реализующим защищенный режим, по сравнению с моделью без него.
   - Напишите программу в кодах DP32 и ко-симулируйте ее с проектом и без в среде FPGA Advantage 7.0.
   - Если нужно, оттестируйте модель DP32 с проектом и без.
3. Представьте структурную модель DP32 с проектом, реализующим защищенный режим, в среде FPGA Advantage 7.0.
4. Продемонстрируйте преимущества работы структурной модели DP32 с проектом, реализующим защищенный режим, по сравнению с моделью без него.
   - Напишите программу в кодах DP32 и ко-симулируйте ее с проектом и без в среде FPGA Advantage 7.0.
   - Если нужно, оттестируйте модель DP32 с проектом и без.
5. Представьте поведенческую модель DP32 с проектом конвейера в среде FPGA Advantage 7.0.
6. Продемонстрируйте преимущества работы поведенческой модели DP32 с конвейером по сравнению с моделью без него.
   - Напишите программу в кодах DP32 и ко-симулируйте ее с проектом и без в среде FPGA Advantage 7.0.
   - Если нужно, оттестируйте модель DP32 с проектом и без.
7. Представьте структурную модель DP32 с проектом конвейера в среде FPGA Advantage 7.0.
8. Продемонстрируйте преимущества работы структурной модели DP32 с конвейером по сравнению с моделью без него.
   - Напишите программу в кодах DP32 и ко-симулируйте ее с проектом и без в среде FPGA Advantage 7.0.
   - Если нужно, оттестируйте модель DP32 с проектом и без.



Часть V

Язык проектирования Verilog HDL. Примеры, инструментарий



Глава 25

Язык проектирования Verilog HDL

Язык Verilog HDL, аналогично языку VHDL, включает структурное и функциональное (поведенческое) описания устройства.

Структурное описание определяет точный физический характер устройства, его структурные компоненты, описание связи между ними.

Функциональное описание определяет поведенческую суть проектируемого устройства.

Такое разделение в языке не является явным. Оно задается по ходу разработки и определяется набором вызываемых модулей (структурное) и набором функций и задач (функциональное).

В дальнейшем, при разборе Verilog-проектов предполагается более подробное изучение языка Verilog HDL.

25.1. Структурное описание

Основные конструкции для задания структурного описания:

- модули (Modules);
- макромодули (Macromodules);
- объявление портов (Port Definitions);
- структура модулей (Module Statements and Constructs).

25.1.1. Модули (Modules)

Основным элементом языка Verilog являются модули. В языке VHDL им соответствуют компоненты.

Модуль содержит:

- имя модуля;

- описание входных и выходных портов;
- функциональную или исполнительную часть;
- поименованные вызовы — подстановки модулей;
- описание модуля заключается в ключевые слова `module` и `endmodule`.

Функциональная и исполнительная части содержат описание поведения компонента-модуля, сигналов внутри него.

Поименованные вызовы определяют иерархию компонентов, из которых состоит проектируемый элемент, то есть определяет его структуру.

Далее показан пример задания модуля `NAND` с двумя входами `a`, `b` и выходом `z`:

```
module NAND(a,b,z);
input a,b;
output z;
wire and_out;
AND instance1(a,b,and_out);
INV instance2(and_out, z);
endmodule
```

После определения портов вызываются два модуля `AND` и `INV` с соответствующими параметрами. При этом вызову модуля присваивается определенное имя. В данном примере это `instance1` и `instance2`.

Предполагается, что вызываемые модули являются библиотечными или описаны разработчиком выше.

В ходе проектирования у разработчика образуется иерархия модулей, представляющих структуру устройства. Каждый модуль можно отдельно скомпилировать, если для него определены вызываемые модули (которые стоят ниже в иерархии), а вышестоящие не затрагиваются.

25.1.2. Макромодули (Macromodules)

Макромодули имеют тоже назначение, что и сами модули. Но они позволяют повысить эффективность работы симулятора за счет явного образования иерархии компонентов разрабатываемого устройства. Структура макромодуля, в общем случае, не отличается от структуры модуля.

Использование макромодуля направлено на более удобную и наглядную разработку элемента, а также на задание глобальных процессов.

Далее показан пример макромодуля:

```
macromodule adder (in1,in2,out1);
input [3:0] in1,in2;
```

```
output [4:0] out1;
assign out1 = in1 + in2;
endmodule
```

Описание операций языка Verilog HDL и примеры их применения представлены в *приложении 2*.

25.1.3. Объявление портов (Port Definition)

Список портов задается в виде входов и выходов после имени модуля при объявлении модуля.

```
module name ( port_list )
```

В качестве имени порта могут выступать:

- ❒ идентификатор;
- ❒ один бит, выделенный из какого-нибудь вектора;
- ❒ группа бит, выделенных из вектора;
- ❒ объединение с помощью конкатенации нескольких бит или векторов.

В скобках перечисляются только порты устройства.

Описание портов

Направление порта определяется сразу после его объявления внутри модуля в разделе объявлений переменных модуля.

Выходной порт определяется с помощью ключевого слова `output`, входной — `input`, двунаправленный — `inout`. Использование порта возможно только после его описания. Порт можно задать в виде вектора или одного бита.

Примеры:

```
inout a;
inout [2:0]b;
output a;
output [2:0]b;

module ex1( a, b, z );
input a, b;
output z;
endmodule

module ex2( a[1], a[0], z );
input [1:0] a;
```

```
output z;
endmodule

module ex3( {a,b}, z );
input a,b;
output z;
endmodule
```

25.1.4. Структура модуля

В общем случае модуль может содержать следующие инструкции:

- задание переменных модуля ключевым словом `parameter`;
- задание переменных модуля типами переменных: `wire`, `wand`, `wor`, `supply0`, `supply1`, `tri`;
- задание входов;
- задание выходов;
- задание двунаправленных портов;
- задание регистровых переменных;
- подстановки модулей (вызовы модулей);
- вызовы вентилей (вентильное проектирование);
- определение функций;
- определение задач (`task`);
- определение блока `always` (оператор `always` аналогичен оператору `process` в языке VHDL, см. описание оператора `always`).

Задание переменных модуля ключевым словом *parameter*

На примерах показано, как можно задать конкретное значение параметру с помощью ключевого слова `parameter`:

```
parameter TRUE=1, FALSE=0;
parameter [1:0] S0=3, S1=1, S2=0, S3=2;
```

Параметр находится в области видимости того модуля, в котором он объявлен, и изменять его значение нельзя. Он позволяет использовать не значение параметра при разработке, а его символическое имя. В качестве параметра можно задать переменную любого типа: целого, действительного, логического.

Задать параметр можно в любом месте объявления переменных модуля, но до момента использования параметра. Поэтому рекомендуется определять параметр сразу при описании самого модуля.

Задание переменных модуля типами переменных: *wire*, *wand*, *wor*, *supply0*, *supply1*, *tri*

Рассмотрим отдельно задание переменных каждым из перечисленных типов.

Тип переменных *wire*

Переменные данного типа используются для задания соединений между портами модулей.

Аналогом подобной переменной является провод. В VHDL аналогом переменной этого типа является сигнал.

Переменная данного типа не хранит значения в отличие от регистровой переменной.

Использование переменной `wire` необходимо в случаях:

- когда требуется соединить внутреннюю переменную модуля с его выходным портом;
- когда требуется задать значение в операторе непрерывного назначения.

Примеры объявлений:

```
wire a;
wire [2:0] b;
```

В объявление переменной можно включать временную задержку.

Тип переменных *wand*

Данный тип представляет собой разновидность типа `wire` — `wire and`. Использование переменных этого типа поясняется на примере:

```
module wand_test(a, b, c);
input a, b;
output c;
wand c;
assign c = a;
assign c = b;
endmodule
```

Здесь переменной `c` типа `wand` (`wire`) присваивается результат логической операции "И" между переменными `a` и `b`. Использование подобной переменной можно заменить искусственным путем, выполнив сначала логическую операцию, а затем назначив результат.

В объявление переменной можно включать временную задержку.

Тип переменных *wor*

Тип подобен `wand`, но сначала совершается логическая операция "ИЛИ", а затем идет назначение.

```
module wor_test(a, b, c);
input a, b;
output c;
wor c;
assign c = a;
assign c = b;
endmodule
```

В объявление переменной можно включать временную задержку.

Тип переменных *tri*

`tri` (three-state) — разновидность типа `wire`. Только одна из переменных, которая управляет переменной типа `tri`, может иметь значение, отличное от `Z` (high impedance).

```
module tri_test (out, condition);
  input [1:0] conditon;
  output out;
  reg a, b, c;
  tri out;
 always @ ( condition ) begin
  a = 1'bz;
  b = 1'bz;
  c = 1'bz;
 case ( condition )
  2'b00 : a = 1'b1;
  2'b01 : b = 1'b0;
  2'b10 : c = 1'b1;
 endcase
 end
 assign out=a;
 assign out = b;
 assign out = c;
endmodule
```

Тип переменных *supply0*, *supply1*

Переменные данного типа используются подобно объявлению констант для задания значений "земля", "питание". Их можно применять для сброса и установки триггеров и регистров.

```
supply0 gnd;
supply1 power;
```

Оператор непрерывного назначения *assign*

Оператор используется в случаях, когда необходимо назначить (присвоить) значение, полученное внутри модуля, какой-то из переменных типа `wire`, `wand`, `wor`, `tri`.

```
wire a;
assign a = b & c;
wire a = b & c;
```

Задание регистровых переменных

Этот тип переменных используется внутри модуля для временного хранения значения переменной, то есть в качестве буфера.

```
reg x;
reg a,b,c;
reg [7:0] q;
```

Подстановки модулей (вызовы модулей)

Вызываемый модуль выполняет какую-то законченную функцию, которую использует внешний модуль. Вызывать один и тот же модуль можно несколько раз, но указывая при этом разные имена вызовов.

Фактически, таким образом можно вырезать из структуры любого устройства (внешнего модуля) некоторую часть и определить ее в другом модуле.

Пример подстановки или вызова модуля `SEQ` в модуль `top`:

```
module SEQ(BUS0,BUS1,OUT);
input BUS0, BUS1;
output OUT;
...
endmodule

module top( D0, D1, D2, D3, OUT0, OUT1 );
input D0, D1, D2, D3;
output OUT0, OUT1;
```

```
SEQ SEQ_1(D0,D1,OUT0),
SEQ_2(.OUT(OUT1),.BUS1(D3),.BUS0(D2));
endmodule
```

В первой подстановке `SEQ SEQ_1: D0` соединяется с `BUS0`, `D1` с `BUS1`, `OUT0` с `OUT`.

Во второй подстановке `SEQ_2`: `OUT` с `OUT1`, `BUS1` с `D3`, `BUS0` с `D2`.

Представленные подстановки являются позиционной и поименной соответственно.

При поименной подстановке явно определяется, какой порт модуля `top` подключается к каждому порту в списке модуля `SEQ`. Необозначенные порты в модуле не связываются.

При позиционной подстановке указываются порты и определяются связи с модулем согласно позициям в списке.

Параметризованные проекты

Verilog позволяет создавать параметризованные проекты, не учитывая при этом значений объявленного внутри модуля параметра во время подстановки модуля.

```
module foo (a,b,c);
parameter width = 8;
input [width-1:0] a,b;
output [width-1:0] c;
assign c = a & b;
endmodule
```

В данном примере задается параметр `width=8`, но его можно переопределить во время вызова (подстановки) данного модуля. Такой модуль выступает как шаблон.

В следующем примере вызываем ранее описанный модуль `foo`, задавая ему новое значение параметра, равное 4:

```
module param (a,b,c);
input [3:0] a,b;
output [3:0] c;
foo #(4) U1(a,b,c);
endmodule
```

Вызываем модуль `foo` с параметром 4 и именем `U1`. Проверку параметров, их разрядность и тип компилятор делает автоматически при вызове (подстановке) шаблона. Это позволяет избежать неоднозначности в случаях, когда задано несколько параметров. (В данном случае объявили разрядность портов 4 и параметр при вызове определили как 4.)

Вызовы вентилей (вентильное проектирование)

Данный раздел предоставляет информацию о простейших модулях-вентилях:

```
and; nand; or; nor; xor; xnor; buf; not; tran
```

Назначение каждого модуля понятно по его названию.

Примеры:

```
buf (buf_out,e);
and and4(and_out,a,b,c,d);
```

Обычно первый параметр является выходным, а оставшиеся параметры — входными.

25.2. Функциональное описание

Функциональное описание включает в себя:

- применение последовательных операторов (`if-else`, `while`, `for` и т. д.);
- объявление функций;
- описание работы функций;
- оператор задач `task`;
- оператор (блок) `always`.

25.2.1. Последовательные операторы

Применение последовательных операторов происходит при реализации комбинационной логики и алгоритма работы модуля.

```
x = b;
if (y)
x = x + a;
```

Иногда при реализации комбинационной логики применение простейших арифметических операций невозможно, например оператор "+". В этом случае приходится определять функцию суммирования или сумматор.

25.2.2. Объявление функций

Использование функций в языке Verilog позволяет реализовать комбинационную логику. Функции можно вызывать из структурной части модуля, используя оператор непрерывного назначения, из других функций и внутри блока `always`.

Объявление функций подразумевает задание следующих компонентов:

- объявление входов функций;
- объявление регистров, используемых внутри функции;

- объявление памяти;
- задание параметров;
- задание целых переменных.

Описание функции происходит по форме:

```
function [ range] name_of_function ;
    [ func_declaration]*
      statement_or_null
endfunction
```

После ключевого слова `function` указывается разрядность имени функции. Имени функции присваивается выходное значение. Далее указывается имя функции.

Внутри функции объявляются входные переменные, затем переменные целого типа.

После этого описывается тело функции. Описание заканчивается ключевым словом `endfunction`.

Пример:

```
function [7:0] scramble;
input [7:0] a;
input [2:0] control;
integer i;
begin
for (i = 0; i <= 7; i = i + 1)
scramble[i] = a[ i ^ control ];
end
endfunction
```

Объявление входов

Входы объявляются сразу после заголовка функции с помощью ключевого слова `input`. При этом же указывается их разрядность.

```
input [ range] list_of_variables ;
```

Выход функции

Выходное значение присваивается имени функции, поэтому функция возвращает только одно значение. Но оно может быть многоразрядным. Если требуется возвратить несколько значений, то можно воспользоваться операцией конкатенации нескольких значений в выходной вектор. А после вызова функции разбить на части полученный вектор.

```
function [9:0] signed_add;
input [7:0] a, b;
reg [7:0] sum;
```

```
reg carry, overflow;
begin
...
signed_add = {carry, overflow, sum};
end
endfunction
...
assign {C, V, result_bus} = signed_add(busA, busB);
```

Объявление регистров

Объявление регистров внутри функции осуществляется с помощью слова `reg`:

```
reg [ range] list_of_register_variables ;
```

Переменную можно задать как вектор или как бит. Она предназначена для временного хранения значения переменной внутри функции.

Примеры:

```
reg x;
reg a, b, c;
reg [7:0] q;
```

Объявление памяти

Verilog HDL позволяет моделировать память как набор регистров. Для этого можно воспользоваться оператором регистров:

```
reg [7:0] byte_reg;           // Объявили вектор регистров
reg [7:0] mem_block [255:0];  // Задали массив из 256 8-разрядных
                              // регистров
```

Доступ к регистру осуществляется по индексу полученного массива `mem_block[]`. Обратиться к какому-нибудь биту из выбранного 8-разрядного регистра невозможно. Если возникает в этом необходимость, следует скопировать выбранный регистр во временную переменную, а затем можно "вытащить" из него бит, как показано ниже:

```
byte_reg = mem_block [7];
individual_bit = byte_reg [3];
```

Для записи бита в память можно воспользоваться заданием маски и соответствующей маской.

Задание параметров

Параметры задаются, как и в модулях, с помощью ключевого слова `parameter`.

```
function gte;
parameter width = 8;
```

```
input [width-1:0] a,b;
gte = (a >= b);
endfunction
```

Объявление целых переменных

Задать целую переменную можно с помощью ключевого слова `integer`. Если разрядность при этом не указывается, то по умолчанию это может быть переменная с разрядностью 32.

Переменная может быть задана локально — внутри функции, или глобально — в модуле, вызывающем эту функцию.

```
integer a;
integer b, c;
```

25.2.3. Описание функций и операторы языка Verilog HDL

Функция используется для реализации определенной логики, вычисления каких-то регистровых переменных в модуле по не прямому алгоритму.

Описание функций сводится к использованию следующих конструкций:

- процедурных назначений;
- RTL-назначений;
- блочного оператора `begin … end`;
- операторов условия `if … else`, `case …`, `casex …`, `casez`;
- операторов цикла `for`, `while`, `forever`;
- оператора прерывания `disable`.

Процедурные назначения

Их использование подразумевает применение операций (перечисленных в табл. 1 *приложения 2*), результат которых требуется присвоить какой-либо переменной. Это сдвиги, конкатенация, арифметические выражения и т. д.

Примеры:

```
sum = a + b;
control[5] = (instruction == 8'h2e);
{carry_in, a[7:0]} = 9'h 120;
```

RTL-назначения

Применение данного оператора в основном связано с будущим программированием проектируемого устройства на PLD или FPGA.

Его назначение поясним на примерах.

```
rega = #5 arg1 + arg2;
```

Здесь вычисляется сумма, но только после 5-ти временных отрезков. Сразу после вычисления сумма присваивается переменной `rega`. Понятие "временной отрезок" является относительным. Его величина задается симулятором в опциях или с помощью оператора `timescale 100ps/10ps`. Задается временная шкала длиной 100 с шагом 10.

В следующем примере вычисление суммы производится сразу, а присвоение переменной в левой части — после заданного временного интервала:

```
rega <= #5 arg1 + arg2;
```

> ***Примечание***
>
> В некоторых симуляторах данная операция не поддерживается и игнорируется.

Блочный оператор *begin ... end*

Данный оператор используется для объединения нескольких инструкций, чтобы применить для них оператор цикла или условия.

```
begin : block_name
reg local_variable_1;
integer local_variable_2;
parameter local_variable_3;
... statements ...
end
```

Из формы видно, что данный блок можно поименовать, указав имя после `begin :`, но это необязательно. Поименованный блок используется в случае применения оператора `disable`.

Операторы условия

if... else

Использование оператора условия:

```
if ( e xpr )
begin
... statements ...
end
else
begin
... statements ...
end
```

Если условие истинно, выполняется первый блок после `if`. В противном случае выполняется блок после `else`. Применение конструкции `else` не обязательно, и ее можно опустить.

Использование данного оператора направлено на реализацию логики мультиплексора:

```
if (c)
    x = a;
else
    x = b;
```

Следующий пример показывает возможность применения вложенных конструкций `if ... else`:

```
if (instruction == ADD)
begin
  carry_in = 0;
  complement_arg = 0;
end
else if (instruction == SUB)
  begin
   carry_in = 1;
   complement_arg = 1;
  end
else
  illegal_instruction = 1;

if (select[1])
 begin
  if (select[0])
     out = in[3];
  else
     out = in[2];
 end
else
 begin
  if (select[0])
      out = in[1];
  else
      out = in[0];
 end
```

Оператор условия можно применять для реализации защелки, когда выходной переменной присваивается или нет значение внутренней переменной (используемый оператор `always` будет рассмотрен позже, см. описание оператора `always`):

```
always begin
if ( c ) begin
value = x;
end
Y = value;
end
```

case, *casex*, *casez*

Данные операторы позволяют избежать многократного применения оператора `if... else`.

```
case ( expr )
case_item1 : begin
... statements ...
end
case_item2 : begin
... statements ...
end
default : begin
... statements ...
end
endcase
```

Содержание оператора определяется внутри ключевых слов `case… endcase`.

После слова `case` в скобках указывается выражение, которое проверяется на истинность. Ниже указываются возможные значения `case_item :`. После двоеточия идет структура, обрабатывающая данный случай.

Если условию не отвечает ни одно из перечисленных выражений, то обработка передается операторам после слова `default :`.

Пример:

```
case (state)
IDLE: begin
if (start)
next_state = STEP1;
else
next_state = IDLE;
end
```

```
STEP1: begin
/* do first state processing here */
next_state = STEP2;
end
STEP2: begin
/* do second state processing here */
next_state = IDLE;
end
endcase
```

Операторы `casex casez` и `?` аналогичны предыдущему оператору, но при проверке на истинность бит, вместо которого поставлен `x`, `z`, `?`, игнорируется.

Пример:

```
reg [3:0] cond;
casex (cond)
4'b100x: out = 1;
default: out = 0;
endcase
```

На выходе будет 1, если выражение примет значения `4'b1000` или `4'b1001`.

Например, следующие выражения:

```
if (cond[3]) out = 0;
else if (!cond[3] & cond[2] ) out = 1;
else if (!cond[3] & !cond[2] & cond[1] ) out = 2;
else if (!cond[3] & !cond[2] & !cond[1] & cond[0] ) out = 3;
else if (!cond[3] & !cond[2] & !cond[1] & !cond[0] ) out = 4;
```

можно заменить на:

```
casex (cond)
4'b1???: out = 0;
4'b01??: out = 1;
4'b001?: out = 2;
4'b0001: out = 3;
4'b0000: out = 4;
endcase
```

Использовать одновременно в одном выражении `x`, `z`, `?` нельзя. Не допускается также использование одного из значков внутри оператора условия в проверяемом выражении:

```
express = 1'bz;
...
```

```
casez (express)
...
endcase    // Недопустимое применение !
       Пример casez:
casez (what_is_it)
2'bz0: begin

it_is = even;
end
2'bz1: begin

it_is = odd;
end
endcase
```

Операторы цикла

for

Применяется, когда известно число итераций. При этом следует указать нижнюю и верхнюю границы и шаг. Форма записи следующая:

```
for (index = low_range;index < high_range;index= index + step)
for (index = high_range;index > low_range;index= index - step)
for (index = low_range;index <= high_range;index= index + step)
for (index = high_range;index >= low_range;index= index - step)
```

Пример:

```
for (i = 0; i <= 31; i = i + 1)
  begin
   s[i] = a[i] ^ b[i] ^ carry;
   carry = a[i] & b[i] | a[i] & carry | b[i] & carry;
  end

for (i = 6; i >= 0; i = i - 1)
  for (j = 0; j <= i; j = j + 1)
    if (value[j] > value[j+1])
       begin
         temp = value[j+1];
         value[j+1] = value[j];
         value[j] = temp;
      end
```

while

Применяется, когда число итераций заранее не известно. Выполнение цикла продолжается до выполнения определенного условия.

Форма записи:

```
While (condition)
Begin
  …statements…
end
```

Примеры:

```
always
while (x < y)
x = x + z;

always
begin @ (posedge clock)
while (x < y)
begin
@ (posedge clock);
x = x + z;
end
end;
```

Применение данного оператора внутри оператора `always` может повлечь за собой возникновение комбинационной обратной связи. Поэтому следует использовать при этом также оператор `@ (posedge clock)` или `@ (negedge clock)`.

```
always
begin @ (posedge clock)
while (x < y)
begin
@ (posedge clock);
x = x + z;
end
end;
```

disable

Оператор используется с целью прервать выполнение текущего блока и выйти из него.

```
begin : compare
for (i = 7; i >= 0; i = i - 1)
```

```
begin
 if (a[i] != b[i]) begin
    greater_than = a[i];
    less_than = ~a[i];
    equal_to = 0;
    disable compare;
   end
end
greater_than = 0;
less_than = 0;
equal_to = 1;
end
```

Когда доходим до оператора `disable`, прерывается выполнение поименованного блока `compare`, выполняется переход на следующую операцию после поименованного блока (`greater_than = 0`).

25.2.4. Использование оператора задачи *task*

Конструкция `task` является подобием функций, но только она в отличие от функций имеет выходные и двунаправленные порты.

Пример реализации `task` (сумматора) внутри модуля `task_example`:

```
module task_example (a,b,c);
input [7:0] a,b;
output [7:0] c;
reg [7:0] c;
// описание task
task adder;
input [7:0] a,b;
output [7:0] adder;
reg c;
integer i;
begin
c = 0;
for (i = 0; i <= 7; i = i+1) begin
adder[i] = a[i] ^ b[i] ^ c;
c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
end
```

```
end
endtask
//конец описания
always
adder (a,b,c); // c is a reg
endmodule
```

Задачи описываются внутри модуля с помощью ключевых слов `task … endtask`. По структуре они похожи на функции. Внутри задачи объявляются входы и выходы. Далее идет функциональное описание задачи.

Внутри модуля, где объявлен `task`, должен быть вызов задачи с подстановкой параметров. Поэтому порядок описания входов и выхода задачи имеет значение. Соответственно, следует соблюдать и порядок при вызове задачи.

Вызов задачи должен бать внутри оператора `always`.

25.2.5. Оператор *always*

Операции, записанные внутри данного оператора, выполняются при наступлении некоторого события, указанного в скобках после ключевого слова `always`.

С помощью этого оператора можно определить защелку, триггер или комбинационную логику.

Форма записи:

```
always @ ( event-expression [or event-expression*] )
begin
... statements ...
end
```

Выражение в скобках определяет момент, когда начинают выполняться операторы внутри `always`. Обычно это перепад какого-либо сигнала.

```
always @ ( a or b or c ) begin
f = a & b & c
end
```

В примере при изменении значения одного из сигналов пересчитывается значение функции `f`. С помощью логических операций можно определить момент наступления события для нескольких переменных.

Можно задать только спад сигнала или фронт сигнала как момент наступления события. Для этого следует применить ключевые слова `negedge` и `posedge` соответственно.

Пример:

```
always @ (posedge c or posedge p)
if (p)
   z = d;
else
   z = a;

always @ ( posedge CLOCK or posedge event1 or
negedge event2 ) begin
if ( event1 ) begin
... statements ...
end
else if ( ! event2 ) begin
... statements ...
end
else begin
... statements ...
end
end
```

Данный оператор можно использовать для задания синхронизации в модуле.

```
always @ ( posedge CLOCK or posedge event1 or
negedge event2 ) begin
if ( event1 ) begin
... statements ...
end
else if ( ! event2 ) begin
... statements ...
end
else begin
... statements ...
end
end
```

Внутри модуля может быть несколько операторов `always`. Внутри данного оператора невозможен вызов модуля, но можно вызвать функцию или задачу. В принципе, структура этого оператора определяет поведение модуля.

Аналогом оператора `always` в языке VHDL является оператор `process()`.

Лабораторная работа. Исследование Verilog HDL-проектов импульсного фильтра, параллельного регистра и АЛУ с помощью системы VeriLogger Pro / TestBencher Pro

Цель работы

Знакомство с языком Verilog HDL и с работой в системе VeriLogger Pro / TestBencher Pro на примере Verilog-проектов импульсного фильтра, параллельного регистра и АЛУ. Спецификация и Verilog-код этих проектов представлены в *главах 26* и *27*.

Описание работы с системой (пакетом) VeriLogger Pro / Testbencher Pro

Установка пакета

Для установки пакета необходимо запустить файл Vlogeval.exe. Далее указываем путь для установки в рабочий каталог. Установка завершена.

О пакете VeriLogger Pro / Testbencher Pro

Пакет VeriLogger Pro / Testbencher Pro предназначен для проектирования, разработки, симуляции и анализа функционирования цифровых устройств с использованием языка моделирования Verilog HDL.

Пакет содержит все необходимые инструменты для работы с ним:

- текстовый редактор используется для написания кода программы, описывающей функционирование устройства;
- компилятор, который транслирует полученный текст программы на языке Verilog HDL в поведенческую модель;
- симулятор, который демонстрирует полученную модель с помощью временных диаграмм функционирования.

Главное окно пакета содержит:

- Окно **Projects**, показывающее файлы текущего проекта. Оно содержит имя текущего проекта и путь, по которому он находится. Имя проекта образует верхний уровень иерархии (дерева) проекта. Развернув его, можно увидеть следующие элементы дерева — список модулей (они являются основными элементами языка Verilog HDL), а также для каждого модуля — список портов (входных и выходных) и сигналов, присутствующих внутри модуля;
- Окно-текстовый редактор. Оно открывается, если щелкнуть правой кнопкой мыши на имени файла проекта и выбрать пункт **Open "Имя файла"**;
- Окно **Diagram** используется для построения временных диаграмм функционирования и непосредственной симуляции устройства;
- Окно **Report** предназначено для комментариев разработчика, информационных сообщений и ошибок от компилятора и симулятора, содержит окно отладчика. Для разработки не используется. Окно содержит несколько вкладок:
  - **Verilog.log** и **Waveperl.log** — содержат информацию от компилятора и симулятора о процессе компиляции и симуляции;
  - **Breakpoints** — окно точек остановки, используется для отладки;
  - **Errors** — содержит сообщения об ошибках в программе;
  - **HDL** — описывает поведенческую модель, и ее содержимое формируется в результате компиляции программы.

По умолчанию модель сохраняется в файле untitledTim.v, и пользователь может сохранить его под своим именем. Остальные вкладки не используются.

Создание проекта

При запуске пакета автоматически происходит открытие нового проекта. Создать проект можно, нажав правую кнопку мыши в окне проекта и выбрав пункт **New Project**, или в меню выбрать **Project | New HDL Project**. Имя проекта принимается по умолчанию untitled. Файл проекта имеет расширение hpj.

Открытие проекта

Проект можно открыть, если нажать правую кнопку мыши и выбрать пункт **Open Project**, или в меню выбрать **Project | Open HDL Project**. После открытия в окне проекта появятся существующие в этом проекте файлы с расширениями v или vo.

Сохранение проекта

Проект можно сохранить аналогичным образом, выбрав пункт **Save HDL Project**. При этом имени файла проекта по умолчанию присваивается расширение hpj.

Добавление файлов в проект

Файлы можно добавить, если выбрать пункт **Add HDL files**. Все файлы должны иметь расширение v или vo. Добавленный файл будет показан в окне проекта.

Создание файлов в проекте

Файл можно создать, если выбрать в меню **Editor** пункт **New HDL file**. Созданный файл необходимо самостоятельно добавить в проект, предварительно его сохранив.

Сохранение файлов

При сохранении файла необходимо ввести имя файла и указать его расширение. По умолчанию расширение у файла будет отсутствовать.

Удаление файла из проекта

Файл можно удалить, если нажать правую кнопку мыши и выбрать пункт **Remove file**.

Работа в окне проекта

Если щелкнуть правой кнопкой мыши на имени файла, можно открыть этот файл в текстовом редакторе, откомпилировать или удалить из проекта, выбрав соответствующий пункт. После компиляции файла, если она успешна, в окне проекта появится список модулей, содержащихся в этом файле, а также — список входных и выходных портов устройства и список сигналов внутри него. Если открыть нужный пункт, можно их просмотреть.

Работа в текстовом редакторе

Здесь подразумевается непосредственное написание кода программы на языке Verilog HDL. Все зарезервированные слова и конструкции этого языка отмечаются синим цветом, а остальные печатаются черным шрифтом.

Комментарии можно сделать с помощью знаков `//` или `/* */`, соответственно, для комментирования одной строки или блока. Комментарии выделяются зеленым цветом. После написания файла его можно откомпилировать, нажав кнопку **RUN** (зеленая стрелка вправо) на панели инструментов, или выбрать команду **Run** в меню **Simulate**. Также можно воспользоваться клавишей <F5>. Если компиляция прошла неуспешно, то все ошибки будут показаны в окне **Report** на вкладке **Errors**. Если щелкнуть два раза на ошибке, компилятор укажет стрелкой слева строку, где находится ошибка.

Работа в редакторе временных диаграмм

Процесс симуляции демонстрируется в окне **Diagram**. Слева в столбце после компиляции автоматически показываются входные и выходные сигналы устройства

(главного модуля). Входные сигналы отмечаются черным цветом, а выходные — розовым. При необходимости можно ненужные для рассмотрения сигналы удалить. Если нужно добавить еще сигнал, надо нажать кнопку **Add Signal** и дважды щелкнуть по появившемуся сигналу. Откроется меню, где необходимо указать имя сигнала. Если нужно добавить многоразрядный порт, то следует воспользоваться кнопкой **Add Bus**. В открывшемся меню следует указать имя сигнала и его разрядность. Вид сигнала необходимо сделать **Virtual Bus**.

После создания сигналов, необходимо определить значения входных сигналов и характер их изменений.

Если имеем дело с одноразрядными портами, то достаточно использовать кнопки с надписями **HIGH** и **LOW**. Нажатие соответствующей кнопки позволяет нарисовать сигнал верхнего или нижнего уровней соответственно (логическая 1 и 0). Для обозначения длительностей сигналов используется временная шкала. Кнопка с обозначением **TRI** применяется в устройствах с тремя состояниями функционирования (1, 0 и выключено).

Если работаем с многоразрядными портами, то для формирования сигналов используется кнопка **VAL**. Таким образом создается виртуальная шина, которой можно задать значения в шестнадцатеричной системе, если выделить нужный кусок временного сигнала на диаграмме и дважды щелкнуть на нем мышью. В появившемся окне указываем значение и переходим на следующий временной интервал для указания его значения. После того как входные сигналы сформированы, можно приступать к симуляции.

Симуляция

Симуляция может происходить в двух режимах: автоматическом и пользовательском. Выбор режима осуществляется по нажатию кнопки **Debug Run/Auto Run**.

Автоматический режим означает, что симуляция происходит каждый раз сразу после изменения сигнала или его временной части независимо от пользователя.

Пользовательский режим означает, что симуляция происходит после формирования всех сигналов и активизируется пользователем.

Симуляция происходит по нажатию кнопки **RUN** (зеленая кнопка со стрелкой вправо). Результатом симуляции являются временные диаграммы для выходных портов.

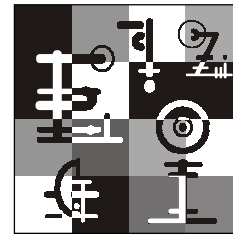
Можно менять временной масштаб, выбрав в меню **Options | Display Unit** — порядок (милли-, микро-, наносекунды) или использовав кнопки **Zoom Out** и **Zoom In**.

Нажатие кнопки с обозначением **TB** (желтого цвета) приведет к созданию новой временной диаграммы.

К недостаткам данного пакета можно отнести то, что у него отсутствуют библиотеки стандартных и простейших устройств, например, триггеров, счетчиков, регистров.

Программа работы

1. Открыть в текстовом редакторе системы VeriLogger Pro / TestBencher Pro Verilog-проект импульсного фильтра, рассмотренный в *главе 26*.
2. В редакторе временных диаграмм (вэформере) в соответствии со спецификацией задать на входы проекта необходимые сигналы.
3. Просимулировать Verilog-проект импульсного фильтра, провести его функциональное тестирование на соответствие спецификации.
4. Открыть в текстовом редакторе Verilog-проект параллельного регистра, рассмотренный в *главе 26*.
5. В редакторе временных диаграмм в соответствии со спецификацией задать на входы проекта необходимые сигналы.
6. Просимулировать Verilog-проект параллельного регистра, провести его функциональное тестирование на соответствие спецификации.
7. Открыть в текстовом редакторе Verilog-проект АЛУ, рассмотренный в *главе 27*.
8. В редакторе временных диаграмм в соответствии со спецификацией задать на входы проекта необходимые сигналы.
9. Просимулировать Verilog-проект АЛУ, провести его функциональное тестирование на соответствие спецификации.



Глава 26

Verilog HDL-проекты импульсного фильтра и параллельного регистра

С целью дальнейшего понимания принципов описания вычислительных устройств языком проектирования Verilog HDL, рассмотрим Verilog HDL-проекты импульсного фильтра и параллельного регистра или регистровой памяти.

26.1. Импульсный фильтр (спецификация проекта)

Импульсный фильтр обеспечивает формирование постоянного выходного сигнала низкого уровня, который фильтруется (остается неизменным) в случае прихода помехи в виде сигнала высокого уровня и небольшой длительности.

Обычно в цифровых устройствах сигнал низкого уровня может быть искажен помехой и может быть воспринят как сигнал высокого уровня. Чтобы исключить такой кратковременный импульс, используем простейший *импульсный фильтр*. Основными частями устройства являются RS-триггеры, собранные на элементах "И-НЕ", и элемент "И" (рис. 26.1).

Входные порты:

- X — фильтруемый сигнал;
- A — дополнительный импульс;
- B — дополнительный импульс.

Выходные порты:

- Y — выходной сигнал.

Описание функционирования импульсного фильтра: сигнал X длительности $T1$, подлежащий фильтрации, и импульсы помех длительностью $DT1$ поступают на вход фильтра, они одновременно подаются на входы S четырех триггеров RS-типа.

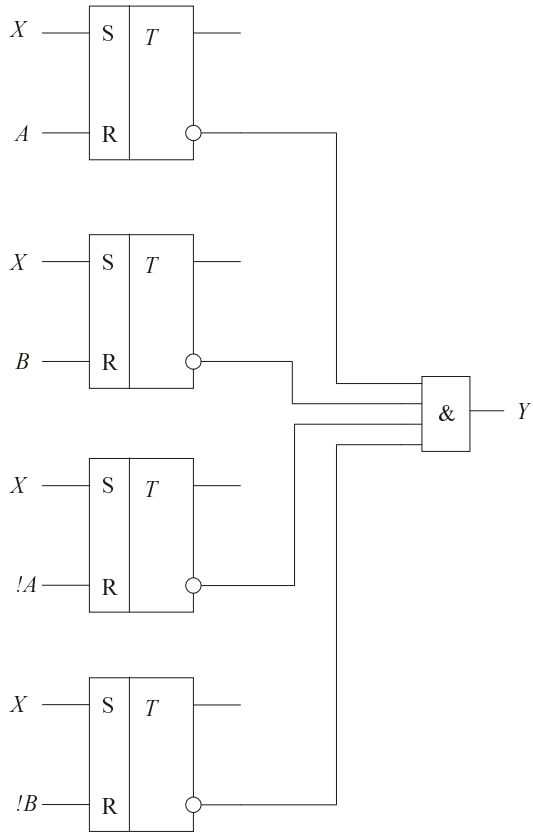


Рис. 26.1. Функциональная схема импульсного фильтра

На входы R-триггеров фильтра поступают дополнительные импульсные сигналы A, B, $!A$ (отрицание A), $!B$ (отрицание B) с длительностями импульсов $T2$. Длительности импульсов дополнительного сигнала $T2$ и помехи $DT1$ связаны соотношением $DT1$ меньше или равно $T2$.

Каждый RS-триггер начинает работать при подаче на его входы входного фильтруемого сигнала X и дополнительных сигналов A и B. Далее с инверсного выхода каждого триггера сигналы объединяются на входе элемента "И".

Помеха в зависимости от ее временного положения подавляется на одном из четырех триггеров.

На выходе Y формируется сигнал без помехи.

Для нормальной работы фильтра необходимо сигналы A и B на входы R-триггеров передавать в противофазе, а для того чтобы помеха не переключила фильтруемый

сигнал в момент переключения дополнительных сигналов на противоположные, их рекомендуется немного сдвинуть во времени относительно друг друга.

Текст Verilog HDL-проекта импульсного фильтра

```
// объявление модуля RS-триггера и его параметров:
module RStrigger (out,x,xdop);

// определение параметров
  // входных и выходных: x — S; xdop — R; out - !Q
   input x,xdop;
   output out;

  // регистровых
   reg res;

// module RStrigger — описание и RS-триггера на И-НЕ,
// и RS-триггера на ИЛИ-НЕ.
// Предполагается, что до начала работы на все RS-триггеры
// поступает комбинация хранения.
// Состояние триггера меняется, если
// изменится значение переменных x или xdop.
   always @(xdop or x)
      begin

// если изменить на противоположный сигнал на входе R,
// то в триггер пишем 0:
        if (~xdop)
           res = 0;
        else
// а если изменить на противоположный сигнал на входе S,
// то в триггер пишем 1:
          if (~x)
           res = 1;
      end
// инвертируем и выдаем значение триггера на выход:
   assign out = !res;
endmodule
// конец модуля
```

```
// объявление главного модуля импульсного фильтра и его параметров:
        module Filter (OutResult, X, A, B);

// определение параметров:
  // входных и выходных

   input   X, A, B;       //фильтруемый сигнал, дополнительные сигналы A и B
   output  OutResult;     //выходной порт — результат —
                          //отфильтрованный сигнал

// вызываем стандартные модули, выполняющие логические операции
// инвертирования с именами NOT1 и NOT2
// инвертируем переменные A и B и присваиваем значения
// переменным AInv и BInv соответственно
   not NOT1 (AInv, A);
   not NOT2 (BInv, B);

// вызываем 4 раза модуль RSTrigger, т.к. у нас в схеме 4 триггера,
// подставляя соответствующие параметры входных сигналов данного модуля
   RStrigger call1 (out1, X, A);
   RStrigger call2 (out2, X, B);
   RStrigger call3 (out3, X, AInv);
   RStrigger call4 (out4, X, BInv);

//вызов модуля, выполняющего логическую операцию И с именем AND1:
   and  AND1 (OutResult, out1, out2, out3, out4);
endmodule
// конец модуля
```

Программа содержит два модуля, которыми описывается устройство.

Первый модуль `RSTrigger` описывает поведение RS-триггера. Он имеет входные порты `X` и `Xdop` и выходной порт — `Out`. А также регистровую переменную `Res` для хранения промежуточных значений.

Второй модуль `Filter` является главным и имеет три входных порта `A`, `B`, `X` и один выходной `OutResult`. Этот модуль вызывает четыре раза первый модуль с передачей соответствующих параметров, а также осуществляет операции инвертирования и логического "И". Каждый вызов модуля `RSTrigger` имеет свое имя `call1`, `call2`, `call3`, `call4`.

26.2. Параллельный регистр (спецификация проекта)

Данное устройство представляет собой *восьмиразрядный параллельный регистр* и предназначено для работы в составе блоков обработки данных цифровых вычислительных устройств.

Устройство позволяет осуществлять: запись информации; хранение и регенерацию данных; установку в ноль всех разрядов регистра.

Регистр состоит из восьми триггеров D-типа с соответствующими схемами управления и восьми выходных буферов, имеющих на выходе состояние "Выключено" (рис. 26.2).

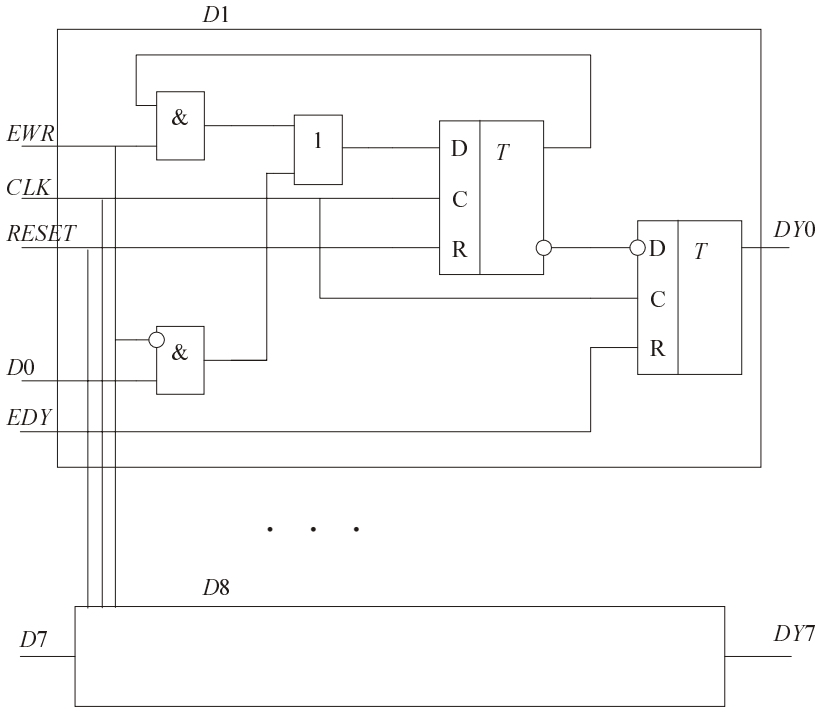


Рис. 26.2. Функциональная схема параллельного регистра

Описание функционирования параллельного регистра:

Входные порты устройства:

- *EWR* — управляющий вход для записи или хранения (регенерации);

- $RESET$ — входной сигнал для обнуления регистра;
- $CLOCK$ — синхроимпульс;
- $DATA[7:0]$ — восьмиразрядная шина данных;
- EDY — вход разрешения считывания информации;

Выходные порты:

- $OUTRESULT[7:0]$ — выходная шина данных.

Запись информации, поступающей на входы $DATA[7:0]$, осуществляется по спаду тактового сигнала $CLOCK$ в D-триггер при наличии сигнала низкого уровня на входе EWR — "Разрешение записи" и сигнала низкого уровня на входе $RESET$ — "Сброс". При этом состояние входа EDY не имеет значения.

Сброс регистра в состояние "0" производится подачей на вход $RESET$ сигнала высокого уровня независимо от состояния других входов устройства.

Хранение и регенерация информации осуществляется при наличии на входе EWR сигнала высокого уровня. D-триггеры сохраняют свое состояние.

Информация, записанная в D-триггеры, будет передаваться через выходные буферы на выводы $OUTRESULT[7:0]$ по положительному фронту сигнала $CLOCK$ и при наличии на входе EDY сигнала низкого уровня.

Перевод выводов $OUTRESULT[7:0]$ в состоянии "Выключено" не изменяет записанной информации и осуществляется подачей на вход EDY сигнала высокого уровня.

Текст Verilog HDL-проекта параллельного регистра

```
// модуль D-триггер
  module DTrigger (Result, Data, Clock, Reset, Ewr);
// определение параметров:
  // входных и выходных
   input  Data, Clock, Reset, Ewr; //входные данные, синхроимпульс,
                    // сброс, сигнал управления хранением-записью
   output Result; // выход триггера
// регистровых:
   reg Res, Buf; // регистровые переменные

// меняем состояние D—триггера, если меняется один из сигналов
   always @(negedge Clock or Reset or Ewr)
      begin
```

```
// если RESET==1, то устанавливаем в ноль регистр Res
   if (Reset)
      Res = 0;
   else
// иначе:
     begin
// если Ewr==0, запись данных в регистр со входа Data
// и в регистр Buf для временного хранения и перезаписи:
       if (!Ewr)
          begin
            Res = Data;
            Buf = Data;
          end
       else
// если Ewr==1, то храним и перезаписываем из регистра Buf
         Res = Buf;
     end
  end
// подаем на выход триггера содержимое регистровой переменной
  assign Result = !Res;
endmodule
// конец модуля

module MainRegister (OUTRESULT, EWR, CLOCK, RESET, DATA, EDY);
// объявление параметра:
  parameter numbits=7; // разрядность портов
// объявление входов и выходов:
// многоразрядных:
  input [numbits:0] DATA; // данные
// одноразрядных:
  input EWR, CLOCK, RESET, EDY; //хранение-запись, синхронизация, сброс,
                                //разрешение чтения с выхода
// выходных:
  output [numbits:0] OUTRESULT;
// регистровых:
  reg [numbits:0] res;
  wire[numbits:0] Res;

// вызов модулей Dtrigger:
```

```
  DTrigger Dtrig (Res[0], DATA[0], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[1], DATA[1], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[2], DATA[2], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[3], DATA[3], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[4], DATA[4], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[5], DATA[5], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[6], DATA[6], CLOCK, RESET, EWR);
  DTrigger Dtrig (Res[7], DATA[7], CLOCK, RESET, EWR);
// если меняется одна из переменных, то
  always @ (RESET or posedge CLOCK)
     begin
// если RESET==1, то сброс в 0 буферного регистра
  if (RESET)
     for (i=0; i<=7; i=i+1)
         res[i] = 0;
  else
// иначе -
    begin
//если нет сброса в 0 и разрешен выход, то
//читаем выход D-триггера (пишем в регистр)
      if (~EDY)
         res = ~Res;
// иначе — выход заперт:
      else res = 'bx;
    end
end
// назначаем результат на выход:
  assign OUTRESULT = res;
endmodule
```

Программа содержит два модуля `DTrigger` и `MainRegister`.

Модуль `Dtrigger` описывает функционирование D-триггера и схему управления хранением и записью.

D-триггер выполняет функцию хранения информации, поступающей на вход данных `DATA`, если на вход `RESET` подан логический нуль. Если подать логическую единицу, то в триггер запишется "0" (произойдет сброс). Запись в триггер осуществляется по спаду входного сигнала `CLOCK`.

Информация, которую необходимо записать в триггер, определяется схемой управления, состоящей из двух элементов "И" и элемента "ИЛИ".

Под хранением и регенерацией следует понимать перезапись содержимого триггера подачей хранимого разряда на вход триггера. Такую операцию можно сделать благодаря обратной связи, соединяющей выход триггера со входом схемы управления.

Таким образом, подавая соответствующий сигнал на вход `EWR`, можно выбирать записываемую информацию (новые данные на входе устройства) или данные из триггера.

Модуль `MainRegister` вызывает первый модуль, передавая в качестве параметров соответствующие входные данные и регистровую переменную для результата действия первого модуля.

Этот результат сохраняется в регистровой переменной главного модуля `Res[i]` и перезаписывается в `res[i]` в зависимости от сигнала `EDY`.

При наличии сигнала низкого уровня на входе `EDY` происходит считывание этой информации, которая передается на выход устройства.

Если на входе `EDY` сигнал высокого уровня, то этот триггер оказывается запертым, и его выход тоже.

Запись в этот регистр осуществляется по положительному фронту сигнала `CLOCK`.

Следует обратить внимание, что схема управления хранением/записью не реализована так, как показана на рис. 26.2 — в виде элементов "И" и "ИЛИ". В этом преимущество HDL-языков над схемотехническим проектированием. То есть часть логики можно реализовать, например, с помощью переменной, а не с помощью модуля. Хотя подобный вариант не исключен.

Лабораторная работа. Исследование Verilog-проектов импульсного фильтра, параллельного регистра и АЛУ с помощью системы QUARTUS II

Цель работы

Знакомство с языком Verilog HDL и работой в системе QUARTUS II на примере Verilog-проектов импульсного фильтра, параллельного регистра и АЛУ.

Описание работы с системой (пакетом) QUARTUS II

Пакет предназначен для проектирования цифровых устройств с помощью современных средств моделирования. QUARTUS позволяет выполнять симуляцию электронных устройств во времени, использовать для моделирования все возможные средства:

- графические схемы;
- VHDL;
- Verilog HDL;
- Tcl.

Установка пакета

Для установки пакета необходимо запустить файл Setup.exe. Далее указываем путь для установки в рабочий каталог. Установка завершена.

О пакете QUARTUS II

Средство разработки QUARTUS II — это следующий шаг в проектировании устройств с высокой степенью интеграции, включая разработку законченных систем на одном программируемом кристалле (System-On-a-Programmable-Chip (SOPC)).

Программное обеспечение QUARTUS II предоставляет полный цикл для создания высокопроизводительных систем на кристалле. QUARTUS II объединяет в себе проектирование, синтез, размещение элементов, трассировку соединений и верификацию, связь с системами проектирования других производителей.

Разработка систем на кристалле требует от разработчиков эффективной командной работы. Изменения в одной части проекта должно иметь минимальное влияние на других членов команды. Программное обеспечение QUARTUS II — это наиболее комплексная среда для разработки систем на кристалле SOPC, доступная в настоящее время. QUARTUS II включает в себя блочный метод разработки LogicLock.

LogicLock — это новая блочная методология проектирования, доступная исключительно в программном обеспечении QUARTUS II. QUARTUS II совместно с LogicLock — единственное программное обеспечение для разработки устройств на основе программируемой логики, которое включает в себя блочную методологию проектирования как стандартную функцию. Это помогает увеличить эффективность работы разработчиков, снизить время проектирования и верификации. LogicLock позволяет проектировать и проверять каждый модуль отдельно. Разработчики могут объе-

динять готовые модули в проект верхнего уровня, сохраняя производительность каждого модуля в процессе объединения. LogikLock снижает время разработки и верификации, т. к. каждый модуль оптимизируется только один раз.

Рассмотрим упрощенную методику работы с пакетом QUARTUS II.

Создание проекта

Для создания проекта используется встроенный **Project Wizard**, который запускается, если выбрать в меню **File** пункт **New Project Wizard** или **File | New**.

Откроется меню **New**.

На закладке **Project Files** надо выбрать пункт **Project File**, далее следует в появившемся окне указать в верхнем поле каталог, где будет находиться ваш проект (если он не существует, то QUARTUS его создаст); в среднем поле необходимо указать имя проекта. При этом в нижнем поле автоматически продублируется имя проектируемого устройства (а именно имя того компонента или модуля, который является основным (**Top-level**)).

Теперь QUARTUS создаст проект с заданным именем.

Слева располагается окно **Project Navigator**, которое отображает содержимое проекта.

На вкладке **Hierarchies** можно видеть значок вашего проекта.

На вкладке **Files** отображаются в папках **Design Files** и **Other Files** файлы, которые будут включены в проект.

Вкладка **Design Units** отображает все структуры (модули, функции), встречающиеся в вашем проекте. Они появляются после этапа компиляции.

После создания проекта необходимо создать или включить в проект рабочие файлы.

Добавление файлов в проект

Для добавления файлов в проект следует в меню **Project** выбрать пункт **Add Files to Project**. Появится меню, где на вкладке **Add Files** следует выбрать файл из текущей директории. Если в текущей директории имеются файлы, которые распознает QUARTUS, то он их отобразит в поле в виде списка (указав его тип). После нажатия **Add All**, все файлы будут добавлены в проект.

Также можно из списка удалить ненужные для добавления файлы.

Если у разработчика имеется библиотека своих элементов, то он может подключить ее аналогичным образом на вкладке **User Libraries**. Библиотека будет добавлена в проект. Добавленные в проект файлы будут отображены в **Project Navigator** на вкладке **Files**.

Создание новых файлов в проекте

Файл создается в меню **File** | **New**. В открывшемся окне на вкладке **Design Files** выберите файл того типа, который вам нужен: **Verilog HDL**, **VHDL** или **AHDL**.

В результате создастся файл для написания кода на соответствующем языке.

Файл **Block Diagram/Schematic File** будет создан для проектирования в графическом редакторе.

Открытие проекта

Проект открывается с помощью меню **File** | **Open Project**. При этом откроются все файлы из этого проекта.

Работа с проектом

Обычно проект разработчика состоит из нескольких файлов. QUARTUS позволяет составлять проекты из файлов комбинированных типов: на Verilog HDL, VHDL и с использованием графического редактора.

При этом у вас один из файлов будет главным, содержащим основной модуль устройства. Его необходимо определить как **Top-level**.

Для этого на имени файла в окне **Project Manager** в контекстном меню следует выбрать пункт **Add Current Entity at Top Level & Set Focus**. Если в данном файле описаны несколько модулей, то компилятор сам определит главный из них.

Компиляция проекта

Компиляция выполняется при выборе пункта меню **Start Compilation** в меню **Processing**. Можно также воспользоваться кнопками на панели инструментов.

Процесс компиляции состоит из нескольких шагов, которые отображаются в окне **Status**. После каждого шага компилятор выдает результаты, которые были получены на данном этапе. После окончания компиляции, если она прошла без ошибок, показываются ее результаты.

На вкладке **Design Units** появится список всех модулей проекта с указанием имени файла, где он описан (если файлов в проекте несколько).

На вкладке **Hierarchie** будет показана иерархия вызовов модулей с указанием имени вызова (подстановки).

Работа с вэформером и симуляция проекта

Процесс симуляции возможен при условии успешной компиляции, т. е. при отсутствии ошибок. Наличие предупреждений нежелательно, но с ними симуляция допустима.

Перед непосредственной работой следует создать файл временной диаграммы, где надо будет задать входные и выходные сигналы. Для этого в меню **File | New** на вкладке **Other Files** выберите тип файла **Vector Waveform File**. После чего появится окно для временной диаграммы и панель, с помощью которой можно будет задавать входные сигналы. Теперь в меню **Edit** выберите пункт **Insert Node or Bus**. В открывшемся окне нажмите кнопку **Node Finder**, в следующем окне из списка **Filter** выберите **Pins : All**. По нажатию кнопки **List** будут найдены входные и выходные порты устройства. Поместите необходимые порты в поле **Selected Nodes** и нажмите кнопку **Ok**. С помощью панели сигналов задайте входные импульсы для каждого порта. (Выделите необходимую часть сигнала мышью, и задайте 1 или 0, — импульс будет нарисован.)

После определения всех входных импульсов следует сохранить файл временной диаграммы. Файлу будет присвоено расширение vwf.

Теперь откройте окно опций симулятора в **Tools | Simulator Tool**. Здесь можно указать время окончания симуляции (**End Time**). Окончание симуляции можно определить временем или периодом, в течение которого заданы входные сигналы.

В графе **Simulation Input** задайте сформированный ранее файл временных диаграмм.

В поле **Simulation Mode** выберите режим симуляции **Functional**. Данный режим позволяет проверить функционирование устройства, не "привязывается" к выбранному на этапе компиляции, PLD. Нажмите кнопку **Generate Functional Simulation Netlist**. После этого можно запускать симуляцию **Processing | Start Simulation**.

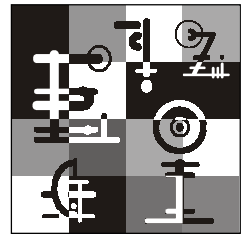
После окончания симуляции на временной диаграмме должны появиться результаты.

При необходимости можно сохранить полученную диаграмму.

Программа работы

1. Открыть в текстовом редакторе системы QUARTUS II Verilog-проект импульсного фильтра, представленного в этой главе.
2. В редакторе временных диаграмм (вэформере) в соответствии со спецификацией задать на входы проекта необходимые сигналы.
3. Просимулировать Verilog-проект импульсного фильтра, провести его функциональное тестирование на соответствие спецификации.
4. Открыть в текстовом редакторе Verilog-проект параллельного регистра, представленного в этой главе.
5. В редакторе временных диаграмм в соответствии со спецификацией задать на входы проекта необходимые сигналы.

6. Просимулировать Verilog-проект параллельного регистра, провести его функциональное тестирование на соответствие спецификации.
7. Открыть в текстовом редакторе Verilog-проект АЛУ, рассмотренный в *главе 27*.
8. В редакторе временных диаграмм в соответствии со спецификацией задать на входы проекта необходимые сигналы.
9. Просимулировать Verilog-проект АЛУ, провести его функциональное тестирование на соответствие спецификации.
10. Для следующей лабораторной работы необходимо написать Verilog-модель простейшего устройства ВТ (дешифратора, шифратора, компаратора, мультиплексора, счетчика и т. д.).



Глава 27

Verilog HDL-проект арифметико-логического устройства (спецификация проекта)

Арифметико-логическое устройство (АЛУ) предназначено для выполнения различных операций над числами.

Рассматриваемое АЛУ (рис. 27.1) выполняет над числами простейшие операции: сложение, вычитание, умножение, деление без остатка (нацело), поразрядные операции "И", "ИЛИ" и инвертирования. Это АЛУ позволяет передавать без изменения входную информацию на выход.

После операции на выходе устанавливаются соответствующие флаги: флаги переполнения, отрицательного результата и нулевого результата.

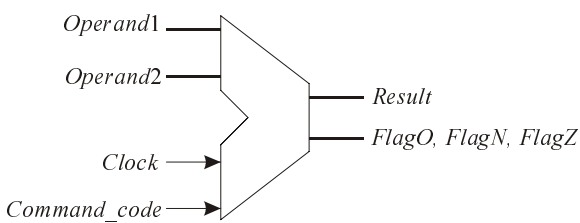


Рис. 27.1. Схема АЛУ

Описание функционирования АЛУ:

Входные порты устройства:

- $Operand1[3:0]$ — первый входной операнд;
- $Operand2[3:0]$ — второй входной операнд;
- $Command_code[2:0]$ — код выполняемой над операндами команды;
- $Clock$ — синхронизирующий импульс.

Выходные порты устройства:

- $Result[7:0]$ — результат операции над операндами;
- $FlagO$ — флаг переполнения;
- $FlagN$ — флаг отрицательного результата;
- $FlagZ$ — флаг нулевого результата.

На входы $Operand1$ и $Operand2$ АЛУ поступают два четырехразрядных операнда $XData$ и $YData$, над которыми выполняется арифметическая или логическая операция.

Каждый операнд задается четырьмя разрядами. Минимальное значение операнда — 0 (0000), а максимальное — 15 (1111). Если хотим работать с числами по значению большими, чем 15, необходимо увеличивать разрядность входных операндов до соответствующего уровня.

Сама операция задается входным сигналом $Command_code$, состоящим из трех разрядов. Подача определенной комбинации нулей и единиц на эти разряды соответствует определенной операции над операндами. Такое соответствие определено в табл. 27.1.

Таблица 27.1. *Коды команд АЛУ*

Код команды	Операция
000 (0)	Пропуск
001 (1)	Арифметическое сложение
010 (2)	Арифметическое вычитание
011 (3)	Арифметическое умножение
100 (4)	Деление (целая часть)
101 (5)	Логическое "И"
110 (6)	Логическое "ИЛИ"
111 (7)	Поразрядное инвертирование

Среди имеющихся операций можно выделить двухоперандные и однооперандные.

К первым относятся сложение, вычитание, умножение, деление, логические "И" и "ИЛИ". Ко вторым — пропуск и поразрядное инвертирование.

В однооперандных командах используется только первый операнд $Operand1$ независимо от второго $Operand2$.

Выходной сигнал $Result$ имеет разрядность в два раза больше, чем входные операнды. Это необходимо в случае умножения, так как данная операция может привести к удвоению разрядности результата и предотвращает переполнение. Соответственно, максимальное значение результата будет $255_{10} = 2^8 - 1$ или 11111111_2.

Каждый выходной результат сопровождается установкой соответствующих бит-флагов переполнения, нуля и отрицательного значения.

Флаг переполнения $FlagO$ устанавливается в состояние "1", если произошло переполнение результата. Эта ситуация возникает при делении на нуль, т. е. когда на вход второго операнда подать "0" и выбрать операцию деления. Результат будет неизвестен. Во всех остальных случаях флаг сбрасывается в нулевое состояние.

Флаг нуля $FlagZ$ устанавливается в "1", если полученный результат принимает нулевое значение. Такая ситуация возникает, когда происходит деление нуля на число, отличное от нуля, или вычитания двух одинаковых чисел, а также в результате побитовых операций. Если результат не нуль, то этот флаг устанавливается в "0".

Флаг отрицательного значения $FlagN$ устанавливается в "1", если результат операции отрицательный. При этом выходное значение $Result$ имеет положительное значение, равное по модулю полученному отрицательному. Нулевое значение этот флаг принимает, если значение $Result$ является положительным.

Сигнал $Clock$ используется для синхронизации. По его положительному фронту выдается на выход результат и флаги.

27.1. Текст Verilog HDL-проекта АЛУ

```
// Модуль - АЛУ.
Module ALU (command_code,xdata,ydata,clock,result,
flagZ,flagO,flagN);
parameter numbits=3;

   input    [2:0] command_code;       // код команды
   input    clock;                    // синхроимпульс
   input    [numbits:0] xdata, ydata; // входные данные
   output   [2*numbits+1:0] result;   // результат операции
   output   flagZ,flagO,flagN;        // флаги 0, переполнения
                                      // и отрицательного значения
   reg      [2*numbits+1:0] res;
   reg      z,o,n;
```

```
//*********************************************
// объявление
// Task - задание - выход==вход
task Disable;
// выходной результат присваивается имени задачи
  output [2*numbits+1:0] Disable;
// входы модуля - задаче Disable
  input  [numbits:0] x, y;
  begin
    y=x;     // второй операнд не нужен
    Disable=x; // выход=вход
  end
endtask
// конец задачи
//*********************************************

//*********************************************
// Task - задание - сумматор
task Summator;
  output [2*numbits+1:0] Summator; // выход
  input  [numbits:0] x, y; // операнды
  input  c_in; // бит переноса
// переменная целого типа:
  integer i;
  reg a,b,c,d,a1,b1,c1,bit,summa; // регистры для временного хранения

  begin
// цикл
  for (i=0; i<=numbits; i=i+1)
    begin
// побитовые операции И(&), ИЛИ(|), НЕ(~):
       a= ~x[i]&~y[i]&c_in;
       b= ~x[i]&y[i]&~c_in;
       c=c_in&x[i]&y[i];
       d=x[i]&~y[i]&~c_in;
// определяем значение имени задачи, т.е. находим сумму в разряде i:
       Summator[i]=a|b|c|d;
       a1=x[i]&y[i]&~c_in;
       b1=x[i]&~y[i]&c_in;
```

```
       c1=~x[i]&y[i]&c_in;
// находим бит-перенос
       bit=c|a1|b1|c1;
       c_in=bit;
    end
//записываем пренос в старший разряд:
    Summator[numbits+1]=bit;
// неиспользуемые при сложении старшие биты результата обнуляем в цикле:
    for (i=numbits+2; i<=2*numbits+1; i=i+1)
         Summator[i]=0;
   end
endtask
//*********************************************

//*********************************************
// Task - задание - вычитание
task Substance;
  output [2*numbits+1:0] Substance;
  input  [numbits:0] x, y;
  begin
// определяем модуль разности, сравнивая два числа
// и выполняя соответствующее вычитание:
    if (x>=y)
       Substance=x-y;
    else
       Substance=y-x;
  end
endtask
//*********************************************

//*********************************************
// Task - задание - умножение
task Multiple;
  output [2*numbits+1:0] Multiple;
  input  [numbits:0] x, y;
  begin
// если произведение положительно, то умножаем:
    if ((x>=0 && y>=0)||(x<=0 && y<=0))
       Multiple=x*y;
```

```
// иначе умножаем еще на –1:
    if ((x>0 && y<0)||(x<0 && y>0))
       Multiple=-x*y;
  end
endtask
//*********************************************

//*********************************************
// Task - задание - деление
task Divide;
  output [2*numbits+1:0] Divide;
  input  [numbits:0] x, y;
  begin
// если второй операнд – 0, то переполнение
    if (!y)
       Divide='bx;
// иначе
    else
      begin
// формируем положительный результат:
        if ((x>=0 && y>0)||(x<=0 && y<0))
           Divide=x/y;
        if ((x>0 && y<0)||(x<0 && y>0))
           Divide=-x/y;
      end
  end
endtask
//*********************************************

//*********************************************
// Task - задание - побитовое И
task Operation_and;
  output [2*numbits+1:0] Operation_and;
  input  [numbits:0] x, y;
  integer i;
  begin
// осуществляем поразрядное И
    for (i=0; i<=numbits; i=i+1)
        Operation_and[i]=x[i]&&y[i];
```

```
// обнуляем неиспользуемые старшие разряды результата:
    for (i=numbits+1; i<=2*numbits+1; i=i+1)
        Operation_and[i]=0;
  end
endtask
//*********************************************

//*********************************************
// Task - задание - побитовое ИЛИ
task Operation_or;
  output [2*numbits+1:0] Operation_or;
  input  [numbits:0] x, y;
  integer i;
  begin
// осуществляем поразрядное ИЛИ
    for (i=0; i<=numbits; i=i+1)
        Operation_or[i]=x[i]||y[i];
// обнуляем неиспользуемые старшие разряды результата:
    for (i=numbits+1; i<=2*numbits+1; i=i+1)
        Operation_or[i]=0;
  end
endtask
//*********************************************

//*********************************************
// Task - задание - побитовое инвертирование
task Operation_not;
  output [2*numbits+1:0] Operation_not;
  input  [numbits:0] x, y;
  integer i;
  begin
    y=x;
// инвертируем первый операнд и записываем в результат:
    for (i=0; i<=numbits; i=i+1)
        Operation_not[i]=~x[i];
// обнуляем неиспользуемые старшие разряды результата:
    for (i=numbits+1; i<=2*numbits+1; i=i+1)
        Operation_not[i]=0;
  end
```

```
endtask

//*****************************************

// вызываем задачи для выполнения операций
// пересчитываем каждый раз, когда Clock=1
   always @(posedge clock)
   begin
// выбираем тип операции по ее коду-номеру
   case (command_code)
         'b000:
            begin
              Disable(res,xdata,ydata);
// если результат 0, то z=1? Иначе 0
               z = (res==0) ? 1 : 0;
// если результат отрицательный, то 1, иначе 0:
               if (res<0)
                   n=1;
               else
                   n=0;
// переполнение = 0 (не может возникнуть)
               o=0;
            end
// Сложение
         'b001:
            begin
               Summator(res,xdata,ydata,0);
// если результат 0, то z=1? Иначе 0
               z = (res==0) ? 1 : 0;
// если результат отрицательный, то 1, иначе 0:
               if (res<0)
                   n=1;
               else
                   n=0;
               o=0;
            end
// Вычитание
         'b010:
            begin
```

```
// если результат 0, то z=1? Иначе 0
             Substance(res,xdata,ydata);
               z = (res==0) ? 1 : 0;
// если результат отрицательный, то 1, иначе 0:
               if (xdata<ydata)
                   n=1;
               else
                   n=0;
               o=0;
            end
// Умножение
         'b011:
            begin
// если результат 0, то z=1? Иначе 0
Multiple(res,xdata,ydata);
               z = (res==0) ? 1 : 0;
// если результат отрицательный, то 1, иначе 0:
               if (res<0)
                   n=1;
               else
                   n=0;
// если результат отрицательный, то 1, иначе 0:
               if ((xdata<0&&ydata>0)||(xdata>0&&ydata<0))
                  n=1;
               else
                  n=0;
               o=0;
            end
// Деление
         'b100:
            begin
// переполнение?
               o=(ydata==0) ? 1 : 0;
               Divide(res,xdata,ydata);
// если результат 0, то z=1? Иначе 0
             z = (res==0) ? 1 : 0;
// если результат отрицательный, то 1, иначе 0:
               if ((xdata<0&&ydata>0)||(xdata>0&&ydata<0))
                  n=1;
               else
```

```
                  n=0;
            end
// Логическое  И
         'b101:
            begin
               Operation_and(res,xdata,ydata);
// если результат 0, то z=1? Иначе 0
               z = (res==0) ? 1 : 0;
               n=0;
               o=0;
            end
// Логическое ИЛИ
         'b110:
            begin
// если результат 0, то z=1? Иначе 0
           Operation_or(res,xdata,ydata);
               z = (res==0) ? 1 : 0;
               n=0;
               o=0;
            end
// Побитное инвертирование, логическое НЕ
         'b111:
            begin
               Operation_not(res,xdata,ydata);
// если результат 0, то z=1? Иначе 0
               z = (res==0) ? 1 : 0;
               n=0;
               o=0;
              end
//Необрабатываемые коды команды
          default: res='bx;
    endcase
   end
// формируем выходные результаты:
   assign result=res;  //  Результат
   assign flagZ=z;     //  Флаг -  результат-ноль
   assign flagN=n;     //  Флаг -  результат-отрицательный
   assign flagO=o;     //  Флаг -  переполнение
endmodule
// конец модуля.
```

Программа имеет следующую структуру: главный модуль `ALU` вызывает задачи-обработчики операндов, т. е. соответствующие части программы, которые описывают соответствующую операцию. Задача вызывается главным модулем как функция с передачей соответствующих параметров главного модуля.

По результату вызова задачи главный модуль производит установку соответствующих флагов.

По положительному фронту сигнала `clock` результат и флаги подаются на выход устройства.

Лабораторная работа. Исследование с помощью систем VeriLogger Pro/TestBencher Pro и QUARTUS II Verilog-проекта, написанного по индивидуальному заданию

Цель работы

Знакомство с языком Verilog HDL и этапами проектирования цифровых устройств с помощью языка Verilog HDL: от получения спецификации (технического задания) на проектируемое устройство, до симуляции его работы и функционального тестирования этого устройства.

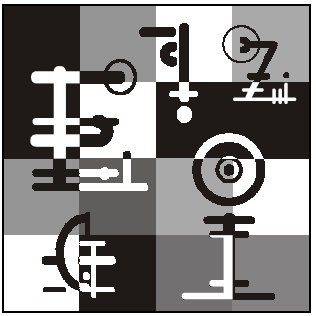
Примерные варианты индивидуальных заданий

1. Требуется спроектировать регистр (буфер), накапливающий последовательно поступающие на вход разряды числа, а затем, по команде, выдающий на многоразрядный выход это число в параллельном коде. Разрядность числа выбирается по желанию (минимум 4).
2. Спроектировать регистр, в который в параллельном коде записывается многоразрядное число, а затем с выхода в последовательном коде (поразрядно) снимается записанное число. Разрядность числа выбирается по желанию (минимум 4).
3. Спроектировать счетчик до 15 с возможностью сброса в процессе счета и возможностью начала отсчета с заданного числа.

4. Спроектировать компаратор двух чисел (больше, меньше, равно).
5. Спроектировать устройство, на выходе которого будем иметь максимальное и минимальное число из поданных перед этим на его входы четырех чисел.
6. Создать мультиплексор, выбирающий один из четырех входов данных.
7. Создать устройство, на выходе отображающее наоборот поданное на вход многоразрядное число. Разрядность числа выбирается по желанию (минимум 4).
8. Спроектировать дешифратор на 4 входа и 10 выходов. Неопределенные входные значения обнуляют соответствующие выходы.
9. Используя RS-триггер, построить JK-триггер.
10. Реализовать делитель тактовой частоты на 2.

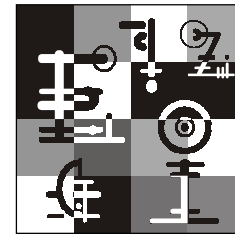
Программа работы

1. Открыть текстовый редактор системы VeriLogger Pro / TestBencher Pro, ввести Verilog-проект, написанный по индивидуальному заданию.
2. В редакторе временных диаграмм (вэформере) в соответствии со спецификацией задать на входы проекта необходимые сигналы.
3. Просимулировать Verilog-проект, написанный по индивидуальному заданию, убедиться в правильности его работы.



Часть VI

Средства проектирования фирмы Mentor Graphics



Глава 28

Редакторы системного и архитектурного уровней (HDL Designer). Примеры использования

Средство автоматического проектирования FPGA Advantage фирмы Mentor Graphics предназначено для создания проектов, с использованием языков VHDL и Verilog HDL, моделирования и синтеза, плавно переходя от одного шага проектирования к следующему.

В состав пакета входят такие инструментальные средства:

- HDL Designer — служит средой создания проекта, повторного использования и управления;
- ModelSim — средство симуляции проекта, предоставляет возможность пошаговой отладки проекта;
- Leonardo Spectrum — средство синтеза и оптимизации HDL-проектов.

Все перечисленные подсистемы объединены одной графической оболочкой проектов Design Browser, позволяющей создавать, открывать, закрывать, копировать файлы проектов и библиотек, а также осуществлять запуск подсистем ModelSim и Leonardo Spectrum.

HDL Designer — это среда для создания, повторного использования, управления HDL-проектами. VHDL- и Verilog HDL-проекты создаются и редактируются с помощью текстовых и графических HDL-редакторов.

В состав HDL Designer входят следующие графические редакторы:

- Block Diagram — редактор структурных блок-схем, компоненты представляются в виде блоков, объединенных между собой сигналами и шинами. Каждый блок может быть описан в любом из редакторов;
- State Diagram — редактор конечных автоматов;
- Flow Chart — редактор алгоритмов;
- Truth Table — редактор таблиц истинности.

Графические редакторы ускоряют создание проекта и автоматически генерируют эффективный VHDL- и Verilog HDL-код.

28.1. Оболочка Design Browser

Оболочка содержит меню команд, панель быстрого вызова, а также следующие окна:

- **Source** — окно со списком каталогов и файлами проектов;
- **HDL** — окно с файлами VHDL-, Verilog HDL-описаний открытого проекта;
- **Downstream** — окно с вкладками **Model Sim** (для отображения файлов, создаваемых при моделировании проекта) и **Leonardo** (для файлов, полученных в результате синтеза).

Кнопки панели быстрого запуска:

- — открыть библиотеку;
- — закрыть библиотеку;
- — генерация HDL для графического описания;
- — компиляция проекта;
- — запуск симуляции проекта с помощью Model Sim;
- — запуск синтеза проекта с помощью Leonado Spectrum.

28.2. Редактор Block Diagram

Для создания новой структурной блок-схемы проекта, необходимо выполнить команду **File** | **New** | **Block Diagram** (Файл | Новый | Блок-схема) в главном меню оболочки Design Browser.

В данном редакторе можно использовать следующие элементы:

- **Add** | **Block** (Добавить | Блок) — добавление пустого блока в структурную схему. Каждый блок в свою очередь может быть описан структурной схемой, блок-схемой алгоритма, конечным автоматом, таблицей истинности или непосредственно HDL-описанием. Для описания какого-либо блока необходимо дважды щелкнуть по нему левой кнопкой мыши, после этого выбрать редактор, с помощью которого предполагается описывать блок;
- **Add** | **Component** (Добавить | Компонент) — добавление готового компонента, т. е. компонента, уже разработанного заранее в данном проекте, либо из других библиотек. В данном случае компонент должен иметь файл символа (Symbol);

- **Add** | **ModuleWare** (Добавить | Параметрический модуль) — добавление компонента из параметризированной библиотеки;
- **Add** | **IP** (Добавить | Файл интеллектуальной собственности) — добавление файла интеллектуальной собственности, необходимо указать путь к HDL-файлу;
- **Add** | **Embedded Block** (Добавить | Встроенный блок) — добавление встроенного блока. Внедренный блок сохраняется как часть блок-схемы исходного проекта и не налагает иерархию. Имя внедренного блока должно быть уникально на схеме, оно используется как метка в сгенерированном HDL-описании;
- **Add** | **Signal** (Добавить | Сигнал) — добавление сигналов, объединяющих блоки. Каждый сигнал имеет уникальное имя, которое можно редактировать;
- **Add** | **Bus** (Добавить | Шина) — добавление шины;
- **Add** | **Global Connector** (Добавить | Глобальный соединитель) — добавление глобального соединителя. После добавления такого соединителя (круг желтого цвета), сигнал, линия которого будет с ним соединена, станет глобальным (например, сигнал синхронизации);
- **Add** | **Port In** (Добавить | Входной порт) — добавление входного внешнего порта;
- **Add** | **Port Out** (Добавить | Выходной порт) — добавление выходного внешнего порта;
- **Add** | **Port InOut** (Добавить | Двунаправленный порт) — добавление двунаправленного внешнего порта.

28.3. Редактор State Diagram

В данном редакторе блок описывается в виде направленного графа, с условными и безусловными переходами, т. е. блок описывается в виде конечного автомата.

Начальное состояние отображается зеленым кругом, остальные — синими. Также возможно создание иерархических диаграмм, тогда состояние, имеющее дочернюю диаграмму, отображается в виде темно-синего круга с тройной рамкой.

Для описания блока в виде конечного автомата, необходимо выполнить команду **File** | **New** | **State Diagram** (Файл | Новый | Диаграмма состояний) или дважды щелкнуть мышью по нужному блоку на структурной блок-схеме.

В данном редакторе можно использовать следующие элементы:

- **Add** | **State** (Добавить | Состояние) — добавление нового состояния;
- **Add** | **Transition** (Добавить | Переход) — добавление перехода между состояниями;

- **Add** | **Hierarchical State** (Добавить | Иерархическое состояние) — добавление иерархического состояния, содержимое которого определяется вложенной диаграммой состояний;
- **Add** | **Entry Point** (Добавить | Точка входа) — добавление точки входа в диаграмму состояний (только для вложенных диаграмм);
- **Add** | **Exit Point** (Добавить | Точка выхода) — добавление точки выхода из диаграммы состояний (только для вложенных диаграмм).

28.4. Редактор Flow Chart

Для описания блока в виде блок-схемы алгоритма, необходимо выполнить команду **File** | **New** | **Flow Chart** (Файл | Новый | Блок-схема алгоритма) или дважды щелкнуть мышью по нужному блоку на структурной блок-схеме.

В данном редакторе можно использовать следующие элементы:

- **Add** | **Start Point** (Добавить | Точка старта) — добавление начального элемента блок-схемы;
- **Add** | **End Point** (Добавить | Точка конца) — добавление конечного элемента блок-схемы;
- **Add** | **Action Box** (Добавить | Блок действий) — добавление функционального блока;
- **Add** | **Decision Box** (Добавить | Блок условного перехода) — добавление условного перехода;
- **Add** | **Case** (Добавить | Блок выбора) — добавление условного перехода по набору значений переменной, сигнала, константы или параметра, стоящих в условии;
- **Add** | **Wait Box** (Добавить | Блок ожидания) — добавление блока ожидания до выполнения определенного условия, заданного пользователем;
- **Add** | **Loop** (Добавить | Начало цикла) — добавление блока начала тела цикла;
- **Add** | **End Loop** (Добавить | Конец цикла) — добавление блока окончания тела цикла;
- **Add** | **Flow** (Добавить | Направленная линия) — добавление направленной линии, соединяющей блоки;
- **Add** | **Hierarchical Action Box** (Добавить | Иерархический блок) — добавление иерархического функционального блока, содержимое которого представляется отдельной вложенной блок-схемой.

28.5. Редактор Truth Table

Для описания блока в виде таблицы истинности, необходимо выполнить команду **File | New | Truth Table** (Файл | Новый | Таблица истинности) или дважды щелкнуть мышью по нужному блоку на структурной блок-схеме.

При работе с данным редактором, можно выполнять следующие действия, которые доступны только из контекстного меню:

- **Add Column / Delete Column** (Добавить колонку / Удалить колонку) — добавление/удаление нового столбца таблицы истинности;
- **Add Row / Delete Row** (Добавить строку / Удалить строку) — добавление/удаление новой строки таблицы истинности.

Лабораторная работа. Знакомство с HDL-дизайнером на примере проекта "Таймер"

Цель работы

Освоение работы в редакторах HDL-дизайнера на примере описания ими проекта "Таймер" по его спецификации.

Спецификация проекта "Таймер"

Таймер выводит данные времени посредством двух шин по четыре бита, представляющих значения десятков (`high`) и единиц (`low`). Есть также выход, который запускает звуковой сигнал (`alarm`). Входные сигналы поступают через шину данных из 10 разрядов и управляющие входы `start`, `stop`, `reset` и `clock` (табл. 28.1).

Таблица 28.1. *Входные и выходные сигналы таймера*

Inputs	**Outputs**
`start` (logic signal)	`high` (4-bit bus)
`stop` (logic signal)	`low` (4-bit bus)
`reset` (logic signal)	`alarm` (logic signal)
`clk` (logic signal)	
`d` (10-bit bus)	

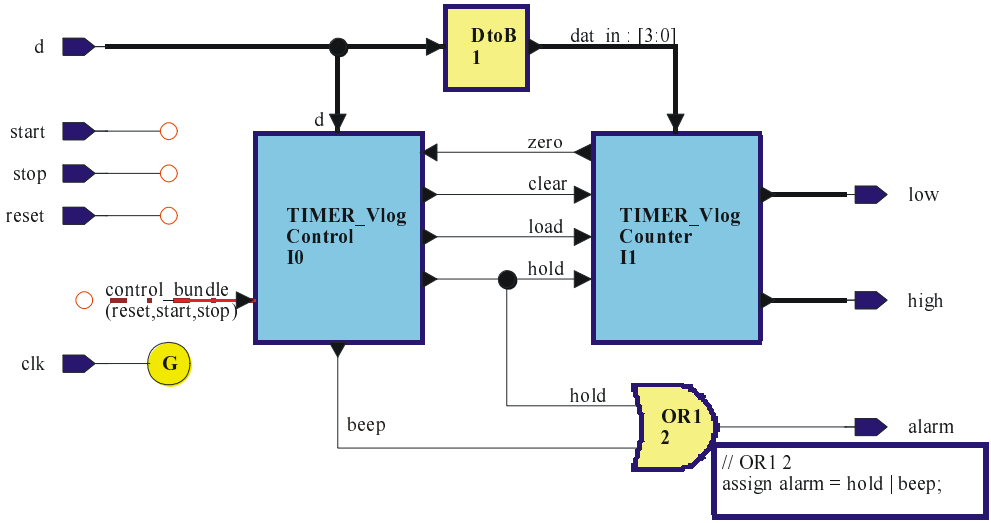


Рис. 28.1. Блок-схема таймера

Блок-схема проекта "Таймер" приведена на рис. 28.1.

Для начала работы необходимо установить таймер в рабочее положение и подать на информационный вход некоторое число. По завершении счета таймер выдает соответствующий сигнал. Предусмотрена возможность временно приостановить работу таймера.

Блок `Timer` содержит входы: `start` — для начала работы таймера, `stop` — для остановки, `reset` — для сброса значения таймера, `d` — информационный вход, `clk` — синхронизация. На выходе получаем значение таймера (`high` — десятки, `low` — единицы) и сигнал о завершении счета — `beep`.

Таймер состоит из счетчика и управляющего им устройства. На вход таймера поступает десятичное число `d`, поэтому необходим блок перевода его в двоичную систему.

Устройство управления (`Control`) устанавливает сигналом (`load`) и сбрасывает сигналом (`clear`) значение счетчика, сигнализирует о начале и остановке счета сигналом (`hold`), подает сигнал о его завершении (`beep`).

Счетчик (`Counter`) выдает текущее значение, а также сообщает устройству управления о завершении счета сигналом (`zero`).

Блок `DtoB` конвертирует десятичное число в двоичное.

Описание блока `Control` представлено на рис. 28.2.

Устройство управления описано с помощью конечного автомата. Система находится в любом состоянии не менее одного такта, а переходы осуществляются при выполнении соответствующих условий. Альтернативные переходы имеют приоритеты.

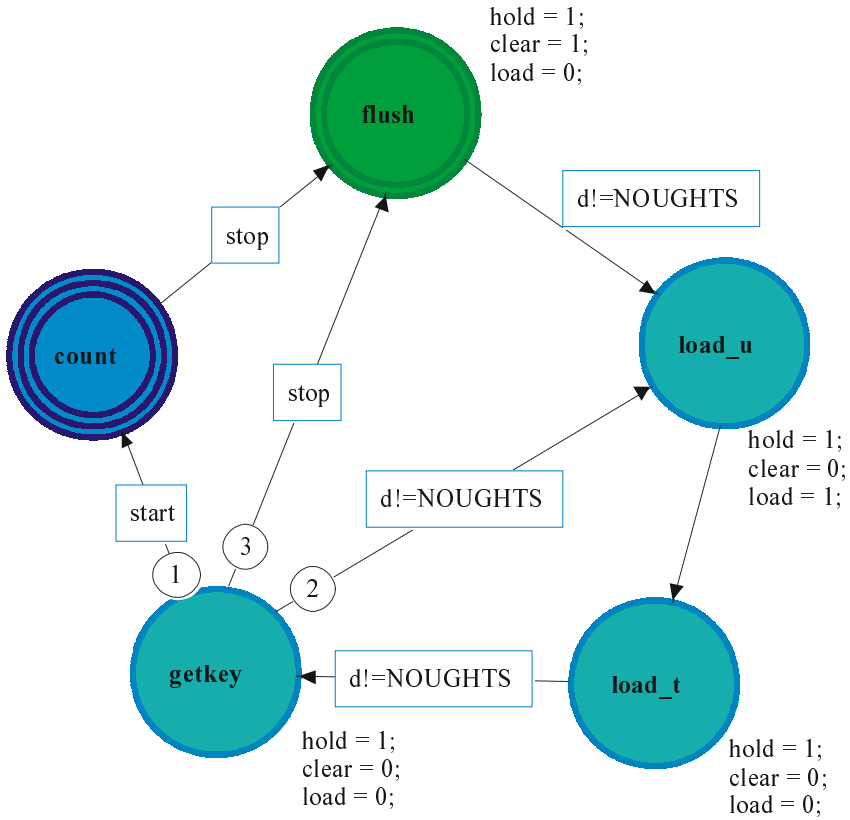


Рис. 28.2. Конечный автомат блока `Control`

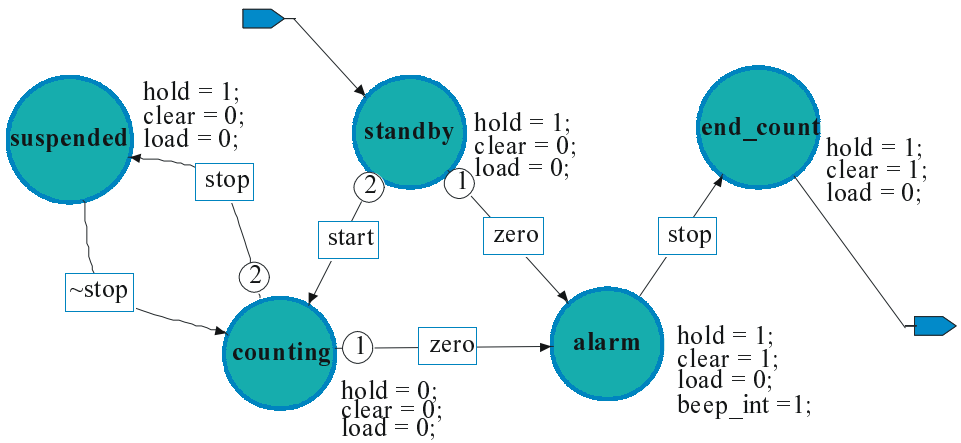


Рис. 28.3. Дочерняя схема состояния `count`

Из начального состояния (`flush`) устройство выходит при условии, что на информационном входе не нуль. Устройство посылает счетчику сигнал о запоминании поданного на вход числа и ожидает команды `Start` для начала работы счетчика.

Состояние `count` является составным. Описывающий его конечный автомат приведен на рис. 28.3. Входное состояние здесь `standby`. При получении сигнала `start` система посылает сигнал о включении счетчика (`hold<=0`). При поступлении сигнала `stop` счетчик останавливается (`hold<=1`), а при отсутствии сигнала `stop` счет продолжается. При получении от счетчика сигнала о завершении работы (`zero=1`) устройство посылает сигнал `beep`, пока не получит сигнал `stop`.

Описание блока `Counter` представлено на рис. 28.4.

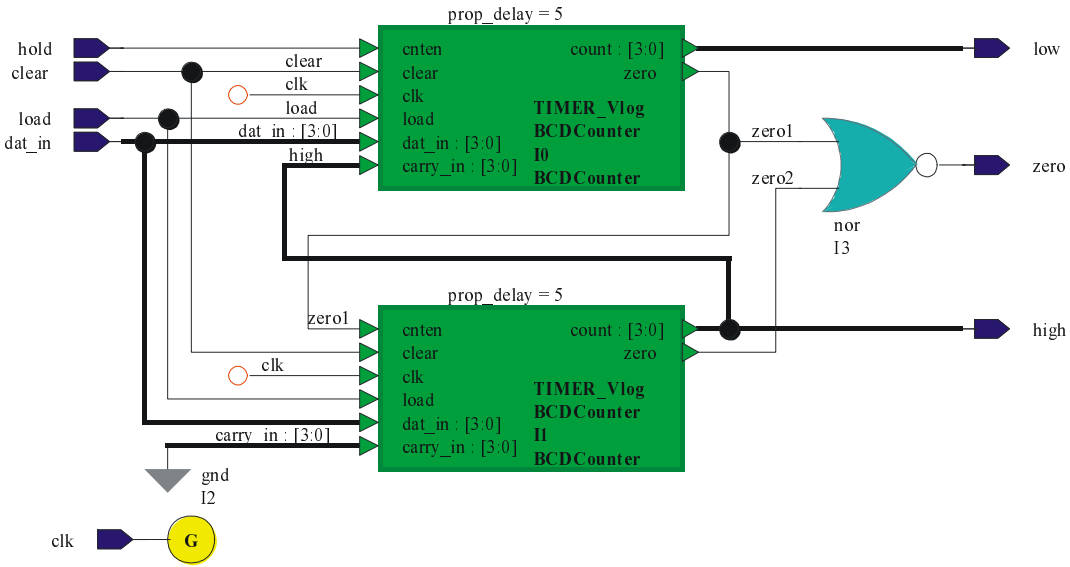


Рис. 28.4. Блок-схема `Counter`

Блок `Counter` состоит из двух блоков `BCDCounter`. Один для счета десятков, второй — единиц. Блок десятков имеет выходы `high` и `zero_0`. У блока единиц — `low` и `zero_1`.

Счет начинается при приеме сигнала `hold=0`. Начальное значение и десятков и единиц считывается с шины `dat_in`. Когда показатель счетчика единиц уменьшается до нуля, он посылает нуль на `zero_1`, и если значение десятков не нуль, на счетчике устанавливается значение 9. При появлении на `zero_1` нуля счетчик десятков уменьшает свое значение на 1. Счетчик посылает сигнал о завершении счета (`zero=1`), если на `zero_1` и `zero_2` подан 0.

Каждый блок `BCDCounter` описан на языке Verilog HDL:

```
module BCDCounter (cnten, clear, clk, load, dat_in, carry_in, count, zero);

  parameter prop_delay = 10;

 input cnten, clear, clk, load;
  input [3:0] dat_in;
  input [3:0] carry_in;
  output zero;
  output [3:0] count;
  reg zero;
  reg [3:0] count;
  reg [3:0] int_count;

  always @ (posedge clk)
  begin
    if (clk == 1)
    begin
      zero = 1;
      if (clear == 1) int_count = 4'b0000 ;
      else if (load == 1) int_count = dat_in ;
      else if (cnten == 0)
      begin
        if ((int_count == 4'b0000) && (carry_in != 4'b0000)) int_count =
4'b1001 ;
        else int_count = int_count - 1'b1 ;
      end
      else int_count = int_count ;
    end
    zero = (int_count[3] | int_count[2]) | (int_count[1] | int_count[0]);
    #prop_delay count = int_count ;
  end
endmodule
```

Блок преобразования десятичных чисел в двоичные числа (`DtoB`) организован с помощью таблицы истинности (табл. 28.2).

Таблица 28.2. *Таблица истинности блока* `DtoB`

d[9]	d[9]	d[9]	d[9]	d[9]	d[9]	d[9]	d[9]	d[9]	d[9]	d_out
									1	4'b0000
								1		4'b0001
							1			4'b0010
						1				4'b0011
					1					4'b0100
				1						4'b0101
			1							4'b0110
		1								4'b0111
	1									4'b1000
1										4'b1001
										4'b0000

Программа работы

Работа с редакторами HDL-дизайнера рассматривается на примере создания рассмотренного проекта "Таймер".

1. Создание библиотеки.

   Для создания библиотеки выберите **New Library** (Новая библиотека) из меню **File** в окне **Design browser**, затем в окне **Add New Library Mapping** введите имя библиотеки (например, TUTORIAL) и определите имя пути для корневого каталога, который будет содержать ваши библиотечные данные (например, D:\Designs).

   Каталог данных проекта отображается в браузере проекта как открытая "книга".

2. Установка языка по умолчанию.

   Выберите **Options** | **Main**, вкладку **General** и проверьте, что **Verilog** установлен как заданный по умолчанию язык.

3. Создание блок-схемы.

   Для создания новой блок-схемы нужно выбрать пиктограмму **New** на панели инструментов или использовать команду меню **File** | **New** | **Block Diagram**.

Создается пустое поле с фоновой сеткой, списком директив компилятора (**'resetall** и **'timescale 1ns/10ps** по умолчанию) и пустых текстовых полей с метками для объявлений портов и сигналов схемы.

4. Добавление блоков.

   Для добавления блока нужно выбрать пиктограмму **Add Block** на панели инструментов или использовать команду меню **Add | Block**.

   Блоки добавляются с заданными по умолчанию библиотекой `<Iibrary>`, именем `<block>` и уникальным идентификатором (`I0`, `I1`, ...).

   Команда вставки повторяется автоматически, пока не будет выбрана другая команда или нажата правая кнопка мыши, или <Esc>.

   Добавьте два блока в схему. Расположение блоков сейчас и в дальнейшем следует вести в соответствии с рис. 28.1.

5. Добавление внедренных блоков.

   Добавьте два внедренных блока, выбрав пиктограмму **Add Embedded Block** или используя команду меню **Add | Embedded Block**.

6. Добавление портов и сигналов.

   Выберите **Add Add Signal with a Port** (Добавить сигнал с портом) из списка на панели инструментов, чтобы создать три сигнала, выходящие из блока `I0` слева к блоку `I1`, и сигнал от `I1` к `I0` (см. рис. 28.1). Сигналы добавлены с уникальными именами (`sig0`, `sig1`, `sig2` и `sig3`) и заданным по умолчанию типом `wire`.

   Добавьте сигналы от `I0` и от точки на `sig2` до внедренного блока `eb2` и сигнал от `eb2` до правой стороны схемы, дважды щелкнув мышью. Обратите внимание, что в данном случае добавляется символ выходного порта, и его объявление добавляется в список портов схемы.

   Выберите **Add Bus with a Port** (Добавить шину с портом) из списка на панели инструментов, чтобы добавить шину от свободного пространства левой части схемы к внедренному блоку `eb1`. И шину, начинающуюся на этой шине и заканчивающуюся на блоке `I0`. Обратите внимание, что обе шины имеют заданное по умолчанию имя `dbus0`. Полное объявление показывает заданные по умолчанию тип шины и границы `wire [15:0]`.

   Добавьте шину `dbus1` от `eb1` до `I1` и две шины (`dbus2` и `dbus3`) от `I1`, заканчивающихся выходными портами в правой части схемы.

7. Добавление пучка и глобального коннектора.

   Выберите **Add Signal with a Port** из списка на панели инструментов, чтобы добавить три сигнала в левой части схемы, заканчивающихся в пустом пространстве.

Выберите эти три сигнала (перемещая прямоугольник выбора, удерживая левую кнопку мыши) и используйте команду **Add Bundle** (Добавить пучок) на панели инструментов, чтобы подключить пучок, содержащий эти три сигнала, к блоку `IO`. Обратите внимание, что пучок имеет заданное по умолчанию имя `bundle0`, а три выбранных сигнала автоматически включены в пучок с их именами, перечисленными в имени пучка.

Выберите **Add Global Connector**, чтобы добавить глобальный соединитель на схеме ниже пучка, и **Add Signal with a Port**, чтобы добавить сигнал между глобальным соединителем и входом. (Это будет синхросигналом, который связан с каждым блоком на схеме.)

8. Сохранение блок-схемы.

   Диалоговое окно **Save As** позволяет выбирать библиотеку и определять модуль проекта и имя модуля. Выберите библиотеку `TUTORIAL` (или ваше имя), и введите модуль проекта `Timer`.

   Имя может быть введено при использовании любого действительного HDL-идентификатора, но обычно это значения по умолчанию:

   - struct.bd — блок-схема;
   - struct.ibd — представление интерфейса проекта;
   - fsm.sm — автомат состояний;
   - flow.fc — алгоритм;
   - tbl.tt — таблица истинности;
   - symbol.sb — символ.

   Если расширение, обозначаемое двумя символами, опущено, оно автоматически добавится, чтобы идентифицировать тип сохраняемой схемы. Заданные по умолчанию имена могут быть изменены за исключением расширения.

9. Редактирование имен блоков и сигналов.

   Переименуйте блоки согласно рис. 28.1.

   Для редактирования имен сигналов и портов можно использовать прямое редактирование текста. Также можно использовать диалоговое окно, которое позволяет редактировать свойства выбранного объекта. Выберите **Object Properties** (Свойства объекта) на панели инструментов или пункт меню **Edit | Object Properties**, вкладку **Declarations** (Объявление).

   Измените имена портов и сигналов в соответствии с табл. 28.3 и 28.4.

Таблица 28.3. *Порты*

Old Name	New Name	Type	Bounds
dbus0	d	wire	9 : 0
sig6	start	wire	none
sig7	stop	wire	none
sig8	reset	wire	none
sig9	clk	wire	none
dbus2	low	wire	3 : 0
dbus3	high	wire	3 : 0
sig 5	alarm	wire	none

Таблица 28.4. *Сигналы*

Old Name	New Name	Type	Bounds
dbus1	dat_in	wire	3 : 0
sig0	clear	wire	none
sig1	load	wire	none
sig2	hold	wire	none
sig3	zero	wire	none
sig4	beep	wire	none

На вкладке **Bundles** измените имя пучка на `control _bundle`.

10. Добавление HDL-описания.

    Выберите внедренный блок `OR1` и откройте контекстное меню, с помощью правой кнопки мыши. Выберите **New View** (Новое представление) из меню **Open**, чтобы отобразить диалоговое окно **Create Embedded View**. Выберите **Text** из списка представлений в диалоговом окне.

    Когда подтвердите выбор, на блок-схеме отобразится HDL-текст рядом со встроенным блоком. Текст можно редактировать на схеме или в контекстном меню **Object Properties** вкладки **Text**.

    Отметьте опцию **Resize to fit text** и введите следующую инструкцию Verilog под комментарием *`// OR1 2`* в диалоговом окне:

    ```
    assign alarm = hold | beep;
    ```

    Функциональные блоки на схеме по умолчанию имеют простую прямоугольную форму. Однако иногда полезно использовать представление логического бло-

ка в соответствии с его логической функцией. Например, в этой блок-схеме, внедренный блок `OR1` представляет логическую функцию `OR`. Вы можете изменить форму представления блока.

Выберите этот блок и из списка на кнопке **AutoShapes** (Автоформа) панели инструментов **Comment Graphics** выберите **Or**. Скройте стрелки входа, убрав флажок **Show Ports when connected** на вкладке **Embedded Blocks** диалогового окна **Object Properties**.

11. Создание диаграммы состояний.

    Щелкните правой кнопкой мыши на блоке **Control**, выберите **Open As | New View**.

    Из списка представлений выберите **State Diagram**.

    Новая диаграмма состояний (TUTORIAL \Control\fsm ['machine0']) создана как дочернее представление блока `Control`.

    Заданное по умолчанию имя `machine0` конечного автомата добавлено в конец к модулю проекта и имени представления, но может быть изменено выбором **Rename Concurrent State Machine** в меню **Diagram**.

    Диаграмма состояний включает текстовые объекты для заданного по умолчанию списка директив компилятора и меток для глобальных действий (**global actions**), параллельных операторов (**concurrent statements**), объявлений модулей (**module declarations**), статусов сигналов (**signals status**) и операторов регистра состояния (**state register statements**).

12. Добавление состояний и переходов.

    Используйте **Add State** на панели инструментов или команду меню **Add | State**, чтобы добавить пять состояний на диаграмму. Состояния добавлены с заданными по умолчанию именами `s0`, `s1`, `s2`, `s3` и `s4`. Обратите внимание, что первое состояние — стартовое, имеет зеленый цвет и двойную рамку.

    Выберите пиктограмму **Add Transition** (Добавить переход) или команду меню **Add | Transition** и добавьте переходы в соответствии с рис. 28.3. Обратите внимание, что, когда добавляете больше чем один переход, приоритет перехода обозначается номером.

    Сохраните диаграмму.

13. Редактирование состояний.

    Выберите начальное состояние `s0` и пиктограмму **Object Properties**, откройте вкладку **States** диалогового окна **State Machine Object Properties**. (Так же можно отобразить диалоговое окно, дважды щелкнув на состоянии.)

    Вкладка **States** позволяет вводить имя и текст действий для одного или более выбранных состояний на диаграмме.

Введите следующие имена состояний и действий (значений сигналов), заменяющие заданные по умолчанию, в соответствии с табл. 28.5.

Таблица 28.5. *Имена состояний и действий*

Old Name	New Name	Style	Actions

s0	flush	IF	hold = 1;
			clear = 1;
			load = 0;

s1	count	IF	(no actions)

s2	getkey	IF	hold = 1;
			clear = 0;
			load = 0;

s3	load_t	IF	hold = 1;
			clear = 0;
			load = 0;

s4	load_u	IF	hold = 1 ;
			clear = 0;
			load = 1;

14. Редактирование переходов.

    Добавьте условия к переходам диаграммы в соответствии с табл. 28.6.

Таблица 28.6. *Условия переходов*

Origin	Destination	Priority	Condition
count	flush	1	stop
getkey	flush	3	stop
getkey	count	1	start
getkey	load_u	2	d!=ZEROS
flush	load_u	1	d!=ZEROS
load_u	load_t	1	(none)
load_t	getkey	1	d!=ZEROS

15. Создание иерархической диаграммы состояний.

    Выберите состояние `count` и команду меню **Diagram** | **Change To** | **Hierarchical State** (Диаграмма | Изменить на | Иерархическое состояние).

Щелкните два раза мышью на иерархическом состоянии (или выберите **Open Down** из контекстного меню), чтобы создать дочернюю диаграмму. Новая диаграмма инициализируется с состоянием, связанным с точкой входа и точкой выхода.

Выберите символ выхода и **Diagram | Change to State** (Диаграмма | Изменить на состояние). Щелкните правой кнопкой мыши на первом состоянии и тащите курсор левее и ниже первого состояния, отпустите кнопку. Выберите **Copy Here** из контекстного меню, чтобы создать копию состояния в позиции курсора.

Повторите эту процедуру, чтобы добавить еще два состояния на диаграмме.

Используйте **Add Exit Point**, чтобы добавить новый выход, а также создайте переходы между состояниями в соответствии с рис. 28.3.

Используйте вкладку **States** диалогового окна **State Machine Object Properties**, чтобы переименовать состояния и добавить действия в соответствии с табл. 28.7.

Таблица 28.7. *Имена состояний и действия для дочерней диаграммы*

Old Name	**New Name**	**Style**	**Actions**

s0	standby	IF	hold= 1;
			clear = 0;
			load = 0;

s1	alarm	IF	hold= 1;
			clear = 1;
			load = 0;
			beep= 1;

s2	counting	IF	hold = 0;
			clear = 0;
			load = 0;

s3	suspended	IF	hold= 1;
			clear = 0;
			load = 0;

s4	end_count	IF	hold=1;
			clear = 1;
			load = 0;

Используйте вкладку **Transitions**, чтобы добавить условия переходов в соответствии с табл. 28.8.

Таблица 28.8. *Условия переходов для дочерней диаграммы*

Origin State	Destination State	Priority	Condition	Actions
suspended	counting	1	stop	none
counting	suspended	2	stop	none
counting	alarm	1	zero	none
standby	counting	2	start	none
standby	alarm	1	zero	none
alarm	end_count	1	stop	none

Сохраните диаграмму состояний.

16. Редактирование свойств конечного автомата.

    Используйте команду меню **File** | **Open Up**, чтобы открыть родительскую диаграмму состояний.

    Дважды щелкните левой кнопкой мыши по метке **Global Actions** (Глобальные действия) или **Concurrent Statements** (Параллельные операторы) или **State Registers** (Регистры состояния) состояния, чтобы открыть вкладку **Statement Blocks** диалогового окна **State Machine Properties**. Уберите флажки **Visible** (Видимый).

    Выберите вкладку **Module Declarations** и введите следующее объявление для константы `ZEROS` на 10 бит

    ```
    parameter   ZEROS   =   10'b0;
    ```

    Выберите вкладку **Signals Status**. Обратите внимание, что выходные сигналы, перечисленные в диалоговом окне, имеют значение по умолчанию **0** и статус **COMBINATORIAL**.

    Для сигнала `beep` установите значение поля **Status** в **REGISTERED**, а для поля **Reset** — значение **0**.

    Обратите внимание, что сигнал `beep` (в состоянии `alarm` на дочерней иерархической диаграмме состояний) был заменен внутренним именем `beep_int`.

17. Свойства генерации HDL-кода.

    Откройте диалоговое окно **State Machine Properties** и выберите закладку **Generation**. Эта закладка устанавливает характеристики, используемые для HDL-генерации.

    Выберите опцию **Synchronous** (Синхронный) в поле **Machine**. Используйте опцию по умолчанию **Case and 3 Always Blocks** для HDL-стиля.

Опции **Register state actions on next state** и **Use delay for current state assign** должны быть выключены, а **Assignment type** установлена в **Mixed**.

Выберите **clk** из списка имен синхросигналов и **Falling** в поле **Edge** (синхронизация по спаду импульса). Установите опцию **Asynchronous** (Асинхронный сброс) и выберите имя сигнала **reset** с высоким (**High**) уровнем.

Сохраните диаграмму состояний.

18. Генерация HDL-кода для конечного автомата.

    Выберите **Generate** на панели инструментов в окне диаграммы состояний. Ход HDL-генерации отображается в окне **Log Window**, которое выводит любые ошибки и предупреждения, происходящие в течение генерации.

    Сгенерированные файлы могут быть открыты из окна **HDL**.

19. Импорт модуля проекта `BCDCounter`.

    Выберите **HDL | HDL Import | Text HDL Import**, чтобы открыть **HDL Import wizard**. Выберите **Specify HDL files** на первой странице мастера и нажмите **Next**, чтобы отобразить страницу **Specify HDL Source Files**. Выберите файлы **Verilog** из фильтра **Files of type** и используйте кнопку **Brouse**, чтобы определить местоположение каталога ..\examples\tutorial_ref\Import. Выберите файл Timer_BCDCounter.v и добавьте его в список **Files to Convert**, используя кнопку **Add**. Нажмите **Next**, чтобы отобразить страницу **Design Units**. Эта страница позволяет выбирать из списка модули проекта, найденные в HDL-файлах. В данном случае, есть только **BCDCounter**, и должна быть выбрана опция **All**. Нажмите **Next**, чтобы отобразить страницу **Target Libraries**. Эта страница позволяет выбирать заданную по умолчанию библиотеку и добавлять библиотеки.

    Проверьте, что выбрана библиотека TUTORIAL и нажмите **Next**, чтобы отобразить страницу **Confirm HDL Import**.

    Нажмите **Finish**, чтобы завершить импорт HDL. В окне файла регистрации HDL должно быть отображено следующее сообщение:

    HDL Import complete.

20. Создание дочерней блок-схемы.

    Откройте блок-схему **Timer**.

    Дважды щелкните мышью (или выберите **Open** в контекстном меню) на теле блока `Counter`, чтобы отобразить диалоговое окно **Open Down Create New View**.

    Выберите **Block Diagram** из списка. Дочерняя блок-схема инициализирована с объявлениями и портами, соответствующими сигналам и шинам на схеме родительского блока (включая глобальный сигнал `clk`).

Используйте **Add Component** на панели инструментов, чтобы открыть диалоговое окно **Add Instance**. Используйте диалоговое окно, чтобы добавить два элемента (`I0` и `I1`) `BCDCounter`, разместив один ниже другого (см. рис. 28.4).

Тащите сигнал `hold` до входа `cnten` на `I0`.

Точно так же соедините `low` с `count` на `I0` и `high` с `count` на `I1`. Выделите оба элемента и выберите **Connect** из контекстного меню, для соединения.

Проведите сигнал между входом `zero` на `I0` и выходом `cnten` на `I1`. Этот сигнал автоматически будет назван `zeroI`.

Подключите шину между точкой на шине `high` и `carry_in` на `I0`.

Подключите сигнал `zero` с портом `zero`. Этот сигнал автоматически будет назван `zero2`.

Выделите оба элемента `I0` и `I1`. Выберите команду **Add Signal Stubs** (Добавить выводы сигналов) из контекстного меню.

После этого порты и входы элементов с одинаковыми именами неявно связываются. Нет необходимости рисовать соединения на схеме и можно сделать схему более "чистой".

Вы можете автоматически произвести соединения на схеме, используя команду **Diagram | Autoroute**. Проверьте опции для автоматической трассировки, выбрав **Autoroute | Autoroute Options**, чтобы отобразить диалоговое окно **Block Diagram Layout/Routing Options**.

Проверьте, что отмечено **Explicit connection**. Снимите отметки с **Align port**, **Global Connectors** и **Move component ports** и нажмите **OK**.

Выберите `clear`, `load` и `dat_in`. Выберите **Autoroute | Autoconnect** из всплывающего меню. Сигналы автоматически будут связаны с входами на `I0` и `I1`.

21. Редактирование отображения параметров.

Отображение параметров позволяет изменять значение Verilog-параметров, которые объявлены в символе, определяющем элемент, и могут иметь различные значения для каждого элемента.

Выберите элемент `I0`, затем **Object Properties**, чтобы отобразить диалоговое окно **Block Diagram Object Properties**, вкладку **Parameters**. Нажмите кнопку **Available**, чтобы отобразить список параметров, определенных в символе, и **Add All**, чтобы переместить параметр `prop_delay` в список переопределяемых параметров.

Измените значение параметра на 5 в поле **Change value**. Параметр добавляется сверху элемента на схеме как текстовый объект, который может быть перемещен.

Повторите эту процедуру, чтобы установить отображение параметра для второго элемента (`I1`) `BCDCounter`.

22. Добавление ModuleWare-компонентов.

Откройте браузер **ModuleWare**, выбрав пункт меню **Window | Module Ware Browser**. Браузер показывает содержание библиотеки moduleware, которая разделена на множество папок. Щелкните на "+" для открытия папки Constant, используйте левую кнопку мыши, чтобы перетащить элемент `gnd` на блок-схему, разместив его около входа `carry_in` элемента `I1`. Элемент ModuleWare добавится с заданным по умолчанию именем `I2`.

Соедините элемент `gnd` с `carry_in` и выберите **Connect** из контекстного меню.

Обратите внимание, что вход `carry_in` имеет 4 разряда. Элемент `Gnd` имеет разрядность 1 и должен быть согласован с входом. Щелкните дважды по элементу `gnd` (или выберите **ModuleWare Parameters** из контекстного меню), чтобы открыть диалоговое окно **ModuleWare Parameters**, выберите сигнал `dout` и установите его размер, равный 4. Проверьте, что опция **In-Line** установлена для HDL-генерации.

Повторите эту процедуру, чтобы добавить элемент `nor2` из папки Logic библиотеки moduleware.

Соедините элементы согласно рис. 28.4.

23. Добавление пользовательских объявлений.

Дважды щелкните мышью по метке **Declarations** на схеме, чтобы раскрыть вкладку **Declarations** диалогового окна **Object Properties**. Нажмите **User Declarations**, определите сигнал `gnd` как бинарную нулевую константу на 4 бита. Пользовательские объявления помещаются на схеме под меткой `Pre user`.

Сохраните блок-схему.

24. Создание таблицы истинности.

Откройте блок-схему `Timer`. Дважды щелкните по блоку `DtoB`, чтобы открыть диалоговое окно **Create Embedded View**, и выберите **Truth Table**.

25. Редактирование таблицы истинности.

Щелкните мышью на одном из синих столбцов (A или B) и используйте опцию **Add Column** из контекстного меню, чтобы добавить еще восемь столбцов (от C до J).

Щелкните мышью на одной из синих строк (строки 2, 3, 4 или 5) и используйте **Add Row**, чтобы добавить еще семь строк (6—12) к таблице истинности.

Переименуйте входные столбцы, от `d [9]` до `d [0]`. (Сигнал `clk` не используется и может быть переименован.)

Переименуйте столбец на `d_out` и введите следующие значения:

```
4'b0000
4'b0001
4'b0010
4'b0011
4'b0100
4'b0101
4'b0110
4'b0111
4'b1000
4'b1001
4'b0000
```

Таблица истинности должна выглядеть, как табл. 28.2.

Последняя пустая строка в таблице истинности представляет условие `ELSE` и задает выходное значение, когда ни одно из входных выражений не истинно.

26. Установка свойств таблицы истинности.

    Выберите **Truth Table Properties** из меню **Table**, чтобы открыть диалоговое окно **Truth Table Properties**, вкладку **Concurrent Statements** и введите:

    ```
    assign   dat_in   =   d_out;
    ```

    Данная инструкция связывает `d_out` и `dat_in`. Это необходимо, потому что Verilog-трансляция требует, чтобы выход таблицы истинности имел тип `reg`, но `dat_in` имеет тип `wire`.

    Сигнал `d_out` должен быть также объявлен в модуле проекта, а так как таблица истинности — внедренный модуль проекта `Timer`, то это надо сделать в исходной схеме (рассмотрено в пункте 9).

27. Объявление модуля.

    Откройте блок-схему `Timer`. Дважды щелкните по метке **Declarations**, чтобы открыть закладку **Declarations** диалогового окна **Object Properties**. Нажмите кнопку **User Declarations**. Введите следующее объявление сигнала:

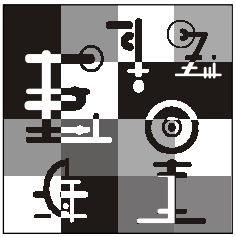
    ```
    reg[3:0] d_out;
    ```

    Сохраните блок-схему `Timer`.

28. Генерация HDL-кода для иерархической структуры.

    Выделите модуль проекта **Timer** в окне браузера проекта и выберите пункт меню **HDL | Generate | Hierarchy Through Components**.

    Ход генерации отображается в окне **Log Window**, включая любые ошибки и предупреждения, найденные в процессе генерации.



Глава 29

Симулятор (Model Sim). Пример использования

Model Sim — инструмент для симуляции и отладки VHDL-, Verilog HDL- и смешанных проектов.

Запуск проектов на симуляцию в Model Sim можно осуществлять из единого интерфейса Design Browser, из него можно автоматически генерировать и компилировать проекты.

29.1. Графический интерфейс пользователя Model Sim

Графический интерфейс пользователя Model Sim (табл. 29.1) состоит из различных окон, которые дают доступ к многочисленным средствам отладки.

Таблица 29.1. *Состав графического интерфейса пользователя Model Sim*

Окно	Описание
Main window	Центральное окно графического интерфейса
Active Processes	Отображает все процессы, которые запланированы к запуску в течение текущего цикла
Dataflow	Отображает "физическую" связность компонентов и позволяет прослеживать события (причинную связь)
List	Показывает данные временной диаграммы в табличном формате
Memory	Показывает блоки памяти и их содержимое
Watch	Показывает значения сигналов или переменных в текущее время симуляции
Objects	Показывает значения всех объявленных данных выбранных объектов

Таблица 29.1 *(окончание)*

Окно	Описание
Source	Текстовый редактор для просмотра и редактирования HDL-файлов
Transcript	Отображает историю запуска команд и сообщения, а также обеспечивает интерфейс командной строки
Wave	Отображает временные диаграммы
Workspace	Обеспечивает легкий доступ к проектам, библиотекам и т. д.

29.1.1. Окно *Main window*

Окно **Main window** обеспечивает удобный доступ к библиотекам проекта и объектам, командам отладки, сообщениям при моделировании и т. д.

После запуска окно содержит следующие компоненты:

- Workspace — отображает объекты проекта в иерархическом формате;
- Transcript — отображает историю запуска команд и сообщения, а также обеспечивает интерфейс командной строки, где можно вводить команды Model Sim;
- Objects — отображает объекты проекта, такие как сигналы, цепи и т. д.

Компонент Workspace

Workspace обеспечивает удобный доступ к проектам, библиотекам, файлам проекта, скомпилированным единицам проекта. Его окно может быть скрыто или отображено командой **View** | **Workspace**.

Окно **Workspace** содержит следующие вкладки:

- **Project tab** — показывает все файлы, которые включены в открытый проект;
- **Library tab** — отображает библиотеки проекта и откомпилированные единицы проекта;
- **Structure tabs** — отображает иерархическое представление данных для симуляции (вкладка **Sim**);
- **Files tab** — показывает исходные файлы для загруженного проекта;
- **Memories tab** — отображает иерархический список всех блоков памяти проекта;
- **Compare tab** — отображает объекты, которые были созданы при сравнении временных диаграмм.

Компонент Transcript

Отображает историю введенных команд и сообщений, а также поддерживает командную строку. Имеется возможность листать историю команд и повторно вызывать команды с помощью курсорных клавиш.

29.1.2. Окно *Dataflow*

Окно **Dataflow** позволяет исследовать "физические" связи проекта, прослеживать события, происходящие во время симуляции проекта, и идентифицировать причину неожиданных результатов.

Отображаются процессы, сигналы, регистры и соединительные провода.

Окно имеет встроенные изображения для всех примитивов Verilog HDL (таких как AND, OR и т. д.).

Для компонентов, отличных от примитивов Verilog HDL, можно определить соответствие между процессами и встроенными символами.

Лабораторная работа. Симуляция проекта "Таймер" и устройств ЦТ с помощью Model Sim

Цель работы

При симуляции проектов с помощью Model Sim необходимо создание Test Bench (испытательных стендов). В данной лабораторной работе показано, как создавать испытательные стенды с помощью редактора Flow Chart, проведена симуляция проекта для этого теста.

Программа работы

Создание Test Bench

1. Открытие диалогового окна и загрузка тестируемого проекта.

   Выберите модуль проекта `Timer` в браузере проекта и пункт меню **File** | **Create Test Bench** (Файл | Создать Test Bench), чтобы открыть диалоговое окно **Create Test Bench**. Обратите внимание, что по умолчанию выбраны библиотека `TUTORIAL`, имя теста `Timer_tb`, имя графического блока теста `TimerTester`.

Элемент `TimerTester` имеет входы и выходы, которые идентичны блоку `Timer`, но то, что является входом у одного, у другого является выходом и наоборот. Порты обоих блоков неявно связаны по имени, и нет необходимости явно задавать связи.

2. Создание алгоритма.

   Дважды щелкните по блоку `TimerTester` на блок-схеме и выберите **Flow Chart** в диалоговом окне **Open Down Create New View**. Новая блок-схема создается как дочернее представление блока `TimerTester`.

   Выберите **Diagram** | **Rename Flow Chart** и введите новое имя `Monitor`.

3. Свойства Flow Chart.

   Дважды щелкните мышью по метке **Module Declarations** на блок-схеме, чтобы отобразить вкладку **Module Declarations** диалогового окна **Flow Chart Properties**.

   Введите следующее объявление, которое установит некоторые внутренние переменные и простую процедуру ожидания:

```
integer   j;
reg    [9:0]d_var ;
reg    [3 : 0]b_var;
reg    [3 : 0]  high_temp;
reg   [3:0]low_temp;
initial
begin : SetVar
b_var = 4'b0;
d_var = 10'b0;
end
task wait_clock;
input clk_ticks;
integer clk_ticks;
repeat (clk_ticks) @(posedge clk);
endtask
```

   Используйте кнопку **Apply**, чтобы отобразить эти свойства на блок-схеме.

   Выберите закладку **Concurrent Statements**. Введите следующие инструкции, которые генерируют синхросигнал для теста:

```
initial
begin : ClockGen
parameter clk_prd = 50;
clk =0;
forever #clk_prd clk = ~clk;
end
```

4. Добавление точки старта и блока действий.

   Используйте пиктограмму **Add Start Point** на панели инструментов, чтобы добавить точку старта сверху блок-схемы.

   С помощью пиктограммы **Add Action Box** добавьте блок действий ниже. Нажатие **Add Flow** добавит стрелку, соединяющую эти блоки.

   Дважды щелкните мышью по блоку `action box`, чтобы отобразить вкладку **Action Boxes** диалогового окна **Object Properties**.

   Вы можете изменить имя блока и ввести инструкции действия. Однако имя на блок-схеме должно быть уникально. Оно не может быть изменено, когда выбран больше чем один блок.

   Измените заданное по умолчанию имя на `Initialization` и введите следующие действия:

```
stop=0;
start=0;
reset=0;
d=4'b0;
wait_clock(2);
reset=1;
wait_clock(6);
reset=0;
```

5. Добавление цикла и связанных комментариев.

   Используйте **Add Loop** на панели инструментов, чтобы добавить начало цикла.

   Дважды щелкните по блоку действий на блок-схеме, чтобы отобразить вкладку **Loops** диалогового окна **Object Properties**. Выберите кнопку **Specify** и введите следующую инструкцию проверки цикла:

```
for (j=0;j<=9;j=j+1)
```

   Измените имя на `L1`.

   По умолчанию, цикл будет повторяться бесконечно.

   С помощью пиктограммы **Add Action Box** добавьте ниже блок действий. И определите для него следующие действия:

```
d_var=10'b0;
d_var[j]=1;
d=d_var;
wait_clock(4) ;
if ( (high!=b_var)&&(low!=b_var))
$display ("Decoder or Load failure.");
b_var=b_var+1'b1;
```

Измените имя блока на `Decoder`.

Используя пиктограмму **Add End Loop**, добавьте конец цикла ниже блока `Decoder` и свяжите блоки.

Используйте пиктограмму **Text**, чтобы войти в текстовый режим. Щелкните на свободном пространстве около цикла и введите следующий текст в поле ввода комментария:

```
Loop through all possible decoder values and check output values
```

Нажмите левую кнопку мыши вне поля ввода текста, чтобы завершить ввод. Выберите текст комментария и команду **Include in HDL** из контекстного меню, затем **Before Object**. Щелкните левой кнопкой мыши по началу цикла, чтобы связать примечание с циклом.

Текст комментария будет включен в сгенерированный HDL для алгоритма.

Добавьте еще один блок действий, переименуйте его на `Store` и определите для него следующие действия:

```
d=10'b0000001000;
wait_clock(4);
start=1;
wait_clock(4) ;
start=0;
d=10'b0;
wait_clock(8);
stop=1;
wait_clock(1) ;
high_temp = high;
low_temp = low;
wait_clock(3);
```

6. Добавление иерархического блока действий.

   Когда блок-схема содержит большое количество процедурных инструкций, она может стать очень большой, однако диаграмма может быть представлена множеством отдельных иерархических диаграмм.

   С помощью пиктограммы **Add Hierarchical Action Box** добавьте иерархический блок и переименуйте его на `Check_Output`. Двойной щелчок на `Check_Output` откроет новую дочернюю схему.

   Дочерняя блок-схема имеет стартовый и конечный блоки, а также блок действий.

7. Добавление блока выбора.

   Удалите связи у стартового и конечного блоков.

   Используйте пиктограмму **Add Decision Box**, чтобы добавить блок выбора. Чтобы изменить имя блока на `Check_Suspend`, перейдите на вкладку **Decision Boxes** диалогового окна **Object Properties**, введите следующее условие:

   ```
   (high==high_temp)
   && (low==low_temp)
   ```

   Выберите уже существующий блок действий. Используйте вкладку **Action Boxes** диалогового окна **Object Properties**, чтобы изменить его имя на `Pass`, и введите следующее действие:

   ```
   $display ("Count suspended correctly");
   ```

   Добавьте новый блок действий для ветки `False` (`F`). Используйте вкладку **Action Boxes**, чтобы изменить его имя на `Fail`, и введите следующие действия:

   ```
   $display ("Count did not suspend");
   ```

8. Добавление блока `Wait`.

   Добавьте ниже блока `Pass` новый блок действий, используя вкладку **Action Boxes** диалогового окна **Object Properties**, чтобы ввести следующие действия:

   ```
   wait_clock(4) ;
   stop=0;
   ```

   Измените имя блока на `Continue` и, используя пиктограмму **Add Wait Box**, добавьте ниже блок `wait`. Дважды щелкните по блоку, чтобы отобразить вкладку **Wait Boxes** диалогового окна **Object Properties**.

   Переименуйте блок на `Wait1` и определите условие ожидания:

   ```
   wait ((high==4'd0)&&(low==4'd0));
   ```

   Добавьте блок действий ниже ждущего блока. Переименуйте его на `Delay` и введите следующее действие:

   ```
   wait_clock(4) ;
   ```

9. Копирование дерева выбора.

   Выберите блок `Check_Suspend`, плюс `Pass` и `Fail`, используйте пиктограмму **Copy** (или **Edit** | **Copy**).

   Щелкните левой кнопкой мыши ниже блока `Delay` и используйте **Paste**, чтобы вставить копию объектов из буфера обмена. Обратите внимание, что каждому из вставленных объектов автоматически дается уникальное имя.

   Используйте диалоговое окно **Object Properties**, чтобы изменить имена, условия и действия (табл. 29.2).

Таблица 29.2. *Имена блоков и действия*

Имя	**Действие**
`Check_Alarm`	`alarm==1`
`Alarm_Pass`	`$display ("Alarm asserted correctly");`
`Alarm_Fail`	`$display ("No alarm");`

Добавьте блоки конца на одну и вторую ветку.

Используйте **File** | **Open Up**, чтобы отобразить исходный алгоритм, и добавьте блок действий ниже иерархического блока. Откройте диалоговое окно **Object Properties**, чтобы изменить имя этого блока на `Clear`, и введите следующие действия:

```
stop=1;
wait_clock(2);
stop=0;
wait_clock(4) ;
$display ("Timer test completed");
$stop;
```

Эти действия выполняются, когда проведение испытаний таймера завершено успешно, а системная задача `$stop` используется, чтобы остановить симулятор.

Добавьте конечный блок и сохраните алгоритм.

10. Генерация HDL для Test Bench.

    В редакторе алгоритмов откройте окно **Flow Chart Properties** из меню **Diagram**.

    Выберите вкладку **Generation** и проверьте, что установлена опция **Combinatorial**. Установите в **Block Type** опции **initial** и **begin/end**.

    Выберите модуль `Timer_tb` в браузере проекта и используйте **HDL** | **Generate** | **Hierarchy Through Components** для начала генерации HDL-кода.

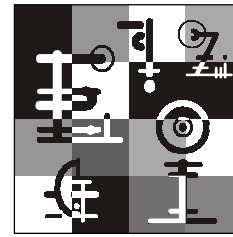
    Ход HDL-генерации отображается в окне **Log Window**.

Вызов симулятора Model Sim

Симулятор может быть вызван из браузера проекта, редакторов блок-схемы или конечного автомата. Однако он может быть вызван на уровне, содержащем полную ветвь проекта, которую вы хотите симулировать.

1. Выберите модуль `Timer _tb` в браузере проекта и используйте пункт меню **HDL** | **Start Simulator**.

2. Для того чтобы добавить сигналы на временную диаграмму, щелкните правой кнопкой мыши на `Timer_tb` в окне **Workspace**, выберите вкладку **Sim** и нажмите **Add** | **Add to Wave**.
3. Откроется окно **Wave** с сигналами.
4. Запустить симуляцию можно с помощью пиктограммы **Run-All**.
5. Исследуйте результаты, отображенные в окне **Wave**, и проверьте, что симуляция выполнена верно, то есть результаты симуляции совпадают со спецификацией проекта "Таймер".



Глава 30

Синтез логических схем. Получение файлов для конфигурирования ПЛИС

Синтез логической схемы — это процесс построения схемы из заданных логических элементов по заданным функциям.

Логические элементы, из которых строится схема, называются *базисными* (библиотечными) элементами и в совокупности образуют *базис* (библиотеку) синтеза. Функция, реализуемая библиотечным элементом, называется библиотечной функцией.

Получение файлов для конфигурирования ПЛИС рассматривается на примере системы автоматического синтеза Leonardo Spectrum.

В системе Leonardo задание базиса сводится либо к указанию типа ПЛИС, в которую будет "вкладываться" синтезируемая схема, либо к указанию имени библиотеки, если синтез ведется для заказных СБИС. Библиотечными базисными элементами должны быть одно- или двухвходовые комбинационные логические элементы и триггеры.

Основными критериями оптимизации при решении задачи синтеза является сложность (стоимость) схемы и ее быстродействие. Обычно стремятся найти схему, характеризуемую либо наименьшей сложностью, либо наибольшим быстродействием (наименьшей задержкой), или достигается определенный компромисс между двумя этими параметрами.

Если проектирование в Leonardo ведется для указанной проектировщиком ПЛИС, то полученное в результате работы синтезатора структурное описание схемы должно быть передано в соответствующую САПР, которая позволяет создать файл программирования PLD либо создать файл для загрузки конфигурации — для FPGA.

В системе Leonardo имеются внутренние описания целевых библиотек для каждого предлагаемого на выбор типа кристалла PLD либо FPGA.

Основными шагами синтеза являются создание проекта и установка опций синтеза. Проект в системе Leonardo, так же как и в системах моделирования, — это совокупность исходных HDL-описаний, требуемых пакетов, библиотек.

30.1. Интерфейс системы Leonardo Spectrum. Элемент управления FlowTabs

Элемент управления FlowTabs позволяет по шагам создать проект, установить все требуемые настройки и ограничения на него. FlowTabs состоит из нескольких вкладок, которые в свою очередь могут иметь внутренние вкладки. Заполняя по очереди поля вкладок, разработчик проходит все этапы от создания проекта до получения синтезированной схемы проекта. Далее рассматривается структура элемента управления FlowTabs.

30.1.1. Загрузка библиотеки технологий (вкладка *Technology*)

Загрузка технологической библиотеки — первый шаг в процессе синтеза. Среди различных типов кристаллов PLD и FPGA, выпускаемых разными фирмами, можно выбрать кристалл требуемого типа.

Внутренние вкладки:

- **Devise** — выбор микросхемы нужного семейства;
- **Speed** — это степень быстродействия выбранной микросхемы. Leonardo Spectrum выбирает значение автоматически, или его можно выбрать из списка;
- **Max Fanout** — поле **Max Fanout** используется для задания максимального числа ветвлений на выходе (нагрузочная способность). Однако большое число ветвлений может стать проблемой для утилиты трассировки и размещения и может стать причиной существенных временных задержек. Заданные по умолчанию пределы ветвления на выходе получены из библиотеки синтеза;
- **Lock Lcells** (блокировка логической ячейки) — делает доступным список **Exclude Gates** (Исключить логические элементы) и дает возможность выбирать необходимые логические элементы. Выбранные логические элементы исключаются из библиотеки. Кроме того, исключенные логические элементы не сохраняются как часть проекта;
- **Map Cascades** — по умолчанию эта опция активна. Leonardo Spectrum отображает синтезируемую логическую схему каскадом логических элементов, но только там, где это возможно.

30.1.2. Чтение проекта (вкладка *Input*)

Последовательность действий:

1. Установить рабочий каталог.

   Определяет место, куда помещаются все сгенерированные выходные файлы.

2. Открыть исходные файлы.

   Файлы могут быть открыты из любого каталога.

3. Нажать **Read**, чтобы добавить файлы проекта в базу данных.
4. Вкладка **Input** имеет описанные далее внутренние вкладки.

Вкладка *VHDL Options*

Top Entity — определяет верхний уровень иерархии проекта. Можно определить имя объекта верхнего уровня.

Architecture — определяет имя архитектуры (только строчные буквы). Если используется с опцией **Entity**, определяет верхний уровень иерархии проекта.

Generic (data_width=5) — устанавливает значение для указанного параметра в формате

```
<generic>=<value> [<generic>=<value>...]
```

Вкладка *Verilog Options*

Используется, когда входной проект находится в формате Verilog. Leonardo Spectrum автоматически выбирает главный модуль, но его можно определить вручную в поле **Top Module**.

Parameters — поле, позволяющее устанавливать значения для параметров. Можно определить несколько параметров:

```
<parameter>=<value>[<parameter>=<value>...]
```

Full Case — директива полного синтеза. По умолчанию задано — не использовать, предотвращает создание дополнительных триггеров.

Parallel Case — это директива параллельного синтеза. В случае, когда состояния являются взаимоисключающими, будет создан мультиплексор.

30.1.3. Установки синхронизации (вкладка *Constraints*)

Эта вкладка в свою очередь состоит из нескольких вкладок.

Вкладка *Global*

Specify Clock Frequency, (Mhz) — определяет требуемую частоту для проекта.

Specify Clock Period, (ns) — определяет требуемый период.

Кнопка **Specify Maximum Delay Between all** дает возможность установить время задержки при передаче информации от порта к регистру (**port to register**), от регистра к регистру (**register to register**), от регистра к порту (**register to port**) и от порта к порту (**port to port**) в наносекундах.

Вкладка *Clock*

Reference Clock Properties (Свойства синхро-сигналов):

- **Frequency** — частота (МГц);
- **Период** — период;
- **Offset, ns** — это время задержки, абсолютное время;
- **Pulse Width, ns** — смещение — длительность импульса (продолжительность импульса в наносекундах).

Вкладка *Input*

Arrival Time (ns) — время поступления, которое определяет максимальную задержку во внешней логике по отношению к синтезируемому проекту.

Input Drive — определяет дополнительное время задержки для выбранного входа.

Max Transition, ns — фронт — время, требуемое для перехода импульса от низкого уровня к высокому.

Max Transition, ns — спад — время, требуемое для перехода импульса от высокого уровня к низкому уровню.

30.1.4. Оптимизация проекта (вкладка *Optimize*)

Run type — тип синтеза, может иметь два значения:

- **Optimize** (значение по умолчанию) — выполняет множественные оптимизационные проходы. Оптимизация средствами уменьшения и улучшения логической схемы проекта по используемым ресурсам и времени задержки;
- **Remap** — оптимизация не выполняется.

Extended Optimization Effort — если это поле выбрано, то Leonardo Spectrum выполняет дополнительные три алгоритма оптимизации. Выбор дополнительных методов оптимизации может вызвать более медленное выполнение оптимизации.

Optimize for — режим оптимизации:

- **Delay** — минимизируется задержка распространения сигнала за счет увеличения физических ресурсов;
- **Area** — минимизация ресурсов, используемых проектом.

30.1.5. Сохранение полученных результатов

Результаты синтеза и оптимизации должны быть сохранены в одном из стандартных форматов, описывающих схему проекта. В таком виде данные могут быть прочитаны любой САПР, поддерживающей стандартные форматы файлов структуры.

Таким образом, результаты синтеза могут быть переданы, например, для верификации в подсистему Model Sim или для выполнения функций размещения и трассировки в соответствующие САПР.

Сохранение данных выполняется с помощью кнопки **Write** (Запись).

В поле **Format** имеется возможность выбрать один из выходных форматов:

- **Auto** — автоматический выбор формата (в зависимости от выбранной целевой библиотеки);
- **VHDL** — сохранение данных в структурном стиле языка;
- **Verilog** — сохранение данных в формате Verilog;
- **XNF, NCF** — сохранение данных в формате САПР Xilinx Foundation Series;
- **EDIF** — сохранение данных в наиболее распространенном структурном формате;
- **SDF** — сохранение информации о задержках в структурных компонентах в специализированный файл sdf;
- **XDB** — сохранение файла внутреннего формата Leonardo Spectrum для сохранения как проекта, так и всех его настроек.

Поле **Write vendor constraints file** служит для включения режима создания настроечного файла, используемого в САПР конкретного производителя, параллельно с файлом схемы. Для проектов на ПЛИС это поле должно быть отмечено.

Поле **Pre-process netlist** служит для включения режима записи, при котором выполняются все необходимые преобразования имен цепей, компонентов и т. п. Для проектов на ПЛИС это поле должно быть отмечено.

Поле **Write only the top level of hierarchy to file** позволяет записать в выходной файл только самый верхний уровень иерархии.

Поле **Technology Cells** обеспечивает запись в файл только структурной схемы, включающей библиотечные элементы, чего достаточно для передачи данных в подсистемы размещения и трассировки.

Поле **Primitives** позволяет получить в выходном файле помимо структурного описания поведенческие фрагменты, обеспечивающие возможность моделирования полученного файла без привлечения библиотек элементов.

30.2. Средство анализа схемы проекта Leonardo Insight

Leonardo Insight — мощное средство анализа схемы проекта. Оно позволяет получить графическое отображение схемы устройства.

Имеется возможность получить две разновидности схем.

- Первый вид изображает логическую схему устройства на уровне регистровых передач (RTL — Register Transfer Level), не выраженную библиотечными элементами целевой технологии. Эта схема подготавливается на этапе предварительной оптимизации при считывании файлов задания. Для получения схемы необходимо использовать кнопку **View RTL Schematic**.
- Второй вид выражен библиотечными элементами целевой библиотеки и отображает результаты синтеза и оптимизации. Для получения этого вида схемы необходимо воспользоваться кнопкой **View Technology Schematic**.

Лабораторная работа. Синтез проектов "Таймер" и устройств ЦТ

Цель работы

Данная работа предназначена для ознакомления с системой синтеза логических схем Leonardo Spectrum на примере проекта "Таймер".

Программа работы

Работа с проектом ведется в оболочке Design Browser.

1. Запуск инструмента синтеза.

   Для синтеза необходимо выбрать модуль `Timer` в браузере проекта и использовать пиктограмму **Synthesis Flow**.

   Проект будет повторно сгенерирован, в случае необходимости, и откомпилирован. Диалоговое окно **Invoke Settings** отображается, для подтверждения или изменения параметров синтеза. Здесь вы можете выбрать технологию, которую будете использовать.

   Поля **Device**, **Speed Grade** и **Wire Table** выбираются автоматически, при выборе технологии. Выберите **FPGA/CPLD | Actel | PA**, в поле **Device — apa075pqfp208**. Завершите редактирование параметров, введя частоту синхросигнала 40 МГц.

   После подтверждения выбранных настроек запустится Leonardo Spectrum и автоматически начнется процесс синтеза.

2. Просмотр отчетов (Report).

   Нажмите кнопку **Report Area** на вкладке **Output**, в окне **Transcript Window** будет выведена следующая информация:

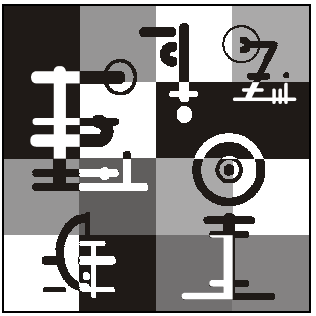
```
*****************************************************
*****************************************************
Cell: Timer    View: INTERFACE    Library: timer_vlog
Cell: Timer    View: INTERFACE    Library: timer_vlog
*****************************************************
*****************************************************
 Cell       Library  References     Total Area
 Cell       Library  References     Total Area
AO21FTF     apa     6 x      1      6 Core Cells
AO21FTT     apa     2 x      1      2 Core Cells

   •           •           •

OR3FFT      apa     8 x      1      8 Core Cells
OR3FTT      apa     9 x      1      9 Core Cells
XNOR2       apa     6 x      1      6 Core Cells
 Number of ports :                      23
 Number of nets :                      167
 Number of instances :                 153
 Number of references to this view :     0

Total accumulated area :
 Number of Core Cells :                128
 Number of GND :                         2
 Number of IB33 :                       14
 Number of OB33PH :                      9
 Number of accumulated instances :     153
*********************************************
Device Utilization for apa150pqfp208
*********************************************
Resource                Used    Avail   Utilization
-----------------------------------------------
```

```
IOs                     23      158      14.56%
Core Cells              128     6144      2.08%
```

Здесь показано, какое количество ячеек кристалла будет задействовано, а также процентное отношение использованных ресурсов, в данном случае — 14.56% портов ввода/вывода и 2.08% технологических ячеек, от общего числа.

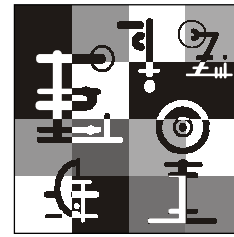
3. Просмотр графического отображения схем проекта.

   Используйте пиктограмму **View RTL Schematic** на инструментальной панели, чтобы отобразить схему RTL (уровень регистровых передач), или пиктограмму **View Technology Schematic**, чтобы отобразить технологическую схему проекта.



Часть VII

Средства проектирования фирмы Actel



Глава 31

Libero IDE. Возможные методы проектирования

Libero IDE (Integrated Development Environment — интегрированная среда разработки) — всестороннее и мощное средство проектирования FPGA и доступные утилиты, обеспечивающие весь цикл проектирования. Libero IDE комбинирует инструменты Actel с такими САПР, как Synplify, Model Sim, ViewDraw, WaveFormer Lite и Silicon Explorer.

С помощью Libero IDE возможно программирование в такие ПЛИС, как ProASIC^{PLUS} и Axcelerator.

Траектория проектирования с помощью Libero IDE состоит из следующих шагов.

1. Создание нового или открытие существующего проекта.
2. Создание исходных файлов (схема, HDL, ACTgen Macros — инструмент автоматического генерирования макросов).
3. Импорт исходных файлов в проект.
4. Синтез проекта с помощью Synplify (только HDL-проекты).
5. Выполнение функциональной симуляции с помощью Model Sim.
6. Размещение и трассировка проекта с помощью Designer.
7. Выполнение временной симуляции с помощью Model Sim.
8. Программирование ПЛИС.
9. Тестирование готового устройства.

31.1. Инструменты Libero

Libero Project Manager интегрирует средства проектирования, упрощает разработку, управляет проектированием и журналами, передает данные проекта между различными инструментами (табл. 31.1).

Таблица 31.1. *Состав и назначение инструментов Libero IDE*

Функция	Инструмент	Производитель
Design Entry (HDL)	HDL Editor	Actel
Design Entry (Schematic)	ViewDraw AE	Actel
Synthesis	Synplify	Synplicity
Simulation	ModelSim	Mentor Graphics
Automatic Testbench Generator	WaveFormer Lite	SynaptiCAD
Automatic Macro Generator	ACTgen	Actel
Place-and-Route	Designer	Actel
Programming Software	FlashPro	Actel
In-Silicon Debug	Silicon Explorer II	Actel

Рассмотрим подробнее инструменты, используемые в рекомендуемой фирмой Actel траектории проектирования.

- HDL Editor — редактор HDL, предназначен для создания HDL-кода. HDL Editor поддерживает синтаксис VHDL и Verilog HDL. В дополнение к основным функциям, редактор обеспечивает проверку синтаксиса.
- ViewDraw AE — инструмент ввода графических схем.
- Synplify — инструмент синтеза логических схем.
- Model Sim от Mentor Graphics — средство моделирования (рассмотрено в *главе 29*).
- WaveFormer Lite от SynaptiCAD позволяет создавать испытательные стенды HDL (Test Benches) в графическом редакторе и управляет многочисленными испытательными стендами, необходимыми для различных конфигураций проекта. Генератор испытательных стендов по графическим данным идеален для проектировщиков, незнакомых с процессом создания испытательных стендов, или для экспертов желающих экономить время.
- FlashPro — средство программирования ПЛИС фирмы Actel.
- Designer от Actel обеспечивает удобный интерфейс для реализации проекта в ПЛИС.

Основные этапы цикла работы Designer.

1. Компилирование проекта.
2. Запуск размещения и трассировки проекта.

3. Обратное аннотирование проекта.
4. Генерация файла программирования (Fuse или Bitstream).
5. Программирование устройства.

Designer включает следующие компоненты:

- PinEditor — приложение для отображения и конфигурирования назначения и свойств вводов/выводов;
- ChipEditor — графическое приложение просмотра и размещения вводов/выводов и логических ячеек ПЛИС. Особенно полезен, когда проектировщик нуждается в максимальном контроле над размещением проекта;
- NetlistViewer — средство просмотра схем проекта. Читает Netlist (список соединений) проекта *(см. далее)* и генерирует схематическое представление;
- SmartPower — инструмент анализа напряжений;
- Timer — статический временной анализ и редактор ограничений.

31.2. Управление проектом в Libero

Рекомендуемая фирмой Actel траектория проектирования представлена на рис. 31.1.

Проектирование состоит из шести шагов.

1. Создание проекта.

   Проект можно вводить как HDL-описание (VHDL или Verilog HDL) или в виде структурной схемы, или в смешанном виде (схема и RTL).

2. Функциональная верификация (симуляция).

   После описания проекта надо проверить его функционирование. С помощью WaveFormer Lite создается Test Bench. Для верификации используется Model Sim или VHDL- или Verilog HDL-симулятор.

3. Синтез.

   Проект должен быть синтезирован, если он был создан, используя VHDL- или Verilog HDL-описания. С помощью программы Synplify описание на языках HDL переводится в структурное иерархическое описание схемы из библиотечных макроэлементов (логические элементы ПЛИС). Результаты синтеза представляются в формате EDIF (Electronic Data Interchange Format — формат электронного обмена данными), полученный файл называют Netlist — список соединений.

4. Размещение и трассировка проекта.

   После функциональной верификации проекта следующий шаг — это реализация проекта в ПЛИС с использованием Designer. Программное обеспечение Designer

автоматически выполняет размещение и трассировку проекта и возвращает информацию о временных задержках (файл с расширением sdf). Можно использовать инструменты, которые входят в Designer, для дальнейшей оптимизации:

- Timer — для проведения статического временного анализа;
- PinEditor — для настройки входов-выходов;
- SmartPower — для анализа напряжений;
- NetlistViewer — для просмотра схем.

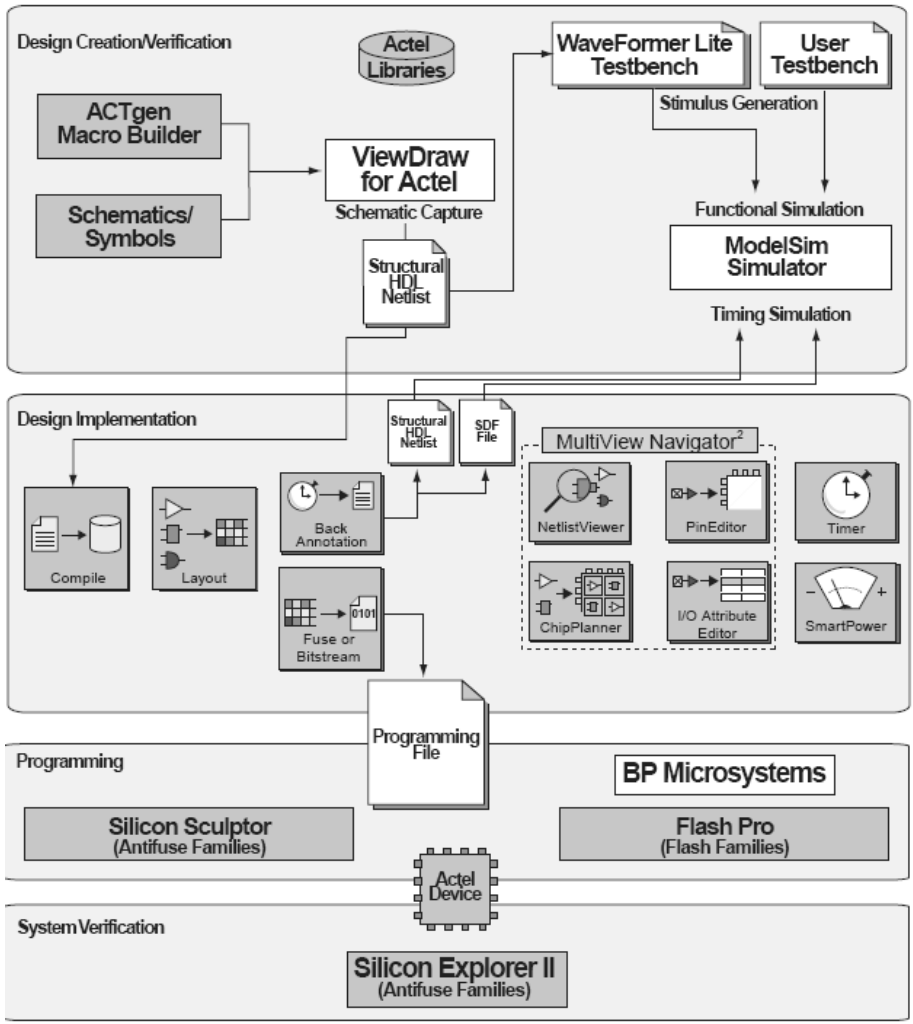


Рис. 31.1. Траектория проектирования

5. Временная симуляция.

   После размещения и трассировки проекта в ПЛИС проверяется его соответствие временным техническим требованиям. Для временной симуляции используется средство синтеза Model Sim.

6. Программирование микросхемы.

   Как только завершена разработка и проведена временная симуляция, создается файл прошивки. В зависимости от семейства микросхемы *(см. главу 33)*, генерируются файлы Fuse (для ПЛИС, выполненных по технологии Antifuse) или Bitstream (для ПЛИС, выполненных по технологии Flash).

Лабораторная работа. Знакомство на примерах со средством проектирования Libero IDE фирмы Actel

Цель работы

Данная работа предназначена для ознакомления с системой проектирования Libero IDE на простейшем примере: реализация логической операции "И".

Программа работы

Проект будет выполняться по следующей траектории:

1. Создание нового проекта.
2. Симуляция перед синтезом.
3. Синтез проекта в Synplify.
4. Симуляция после синтеза.
5. Реализация проекта с помощью Designer.
6. Временная симуляция с помощью утилиты обратной аннотации времени.
7. Генерация файла прошивки.
8. Прошивка микросхемы.

Примечание

В данной лабораторной работе конечным пунктом проектирования будет временная симуляция проекта.

Шаг 1 — создание нового проекта

В меню **File** выберите **New Project**. В поле **Project Name** введите `example`, выберите **PA** (семейство микросхем ProASIC^{PLUS}) в поле **Family** (Семейство ПЛИС), **HDL Type** в поле **VHDL**. Нажмите **Finish**, проект "example" создастся и откроется.

В меню **File** выберите **New**, откроется диалоговое окно **New**. Выберите **VHDL Entity** (Модуль VHDL) в поле **File Type** (Тип файлов) и введите имя `andgate`.

Скопируйте следующий текст в созданный файл и сохраните его.

```
-- AND Gate Tutorial for APA Evaluation Board
LIBRARY ieee;
USE ieee.std_logic_1164. ALL;
ENTITY andgate is
port (A, B : in std_logic;-- Data Inputs
OUTPUT : out std_logic); -- Output= A AND B
end andgate;
architecture behaviour of andgate is
begin
OUTPUT <= A AND B;
end behaviour;
```

В окне **Design Explorer** на вкладках **Design Hierarchy** и **File Manager** появился файл andgate.vhd. Щелкните правой кнопкой мыши на этом файле и выберите **Check HDL** (проверка HDL-кода) — будет проведена проверка на ошибки. Если ошибки имеются, сообщение о них будет выведено в окне **Log Window** снизу.

Шаг 2 — выполнение симуляции перед синтезом

Создание Test Bench, используя WaveFormer Lite

Создание Test Bench с помощью WaveFormer Lite проводится в три этапа:

1. Импорт информационных сигналов.

   Запустите **WaveFormer Lite**, дважды щелкнув по его обозначению в окне **Process Window** или щелкнув правой кнопкой мыши по файлу andgate.vhd, выберите **Create Stimulus** (Создать стимул (входные сигналы)). WaveFormer Lite запустится с сигнальными портами в окне **Diagram**.

2. Рисование временной диаграммы.

   Для рисования используйте кнопки режимов на верхней панели:

Текущий режим обозначается красным цветом, следующий красным треугольником сверху.

Длительность периодов можно контролировать частотой сетки **Options | Grid Settings** (Опции | Настройки сетки).

Также можно выполнять копирование и вставку секций временных диаграмм, перезапись или вставку сигналов в диаграмму. Для копирования и вставки секций временной диаграммы нужно:

- выбрать названия требуемых сигналов. Если ни один сигнал не выбран, командой **Block Copy** (Копировать блок) будет произведено копирование всех сигналов на схеме;
- выбрать **Edit | Block Copy Waveforms**. Откроется диалоговое окно **Block Copy Waveforms** с выбранными сигналами, отображенными в списке **Change Waveform Destination**;
- в диалоге введите значения, которые определяют копирование и вставку.

3. Экспорт Test Bench.

На этом шаге создается файл временных диаграмм определением значения входных сигналов `A`, `B` и генерируется Test Bench.

Создайте во времени формы сигналов `A` и `B` в соответствии с табл. 31.2.

Таблица 31.2. *Значения входных сигналов*

A	Низкий уровень 0 ns —100 ns
	Высокий уровень 100 ns — 1 us
B	Низкий уровень 0 ns — 300 ns
	Высокий уровень 300 ns — 330 ns
	Низкий уровень 330 ns — 1 us

После успешного создания временной диаграммы выберите **Save As** (Сохранить как) в меню **File**. В диалоговом окне **Save As** введите `andgate_stim.btim` как имя файла и нажмите кнопку **Save**.

После сохранения файла временной диаграммы выберите **Export Timing Diagram As** в меню **Export**.

Выберите **VHDL Top Level Test Bench (*vhd)** в поле **Files of Type** и введите `andgate_stim.vhd` для имени файла.

Выйдите из WaveFormer Lite (**File | Exit**). В Libero IDE на вкладке **File Manager** отображается файл диаграммы. Проект готов для симуляции в Model Sim.

Симуляция перед синтезом

1. Выберите файл стимула. Щелкните правой кнопкой по схеме `andgate` в окне **Design Hierarchy** и выберите **Select Stimulus**.
2. Выберите **andgate_stim.vhd** в **Project List** и нажмите **Add**, чтобы добавить файл к списку **Associated Files**.
3. Щелкните **OK**. Рядом с Waveformer Lite в окне **Process window** появится зеленая галочка.
4. Дважды щелкните по иконке **ModelSim** в окне **Process** или щелкните правой кнопкой мыши на **andgate** на вкладке **Design Hierarchy** и выберите **Run Pre-synthesis Simulation** (Запустить симуляцию перед синтезом). Model Sim откроет и скомпилирует исходные файлы.
5. Как только компиляция завершается, производится симуляция для заданного по умолчанию значения времени 1000 ns, и в окне **Wave** отображаются результаты симуляции.
6. Для выхода из программы симуляции в окне **Modelsim** выберите **File** | **Quit**.

Шаг 3 — синтез проекта в *Synplify*

На следующем шаге генерируется EDIF Netlist, после синтеза проекта в Synplify. Для HDL-проектов интегрированная среда разработки Libero автоматически запускает Synplify-синтезатор с соответствующими файлами проекта.

Создание EDIF Netlist для проекта, используя Synplify

1. В Libero IDE дважды щелкните значок **Synplify Synthesis** или щелкните правой кнопкой мыши файл **andgate** на вкладке **Design Hierarchy** и выберите **Synthesize**. Запустится Synplify с соответствующими файлами проекта.
2. В меню **Project** выберите **Implementation Options** (Опции реализации). Установите следующие параметры:
   - **Technology**: **Actel PA** (этот параметр уже был установлен при создании проекта в Libero IDE);
   - **Part**: **APA075**;
   - **Fanout Guide**: **12** (по умолчанию);
   - **Hard Limit to Fanout**; **Off** (выключен, по умолчанию). Это предел разветвления на выходе.

В основном окне нажмите **Run**. Synplify скомпилирует и синтезирует проект в Netlist (файл andgate.edn). Затем файл andgate.edn будет автоматически оттранслирован

Libero в VHDL Netlist (файл andgate.vhd). Результирующие EDIF- и VHDL-файлы отображаются под заголовком **Implementation Files** на вкладке **File Manager** интегрированной среды разработки Libero.

Сохраните проект и закройте Synplify.

Шаг 4 — размещение и трассировка

1. Дважды щелкните значок **Designer Place and Route** (Размещение и трассировка проекта с помощью инструмента Designer) в окне **Process** или щелкните правой кнопкой мыши на **andgate** в окне **Design Hierarchy** и выберите **Run Designer**. Откроется приложение Designer.
2. Компиляция проекта.

   В меню **Tools** выберите **Device Selection** (Выбор микросхемы). Выберите **APA075** в поле **Die** и **208 PQFP** в поле **Package**. Примите заданную по умолчанию степень быстродействия и напряжение кристалла и нажмите **Next**. Дважды щелкните значок **Compile**. Designer компилирует проект и показывает процент использования ресурсов выбранной микросхемы. Значок **Compile** меняет цвет индикации на зеленый, что показывает — компиляция завершена успешно.
3. Назначение сигналов проекта выводам микросхемы.

   Для этого используется утилита PinEditor — щелкните на кнопке **PinEditor**. Задайте расположение выводов в соответствии с табл. 31.3.

   Таблица 31.3. *Соответствие сигналов выводам микросхемы*

Signal (Имя сигнала)	Direction (Направление сигнала)	PIN (Вывод микросхемы)
A	Input	55
B	Input	63
OUTPUT	Output	87

   Как только назначены все сигналы, выберите **Commit** (Фиксировать) в меню **File** и закройте окно **PinEditor**.
4. Размещение проекта.

   Нажмите значок **Layout** (Размещение), откроется диалоговое окно **Layout Options**. Нажмите **OK**, чтобы принять заданные по умолчанию параметры размещения. После этого выполнится размещение и трассировка проекта.
5. Обратная аннотация проекта.

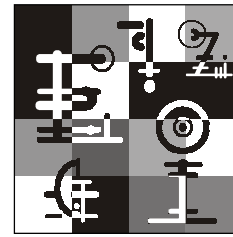
   При обратной аннотации получается файл, содержащий информацию о временных задержках, который генерируется по результатам размещения проекта

в ПЛИС. Эта информация необходима для временной симуляции. В Designer нажмите значок **Back Annotate** (Обратная аннотация). Примите настройки по умолчанию. Значок **Back Annotate** станет зеленым.

Шаг 5 — временная симуляция

После завершения размещения и обратной аннотации проекта, выполняется симуляция проекта с учетом временных задержек с помощью Model Sim.

Для проведения временной симуляции следует щелкнуть иконку **ModelSim Simulation** в интегрированной среде разработки Libero (окно **Process**) или щелкнуть правой кнопкой мыши файл **andgate** в окне **Design Hierarchy** и выбрать **Run Post-Layout Simulation**. Запустится ModelSim, который компилирует обратно-аннотированный VHDL Netlist и Test Bench. Как только компиляция завершится, выполнится симуляция для 1000 ns, и откроется окно **Wave** для отображения результатов. Убедитесь, что симуляция выполняется верно.



Глава 32

Стартовый комплект для начала работы с ПЛИС Actel

Комплект ProASIC^{PLUS} Starter Kit позволяет ознакомиться с особенностями работы с ПЛИС фирмы Actel.

В комплект ProASIC^{PLUS} Starter Kit входят:

- отладочная плата с микросхемой APA075;
- программный пакет Libero;
- программатор FlashPro Lite с соединительным кабелем.

32.1. Макетная плата ProASIC^{PLUS} Evaluation Board

Плата включает (рис. 32.1):

- разъем для подачи напряжения питания, с выключателем и светодиодным индикатором;
- перемычку для выбора напряжения логики входа/выхода 2,5 и 3,3 В;
- 40 МГц генератор тактовых импульсов;
- 8 светодиодов, управляемых выводами микросхемы;
- перемычки, позволяющие произвести разъединение всей внешней электрической схемы от ПЛИС;
- 4 кнопки, обеспечивающие управление входами микросхемы.

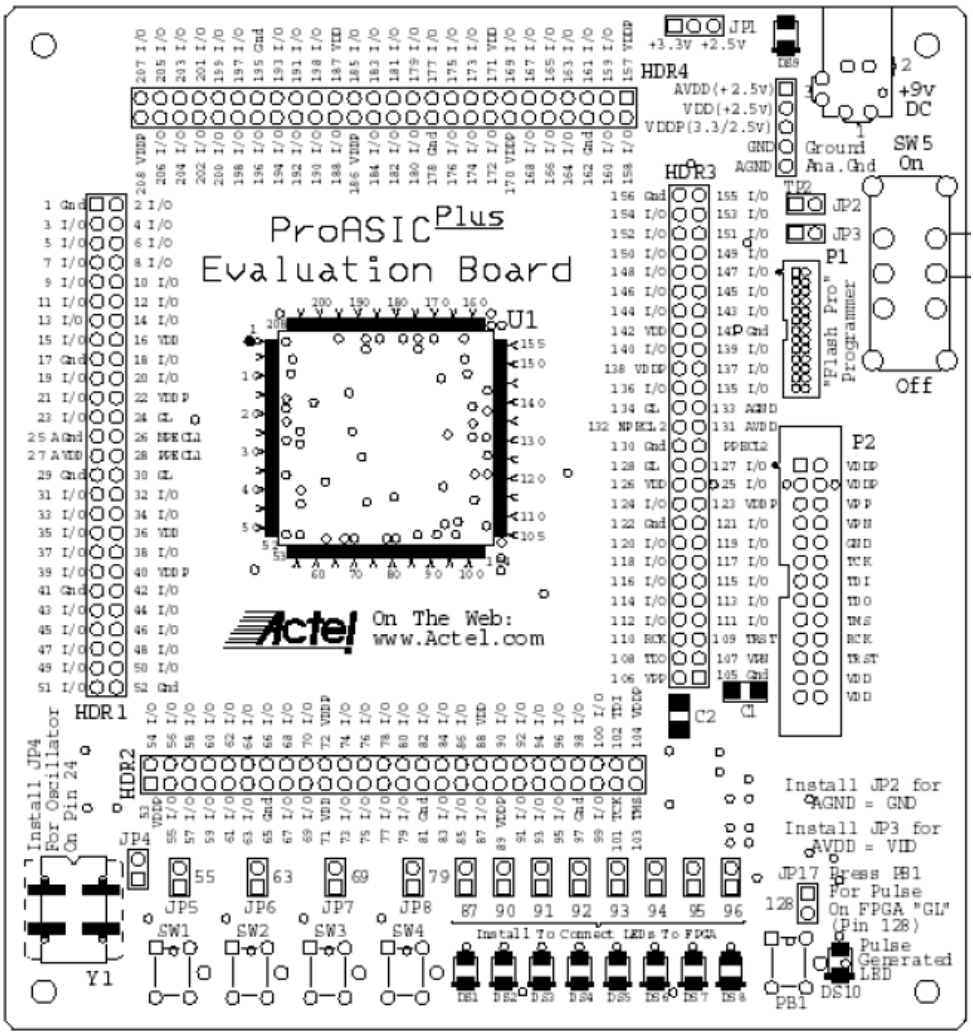


Рис. 32.1. Макетная плата ProASIC^{PLUS} Evaluation Board

32.1.1. Источник питания

Напряжение питания подается на плату от внешнего блока питания. Управление подачей питания осуществляется с помощью переключателя "Вкл/Выкл".

Светодиод DS9 указывает, что внешний блок питания подсоединен к плате.

Перемычка JP1 используется, чтобы выбрать напряжение 3,3 или 2,5 В для выводов микросхемы.

32.1.2. Контакты программатора

Плата снабжена маленькой контактной площадкой, подходящей для программаторов FlashPro/FlashPro Lite и Silicon Sculptor II.

32.1.3. Схема синхронизации

Плата имеет две схемы синхронизации, 40 МГц генератор и генератор с ручным управлением.

- 40 МГц генератор на плате связан с перемычкой JP4. JP4 подключает генератор к ножке 24 микросхемы. Ножка 24 — глобальный вход синхронизации.
- Генератор с ручным управлением. Нажатие на кнопку PB1 инициирует генерацию синхроимпульса, при этом зажигается светодиод DS10. Активизация генератора производится установкой перемычки JP17. JP17 соединяет генератор с 128-ой ножкой микросхемы.

32.1.4. Подключение светодиодов

Восемь светодиодов связаны с микросхемой перемычками (табл. 32.1). Если перемычки поставлены, выводы микросхемы могут управлять светодиодами. Изменение состояния светодиодов происходит следующим образом:

- '1' на выводе микросхемы зажигает светодиод;
- '0' на выводе микросхемы выключает светодиод;
- незапрограммированный или неопределенный вывод может слабо зажечь светодиод.

Таблица 32.1. *Соответствие диодов ножкам микросхемы*

Обозначение светодиода	Вывод микросхемы
DS1	87
DS2	90
DS3	91
DS4	92
DS5	93
DS6	94
DS7	95
DS8	96

32.1.5. Подключение кнопок

Четыре кнопки подключаются к микросхеме с помощью перемычек (в соответствии с табл. 32.2). Если перемычки установлены, то с помощью кнопок можно управлять выводами микросхемы:

- нажатие на кнопку подает '1' на микросхему. '1' продолжает поступать, пока нажата кнопка;
- отпускание кнопки подает на микросхему "0".

Таблица 32.2. *Соответствие кнопок ножкам микросхемы*

Обозначение кнопки	Вывод микросхемы
SW1	55
SW1	63
SW1	69
SW1	79

Лабораторная работа. Пример проекта для ProASIC^{PLUS} Evaluation Board

Цель работы

В данной работе необходимо в системе Libero IDE провести проектирование до этапа симуляции сначала представленного, затем индивидуального проектов.

Спецификация представленного проекта

В данной плате проект использует средства индикации (8 светодиодов) и средства ввода (4 кнопки).

На диоды выводятся либо значения восьмиразрядного счетчика (который считает как вперед, так и назад), либо значения сдвигающего регистра (также двунаправленного) в зависимости от состояния системы. Состояния задаются с помощью двух левых кнопок платы.

При включении питания устройство находится в состоянии счета в сторону увеличения, состояние счетчика можно отслеживать на 8-ми светодиодах, расположен-

ных на плате. При нажатии на первую кнопку слева светодиоды будут отображать состояние сдвигового регистра.

При нажатии на вторую кнопку слева изменяется направление счета или сдвига.

Описание проекта блок-схемой

Общая схема состоит из следующих блоков, каждый из которых реализован на языке Verilog HDL (рис. 32.2):

- `Divisor` (листинг 32.1) — делитель синхрочастоты (40 МГц). *sec* — импульсы частоты изменения индикации, `buttonclk` — частота опроса кнопок;
- `Filter` (листинг 32.2) — блок фильтрации дребезга контактов кнопок;
- `Timer` (листинг 32.3) — восьмиразрядный двунаправленный счетчик с предустановкой;
- `Shift` (листинг 32.4) — сдвиговый (сдвигающий) регистр;
- `Mux` (листинг 32.5) — мультиплексор, выводит или значение счетчика или сдвигового регистра;
- `Control` (листинг 32.6) — блок управления системой.

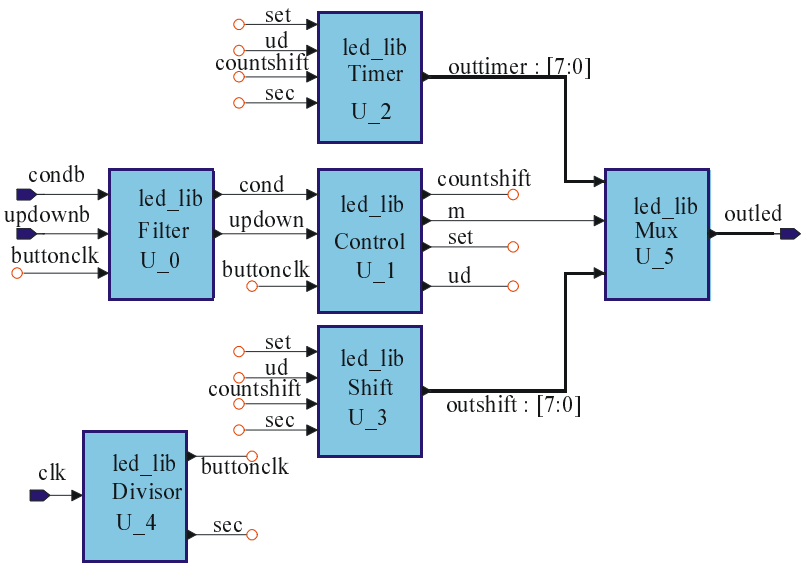


Рис. 32.2. Блок-схема проекта

Листинг 32.1. Divisor

```
module Divisor(
   clk,
   buttonclk,
   sec
);

input   clk;
output  buttonclk;
output  sec;

reg b, s, n;
reg [22:0] y;
reg [4:0] k;

parameter d = 22'b0011010000100100000000; //для процесса симуляции
                                          //d = 22'b01;

      always @ (posedge clk)
      begin
               if ( n == 1 )
               begin
              y = y + 1'b1;
               end
               else
               begin
               y = 0;
               k = 4'b0000;
               n = 1;
               b=0;
         s=0;
               end

               if (y == d)
            begin
               b = ~b;
               k = k + 4'b0001;
               y = 0;
```

```
            end
         if (k==4'b1010)
            begin
               s = ~s;
               k = 4'b0000;
            end
      end
assign buttonclk = b;
assign sec = s;

endmodule
```

Во избежание дребезга контактов необходимо на входе ставить фильтр. Пример такого фильтра для платы ProASIC PLUS Evaluation Board (4 кнопки) показан в листинге 32.2.

Листинг 32.2. Filter

```
`resetall
`timescale 1ns/10ps
module Filter(
   buttonclk,
   condb,
   updownb,
   cond,
   updown
);

input   buttonclk;
input   condb;
input   updownb;
output  cond;
output  updown;

wire buttonclk;
wire condb;
wire updownb;
wire cond;
wire updown;

reg [3:0] shift1, shift2;
```

```
reg b1, b2;

always @ (posedge buttonclk)
begin
  shift1[3:1] = shift1[2:0];
  shift1[0] = condb;
  if ( shift1 == 4'b1111 )
       b1 = 1;
  else
       b1 = 0;
  shift2[3:1] = shift2[2:0];
  shift2[0] = updownb;
  if ( shift2 == 4'b1111 )
       b2 = 1;
  else
       b2 = 0;

end
assign cond = b1;
assign updown = b2;
endmodule
```

Листинг 32.3. Timer

```
`resetall
`timescale 1ns/10ps
module Timer(
   countshift,
   sec,
   set,
   ud,
   outtimer
);

input        countshift;
input        sec;
input        set;
input        ud;
output [7:0] outtimer;

wire countshift;
```

```
wire sec;
wire set;
wire ud;
wire [7:0] outtimer;

reg [7:0]count;

always @ ( posedge sec or posedge set)
begin
        if(set)
              count = 8'b00000000;
        else if(countshift)
              if(~ud)
                     count = count + 1'b1;
              else
                     count = count - 1'b1;
end

assign outtimer = count;

endmodule
```

Листинг 32.4. Shift

```
`resetall
`timescale 1ns/10ps
module Shift(
   countshift,
   sec,
   set,
   ud,
   outshift
);

input        countshift;
input        sec;
input        set;
input        ud;
output [7:0] outshift;

wire countshift;
```

```
wire sec;
wire set;
wire ud;
wire [7:0] outshift;

reg [7:0]sh;

always @ ( posedge sec or posedge set)
begin
        if(set)
              sh = 8'b00000001;
        else if(~countshift)
              if(ud)
                begin
                     if(sh==8'b10000000)
                            sh=8'b00000001;
                     else
                            sh = sh >> 1;
                end
              else
                begin
                     if(sh==8'b00000001)
                            sh=8'b10000000;
                     else
                            sh = sh << 1;
                end

end

assign outshift = sh;

endmodule
```

Листинг 32.5. `Mux`

```
`resetall
`timescale 1ns/10ps
module Mux(
```

```
   m,
   outshift,
   outtimer,
   outled
);

input        m;
input  [7:0] outshift;
input  [7:0] outtimer;
output [7:0] outled;

wire m;
wire [7:0] outshift;
wire [7:0] outtimer;
wire [7:0] outled;

reg [7:0] Result;

  always @(outshift or outtimer or m)
  begin

    case (m)
      0 : Result = outtimer;
      1 : Result = outshift;
      default : Result = 8'bx;
    endcase
  end
assign outled = Result;

endmodule
```

Блок `Control` (листинг 32.6) описывает 4 состояния автомата (рис. 32.3):

- `countup` — счет вперед;
- `countdown` — счет назад;
- `shiftup` — сдвиг влево;
- `shiftdown` — сдвиг вправо.

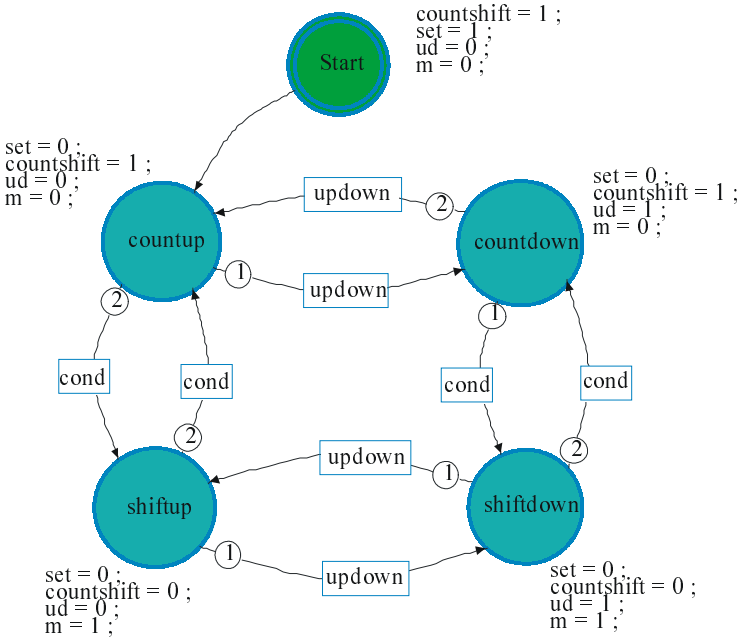


Рис. 32.3. Описание блока `Control` автоматом с 4 состояниями

Листинг 32.6. `Control`

```
`resetall
`timescale 1ns/10ps
module Control(
   buttonclk,
   cond,
   updown,
   countshift,
   m,
   set,
   ud
);



input   buttonclk;
```

```
input   cond;
input   updown;
output  countshift;
output  m;
output  set;
output  ud;


wire buttonclk;
wire cond;
wire updown;
reg countshift;
reg m;
reg set;
reg ud;

parameter
          Start     = 3'd0,
          countup   = 3'd1,
          countdown = 3'd2,
          shiftup   = 3'd3,
          shiftdown = 3'd4;

reg [2:0] current_state, next_state;

//-----------------------------------------------------------------
always @(
   cond or
   current_state or
   updown
)
begin : next_state_block_proc
   case (current_state)
      Start: begin
         if (1)
            next_state = countup;
         else
            next_state = Start;
      end
```

```
      countup: begin
         if (updown)
            next_state = countdown;
         else if (cond)
            next_state = shiftup;
         else
            next_state = countup;
      end
      countdown: begin
         if (cond)
            next_state = shiftdown;
         else if (updown)
            next_state = countup;
         else
            next_state = countdown;
      end
      shiftup: begin
         if (updown)
            next_state = shiftdown;
         else if (cond)
            next_state = countup;
         else
            next_state = shiftup;
      end
      shiftdown: begin
         if (updown)
            next_state = shiftup;
         else if (cond)
            next_state = countdown;
         else
            next_state = shiftdown;
      end
      default:
         next_state = Start;
   endcase
end

//-----------------------------------------------------------------
always @(
```

```
   current_state
)
begin : output_block_proc

   // Combined Actions
   case (current_state)
      Start: begin
         countshift = 1 ;
         set = 1 ;
         ud = 0 ;
         m = 0 ;
      end
      countup: begin
         set = 0 ;
         countshift = 1 ;
         ud = 0 ;
         m = 0 ;
      end
      countdown: begin
         set = 0 ;
         countshift = 1 ;
         ud = 1 ;
         m = 0 ;
      end
      shiftup: begin
         set = 0 ;
         countshift = 0 ;
         ud = 0 ;
         m = 1 ;
      end
      shiftdown: begin
         set = 0 ;
         countshift = 0 ;
         ud = 1 ;
         m = 1 ;
      end
      default: begin
      end
   endcase
```

```
end

//-----------------------------------------------------------------
always @(
   posedge buttonclk
)
begin : clocked_block_proc
   current_state <= next_state;
end

endmodule
```

Задания для самостоятельной работы

Используя описание платы ProASIC^{PLUS} Evaluation Board и пример представленного для этой платы проекта, реализовать индивидуальный проект в соответствии с одним из следующих заданий:

1. АЛУ.

   Реализовать АЛУ (3—4 операции над трехразрядными операндами). Операнды вводить последовательно (сначала один, затем другой) посредством кнопок, расположенных на плате. Также с помощью кнопок выбирать операцию. Отображение хода ввода операндов и вывод результата производить на светодиодах.

2. Регистровая память.

   Реализовать 8 трехразрядных регистров, в которые можно записать и считать из них данные. Ввод осуществлять посредством кнопок, расположенных на плате. Отображение хода ввода и вывод производить на светодиодах. Предусмотреть выбор любого регистра.

3. Делитель частоты.

   Реализовать делитель со ступенчатой регулировкой частоты. Предусмотреть ввод коэффициента деления частоты, также с помощью двух кнопок предусмотреть возможность ступенчатого динамического увеличения и уменьшения этого коэффициента.

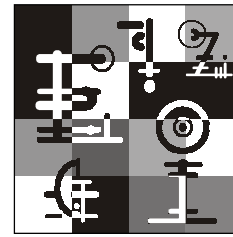
4. "Игральные кости".

   Спроектировать устройство, имитирующее "Игральные кости".

Восьмиразрядное число (с "1" только в одном бите). По сигналу с управляющего входа начинается циклический сдвиг данного числа. Во время сдвига не отображать значение числа на светодиодах. По следующему сигналу на управляющем входе сдвиг останавливается и "выводится" текущее значение. Далее по сигналу происходит сдвиг в другую сторону и т. д. Предусмотреть также ручную установку значения числа.

Программа работы

Используя программу выполнения предыдущей лабораторной работы, провести проектирование до этапа симуляции представленного проекта и индивидуального проекта, реализованного самостоятельно в соответствии с выбранным заданием.



Глава 33

Описание программируемых логических ИС (ПЛИС) фирмы Actel

Рассмотрим ПЛИС фирмы Actel: ПЛИС Antifuse (технология Antifuse, семейство FPGA eX) и ПЛИС серии ProASIC^{PLUS} (технология Flash, семейство ProASIC^{PLUS}).

33.1. Семейство ProASIC^{PLUS}

Семейство ProASIC^{PLUS} — ПЛИС второго поколения. Представляют собой конфигурируемые устройства, выполненные по процессу CMOS 0.22 микрона с 4-мя уровнями металлизации по Flash-технологии. Они являются эффективными устройствами с системными скоростями до 100 МГц и позволяют проектировщикам напрямую подключать ПЛИС к другим устройствам, имеющим входные и выходные сигналы для логики уровней 3,3 и 2,5 В. Микросхема данного семейства содержит два блока обработки синхрочастоты — каждый состоит из ядра PLL (Phase-Locked Loop, система фазовой автоподстройки частоты), линии задержки и умножителя/делителя синхрочастоты. Входные частоты для PLL от 1,5 до 240 МГц, а выходные частоты от 24 МГц до 240 МГц. Кроме того, микросхемы имеют две высокоскоростные дифференциальные пары, работающие в стандарте LVPECL и предназначенные для скоростной передачи данных. Имеется возможность программировать микросхемы непосредственно в системе пользователя (ISP — In-System Programming, внутрисистемное программирование, возможность программирования установленных на плату ПЛИС), эта возможность поддерживается через интерфейс JTAG по стандарту IEEE 1149.1.

ПЛИС данного семейства начинают работать непосредственно при подаче питания — "live-at-power-up".

33.1.1. Характеристики микросхем серии ProASIC^{PLUS}

Микросхемы серии ProASIC^{PLUS} представляют на сегодняшний день семейство, состоящее из 7 устройств, основные характеристики которых приведены в табл. 33.1.

Микросхемы данной серии имеют достаточно большой диапазон по числу системных вентилей — от 75 000 до 1 000 000, встроенной памяти — от 27 до 198 Кбит SRAM, от 66 и до 712 пользовательских выводов.

По корпусам микросхемы совместимы, это позволяет на этапе отладки проекта применить микросхему большей емкости, а после отладки проекта использовать в серийных изделиях микросхему минимальной для данного проекта емкости.

Таблица 33.1. *Основные характеристики микросхем серии ProASIC^{PLUS}*

Устройство	**APA075**	**APA150**	**APA300**	**APA450**	**APA600**	**APA750**	**APA1000**
Максимальное число системных вентилей	75,000	150,000	300,000	450,000	600,000	750,000	1,000,000
Максимальное число регистров	3,072	6,144	8,192	12,288	21,504	32,768	56,320
Встроенная RAM, биты	27k	36k	72k	108k	126k	144k	198k
Встроенная RAM, блоки (256 X 9)	12	16	32	48	56	64	88
LVPECL (Low-Voltage Posi-tive Emitter-Coupled Logic)	2	2	2	2	2	2	2
PLL	2	2	2	2	2	2	2
Глобальные сигналы	4	4	4	4	4	4	4
Максимальное число синхрочастот	24	32	32	48	56	64	88
Максимальное число пользовательских I/Os	158	242	290	344	454	562	712
JTAG ISP	да	да	да	да	да	да	да
PCI	да	да	да	да	да	да	да
Корпус (по числу выводов)							
TQFP	100	100					
PQFP	208	208	208	208	208	208	208
PBGA		456	456	456	456	456	456
FBGA	144	144, 256	144, 256	144, 256, 484	256, 484, 676	676, 896	896, 1152

33.2. Архитектура ProASIC^{PLUS}

Архитектура ProASIC^{PLUS} обеспечивает степень дискретизации, сопоставимую с вентильными матрицами. Ядро устройства состоит из Sea-of-Tiles™ (Море плиток) (рис. 33.1).

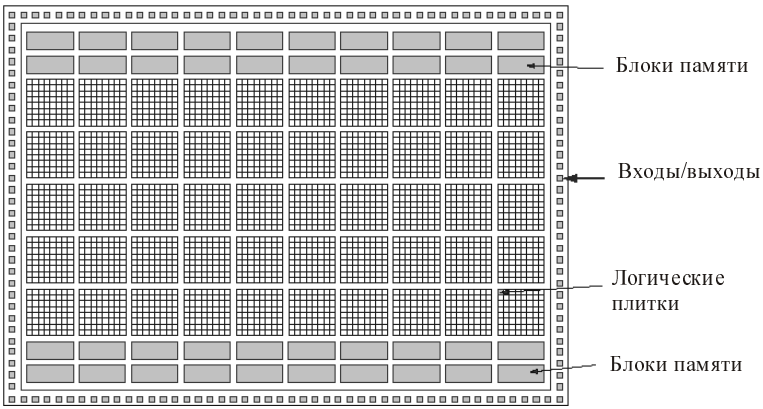


Рис. 33.1. Архитектура ПЛИС семейства ProASIC^{PLUS}

Каждая плитка конфигурируется в определенную пользователем логическую функцию с помощью Flash-выключателей (рис. 33.2 и 33.3).

Flash-выключатели обеспечивают энергонезависимость устройства. В ProASIC^{PLUS} каждый Flash-выключатель содержит два транзистора, которые совместно используют плавающий затвор, для хранения информации программирования. Первый транзистор применяется только для чтения и проверки напряжения плавающего затвора. Второй транзистор используется для подключения или отключения ячейки к ресурсам маршрутизации или для конфигурирования логики, а также для стирания конфигурации. Поскольку Flash-выключатель при выключении/включении питания сохраняет свою конфигурацию такой, как она была запрограммирована, то и вся микросхема в целом начинает выполнять свои функции непосредственно при включении питания.

Таким образом, ProASIC^{PLUS} представляет собой одночипное решение, то есть для ее работы нет необходимости использовать дополнительную микросхему-загрузчик конфигурации.

Логическая ячейка устройства имеет 3 входа (любой из которых может быть инвертирован) и один выход (который может быть подключен к быстрой локальной или к высокоэффективной длинной линии связи). Одну ячейку можно переконфигурировать в несколько разных устройств: либо устройство, реализующее любую

логическую функцию, имеющую 3 входа и 1 выход (кроме трехвходовой XOR); либо регистр с входами сброса и установки; либо триггер с входами сброса, установки и со счетным входом.

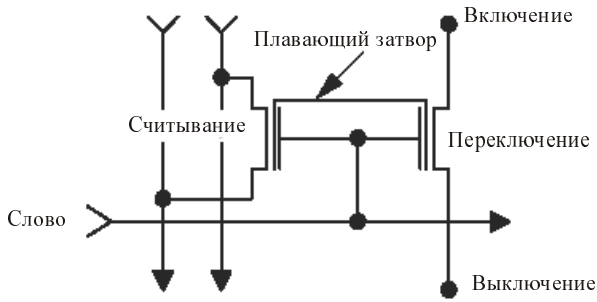


Рис. 33.2. Flash-выключатель

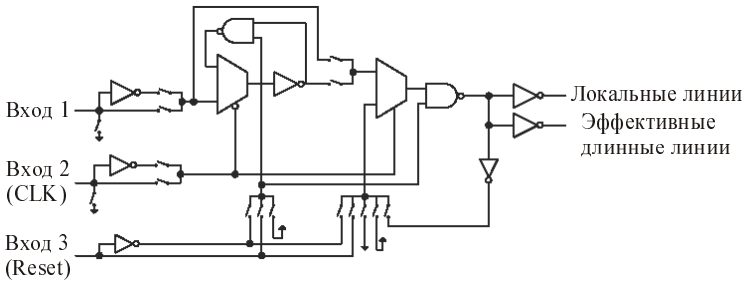


Рис. 33.3. Логическая плитка (ячейка) ядра

33.2.1. Ресурсы маршрутизации

Структура маршрутизации устройств ProASIC^{PLUS} разработана таким образом, чтобы обеспечить высокую эффективность путем использования гибкой иерархии с четырьмя уровнями ресурсов маршрутизации: ультрабыстрые локальные линии; эффективные длинные линии; высокоскоростные очень длинные линии для связи ресурсов и высокоэффективные глобальные сети.

Ультрабыстрые локальные линии (рис. 33.4) — выделенные линии, которые позволяют выход каждой плитки соединять непосредственно с каждым входом восьми окружающих плиток.

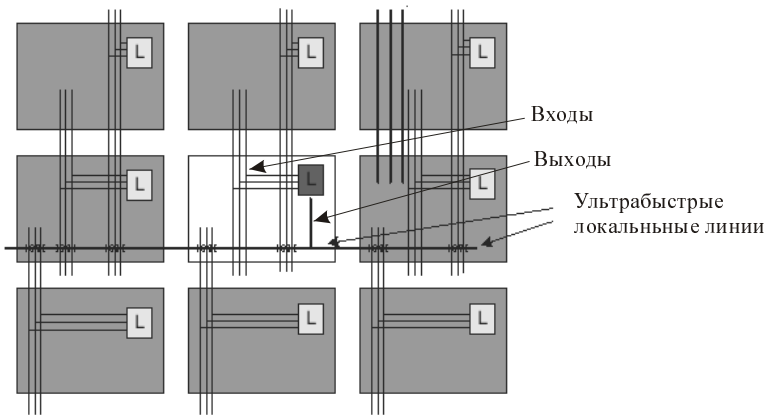


Рис. 33.4. Ультрабыстрые локальные линии

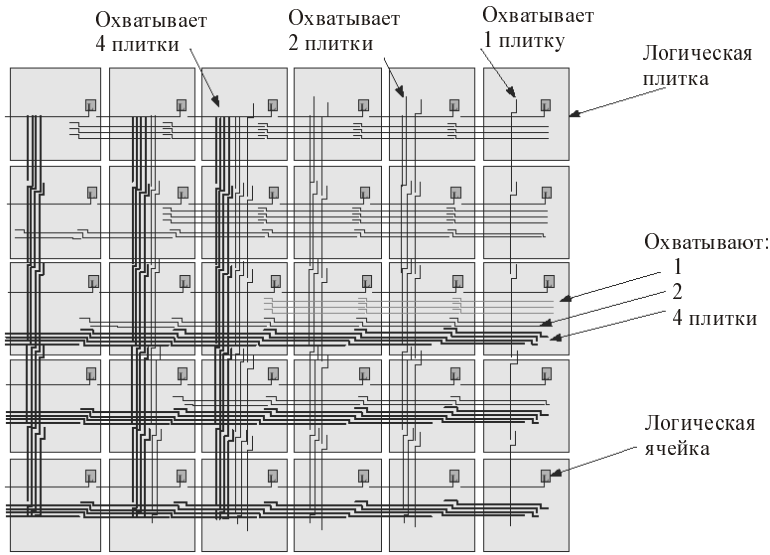


Рис. 33.5. Эффективные длинные линии

Эффективные длинные линии (рис. 33.5) обеспечивают маршрутизацию для более длинных расстояний и для подключений, имеющих большее число разветвлений сигнала на выходе. Эти линии разные по длине, охватывают 1, 2 или 4 плитки, работают и вертикально и горизонтально и покрывают все устройство ProASIC^{PLUS}.

Каждая плитка может управлять сигналами на эффективных длинных линиях, которые могут, в свою очередь, обратиться к каждому входу каждой ячейки.

Активные буферы ограничивают нагрузку на выходы ячеек и вставляются автоматически при конфигурировании ПЛИС в зависимости от расстояний между входами и выходами, а также в зависимости от величины разветвления сигналов на выходах.

Высокоскоростные очень длинные линии (рис. 33.6) охватывают все устройство и работают с минимальной задержкой, они используются для того, чтобы организовать по всему устройству очень длинную связь или связь, имеющую очень высокое разветвление сигнала на выходе.

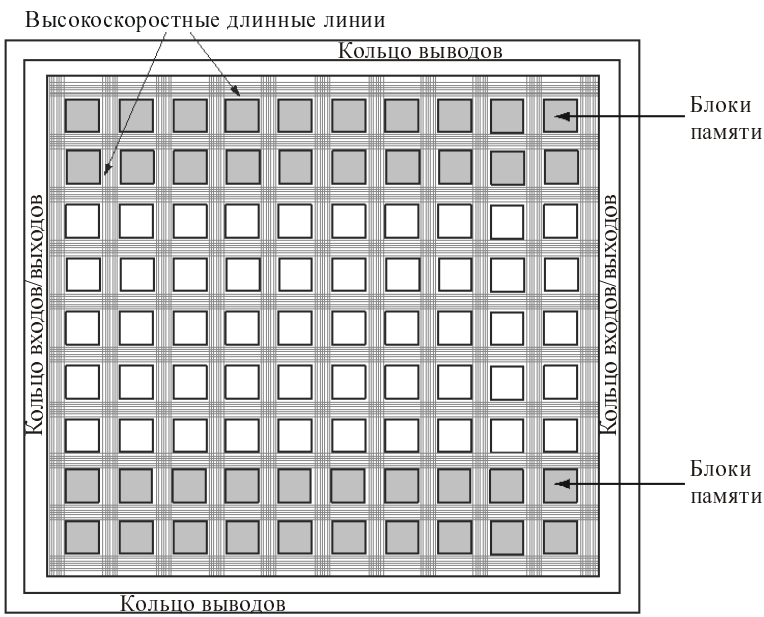


Рис. 33.6. Высокоскоростные длинные линии

Высокоэффективные глобальные линии (рис. 33.7) — линии с высоким разветвлением сигналов на выходе, с низкой асимметрией, что позволяет пропускать по ним высокочастотные сигналы. На глобальные линии подаются сигналы от внешних входов или от внутренней логики. Они обычно используются, чтобы распределить сигналы синхронизации, сброса и другие сигналы, требующие большого разветвления на выходе и минимальной асимметрии. Глобальные линии расположены по кристаллу в виде древовидной структуры. Далее они могут быть соединены по иерархии, таким образом, что будут подключены к каждому входу на всех плитках.

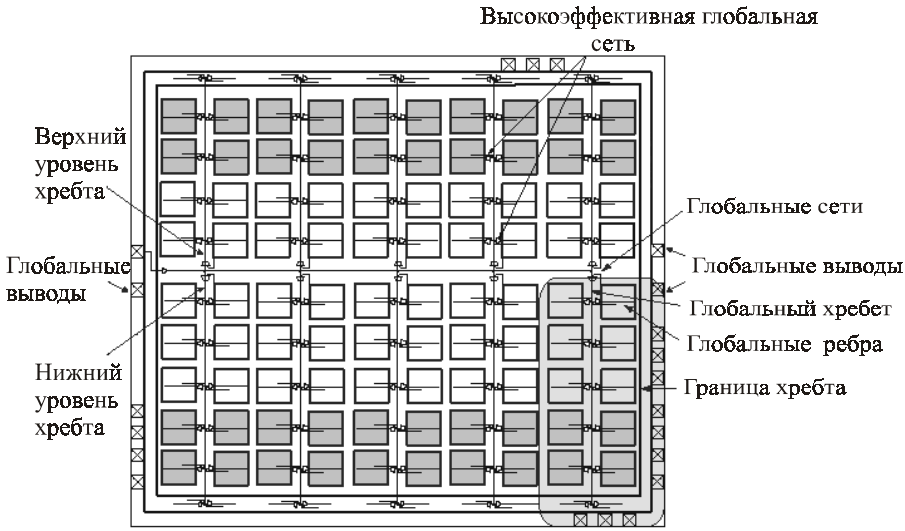


Рис. 33.7. Глобальные сети

33.2.2. Ресурсы синхронизации

Семейство ProASIC^{PLUS} предлагает мощное и гибкое управление схемой таймера (синхронизации) посредством использования специальной аналоговой схемы. Каждый чип имеет два блока обработки синхрочастоты. Каждый блок содержит: ядро PLL (система фазовой автоподстройки частоты), работающее на частоте до 240 МГц; линии задержки; устройства, сдвигающие фазу сигнала (0°, 90°, 180°, 270°); умножители/делители частоты и все схемы управления, необходимые для выбора и подключения входов к глобальной линии таким образом, чтобы обеспечивался двунаправленный доступ к PLL. Это позволяет блоку PLL управлять входами и/или выходами через две глобальных линии на каждой стороне чипа (всего имеется четыре таких линии).

Цепи синхронизации

Одно из основных архитектурных преимуществ семейства — это установка мощности и задержки сигнала в глобальной сети. ProASIC^{PLUS} предлагает четыре глобальные сети. Каждая из этих сетей имеет вид позвоночника с отходящими от него ребрами, что позволяет подвести глобальные сигналы к каждой пластинке (см. рис. 33.7). Числовые характеристики глобальных сетей синхронизации семейства приводятся в табл. 33.2.

Таблица 33.2. *Цепи синхронизации*

	APA075	APA150	APA300	APA450	APA600	APA750	APA1000
Глобальные сети синхронизации/сетей	4	4	4	4	4	4	4
Число ветвей (хребтов) в сети	6	8	8	12	14	16	22
Всего ветвей	24	32	32	48	56	64	88
Высота верхних и нижних ветвей/плитки	16	24	32	32	48	64	80
Количество плиток, охватываемых каждой ветвью	512	768	1,024	1,024	1,536	2,048	2,560
Всего охватываемых плиток	3,072	6,144	8,192	12,288	21,504	32,768	56,320

32.2.3. Блоки ввода/вывода

Чтобы удовлетворить сложным системным требованиям, семейство ProASIC^{PLUS} предлагает устройства с большим количеством пользовательских выводов, например, в микросхеме APA1000 их до 712. Все микросхемы имеют совместимые по выводам корпуса для всех устройств семейства ProASIC^{PLUS}. Если питание для выходов (VDDP) выбрано равным 3,3 В, то каждый вход может быть выборочно сконфигурирован для работы с уровнями логики в 2,5 и 3,3 В.

Таблица 33.3. *Напряжения питания*

	VDDP	
	2,5 В	**3,3 В**
Совместимость по входу	2,5 В	3,3 В, 2,5 В
Выходные уровни	2,5 В	3,3 В, 2,5 В

В табл. 33.3 показаны доступные конфигурации напряжения питания (блок PLL использует независимое питание — 2,5 В на выводах AVDD и AGND). Все выводы имеют схемы защиты от электростатического разряда. Каждый вывод был протестирован на соответствие человеческой модели тела (MIL-STD-883, Метод 3015) при 2000 В.

Выводы имеют полностью перестраиваемую конфигурацию, чтобы обеспечить максимальную гибкость и быстродействие. Каждый вывод может быть сконфигурирован как вход, выход, вывод с тремя состояниями или двунаправленный буфер.

Таблица 33.4. *Характеристики входов-выходов*

Функция	**Описание**
Выводы, сконфигурированные как входы	Индивидуально выбираемые пороговые уровни 2,5 или 3,3 В. Произвольно вход можно сконфигурировать как триггер Шмидта. Опция входа — триггер Шмидта может быть использована только для входа, а не для двунаправленного буфера. Этот тип входа может быть медленнее, чем стандартный вход при тех же условиях, и имеет типичный гистерезис ±0,3 В. Может быть включен нагрузочный резистор. Соответствует стандартам PCI — 3,3 В
Выводы, сконфигурированные как выходы	Индивидуально выбираемые сигналы выхода, соответствующие уровням логики 2,5 или 3,3 В. Соответствует стандартам PCI — 3,3 В. Выбираемая при конфигурации мощность выходного драйвера. Выбираемая при конфигурации скорость нарастания сигнала на выходе. Каждый выход может работать как трехуровневый
Выводы, сконфигурированные как двунаправленные буферы	Индивидуально выбираемые сигналы, соответствующие уровням логики 2,5 или 3,3 В. Соответствует стандартам PCI — 3,3 В. Может быть включен нагрузочный резистор. Выбираемая при конфигурации мощность выходного драйвера. Выбираемая при конфигурации скорость нарастания сигнала на выходе. Каждый выход может работать как трехуровневый

33.2.4. Управление таймером и его характеристики

Семейство ProASIC^{PLUS} обеспечивают проектировщика очень гибкой системой синхронизации. Каждая микросхема семейства содержит два блока PLL, которые выполняют следующие функции:

- установка фазы синхроимпульсов через программируемое время задержки (в диапазоне от –8 до +8 нс с шагом 250 пс);

- минимизация расфазировки (сдвига) тактовых сигналов;
- синтез синхрочастоты.

Каждый блок PLL имеет следующие основные особенности:

- входной частотный диапазон от 1.5 до 180 МГц;
- частотный диапазон обратной связи от 1.5 до 180 МГц;
- выходной частотный диапазон от 6 до 180 МГц;
- выходной фазовый сдвиг равен 0°, 90°, 180° и 270°;
- низкая выходная флуктуация.

33.2.5. Защита проекта пользователя

При увеличении сложности проектов в FPGA появляется потребность защитить интеллектуальную собственность, реализованную пользователем в своем проекте в FPGA. Энергонезависимые перепрограммируемые FPGA, подобные семейству ProASIC^{PLUS} от Actel, предлагают уровни защиты проекта гораздо более высокого уровня, чем у обычного FPGA, выполненного на основе SRAM-технологии или решения в ASIC. В ProASIC^{PLUS} пользовательский проект может быть запрограммирован с установленным ключом защиты, имеющим разрядность кодирования в пределах 79—263 бит, что блокирует внешние попытки прочитать или изменить параметры настройки конфигурации. Если устройство заблокировано, то пользователь может только перепрограммировать устройство, используя определяемый пользователем ключ защиты. Таким образом, можно защитить устройство так, чтобы было невозможно прочитать и скопировать его содержание.

33.2.6. Встроенная память

Встроенная память расположена вдоль верхней и нижней границы микросхемы блоками по 256x9 бит. Каждый блок может программироваться, как независимое запоминающее устройство, или блоки могут быть объединены, чтобы образовать большие, более сложные блоки памяти. Единая конфигурация памяти может включать блоки, размещенные сверху и снизу.

33.4. Семейство FPGA eX

Рассмотрим ПЛИС фирмы Actel семейства FPGA eX, ее особенности и область применения.

33.4.1. Технология Antifuse

Antifuse ПЛИС Actel объединяют в себе преимущества программируемой логики и базовых матричных кристаллов (БМК). Они дают потребителю возможность производить БМК непосредственно у себя на столе.

Являясь идеальной альтернативой заказным интегральным схемам, Antifuse ПЛИС ACTEL сэкономят финансовые средства производителя даже на маленьких партиях (приблизительно от 70 микросхем) при замене традиционных CPLD.

Основными преимуществами Antifuse ПЛИС Actel являются:

- совместимость по выводам микросхем с различным количеством вентилей;
- рекордная системная производительность — свыше 500 МГц;
- экономия площади печатной платы за счет отсутствия CPLD-загрузчика;
- исключена возможность несанкционированного считывания конфигурации.

Подробнее архитектуру Antifuse рассмотрим на семействе eX.

Основные характеристики:

- 240 МГц системная производительность;
- 350 МГц внутренняя производительность.

Технические условия:

- от 3 000 до 12 000 доступных системных вентилей;
- максимум 512 триггеров;
- 0,22 μm технология КМОП;
- до 132 программируемых пользователем вводов.

Свойства:

- высокопроизводительная, с низким энергопотреблением FPGA Antifuse;
- режим LP/Sleep для дополнительного энергосбережения;
- улучшенный малогабаритный корпус;
- одночипное решение;
- энергонезависимость;
- не требуется соблюдать последовательности включения/выключения питания;
- индивидуальный контроль скорости нарастания выходного напряжения.

33.4.2. Описание семейства FPGA eX

Архитектура семейства FPGA eX

Для семейства используется структура "sea-of-modules" (море-модулей), где полный слой микросхемы составляет сетка модулей логической схемы и фактически нет участков, использующихся на связь или маршрутизацию.

Соединение этих логических модулей достигается использованием программируемых антиплавких перемычек "металл-к-металлу". Эти перемычки представляют собой обычно разомкнутую цепь, и при программировании образуется постоянное низкоомное соединение.

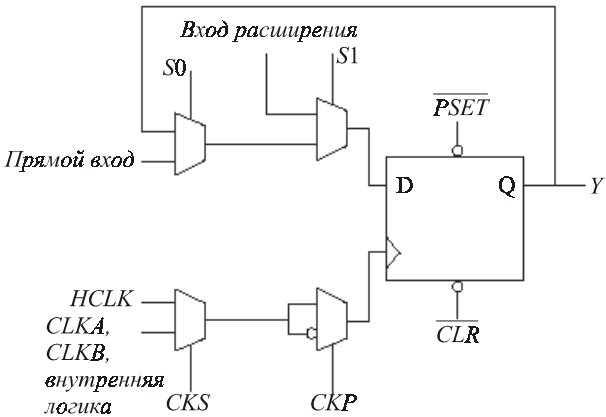


Рис. 33.8. R-ячейка

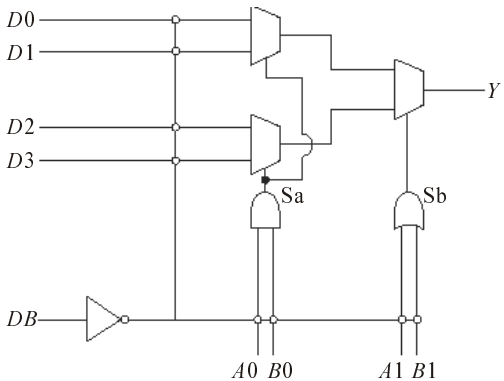


Рис. 33.9. C-ячейка

Логические модули бывают двух типов, регистровая ячейка (R-ячейка) и комбинаторная ячейка (C-ячейка).

R-ячейка (рис. 33.8) содержит триггер с асинхронным сбросом, асинхронной предустановкой и сигналами разрешения подачи синхроимпульсов ($S0$ и $S1$).

C-ячейка (рис. 33.9) выполняет ряд комбинаторных функций и имеет до пяти входов.

Две C-ячейки могут быть объединены вместе для создания триггера и подражания R-ячейке. Это особенно полезно при осуществлении несинхронизируемых цепей, и когда разработчик исчерпает R-ячейки.

Структура модуля

C-ячейки и R-ячейки логического модуля размещаются в горизонтальные банки, называемые кластерами, каждый из которых содержит две C-ячейки и одну R-ячейку в топологии C-R-C.

Кластеры организованы в модули, называемые суперкластерами, для улучшения эффективности проектов и производительности микросхемы (рис. 33.10).

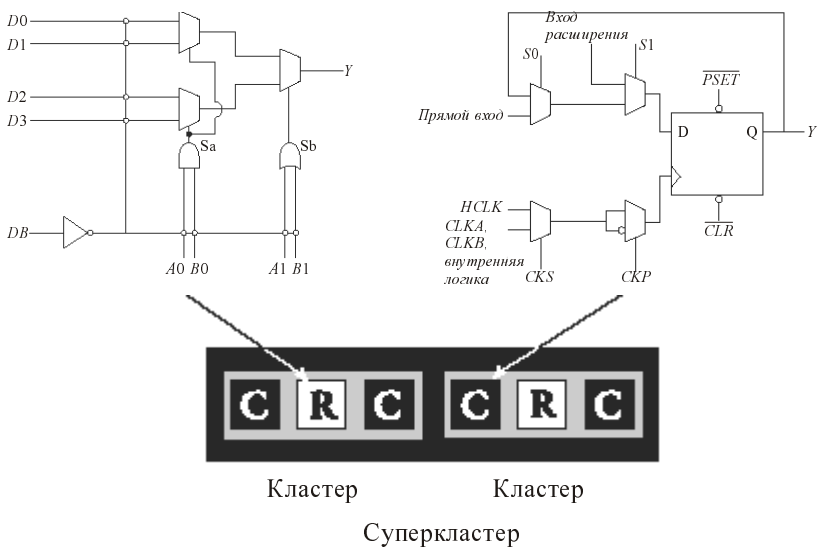


Рис. 33.10. Кластерная архитектура

Ресурсы маршрутизации

Кластеры и суперкластеры могут быть связаны с помощью двух локальных ресурсов маршрутизации, называемых **FastConnect** и **DirectConnect**, которые позволяют

чрезвычайно быстро и прогнозируемо соединять модули в пределах кластеров и суперкластеров. Эта архитектура маршрутизации также уменьшает число антиплавких перемычек, требуемых для замыкания схемы, обеспечивая самые высокие возможные характеристики.

DirectConnect — горизонтальный ресурс маршрутизации, который обеспечивает связь от С-ячеек до соседних R-ячеек в суперкластере. DirectConnect использует аппаратный путь сигнала, не требующий программируемого соединения, для достижения быстрого распространения сигнала, меньше чем 0.1нс.

FastConnect — обеспечивает горизонтальную маршрутизацию между любыми двумя модулями логической схемы в пределах суперкластера и вертикальную маршрутизацию, для соединения с суперкластером, находящимся непосредственно ниже его. Только одно программируемое соединение используется в FastConnect цепи, давая максимальное распространение сигнала 0.3 нс.

В дополнение к DirectConnect и FastConnect, архитектура включает два глобальных ориентируемых ресурса маршрутизации, известных как **сегментированная маршрутизация** (segmented routing) и **высокоскоростная маршрутизация** (high-drive routing).

Сегментная структура маршрутизации обеспечивается проводниками разной длины для чрезвычайно быстродействующей маршрутизации между суперкластерами. Точная комбинация длин проводников и антиплавких перемычек в пределах каждой цепи выбирается полностью автоматически программным обеспечением размещения и трассировки (place-and-route), чтобы минимизировать время задержки распространения сигнала.

Ресурсы синхронизации

Высокоскоростная структура маршрутизации обеспечивает три сети синхронизации.

Первая, названа HCLK, проходит от буфера HCLK на мультиплексор в каждой R-ячейке. HCLK не может быть связана с комбинационными логическими ячейками. Это обеспечивает быстрый путь распространения для синхросигнала, до 3,9 нс.

Остальные две сети синхронизации (CLKA, CLKB) являются глобальными сетями синхронизации, которые могут брать начало от внешних выводов или от внутренних сигналов логической схемы в пределах микросхемы. CLKA и CLKB могут быть связаны с последовательностными или комбинационными логическими модулями.

Производительность

Комбинация архитектурных возможностей, описанных ранее, позволяет ПЛИС eX работать с внутренними частотами, превышающими 350 МГц для очень быстрого выполнения сложных логических функций.

Модули ввода/вывода

Каждый вывод микросхемы eX может быть сконфигурирован как вход, выход, вывод с тремя состояниями, или двунаправленный вывод.

Все неиспользованные выводы автоматически конфигурируются как выводы с тремя состояниями.

Лабораторная работа. Программирование проектов в ПЛИС фирмы Actel

Цель работы

Получение навыков программирования проектов в ПЛИС фирмы Actel с использованием программатора FlashPro и макетной платы ProASIC^{PLUS} Evaluation Board.

Программа работы

По следующему далее описанию сначала запрограммируйте в ПЛИС фирмы Actel проект, предложенный в предыдущей лабораторной работе, а затем запрограммируйте свой, индивидуальный, проект.

Описание программирования проектов в ПЛИС фирмы Actel

После генерации файла программирования выполняется программирование ПЛИС с использованием программатора и программного обеспечения FlashPro.

1. Начальная установка.

   Перед выполнением любых действий с программатором FlashPro, его нужно должным образом настроить. Подключите FlashPro ленточным кабелем к разъему для программирования и включите питание.

2. Установка FlashPro Lite.
   - В меню **File** выберите **Connect** (Подключить).
   - В списке **Port** (Порт) выберите порт, с которым связан FlashPro — **lpt1**.
   - В списке **Configuration** (Конфигурация) выберите **ProASIC^{PLUS}**.
   - Щелкните **Connect**. Сообщение о благополучном соединении или об ошибках выводится в окне **Log**.

3. Выбор микросхемы.

   Выберите **APA 075** в списке **Device** (Устройство).

4. Загрузка STAPL-файла.

   Программатор FlashPro Lite использует файл STAPL (*.stp), для программирования микросхемы.

   - Нажмите кнопку **Open File** на панели инструментов или в меню **File** и выберите **Open STAPL file**.
   - Выберите файл **STAPL** и нажмите **Open**.

   FlashPro загрузит файл.

5. Выбор действий.

   После загрузки, выберите действие из списка **Action** (Действие) (табл. 33.5).

Таблица 33.5. *Список действий*

Option	Action
QUERY_SECURITY	Проверка защитных свойств. Если микросхема программировалась со свойствами защиты, то при обработке этого действия параметры "запрет чтения" и "запрет записи" будут равны 1. Если защиты нет, то эти параметры будут равны 0
ERASE	Стирает микросхему
READ_IDCODE	Читает ID-код микросхемы
VERIFY	Проверяет, программировалась ли микросхема загруженным файлом STPL. Если загружена неправильная картотека STPL, в окне **Log** появляется результат **Exit 11**. Успешная операция приводит к **Exit 0**
PROGRAM	Программирование микросхемы
DEVICE_INFO	Отображает серийный номер микросхемы, имя проекта, запрограммированного в нее, и контрольную сумму программы

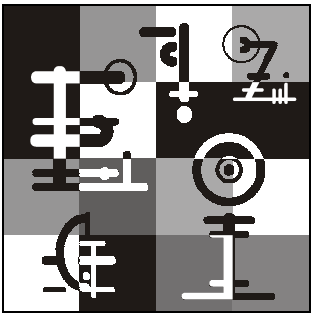
6. Программирование микросхемы.

   - В списке **Action** выберите **PROGRAM**.
   - В списке **Device** выберите **APA 075**.
   - Нажмите кнопку **Execute** (Выполнить) на инструментальной панели.

     Прогресс программирования отображается в окне **Log**. Сообщение **Exit 0** указывает, что прибор успешно запрограммирован.

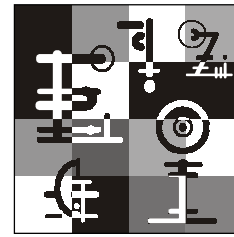
ОБРАТИТЕ ВНИМАНИЕ!

Не прерывайте процесс программирования, это может разрушить прибор или программатор.



Часть VIII

Средства проектирования фирмы Altera для цифровых устройств средней интеграции



Глава 34

Стартовый комплект для работы с ПЛИС Altera

Стартовый комплект Altera UP2 (University Program) служит для ознакомления с особенностями работы двух семейств ПЛИС фирмы Altera — MAX 7000S и FLEX 10K.

В состав комплекта входит:

- макетная плата UP2 Education Board;
- программатор ByteBlasterMV;
- программное обеспечение MAX II Plus 10.0.

34.1. Макетная плата UP2 Education Board

На плате представлены элементы двух семейств — FLEX 10K и MAX 7000S.

Представитель семейства FLEX 10K — EPF10K70. Элемент EPF10K70 построен по технологии SRAM (Static Random-Access Memory — статическое запоминающее устройство с произвольной выборкой). Он доступен в корпусе с 240 выводами и имеет 3 744 логических элемента (LE) и девять встроенных блоков памяти.

Каждый LE состоит из таблицы перекодировок LUT (look-up table) с четырьмя входами, программируемого триггера и специальных цепей для функции переноса и каскадирования.

Каждый встроенный блок памяти обеспечивает 2 048 бит, которые могут использоваться для создания RAM, ROM, или FIFO.

Представитель семейства MAX 7000S — EPM7128S. Прибор EPM7128S — член семейства высокоэффективных элементов высокой плотности MAX 7000S, основан на перепрограммируемых элементах ПЗУ (EEPROM). Установлен в пластиковый разъем с 84 J-образными выводами и имеет 128 макроячеек.

Каждая макроячейка имеет матрицу с фиксированными элементами "ИЛИ" и программируемыми элементами "И", а также конфигурируемый регистр с емкостью в 2500 вентилей.

EPM7128S подходит для разработок как больших комбинаторных, так и последовательностных логических функций.

34.1.1. Состав платы

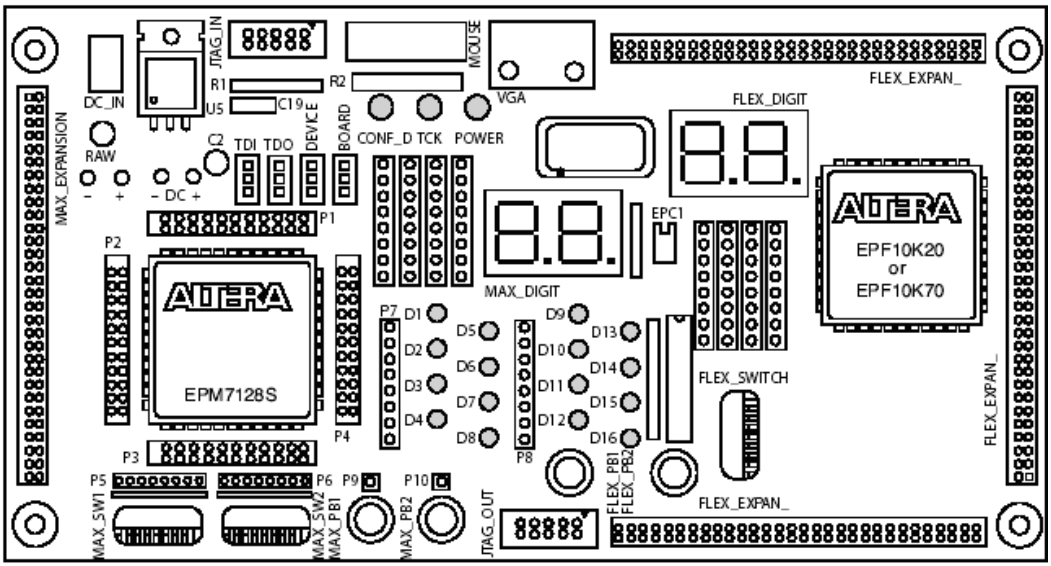


Рис. 34.1. Внешний вид макетной платы

Генератор

Плата содержит генератор на 25,75 МГц. Выход генератора соединен с глобальными входами синхронизации элементов: вывод 83 на EPM7128S и вывод 91 на FLEX 10K.

Разъем программатора

10-штырьковый разъем для подключения программатора Byte Blaster MV. Данные поступают в ПЛИС через вывод TDI и принимаются через TDO.

В табл. 34.1 представлено соответствие номеров выводов их именам.

Таблица 34.1. *Разъем JTAG*

Pin	JTAG Signal	Pin	JTAG Signal
1	TCK	6	Не подключен
2	GND	7	Не подключен
3	TDO	8	Не подключен
4	VCC	9	TDI
5	TMS	10	GND

Перемычки

На плате находятся четыре перемычки (TDI, TDO, DEVICE и BOARD), необходимые для установки конфигурации JTAG (рис. 34.2).

Цепь JTAG может быть установлена в различные конфигурации (такие как программирование только EPM7128S, конфигурирование только EPF10K70, программирование и конфигурирование обоих элементов или совместное соединение нескольких макетных плат) (табл. 34.2).

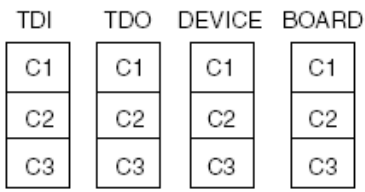


Рис. 34.2. Перемычки

Таблица 34.2. *Положение перемычек для установки необходимой конфигурации*

Desired Action	TDI	TDO	DEVICE	BOARD
Программирование только EPM7128S	C1&C2	C1&C2	C1&C2	C1&C2
Конфигурирование только EPF10K70	C2&C3	C2&C3	C1&C2	C1&C2
Программирование/конфигурирование обоих элементов	C2&C3	C1&C2	C2&C3	C1&C2
Совместное соединение нескольких плат	C2&C3	OPEN	C2&C3	C2&C3

Во время конфигурирования зеленый светодиод CONF_D выключен и мигает диод TCK, указывая, что данные передаются. После успешного конфигурирования, CONF_D будет светиться.

Ресурсы элемента EPM7128S

Макетная плата обеспечивает следующие ресурсы для EPM7128S:

- сигнальные выводы прибора доступны через разъемы;
- две кнопки;
- два восьмеричных переключателя;
- 16 светодиодов;
- 2 семисегментных индикатора.

Вокруг элемента на плате находятся 4 разъема, через которые могут быть доступны сигнальные выводы элемента. Прибор имеет электрическую связь по умолчанию только с разъемом JTAG и генератором, с остальными составляющими связи нет, поэтому, если необходимо использование кнопок, переключателей, светодиодов или цифровых индикаторов, нужно создать соединение с помощью перемычек.

Кнопки

На плате находятся 2 кнопки, которые обеспечивают при нажатии низкий уровень и имеют обозначение MAX_PB1 и MAX_PB2.

Переключатели

Имеется 2 переключателя на 8 ключей каждый. При открытом ключе — логическая 1, при закрытом — логический 0. Переключатели имеют обозначение MAX_SW1 и MAX_SW2.

Светодиоды

Плата UP2 содержит 16 светодиодов D1—D16.

2 семисегментных индикатора

Соответствие выводов ПЛИС и сегментов индикатора определяются в соответствии с рис. 34.3 и табл. 34.3.

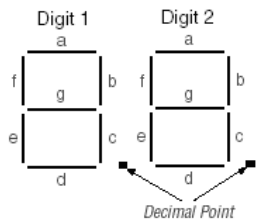


Рис. 34.3. Семисегментные индикаторы

Таблица 34.3. *Соответствие сегментов индикатора выводам ПЛИС*

Сегмент индикатора	Вывод для Digit 1	Вывод для Digit 2
a	58	69
b	60	70
c	61	73
d	63	74
e	64	76
f	65	75
g	67	77
Точка	68	79

Ресурсы элемента EPF10K70

Выводы элемента EPF10K70 семейства ПЛИС FLEX 10K соединены на плате с индикаторами, кнопками и переключателями. Плата UP2 обеспечивает следующие ресурсы для элемента EPF10K70:

- ❒ две кнопки;
- ❒ один переключатель на 8 ключей;
- ❒ 2 семисегментных индикатора;
- ❒ разъем VGA;
- ❒ разъем PS/2.

Кнопки

Кнопка FLEX_PB1 соединена с выводом 28, а FLEX_PB2 с выводом 29 элемента FLEX 10K.

Переключатель

Имеет обозначение FLEX_SW1. Назначение ключей переключателя выводам элемента представлено в табл. 34.4.

Таблица 34.4. *Соответствие ключей выводам ПЛИС*

Ключ	FLEX10KPin	Ключ	FLEX10KPin
FLEX_SWITCH-1	41	FLEX_SWITCH-5	36
FLEX_SWITCH-2	40	FLEX_SWITCH-6	35
FLEX_SWITCH-3	39	FLEX_SWITCH-7	34
FLEX_SWITCH-4	38	FLEX_SWITCH-8	33

Семисегментные индикаторы FLEX_DIGIT

Два семисегментных индикатора (табл. 34.5) позволяют выводить данные с выводов сконфигурированного в ПЛИС проекта.

Таблица 34.5. *Соответствие сегментов индикатора выводам ПЛИС*

Сегмент индикатора	Вывод для Digit 1	Вывод для Digit 2
A	6	17
B	7	18
C	8	19
D	9	20
E	11	21
F	12	23
G	13	24
Точка	14	25

VGA-интерфейс

Интерфейс VGA позволяет прибору FLEX 10K управлять внешним видеомонитором (табл. 34.6).

Таблица 34.6. *Связь выводов разъема и выводов ПЛИС*

Сигнал	Вывод разъема	Вывод FLEX 10K
RED	1	236
GREEN	2	237
BLUE	3	238
GND	6, 7, 8, 10, 11	–
HORIZ_SYNC	13	240
VERT_SYNC	14	239
Не подключены	4, 5, 9, 15	–

Разъем PS/2

Интерфейс позволяет прибору FLEX 10K получать данные от мыши PS/2 или клавиатуры PS/2 (табл. 34.7).

Таблица 34.7. *Связь выводов разъема и выводов ПЛИС*

Сигнал	Вывод разъема	Вывод FLEX 10K
MOUSE_CLK	1	30
MOUSE_DATA	3	31
VCC	5	–
GND	2	–

Лабораторная работа. Программирование и конфигурирование ПЛБИС в системе MAX+PLUS II фирмы Altera

Цель работы

Данная работа предназначена для ознакомления с программированием и конфигурированием ПЛБИС в системе MAX+PLUS II фирмы Altera на предложенных примерах.

Методические указания

Программирование или конфигурирование приборов на плате UP2 требует установки встроенных перемычек и опций программатора JTAG в MAX+PLUS II, а также соединения кабеля Byte Blaster MV с параллельным портом PC и разъемом JTAG_IN на плате.

1. Конфигурирование прибора EPM7128S.

   Чтобы программировать только EPM7128S, установите перемычки TDI, TDO, DEVICE и BOARD в соответствии с рис. 34.4.

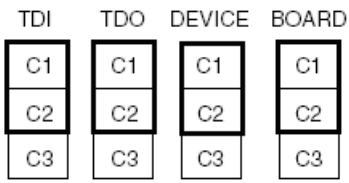


Рис. 34.4. Положение перемычек при программировании прибора EPM7128S

Подсоедините кабель ByteBlasterMV непосредственно на параллельный порт PC и на JTAG_IN на плате.

Установки в MAX+PLUS II для программирования EPM7128S.

- В диалоговом окне **Device** (вызывается командой **Assign** | **Device** (Назначить | Устройство)) в поле **Device Family** (Семейство устройства) выберите **MAX7000S**, а в поле **Device — EPM7128S**.
- Выберите в меню **MAX+PLUS II** | **Programmer** (MAX+PLUS II | Программатор), откроется окно **Programmer**. Обратите внимание, что в окне отображается файл с расширением sof, это файл конфигурации.
- В окне нажмите кнопку **Program** (Программировать), начнется процесс программирования прибора.
- После завершения программирования устройство начнет работать.

2. Конфигурирование прибора EPF10K70.

Чтобы конфигурировать только EPF10K70, установите перемычки TDI, TDO, DEVICE в соответствии с рис. 34.5.

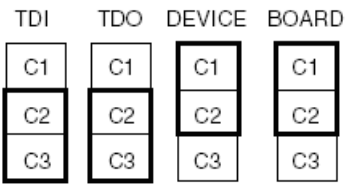


Рис. 34.5. Установка перемычек для программирования только прибора EPF10K70

Подсоедините кабель Byte Blaster MV непосредственно на параллельный порт PC и на JTAG_IN на плате.

Установки в MAX+PLUS II для программирования EPF10K70.

- В диалоговом окне **Device** (**Assign** | **Device** (Назначить | Устройство)) в поле **Device Family** (Семейство устройства) выберите **FLEX10K**, а в поле **Device — EPF10K70**.
- Выберите в меню **MAX+PLUS II** | **Programmer** (MAX+PLUS II | Программатор), откроется окно **Programmer**. Обратите внимание, что в окне отображается файл с расширением sof, это файл конфигурации.
- В окне нажмите кнопку **Configure** (Конфигурировать), начнется процесс программирования прибора.
- После завершения программирования устройство начнет работать.

Программа работы

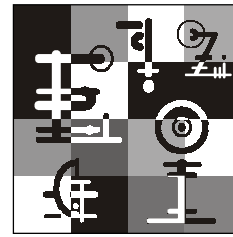
Рассмотрим процесс конфигурирования на примере "Монитор VGA", описанном в *главе 35*.

1. Соедините программатор с макетной платой и параллельным портом PC.
2. Подключите сигнальный кабель VGA-монитора к макетной плате.
3. Откройте графический файл Vga_led.gdf в системе проектирования MAX+PLUS II. Для того чтобы открытый файл рассматривался системой как проект, необходимо выбрать пункт меню **File** | **Project** | **Set Project to Current File** (Файл | Проект | Установить проект в текущий файл).
4. Обратите внимание, что все входы/выходы проекта имеют назначение выводам ПЛИС (надпись темно-красным цветом). Если необходимо снова назначить выводы или отредактировать назначение, выберите в меню **Assign** | **Pin** | **Location** | **Chip** (Назначение | Вывод | Размещение | Чип), откроется диалоговое окно, в котором именам сигналов можно назначить номер вывода используемой ПЛИС.
5. Выберите в меню **Assign** | **Devices** (Назначение | Устройство) и удостоверьтесь, что в поле **Devices** стоит значение, совпадающее с надписью на корпусе ПЛИС, если это не так, измените значение.

Выберите в меню **MAX+PLUS II** | **Programmer**, откроется окно **Programmer**. Обратите внимание, что в окне отображается файл Vga_led.sof. Это файл конфигурации, полученный в процессе компилирования проекта.

Нажмите кнопку **Configure**, начнется процесс конфигурирования ПЛИС. После сообщения об успешном конфигурировании нажмите кнопку **OK**.

Теперь необходимо убедиться, что сконфигурированный проект работает. В данном случае дисплей монитора должен светиться красным цветом, а при нажатии кнопки **PB1** дисплей должен гаснуть.



Глава 35

Примеры проектирования и программирования в ПЛИС фирмы Altera

Рассмотрим примеры реализации на ПЛИС фирмы Altera устройств, обеспечивающих работу VGA-монитора, а также клавиатуры и мыши с разъемом PS/2.

35.1. Пример реализации на ПЛИС фирмы Altera работы VGA-монитора

Стандартный монитор VGA состоит из сетки пикселов, которые могут быть разделены на строки и столбцы. Монитор VGA содержит 480 строк, с 640 пикселами в строке. Каждый пиксел может отобразить различные цвета, в зависимости от состояния сигналов: красного, зеленого и синего.

Каждый монитор VGA имеет внутреннюю синхронизацию, с помощью которой определяется, когда должен обновляться каждый пиксел. Начинается каждый цикл обновления в верхнем левом углу экрана, который можно принять за начало координат плоскости X-Y. Когда монитор получает импульс горизонтальной синхронизации, происходит обновление следующей строки. Это повторяется, пока луч не достигает основания экрана. Затем, после прихода вертикального импульса синхронизации, регенерация начинается сверху экрана, то есть в левом верхнем углу.

35.1.1. VGA-синхронизация

Для корректной работы VGA-монитора он должен своевременно получать необходимые данные. На рис. 35.1 и 35.2 показано, как должны поступать импульсы вертикальной и горизонтальной синхронизации в соответствии с данными о цвете.

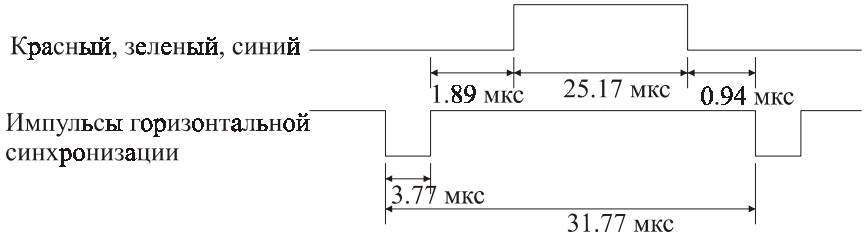


Рис. 35.1. Цикл горизонтальной синхронизации

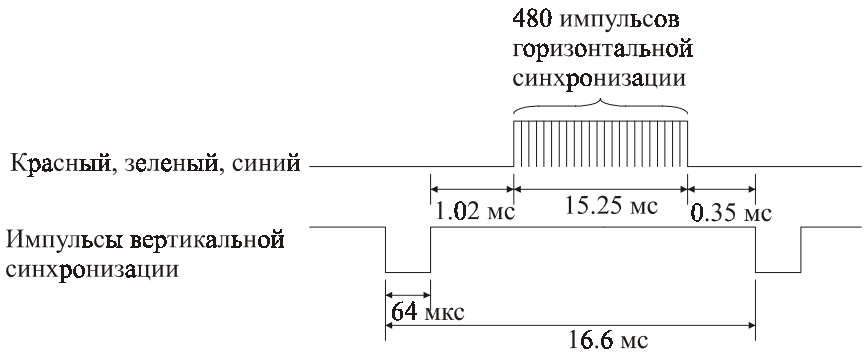


Рис. 35.2. Цикл вертикальной синхронизации

Частота синхронизации и число пикселов, которые монитор должен обновить, определяют время, требуемое для обновления каждого пиксела, и время, требуемое для обновления всего экрана. Следующие уравнения грубо вычисляют время, требуемое монитору для выполнения всех его функций.

$$T_{пиксела} = \frac{1}{f_{CLK}} = 40 нс$$

$$T_{строки} = 31.77$$

$$T_{экрана} = 31.77$$

Где:

$T_{пиксела}$ — время, необходимое для обновления одного пиксела

f_{CLK} — 25.175 МГц

$T_{строки}$ — время, необходимое для обновления одной строки

$T_{экрана}$ — время, необходимое для обновления всего экрана

35.1.2. Использование ПЛИС для генерации видеосигнала VGA

В данном примере используются ресурсы ПЛИС для получения видеосигнала. Необходимо только 5 сигналов — 2 синхросигнала и 3 сигнала цветности RGB (Red, Green, Blue — красный, зеленый, синий).

Для конвертирования TTL-уровня от микросхемы ПЛИС в аналоговый RGB-сигнал используется простая схема из резисторов и диодов. Эта схема поддерживает 2 уровня для каждого RGB-сигнала — высокий и низкий, поэтому возможно производить 8 цветов.

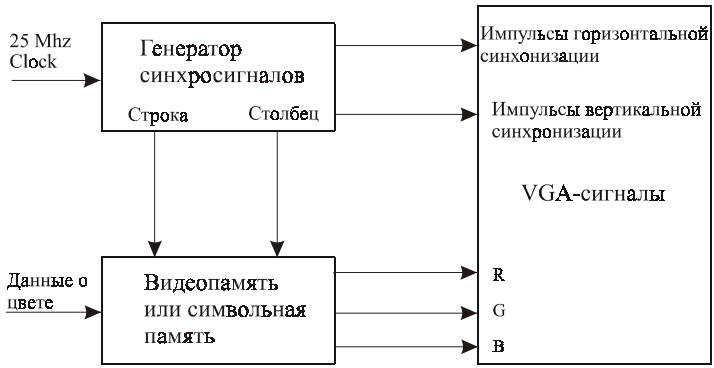


Рис. 35.3. Структурная схема генератора видеосигнала

На рис. 35.3 приведена структурная схема простого генератора видеосигнала. Используется частота синхронизации 25.175 МГц, и на обновление одного пиксела уходит примерно 40 нс. Генератор синхросигналов применяется для получения импульсов вертикальной и горизонтальной синхронизации, а также для генерации адреса строки и столбца. Адрес строки и столбца поступает в видеопамять для графических данных или в память генератора символов при текстовом режиме. Необходимые модули памяти RAM или ROM конструируются внутри кристалла ПЛИС.

Библиотечная функция `VGA_SYNC` (рис. 35.4) может быть использована для генерации временных сигналов, необходимых для работы VGA-монитора. Модуль написан на VHDL, можно использовать его символ при создании своего проекта.

Приведенный в листинге 35.1 VHDL-код генерирует горизонтальные и вертикальные синхросигналы, используя 10-битные счетчики `H_count` — для горизонтальных и `V_count` — для вертикальных импульсов.

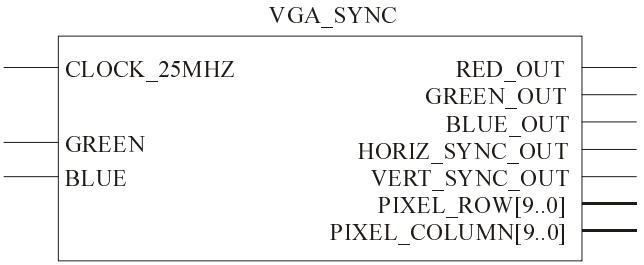


Рис. 35.4. Модуль `VGA_SYNC` библиотеки UPcore

Счетчики `H_count` и `V_count` генерируют адреса для строк и столбцов, а также доступны другим процессам. Эти сигналы используются для определения расположения координат X и Y.

Сигнал `video_on` служит для отключения RGB-данных, когда пикселы не должны отображаться (например, на время обратного хода луча монитора).

На выходе модуля имеем синхросигналы, адреса строки и столбца и RGB-данные.

Листинг 35.1. `VGA_SYNC`

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;
ENTITY VGA_SYNC IS
       PORT(  clock_25МГц, red, green, blue              : IN   STD_LOGIC;
                     red_out, green_out, blue_out, horiz_sync_out,
vert_sync_out  : OUT  STD_LOGIC;
                     pixel_row, pixel_column: OUT STD_LOGIC_VECTOR(9 DOWNTO
0));
END VGA_SYNC;
ARCHITECTURE a OF VGA_SYNC IS
       SIGNAL horiz_sync, vert_sync : STD_LOGIC;
       SIGNAL video_on, video_on_v, video_on_h : STD_LOGIC;
       SIGNAL h_count, v_count :STD_LOGIC_VECTOR(9 DOWNTO 0);
BEGIN
-- video_on is high only when RGB data is displayed
video_on <= video_on_H AND video_on_V;
PROCESS
BEGIN
```

```
        WAIT UNTIL(clock_25МГц'EVENT) AND (clock_25МГц='1');
--Generate Horizontal and Vertical Timing Signals for Video Signal
-- H_count counts pixels (640 + extra time for sync signals)
--
--  Horiz_sync  -----------------------------------_________--------
--  H_count       0                640             659       755    799
        IF (h_count = 799) THEN
                h_count <= "0000000000";
        ELSE
                h_count <= h_count + 1;
        END IF;

--Generate Horizontal Sync Signal using H_count
        IF (h_count <= 755) AND (h_count >= 659) THEN
                horiz_sync <= '0';
        ELSE
                horiz_sync <= '1';
        END IF;

--V_count counts rows of pixels (480 + extra time for sync signals)
--
--  Vert_sync      --------------------------------------_______-------------
--  V_count         0                                480    493-494         524
        IF (v_count >= 524) AND (h_count >= 699) THEN
                v_count <= "0000000000";
        ELSIF (h_count = 699) THEN
                v_count <= v_count + 1;
        END IF;

-- Generate Vertical Sync Signal using V_count
        IF (v_count <= 494) AND (v_count >= 493) THEN
                vert_sync <= '0';
        ELSE
                vert_sync <= '1';
        END IF;

-- Generate Video on Screen Signals for Pixel Data
        IF (h_count <= 639) THEN
                video_on_h <= '1';
                pixel_column <= h_count;
```

```
        ELSE
                video_on_h <= '0';
        END IF;

        IF (v_count <= 479) THEN
                video_on_v <= '1';
                pixel_row <= v_count;
        ELSE
                video_on_v <= '0';
        END IF;
-- Put all video signals through DFFs to elminate any delays that cause a
blurry image
                red_out <= red AND video_on;
                green_out <= green AND video_on;
                blue_out <= blue AND video_on;
                horiz_sync_out <= horiz_sync;
                vert_sync_out <= vert_sync;
END PROCESS;
END a;
```

35.1.3. Работа VGA в текстовом режиме

В библиотеке `UPcore` для хранения изображения символов используется ПЗУ (рис. 35.5 и 35.6). Для одного символа выделяется ячейка 8*8 бит. Например символ А расположен в памяти, начиная с адреса 000001 (табл. 35.1).

Таблица 35.1. *Представление символов в ПЗУ*

Адрес	**Данные символа**
000001000:	00**11**000
000001001:	00**1111**00
000001010:	0**1**00**11**0
000001011:	0**1111**10
000001100:	0**1**00**11**0
000001101:	0**1**00**11**0
000001110:	0**1**00**11**0
000001111:	00000000

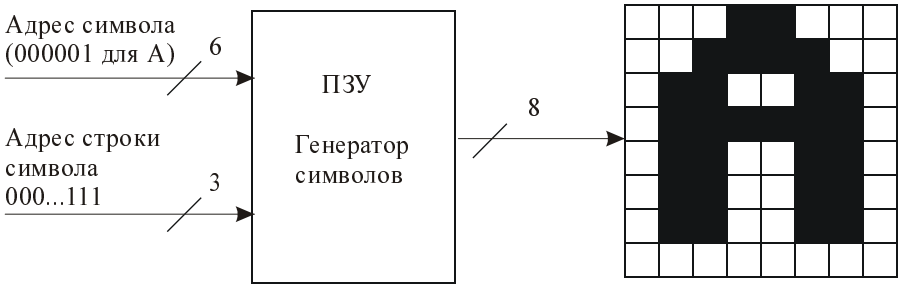


Рис. 35.5. Блок-схема памяти символов

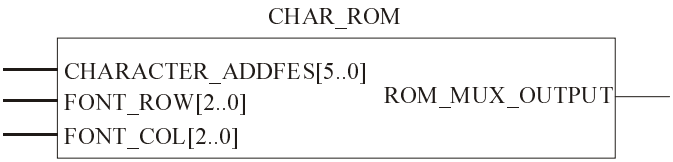


Рис. 35.6. Модуль `CHAR_ROM` библиотеки `UPcore`

Для организации данного модуля используется библиотека параметризованных модулей фирмы Altera (листинг 35.2).

Листинг 35.2. `Char_ROM`

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY Char_ROM IS
       PORT(  character_address                    : IN   STD_LOGIC_VECTOR(5
DOWNTO 0);
                      font_row, font_col                    : IN
       STD_LOGIC_VECTOR(2 DOWNTO 0);
                      rom_mux_output : OUT  STD_LOGIC);
END Char_ROM;

ARCHITECTURE a OF Char_ROM IS
```

```
        SIGNAL rom_data: STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL rom_address: STD_LOGIC_VECTOR(8 DOWNTO 0);
BEGIN
-- Small 8 by 8 Character Generator ROM for Video Display
-- Each character is 8 8-bits words of pixel data
 char_gen_rom: lpm_rom
      GENERIC MAP ( lpm_widthad => 9,
        lpm_numwords => 512,
        lpm_outdata => "UNREGISTERED",
        lpm_address_control => "UNREGISTERED",
-- Reads in mif file for character generator font data
         lpm_file => "tcgrom.mif",
         lpm_width => 8)
      PORT MAP ( address => rom_address, q => rom_data);

rom_address <= character_address & font_row;
-- Mux to pick off correct rom data bit from 8-bit word
-- for on screen character generation
rom_mux_output <= rom_data ( (CONV_INTEGER(NOT font_col(2 downto 0))));

END a;
```

Содержание постоянной памяти символов представлено в табл. 35.1 и должно быть размещено в файле Tcgrom с расширением mif (Tcgrom.mif).

35.2. Пример реализации на ПЛИС фирмы Altera работы клавиатуры PS/2

На плате расположен разъем PS/2 для подключения мыши или клавиатуры. Разъем подведен только к элементу FLEX.

Разъем содержит 6 выводов, включая землю, питание (VDD), линии данных и синхронизации. 2 линии не используются. Линия данных подходит к выводу 31 элемента, а линия синхронизации к выводу 30.

35.2.1. Протокол последовательной передачи данных PS/2

Скан-коды передаются последовательно, используя 11 бит, по двунаправленной линии данных.

Когда ни клавиатура, ни компьютер не отправляют данные, линии данных и синхронизации находятся в состоянии высокого уровня (High).

Передача кодов или команд осуществляется в следующей последовательности:

1. Стартовый бит ('0').
2. 8 бит данных.
3. Бит нечетности.
4. Стоповый бит ('1').

Во время передачи команды от клавиатуры происходят следующие события (рис. 35.7):

1. Клавиатура проверяет, что линии данных и синхронизации не активны. То есть находятся в состоянии High. Если они не активны, клавиатура устанавливает стартовый бит.
2. Клавиатура устанавливает низкий уровень сигнала (Low) на линии синхронизации примерно на 35 мкс.
3. Клавиатура синхронизирует остальные 10 бит импульсами приблизительно по 70 мкс на период.
4. Компьютер отвечает за распознавание стартового бита и прием последовательных данных. Последовательные данные — это 8 бит, за которыми следуют бит нечетности и стоповый бит. Если клавиатура хочет послать больше данных, они следует за 12-ым битом немедленно со следующим стартовым битом.

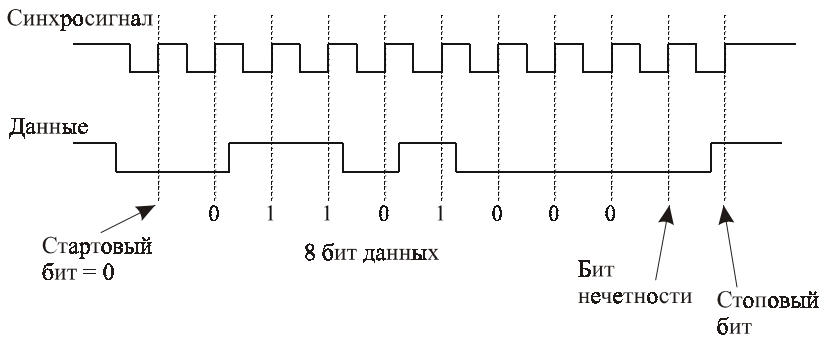


Рис. 35.7. Диаграмма передачи команды от клавиатуры

Для обеспечения надежной связи необходимо фильтровать импульсы синхронизации, т. к. в кабеле могут возникать помехи вследствие отраженного сигнала. Этот фильтр можно реализовать с помощью 8-битного регистра сдвига, исполь-

зуя для его синхронизации 25 МГц. Все биты этого регистра объединяются через схему AND, тогда на выходе схемы AND будем иметь отфильтрованный синхросигнал.

Передача команды от компьютера или чипа FLEX происходит в следующем порядке (рис. 35.8):

1. Система устанавливает '0' на линии синхронизации приблизительно на 60 мкс для запрета клавиатуре любых новых пересылок.
2. Система устанавливает '0' на линии данных и освобождает линию синхронизации.
3. Клавиатура генерирует синхросигнал для принятия всех последовательных бит команды.
4. Система отправляет 8 бит команды, бит нечетности и стоповый бит.
5. После стопового бита устанавливается '1' и освобождается линия данных.

Приведенный в листинге 35.3 модуль (рис. 35.9) принимает скан-коды от клавиатуры.

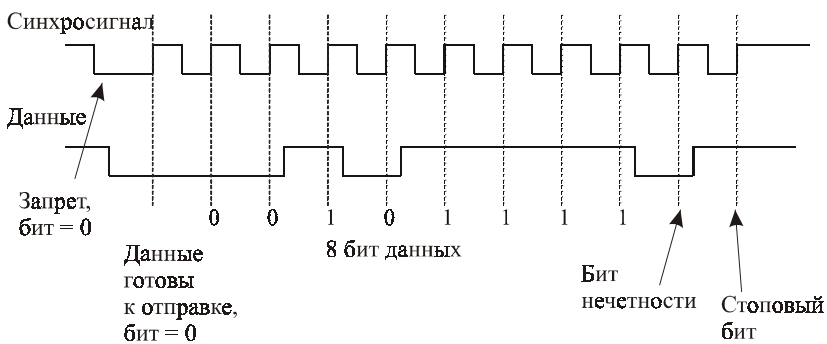


Рис. 35.8. Диаграмма передачи команды клавиатуре

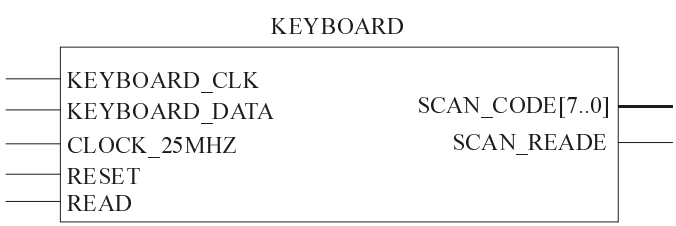


Рис. 35.9. Модуль `Keyboard` библиотеки `UPcore`

Листинг 35.3. keyboard

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY keyboard IS
       PORT(  keyboard_clk, keyboard_data, clock_25МГц ,
                     reset, read          : IN   STD_LOGIC;
                     scan_code            : OUT  STD_LOGIC_VECTOR(7 DOWNTO
0);
                     scan_ready           : OUT  STD_LOGIC);
END keyboard;

ARCHITECTURE a OF keyboard IS
       SIGNAL INCNT                       : std_logic_vector(3 downto 0);
       SIGNAL SHIFTIN                     : std_logic_vector(8 downto 0);
       SIGNAL READ_CHAR                   : std_logic;
       SIGNAL INFLAG, ready_set           : std_logic;
       SIGNAL keyboard_clk_filtered       : std_logic;
       SIGNAL filter                      : std_logic_vector(7 downto 0);
BEGIN

PROCESS (read, ready_set)
BEGIN
  IF read = '1' THEN scan_ready <= '0';
  ELSIF ready_set'EVENT and ready_set = '1' THEN
       scan_ready <= '1';
  END IF;
END PROCESS;



--This process filters the raw clock signal coming from the keyboard using
a shift register and two AND gates
Clock_filter: PROCESS
BEGIN
       WAIT UNTIL clock_25МГц'EVENT AND clock_25МГц= '1';
       filter (6 DOWNTO 0) <= filter(7 DOWNTO 1) ;
```

```
        filter(7) <= keyboard_clk;
        IF filter = "11111111" THEN keyboard_clk_filtered <= '1';
        ELSIF  filter= "00000000" THEN keyboard_clk_filtered <= '0';
        END IF;
END PROCESS Clock_filter;


--This process reads in serial data coming from the terminal
PROCESS
BEGIN
WAIT UNTIL (KEYBOARD_CLK_filtered'EVENT AND KEYBOARD_CLK_filtered='1');
IF RESET='1' THEN
        INCNT <= "0000";
        READ_CHAR <= '0';
ELSE
  IF KEYBOARD_DATA='0' AND READ_CHAR='0' THEN
        READ_CHAR<= '1';
        ready_set<= '0';
  ELSE
                    -- Shift in next 8 data bits to assemble a scan code
    IF READ_CHAR = '1' THEN
        IF INCNT < "1001" THEN
              INCNT                       <= INCNT + 1;
              SHIFTIN(7 DOWNTO 0) <= SHIFTIN(8 DOWNTO 1);
              SHIFTIN(8)                  <= KEYBOARD_DATA;
       ready_set <= '0';
                    -- End of scan code character, so set flags and exit loop
        ELSE
                    scan_code                   <= SHIFTIN(7 DOWNTO 0);
                    READ_CHAR                   <='0';
              ready_set                   <= '1';
              INCNT                       <= "0000";
        END IF;
     END IF;
   END IF;
 END IF;
END PROCESS;
END a;
```

35.3. Пример реализации на ПЛИС фирмы Altera работы мыши PS/2

Так же, как и клавиатура PS/2, мышь PS/2 использует синхронный двунаправленный последовательный протокол, описанный в предыдущем разделе. Формат пакета данных, передаваемых по этому протоколу, представлен в табл. 35.2.

Таблица 35.2. *Формат пакета данных*

Номер пакета	D7	D6	D5	D4	D3	D2	D1	D0
1	YV	XV	YS	XS	1	0	R	L
2	X7	X6	X5	X4	X3	X2	X1	X0
3	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

Где L = состояние левой кнопки (1 = кнопка нажата).

R = состояние правой кнопки (1 = кнопка нажата).

X0 — X7 = передвижение в направлении X.

Y0 — Y7 = передвижение в направлении Y.

XS, YS = знак перемещения (1 = отрицательный).

XV, YV = переполнение (1 = произошло переполнение).

Мышь работает в декартовой системе координат, то есть, двигаясь направо, имеем положительное смещение, двигаясь налево — отрицательное, вверх — положительное и вниз — отрицательное.

Величина передвижения курсора мыши по экрану монитора — функция скорости движения мыши. Чем больше скорость перемещения мыши, тем больше величина передвижения.

35.3.1. Модуль *Mouse* библиотеки *UPcore*

В отличие от модуля `Keyboard`, здесь реализован двунаправленный обмен данными, т. е. здесь можно отправлять команды для мыши (рис. 35.10). Модуль инициализирует мышь и отслеживает все посылки данных от мыши. На выходе имеем адрес курсора мыши и состояние кнопок.

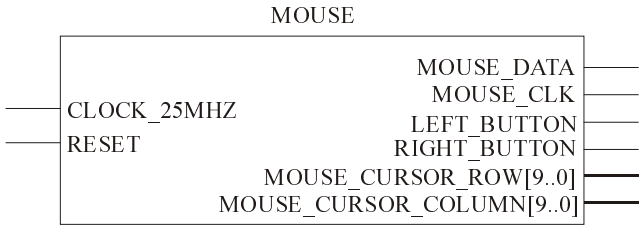


Рис. 35.10. Модуль `Mouse` библиотеки UPcore

Лабораторная работа. Конфигурирование предложенных проектов в ПЛБИС фирмы Altera

Цель работы

Данная работа предназначена для конфигурирования в ПЛБИС трех проектов, описанных в *главе 35*.

Программа работы

Конфигурирование ПЛБИС следует проводить по программе, представленной в предыдущей лабораторной работе.

Работа с монитором

Все файлы примеров находятся в папке VGA. Работать необходимо в системе MAX+PLUS II.

1. Откройте графический файл Vga_led.gdf, проанализируйте схему: представьте, что будет отображено на дисплее монитора. Проведите конфигурирование.
2. Откройте графический файл Vga_bar.gdf, проанализируйте схему: представьте, что будет отображено на дисплее монитора. Проведите конфигурирование.
3. Откройте графический файл Vga_test.gdf, он демонстрирует работу в текстовом режиме, проанализируйте схему: представьте, что будет отображено на дисплее монитора. Проведите конфигурирование.
4. Пример работы с графическим режимом. Откройте файл Ball.vhd. Проведите конфигурирование. Проанализируйте работу проекта.

Работа с клавиатурой

Все файлы примеров находятся в папке keyboard. Работать необходимо в системе MAX+PLUS II.

Откройте графический файл Kbd.gdf, проанализируйте схему. Проведите конфигурирование.

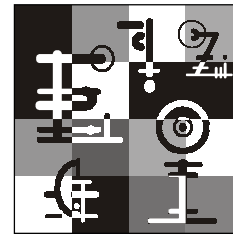
Работа с манипулятором "мышь"

Все файлы примеров находятся в папке mouse. Работать необходимо в системе MAX+PLUS II.

Откройте графический файл Mice.gdf. Проанализируйте схему. Проведите конфигурирование.

Задания на следующую лабораторную работу (самостоятельное выполнение индивидуального проекта)

1. Используя модули `VGA_SYNC` и `CHAR_ROM`, спроектировать устройство, с помощью которого можно выводить на дисплей заданные символы.
2. Изменить файл Ball.vhd таким образом, чтобы можно было перемещать шар вверх-вниз с помощью кнопок на плате.
3. Используя модули `VGA_SYNC`, спроектировать устройство, с помощью которого можно выводить на дисплей положение курсора, а также, используя кнопки мыши, менять цвет курсора.



Глава 36

Описание ПЛИС FLEX 10K фирмы Altera

Семейство FLEX 10K обладает следующими свойствами:

- устойчивая работа на частотах до 450 МГц;
- реализация на кристалле статической памяти и ПЗУ объемом до 24 Кбит;
- независимое использование комбинационной части и триггера каждого логического элемента;
- умножение тактовой частоты;
- работа в системах со смешанным напряжением питания (3,3, 5,0 В);
- реализация неограниченного числа циклов конфигурирования, в том числе "на лету", т. е. без выключения питания ПЛИС;
- использование четырех режимов работы выходных буферов: вход, выход, двунаправленный, открытый коллектор.

Кроме того, все ПЛИС этого семейства совместимы со стандартом шины PCI.

Двухуровневая структура ПЛИС семейства FLEX 10K включает:

- логические блоки (ЛБ), содержащие 8 логических элементов (ЛЭ) с табличной архитектурой и имеющие локальную программируемую матрицу соединений с непрерывной структурой связей;
- встроенные реконфигурируемые модули памяти, позволяющие реализовывать как статическую память и ПЗУ, так и сложные логические функции;
- глобальную программируемую матрицу соединений с одномерной непрерывной структурой (непрерывной по строкам и по столбцам);
- программируемые элементы ввода/вывода с синхронным триггером.

36.1. Общая характеристика семейства FLEX 10K

Перечень ПЛИС, входящих в семейство FLEX 10K (рис. 36.1), и их основные характеристики приведены в табл. 36.1.

Таблица 36.1. *ПЛИС семейства FLEX10K*

Параметры	**EPF10K10 EPF10K10A**	**EPF10K20**	**EPF10K30 EPF10K30A**	**EPF10K40**	**EPF10K50 EPF10K50V**
Логическая емкость, количество эквивалентных вентилей	10 000	20 000	30 000	40 000	50 000
Число логических элементов	576	1152	1728	2304	2880
Число логических блоков	72	144	216	288	360
Встроенные блоки памяти	3	6	6	8	10
Объем памяти (бит)	6144	12 288	12 288	16 384	20 480
Максимальное число выводов пользователя	150	189	246	189	310

Продолжение таблицы 36.1

Параметры	**EPF10K70**	**EPF10K100 EPF10K100A**	**EPF10K130V**	**EPF10K250A**
Логическая емкость, количество эквивалентных вентилей	70 000	100 000	130 000	250 000
Число логических элементов	3744	4992	6656	12 160
Число логических блоков	468	624	832	1520
Встроенные блоки памяти	9	12	16	20
Объем памяти (бит)	18 432	24 576	32 768	40 960
Максимальное число выводов пользователя	358	406	470	470

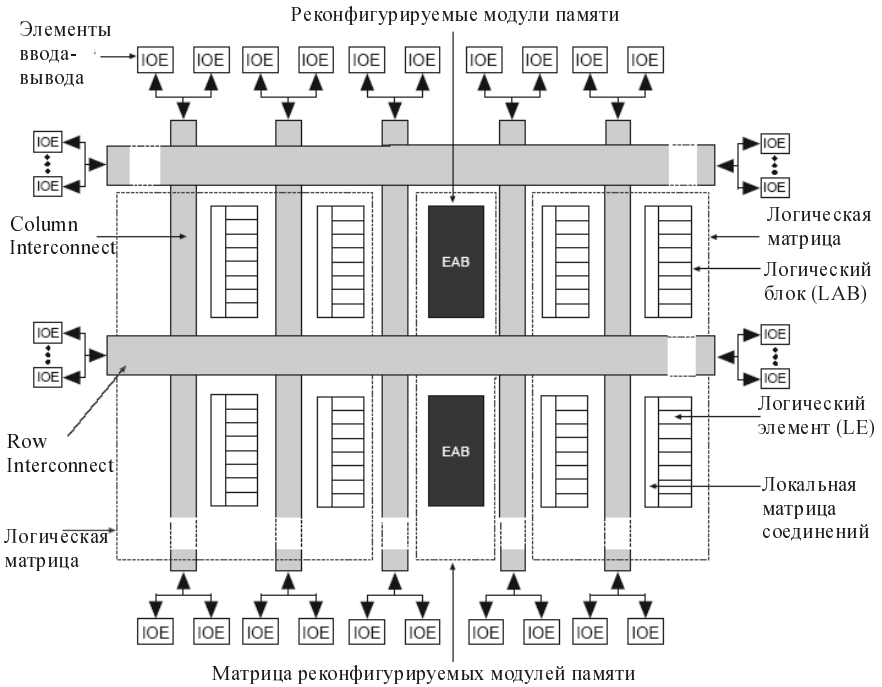


Рис. 36.1. Общая структура FLEX 10K

36.1.1. Основные компоненты структуры FLEX 10K

В основные компоненты структуры входят:

- логические блоки (LAB), содержащие 8 логических элементов (LE) и локальную программируемую матрицу соединений (Local Interconnect);
- реконфигурируемые модули памяти (EAB);
- глобальная программируемая матрица соединений, разделенная на строки и столбцы (Column Interconnect, Row Interconnect);
- межблочные цепи каскада и переноса;
- программируемые элементы ввода/вывода (IOE).

Логические блоки (ЛБ) организованы в виде матрицы. В центре каждой строки матрицы ЛБ расположен реконфигурируемый модуль памяти (РМП). Столбцы и строки матрицы ЛБ разделены столбцами и строками глобальной программируе-

мой матрицы соединений (ГПМС). При этом число строк ГПМС соответствует числу строк матрицы ЛБ, а следовательно, и числу РМП.

Элементы ввода/вывода (ЭВВ) подсоединены непосредственно к каналам строк и столбцов ГПМС.

Глобальные шины распространения управляющих сигналов обеспечивают минимальный сдвиг фронтов тактовых сигналов, поступающих на входы синхронизации триггеров ЛБ и ЭВВ, а также минимальную задержку распространения сигналов управления. Источником сигналов глобальной шины управляющих сигналов являются шесть специализированных входов ПЛИС. А источником сигналов глобальной шины управления вводом/выводом могут служить специализированные входы ПЛИС и выходы первых ЛЭ в ЛБ.

Глобальная программируемая матрица соединений

Глобальная программируемая матрица соединений, обеспечивающая возможность передачи сигналов между ЛБ и элементами ввода/вывода, представляет собой набор непрерывных, проходящих через всю ПЛИС, вертикальных и горизонтальных каналов, сгруппированных в шины, т. е. в столбцы и строки ГПМС.

Некоторые каналы строк ГПМС разделены на две части. Это с одной стороны позволяет существенно увеличить ресурсы разводки ПЛИС, поскольку для соединения близко расположенных ЛБ могут быть использованы каналы половинной длины, а с другой стороны обеспечивает сохранение преимуществ непрерывной структуры соединений перед сегментированной.

Логический блок

Логический блок содержит (рис. 36.2):

- 8 логических элементов (LE [8...1]);
- локальную программируемую матрицу соединений (ЛПСМ);
- цепи каскадов и переносов;
- локальную четырехканальную шину управляющих сигналов (LAB Control Signals);
- восьмиканальную шину сигналов обратной связи;
- схему коммутации;
- программируемый мультиплексор.

Сигналы на вход ЛПМС поступают как со строки ГПМС, так и с выходов каждого из 8 логических элементов, входящих в состав блока. Сигналы с выхода блока поступают как на строку, так и на столбец ГПМС.

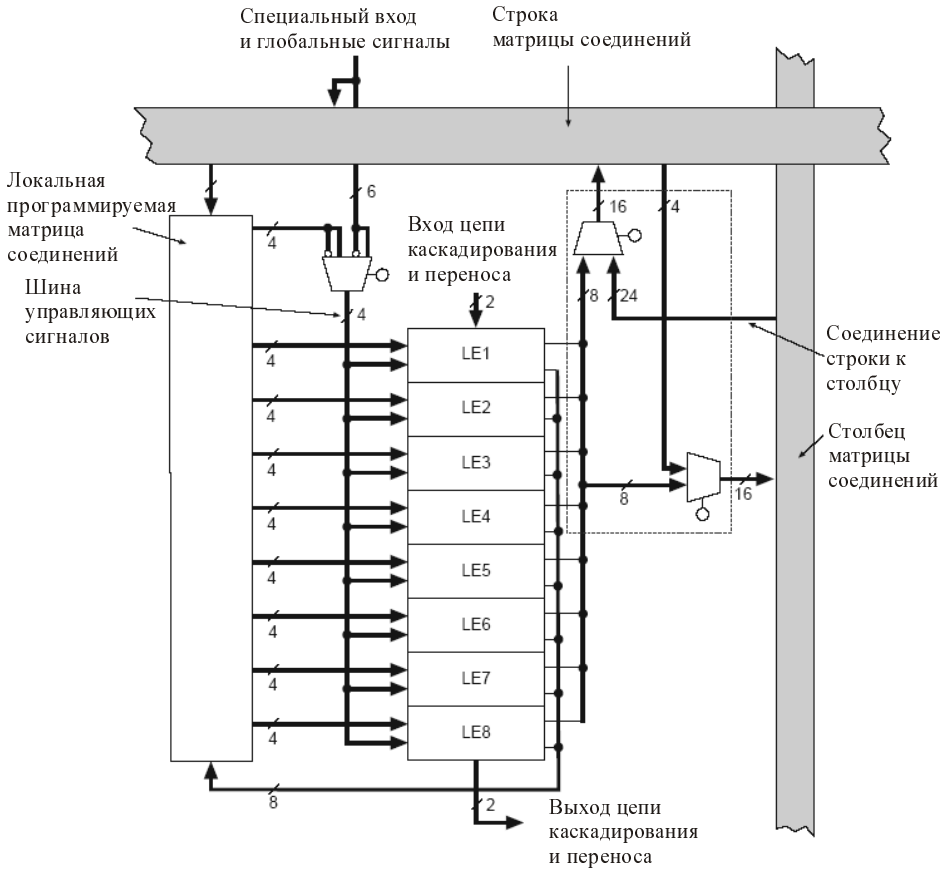


Рис. 36.2. Структура логического блока

Логический элемент

Основные компоненты логического элемента (рис. 36.3):

- четырехвходовая таблица перекодировки (LUT);
- цепь переноса (Carry Chain);
- цепь каскадного наращивания (Cascade Chain), позволяющая реализовать либо функцию "И", либо "ИЛИ";
- синхронный триггер;
- схема управления установкой/сбросом триггера (Clear/Reset Logic);
- набор программируемых мультиплексоров.

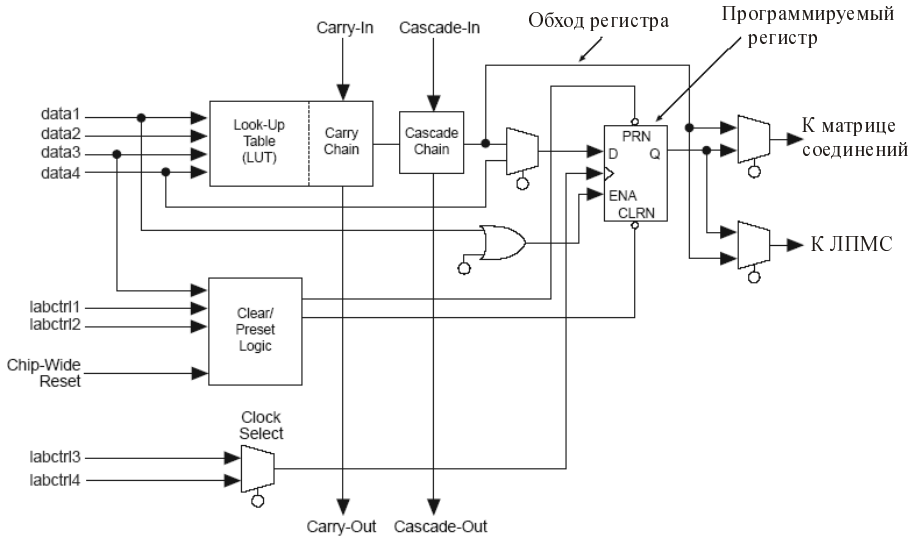


Рис. 36.3. Структура логического элемента

Комбинационную часть ЛЭ образуют:

- четырехвходовая таблица перекодировки и схема каскадного наращивания. Дополнительные логические ресурсы обеспечивает схема управления установкой/сбросом триггера;
- синхронный триггер, тип которого программируется (D, T, JK, RS), имеет входы асинхронного сброса (CLRn) и установки (PRn) с активным низким уровнем, а также вход разрешения записи (ENA).

Структура ЛЭ позволяет как совместно, так и независимо использовать его комбинационную часть и триггер.

При совместном использовании выход комбинационной части соединяется через мультиплексор с информационным входом триггера, а выходной сигнал триггера поступает либо на ГПМС, либо на ЛПМС, либо и туда, и туда одновременно.

При независимом использовании на информационный вход триггера поступает сигнал с входа data4, а выходные сигналы триггеров и комбинационной части могут подаваться либо в ГПМС, либо в ЛПМС.

Режим каскадного наращивания

Четырехвходовая таблица перекодировки позволяет реализовать любую логическую функцию от четырех аргументов.

Для построения логических функций от большего числа аргументов целесообразно использование режима каскадного наращивания.

Цепь каскадного наращивания, содержащая не более восьми ЛЭ, располагается в пределах одного ЛБ. Более длинные цепи организуются путем соединения ЛБ, расположенных в одной строке. Эти соединения осуществляются через межблочные цепи каскадов и переносов.

Для ПЛИС FLEX 10K максимальная длина цепи каскадов составляет 6 ЛБ, что позволяет реализовать логическую функцию от $N = 6 * 8 * 4 = 192$ аргументов.

Реконфигурируемые модули памяти

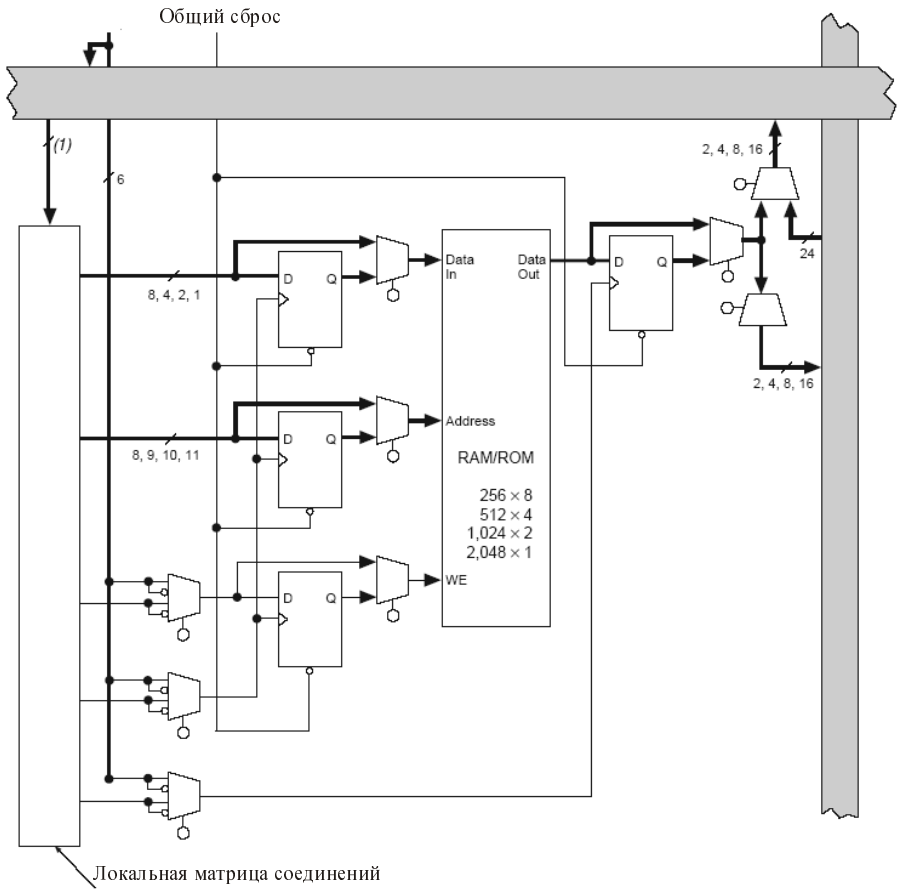


Рис. 36.4. Структура реконфигурируемого модуля памяти

Встроенный блок памяти (рис. 36.4) представляет собой оперативное запоминающее устройство емкостью 2048 (4096) бит и состоит из:

- локальной матрицы соединений;

- модуля памяти;
- набора синхронных регистров;
- набора программируемых мультиплексоров.

Сигналы на вход ЛПМС поступают со строки ГПМС. Тактовые и управляющие сигналы поступают с глобальной шины управляющих сигналов.

Выход модуля памяти может быть скоммутирован как на строку, так и на столбец ГПМС.

Модуль памяти объемом 2048 бит позволяет создавать запоминающие устройства следующих конфигураций: 256x8, 512x4, 1024x2, 2048x1. Он имеет: вход данных (Data in); вход адреса (Address); вход разрешения записи (WE); выход данных (Data out).

Регистры, расположенные на входе/выходе данных и входе адреса модуля памяти, реализованы на синхронных D-триггерах, имеют гибкую, программируемую структуру и обеспечивают возможность буферизации шин разрядностью 8, 4, 2, 1 бит.

Разрядность шин входа/выхода данных и входа адреса модуля памяти, а также соответствующих синхронных регистров устанавливается в зависимости от используемой конфигурации модуля памяти.

Набор программируемых мультиплексоров позволяет задавать синхронный либо асинхронный режимы записи в память.

Гибкость внутренней структуры модуля памяти определяет возможность использования его не только для построения блоков статической памяти (SRAM, FIFO, двухпортовой SRAM) и ПЗУ, но и для реализации (табличной реализации) арифметико-логических устройств: сумматоров, умножителей, делителей, кодеров/декодеров и конечных автоматов, работающих на частотах до 105 МГц.

Элементы ввода/вывода

Элемент ввода/вывода (ЭВВ), соединяющий канал строки/столбца ГПМС с выводом ПЛИС, позволяет осуществлять ввод и вывод бита данных, в том числе с временным его хранением, реализовывать двунаправленный режим передачи и, кроме того, работать в режиме выхода с открытым коллектором.

В состав элемента ввода/вывода входит:

- выходной буфер;
- синхронный триггер D-типа;
- набор программируемых мультиплексоров.

Элемент ввода/вывода соединяет канал строки или столбца ГПМС с выводом микросхемы. ЭВВ позволяет осуществить ввод/вывод бита данных с различными скоростями, временное хранение данных, эмуляцию открытого коллектора.

Выходной буфер имеет:

- вход установки скорости переключения;
- вход установки режима "Открытый коллектор";
- вход разрешения работы.

Скорость переключения буфера (низкая, высокая) устанавливается на этапе программирования ПЛИС.

Низкая скорость переключения обеспечивает уменьшение импульсных шумов в системе, но приводит к дополнительной (порядка 4.5 нс) задержке в формировании фронтов выходного сигнала.

Режим "Открытый коллектор", при котором выход буфера является выходом с открытым коллектором, устанавливается на этапе программирования ПЛИС.

На вход разрешения работы выходного буфера может поступать:

- логический нуль, при этом выход буфера находится в Z-состоянии (элемент ввода/вывода работает в режиме ввода данных);
- логическая единица, при этом буфер открыт (элемент ввода/вывода работает в режиме вывода данных);
- сигнал с одного из каналов глобальной шины управления вводом/выводом или с канала строки/столбца ГПМС, обеспечивающий работу элемента ввода/вывода в режиме двунаправленной передачи.

Синхронный D-триггер, имеющий вход разрешения записи и вход асинхронного сброса с активным низким уровнем, позволяет организовать временное хранение как передаваемого, так и принимаемого элементом ввода/вывода бита данных.

Источником сигнала синхронизации триггера может служить один из двухтактовых управляющих сигналов для глобальной шины либо один из двухтактовых сигналов управления вводом/выводом для глобальной шины.

Набор программируемых мультиплексоров обеспечивает возможность выбора:

- источников управляющих сигналов;
- полярности и источников входного сигнала триггера и выходного буфера;
- источника сигнала, передаваемого в ГПМС.

36.2. Конфигурирование и реконфигурирование ПЛИС семейства FLEX 10K

Программируемые элементы ПЛИС семейства FLEX 10K, осуществляющие настройку ЛЭ и ЭВВ на выполнение требуемых функций и управляющие их подсое-

динением к каналам ГПМС и ЛПМС, реализованы на статических запоминающих ячейках (по SRAM-технологии).

Поэтому ПЛИС данного семейства:

- требуют выполнения конфигурации — (загрузки данных, конфигурирующих данных о состоянии всех программируемых элементов) после каждого отключения питания;
- позволяют неограниченное число раз изменять их конфигурацию;
- обеспечивают возможность выполнения реконфигурации (изменения конфигурации работающей в системе ПЛИС без выключения ее питания), в том числе и частичной реконфигурации.

Конфигурирующие данные могут храниться в:

- последовательных ПЗУ фирмы ALTERA;
- запоминающем устройстве системы, в которую интегрирована ПЛИС.

Лабораторная работа. Программирование (конфигурирование) индивидуального проекта в ПЛБИС фирмы Altera

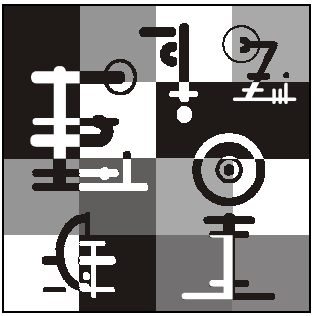
Цель работы

Данная работа предназначена для конфигурирования в ПЛБИС индивидуального проекта, задание на который было описано в предыдущей лабораторной работе.

Программа работы

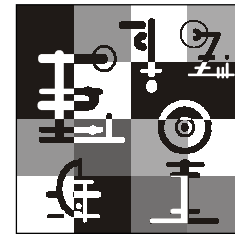
Работа включает в себя выполнение индивидуального задания: его конфигурирование и тестирование.

Конфигурирование ведется по программе, описанной в лабораторной работе в *главе 34*.



Часть IX

Средства проектирования фирмы Altera для цифровых устройств большой интеграции



Глава 37

Описание программируемых логических больших ИС (ПЛБИС) APEX фирмы Altera

Основные характеристики семейства ПЛСБИС APEX20K (табл. 37.1):

- совмещение матричной, вентильной архитектуры и блоков памяти в одном кристалле;
- тип реализуемой памяти: RAM, ROM, FIFO (в т. ч. двухпортовое), CAM (Content-Adressable Memory — память, адресуемая по содержимому);
- гибкое управление тактовыми сигналами;
- совместимость со стандартом PCI от 33 до 66 МГц и от 32 до 64 бит;
- напряжение питания 2,5 В (для микросхем EP20K) и 1,8 В (для микросхем EP20KE);
- усовершенствованная структура внутренних связей;
- индивидуальный сигнал управления Z-состоянием для каждого выхода;
- поддержка "горячего" включения;
- технология SRAM с загрузкой конфигурации при включении питания;
- загрузка конфигурации: с помощью последовательного ПЗУ, процессора/контроллера, через ByteBlaserMV/MasterBlaster;
- программная поддержка — САПР QUARTUS.

Таблица 37.1. *Основные характеристики ПЛИС семейства APEX20K фирмы Altera*

Параметр	**EP20K60E**	**EP20K100 EP20K100E**	**EP20K160E**	**EP20K200 EP20K200E**	**EP20K300E**
Вентили (макс.)	162K	263K	404K	526K	728K
Вентили (типовое)	60K	100K	160K	200K	300K
Логич. элементы LE	2560	4160	6400	8320	11520
Макроячейки LCELL	256	416	640	832	1152
Системные блоки (ESB)	16	26	40	52	72
Объем памяти, бит	32768	53248	81920	106496	147456
Выводы (макс.)	204	252	316	382	408

Продолжение таблицы 37.1

Параметр	**EP20K400 EP20K400E**	**EP20K600E**	**EP20K1000E**	**EP20K1500E**
Вентили (макс.)	1052K	1573K	1772K	2524K
Вентили (типовое)	400K	600K	1000K	1500K
Логич. элементы LE	16640	24320	38400	54720
Макроячейки LCELL	1664	2432	2560	3648
Системные блоки (ESB)	104	152	160	228
Объем памяти, бит	212992	311296	327680	466944
Выводы (макс.)	502	624	716	858

37.1. Архитектура ПЛБИС семейства APEX20K

Архитектура ПЛБИС или ПЛИС семейства APEX20K сочетает в себе достоинства FPGA ПЛИС с их таблицами перекодировок, входящими в состав логического элемента, логику вычисления СДНФ (Совершенная дизъюнктивная нормальная форма), характерную для ПЛИС CPLD, а также встроенные модули памяти (рис. 37.1).

Отличительной особенностью ПЛИС семейства APEX20K является объединение логических блоков (ЛБ) в мегаблок (MegaLAB), имеющий собственную непрерывную матрицу соединений (MegaLAB Interconnect) (рис. 37.2).

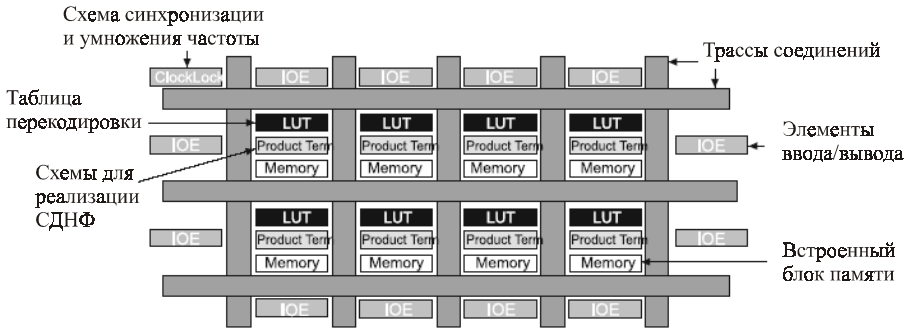


Рис. 37.1. Архитектура ПЛИС семейства APEX20K

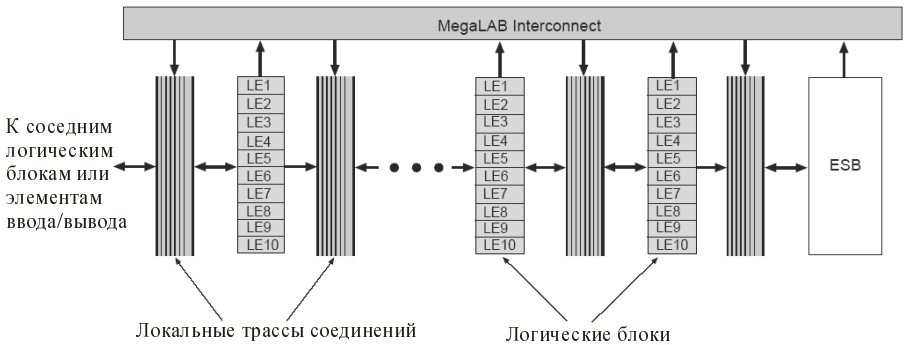


Рис. 37.2. Структура мегаблока ПЛИС семейства APEX20K

Такая организация соединений позволяет выделить дополнительные ресурсы для трассировки, кроме того, в каждом мегаблоке может быть полностью реализована та или иная функционально законченная часть системы, что позволяет при ее модификации не перепрограммировать этот участок и тем самым сохранить все заданные временные параметры. Также такая организация ПЛИС позволяет разумнее организовывать соответствующее программное обеспечение, в том числе создать средства коллективной работы над проектом.

Каждый ЛБ состоит из 10 ЛЭ, имеющих структуру, показанную на рис. 37.3. Каждый ЛЭ имеет возможность коммутации на два столбца глобальной матрицы соединений (ГМС). Матрица соединений мегаблока (МСМ) коммутируется на локальную матрицу соединений (ЛМС) ЛБ и на строки ГМС.

ЛЭ ПЛИС семейства APEX20K имеет возможность формирования управляющих сигналов триггера как с помощью глобальных и локальных сигналов, так и используя сигналы мегаблока. ЛЭ может быть сконфигурирован в нормальном, арифметическом или счетном режиме, допускает каскадирование и цепочечный перенос.

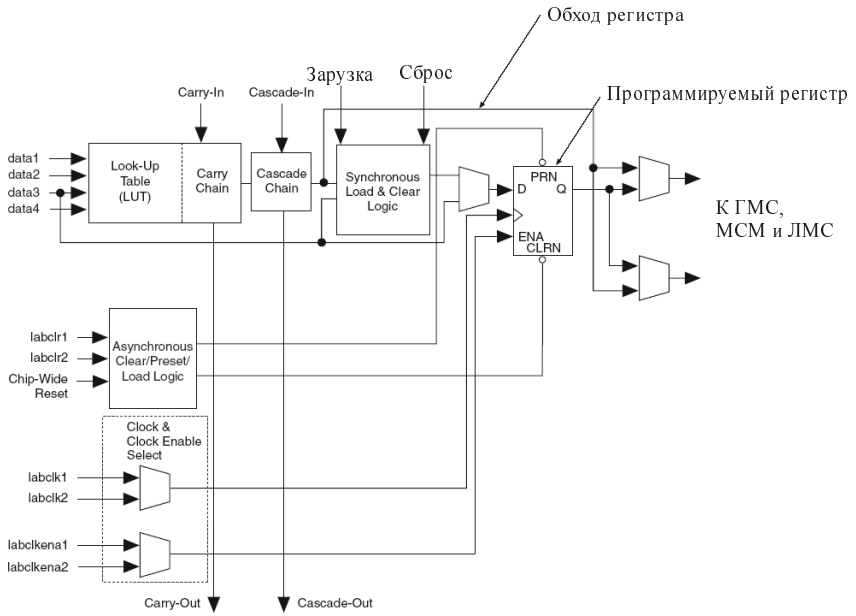


Рис. 37.3. Логический элемент ПЛИС семейства APEX20K

Отличительной особенностью ПЛИС семейства APEX20K являются системные блоки памяти (СБП, ESB — Embedded System Block).

Особенностью СБП является то, что он может быть сконфигурирован как контекстно-адресуемая память (т. е. память, адресуемая по ее содержимому) или как двухпортовая память, что существенно расширяет возможности применения.

Лабораторная работа. Создание модели процессора Nios в среде QUARTUS

Цель работы

Научиться создавать проекты с процессором Nios необходимой конфигурации.

Программа работы

Модель процессора создается в среде QUARTUS с помощью встроенного модуля Sopc Builder.

Данный модуль позволяет сформировать ядро, необходимую периферию и сгенерировать прошивку для конфигурации. (Архитектура процессора Nios будет изложена в *главе 39*.)

Создание модели производится в следующем порядке:

1. Создайте новый проект. Выберите **Tools** | **MegaWizard Plug-In Manager** (Инструменты | Управление работой мастера мегафункций). Убедитесь, что в следующем окне будет выбран пункт создания новой мегафункции пользователя (**Create a new custom megafunction**), и щелкните **Next** (Далее). В крайнем левом окне нового диалогового окна, оно называется **Available Megafunctions**, выберите **Altera SOPC Builder 2.7**. Выберите **Verilog HDL** для типа файла, введите имя и папку для расположения выходного файла. Щелкните **Next**. Запустится **Altera SOPC Builder**.
2. Из левого окна на вкладке **System Contents** (Содержание системы) выберите **Altera Nios 2.2 CPU** (**Altera Nios 2.0 CPU**) и щелкните **Add** (Добавить). Поскольку выбирается архитектура процессора Nios-32, выберите **Standard features** | **Average LE usage** (Стандартные параметры | Среднее число используемых логических элементов) и щелкните **Finish**. Переименуйте процессор, для чего щелкните правой кнопкой мыши по текущему имени и выберите пункт **Rename** (Переименовать). Назовите его **cpu**.
3. Добавьте **On-Chip Memory (RAM or ROM)**. Выберите **ROM (read only)** с шириной данных 32 бита и размером общей памяти 1 Кбайт, щелкните **Next**. Выберите **GERMS Monitor** и нажмите кнопку **Finish**. Переименуйте память на **boot_rom**.
4. Добавьте **UART (RS-232 serial port)**. Выберите **Скорость в бодах** (бит в сек.) **115200** и убедитесь, что опция **Baud rate can be changed by software** (Скорость передачи данных может быть изменена программно) не выбрана. Выберите **None** для четности, 8 бит данных и 2 стоповых бита. Нажмите кноппку **Finish**. Переименуйте это внешнее устройство в **uart1**.
5. Добавьте **PIO (Parallel I/O)**. Выберите разрядность 16 бит и **Output ports only** (Только выходной порт). Нажмите **Finish**. Назовите это внешнее устройство **seven_seg_pio**.
6. Добавьте **Interval Timer**. Из предварительного списка конфигураций (**Preset Configurations list**) выберите **Full-featured (v1.0-compatible)** и щелкните **Finish**. Назовите это внешнее устройство **timer1**.
7. Добавьте **PIO**. Выберите ширину 2 бита с двунаправленными портами с третьим состоянием (**bi-directional (tri-state) ports**) и щелкните **Next**. Отключите опцию **Synchronously capture** (Синхронный сбор данных). Отключите опцию **Generate IRQ** (Генерация запроса на прерывание). Нажмите **Finish**. Назовите это внешнее устройство **led_PIO**.

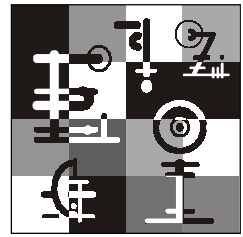
8. Добавьте **PIO**. Выберите ширину 12 бит с **Input ports only** (Только входной порт) и щелкните **Next**. Выберите **synchronously capture** (синхронный сбор данных). Выберите **Either Edge** (Любой фронт). Отметьте **Generate IRQ**. Выберите **Edge** и щелкните **Finish**. Назовите это внешнее устройство **button_pio**.
9. Добавьте **Avalon Tri-State Bridge**. Выберите **Registered** для поступающих сигналов и щелкните **Finish**. Переименуйте это внешнее устройство в **off_chip_bus**.
10. Добавьте **SRAM** (один или два чипа IDT71V016). Выберите **32 bits wide / 256KBytes (Nios 32 default)**, щелкните **Finish**. Назовите эту память **ext_ram**.
11. Добавьте **Flash Memory**. Установите разрядность шины адреса — **19 bits /1M bytes Nios 32 default** и щелкните **Finish**. Назовите эту память **ext_flash**.
12. Отредактируйте базовые адреса и значения IRQ согласно табл. 37.2. Чтобы исправить значение, нужно просто нажать по требуемому полю. После того как закончите, щелкните **Next**.

Таблица 37.2. *Значения базовых адресов и IRQ*

Base	**End**	**IRQ**
0x00000000	0x000003FF	
0x00000400	0x0000041F	26
0x00000420	0x0000042F	
0x00000440	0x0000045F	25
0x00000460	0x0000046F	
0x00000470	0x0000047F	27
0x00040000	0x0007FFFF	
0x00100000	0x001FFFFF	

13. На вкладке **More "cpu" Settings** выберите **boot_rom** как память, читаемую по сбросу и расположенную по адресу 0x0. Выберите **ext_ram**, как место для хранения таблицы векторов (256 байт), расположенной со смещением 0x3ff00. Выберите **ext_ram** для памяти программ и памяти данных. Выберите **uart1** для связи с хостом и для отладки. Введите любое сообщение в поле **Boot ID Message**. Щелкните **Next**.

14. Выделите все опции, кроме **Simulation**. Выберите **APEX 20KE** в списке семейств микросхем (**Device Family**).
15. Теперь щелкните **Generate**. MegaWizard сгенерирует параметризируемую систему процессора Nios, готовую для трансляции в QUARTUS II. После того как MegaWizard закончит работу, щелкните **Exit**.
16. Теперь процессор Nios сгенерирован, и мы должны добавить его к нашей схеме. Дважды щелкните в любом месте окна схемы. Это вызовет диалоговое окно выбора символа. Напечатайте `Nios` в окне названия символа и нажмите **OK**.
17. Сохраните полученную схему, для чего выберите пункт **Save** в меню **File**. Выберите **Start Compilation** в меню **Processing**. Щелкните **OK**, как только компиляция завершится. Результатом компиляции проекта будет файл, готовый к загрузке в макетную плату.



Глава 38

Excalibur — набор разработчика фирмы Altera

Набор разработчика Excalibur является готовым фирменным решением для изучения и разработки систем на основе конфигурируемого процессора Nios и содержит все необходимые инженерам аппаратные и программные средства для разработки высокопроизводительных систем в ПЛИС фирмы Altera.

Этот набор содержит:

- Nios — конфигурируемое процессорное RISC-ядро и периферию;
- компилятор и отладчик GNUPro от фирмы Cygnus;
- САПР программируемой логики QUARTUS;
- загрузочный кабель ByteBlasterMV;
- отладочная плата;
- проекты-образцы различных конфигураций систем на кристалле.

38.1. Описание отладочной платы

Отладочная плата (рис. 38.1) содержит все необходимое для построения и отладки законченной системы и включает:

- ПЛИС большой емкости EP20K200E семейства APEX для реализации проектов;
- два модуля памяти SRAM по 512 Кбайт;
- Flash-память 8 Мбайт;
- ПЛИС конфигурационной логики EPM7128AE;
- два разъема JTAG;
- последовательный порт RS-232;
- четыре определяемых пользователем кнопки;

- двойной 7-сегментный светодиодный индикатор;
- восемь определяемых пользователем светодиодов;
- символьный жидкокристаллический индикатор;
- блок питания 9 В;
- разъемы для подключения устройств пользователя.

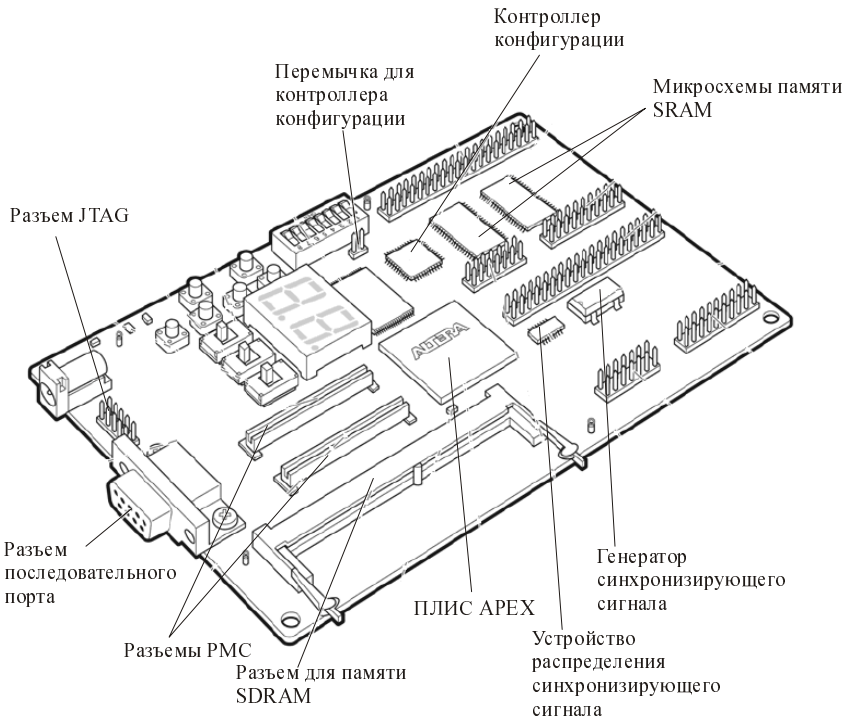


Рис. 38.1. Внешний вид платы

38.1.1. Цепь JTAG

Разъем JP3 — интерфейс JTAG с 10 выводами, совместимый с программаторами Altera ByteBlasterMV и MasterBlaster.

Связь по интерфейсу JTAG может использоваться для любой из трех следующих целей.

- Программное обеспечение QUARTUS может конфигурировать устройство APEX новыми данными из файла конфигурации с расширением sof с использованием поддерживаемых программаторов.

- Программное обеспечение QUARTUS или MAX+PLUS может перепрограммировать контроллер конфигурации новым файлом с расширением pof.
- Программное обеспечение пользователя, запускаемое с хоста, может установить последовательную связь по интерфейсу JTAG с картой, установленной в разъемы PMC (PCI Mezzanine Card), если карта использует сигналы JTAG, которые являются частью стандарта IEEE-1386.

Цепь JTAG на плате проходит через три переключателя и может соединять в цепь устройства в следующих комбинациях:

- SW8 — устройство APEX;
- SW9 — контроллер конфигурации;
- SW10 — карта, включенная в PMC-разъем, если она присутствует.

Для каждого переключателя с двумя положениями приведены обозначения Connect/Bypass (Соединить/Обойти), что соответствует состоянию устройства: включено ли оно в цепь JTAG или исключено.

Связь JTAG обычно используется, чтобы загрузить конфигурацию пользователя из файла в чип устройства APEX во время разработки и отладки проекта. В этом случае, будет наиболее удобно оставить SW8 в положении Connect, а SW9 и SW10 в положении Bypass.

Контроллер конфигурации запрограммирован изготовителем. Проект для контроллера конфигурации поставляется вместе с программным обеспечением процессора Nios.

38.1.2. Контроллер конфигурации

Контроллером конфигурации является ПЛИС EPM7064. Он запрограммирован изготовителем так, чтобы производить конфигурацию устройства APEX файлом данных, запасенных во Flash-памяти при включении питания или после подачи сигнала сброса кнопкой SW2. Программное обеспечение QUARTUS может вырабатывать файлы конфигурации, которые являются непосредственно файлами загрузки, подходящими для хранения во Flash-памяти, как данные конфигурации. Новые файлы могут быть записаны во Flash-память с помощью программного обеспечения, исполняемого на процессоре Nios.

Конфигурация изготовителя и конфигурация пользователя

Контроллер конфигурации может управлять двумя отдельными конфигурациями устройства APEX, находящимися во Flash-памяти.

Эти две конфигурации традиционно называются конфигурацией изготовителя, или фабричной конфигурацией, и конфигурацией пользователя.

При включении питания или после сброса контроллер конфигурации будет пытаться загрузить устройство APEX данными конфигурации пользователя. Если конфигурация пользователя отсутствует, контроллер загрузит устройство APEX данными конфигурации изготовителя.

Конфигурации должны находиться во Flash-памяти в установленных адресах.

В табл. 38.1 показано, как будет выглядеть содержимое Flash-памяти.

Таблица 38.1. *Содержимое Flash-памяти*

0x100000-0x17FFFF	512 Кбайт	Инструкции Nios и энергонезависимая область данных
0x180000-0x1BFFFF	256 Кбайт	Определяемая пользователем конфигурация для APEX-устройства
0x1C0000-0x1FFFFF	256 Кбайт	Конфигурация изготовителя

38.1.3. Перемычка JP2

Перемычка JP2 определяет поведение контроллера конфигурации. Если перемычка присутствует на JP2, контроллер конфигурации игнорирует конфигурацию пользователя и всегда загружает устройство APEX конфигурацией изготовителя.

38.1.4. Кнопка SW2: Reset

Когда кнопка SW2 нажата, значение логического нуля с ее выхода подается на контроллер U7, выполняющий сброс при включении питания, и контроллер устанавливает сигнал сброса, такой же, как и при включении питания.

38.1.5. Кнопка SW3: Clear

Результат нажатия этой кнопки будет зависеть от того, как сконфигурировано устройство APEX. В образцовом проекте Nios сигнал с SW3 используется как вход СБРОСА для CPU, процессор по этому сигналу сбросится и начнет исполнять программу из адреса начальной загрузки — 0.

38.1.6. Источник питания

Плата питается от 9 В нерегулируемого источника питания. В разъеме на центральный контакт подается отрицательное напряжение. На самой плате специальная схема питания вырабатывает напряжения 5, 3,3 и 1,8 В.

Лабораторная работа. Конфигурирование на ПЛСБИС APEX проекта, содержащего процессор Nios, и программирование процессора

Цель работы

Данная работа предназначена для ознакомления с конфигурированием на ПЛСБИС APEX сложных проектов, содержащих процессор Nios, и программированием этого процессора.

Программа работы

Конфигурирование проекта

Откройте оболочку Nios SDK Shell, для этого найдите и запустите файл Nios SDK Shell.bat. Введите режим терминала командой `nr -t`. При использовании порта com2 вы должны будете использовать команду `nr -p com2 -t`.

Вернитесь к окну QUARTUS и откройте файл с расширением sof. Для этого из меню **File** выберите пункт **Open** и из файлов типа Programming Files (**cdf**, **sof**, **pof**) — нужный файл, щелкните **Open**.

Когда откроется окно программирования, отметьте **Program/Configure**. Щелкните **Start**, чтобы загрузить проект.

Вернитесь в оболочку Nios SDK. В окне оболочки должен быть показан результат прохождения теста по программе Peripheral Test. Этот тест запускается, как только проект был загружен в устройство. Нажмите клавишу <Q>, чтобы выйти из этого режима. Теперь вы должны увидеть ваше сообщение начальной загрузки — Boot ID.

Чтобы возвратиться к программе Peripheral Test, нажмите кнопку **Clear** на плате или напечатайте `g140000` в оболочке.

Выполнение пользовательских программ

Программа для процессора может быть написана на языках C/C++, ассемблере.

Для компиляции программы, формирования файла прошивки и загрузки программы в ОЗУ используется консольная программа Cygwin. Она установлена в том же каталоге, что и модуль `SOPC_Builder`, и содержит набор для разработки программ: компилятор gcc, g++, отладчик, ассемблер и т. д.

Необходимо запустить Cygwin и перейти в каталог на диске, где расположен исходный файл.

Компиляция файла выполняется командой `nios-build *.c`. В том же каталоге будет создан выходной файл с тем же именем, но расширением srec.

После компиляции можно выполнить загрузку сформированного файла в ОЗУ командой `nios-run *.srec`. Программа запустится на исполнение на процессоре.

Пример запуска пользовательской программы

Необходимо запустить оболочку Nios SDK и командой `cd` изменить имя текущего каталога на необходимое.

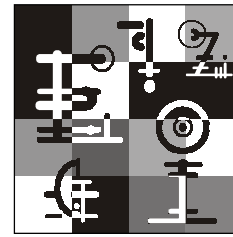
Компиляция файла с расширением с производится командой:

```
nb lcd_demo1.c
```

Затем следует загрузить полученный в результате компиляции файл и запустить его на выполнение с помощью команды:

```
nr led_demo1.srec
```

На жидкокристаллическом индикаторе должно появиться тестовое сообщение о нормальном выполнении программы, которое заранее запрограммировал пользователь.



Глава 39

Встраиваемый процессор Nios

Nios — это встраиваемый процессор общего назначения, с перестраиваемой конфигурацией, который легко программируется в ПЛИС фирмы Altera. При этом процессор занимает небольшую часть программируемой логики ПЛИС, оставляя большую ее часть для размещения там периферийных устройств и пользовательских функций.

Процессор Nios является RISC-процессором (Reduced Instruction Set Computer — ЭВМ с сокращенным набором команд) с конвейерной обработкой команд, большинство команд которого выполняется в единственном цикле синхронизации. Система команд Nios ориентирована на встроенные прикладные программы.

Процессорная система Nios включает в себя процессор, периферийные устройства, память и размещается на одной микросхеме (рис. 39.1). Термин "процессорная система Nios" относится к конфигурируемому ядру процессора, набору встроенной периферии, встроенных модулей памяти и интерфейса внешней памяти, реализованных на одной ПЛИС фирмы Altera.

Пользователь может изменять конфигурацию системы при создании собственных проектов добавлением или удалением возможностей микропроцессора и периферии.

39.1. Архитектура процессора Nios

Архитектура Nios (рис. 39.2) определяет следующие видимые пользователю функциональные блоки:

- регистровый файл;
- арифметико-логическое устройство;
- интерфейс к логике пользовательских инструкций;
- контроллер исключений;

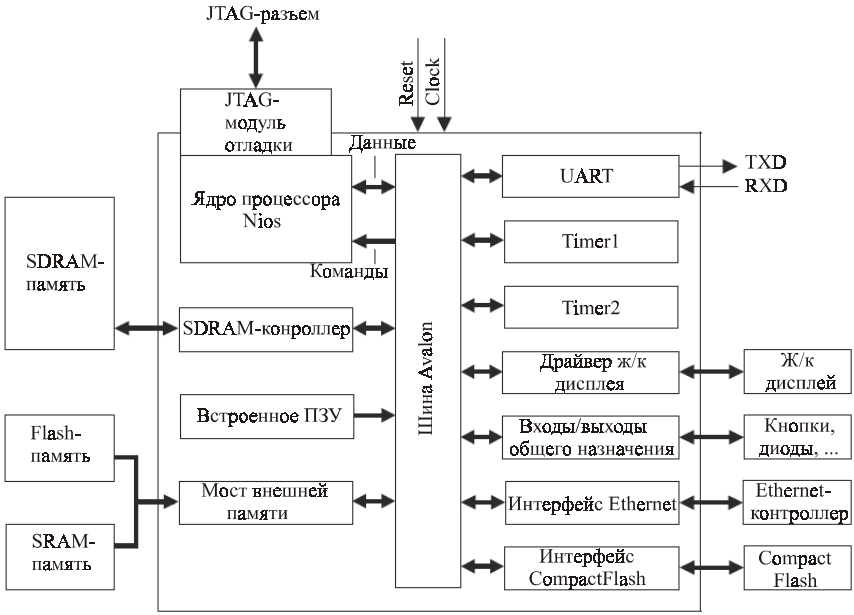


Рис. 39.1. Пример процессорной системы Nios

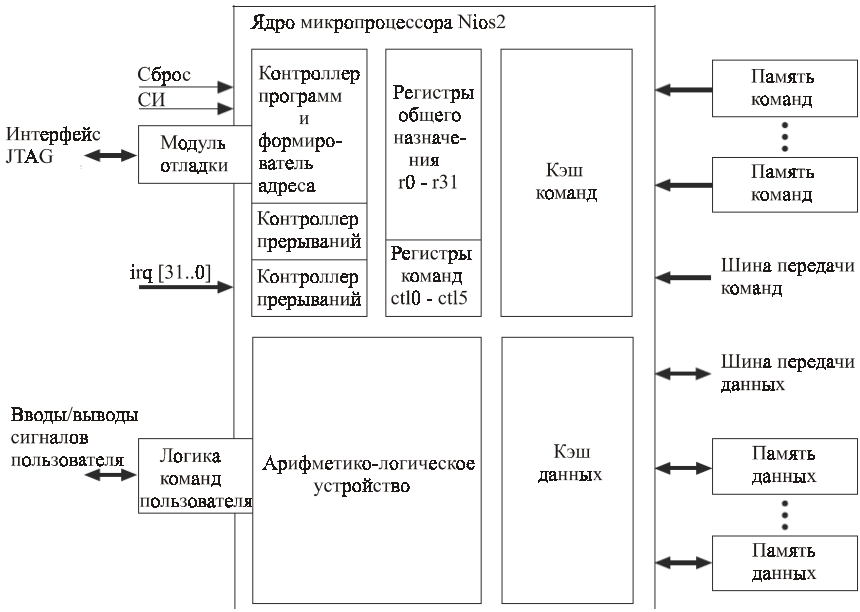


Рис. 39.2. Архитектура микропроцессора Nios

- контроллер прерываний;
- шина передачи команд;
- шина передачи данных;
- кэш команд и данных;
- интерфейс для памяти команд и данных;
- модуль отладки JTAG.

39.1.1. Регистровый файл

Архитектура микропроцессора поддерживает регистровый файл, состоящий из тридцати двух 32-битовых целочисленных регистров общего назначения и шести 32-битовых регистров команд. Также поддерживается режим супервизора и пользователя, что позволяет системному коду защищать регистры команд от ошибок приложений.

39.1.2. Арифметико-логическое устройство

Арифметико-логическое устройство оперирует над данными, хранящимися в регистрах общего назначения. АЛУ совершает операцию с операндами, находящимися в одном или двух регистрах, и сохраняет результат обратно в регистр. АЛУ поддерживает операции, приведенные в табл. 39.1.

Таблица 39.1. *Операции, поддерживаемые АЛУ*

Категория	Описание
Арифметические	Арифметико-логическое устройство поддерживает суммирование, вычитание, умножение и деление над операндами со знаком и без
Отношения	Поддерживаются операции равенства, неравенства, больше или равно, меньше или равно над операндами со знаком и без
Логические	`AND`, `OR`, `NOR` и `XOR`
Операции сдвига	Поддерживаются операции сдвига и вращения, как вправо, так и влево на 0—31 разряд за команду

Неосуществленные команды

Некоторые конфигурации микропроцессора Nios не обеспечивают аппаратную поддержку выполнения операций умножения или деления. Такие операции процессор может эмулировать программным обеспечением, это операции: `mul`, `muli`, `mulxss`, `mulxsu`, `mulxuu`, `div`, `divu`.

Микропроцессор генерирует исключение всякий раз, когда он встречает неосуществленную команду, и обработчик исключений вызывает подпрограмму, которая эмулирует операции программно. Поэтому неосуществленные команды не затрагивают программиста.

Специализированные команды

Архитектура Nios поддерживает определяемые пользователем специализированные команды. Арифметико-логическое устройство Nios соединяется непосредственно с логикой команд пользователя, позволяя конструкторам осуществить аппаратно операции, к которым можно обращаться и использовать их как родные команды.

39.1.3. Контроллеры

Контроллер исключений

Архитектура Nios предоставляет простой, невекторизованный контроллер исключений для обработки всех типов исключений. Все исключения, включая аппаратные прерывания, заставляют микропроцессор передавать управление единственному адресу исключения. Обработчик исключений, находящийся по этому адресу, определяет причину исключения и вызывает соответствующую подпрограмму. Адреса исключений определяются во время генерации системы.

Контроллер прерываний

Система поддерживает 32 внешних аппаратных прерывания. Запросы прерываний поступают на 32 входа ядра микропроцессора, чувствительных к уровню сигнала. Приоритет прерывания определяется программным обеспечением. Поддерживаются вложенные прерывания.

39.1.4. Память и организация ввода/вывода

Так как система, содержащая микропроцессор Nios, в отличие от стандартных микропроцессоров, конфигурируема, то блоки памяти, периферийные устройства изменяются от системы к системе. Ядро микропроцессора для организации доступа к памяти и для обеспечения ввода/вывода использует следующее:

- главный порт команд — главный порт соединяется с памятью команд через многовходовую систему коммутации Avalon;
- кэш команд — быстродействующая кэш-память, встроенная в ядро микропроцессора;

- главный порт данных — главный порт соединяется с памятью данных через многовходовую систему коммутации Avalon;
- кэш данных — быстродействующая кэш-память, встроенная в ядро микропроцессора;
- сильносвязанная память команд и данных — интерфейс к быстродействующей памяти, находящейся вне ядра микроконтроллера.

39.1.5. Шины данных и команд

Архитектура Nios поддерживает раздельные шины данных и команд, что классифицируется как Гарвардская архитектура. Шины команд и данных реализованы как главные порты шины Avalon, которые поддерживают спецификацию интерфейса Avalon.

39.1.6. Кэш-память

Кэш-память находится на одном кристалле, как неотъемлемая часть ядра микропроцессора. Кэш-память может сократить среднее время доступа к памяти для систем, использующих медленные внешние запоминающие устройства типа синхронного динамического ОЗУ.

39.2. Шина Avalon

Avalon — простая шинная архитектура, предназначенная для объединения процессоров на кристалле и внешних устройств этих процессоров в системе.

Avalon — интерфейс, который определяет подключение порта шины между мастером и подчиненными компонентами, определяет синхронизацию, которой эти компоненты связываются.

Основные цели проекта шины Avalon:

- простота — обеспечивает легкий для понимания и изучения протокол;
- оптимизированное использование ресурса кристалла при генерации логики шины — минимизирует затраты логических элементов (LEs) внутри программируемого логического устройства (PLD);
- синхронные операции — хорошо интегрируется с пользовательской логикой, которая сосуществует на том же самом PLD, и у пользователя не возникают проблемы с анализом синхронизации работы с процессором.

Основные операции шины Avalon передают отдельный байт, слово половинной разрядности, или слово (8, 16 или 32 бит) между мастером и подчиненным внешним

устройством. После того как передача завершается, шина становится доступна на следующем же фронте синхрочастоты для другой операции, или между той же самой парой мастер—подчиненный, или между несвязанными хозяином и подчиненными.

Шина Avalon также поддерживает расширенные возможности, типа предварительно установленного времени ожидания для внешнего устройства, внешние устройства, передающие данные потоком и режим мультимастера шины. Такие расширенные возможности работы шины позволяют пересылать несколько блоков данных между внешними устройствами в течение одной операции шины.

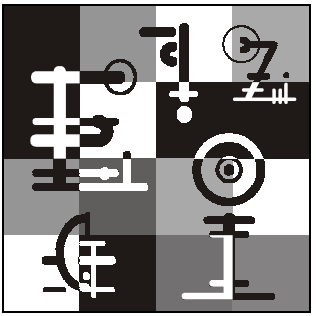
Шина Avalon поддерживает режим работы мультимастера, такая архитектура обеспечивает большую гибкость в конструкции SOPC-систем и необходима при работе внешних устройств с высокой полосой пропускания данных. Например, внешнее устройство может выполнять роль мастера и осуществлять пересылку данных по прямому доступу в память, без привлечения ресурсов процессора на то, чтобы передать данные от этого внешнего устройства в память.

Мастер шины Avalon и подчиненные взаимодействуют друг с другом в соответствии с методикой, называемой арбитражем на стороне подчиненного устройства. Арбитраж на стороне подчиненного устройства определяет, какой из мастеров получит доступ к подчиненному, когда несколько мастеров пытаются одновременно обратиться к одному и тому же подчиненному устройству.

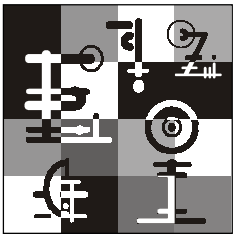
Лабораторная работа. Создание индивидуальных программ для процессора Nios. Тестирование их на макетной плате

Программа работы

1. Самостоятельно напишите программу, использующую любые приборы визуальной индикации, находящиеся на плате.
2. Скомпилируйте программу.
3. Загрузите в память.
4. Запустите на выполнение, используя описания этих действий в предыдущей лабораторной работе.



ПРИЛОЖЕНИЯ



Приложение 1

Основные элементы языка VHDL

Краткая спецификация и описание основных элементов языка VHDL.

Алфавит языка

Как и любой другой язык программирования, VHDL имеет свой алфавит — набор символов, разрешенных к использованию и воспринимаемых компилятором. В алфавит языка входит следующее.

- Латинские строчные и прописные буквы:

  `A`, `B`,..., `Z` и `a`, `b`,..., `z`.

- Цифры от 0 до 9.
- Символ подчеркивания "_". Из символов, перечисленных в этом и предыдущих пунктах, могут конструироваться идентификаторы в программе. Кроме того, написание идентификаторов должно подчиняться следующим правилам:
  - идентификатор не может быть зарезервированным словом языка;
  - идентификатор должен начинаться с буквы;
  - идентификатор не может заканчиваться символом подчеркивания "_";
  - идентификатор не может содержать двух последовательных символов подчеркивания "_".
- Символ "пробел", символ табуляции, символ новой строки.

  Данные символы являются разделителями слов в конструкциях языка. Количество разделителей не имеет значения.

- Специальные символы, участвующие в построении конструкций языка:

  `+ – * / = < > . , ( ) : ; # ' " |`

- Составные символы, воспринимаемые компилятором как один символ:

  `<=` `>=` `=>` `:=` `/=`

  Разделители между элементами составных символов недопустимы.

Комментарии

Признаком комментария являются два символа тире ("--"). Компилятор игнорирует текст, начиная с символов "--" до конца строки, т. е. комментарий может включать в себя символы, не входящие в алфавит языка (в частности, русские буквы).

Числа

В стандарте языка определены числа как целого, так и вещественного типа. Однако средства синтеза ПЛИС допускают применение только целых чисел. Целое число в VHDL может быть представлено в одной из четырех систем счисления: двоичной, десятичной, восьмеричной и шестнадцатеричной. Конкретные форматы написания числовых значений будут описаны далее при рассмотрении различных типов языка.

К разновидности числовых значений можно отнести также битовые строки.

Символы

Запись символа представляет собой собственно символ, заключенный в одиночные кавычки. Например:

```
'A','*',' '
```

В средствах синтеза ПЛИС область применения символов ограничена использованием их в качестве элементов перечислимых типов.

Строки

Строки представляют собой набор символов, заключенных в двойные кавычки. Чтобы включить двойную кавычку в строку, необходимо ввести две двойные кавычки.

Типы данных

Подобно высокоуровневым языкам программирования, VHDL является языком со строгой типизацией. Каждый тип данных в VHDL имеет определенный набор принимаемых значений и набор допустимых операций. В языке предопределено достаточное количество простых и сложных типов, а также имеются средства для образования типов, определяемых пользователем.

Рассмотрим не все типы данных, определенные в стандарте, а только те, которые поддерживаются средствами синтеза ПЛИС.

Простые типы

Следующие простые типы являются предопределенными.

- **BOOLEAN** (логический) — объекты данного типа могут принимать значения `FALSE` (ложь) и `TRUE` (истина).
- **INTEGER** (целый) — значения данного типа представляют собой 32-разрядные числа со знаком.
- Объекты типа `INTEGER` могут содержать значения из диапазона: $-(2^{31} - 1)...(2^{31} - 1)$ или (–2 147 483 647...2 147 483 647).
- **BIT** (битовый) — представляет один логический бит. Объекты данного типа могут содержать значение `'0'` или `'1'`.
- **STD_LOGIC** (битовый) — представляет один бит данных. Этот тип определен стандартом IEEE 1164 для замены типа `BIT`. Объекты типа `STD_LOGIC` могут принимать 9 значений: `'0'`, `'1'`, `'Z'`, `'-'`, `'L'`, `'H'`, `'U'`, `'X'`, `'W'`.

  Для синтеза логических схем могут использоваться следующие значения:

  - `'0'` — логический "0";
  - `'1'` — логическая "1";
  - `'Z'` — третье состояние или значение высокого импеданса, например, на выходе буферных элементов;
  - `'-'` — не подключен;
  - `'X'` — неопределенное значение, например, отражающее неизвестное начальное состояние триггеров.
- **STD_ULOGIC** (битовый) — представляет один бит данных. Объекты данного типа могут принимать 9 значений. Этот тип определен стандартом IEEE 1164 для замены типа `BIT`.
- **ENUMERATED** (перечислимый) — используется для задания пользовательских типов.
- **SEVERITYJ.EVEL** — перечислимый тип, используется только в операторе `ASSERT`.
- **CHARACTER** — символьный тип.

Сложные типы

Из всей совокупности сложных типов, определенных в стандарте языка, для синтеза логических схем используются только массивы (тип `ARRAY`) и записи (тип `RECORD`).

Однако тип RECORD поддерживается не всеми средствами синтеза и рассмотрен не будет.

Следующие типы-массивы являются предопределенными:

- **BIT_VECTOR** — одномерный массив элементов типа BIT;
- **STD_LOGIC_VECTOR** — одномерный массив элементов типа STD_LOGIC;
- **STD_ULOGIC_VECTOR** — одномерный массив элементов типа STD_ULOGIC;
- **STRING** — одномерный массив элементов типа CHARACTER. Направление и границы диапазона индексов не содержатся в определении указанных типов и должны быть указаны непосредственно при объявлении объектов данных типов.

Основные элементы VHDL

Под элементами понимаются те понятия, которые используются при написании кода VHDL.

Синтаксис

Исходный текст программы на языке VHDL состоит из последовательностей операторов, записанных с учетом следующих правил:

- каждый оператор — это последовательность слов, содержащих буквы английского алфавита, цифры и знаки пунктуации;
- слова разделяются произвольным количеством пробелов, табуляций и переводов строки;
- операторы разделяются символами ";";
- в некоторых операторах могут встречаться списки объектов, разделяемые символами "," или ";".

Комментарии могут быть включены в текст программы с помощью двух подряд идущих символов "–". После появления этих символов весь текст до конца строки считается комментарием.

Для указания системы счисления для констант могут быть применены спецификаторы:

- B — двоичная система счисления, например B"0011";
- O — восьмеричная система счисления, например O"3760";
- H — шестнадцатеричная система счисления, например H"F6A0".

Характеристика объектов VHDL

Объекты являются *контейнерами* для хранения различных значений в рамках модели.

Каждый объект характеризуется типом и классом. Типы разделяются на предопределенные в языке VHDL и определяемые пользователем.

Тип показывает, какого рода данные может содержать объект.

Класс показывает, что можно сделать с данными, содержащимися в объекте. В языке VHDL определены следующие классы объектов:

- **`Constant`** — константы. Значение константы определяется при ее объявлении и не может быть изменено. Константы могут иметь любой из поддерживаемых типов данных;
- **`Variable`** — переменные. Значение, хранимое в переменной, меняется везде, где встречается присваивание данной переменной. Переменные могут иметь любой из поддерживаемых типов данных;
- **`Signal`** — сигналы. Сигналы представляют значения, передаваемые по проводам и определяемые присвоением сигналов (отличным от присвоения переменных). Сигналы могут иметь ограниченный набор типов (обычно `bit`, `bit_vector`, `std_logic`, `std_logic_vector`, `integer` и, возможно, другие, в зависимости от среды разработки).

Повторное использование присваивания сигналов в наборе параллельных операторов не допускается. В наборе последовательных операторов такое присваивание допустимо и даст значение сигнала, соответствующее последнему по порядку присваиванию.

Атрибуты

Атрибуты (или иначе свойства) определяют характеристики объектов, к которым они относятся. Стандарт VHDL предусматривает как предопределенные, так и определяемые пользователем атрибуты, однако современные инструментальные средства в большинстве своем поддерживают только предопределенные атрибуты. Для обращения к атрибутам объекта используется символ `'`(например `A1'left`).

В VHDL определены следующие атрибуты:

- `'left` — левая граница диапазона индексов массива;
- `'right` — правая граница диапазона индексов массива;
- `'low` — нижняя граница диапазона индексов массива;
- `'high` — верхняя граница диапазона индексов массива;
- `'range` — диапазон индексов массива;
- `'reverse_range` — обращенный диапазон индексов массива;
- `'length` — ширина диапазона индексов массива.

Компоненты

Объявление компонента в рассматриваемом файле определяет интерфейс к модели объекта на VHDL (**entity** и **architecture**), описанной в другом файле. Компоненты вводятся при структурном описании аппаратуры. Обычно объявление компонента совпадает с соответствующим объявлением entity. Они могут различаться только значениями по умолчанию. Эти значения используются, когда какой-либо из выводов компонента остается не присоединенным (ключевое слово open) при установке компонента в схему.

Оператор объявления компонента может находиться внутри объявления architecture или в заголовке пакета (**package**). Соответствующие компоненту объявления entity и architecture не обязательно должны существовать в момент анализа схемы. В момент моделирования или синтеза должны существовать объявления entity и architecture для компонентов, которые не только объявлены, но и установлены в схему. Это позволяет, например, конструктору задать объявления библиотечных элементов, а реальное их описание (объявления entity и architecture) задавать по мере использования этих элементов в проекте.

Объявление компонента записывается следующим образом:

```
Component name
     [ port ( port_list ) ; ]
end component;
```

Операторы и выражения

С помощью *операторов* описывается алгоритм, определяющий функционирование схемы. Они могут находиться в теле функции, процедуры или процесса.

- **Wait** until condition;

  Приостанавливает выполнение процесса, содержащего данный оператор до момента выполнения условия.

- Signal **<=** expression;

  Оператор присваивания сигнала устанавливает его значение, равным выражению справа.

- Variable **:=** expression;

  Оператор присваивания устанавливает значение переменной, равным выражению справа.

- Procedure_name (parameter {, parameter});

  Оператор вызова процедуры состоит из имени процедуры и списка фактических параметров.

- **If** condition then

  Sequence_of_statements

  { Elsif condition then

  equence_of_statements)

  [else

  sequence_of_statements]

  end if ;

  Оператор `if` используется для ветвления алгоритма по различным условиям.

- **Case** expression is

  When choices_list => sequence_of_statements;

  (When choices_list => sequence of_statements;)

  When others => sequence_of_statements;

  End case;

  Оператор `case` подобно оператору `If` *задает* ветвление алгоритма. Значения в списках разделяются символом "`I`". Когда значение выражения встречается в одном из списков значений, выполняется соответствующая последовательность операторов. Если значение выражения не присутствует ни в одном из списков, то выполняется список операторов, соответствующий ветви `when others`.

- [loop_label :]

  for loop_index_variable **in** range **loop**

  sequence_of_statements

  end loop [loop_label] ;

  Оператор цикла позволяет многократно выполнить последовательность операторов. Диапазон значений задается в виде `value1` **to** `value2` или `value1` **downto** `value2`. Переменная цикла последовательно принимает значения из заданного диапазона. Количество итераций равно количеству значений в диапазоне.

- **Return** expression;

  Этот оператор возвращает значение из функции.

- Null;

  Пустой оператор, не выполняет никаких действий.

Выражения могут содержать следующие операторы: изменение типа, and, or, nand, nor, xor, =, /=, <, <=, >, >=, +, –, &, *, /, mod, rem, abs, not.

В зависимости от избранной САПР при синтезе может поддерживаться подмножество приведенных выше операторов. Порядок вычисления выражений определяется приоритетом операторов:

and, or, nand, nor, xor — самый низкий приоритет

=, /=, <, <=, >, >=

`+`, `-`, `&` (бинарные)

`+`, `-` (унарные)

`*`, `/`, `mod`, `rem`

`abs`, `not` — высший приоритет

Операторы с более высоким приоритетом выполняются раньше. Чтобы изменить такой порядок, используются скобки.

Описание на VHDL объектов проекта: интерфейс, тело объекта и конфигурация

Полное VHDL-описание *объекта проекта* (в дальнейшем объекта) состоит как минимум из двух отдельных описаний: описание интерфейса объекта и описание тела объекта (описание архитектуры).

Интерфейс описывается в объявлении объекта **`entity declaration`** и определяет входы и выходы объекта, его входные и выходные порты **`ports`** и параметры настройки **`generic`**. Параметры настройки отражают тот факт, что некоторые объекты могут иметь управляющие входы, с помощью которых может производиться настройка объектов, в частности задаваться время задержки.

Тело объекта специфицирует его структуру или поведение. Его описание по терминологии VHDL содержится в описании его архитектуры **`architecture`**.

VHDL позволяет отождествлять с одним и тем же интерфейсом несколько архитектур. Это связано с тем, что в процессе проектирования происходит проработка архитектуры объекта: переход от структурной схемы к электрической принципиальной, от поведенческого к структурному описанию.

Тело объекта проекта может быть описано на VHDL тремя разными архитектурами (структурной, поведенческой и потоковой):

1. Средства VHDL *для отображения структур цифровых систем* базируются на представлении о том, что описываемый объект `entity` представляет собой структуру из компонентов (`component`), соединяемых друг с другом линиями связи. Каждый компонент, в свою очередь, является объектом и может состоять из компонентов низшего уровня (иерархия объектов). Взаимодействуют объекты путем передачи сигналов **`signal`** по линиям связи. Линии связи подключаются к входным и выходным портам компонентов. В VHDL *сигналы* отождествляются с линиями связи.

   Имена сигналов и имена линий связи совпадают (они отождествляются). Для сигналов (линий), связывающих компоненты друг с другом, необходимо указывать индивидуальные имена.

Описание структуры объекта строится как описание связей конкретных компонентов, каждый из которых имеет имя, тип и карты портов. Карта портов (`port map`) определяет соответствие портов компонентов поступающим на них сигналам, можно интерпретировать карту портов как разъем, на который приходят сигналы и в который вставляется объект-компонент.

2. Средства VHDL для отображения поведения описываемых архитектур строятся на представлении их как совокупности параллельно взаимодействующих процессов. Понятие процесса (`process`) относится к базовым понятиям языка VHDL.

   Архитектура включает в себя описание одного или нескольких *параллельных процессов*. Описание процесса состоит из последовательности операторов, отображающих действия по переработке информации. *Все операторы внутри процесса выполняются последовательно.* Процесс может находиться в одном из двух состояний — либо пассивном, когда процесс ожидает прихода сигналов запуска или наступления соответствующего момента времени, либо активном — когда процесс исполняется.

   Процессы взаимодействуют путем обмена сигналами.

   В общем случае в поведенческом описании состав процессов не обязательно соответствует составу компонентов, как это имеет место в структурном описании.

   Поведение VHDL-объектов воспроизводится на ЭВМ, и приходится учитывать особенности воспроизведения параллельных процессов на однопроцессорной ЭВМ. Особая роль в синхронизации процессов отводится механизму событийного воспроизведения модельного времени (`now`).

   Когда процесс вырабатывает новое значение сигнала перед его посылкой на линию связи, говорят, что он вырабатывает будущее сообщение (`transaction`). С каждой линией связи (сигналом) может быть связано множество будущих сообщений. Множество сообщений для сигнала называется его драйвером (`driver`).

   Таким образом, драйвер сигнала — это множество пар "время—значение" (множество планируемых событий).

   Язык VHDL реализует механизм воспроизведения модельного времени, состоящий из циклов.

   На первой стадии цикла вырабатываются новые значения сигналов. На второй стадии процессы реагируют на изменения сигналов и переходят в активную фазу. Эта стадия завершается, когда все процессы перейдут снова в состояние ожидания. После этого модельное время становится равным времени ближайшего запланированного события, и все повторяется.

3. В языке VHDL для наиболее часто используемых видов процессов — процессов межрегистровых передач — введено *потоковое описание поведения*. Здесь нет

параллельно взаимодействующих процессов (нет операторов `process`). Здесь выполнение операторов VHDL параллельно, но оно "разведено" во времени с помощью задержек.

Конфигурация объекта. Оператор конфигурации (`configuration`) определяет, какие реализации для каждого компонента должны быть использованы, и позволяет изменять связи компонентов в вашем проекте во время моделирования и синтеза.

Конфигурации не являются обязательными, независимо от того, насколько сложен описываемый проект. При отсутствии конфигурации, стандарт VHDL определяет набор правил, который обеспечивает конфигурацию по умолчанию; например, в случае, когда предусмотрено более одной реализации для блока, последняя скомпилированная реализация получит приоритет и будет связана с объектом.

Описание задержек сигналов

Объект с задержкой можно представить как бы состоящим из двух — идеального элемента и элемента задержки.

В языке VHDL встроены две модели задержек — инерциальная и транспортная.

Инерциальная модель предполагает, что элемент не реагирует на сигналы, длительность которых меньше порога, равного времени задержки элемента. *Транспортная* модель лишена этого ограничения.

Инерциальная модель по умолчанию встроена в оператор назначения сигнала языка VHDL. Например, оператор назначения

```
Y<=X1 and X2 after 10 ns;
```

описывает работу вентиля "И" и соответствует инерциальной модели.

Указание на использование транспортной модели обеспечивается ключевым словом `transport` в правой части оператора назначения. Например, оператор

```
YT<=transport XI andX2 after 10 ns;
```

отображает транспортную модель задержки вентиля.

Задержка может быть задана не константой, а выражением, значение которого может конкретизироваться для каждого экземпляра объекта, используемого как компонент. Для этого ее следует задать как параметр настройки в описании интерфейса объекта.

Описание пакета в VHDL

Описание пакета задается ключевым словом **`package`** и используется, чтобы собирать часто используемые элементы конструкции для глобального применения в других проектах.

Пакет можно рассматривать как общую область хранения, используемую, чтобы хранить описания типов, констант и глобальные подпрограммы. Объекты, определенные в пределах пакета, можно использовать в любом другом проекте на VHDL, и можно откомпилировать в библиотеки для дальнейшего повторного использования. Пакет может состоять из двух основных частей: описания (объявления) пакета и тела пакета.

Описание пакета может содержать следующие элементы:

- объявления типов и подтипов;
- объявления констант;
- глобальные описания сигналов;
- объявления процедур и функций;
- спецификацию атрибутов;
- объявления файлов;
- объявления компонентов;
- объявления псевдонимов;
- операторы включения.

Пункты, появляющиеся в пределах описания пакета, могут стать видимыми в других проектах с помощью оператора включения.

Если пакет содержит описания подпрограмм (функций или процедур) или определяет одну или более задерживаемых констант (константы, чья величина не задана), то в дополнение к описанию необходимо тело пакета.

Тело пакета (которое определяется с использованием комбинации ключевых слов `package body`), должно иметь то же имя, что и соответствующее описание пакета, но может располагаться в любом месте проекта (оно не обязано располагаться немедленно после описания пакета).

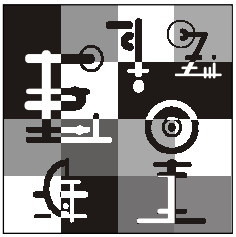
Отношение между описанием и телом пакета отчасти напоминает отношение между описанием и реализацией объекта проекта (тем не менее, может быть только одно тело пакета для каждого описания пакета). В то время как описание пакета обеспечивает информацию, необходимую для использования объектов, определенных в пределах этого пакета (список параметров для глобальной процедуры или имя определенного типа или подтипа), фактическое поведение таких объектов, как процедуры и функции, должно определяться в пределах тела пакета.

Объекты проекта могут базироваться на стандартных средствах языка VHDL — сигналы представляются в алфавите '1', '0', логические операции "И", "ИЛИ", "НЕ" также определяются в этом алфавите.

Во многих случаях приходится описывать поведение объектов в других алфавитах. Например, в реальных схемах сигнал кроме значений '1' и '0' может принимать значение высокого импеданса 'Z' (на выходе буферных элементов) и неопределенное значение 'X ', например, отражая неизвестное начальное состояние триггеров.

Переход к другим алфавитам моделирования осуществляется в VHDL с помощью пакетов.

Использование оператора `package` рассматривается при поведенческом и структурном описании на VHDL процессора DP-32.



Приложение 2

Операции языка Verilog HDL и примеры их применения

Основные операции Verilog HDL

Таблица П2.1. *Основные операции Verilog HDL*

Тип	**Оператор**	**Описание**
Арифметические операции (arithmetic operators)	`+ - * /`	Арифметический (arithmetic)
	`%`	Значение по модулю (modulus)
Операции отношений (relational operators)	`> >= < <=`	Отношение (relational)
Операции совпадения, равенства (equality operators)	`==`	Совпадение (logical equality)
	`!=`	Несовпадения (logical inequality)
Логические операции (logical operators)	`!`	Логическое НЕ (logical NOT)
	`&&`	Логическое И (logical AND)
	`||`	Логическое ИЛИ (logical OR)
Поразрядные операции (bit-wise operators)	`~`	Поразрядное НЕ (bit-wise NOT)
	`&`	Поразрядное И (bit-wise AND)
	`|`	Поразрядное ИЛИ (bit-wise OR)
	`^`	Поразрядное ЛИБО (bit-wise XOR)
	`^~ or ~^`	Поразрядное ЛИБО-НЕ (bit-wise XNOR)

Таблица П2.1 *(окончание)*

Тип	Оператор	Описание
Операции сведения вектора к элементу поразрядными операциями (reduction operators)	`&`	Преобразование вектора в элемент операцией И (reduction AND)
	`|`	Преобразование вектора в элемент операцией ИЛИ (reduction OR)
	`~&`	Преобразование вектора в элемент операцией И-НЕ (reduction NAND)
	`~|`	Преобразование вектора в элемент операцией ИЛИ-НЕ (reduction NOR)
	`^`	Преобразование вектора в элемент операцией ЛИБО (reduction XOR)
	`~^ or ^~`	Преобразование вектора в элемент операцией ЛИБО-НЕ (reduction XNOR)
Операции сдвига (shift operators)	`<<`	Сдвиг влево (left shift)
	`>>`	Сдвиг вправо (right shift)
Операции условия (conditional operators)	`?:`	Условие (conditional)
Операция конкатенации (concatenation)	`{ }`	Конкатенация (concatenation)

Примеры применения операций Verilog HDL

Арифметические операции

```
parameter size=8;
wire [3:0] a,b,c,d,e;
assign c = size + 2; // constant + constant
assign d = a + 1; // variable + constant
assign e = a + b; // variable + variable
```

Операции отношений

```
function [7:0] max( a, b );
input [7:0] a,b;
```

```
if ( a >= b ) max = a;
else max = b;
endfunction
```

Операции совпадения, равенства

```
module is_jump_instruction ( instruction, jump );
parameter JMP = 2'h3;
input [7:0] instruction;
output jump;
assign jump = (instruction[7:6] == JMP);
endmodule
```

Операция сравнения (Handling Comparisons to X or Z)

```
always begin
if (A == 1'bx) // this is line 10
B = 0;
else
B = 1;
End
```

Логические операции

```
module is_valid_sub_inst(inst,mode,valid,unimp);
parameter IMMEDIATE=2'b00, DIRECT=2'b01;
parameter SUBA_imm=8'h80, SUBA_dir=8'h90,
SUBB_imm=8'hc0, SUBB_dir=8'hd0;
input [7:0] inst;
input [1:0] mode;
output valid, unimp;
assign valid = (((mode == IMMEDIATE) && (
(inst == SUBA_imm) ||
(inst == SUBB_imm))) ||
((mode == DIRECT) && (
(inst == SUBA_dir) ||
(inst == SUBB_dir))));
assign unimp = !valid;
endmodule
```

Поразрядные операции

```
module full_adder( a, b, cin, s, cout );
input a, b, cin;
output s, cout;
assign s = a ^ b ^ cin;
assign cout = (a&b) | (cin & (a|b));
endmodule
```

Операции сведения вектора к элементу поразрядными операциями

```
module check_input ( in, parity, all_ones );
input [7:0] in;
output parity, all_ones;
assign parity = ^ in;
assign all_ones = & in;
endmodule
```

Операции сдвига

```
module divide_by_4( dividend, quotient );
input [7:0] dividend;
output [7:0] quotient;
assign quotient = dividend >> 2; // shift right 2 bits
endmodule
```

Операции условия

```
module add_or_subtract( a, b, op, result );
parameter ADD=1'b0;
input [7:0] a, b;
input op;
output [7:0] result;
assign result = (op == ADD) ? a+b : a-b;
endmodule

module arithmetic( a, b, op, result );
parameter ADD=3'h0,SUB=3'h1,AND=3'h2,
OR=3'h3, XOR=3'h4;
input [7:0] a,b;
```

```
input [2:0] op;
output [7:0] result;
assign result = ((op == ADD) ? a+b : (
(op == SUB) ? a-b : (
(op == AND) ? a&b : (
(op == OR) ? a|b : (
(op == XOR) ? a^b : (a))))));
endmodule
```

Операция конкатенации

```
output [7:0] ccr;
wire half_carry, interrupt, negative, zero,
overflow, carry;
...
assign ccr = { 2'b00, half_carry, interrupt,
negative, zero, overflow, carry };
```

```
output [7:0] ccr;
...
assign ccr[7] = 1'b0;
assign ccr[6] = 1'b0;
assign ccr[5] = half_carry;
assign ccr[4] = interrupt;
assign ccr[3] = negative;
assign ccr[2] = zero;
assign ccr[1] = overflow;
assign ccr[0] = carry;
```

Литература

1. ACT Family FPGA Data Book. — Actel, 1990.
2. Altera 1998 Data Book — January 1998.
3. Peter J. Ashenden. The VHDL Cookbook. — 1990.
4. The Programmable Logic Data Book. — Xilinx, 1998.
5. Алексеенко А. Г., Шагурин И. И. Микросхемотехника. — М.: Радио и связь, 1990.
6. Амосов В. В., Кракау Т. К., Черноруцкий И. Г., Черноруцкая Н. Н. Автоматизированный анализ работы логических элементов. — ЛПИ, 1989.
7. Антонов А. П., Мелехин В. Ф., Филиппов А. С. Обзор элементной базы фирмы Altera. — СПб.: ЭФО, 1997.
8. Бибило П. Н. Основы языка VHDL. — М.: Солон-Р, 2000.
9. Бибило П. Н. Синтез логических схем с использованием языка VHDL. — М.: Солон-Р, 2002.
10. Гомоюнов К. К., Кракау Т. К. Элементы и узлы ЭВМ. — ЛПИ, 1984.
11. Поляков А. К. Языки VHDL и Verilog в проектировании цифровой аппаратуры. — М.: СОЛОН-Пресс, 2003.
12. Пухальский Г. И., Новосельцева Т. Я. Цифровые устройства: Учебное пособие для втузов. — СПб.: Политехника, 1996.
13. Стешенко В. Б. ПЛИС фирмы ALTERA: проектирование устройств обработки сигналов. — М.: Додека, 2000.
14. Угрюмов Е. П. Проектирование элементов и узлов ЭВМ: Учебное пособие для вузов. — М.: Высшая школа, 1987.
15. Угрюмов Е. П. Цифровая схемотехника. — СПб.: БХВ-Петербург, 2000.

Предметный указатель

A

Active VHDL 230
APEX20K 497

C

CACHE 199
Cygwin 508

D

Design Center 36
DRAM 189, 194
DV-триггер 95, 191
D-триггер 95, 105, 351, 492, 493

E

EDIF (Electronic Data Interchange Format) 417
EP20K200E 504
EPF10K70 461, 463, 465, 468
EPM7064 506
EPM7128S 461, 462, 463, 464, 467, 468
EPROM 188
Excalibur 504

F

FACM 200
Field Programmable Gate Array (FPGA) 216
FlashPro 456
FLEX 10K 485, 486, 487, 493
FPGA 415
FPGA Advantage 317, 375
FPGA eX 442, 451, 453

G, H

Gate Array (GA) 212
HDL Designer 375
HDL-дизайнеры 221
High-complexity PLD (CPLD) 216

J

JK-триггер 124
JK-триггер с задержкой 120
JTAG 505

L

Leonardo Spectrum 375, 405
Libero IDE 415
LUT 216

M, N

MAX+PLUS II 467, 483
Model Sim 375, 396, 409, 415, 417
Nios 500

P

ProASIC^{PLUS} 442, 444, 445
ProASIC^{PLUS} Starter Kit 425
Programmable Logic Device (PLD) 216
PROM 188

Q

QUARTUS 497, 500
QUARTUS II 355, 356

R

RAM 189, 199
RGB 472
RISC-процессор 510
ROM 188, 199
RST-триггер 124
RS-триггер 91, 93, 96, 104, 119, 122, 347
R-триггер 348

S, T, U

SRAM 189, 191, 192, 199, 492, 494
Test Bench 224, 250, 398, 417
UML (Unified Modeling Language) 222

V

Verilog VHDL 531
VeriLogger Pro / Testbencher Pro 342
VGA 470, 472, 473, 475
VHDL 96, 223, 519
  модель 228, 250, 273, 275, 277, 303, 306
  операторы 138, 139
VHDL Test Bench 241

Z

Z-состояние 47, 493, 497

А

Автоматическая генерация теста 242
АЗУ 189
АЛУ 512
Анализ логической схемы 409
Анализ спроектированных устройств ВТ 226
Анализ цепи 6
Аппаратная коррекция одиночной ошибки 172
APEX20K 498, 500
Арифметико-логическое устройство (АЛУ, ALU) 180, 361

Б

Базис синтеза 405
Базисные элементы 405
Базовая схема
  ЛЭ ТТЛ 45, 49
  ЛЭ ЭСЛ 55
  серий ЭСЛ 51
Биполярный транзистор 10
Бистабильная ячейка 90
БК 209, 210
БК ИС 211
БМК ИС 211
Буфер FIFO 203
Буфер в DP32 293
БЯ БМК 211

В

ВАХ 7
Вентиль 74
Вентильная матрица (ВМ) 212
Вентильное проектирование 74
Видеопамять 201
Виртуальная оперативная память 311
Виртуальный режим i8086 311
Восьмиразрядный параллельный регистр 351
Временная диаграмма 82
Выключатель 89, 90

Г

Гарвардская архитектура 514
Генератор несимметричных импульсов 85
Генератор симметричных импульсов 85

Д

Двунаправленные ключи ввода/вывода 193
Двухпортовая память 500
Двухтактный Т-триггер 121
Двухфазный D-триггер 95
Делитель 125
Демультиплексор (DMX) 151, 156
Дескриптор 309
Дешифратор (DC) 133
  неполный 139
  разрядность 135
Дизъюнктор 31
Дизъюнкция 137
Динамический риск 83
Диод Шотки 43
Длительность импульса 83, 84

Е, З

Емкость 18
Законы Кирхгофа 6
Замещение источников электрической энергии 19
Запоминающие элементы логических устройств (ЗЭ ЛУ) 89
Запоминающий элемент (ЗЭ) 89
Затвор 60
Защелка 95
Защелка (Transparent Latch) в DP32 292, 298
Защелкивающий буфер в DP32 293
Защищенный режим 308, 310, 312
ЗЭ DRAM 194
ЗЭ SRAM 192
ЗЭ на n-МОП транзисторах 192
ЗЭ на К-МОП схемах 193

И

И 31—33, 86, 94, 119, 120, 121—124, 126, 212, 213
Идеальный вентиль 18
Идеальный ключ 19
Иерархия ЗУ 187
И-ИЛИ-НЕ 44, 212
ИЛИ 31—33, 53, 94, 136, 212, 213
ИЛИ-НЕ 32, 33, 43, 53, 63, 72, 74, 91, 92, 93, 104, 119, 121
Импульс синхронизации 105
Импульсный фильтр 347
Инвертор 12, 32, 38, 89, 91
Инвертор на КМДП-схемах 61
Индуктивность 18
И-НЕ 32, 33, 43—45, 49, 63, 74, 93, 95, 135
Исследование логического элемента 35
Исток 60

К

Карта Карно 75
Клавиатура PS/2 477
Классификация ЗУ 188
Код Хемминга 170
Комбинационные логические устройства (КЛУ) 69, 82
Комбинационный одноразрядный двоичный сумматор (См) 72
Комбинационный сумматор 175, 176
Компаратор (CMP) 158, 162
Компаратор в DP32 298
Комплементарные схемы 61
Конвейеризация 307
Контекстно-адресуемая память 500
Контроль четности/нечетности 164, 165
Конъюнктивный терм 136, 149
Ко-симуляция 244, 273
Кэш-память 199, 514

Л

Логическая операция 44
Логическая переменная 44, 51
Логические адреса 308
Логические устройства (ЛУ) 69
ЛЭ на КМДП-схемах 61, 62, 63

М

Мажоритарный элемент 75
Мажоритарный элемент (МЭ) 166
Макетная плата ProASIC^{PLUS} Evaluation Board 425
Макетная плата UP2 Education Board 461
Матричные умножители 182, 184
МДП-транзистор 59, 60
Методология проектирования устройств ВЧ 229
Множительный блок (МБ) 183
Модель теста устройств ВТ (Test Bench) 224
Модифицированный код Хэмминга 171
Монитор VGA 470, 472
МОП-транзистор 59
МПЗУ 205
Мультиплексор (MUX) 148, 150, 156
Мультиплексор в DP32 291
МФЗУ 189
Мышь PS/2 482

Н

Надежность работы мультизадачных ОС 311
Накапливающий сумматор 176, 177

О

Обнаружение и исправление ошибок при кодировании 163
Обратная связь (ОС) 90
Одноразрядный сумматор (СМ) 175
ОЗУ 189
Открытый коллектор 48
Ошибки 163

П

Параллельный регистр 109
ПЗ 209
ПЗУ 189, 205, 492
Пиксел 202
ПЛБИС 483, 498
ПЛИС 209, 212, 405, 415, 417, 425, 442, 461, 462, 464, 465, 466, 467, 469, 470, 472, 477, 482, 485, 504, 506, 510
ПЛИС Antifuse 442
ПЛМ 212, 213
ПЛСБИС 209, 216, 497
ПМЛ 212, 215
Подложка 60
Полусумматор (ПС) 69
Получение произвольных логических функций 152
Последовательностные логические устройства (ПЛУ) 69
Последовательный регистр 107, 108
ППЗУ 189, 205, 212
Правила Де'Моргана 32, 149
Принципиальная электрическая схема 5
Проектирование современных устройств ВТ 221
Процессор DP32 244, 271, 275, 277, 303, 312, 313
  VHDL-модель 248, 259, 273, 275, 277
  VHDL-модель теста 251
  VHDL-модель, ошибки 303, 304, 305
  архитектура 271
  команды 245, 247
  конвейеризация 307
  ко-симуляция 250
  чтение/запись в память 246
Процессор:
  DP32 271
  i80286 308
  Nios 500, 504, 510

Р

Размножение источников напряжения 19
Реальный режим 308
Регистр памяти 109
Регистр сдвига 105
Регистры общего назначения в DP32 296
Регрессионное тестирование 224
Режимы процессора 308
Риск 104
РПЗУ 189, 205

С

САПР 408
САПР фирмы Альтера Max+Plus II 76, 86, 113
СБИС 405
Сбой GP 310
Сегмент 310
Сигнал в VHDL 97
Симофор 95
Симулятор 396
Симуляция спроектированных устройств ВТ 226
Синдром ошибки 170, 172
Синтез логической схемы 405
Синтез спроектированных устройств ВТ 228
Синтезируемость VHDL-кода 97
Синхронизирующие импульсы (СИ) 94
Системы синхронизации 105
Сопротивление 17
Состояние проекта 241
Специализированные БИС 209
Стимулятор 239
Сток 60
Схема замещения 8
Схема простейшего генератора импульсов 86

Схема формирователя
импульсов 87
Схема формирователя импульсов
с парафазным выходом 87
Схемы контроля по модулю 2 164,
165, 167
Схемы памяти 187
Счетный элемент (СЭ) 118, 119, 209
Счетчик команд в DP32 295

Т

Таймер 379
Тестирование спроектированных
устройств ВТ 224
Технология Antifuse 450, 452
Технология Flash 442
Транзистор 9
Транзисторно-транзисторная логика
(ТТЛ) 43, 44
Трехполюсники 8
Т-триггер с задержкой 120

У

Универсальный регистр 110
Усилитель 11
переменного тока 27, 42
постоянного тока 26
с источником шума на входе 42
Усилительный каскад 14
Ускорение распространения
переноса 178, 179

Ф

Физические адреса 308
ФЯ БМК 212

Х, Ц

Характеристика передачи напряжения
(ХПН) 45, 53
ХПН инверторов 90
Цепочка инверторов 28

Ш

Шина Avalon 514
Шифратор 142
двоичный 146
приоритетный (HPRI) 142
разрядность 145

Э

Элемент Шеффера 32
Элементы "И" 462
Элементы "ИЛИ" 462
Элементы памяти 32
Элементы цифровой техники 31
Эмиттерно-связанная
логика (ЭСЛ) 51
Этапы проектирования КЛУ 69

Я

Язык Verilog HDL 321, 329
модули 321, 324
оператор always 340
оператор задачи 339
операторы 329
операторы условия 333, 335
операторы цикла 337, 338
переменные модуля 325, 326, 327
порты 323
функции 329, 332
Язык VHDL 137