БИБЛИОТЕКА СОВРЕМЕННОЙ ЭЛЕКТРОНИКИ

Джон Ф. Уэйкерли

Проектирование цифровых устройств

том II

Перевод с английского Е.В. Воронова, А.Л. Ларина

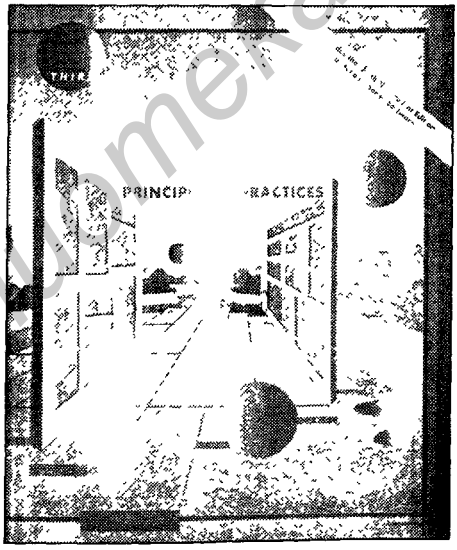
ПОСТМАРКЕТ
МОСКВА
2002

Дж. Ф. Уэйкерли
Проектирование цифровых устройств, том 2.
Москва:
Постмаркет, 2002. – 528 с.

Основополагающий учебник, в котором рассмотрены все направления современной цифровой электроники. Особое внимание уделено программируемым логическим интегральным схемам (ПЛИС).

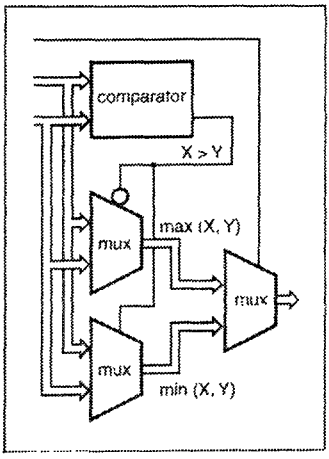
На двух прилагаемых лазерных дисках размещено программное обеспечение фирмы Xilins.

Предназначен для студентов, аспирантов и преподавателей ВУЗов, разработчиков аппаратуры.



ISBN 5-901095-12-X

Содержание



Глава 6

ПРИМЕРЫ ПРОЕКТИРОВАНИЯ КОМБИНАЦИОННЫХ СХЕМ

В предыдущих главах рассмотрены основные понятия, относящиеся к системам счисления, цифровым схемам и комбинационной логике, и описаны главные составные блоки комбинационных устройств: дешифраторы, мультиплексоры и т.п. Все это является необходимой основой. Но конечной целью при изучении цифровой электроники является способность решать реальные задачи, возникающие в процессе разработки цифровых систем. Как правило, для решения этих задач, помимо чтения учебника, требуется опыт. В этой главе мы пытаемся дать вам возможность продвинуться в этом направлении, разбирая примеры проектирования больших комбинационных схем.

Глава состоит из трех параграфов. Первый параграф содержит примеры проектирования путем использования стандартных комбинационных схем. Эти примеры рассматриваются в терминах функций, реализуемых посредством ИС средней степени интеграции. Но те же самые функции широко используются при разработке на основе специализированных ИС и микросхем типа FPGA. Основная мысль здесь состоит в том, чтобы показать, что требуемую комбинационную функцию часто можно реализовать, используя набор меньших по размерам стандартных блоков. Это важно по двум причинам: во-первых, иерархический подход обычно упрощает задачу проектирования в целом; во-вторых, меньшие стандартные блоки часто более эффективно и оптимально реализуются внутри ИС типа FPGA и в специализированных ИС по сравнению с тем, чего вы добились бы, описав требуемое устройство как единое целое и поручив программным средствам его синтезировать.

Во втором параграфе приведены примеры проектирования на языке ABEL. При этом предполагается, что реализация будет осуществлена на небольших ПЛУ типа 16V8 и 20V8. Помимо собственно использования языка ABEL, некоторые примеры служат иллюстрацией того, как нужно принимать решение о разбиении на части, когда проектируемое устройство целиком не умещается в одной интегральной схеме.

Третий параграф посвящен использованию языка VHDL, который лучше всего подходит для больших проектов, реализуемых в одной микросхеме типа CPLD или FPGA и в специализированной ИС. Обратите внимание, что в этих примерах не конкретизируется ИС, в которой должно быть реализовано разрабатываемое уст-

ройство. Несомненно, это – одно из достоинств проектирования на языках описания схем; большинство таких разработок, если не все, оказываются «переносимыми» и их можно реализовать на основе любой из множества технологий.

Единственным необходимым условием для изучения этой главы является знакомство с содержанием предшествующих глав. Параграфы этой главы в значительной степени независимы один от другого, поэтому вы можете не читать материал, относящийся к языку ABEL, если вас интересует только язык VHDL, и наоборот. Кроме того, остальная часть книги написана так, что вы можете прочесть эту главу сейчас или пропустить ее и вернуться к ней позже

6.1. Примеры проектирования на основе стандартных блоков

6.1.1. Устройство быстрого сдвига

Устройство быстрого сдвига (*barrel shifter*) представляет собой комбинационную логическую схему с *n* входами данных, *n* выходами данных и несколькими управляющими входами, сигналы на которых задают сдвиг между данными на входе и данными на выходе. Устройство быстрого сдвига, являющееся частью микропроцессора, обычно характеризуется такими параметрами, как направление сдвига (влево или вправо), тип сдвига (циклический, арифметический или логический) и величина сдвига (обычно от 0 до *n* –1 разрядов, но иногда от 1 до *n* разрядов).

В этом разделе мы будем строить простое 16-разрядное устройство быстрого сдвига, осуществляющее только циклические сдвиги влево; величина сдвига пусть определяется сигналами, поступающими на 4-разрядный управляющий вход S[3:0]. Если, например, входное слово имеет вид: ABCDEFHGIJKLMNOP (где каждая буква представляет собой один бит), а сигнал управления равен 0101 (5), то выходное слово равно FGHIJKLMNOPABCDE.

На первый взгляд, решение этой задачи обманчиво просто. Каждый выходной бит можно получить с помощью 16-входового мультиплексора, на входы данных которого поступают соответствующие биты данных. Величина сдвига определяется сигналами на управляющих входах. Но, глядя на такую схему внимательнее, мы видим, что нужно идти на компромисс между быстродействием и размерами схемы.

Рассмотрим сначала варианты, в которых используются готовые мультиплексоры в виде ИС средней степени интеграции. Одноразрядный 16-входовой мультиплексор можно составить из двух ИС 74x151, используя сигнал на входе S3 и его инверсию в качестве сигналов, поступающих на входы EN_L этих схем, и объединяя с помощью вентилей И-НЕ сигналы на выходах данных Y_L, так, как было показано на рис. 5.66 для 32-входового мультиплексора. Младшие разряды сигнала управления сдвигом S2–S0 подаются на одноименные входы выбора микросхем '151.

Задача будет решена, если мы 16 раз повторим этот 16-входовой мультиплексор и подключим входы данных согласно схеме на рис. 6.1. На верхние ИС '151 в каждой паре поступает сигнал разрешения S3_L, а на нижние – сигнал разрешения S3; остающиеся сигналы выбора подаются на все 32 микросхемы '151. На входы данных D0–D7 каждой из ИС '151 поступают сигналы DIN в том порядке, в каком они перечислены на рисунке слева направо.

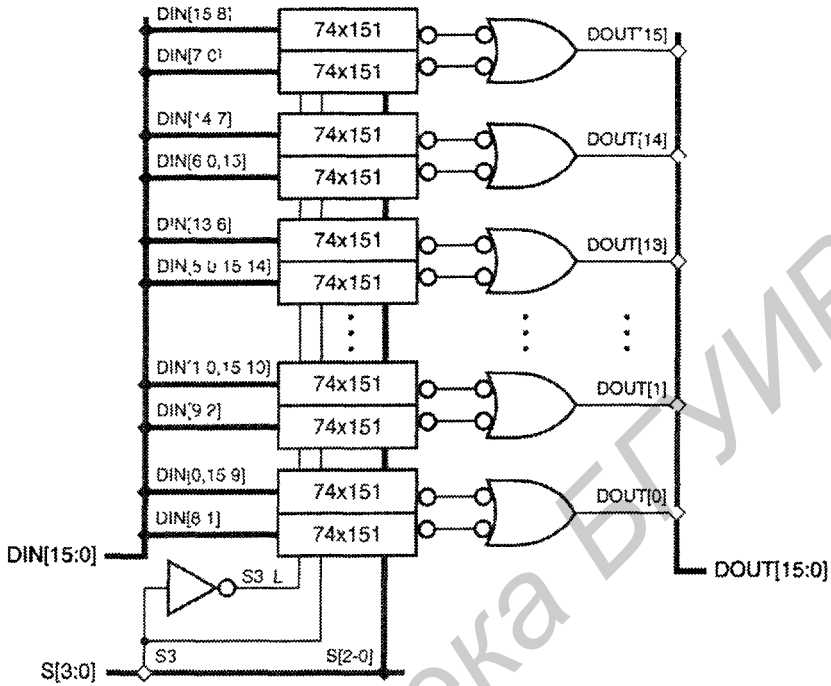


Рис. 6.1. Один из подходов к построению 16-разрядного устройства быстрого сдвига

В первой строке в табл. 6.1 приведены характеристики этого варианта. Для его реализации потребуется 36 микросхем средней и малой степени интеграции (точнее, чуть больше: 32 ИС 74x151, 4 ИС 74x00 и 1/6 ИС 74x04). Число микросхем можно уменьшить до 32, заменяя ИС 74x151 на ИС 74x251 и соединяя вместе их выходы Y с тремя состояниями; в результате получим то, что указано во второй строке таблицы. В каждом из этих вариантов оказывается очень сильно нагруженным источник управляющих сигналов: сигнал каждого разряда управляющего слова S[2:0] необходимо подать на одноименный вход выбора всех 32 мультиплексоров. Входы данных также довольно сильно нагружают источники; сигнал каждого разряда данных должен быть подан на входы 16 мультиплексоров, соответствующих 16 возможным значениям

Табл. 6.1. Свойства четырех различных вариантов устройства быстрого сдвига

Используемый мультиплексор	*Нагрузка по шине данных*	*Задержка данных*	*Нагрузка по шине управления*	*Число ИС*
74x151	16	2	32	36
74x251	16	1	32	32
74x153	4	2	8	16
74x157	2	4	4	16

сдвига. Если предположить, что значительная нагрузка по шине управления и по шине данных не слишком сильно замедляет работу схемы, то вариант с микросхемами 74x251 дает наименьшую задержку по отношению к данным, поскольку сигнал в каждом разряде данных проходит лишь через один мультиплексор

С другой стороны, можно создать устройство быстрого сдвига на 16 2-входовых 4-разрядных мультиплексорах 74x157, параметры такого устройства приведены в последней строке таблицы. Сначала данные проходят через 2-входовой 16-разрядный мультиплексор, образуемый первым набором из четырех микросхем 74x157 (рис. 6.2). В зависимости от значения управляющего сигнала S0 входное слово оказывается сдвинутым влево на 0 разрядов или на 1 разряд Выходы данных первых четырех мультиплексоров соединены с входами второго набора мультиплексоров, которые по сигналу S1 сдвигают свое входное слово влево на 0 разрядов или на 2 разряда. Последовательно пропуская получающиеся слова через третий и четвертый наборы мультиплексоров, управляемые соответственно сигналами S2 и S3, как показано на рис. 6 2, мы можем осуществить сдвиг на 4 и на 8 разрядов. Сигналы на входы 1A–4A и 1B–4B каждой ИС '157 подаются в том порядке, в каком они перечислены на схеме слева направо; то же самое относится к выходам 1Y–4Y каждой ИС.

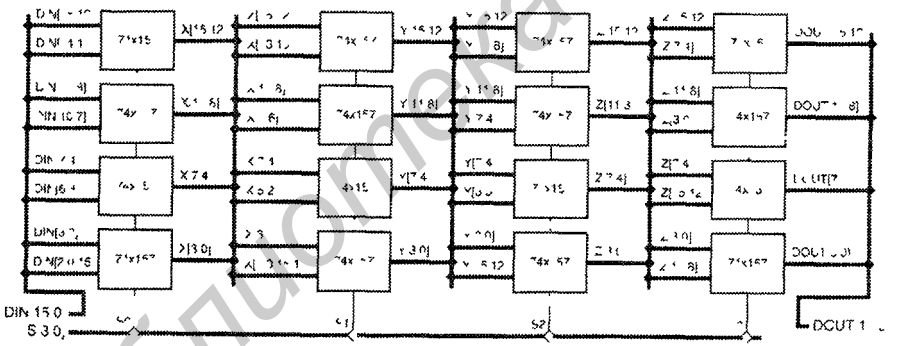


Рис. 6.2. Другой подход к построению 16-разрядного устройства быстрого сдвига

Для устройства на ИС '157 требуется вдвое меньшее число микросхем средней степени интеграции, и оно гораздо меньше нагружает источники управляющих сигналов и сигналов данных. Но такое решение дает наибольшую задержку в канале данных, так как сигнал каждого разряда данных должен пройти через четыре ИС 74x157.

Промежуточным между рассмотренными двумя подходами является вариант, основанный на использовании восьми 4-входовых 2-разрядных мультиплексоров 74x153 для реализации 4-входового 16-разрядного мультиплексирования Два таких набора микросхем нужно включить последовательно один за другим, используя сигналы S[3.2] для сдвига на 0, 4, 8 или 12 разрядов, а сигналы S[1.0] – для сдвига на число разрядов от 0 до 3-х. Рабочие характеристики этого варианта представлены в третьей строке табл. 6.1. Если от вас не требуется достичь минимально возможной задержки данных, то последний вариант представляется наилучшим компромиссом

Подобными соображениями можно руководствоваться и в том случае, когда устройство быстрого сдвига строится не на микросхемах средней степени интеграции, а из ячеек специализированной ИС, только вместо корпусов ИС средней и малой степени интеграции вам нужно будет подсчитывать площадь, занимаемую этими ячейками на поверхности кристалла.

Типичные библиотеки ячеек специализированных ИС содержат 1-разрядные мультиплексоры с числом входов от 2 до 8, обычно реализуемые на логических КМОП-ключах. Для построения мультиплексора бóльших размеров вам следует образовать необходимую комбинацию из ячеек меньших размеров Помимо тех проблем, с которыми мы встретились при выборе подходящего варианта в примере построения устройства на ИС средней степени интеграции, в данном случае возникает еще одна трудность: задержки в КМОП-схемах сильно зависят от нагрузки. Поэтому, в зависимости от выбранного подхода, мы должны решить, где в цепях управляющих сигналов или в цепях сигналов данных – или в тех и в других – необходимо включить буферы, чтобы минимизировать задержки, обусловленные нагрузкой. Может оказаться, что схема, которая хорошо выглядит на бумаге до анализа этих задержек и введения буферов, в действительности будет иметь бóльшую задержку или занимать бóльшую площадь кристалла, чем схема, являющаяся реализацией другого варианта.

6.1.2. Простой шифратор для получения чисел с плавающей точкой

В предыдущем примере многократно повторялась одна и та же ИС – мультиплексор. То, что именно мультиплексор окажется подходящим блоком, было довольно очевидно с самого начала. Следующий пример показывает, что иногда бывает необходимо более пристально взглянуть на постановку задачи, чтобы увидеть решение на основе известных стандартных блоков.

Давайте рассмотрим такую задачу, решение которой с использованием ИС средней степени интеграции не совсем очевидно Пусть нужно построить «шифратор, преобразующий числа с фиксированной точкой в числа с плавающей точкой». Целое двоичное число без знака B из интервала $0 \le B < 2^{11}$ можно представить 11 битами в формате «с фиксированной точкой»: $B = b_{10}b_9 \ . \ b_1b_0$ Для представления чисел из того же диапазона с меньшей точностью в системе обозначений с плавающей точкой достаточно 7 битов: $F = M \cdot 2^E$, где M – 4-разрядная двоичная мантисса $m_3m_2m_1m_0$, а E – 3-разрядный двоичный показатель экспоненты $e_2e_1e_0$. Наименьшее целое число, представимое в этом формате, равно $0 \cdot 2^0$, а наибольшее число равно $(2^4–1) \cdot 2^7$.

Преобразование заданного 11-разрядного двоичного числа B с фиксированной точкой в 7-разрядное число с плавающей точкой можно выполнить «беря» из числа B четыре бита, начиная с самого старшего бита, равного 1. Например:

$11010110100 = 1101 \cdot 2^7 + 0110100$
$00100101111 = 1001 \cdot 2^5 + 01111$
$00000111110 = 1111 \cdot 2^2 + 10$
$00000001011 = 1011 \ \ 2^0 + 0$
$00000000010 = 0010 \cdot 2^0 + 0.$

Последнее слагаемое в каждом равенстве справа представляет собой ошибку усечения в результате потери точности при преобразовании. Имея в виду такую процедуру преобразования, можно в следующем виде сформулировать требования, предъявляемые к устройству, преобразующему числа с фиксированной точкой в числа с плавающей точкой:

- комбинационная схема должна преобразовывать 11-разрядное двоичное целое число без знака B в 7-разрядное число с плавающей запятой M, E, где M и E являются 4-разрядным и 3-разрядным двоичными числами соответственно. Соотношение между числами имеет вид: $B = M \cdot 2^E + T$, где T – ошибка усечения, $0 \le T < 2^E$.

Чтобы рационально построить схему, начиная с подобной постановки задачи, требуется творческий подход: технические требования не содержат никакой подсказки. Некоторые идеи могут появиться в результате более детального изучения того, как мы преобразовываем числа вручную. По существу, мы просматриваем каждое входное число слева направо, чтобы найти первый разряд, содержащий 1, и останавливаемся на разряде b_3, если 1 не найдена. Мы выделяем четыре бита, начинающиеся с найденного разряда и используем их как мантиссу, а номером первого разряда определяется показатель экспоненты. Эти действия уже похожи на то, что выполняют стандартные блоки в виде ИС средней степени интеграции.

«Сканирование до первой 1» – это именно то, что делает универсальный приоритетный шифратор. На выходе приоритетного шифратора появляется число, говорящее нам о положении первой 1. Номером этого разряда определяется показатель экспоненты: когда первая единица находится в одном из разрядов b_{10}–b_3, это соответствует показателю экспоненты от 7 до 0; наличие первой единицы на позициях b_2–b_0 или полное отсутствие единиц означает, что показатель экспоненты равен 0. Поэтому для нахождения первой 1 можно воспользоваться 8-входовым приоритетным шифратором, на входы которого с I7 (высший приоритет) по I0 подаются биты b_{10}–b_3. Сигналы, появляющиеся на выходах A2–A0 приоритетного шифратора, можно непосредственно использовать в качестве показателя экспоненты; если 1 не найдена, то A2–A0 = 000.

«Взятие четырех битов» напоминает процедуру «выбора» при мультиплексировании. 3-разрядный показатель экспоненты определяет, какие четыре бита числа B мы выбираем, поэтому биты показателя экспоненты можно использовать в качестве управляющих сигналов 8-входового 4-разрядного мультиплексора, на выход которого проходят нужные четыре бита числа B, образующие мантиссу M.

Шифратор на основе ИС средней степени интеграции, реализующий эти идеи, показан на рис. 6.3. В схеме осуществлена некоторая оптимизация:

- Так как у имеющегося приоритетного шифратора 74x148 активным является низкий уровень входных сигналов, предполагается, что входное число B поступает по шине B_L [10:0] с низким активным уровнем сигналов. Если число B представлено сигналами с высоким активным уровнем, то можно воспользоваться восьмью инверторами для получения сигналов с низким активным уровнем.

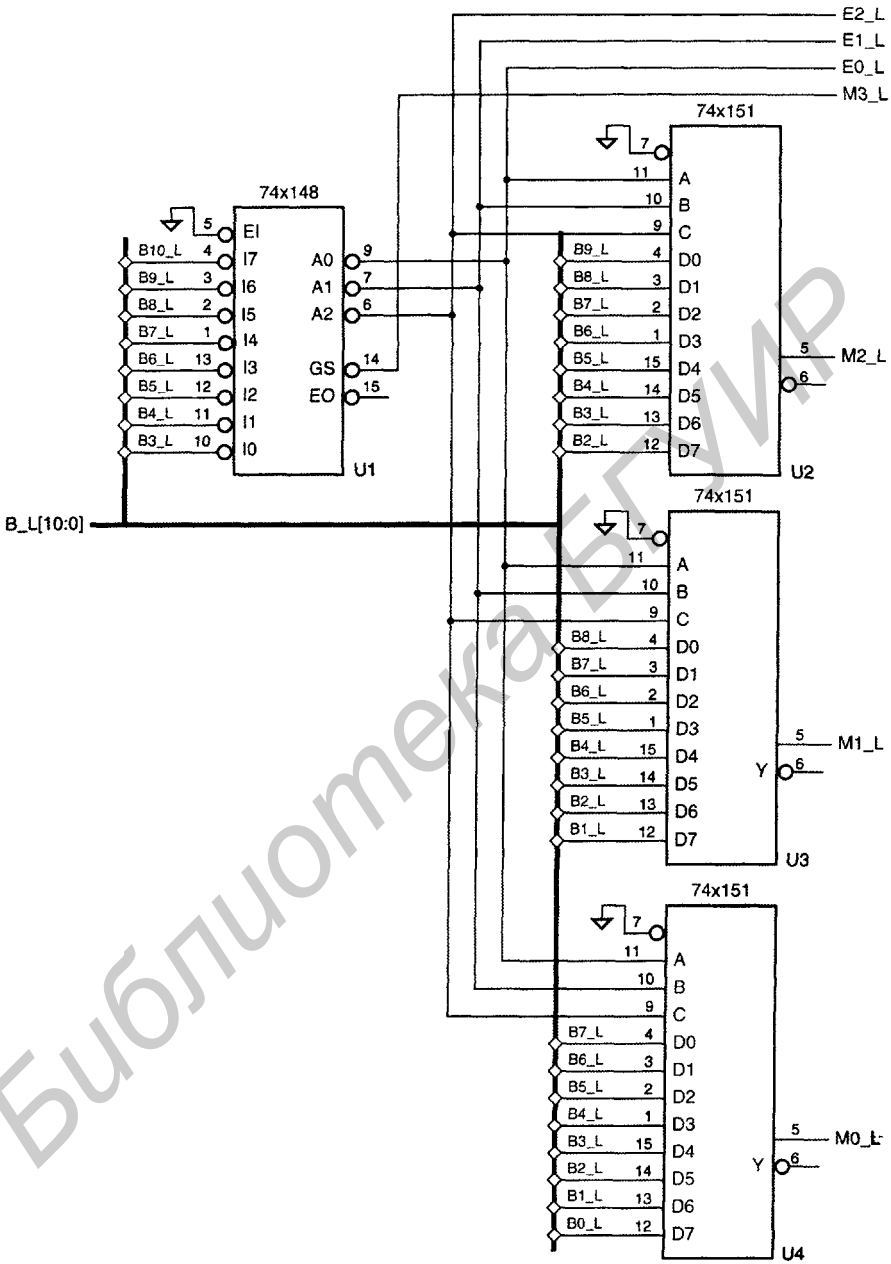


Рис. 6.3. Комбинационный шифратор, преобразующий числа с фиксированной точкой в числа с плавающей точкой

- Если вы внимательнее всмотритесь в процедуру преобразования, то поймете, что старший значащий бит мантиссы m_3 всегда равен 1, за исключением случая, когда 1 не найдена. У ИС '148 есть выход GS_L, на котором сигнал возникает как раз в этом случае, что позволяет нам получить сигнал m_3 без мультиплексора.
- Выходные сигналы ИС '148 имеют низкий активный уровень, поэтому биты показателя экспоненты E0_L–E2_L также представлены сигналами с низким активным уровнем. Естественно, что для получения сигналов с высоким активным уровнем можно было бы воспользоваться тремя инверторами.
- Поскольку все сигналы имеют низкий активный уровень, биты мантиссы также представлены сигналами с низким активным уровнем, но на выходе EO_L ИС '148 и на выходах Y_L ИС '151 имеются также значения битов мантиссы, представленные сигналами с высоким активным уровнем.

Строго говоря, мультиплексоры изображены на рис. 6.3 не вполне корректно. Как показано на рис. 6.4, возможно другое условное обозначение ИС 74x151. Словами это можно выразить так: если входные данные мультиплексора имеют низкий активный уровень, то активный уровень данных на выходах противоположен тому, который указан в исходном условном обозначении. На рис. 6.3 следует предпочесть обозначение «активного низкого уровня данных», так как только в этом случае активные уровни сигналов на входах и выходах ИС '151 будут согласованы с названиями сигналов на этих выводах. Однако при передаче данных и при их хранении разработчики (и автор данной книги тоже) не всегда следуют этому правилу. Обычно бывает ясно из контекста, что при прохождении через мультиплексор и при хранении в многоразрядном регистре (рассматриваемом в разделе 8.2.5) активный уровень данных не изменяется.

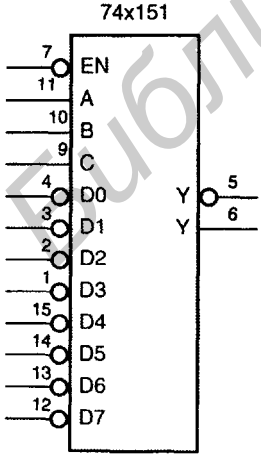


Рис. 6.4. Нетрадиционное условное обозначение 8-входового мультиплексора 74x151

6.1.3. Двойной приоритетный шифратор

Очень часто стандартные ИС средней степени интеграции не могут реализовать свои функции без помощи их «меньших братьев» – простых вентилей. В качестве следующего примера мы хотим построить приоритетный шифратор, который находит среди восьми сигналов запроса не только сигнал с наивысшим приоритетом, но также и сигнал со «вторым по старшинству приоритетом».

Предположим, например, что для входов запроса R_L [0:7] активным является низкий уровень сигнала, причем входу R_L0 принадлежит высший приоритет. Пусть сигналы A[2:0] и AVALID указывают на запрос с наивысшим приоритетом. Уровень сигнала AVALID активен только в том случае, когда присутствует хотя бы один запрос. Сигналы B[2:0] и BVALID пусть указывают на «второй по старшинству приоритета» запрос, причем уровень сигнала BVALID становится активным только тогда, когда имеются, по крайней мере, два запроса.

Обнаружить запрос с наивысшим приоритетом довольно просто; достаточно воспользоваться ИС 74x148. Другая ИС '148 позволяет находить запрос со «вторым по старшинству приоритетом», но только при условии, что сначала мы «исключаем» запрос с высшим приоритетом на входе этой микросхемы. Это можно сделать с помощью дешифратора, выбирающего сигнал, который надо исключить, на основе сигналов A[2:0] и AVALID, поступающих от первой ИС '148. Эти идеи воплощены в решении, приведенном на рис. 6.6. Активный уровень сигнала возникает не более чем на одном из восьми выходов дешифратора 74x138 – на том, который соответствует запросу с наивысшим приоритетом. Сигналы с выхода дешифратора поступают на входы вентилей И-НЕ, чтобы «исключить» запрос с наивысшим приоритетом.

Использованный прием позволяет получить на выходах ИС '148 сигналы с высоким активным уровнем, как это следует из рис. 6.5. Адресные выходы A_L [2:0] можно переименовать так, чтобы они имели высокий активный уровень сигналов, если изменим также имя входа запроса, относящегося к каждой выходной комбинации. В частности, мы инвертируем биты номера запроса. В видоизмененном условном обозначении высший приоритет имеет сигнал запроса на входе I0.

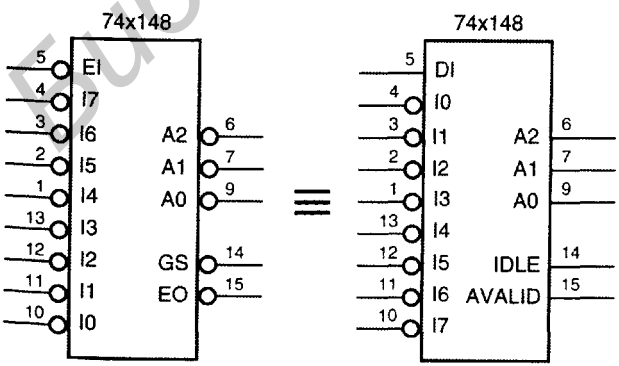


Рис. 6.5. Альтернативные условные обозначения 8-входового приоритетного шифратора 74x148

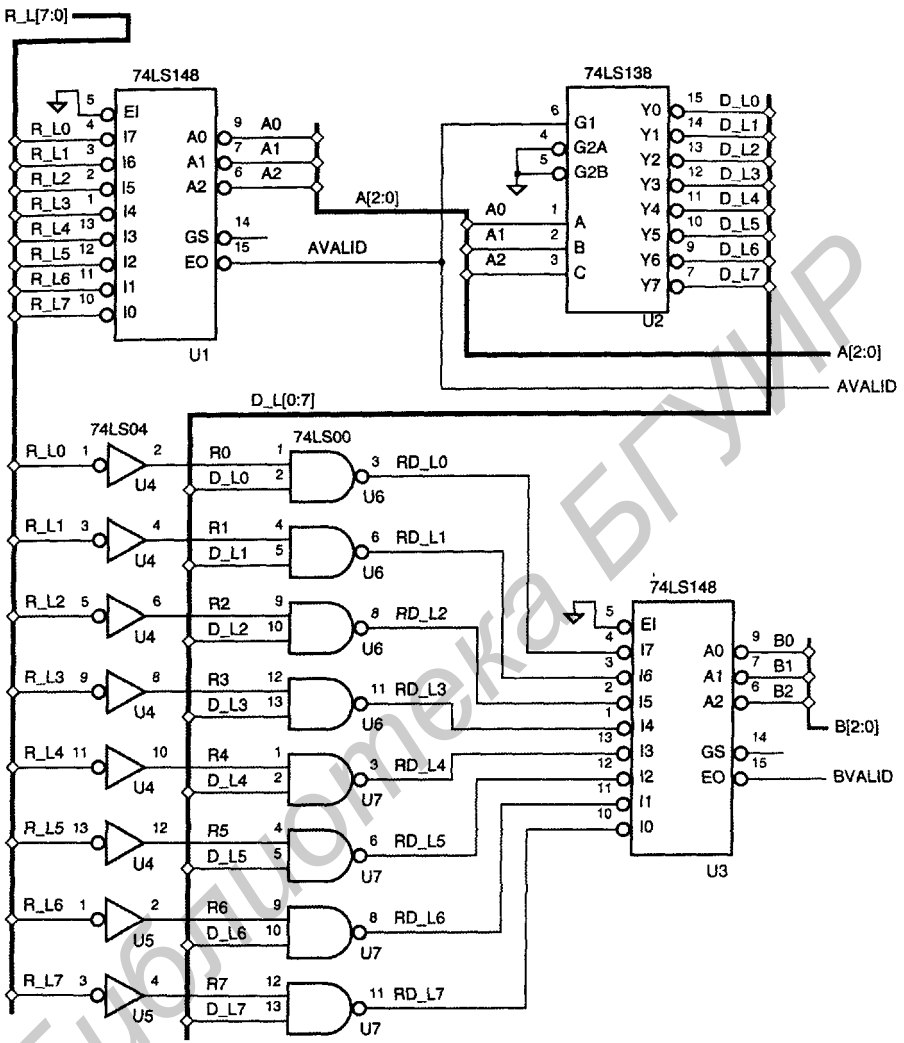


Рис. 6.6. Схема шифратора, обнаруживающего запросы с первым и вторым по старшинству приоритетами

6.1.4. Расширение компараторов

В разделе 5.9.4 мы показали, как можно строить большие компараторы путем каскадного включения 4-разрядных компараторов 74x85. Поскольку ИС 74x85 предусматривают их последовательное включение, на основе этих микросхем можно создавать сколь угодно большие компараторы. У 8-разрядного компаратора 74x682 *вообще нет никаких* входов и выходов для каскадного включения. Поэтому может показаться, на первый взгляд, что этой ИС нельзя воспользоваться для построения больших компараторов. Но это не так.

Если вы задумаетесь о сущности сравнения многоразрядных слов, то станет ясно, что два многоразрядных операнда – скажем, по 32 бита (по четыре байта) в каждом – равны только в том случае, когда равны их соответствующие байты. Если при сравнении необходимо принимать решение вида «больше чем» или «меньше чем», то результат сравнения определяется по самым старшим из неравных байтов.

Реализацией этих идей служит схема (рис. 6.7), позволяющая обнаруживать равенство двух 24-разрядных операндов или выносить решение вида «больше чем» с помощью трех 8-разрядных компараторов 74x682. Сравнение 24-разрядных операндов осуществляется на основе результатов сравнения отдельных 8-разрядных слов; выходные сигналы формируются дополнительной комбинационной схемой согласно следующим равенствам:

$$PEQQ = EQ2 \cdot EQ1 \cdot EQ0$$
$$PGTQ = GT2 + EQ2 \cdot GT1 + EQ2 \cdot EQ1 \cdot GT0.$$

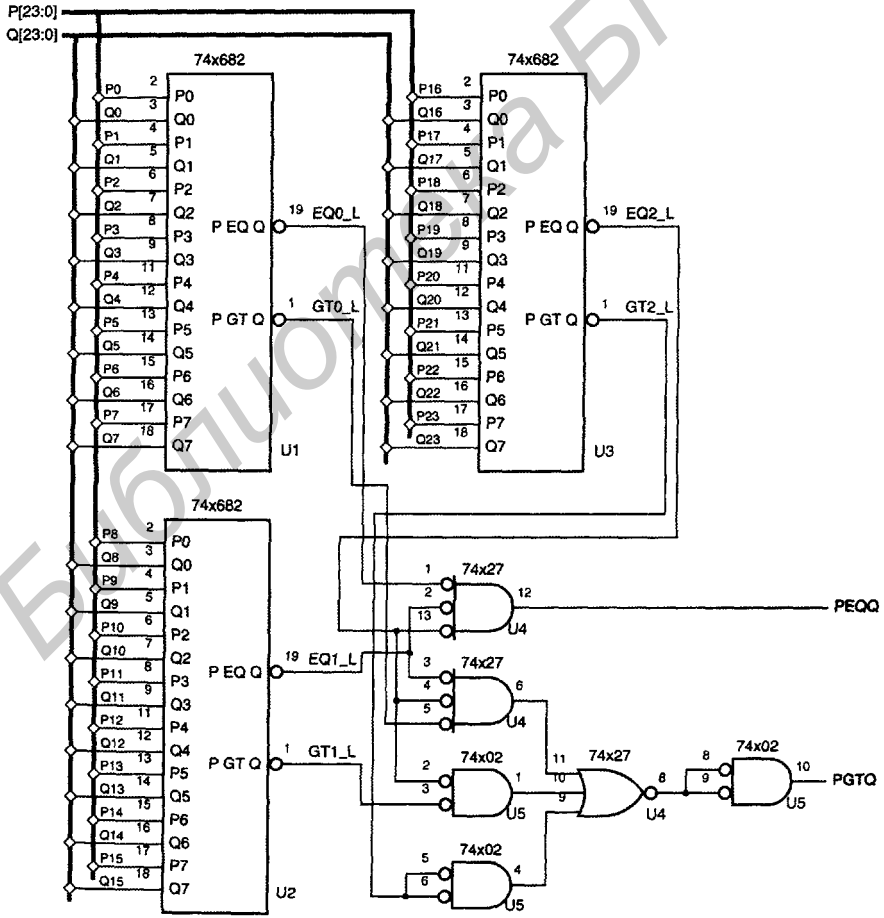


Рис. 6.7. Схема 24-разрядного компаратора

Такой «параллельный» подход к расширению компаратора в действительности дает большее быстродействие, нежели последовательное включение ИС 74x85, потому что исключается задержка распространения сигналов в каждом каскаде по пути от входов до выходов, с помощью которых компараторы соединяются один за другим. Параллельный подход и двухуровневая логика И–ИЛИ для объединения результатов сравнения 8-разрядных слов позволяют создавать компараторы с очень большим числом входов, которое ограничено только коэффициентом объединения по входу схем в логике И–ИЛИ. Применяя для объединения дополнительные логические схемы, можно строить сколь угодно большие компараторы.

6.1.5. Компаратор с управляемым режимом работы

Довольно часто требования, предъявляемые к цифровой схеме делают очевидным решение поставленной задачи на основе ИС средней степени интеграции или других стандартных блоков. Рассмотрим, например, такую задачу:

- Построить комбинационную схему, на входы которой подаются два 8-разрядных целых двоичных числа без знака X и Y и сигнал управления MIN/MAX, а на выходе возникает 8-разрядное целое двоичное число без знака Z, такое что Z = min(X,Y), если MIN/MAX = 1, и Z = max(X, Y), если MIN/MAX = 0.

Нетрудно представить себе эту схему составленной из ИС средней степени интеграции. Очевидно, что для определения, какое из чисел X или Y больше, можно воспользоваться компаратором. Сигнал с выхода компаратора может управлять мультиплексорами, на выходах которых будут вырабатываться сигналы min(X, Y) и max(X, Y), а с помощью другого мультиплексора можно выбрать один из этих результатов в зависимости от значения сигнала MIN/MAX. Блок-схема устройства, в котором реализован этот подход, приведена на рис. 6.8(а).

НЕ СЛЕДУЙТЕ СЛЕПО ЗА РЕКЛАМОЙ!

Расточительность исходного варианта, представленного на рис. 6.8(а), возможно, была очевидна для вас с самого начала, но он демонстрирует важный принцип проектирования на основе стандартных блоков:

- Для обработки данных используйте стандартные блоки и ищите способ заставить одни и те же блоки в разное время выполнять различные функции или работать в различных режимах. По мере надобности создавайте схемы управления для выбора соответствующих функций, чтобы уменьшить общее число компонентов в устройстве.

Как впечатляюще показано на рис. 6.8(с), этот подход позволяет сэкономить много корпусов. При проектировании на основе интегральных микросхем *не следует* поддаваться рекламе: «У нас есть все, что вам нужно, и даже больше»!

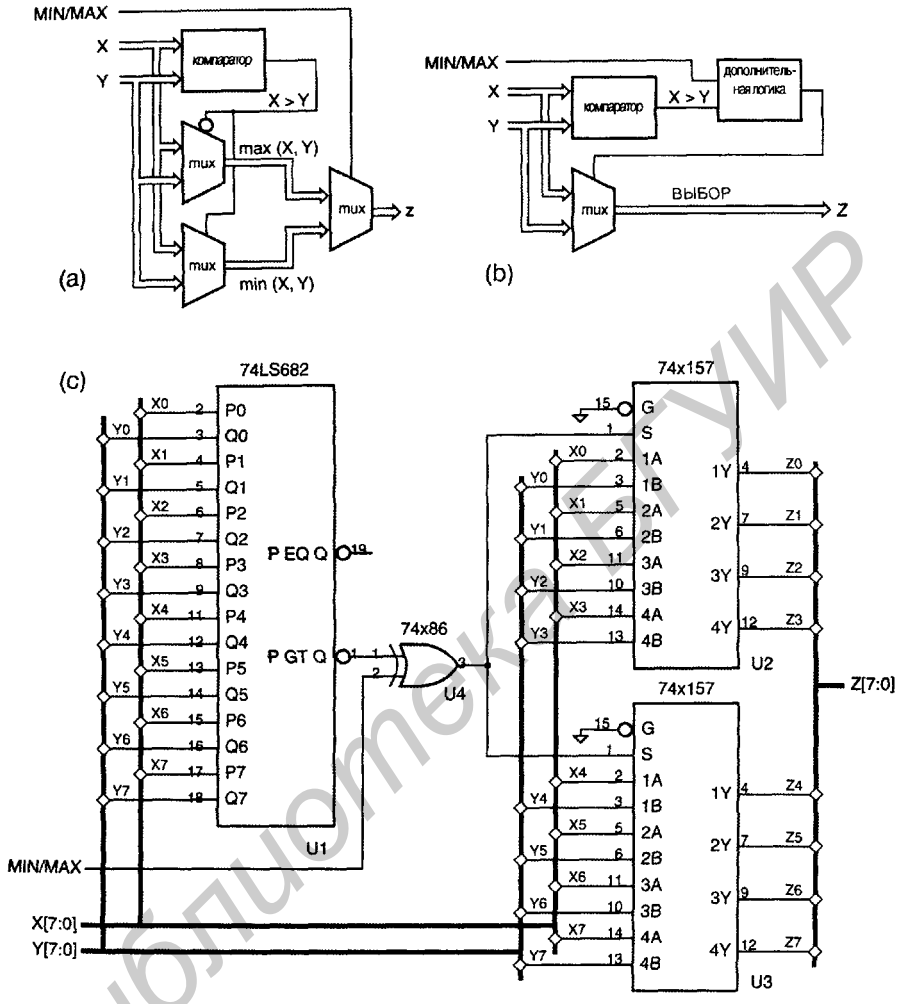


Рис. 6.8. Схема компаратора с управляемым режимом работы: (a) блок-схема первого приходящего на ум решения (mux – мультиплексор); (b) блок-схема более рационального решения с точки зрения стоимости; (c) принципиальная схема для варианта на рис. (b)

Наше первое решение достигает цели, но оно дороже, чем могло бы быть. Хотя схема содержит три двухвходовых мультиплексора, в конце концов нужно выбрать и пропустить на выход только одно из двух имеющихся входных слов X и Y. Поэтому задача состоит в том, чтобы построить схему, в которой решение о выборе одного из двух входных слов осуществлялось бы единственным двухвходовым мультиплексором и некоторой дополнительной логикой. Иллюстрацией такого подхода служат схемы на рис. 6.8(b) и (c). «Дополнительная логика» оказывается совсем простой: это всего лишь единственный вентиль ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ.

6.3. Примеры проектирования с использованием языка VHDL

6.3.1. Устройство быстрого сдвига

В разделе 6.1.1 устройство быстрого сдвига было определено как комбинационная логическая схема с *n* входами данных, *n* выходами данных и набором входных управляющих сигналов, которые задают величину сдвига выходных данных относительно входных. Там же было показано, как построить простое устройство быстрого сдвига, выполняющее только циклические сдвиги влево, используя стандартные микросхемы средней степени интеграции. Затем в разделе 6.2.1 было показано, как на языке ABEL описывается устройство быстрого сдвига, обладающее бóльшими возможностями, но там мы отметили также, что для реализации такого устройства ПЛУ обычно не подходит. В этом разделе мы покажем, как можно воспользоваться языком VHDL для описания как поведения, так и структуры устройств быстрого сдвига при их реализации на основе специализированных ИС или ИС типа FPGA.

В табл. 6.17 представлена поведенческая программа на языке VHDL для 16-разрядного устройства быстрого сдвига, которое выполняет сдвиг для любой из шести возможных комбинаций типа сдвига и направления. Как мы уже видели ранее (см. в табл. 6.3), сдвиги бывают циклическими, логическими и арифметическими, и сдвиг может производиться, естественно, влево или вправо. Как видно из объявления объекта, 4-разрядным входным сигналом управления `S` задается величина сдвига, а 3-разрядным входным сигналом управления `C` определяется режим сдвига (тип и направление). Используя пакет IEEE `std_logic_arith`, мы определяем тип величины сдвига `S` как `UNSIGNED` для того, чтобы позже можно было воспользоваться функцией `CONV_INTEGER` из этого пакета.

Обратите внимание, что объявление объекта включает шесть определений постоянных, которыми устанавливается соответствие между режимами сдвига и значением `C`. Хотя мы не обсуждали это в разделе 4.7, но язык VHDL позволяет помещать константу, тип, сигнал и другие объявления в объявление объекта. Определение таких элементов в объявлении объекта имеет смысл только в том случае, когда они должны быть одними и теми же в любой архитектуре. В нашем случае за режимами сдвига закрепляются вполне определенные двоичные коды, поэтому здесь им самое место. Другим элементам предстоит появиться в определении архитектуры.

В части программы, относящейся к архитектуре, мы определяем шесть функций, по одной для каждого вида сдвига 16-разрядного элемента типа `STD_LOGIC_VECTOR`. В архитектуре определен также подтип `DATAWORD`, чтобы сэкономить на его объявлении в определениях функций.

Табл. 6.17. *Поведенческое VHDL-описание устройства быстрого сдвига, выполняющего 6 видов сдвига*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity barrel16 is
    port (
        DIN: in STD_LOGIC_VECTOR (15 downto 0);  -- Data inputs
        S: in UNSIGNED (3 downto 0);             -- Shift amount, 0-15
        C: in STD_LOGIC_VECTOR (2 downto 0);     -- Mode control
        DOUT: out STD_LOGIC_VECTOR (15 downto 0) -- Data bus output
    );
    constant Lrotate:  STD_LOGIC_VECTOR := "000";   -- Define the coding of
    constant Rrotate:  STD_LOGIC_VECTOR := "001";   -- the different shift modes
    constant Llogical: STD_LOGIC_VECTOR := "010";
    constant Rlogical: STD_LOGIC_VECTOR := "011";
    constant Larith:   STD_LOGIC_VECTOR := "100";
    constant Rarith:   STD_LOGIC_VECTOR := "101";
end barrel16;

architecture barrel16_behavioral of barrel16 is
subtype DATAWORD is STD_LOGIC_VECTOR(15 downto 0);

function Vrol (D: DATAWORD; S: UNSIGNED)
     return DATAWORD is
  variable N: INTEGER;
  variable TMPD: DATAWORD;
  begin
    N := CONV_INTEGER(S); TMPD := D;
    for i in 1 to N loop
      TMPD := TMPD(14 downto 0) & TMPD(15);
    end loop;
    return TMPD;
  end Vrol;
...
begin
process(DIN, S, C)
  begin
    case C is
      when Lrotate  => DOUT <= Vrol(DIN,S);
      when Rrotate  => DOUT <= Vror(DIN,S);
      when Llogical => DOUT <= Vsll(DIN,S);
      when Rlogical => DOUT <= Vsrl(DIN,S);
      when Larith   => DOUT <= Vsla(DIN,S);
      when Rarith   => DOUT <= Vsra(DIN,S);
      when others   => null;
    end case;
  end process;
end barrel16_behavioral;
```

ВАШИ СОБСТВЕННЫЕ СДВИГИ

В действительности у языка VHDL-93 есть встроенные операторы свига `rol`, `ror`, `sll`, `srl`, `sla` и `sra` элементов типа `array`, соответствующие операциям сдвига, перечисленным в табл. 6.3. Так как этих операторов нет в языке VHDL-87, мы определили в табл. 6.17 свои собственные функции. На самом деле, в таблице приведена только одна из них (`Vrol`); определение остальных функций оставлено читателю в качестве задачи (задача 6.11).

В табл. 6.17 полностью приведена только первая функция (`Vrol`); остальные подобны ей, за исключением изменения в одной строке. Используемая в цикле `for` переменная `N` является результатом преобразования величины сдвига `S` в целое число. Кроме того, мы присваиваем значение входного вектора `D` локальной переменной `TMPD`, которая в цикле `for` сдвигается `N` раз. Тело цикла `for` образует единственный оператор присваивания. В нем берется 15-битовый отрезок слова данных [`TMPD(14downto0)`] и осуществляется конкатенация [`&`]; результат возвращается в `TMPD` вместе с битом [`TMPD(15)`], который «выдвинулся» с левого края. Подобными действиями можно описать и другие типы сдвига. Заметьте, что функции сдвига нельзя было бы определить в другом, неповеденческом описании объекта `barrel16`, например, в структурной архитектуре.

Часть «параллельных операторов» в этой архитектуре исчерпывается единственным процессом, список чувствительности которого составляют все входы объекта. Оператор `case` этого процесса присваивает результат выходу `DOUT`, вызывая соответствующую функцию в зависимости от значения сигнала `C` на входе выбора режима.

Процесс, приведенный в табл. 6.17, является хорошим поведенческим описанием устройства быстрого сдвига, но многие средства синтеза не смогут синтезировать схему по такому описанию. Проблема заключается в том, что большинству программных средств требуется, чтобы диапазон цикла `for` был статическим на момент компиляции, тогда как у цикла `for` в функции `Vrol` диапазон динамический: он зависит от значения входного сигнала `S` во время работы схемы.

Ну, действительно трудно представить себе, какую схему могла бы выдать программа синтеза даже в том случае, если бы она была способна обрабатывать циклы `for` с динамическим диапазоном. Это пример того, когда разработчику следует хотя бы немного поруководить средствами синтеза при выборе структуры схемы, если есть желание получить достаточно быстрый и эффективный результат.

На рис. 6.2 было показано, как может выглядеть 16-разрядное устройство быстрого сдвига, выполняющее циклические сдвиги влево, собранное из стандартных ИС средней степени интеграции. В этой схеме последовательно один за другим включены четыре 16-разрядных 2-входовых мультиплексора, осуществляющих сдвиг входных данных на 0 разрядов или на 1, 2, 4 и 8 разрядов в зависимости от значений сигналов S0–S3 соответственно. Подобный характер поведения и структуру такого вида можно описать средствами языка VHDL так, как это сделано в программе, приведенной в табл. 6.18. Несмотря на то, что в программе используется процесс и она написана в «поведенческом» стиле, можно быть более или менее уверенным в том, что для каждого оператора "`if`" в большинстве случаев синтезатор создаст по 2-входовому мультиплексору, которые образуют последовательную цепочку, подобную приведенной на рис. 6.2.

Табл. 6.18. Программа на языке VHDL для 16-разрядного устройства быстрого сдвига, осуществляющего только циклический сдвиг влево

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rol16 is
    port (
        DIN: in STD_LOGIC_VECTOR(15 downto 0),   -- Data inputs
        S: in STD_LOGIC_VECTOR (3 downto 0),     -- Shift amount, 0 15
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );
end rol16;

architecture rol16_arch of rol16 is
begin
process(DIN, S)
  variable X, Y, Z  STD_LOGIC_VECTOR(15 downto 0);
  begin
    if S(0)='1' then X  = DIN(14 downto 0) & DIN(15); else X := DIN; end if;
    if S(1)='1' then Y := X(13 downto 0) & X(15 downto 14); else Y  = X; end if;
    if S(2)='1' then Z := Y(11 downto 0) & Y(15 downto 12); else Z  = Y; end if;
    if S(3)='1' then DOUT <= Z(7 downto 0) & Z(15 downto 8); else DOUT <= Z; end if;
  end process;
end rol16_arch;
```

Но в решаемой здесь задаче требуется, чтобы устройство быстрого сдвига могло выполнять сдвиг и влево, и вправо. В табл. 6.19 представлена исправленная предыдущая программа, способная выполнять циклические сдвиги в любом направлении. Направление сдвига задается дополнительным входным сигналом `DIR`: 0 – для сдвига влево, 1 – для сдвига вправо. В каждом из звеньев, образующих последовательную цепочку, характер сдвига определяется оператором `case`, выбирающим одну из четырех возможностей по значению сигнала `DIR` и того бита `S`, который управляет этим звеном. Обратите внимание, что мы ввели локальные 2-разрядные переменные `CTRL1` для хранения пары значений `DIR` и `S(1)`; каждый оператор `case` управляется одной из этих переменных. У вас может появиться желание исключить эти переменные и просто управлять каждым оператором `case` с помощью конкатенации `DIR & S(1)`, но синтаксис языка VHDL не позволяет это сделать, потому что тип конкатенации был бы неизвестен.

Типичный синтезатор языка VHDL создаст 3- или 4-входовой мультиплексор для каждого оператора `case` в табл. 6.19. Для последнего оператора `case` хороший синтезатор сформирует только 2-входовой мультиплексор.

Итак, теперь у нас есть устройство быстрого сдвига, которое будет выполнять циклические сдвиги влево или вправо, но мы сделали еще не все: необходимо позаботиться о логических и арифметических сдвигах в обоих направлениях. На рис. 6.14 представлен наш план завершения проекта. Согласно этому плану наше устройство начинается с блока `ROLR16`, разработку которого мы только что завершили, а для управления направлением сдвига, в зависимости от сигнала `C`, используется дополнительная логика.

Теперь необходимо «скорректировать» некоторые из полученных битов, если выполняется логический или арифметический сдвиг. При логическом или арифметическом сдвиге на n разрядов влево мы должны присвоить правым $n-1$ би-

там значение 0 или первоначальное значение бита, находившегося в крайнем правом разряде соответственно. Для логического или арифметического сдвига на n разрядов вправо мы должны присвоить левым $n-1$ битам значение 0 или первоначальное значение бита, находившегося в крайнем левом разряде соответственно.

Табл. 6.19. Программа на языке VHDL для 16-разрядного устройства быстрого сдвига, выполняющего циклические сдвиги влево и вправо

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rolr16 is
    port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0);  -- Data inputs
        S:    in STD_LOGIC_VECTOR (3 downto 0);  -- Shift amount, 0-15
        DIR:  in STD_LOGIC;                      -- Shift direction, 0=>L, 1=>R
        DOUT: out STD_LOGIC_VECTOR(15 downto 0)  -- Data bus output
    );
end rolr16;

architecture rol16r_arch of rolr16 is
begin
process(DIN, S, DIR)
  variable X, Y, Z: STD_LOGIC_VECTOR(15 downto 0);
  variable CTRL0, CTRL1, CTRL2, CTRL3: STD_LOGIC_VECTOR(1 downto 0);
  begin
    CTRL0 := S(0) & DIR; CTRL1 := S(1) & DIR; CTRL2 := S(2) & DIR; CTRL3 := S(3) & DIR;
    case CTRL0 is
      when "00" | "01" => X := DIN;
      when "10" => X := DIN(14 downto 0) & DIN(15);
      when "11" => X := DIN(0) & DIN(15 downto 1);
      when others => null;  end case;
    case CTRL1 is
      when "00" | "01" => Y := X;
      when "10" => Y := X(13 downto 0) & X(15 downto 14);
      when "11" => Y := X(1 downto 0) & X(15 downto 2);
      when others => null;  end case;
    case CTRL2 is
      when "00" | "01" => Z := Y;
      when "10" => Z := Y(11 downto 0) & Y(15 downto 12);
      when "11" => Z := Y(3 downto 0) & Y(15 downto 4);
      when others => null;  end case;
    case CTRL3 is
      when "00" | "01" => DOUT <= Z;
      when "10" | "11" => DOUT <= Z(7 downto 0) & Z(15 downto 8);
      when others => null;  end case;
  end process;
end rol16r_arch;
```

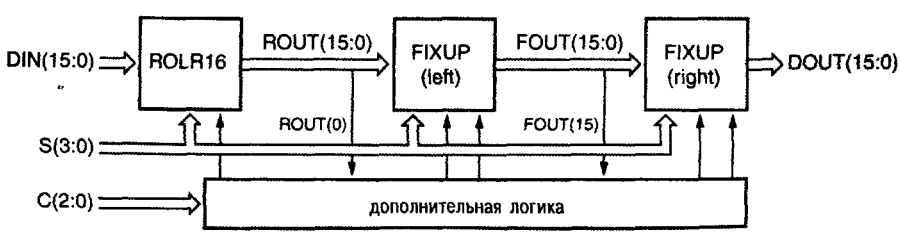


Рис. 6.14. Блок-схема устройства быстрого сдвига

Как показано на рис. 6.14, наша стратегия состоит в том, чтобы вслед за устройством ROLR16, осуществляющим циклический сдвиг, включить схему коррекции FIXUP(left), которая заполняет нужные младшие разряды при логическом или арифметическом сдвиге влево, и схему коррекции FIXUP(right), которая заполняет нужные старшие разряды при логическом или арифметическом сдвиге вправо.

В табл. 6.20 приведена поведенческая программа на языке VHDL для схемы коррекции при сдвиге влево. У нее имеются 16-разрядные входы и выходы данных DIN и DOUT соответственно. На входы управления поступают сигналы S, которыми задается величина сдвига, сигнал разрешения коррекции FEN и новая величина FDAT для вставления в корректируемые разряды выходных данных. Схема помещает значение корректирующего бита в каждый разряд данных на выходе DOUT(i), если i меньше S и коррекция разрешена; в противном случае на выходы схемы передаются значения битов на входе DIN(i) без изменений.

Табл. 6.20. *Поведенческая программа на языке VHDL для схемы коррекции при сдвиге влево*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity fixup is
    port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0); -- Data inputs
        S:    in UNSIGNED(3 downto 0);          -- Shift amount, 0-15
        FEN:  in STD_LOGIC;                     -- Fixup enable
        FDAT: in STD_LOGIC;                     -- Fixup data
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );
end fixup;

architecture fixup_arch of fixup is
begin
process(DIN, S, FEN, FDAT)
  begin
    for i in 0 to 15 loop
      if (i < CONV_INTEGER(S) and (FEN = '1') then DOUT(i) <= FDAT;
      else DOUT(i) <= DIN(i); end if;
    end loop;
  end process;
end fixup_arch;
```

Блок, соответствующий циклу for в табл. 6.20, синтезировать легко, но заранее нельзя быть уверенным, что его логика будет вполне подходящей. В частности, наличие операции "<", выполняемой на каждом проходе цикла, может заставить синтезатор включить универсальный компаратор, сравнивающий значения величин, несмотря на то, что один из операндов является константой, и поэтому сигнал на каждом выходе можно было бы получить с помощью небольшого числа вентилей. (Например, реализация логики "7 < CONV_INTEGER(S)" состоит всего лишь в подведении провода S(3)!) В рассуждении, вынесенном за пределы основного текста и озаглавленном «Последовательная структура схемы коррекции», говорится о структурном варианте этой функции.

ПОСЛЕДОВАТЕЛЬНАЯ СТРУКТУРА СХЕМЫ КОРРЕКЦИИ

Структурная архитектура схемы коррекции приведена в табл. 6.21. По существу, здесь определена итерационная схема, вырабатывающая 16-разрядный вектор FSEL с равным 1 значением FSEL(i), если i-й бит нуждается в коррекции. Процедура начинается с того, что биту FSEL(15) присваивается значение 0, поскольку в этом разряде никогда не требуется коррекция. Для остальных значений i величина FSEL(i) должна равняться 1, если значение S равно i+1, и в том случае, когда биту FSEL(i+1) уже присвоено единичное значение. Таким образом, присваивание значений битам FSEL оператором generate создает последовательную цепочку 2-входовых вентилей ИЛИ: один из входов каждого вентиля ИЛИ предназначен для подачи 1, когда S=i (что обнаруживается по результату декодирования 4-входовым вентилем И), а другой вход каждого следующего вентиля ИЛИ соединен с выходом предыдущего вентиля ИЛИ. Оператор присваивания DOUT(i) создает 16 двухвходовых мультиплексоров, каждый из которых выбирает бит входных данных DIN(i) или корректирующий бит (FDAT) в зависимости от значения FSEL(i).

Реализация схемы коррекции в виде последовательной цепочки оказывается компактной, но очень медленной по сравнению со схемой, в которой сигнал на каждом выходе FSEL вырабатывается двухуровневой логикой по выражениям вида «сумма произведений». Однако в данном случае большая задержка не имеет значения, поскольку схема коррекции располагается в конце пути прохождения данных. Если все же быстродействие существенно, то можно воспользоваться «бесплатным» приемом, который позволяет уменьшить задержку вдвое (см. задачу 6.12).

При сдвиге вправо коррекция начинается с противоположной стороны слова данных, поэтому может показаться, что необходима еще одна схема коррекции. Однако, как мы скоро увидим, для этого можно воспользоваться уже имеющейся схемой, если только изменить порядок следования входных и выходных битов на противоположный.

Табл. 6.21. Структурная VHDL-архитектура схемы коррекции при сдвиге влево

```
architecture fixup_struc of fixup is
signal FSEL: STD_LOGIC_VECTOR(15 downto 0);       -- Fixup select
begin
  FSEL(15) <= '0'; DOUT(15) <= DIN(15);
  U1: for i in 14 downto 0 generate
    FSEL(i) <= '1' when CONV_INTEGER(S) = i+1 else FSEL(i+1);
    DOUT(i) <= FDAT when (FSEL(i) = '1' and FEN = '1') else DIN(i);
  end generate;
end fixup_struc;
```

В табл. 6.22 приведена объединенная структурная архитектура полного 16-разрядного устройства быстрого сдвига с 6-ю режимами работы, в котором реализован подход, изображенный на рис. 6.14. Объявление объекта barrel16

оставлено таким же, как и в табл. 6.17. В архитектуре объявлены два компонента: `rolr16` и `fixup`, для которых используются наши предыдущие определения объектов. Обращение к этим компонентам происходит в части программы, содержащей исполняемые операторы. Там же имеется несколько операторов присваивания, которые вырабатывают необходимые управляющие сигналы (то есть реализуют «дополнительную логику», указанную на рис. 6.14).

Табл. 6.22. Структурная VHDL-архитектура устройства быстрого сдвига с 6-ю режимами работы

```
architecture barrel16_struc of barrel16 is

component rolr16 port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0); -- Data inputs
        S:    in UNSIGNED(3 downto 0),          -- Shift amount, 0-15
        DIR:  in STD_LOGIC;                     -- Shift direction, 0=>L, 1=>R
        DOUT  out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    ),  end component,

component fixup port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0); -- Data inputs
        S:    in UNSIGNED(3 downto 0);          -- Shift amount, 0-15
        FEN.  in STD_LOGIC;                     -- Fixup enable
        FDAT  in STD_LOGIC,                     - Fixup data
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );  end component;

signal DIR_RIGHT, FIX_RIGHT, FIX_RIGHT_DAT, FIX_LEFT, FIX_LEFT_DAT  STD_LOGIC;
signal ROUT, FOUT, RFIXIN, RFIXOUT. STD_LOGIC_VECTOR(15 downto 0),

begin
  DIR_RIGHT <= '1' when C = Rrotate or C = Rlogical or C = Rarith else '0',
  FIX_LEFT <= '1' when DIR_RIGHT='0' and (C = Llogical or C = Larith) else '0',
  FIX_RIGHT <- '1' when DIR_RIGHT='1' and (C = Rlogical or C = Rarith) else '0',
  FIX_LEFT_DAT <= DIN(0) when C = Larith else '0';
  FIX_RIGHT_DAT <= DIN(15) when C = Rarith else '0';
  U1: rolr16 port map (DIN, S, DIR_RIGHT, ROUT),
  U2: fixup port map (ROUT, S, FIX_LEFT, FIX_LEFT_DAT, FOUT);
  U3: for i in 0 to 15 generate RFIXIN(i) <= FOUT(15-i); end generate,
  U4: fixup port map (RFIXIN, S, FIX_RIGHT, FIX_RIGHT_DAT, RFIXOUT),
  U5: for i in 0 to 15 generate DOUT(i) <= RFIXOUT(15-i), end generate,
end barrel16_struc,
```

Например, первый оператор присваивания устанавливает единичное значение сигнала `DIR_RIGHT`, когда значением `C` задается один из сдвигов вправо. При логических и арифметических сдвигах влево и вправо вырабатываются сигналы разрешения для схем коррекции `FIX_LEFT` и `FIX_RIGHT`. Значениям корректирующих битов присвоены имена `FIX_LEFT_DAT` и `FIX_RIGHT_DAT`.

Хотя все операторы в этой архитектуре выполняются одновременно, для удобства чтения они перечислены в табл. 6.22 в порядке фактического потока данных. Сначала вызывается компонент `rolr16` для выполнения основного циклического сдвига влево или вправо. Результат этого сдвига подается на вход первого компонента `fixup` (`U2`) для осуществления коррекции битов при логическом и арифметическом сдвигах влево. Затем следует оператор `generate` (`U3`), который изме-

СТИЛЬ УПРЯТЫВАНИЯ ИНФОРМАЦИИ

Зная, как кодируется управляющий сигнал C, вы, возможно, захотите написать первый оператор присваивания в табл. 6.22 в виде `DIR_RIGHT<=C(0)`, что гарантировало бы более эффективную реализацию схемы, которая вырабатывает этот управляющий сигнал: схема состояла бы всего лишь из одного соединения! Но при этом нарушился бы программистский стиль упрятывания информации, и это могло бы привести к появлению конструктивных недостатков.

Мы в явном виде записали коды сдвигов в объявлении объекта `barrel16` посредством определения констант. Архитектуре не нужно знать детали кодирования. Предположим, однако, что в нашей архитектуре произведена все же предложенная выше замена. Если бы кто-то другой (или мы сами!) захотел позднее изменить определения `constant` в объявлении объекта `barrel16`, задавая коды сдвигов иначе, то при новом способе кодирования уже нельзя было бы воспользоваться данной архитектурой! В задаче 6.13 требуется так изменить определения, чтобы объявление объекта позволяло уменьшить стоимость проекта, осуществив предложенную нами замену.

няет порядок следования битов данных для следующего обращения к компоненту `fixup` (`U4`), производящему коррекцию при логическом и арифметическом сдвигах вправо. Наконец, другой оператор `generate` (`U5`) возвращает прежний порядок следования битов, измененный оператором `U3`. Заметьте, что исполнение `U3` и `U5` заключается в простом изменении порядка соединений.

Для исходного объекта `barrel16` можно написать много других архитектур. В задаче 6.14 мы предлагаем архитектуру, которая позволяет выполнять циклический сдвиг с помощью объекта `rol16`, использующего только 2-входовые мультиплексоры, а не с помощью более дорогого объекта `rolr16`.

6.3.2. Простой шифратор для получения чисел с плавающей точкой

В разделе 6.1.2 мы определили простой формат числа с плавающей точкой и изложили задачу проектирования преобразователя числа с фиксированной точкой в число с плавающей точкой. Задача нахождения показателя экспоненты числа с плавающей точкой легко решается с помощью приоритетного шифратора, выполненного в виде ИС средней степени интеграции. При программировании на любом из языков описания схем решение той же самой задачи отображается в виде вложенных операторов "`if`". В табл. 6.23 приведена поведенческая VHDL-программа шифратора для получения чисел с плавающей точкой. В пределах архитектуры `fpenc_arch` с помощью вложенного оператора "`if`" проверяется величина входной переменной `B` и устанавливаются соответствующие значения `M` и `E`. Обратите внимание, что в программе используется пакет IEEE `std_logic_arith`; это сделано для того, чтобы у нас были тип `UNSIGNED` и операции сравнения, которые сопровождают его, как было объяснено в разделе 5.9.6. Ради представления программы в компактном виде введена переменная `BU`, выражающая значение переменной `B` в формате типа `UNSIGNED`; в принципе, в каждом вложенном операторе "`if`" вместо `BU` можно написать `UNSIGNED(B)`.

Табл. 6.23. Поведенческая VHDL-программа для преобразования чисел с фиксированной точкой в числа с плавающей точкой

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity fpenc is
  port (
    B: in STD_LOGIC_VECTOR(10 downto 0); -- fixed-point number
    M: out STD_LOGIC_VECTOR(3 downto 0); -- floating-point mantissa
    E: out STD_LOGIC_VECTOR(2 downto 0)  -- floating-point exponent
  );
end fpenc;

architecture fpenc_arch of fpenc is
begin
  process(B)
  variable BU: UNSIGNED(10 downto 0);
  begin
    BU := UNSIGNED(B);
    if    BU < 16   then M <= B( 3 downto 0); E <= "000";
    elsif BU < 32   then M <= B( 4 downto 1); E <= "001";
    elsif BU < 64   then M <= B( 5 downto 2); E <= "010";
    elsif BU < 128  then M <= B( 6 downto 3); E <= "011";
    elsif BU < 256  then M <= B( 7 downto 4); E <= "100";
    elsif BU < 512  then M <= B( 8 downto 5); E <= "101";
    elsif BU < 1024 then M <= B( 9 downto 6); E <= "110";
    else                 M <= B(10 downto 7); E <= "111";
    end if;
  end process;
end fpenc_arch;
```

ПЕРЕМЕННАЯ "B" НЕ МОЕГО ТИПА

В табл. 6.23 мы использовали выражение `UNSIGNED(B)` для преобразования переменной `B`; массив типа `STD_LOGIC_VECTOR` преобразуется в массив типа `UNSIGNED`. Эта операция называется *явным преобразованием типов*. Язык VHDL позволяет преобразовывать тесно связанные между собой типы, записывая желаемый тип, за которым в круглых скобках следует преобразуемая величина. Два типа массивов считаются «тесно связанными», если у них один и тот же тип элементов, одна и та же размерность и одинаковые типы индексов (обычно `INTEGER`), а также те массивы, тип которых можно преобразовать. Элементы старого массива размещаются в новом массиве на соответствующих позициях в том же порядке слева направо.

Хотя программа, текст которой приведен в табл. 6.23, полностью синтезируема, некоторые программные средства синтеза могут оказаться не настолько толковыми, чтобы распознать, что во вложенных сравнениях на каждом уровне нужно проверять только один бит. Вместо этого такие программные средства могут для каждого уровня создать полный 11-разрядный компаратор. Такая логическая схема была бы намного больше и работала бы медленнее, чем то, что можно было бы сделать. Когда мы сталкиваемся с такой проблемой, всегда можно записать архитектуру немного иначе и в более явном виде, чтобы помочь программе выйти из затруднения, как это сделано в табл. 6.24

Табл. 6.24. Другой вариант VHDL-архитектуры для преобразования чисел с фиксированной точкой в числа с плавающей точкой

```
architecture fpence_arch of fpenc is
begin
  process(B)
  begin
    if   B(10) = '1' then M <= B(10 downto 7); E <= "111";
    elsif B(9) = '1' then M <= B( 9 downto 6); E <= "110";
    elsif B(8) = '1' then M <= B( 8 downto 5); E <= "101";
    elsif B(7) = '1' then M <= B( 7 downto 4); E <= "100";
    elsif B(6) = '1' then M <= B( 6 downto 3); E <= "011";
    elsif B(5) = '1' then M <= B( 5 downto 2); E <= "010";
    elsif B(4) = '1' then M <= B( 4 downto 1); E <= "001";
    else                  M <= B( 3 downto 0); E <= "000";
    end if;
  end process;
end fpence_arch;
```

С другой стороны, для увеличения функциональных возможностей нашего устройства может появиться желание воспользоваться реальными компараторами и затратить даже большее число вентилей. В частности, наш вариант устройства при нахождении битов мантиссы выполняет усечение, а не округление. Более точный результат достигается при округлении, но это приводит к гораздо более сложному устройству. Во-первых, чтобы прибавить 1 к выбранным битам мантиссы при округлении вверх, необходим сумматор. Однако добавление 1, когда мантисса уже равна 1111, выталкивает нас в диапазон, представляемый следующим значением показателя экспоненты, так что надо быть готовым к этому случаю. Наконец, никогда нельзя выполнить округление вверх, если до округления мантисса и показатель экспоненты равны 1111 и 111, поскольку в нашем представлении числа с плавающей точкой отсутствует большее значение числа, до которого следует округлять.

Программа, приведенная в табл. 6.25, выполняет желаемое округление. Функция `round` берет биты из 5 определенных разрядов числа с фиксированной точкой и возвращает в качестве результата четыре старших разряда из них с добавленной к ним 1, если младший разряд равен 1. Таким образом, если считать, что непосредственно слева от младшего разряда находится двоичная точка, то

округление происходит в том случае, когда отбрасываемая часть мантиссы равна 1/2 или больше. В каждом предложении вложенного оператора "`if`" в процессе выполняется сравнение, чтобы при округлении вверх выбранной величины не происходило «переполнения», которое переводило бы результат в диапазон чисел, представляемых следующим значением показателя экспоненты. В противном случае преобразование и округление происходят в следующем предложении. Последним предложением гарантируется, что не произойдет округления вверх, когда мы находимся на краю диапазона чисел, представимых в формате с плавающей точкой.

Табл. 6.25. Поведенческая VHDL-архитектура для преобразования числа с фиксированной точкой в число с плавающей точкой с округлением

```
architecture fpencr_arch of fpenc is
function round (BSLICE: STD_LOGIC_VECTOR(4 downto 0))
    return STD_LOGIC_VECTOR is
  variable BSU: UNSIGNED(3 downto 0);
  begin
    if BSLICE(0) = '0' then return BSLICE(4 downto 1);
    else null;
      BSU := UNSIGNED(BSLICE(4 downto 1)) + 1;
      return STD_LOGIC_VECTOR(BSU);
    end if;
  end;
begin
  process(B)
  variable BU: UNSIGNED(10 downto 0);
  begin
    BU := UNSIGNED(B);
    if    BU < 16    then M <= B( 3 downto 0); E <= "000";
    elsif BU < 32-1  then M <= round(B( 4 downto 0)); E <= "001";
    elsif BU < 64-2  then M <= round(B( 5 downto 1)); E <= "010";
    elsif BU < 128-4 then M <= round(B( 6 downto 2)); E <= "011";
    elsif BU < 256-8 then M <= round(B( 7 downto 3)); E <= "100";
    elsif BU < 512-16 then M <= round(B( 8 downto 4)); E <= "101";
    elsif BU < 1024-32 then M <= round(B( 9 downto 5)); E <= "110";
    elsif BU < 2048-64 then M <= round(B(10 downto 6)); E <= "111";
    else                    M <= "1111";                E <= "111";
    end if;
  end process;
end fpencr_arch;
```

Еще раз: результаты синтеза, выполненного по этой поведенческой программе, не обязательно окажутся эффективными. Помимо многочисленных операторов сравнения, мы должны теперь побеспокоиться относительно большого числа 4-разрядных сумматоров, которые могут возникнуть при синтезе как следствие многократных обращений к функции округления `round`. Вопрос о том, как следует изменить архитектуру, чтобы по ней синтезировался только один сумматор, оставлен читателю в качестве задачи 6.15.

«ПОЕДАНИЕ ВЕНТИЛЕЙ»

Для операции округления не требуется 4-разрядный сумматор, необходима только схема увеличения числа на 1, так как одно из слагаемых – всегда 1. Некоторые VHDL-средства могут выдать в результате синтеза полный сумматор, в то время как другие могут оказаться настолько сообразительными, что синтезируют схему увеличения числа на 1, состоящую из существенно меньшего количества вентилей.

В некоторых случаях это может не быть существенным. Самые развитые программные средства проектирования устройств на основе ИС типа FPGA и специализированных ИС содержат программы *поглощения вентилей*. Эти программы ищут вентили с постоянными сигналами на входах и либо исключают такие вентили целиком, либо уменьшают число входов у таких вентилей. Например, можно исключить вентиль И, на одном из входов которого постоянно присутствует 1, а вентиль И с постоянно присутствующим 0 на одном из его входов можно заменить постоянным сигналом, равным 0.

Программа поглощения вентилей прослеживает влияние постоянных значений входных сигналов настолько далеко в схеме, насколько это возможно. Следовательно, такая программа может преобразовать 4-разрядный сумматор с постоянной 1 на одном из входов в более экономичную 4-разрядную схему увеличения числа на 1 .

6.3.3. Двойной приоритетный шифратор

В этом примере мы воспользуемся языком VHDL для поведенческого описания реализуемого в ПЛУ приоритетного шифратора, который находит сигналы с активным уровнем с самым высоким приоритетом и со вторым по старшинству приоритетом в наборе из восьми входных сигналов запроса с высоким активным уровнем `[R0..R7]`, среди которых сигнал `R0` имеет высший приоритет. Сигналы `A(2 downto 0)` и `AVALID` будем использовать для представления запроса с самым высоким приоритетом, причем сигнал `AVALID` пусть принимает единичное значение только тогда, когда запрос с высшим приоритетом присутствует. Точно так же сигналы `B(2 downto 0)` и `BVALID` пусть представляют запрос со вторым по старшинству приоритетом.

В табл. 6.26 представлена поведенческая VHDL-программа для приоритетного шифратора. Вместо вложенных операторов "`if`", как предыдущем примере, мы применяем здесь цикл "`for`". Этот подход позволяет нам обработать запросы как наивысшего, так и второго по старшинству приоритетов в пределах одного и того же цикла, проходя весь путь от сигнала с высшим приоритетом до сигнала с наименьшим приоритетом. Помимо пакета `std_logic_1164`, в программе используется пакет IEEE `std_logic_arith`, из которого берется функция `CONV_STD_LOGIC_VECTOR`. Эта функция в явном виде была приведена в табл. 4.39.

Обратите внимание, что порты `AVALID` и `BVALID` объявлены в программе как выходные сигналы вида `buffer`, поскольку они читаются в пределах архи-

тектуры. Если бы вы поместили объявление сигналов AVALID и BVALID в определение объекта и указали бы, что они являются выходными сигналами вида out, то в архитектуре можно было бы использовать тот же самый подход, но при этом вам пришлось бы объявить внутри процесса локальные переменные, соответствующие сигналам AVALID и BVALID. Обратите также внимание на то, что мы включили AVALID и BVALID в список чувствительности процесса. Хотя, строго говоря, делать это не обязательно, однако такой шаг предотвратит появление предупреждений, которые в противном случае стал бы выдавать компилятор по поводу использования значений сигналов, которых нет в списке чувствительности.

Табл. 6.26. Поведенческая VHDL-программа для двойного приоритетного шифратора

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Vprior2 is
    port (
        R: in STD_LOGIC_VECTOR (0 to 7);
        A, B: out STD_LOGIC_VECTOR (2 downto 0);
        AVALID, BVALID: buffer STD_LOGIC
    );
end Vprior2;

architecture Vprior2_arch of Vprior2 is
begin
  process(R, AVALID, BVALID)
  begin
    AVALID <= '0'; BVALID <= '0'; A <= "000"; B <= "000";
    for i in 0 to 7 loop
      if R(i) = '1' and AVALID = '0' then
      A <= CONV_STD_LOGIC_VECTOR(i,3); AVALID <= '1';
      elsif R(i) = '1' and BVALID = '0' then
      B <= CONV_STD_LOGIC_VECTOR(i,3); BVALID <= '1';
      end if;
    end loop;
  end process;
end Vprior2_arch;
```

Возможен также другой подход к созданию двойного приоритетного шифратора с вложенным оператором "if". Пример программы, реализующей этот подход, приведен в табл. 6.27. Такой подход приводит к более длинной программе с бóльшим числом возможных сбоев, но, с другой стороны, этот вариант может дать лучший результат синтеза; единственный способ узнать возможности конкретной программы состоит в том, чтобы синтезировать схему и проанализировать результаты с точки зрения задержки и числа логических ячеек или вентилей.

Табл. 6.27. Другой вариант VHDL-архитектуры для двойного приоритетного шифратора

```
architecture Vprior2i_arch of Vprior2 is
begin
  process(R, A, AVALID, BVALID)
  begin
    if    R(0) = '1' then A <= "000"; AVALID <= '1';
    elsif R(1) = '1' then A <= "001"; AVALID <= '1';
    elsif R(2) = '1' then A <= "010"; AVALID <= '1';
    elsif R(3) = '1' then A <= "011"; AVALID <= '1';
    elsif R(4) = '1' then A <= "100"; AVALID <= '1';
    elsif R(5) = '1' then A <= "101"; AVALID <= '1';
    elsif R(6) = '1' then A <= "110"; AVALID <= '1';
    elsif R(7) = '1' then A <= "111"; AVALID <= '1';
    else                  A <= "000"; AVALID <= '0';
    end if;
    if    R(1) = '1' and A /= "001" then B <= "001"; BVALID <= '1';
    elsif R(2) = '1' and A /= "010" then B <= "010"; BVALID <= '1';
    elsif R(3) = '1' and A /= "011" then B <= "011"; BVALID <= '1';
    elsif R(4) = '1' and A /= "100" then B <= "100"; BVALID <= '1';
    elsif R(5) = '1' and A /= "101" then B <= "101"; BVALID <= '1';
    elsif R(6) = '1' and A /= "110" then B <= "110"; BVALID <= '1';
    elsif R(7) = '1' and A /= "111" then B <= "111"; BVALID <= '1';
    else                                 B <= "000"; BVALID <= '0';
    end if;
  end process;
end Vprior2i_arch;
```

Вложенные операторы "`if`" и "`for`" могут приводить в процессе синтеза к появлению цепочек с большими задержками. Чтобы гарантировать получение быстрого двойного приоритетного шифратора, необходимо при проектировании следовать структурному или полуструктурному подходу. Можно, например, начать с описания модели быстрого 8-входового приоритетного шифратора в стиле потока данных, используя идеи, нашедшие свое отражение в принципиальной схеме ИС 74x148, приведенной на рис. 5.50, или в соответствующей программе на языке ABEL (табл. 5.24). Затем можно два таких шифратора поместить в одну структуру, где для нахождения входа со вторым по старшинству приоритетом исключается вход с высшим приоритетом, чтобы найти второй вход, как это было в схеме, изображенной на рис. 6.6.

6.3.4. Расширение компараторов

Каскадное включение компараторов является чем-то таким, что мы обычно не стали бы делать в поведенческой модели, написанной на языке VHDL, потому что этот язык и пакет IEEE `std_logic_arith` позволяют нам непосредственно определять компараторы любой желаемой длины. Однако, в действительности может потребоваться запись структурных или полуструктурных VHDL-программ, которые специальным образом включают меньшие компоненты компаратора для получения высокой эффективности.

В табл. 6.28 приведена простая поведенческая модель 64-разрядного компаратора с выходами «равно» и «больше чем». В этой программе используется пакет IEEE std_logic_unsigned, чьи встроенные функции сравнения автоматически воспринимают все сигналы типа STD_LOGIC_VECTOR как целые числа без знака. Хотя эта программа безусловно синтезируема, быстродействие и размеры результирующей схемы зависят от «интеллектуальных возможностей» тех программных средств, которыми вы пользуетесь.

Табл. 6.28. Поведенческая VHDL-программа для 64-разрядного компаратора

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity comp64 is
    port ( A, B: in STD_LOGIC_VECTOR (63 downto 0);
           EQ, GT: out STD_LOGIC );
end comp64;

architecture comp64_arch of comp64 is
begin
  EQ <= '1' when A = B else '0';
  GT <= '1' when A > B else '0';
end comp64_arch;
```

Альтернативой может служить последовательное включение таких, например, меньших компонентов, как 8-разрядные компараторы. В табл. 6.29 представлена поведенческая модель 8-разрядного компаратора. Развитые программные средства синтеза могут по этой программе создать очень быстрый компаратор, но даже при меньших возможностях программных средств можно быть уверенным, что в любом случае такой компаратор будет значительно более быстрым, чем 64-разрядный компаратор.

Табл. 6.29. VHDL-программа для 8-разрядного компаратора

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity comp8 is
    port ( A, B: in STD_LOGIC_VECTOR (7 downto 0);
           EQ, GT: out STD_LOGIC );
end comp8;

architecture comp8_arch of comp8 is
begin
  EQ <= '1' when A = B else '0';
  GT <= '1' when A > B else '0';
end comp8_arch;
```

Теперь мы можем написать структурную программу, которая предусматривает создание восьми таких 8-разрядных компараторов, выходные сигналы которых пропускаются через дополнительную логику, чтобы найти полный результат сравнения. Один из способов, каким это можно сделать, показан в табл. 6.30. Оператор `generate` создает не только отдельные 8-разрядные компараторы, но также и логику покаскадного включения, с помощью которой информация, необходимая для выдачи окончательного результата, собирается путем последовательного учета сигналов на выходах отдельных каскадов, начиная с самого старшего и кончая самым младшим.

Табл. 6.30. Структурная VHDL-архитектура для 64-разрядного компаратора

```
architecture comp64s_arch of comp64 is
component comp8
  port ( A, B  in STD_LOGIC_VECTOR (7 downto 0),
         EQ, GT  out STD_LOGIC),
end component,
signal EQ8, GT8  STD_LOGIC_VECTOR (7 downto 0); -- =, > for 8 bit slice
signal SEQ, SGT  STD_LOGIC_VECTOR (8 downto 0), -- serial chain of slice results
begin
  SEQ(8) <= '1'; SGT(8) <= '0';
  U1: for i in 7 downto 0 generate
        U2. comp8 port map (A(7+i*8 downto i*8), B(7+i*8 downto i*8), EQ8(i), GT8(i));
        SEQ(i) <= SEQ(i+1) and EQ8(i);
        SGT(i) <= SGT(i+1) or (SEQ(i+1) and GT8(i));
  end generate,
  EQ <= SEQ(0), GT <= SGT(0),
end comp64s_arch,
```

Если по архитектуре, приведенной в табл. 6.28, относительно простые программные средства могут выдать в качестве результата синтеза схему медленного итерационного компаратора, то в случае архитектуры из табл. 6.30 результатом синтеза будет более быстрое устройство, поскольку в нем в более явной форме «извлекается» информация из каждого 8-разрядного звена, которая затем пропускается через более быструю комбинационную схему (состоящую всего лишь из 8-уровневой логики И–ИЛИ, а не из 64-уровневой). Более сильные программные средства могут «распараллелить» 8-разрядный компаратор, предложив более быструю неитерационную структуру наподобие ИС средней степени интеграции 74x682 (см. рис. 5.84), и преобразовать нашу итерационную логику объединения сигналов, поступающих от отдельных каскадов, в двухуровневую схему, реализующую выражения вида «сумма произведений», подобные тем, какие были приведены в программе на языке ABEL в табл. 6.8.

6.3.5. Компаратор с управляемым режимом работы

В качестве следующего примера давайте предположим, что имеется система, в которой нужно, как правило, сравнивать два 32-разрядных двоичных слова, но иногда во входных словах необходимо игнорировать значения одного или двух младших разрядов. Режим работы задается двумя битами M1 и M0, как указано в табл. 6.9.

Желаемое функционирование разрабатываемого устройства совсем легко обеспечить средствами языка VHDL, используя оператор `case` для выбора нужного

поведения, как это сделано в программе в табл. 6.31. Эта программа представляет собой вполне доброкачественное поведенческое описание, которое также полностью синтезируемо. Однако у такого описания имеется все же существенный недостаток: оно, вероятнее всего, приведет к созданию в процессе синтеза трех отдельных компараторов для обнаружения равенства или неравенства сравниваемых величин (представленных 32, 31 и 30 двоичными разрядами), по одному на каждый из случаев в операторе `case`. Отдельные компараторы могут при этом быть или не быть быстродействующими, как это обсуждалось в предыдущем разделе, но в данном примере мы не будем подробно разбирать этот вопрос.

Табл. 6.31. VHDL-программа с поведенческой архитектурой для 32-разрядного компаратора с управляемым режимом работы

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Vmodecmp is
    port ( M  : in STD_LOGIC_VECTOR (1 downto 0);      -- mode
           A, B : in STD_LOGIC_VECTOR (31 downto 0);   -- unsigned integers
           EQ, GT : out STD_LOGIC );                   -- comparison results
end Vmodecmp;

architecture Vmodecmp_arch of Vmodecmp is
begin
  process (M, A, B)
  begin
    case M is
      when "00" =>
        if A = B then EQ <= '1'; else EQ <= '0'; end if;
        if A > B then GT <= '1'; else GT <= '0'; end if;
      when "01" =>
        if A(31 downto 1) = B(31 downto 1) then EQ <= '1'; else EQ <= '0'; end if;
        if A(31 downto 1) > B(31 downto 1) then GT <= '1'; else GT <= '0'; end if;
      when "10" =>
        if A(31 downto 2) = B(31 downto 2) then EQ <= '1'; else EQ <= '0'; end if;
        if A(31 downto 2) > B(31 downto 2) then GT <= '1'; else GT <= '0'; end if;
      when others => EQ <= '0'; GT <= '0';
    end case;
  end process;
end Vmodecmp_arch;
```

Более эффективное решение заключается в выполнении только одного сравнения входных слов по 30 старшим битам и получении окончательного результата с помощью дополнительной логики, которая реализует зависящую от режима работы функцию и позволяет, по мере необходимости, учитывать значения младших разрядов. Этот подход продемонстрирован в табл. 6.32. Результат сравнения 30 старших битов представлен внутри процесса двумя переменными: `EQ30` и `GT30`. Затем используется оператор `case`, аналогичный приведенному в предыдущей архитектуре, посредством которого получается окончательный результат в зависимости от режима работы. Если желательно, то 30-разрядный компаратор можно оптимизировать в отношении быстродействия методами, которые были рассмотрены в предыдущем разделе.

Табл. 6.32. Более эффективная архитектура для 32-разрядного компаратора с управляемым режимом работы

```
architecture Vmodecpe_arch of Vmodecmp is
begin
  process (M, A, B)
  variable EQ30, GT30  STD LOGIC, -- 30 bit comparison results
  begin
    if A(31 downto 2) = B(31 downto 2) then EQ30  = '1', else EQ30  = '0', end if,
    if A(31 downto 2) > B(31 downto 2) then GT30  = '1', else GT30  = '0', end if,
    case M is
      when "00" =>
        if EQ30='1' and A(1 downto 0) = B(1 downto 0) then
          EQ <  '1', else EQ <= '0', end if,
        if GT30='1' or (EQ30='1  and A(1 downto 0) > B(1 downto 0)) then
          GT <= '1', else GT <= '0', end if,
      when "01" =>
        if EQ30='1' and A(1) = B(1) then EQ <= '1', else EQ <= '0', end if,
        if GT30='1' or (EQ30='1' and A(1) > B(1)) then
          GT <= '1', else GT <= '0', end if,
      when "10" =>  EQ <= EQ30,  GT <= GT30,
      when others => EQ <= '0',  GT <= '0',
    end case,
  end process;
end Vmodecpe_arch,
```

6.3.6. Счетчик числа единиц

Несколько важных алгоритмов предусматривают счет числа единичных битов в слове данных. Подсчет числа единиц недавно был включен в системы команд ряда микропроцессоров в качестве одной из основных операций. В этом примере предполагается, что нам надо построить комбинационную схему, считающую число единиц в 32-разрядном двоичном слове, которая могла бы быть частью арифметическо-логического устройства микропроцессора.

Подсчет числа единиц совсем нетрудно описать в поведенческой VHDL-программе, как это видно из табл. 6.33. Эта программа вполне синтезируема, но результатом синтеза может оказаться очень медленная и неэффективная реализация, состоящая из 32 последовательно включенных 5-разрядных сумматоров.

Чтобы синтезировать счетчик числа единиц с лучшими параметрами, необходимо придумать экономичную структуру и затем описать ее в виде архитектуры. Такой структурой является дерево сумматоров, показанное на рис. 6.15. Полный сумматор (FA) выполняет сложение трех входных битов, вырабатывая 2-разрядную сумму. Пары 2-разрядных чисел складываются с помощью 2-разрядных сумматоров ADDER2, у каждого из которых имеется вход переноса, позволяющий добавить к сумме значение еще одного бита. Полученные 3-разрядные суммы объединяются 3-разрядными сумматорами ADDER3, а последняя пара 4-разрядных сумм складывается в 4-разрядном сумматоре ADDER4. С учетом сигналов, подаваемых на входы переноса, эта древовидная структура обеспечивает подсчет числа единиц в 31 разряде. При наличии единицы в оставшемся входном разряде это обстоятельство учитывается с помощью отдельного 5-разрядного устройства INCR5 увеличения числа на единицу.

Табл. 6.33. Поведенческая VHDL-программа для счетчика числа единиц в 32-разрядном слове

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Vcnt1s is
    port ( D: in STD_LOGIC_VECTOR (31 downto 0);
           SUM: out STD_LOGIC_VECTOR (4 downto 0) );
end Vcnt1s;

architecture Vcnt1s_arch of Vcnt1s is
begin
  process (D)
  variable S: STD_LOGIC_VECTOR(4 downto 0);
  begin
    S := "00000";
    for i in 0 to 31 loop
      if D(i) = '1' then S := S + "00001"; end if;
    end loop;
    SUM <= S;
  end process;
end Vcnt1s_arch;
```

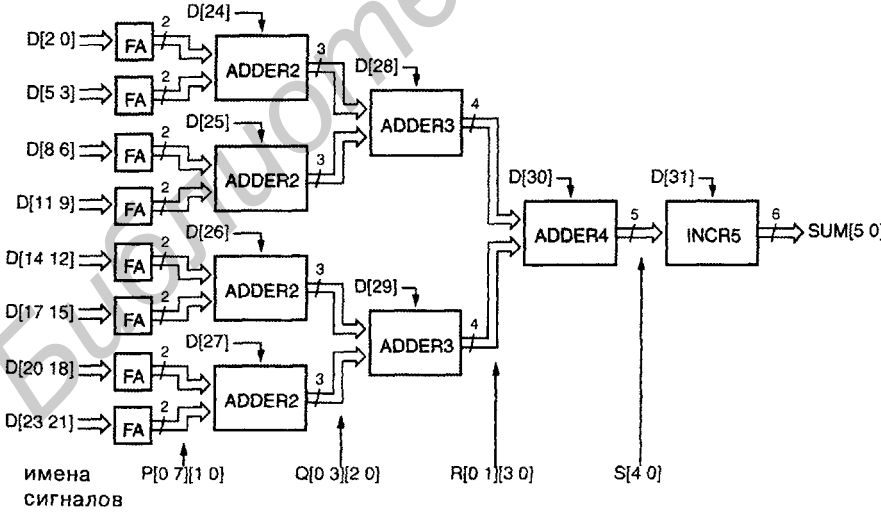


Рис. 6.15. Структура 32-разрядного счетчика числа единиц

Устройство, изображенное на рис. 6.15, отлично создается структурной VHDL-архитектурой, приведенной в табл. 6.34. Программа начинается с объявления всех компонентов, которые будут использованы в проекте, соответствующих блокам на рисунке.

Табл. 6.34. Структурная VHDL-архитектура для 32-разрядного счетчика числа единиц

```
architecture Vcnt1str_arch of Vcnt1str is

component FA port ( A, B, CI: in  STD_LOGIC;
                    S, CO:    out STD_LOGIC );
end component;

component ADDER2 port ( A, B: in  STD_LOGIC_VECTOR(1 downto 0);
                        CI:   in  STD_LOGIC,
                        S:    out STD_LOGIC_VECTOR(2 downto 0) );
end component,

component ADDER3 port ( A, B. in  STD_LOGIC_VECTOR(2 downto 0),
                        CI.   in  STD_LOGIC;
                        S:    out STD_LOGIC_VECTOR(3 downto 0) );
end component;

component ADDER4 port ( A, B: in  STD_LOGIC_VECTOR(3 downto 0);
                        CI:   in  STD_LOGIC;
                        S:    out STD_LOGIC_VECTOR(4 downto 0) );
end component;

component INCR5 port ( A:  in  STD_LOGIC_VECTOR(4 downto 0);
                       CI: in  STD_LOGIC;
                       S:  out STD_LOGIC_VECTOR(5 downto 0) );
end component;

type Ptype is array (0 to 7) of STD_LOGIC_VECTOR(1 downto 0);
type Qtype is array (0 to 3) of STD_LOGIC_VECTOR(2 downto 0);
type Rtype is array (0 to 1) of STD_LOGIC_VECTOR(3 downto 0);
signal P: Ptype;  signal Q: Qtype;  signal R: Rtype;
signal S: STD_LOGIC_VECTOR(4 downto 0);

begin
  U1: for i in 0 to 7 generate
    U1C: FA port map (D(3*i), D(3*i+1), D(3*i+2), P(i)(0), P(i)(1));
  end generate;
  U2: for i in 0 to 3 generate
    U2C: ADDER2 port map (P(2*i), P(2*i+1), D(24+i), Q(i));
  end generate;
  U3: for i in 0 to 1 generate
    U3C: ADDER3 port map (Q(2*i), Q(2*i+1), D(28+i), R(i));
  end generate;
  U4: ADDER4 port map (R(0), R(1), D(30), S);
  U5: INCR5 port map (S, D(31), SUM);
end Vcnt1str_arch;
```

Под каждым столбцом сигналов в схеме на рис. 6.15 указано имя, присвоенное этой совокупности сигналов в программе. Каждый из сигналов P, Q и R – это массив, позволяющий представить все соединения в соответствующем столбце в виде одного вектора типа STD_LOGIC_VECTOR. Объявлению этих сигналов в программе предшествует определение соответствующих типов.

В этой программе создание однотипных сумматоров – восьми полных сумматоров FA, четырех сумматоров ADDER2 и двух сумматоров ADDER3 – успешно осуществляется операторами `generate`, а затем происходит обращение к компонентам `ADDER4` и `INCR5`.

Определение компонентов счетчика числа единиц в виде отдельных объектов и архитектур, начиная с полного сумматора `FA` и кончая устройством `INCR` увеличения числа на 1, можно сделать в отдельных структурных или поведенческих программах. Например, в табл. 6.35 приведена структурная программа для полного сумматора `FA`. Описание остальных блоков оставлено в качестве задач 6.20–6.22.

Табл. 6.35. Структурная VHDL-программа для полного сумматора

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FA is
    port ( A, B, CI: in  STD_LOGIC;
           S, CO:    out STD_LOGIC );
end FA;

architecture FA_arch of FA is
begin
  S <= A xor B xor CI;
  CO <= (A and B) or (A and CI) or (B and CI);
end FA_arch;
```

6.3.7. Игра в крестики и нолики

Наш последний пример состоит в проектировании комбинационной схемы, которая выбирает очередной ход игрока в игре в крестики и нолики. Те, кто не знаком с этой игрой, могут прочитать правила, приведенные в разделе 6.2.7. Здесь мы повторим нашу стратегию игры с целью достижения победы:

1. Ищем строку, столбец или диагональ, в которых имеется две наших метки (X или O в зависимости от того, за кого мы играем) и одна пустая клетка. Если такая комбинация существует, то помещаем свою метку в пустую клетку. Мы выиграли!
2. В противном случае ищем строку, столбец или диагональ, в которых имеется две метки противника и одна пустая клетка. Если такая комбинация обнаруживается, то помещаем свою метку в пустую клетку, чтобы блокировать возможную победу противника.
3. Если не найдены две предыдущие комбинации, то выбираем клетку «на основании опыта». Например, если свободна центральная клетка, то обычно хорошим ходом является ее занятие. Другими хорошими ходами считается занятие угловых клеток. При выборе хода умные игроки могут также принять во внимание развитие конфигурации противником и заблокировать его, воспользовавшись «предвидением».

Чтобы избежать путаницы в наших программах между символами "O" и "0", присвоим второму игроку имя "Y". Теперь можно подумать о том, как кодировать входные и выходные сигналы схемы. Входные сигналы – это текущее состояние игрового поля. Всего в игровом поле девять клеток и каждая клетка находится в одном из трех возможных состояний (пустая, заполненная меткой X, заполненная меткой Y). Выходной сигнал – это ход, который надо сделать; предполагается, что очередной ход за игроком X. Игроку предстоит сделать только один из девяти возможных ходов, так что для представления выходного сигнала достаточно четырех битов.

Можно несколькими способами кодировать состояние одной клетки. Поскольку игра симметрична, выберем симметричное кодирование:

00 – клетка пуста;
10 – клетка занята меткой X;
01 – клетка занята меткой Y.

Такое представление поможет нам позднее.

Итак, состояние игрового поля размером 3×3 можно представить 18-ю битами. Поскольку язык VHDL поддерживает массивы, полезно определить тип массива `TTTgrid`, элементы которого соответствуют клеткам игрового поля. Так как мы повсюду в нашем проекте будем использовать этот тип, удобно включить его определение, наряду с несколькими другими, о которых будет сказано позднее, в VHDL-пакет, как показано в табл. 6.36.

Табл. 6.36. VHDL-пакет с определениями для устройства, играющего в крестики и нолики

```
library IEEE;
use IEEE.std_logic_1164.all;

package TTTdefs is

type TTTgrid is array (1 to 3) of STD_LOGIC_VECTOR(1 to 3);
subtype TTTmove is STD_LOGIC_VECTOR (3 downto 0);

constant MOVE11: TTTmove := "1000";
constant MOVE12: TTTmove := "0100";
constant MOVE13: TTTmove := "0010";
constant MOVE21: TTTmove := "0001";
constant MOVE22: TTTmove := "1100";
constant MOVE23: TTTmove := "0111";
constant MOVE31: TTTmove := "1011";
constant MOVE32: TTTmove := "1101";
constant MOVE33: TTTmove := "1110";
constant NONE:   TTTmove := "0000";

end TTTdefs;
```

Было бы естественно определить тип TTTgrid как двумерный массив элементов типа STD_LOGIC, но не все VHDL-средства поддерживают двумерные массивы. Поэтому введем массив 3-разрядных векторов типа STD_LOGIC_VECTOR, что является почти тем же самым. Для представления игрового поля игры в крестики и нолики воспользуемся двумя сигналами этого типа X и Y, где элемент переменной равен 1, когда у игрока с соответствующим именем стоит метка в данной клетке. На рис. 6.16 показано соответствие между именами сигналов и клетками на игровом поле.

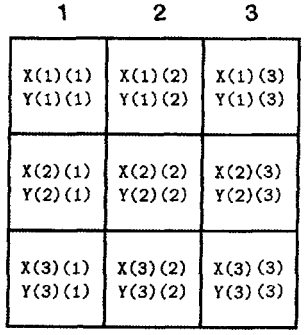


Рис. 6.16. Поле для игры в крестики и нолики и имена сигналов в VHDL-программе

В пакете в табл. 6.36 определен также 4-разрядный тип TTTmove для представления ходов. У игрока есть девять возможных ходов, и нужно еще одно кодовое слово для случая, когда никакой ход не возможен. Выбор именно такого способа кодирования и его включение в пакет обусловлено только тем, что точно так же мы поступили в примере проектирования нашего устройства средствами языка ABEL в разделе 6.2.7. Благодаря тому, что способ кодирования определен в пакете, можно позднее изменить это определение без необходимости вносить изменения в использующие это определение объекты (см., например, задачу 6.23).

Имеет смысл не пытаться разработать устройство, выбирающее ход в игре в крестики и нолики, как единое целое, а попробовать разбить его на меньшие блоки. Действительно, представляется разумным разбиение устройства на блоки в соответствии с тремя шагами стратегии игры, указанными в начале этого раздела.

Отметим, что 1-й и 2-й шаги в нашей стратегии очень похожи: они отличаются только тем, что игроки меняются ролями. Объект, находящий победный ход для меня, может также находить ход, блокирующий выигрыш моего противника. С другой стороны, это означает, что объект, который находит победный ход для меня, может найти для меня и блокирующий ход, если поменять местами представление игрового поля с моей точки зрения и с точки зрения моего противника. Именно здесь дает выигрыш наше симметричное кодирование: игроков легко менять местами простой перестановкой сигналов X и Y.

Имея это в виду, воспользуемся для выполнения 1-го и 2-го шагов двумя экземплярами одного и того же объекта TwoInRow, как показано на рис. 6.17. Обратите внимание, что сигнал X подан на верхний вход первого объекта TwoInRow и на нижний вход такого же второго объекта. Третий объект PICK выбирает победный ход, если он имеется на выходе объекта U1, в противном случае объект PICK выбирает блокирующий ход, если он имеется на выходе объекта U2; когда ни того, ни другого хода нет, в объекте PICK «используется опыт» для выбора хода на 3-м шаге.

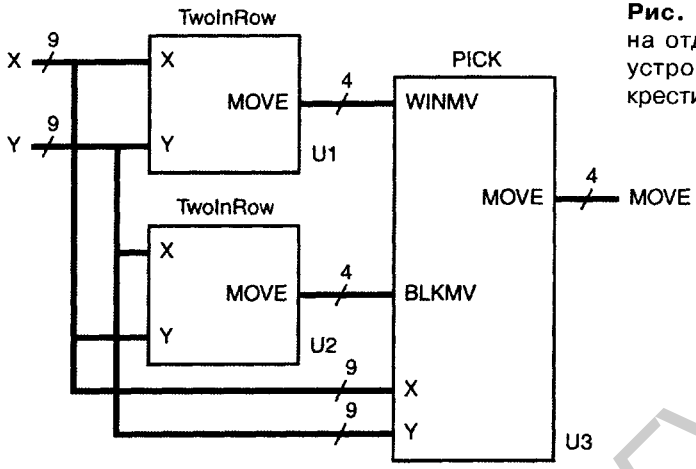


Рис. 6.17. Разбиение на отдельные объекты устройства для игры в крестики и нолики

Табл. 6.37 представляет собой структурную VHDL-программу для объекта верхнего уровня `GETMOVE`. Помимо пакета IEEE `std_logic_1164` в ней используется наш пакет `TTTdefs`. Обратите внимание, что предложением "`use`" предписывается сохранение пакета `TTTdefs` в библиотеке "`work`", которая создается автоматически для нашего проекта.

Табл. 6.37. Структурный VHDL-объект верхнего уровня для выбора хода в игре в крестики и нолики

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.TTTdefs.all;

entity GETMOVE is
    port ( X, Y: in  TTTgrid;
           MOVE: out TTTmove );
end GETMOVE;

architecture GETMOVE_arch of GETMOVE is

component TwoInRow port ( X, Y: in  TTTgrid;
                          MOVE: out STD_LOGIC_VECTOR(3 downto 0) );
end component;

component PICK port ( X, Y:          in  TTTgrid;
                      WINMV, BLKMV: in  STD_LOGIC_VECTOR(3 downto 0);
                      MOVE:         out STD_LOGIC_VECTOR(3 downto 0) );
end component;

signal WIN, BLK: STD_LOGIC_VECTOR(3 downto 0);

begin
  U1: TwoInRow port map (X, Y, WIN);
  U2: TwoInRow port map (Y, X, BLK);
  U3: PICK port map (X, Y, WIN, BLK, MOVE);
end GETMOVE_arch;
```

В архитектуре в табл. 6.37 объявлены и используются только два компонента: TwoInRow и PICK, которые вскоре будут определены. От двух блоков TwoInRow к блоку PICK поступают только два внутренних сигнала WIN и BLK, приводящие к победному или блокирующему ходу, как и на рис. 6.17. Обработка этих сигналов производится всего лишь тремя операторами в исполняемой части архитектуры, соответствующими блокам, указанным на рисунке.

Теперь пришла очередь интересной работы: нужно создать отдельные объекты, изображенные на рис. 6.17. Начнем с объектов TwoInRow, так как они составляют две трети проекта. Согласно табл. 6.38, объявление такого объекта не представляет труда. Но в отношении его архитектуры, приведенной в табл. 6.39, есть целый ряд вопросов, которые следует обсудить.

Табл. 6.38. Объявление объекта TwoInRow

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.TTTdefs.all;

entity TwoInRow is
    port ( X, Y: in  TTTgrid;
           MOVE: out TTTmove );
end TwoInRow;
```

В архитектуре определены несколько функций, в каждой из которых решается, будет ли победным (с точки зрения игрока X) ход в конкретную клетку i,j. Победный ход существует, когда клетка i,j пуста и две другие клетки в той же строке, в том же столбце или на той же диагонали содержат метку X. Функции R и C выполняют поиск победного хода в клетки i,j по строкам и столбцам соответственно. Функции D и E осуществляют то же самое по двум диагоналям.

В единственном процессе архитектуры объявлены девять переменных G11–G33 типа BOOLEAN для указания возможности победного хода в каждую клетку. В начале процесса каждой из этих переменных присваивается значение TRUE, если посредством вызова нужных функций и объединения их результатов устанавливается, что ход в клетку i,j возможен.

Остальная часть процесса представляет собой оператор "if" с большой глубиной вложений, который ищет победный ход во все возможные клетки. Хотя обычно это приводит к синтезу более медленной логики, тем не менее, вложенный оператор "if" предпочтительнее, нежели какая-либо разновидность оператора "case", поскольку может быть несколько допустимых ходов. Если победного хода нет, то соответствующей переменной присваивается значение "NONE".

ФУНКЦИИ ЧИСТЫЕ И НЕЧИСТЫЕ

Помимо индекса клетки i,j, в функции R, C, D и E в табл. 6.39 передается состояние игрового поля в виде массивов X и Y. Это необходимо делать потому, что, по умолчанию, VHDL-функции являются *чистыми* (*pure*), а это означает, что сигналы и переменные, объявленные в структуре, порождающей функцию, непосредственно *не видимы* в пределах функции. Однако это ограничение можно ослабить путем явного объявления функции *нечистой*, помещая ключевое слово impure перед ключевым словом function в ее определении

Табл. 6.39. Архитектура объекта TwoInRow

```
architecture TwoInRow_arch of TwoInRow is

function R(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;
  begin                                -- Find 2-in-row with empty cell i,j
    result := TRUE;
    for jj in 1 to 3 loop
      if jj = j then result := result and X(i)(jj)='0' and Y(i)(jj)='0';
      else result := result and X(i)(jj)='1'; end if;
    end loop;
    return result;
  end R;

function C(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;
  begin                                -- Find 2-in-column with empty cell i,j
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(j)='0' and Y(ii)(j)='0';
      else result := result and X(ii)(j)='1'; end if;
    end loop;
    return result;
  end C;

function D(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;          -- Find 2-in-diagonal with empty cell i,j.
  begin                              -- This is for 11, 22, 33 diagonal.
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(ii)='0' and Y(ii)(ii)='0';
      else result := result and X(ii)(ii)='1'; end if;
    end loop;
    return result;
  end D;

function E(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;          -- Find 2-in-diagonal with empty cell i,j.
  begin                              -- This is for 13, 22, 31 diagonal.
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(4-ii)='0' and Y(ii)(4-ii)='0';
      else result := result and X(ii)(4-ii)='1'; end if;
    end loop;
    return result;
  end E;
begin
  process (X, Y)
  variable G11, G12, G13, G21, G22, G23, G31, G32, G33: BOOLEAN;
  begin
    G11 := R(X,Y,1,1) or C(X,Y,1,1) or D(X,Y,1,1);
    G12 := R(X,Y,1,2) or C(X,Y,1,2);
    G13 := R(X,Y,1,3) or C(X,Y,1,3) or E(X,Y,1,3);
    G21 := R(X,Y,2,1) or C(X,Y,2,1);
    G22 := R(X,Y,2,2) or C(X,Y,2,2) or D(X,Y,2,2) or E(X,Y,2,2);
    G23 := R(X,Y,2,3) or C(X,Y,2,3);
```

Табл. 6.39. Архитектура объекта `TwoInRow` (продолжение)

```
    G31 := R(X,Y,3,1) or C(X,Y,3,1) or E(X,Y,3,1);
    G32 := R(X,Y,3,2) or C(X,Y,3,2);
    G33 := R(X,Y,3,3) or C(X,Y,3,3) or D(X,Y,3,3);
    if    G11 then MOVE <= MOVE11;
    elsif G12 then MOVE <= MOVE12;
    elsif G13 then MOVE <= MOVE13;
    elsif G21 then MOVE <= MOVE21;
    elsif G22 then MOVE <= MOVE22;
    elsif G23 then MOVE <= MOVE23;
    elsif G31 then MOVE <= MOVE31;
    elsif G32 then MOVE <= MOVE32;
    elsif G33 then MOVE <= MOVE33;
    else           MOVE <= NONE;
    end if;
  end process;
end TwoInRow_arch;
```

Табл. 6.40. VHDL-программа для блока, который делает победный или блокирующий ходы в игре в крестики и нолики, либо «использует опыт» при выборе очередного хода, когда победного и блокирующего ходов нет

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.TTTdefs.all;

entity PICK is
  port ( X, Y:          in  TTTgrid;
         WINMV, BLKMV: in  STD_LOGIC_VECTOR(3 downto 0);
         MOVE:          out STD_LOGIC_VECTOR(3 downto 0) );
end PICK;

architecture PICK_arch of PICK is
function MT(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  begin                              -- Determine if cell i,j is empty
    return X(i)(j)='0' and Y(i)(j)='0';
  end MT;
begin
  process (X, Y, WINMV, BLKMV)
  begin                                          -- If available, pick:
    if    WINMV /= NONE then MOVE <= WINMV;  -- winning move
    elsif BLKMV /= NONE then MOVE <= BLKMV;  -- else blocking move
    elsif MT(X,Y,2,2)   then MOVE <= MOVE22; -- else center cell
    elsif MT(X,Y,1,1)   then MOVE <= MOVE11; -- else corner cells
    elsif MT(X,Y,1,3)   then MOVE <= MOVE13;
    elsif MT(X,Y,3,1)   then MOVE <= MOVE31;
    elsif MT(X,Y,3,3)   then MOVE <= MOVE33;
    elsif MT(X,Y,1,2)   then MOVE <= MOVE12; -- else side cells
    elsif MT(X,Y,2,1)   then MOVE <= MOVE21;
    elsif MT(X,Y,2,3)   then MOVE <= MOVE23;
    elsif MT(X,Y,3,2)   then MOVE <= MOVE32;
    else                     MOVE <= NONE;   -- else grid is full
    end if;
  end process;
end PICK_arch;
```

В объекте PICK результаты работы двух объектов TwoInRow объединяются согласно программе, приведенной в табл. 6.40. Устанавливается приоритет победного хода по отношению к блокирующему ходу. Если таких ходов нет, то вызывается функция MT, которая для каждой клетки, начиная с центральной клетки и заканчивая клетками по бокам игрового поля, находит возможный ход. Этим завершается разработка устройства для игры в крестики и нолики.

7.2. Защелки и триггеры

Защелки и триггеры являются основными составными элементами в большинстве последовательностных устройств. В типичных цифровых системах используются защелки и триггеры, находящиеся внутри функционально законченных узлов и блоков в стандартной интегральной схеме. В среде проектирования на основе специализированных ИС защелки и триггеры обычно бывают готовыми библиотечными элементами, которые поставляет продавец самих интегральных микросхем. Однако внутри как обычных ИС, так и специализированных ИС, защелка или триггер представляют собой последовательностную схему с обратной связью, состоящую из отдельных логических вентилей и содержащую петли обратной связи. Мы займемся изучением этих конструкций из дискретных компонентов по двум причинам: во-первых, чтобы лучше понять поведение готовых элементов, а во-вторых, чтобы вы могли приобрести навыки построения защелок или триггеров, начиная с нуля, как это порой бывает необходимо в практике разработки цифровых устройств и как это часто требуется на экзаменах по цифровому проектированию.

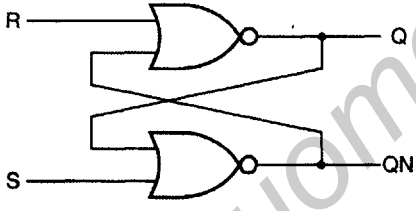
Все специалисты по цифровой электронике называют *триггером* (*flip-flop*) последовательностную схему, в которой значения входных сигналов принимаются во внимание и выходные сигналы изменяются только в моменты времени, задава-

емые тактовым сигналом. С другой стороны, большинство разработчиков цифровой техники используют название «*защелка*» (*latch*) для последовательностной схемы, которая чувствительна к сигналам на ее входах непрерывно в течение всего времени, и значения сигналов на выходах такой схемы могут изменяться в любой момент независимо от тактового сигнала. В этом учебнике мы будем следовать этому правилу. Однако в других учебниках и другими инженерами-электронщиками (неправильно) называется «триггером» устройство, которое мы называем «защелкой».

Поскольку функциональное поведение защелок и триггеров совершенно различно, разработчику в любом случае важно знать, к какому типу относится используемый им элемент; при этом все равно, откуда он узнает об этом: глядя на номер микросхемы (например, 74x374 или 74x373), или по той информации, какую можно почерпнуть из контекста. В следующих разделах мы рассмотрим защелки и триггеры наиболее распространенных типов.

7.2.1. SR-защелка

На рис. 7.5(a) показана *SR-защелка* (*set-reset latch, S-R latch*) на вентилях ИЛИ-НЕ. У этой схемы два входа S и R и два выхода Q и QN, где сигнал QN в нормальных условиях представляет собой инверсию сигнала Q. Сигнал QN иногда обозначают и так: $\overline{Q}$ или Q_L.



(a)

(b)

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

Рис. 7.5. SR-защелка: (a) принципиальная схема на вентилях ИЛИ-НЕ; (b) таблица, описывающая работу схемы (last – последнее значение)

Если оба входных сигнала S и R равны 0, то схема ведет себя аналогично элементу с двумя устойчивыми состояниями: благодаря наличию петли обратной связи сохраняется одно из двух логических состояний – Q = 0 или Q = 1. Как указано в таблице на рис. 7.5(b), подавая сигналы на входы S и R можно заставить схему с петлей обратной связи переходить в желаемое состояние. Сигнал S *устанавливает* (*set*) или *задает* (*preset*) состояние, при котором выходной сигнал Q равен 1; сигнал R *сбрасывает* (*reset*) или *очищает* (*clear*) схему, в результате чего выходной сигнал Q становится равным 0. После того, как входной сигнал S или R переходит на неактивный уровень, защелка остается в том состоянии, в какое ее установил этот сигнал. На рис. 7.6 представлено функциональное поведение SR-защелки при воздействии на нее типичной последовательности входных сигналов. Цветными стрелками указана причинно-следственная связь, то есть показано, какие переходы во входных сигналах вызывают те или иные переходы в выходных сигналах.

ДОСТОИНСТВА И НЕДОСТАТКИ ОБОЗНАЧЕНИЙ Q И QN

В большинстве приложений SR-защелок выходной сигнал QN (известный также под именем $\overline{Q}$) всегда является инверсией выходного сигнала Q. Однако обозначение $\overline{Q}$ не вполне корректно, поскольку существуют обстоятельства, при которых этот выходной сигнал не является инверсией сигнала Q. Когда оба входных сигнала S и R равны 1, как это происходит в нескольких местах на рис. 7.5(b), оба выходных сигнала вынужденно принимают значение 0. Как только любой из входных сигналов снимается, схема возвращается к работе в обычном режиме, и выходные сигналы становятся инверсными один по отношению к другому. Однако если входные сигналы снимаются одновременно, то состояние, в которое защелка перейдет в следующий момент времени, непредсказуемо, и при этом могут возникнуть колебания, либо схема может войти в метастабильное состояние. Неустойчивость может наступать также в том случае, когда единичный импульс, подаваемый на входы S или R, слишком короткий.

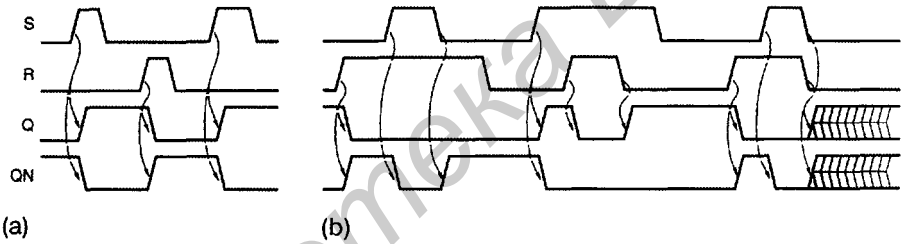


Рис.7.6. Типичная работа SR-защелки: (a) «нормальные» входные сигналы; (b) сигналы S и R имеют активный уровень одновременно

На рис. 7.7 приведены три условных обозначения одной и той же SR-защелки. Эти обозначения различаются тем, как изображен инверсный выход. По исторически сложившейся традиции в первом из этих условных обозначений низкий активный уровень сигнала на этом выходе указывается в имени сигнала внутри прямоугольника, изображающего функциональный блок. Однако в логическом проектировании с использованием принципа «инверсия к инверсии» более предпочтительным является второе условное обозначение, согласно которому символ инверсии располагается снаружи функционального прямоугольника. Последнее условное обозначение, очевидно, является ошибочным.

На рис. 7.8 перечислены временные параметры SR-защелки. *Задержка распространения* (*propagation delay*) – это время, которое уходит на то, чтобы переход во входном сигнале привел к переходу в сигнале на выходе. Для данной защелки или для данного триггера может быть указано несколько значений задержки распространения, по одному на каждую пару вход-выход. Кроме того, значения задержки распространения могут быть различными в зависимости от того, в каком направлении изменяется выходной сигнал: от низкого уровня до

высокого или наоборот. У SR-защелки переход входного сигнала S с низкого уровня на высокий может вызвать переход выходного сигнала Q с низкого уровня на высокий; этому переходу соответствует задержка распространения $t_{pLH(SQ)}$ (случай 1 на рис. 7.8). Аналогично, переход входного сигнала R с низкого уровня на высокий может вызвать переход выходного сигнала Q с высокого уровня на низкий с задержкой распространения $t_{pHL(RQ)}$ (случай 2 на рис.7.8). Не показанные на рисунке соответствующие переходы выходного сигнала QN можно было бы охарактеризовать задержками распространения $t_{pHL(SQN)}$ и $t_{pLH(RQN)}$.

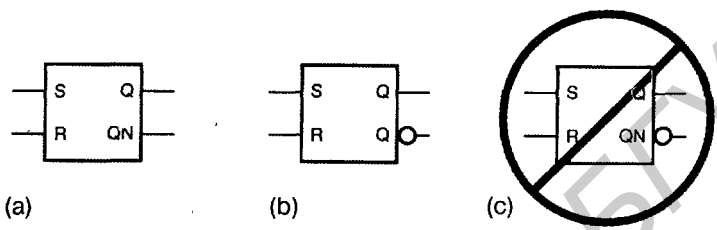


Рис. 7.7. Условное обозначение SR-защелки: (a) без символа инверсии; (b) предпочтительное обозначение при логическом проектировании с использованием принципа «инверсия к инверсии»; (c) неправильное обозначение, поскольку отрицание указано дважды

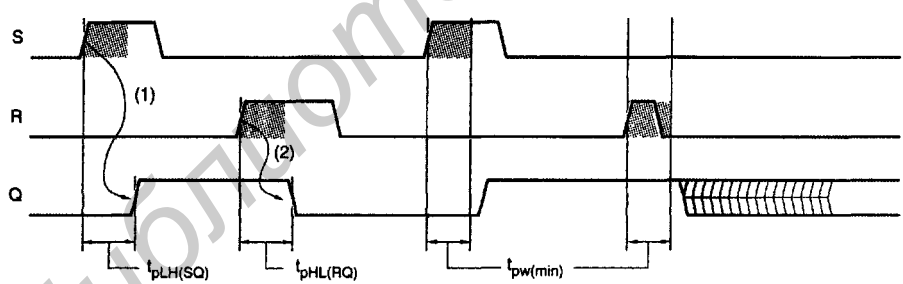


Рис. 7.8. Временные параметры SR-защелки

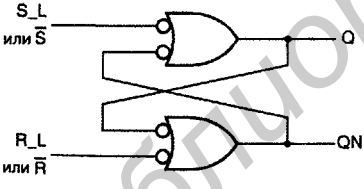
Для входных сигналов S и R обычно бывает задана *минимальная длительность импульса* (*minimum pulse width*). Как показано на рис 7.8, защелка может войти в метастабильное состояние и оставаться в нем в течение отрезка времени случайной длительности, если на входах S или R действует импульс, длительность которого меньше минимального значения $t_{pw(min)}$. Надежное непопадание защелки в метастабильное состояние возможно только в том случае, когда требования в отношении минимальной длительности импульса на входах S или R удовлетворены или превышены.

ЧТО ЗНАЧИТ «БЛИЗКО»?

Как упоминалось в предыдущем замечании, SR-защелка может войти в метастабильное состояние, если входные сигналы S и R переходят на неактивный уровень одновременно. В отношении серийно выпускаемых защелок часто, хотя и не всегда, понятие «одновременно» характеризуется вполне определенной величиной (это имеет место, например, если переходы сигналов S и R на неактивный уровень происходят в пределах отрезка времени длительностью не более 20 нс). Соответствующий параметр иногда называют *временем восстановления* (*recovery time*) t_{rec}. Это минимальный временной сдвиг между переходами S и R на неактивный уровень, при котором эти события считаются неодновременными. Параметр t_{rec} тесно связан с минимальной длительностью импульса. Обе характеристики являются мерой того, как долго петля обратной связи в защелке приходит в равновесие при изменении состояния.

7.2.2. $\overline{SR}$-защелка

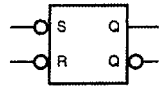
На рис. 7.9 представлена *$\overline{SR}$-защелка* ($\overline{S}$-$\overline{R}$ *latch*; читается: S-bar-R-bar latch, S-с-чертой-R-с-чертой-защелка) на вентилях И-НЕ с низким активным уровнем сигналов установки и сброса. В логических семействах ТТЛ и КМОП *$\overline{SR}$*-защелки используются гораздо чаще, чем SR-защелки, поскольку вентили И-НЕ предпочтительнее вентилей ИЛИ-НЕ.



(a)

S_L	R_L	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	last Q	last QN

(b)



(c)

Рис. 7.9. $\overline{SR}$-защелка: (a) принципиальная схема на вентилях И-НЕ; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

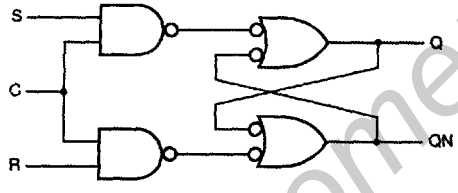
Как видно из таблицы, описывающей работу схемы, принцип действия $\overline{S}\,\overline{R}$-защелки подобен механизму функционирования SR-защелки; правда, с двумя важными отличиями. Во-первых, у сигналов $\overline{S}$ и $\overline{R}$ активным является низкий уровень, так что при $\overline{S}$=$\overline{R}$=1 защелка помнит свое предыдущее состояние; на активный низкий уровень входных сигналов указывают символы инверсии на входах в условном обозначении. Во-вторых, при одновременном наличии сигналов активного уровня на входах $\overline{S}$ и $\overline{R}$ оба выходных сигнала защелки становятся

равными 1, а не 0, как это было в SR-защелке. За исключением этих отличий, $\overline{SR}$-защелка работает точно так же, как SR-защелка, в том числе в отношении временных требований и метастабильности.

7.2.3. SR-защелка с входом разрешения

SR- и $\overline{S}\,\overline{R}$-защелки чувствительны к входным сигналам S и R в течение всего времени. Однако их легко видоизменить таким образом, чтобы схема была чувствительна к этим входным сигналам только тогда, когда подан сигнал на вход разрешения C. Такая *SR-защелка с входом разрешения* (*S-R latch with enable*) показана на рис. 7.10. Как видно из таблицы, описывающей работу схемы, при C, равном 1, данная схема ведет себя как SR-защелка, а при C, равном 0, она удерживается в прежнем состоянии. На рис. 7.11 приведены временные диаграммы, иллюстрирующие поведение этой схемы при типичном наборе входных воздействий. Если оба сигнала S и R равны 1 в момент, когда сигнал C переходит из 1 в 0, то схема ведет себя подобно SR-защелке при одновременном переходе сигналов S и R на неактивный уровень: следующее состояние непредсказуемо и выходная цепь может стать метастабильной.

(a)



(b)

S	R	C	Q	QN
0	0	1	last Q	last QN
0	1	1	0	1
1	0	1	1	0
1	1	1	1	1
x	x	0	last Q	last QN

(c)

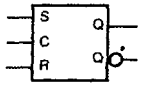


Рис. 7.10. SR-защелка с входом разрешения: (a) принципиальная схема на вентилях И-НЕ; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

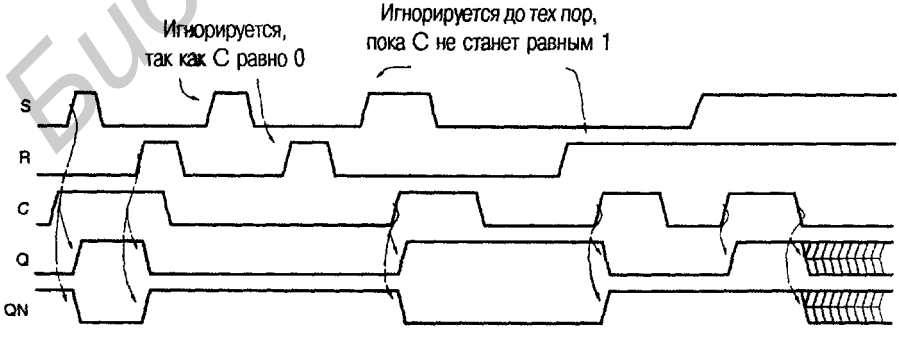


Рис. 7.11. Работа SR-защелки с входом разрешения в типичных условиях

7.2.4. D-защелка

SR-защелки полезны в разного рода управляющих устройствах, где часто возникают ситуации, когда в качестве реакции на выполнение того или иного условия нужно «выставлять флаг», а при изменении условий его сбрасывать. В таких случаях управление входами установки и сброса осуществляется в определенной степени независимо. Однако часто бывают нужны защелки, чтобы просто запомнить биты информации, когда каждый бит поступает по отдельной сигнальной линии и его надо как-то сохранить. В задачах такого рода можно воспользоваться *D-защелками* (*D latch*)

D-защелка показана на рис. 7.12. Ее схема состоит из SR-защелки с входом разрешения и дополнительного инвертора, обеспечивающего формирование входных сигналов S и R из единственного входного сигнала D (data, данные). При этом устраняется присущая SR-защелке неприятность, связанная с одновременной подачей входных сигналов S и R активного уровня. На рис. (c) управляющий входной сигнал обозначен буквой C, но иногда его называют ENABLE, CLK или G и в некоторых схемах D-защелок у этого сигнала активным является низкий уровень.

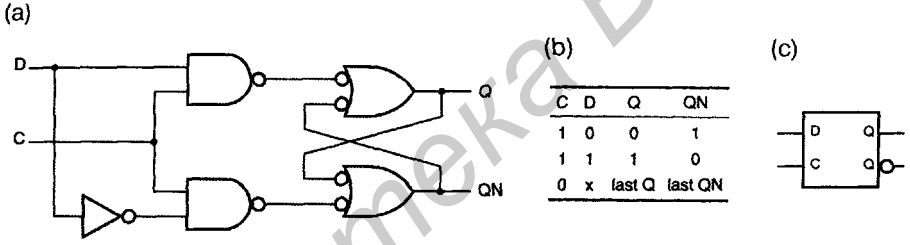


Рис. 7.12. D-защелка: (a) принципиальная схема на вентилях И-НЕ; (b) таблица, описывающая работу схемы (last – *последнее значение*); (c) *условное* обозначение

Пример функционального поведения D-защелки приведен на рис. 7.13. Когда подан входной сигнал C, выходной сигнал Q повторяет значения входного сигнала D. В этом случае говорят, что защелка «прозрачна» и путь от входа до выхода Q «открыт»; по этой причине данную схему часто называют *прозрачной защелкой* (*transparent latch*). Когда сигнал C снимается, защелка запирается; выходной сигнал Q сохраняет свое последнее значение и больше не реагирует на изменения входного сигнала D, пока C остается на неактивном уровне.

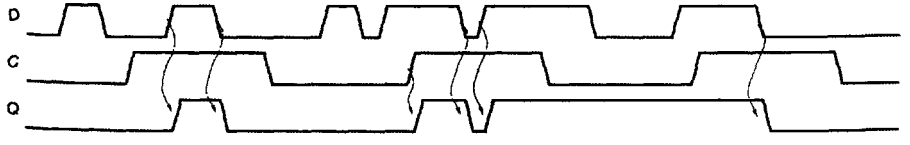


Рис. 7.13. Функциональное поведение D-защелки при различных входных сигналах

На рис. 7.14 подробнее показано, что происходит в D-защелке с течением времени. Четыре различных параметра задержки характеризуют прохождение сигналов от входов C и D до выхода Q. Например, к моментам переходов 1 и 4 защелка «заперта» и значение входного сигнала D противоположно значению выходного сигнала Q, поэтому когда C становится равным 1, защелка «отпирается» и выходной сигнал Q изменяет свое значение с задержками $t_{pLH(CQ)}$ и $t_{pHL(CQ)}$. В моменты переходов 2 и 3 входной сигнал C равен 1 и защелка открыта, так что выходной сигнал Q напрямую повторяет переходы во входном сигнале D, но с задержками $t_{pHL(DQ)}$ и $t_{pLH(DQ)}$. Четыре другие задержки, не указанные на рисунке, описывают запаздывание, с которым происходят изменения выходного сигнала QN.

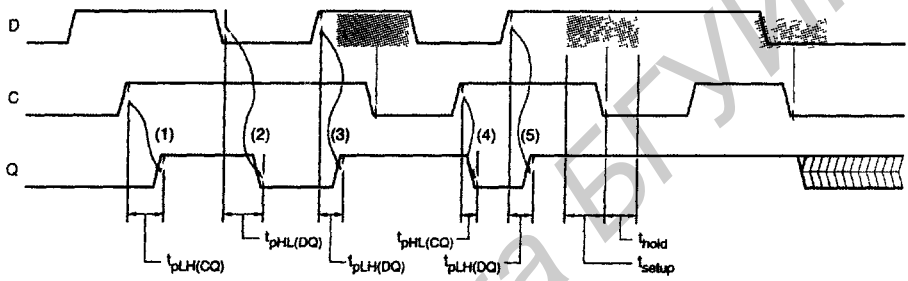


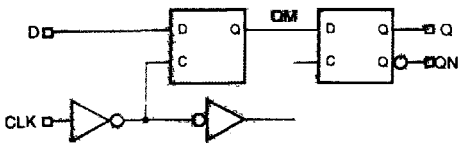
Рис. 7.14. Временные параметры D-защелки

Хотя с D-защелкой не возникает той проблемы, с какой мы сталкиваемся в случае SR-защелки, когда S = R = 1, затруднения, связанные с метастабильностью, все же остаются. Как показано на рис. 7.14, существует заштрихованный интервал времени в окрестности спадающего фронта в сигнале C, в пределах которого входной сигнал D не должен изменяться. Этот интервал начинается за время t_{setup} до спадающего фронта в сигнале C (до момента защелкивания); параметр t_{setup} называется *временем установления* (*setup time*). Заканчивается рассматриваемый интервал спустя время t_{hold}; величину t_{hold} называют *временем удержания* (*hold time*). Если входной сигнал D изменяется в какой-то момент внутри интервала, состоящего из времени установления и времени удержания, то значение сигнала на выходе защелки непредсказуемо и состояние выходной цепи может оказаться метастабильным; этот случай изображен на рисунке вслед за последним защелкивающим перепадом.

7.2.5. D-триггер, переключающийся по фронту

Показанное на рис. 7.15 объединение пары D-защелок, называемое *D-триггером, переключающимся по положительному фронту* (*positive-edge-triggered D flip-flop*), представляет собой схему, в которой опрос ее входа D и изменение ее выходных сигналов Q и QN происходит только в моменты времени, задаваемые нарастающим фронтом управляющего сигнала CLK. Первая защелка называется *ведущей* (*master*); при значении CLK, равном 0, она открыта и ее выходной сигнал повторяет входной сигнал. Когда сигнал CLK становится равным 1, ведущая защелка запирается и ее выходной сигнал переносится во вторую защелку, называемую

ведомой (*slave*). Ведомая защелка открыта в течение всего времени, пока значение CLK остается равным 1, но изменение сигнала на ее выходе возможно только в самом начале этого интервала, так как ведущая защелка заперта и сигнал на ее выходе остается неизменным на протяжении всего этого отрезка времени.



D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

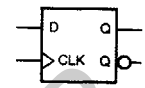


Рис. 7.15. D-триггер, переключающийся по положительному фронту: (a) принципиальная схема на D-защелках; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

Треугольник на входе CLK у D-триггера указывает на срабатывание схемы по фронту и носит название *указателя динамического входа* (*dynamic-input indicator*). Примеры функционального поведения триггера при нескольких переходах во входных сигналах приведены на рис. 7.16. Фигурирующий на этих временных диаграммах сигнал QM – это выходной сигнал ведущей защелки. Заметьте, что сигнал QM изменяется только при CLK, равном 0. Когда CLK становится равным 1, текущее значение QM переносится на выход Q, тогда как изменение сигнала QM невозможно до тех пор, пока CLK снова не станет равным 0.

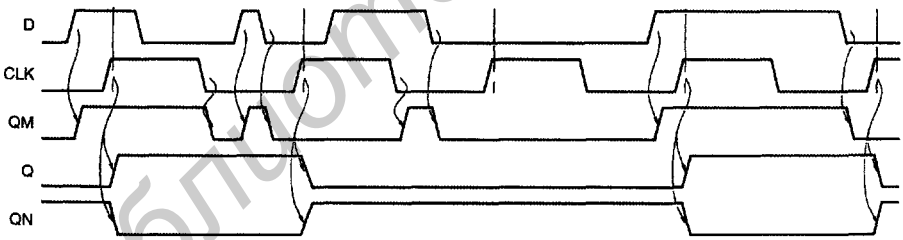


Рис. 7.16. Функциональное поведение D-триггера, переключающегося по положительному фронту

На рис. 7.17 временные зависимости сигналов в D-триггере представлены более подробно. Все задержки распространения измеряются относительно нарастающего фронта в сигнале CLK, поскольку только это событие вызывает изменение выходного сигнала. Задержки могут быть различными при переходе выходного сигнала с низкого уровня на высокий и в обратном направлении.

Так же, как и в случае D-защелки, у D-триггера есть интервал времени, состоящий из времени установления и времени удержания, в течение которого сигнал на входе D не должен изменяться. Этот интервал находится в окрестности переключающего фронта сигнала CLK и указан на рис. 7.17 цветной штриховкой. Если требования, предъявляемые временем установления и временем удержания, не удовлетворяются, то выход триггера, как правило, переходит в одно из устойчивых состояний

0 или 1, хотя предсказать, в какое именно, нельзя. Однако в некоторых случаях на выходе могут возникнуть колебания, либо выходная цепь может перейти в метастабильное состояние посередине между 0 и 1, как показано на рисунке после предпоследнего такта в управляющем сигнале. Если триггер попадает в метастабильное состояние, то со случайной задержкой он, в конце концов, сам вернется в одно из своих устойчивых состояний; подробнее это обстоятельство разбирается в параграфе 8.9. Можно принудительно перевести триггер в устойчивое состояние в момент действия следующего переключающего фронта в тактовом сигнале, по отношению к которому сигнал на входе D удовлетворяет требованию не изменяться в пределах времени установления и времени удержания; этот случай изображен на рис. 7.17, где он приходится на последний такт в управляющем сигнале.

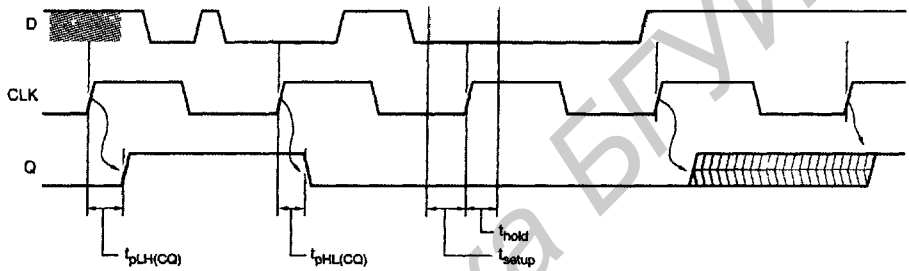
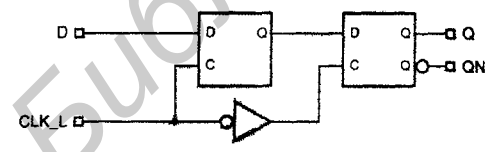


Рис. 7.17. Временные зависимости сигналов в D-триггере, переключающемся по положительному фронту

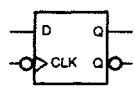
Чтобы получить *D-триггер, переключающийся по отрицательному фронту* (*negative-edge-triggered D flip-flop*), достаточно инвертировать тактовый входной сигнал, так что все переключения будут происходить на спадающем фронте сигнала CLK_L; при срабатывании по спадающему фронту принято считать активным низкий уровень сигнала. Таблица, описывающая работу такого триггера, и его условное обозначение приведены на рис. 7.18.



(a)

D	CLK_L	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN

(b)



(c)

Рис. 7.18. D-триггер, переключающийся по отрицательному фронту: (a) принципиальная схема на D-защелках; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

У некоторых D-триггеров бывают *асинхронные входы* (*asynchronous inputs*), воздействуя на которые можно переводить триггер в то или другое состояние независимо от сигналов на входах CLK и D. Обычно эти входы обозначаются PR (*preset*, установка в единичное состояние) и CLR (*clear*, сброс); по отношению к сигналам на этих входах D-триггер ведет себя подобно тому, как SR-защелка реагирует на сигналы на входе установки в единичное состояние и входе сброса. Услов-

ное обозначение переключающегося по фронту D-триггера с такими входами и его принципиальная схема на вентилях И-НЕ приведены на рис. 7.19. Хотя некоторые разработчики логических схем и используют асинхронные входы для реализации хитрых последовательностных функций, лучше всего сохранить их для целей тестирования и инициализации, то есть для установки последовательностной схемы в заданное начальное состояние; подробнее эти вопросы рассмотрены в параграфе 8.7 в связи с методологией синхронного проектирования.

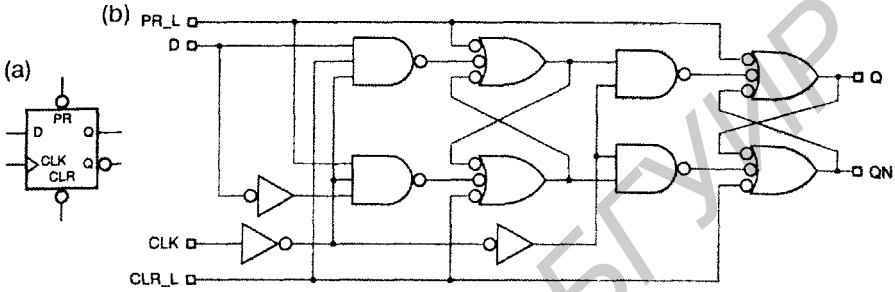


Рис. 7.19. D-триггер, переключающийся по положительному фронту, с входами установки и сброса: (a) условное обозначение; (b) принципиальная схема на вентилях И-НЕ

КАК УСТРОЕНЫ «НАСТОЯЩИЕ» ТРИГГЕРЫ

Серийно производимые в семействах ТТЛ D-триггеры, переключающиеся по положительному фронту, устроены не по принципу «ведущий–ведомый», на основе которого действуют схемы, приведенные на рис. 7.15 и 7.19. Вместо этого в триггерах типа 74LS74 реализована схема на 6 вентилях (рис. 7.20), которая меньше по объему и быстрее. В параграфе 7.9 мы покажем, как формально проводится анализ переходов в каждом случае.

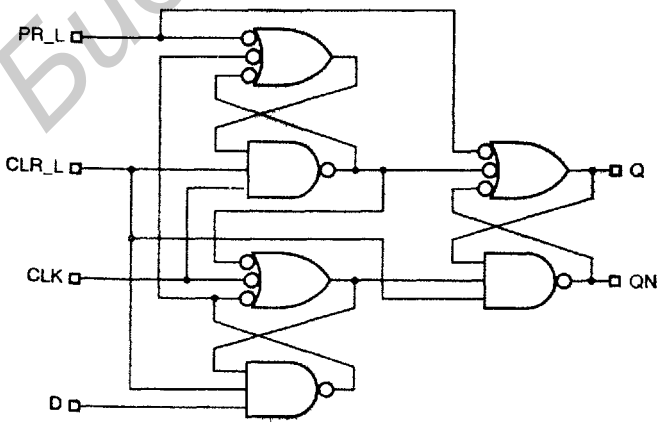
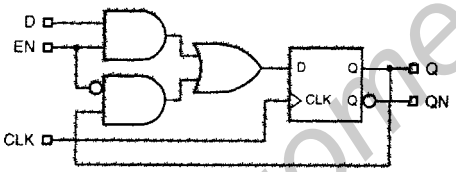


Рис. 7.20. Серийная схема D-триггера, переключающегося по положительному фронту, типа 74LS74

7.2.6. Переключающийся по фронту D-триггер с входом разрешения

Обычно бывает желательным, чтобы выполняемая D-триггерами функция состояла в удержании последнего запомненного значения, а не в загрузке нового значения на каждом такте управляющего сигнала. Это реализуется путем добавления *входа разрешения* (*enable input*), обозначаемого как EN или CE (*clock enable, разрешение тактового сигнала*). Название «разрешение тактового сигнала» является описательным; оно вовсе не означает, что функция дополнительного входа состоит в пропускании или непропускании тактового сигнала. Напротив, как показано на рис. 7.21(a), с помощью 2-входового мультиплексора выбирается значение, подаваемое на внутренний D-вход триггера. Когда действует разрешающее значение сигнала EN, выбирается сигнал, поступающий на внешний D-вход; когда сигнал EN переходит на неактивный уровень, через мультиплексор проходит текущее значение сигнала на выходе триггера. В результате таблица, описывающая работу схемы, имеет вид, представленный на рис. (b). Условное обозначение такого триггера указано на рис. (c); у некоторых триггеров активным является низкий уровень сигнала на входе разрешения, что находит свое отражение в помещении символа инверсии на этом входе.

(a)



(b)

D	EN	CLK	Q	QN
0	1	↑	0	1
1	1	↑	1	0
x	0	↑	last Q	last QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN

(c)

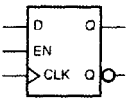


Рис. 7.21. Переключающийся по положительному фронту D-триггер с входом разрешения: (a) принципиальная схема; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

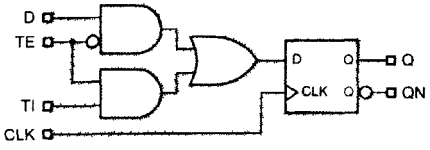
7.2.7. Тестируемый триггер

При тестировании специализированных ИС важным свойством триггера является так называемая *возможность опроса* (*scan capability*). Идея состоит в том, чтобы во время тестирования подавать на D-вход триггера данные от альтернативного источника. Когда все триггеры в данной специализированной ИС переведены в режим тестирования, в них – через альтернативные входы данных – «загружается» тестовая комбинация. После этого триггеры возвращаются в «нормальный» режим работы, и в этом режиме на все триггеры обычным образом подается тактовый сигнал. После одного или нескольких тактов, триггеры снова переводятся в режим тестирования, и результаты тестирования «считываются».

На рис. 7.22 показан типичный тестируемый триггер. Все, что есть в данной схеме, – это D-триггер с 2-входовым мультиплексором на входе D. Когда сигнал TE на *входе разрешения тестирования* (*test enable input*) имеет неактивное зна-

чение, схема ведет себя как обычный D-триггер. Когда же подан сигнал TE, разрешающий тестирование, триггер берет данные с входа TI (*test input, тестовый вход*), а не с входа D. Таблица, описывающая работу схемы, представлена на рис. (b), а условное обозначение данного устройства – на рис. (c).

(a)



(b)

TE	TI	D	CLK	Q	QN
0	x	0	↑	0	1
0	x	1	↑	1	0
1	0	x	↑	0	1
1	1	x	↑	1	0
x	x	x	0	last Q	last QN
x	x	x	1	last Q	last QN

(c)

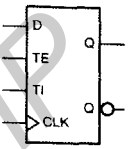


Рис. 7.22. Тестируемый D-триггер, переключающийся по положительному фронту: (a) принципиальная схема; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

Дополнительные входы используются для соединения всех триггеров в специализированной ИС с целью тестирования в одну *цепочку сканирования* (*scan chain*).

На рис. 7.23 приведен простой пример цепочки сканирования, состоящей из четырех триггеров. Входы TE всех триггеров объединяются вместе в один глобальный вход TE, тогда как выход Q каждого триггера соединяется с входом TI другого триггера, образуя последовательную цепочку (daisy chain). Соединения, относящиеся ко входам TI и TE, а также к выходу TO (*test output, тестовый выход*), предназначены исключительно для целей тестирования; другие соединения, относящиеся ко входам D и выходам Q, необходимые для того чтобы схема в целом могла делать что-то полезное, на рисунке не показаны.

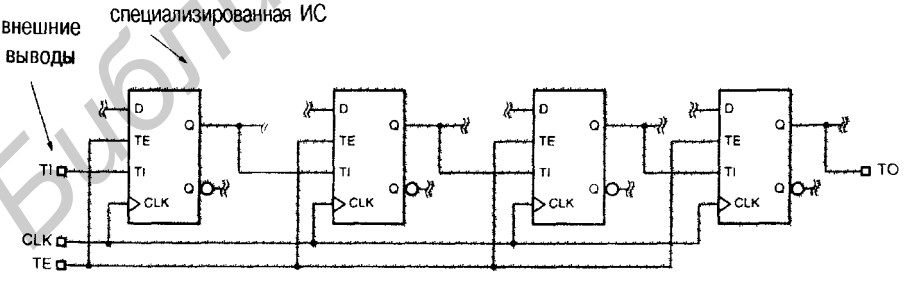


Рис. 7.23. Цепочка сканирования из четырех триггеров

Для того чтобы протестировать схему, в том числе и основную логику, разрешающий сигнал удерживается на глобальном входе TE в течение *n* периодов тактового сигнала, в то время как через глобальный вход TI в *n* триггеров посредством сдвига загружается *n*-разрядный тестовый вектор; на рис. 7.23 число *n* равно 4. Затем сигнал TE снимается и схеме предоставляется возможность функционировать в течение одного или нескольких следующих тактов. Новое состояние схемы,

представленное новыми значениями сигналов на выходах *n* триггеров, можно считать, наблюдая сигнал на выходе TO в течение очередных *n* тактов при условии, что на входе TE действует разрешающий сигнал. После того как результаты одного тестирования прочитаны, можно загрузить другой проверочный вектор; такая возможность делает процесс тестирования более эффективным.

Существует столько различных типов тестируемых триггеров, сколько имеется различных способов реализации самих триггеров. Например, возможностью тестирования мог бы быть наделен D-триггер с входом разрешения, изображенный на рис. 7.21, путем замены его 2-входового внутреннего мультиплексора на 3-входовой. Тогда, в зависимости от значений сигналов EN и TE, в триггеры на каждом такте записывались бы сигналы с входов D или TI или его собственное текущее состояние. Возможность тестирования можно добавить также и в случае триггеров других типов, в частности JK-триггеров и T-триггеров, о которых речь пойдет позднее в этом параграфе.

*7.2.8. Двухтактный SR-триггер

Как указывалось ранее, SR-защелки полезны в «управляющих устройствах», где возможны независимые условия установки и снятия контрольного бита. Если предполагается, что контрольный бит должен меняться только в определенные моменты времени, привязанные к тактовому сигналу, то нужен SR-триггер, подобный D-триггеру, у которого сигналы на выходах изменялись бы только на определенном перепаде тактового сигнала. В этом и в двух следующих разделах описаны триггеры, полезные при решении таких задач.

Если D-защелки в D-триггере, переключающемся по отрицательному фронту [рис. 7.18(а)], заменить SR-защелками, то получим *двухтактный SR-триггер* (*master/slave S-R flip-flop*; триггер, действующий по принципу ведущий–ведомый), показанный на рис. 7.24. Аналогично D-триггеру, выходные сигналы SR-триггера изменяются только по спадающему фронту в тактовом сигнале C. Однако новое значение выходного сигнала зависит теперь не от значений входных сигналов точно в тот момент времени, на который приходится спадающий фронт, а от значений входных сигналов на протяжении всего интервала времени, в течение которого сигнал C был равен 1 перед отрицательным перепадом. Как следует из временных диаграмм на рис. 7.25, короткий импульс S в любом месте в пределах указанного интервала времени может установить ведущую защелку в единичное состояние; точно так же импульс на входе R может сбросить ее. Значение, переносимое на выход триггера в момент действия спадающего фронта в сигнале C, зависит от того, каким было последнее состояние ведущей защелки, пока сигнал C оставался равным 1.

Согласно рис. 7.24(с), в условном обозначении двухтактного SR-триггера не используется указатель динамического входа, поскольку этот триггер не является истинно переключающимся по фронту. Он скорее похож на защелку, в которой происходит повторение ее входного сигнала в течение интервала времени, на котором C равняется 1, но изменения сигнала на выходе триггера отражают конечное защелкнутое значение только тогда, когда C становится равным 0. На тот факт, что выходной сигнал не изменяется до тех пор, пока сигнал на входе разрешения C не перейдет на неактивный уровень, указывает *индикатор задержки*

Рис. 7.24. Двухтактный SR-триггер: (a) схема на SR-защелках; (b) таблица, описывающая работу схемы (last – последнее значение, undef. – неопределенное значение); (c) условное обозначение

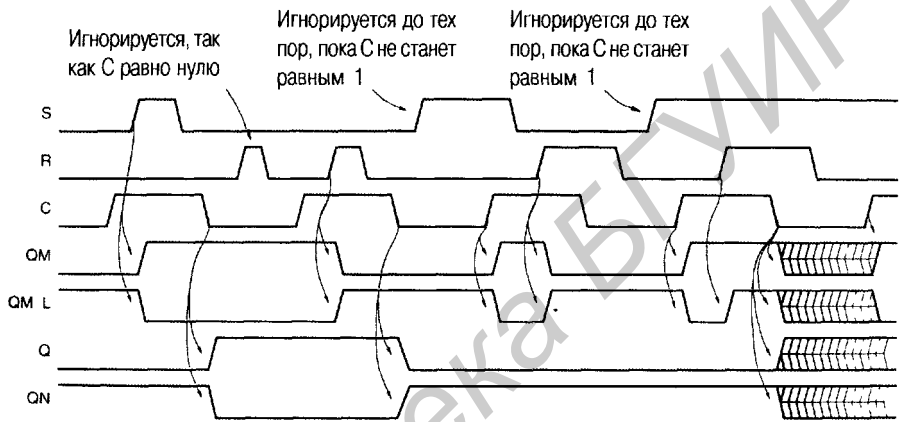


Рис. 7.25. Временные диаграммы, иллюстрирующие работу двухтактного SR-триггера

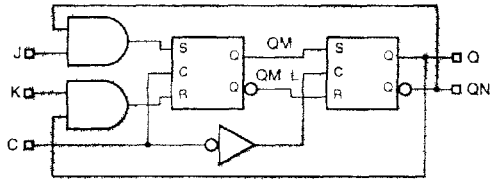
Работа двухтактного SR-триггера оказывается непредсказуемой, если в момент отрицательного перепада в сигнале C оба входных сигнала S и R имеют активный уровень. В этом случае к моменту действия спадающего фронта оба выходных сигнала ведущей защелки Q и QN равны 1. Когда сигнал C становится равным 0, значения сигналов на выходах ведущей защелки предсказать нельзя; более того, ее выходная цепь может оказаться метастабильной. В то же самое время ведомая защелка открывается и весь этот «мусор» проходит на выход триггера.

*7.2.9. Двухтактный JK-триггер

Проблема с тем, что делать, когда одновременно действуют сигналы S и R, решается в *двухтактном JK-триггере* (*master/slave J-K flip-flop*). Входы J и K аналогичны входам S и R. Однако, как видно из рис. 7.26, сигнал J попадает на вход S ведущей защелки только в том случае, когда текущее значение сигнала на выходе триггера QN равно 1 (то есть сигнал Q равен 0), а сигнал K попадает на

вход R ведущей защелки только тогда, когда текущее значение Q, равно 1. Таким образом, при одновременном действии сигналов на входах J и K триггер переходит в состояние, противоположное тому, в котором он находится.

(a)



(b)

J	K	C	Q	QN
x	x	0	last Q	last QN
0	0	⎍	last Q	last QN
0	1	⎍	0	1
1	0	⎍	1	0
1	1	⎍	last QN	last Q

(c)

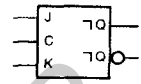


Рис. 7.26. Двухтактный JK-триггер (a) схема на SR-защелках; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

Временные диаграммы на рис. 7.27 иллюстрируют работу двухтактного JK-триггера при типичном наборе входных сигналов. Заметьте, что для изменения сигнала на выходе триггера в момент окончания переключающего импульса нет необходимости подавать входные сигналы J и K именно в этот момент времени. Действительно, коль скоро только один из входных сигналов проходит через вентили на входы S и R ведущей защелки, сигнал на выходе триггера может стать равным 1 даже в том случае, когда к концу переключающего импульса действует только сигнал K, а сигнал J снят. Такое поведение, известное под названием *захвата единиц* (*1s catching*), продемонстрировано на рисунке: оно имеет место на предпоследнем переключающем импульсе. Аналогичное явление, называемое *захватом нулей* (*0s catching*), можно наблюдать на последнем переключающем импульсе. Из-за этого свойства двухтактного JK-триггера входные сигналы J и K должны оставаться неизменными в течение всего интервала времени, пока C равно 1.

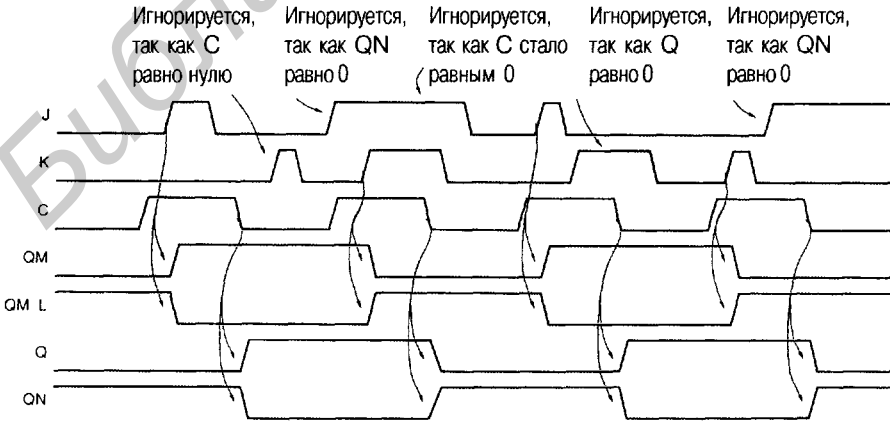
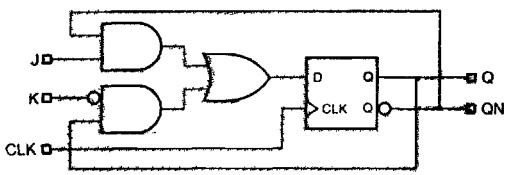


Рис. 7.27. Временные диаграммы, иллюстрирующие работу двухтактного JK-триггера

7.2.10. JK-триггер, переключающийся по фронту

Проблема захвата единиц и нулей решена в *JK-триггере, переключающемся по фронту* (*edge-triggered J-K flip-flop*), функциональный эквивалент которого изображен на рис. 7.28. Внутри него используется переключающийся по фронту D-триггер, благодаря чему входы JK-триггера опрашиваются на нарастающем фронте тактового сигнала и очередное значение выходного сигнала вырабатывается в соответствии с «характеристическим уравнением»: $Q^* = J \cdot Q' + K' \ Q$ (см.раздел 7.3.3).

(a)



(b)

J	K	CLK	Q	QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN
0	0	↓	last Q	last QN
0	1	↓	0	1
1	0	↓	1	0
1	1	↓	last QN	last Q

(c)

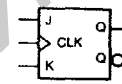


Рис. 7.28. JK-триггер, переключающийся по фронту: (a) функциональный эквивалент на переключающемся по фронту D-триггере; (b) таблица, описывающая работу схемы (last – последнее значение); (c) условное обозначение

Типичное поведение переключающегося по фронту JK-триггера представлено на рис. 7.29. Так же, как и в случае сигнала на входе D переключающегося по фронту D-триггера, для надлежащей работы JK-триггера его входные сигналы J и K в окрестности переключающего фронта тактового сигнала должны удовлетворять требованиям, вытекающим из известных значений времени установления и времени удержания.

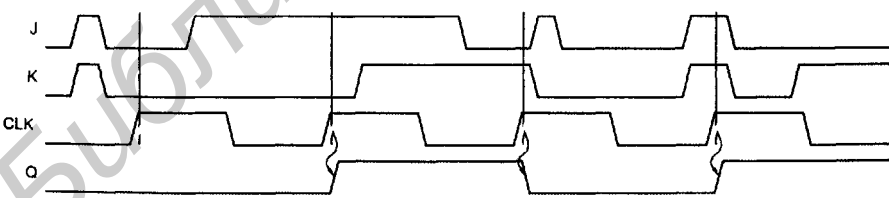


Рис. 7.29. Поведение JK-триггера, переключающегося по положительному фронту

Поскольку в JK-триггерах, переключающихся по фронту, исключены проблемы захвата единиц и нулей и проблемы, связанные с одновременным действием управляющих входных сигналов, эти триггеры по существу вытеснили переключающиеся по фронту триггеры более старых типов. Микросхемы *74x109* в семействе ТТЛ представляют собой $J\overline{K}$-триггеры, переключающиеся по положительному фронту, с активным низким уровнем входного сигнала K (который называется $\overline{K}$ или K_L).

ЕЩЕ ОДИН «НАСТОЯЩИЙ» ТРИГГЕР

По своей внутренней структуре ИС 74x109 очень похожа на ИС 74LS74, схема которой была приведена на рис. 7.20. Как следует из сравнения схем на рис. 7.20 и 7.30, микросхема '109 получается из микросхемы '74 простой заменой нижнего левого вентиля, реализующего характеристическое уравнение Q* = D, на структуру И–ИЛИ, реализующую характеристическое уравнение JK-триггера: $Q^* = J \cdot Q' + K\_L \cdot Q$.

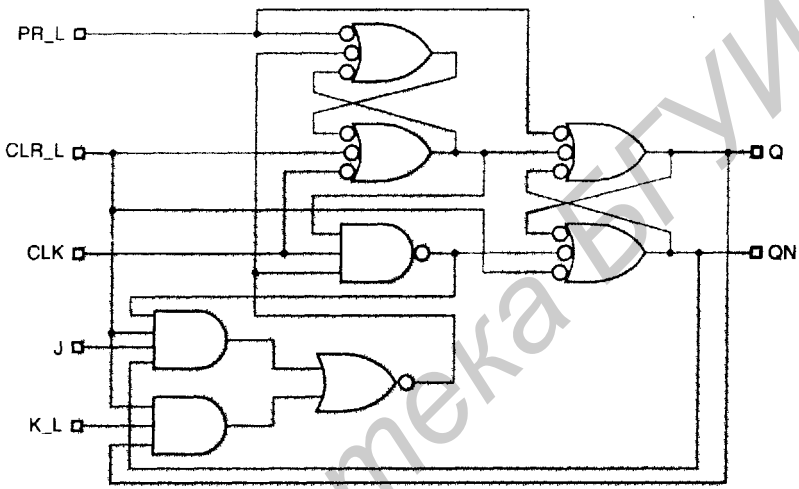


Рис. 7.30. Внутреннее устройство переключающегося по положительному фронту JK-триггера 74LS109

Самым распространенным применением JK-триггеров являются тактируемые синхронные конечные автоматы. Как мы объясним в разделе 7.4.5, логика переходов у JK-триггеров иногда проще, чем у D-триггеров. Однако в большинстве случаев конечные автоматы строятся на D-триггерах, поскольку методология проектирования при этом чуть проще и большинство проектируемых последовательностных устройств состоит из D-триггеров, а не из JK-триггеров. Поэтому в основном мы будем обращать внимание на D-триггеры.

7.2.11. T-триггер

T-триггер (*T flip-flop*; T – toggle, переключать) изменяет свое состояние на каждом периоде тактового сигнала. На рис. 7.31 приведены условное обозначение и временные диаграммы для переключающегося по положительному фронту T-триггера. Заметьте, что частота переключений сигнала на выходе триггера Q равна точно половине частоты переключений входного сигнала T. На рис. 7.32 показано, как получить T-триггер из D- или JK-триггера. Как мы увидим в параграфе 8.4, T-триггеры чаще всего используются в счетчиках и делителях частоты.

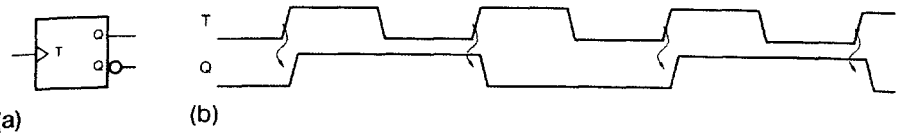


Рис. 7.31. Т-триггер, переключающийся по положительному фронту: (a) условное обозначение; (b) функциональное поведение

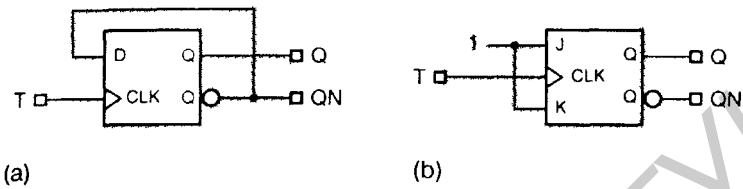


Рис. 7.32. Возможные схемы Т-триггеров (a) на основе D-триггера; (b) на основе JK-триггера

Во многих приложениях Т-триггеров требуется, чтобы такой триггер переключался не на каждом такте. В таком случае можно воспользоваться *Т-триггером с входом разрешения* (*T flip-flop with enable*). Как показано на рис. 7.33, такой триггер изменяет свое состояние на переключающем фронте тактового сигнала только в том случае, когда на вход EN подан разрешающий сигнал. Подобно сигналам на входах D, J и K у других триггеров, переключающихся по фронту, сигнал на входе EN должен удовлетворять требованию оставаться неизменным в интервале, состоящем из времени установления и времени удержания, в окрестности переключающего фронта тактового сигнала. Схемы, приведенные на рис.7.32, легко видоизменить таким образом, чтобы имелся вход разрешения EN, как это сделано на рис. 7.34.

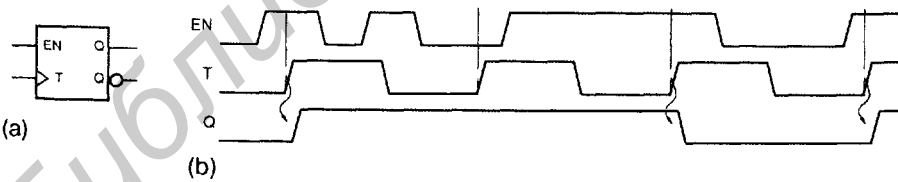


Рис. 7.33. Переключающийся по положительному фронту Т-триггер с входом разрешения: (a) условное обозначение; (b) функциональное поведение

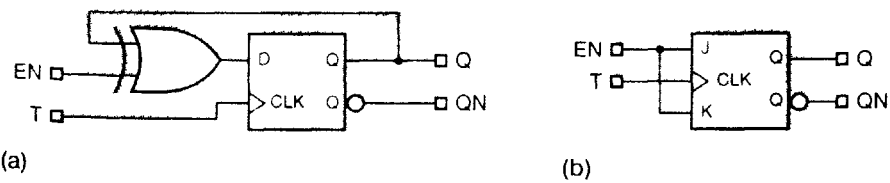


Рис. 7.34. Возможные схемы Т-триггеров с входом разрешения: (a) на основе D-триггера, (b) на основе JK-триггера

*7.6. Синтез конечных автоматов на основе списка переходов

Построением диаграммы состояний автомата и выбором способа кодирования, по существу, исчерпывается творческая часть процесса проектирования. Остальную часть процедуры синтеза можно выполнить с помощью программ системы CAD.

Как показано в предыдущем параграфе список переходов составляется по диаграмме состояний автомата с учетом выбранных кодов состояний. В этом параграфе мы продемонстрируем, как по списку переходов синтезируется сам конечный автомат. Подробно рассматривается ряд возможностей и нюансов, возникающих при проектировании конечного автомата на основе списка переходов. Хотя материал этого параграфа и полезен для синтезирования автоматов вручную, его главное назначение – помочь вам понять логику работы и возможные капризы программ и языков, используемых в системах CAD при конструировании конечных автоматов.

*7.6.1. Уравнения переходов

Первый шаг при синтезе конечного автомата по списку переходов заключается в выводе системы уравнений переходов, которыми задаются следующие значения V* всех переменных состояния как функции текущего состояния и входного воздействия. Список переходов можно считать своего рода гибридной таблицей истинности, в которой комбинации переменных состояния перечислены явно, а комбинации входных сигналов приведены в алгебраической форме. Двигаясь сверху вниз по столбцу V* в списке переходов, мы получаем последовательность нулей и единиц, то есть значений V* для различных комбинаций состояние/вход (при условии, конечно, что мы все записали в списке переходов правильно).

Выражение для следующего значения переменной состояния V* в уравнении переходов может быть разновидностью гибридной канонической суммы:

$$V^* = \sum_{\text{по всем строкам списка переходов, в которых } V^* = 1} \text{(р-терм перехода)}.$$

Другими словами, в уравнение переходов входит по одному «р-терму перехода» на каждую строку списка переходов, содержащую 1 в столбце V*. *р-терм перехода* (*transition p-term*) данной строки – это произведение минтерма текущего состояния и выражения перехода.

Согласно списку переходов в табл. 7.17 уравнение переходов для переменной Q2* автомата, управляющего задними огнями автомобиля марки Ford Thunderbird, можно записать в виде суммы, состоящей из восьми р-термов:

$$\begin{aligned} Q2^* &= Q2' \cdot Q1' \cdot Q0' \cdot (HAZ + LEFT \cdot RIGHT) \\ &+ Q2' \cdot Q1' \cdot Q0' \cdot (RIGHT \cdot HAZ' \cdot LEFT') \\ &+ Q2' \cdot Q1' \cdot Q0 \cdot (HAZ) \\ &+ Q2' \cdot Q1 \cdot Q0 \cdot (HAZ) \\ &+ Q2 \cdot Q1' \cdot Q0 \cdot (HAZ') \\ &+ Q2 \cdot Q1' \cdot Q0 \cdot (HAZ) \\ &+ Q2 \cdot Q1 \cdot Q0 \cdot (HAZ') \\ &+ Q2 \cdot Q1 \cdot Q0 \cdot (HAZ). \end{aligned}$$

Непосредственными алгебраическими преобразованиями это соотношение упрощается в результате объединения первых двух, следующих двух и последних четырех р-термов:

$$\begin{aligned} Q2^* &= Q2' \cdot Q1' \cdot Q0' \cdot (HAZ + RIGHT) \\ &+ Q2' \cdot Q0 \cdot (HAZ) \\ &+ Q2 \cdot Q0. \end{aligned}$$

Аналогично могут быть получены уравнения переходов для Q1* и Q0*:

$$\begin{aligned} Q1^* &= Q2' \cdot Q1' \cdot Q0' \cdot (HAZ') \\ &+ Q2' \cdot Q1 \cdot Q0 \cdot (HAZ') \\ &+ Q2 \cdot Q1' \cdot Q0 \cdot (HAZ') \\ &+ Q2 \cdot Q1 \cdot Q0 \cdot (HAZ') \\ &= Q0 \cdot HAZ' \\ Q0^* &= Q2' \cdot Q1' \cdot Q0' \cdot (LEFT \cdot HAZ' \cdot RIGHT') \\ &+ Q2' \cdot Q1' \cdot Q0' \cdot (RIGHT \cdot HAZ' \cdot LEFT') \\ &+ Q2' \cdot Q1' \cdot Q0 \cdot (HAZ') \\ &+ Q2 \cdot Q1' \cdot Q0 \cdot (HAZ') \\ &= Q2' \cdot Q1' \cdot HAZ' (LEFT \oplus RIGHT) + Q1' \cdot Q0 \cdot HAZ'. \end{aligned}$$

За исключением переменной Q1*, нет никаких гарантий, что найденные соотношения для переходов являются в том или ином смысле минимальными: действительно, выражения для Q2* и Q0* не имеют даже стандартного вида «суммы произведений» или «произведения сумм». Упрощенные или неупрощенные уравнения служат лишь четко сформулированной отправной точкой, чтобы можно было выбрать какой-нибудь комбинационный метод синтеза логики возбуждения в конечном автомате: на основе структуры И-НЕ–И-НЕ, на ИС средней степени интеграции или как-то еще. При разработке на основе ПЛУ вы могли бы просто вставить эти уравнения в программу на языке ABEL и велеть компьютеру найти

минимальные выражения вида «сумма произведений» для реализации на решетке И–ИЛИ в ПЛУ.

*7.6.2. Уравнения возбуждения

Несмотря на то, что мы уже подошли к логике возбуждения, до сих пор нами выведены только *уравнения переходов*, но не *уравнения возбуждения*. Однако в случае, когда в качестве элементов памяти в наших конечных автоматах применяются D-триггеры, уравнения возбуждения являются тривиальным следствием уравнений переходов, поскольку характеристическое уравнение D-триггера имеет вид: Q* = D. Поэтому из уравнения переходов для переменной состояния Qi*

Qi* = выражение

получаем следующее уравнение возбуждения для сигнала на входе соответствующего D-триггера:

Di = выражение.

Рациональные уравнения возбуждения для триггеров других типов, особенно для JK-триггеров, выводятся не так легко (см. задачу 7.63). По этой причине в подавляющем большинстве случаев в конечных автоматах, создаваемых на дискретных компонентах, на основе ПЛУ или специализированных ИС, используются D-триггеры.

*7.6.3. Варианты схем

Существуют и другие способы получения уравнений переходов и уравнений возбуждения из списка переходов. Если столбец со следующими значениями какой-то одной переменной состояния содержит меньше нулей, чем единиц, то имеет смысл записать уравнение переходов для этой переменной в терминах нулей в этом столбце:

$$V^{*\prime} = \sum_{\text{по всем строкам списка переходов, в которых } V^* = 0} \text{(р-терм перехода)}.$$

Другими словами, V*′ равно 1 для всех р-термов, для которых V* равно 0. Следовательно, уравнения переходов для Q2*′ можно записать в виде суммы, состоящей из семи р-термов:

$$\begin{aligned} Q2^{*\prime} &= Q2' \text{ Ч } Q1' \cdot Q0' \cdot ((LEFT + RIGHT + HAZ)') \\ &+ Q2' \cdot Q1' \cdot Q0' \cdot (LEFT \cdot HAZ' \cdot RIGHT') \\ &+ Q2' \cdot Q1' \cdot Q0 \cdot (HAZ') \\ &+ Q2' \cdot Q1 \cdot Q0 \cdot (HAZ') \\ &+ Q2' \cdot Q1 \cdot Q0' \cdot (1) \\ &+ Q2 \cdot Q1 \cdot Q0' \cdot (1) \\ &+ Q2 \;\; Q1' \cdot Q0' \cdot (1) \\ &= Q2' \cdot Q1' \cdot Q0' \cdot HAZ' \cdot RIGHT' + Q2' \cdot Q0 \cdot HAZ' + Q1 \cdot Q0' + Q2 \;\; Q0'. \end{aligned}$$

Чтобы придти к уравнению для Q2*, достаточно просто взять дополнения обеих частей последнего, свернутого равенства.

Выражение для следующего значения переменной состояния V* можно получить из нулей в списке переходов непосредственно, беря дополнение правой части общего выражения для V*′, тогда на основании теоремы Де Моргана приходим к разновидности гибридного канонического произведения:

$$V^* = \prod_{\text{по всем строкам списка переходов, в которых } V^* = 0} (\text{s-терм перехода}).$$

Здесь *s-терм перехода* (*transition s-term*) данной строки – это сумма макстерма текущего состояния и дополнения выражения перехода. Если выражение перехода является простым термом-произведением, то его дополнение есть сумма, и поэтому переменная V* в уравнении переходов представляется в виде произведения сумм.

*7.6.4. Реализация конечного автомата

После того как уравнения возбуждения для конечного автомата получены, все, что нам остается, – это решить задачу построения комбинационной логики с несколькими выходами. Конечно, реализовать комбинационную логику, представленную уравнениями, можно многими способами, но простейший из них состоит в том, чтобы набрать соответствующую программу на языке ABEL или VHDL и воспользоваться компилятором для синтеза схемы в ПЛУ, в ИС типа FPGA или в специализированной ИС.

В комбинационных ПЛУ типа PAL16L8 и GAL16V8, рассмотренных нами в параграфе 5.3, можно реализовать уравнения возбуждения с числом входов, выходов и термов-произведений, не превосходящим определенного значения. Лучше все же воспользоваться последовательностными ПЛУ, которые будут представлены в параграфе 8.3: в них на одном кристалле имеются и D-триггеры и комбинационная решетка И–ИЛИ. При одном и том же числе выводов, используемых в качестве входов и выходов, в последовательностных ПЛУ можно реализовать конечные автоматы большего объема, нежели в эквивалентных комбинационных ПЛУ, поскольку не нужно выводить сигналы возбуждения за пределы кристалла. В разделе 9.1.3 мы покажем, как реализуется автомат, управляющий задними огнями автомобиля марки Ford Thunderbird, в последовательностном ПЛУ.

7.12. Особенности проектирования последовательностных схем на языке VHDL

Бóльшая часть средств, предоставляемых в языке VHDL и используемых при проектировании последовательностных схем, уже была введена нами в параграфе 4.7, в частности, процессы, и мы пользовались этими средствами в параграфах главы 5, связанных с употреблением языка VHDL. В данном параграфе мы познакомим вас еще с несколькими возможностями и приведем простые примеры того, как ими воспользоваться. Примеры проектирования более сложных схем будут даны в относящихся к языку VHDL параграфах главы 8.

7.12.1. Последовательностные схемы с обратной связью

Фундамент для работы с последовательностными схемами с обратной связью средствами языка VHDL образуют VHDL-процессы и механизм списка событий моделирующей программы. Напомним, что состояние последовательностной схемы с обратной связью может измениться под воздействием входных сигналов, и переход в новое состояние проявляется в изменениях, распространяющихся по петле обратной связи до тех пор, пока в петле не наступит стабилизация. При моделировании переход от одного состояния к другому сопровождается занесением изменений сигналов в список событий и составлением расписания, по которому процессы запускаются вновь с элементарным сдвигом по времени; при этом изменения сигналов продолжаются до тех пор, пока их список не будет исчерпан.

В табл. 7.36 приведена VHDL-программа для SR-защелки. Структура содержит два параллельных оператора присваивания, каждый из которых запускает процесс, как это было объяснено в разделе 4.7.9. Взаимодействие этих процессов реализует одиночную процедуру защелкивания в SR-защелке.

Табл. 7.36. Потоковая VHDL-программа для SR-защелки

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vsrlatch is
  port (S, R: in STD_LOGIC;
        Q, QN: buffer STD_LOGIC );
end Vsrlatch;

architecture Vsrlatch_arch of Vsrlatch is
begin
  QN <= S nor Q;
  Q  <= R nor QN;
end Vsrlatch_arch;
```

Моделирование в среде VHDL является достаточно точным, чтобы справиться со случаем, когда одновременно поданы оба сигнала S и R. Самый интересный результат моделирования получается тогда, когда сигналы S и R снимаются одновременно. В первом из замечаний в разделе 7.2.1, вынесенных за пределы основного текста, уже объяснялось, что в этой ситуации в реальной SR-защелке могут начаться колебания, либо она может войти в метастабильное состояние. При моделировании это приведет к потенциально бесконечному циклу, в котором каждое исполнение одного из операторов присваивания будет запускать очередное исполнение другого. После некоторого числа повторений хорошее средство моделирования «раскусит» проблему – число элементарных сдвигов по времени растет, а время в модели стоит на месте – и остановит процесс моделирования.

НЕ ВОСПОЛЬЗОВАТЬСЯ ЛИ НАМ СИГНАЛОМ 'U'?

Конечно, было бы замечательно, если бы в модели SR-защелки, представленной в табл. 7.36, при одновременном переходе сигналов S и R на неактивный уровень вырабатывался выходной сигнал 'U', но ведь *этого* нет. Однако язык VHDL является достаточно мощным, чтобы разработчик, имеющий опыт работы с VHDL, мог легко описать модель, обладающую таким свойством. В подобной модели надо было бы воспользоваться средствами языка VHDL, позволяющими имитировать течение времени (мы не рассматриваем эти средства), чтобы учесть «время восстановления» защелки (см. второе замечание в разделе 7.2.1, вынесенное за пределы основного текста) и вырабатывать сигнал на выходе 'U', если второе изменение во входных сигналах происходит слишком близко по времени. Таким способом можно смоделировать даже максимально допустимое разрешенное время пребывания в состоянии метастабильности.

Заметьте, что в случае, когда у схемы есть возможность попасть в метастабильное состояние, нет гарантии, что моделирующая программа обнаружит это, особенно в больших проектах. Лучший способ избежать каких бы то ни было проблем с метастабильностью при проектировании систем заключается в ясном задании и защите асинхронных входов в соответствии с тем, как это обсуждается в параграфе 8.9.

7.12.2. Тактируемые схемы

На практике большинство устройств, проектируемых и моделируемых в среде VHDL, представляют собой тактируемые синхронные системы, в которых используются переключающиеся по фронту триггеры. В дополнение к тому, что вам уже известно о возможностях языка VHDL, для описания переключающегося по фронту элемента нам понадобится еще средство, а именно – *признак* `event` (`event` *attribute*), который можно присоединить к имени сигнала, чтобы получить переменную типа `boolean`, принимающую значение `true`, если то или иное событие в сигнале запускает объемлющий процесс в текущем цикле моделирования, и значение `false` – в противном случае.

Используя признак `event`, можно смоделировать поведение переключающегося по положительному фронту D-триггера с асинхронным входом сброса так, как это сделано в табл. 7.37. Здесь асинхронный сигнал CLR на входе сброса преобладает над тактовым входным сигналом CLK и поэтому проверяется первым в предложении "`if`". Только тогда, когда сигнал на входе CLR имеет неактивный уровень, вступает в действие то, что предусмотрено предложением "`elsif`", и имеющиеся в нем операторы исполняются по положительному фронту сигнала CLK. Заметьте, что величина "`CLK'event`" истинна при любом изменении сигнала CLK, поэтому для переключения только по положительному перепаду в сигнале CLK предусмотрена проверка "`CLK = '1'`". Существует много других способов задать процесс или составить оператор, отражающие чувствительность к перепаду сигнала; еще два способа описания D-триггера (без входа сброса) приведены в табл. 7.38.

Табл. 7.37. Поведенческое описание переключающегося по положительному фронту D-триггера на языке VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity VposDff is
  port (CLK, CLR, D: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end VposDff;

architecture VposDff_arch of VposDff is
begin
  process (CLK, CLR)
  begin
    if CLR='1' then Q <= '0'; QN <= '1';
    elsif CLK'event and CLK='1' then Q <= D; QN <= not D;
    end if;
  end process;
end VposDff_arch;
```

При тестировании тактируемой схемы вам понадобится еще одна вещь: нужно будет генерировать системный тактовый сигнал. Это совсем легко реализовать, организуя цикл внутри процесса, как это показано в табл. 7.39 для тактовой частоты 100 МГц с коэффициентом заполнения 60%.

«ВНУТРЕННОСТИ» СИНТЕЗА

Вам, наверное, интересно узнать, как программные средства синтеза реализуют в настоящем триггере описание чувствительности к фронту, приведенное в табл. 7.37 и 7.38. Большинство программных средств распознает только небольшое число предопределенных способов описания поведения схемы, переключающейся по фронту, и отображает их в предопределенные компоненты внутри программируемой ИС.

Программа синтеза фирмы Synopsis (Synopsis synthesis engine) распознает используемое нами в этой книге выражение "`CLK'event and CLK='1'`" с помощью программного продукта Foundation Series 1.5 фирмы Xilinx. Но язык VHDL предоставляет также и другие возможности для выражения того же самого функционального поведения, что и в табл. 7.38. Питер Ашенден (Peter Ashenden), автор *Справочника проектировщика по языку VHDL* (*The Designer's Guide to VHDL*. Morgan Kaufmann, 1996), испытал способность различных средств синтеза воспринимать три приведенные здесь формы задания чувствительности к перепаду и еще одну, немного модифицированную. Только одно из программных средств смогло синтезировать по трем из четырех форм; большинство испытанных программ синтеза справляется только с двумя. Следовательно, вам нужно использовать тот метод, который предписан имеющимися у вас программными средствами.

Табл. 7.38. Два других способа описания переключающегося по положительному фронту D-триггера

```
process
  wait until CLK'event and CLK='1';
  Q <= D;
end process;

Q <= D when CLK'event and CLK='1' else Q;
```

Табл. 7.39. Тактовый процесс для тестирования

```
architecture TB_arch of TB is
signal MCLK: STD_LOGIC;
signal ... -- Declare other input and output signals

process -- Clock generator
begin
  MCLK <= 1;  -- Start at 1 at time 0
  loop
    MCLK <= 0 after 6 ns;
    MCLK <= 1 after 4 ns;
  end loop;
end process;

process -- Generate the rest of the input stimuli, check outputs
begin
  ...
end;
```

8.4. Счетчики

В общем случае *счетчиком* (*counter*) называют любую тактируемую последовательностную схему, у которой диаграмма состояний представляет собой единственное кольцо (рис. 8.26). *Модулем счета* (*modulus*) называется число состояний в этом кольце. О счетчике с *m* состояниями говорят как о *счетчике с модулем счета m* (*modulo-m counter*); иногда его называют также *делителем частоты на m* (*divide-by-m counter*). У счетчика с модулем счета, не равным степени 2, есть лишние состояния, в которые он не попадает при нормальной работе

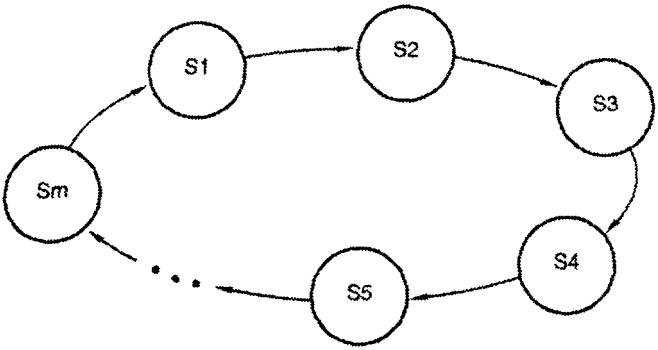


Рис. 8.26. Общий вид диаграммы состояний счетчика – единственное кольцо

По-видимому, самыми распространенными являются *n-разрядные двоичные счетчики* (*n-bit binary counter*). Такой счетчик состоит из n триггеров, и у него имеется 2^n состояний, через которые он проходит в последовательности 0, 1, 2, … , $2^n - 1$, 0, 1, … . Кодом каждого состояния служит соответствующее n-разрядное двоичное число.

8.4.1. Счетчики с последовательным переносом

При любом значении n можно так построить n-разрядный двоичный счетчик, что в нем будут только n триггеров и не будет никаких других компонентов. Для $n = 4$ такой счетчик показан на рис. 8.27. Напомним, что состояние Т-триггера меняется (на противоположное) с каждым нарастающим фронтом сигнала на его тактовом входе. Таким образом, содержимое того или иного разряда в счетчике меняется на противоположное тогда и только тогда, когда значение бита в разряде, непосредственно предшествующем этому разряду, изменяется с 1 на 0. Это соответствует двоичному счету в прямом направлении: когда бит, хранящийся в данном разряде, изменяется с 1 на 0, возникает перенос в следующий по старшинству разряд. Такой счетчик называют *счетчиком с последовательным переносом* (*ripple counter*), так как информация о переносе поочередно передается от младших разрядов к старшим, по одному биту за раз.

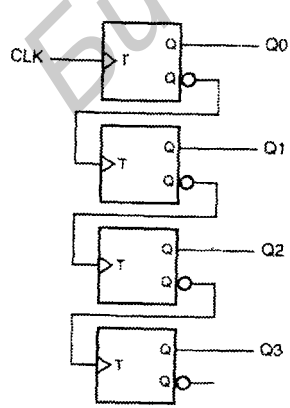


Рис. 8.27. 4-разрядный двоичный счетчик с последовательным переносом

8.4.2. Синхронные счетчики

Хотя для счетчика с последовательным переносом требуется меньше компонентов, чем для двоичного счетчика любого другого типа, за это приходится платить наименьшим быстродействием по сравнению с любыми другими счетчиками. В худшем случае, когда должен измениться бит в самом старшем разряде, выходной сигнал примет правильное значение только спустя время, равное $n \cdot t_{TQ}$, после нарастающего фронта тактового сигнала CLK, где t_{TQ} – задержка распространения в Т-триггере от входа до выхода.

В *синхронном счетчике* (*synchronous counter*) к тактовым входам всех триггеров подводится один и тот же общий тактовый сигнал CLK, так что изменения значений сигналов на выходах всех триггеров происходят в один и тот же момент времени с задержкой только на t_{TQ} наносекунд. Как видно из рис. 8.28, для этого нужно воспользоваться Т-триггерами со входом разрешения; сигнал на выходе триггера примет противоположное значение в момент, задаваемый нарастающим фронтом сигнала на его входе Т, только в том случае, если сигнал разрешения EN имеет активный уровень. Какие именно триггеры перейдут в состояние, противоположное предыдущему, на очередном нарастающем фронте сигнала на входе Т, определяется комбинационной логикой, включенной на входах разрешения EN.

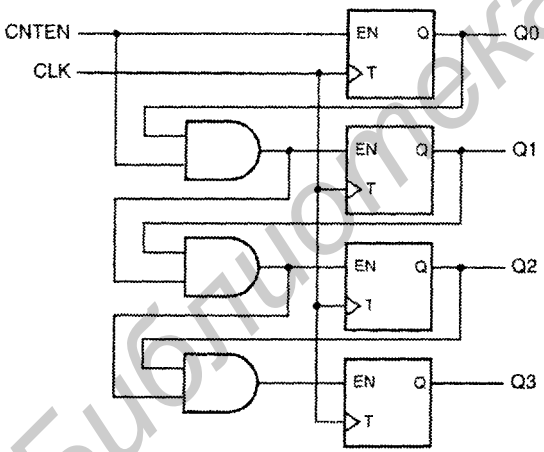


Рис. 8.28. 4-разрядный синхронный двоичный счетчик с последовательной логикой разрешения

На рис. 8.28 показано, что счетчик можно снабдить главным сигналом разрешения CNTEN. Любой из Т-триггеров может переключиться тогда и только тогда, когда сигнал CNTEN имеет единичное значение и равны 1 биты во всех разрядах, младше данного. Как и в случае двоичного счетчика с последовательным переносом, *n*-разрядный синхронный счетчик можно построить так, чтобы на один разряд приходилось фиксированное число логических схем; в данном случае в каждом разряде необходимы Т-триггер с входом разрешения и 2-входовой вентиль И.

Счетчик, изображенный на рис. 8.28, иногда называют *последовательным синхронным счетчиком* (*synchronous serial counter*), поскольку сигналы разрешения проходят через комбинационную логику последовательно от младшего разряда к старшему. Если период тактового сигнала слишком мал, то изменение в младшем разряде счетчика может не успеть дойти за это время до старшего разряда. Это затруднение преодолено в схеме на рис. 8.29, где сигнал разрешения на входе разрешения EN каждого триггера вырабатывается соответствующим вентилем И всего лишь с одним уровнем логики. В результате получается схема двоичного счетчика с самым высоким быстродействием, который носит название *параллельного синхронного счетчика* (*synchronous parallel counter*).

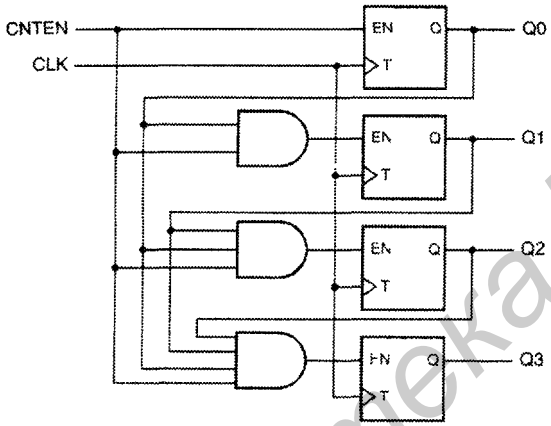


Рис. 8.29. 4-разрядный синхронный двоичный счетчик с параллельной логикой разрешения

8.4.3. Счетчики в ИС средней степени интеграции и их применение

Самым популярным счетчиком в ИС средней степени интеграции является 4-разрядный синхронный двоичный счетчик *74x163* с входами сброса и загрузки; сигналы на этих входах имеют низкий активный уровень. Традиционное условное обозначение такого счетчика приведено на рис. 8.30. Работа этого счетчика описывается таблицей состояний (табл. 8.11), а его принципиальная схема показана на рис. 8.31.

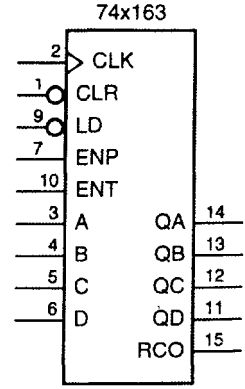


Рис. 8.30. Традиционное условное обозначение ИС 74x163

Табл. 8.11. Таблица состояний 4-разрядного двоичного счетчика 74x163

Входы				Текущее состояние				Следующее состояние			
CLR_L	LD_L	ENT	ENP	QD	QC	QB	QA	QD*	QC*	QB*	QA*
0	x	x	x	x	x	x	x	0	0	0	0
1	0	x	x	x	x	x	x	D	C	B	A
1	1	0	x	x	x	x	x	QD	QC	QB	QA
1	1	x	0	x	x	x	x	QD	QC	QB	QA
1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	0	1	1
1	1	1	1	0	0	1	1	0	1	0	0
1	1	1	1	0	1	0	0	0	1	0	1
1	1	1	1	0	1	0	1	0	1	1	0
1	1	1	1	0	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0	1	0	0	1
1	1	1	1	1	0	0	1	1	0	1	0
1	1	1	1	1	0	1	0	1	0	1	1
1	1	1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	0	0	1	1	0	1
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	0	0	0

Внутри ИС '163 используются не T-триггеры, а D-триггеры, чтобы упростить функции загрузки и сброса. На D-вход каждого триггера сигнал поступает с выхода 2-входового мультиплексора, состоящего из вентиля ИЛИ и двух вентилей И. Выходной сигнал мультиплексора равен 0, если подан входной сигнал CLR_L. В противном случае верхний из вентилей И пропускает входной сигнал данных (A, B, C или D) на выход, если подан сигнал LD_L. Если ни у одного из сигналов CLR_L и LD_L уровень не является активным, то нижний из вентилей И пропускает на выход мультиплексора выходной сигнал вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ.

Функция счета в ИС '163 выполняется с помощью вентилей ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ. На один из входов этого вентиля в каждом разряде поступает бит, хранящийся в этом разряде (QA, QB, QC или QD); на другой вход подана логическая 1, благодаря чему на выходе этого вентиля вырабатывается дополнение к биту, хранящемуся в данном разряде, но только в том случае, когда оба сигнала разрешения ENP и ENT имеют активный уровень и во всех разрядах счетчика,

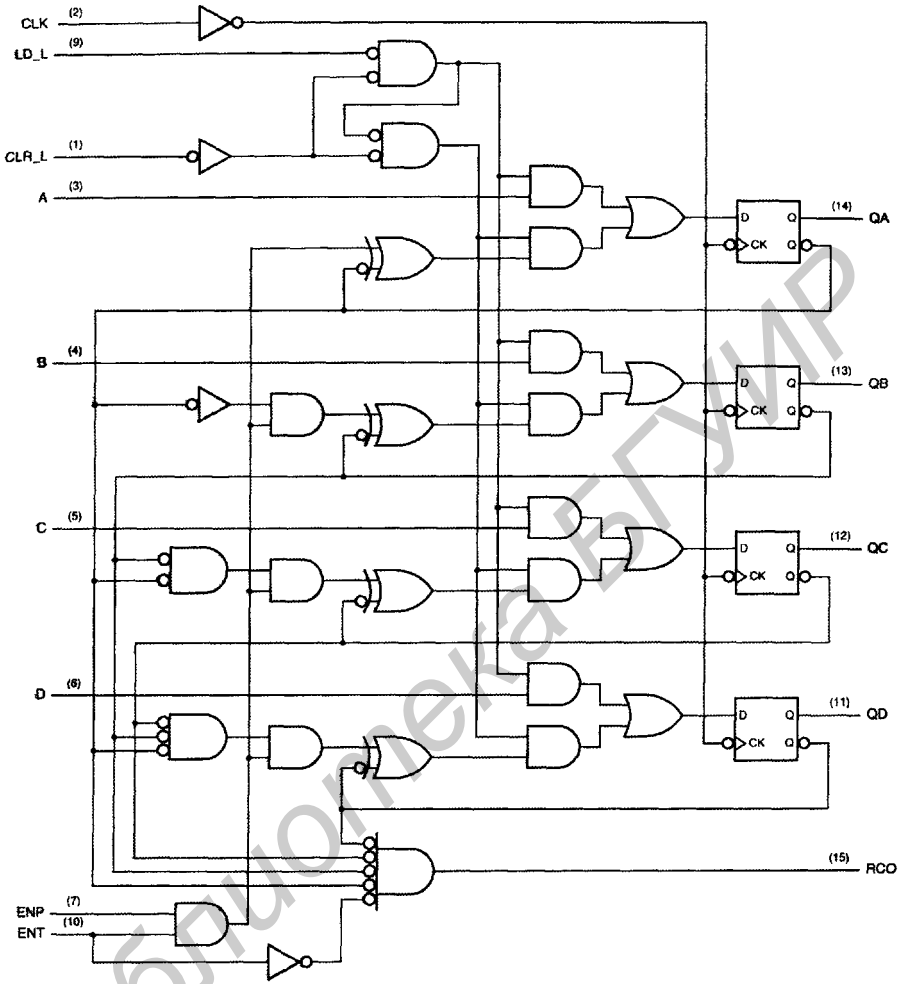


Рис. 8.31. Принципиальная схема синхронного 4-разрядного двоичного счетчика 74x163 с цоколевкой для стандартного DIP-корпуса с 16 выводами

младше данного, биты равны 1. Сигнал RCO («выход сквозного переноса») означает наличие переноса из самого старшего разряда; он равен 1, когда равны 1 биты, хранящиеся во всех разрядах счетчика, и подан сигнал разрешения ENT.

Несмотря на наличие входа разрешения, *счетчики*, выполненные в виде ИС средней степени интеграции, часто *работают в непрерывном режиме* (*free-running counters*), когда счет разрешен постоянно. На рис. 8.32 показано, какие соединения необходимо осуществить, чтобы счетчик '163 работал в таком режиме, а на рис. 8 33 приведены соответствующие этому режиму временные диаграммы. Обратите внимание: начиная с сигнала QA, частота переключений в каждом следующем сигнале вдвое меньше, чем в предыдущем. Таким образом, в режиме непрерывного счета ИС '163 может играть роль делителя частоты на 2, 4, 8 или 16; при этом ненужные старшие разряды игнорируются.

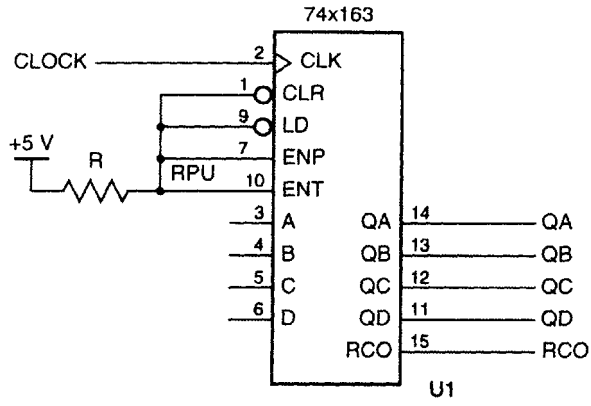


Рис. 8.32. Включение ИС 74x163 для работы в режиме непрерывного счета

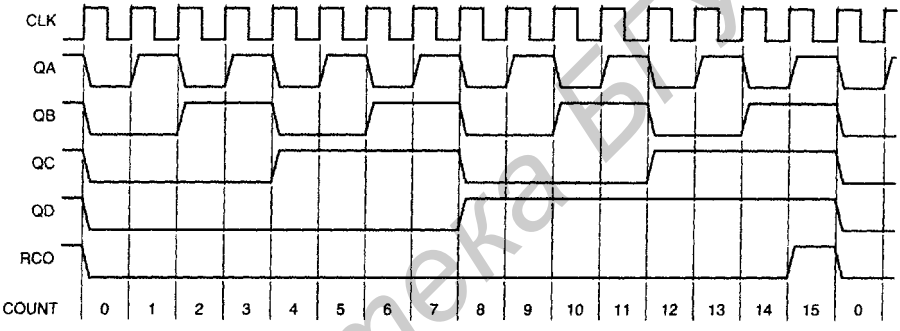


Рис. 8.33. Временные диаграммы тактового сигнала и сигналов на выходах отдельных разрядов в делителе частоты на 16

Заметьте, что счетчик '163 является полностью синхронным; это означает, что сигналы на его выходах изменяются только на нарастающем фронте сигнала CLK. Но в некоторых приложениях бывает необходимо осуществлять сброс асинхронно; это позволяет ИС *74x161*. У микросхемы '161 такая же цоколевка, как и у ИС '163, но внутри нее вход CLR_L подключен к асинхронным входам сброса ее триггеров.

Другими вариантами счетчиков с точно такой же цоколевкой являются ИС *74x160* и *74x162*; их функции в целом такие же, как и у микросхем '161 и '163, за исключением того, что в них последовательность счета изменена: за состоянием 9 следует состояние 0. Другими словами, эти ИС представляют собой счетчики по модулю 10; их иногда называют *декадными счетчиками* (*decade counters*). На рис. 8.34 приведены временные диаграммы для счетчиков '160 и '162 в режиме непрерывного счета. Хотя частота сигналов на выходах QD и QC равна одной десятой от частоты сигнала CLK, коэффициент заполнения у сигналов QD и QC не равен 50%, а сигнал QB, хотя и имеет частоту, в пять раз меньшую, чем частота тактового сигнала, его коэффициент заполнения не остается постоянным. Позднее в этом разделе мы покажем, как осуществляется деление частоты на 10 с 50%-ным коэффициентом заполнения у выходного сигнала.

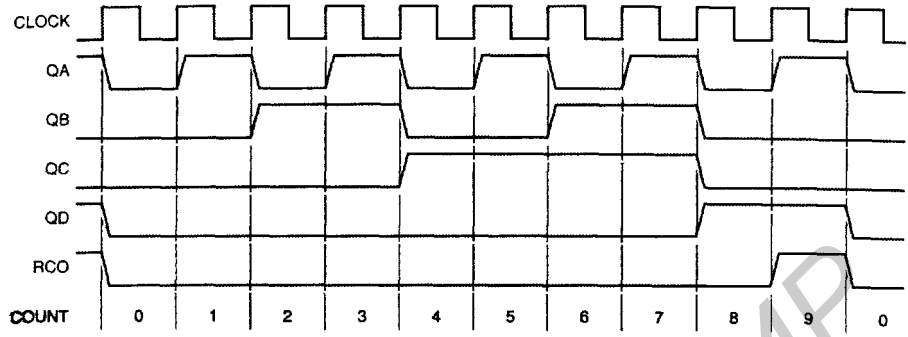


Рис. 8.34. Временные диаграммы тактового сигнала и сигналов на выходах отдельных разрядов делителя частоты на 10 в режиме непрерывного счета

Счетчик '163 сам по себе считает по модулю 16, но с помощью сигналов CLR_L и LD_L его можно заставить считать по меньшему модулю, чем 16, укоротив проходимую им последовательность состояний. На рис. 8.35, например, показано использование ИС '163 в качестве счетчика по модулю 11. Когда счетчик находится в состоянии 15, на выходе RCO возникает единичный сигнал, который заставляет счетчик перейти в следующее состояние, равное 5; поэтому схема считает от 5 до 15 и снова начинает счет с 5, так что всего в цикле счета 11 состояний.

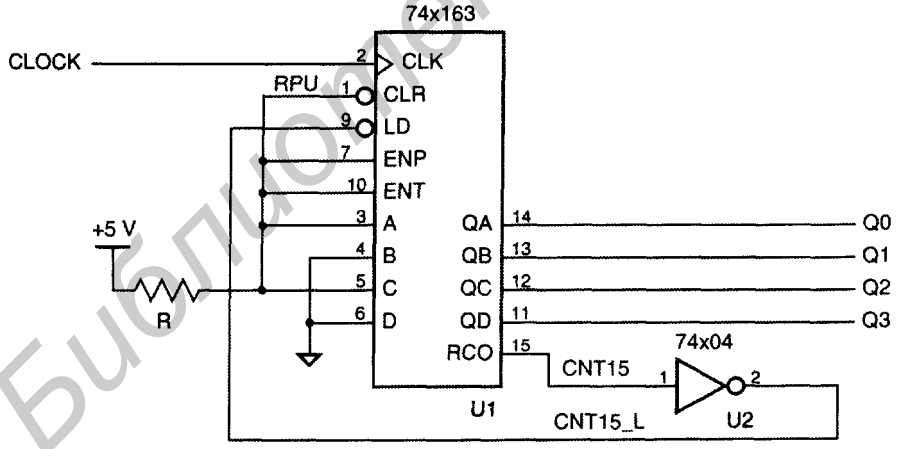


Рис. 8.35. Применение ИС 74x163 в качестве счетчика по модулю 11 с последовательностью счета 5, 6, ... , 15, 5, 6, ...

На рис. 8.36 демонстрируется другой подход к решению задачи о счете по модулю 11. В этой схеме для обнаружения состояния счетчика, равного 10, используется вентиль И-НЕ; сигнал, возникающий на выходе этого вентиля, заставляет счетчик перейти в состояние 0. Обратите внимание на то, что для обнаружения состояния 10 (в двоичной записи – 1010) нужен вентиль только с двумя входами. Хотя для обнаружения состояния CNT10 = Q3 · Q2′ · Q1 · Q0′ более

естественным было бы применение 4-входового вентиля, можно обойтись 2-входовым вентилем благодаря тому, что в проходимой счетчиком последовательности состояний 0–10 нет другого состояния с Q3 = 1 и Q1 = 1. И в общем случае, для обнаружения состояния N двоичного счетчика, считающего от 0 до N, нужно объединить по И выходы только тех разрядов, на которых имеются единицы в двоичном представлении числа N.

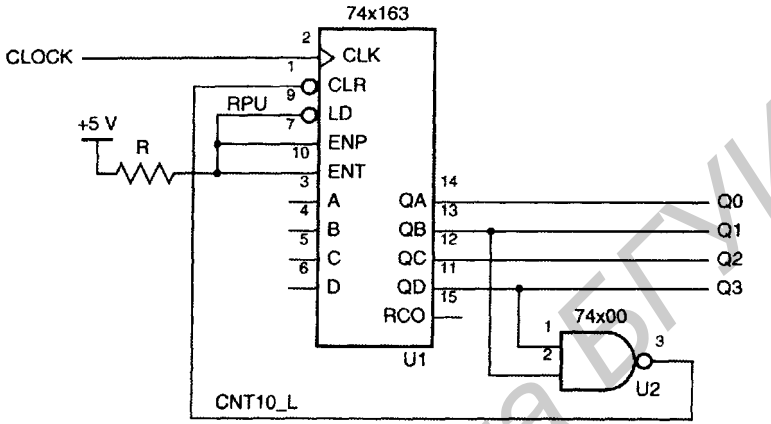


Рис. 8.36. Применение ИС 74x163 в качестве счетчика по модулю 11 с последовательностью счета 0, 1, 2, ... , 10, 0, 1, ...

Существует много других способов заставить счетчик '163 считать по модулю 11. Выбор того или иного подхода зависит от условий применения; в частности, это может быть один из приведенных способов или их комбинация (см. задачу 8.36). Приведем еще один пример. В параграфе 2.10 мы обещали показать, как строится схема, осуществляющая счет в десятичном коде с избытком 3, указанном в табл. 2.9. Такая схема на основе ИС '163 представлена на рис. 8.37. С помощью вентиля И-НЕ обнаруживается состояние 1100 и обеспечивается загрузка комбинации 0011 в качестве следующего состояния. На рис. 8.38 приведены результирующие временные диаграммы. Заметьте, что y выходного сигнала Q3 коэффициент заполнения равен 50%, – это может быть полезным в ряде приложений.

Включая последовательно несколько ИС 74x163, можно реализовать двоичный счет по модулю, большему 16. На рис. 8.39 показана общая структура такой схемы. Входы CLK, CLR_L и LD_L всех ИС '163 соединены параллельно, так что все они переключаются, сбрасываются и загружаются в одно и то же время. Главный сигнал разрешения счета CNTEN подается на ИС '163, ответственную за младшие разряды. На выходе RCO4 сигнал возникает только в том случае, если младшая ИС '163 находится в состоянии 15 *и* подан сигнал разрешения CNTEN; выход RCO4 соединен с входами разрешения ИС '163, ответственной за старшие разряды. Таким образом, оба сигнала – информация о переносе и разрешение счета в целом – переходят от одного 4-разрядного счетчика к другому. Как и в случае последовательного синхронного счетчика, приведенного ранее на рис. 8.28, по этому принципу можно построить счетчик с любым числом разрядов; максимальная скорость счета ограничена задержкой прохождения сигнала переноса через все каскады (см. задачу 8.38).

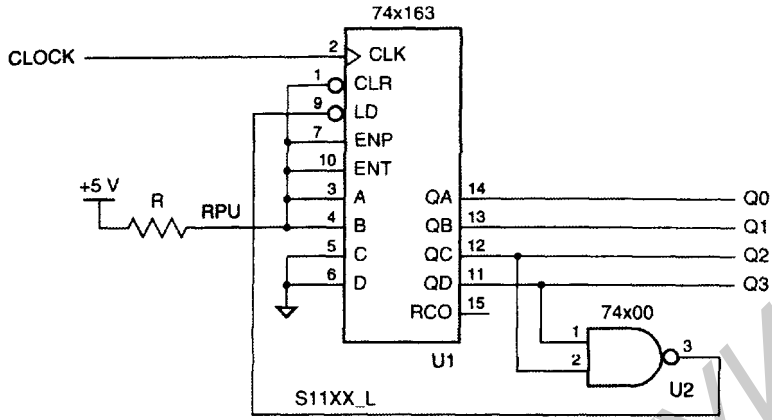


Рис. 8.37. Счетчик 74x163 в схеме, осуществляющей десятичный счет в коде с избытком 3

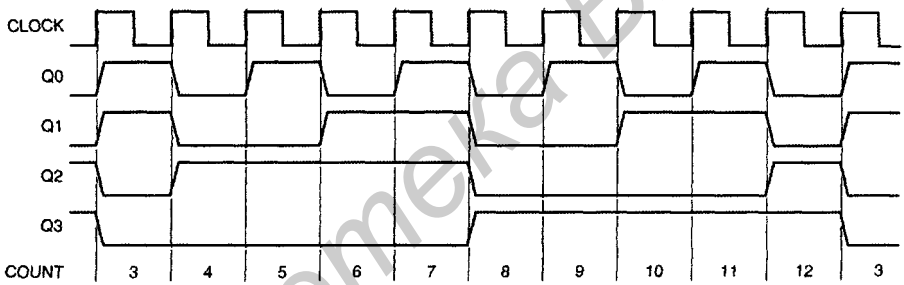


Рис. 8.38. Временные диаграммы для схемы на основе ИС '163, осуществляющей десятичный счет в коде с избытком 3

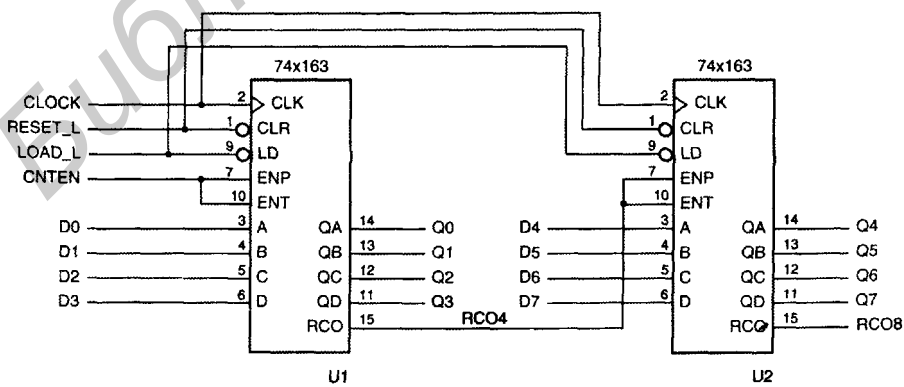


Рис. 8.39. Общая структура последовательного включения счетчиков 74x163

Даже опытные конструкторы не всегда учитывают разницу между входами разрешения ENP и ENT в счетчиках типа '163, поскольку счет возможен только тогда,

когда оба этих сигнала имеют активный уровень. Однако достаточно взглянуть на внутреннюю структуру ИС '163, представленную на рис. 8.31, чтобы увидеть вполне очевидное различие между этими сигналами: сигнал ENT проходит также на выход сквозного переноса. Во многих случаях это различие является существенным.

Рассмотрим, например, построенный на основе ИС '163 счетчик по модулю 193, который считает от 63 до 255 (рис. 8.40). Выходной сигнал MAXCNT принимает единичное значение, когда счетчик оказывается в состоянии 255, и останавливает счет до тех пор, пока снова не появится сигнал загрузки GO_L. С приходом сигнала GO_L в счетчик загружается состояние 63 и он возобновляет счет до 255. (Заметьте, что схема чувствительна к сигналу GO_L только в том случае, когда счетчик находится в состоянии 255.) Для того чтобы счет был остановлен при достижении состояния 255, необходимо обеспечить наличие единичного сигнала на выходе MAXCNT, несмотря на то, что счет прерван. Поэтому на вход ENT младшего счетчика подан постоянный сигнал разрешения, выход RCO этого счетчика соединен с входом ENT старшего счетчика и состояние 255 обнаруживается по сигналу MAXCNT, хотя значение сигнала CNTEN равно нулю (сравните с сигналом RCO8 в схеме на рис. 8.39). Для разрешения счета сигнал CNTEN подают параллельно на входы ENP. Активным уровнем сигнала RELOAD_L на выходе вентиля И-НЕ счетчик возвращается в состояние 63 только в том случае, когда он находится в состоянии 255 и поступает сигнал GO_L.

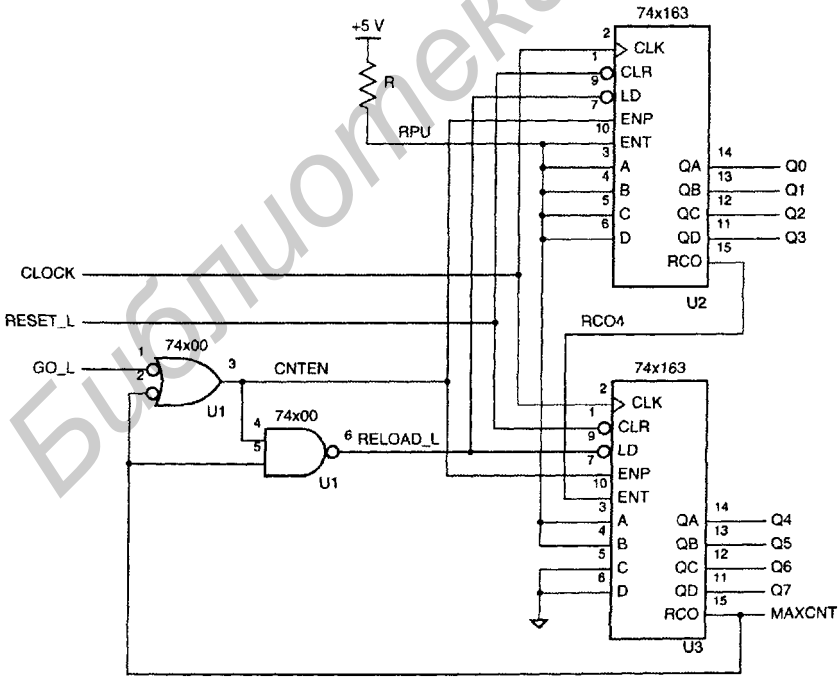


Рис. 8.40. Счетчик по модулю 193 на ИС 74x163 с последовательностью счета 63, 64, ... , 255, 63, 64, ... (На входы LD_L должен подаваться сигнал, являющийся результатом объединения по ИЛИ сигналов RESET_L и RELOAD_L. – *Исправление автора*.)

Другой счетчик, похожий по выполняемым функциям на ИС 74x163, – это микросхема *74x169*; ее условное обозначение приведено на рис. 8.41. Одно из отличий счетчика '169 состоит в том, что выходной сигнал переноса и сигналы на входах разрешения имеют активный низкий уровень. Более существенно то, что ИС '169 является *реверсивным счетчиком* (*up/down counter*): он может считать в сторону увеличения или уменьшения содержащегося в нем двоичного числа в зависимости от значения входного сигнала UP/DN. Когда сигнал UP/DN равен 1, происходит счет в сторону увеличения; когда значение сигнала UP/DN равно 0, счет ведется в сторону уменьшения.

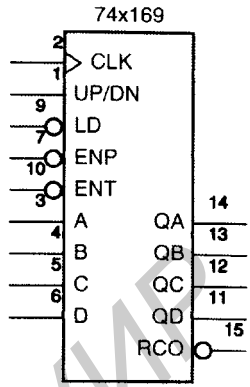


Рис. 8.41. Условное обозначение реверсивного счетчика 74x169

8.4.4. Декодирование состояний двоичного счетчика

Объединяя двоичный счетчик с дешифратором, можно вырабатывать кодовые слова кода «1 из *m*», содержащие по одному единичному сигналу на каждое состояние счетчика. Это бывает полезно в том случае, когда с помощью счетчика управляют набором устройств: сигналы разрешения поступают на отдельные устройства, когда счетчик находится в соответствующем состоянии. При таком подходе каждый из выходных сигналов дешифратора служит сигналом разрешения для одного из устройств.

На рис. 8.42 показано, как можно связать между собой счетчик 74x163, считающий по модулю 8, и дешифратор 3×8 типа 74x138; в результате получается схема, вырабатывающая восемь сигналов, каждый из которых соответствует одному из состояний счетчика. На рис. 8.43 приведены типичные временные диаграммы для этой схемы. Сигнал на каждом выходе дешифратора имеет активный уровень в течение определенного периода тактового сигнала.

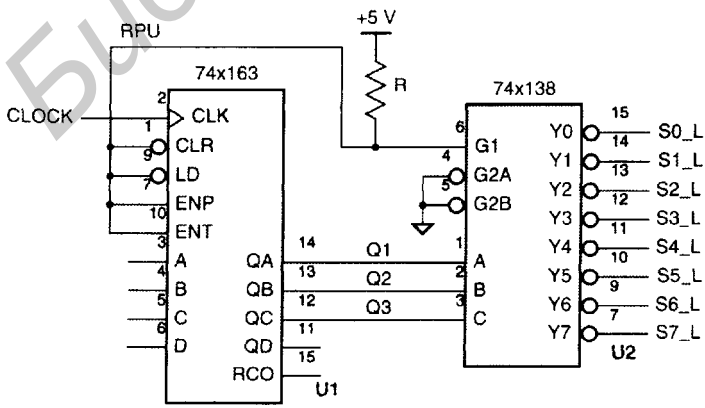


Рис. 8.42. Двоичный счетчик по модулю 8 с дешифратором

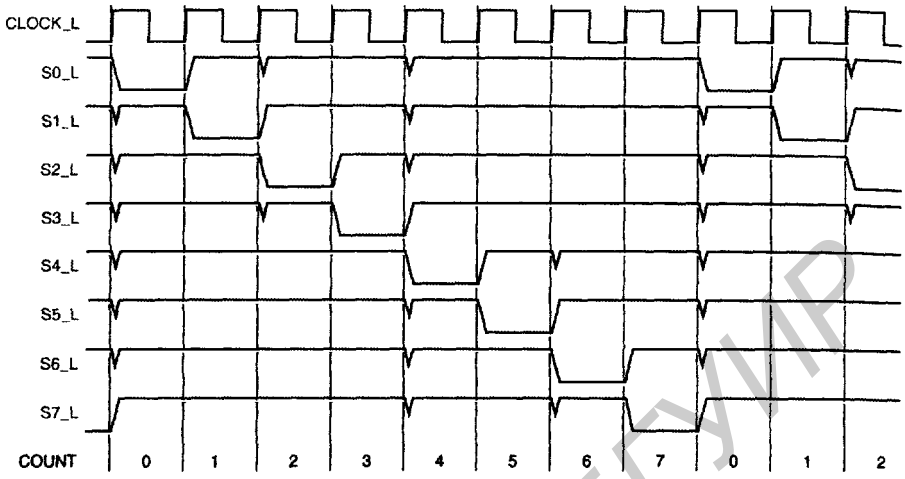


Рис. 8.43. Временные диаграммы для двоичного счетчика по модулю 8 с дешифратором, на которых видны паразитные импульсы на выходах дешифратора

Обратите внимание на то, что при изменении в счетчике содержимого двух или большего числа разрядов на выходах дешифратора могут возникать «паразитные импульсы», несмотря на отсутствие паразитных импульсов на выходах ИС '163 и статических источников опасности у дешифратора '138. В синхронном счетчике типа '163 выходные сигналы изменяются не точно в одно и то же время. Но более важным является то, что в дешифраторе типа '138 задержки прохождения сигналов по разным путям различны; например, задержка прохождения сигнала от входа B к выходу Y1_L меньше, чем задержка на пути от входа A к выходу Y1_L. Поэтому даже если значения входных сигналов, равные 011, изменяются одновременно и становятся равными 100, в сигнале на выходе Y1_L может появиться паразитный импульс. В данном примере мы видим, что паразитные импульсы могут возникать при *любой* реализации функции двоичного декодирования; в таком случае говорят о наличии *функционального источника опасности* (*function hazard*).

В большинстве приложений выходные сигналы дешифратора, изображенные на рис. 8.43, используются в качестве сигналов, управляющих регистрами, счетчиками и другими устройствами, переключающимися по фронту (например, в качестве входных сигналов EN_L, LD_L и ENP_L, подаваемых на ИС 74x377, 74x163 и 74x169 соответственно). В этом случае указанные на рисунке паразитные импульсы на выходах дешифратора не страшны, поскольку они возникают строго *после* перепада в тактовом сигнале и кончаются задолго до очередного фронта тактового сигнала, когда выходные сигналы дешифратора принимаются во внимание другими устройствами, переключающимися по фронту. Однако паразитные импульсы могли бы вызвать затруднения при попытке использовать выходные сигналы дешифратора в качестве управляющих сигналов $\overline{SR}$-защелки типа S_L или R_L. Точно так же подобные сигналы, в которых потенциально могут иметь место паразитные импульсы, ни в коем случае нельзя применять в качестве тактовых сигналов для переключающихся по фронту устройств.

Если нужно «очистить» сигналы от паразитных импульсов, указанных на рис. 8.43, то один из способов сделать это состоит в подаче выходных сигналов дешифратора '138 на другой регистр, в котором установившиеся значения этих сигналов фиксировались бы на следующем фронте тактового сигнала (рис. 8.44). Заметьте, что сигналам на выходах регистра присвоены другие имена, чтобы учесть задержку на один такт при прохождении через регистр. Но коль скоро вы готовы заплатить за решение проблемы путем применения 8-разрядного регистра, имейте в виду, что существует менее дорогое решение, состоящее в использовании 8-разрядного «кольцевого счетчика», на выходах которого непосредственно вырабатываются декодированные сигналы без паразитных импульсов, как это будет показано в разделе 8.5.6.

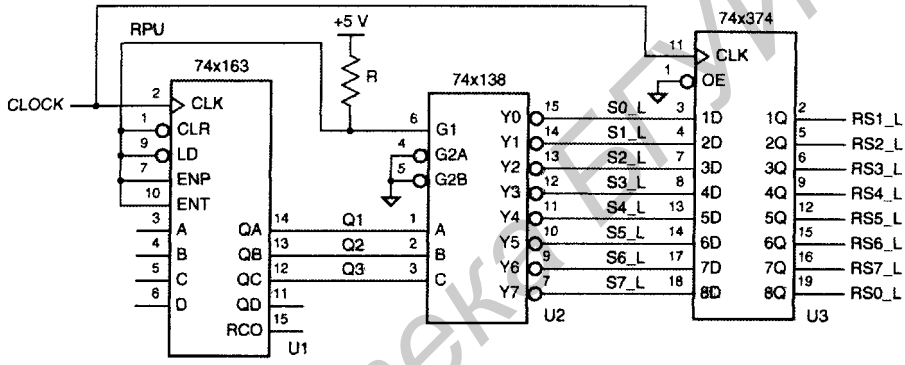


Рис. 8.44. Схема двоичного счетчика по модулю 8 с дешифратором без паразитных импульсов на выходах

8.4.5. Описание счетчиков на языке ABEL и их реализация в ПЛУ

Двоичные счетчики удобно создавать, описывая их на языке ABEL и реализуя в ПЛУ, по нескольким причинам:

- Большой конечный автомат часто можно разбить на два или большее число меньших по размерам конечных автоматов так, что один из них является двоичным счетчиком, задающим время, в течение которого другой блок будет оставаться в том или ином состоянии. Такой подход позволяет упростить как проект в целом, так и его схемное воплощение.
- Во многих приложениях бывают нужны счетчики с модулем счета более или менее кратным степени 2, к которым предъявляются определенные требования в отношении инициализации, а также обнаружения или пропуска тех или иных состояний. Например, счетчик в контроллере лифта может пропускать состояние 13. Вместо использования стандартного двоичного счетчика и применения дополнительной логики для удовлетворения предъявляемых требований разработчик может в программе на языке ABEL точно задать требуемые функции.

- У большинства обычных счетчиков в ИС средней степени интеграции только 4 разряда, в то время как в одном ПЛУ с 24 выводами можно образовать двоичный счетчик с числом разрядов, доходящим до 10.

Самым популярным счетчиком в ИС средней степени интеграции является 4-разрядный двоичный счетчик 74x163, представленный на рис. 8.31. Даже поверхностного взгляда на схему этого счетчика достаточно, чтобы увидеть, что его логика возбуждения не так проста, особенно с точки зрения использования в ней вентилей ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ. А язык ABEL позволяет самым непосредственным образом описать поведение счетчика, к чему мы и приступаем.

Напомним, что в языке ABEL символ "+" употребляется для сложения целых чисел. Когда с помощью данного оператора «складываются» два набора, каждый из них интерпретируется как двоичное число; самый правый элемент набора соответствует младшему разряду числа. С учетом этого функцию ИС 74x163 можно задать так, как это сделано в программе на языке ABEL, приведенной в табл. 8.12. Когда счет разрешен, к текущему состоянию добавляется 1.

Табл. 8.12. Программа на языке ABEL для 4-разрядного двоичного счетчика типа 74x163

```
module Z74X163
title '4-bit Binary Counter'

" Input pins
CLK, LD_L, CLR_L, ENP, ENT        pin;
A, B, C, D                        pin;
" Output pins
QA, QB, QC, QD                    pin istype 'reg';
RCO                               pin istype 'com';

" Set definitions
INPUT = [D, C, B, A];
COUNT = [QD, QC, QB, QA];

LD = !LD_L; CLR = !CLR_L;       " Active-level conversions

equations

COUNT.CLK = CLK;

COUNT := !CLR & ( LD & INPUT
                # !LD & (ENT & ENP) & (COUNT + 1)
                # !LD & !(ENT & ENP) & COUNT);

RCO = (COUNT == [1,1,1,1]) & ENT;

end Z74X163
```

Табл. 8.13. Минимизированные равенства для 4-разрядного двоичного счетчика из табл. 8.12

```
QA := (CLR_L & LD_L & ENT & ENP & !QA
     # CLR_L & LD_L & !ENP & QA
     # CLR_L & LD_L & !ENT & QA
     # CLR_L & !LD_L & A);

QB := (CLR_L & LD_L & ENT & ENP & !QB & QA
     # CLR_L & LD_L & QB & !QA
     # CLR_L & LD_L & !ENP & QB
     # CLR_L & LD_L & !ENT & QB
     # CLR_L & !LD_L & B);

QC := (CLR_L & LD_L & ENT & ENP & !QC & QB & QA
     # CLR_L & LD_L & QC & !QA
     # CLR_L & LD_L & QC & !QB
     # CLR_L & LD_L & !ENP & QC
     # CLR_L & LD_L & !ENT & QC
     # CLR_L & !LD_L & C);

QD := (CLR_L & LD_L & ENT & ENP
          & !QD & QC & QB & QA
     # CLR_L & !LD_L & D
     # CLR_L & LD_L & QD & !QB
     # CLR_L & LD_L & QD & !QC
     # CLR_L & LD_L & !ENP & QD
     # CLR_L & LD_L & !ENT & QD
     # CLR_L & LD_L & QD & !QA);

RCO = (ENT & QD & QC & QB & QA);
```

В табл. 8.13 представлены минимизированные логические равенства, которые генерирует компилятор языка ABEL для 4-разрядного счетчика. Заметьте, что при переходе в сторону старших разрядов каждому следующему выходному сигналу требуется на один терм-произведение больше. В результате этого размер счетчиков, которые можно реализовать в ПЛУ 16V8 или даже 20V8 обычно ограничен пятью или шестью разрядами. В других устройствах, в том числе в ПЛУ X-серии и в некоторых ИС типа CPLD, имеется структура ИСКЛЮЧАЮЩЕЕ ИЛИ, позволяющая строить бóльшие по размерам счетчики без увеличения требуемого числа термов-произведений.

Проектирование счетчика с заданной последовательностью состояний на языке ABEL много проще, нежели приспособление для этих целей обычного двоичного счетчика. Например, ABEL-программу из табл. 8.12 можно заставить считать в коде с избытком 3 (см. рис. 8.38), следующим образом изменив равенства:

```
COUNT := !CLR & ( LD & INPUT
       # !LD & (ENT & ENP) &
            ((COUNT==12) & 3) # ((COUNT!=12) & (COUNT + 1))
       # !LD & !(ENT & ENP) & COUNT);

RCO = (COUNT == 12) & ENT;
```

Чтобы получить счетчики с бóльшим числом разрядов, можно включить несколько ПЛУ последовательно, предусмотрев в каждом каскаде выходной сигнал переноса, служащий индикатором того, что счетчик в этом каскаде собирается начать счет сначала. Существует два основных подхода к вырабатыванию сигнала переноса:

- *Комбинационный выходной сигнал переноса* (*combinational carry output*) указывает, что счет разрешен и в данный момент счетчик находится в своем последнем состоянии перед тем, как начать счет сначала. Для 5-разрядного счетчика при счете в прямом направлении имеем:

```
COUT = CNTEN & Q4 & Q3 & Q2 & Q1 & Q0;
```

Поскольку правая часть равенства содержит сигнал CNTEN, такой подход допускает сквозной перенос в многокаскадных счетчиках, если выход COUT в каждом каскаде соединен с входом CNTEN в следующем каскаде.

- Правая часть равенства для *сигнала переноса на регистровом выходе* (*registered carry output*) указывает, что следующим состоянием счетчика будет последнее его состояние перед тем, как счет начнется сначала. Таким образом, на следующем такте счетчик входит в свое последнее состояние и сигнал переноса переходит на активный уровень. В случае 5-разрядного счетчика с входами загрузки и сброса имеем:

```
COUT := !CLR & !LD & CNTEN
          & Q4 & Q3 & Q2 & Q1 & !Q0
      # !CLR * !LD * !CNTEN
          & Q4 & Q3 & Q2 & Q1 & Q0
      # !CLR & LD
          & D4 & D3 & D2 & D1 & D0;
```

Достоинство второго подхода заключается в том, что сигнал COUT вырабатывается с меньшей задержкой, чем при комбинационном подходе. Но теперь требуются внешние вентили между каскадами, поскольку сигнал CNTEN в каждом каскаде должен быть результатом объединения по И главного сигнала разрешения счета и выходных сигналов COUT всех каскадов, младше данного. Необходимости помещения внешних вентилей можно избежать при наличии у старших каскадов счетчика нескольких входов разрешения.

8.4.6. Описание счетчиков на языке VHDL

Как и язык ABEL, VHDL позволяет совсем легко описывать счетчики. Наибольшее затруднение при этом может возникнуть только из-за строгих требований в языке VHDL к типам сигналов, которые должны быть определены правильно и последовательно.

В табл. 8.14 представлена VHDL-программа для двоичного счетчика типа 74x163. В программе используется библиотека IEEE.std_logic_arith.all, включающая тип UNSIGNED, как это было объяснено в разделе 5.9.6. Эта библиотека содержит определения операторов "+" и "–", посредством которых выполняются сложение и вычитание без знака операндов типа UNSIGNED. В программе для счетчика входы и выходы счетчика объявлены как векторы типа UNSIGNED, а с помощью оператора "+" осуществляется требуемое инкрементирование содержимого счетчика.

Для хранения содержимого счетчика в программе определен внутренний сигнал IQ. Можно было бы использовать для этого сигнал Q непосредственно, но тогда мы должны были бы объявить его выходным сигналом типа buffer, а не out. Кроме того, мы могли бы определить тип портов D и Q как STD_LOGIC_VECTOR, но тогда нам пришлось бы выполнять преобразование типов в теле процесса (см. задачу 8.51).

Табл. 8.14. VHDL-программа для 4-разрядного двоичного счетчика типа 74x163

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity V74x163 is
    port ( CLK, CLR_L, LD_L, ENP, ENT: in STD_LOGIC;
           D: in UNSIGNED (3 downto 0);
           Q: out UNSIGNED (3 downto 0);
           RCO: out STD_LOGIC );
end V74x163;

architecture V74x163_arch of V74x163 is
signal IQ: UNSIGNED (3 downto 0);
begin
process (CLK, ENT, IQ)
  begin
    if (CLK'event and CLK='1') then
      if CLR_L='0' then IQ <= (others => '0');
      elsif LD_L='0' then IQ <= D;
      elsif (ENT and ENP)='1' then IQ <= IQ + 1;
      end if;
    end if;
    if (IQ=15) and (ENT='1') then RCO <= '1';
    else RCO <= '0';
    end if;
    Q <= IQ;
  end process;
end V74x163_arch;
```

Воспользовавшись поведенческим описанием на языке VHDL, столь же легко, как и на языке ABEL, задать определенную последовательность состояний. В табл. 8.15, например, счетчик типа 74x163 видоизменен таким образом, чтобы счет происходил согласно коду с избытком 3 (3, ..., 12, 3, ...).

К сожалению, некоторые VHDL-средства синтезируют счетчики не совсем удачно. В частности, они пытаются реализовать одиночный шаг в счете посредством двоичного сумматора, операндами которого служат содержимое счетчика и константа, равная 1. При таком подходе требуется много больше комбинационной логики, чем в счетчиках, изготовляемых в виде отдельных ИС, и этот подход оказывается особенно расточительным применительно к ИС типа CPLD и FPGA, содержащим Т-триггеры, вентили ИСКЛЮЧАЮЩЕЕ ИЛИ и другие структуры, специально оптимизированные для построения счетчиков. В этом случае полезной альтернативой является написание структурной VHDL-программы, ориентированной на имеющиеся в наличии ячейки в тех конкретных ИС типа CPLD и FPGA или в специализированных ИС, в которых предстоит реализовать проектируемое устройство.

Табл. 8.15. VHDL-архитектура для счета в порядке, задаваемом кодом с избытком 3

```
architecture V74xs3_arch of V74x163 is
signal IQ. UNSIGNED (3 downto 0),
begin
process (CLK, ENT, IQ)
  begin
    if CLK'event and CLK='1' then
      if CLR_L='0' then IQ <= (others => '0'),
      elsif LD_L='0' then IQ <= D;
      elsif (ENT and ENP)='1' and (IQ=12) then IQ <= ('0','0','1','1');
      elsif (ENT and ENP)='1' then IQ <= IQ + 1;
      end if;
    end if;
    if (IQ=12) and (ENT='1') then RCO <= '1',
    else RCO <= '0';
    end if;
    Q <= IQ;
  end process;
end V74xs3_arch;
```

Одноразрядную ячейку для счетчика типа 74x163 можно построить, например, так, как показано на рис. 8.45. Эта схема рассчитана на последовательное распространение битов переноса, так что ею можно воспользоваться в любом каскаде произвольно большого счетчика; единственным ограничением будет коэффициент разветвления по выходу источников сигналов, являющихся общими для всех каскадов. Определения сигналов в одноразрядной ячейке таковы:

CLK	Тактовый сигнал, общий для всех каскадов.
LDNOCLR	Общий для всех каскадов сигнал, принимающий единичное значение, когда на вход счетчика LD сигнал подан, а сигнал CLR отсутствует.
NOCLRORLD	Общий для всех каскадов сигнал, принимающий единичное значение, когда отсутствуют сигналы на обоих входах счетчика CLR и LD.
CNTENP	Общий для всех каскадов сигнал, равный 1, если на вход счетчика ENP подан сигнал разрешения.
Di	Индивидуальный входной сигнал загрузки данных *i*-й ячейки.
CNTENi	Индивидуальный последовательный входной сигнал разрешения счета *i*-й ячейки.
CNTENi+1	Индивидуальный последовательный выходной сигнал разрешения счета *i*-й ячейки.
Qi	Индивидуальный выходной сигнал счетчика в *i*-м разряде.

В табл. 8.16 приведена VHDL-программа, соответствующая схеме одноразрядной ячейки, показанной на рис. 8.45. В этой программе предполагается, что D-триггер в виде компонента `Vdffqqn` уже определен; он подобен D-триггеру из

табл. 8.6 с добавлением инверсного выхода QN. При проектировании на основе специализированной ИС или ИС типа FPGA тип компонента, являющегося триггером, следует выбрать из библиотеки стандартных элементов производителя.

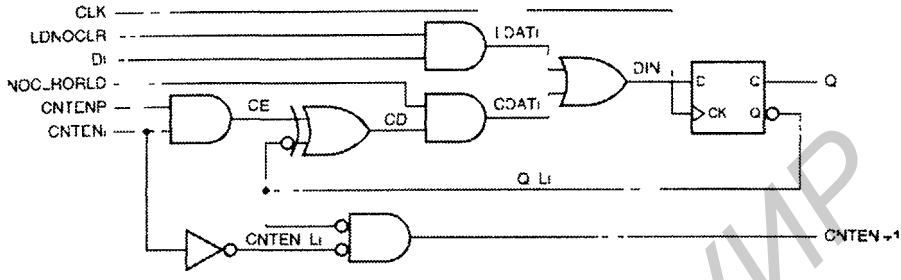


Рис. 8.45. Одноразрядная ячейка для синхронного счетчика с последовательным переносом типа 74x163

Табл. 8.16. VHDL-программа для ячейки, показанной на рис. 8.45

```
library IEEE;
use IEEE.std_logic_1164.all;

entity syncsercell is
  port( CLK, LDNOCLR, NOCLRORLD, CNTENP, D, CNTEN: in STD_LOGIC;
        CNTENO, Q: out STD_LOGIC );
end syncsercell;

architecture syncsercell_arch of syncsercell is
component Vdffqqn
  port( CLK, D: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end component;
signal LDAT, CDAT, DIN, Q_L: STD_LOGIC;
begin
  LDAT <= LDNOCLR and D;
  CDAT <= NOCLRORLD and ((CNTENP and CNTEN) xor not Q_L);
  DIN  <= LDAT or CDAT;
  CNTENO <= (not Q_L) and CNTEN;
  U1: Vdffqqn port map (CLK, DIN, Q, Q_L);
end syncsercell_arch;
```

В табл. 8.17 показано, как на основе рассмотренной одноразрядной ячейки строится 8-разрядный синхронный счетчик с последовательным переносом. Первыми двумя операторами присваивания в теле архитектуры синтезируются общие для всех разрядов сигналы LDNOCLR и NOCLRORLD. Следующие два оператора отражают граничные условия для последовательной цепочки разрешения счета. Наконец, оператор `generate` (см. раздел 5.11.3) реализует восемь 1-разрядных ячеек счетчика и связывает между собой звенья цепочки разрешения счета.

ВОПРОС СТИЛЯ

Программа, приведенная в табл. 8.16, написана в стиле, представляющем собой комбинацию потокового и структурного стилей в языке VHDL. Ее можно было бы написать полностью структурно, воспользовавшись, например, определениями компонентов вентилей производителя данной специализированной ИС, гарантируя тем самым, что результат синтеза будет точно соответствовать схеме на рис. 8.45. Однако большинство средств синтеза сами способны выбрать лучшую реализацию вентилей по простым сигнальным операторам присваивания, указанным в программе.

Табл. 8.17. VHDL-программа для 8-разрядного синхронного счетчика с последовательным переносом типа 74x163

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x163s is
  port( CLK, CLR_L, LD_L, ENP, ENT  in STD_LOGIC;
        D  in STD_LOGIC_VECTOR (7 downto 0);
        Q  out STD_LOGIC_VECTOR (7 downto 0);
        RCO  out STD_LOGIC );
end V74x163s;

architecture V74x163s_arch of V74x163s is
component syncsercell
  port( CLK, LDNOCLR, NOCLRORLD, CNTENP, D, CNTEN  in STD_LOGIC;
        CNTENO, Q  out STD_LOGIC );
end component;
signal LDNOCLR, NOCLRORLD: STD_LOGIC; -- common signals
signal SCNTEN  STD_LOGIC_VECTOR (8 downto 0); -- serial count-enable inputs
begin
  LDNOCLR <= (not LD_L) and CLR_L; -- create common load and clear controls
  NOCLRORLD <= LD_L and CLR_L;
  SCNTEN(0) <= ENT, -- serial count-enable into the first stage
  RCO <= SCNTEN(8);  -- RCO is equivalent to final count-enable output
  g1  for i in 0 to 7 generate     -- generate the eight syncsercell stages
      U1: syncsercell port map ( CLK, LDNOCLR, NOCLRORLD, ENP, D(i), SCNTEN(i),
                                 SCNTEN(i+1), Q(i));
  end generate;
end V74x163s_arch;
```

Очевидно, что легко построить счетчик бóльших или меньших размеров простым изменением нескольких определений в этой программе. Удобно воспользоваться также оператором `generic` языка VHDL, чтобы задавать число разрядов в счетчике путем изменения всего лишь одной строки в тексте программы (см. задачу 8.53).

8.5. Регистры сдвига

8.5.1. Структура регистра сдвига

Регистр сдвига (*shift register*) – это *n*-разрядный регистр, содержимое которого можно сдвигать на один разряд на каждом такте. На рис. 8.46 показана структура регистра сдвига с *последовательным вводом* (*serial input*) и *последовательным выводом* (*serial output*). Последовательный входной сигнал SERIN – это новый бит, который «вдвигается» с одного конца на данном такте. Этот бит появляется на последовательном выходе SEROUT спустя *n* тактов и теряется на следующем такте. Таким образом, *n*-разрядный регистр с последовательным вводом и последовательным выводом можно использовать для задержки сигнала на *n* тактов.

У *регистра сдвига с последовательным вводом и параллельным выводом* (*serial-in, parallel-out shift register*), приведенного на рис. 8.47, имеются выходы для всех хранимых в нем битов, благодаря чему они доступны для других схем. Таким регистром можно воспользоваться для выполнения *преобразования последовательного кода в параллельный* (*serial-to-parallel conversion*), как это будет объяснено в данном параграфе позднее.

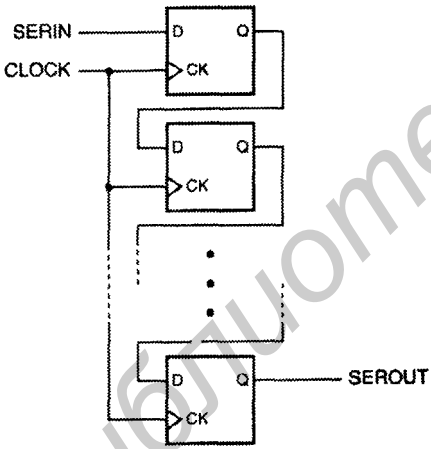


Рис. 8.46. Структура регистра сдвига с последовательным вводом и последовательным выводом

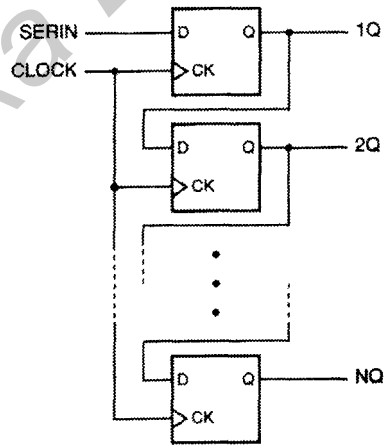


Рис. 8.47. Структура регистра сдвига с последовательным вводом и параллельным выводом

Можно поступить и наоборот, построив *регистр сдвига с параллельным вводом и последовательным выводом* (*parallel-in, serial-out shift register*). На рис. 8.48 представлена общая структура такого устройства. В зависимости от значения сигнала на управляющем входе LOAD/SHIFT (этот сигнал можно было бы назвать также LOAD или SHIFT_L) на каждом такте происходит либо загрузка новых данных с входов 1D–ND, либо сдвиг уже имеющегося содержимого регистра. В схеме этого устройства на D-входе каждого триггера стоит 2-входовой мультиплексор, позволяющий выбирать тот или иной сигнал. С помощью регистра сдвига с параллельным вводом и последовательным выводом можно осуществить *преобразование параллельного кода в последовательный* (*parallel-to-serial conversion*), о чем также пойдет речь позднее.

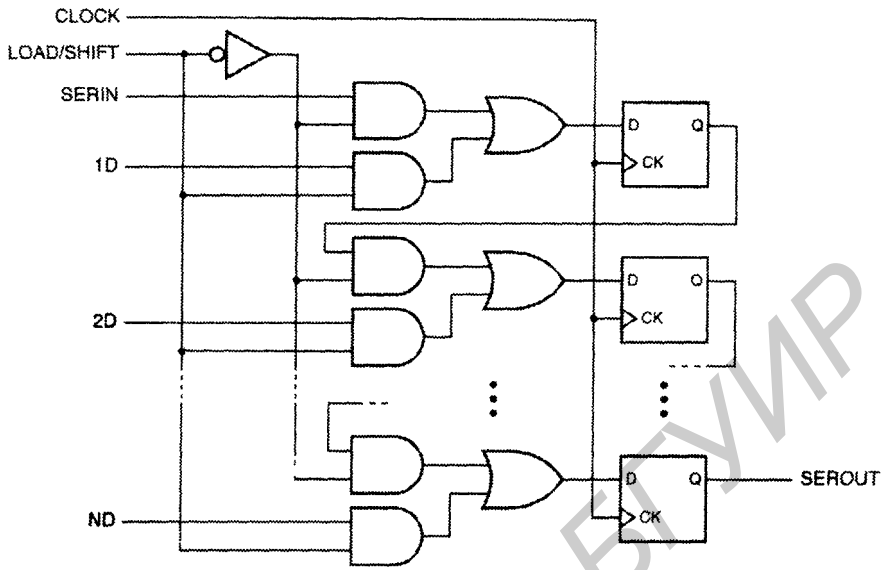


Рис. 8.48. Структура регистра сдвига с параллельным вводом и последовательным выводом

Если регистр сдвига с параллельным вводом снабдить выводами для всех сохраняемых в нем битов, то получится показанный на рис. 8.49 *регистр сдвига с параллельным вводом и параллельным выводом* (*parallel-in, parallel-out shift register*). В общем случае такого устройства достаточно для любых применений из числа упомянутых выше.

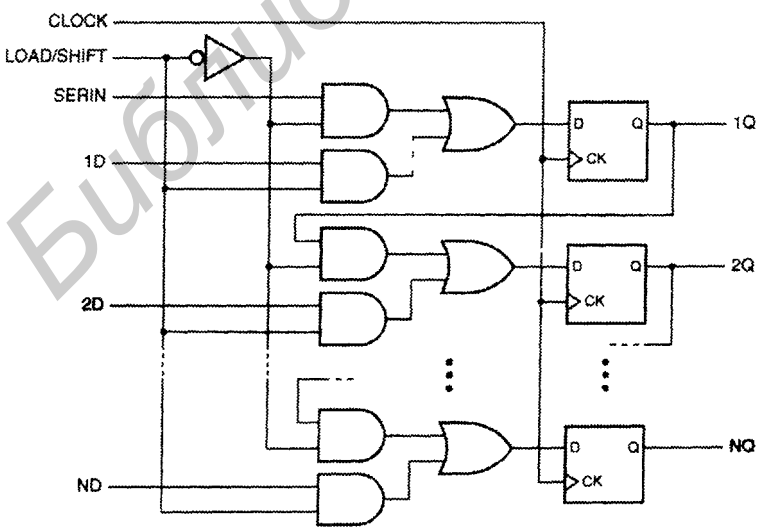


Рис. 8.49. Структура регистра сдвига с параллельным вводом и параллельным выводом

8.5.2. Регистры сдвига в ИС средней степени интеграции

На рис. 8.50 приведены условные обозначения трех популярных 8-разрядных регистров сдвига, выполненных в виде ИС средней степени интеграции. ИС *74x164* – это устройство с последовательным вводом и параллельным выводом, у которого имеется также асинхронный вход сброса CLR_L. У этого регистра два последовательных входа, объединяемые внутри ИС по правилу логического И. Другими словами, для записи единицы в первый разряд регистра необходимо, чтобы единичные значения были у обоих входных сигналов SERA и SERB.

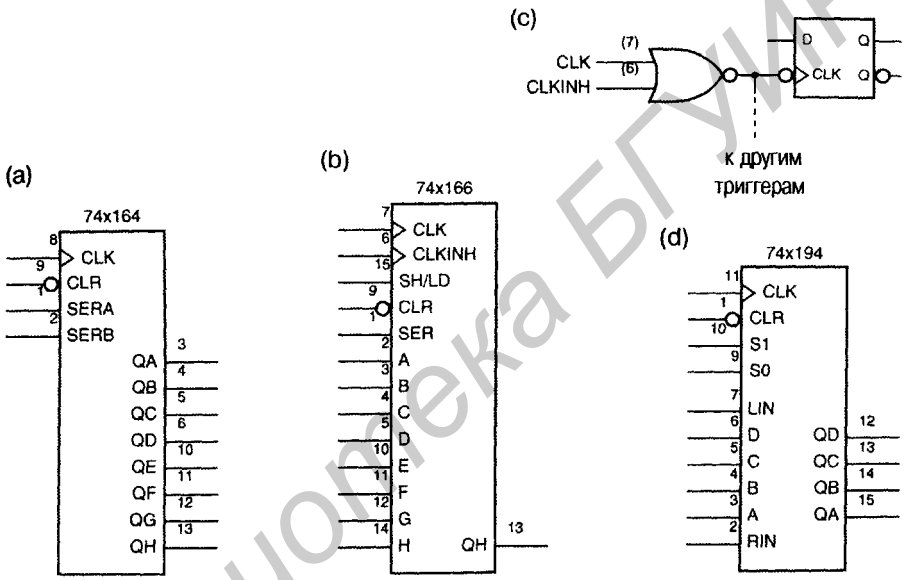


Рис. 8.50. Традиционные условные обозначения регистров сдвига, выполненных в виде ИС средней степени интеграции: (a) 8-разрядный регистр сдвига с последовательным вводом и параллельным выводом 74x164, (b) 8-разрядный регистр сдвига с параллельным вводом и последовательным выводом 74x166; (c) эквивалентная схема входной цепи для тактового сигнала в ИС 74x166; (d) универсальный регистр сдвига 74x194

У регистра сдвига с параллельным вводом и последовательным выводом *74x166* также есть асинхронный вход сброса. Сдвиг в этом устройстве происходит в том случае, когда входной сигнал SH/LD равен 1; в противном случае загружаются новые данные. В ИС '166 необычной является схема обработки тактового сигнала, который называют в этом случае «стробируемым тактовым сигналом» (см. также раздел 8.8.2): имеются два тактовых входа, подключенных к триггерам внутри ИС так, как показано на рис. 8.50(c). Создатели ИС '166 имели в виду, что на вход CLK будет поступать сигнал от источника тактового сигнала, работающего в непрерывном режиме, а на вход CLKINH будет подаваться сигнал запрета CLK, чтобы в пределах отрезка времени, пока действует сигнал CLKINH, не происходили ни сдвиг, ни загрузка, то есть текущее содержимое реги-

стра сохранялось. Но для того, чтобы схема работала именно так, сигнал CLKINH должен изменяться только в те моменты времени, когда CLK равен 1; в противном случае на тактовых входах триггеров внутри ИС будут возникать нежелательные перепады сигнала. Значительно безопаснее реализовать функцию «удержания» с помощью устройств, к рассмотрению которых мы теперь переходим.

ИС *74x194* представляет собой 4-разрядный двунаправленный регистр сдвига с параллельным вводом и параллельным выводом. Его принципиальная схема приведена на рис. 8.51. Регистры сдвига, с которыми мы познакомились до сих пор, называют *однонаправленными регистрами сдвига* (*unidirectional shift registers*), поскольку сдвиг в них может происходить только в одном направлении. ИС '194 является *двунаправленным регистром сдвига* (*bidirectional shift register*), так как его содержимое можно сдвигать в ту или другую сторону, в зависимости от значения управляющего входного сигнала. Про эти два направления говорят «сдвиг влево» и «сдвиг вправо», хотя графическое изображение принципиальной схемы и условное обозначение не обязательно соответствуют этим пространственным представлениям. Под *сдвигом влево* (*left*), применительно к ИС '194, понимают «сдвиг в направлении от QD к QA», а под *сдвигом вправо* (*right*) – «сдвиг в направлении от QA к QD». В нашем случае принципиальная схема и условное обозначение согласуются с этими названиями, если повернуть их на 90° по часовой стрелке.

Табл. 8.18 представляет собой функциональное описание ИС 74x194 в сжатом виде: в ней отсутствуют столбцы, соответствующие большинству входов (A–D, RIN, LIN) и текущему состоянию (QA–QD). Но поскольку каждое следующее состояние представлено в виде функции этих неявных переменных, этой таблицей полностью определено поведение ИС '194 для всех 2^{12} возможных комбинаций текущего состояния и значений входных сигналов без необходимости записи 4096 строк!

Табл. 8.18. Функциональное описание 4-разрядного универсального регистра сдвига 74x194

Функция	Входы		Следующее состояние			
	S1	S0	QA*	QB*	QC*	QD*
Хранение	0	0	QA	QB	QC	QD
Сдвиг вправо	0	1	RIN	QA	QB	QC
Сдвиг влево	1	0	QB	QC	QD	LIN
Загрузка	1	1	A	B	C	D

Заметьте, что «левый вход» LIN (left-in) ИС '194 принципиально размещается на микросхеме «справа», поскольку он служит для последовательного ввода *при сдвиге влево*. Аналогично, вход RIN используется для последовательного ввода *при сдвиге вправо*.

ИС '194 иногда называют *универсальным* регистром сдвига, так как его можно заставить вести себя как любой из регистров сдвига менее общего типа, которые мы рассматривали до сих пор (например, как однонаправленный регистр сдвига, как регистр сдвига с последовательным вводом и параллельным выводом или как регистр сдвига с параллельным вводом и последовательным выводом). По-

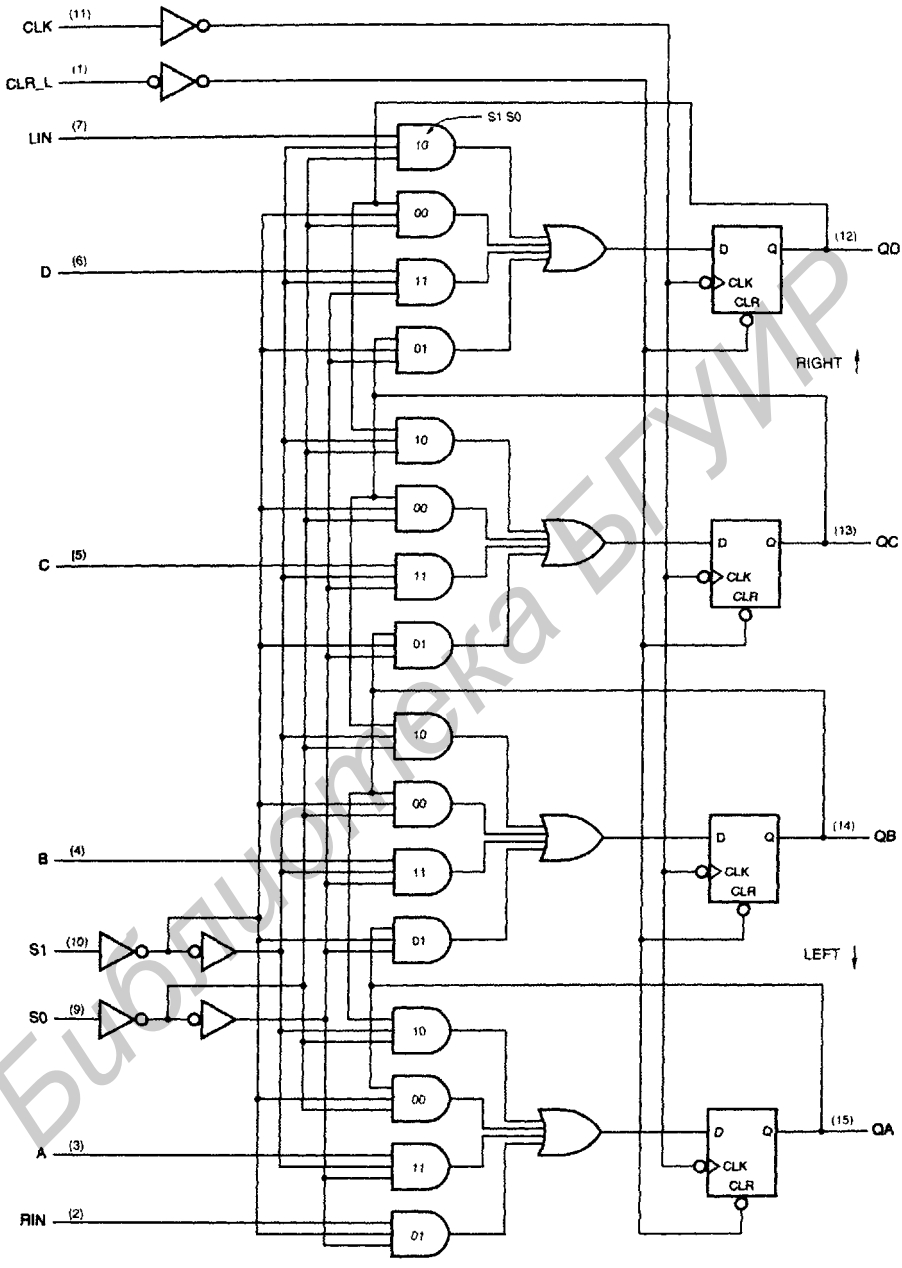


Рис. 8.51. Принципиальная схема 4-разрядного универсального регистра сдвига 74x194 с цоколевкой для стандартного DIP-корпуса с 16 выводами

этому во многих примерах в дальнейшем фигурирует ИС '194, включенная таким образом, что используется подмножество функций из числа тех, которые способна выполнять эта микросхема.

ИС *74x299* является 8-разрядным универсальным регистром сдвига, размещенным в корпусе с 20 выводами; ее условное обозначение и принципиальная схема приведены на рис. 8.52 и 8.53. Как видно из табл. 8.19, по выполняемым функциям и по функциональному описанию эта микросхема похожа на ИС '194. Чтобы сэкономить на числе выводов, в ИС '299 в качестве входов и выходов используются одни и те же сигнальные линии с тремя состояниями. При загрузке (S1 S0 = 11) буферы с тремя состояниями заперты и записываемые данные поступают через выводы AQA–HQH. В других случаях при подаче сигналов на входы G1_L и G2_L запомненные биты выводятся на те же самые контакты. Содержимое крайнего левого и крайнего правого разрядов доступно другим схемам в течение всего времени на отдельных выводах QA и QH, которые служат только выходами.

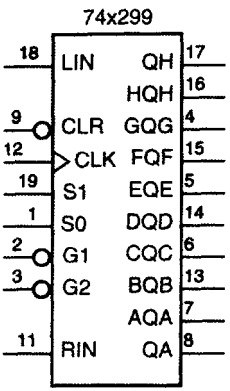


Рис. 8.52. Традиционное условное обозначение ИС 74x299

Табл. 8.19. Функциональное описание 8-разрядного универсального регистра сдвига 74x299

	Входы		Следующее состояние							
Функция	***S1***	***S0***	***QA****	***QB****	***QC****	***QD****	***QE****	***QF****	***QG****	***QH****
Хранение	0	0	QA	QB	QC	QD	QE	QF	QG	QH
Сдвиг вправо	0	1	RIN	QA	QB	QC	QD	QE	QF	QG
Сдвиг влево	1	0	QB	QC	QD	QE	QF	QG	QH	LIN
Загрузка	1	1	AQA	BQB	CQC	DQD	EQE	FQF	GQG	HQH

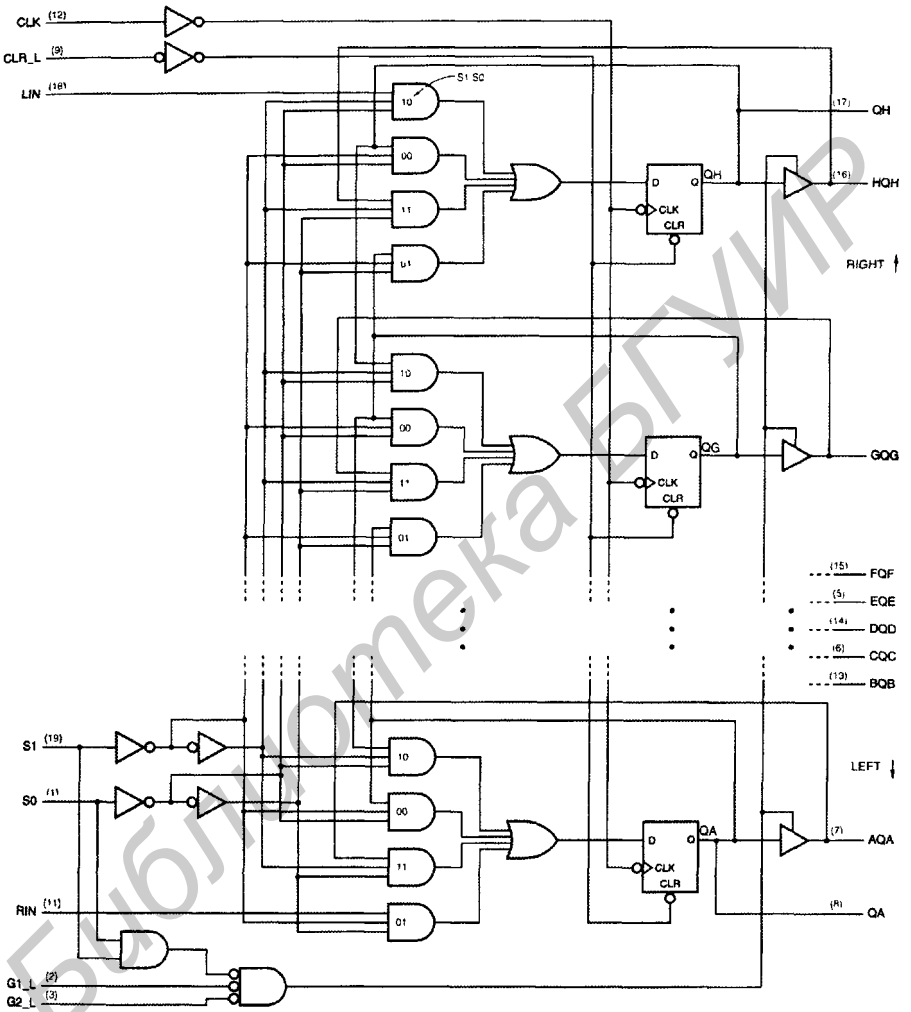


Рис. 8.53. Принципиальная схема 8-разрядного универсального регистра сдвига 74x299 с цоколевкой для стандартного DIP-корпуса с 20 выводами (RIGHT – вправо, LEFT – влево)

8.5.3. Самое распространенное в мире применение регистров сдвига

Чаще всего регистры сдвига применяются для того, чтобы преобразовать параллельные данные в последовательный формат при передаче и при записи, а также для обратного преобразования последовательных данных в параллельный формат для целей обработки и отображения (см. раздел 2.16.1). Самый распространенный пример преобразования последовательных данных, с которыми вы почти наверняка встречаетесь каждый день, – это *цифровая телефония* (*digital telephony*).

За прошедшие годы телефонные компании установили на своих *центральных станциях* (*central offices, COs*) оборудование с цифровой коммутацией. Большинство домашних телефонов связано с центральной станцией двухпроводными аналоговыми соединениями. Когда аналоговый речевой сигнал поступает на центральную станцию, из него – с помощью аналого-цифрового преобразователя – 8000 раз в секунду берутся выборки (каждые 125 микросекунд), в результате чего образуется последовательность байтов, – 8-разрядных двоичных слов, – выражающих собой знак и величину аналогового сигнала в момент взятия выборки. Затем повсюду в телефонной сети ваш голос передается в цифровом виде со скоростью 64 кбит/с по *последовательным каналам* (*serial channels*) до тех пор, пока он не преобразуется обратно в аналоговую форму цифро-аналоговым преобразователем на центральной станции адресата.

НУ, УЖ НЕ ЗНАЮ...

К концу 80-х годов получила развитие технология ISDN (Integrated Services Digital Network, цифровая сеть связи с комплексными услугами), целью которой было дотянуть до домашних телефонов последовательные цифровые каналы со скоростью передачи 144 кбит/с. Идея состояла в том, чтобы по одной паре проводов передавать два телефонных разговора со скоростью 64 кбит/с плюс сигналы управления, следующие со скоростью 16 кбит/с; в результате увеличивается пропускная способность уже проложенных телефонных линий.

В первом издании этой книги мы обратили внимание читателей на то, что из-за задержки с развертыванием сети ISDN среди специалистов появилась другая расшифровка аббревиатуры ISDN: "Imaginary Services Delivered Nowhere"(«воображаемые услуги, не доставляемые никуда»). К середине 90-х годов сеть ISDN утвердилась, наконец, в США, но не столько для передачи речевых сигналов, сколько для обеспечения «скоростных» соединений Интернета.

К большому сожалению телефонных компаний внедрение сети ISDN затормозилось сначала распространением недорогих 56-килобитовых аналоговых модемов, а затем – растущей доступностью высокоскоростных соединений (со скоростью передачи от 160 кбит/с до 2 Мбит/с и больше), в которых используются более новые кабель-модемная технология и технология DSL (Digital Subscriber Line, цифровая абонентская линия).

Полоса пропускания, необходимая для передачи одного оцифрованного речевого сигнала со скоростью 64 кбит/с, много *меньше* той, какую предоставляет одна цифровая сигнальная линия и коммутационное оборудование, построенное на цифровых ИС. Поэтому в большинстве случаев на цифровых телефонных станциях осуществляется *мультиплексирование* (*multiplexing*) нескольких 64-килобитовых каналов в одну линию передачи, на чем экономятся как провода, так и объем коммутационного оборудования, выражаемый числом интегральных схем. В следующем разделе мы покажем, как с помощью небольшого числа ИС средней степени интеграции можно обрабатывать сигналы, передаваемые по 32 каналам, причем функции, выполняемые этими микросхемами, легко реализовать в одной ИС типа CPLD. Это классический пример *пространственно-временно́го обмена* (*space/time tradeoff*) при цифровом проектировании: заставляя интегральные схемы работать быстрее, вы решаете задачу большего объема, используя меньшее число микросхем. В этом заключается главная причина, по которой телефонная сеть «стала цифровой».

8.5.4 Последовательно-параллельное преобразование

На рис. 8.54 приведен типичный пример последовательной передачи данных между двумя модулями (такое соединение может быть частью коммутационного оборудования на телефонной станции). Обычно передача в таком соединении от источника сигналов к месту назначения происходит по трем сигнальным линиям:

- *Тактовый сигнал* задает темп передачи, указывая интервалы времени, отводимые на передачу одного бита. Если система состоит всего лишь из двух модулей, то тактовый сигнал может генерироваться блоком управления, размещенным в передатчике сигнала, как показано на рисунке. В более крупных системах может быть один общий источник тактового сигнала, разводимого по модулям.
- *Последовательные данные*: сами по себе данные передаются по одной линии.
- *Синхронизация*. *Импульсом синхронизации* (*synchronization pulse* или *sync pulse*) указывается точка отсчета в формате данных, например, начало байта или слова в последовательном потоке данных. В некоторых системах этот сигнал бывает опущен, а синхронизация достигается передачей по линии данных последовательности специального вида.

ОБЩЕНАЦИОНАЛЬНОЕ ВРЕМЯ

Поверите вы мне или нет, но используемый в телефонной сети тактовый сигнал, частота которого с высокой точностью равна 8 кГц, генерируется в г. Сент-Луисе и распространяется повсеместно в США! Тактовый сигнал в конкретной части коммутационного оборудования местной телефонной станции обычно является производным от национального тактового сигнала. В частности, фигурирующий в примере этого параграфа тактовый сигнал с частотой 2.048 МГц мог бы быть получен с помощью схемы фазовой автоподстройки частоты путем умножения частоты национального тактового сигнала на 256.

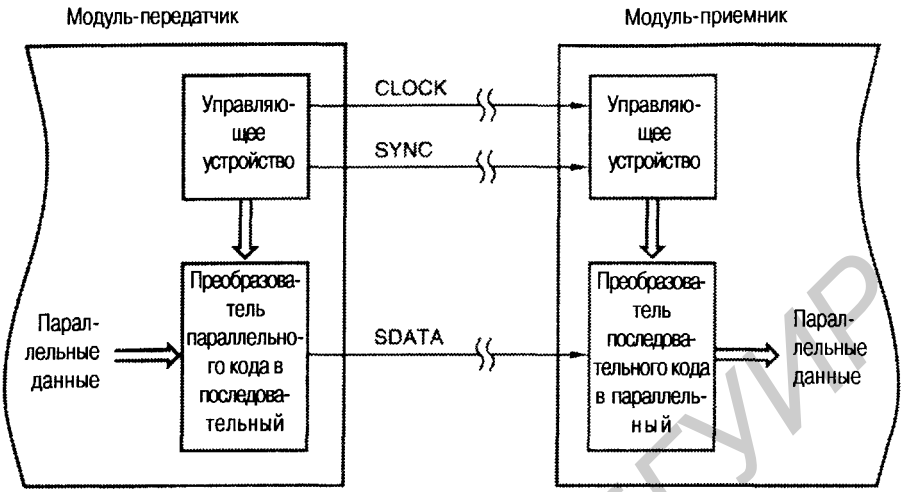


Рис. 8.54. Система с последовательной передачей данных между модулями

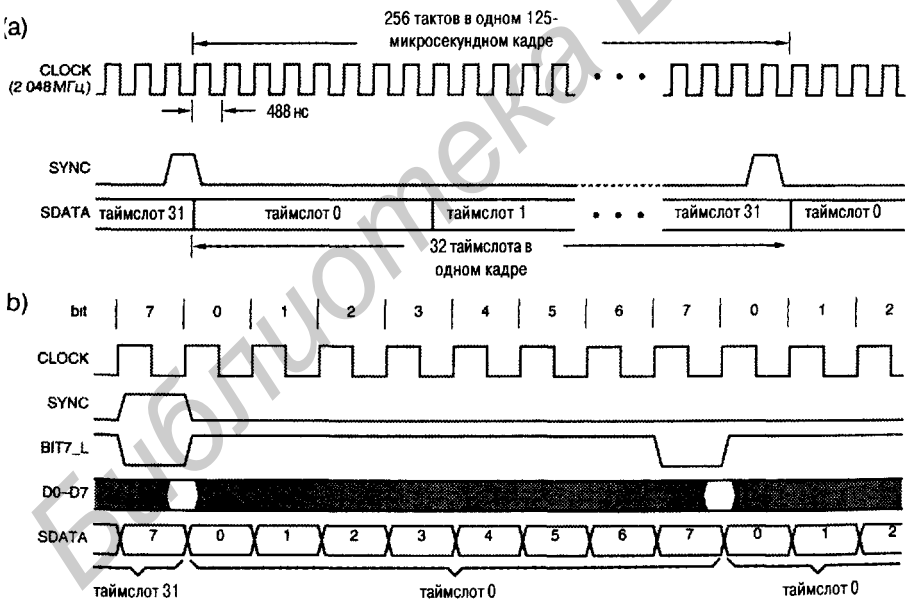


Рис. 8.55. Временные диаграммы преобразования параллельного кода в последовательный: (a) полный кадр; (b) один байт в начале каждого кадра

В ситуации, типичной для цифровой телефонии, временные параметры этих сигналов имеют значения, приведенные на рис. 8.55(a). Частота сигнала CLOCK равна 2.048 МГц, и это позволяет передавать 32×8000 8-битовых байтов в секунду. Импульсом сигнала SYNC длительностью в 1 бит определяется начало 125-микросекундного интервала, называемого *кадром* (*frame*). За это время по линии SDATA передается 256 битов, причем весь этот интервал разбит на 32

таймслота (*timeslot*), содержащих по 8 битов каждый. В каждом таймслоте передается в цифровом виде один речевой сигнал. Номера таймслотов и расположение битов внутри каждого из них отсчитываются от импульса сигнала SYNC.

На рис. 8.56 представлена схема, осуществляющая преобразование параллельных данных в последовательный формат, указанный на рис. 8.55(a), с учетом деталей, приведенных на рис. (b). Две ИС 74x163 образуют счетчик по модулю 256, работающий в непрерывном режиме; этим счетчиком задается кадр. Пять старших разрядов счетчика указывают номер таймслота, а три младших разряда – номер бита в пределах таймслота.

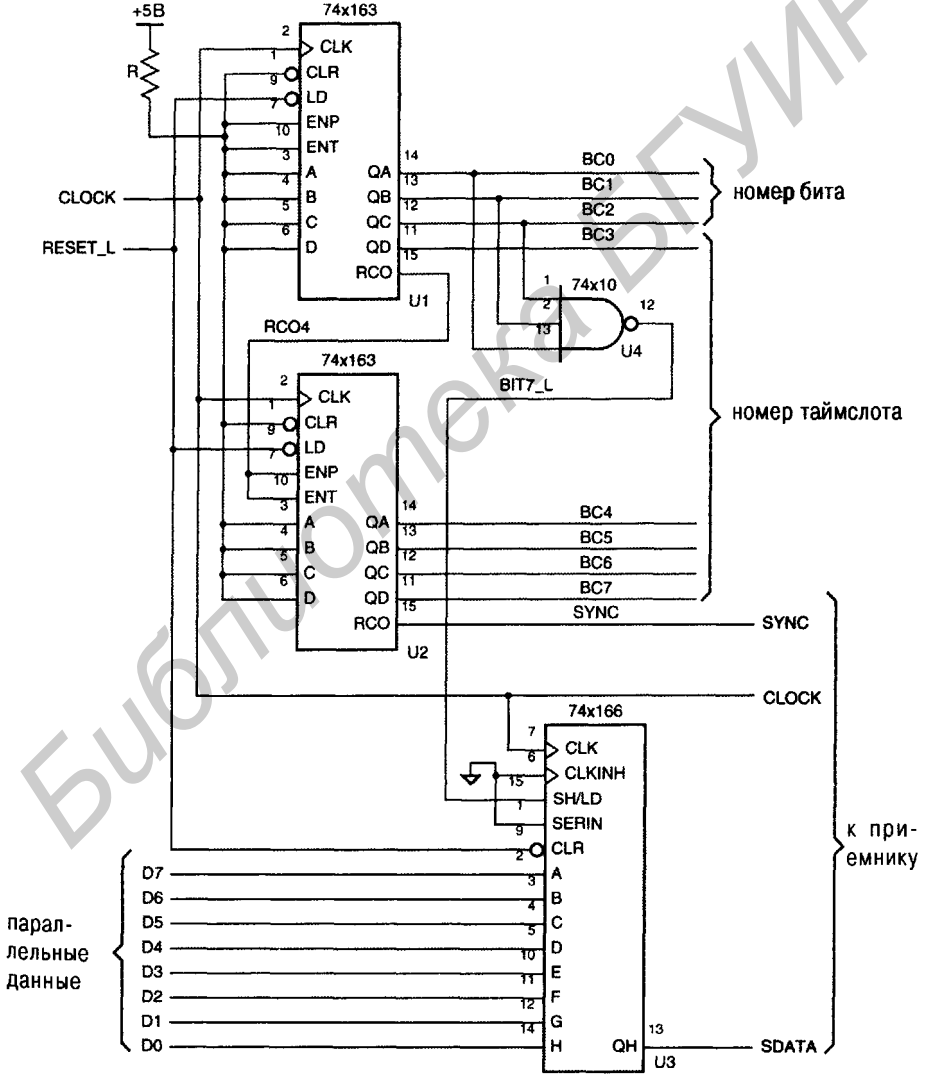


Рис.8.56. Преобразование параллельного кода в последовательный с помощью регистра сдвига с параллельным вводом

КАКОЙ БИТ ПЕРВЫЙ?

В действительности, в большинстве последовательных каналов передачи оцифрованного речевого сигнала первым передается 7-й бит, поскольку он первым появляется на выходе аналого-цифрового преобразователя, переводящего речевой сигнал в цифровую форму. Однако, ради простоты, мы указываем в наших примерах, что первым передается 0-й бит, так что в состояние счетчика номер бита входит непосредственно.

Регистр сдвига с параллельным вводом 74x166 осуществляет преобразование параллельного кода в последовательный. 0-й бит параллельных данных (D0–D7) подается на вход ИС '166, ближайший к выходу SDATA, так что биты передаются последовательно в порядке 0, ..., 7. При передаче 7-го бита в каждом таймслоте вырабатывается сигнал BIT7_L, который приводит к загрузке ИС '166 данными D0–D7. Значения сигналов на входах D0–D7 несущественны в течение всего времени, за исключением времени установления и времени удержания в окрестности того перепада в тактовом сигнале, на котором ИС '166 загружается; интервалы времени, в пределах которых значения сигналов на входах данных безразличны, на временны*х* диаграммах заштрихованы. Из этого следует, что шиной, по которой поступают параллельные данные, в другое время можно пользоваться для решения каких-то других задач (см. задачу 8.54).

В модуле-приемнике преобразование последовательных данных обратно в параллельный формат может осуществляться схемой, приведенной на рис. 8.57. Счетчик по модулю 256, состоящий из двух ИС '163, позволяет восстановить номера таймслотов и битов. Поскольку сигнал SINC вырабатывается в то время, когда счетчик в модуле-передатчике находится в состоянии 255, и по этому сигналу выполняется загрузка в счетчик модуля-приемника нулевого содержимого, оба счетчика переходят в нулевое состояние по одному и тому же фронту тактового сигнала. Старшие биты счетчика (номер таймслота) никак не используются на рисунке, но они могут позволить другим схемам в модуле-приемнике идентифицировать байты, удерживаемые на шине параллельных данных (PD0–PD7) в пределах того или иного таймслота.

На рис. 8.58 приведены подробные временны*е* диаграммы для схемы, осуществляющей преобразование последовательного кода в параллельный. Полностью принятый байт присутствует на параллельном выходе регистра сдвига 74x164 в течение периода тактового сигнала, следующего за приемом последнего (7-го) бита в байте. В нашем примере параллельные *данные дважды буферизованы* (*double-buffered data*): будучи полностью приняты, они переносятся в регистр 74x377, на выходах которого PD0–PD7 они доступны другим частям системы в течение восьми полных периодов тактового сигнала до окончания приема следующего байта. Сигнал разрешения BIT0_L обеспечивает загрузку ИС '377 в надлежащий момент времени. При наличии дополнительных регистров и декодирования можно было бы загружать байты из различных таймслотов в соответствующие регистры, в которых каждый байт удерживался бы на протяжении 125 мкс (см. задачу 8.57).

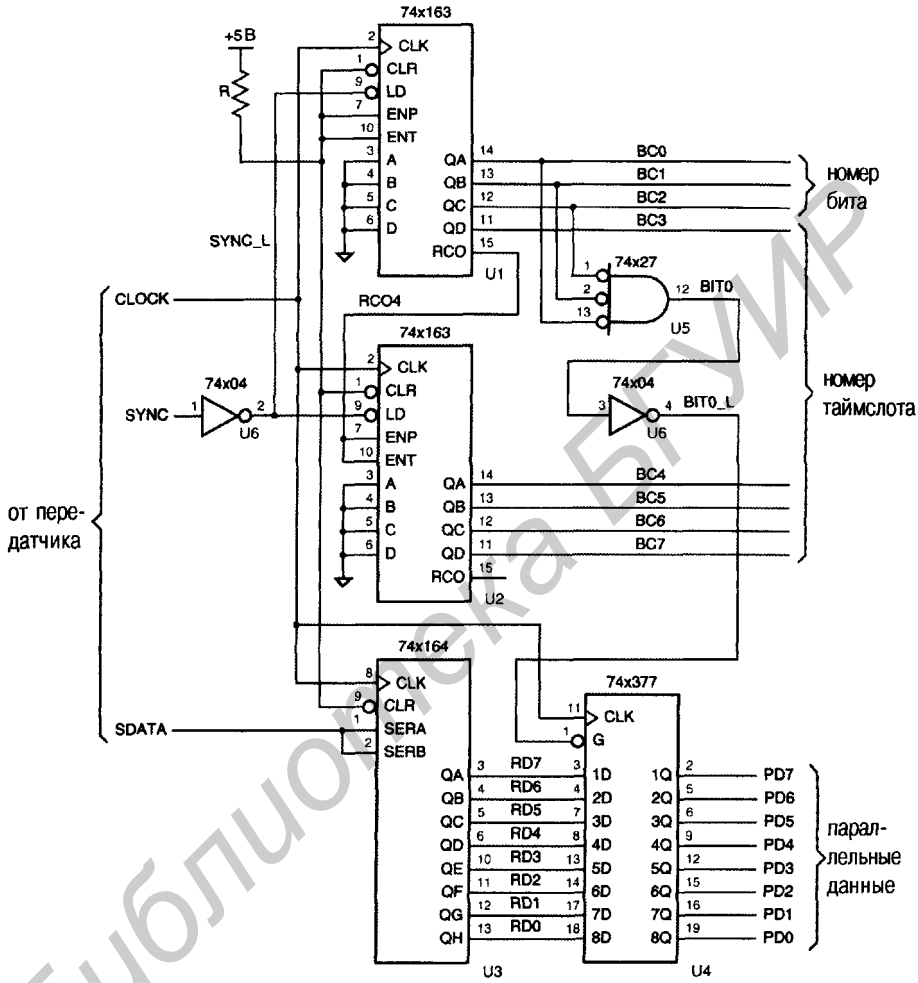


Рис. 8.57. Преобразование последовательного кода в параллельный с помощью регистра сдвига с параллельным выводом

Представленные в параллельном формате принятые данные легко запоминать и модифицировать в других цифровых схемах; в разделе 10.1.6 будут приведены соответствующие примеры. В цифровой телефонии принятые параллельные данные преобразуются обратно в аналоговое напряжение, которое фильтруется и отправляется в телефонную трубку или на громкоговоритель в течение 125 мкс, то есть до тех пор, пока не поступит следующее выборочное значение речевого сигнала.

ПРЯМОЙ И ОБРАТНЫЙ ПОРЯДОК СЛЕДОВАНИЯ

В истории развития цифровых систем был момент, когда обсуждение вопроса о том, в каком порядке нужно передавать биты и байты, стало носить характер религиозного спора. В своей знаменитой статье «Священные войны и призыв к миру» ("On Holy Wars and a Plea for Peace". *Computer*, October 1981, pp. 48–54) Дэнни Коэн (Danny Cohen) описал различие между соглашениями о порядке следования битов и байтов и указал на возможные (и проявившиеся в дальнейшем) отрицательные последствия этого различия.

Твердый стандарт так и не был установлен, и сегодня существуют популярные семейства компьютеров, в которых принят порядок нумерации и передачи байтов 32-разрядного слова, начиная с младшего байта (так называемые IBM-совместимые компьютеры) и начиная со старшего байта (компьютеры Apple Macintosh). Согласно терминологии Коэна, в первом случае говорят о прямом порядке следования ("Little Endian"), а во втором – об обратном порядке следования ("Big Endian"), и по-прежнему продолжается дискуссия о том, какой из них предпочтительнее (about "endianness"), как если бы это что-нибудь значило.

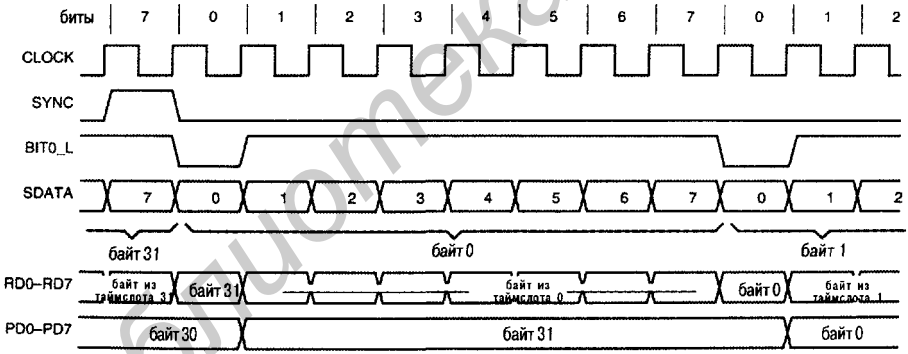


Рис. 8.58. Временные диаграммы для преобразования последовательного кода в параллельный

8.5.5. Счетчики на регистрах сдвига

Последовательно/параллельное преобразование представляет собой «обработку» данных, но регистры сдвига применяются также и в тех случаях, когда речь не идет о «данных». В результате объединения регистра сдвига с комбинационной логикой образуется конечный автомат, у которого диаграмма состояний является циклической. Такую схему называют *счетчиком на регистре сдвига* (*shift-register counter*). В отличие от двоичного счетчика последовательность состояний счетчика на регистре сдвига не образует ряд двоичных чисел, перебираемых в сторону увеличения или уменьшения, но такая схема все же полезна во многих приложениях, связанных с «управлением».

8.5.6. Кольцевые счетчики

В простейшем случае, используя *n*-разрядный регистр сдвига, можно получить счетчик с *n* состояниями, называемый *кольцевым счетчиком* (*ring counter*). На рис. 8.59 показана схема кольцевого счетчика. Универсальный регистр сдвига 74x194 включен так, что в нем обычно происходит сдвиг влево. Но если подан сигнал RESET, то в него загружается комбинация 0001 [см. функциональную таблицу ИС '194 (табл. 8.18)]. Если сигнал RESET снят, то на каждом такте происходит сдвиг содержимого ИС '194 влево. Последовательный вход LIN соединен с «крайним левым» выходом, так что последовательность состояний имеет вид: 0010, 0100, 1000, 0001, 0010, ... Следовательно, счетчик проходит через четыре различных состояния, прежде чем они начинают повторяться. На рис. 8.60 приведены соответствующие временные диаграммы. В общем случае *n*-разрядный кольцевой счетчик проходит в цикле через *n* состояний.

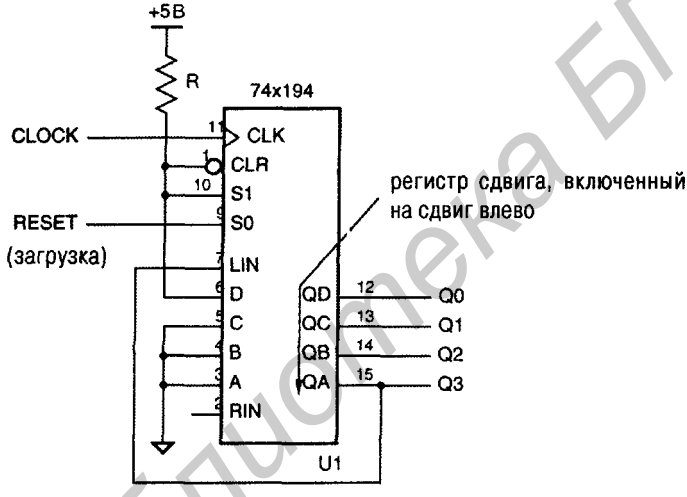


Рис. 8.59. Простейший 4-разрядный кольцевой счетчик с 4 состояниями, в котором циркулирует одна 1

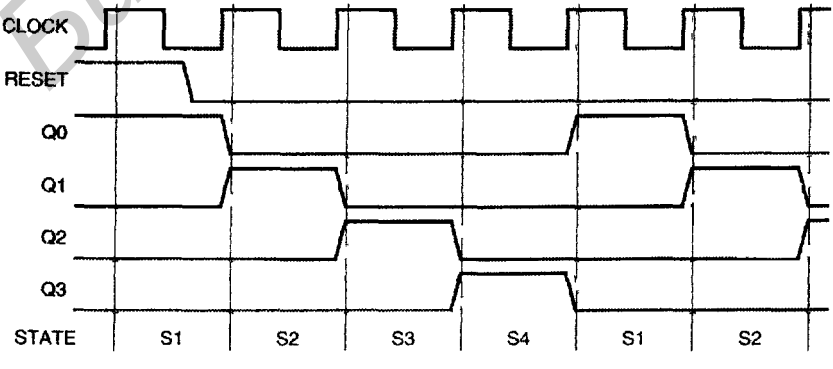


Рис. 8.60. Временные диаграммы для 4-разрядного кольцевого счетчика

У кольцевого счетчика на рис. 8.59 есть один недостаток: он не надежен. Если циркулирующая в нем единственная 1 будет потеряна вследствие временной аппаратной неисправности (например, из-за шумов), то счетчик войдет в состояние 0000 и останется в нем навсегда. Точно так же возникновение лишней 1 (например, переход в состояние 0101) вынудит счетчик в течение всего времени в дальнейшем проходить по неправильному циклу. Эти проблемы становятся вполне очевидными, если начертить *полную* диаграмму состояний такого счетчика, число которых равно 16. Как видно из рис. 8.61, 12 состояний не являются частью нормального цикла работы этой схемы. Если счетчик почему-либо выйдет из нормального цикла, то он уже не вернется в него.

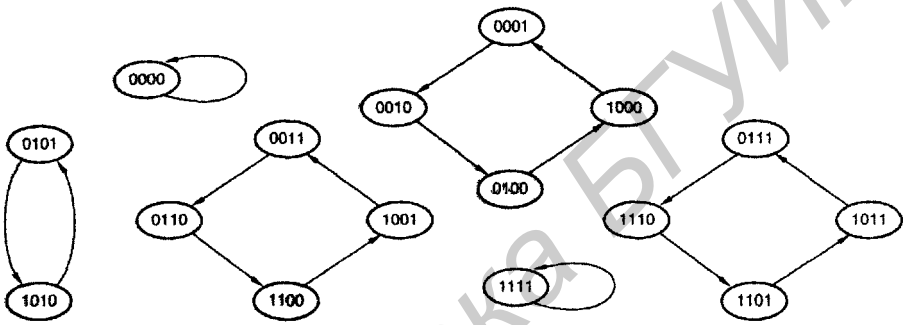


Рис. 8.61. Диаграмма состояний простого кольцевого счетчика

Самокорректирующийся счетчик (*self-correcting counter*) бывает построен так, что из всех неправильных состояний имеются переходы, приводящие в нормальные состояния. Самокорректирующиеся счетчики желательны по той же причине, какая служит основанием кодирования состояний по принципу минимального риска (см. раздел 7.4.3): если происходит что-то неожиданное, то счетчик или конечный автомат должен попадать в «безопасное» состояние.

На рис. 8.62 показана схема *самокорректирующегося кольцевого счетчика* (*self-correcting ring counter*). В ней с помощью вентиля ИЛИ-НЕ единица возникает на входе LIN только в том случае, когда содержимое трех младших разрядов равно 0. В результате получаем диаграмму состояний, приведенную на рис. 8.63; из всех неправильных состояний схема возвращается в нормальный цикл. Заметьте, что в этой схеме больше нет необходимости в подаче сигнала RESET. Независимо от начального состояния, в которое счетчик попадает при включении, он окажется в состоянии 0001 в пределах первых четырех тактов. Поэтому принудительный сброс требуется только в том случае, если нужно, чтобы счетчик начинал правильно работать синхронно с другими узлами в системе, или для задания известной начальной точки при моделировании.

В общем случае для n-разрядного самокорректирующегося кольцевого счетчика нужен $(n-1)$-входовой вентиль ИЛИ-НЕ, и такая схема входит в нормальное состояние не позднее, чем за $n-1$ тактов.

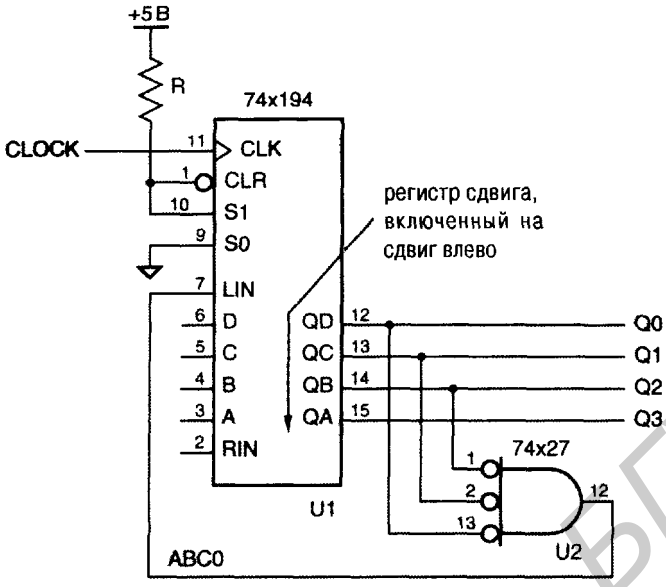


Рис. 8.62. Самокорректирующийся 4-разрядный кольцевой счетчик с 4 состояниями, в котором циркулирует одна 1

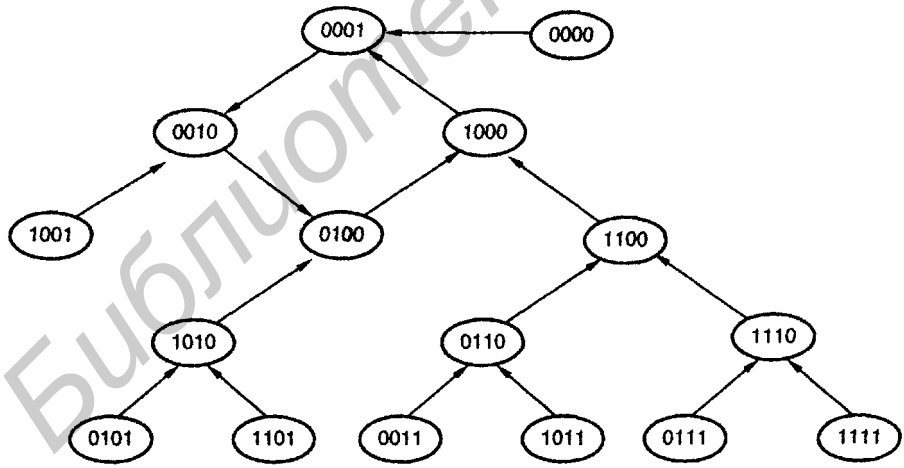


Рис. 8.63. Диаграмма состояний для самокорректирующегося кольцевого счетчика

В логических КМОП- и ТТЛ-семействах большое число входов чаще бывает у вентилей И-НЕ, а не у вентилей ИЛИ-НЕ, поэтому может оказаться более удобным построить самокорректирующийся кольцевой счетчик по схеме представленной на рис. 8.64. В каждом из состояний, образующих нормальный цикл такого счетчика, имеется только один 0.

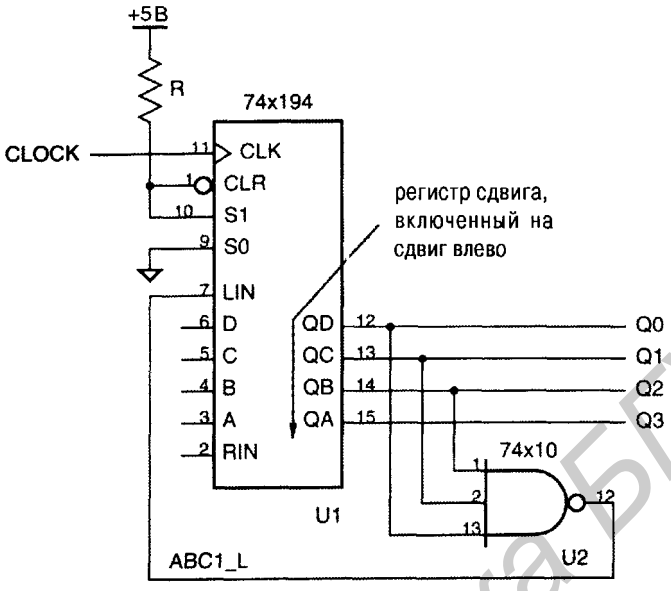


Рис. 8.64. Самокорректирующийся 4-разрядный кольцевой счетчик с 4 состояниями, в котором циркулирует один 0

Основное достоинство кольцевого счетчика с точки зрения его применения в задачах управления состоит в том, что его состояния, выражаемые совокупностью сигналов на выходах триггеров, являются словами кода "1 из *n*". Это значит, что всегда только один из выходных сигналов триггеров имеет активный уровень. Кроме того, в выходных сигналах кольцевых счетчиков нет паразитных импульсов; сравните этот подход со случаем, когда двоичный счетчик дополняется дешифратором (см. рис. 8.42).

*8.5.7. Счетчики Джонсона

У *n*-разрядного регистра сдвига с инвертором в цепи обратной связи между последовательным выходом и последовательным входом имеется 2*n* состояний. Такая конструкция носит название *скрученного кольцевого счетчика* (*twisted-ring counter*), *счетчика Мебиуса* (*Moebius counter*) или *счетчика Джонсона* (*Johnson counter*). Основная схема счетчика Джонсона показана на рис. 8.65, а его временные диаграммы – на рис. 8.66. Нормальные состояния этого счетчика перечислены в табл. 8.20. Как видно из таблицы, при наличии обоих выходных сигналов всех триггеров каждое нормальное состояние счетчика можно обнаружить с помощью 2-входового вентиля И или И-НЕ. На выходах этих вентилей нет паразитных импульсов.

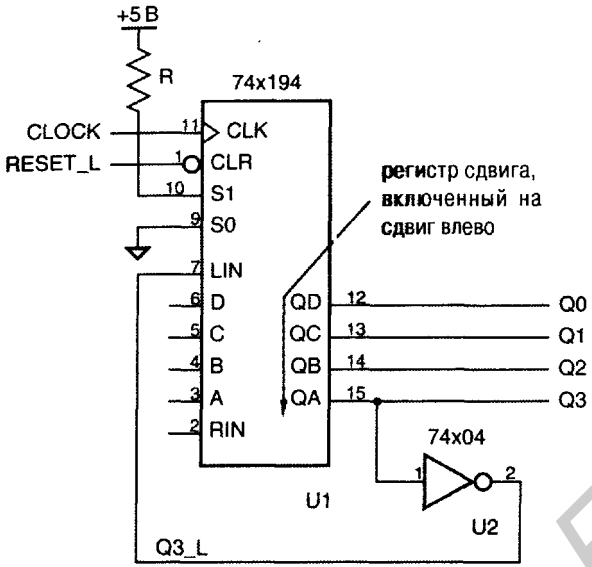


Рис. 8.65. Основная схема 4-разрядного счетчика Джонсона с 8 состояниями

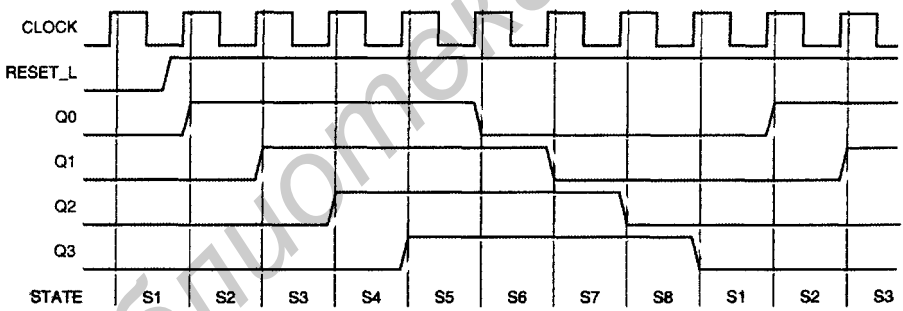


Рис. 8.66. Временные диаграммы для 4-разрядного счетчика Джонсона

Табл. 8.20. Состояния 4-разрядного счетчика Джонсона

Имя состояния	*Q3*	*Q2*	*Q1*	*Q0*	*Декодирование*
S1	0	0	0	0	$Q3' \cdot Q0'$
S2	0	0	0	1	$Q1' \cdot Q0$
S3	0	0	1	1	$Q2' \cdot Q1$
S4	0	1	1	1	$Q3' \cdot Q2$
S5	1	1	1	1	$Q3 \cdot Q0$
S6	1	1	1	0	$Q1 \cdot Q0'$
S7	1	1	0	0	$Q2 \cdot Q1'$
S8	1	0	0	0	$Q3 \cdot Q2'$

САМОКОРРЕКТИРУЮЩИЕСЯ СХЕМЫ ОБЕСПЕЧИВАЮТ ИСПРАВЛЕНИЕ САМИ ПО СЕБЕ!

Можно следующим образом доказать, что схема коррекции в самокорректирующемся счетчике Джонсона осуществляет исправление любого неправильного состояния. Неправильное состояние всегда можно представить в виде x...x10x...x, так как только нормальные состояния (00...00, 11...11, 01...1, 0...01...1 и 0...01) нельзя записать в таком виде. Поэтому не более чем через $n-2$ такта регистр сдвига будет содержать комбинацию 10x...x. На следующем такте его содержимое станет равным 0x...x0, и, такт спустя, в него будет загружено нормальное состояние 00...01.

У n-разрядного счетчика Джонсона есть 2^n-2n неправильных состояний, поэтому он также ненадежен, как и кольцевой счетчик. Но, как показано на рис. 8.67, можно построить *самокорректирующийся счетчик Джонсона* (*self-correcting Johnson counter*). В этой схеме происходит загрузка комбинации 0001 в качестве следующего состояния, если текущее состояние имеет вид 0xx0. По такому же принципу с помощью одного 2-входового вентиля ИЛИ-НЕ можно осуществлять коррекцию в счетчике Джонсона с любым числом разрядов. Схема коррекции должна загружать комбинацию 00...01 в качестве следующего состояния всякий раз, когда текущим оказывается состояние вида 0x...x0.

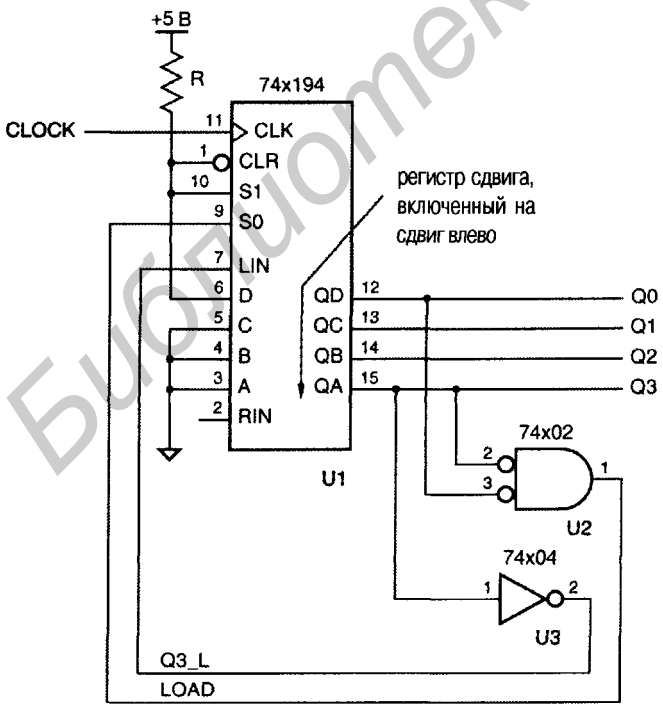


Рис. 8.67. Самокорректирующийся 4-разрядный счетчик Джонсона с 8 состояниями

*8.5.8. Счетчики на регистрах сдвига с линейной обратной связью

Число нормальных состояний у рассматривавшихся до сих пор счетчиков на n-разрядных регистрах сдвига было далеко от максимально возможного числа состояний, равного 2^n. Счетчик на основе n-разрядного *регистра сдвига с линейной обратной связью* (*linear feedback shift-register*, *LFSR*) имеет $2^n - 1$ состояний, то есть почти максимум. Такой счетчик часто называют *генератором последовательности максимальной длины* (*maximum-length sequence generator*).

LFSR-счетчики строятся на основе теории *конечных полей* (*finite fields*), развитой французским математиком Эваристом Галуа (1811–1832) незадолго до того, как он был убит на дуэли его политическим противником. В работе LFSR-счетчика реализуются операции над 2^n элементами в конечном поле.

На рис. 8.68 представлена структура n-разрядного LFSR-счетчика. На последовательный вход регистра сдвига поступает сумма по модулю 2 битов, содержащихся в определенном наборе разрядов регистра сдвига. Этой обратной связью определяется последовательность состояний, через которые проходит счетчик. Принято всегда нумеровать разряды так, как показано на рисунке, и считать, что сдвиг происходит в указанном направлении.

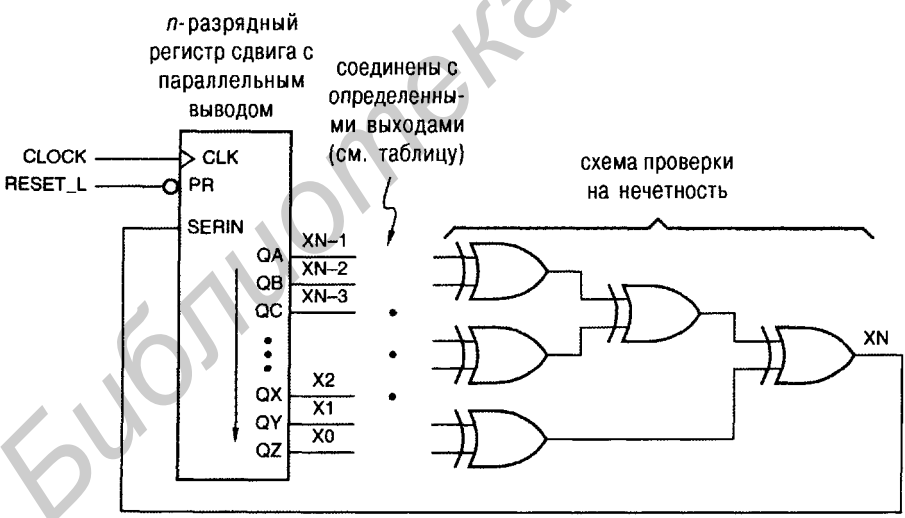


Рис. 8.68. Общая структура счетчика на основе регистра сдвига с линейной обратной связью

В табл. 8.21 для ряда значений n приведены уравнения, описывающие цепь обратной связи в тех случаях, когда результирующая последовательность оказывается последовательностью максимальной длины. Для каждого значения n больше 3-х существует много других уравнений обратной связи, обеспечивающих генерирование последовательностей максимальной длины, причем различным уравнениям соответствуют разные последовательности.

ДЕЙСТВИЯ В КОНЕЧНОМ ПОЛЕ

Конечное поле содержит конечное число элементов, и в нем определены две операции – сложение и умножение, – удовлетворяющие ряду требований. Примером конечного поля с P элементами, где P – простое число, может служить совокупность целых чисел по модулю P. Операциями в этом поле являются сложение и умножение по модулю P.

Согласно теории, конечные поля обладают следующим свойством: если вы начнете с ненулевого элемента E и станете многократно умножать его на так называемый «примитивный» элемент α, то в течение $P - 2$ шагов вы будете получать все другие ненулевые элементы поля, прежде чем снова возникнет элемент E. Оказывается, что в поле с P элементами любое целое число из интервала 2, ..., $P - 1$ является примитивным элементом. Вы можете убедиться в этом сами, взяв, например, $P = 7$ и $\alpha = 2$. Элементами поля при этом являются числа 0, 1, ..., 6, а операциями – сложение и умножение по модулю 7. (Здесь автор ошибается: не все элементы 2, ..., $P - 1$ являются примитивными; в частности, не является примитивным элемент 2. – *Прим. перев.*)

В предыдущем абзаце приведена центральная идея, на которой основывается теория генераторов последовательностей максимальной длины. Но для того, чтобы этой идеей можно было воспользоваться применительно к цифровым схемам, нам необходимо поле с 2^n элементами, где n – требуемое число разрядов. С одной стороны, нам повезло, так как Галуа доказал, что существуют конечные поля с P^n элементами при любом целом n, если только P – простое число, включая случай $P = 2$. Но, с другой стороны, приходится лишь сожалеть о том, что при $n > 1$ операции в полях с P^n элементами (в том числе с 2^n элементами) принципиально отличаются от обычных сложения и умножения целых чисел. Кроме того, труднее находить примитивные элементы.

Если вы, как и я, любите математику, то, должно быть, вас приводит в восхищение теория конечных полей, на основе которой строятся генераторы последовательностей максимальной длины и другие LFSR-схемы (см. Обзор литературы). В противном случае, доверьтесь и следуйте рекомендациям этого параграфа по тому же принципу, по которому вы пользуетесь рецептами из «Книги о вкусной и здоровой пище».

LFSR-счетчик со структурой, указанной на рис. 8.68, никогда не проходит в цикле через все возможные 2^n состояний. Независимо от конфигурации соединений следующим состоянием за тем, при котором во всех разрядах находятся нули, является то же самое состояние с нулями во всех разрядах

На рис. 8.69 показана принципиальная схема 3-разрядного LFSR-счетчика. Последовательность состояний этого счетчика приведена в левых трех столбцах табл. 8.22. Начиная с любого ненулевого состояния (в таблице – с состояния 100), счетчик проходит через семь состояний, прежде чем он возвращается в исходное состояние.

n	*Уравнения обратной связи*
2	X2 = X1 ⊕ X0
3	X3 = X1 ⊕ X0
4	X4 = X1 ⊕ X0
5	X5 = X2 ⊕ X0
6	X6 = X1 ⊕ X0
7	X7 = X3 ⊕ X0
8	X8 = X4 ⊕ X3 ⊕ X2 ⊕ X0
12	X12 = X6 ⊕ X4 ⊕ X1 ⊕ X0
16	X16 = X5 ⊕ X4 ⊕ X3 ⊕ X0
20	X20 = X3 ⊕ X0
24	X24 = X7 ⊕ X2 ⊕ X1 ⊕ X0
28	X28 = X3 ⊕ X0
32	X32 = X22 ⊕ X2 ⊕ X1 ⊕ X0

Табл. 8.21. Уравнения обратной связи для счетчиков на основе регистров сдвига с линейной обратной связью

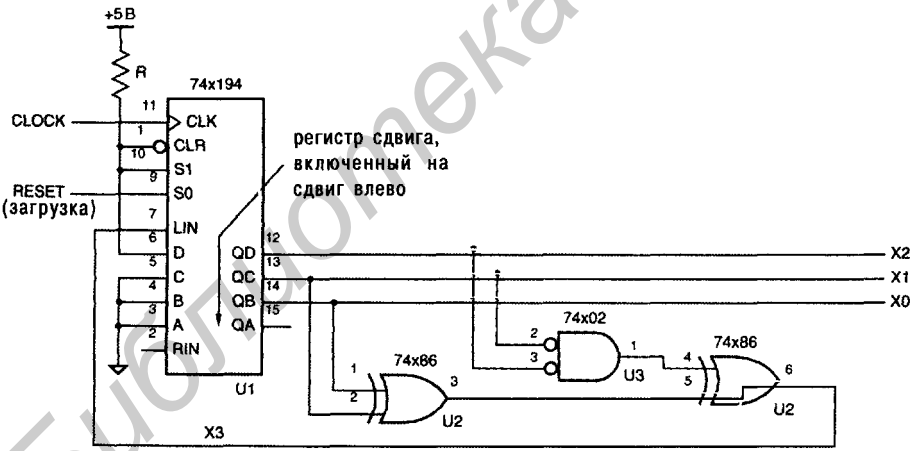


Рис. 8.69. 3-разрядный LFSR-счетчик; модификацией схемы, указанной синим цветом, достигается включение состояния со всеми нулями

Схему LFSR-счетчика можно видоизменить так, чтобы у него было 2^n состояний, включая состояние со всеми нулями; в схеме 3-разрядного счетчика, приведенной на рис. 8.69, синим цветом показано, как это сделать. В результате последовательность состояний будет такой, какая указана в правых трех столбцах табл. 8.22. То же самое можно сделать и в случае *n*-разрядного LFSR-счетчика: для этого необходимы вентиль ИСКЛЮЧАЮЩЕЕ ИЛИ и вентиль ИЛИ-НЕ с $n-1$ входами, которые должны быть подключены к выходам регистра сдвига, за исключением выхода X0.

Табл. 8.22. Последовательность состояний 3-разрядного LFSR-счетчика, приведенного на рис. 8.69

Исходная последовательность			*Модифицированная последовательность*		
X2	***X1***	***X0***	***X2***	***X1***	***X0***
1	0	0	1	0	0
0	1	0	0	1	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1
0	1	1	0	1	1
0	0	1	0	0	1
1	0	0	0	0	0
.	.	.	1	0	0
.	.	.	.	.	.

LFSR-счетчик переходит из одного состояния в другое не в порядке двоичного счета. Но во многих приложениях именно это свойство LFSR-счетчиков и является их достоинством. Основное применение LFSR-счетчиков состоит в генерировании тестовых входных сигналов для логических схем. В большинстве случаев «псевдослучайная» последовательность комбинаций, выдаваемых LFSR-счетчиком, предпочтительнее с точки зрения обнаружения ошибок, нежели последовательность комбинаций, перебираемых в порядке двоичного счета. Кроме того, LFSR-схемы используются для кодирования и декодирования применительно к кодам определенного вида, обнаруживающим и исправляющим ошибки, в том числе – к циклическим кодам, рассмотренным в разделе 2.15.4.

При передаче данных LFSR-счетчик часто является одним из узлов высокоскоростных модемов и сетевых интерфейсов, с помощью которых осуществляется «скремблирование» и «дескремблирование» данных. Достигается это путем пропускания выходного сигнала LFSR-схемы и потока данных пользователя через элемент ИСКЛЮЧАЮЩЕЕ ИЛИ. Даже в том случае, когда в потоке данных пользователя имеются длинные последовательности нулей и единиц, смешивание их с псевдослучайным выходным сигналом LFSR-схемы улучшает баланс передаваемого сигнала по постоянному току и создает достаточно большое число переходов, облегчающих извлечение приемником информации о тактовом сигнале.

8.5.9. Описание регистров сдвига на языке ABEL и их реализация в ПЛУ

На языке ABEL совсем легко описывать регистры сдвига общего назначения, а также эффективно размещать их в типичных последовательностных ПЛУ. На рис. 8.70, например, и в табл. 8.23 показано, как с помощью ИС 16V8 реализовать функции, подобные тем, которые выполняет универсальный регистр сдвига 74x194. Обратите внимание: один из выводов I/O (вывод 12 ИС 16V8) используется как вход.

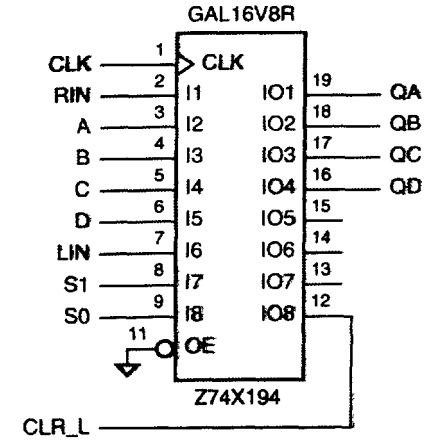


Рис. 8.70. Реализация в ПЛУ функций универсального регистра сдвига 74x194 с синхронным входом сброса

Табл. 8.23. Программа на языке ABEL для 4-разрядного универсального регистра сдвига

```
module Z74x194
title '4-bit Universal Shift Register'
Z74X194 device 'P16V8R';

" Input and output pins
CLK, RIN, A, B, C, D, LIN                pin 1, 2, 3, 4, 5, 6, 7;
S1, S0, CLR_L                            pin 8, 9, 12;
QA, QB, QC, QD                          pin 19, 18, 17, 16 istype 'reg';

" Active-level translation
CLR = !CLR_L;

" Set definitions
INPUT   = [  A,  B,  C, D  ];
LEFTIN  = [ QB, QC, QD, LIN];
RIGHTIN = [RIN, QA, QB, QC ];
OUT     = [ QA, QB, QC, QD ];

CTRL  = [S1,S0];
HOLD  = (CTRL == [0,0]);
RIGHT = (CTRL == [0,1]);
LEFT  = (CTRL == [1,0]);
LOAD  = (CTRL == [1,1]);

equations
OUT.CLK = CLK;

OUT := !CLR & (
           HOLD & OUT
         # RIGHT & RIGHTIN
         # LEFT & LEFTIN
         # LOAD & INPUT);

end Z74x194
```

Реализация в ИС 16V8 функций регистра '194 отличается от прототипа только тем, как действует входной сигнал CLR_L. У настоящей ИС '194 входной сигнал CLR_L является асинхронным, тогда как ИС 16V8 чувствительна к значению этого сигнала – так же, как и к значению всех других входных сигналов, – только на нарастающем фронте тактового сигнала CLK.

Табл. 8.24. Программа на языке ABEL для регистра сдвига со многими функциональными возможностями

```
module shifty
title '8-bit shift register with decoded load'

" Inputs and Outputs
CLK, OP3..OP0                pin;
Q7..Q0                       pin istype 'reg';

" Definitions

Q = [Q7..Q0];
OP = [OP3..OP0];

HOLD = (OP == 0);
CLEAR = (OP == 1);
LEFT = (OP == 2);
RIGHT = (OP == 3);
NOP = (OP >= 4) & (OP < 8);
LOADQ0 = (OP == 8);
LOADQ1 = (OP == 9);
LOADQ2 = (OP == 10);
LOADQ3 = (OP == 11);
LOADQ4 = (OP == 12);
LOADQ5 = (OP == 13);
LOADQ6 = (OP == 14);
LOADQ7 = (OP == 15);

Equations

Q.CLK = CLK;

WHEN HOLD THEN Q := Q;
ELSE WHEN CLEAR THEN Q := 0;
ELSE WHEN LEFT THEN Q := [Q6..Q0, Q7];
ELSE WHEN RIGHT THEN Q := [Q0, Q7..Q1];
ELSE WHEN LOADQ0 THEN Q := 1;
ELSE WHEN LOADQ1 THEN Q := 2;
ELSE WHEN LOADQ2 THEN Q := 4;
ELSE WHEN LOADQ3 THEN Q := 8;
ELSE WHEN LOADQ4 THEN Q := 16;
ELSE WHEN LOADQ5 THEN Q := 32;
ELSE WHEN LOADQ6 THEN Q := 64;
ELSE WHEN LOADQ7 THEN Q := 128;
ELSE Q := Q;

end shifty
```

Если вам, на самом деле, нужно осуществлять сброс асинхронно, то можно воспользоваться ИС 22V10, где имеется отдельная линия объединения входных сигналов по И для вырабатывания сигнала на входах сброса всех триггеров (см. задачу 8.69).

Гибкость языка ABEL позволяет создавать регистры сдвига с бóльшими или отличными от уже рассмотренных функциональными возможностями. В табл. 8.24, например, описан 8-разрядный регистр сдвига с входом сброса, с возможностью загрузки одной 1 в любой разряд и с функциями сдвига влево, сдвига вправо и хранения. Операция, выполняемая на каждом такте, определяется 4-разрядным кодом операции OP[3:0]. Несмотря на большое разнообразие условий, выражаемых операторами "WHEN", оказывается возможным синтезировать схему, у которой на каждый выход приходится всего лишь пять термов-произведений.

С помощью языка ABEL легко создавать различного типа счетчики на регистрах сдвига, о которых говорилось в предыдущих разделах. В табл. 8.25, например, приведена программа для 8-разрядного кольцевого счетчика. Мы воспользовались здесь предоставляемой ПЛУ дополнительной возможностью и добавили еще две функции, которых не было в рассмотренном выше случае применения ИС средней степени интеграции: счет происходит только тогда, когда подан сигнал разрешения CNTEN и схема принудительно переводится в состояние *S0* сигналом *RESTART*.

Табл. 8.25. Программа для 8-разрядного кольцевого счетчика

```
module Ring8
title '8-bit Ring Counter'

" Inputs and Outputs
MCLK, CNTEN, RESTART                    pin;
S0..S7                                  pin istype 'reg';

equations

[S0..S7].CLK = MCLK;

S0 := CNTEN & !S0 & !S1 & !S2 & !S3 & !S4 & !S5 & !S6   " Self-sync
    # !CNTEN & S0                                       " Hold
    # RESTART;                                          " Start with one 1
[S1..S7] := !RESTART & ( !CNTEN & [S1..S7]      " Shift
                       # CNTEN & [S0..S6] );    " Hold
end Ring8
```

Кольцевые счетчики часто применяются в цифровых системах для генерирования многофазных тактовых сигналов и сигналов разрешения, потребность в которых велика и требования к которым в различных системах весьма разнообразны. Особым достоинством проектирования на языках описания схем является возможность легко изменять поведение счетчика посредством перепрограммирования.

На рис. 8.71 показана совокупность тактовых сигналов или сигналов разрешения; такие сигналы могут понадобиться цифровой системе, работу которой можно разбить на шесть различных фаз. Система остается в каждой фазе в течение двух тактов основного тактового сигнала MCLK: на протяжении этого интер-

вала времени вырабатывается соответствующий сигнал Pi_L с низким активным уровнем, указывающий на пребывание в данной фазе. Такого рода временные диаграммы можно получить с помощью кольцевого счетчика, добавив еще один триггер, для отсчета двух тактов в каждой фазе так, чтобы сдвиг происходил *на втором* такте в пределах каждой фазы.

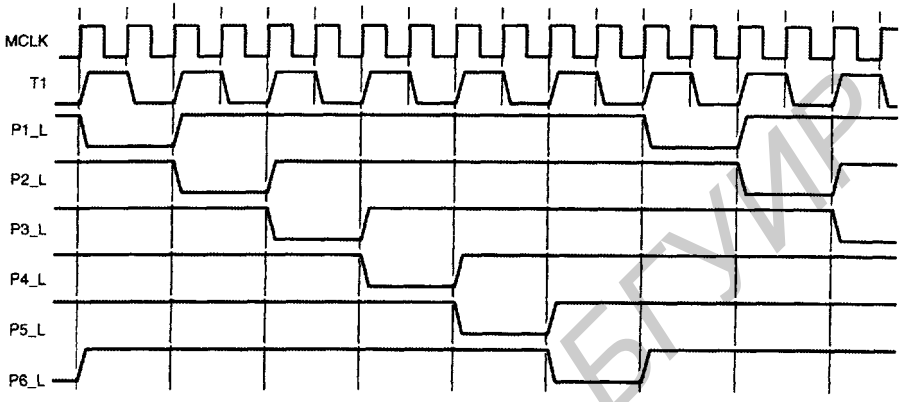


Рис. 8.71. Временные диаграммы совокупности управляющих сигналов с шестью фазами, которые могут понадобиться в той или иной цифровой системе

Табл. 8.26. Программа для генератора шестифазных колебаний

```
module TIMEGEN6
title 'Six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                    pin;
T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L       pin istype 'reg';

" State definitions
PHASES = [P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];
NEXTPH = [P6_L, P1_L, P2_L, P3_L, P4_L, P5_L];
SRESET = [1, 1, 1, 1, 1, 1];
P1 =     [0, 1, 1, 1, 1, 1];

equations
T1.CLK = MCLK; PHASES.CLK = MCLK;

WHEN RESET THEN {T1 := 1; PHASES := SRESET;}
ELSE WHEN (PHASES==SRESET) # RESTART THEN {T1 := 1; PHASES := P1;}
ELSE WHEN RUN & T1 THEN {T1 := 0; PHASES := PHASES;}
ELSE WHEN RUN & !T1 THEN {T1 := 1; PHASES := NEXTPH;}
ELSE {T1 := T1; PHASES := PHASES;}

end TIMEGEN6
```

Генератор таких многофазных сигналов можно создать в ПЛУ, предусмотрев наличие нескольких входов и выходов. Тремя управляющими входными сигналами задается следующее поведение устройства:

RESET — Когда подан этот сигнал, все выходные сигналы имеют неактивный уровень. После того, как сигнал RESET снимается, счетчик всегда переходит к первому такту фазы 1.

RUN — Наличие сигнала на этом входе позволят счетчику перейти ко второму такту в текущей фазе или к первому такту следующей фазы; в противном случае на данном такте продолжится пребывание в текущей фазе.

RESTART — Подача этого сигнала вызывает возврат счетчика к первому такту фазы 1 даже в том случае, когда действует сигнал RUN.

Табл. 8.26 представляет собой программу, посредством которой обеспечивается требуемое поведение. Заметьте, что для задания требуемой реакции на сигналы RESET, RESTART и RUN в любой фазе работы счетчика весьма эффективно использованы наборы.

Генератор, вырабатывающий точно такие же сигналы, какие указаны на рис. 8.71, можно представить в виде конечного автомата, как это сделано в табл. 8.27. Эта программа на языке ABEL длиннее предыдущей, но поведение синтезируемых по этим программам устройств одинаково, тогда как последнюю программу – с определенной точки зрения – легче понять. Все же реализация данной программы требует от 8 до 20 термов И на выход при необходимости только 3–5 термов-произведений на выход в исходном варианте кольцевого счетчика. Это хороший пример того, как можно повысить эффективность расходования схемных ресурсов и получить лучшие характеристики, приспосабливая к «требованиям заказчика» простую стандартную структуру, а не вымучивать из себя решение задачи «в лоб» путем построения конечного автомата.

Давайте рассмотрим теперь другой вариант многофазных колебаний, которые также могут понадобиться в той или иной системе. На рис. 8.72 приведены временные диаграммы колебаний, подобных рассмотренным выше, но отличающихся тем, что выходные сигналы Ri_L, посредством которых отмечаются разные фазы, удерживаются на активном уровне только в пределах одного периода тактового сигнала в каждой из фаз. Это небольшое отличие может быть очень важным для проекта в целом.

В исходном варианте мы воспользовались 6-разрядным кольцевым счетчиком и одной дополнительной переменной состояния T1, с помощью которой отображались два состояния в пределах каждой фазы. В случае новых колебаний так поступить нельзя. В состояниях, которые имеют место между импульсами с низким активным уровнем (STATE = 0, 2, 4 и т.д. на рис. 8.72), все выходные сигналы принимают неактивное значение, так что по ним уже нельзя судить о том, в какое следующее состояние следует перейти. Необходимо что-то другое для отслеживания цепочки состояний.

Табл. 8.27. Альтернативный вариант программы для генератора шестифазных колебаний

```
module TIMEGN6A
title 'Six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                    pin;
T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L       pin istype 'reg';

" State definitions
TSTATE = [T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];
SRESET = [1, 1, 1, 1, 1, 1, 1];
P1F =    [1, 0, 1, 1, 1, 1, 1];
P1S =    [0, 0, 1, 1, 1, 1, 1];
P2F =    [1, 1, 0, 1, 1, 1, 1];
P2S =    [0, 1, 0, 1, 1, 1, 1];
P3F =    [1, 1, 1, 0, 1, 1, 1];
P3S =    [0, 1, 1, 0, 1, 1, 1];
P4F =    [1, 1, 1, 1, 0, 1, 1];
P4S =    [0, 1, 1, 1, 0, 1, 1];
P5F =    [1, 1, 1, 1, 1, 0, 1];
P5S =    [0, 1, 1, 1, 1, 0, 1];
P6F =    [1, 1, 1, 1, 1, 1, 0];
P6S =    [0, 1, 1, 1, 1, 1, 0];

equations
TSTATE.CLK = MCLK;
WHEN RESET THEN TSTATE := SRESET;

state_diagram TSTATE

state SRESET: IF RESET THEN SRESET ELSE P1F;

state P1F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P1S ELSE P1F;

state P1S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P2F ELSE P1S;

state P2F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P2S ELSE P2F;

state P2S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P3F ELSE P2S;

state P3F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P3S ELSE P3F;

state P3S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P4F ELSE P3S;

state P4F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P4S ELSE P4F;
```

Табл. 8.27. Альтернативный вариант программы для генератора шестифазных колебаний (*продолжение*)

```
state P4S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P5F ELSE P4S;

state P5F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P5S ELSE P5F;

state P5S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P6F ELSE P5S;

state P6F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P6S ELSE P6F;

state P6S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P1F ELSE P6S;

end TIMEGN6A
```

НАДЕЖНЫЙ СБРОС

Заметьте, что в табл. 8.27 значение присваивается набору TSTATE в разделе equations, а результат используется в разделе state_diagram. Мы сейчас объясним, что это сделано с вполне определенной целью: обеспечить в программе переход в состояние SRESET из любого не определенного состояния.

Программа ABEL *пополняет множество включений*, относящееся к данному выходу, всякий раз, когда сигнал на этом выходе встречается в левой части равенства (см. раздел 4.6.3, где этот вопрос был рассмотрен применительно к комбинационным выходам). В случае регистровых выходов множество включений для каждого состояния в векторе состояний увеличивается на единицу при каждом упоминании "state" в разделе state_diagram. Всякая комбинация входных сигналов, *которая вызывает появление* 1 на выходе для каждой переменной состояния, добавляется в множество включений очередным предложением, начинающимся с ключевого слова state.

У конечного автомата, задаваемого программой в табл. 8.27, всего $2^7 = 128$ состояний, из которых явно определены только 13, и только для них указаны переходы в состояние SRESET. Но равенства, содержащие операторы WHEN, гарантируют переход автомата в состояние SRESET, когда бы *ни возник сигнал* RESET. Это справедливо независимо от определений state в разделе state_diagram. Когда сигнал RESET переходит на активный уровень, содержащее одни единицы кодовое имя состояния SRESET, в действительности, объединяется по правилу ИЛИ со следующим состоянием, если только оно задается в разделе state_diagram. При таком подходе нельзя обеспечить надежный сброс, если кодовое имя состояния SRESET содержит, *например*, только нули.

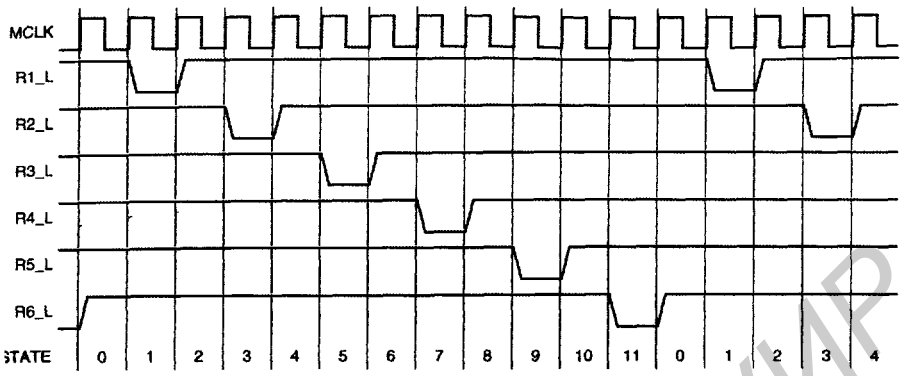


Рис. 8.72. Временные диаграммы для видоизмененных многофазных колебаний

Эту проблему можно решить многими способами. Одна из идей состоит в том, чтобы выбрать в качестве отправной точки исходный проект (табл. 8.26), используя фазовые сигналы P1_L, P2_L и т.д. только в качестве указателей на внутренние состояния. Тогда можно считать, что каждый фазовый сигнал Ri_L является комбинационным выходом автомата Мура, принимающим активное значение, когда соответствующий сигнал Pi_L имеет активный уровень *и* схема находится на втором такте в пределах данной фазы. Для реализации этого первого подхода программу на языке ABEL нужно дополнить строками, приведенными в табл. 8.28.

Табл. 8.28. Добавления в программу, содержащуюся в табл. 8.26, для модифицированного генератора шестифазных колебаний

```
odule TIMEG12K
..
1_L, R2_L, R3_L, R4_L, R5_L, R6_L          pin istype 'com';
..
UTPUTS = [R1_L, R2_L, R3_L, R4_L, R5_L, R6_L];

quations
..
OUTPUTS = !PHASES & !T1;

nd TIMEG12K
```

Этот первый подход легко реализовать и он дает прекрасные результаты, если мы собираемся использовать сигналы Ri_L только в качестве сигналов разрешения или других управляющих входных сигналов. Но это плохая идея в том случае, когда данным сигналам предстоит играть роль тактовых сигналов, поскольку, как мы сейчас объясним, у них могут быть паразитные импульсы. Сигналы T_L и T1 являются выходными сигналами триггеров, переключающихся одним и тем же основным тактовым сигналом MCLK. Хотя сигналы изменяются на выходах триггеров примерно в одно и то же время, это никогда не происходит точно

одновременно. Один из них может изменяться быстрее, а другой – медленнее; это обстоятельство называется *расхождением выходных сигналов по времени* (*output timing skew*). Предположим, например, что при переходе из состояния 1 в состояние 2 на рис. 8.71 низкий уровень появляется в сигнале P2_L раньше, чем уровень сигнала T1 становится высоким. В этом случае в выходном сигнале R2_L может возникнуть короткий паразитный импульс.

Чтобы обеспечить отсутствие паразитных импульсов, мы должны разработать схему, у которой каждый фазовый сигнал был бы регистровым выходным сигналом. Один из способов достичь этого заключается в применении 12-разрядного кольцевого счетчика; для получения сигналов желаемого вида используются выходы только каждого второго триггера. Программа на языке ABEL, реализующая этот подход, приведена в табл. 8.29.

Табл. 8.29. Программа на языке ABEL для модифицированного генератора шестифазных колебаний

```
module TIMEG12
title 'Modified six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                    pin;
P1_L, P2_L, P3_L, P4_L, P5_L, P6_L           pin istype 'reg';
P1A, P2A, P3A, P4A, P5A, P6A                 pin istype 'reg';

" State definitions
PHASES = [P1A, P1_L, P2A, P2_L, P3A, P3_L, P4A, P4_L, P5A, P5_L, P6A, P6_L];
NEXTPH = [P6_L, P1A, P1_L, P2A, P2_L, P3A, P3_L, P4A, P4_L, P5A, P5_L, P6A];
SRESET = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
P1 =     [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];

equations

PHASES.CLK = MCLK;

WHEN RESET THEN PHASES := SRESET;
ELSE WHEN RESTART # (PHASES == SRESET) THEN PHASES := P1;
ELSE WHEN RUN THEN PHASES := NEXTPH;
ELSE PHASES := PHASES;

end TIMEG12
```

Еще один очевидный возможный путь – с учетом того, что в пределах цикла совокупность колебаний проходит через 12 состояний, – это построение двоичного счетчика по модулю 12 и дешифратора состояний такого счетчика. В табл. 8.30 представлена программа на языке ABEL, в которой использован данный подход. Состояниям счетчика соответствуют значения переменной “STATE” на рис. 8.72. Поскольку фазовые выходные сигналы объявлены как регистровые, в них нет паразитных импульсов. Заметьте, что для компенсации задержки при декодировании на один такт предусмотрено декодирование на один период тактового сигнала раньше. Кроме того, при сбросе счетчик переводится в состояние 15, а

не в состояние 0, чтобы при осуществлении сброса не возникло активное значение сигнала P1_L.

Табл. 8.30. Программа для генератора шестифазных колебаний на основе счетчика

```
module TIMEG12A
title 'Counter-based six-phase master timing generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                pin;
P1_L, P2_L, P3_L, P4_L, P5_L, P6_L       pin istype 'reg';
CNT3..CNT0                               pin istype 'reg';

" Definitions
CNT = [CNT3..CNT0];
P_L = [P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];

equations

CNT.CLK = MCLK;  P_L.CLK = MCLK;

WHEN RESET THEN CNT := 15
ELSE WHEN RESTART THEN CNT := 0
ELSE WHEN (RUN & (CNT < 11)) THEN CNT := CNT + 1
ELSE WHEN RUN THEN CNT := 0
ELSE CNT := CNT;

P1_L := !(CNT == 0);
P2_L := !(CNT == 2);
P3_L := !(CNT == 4);
P4_L := !(CNT == 6);
P5_L := !(CNT == 8);
P6_L := !(CNT == 10);

end TIMEG12A
```

8.5.10. Описание регистров сдвига на языке VHDL

На языке VHDL регистры сдвига можно задавать структурно и в поведенческом стиле; мы рассмотрим несколько поведенческих описаний и их применение. Табл. 8.31 представляет собой описание работы 8-разрядного регистра сдвига с расширенным набором функций. Помимо хранения, загрузки и сдвига, выполняемых ИС 74x194 и 74x299, в данном регистре осуществляются операции циклического и арифметического сдвига. При *циклическом сдвиге* (*circular shift*) бит, «теряющийся» в результате сдвига с одного конца регистра, поступает на другой его конец. При *арифметическом сдвиге* (*arithmetic shift*) значение бита, «заталкиваемого» в регистр, устанавливается из соображений умножения или деления на 2: при сдвиге влево на правый вход подается 0, а при сдвиге вправо повторяется крайний левый (знаковый) бит.

Табл. 8.31. Описание работы 8-разрядного регистра сдвига с расширенными функциями

Функция	Входы			Следующее состояние							
	S2	*S1*	*S0*	*Q7+*	*Q6+*	*Q5+*	*Q4+*	*Q3+*	*Q2+*	*Q1+*	*Q0+*
Хранение	0	0	0	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
Загрузка	0	0	1	D7	D6	D5	D4	D3	D2	D1	D0
Сдвиг вправо	0	1	0	RIN	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Сдвиг влево	0	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	LIN
Циклический сдвиг вправо	1	0	0	Q0	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Циклический сдвиг влево	1	0	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	Q7
Арифметический сдвиг вправо	1	1	0	Q7	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Арифметический сдвиг влево	1	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	0

В табл. 8.32 приведена поведенческая VHDL-программа для регистра сдвига с расширенными функциями. Как и в предыдущих примерах, определяется процесс и для обеспечения желаемого переключения по фронту тактового сигнала используется признак `event`. Заслуживают внимание следующие особенности этой программы:

- Введен внутренний сигнал `IQ`, который в конце концов становится выходным сигналом `Q`, но таким, что его могут читать и писать операторы процесса. Можно было бы поступить и иначе, определив тип выходного сигнала `Q` как "`buffer`".
- Вход `CLR` является асинхронным; поскольку этот сигнал входит в список чувствительности процесса, он проверяется всякий раз, когда претерпевает изменение. Операторам `IF` придана такая структура, что учет значения `CLR` предшествует анализу любого другого условия.
- Для определения операций, реализуемых регистром сдвига при восьми возможных значениях входных сигналов выбора `S(2 downto 0)`, применен оператор `CASE`.
- В операторе `CASE` необходимо предусмотреть случай "`when others`", чтобы предотвратить предупреждение компилятора о том, что примерно 2^{32} случаев остаются не принятыми во внимание.
- Оператор "`null`" указывает, что в некоторых случаях никаких действий производить не надо. Заметьте, что ничего не нужно делать в случае 0; бездействие зарезервировано в качестве сигнала удержания сохраняемых регистром данных до тех пор, пока не будет велено поступить иначе.
- В большинстве случаев для образования 8-разрядного массива из 7-разрядного подмножества `IQ` и еще одного бита применяется оператор конкатенации "`&`".
- Из-за строгих требований языка VHDL к согласованию типов в операторе `CASE` используется определенная в разделе 4.7.4 функция `CONV_INTEGER` для преобразования входного сигнала выбора `S` типа `STD_LOGIC_VECTOR` в целое число. Можно было бы сделать иначе, записав метку каждого случая как элемент типа `STD_LOGIC_VECTOR` [например: ('0','1','1'), а не целое число 3].

Табл. 8.32. VHDL-программа для 8-разрядного регистра сдвига с расширенными функциями

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vshftreg is
    port (
        CLK, CLR, RIN, LIN: in STD_LOGIC;
        S: in STD_LOGIC_VECTOR (2 downto 0); -- function select
        D: in STD_LOGIC_VECTOR (7 downto 0); -- data in
        Q: out STD_LOGIC_VECTOR (7 downto 0) -- data out
    );
end Vshftreg;

architecture Vshftreg_arch of Vshftreg is
signal IQ: STD_LOGIC_VECTOR (7 downto 0);
begin
process (CLK, CLR, IQ)
  begin
    if (CLR='1') then IQ <= (others=>'0'); -- Asynchronous clear
    elsif (CLK'event and CLK='1') then
      case CONV_INTEGER(S) is
        when 0 => null;                          -- Hold
        when 1 => IQ <= D;                       -- Load
        when 2 => IQ <= RIN & IQ(7 downto 1);    -- Shift right
        when 3 => IQ <= IQ(6 downto 0) & LIN;    -- Shift left
        when 4 => IQ <= IQ(0) & IQ(7 downto 1); -- Shift circular right
        when 5 => IQ <= IQ(6 downto 0) & IQ(7); -- Shift circular left
        when 6 => IQ <= IQ(7) & IQ(7 downto 1); -- Shift arithmetic right
        when 7 => IQ <= IQ(6 downto 0) & '0';    -- Shift arithmetic left
        when others => null;
      end case;
    end if;
    Q <= IQ;
  end process;
end Vshftreg_arch;
```

Одно из применений регистров сдвига – это кольцевые счетчики; примером такого применения служит рассмотренный в предыдущем разделе генератор шестифазных колебаний, изображенных на рис. 8.71. В табл. 8.33 приведена VHDL-программа, обеспечивающая такое же поведение устройства. Как и в предыдущем VHDL-примере для чтения и записи используется внутренний сигнальный вектор `IP` с высоким активным уровнем, становящийся в конце концов выходным сигналом устройства, чтобы получить требуемый выходной сигнальный вектор с низким активным уровнем, удобно инвертировать этот внутренний сигнал в последнем операторе. Остальная часть программы не содержит никаких особенностей, но заметьте, что у вложенного оператора `IF` имеются три уровня.

Табл. 8.33. VHDL-программа для генератора шестифазных колебаний

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vtimegn6 is
    port (
        MCLK, RESET, RUN, RESTART: in STD_LOGIC; -- clock, control inputs
        P_L: out STD_LOGIC_VECTOR (1 to 6)       -- active-low phase outputs
    );
end Vtimegn6;

architecture Vtimegn6_arch of Vtimegn6 is
signal IP: STD_LOGIC_VECTOR (1 to 6); -- internal active-high phase signals
signal T1: STD_LOGIC;                 -- first tick within phase
begin
process (MCLK, IP)
  begin
    if (MCLK'event and MCLK='1') then
      if (RESET='1') then
        T1 <= '1'; IP <= ('0','0','0','0','0','0');
      elsif ((IP=('0','0','0','0','0','0')) or (RESTART='1')) then
        T1 <= '1'; IP <= ('1','0','0','0','0','0');
      elsif (RUN='1') then
        T1 <= not T1;
        if (T1='0') then IP <= IP(6) & IP(1 to 5); end if;
      end if;
    end if;
    P_L <= not IP;
  end process;
end Vtimegn6_arch;
```

Возможной модификацией рассмотренного приложения является устройство, выходные колебания которого удерживаются на активном уровне только во втором такте каждой фазы длительностью в два такта; эти колебания были показаны на рис. 8.72. Один из способов достичь этого заключается в создании 12-разрядного кольцевого счетчика и использовании выходов только каждого второго триггера. При реализации такого устройства VHDL-средствами в определении объекта фигурировали бы только шесть фазовых выходных сигналов `P_L(1 to 6)`. Шесть дополнительных сигналов, названных `NEXT P(1 to 6)`, объявлены в определении архитектуры и являются локальными. На рис. 8.73 показано соотношение между этими сигналами при выполнении операции сдвига, а в табл. 8.34 приведена соответствующая VHDL-программа.

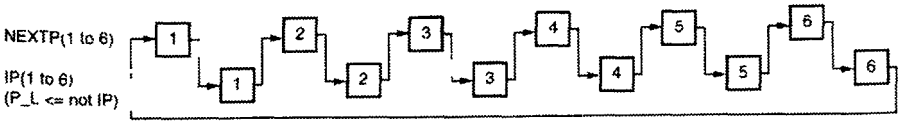


Рис. 8.73. Последовательность сдвигов в генераторе шестифазных колебаний на основе 12-разрядного кольцевого счетчика

Табл. 8.34. VHDL-программа для модифицированного генератора шестифазных колебаний

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vtimeg12 is
    port (
        MCLK, RESET, RUN, RESTART: in STD_LOGIC; -- clock, control inputs
        P_L: out STD_LOGIC_VECTOR (1 to 6)       -- active-low phase outputs
    );
end Vtimeg12;

architecture Vtimeg12_arch of Vtimeg12 is
signal IP, NEXTP: STD_LOGIC_VECTOR (1 to 6); -- internal active-high phase signals
begin
process (MCLK, IP, NEXTP)
  variable TEMP: STD_LOGIC_VECTOR (1 to 6);  -- temporary for signal shift
  constant IDLE: STD_LOGIC_VECTOR (1 to 6) := ('0','0','0','0','0','0');
  constant FIRSTP: STD_LOGIC_VECTOR (1 to 6) := ('1','0','0','0','0','0');
  begin
    if (MCLK'event and MCLK='1') then
      if (RESET='1') then IP <= IDLE;  NEXTP <= IDLE;
      elsif (RESTART='1') or (IP=IDLE and NEXTP=IDLE) then IP <= IDLE;  NEXTP <= FIRSTP;
      elsif (RUN='1') then
        if (IP=IDLE) and (NEXTP=IDLE) then NEXTP <= FIRSTP;
        else TEMP := IP;  IP <= NEXTP;  NEXTP <= TEMP(6) & TEMP(1 to 5);
        end if;
      end if;
    end if;
    P_L <= not IP;
  end process;
end Vtimeg12_arch;
```

Как и в предыдущей программе, в теле архитектуры объявлен 6-разрядный сигнал IP с высоким активным уровнем, который используется для чтения и записи и становится в конце концов выходным сигналом устройства P_L с низким активным уровнем. Добавочный 6-разрядный сигнал NEXTP хранит остающиеся 6 двоичных переменных состояния. Константы IDLE и FIRSTP делают программу более удобочитаемой.

Заметьте, что 6-разрядная переменная TEMP используется только как место временного хранения старого значения IP при осуществлении сдвига: в IP загружается *содержимое* NEXTP, а в NEXTP – сдвинутое старое значение IP. Поскольку операторы присваивания в процессе выполняются *последовательно*, мы не могли бы обойтись простой записью "IP <= NEXTP; NEXTP <= IP(6)&IP(1 to 5);". Если бы мы так сделали, то в NEXTP попало бы *новое* значение IP, а не старое. Обратите также внимание на следующее: так как TEMP является локальной переменной, а не сигналом, ее значение вне процесса не видно. Кроме того, при очередном обращении к процессу никогда не используется значение TEMP, образовавшееся при предыдущем вызове процесса. Поэтому VHDL-компилятору не нужно синтезировать никаких триггеров для хранения значения TEMP.

8.8. Трудности синхронного проектирования

Хотя синхронный подход является наиболее прямым и самым надежным способом проектирования цифровых систем, ряд неприятных практических обстоятельств может затруднить продвижение по этому пути. В данном параграфе мы рассмотрим эти обстоятельства.

8.8.1. Разброс задержек тактового сигнала

Синхронная система на переключающихся по фронту триггерах работает правильно только в том случае, когда переключающий фронт тактового сигнала поступает на все триггеры в один и тот же момент времени. На рис. 8.85 показано, что может произойти в противном случае. В этом примере два триггера теоретически управляются одним и тем же тактовым сигналом, но триггер FF2 «видит» тактовый сигнал, который заметно задержан по отношению к тактовому сигналу, который «видит» триггер FF1. Это различие между моментами прихода тактового сигнала на различные устройства носит название *разброса задержек тактового сигнала* (*clock skew*).

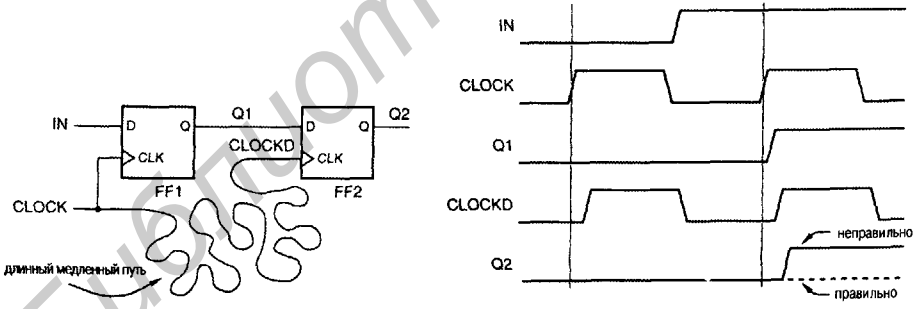


Рис. 8.85. Пример разброса задержек тактового сигнала

Мы назвали задержанный сигнал в схеме на рис. 8.85(а) "CLOCKD". Если задержка распространения в триггере FF1 от входа CLOCK до выхода Q1 мала и

если физическое соединение выхода Q1 с триггером FF2 является коротким, то изменение значения Q1 по фронту сигнала CLOCK может и в самом деле достичь триггера FF2 *до того*, как на этот триггер поступит фронт сигнала CLOCKD. В этом случае триггер FF2 может перейти в неправильное следующее состояние, определяемое *следующим* состоянием триггера FF1, а не текущим его состоянием, как показано на рис. (b). Если изменение значения Q1 приходит на триггер FF2 лишь немного раньше относительно сигнала CLOCKD, то может оказаться нарушенным требование, касающееся времени удержания триггера FF2, и в этом случае триггер FF2 может стать метастабильным и выработать непредсказуемое значение сигнала на своем выходе.

Если рис. 8.85 напоминает вам существенный источник опасности, изображенный на рис. 7.101, то вы не далеки от истины. Проблему разброса задержек тактового сигнала можно считать просто проявлением наличия существенных источников опасности в переключающихся по фронту устройствах.

Наличие или отсутствие проблемы, связанной с разбросом задержек тактового сигнала в данной системе можно описать количественно, определив t_{skew} как величину расхождения задержек тактового сигнала и воспользовавшись другими временными параметрами, указанными на рис. 8.1. Для правильной работы необходимо, чтобы выполнялось неравенство:

$$t_{ffpd(min)} + t_{comb(min)} - t_{hold} - t_{skew(max)} > 0.$$

Другими словами, разброс задержек тактового сигнала вычитается из запаса по времени удержания, введенного в разделе 8.1.4.

Приведенный на рис. 8.85 пример, рассматриваемый сам по себе, может показаться несколько преувеличенным. В конце концов, почему бы это конструктор стал создавать короткое соединение для данных и длинное соединение для тактового сигнала, когда пути этих сигналов вполне могли бы проходить рядом? Существует несколько ситуаций, в которых это все же может случиться; одни из них являются результатом ошибок, тогда как другие неизбежны.

В большой системе коэффициент разветвления по выходу у источника тактового сигнала может быть недостаточным для подачи одного и того же сигнала на тактовые входы всех устройств; поэтому могут понадобиться две или большее число копий тактового сигнала. Метод буферизации, указанный на рис. 8.86(a), очевидно, вносит дополнительный разброс задержек тактового сигнала, поскольку сигналы CLOCK1 и CLOCK2 оказываются задержанными по отношению к сигналу CLOCK на время прохождения через буферы.

Рекомендуемый способ размножения сигналов приведен на рис. 8.86(b). Все тактовые сигналы проходят через идентичные буферы с примерно равными задержками. В идеальном случае все буферы должны быть элементами, находящимися в одной и той же интегральной схеме; тогда все они обладают близкими временными характеристиками, а также работают при одной и той же температуре и одном и том же напряжении питания. Некоторые производители выпускают специальные буферы, предназначенные как раз для таких приложений, и указывают разброс задержек отдельных буферов в одном корпусе ИС, который, в худшем случае, может составлять всего несколько десятых долей наносекунды.

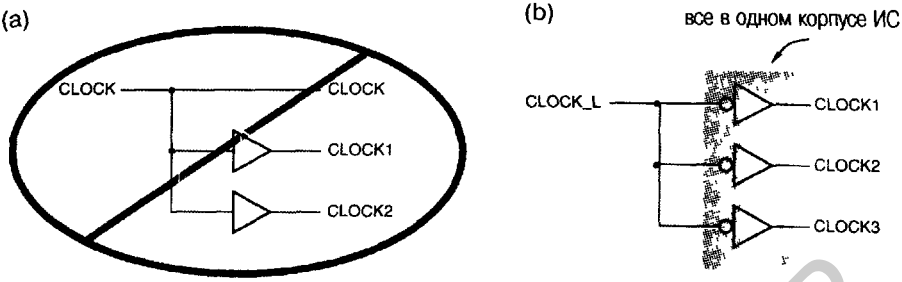


Рис. 8.86. Буферизация тактового сигнала: (a) внесение дополнительного разброса задержек; (b) находящийся под контролем разброс задержек

Однако способ буферизации, показанный на рис. 8.86, также может вносить дополнительный разброс задержек тактового сигнала, если нагрузка для одного из сигналов оказывается значительно большей, чем для других: переходы в тактовом сигнале с большей нагрузкой будут происходить позднее из-за увеличения задержки переключения выходных транзисторов, а также из-за того, что становятся бóльшими время нарастания и время спада сигнала. Поэтому внимательный разработчик старается сбалансировать нагрузку различных тактовых сигналов, принимая во внимание как нагрузку по постоянному току (коэффициент разветвления по выходу), так и нагрузку по переменному току (емкость проводников и входные емкости).

При автоматизированной разводке соединений на печатных платах и внутри специализированных ИС с помощью соответствующих программных средств может возникнуть другая неприятная ситуация. На рис. 8.87 показана печатная плата или специализированная ИС с большим числом триггеров и более сложных элементов, которые переключаются общим тактовым сигналом CLOCK. Программные средства автоматизированной системы проектирования развели сигнал CLOCK таким образом, что он вьется серпантином между тактируемыми устройствами. Для других же сигналов – в каждом случае от выхода к небольшому числу входов – проложены более короткие пути. Еще хуже обстоит дело, когда внутри специализированной ИС распространение сигнала «по проводам» происходит медленнее, чем посредством соединений, выполненных по другой технологии (поликристаллические кремниевые соединения в КМОП-структуре). В результате фронт тактового сигнала CLOCK может поступать на триггер FF2 чуть позднее момента изменения данных на линии Q1.

Один из способов свести к минимуму проблему такого рода состоит в том, чтобы организовать распределение тактового сигнала CLOCK по древовидной структуре, как показано на рис. 8.88, с помощью самых быстрых соединений. Обычно такое «дерево для тактового сигнала» создается вручную или путем применения специализированных систем CAD. Но в большом проекте может случиться так, что нельзя гарантировать повсеместно поступление фронта тактового сигнала до того, как произойдет самое раннее изменение данных. Для обнаружения этого, как правило, применяются программы рисующие временные диаграммы; решением проблемы в общем случае может оказаться лишь включение дополнительной задержки (например, вставление пары инверторов) на том пути тактового сигнала, по которому он проходит слишком быстро.

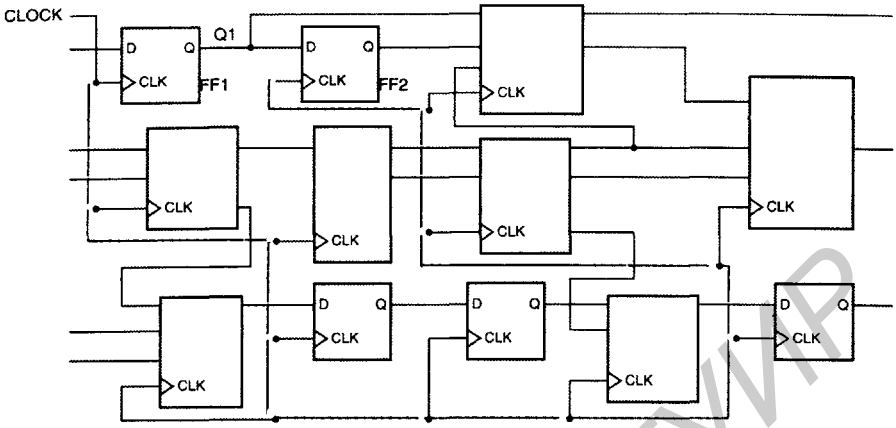


Рис. 8.87. Разводка тактового сигнала, которая может вызвать дополнительный разброс задержек на сложных печатных платах и внутри специализированных ИС

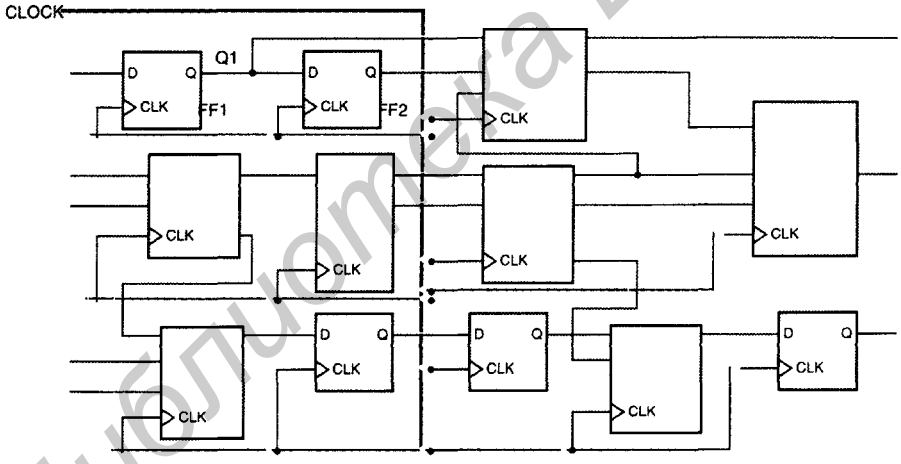


Рис. 8.88. Разводка тактового сигнала, минимизирующая разброс задержек

Хотя методология синхронного проектирования позволяет упростить алгоритм работы большой системы, мы видим все же, что в случае, когда в качестве запоминающих элементов используются переключающиеся по фронту триггеры, главной проблемой может оказаться разброс задержек тактового сигнала. Чтобы преодолеть это затруднение, во многих высокоскоростных системах и в СБИС применяется *проектирование по принципу двухфазных защелок* (*two-phase latch design*), с которым можно ознакомиться по литературе. При таком проектировании каждый переключающийся по фронту D-триггер разбивается на две составляющие его защелки, которыми управляют тактовые сигналы с неперекрывающимися рабочими фазами. Зазор между рабочими фазами поглощает разброс задержек тактового сигнала.

КАК ИЗБЕЖАТЬ ПОСЛЕДСТВИЙ РАЗБРОСА ЗАДЕРЖЕК

Различие в длине проводников и разная нагрузка являются очевидными причинами разброса задержек тактового сигнала. Но существует много других, менее явных источников расхождения. В частности, разброс задержек тактового сигнала может быть вызван *перекрестными помехами*, то есть наводками с одной сигнальной линии на другую. Перекрестные помехи неизбежны, если проводники проходят на печатной плате или внутри микросхемы параллельно на малом расстоянии друг от друга; при этом наводки возникают в моменты переходов сигналов с одного уровня на другой. Сигнал, соседний с тактовым, может изменяться в том же направлении или в противоположном; в зависимости от этого переход в тактовом сигнале может ускоряться или замедляться, в результате чего будет казаться, что переход происходит раньше или позже.

В большом устройстве на печатной плате или внутри специализированной ИС, как правило, нереально отследить и устранить все возможные источники разброса задержек тактового сигнала. Поэтому большинство производителей специализированных ИС требуют от проектировщиков, чтобы те предусмотрели дополнительный запас по времени установления и времени удержания, эквивалентный задержке нескольких вентилей, в дополнение к тому минимуму, который следует из временных диаграмм, полученных путем моделирования. Только в этом случае все неизвестные факторы оказываются преодоленными.

8.8.2. Стробирование тактового сигнала

У большинства последовательностных ИС средней степени интеграции, упомянутых в этой главе, есть синхронный вход разрешения на выполнение данной ИС ее функций. Другими словами, сигналы на входах разрешения у таких ИС фиксируются в тот же момент, что и данные, на фронте тактового сигнала. Первым из рассмотренных нами примеров был регистр 74x377 с синхронным входом разрешения загрузки; другие подобные ИС – это счетчик 74x163 и регистр сдвига 74x194 с синхронными входами разрешения загрузки, счета и сдвига. Тем не менее, у многих ИС средней степени интеграции, а также у макроэлементов для ИС типа FPGA и у ячеек в специализированных ИС нет синхронных входов разрешения соответствующих действий; например, 8-разрядный регистр 74x374 имеет выходы с тремя состояниями, но у него нет входа разрешения загрузки.

Спрашивается: что может сделать разработчик, если требуется, чтобы у 8-разрядного регистра были *и* вход разрешения загрузки, *и* выходы с тремя состояниями? Одно из решений этой задачи состоит в использовании ИС 74x377 с входом разрешения загрузки и включении вслед за этой ИС буфера 74x241 с выходами с тремя состояниями. Однако при этом возрастут стоимость и задержка. Другой вариант заключается в применении большего по размерам и более дорогого изделия, а именно – ИС 74x823, которая обладает обоими требуемыми свойствами и у которой, кроме того, есть асинхронный вход сброса CLR_L. Но если разработчик не знает ничего лучше, кроме ИС '374, то более рискованным

решением является запирание сигнала на тактовом входе этой ИС на интервале времени, в течение которого не предполагается производить загрузку. Такое действие называют *стробированием тактового сигнала* (*gating the clock*).

На рис. 8.89 демонстрируется очевидный, но ошибочный способ стробирования тактового сигнала. Сигнал CLKEN подается для того, чтобы разрешить прохождение тактового сигнала CLOCK через вентиль И и выработать стробированный тактовый сигнал GCLK. В случае применения такого способа возникают две проблемы:

1. Если сигнал CLKEN является выходным сигналом конечного автомата или другим сигналом, который вырабатывается каким-либо регистром, то изменение сигнала CLKEN происходит чуть *позднее* того момента, когда сигнал CLOCK переходит на высокий уровень. Как показано на рис. 8.89(b), при этом в сигнале GCLK возникают паразитные импульсы, которые будут приводить к ошибочным переключениям регистров, на входы которых поступает этот сигнал.
2. Но даже если сигнал CLKEN каким-то образом вырабатывается раньше нарастающего фронта сигнала CLOCK (например, на выходе регистра, переключающегося по *спадающему* фронту сигнала CLOCK, что является особенно нежелательным), задержка в вентиле И приведет к увеличению разброса задержек тактового сигнала в системе, в результате чего могут возникнуть определенные проблемы в других местах.

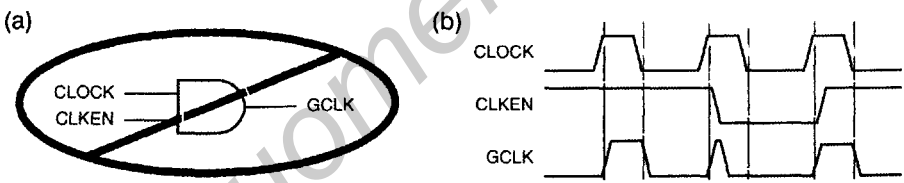


Рис. 8.89. Как не надо стробировать тактовый сигнал: (a) схема, решающая задачу «в лоб»; (b) временные диаграммы

Способ стробирования тактового сигнала, обеспечивающий минимальный разброс задержек, представлен на рис. 8.90. В приведенной схеме нестробированный тактовый сигнал и несколько стробированных тактовых сигналов вырабатываются от одного и того же главного тактового сигнала с низким активным уровнем. Чтобы минимизировать возможные различия в задержках, следует использовать вентили, находящиеся в одном и том же корпусе ИС. Сигнал CLKEN может меняться произвольно, пока сигнал CLOCK_L остается на низком уровне, а сигнал CLOCK имеет высокий уровень. Вот и прекрасно: сигнал CLKEN вырабатывается, как правило, конечным автоматом, у которого изменение сигналов на его выходах происходит строго после того, как сигнал CLOCK переходит на высокий уровень.

Подход, иллюстрируемый рис. 8.90, приемлем только в том случае, когда возникающий при этом разброс задержек является допустимым. Кроме того, заметьте, что сигнал CLKEN должен оставаться неизменным в течение всего интервала времени, на котором сигнал CLOCK_L имеет высокий уровень (а сигнал

CLOCK – низкий). Таким образом, запас по времени будет зависеть от коэффициента заполнения тактового сигнала особенно тогда, когда сигнал CLKEN оказывается значительно сдвинутым по отношению к переключающему фронту тактового сигнала из-за большой задержки в комбинационной логике (t_{comb}). В настоящей синхронной системе входной сигнал разрешения выполнения той или иной функции может изменяться почти в любой момент времени в пределах всего периода тактового сигнала вплоть до границы времени установления перед переключающим фронтом тактового сигнала; такого рода решение применительно к разрешению загрузки ИС 74x377 приведено на рис. 8.13.

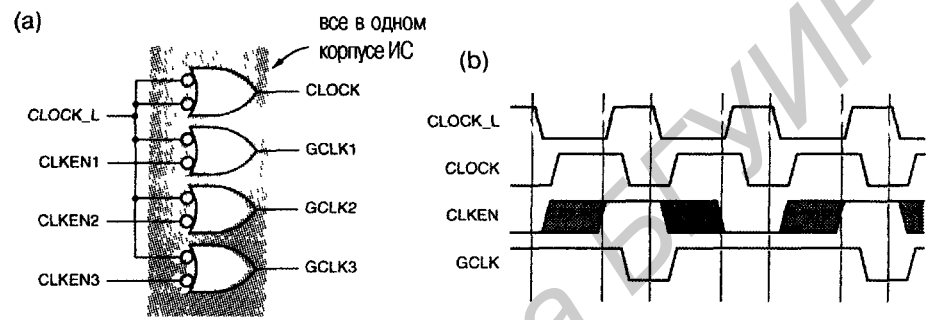


Рис.8.90. Приемлемый способ стробирования тактового сигнала: (a) схема; (b) временные диаграммы

8.8.3. Асинхронные входы

Хотя теоретически можно построить полностью синхронный компьютер, вы вряд ли сможете им воспользоваться, если только не окажетесь способным нажимать на клавиши синхронно с 500-мегагерцным тактовым сигналом. Цифровой системе любого типа всегда приходится иметь дело с *асинхронными входными сигналами* (*asynchronous input signals*), которые не привязаны к тактовому сигналу системы.

Асинхронные входные сигналы часто бывают запросами на обслуживание (например, прерывания в компьютере) или признаками возникновения тех или иных условий (например, сигнал о том, что какой-то ресурс стал доступен). Обычно такие сигналы изменяются медленно по сравнению с тактовой частотой системы, и нет необходимости распознавать их на определенном такте. Если переход в таком сигнале будет пропущен на данном такте, его всегда можно обнаружить на следующем. Скорость переключений асинхронных сигналов может простираться от одного изменения в секунду (когда за клавиатурой сидит медлительный человек) до 100 МГц и более (запросы на доступ к совместно используемой памяти в 500-мегагерцной многопроцессорной системе).

Если не принимать во внимание проблему метастабильности, то нетрудно построить *синхронизирующее устройство* (*synchronizer*), то есть схему, в которой будет браться выборка асинхронного входного сигнала и вырабатываться выходной сигнал, удовлетворяющий требованиям синхронной системы по времени установления и времени удержания. Как видно из рис. 8.91, D-триггер берет выборки асинхронного входного сигнала на каждом такте системного тактового

сигнала и вырабатывает синхронный выходной сигнал, значение которого удерживается в течение следующего периода тактового сигнала.

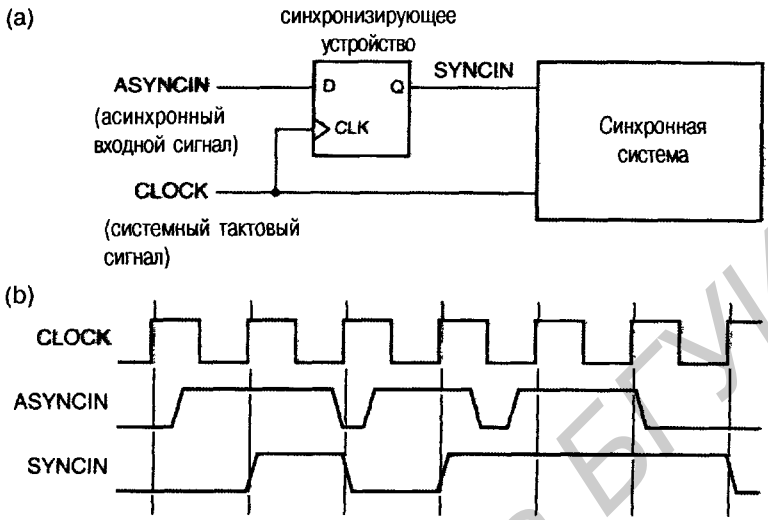


Рис. 8.91. Простое синхронизирующее устройство на одном D-триггере: (a) схема; (b) временные диаграммы

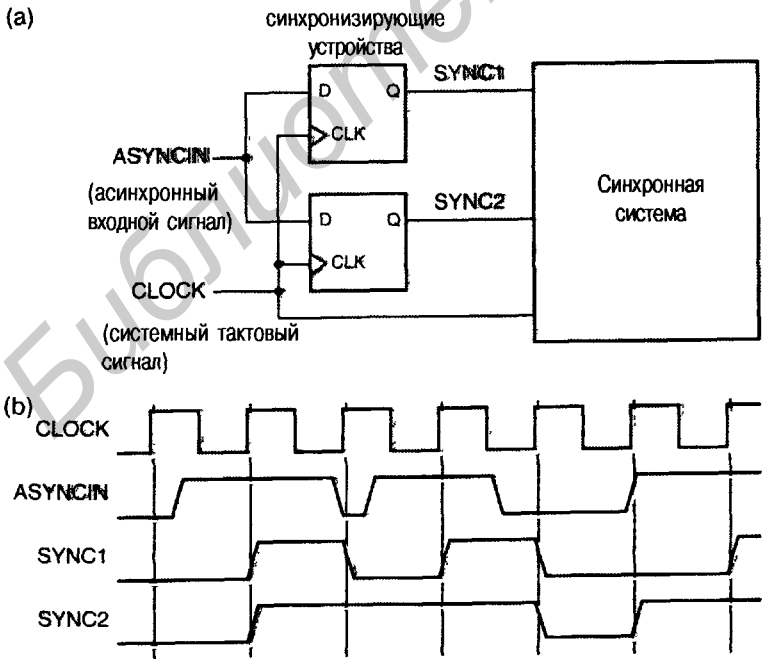


Рис. 8.92. Два синхронизирующих устройства для одного и того же асинхронного входного сигнала: (a) схема; (b) возможные временные диаграммы

Для асинхронных входных сигналов существенно, чтобы их привязка к тактовому сигналу производилась в системе только *в одном месте*; на рис. 8.92 показано, что может произойти в противном случае. Из-за физических задержек в схеме два триггера не видят тактовый сигнал и входные сигналы точно в один и тот же момент времени. Поэтому в случае, когда асинхронный сигнал изменяется вблизи фронта тактового сигнала, существует небольшое временно́е окно, в пределах которого один триггер может воспринять и зафиксировать входной сигнал как 1, а другой – как 0. Противоположные значения сигналов на выходах этих триггеров могут стать причиной неправильной работы системы, одна часть которой будет вести себя так, как если бы входной сигнал был равен 1, а отклик другой части будет таким, как если бы этот сигнал был равен 0.

Если сигнал проходит через комбинационную логику, то наличие двух синхронизирующих устройств может не проявиться (рис. 8.93). Из-за различных задержек прохождения сигнала по разным путям в комбинационной логике вероятность появления на выходах триггеров несовместимых результатов только возрастает. Это распространенный случай, особенно тогда, когда асинхронные сигналы являются входными сигналами конечного автомата, поскольку две или более переменные состояния, вырабатываемые логикой возбуждения, могут зависеть от значения асинхронного входного сигнала. Правильный способ подачи асинхронного сигнала на вход конечного автомата показан на рис. 8.94. Логика возбуждения видит только один синхронизированный входной сигнал SYNCIN.

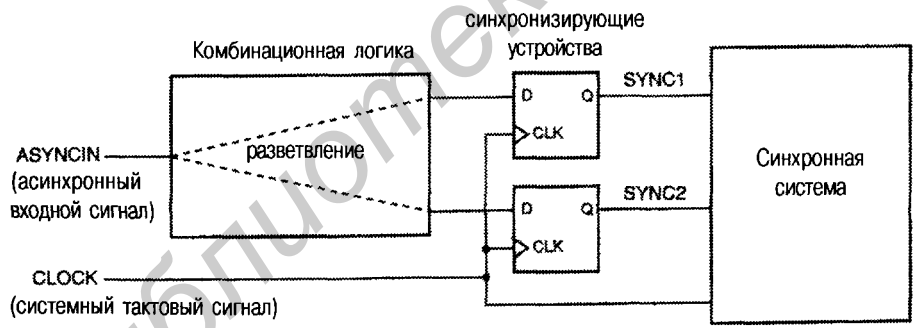


Рис. 8.93. Асинхронный входной сигнал, поступающий на входы двух синхронизирующих устройств после прохождения через комбинационную логику

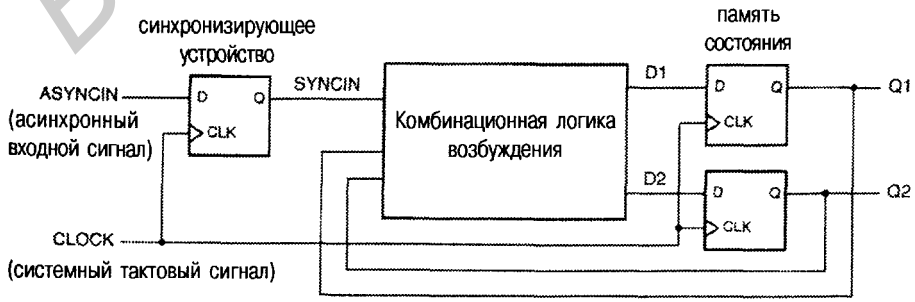


Рис. 8.94. Асинхронный сигнал на входе конечного автомата, пропущенный через единственное синхронизирующее устройство

СТОИТ ЛИ ВОЛНОВАТЬСЯ?

Вы, возможно, догадываетесь, что синхронизирующие устройства, изображенные на рис. 8.91 и 8.94, могут сработать неправильно. Это может произойти из-за нарушения требований, предъявляемых временем установления и временем удержания синхронизирующих триггеров, поскольку асинхронный входной сигнал может измениться в любой момент времени. «Стоит ли волноваться?», скажете вы. «Если сигнал на входе D изменяется вблизи фронта тактового сигнала, то триггер либо увидит это изменение на данном такте, либо пропустит его сейчас и обнаружит на следующем такте. В любом случае это меня вполне устраивает!» Проблема все же есть, и она связана с существованием третьей возможности, рассматриваемой в параграфе 8.9.

8.9. Сбой в работе синхронизирующего устройства и метастабильность

В параграфе 7.1 было показано, что в случае, когда требования в отношении времени установления и времени удержания триггера не удовлетворены, триггер может войти в третье, *метастабильное* состояние посередине между 0 и 1. Хуже всего то, что время пребывания в этом состоянии, то есть время до того момента, когда триггер «свалится» в одно из его законных состояний – в состояние 0 или в состояние 1, – теоретически неограниченно. Некоторые из вентилей и триггеров, на входы которых поступает метастабильный сигнал, могут интерпретировать его как 0, тогда как другие вентили и триггеры будут воспринимать его как 1, в результате чего возникнет того или иного рода несовместимость типа той, какая была указана на рис. 8.92. Впрочем, вентили и другие триггеры с метастабильным сигналом на входе, сами могут вырабатывать метастабильные сигналы на своих выходах (ведь, в конце концов, эти схемы оказываются в линейной части их передаточной характеристики). К счастью, вероятность того, что сигнал на выходе триггера и дальше останется метастабильным, уменьшается со временем экспоненциально, хотя никогда и не становится равной нулю.

8.9.1. Сбой в работе синхронизирующего устройства

Говорят, что в работе *синхронизирующего устройства* произошел *сбой* (*synchronizer failure*), когда в системе используется выходной сигнал этого устройства, несмотря на то, что он остается метастабильным. Система может обезопасить себя от сбоев в синхронизирующем устройстве, если будет «достаточно долго» ждать, прежде чем воспользуется выходным сигналом этого устройства. Но что значит «достаточно долго»? Для этого необходимо, по крайней мере, чтобы среднее время между сбоями в синхронизирующем устройстве было на несколько порядков больше, чем планируемое разработчиком время использования системой выходного сигнала этого устройства.

Метастабильность – это нечто большее, чем академическая проблема. Многим конструкторам довелось стать создателями высокоскоростных цифровых си-

стем, которые страдали тем, что время от времени происходили сбои в их синхронизирующих устройствах (и которые, тем не менее, были доведены до серийного производства). Говорят, что связанные с метастабильностью проблемы первоначально были у целого ряда популярных ИС, в частности, у таких микросхем, как системный времязадающий контроллер AMD 9513, контроллер прерываний AMD 9519, последовательный интерфейс ввода/вывода Z-80 фирмы Zilog, однокристальный микрокомпьютер 8048 фирмы Intel и RISC-процессор AMD 29000. Вы, наверное, задаетесь вопросом: «И что, этих разработчиков еще не уволили?».

Существует два способа избавиться от пребывания триггера в метастабильном состоянии:

1. Принудительно переводить его в одно из его законных состояний с помощью сигналов, удовлетворяющих объявленным требованиям в отношении минимальной длительности импульса, времени установления и т.д.
2. Подождать «достаточно долго», пока триггер сам собой не выйдет из состояния метастабильности.

Неопытные проектировщики часто пытаются обойти метастабильность другим путем и, как правило, терпят неудачу. Одна из таких попыток представлена на рис. 8.95: коль скоро метастабильность является «аналоговой» проблемой, ее решение также должно быть «аналоговым», – так думает разработчик. Действительно, вентили с триггерами Шмитта на входах и с конденсаторами могут быть использованы в обычных условиях для очистки сигналов от шума. Однако вместо исключения метастабильности, такая схема только усилит этот эффект: построенная из вполне «достойных» элементов, эта схема навсегда войдет в режим колебаний, как только одновременно будут переведены на неактивный уровень сигналы S_L и R_L. (Автор должен признаться, что больше 20 лет назад попытался это сделать!) В задачах 8.97 и 8.94 приведены примеры отважных, но неудачных попыток исключить метастабильность. Эти примеры позволят вам почувствовать, что проблемы, возникающие в связи с синхронизирующими устройствами, могут быть очень тонкими, так что необходимо быть бдительным. Единственный способ сделать синхронизирующее устройство надежным состоит в том, чтобы ждать достаточно долго, пока выходной сигнал не перестанет быть метастабильным. На вопрос: «Как долго надо ждать, чтобы этого было достаточно?» мы ответим в этом параграфе позже.

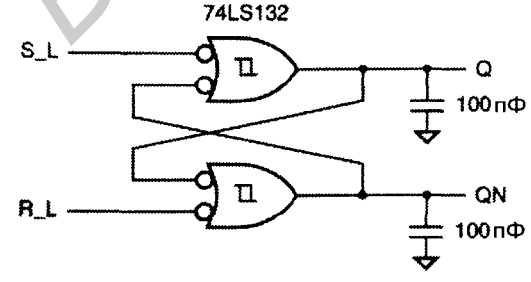


Рис. 8.95. Неудачная попытка построить $\overline{SR}$-триггер, защищенный от метастабильности

8.9.2. Время выхода из метастабильности

Если требования D-триггера по времени установления и времени удержания удовлетворены, то триггер устанавливается в новое состояние в пределах интервала времени t_{pd}, после того как прошел фронт тактового сигнала. Если эти требования нарушены, то выход триггера может быть метастабильным сколь угодно долго. Проектируя некоторую систему, мы пользуемся параметром t_r, носящим название *времени выхода из метастабильности* (*metastability resolution time*), для обозначения максимального времени, в течение которого выходной сигнал может оставаться метастабильным без ущерба для работы синхронизирующего устройства (и системы).

Рассмотрим, например, конечный автомат, изображенный на рис. 8.94. В этом случае мы располагаем следующим временем выхода из метастабильности:

$$t_r = t_{clk} - t_{comb} - t_{setup},$$

где t_{clk} – период тактового сигнала, t_{comb} – задержка распространения сигнала по комбинационной логике и t_{setup} – время установления триггеров, используемых в памяти состояния.

8.9.3. Разработка надежного синхронизирующего устройства

Самое надежное синхронизирующее устройство – это такое устройство, которое успевает за отведенное время выйти из метастабильности. Но при проектировании цифровой системы мы редко можем позволить себе роскошь *понизить* тактовую частоту ради надежности. Обычно, напротив, от нас требуют *повысить* тактовую частоту, чтобы система обладала лучшими характеристиками. Поэтому чаще всего нам нужно, чтобы синхронизирующее устройство работало надежно при очень малых значениях периода тактового сигнала. Мы представим несколько таких схем и покажем, как можно оценить их надежность.

Как сказано выше, у конечного автомата с асинхронным входом, структура которого показана на рис. 8.94, время выхода из метастабильности равно $t_r = t_{clk} - t_{comb} - t_{setup}$. Чтобы сделать t_r возможно бóльшим при заданном периоде тактового сигнала, нам следует минимизировать t_{comb} и t_{setup}. Значение t_{setup} определяется типом триггеров, используемых в памяти состояния; в общем случае, у более быстрого триггера время установления меньше. Минимальное значение t_{comb} равно 0 и достигается в синхронизирующем устройстве, приведенном на рис. 8.96; сейчас мы объясним, как работает эта схема.

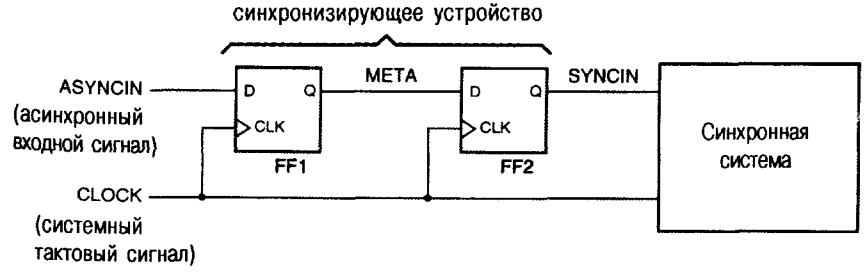


Рис. 8.96 Рекомендуемая схема синхронизирующего устройства

Сигналы на входе триггера FF1 асинхронны по отношению к тактовому сигналу и могут поступать с нарушением требований, предъявляемых временем установления и временем удержания. Если это происходит, то выходной сигнал META становится метастабильным и остается в этом состоянии произвольно долго. Предположим, однако, что максимальное время метастабильности, после того как прошел фронт тактового сигнала, равно t_r. (В следующем разделе мы покажем, как найти вероятность того, что наше предположение верно.) Коль скоро период тактового сигнала больше, чем t_r плюс время установления триггера FF2, сигнал SYNCIN становится синхронной копией асинхронного входного сигнала на следующем такте тактового сигнала, никогда не оказываясь метастабильным. Далее сигнал SYNCIN можно использовать повсеместно в системе по мере необходимости.

8.9.4. Анализ времени пребывания в состоянии метастабильности

На рис. 8.97 приведены временные параметры, учитываемые при анализе времени пребывания в метастабильном состоянии. Обозначим указываемые производителем время установления и время удержания по обе стороны фронта в тактовом сигнале через t_s и t_h; эти два интервала времени образуют *окно принятия решения* (*decision window*): на этом отрезке триггер берет выборку сигнала на входе данных и решает, нужно ему изменять выходной сигнал или нет. Если сигнал на входе D изменяется за пределами этого окна, как показано на рис. (a), то производитель гарантирует переключение триггера и его переход в одно из его законных логических состояний не позднее времени t_{pd}. Если входной сигнал D изменяется в пределах окна принятия решения, как показано на рис. (b), то метастабильность может возникнуть и просуществовать до конца интервала времени t_r.

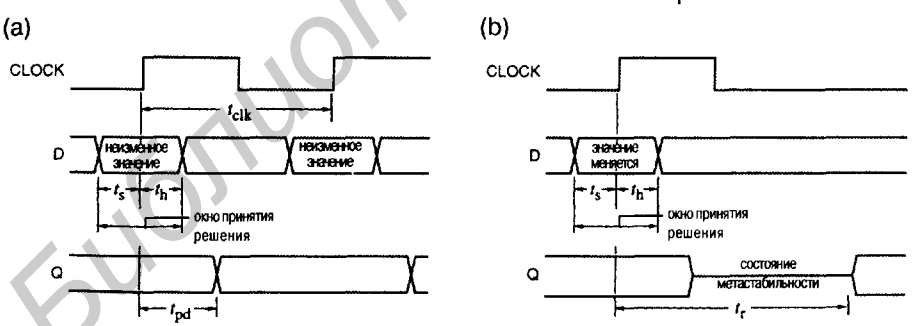


Рис. 8.97. Временные параметры, учитываемые при анализе метастабильности: (a) нормальная работа триггера; (b) метастабильное поведение

НЕБОЛЬШОЕ УТОЧНЕНИЕ

Рассматривая синхронизирующее устройство, изображенное на рис. 8.96, мы не допускали возможности возникновения метастабильности на выходе триггера FF2 даже на короткое время, поскольку предполагалось, что система спроектирована с нулевым запасом по времени. Но если система допускает небольшое увеличение задержки прохождения сигнала через триггер FF2, то значение MTBF, вычисляемое в разделе 8.9.4, будет немного лучше.

Теоретическое исследование показывает (а практический опыт подтверждает), что при изменении асинхронного входного сигнала в пределах окна принятия решения длительность пребывания выхода в метастабильном состоянии описывается экспоненциальной зависимостью:

$$\mathrm{MTBF}(t_{\mathrm{r}}) = \frac{\exp(t_{\mathrm{r}}/\tau)}{T_0 \cdot f \cdot a}.$$

Здесь MTBF – *среднее время между отказами* (*Mean Time Between Failures*) синхронизирующего устройства, если считать, что отказ происходит в том случае, когда метастабильность выходит за пределы отрезка времени длительностью t_{r} после фронта тактового сигнала; $t_{\mathrm{r}} \geq t_{\mathrm{pd}}$. Значение MTBF зависит от частоты сигнала f на тактовом входе триггера; a – число изменений асинхронного входного сигнала, поступающего на вход данных триггера, в секунду; T_0 и τ – константы, зависящие от электрических характеристик триггера. В типичном случае – для ИС 74LS74 – $T_0 \approx 0.4$ с, $\tau \approx 1.5$ нс.

Предположим теперь, что создается микропроцессорная система с частотой тактового сигнала 10 МГц и со схемой, преобразующей асинхронный входной сигнал в синхронный, изображенной на рис. 8.96. Если сигнал ASYNCIN изменяется в пределах окна принятия решения триггера FF1, то выход META может оставаться в состоянии метастабильности в течение интервала времени t_{r}. Если сигнал META все еще остается метастабильным к началу окна принятия решения триггера FF2, то происходит сбой в работе синхронизирующего устройства, так как выход триггера FF2 может оказаться метастабильным; в этом случае поведение системы непредсказуемо.

Пусть D-триггерами в схеме на рис. 8.96 являются триггеры из ИС 74LS74. Время установления t_{s} для такого триггера равно 20 нс, тогда как период тактового сигнала в нашем примере микропроцессорной системы равен 100 нс; таким образом, допустимая продолжительность пребывания в метастабильном состоянии составляет 80 нс. Если асинхронный входной сигнал изменяется 100000 раз в секунду, то среднее время между сбоями синхронизирующего устройства равно

$$\mathrm{MTBF}(80\ \text{нс}) = \frac{\exp(80/1.5)}{0.4 \cdot 10^{7} \cdot 10^{5}} = 3.6 \cdot 10^{11}\ \text{с}.$$

Это совсем неплохо: 100 столетий между сбоями! Правда, если бы нам удалось продать 10 000 таких систем, то одна из них давала бы сбой раз в году. Но рассмотрим все же более серьезную проблему.

Предположим, что, модернизируя нашу систему, мы используем кристалл процессора с тактовой частотой 16 МГц. Возможно, нам понадобится заменить некоторые компоненты, чтобы система работала с большей скоростью, но триггеры в ИС 74LS74 вполне успешно переключаются с частотой 16 МГц. Или их тоже надо заменить? Коль скоро период тактового сигнала теперь равен 62.5 нс, новое значение MTBF для нашего синхронизирующего устройства равно

$$\mathrm{MTBF}(42.5\ \text{нс}) = \frac{\exp(42.5/1.5)}{0.4 \cdot 1.6 \cdot 10^{7} \cdot 10^{5}} = 3.1\ \text{с!}$$

ЧТО ПОНИМАТЬ ПОД *a* И *f*?

Выход триггера может перейти в метастабильное состояние *только* в том случае, если сигнал на входе D изменяется в пределах окна принятия решения. Но в формулу для MTBF явным образом не входит число таких попаданий. Вместо этого в ней фигурирует общее число *a* изменений асинхронного входного сигнала в секунду и предполагается, что эти изменения распределены равномерно по периоду тактового сигнала. Поэтому доля изменений входного сигнала, действительно происходящих в пределах окна принятия решения, окажется «учтенной» в параметре *f*: с увеличением частоты тактового сигнала *f* все большее число изменений входного сигнала попадает в окно принятия решения.

Если в создаваемой системе изменения входного сигнала не распределены равномерно по периоду тактового сигнала, а группируются в окрестности окна принятия решения (это может происходить в том случае, когда изменения в синхронизуемом входном сигнале как-то привязаны к системному тактовому сигналу с фиксированным, но неизвестным сдвигом по фазе), то полезное правило оценки заключается в том, чтобы в качестве частоты брать величину, обратную длительности окна принятия решения (на основании сообщаемых производителем значений времени установления и времени удержания), умноженную для надежности на коэффициент порядка 10.

Единственное достоинство рассматриваемого синхронизирующего устройства состоит в том, что из-за его отвратительного поведения на частоте 16 МГц проблема, вероятнее всего, обнаружится на стадии лабораторных испытаний, а не после того, как изделие поступит в продажу! Не дай бог, чтобы значение MTBF было порядка года.

8.9.5. Более совершенные синхронизирующие устройства

Имеется несколько возможностей построения более надежных синхронизирующих устройств, чем в случае использования для этих целей ИС 74LS74, когда характеристики синхронизирующего устройства оказываются совсем плохими уже при средних значениях тактовой частоты. Простейшее решение – это применение триггеров, изготовленных по технологии, обеспечивающей большее быстродействие; в большинстве случаев это позволят удовлетворить требованиям, предъявляемым к разрабатываемой системе. В настоящее время имеются триггеры со значительно бóльшим быстродействием, как в отдельных микросхемах, так и внутри ПЛУ, ИС типа FPGA и в специализированных ИС.

В табл. 8.35 перечислены параметры, относящиеся к метастабильности, для нескольких распространенных логических семейств; эти сведения взяты, главным образом, из технических характеристик, публикуемых производителями. Числовые значения параметров в очень сильной степени зависят от схемных решений и технологии изготовления ИС. В отличие от гарантированных логических уровней сигналов и их временных параметров, числовые значения величин, характеризующих метастабильность, могут изменяться в очень широких пределах

для ИС одного и того же типа в зависимости от производителя, и поэтому пользоваться ими следует с большой осторожностью. Например, ИС 74F74 одного производителя могут давать приемлемые характеристики метастабильности, тогда как в случае другого производителя это условие оказывается невыполненным.

Табл. 8.35. Параметры, характеризующие метастабильность, для ряда распространенных ИС

Источник информации	*Тип ИС*	τ (*нс*)	T_0(*с*)	t_r (*нс*)
Chaney (1983)	74LS74	1.50	$4.0 \cdot 10^{1}$	77.71
Chaney (1983)	74S74	1.70	$1.0 \ 10^{-6}$	66.14
Chaney (1983)	74S174	1.20	$5.0 \cdot 10^{-6}$	48 62
Chaney (1983)	74S374	0.91	$4.0 \ 10^{-4}$	40.86
Chaney (1983)	74F74	0.40	$2.0 \cdot 10^{-4}$	17.68
TI (1997)	74LSxx	1.35	$4.8 \cdot 10^{-3}$	63.97
TI (1997)	74Sxx	2.80	$1.3 \cdot 10^{-9}$	90.33
TI (1997)	74ALSxx	1.00	$8.7 \cdot 10^{-6}$	41.07
TI (1997)	74ASxx	0.25	$1.4 \cdot 10^{3}$	14.99
TI (1997)	74Fxx	0.11	$1.9 \ 10^{8}$	7.90
TI (1997)	74HCxx	1.82	$1.5 \cdot 10^{-6}$	71.55
Cypress (1997)	PALC16R8-25	0.52	$9.5 \cdot 10^{-12}$	14.22*
Cypress (1997)	PALC22V10B-20	0.26	$5.6 \ 10^{-11}$	7.57*
Cypress (1997)	PALCE22V10-7	0.19	$1.3 \cdot 10^{-13}$	4.38*
Xilinx (1997)	CPLD-серия 7300	0.29	$1.0 \ 10^{-15}$	5.27*
Xilinx (1997)	CPLD-серия 9500	0.17	$9.6 \cdot 10^{-18}$	2.30*

Заметьте, что у различных авторов и производителей параметры метастабильности могут быть определены по-разному. Например, Чейни (Chaney) и Texas Instruments (TI) отсчитывают время выхода из метастабильности от переключающего фронта тактового сигнала, то есть так же, как это делали мы в предыдущем разделе. В противоположность этому, Cypress и Xilinx под t_r понимают время, которое должно быть *добавлено* к нормальному времени задержки t_{pd} от тактового входа до выхода.

Критерий качества, указанный в последнем столбце таблицы, в какой-то степени произволен: это значение времени выхода из метастабильности t_r, необходимое для того, чтобы величина MTBF составляла 1000 лет при работе синхронизирующего устройства с частотой 25 МГц и при 100000 изменений асинхронного входного сигнала в секунду. Для схем фирм Cypress и Xilinx значения t_r помечены звездочкой в знак того, что имеется в виду их собственное определение этой величины согласно сказанному выше.

Как вы можете видеть, ИС 74LS74 – одна из худших микросхем, перечисленных в таблице. Если в 16-магагерцной микропроцессорной системе, рассматривавшейся в качестве примера в предыдущем разделе, заменить FF1 на триггер из ИС 74ALS74, то получим:

$$\mathrm{MTBF}(42.5\text{нс}) = \frac{\exp(42.5/1.00)}{8.7\cdot10^{-6}\cdot1.6\cdot10^{7}\cdot10^{5}} = 2.06\cdot10^{11}\,\text{с}.$$

Если вам годится синхронизирующее устройство со значением MTBF, равным 65 столетиям, для каждого из проданных изделий, то на этом можно остановиться. Но если на триггер из ИС 74ALS74 заменить также и FF2, то значение MTBF станет еще лучше, так как у схем 'ALS74 время установления меньше чем у ИС 'LS74, и составляет всего 10 нс. В результате использования триггера из ИС 'ALS74 на месте FF2 значение MTBF станет примерно в 20000 раз бóльшим:

$$\mathrm{MTBF}(52.5\text{нс}) = \frac{\exp(52.5/1.00)}{8.7\cdot10^{-6}\cdot1.6\cdot10^{7}\cdot10^{5}} = 4.54\cdot10^{15}\,\text{с}.$$

Даже если мы продадим миллион изделий с таким синхронизирующим устройством, у нас (или у наших преемников) сбой будет происходить в синхронизирующем устройстве раз в 144 года. Сегодня это обеспечивает нам надежную работу!

На самом деле указанный запас надежности не так велик, как это может показаться. (Насколько большим кажется *вам* интервал длиной 144 года?) Большинство числовых значений в табл. 8.35 являются *средними*, и редко кто из производителей ИС указывает эти параметры в технических данных, не говоря уже о том, чтобы гарантировать их величину. Кроме того, вычисляемое значение MTBF крайне чувствительно к значению τ, которое, в свою очередь, может зависеть от температуры, напряжения и фазы луны. Поэтому данный триггер в реальной системе может работать много хуже (или много лучше), чем то, что предсказывает наша таблица.

Рассмотрим, например, что произойдет, если мы увеличим тактовую частоту в нашей 16-мегагерцной системе до 20 МГц, то есть всего на 25%. Вы, естественно, можете подумать, что метастабильность ухудшится на 25%, или, быть может, на 250%, если считать с большим запасом. Но вы подсчитайте, и тогда увидите, что при использовании элементов ИС 'ALS74 на месте обоих триггеров FF1 и FF2 значение MTBF упадет до $3.7\cdot10^{9}$с, то есть станет хуже более чем в миллион раз! Новое значение MTBF составляет примерно 429 лет, и это прекрасно, если иметь в виду одну систему; но если вы продадите миллион таких систем, то отказ в одной из них будет происходить раз в четыре часа. В результате, вместо того, чтобы в течение долгого времени оставаться незаменимым работником вашей фирмы, вы становитесь в ней козлом отпущения!

8.9.6. Другие схемы синхронизирующих устройств

Выше мы пообещали рассказать о том, как строить более надежные синхронизирующие устройства. Первый способ уже указан: применяйте триггеры с бóльшим быстродействием, в результате чего уменьшится значение τ в выражении для MTBF. Второй способ очевиден: нужно увеличить допустимое время t_r в выражении для MTBF.

Самое большое значение t_r в схеме на рис. 8.96 при заданной частоте системного тактового сигнала равно t_{clk}, и это достижимо в том случае, когда время установления триггера FF2 равно 0. Но значение t_r можно увеличить на порядок и сделать равным $n \cdot t_{clk}$, воспользовавшись схемой *многотактного синхронизирующего устройства* (*multiple-cycle synchronizer*), приведенной на рис. 8.98. Здесь частота системного тактового сигнала делится на n, полученное таким образом колебание играет роль тактового сигнала для синхронизирующего устройства и допустимое время метастабильности возрастает до $t_r = (n \cdot t_{clk}) - t_{setup}$. Обычно значения $n = 2$ или $n = 3$ обеспечивают подходящую надежность синхронизирующего устройства.

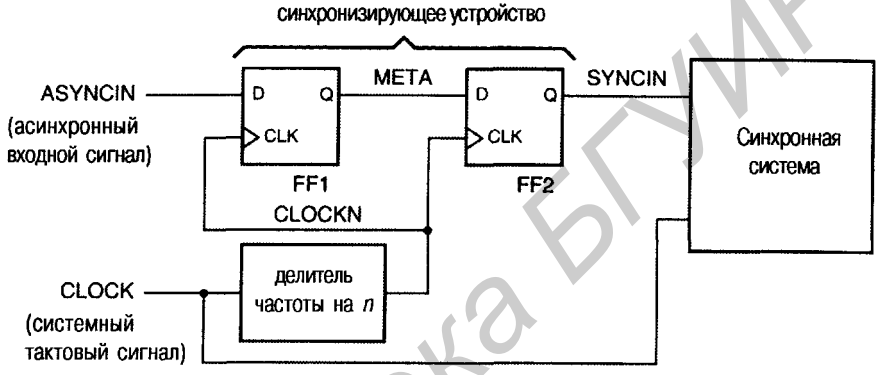


Рис. 8.98. Многотактное синхронизирующее устройство

Обратите внимание на то, что в этой схеме перепады в сигнале CLOCKN будут отставать от перепадов в сигнале CLOCK, поскольку CLOCKN является выходным сигналом счетчика, состоящего из триггеров, переключающихся по фронту сигнала CLOCK. Это означает, что сигнал SYNCIN, в свою очередь, будет задержан или затянут по отношению к другим сигналам в синхронной системе, которые вырабатываются триггерами, переключающимися по фронту сигнала CLOCK непосредственно. Если сигнал SYNCIN проходит в синхронной системе через дополнительную комбинационную логику, прежде чем достигает входов системных триггеров, то требования, предъявляемые временем установления этих триггеров, могут оказаться невыполненными. В этом случае можно воспользоваться схемным решением, представленным на рис. 8.99. Здесь сигнал SYNCIN еще раз тактируется триггером FF3 по сигналу CLOCK, в результате чего вырабатывается сигнал DSYNCIN, временные параметры которого будут такими же, как и у сигналов на выходах других триггеров синхронной системы. Задержка сигнала CLOCKN по отношению к сигналу CLOCK должна быть достаточно малой, чтобы удовлетворять требованию, предъявляемому временем установления триггера FF3.

Чем больше n в n-тактном синхронизирующем устройстве, тем дольше синхронной системе не видно изменение асинхронного входного сигнала. Эта задержка является ценой, которую необходимо уплатить за надежную работу системы. В типичной микропроцессорной системе бóльшая часть асинхронных входных сигналов извещает систему о внешних событиях (прерывания, требования прямого доступа в память и т.д.), так что не требуется распознавать их очень быстро

с точки зрения задержки в синхронизирующем устройстве. Когда обращение к памяти критично по времени, опытные разработчики заставляют подсистему памяти работать от тактового сигнала процессора, если только это возможно. При этом надобность в синхронизирующем устройстве пропадает и система функционирует с наибольшим возможным быстродействием.

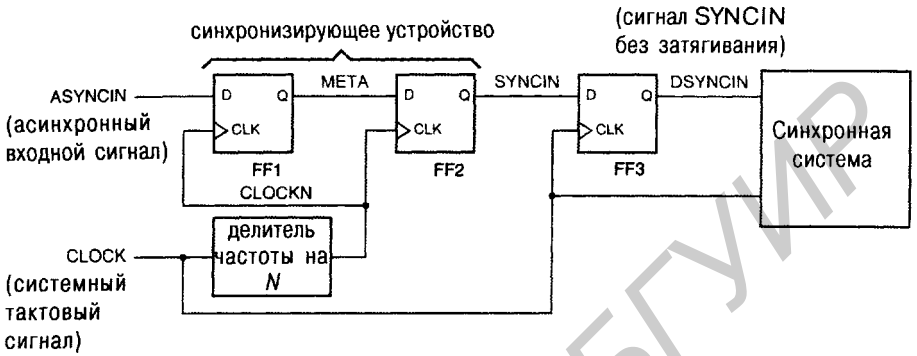


Рис. 8.99. Многотактное синхронизирующее устройство с компенсацией затягивания

На более высоких частотах возможность реализации многотактного синхронизирующего устройства по схеме, приведенной на рис. 8.98, ограничена разбросом задержек тактового сигнала. По этой причине некоторые проектировщики вместо деления частоты системного тактового сигнала на *n* применяют *последовательно включенные синхронизирующие устройства* (*cascaded synchronizers*). При таком подходе используется цепочка из *n* триггеров (регистр сдвига), в которой все триггеры переключаются быстрым системным тактовым сигналом. Соответствующая схема показана на рис. 8.100

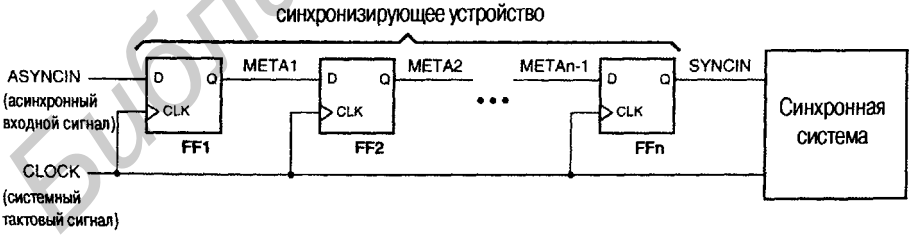


Рис. 8.100. Многокаскадное синхронизирующее устройство

Принцип действия многокаскадного синхронизирующего устройства основан на том, что с некоторой вероятностью выход из состояния метастабильности произойдет уже в первом триггере, а в случае неудачи – с равной вероятностью в каждом следующем из триггеров, включенных последовательно. Таким образом, вероятность отказа синхронизирующего устройства в целом оказывается порядка *n*-й степени вероятности отказа на данной частоте системного тактового сигнала синхронизирующего устройства с одним триггером. И хотя это отча-

сти верно, все же величина MTBF для многокаскадного синхронизирующего устройства меньше, чем для многотактного синхронизирующего устройства с тем же временем задержки ($n \cdot t_{clk}$). В случае многокаскадного устройства время установления триггера t_{setup} необходимо вычесть n раз из времени t_r, тогда как в случае многотактного устройства значение t_{setup} вычитается только один раз.

Для построения синхронизирующего устройства можно воспользоваться внутренними триггерами ПЛУ; при этом оба триггера в схеме на рис. 8 96 находятся в одном ПЛУ. В большинстве приложений это очень удобно, так как исключается необходимость применения внешних триггеров, размещенных в отдельной ИС. Однако, как правило, значение MTBF для синхронизирующего устройства, образованного внутри ПЛУ, хуже, чем при использовании отдельных ИС, созданных по той же или подобной технологии. Это происходит потому, что на D-входе каждого триггера в ПЛУ имеется комбинационная логическая матрица, увеличивающая его время установления и тем самым уменьшающая время t_r, в течение которого должен произойти выход из состояния метастабильности, при заданном периоде t_{clk} системного тактового сигнала. Чтобы сделать значение t_r максимально возможным, не используя для этого специальных компонентов, в качестве FF2 в схеме на рис. 8.96 следует применить триггер из отдельной ИС с малым временем установления.

8.9.7. Триггеры с защитой от метастабильности

В конце 80-х годов фирма Texas Instruments и другие производители приступили к выпуску ИС малой и средней степени интеграции с триггерами, специально предназначенными для использования в синхронизирующих устройствах, встраиваемых в систему на уровне печатных плат. Микросхема *74AS4374* была, например, подобна ИС 74AS374, но с тем отличием, что отдельные триггеры заменены парами триггеров, включенных по схеме, представленной на рис. 8.101. Каждую пару триггеров можно было применить в качестве синхронизирующего устройства типа устройства, приведенного на рис. 8.96, так что с помощью одной ИС 74AS4374 оказалось возможным синхронизировать восемь асинхронных сигналов.

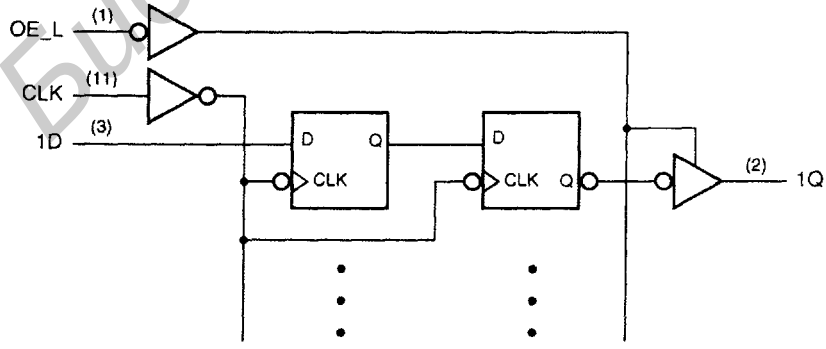


Рис. 8.101. Принципиальная схема одного из 8 сдвоенных D-триггеров в ИС 74AS4374

Внутренняя структура ИС 'AS4374 была усовершенствована таким образом, чтобы уменьшить значения τ и T_0 по сравнению с другими триггерами серии 74AS, но самым замечательным достижением было значительное сокращение времени установления t_{setup}. Поскольку вся схема синхронизирующего устройства, приведенного на рис. 8.96, в данном случае размещается в одном кристалле, между триггерами FF1 и FF2 нет входных и выходных буферов, и значение t_{setup} для триггера FF2 составляет всего 0.5 нс. У обычного триггера серии 74AS эта величина равняется 5 нс, поэтому – при τ = 0.40 нс – переход на ИС 74AS4374 приводит к увеличению среднего времени между сбоями MTBF в exp(4.5/.40) ≈ 77000 раз.

В последние годы по мере движения в сторону КМОП-технологий, обеспечивающих большее быстродействие и бóльшую плотность упаковки, специализированные компоненты типа 'AS4374 почти полностью вышли из употребления. Как можно видеть из табл. 8.35, быстродействующие ПЛУ и ИС типа CPLD вполне конкурентоспособны по величине τ с самыми быстродействующими устройствами, собранными на отдельных ИС, и в то же время предоставляют возможность объединить синхронизацию со многими другими функциями. Но все же подход, примененный в ИС 'AS4374, заслуживает воспроизведения при проектировании на основе ИС типа FPGA и на основе специализированных ИС. Другими словами, на любой стадии осуществления контроля за компоновкой схемы синхронизирующего устройства следует располагать триггеры FF1 и FF2 как можно ближе один к другому и соединять их между собой сигнальными линиями с наибольшей доступной скоростью прохождения сигнала; это обеспечит максимизацию времени установления триггера FF2.

8.9.8. Синхронизация при высокоскоростной передаче данных

Широко распространенной проблемой, возникающей в компьютерных системах, является синхронизация переноса данных, поступающих по внешним линиям, с внутренним тактовым сигналом компьютера. Простым примером служит согласование между сетевой картой персонального компьютера и линией Ethernet со скоростью передачи 100 Мбит/с. Сетевая карта может быть вставлена в разъем шины PCI с тактовой частотой 33.33 МГц. Хотя скорость передачи в сети Ethernet приблизительно кратна частоте тактового сигнала в шине компьютера, сигнал, поступающий из линии Ethernet, был отправлен другим компьютером, а тактовые сигналы на передающем и приемном конце в любом случае не синхронизированы. Тем не менее, сетевая карта обязана надежно выдать данные на шину PCI.

Эта проблема схематически представлена на рис. 8.102. Последовательные данные RDATA, представленные в коде NRZ, принимаются по линии Ethernet со скоростью 100Мбит/с. Цифровая схема ФАПЧ (Digital Phase-Locked Loop, DPLL) извлекает 100-мегагерцный тактовый сигнал RCLK из потока данных, поступающих со скоростью 100 Мбит/с, и позволяет заталкивать данные побитно в 8-разрядный регистр сдвига. В то же самое время схема синхронизации по байтам ищет в принимаемом потоке данных последовательность битов специального вида, которой отмечаются границы между байтами. Обнаруживая одну из них, схема синхронизации по байтам выдает сигнал SYNC и поступает так на

каждом восьмом такте сигнала RCLK; таким образом, сигнал SYNC возникает всякий раз, когда регистр сдвига содержит выровненный по границам 8-битовый байт принимаемых данных. В остальной части системы тактирование осуществляется тактовым сигналом SCLK с частотой 33.33 МГц. Нам необходимо переносить каждый выровненный по границам байт RBYTE[7:0] в регистр SREG, находящийся в той части системы, которая работает с тактовым сигналом SCLK. Как это можно сделать?

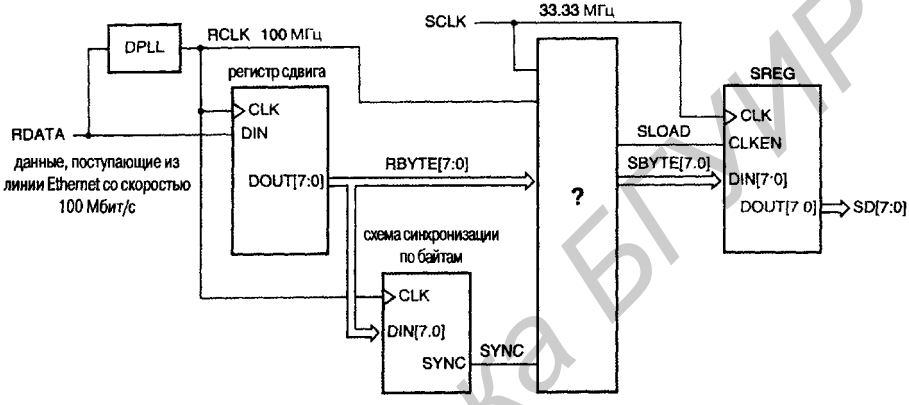


Рис. 8.102. Схематическое изображение проблемы синхронизации с сетью Ethernet

ПО ПОЛУБАЙТУ ЗА РАЗ

Здесь приводится сильно упрощенное описание процедуры приема сигналов из 100-мегабитной линии Ethernet, но этого достаточно для рассмотрения проблемы синхронизации. В действительности, скорость поступления принимаемых двоичных сигналов равна 125 Мбит/с, причем каждые 4 бита данных пользователя представлены в так называемом коде 4B5B, то есть символом, состоящим из 5 битов. Используется только 16 из 32 возможных слов кода 4B5B, и этим гарантируется, что, независимо от последовательности данных, передаваемой пользователем, число переходов в принимаемом потоке будет достаточным для извлечения из него тактового сигнала. Кроме того, код 4B5B включает специальные слова, которые передаются периодически и позволяют совсем просто осуществить синхронизацию по полубайтам (состоящим из 4 битов) и байтам.

В результате синхронизации по полубайтам типичный интерфейс 100-мегабитной сети Ethernet не видит несинхронизированного 100-мегагерцного потока битов. Вместо этого он имеет дело с несинхронизированным 25-мегагерцным потоком полубайтов. Поэтому реальное синхронизирующее устройство для сети Ethernet в деталях отличается от рассматриваемого нами, но принцип действия тот же.

На рис. 8.103 приведены несколько временных диаграмм. Сразу видно, что сигнал выравнивания байтов по границам SYNC имеет активный уровень только в течение 10 нс в пределах байта. Нет никакой надежды, что удастся каждый раз привязывать этот сигнал к системному тактовому сигналу SCLK, период которого, равный 30 нс, много больше.

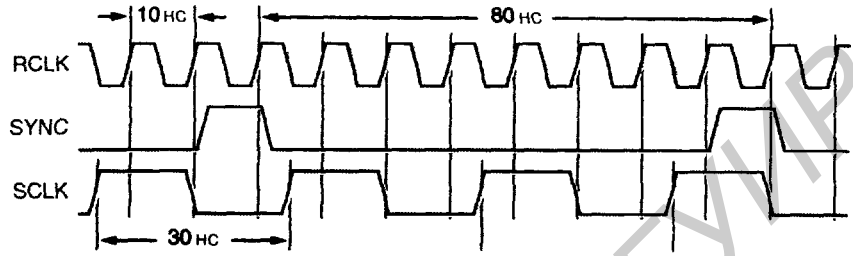


Рис. 8.103. Временные диаграммы сигналов в линии Ethernet и системный тактовый сигнал

Стратегия, которой следуют практически всегда в ситуации подобного рода, состоит в том, что сначала выровненные по границам данные заносят в регистр хранения HREG по тактовому сигналу RCLK *из принимаемого потока данных*. Это дает нам значительно больше времени, в данном случае – 80 нс, чтобы разобраться с принятым байтом. Таким образом, блок, помеченный вопросительным знаком "?" на рис. 8.102, можно заменить схемой, показанной на рис. 8.104, состоящей из регистра HREG и узла, названного "SCTRL". Функция этого узла заключается в вырабатывании сигнала SLOAD в течение точно одного периода системного тактового сигнала SCLK, равного 30 нс, так, чтобы сигналы на выходах регистра HREG на этом интервале оставались постоянными и тем самым было удовлетворено требование неизменности сигнала в течение времени установления и времени удержания регистра SREG, переключающегося по сигналу SCLK. Для остальной части интерфейса возникновение сигнала SLOAD означает, что «приняты новые данные» и что байт новых данных выдается на шину SBYTE[7:0] в течение очередного периода сигнала SCLK. На рис. 8.105 представлены возможные временные диаграммы для сигналов SLOAD и SBYTE в случае реализации такого подхода и с учетом временных диаграмм, приведенных ранее.

На рис. 8.106 показана схема, способная вырабатывать сигнал SLOAD с желательными свойствами. Основная идея состоит в использовании сигнала SYNC для установки SR-защелки в единичное состояние всякий раз, как новый байт становится доступным. Выходной сигнал этой защелки NEWBYTE опрашивается триггером FF1 по сигналу SCLK. Поскольку сигнал NEWBYTE не синхронизован с сигналом SCLK, триггер FF1 может оказаться в состоянии метастабильности, но его выходной сигнал безразличен для триггера FF2 до следующего переключающего фронта в тактовом сигнале, то есть на протяжении 30 нс. В предположении, что вентиль И является достаточно быстродействующим, мы имеем много времени для выхода из метастабильности. Сигнал на выходе триггера FF2 является требуемым сигналом SLOAD. Вентиль И позволяет удерживать единичное значение сигнала SLOAD только на интервале времени, равном периоду сигнала SCLK;

когда значение SLOAD уже равно 1, оно не может оставаться таким же с приходом очередного переключающего фронта тактового сигнала. Таким образом, у SR-защелки есть время быть сброшенной сигналом SLOAD и подготовиться к следующему байту.

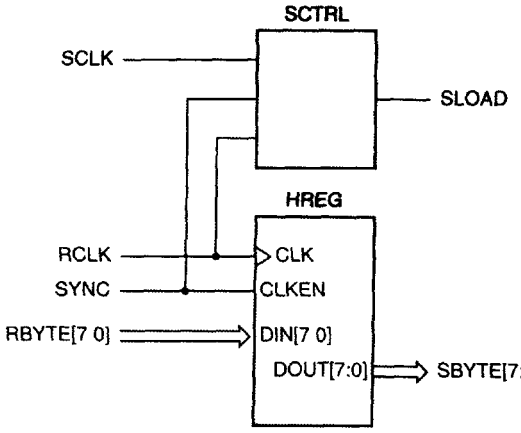


Рис. 8.104. Регистр хранения и управляющий узел

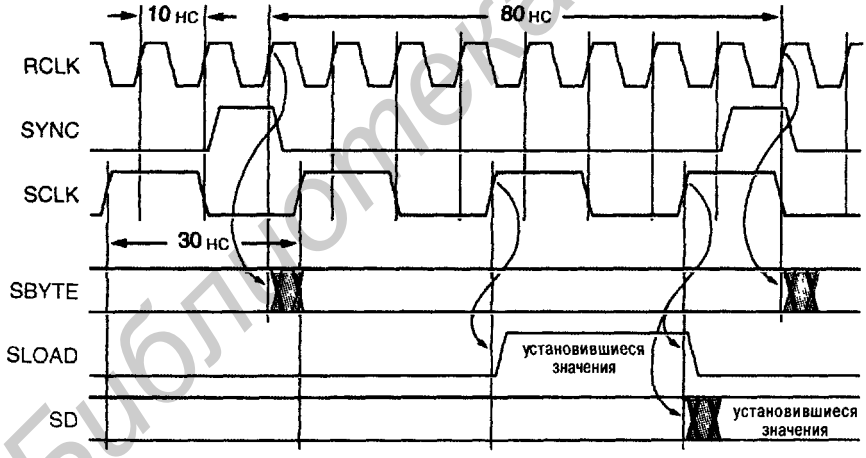


Рис. 8.105. *Временные диаграммы* сигналов в устройстве синхронизации, включая сигналы на шинах SBYTE и возможный вид сигнала SLOAD

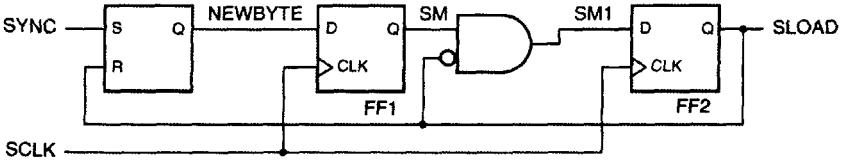


Рис. 8.106. Схема узла SCTRL, вырабатывающего сигнал SLOAD

На рис. 8.107 приведены временные диаграммы для узла SCTRL с «типичными» временными соотношениями между сигналами. Поскольку сигналы SCLK и RCLK асинхронны, сигнал SCLK может произвольно располагаться на оси времени по отношению к сигналам RCLK и SYNC. На рисунке показан случай, когда очередной нарастающий фронт сигнала SCLK приходит спустя заметное время после возникновения сигнала NEWBYTE. Хотя на рисунке изображены окна, в которых сигналы SM и SM1 могли бы быть метастабильными в общем случае, однако в той ситуации, какая указана на рисунке, метастабильность, в действительности, не наступает. Позднее мы рассмотрим, что может произойти, если фронт сигнала SCLK совпадает по времени с изменением сигнала NEWBYTE.

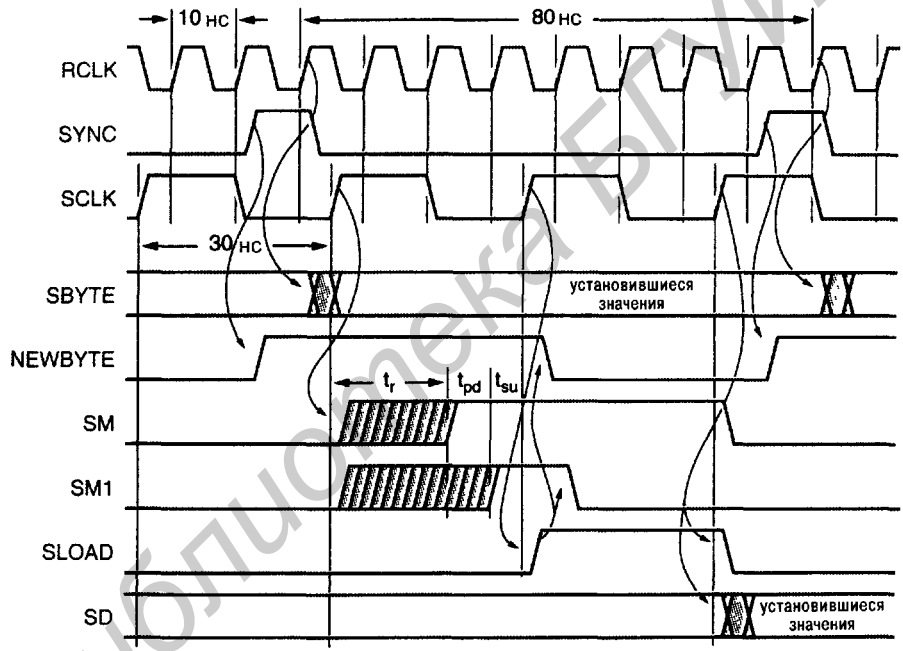


Рис. 8.107. Временные диаграммы для схемы SCTRL, приведенной на рис. 8.106

По поводу схемы на рис. 8.106 нужно сделать несколько замечаний. Во-первых, у сигнала SYNC не должно быть паразитных импульсов, т.к. он подан на вход S защелки, и он должен быть достаточной длительности, чтобы удовлетворить требованию защелки к минимальной ширине импульса. Поскольку защелка устанавливается в единичное состояние по нарастающему фронту сигнала SYNC, получается, что мы немного смошенничали: сигнал NEWBYTE может возникнуть чуть *раньше* того момента, когда новый байт действительно будет доступен на выходах регистра HREG. Но это не страшно: прежде чем произойдет загрузка регистра SREG, пройдет два периода сигнала SCLK после того, как триггер FF1 «увидит» сигнал NEWBYTE. На самом деле, можно было словчить еще больше, если имеется опережающая версия сигнала SYNC (см. задачу 8.95).

Пусть время установления D-триггера равно t_{su}, а задержка распространения для вентиля И – t_{pd}; тогда для выхода из метастабильности мы располагаем временем t_r, равным периоду сигнала SCLK (30 нс) минус $(t_{su} + t_{pd})$, как показано на рис. 8.107. Из временных диаграмм также видно, почему нельзя воспользоваться непосредственно сигналом SM для сброса SR-защелки. Возможная метастабильность сигнала SM могла бы полностью нарушить работу схемы. Например, половинное по отношению к высокому уровню значение сигнала SM может оказаться достаточным, чтобы сбросить защелку, но затем сам сигнал SM может перейти обратно на низкий уровень, и тогда сигнал SLOAD не будет выработан и мы пропустим байт. Используя выходной сигнал синхронизирующего устройства (SLOAD) как для сброса защелки, так и в качестве сигнала загрузки в части интерфейса, управляемой тактовым сигналом SCLK, мы гарантируем, что новый байт будет обнаружен и надлежащим образом воспринят в обеих частях интерфейса, работающих с тактовыми сигналами RCLK и SCLK.

Приведенные на рис.8.107 временные диаграммы относятся к штатной ситуации, но нам необходимо проанализировать также, что произойдет при другом сдвиге по времени между сигналом SCLK и сигналами RCLK и SYNC, нежели тот, какой указан на рисунке. Вы, наверное, сами сможете убедиться в том, что все будет работать хорошо, как и в рассмотренном случае, если фронт сигнала SCLK придется на тот интервал времени, когда сигнал NEWBYTE только-только переходит на высокий уровень; просто в этом случае перенос данных заканчивается чуть раньше. Более интересен случай, когда сигнал SCLK приходит чуть позже, в результате чего сигнал NEWBYTE, переходящий на высокий уровень, оказывается пропущенным и обнаруживается на один период сигнала SCLK позднее. В этом случае временные диаграммы имеют вид, указанный на рис. 8.108.

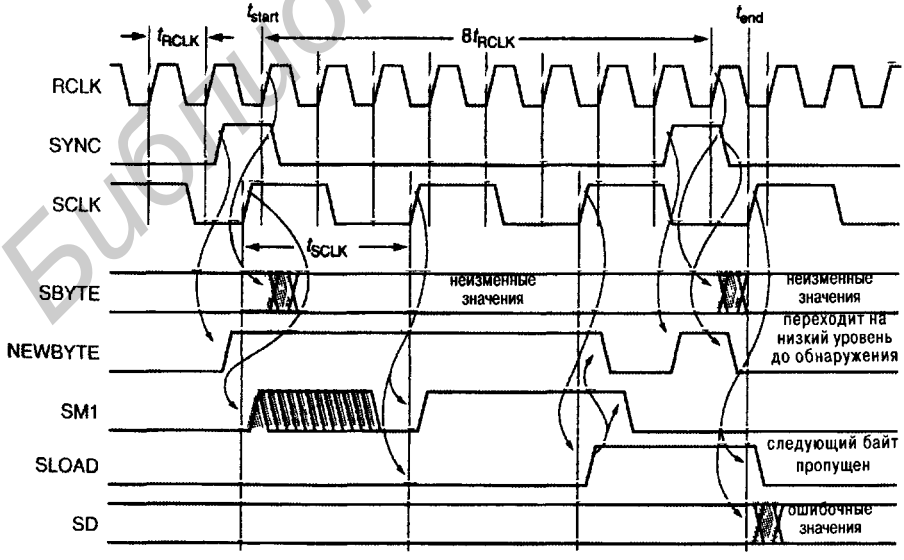


Рис. 8.108. Временные диаграммы для схемы SCTRL в случае максимальной задержки

На этих временны*х* диаграммах изображен случай, в котором сигнал NEWBYTE переходит на высокий уровень примерно в то же время, когда поступает фронт сигнала SCLK, точнее – перед приходом фронта сигнала SCLK на расстоянии, меньшем времени t_{su} триггера FF1. Таким образом, триггер FF1 может «не увидеть», что сигнал NEWBYTE имеет высокий уровень, либо выход триггера может стать метастабильным и единичное значение сигнала NEWBYTE будет надежно зафиксировано спустя лишь один период сигнала SCLK. После этого только через два периода сигнала SCLK по фронту этого сигнала произойдет загрузка значений сигналов с шины SBYTE в регистр SREG.

При таком взаимном расположении сигналов по времени возникают неприятности, потому что к моменту загрузки значения сигналов на шине SBYTE уже изменились и выражают собой *следующий* принятый байт. Кроме того, сигнал SLOAD может иметь единичное значение в то время, когда приходит импульс SYNC, соответствующий следующему принятому байту, и чуть позднее. Но тогда на обоих входах защелки S и R одновременно действуют сигналы активного уровня, и если они снимаются одновременно, то выход защелки может войти в метастабильное состояние. Если же сигнал сброса R действует дольше, как это показано на временны*х* диаграммах, то защелка останется в нулевом состоянии, и следующий принятый байт вообще не будет обнаружен и не будет передан той части интерфейса, которая работает по тактовому сигналу SCLK.

Следовательно, нам необходимо более внимательно проанализировать временны*е* диаграммы в случае максимальной задержки и решить, будет ли синхронизирующее устройство функционировать надлежащим образом. Начальной точкой в нашем анализе схемы SCTRL будет указанный на рис. 8.108 момент времени t_{start}, когда байт загружается в регистр HREG по фронту сигнала RCLK в конце импульса SYNC. Процедура заканчивается в момент времени t_{end}, когда значения сигналов с шины SBYTE загружаются в регистр SREG по фронту сигнала SCLK. Максимальная задержка между этими двумя моментами времени, которую мы назовем t_{maxd}, равна сумме следующих составляющих:

– t_{RCLK} Минус один период сигнала RCLK, расстояние между моментом t_{start} и предшествующим фронтом этого сигнала, по которому сигнал SYNC принял единичное значение. Это число отрицательно, поскольку сигнал SYNC возникает на один период тактового сигнала раньше того момента, когда фактически происходит загрузка в регистр HREG.

t_{CQ} Максимальная задержка в триггере от входа CLK до выхода Q. Предполагается, что сигнал SYNC является выходным сигналом триггера, переключающегося по сигналу RCLK. Тогда величина t_{CQ} – это интервал времени между фронтом сигнала RCLK и моментом, когда возникает сигнал SYNC.

t_{SQ} Максимальная задержка в SR-защелке на рис. 8.106 от входа S до выхода Q. Это задержка, с которой появляется сигнал NEWBYTE.

t_{su} Время установления триггера FF1 на рис. 8.106. Необходимо, чтобы сигнал NEWBYTE принимал единичное значение к началу этого интервала времени или до него. Только тогда гарантируется обнаружение.

t_{SCLK} Период сигнала SCLK. Поскольку сигналы RCLK и SCLK асинхронны, задержка между моментом возникновения сигнала NEWBYTE и моментом его фиксации в триггере FF1 по фронту сигнала SCLK может равняться одному периоду этого сигнала.

t_{SCLK} После того, как сигнал NEWBYTE обнаружен триггером FF1, проходит еще один период сигнала SCLK, прежде чем возникнет сигнал SLOAD.

t_{SCLK} Значения сигналов на шине SBYTE загружается в регистр SREG на следующем такте после возникновения сигнала SLOAD.

Таким образом, $t_{maxd} = 3t_{SCLK} + t_{CQ} + t_{SQ} + t_{su} - t_{RCLK}$. Чтобы завершить анализ, необходимо определить еще несколько параметров:

t_h Время удержания регистра SREG.

$t_{CQ(min)}$ Минимальная задержка в регистре HREG от входа CLK к выходу Q; в качестве заниженной оценки этой величины принимается 0.

t_{rec} Время восстановления SR-защелки, минимально допустимое время между переходами S и R на неактивный уровень (см. замечание, вынесенное за пределы основного текста, в конце раздела 7.2.1).

Чтобы значения сигналов на шине SBYTE были успешно загружены в регистр SREG, они должны оставаться неизменными, по крайней мере, до момента $t_{end} + t_h$. Момент времени, когда значения сигналов SBYTE изменяются, равен t_{start} плюс 8 периодов сигнала RCLK плюс $t_{CQ(min)}$. Следовательно, для надлежащей работы схемы необходимо, чтобы выполнялось неравенство:

$$t_{end} + t_h \le t_{start} + 8t_{RCLK}.$$

В случае максимальной задержки мы подставляем в это неравенство $t_{end} = t_{start} + t_{maxd}$ и, вычитая t_{start} из обеих его частей, получаем:

$$t_{maxd} + t_h \le 8t_{RCLK}.$$

Подставляя значение t_{maxd} и выполняя преобразования, получим следующее условие правильной работы схемы:

$$3t_{SCLK} + t_{CQ} + t_{SQ} + t_{su} + t_h \le 9t_{RCLK}. \quad (8.1)$$

Это очень плохо. Даже если предположить, что задержки t_{CQ}, t_{SQ}, t_{su} и t_h совсем малы, все же $3t_{SCLK}$ (90 нс) плюс что-то заведомо больше, чем $9t_{RCLK}$ (тоже 90 нс). Значит, это устройство никогда не будет работать как следует при максимальной задержке.

Но даже если результат анализа времени задержки был бы хорошим, то и в этом случае нам все же следует рассмотреть требования, которым должна удовлетворять схема SCTRL, чтобы этот узел работал как надо. В частности, необходимо гарантировать, что импульс SYNC, относящийся к следующему байту, не окончится раньше, чем через время t_{rec} после того, как будет снят сигнал SLOAD, относящийся к предыдущему байту. Таким образом, мы получаем еще одно условие правильной работы:

$$t_{end} + t_{CQ} + t_{rec} \le t_{start} + 8t_{RCLK} + t_{CQ(min)}.$$

Производя подстановки и выполняя такие же преобразования, как и раньше, мы приходим к еще одному требованию, которое не удовлетворяется в нашем устройстве:

$$3t_{SCLK} + 2t_{CQ} + t_{SQ} + t_{su} + t_{rec} \le 9t_{RCLK}. \quad (8.2)$$

Можно несколькими способами внести изменения в устройство, чтобы удовлетворить требованиям, предъявляемым к нему в худшем случае. В начале нашего рассмотрения мы уже упоминали о «мошенничестве», когда сигнал SYNC вырабатывается за один период сигнала RCLK до того, как данные оказываются записанными в регистр HREG; действительно, сигнал SYNC можно было бы выставлять и еще раньше. Если так поступать, то это поможет нам удовлетворить требованиям, относящимся к случаю максимальной задержки за счет уменьшения величины, стоящей справа в полученных соотношениях. Если, например, сигнал SYNC возникал бы на два периода сигнала RCLK раньше, то вместо "$8t_{RCLK}$" в правой части неравенств можно было бы написать "$6t_{RCLK}$". Но «бесплатный сыр бывает только в мышеловке»: мы не можем выставлять сигнал SYNC сколь угодно рано. Нам необходимо рассмотреть также случай *минимальной задержки*, чтобы гарантировать фактическое наличие нового байта в регистре HREG, когда значения сигналов на шине SBYTE будут загружаться в регистр SREG. Минимальная задержка t_{mind} между моментами t_{start} и t_{end} складывается из следующих составляющих:

$-nt_{RCLK}$ Минус n периодов сигнала RCLK, интервал времени в обратном направлении между моментом t_{start} и фронтом сигнала SYNC. В исходном варианте устройства $n = 1$.

$t_{CQ(min)}$ Это минимальная задержка между фронтом сигнала RCLK и моментом установления сигнала SYNC; в качестве заниженной оценки принимается значение 0.

t_{SQ} Это задержка, с которой вырабатывается сигнал NEWBYTE, также полагаемая равной 0.

$-t_h$ Минус время удержания триггера FF1 на рис. 8.106б. Сигнал NEWBYTE может возникнуть к концу времени удержания и все же оказаться обнаруженным.

$0t_{SCLK}$ Равное нулю число периодов сигнала SCLK. Может случиться так, что «на наше счастье» фронт сигнала SCLK придется как раз на момент окончания времени удержания триггера FF1.

t_{SCLK} Задержка на один период сигнала SCLK, с которой, как и ранее, возникает сигнал SLOAD.

t_{SCLK} Задержка на один период сигнала SCLK, с которой, как и ранее, происходит загрузка значений сигналов на шине SBYTE в регистр SREG.

Другими словами, $t_{mind} = 2t_{SCLK} - t_h - nt_{RCLK}$.

В этом случае мы должны гарантировать, что новый байт достигнет выходов регистра HREG к моменту начала времени установления регистра SREG; следовательно, должно выполняться неравенство:

$$t_{end} + t_{su} \ge t_{start} + t_{co},$$

где t_{co} – максимальная задержка в регистре HREG от тактового входа до выход
Подставляя $t_{end} = t_{start} + t_{mind}$ и вычитая t_{start} с обеих сторон, получаем:

$$t_{mind} - t_{su} \geq t_{co}.$$

Подставляя значение t_{mind} и производя преобразования, мы приходим к оконч тельному условию:

$$2t_{SCLK} - t_h - t_{su} - t_{co} \geq nt_{RCLK}. \quad (8..$$

Если, например, каждая из величин t_h, t_{su} и t_{co} равна 10 нс, то максимальное зн чение n равно 3; нельзя вырабатывать сигнал SYNC более чем на два перио тактового сигнала ранее его первоначального положения, указанного на рис. 8.10 В зависимости от других значений задержек этого может быть достаточно дл решения проблемы в случае максимальной задержки, но может быть и не дост точно; для конкретного набора компонентов этот вопрос рассматривается в з даче 8.95.

Может случиться так, что сдвиг импульса SYNC в сторону более раннего вр мени недостаточен для компенсации задержек или его нельзя осуществить в то или иной системе. Существует другое решение проблемы, которое всегда можн осуществить. Оно состоит в увеличении времени между последовательными пе реносами данных из части схемы, работающей с одним тактовым сигналом, в част схемы, работающую с другим тактовым сигналом. Это всегда возможно за сче переноса каждый раз большего числа битов. В интерфейсе сети Ethernet, напри мер, мы могли бы собирать по 16 битов в той части устройства, где переключени происходит по сигналу RCLK, и переносить по 16 битов за раз в часть устройства переключением по сигналу SCLK. В результате фигурировавшая ранее величин $8t_{RCLK}$ заменяется на $16t_{RCLK}$ и тем самым обеспечивается гораздо больший запа по времени для случая максимальной задержки. Перенося за один раз 16 битов часть устройства, работающую по тактовому сигналу SCLK, мы можем затем раз бить их на два 8-битовых отрезка, если нужно обрабатывать данные побайтно.

Характеристики устройства можно улучшить, видоизменив схему узла SCTRL На рис. 8.109 показан вариант этой схемы, в котором сигнал SLOAD вырабатыва ется непосредственно триггером, на вход данных которого поступает сигна NEWBYTE. При этом сигнал SLOAD появляется на один период сигнала SCLK рань ше, чем в нашей исходной схеме SCTRL. Кроме того, раньше сбрасывается SR защелка. Эта схема работает только в том случае, если оказываются выполнен ными следующие существенные предположения:

1. Для триггера FF1 приемлемым является уменьшенное время выхода из ме тастабильности, равное интервалу времени, в течение которого сигнал SCLK остается на высоком уровне. Метастабильность должна разрешиться до того как сигнал SCLK перейдет на низкий уровень, так как в этот момент произой дет сброс SR-защелки, если сигнал SLOAD будет иметь высокий уровень.
2. Время установления регистра SREG по входу CLKEN (рис. 8.102) меньше или равно времени, в течение которого сигнал SCLK пребывает на низком уровне. Если справедливо предыдущее предположение, то сигнал SLOAD, поданный на вход CLKEN, может оставаться метастабильным до тех пор, пока сигнал SCLK не перейдет на низкий уровень.

3. Интервал времени, в пределах которого сигнал SCLK имеет низкий уровень, *достаточно велик для того*, чтобы был выработан импульс сброса в точке RNEW, удовлетворяющий требованию SR-защелки в отношении минимальной *длительности* импульса.

Заметьте, что при выполнении этих условий правильность работы схемы зависит от коэффициента заполнения сигнала SCLK. Если сигнал SCLK является относительно медленным и его коэффициент заполнения близок к 50%, то данная схема прекрасно работает. Но если частота сигнала SCLK слишком велика, либо его коэффициент заполнения очень мал, очень велик или непредсказуем, то необходимо воспользоваться первоначальной конструкцией.

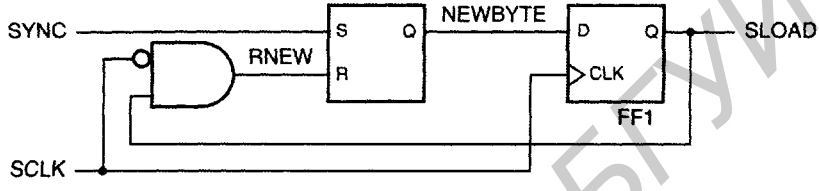


Рис. 8.109. Схема SCTRL, вырабатывающая сигнал SLOAD на половине периода тактового сигнала

Для правильной работы каждой из рассмотренных схем синхронизации требуется, чтобы частота тактового сигнала находилась в определенном диапазоне значений; у каждой схемы этот диапазон свой. Это необходимо учитывать при тестировании, когда тактовые сигналы обычно бывают более медленными, а также при модернизации, связанной с увеличением одной или обеих тактовых частот. В случае интерфейса сети Ethernet, например, не предполагается изменение стандартной частоты 100 Мбит/с, но частота тактового сигнала в шине PCI может быть повышена и стать равной не 33 МГц, а 66 МГц.

Проблемы, возникающие в связи с изменением частоты тактового сигнала, могут быть довольно тонкими. Чтобы получить представление о том, что может нарушиться, полезно посмотреть, как будет работать (*или не* работать!) синхронизирующее устройство, если одну из тактовых частот изменить в 10 раз или более.

Что произойдет, например, с временными диаграммами, представленными на рис. 8.107, если мы изменим частоту сигнала RCLK и сделаем ее равной не 100 МГц, а 10 МГц? На первый взгляд кажется, что все будет хорошо, поскольку теперь байт поступает раз в 800 нс и имеется много больше времени для переноса его в часть схемы, работающую с тактовым сигналом SCLK. Верно: неравенства (8.1) и (8.2) в данном случае удовлетворяются с много бóльшим запасом. Однако неравенство (8.3) более не выполняется, если только мы не уменьшим значение *n* до нуля! Это можно было бы исправить, вырабатывая сигнал SYNC на один такт позднее сигнала RCLK, нежели это показано на рис. 8.107.

Но даже при таком изменении некоторая проблема все же остается. На рис. 8.110 приведены новые временные диаграммы, в том числе для вырабатываемого позднее сигнала SYNC. Проблема заключается в том, что теперь длительность импульса SYNC равна 100 нс. Как и ранее, сигнал NEWBYTE (на выходе SR-защел-

ки в схеме на рис. 8.106) принимает активное значение по сигналу SYNC и сбрасывается сигналом SLOAD, но когда сигнал SLOAD заканчивается, сигнал SYNC все еще остается на активном уровне, как это видно из новых временных диаграмм. Следовательно, новый байт будет обнаружен и передан далее дважды!

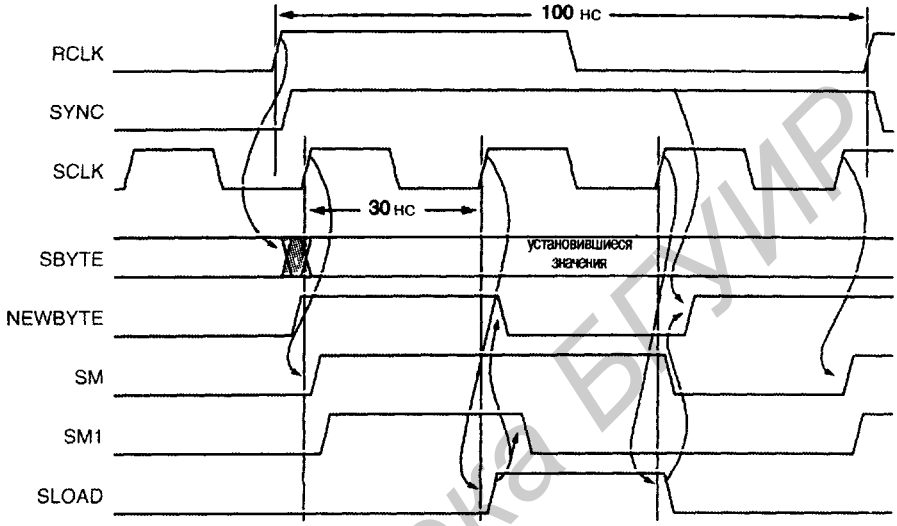


Рис. 8.110. Временные диаграммы для синхронизирующего устройства в случае медленного тактового сигнала (с частотой 10 МГц)

Решение этой проблемы состоит в том, чтобы реагировать только на нарастающий фронт сигнала SYNC, и тогда схема окажется нечувствительной к длительности импульса SYNC. В общем случае это делается путем замены SR-защелки переключающимся по фронту D-триггером, как показано на рис. 8.111. По нарастающему фронту сигнала SYNC триггер устанавливается в единичное состояние, а сигналом SLOAD, как и ранее, осуществляется асинхронный сброс.

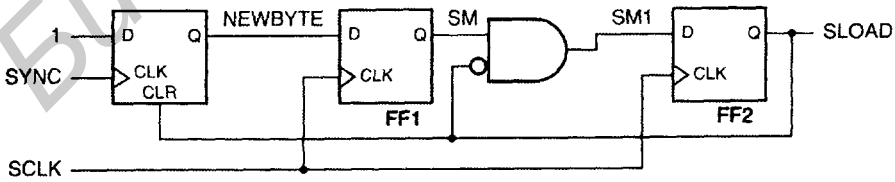


Рис. 8.111. Синхронизирующее устройство с обнаружением сигнала SYNC посредством переключения по его фронту

Приведенная на рис. 8.111 схема позволяет решить проблему, возникающую при слишком медленном сигнале RCLK, но при этом изменяются также выкладки, приводящие к соотношениям (8.1)–(8.3), результатом чего могут стать временные ограничения в каких-то других местах (см. задачу 8.96). Еще один недостаток последней

схемы состоит в том, что ее нельзя реализовать в типичном ПЛУ, у которого все триггеры переключаются одним и тем же тактовым сигналом; поэтому для обнаружения сигнала SYNC необходимо воспользоваться отдельным триггером.

Прочтя почти десять страниц, посвященные анализу всего лишь одного «простого» примера, вы, по-видимому, получили представление о том, как трудно правильно сконструировать синхронизирующее устройство. Вот несколько правил, которые могут вам помочь:

- Минимизируйте число подсистем, работающих с различными тактовыми сигналами.
- Четко определите границы между всеми тактовыми сигналами и в явном виде поместите на этих границах синхронизирующие устройства.
- Обеспечьте для каждого синхронизирующего устройства достаточное время выхода из метастабильности, чтобы сбои синхронизирующих устройств были редкими и происходили с много меньшей вероятностью, чем возникновение неисправности в других местах.
- Проанализируйте работу синхронизирующего устройства при различных возможных сдвигах сигналов во времени, в том числе при более быстрых и более медленных тактовых сигналах, которые могут подаваться на схему при моделировании или при модернизации системы.
- Осуществите моделирование работы системы в широком диапазоне возможных временны*х* соотношений между сигналами.

Последнее правило может оказаться ловушкой для тех разработчиков, кто полагается на современные мощные и быстродействующие средства моделирования при поиске своих ошибок. Само по себе моделирование не может избавить от необходимости следования предыдущим четырем правилам. Если игнорировать эти правила, то можно столкнуться с проблемами, которые не обнаруживаются моделированием в типичных случаях, когда перебирается небольшое число вариантов соотношений между сигналами. Из всех цифровых схем синхронизирующие устройства являются такими конструкциями, для которых важнее всего быть «правильным по идее»!

9.2. Примеры проектирования на языке VHDL

В параграфе 7.12 мы уже объяснили, что основные правила языка VHDL, введенные нами еще в параграфе 4.7, в том числе, процессы – это почти все, что нужно для описания работы последовательностных схем. В отличие от языка ABEL в языке VHDL нет никаких специальных языковых средств для создания конечных автоматов. Вместо этого большинство программистов описывает конечные автоматы комбинацией имеющихся «обычных» средств, чаще всего с помощью перечислимых типов и операторов `case`. Мы воспользуемся этим методом в примерах данного параграфа.

9.2.1. Несколько простых автоматов

В разделе 7.4.1 мы проиллюстрировали процедуру создания автомата по таблице состояний на примере следующей простой задачи:

> Построить тактируемый синхронный конечный автомат с двумя входами A и B и одним выходом Z, сигнал на котором равен 1, если
>
> – сигнал на входе A имел одно и то же значение на двух последних тактах *или*
>
> – сигнал на входе B оставался равным 1 с тех пор, когда в последний раз было выполнено первое условие.
>
> В противном случае выходной сигнал должен равняться 0.

В среде проектирования на основе того или иного языка описания схем можно многими способами написать программу, удовлетворяющую сформулированным требованиям. Мы рассмотрим несколько таких возможностей.

Первый подход заключается в том, чтобы составить от руки таблицу состояний и значений выходного сигнала, а затем вручную преобразовать ее в программу. Поскольку в разделе 7.4.1 нами уже была составлена такая таблица состояний, то почему бы нам не воспользоваться ею? В табл. 9.12 эта таблица воспроизведена заново, а в табл. 9.13 приведена соответствующая VHDL-программа.

Табл. 9.12. Таблица состояний и значений выходного сигнала в рассматриваемом примере

	A B				
S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
			S*		

Как обычно, объявление объекта в языке VHDL содержит только указания на входы и выходы; в нашем примере это – `CLOCK`, `A`, `B` и `Z`. Определением архитектуры задается внутреннее функционирование конечного автомата. Первое, что делается в архитектуре, – это образование перечислимого типа `Sreg_type`, значениями которого являются идентификаторы, соответствующие именам состояний. Затем объявляется сигнал `Sreg`, который будет применен для хранения текущего состояния автомата. С учетом последующего использования этого сигнала, он будет при синтезе отображен в переключающийся по фронту регистр.

В части архитектуры, содержащей операторы, имеется два параллельных оператора: процесс и оператор избирательного присваивания. Процесс чувствителен только к сигналу `CLOCK` и им осуществляются все переходы из одного состояния в другое, которые происходят по нарастающему фронту сигнала `CLOCK`. Оператором "`if`" в пределах процесса обнаруживается нарастающий фронт, а в операторе `case` для каждого состояния перебираются все возможные переходы.

В операторе `case` имеется шесть альтернатив, соответствующих пяти состояниям, определенным явно, и случаю, в котором собраны все другие состояния. Ради надежности, в случае "`others`" автомат отсылается назад в состояние `INIT`. В каждой из альтернатив используется вложенный оператор "`if`", чтобы непосредственно перечислить все комбинации входных сигналов `A` и `B`. Однако, строго говоря, нет необходимости включать те комбинации, при которых автомат остается в текущем состоянии; сигнал `Sreg` сохраняет текущее состояние до тех пор, пока в каком-то из возобновлений процесса ему не будет присвоено новое значение.

Оператор избирательного присваивания в конце табл. 9.13 вырабатывает единственный выходной сигнал данного автомата – сигнал `Z` типа Мура, значение которого зависит от текущего состояния. Возможно, легче было определить также в пределах этого оператора выходные сигналы типа Мили. Другими словами, значе-

ние Z могло бы быть функцией не только текущего состояния, но и входных сигналов. Поскольку оператор "with" является параллельным оператором, любые изменения входных сигналов сразу же отражаются на значении выходного сигнала Z.

Табл. 9.13. VHDL-программа для рассматриваемого конечного автомата

```
library IEEE;
use IEEE.std_logic_1164.all;

entity snexaaip is
  port ( CLOCK, A, B: in STDJLOGIC;
         Z: out STDJLQGIC );
end;

architecture smexamp_arch of smexamp is
type Sregjtype is (INIT, AO, Al, 0K0, OKI);
signal Sreg: Sreg_type;


  process (CLOCK) -- state-machine states and transitions
  begin
    if CLOCK'event and CLOCK = 'i' then
      case Sreg is
        when INIT -> if    A='0' then Sreg <* A0;
                      elsif A='1' then Sreg <- Al;   end if;
        when AO =>    if    A='0' then Sreg <* 0K0;
                      elsif A='1' then Sreg <~ Al;  end if;
        when Al ->    if    A='0' then Sreg <* A0;
                      elsif A» 'i' then Sreg <» OKI;  end if;
        when 0K0 =>   if    A~'0' then Sreg <= 0K0;
                      elsif A='i' and f3-'0' then Sreg <- Al;
                      elsif A~ 1' and B"" X  then Sreg <~ OKI;  end if;
        when OKI «>   if    k~'0' and I      then Sreg <- AO;
                      elsif A='0' and I      then Sreg <~ OKO;
                      elsif A='1' then Sreg <= OKI;             end if
        when others       I <* INIT;
      end case;
    end if;
  end process;

  with Sreg select  -- output values based on state
    Z <» '0' when INIT | AO I Al,
         '1' when 0K0 ! OKI,
         '0' when others;

end saexamp_arch;
```

На самом деле, нам следовало предусмотреть в табл. 9.13 вход сброса (см. заключенное рамку замечание в конце раздела 9.1.4). Сигнал RESET легко включить, видоизменив объявление объекта и добавив еще одно предложение в оператор "if" в определении архитектуры. Если сигнал RESET принимает активное значение, то автомат должен перейти в состояние INIT; в противном случае еле-

дует исполнять оператор case. В зависимости от того, когда проверяется сигнал RESET, – до проверки поступления фронта тактового сигнала или после, – реализуется асинхронный или синхронный сброс (см. задачу 9.10).

Так как же насчет кодирования состояний? Табл. 9.13 не содержит никакой информации о том, как должны присваиваться комбинации переменных состояния состояниям с теми или иными именами. Ничего не известно даже о том, сколько необходимо двоичных переменных состояния.

Средства синтеза вольны связывать с идентификаторами перечислимого типа любые целые числа или двоичные комбинации, какие им только понравятся, но в типичном случае состояниям будут поставлены в соответствие целые числа, начиная с 0, в том порядке, в каком перечислены их имена. Затем для представления этих чисел будет использовано наименьшее возможное число битов, равное $\lceil \log_2 s \rceil$ при наличии s состояний. Таким образом, синтез по программе из табл. 9.13 будет происходить с тем же самым «простейшим» кодированием состояний, которое было выбрано нами в исходном примере (см. табл. 7.7). Однако средствами языка VHDL можно заставить компилятор принять какой-то другой способ кодирования.

Один из способов навязать определенное кодирование состояний заключается в употреблении оператора "attribute", как это сделано в табл. 9.14. Здесь "enum_encoding" – определяемый пользователем признак, значением которого является строка, указывающая, какое именно кодирование путем перечисления должно быть использовано средствами синтеза. Процессор языка VHDL игнорирует это значение, но передает имя признака и его значение средствам синтеза. Признак "enum_encoding" определен в большинстве средств синтеза и известен им, в том числе средствам синтеза фирмы Synopsys, Inc. Заметьте, что программой должен «использоваться» пакет "attributes" фирмы Synopsys; это необходимо для того, чтобы VHDL-компилятор распознал "enum_encoding" как законный определяемый пользователем признак. Между прочим, кодирование состояний, указанное в последней программе, эквивалентно «почти прямому» кодированию из табл. 7.7.

```
library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;
...
architecture smexampe_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
attribute enum_encoding of Sreg_type: type is
          "0000 0001 0010 0100 1000";
signal Sreg: Sreg_type;
...
```

Табл. 9.14. Использование признака для того, чтобы заставить средства синтеза кодировать состояния по правилу перечисления

Другой способ навязать то или иное кодирование состояний без обращения к внешним пакетам и без учета признаков при синтезе заключается в более явном задании регистра состояний с помощью обычных логических типов данных. Этот подход представлен в табл. 9.15. Здесь сигнал Sreg определен как 4-разрядный

элемент типа STD_LOGIC_VECTOR и введены константы, позволяющие повсюду в программе ссылаться на состояния по их именам. Никаких других изменений в программе не требуется.

Табл. 9.15. Использование обычной логики и констант для задания способа кодирования состояний

```
library IEEE;
use IEEE.std_logic_1164.all;
...
architecture smexampc_arch of smexamp is
subtype Sreg_type is STD_LOGIC_VECTOR (1 to 4);
constant INIT: Sreg_type := "0000";
constant A0  : Sreg_type := "0001";
constant A1  : Sreg_type := "0010";
constant OK0 : Sreg_type := "0100";
constant OK1 : Sreg_type := "1000";
signal Sreg: Sreg_type;
...
```

Возвращаясь к нашей исходной VHDL-программе в табл. 9.13, отметим, что возможна еще одна интересная модификация. Исходной программой определяется обычный конечный автомат Мура со структурой, показанной на рис. 9.8(а). Что произойдет, если мы преобразуем оператор избирательного присваивания выходной логики в оператор case и переместим его в процесс, осуществляющий переходы из одного состояния в другое? Поступив так, мы создадим автомат, который в результате синтеза, вероятнее всего, будет иметь структуру, показанную на рис. 9.8(b). По существу, это автомат Мили с конвейерными выходами, и его поведение неотличимо от поведения исходного автомата, за исключением временных характеристик. Мы сократили задержку распространения от входа CLOCK до выхода Z, вырабатывая сигнал Z непосредственно на регистровом выходе, но одновременно с этим увеличилось требуемое время установления сигналов A и B по отношению к сигналу CLOCK из-за дополнительной задержки на прохождение сигнала через выходную логику ко входу D выходного регистра.

Все решения задачи построения рассматриваемого конечного автомата, о которых шла речь до сих пор, основывались на таблице состояний, которую мы первоначально составили вручную в разделе 7.4.1. Можно, однако, написать VHDL-программу непосредственно, без составления таблицы состояний вручную.

Основная идея упрощения следует из исходной формулировки задачи, приведенной в начале данного раздела, и состоит в исключении последнего значения входного сигнала A из определения состояний. Вместо этого предусматривается наличие отдельного регистра LASTA для отслеживания упомянутой величины. В этом случае необходимо определить только два состояния, помимо исходного состояния INIT: состояние LOOKING («смотреть дальше в ожидании совпадения») и состояние OK («имеет место совпадение двух последовательных значений A или сигнал B остается равным 1 с момента последнего совпадения»). VHDL-архитектура, реализующая этот подход приведена в табл. 9.16. В процессе, возбуждаемом сигналом

CLOCK, первый оператор присваивания создает регистр LASTA, а оператор case создает автомат с тремя состояниями. В конце программы выходной сигнал Z определяется как результат простого комбинационного обнаружения состояния OK.

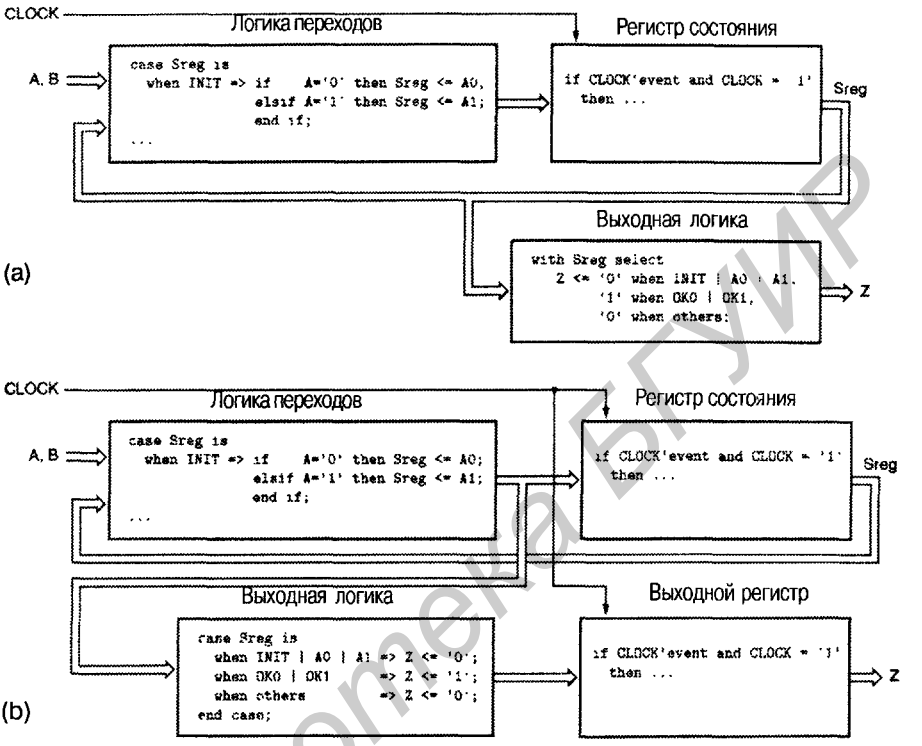


Рис. 9.8. Структура конечных автоматов, подразумеваемая VHDL-программами: (a) автомат Мура с комбинационной выходной логикой; (b) конвейерный автомат Мили с выходным регистром

КОВАРНОЕ ВРЕМЯ

При записи VHDL-архитектуры, соответствующей схеме на рис. 9.8(b), очень важно добавить сигнал Sreg в список чувствительности процесса. Действительно, значение Z определяется оператором case выходной логики как функция от значения Sreg. При первом исполнении процесса с приходом нарастающего фронта тактового сигнала значение Sreg повсюду соответствует старому состоянию автомата. Это имеет место потому, что Sreg является сигналом, а не переменной. Как было объяснено в разделе 4.7.9, сигналы, изменяющиеся внутри процесса, не принимают новых значений до тех пор, пока не будет сделан по крайней мере один элементарный сдвиг по времени *после* того, как процесс начал исполняться. Помещая Sreg в список чувствительности, мы гарантируем, что процесс будет повторен, так что конечное значение Z будет выработано с учетом нового значения Sreg.

Табл. 9.16. Упрощенный подход к построению рассматриваемого конечного автомата средствами VHDL

```
architecture smexampa_arch of smexamp is
type Sreg_type is (INIT, LOOKING, OK);
signal Sreg: Sreg_type;
signal lastA: STD_LOGIC;
begin
  process (CLOCK) -- state-machine states and transitions
  begin
    if CLOCK'event and CLOCK = '1' then
      lastA <= A;
      case Sreg is
        when INIT    =>                    Sreg <= LOOKING;
        when LOOKING => if    A=lastA then Sreg <= OK;
                        else               Sreg <= LOOKING;
                        end if;
        when OK      => if    B='1'   then Sreg <= OK;
                        elsif A=lastA then Sreg <= OK;
                        else               Sreg <= LOOKING;
                        end if;
        when others  =>                    Sreg <= INIT;
      end case;
    end if;
  end process;

  with Sreg select  -- output values based on state
    Z <= '1' when OK,
         '0' when others;

end smexampa_arch;
```

Другим простым примером конечного автомата является «автомат, считающий число единиц». Задача формулируется так:

Построить тактируемый синхронный конечный автомат с двумя входами X и Y и одним выходом Z. Выходной сигнал должен равняться 1, если число единиц, поступивших на входы X и Y с момента запуска, кратно 4; в противном случае сигнал Z должен равняться 0.

В табл. 7.12 имеется составленная нами для этого автомата таблица состояний. Однако мы воспользуемся возможностями счета, предоставляемыми пакетом IEEE `std_logic_arith`, и напишем VHDL-программу для такого автомата непосредственно.

Как всегда существует много способов решить эту проблему. Наше решение представлено в табл. 9.17. Мы выбрали такой способ, который позволяет проиллюстрировать несколько различных особенностей языка. Внутри архитектуры объявлен подтип `COUNTER` для двухразрядных величин типа `UNSIGNED`. Подсчитываемое число единиц хранится в сигнале `COUNT` этого подтипа, а постоянная `ZERO` того же подтипа нужна для инициализации и проверки значения `COUNT`.

Табл. 9.17. VHDL-программа для автомата, считающего число единиц

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Vonescnt is
  port ( CLOCK, RESET, X, Y: in STD_LOGIC;
         Z: out STD_LOGIC );
end;

architecture Vonescnt_arch of Vonescnt is
subtype COUNTER is UNSIGNED (1 downto 0);
signal COUNT: COUNTER;
constant ZERO: COUNTER := "00";
begin

process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then COUNT <= ZERO;
      else COUNT <= COUNT + ('0', X) + ('0', X);
      end if;
    end if;
  end process;

Z <= '1' when COUNT = ZERO else '0';

end Vonescnt_arch;
```

В процессе применен обычный метод обнаружения нарастающего фронта сигнала CLOCK. Предложением "if" выполняется синхронный сброс, а предложением else осуществляется простое добавление 0, 1 или 2 к содержимому COUNT, в зависимости от значений X и Y. Напомним, что выражение вида "('0', X)" – это массив-литерал; он образован здесь двумя элементами типа std_logic: нулем '0' и текущим значением X. Тип этого литерала совместим с типом UNSIGNED, поскольку число элементов у них и их тип одинаковы; поэтому их можно объединить операцией "+", определенной в пакете std_logic_arith. Параллельный сигнальный оператор присваивания, расположенный вне процесса, вырабатывает 1 на выходе типа Мура Z, когда значение COUNT равно 0.

С точки зрения синтеза оператор "if" и оператор присваивания значения сигналу COUNT не обязательно порождают компактную и быстродействующую схему. В случае простых средств синтеза это могут быть два 2-разрядных сумматора, соединенные последовательно. В табл. 9.18 показан другой подход, при котором «умные» средства оказываются способными синтезировать более компактную схему инкрементирования для каждого из двух сложений. В любом случае, представление в виде альтернатив оператора case позволяет двум сумматорам или схемам инкрементирования работать параллельно, а для переключения на один из выходов в соответствии с выбираемой альтернативой можно воспользоваться мультиплексором.

Табл. 9.18. Другой вариант процесса для автомата, считающего число единиц

```
process (CLOCK)
variable ONES: STD_LOGIC_VECTOR (1 to 2);
begin
  if CLOCK'event and CLOCK = '1' then
    ONES := (X, Y);
    if RESET = '1' then COUNT <= ZERO;
    else case ONES is
           when "01" | "10" => COUNT <= COUNT + "01";
           when "11"        => COUNT <= COUNT + "10";
           when others      => null;
         end case;
    end if;
  end if;
end process;
```

Наш последний пример в этом разделе – конечный автомат, управляющий кодовым замком, из параграфа 7.4 (ниже выход HINT, имевшийся в исходном варианте, опущен):

> Построить тактируемый синхронный конечный автомат с одним входом X и одним выходом UNLK. Сигнал на выходе UNLK должен принимать значение 1 тогда и только тогда, когда X равно 0 и последовательность значений входного сигнала X на семи предшествующих тактах имела вид: 0110111.

В табл. 7.14 приведена составленная нами таблица состояний. Но мы снова применим другой, более наглядный подход. Примем во внимание, что в данном случае сигнал на выходе автомата в любой момент времени полностью определяется значениями его входного сигнала на последних восьми тактах. Поэтому при проектировании этого автомата можно использовать так называемый принцип «конечной памяти» (см. помещенное в рамку замечание в конце раздела 9.1.2). В соответствии с этим принципом мы в явном виде отслеживаем семь последних значений входного сигнала и вырабатываем выходной сигнал как комбинационную функцию этих значений.

В табл. 9.19 приведена VHDL-программа, реализующая этот принцип. Архитектура содержит процесс, осуществляющий слежение за семью последними значениями X с помощью конструкции, являющейся, по существу, регистром сдвига, в котором бит на позиции с номером 7 представляет собой самое старое значение X. (Напомним, что оператор "&" в языке VHDL выполняет конкатенацию массивов.) Расположенный вне процесса параллельный сигнальный оператор присваивания вырабатывает 1 на выходе типа Мили UNLK, когда X равен 0, а семь предыдущих битов согласуются с ожидаемой комбинацией.

Табл. 9.19. VHDL-программа, реализующая принцип конечной памяти применительно к конечному автомату, управляющему кодовым замком

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vcomblck is
  port ( CLOCK, RESET, X: in STD_LOGIC;
         UNLK: out STD_LOGIC );
end;

architecture Vcomblck_arch of Vcomblck is
signal XHISTORY: STD_LOGIC_VECTOR (7 downto 1);
constant COMBINATION: STD_LOGIC_VECTOR (7 downto 1) := "0110111";
begin

  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then XHISTORY <= "0000000";
      else XHISTORY <= XHISTORY(6 downto 1) & X;
      end if;
    end if;
  end process;

  UNLK <= '1' when (XHISTORY=COMBINATION) and (X='0') else '0';

end Vcomblck_arch;
```

9.2.2. Задние огни автомобиля марки Ford Thunderbird

В параграфе 7.5 был описан и построен автомат для управления задними огнями автомобиля марки Ford Thunderbird. В табл. 9.20 представлена соответствующая VHDL-программа. Переходы автомата из одного состояния в другое происходят точно так же, как это изображено в виде диаграммы состояний на рис. 7.64. Применена запись состояний в форме выходного кода; это оказывается возможным, поскольку каждому из состояний автомата соответствует своя комбинация значений выходных сигналов, зажигающих задние фонари.

ВОЗМОЖНЫЕ УСОВЕРШЕНСТВОВАНИЯ

При описании конечного автомата на языке VHDL нет необходимости в явном присваивании следующего состояния, если оно остается тем же самым состоянием, в котором автомат уже находится. При исполнении процесса сигнал в языке VHDL сохраняет свое значение, если не выполняется присвоение ему нового значения. Поэтому заключительное предложение "`else`" в табл. 9.20 в состоянии `IDLE` можно было бы опустить, и это не повлияло бы на работу автомата.

Кроме того, можно повысить надежность конечного автомата, заменив оператор "`null`" в случае "`when others`" переходом в состояние `IDLE`.

Табл. 9.20. VHDL-программа для автомата, управляющего задними огнями автомобиля марки Ford Thunderbird

```
entity Vtbird is
  port ( CLOCK, RESET, LEFT, RIGHT, HAZ: in STD_LOGIC;
         LIGHTS  buffer STD_LOGIC_VECTOR (1 to 6) );
end;

architecture Vtbird_arch of Vtbird is
constant IDLE : STD_LOGIC_VECTOR (1 to 6) := "000000",
constant L3   : STD_LOGIC_VECTOR (1 to 6) := "111000",
constant L2   : STD_LOGIC_VECTOR (1 to 6) := "110000";
constant L1   : STD_LOGIC_VECTOR (1 to 6) := "100000";
constant R1   : STD_LOGIC_VECTOR (1 to 6)  = "000001",
constant R2   : STD_LOGIC VECTOR (1 to 6) := "000011";
constant R3   : STD_LOGIC VECTOR (1 to 6) := "000111",
constant LR3    STD_LOGIC_VECTOR (1 to 6)  = "111111',
begin
  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then LIGHTS <= IDLE, else
        case LIGHTS is
          when IDLE -> if HAZ='1' or (LEFT='1' and RIGHT='1') then LIGHTS <= LR3;
                       elsif LEFT= 1'                         then LIGHTS <= L1;
                       elsif RIGHT='1'                        then LIGHTS <= R1,
                       else                                        LIGHTS <= IDLE;
                       end if,
          when L1   => if HAZ='1' then LIGHTS <= LR3, else LIGHTS <= L2; end if;
          when L2   -> if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L3, end if,
          when L3   => LIGHTS <= IDLE;
          when R1   => if HAZ='1' then LIGHTS <= LR3, else LIGHTS <= R2, end if,
          when R2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R3, end if,
          when R3   => LIGHTS <= IDLE;
          when LR3  => LIGHTS <= IDLE,
          when others => null,
        end case,
      end if;
    end if;
  end process,
end Vtbird_arch,
```

9.2.3. Игра на угадывание

Задача построения автомата для «игры на угадывание» была сформулирована следующим образом:

> Построить тактируемый синхронный автомат с четырьмя входами G1–G4, подключенными к кнопкам. У автомата четыре выхода L1–L4, к которым подключены лампочки или светодиоды, расположенные рядом с кнопками с теми же номерами. Имеется также выход ERR, к которому подключена красная лампочка. При нормальной работе на выходах L1–L4 индицируется комбинация «1 из 4». На каждом такте комбинация сдвигается на одну позицию; частота тактового сигнала равна 4 Гц.
>
> Задача игрока состоит в том, чтобы вовремя нажать кнопку, соответствующую горящей лампочке. При нажатии *i*-ой кнопки вырабатывается единичный сигнал Gi. Если подан «неправильный» сигнал, то возникает сигнал на выходе ERR и загорается красная лампочка, это происходит в том случае, когда автомат на очередном такте обнаруживает сигнал, номер которого не совпадает с номером лампочки, зажженной на предыдущем такте. Когда кнопка нажата, игра останавливается, и сигнал на выходе ERR сохраняет свое значение в течение одного или нескольких тактов, пока не будет снят удерживаемый вами сигнал Gi, и тогда игра возобновляется.

Как мы видели в разделе 7.7.1, у этого автомата шесть состояний: четыре состояния соответствуют зажженным лампочкам, а два – для тех случаев, когда игра остановлена после правильного или ошибочного нажатия кнопки. VHDL-программа для игры на угадывание приведена в табл. 9.21. В этом варианте устройство имеет также вход RESET: сигнал на этом входе заставляет устройство перейти в известное начальное состояние.

Эта программа является почти непосредственным переводом на язык VHDL диаграммы состояний, изображенной на рис. 7.66. Единственной, по-видимому, ее особенностью, которая заслуживает упоминания, является случай "SOK | SERR". Поскольку переходы из этих двух состояний в следующие состояния совершенно одинаковы (нужно переходить в состояние S1 или оставаться в текущем состоянии), их можно обрабатывать как один и тот же случай. Однако эта хитрость, позволяющая уменьшить число строк в тексте программы, нежелательна, в частности, с точки зрения документации на этот конечный автомат и его отладки. Но автору эта хитрость позволила сократить размер программы до одной страницы в книге!

В программе, приведенной в табл. 9.21, кодирование состояний не задано; типичный синтезатор использует три бита для Sreg и кодирует шесть состояний в порядке следования двоичных комбинаций 000–101. Применительно к этому конечному автомату можно воспользоваться также записью состояний в форме выходного кода, то есть представить их с помощью уже имеющихся сигналов зажигания лампочек и сигнала ошибки. В языке VHDL нет удобного механизма для объединения выходных сигналов, определенных в данном объекте, в одно целое и представления ими состояний, но этого можно все-таки достичь так, как показано в табл. 9.22. Соответствие между входными сигналами и битами в новом 5-разрядном регистре Sreg указанно в комментарии, а операторы присваивания значений выходным сигналам видоизменены таким образом, чтобы выбирать подходящий бит, а не обнаруживать состояние в целом.

Табл. 9.21. VHDL-программа, описывающая автомат для игры на угадывание

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vggame is
  port ( CLOCK, RESET, G1, G2, G3, G4: in  STD_LOGIC;
         L1, L2, L3, L4, ERR:          out STD_LOGIC );
end;

architecture Vggame_arch of Vggame is
type Sreg_type is (S1, S2, S3, S4, SOK, SERR);
signal Sreg: Sreg_type;
begin

  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then Sreg <= SOK; else
        case Sreg is
          when S1 => if    G2='1' or G3='1' or G4='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S2;
                     end if;
          when S2 => if    G1='1' or G3='1' or G4='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S3;
                     end if;
          when S3 => if    G1='1' or G2='1' or G4='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S4;
                     end if;
          when S4 => if    G1='1' or G2='1' or G3='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S1;
                     end if;
          when SOK | SERR => if G1='0' and G2='0' and G3='0' and G4='0'
                             then Sreg <= S1; end if;
          when others => Sreg <= S1;
        end case;
      end if;
    end if;
  end process;

  L1  <= '1' when Sreg = S1   else '0';
  L2  <= '1' when Sreg = S2   else '0';
  L3  <= '1' when Sreg = S3   else '0';
  L4  <= '1' when Sreg = S4   else '0';
  ERR <= '1' when Sreg = SERR else '0';

end Vggame_arch;
```

Табл. 9.22. VHDL-архитектура для игры на угадывание с записью состояний в форме выходного кода

```
architecture Vggameoc_arch of Vggame is
signal Sreg: STD_LOGIC_VECTOR (1 to 5);
-- bit positions of output-coded assignment: L1, L2, L3, L4, ERR
constant S1:   STD_LOGIC_VECTOR (1 to 5) := "10000";
constant S2:   STD_LOGIC_VECTOR (1 to 5) := "01000";
constant S3:   STD_LOGIC_VECTOR (1 to 5) := "00100";
constant S4:   STD_LOGIC_VECTOR (1 to 5) := "00010";
constant SERR: STD_LOGIC_VECTOR (1 to 5) := "00001";
constant SOK:  STD_LOGIC_VECTOR (1 to 5) := "00000";
begin

  process (CLOCK)
  ...              (no change to process)
  end process;

  L1  <= Sreg(1);
  L2  <= Sreg(2);
  L3  <= Sreg(3);
  L4  <= Sreg(4);
  ERR <= Sreg(5);

end Vggameoc_arch;
```

9.2.4. Продолжение работы над контроллерами светофоров

Тем из вас, кто прочел пример из раздела 9.1.5, уже известны мои разглагольствования об ужасных контроллерах светофоров в г. Саннивейл, шт. Калифорния. Ситуация на самом деле выглядит такой, как если бы контроллеры были *специально* спроектированы так, чтобы сделать возможно бóльшим время простоя автомобилей на перекрестках. В этом разделе мы разработаем контроллер для светофора, поведение которого будет подчеркнуто напоминать работу светофоров в г. Саннивейл.

На незагруженном перекрестке имеются датчики движения и светофоры, показанные на рис. 9.5. (Если бы это было в Чикаго, то самое большее, чем был бы отмечен такой перекресток, – это знак «уступи дорогу».) Светофорами управляет автомат с частотой тактового сигнала 1 Гц; у него имеются таймер и четыре входа:

NSCAR — Сигнал принимает активное значение, если хотя бы один автомобиль находится в поле действия одного из датчиков в направлении движения «север-юг» по любую сторону перекрестка.

EWCAR — Сигнал принимает активное значение, если хотя бы один автомобиль находится в поле действия одного из датчиков в направлении движения «восток-запад» по любую сторону перекрестка.

TMLONG — Сигнал принимает активное значение, если с момента начала работы таймера прошло более пяти минут; он остается на активном уровне до тех пор, пока таймер не будет сброшен.

TMSHORT — Сигнал принимает активное значение, если с момента начала работы таймера прошло более пяти секунд; он остается на активном уровне до тех пор, пока таймер не будет сброшен.

У этого конечного автомата семь выходных сигналов.

NSRED, NSYELLOW, NSGREEN — Сигналы, зажигающие красный, желтый и зеленый свет в направлении «север-юг» соответственно

EWRED, EWYELLOW, EWGREEN — Сигналы, зажигающие красный, желтый и зеленый свет в направлении «восток-запад» соответственно.

TMRESET — Когда этот сигнал принимает активное значение, таймер сбрасывается, а сигналы TMSHORT и TMLONG переходят на неактивный уровень. Таймер начинает отсчет времени, когда на неактивный уровень переходит сигнал TMRESET.

VHDL-программой, приведенной в табл. 9.23, реализуется типичный, одобренный местными властями алгоритм управления светофорами. Этот алгоритм обеспечивает два часто наблюдаемых режима работы нашего «проворного» светофора. Ночью, при малой интенсивности движения, он удерживает автомобиль в состоянии ожидания до пяти минут, если только не появляется автомобиль на поперечной улице; в этом случае светофор переключается так, чтобы остановить движение в поперечном направлении и пропустить ожидающий автомобиль (Датчик «раннего оповещения» установлен достаточно далеко, чтобы сигналы светофора успели измениться до того, как приближающийся автомобиль достигнет перекрестка.) Днем, при напряженном движении всегда имеются автомобили, ожидающие проезда в обоих направлениях; тогда светофор переключается каждые пять секунд, чтобы минимизировать пропускную способность перекрестка и максимизировать время ожидания для всех, подталкивая тем самым возмущенную общественность к мысли о необходимости повышения налогов для решения этой проблемы.

При написании этой программы мы воспользовались имевшейся возможностью и добавили контроллеру вход OVERRIDE. Подавая сигнал на этот вход, полицейский может заблокировать работу контроллера и заставить светофор мигать красным светом (с частотой тактового сигнала FLASHCLK), и тогда у него появляется возможность вручную растаскивать пробки, возникающие благодаря этому удивительному изобретению.

Как и в большинстве других наших примеров, в табл. 9.23 не указано какое-либо определенное кодирование состояний, и, подобно многим другим примерам, этот конечный автомат хорошо работает при записи состояний в форме выходного кода. Многие состояния можно отождествить с единственной комбинацией значений выходных сигналов, зажигающих огни светофора. Но существуют также три пары неразличимых состояний, если иметь в виду только световые сигналы: (NSWAIT, NSWAIT2), (EWWAIT, EWWAIT2) и (NSDELAY, EWDELAY). Это затруднение можно преодолеть, добавив еще одну переменную состояния "EXTRA" с различными значениями в состояниях, образующих каждую пару. Эта идея реализована в видоизмененной программе, приведенной в табл. 9.24.

Табл. 9.23. VHDL-программа для контроллера светофора в г. Саннивейл

```
library IEEE,
use IEEE.std_logic_1164.all,

entity Vsvale is
  port ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG  in  STD_LOGIC,
         OVERRIDE, FLASHCLK                           in  STD_LOGIC,
         NSRED, NSYELLOW, NSGREEN                     out STD_LOGIC,
         EWRED, EWYELLOW, EWGREEN, TMRESET            out STD_LOGIC ),
end,

architecture Vsvale_arch of vsvale is
type Sreg_type is (NSGO, NSWAIT, NSWAIT2, NSDELAY,
                   EWGO, EWWAIT, EWWAIT2, EWDELAY),
signal Sreg  Sreg type,
begin

process (CLOCK)
begin
  if CLOCK'event and CLOCK = '1' then
    if RESET = '1' then Sreg <= NSDELAY, else
      case Sreg is
        when NSGO =>                                          -- North-south green
          if    TMSHORT='0' then Sreg <= NSGO,                -- Minimum 5 seconds
          elsif TMLONG='1'  then Sreg <= NSWAIT,              -- Maximum 6 minutes
          elsif EWCAR='1' and NSCAR='0' then Sreg <= NSGO,    -- Make EW car wait
          elsif EWCAR='1' and NSCAR='1' then Sreg <= NSWAIT, -- Thrash if cars both ways
          elsif EWCAR='0' and NSCAR='1' then Sreg <= NSWAIT, -- New NS car? Make it stop!
          else                               Sreg <= NSGO,      No one coming, no change
          end if,
        when NSWAIT  => Sreg <= NSWAIT2;                      -- Yellow light,
        when NSWAIT2 => Sreg <= NSDELAY,                      -   two ticks for safety
        when NSDELAY => Sreg <= EWGO,                         -- Red both ways for safety
        when EWGO =>                                          -- East-west green
          if    TMSHORT='0' then Sreg <= EWGO,                -- Same behavior as above
          elsif TMLONG='1'  then Sreg <= EWWAIT,
          elsif NSCAR='1' and EWCAR='0' then Sreg <= EWGO,
          elsif NSCAR='1' and EWCAR='1' then Sreg <= EWWAIT,
          elsif NSCAR='0' and EWCAR='1' then Sreg <= EWWAIT,
          else                               Sreg <= EWGO,
          end if,
        when EWWAIT  => Sreg <= EWWAIT2,
        when EWWAIT2 => Sreg <= EWDELAY,
        when EWDELAY => Sreg <= NSGO,
        when others  => Sreg <= NSDELAY,                      -- 'Reset' state
      end case,
    end if,
  end if,
end process,

TMRESET  <= '1' when Sreg=NSWAIT2 or Sreg=EWWAIT2 else '0';
NSRED    <= FLASHCLK when OVERRIDE='1' else
            '1' when Sreg/=NSGO and Sreg/=NSWAIT and Sreg/=NSWAIT2 else '0',
NSYELLOW <= '0' when OVERRIDE='1' else
            '1' when Sreg=NSWAIT or Sreg=NSWAIT2 else '0';
NSGREEN  <= '0' when OVERRIDE='1' else '1' when Sreg=NSGO else '0',
EWRED    <= FLASHCLK when OVERRIDE='1' else
            '1' when Sreg/=EWGO and Sreg/=EWWAIT and Sreg/=EWWAIT2 else '0';
EWYELLOW <= '0' when OVERRIDE='1' else
            '1' when Sreg=EWWAIT or Sreg=EWWAIT2 else '0',
EWGREEN  <= '0' when OVERRIDE='1' else '1' when Sreg=EWGO else '0',

end Vsvale_arch;
```

Табл. 9.24. Определения для автомата, управляющего светофором в г. Саннивейл, при записи состояний в форме выходного кода

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vsvale is
  port ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG: in  STD_LOGIC;
         OVERRIDE, FLASHCLK:                          in  STD_LOGIC;
         NSRED, NSYELLOW, NSGREEN:                    out STD_LOGIC;
         EWRED, EWYELLOW, EWGREEN, TMRESET:           out STD_LOGIC );
end;

architecture Vsvaleoc_arch of Vsvale is
signal Sreg: STD_LOGIC_VECTOR (1 to 7);
-- bit positions of output-coded assignment:  (1) NSRED, (2) NSYELLOW, (3) NSGREEN,
--                                            (4) EWRED, (5) EWYELLOW, (6) EWGREEN, (7) EXTRA
constant NSGO:    STD_LOGIC_VECTOR (1 to 7) := "0011000";
constant NSWAIT:  STD_LOGIC_VECTOR (1 to 7) := "0101000";
constant NSWAIT2: STD_LOGIC_VECTOR (1 to 7) := "0101001";
constant NSDELAY: STD_LOGIC_VECTOR (1 to 7) := "1001000";
constant EWGO:    STD_LOGIC_VECTOR (1 to 7) := "1000010";
constant EWWAIT:  STD_LOGIC_VECTOR (1 to 7) := "1000100";
constant EWWAIT2: STD_LOGIC_VECTOR (1 to 7) := "1000101";
constant EWDELAY: STD_LOGIC_VECTOR (1 to 7) := "1001001";

begin

process (CLOCK)
...             (no change to process)
end process;

TMRESET  <= '1' when Sreg=NSWAIT2 or Sreg=EWWAIT2 else '0';
NSRED    <= Sreg(1);
NSYELLOW <= Sreg(2);
NSGREEN  <= Sreg(3);
EWRED    <= Sreg(4);
EWYELLOW <= Sreg(5);
EWGREEN  <= Sreg(6);

end Vsvaleoc_arch;
```

10.5. Интегральные схемы типа CPLD

С момента своего появления несколько лет назад программируемые логические устройства (ПЛУ) типа 16V8 и 22V10 стали основным, очень гибким инструментом цифрового проектирования. По мере развития технологии ИС, естественно, возрастал интерес к созданию ПЛУ с более развитой архитектурой, что позволило бы воспользоваться достоинствами повышенной плотности упаковки в кристалле. Вопрос заключается в следующем: почему производители не пошли по пути увеличения существующих устройств?

Плотность упаковки в динамических ОЗУ, например, увеличилась за последние 10 лет в 64 раза. Почему производители ПЛУ не смогли перейти от ИС 16V8 к выпуску более крупных ИС "128V64"? Такое устройство имело бы 64 входных

вывода, 64 I/O-вывода и некоторое количество термов-произведений (скажем, 8) со 128-переменными для каждой из 128 своих логических макроячеек. В таком устройстве можно было бы объединить функции, реализуемые большим числом ИС 16V8, и предложить потрясающую эффективность и гибкость функционального использования любых входных и выходных сигналов.

Было ли это возможно? Да, микросхема 128V64 была бы очень гибким компонентом, но она не обладала бы очень хорошими характеристиками. В отличие от ИС 16V8, у которой на каждый элемент И приходится 32 входа (16 сигналов и 16 их дополнений), в устройстве 128V64 у каждого элемента И было бы 256 входов. Из-за влияния емкостей, токов утечки и по другим причинам такая большая структура монтажного И была бы, по крайней мере, в восемь раз медленнее, чем матрица И в ИС 16V8.

С точки зрения производителя ситуация выглядит еще хуже: в микросхеме 128V64 экономически очень не эффективно использовалась бы площадь кристалла. Потребовался бы кристалл с площадью примерно в 64 раза больше, чем для ИС 16V8, при возможном числе входов и выходов всего лишь в восемь раз большем, чем у ИС 16V8. Другими словами, для n-кратного увеличения логики (в терминах числа входов, выходов и элементов И) микросхеме 128V64 понадобился бы кристалл с площадью, в n^2 раз большей. С точки зрения эффективного использования площади кристалла умный разработчик выиграл бы, разбив желаемую функцию так, чтобы реализовать ее на восьми ИС 16V8, если только такое разбиение возможно.

Эта идея была использована в *сложных программируемых логических устройствах* (*complex programmable logic device, CPLD*). Как показано на рис. 10.37, ИС типа CPLD является всего лишь совокупностью отдельных ПЛУ на одном кристалле с программируемой структурой взаимных связей, которая позволяет отдельным ПЛУ в пределах кристалла подключаться друг к другу так же, как это мог бы сделать талантливый разработчик с отдельными ПЛУ вне кристалла. Здесь площадь кристалла, необходимая для реализации увеличенной в n раз логики, равна площади только n одиночных ПЛУ плюс площадь, занимаемая программируемой структурой взаимных связей.

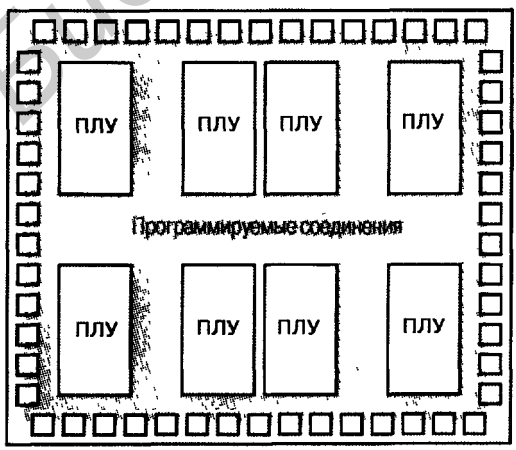


Рис. 10.37. Общая архитектура ИС типа CPLD

Разные производители перепробовали много различных вариантов общей архитектуры, показанной на рисунке. Эти варианты отличаются блоками ПЛУ (состоящими из матрицы И и макроячеек), блоками ввода/вывода и структурой программируемых соединений. В данном параграфе мы рассмотрим каждую из этих составляющих, воспользовавшись в качестве примера архитектурой ИС типа CPLD серии 9500 фирмы Xilinx.

10.5.1. Семейство ИС XC9500 фирмы Xilinx

Микросхемы XC9500 фирмы Xilinx представляют собой семейство ИС типа CPLD одинаковой архитектуры, но с различным числом внешних I/O-выводов и с разным числом внутренних ПЛУ, которые фирма Xilinx называет *функциональными блоками* (*functional blocks*, *FBs*). Как мы увидим позже, у каждого внутреннего ПЛУ 36 входов, оно содержит 18 макроячеек и имеет 18 выходов; такое ПЛУ можно было бы назвать "36V18". Как следует из табл. 10.8, маркировка микросхем, определяется числом имеющихся в них макроячеек. Самый маленький представитель семейства содержит 2 функциональных блока с 36 макроячейками, а самый большой – 16 функциональных блоков с 288 макроячейками.

Табл. 10.8. Функциональные блоки и внешние I/O-выводы микросхем типа CPLD серии 9500 фирмы Xilinx

	Номер микросхемы					
	XC9536	***XC9572***	***XC95108***	***XC95144***	***XC95216***	***XC95288***
Число функциональных блоков и число макроячеек:	2/36	4/72	6/108	8/144	12/216	16/288
Тип корпуса	***Число I/O-выводов***					
VQFP с 44 выводами	34					
PLCC с 44 выводами	34	34				
CSP с 48 выводами	34					
PLCC с 84 выводами		69	69			
TQFP с 100 выводами		72	81	81		
PQFP с 100 выводами		72	81	81		
PQFP с 160 выводами			108	133	133	
HQFP с 208 выводами					166	168
BGA с 352 выводами					166	192

Другой важной особенностью этого семейства и большинства других семейств ИС типа CPLD является то, что одна и та же микросхема, скажем XC95108, выпускается в нескольких различных корпусах. Это существенно не только с точки зрения удовлетворения требований, предъявляемых различными технологиями производства, но также и для обеспечения определенного выбора и возможности сэкономить на числе внешних I/O-выводов. В большинстве случаев не требуется, чтобы все внутренние сигналы конечного автомата или подсистемы были видимы остальной частью системы и использовались ею.

Так, ИС XC95108 содержит 108 внутренних макроячеек, но при ее размещении в корпусе типа PLCC с 84 выводами наружу могут быть выведены выходы самое большее 69 макроячеек. На самом деле, как правило, большинство из 69 I/O-выводов используются как входы, поэтому извне будет доступно еще меньшее число выходов. И это правильно: остальные выходы макроячеек вполне можно использовать внутри, так как к ним можно подключиться внутри через структуру программируемых соединений. Макроячейки, выходы которых доступны только внутри, иногда называют *скрытыми макроячейками* (*buried macrocells*).

Еще одним важным обстоятельством является то, что в одной строке в табл. 10.8 перечислены несколько микросхем. Оказывается, что в одинаковых корпусах любого типа, кроме двух, могут быть размещены, по крайней мере, два различных устройства с совместимыми выводами. Это значительно облегчает жизнь при изменении проекта в последнюю минуту. Предположим, например, что при проектировании вы выбрали ИС XC9572 в корпусе PLCC с 84 выводами. Возможно вы считаете, что 69 I/O-выводов, имеющихся у этой микросхемы, вполне достаточно. Вы хотели бы воспользоваться ИС XC9572 из-за ее низкой стоимости. Но если в вашем начальном проекте используются 68 из 72 макроячеек имеющихся внутри данной ИС, то это должно вызвать у вас определенную тревогу (со мной было бы именно так!). Глядя в табл. 10.8, можно быть спокойным, зная, что если обнаружатся ошибки или изменятся технические требования к проекту и потребуется более сложная внутренняя структура, то всегда можно перейти к ИС XC95108 в том же самом корпусе и воспользоваться еще 36 макроячейками.

На рис. 10.38 приведена блок-схема внутренней архитектуры типичной ИС типа CPLD из семейства XC9500. Ниже объясняется, что каждый внешний I/O-вывод можно использовать в качестве входа, выхода или двунаправленного вывода в соответствии с тем, как запрограммировано устройство. Выводы, расположенные в нижней части рисунка, можно использовать также для тех или иных специальных целей. На любой из трех выводов GCK можно подавать «общие тактовые сигналы»; как мы увидим позже, каждую макроячейку можно запрограммировать так, чтобы на нее поступал тактовый сигнал с выбранного входа. Один вывод GSR можно использовать для подачи сигнала «общая установка/сброс»; снова, каждую макроячейку можно запрограммировать так, чтобы с помощью этого сигнала производилась асинхронная предварительная установка или сброс. Наконец, на любой из двух или из четырех выводов GTS (в зависимости от типа устройства) можно подавать сигнал, осуществляющий «общее управление третьим состоянием»; в каждой макроячейке можно выбрать один из этих сигналов для отпирания или запирания соответствующего выхода, когда выход макроячейки подключен к внешнему I/O-выводу.

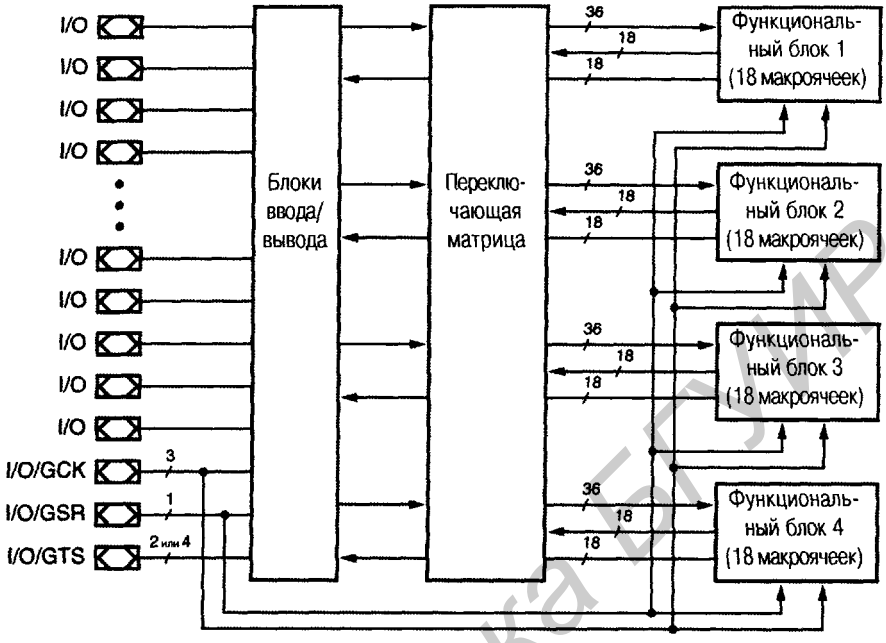


Рис. 10.38. Архитектура ИС типа CPLD семейства 9500 фирмы Xilinx

На рисунке показаны только четыре функциональных блока, но архитектура семейства XC9500 допускает наличие в ИС XC95288 16 функциональных блоков. Независимо от особенностей микросхемы, входящей в состав этого семейства, на входы каждого функционального блока путем программирования переключающей матрицы подаются 36 сигналов. На входы переключающей матрицы поступают сигналы с 18 выходов макроячеек от каждого функционального блока и внешние входные сигналы с I/O-выводов. Более подробно о том, как осуществляется коммутация в переключающей матрице, говорится в разделе 10.5.4.

Кроме того, у каждого функционального блока есть 18 выходов, сигналы на которых проходят «мимо» переключающей матрицы, как показано на рис. 10.38, и поступают на блоки ввода/вывода. Это просто сигналы разрешения выхода для выходных каскадов блока ввода/вывода; эти сигналы действуют в том случае, когда выход макроячейки данного функционального блока подключен к внешнему I/O-выводу.

10.5.2. Архитектура функционального блока

Архитектура функционального блока семейства XC9500 приведена на рис. 10.39. В программируемой матрице И имеется только 90 термов-произведений. По сравнению с такими ПЛУ как 16V8 и 22V10 у ИС типа XC9500 и у большинства других ИС типа CPLD на одну макроячейку приходится меньшее число И-термов: у ИС XC9500 их всего лишь 5, в то время как у микросхемы 16V8 – 8, а у микросхемы 22V10 – от 8 до16. Однако все не так плохо благодаря возможности *распределения термов-произведений* (*product-term allocation*). У микросхем серии

XC9500, как и у других ИС типа CPLD, имеются *распределители термов-произведений* (*product-term allocators*), поэтому термы-произведения, не востребованные в одной макроячейке, можно использовать в других, соседних макроячейках того же функционального блока.

Рис. 10.39. Архитектура функционального блока

На рис. 10.40 представлена принципиальная схема распределителя термов-произведений и макроячейки ИС серии XC9500. На этом рисунке прямоугольниками с именами S1–S8 обозначены программируемые элементы, посредством которых сигналы, действующие на их входах, направляются на один из имеющихся у них выходов. Трапециевидные символы, обозначенные M1–M5, представляют собой программируемые мультиплексоры, которые подключают один из двух или четырех имеющихся у них входов к своему выходу.

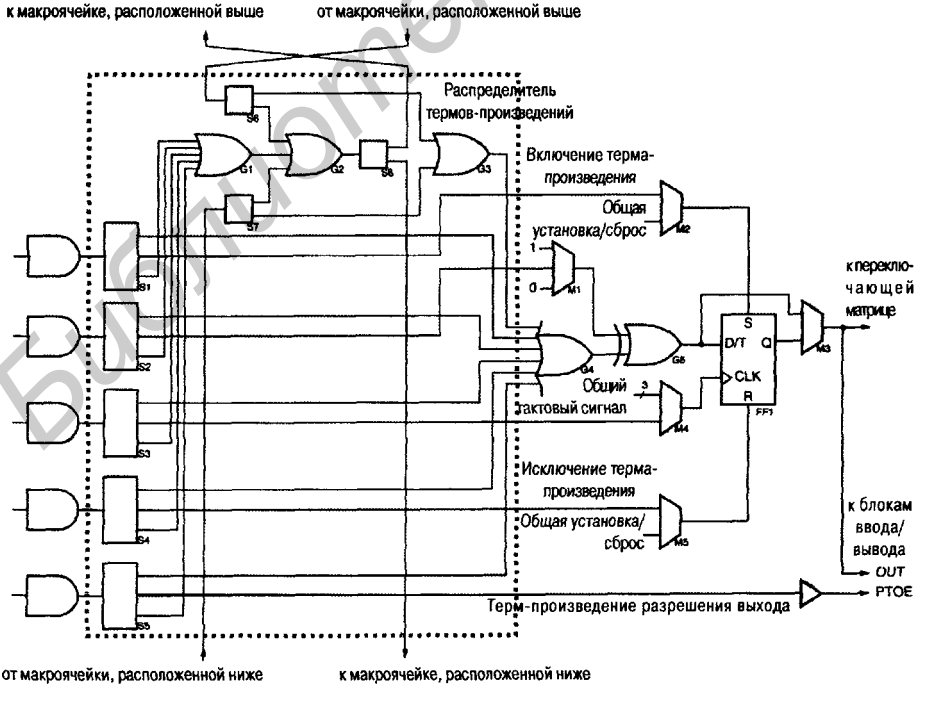


Рис. 10.40. Распределитель термов-произведений и макроячейка ИС серии XC9500

В левой части рисунка изображены 5 вентилей И, относящихся к данной макроячейке. Выходы каждого из них соединены с входами элементов, осуществляющих направление сигнала по тому или иному пути. С верхних выходов этих элементов термы-произведения поступают на вентиль G4 – главный вентиль ИЛИ данной макроячейки. Из сказанного можно заключить, что данной макроячейке доступны всего лишь пять термов-произведений. Однако верхний, шестой вход вентиля G4 соединен с выходом другого вентиля ИЛИ (G3), на который поступают термы-произведения от макроячеек, расположенных выше и ниже данной макроячейки.

Любые не используемые в данной макроячейке термы-произведения можно с помощью направляющих узлов S1–S5 подать на входы объединяющего вентиля ИЛИ (G1), сигнал с выхода которого через элемент S8, в конце концов, может быть отправлен в макроячейку, расположенную выше, или в макроячейку, расположенную ниже. Перед направлением в другую макроячейку эти термы-произведения можно с помощью элементов S6, S7 и G2 объединить с термами-произведениями макроячеек, расположенных выше или ниже данной макроячейки. Таким образом, возможно «гирляндное подключение» термов-произведений через следующие одна за другой макроячейки для образования суммы, состоящей из большего числа произведений. В принципе, можно объединить и направить в одну макроячейку все 90 термов-произведений, имеющихся в данном функциональном блоке, хотя при этом 17 из 18 макроячеек этого функционального блока останутся вообще без термов-произведений.

За передачу термов-произведений по гирляндной цепочке соединений приходится платить не только тем, что другие макроячейки лишаются своих термов-произведений. При каждой «пересылке» терма-произведения вносится небольшая дополнительная задержка, которую можно минимизировать аккуратным размещением макроячеек, испытывающих недостаток термов-произведений, так чтобы они оказались соседними с макроячейками, в которых используется мало термов-произведений. Например, в макроячейке можно использовать 13 термов-произведений с задержкой, вносимой только одной дополнительной пересылкой, при условии, что эта макроячейка расположена между двумя макроячейками, в которых задействовано лишь по одному терму-произведению.

ДВИЖЕНИЕ В ОДНУ СТОРОНУ

При программировании данной макроячейки ИС типа XC9500 обычно не отправляют термы-произведения по гирляндной цепочке соединений назад, то есть в том направлении, откуда они поступают. Например, если терм-произведение попадает на вход элемента S6 сверху, то мы можем использовать его на месте, направляя сигнал с входа S6 к вентилю G3, или передать его на вентиль G2. В последнем случае элемент S8 должен направить сигнал с выхода G2 к макроячейке, расположенной ниже; снова направлять терм-произведение в верхнюю макроячейку нет никакого смысла. Если бы элементы S6 и S8 данной макроячейки отправляли терм-произведение вверх, а элементы S7 и S8 верхней макроячейки направляли бы его вниз, то у нас возникло бы нежелательное зацикливание.

Третий вариант, когда сигнал появляется на среднем выходе какого либо из элементов S1–S5, служит для использования терма-произведения в качестве «специальной функции». Специальные функции – это подача сигнала на тактовый вход триггера, установка его в единичное состояние и сброс, а также управление вентилем ИСКЛЮЧАЮЩЕЕ ИЛИ и разрешение выхода. Обычно большинство этих специальных функций не используется.

Сердцевину макроячейки образует вентиль ИЛИ G4, на выходе которого возникает сумма всех выбранных термов-произведений, и вентиль ИСКЛЮЧАЮЩЕЕ ИЛИ G5, на один из входов которого подается эта сумма произведений. Сигнал на другом входе вентиля G5 может быть равен 0 или 1, а также может быть термом-произведением в зависимости от того, что выбрано мультиплексором M1. При установке на этом входе вентиля G5 единицы, сумма произведений, поступающая с выхода вентиля G4, инвертируется, поэтому макроячейку можно сконфигурировать так, чтобы получить минимизированные логические выражения любой полярности. Подача на этот вход терма-произведения полезна при построении счетчиков. Если терм-произведение принимает значение 1 в том случае, когда биты младших разрядов счетчика равны 1 и счет разрешен, а сигнал на выходе вентиля G4 выражает собой текущее значение бита в данном разряде счетчика, то бит в данном разряде счетчика инвертируется, как и должно происходить в счетчике.

Триггер макроячейки FF1 можно запрограммировать для работы в качестве D-триггера или в качестве T-триггера с входом разрешения счета; последний вариант полезен при реализации счетчиков того или иного типа. С помощью мультиплексора M4 выбирается сигнал, подаваемый на тактовый вход триггера; этим сигналом может быть один из четырех входных сигналов мультиплексора: один из трех общих тактовых сигналов на входах ИС или терм-произведение. На выбор последнего сигнала в синхронных проектах наложен запрет, за исключением тщательно проработанных синхронизирующих устройств типа схемы, приведенной на рис. 8.111.

У триггера есть также входы асинхронной установки в единичное состояние и сброса. Выбор сигналов, подаваемых на эти входы, осуществляется мультиплексорами M2 и M5. В большинстве случаев входы установки и сброса бывают соединены с общим входом установка/сброс данной ИС и используются только при начальном запуске системы. Однако по этим входам можно также получить доступ к SR-защелке, у которой вход CLK не используется. Кроме того, тактовым входом CLK и входами S или R можно воспользоваться в синхронизирующих устройствах так, как это сделано в схеме на рис. 8.111, но при этом нужно быть очень внимательным.

В качестве выходного сигнала макроячейки OUT с помощью еще одного мультиплексора M3 выбирается сигнал с выхода триггера или сигнал, поступающий на его вход данных. Выходной сигнал OUT поступает на переключающую матрицу, где он может быть использован любой другой макроячейкой. Он может быть отправлен также к блокам ввода/вывода вместе с термом-произведением, выбранным элементом S5, который при необходимости, можно использовать как сигнал разрешения выхода PTOE.

10.5.3. Архитектура блока ввода/вывода

Структура *блока ввода/вывода* (*I/O block*, *IOB*) в ИС семейства XC9500 показана на рис. 10.41. Имеются семь вариантов выбора сигнала разрешения выхода для выходного буфера с тремя состояниями. Буфер может быть всегда открытым, всегда запертым, его состояние может определяться термом-произведением PTOE, поступающим от соответствующей макроячейки, или любым из четырех сигналов общего разрешения выхода. Сигналы общего разрешения выхода могут иметь как высокий активный уровень, так и низкий активный уровень, в зависимости от сигналов на внешних выводах GTS.

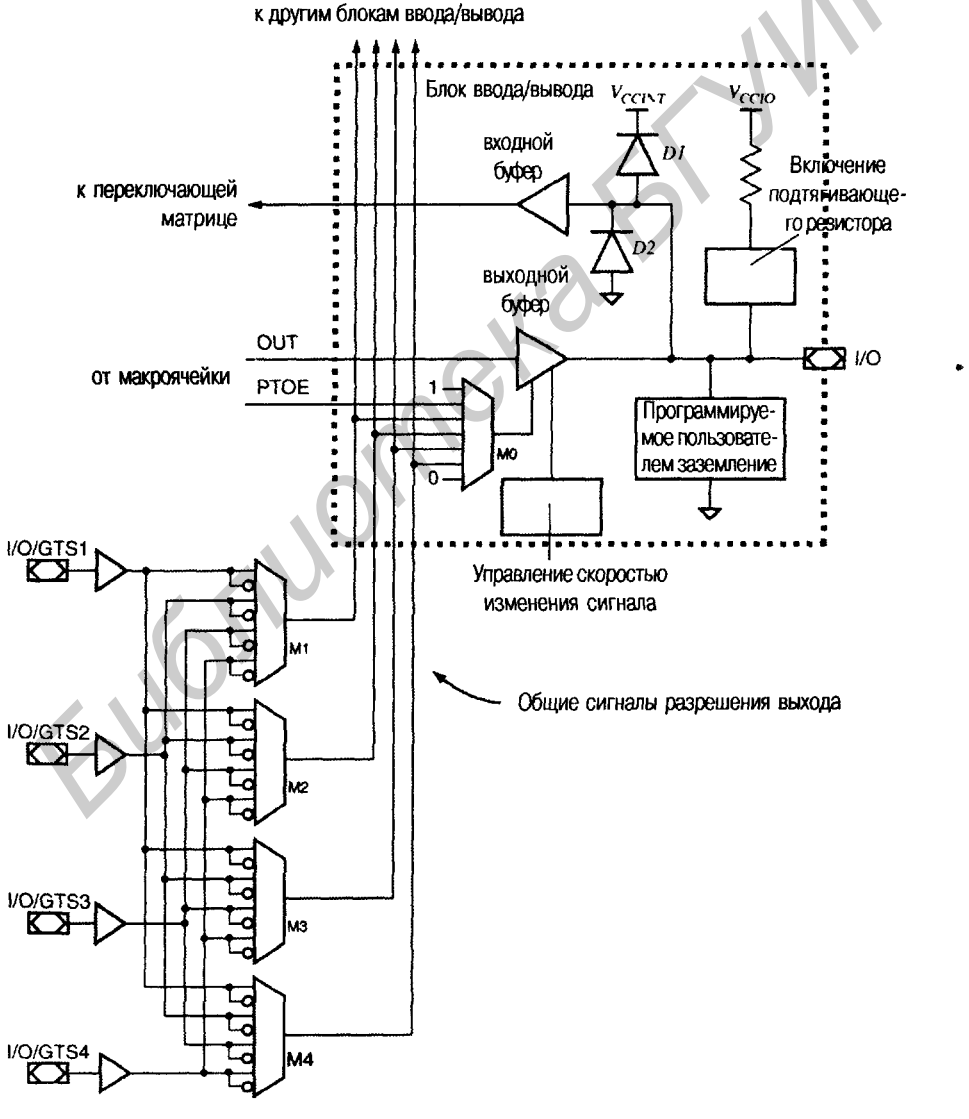


Рис. 10.41. Блок ввода/вывода микросхем серии XC9500

Блок ввода/вывода микросхем семейства XC9500 является хорошим примером важной тенденции в архитектуре блоков ввода/вывода микросхем типа CPLD и FPGA: в нем, кроме управления «логическими» действиями вроде разрешения выхода, имеется возможность изменять многие «аналоговые» параметры. В блоке ввода/вывода можно изменять значения трех аналоговых параметров:

- *Управление скоростью изменения выходного напряжения.* Для того чтобы получать быстродействующее или медленно работающее устройство, можно задавать время нарастания и спада выходных сигналов. Установка наибольшего быстродействия обеспечивает наименьшую возможную задержку распространения, в то время как задание режима медленной работы устройства позволяет уменьшить «звон» в линии передачи и шумы в системе за счет небольшой дополнительной задержки.
- *Включение резистора нагрузки между выходом и шиной питания.* Когда резистор нагрузки включен, он предотвращает появление на выходном выводе плавающего напряжения при подаче на микросхему напряжения питания. Это полезно в том случае, когда выходные сигналы поступают на входы разрешения других логических устройств с низким активным уровнем, в отношении которых не предполагается, что в момент включения питания на них будет подан сигнал, имеющий активное значение.
- *Образование программируемых пользователем выводов земли.* Эта возможность фактически позволяет перераспределять I/O-выводы так, чтобы те или иные выводы были выводами земли, а вовсе не сигнальными выводами. Это оказывается полезным в быстродействующих устройствах с высокой скоростью изменения сигналов. Необходимость в дополнительных выводах земли возникает в тех случаях, когда имеют место большие броски тока, возникающие при одновременном переключении сигналов на нескольких выходах.

Кроме этих особенностей, ИС семейства XC9500 совместимы с внешними устройствами с напряжением питания 5 В и 3.3 В. Входной буфер и внутренняя логика работают от источника питания с напряжением V_{CCINT}, равным 5 вольтам. В зависимости от напряжения питания внешних устройств, в выходном каскаде используется напряжение питания V_{CCIO}, равное 5 В или 3.3 В. Обратите внимание, что включение резистора между выходом и шиной питания подтягивает напряжение на выходе до напряжения питания блока I/O, то есть до напряжения V_{CCIO}. Диоды *D1* и *D2* необходимы для фиксации выходного напряжения на уровне не выше значения V_{CCINT} и не ниже уровня земли. За эти границы величина выходного напряжения могла бы выходить из-за «звона» в линии передачи.

10.5.4. Переключающая матрица

Теоретически программируемая структура соединений внутри ИС типа CPLD должна допускать соединение любого внутреннего выхода ПЛУ и любого внешнего входа с любым внутренним входом ПЛУ. Аналогично, должно быть возможным подключение любого выхода любого из внутренних ПЛУ к любому внешнему выводу. Но если присмотреться внимательнее, то становится ясно, что мы возвращаемся к той же самой «проблеме n^2», которая возникала бы при попытке создания ИС 128V64.

Рис. 10.42 служит иллюстрацией требований, предъявляемых к переключающей матрице, на примере ИС XC95108 – типичного представителя семейства XC9500 фирмы Xilinx. У этой микросхемы имеются 108 выходов внутренних макроячеек и 108 внешних входных выводов, так что полное число сигналов, которые должны быть поданы на переключающую матрицу в качестве входных, составляет 216. Так как в ИС XC95108 имеется 6 функциональных блоков с 36 входами каждый, то переключающая матрица должна иметь в случае наибольшей полноты 216 мультиплексоров с 216 входами каждый, так чтобы сигнал с выхода каждого мультиплексора поступал на один из входов матрицы И одного из функциональных блоков.

Переключающая матрица, показанная на рисунке, может быть выполнена в микросхеме в виде прямоугольной структуры, у которой входы расположены по столбцам, а выходы – по строкам, с проходным транзистором (или логическим ключом) в каждой точке пересечения, позволяющим соединить данный вход с соответствующим выходом. Приведенный пример показывает, что это все же слишком большая структура: 216 строк и 216 столбцов!

При современной технологии создания ИС с высокой плотностью размещения компонентов проблема состоит не столько в размерах кристалла, сколько в быстродействии устройства. Большое число транзисторов, включенных в каждой строке и в каждом столбце, обуславливает наличие значительной емкости, которая приводит к уменьшению быстродействия. Поэтому производители ИС типа CPLD ищут способы уменьшить размер переключающей матрицы.

Глядя на рис. 10.42, можно увидеть, например, что не для каждого входа переключающей матрицы необходимо предусмотреть возможность его подключения к каждому выходу. Нужно лишь, чтобы каждый вход мог быть подключен к *какому-либо* входу каждого функционального блока, поскольку любой вход функционального блока может быть соединен с любым вентилем И матрицы И данного функционального блока.

Но снова задача совсем не так проста, как мы только что ее сформулировали. Предположим, что сигнал на выходе переключающей матрицы для i-го входа рассматриваемого функционального блока ($0 < i < 35$) вырабатывается простым 8-входовым мультиплексором, входы которого являются входами переключающей матрицы с номерами от $8i$ до $8i + 7$. С помощью такой переключающей матрицы нельзя осуществить многие из вариантов межсхемных соединений. Например, подключение входов переключающей матрицы с номерами от 0 до 35 к тому же самому функционального блоку было бы невозможно. Как только вход 0 переключающей матрицы соединяется с входом 0 одного их функциональных блоков, входами переключающей матрицы с номерами от 1 до 7 уже нельзя воспользоваться.

Таким образом, требования, предъявляемые к возможности осуществления с помощью переключающей матрицы тех или иных соединений, следует сформулировать более широко: для каждого функционального блока должно быть возможным подключение любой комбинации входов переключающей матрицы к какой-либо комбинации входов данного функционального блока.

Переключающая матрица типичной ИС типа CPLD представляет собой компромисс между минимальной схемой мультиплексора и полной, неблокирующей структурой коммутационной матрицы. Если возможности переключающей мат-

рицы меньше, чем у неблокирующей матрицы, то проблема распределения соединений вход-выход в ней становится нетривиальной. При проектировании различных устройств на основе ИС типа CPLD каждый раз необходимо найти соответствующий набор связей, реализуемых переключающей матрицей; это делается с помощью «программы компоновки», которой производитель ИС типа CPLD сопровождает свою продукцию.

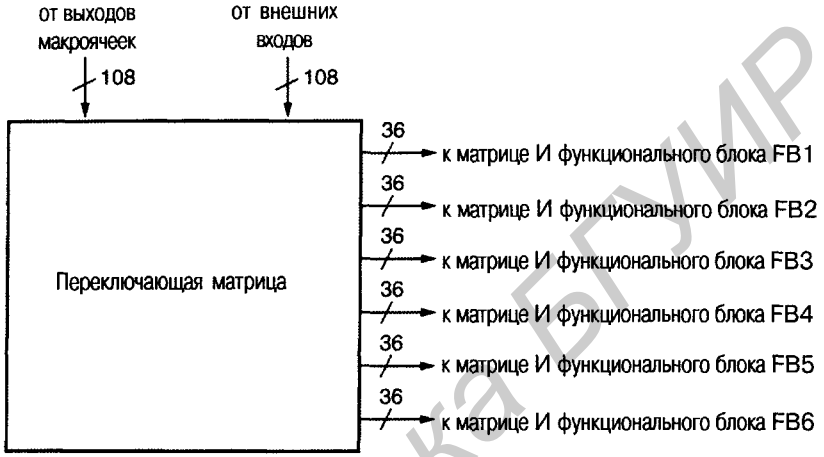


Рис. 10.42. Требования, предъявляемые к переключающей матрице ИС XC95108

Нахождение всех возможных комбинаций входов и выходов при разреженной переключающей матрице является одной из тех *NP*-полных задач, о которых вы, по-видимому, слышали в информатике. На практике это означает, что применительно к некоторым проектам, программе компоновки, вероятно, придется работать намного дольше, чем вам хотелось бы, чтобы узнать существует ли решение. Если в переключающей матрице слишком мало точек пересечения, как в нашем примере с очень маленькими мультиплексорами, то даже лучшая из программ компоновки при «сколь угодно долгой» работе в целом ряде случаев не сможет осуществить полный перебор всех возможных соединений.

Таким образом, структура переключающей матрицы ИС типа CPLD – это компромисс между характеристиками микросхемы (быстродействие, площадь кристалла, стоимость) и возможностями программы компоновки. Программа компоновки обычно не только устанавливает соединения в переключающей матрице, но также производит назначение входов и выходов функциональных блоков и макроячеек и их привязку к внешним выводам микросхемы, а также задает «внутреннюю логику» функциональных блоков и макроячеек. Эти назначения, в свою очередь, оказывают влияние на реализацию соединений внутри переключающей матрицы и на распределение термов-произведений. Решение этих проблем является «секретным ноу-хау» производителей ИС типа CPLD и разработчиков программного обеспечения и, как правило, ими не раскрывается.

ПРИВЯЗКА ВЫВОДОВ

Другой важной проблемой при разработке ИС типа CPLD и программного обеспечения является *привязка выводов* (*pin locking*). В большинстве приложений ИС типа CPLD считается нормальным разрешить программе компоновки выбирать любые возможные выводы в качестве внешних входов и выходов данного устройства. Но когда проект закончен и изготовлена печатная плата, разработчик может пожелать «зафиксировать» назначение выводов, так чтобы они оставались теми же самыми при небольших (или даже при больших!) изменениях, связанных с исправлением ошибок в проекте. Это приводит к экономии времени и стоимости и позволяет преодолеть препятствия, возникающие при переработке или доработке и переделке печатной платы.

Желаемая «привязка» выводов обычно указывается в файле, который читается программой компоновки. У первых ИС типа CPLD и FPGA привязка выводов до выполнения даже небольших изменений не гарантировала успеха: программа компоновки «поднимала руки» и жаловалась, что слишком много ограничений. Если вы разблокируете назначение выводов, то программа компоновки, возможно, найдет новое распределение, которое будет работать, но оно может оказаться совершенно не похожим на исходное.

Эти проблемы вовсе не обязательно возникали по вине программы компоновки; просто у ИС типа CPLD и FPGA не было достаточного количества внутренних связей, чтобы выдерживать постоянные изменения проекта при условии привязки выводов. Производители извлекли урок из опыта работы с этими первыми устройствами и усовершенствовали их внутреннюю архитектуру, приспособив ее к частым изменениям в процессе разработки проекта. В результате некоторые микросхемы теперь имеют, например, «выходную переключающую матрицу», которая гарантирует возможность соединения любого входа или выхода макроячейки, находящейся внутри данной ИС, с любым внешним I/O-выводом.

10.6. Интегральные схемы типа FPGA

Программируемая в условиях эксплуатации матрица вентилей (*field-programmable gate array, FPGA*) в какой-то мере подобна CPLD, вывернутой изнутри наружу. Как показано на рис. 10.43, на кристалле расположено большое число программируемых логических блоков, каждый из которых меньше, чем ПЛУ. Они распределены по всему кристаллу среди программируемых соединений, а вся матрица окружена программируемыми блоками ввода/вывода. Программируемый логический блок ИС типа FPGA обладает меньшими возможностями, чем типичное ПЛУ, но одна микросхема типа FPGA содержит гораздо больше логических блоков, чем ИС типа CPLD при том же самом размере кристалла.

Микросхемы типа FPGA были изобретены фирмой Xilinx, Inc., и в этом параграфе для иллюстрации архитектуры ИС типа FPGA мы воспользуемся одним из популярных семейств этой фирмы – семейством XC4000E.

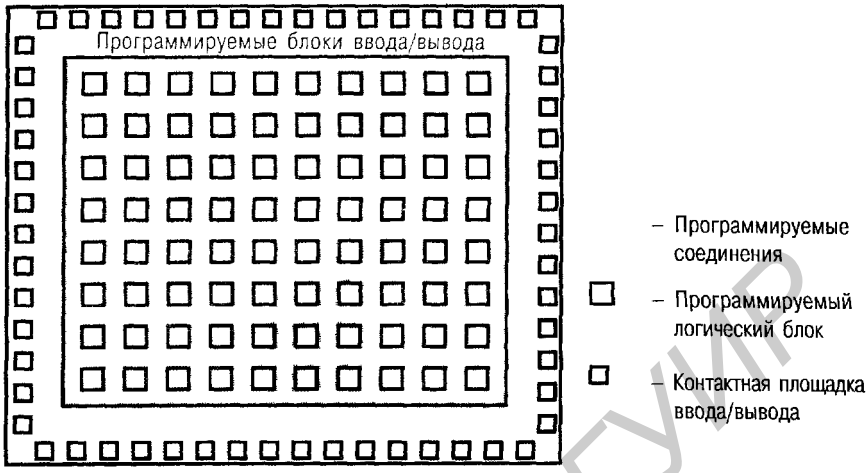


Рис. 10.43. Общая архитектура кристалла ИС типа FPGA

10.6.1. Семейство ИС типа FPGA XC4000 фирмы Xilinx

Программируемые логические блоки в ИС типа FPGA семейства XC4000E фирмы Xilinx названы *перестраиваемыми логическими блоками* (*configurable logic blocks*, *CLBs*). Наименьшая микросхема XC4003E содержит матрицу логических блоков размером 10×10, а в самой большой ИС XC4025E – 1024 логических блока в виде матрицы размером 32×32. Фирмой Xilinx на основе семейства XC4000E созданы также расширенные семейства XC4000EX и XC4000XL, у которых имеются дополнительные возможности и которые обладают не рассматриваемыми здесь свойствами. Самая большая ИС в расширенных семействах XC4085XL содержат 3136 логических блоков. В табл. 10.9 приведены о данные о выпускавшихся в 1999 году фирмой Xilinx микросхемах семейства XC4000.

Подобно семейству CPLD, семейство XC4000 включает набор микросхем разных размеров с разным числом I/O-выводов. В столбце таблицы «Макс. число доступных входов/выходов» указано максимальное число доступных пользователю блоков ввода/вывода, имеющихся в микросхеме. Однако ИС серии XC4000 размещают в различных корпусах и в случае корпуса меньших размеров не все имеющиеся входы/выходы выведены на внешние контакты. Как и в случае с ИС типа CPLD, пользователь микросхем типа FPGA имеет возможность перенести свой проект, выполненный на основе небольшой ИС, в микросхему большего объема, размещенную в том же самом корпусе, и наоборот.

В столбце «Число триггеров», как мы увидим позже, учтены все триггеры в устройстве, по два в каждом логическом блоке и по два в каждом блоке ввода/вывода. В типичном проекте используется только часть имеющихся триггеров, но полное их число является показателем возможностей, на которые ориентируется разработчик при грубой оценке размеров требуемой ИС типа FPGA. Данные столбца «Максимальное число битов в ОЗУ» являются другим показателем качества. Как мы увидим, каждый логический блок может либо выполнять логические операции, либо представлять собой небольшое статическое ОЗУ, хранящее до 32 битов.

Табл. 10.9. Микросхемы типа FPGA серии XC4000 фирмы Xilinx

Название микросхемы	Размер матрицы логических блоков	Число логических блоков	Макс. число доступных входов/выходов	Число триггеров	Макс. число битов в ОЗУ (без логики)	Макс. число вентилей (без ОЗУ)	Типичное число вентилей (в логике и в ОЗУ)
XC4002XL	8×8	64	64	256	2 048	1 600	1 000–3 000
XC4003E	10×10	100	80	360	3 200	3 000	2 000–5 000
XC4005E/XL	14×14	196	112	616	6 272	5 000	3 000–9 000
XC4006E	16×16	256	128	768	8 192	6 000	4 000–12 000
XC4008E	18×18	324	144	936	10 368	8 000	7 000–15 000
XC4010E/XL	20×20	400	160	1 120	12 800	10 000	7 000–20 000
XC4013E/XL	24×24	576	192	1 536	18 432	13 000	10 000–30 000
XC4020E/XL	28×28	784	224	2 016	25 088	20 000	13 000–40 000
XC4025E	32×32	1 024	256	2 560	32 768	25 000	15 000–45 000
XC4028EX/XL	32×32	1 024	256	2 560	32 768	28 000	18 000–50 000
XC4036EX/XL	36×36	1 296	288	3 168	41 472	36 000	22 000–65 000
XC4044XL	40×40	1 600	320	3 840	51 200	44 000	27 000–80 000
XC4052XL	44×44	1 936	352	4 576	61 952	52 000	33 000–100 000
XC4062XL	48×48	2 304	384	5 376	73 728	62 000	40 000–130 000
XC4085XL	56×56	3 136	448	7 168	100 352	85 000	55 000–180 000

У параметра «Максимальное число вентилей» нет четкого определения. Согласно таблице каждый логический блок в ИС XC4002XL может выполнять функцию, реализуемую примерно 25 вентилями в дискретном исполнении. Микросхема XC4003E в этом отношении даже лучше: в ней каждый логический блок соответствует 30 вентилям. Так ли это? И еще: что такое «вентиль»? Считать ли схему ИСКЛЮЧАЮЩЕЕ ИЛИ или схему И-НЕ с большим числом входов одним вентилем? Вы можете решить это для себя позже, после того, как больше узнаете об архитектуре логического блока. Тогда же вы сможете решить, кем был составлен этот столбец, – разработчиками или людьми, занимающимися маркетингом.

Последний столбец таблицы, скорее всего, написан специалистами по маркетингу, так как верхняя граница каждого «типичного» диапазона для числа вентилей превышает максимальное значение в предыдущем столбце! На самом деле *имеется* приемлемое объяснение этого кажущегося противоречия. Данные этого столбца предполагают, что 20–30% логических блоков используется в качестве статических ОЗУ емкостью 32 бита каждое, а не в качестве логики. Для образования ячейки статического ОЗУ необходимо, как минимум, четыре вентиля, когда роль ячейки ОЗУ играет D-защелка (см. схему на рис. X7.27), и поэтому можно считать, что каждый логический блок, используемый в качестве статического ОЗУ, эквивалентен 128 вентилям. Таким образом, величина в последнем столбце равна числу вентилей, реализующих логические функции, *плюс* число эквивалентных логических схем, на которых построены статические ОЗУ.

10.6.2. Перестраиваемый логический блок

Так как ИС типа FPGA может содержать громадное число логических блоков, важно прежде всего разобраться в них! На рис. 10.44 показана внутренняя структура логического блока в микросхемах серии XC4000.

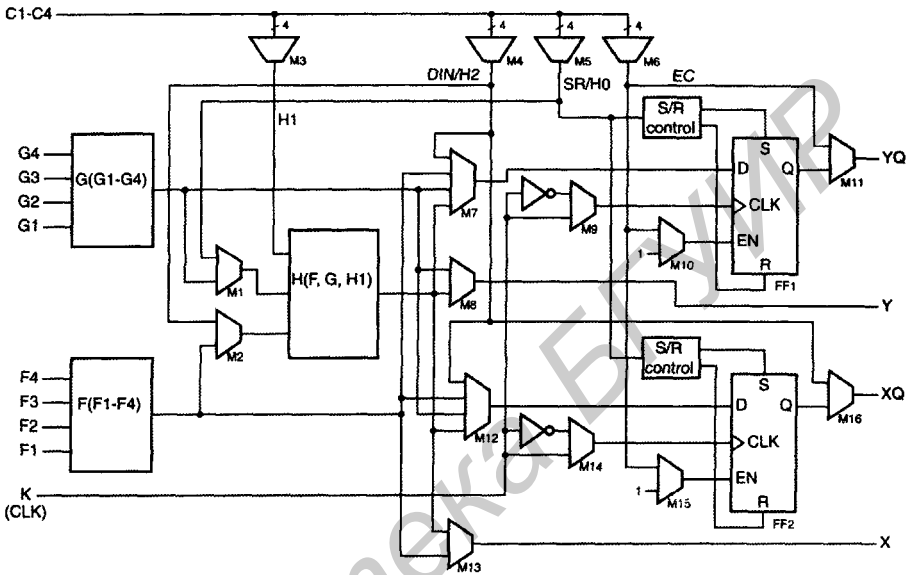


Рис. 10.44. Перестраиваемый логический блок микросхем семейства XC4000

Наиболее важными программируемыми элементами логического блока являются схемы F, G и H, вырабатывающие значения логических функций. С помощью элементов F и G можно реализовать любую комбинационную логическую функцию четырех переменных, а элемент H позволяет сформировать значение любой комбинационной логической функции трех переменных.

Как работают схемы F, G и H? Сколько вентилей вы затратили бы на построение «универсальной» логической схемы, реализующей функцию 4 переменных? Подумайте об этом, а мы возвратимся к этому позже.

Как и при рассмотрении структуры ИС типа CPLD, трапециевидные символы на рис. 10.44 представляют собой программируемые мультиплексоры. Обратите внимание, что сигналы с выходов схем F и G, а также сигналы, поступающие на дополнительные входы логического блока можно подать через мультиплексоры M1–M3 на входы схемы H, поэтому можно реализовать логические функции с числом переменных больше четырех. Ниже приведен перечень функций, которые можно реализовать с помощью схем F, G и H в одном логическом блоке:

- Любая функция с числом переменных не более четырех, плюс любая другая функция с числом переменных не более четырех, которые не связаны с переменными первой функции, плюс любая третья функция с числом независимых переменных не более трех.

- Любая одна функция пяти переменных (см. задачу 10.34).
- Любая функция четырех переменных, плюс некоторые другие функции шести переменных, не зависящих от переменных первой функции.
- Некоторые функции с числом переменных до девяти, включая проверку на четность и проверку равенства двух 4-разрядных двоичных слов; в последнем случае возможно последовательное включение схем проверки (см. задачи 10.35 и 10.36).

При соответствующем программировании мультиплексоров M7–M8 и M12–M13 сигналы с выходов схем, вырабатывающих значения функций, могут быть выведены на выходы X и Y логического блока или запомнены в переключающихся по фронту D-триггерах FF1 и FF2. Триггеры могут срабатывать по нарастающему или по спадающему фронту общего тактового сигнала на входе K, в зависимости от выбора, сделанного с помощью мультиплексоров M9 и M14. С помощью мультиплексоров M10 и M15 в триггерах может быть также использован вход разрешения тактового сигнала EC. Сигнал EC и три других внутренних сигнала выбираются мультиплексорами M3–M6, изображенными в верхней части рис.10.44, из четырех различных сигналов, поступающих на входы C1–C4.

На выходы XQ и YQ выводятся сигналы с выходов триггеров данного логического блока. Если в каком-то логическом блоке триггеры не используются, то с помощью мультиплексоров M11 или M16 осуществляется «сквозное» прохождение на выходы XQ или YQ входных сигналов, выбранных мультиплексорами M4 и M6.

При программировании блока узлы, названные "S/R control", задают начальное состояние каждого триггера: 1 или 0. Этими узлами устанавливается также, на какой сигнал реагируют триггеры: на общий сигнал установки/сброса (не показанный на рисунке) или на сигнал SR данного логического блока, выбираемый мультиплексором M5.

Ничего себе, как много возможностей для программирования! Естественно, что задание структуры логического блока в микросхемах серии XC4000 – независимо от того, сколько логических блоков они содержат: 3136 или только 64, – не выполняется вручную. Производитель обеспечивает пользователя программой компоновки, которая распределяет, конфигурирует и соединяет логические блоки так, чтобы схема соответствовала описанию проекта на более высоком уровне, то есть описанию на языках ABEL, VHDL или Verilog, либо в виде принципиальной схемы.

Давайте вернемся к нашему вопросу: как построить универсальную схему, реализующую логические функции 4 переменных? Если вы решаете эту задачу на уровне вентилей, то она оказывается очень сложной, но если посмотреть на нее с другой точки зрения, то ее решение значительно облегчается. Любая функция 4 переменных может быть описана таблицей истинности, состоящей из 16 строк. Предположим, что мы храним таблицу истинности в 1-разрядной памяти на 16 слов. Подавая на адресные входы памяти четыре входных бита, мы получаем на выходе значение функции для этой комбинации значений переменных.

Именно такой подход был принят разработчиками ИС типа FPGA в фирме Xilinx. Схемы F и G, вырабатывающие значения логических функций, фактически являются всего лишь очень компактными и быстрыми статическими ОЗУ

16×1, а схема H представляет собой статическое ОЗУ 8×1. Когда логический блок используется для выполнения логических операций, при формировании его структуры в статическое ОЗУ из внешнего ПЗУ загружаются таблицы истинности логических функций F, G и H. Программирование мультиплексоров, указанных на рис. 10.44, также осуществляется путем загрузки ячеек памяти, представляющих собой отдельные D-защелки, управляющие мультиплексорами; информация в них тоже заносится при конфигурировании ИС. Такого рода программирование выполняется для всех логических блоков, входящих в состав ИС типа FPGA.

Помимо удобства программирования, применение памяти для хранения таблиц истинности имеет другое важное достоинство. Любой логический блок в микросхеме серии XC4000 при запуске можно сконфигурировать так, чтобы использовать его в качестве памяти, а не логики. Возможны несколько разных режимов формирования структуры логического блока:

- *Два статических ОЗУ 16×1*. Схемы F и G используются в качестве статических ОЗУ с независимыми адресными входами и входами записываемых данных. Однако вход разрешения записи у них общий.
- *Одно статическое ОЗУ 32×1*. Одни и те же четыре адресных бита подаются на входы схем F и G, а пятый адресный бит, поступающий на схему H и на схему разрешения записи, позволяет выбирать между верхней и нижней половинами памяти F и G.
- *Асинхронный или синхронный режим работы*. Структура упомянутых выше статических ОЗУ может быть сформирована так, чтобы при записи происходила «нормальная» асинхронная фиксация данных, либо запоминание данных осуществлялось по заданному фронту тактового сигнала K.
- *Одно статическое ОЗУ 16×1 с двумя портами*. Возможно независимое выполнение чтения и записи для двух разных ячеек одного и того же статического ОЗУ по двум наборам сигналов на адресных входах. В данном режиме поддерживаются только синхронные операции записи.

В этих режимах сигналы на функциональных входах F1–F4 и G1–G4 играют роль адреса, а на входы H0–H2 логического блока подаются данные и сигнал разрешения записи; сигналы данных, появляющиеся на выходах F и G, можно запомнить в триггерах FF1 и FF2 или вывести на выходы X и Y данного логического блока.

10.6.3. Блок ввода/вывода

Структура *блока ввода/вывода* (*I/O block, IOB*) в ИС семейства XC4000 показана на рис. 10.45. I/O-вывод можно использовать в качестве входа или выхода, либо в качестве того и другого.

У блока ввода/вывода в микросхемах семейства XC4000 больше средств «логического» управления, чем у его ближайшего «родственника» в ИС типа CPLD семейства XC9500. В частности, на пути входного и выходного сигналов имеются переключающиеся по фронту D-триггеры, возможность записи в которые определяется мультиплексорами M5–M7. Размещение входного и выходного триггеров «рядом» с I/O-выводами является особенно полезным свойством ИС типа

FPGA. Относительно большие задержки при прохождении сигналов от выходов внутренних триггеров логических блоков до блоков ввода/вывода могут затруднить стыковку данной ИС со стороны ее выходов с внешними синхронными системами, если частота тактового сигнала очень высока. Большие задержки от I/O-выводов до входов триггеров в логических блоках могут затруднить сопряжение данной ИС со стороны ее входов с внешней системой с точки зрения удовлетворения требованиям по времени установления и времени удержания, если внешние входные сигналы поступают непосредственно на тактовые входы триггеров внутри логических блоков, а не фиксируются сначала триггерами в блоках ввода/вывода. Конечно, применение триггеров в блоках ввода/вывода возможно только в том случае, когда технические требования к внешнему интерфейсу ИС типа FPGA допускают «конвейерный режим» работы по входам и выходам.

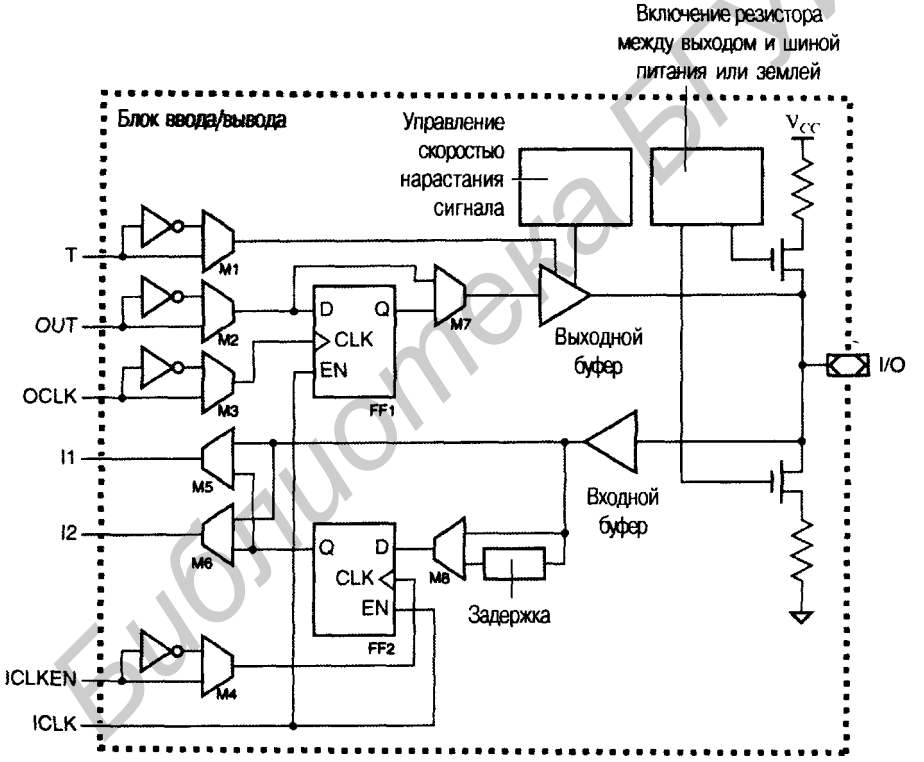


Рис. 10.45. Блок ввода/вывода в ИС семейства XC4000

Чтобы обеспечить конвейерный режим работы по входу в блоке ввода/вывода ИС серии XC4000 фактически делается еще один шаг в этом направлении: с помощью мультиплексора M8 на пути к D-входу входного триггера FF2 может быть введена задержка. Действие этого элемента заключается в задержке сигнала на D-входе относительно копий системных тактовых сигналов внутри данной ИС, гарантирующей, что время удержания по входу относительно внешнего тактового сигнала будет равно нулю. Этот режим достигается, конечно, за счет увеличения времени установления.

Другой возможностью логического управления в блоке ввода/вывода является выбор – с помощью мультиплексоров M1–M4 – полярности четырех входных сигналов, поступающих из матрицы логических блоков по структуре программируемых соединения. Этими входными сигналами являются: выходной бит OUT, сигнал разрешения T, открывающий выход с тремя состояниями, выходной тактовый сигнал OCLK и сигнал разрешения ICLKEN по тактовому входу.

Подобно блокам ввода/вывода в микросхемах серии XC9500, в блоках ввода/вывода микросхем серии XC4000 возможно управление аналоговыми параметрами выходного сигнала. Можно запрограммировать скорость изменения сигнала, вырабатываемого выходным буфером, а между I/O-выводом и шиной питания или землей можно включить резистор.

10.6.4. Программируемые соединения

Итак, лучшее мы оставили напоследок. Архитектура программируемых соединений в микросхемах серии XC4000 – прекрасный пример структуры, обеспечивающей богатые возможности образования необходимых связей и занимающей небольшую площадь на поверхности кремниевого кристалла.

Как показано на рис. 10.43, каждый логический блок в ИС типа FPGA окружен структурой соединений, которая в действительности является всего лишь совокупностью «проводов» с возможностью подключения к ним посредством соответствующего программирования. На рис. 10.46 структура соединений в ИС серии XC4000 изображена немного подробнее. Сигнальные линии действительно не являются «собственностью» какого-либо одного логического блока, а матрица логических блоков внутри ИС представляет собой мозаику, составленную точно из таких структур, какая показана на рисунке. Например, 100-кратное повторение этого рисунка дает матрицу логических блоков микросхемы XC4003 размером 10×10.

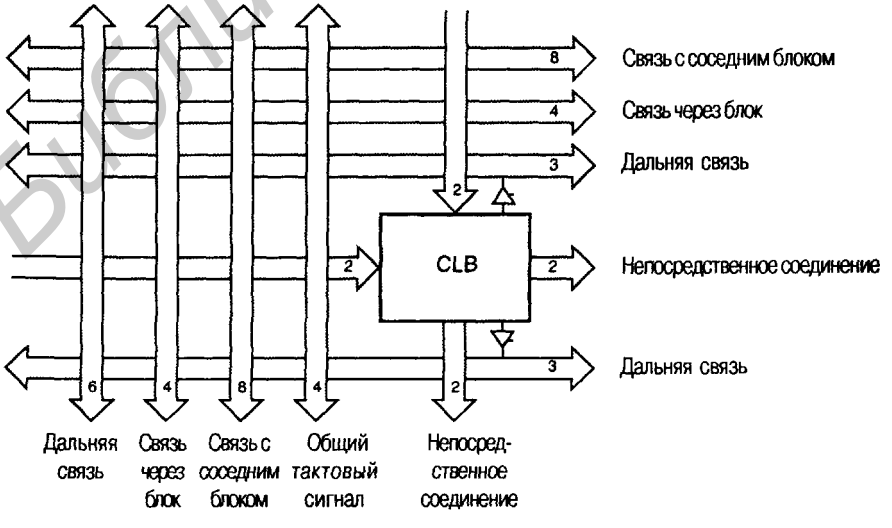


Рис. 10.46. Общая структура соединений в микросхемах серии XC4000

Цифрами внутри стрелок указано число сигнальных линий в каждой шине. Таким образом, мы видим, что у логического блока есть две выходные шины, идущие к логическим блокам, расположенным непосредственно под данным блоком и справа от него. Кроме того, каждый логический блок соединяется с тремя шинами, изображенными над ним, с одной шиной, находящейся под ним и с четырьмя шинами, расположенными слева от него. Сигналы по этим шинам могут передаваться в любом направлении.

Четыре сигнальные линии в шине, названной «Общий тактовый сигнал», позволяют логическому блоку наилучшим образом использовать тактовые входные сигналы, обеспечивая малую задержку и минимальный разброс задержек. Две шины «Связь с соседним блоком» предназначены для гибкой связи между расположенными рядом блоками помимо шин «Непосредственное соединение» с небольшим числом сигнальных линий и однонаправленной передачей сигналов.

По шине «Связь с соседним блоком» данный логический блок можно соединить не только с соседним, но и с другим блоком, но для этого потребуется более одной пересылки; при этом каждый раз сигналы должны пройти через программируемый переключатель, в результате чего задержка увеличивается. По шинам «Связь через блок» сигналы проходят мимо двух логических блоков, прежде чем попадают на переключатель, что позволяет получить меньшие задержки при более длинных связях. Для действительно длинных связей используются шины «Дальняя связь», в которых сигналы вообще не проходят через какие-либо программируемые переключатели; вместо этого сигналы вырабатываются буферами с тремя состояниями, помещенными рядом с логическим блоком, и проходят вдоль всей строки матрицы, образованной логическими блоками, или вдоль всего столбца.

Более детально логический блок и линии связи показаны на рис. 10.47. Маленькими квадратиками обозначены программируемые соединения: горизонтальная линия либо соединена с вертикальной линией, либо не соединена, в зависимости от значения бита (который опять хранится в защелке), указанного при программировании этого ключа. По краям матрицы, образованной логическими блоками, имеются дополнительные, специализированные программируемые соединения, обеспечивающие подключение к блокам ввода/вывода.

Заштрихованная область на рисунке, выделенная цветом, называется *матрицей программируемых переключателей* (*programmable switch matrix*, *PSM*). Более подробно эта матрица показана на рис. 10.48. Каждый ромбик на рис. (a) представляет собой *элементарный программируемый переключатель* (*programmable switch element*, *PSE*), с помощью которого любую линию можно соединить с любой другой, как показано на рис. (b). На рис. (b) приведены 6 возможных попарных соединений четырех линий, и для каждого из них в элементарном программируемом переключателе имеется логический ключ. Могут быть задействованы несколько логических ключей, ни одного из них или все; это снова определяется конфигурацией битов, которые хранятся в защелках. Таким образом, как показано на рис. (c), возможно много различных вариантов связи.

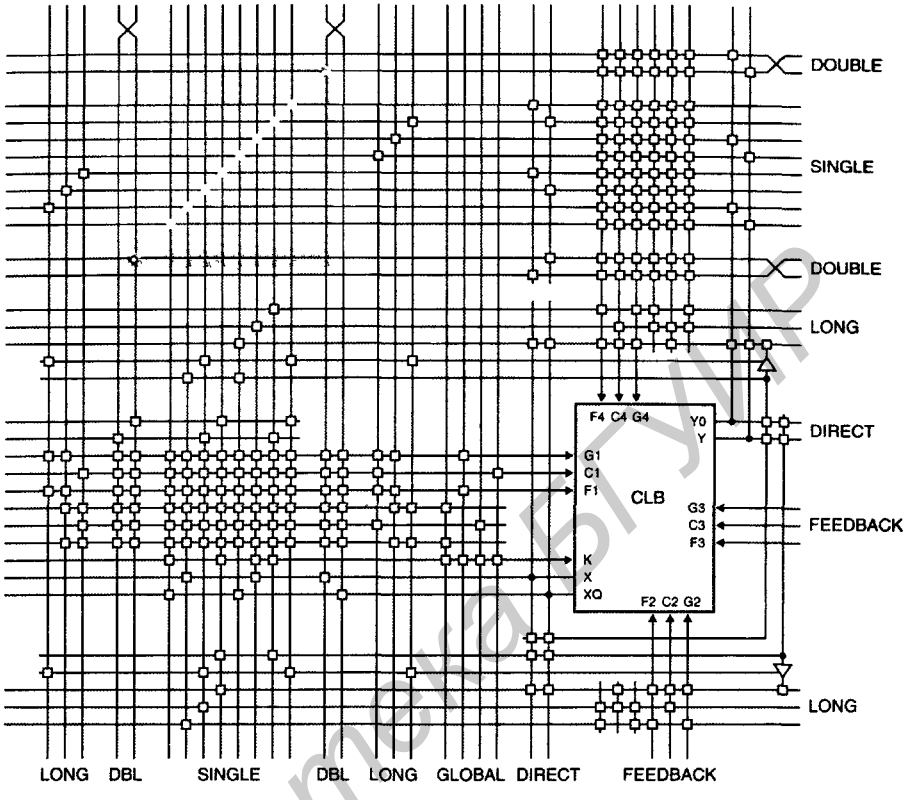


Рис. 10.47. Логический блок микросхем серии XC4000 и окружающая его структура соединений (LONG – дальняя связь; DBL, DOUBLE – связь через блок; SINGLE – связь с соседним блоком; GLOBAL – общий тактовый сигнал; DIRECT – непосредственное соединение; FEEDBACK – обратная связь)

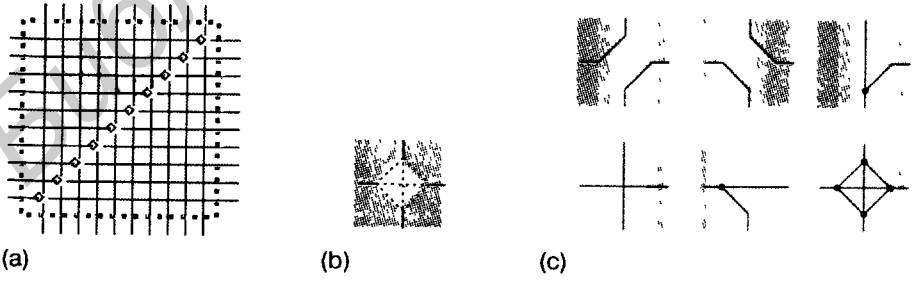


Рис. 10.48. Программируемые соединения в ИС серии XC4000: (a) матрица программируемых переключателей (PSM); (b) элементарный программируемый переключатель (PSE); (c) несколько возможных соединений

Матрица программируемых переключателей является существенным компонентом структуры соединений, представленной на рис. 10.47, обеспечивающей

связь между блоками. Установление соединений или их размыкание в элементарном программируемом переключателе позволяет продлевать и разрывать проводящие сегменты в шинах «Связь с соседним блоком» и «Связь через блок». Еще более важно, что матрица программируемых переключателей позволяет «повернуть сигналы на 90°» путем соединения горизонтального и вертикального проводников. Без этого нельзя было бы соединить данный логический блок с другими блоками, находящимися в другой строке или в другом столбце матрицы, образуемой логическими блоками.

Хотя матрица программируемых переключателей является необходимым компонентом, но за его использование приходится платить: при каждом прохождении сигналов через такую матрицу вносится небольшая задержка. Поэтому хорошая программа компоновки для ИС типа FPGA ищет не только какие-либо возможные размещения логических блоков и какую-то комбинацию соединений, которые будут работать. Программа «размещения и трассировки» затрачивает много времени, пытаясь оптимизировать характеристики устройства путем нахождения такого размещения, которое позволило бы сделать соединения короткими, и только после этого осуществляет реализацию самих соединений.

Подобно ИС типа CPLD, о достоинствах микросхем типа FPGA судят по гибкости их структур и устойчивости результатов, даваемых программой компоновки, при небольших изменениях в проекте. Самое большое разочарование наступает тогда, когда обнаруживается, что при небольших изменениях в большом проекте уже не выполняются требуемые временные соотношения. Как следствие этого, производители ИС типа FPGA научились обеспечивать наличие в архитектуре этих микросхем «дополнительных» ресурсов, чтобы помочь пользователям гарантированно получать стабильные результаты.

ХОРОШАЯ ПРАКТИКА

Размещение и трассировка являются понятной задачей, поскольку она является главной составляющей в действиях «внутреннего плана» при проектировании любой заказной микросхемы. Таким образом, одни и те же программы и одни и те же поставщики программ оказываются вовлеченными в решение задачи размещения и трассировки как в случае ИС типа FPGA, так и в случае специализированных ИС. Таким образом, можно считать, что любой опыт, приобретенный при работе с ИС типа FPGA, служит хорошей практикой проектирования специализированных ИС!