

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«МИРЭА - Российский технологический университет»
(РТУ МИРЭА)

Тарасов И. Е.

**Схемотехника устройств компьютерных систем.
Ч. 2**
Учебное пособие

Москва 2022

УДК 004
ББК 32.972
Т 19

Тарасов И.Е. Схемотехника устройств компьютерных систем. Часть 2 [Электронный ресурс]: Учебное пособие / Тарасов И.Е. — М.: МИРЭА – Российский технологический университет, 2022. — 1 электрон. опт. диск (CD-ROM)

В учебном пособии рассматриваются сведения о схемотехнических решениях, применяемых в компьютерных устройствах.

Предназначено для студентов, обучающихся по направлению подготовки бакалавров 09.03.01 «Информатика и вычислительная техника».

Учебное пособие издается в авторской редакции.

Автор: Тарасов Илья Евгеньевич.

Рецензенты:

Андреева Ольга Николаевна, д.т.н., доцент, начальник отдела научной работы АО «Концерн «Моринсис-Агат»

Лямин Юрий Алексеевич, к.т.н., с.н.с., начальник отдела ФГАУ НИИ «Восход»

Системные требования:

Наличие операционной системы Windows, поддерживаемой производителем.

Наличие свободного места в оперативной памяти не менее 128 Мб.

Наличие свободного места в памяти постоянного хранения (на жестком диске) не менее 30 Мб.

Наличие интерфейса ввода информации.

Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).

Подписано к использованию по решению Редакционно-издательского совета

МИРЭА — Российский технологический университет.

Объем: 12,2 мб

Тираж: 10

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	7
1. ОСНОВНЫЕ ПОНЯТИЯ КУРСА	8
1.1. Состав компьютерной системы	8
1.2. Маршрут проектирования компьютерной системы	9
1.3. Цифровая, измерительная и силовая электроника в составе компьютерной системы, методы и инструменты их проектирования.....	11
1.4. Проектирование цифровых устройств	14
1.5. Уровни проектирования	16
1.6. Инструменты проектирования	19
1.7. Выводы по разделу	23
2. ПОРЯДОК РАЗРАБОТКИ ЦИФРОВОГО УСТРОЙСТВА И ОСНОВНЫЕ ТЕНДЕНЦИИ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ	24
2.1. Основные сведения о технологии производства интегральных схем	24
2.2. Технологический процесс, его характеристики	25
2.3. Понятие технологического сдвига	28
2.4. Потребление энергии интегральными схемами и понятие Power-Delay Product.....	29
2.5. Современные тенденции и проблемы: темный кремний, GALS, стена памяти	32
2.6. Синхронный стиль проектирования.....	35
2.7. Выводы по главе	43
3. ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ	44
3.1. Основные сведения	44
3.2. Архитектура логической ячейки.....	48
3.3. Дополнительные компоненты FPGA	52
3.4. Инструменты разработки для FPGA	64
3.5. Выводы по главе	69
4. РЕАЛИЗАЦИЯ БАЗОВЫХ УСТРОЙСТВ ЦИФРОВОЙ СХЕМОТЕХНИКИ	71
4.1. Основные цифровые элементы	71
4.2. Реализация логических выражений в ПЛИС.....	73
4.3. Дополнительные возможности логических ячеек ПЛИС	76
4.4. Выводы по главе	80
5. РЕАЛИЗАЦИЯ ОСНОВНЫХ АРИФМЕТИЧЕСКИХ ФУНКЦИЙ	82

5.1. Содержание раздела	82
5.2. Сумматор и его логическая функция	82
5.3. Особенности описания сумматора	84
5.4. Цепи ускоренного переноса	86
5.5. Вычитание. Смена знака.....	88
5.6. Умножение	88
5.7. Деление.....	92
5.8. Числа с фиксированной точкой	93
5.9. Числа с плавающей точкой	94
5.10. Вычисление трансцендентных функций на базе алгоритма CORDIC .	98
5.11. Табличная реализация функций	101
5.12. Выводы по разделу.....	102
6. РЕАЛИЗАЦИЯ ОСНОВНЫХ СИНХРОННЫХ УСТРОЙСТВ	104
6.1. Содержание раздела	104
6.2. Триггер, его разновидности и порядок описания	104
6.3. Асинхронный и синхронный сброс, рекомендуемые практики применения управляющих сигналов	107
6.4. Управляющие наборы (control sets) в ПЛИС.....	108
6.5. Особенности проектирования и моделирования сигнала сброса.....	109
6.6. Разработка счетчика	110
6.7. Широтно-импульсная модуляция.....	113
6.8. Выводы по разделу	115
7. РЕАЛИЗАЦИЯ НАКРИСТАЛЬНОЙ ПАМЯТИ И ВНЕШНИХ ИНТЕРФЕЙСОВ ПАМЯТИ	116
7.1. Содержание раздела	116
7.2. Характеристики памяти	116
7.3. Описание памяти на поведенческом уровне. Описание памяти на структурном уровне	118
7.4. Ресурсы для реализации памяти в ПЛИС	122
7.5. Реализация памяти в СБИС. Memory compiler.....	127
7.6. Интерфейсы внешней памяти	128
7.7. Выводы по разделу	141
8. ГЛОБАЛЬНО АСИНХРОННЫЕ, ЛОКАЛЬНО СИНХРОННЫЕ СХЕМЫ.....	143
8.1. Содержание раздела	143
8.2. Временные характеристики синхронных устройств и понятие метастабильности	143

8.3. Виды синхронизации цифровых узлов	148
8.4. Архитектура GALS и схемы ресинхронизации	151
8.5. Примеры подключения устройств, требующих ресинхронизации данных	157
8.6. Выводы по разделу	160
9. КОНЕЧНЫЕ АВТОМАТЫ.....	162
9.1. Понятие конечного автомата	162
9.2. Автоматы Мили и Мура	166
9.3. Кодирование состояний конечного автомата.....	167
9.4. Однопроцессное и трехпроцессное описание конечного автомата	168
9.5. Выводы по разделу	173
10. ПРОСТЫЕ ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА.....	175
10.1. Периферийные устройства в компьютерных системах.....	175
10.2. UART	175
10.3. ШИМ.....	181
10.4. SPI	186
10.5. Интерфейс I2C	193
10.6. Интерфейс LCD (жидкокристаллического индикатора)	194
10.7. Интерфейс VGA.....	196
10.8. Однопроводная передача данных	200
10.8. Таймеры.....	203
10.9. Параметризация модулей	205
10.10. Выводы по разделу.....	207
11. ПРОЦЕССОРНОЕ ЯДРО	209
11.1. Основные сведения о процессорных ядрах	209
11.2. Преобразование конечного автомата в процессор.....	210
11.3. Двухтактная архитектура	212
11.4. Организация моделирования процессора	222
11.5. Выводы по разделу.....	225
12. ПРОЦЕССОРНОЕ ЯДРО – ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О ПРОЕКТИРОВАНИИ.....	227
12.1. Регистровая модель процессора и регистровый файл	227
12.2. Арифметико-логическое устройство.....	241
12.3. Конвейер процессора. Архитектура с трехступенчатым конвейером	245
12.4. Зависимости по данным. Архитектура MIPS	249
12.5. Архитектуры с многоступенчатым конвейером	252
12.6. Архитектура VLIW (сверхдлинное командное слово)	256

12.7. Прототипирование процессоров на базе ПЛИС	257
12.8. Выводы по разделу.....	260
13. СИСТЕМНЫЕ ШИНЫ ПРОЦЕССОРНЫХ УСТРОЙСТВ	262
13.1. Системная шина в компьютерной системе.....	262
13.2. Некоторые виды системных шин в компьютерной технике	264
13.3. Дополнительные возможности системных шин	270
13.4. Выводы по разделу.....	274
14. СОПРЯЖЕНИЕ ИЗМЕРИТЕЛЬНЫХ И СИЛОВЫХ УСТРОЙСТВ С ЦИФРОВЫМИ СИСТЕМАМИ.....	276
14.1. Ввод аналоговых сигналов в компьютерных системах.....	276
14.2. АЦП, его характеристики	276
14.3. Архитектуры и интерфейсы АЦП. Сопряжение АЦП с цифровыми системами	281
14.4. ЦАП. Интерфейсы ЦАП. Сопряжение ЦАП с цифровыми системами.....	288
14.5. Управление силовыми устройствами с помощью ШИМ.....	289
14.6. Выводы по разделу.....	291
15. СТРАТЕГИИ МОДЕЛИРОВАНИЯ И ВЕРИФИКАЦИИ КОМПЬЮТЕРНЫХ СИСТЕМ.....	292
15.1. Поведенческое и физическое моделирование, их отличия и место в маршруте проектирования.....	292
15.2. Входные воздействия и наблюдение реакции системы	297
15.3. Понятие стратегии моделирования	300
15.4. Системное моделирование	303
15.5. Выводы по разделу.....	304
16. ПРАКТИЧЕСКИЕ ВОПРОСЫ ПРОЕКТИРОВАНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ.....	306
16.1. Постановка задачи проектирования	306
16.2. Жизненный цикл проекта и связь со смежными специалистами.....	307
16.3. Математические модели предметной области, проектирование на системном уровне.....	310
16.4. Выводы по разделу.....	312
СПИСОК ИСТОЧНИКОВ И ЛИТЕРАТУРЫ	314
СВЕДЕНИЯ ОБ АВТОРАХ	315

ВВЕДЕНИЕ

Методическое пособие излагает основные сведения о проектировании основных узлов компьютерных систем. Рассматриваются основы маршрута проектирования цифровых устройств, этапы разработки и используемые инструменты. Описывается порядок проектирования основных узлов цифровой схемотехники, базовых арифметических операций, трансцендентных функций, модулей памяти, конечных автоматов, контроллеров периферийных устройств и других компонентов вычислительных систем. Даются сведения об архитектурах и схемотехнике конечных автоматов, процессорных ядер и систем на их основе.

1. ОСНОВНЫЕ ПОНЯТИЯ КУРСА

1.1. Состав компьютерной системы

На рис. 1.1 показана архитектура электронно-вычислительной машины (ЭВМ), предложенной фон Нейманом, который заложил теоретические основы вычислительной техники в середине 20 века. В этой схеме данные, которые обрабатывает ЭВМ, находятся в памяти. Операции по преобразованию данных выполняются арифметико-логическим устройством (АЛУ) на основе сигналов, формируемым управлением устройством управления. В ЭВМ имеются также устройства ввода и вывода данных.

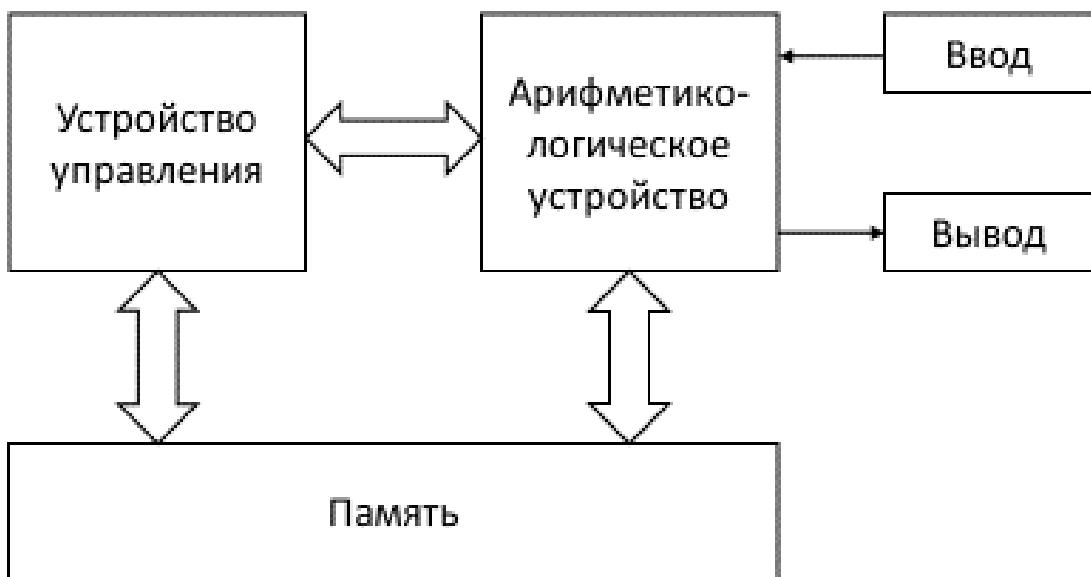


Рисунок 1.1. Архитектура ЭВМ по фон Нейману

В схеме фон Неймана не накладываются ограничения на реализацию, форматы данных или типы команд. Устройства ввода и вывода также могут быть различными. Сутью предложенного подхода является описание взаимодействия между компонентами ЭВМ, параметры которых могут уточняться и развиваться по мере перехода к новым технологиям и появлению новых потребностей.

Современной тенденцией является повсеместное использование вычислительных устройств, в том числе для решения задач, которые исходно не требовали вычислителей. Например, устройства «умного дома», носимой электроники, распределенные измерительные устройства, бытовая техника реализуются на основе компьютерной системы. Такие компьютерные устройства образуют широкий класс *встраиваемых* (embedded) систем. Требования к производительности вычислений, объему памяти, типам устройств ввода-вывода

и другим параметрам в этом случае различаются очень существенно. Еще больше они различаются при рассмотрении крупных категорий вычислительных систем, которые можно условно представить следующим списком:

- встраиваемые системы;
- мобильные компьютеры (смартфоны, планшеты, ноутбуки);
- стационарные компьютеры (десктопы, рабочие станции);
- серверы, рабочие станции центров обработки данных (ЦОД);
- суперкомпьютеры.

По приведенному списку можно сделать наблюдение о том, что указать какой-то универсальный подход к проектированию невозможно. В случае суперкомпьютера важно обеспечить высокую производительность вычислений и большой объем памяти. Для смартфона, работающего от аккумулятора, важно также время работы, поэтому обеспечивать чрезмерную производительность ценой повышенного энергопотребления будет неправильным. Поэтому по мере развития потребностей в вычислительной технике остается актуальным вопрос поиска новых архитектур, разработки новых процессоров, устройств ввода-вывода и систем на их базе.

1.2. Маршрут проектирования компьютерной системы

При проектировании компьютерной системы нужно учитывать, что разработанное аппаратное обеспечение будет взаимодействовать с программным обеспечением. Может сложиться ситуация, когда операции, требуемые программы, не смогут быть выполнены вообще или будут выполняться длительное время. Это ухудшит характеристики системы в целом, вплоть до невозможности ее использования. Напротив, если часто используемые операции будут выполняться быстро, эффективность аппаратного обеспечения окажется существенно выше. На этом пути всегда существуют компромиссы – например, полностью ориентировать процессор на выполнение одной определенной программы часто нерационально.

Для современной ситуации в области вычислительной техники характерна тенденция к разработке частично специализированных вычислительных систем. Они обозначаются термином «проблемно-ориентированные системы» (domain-specific), который отражает тот факт, что они нацелены на преимущественное решение одной задачи или подкласса похожих задач. Например, графический сопроцессор (GPU) хорошо выполняет преобразования координат в трехмерном пространстве и вычисляет положение текстур на экране с учетом масштабирования, однако в целом не может запускать программы,

предназначенные для центрального процессора. С другой стороны, центральный процессор при построении изображений на экране покажет слишком низкую производительность.

Подход к проектированию аппаратной части с учетом требования будущего программного обеспечения обозначается термином «совместное проектирование» (в зарубежной литературе «hardware & software co-design»). Это не относится к какому-то одному общепризнанному алгоритму или программному продукту. Процесс проектирования в этом случае заключается в разработке варианта аппаратного обеспечения и его поэтапной модификации по мере проверки эффективности программ, которые предполагается на нем запускать. Для такой проверки изготовление микросхем не требуется, вместо этого используется их компьютерное моделирование.

Основные этапы проектирования компьютерной системы проиллюстрированы на рис. 1.2.

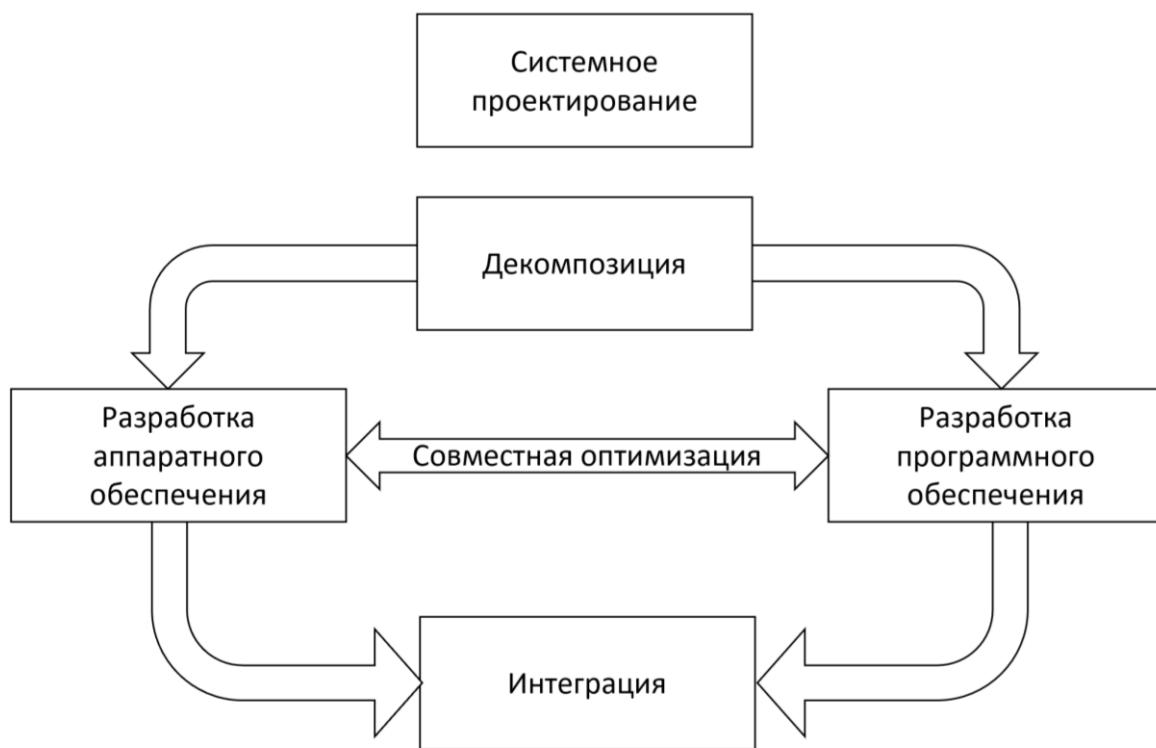


Рисунок 1.2. Основные этапы проектирования компьютерной системы

Проектирование системы начинается с этапа системного проектирования. Результаты этого этапа будут использоваться для *декомпозиции* – распределения задач между аппаратным и программным обеспечением. Это распределение не обязано быть окончательным, поскольку в процессе разработки могут выявиться

дополнительные сведения, которые заставят перенести задачу из одной подсистемы в другую.

Например, программист может выяснить, что определенная функция занимает слишком много времени, снижая общую производительность системы. Общим подходом с точки зрения системного анализа является перенос этой функции в аппаратную подсистему, для чего нужно разработать соответствующее устройство, которое обеспечит выполнение этих действий на уровне аппаратуры без участия процессора.

Не существует единственного общепризнанного подхода к организации подобных проектов. Например, после выхода нового процессора основной идеей проекта может быть обеспечение этого процессора соответствующими программами. В этом случае сам процессор не будет подвергаться модификациям, однако для него можно разработать устройства ввода-вывода и оптимизировать существующее программное обеспечение.

Если проект фокусируется на определенном программном продукте, возможна разработка аппаратных компонентов, которые будут ускорять его выполнение (или как-то еще способствовать улучшению условий использования – например, путем уменьшения потребляемой мощности). Примером могут быть системы беспроводной связи или нейропроцессоры. В этих случаях уже разработанные алгоритмы получают аппаратное обеспечение, которое ускоряет выполнение соответствующих программ, разгружая центральный процессор.

Организацию проектирования компьютерных систем следует выполнять с привлечением методов работы из области управления проектами.

1.3. Цифровая, измерительная и силовая электроника в составе компьютерной системы, методы и инструменты их проектирования

Электронные компоненты, применяемые в компьютерной системе, можно разделить на цифровые и аналоговые. Цифровые компоненты оперируют дискретными состояниями сигналов (0 и 1), тогда как аналоговые используют непрерывный диапазон напряжений и токов. В свою очередь, аналоговые компоненты можно разделить на измерительные (с приоритетом на обеспечение высокой точности измерений) и силовые (оперирующие сигналами высокой мощности – например, регулирующими ток, протекающий через нагреватель или обмотку электромотора).

Аналоговые компоненты требуют применения других средств проектирования и ставят более сложные задачи. Это связано с тем, что в цифровых схемах для обеспечения уровня 0 или 1 достаточно, чтобы напряжение

находилось в определенном диапазоне значений, а для аналоговой техники изменения уровня сигнала недопустимы, поскольку невозможно понять, что это такое – результат действия помех или действительные значения напряжений и токов. Параметры электронных схем зависят от температуры и подвержены технологическому разбросу. Для цифровых схем это может приводить к снижению допустимой тактовой частоты, но для аналоговых может привести к существенным отклонениям от ожидаемого поведения.

На рис. 1.3 показан графический интерфейс системы моделирования аналоговых схем LTSpice. Аббревиатура SPICE означает «Simulation Program with Integrated Circuit Emphasis» и относится к широко распространенному подходу к описанию и моделированию электронных схем. Начальная реализация подобной программы относится к началу 1970-х годов и на протяжении развития электронной техники появлялись многочисленные варианты программного обеспечения и несколько модификаций стандарта.

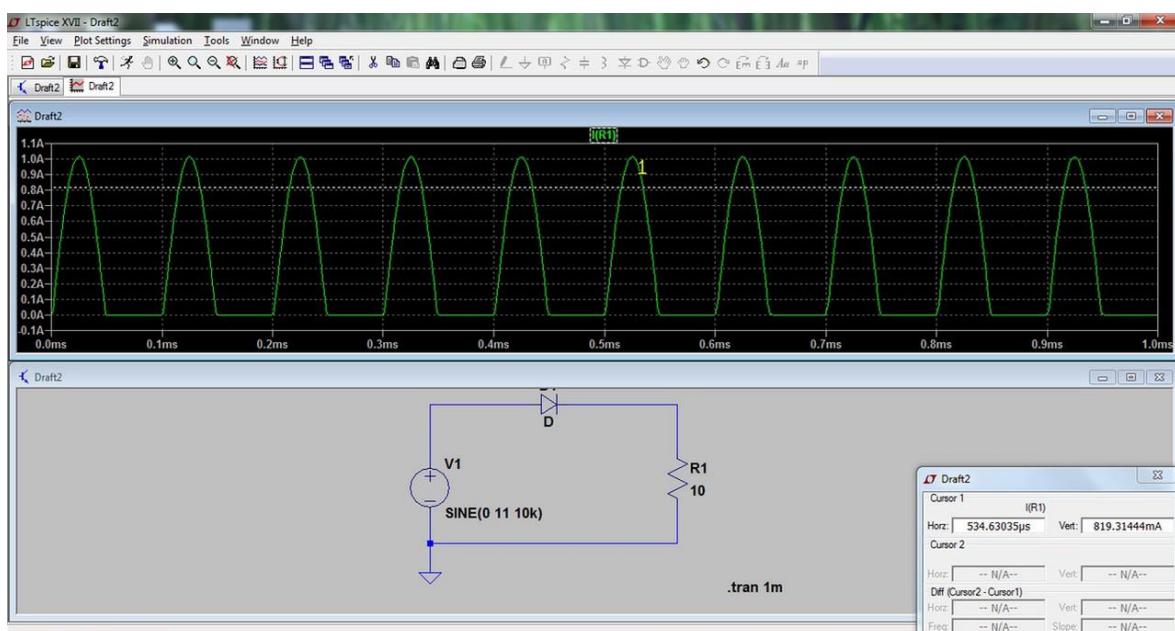


Рисунок 1.3. Интерфейс системы моделирования аналоговых схем LTSpice

Общей тенденцией вычислительной техники является перенос основных операций по обработке данных в цифровую часть системы. Например, вместо установки усилителя аналоговых сигналов, умножающего напряжение на какую-то величину, применяется аналого-цифровое преобразование, в результате которого аналоговый сигнал преобразуется в пропорциональный цифровой код, который может быть уже умножен на нужное число соответствующей операцией процессора. Такой подход уменьшает влияние помех на результат, поскольку

чем меньше аналоговых цепей находится в составе устройства, тем меньше оказывается мест, на которые могут действовать помехи.

При этом измерительная и силовая электроника относятся к таким элементам, которые невозможно заменить цифровыми устройствами. Измерительные схемы обычно основаны на микросхемах АЦП (аналого-цифровой преобразователь), однако эти микросхемы работают в определенном диапазоне входных аналоговых сигналов. Поэтому, если измеряемые сигналы не соответствуют диапазонам работы используемых АЦП, требуется применение дополнительных компонентов, усиливающих (или ослабляющих) аналоговые величины по мере необходимости.

Применение силовых компонентов обусловлено тем, что цифровые схемы не могут обеспечить большую выходную мощность. Уровень выходного напряжения составляет 5 или 3,3 В, а выходной ток измеряется десятками миллиампер. Этого достаточно для сигнального светодиода, но такие элементы, как устройство освещения, электромотор или нагреватель, не могут питаться от выхода цифровой микросхемы. Для решения этой задачи в составе системы присутствуют мощные переключающие элементы (обычно транзисторы), которые способны оперировать высокими уровнями напряжения и переключать токи в единицы или десятки ампер. Управление такими элементами осуществляется цифровыми сигналами.

На рис. 1.4 показан внешний вид системы автоматизированного проектирования EasyEDA. Это пример САПР для разработки принципиальных электрических схем и печатных плат, которая позволяет использовать как цифровые, так и аналоговые компоненты.

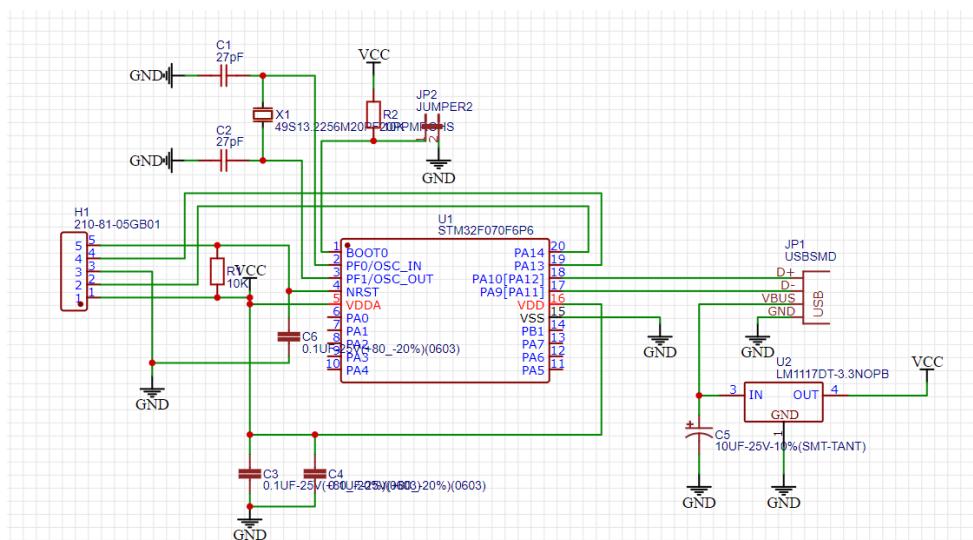


Рисунок 1.4. Внешний вид онлайн-САПР EasyEDA в режиме разработки принципиальной электрической схемы

При проектировании компьютерной системы необязательно использовать полный цикл проектирования аналоговых компонентов и их моделирование в системе SPICE. Измерительные устройства, силовые подсистемы, устройства связи и другие подобные элементы могут быть применены в качестве готовых модулей, производимых сторонними организациями.

1.4. Проектирование цифровых устройств

При проектировании цифровых устройств имеется несколько этапов, на каждом из которых проводятся различные виды работ. На рис. 1.5 схематично показаны основные этапы проектирования и характерные для них проблемы.



Рисунок 1.5. Основные этапы проектирования цифровых устройств в составе компьютерной системы

На уровне системного моделирования используются высокоуровневые средства проектирования, включая программы на обычных языках программирования высокого уровня, имитирующие выполнение действий компонентами компьютера. Данный этап нужен для того, чтобы с минимальной трудоемкостью проверить работу будущей системы в основных сценариях ее

использования, не вдаваясь в технические детали. Может быть написана программа-эмулятор.

Согласно ГОСТ 15971-90, эмуляция определяется следующим образом: «имитация функционирования одного устройства посредством другого устройства или устройств вычислительной машины, при которой имитирующее устройство воспринимает те же данные, выполняет ту же программу и достигает того же результата, что и имитируемое».

Таким образом, программа-эмулятор может выполнять операции над числами в массиве данных, предполагая, что в будущем этот массив будет заменен на микросхему памяти. Аналогично, выполняя операции над переменными, можно считать, что таким образом проверяется работа регистров будущего процессора.

На уровне, обозначенном как RTL (Register Transfer Level), производится разработка принципиальной электрической схемы. На современном уровне проектирование путем рисования графических символов и соединения их линиями используется ограниченно из-за очень большого объема такой схемы. Этот способ проектирования может быть использован для отдельных элементов, или наоборот, для представления взаимодействия между крупными компонентами системы на верхнем уровне (в этом случае схема становится не «принципиальной электрической», а «структурной» или «схемой соединений», в зависимости от изображаемых на ней элементов).

Основным инструментом описания цифровых схем являются языки описания аппаратуры (HDL, Hardware Description Language). На этом уровне необходимо решить ряд задач, которые были неактуальны для более высокого уровня проектирования. Например, выполнить ряд требований по обеспечению их стабильной работы с учетом применяемой технологии. Они будут рассмотрены далее в этом пособии.

На топологическом уровне выполняется разработка полупроводникового кристалла. Если применяется готовая элементная база или программируемая логическая интегральная схема (ПЛИС), ее изготовление не требуется, однако для ПЛИС проводится размещение и трассировка компонентов. Этот этап является достаточно длительным, поэтому при переходе к нему уже нецелесообразно экспериментировать со схемой или изменять алгоритмы – это приведет к повторному проведению всех операций по размещению и трассировке.

Разработчик не обязан одинаково хорошо владеть всеми видами проектирования. В составе коллектива могут быть взаимодействующие

специалисты, каждый из которых выполняет работы в соответствии со своими навыками, однако представляет и проблемы коллег, чтобы не принимать технические решения, которые затрудняют их работу. Как и для совместной разработки программного и аппаратного обеспечения, между показанными уровнями возможно взаимодействие для совместной оптимизации компонентов проекта на разных этапах его выполнения.

1.5. Уровни проектирования

Уровни проектирования позволяют сосредотачиваться на отдельных важных аспектах разработки, не выполняя постоянно полный комплекс работ по проектированию. Например, если необходимо определить требуемый объем памяти для процессора, для этого можно не разрабатывать процессор целиком, включая его корпус, печатную плату для установки и микросхему памяти. Моделирование выполнения программы вполне способно выяснить, сколько памяти достаточно для ее выполнения. Можно привести много подобных примеров, когда целесообразность изменения схемы выясняется без ее полного проектирования и тем более изготовления. Поэтому проектирование разбивается на крупные уровни, которые позволяют привлекать для работы специалистов разной направленности.

На уровне системной модели вычислительное устройство представляется в виде программ, имитирующих его будущую работу. При этом важно сохранить баланс между упрощением такой модели и возможности получить на их основе схему. Например, если модель представляет собой блок, обозначенный как «искусственный интеллект, принимающий оптимальное решение», она бесполезна на практике. Впоследствии другому специалисту придется полностью разработать такой блок, о детализированной работе которого ему не представили никаких сведений. Вместо этого модели должны имитировать работу конкретных компонентов, поведение которых уже известно и может быть описано с достаточным уровнем детализации. Например, модель сумматора в виде строки $a = b + c$ является достаточной, поскольку общие принципы реализации таких компонентов известны.

Вместе с этим системная модель должна отслеживать потенциально проблемные ситуации при последующей реализации схемы. Например, для электронных компонентов не допускается одновременная подача сигналов на вход из двух и более источников. Если представить системную модель в виде выражений $a = 0; a = 1$, это не является проблемой с точки зрения программиста, поскольку в этом случае процессор выполнит операции последовательно. Если

же передать такую системную модель для аппаратной реализации, придется определить, что именно следует подать на вход элемента, который будет хранить переменную a . Вместо этого в системной модели необходимо описать процесс так, чтобы было понятно, что присваивания выполняются в разные моменты времени.

Для описания такого стиля моделирования часто используется понятие транзакции. Оно означает такую операцию, которая может быть выполнена только полностью. Например, процесс записи значения в ячейку памяти означает подачу на модуль памяти всех сигналов – адреса ячейки, данных для этой ячейки и сигнала, указывающего на необходимость выполнения операции записи. Все эти действия должны быть смоделированы совместно. При этом для программиста операторы $addr = 123; data = 45; write = 1$ выглядят независимыми действиями, однако он должен описывать модель так, чтобы все сигналы были согласованными и полностью описывали поведение моделируемого блока в определенный момент времени.

Назначением системного моделирования является раннее выяснение будущих свойств проектируемой системы. Например, операции с памятью, описанные в виде разработанных моделей $read_memory(addr, data)$ и $write_memory(addr, data)$, позволяют быстро определить, что будет храниться в памяти после выполнения определенной программы. Альтернативой была бы разработка схемы такого модуля памяти, что займет существенно больше времени.

После разработки системной модели становится понятно, какие функции должен выполнять процессор, какой объем памяти ему требуется, какие устройства должны быть к нему подключены и т.д. На основе системных моделей можно приступать к разработке электрической схемы, которая представляется на уровне *регистровых передач* (Register Transfer Level). На этом уровне схема описывается в виде фрагментов, передающих данные от одного регистра к другому с их возможным преобразованием. Например, если существуют регистры, хранящие сигналы a , b , c , то на уровне RTL-представления возможна запись $a = b + c$, которую можно преобразовать в конкретные цифровые компоненты.

При работе на уровне RTL-описания схемы обычно используются языки описания аппаратуры (Hardware Description Language), наиболее распространенными из которых являются VHDL (VHSIC HDL – Very High Speed Integrated Circuit HDL) и Verilog HDL (часто сокращается до Verilog в неформальном обсуждении). Тексты на этих языках преобразуются с помощью

программ-синтезаторов в представление, которое называется *списком связей* (netlist).

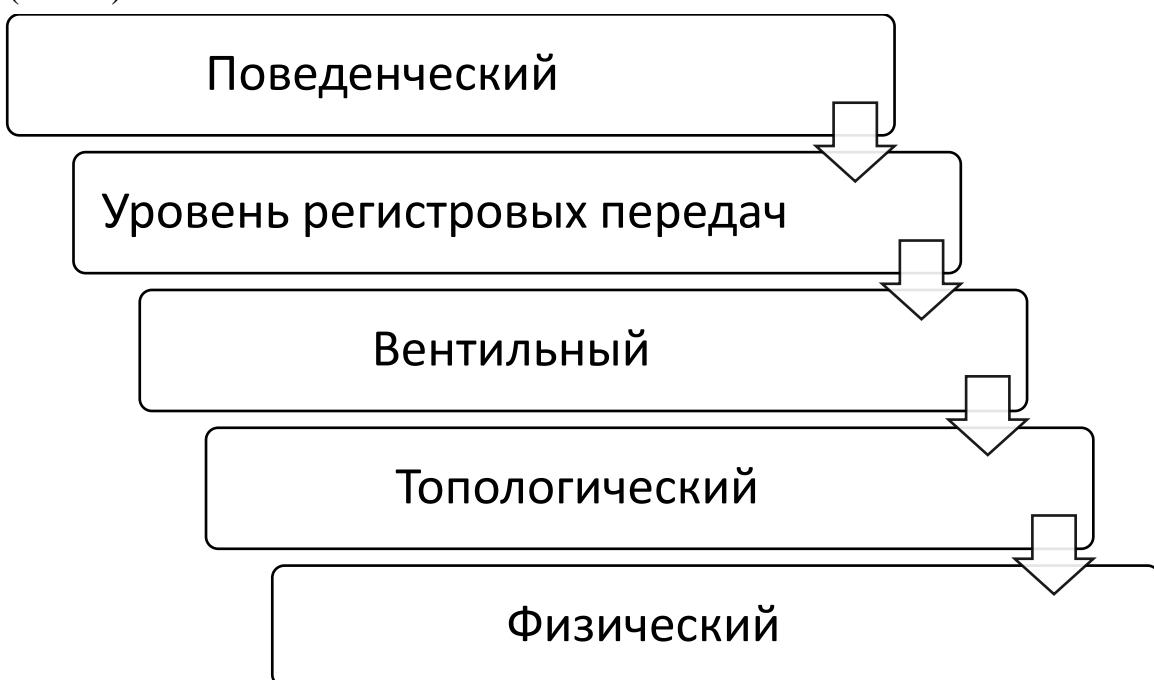


Рисунок 1.6. Взаимодействие основных этапов проектирования

Можно перечислить следующие уровни проектирования:

1. Поведенческий (**behavioral level**).

На этом уровне устройства и их элементы представляются в виде абстрактных моделей. Например, сумматор может быть представлен в виде математического выражения $a = b + c$, не вдаваясь в технические детали того, как это выражение реализуется с помощью цифровых компонентов. На поведенческом уровне могут также использоваться выражения, моделирующие задержки распространения сигналов, однако такие задержки не рассчитываются, а принудительно задаются разработчиками, чтобы выяснить, как устройство будет работать, если его компоненты будут срабатывать не мгновенно. Например, сумматор может быть описан следующим выражением на VHDL:

$a \leq b + c \text{ after } 2 \text{ ns};$

В этом случае программы моделирования покажут изменение сигнала на выходе сумматора через 2 нс после подачи входных значений, но это будет сделано принудительно, а не по результатам моделирования какого-либо конкретного сумматора.

2. Уровень регистрационных передач (RTL, register transfer level)

На этом уровне описание схоже с поведенческим, однако схема ориентирована на описание схем, соединяющих регистры друг с другом. На

уровне регистровых передач игнорируются задержки, задаваемые разработчиком.

3. Вентильный уровень (gate level)

На уровне отдельных логических элементов (вентиляй, *gate*) большая схема, реализующая некоторую логическую функцию, разбивается на отдельные вентили. Сложное выражение может не иметь готового решения в виде отдельного вентиля, поэтому программное обеспечение строит конечную схемотехническую реализацию из тех вентиляй, которые есть в наличии. Например, функцию 4И можно составить из нескольких вентиляй 2И.

4. Топологический уровень (cells level)

Уровень отдельных ячеек (cell), представляющих собой логические вентили, триггеры, мультиплексоры и прочие базовые компоненты цифровых схем.

5. Физический уровень (switch/mask level)

Проектирование на этом уровне предусматривает размещение отдельных транзисторов и их элементов. Обычно компоненты физического уровня поставляются в виде готовых библиотек.

1.6. Инструменты проектирования

Моделирование будущей микросхемы является важным, но не единственным шагом перед ее производством. Некорректно составленная модель может иметь скрытые логические ошибки – разработчик, неправильно понимающий моделируемые процессы, может не задать в модели требуемые сценарии проверки. Поэтому кроме компьютерного моделирования необходимо создать макет будущей микросхемы. Для этого используются программируемые логические интегральные схемы (ПЛИС), с помощью которых можно собрать цифровую схему из конфигурируемых ячеек, изменяя ее внутренние соединения по мере необходимости. Этот процесс чем-то напоминает сборку будущей пластмассовой модели из кубиков Лего. Получаемая конструкция явно не соответствует штампованной фигурке, не может реализовать какие-то сложные формы, однако дает общее представление о том, что получится при изготовлении пресс-формы для штамповки пластмассы. Эта аналогия хороша тем, что как сборку фигурки из Лего, так и программирование ПЛИС можно выполнять множество раз без потери материальных ресурсов, а изготовление пресс-формы для пластмассы требует денег и времени. Еще больше стоит подготовка производства для изготовления полупроводникового кристалла.

Кроме того, что макет на базе ПЛИС предоставляет наглядную демонстрацию работы будущего устройства, он также выявляет так называемые «эффекты реального мира». Например, частота тактовых сигналов на практике подвержена небольшим изменениям, которые трудно адекватно смоделировать в силу случайного характера процессов. Изменения сигналов на входах микросхемы, заданные моделью, также вряд ли будут происходить в те моменты, которые задаются моделью. Окажет ли это существенное влияние на работоспособность микросхемы? Умозрительно ответить на этот вопрос достаточно сложно, тем более что риск получить некорректно работающую микросхему высок, а проверить гипотезу путем создания соответствующей схемы в ПЛИС относительно недорого. Поэтому предварительное создание макета на базе ПЛИС является практически обязательным шагом при разработке новых микросхем.

Маршрут проектирования цифрового устройства на базе ПЛИС показан на рис. 1.7.

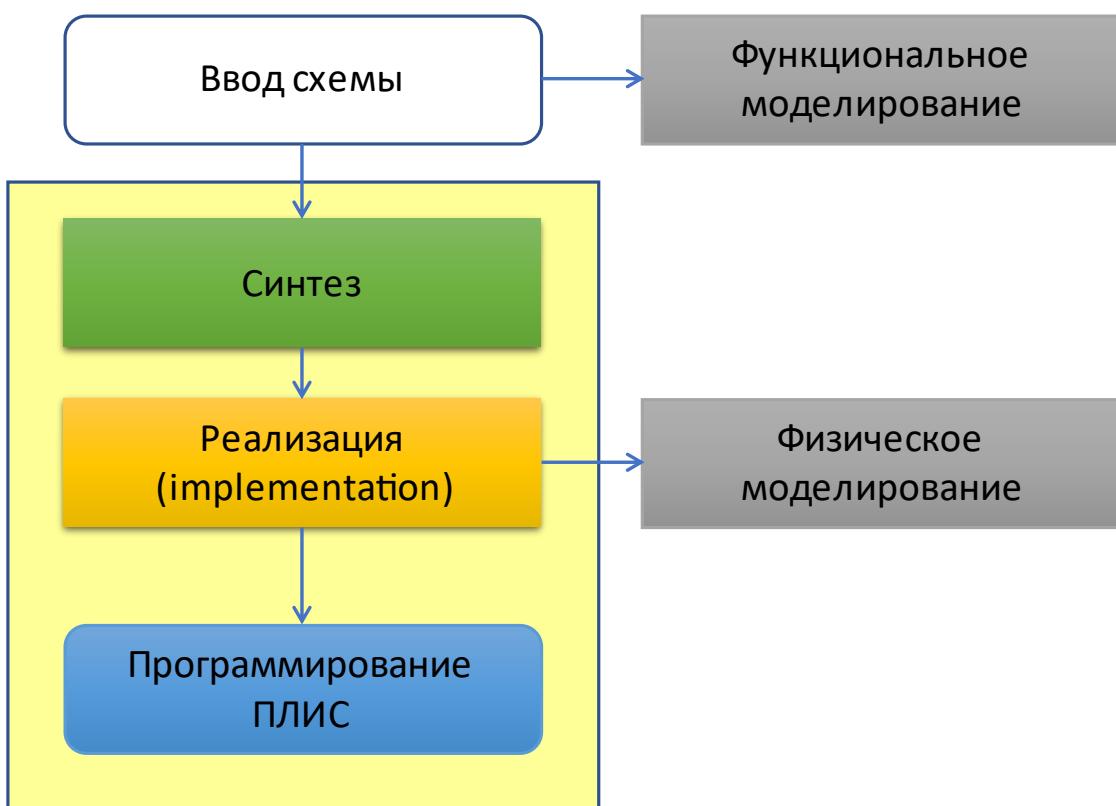


Рисунок 1.7. Маршрут проектирования цифрового устройства на базе ПЛИС

На рисунке видны основные этапы, показанные укрупненно. Эти этапы имеют явное соответствие в САПР. Ввод схемы происходит в основном на поведенческом или RTL уровнях с применением языков описания аппаратуры.

Можно провести предварительное функциональное моделирование, анализируя только конструкции языка (без попытки привязать их к элементам ПЛИС). Этот этап можно проводить и без ПЛИС, с помощью в том числе и свободно распространяемых программ моделирования.

На рис. 1.8 показаны примеры отладочных плат на базе ПЛИС. Такие платы содержат дополнительные компоненты (подсистему питания, память, порты USB и Ethernet, разъемы и т.д.) и готовы к применению, обычно с подключением к компьютеру с помощью USB.



Рисунок 1.8. Примеры отладочных плат на базе ПЛИС

Платы, показанные на рис. 1.8, представляют собой достаточно широкий диапазон ресурсов. В простейшем варианте (верхний левый угол) показана плата, предназначенная для обучения и освоения. В нижнем ряду показан модуль компании Synopsys, предназначенный для моделирования будущих СБИС большого объема.

Поскольку ПЛИС могут использоваться и в составе самостоятельных изделий (а не только как макеты будущих микросхем), для них применяются и другие средства описания схем, кроме языков описания аппаратуры. Соотношение инструментов проектирования для ПЛИС показано на рис. 1.9. В основном дополнительные инструменты являются специализированными, и работают по принципу автоматической генерации текстов на языках описания аппаратуры.

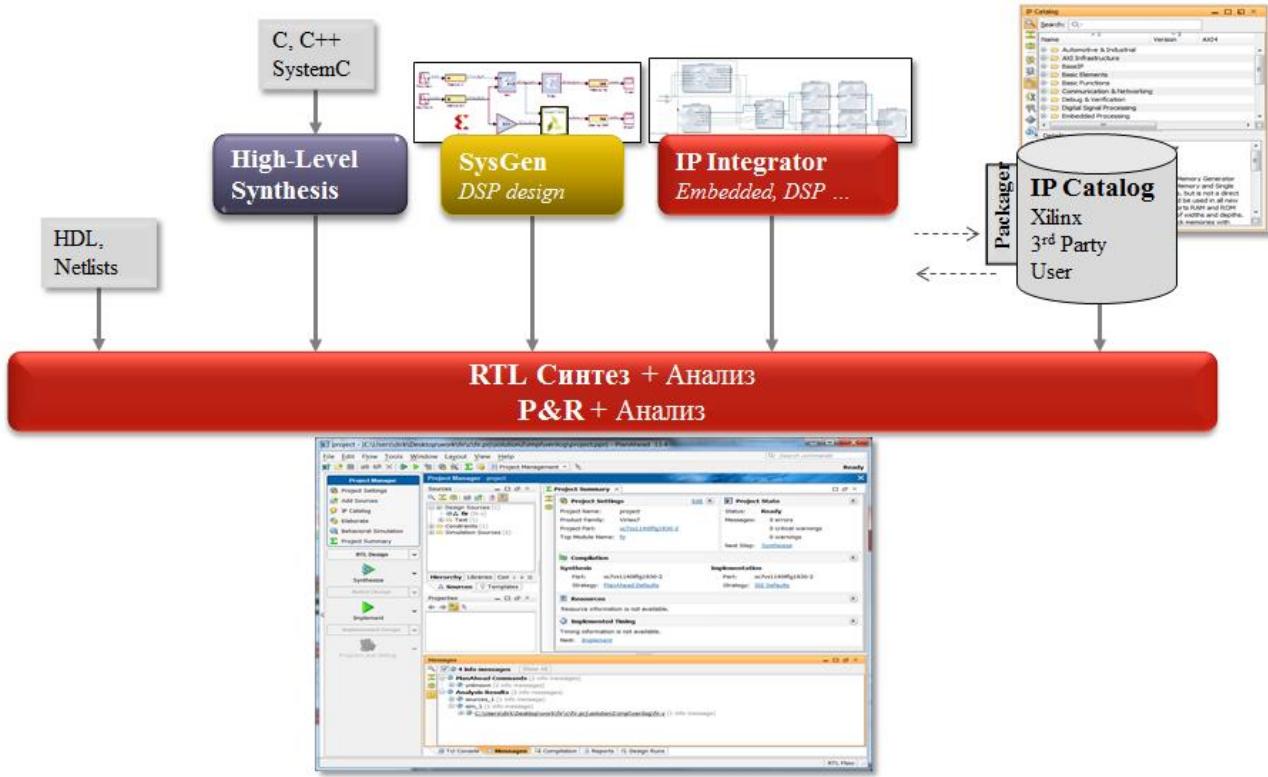


Рисунок 1.9. Основные инструменты проектирования на базе ПЛИС

Среди современных инструментов проектирования можно упомянуть:

- языки высокого уровня (HLL, High Level Languages), основанные на Си (C++, System C);
- генератор систем цифровой обработки сигналов (цифровых фильтров, модулей спектрального анализа и т.п.) System Generator for DSP;
- генератор IP-ядер (готовых компонентов различного назначения).

Важным понятием является IP-ядро (IP – Intellectual Property, «интеллектуальная собственность»). Это готовый блок, который может быть использован в составе проекта, выполняя определенную функцию. У таких блоков нет четкого определения, ни по формату предоставления (схема или фрагмент кристалла), ни по назначению (блоком может быть простой сумматор или сложный контроллер). Роль IP-ядер в проектировании – предоставление готовых фрагментов для сложного проекта. Принципиальным вопросом здесь является возможность быстрой интеграции такого ядра в проект. Можно еще раз отметить, что не существует какого-либо специального формата или языка описания для IP-ядер. Иногда отмечается, что отличием IP-ядра является возможность получения технической поддержки от производителя.

Дополнительные инструменты проектирования не всегда могут быть использованы для разработки СБИС, поскольку в ряде случаев они привязаны к особенностям ПЛИС и используют специализированные компоненты. Эти

компоненты не будут автоматически перенесены в СБИС на основании только предоставленного описания, поскольку в таком высокоуровневом описании будет применяться ссылка на «черный ящик», который имеется только у компании-производителя ПЛИС. При проектировании собственной микросхемы такие компоненты, как контроллер PCI Express, USB, Ethernet, блоки цифровой обработки сигналов (и ряд других) необходимо разработать самостоятельно (или приобрести лицензию на готовый компонент).

1.7. Выводы по разделу

Проектирование вычислительной техники включает в себя деятельность в нескольких смежных направлениях – разработка аппаратной составляющей, разработка программного обеспечения, исследование предметной области и др. Для упрощения взаимодействия между специалистами процесс проектирования разбивается на этапы по времени разработки, а также имеет различные уровни проектирования, от верхних уровней математических моделей до низкоуровневых деталей технологической реализации.

Важными предварительными этапами являются моделирование с помощью специального программного обеспечения (*simulator*) и макетирование путем загрузки разработанной схемы в ПЛИС. Этап макетирования важен, поскольку он позволяет наглядно продемонстрировать работу будущей микросхемы, к тому же выявляет возможные несоответствия модели реальному миру.

Процесс разработки на базе ПЛИС в целом соответствует процессу разработки обычной микросхемы, однако ПЛИС содержат и дополнительные компоненты, которые не будут автоматически перенесены в будущую микросхему. Поэтому макетирование необходимо проводить с учетом возможностей будущего производства и имеющихся в наличии компонентов.

Контрольные вопросы:

1. Что входит в состав компьютера в модели фон Неймана?
2. На какие компоненты можно разбить вычислительную систему?
3. Чем отличаются системная модель, уровень регистрационных передач и топологическое представление?
4. Почему для разработки макета используется ПЛИС? Можно ли обойтись компьютерным моделированием схемы?

2. ПОРЯДОК РАЗРАБОТКИ ЦИФРОВОГО УСТРОЙСТВА И ОСНОВНЫЕ ТЕНДЕНЦИИ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

2.1. Основные сведения о технологии производства интегральных схем

Интегральная микросхема в упрощенном представлении выглядит как кремниевая пластина с тонким слоем оксида кремния (который получается окислением исходной пластины) и нанесенным слоем металла. Получившаяся структура дает название и всей технологии CMOS (КМОП) – Complementary Metal – Oxide – Semiconductor (Комплементарная структура Металл – Оксел – Полупроводник). При этом оксидная пленка выступает как диэлектрик, а термин «комплементарная» обозначает тот факт, что используются как р-, так и н-полупроводники, образующие комплементарную пару транзисторов.

Области пластины могут быть закрыты от обработки специальным материалом – фоторезистом. После его нанесения пластина освещается через фотошаблон, при этом часть фоторезиста разрушается и может быть удалена, открывая области пластины для обработки (например, химического осаждения металла). Процесс нанесения фоторезиста, его экспонирования, удаления излишков и обработки пластины с открывшимися участками повторяется многократно, в результате чего получается многослойная структура, в основе которой лежит полупроводниковая пластина.

Подготовка производства (включая изготовление уникального комплекта фотошаблонов) – весьма дорогостоящий процесс, причем по мере перехода к новым технологиям его стоимость возрастает почти экспоненциально. Это означает, что экономически более выгодно массовое производство микросхем, поскольку в этом случае стоимость подготовки производства будет распределена среди большего количества произведенных пластин. При этом себестоимость производства самой пластины крайне мала в сравнении с подготовкой фотошаблонов.

Высокая стоимость подготовки производства означает, что у разработчиков нет возможности проводить многократные эксперименты со схемой устройства, исправляя найденные ошибки, поскольку такие исправления означают повторные затраты на подготовку производства (NRE, Non-Recurring Engineering, безвозвратные инженерные расходы).

Такая ситуация означает, что при разработке необходимо минимизировать количество запусков производства (в идеале обойтись единственной итерацией

подготовки производства), сосредоточившись на компьютерном моделировании и макетах на базе ПЛИС.

2.2. Технологический процесс, его характеристики

Под технологическим процессом понимается последовательность операций, выполняемая при производстве полупроводниковой пластины. Часто используются не вполне корректный термин вида «техпроцесс 10 нм». В действительности не существует единственного порядка операций для пластин «10 нм». Корректнее говорить о «технологическом процессе, используемом для нормы проектирования 10 нм», однако на практике эта фраза упрощается.

Понятие «норма технологического процесса» отражает тот факт, что на пластине уже не существует одного универсального размера, тем более относящегося к какому-то ключевому параметру схемы. Ранее этот параметр относился к так называемой *circuit line width* («ширина линии на схеме») и обозначал наименьшую толщину проводника, которую можно проложить на кристалле. По мере уменьшения размеров компонентов какие-то из них переставали уменьшаться линейно, а какие-то пришли к определенному минимально допустимому размеру. Вместе с тем, производители используют условное обозначение «норма технологического процесса», которое обозначает некий базовый размер, относительно которого рассчитываются другие параметры.

Можно также обратить внимание на следующий ряд технологических норм:

$$180 - 130 - 90 - 65 - 45 - 28 - 20 - 14 - 10 - 7 - 5 - 3 - 2$$

В этом ряду следующее значение приблизительно в 1,41 раза (квадратный корень из 2) меньше предыдущего. Тогда получается, что плотность компонентов следующего поколения технологических процессов примерно в 2 раза выше. Это отражает не только технические законы, но и рыночные требования, поскольку тратить большие средства на освоение нового поколения технологии нерационально, если при этом нет значимого улучшения технических показателей.

Кроме уменьшения размеров элементов, используются и модификации архитектуры транзисторов. На рис. 2.1 показан срез традиционного планарного («плоскостного») транзистора. В нем можно видеть, что в основной пластине образованы области истока и стока (source и drain), между которыми образуется канал, управляемый полем затвора (gate). В зависимости от напряжения на затворе в области канала появляются носители заряда, что позволяет транзистору пропускать ток от истока к стоку.

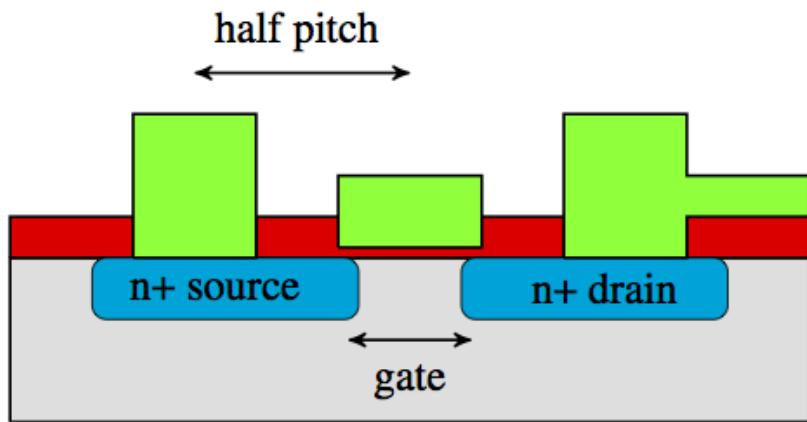


Рисунок 2.1. Срез полупроводникового транзистора

Можно понять и суть проблемы улучшения параметров транзистора – затвор должен находится как можно ближе, но при этом не контактировать с каналом. Уменьшать толщину диэлектрика до бесконечности нельзя, поэтому влияние затвора в плоском транзисторе рано или поздно дойдет до предела. Чтобы обойти это ограничение, разработаны варианты транзистора, где затвор тем или иным образом «охватывает» канал. Такая конструкция сложнее в изготовлении, однако влияние затвора на канал больше, а значит, можно добиться большей скорости переключения и/или уменьшить потребляемую мощность.

Подобные модификации транзистора появились, начиная с норм 16 нм, где был использован транзистор типа FinFET. Здесь аббревиатура FET означает Field Effect Transistor, а Fin в переводе означает «плавник», поскольку такая форма канала, выступающего вверх, напоминает спинной плавник рыбы.

На рис. 2.2 показаны варианты транзисторов.

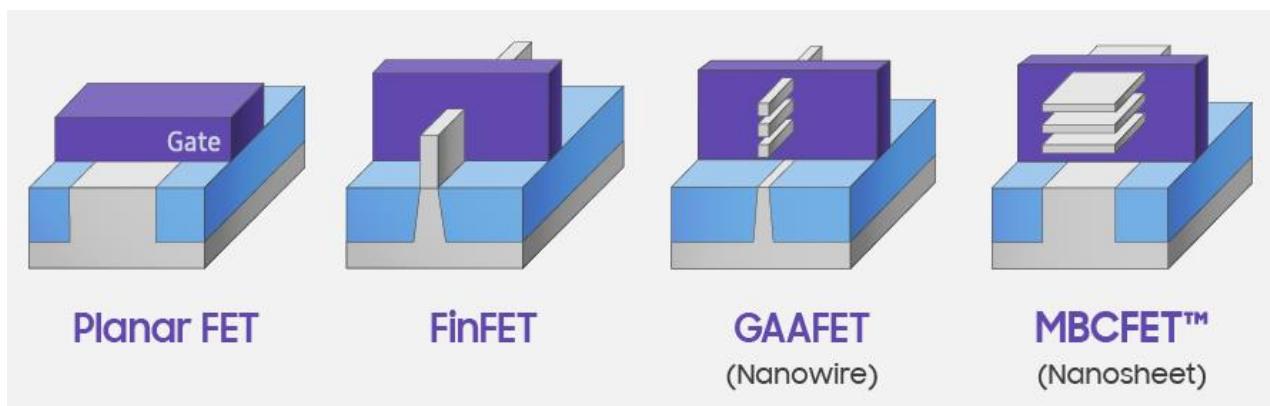


Рисунок 2.2. Варианты транзисторов, выполняемых по различным интегральным технологиям

Модификация GAAFET расшифровывается как Gate All Around, т.е. «полностью охватывающий затвор». Это следующий шаг по сравнению с FinFET. Аббревиатура MBCFET означает Multi-Bridge Channel. На рис. 2.2 видно, что новые модификации ориентированы на возможно более «плотный» охват канала затвором. На данный момент только технология FinFET получила устоявшееся применение, остальные же являются экспериментальными и находятся в стадии освоения. В целом можно отметить, что отклонения от плоскостного исполнения элементов транзистора усложняют технологические операции.

В таблице 2.1 приведены сравнительные характеристики современных технологических процессов.

Таблица 2.1. Некоторые характеристики современных технологических процессов

Год	2013	2015	2017	2019	2021
Обозначение	16/14	10	7	5	3
«Шаг» координатной сетки (half pitch), нм	40	32	25	20	16
Ширина транзистора FinFET, нм	7,6	7,2	6,8	6,4	6,1
Число логических вентилей 4И на кв. мм., млн	4	6,4	10,1	16,1	25,5
Напряжение питания транзисторов, В	0,86	0,83	0,8	0,77	0,74
Максимальное число слоев для трассировки	13	13	14	14	15
Длина затвора транзистора, нм	20	17	14	12	10

Анализ таблицы позволяет увидеть, что многие характеристики достигли насыщения и практически перестали изменяться. В то же время, например, плотность компонентов («логических вентилей 4И»), продолжает расти почти линейно. Это позволяет говорить о том, что следующее поколение технологических процессов обеспечивает значимое улучшение характеристик по сравнению с предыдущим, однако это изменение не всегда затрагивает каждый конкретный параметр.

2.3. Понятие технологического сдвига

В ряде источников используется термин «технологический сдвиг». Он иллюстрирует ситуацию в микроэлектронике, когда изменение технологических норм приводит к качественному изменению подходов к проектированию. Если говорить о всем многообразии цифровых схем, не все из них будут устойчиво работать по мере уменьшения технологических норм.

Важным этапом стал переход к нормам 130-90 нм. При этом задержки на проводниках стали сопоставимы с задержками на логических элементах (до этого задержки на проводниках были сравнительно малы). Поэтому ряд схем, созданных в предположении, что задержкой на соединительных проводах можно пренебречь, стали работать неустойчиво. В первую очередь это затронуло схемы со стробированием, формирователями импульсов, защелками, и другими узлами схем, где в расчет принималось только количество компонентов в цепочке преобразования сигнала. Вместо этого актуальным стал так называемый синхронный стиль проектирования схемы. Он подразумевает, что схема строится вокруг набора триггеров, тактируемых фронтом тактового сигнала, который подается на них одновременно. Правила синхронного проектирования будут рассмотрены далее в материалах курса.

Другим важным переходом стали нормы 28-16 нм. Влияние проводников стало еще больше, к тому же добавились существенные вариации параметров при изготовлении микросхем. Указывая определенные величины в проекте, сложно рассчитывать, что все микросхемы будут изготовлены в точности с этими параметрами. Подробнее о комплексе проблем излагается в разделе 2.6.

Значение технологического сдвига заключается в том, что в зависимости от используемого технологического процесса необходимо учитывать, какие проблемы возникнут при реализации проектируемой схемы. Необходимо применять соответствующие архитектурные решения, отвечающие требованиям технологического процесса. Практическим выводом является то, что описание модуля на языке описания аппаратуры, хотя и декларируется как переносимое

между технологиями и платформами, не всегда может быть реализовано с применением технологического процесса.

2.4. Потребление энергии интегральными схемами и понятие Power-Delay Product

Определение потребляемой энергии играет роль не только в определении стоимости эксплуатации микросхемы или времени ее автономной работы от аккумулятора. Важнейшим фактором здесь является плотность выделяемого тепла, которая экспоненциально возрастает по мере уменьшения технологических норм. Это представляет большую проблему, которая не вполне решается увеличением размера радиатора или установкой более мощного вентилятора для процессора.

Процесс передачи тепла описывается уравнением теплопроводности Фурье. Оно определяет, что при контакте двух тел количество передаваемой теплоты пропорционально площади контакта, коэффициенту теплопроводности, а также величине градиента температуры (т.е. разницы температур, деленной на расстояние между точками, где эта температура измерена). Таким образом, чтобы тепло передавалось от микросхемы к радиатору, микросхема должна иметь более высокую температуру. Кроме того, микросхема выделяет тепло не в районе крышки корпуса, где установлен радиатор, а на полупроводниковом кристалле. Поэтому даже при достаточно холодном радиаторе крышка корпуса будет иметь более высокую температуру, а кристалл внутри корпуса – еще более высокую. При превышении предельной температуры микросхема разрушится (причем достаточно превысить температуру в небольшой локальной области для ее выхода из строя). Все это обуславливает внимание к проблемам выделения тепла микросхемами.

Энергопотребление микросхем делят на две большие составляющие – статическое и динамическое потребление. Статическое потребление не зависит от режима работы микросхемы и существует всегда (хотя может быть и довольно небольшим). Динамическое потребление обычно пропорционально тактовой частоте, на которой работает микросхема, и обусловлено необходимость затрат энергии на переключение транзисторов и перезарядку емкостей, образующихся между проводниками внутри схемы.

Проекты на базе ПЛИС могут быть оценены с точки зрения потребляемой энергии. Такая оценка производится средствами САПР и использует достаточно простые, приближенные модели, на основе простого подсчета количества компонентов ПЛИС, использованных в проекте. Это дает приемлемую картину

потребления, позволяющую выбрать систему охлаждения. На рис. 2.3 показан пример расчета потребляемой энергии в САПР ПЛИС (Xilinx Vivado). Можно видеть, что выделены статическая и динамическая составляющие, а динамическая дополнительно разбита по группам компонентов в составе ПЛИС.

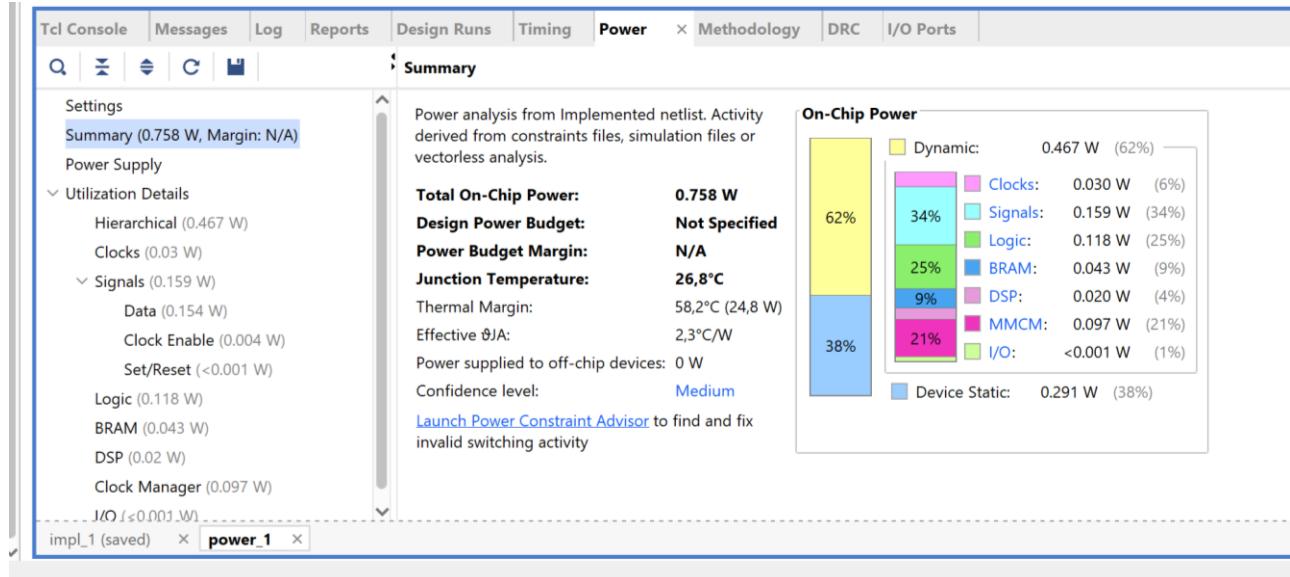


Рисунок 2.3. Пример расчета потребляемой энергии в САПР ПЛИС

Потребление компонентов ПЛИС определяется их конструкцией и зависит от того, какие именно варианты компонентов были применены производителем. В то же время в СБИС существуют библиотеки компонентов, содержащие разные варианты одного и того же элемента. Если обратиться к чертежу транзистора, можно увидеть, что путем изменения размеров затвора и диэлектрика можно регулировать задержку переключения, однако ценой потребляемой мощности. Например, если увеличить площадь затвора и уменьшить толщину диэлектрика, влияние затвора на канал станет сильнее, и переключение будет происходить быстрее. Однако такой затвор потребует больше энергии для переключения, а тонкий диэлектрик обусловит и больший ток утечки. В целом имеет место зависимость «больше быстродействие – больше энергопотребление». Пример такой зависимости показан на рис. 2.4.

На рис. 2.4 видно, что добиться снижения задержки ниже определенного порога не удается – потребление энергии начинает быстро возрастать, а задержка при этом так и не снижается. Аналогично, если увеличивать задержку, то энергопотребление падает до какой-то величины, после чего перестает снижаться. Это означает, что существует какое-то оптимальное (не обязательно в строгом математическом смысле слова) сочетание задержки переключения и энергопотребления.

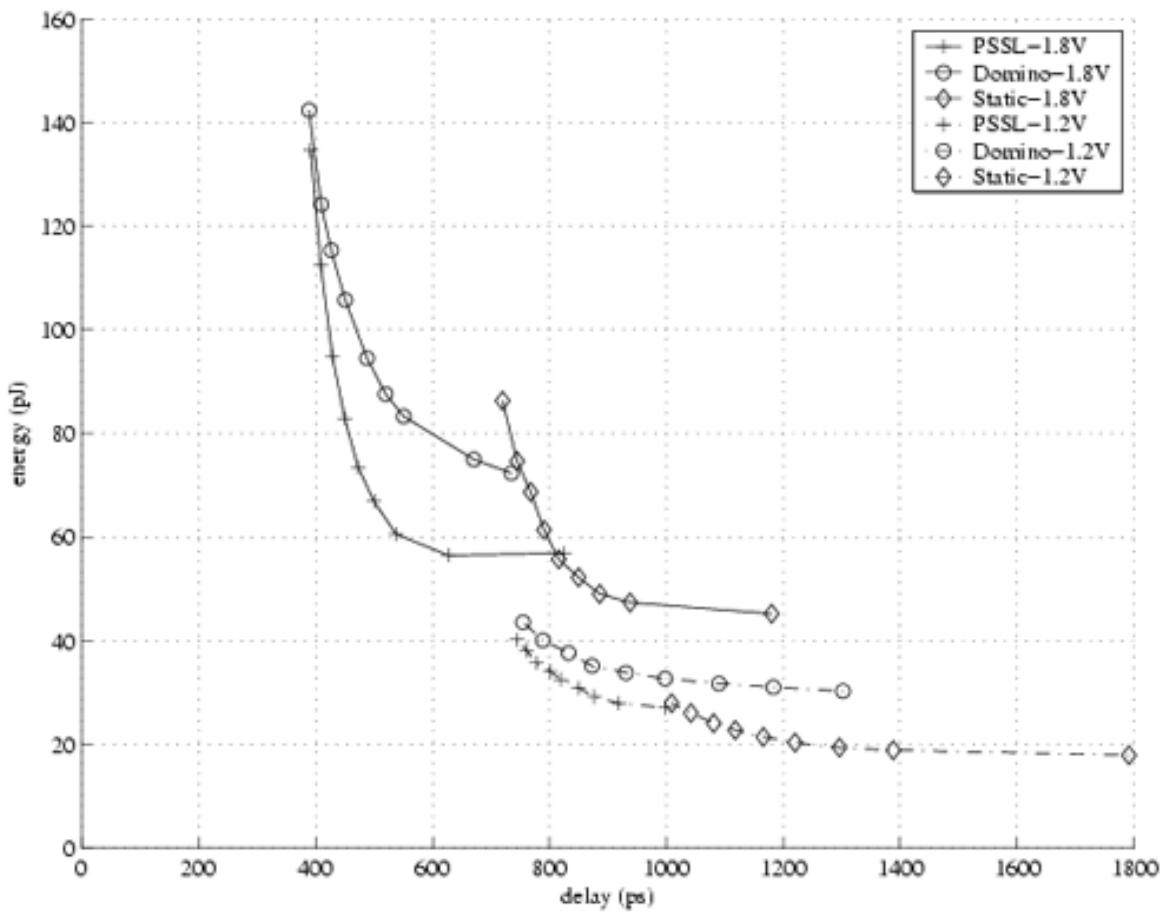


Рисунок 2.4. Пример исследования зависимости между задержкой распространения сигнала и энергией переключения

Можно оценить компонент по произведению задержки и энергии переключения (или мощности переключения). Это произведение называется Power-Delay Product (PDP), или Energy-Delay Product (EDP). Оно не имеет смысла для ПЛИС, поскольку компоненты ПЛИС уже выбраны производителем и имеют вполне определенные характеристики. Однако при разработке новой микросхемы необходимо избегать как слишком медленных компонентов, так и слишком много потребляющих. Как правило, речь идет не о подстройке каждого отдельного логического элемента, а о выборе наиболее подходящего варианта библиотеки компонентов. Кроме того, современные САПР обладают возможностями автоматической оптимизации фрагментов схем с выбором такого варианта, который не создает проблем из-за слишком большой задержки, и имеет при этом небольшое энергопотребление.

2.5. Современные тенденции и проблемы: темный кремний, GALS, стена памяти

Технологические процессы, начиная с норм приблизительно 28 нм, формируют ряд фундаментальных проблем, которые не могут быть автоматически обойдены в САПР по принципу «нажатия кнопки». Если в проекте изначально заложены решения, игнорирующие эти проблемы, в процессе работы специалисты неизбежно столкнутся с существенными сложностями, которые приведут к снижению характеристик микросхемы относительно ожидаемых.

Проблема «темного кремния» (dark silicon) заключается в том, что при большой плотности компонентов они выделяют слишком много тепла. Отдельный вопрос – повышение качества системы теплоотвода, однако для этого нужно обеспечить хороший тепловой контакт между полупроводниковой пластиной и радиатором. Это осложняется тем, что вещества, хорошо проводящие тепло, обычно хорошо проводят и ток. Поэтому в отрасли пока не удается разместить на кристалле большое количество одновременно работающих транзисторов.

Для того, чтобы снизить тепловыделение, можно уменьшить его динамическую составляющую. Например, если комплементарная пара транзисторов не переключается из одного логического состояния в другое, для нее отсутствует процесс протекания сквозного тока, который является составной частью динамического потребления. Этот процесс может быть специально запланирован – например, если в микросхеме имеются модули, которые заранее не работают одновременно, выделять тепло, соответственно, будут те модули, в которых действительно переключаются сигналы. Остальные (неработающие) модули в этом представлении выглядят «темными», т.е. неактивными или отключенными.

Эта проблема не имеет общепризнанного решения. Разработка должна планироваться таким образом, чтобы рядом не оказалось большого количества компонентов, которые будут работать одновременно. Если это происходит, в худшем случае на кристалле придется предусматривать промежутки между одновременно работающими модулями, что приводит к неоправданному росту площади, расходуемой впустую. В ряде случаев схема заранее работает так, что только часть компонентов активны в каждый момент времени. Например, арифметико-логическое устройство содержит несколько узлов, каждый из которых выполняет конкретную операцию. Если не изменять операнды на

входах тех узлов, которые не используются в настоящий момент, их энергопотребление будет снижено.

Высокоспециализированные схемы напротив, часто содержат только постоянно работающие компоненты. Для таких схем проблема темного кремния может быть существенной.

Аббревиатура GALS обозначает Globally Asynchronous, Locally Synchronous (глобально асинхронные, локально синхронные) схемы. Она описывает архитектуру микросхемы, в которой вынужденно присутствует несколько тактовых сигналов. Разработчик может запланировать один тактовый сигнал, однако если схема занимает большую площадь, такой сигнал может оказаться сложно развести по кристаллу на всей необходимой площади. На рис. 2.5 показан пример тактовой сети, проведенной в ПЛИС (линия выделена красным). Для такого сигнала нужно, чтобы он был проведен линиями приблизительно одинаковой длины, иначе разные компоненты будут срабатывать несинхронно. Один из способов достижения этого – проведение тактовых сигналов в форме, напоминающей ветки дерева (clock tree – тактовое дерево).

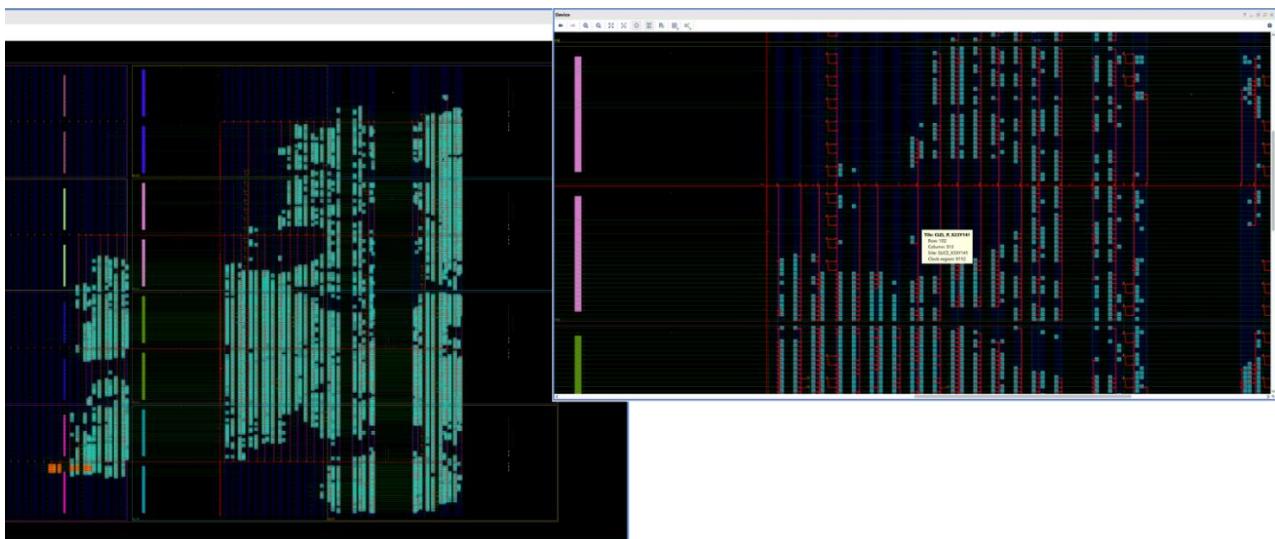


Рисунок 2.5. Пример трассировки тактовой сети в ПЛИС

Поскольку тактовое дерево по мере возрастания площади становится все сложнее реализовать, рано или поздно придется ограничить площадь, на которой это дерево размещено. Если микросхема все же должна иметь большую площадь, на ней можно разместить несколько тактовых сетей, которые будут размещены в независимых регионах. Компоненты в каждом из регионов работают синхронно (т.е. они «локально синхронны»), однако разные регионы используют разные тактовые сигналы («глобально асинхронны»). Пример микросхемы большой площади с несколькими тактовыми регионами показан на рис. 2.6.

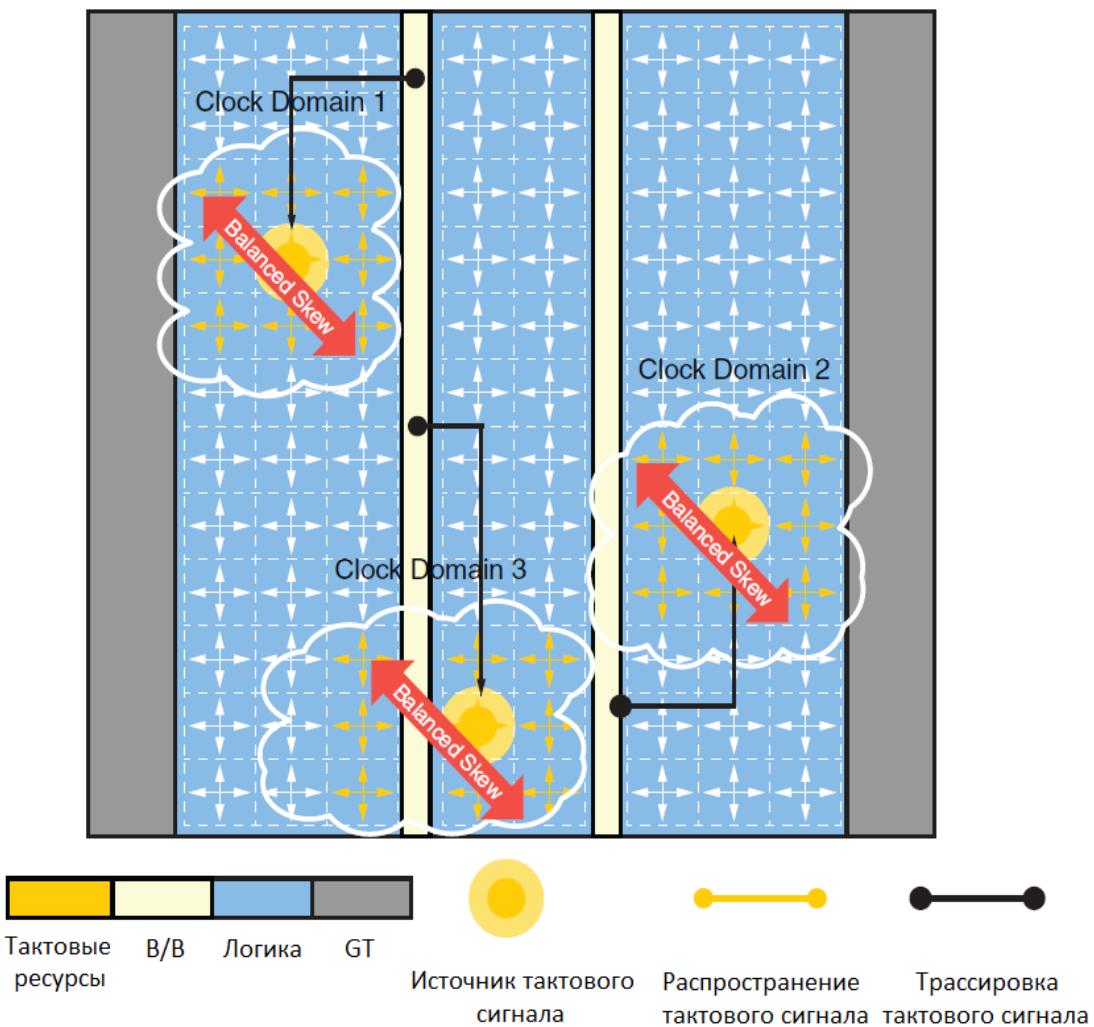


Рисунок 2.6. Архитектура тактовых ресурсов ПЛИС большого логического объема

Необходимость следования архитектуре GALS не следует из каких-либо теоретических положений цифровой схемотехнике. Можно описать схему, которая будет занимать большую площадь, и только при попытке реализовать тактовые сигналы станет понятно, что не удается обеспечить одновременное поступление тактового сигнала на все компоненты, которые этого требуют. Разделение микросхемы на несколько регионов помогает решить эту проблему.

Термин «стена памяти» и близкий к нему по смыслу термин «стена интерфейсов» отражают тот факт, что по мере возрастания размеров микросхемы производительность вычислений растет пропорционально квадрату размера (т.е. площади), однако данные из внешних источников (памяти и других интерфейсов) поступают через периферийные интерфейсы, пропускная способность которых растет пропорционально не квадрату, а линейному размеру кристалла. Эта проблема носит фундаментальный характер, поскольку небольшие колебания (например, улучшение характеристик конкретного

внешнего интерфейса) не может принципиально изменить общую картину. Например, кристалл, состоящий из двумерной сетки процессорных ядер, потребует передачи большого потока данных для обработки. Для ядер, расположенных на периферии кристалла, это возможно, однако ядра, расположенные внутри, вынуждены получать данные, пропускаемые через соседние ядра, или же простаивать.

Проблема «стены памяти» также не имеет универсального общепризнанного решения. Она несколько смягчена для микросхем, которые выполнены по архитектуре «системы на кристалле» (System On Chip, SOC). В этой архитектуре предполагается, что основные потоки данных передаются внутри микросхемы, между ее компонентами, а внешняя память большого объема не так важна. Архитектура класса «система на кристалле» не имеет единственно возможного варианта реализации, этот термин используется для большого класса микросхем и его конкретные признаки несколько размыты. В целом считается, что к СНК можно отнести микросхему, в которой используется несколько разнородных подсистем, образующих готовое устройство – например, процессор, памяти и периферийные контроллеры.

2.6. Синхронный стиль проектирования

Начиная с технологических норм 130 – 90 нм, в цифровой электронике стало важным использовать синхронный стиль проектирования. Если ранее задержки распространения сигналов в схеме можно было достаточно хорошо предсказывать на основе подсчета количества элементов на отдельных участках схемы, то при переходе к указанным нормам задержки распространения на логических элементах уменьшились, а задержки в соединительных линиях в целом изменились несущественно. Поэтому более надежным вариантом стало применение синхронных схем. Они основаны на триггерах, изменяющих свое состояние по фронту тактового сигнала (D-триггерах). Между триггерами находятся комбинационные схемы, состоящие из логических вентилей и подобных им компонентов. По фронту тактового сигнала все триггеры записывают новое значение, определяемое их входом данных. После этого на входах комбинационных схем появляются новые значения, и они начинают изменение своих выходов с определенной задержкой. К моменту прихода следующего фронта тактового сигнала все триггеры должны получить правильное значение на своих входах.

При таком подходе для схемы появляется понятие «максимально допустимая тактовая частота». Если превысить ее, для какого-то триггера

окажется, что на его входе еще не появилось правильное новое значение, и схема в целом будет работать неправильно. Однако это же позволяет и простым способом обеспечить работоспособность схемы – понизить тактовую частоту до величины, когда период тактового сигнала будет превышать максимальную задержку распространения, которая есть в этой схеме.

На рис. 2.7 показан пример цифровой схемы, соответствующей требованиям синхронного стиля проектирования. В этой схеме используется один тактовый сигнал, поданный на все триггеры, между которыми размещены комбинационные узлы (показанные овалами).

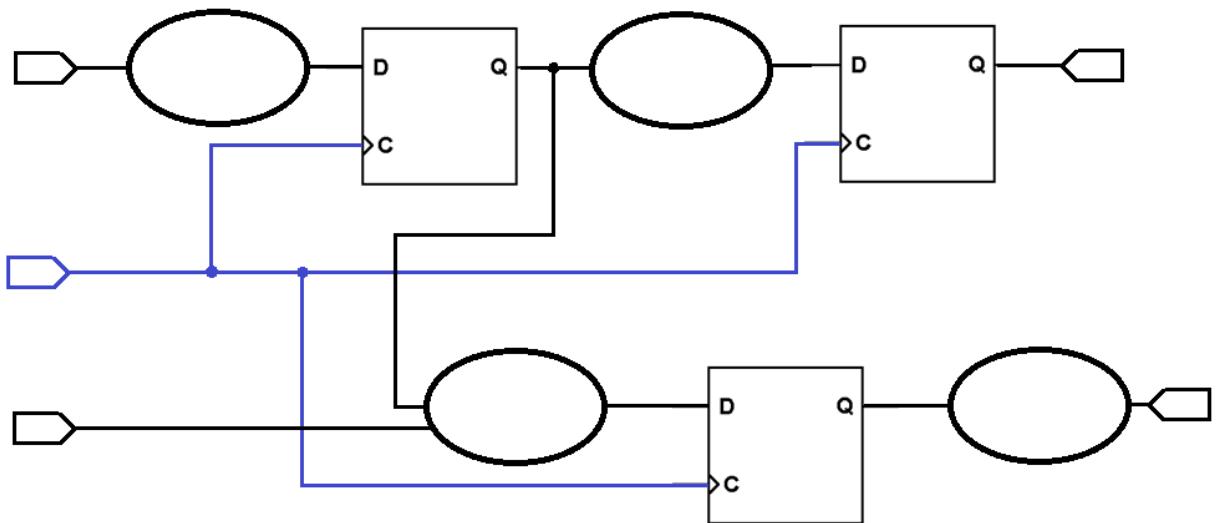


Рисунок 2.7. Пример цифровой схемы, соответствующей требованиям синхронного стиля проектирования

Для практического использования синхронного стиля необходимо использовать соответствующие компоненты, специально предназначенные для работы с тактовыми сигналами. Например, в ПЛИС для этого существуют аппаратные модули, которых не появляются автоматически при переносе проекта в СБИС. Кроме того, за применением этих модулей в проекте необходимо следить, иначе тактовые сигналы будут проведены не с применением специально размещенных в ПЛИС тактовых деревьев, а с помощью обычных трассировочных линий, которые не обеспечивают одновременную подачу фронта тактового сигнала на триггеры ПЛИС.

На рис. 2.8 показан специализированный компонент для работы в тактовых сетях ПЛИС Xilinx – BUFGCTRL. Он имеет множество функций, и при

необходимости реализует некоторые типичные сценарии работы с тактовым сигналом.

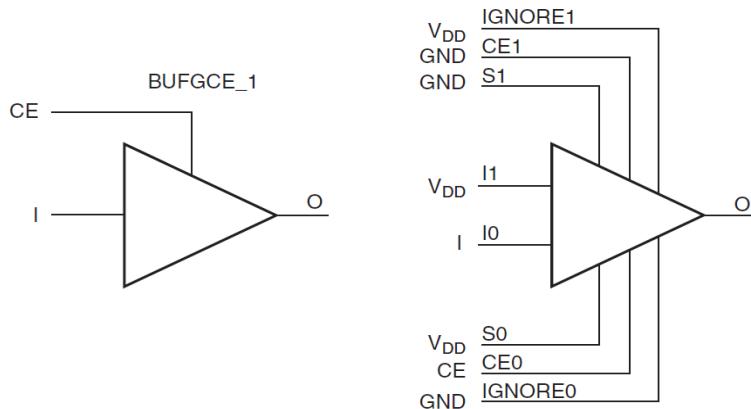


Рисунок 2.8. Специализированный компонент для работы в тактовых сетях – BUFGCTRL (ПЛИС Xilinx)

На рис. 2.9 показаны временные диаграммы компонента BUFGCTRL при управляемом отключении тактового сигнала. Может показаться, что можно использовать вентиль И, на один из входов которого подать тактовый сигнал, а на второй – сигнал управления. Если этот сигнал равен 1, то выход И будет определяться тактовым сигналом, т.е. просто копировать его, но если подать на управляющий вход 0, то для элемента И значение выхода в любом случае станет равно 0. Однако это пример сценария, когда применять обычный вентиль И нельзя из-за потенциальных неопределенностей в работе схемы. Если отключить тактовый сигнал в произвольный момент, может оказаться, что уже начавшийся выходной импульс логической единицы окажется слишком коротким. Поскольку разные триггеры микросхемы имеют технологический разброс параметров, какие-то из них могут среагировать на короткий импульс, а для каких-то его длительности окажется недостаточно. Это приведет к непредсказуемой работе схемы.

Аналогичная ситуация может возникнуть при включении тактового сигнала в произвольный момент времени. Поэтому компонент BUFGCTRL при изменении значения управляющего сигнала нормальным образом завершает текущий импульс логической единицы или же не начинает новый импульс, если включение произошло в момент, когда логическая единица уже была на выходе.

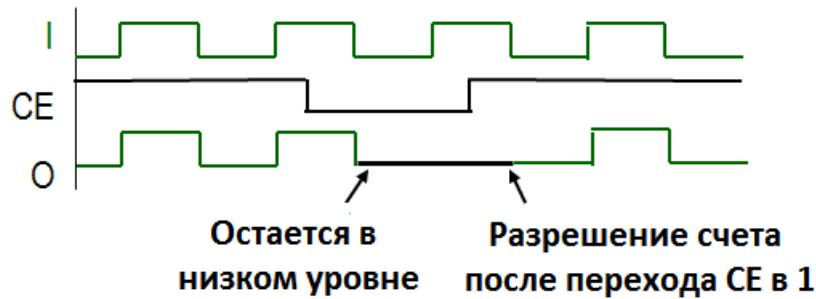


Рисунок 2.9. Временные диаграммы компонента BUFGCTRL в режиме управляемого отключения тактового сигнала

Похожая проблема возникает при переключении между двумя тактовыми сигналами. Если произвести его в произвольный момент времени, нельзя гарантировать, что на выходе не окажется короткий импульс логической единицы, который приведет к срабатыванию только некоторых триггеров схемы. Аналогично, компонент BUFGCTRL в режиме переключения между тактовыми сигналами гарантирует нормальное завершение текущего импульса и переключается на второй тактовый сигнал только в момент, когда этот вход находится в состоянии логического нуля.

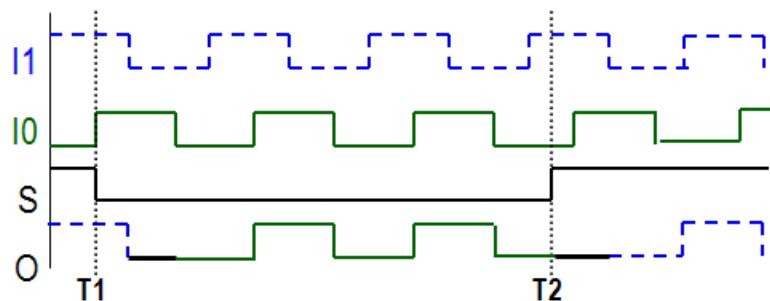


Рисунок 2.10. Временные диаграммы компонента BUFGCTRL в режиме переключения (мультиплексирования) тактовых сигналов

Компонент BUFGCTRL является примером узла, который требуется для нормальной работы цифровой схемы, однако не является автоматически переносимым между проектами. Например, при описании поведения тактового сигнала на уровне RTL невозможно указать, что конкретный сигнал должен проходить через специальный компонент (если только не использовано так называемое структурное описание схемы). Более того, сам факт наличия компонентов для управления тактовым сигналом зависит от аппаратной платформы – серии ПЛИС, ее производителя (например, компонент BUFGCTRL, описанный для ПЛИС Xilinx, отсутствует в ПЛИС Intel), а для СБИС такие компоненты могут требовать приобретения отдельных лицензий.

В целом сведения о специальных компонентах для управления тактовым сигналом приведены для того, чтобы проиллюстрировать практические сложности преобразования абстрактной цифровой схемы в реально работающее устройство. Теоретическое проектирование может проводиться в предположении, что все сигналы идеально соответствуют абстрактным моделям, компоненты срабатывают мгновенно, а тактовый сигнал подается на все триггеры в строго один и тот же момент времени. На практике же это не так. Для ПЛИС требуется выполнение рекомендаций производителя по применению специальных компонентов для управления тактовым сигналом, а для СБИС само наличие этих компонентов зависит от технологического процесса и имеющихся для него библиотек.

Для синхронного стиля проектирования существует ряд типичных ошибок, которые неочевидны при рассмотрении идеализированных моделей, однако они проявляются, когда эта схема оказывается реализованной в конкретной микросхеме (например, ПЛИС).

На рис. 2.11 показаны некорректный (сверху) и корректный вариант схемы делителя частоты.

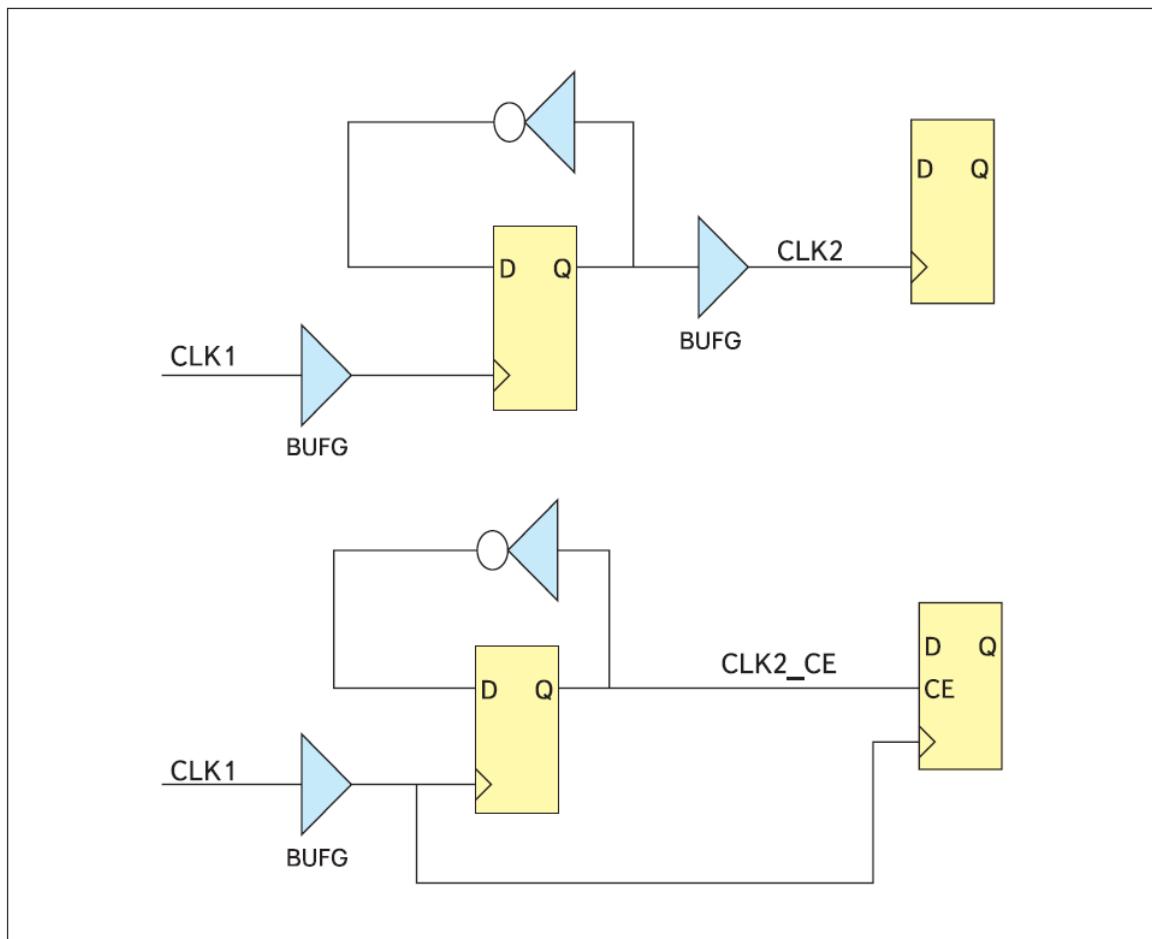


Рисунок 2.11. Некорректная (сверху) и корректная (снизу) схемы делителя частоты

Предполагается, что второй триггер должен переключаться в два раза реже по сравнению с первым. Формально эта схема является работоспособной, и она будет работать, если собрать ее из дискретных компонентов. Однако можно заметить, что второй триггер на верхнем варианте схемы тактируется сигналом CLK2, который, хотя и подан через специальный тактовый буфер BUFG, отстает по фазе от сигнала CLK1. Если триггеры будут срабатывать в разные моменты времени, это не обязательно приведет к ошибке, однако опасность подобных решений именно в том, что появление проблем непредсказуемо и зависит от технологического разброса параметров компонентов, которые использованы для реализации такой схемы.

В варианте, показанном внизу, оба триггера используют один и тот же тактовый сигнал. Рассчитывая на то, что тактовая сеть в микросхеме реализована правильно и обеспечивает минимальное рассогласование по времени между приходом фронта на разные триггеры, можно полагать, что оба триггера срабатывают одновременно. Вместо того, чтобы подавать тактовый сигнал с частотой в 2 раза меньше, используется сигнал Clock Enable («разрешение счета»), который подается в 2 раза реже. Поэтому, несмотря на то что тактовый сигнал второго триггера имеет такую же частоту, как и первого, второй триггер может реагировать только на один фронт, а следующий будет пропускать, поскольку сигнал Clock Enable будет переключен в неактивное состояние.

На рис. 2.12 показан другой вариант схемы, которая формально является корректной, но может вызвать проблемы из-за реальных характеристик компонентов. Предполагается, что выходная частота в 16 раз меньше входной. Это планировалось выполнить следующим образом: 4-разрядный двоичный счетчик по мере счета перебирает выходные состояния от 0000 до 1111 (всего 16 состояний). Если состояние счетчика равно 1111 (1510), вентиль И формирует выходной сигнал логической 1, который в момент появления воспринимается триггером как фронт тактового сигнала.

Корректность работы такой схемы определяется характеристиками соединительных проводов. Например, если один из проводников короче остальных, переход от 0111 к 1000 может произойти так, что сначала изменится старший разряд, а потом остальные. Поэтому изменение произойдет в порядке 0111 – 1111 – 1000. Если бы элемент И имел большую задержку переключения, а проводники передавали сигналы за пренебрежимо маленькое время (как это было для технологических норм 250 или 180 нм), выход элемента И не успел бы переключиться в состояние логической 1, пока на его входах существовала комбинация 1111. Однако по мере уменьшения времени срабатывания

логических элементов и увеличения технологического разброса уже не следует полагаться, что элемент И не успеет сформировать короткий выходной импульс («иголку»).

Важно, что проявление этого эффекта нестабильно и зависит как от экземпляра микросхемы, так и от температуры и колебаний напряжения питания. От такого влияния практически свободны синхронные схемы. Даже если на выходе логического элемента появятся «иголки», триггер запишет новое состояние по фронту тактового сигнала, который придет заведомо позже, когда эффекты «иголок» уже завершатся.

Корректная схема использует один и тот же тактовый сигнал для счетчика и триггера. Как и в предыдущем примере, вместо вмешательства в тактовый сигнал работа триггера приостанавливается с помощью сигнала Clock Enable.

Для обозначения типичных ошибок организации тактирования применяются следующие понятия:

- *gated clock* (дословно – «тактовый сигнал, управляемый вентилем») – схема, в которой элемент И используется для отключения тактового сигнала, недостаток этой схемы заключается в том, что выходной сигнал смещен по времени относительно входного;

- *divided clock* («деленная тактовая частота») – схема, показанная на рис. 2.11, использующая схему Т-триггера (триггер, переключающийся каждым тактом в противоположное состояние); как и в предыдущем примере, недостатком является смещение по времени выходного тактового сигнала;

- *derived clock* («производный тактовый сигнал») – тактовый сигнал, получаемый из выходных сигналов счетчика с помощью логических элементов; аналогичный недостаток – смещение по времени выходного тактового сигнала.

Для синхронного стиля проектирования часто используется конвейер – последовательная передача данных от одного триггера к другому (возможно, с обработкой комбинационной логикой). Для такой схемы особенно важно, чтобы вся цепочка триггеров использовала один и тот же тактовый сигнал. Тогда можно будет достаточно просто анализировать ее работу, считая, что сигнал, поданный на первый в цепочке триггер, каждым тактом будет продвигаться на одну ступеньку дальше. Иллюстрация показана на рис. 2.13.

Конвейеризация активно используется в цифровых схемах, позволяя разбить сложную операцию на последовательность простых. Если для вычисления сложного выражения необходимо использовать слишком много логических элементов, которые в сумме будут иметь большую задержку, тактовая частота такой схемы окажется небольшой.

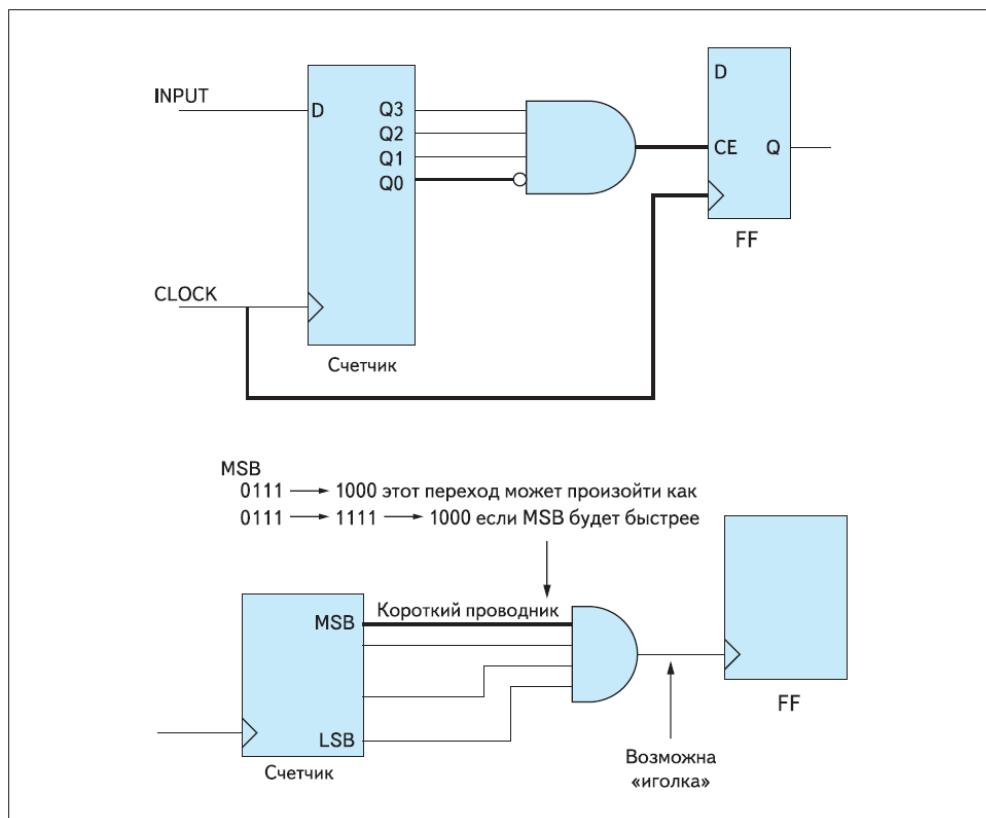


Рисунок 2.12. Некорректная (снизу) и корректная (сверху) схемы делителя частоты

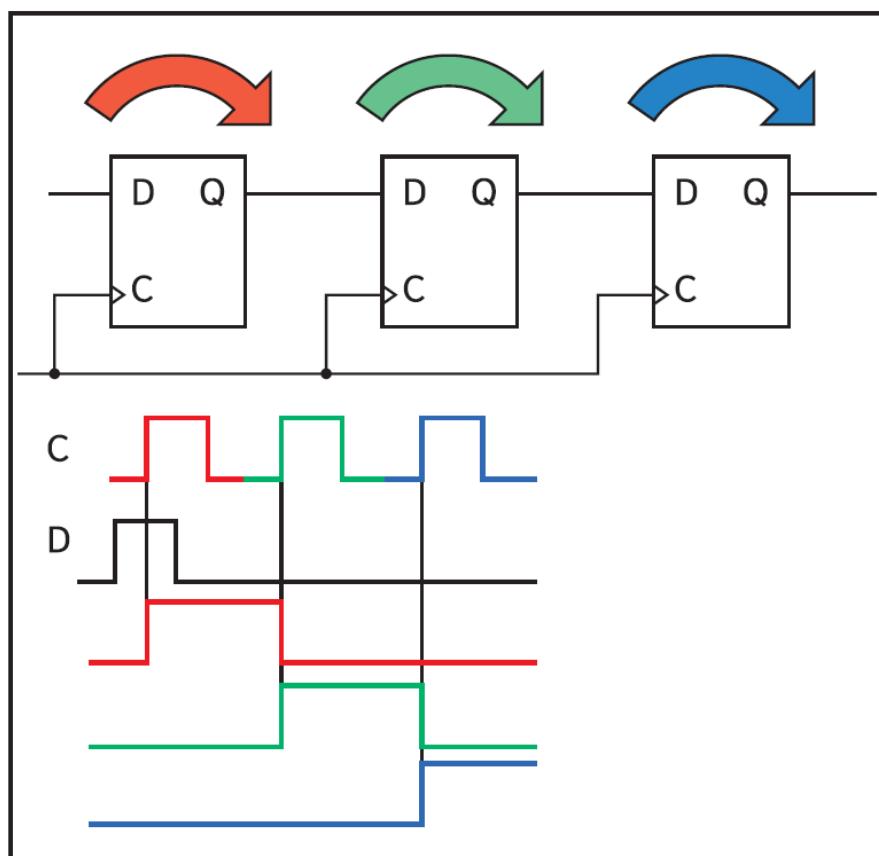


Рисунок 2.13. Иллюстрация к понятию конвейеризованной синхронной схемы

Конвейеризация позволяет выполнить только часть преобразований, записав промежуточный результат в триггеры, а на следующем такте продолжив вычисление в другой части комбинационной схемы. Этот прием помогает увеличить тактовую частоту, а использование одного и того же тактового сигнала делает такую схему стабильной и хорошо моделируемой современными САПР.

2.7. Выводы по главе

Современные тенденции микроэлектроники показывают регулярное уменьшение технологических норм. При этом далеко не все параметры микросхем получают пропорциональное улучшение. Прежде всего, сохраняется рост плотности компонентов на единицу площади кристалла, однако тактовая частота повышается непропорционально. Вместе с этим растут стоимость подготовки производства и тепловыделение.

Фундаментальными проблемами являются «темный кремний», «стена памяти» и сложность реализации тактового сигнала для большой площади кристалла, что привело к появлению архитектуры GALS (глобально асинхронные, локально синхронные схемы). Эти проблемы не имеют универсального решения и должны учитываться при выборе архитектуры вычислительного устройства.

Особенности технологического процесса изготовления цифровых микросхем обуславливают применение определенных подходов к проектированию. Увеличение технологического разброса характеристик компонентов заставляет переходить к синхронному стилю проектирования, в котором триггеры схемы используют один и тот же тактовый сигнал.

Контрольные вопросы:

1. Что такое норма технологического процесса?
2. Как изменяются плотность компонентов, задержка распространения сигнала, энергопотребление и стоимость при переходе к меньшим нормам технологического процесса?
3. Почему планарная структура транзистора стала заменяться на другие?
4. Что такое «темный кремний»?
5. Представив 256-ядерный процессор в виде матрицы 16x16 ядер, какие проблемы можно предполагать при его проектировании?

3. ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ

3.1. Основные сведения

Программируемые логические интегральные схемы (ПЛИС) – отдельный подкласс микросхем, представляющих собой набор цифровых компонентов, соединения между которыми могут программироваться уже после ее изготовления. Это позволяет создавать в ПЛИС практически произвольные цифровые схемы, что делает ее удобной для создания макетов будущих микросхем с целью их проверки. Кроме того, для небольших партий изделий, где не подходят универсальные процессоры, оригинальная схема, реализованная в ПЛИС, может оказаться эффективной с точки зрения производительности и цены.

Общим термином для ПЛИС является PLD – Programmable Logic Device. Этот термин отражает саму возможность программирования соединений между компонентами микросхемы. В зависимости от типа компонентов и соединений ПЛИС могут иметь различные архитектуры. Например, одной из первых является SPLD – Simple Programmable Logic Device. Ее развитием стала CPLD – Complex Programmable Logic Devices, которая до сих пор применяется. Более массовая и известная архитектура – FPGA (Field-Programmable Gate Array). Этот термин нуждается в пояснении. Прежде всего, gate array – это, по сути, «микросхема» (дословно – «массив вентилей»). Понятие field-programmable дословно переводится как «программируемый в поле». Термин in-the-field, от которого образовано field programmable, обозначает устройства, которые программируются не на предприятии-изготовителе, а пользователем (т.е. «в поле», за пределами завода).

Архитектура FPGA подразумевает размещение матрицы логических ячеек, между которыми проложены соединительные линии («трассировочные»). Параметры самих ячеек могут настраиваться, подключение к трассировочным линиям также управляется. Переключение трассировочных линий и настройка режима работы ячеек производится путем загрузки в FPGA специального конфигурационного файла. Обычно формат этого файла закрыт производителем во избежание восстановления схемы по имеющемуся или прочитанному из FPGA файлу.

Схема взаимодействия логических ячеек в ПЛИС с архитектурой FPGA показана на рис. 3.1.

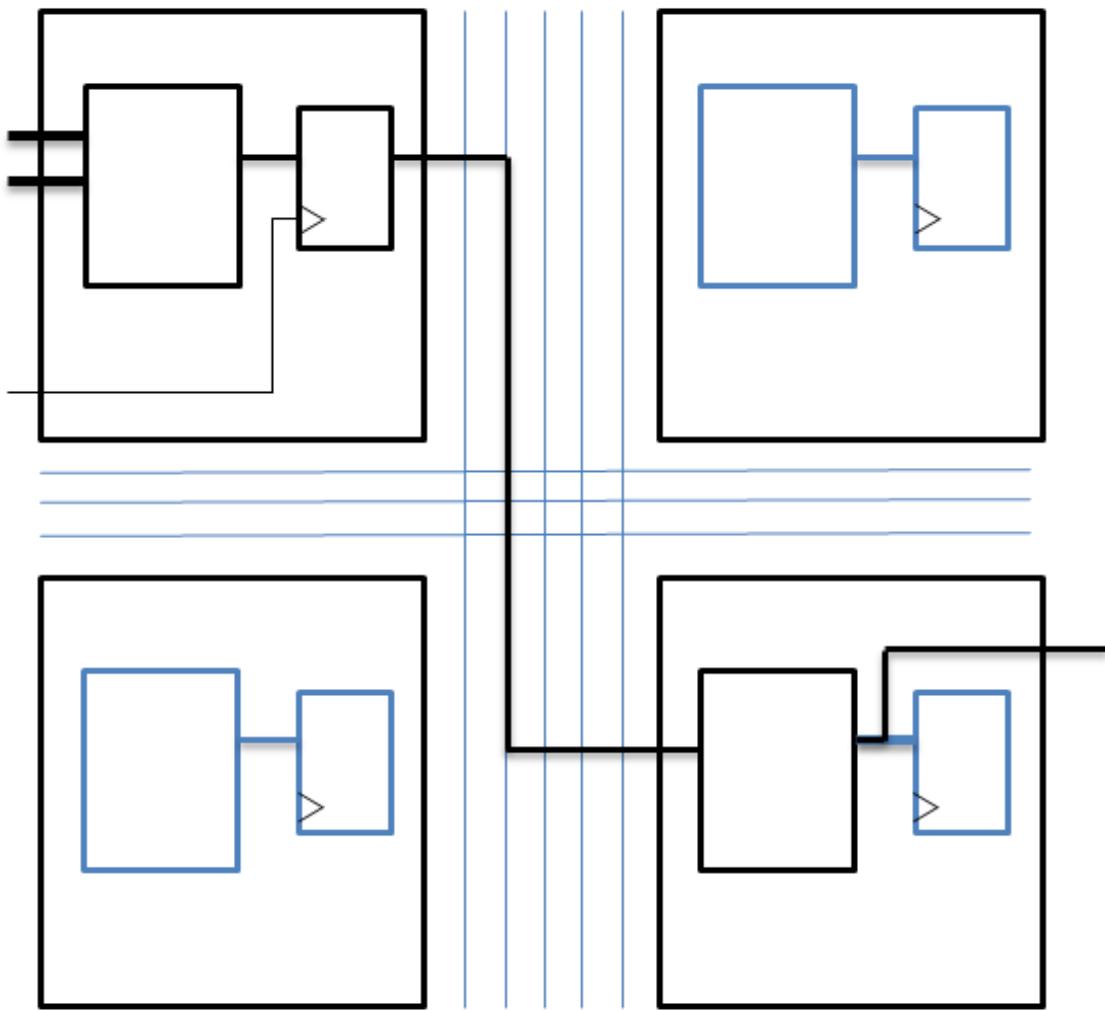


Рисунок 3.1. Схема взаимодействия логических ячеек в ПЛИС с архитектурой FPGA

На рис. 3.1 видно, что выход одной из ячеек (слева сверху) подключен к одной из вертикальных трассировочных линий и идет вниз, подключаясь к входу правой нижней ячейки. Во второй ячейке не используется триггер, ее выходом является выход блока комбинационной логики.

Управление ячейками и трассировочными линиями крайне редко производится вручную. Создание файла с конфигурационными данными для загрузки в FPGA выполняется в САПР в автоматическом режиме.

Логические ячейки и трассировочные линии – не единственные компоненты FPGA. Пример простой ПЛИС серии Spartan-3 производства Xilinx показан на рис. 3.2.

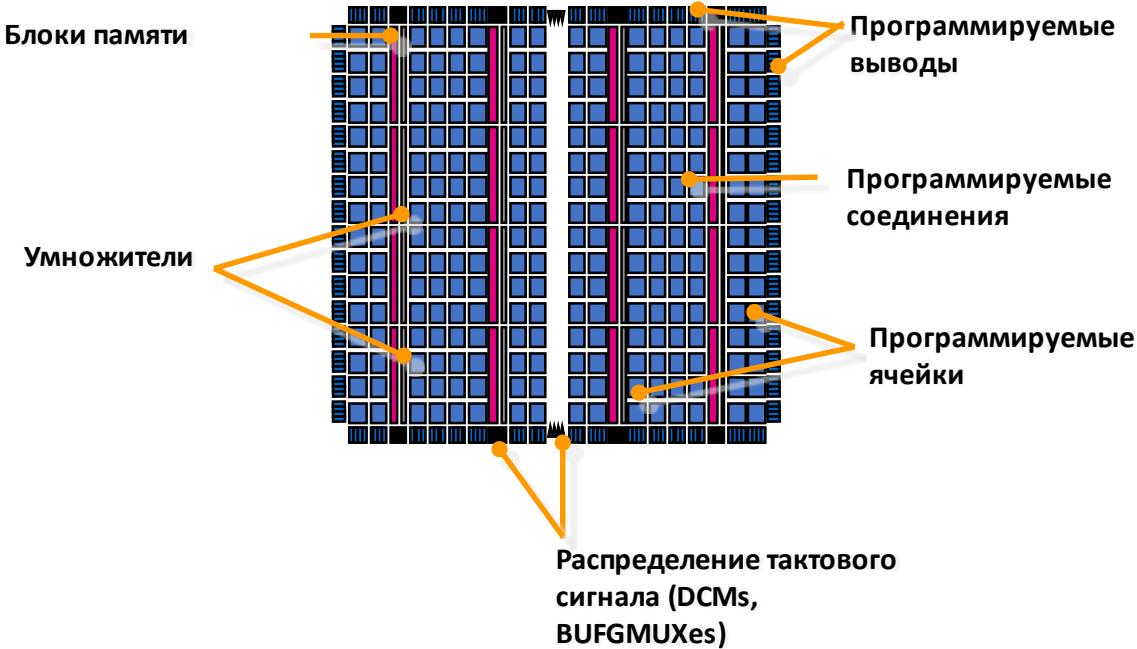


Рисунок 3.2. Архитектура ПЛИС Xilinx с архитектурой FPGA серии Spartan-3

Важнейшим дополнением к матрице ячеек и трассировочных линий являются внешние выводы. Они также являются настраиваемыми (программируемыми), поскольку заранее неясно, будет ли конкретный вывод входом или выходом. Кроме направления передачи данных, у выводов FPGA можно также настраивать реализуемые электрические интерфейсы. Не все выводы FPGA являются программируемыми, часть выводов используется для подключения питания, загрузки конфигурации, а в более сложных FPGA существуют специальные выводы с фиксированными функциями.

На рис. 3.2 также видны примеры дополнительных компонентов, которые часто являются основой для построения эффективных схем в ПЛИС. В Spartan-3 это блоки памяти и блоки умножения независимых operandов. Такие блоки можно было бы реализовать и путем соединения логических ячеек, но в этом случае возможности ячеек будут использованы нерационально. Память, реализованная в виде компактного блока, займет существенно меньше места, чем память такого же объема, реализованная в виде настроенных и соединенных логических ячеек. Аналогично, умножение вида $a*b$ можно сделать и путем соединения логических ячеек, но готовый аппаратный модуль займет меньше места и будет работать быстрее. К моменту появления серии Spartan-3 стало понятно, что память и блоки умножения требуются во многих проектах, и если добавить к FPGA аппаратные (непрограммируемые) блоки, производительность и возможности существенно вырастут.

Отдельно показаны компоненты для формирования тактовых сигналов и их распределения по кристаллу FPGA. Эти компоненты относятся не к «ускоряющим работу», а к обязательным для применения.

На рис. 3.3 показана архитектура ПЛИС Xilinx серии Kintex-7. Сравнение с рис. 3.2 показывает, что дальнейшее развитие архитектуры FPGA привело к появлению большего количества аппаратных блоков и увеличению их разнообразия.

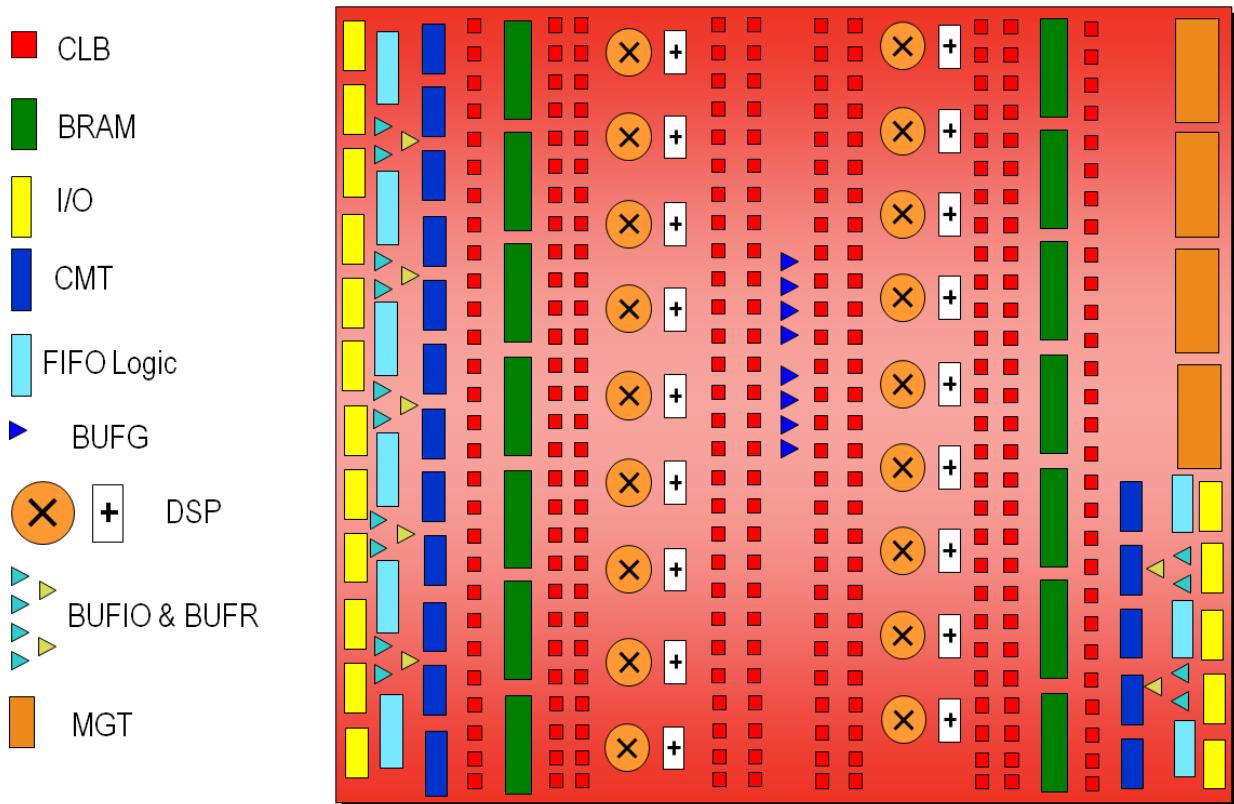


Рисунок 3.3. Архитектура ПЛИС серии Xilinx Kintex-7

Новым по сравнению со Spartan-3 является компонент MGT (Multi-Gigabit Transceiver). Это блок, который способен работать с высокоскоростными («мультигигабитными») сигналами, которые передаются по дифференциальным парам. Примерами интерфейсов, которым требуются MGT, являются PCI Express, Serial ATA (SATA), 10G Ethernet и множество интерфейсов со скоростями передачи данных 1-10 Гбит/с и более (современные интерфейсы достигают скоростей 116 Гбит/с). Такой сигнал невозможно обработать на базе цифровых компонентов, а тем более путем соединения логических ячеек, которые заведомо работают медленнее из-за более длинных и сложных трассировочных линий. Поэтому блок, который может полностью выполнить прием и передачу сигнала для высокоскоростной линии, помещается на кристалл

ПЛИС и подключается к трассировочным линиям с помощью параллельных интерфейсов. Блоки MGT появились в FPGA в период возникновения потребности в магистральном сетевом оборудовании. Например, магистральный сетевой коммутатор может оперировать десятками оптоволоконных линий, для которых необходимо произвести маршрутизацию пакетов. ПЛИС большого логического объема с блоками MGT широко используются для разработки сетевого оборудования, которое требуется в относительно небольших количествах (поэтому разработка специальной микросхемы экономически не оправдана).

3.2. Архитектура логической ячейки

Понятие «логическая ячейка» (logic cell) для FPGA подразумевает комбинацию из логического генератора и триггера. Пример структурной схемы логической ячейки FPGA (соответствующей Kintex-7 и более поздним) показан на рис. 3.4.

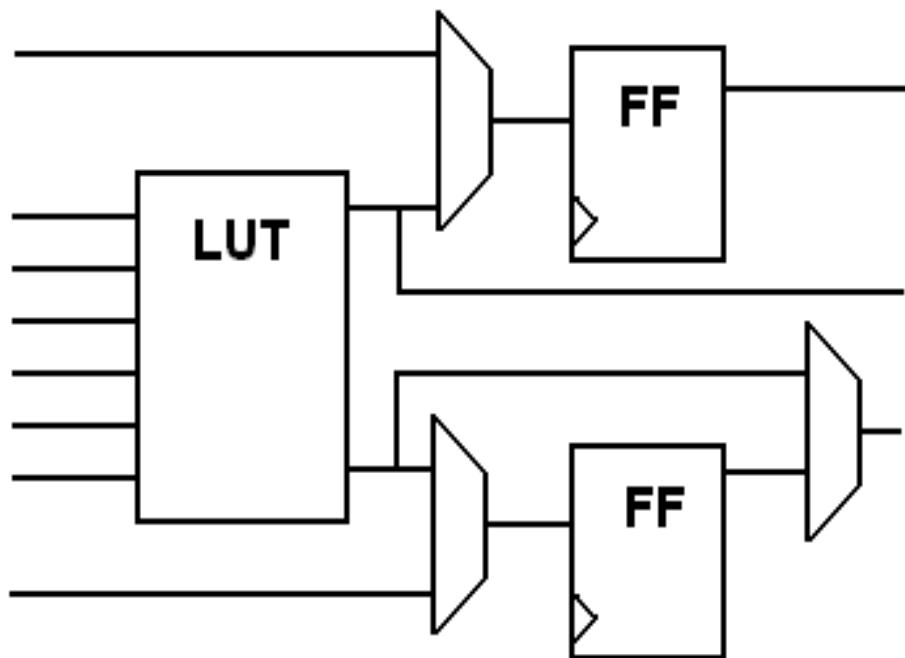


Рисунок 3.4. Структурная схема логической ячейки FPGA

Генератор произвольных логических функций обозначен на рисунке как LUT (Look-Up Table, «таблица истинности»). В действительности он не содержит внутри логические элементы, а представляет собой обычную память. Входы блока являются адресными линиями этой памяти, а выход – это выход данных. Если составить желаемую таблицу истинности для такого блока и записать ее в память, то при работе FPGA подача комбинации входных сигналов будет выбирать соответствующую ячейку памяти. Можно представлять, что

внутри блока размещена соответствующая схема из логических элементов, однако это просто память с N входами, в которую записаны нужные реакции на каждую входную комбинацию, как показано на рис. 3.5.

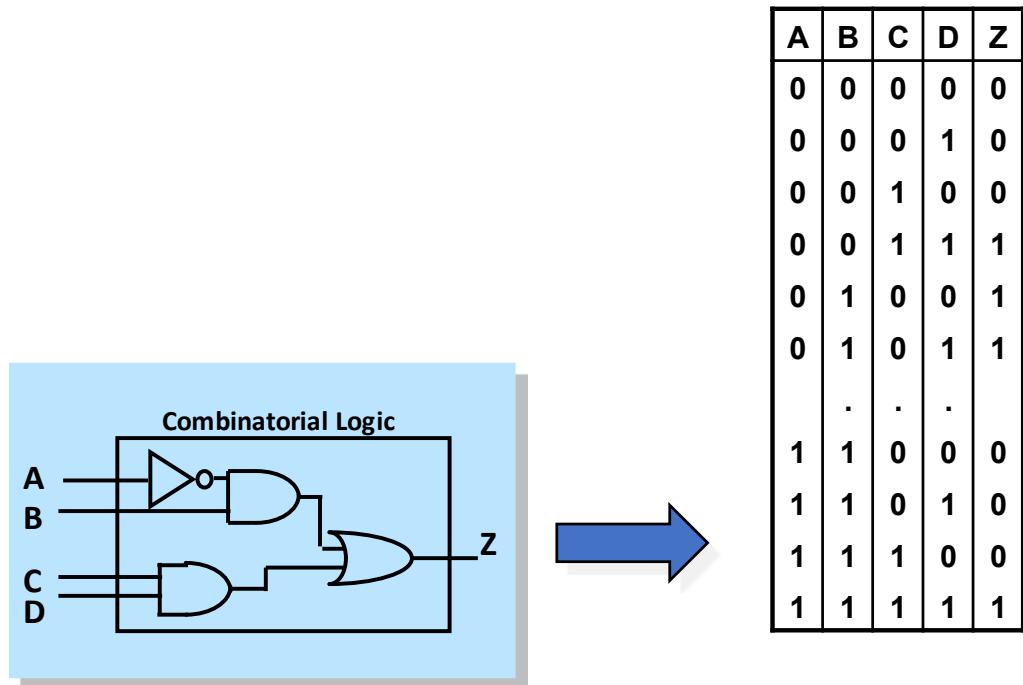
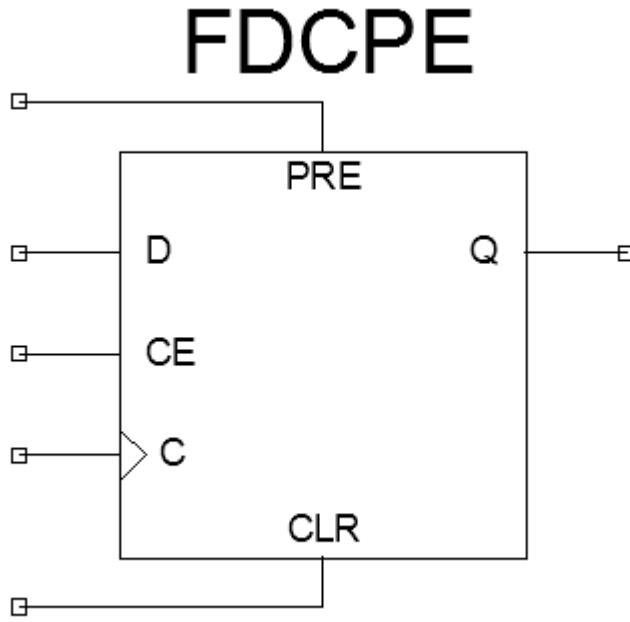


Рисунок 3.5. Принцип реализации комбинационных схем на базе таблицы истинности

Количество входов в LUT существенно определяет, схемы какой сложности можно реализовать. Долгое время промышленным стандартом была 4-входовая LUT. Во многих случаях схему выражают в количестве «эквивалентных логических ячеек», под которыми понимают набор из 4-входовой LUT и триггера. На рис. 3.4 можно видеть, что современные FPGA используют 6-входовые LUT и два триггера.

Вторым компонентом ячейки является триггер. Его условное графическое обозначение показано на рис. 3.6, а временные диаграммы, иллюстрирующие работу – на рис. 3.7. Показанное условное графическое изображение отражает представление САПР



*Рисунок 3.6. Условное графическое изображение триггера в составе ячейки
FPGA*

На рис. 3.7 показаны основные сценарии работы триггера. Входы CLR (очистка) и PRE (предустановка) являются асинхронными, т.е. появление сигнала логической единицы на них приводит к немедленному изменению состояния триггера. Сигнал CLR приводит к сбросу (переходу выхода в 0), а PRE – к переходу выхода в 1. Современные FPGA обычно позволяют физически реализовать только один из этих сигналов (кроме того, асинхронный сброс или установка триггера настоятельно не рекомендуется).

Если ни один из сигналов CLR, PRE неактивен, то триггер записывает состояние входа данных D в момент появления фронта тактового сигнала. Однако дополнительный сигнал CE (Clock Enable) позволяет разрешать или запрещать реакцию триггера на тактовый сигнал. На рис. 3.7 видно, что в момент времени 25 нс на входе C был фронт тактового сигнала, однако в этот момент на входе CE был 0, поэтому триггер не отреагировал на фронт и не изменил свое состояние. В момент времени 35 нс сигнал CE был в состоянии 1 («счет разрешен»), поэтому выход триггера Q записал уровень 1, который был в этот момент на входе D.

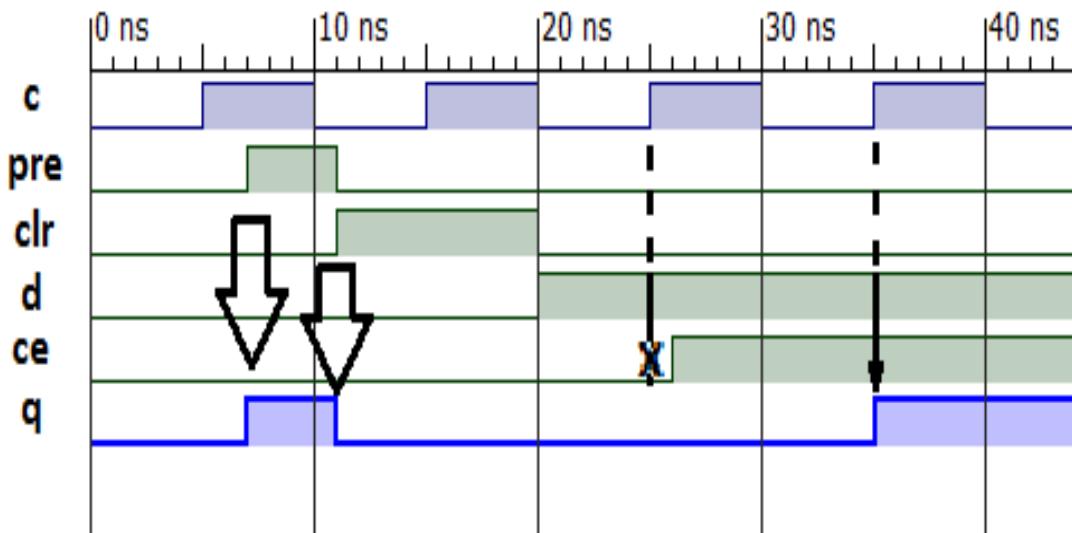


Рисунок 3.7. Временные диаграммы работы триггера

Логические ячейки объединяют в FPGA в более крупные структуры – секции (slice) и конфигурируемые логические блоки (Configurable Logic Block, CLB). На рис. 3.8 показана структура логической секции FPGA Xilinx серии 7. В секции размещены 4 логические ячейки и несколько дополнительных компонентов. Например, каждая пара логических генераторов объединена мультиплексором, а два мультиплексора объединены третьим. Это упрощает реализацию логических функций с большим количеством входов. Другим компонентом, который относится к секции, а не какой-то отдельной ячейке, является цепь ускоренного переноса (fast carry chain). На рисунке она видна как вертикальная цепочка вентилей. Назначением этой цепочки является передача между разрядами бита переноса при реализации операций сложения и вычитания. Ее роль подобна аппаратным блокам. В принципе, можно реализовать блок вычисления переноса и на базе LUT, но это приведет к нерациональному использованию ресурсов ПЛИС. В то же время вычисление бита переноса занимает немного места, а требуется достаточно часто. Поэтому такие цепочки добавлены в секции FPGA и используются при построении сумматоров и вычитателей по мере необходимости.

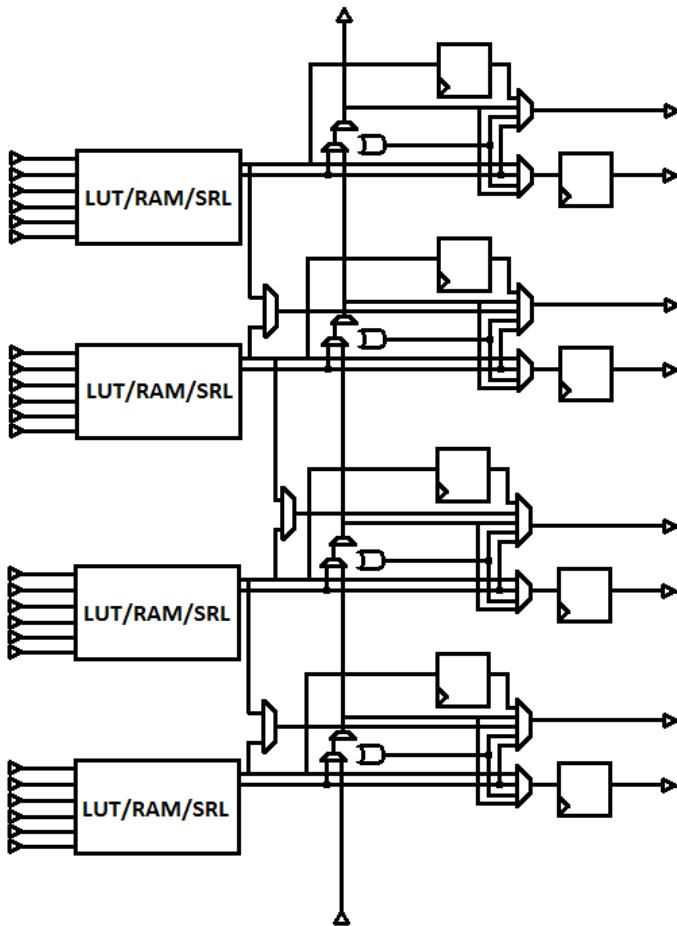


Рисунок 3.8. Структура логической секции ПЛИС FPGA (Xilinx, серия 7 и последующие)

3.3. Дополнительные компоненты FPGA

Рассмотренные выше дополнительные компоненты FPGA активно используются в тех проектах, где FPGA являются именно серийной компонентной базой, а не макетом будущего изделия. Обычно ПЛИС по сравнению с микроконтроллерами или процессорами проигрывает по ряду характеристик. Если сравнить универсальную микросхему (ПЛИС) и специализированный процессор, изготовленный по сопоставимому технологическому процессу, будет видно, что ПЛИС оказывается медленнее и дороже. Это является следствием того, что схема в ПЛИС создается на базе конфигурируемых логических ячеек, которые работают в определенном режиме, а значит, используются не полностью. Частота работы схемы в ПЛИС меньше, потому что соединения между ячейками выполняются с помощью конфигурируемых трассировочных линий, а не металлических проводников, проводимых строго между точками, которые необходимо соединить.

Вместе с тем, возможность создания схемы с нестандартной архитектурой является сильной стороной ПЛИС. На рис. 3.8 показана иллюстрация к проблеме

создания систем цифровой обработки сигналов. Такие технологии, как беспроводные сети новых поколений (4G, 5G), обработка изображений высокого разрешения, видеопотоков и ряд других, требуют все возрастающей производительности специального подкласса вычислений, которые обозначаются как «цифровая обработка сигналов» (ЦОС). Это достаточно обширный класс алгоритмов, однако с точки зрения вычислительной техники в качестве ключевой характеристики указывают операцию умножения с накоплением (Multiply and Accumulate, MAC). Эта операция вида $s = s + k*x$ требует блока умножения, который сложнее в реализации, чем остальные базовые арифметические операции. Обычно разместить множество блоков умножения в процессоре общего назначения оказывается нерационально, потому что нет гарантии, что все программы будут их эффективно использовать. На практике производители микроконтроллеров воздерживаются от поддержки параллельного выполнения множества операций умножения с накоплением, потому что нет гарантии, что эти ресурсы существенно повысят производительность программ.

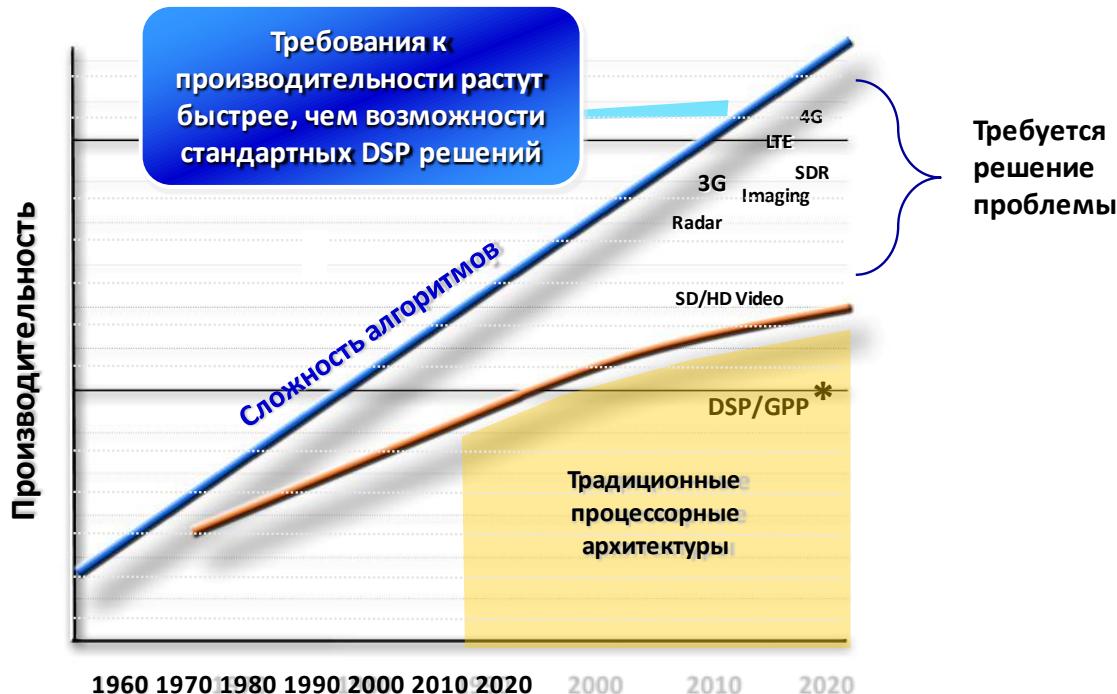


Рисунок 3.8. Иллюстрация к областям применения ПЛИС в задачах цифровой обработки сигналов

Вместе с этим для ПЛИС такая проблема стоит не так остро. Размещенные в ПЛИС блоки, аппаратно выполняющие умножение с накоплением, можно соединить требуемым образом с помощью настраиваемых трассировочных

линий. Блоки могут быть распределены между несколькими схемами, объединены, отключены и т.д.

Эволюция блоков цифровой обработки сигналов показана на рис. 3.9.

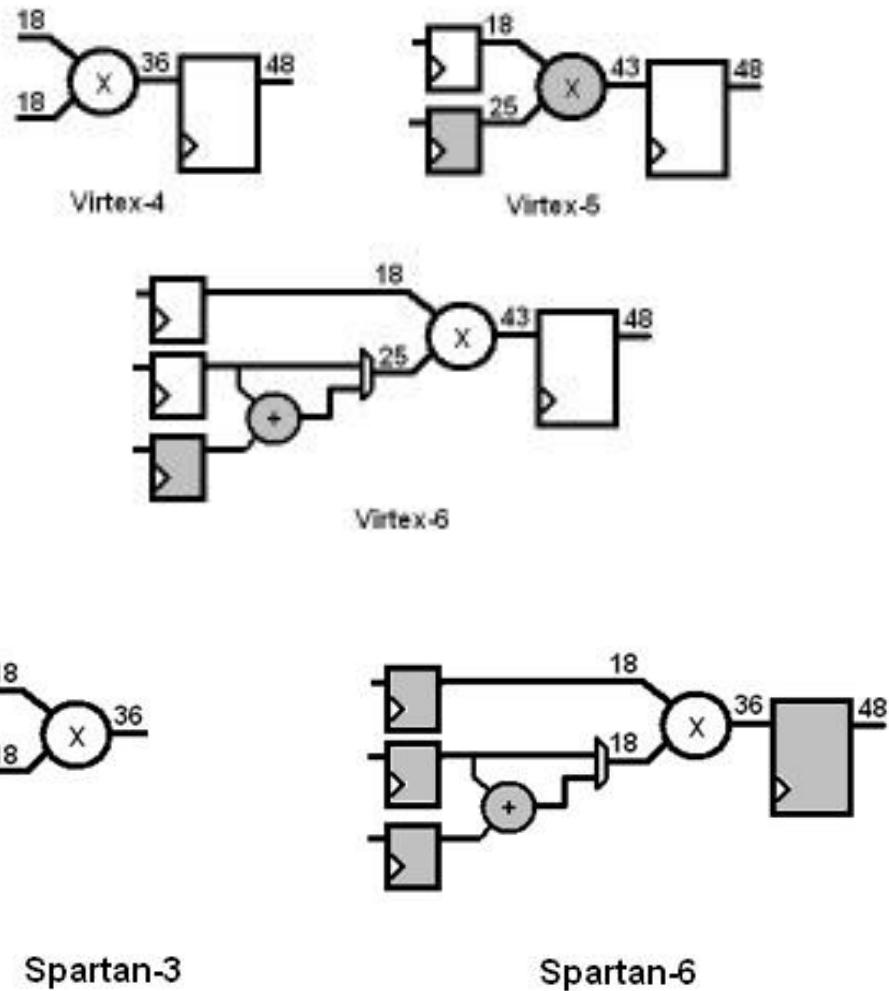


Рисунок 3.9. Блоки цифровой обработки сигналов в ПЛИС Xilinx

В простых ПЛИС Spartan-3 блок цифровой обработки представляет собой просто аппаратный умножитель. Он может перемножить два 18-разрядных операнда, давая на выходе 36-разрядное произведение. Преимуществом такого блока является его компактность по сравнению с такой же схемой, реализованной на базе логических ячеек. Кроме того, ячейки, соединяемые конфигурируемыми трассировочными линиями, дают существенно большую задержку при вычислении произведения, поэтому умножитель, реализующий только одну операцию, оказался быстрее, меньше по площади и энергопотреблению. Однако практическое применение ПЛИС быстро показало, что операция умножения часто сопровождается операцией накопления (сложения с аккумулятором). Аккумулятор можно реализовать на базе ячеек, но он сразу же снижает тактовую частоту. Поэтому первым же шагом в развитии блока цифровой обработки сигналов стало добавление аппаратного аккумулятора, способного работать на такой же высокой частоте, что и умножитель. С учетом того,

что сложение двух 36-разрядных чисел может дать 37 разрядов, для аккумулятора требуется разрядность больше, чем выход умножителя. Первое поколение блоков цифровой обработки сигналов в серии Virtex-4 имело 18-разрядные операнды для умножения и 48-разрядный аккумулятор для накопления суммы. Иногда используется запись вида $18 \times 18 = 48$, которая показывает, что перемножаются два 18-разрядных операнда, а аккумулятор имеет 48 разрядов.

Одним из самых известных алгоритмов цифровой обработки сигналов является фильтр с конечной импульсной характеристикой (КИХ-фильтр, также FIR – Finite Impulse Response). Его структурная схема показана на рис. 3.10.

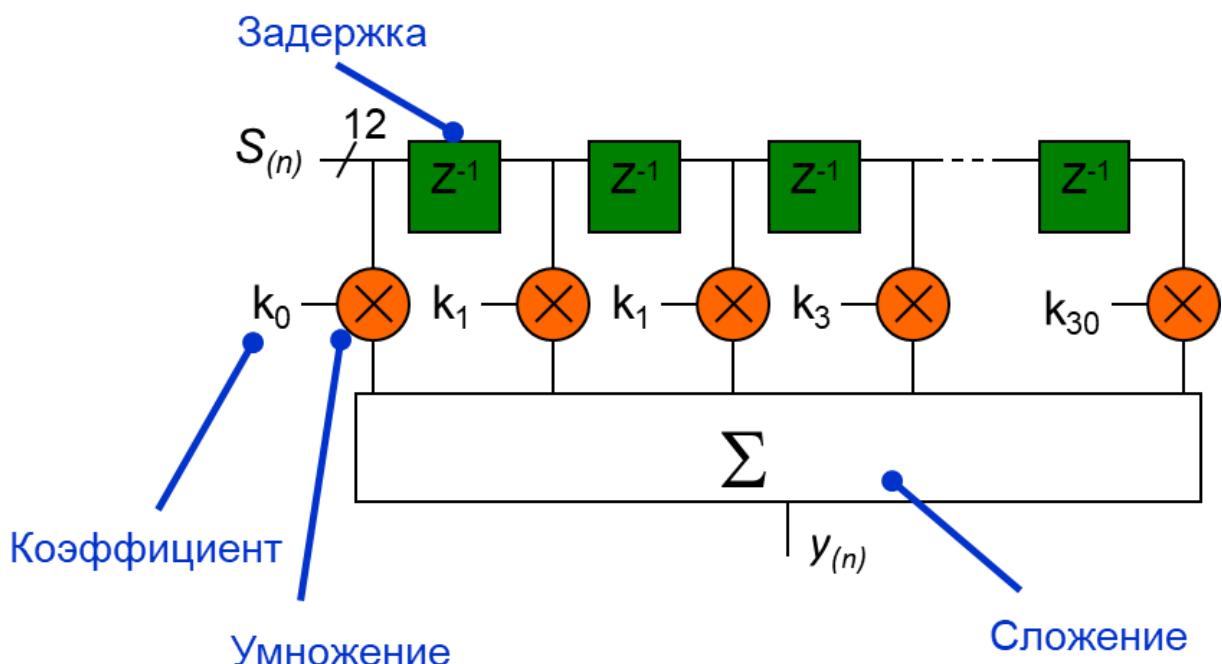


Рисунок 3.10. Структурная схема цифрового фильтра с конечной импульсной характеристикой

Фильтр реализует вычисление вида:

$$y = \sum_{i=1}^n x_i \cdot k_i$$

Величины x_i – подаваемые на вход фильтра значения входного сигнала. Они также называются «отсчетами» (sample). Коэффициенты k являются постоянными и определяют поведение фильтра. Количество коэффициентов в фильтре называется *порядком фильтра* (filter order).

Увеличение порядка фильтра позволяет проводить более сложные преобразования с входным сигналом. Поэтому фильтры с большим количеством коэффициентов (а следовательно, и операций умножения) представляют

практический интерес для обработки сигналов. При этом не слишком эффективно использовать для такой операции процессоры общего назначения. Для вычисления выходного значения фильтра потребуется выполнять операции умножения n последних отсчетов на соответствующие коэффициенты, т.е. организовать цикл на n итераций. Это пропорционально снижает производительность процессора, даже если одна итерация выполняется быстро.

Поскольку все операции для вычисления выходного значения фильтра можно выполнять одновременно, это делает FPGA удобной аппаратной платформой для цифровой фильтрации. Если необходимо реализовать фильтр с n коэффициентами, можно использовать n аппаратных блоков, выполняющих умножение с накоплением. При этом тактовая частота такого фильтра остается высокой.

Цифровые фильтры являются широко распространенными вычислительными модулями, поэтому их разработка часто автоматизируется в САПР. На рис. 3.11 показан генератор IP-ядра КИХ-фильтра, который генерирует RTL-представление такого компонента, основываясь на таких входных параметрах, как порядок фильтры, разрядность коэффициентов и т.п.

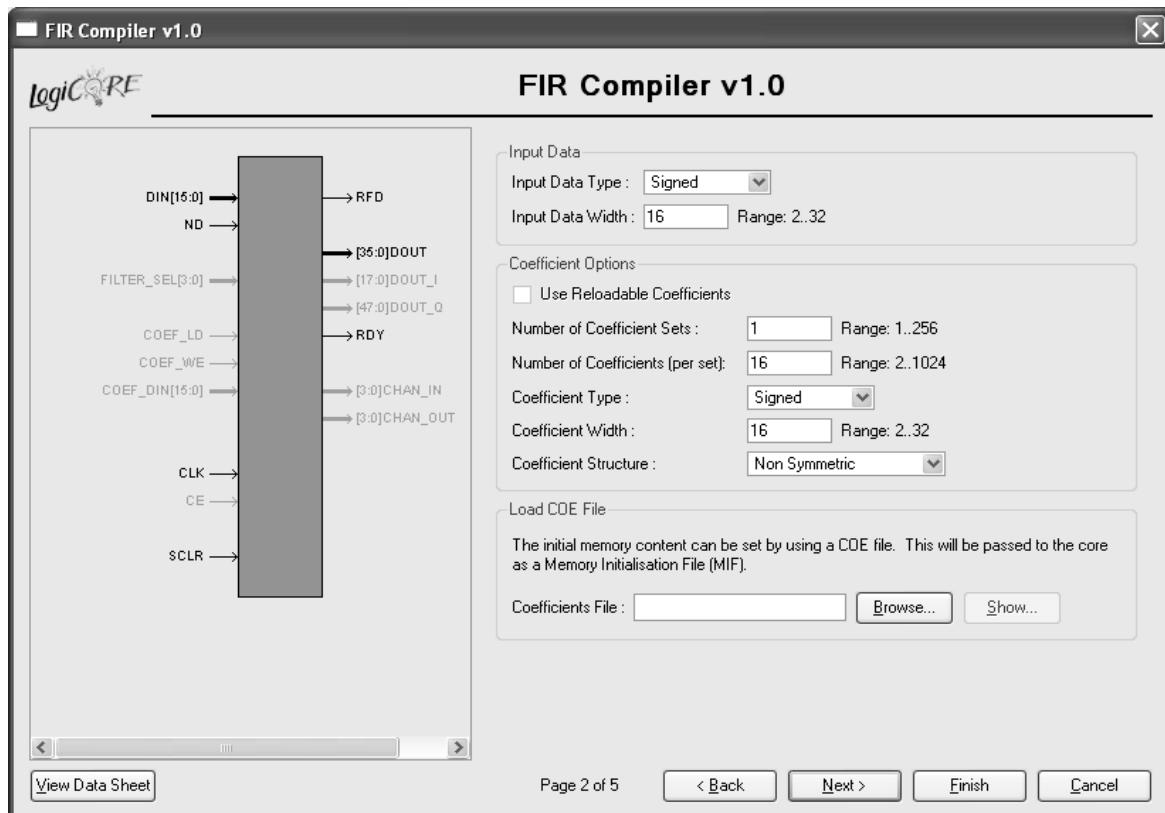


Рисунок 3.11. Генератор IP-ядер цифровых фильтров в составе САПР ПЛИС

Определение коэффициентов фильтра – отдельное крупное направление в цифровое обработку сигналов. Кроме обширного теоретического материала, в этой области существует большое количество программного обеспечения, выполняющего расчеты коэффициентов фильтра на основе их желаемых характеристик. Пример такого инструмента показан на рис. 3.12.

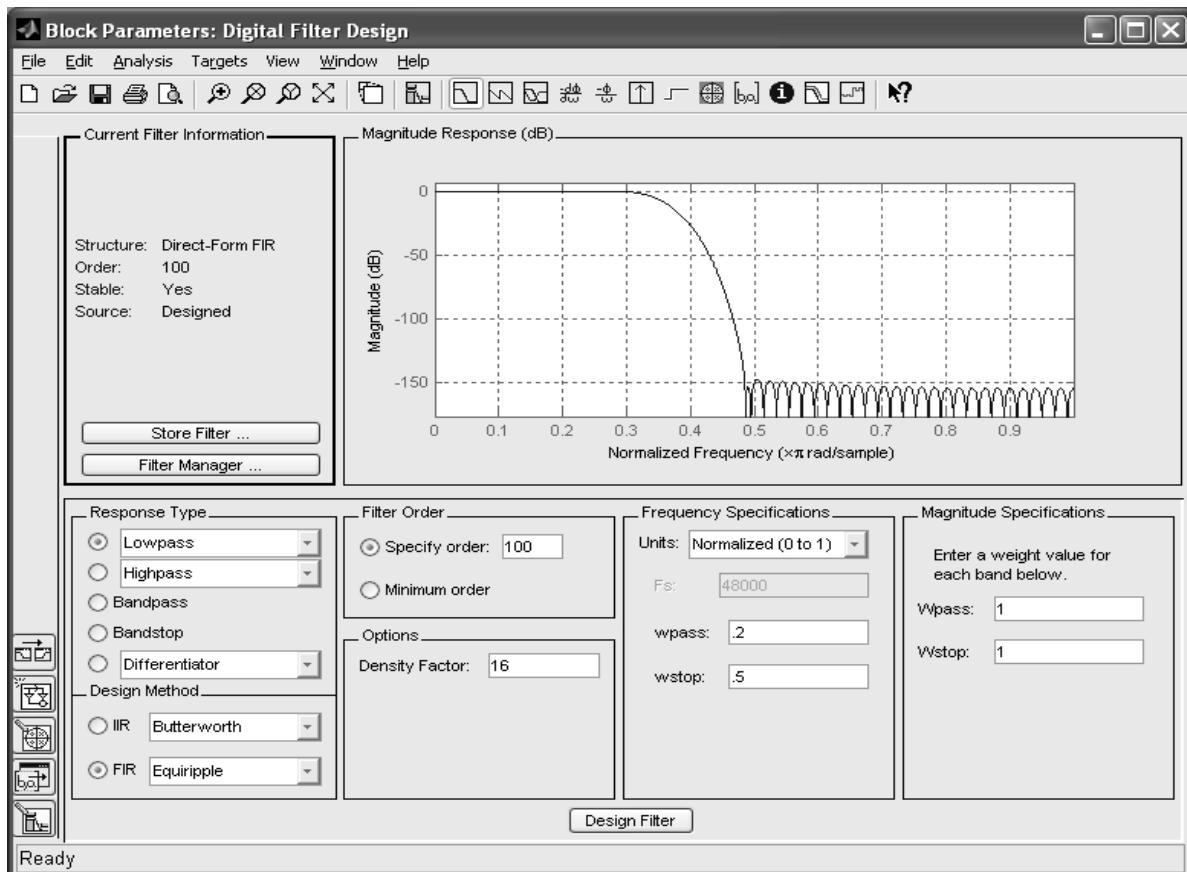


Рисунок 3.12. Создание цифрового фильтра в программном пакете Matlab

Другим типом аппаратного блока, который существенно улучшает характеристики проекта в ПЛИС по сравнению с конфигурируемыми ячейками, является блочная память (Block RAM, BRAM). Несмотря на то, что в ячейках FPGA имеются триггеры, которые могут хранить данные, остальная площадь ячейки используется для других целей (исключением является специальный режим распределенной памяти, доступный для FPGA Xilinx). Если описать большой блок памяти с расчетом на его реализацию в ячейках, FPGA будет использована малоэффективно. Если бы внутри был блок обычной памяти, он бы занял существенно меньше места при том же объеме в битах. Именно с этой целью в FPGA добавляют блоки статической памяти, которые занимают существенно меньше места по сравнению с ячейками, которые могли бы хранить такой же объем данных.

Графическое представление блока памяти в FPGA Xilinx показано на рис. 3.13. Это не единственный возможный вариант блока памяти, однако именно в этом виде он реализуется в нескольких семействах FPGA. Другие производители используют похожие решения.

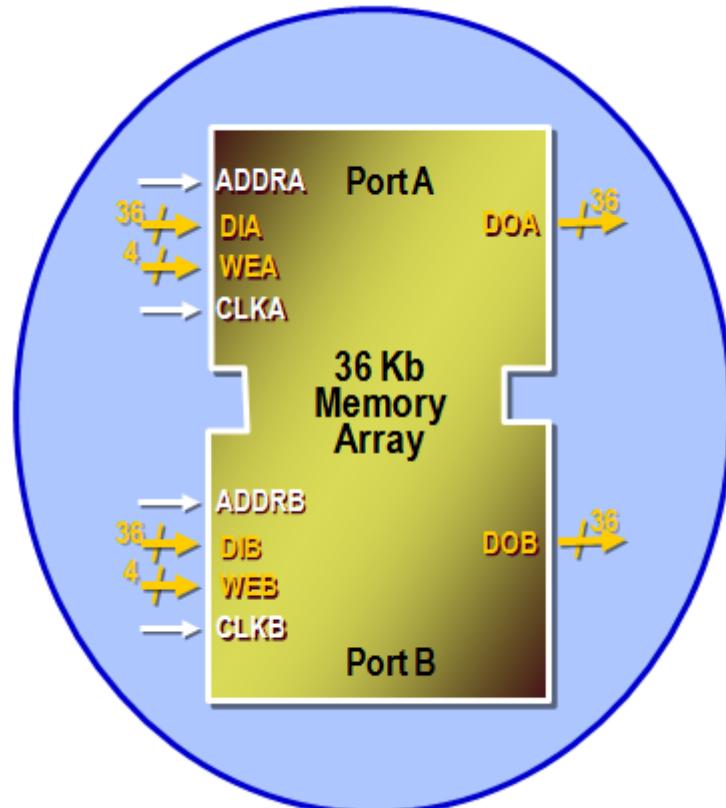


Рисунок 3.13. Блок статической памяти в ПЛИС Xilinx

Блок памяти реализует двупортовую статическую синхронную память. Термин «статическая» означает, что такая память сохраняет данные при наличии напряжения питания, и не требует специальных циклов обновления данных («регенерации»), как в динамической памяти. Термин «синхронная» отражает тот факт, что все операции с памятью происходят по фронту тактового сигнала. Альтернативой является асинхронный интерфейс, сигналы которого позволяют производить запись или чтение без привязки к специальным моментам времени. Однако синхронный интерфейс соответствует тенденциям цифровой электроники, рассмотренным в гл. 2, где синхронные схемы являются предпочтительными из-за большей предсказуемости временных характеристик.

Под термином «двупортовая» (dual-port) подразумевается наличие двух независимых наборов данных (портов), которые позволяют проводить с блоком памяти две операции одновременно. Двупортовую память также подразделяют на true dual-port («истинно двупортовую») и simple dual-port (также pseudo dual-

port, «псевдо двупортовая»). Отличием является то, что в «псевдо двупортовой памяти» только один порт является универсальным, а второй может производить только чтение данных. Для «истинно двупортовой» памяти таких ограничений нет, и оба интерфейса могут использоваться для чтения или записи в любых сочетаниях. Если производится одновременная запись в одну и ту же ячейку памяти, результат неопределен. Такие ситуации должны отслеживаться на уровне схемы или общей архитектуры системы.

В некоторых случаях вместе с блоками памяти реализуют и дополнительные схемы, облегчающие построение часто используемых цифровых модулей. На рис. 3.14 показана схема компонента FIFO, реализуемого в ряде серий FPGA.

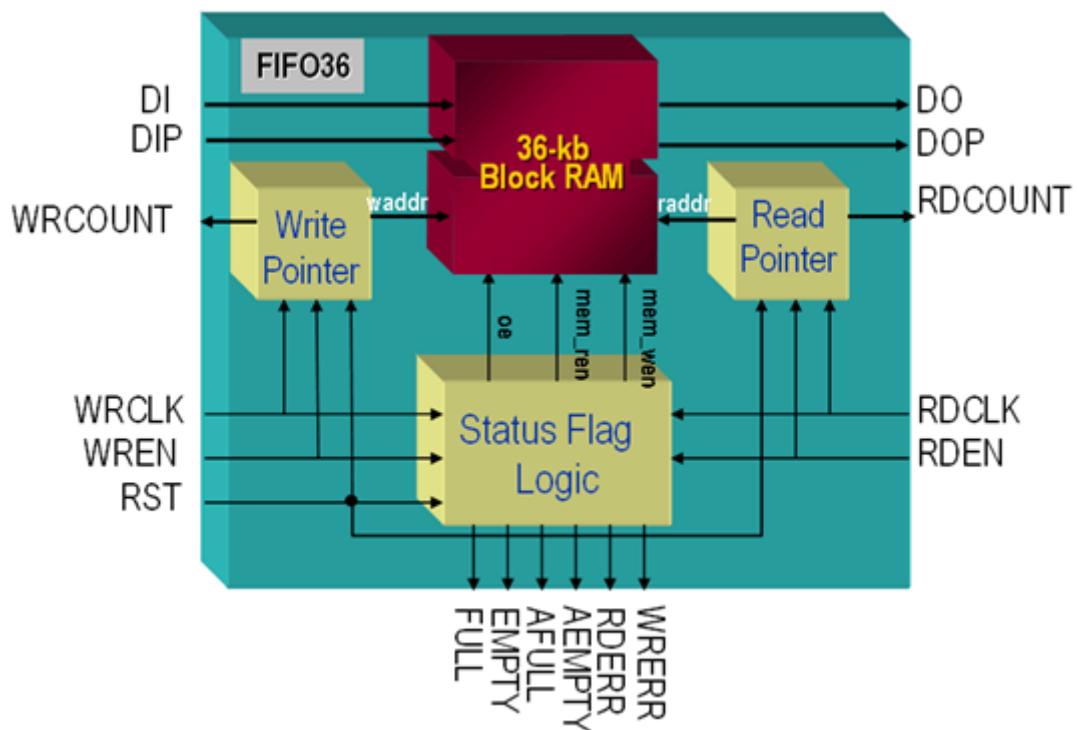


Рисунок 3.14. Компонент FIFO на базе блока статической памяти

Компонент FIFO (First In, First Out), называемый также «очередью», представляет собой массив данных, запись в который производится с одного «конца», а чтение – с другого. Это удобное представление того, что происходит с данными, но в действительности пересылка данных в процессе записи и чтения не происходит. Вместо этого при чтении и записи изменяются значения указателей на «голову» и «хвост» активных данных в FIFO. Для упрощения реализации таких схем указатели уже встроены в специализированный контроллер, который может быть использован для реализации компонента FIFO, если он требуется в проекте.

Следующий компонент относится к обязательным для установки в проект. Это генератор тактовых сигналов. Его назначением является подстройка фазы выходного тактового сигнала относительно входного (опорного). При работе микросхемы задержка распространения сигналов зависит от сочетания параметров, которое обозначается аббревиатурой PVT (Process, Voltage, Temperature). Эта аббревиатура отражает вариации технологического процесса (т.е. разброс параметров от одного кристалла к другому), колебания напряжения питания и колебания температуры. Все это приводит к изменению фазы тактового сигнала, что негативно отражается на стабильности работы цифровой схемы.

Для коррекции фазы тактового сигнала в процессе работы микросхемы используются компоненты тактовых генераторов. Это аппаратные блоки, которые сравнивают фазу входного тактового сигнала CLKIN и сигнала обратной связи CLKFBIN (от feedback – «обратная связь»), которыйложен по кристаллу микросхемы так, чтобы наиболее показательно отражать сдвиг фазы. Тактовый генератор может быть построен по одному из подходов: Delay Locked Loop (DLL), Phase Locked Loop (PLL), или может использоваться их сочетанию (блок MMCM - Mixed Mode Clock Manager).

Применение генератора тактовых сигналов в ПЛИС показано на рис. 3.15.

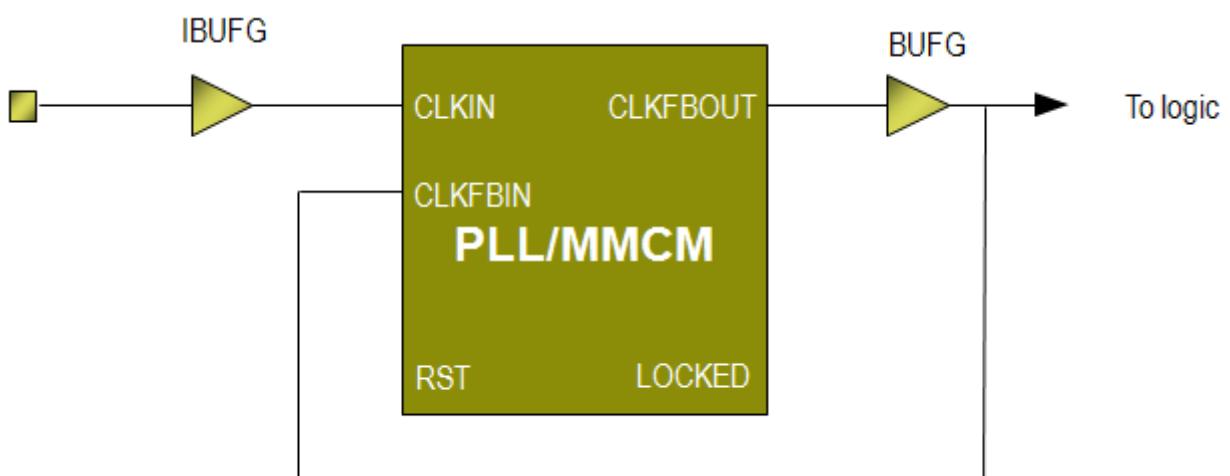


Рисунок 3.15. Применение генератора тактовых сигналов в ПЛИС

В простейшем случае выходная частота такого генератора равна входной. Может показаться, что он не обязателен в проекте, тем более что такая схема, по сути, не имеет своего RTL-описания (ее можно представить как `clk_out <= clk_in`). Однако необходимость подстраивать фазу тактового сигнала в процессе работы делает блок PLL или MMCM практически обязательным, если схему

предполагается загружать в реальную ПЛИС. Поведенческое моделирование можно выполнить и без установки в проект такого блока, потому что при этом используются идеализированные модели сигналов и компонентов, и блок тактового генератора полезной работы в модели не выполняет.

Установку тактового генератора в проект можно выполнить с помощью вспомогательных инструментов САПР. На рис. 3.16 показан генератор IP-ядра генератора тактового сигнала. Он позволяет указать значение входной тактовой частоты и настроить требуемые значения выходных частот.

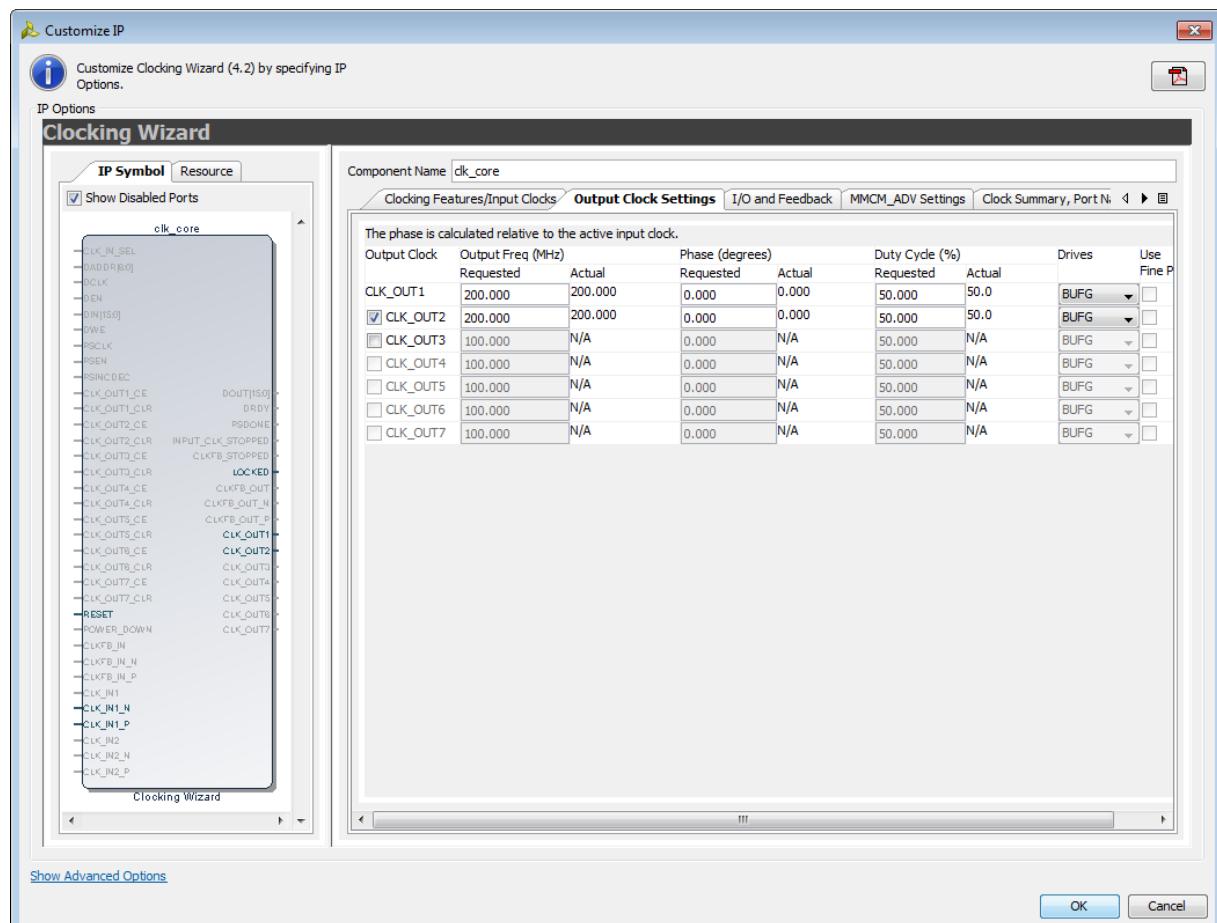


Рисунок 3.16. Генератор IP-ядра генератора тактовых сигналов в САПР ПЛИС

В современных блоках PLL и MMCM выходная частота не обязана совпадать с входной. В генераторах имеется возможность установить каждый из нескольких выходов (4-6) в индивидуальное значение частоты, определяемое формулой вида:

$$F = F_{in} * M/D$$

где M , D – целые коэффициенты (например, 1-32 или 1-64 для различных моделей генераторов).

Таким образом, можно установить в проекте частоту, равную удвоенной или половинной входной, а также, например, 31/32 от величины входной частоты.

Применение тактового генератора настоятельно рекомендуется для проектов в ПЛИС. Отказ от него приводит к непредсказуемым нарушениям работы проекта, которые могут проявляться при определенных условиях, что дополнительно затрудняет не только их исправление, но и выявление.

Еще одной разновидностью аппаратного блока является MGT (Multi-Gigabit Transceiver), или «высокоскоростной последовательный приемопередатчик». Архитектура такого блока показана на рис. 3.17.

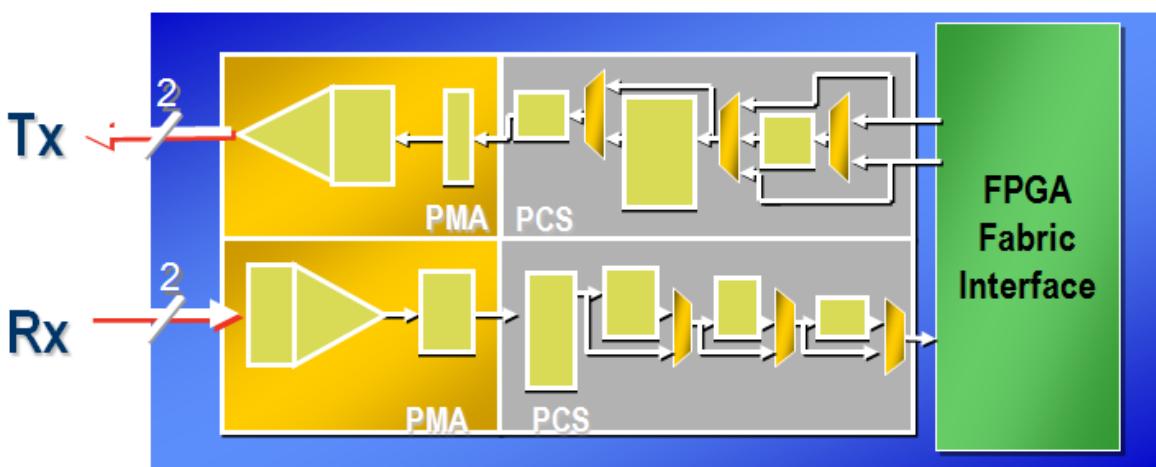


Рисунок 3.17. Архитектура блока MGT в FPGA

Блок существенно отличается от обычного цифрового вывода. Вместо единственного проводника сигналы передаются по дифференциальной паре. Это разновидность электрического интерфейса, в которой уровень 0 или 1 определяется не абсолютным значением напряжения, а разностью («дифференциалом») напряжений между двумя линиями. Такой подход позволяет существенно увеличивать частоту передаваемых данных и увеличить устойчивость к помехам, т.к. помехи действуют одновременно на два расположенных рядом проводника, а разность напряжений остается такой же. Можно представить две величины, к которым добавили одно и то же значение. Независимо от добавляемого значения разность между величинами останется такой же.

Для дифференциальных интерфейсов существует множество вариантов скорости передачи данных и форматов передаваемых пакетов. Нижняя граница находится приблизительно на уровне 600 Мбит/с, а верхняя граница зависит от типа блока MGT и может составлять 3,125, 6,5, 10, 11,3, 28, 32, 36 Гбит/с. Верхняя граница скорости передачи постоянно изменяется по мере выхода новых серий

элементной базы. Некоторые разновидности блоков MGT способны обеспечивать скорость передачи данных 116 Гбит/с.

Дифференциальная пара не может быть подключена непосредственно к цифровым сигналам внутри ПЛИС, поскольку цифровые элементы не могут работать на такой высокой тактовой частоте, чтобы обеспечить требуемый поток данных. Поэтому блок MGT использует подключение к ячейкам ПЛИС по параллельному интерфейсу и реализует аппаратно два уровня классической 7-уровневой сетевой модели OSI. Это уровни PMA (Physical Media Attachment) и PCS (Physical Code Sublayer). Они относятся к 2-му и 3-му уровням модели OSI (первый уровень – это непосредственно носитель данных, т.е. медные проводники, оптоволокно и т.д.). Таким образом, со стороны схемы в ПЛИС необходимо обеспечить поток данных (например, разрядностью 32 бита), а блок MGT сформирует требуемые сигналы для передачи по дифференциальной паре. Необходимо отметить, что такое подключение требует реализации соответствующего протокола обмена, т.е. ПЛИС не может просто передать 0 или 1 с помощью блока MGT в другую микросхему. Обмен данными с помощью MGT происходит с помощью пакетов. В самом блоке нет ограничений на формат пакета, поскольку в модели OSI это формируется на более высоких уровнях. Практическое использование MGT обычно подразумевает использование генераторов IP-ядер и библиотечных компонентов. В целом работа с этими блоками требует соответствующего опыта, поскольку даже простая демонстрация передачи пакетов требует большого объема RTL-описаний и программного кода.

Блоки MGT используются при реализации таких интерфейсов, как PCI Express, Serial ATA (SATA), 10 G Ethernet и многих других. Практическое проектирование таких систем является предметом отдельного изучения. Оно осложняется тем, что при передаче высокочастотных сигналов на их параметры влияют многие факторы, в том числе геометрические характеристики проводников на печатных платах. Для нормальной передачи сигнала с частотой 1 ГГц и выше недостаточно просто соединить контакты на печатной плате, поскольку емкость и индуктивность дорожки будет искажать форму высокочастотных сигналов. Разработка печатных плат для высокочастотных сигналов требует применения специальных САПР, и несмотря на то, что блоки MGT имеются в ПЛИС, их работоспособность во многом определяется качеством проектирования и изготовления печатной платы.

Широкое применение блоков MGT в ПЛИС обусловлено быстрым развитием проводных и беспроводных сетей. Например, магистральные

коммутаторы в оптоволоконных сетях могут быть реализованы на базе ПЛИС с десятками блоков MGT, маршрутизация пакетов между которыми обеспечивается цифровыми схемами, реализуемыми с помощью логических ячеек.

3.4. Инструменты разработки для FPGA

Разработку схемы на базе ПЛИС можно провести с помощью систем автоматизированного проектирования (САПР). Особенностью ПЛИС является закрытый производителем формат файла, описывающего конфигурацию проекта, поэтому по крайней мере завершающие стадии разработки должны выполняться с помощью программных продуктов, поставляемых производителем. Ранние стадии (ввод проекта, моделирование и синтез) доступны и для САПР других компаний.

Производители ПЛИС обычно обеспечивают САПР, поддерживающие полный цикл разработки, от ввода проекта до загрузки получаемого файла в ПЛИС. Внешний вид окна САПР Xilinx Vivado показан на рис. 3.18.

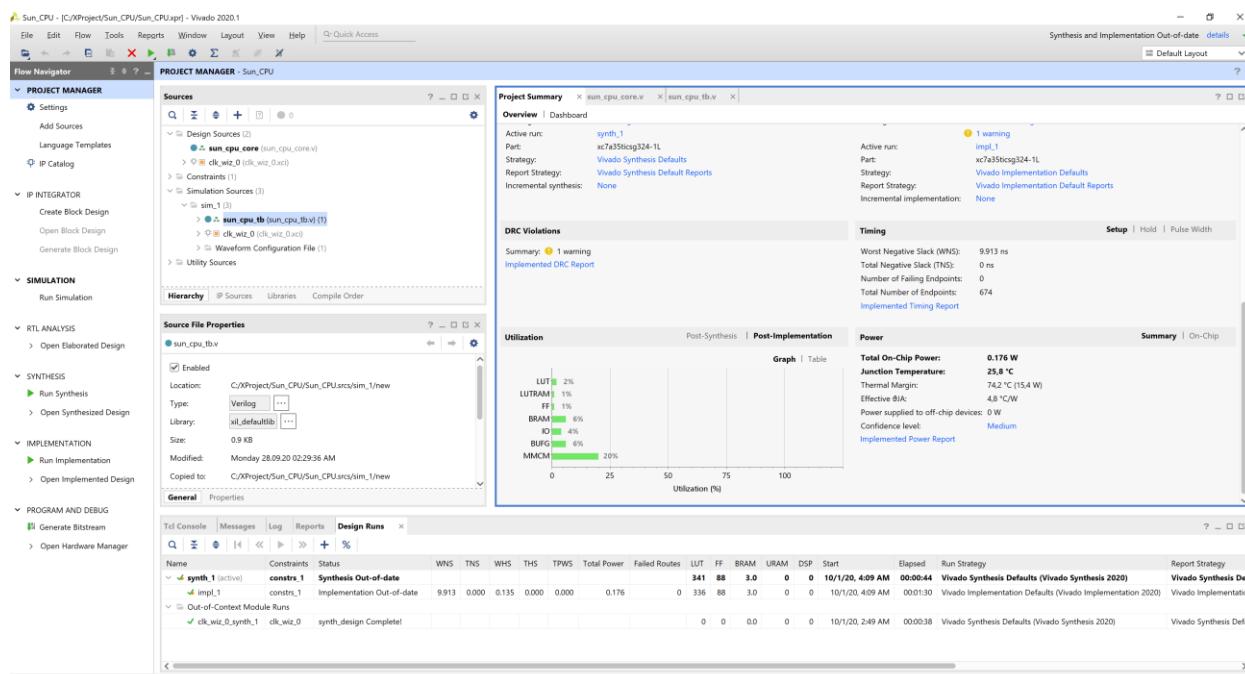


Рисунок 3.18. Внешний вид окна САПР ПЛИС Xilinx Vivado

Для быстрого освоения элементной базы производители микросхем часто выпускают «отладочные платы». Это платы, на которых кроме основной микросхемы установлены вспомогательные компоненты, интерфейс для программирования и наиболее показательные периферийные устройства. Назначением такой платы является обучение и предоставление разработчикам

образца подключения часто используемых внешних устройств. Отладочные платы могут выпускаться как самим производителем микросхем, так и другими компаниями. Внешний вид отладочной платы для ПЛИС показан на рис. 3.19. Показанный вариант компоновки является не единственным. Отладочные платы могут иметь интерфейс PCI Express и устанавливаться в настольный ПК, или наоборот, размещаться в небольших габаритах.

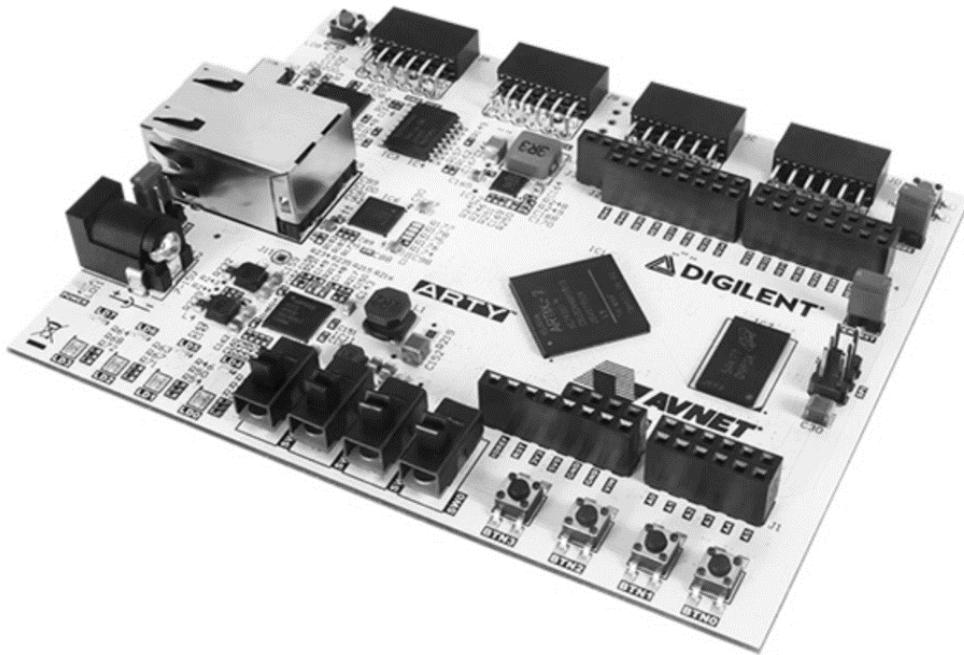


Рисунок 3.19. Пример отладочной платы на базе ПЛИС

Сочетание компьютера с установленной САПР ПЛИС и отладочной платы позволяет создать цифровую схему и загрузить макет устройства в ПЛИС без привлечения дополнительного оборудования и инструментов. Упрощенный маршрут проектирования для ПЛИС показан на рис. 3.20.

В качестве минимального набора входных данных требуется файл верхнего уровня (top level module), описывающий схему проекта, а также файл проектных ограничений (constraints).

Первый этап, синтез (synthesize), состоит в построении схемы соединений на основе исходного файла на языке описания аппаратуры. Полученный список связей (netlist) является достаточно абстрактным и не привязывает элементы схемы к конкретным компонентам ПЛИС. Кроме того, внешние выводы проекта следует привязать к выводам конкретной микросхемы, которая будет использоваться.

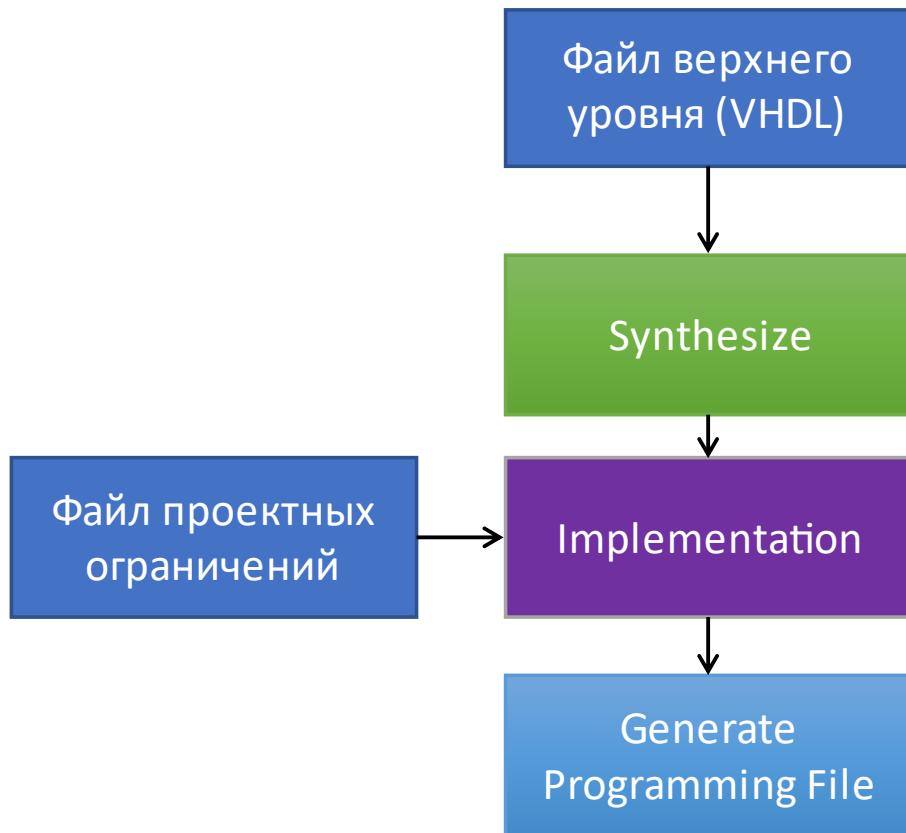


Рисунок 3.20. Упрощенный маршрут проектирования для ПЛИС

Эту информацию невозможно привести в тексте на языке описания аппаратуры, поэтому в САПР добавляется специальный файл проектных ограничений, где указано, например, что сигнал «a_in» должен быть связан с выводом ПЛИС «A1». Поскольку каждая модель ПЛИС имеет собственные варианты исполнения, проектные ограничения являются специфическими для каждой ПЛИС, отладочной платы и проекта в целом.

Имея список связей и сопровождающий его перечень проектных ограничений, САПР может приступить ко второму крупному этапу – implementation. В этот этап включены шаги по определению, какой элемент ПЛИС будет выполнять конкретные действия и как будут проведены соединительные линии между ними. Наиболее длительное действие здесь – «размещение и трассировка» (Place and Route). Для сложных проектов может быть потрачено очень большое время (часы и даже дни) на получение приемлемого результата, поскольку при большом объеме ПЛИС существует очень много вариантов размещения элементов схемы в отдельных ячейках и проведения связей между ними.

Часто в проектные ограничения включают также требования к времени распространения сигналов. В этом случае установка компонентов в

противоположные углы ПЛИС имеет существенный риск превысить допустимые времена задержек. Очевидно, что для достижения высокой тактовой частоты ячейки ПЛИС, которые должны быть соединены, следует поставить рядом. Однако в сложных проектах часто появляются разветвленные линии связей между множеством ячеек, поэтому добиться компактной расстановки схемы часто бывает невозможно. Именно поэтому этап *implementation* занимает длительное время, а повышение тактовой частоты проекта в ПЛИС требует определенной квалификации разработчика.

На рис. 3.21 показан внешний вид внутреннего инструмента САПР ПЛИС – редактора топологии. Этот редактор не является необходимым при разработке проекта, однако позволяет показать структуру ПЛИС и режимы работы ячеек.

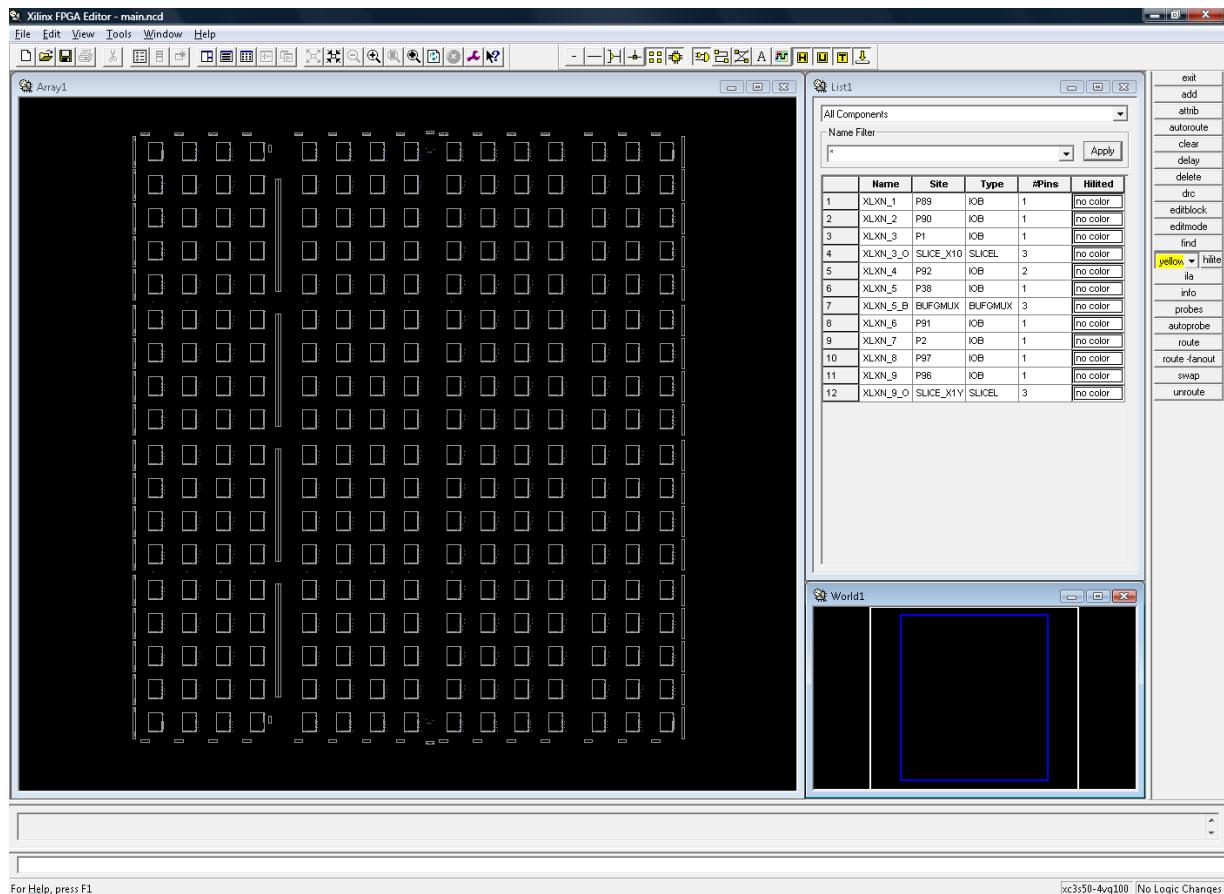


Рисунок 3.21. Внешний вид ПЛИС в редакторе топологии

На рис. 3.22 показан фрагмент редактора топологии после выполнения трассировки проекта. Видны добавленные трассировочные линии, показанные голубым цветом.

Можно отметить, что показанная ПЛИС относится к устаревшему семейству Spartan-3 и имеет минимальный логический объем. Даже в этом случае микросхема содержит несколько сотен логических ячеек и проектирование путем их ручного

соединения – крайне непродуктивный путь. Изучение соединений является вспомогательным инструментом и обычно применяется на уровне крупных блоков проекта для оптимизации их взаимной расстановки.

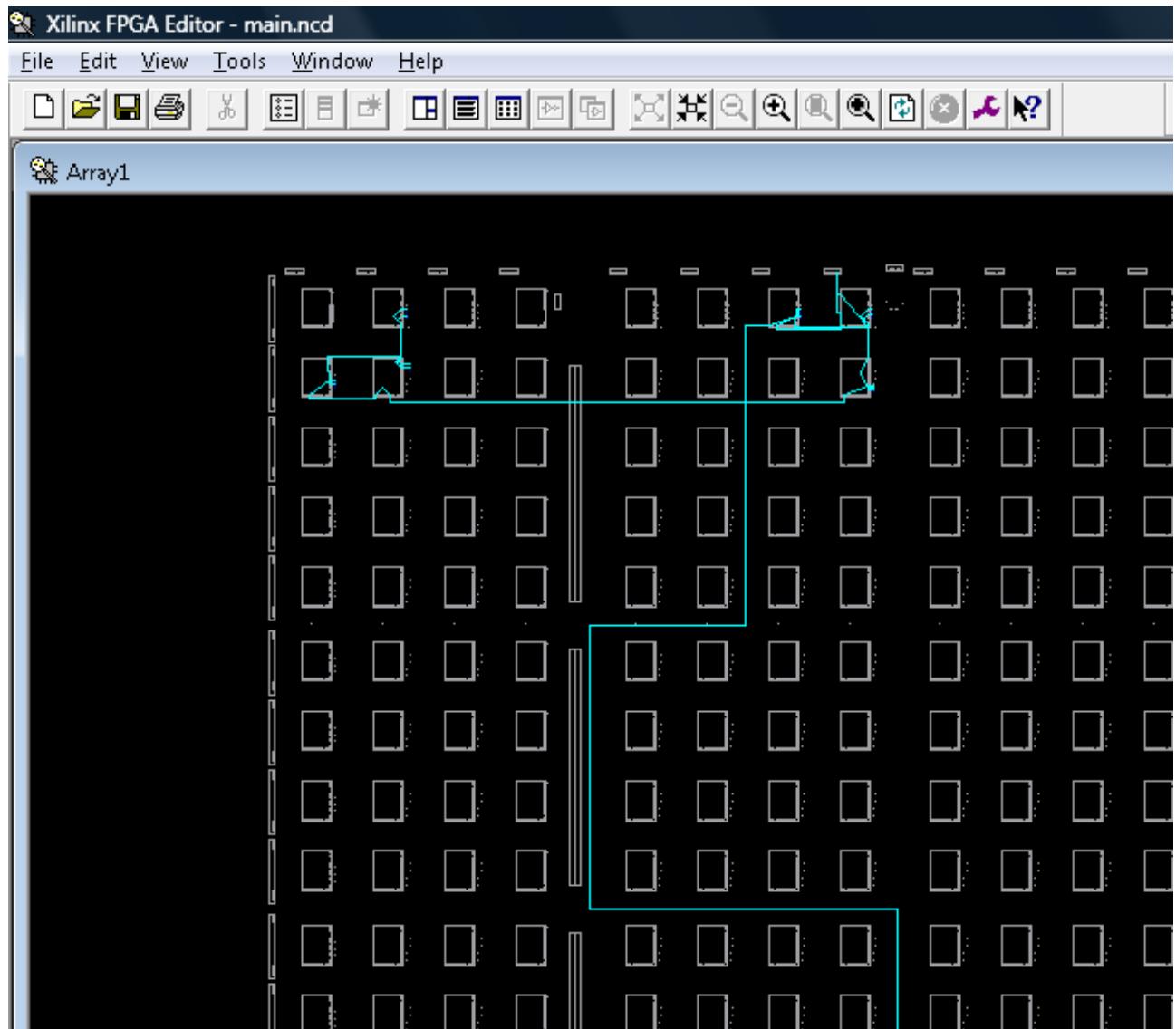


Рисунок 3.22. Внешний вид ПЛИС в редакторе топологии с выполненной трассировкой проекта

Средства загрузки конфигурации ПЛИС обычно являются частью САПР. Большинство отладочных плат уже содержат схемы для загрузки конфигурации, подключаемые к порту USB компьютера, однако сами по себе ПЛИС не имеют такой возможности. Поэтому при выборе отладочных плат для проекта необходимо проверять, требуется ли для работы с ними отдельный внешний программатор.

3.5. Выводы по главе

Программируемые логические интегральные схемы (ПЛИС) – удобный и практически безальтернативный способ получения работоспособного макета цифровой микросхемы. Поскольку ПЛИС хранит конфигурацию в статической энергозависимой памяти, она имеет неограниченный ресурс перезагрузки конфигурации и может быть использована для создания макетов схожим образом с обычной компиляцией программ. САПР ПЛИС поддерживают сквозной маршрут проектирования – от ввода схемы до загрузки полученной конфигурации в микросхему.

В составе ПЛИС с архитектурой FPGA находятся конфигурируемые логические ячейки, способные при соединении представить практически любой цифровой узел. Кроме ячеек, в FPGA добавляют аппаратные компоненты, которые не конфигурируются, однако реализуют часто востребованные в практических проектах схемы. К таким компонентам относятся блоки статической памяти, блоки «умножение с накоплением» и блоки MGT (высокоскоростных последовательных приемопередатчиков). На базе этих компонентов часто создают серийные изделия, в которых ПЛИС используются не как макеты будущих микросхем, а как элемент конечного продукта, который не планируется для замены. Это становится возможным потому, что десятки (а в больших ПЛИС сотни и тысячи) аппаратных компонентов обеспечивают высокую производительность ПЛИС в задачах цифровой обработки сигналов и сетевом оборудовании.

При разработке схем на базе ПЛИС необходимо следовать тем же рекомендациям, которые характерны для обычных цифровых микросхем. Например, синхронные схемы следует использовать и в проектах ПЛИС. При превышении определенного размера проекта (обычно сотни тысяч логических ячеек) следует использовать подход GALS. Проблема «темного кремния» для ПЛИС пока не стоит остро, поскольку ячейки ПЛИС сами по себе являются достаточно «разреженными» и ресурсы кристалла не используются на 100%. Поэтому производители обычно могут обеспечить допустимый уровень тепловыделения и ПЛИС могут охлаждаться распространенными несложными системами охлаждения.

Контрольные вопросы:

1. Что такое ПЛИС? Что является отличительной чертой архитектуры FPGA?

2. Какие основные компоненты присутствуют в ПЛИС с архитектурой FPGA?
3. Как изменяется номенклатура и удельный вес аппаратных компонентов в составе ПЛИС FPGA?
4. Какие основные компоненты имеет логическая ячейка FPGA?

4. РЕАЛИЗАЦИЯ БАЗОВЫХ УСТРОЙСТВ ЦИФРОВОЙ СХЕМОТЕХНИКИ

4.1. Основные цифровые элементы

В цифровой схемотехнике выделяют асинхронные (также называемые комбинационными) и синхронные элементы. Комбинационные элементы отличаются тем, что состояние их выхода однозначно определяется комбинацией значений на входах. Основными разновидностями комбинационных логических элементов являются элементы НЕ, И, ИЛИ. Их графическое изображение показано на рис. 4.1. Верхний ряд соответствует изображению по стандарту IEEE, нижний – по ЕСКД.

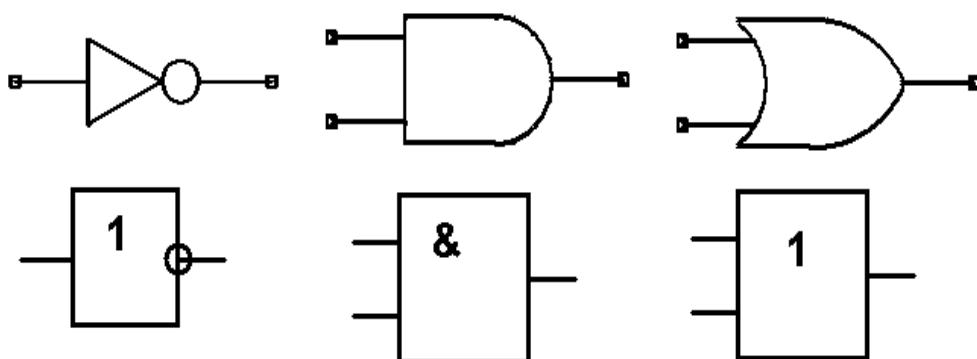


Рисунок 4.1. Базовые логические элементы

К дополнительным логическим элементам относят ИСКЛЮЧАЮЩЕЕ ИЛИ, И-НЕ, ИЛИ-НЕ. Они показаны на рис. 4.2. Аналогично предыдущему рисунку, верхний ряд соответствует изображению по стандарту IEEE, нижний – по ЕСКД.

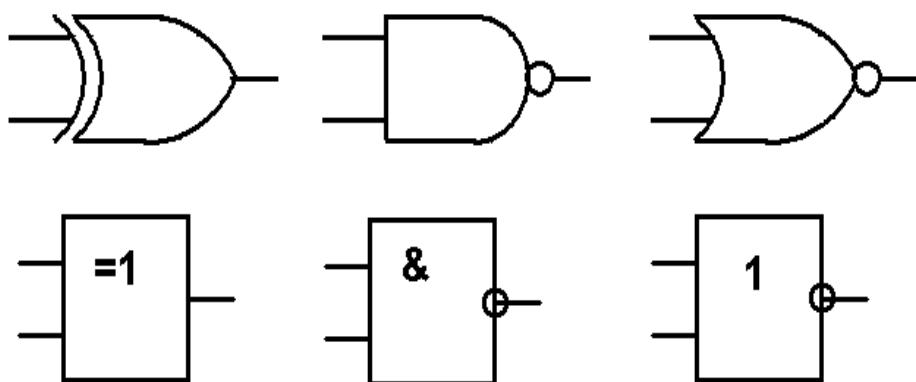


Рисунок 4.2. Дополнительные логические элементы

Удобным способом описания работы логических вентилей является составление таблиц истинности. В такую таблицу записывается состояние выхода в зависимости от комбинации входных сигналов. Входы обычно обозначаются начальными буквами латинского алфавита, т.е., A, B, C, D, а для выхода используют обозначение Q. Для элементов, показанных на рис. 4.1, таблицы истинности будут выглядеть следующим образом.

Таблица 4.1. Таблицы истинности для базовых логических элементов

Элемент НЕ	
A	Q
0	1
1	0

Элемент И		
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Элемент ИЛИ		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

Таблицы истинности для элементов, показанных на рис. 4.2, приведены в табл. 4.2.

Таблица 4.2. Таблицы истинности для вспомогательных логических элементов

Элемент ИСКЛЮЧАЮЩЕЕ ИЛИ		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Элемент И-НЕ		
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

Элемент ИЛИ-НЕ		
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

4.2. Реализация логических выражений в ПЛИС

Особенностью ПЛИС является отсутствие показанных выше элементов в явном виде. Вместо этого в логических ячейках имеются генераторы комбинационных выражений (LUT, Look-Up Table), которые непосредственно хранят таблицу истинности.

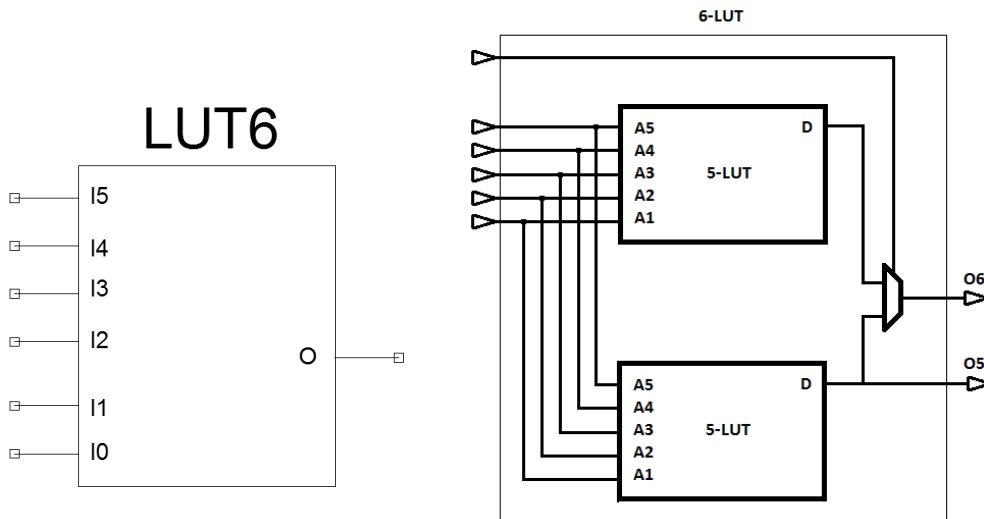


Рисунок 4.3. Таблица истинности в составе ПЛИС

Реализация комбинационных выражений в цифровой электронике производится с помощью оператора assign (в языке Verilog). Он также называется *оператором непрерывного присваивания* (continuous assignment operator). Пример выражения для элемента И показан ниже:

```
assign c = a & b;
```

В отличие от языков программирования, где знак «равно» воспринимается как «присвоить переменной значение и перейти к следующему оператору», в языках описания аппаратуры такие операторы удобнее интерпретировать как «соединить». После такого соединения назначить другое выражение сигналу невозможно, поскольку это означало бы короткое замыкание между двумя проводниками.

Это различие очень важно для понимания языков описания аппаратуры. Программистам привычнее воспринимать операторы как однократные действия, после которых можно совершить другие действия с этой переменной. Попытки же воспринимать тексты на Verilog в подобном ключе приведут к неустранимым ошибкам синтезатора, который не сможет создать схему, в которой один и тот же сигнал будет описываться разными правилами.

Для оператора assign необходимо искать такие выражения, которые будут справедливы на протяжении всего времени работы схемы. Последовательная смена состояний, аналогичная выполнению программы, реализуется с помощью тактируемых схем – например, конечных автоматов.

Ниже показан более сложный пример, использующий основные логические операторы, имеющиеся в Verilog. Из примера видно, что выражения в Verilog записываются в целом по тем же правилам, что и в языках программирования. Обозначения операторов аналогичны языку Си (используются символы $\&$ $|$ \wedge \sim , а не буквенные операторы and or xor not).

```
assign q = (~a | b) ^ (c & d);
```

Реализация логических функций в виде таблицы истинности предполагает анализ не сложности выражений, а количества входов в этом выражении.

Рассмотрим выражение на Verilog:

```
assign q = (a & b) | ((c ^ d) & (e | f));
```

Оно выглядит громоздким, однако использует 6 входных аргументов. Поэтому его реализация потребует всего одной LUT, как показано на рис. 4.4.

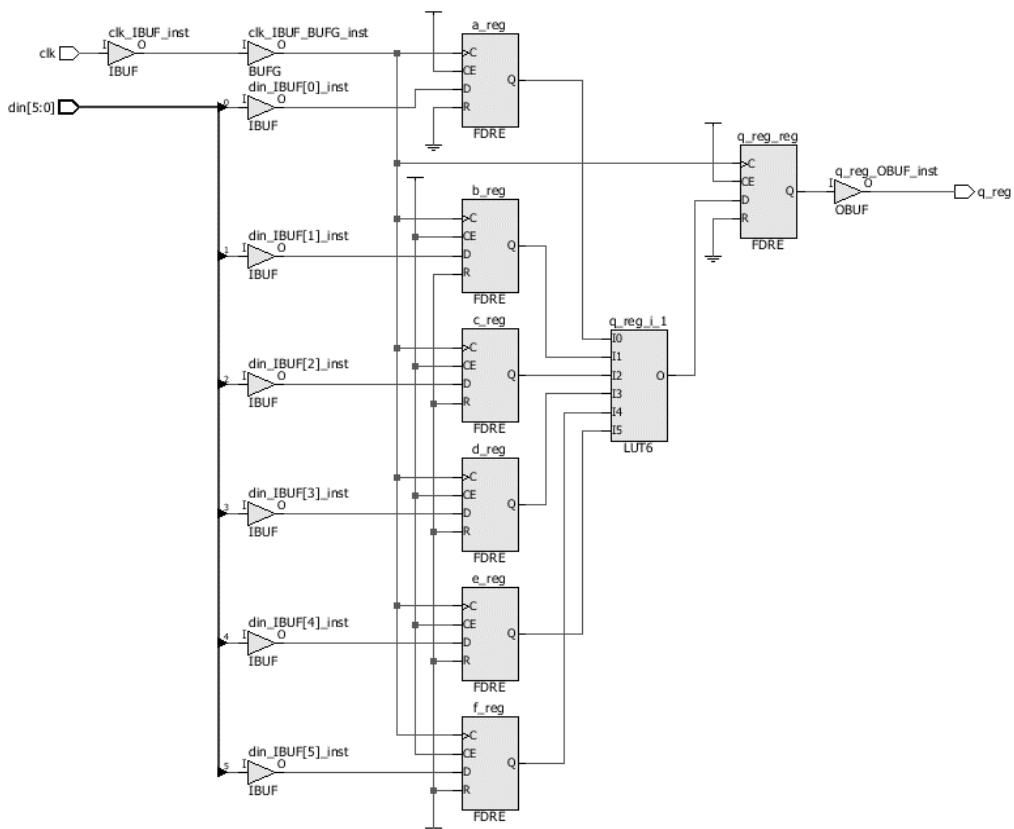


Рисунок 4.4. Реализация комбинационного выражения

На нем видно, что входные и выходной сигналы, которые хранятся в триггерах FDRE, подключены к единственной LUT. Вне зависимости от количества операторов в выражении, 6 входов гарантируют, что любая таблица истинности поместится в 6-входовой LUT.

Для сравнения рассмотрим выражение, которое выглядит более простым:

$$q \leq '1' \text{ when } a = b \text{ else '0';}$$

Тем не менее, сигналы a, b объявлены в модуле как 32-разрядные. Поэтому количество входов в логическом выражении равно 64, что очевидно не помещается в одну LUT. Результаты синтеза такого выражения для ПЛИС показаны на рис. 4.5.

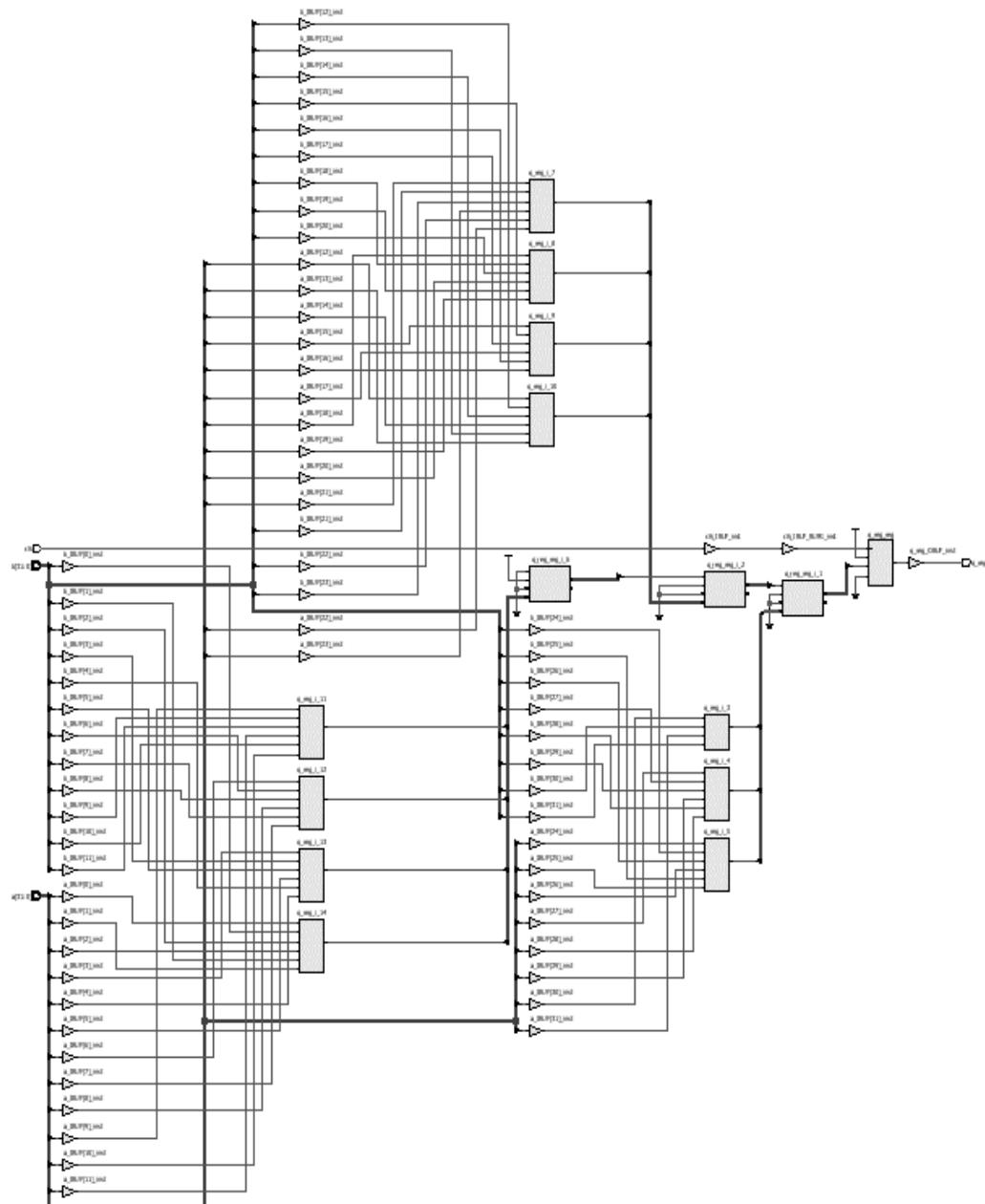
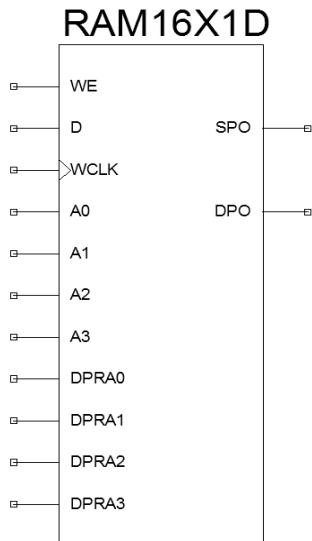


Рисунок 4.5. Реализация устройства сравнения двух operandов

4.3. Дополнительные возможности логических ячеек ПЛИС

Поскольку логический генератор представляет собой элемент статической памяти, его можно использовать и в этом качестве. Этот режим поддерживается не всеми производителями ПЛИС, например, Xilinx обеспечивает работу LUT в режимах распределенной памяти (*distributed memory*) и сдвиговых регистров.



*Рисунок 4.6. Таблица истинности в режиме распределенной памяти (*distributed memory*)*

Распределенная память может работать в однопортовом или простом двупортовом (*simple dual-port*) режимах. Простой двупортовый режим имеет то ограничение, что только один порт может использоваться для чтения и записи, а второй предназначен только для чтения. Показанный на рис. 4.6 элемент распределенной памяти имеет следующие сигналы:

- we – сигнал разрешения записи;
- d – данные для записи;
- wclk – вход тактового сигнала;
- a – адрес для записи;
- dpra – адрес для чтения, второй порт (*dual port read address*);
- spo – выход первого порта (*single port output*);
- dpo – выход второго порта (*dual port output*).

Временные диаграммы работы распределенной памяти показаны на рис. 4.7.

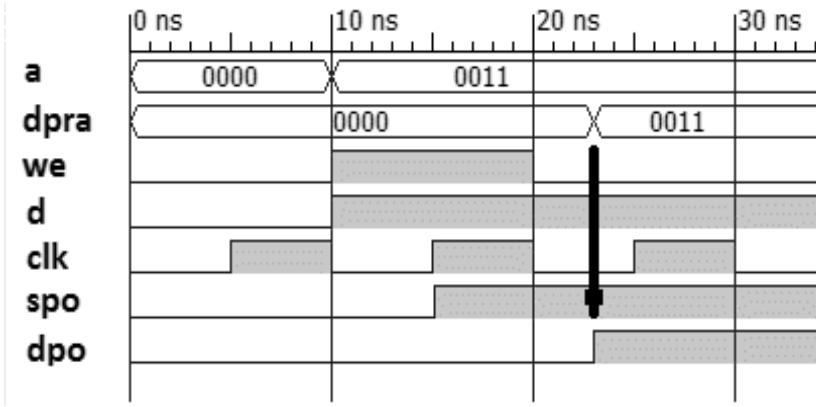


Рисунок 4.7. Временные диаграммы работы распределенной памяти

На рис. 4.7 видно, что запись в память происходит синхронно (в момент времени 15 нс, когда данные $d = 1$ записываются по адресу $addr = 00112 = 310$), а чтение – асинхронно. В момент времени 23 нс изменение адреса на входе dpra приводит к соответствующему изменению выхода dpo. Этот момент (без фронта тактового сигнала) был выбран специально, чтобы продемонстрировать, что память реагирует на изменение адреса асинхронно. Аналогично, изменение состояния входа a немедленно приводит к появлению на выходе spo значения из ячейки с адресом a.

Распределенная память удобна для организации небольших блоков данных – буферов, линий задержки, небольших таблиц. Реализация больших блоков на распределенной памяти в общем случае нецелесообразна из-за сильной фрагментации такого блока.

Другой вариант использования логического генератора – реализация на его базе сдвигового регистра. Графическое изображение такого компонента показано на рис. 4.8.

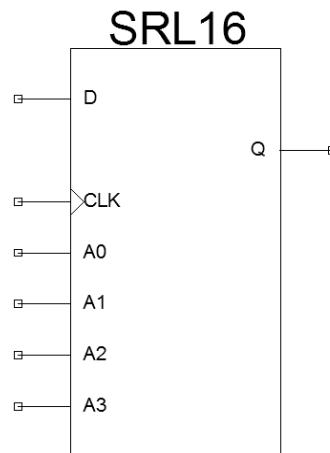


Рисунок 4.8. Таблица истинности в режиме сдвигового регистра

Сдвиговый регистр имеет следующие сигналы:

- d – данные для записи в сдвиговый регистр;
- clk – тактовый сигнал;
- a – адресный вход;
- q – выход данных.

В простейшем варианте сдвиговый регистр может представлять собой модуль, задерживающий входной сигнал din на N тактов. Временные диаграммы работы 8-разрядного сдвигового регистра показаны на рис. 4.9.



Рисунок 4.9. Временные диаграммы работы сдвигового регистра

В составе матрицы ресурсов FPGA есть не только LUT и триггеры. Для реализации некоторых часто используемых цифровых узлов возможности LUT избыточны. Поэтому к группе LUT и триггеров добавляются дополнительные компоненты, выполняющие единственную функцию, которая часто оказывается полезной в проектах. Набор LUT, триггеров и дополнительных компонентов называется в FPGA *секцией* (slice). Этот термин используется Xilinx и не всегда применяется другими производителями в том же качестве.

На рис. 4.10 показаны основные компоненты логической секции ПЛИС Xilinx.

К дополнительным компонентам можно в первую очередь отнести мультиплексоры и цепи ускоренного переноса. В левой части рис. 4.10 видно, что выходы LUT объединяются мультиплексорами F7MUX, а выходы этих мультиплексоров – дополнительно мультиплексором F8MUX.

Цифры 7 и 8 в обозначении мультиплексоров показывают, что они помогают расширить количество входов с 6 в LUT до 7 и 8 соответственно. На рис. 4.11 показано, как синтезатор использует эти мультиплексоры для реализации 16-входового мультиплексора. Для 16 входов требуется 4 разряда управляющего сигнала, поэтому общее количество входов становится равным 20. Это потребует 4 логических генератора, выходы которых нужно чем-то

объединить. Мультиплексоры F7MUX и F8MUX позволяют завершить построение схемы в рамках одной секции, не привлекая еще одну LUT.

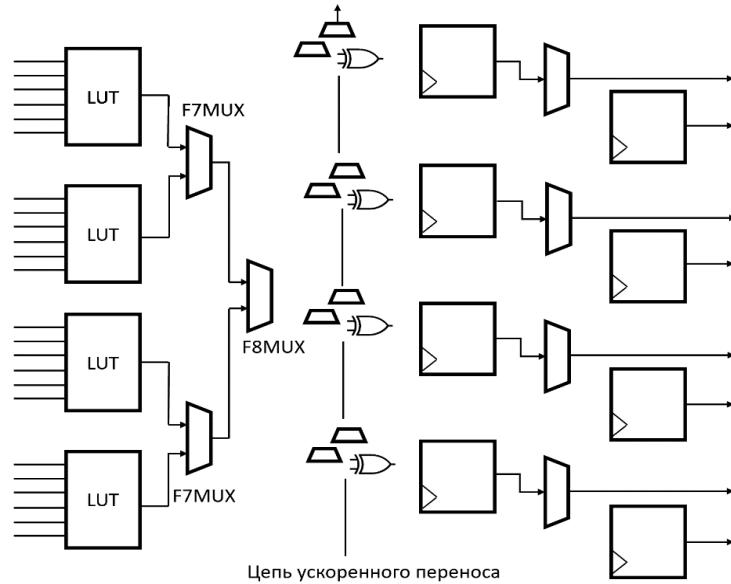


Рисунок 4.10. Основные компоненты логической секции ПЛИС

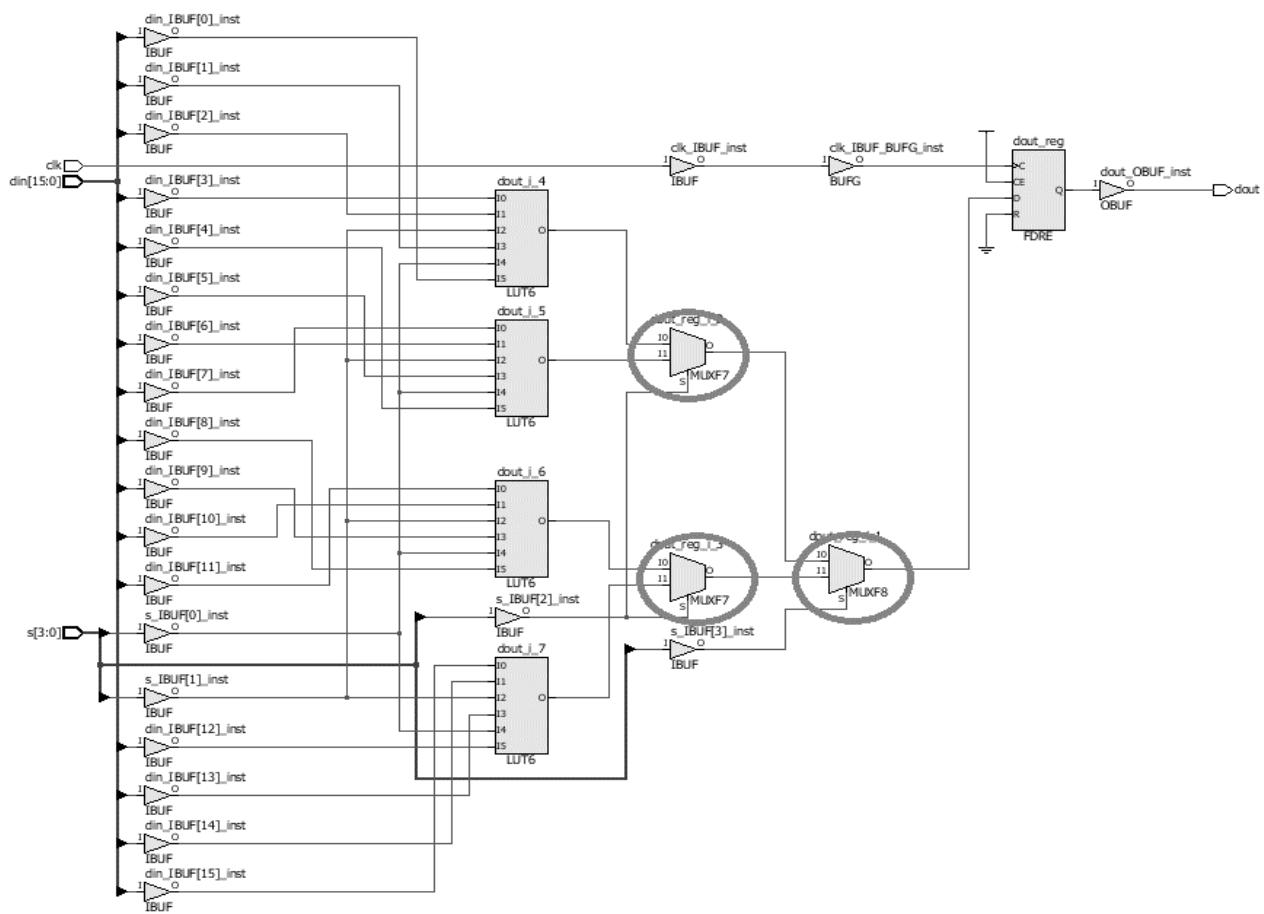


Рисунок 4.11. Применение дополнительных компонентов логической секции (мультиплексоров) для реализации более сложного комбинационного выражения

Синтезаторы применяют дополнительные мультиплексоры по мере возможности. Следует использовать оператор switch по мере возможности, чтобы облегчить распознавание шаблона мультиплексора. Пример:

```
always @ *
begin
  case (sel)
    2'b00 : q = a;
    2'b01 : q = b;
    2'b10 : q = c;
    2'b11 : q = d;
    default : q = 1'bx;
  endcase
end
```

В этом примере сигнал sel, очевидно, является сигналом выбора входа мультиплексора. Подобные шаблоны легко распознаются синтезаторами.

На рис. 4.10 также показана цепь ускоренного переноса. Она используется для реализации схем сложения и вычитания, которые будут рассмотрены далее. Как и мультиплексоры, эта цепь занимает немного места в секции, но помогает создавать схемы, для которых в противном случае были бы применены LUT.

4.4. Выводы по главе

Базовые логические операции реализуются в языках описания аппаратуры с помощью оператора непрерывного присваивания. В языке Verilog это оператор assign. В отличие от языков программирования, присваивание должно быть единственным в модуле для каждого сигнала. Удобно рассматривать такой оператор как описание соединений, которые будут оставаться в схеме на всем протяжении ее работы.

Особенностью ПЛИС является применение таблиц истинности для реализации логических выражений. Поэтому при проектировании нужно следить не за сложностью выражений, а за количеством его входов.

Ячейки ПЛИС имеют дополнительные возможности, которые автоматически используются синтезаторами. Такие схемы можно создать и на базе только логических генераторов в таблицах истинности, но дополнительные компоненты делают их компактнее и быстрее. К дополнительным функциям относятся:

- распределенная память;
- сдвиговые регистры;

- аппаратные мультиплексоры на выходах логических генераторов;
- цепи ускоренного переноса.

Контрольные вопросы:

1. Какие виды логических элементов существуют в цифровой схемотехнике?
2. Что такое таблица истинности и как она используется в ПЛИС?
3. Влияет ли сложность выражения на количество таблиц истинности, необходимых для его реализации?
4. Какие дополнительные компоненты имеются в секции ПЛИС? Какие функции они помогают реализовать?
5. Почему с помощью оператора assign нельзя реализовать двоичный счетчик?

5. РЕАЛИЗАЦИЯ ОСНОВНЫХ АРИФМЕТИЧЕСКИХ ФУНКЦИЙ

5.1. Содержание раздела

В вычислительной технике активно используются арифметические выражения. Часто именно арифметические операции ассоциируются с понятием «обработка данных». Для программистов вполне естественно ожидать от процессорной системы поддержки привычного набора действий, известных из математики.

В то же время реализация в цифровой электронике даже известных из начальной школы операций имеет различную сложность. Если рассматривать четыре действия арифметики, то сложение и вычитание реализуются достаточно просто, для умножения требуется уже более сложная схема, а деление вызывает существенные проблемы и обычно не выполняется за один такт.

Для более сложных операций, например, трансцендентных функций, существует множество вариантов реализации. В зависимости от требований по быстродействию, точности и размеру, могут быть применены разные подходы к реализации таких функций.

5.2. Сумматор и его логическая функция

С операциями сложения и вычитания обычно связывают объяснение понятия «дополнительная двоичная арифметика». Термин «дополнительная» может быть несколько дезориентирующим, поскольку можно посчитать, что существует какая-то «основная» арифметика. В действительности исходным термином является *complementary*, т.е. перевод ближе к «дополняющая».

Для понимания термина можно рассмотреть пример, показанный на рис. 5.1. В этом примере к числу 255, которое в двоичной системе представляется как 11111111, прибавляется 1. Видно, что получаемый результат 256 должен быть записан уже в 9 битах. Если выход такого сумматора не имеет 9-го бита, в результате получится 0.

Такой результат можно интерпретировать немного иначе. Если принять, что было вычислено $x + 1 = 0$, то x очевидно равен «-1». Формально знак минус нигде не хранится и не представлен в виде какого-то особого электрического сигнала, однако это удобный способ объяснить, почему при вычислениях получился 0.

$$\begin{array}{r}
 + 11111111 \\
 00000001 \\
 \hline
 100000000
 \end{array}$$

Рисунок 5.1. Иллюстрация к реализации сложения чисел в дополнительной двоичной арифметике

Иными словами, в дополнительной двоичной арифметике отрицательные числа представляются в виде обычных двоичных чисел, при сложении которых с их модулями в результате получится 0. Например, число, в котором все разряды установлены в 1, при любой разрядности равно «-1», поскольку при прибавлении 1 результат будет равен 0 из-за переполнения разрядной сетки.

С точки зрения логики, нет какого-то принципиального ограничения, что считать положительными, а что отрицательными числами. Например, 11111111 можно рассматривать как -1 или 255. Аналогично, 11111110 – это -2 или 254. Эти числа хранятся в памяти совершенно одинаково, и их интерпретация зависит уже от программиста.

В языках программирования в основном используется именно дополнительная двоичная арифметика. Например, в спецификации языка может быть указан диапазон для 8-разрядных переменных -128 .. 127. Это указывает на применение дополнительного двоичного представления. В то же время модификатор `unsigned` («беззнаковое») показывает, что число следует трактовать как 0 .. 255, хотя его двоичное представление никак не меняется. Как правило, граница между положительными и отрицательными числами установлена так, что появление 1 в старшем разряде числа означает, что это отрицательное число.

При выполнении последовательностей сложений и вычитаний над числами в дополнительном двоичном формате конечный результат будет правильным (если только он не выйдет за пределы разрядной сетки).

Модификатор `signed` в цифровой электронике имеет немного другое действие, чем в программировании. Например, объявление `signed int x` может означать, что если в переменной `x` все биты равны 1, то следует напечатать -1. В цифровой электронике понятие «число со знаком» (`signed`) означает, что число хранится в формате «знак, значение». Обычно бит знака является старшим. Таким образом, 8-разрядное число -1 в знаковом формате запишется так:

1 0000001

Здесь старший (подчеркнутый) бит показывает, что число отрицательное, а остальные разряды показывают его абсолютное значение.

Формат со знаком сложнее для сложения и вычитания, поскольку для определения операции, которую в действительности нужно выполнить, следует проанализировать сочетание знаков обоих операндов. Однако для умножения удобнее именно формат со знаком.

5.3. Особенности описания сумматора

Сложение однобитных двоичных чисел происходит следующим образом. Возможные варианты представлены ниже:

$$\begin{aligned}0 + 0 &= 00 \\0 + 1 &= 01 \\1 + 0 &= 01 \\1 + 1 &= 10 \text{ (1 переносится в следующий разряд)}\end{aligned}$$

Видно, что младший разряд определяется с помощью функции xor, а перенос в следующий разряд – с помощью функции and. Можно было бы реализовать однобитный сумматор на базе отдельных вентилей или LUT в ПЛИС.

Практические рекомендации для сложения и вычитания в современной ситуации вполне однозначны – вместо базовых логических элементов следует использовать операторы сложения и вычитания. Следующий пример показывает реализацию сумматора на языке Verilog.

```
module sum(
    input [31:0] a,
    input [31:0] b,
    output [31:0] c
);

    assign c = a + b;

endmodule
```

Может возникнуть вопрос, почему стоит полагаться на синтезатор, который оперирует высокоуровневым оператором сложения. Не будет ли сумматор, описанный отдельными вентилями, эффективнее?

В действительности в ПЛИС такое решение снизит эффективность. Если сконфигурировать LUT для формирования бита результата и бита переноса, ресурсы логической ячейки будут использованы крайне неэффективно. В главе, посвященной архитектуре ПЛИС, уже рассматривались дополнительные ресурсы логической ячейки, среди которых была и цепь ускоренного переноса (fast carry chain). Это аппаратный узел сумматора, который занимает немного места по сравнению с остальными элементами ячейки, однако целиком реализует сложение однобитных чисел и формирует бит переноса. Таким образом, несколько узлов сумматора могут объединяться, реализуя сложение или вычитание многоразрядных чисел.

Для схемы в СБИС существует множество вариантов реализации многоразрядного сумматора. Рассмотренный пример, когда при сложении получается бит переноса, который используется следующим разрядом, является самым простым и называется ripple carry. В схемотехнике существует множество подходов, различающихся быстродействием, энергопотреблением и размером. Конкретная реализация является предметом отдельного исследования.

Исходя из соображений разделения уровней абстрагирования, при проектировании СБИС на уровне схемотехники целесообразно выделить компонент «сумматор», не вдаваясь в детали его реализации. Первая реализация сумматора может быть описана обычным выражением `assign c = a + b`. Если получаемый при синтезе результат не устраивает разработчика, можно рассматривать альтернативные варианты. Тем не менее, сложно сказать заранее, какую именно схему следует использовать, поскольку это зависит и от общих требований к проекту («повысить частоту?», «снизить энергопотребление?», «уменьшить площадь?»), и от окружающих сумматор компонентов, и от возможностей используемой топологической библиотеки. Выделение сумматора в отдельный модуль позволяет использовать его в модулях более высокого уровня, а при необходимости выполнить оптимизацию, не изменяя другие компоненты проекта.

Следует обращать внимание, что некоторые схемы сумматора предусматривают многотактную работу. Такие схемы не могут быть автоматически установлены вместо сумматора, описываемого оператором `assign`, поскольку в этом случае подразумевается, что результат получается комбинационно (т.е. после подачи операндов результат появляется на выходе

просто с некоторой задержкой, без необходимости подавать такты синхронизации).

5.4. Цепи ускоренного переноса

На рис. 5.2 показано расположение цепей ускоренного переноса в секции ПЛИС Xilinx.

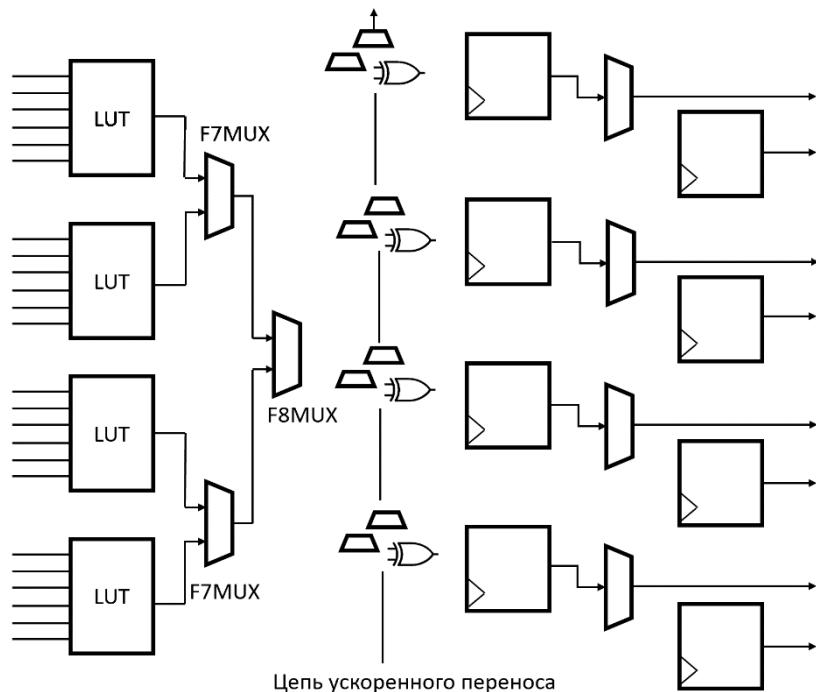


Рисунок 5.2. Иллюстрация к реализации сложения чисел в дополнительной двоичной арифметике

Показанные цепи будут автоматически применены по мере возможности. Именно поэтому рекомендацией по реализации сумматора является использование обычного оператора $+$, без попыток перейти к описанию сумматора с помощью логических функций.

На рис. 5.3 показана реализация сумматора в ПЛИС. Это представление схемы после синтеза, где можно посмотреть, какие конкретно компоненты предполагаются для реализации описания на Verilog. Видно, что синтезатором применен компонент CARRY4 (это не единственный компонент такого типа, поскольку описан 32-разрядный сумматор).

На рис. 5.4 показана та же схема в топологическом представлении ПЛИС. Как и на рис. 5.3, можно видеть, что в проекте использованы компоненты в цепи ускоренного переноса.

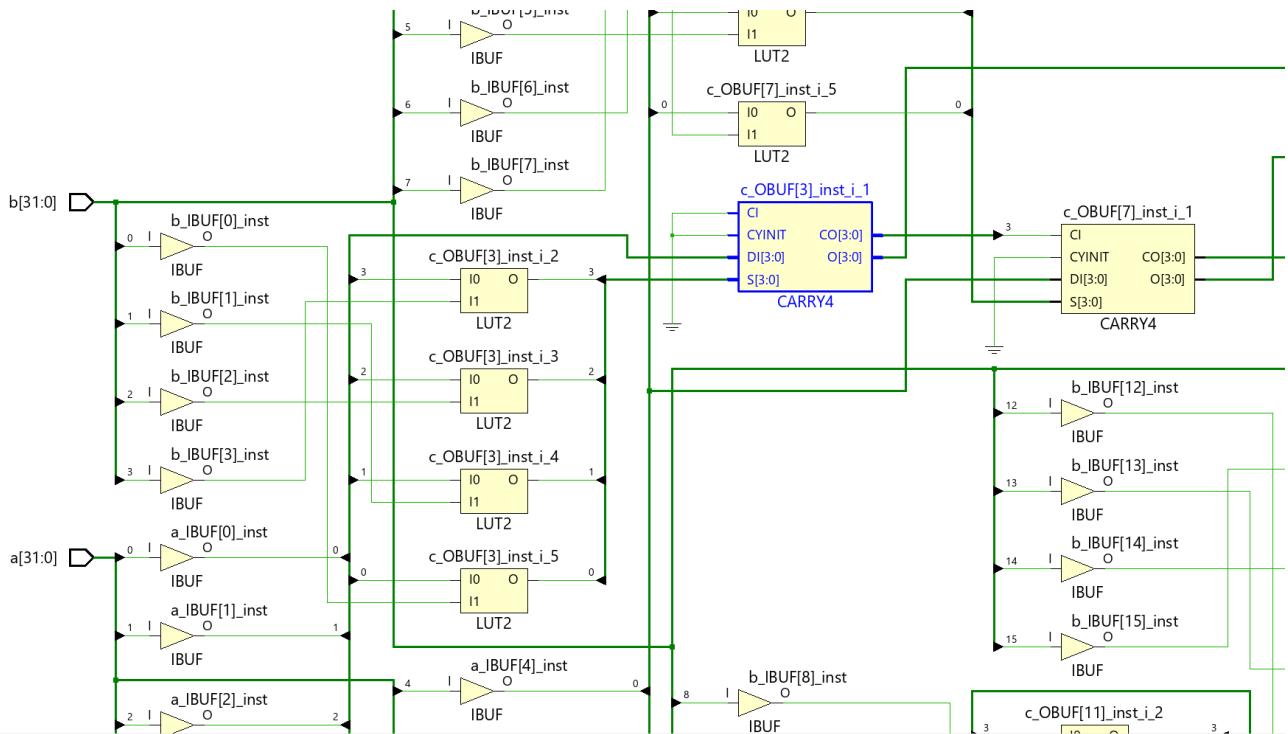


Рисунок 5.3. Реализация сумматора в ПЛИС

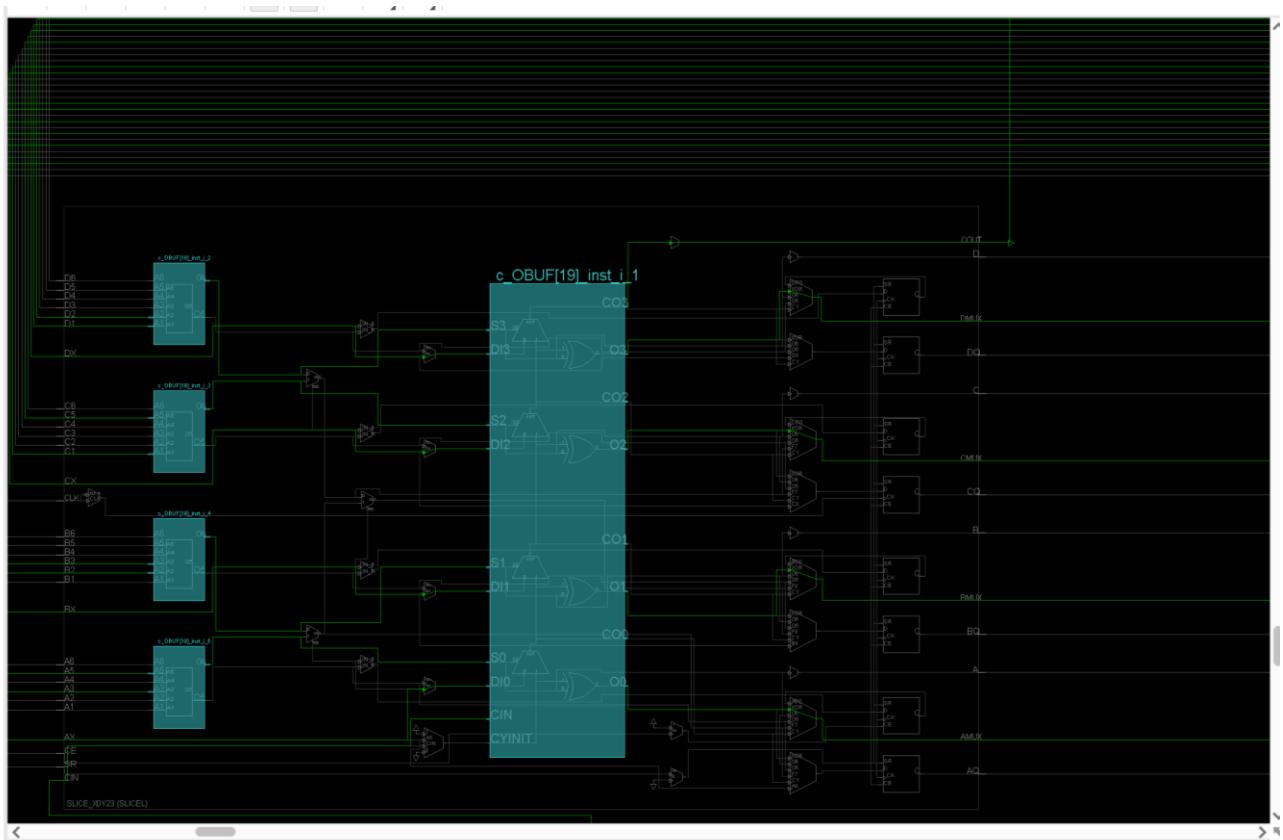


Рисунок 5.4. Реализация сумматора в топологическом представлении ПЛИС

Таким образом, оператор $+$ является не только допустимым, но и рекомендуемым способом описания сумматора в цифровой электронике. Для

ПЛИС это означает возможность применения аппаратных компонентов, специально предназначенных для построения таких схем, а для СБИС следует разделять функциональное проектирование, где сумматор является одним из узлов, и топологическую оптимизацию отдельных компонентов, выполняемую при необходимости. Например, если схема состоит из большого количества сумматоров, проектирование оптимального элемента может иметь смысл. Тогда сумматор может быть добавлен в схему в качестве субмодуля, который будет впоследствии оптимизирован. В обычной ситуации заменять сумматоры, получаемые синтезатором из описания вида $a + b$, не имеет практического смысла.

5.5. Вычитание. Смена знака

Вычитание производится аналогично сложению. Для него используется выражение вида «`assign c = a - b;`». Как и для сложения, рекомендуется использовать именно оператор вычитания, давая возможность синтезатору задействовать аппаратные компоненты ПЛИС.

Вычитание также можно рассматривать как сложение с отрицательным числом, т.е.

$$a - b = a + (-b)$$

Изменить знак числа в дополнительной двоичной арифметике достаточно просто. Для этого нужно проинвертировать все разряды этого числа и прибавить к результату 1.

Это несложно проверить, например, для числа 1.

$$\begin{aligned} 00000001 &\rightarrow 11111110 \rightarrow 11111111 \\ 11111111 &\rightarrow 00000000 \rightarrow 00000001 \end{aligned}$$

Поскольку вычитание реализуется схожим способом, синтезаторы часто объединяют сумматор и вычитатель в один компонент, использующий цепи ускоренного переноса. В отчетах синтезатора для ПЛИС часто можно видеть упоминание компонента `adder/subtractor` («сумматор/вычитатель»), который как раз и представляет собой объединенный компонент, основанный на аппаратных цепях ускоренного переноса.

5.6. Умножение

Для понимания простой схемы умножения можно рассмотреть процесс умножения в столбик. Для двоичного представления применяются простые правила:

$$0 * 0 = 0$$

$$0 * 1 = 0$$

$$1 * 0 = 0$$

$$1 * 1 = 1$$

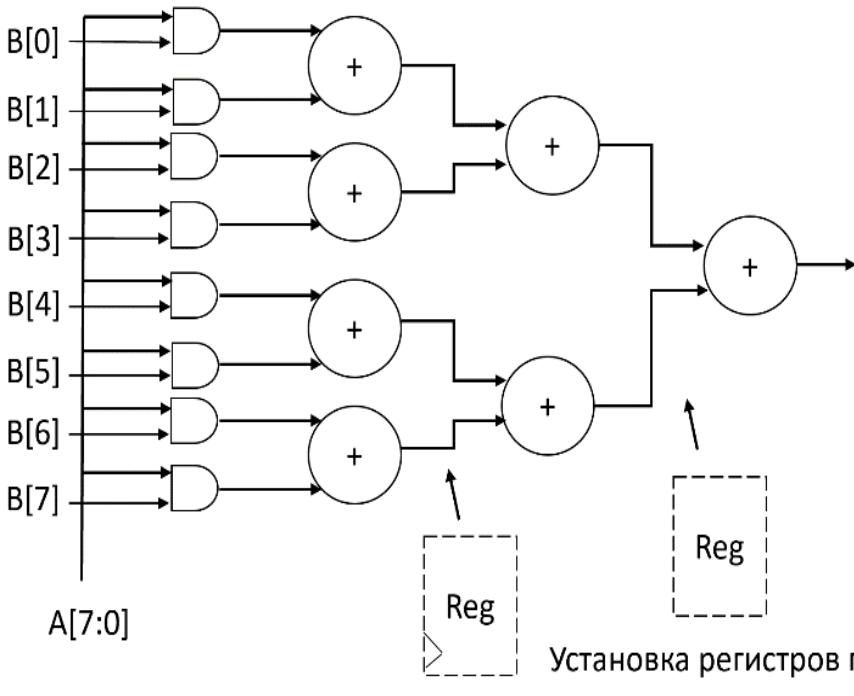
Можно видеть, что умножение однобитных чисел выполняется простым элементом И. Для многоразрядных чисел это правило несущественно изменяется – умножение любого числа на 0 дает в результате 0, а умножение на 1 – само это число.

$$\begin{array}{r} 110010 \\ \quad 1101 \\ \hline + \quad 110010 \\ + \quad 000000 \\ + \quad 110010 \\ + \quad 110010 \\ \hline 1010001010 \end{array}$$

Рисунок 5.5. Умножение двоичных чисел в столбик

Полученные промежуточные произведения (первый операнд на один из разрядов второго операнда) нужно сложить, сдвигая как показано на рис. 5.5. Удобнее складывать попарно, тогда максимальная длина цепочки сумматоров будет наименьшей. Например, для 8 промежуточных произведений нужно 4 сумматора, которые дадут 4 выхода. Эти выходы можно сложить уже двумя сумматорами, и, наконец, получить конечный результат одним сумматором. Получаемая структура называется деревом сумматоров (adder tree), она показана на рис. 5.6.

Если бы сложение производилось не попарно, а последовательно, цепочка содержала бы 7 сумматоров и задержка распространения сигнала была бы больше. Дерево сумматоров позволяет реализовать умножение с $\log_2(N)$ слоями сумматоров.



Установка регистров повышает частоту, но увеличивает число тактов до получения результата

Рисунок 5.6. Реализация умножения с помощью схемы «дерево сумматоров»

На рис. 5.6 показаны регистры, которые можно установить между отдельными слоями сумматоров. Тогда общая задержка распределяется между отдельными стадиями конвейера, и частота работы повышается. Однако это означает, что результат не может быть получен на том же такте. Дополнительное количество тактов определяется тем, сколько слоев регистров было установлено между сумматорами.

Применять ли конвейеризацию для умножения – неоднозначный вопрос. Если установить регистры после каждого слоя сумматоров, можно существенно повысить тактовую частоту этой схемы. Однако если результат умножения будет получаться через 3-4 или больше тактов, другие компоненты схемы могут простаивать. Это проявляется, например, если процессор проверяет условие, в которое входит результат умножения.

В то же время, если процессор выполняет умножение относительно редко, а медленный умножитель снижает его общую частоту, имеет смысл применить конвейеризацию, чтобы умножитель не стал самой медленной частью схемы процессора. Тогда операцию умножения будет необходимо разбить на несколько команд.

Для систем цифровой обработки сигналов часто используют конвейеризацию. Обычно потоковая обработка сигнала не накладывает ограничений на задержку в тактах.

Топологическая реализация дерева сумматоров представляет определенную сложность. Такая регулярная структура на первый взгляд кажется простой, однако требуется вмешательство разработчика для указания областей, в которых следует размещать отдельные сумматоры. Топологическое представление умножителя оказывается существенно зависящим от квалификации разработчика.

Производители ПЛИС размещают на кристалле большое количество аппаратных блоков, выполняющих умножение или умножение с накоплением. Компонент «умножение с накоплением», который называется также «секция DSP», показан на рис. 5.7.

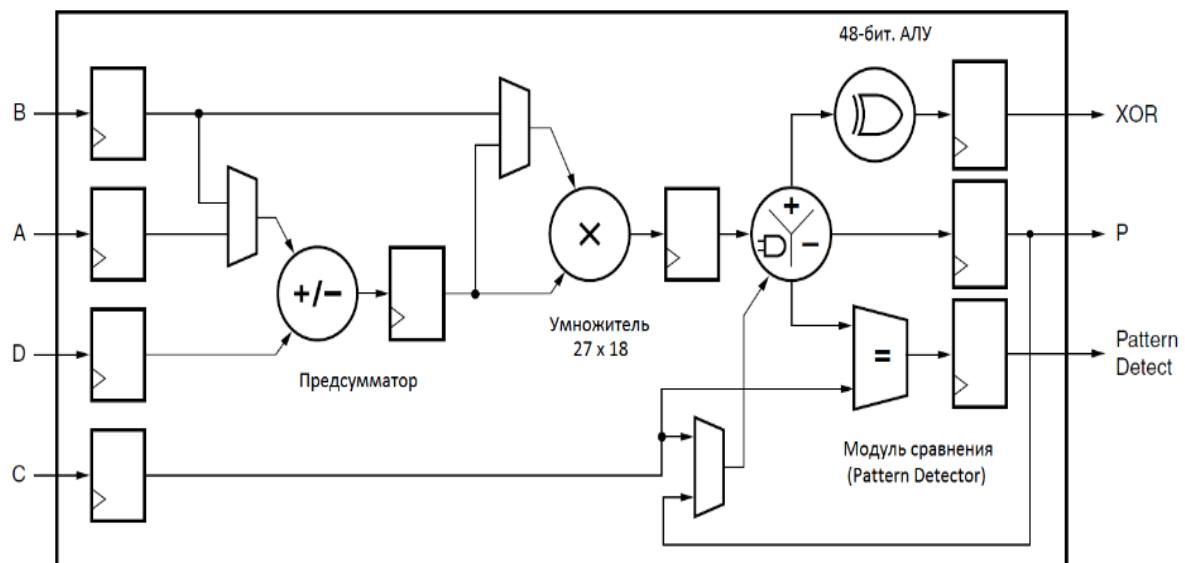


Рисунок 5.7. Компонент «умножение с накоплением» (секция DSP) в ПЛИС

Аппаратные умножители применяются синтезаторами в ПЛИС автоматически, по мере использования оператора *

```
assign m = a * b;
```

Также можно синтезировать умножение с накоплением:

```
assign sum = sum + k * x;
```

Дополнительные настройки синтезатора могут отключать использование секций DSP. Если в отчете синтезатора эти секции отсутствуют, необходимо проверить, установлено ли их использование для конкретного проекта.

Если проектирование выполняется для платформы, в которой отсутствуют аппаратные блоки умножения, имеет смысл выделить такой блок в субмодуль. На ранних этапах разработки внутри такого субмодуля может использоваться оператор `*`, чтобы синтезатор мог построить схему автоматически. При необходимости оптимизации этой схемы субмодуль может быть модифицирован, не затрагивая работу всей схемы. Например, при прототипировании СБИС можно применять секции DSP в ПЛИС, однако потом этот умножитель необходимо будет заменить.

Частным случаем умножения является умножение на целые степени двойки. Например, для умножения на 2 достаточно сдвинуть число на один разряд влево.

5.7. Деление

Деление иллюстрируется алгоритмом деления в столбик, как показано на рис. 5.8. В отличие от умножения в столбик, где можно рассчитывать произведения параллельно, при делении удобнее реализовать последовательный алгоритм, поскольку делимое на каждом шаге уменьшается или остается неизменным, в зависимости от величины делителя.

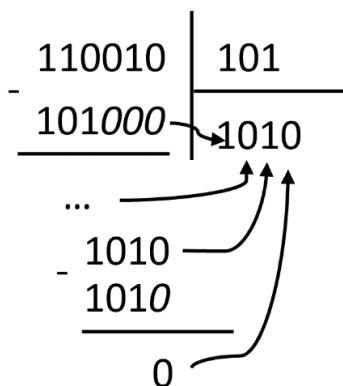


Рисунок 5.8. Деление двоичных чисел в столбик

Обычной практикой в АЛУ процессоров является применение пошагового деления целых чисел. Существует множество алгоритмов деления, отличающихся размером и числом тактов, среди которых невозможно указать наилучший.

В составе библиотеки Vivado имеется IP-ядро модуля деления целых чисел. Внешний вид диалогового окна настройки показан на рис. 5.9.

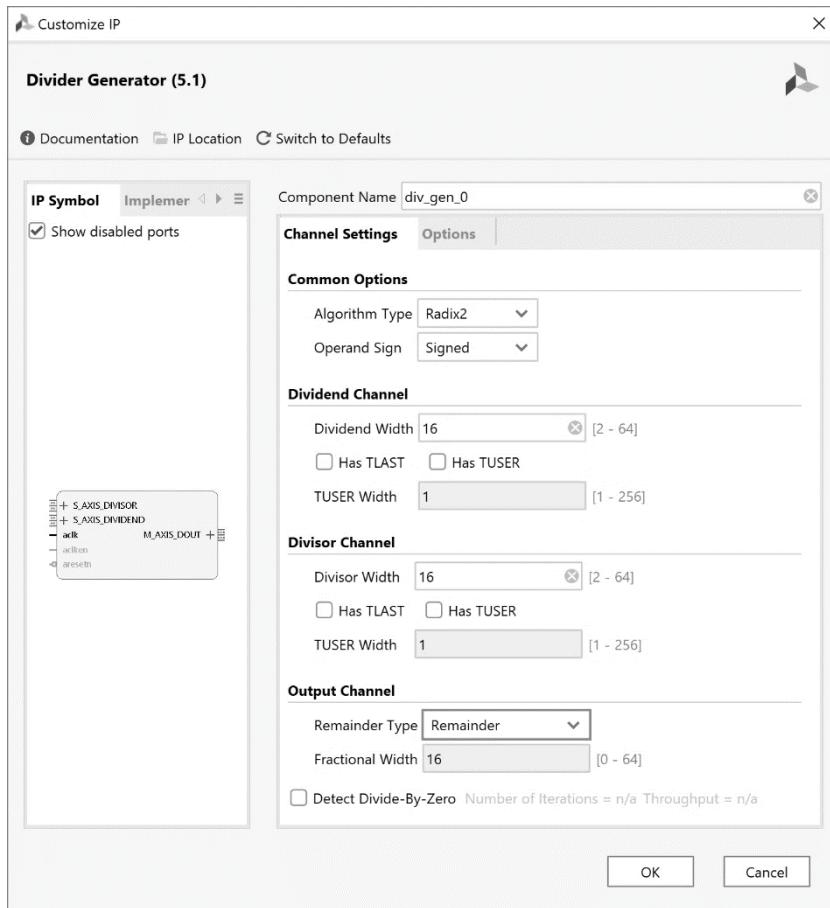


Рисунок 5.9. Генератор IP-ядер для деления двоичных чисел

Частным случаем деления является деление на целые степени двойки. Например, деление на 2 эквивалентно сдвигу делимого на один разряд вправо.

5.8. Числа с фиксированной точкой

Для двоичного числа можно представить, что часть его младших разрядов отделена двоичной точкой, и число имеет разряды не только с положительной степенью двойки, но и с отрицательной. Такие степени обрабатываются так же, как и положительные. Ниже приведены веса отдельных разрядов двоичного числа, у которого три младшие разряда отделены точкой.

Номер разряда	4	3	2	1	0	-1	-2	-3
Вес разряда	16	8	4	2	1	1/2	1/4	1/8

При показанной нумерации разрядов двоичное число 10001.010 будет переведено в десятичный вид как $1*16 + 1*1 + 1 * 1/4 = 17,25$. Двоичная точка отделяет три младших разряда, поэтому число, показанное в примере, может

обозначаться как 5.3, где 5 – количество разрядов до двоичной точки, а 3 – количество разрядов после нее.

Сложение и вычитание чисел с фиксированной точкой выполняется по тем же правилам, что и для целых чисел, однако количество разрядов, отделяемых двоичной точкой, должно быть одинаково. Если это не так, одно из чисел следует сдвинуть влево до совпадения положения двоичной точки.

При умножении числа могут быть перемножены без сдвига двоичного представления. Однако двоичная точка должна отделять столько разрядов в результате, сколько было отделено точкой у операндов в сумме. Т.е. при умножении чисел формата 8.8 и 8.4 результат будет иметь 28 разрядов, из которых 12 (8+4) должны быть отделены точкой, т.е. итоговый формат представляется как 16.12.

Двоичная точка сама по себе никак не представляется в аппаратуре. Разработчик самостоятельно приписывает разрядам двоичные веса, и именно разработчик ответственен за интерпретацию результата.

5.9. Числа с плавающей точкой

Формат с фиксированной точкой дает возможность представлять дробные числа, однако у него имеется недостаток, схожий с недостатком целых чисел. Рассмотрим сложение чисел 2^7 и 2^{-7} . Первое число выглядит в двоичном представлении как 10000000, а второе, с учетом разрядов после двоичной точки, как 0.00000001. Чтобы иметь возможность сложить эти числа, их необходимо записать в одинаковом формате, т.е. оба числа должны иметь 8 разрядов до точки, и 8 после нее. Поэтому для фиксированной точки возникает проблема увеличения разрядности, если необходимо представлять числа, существенно отличающиеся по величине. С учетом того, что в технике числа могут иметь диапазон от величин микромира до величин, описывающих характеристики Вселенной, разрядность числа с фиксированной точкой, способного представить любую величину, которая могла бы иметь практический смысл, очень велика. Принимая приближенно, что 3 десятичных разряда ($10^3=1000$) соответствуют 10 двоичным ($2^{10}=1024$), и рассматривая числа от постоянной Планка до числа атомов во Вселенной, можно получить разницу десятичных порядков на уровне 110-120, что эквивалентно примерно 400 разрядам двоичного числа.

Вместо резервирования избыточного количества разрядов можно записать только старшие значащие разряды двоичного представления, и отдельно указать, какому положению соответствует старший разряд полученной записи. Например, в записи 0.00000101 старшие нули могут быть отброшены, т.к. не

являются значащими. Чтобы понять, сколько разрядов было отброшено, необходимо отдельно записать это число.

В результате получается формат представления чисел с плавающей точкой. Он состоит из мантиссы (M) и порядка (E). Зная эти компоненты, можно получить число как $X = M * 2^E$. Для мантиссы отдельно представляется знак (S). В общем виде число с плавающей точкой записывается в двоичном виде в нескольких распространенных форматах, показанных на рис. 5.10.

S	E	M	
	5	10 (+1)	Half (16)
	8	23 (+1)	Single (32)
	11	52 (+1)	Double (64)

Рисунок 5.10. Форматы чисел с плавающей точкой

Например, для формата single порядок записывается в 8 разрядах, а мантисса – в 23. Однако с учетом того, что мантисса записывается в нормализованном виде, т.е. ее старший значащий разряд всегда равен 1, этот разряд можно не хранить. Поэтому к 23 разрядам, отведенным для хранения мантиссы, следует добавить еще один. Такой подход соблюдается для всех показанных форматов представления чисел. Считается, что мантисса лежит в диапазоне от $\frac{1}{2}$ до 1. Если смещение двоичной точки не требуется, то порядок принимается равным половине возможного диапазона, т.е. для числа одинарной точности с 8-разрядным порядком отсутствие смещения представляется числом 127.

При сложении и вычитании чисел с плавающей точкой необходимо сначала совместить положение точки. Для этого мантисса числа, порядок которого меньше, сдвигается вправо на величину разницы порядков ($E_2 - E_1$). После этого можно складывать или вычитать мантиссы.

Однако после сложения мантисс может оказаться, что результат не помещается в разряды, отведенные для хранения мантиссы. В этом случае необходимо выполнить нормализацию, т.е. привести мантиссу к диапазону $\frac{1}{2}..1$.

Порядок сложения чисел с плавающей точкой проиллюстрирован на рис. 5.411.

$$\begin{array}{l} .1100E2 \\ .1011E1 \end{array} \rightarrow \begin{array}{l} .1100E2 \\ .0101E2 \end{array} \rightarrow (.1100+.0101)E2 = 1.0001E2 = .1000E3$$

Выровнять
порядки

Сложить
мантицы

Нормализовать

Рисунок 5.11. Последовательность операций при сложении чисел в формате с плавающей точкой

На рис. 5.11 можно увидеть, что при выравнивании порядков младший разряд второго слагаемого оказался потерян, т.к. вышел за границы разрядной сетки, отведенной для хранения мантицы. Такое поведение является некоторой проблемой для чисел с плавающей точкой, и известно программистам как потеря точности и связанное с этим понятие «машинного нуля». Под машинным нулем понимается такая величина x , прибавление которой к 1 не изменяет двоичное представление результата в формате с плавающей точкой. Это легко понять, если представить сложение чисел, порядки которых отличаются больше, чем на размер мантицы. Тогда при попытке совместить двоичные точки все значащие разряды мантицы меньшего по модулю числа окажутся за пределами разрядной сетки, и при сложении мантисс первое число сложится с нулем. Проблема потери точности при операциях с плавающей точкой является объективным следствием выбранного формата представления этих чисел. При вычислениях следует иметь в виду данный эффект, который не имеет гарантированного способа устранения. Возможную потерю точности следует учитывать при анализе результата.

При умножении чисел с плавающей точкой их мантицы перемножаются, а порядки складываются (аналогично тому, как это происходит при умножении чисел, представленных в десятичной записи). При делении, соответственно, мантицы делятся, а порядок делителя вычитается из порядка делимого. Результат в том и другом случае может оказаться необходимо привести к нормализованному виду.

Числа с плавающей точкой и операции с ними описаны в широко используемом формате IEEE754. Этот формат поддерживается большинством процессоров, и содержит также информацию о правилах округления результата и хранения некоторых специальных величин – например, «бесконечность» и «не-число» (NaN, Not a Number). В силу широкой распространенности этих операций существует множество реализаций модулей для работы с числами с плавающей точкой. Их самостоятельная реализация чаще всего нецелесообразна из-за

наличия оптимизированных IP-ядер, имеющих хорошие характеристики производительности и учитывающие особенности архитектуры ПЛИС. На рис. 5.12. показано диалоговое окно настройки IP-ядра для работы с числами с плавающей точкой. Это ядро поставляется в составе САПР Vivado и имеет широкий диапазон настроек, включая базовые арифметические операции, преобразования форматов и др. Поддерживаются как показанные на рис. 5.11 основные форматы представления чисел, так и формат, настраиваемый пользователем, в котором можно указать собственные значения разрядности мантиссы и порядка.

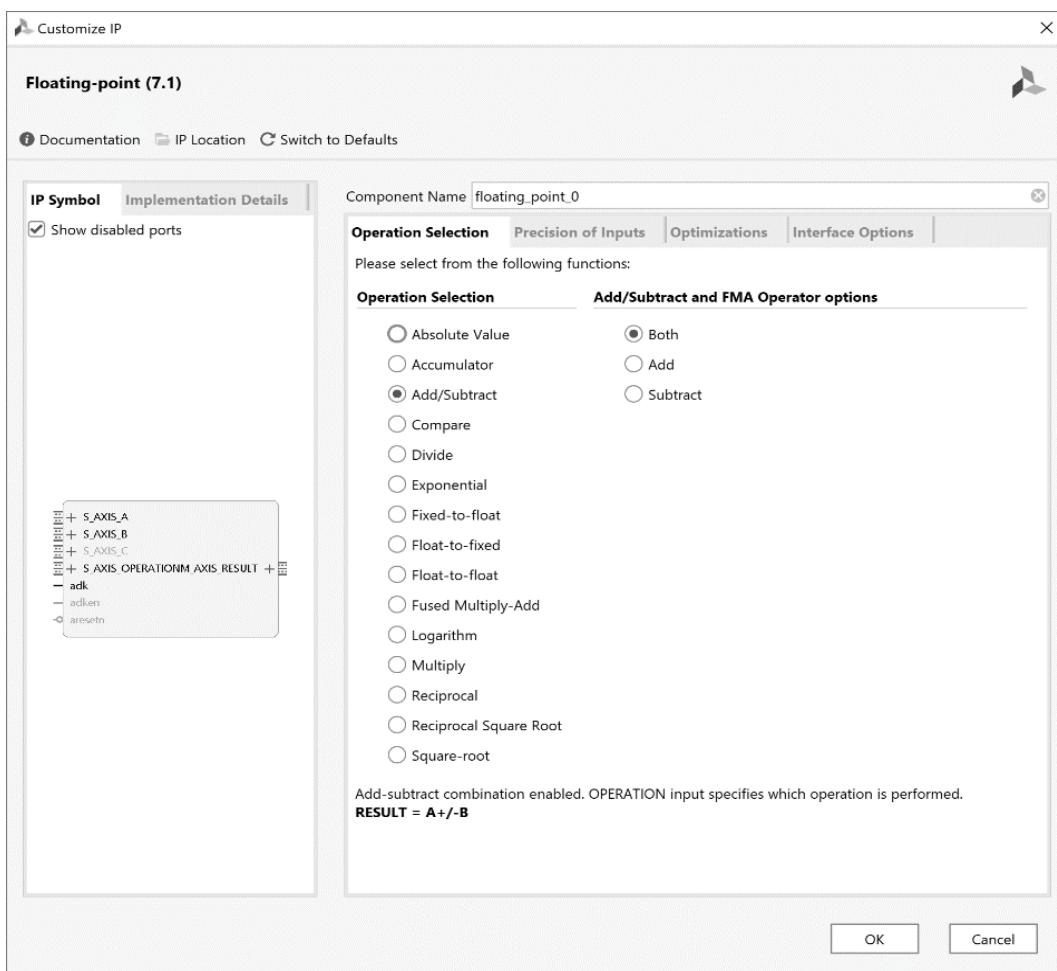


Рисунок 5.12. Генератор IP-ядер для выполнения операций с плавающей точкой

Можно указать важную причину для использования чисел с плавающей точкой – большой диапазон возможных значений для обрабатываемых чисел, который сложно записать в целочисленном формате или формате с фиксированной точкой. Например, такие формулы, как вычисление энергии электрона $E = e \cdot U$ или энергии кванта $E = \hbar \cdot v$ оперируют со значениями, существенно различающимся по значению десятичного порядка. Заряд

электрона равен $1,6 \cdot 10^{-19}$ Кл, а постоянная Планка $\hbar = 6,626 \cdot 10^{-34}$ Дж · с, тогда как разность потенциалов и особенно частота электромагнитной волны существенно больше. Подобные формулы, очевидно, требуют чисел с плавающей точкой для представления отдельных составляющих. При умножении чисел с плавающей точкой не происходит потеря точности, так как мантиссы можно перемножать без необходимости их взаимного сдвига.

5.10. Вычисление трансцендентных функций на базе алгоритма CORDIC

Алгоритм CORDIC (Coordinate Rotating Digital Computer, т.е. «цифровой вычислитель для вращения координат») представляет удобный способ для точного вычисления тригонометрических функций без потери точности.

Алгоритм основан на том, что при вращении координат некоторого вектора отслеживаются как угол поворота вектора, так и связанные с ним координаты конца этого вектора x , y , которые при единичной длине вектора представляют собой косинус и синус от угла поворота. Иллюстрация к алгоритму показана на рис. 5.13.

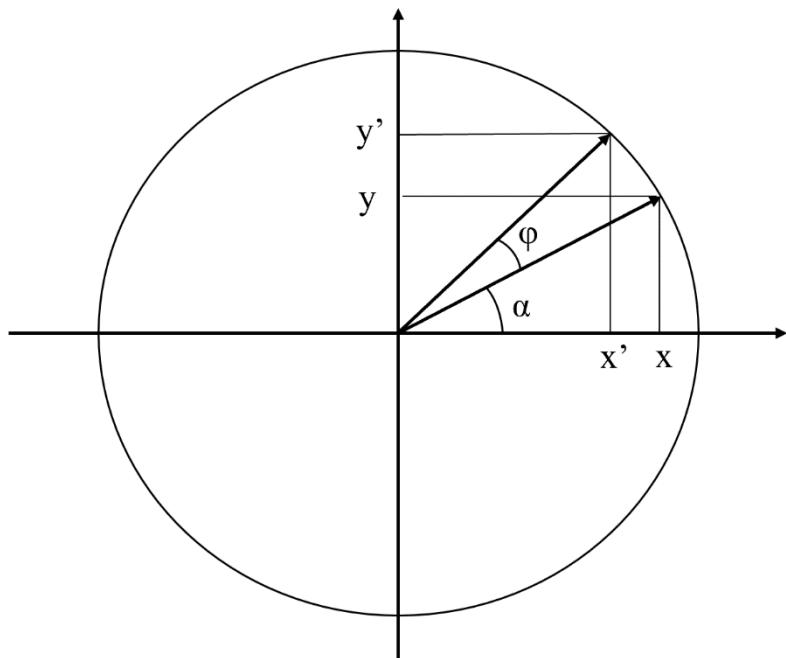


Рисунок 5.13. Иллюстрация к вычислению трансцендентных функций

При повороте вектора $(x; y)$ с углом к оси абсцисс α на угол φ его новые координаты $(x'; y')$ станут равны:

$$x' = \cos(\alpha + \varphi) = \cos \alpha \cdot \cos \varphi - \sin \alpha \cdot \sin \varphi$$

$$y' = \sin(\alpha + \varphi) = \sin \alpha \cdot \cos \varphi + \cos \alpha \cdot \sin \varphi$$

Вынесем за скобку $\cos \varphi$:

$$\begin{aligned}x' &= \cos \varphi (x \cdot 1 - y \cdot \tan \varphi) \\y' &= \cos \varphi (y \cdot 1 + x \cdot \tan \varphi)\end{aligned}$$

Выражение $\tan \varphi$ в скобках появилось из-за того, что при вынесении $\cos \varphi$ понадобилось сначала умножить $\sin \varphi$ на дробь $\cos \varphi / \cos \varphi$. Выражение $\cos \varphi$ в числителе оказалось возможным вынести за скобку, при этом оставшиеся $\sin \varphi / \cos \varphi$ образовали функцию $\tan \varphi$.

На первый взгляд, полученные выражения ничего принципиально не изменили. Однако условимся, что поворот будет производиться только на такие углы, для которых $\tan \varphi$ равен целой степени двойки (т.е. 1, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ и т.д.). В этом случае умножение $x \cdot \tan \varphi$ равносильно сдвигу x на 0, 1, 2, 3... позиции вправо. Аналогично выполняется вычисление $y \cdot \tan \varphi$. Таким образом, умножение на синус и косинус в алгоритме оказалось заменено на умножение на 1 (благодаря вынесению за скобки $\cos x$) и умножение на степени двойки, которое может быть выполнено сдвигом.

Имея набор углов поворота φ , тангенсы которых равны целой степени двойки, можно последовательными поворотами на положительные и отрицательные углы добиться того, чтобы суммарный угол поворота стал равен некоторому значению. Для этого необходимо учесть, что:

$$\begin{aligned}x' &= \cos(\alpha - \varphi) = \cos \alpha \cdot \cos \varphi + \sin \alpha \cdot \sin \varphi \\y' &= \sin(\alpha - \varphi) = \sin \alpha \cdot \cos \varphi - \cos \alpha \cdot \sin \varphi\end{aligned}$$

Имея таблицу углов, и проинициализировав $x = 1$, $y = 0$, можно повернуть угол φ максимально близко к искомому углу φ_0 , заданному в качестве аргумента. Если после очередного поворота $\varphi > \varphi_0$, то на следующей итерации алгоритма поворот происходит на отрицательный угол. Таким образом, методом последовательного приближения φ устремляется φ_0 , а его координаты x , y естественным образом представляют косинус и синус угла φ .

Можно заметить, что на каждой итерации за скобки выносится $\cos \varphi$, на которую производится умножение. Однако поскольку значения углов на каждой итерации известны, итоговое произведение можно вычислить заранее и учесть его при коррекции результата. Также можно инициализировать x значением, обратным произведению всех величин $\cos \varphi$.

Алгоритм является хорошо известным и его самостоятельная реализация хотя и возможна, но при первом приближении даст худшие результаты по сравнению с оптимизированным IP-ядром, входящим в поставку САПР Vivado. Внешний вид диалогового окна настройки этого ядра показан на рис. 5.14. Можно видеть, что ядро выполняет как вычисление синуса и косинуса (эти операции выполняются одновременно, поскольку данные функции являются проекциями одного и того же вектора), гиперболические синус и косинус $\sinh x$, $\cosh x$, а также арктангенс, гиперболический арктангенс и квадратный корень.

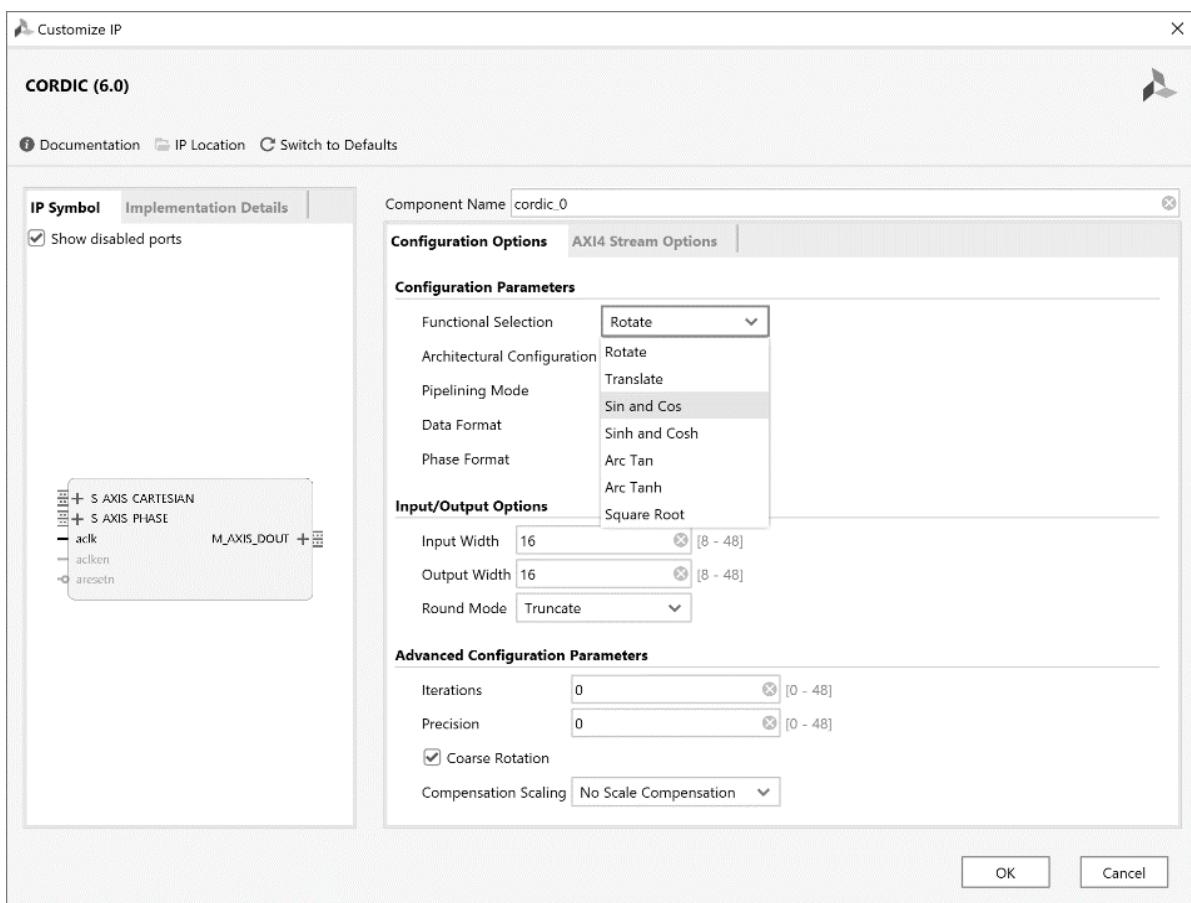


Рисунок 5.14. Генератор IP-ядер для реализации операций на базе алгоритма CORDIC

Алгоритм CORDIC, в отличие от разложения в ряд, дает точные (в пределах разрядности) значения тригонометрических функций и может быть практически неограниченно расширен до требуемой точности. Он настоятельно рекомендуется как основной при необходимости вычисления тригонометрических функций в проекте.

Вычисление экспоненты базируется на формулах гиперболических функций:

$$\begin{aligned}\sinh x &= (e^x - e^{-x}) / 2, \\ \cosh x &= (e^x + e^{-x}) / 2.\end{aligned}$$

Сложив левые и правые части этих уравнений, получаем:

$$\begin{aligned}\sinh x + \cosh x &= e^x, \\ \cosh x - \sinh x &= e^{-x}.\end{aligned}$$

Вычисление гиперболических функций может быть произведено с помощью алгоритма CORDIC с соответствующим оптимизированным IP-ядром. Однако особенностью этого алгоритма при вычислении гиперболических функций является ограниченный диапазон возможных аргументов. Дело в том, что вычисление этих функций происходит путем замены знаков в формулах поворота угла, в результате чего конец вектора движется по гиперболической кривой с уравнением $x^2 - y^2 = 1$. При слишком больших значениях аргумента вектор, проведенный под таким углом, не пересечет гиперболическую кривую, и результат последовательных поворотов вектора окажется некорректным.

Для решения этой проблемы следует воспользоваться свойством степенных функций:

$$e^{a+b} = e^a \cdot e^b.$$

Если разложить аргумент экспоненты на два слагаемых, результирующую экспоненту можно будет вычислить как произведение двух составляющих. При этом одна часть аргумента может быть представлена в таблице, а вторая – получена с помощью алгоритма CORDIC через гиперболические функции. Первую часть аргумента следует выбирать, например, с шагом 0,5, т.е. задавать в таблице значения экспонент для аргументов 0,5, 1, 1,5, 2 и т.д. Поскольку экспонента быстро возрастает при росте аргумента (и быстро убывает для отрицательных аргументов), размер таблицы будет невелик и составит несколько десятков значений для 32-разрядного представления.

5.11. Табличная реализация функций

Табличное представление является одним из простейших и применимо для любых функций. Если функция является алгебраической, ее табличное представление окажется существенно расточительнее по ресурсам по сравнению с вычислением на базе логических ячеек и, возможно, блоков DSP48.

Целесообразность использования табличного представления функции напрямую зависит от разрядности ее аргумента. Для небольшой разрядности (8 – 12) вполне допустимо выделить таблицу в 256 – 4096 значений функции, напрямую записанных в память ПЛИС. Ресурсов блочной памяти вполне достаточно для представления таблиц таких размеров. Однако уже для 16-разрядного аргумента размером необходимого блока памяти нельзя пренебрегать, а для 32-разрядного аргумента потребуется хранение 4 Гб, что является чрезмерным.

Для тригонометрических функций \sin , \cos можно воспользоваться следующим их свойством:

$$\begin{aligned}\sin(\alpha + \beta) &= \sin \alpha \cdot \cos \beta + \cos \alpha \cdot \sin \beta, \\ \cos(\alpha + \beta) &= \cos \alpha \cdot \cos \beta - \sin \alpha \cdot \sin \beta.\end{aligned}$$

Если представлять угол как сумму «грубого» угла α (например, от 0 до 90 с шагом в 1°) и «точного» угла β (например, от 0 до 1° с шагом в 0.01°), то для «грубого» угла потребуется таблица на 90 значений, а для «точного» – на 100. Вместе с тем приведенные формулы обеспечивают вычисление синуса или косинуса для 9000 возможных значений аргумента от 0 до 90 с шагом $0,01^\circ$. Для вычисления синуса или косинуса суммы потребуются также блоки умножения, однако такой подход позволит сократить размер памяти за счет привлечения блоков DSP48.

Такие функции, как факториал, или числа Фибоначчи, являются интересным примером применения таблиц для их представления. Несмотря на то, что факториал часто приводится в качестве примера, демонстрирующего применение рекурсивного алгоритма, на практике диапазон возможных аргументов для факториала довольно мал. Например, уже $100! = 9,3326\text{E}+157$, поэтому для хранения практически используемых значений факториала достаточно небольшой таблицы.

5.12. Выводы по разделу

Основные арифметические функции существенно различаются по сложности реализации в вычислительных устройствах. Проще всего реализуются поразрядные логические операции. Реализация сложения и вычитания несущественно сложнее, однако уже на этом уровне необходимо следить за синтезируемой схемой и использованием специализированных аппаратных ресурсов.

Операция умножения требует существенно больше ресурсов и представляет определенные проблемы с точки зрения оптимальной реализации топологии. Обычной практикой является применение готовых компонентов для реализации умножения – например, секций DSP в ПЛИС.

Операция деления целых чисел в силу особенностей алгоритма обычно выполняется последовательно, с вычислением одного бита результата за такт работы схемы.

Для поддержки типичных вычислительных задач используют форматы с фиксированной точкой и с плавающей точкой. Они имеют несколько отличающиеся свойства, однако позволяют расширять диапазон чисел, которые можно представить в вычислительной системе, по сравнению с целочисленным форматом.

Основой для реализации трансцендентных функций является алгоритм CORDIC. Он математически точек (один такт на разряд результата) и имеет различные варианты практической реализации.

Контрольные вопросы:

1. Как выглядит таблица истинности для сумматора одноразрядных операндов?
2. Что такое дополнительное двоичное представление числа? Как изменить знак такого числа?
3. Сколько сумматоров необходимо в модуле умножения для 8-разрядных операндов? Для 16-разрядных?
4. Как выполнить операцию деления для двоичных чисел?
5. Можно ли выполнить умножение целых чисел последовательно?
6. Как повернуть вектор, представленный в алгебраическом виде, на заданный угол? Как это используется в алгоритме CORDIC?
7. Какие форматы чисел удобнее для представления следующих величин (необходимо обосновать выбор между целочисленным, фиксированной точкой и плавающей точкой):
 - температура воздуха в помещении;
 - напряжение аккумулятора;
 - суммарная энергия, прошедшая через трансформаторную подстанцию;
 - механическое напряжение в элементе конструкции при расчете в САПР.

6. РЕАЛИЗАЦИЯ ОСНОВНЫХ СИНХРОННЫХ УСТРОЙСТВ

6.1. Содержание раздела

Ранее было упомянуто, что современные цифровые схемы должны основываться на синхронных элементах. Это элементы, изменяющие свое состояние не в произвольные моменты времени, а в строго определенный – обычно это фронт тактового сигнала. В данном разделе рассматриваются разновидности синхронных цифровых узлов и способы их проектирования на языках описания аппаратуры.

6.2. Триггер, его разновидности и порядок описания

Наиболее распространенным синхронным элементом является триггер. Формально, к триггерам относится целый ряд цифровых элементов, которые могут находиться в устойчивом состоянии, однако не все триггеры являются синхронными. На рис. 6.1 показано условное изображение D-триггера, который представляет собой синхронный элемент.

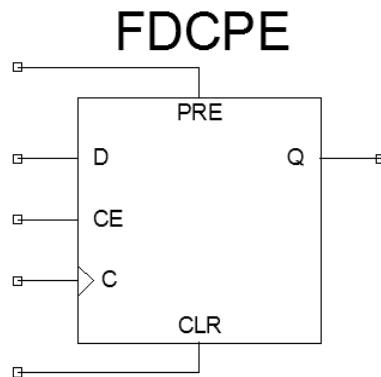


Рисунок 6.1. Условное изображение D-триггера

Показанный триггер имеет следующие выводы:

- С – вход тактового сигнала;
- D – вход данных;
- CE – вход разрешения записи (clock enable);
- CLR – вход асинхронного сброса (clear);
- PRE – вход асинхронной установки в логическую единицу (preset);
- R – вход синхронного сброса (reset);
- S – вход синхронной установки в логическую единицу (set);
- Q – выход данных.

В D-триггере не обязательно использовать все входы. В минимальном варианте достаточно иметь тактовый вход C (также часто обозначается как CLK), и вход данных D. Очевидно, что выход данных Q должен быть всегда.

В целом работу D-триггера можно описать простым правилом:

«В момент прихода фронта тактового сигнала выход Q принимает состояние входа D».

Дополнительные входы управления корректируют выполнение этого правила. На рис. 6.2 показаны временные диаграммы работы триггера, у которого присутствуют дополнительные входы управления, в том числе и асинхронные.

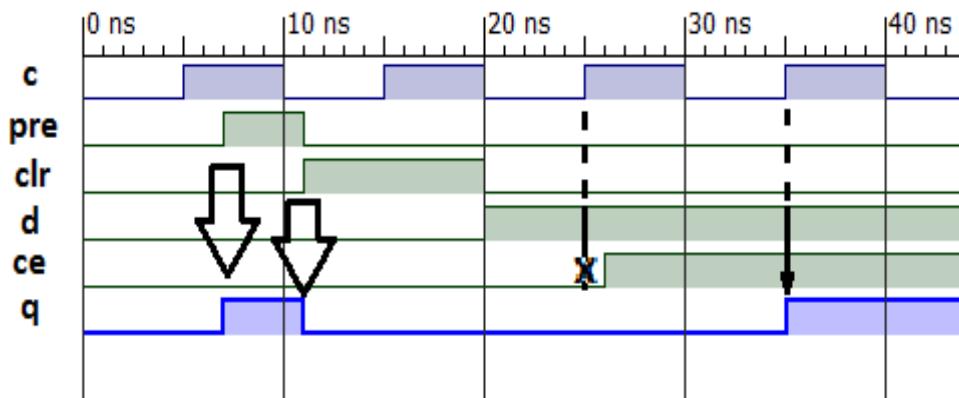


Рисунок 6.2. Временные диаграммы работы триггера

На диаграмме можно видеть следующие основные события, происходящие с триггером.

1. В момент времени 7 нс появляется сигнал PRE — это сигнал асинхронной установки. Наличие этого сигнала переводит триггер в состояние логической единицы вне зависимости от наличия фронта тактового сигнала.

2. В момент времени 11 нс появляется сигнал CLR — это сигнал асинхронного сброса. Наличие этого сигнала переводит триггер в состояние логического нуля вне зависимости от наличия фронта тактового сигнала.

Если действуют оба сигнала, PRE и CLR, поведение триггера зависит от его реализации. Часто оба асинхронных сигнала не реализуются. Кроме того, асинхронный сброс и установка не рекомендуются к применению, хотя и допустимы с точки зрения цифровой схемотехники.

3. В момент времени 25 нс нет асинхронных сигналов, есть фронт тактового сигнала, но триггер не реагирует на вход данных, поскольку у него неактивен вход CE («разрешение счета»).

4. В момент времени 35 нс триггер находится в наиболее показательном состоянии — нет сигналов PRE, CLR, активен сигнал CE, поэтому происходит запись в триггер того уровня, который присутствует на входе D.

Описание D-триггера на языке Verilog приведено ниже:

```
module dff( input clk,
             input d,
             output reg q) ;

    always @ (posedge clk)
        q <= d;

endmodule
```

В VHDL триггер описывается аналогично. Можно обратить внимание на отсутствие спецификатора `reg` для выходного сигнала, как в Verilog. Синтезатор VHDL определяет необходимость выбора триггера автоматически.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dff is
    Port ( clk : in STD_LOGIC;
            d : in STD_LOGIC;
            q : out STD_LOGIC);
end dff;

architecture Behavioral of dff is

begin

process(clk)
begin
    if rising_edge(clk) then
        q <= d;
    end if;
end process;

end Behavioral;
```

6.3. Асинхронный и синхронный сброс, рекомендуемые практики применения управляющих сигналов

При рассмотрении работы триггера были показаны варианты сброса и установки. Эти действия имеют приоритет перед записью данных с входа D. При этом сброс и установка могут быть как асинхронными (т.е. происходить сразу при появлении на входе соответствующего сигнала), так и синхронными (т.е. происходить по фронту тактового сигнала). Синтезаторы могут выбирать соответствующий вариант триггера на основе его RTL-представления.

Триггер с синхронным сбросом:

```
always @ (posedge clk)
  if (reset)
    begin
      q <= 0;
    end
  else
    begin
      q <= d;
    end
```

Триггер с асинхронным сбросом:

```
always @ (posedge clk or posedge reset)
  if (reset) begin
    q <= 0;
  end
  else if (ce) begin
    q <= d;
  end
```

Можно обратить внимание, что для асинхронного сброса в список чувствительности добавлено выражение posedge reset. Это необходимо, чтобы при моделировании такой схемы изменение сигнала reset правильно обрабатывалось программой-симулятором.

Сравнивая синхронный и асинхронный сброс, может показаться, что асинхронный сброс выглядит привлекательнее. Триггер можно сбросить в любой момент, даже коротким импульсом. Для синхронного сброса нужно дождаться фронта тактового сигнала.

Тем не менее, проблема кроется в замечании «даже коротким импульсом». При уменьшении технологических норм это становится проблемой. Логическая модель покажет короткие импульсы, но реальный кристалл из-за вариаций техпроцесса может исказить их длительность и момент прихода на входы триггеров.

Правильно работающая модель с асинхронными сбросами не является доказательством того, что реальный кристалл воспроизведет эти импульсы также. Некоторые триггеры могут не воспринять короткий сигнал сброса.

Таким образом, рекомендуемый вариант сброса – синхронный.

Внешние источники сигналов сброса асинхронны по отношению к проекту. В этом случае требуется *синхронизация* сигналов, что будет рассмотрено в разделе 8.

6.4. Управляющие наборы (control sets) в ПЛИС

Ячейки современных ПЛИС имеют некоторые особенности организации триггеров. Например, секция в FPGA Xilinx, содержащая 8 триггеров, обобщает все сигналы управления (clk, ce, set/reset). Речь идет не просто о том, что сигнал ce должен иметь высокий активный уровень, а о том, что все триггеры должны использовать один и тот же сигнал. Это сделано производителем для экономии ресурсов, поскольку 8 независимых наборов сигналов займут слишком много места. Нельзя утверждать, что это универсальное правило для всех ПЛИС, однако текущие семейства имеют такую конструкцию ячеек.

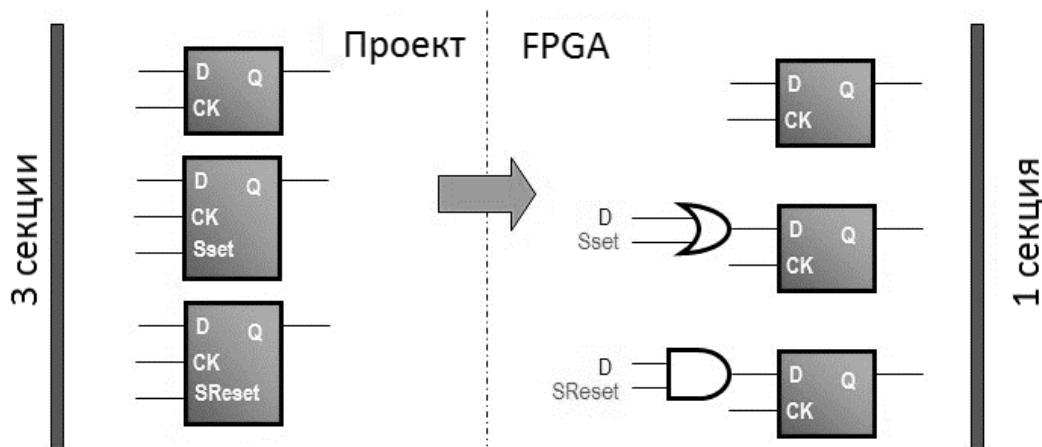


Рисунок 6.3. Примеры преобразования управляющих сигналов в триггерах ПЛИС

Это ограничение не препятствует реализации схем с разными сигналами, просто триггеры могут быть размещены в разных секциях. Кроме этого, синтезатор может использовать приемы преобразования управляющих сигналов

в вентили, управляющие данными. Примеры таких автоматических преобразований показаны на рис. 6.3.

Управляющие наборы не являются предметом постоянного внимания при разработке. Тем не менее, наличие ограничений может объяснить, почему схема занимает больше ячеек ПЛИС, чем предполагалось разработчиком.

6.5. Особенности проектирования и моделирования сигнала сброса

Сброс регистров в заранее определенное состояние является нормальной практикой проектирования цифровых систем. Это устраняет неопределенность, которая могла бы возникнуть при разнообразных процессах в момент включения питания, и широко практикуется при построении самых разных узлов. При разработке СБИС сброс синхронных компонентов является обязательным.

Те же соображения применимы и для FPGA, однако здесь имеется важное дополнение. Сами микросхемы FPGA уже имеют схемы сброса, и установка всех синхронных компонентов в начальное состояние является частью процесса загрузки конфигурации. Поэтому попытка установить все регистры в начальное состояние при старте проекта будет просто дублированием тех действий, которые только что были проведены. Может показаться, что это несущественная деталь, которая просто увеличивает надежность проекта, однако на рис. 6.4 можно ознакомиться с примером трассировки избыточного сигнала сброса.

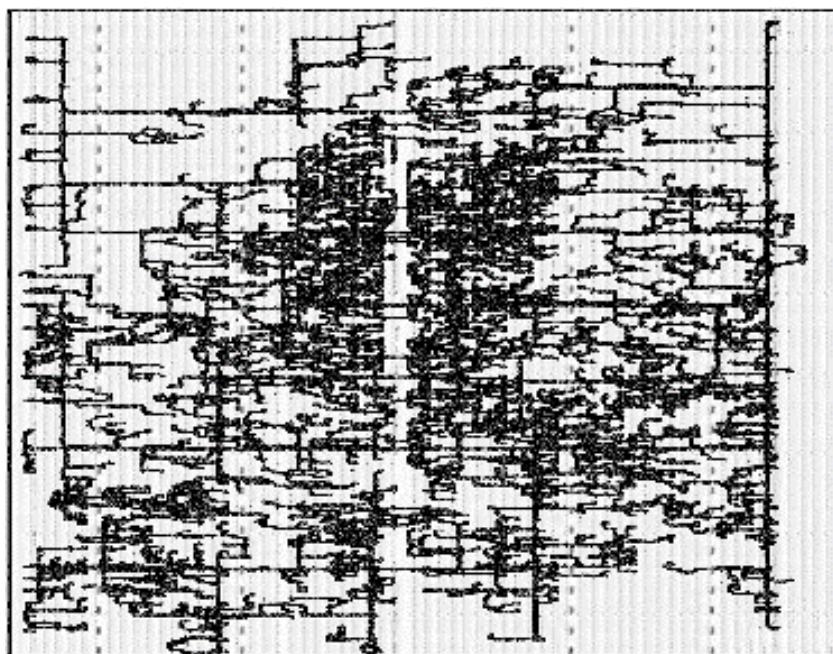


Рисунок 6.4. Примеры трассировки цепи сброса в ПЛИС

Разумеется, такой сигнал вряд ли сделает трассировку проекта невозможной, однако эти линии могут занять места, которые в ином случае были бы использованы для трассировки более критичных к задержкам сигналов.

Некоторые компоненты ПЛИС не могут быть сброшены. Например, массив памяти не имеет сигнала сброса, который мог бы обнулить все ячейки памяти. Попытка описать сброс для массива памяти вызовет его реализацию на базе триггеров, что в целом нерационально, поскольку блок памяти существенно компактнее по сравнению с набором триггеров такого же логического объема.

В целом можно придерживаться следующих соображений:

1. Лучшим приемом является отказ от глобального сброса, поскольку он уже реализован в ПЛИС.

2. Локальные сигналы сброса, если они необходимы для проекта, должны быть синхронными и использовать высокий логический уровень.

При поведенческом описании сигналов можно использовать инициализацию. Она будет использована не только для моделирования, но и для загрузки в соответствующий регистр при инициализации ПЛИС.

6.6. Разработка счетчика

Счетчик выполняет последовательное увеличение или уменьшение своего выходного значения по каждому тактовому импульсу. Простейший вариант счетчика показан ниже:

```
reg [7:0] cnt;  
always @ (posedge clk)  
    cnt <= cnt + 1;
```

Поскольку для хранения значения счетчика выбрано 8 разрядов, счетчик будет осуществлять циклическое приращение своего значения от 0 до 255, после чего операция 255+1 опять сделает значение `cnt` равным 0.

В показанном примере счетчик работает строго в диапазоне от 0 до 255. Переключение к новой последовательности происходит естественным образом, в результате переполнения регистра-счетчика.

Если нужно обеспечить счет до определенного значения, которое не может быть обеспечено простым переполнением, следует использовать внутри процесса `always` условный оператор, загружающий в счетчик 0, если достигнут предел счета, или продолжающий счет в противном случае.

Для счетчиков используются следующие возможности:

- разрешение счета;

- регулируемое направление счета (up/down);
- возможность сброса;
- возможность загрузки.

Счетчик с регулируемым направлением счета приведен в следующем примере:

```
reg [7:0] cnt;

always @ (posedge clk)
  if (up_down)
    cnt <= cnt + 1;
  else
    cnt <= cnt - 1;
```

Счетчик имеет дополнительный управляющий вход up_down. Высокий логический уровень на этом входе означает счет на увеличение значения счетчика, а низкий – на уменьшение.

Счетчик с загрузкой имеет дополнительные входы – разрешение загрузки и загружаемые данные. Если на входе разрешения загрузки присутствует активный уровень, то счетчик принимает значение, заданное внешней шиной. Таким образом, с помощью этого интерфейса можно принудительно задать требуемое значение счетчика.

Пример счетчика с синхронным сбросом и загрузкой:

```
reg [7:0] cnt;

always @ (posedge clk)
  if (reset)
    cnt <= 0;
  else if (ce)
    if (load)
      cnt <= d_in;
    else
      cnt <= cnt + 1;
```

Как и для остальных цифровых модулей на базе ПЛИС, рекомендуется использовать синхронный сброс вместо асинхронного.

Двоичное кодирование является не единственным возможным алгоритмом работы счетчика. Вместо последовательного перебора двоичных значений в процессе работы возможно использование и других кодировок. Например, код

Грея (Gray code) имеет то свойство, что для перехода к следующему значению достаточно изменить значение единственного разряда. Это полезно при обработке сигналов, в которых существует вероятность сдвига по времени между отдельными разрядами. Например, при переходе от двоичного состояния 0111 (7_{10}) к 1000 (8_{10}) из-за неодновременной смены разрядов может появиться состояние 0000, 1111 (или любое другое, в зависимости от порядка смены разрядов). В то же время подобный эффект при использовании кода Грея приведет к максимальной ошибке, равной 1.

Пример реализации счетчика, основанного на коде Грея:

```
parameter gray_width = 8;

reg [gray_width-1:0] binary_value;
reg [gray_width-1:0] gray_value;

always @ (posedge clk)
  if (reset) begin
    binary_value <= {{gray_width{1'b0}}, 1'b1};
    gray_value <= {gray_width{1'b0}};
  end
  else if (ce) begin
    binary_value <= binary_value + 1;
    gray_value <=
      (binary_value >> 1) ^ binary_value;
  end
```

Другой разновидностью кодирования является LFSR (*Linear Feedback Shift Register*) – сдвиговый регистр с линейной обратной связью. Пример 4-разрядного LFSR:

```
reg [3:0] lfsr;

always @ (posedge clk)
  if (reset)
    lfsr <= 4'h0;
  else if (ce) begin
    lfsr[3:1] <= lfsr[2:0];
    lfsr[0] <= ~^lfsr[4:3];
  end
```

Особенностью кодирования по LFSR является более быстрая смена состояний по сравнению с двоичным счетчиком, поскольку при двоичном кодировании возможна ситуация, когда прибавление единицы сменит все разряды, включая самый старший. На распространение бита переноса по всем разрядам двоичного счетчика тратится дополнительное время, по сравнению со сдвиговыми регистрами, в которых каждый разряд получает свое значение от соседнего разряда. Вдвигаемое в LFSR значение определяется в строке `lfsr[0] <= ~^lfsr[4:3]`, и зависит от разрядности счетчика.

Недостатком счетчика LFSR является меньшее число уникальных состояний по сравнению с двоичным счетчиком той же разрядности.

6.7. Широтно-импульсная модуляция

Широтно-импульсная модуляция (ШИМ, также PWM – Pulse-Width Modulation) является эффективным способом цифрового управления силовыми системами. При этом регулирование мощности осуществляется путем управления отношением времени, в течение которого сигнал включен, ко времени, в течение которого он выключен. Такой способ управления обладает как минимум двумя достоинствами – он удобен для реализации в цифровой системе, и является энергоэффективным, поскольку в ключевом режиме работы управляющий элемент потребляет минимальную мощность (в закрытом состоянии ток через него стремится к нулю, а в открытом стремится к нулю падение напряжения, поскольку сопротивление переключающих элементов стремится уменьшить).

Модуль ШИМ разрабатывается следующим образом. Входной сигнал `d` задает число тактов, в течение которых следует удерживать высокий логический уровень на выходе модуля. Внутренний сигнал `cnt` циклически перебирает состояния от 0 до максимального значения, которое может быть подано в качестве входного. Допустим, что внутренний счетчик является 8-разрядным (т.е. полный цикл счета содержит 256 тактов). Тогда при подаче на вход числа 10 на выходе такого модуля будет логическая единица в течение 10 тактов, а в течение остальных 246 – логический ноль. Увеличивая значение числа, поданного на вход `d`, можно увеличивать отношение времени включения выходного сигнала к общему времени цикла счета.

```
module pwm (
    input clk,
    input [7:0] d,
    output pwm
```

```

) ;

reg [7:0] cnt = 0;

always @ (posedge clk)
  cnt <= cnt + 1;

assign pwm = (d > cnt) ? 1 : 0;

endmodule

```

Для демонстрации работы созданного модуля используется следующая тестовая последовательность: формируется тактовый сигнал с периодом 20 нс, и на вход данных подается фиксированное значение 100_{10} . Тестовый модуль `pwm_tb` показан ниже:

```

module pwm_tb;

// Inputs
reg clk;
reg [7:0] d;

// Outputs
wire pwm;

// Instantiate the Unit Under Test (UUT)
pwm uut (
  .clk(clk),
  .d(d),
  .pwm(pwm)
);

initial begin
  // Initialize Inputs
  clk = 0;
  d = 100;
  forever clk = #10 ~clk;
end

endmodule

```

Для получения периода сигнала ШИМ, не равного целой степени двойки, следует обеспечить изменение счетчика в диапазоне от 0 до необходимого максимального значения.

6.8. Выводы по разделу

Для цифровых схем используется синхронный стиль проектирования:

- один тактовый сигнал для модуля;
- основной триггер – D-триггер (с разрешением счета);
- сброс – синхронный, предпочтительно с активным высоким уровнем.

Счетчик является основой для построения других схем, например, широтно-импульсного модулятора.

Контрольные вопросы:

1. Какие сигналы есть у D-триггера?
2. Какой сброс предпочтительнее для триггера – синхронный или асинхронный? Почему?
3. Сколько разрядов должен иметь счетчик, способный считать до 200?
4. Какое выражение на Verilog необходимо использовать, чтобы временно остановить счетчик?
5. Мощный транзистор необходимо переключать с частотой не менее 100 кГц. Требуется обеспечить погрешность не хуже 0.1%. Какую тактовую частоту необходимо подать на модуль ШИМ, управляющий таким транзистором?

7. РЕАЛИЗАЦИЯ НАКРИСТАЛЬНОЙ ПАМЯТИ И ВНЕШНИХ ИНТЕРФЕЙСОВ ПАМЯТИ

7.1. Содержание раздела

Память часто используется в проектах, как внутри микросхем, так и в качестве внешних устройств. Если необходимо подключить внешнюю микросхему памяти, в разрабатываемой схеме требуется добавить соответствующий интерфейс.

Интерфейсы памяти существенно различаются по сложности реализации. Кроме того, память с высокой частотой передачи данных требует выполнения ряда требований по проектированию печатной платы. Поэтому некоторые виды памяти желательно подключать с помощью готовых IP-ядер.

7.2. Характеристики памяти

Память представляет собой массив однотипных ячеек, хранящих данные определенного типа. Входом модуля памяти является номер ячейки, называемый также адресом. Поскольку адрес передается в двоичном виде, большинство блоков памяти имеет размер, равный степени двойки. Разрядность ячейки в принципе не имеет ограничений, однако обычно выбирается такой, чтобы удобно представлять форматы данных, используемые в вычислительной технике.

Естественной является разрядность данных, равная 8, 16 и 32. В то же время, встречаются устройства памяти, в которых данные кратны 9 разрядам. Это связано с тем, что одним из способов контроля целостности данных (то есть отсутствия ошибок при записи) является запись дополнительного *бита четности*. Он вычисляется путем выполнения операции xor над всеми 8 битами данных. Таким образом получается дополнительный 9 бит, который может быть записан вместе с основными 8 битами данных. Если один из битов окажется инвертирован, будет инвертирован и бит четности. Таким образом, можно выявлять наличие одиночных ошибок (если будет изменено два бита, бит четности останется таким же). Для этого при чтении вычисляется бит четности для 8 разрядов и сравнивается с прочитанным 9-м битом. Этот подход является простейшим, однако в силу простоты реализации используется достаточно широко.

Таким образом, главной характеристикой памяти является количество ячеек и их разрядность. Эти параметры однозначно определяют объем данных, который может хранить этот модуль памяти.

Другой важной характеристикой является время доступа к данным, находящимся в выбранной ячейке памяти. Это время может быть разным для чтения и записи данных. Кроме того, некоторые виды памяти требуют подготовительных операций для начала доступа к последовательно расположенным ячейкам.

По характеру выполняемых с ней операций память подразделяется на:

- постоянные запоминающие устройства (ПЗУ, также ROM – Read-Only Memory), которые хранят фиксированные данные, которые можно изменить с помощью специального оборудования;
- оперативные запоминающие устройства (ОЗУ, также RAM – Random Access Memory), допускающие изменение записанных данных.

С точки зрения технической реализации память также имеет смысл подразделять на энергозависимую (сохраняющую данные только при поданном питании) и энергонезависимую. Для ранних вариантов исполнения энергонезависимая память являлась практически синонимом ПЗУ, поскольку техническая реализация таких микросхем подразумевала хранение данных в ячейках с пережигаемыми перемычками, ультрафиолетовым стиранием и т.п. Все эти принципы исполнения обеспечивали сохранность данных при отключении питания, но и не позволяли изменять содержимое памяти без специального оборудования – например, память с ультрафиолетовым стиранием требовала, как следует из ее названия, источника УФ излучения для стирания данных и специального программатора. В настоящее время существуют устройства энергонезависимой памяти, которые допускают изменение содержимого без специального программатора. Например, flash-память, память с электрическим стиранием, память FRAM, другие перспективные типы памяти. Часть из них требует отдельного цикла стирания, а часть позволяет произвольно перезаписывать данные, как для микросхем ОЗУ.

Энергозависимая память подразделяется на статическую и динамическую. Для статической памяти достаточно подать питание на микросхему, чтобы записанные данные сохранялись. Для динамической памяти требуется производить специальную операцию *регенерации* данных, которая заключается в периодическом считывании содержимого всех ячеек, в процессе чего их состояние обновляется. Остановка процесса регенерации приводит к быстрой потере записанных данных вследствие утечки электрического заряда.

По способу организации интерфейса модули памяти подразделяются на модули с асинхронным или синхронным интерфейсом, а также с параллельным или последовательным доступом.

Примерная классификация памяти показана на рис. 7.1. В силу большого разнообразия способов классификации трудно привести исчерпывающую схему, поэтому показаны только основные разновидности.



Рисунок 7.1. Виды памяти

7.3. Описание памяти на поведенческом уровне. Описание памяти на структурном уровне

Память в языках описания аппаратуры может быть реализована двумя способами. Первым является описание на поведенческом уровне. При этом, как и в языках программирования, объявляется массив, для которого описываются операции чтения и записи (запись – для ОЗУ). При анализе такого текста синтезатор выявляет шаблон и сопоставляет его с теми ресурсами памяти, которые есть в используемой библиотеке компонентов.

Вторым способом является использование структурного описания памяти. В этом случае поведение памяти не описывается, а вместо этого в проект вставляется компонент, соответствующий компоненту памяти из имеющейся библиотеки компонентов. Этот способ менее гибок, поскольку привязан к конкретному компоненту, однако гарантирует применение указанного разработчиком модуля памяти, не оставляя это решение синтезатору, который в ряде случаев реализует неоптимальные варианты памяти.

Самым простым вариантом модуля памяти является постоянное запоминающее устройство с асинхронным интерфейсом. Такой модуль является фактически таблицей, содержащей определенное количество ячеек. Подача на вход адреса (номера) ячейки приводит к появлению на выходе данных, записанных в ячейке с этим адресом. Графическое изображение постоянного запоминающего устройства показано на рис. 7.2.

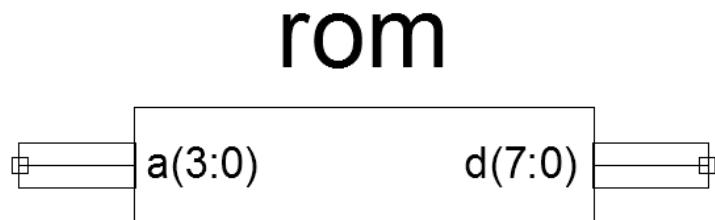


Рисунок 7.2. Постоянное запоминающее устройство с асинхронным интерфейсом

Постоянное запоминающее устройство может быть описано на Verilog с помощью оператора case.

```

module rom(
    input [3:0] a,
    output [7:0] d
);
reg [7:0] data;

always @ (a)
case (a)
    0 : data <= 1;
    1 : data <= 3;
    2 : data <= 4;
    default : data <= 0;
endcase
assign d = data;
endmodule
  
```

Значения 1, 3, 4 и 0, записанные в ячейках, приведены в качестве примера. В реальном модуле памяти данные, очевидно, могут быть другими. Для конкретного примера собственно модуль памяти скорее всего не будет реализован синтезатором, поскольку вместо этого появится 8 логических узлов на базе логических вентилей, причем каждый узел будет иметь 4 входа. Нетрудно убедиться, что старшие разряды данных в показанном примере всегда равны нулю, поэтому для них не нужно анализировать комбинацию на адресных входах, достаточно просто подать 0 на эти выходы. Подобные преобразования достаточно характерны для синтеза памяти, описанной на поведенческом уровне, причем изменение данных повлечет за собой и изменение схемы. Поэтому такой подход не следует применять в качестве временного решения для оценки размера схемы.

Следующим примером является оперативное запоминающее устройство с синхронным интерфейсом. Его графическое изображение показано на рис. 7.3. Показанные разрядности сигналов условны и относятся только к конкретному примеру. Блоки могут иметь и другие разрядности сигналов.

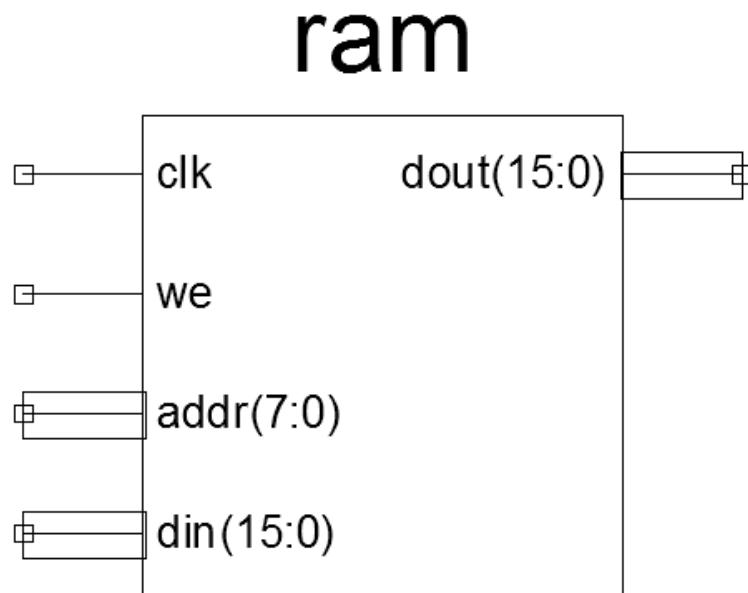


Рисунок 7.3. Оперативное запоминающее устройство с синхронным интерфейсом

Работу такого блока можно достаточно просто представить, если обратиться к общим правилам построения синхронных схем.

«По фронту тактового сигнала данные с входа **din записываются в ячейку с адресом **addr**, если сигнал **we** равен 1».**

Если сигнал we (Write Enable, «разрешение записи») равен 0, вместо записи выполняется чтение. На выходе dout появляются данные из ячейки, адрес которой равен addr.

Примеры временных диаграмм работы синхронной памяти показаны на рис. 7.4. Значение XXXX соответствует ситуации «не имеет значения», т.е. данные на этом входе в показанные моменты времени не используются. Также на рисунке можно видеть иногда используемый сигнал EN (Enable, «разрешение»), который используется в качестве глобального сигнала разрешения работы.

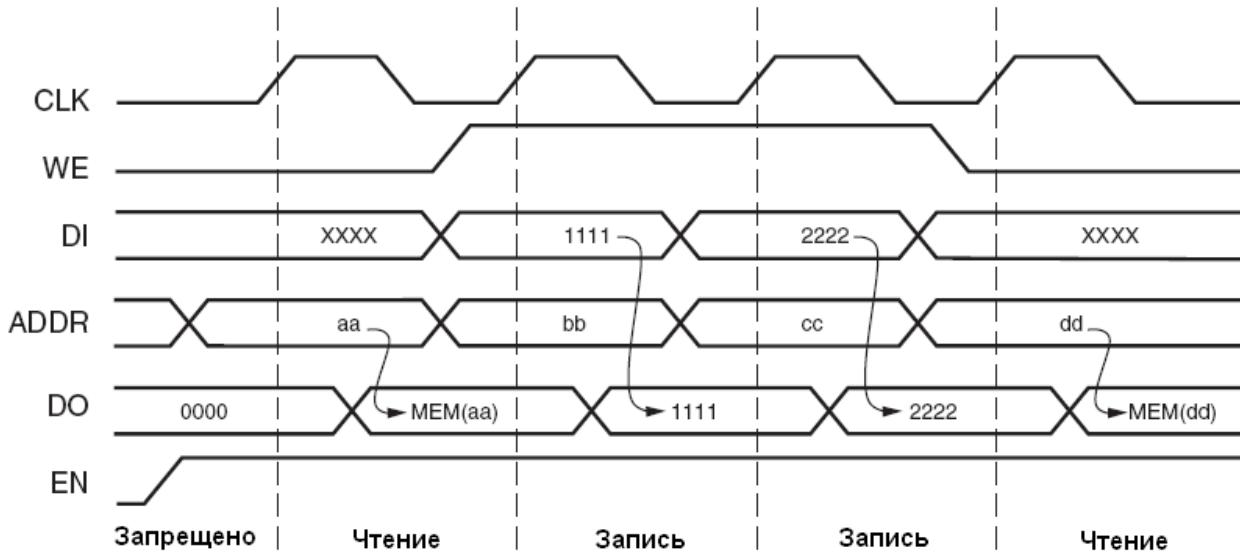


Рисунок 7.4. Диаграммы работы оперативного запоминающего устройства с синхронным интерфейсом

На диаграмме также видно, что данные, которые записываются в память, по фронту тактового сигнала появляются и на выходе dout. Это не единственное возможное поведение памяти. Можно упомянуть три возможных сценария работы при записи данных в определенную ячейку:

1. «Чтение после записи» (Read after Write) – сценарий, показанный на рис. 7.4, когда записываемые данные одновременно появляются и на выходе.

2. «Чтение перед записью» (Read before Write) – старое значение из ячейки данных появляется на выходе, в то время как в ячейку записываются данные din.

3. «Нет чтения» (No Read on Write) – значение выхода данных не изменяется. В этом режиме память потребляет меньше мощности, поскольку нет перезаписи данных на выходе.

Требуемый режим работы будет автоматически синтезирован на основе поведенческого описания модуля памяти. Если используется структурное представление, режим работы обычно задается в конструкции используемого компонента и в ряде случаев может регулироваться.

7.4. Ресурсы для реализации памяти в ПЛИС

В современных ПЛИС память представлена несколькими вариантами компонентов. Компания Xilinx обеспечивает для логических генераторов специальный режим распределенной памяти (*distributed memory*). Поскольку таблица истинности – это просто небольшой блок статической памяти, есть возможность использовать его и в проектах пользователя, если предоставить доступ к интерфейсу, позволяющему записывать в эту память собственные данные (а не только часть конфигурации ПЛИС). Графическое изображение компонента распределенной памяти показано на рис. 7.5.

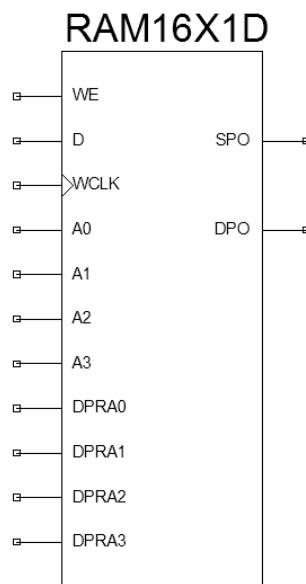


Рисунок 7.5. Компонент распределенной памяти в ПЛИС Xilinx

У показанного компонента есть два порта. Под портом в данном случае понимается независимый набор интерфейсных сигналов, позволяющий получать доступ к ячейкам данных. На рис. 7.5 сигналы второго порта начинаются с символов DP (от «Dual Port»). Это сокращенный набор сигналов – DPRA0 .. DPRA3 соответствуют адресу (Dual Port Read Address), а соответствующее значение при чтении появляется на выходе DPO (Dual Port Output).

Первый порт имеет уже показанный выше для статической памяти набор сигналов, включая тактовый сигнал. Можно обратить внимание, что для второго порта тактового сигнала нет – этот порт работает в асинхронном режиме и данные появляются на выходе «сразу» (т.е. по факту с некоторой задержкой, обусловленной распространением сигнала) после изменения адреса. Временные диаграммы работы компонента распределенной памяти показаны на рис. 7.6.

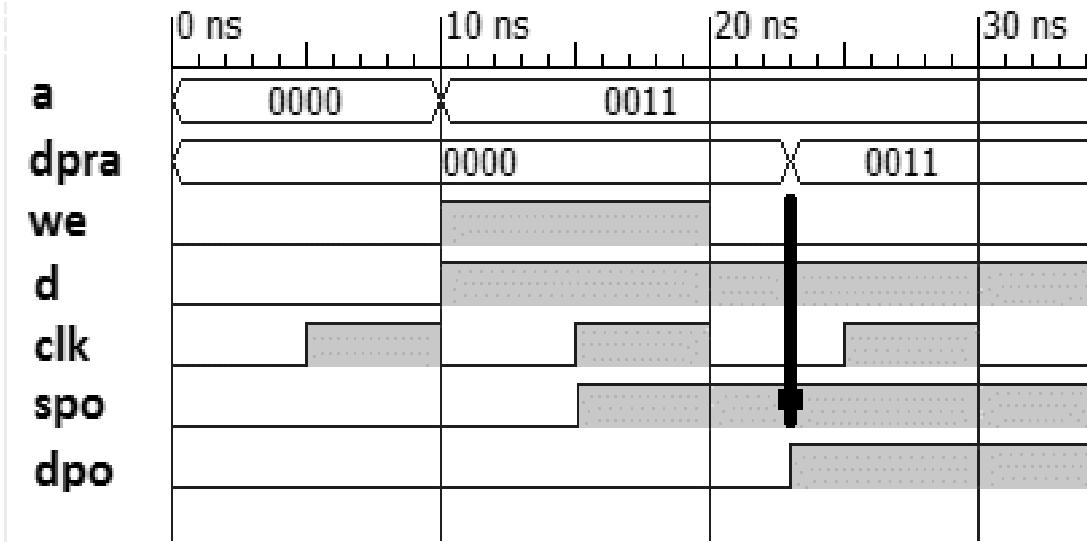


Рисунок 7.6. Временные диаграммы работы компонента распределенной памяти

На рисунке видно, что в момент времени 15 нс в ячейку с адресом 0011 записывается 1. При этом сигнал we также равен 1, что соответствует операции «запись». Записанная единица сразу появляется на выходе данных первого порта spo. Выход данных второго порта пока не изменяется, поскольку его адрес равен 0000. В момент времени 22 нс адрес второго порта также устанавливается в 0011, и на выходе dpo также появляется 1.

Режим, при котором один порт может использоваться и для чтения, и для записи, а второй – только для чтения, называют «простым двупортовым» (simple dual port) или «псевдо двупортовым» (pseudo dual port). Для распределенной памяти асинхронное чтение является следствием того, что на самом деле эта память имитирует работу обычных логических элементов.

Режим распределенной памяти обеспечивают не все производители ПЛИС, поскольку он требует добавления трассировочных линий и немного замедляет работу памяти. Кроме этого, не все логические ячейки в матрице ресурсов могут конфигурироваться как распределенная память. В целом такой режим работы не рекомендуется для создания крупных блоков памяти, поскольку большая часть ресурсов ячейки оказывается незадействованной.

Более типичным компонентом памяти является аппаратно реализованный блок статической памяти. Его графическое изображение показано на рис. 7.7.

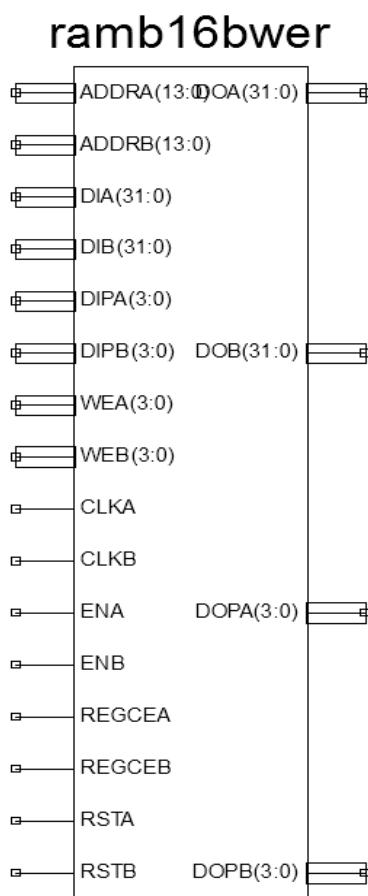


Рисунок 7.7. Компонент блочной памяти в ПЛИС Xilinx

Блок памяти имеет следующие сигналы:

- `addra`, `addrb` – адреса портов А и В соответственно;
- `dia`, `dib` – данные для записи для портов А и В (32 бита);
- `dipa`, `dipb` – дополнительные данные для записи (4 бита);
- `wea`, `web` – входы разрешения записи (побайтно);
- `clka`, `clkb` – тактовые сигналы для портов;
- `ena`, `enb` – входы разрешения работы блока памяти (при чтении состояние выходов не обновляется, если нет разрешающего сигнала);
- `regcea`, `regceb` – разрешение работы выходных регистров;
- `rsta`, `rstb` – сброс выходных регистров (не влияет на содержимое массивов памяти);
- `doa`, `dob` – выходы данных для портов А и В (32 бита);
- `dopa`, `dopb` – дополнительные выходы данных для портов А и В (4 бита).

Физически размещенные в FPGA блоки памяти являются 18- или 36-битными (в зависимости от семейства микросхем). Такая разрядность позволяет реализовывать схемы контроля четности, когда каждые 8 бит имеют дополнительный 9-й бит для хранения бита четности. Соответственно, каждые

16 бит имеют 2 дополнительных бита четности, а 32 – 4 бита. Для удобства работы с дополнительными битами в графическом представлении модуля они выделены в отдельные шины dipa, dipb, dopa, dopb.

Дополнительные биты не являются автоматически заполняемыми и представляют собой разряды, доступные для записи в них произвольных значений. Разработчик может выбирать требуемую ему разрядность, включая 9, 18 или 36 бит.

Блочная память является эффективным аппаратным ресурсом, который допускает работу на *системной тактовой частоте*. Это максимальная тактовая частота, на которой теоретически мог бы работать проект, если бы он не содержал цепей комбинационной логики, проходящих более чем через один логический генератор, и слишком длинных трассировочных цепей. На практике работа на системной тактовой частоте оказывается доступной только для относительно компактных фрагментов проекта, и важно, что блочная память не ухудшает эти параметры. Следует также иметь в виду, что удельная стоимость блочной памяти ниже, чем памяти того же объема, реализованной на логических ячейках. Сложно указать точную границу объема, превышение которой делает блочную память однозначно более эффективным решением, однако можно ориентироваться на технические характеристики разных устройств (16x1 или 64x1 бит в логическом генераторе, 1024x18 бит в блоке памяти), а конкретный способ реализации памяти выбирать, исходя из доступных ресурсов проекта и требуемых технических характеристик.

Быстрое создание модуля памяти возможно с помощью генератора IP-ядер. На рис. 7.8 показан интерфейс генератора IP-ядер в режиме создания блочной памяти, а на рис. 7.9 – для создания компонентов на основе распределенной памяти. Генераторы обладают широкими возможностями настройки (в рамках допустимого для соответствующих компонентов памяти) и создают описание, гарантированно использующее указанные разработчиком ресурсы.

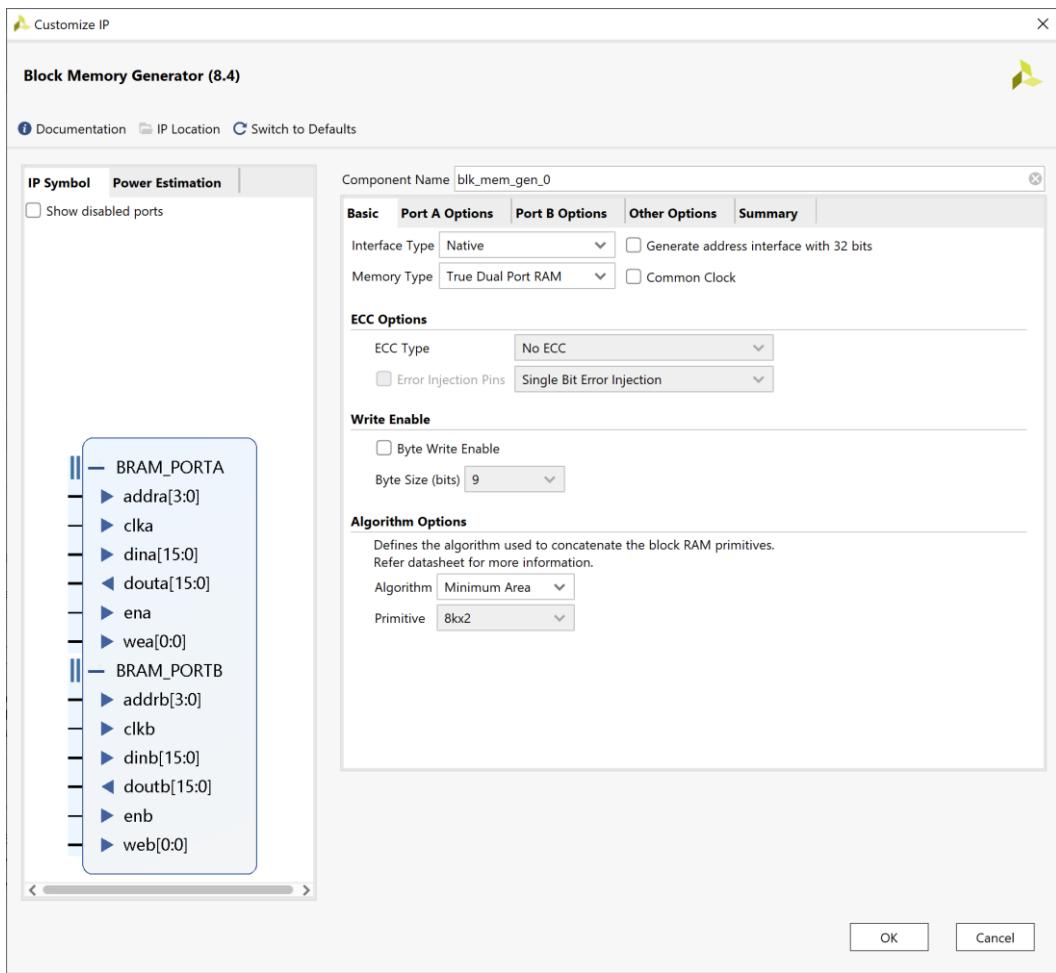


Рисунок 7.8. Интерфейс генератора IP-ядер памяти в САПР Xilinx Vivado

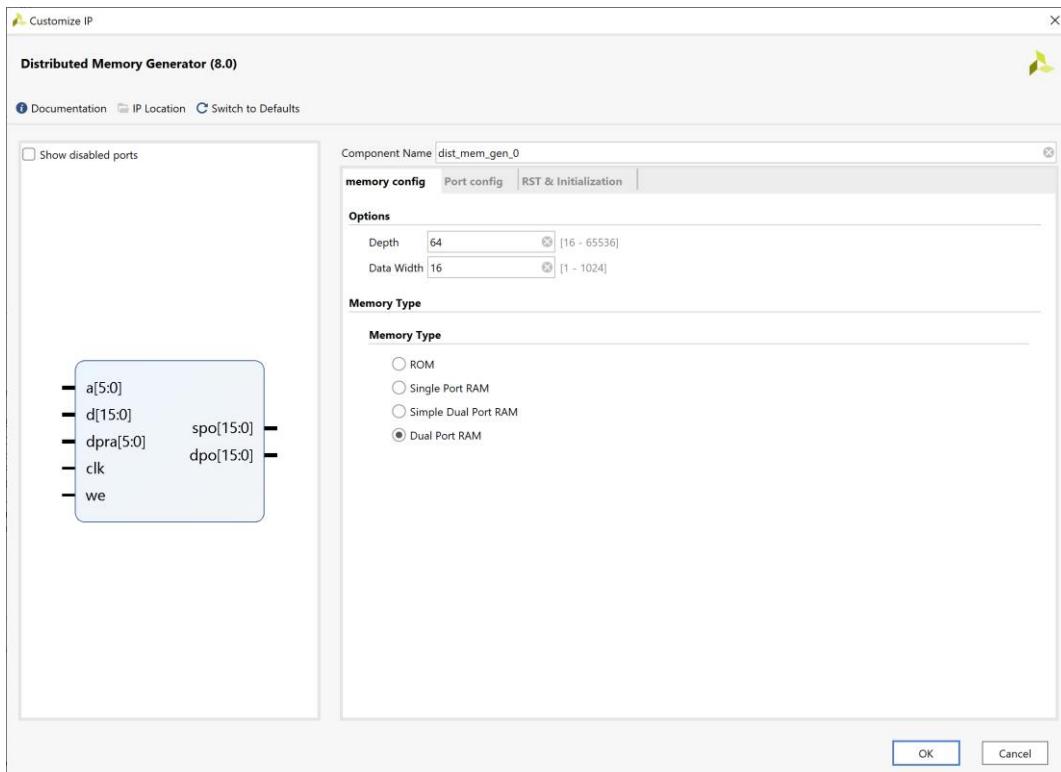


Рисунок 7.9. Интерфейс генератора IP-ядер памяти в САПР Xilinx Vivado

7.5. Реализация памяти в СБИС. Memory compiler

Блочная память ПЛИС представляет собой аппаратный компонент. Это означает, что такая память не содержит внутри трассировочных линий или других элементов, которые позволяли бы конфигурировать память в процессе работы, изменяя ее схему. Для схемы, построенной на базе логических ячеек, при переходе к аппаратной реализации можно ожидать существенного улучшения всех характеристик – из-за отказа от схем конфигурирования можно уменьшить размер, повысить тактовую частоту и уменьшить энергопотребление. Однако блочная память ПЛИС не содержит дополнительных ресурсов и не следует ожидать существенного улучшения ее характеристик. Более того, при разработке СБИС необходимо разработать и модули памяти.

Статическая память в СБИС организована в виде двумерного массива, как показано на рис. 7.10. Основой этого массива является так называемая 6T Cell (шеститранзисторная ячейка), которая хранит один бит данных.

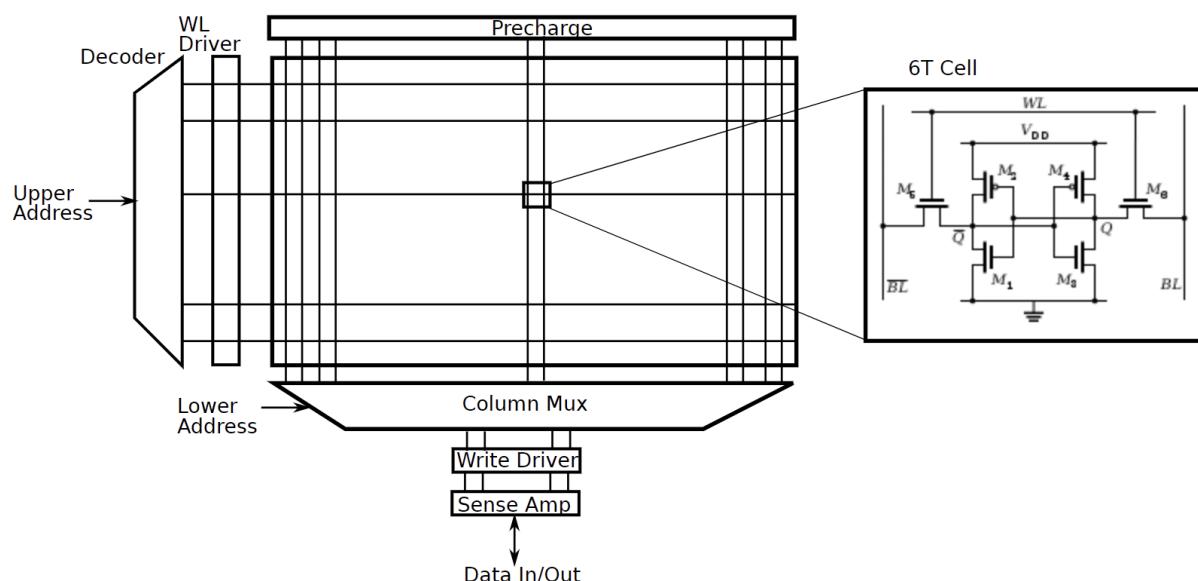


Рисунок 7.10. Организация статической памяти в СБИС

Чтобы избежать хаотического расположения транзисторов, расстановка ячеек и управляющих компонентов производится с помощью специальных генераторов – «компиляторов памяти» (memory compiler). Такие компиляторы существенно привязаны к используемому технологическому процессу и их выбор следует проводить, зная фабрику и особенности конкретного технологического процесса, по которому планируется производить СБИС.

Маршрут проектирования компонентов памяти показан на рис. 7.11.

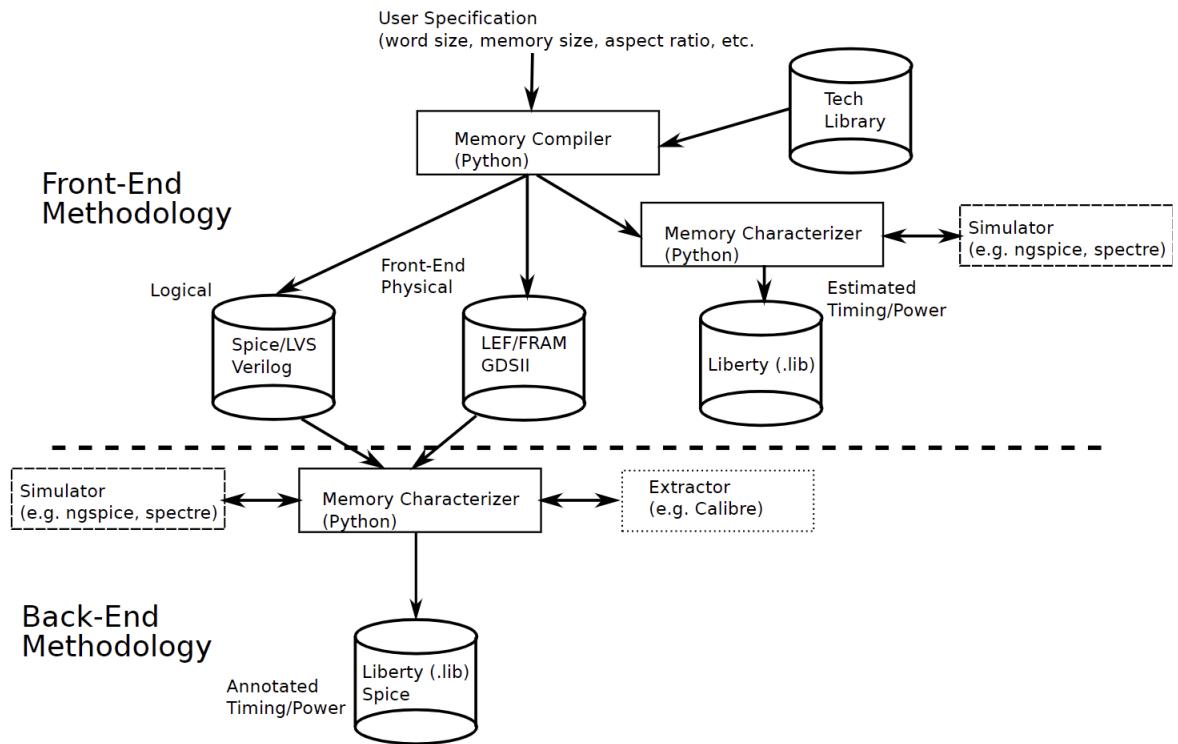


Рисунок 7.11. Маршрут проектирования компонентов памяти

При разработке макета СБИС на базе ПЛИС следует выделять модули памяти в отдельные субмодули, которые будут иметь внешний интерфейс, подходящий и для блока памяти в СБИС, и для блока памяти в ПЛИС. В процессе макетирования внутри такого субмодуля можно использовать поведенческое описание или ссылку на компонент ПЛИС. При передаче проекта для изготовления СБИС необходимо передать разработчикам топологии перечень субмодулей памяти и их характеристики.

7.6. Интерфейсы внешней памяти

Ряд компонентов памяти часто располагаются снаружи, в виде отдельных микросхем. Это может быть связано с меньшей стоимостью специализированной микросхемы памяти по сравнению с размещением памяти такого же объема внутри ПЛИС или СБИС. Кроме того, только статическая память строится из стандартных цифровых элементов (триггеров). Динамическая память, флэш-память и другие виды памяти изготавливаются иначе, поэтому их сложно или просто невозможно разместить на том же полупроводниковом кристалле, что и цифровые компоненты.

Из-за этого большое внимание при построении вычислительных систем уделяется интерфейсам внешних компонентов памяти. Примером является микросхема с асинхронным интерфейсом, показанная на рис. 7.12.

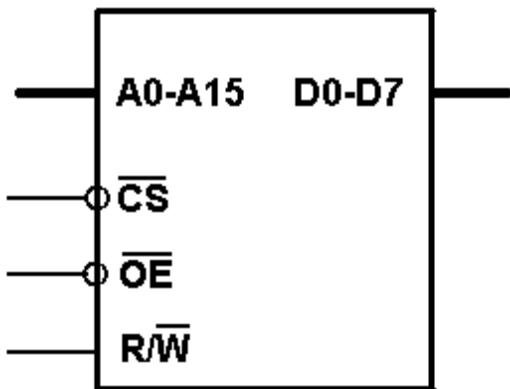


Рисунок 7.12. Графическое представление микросхемы памяти с асинхронным интерфейсом

Можно заметить, что с таким интерфейсом выпускаются и микросхемы статической памяти, и микросхемы флэш-памяти. Для флэш-памяти перед записью непосредственно данных требуется передача в микросхему специальных кодов, предваряющих такую запись. Последовательность операций для записи байта данных в определенную ячейку определяется документацией производителя, хотя имеет место унификация базовых операций с флэш-памятью.

На рис. 7.13 показаны временные диаграммы чтения из памяти с асинхронным интерфейсом. Интерфейс имеет три управляющих сигнала: $\sim\text{CS}$ – «выбор кристалла» (Chip Select), на котором должен присутствовать низкий логический уровень для того, чтобы микросхема воспринимала интерфейсные сигналы. Отсутствие активного уровня (можно еще раз подчеркнуть, что в данном случае активным является ноль) позволяет быстро остановить работу микросхемы. Например, если система памяти построена из множества микросхем, почти все сигналы можно обобщить. Однако в таком случае на всех микросхемах будут изменяться адреса, данные и сигналы записи, что привело бы к копированию записи во все микросхемы. Для того, чтобы запись состоялась только в одну из них, используется сигнал Chip Select. Если он неактивен, все остальные действия микросхемой игнорируются. Сигнал CS можно встретить во многих микросхемах памяти, интерфейсов и т.д.

В отличие от CS, который имеет широкое применение, сигнал $\sim\text{OE}$ (Output Enable, «разрешение выхода») разрешает подачу сигналов на выходы данных. Он также полезен при объединении множества микросхем. Если просто объединить выходы данных двух микросхем, возникнет электрический конфликт: две микросхемы могут содержать разные данные в своих ячейках с одинаковым адресом, поэтому на объединенных линиях данных будет неопределенное состояние. Чтобы избежать этого, необходимо разрешить подключение выходов только какой-то одной микросхемы. Для этого ей подается низкий уровень сигнала $\sim\text{OE}$, и ее выходы

становятся активными. Если на $\sim\text{OE}$ подан сигнал логической 1, выходы отключаются – переводятся в состояние *высокого импеданса*, которое также называется «третьим состоянием», или «Z-состоянием». Можно представить это так, что между выходом микросхемы и проводником включается резистор с сопротивлением несколько МОм. Внутри микросхемы по-прежнему генерируется какой-то логический уровень, но это напряжение уже существенно меньше влияет на внешний проводник из-за большого сопротивления, включенного в эту цепь. Термин «импеданс» означает «полное сопротивление» и включает в себя как активную составляющую, так и реактивную (зависящую от частоты сигнала).

Сигнал R/W означает Read/Write (чтение/запись). Как следует из его обозначения, логическая 1 соответствует состоянию «чтение», а логический 0 – записи.

Можно еще раз отметить, что низкий активный уровень обозначается на схемах несколькими способами: кружочком на точке входа сигнала в обозначение микросхемы, горизонтальной чертой над именем сигнала, а также символами \sim # n перед именем сигнала, если можно использовать только текст. Т.е. показать, что сигнал CS имеет низкий активный уровень, в тексте можно как $\sim\text{CS}$, #CS, nCS.

На рис. 7.13 можно отследить процесс чтения. Управляющие сигналы приводятся в активное состояние ($\text{CS}=0$, $\text{OE}=0$, $\text{R/W}=1$, т.е. «микросхема выбрана», «выход разрешен», «чтение»). Тогда при подаче на входы адреса нужной комбинации через некоторое время t на выходах данных появляется соответствующее значение, записанное в этой ячейке.

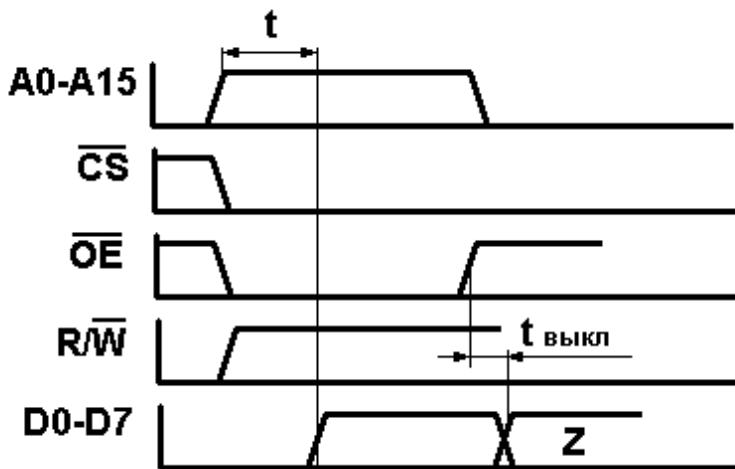


Рисунок 7.13. Временные диаграммы чтения из памяти с асинхронным интерфейсом

На рис. 7.13 можно также заметить, что после отключения сигнала OE через некоторое время выходы переходят в состояние высокого импеданса.

Нетрудно заметить, что скорость работы с памятью определяется временем чтения данных, т.е. задержкой от подачи адреса до появления соответствующих данных. Это время указывается производителем микросхемы.

В настоящее время микросхемы памяти с асинхронным интерфейсом имеют несколько групп задержек. Для асинхронной памяти это 150-200 нс для медленных вариантов, 55-70-90 нс для большой группы со средними характеристиками и 8-10-12 нс для памяти, которую можно характеризовать как высокоскоростная. При этом нужно учитывать, что распространение сигналов по печатным проводникам и внутри периферийных блоков ПЛИС или СБИС может также добавить 5-10 нс задержки.

Микросхемы флэш-памяти имеют параметр задержки в диапазоне от 55 до 120 нс в основной группе.

На рис. 7.14 показана временная диаграмма записи в память с асинхронным интерфейсом. Она отличается от чтения тем, что момент записи определяется не самим сигналом записи, а его возвратом в неактивное состояние. Иными словами, активизация управляющих сигналов еще не означает запись. В данном случае подготовка процесса происходит как CS=0 (микросхема выбрана), OE=1 (выход неактивен, поскольку теперь данные являются входом) и RW=0 (запись). Это состояние должно оставаться в течение времени t_{wr} («время записи»), которое также специфицируется производителем (обычно оно равно или сопоставимо с временем чтения). Как только один из управляющих сигналов перейдет в неактивное состояние, происходит запись. Чаще всего используется сигнал RW, который на рис. 7.14 возвращается в состояние «чтение». Такая запись называется в документации «WR-controlled». Обычно можно использовать и сигнал OE или CS, однако такие режимы необходимо проверить в документации на микросхему.

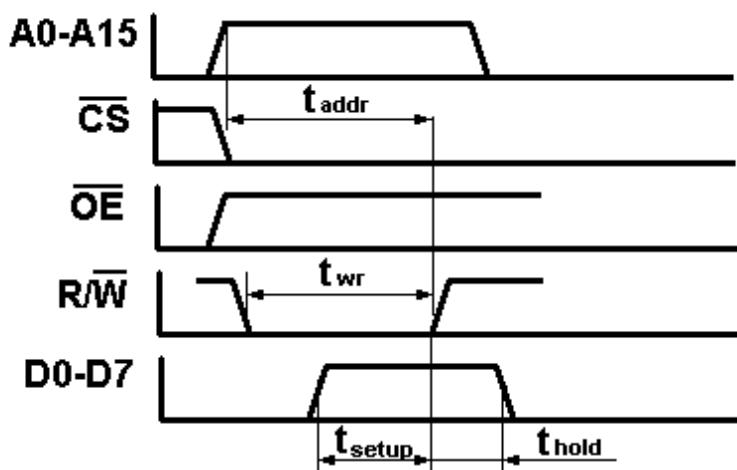


Рисунок 7.14. Временные диаграммы записи в память с асинхронным интерфейсом

Также на рис. 7.14 показаны два интервала времени, которые активно используются для проверки временных соотношений между сигналами. Это t_{setup}

«время установки» и t_{hold} («время удержания»). Эти интервалы времени измеряются от момента, в который производится операция (в данном случае это операция записи). Они означают, что данные должны быть на выходе не просто в момент записи, а поданы предварительно, не менее чем за t_{setup} . Аналогично, после проведения записи данные нельзя убирать «мгновенно» в строгом значении этого слова, а необходимо удерживать в течение времени t_{hold} .

Разработка интерфейса для памяти с асинхронным интерфейсом не представляет большой сложности. Фактически это набор выходных регистров для тех сигналов, которые являются входами у микросхемы памяти, и двунаправленная шина для данных. При этом сигнал OE, подаваемый на внешнюю микросхему, необходимо использовать для переключения направления передачи данных внутри интерфейса.

В вычислительной технике очень широко распространена динамическая память. В основном компьютеры, смартфоны и планшеты в качестве объема памяти указывают именно объем динамической памяти, которая существенно, на несколько порядков, превосходит по объему статическую память.

При большом объеме и относительно низкой стоимости динамическая память отличается существенно более сложным интерфейсом. Это связано с ее физической организацией, которая основана на двумерной матрице конденсаторных ячеек памяти. Структурная схема динамической памяти показана на рис. 7.15.

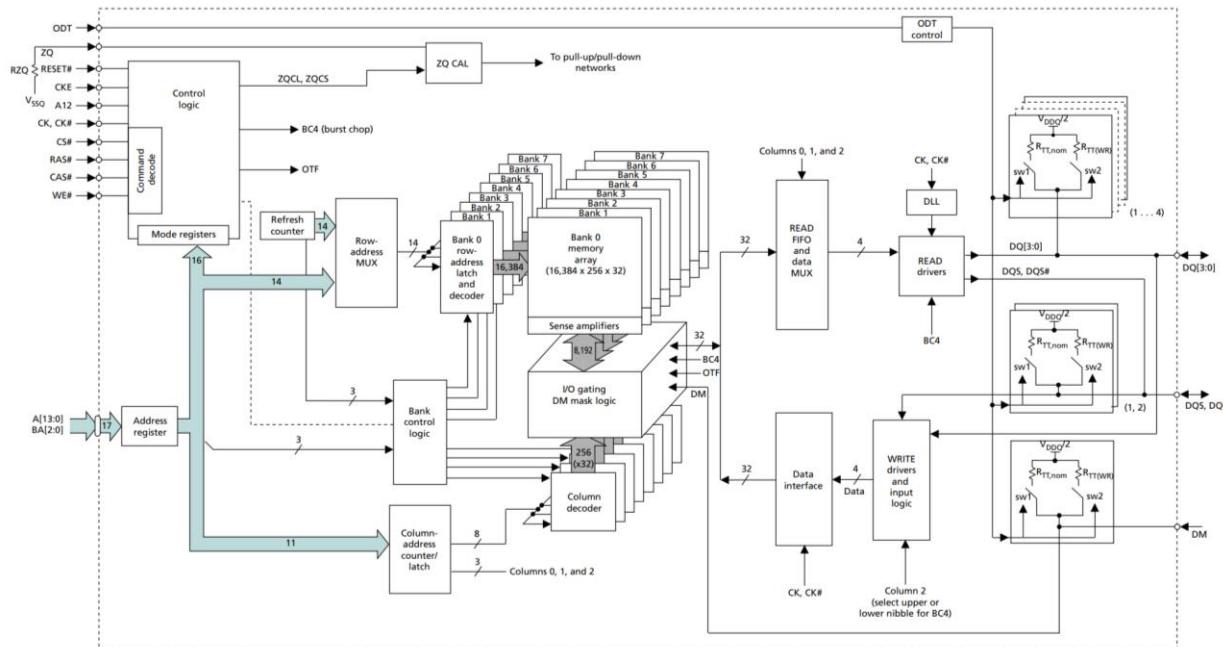


Рисунок 7.15. Структурная схема микросхемы динамической памяти

Для чтения из определенной ячейки необходимо активизировать два компонента адреса – колонку (Column) и ряд (Row). Таких двумерных матриц внутри микросхемы может быть несколько, в терминах памяти они называются банками. Таким образом, требуется указать не только Column, Row, но и Bank. Однако для совместимости адрес ячейки представляется в виде двух компонентов, которые обозначаются как Column и Row.

Соответственно, разрядность внешнего адреса у динамической памяти в два раза меньше, чем требуемая для адресации всех ячеек. Подача адреса сопровождается сигналов CAS (Column Address Strobe) или RAS (Row Address Strobe). В зависимости от того, какой из сигналов активен, внутренний контроллер обновляет сведения о соответствующем компоненте адреса.

Такой подход не позволяет рассматривать интерфейс динамической памяти как обычный доступ к ячейке по заданному адресу. Фактически обмен данными ведется с внутренним контроллером микросхемы памяти, который принимает команды и компоненты адреса. Пример временных диаграмм работы с динамической памятью показан на рис. 7.16. Приведенный рисунок демонстрирует, что протоколы работы с памятью относительно сложны.

Для повышения производительности обмена данными при передаче используется не только фронт тактового сигнала, но и спад. Это не вполне обычная практика, но она является традиционной для динамической памяти на протяжении последних десятилетий. Если рассматривать полное обозначение динамической памяти, оно выглядит как SDRAM – Synchronous Dynamic Random Access Memory. Ранние варианты обозначались как SDR SDRAM, чтобы отличить их от более поздних DDR SDRAM. Соответственно, SDR означает Single Data Rate, а DDR – Double Data Rate. Имеется в виду, что память типа SDR производит обмен данными только по фронту тактового сигнала, а DDR – и по фронту, и по спаду. Характеристика скорости памяти обозначает суммарное количество фронтов и спадов. Например, DDR3200 означает, что тактовая частота составляет 1600 МГц, и за секунду происходит 3200 млн. транзакций (т.е. событий «фронт или спад»).

Современные микросхемы динамической памяти часто обозначают просто как DDR, с указанием номера версии интерфейса. Например, в настоящее время в основном происходит переход от DDR3 к DDR4.

Switching Waveforms

Figure 5. Read/Write Timing [24, 25, 26]

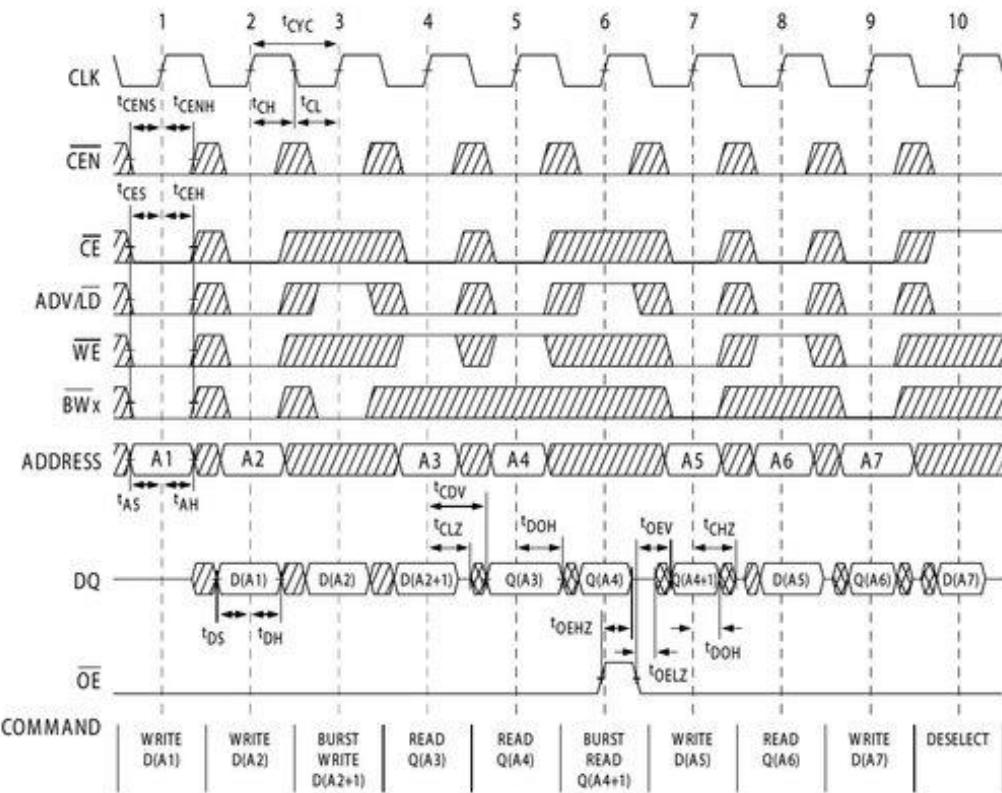


Рисунок 7.16. Пример временных диаграмм интерфейса динамической памяти

Важной особенностью динамической памяти является то, что пиковые характеристики производительности достигаются только в определенных сценариях работы, при доступе к последовательно расположенным ячейкам. Такой режим обозначается как Burst (например, Burst Read на рис. 7.16). Если требуется переход к другой колонке данных, необходимо подать соответствующий компонент адреса, сопроводив его сигналом CAS. Однако просто записать адрес недостаточно, необходимо дождаться, пока контроллер переключит внутренние коммутаторы на эту колонку. Это происходит в течение количества тактов, обозначаемого в документации на микросхему как CAS Latency (CL). Если требуется произвести следующую операцию с совершенно другой ячейкой памяти, время доступа будет существенно больше. На практике вместо одного такта доступ к данным может потребовать 20-30 тактов, и это количество имеет тенденцию к росту по мере повышения тактовой частоты обмена данными с контроллером памяти. Поэтому динамическую память обычно используют вместе со статической, которая при формально меньшей тактовой частоте работы не накладывает ограничений на порядок изменения адресов.

Таким образом, можно видеть, что организация работы с динамической памятью требует поддержки достаточно сложных протоколов обмена данными.

Другой важной проблемой является организация высокоскоростного электрического интерфейса. Его реализация осложняется тем, что в процессе работы задержки сигналов в микросхеме памяти изменяются, и требуется постоянная подстройка фазы тактового сигнала в контроллере. Микросхема памяти генерирует вспомогательные сигналы DQS (Data Strobe), которые являются ориентирами для выравнивания фазы. Изучение этого вопроса является длительным и сложным процессом, и реалистичным сценарием использования динамической памяти является генерация IP-ядра контроллера. Интерфейс IP-генератора показан на рис. 7.17 и рис. 7.18.

Настройка IP-ядра не ограничивается двумя вкладками. Перечень окон можно видеть на инструментальной панели слева на рис. 7.17, 7.18.

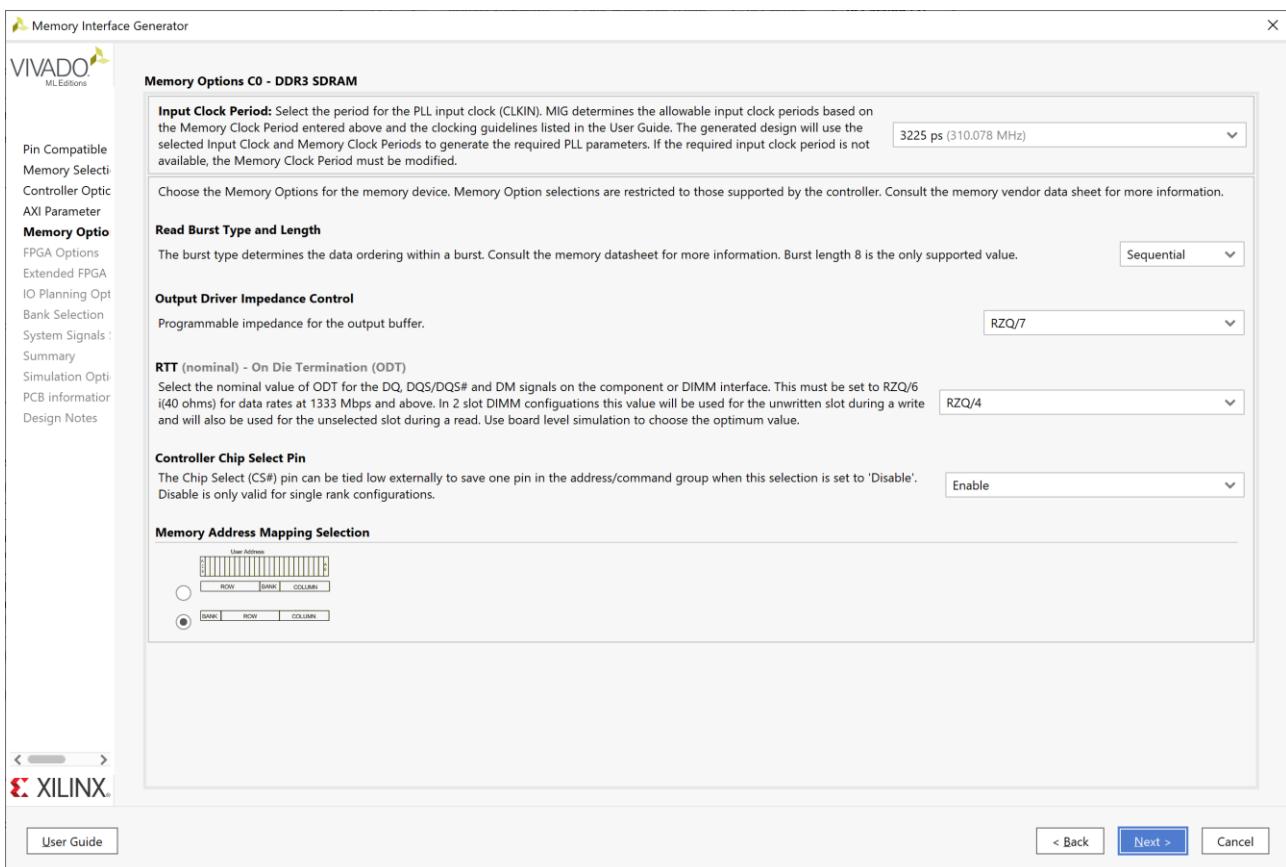


Рисунок 7.17. Интерфейс генератора IP-ядра контроллера динамической памяти

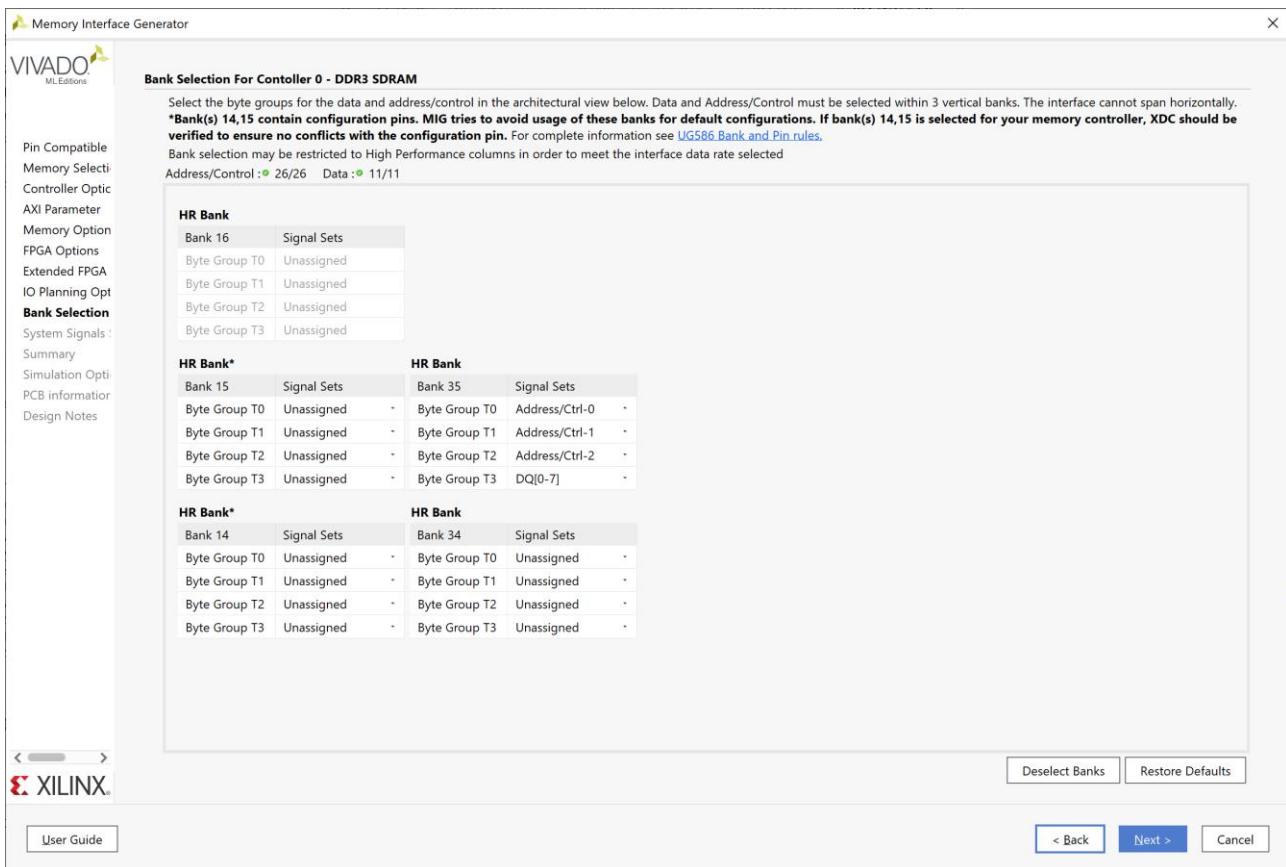


Рисунок 7.18. Интерфейс генератора IP-ядра контроллера динамической памяти

При планировании использования динамической памяти в вычислительной системе необходимо сопоставить сложность настройки контроллера памяти и достигаемые характеристики. Большой объем, измеряемый гигабайтами, является очевидным преимуществом динамической памяти, однако необходимо учитывать реальную производительность при доступе к данным.

На рис. 7.19 и 7.20 показаны сведения о латентности (задержке доступа в тактах) к данным для динамической памяти разных поколений. Видно, что рост тактовой частоты сопровождается и ростом латентности, как общей (True Latency), так и одного из важных компонентов – CAS Latency. Иными словами, если пытаться работать с динамической памятью, каждый раз инициируя обращение к ячейке заново, до появления данных на выходе пройдет время «True Latency». В таблице видно, что с 24 нс оно уменьшилось всего до 13,5 нс, тогда как частота обмена с контроллером за это время выросла более чем в 26 раз. Поэтому на практике работа с динамической памятью должна организовываться в виде обращений к последовательно расположенным ячейкам.

SPEED VS. LATENCY AS MEMORY TECHNOLOGY HAS MATURED (INDUSTRY STANDARDS)				
TECHNOLOGY	MODULE SPEED (MT/s)	CLOCK CYCLE TIME (ns)	CAS LATENCY (CL)	TRUE LATENCY (ns)
SDR	100	8.00	3	24.00
SDR	133	7.50	3	22.50
DDR	335	6.00	2.5	15.00
DDR	400	5.00	3	15.00
DDR2	667	3.00	5	15.00
DDR2	800	2.50	6	15.00
DDR3	1333	1.50	9	13.50
DDR3	1600	1.25	11	13.75
DDR4	1866	1.07	13	13.93
DDR4	2133	0.94	15	14.06
DDR4	2400	0.83	17	14.17
DDR4	2666	0.75	18	13.50

Рисунок 7.19. Латентность доступа к данным в некоторых типах динамической памяти

При переходе к следующему поколению динамической памяти также происходит и увеличение латентности на сопоставимой частоте обмена. На рис. 7.20 можно видеть сравнение CAS Latency для DDR3 и DDR4, некоторые модификации которых имеют одинаковую тактовую частоту. При этом латентность выше для DDR4.

Memory clock	DDR3 CAS latency	DDR4 CAS latency
1600 MHz	10	
1866 MHz	11	
2133 MHz	11	15
2400 MHz	11	15
2666 MHz	11	15
2800 MHz	12	16
3000 MHz		16
3200 MHz		16
3333 MHz		16
3466 MHz		18
3600 MHz		18

Рисунок 7.20. Латентность доступа к данным в некоторых типах динамической памяти

Практические эффекты от сочетания статической и динамической памяти иллюстрирует рис. 7.21. На нем показаны результаты исследования производительности компьютеров на базе процессоров с архитектурой x86. В этих процессорах имеется несколько уровней кэш-памяти, которая представляет собой синхронную статическую память. На графиках можно отследить, как при переходе через границу 32 кб график латентности делает скачок. Это происходит потому, что 32 кб данных помещались в кэш-память первого уровня, имеющую минимальную латентность, но и небольшой размер. Следующий скачок соответствует переходу от кэш-памяти второго уровня к кэш-памяти третьего уровня. Соответственно, латентность с 12-15 тактов увеличивается до 20-22. Наконец, при переходе к динамической памяти латентность увеличивается существенно, что зависит и от типа процессора, и от поколения динамической памяти (самое большое значение соответствует DDR4).

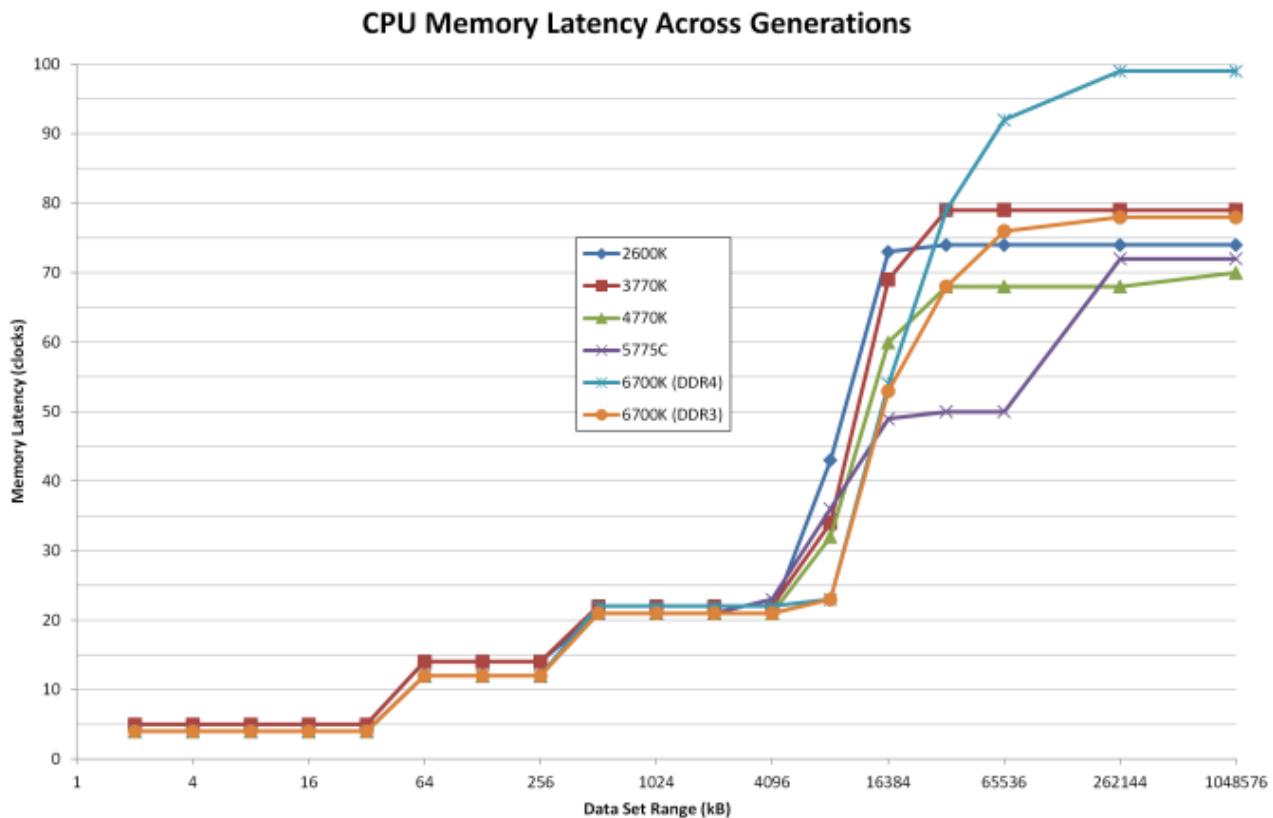


Рисунок 7.21. Зависимость латентности доступа к данным от объема массива данных для некоторых типов процессоров x86

Промежуточным решением является PSRAM, Pseudo Static RAM («псевдостатическая память»). Такие решения выпускаются, например, компанией ISSI. Это массив ячеек динамической памяти, контроллер которого имитирует поведение статической памяти. Подключение такого устройства существенно

проще, однако объем и быстродействие в имеющихся решениях хуже, чем у динамической памяти.

Микросхемы постоянных запоминающих устройств часто выпускаются с интерфейсом SPI. Подробно он будет рассмотрен в главе 10. Интерфейс использует всего 4 линии для подключения (тактовый сигнал CLK, выбор микросхемы CS, однобитные сигналы для обмена данными DI, DO). Пример графического изображения микросхемы флэш-памяти показан на рис. 7.22. Такие микросхемы часто используются для хранения конфигурации ПЛИС или расширения встроенной флэш-памяти микроконтроллеров. Также с интерфейсом SPI выпускаются ПЗУ типа EEPROM («с электрическим стиранием»), которые отличаются от флэш-памяти повышенным ресурсом циклов стирания-перезаписи. Некоторые менее распространенные типы памяти (FRAM, MRAM) часто выпускаются с интерфейсом SPI для упрощения их применения, поскольку такой интерфейс обычно оказывается хорошо известен разработчикам. В ряду случаев производители новых типов памяти полностью воспроизводят не только интерфейс, но и внутренние команды флэш-памяти.

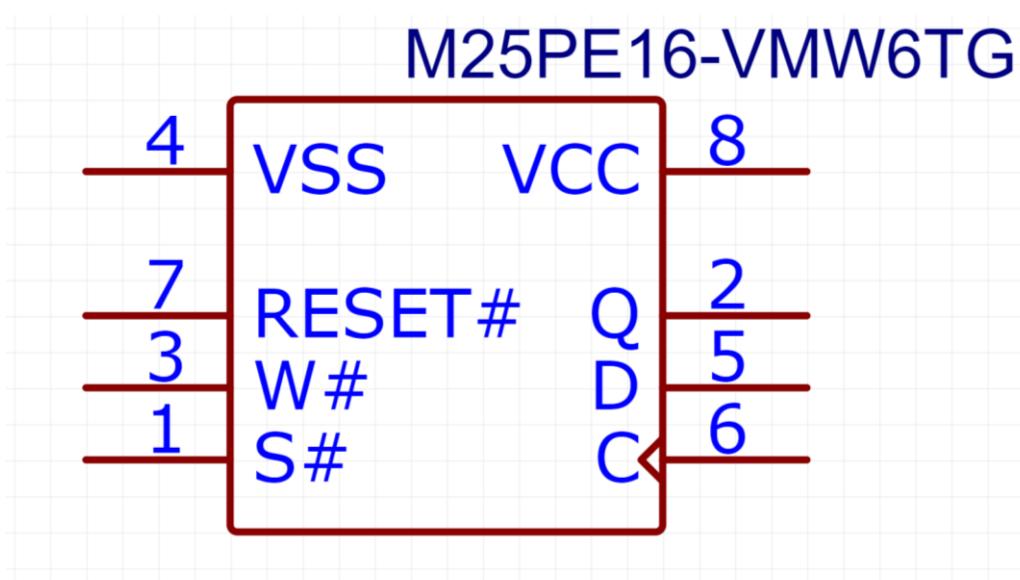


Рисунок 7.22. Графическое обозначение микросхемы флэши-памяти с интерфейсом SPI

Пример временных диаграмм работы SPI показан на рис. 7.23.

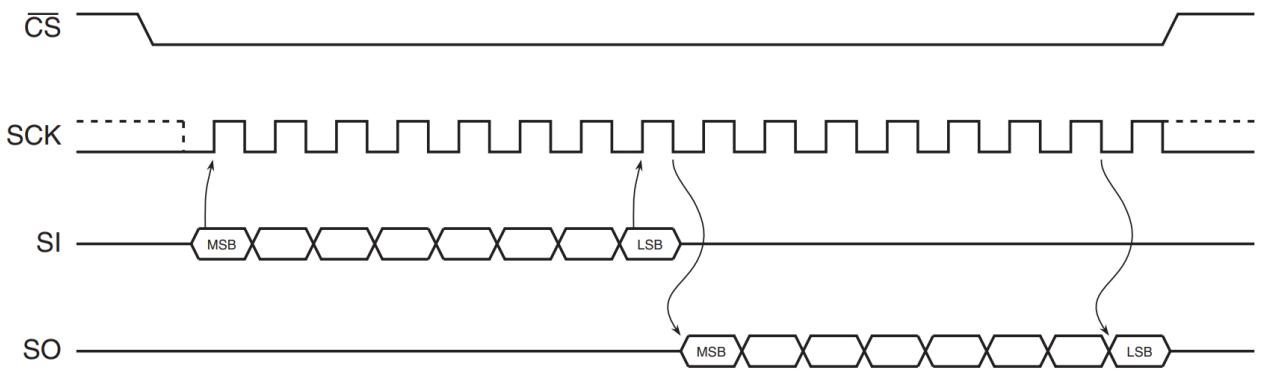


Рисунок 7.23. Временные диаграммы работы интерфейса SPI

Обмен происходит следующим образом. Если сигнал CS переходит от неактивного к активному уровню (т.е. от 1 к 0), это сигнализирует о начале обмена данными. По фронту тактового сигнала микросхема флэш-памяти читает один бит данных с линии SI (она может обозначаться как DI или D). Сам по себе интерфейс SPI предполагает, что по фронту тактового сигнала может происходить взаимный обмен данными, т.е. внешнее устройство также может прочитать данные, которые выдаются на выход SO (также DO или Q). Тем не менее, для флэш-памяти пока нечего выдать на выход.

Работа с флэш-памятью по интерфейсу SPI начинается с передачи команды. Команда может занимать от 1 до 6 байт, многие команды требуют передачи дополнительных аргументов. Формат команды и их перечень сложился из практики выпуска микросхем их основными производителями, и в настоящее время наблюдается его определенная унификация. В то же время, при работе с конкретной микросхемой следует обращаться к документации производителя.

Например, чтение данных для микросхемы, показанной на рис. 7.22, начинается с передачи команды 03h, за которой следуют три байта, представляющих собой 24-разрядный адрес ячейки памяти. После передачи последнего бита следующие фронты тактового сигнала будут выдавать на выход содержимое памяти по указанному адресу.

Скорость обмена данными по интерфейсу SPI относительно невысока, однако постоянное запоминающее устройство часто используется в основном для чтения. Поэтому при начале работы устройства содержимое ПЗУ копируется в ОЗУ, с которым и происходит основная работа. При необходимости интенсивной работы с флэш-памятью следует использовать параллельный интерфейс.

Существует большой подкласс микросхем флэш-памяти, использующий внутреннюю организацию, в чем-то подобную динамической памяти – это память NAND-flash. Как и динамическая память, она имеет сложный протокол обмена

данными, но при этом большую емкость, и обычно используется в составе субмодулей с собственным интерфейсом – например, в SSD с интерфейсом PCI Express, или в флэш-накопителях с интерфейсом USB.

7.7. Выводы по разделу

Память представляет собой массив однотипных ячеек, хранящих данные. Самым простым вариантом является использование параллельного интерфейса, позволяющего подать адрес и получить данные, находящиеся по этому адресу. Этот интерфейс часто реализуется для статической памяти.

Языки описания аппаратуры позволяют реализовать фрагменты памяти в ПЛИС или СБИС. Для этого можно использовать поведенческое описание, на основании которого синтезатор выберет один из видов ресурсов, доступный в конкретной ПЛИС. В то же время, можно использовать и структурное описание, напрямую указывая тип аппаратного блока памяти, который необходимо использовать. Этот вариант несколько более сложен и менее нагляден, однако гарантирует использование аппаратного блока в тех случаях, когда синтезатор не может распознать шаблон описания памяти в поведенческом стиле.

Флэш-память небольшого объема может быть достаточно просто подключена по 4-проводному интерфейсу SPI. Он имеет невысокую скорость обмена данными и требует реализации протокола, подразумевающего посылку команд и адресов, однако это относительно простая задача.

Динамическая память (например, типа DDR3, DDR4) и флэш-память типа NAND имеют большие объемы при относительно невысокой стоимости, однако их подключение требует реализации сложного контроллера. Такой контроллер имеет высокую трудоемкость разработки, поэтому рекомендуется выбирать готовые IP-ядра, поставляемые в том числе производителями ПЛИС.

Применение динамической памяти требует предварительного анализа на уровне архитектуры системы. Пиковые характеристики производительности динамической памяти достигаются в определенных сценариях работы. Попытка произвольного доступа приводит для нее к резкому падению производительности, поэтому динамическую память обычно используют вместе со статической.

Контрольные вопросы:

1. Какие ресурсы имеет ПЛИС для реализации памяти?
2. Какие сигналы имеет постоянное запоминающее устройство с асинхронным интерфейсом?

3. Какие сигналы имеет оперативное запоминающее устройство с синхронным интерфейсом?
4. Для чего предназначены сигналы CS, OE, RW у статической памяти?
5. Чем отличается ячейка динамической памяти от ячейки статической памяти? Почему у динамической памяти существенно больше объем?
6. Почему с микросхемой динамической памяти нельзя работать как с микросхемой статической памяти: подать адрес и прочитать данные?
7. Что такое латентность доступа и как она изменялась на протяжении смены поколений динамической памяти?
8. Почему динамическую память используют вместе со статической?
9. Какие преимущества и недостатки имеет интерфейс SPI по сравнению с параллельным интерфейсом?
10. Как подключить флэш-память с интерфейсом SPI?

8. ГЛОБАЛЬНО АСИНХРОННЫЕ, ЛОКАЛЬНО СИНХРОННЫЕ СХЕМЫ

8.1. Содержание раздела

Глобально асинхронные, локально синхронные схемы (GALS – Globally Asynchronous, Locally Synchronous) стали реакцией на возрастание площади цифровых микросхем и невозможность обеспечить трассировку тактового сигнала по такой большой площади. Вместо попытки синхронизировать все триггеры кристалла можно разбить проект на несколько фрагментов, внутри каждого из которых выполняются правила синхронного проектирования. Однако передавать данные между такими фрагментами («тактовыми доменами») можно с соблюдением определенных мер, иначе можно получить в приемном триггере неопределенное состояние («метастабильное»).

Кроме этого, внешние источники сигналов (микросхемы, кнопки и переключатели и т.д.) также обычно не привязаны к внутреннему источнику тактового сигнала. Для надежного приема логического уровня в проекте такие сигналы необходимо синхронизировать.

В этом разделе рассматривается понятие метастабильности и методы проектирования схем, позволяющие избежать негативных эффектов от его появления.

8.2. Временные характеристики синхронных устройств и понятие метастабильности

Для того, чтобы рассмотреть причины появления метастабильности, необходимо сначала уточнить требования к работе триггера. Ранее было показано, что D-триггер по фронту тактового сигнала записывает состояние, которое в этот момент было на его входе данных D. Этот процесс показан на рис. 8.1. Однако возникает вопрос: что именно понимается под термином «в этот момент»? Или, иными словами, насколько мал может быть тот интервал времени, который оказывает существенное влияние на процесс записи данных?

Производители микросхем на основе своих технологических библиотек предоставляют разработчикам данные об этих временных интервалах. Данные на входе D должны быть стабильными в течение *времени установки* t_{setup} до фронта тактового сигнала и оставаться такими же в течение *времени удержания* t_{hold} после фронта тактового сигнала.

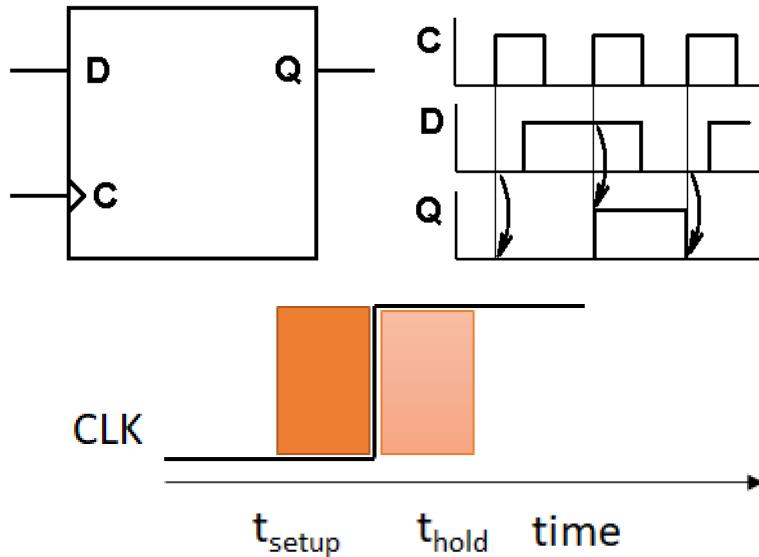


Рисунок 8.1. Временные диаграммы работы триггера

Естественный вопрос – что произойдет, если эти требования будут нарушены и состояние входа D изменится внутри показанных интервалов? Такое нарушение (называемое *timing violation*) может привести к попаданию транзисторов триггера в аналоговый режим работы, поскольку в момент записи данных на их затворах будет промежуточное напряжение, не соответствующее ни гарантированному нулю, ни гарантированной единице. На выходе триггера при этом также может появиться аналоговое напряжение, не соответствующее ни нулю, ни единице, как показано на рис. 8.2. С течением времени триггер придет к определенному цифровому состоянию, но будет это 0 или 1, невозможно сказать заранее.

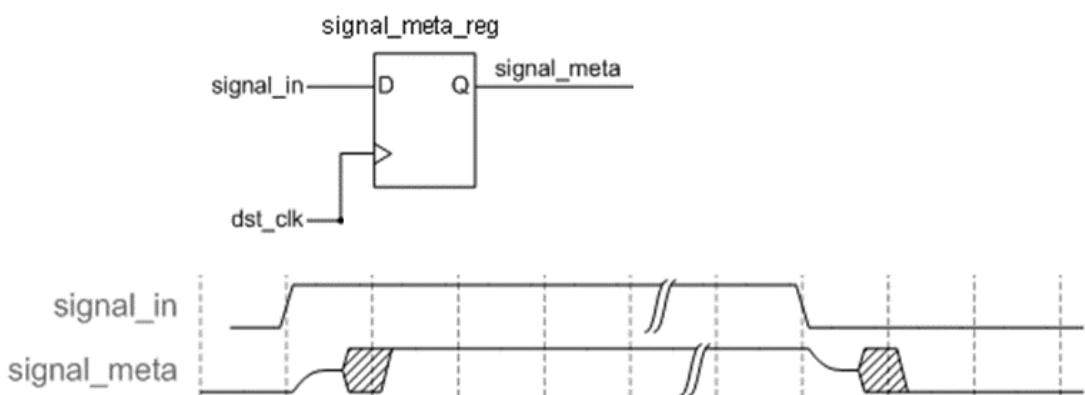


Рисунок 8.2. Временные диаграммы работы триггера при нарушении требований установки и удержания сигнала

Именно это состояние триггера, в котором на его выходе временно присутствует аналоговое напряжение, не соответствующее ни 0, ни 1, называется метастабильным состоянием. Негативный эффект от метастабильности показан на рис. 8.3.

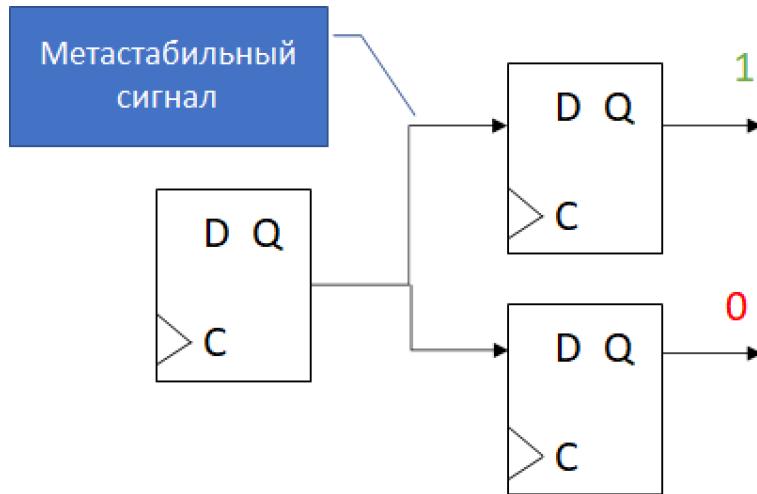


Рисунок 8.3. Пример негативного эффекта от метастабильности

На этом рисунке видно, что выход триггера, попавшего в метастабильное состояние, подан на входы двух других триггеров. По схеме ожидается, что в эти триггеры на следующем такте будет записано одно и то же логическое значение. Однако из-за метастабильности напряжение на входах этих триггеров находится в неопределенной зоне, и триггеры вследствие технологического разброса параметров вполне могут записать разные значения. Работа системы в этом случае становится непредсказуемой.

Самым простым способом устранения этого эффекта является применение цепочки из двух триггеров, как показано на рис. 8.4.

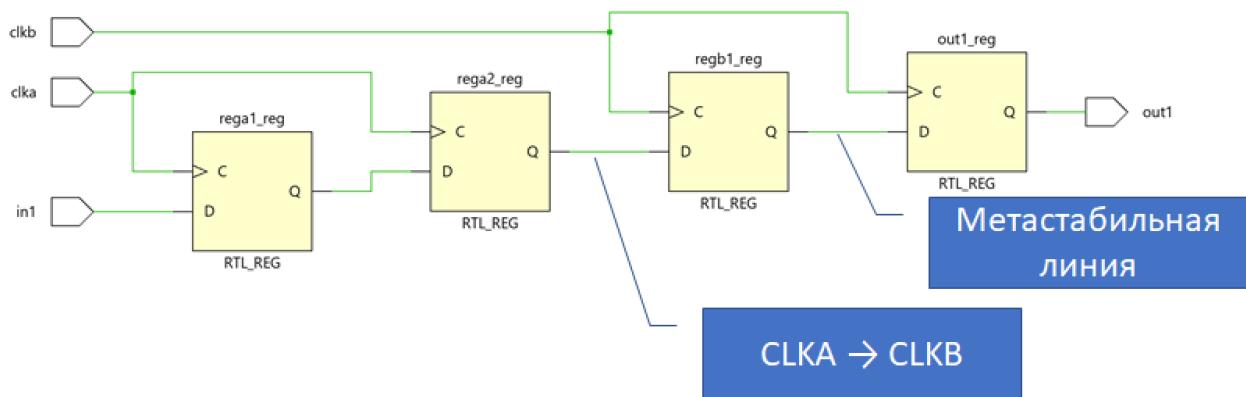


Рисунок 8.4. Пример передачи сигнала между тактовыми доменами

На этом рисунке существует линия, передаваемая от триггера, тактируемого сигналом CLKA, к триггеру, тактируемому сигналом CLKB. Совокупность компонентов, тактируемых одним и тем же тактовым сигналом, называется *тактовым доменом* (*clock domain*). Поэтому более короткими словами можно сказать, что происходит передача данных из домена CLKA в домен CLKB.

Подавление негативного эффекта происходит за счет того, что получаемая метастабильная линия подключена только к одному триггеру. Этот триггер (самый правый на схеме) однозначно определяет, перейдет ли он в состояние 0 или в состояние 1. Даже если это произойдет не в соответствии с ожиданиями разработчика, ситуация будет исправлена на следующем такте.

Метастабильное состояние имеет вероятностный характер. Нарушение требований по времени установки и удержания не означает автоматического попадания в метастабильное состояние, а только увеличивает вероятность такого события. Невозможно также и предсказать, в какое состояние (0 или 1) попадет триггер, выходя из этого состояния, как невозможно, например, угадать, в какую сторону упадет карандаш, если поставить его вертикально на острие. Можно лишь достаточно уверенно утверждать, что карандаш рано или поздно упадет, а если отклонить его от вертикали на какой-то угол, то он почти гарантированно упадет в соответствующую сторону. Аналогично тому, как карандаш может некоторое время стоять на острие, триггер также может некоторое время находиться в промежуточном состоянии. Тогда и второй триггер имеет вероятность попасть в свою очередь в метастабильное состояние на следующем такте. Однако вероятность этого на практике довольно мала. Тем не менее, в ряде случаев используют цепочки из трех триггеров – вероятность попадания в метастабильное состояние каждого последующего триггера уменьшается по экспоненциальному закону.

Неправильная работа с метастабильными сигналами часто является источником труднообнаружимых ошибок. Эта проблема усугубляется тем, что системы моделирования не могут адекватно воспроизвести попадание в метастабильное состояние, поскольку даже использование генератора случайных чисел не сможет воспроизвести именно то состояние, в которое попадет конкретный триггер. Поэтому логическое моделирование основывается на том, что триггер запишет то значение, которое было вычислено симулятором для его входа в момент времени, соответствующий фронту тактового сигнала. Независимо от того, насколько раньше фронта это состояние появилось, и как быстро сменится, состояние триггера будет смоделировано как стабильное. Например, на рис. 8.5 показаны результаты моделирования схемы,

представленной на рис. 8.4. Видно, что никакого метастабильного состояния нет, и предупреждения о его возможном возникновении также не выдается.

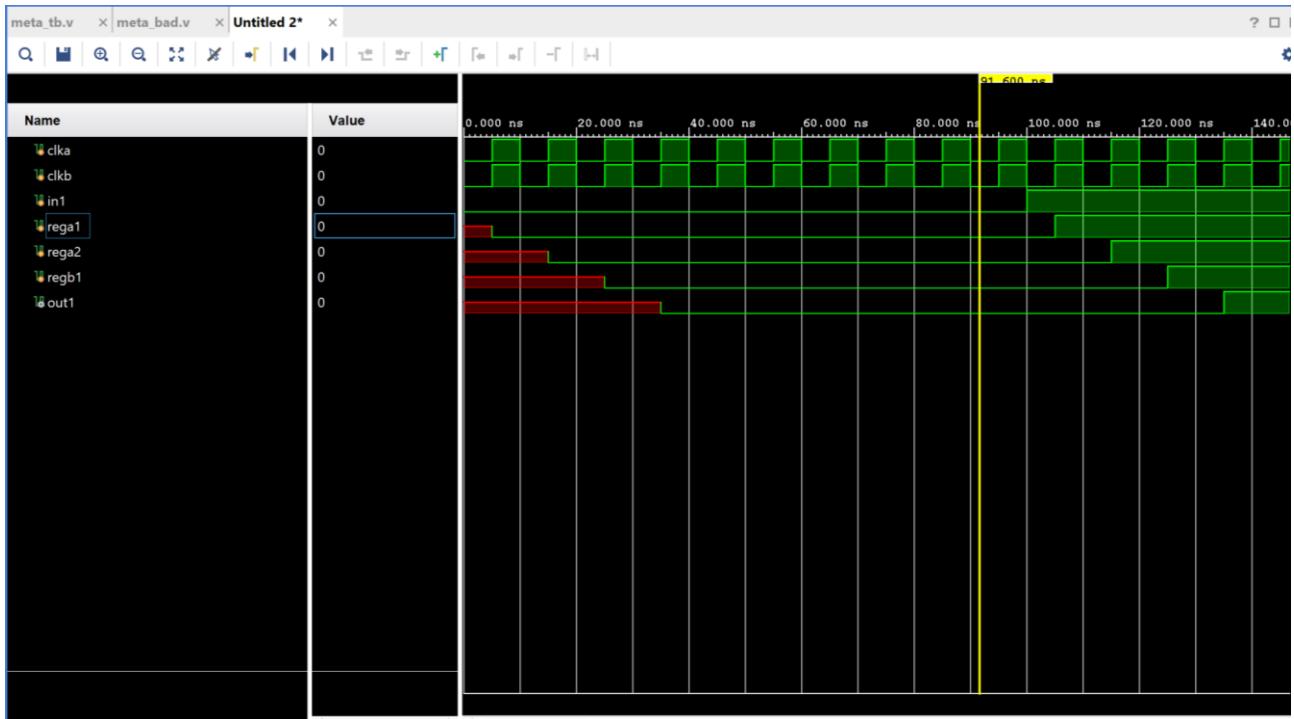


Рисунок 8.5. Моделирование передачи сигнала между тактовыми доменами.

Метастабильность не может быть выявлена в модели

Важным выводом здесь является то, что успешное моделирование цифровой схемы с несколькими тактовыми доменами не является гарантированным подтверждением ее нормальной работы.

Может возникнуть вопрос, почему же смоделированная схема будет работать некорректно, если показанные тактовые сигналы имеют одинаковую частоту? Моменты появления их фронтов также совпадают, поэтому нарушений времени установки или удержания в схеме быть не должно.

На практике оказывается, что тактовые сигналы, номинально имеющие одинаковую частоту, в действительности имеют разную форму. Прежде всего, у каждого тактового генератора имеется технологический разброс. Типичным значением разброса для кварцевого генератора общего назначения является 20-50 ppm (points per million, «единиц на миллион»). Т.е., при номинальной частоте 100 МГц разброс составит $100\text{МГц} * 50 / 1000000 = 5000$ Гц. Именно на это значение могут отличаться реальные частоты двух кварцевых генераторов. Кроме того, в процессе работы частота может изменяться из-за колебаний температуры. Поэтому модель тактового сигнала, описанная в виде номинального значения периода, не будет воспроизводиться в реальной

микросхеме. Применение генератора случайных чисел также не помогает, поскольку смоделированный таким образом сигнал никак не определит поведения реального кварцевого генератора.

Единственным способом устранения негативных эффектов от метастабильного состояния является применение рекомендованных схем ресинхронизации – передачи сигналов между тактовыми доменами.

8.3. Виды синхронизации цифровых узлов

По способу взаимодействия тактовых сигналов цифровые схемы можно разделить на следующие типы:

- System synchronous
 - Микросхемы (или модули одной микросхемы) имеют общий тактовый сигнал. САПР обеспечивает анализ времени распространения данных и проверяет t_{setup} и t_{hold}
- Source synchronous («синхронизированный с источником»)
 - Микросхемы (или модули одной микросхемы) тактируются собственными тактовыми сигналами. С точки зрения микросхемы-приемника, данные могут измениться в любой момент времени
- Self-synchronous
 - Отдельного тактового сигнала нет. Отдельные фрагменты данных определяются по перепадам, фиксированным временным интервалам, частоте, фазе и т.д.

На рис. 8.6 показаны две схемы синхронизации цифровых узлов (это же можно применить и к двум отдельным микросхемам). Системная синхронизация удобна тем, что для нее нет необходимости выполнять ресинхронизацию, поскольку обе части схемы используют один и тот же тактовый сигнал. Однако такое поведение удается обеспечить не всегда. Препятствовать этому может разрастание площади синхронной схемы, вынуждающей использовать другую тактовую сеть, техническая невозможность подать на два устройства один и тот же тактовый сигнал или другие причины. Можно также отметить, что системно синхронные устройства обычно ограничивают тактовую частоту, особенно если речь идет о двух отдельных микросхемах. Время распространения сигнала по печатным проводникам может быть достаточно большим, что приведет к необходимости снижать тактовую частоту, давая сигналам достаточно времени для передачи из одной микросхемы в другую.

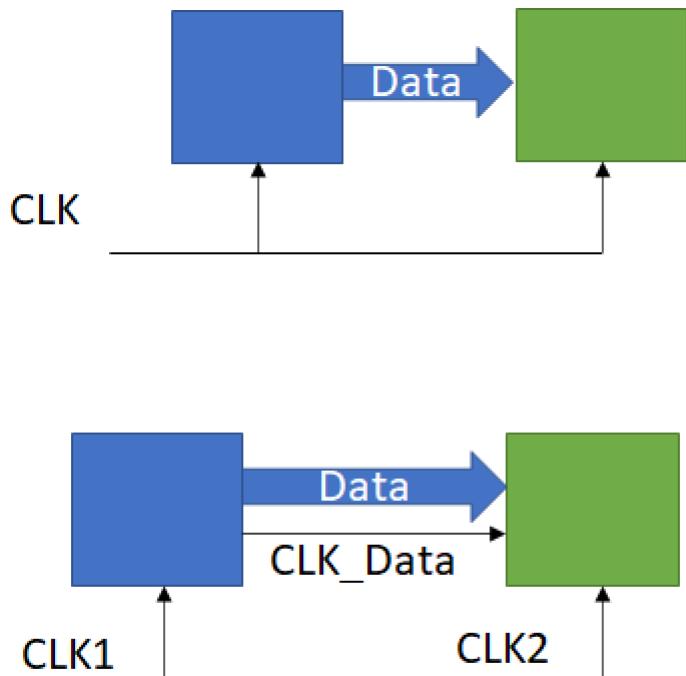


Рисунок 8.6. Синхронная (сверху) и «синхронизированная с источником» (снизу) схемы

Сигнал, синхронизированный с источником, может использовать общий тактовый сигнал, однако в любом случае микросхема-источник формирует собственный тактовый сигнал, фронт которого соответствует ситуации гарантированной корректности данных на выходе этой микросхемы. Преимуществом такой схемы является ее практически независимость от длины проводников, соединяющих две микросхемы. Если данные запаздывают на некоторое время из-за распространения по печатным проводникам, то на примерно такое же время будет запаздывать и фронт тактового сигнала. При этом можно рассчитывать на то, что если уж эти сигналы были сформированы в один и тот же момент времени, то при одинаковых задержках они придут на микросхему-приемник также в одинаковые моменты времени.

Чтобы использовать эти соображения, при разработке печатной платы необходимо тщательно следить за выравниванием длин проводников. Кроме того, не должно быть источников помех или заземляющих полигонов большой площади, которые находились бы рядом с некоторыми сигналами – в этом случае такие сигналы даже при равной длине проводника будут иметь другую форму, с более пологими фронтами, что приведет к более позднему пересечению порога устойчивого логического состояния. Обычно эти вопросы решаются отдельно, при разработке печатной платы, однако разработчик печатной платы должен

получить соответствующие комментарии в виде требований к максимально допустимой неравномерности длины проводников в определенной группе.

Получаемый приемником сигнал в виде параллельной шины данных является несинхронизированным, поскольку он тактируется на рис. 8.6 сигналом CLK_Data, а приниматься и обрабатываться должен схемой, тактируемой сигналом CLK2.

Выше было указано, что нет никакого способа принудительно сделать два тактовых сигнала синхронными. Вместо этого необходимо выявлять тактовые сигналы, между доменами которых передаются данные, и проверять для них наличие схем ресинхронизации. Часто САПР имеют встроенные инструменты для выявления взаимодействия между тактовыми доменами. Например, на рис. 8.7 показаны результаты выполнения команды report_clock_interaction в САПР Xilinx Vivado. Анализируемая схема соответствует показанной на рис. 8.4.

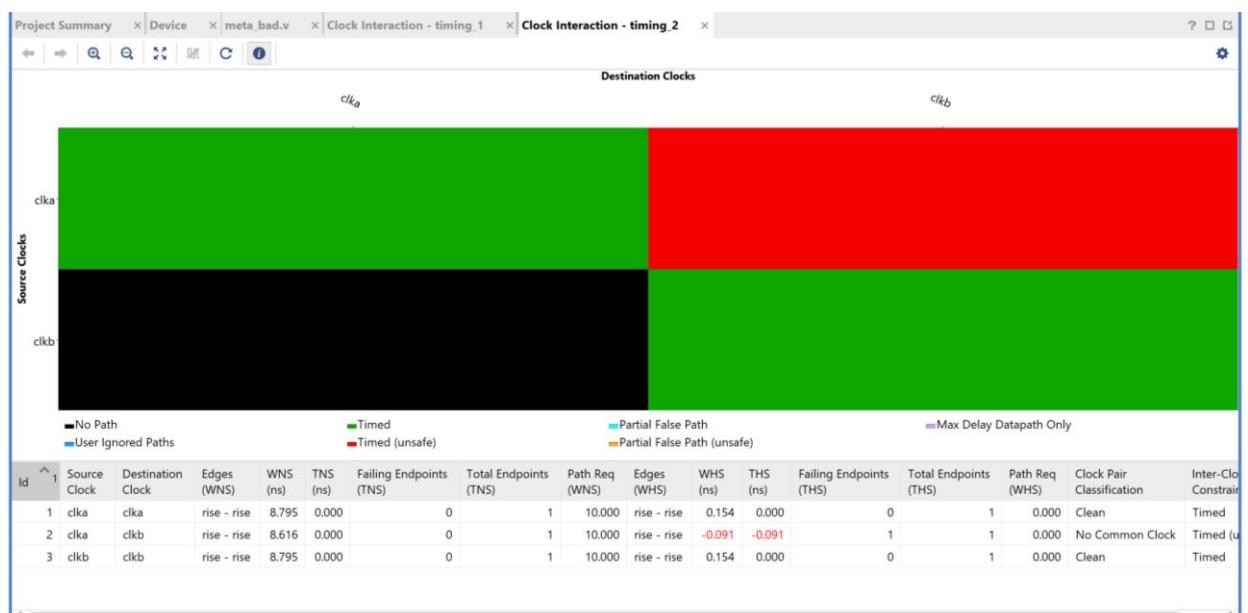


Рисунок 8.7. Анализ взаимодействия тактовых доменов в САПР Xilinx Vivado

Результаты анализа выводятся в виде таблицы, ячейки которой закрашены в зависимости от обнаруженного взаимодействия. По вертикали и горизонтали помещаются тактовые сигналы, имеющиеся в схеме. Зеленый цвет соответствует ситуации, когда взаимодействие нормально проанализировано в САПР. Очевидно, что зеленый цвет показан для ситуаций CLKA – CLKA и CLKB – CLKB, поскольку это синхронные схемы, тактируемые одними и теми же сигналами каждой.

Для ситуации CLKB – CLKA цепочек не обнаружено, поскольку на рис. 8.4 нет ни одного триггера в домене CLKB, который передавал бы данные в домен

CLKA. В другой схеме ситуация, очевидно, может быть и не такой. Тем не менее, ячейка показана черной, что соответствует ситуации «такого взаимодействия не обнаружено». Соответственно, эта ячейка и не является проблемной.

Оттенки красного цвета соответствуют ситуациям, когда выявленное взаимодействие может вызвать проблемы из-за некорректного моделирования. В данном случае красным помечена ячейка CLKA – CLKB, и на схеме можно видеть, что есть триггер, тактируемый CLKA, выход которого подключен к входу, тактируемому CLKB. Красным цветом САПР указывает на обнаружение таких цепей, и задачей разработчика является добавление *проектных ограничений*, которые перечислили бы цепи, в работоспособности которых разработчик уверен исходя из архитектурных соображений.

Можно подчеркнуть, что добавление проектных ограничений не превращает несинхронизированные сигналы в синхронизированные. Это просто информирование САПР о том, что какой-то сигнал в действительности не должен анализироваться, поскольку схема построена так, чтобы его метастабильное поведение не вызвало проблем в проекте. САПР не может выполнить ресинхронизацию самостоятельно или каким-то образом подстроить фазу тактового сигнала на основании проектных ограничений.

Если несколько тактовых сигналов формируются в одном генераторе (PLL или MMCM), то они не образуют перехода между тактовыми доменами. Поскольку тактовый генератор подстраивает фазу для целой группы своих выходов, можно сформировать в одном PLL сигналы 50, 100, 200 МГц, и положение их фронтов будет совпадать. В терминах САПР такие сигналы называются *related clocks*, в технически корректном переводе «связанные тактовые сигналы». В противоположность этому сигналы, полученные из разных источников, или даже сигналы одинаковой частоты, сформированные разными PLL, обозначаются как *unrelated clocks*, и к ним применимы все описанные соображения – передача данных между доменами этих сигналов требует ресинхронизации.

8.4. Архитектура GALS и схемы ресинхронизации

Для общего представления о схемах ресинхронизации необходимо ознакомиться с архитектурой GALS (Globally Synchronous, Locally Asynchronous). В общем виде ее можно представить на рис. 8.8, где имеются две подсистемы, каждая из которых является синхронной внутри, однако они используют независимые тактовые сигналы CLK1 и CLK2. Попытка соединить

компоненты подсистем вызовет появление метастабильного сигнала, поэтому любая передача данных требует применения схем ресинхронизации.

В целом, ресинхронизация может быть выполнена либо в виде цепочки триггеров (минимум двух), как было показано выше, либо с помощью компонентов, которые аппаратно поддерживают два таймовых сигнала. Именно с этой целью в ПЛИС используют двухпортовую блочную память, каждый порт которой может использовать собственный таймовый сигнал.

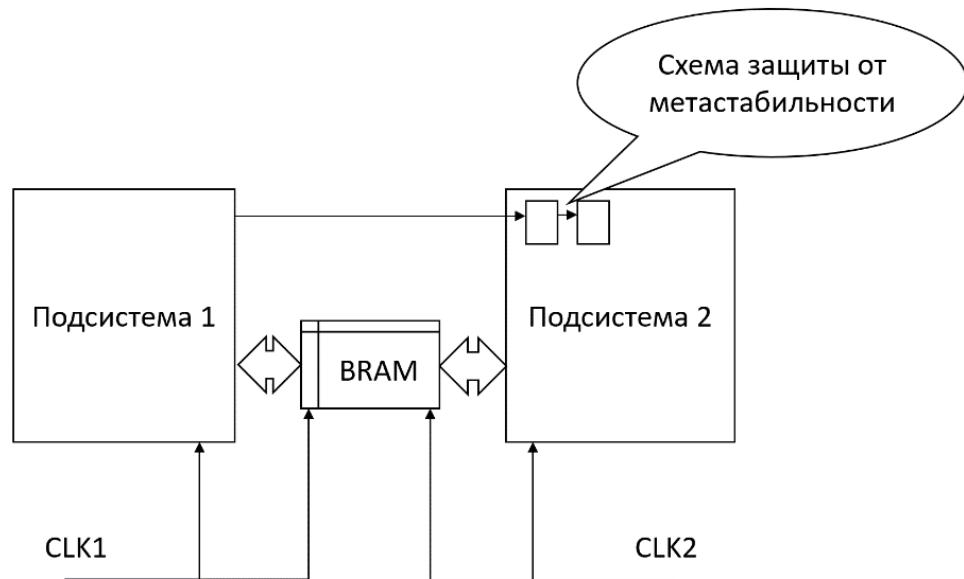


Рисунок 8.8. Пример системы, выполненной в соответствии с архитектурой GALS

Можно еще раз перечислить основные приемы ресинхронизации данных. На рис. 8.9 показана схема на основе цепочки триггеров. Предполагается, что второй триггер с меньшей вероятностью попадет в метастабильное состояние, поэтому его выходом можно пользоваться, не опасаясь появления промежуточного уровня напряжения, неоднозначно воспринимаемого другими компонентами.

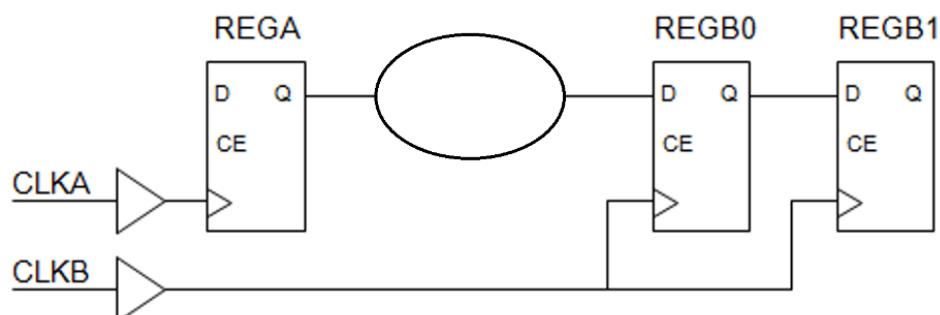


Рисунок 8.9. Простейшая схема передачи данных между таймовыми доменами («схема ресинхронизации»)

Для упрощения реализации схем ресинхронизации их шаблоны приведены в справочной системе САПР, как показано на рис. 8.10. Они сгруппированы в разделе Clock-Domain Crossing (CDC), что соответствует понятию «ресинхронизация».

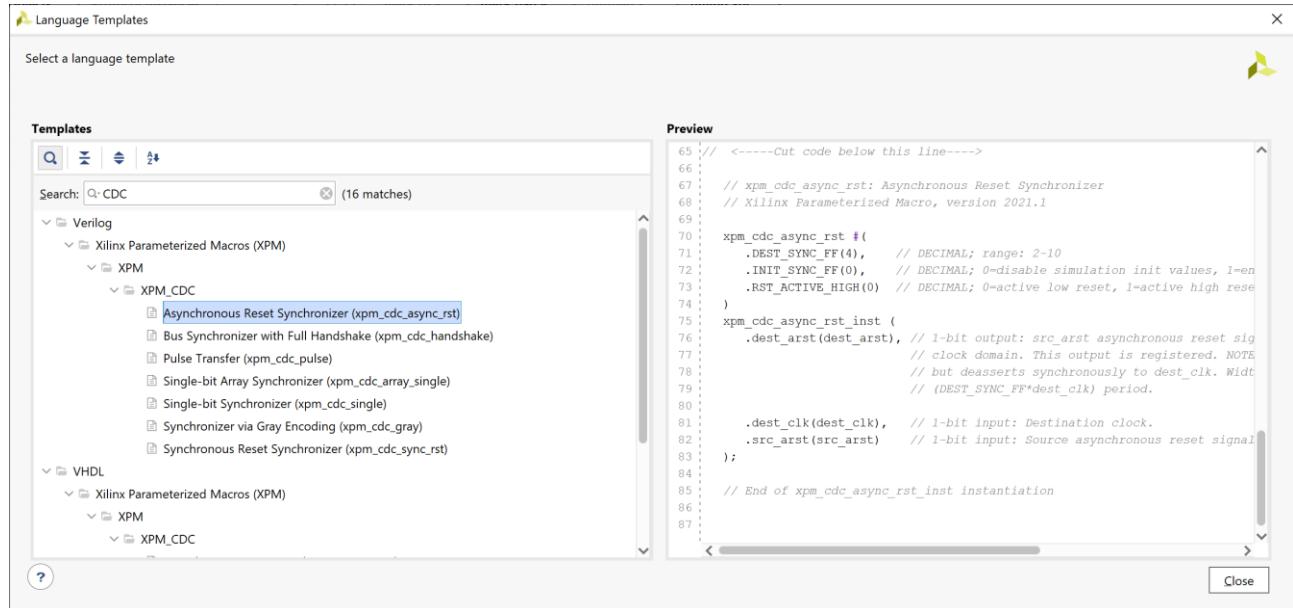


Рисунок 8.10. Шаблоны схем ресинхронизации в справочной системе САПР Vivado

Один из подходов ресинхронизации данных с помощью двупортовой памяти показан на рис. 8.11.

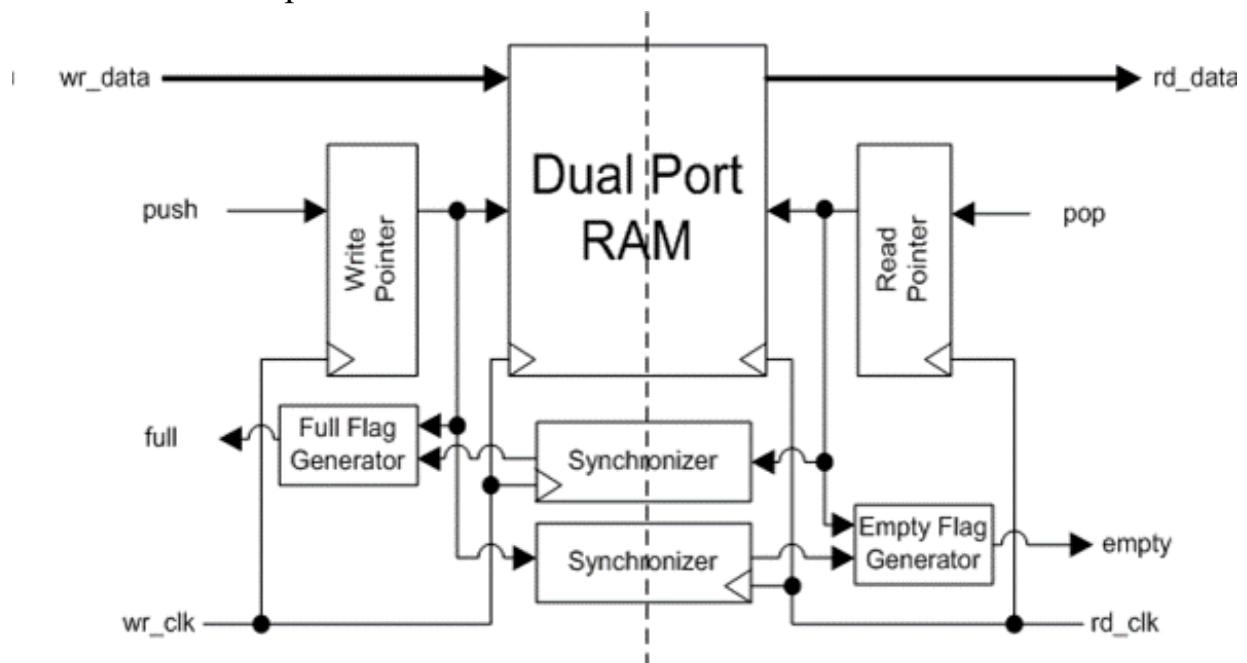


Рисунок 8.11. Схема ресинхронизации на основе двупортовой памяти

В этой схеме двупортовая память используется для организации очереди (FIFO – First In, First Out). Данные, передаваемые между тактовыми доменами, записываются в FIFO с одной стороны и читаются с другой. В действительности физического перемещения данных внутри FIFO не происходит, вместо этого перемещаются указатели на «голову» и «хвост» очереди данных. Для FIFO используются однобитные сигналы («флаги») empty и full, т.е. «пустое» и «заполненное». Если указатели головы и хвоста совпадают, это можно трактовать так, что данных для чтения нет. При записи данных указатель записи смещается и флаг empty снимается. Поскольку флаги передаются в другой тактовый домен через схемы ресинхронизации (показанные на рис. 8.11 как Synchronizer), домен-получатель узнает о наличии данных гарантированно после их физической записи в память. Иными словами, если флаг empty становится равным 0, можно производить чтение данных.

Указатели чтения и записи, а также поддержка флагов empty и full часто генерируются в модуле FIFO. Например, в ПЛИС Xilinx существуют аппаратные контроллеры FIFO, совмещенные с компонентами BRAM, которые могут применяться при необходимости. Автоматический синтез таких компонентов из поведенческого описания в настоящий момент не поддерживается.

Еще одним приемом ресинхронизации является выделение фронта тактового сигнала. Этот прием можно эффективно использовать при разработке контроллеров SPI, работающих на прием данных (slave, или «ведомый» контроллер). Иллюстрация к работе такой схемы показана на рис. 8.12.

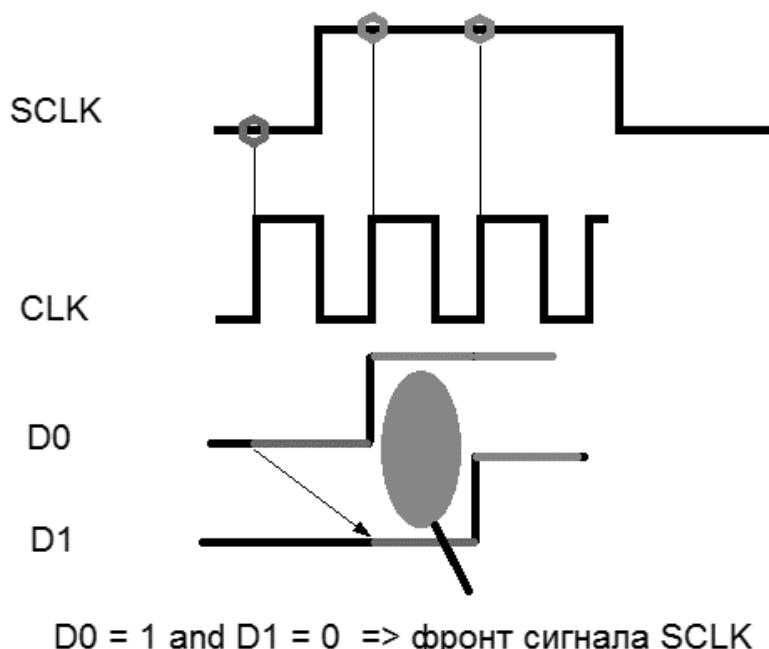
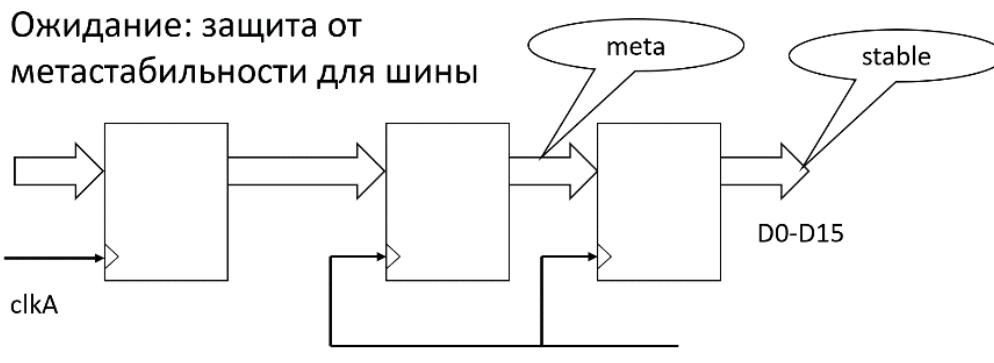


Рисунок 8.12. Выделение фронта тактового сигнала

Если тактировать цепочку триггеров D0, D1 высокочастотным тактовым сигналом CLK, то эта цепочка будет захватывать последовательные состояния линии SCLK, причем триггер D1 будет содержать предыдущее состояние линии SCLK, а D0 – текущее состояние. Появление фронта сигнала на линии SCLK приведет к тому, что старое состояние линии, хранящееся в D1, будет равно 0, а новое, прочитанное в D0, станет равно 1. Если условие $D0 = '0' \text{ and } D1 = '1'$ использовать в качестве разрешения работы, сигналы данных интерфейса SPI смогут быть приведены к тактовой сети CLK.

Важным замечанием к ресинхронизации является вопрос использование цепочки триггеров для параллельных шин. На рис. 8.13 показана иллюстрация такого подхода. Если разработчик ожидает, что цепочка триггеров, обеспечивающая ресинхронизацию одного сигнала, точно так же работает с параллельной шиной, это будет неявно методической ошибкой, которая может быть выявлена только экспериментально. В действительности попадание в метастабильное состояние и сценарий выхода из него будет индивидуален для каждого разряда параллельной шины, и в большинстве случаев на выходе такого блока будет комбинация «старых» и «новых» данных. Если учесть, что состояние какого-то разряда может и не изменяться, идентифицировать подобные ошибки достаточно сложно.



На практике: независимые цепи

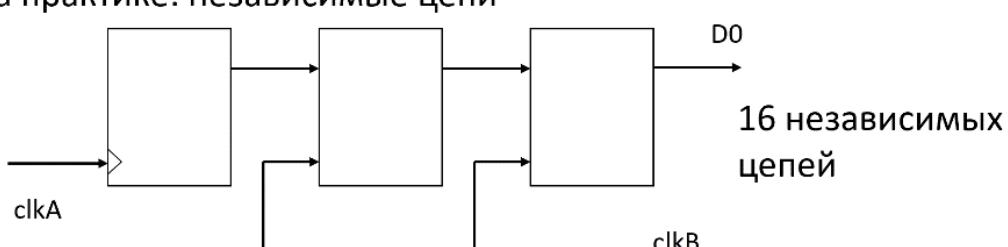


Рисунок 8.13. Замечание к вопросу ресинхронизации параллельных шин

Для правильного анализа метастабильных линий в проект следует добавить *проектные исключения* (constraint). Это необходимо для того, чтобы САПР исключала из анализа временные характеристики сигналов, передающих данные между тактовыми доменами. Пример диалогового окна, помогающего настройке этих параметров, приведено на рис. 8.14.

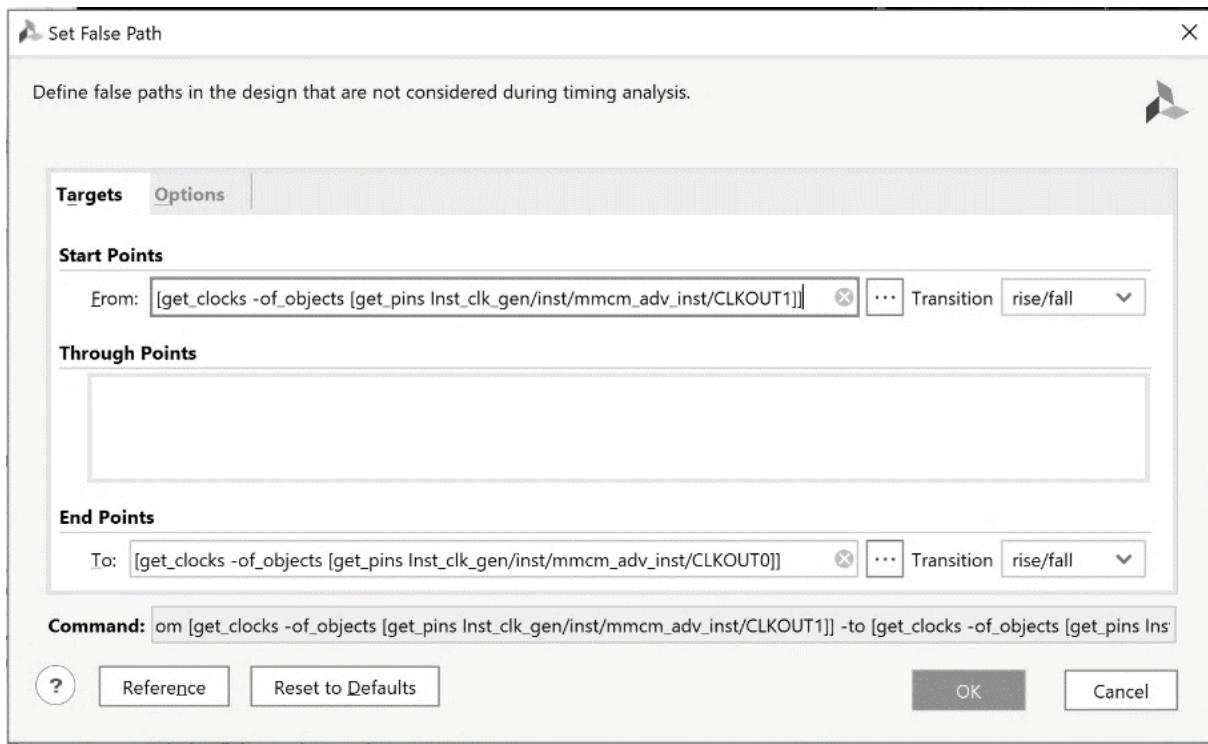


Рисунок 8.14. Диалоговое окно настройки проектных исключений для ресинхронизации данных

Ниже приведены строки на языке описания проектных ограничений xdc. Он основан на скриптовом языке Tcl и используется для ПЛИС Xilinx. Язык очень близок к более распространенному sdc, используемому для описания проектных ограничений СБИС.

```
set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
```

```
set_clock_groups -physically_exclusive -group CLKA -group CLKB
```

```
set_clock_groups -asynchronous -group CLKA -group CLKB
```

```
set_max_delay -from [get_cells REGA] -to [get_cells REGB0] 5 -datapath_only
```

Смыслом приведенных описаний является указание САПР на то, что время распространения сигнала между триггерами, тактируемыми CLKA и CLKБ, можно не анализировать. Первые три выражения являются более общими и работают на всю группу таких сигналов.

Команда `set_false_path` указывает, что цепь является «ложной» (`false`), т.е. несмотря на ее наличие задержка сигнала на этой цепи может быть любой, и это не окажет влияние на работоспособность проекта. Формально, любую цепь можно объявить «ложной», однако ответственность за такое решение лежит целиком на разработчике. Для передачи данных между тактовыми доменами такое объявление корректно.

Две следующие строки примерно одинаковы и используют варианты команды `set_clock_groups`, различаясь параметрами. В первом случае указывается параметр `physically_exclusive`, который в том числе делает линии данных «ложными». Во втором случае параметр `asynchronous` имеет тот же эффект. Детали различия требуют более глубокого ознакомления с правилами описания тактовых сетей.

Наконец, команда `set_max_delay` является более низкоуровневой с точки зрения `xdc`. В данном случае она указывает САПР, что линия между двумя конкретными регистрами не может иметь задержку более заданной. Параметр 5 означает наносекунды и не имеет каких-либо жестких требований. Отличие от предыдущих команд заключается в том, что исключение цепей из анализа задержки формально дает возможность размещать соединяемые компоненты на каком угодно удалении друг от друга. Указание большой, но реалистичной максимальной задержки требует размещения компонентов относительно близко друг от друга.

В целом можно отметить, что управление проектными ограничениями является отдельным процессом при разработке. Добавление исключений следует проводить до тех пор, пока анализ взаимодействия тактовых доменов (`clock interaction`) не перестанет выявлять взаимодействие без явно описанных исключений (красные и оранжевые ячейки).

8.5. Примеры подключения устройств, требующих ресинхронизации данных

Ряд распространенных периферийных устройств являются асинхронными по отношению к той микросхеме, с которой они взаимодействуют. Поэтому при разработке интерфейсов для их подключения требуется реализовать схему ресинхронизации.

На рис. 8.15 показан интерфейс аналого-цифрового преобразователя с синхронным параллельным интерфейсом, синхронизированным с источником (source-synchronous). Такой подход часто используется по мере повышения тактовой частоты (например, на уровне 100 МГц или больше). Микросхема АЦП использует внешний опорный сигнал тактовой частоты, который преобразуется внутренним генератором (например, PLL). В результате выходная тактовая частота, сопровождающая передаваемые данные, имеет то же номинальное значение, однако отличающееся по фазе от опорного сигнала. Поэтому даже если входной сигнал подается из проектируемого интерфейса, нельзя ожидать, что данные можно будет захватывать по сигналу CLKin.

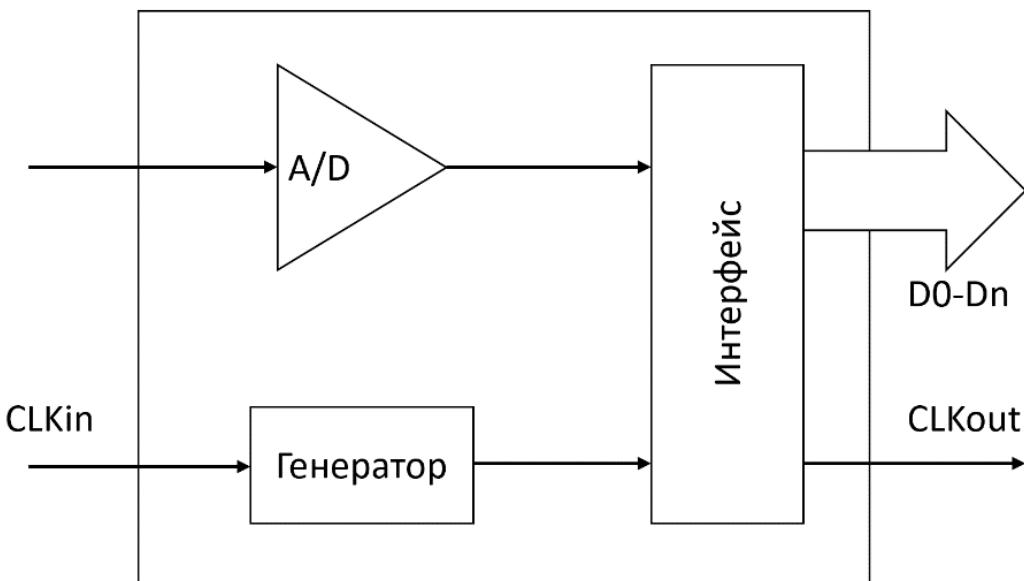


Рисунок 8.15. Аналого-цифровой преобразователь, подключаемый по схеме входа, синхронизированного с источником

Другим примером является цифровая камера. На рис. 8.16 показано подключение камеры с интерфейсом Camera Parallel Interface (CPI). В качестве опорного сигнала тактовой частоты камере требуется сигнал XCLK. Передаваемые данные Data сопровождаются фронтом тактового сигнала PCLK (Pixel Clock), который имеет то же номинальное значение, что и XCLK, однако отличается по фазе. Также камера передает синхросигналы HREF, VSYNC, а для ее настройки используется двухпроводной интерфейс с сигналами SCL, SDA, который подобен I2C.

На рис. 8.17 показано подключение микросхемы Ethernet PHY («физического уровня»). Можно видеть, что приемник (RX) и передатчик (TX)

имеют собственные тактовые сигналы RXCLK и TXCLK, совпадающие по направлению с передаваемыми данными.

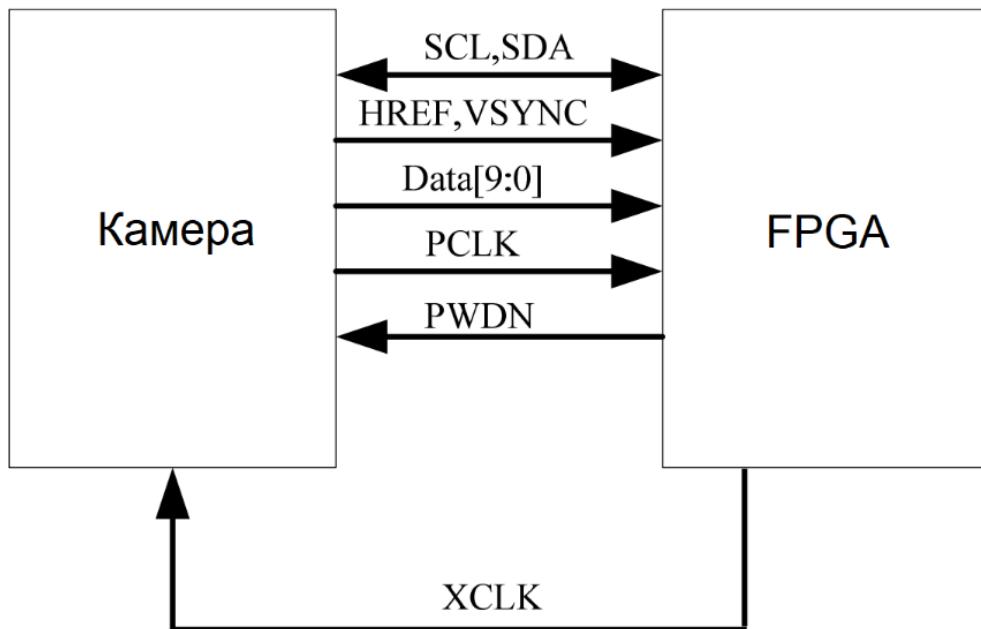


Рисунок 8.16. Видеокамера с интерфейсом CPI (Camera Parallel Interface)

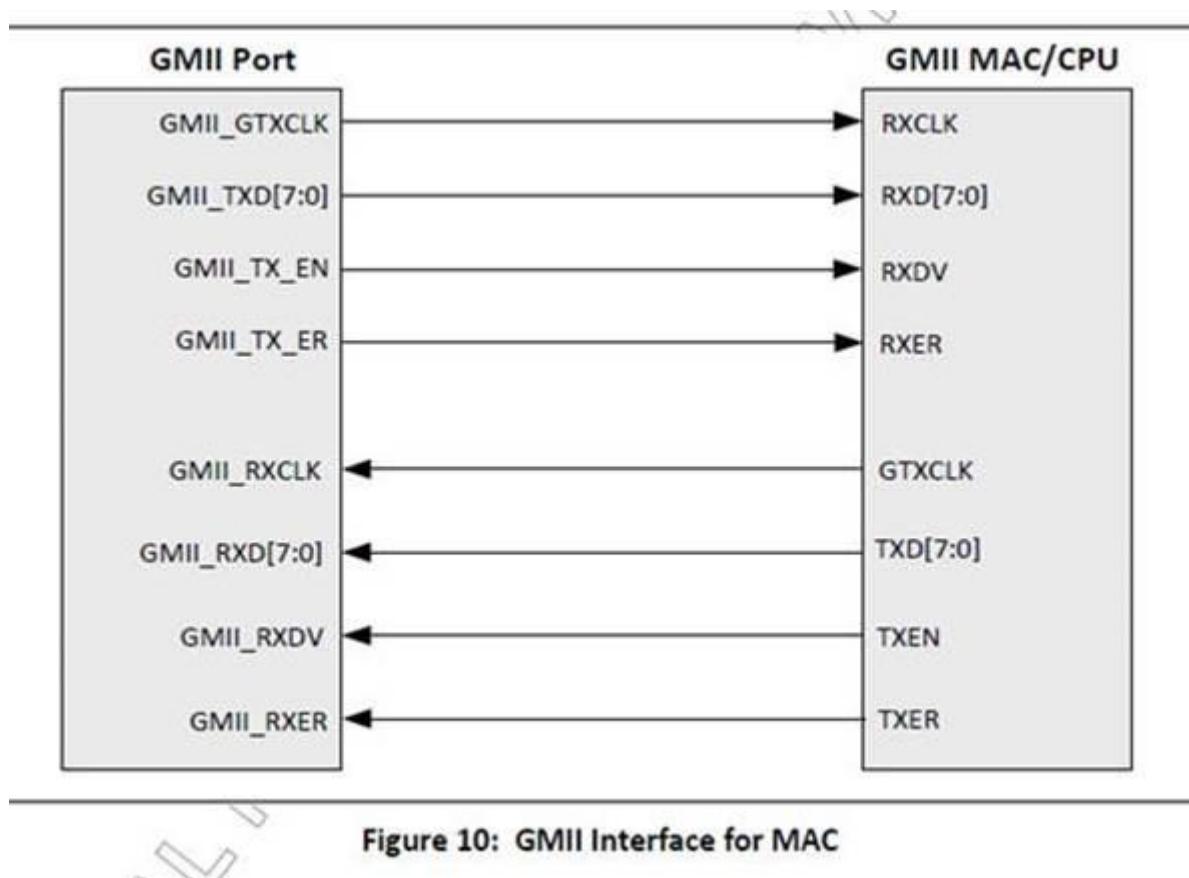


Figure 10: GMII Interface for MAC

Рисунок 8.17. Подключение внешней микросхемы Ethernet PHY

8.6. Выводы по разделу

Попытка изменять данные на входе триггера при подаче фронта тактового сигнала может привести к тому, что триггер попадает в аналоговый режим работы и на его выходе в течение какого-то времени присутствует напряжение между логическими 0 и 1. Невозможно заранее определить, как повлияет такое состояние триггера на подключенные к нему элементы, поэтому работа цифровой схемы становится непредсказуемой. Это состояние называется метастабильным и не устраняется проектными ограничениями или моделированием. Единственным способом избежать его негативного влияния на проект является применение схем ресинхронизации – привязки моментов изменения данных к фронтам тактового сигнала.

Простейшей схемой ресинхронизации является применение цепочки триггеров. Если первый триггер попадает в метастабильное состояние, то для второго триггера вероятность уменьшается. Этот прием не следует применять для ресинхронизации многоразрядных шин, поскольку каждый разряд в такойшине будет вести себя индивидуально.

Архитектурно устойчивым к метастабильности является компонент двупортовой памяти. На его основе можно создать очередь (FIFO), заполняемую на основе одного тактового сигнала и читаемую на основе второго тактового сигнала. Важно, чтобы чтение происходило при гарантированном наличии записанных данных, для чего необходимо анализировать однобитовый сигнал («флаг»), показывающий, что FIFO не пусто.

При анализе схем с несколькими тактовыми доменами не следует полагаться на результаты моделирования, поскольку формальные модели тактовых сигналов показывают поведение схемы в идеализированном варианте и не могут адекватно моделировать метастабильное состояние, которое имеет вероятностный характер.

Контрольные вопросы:

1. Что такое метастабильное состояние триггера?
2. Почему данные могут быть не синхронизированы с тактовым сигналом внутри микросхемы?
3. Второй триггер в цепочке не исправляет метастабильное состояние первого триггера, в чем же тогда заключается его роль?
4. Как использовать двупортовую память для ресинхронизации данных?
5. Как узнать, имеются ли в проекте взаимодействующие тактовые домены?

6. Какие проектные исключения применяются для описания сигналов, требующих ресинхронизации?

7. Почему крупные проекты могут разбиваться на несколько тактовых доменов?

8. Что такое «интерфейс, синхронизированный с источником»? Где он применяется?

9. КОНЕЧНЫЕ АВТОМАТЫ

9.1. Понятие конечного автомата

Конечный автомат (КА), также Finite State Machine (FSM) – цифровая схема, которая может находиться в конечном числе разных состояний. Каждое из состояний однозначно определяется комбинацией внутренних сигналов. Переход между состояниями происходит на основе внешних сигналов и текущего состояния конечного автомата. Именно с помощью конечного автомата можно реализовать цифровую схему, которая будет вести себя подобно компьютерной программе – последовательно выполняя запрограммированные действия. Для этого необходимо распределить требуемые действия по разным состояниям КА и задать правила перехода между этими состояниями в требуемой последовательности.

Выходные сигналы КА могут быть одинаковыми при нахождении в нескольких состояниях, однако обратное не гарантируется. С помощью конечных автоматов можно описывать работу устройств самого разного типа. Например, на рис. 9.1 показан модуль конечного автомата для управления продажей кофе. Модуль выбран в качестве примера и не соответствует реальному устройству. Он необходим для иллюстрации процесса разработки КА.

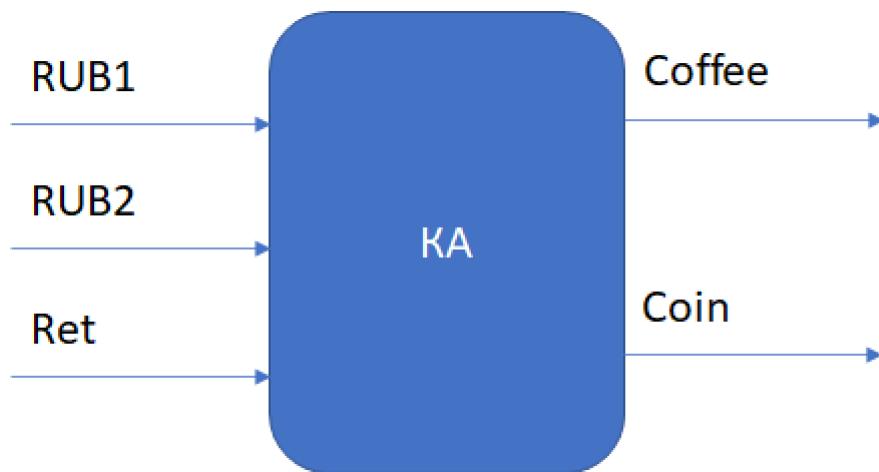


Рисунок 9.1. Модуль конечного автомата по управлению продажей кофе

На этом рисунке видно, что у кофейного аппарата есть датчики опускания монет Rub1 и Rub2, реагирующие на попадание соответствующих монет в монетоприемник. Также есть кнопка Ret, управляющая возвратом сдачи. У автомата есть возможность сварить кофе и выдать сдачу монетой в 1 рубль. Примем, что для этого достаточно установить в 1 соответствующие выходные сигналы.

На основе данного описания можно показать процесс разработки конечного автомата. Описание КА можно представить в виде графа, представляющего собой «островки» - состояния конечного автомата, соединенные стрелками. Над стрелками записывается условие, по которому происходит переход от одного состояния в другое. Автомат является синхронным устройством, поэтому его переходы привязаны к тактовому сигналу. За одно действие автомат может перейти в какое-то одно состояние, но не пройти по длинной цепочке, даже если выполняются все условия. Если автомат попадает в какое-то состояние, ему однозначно соответствует набор выходных сигналов.

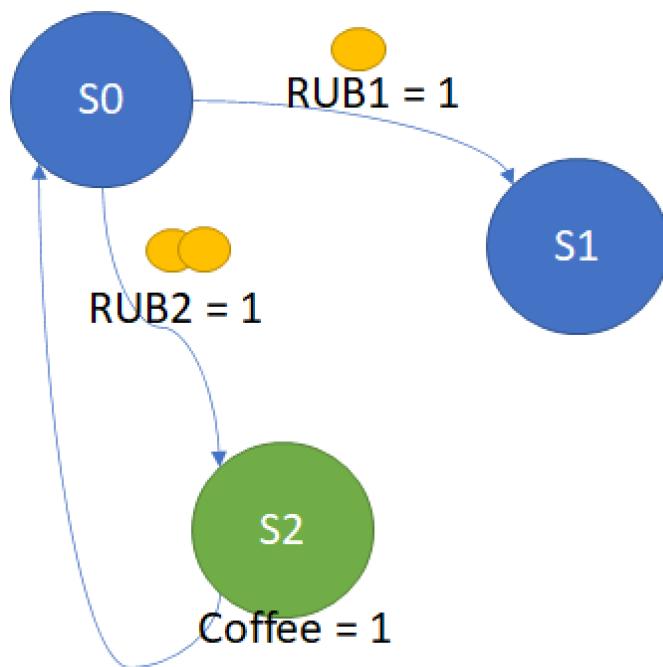


Рисунок 9.2. Граф описания конечного автомата по управлению продажей кофе

Работа автомата, выбранного в качестве примера, начинается с состояния S0. В этом состоянии у него нет активных выходов (т.е. он не готовит кофе и не выдает монету в виде сдачи). Если нет внешних сигналов, КА может находиться в состоянии S0 сколь угодно долго.

В этом состоянии можно нажимать кнопку сдачи, но в этом случае ничего не должно происходить. Могут произойти два интересующих нас события:

1. Опущена монета 1 рубль. Этого пока недостаточно для выдачи кофе, но внутреннее состояние автомата изменилось (хотя внешние сигналы остались теми же – нет выдачи кофе, нет сдачи). Поэтому необходимо завести новое состояние S1, которое будет соответствовать ситуации «уже есть 1 рубль».

2. Опущена монета 2 рубля. Этого достаточно для приготовления кофе. Автомат переходит в состояние S2, в котором есть выходной сигнал Coffee = 1.

Из этого состояния автомат безусловно переходит в S0 и становится готов к приему оплаты за следующую порцию кофе. Детали приготовления кофе в данном случае неважны. Понятно, что это длительный процесс, из которого нельзя мгновенно перейти к S0, однако для упрощения на схеме нет ожидания завершения.

Поскольку цепочка S0 – S2 – S0 в целом понятна, осталось завершить описание состояния S1. Возможны два сценария:

1. Опущен еще 1 рубль. Теперь автомат может готовить кофе, и для этого уже есть состояние S2. В него можно перейти, если КА уже был в состоянии S1, и сработал датчик RUB1. Таким образом, в S2 можно попасть, либо сразу опустив монету 2 рубля, либо последовательно опустив две монеты в 1 рубль.

2. Человек, опустивший одну монету, обнаружил, что у него больше нет подходящих монет. Он хочет получить сдачу, для чего нажимает кнопку Ret («возврат»). Тогда КА должен перейти в состояние S3, в котором формируется выходной сигнал Coin = 1, управляющий устройством по выдаче монеты 1 рубль.

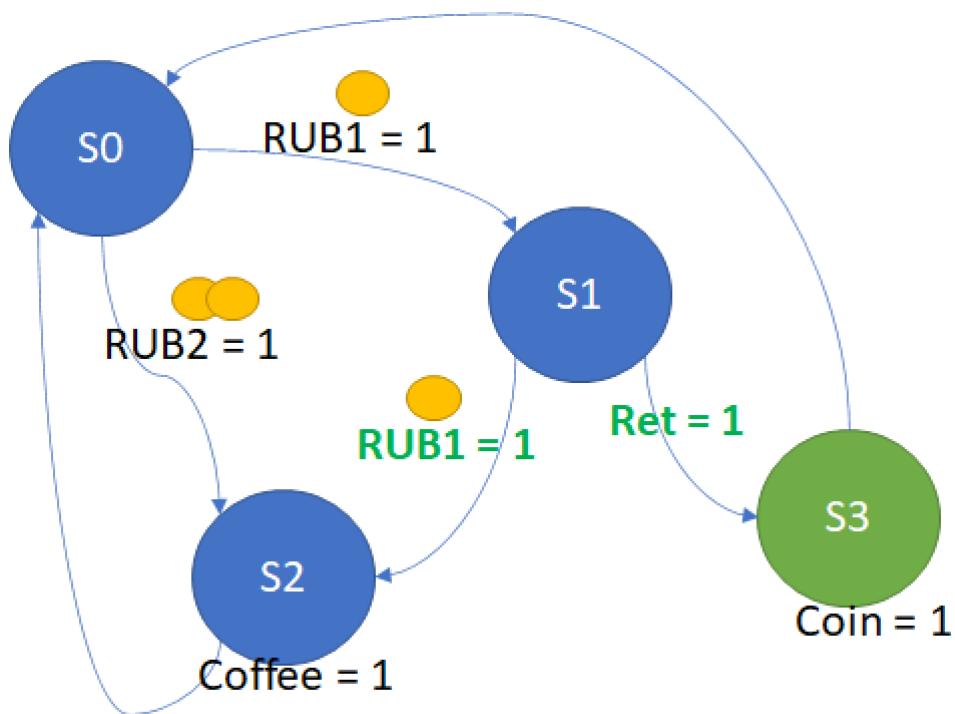


Рисунок 9.3. Граф описания конечного автомата по управлению продажей кофе

Наконец, возможен еще один сценарий. Если после монеты 1 рубль будет опущена монета 2 рубля, это вполне допустимо, поскольку монетоприемник рассчитан и на те, и на другие монеты. Однако тогда автомат получит 3 рубля, и должен будет вернуть сдачу. На первый взгляд, уже существует состояние S2, в

котором готовится кофе, но по правилам описания КА состояние нельзя модифицировать в зависимости от того, по какому пути автомат попал в него. Если требуется другое поведение («одновременно и варка кофе, и выдача сдачи»), необходимо добавить еще одно состояние. На рис. 9.4 это состояние S4, в котором есть два выходных сигнала – Coffee = 1, Coin = 1.

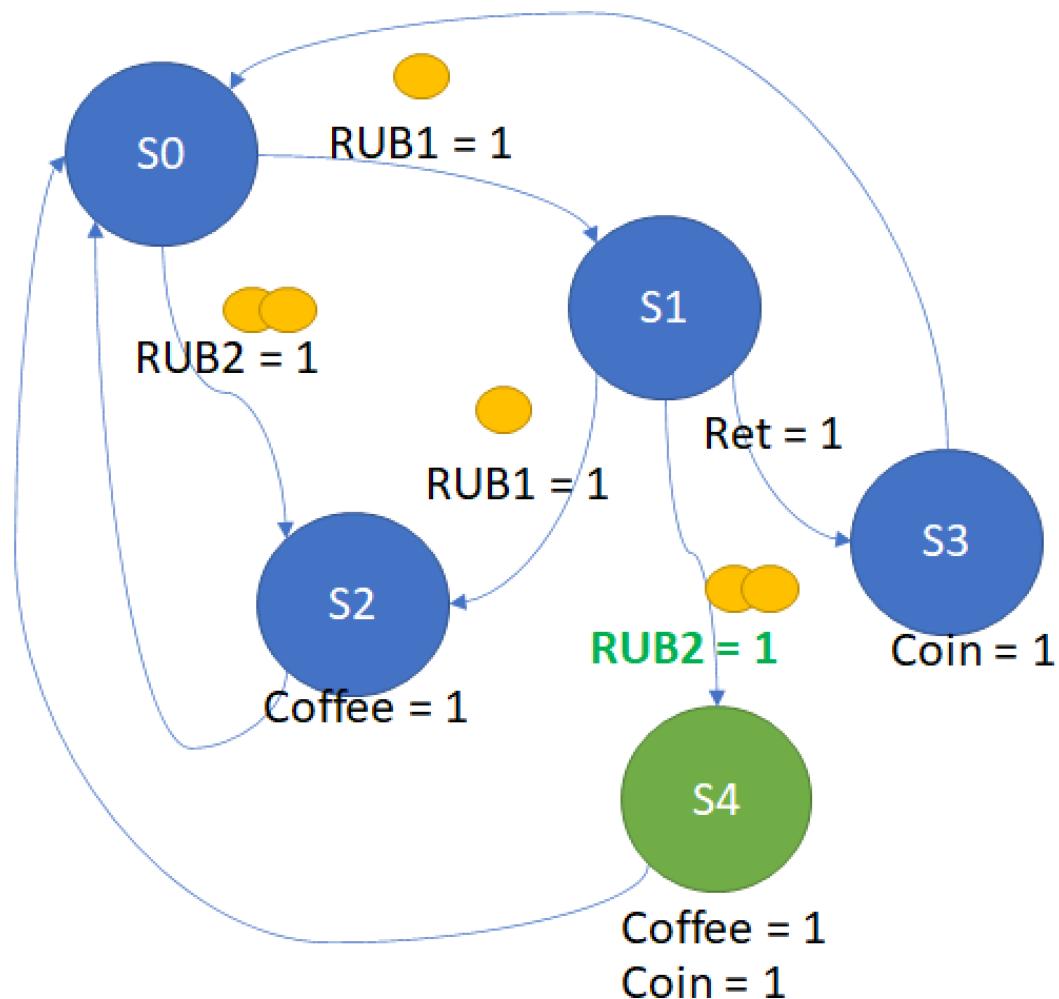


Рисунок 9.4. Граф описания конечного автомата по управлению продажей кофе

Представление КА в виде графа достаточно наглядно, хотя для графов с большим количеством узлов и сложной структурой переходов анализ может стать затрудненным. Визуальное представление позволяет, в частности, быстро отследить две проблемы, которые препятствуют нормальной работе КА:

- тупиковые состояния – состояния, в которые можно попасть, но из которых не ведет ни одной выходной линии;
- недостижимые состояния – состояния, в которые невозможно попасть.

В обоих случаях реализованный КА будет работать хотя и в соответствии с разработанным графиком, но скорее всего некорректно с точки зрения общей

постановки задачи. В общем случае, тупиковых и недостижимых состояний быть не должно. Исключением может быть, например, состояние полной блокировки системы, в которое автомат может попасть один раз, а выход из которого производится путем полного перезапуска, включающего и инициализацию КА. Кроме того, в КА обычно предусмотрен сброс в начальное состояние (в приведенном примере это состояние S0, с которого начинается работа).

9.2. Автоматы Мили и Мура

Существуют две основные разновидности конечных автоматов, называемые автоматами Мили (Mealy) и Мура (Moore). Их основное отличие заключается в том, что значения выходных сигналов для автомата Мили зависят и от состояния автомата, и от значений входных сигналов, а для автомата Мура – только от состояния автомата. Эту разницу иллюстрирует рис. 9.5.

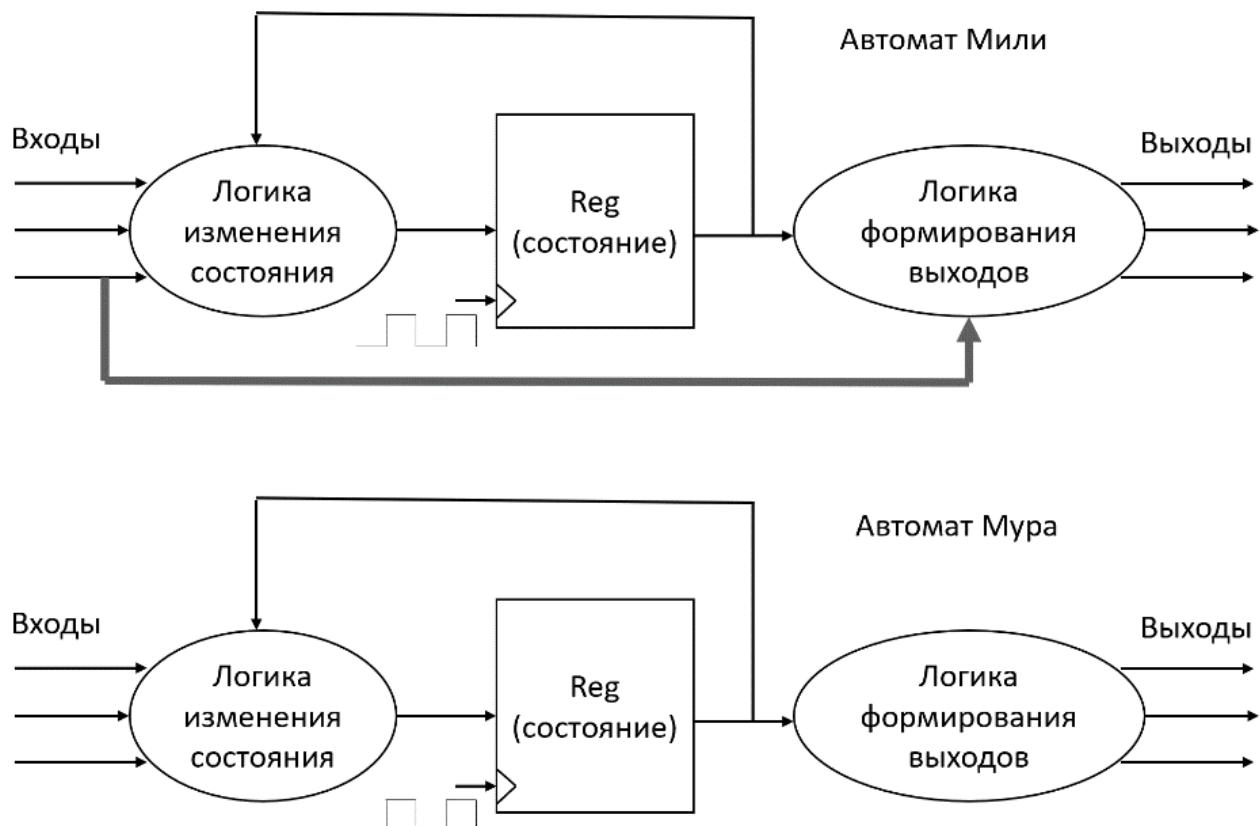


Рисунок 9.5. Структуры автоматов Мили и Мура

На первый взгляд, разница между этими автоматами несущественна и заключается в одной дополнительной связи – между входными и выходными сигналами (на рис. 9.5 она показана жирной линией). Автомат Мили превращается в автомат Мура при отсутствии прямой связи между входами и выходами. Однако недостаток автомата Мили с точки зрения цифровой

схемотехники заключается именно в этой прямой связи. Сквозное прохождение сигналов через ПЛИС имеет большую задержку распространения сигнала, а в автомате Мура входные сигналы приходят в конечном итоге в группу регистров, хранящую состояние автомата. Поэтому в целом автомат Мура может рассчитывать на более высокую производительность.

9.3. Кодирование состояний конечного автомата

Конечные автоматы могут использовать несколько схем кодирования состояний. Очевидной является двоичное кодирование, при котором каждому состоянию приписывается номер, представляемый в двоичном коде. Для N -разрядного регистра возможно представление 2^N состояний автомата, поэтому такой способ не только очевиден, но и наиболее эффективно использует триггеры ПЛИС.

С другой стороны, для проверки того, что КА находится в определенном состоянии, потребуется проверить N -разрядное число. Такая проверка заложена в операторе case, для каждой ветки необходимо проверить регистр состояния на равенство некоторой константе.

Одним из альтернативных вариантов является т.н. one-hot кодирование. Из названия one-hot видно, что при таком кодировании только один разряд регистра состояния равен 1. Таким образом, в N разрядах можно закодировать только N состояний. По сравнению с двоичным кодированием это приведет к расточительному использованию триггеров ПЛИС, поскольку при двоичном кодировании для 256 состояний потребуется 8 триггеров, а для one-hot – 256 триггеров. Однако для проверки каждой из веток оператора case достаточно проверить только один триггер.

Преимущество one-hot кодирования проявляются при небольшом количестве состояний КА. В этом случае рост количества триггеров еще не так существенен, но уже имеется эффект от простой проверки состояния.

Другими вариантами кодирования являются LFSR, счетчик Джонсона, счетчик Грэя или сдвиговый регистр с определенным кодом, не повторяющимся при сдвиге. Такие способы кодирования могут быть эффективны, например, при наличии в КА большого количества последовательных переключений.

Синтезаторы языков описания аппаратуры обычно имеют режим автоматического выбора кодирования конечных автоматов. Достаточно установить соответствующую настройку в Auto, чтобы был подобран оптимальный способ кодирования состояний КА (также можно задать этот способ принудительно).

Часто полезно применять для описания состояний КА перечислимый тип (enumerated type). В перечислимом типе можно указать символические обозначения состояний, которые должен принимать сигнал, как показано в примере.

```
enum reg[2:0] {S0, S1, S2, S3, S4} state;
```

При описании в виде перечислимого типа синтезатор может автоматически подобрать тип кодирования состояний, наиболее подходящий для описанного КА. На практике автоматы с большим количеством состояний кодируются в двоичном виде, а при небольшом количестве используются варианты one-hot, Johnson, LFSR.

9.4. Однопроцессное и трехпроцессное описание конечного автомата

Самым простой способ разработки КА – описать все переходы между состояниями графа внутри одного оператора case.

```
always @ (posedge clk)
begin
  case (state)
    3'b000 : state <= ...; output_reg1 <= ...
    3'b001 :
    3'b010 :
    3'b011 :
    3'b100 :
    default : q = 1'bx;
  endcase
end
```

В каждой ветке оператора case необходимо установить следующее состояние и выходные сигналы. Для автомата, показанного на рис. 9.4, описание модуля на Verilog может выглядеть следующим образом.

```
module coffee_fsm(
  input clk,
  input rub1,
  input rub2,
  input ret,
  output reg coffee,
  output reg coin
);
```

```

reg [2:0] state;

always @ (posedge clk)
begin
    case (state)
        3'b000 : if (rub1)
            begin
                state <= 3'b001;
                coffee <= 1'b0;
                coin <= 1'b0;
            end else
                if (rub2)
                    begin
                        state <= 3'b010;
                        coffee <= 1'b1;
                        coin <= 1'b0;
                    end
        3'b001 : if (rub1)
            begin
                state <= 3'b010;
                coffee <= 1'b1;
                coin <= 1'b0;
            end else
                if (rub2)
                    begin
                        state <= 3'b100;
                        coffee <= 1'b1;
                        coin <= 1'b1;
                    end else
                        if (ret)
                            begin
                                state <= 3'b011;
                                coffee <= 1'b0;
                                coin <= 1'b1;
                            end
        3'b010 : begin
            state <= 3'b000;

```

```

        coffee <= 1'b0;
        coin <= 1'b0;
    end
3'b011 : begin
    state <= 3'b000;
    coffee <= 1'b0;
    coin <= 1'b0;
end
3'b100 : begin
    state <= 3'b000;
    coffee <= 1'b0;
    coin <= 1'b0;
end
endcase
end
endmodule

```

Можно воспользоваться полезным приемом под названием «мультипроцессное описание КА». В примере, показанном выше, все описание, по сути, было сведено к одному оператору case, в отдельных ветках которого описывались все действия, выполняемые в конкретном состоянии. При возрастании сложности автомата его работу можно разделить на три процесса:

- процесс назначения состояния;
- процесс декодирования входных сигналов и определения нового состояния;
- процесс назначения выходных сигналов.

Процесс назначения состояния является наиболее простым. Его буквальный вид показан в следующем листинге.

```

always @ (posedge clk)
begin
    if (reset) state <= initial_state
    else state <= new_state;
end

```

В данном примере появились новые сигналы initial_state (это, по сути, константа, которая требуется для перевода КА в начальное состояние) и

`new_state`. Сигнал `new_state` должен формироваться комбинационной логикой в отдельном процессе. Процесс может быть описан с помощью оператора `case` или с помощью цепочки вложенных операторов `if`.

```
always @ *
begin
  case (state)
    3'b000 : ...
  endcase

always @ *
begin
  if (state = 3'b000) new_state <=
  ...

```

При небольшой сложности автомата можно использовать оператор `assign`.

```
assign new_state = ...
```

При использовании процессов необходимо отслеживать формирование защелок. Если в какой-то ветке оператора `case` новое состояние `new_state` будет не определено, для него будет синтезирована защелка (`latch`) на основе специального режима триггера. Защелка не рекомендуется к применению в синхронном стиле проектирования.

Наконец, в третьем процессе необходимо назначить состояния выходным сигналам. В зависимости от типа автомата (Мура или Мили) потребуется анализ только текущего состояния, или состояния и входных сигналов. Выделение отдельного процесса может привести к более простому описанию, которое проще анализировать и сопровождать. Например, для кофейного КА можно заметить, что включение варки кофе происходит в состояниях `S2` и `S4`, а выдача сдачи – в `S3` и `S4`.

Латентность (`latency`), т.е. дополнительная задержка в тактах на прохождение сигнала по цепи обработки данных может оказаться неприемлемой для многих устройств. Однако можно указать и множество вариантов, когда дополнительные регистры на пути входных или выходных данных не влияют на принципиальную работоспособность схемы. Например, если речь идет о механических переключателях или кнопках, то момент распознавания

автоматом их срабатывания совершенно неважен. Поскольку реакция человека оценивается в доли секунды, лишние десятки наносекунд, обусловленные прохождением сигнала от кнопки через один или несколько регистров, очевидно не будут замечены при наблюдении за работой устройства. Аналогично, сигнальные светодиоды могут срабатывать на несколько тактов позже. Электрические моторы также обладают инерцией, на порядки больше периода тактовой частоты проекта на ПЛИС, поэтому управляющие сигналы для них также могут быть конвейеризованы.

На рис. 9.6 показана иллюстрация, облегчающая понимание трехпроцессного описания конечного автомата. Три показанные компонента КА распределяются по трем процессам в языке описания аппаратуры, что упрощает анализ каждого из этих процессов. Например, процесс назначения нового состояния предельно прост, и заключается в том, что в текущее состояние записывается «новое состояние». Тогда можно сосредоточиться на анализе того, по каким правилам определяется это новое состояние, однако текст описания не перемешан с назначением сигнала state и выходных сигналов.

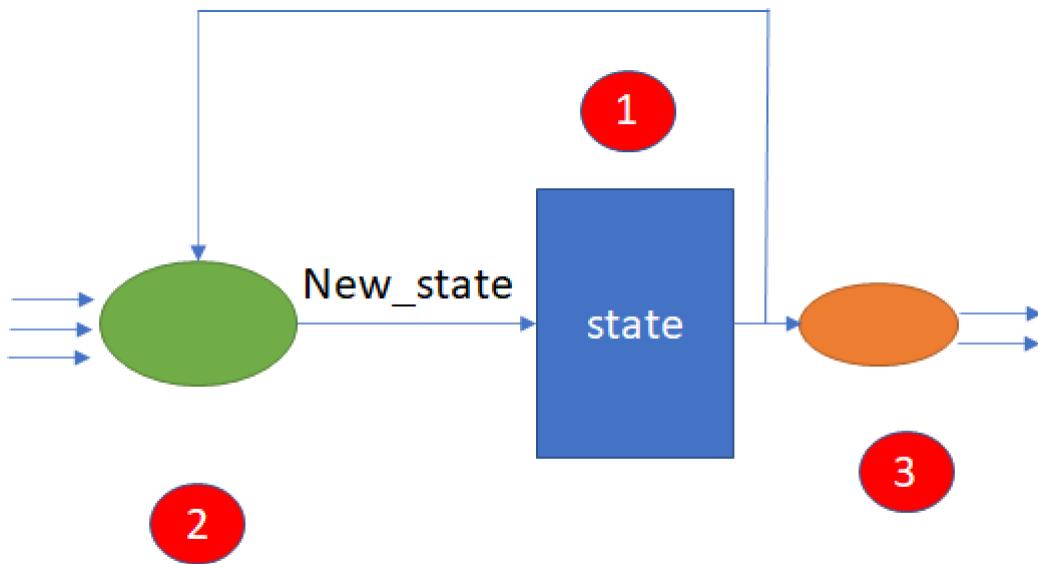


Рисунок 9.6. Схема конечного автомата с компонентами, соответствующими процессам в RTL-описании

При моделировании конечного автомата следует проверять основные сценарии его работы. Временные диаграммы работы КА показаны на рис. 9.7.



Рисунок 9.7. Временные диаграммы работы конечного автомата по управлению продажей кофе

В примере можно видеть, что проверены все основные маршруты графа:

1. Последовательное опускание двух монет по 1 рублю – можно наблюдать появление сигнала coffee.
2. Опускание монеты 2 рубля также приводит к появлению сигнала coffee.
3. Последовательное опускание монет 1 рубль и 2 рубля приводит к одновременному появлению сигналов coffee и coin.
4. Опускание монеты 1 рубль и последующее нажатие кнопки «возврат» приводит к появлению сигнала coin.

Таким образом, моделирование позволяет проверить корректность описания КА и его соответствие тем соображениям, которые имелись у разработчика. Для КА с большим числом состояний и возможных переходов исчерпывающее моделирование может быть затруднено.

9.5. Выводы по разделу

Конечный автомат – эффективное средство реализации схем, последовательно переключающихся в зависимости от внешних управляющих сигналов. Он может быть построен на основе графа, описывающего условия перехода между состояниями. При проектировании автомата можно на ранних стадиях убедиться в его формальной корректности – например, отсутствии противоречивых условий перехода, отсутствии недостижимых и тупиковых состояний.

Можно для краткости представить, что строки программы для процессора соответствуют состояниям конечного автомата. Тогда можно реализовать автомат, который будет, последовательно переходя между состояниями, выполнять в каждом состоянии отдельный оператор.

Существует множество инструментов проектирования и анализа конечных автоматов, в том числе алгоритмы синтеза цифровых схем по RTL-описаниям. В целом конечный автомат соответствует правилам синхронного проектирования и хорошо поддерживается ячейками FPGA.

В зависимости от сложности автомата может использоваться однопроцессное описание, в котором все правила работы находятся внутри одного процедурного блока. Альтернативой является разделение автомата на три процесса (трехпроцессное описание), в котором отдельно описывается запись нового состояния, вычисление нового состояния и назначение выходов

Контрольные вопросы:

1. Могут ли два состояния конечного автомата иметь одинаковые комбинации выходных сигналов? Можно ли заменить их в таком случае одним состоянием?
2. Сколько разрядов требуется для представления 8 состояний КА при двоичном кодировании? При one-hot кодировании?
3. Как с помощью конечного автомата реализовать управление мигающим светодиодом? Тактовая частота внутри схемы существенно больше, чем требуемая частота мигания.
4. Что такое тупиковые и недостижимые состояния КА и почему их следует избегать?
5. Что описывается в отдельных процессах при трехпроцессном описании конечного автомата?

10. ПРОСТЫЕ ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА

10.1. Периферийные устройства в компьютерных системах

Периферийные устройства («устройства ввода-вывода») часто рассматриваются как вспомогательные компоненты вычислительной системы, однако они принципиально важны, если рассматривается взаимодействие системы с окружающим миром. Если устройство не имеет выходных сигналов, оно не оказывает влияния на сопряженные с ним системы, и, следовательно, не выполняет полезных действий. Поэтому результаты вычислений должны быть тем или иным образом переданы в окружающий мир, в виде изображений, звука, электрических или оптических сигналов и т.д.

Поскольку результаты вычислений могут быть переданы внешним устройствам различными способами, нерационально жестко связывать вычислительные модули с модулями для подключения к внешним системам. Поэтому существует множество интерфейсов подключения.

Согласно определению, интерфейс – это «совокупность средств и правил, обеспечивающих взаимодействие устройств вычислительной машины».

С понятием интерфейса тесно связано понятие протокола. Под протоколом понимается набор правил, регламентирующих использование интерфейса. Например, если в интерфейсе указано, что соединение выполняется двумя сигналами, то можно использовать эти сигналы разными способами. Конкретный способ использования интерфейса (например, очередность подачи сигналов, способ кодирования данных и т.д.) и будет определять протокол.

В современных вычислительных системах используется множество разновидностей периферийных устройств. Они существенно отличаются по назначению, сложности реализации, скорости обмена данными, функциональным возможностям и прочим параметрам. Некоторые устройства являются распространенными, и при этом достаточно простыми для самостоятельной реализации.

10.2. UART

Аббревиатура UART обозначает Universal Asynchronous Receiver-Transmitter – «универсальный асинхронный приемопередатчик». Часто используется несколько терминов, которые так или иначе относятся к UART – например, может упоминаться RS-232 или СОМ-порт. Термин UART относится к аппаратному устройству, реализующему обмен данными. Физическая передача данных может производиться с помощью различных электрических стандартов,

например, RS-232, RS-485 и др. Понятие COM-порт (TTY в ОС Linux) относится к файлу, связанному в операционной системе с устройством UART.

RS-232 долгое время был стандартным последовательным интерфейсом для персональных компьютеров. Он очень прост как с точки зрения схемотехники, так и с точки зрения поддержки программного обмена, поэтому даже сегодня существует множество преобразователей типа USB-UART, BlueTooth-UART, Ethernet-UART и др. Использование в устройстве UART во многих случаях вполне обосновано, поскольку можно быстро подключить его к внешнему компьютеру.

Варианты реализации UART показаны на рис. 10.1. Верхний рисунок показывает варианты соединительных кабелей, применявшимися в ранних вариантах UART. Ниже показаны модули преобразователей USB-UART и BlueTooth-UART.



Рисунок 10.1. Варианты реализации UART

Для асинхронной передачи по интерфейсу RS-232 достаточно всего двух сигнальных линий – TxD (Transmit Data) и RxD (Receive Data). Эти сигналы соединяются перекрестно – сигнал Tx одного устройства подается на сигнал Rx другого, и наоборот. Очевидно, в соединительном кабеле должна быть и линия земли. Сопряжение множества устройств таким образом невозможно, с помощью UART можно соединить только два устройства. Для электрического сопряжения сигналов потребуются стандартные микросхемы преобразования уровней RS-232 (лежащих в диапазоне $-12 \dots +12$ В) в логические уровни ТТЛ/КМОП. Такие микросхемы выпускаются многими производителями, обычно в их обозначении есть число 232 (например, ADM232). После преобразования к выводам ПЛИС окажутся подключены два сигнала, которые можно обозначить как rx и tx.

Протокол передачи UART представлен на рис. 10.2. Передача начинается установкой низкого логического уровня на входе приемника. Низкий уровень сохраняется на время длительности одного бита, который называется стартовым. Далее передаются биты данных (число которых зависит от настроек протокола). В последовательных портах РС передача начинается с младшего бита. После передачи последнего бита данных вводится бит четности, формируемый по позитивной или негативной четности в зависимости от настроек протокола. Теми же настройками передача бита четности может быть отключена. В завершение передается стоповый бит, после которого линия остается в состоянии логической единицы вплоть до следующей посылки данных. Длительность «стоп-бита» в действительности может быть равна длительности 1, 1,5 или 2 бит. Он необходим для разделения передаваемых порций данных. Популярной настройкой является 8N1 (8 бит данных, нет бита четности, стоп-бит передается в течение 1 бита данных).

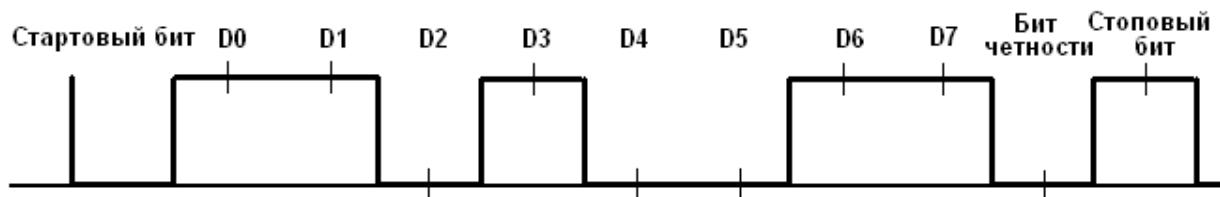


Рисунок 10.2. Временные диаграммы передачи данных по протоколу UART

В UART нет встроенного способа передать длительность одного бита. Приемник и передатчик должны быть настроены на одну и ту же скорость обмена данными. Традиционно использовался ряд скоростей обмена данными –

150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 115200 бит/с. Единица измерения скорости бит/с также называется «бод» (baud). В ряде устройств применяются и другие значения скоростей (например, 57600 или 921600). Популярными являются скорости 115200 и 9600, которые часто указываются в примерах проектов, работающих с UART.

Передача данных начинается со стартового бита, причем время ожидания в общем случае не определено. Разумным решением будет ввод в состав конечного автомата состояния «ожидание начала». Из этого состояния его может вывести появление логического нуля на входе данных rx. Далее необходимо подождать начала передачи младшего бита данных. В терминах КА это означает последовательный переход между вспомогательными состояниями до тех пор, пока не будет достигнута середина бита D0.

Контроллер UART, осуществляющий прием данных, показан в следующем листинге.

```
module uart(
    input rx,
    output tx,
    input clk,
    input reset,
    output reg [7:0] rdata,
    output reg received,
    input [7:0] tdata,
    input transmit
);

`define FCLK 50000000
`define UART_BAUDRATE 115200
reg [31:0] rxstate = 0;
reg [31:0] txstate = 0;

reg [7:0] ridata;

always @ (posedge clk)
begin
    if (reset)
        rxstate <= 0;
```

```

    received <= 1'b0;
else
  case(rxstate)
    0 : if (~rx) rxstate <= 1;
    3 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[0] <= rx; rxstate <= rxstate + 1;
      end
    5 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[1] <= rx; rxstate <= rxstate + 1;
      end
    7 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[2] <= rx; rxstate <= rxstate + 1;
      end
    9 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[3] <= rx; rxstate <= rxstate + 1;
      end
    11 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[4] <= rx; rxstate <= rxstate + 1;
      end
    13 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[5] <= rx; rxstate <= rxstate + 1;
      end
    15 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[6] <= rx; rxstate <= rxstate + 1;
      end
    17 * `FCLK / 2 / `UART_BAUDRATE :
      begin
        ridata[7] <= rx; rxstate <= rxstate + 1;
      end
    9 * `FCLK / `UART_BAUDRATE :
      begin

```

```

        received <= 1'b1; rdata <= ridata;
        rxstate <= rxstate + 1;
    end
    9 * `FCLK / `UART_BAUDRATE + 1 :
begin received <= 1'b0; rxstate <= 0; end
default : rxstate <= rxstate + 1;
endcase
end

endmodule

```

Приведенный модуль реализует конечный автомат, осуществляющий чтение данных, приходящих из внешнего устройства UART. Интерфейс модуля содержит сигналы и для приемника, и для передатчика, чтобы можно было легко модифицировать его, добавив аналогичный КА для передатчика.

Описание содержит некоторые практические приемы проектирования. Например, в модуле определены параметры FCLK и UART_BAUDRATE (с помощью директивы define). Тогда выражение `3 * `FCLK / 2 / `UART_BAUDRATE` будет всегда соответствовать моменту времени, когда наблюдается середина бита 0 в принимаемой последовательности. Здесь следует сделать замечание, что выражение не будет синтезироваться в вычислительные узлы, так как в его составе находятся только константы, известные на этапе синтеза. Фактически, в результате получится число, которое и будет подставлено в оператор case.

Применив в описании модуля параметры, определенные с помощью define, можно быстро модифицировать автомат так, чтобы он работал с другой скоростью UART или с другой тактовой частотой.

Разработка передатчика UART может быть проведена самостоятельно, основываясь на примере автомата приемника. Передатчик должен использовать собственное состояние автомата txstate, в котором он будет находиться до тех пор, пока не получит внешний сигнал transmit. Тогда автомат может начинать последовательные переходы между состояниями, сначала формируя старт-бит (он всегда равен 0), а затем передавая биты 0 – 7, которые удерживаются на входе модуля tdata. Передача завершается удержанием выхода tx в состоянии 1 в течение 1, 1,5 или 2 бит.

10.3. ШИМ

Аббревиатура ШИМ расшифровывается как «широкотно-импульсная модуляция». Она аналогична термину PWM (Pulse-Width Modulation). С помощью сигнала, модулированного таким способом, можно изменять уровень мощности внешнего устройства с небольшими потерями энергии. Пример сигнала такого типа показан на рис. 10.3.

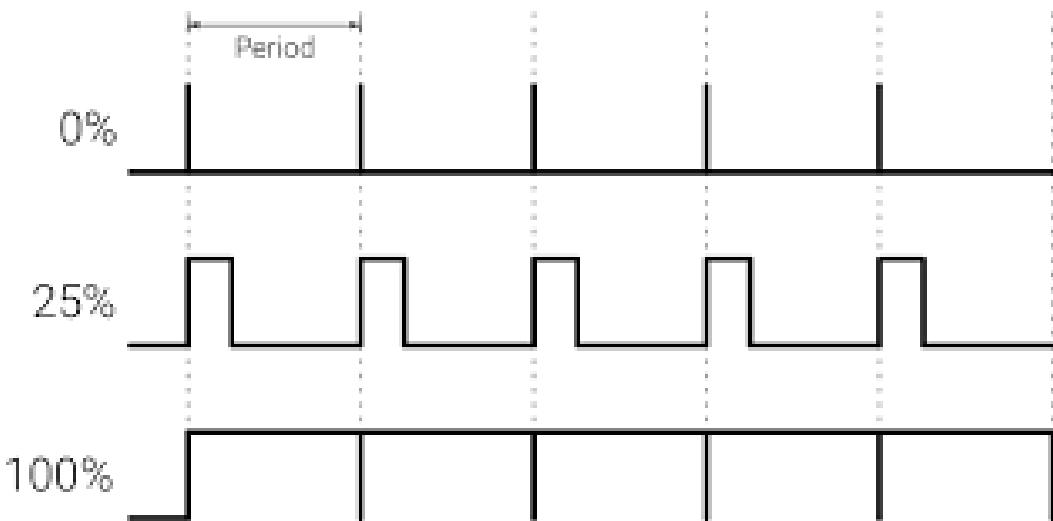


Рисунок 10.3. Принцип управления уровнем выходного сигнала при широтно-импульсной модуляции

На рисунке видно, что установка соответствующего уровня сигнала производится не с помощью изменения уровня аналогового напряжения, а включениями логической единицы на соответствующий процент времени. Такой способ управления не заменяет формирование аналогового сигнала, однако во многих случаях не только приемлем, но и эффективен с точки зрения к.п.д. управления.

Например, если требуется подать на мощное внешнее устройство только 25% напряжения, аналоговая система управления включит управляющий элемент в такой режим, чтобы он погасил 75% напряжения. Это напряжение будет приложено к самому элементу (обычно транзистору), а умножив его на протекающий ток, можно узнать тепловую мощность, которая будет неоправданно потрачена на нагрев управляющего транзистора. В итоге получится, что экономии энергии достичь не удается, и более того, управляющий транзистор чрезмерно нагревается, что сокращает его срок службы.

При управлении с помощью ШИМ переключающий элемент (часто в этом качестве используется полевой транзистор) находится либо в открытом состоянии (он пропускает ток, при этом падение напряжения на нем мало,

значит, мала и тепловая мощность), либо в закрытом состоянии (тогда его сопротивление велико и протекающий ток близок к нулю). В любом состоянии мощность мала (потому что либо $U \approx 0$, либо $I \approx 0$). Если подключенная нагрузка может работать в таком режиме, можно удобно управлять усредненным уровнем сигнала на ней. Пример схемы подключения нагрузки показан на рис. 10.4.

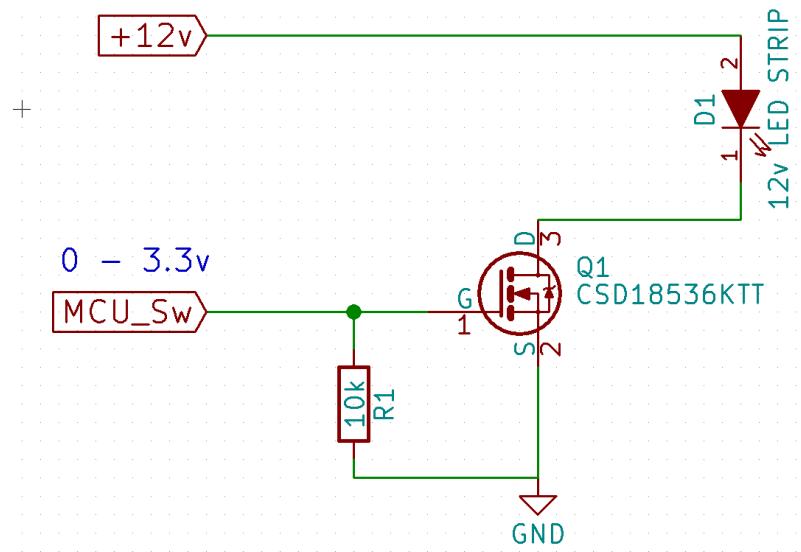


Рисунок 10.4. Управление мощной нагрузкой с помощью полевого транзистора, управляемого ШИМ-сигналом

Обычно ШИМ-управление можно эффективно применять для нагревателей, электромоторов, осветительных устройств и подобных им. Тепловая инерция нагревательного элемента очень высока, и быстрое переключение тока не приведет к такому же быстрому нагреву и охлаждению.

Кроме этого, некоторые устройства используют ШИМ в качестве входного интерфейсного сигнала, самостоятельно формируя требуемые сигналы управления. Например, на рис. 10.5 показан компьютерный вентилятор, широко используемый для охлаждения системных блоков. Управление скоростью вращения такого вентилятора производится с помощью ШИМ (входы Control, отмеченные на разъемах синим цветом).

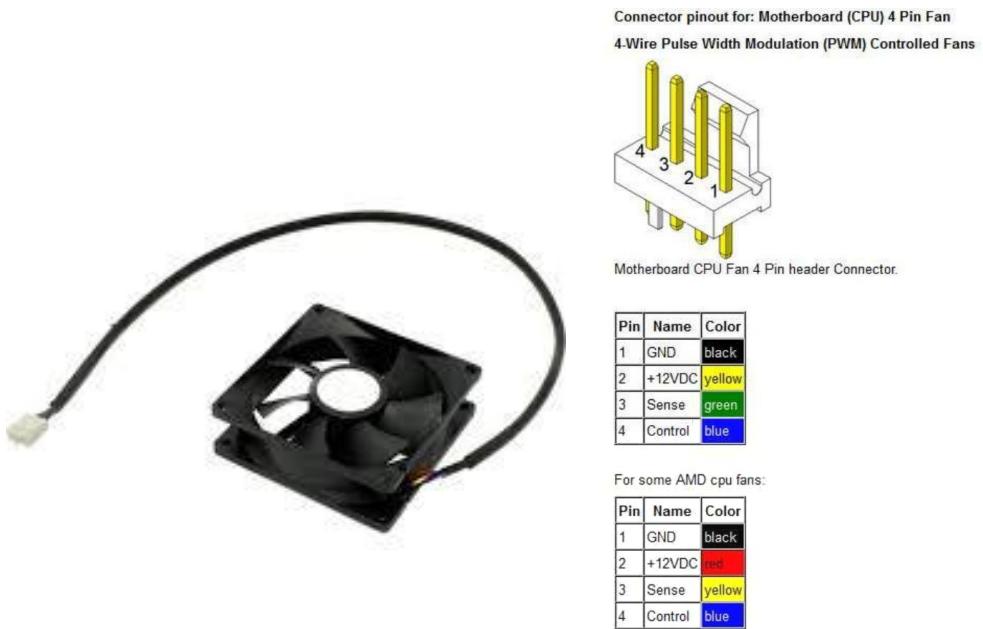


Рисунок 10.5. Компьютерный вентилятор с встроенным контроллером, управляемым ШИМ

Часто транзисторы с сопутствующими компонентами выпускаются в виде внешних модулей, пример которого показан на рис. 10.6. Такие модули имеют цифровые управляющие сигналы и обеспечивают переключение токов до единиц или десятков ампер.

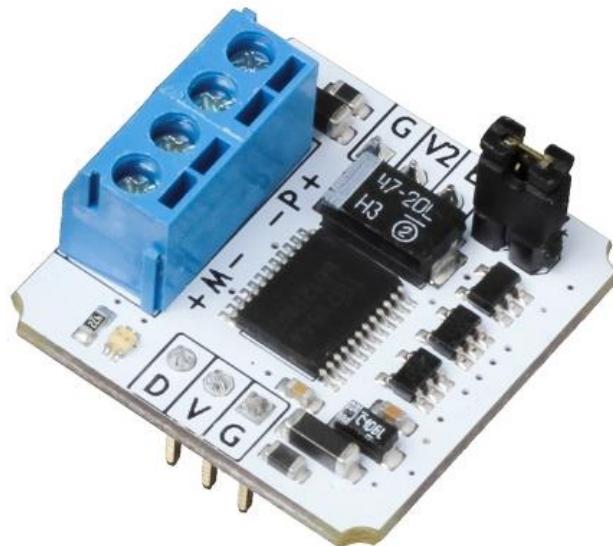


Рисунок 10.6. Модуль управления силовой нагрузкой на основе транзисторов, управляемых ШИМ

Для управления электромоторами с изменением направления движения используют так называемые мостовые схемы. Принцип ее применения показан на рис. 10.7.

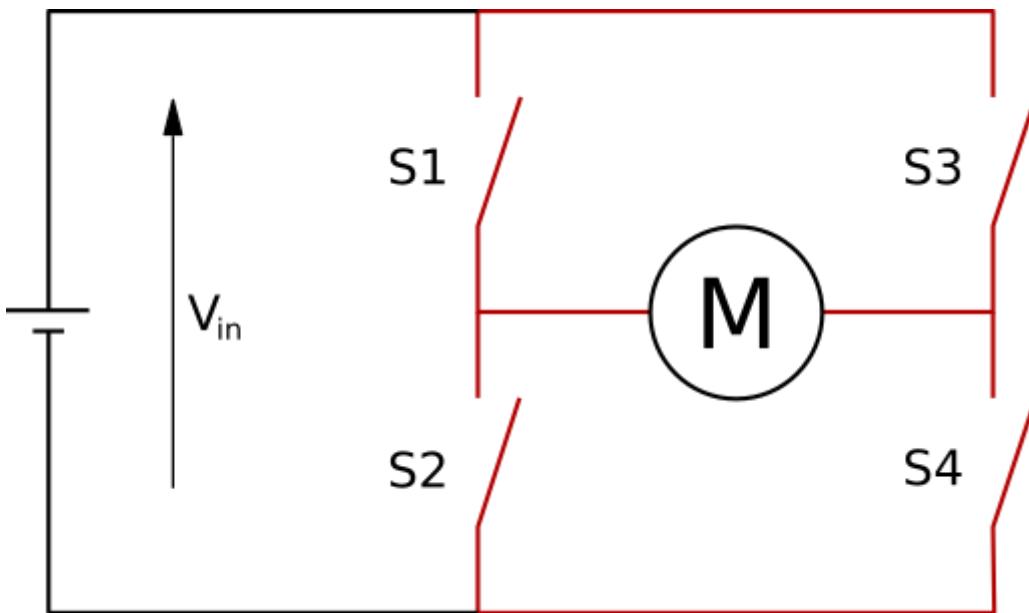


Рисунок 10.7. Управление электромотором с помощью мостовой схемы

В этой схеме можно изменять направление тока, протекающего через мотор, обозначенный символом М. Если замкнуты ключи S1 и S4, направление протекания тока оказывается слева направо, а если замкнуть S2 и S3 – справа налево. При этом один или оба ключа могут управляться с помощью ШИМ, что позволяет не просто изменять направление тока, но и регулировать его среднюю величину.

Важным вопросом является выбор частоты следования импульсов в ШИМ-сигнале. В ряде случаев могут быть требования к ее точному значению. Например, в компьютерных вентиляторах используется частота 25 кГц, а в некоторых шаговых двигателях 50 Гц.

С точки зрения мощной нагрузки, имеются два соображения, определяющих границы тактовой частоты.

Если частота слишком низкая, объект управления может не усреднить его характеристики. Например, мигание светодиода с частотой 1 Гц будет замечено человеком, поэтому для создания эффекта плавного управления яркостью необходимо повысить эту частоту выше предела аккомодации (25 Гц). Обмотки электродвигателей также лучше усредняют высокочастотные сигналы.

С другой стороны, если требуется установить частоту импульсов ШИМ 500 Гц, и при этом изменять ширину с шагом в 0,1%, потребуется 1000 тактов на каждый импульс. Поэтому тактовая частота будет составлять $500 \times 1000 = 500$ кГц. Конкретно это значение не представляет проблем для реализации, однако

если частота ШИМ будет составлять сотни кГц, тактовая частота может потребоваться около 10 или 100 МГц.

Слишком высокая частота следования импульсов ШИМ создает проблему не только в виде слишком высокой тактовой частоты. Мощные транзисторы не могут переключаться мгновенно, поэтому между состояниями «нет тока» и «нет падения напряжения» имеется период, когда сопротивление транзистора находится между очень маленьким и очень большим значениями. В этот период есть и ток, протекающий через транзистор, и падение напряжения на нем, поэтому на транзисторе рассеивается тепловая мощность. Чем чаще происходит переключение транзистора, тем чаще выделяется эта порция мощности. Слишком высокая частота переключения приведет к перегреву транзистора, а кроме того, часть энергии системы будет непроизводительно потрачена на нагрев. Таким образом, слишком высокая частота ШИМ уменьшает к.п.д. регулятора.

Конкретное значение частоты ШИМ выбирается исходя из объекта регулирования, применяемых транзисторов и возможностей цифрового контроллера. В современных блоках питания для компьютеров и смартфонов часто используются импульсные стабилизаторы, построенные с использованием ШИМ. В них используются частоты от 75 кГц до 300 кГц и более, в зависимости от мощности и применяемого цифрового контроллера.

Пример простого контроллера ШИМ показан ниже.

```
module pwm(
    input clk,
    input [7:0] d,
    output pwm
);

reg [7:0] cnt = 0;

always @ (posedge clk)
    cnt <= cnt + 1;

assign pwm = (d > cnt) ? 1 : 0;

endmodule
```

В приведенном примере для простоты сделаны некоторые допущения.

1. Использован простой 8-разрядный счетчик на базе регистра cnt. Он будет считать от 0 до 255, а потом переполнится естественным образом. Это не вполне

удобный способ, поскольку так можно описать только счетчики, считающие до целой степени двойки. Следует использовать условный оператор, проверяющий достижение счетчиком максимального значения и переходящий к 0.

2. Если управляющий регистр также имеет 8 разрядов, как показано в примере, то выход ШИМ можно отключить ($d = 0$, $cnt = 0$, $d > cnt$ не выполняется). Однако нельзя включить выход в 1 на все время ($d = 255$, $cnt = 255$, $d > cnt$ также не выполняется), поэтому при установке максимального значения на выходе будут наблюдаться кратковременные интервалы нуля (1 такт из 256).

В управлении нагрузкой нецелесообразно чрезмерно повышать разрядность ШИМ. Например, уже 16-разрядный регистр даст возможность задавать от 0 до 65535 тактов в течение одного импульса. Это слишком много для непосредственного наблюдения за изменением яркости светодиода, а для мощной нагрузки (нагревателя, электромотора) будут также играть роль вариации параметров самих объектов управления и мощных транзисторов, которые в процессе работы изменяются существенное, чем такой маленький шаг управления.

В целом, широтно-импульсная модуляция является эффективным способом управления внешними устройствами. Такие периферийные устройства часто размещаются в современных микроконтроллерах.

10.4. SPI

Последовательный интерфейс, SPI (Serial Peripheral Interface) широко распространен в цифровых устройствах. Его достоинством является небольшое число линий (4 или в ряде случаев 3) в сочетании с относительно высокой скоростью передачи данных. Часто эта скорость ограничена требованиями подключаемой внешней микросхемы, а встречающимися в документации максимальными значениями являются 5, 20, 80 и 105 Мбит/с. Шина является синхронной (т.е. имеет явный сигнал синхронизации), что и обеспечивает более высокую по сравнению с UART скорость передачи.

Шина SPI подразумевает наличие ведущего устройства (master) и нескольких ведомых (Slave). Ведущее устройство формирует тактовый сигнал, подаваемый на все ведомые устройства. Шина имеет следующие сигналы:

- SCLK – тактовый сигнал, выход ведущего и вход ведомых устройств;
- MOSI (Master Out, Slave In) – выход данных для ведущего и вход для ведомых устройств;
- MISO (Master In, Slave Out) – вход данных для ведущего и выход для ведомых устройств;

– CS (Chip Select) – выбор ведомого устройства.

Сигналы CS являются индивидуальными для каждого из ведомых устройств, в то время как остальные объединяются. Может показаться, что это образует проблему для сигнала MISO, который объединяет выходы нескольких ведомых устройств. Однако только одно ведомое устройство может быть активным, а неактивные обязаны перевести свой сигнал MISO в состояние высокого импеданса. За активизацию ведомых устройств отвечают сигналы CS, имеющие активный низкий уровень. Ведущее устройство обязано следить, чтобы было активировано не более одного ведомого устройства. Схема подключения нескольких устройств SPI показана на рис. 10.9.

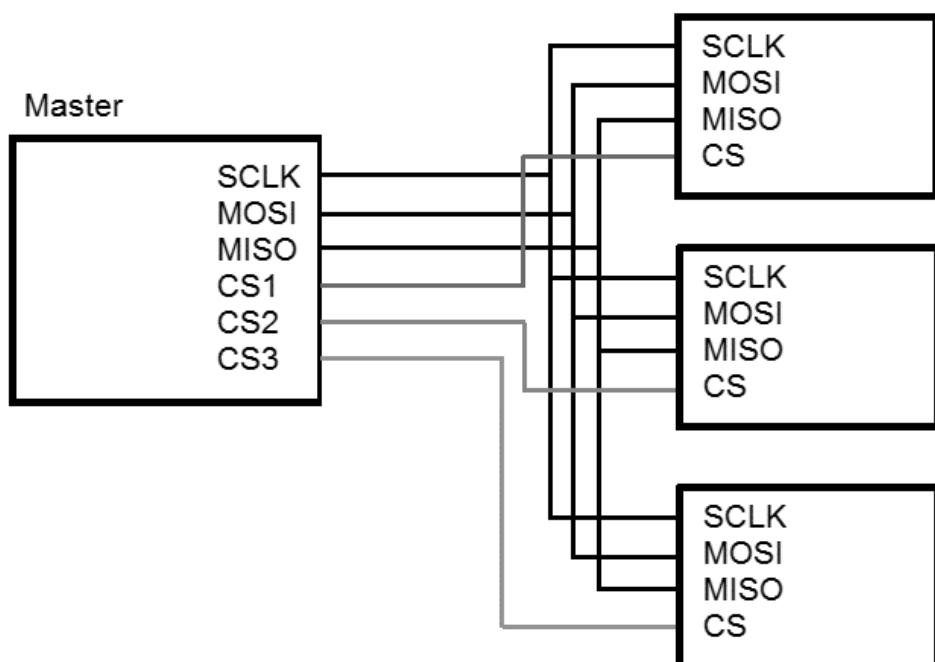


Рисунок 10.8. Подключение нескольких ведомых устройств к ведущему с помощью интерфейса SPI

Временные диаграммы работы SPI показаны на рис. 10.9. Начало передачи идентифицируется низким уровнем сигнала CS. Далее в течение нескольких тактов сигнала SCLK передаются данные, причем в зависимости от подключаемой микросхемы ведомое устройство может как передавать свой выходной буфер, так и не передавать. Например, при подключении флеш-памяти с интерфейсом SPI после передачи мастером адреса ведомое устройство памяти начинает передавать содержимое ячейки с запрошенным адресом.

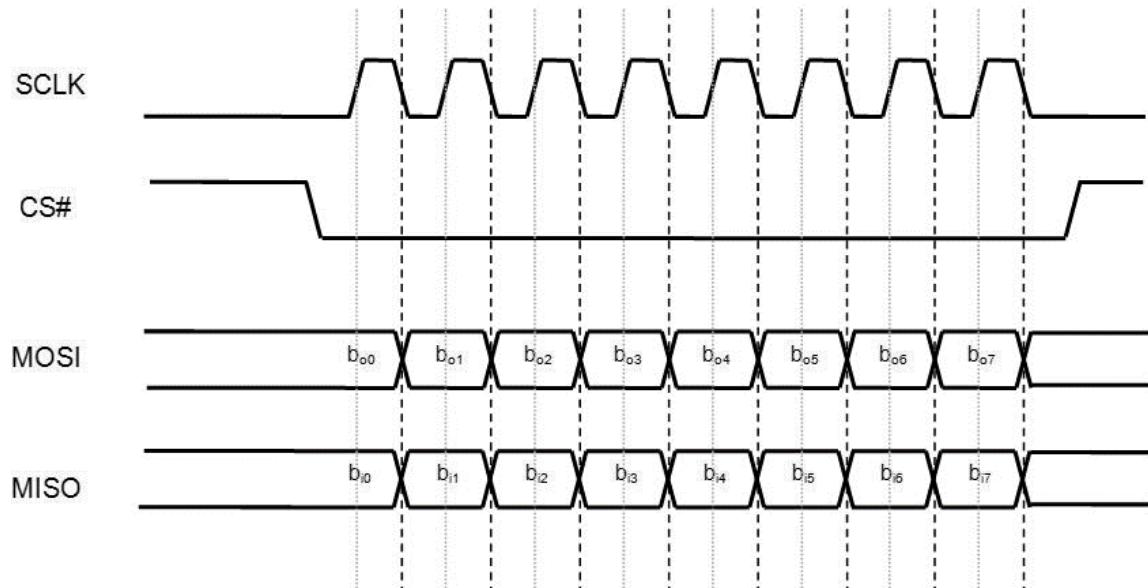


Рисунок 10.9. Временные диаграммы интерфейса SPI

В SPI нет единственного протокола передачи данных. Каждый производитель микросхем может устанавливать размеры передаваемых пакетов и значения команд по своему усмотрению.

На рис. 10.10 показаны широко распространенные микросхемы флеш-памяти с интерфейсом SPI. Такие микросхемы выпускаются многими производителями, в настоящий момент доступны микросхемы с объемом 1 - 128 Мбит.

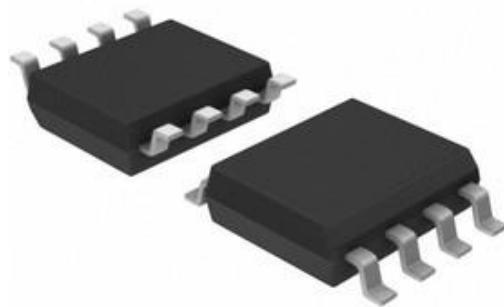


Рисунок 10.10. Микросхемы флеш-памяти с интерфейсом SPI

Формально, каждый производитель может установить собственный протокол обмена данными. Однако многими производителями поддерживаются унифицированные протоколы обмена. Пример такого протокола показан на рис. 10.11.

На рисунке видно, что после начала обмена данными, запускаемым переводом сигнала CS в низкий уровень, первый же фронт тактового сигнала вызывает передачу старшего бита данных в микросхему памяти. Контроллер флеш-памяти начинает работу с приема команды, занимающей один байт. Некоторые команды ожидают передачу операндов после кода команды. Например, команды чтения с кодами 0Bh или 03 ожидают после кода команды передачи 32 бит адреса, с которого надо начать чтение. В приведенном примере флеш-память использует только 24 бита. После передачи адреса последующие фронты тактового сигнала будут вызывать появление последовательных битов данных с указанного адреса.

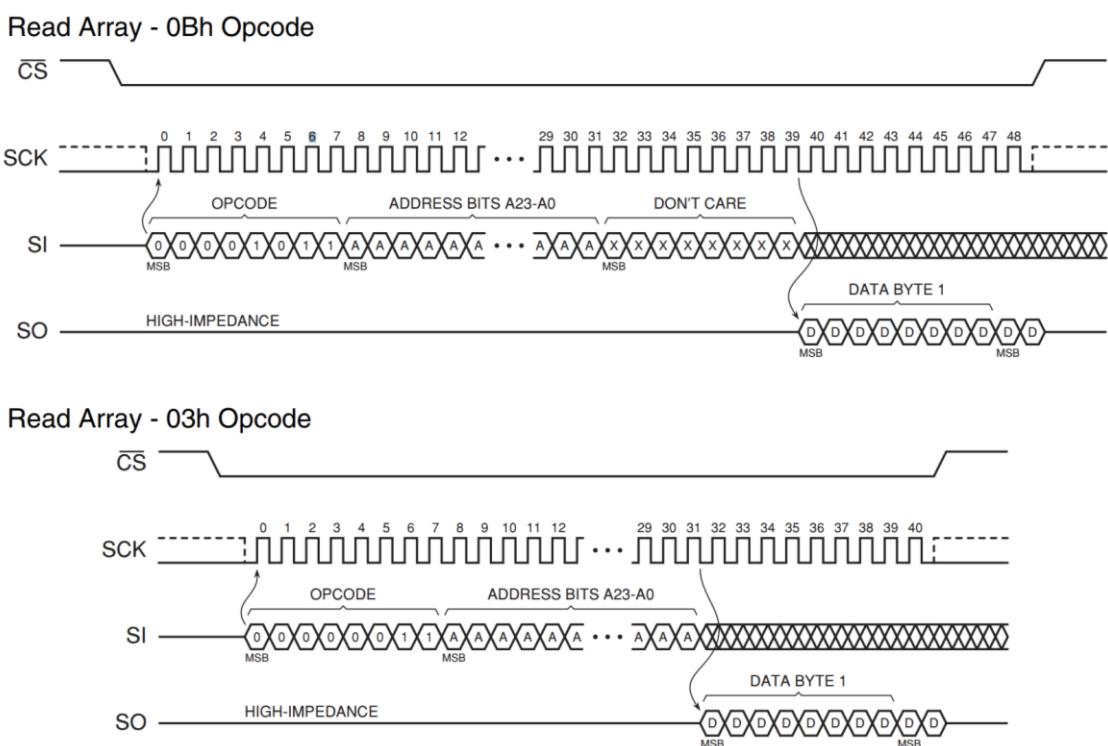


Рисунок 10.11. Пример обмена данными с микросхемой флеш-памяти с интерфейсом SPI

Другим примером являются аналого-цифровые преобразователи (АЦП). Эти микросхемы формируют на выходе цифровой код, пропорциональный поданному на вход аналоговому напряжению. Таким образом, они ориентированы на передачу данных только в одном направлении – из АЦП к подключенному устройству, тогда как микросхемы флеш-памяти поддерживают чтение, запись, стирание, чтение идентификаторов и другие команды.

Пример передачи данных из АЦП MCP3201 показан на рис. 10.12.

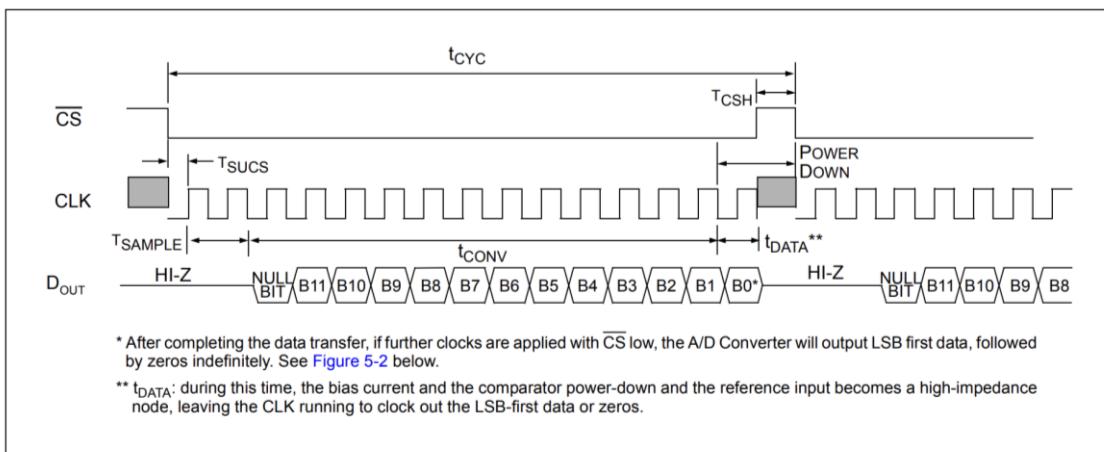


FIGURE 5-1: Communication with MCP3201 device using MSB first Format.

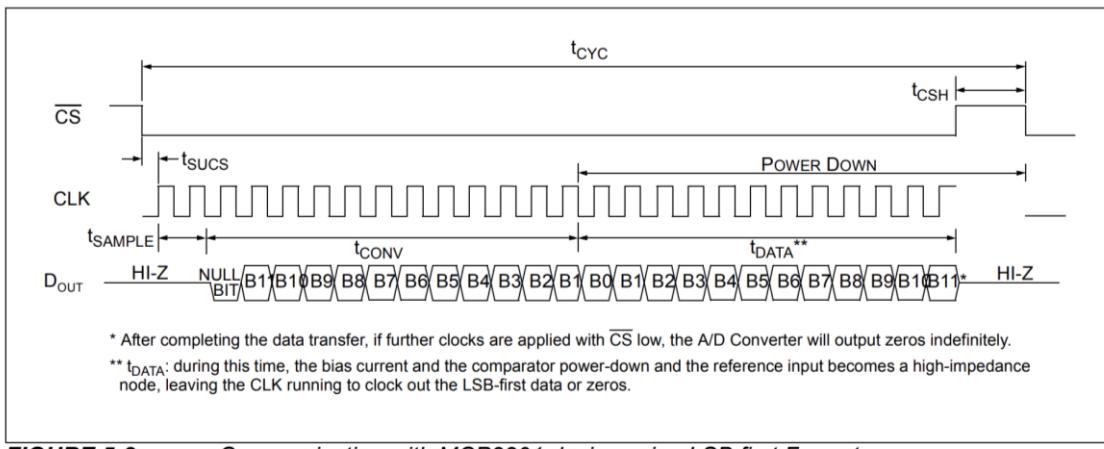


FIGURE 5-2: Communication with MCP3201 device using LSB first Format.

Рисунок 10.12. Пример обмена данными с аналого-цифровым преобразователем с интерфейсом SPI

Для формирования выходного тактового сигнала для SPI рекомендуется использовать специальный прием, обозначаемый в САПР как *clock forwarding*. Англоязычный вариант термина приводится потому, что САПР могут выявлять попытки подключить внутренний тактовый сигнал напрямую к выводу микросхемы и выводят соответствующее сообщение, в котором указывают на необходимость вывести сигнал с помощью «*clock forwarding*». Суть этого схемотехнического приема показана на рис. 10.13.

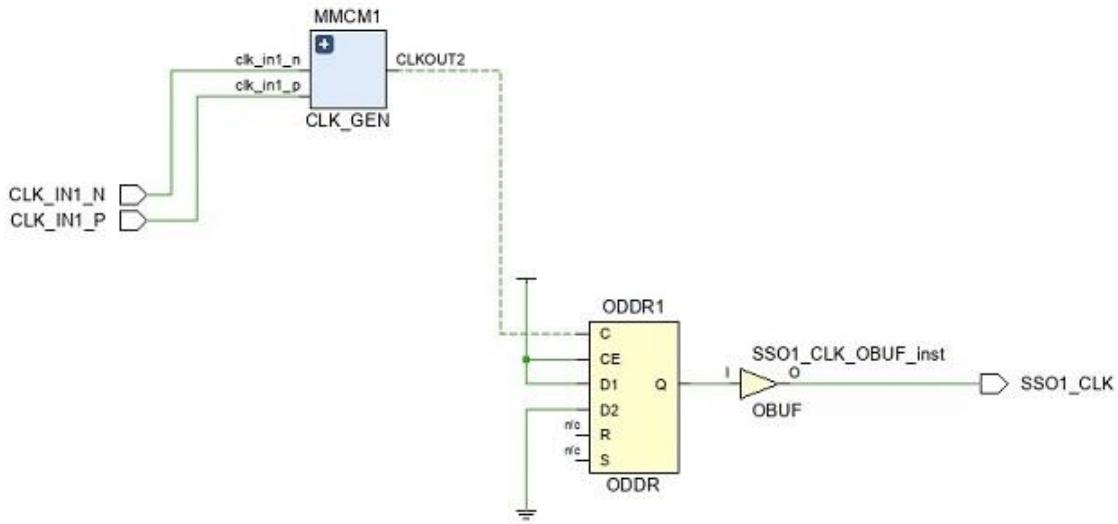


Рисунок 10.13. Формирование тактового сигнала с помощью приема clock forwarding

Схема работает следующим образом. Триггер ODDR имеет два входа данных – D0 и D1, которые связаны с тактовыми входами C0 и C1 соответственно. В примере видно, что на D0 подана логическая единица, а на D1 – логический ноль. Поэтому по фронту тактового сигнала clk, который также подан на C0, триггер запишет логическую единицу (с D0), а по спаду clk, который после инвертирования окажется фронтом для C1, будет записан логический ноль с D1. Таким образом, на выходе окажется не сам тактовый сигнал, но данные, которые будут меняться точно так же. Этот прием может быть применен для различных семейств ПЛИС, однако для Xilinx Spartan-6 он является обязательным.

В режиме slave может показаться, что единственным правильным для ПЛИС решением будет подача тактового сигнала SPI на один из так называемых *clock-capable* входов – т.е. входов, имеющих специальные буферы для тактовых сигналов. Однако такой прием повлечет за собой необходимость передачи данных, захваченных по одному тактовому сигналу, в регистр, тактируемый другим (системным) тактовым сигналом. Это потребует реализации схемы защиты от метастабильности, что само по себе не является непреодолимой проблемой, однако для подобных ситуаций существует и другой эффективный прием.

Ввиду того, что тактовый сигнал SPI предполагается меньшей частоты, чем системный тактовый сигнал внутри ПЛИС, можно произвести выделение положения фронта тактового сигнала с помощью последовательного высокочастотного опроса линии SCLK, что проиллюстрировано на рис. 10.14.

Если тактировать цепочку триггеров D0, D1 высокочастотным тактовым сигналом CLK, то эта цепочка будет захватывать последовательные состояния линии SCLK, причем триггер D1 будет содержать предыдущее состояние линии SCLK, а D0 – текущее состояние. Появление фронта сигнала на линии SCLK приведет к тому, что старое состояние линии, хранящееся в D1, будет равно 0, а новое, прочитанное в D0, станет равно 1. Если условие $D0 == 0 \& D1 == 1$ использовать в качестве разрешения работы, сигналы данных интерфейса SPI смогут быть приведены к тактовой сети CLK. Такой подход требует конвейеризации и для сигнала данных, поскольку фронт сигнала SCLK выявляется с задержкой.

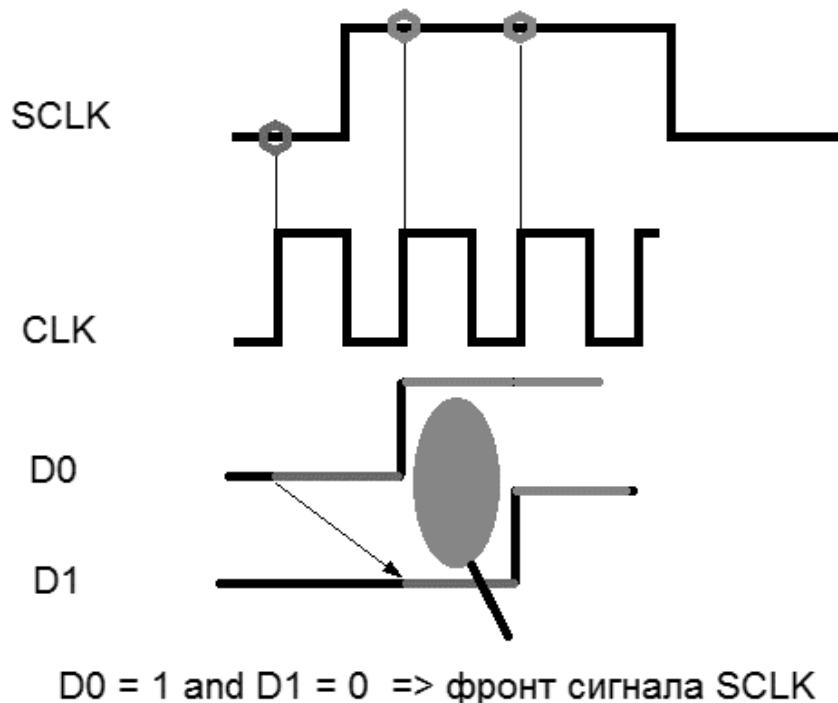


Рисунок 10.14. Принцип выделения фронта тактового сигнала с использованием внутреннего тактового сигнала

Контроллеры SPI могут быть дополнены другими элементами, облегчающими их применение. Например, использование FIFO позволит реже обращаться к контроллеру для передачи сигналов управления, поскольку большой буфер позволит накопить принимаемые данные или обеспечить заполнение буфера для передачи с их последующей отправкой с более медленным темпом.

Для вычислительных систем нет специальных ограничений на количество контроллеров SPI. Несмотря на то, что к одному набору линий можно подключить множество ведомых контроллеров (с индивидуальным сигналом CS

для каждого), это не означает, что ведущий контроллер должен быть единственным. Дополнительные контроллеры могут быть установлены не только для обеспечения одновременной работы, но и для повышения помехоустойчивости сигналов на печатной плате.

10.5. Интерфейс I2C

Интерфейс I2C (Inter-Integrated Circuit) был разработан компанией Philips в 1980-х годах и предусматривал несложное подключение множества периферийных устройств к одной шине. Используются всего два сигнала, SCL (тактовый сигнал) и SDA (двунаправленный сигнал данных). Все устройства на шине подключаются к этим линиям, однако только ведущее устройство может управлять линией SCL, и только одно (ведущее или одно из ведомых) может быть подключено к линии данным.

Важно иметь в виду, что сигнал SDA должен быть реализован в схеме как двунаправленный (inout). Если устройство неактивно, его выходной буфер должен быть в состоянии высокого импеданса.

В отличие от SPI, выбор устройства производится с помощью стандартизованного протокола обмена. На рис. 10.15 показаны временные диаграммы обмена данными по I2C.

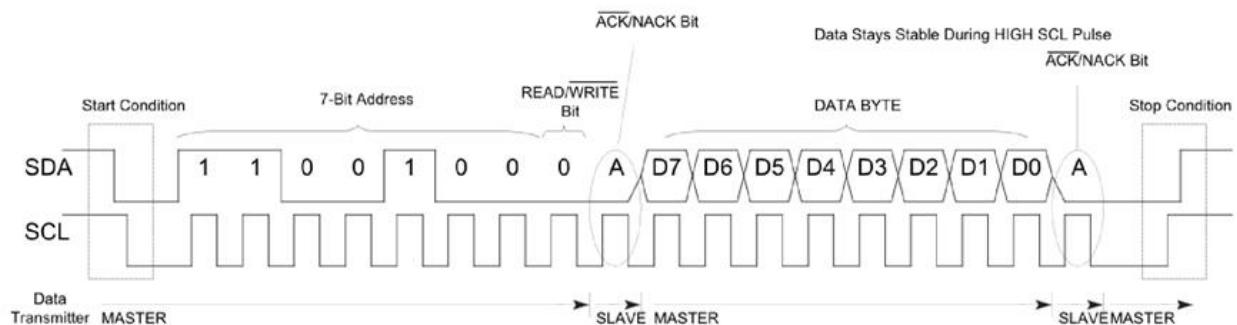


Рисунок 10.15. Пример передачи данных по интерфейсу I2C.

Начало передачи по I2C распознается, если в момент, когда на линии SCL присутствует 1, на линии данных происходит перепад из 1 в 0. Тогда в течение следующих 8 периодов тактового сигнала передается 7-битный адрес ведомого устройства и тип операции (1 – чтение, 0 – запись). После этого ведомое устройство передает один бит подтверждения (~ACK, Acknowledge, «подтверждено») и передаются или принимаются 8 бит данных, начиная со старшего. Передача данных также завершается битом подтверждения. Нулевой бит подтверждения всегда выставляется приемником данных. Останов передачи

по I²C происходит, если при высоком уровне на линии SCL линия данных переходит из 0 в 1.

Скорость обмена данными определена стандартом. Первая версия устанавливала скорость обмена 100 кГц, впоследствии были добавлены режимы 400 кГц и реже встречающиеся 1 МГц и 5 МГц.

Адрес устройства определяется производителем микросхемы с интерфейсом I²C и приводится в технической документации. Теоретически возможна ситуация, когда в системе окажется несколько ведомых устройств с одним и тем же адресом. Перепрограммирование адреса в I²C не предусмотрено, поэтому при использовании множества внешних микросхем с таким интерфейсом.

В настоящее время I²C используется для обмена данными с относительно медленными устройствами. Встречаются жидкокристаллические дисплеи, датчики, устройства управления силовой нагрузкой, сенсорные панели и другие подобные устройства с интерфейсом I²C.

В ряде видеокамер используется аналогичный I²C подход для реализации управляющей шины SCCB (Serial Camera Control Bus). Диаграммы сигналов для этой шины аналогичны I²C, однако временные интервалы могут различаться для разных производителей (в частности, недорогие видеокамеры Omnivision). Это препятствует прямому подключению видеокамер к аппаратным контроллерам I²C, но позволяет использовать аналогичный интерфейс с программной реализацией протоколов обмена.

10.6. Интерфейс LCD (жидкокристаллического индикатора)

Интерфейс LCD (жидкокристаллического индикатора) показан на рис. 10.16. Несмотря на множество доступных вариантов подключения непосредственно индикаторов, стандартом де-факто стал вариант интерфейса, реализованный, например, в микросхемах hd44780 или ks066. Большинство символьных индикаторов использует именно эту схему подключения.

Интерфейс состоит из 8-разрядной шины данных, в которой можно оставить подключенными только 4 старших разряда. Линия RS управляет передачей инструкции контроллеру (0) или данных для вывода (1). Линия R/W управляет записью данных (0) или чтением (1). Контроллер можно использовать и в режиме «только запись», подключив R/W к низкому логическому уровню, поскольку читаемая информация о состоянии контроллера и содержимом его памяти не является критически важной для сохранения работоспособности. Сигнал E используется в качестве *стробирующего*. В отличие от тактового сигнала,

высокий уровень на линии E означает, что в любой момент, пока $E = '1'$, данные на шине D достоверны (valid).

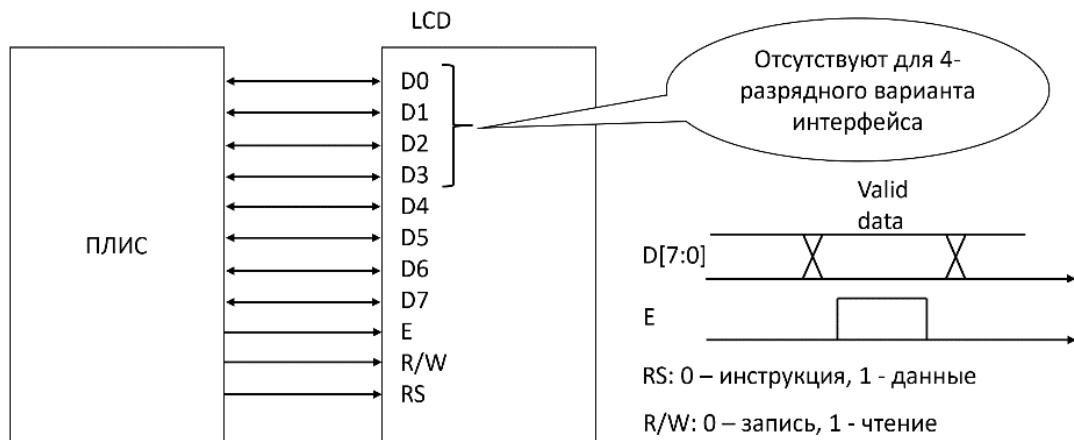


Рисунок 10.16. Интерфейс распространенных контроллеров LCD и временные диаграммы обмена

Для работы LCD обычно должен получить последовательность команд (команды показаны с битовыми полями, включающими конкретные режимы работы):

1. 0x28 (команда Function Set).
2. 0x06 (команда Entry Mode Set).
3. 0x0C (команда Display On/Off).
4. 0x01 (команда Clear Display).

После этого можно выводить данные, установив линию RS в 1. Выводимые символы будут сопровождаться соответствующими изменениями позиции курсора, если это было установлено в полях команды Entry Mode Set.

Особенностью контроллеров LCD являются достаточно длительные задержки, предусмотренные для сигнала E и между посылкой отдельных команд контроллеру. Ввиду наличия множества совместимых с данным интерфейсом контроллеров, точные характеристики следует уточнять в технической документации на конкретную модель. Однако, например, длительность строба сигнала E может составлять порядка 250 нс, а после команды инициализации контроллер может требовать паузы около 15 мс (миллисекунд!). Паузы требуются практически после всех инструкций.

10.7. Интерфейс VGA

Аббревиатура VGA означает Video Gate Array, где под gate array понимается буквально «вентильный массив», т.е., по сути, «микросхема». Таким образом, термин VGA относится фактически к понятию «видеоконтроллер». Этот интерфейс воспроизводит управляющие сигналы дисплеев предыдущего поколения на базе электронно-лучевых трубок, однако вследствие массового распространения как дисплеев, так и видеоконтроллеров этот интерфейс был сохранен и при переходе к жидкокристаллическим дисплеям. При последующем переходе к DVI и HDMI принцип формирования изображения был сохранен, однако эти интерфейсы используют высокоскоростные последовательные сигналы.

Для формирования видеоизображения используются три отдельных сигнала яркости цветовых каналов R (Red), G (Green), B (Blue). Эти сигналы являются аналоговыми, поэтому представляются в виде одиночных выводов. Внутри видеоконтроллера необходимо выполнить цифро-аналоговое преобразование, чтобы получить сигналы, подаваемые на разъем VGA.

Изображение на экране формируется построчно. Начало нового кадра (т.е. переход к левому верхнему углу) отмечается *вертикальным синхросигналом* (VS, Vertical Synchronization). После перемещения до конца строки переход к новой строке отмечается *горизонтальным синхросигналом* (HS, Horizontal Synchronization). Количество строк в кадре и частота смены кадров определяются, таким образом, частотой сигналов горизонтальной и вертикальной синхронизации. Форма и временные интервалы синхросигналом показаны на рис. 10.17. Значения временных интервалов для некоторых видеорежимов приведены в таблице 10.1.

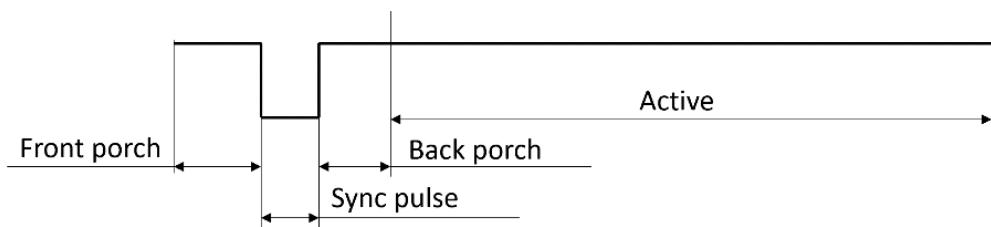


Рисунок 10.17. Параметры сигналов синхронизации интерфейса VGA

На рис. 10.17 показан сам синхросигнал (Sync pulse) и «обрамляющие» его интервалы, которые в электронно-лучевых трубках соответствовали проходу электронного пучка за границами экрана. В течение интервала Active передается сигнал, соответствующий пикселам на очередной строке экрана.

На рис. 10.18 показан разъем VGA и пример подключения этих сигналов к цифровой микросхеме. В данном случае использован пример отладочной платы Basys2. Несмотря на то, что цифровые сигналы имеют постоянный уровень логической единицы, можно в определенных пределах регулировать аналоговое напряжение на выходе, используя показанную на рис. 10.17 схему *токового сумматора*. Он представляет собой простой набор резисторов, каждый последующих из которых имеет сопротивление в 2 раза больше, чем предыдущий. Это означает, что через такой резистор будет протекать ток, в 2 раза меньший, а значит, и его вклад в выходное напряжение будет в 2 раза меньше. Три резистора могут сформировать 8 различных уровней выходного аналогового напряжения, т.е. представляют собой простой 3-разрядный цифро-аналоговый преобразователь.

Можно отметить, что интегральные цифро-аналоговые преобразователи часто строятся именно по этому принципу. Однако получение больших разрядностей (например, 8 разрядов для цвета) сопряжено с тем, что неточные соотношения сопротивлений сделают такую схему нелинейной, а слишком большие сопротивления (в 64, 128 или 256 раз больше начального значения) будут слишком ограничивать ток, и соответствующая линия будет подвержена помехам. Поэтому для 6, 8 или 10 разрядов цвета следует использовать специальные цифро-анalogовые преобразователи, которые выпускаются многими производителями именно для формирования сигналов VGA.

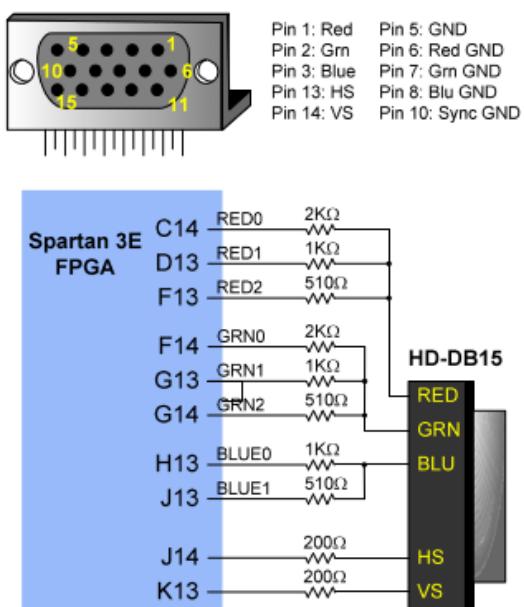


Figure 13. VGA pin definitions and Basys2 circuit

Рисунок 10.18. Пример подключения сигналов VGA для реализации 8-битного цвета

Анализ таблицы 10.1 позволяет выявить требования к характеристикам видеоконтроллеров. Наиболее важным для разработки параметром является «частота следования данных», или иначе «pixel clock». Именно с этой частотой видеоконтроллер должен изменять цифровой код для очередного пикселя на экране. Например, для широко распространенного сегодня формата FullHD с разрешением 1920x1080 пикселов частота следования пикселов составляет почти 150 МГц. Для более современного формата 4К каждая из сторон экрана будет иметь в 2 раза больше пикселов, т.е. на экране из будет в 4 раза больше, и pixel clock становится равен уже 600 МГц. Для формата 8К эта величина увеличится уже до 2400 МГц.

Таблица 10.1. Характеристики сигналов VGA для некоторых видеорежимов

	1368 x 768 @ 60 Hz	1280x720 @ 60 Hz	1920x1080 @ 60 Hz	1920x1140 @ 75 Hz
Частота следования данных, МГц	85.86	74.25	148.5	297
Активная зона, тактов	1368	1280	1920	1920
Front porch, тактов	72	72	88	144
Hsync, тактов	144	80	44	224
Back porch, тактов	216	216	148	352
Активная зона, строк	768	720	1080	1440
Front porch, строк	1	3	4	1
Vsync, строк	3	5	5	3
Back porch, строк	23	22	36	56

При увеличении частоты следования пикселов аналоговые сигналы получают искажения при передаче по длинному кабелю. Например, попытка передавать чередование черных и цветных пикселов приведет к тому, что переходы от нуля к максимальному напряжению будут не прямоугольными, а сглаженными, и изображение приобретает характерный «замыленный» вид. Поэтому для высокого разрешения необходимо использовать цифровые интерфейсы, такие как DVI и быстро вытеснивший его HDMI.

На рис. 10.19 показана структурная схема микросхемы ADV7511, которая представляет собой преобразователь из цифрового видеосигнала в физические сигналы HDMI (4 дифференциальные пары, показанные справа). Входной видеосигнал имеет 36 разрядов видеоданных (сигнал D[35:0]), по 12 разрядов на каждый цветовой канал, два сигнала синхронизации (вертикальной и горизонтальной), общий тактовый сигнал CLK, а также дополнительный сигнал DE (Data Enable), который должен быть равен 1, когда данные находятся в «активной области», т.е. оба синхросигнала находятся в своей зоне Active.

Можно видеть, что настройка параметров этой микросхемы производится с помощью интерфейса I²C.

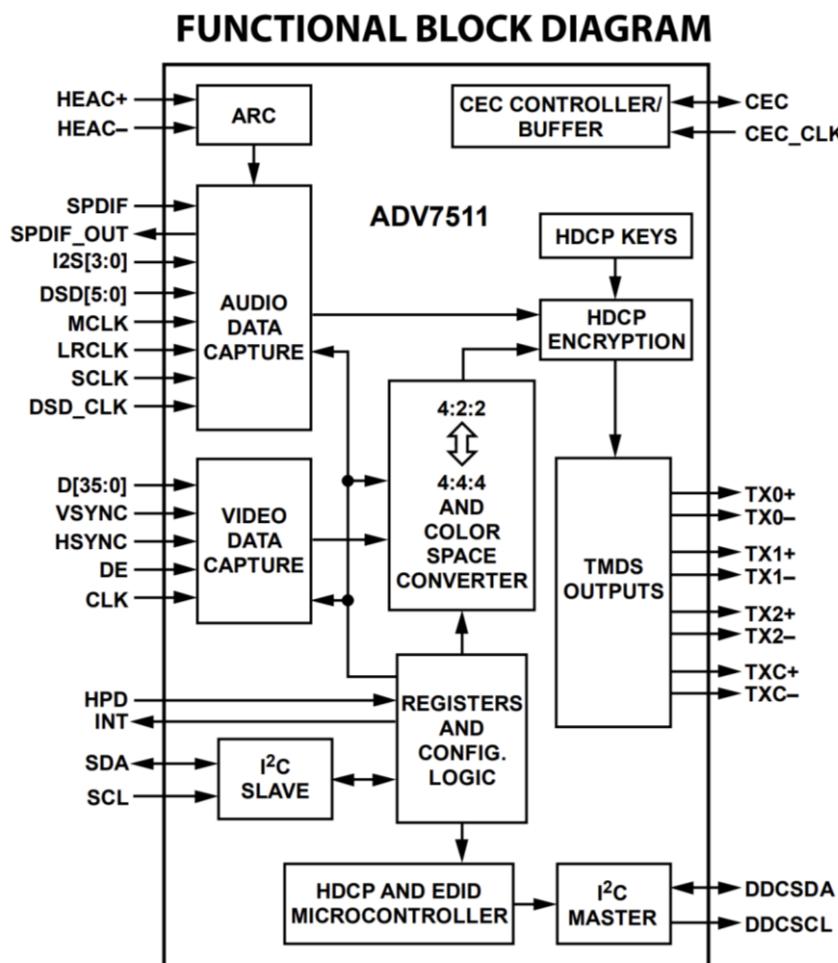


Рисунок 10.19. Структурная схема микросхемы формирования сигналов HDMI

Понимание структуры видеосигнала полезно для подключения видеокамер. Например, распространенный интерфейс CPI (Camera Parallel Interface) копирует сигналы цифровой части VGA. Его подключение к ПЛИС показано на рис. 10.20.

На этом рисунке видно, что камера формирует сигналы PCLK (Pixel Clock), данные о цвете очередного пикселя (Data [9:0]) и сигналы синхронизации HREF, VSYNC. Камеру можно перевести в режим пониженного энергопотребления (при этом передача изображения прекращается) сигналом PWDN (Power Down). Конфигурирование камеры производится с помощью сигналов SCL, SDA, но этот интерфейс не полностью совпадает с I2C по временным диаграммам. Для настройки параметров камеры (индивидуально для каждой модели) следует обращаться к документации производителя, а интерфейс носит название SCCB (Serial Camera Control Bus).

Видеокамеры обычно требует ввода «опорного» тактового сигнала XCLK, на основе которого формируется и выходной сигнал PCLK.

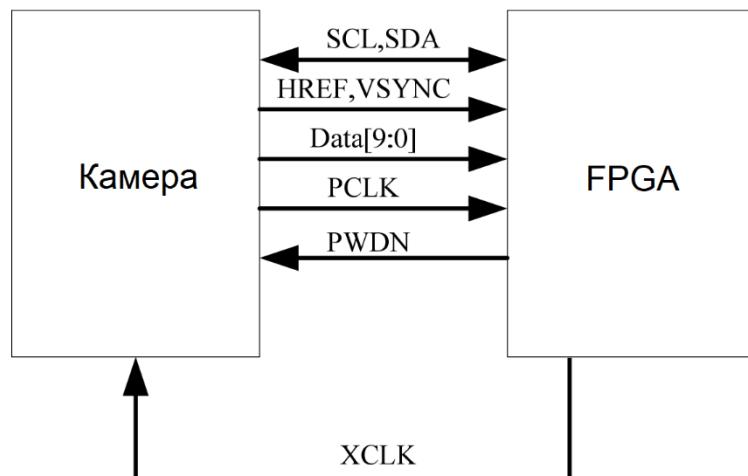


Рисунок 10.20. Пример подключения видеокамеры с интерфейсом Camera Parallel Interface

10.8. Однопроводная передача данных

Если в интерфейсе есть только один сигнал, необходимо отделять один бит от другого. Это несложно сделать, если есть дополнительный сигнал синхронизации, но тогда придется вводить дополнительные линии. Например, в UART данные в каждом байте никак не отделяются – источник и приемник настраиваются на одну и ту же скорость передачи, и приемник захватывает значения отдельных битов в определенные моменты времени. Рано или поздно из-за рассогласования тактовых генераторов эти моменты существенно

сдвинутся, однако в UART предусмотрен специальный бит «старт», по которому и производится синхронизация источника и приемника.

Подход, примененный в UART, не является единственным. Например, протоколы 1-wire и Manchester (а также его вариант Manchester-II) используют варианты *самосинхронизации* на основе перепадов логического уровня.

Еще один способ кодирования используется в получивших популярность светодиодных лентах. Для управления большим количеством многоцветных светодиодов необходимо передавать данные о яркости каждого цветового канала каждого светодиода. Это могло бы потребовать слишком много сигналов, если бы каждый светодиод подключался независимо. Однако в широко распространенных светодиодных лентах есть только одна линия данных, подключенная к первому светодиоду. Выход этого светодиода подключен ко второму светодиоду и т.д. После получения данных о яркости каждый светодиод начинает пропускать приходящие к нему данные на следующий светодиод. Способ передачи одного бита основан на измерении длительности импульса, как показано на рис. 10.21.

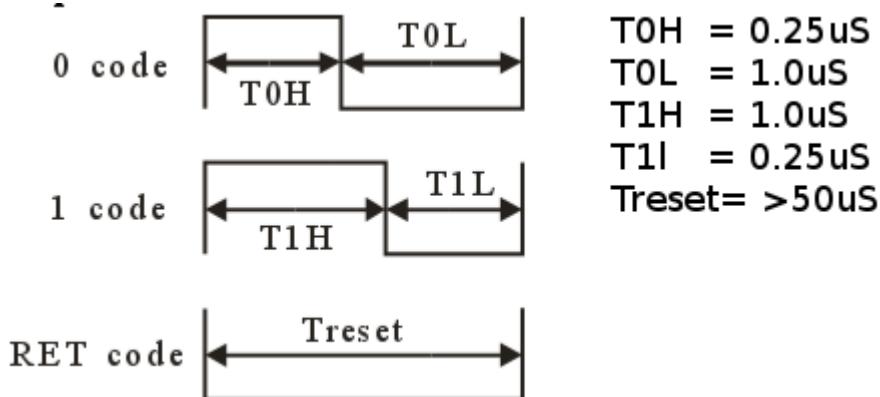


Рисунок 10.21. Временные диаграммы управления адресуемой светодиодной лентой по однопроводному последовательному интерфейсу

В нормальном состоянии линия находится в состоянии логической 1. Если на вход первого светодиода подается логический 0 на время не менее 50 мкс, все светодиоды в цепочке сбрасываются, и готовы принимать данные о яркости (однако первый же пакет данных будет принят первым светодиодом в ленте). Протокол передачи данных светодиода WS2812b показан на рис. 10.23.

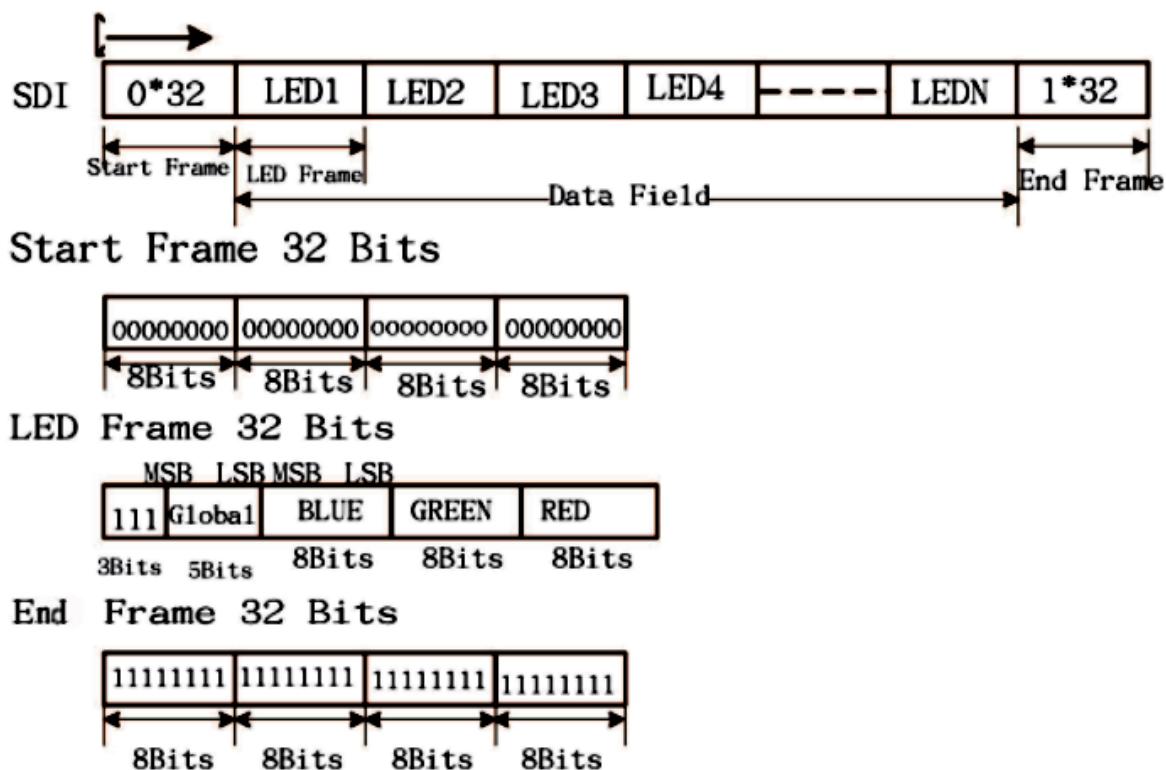


Рисунок 10.22. Протокол управления адресуемой светодиодной лентой по однопроводному интерфейсу

После сброса каждый светодиод требует 32 бита. Первый байт такого пакета содержит 111 в старших битах, за которыми идут 5 бит, регулирующих «глобальную яркость», т.е. применяемых как множитель ко всем трем цветовым каналам. За первым байтом идут три байта, задающих яркость синего, зеленого и красного каналов соответственно.

Внешний вид светодиодной ленты показан на рис. 10.23.



Рисунок 10.23. Внешний вид адресуемой светодиодной ленты

Светодиодные ленты получили широкое распространение, в том числе как декоративный элемент компьютерных корпусов, вентиляторов, модулей памяти и других устройств. Это относительно простой способ получить эффектное цифровое устройство.

10.8. Таймеры

Таймер отличается от обычного счетчика наличием дополнительных возможностей по запуску, перезагрузке и останову. В примере показан простейший таймер, который обеспечивает задержку появления выходного сигнала на 200 тактов относительно появления входного сигнала `reload`. После достижения максимального значения таймер не повторяет цикл счета с нуля, а останавливается до появления сигнала `reload`. В таком режиме данное устройство может использоваться как сторожевой таймер (*watchdog timer*). Эта разновидность таймера предназначена для формирования предупреждений о том, что какое-то событие, вызывающее появление сигнала перезагрузки таймера, не происходило уже длительное время. Сторожевой таймер часто используется в микроконтроллерных системах управления, где его перезагрузка происходит в процессе выполнения программы. Отсутствие перезагрузки в течение времени, соответствующего полному циклу счета, свидетельствует о том, что микроконтроллер перестал периодически выполнять команды, приводящие к сбросу счетчика. Это, вероятнее всего, является следствием аппаратного или программного сбоя системы, и выход сторожевого таймера может использоваться как сигнал аппаратного сброса микроконтроллера или индикатор аварийного состояния системы. Пример таймера, сбрасываемого по сигналу `reload`, и прекращающего счет при достижении заданного состояния, приведен ниже:

```
module timer(
    input clk,
    input reload,
    output timer_out
);

reg [7:0] cnt;

always @ (posedge clk)
    if (reload) cnt <= 0;
    else if (cnt < 199) cnt <= cnt + 1;
```

```
assign timer_out = (cnt == 199) ? 1 : 0;  
endmodule
```

У сторожевого таймера есть полезное применение, эффективно решающее достаточно важную проблему подавления «дребезга» механических кнопок. Это известная проблема в цифровой технике, заключающаяся в том, что механические кнопки, которые субъективно воспринимаются человеком как источники коротких (в масштабах человеческого восприятия) импульсов, на практике при нажатии и отпускании формируют пачку импульсов. Причиной такого поведения является неизбежное окисление контактов, приводящее к нестабильному соединению в первый момент касания. С точки зрения человека, процесс нажатия может занимать доли секунды, однако в масштабах цифровой системы может пройти от сотен до тысяч тактов системного генератора, прежде чем последовательность 0 и 1 перейдет в стабильное состояние. Поэтому использовать кнопку в качестве источника единичного импульса на практике невозможно.

Некоторые схемы подавления дребезга обладают принципиальным недостатком – они ориентируются на вполне конкретные характеристики процесса переключения. Установка внешних RC-фильтров решает проблему только в частных случаях, поскольку даже при подборе номиналов для компонентов фильтра дальнейшие процессы окисления или просто особенности динамики нажатия кнопки приведут к тому, что процесс «дребезга» затянется, и примененный фильтр перестанет надежно подавлять импульсы. То же относится и к цифровым схемам подавления «дребезга», построенным по принципу выявления коротких импульсов. Проблема заключается в том, что через некоторое время эксплуатации длительность таких импульсов вполне может измениться.

Генератор импульсов, основанный на сторожевом таймере, является хорошим вариантом, способным формировать при нажатии кнопки один длинный импульс. При этом вход reload устанавливает выход в 1 и загружает счетчик начальным значением. Когда на входе reload присутствует логический 0, счетчик сторожевого таймера уменьшается вплоть до 0. При таком подходе короткие импульсы нуля на входе не будут успевать обнулять счетчик во время процессов «дребезга», а удержание кнопки будет приводить к перезагрузке максимального значения счетчика на каждом такте. Длительность счета должна быть выбрана заранее большей длительности любого процесса «дребезга», с

запасом на ухудшение состояния контактов. На практике человек воспринимает в качестве однократных нажатия длительностью в десятые доли секунды.

10.9. Параметризация модулей

Параметры являются эффективным средством разработки масштабируемых модулей, т.е. модулей, для которых возможна быстрая смена разрядности, числа каналов, пределов счета и тому подобных численных характеристик, при сохранении алгоритмов работы. Параметр определяется с помощью директивы `define

```
`define <имя_параметра> <значение>
```

Например:

```
`define DATA_WIDTH 8
```

Впоследствии при упоминании в тексте DATA_WIDTH вместо него будет подставлено значение 8.

Параметризованные модули являются достаточно привлекательными для разработки по многим причинам. Рассмотрим, например, реализацию регистров различной разрядности.

8-разрядный регистр:

```
module reg8( input clk,
              input [7:0] d,
              output reg [7:0] q);

    always @ (posedge clk)
        q <= d;
endmodule
```

16-разрядный регистр:

```
module reg16( input clk,
               input [15:0] d,
               output reg [15:0] q);

    always @ (posedge clk)
        q <= d;
```

```
endmodule
```

Из описания модулей можно видеть, что кроме имени, они различаются только разрядностью сигналов d и q. Нетрудно понять, что регистр любой разрядности будет описываться процедурным блоком, в котором будет находиться оператор `q <= d;`. Таким образом, для перехода к другой разрядности необходимо отредактировать только разрядность портов d и q.

Используем для управления разрядностью директиву `define.

```
`define DATA_WIDTH 8
module reg8( input clk,
              input [`DATA_WIDTH - 1:0] d,
              output reg [`DATA_WIDTH - 1:0] q);

    always @ (posedge clk)
        q <= d;
endmodule
```

При объявлении портов теперь используется не прямое указание числа, а ссылка на определенный через `define параметр, названный DATA_WIDTH (буквально «ширина данных», или разрядность данных). Поскольку параметр в примере задан равным 8, выражение `[`DATA_WIDTH - 1:0]` эквивалентно `[7:0]`, что и требуется для 8-разрядного порта. Можно заметить, что редактирование единственной строки, где определен параметр, автоматически изменяет разрядности обоих сигналов – d и q, тогда как при ручном редактировании за приведением разрядности этих сигналов в соответствие друг другу необходимо следить отдельно. Представление требуемых величин в виде параметров особенно эффективно, если они используются в различных модулях – например, при разработке системы цифровой обработки сигналов, в которой каждый модуль производит очередное преобразование сигнала. При необходимости изменить разрядность обрабатываемых данных пришлось бы производить коррекции во всех модулях.

Verilog поддерживает также локальные параметры. Они имеют ограниченную область видимости (только в том блоке, в котором объявлены) и предназначены для того, чтобы не перекрывать имена ранее заданных параметров. Например, идентификатор width («ширина») является достаточно популярным при определении разрядности обрабатываемых данных. При его

частом использовании в различных модулях может оказаться, что при их совместной трансляции в большом проекте значение параметра `width` окажется переопределенным одним из загруженных файлов. Для ограничения области видимости и используются локальные параметры, определяемые с помощью ключевого слова `localparam`. Пример:

```
localparam x = 1;
```

Широкое использование параметров не является самоцелью, однако следует обратить внимание на те возможности по организации разработки, которые предоставляет параметризация. При создании модулей, которые используются во многих проектах, или для которых часто применяется частичная коррекция (например, изменение разрядности, пределов счета, начальных значений, порогов срабатывания и т.п.), настоятельно рекомендуется оформить часто изменяемые значения как параметры, вынеся их в отдельный текстовый блок (в пределах модуля или, при необходимости, во внешний загружаемый файл). В этом случае уменьшатся шансы случайно испортить исходные тексты модуля при внесении коррекции в величины, упоминающиеся по всему исходному тексту, поскольку исправления можно будет ограничить компактным текстовым фрагментом.

10.10. Выводы по разделу

Большое количество датчиков, исполнительных устройств, дисплеев и прочих внешних модулей имеют один из видов несложных цифровых интерфейсов, которые были рассмотрены в данной главе. Например, на рис. 10.24 показана веб-страница с модулями различных устройств (датчиков, внешних преобразователей интерфейсов, систем связи и т.п.), которые могут быть подключены к управляющему устройству с помощью обычных цифровых сигналов (непосредственно управляющих работой), либо сигналов, реализующих SPI, I2C, ШИМ и т.д.

Умение подключить к системе готовый внешний модуль позволяет быстро проверить принципиальную работоспособность системы и является очень полезным практическим навыком. При этом знание принципов работы простых цифровых интерфейсов является ключевым фактором такой деятельности.

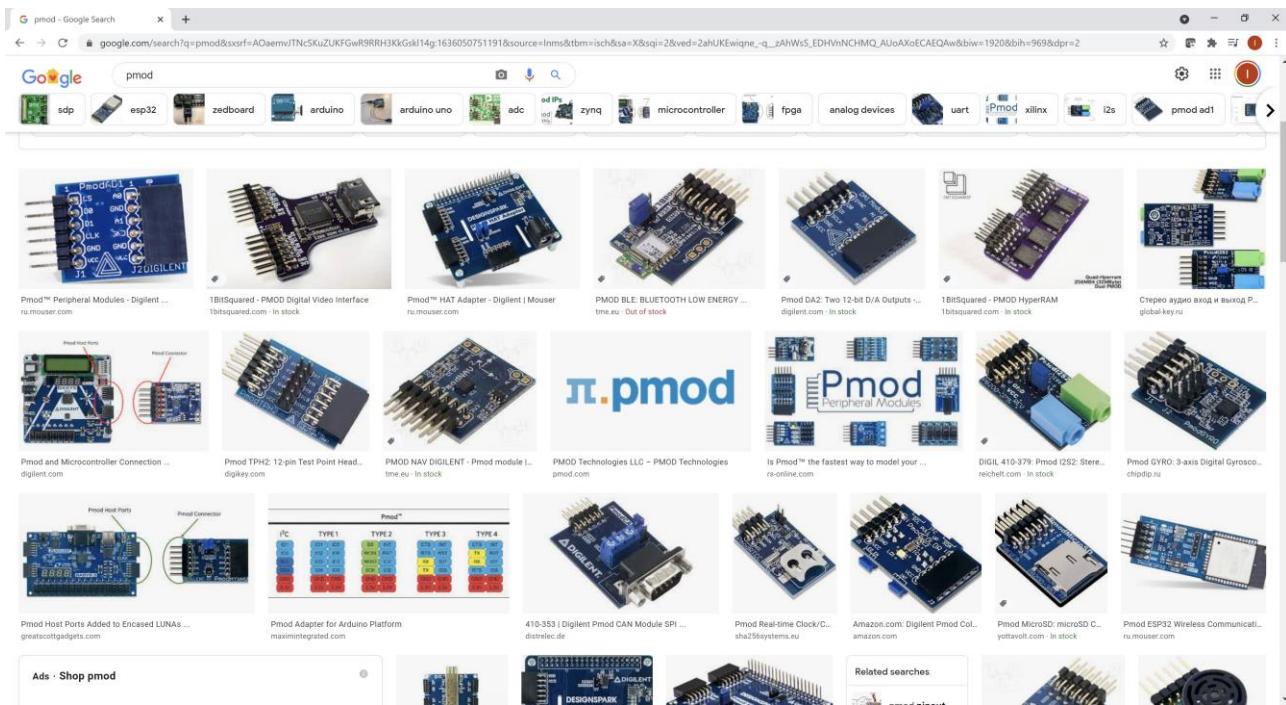


Рисунок 10.24. Примеры внешних модулей с простыми цифровыми интерфейсами

Контрольные вопросы:

1. Какие сигналы используются в интерфейсе SPI?
2. Если используется 3 ведомых устройства с интерфейсом SPI, какое минимальное количество цифровых линий требуется для их подключения?
3. Зачем требуется адрес в протоколе обмена по интерфейсу I2C?
4. Каким образом устанавливается скорость обмена данными в интерфейсе UART?
5. Можно ли считать UART устаревшим интерфейсом?
6. Какие сигналы требуются для вывода изображения с помощью интерфейса VGA?
7. Почему управление с помощью ШИМ часто используется для управления силовой нагрузкой?

11. ПРОЦЕССОРНОЕ ЯДРО

11.1. Основные сведения о процессорных ядрах

Процессоры широко применяются в самых разных устройствах и являются основой вычислительных систем. Удешевление процессоров по мере развития микроэлектронной технологии привело к тому, что даже в простых устройствах используются недорогие процессоры. Вместо изменения схемы (т.е. повторного изготовления печатной платы и монтажа компонентов) для коррекции работы устройства часто достаточно изменить программу управляющего им процессора. Это существенно быстрее и открывает перед разработчиками больше возможностей, поэтому недорогой процессор часто используется для управления самыми разными приборами, образуя огромный класс *встраиваемых* (embedded) устройств.

Процессор не обязан быть единственным в системе. Например, современный автомобиль содержит десятки процессоров различных типов. Кроме устройств, которые очевидно распознаются как элементы бортового компьютера (например, приборная панель автомобиля), простые процессоры используются для управления стеклоподъемниками, освещением, зажиганием и другими функциями автомобиля. Все эти процессоры имеют разные требования по производительности, объему памяти и подключаемым устройствам.

Поэтому в настоящее время вполне актуальна разработка процессоров для решения отдельных задач. Не следует полагать, что такие системы, как x86, ARM, RISC-V, AVR в определенном сочетании достаточны и оптимальны для решения любой задачи. Разработка новых процессорных архитектур продолжается в мировой микроэлектронике до сих пор, поэтому получение такого навыка может оказаться полезным.

Понятие «процессорное ядро» до сих пор не получило однозначного определения. Тем не менее, при проектировании вычислительных устройств можно разделить процессор на внутренние компоненты и подключенные к ним периферийные устройства. На практике оказывается, что имея некий минимальный набор узлов («ядро»), можно несложным образом подключать к нему дополнительные модули, например, UART, SPI или ШИМ. Такие модули ничего не изменяют в конструкции ядра. Поэтому под ядром процессора понимается набор, который обычно включает в себя доступные программисту регистры, арифметико-логическое устройство и систему управления выполнением команд. Эти компоненты должны правильно взаимодействовать друг с другом, поэтому они требуют согласованного проектирования.

11.2. Преобразование конечного автомата в процессор

Для понимания принципов работы процессора можно вернуться к понятию конечного автомата, который показан на рис. 11.1.

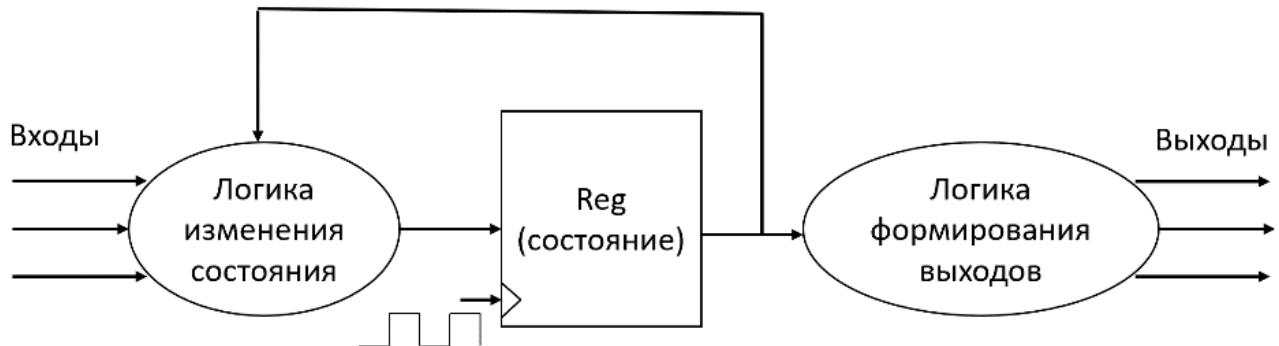


Рисунок 11.1. Конечный автомат

Корректируя схему «логика изменения состояния», можно добиться изменения поведения конечного автомата. Это неудобно, потому что для реализации сложного алгоритма потребуется постоянно модифицировать эту схему и повторять моделирование или загрузку конфигурации ПЛИС. В то же время видно, что формирование выходных сигналов осуществляется довольно регулярным способом – на основе анализа состояния автомата. Сделать КА более гибкой можно, если добавить таблицу с правилами перехода между состояниями, как показано на рис. 11.2.

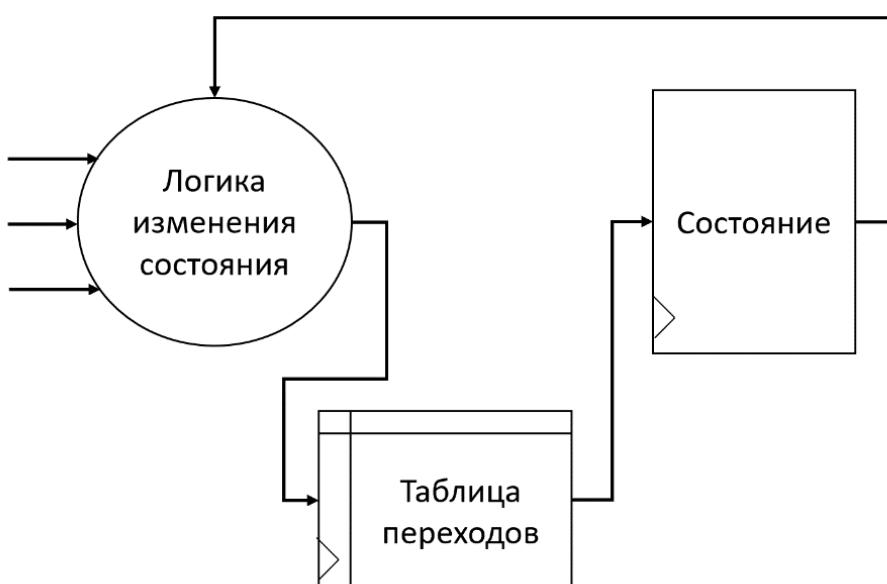


Рисунок 11.2. Преобразование конечного автомата

Показанная на рис. 11.2 схема очень близка к процессору. Действительно, представим, что таблица переходов представляет собой память программы. Входной сигнал является адресом этой памяти, а содержимое (данные) – номером состояния, в который должен перейти КА. Тогда можно будет изменять таблицу переходов, записывая в память новые значения, и автомат начнет переходить в совершенно новые последовательности состояний.

При рассмотрении процессора важным понятием является абстрактная модель – машина Тьюринга. Она представляет собой бесконечную ленту с «символами», под которыми понимается некое абстрактное понятие данных, которое может принимать любую требуемую форму (например, байт, 32 или 64 бита и т.д.). Имеется указатель, который может:

1. Прочитать символ.
2. Заменить его (или не заменять).
3. Перейти к другой позиции на ленте.

В целом считается, что любой алгоритм может быть выполнен с помощью машины Тьюринга. Это утверждение будет убедительнее, если учесть, что машина может выполнять с символом любое требуемое действие, переходить на другой символ по любому требуемому алгоритмом правилу, а сама лента имеет сколько угодно ячеек.



Рисунок 11.3. Абстрактная модель – машина Тьюринга

Поскольку машина Тьюринга предполагает бесконечную ленту, она не может быть реализована на практике. Тем не менее, если у вычислительного устройства есть достаточно памяти для выполнения конкретного алгоритма, то в данном частном случае он может рассматриваться как подобие машины Тьюринга. Можно представить, что лента с символами – это память с данными, действия над символами задаются набором команд арифметико-логического устройства, а перемещение указателя – командами вычисления адреса и командами перехода к новой операции.

Для процессора рассматриваются два важных понятия, которые и определяют его возможности:

1. Архитектура системы команд (ACK) – какие команды есть у процессора.
2. Микроархитектура – как эти команды выполняются на аппаратном уровне.

В целом микроархитектура и ACK не обязаны строго соответствовать друг другу. Например, один из первых вариантов процессора Intel 8051 использовал команды, выполняющиеся за 12 или 24 такта (1 или 2 «машинных цикла», каждый из которых состоял из 12 тактов). Процессор оказался настолько популярным, что многие разработчики продолжали его использовать, даже когда оригинальный вариант перестал выпускаться. Ряд компаний выпустил свои аналоги 8051, причем какие-то полностью сохраняли поведение, а другие сохраняли только систему команд, которые стали выполнять уже за 1 или 2 такта. Это пример того, как при одной и той же системе команд изменилась микроархитектура.

С другой стороны, некоторые команды требуют определенной микроархитектуры. Например, операции с данными в памяти требуют и чтения, и записи памяти, а какая-то микроархитектура может допускать только одно действие с памятью для каждой команды.

Кроме того, микроархитектуры могут по-разному сочетаться с разными наборами команд. Взаимное согласование микроархитектуры и системы команд является важным процессом в проектировании ядра и представляет собой в чем-то элемент искусства, поскольку трудно сформулировать четкие критерии оптимальности итоговой конструкции.

11.3. Двухтактная архитектура

Двухтактная архитектура процессора является наиболее простой для понимания. Как следует из названия, она реализует работу процессора за два такта. Первый – это чтение команды из памяти. Этот процесс иллюстрирован рис. 11.4.

Работа памяти происходит по обычным правилам. По фронту тактового сигнала на выходе памяти появляется содержимое ячейки с номером addr. В случае процессора адрес – это номер команды. Традиционно этот адрес хранится в регистре PC (Program Counter, счетчик команд). Имя регистра общеупотребительно, а самым известным исключением является аналогичный регистр IP (Instruction Pointer, указатель инструкций), использованный в процессоре Intel 8086. Корпорация Intel поясняет это тем, что в процессоре 8086 был реализован внутренний буфер для накопления команд, поэтому регистр IP содержал адрес команды, которая выполнялась в данный момент, причем в это

время процессор мог выполнять чтение другой команды, и адрес, подаваемый им на внешнюю память, мог быть другим.

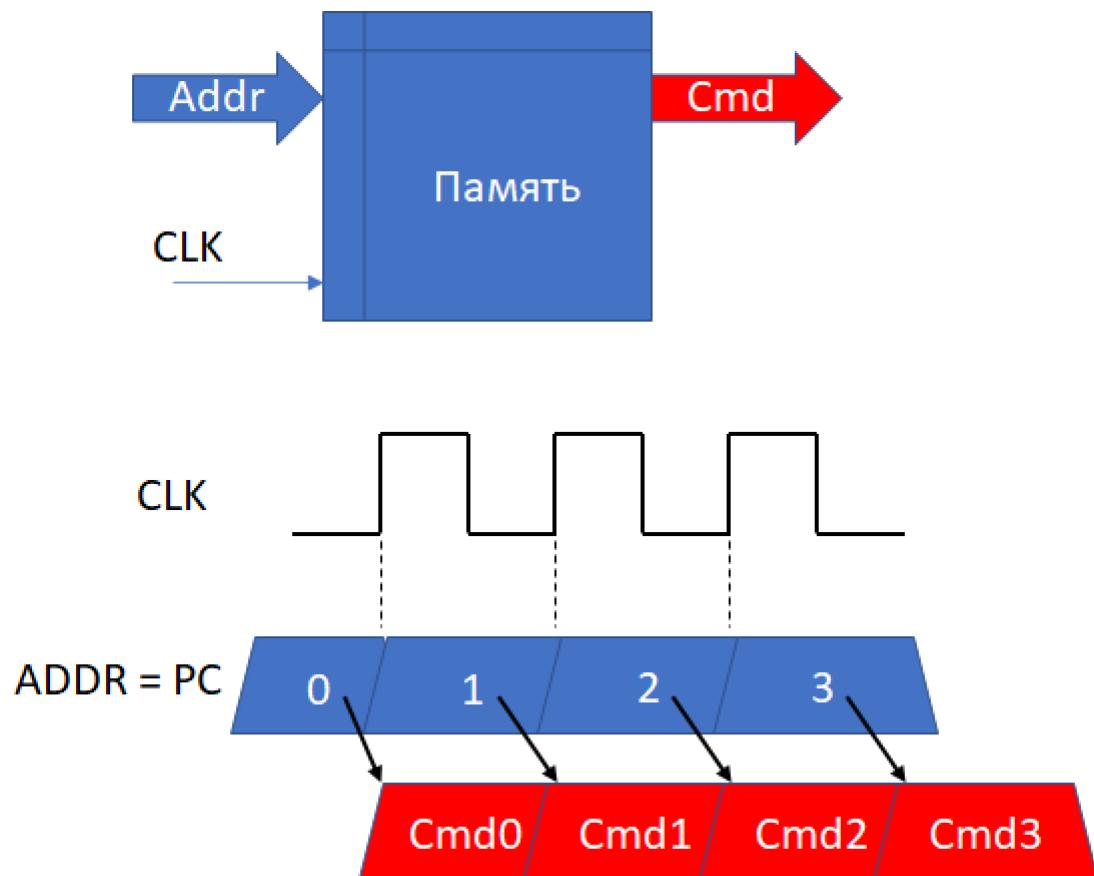


Рисунок 11.4. Чтение команд из памяти

Когда команда уже известна, следующим тактом можно выполнить ее. С точки зрения языков описания аппаратуры, можно реализовать требуемые действия в виде оператора `case`, где выбор производится на основе кода прочитанной команды. Таким образом, работа простого ядра состоит из повторения двух действий:

1. Чтение команды, или ее «выборка» (fetch).
2. Исполнение команды (Execute).

Последовательность таких циклов показана на рис. 11.5.

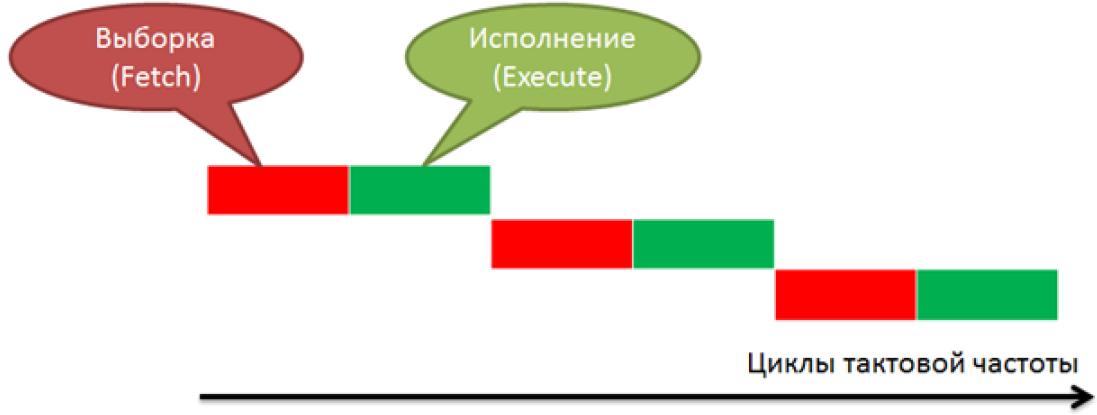


Рисунок 11.5. Двухтактная работа процессора

Под выполнением команды понимается запись новых значений во все регистры, которые этого требуют. Особенно важно обновить регистр РС, иначе процессор остановится на определенной ячейке памяти. В дальнейшем будет показано, что обновление РС в некоторых случаях можно производить и не дожидаясь получения команды.

На рис. 11.6 показана обобщенная структурная схема простого процессорного ядра с двухтактным циклом работы, который уже имеет все основные элементы. На рис. 11.7 эти элементы детализированы. Например, «регистры» представлены в виде двух регистров – RegA и RegB. Практически используемые процессоры имеют существенно больше регистров, однако простой пример позволяет нагляднее продемонстрировать основные правила проектирования.

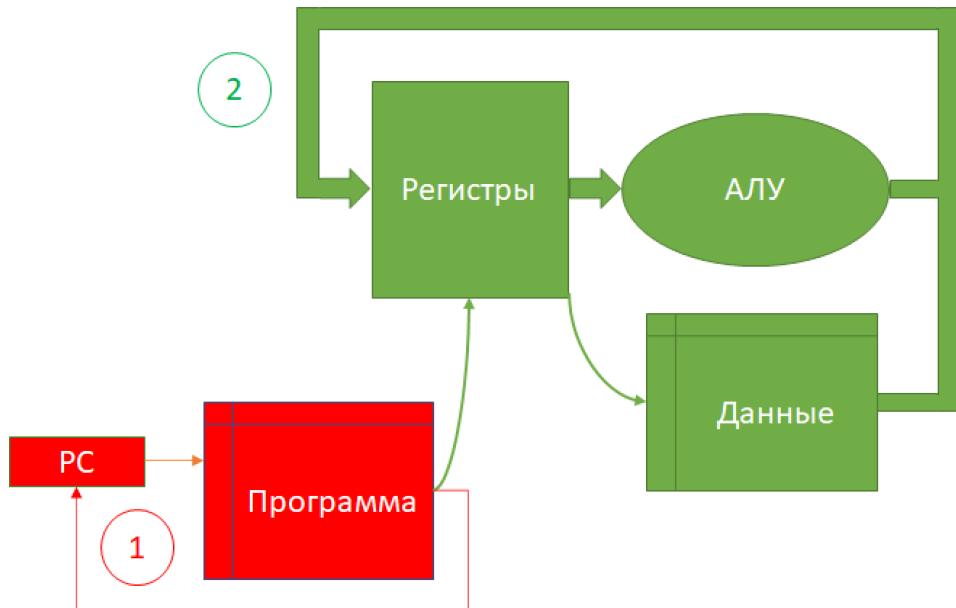


Рисунок 11.6. Структурная схема процессора с двухтактным циклом работы

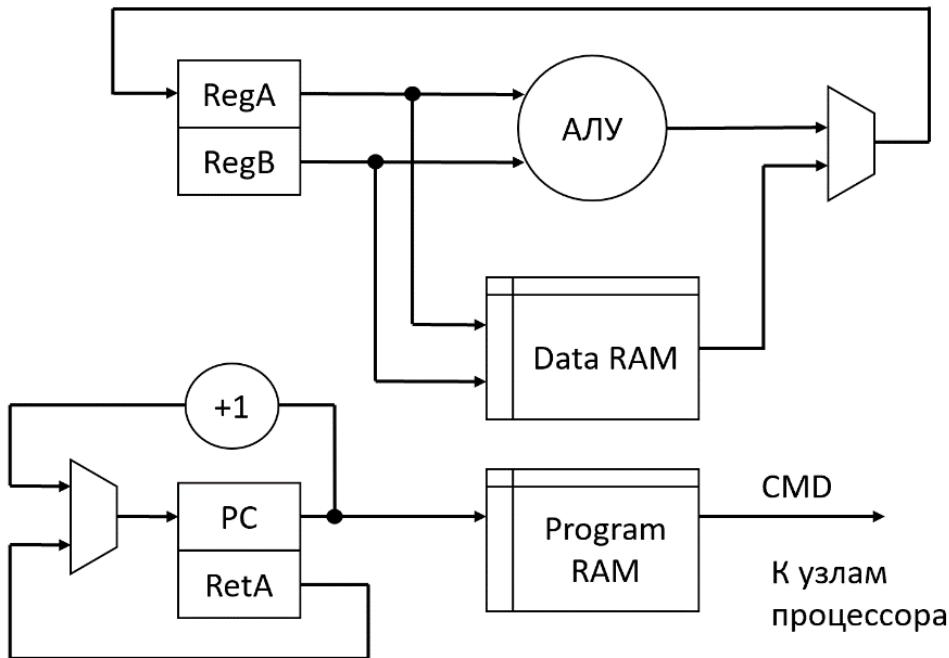


Рисунок 11.7. Пример простого процессорного ядра

В схеме, показанной на рис. 11.7, есть важная особенность. Если процессор выполняет команду перехода, без ее чтения непонятно, из какого адреса необходимо читать следующую команду. Пока для простоты можно принять, что каждая команда процессора будет выполняться за два такта. На первом такте процессор читает команду из ячейки памяти программ по адресу РС. На втором такте, имея в своем распоряжении код команды, на его основе можно определить, какой результат (и в какой регистр) записывать, а также адрес следующей команды.

Часто процессор выполняет линейную последовательность команд, поэтому можно заранее предполагать, что будет выполнено действие $PC = PC + 1$. Это позволяет проектировать конвейеризованные процессорные ядра, которые начинают выполнение следующей команды до завершения предыдущей.

Процессор, показанный в качестве примера, разрабатывается следующим образом. Интерфейс процессора объявляется в модуле:

```
module cpu_top(
    input clk_in, // тактовый сигнал
    input reset, // сброс
    input rx, // UART
    output tx, // UART
    input [3:0] sw, // входы - переключатели
```

```

output [3:0] led, // выходы - светодиоды
// интерфейс загрузки программы
input cmdwe,
input [15:0] cmdaddr,
input [35:0] cmddata
);

```

В процессорном ядре добавлен интерфейс UART, который с большой вероятностью понадобится впоследствии, хотя и не задействован в приведенном примере. Кроме того, если описать неподключенное ядро процессора, синтезатор исключит его из проекта. Поэтому требуется добавить выходы какого-либо простого периферийного устройства – в примере это 4-разрядная шина led.

Чтобы синтезатор не заменил память на небольшой набор констант, память программ должна иметь внешний интерфейс, который в принципе способен записать в нее новые значения. Если оперировать в модели процессора небольшими программами, синтезатор скорее всего будет оптимизировать их содержимое. Если же у памяти программ объявлен второй интерфейс, по которому может производиться запись, такая оптимизация уже не будет состоятельной, поскольку содержимое памяти, с точки зрения синтезатора, сможет впоследствии измениться.

В модуле процессора необходимо сделать объявления регистров и параметров, как показано ниже. Директивы define позволяют впоследствии пользоваться мнемоническими обозначениями, быстро изменения размеры памяти при необходимости. Это не строго обязательный, но настоятельно рекомендуемый подход в программировании – вместо чисел, смысл которых при последующем обращении к этому коду может оказаться непонятным, используются имена, напрямую указывающие на смысл этого параметра.

Показанные на структурной схеме регистры процессора определены с помощью объявлений reg. Память программ и память данных представлены отдельными двумерными массивами, что позволяет синтезатору распознать шаблон модуля памяти.

```

`define PCWIDTH 16
`define PROGRAMSIZE 1024
`define DATASIZE 1024
reg [`PCWIDTH - 1 : 0] pc, newpc, RetReg;
reg [35:0] Program [`PROGRAMSIZE - 1 : 0];

```

```

reg [35:0] cmd;
reg [31:0] Data [^DATASIZE - 1 : 0];
reg [31:0] RegA, RegB, newRegA, newRegB, dmem;
reg [3:0] RegLED;
initial begin
    $readmemh("program.mem", Program);
end

```

Установка модуля тактового генератора происходит с помощью генератора IP-ядер, входящего в состав САПР. Аппаратный компонент тактового генератора настоятельно рекомендуется, если проект будет программироваться в ПЛИС. Для моделирования этот шаг необязателен. В приведенном ниже примере тактовый сигнал clk, который будет использован в качестве основного для процессора, напрямую соединен с входом clk_in. Если же тактовый генератор действительно будет установлен, строку с таким прямым соединением необходимо закомментировать, а в объявлении IP-ядра тактового генератора вместо .clk_out1(null) поместить стоящий за ним закомментированный текст.

```

clk_wiz_0 inst_gen
( .clk_out1(null), // .clk_out1(clk),
  .clk_in1(clk_in)
);
assign clk = clk_in; // это объявление не требуется, если подключен выход
генератора

```

Для демонстрации работы процессора можно выбрать небольшую систему команд. Она также задается мнемоническими именами с помощью директивы define.

```

`define NOP 0
`define ATOB 1
`define BTOA 2
`define ADD 3
`define SUB 4
`define OUTLED 5
`define INSW 6
`define WRITEMEM 7

```

```

`define READMEM 8
`define RET 9
// Специальные случаи
// 1111xxxxxx call xxxxxx
// 1110xxxxxx jmp xxxxxx
// 1101xxxxxx jmp xxxxxx if A = 0
// 0001xxxxxx загрузка в A xxxxxx
// 0010xxxxxx загрузка в B xxxxxx

```

В конце списка показаны специальные случаи, которые содержат в поле команды литералы, т.е. двоичные данные, воспринимаемые *буквально* (literally), а не как код определенного действия. Если старшие 4 бита команды принимают показанные значения, то в остальных 32 содержится число, требуемое этой команде. Использование 36-разрядной команды довольно расточительно, однако это позволяет не усложнять пример процессора.

Далее необходимо описать поведение основных компонентов процессора. Для упрощения понимания и отладки кода можно воспользоваться тем же приемом, что и при разбиении описания конечного автомата на несколько процессов. В одном процессе, выполняющемся синхронно по фронту тактового сигнала, в регистр записывается «новое значение». Чему равно это значение, определяет другой процесс, выполняемый в виде комбинационного выражения. Например, процесс обновления регистра pc описан в простейшем виде – по сигналу сброса записывается 0, иначе newpc.

```

always @(posedge clk)
begin
  if (reset) pc <= 0;
  else pc <= newpc;
end

```

Значение сигнала newpc на каждом такте описано ниже. Случаями, когда новое значение вычисляется не как pc + 1, являются команды перехода, условного перехода, вызова подпрограммы и возврата из подпрограммы. В первых трех случаях новое значение – это разряды 15:0 команды, а при возврате из подпрограммы – значение регистра RetReg.

```

always @ *
begin
  if ((cmd[35:33] == 3'b111) || ((cmd[35:32] == 4'b1101) && (RegA == 0)))
newpc <= cmd[15:0];
  else if (cmd == `RET) newpc <= RetReg;
  else newpc <= pc + 1;
end

```

Регистр адреса возврата принимает значение $pc + 1$, если процессор выполняет команду вызова подпрограммы. В этом случае нужно запомнить в регистре адрес, с которого следует продолжать выполнение программы. В данном простом примере регистр адреса возврата только один, поэтому вызывать вложенные подпрограммы нельзя. В практических используемых процессорах адреса возврата хранятся на стеке, что обеспечивает последовательное помещение адресов на стек и снятие их со стека в том же порядке.

```

always @(posedge clk)
begin
  if (reset) RetReg <= 0;
  else if (cmd[35:32] == 4'b1111) RetReg <= pc + 1;
end

```

Память программ описана в виде массива ячеек разрядностью 36 бит. Чтобы этот массив не оказался заменен на нули или небольшую таблицу, для него описывается запись нового значения выражением `if (cmdwe) Program[cmdaddr] <= cmddata;` В этом случае все ячейки памяти будут подключены к внешним сигналам и оптимизировать такое выражение на этапе синтеза окажется невозможным.

```

always @(posedge clk)
begin
  if (cmdwe) Program[cmdaddr] <= cmddata;
  cmd <= Program[pc];
end

```

Описание регистров данных выполнено по схожей схеме с описанием регистра `pc`. В регистр записывается 0 в момент сброса или «новое значение». Для упрощения описания в синхронном процессоре описана также и загрузка литерала, чтобы подчеркнуть важность этой операции.

```

always @(posedge clk)
begin
    if (reset) RegA <= 0;
    else if (cmd[35:32] == 4'b0001) RegA <= cmd[31:0];
    else RegA <= newRegA;
end

always @(posedge clk)
begin
    if (reset) RegB <= 0;
    else if (cmd[35:32] == 4'b0010) RegB <= cmd[31:0];
    else RegB <= newRegB;
end

```

Описание арифметико-логического устройства представляет собой комбинационное выражение, которое вычисляет результат, помещаемый впоследствии в `RegA` и в `RegB` (для каждого регистра используется свой процесс `always`). Такой подход позволяет наглядно проверить отсутствие конфликта сигналов, если, например, в результате анализа двух процессов окажется, что в `RegA` должны быть записаны два значения одновременно.

```

always @ *
begin
    case (cmd)
        `NOP : newRegA <= RegA;
        `BTOA : newRegA <= RegB;
        `ADD : newRegA <= RegA + RegB;
        `SUB : newRegA <= RegA - RegB;
        `INSW : newRegA <= sw;
        `READMEM : newRegA <= dmem;
        default : newRegA <= RegA;
    endcase

```

```

end

always @ *
begin
  case (cmd)
    `NOP : newRegB <= RegB;
    `ATOB : newRegB <= RegA;
    default : newRegB <= RegB;
  endcase
end

```

Память данных так же, как и память программ, описана в виде двумерного массива. Особенностью памяти является то, что по фронту тактового сигнала нельзя одновременно и прочитать значение какой-то ячейки, и записать ее в регистр назначения. По первому фронту требуемое значение появится на выходе блока памяти, и только следующим тaktом оно может быть записано в требуемый регистр. Такое поведение характерно для многих процессорных ядер.

```

always @(posedge clk)
begin
  if (cmd == `WRITEMEM) Data[RegB] <= RegA;
  dmem <= Data[RegA];
end

```

Описание периферийных устройств состоит из описания регистра, хранящего данные, записываемые из RegA. В систему команд процессора добавлена специальная команда OUTLED, хотя обычно периферийные устройства не имеют специальных команд для доступа к их конкретному регистру. Следует использовать подключение периферийных устройств к системной шине процессора, определяя отдельно адрес устройства, и отдельно данные для него. Однако для простого примера специальная команда упростит запуск процессорного ядра и понимание принципов его работы.

```

always @(posedge clk)
begin
  if (cmd == `OUTLED) RegLED <= RegA[3:0];
end

```

```
assign led = RegLED;
```

Можно видеть, что описание разбито на описание поведение регистра и его подключение к выходным сигналам оператором `assign`. Это сделано потому, что выходы `led` не имеют спецификатора `reg` и записать в них данные по фронту тактового сигнала из-за этого невозможно. Добавить спецификатор `reg` в описание верхнего уровня в принципе можно, однако это не даст возможности прочитать значение этого сигнала, если впоследствии при модификации процессора возникнет такая необходимость. Показанный пример является более корректным с точки зрения сопровождения проекта.

11.4. Организация моделирования процессора

Чтобы проверить работу описанного процессора, можно воспользоваться поведенческим моделированием. Для этого необходимо создать файл модели, содержащий внутри подключенный модуль процессора и описание поведение внешних сигналов, подаваемых на этот модуль. Файл модели не имеет внешних сигналов, поэтому их список пуст. В показанном ниже примере объявлен модуль `cpu_top_tb`, где символы `_tb` не являются обязательными, но стилистически показывают, что это не файл для синтеза схемы, а файл модели (сокращение `tb` от «`testbench`»). Внутри необходимо объявить те же сигналы, что используются в процессоре в качестве внешних, однако для входов процессора необходимы сигналы `reg` (поскольку они будут хранить значения, подаваемые на входы процессора), а сигналы, подключаемые к выходам, объявлены как `wire` (поскольку управляющие ими регистры находятся внутри моделируемого процессора).

```
module cpu_top_tb(  
);  
  
reg clk_in;  
reg reset;  
reg rx;  
wire tx;  
reg [3:0] sw;  
wire [3:0] led;  
reg cmdwe;  
reg [15:0] cmdaddr;  
reg [35:0] cmddata;
```

После объявления сигналов к ним подключается процессор. Указать симулятору Verilog, что требуется определенный файл, можно по имени модуля этого файла. В данном случае это сри_top, поскольку именно такое имя было приведено в примере выше. Запись «сри_top uut» можно сопоставить с объявлением в языке Си «*int x*», где сначала указан тип создаваемого элемента, а затем выбранное для него имя. В данном случае сри_top – это «тип» модуля, а uut – выбранное имя. Оно также не регламентируется строго, а означает в данном случае сокращение от Unit Under Test.

```
сри_top uut (
    .clk_in(clk_in),
    .reset(reset),
    .rx(rx),
    .tx(tx),
    .sw(sw),
    .led(led),

    .cmdwe(cmdwe),
    .cmdaddr(cmdaddr),
    .cmddata(cmddata)
);
```

После «установки» в модель разработанного процессора необходимо описать сигналы, подаваемые на него. Обязательным является описание тактового сигнала. Оно показано в процессе always. Сигналу clk_in последовательно присваиваются значения 0 и 1, с паузой в 5 «единиц времени» между ними. По умолчанию в симуляторах это наносекунды. Такое назначение было бы невозможным для синтезированного кода (вызывая «короткое замыкание» 0 и 1 в одной и той же точке), однако при моделировании симулятор Verilog обрабатывает данные конструкции иначе. Процесс с ключевым словом always будет повторяться бесконечно по мере моделирования.

```
always
begin
    clk_in <= 1'b0;
    #5;
    clk_in <= 1'b1;
```

```
#5;  
end
```

Начальная установка сигналов процессора определяется процессом initial. Этот процесс, в отличие от always, выполняется только один раз при старте моделирования. Необходимо установить начальные значения для всех сигналов, в том числе и тех, которые не будут использоваться для проверки работы процессора. Это необходимо потому, что неопределенное состояние какого-либо сигнала вызовет при моделировании неопределенное состояние и того сигнала, который каким-либо образом от него зависит. В результате каскадного распространения таких неопределенностей симулятор может показать неопределенное состояние всего процессора.

```
initial begin  
    rx <= 1'b1;  
    sw <= 4'b0000;  
    cmdwe <= 1'b0;  
    cmdaddr <= 16'h0000;  
    cmddata <= 36'h0000000000;  
  
    reset <= 1'b1;  
    #20;  
    reset <= 1'b0;  
  
end
```

Для сигнала сброса задается установка в 1 на 20 нс с последующим переходом в 0. Это обеспечивает сброс регистров процессора.

В синтезированном коде была использована специальная директива Verilog readmemh. Она позволяет инициализировать описанную память из текстового файла. Можно задать содержимое памяти программы, инициализируемое файлом «program.mem», в котором будет выполняться простая программа, складывающая два числа, выводящая результат на светодиоды в двоичном коде, а затем демонстрирующая вызов подпрограммы и возврат из нее.

```
100000002 // RegA = 2  
000000001 // B <- A
```

```

100000003 // RegA = 3
00000003 // A = A + B
00000005 // A->LED
f00000010 // call 0x10
00000000 // nop
@10      // символ @ задает адрес, с которого следует продолжать
          // заполнение памяти
000000009 // RET

```

Завершив описание, можно запустить моделирование. Результаты моделирования показанного примера приведены на рис. 11.8.

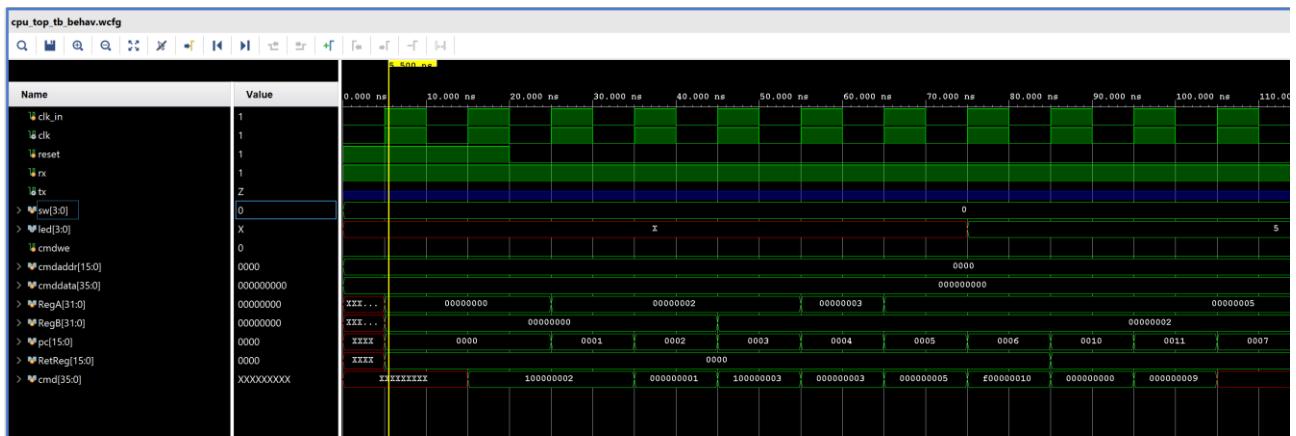


Рисунок 11.8. Пример моделирования процессорного ядра

Таким образом, показан пример разработки простого процессорного ядра и организации его моделирования. Приведенный пример специально выбран крайне простым (однако способным выполнить типичные для процессора операции) и очевидно требует развития.

11.5. Выводы по разделу

Процессорное ядро является полезным на практике инструментом для управления работой других узлов проекта. В отличие от конечного автомата, изменение работы процессора происходит не путем коррекции схемы, а путем замены содержимого памяти программ. Это можно сделать существенно быстрее, чем произвести еще одну итерацию синтеза, трассировки и программирования ПЛИС. Поэтому для организации сложных протоколов или задания последовательности смены сигналов часто полезно иметь в схеме процессор, даже если он и не является высокопроизводительным или совместимым с распространенными языками программирования.

При проектировании процессора важно начинать с общей регистровой структуры и формата команды. В следующей лекции будет проведена систематизация этих вопросов.

При описании RTL-представления следует придерживаться «от выходов» - always @ (posedge clk) reg <= newreg; вместо описания процессора «как программы». Удобнее начинать с поведения счетчика команд и загрузки литералов, они сильнее влияют на свойства процессора

Моделирование на системном уровне – удобный способ организовать анализ поведения схемы при выполнении программы. Наблюдать за процессором, выполняющим программу, существенно информативнее, чем тестировать отдельные узлы процессора, не имея возможности увидеть конечный результат их взаимодействия в процессе работы. Такое моделирование организуется с помощью нескольких несложных шагов.

1. Память заполняется программой
2. Подается тактовый сигнал
3. Подается сброс
4. Производится наблюдение смены состояния регистров и выходов процессора.

Контрольные вопросы:

1. Какие основные компоненты содержит процессор?
2. Как описать регистровый файл из N регистров с двумя выходами?
3. Какие действия выполняет процессорное ядро на каждом из тактов в 3-ступенчатом конвейере?
4. Что такое литерал и как он реализуется в процессоре?
5. Как организовать моделирование работы процессора?

12. ПРОЦЕССОРНОЕ ЯДРО – ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О ПРОЕКТИРОВАНИИ

12.1. Регистровая модель процессора и регистровый файл

Регистровая модель процессора описывает процессор в том виде, в котором он представляет интерес для программиста. Язык ассемблера предусматривает описание команд в виде тех действий, которые производятся с регистрами процессора, которые в целом могут восприниматься как переменные программы. Кроме регистров, хранящих обрабатываемые данные, ряд регистров имеет специальное назначение.

К обязательным регистрам относится счетчик команд. Традиционно он имеет имя PC (Program Counter), однако для процессоров Intel этот регистр называется IP (Instruction Pointer). С точки зрения схемотехники, этот регистр содержит адрес следующей команды, читаемой из памяти программ.

Для ранних вариантов процессоров x86 (как и для ряда других процессоров) используется сегментированная адресация. Это означает, что содержимое регистров, содержащих адреса, выступает в качестве смещения относительно некоторого базового адреса. Такой подход требуется из-за того, что 16-разрядный регистр способен адресовать только $2^{16} = 65536$ ячеек памяти, а в ранних процессорах i8086 адресуемая память составляла 1 Мбайт, что требовало 20 разрядов адреса. Поэтому адреса в i8086 формируются следующим образом. К начальному адресу, хранящемуся в специальном сегментном регистре, добавляется смещение согласно формуле $addr = segment * 16 + offset$, где $segment$ – содержимое сегментного регистра, $offset$ – смещение. Выполняемые команды читаются по адресу $cs * 16 + pc$. Для удобства такое выражение записывается в i8086 также как CS:IP. Сегментные регистры в i8086 имеют предопределенное назначение:

- CS (code segment) – сегментный адрес кода;
- DS (data segment) – сегментный адрес данных, используется по умолчанию для чтения и записи данных;
- SS (stack segment) – сегментный адрес стека, используется совместно с указателем стека SP;
- ES (extra segment) – дополнительный сегмент, использующийся в некоторых командах доступа к памяти.

В процессоре i80386 были добавлены также сегментные регистры FS и GS. Способы адресации процессора i80386 были существенно расширены относительно базового варианта i8086 и в целом являются предметом отдельного

рассмотрения. Здесь же имеет смысл ограничиться примером хорошо знакомой многим разработчикам архитектуры

Сегментированная адресация памяти была использована в i8086 для расширения адресного пространства, поскольку размер адреса оказался больше, чем размер регистров данных. Ее можно рассматривать как практический прием, преследующий те же цели (т.е. адресация очень больших объемов памяти), однако для практических примеров может оказаться неудобным следить за содержимым сегментного регистра. Постоянная перезагрузка сегментного регистра приведет к снижению производительности процессора.

Другим практически обязательным регистром является указатель стека. Для большинства процессоров он обозначается как SP (Stack Pointer). Стек является практически обязательным элементом процессорного проекта, так как вызов подпрограмм требует запоминания адреса возврата. Если адрес возврата запоминается в отдельном единственном регистре, вложенные вызовы подпрограмм станут невозможны. Простой путь обеспечения вложенных вызовов подпрограмм – организация области памяти в виде стека, называемого также память LIFO (Last In, First Out, т.е. «последним зашел – первым вышел»). Стек удобнее всего представлять в виде стопки листов бумаги, где последний листок, положенный на вершину стопки, будет первым с нее снят. Соответственно, команды вызова подпрограммы call и возврата из подпрограммы ret должны корректировать содержимое регистра SP, «помещая» и «снимая» числа со стека. Физически числа со стека не удаляются, вместо этого в регистр SP помещается новый адрес, указывающий на следующую ячейку памяти. Традиционно занятые ячейки стека располагаются в конце памяти, а помещение нового числа уменьшает регистр SP. Это связано с тем, что в ранних процессорах стек и данные располагались в одной и той же памяти, и было неудобно резервировать неизвестное заранее количество ячеек для стека в начальных адресах памяти. Поэтому вершина стека располагалась в наибольшем («старшем») адресе памяти и стек рос навстречу данным.

Основные операции с данными выполняются с помощью регистров общего назначения (РОН). Они обычно обозначаются буквами латинского алфавита A, B, C, или R0-Rx (например, R0-R7 или R0-R31). Для обозначения частей регистра используются символы L (Low, младшая часть) и H (High, старшая часть). Регистровая модель процессора i8086 показана на рис. 12.1. На этом рисунке видно, что регистры общего назначения разбиты на две части, каждая из которых является 8-разрядной. Части каждого регистра образуют 16-разрядный регистр,

обозначаемый собственным именем, т.е. название AX соответствует 16-разрядному регистру, а AH и AL – его половинам.

Другим важным регистром, связанным с обработкой данных, является регистр флагов. Он обычно обозначается F или Flags и содержит одноразрядные признаки выполнения арифметических и логических операций. Широко используемыми флагами являются:

1. Флаг нуля (Zero Flag, обозначается как Z или ZF). Этот флаг равен 1, если результат последней выполненной операции был равен нулю.

2. Флаг переноса (Carry Flag, CF). Устанавливается в 1, если при выполнении последней математической операции результат не уместился в разрядной сетке (произошел перенос в следующий, отсутствующий разряд или заем из отсутствующего разряда). С помощью флага переноса оказывается возможной организация операций с разрядностью большей, чем регистры общего назначения, поскольку установленный флаг переноса свидетельствует, что при сложении старших частей числа необходимо учесть перенос, полученный при операции над младшими частями.

3. Флаг четности (Parity Flag, PF). Устанавливается в 1, если количество разрядов результата, равных 1, является четным. Эта проверка явно отличается от проверки на четность самого результата (у всех четных чисел младший разряд двоичного представления равен 0), и используется для организации простейшей схемы целостности данных, принимаемых по внешним интерфейсам.

4. Флаг знака (Sign Flag, SF). Копирует старший разряд результата, поскольку в дополнительном двоичном коде старший разряд свидетельствует о знаке числа. Копирование этого разряда в отдельный регистр флага позволяет использовать команды условного перехода, выполняемые по результату проверки флага. Это быстрее, чем проверять старший разряд числа отдельной командой.

5. Флаг переполнения (Overflow Flag, OF). Этот флаг является вспомогательным для флага переноса и устанавливается в 1, если результат не может быть представлен в разрядной сетке регистра назначения. Отличием от флага переноса является то, что флаг переполнения устанавливается в случаях, когда число из-за переполнения изменяет свой старший разряд. Например, для 8-разрядного регистра результат операции $127+1$ формально помещается в 8 разрядов, однако получившееся число 128 из-за установленного старшего разряда будет трактоваться как отрицательное, хотя результат подразумевался положительным. Ввиду этого флаг CF будет равен 0, но флаг OF установится в 1.

Влияние различных команд на флаги является предметом отдельного рассмотрения. Например, общей практикой является то, что команды обычной загрузки в регистры не влияют на флаги, т.е. пересылка в регистр нулевого значения из другого регистра не устанавливает флаг нуля. Однако часто флаг переноса устанавливается в 0 при выполнении поразрядных логических команд, что отражает тот факт, что результат целиком поместился в регистр назначения.

Кроме флагов, устанавливаемых по результатам арифметических и логических операций, процессоры могут иметь флаги, управляющие их работой. Например, возможность реакции на прерывания может быть глобально разрешена и запрещена регулированием соответствующего флага (IF, Interrupt Flag в x86).



Рисунок 12.1. Регистровая модель процессора i8086

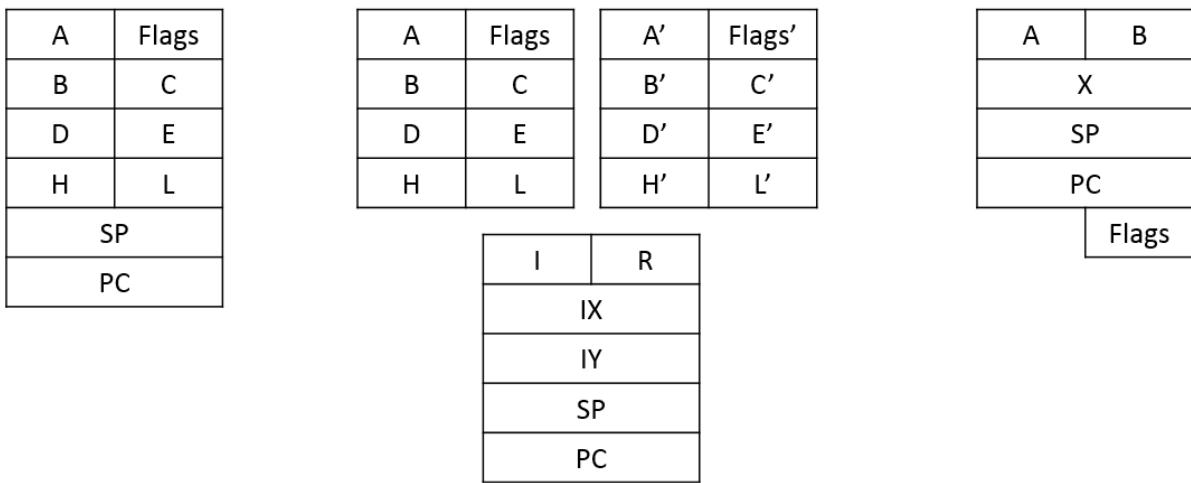
На рис. 12.2 приведены регистровые модели процессоров i8080 (Intel), Z80 (Zilog) и MC6800 (Motorola). Все эти процессоры относятся к середине 70-х годов и достаточно показательны для изучения. Процессор i8080 (имеющий советский аналог 580ВМ80) имел регистр-аккумулятор A, регистры общего назначения B, C, D, E, H, L и регистр флагов. Отдельными 16-разрядными

регистрами были описанные выше PC и SP. Для регистра-аккумулятора были доступны некоторые команды, которые не могли быть выполнены с другими регистрами общего назначения. С другой стороны, остальные регистры могли объединяться в 16-разрядные пары BC, DE и HL, которые использовались для адресации памяти и операций над 16-разрядными числами. Для пары HL был доступен более широкий перечень команд.

Таким образом, регистры процессора i8080 не полностью идентичны. Это свойство является достаточно важным при планировании совместной работы аппаратной и программной компонентов системы.

Процессор Z80 был создан в 1976 году компанией Zilog в качестве некоторого ответа на проект i8080. Сохраняя обратную совместимость по машинному коду (т.е. выполняя все программы для i8080), процессор Z80 имел также вспомогательные индексные регистры IX и IY (выполнявшие в целом ту же роль, что и регистровая пара HL), а также имел так называемый теневой набор регистров, показанный на рис. 12.2 в виде блока регистров со штрихами. В конкретный момент времени был активен только один из наборов, а быстро переключение наборов производилось всего одной командой. При этом выполнялось не физическое перемещение данных, а обычная перекоммутация, причем было невозможно определить, какой именно физический набор регистров активен в данный момент. Процессор Z80 оказал заметное влияние на российскую школу цифровой техники, поскольку был основой компьютера Sinclair ZX-Spectrum, популярного в России в начале 90-х годов и собирающегося самостоятельно широким кругом радиолюбителей.

Процессор MC6800 компании Motorola имел всего два регистра-аккумулятора A и B, индексный регистр X, а также обязательные регистры SP и PC. Интерес представляет наличие отдельного индексного регистра (этот процессор был выпущен раньше, чем Z80 с индексными регистрами IX, IY). Кроме того, вся работа с данными должна производиться с помощью всего двух регистров-аккумуляторов, что подразумевает интенсивное использование памяти.



i8080 (580BM80)

Z80

MC6800

Рисунок 12.2. Регистровые модели некоторых процессоров

Прежде чем приступить к перечислению возможных для процессора команд, необходимо рассмотреть основные подходы к проектированию системы команд. Бессистемное назначение кодов операций скорее всего приведет к хаосу как в схемотехнических решениях, так и в инструментальном программном обеспечении.

Наиболее известной классификацией является разделение процессоров на CISC и RISC. Эти аббревиатуры означают соответственно Complex Instruction Set Computer и Reduced Instruction Set Computer, т.е. компьютеры (процессоры) с полным набором команд и процессоры с сокращенным набором команд. Это разделение является уже достаточно старым, и соответствует выбору между сложным многотактным управляющим автоматом и простой системой работы, при которой каждый машинный цикл соответствует завершенной команде. Именно RISC-подход был приведен в примере процессора, рассмотренного в предыдущей статье цикла. Современные процессоры обычно выполняются в соответствии с архитектурой RISC, однако это не означает, что их возможности в чем-то сокращены. Речь идет о том, что в состав команд RISC включены только те команды, которые имеют простую реализацию и не требуют для выполнения нескольких машинных циклов. Команды CISC обычно могут быть выражены программно с помощью нескольких команд RISC.

Для системы команд можно также упомянуть несколько важных свойств. Первое из них это **ортогональность**, которое находится в тесной связи с регистровой моделью процессора. Под ортогональностью понимается наличие множества методов адресации данных, которые могут быть использованы с

любым сочетанием регистров процессора. Термин ведет происхождение от определения ортогональных (непересекающихся) векторов в геометрии. Под ортогональной системой координат, основанных на таких векторах, можно понимать некоторое пространство проектирования, в котором одно действие не оказывает влияние на другие. Например, в ортогональной системе команд можно выбирать один из операндов команды, не заботясь о том, какие ограничения это накладывает на второй операнд.

В примерах регистровых моделей, показанных выше, ортогональность в чистом виде нигде не наблюдается. Регистры часто имеют специальное назначение и с ними могут выполняться команды, недоступные для других регистров. Например, в i8086 регистр AX выступает аккумулятором, регистр BX – адресом памяти, CX – счетчиком циклов, а DX – операндом для команд умножения, деления и операций ввода-вывода. Можно указать, что введение частичной специализации позволяет в ряде случаев упростить аппаратную часть процессора и сократить разрядность команд. Примером специально введенной неортогональности является набор команд Thumb в процессорах ARM. В этом режиме команда имеет всего 16 разрядов вместо 32, однако доступны не все регистры и не все сочетания операндов.

Решение о том, следует ли использовать ортогональную систему команд, является результатом тщательных многоплановых исследований. В первую очередь играет роль используемая программная модель вычислений и типичные задачи, которые будут решаться с применением разрабатываемого процессора. Показательным примером является расширение Jazelle, которое, как и Thumb, используется как разновидность системы команд ARM. Это расширение предназначено для аппаратного ускорения выполнения приложений, написанных на языке Java, которые используют специфичную вычислительную модель – байт-код, выполняемый на виртуальной стековой машине.

Другой важной на практике характеристикой является **адресность** команд. Под адресностью понимается количество адресов (индексов регистров процессора), которые участвуют в выполнении команды. «Адрес» можно трактовать в широком смысле, как любое указание на ресурс процессора, участвующий в вычислениях или получающий результат.

Примером трехадресной команды является команда вида
 $R1 = R2 + R3$

В данном случае команда содержит указания на три регистра – получатель результата, первый operand и второй operand. Для того, чтобы определить все три регистра, команда должна иметь три битовых поля, в которых будут

помещены соответствующие номера. Размер этих полей зависит от количества регистров, которые можно адресовать таким способом. Например, 16 регистров потребуют 4-разрядных полей. Примером 3-адресных команд являются команды процессоров ARM.

В двухадресной команде регистр назначения совпадает с одним из операндов. Примером двухадресной архитектуры является Intel 8086. Например, рассматривая команду ассемблера *add ax, bx*, в ее описании можно увидеть, что действием команды является сложение данных из регистров *ax* и *bx* и помещение результата в *ax*. Таким образом, первый operand команды одновременно является и получателем результата.

Одноадресные команды используются в ряде процессоров цифровой обработки сигналов (например, Texas Instruments TMS320). В таких процессорах часто используется выделенный регистр-аккумулятор, который является и первым operandом, и получателем результата.

Нуль-адресная, или безадресная система команд также существует. Как следует из названия, она должна однозначно определять все operandы команд. Может показаться, что единственным вариантом, по аналогии с одноадресной архитектурой, является использование единственного возможного сочетания регистров для выполнения всех операций. Однако существует еще одна вычислительная модель, представляемая безадресными командами – стековые процессоры. В стековых вычислениях команды автоматически применяются к operandам, находящимся на вершине стека данных, а результат помещается также на стек. Примером является виртуальная стековая машина Java, аналогичный подход использован в Common Intermediate Language виртуальной машины .net корпорации Microsoft. Особенностью стековых вычислений является их независимость от конкретной регистровой архитектуры. Нетрудно представить, что машинный код, использующий только 4 регистра, будет недостаточно эффективен для процессора, имеющего 16 или 32 регистра. Однако код, ориентированный на 32 регистра, не сможет быть перенесен на процессор с меньшим количеством регистров. Стековое представление в данном случае предоставляет промежуточную вычислительную модель, которая может быть эмулирована любой регистровой архитектурой (представлением стека в памяти), а специальные процессоры с аппаратной поддержкой стека способны существенно ускорить выполнение стековых команд.

В стековом процессоре используется отдельный стек данных, на который не помещаются адреса возврата из подпрограмм (для этого используется обычный стек). Соответственно, используется отдельный регистр – указатель вершины

стека данных. Это сделано для того, чтобы отдельные подпрограммы стековой модели вычислений могли оставлять результаты на вершине стека, которые потом будут использованы другими подпрограммами.

Исторически стековая модель вычислений была реализована в языке программирования Форт (Forth). Кроме простой вычислительной модели, язык также отличается простой в реализации грамматикой, делающей его пригодным для быстрой разработки инструментального программного обеспечения. Речь идет в данном случае не столько об использовании существующих компиляторов Форта, сколько о реализации элементов Форт-машины в программном обеспечении для быстрого построения генераторов кода для новых процессоров.

Виды регистровых моделей с различной адресностью приведены на рис 12.3.



Рисунок 12.3. Виды регистровых моделей

Команды с адресностью, большей 3, также могут быть реализованы. В данном случае речь может идти о процессорах с несколькими командами, выполняемыми параллельно – т.н. архитектура VLIW (Very Long Instruction Word). Несколько выполняемых команд требуют и несколько наборов операндов, поэтому адресность может быть равна 4, 6 и более. Архитектура VLIW в настоящее время получила развитие в виде архитектуры EPIC ((Explicitly Parallel Instruction Computing, т.е. архитектура с явно заданным параллелизмом) и может быть реализована в ПЛИС вследствие возможности описания памяти с разрядностью 64, 128 и более.

Не вполне очевидным вариантом 4-адресной команды является задание, кроме индексов операндов и получателя, также и адреса следующей команды. Можно рассматривать такую команду в качестве параллельной выполняющихся

инструкций, одна из которых представляет собой обычную математическую операцию, а вторая – команду перехода. Обоснованность такого подхода определяется прежде всего интенсивностью использованию команд перехода, а также общими критериями эффективности, выбранными для процессора.

Вообще говоря, выбор архитектуры команд является предметом отдельного рассмотрения применительно к каждому конкретному проекту. Сильное влияние на эффективность выбранной организации команд оказывают типичные задачи, которые планируется решать на разрабатываемом процессоре. Поэтому невозможно заранее указать наиболее эффективную регистровую модель или подход к кодированию команд.

Пример формата команд для процессоров с различной адресностью показан на рис. 12.4. Можно убедиться, что при уменьшении адресности в команде освобождаются дополнительные поля, поэтому наибольшая компактность кода теоретически достижима в безадресной архитектуре, а трехадресная дает наибольшие возможности в генерации машинного кода.

Разряды								
3-адресная команда								
31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0	
type	operation	Dest	op1	op2				
type	operation	Dest			Literal			
2-адресная команда								
type	operation	Dest/op1	op2		Literal			
1-адресная команда								
type	operation	op1						
0-адресная команда								
type	operation							

Рисунок 12.4. Примеры организации команд для процессоров с разными видами регистровых моделей

Процессоры в целом имеют весьма разнообразные наборы команд, состав которых зависит от назначения процессора и ограничен его программной моделью. Тем не менее, можно выделить группы команд, которые часто присутствуют в процессорах.

Команды можно сразу разделить на крупные группы. Например, команды арифметики, сдвига и манипуляций с битовыми полями объединяет то, что они работают с данными и модифицируют регистры или память. Поэтому реализация еще одной команды подобного типа будет достаточно простой. Однако команды

перехода воздействуют на счетчик команд, и их реализация будет существенно отличаться от команд, работающих с АЛУ и регистрами.

Поэтому в поле type команды, показанной на рис. 5, можно занести код типа этой команды, выбирая его из списка

- безоперандные команды, такие как NOP (нет операции), прочие специальные команды, модифицирующие один из служебных регистров;
- команды работы с данными: арифметико-логические операции
- команды работы с памятью и внешними устройствами: чтение/запись, такие команды требуют формирования адреса для внешнего по отношению к процессорному ядру устройства;
- команды управления порядком выполнения программы: переходы, условные переходы, вызовы подпрограмм и возврат из подпрограмм; для этих команд следует рассмотреть организацию конвейера, а для команд вызова/возврата также и организацию работы со стеком.

Для указания на данные используются различные методы адресации. Список методов не является обязательным или исчерпывающим, однако процессор, неспособный указать на операнды ни одним из способов, не имеет практического смысла.

Регистровая адресация подразумевает указание на операнд, содержащийся в одном из регистров. Примером прямой адресации является выражение вида ax или r0, причем номер регистра в явном виде присутствует в поле команды. Это самый простой вид адресации, который, тем не менее, недостаточен для работы с процессором.

Непосредственная адресация (Indirect). Если прямую адресацию можно назвать «адресация посредством регистра», то при непосредственной адресации часть содержимого команды загружается в регистр назначения. Непосредственный operand также называют литералом (literal – «буквальный»). Например, если команда загрузки в регистр R0 имеет код 123, то загрузить в R0 число 2 можно командами «123 2». В этом примере подразумевается, что команда самостоятельно прочитает следующее число из памяти программ и использует его как литерал. Если размер команды достаточно большой, то литерал может быть размещен и внутри кода команды, как показано на рис. 5. Такой подход можно рассматривать только в качестве альтернативного, поскольку он упрощает кодирование на HDL, однако литералы занимают дополнительное место в коде команды.

Прямая адресация указывает на данные в памяти, адрес которых содержится в команде. Если рассматривать аналогию с предыдущим примером,

последовательность «123 2» должна загрузить число, содержащееся в ячейке памяти по адресу 2. Такая адресация требует дополнительной работы с памятью данных.

Косвенная адресация использует вместо адреса не число (литерал), а содержимое одного из регистров. В командах ассемблера такой способ адресации обычно обозначается скобками: `mov ax, [bx]`. Подразумевается, что регистр `bx` содержит адрес ячейки памяти, где расположены требуемые данные.

Эти методы адресации не образуют исчерпывающего списка. Например, способы адресации могут комбинироваться для достижения большей гибкости и более полного соответствия типовым алгоритмам. Например, для i80386 возможна команда

```
mov eax, dword ptr [ebp + 100 + ecx*4]
```

В этой команде используется сложный способ вычисления адреса, однако он может быть обоснован с точки зрения программирования. Представим, что в регистр `ebp` загружен начальный адрес сложной структуры данных, в которой со смещения 100 начинается массив 4-байтовых полей. Номер поля загружен в регистр `ecx`, поэтому выражение в квадратных скобках позволяет вычислить полный адрес интересующего нас элемента в сложной структуре данных. При этом содержимое регистров `ebp` и `ecx` не изменяется и может быть использовано в дальнейшем для вычисления адреса другого элемента.

Команды перехода используются для управления порядком выполнения программы. Эти команды работают преимущественно с регистром РС, загружая в него новое значение (или сохраняя линейный порядок выполнения, если речь идет об условном переходе).

Для команды перехода в языке ассемблера обычно используют производные от слова `Jump` («прыжок») или `Branch` («ответвление»). Мнемоника ассемблерной команды может выглядеть как `JMP`, `JP` или `BR`. Для команд перехода также справедливо подразделение по методам адресации – переход может быть совершен по адресу, указанному в коде, по адресу, содержащемуся в указанной ячейке памяти, или по адресу, хранящемуся в регистре. Полный набор способов представления адреса не является обязательным и применяется в процессорах с различными вариациями.

Кроме указания адреса, на который необходимо совершить переход, используется также относительная адресация перехода (*relative jump*). При этом указывается не сам адрес, а то, на сколько он отстоит от команды, которая

должна была бы выполняться при нормальном продолжении программы. Такой подход имеет как минимум два основных преимущества:

- программы становятся переносимыми, т.е. перемещение фрагмента программного кода по другому абсолютному адресу в памяти не нарушит относительное расположение команд, поэтому переходы с относительной адресацией будут по-прежнему выполняться правильно;
- для относительных переходов часто используется формат записи смещения в пределах одного или двух байтов, что сокращает размер программы, а переходы в пределах -128...+127 или -32768...+32767 адресов достаточно характерны для фрагментов программ, насыщенных проверками условий, так что команды `if` часто могут генерировать смещение, укладывающееся в короткий переход.

Команды условного перехода традиционно используют проверку условий, представленную флагами. Иными словами, для выполнения условного перехода сначала вычисляется условие, а затем производится переход при условии равенства нужного флага 0 или 1. Наличие в процессоре флага не делает обязательным добавлением команды условного перехода с проверкой этого флага (тем более для обоих вариантов значения флага). Перечисленные выше флаги, устанавливаемые по результатам выполнения команды в АЛУ, являются наиболее употребительными.

Команда вызова подпрограммы использует в основном те же способы представления адреса, как и команда перехода. Отличием от команды перехода является помещение на стек адреса команды, который будет использован для продолжения выполнения работы. В свою очередь, команда возврата из подпрограммы снимает со стека адрес и передает на него управление. Команды вызова подпрограмм также могут быть условными.

Отдельно можно упомянуть команды для работы с внешними устройствами. Несмотря на относительную простоту, управление работой внешних схем требует некоторого планирования. Например, запись данных во внешнее устройство необходимо представить отдельной командой, предусмотрев способ задания адреса устройства, данных, и назначив отдельный код команды, выполнение которого может не приводить к изменениям внутри ядра процессора, однако формировать специальный сигнал сопровождения (разрешения записи) для внешней схемы.

Регистровый файл – это группа регистров, имеющих сходное назначение. Для регистрационного файла обычно используется общий интерфейс, допускающий

единообразный доступ к любому регистру. Поэтому удобнее описывать регистровый файл в виде линейного массива регистров.

Рассмотрим процесс проектирования регистрового файла с учетом рассмотренных выше подходов к регистровым моделям процессоров. Модуль должен использовать тактовый сигнал и производить запись данных по фронту этого сигнала. Поскольку внутри регистрового файла имеется несколько регистров, необходимо указать номер (адрес) регистра для записи. Кроме того, в ряде случаев запись не производится (очевидно, при выполнении команды por, но это далеко не единственный пример), поэтому необходим отдельный сигнал разрешения записи данных.

Для получения операндов необходимо указывать их индексы. В общем случае они не совпадают с индексом регистра-получателя данных. Операнды являются выходами модуля.

Графическое изображение модуля регистрового файла представлено на рис. 12.5.

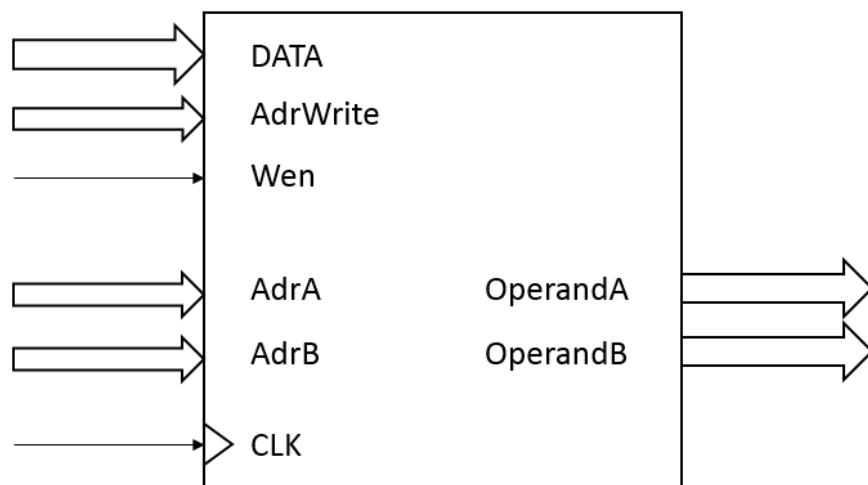


Рисунок 12.5. Интерфейс регистрового файла

Описание регистрового файла можно произвести на основе объявления двумерного массива. Запись в регистр, адресуемый сигналом AdrWrite производится по фронту тактового сигнала, где сигнал Wen (разрешение записи) глобально управляет этим процессом (т.е. if (wen) Reg[AdrWrite] <= Data). В зависимости от аппаратной платформы может быть использовано как поведенческое описание, так и структурное, с прямым указанием подключения требуемых аппаратных компонентов.

Чтение содержимого регистров с номерами AdrA, AdrB технически может производиться асинхронно, т.е. без фиксирования результата по фронту

тактового сигнала. Далее будет рассмотрен вариант трехступенчатого конвейера, где операнды не просто выбираются мультиплексором, а записываются в специальные регистры операндов по фронту тактового сигнала.

12.2. Арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ) – это основной вычислительный узел процессора, выполняющий операции над данными. АЛУ выполняет операции над входными operandами (получаемыми из регистрового файла или памяти данных), основываясь на коде выполняемой команды. Соответственно, его реализация достаточно проста - простой мультиплексор, управляемый кодом команды (по крайней мере, частью этого кода).

Состав команд АЛУ может варьироваться в достаточно широких пределах в зависимости от назначения процессора. В целом существует узнаваемый набор операций, характерных для АЛУ, но это не означает, что все операции или даже все группы операций должны быть реализованы в конкретном проекте. В целом можно отметить, что для АЛУ можно ориентироваться на возможности языка описания аппаратуры, поскольку в нем уже описаны арифметические и логические команды.

Тем не менее, можно привести примерный перечень операций, характерных для АЛУ. При этом важно отметить различие в используемых форматах чисел.

Беззнаковое представление числа соответствует ситуации, когда отрицательные числа представляются в так называемом дополнительном двоичном коде. Имеется в виду, что отрицательное число не имеет специального признака, а определяется путем дополнения. Это означает, что число $-X$ получается в результате вычисления выражения $(0 - X)$, а поскольку при ограниченной разрядности вычитание из нуля возможно только при заеме из несуществующего разряда, число -1 представляется в виде всех разрядов, установленных в 1. Проверить это можно, прибавив к такому числу 1. Теоретически, 8-разрядное число 255 ($0b11111111$) при сложении с 1 даст 1_0000_000 (подчеркиванием выделены группы разрядов), но так как для 8-разрядного представления старший разряд будет некуда записать, результатом операции будет 0. Следовательно, число $0b11111111$ можно трактовать как 8-разрядное представление числа -1 .

Беззнаковое представление удобно для сложения и вычитания. Можно свободно оперировать с двоичными числами, не заботясь об отдельной обработке знака числа. Если конечный результат помещается в разрядную сетку

регистра, любая последовательность сложений и вычитаний приведет к правильному результату.

Для смены знака числа, представленного в дополнительном коде, необходимо проинвертировать все разряды этого числа и прибавить к результату 1. Это можно проверить, изменяя знак числа 0000_0001. Инвертирование дает 1111_1110, что соответствует числу -2, тогда как исходный операнд был равен 1. Добавление к этому числу единицы даст правильное представление 1111_1111.

Знаковое (signed) представление чисел подразумевает, что в его старшем разряде находится признак знака (0 для положительного, 1 для отрицательного числа), а в остальных разрядах – модуль числа. Таким образом, 8-разрядное число -1 будет записано как 0b1000_0001.

Знаковое представление удобно для операций умножения и деления. Для этого достаточно умножить/разделить модули, а знак результата получить операцией ИСКЛЮЧАЮЩЕЕ ИЛИ над знаками операндов (разные знаки дадут в итоге 1, одинаковые – 0). Попытка перемножить числа, представленные в дополнительном коде, даст в итоге результат, соответствующий беззнаковому умножению. Поэтому знаковое и беззнаковое умножение должны быть реализованы отдельно.

Арифметические операции.

Сложение (addition) соответствует обычному сложению беззнаковых чисел. Сложение описывается выражением $result \leq a + b$.

Сложение с учетом переноса (addition with carry, часто используется mnemonic `adc`) добавляет к результату значение флага переноса. Такая операция используется для обеспечения обработки чисел, имеющих разрядность больше, чем разрядность регистров. После сложения младших частей чисел командой `add` будет установлено соответствующее значение флага переноса, и последующие команды `adc` для старших частей слагаемых будут учитывать перенос, возникший при сложении предыдущих частей.

Вычитание (subtraction) в целом аналогично сложению и описывается выражением $result \leq op1 - op2$. В отличие от сложения, важен порядок operandов.

Вычитание имеет свой аналог, учитывающий флаг переноса (subtraction with carry).

В ПЛИС умножение выполняется специализированными аппаратными блоками (DSP48 в FPGA Xilinx серии 7 и последующих, аппаратные умножители в предыдущих семействах). Синтезатор самостоятельно строит схему,

состоящую из нескольких умножителей, при необходимости перемножить числа большой разрядности.

Для прототипирования СБИС необходимо учитывать, что умножитель представляет собой компонент со сложной структурой (наиболее простой подход – т.н. «дерево сумматоров», попарно складывающее частные произведения первого операнда на каждый из разрядов второго операнда). Топологическая реализация дерева сумматоров может представлять проблему в смысле достижения хороших показателей площади, тактовой частоты и потребляемой мощности. В FPGA умножители являются аппаратным компонентом с гарантированными характеристиками, поэтому могут использоваться свободно.

Операция деления в общем случае не имеет простой реализации для произвольных операндов. Как правило, эта операция выполняется последовательно, за несколько тактов, с реализацией алгоритма «двоичного деления в столбик». Поэтому деление обычно не реализуется в АЛУ в прямом виде.

Логические операции обычно имеют поразрядные (битовые) представления. Это означает, что числа рассматриваются как наборы отдельных разрядов, и логические операции применяются к каждой паре разрядов. Это отличается от операций булевой алгебры, выполняемой над числами в формате boolean, так как если считать, что любое ненулевое значение соответствует ИСТИНЕ, то результат выражения 1 AND 2 должен также дать ИСТИНУ с точки зрения проверки логического условия. Однако в поразрядных представлениях этих чисел нет разрядов, которые имели бы значение 1 одновременно, поэтому результат поразрядного И будет равен 0.

Поразрядные логические операции имеют следующие варианты.

- поразрядное И (and).
- поразрядное ИЛИ (or).
- поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ (xor).
- поразрядная инверсия (not).

Возможны инверсные варианты этих операций, т.е., поразрядное И-НЕ и т.д., однако вопрос реализации полного набора поразрядных операций остается предметом изучения в каждом конкретном проекте. Частое использование какой-то операции в программах делает ее хорошим кандидатом на добавление соответствующей команды в АЛУ.

Операции сдвига имеют следующие варианты.

Логический сдвиг влево (shift left) описывается командой `result <= opa sll 1`, где команда `sll` является аббревиатурой Shift Left Logical. При выполнении этой команды битовое представление числа сдвигается влево, а в освободившийся младший разряд помещается 0. Допустим сдвиг на произвольное количество разрядов, если указать, например, `opa sll 3`. Сдвиг на переменное число разрядов требует реализации мультиплексора.

Логический сдвиг вправо (logical shift right) отличается от арифметического сдвига вправо (arithmetic shift right). Это связано с побочным эффектом операции сдвига. Сдвиг влево соответствует умножению на степени двойки, так что сдвиг на 1 разряд соответствует умножению на 2, сдвиг на 2 разряда – на 4 и т.д. Эта операция выполняется даже без аппаратного умножителя, который не имеет ограничений на значение сомножителя (с помощью единственного сдвига невозможно умножить на 3 или на 5). Аналогично, сдвиг вправо соответствует операции деления на степени двойки.

Если сопоставить эту информацию с наличием беззнакового представления числа, можно обнаружить некоторую проблему. Предположим, двоичное число -2 (имеющее представление `1111_1110`) делится на 2 операцией сдвига вправо. Если выполнить логический сдвиг, старший разряд станет равен 0, а младший разряд будет выдвинут за пределы разрядной сетки. Поэтому результатом сдвига будет `0111_1111`, т.е. 127. Таким образом, получен неожиданный результат: $-2/2 = 127$. Обязательное помещение логической единицы в старший разряд не решает проблему, поскольку теперь даже положительные числа будут превращаться в отрицательные. Правильным решением является помещение в старший разряд того же самого значения, которое было в нем до операции сдвига вправо.

Выполнять такое преобразование, или же реализовать обычный логический сдвиг, зависит целиком от смысла реализуемого алгоритма. Удобно иметь в составе АЛУ оба варианта сдвига несмотря на то, что они в принципе могут быть выражены через другие операции.

Операции вращения (rotation) подразумевают, что разряд, выдвигаемый из числа при вращении, помещается в него с другой стороны. Операция вращения также имеет разновидность, включающую в перемещение разрядов и бит переноса.

Команды сравнения используются для формирования логических (булевых) условий. Они могут быть как реализованы в АЛУ в виде отдельных операций, так и в качестве побочных эффектов при выполнении арифметико-логический

действий. Например, после операции `or ax`, `ax` в процессоре i8086 флаг нуля будет установлен, если в `ax` содержался 0, а флаг переноса будет обязательно сброшен.

Прочие команды АЛУ добавляются по мере необходимости и могут описывать действия, являющиеся комбинацией различных известных команд. Например, можно добавить обмен старшей и младшей части операнда, смену направления разрядов (так, что число `1010_0001` превратится в `0001_0101`), наложение битовой маски на результат и т.д. Основанием для введения дополнительных команд, как и в других подобных случаях, является их частое использование в практических программах, для которых предназначается процессор.

12.3. Конвейер процессора. Архитектура с трехступенчатым конвейером

Много процессорных архитектур являются конвейеризованными. Это означает, что преобразование данных происходит в них не за один такт, а с применением цепочки регистров, по которой данные из регистрового файла проходят последовательно, подвергаясь тем или иным преобразованиям на каждой стадии. Основной смысл конвейера – разбить линию с большой задержкой передачи данных на несколько более коротких линий, что позволит повысить тактовую частоту.

На рис. 12.6 показана структурная схема тракта данных процессора с регистровым файлом. На выходе регистрового файла находятся два мультиплексора, выбирающие из блока регистров операнды A и B. Эти операнды подаются на арифметико-логическое устройство, содержащее несколько вычислительных узлов для разных операций, выходы которых также подаются на мультиплексор, выбирающий один из результатов в зависимости от выполняемой команды.

На этом рисунке видно, что суммарная задержка такой схемы определяется задержками в ее составных частях. Если установить конвейеризующие регистры, можно будет повысить тактовую частоту. Очевидным местом для установки регистров является точка между мультиплексорами ОрA, ОрB и АЛУ. Возможно, она разделит задержку на не совсем равные части, однако именно здесь количество регистров будет минимально.

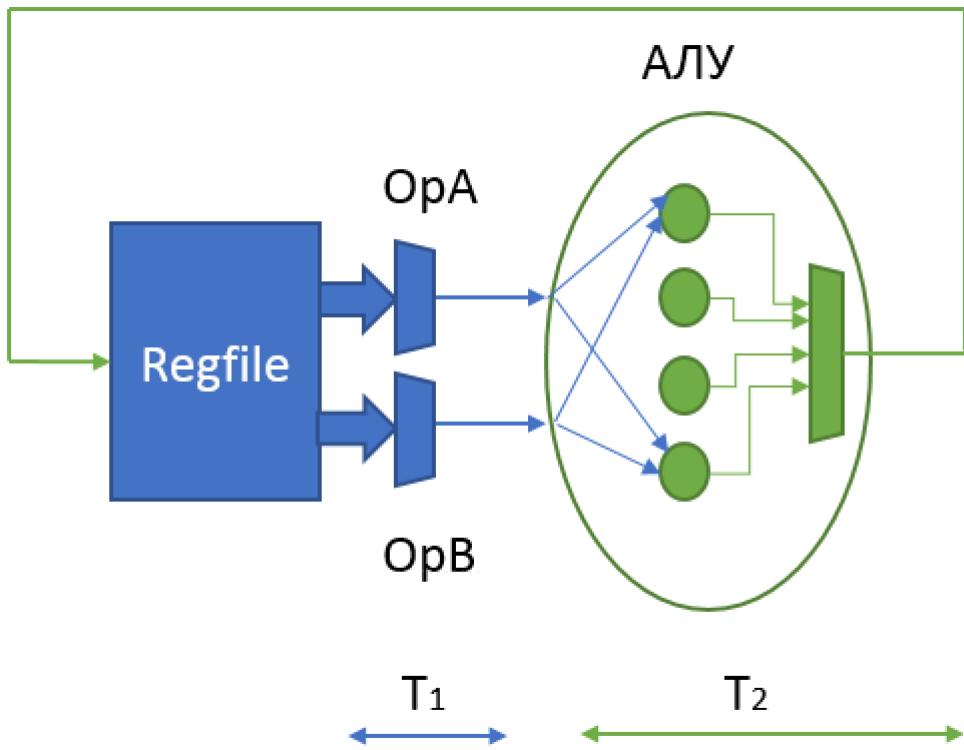


Рисунок 12.6. Иллюстрация к проблеме конвейеризации

Если произвести такую установку, работа процессора выполняется за три такта, что проиллюстрировано рис. 12.7.

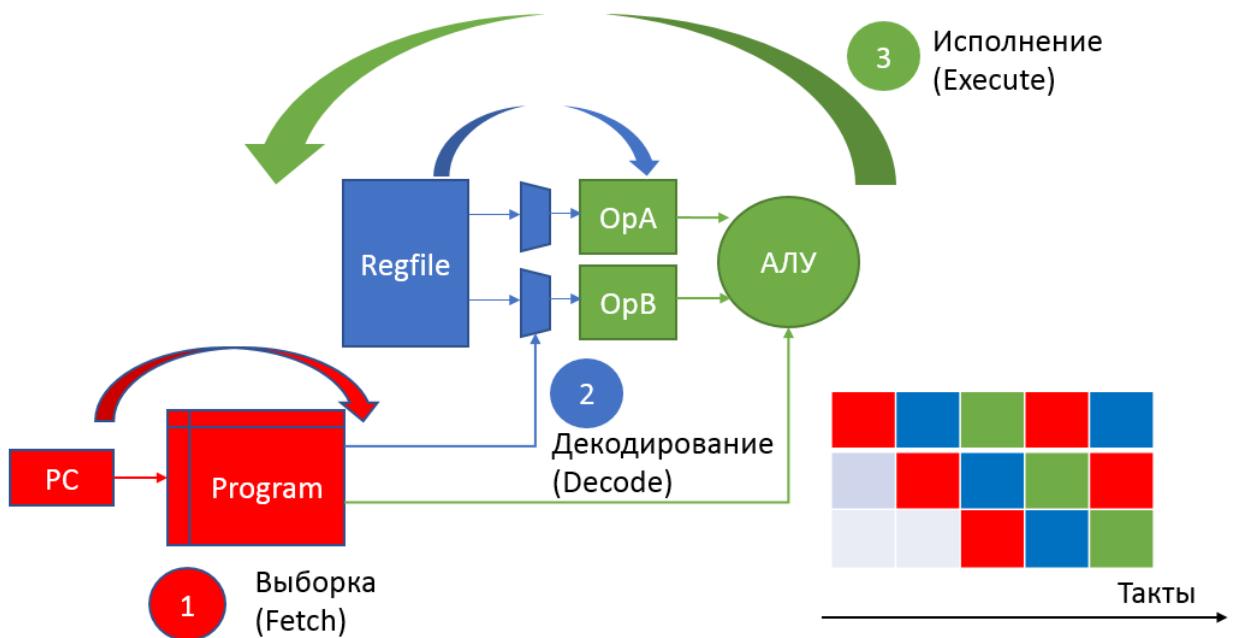


Рисунок 12.7. Порядок взаимодействия узлов процессора с трехступенчатым конвейером

На первом такте (выборка, fetch) по значению счетчика команд PC производится чтение кода очередной команды.

На втором такте (декодирование, decode) на основе команды из регистрового файла выбираются операнды, участвующие в этой команде. Здесь не показаны действия с другими узлами процессора, которые тоже зависят от кода команды, однако их также можно выполнять на стадии декодирования.

На третьем такте (исполнение, execute) уже имеются операнды. В соответствии с кодом команды результат требуемой операции записывается в регистр назначения.

Можно обратить внимание, что код прочитанной команды также необходимо передать на стадию исполнения через конвейеризующий регистр. Если выполнять стадии команды последовательно, то после чтения кода команды на следующем такте на выходе памяти программ будет уже код следующей команды, а следовательно, информация о выбранной операции и номере регистра назначения должна быть также передана на следующую стадию через конвейеризующий регистр.

Введение дополнительных стадий конвейера образует задержку выполнения программы на дополнительное число тактов. Рассмотрим программу следующего вида, проиллюстрированную рис. 12.8.

1. (PC=3). Выполнение $R2 = R0 + R1$. Из памяти прочитана команда JMP, вычисление $PC = PC + 1$.

2. (PC=4). Выполнение команды JMP, из памяти прочитана команда вывода в порт, присваивание $PC = 0x10$.

3. (PC=0x10). Выполнение команды вывода в порт, прочитанной на предыдущем такте, чтение новой команды.

Таким образом, задержка в конвейере на один такт привела к тому, что команда, читающаяся из последовательного адреса памяти программ одновременно с присваиванием нового значения PC, успела попасть на выход памяти программ и выполниться на следующем такте, когда новая команда по адресу PC=0x10 еще только читалась.

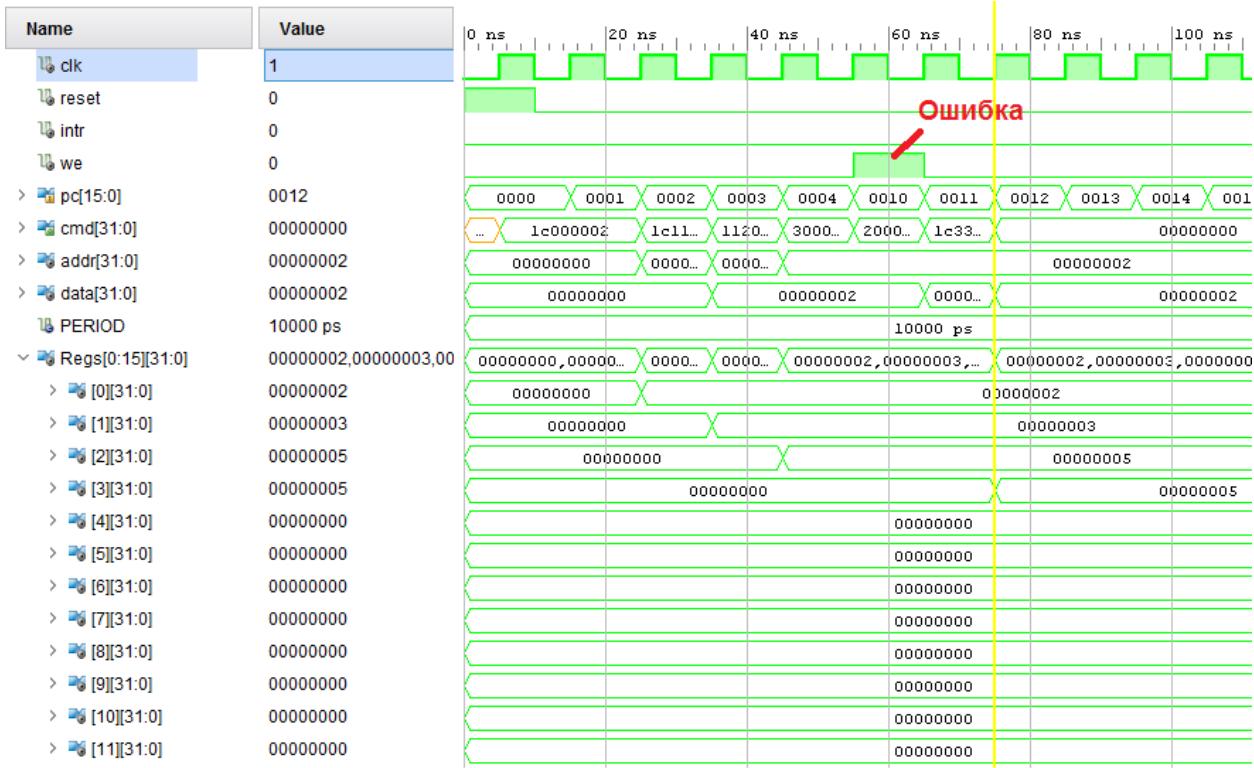


Рисунок 12.8. Временные диаграммы работы тестовой программы с выполнением инструкции, следующей за командой перехода

Для решения этой проблемы необходимо ввести *состояние конвейера*, как показано на рис. 12.9. Это специальный сигнал st, который сбрасывается, если процессор переходит не к адресу PC+1, и устанавливается в противоположном случае. Сигнал st проверяется при записи данных в регистры, память или внешние устройства, если он равен 0, запись не производится.

Введение в конвейер процессора состояния ожидания не является единственным возможным вариантом. В некоторых процессорных ядрах существует понятие *отложенного перехода* – команды, которые успели попасть в конвейер, выполняются независимо от того, что они находятся после команды перехода. Это создает дополнительную нагрузку на компилятор, который должен размещать команду перехода в коде перед тем, как она действительно потребуется, или же компилировать команды NOP после переходов, чтобы избежать нежелательных побочных эффектов.

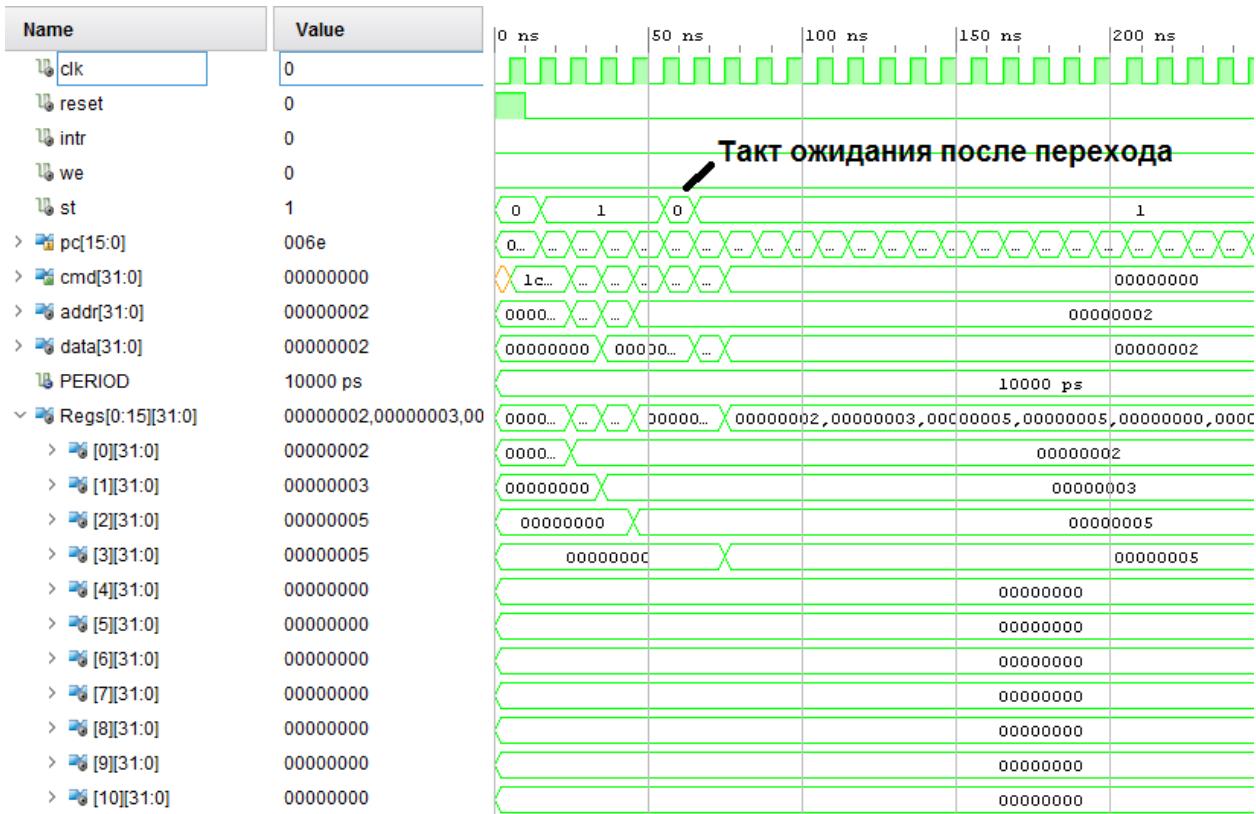


Рисунок 12.9. Временные диаграммы выполнения тестовой программы с введением такта ожидания после нарушения линейного порядка выполнения

12.4. Зависимости по данным. Архитектура MIPS

Рассматривая многотактные конвейеры, необходимо перейти к важной проблеме, которая неминуемо возникает при такой схеме – проблеме конфликтов и вызванных ими блокировок конвейера. Для иллюстрации можно использовать следующий псевдокод.

```
R1 = 1
R0 = 2
R1 = 3
R2 = R0 + R1
```

В приведенном листинге в регистры R0 и R1 загружаются операнды, которые затем складываются. Исходя из псевдокода, результатом должно быть число 5. Однако временные диаграммы, показанные на рис. 12.10, демонстрируют явную ошибку – вместо сложения чисел 2 и 3 складываются 2 и 1. Из псевдокода видно, что 1 – это ранее загруженное в R1 значение, и именно оно оказалось использовано для команды сложения.

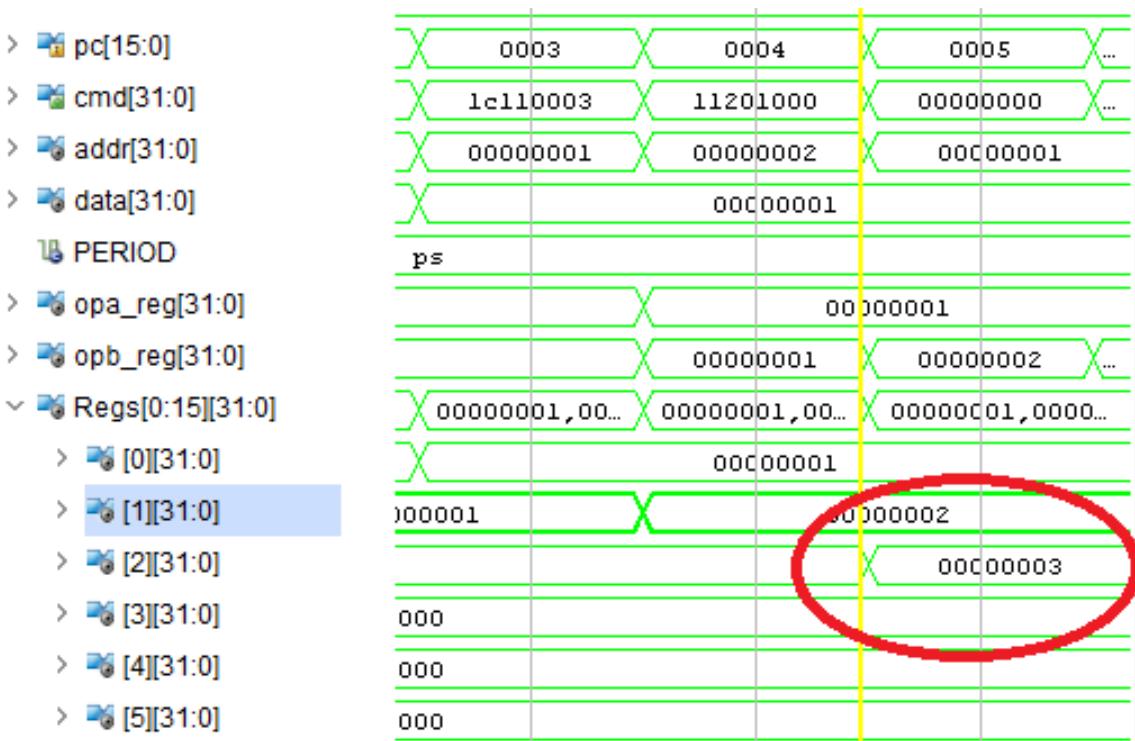


Рисунок 12.10. Проблема зависимости по данным в конвейеризованном процессоре

Такое поведение объясняется тем, что из-за конвейеризации процессору требуется дополнительный такт, чтобы обновить значение в регистре. В момент выполнения команды $R1 = 3$ следующая команда находилась на стадии «декодирование», т.е. читала операнды из регистрового файла. Поскольку новое значение для $R1$ находилось еще только на входе данных регистрового файла, по очередному фронту тактового сигнала произошло следующее:

1. Старое значение $R1$ записано в регистр операнда.
2. Новое значение записано в регистр $R1$.

Таким образом, в данном случае имеет место *зависимость по данным*. Она проявляется в том, что результат выполнения команды зависит от того, какие команды выполнялись непосредственно перед ней. Всего существуют 4 ситуации взаимного влияния команд.

1. «Чтение после чтения» (RAR, Read After Read) – из одного и того же физического ресурса (регистра или ячейки памяти) производятся два последовательных чтения. Эта ситуация не формирует конфликтов, поскольку первое чтение не влияет на содержимое регистра и не мешает прочитать его второй раз.

2. «Чтение после записи» (RAW, Read After Write) – наиболее проблемная ситуация, соответствующая показанной на рис. 4. При наличии конвейеризации может возникнуть ситуация, когда данные, предназначенные для записи, еще не

попали в регистр (или память), однако следующая команда, находясь в конвейере, уже производит их чтение.

3. «Запись после чтения» (Write After Read)
4. «Запись после записи» (Write After Write)

Эти ситуации, в принципе, не вызывают проблем. Обращать внимание на них следует только в случае более сложного варианта микроархитектуры, который еще не был рассмотрен, и соответствует случаю нескольких параллельно работающих конвейеров с внеочередным выполнением команд (обозначается термином *out-of-order*). По подобной схеме работает множество высокопроизводительных процессоров, где показательным примером является первый вариант процессора Intel Pentium, имевшего два конвейера (т.н. U- и V-конвейеры). В более сложном конвейере команды могут быть запущены по мере готовности данных для них и выполняться путем прохождения разного количества стадий конвейера. Может возникнуть ситуация, когда команда, прочитанная из памяти позже, запустится на исполнение и будет исполнена за меньшее количество тактов, чем предыдущая команда. В этом случае необходимо проверить, не используют ли эти команды один и тот же ресурс процессора, так как порядок записи в регистр назначения может быть нарушен.

Проектирование многотактных конвейеров с внеочередным выполнение команд требует существенно большего внимания, поскольку требуется отслеживать потенциальные конфликты по доступу к ресурсам процессора и приостанавливать выполнение команд в конвейерах, если не готовы операнды для них, или регистр, в который должна быть произведена запись, еще не был прочитан командой, запущенной на исполнение ранее.

Кроме того, в процессе доступа в память может возникнуть т.н. структурный конфликт. Если в конвейере предусмотрена последовательность «чтение данных из памяти – обработка данных – запись данных в память», то возможна ситуация, когда одна команда производит запись в память, но другая команда в то же время находится на стадии «чтение данных из памяти». В этом случае придется сделать выбор, какую из команд выполнять, поэтому либо не будет произведена запись, либо чтение из памяти не состоится. Разумеется, оба варианта приводят к ошибочному поведению процессора.

Процессор без блокировок в конвейере (*Microprocessor without Interlocked Pipeline Stages*) – это и способ организации конвейера, и конкретный продукт, разработанный компанией MIPS Technologies (ранее MIPS Computer Systems). В данной архитектуре используется полезный прием разрешения конфликтов по

доступу к регистрам, называемый «продвижение данных» (data bypass). Его суть можно проиллюстрировать на рис. 12.11.

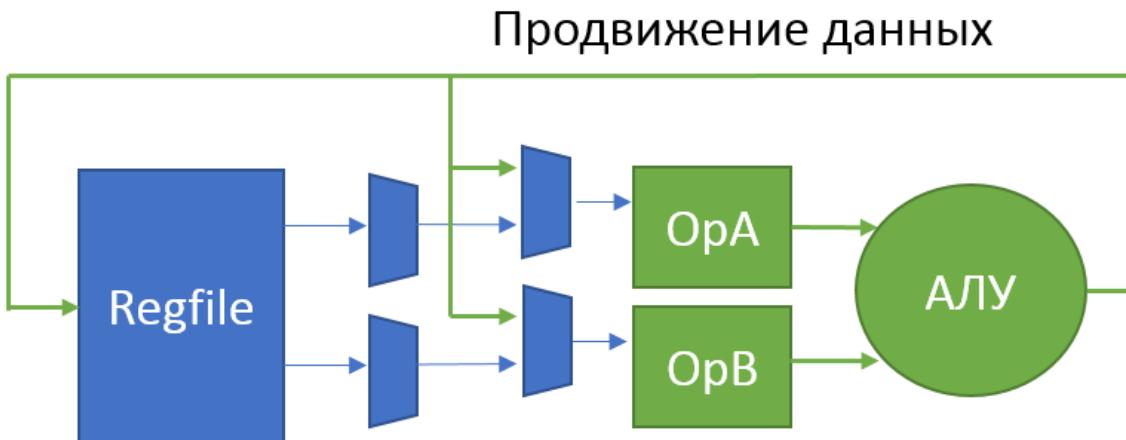


Рисунок 12.11. Прием «продвижение данных» для решения проблемы зависимости по данным в конвейере

На рис. 12.11 можно видеть способ разрешения конфликта при последовательном доступе к одному и тому же регистру на запись и на чтение. Как было показано выше, такая ситуация должна вызвать конфликт, поскольку в момент записи нового значения на вход АЛУ попадает старое содержимое регистра. При реализации подхода data bypass новое значение записывается не только в сам регистр, но и в регистр операнда, если номер регистра для записи результата совпадает с номером регистра-операнда.

Несмотря на то, что на рис. 12.11 показан дополнительный мультиплексор, при должном подходе снижение тактовой частоты такой схемы окажется несущественным. Для этого необходимо убедиться, что данные из АЛУ подаются именно на последний мультиплексор в цепочке, т.е. после мультиплексирования одного из регистров в регистры OpA, OpB записывается либо значение регистра, либо данные из АЛУ. В этой случае суммарная задержка распространения сигнала от OpA, OpB через АЛУ увеличится всего на один мультиплексор 2-в-1.

12.5. Архитектуры с многоступенчатым конвейером

Дальнейшим развитием данной архитектуры является 5-ступенчатый конвейер, который также часто используется в процессорных ядрах. При рассмотрении 3-ступенчатого конвейера можно убедиться, что с его помощью невозможно обеспечить чтение данных из синхронной памяти за один машинный цикл (т.е. при прохождении команды по всем трем стадиям).

Действительно, при выполнении команды вида `mov R0, [R1]` будут выполнены следующие действия:

1. Команда прочитана из памяти программ.
2. Содержимое регистра R1 записано в выходной регистр и подано на вход адреса для памяти.
3. Содержимое памяти по адресу, определяемым регистром R1, прочитано и находится на выходе блока памяти.

Соответствующая схема соединений показана на рис. 12.12. Поскольку адрес получается на стадии вычисления результата в АЛУ, он может иметь достаточно сложную структуру – например, являться суммой двух регистров, регистра и константы и т.д.

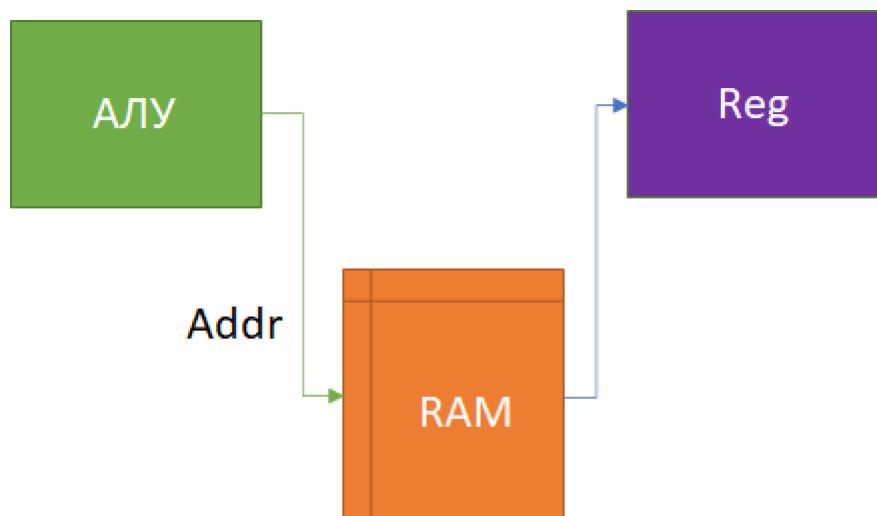


Рисунок 12.12. Стадия работы с памятью в процессоре с 5-ступенчатым конвейером

Очевидно, что для завершения команды требуется еще один такт – запись прочитанного значения в регистр R0 (т.е. в регистр назначения). После трех тактов удалось добиться только того, что требуемое значение появилось на выходе блока памяти данных. Данная проблема имеет несколько схемотехнических решений, одно из которых – введение двухцикловых команд чтения из памяти. В таком случае первая команда инициирует собственно процесс чтения, а вторая (которая завершится на такт позже) запишет данные, находящиеся на выходе блока памяти, в регистр назначения.

Можно попытаться использовать в качестве адреса памяти значение операнда до конвейеризующего регистра. Тогда после чтения команды на стадии декодирования будет прочитана память данных, а на стадии исполнения прочитанное значение будет готово для записи. Однако такой подход

потенциально вызывает конфликты другого типа, которые будут рассмотрены далее.

При интенсивной работе с памятью в конвейер могут быть добавлены такты, на которых и будут происходить чтение и запись в память. При 5-ступенчатой архитектуре эти такты носят название «Memory Phase» и «Writeback» (рис. 12.13). На 4-м такте производятся операции с памятью, а на 5-м (writeback) – запись результата по назначению.



Рисунок 12.12. Выполнение операций в процессоре с 5-ступенчатым конвейером

По мере того, как происходило распространение RISC-архитектур, 3- и 5-ступенчатые конвейеры получили широкое распространение в качестве наиболее употребительных. Однако в цифровой схемотехнике известно, что конвейеризация позволяет разбить критические пути распространения сигнала на отдельные стадии и тем самым увеличить тактовую частоту. Такой подход увеличивает латентность, и в определенный момент все же заставит вводить такты ожидания, если будет иметь место существенная зависимость по данным. Однако для целого ряда приложений, например, мультимедийных, или, в более широком смысле, задач цифровой обработки сигналов, можно использовать сильно конвейеризованные схемы для обработки потока входных значений.

Увеличение количества стадий конвейера можно было наблюдать в процессорах Intel Pentium. Если процессор P5 (Pentium первого поколения) имел 5 стадий, то Pentium III использовал уже 10, а Pentium 4 – 20 стадий (по некоторым данным, ядро Prescott имело 31-ступенчатый конвейер). Можно отметить, что тактовые частоты Pentium 4 при этом достигали 3-4 ГГц, что сопровождалось и ростом потребляемой мощности – до 120 Вт. В конечном итоге подход, использованный в Pentium, был заменен в процессорах Intel Core Solo/ Core Duo на более короткий конвейер, что снизило тактовую частоту, однако итоговая производительность оказалась выше.

Недостатком сильно конвейеризованной архитектуры является чрезмерное возрастание сложности контроля зависимости по данным. В длинном конвейере перед чтением данных из регистра необходимо проверять, что ни одна из стадий конвейера не занимается выполнением операции, которая при своем завершении обновит этот регистр. Подход *data bypass* имеет здесь ограниченное применение,

поскольку может быть необходимым передача данных не просто на предыдущую стадию, а на стадию, отстоящую достаточно далеко по конвейеру.

В конечном итоге, конвейеризация имеет двойственный эффект. С одной стороны, при конвейеризации тракта данных возрастает его тактовая частота (однако этот процесс рано или поздно войдет в насыщение, поскольку сложно обеспечить совершенно равномерное разбиение тракта данных на стадии). С другой стороны, добавление стадий конвейера усложняет проверки, поэтому частота системы управления, контролирующей возможность передачи данных на каждую из стадий, будет падать. Этот процесс условно показан на рис. 12.14, где нужно отметить, что этот рисунок не соответствует какой-то определенной архитектуре.

На рис. 12.14 можно видеть, что при чрезмерно агрессивной оптимизации частота тракта данных будет достаточно высокой, но при этом общая частота процессора станет ограничиваться уже сложностью управляемых систем, проверяющих конфликты из-за зависимостей по данным на разных стадиях конвейера.

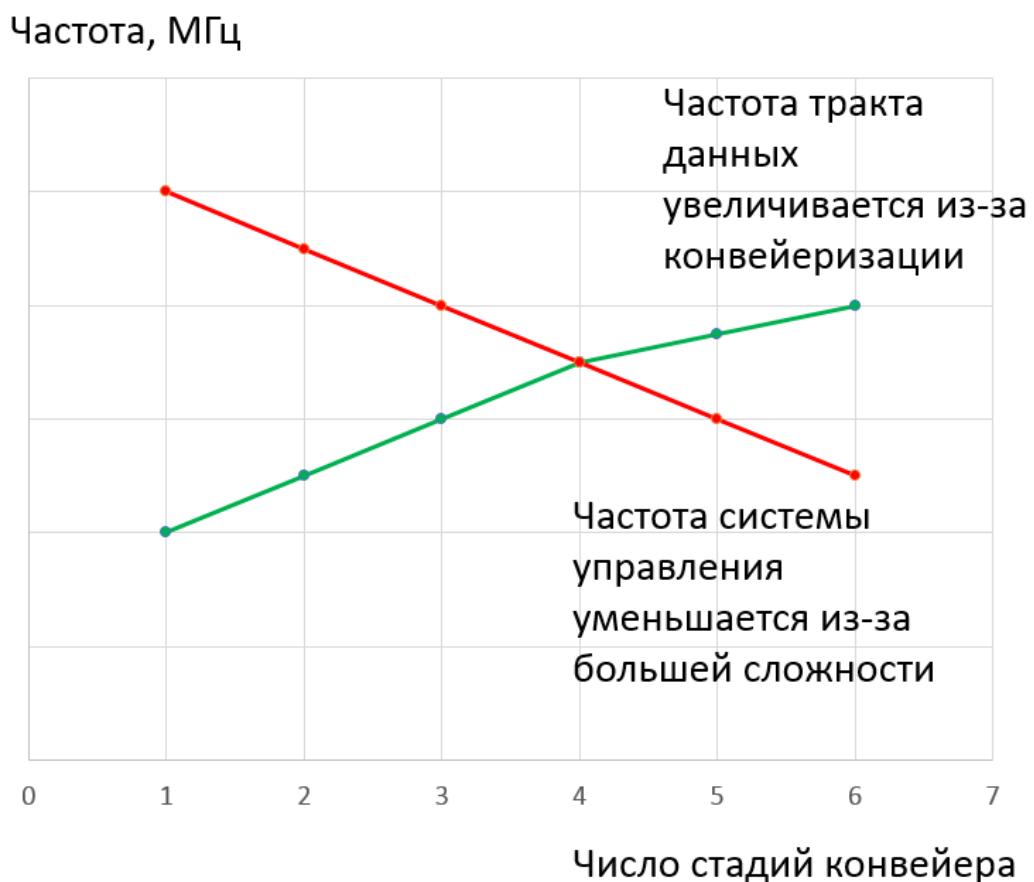


Рисунок 12.14. Пример (абстрактный) увеличения сложности управляемой схемы многоступенчатого конвейера

12.6. Архитектура VLIW (сверхдлинное командное слово)

Сверхдлинное командное слово требуется, если в команде процессора оказывается много отдельных полей. Потребность в дополнительных полях может возникнуть, если за один такт будут параллельно выполняться несколько операций. Пример организации конвейера данных VLIW-процессора показан на рис. 12.15. На этом рисунке показано, что несколько регистровых файлов (Reg1 – Reg3) через коммутаторы подают operandы на несколько АЛУ (АЛУ1 – АЛУ3). Коммутаторы могут иметь разные варианты организации (в примере показано, что выходы любого регистра могут быть поданы на входы любого АЛУ), поэтому в таком процессоре возможно параллельное выполнение трех разных команд в трех АЛУ. Подобный подход соответствует *параллелизму уровня инструкций* (ILP, Instruction Level Parallelism).

Размер команды на рис. 12.15 увеличивается, потому что каждому из АЛУ необходимо определить выполняемую на этом такте операцию, а каждому коммутатору – выбираемый им operand. Однако это не означает, что размер команды пропорционален количеству АЛУ. Если предположить, что команда имеет 32 разряда, добавление АЛУ потребует 4-5 разрядов для выбора операции, а дополнительный мультиплексор потребует 2-3 разряда (в зависимости от количества его входов).

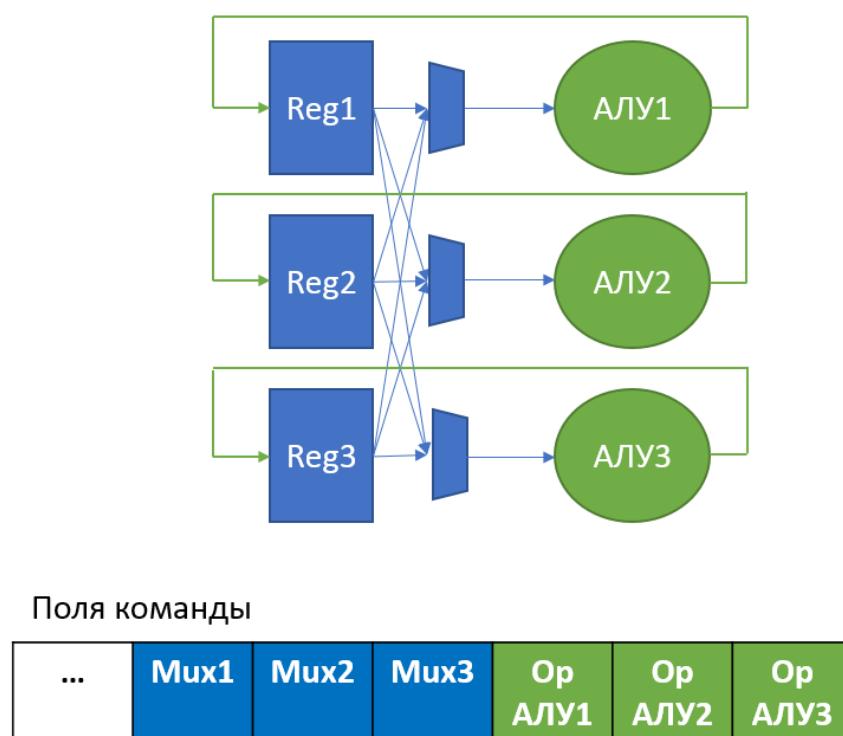


Рисунок 12.15. Принцип организации процессора с архитектурой сверхдлинного командного слова (VLIW)

Размер команды для VLIW-процессоров указывается в пределах 128 - 256 разрядов. Рекордсменом является процессор Avispa+ компании Silicon Hive, имеющий 768-разрядную команду, однако при такой разрядности команды этот процессор имеет 60 слотов операций.

Преимуществом VLIW является, как можно понять, возможность выполнения нескольких операций за один такт. Однако это формальное преимущество не всегда можно использовать на практике. Для того, чтобы можно было параллельно выполнить несколько операций, для них должны быть готовы данные. Если типичные целевые алгоритмы предусматривают в основном линейные последовательности команд, в которых вычисленный результат используется как operand следующей команды, дополнительные АЛУ, скорее всего, будут простаивать. Поэтому для VLIW большую роль играют предварительные исследования алгоритмов и возможности используемого компилятора.

Архитектура VLIW имеет разновидность, отмечающую тот факт, что в процессоре происходит прямое управление всеми вычислительными и коммутационными устройствами. Такая архитектура называется Explicit Parallel Instruction Computer (EPIC) – процессор с явным параллелизмом. Если для VLIW-процессоров часто использовались аппаратные планировщики, обеспечивавшие автоматический выбор operandов для АЛУ по мере их готовности, то в EPIC делается больший упор на возможности компилятора по прямому управлению всеми устройствами процессора. Как и для VLIW, сам факт применения архитектуры EPIC не позволяет достичь увеличения производительности, если особенности выполняемой программы не предполагают параллельных вычислений.

12.7. Прототипирование процессоров на базе ПЛИС

Необходимо разделить проекты процессоров на прототипы будущих заказных микросхем (ASIC – Application-Specific Integrated Circuits) и на так называемые софт-процессоры – процессоры, собираемые из программируемых ячеек и изначально проектируемые для будущей эксплуатации в составе проектов в ПЛИС. Формально прототип процессора также является софт-процессором, поскольку он состоит из программируемых ресурсов ПЛИС. Однако смысл разделения заключается в том, что решения, эффективные для ПЛИС с учетом архитектуры этих микросхем, не обязательно будут эффективны для ASIC и наоборот. Поэтому важно понимать, будет ли проект в ПЛИС макетом будущей микросхемы или же самостоятельным изделием. В первом

случае оптимизировать проект следует не с точки зрения САПР ПЛИС и архитектуры этих микросхем, а с точки зрения той технологической библиотеки, которая будет впоследствии использована при производстве процессора. В этом случае тактовая частота макета процессора в ПЛИС может быть невысокой.

Если ПЛИС предполагается как основная платформа для изделия, для практических целей можно ответить на следующие вопросы:

1. Предусматривает ли задача интенсивное использование несложных параллельных вычислений?

Очевидным применением являются приложения цифровой обработки сигналов, получающие существенные преимущества от независимых параллельно работающих блоков «умножение с накоплением» (компоненты DSP48 в современных FPGA Xilinx). В этом случае софт-процессор будет играть в системе вспомогательную роль и служить только для упрощения настройки цифровых фильтров, ИР-ядер БПФ и прочих подобных устройств. Он может иметь весьма умеренную тактовую частоту (поскольку не является компонентом, определяющим производительность системы), однако должен обеспечивать поддержку внешних интерфейсов, осуществляющих доступ к подсистеме цифровой обработки сигналов со стороны оператора. Подобную роль в настоящее время успешно выполняют софт-процессоры MicroBlaze.

2. Имеются ли в системе независимые процессы, требующие обработки в режиме реального времени?

Данная задача является весьма неоднозначной и требующей тщательного внимания со стороны разработчиков. При попытке ее решения с помощью одного процессорного ядра (пусть даже и заявленного как real-time) перед программистом неминуемо возникает целый ряд вопросов. Каков реальный промежуток времени, проходящий между поступлением в процессор запроса на обработку и переходом к формированию выходного управляющего воздействия? Этот ответ существенно отличается от «сколько тактов процессор тратит на вход в прерывание», поскольку, например, срабатывание аварийного датчика требует отключения соответствующего силового компонента, и факт входа в прерывание еще не означает, что процессор уже послал сигнал отключения в соответствующий порт. Аппаратные решения способны обеспечить задержку реакции порядка десятков наносекунд, что существенно превосходит возможности программных решений (впрочем, подобные возможности часто избыточны).

Однако кроме обеспечения минимального времени реакции на внешние воздействия системы реального времени могут поставить и другую, более

сложную задачу. Каким образом необходимо обрабатывать запросы, если подпрограммы их обработки перекрываются во времени? Наличие только одного потока исполнения команд вынуждает программиста устанавливать приоритеты в обработке запросов, причем добавление нового устройства или даже изменение основной программы требует проверять, не нарушается ли нормальная работа с внешним оборудованием, формирующим запросы на прерывания. Проблемы здесь вполне возможны, например, из-за применения библиотек или фрагментов кода, запрещающих прерывания в процессе выполнения каких-либо действий.

Простейшим решением со стороны soft-процессоров здесь является применение нескольких процессорных ядер, в том числе и специально выделенных для работы с критичными источниками запросов на обработку. Здесь можно говорить не только о датчиках аварийных состояний, но и, например, о большом количестве периферийных устройств, использующих медленные протоколы обмена данными. Чтобы не тратить ресурсы основного процессора на их реализацию, можно использовать не только аппаратные ускорители уровня конечных автоматов, но и несложные процессорные ядра, оптимизированные для простых операций с портами периферийного устройства.

Можно отметить, что популярные микроконтроллеры семейства PIC появились именно в подобном качестве. Аббревиатура PIC расшифровывается как Peripheral Interface Controller, т.е. «контроллер периферийных интерфейсов», и первый микроконтроллер PIC появился в качестве микросхемы для расширения возможностей ввода-вывода процессора СР1600. Даже ПЛИС начального уровня способны вместить несколько несложных процессоров, которые существенно облегчат работу программиста, освобождая его от необходимости «жонглировать» в основной программе несколькими независимыми процессами вычислений.

3. Имеются ли в задаче специфичные вычисления, последовательности команд или шинные циклы?

Положительные ответы на эти вопросы могут являться основанием для разработки собственного процессорного ядра. Действительно, если обратиться к первой части данного цикла статей, в ней имелась ссылка к статье, анализирующей микроконтроллеры начального уровня. Там упоминалось, что операции записи в выходной порт (микроконтроллерный hello world в виде мигания светодиода) требуют от 3 до 20 тактов в некоторых процессорных ядрах. Естественным решением было бы ускорение таких операций не только путем добавления соответствующих периферийных IP-ядер, но и путем введения в процессор специализированных команд, например, для ускорения доступа к

периферии, для выполнения автоинкремента/декремента, групповых арифметических и логических операций и т.п. При этом умеренная тактовая частота процессора будет не так важна, если процессор будет выполнять значимые для пользователя фрагменты кода за меньшее число тактов.

12.8. Выводы по разделу

Разработка процессорного ядра имеет следующие основные этапы.

1. Проектирование и моделирование.

Данный этап не потребует существенных материальных и временных затрат. Можно использовать практически любой язык программирования высокого уровня для работы с программными моделями процессора и бесплатные версии САПР ПЛИС для экспериментов с RTL-описанием и моделирования на уровне регистровых передач. В итоге разработчик сможет выявить работоспособность принятых им технических решений, оценить тактовую частоту и объем ресурсов, которые будут заняты в ПЛИС.

2. Реализация в макете.

Простые отладочные платы на базе ПЛИС начального уровня вполне позволяют продемонстрировать практическую работоспособность процессора и создать примеры устройств, управляющих периферийным оборудованием. При умеренных финансовых затратах можно будет продемонстрировать практическое функционирование процессорной системы.

3. Внедрение в практику.

Наличие информации о достижимых характеристиках процессора и возможных путях его доработки, расширения и адаптации позволяет говорить о его внедрении в практику, в первую очередь, по месту основной работы в профильной организации. Как было упомянуто, важное практическое преимущество оригинальной архитектуры заключается в том, что она может быть быстро адаптирована для управления сложными IP-ядрами, расположенными на кристалле ПЛИС. Без удобного в применении процессора отладка таких ядер производилась бы с помощью длительных итераций моделирования, правки управляющих непрограммируемых контроллеров и генерации новых конфигурационных файлов. Управление сигналами IP-ядер со стороны встроенного в проект процессора, по меньшей мере, сократит количество итераций создания конфигурационных файлов.

Контрольные вопросы:

1. Сколько регистров требуется процессору? В чем преимущества и недостатки большого и маленького регистрового файла?
2. Почему во многих процессорах регистры имеют дополнительные функции?
3. Что происходит с конвейером процессора при переходе к новому адресу?
4. Что такое зависимость по данным? В чем заключается негативный эффект от ситуации, когда следующая команда читает регистр, который записывается предыдущей командой?
5. Что происходит с конструкцией процессора при увеличении количества тактов конвейера? Как это отражается на выполнении программ?
6. В архитектуре со сверхшироким командным словом используется несколько АЛУ. Какие препятствия могут возникнуть, если разработчик ожидает пропорционального роста производительности такого процессора?

13. СИСТЕМНЫЕ ШИНЫ ПРОЦЕССОРНЫХ УСТРОЙСТВ

13.1. Системная шина в компьютерной системе

Системная шина является неотъемлемой частью проекта на базе процессора и служит для соединения процессорного ядра и периферийных устройств. После работ по созданию ядра необходимо обеспечить его подключение к периферийным устройствам, без чего наглядно продемонстрировать практический эффект от процессора невозможно. Может оказаться так, что неэффективная реализация системной шины сведет на нет усилия по оптимизации процессорного ядра.

Современные микроконтроллеры обычно имеют в своем составе достаточно много периферийных устройств. Пример фрагмента документации, содержащего таблицу регистров управления, показан на рис. 13.1.

23. Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x3F	SREG	I	T	H	S	V	N	Z	C	page 8
0x3E	SPH	—	—	—	—	—	—	SP9	SP8	page 11
0x3D	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	page 11
0x3C	Reserved	—	—	—	—	—	—	—	—	
0x3B	GIMSK	—	INT0	PCIE	—	—	—	—	—	page 51
0x3A	GIFR	—	INTF0	PCIF	—	—	—	—	—	page 52
0x39	TIMSK	—	OCIE1A	OCIE1B	OCIE0A	OCIE0B	TOIE1	TOIE0	—	pages 81, 102
0x38	TIFR	—	OCF1A	OCF1B	OCFOA	OCFOB	TOV1	TOV0	—	page 81
0x37	SPMCSR	—	—	RSIG	CTPB	RFLB	PGWRT	PGERS	SPMEN	page 145
0x36	Reserved	—	—	—	—	—	—	—	—	
0x35	MCUCR	BODS	PUD	SE	SM1	SM0	BOODE	ISC01	ISC00	pages 37, 51, 64
0x34	MCUSR	—	—	—	—	WDRF	BORF	EXTRF	PORF	page 44,
0x33	TCCR0B	FOC0A	FOC0B	—	—	WGM02	CS02	CS01	CS00	page 79
0x32	TCNT0	—	—	—	—	—	—	—	—	page 80
0x31	OSCCAL	—	—	—	—	Oscillator Calibration Register	—	—	—	page 31
0x30	TCCR1	CTC1	PWM1A	COM1A1	COM1A0	CS13	CS12	CS11	CS10	pages 89, 100
0x2F	TCNT1	—	—	—	—	—	—	—	—	pages 91, 102
0x2E	OCR1A	—	—	—	—	—	—	—	—	pages 91, 102
0x2D	OCR1C	—	—	—	—	—	—	—	—	pages 91, 102
0x2C	GTCCR	TSM	PWM1B	COM1B1	COM1B0	FOC1B	FOC1A	PSR1	PSR0	pages 77, 90, 101
0x2B	OCR1B	—	—	—	—	—	—	—	—	page 92
0x2A	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	—	—	WGM01	WGM00	page 77
0x29	OCR0A	—	—	—	—	—	—	—	—	page 80
0x28	OCR0B	—	—	—	—	—	—	—	—	page 81
0x27	PLLCSCR	LSM	—	—	—	—	PCKE	PLLE	PLOCK	pages 94, 103
0x26	CLKPR	CLKPCE	—	—	—	CLKPS3	CLKPS2	CLKPS1	CLKPS0	page 32
0x25	DT1A	DT1AH3	DT1AH2	DT1AH1	DT1AH0	DT1AL3	DT1AL2	DT1AL1	DT1AL0	page 107
0x24	DT1B	DT1BH3	DT1BH2	DT1BH1	DT1BH0	DT1BL3	DT1BL2	DT1BL1	DT1BL0	page 107
0x23	DTPS1	—	—	—	—	—	—	DTPS11	DTPS10	page 106
0x22	DWDR	—	—	—	—	DWDR(7:0)	—	—	—	page 140
0x21	WDTCR	WDIF	WDIE	WDOP3	WDCE	WDE	WDOP2	WDOP1	WDOP0	page 45
0x20	PRR	—	—	—	—	—	PRTIM1	PRTIM0	PRUSI	PRADC
0x1F	EEARH	—	—	—	—	—	—	—	—	page 20
0x1E	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	page 21
0x1D	EEDR	—	—	—	—	—	EEPROM Data Register	—	—	page 21
0x1C	EECR	—	—	EEPM1	EEP0	EERIE	EEMPE	EEPE	EERE	page 21
0x1B	Reserved	—	—	—	—	—	—	—	—	
0x1A	Reserved	—	—	—	—	—	—	—	—	
0x19	Reserved	—	—	—	—	—	—	—	—	
0x18	PORTB	—	—	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	page 64
0x17	DDRB	—	—	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	page 64
0x16	PINB	—	—	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	page 64
0x15	PCMSK	—	—	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	page 52
0x14	DIDR0	—	—	ADC0D	ADC2D	ADC3D	ADC1D	AIN1D	AIN0D	pages 121, 138
0x13	GPIOR2	—	—	—	—	General Purpose I/O Register 2	—	—	—	page 10
0x12	GPIOR1	—	—	—	—	General Purpose I/O Register 1	—	—	—	page 10
0x11	GPIOR0	—	—	—	—	General Purpose I/O Register 0	—	—	—	page 10
0x10	USIBR	—	—	—	—	USI Buffer Register	—	—	—	page 115
0x0F	USIDR	—	—	—	—	USI Data Register	—	—	—	page 115
0x0E	USISR	USISIF	USIOIF	USIPF	USIDC	USICNT3	USICNT2	USICNT1	USICNT0	page 115
0x0D	USICR	USISIE	USIOIE	USIWMM1	USIVWM0	USIC51	USIC50	USICLK	USITC	page 116
0x0C	Reserved	—	—	—	—	—	—	—	—	
0x0B	Reserved	—	—	—	—	—	—	—	—	
0x0A	Reserved	—	—	—	—	—	—	—	—	
0x09	Reserved	—	—	—	—	—	—	—	—	
0x08	ACSR	ACD	ACBG	ACO	ACI	ACIE	—	ACIS1	ACIS0	page 120
0x07	ADMUX	REFS1	REFS0	ADLAR	REFS2	MUX3	MUX2	MUX1	MUX0	page 134
0x06	ADC5RA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	page 136
0x05	ADCH	—	—	—	—	ADC Data Register High Byte	—	—	—	page 137
0x04	ADCL	—	—	—	—	ADC Data Register Low Byte	—	—	—	page 137
0x03	ADC5RB	BIN	ACME	IPR	—	—	ADTS2	ADTS1	ADTS0	pages 120, 137
0x02	Reserved	—	—	—	—	—	—	—	—	
0x01	Reserved	—	—	—	—	—	—	—	—	
0x00	Reserved	—	—	—	—	—	—	—	—	

Рисунок 13.1. Регистры управления периферийными устройствами 8-разрядного микроконтроллера (фрагмент технической документации)

Этот пример соответствует простому 8-разрядному микроконтроллеру, в более сложных микросхемах подобных регистров существенно больше. Добавить в систему команд процессора отдельные команды для всех действий с периферийными контроллерами крайне сложно и обычно неэффективно. Поэтому периферийные устройства подключаются к процессорному ядру с помощью небольшого набора сигналов, а обмен данными с ними производится с помощью команд `in` и `out`. Эти команды оперируют парой аргументов – адрес устройства и данные для него.

Назначением системной шины является объединение отдельных компонентов процессорной системы и обеспечение взаимодействия между ними. Минимальный состав системной шины можно проиллюстрировать рис. 13.2. В целом системная шина подразделяется на шину адреса (ША), шину данных (ШД) и шину управления (ШУ). Конкретный состав сигналов и правила работы определяются многими факторами, с учетом широкого спектра системных шин и особенностей их назначения. Например, применявшаяся в ранних поколениях РС системная шина ISA использовала сигналы стробирования вместо тактового сигнала, 8-разрядную шину данных (с вариантом в 16 разрядов), и была предназначена для подключения внешних устройств с помощью плат расширения. Соответственно, скорость обмена по этой шине была невысокой.

Следует заранее разделить шины на внешние и накристальные. К внешним шинам предъявляются несколько иные требования, выполнение которых в пределах одного кристалла может оказаться нецелесообразным. Например, важным вопросом является реализация шины данных. В примере на рис. 13.2 используются две отдельные шины данных, отличающиеся направлением передачи. Вместе с тем для внешних шин данные обычно мультиплексированы, т.е. используется одна двунаправленная шина, переключающая направление передачи в соответствии с сигналами управления (чтение/запись).

Шина, показанная на рис. 13.2, не обеспечивает несколько часто используемых возможностей:

- предварительная проверка готовности устройства, к которому производится обращение (*handshaking*, «рукопожатие»);
- передача управления шиной другому устройству («захват шины»).

Эти возможности требуют введения дополнительных сигналов и будут рассмотрены далее.

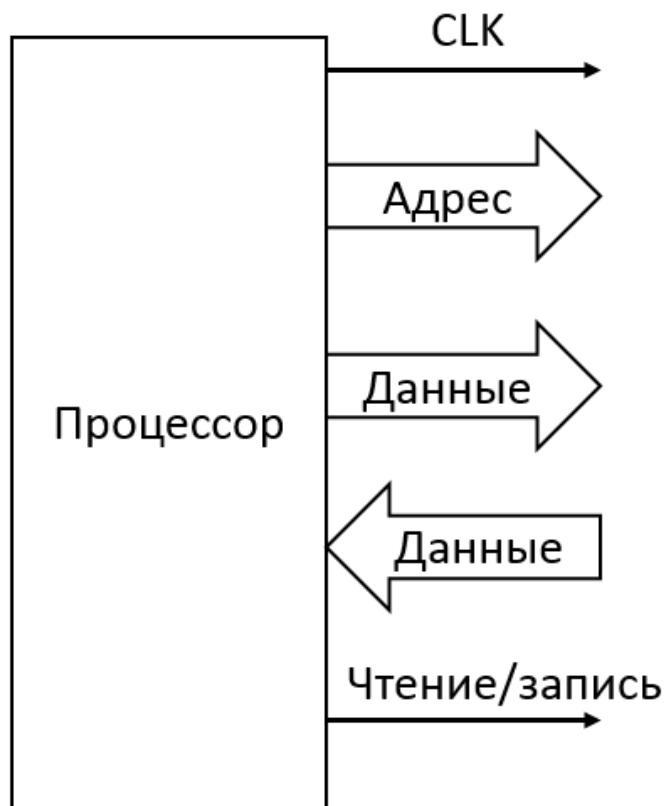


Рисунок 13.2. Общий вид интерфейса простой системной шины

13.2. Некоторые виды системных шин в компьютерной технике

В ранних поколениях персональных компьютеров IBM PC использовалась системная шина ISA. Ее основные сигналы и примерный вид временных диаграмм показаны на рис. 13.3.

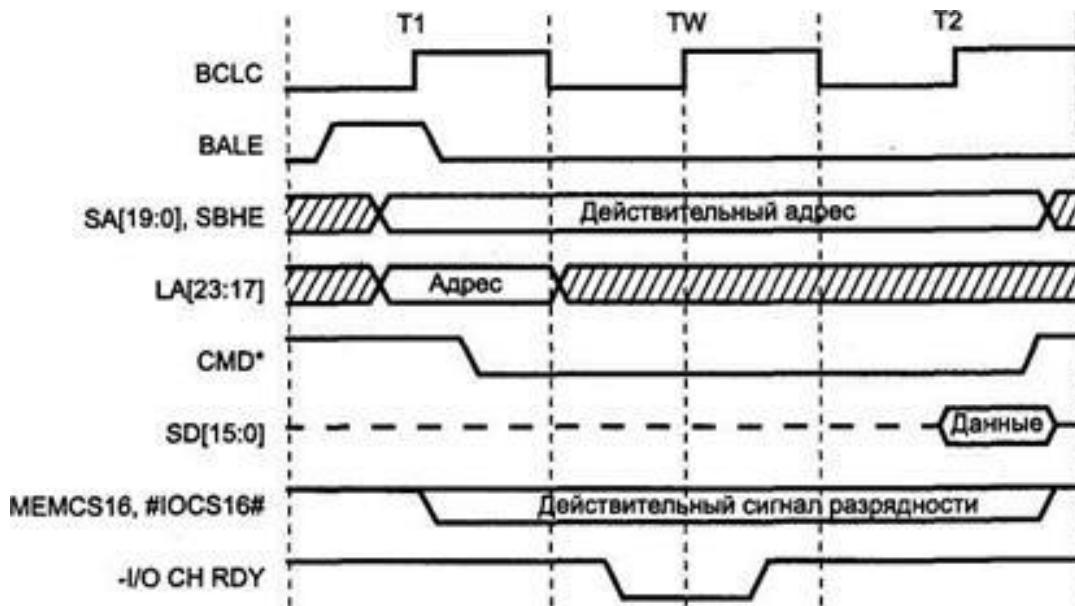


Рисунок 13.3. Временные диаграммы системной шины ISA ранних модификаций IBM PC

Передача данных по шине ISA производится асинхронно, под управлением сигналов чтения и записи. На рис. 13.4 показана временная диаграмма записи с основными сигналами шины. В момент положительного перепада сигнала IOW (Input-Output Write) шина адреса содержит адрес требуемого устройства, а шина данных – действительные данные от процессора для записи. Длительность логического нуля для сигнала IOW регламентирована в документации на шину и имеет гарантированное минимальное время действия, на которое может ориентироваться устройство, подключаемое к шине. Подразумевается, что быстродействие периферийного устройства позволяет ему за задаваемое стандартом время проверить адрес, сравнить его с адресами своих регистров, и при совпадении подать на вход Write Enable этого регистра разрешающий сигнал.

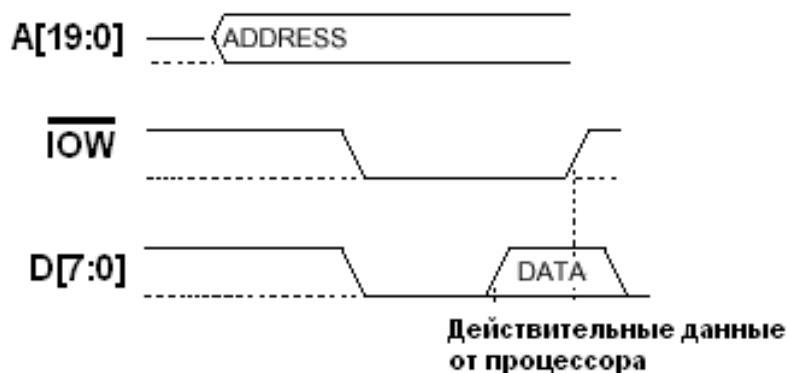


Рисунок 13.4. Временная диаграмма записи в шине ISA

Шина ISA в настоящее время практически не используется в ПК, за исключением ее промышленного аналога PC-104, который имеет те же сигналы, но другой вид разъема. Быстродействие шины, управляемой сигналами IOW, IOR, ограничено тем, что для таких сигналов необходимо обеспечивать минимальную длительность, а при попытке передавать данные быстрее можно столкнуться с тем, что выпущенные ранее периферийные устройства не будут успевать реагировать на поданные сигналы адреса и своевременно формировать сигналы разрешения для своих регистров.

Дополнительные резервы повышения быстродействия обеспечивают синхронные шины. Например, шина PCI, долгое время используемая в ПК, имеет тактовый сигнал с частотой 33 МГц (или 66 МГц, если все подключенные периферийные устройства явным образом подтвердили поддержку такой частоты). Сигналы и временные диаграммы PCI показаны на рис. 13.5.

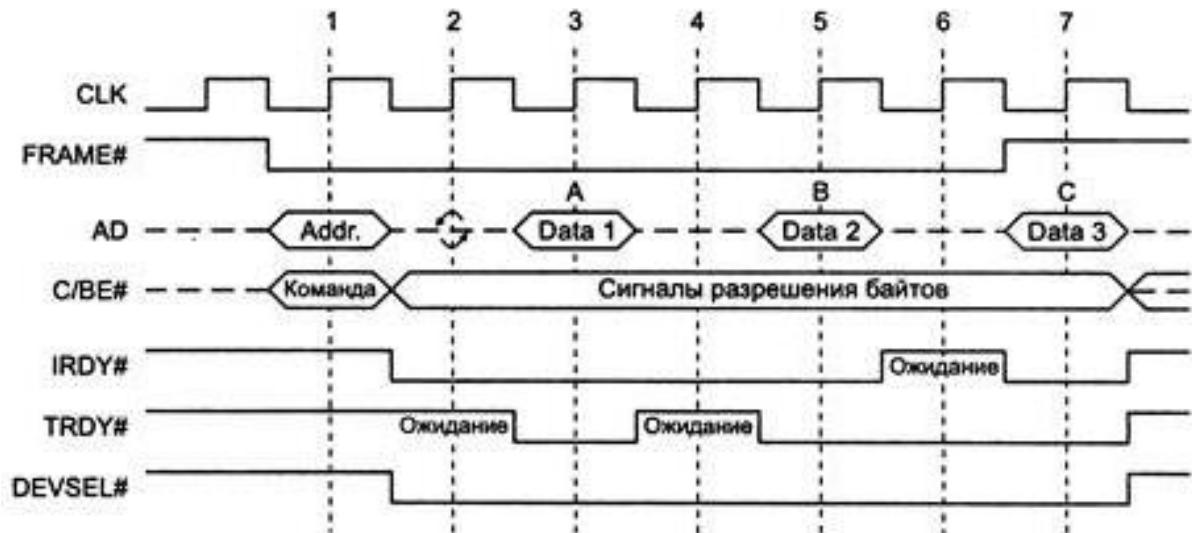


Рисунок 13.5. Временные диаграммы системной шины PCI

Шина PCI обеспечивает более широкие возможности по сравнению с ISA. Например, периферийное устройство не обязано успевать проверить поданный адрес за один период тактового сигнала. Процесс работы шины показан на рис. 13.6.

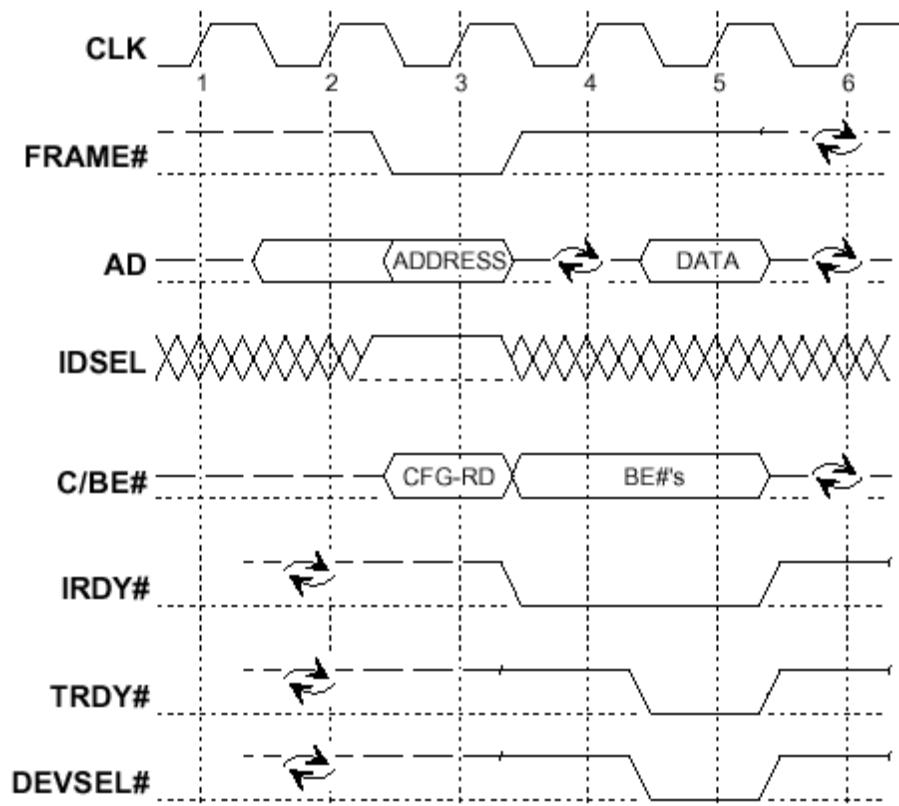


Рисунок 13.6. Временные диаграммы системной шины PCI

Поскольку тактовый сигнал CLK действует постоянно, начало работы с устройством определяется по активному уровню сигнала #FRAME. Он имеет

низкий активный уровень, т.е. «кадр» (frame) начинается, когда #FRAME становится равным 0. В этот момент на шине AD (Address/Data) присутствует адрес, а на шине C/BE# (Command/ Byte Enable) – 4-разрядный код команды. На первом такте кадра AD и С имеют именно эти значения (адрес и код команды), после чего они переключаются на передачу данных, сопровождаемых сигналами разрешения отдельных байтов (Byte Enable). Это позволяет передавать по 32-разряднойшине AD только один байт, если поставить в сигнале BE# активным только один бит.

Процесс передачи регулируется еще двумя сигналами - #IRDY (Initiator Ready, передатчик готов) и #TRDY (Target Ready, приемник готов). Если приемник не готов, контроллер PCI продолжает удерживать данные и сигнал IRDY. Как только он получает в ответ активный #TRDY, по следующему фронту тактового сигнала происходит передача данных.

Другой возможностью PCI является возможность конфигурирования устройств нашине. Для ранних версий ISA одной из проблем было возможное совпадение адресов нескольких периферийных устройств, вставляемых в разные слоты системной платы. Для этого производители предусматривали набор перемычек на плате, которые позволяли установить разные начальные («базовые») адреса блока регистров, которые присутствовали на подключаемой плате. Правильная настройка нескольких плат была важным шагом сборки компьютера.

Для шины PCI реализована функция автоматического распределения адресов подключаемых устройств по неперекрывающимся диапазонам. Для этой цели каждое устройство PCI реализует специальный набор управляющих регистров, доступных по специальной команде. Для доступа к блоку этих регистров каждый слот PCI имеет независимый сигнал выбора. Состав регистров PCI показан на рис. 13.7.

Базовым адресом блока регистров управляет регистр BAR (Base Address Register). В каждом устройстве PCI предусмотрено 6 таких регистров, что позволяет каждой плате PCI запрашивать не более 6 диапазонов адресов памяти или устройств ввода-вывода. При старте устройства каждый регистр BAR содержит код, соответствующий размеру запрашиваемой области памяти или регистров ввода-вывода, а BIOS системы записывает в этот регистр базовый адрес блока.

Шина PCI до сих пор встречается в системных платах. Внешний вид разъемов PCI показан на рис. 13.8.

31	16	15	0			
Device ID		Vendor ID				
Status		Command				
Class Code		Revision ID				
BIST	Header Type	Latency Timer	Cache Line Size			
Base Address Registers						
Cardbus CIS Pointer						
Subsystem ID	Subsystem Vendor ID					
Expansion ROM Base Address						
Reserved						
Reserved						
Max_Lat	Min_Grt	Interrupt Pin	Interrupt Line			

Рисунок 13.7. Регистры периферийного устройства, подключаемого к шине PCI

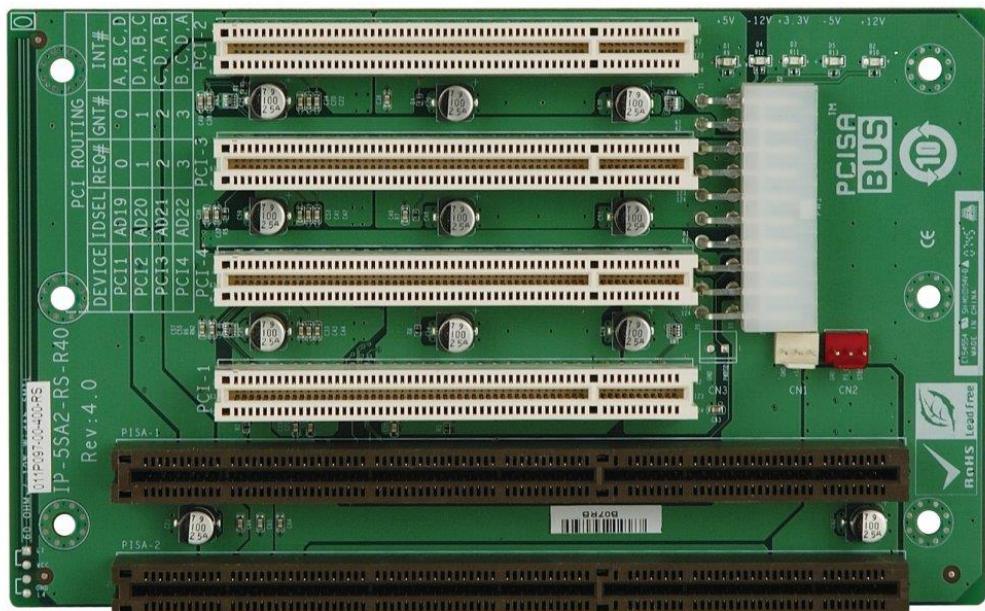


Рисунок 13.8. Внешний вид разъемов системной шины PCI (сверху)

Более современная шина PCI Express логически совпадает с PCI, т.е. имеет ту же структуру регистров. Однако вместо параллельной шины адреса-данных и сигналов управления PCI Express использует высокоскоростную дифференциальную пару (по одной в каждом направлении) для передачи пакета, содержащего адрес, данные и команды. Реализация PCI Express существенно сложнее, чем для обычных цифровых сигналов, поэтому не рассматривается в данном курсе.

На рис. 13.9 показан интерфейс системной шины Wishbone, используемой в ряде микроконтроллеров и систем на кристалле. Эта шина является внутренней, поэтому использует отдельные шины для адреса и данных, причем шины данных на запись и чтение разделены.



Рисунок 13.9. Интерфейс системной шины Wishbone

Как и в PCI, шина Wishbone имеет специальный сигнал, который становится активным в момент начала доступа к внешнему устройству, операция записи сопровождается сигналом we (write enable) и дополнительно сигналом sel (select)

для сопровождения каждого из байтов. Реализован также механизм подтверждения готовности – ведущее устройство сигнализирует о готовности обмена сигналом *stb* (strobe), а ведомое устройство должно ответить сигналом *ack* (acknowledge).

В вычислительной системе можно использовать несколько шин. Часто нет практического смысла подключать медленное устройство (например, UART) к высокопроизводительной системной шине со сложным в реализации интерфейсом. Такая сложная шина может иметь подключенное устройство, которое при обращении к нему формирует сигналы более простой системной шины. Это устройство называется *мостом* (bridge). Пример применения моста показан на рис. 13.10. Системная шина процессора Processor Local Bus (PLB) для процессора IBM Power PC подключает мост PLB-OPB, соединяющий ее с более простой шиной On-Chip Peripheral Bus. К этой шине можно подключать широкий спектр устройств, которые не требуют высокой производительности при передаче данных.

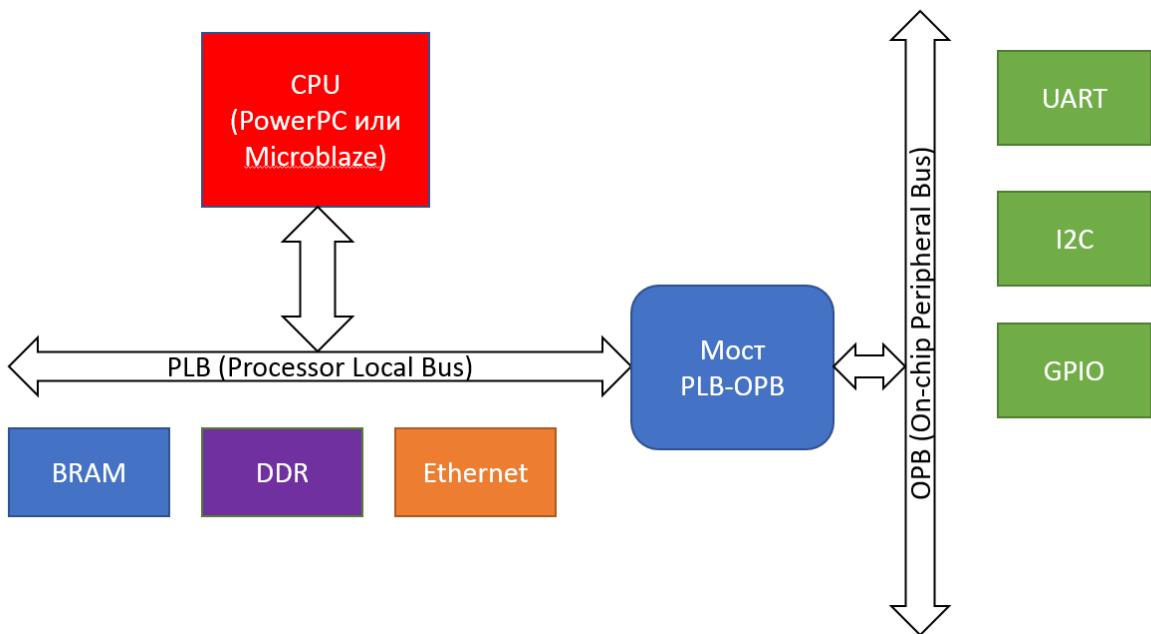


Рисунок 13.10. Иллюстрация к понятию моста (bridge)

13.3. Дополнительные возможности системных шин

В вычислительной системе может возникнуть ситуация, когда инициатором обмена данными является не центральный процессор. Кроме других процессоров системы таким инициатором может быть и периферийное устройство, например, контроллер Ethernet. Можно передать большой блок данных из такого контроллера в память, сделав это быстрее, чем сделал бы процессор

программным путем, в цикле читая и записывая данные. Для этого необходимо временно отключить центральный процессор от шины.

При необходимости управления работой системной шины от нескольких устройств требуется специальное устройство управления, которое и будет определять, какое именно устройство будет подключать свои сигналы к системной шине. В таких системах используется следующая терминология:

- ведущее устройство (*master*) – устройство, которое может инициировать обмен данными по системной шине, формируя сигналы адреса и управления, и принимая или передавая данные;

- ведомое устройство (*slave*) – устройство, которое наблюдает за сигналами адреса и управления, передавая или принимая данные по запросу;

- арбитр (*arbiter*) – устройство, которое определяет, какое именно ведущее устройство является в данный момент активным и может управлять системной шиной.

Пример подключения арбитра и нескольких ведущих устройств показан на рис. 13.11. Предполагается, что контроллер Ethernet используется в высокоскоростном режиме и способен сформировать поток данных с высокой интенсивностью. Поэтому, чтобы не занимать ресурсы процессора, этот контроллер самостоятельно подключается к системной шине и передает данные непосредственно в память, подключенную к ней.

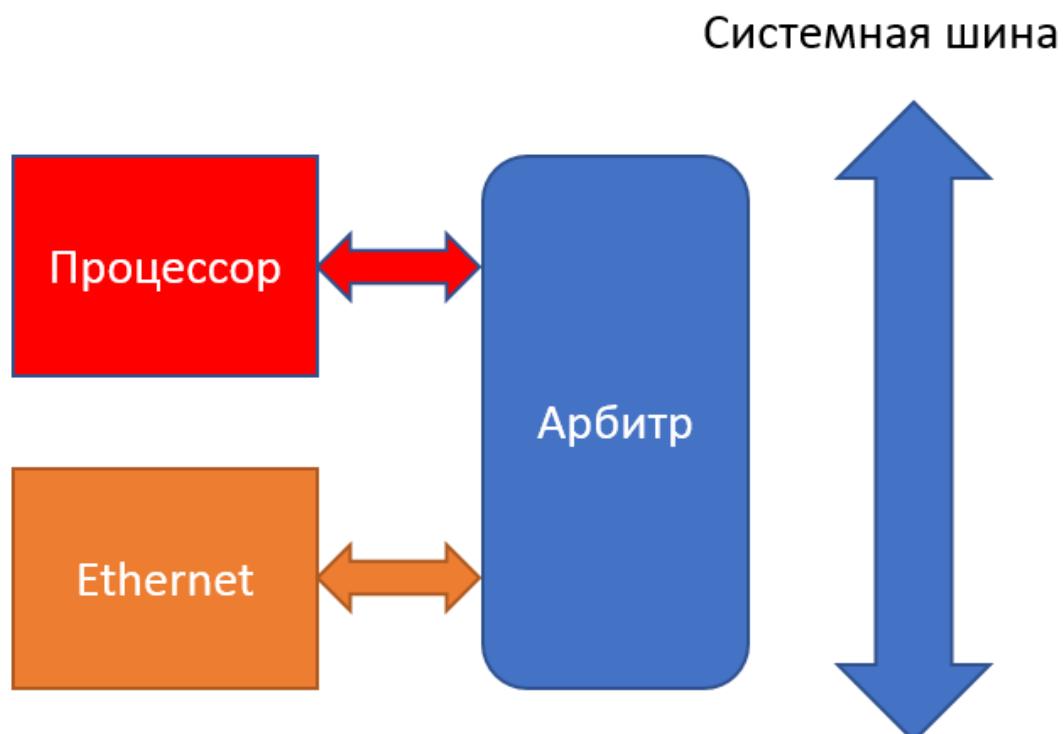


Рисунок 13.11. Иллюстрация к понятию арбитра системной шины

Проектирование арбитра системной шины также имеет свои особенности. Если ведущие устройства используют пакетную передачу данных, необходимо исключить ситуацию, когда цикл обмена будет неожиданно прерван, а управление шиной передано другому ведущему устройству. Такая же ситуация может возникнуть и в других случаях, когда цикл обмена занимает более одного такта.

Самым распространенным способом управления шиной является использование трехступенчатой схемы. При этом используется следующая последовательность:

1. Ведущее устройство, претендующее на работу с шиной, устанавливает сигнал запроса (обычно он обозначается REQ, от слова request).
2. Арбитр, анализируя состояние шины, выдает устройству сигнал подтверждения (GNT, от grant).
3. Ведущее устройство подтверждает захват шины и устанавливает сигнал LOCK (locked).

Пока ведущее устройство удерживает сигнал LOCK, арбитр не предоставляет доступ к шине другим ведущим устройствам. По завершению работы с шиной ведущее устройство снимает сигнал LOCK, и арбитр может предоставить доступ другому ведущему устройству.

Порядок выбора ведущих устройств для доступа к шине является предметом отдельного рассмотрения. Если на шине находятся несколько ведущих устройств, простейшими способами выбора активного устройства являются доступ с фиксированным приоритетом и поочередный доступ.

При доступе с фиксированным приоритетом каждое из ведущих устройств подключается к соответствующему входу арбитра. Арбитр предоставляет доступ «верхнему» из всех устройств, дающих запросы на доступ. Если пронумеровать входы REQ, начиная с 0, то устройство номер 0 будет получать доступ с наивысшим приоритетом, т.е. захватывать шину, если она свободна, а устройство номер 1 – только если устройство номер 0 не претендует в этот момент на захват шины. Такой порядок работы чреват ситуациями, когда ведущие устройства с низким приоритетом будут получать доступ к системной шине слишком редко, что приведет к их простоям и снижению общей производительности. С другой стороны, правильный выбор приоритетов доступа будет способствовать своевременной обработке запросов от ведущих устройств, выполняющих наиболее критичные задачи.

Поочередный доступ к шине предусматривает, что ведущее устройство, получившее доступ, при освобождении шины перемещается в конец списка, и

при очередном запросе наивысший приоритет будет иметь следующее за ним устройство. Такой подход способствует более равномерному распределению времени между претендентами на системную шину.

Возможны также промежуточные варианты – например, в зависимости от критичности выполняемых задач и интенсивности обмена некоторые устройства могут перемещаться не в конец списка, а в его середину, или в фиксированное место (получая таким образом 50% времени доступа). Вопросы проектирования арбитров системной шины нельзя считать до конца изученными и в этой области остается пространство для дальнейших практических работ.

Другим приемом при проектировании системной шины сложного вычислительного устройства является использование коммутатора (crossbar). Он представляет собой мультиплексор, обеспечивающий подключение к каждой из системных шин одного из процессоров. Условное обозначение такого узла на схеме и принцип реализации показаны на рис. 13.12.

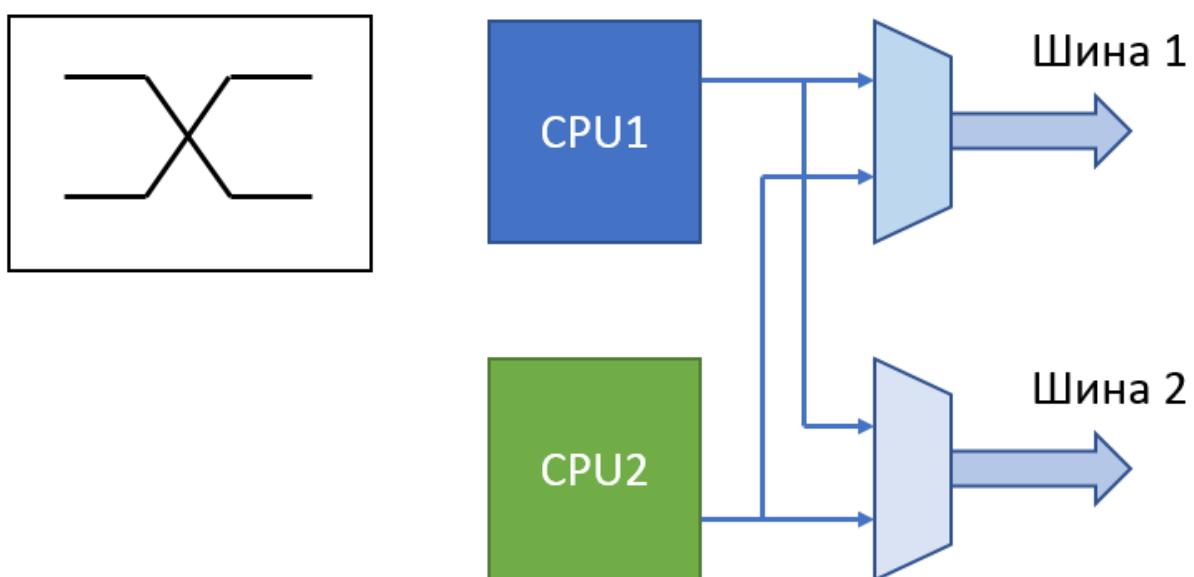


Рисунок 13.12. Многослойная системная шина

Назначение нескольких шин («слоев», layer) состоит в том, чтобы обеспечить независимый доступ отдельных процессоров к периферийным устройствам, если они в один и тот же момент времени обращаются каждый к своему устройству. Если бы системная шина была одна, то любой процессор заблокировал бы работу всех остальных процессоров, даже если им потребовалась бы очень простая операция. Если же в системе есть несколько слоев системной шины, то длительная операция одного процессорного ядра (например, пересылка данных из сетевого контроллера) не помешает второму

ядру выполнять операции с UART, используя другой слой. Коммутатор нужен для того, чтобы периферийные устройства не были жестко привязаны каждое к определенному ядру, а могли при необходимости переключаться между процессорами.

Для каждого из слоев системной шины требуется подключение только одного процессора (хотя и необязательно, чтобы в каждый момент какой-то процессор осуществлял обмен данными по шине). Эта задача также решается арбитром, который в данном варианте несколько усложняется. Увеличение количества процессорных ядер и системных шин ведет к дальнейшему усложнению коммутаторов и схем предоставления доступа, поэтому такой подход к реализации многоядерных систем имеет свои естественные ограничения.

13.4. Выводы по разделу

Системная шина является важным элементом вычислительной системы, обеспечивающим подключение процессора к периферийным устройствам. Важность проектирования системной шины подчеркивается фундаментальной проблемой «стены памяти» / «стены периферии» - отставанием пропускной способности внешних интерфейсов по сравнению с достижимой производительностью вычислений процессорного ядра.

При проектировании шины существенно, является эта шина внешней (подключаемой к внешним выводам микросхемы), или внутренней (соединяющей компоненты системы на кристалле). Как правило, внутри микросхемы можно использовать больше сигналов и более высокую тактовую частоту, поэтому пропускная способность такой шины окажется выше.

Ряд технических приемов организации системной шины требуют отдельного рассмотрения. Например, реализация протокола рукопожатия (handshaking) необходима при явном требовании подключения медленных периферийных устройств, неспособных обеспечить работу на тактовой частоте системной шины. Количество слоев шины, порядок соединения, алгоритмы работы арбитра и другие вопросы являются предметом индивидуального рассмотрения при проектировании конкретной вычислительной системы.

Контрольные вопросы:

1. Какие основные сигналы имеет системная шина?

2. Как можно решить проблему подключения к системной шине периферийного устройства, которое заведомо медленнее процессорного ядра и не успевает подключиться к шине в течение одного периода тактового сигнала?
3. Как организовать использование одно и того же периферийного устройства двумя процессорными ядрами в составе одной микросхемы?
4. Модификации шины PCI предусматривают 32 или 64 разряда данных и тактовую частоту 33 МГц или 66 МГц. Можно ли увеличить пропускную способность PCI, переходя к 128 или 256 разрядам, и повышая тактовую частоту до 100, 150 МГц и выше?

14. СОПРЯЖЕНИЕ ИЗМЕРИТЕЛЬНЫХ И СИЛОВЫХ УСТРОЙСТВ С ЦИФРОВЫМИ СИСТЕМАМИ

14.1. Ввод аналоговых сигналов в компьютерных системах

Большинство физических величин являются аналоговыми, т.е. принимают значения из непрерывного диапазона чисел. Для ввода таких величин в вычислительную систему необходимо преобразовать их в цифровую форму. Микросхемы, предназначенные для этого, называются *аналого-цифровыми преобразователями* (АЦП, также ADC – Analog-to-Digital Converter).

Ввод аналоговых сигналов широко применяется во встраиваемых системах. Многие микроконтроллеры имеют АЦП среди установленных на кристалл периферийных устройств. То, что некоторые датчики предоставляют цифровые интерфейсы, в действительности является следствием того, что аналоговый сигнал, полученный с первичного чувствительного элемента, был подан на АЦП смонтированного в датчике микроконтроллера.

14.2. АЦП, его характеристики

АЦП преобразует входной сигнал из диапазона 0..X_{max} (или X_{min..max}) в N-разрядный цифровой код, где двоичное значение 0 соответствует минимальной величине входного диапазона, а максимальное двоичное значение 2^N-1 – максимальной величине. Таким образом, разрядность цифрового выхода является одним из важнейших параметров АЦП. Часто именно его упоминают в первую очередь.

Часто с высокой разрядностью АЦП связывают его точность. При этом понятие точности определяется следующим образом: **«точность средства измерений — качество средства измерений, отражающее близость к нулю его погрешностей».**

Часто термин «точность» используют в бытовом, разговорном смысле, как синоним небольшой ошибки. С точки зрения формального определения следует рассматривать, насколько цифровой код на выходе АЦП соответствует входному напряжению, измеренному в международной системе единиц СИ в вольтах. В этом кроется фундаментальное препятствие – в действительности, принципом работы АЦП является сравнение входного напряжения с эталоном. В качестве эталона может использоваться внутренний источник напряжения или же внешний источник, напряжение с которого подается на специальный вход АЦП. На сегодняшний день обеспечить погрешность эталонов удается на вполне определенном уровне. Например, с помощью полупроводниковых

стабилитронов можно обеспечить погрешность, соответствующую работе 12-разрядного АЦП. Напряжение такого эталона существенно зависит от температуры и имеет дрейф с течением времени.

Следующим существенным шагом в повышении стабильности эталона напряжения может быть использование «нормального элемента Вестона», представляющего собой химическую батарею. Его относительная погрешность обычно указывается от $2 \cdot 10^{-6}$ до $40 \cdot 10^{-6}$, в зависимости от условий измерения, но сложность конструкции существенно затрудняет практическое применение.

Наименьшей погрешности на сегодняшний день можно добиться с помощью квантового эффекта Джозефсона. Он наблюдается в сверхпроводящем состоянии проводника, поэтому конструкция эталона на таком эффекте чрезмерно сложна для широкого применения.

Наличие эталонного напряжения является определяющим фактором при обеспечении низкой погрешности, поэтому при проектировании крайне важно различать, о чем идет речь – о различии одного уровня сигнала относительно другого (это обеспечивается как раз повышением разрядности АЦП), или о сопоставлении цифрового кода с выхода АЦП аналоговому сигналу в системе СИ (для этого нужен эталон напряжения с соответствующей малой погрешностью).

Кроме того, что цифровой код измеряет входной сигнал «в единицах эталона, подключенного к АЦП», сама микросхема может вносить в измерения дополнительные погрешности. В идеальном случае выходной код АЦП будет представлять собой серию последовательных цифровых значений, где переход между ними будет происходить через строго одинаковые интервалы входного аналогового сигнала.

На рис. 14.1 показан график, иллюстрирующий основные виды погрешностей АЦП. Идеальной характеристикой была бы прямая с наклоном, равным 1, однако дискретный характер выходных значений АЦП превращает такую прямую в серию ступенек, показанную как «идеально-реальная характеристика».

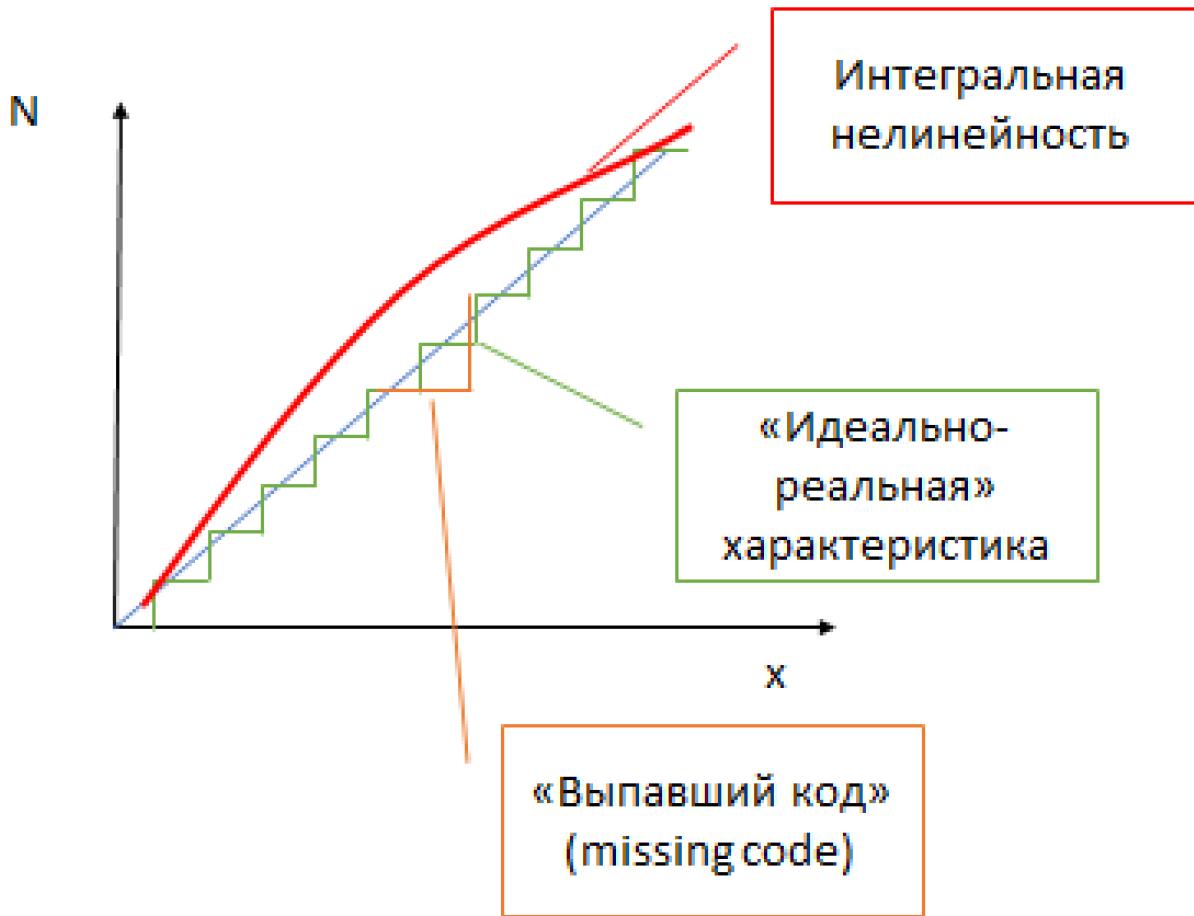


Рисунок 14.1. Преобразование данных в аналого-цифровом преобразователе

В реальных микросхемах существуют отклонения даже от такой характеристики, учитывающей дискретный характер выхода АЦП. Отклонения в целом подразделяются на дифференциальные (т.е. локализованные) и интегральные. Например, если при изменении входного сигнала цифровой код не изменяется, а потом изменяется резким скачком (так что какое-то двоичное значение никогда не появляется на выходе), такую погрешность относят к дифференциальным. Конкретно этот вид погрешности называется «выпавшим кодом» (missing code) и в ряде случаев производители АЦП при контроле качества относят такую микросхему к браку. В технической документации можно встретить термин «no missing codes», который обозначает гарантию производителя на отсутствие подобных погрешностей.

Интегральная нелинейность проиллюстрирована на рис. 14.1 плавной кривой. В действительности ее вид может отличаться, однако она отражает суть интегральных нелинейностей – при отсутствии локализованных погрешностей и в целом равномерном изменении выходного кода АЦП сами коды располагаются

не вдоль прямой линии, а вдоль слегка искаженной кривой, причем на каких-то участках погрешность может быть существенной.

В документации на АЦП указывается максимальная величина погрешности, обусловленная дифференциальной составляющей (DNL, Differential Non-Linearity) и интегральной составляющей (INL, Integral Non-Linearity).

Основным выводом из приведенных сведений является то, что повышать разрядность АЦП есть смысл только до определенного предела, за которым погрешности эталона и нелинейности микросхемы сведут усилия на нет.

Важным замечанием к разрядности АЦП является также динамический диапазон входного сигнала. Он никак не зависит от конструкции АЦП и определяется исключительно внешним объектом. Например, если уровень сигнала изменяется в 10 раз, то выбрав 8-разрядный АЦП и установив цифровой код 255 для максимального уровня, для минимального уровня он окажется равен всего 25. Следовательно, относительная ошибка для минимального уровня окажется существенно выше, и приняв погрешность равной 1 разряду АЦП, можно убедиться, что для максимального уровня она равна 0.4%, а для минимального – уже 4%, что может оказаться неприемлемым. Поэтому разрядность АЦП следует выбирать так, чтобы даже при небольших значениях входного аналогового сигнала его цифровое представление обеспечивало достаточную для задачи разрешающую способность.

Понятие «достаточную для задачи» не может быть строго формализовано. Не существует универсальных способов расчета требуемой разрядности АЦП, одинаково применимых к любой заранее неизвестной задаче. Определение разрядности АЦП требует предварительного моделирования системы и выяснения, какие погрешности вносятся из-за неточного воспроизведения аналоговых значений в цифровой форме и как это влияет на итоговые характеристики разрабатываемого устройства.

Вторым важным параметром АЦП является время преобразования аналогового сигнала в цифровой код. Вместо времени часто удобно указывать максимальную частоту, с которой АЦП способен обновлять цифровой код. Частота преобразования и разрядность обычно связаны, причем чем больше разрядность АЦП, тем сложнее обеспечить высокую частоту преобразования.

На рис. 14.2 показан фрагмент сайта одного из производителей АЦП, компании Analog Devices. На сайте видна классификация АЦП по сочетанию разрядности (Resolution) и частоте преобразования (SPS, Sample Per Second).

The screenshot shows a table comparing ADC throughput rates. The left side is titled 'Precision and General Purpose ADC Finder' and the right side is titled 'High-Speed ADC Finder'. Both sides have a header 'Resolution (Bits)' and a sub-header 'ADC Throughput Rate (SPS)'. The left side has columns for <1K, 1 - 100k, 100 - 250k, 250 - 500k, 500k - 1M, and 1M - 20M. The right side has columns for 10 - 50M, 50 - 100M, 100 - 250M, 250M - 1G, and >1G. The table is divided into four quadrants by resolution (21-32, 17-20, 14-16, 8-13) and speed (>/=16, 14-15, 12-13, </=11).

Resolution (Bits)	ADC Throughput Rate (SPS)						Resolution (Bits)	ADC Throughput Rate (SPS)				
	<1K	1 - 100k	100 - 250k	250 - 500k	500k - 1M	1M - 20M		10 - 50M	50 - 100M	100 - 250M	250M - 1G	>1G
21 - 32	✓	✓	✓	✓	✓	✓	>/=16	✓	✓	✓	✓	✓
17 - 20	✓	✓	✓	✓	✓	✓	14 - 15	✓	✓	✓	✓	✓
14 - 16	✓	✓	✓	✓	✓	✓	12 - 13	✓	✓	✓	✓	✓
8 - 13	✓	✓	✓	✓	✓	✓	</=11	✓	✓	✓	✓	✓

Рисунок 14.2. Сочетание разрядности и частоты преобразования современных АЦП (на примере сайта компании Analog Devices)

Номенклатура современных АЦП довольно широка. Например, на рис. 14.3 показана классификация, предлагаемая компанией Maxim Integrated в разделе сайта, относящегося к АЦП. На рис. 14.4 для сравнения показана аналогичная информация с сайта компании Texas Instruments.

The screenshot shows a grid of categories for ADCs. At the top, there are three main categories: 'Precision ADC Selector Guide', 'All Precision ADCs', and 'High-Speed ADCs'. Below these are six more specific categories: 'Successive Approximation ADCs', 'Low Power Precision ADCs', 'Integrated Precision ADCs', 'Simultaneous Sampling ADCs', 'Extended Range Input ADCs', and 'Sigma-Delta ADCs'. Each category has a 'Find Products' button.

Рисунок 14.3. Некоторые классы современных АЦП (на примере сайта компании Maxim Integrated)

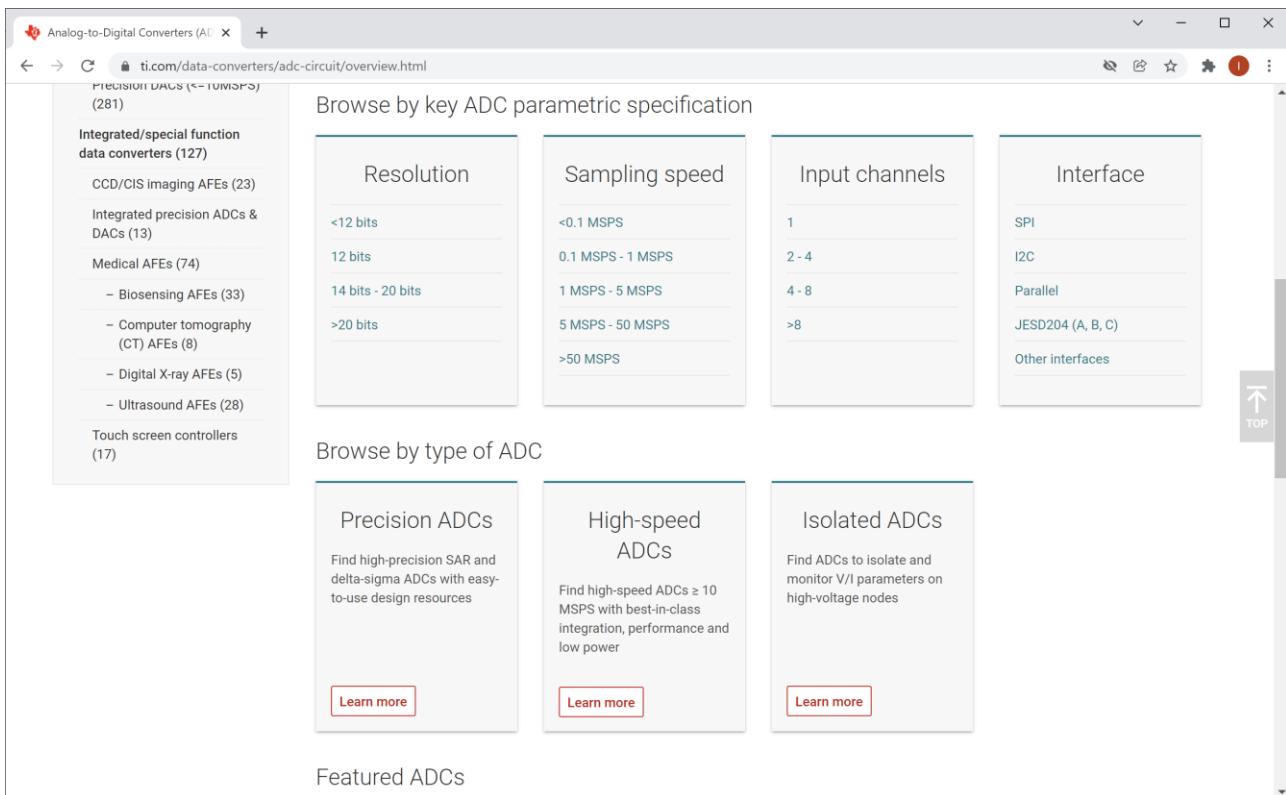


Рисунок 14.4. Некоторые классы современных АЦП (на примере сайта компании Texas Instruments)

14.3. Архитектуры и интерфейсы АЦП. Сопряжение АЦП с цифровыми системами

Характеристики АЦП во многом определяются их внутренней архитектурой. Самой простой архитектурой обладают «АЦП прямого преобразования». Они содержат внутри набор аналоговых компараторов, сравнивающих входной сигнал с пороговыми значениями. Для достижения высокой разрядности требуется большой набор компараторов, поэтому такая архитектура имеет небольшую разрядность (8, реже 10), однако обеспечивает высокое быстродействие.

В среднем диапазоне разрядности находится «АЦП последовательного приближения» (SAR, «successive-approximation ADC»). Принцип действия таких АЦП основан на алгоритме последовательного деления пополам (дихотомии). Например, сравнив входное напряжение с половиной от максимального, можно определить старший бит выходного цифрового кода. Далее необходимо сравнить, в зависимости от результата, с 0,25 или 0,75 от максимального и т.д. Этот процесс несколько медленнее по сравнению с прямым преобразованием, где сравнение выполняется однократно и параллельно, однако требует меньше ресурсов схемы, а повышение разрядности обеспечивается увеличением шагов алгоритма, поэтому имеет определенный резерв для наращивания. АЦП

последовательного приближения распространены достаточно широко и типично имеют 12-14-16 разрядов.

Относительно высокая частота преобразования обуславливает применение для таких АЦП параллельного интерфейса. Он может быть как синхронным, так и асинхронным и определяется компанией-производителем (единого стандарта на параллельный интерфейс АЦП не существует). На рис. 14.5 показан фрагмент документации, в котором видна структурная схема и внешние сигналы АЦП последовательного приближения TLC1550.

**TLC1550I, TLC1550M, TLC1551I
10-BIT ANALOG-TO-DIGITAL CONVERTERS
WITH PARALLEL OUTPUTS**
SLAS043G – MAY 1991 – REVISED NOVEMBER 2003

functional block diagram

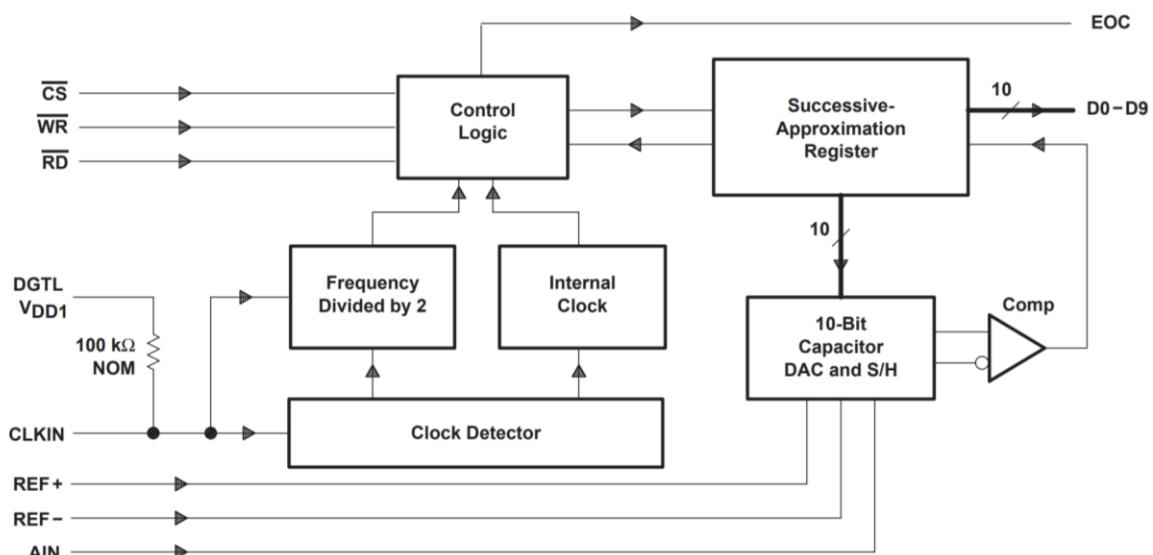


Рисунок 14.5. Структурная схема АЦП с параллельным интерфейсом

На рис. 14.5 можно видеть, что преобразование АЦП производится на основе входного тактового сигнала, который обновляет выходы D0-D9. Сигнал ЕОС часто применяется в АЦП и обозначает окончание преобразования (End Of Conversion).

Если от АЦП не требуется высокая частота преобразования (это позволяет, в частности, снизить стоимость такой микросхемы), вместо параллельного интерфейса может быть применен последовательный. При этом АЦП может иметь как непосредственно интерфейс SPI, так и схожий с ним по принципу работы, однако не имеющий полного списка сигналов SPI.

На рис. 14.6 показана структурная схема и расположение выводов корпуса АЦП последовательного приближения с последовательным интерфейсом. Видно, что для такого АЦП достаточно 8-выводного корпуса.

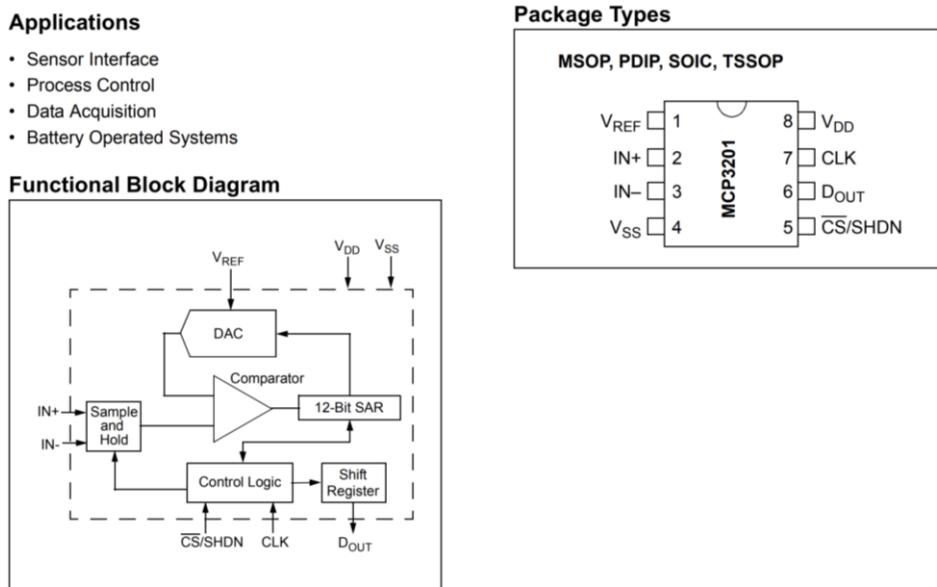


Рисунок 14.6. Структурная схема и расположение выводов корпуса АЦП с последовательным интерфейсом

На рис. 14.7 показаны временные диаграммы АЦП с последовательным интерфейсом. Это не относится к универсальному стандарту, поскольку производители могут вносить в интерфейс собственные изменения.

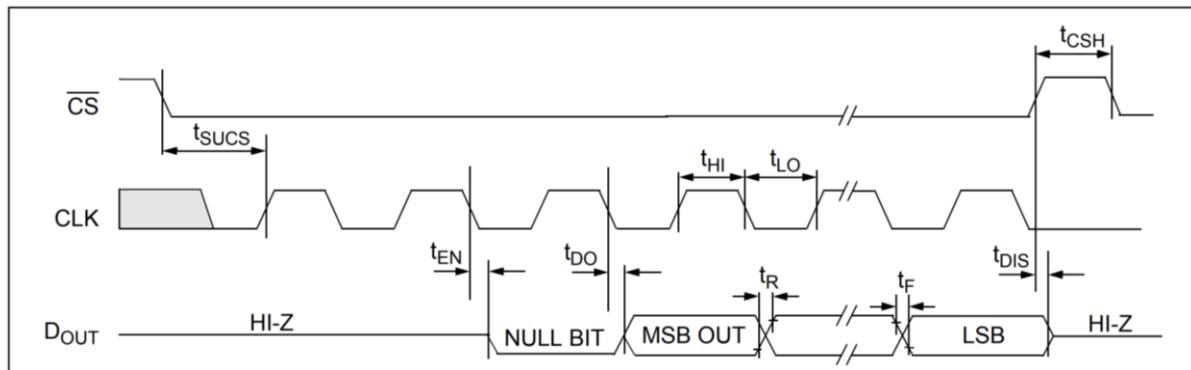


FIGURE 1-1: Serial Timing.

Рисунок 14.7. Временные диаграммы передачи данных из АЦП с последовательным интерфейсом

Для показанного примера видно, что после перехода сигнала CS в активное состояние (логический 0) через несколько тактов АЦП переводит выход из состояния высокого импеданса (High-Z) в состояние логического 0, сигнализируя таким образом о готовности цифрового кода. Далее по каждому фронту тактового сигнала АЦП передает очередной бит, начиная со старшего. После приема последнего бита внешняя микросхема должна перевести CS в неактивное состояние.

Можно еще раз отметить, что показанная временная диаграмма не является универсальной. Подключение конкретных АЦП должно начинаться с изучения технической документации и определения сигналов, требующих управления.

В некоторых семействах ПЛИС имеются встроенные в кристалл АЦП. На рис. 14.8 показаны АЦП в ПЛИС серии 7 компании Xilinx (AMD). Они имеют разрядность 12 бит и невысокую частоту преобразования 1 МГц. Такие АЦП используются в основном для мониторинга состояния ПЛИС и контроля таких внешних параметров, как напряжение питания и температура.

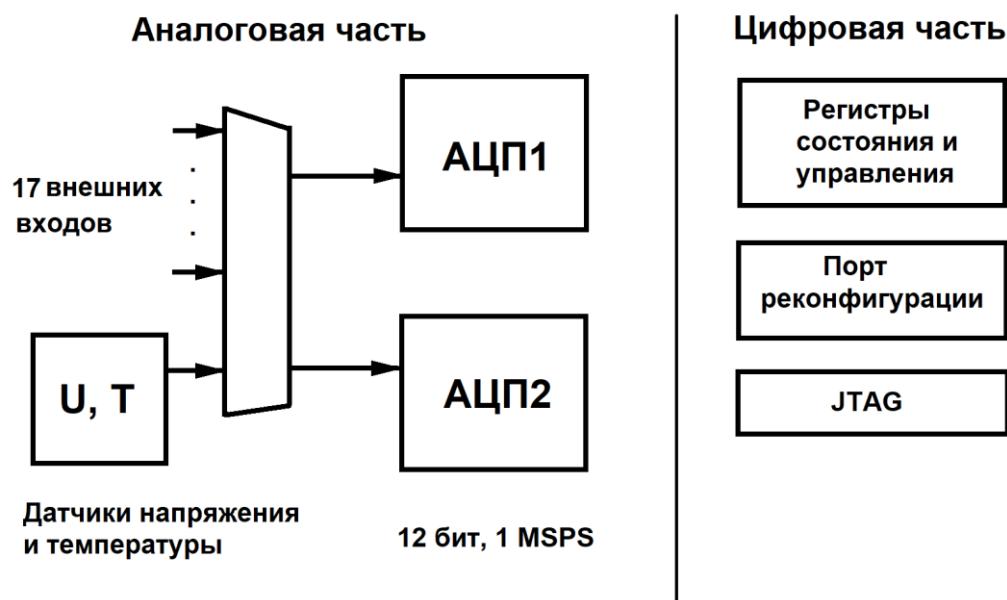


Рисунок 14.8. Встроенные модули АЦП в некоторых семействах современных ПЛИС

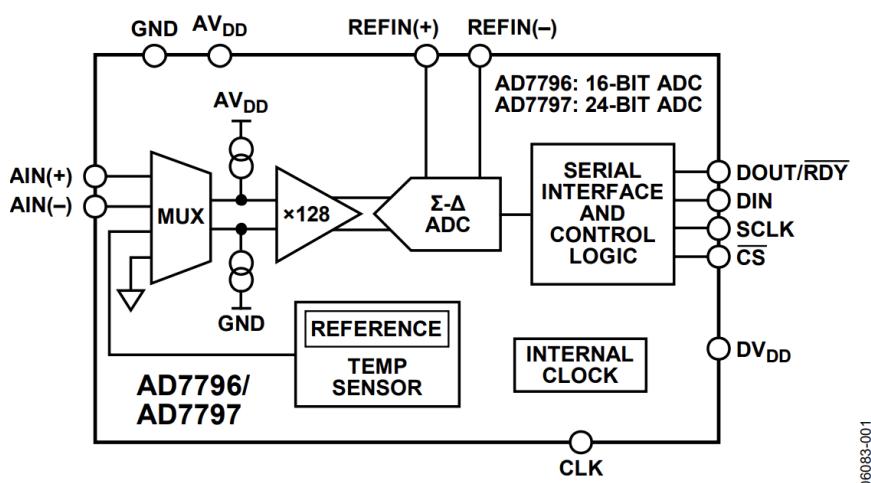
Наиболее высокую точность, но низкую частоту преобразования, обеспечивают АЦП с архитектурой «сигма-дельта». Принцип их действия основан на постепенном добавлении к входному сигналу модулирующего напряжения (отклонение обычно обозначается символом «дельта»). Смысл такого действия становится понятен, если учесть, что при получении цифрового кода в принципе нельзя понять, как близко входное напряжение находится относительно следующего порога – например, если код равен 10, то он может быть преобразован из аналогового уровня 10,1, 10,2, 10,3 и т.д. Если входное напряжение находилось рядом со следующим порогом напряжения, переход к нему произойдет быстро. Например, если добавлять 256 шагов модулирующего напряжения, можно понять, на каком из них состоится переход к следующему цифровому коду и добавить к цифровому коду еще 8 двоичных разрядов, хранящих номер шага, при котором произошел переход. Такой процесс

добавления модулирующего напряжения можно повторять многократно, накапливая дополнительные разряды. Операция сложения обозначается символом «сигма». Из этих символов, обозначающих основные операции, образуется название архитектуры АЦП – «сигма-дельта».

Как правило, АЦП с такой архитектурой обеспечивают 16 и более разрядов. Встречаются микросхемы с разрядностью 24 бита, однако для них специально отмечается необходимость подключения источника опорного (эталонного) напряжения с соответствующей небольшой погрешностью. При этом время измерений может достигать десятых долей секунды.

Структурная схема и сигналы сигма-дельта АЦП показаны на рис. 14.9.

FUNCTIONAL BLOCK DIAGRAM



06083-001

Рисунок 14.9. Структурная схема прецизионного АЦП с архитектурой сигма-дельта

Из-за крайне медленного процесса преобразования для сигма-дельта АЦП обычно достаточно последовательного интерфейса.

Для аналого-цифрового преобразования крайне важно обеспечить достаточно высокую частоту формирования цифрового кода. Этот пункт является одним из основных в теории цифровой обработки сигналов. На рис. 14.10 показана иллюстрация к проблеме наложения частот, которое в ряде случаев упоминается в виде прямой транслитерации англоязычного термина «aliasing».

Суть проблемы состоит в том, что если частота отсчетов АЦП меньше, чем частота самого сигнала, цифровые отсчеты будут соответствовать моментам времени, принадлежащим разным периодам исходного аналогового сигнала. Попытка соединить эти точки приведет к получению сигнала совершенно другой частоты, существенно ниже той, которая подается в действительности.

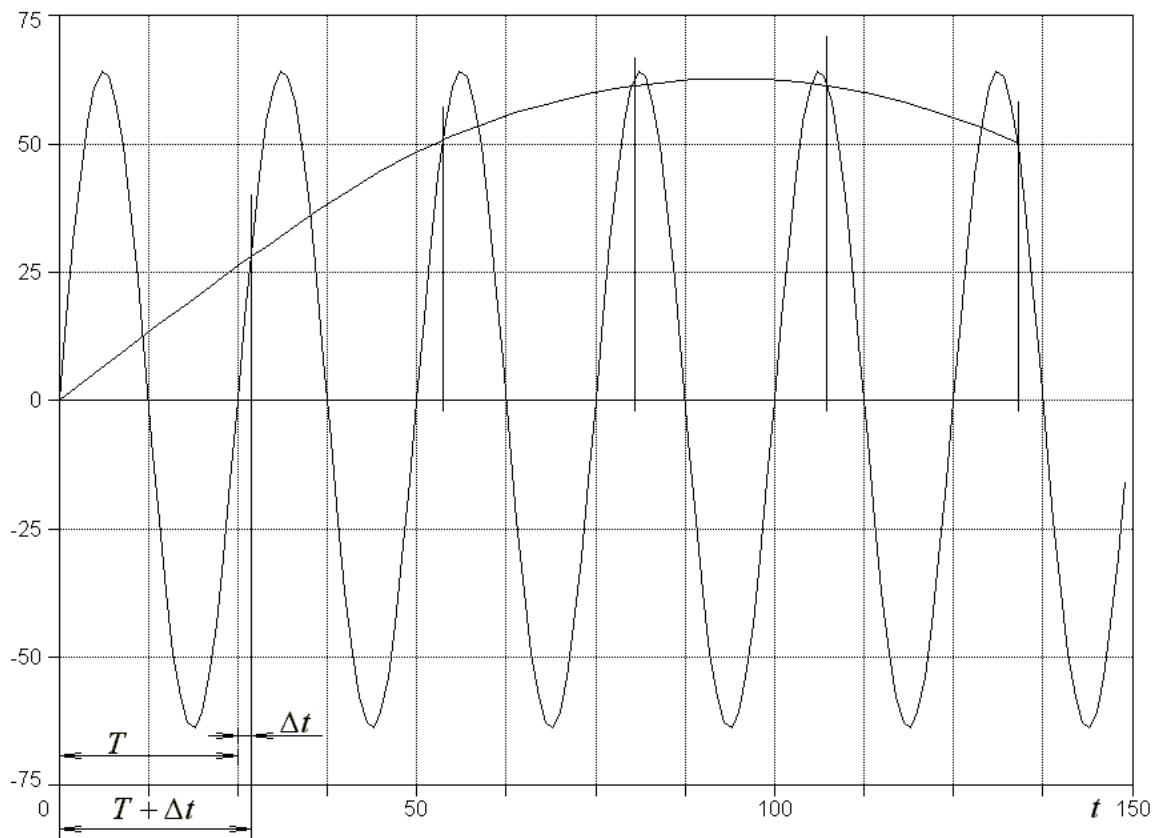


Рисунок 14.10. Иллюстрация к проблеме наложения («алиасинг», aliasing)

Фундаментальным правилом в цифровой обработке сигналов является теорема Котельникова (упоминаемая также как теорема Найквиста-Шеннона). Она утверждает, что на каждый период исходного аналогового сигнала необходимо не менее двух цифровых отсчетов для его правильного восстановления.

Это правило показывает, что повышение частоты АЦП является одним из приоритетных направлений в цифровой обработке сигналов, которое влечет за собой и увеличение производительности вычислительных устройств, которые должны обрабатывать такой поток цифровых отсчетов.

Устройство для цифровой обработки сигналов может быть полезной составной частью вычислительной системы. Для моделирования цифровой обработки сигналов часто используют универсальные процессоры, однако на практике их работа будет сведена к постоянному повторению несложных действий. Если одновременно с цифровой обработкой непрерывного потока цифровых отсчетов процессор выполняет и другие операции, его отвлечение на посторонние действия приведет к тому, что часть цифровых отсчетов будет пропущена, и выходной сигнал окажется искаженным. Это не представляет существенной проблемы при исследовании алгоритмов или обработке заранее

записанного сигнала, однако на практике при работе с радиосигналами, видеопотоком, потоками данных с высокоскоростных датчиков результат обработки может получить заметные искажения, если процессор периодически переходил от обработки сигнала к работе с другими подпрограммами.

Таким образом, сочетание универсального процессора и аппаратной подсистемы цифровой обработки сигналов может дать в результате систему, обеспечивающую лучшие метрологические характеристики и не подверженную эффекту внезапных искажений при отвлечении процессора от обработки сигнала (например, при активных действиях оператора системы). Пример архитектуры системы, содержащей управляющий процессор и аппаратный ускоритель цифровой обработки сигналов, показан на рис. 4.11.

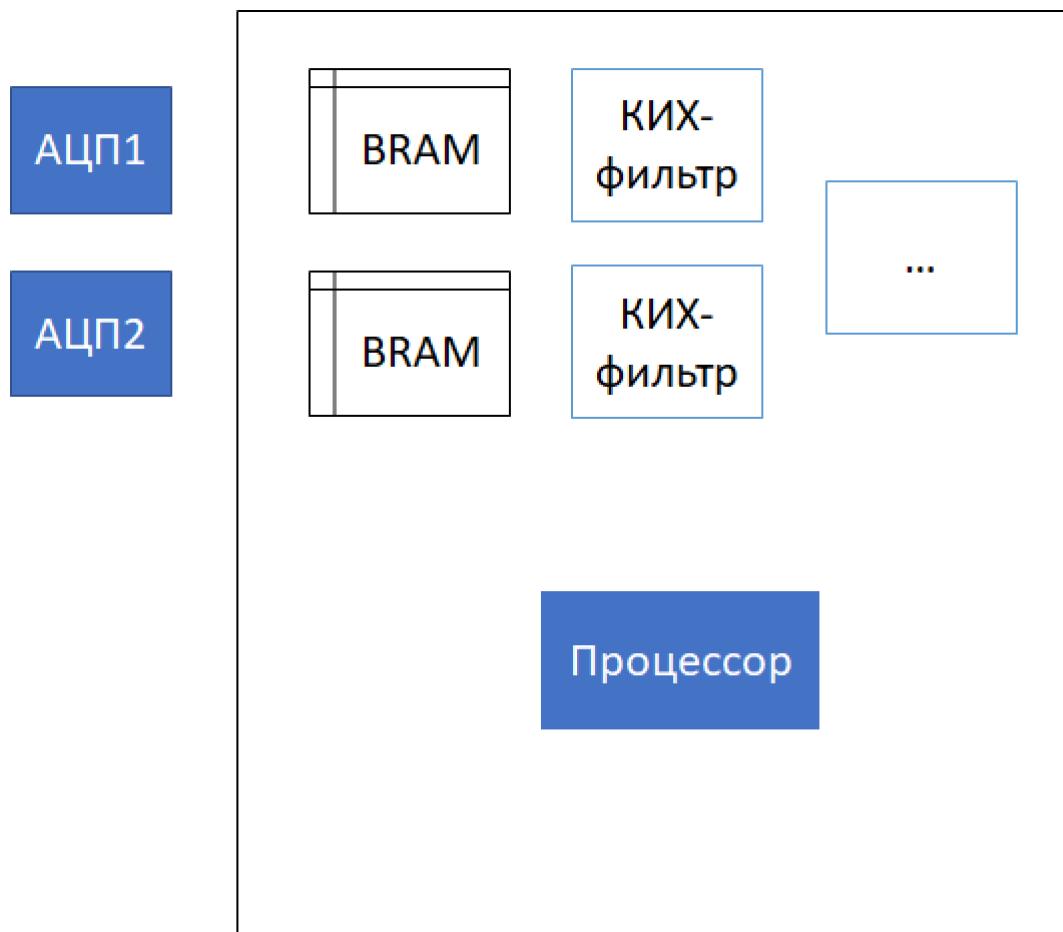


Рисунок 14.11. Архитектура вычислительной системы с применением управляющего процессора и аппаратного ускорителя цифровой обработки сигналов

В настоящее время вопросы совместной работы универсальных процессоров и ускорителей цифровой обработки сигналов еще не получили окончательного решения. Исследования и практические проекты в этой сфере

являются перспективными и представляют существенный интерес во многих областях техники, таких как измерительная техника, управление электромоторами, медицинские системы, беспроводная связь, обработка видео, нейросетевые ускорители и др.

14.4. ЦАП. Интерфейсы ЦАП. Сопряжение ЦАП с цифровыми системами

Цифро-аналоговый преобразователь (ЦАП, также Digital-to-Analog Converter, DAC) преобразует цифровой код в пропорциональное ему аналоговое напряжение. Как и для АЦП, для этого требуется эталонный источник напряжения, относительно которого формируется выходное напряжение. Структурная схема одной из микросхем ЦАП показана на рис. 14.12.

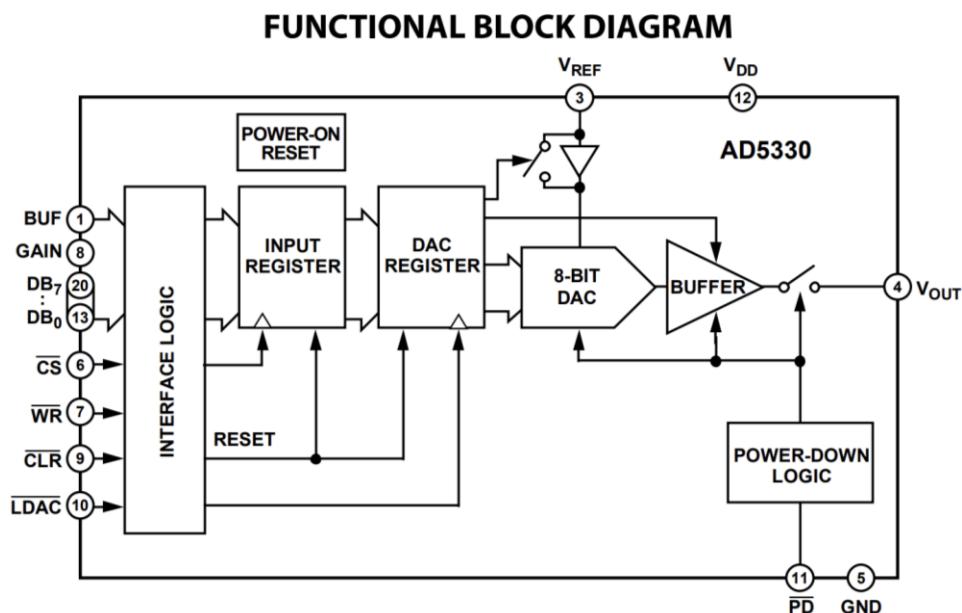


Рисунок 14.12. Структурная схема ЦАП с параллельным интерфейсом

В целом, ЦАП имеют основные характеристики, схожие с АЦП. Основным параметром, определяющим погрешность формирования напряжения, является разрядность. Способность ЦАП быстро изменять выходное напряжение определяется временем цифро-аналогового преобразования, из которого может быть легко получена максимальная частота обновления цифрового кода.

Особенностью ЦАП является применение в них высокоточных резисторов, с помощью которых формируются доли эталонного напряжения. Эти резисторы обычно выполняются на базе угольного композита, который требует специальных технологических операций для реализации на полупроводниковом кристалле. Кроме этого, точное значение резистора обеспечивается путем его

лазерной подгонки («тримминга»), что имеет определенный риск брака. Поэтому размещение ЦАП внутри цифровой микросхемы сопряжено с повышением вероятности брака, производится весьма редко, и характерно для отдельных серий микроконтроллеров.

Пример временных диаграмм записи данных в ЦАП показан на рис. 14.13.

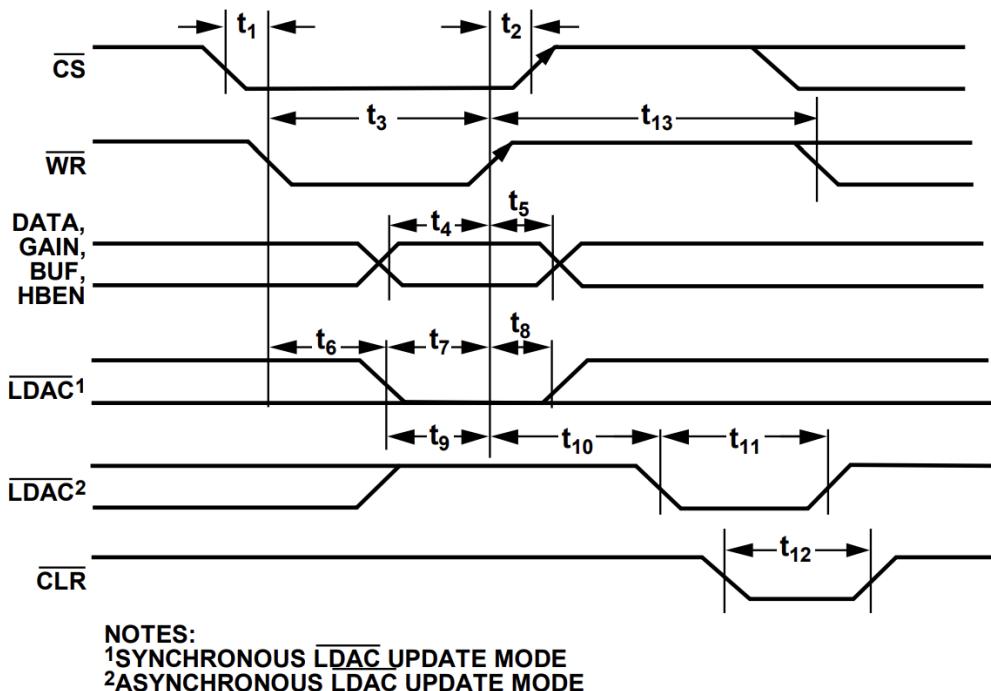


Figure 2. Parallel Interface Timing Diagram

Рисунок 14.13. Временные диаграммы записи данных в ЦАП с параллельным интерфейсом

В показанном примере микросхема ЦАП реализует интерфейс на базе сигналов CS и WR, где тактовый сигнал как таковой отсутствует. Так же, как и для АЦП, это не является общепринятым стандартом, и в микросхемах может быть использован как синхронный параллельный интерфейс, так и последовательный – SPI, I2C или модификация SPI без сигнала данных, возвращаемого микросхемой.

14.5. Управление силовыми устройствами с помощью ШИМ

Сочетание АЦП и ЦАП (или ШИМ) дает возможность построить функционально завершенную схему автоматизированного регулятора. Подобные устройства являются предметом исследования теории систем автоматизированного управления. Они достаточно широко распространены в технике – например, адаптер для заряда смартфона представляет собой

регулятор, поддерживающий на выходе напряжение 5 В независимо от протекающего тока. Это невозможно при использовании только пассивных компонентов – например, включение в цепь заряда последовательного резистора для снижения напряжения на нагрузке приведет к тому, что падение напряжения на этом резисторе будет прямо пропорционально протекающему через него току согласно закону Ома. Поэтому зарядное устройство содержит регулятор, сравнивающий выходное напряжение с эталонным значением 5 В (оно обеспечивается внутренним источником опорного напряжения) и изменяющее сопротивление регулирующего элемента в зависимости от этого значения.

Чтобы избежать непродуктивного рассеивания излишней энергии на регуляторе, часто используют управление с помощью широтно-импульсной модуляции. При этом регулируемой величиной может быть не только напряжение, но и ток, температура, скорость вращения электродвигателя и т.д. Структурная схема системы автоматизированного управления показана на рис. 14.14.

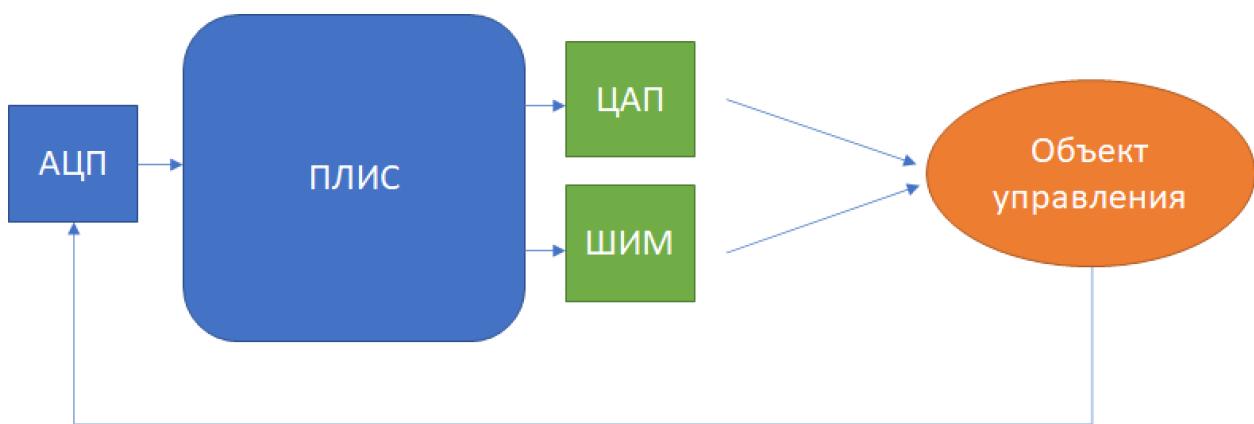


Рисунок 14.14. Структурная схема системы автоматизированного управления (стабилизации выбранного параметра объекта управления)

В такой системе параметр, подлежащий регулированию, измеряется датчиком (чувствительным элементом, ЧЭ) и преобразуется в цифровой код с помощью АЦП. Цифровое вычислительное устройство определяет, какое воздействие необходимо подать на исполнительный элемент – нагреватель, электродвигатель, переключающий транзистор и т.д. Алгоритмы управления достаточно разнообразны, и, как упоминалось выше, являются предметом изучения теории систем автоматизированного управления.

14.6. Выводы по разделу

Для преобразования сигналов между аналоговым и цифровым представлением используются специальные микросхемы – аналого-цифровые (АЦП) и цифро-аналоговые (ЦАП) преобразователи. Они широко используются во встраиваемых устройствах, управляющих бытовыми приборами, устройствами связи, промышленными установками и т.д.

Основными характеристиками АЦП и ЦАП являются разрядность цифрового сигнала и максимальная частота преобразования.

Интерфейсы АЦП и ЦАП не имеют общепринятого стандарта и реализуются производителями микросхем в зависимости от частоты преобразования и назначения микросхемы.

АЦП являются важным элементом систем цифровой обработки сигналов. Теория цифровой обработки сигналов является отдельным большим направлением прикладной науки, а системы цифровой обработки сигналов представляют в настоящее время большой интерес в связи с высокими требованиями к производительности. Представляется перспективной разработка специализированных вычислительных систем, обеспечивающих высокую производительность при обработке потока данных от АЦП.

Дискретное представление цифровых сигналов неминуемо вносит погрешность в измеряемые величины, которые обычно являются аналоговыми. Разрядность АЦП и необходимая частота преобразования требуют отдельного исследования с применением средств математического моделирования.

Контрольные вопросы:

1. Какими параметрами можно описать аналого-цифровой преобразователь?
2. Почему некорректно рассуждение: «АЦП имеет 10 разрядов, т.е. способно представить 1024 выходных значения, поэтому его точность – 0,1%»?
3. Какие интерфейсы применяются для подключения АЦП?
4. Что утверждает теорема Котельникова? Является ли это условие достаточным для анализа сигнала любой формы?
5. Какими параметрами можно описать цифро-аналоговый преобразователь?

15. СТРАТЕГИИ МОДЕЛИРОВАНИЯ И ВЕРИФИКАЦИИ КОМПЬЮТЕРНЫХ СИСТЕМ

15.1. Поведенческое и физическое моделирование, их отличия и место в маршруте проектирования

Моделирование цифровых систем является важным шагом в маршруте их разработки. Возрастание сложности проектируемых устройств заставляет разработчиков тратить все больше времени на их моделирование. Целями моделирования могут быть как исследование алгоритмов работы проектируемого устройства, так и верификация характеристик, получаемых при его аппаратной реализации. В первом случае производится моделирование на верхних уровнях абстрагирования (т.е. преимущественно на поведенческом, и, возможно, RTL), а моделирование на физическом уровне призвано проверить возможность работы созданного устройства в заданных условиях эксплуатации (т.е. проверяется возможность работы на заданной тактовой частоте, с требуемыми длительностями сигналов, в заданном температурном диапазоне и т.д.).

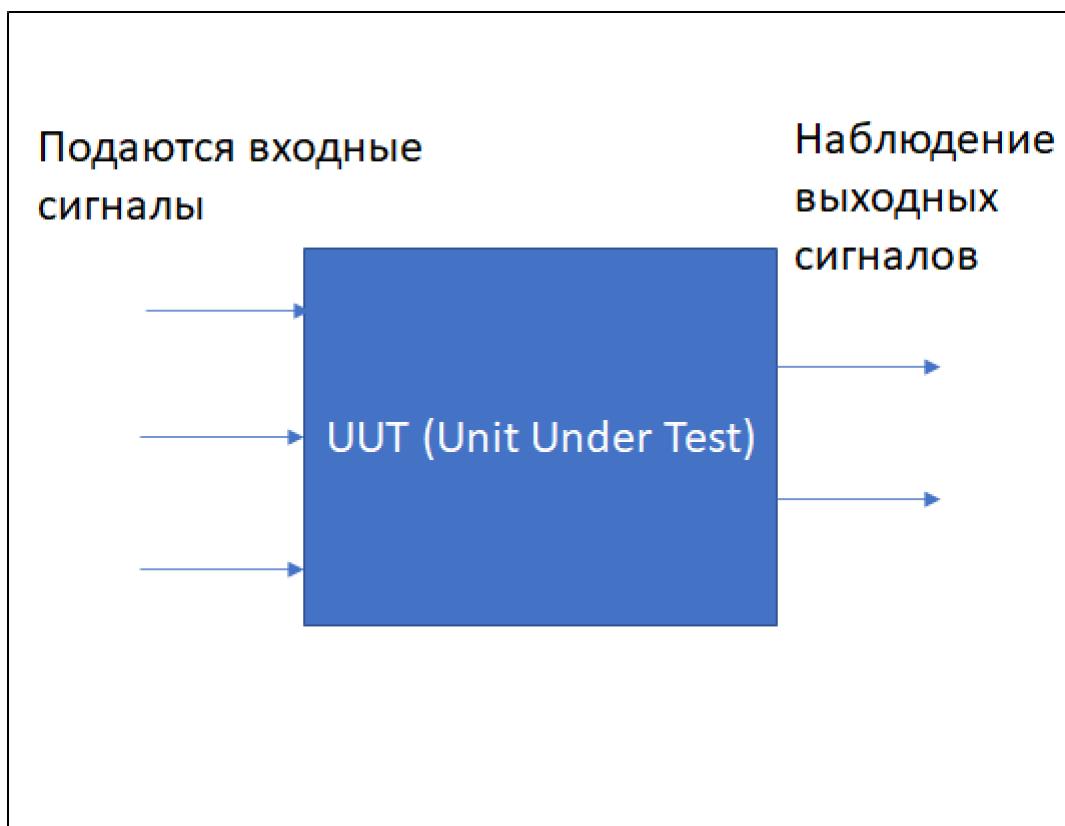


Рисунок 15.1. Иллюстрация к организации процесса моделирования

При моделировании используется подход, основанный на «испытательном стенде» (*testbench*). Моделируемое устройство (в англоязычной литературе *UUT*, *Unit Under Test*) представляется своим синтезируемым кодом, а для проверки его поведения в различных условиях создаются описания тестовых воздействий («моделируемый код»).

На рис. 15.1 показана общая иллюстрация к организации процесса моделирования.

Задание тестовых входных воздействий может быть выполнено на Verilog с помощью модуля типа *Verilog Test Fixture*. Для этого в диалоговом окне добавления компонента к проекту (Add Sources) выбирается пункт Add or Create Simulation Source. Необходимо выбрать тип файла Verilog и задать его имя.

Для унификации обозначений обычно применяется сочетание символов *tb* (от *testbench*, «испытательный стенд»). Так, если модуль имеет название *my_and* (двухходовой элемент И), то с ним удобно сопоставить тестовый файл *my_and_tb*.

Пример шаблона файла с описанием тестовых воздействий показан ниже.

```
`timescale 1ns / 1ps

module my_and_tb;
    // Inputs
    reg a;
    reg b;

    // Outputs
    wire c;

    // Instantiate the Unit Under Test (UUT)
    my_and uut (
        .a(a),
        .b(b),
        .c(c)
    );

    initial begin
        // Initialize Inputs
```

```

a = 0;
b = 0;

// Wait 100 ns for global reset

#100;

// Add stimulus here

end

endmodule

```

Это описание необходимо модифицировать для получения требуемого эффекта от моделирования. Предположим, что требуется проверить поведение модуля при последовательной установке в логическую единицу сначала входа a, а затем входа b. Тогда раздел «инициализация входов» (Initialize Inputs) будет выглядеть так.

```

initial begin
    // Initialize Inputs
    a = 0; # 10; a = 1;
    b = 0; # 20; b = 1;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

end

```

Запуск моделирования в Vivado производится в панели управления, как показано на рис. 15.2.

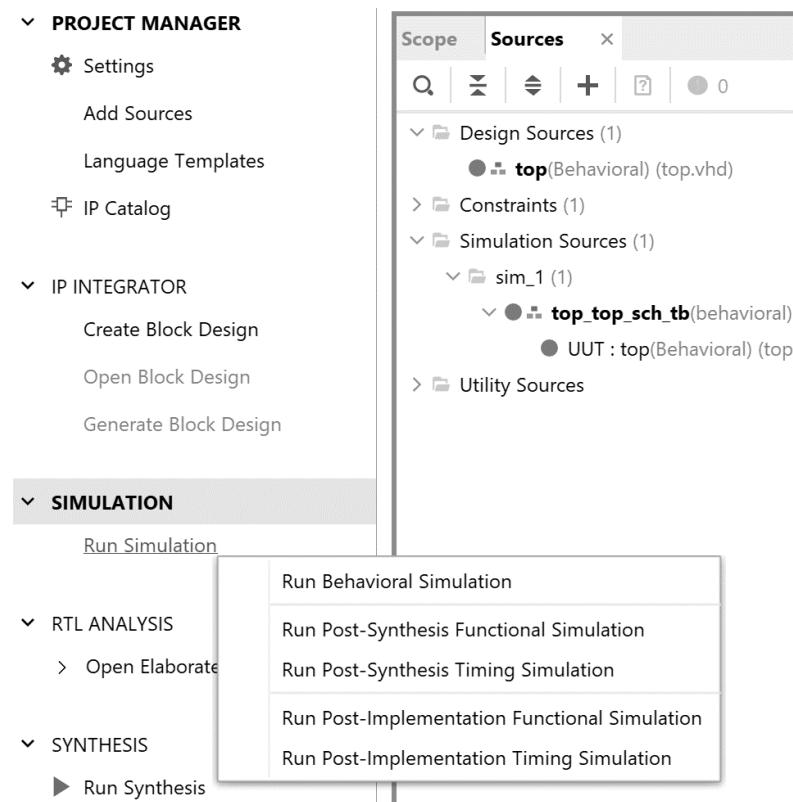


Рисунок 15.2. Запуск моделирования в САПР ПЛИС

Запуск поведенческого моделирования (Behavioral Simulation) приведет к построению временных диаграмм работы модуля. На рис. 15.3 показано, что для заданных входных воздействий симулятор автоматически определил правильное состояние выхода с.

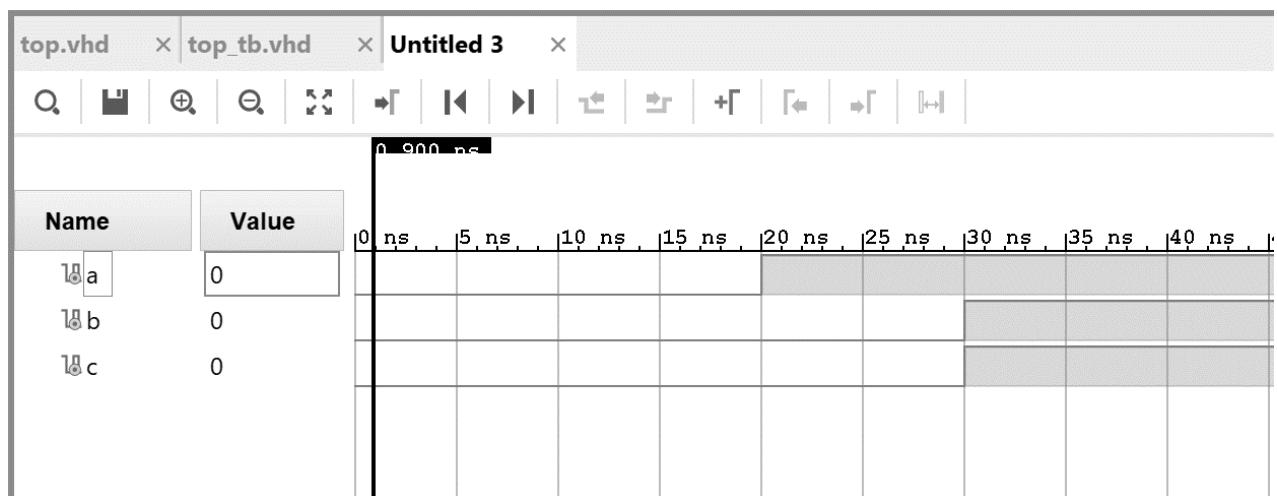


Рисунок 15.3. Пример временных диаграмм, создаваемых программой моделирования в САПР ПЛИС (поведенческое моделирование)

В САПР ПЛИС возможно проведение моделирования на физическом уровне, когда задержки распространения сигналов не принимаются равными

тем, которые указаны в поведенческом описании, а рассчитываются, исходя из физических моделей компонентов ПЛИС и трассировки конкретного проекта. Для моделирования в таком режиме необходимо выбрать пункт Run Post-Implementation Timing Simulation. Результаты моделирования показаны на рис. 15.4. С учетом задержки распространения сигнала логическая единица на выходе с появляется не в момент времени 30 нс, а в момент 33,706 нс.

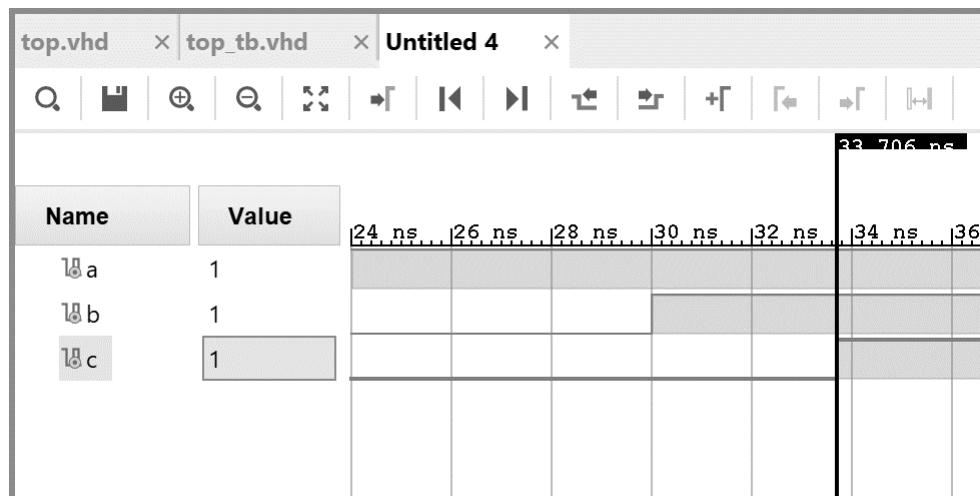


Рисунок 15.4. Пример временных диаграмм, создаваемых программой моделирования в САПР ПЛИС (физическое моделирование)

Дополнительная величина 3,706 нс не является задержкой распространения через единственный логический вентиль, что было бы чрезмерно завышенным результатом для 28-нм элементной базы. Данная задержка определена с учетом влияния входных буферов для сигналов a и b и выходного буфера выхода c, подключенных к выводам ПЛИС.

Для получения результатов Post-Route моделирования необходимо, как следует из названия, выполнить трассировку (Routing) проекта. Таким образом, результат оказывается привязан к конкретной элементной базе. Кроме того, его получение связано с необходимостью анализа физических моделей компонентов, что требует существенно большего времени по сравнению с выполнением поведенческого моделирования. Однако такой результат существенно точнее, поскольку времена распространения сигналов рассчитываются по реальной трассировке, а не вводятся в модель из субъективных соображений.

При проектировании цифровых систем моделирование на физическом уровне обычно используется на завершающих этапах верификации проекта, когда требуется подтвердить не просто правильность выполнения

преобразований, а соблюдение временных характеристик установления и распространения сигналов. В процессе отладки удобнее использовать моделирование на поведенческом уровне, поскольку оно производится существенно быстрее. В этом случае даже ориентировочные величины задержек позволяют иллюстрировать временной сдвиг сигналов друг относительно друга, что позволяет отлаживать архитектуру устройства.

15.2. Входные воздействия и наблюдение реакции системы

Более сложные моделирующие конструкции можно создавать с помощью процедурных блоков `initial` и `always`. Процедурный блок `initial` выполняется однократно в процессе моделирования, в момент времени 0, а блок `always` – каждый раз, когда изменяется любой из сигналов, перечисленных в списке чувствительности.

Пример 1:

```
initial
begin
    clk = 1b'0;
    forever #10 clk = ~clk;
end
```

В этом примере с помощью ключевого слова `forever` организуется бесконечный цикл – через каждые 10 нс значение сигнала `clk` меняется на противоположное (подразумевается, что единицы времени для моделирования установлены равными 1 нс, как это обычно бывает).

Пример 2:

```
initial
begin
    q = 1b'0;
end

always @ (a,b)
begin
    q = a & b;
end
```

В примере выходному сигналу q присваивается значение, равное нулю. Далее следует процедурный блок always, который описывает логический вентиль И.

Основным преимуществом моделируемого подмножества Verilog является возможность создания моделей, описывающих задержки распространения сигналов. Это позволяет применять такие задержки к элементам, основываясь только на информации, указанной разработчиком модели, что существенно быстрее, чем получение этой информации путем анализа физической модели этого компонента. Таким образом, разработчик модели обязан обеспечить правильные величины задержек, но это компенсируется увеличением скорости моделирования.

Задержка указывается с помощью символа #. Например, для непрерывного присваивания.

```
assign #3 q = a & b;
```

#3 показывает, что задержка распространения сигнала составляет 3 нс (точнее, 3 «единицы времени», величина которых определяется директивой `timescale, и обычно равна 1 нс).

Для логических вентилей могут указываться три величины задержек, соответствующие следующим величинам:

- время перехода в высокий уровень (rise time);
- время перехода в низкий уровень (fall time);
- время отключения (turn off time).

Эти времена указываются после символа # в скобках, в порядке, приведенном в списке.

```
assign #(2,3,4) q = a & b;
```

Необходимо еще раз подчеркнуть, что приводимые таким образом задержки используются только при моделировании. Они игнорируются средствами синтеза, которые вместо этого могут рассчитать реальные задержки распространения, учитывающие используемые компоненты, соединяющие их проводники, условия работы и т.д.

Следующие конструкции не являются синтезируемыми и предназначены только для повышения удобства описания моделей.

Конструкция wait предназначена для синхронизации работы процедурных блоков. Например, в одном процедурном блоке может моделироваться установка сигнала, который используется в другом блоке.

```
wait (changed_signal)
      a = b;
```

Указанный блок (в примере состоящий из оператора `a = b`) будет выполняться каждый раз, когда изменится значение сигнала `changed_signal`.

Конструкция while представляет собой цикл с проверкой условия. Она имеет следующий синтаксис:

```
while (<условие>)
<оператор>;
```

При необходимости выполнять в цикле несколько операторов, как обычно в подобных случаях, используются «операторные скобки» begin/end.

Команда forever представляет собой бесконечный цикл.

```
initial
begin
  clk = 1'b0;
  forever #10 clk = ~clk;
end
```

Команда repeat повторяет цикл заданное число раз.

```
initial
begin
  counter = 0;
  repeat (256)
    begin
      $display("Counter = %d", counter);
      #10 counter = counter + 1;
    end
  end
end
```

15.3. Понятие стратегии моделирования

В процессе проектирования системы разработчик может потратить практически неограниченное время на редактирование файла входных воздействий и запуск моделирования. Если набранные им тексты синтаксически корректны, при всех запусках будут строиться какие-то временные диаграммы, которые, однако, не обязательно означают, что устройство работает правильно.

Для того, чтобы моделирование не превратилось в бесполезный процесс, необходимо определить, с какой степенью детализации будут проводиться проверки отдельных систем, сколько будет таких проверок, какие специальные условия и сочетания сигналов необходимо проверить и что является признаком успешного окончания проверки. Все это образует понятие стратегии моделирования.

Важность выбора стратегии моделирования является следствием того, что выбрать единственную «оптимальную стратегию» на практике невозможно. Например, моделирование путем полного перебора входных значений выглядит привлекательным, поскольку на первый взгляд позволяет проверить все возможные варианты работы устройства. Однако даже для 32-разрядного сумматора такой перебор означает подачу 2^{64} сочетаний входных значений, т.е. $1,8 \cdot 10^{19}$. Такой перебор займет сотни лет, хотя речь идет о проверке одного небольшого цифрового узла.

С другой стороны, проверка 2-3 очевидных случаев оставляет открытыми множество вопросов. Например, как поведет себя устройство при переполнении разрядной сетки результата? Если проверена передача байтов 0 и 255, состоящие из одинаковых битов, то в правильном ли порядке передаются отдельные биты? Как поведет себя системная шина, если адресуемое устройство не выдает сигнал подтверждения готовности? Все эти вопросы требуют специального внимания к организации проверок.

Одним из практических подходов является сочетание *направленного тестирования* (directed testing) и *псевдослучайного тестирования* (pseudo-random testing). Например, в случае сумматора могут быть проверены ситуации, для которых разработчик потенциально может ожидать некорректного поведения – например, сложение чисел, при котором выполняется перенос бита в старшие разряды. Эти проверки можно дополнить подачей приемлемо большого количества чисел, выбираемых генератором псевдослучайных чисел.

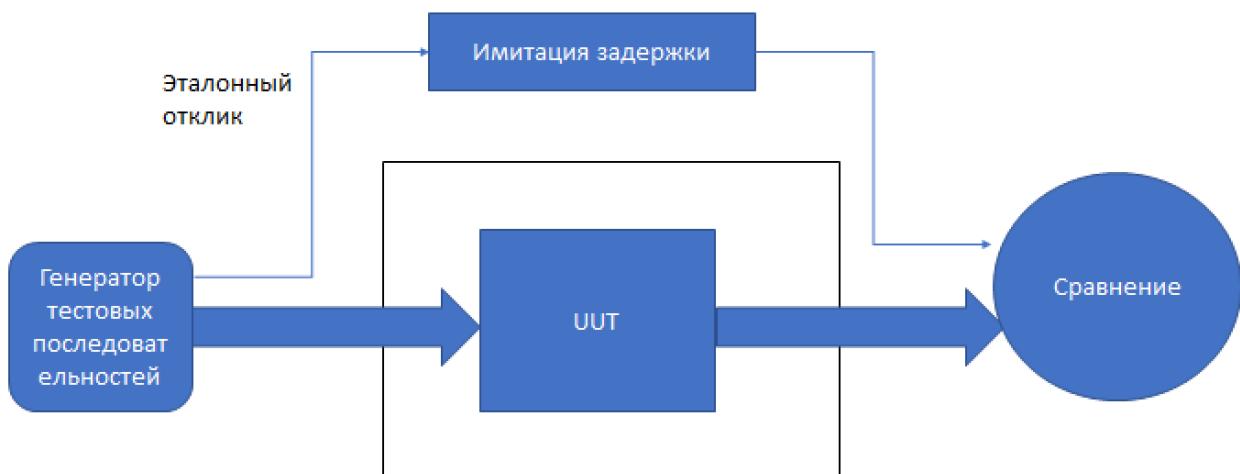
Формально, используемые в программировании генераторы являются псевдослучайными, поскольку используют алгоритмы генерации, основанные на получении следующего значения последовательности из предыдущего по

детерминированной формуле. Случайными являются, например, аппаратные генераторы на основе измерения тепловых шумов полупроводникового диода, которые редко используются в технике. В то же время, использование функции random с учетом неконтролируемого начального значения последовательности для практики можно отнести к генераторам случайных чисел.

Тем не менее, при создании тестовых воздействий рекомендуется использовать именно контролируемые последовательности. Это необходимо для того, чтобы после обнаружения ошибки ее можно было воспроизвести.

Таким образом, в качестве отправной точки для организации моделирования можно выбирать сочетание проверок сценариев с очевидно понятными результатами и подачу достаточно большой последовательности псевдослучайных значений, которые могут способствовать выявлению неявных проблем в проекте.

С практической точки зрения полезно организовывать автоматизированное тестирование. На рис. 15.5 показан простой пример проверки периферийного устройства.



- «Проведено N тестов, закончились успешно N, ошибок 0»

Рисунок 15.5. Организация автоматизированного тестирования

Тестируемое устройство, показанное как UUT, подключено к генератору тестовых последовательностей. Генератор формирует, например, сигнал rx интерфейса UART. При этом ожидаемый эталонный отклик (т.е. передаваемы байт) подается внутри модели через блок имитации задержки на устройство сравнения. Имитация задержки требуется для того, чтобы воспроизвести работу контроллера UART, который записывает принятый байт через определенное

время после начала передачи. Сравнение, реализованное в модели, определяет, совпадает ли выход UART с эталонным значения передаваемого байта.

В такой модели легко реализовать проверки по разным сценариям. Например, описывая правила генерирования разных значений, можно реализовать как направленное тестирование, так и псевдослучайное. Конечным результатом такой проверки можно считать отчет «проведено N тестов, закончились успешно N, ошибок 0». При необходимости можно подробно изучить правила генерации входных значений, использованные алгоритмы и правила проверки.

При разработке цифровых систем применяются следующие виды тестирования.

Моделирование представляет собой запуск программы-симулятора, демонстрирующей реакцию разрабатываемого устройства на входные воздействия в виде диаграмм сигналов и текстовых файлов. Моделирование интенсивно используется на ранних стадиях разработки, когда требуется уточнить поведение системы, проверить корректность разработанных описаний и убедиться, что реакция на входные воздействия соответствует техническому заданию, а система в целом действительно решает поставленную задачу.

Стендовые испытания (испытания в лабораторных условиях) подразумевают проверку работы устройства в контролируемой среде. При этом используются лабораторные генераторы, контрольно-измерительное оборудование, а специальные воздействия ограничиваются. Поскольку испытания проводятся на реальном экземпляре разрабатываемого изделия, становится возможным проверить адекватность функциональных моделей и убедиться, что использованные схемотехнические решения действительно работоспособны на реальной микросхеме. На этом этапе могут быть выявлены отклонения от синхронного стиля проектирования (симулятор не может автоматически проверить уход фазы тактового сигнала, влияние джиттера, возникновение метастабильных состояний и прочие отклонения реального кристалла от идеализированной математической модели), проблемы в организации питания, отвода тепла от микросхемы, влияние эффектов дискретизации по уровню и времени при обработке аналоговых сигналов.

Интеграционные тесты проводятся для исследования влияния системных эффектов. Они проводятся при сопряжении прибора с реальными устройствами, причем на этом этапе может быть обнаружено, что синтетические тесты неадекватно отразили реальную ситуацию. Например, блок питания системы может иметь повышенный уровень шумов по сравнению с лабораторным

источником питания, который использовался при отладке. Дополнительные шумы могут также генерироваться источниками, которые не были учтены при лабораторных испытаниях. Реальная коммуникационная среда может иметь другой уровень нагрузки (количество пакетов, их состав, уровень потерь и т.п.). Это может привести к тому, что результаты лабораторного моделирования будут признаны неадекватно отражающими реальную картину.

Испытания в полевых условиях подразумевает проведение тестовой эксплуатации устройства в реальных условиях. На этом этапе могут быть выявлены такие эффекты, как неработоспособность при реальных сочетаниях температуры эксплуатации, уровня помех, механических и электромагнитных воздействий, низкая эргономичность или ремонтопригодность.

В целом можно отметить, что использование программ-симуляторов и даже стендовых испытаний не является абсолютной гарантией работоспособности изделия в реальных условиях эксплуатации. Поэтому необходимо планировать дополнительное время на проведение испытаний в реальных условиях с последующей коррекцией схемы или даже конструкции изделия в целом.

15.4. Системное моделирование

Под системным моделированием понимается тестирование поведения функционально завершенной системы, в отличие от проверки отдельных узлов и сценариев их взаимодействия. Например, для системы цифровой обработки сигналов такой проверкой может быть фильтрация зашумленного сигнала с правильным измерением параметров полезной информационной составляющей. Для процессорного устройства это может быть моделирование выполнения прикладной программы с получением итогового результата.

Системное моделирование процессора может быть основано на загрузке в память процессора кодов программы и запуске модели с поданным тактовым сигналом. При этом записанная в памяти программа будет самостоятельно инициировать выполнение определенных действий. При необходимости следует сформировать входные воздействия на периферийные устройства процессора.

Системная модель генерирует существенно больше данных для проверки по сравнению с тестированием отдельных узлов процессора. Возможно выявление скрытых дефектов проекта, связанных с неверной архитектурой или ошибками в проектировании логики управления.

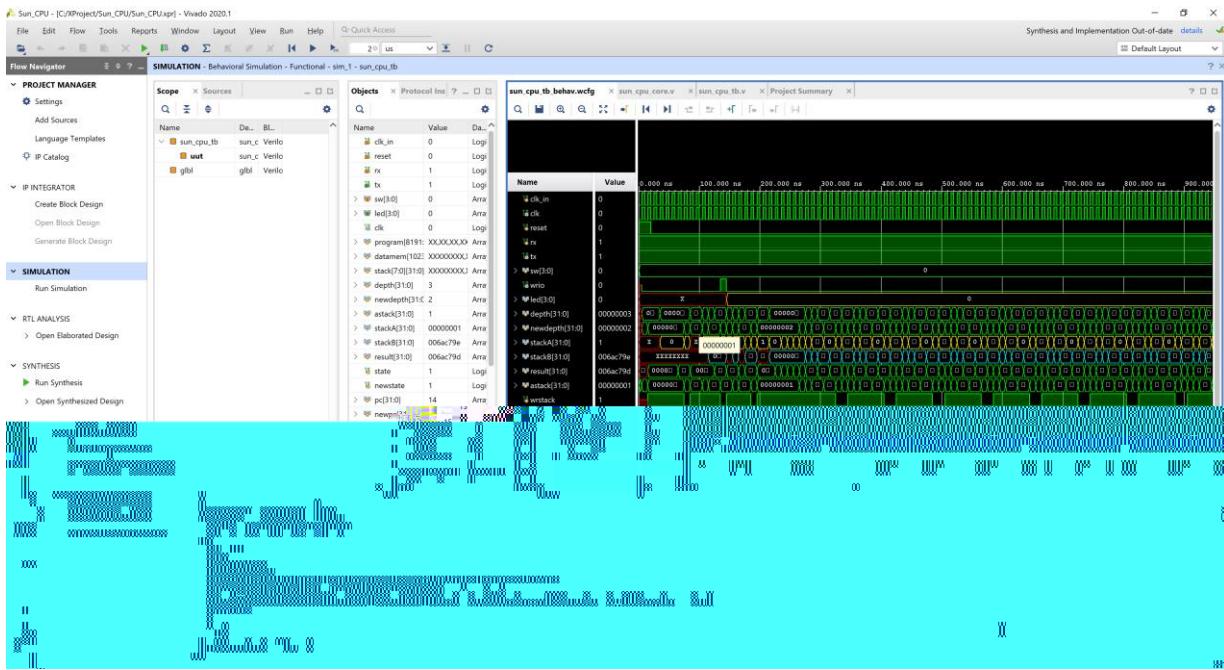


Рисунок 15.6. Пример временных диаграмм при системном моделировании процессора

15.5. Выводы по разделу

Моделирование позволяет разработать и отладить цифровую систему существенно быстрее и с меньшими затратами по сравнению с экспериментальными проверками изготовленного образца устройства. Основой моделирования является описание входных воздействий на цифровую схему с наблюдением результатов в программе-симуляторе.

Важным вопросом является выбор стратегии моделирования – правил задания входных воздействий, количества проверок и правил сопоставления выходов с ожидаемыми значениями. Важность выбора стратегии моделирования обусловлена тем, что полный перебор сочетаний входных сигналов занимает неприемлемо большое время.

Моделирование необходимо сочетать с лабораторными испытаниями, поскольку в модели могут оказаться неучтенными существенные эффекты – например, шумы, импульсные помехи, отклонения частоты тактового генератора и т.п., которые являются предметом экспериментальной проверки.

Процессы моделирования следует автоматизировать, чтобы избегать непроизводительных ручных запусков САПР с наблюдением отдельных сценариев. Например, для процессора эффективным приемом является заполнение памяти процессора программой и запуск моделирования, при котором команды программы будут самостоятельно инициировать выполнение тех или иных операций.

Контрольные вопросы:

1. Как организовать моделирование цифрового устройства?
2. Можно ли использовать в описании входных воздействий операторы, управляющие задержкой сигнала? Как описать задержку на 10 нс?
3. Как систематизировать процесс моделирования, организовав автоматическую проверку и выявление ошибок при проектировании?
4. Что произойдет, если память процессора заполнить программой и запустить процесс моделирования?

16. ПРАКТИЧЕСКИЕ ВОПРОСЫ ПРОЕКТИРОВАНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

16.1. Постановка задачи проектирования

Для проектной деятельности применимы два взаимодополняющих подхода к разработке: «снизу вверх» (также bottom up) и «сверху вниз» (top down). В первом случае модули проекта последовательно разрабатываются и отлаживаются, после чего производится их сборка и проверка корректности совместной работы. Процесс продолжается до тех пор, пока в результате объединения не будет получен верхний уровень схемы. При построении проекта «сверху вниз» спроектированный верхний уровень разбивается на субмодули до тех пор, пока не будет получен набор компонентов, каждый из которых достаточно прост для реализации.

Категорическое следование одному из подходов обычно оказывается малоэффективным. При проектировании «снизу вверх» может оказаться, что разработчики сосредоточились на реализации малозначимых деталей, и по истечении длительного времени выяснилось, что к проектированию ключевых компонентов проекта так никто и не приступил (однако реализовано большое количество элементарных субмодулей уровня «мультиплексор», «счетчик» и т.п.).



Рисунок 16.1. Этапы проектирования вычислительной системы

При проектировании «сверху вниз» можно оказаться в подобной ситуации, когда проект будет разбит на абстрактные модули «обработка», «интерфейс» и т.п., однако конкретные технические параметры, алгоритмы обработки, используемые сигналы так и не будут четко определены, чтобы разработчики смогли приступить к непосредственному кодированию.

Этапы проектирования вычислительной системы укрупненно показаны на рис. 16.1 Практическим выводом такой классификации является возможность создания рабочих групп и привлечения разработчиков с разной специализацией, чтобы они могли сосредоточиться на определенном круге задач.

16.2. Жизненный цикл проекта и связь со смежными специалистами

В проектной деятельности существует понятие жизненного цикла разработки. Под этим термином понимается последовательность операций при выполнении проекта. Она не ограничивается отдельными действиями, известными программисту или конструктору цифровой схемы. Более того, отдельно подчеркивается, что разработка является только одним из этапов жизненного цикла. Приводится следующий список:

1. Исследование концепции
2. Исследование системы
3. Требования
4. Разработка проекта
5. Внедрение
6. Установка
7. Эксплуатация и поддержка
8. Сопровождение
9. Вывод из эксплуатации

Для инженеров, выполняющих проектирование цифровой части, основной интерес представляет пункт 4 – «разработка проекта». Более того, для них этот пункт может восприниматься не только как преобладающий по важности, но и как единственный, имеющий реальное значение. С одной стороны, чрезмерное внимание к организационным вопросам может быть следствием недостаточной квалификации. С другой, полное игнорирование, и даже пренебрежение организационными аспектами на практике приводит к затягиванию сроков выпуска коммерчески успешного продукта на рынок, и даже к затягиванию сроков получения работоспособного изделия.

Концентрация на процессе разработки не позволяет адекватно ответить на целый ряд важных вопросов. Например, сохранит ли устройство актуальность до

завершения разработки? Может ли поставленная задача быть решена на базе выбранных алгоритмов? Как соотносятся технические параметры и потребительская ценность изделия? Можно указать на тот факт, что такие показатели как «количество операций умножения с накоплением» или «суммарная пропускная способность приемопередатчиков» только косвенно отражает ценность продукта для конечного пользователя. Установить связь между техническими и потребительскими характеристиками необходимо именно в процессе организационных мероприятий.

В процессе выполнения проекта могут использоваться различные модели жизненного цикла. Эти модели не образуют исчерпывающего списка, применяемые методы могут комбинироваться, однако все же рекомендуется отслеживать соответствие организации работ особенностям проекта.

Каскадная модель жизненного цикла подразумевает последовательное выполнение шагов. В рамках этой модели не производится возврат к ранее завершенным шагам, т.е., например, разработчик не может по собственному желанию изменить алгоритмы работы или спецификации интерфейсов. Таким образом, каскадная модель не обеспечивает большой гибкости, однако при условии тщательного планирования позволяет разбить проект на известное количество шагов, каждый из которых может быть подробно описан. Качественная реализация каждого шага является залогом успешного выполнения следующего шага. Каскадная модель эффективна при реализации проектов в известной производителю области, когда работа производится квалифицированным коллективом, техническое задание не содержит элементов, подлежащих экспериментальной проверке, а сроки и стоимость исполнения проекта не вызывают опасений.

Инкрементная модель предполагает выполнение нескольких итераций (обычно трех) в процессе разработки. Эта модель может быть применена при наличии заметной исследовательской составляющей в проекте. В этом случае трудно полагаться на адекватность умозрительных оценок, которые вынуждены быть сформулированы без исследования реальных сигналов, при отсутствии образца платы с ПЛИС и неполной формализации задачи. Каждая итерация доводится до этапа пробной эксплуатации, причем заранее предполагается, что первые итерации являются пробными и не должны поступать в коммерческую эксплуатацию. Инкрементная модель обеспечивает большую гибкость по сравнению с каскадной и позволяет планировать работы, для которых трудно сформулировать точные алгоритмы функционирования, определить потребляемую мощность или достижимую частоту.

Сpirальная модель схожа с инкрементной, однако процесс внесения изменений является непрерывным. Следующая фаза планируется параллельно с реализацией текущей фазы и основывается на достигнутых параметрах.

Приведенным списком модели жизненного цикла не ограничиваются. В данной публикации они приведены в качестве иллюстрации того, что организация работ может быть выполнена различным образом, и не стоит ожидать от исследовательской группы тех же промежуточных показателей, что и от конструкторского отдела, выполняющего модификацию ранее разработанного изделия. В первом случае можно ожидать получения первой итерации инкрементной модели, а во втором – завершения проектирования по каскадной модели разработки. Однако во втором случае сложно ожидать существенного улучшения характеристик по сравнению с предыдущей моделью изделия, а тем более – эффективного решения новой задачи.

Выбор подходящей модели жизненного цикла позволит определить «стиль» выполняемых проектных работ. Это позволит коллективу исполнителей избегать проектных мероприятий, явно идущих вразрез с выбранной моделью. Например, попытка отдельного разработчика спонтанно организовать глубокую переработку алгоритмов обработки сигнала в крупном конструкторском бюро с высокой вероятностью обречена на неудачу, так как это повлечет за собой потерю результатов, полученных его коллегами в рамках реализации других алгоритмов. Если его решение основано на кажущемся упрощении его собственного фронта работ, то правильнее было бы сосредоточиться на выполнении текущего технического задания, а не нарушать согласованную работу большого коллектива. Аналогично, в малой исследовательской группе крайне неэффективно было бы доводить любой эксперимент с алгоритмами обработки сигналов до стадии выпуска полного комплекта конструкторской документации, поскольку от исследовательской группы ожидается максимально полный охват вопросов, связанных с архитектурой изделия и алгоритмами его работы.

В целом можно указать на пользу от привлечения смежных областей знаний в процесс собственно разработки цифровой аппаратуры. Привнесение организационного компонента проектирования не должно заменять собой собственно конструкторскую деятельность, однако игнорирование процессов организации, и даже противопоставление инженеров и менеджеров проекта следует считать недостатком.

16.3. Математические модели предметной области, проектирование на системном уровне

При исследовании проектируемой системы и составлении математических моделей сигналов может оказаться, что чрезмерно упрощенные и благоприятные предположения о характеристиках обрабатываемого сигнала дают возможность использовать простые для реализации схемы для их обработки. Часто сам термин «фильтр» наводит на мысль, что помехи во входном сигнале будут устраниены, а качество фильтрации является легко регулируемым параметром, решаемым подбором характеристик фильтра.

Можно привести разновидности нереалистичных представлений о входных сигналах.

1. Сигнал является чисто гармоническим.

Такой сигнал, изображенный в идеальном виде, имеет на каждом периоде единственные максимум и минимум, а также две точки, в которых график пересекает ось X. Эти особенности идеального гармонического сигнала позволяют применять методы обработки, корректные с точки зрения теоретической математики, но не работающие для реальных сигналов, в которых из-за шумов будет наблюдаться множественное пересечение оси X, множество локальных максимумов и минимумов, причем полагаться на какие-то определенные характеристики помех невозможно.

2. Сигнал имеет помеху в виде «белого шума».

Добавление к сигналу смоделированного шума, на первый взгляд, делает исследование более корректным. Однако зачастую такое зашумление устраняется достаточно простыми методами, такими как ФНЧ.

На практике может оказаться, что кроме белого шума в сигнале присутствуют «выбросы» и другие виды нерегулярных помех. В сигнале могут также присутствовать долговременные тренды, такие как смещение постоянной составляющей, изменение дисперсии шума, его частотной зависимости и т.д. Можно обратить внимание, что теория линейных систем, на которой в основном базируются математические методы расчета фильтров, подразумевает стационарность сигнала, т.е., для практического приложения это означает, что характеристики частотных составляющих, помех, шумов и т.п. будут одинаковыми для любого момента времени. Современные сложные системы ставят это под большое сомнение. Помехи от источников питания (особенно импульсных), ШИМ-регулирование силовых исполнительных устройств, работа радиостанций и других источников помех – далеко не полный перечень

факторов, которые могут нарушить стационарность сигнала и заметно ухудшить показатели качества системы.

На рис. 16.2 показан пример сигнала, полученного экспериментально. Ожидая, что сигнал будет синусоидальным с присущими функции синуса идеализированными свойствами, разработчик может неоправданно сформулировать заниженные требования к операциям по обработке такого сигнала.

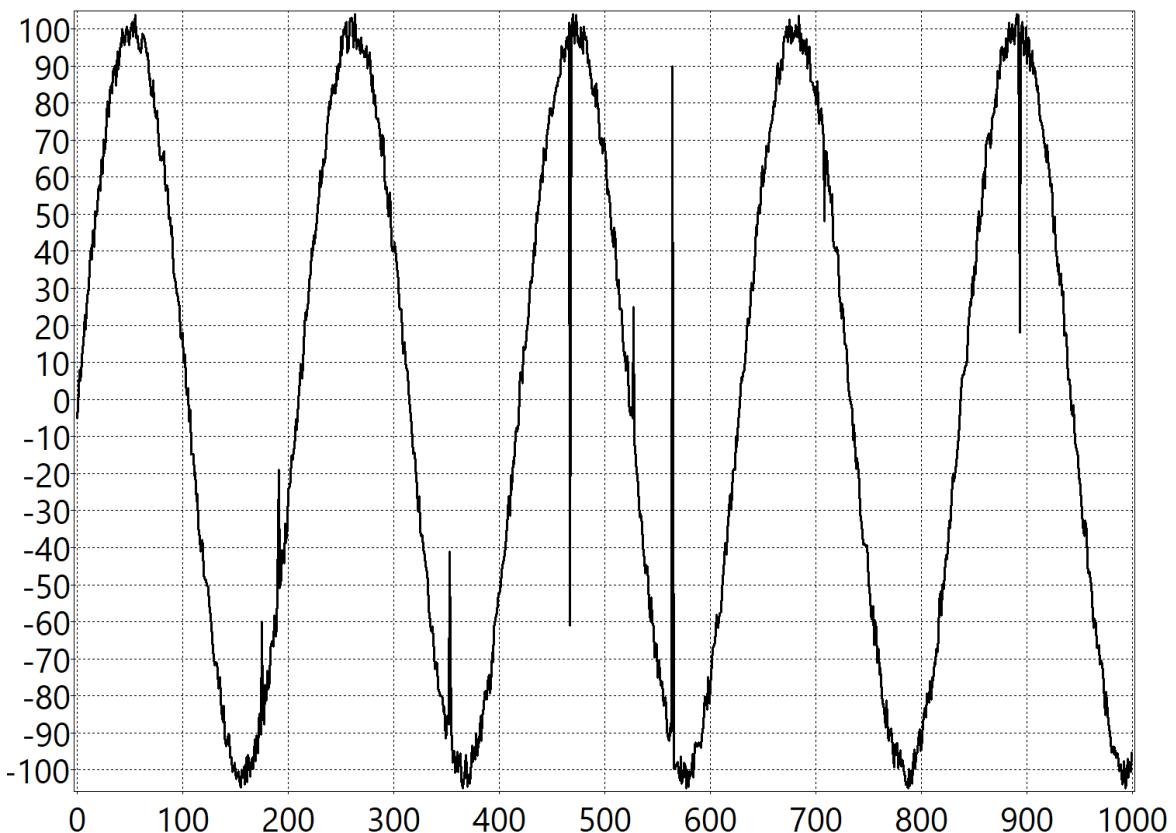


Рисунок 16.2. Пример сигнала, полученного экспериментально

Временной альтернативой получению экспериментальных данных является разработка математически обоснованных моделей таких сигналов. Однако при появлении возможности проведения исследований с использованием макета следует экспериментально проверить корректность математической модели.

В целом на принятие схемотехнических решений оказывает влияние целый ряд факторов, показанных на рис. 16.3.

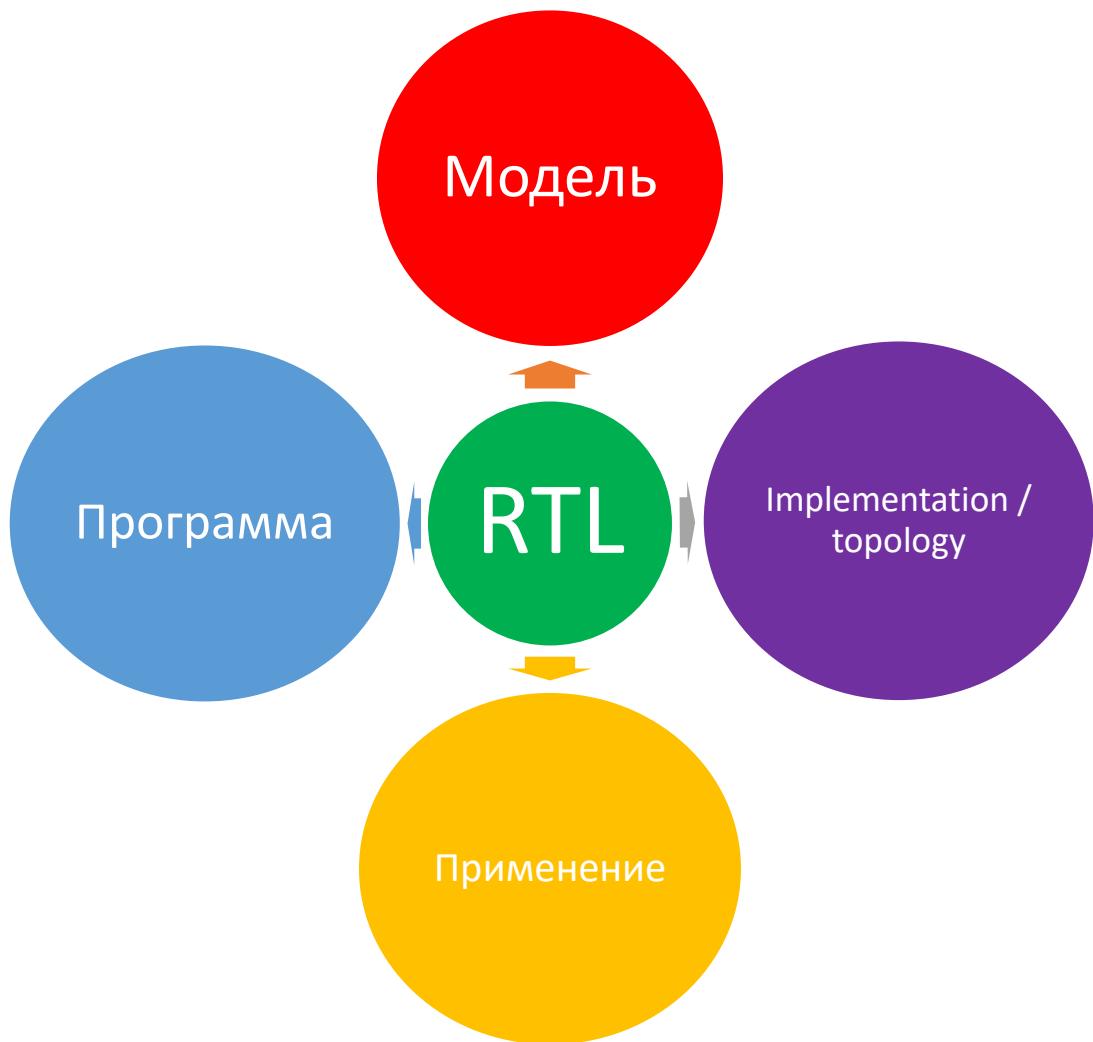


Рисунок 16.3. Компоненты проекта, влияющие на принятие схемотехнических решений

Итоговый вид разработанного изделия обычно является результатом принимаемых компромиссов между техническими требованиями, экономическими факторами, возможностями используемых технологий и инструментов, наличием подготовленного персонала и др. Именно поэтому проектирование требует высокой квалификации специалиста, способного принять работоспособные решения и реализовать их с учетом имеющихся ресурсов.

16.4. Выводы по разделу

Организация проектной деятельности – сложный процесс, слабо подлежащий формализации. Выполнение технических работ является следствием организационных решений, которые должны быть предварительно запланированы, а ожидаемые результаты – оценены с последующей проверкой достижения поставленных целей. Получение конкретных технических

результатов должно использоваться для коррекции планов и принятию уточняющих требований к следующим этапам работ.

Конечный вид изделия, используемые материалы, компоненты, технологии и инструменты являются итогом комплекса принятых технических и организационных решений, которые учитывают как требования к будущему продукту, так и имеющиеся ресурсы.

Контрольные вопросы:

1. Какие факторы влияют на процесс проектирования?
2. Какие явные противоречия в требованиях могут возникать в процессе принятия технических решений?
3. Какие отклонения от идеальной формы могут иметь реальные сигналы?
4. Какие отклонения от идеального сценария могут возникать в компьютерных сетях?
5. Какие организационные подходы к разработке существуют и как их можно использовать для планирования технических работ?

СПИСОК ЛИТЕРАТУРЫ

1. Тарасов И. Е. ПЛИС Xilinx. Языки описания аппаратуры VHDL и Verilog, САПР, приемы проектирования. М.: Издательство: Горячая линия - Телеком, 2019 г. ISBN: 978-5-9912-0802-4
2. Паттерсон Д., Хенnessи Дж. Архитектура компьютера и проектирование компьютерных систем. 4-е изд. СПб.: Питер, 2012. – ISBN 978-5-459-00291-1;
3. Харрис Дэвид М., Харрис Сара Л. Цифровая схемотехника и архитектура компьютера. Издательство: ДМК-Пресс, 2018 г.
4. Максфилд К. Проектирование на ПЛИС. Курс молодого бойца. – М.: Издательский дом «Додэка-XXI», 2007. – 408 с.: илл. (Серия «Программируемые системы»).
5. Макконнелл С. Совершенный код. Мастер-класс / Пер. с англ. – М.: Издательство «Русская редакция», 2013. – 896 с.: ил.
6. Шафер Д., Фатрелл Р., Шафер Л. Управление программными проектами: достижение оптимального качества при минимуме затрат: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. – 1136 с.: ил. – Парал. тит. англ.
7. Таненбаум Эндрю, Остин Т. Архитектура компьютера. Издательство: Питер, 2019 г. Серия: Классика computer science. ISBN: 978-5-4461-1103-9, 816 с.
8. Вонг Б.П., Миттал А., Цао Ю., Стэрр Г. Нано-КМОП-схемы и проектирование на физическом уровне. Москва: Техносфера, 2014. – 432 с., ISBN 978-5-94836-377-6.
9. Джонс М.Х. Электроника – практический курс. Изд. 3-е, исправленное. Москва: Техносфера, 2021. – 512 с. ISBN 978-5-94836-341-7.

СВЕДЕНИЯ ОБ АВТОРАХ

Tarasov Илья Евгеньевич, доктор технических наук, доцент, профессор кафедры «Корпоративные информационные системы» РТУ МИРЭА.