Цифровая электроника в живых картинках

А. А. Григорьев

Чтобы освоится с цифровой электроникой, важно не только уяснить её общие принципы, но и приобрести практический опыт конструирования и анализа логических схем. Неоценимую услугу оказывает здесь простая среда схемотехнического моделирования Logicky. Она позволяет без особых трудозатрат, как на листке бумажки, набросать логическую схему из примитивов и тут же, практически в ходе нарисования, посмотреть как она работает.

Данный текст задуман как краткое введение цифровую электронику с опорой на возможности этой среды для подготовки иллюстраций и упражнений. Основное изложение перемежается обращениями к Logicly, подобно тому как обычный текст – обращениями к картинкам. Картинки отделены от текста – это минус. Но зато они «живые» – тумблеры переключаются, кнопки нажимаются, лампочки мигают – это плюс. К тому же, не возбраняется корежить картинки как вздумается и рисовать их самостоятельно.

Большинство иллюстраций подготовлено и разложено по тематическим папкам в файлах с расширениями .Logicly. Что-то придется изменить или нарисовать самостоятельно. Навыки рисования и наблюдения работы схем приобретаются без труда.

1. Цифровые системы

Подобно всем сложным устройствам, цифровые системы состоят из относительно простых подсистем, которые передают и принимают электрические сигналы $x(t)$, рис. 1. Двоичные цифровые системы отличает то, что сигналы эти могут принимать ровно два значения – единица (1)-«истина» и нуль (0)-«ложь». Конкретные соглашения относительно того, какие состояния на линии связи считать нулем, а какие единицей, зависят от технологической платформы, на которой цифровая система реализуется. Чаще всего состояниями 1 и 0 считают факты присутствия на линии напряжения, превышающего некоторый порог u_{up}, или находящегося ниже порога u_{dn}. Возможность попадания этого напряжения внутрь зоны неопределенности, когда состояние на линии формально не определено, не создает заметных технических проблем. Заострять на ней внимание не стоит.

Набор из n двоичных сигналов образует параллельную шину, по которой передаются n-разрядные двоичные коды – n-блоки нулей и единиц. На n-разрядной шине существует ровно 2^n различных комбинаций нулей-единиц – значений n-разрядного кода. Иногда этим значениям придается та или иная смысловая интерпретация.

Чаще всего набор битов на n-разрядной шине интерпретируется как код целого числа. Линии шины (разряды) нумеруются числами j от 0 до $n-1$ и им присваиваются веса 2^j по степеням двойки. Тогда n-блок $\bar{a} = (a_{n-1}, \ldots, a_0)$ рассматривается как код числа

$$a = \sum_{j=0}^{n-1} a_j 2^j.$$

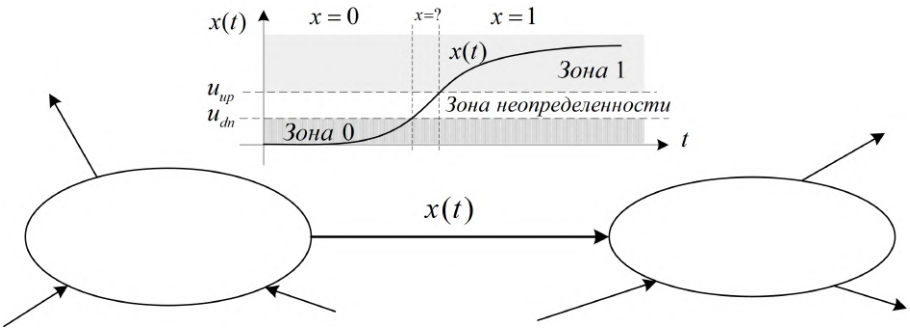


Рис. 1. Фрагмент цифровой системы

Это двоичный позиционный код. В n разрядах им можно представить числа в диапазоне от 0 до $2^n - 1$. К примеру, на 8-разрядной (байтовой) шине кодируются числа от 0 до $2^8 - 1 = 255$.

2. Комбинаторная логика

Цифровая электроника базируется на категории комбинаторного блока – устройства, которое вычисляет двоичные значения $y = 0/1$ некоторой функции $y = F(x_0, \ldots, x_{n-1})$ от $n \geq 1$ двоичных же входных переменных $x_j = 0/1$, рис. 2. Такие функции называют булевыми.

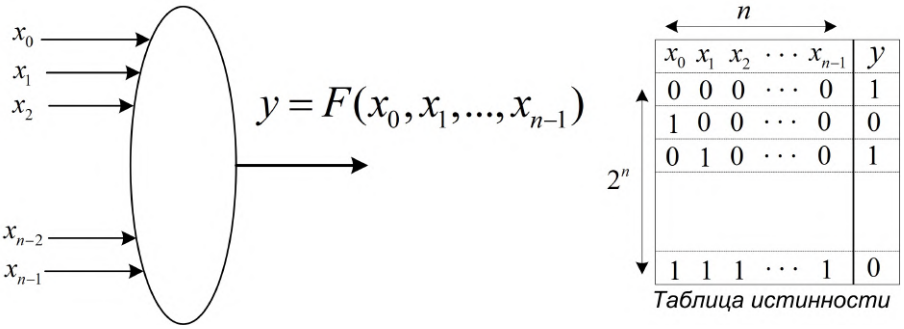


Рис. 2. Комбинаторный блок

Область определения булевой функции – множество всех возможных двоичных входов $(x_0, \ldots, x_{n-1})$ – конечно, насчитывает ровно 2^n элементов. Так что любая булева функция может быть задана конечной таблицей, в которой представлены значения y для всех возможных наборов $(x_0, \ldots, x_{n-1})$. Это таблица истинности – *Look Up Table* (LUT). При n входных переменных она насчитывает ровно 2^n строк. В каждой из строк можно поставить любое из двух значений y. Так что всего имеется 2^{2^n} булевых функций от n переменных.

Проблема технической реализации комбинаторного блока по предъявленной таблице истинности и составляет содержание раздела цифровой электроники, известного как комбинаторная логика.

Ее решение открывает чрезвычайно широкие возможности. Простой пример показан на рис. 3. Это 8-разрядный перемножитель, который принимает на входы двоичные позиционные коды чисел $a = \sum_{j=0}^{7} a_j 2^j$ и $b = \sum_{j=0}^{7} b_j 2^j$ и выдает 16-разрядный

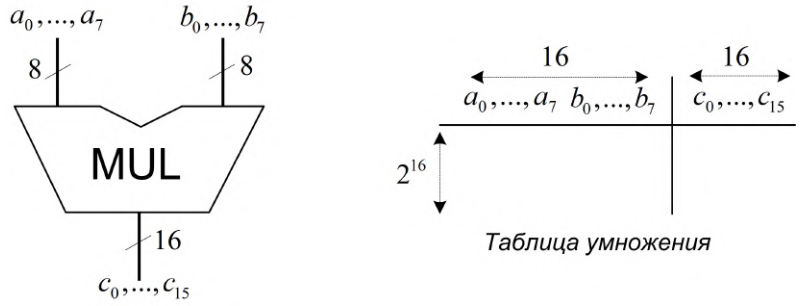


Рис. 3. Комбинаторный перемножитель

код произведения $c = ab = \sum_{j=0}^{15} c_j 2^j$. Чтобы построить такой перемножитель, нужно реализовать 16 комбинаторных блоков $c_j = F_j(a_0, \ldots, a_7, b_0, \ldots, b_7)$ от 16-и входных переменных каждый. Их таблицы истинности составит каждый, заполнив таблицу умножения на рисунке. Другое дело, что в этой таблице будет $2^{16} = 65536$ строк. Но это сложность не принципиального, а технического плана. Такой умножитель реализуется, к примеру, на постоянной запоминающем устройстве емкостью всего в 2^{16} 16-разрядных слов. Это совсем не много.

С позиций теории алгоритмов, комбинаторная логика позволяет реализовать вычисление любой частично рекурсивной (всюду определенной) функции. Класс алгоритмически вычислимых (вычислимых машиной Тьюринга) функций значительно шире. Он включает общерекурсивные функции, не определенные на некоторых входах. На таких входах машина Тьюринга имеет право никогда не останавливаться, в то время как комбинаторные блоки циклить не способны.

Хотя таблица истинности и содержит исчерпывающую информацию о булевой функции $y = F(x_0, \ldots, x_{n-1})$, она мало что дает для ее реализации. Возникает естественное желание представить эту функцию некоторой формулой, которая позволяла бы не брать значения из таблицы, а эффективно вычислять их посредством некоторых элементарных операций над входами. Инструментарий для построения таких формул и дает двоичная булева алгебра.

2.1. Булевы алгебры

Булева алгебра – это аксиоматическая система с переменными $x, y, z, \ldots$, двумя константами 0 и 1, парой двухместных операций

$$\text{сложением} \quad \text{OR} \quad z = x \vee y = x + y,$$

$$\text{умножением} \quad \text{AND} \quad z = x \wedge y = x \cdot y = xy$$

и одной одноместной операцией отрицания NOT $z = x' = \bar{x}$, которая определяется набором из четырех аксиом (1)-(4):

1.	$x + y = y + x$	$\parallel$	$xy = yx$	*коммутативность*
2.	$x + 0 = x$	$\parallel$	$x1 = x$	*нейтральные элементы*
3.	$x(y + z) = xy + xz$	$\parallel$	$x + yz = (x + y)(x + z)$	*дистрибутивность*
4.	$x + x' = 1$	$\parallel$	$xx' = 0$	*законы дополнения*
5^*.	$x + (y + z) = (x + y) + z$	$\parallel$	$x(yz) = (xy)z$	*ассоциативность*

Если словами, то сложение и умножение коммутативны (*перестановочны*) (аксиомы 1), обладают *нейтральными элементами* по сложению (0) и умножению (1)

(аксиома 2). Умножение *дистрибутивно* относительно сложения, а сложение – относительно умножения (аксиома 3). Аксиома 4 вводит операцию *отрицания* (дополнения). Наконец, аксиома ассоциативности 5 на самом деле является теоремой – может быть выведена из первых четырех.

Помимо присутствия отрицания, булеву алгебру радикально отличает от привычной числовой алгебры отсутствие обратных элементов как по сложению, так и по умножению. Слагаемые можно, к примеру, как угодно переставлять местами, но нельзя перенести с одной стороны от знака равенства в другую – сложение не обратимо. Так что к работе с булевыми формулами нужно привыкнуть.

Рассматривая аксиомы для сложения (слева) и умножения (справа), нетрудно обнаружить, что одни переходят в другие заменой операций сложения на операции умножения ($\cdot \leftrightarrow +$), а нулей – на единицы ($0 \leftrightarrow 1$). Это великая двойственность, фундаментально присущая булевой алгебре. Всякая ее теорема $A = B$, приравнивающая булевы формулы A и B, имеет двойственную теорему – $A^D = B^D$, которая приравнивает формулы A^D, B^D, построенные из A, B заменами ($\cdot \leftrightarrow +$), ($0 \leftrightarrow 1$).

Помимо ассоциативности (5), из аксиом (1)-(4) выводится масса теорем разной степени очевидности. Начнем с простого:

$$0' = 1 \quad \| \quad 1' = 0 \quad \textit{отрицания 0 и 1}$$

Вот вывод этих фактов из аксиом – синтаксическое доказательство:

$$1 =_4 0 + 0' =_2 0', \quad 0 =_4 1 * 1' =_2 1'.$$

Впрочем, второе доказывать не обязательно, поскольку оно двойственно к первому.

Обе операции идемпотентны:

$$x + x = x \quad \| \quad xx = x \quad \textit{идемпотентоность},$$

поскольку

$$x =_2 x1 =_4 x(x + x') =_3 xx + xx' =_4 xx + 0 =_2 xx.$$

Двойственное утверждение каждый может доказать самостоятельно. Полезное упражнение.

Нейтральные элементы поглощают переменные

$$x + 1 = 1 \quad \| \quad x0 = 0 \quad \textit{слабое поглощение},$$

$$x + 1 =_2 (x + 1)1 =_4 (x + 1)(x + x') =_3 x + 1x' =_2 x + x' =_4 1.$$

Справедлива общая теорема поглощения:

$$x + xy = x \quad \| \quad x(x + y) = x \quad \textit{поглощение},$$

$$x + xy =_2 x1 + xy =_3 x(1 + y) = x1 =_2 x.$$

В доказательствах теорем с отрицанием важную роль играет лемма о единственности отрицания, которая звучит так: Пусть $x + a = 1$, $xa = 0$ и $x + b = 1$, $xb = 0$. Тогда $a = b$.

В самом деле,

$$a = a1 + 0 = a(x + b) + xb = ax + ab + xb = 0 + b(x + a) = 0 + b1 = b.$$

Поскольку $x + x' = 1$ и $xx' = 0$, из $x + a = 1$ и $xa = 0$ вытекает $a = x'$, что и означает единственность обратного элемента x'.

Отсюда легко выводится теорема об идемпотентности отрицания:

$$(x')' = x.$$

Имеем $x' + (x')' = 1$ и $x'(x')' = 0$, $x' + x = 1$ и $x'x = 0$. А это означает, что $x = (x')'$.

Важнейшую роль в булевой алгебре играет знаменитая теорема де-Моргана:

$$(x + y)' = x'y' \quad \| \quad (xy)' = x' + y' \quad \textit{теоремы де Моргана},$$

Чтобы доказать ее, проверим, что пара $(x + y)$ и $x'y'$ удовлетворяют законам дополнения:

$$(x + y) + x'y' = (x + y + x')((x + y + y') = (y + 1)(x + 1) = 1,$$
$$(x + y)x'y' = xx'y + yx'y' = 0 + 0 = 0.$$

А поэтому $x'y' = (x + y)'$.

Тождества де-Моргана обнаруживают вторую примечательную двойственность в булевой алгебре – двойственность де-Моргана: Если

$$y = F(x_1, \dots x_n)$$

некоторая булева формула от переменных $x_1, \dots, x_n$, то

$$y' = F^D(x'_1, \dots, x'_n)$$

Отрицание «опускается» на переменные с заменой выражения F на двойственное F^D, которое получается заменами $(\cdot \leftrightarrow +)$, $(0 \leftrightarrow 1)$.

Аксиоматическая булева алгебра допускает множество различных моделей. Чтобы построить такую модель, нужно поставить в соответствие переменным $x, y, z, \dots$ и константам 0, 1 некоторые объекты и определить операции $(\cdot, +, ')$ над этими объектами так, чтобы все аксиомы $(1) - (4)$ оказались выполненными.

Естественные модели булевой алгебры получаются, когда в качестве объектов берутся все возможные подмножества $A \subseteq \Omega$ некоторого множества Ω, включая все множество $\Omega = 1$ и пустое подмножество 0. Операция умножения вводится как пересечение подмножеств $(AB = A \cap B)$, сложения – как объединение $(A + B = A \cup B)$, а отрицания – как дополнение $(A' = \Omega \setminus A)$. Аксиомы $(1) - (4)$ оказываются выполненными (простая проверка), а вместе с ними выполняются и все синтаксически вытекающие из этих аксиом теоремы. Исторически аксиоматика булевой алгебры и появилась на свет как формализация этой теоретико-множественной модели.

Покорившая цифровой мир модель двоичной булевой алгебры получается, когда множество Ω состоит из единственного элемента 1. Его подмножества – это пустое подмножество 0 и все множество $\Omega = 1$. Несмотря на крайнюю простоту, в этой модели, не содержащей ничего кроме нуля и единицы, выполняются все аксиомы и теоремы булевой алгебры.

В технических реализациях двоичной булевой алгебры операции $('\,, \cdot, +)$ реализуют инвертор НЕ/NOT $y = x'$ и вентили И/AND $y = x_1x_2$ и ИЛИ/OR $y = x_1 + x_2$. Это базисные комбинаторные блоки, из которых строится все остальное.

2.2. Знакомимся с вентилями

*** **Logicly** ***

⋆ Запустите Logicly. Слева – разбитая по категориям библиотека примитивов, справа – фрагмент неограниченного во все стороны рабочего листа. Перетащите из библиотеки на лист инвертор *NOT Gate*, переключатель *Toggle Switch* и лампочку *Light*

Bulb. В появляющихся диалогах будут просить дать входу и выходу экспортные имена. Можно отложить это на потом, закрыв диалоги. Соедините компоненты проводниками – курсор мыши от одной клеммы до другой. И вот оно уже работает – положение переключателя изменяется щелчком мыши, лампочка загорается и гаснет.

Имеется два режима курсора: основной – *Select* и режим *Pan* выбора видимого в окне фрагмента листа. Они переключаются инструментом «ладошка», но лучше выучить горячие клавиши S/P. Освойтесь с инструментами масштабирования картинки (справа внизу). Работает и колесо мыши. Цвета проводников можно выбрать в диалоге по *Edit/Application Settings .../Editor*.

⋆ Откройте в Logicly файл **Comb/gates**. Посмотрите на общепринятые схемные изображения вентилей. Изучите логику функционирования инвертора NOT и вентилей AND и OR. Выведите их таблицы истинности – выделяем схему в бокс мышью и нажимаем клавишу *Generate Truth Table* в панели инструментов.

Чтобы вывести таблицу, нужно дать всем входам (переключателям) и выходам (лампочкам) уникальные экспортные имена – они появятся в заголовках столбцов.

Обратите внимание, что наблюдение единицы на выходе AND позволяет однозначно предсказать состояния на всех его входах, в то время как наблюдение нуля не говорит о них почти ничего. С вентилем OR все наоборот. Таким образом, состояние 1 на выходе AND и 0 на выходе OR является выделенными (особыми). Вентиль AND регистрирует факт совпадения двух единиц на входах и подтверждает его выдачей выделенной единицы на выход. Вентиль OR регистрирует факт совпадения двух нулей и подтверждает его выдачей выделенного нуля.

*** **Logicly** ***

⋆ Создав новый файл по File/New ..., соберите нужные для этого схемы и проверьте выполнение теорем об идемпотентности умножения ($xx = x$) и сложения ($x + x = x$).

⋆ Проверьте теоремы поглощения: $x + 1 = 1$, $x0 = 0$, $x + xy = x$, $x(x + y) = x$.

⋆ Проверьте выполнение аксиом дистрибутивности:

$$x(y + z) = xy + yz, \quad x + yz = (x + y)(x + z).$$

⋆ Проверьте выполнение теорем де-Моргана:

$$xy = (x' + y')', \quad x + y = (x'y')'.$$

Обратите внимание, что при инвертировании логики – замене ($0 \leftrightarrow 1$) – AND превращается в OR, а OR в AND. Так что между этими вентилями нет принципиальной разницы.

2.3. Иногда они бывают многовходовыми

Многовходовые вентили AND, OR реализуют вычисление сумм и произведений нескольких переменных: $y = x_1 + x_2 + \cdots + x_n$, $y = x_1x_2 + \ldots x_n$. Коммутативность и ассоциативность булевой алгебры гарантирует, что результаты вычислений по этим формулам не зависят от порядка выполнения операций. Можно расставить скобки так:

$$y = x_1 + x_2 + x_3 + x_4 = (((x_1 + x_2) + x_3) + x_4), \quad y = x_1x_2x_3x_4 = (((x_1x_2)x_3)x_4),$$

а можно иначе

$$y = x_1 + x_2 + x_3 + x_4 = ((x_1 + x_2) + (x_3 + x_4)), \quad y = x_1x_2x_3x_4 = ((x_1x_2)(x_3x_4)).$$

Это ведет к лестничным и древовидным схемам реализации многовходовых вентилей.

* * * **Logicly** * * *

⋆ Изучите их, открыв файл **Comb/ngates**. Самостоятельно соберите схемы 6-входового вентиля AND лестницей $y = ((((x_1x_2)x_3)x_4)x_5)x_6$ и деревом $y = ((x_1x_2)(x_3x_4))(x_5x_6)$. Сравните наибольшие длины комбинаторных путей (в числе вентилей от входа до выхода) в одной и другой реализациях.

При большом числе n входов древовидные структуры обладают глубиной порядка $\log_2 n$ слоев логики, в то время как глубина лестничных достигает $n-1$. Древовидные реализации выигрывают по задержке при том же числе двухвходовых вентилей.

Многовходовый вентиль AND по прежнему опознает факт совпадения всех единиц на входах и подтверждает его выделенным значением 1 на выходе. Вентиль OR опознает совпадение нулей, подтверждая его нулем на выходе. Инвертирование логики – замена $(0 \leftrightarrow 1)$ – переводит AND в OR и наоборот.

2.4. Нормальные формы

Удобно считать, что булева функция $F(x, y, z, \dots)$ от n переменных на самом деле определена на множестве из 2^n вершин n-мерного единичного гиперкуба, подобного показанным на рис. 4 для случаев $n = 1, 2, 3$. Поставщик таблицы истинности разбросал по вершинам гиперкуба нужные ему значения – 0 или 1. Требуется построить формулу, которая принимает именно эти значения во всех вершинах. Нормальные формы и дают регулярный метод решения этой задачи.

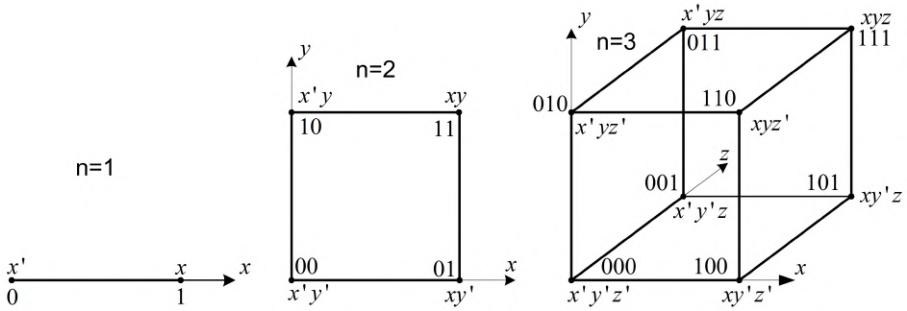


Рис. 4. Единичные кубы

Все начинается с реализации конъюнктивного (AND) терма – функции, которая принимает единственное единичное значение в одной конкретной вершине куба. Если это вершина $(1, 1, 1, \dots)$ с единичными координатами, решение известно. Это просто n-входовый вентиль AND. Единица на его выходе появляется только в ответ на единицы на всех входах. Если же среди координат вершины имеются нули, их просто нужно «переделать» в единицы инверторами, как показано на рис. 5a.

Имеется ровно 2^n вариантов расстановки инверторов на n входах, и они дают реализации конъюнктивных термов для всех 2^n вершин гиперкуба. Соответствующие формулы приведены при всех вершинах на рисунках.

Дизъюнктивный (OR) терм на рис. 5b – это двойственная структура. Он реализует функцию, принимающую единственное нулевое значение в выбранной вершине.

Теперь, чтобы реализовать произвольную булеву функцию, нужно построить термы-конъюнктеры для всех вершин, в которых эта функция принимает единич-

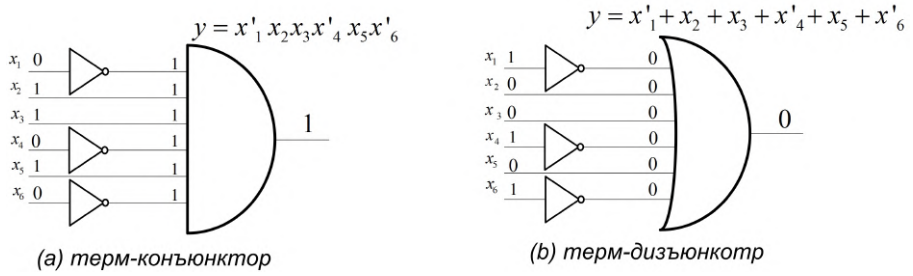


Рис. 5. Реализации термов

ное значение, и объединить их выходы по логике OR. То, что получится, и называют нормальной конъюнктивной формой реализации булевой функции, рис. 6a.

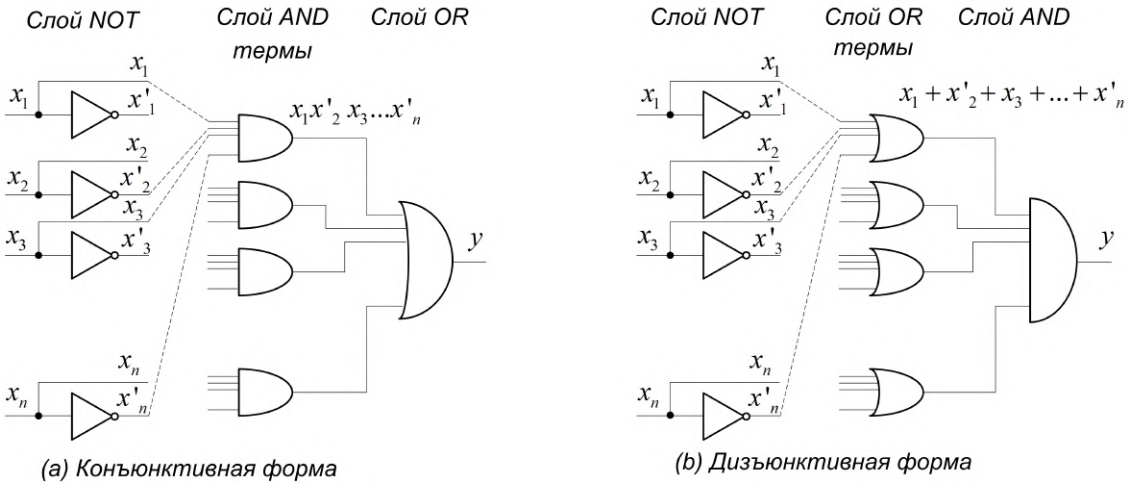


Рис. 6. Нормальные формы

Нормальная форма трехслойна. Слой инверторов формирует прямые и инверсные копии входных сигналов. Слой вентилей AND вычисляет термы. Вентиль OR объединяет их выходы. Количество термов равно числу единичных значений в таблице истинности реализуемой функции. Сама же функция определяется связями между выходами инверторов и входами вентилей AND. В микросхемах программируемых логических матриц PLM изначально присутствуют все эти связи – каждый выход слоя NOT соединен проводниками со всеми входами слоя AND. В процессе программирования ненужные связи разрушаются (например, необратимо прожигаются).

В нормальной дизъюнктивной форме на рис. 6b термы реализуют нулевые значения функции, которые объединяются по логике AND. Количество термов равно числу нулей в таблице истинности.

* * * **Logicly** * * *

⋆ Самостоятельно реализуйте термы конъюнктор и дизъюнктор для вершины с координатами (1,0,1,1,0).

⋆ Реализуйте нормальные конъюнктивную и дизъюнктивную формы для функции исключительно ИЛИ (XOR) $z = x \oplus y = x'y + y'x = (x + y)(x' + y')$. Она известна

также как сумматор по модулю два или функция неравнозначность.

2.5. Минимизация по Карно

Нормальные формы, как правило, поддаются упрощению. Известен приписываемый Карно регулярный метод минимизации, в основе которого лежит объединение термов в соседних вершинах гиперкуба.

Две вершины гиперкуба назовем соседними, если они соединены коротким ребром единичной длины. Их термы отличаются инверсией ровно одной из n переменных: AxB и $Ax'B$, где A и B – некоторые фиксированные произведения прочих переменных и их инверсий.

Пусть дана конъюнктивная нормальная форма. Она является суммой некоторого количества термов. Каждому терму отвечает вершина гиперкуба, в которой он принимает единственное единичное значение. Пометим вершины, термы которых входят в нормальную форму, черным цветом, прочие - белым.

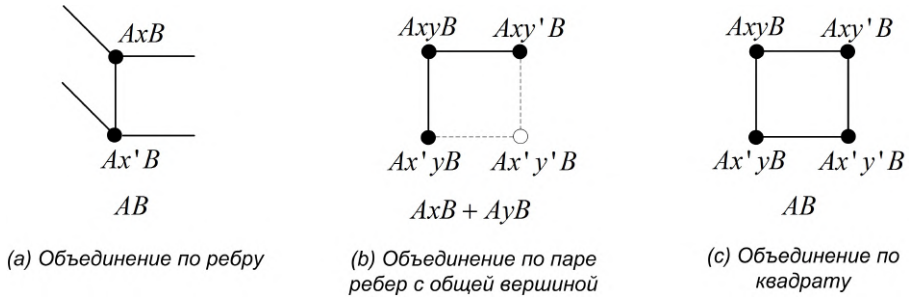


Рис. 7. К минимизации нормальной формы

Если обе соседние вершины – черные (входят в нормальную форму), рис. 6a, их термы удается объединить в один, воспользовавшись дистрибутивностью сложения:

$$AxB + Ax'B = A(x + x')B = A1B = AB$$

В результате вместо двух n-термов от n переменных останется один $(n-1)$-терм. Заметная экономия. Метод Карно состоит в том, чтобы выполнить такие объединения для всех пар соседних черных вершин в гиперкубе.

В n-мерном гиперкубе у каждой вершины имеется n соседей. Её объединение с одним из них не мешает выполнить такие же объединения с прочими. Терм этой вершины нужно просто «размножить», пользуясь идемпотентостью сложения: $x = x + x + x + \ldots$ Вот пример объединения трех вершин на рис. 6b:

$$xy+x'y+xy' = xy+xy+x'y+xy' = (xy+x'y)+(xy+xy') = (x+x')y+x(y+y') = x+y.$$

Вместо трех n-термов остается два $(n-1)$-терма. Менее эффективно, но все равно не плохо. Особенно хорошо удаются объединения по квадратам и, вообще, по k-мерным $(k \le n)$ подкубам, рис. 6c:

$$xy + x'y + xy' + x'y' = (x + x')y + y'(x + x') = (x + x')(y + y') = 1.$$

Вместо четырех n-термов остается один $(n-2)$-терм.

Чтобы выполнить минимизацию по Карно, нужно построить граф соседних черных вершин и выполнить объединения по всем его связным компонентам. Порядок

выполнения объединений не существенен. Это экспоненциально сложная, даже NP-полная задача. Существуют эффективные субоптимальные алгоритмы ее решения, которые реализованы в программных синтезаторах логических структур. Попадаются интернет ресурсы, предоставляющие бесплатные услуги по минимизации булевых функций.

Широко известны приемы минимизации построением карт Карно. На поверку эти карты оказываются просто остроумными приемами расположить вершины многомерного куба на плоскости так, чтобы сохранить информацию об их соседстве. К сожалению, карты Карно работают только в малых размерностях, до 4 включительно.

Приходится сталкиваться и с частично определенными функциями, значения которых на некоторых комбинациях входов безразличны. В их таблицах истинности наряду с двоичными значениями 0 и 1 присутствует символ X – безразлично (*Don't Care*). При минимизации их нормальных форм открываются дополнительные возможности – безразличные значения X можно заменить на 0 или 1 так, чтобы обеспечить наибольшую эффективность минимизации.

Двойственные, нормальные дизъюнктивные формы минимизируются аналогично, с опорой на дистрибутивность умножения:

$$(A + x + B)(A + x' + B) = xx' + (A + B) = 0 + (A + B) = (A + B).$$

Существуют булевы функции, нормальные формы которых не поддаются минимизации. Их гиперкубы не содержат ни одной пары соседних черных вершин. Пример – сумматор по модулю два от любого числа переменных $y = x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

* * * **Logicly** * * *

⋆ Постройте нормальную конъюнктивную форму мажоритарного элемента – функции $MAJ(x, y, z)$, которая принимает единичное значение, когда среди значений переменных x, y, z единиц больше чем нулей. Минимизируйте ее по Карно. Проверьте результат экспериментально. Ответ:

$$MAJ(x, y, z) = x'yz + xy'z + xyz' + xyz = xy + xz + yz.$$

⋆ Экспериментально проверьте теорему о консенсусе из булевой алгебры. Это важный инструмент минимизации.

$$xz + yz' + xy = xz + yz'.$$

Докажите ее. Можно лобовой минимизацией по Карно, можно проще.

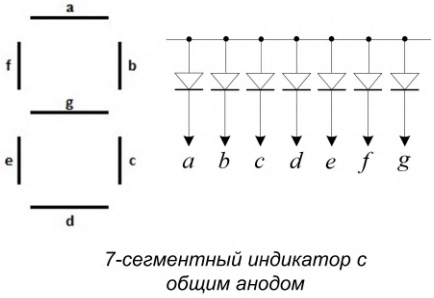
⋆ Реализуйте конъюнктивную нормальную форму функции $b(x_0, x_1, x_2, x_3)$ управления сегментом b индикатора на рис. 8. Минимизируйте ее по Карно к

$$b = x_0x'_1x_2x'_3 + x_1x_2x_3 + x'_0x_2x_3 + x_0x_1x_3 + x'_0x_1x_2.$$

Проверьте результат экспериментально.

⋆ Если ограничиться BCD (*binary coded decimal*) кодом на входах x со значениями от 0 до 9, формула упрощается к $b = x_0x'_1x_2+x'_0x_1x_2$. Проверьте это экспериментально. Какие значения эта формула принимает на недопустимых входах от A до F

	x3	x2	x1	x0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0
A	1	0	1	0	0	0	0	1	0	0	0
B	1	0	1	1	1	1	0	0	0	0	0
C	1	1	0	0	0	1	1	0	0	0	1
D	1	1	0	1	1	0	0	0	0	1	0
E	1	1	1	0	0	1	1	0	0	0	0
F	1	1	1	1	0	1	1	1	0	0	0



7-сегментный индикатор с общим анодом

Рис. 8. Преобразование позиционного кода в семисегментый

2.6. Базисы и комбинированные вентили

Существование нормальных форм доказывает, что набор элементов NOT, AND и OR образует базис комбинаторной логики – располагая этими элементарными функциями, можно реализовать любой комбинаторный блок. Вопрос только в сложности этой реализации. Базис этот избыточен, поскольку AND и OR связаны по де-Моргану:

$$xy = (x' + y')', \quad x + y = (x'y')'.$$

Минимально достаточно либо пары NOT, AND, либо пары NOT, OR.

Комбинированные вентили NAND (*NOT-AND/И-НЕ*) – $z = (xy)'$ и NOR (*NOT-OR/ИЛИ-НЕ*) – $z = (x + y)'$ содержат инвертор в себе и, поэтому, образуют базисы из единственного элемента. Вот формулы для реализации NOT, AND и OR в базисе NAND:

$$x' = (1x)' = (xx)'; \quad xy = ((xy)')'; \quad x + y = (x'y')'.$$

Вентиль NAND ($(xy)' = x' + y'$) – это одновременно и AND с инвертором на выходе (в инверсной логике по выходу), и OR с парой инверторов на входах (в инверсной логике по входам).

То же в базисе NOR:

$$x' = (0 + x)' = (x + x)'; \quad x + y = ((x + y)')'; \quad xy = (x' + y')'.$$

Вентиль NOR ($(x + y)' = x'y'$) – это одновременно и OR с инвертором на выходе (в инверсной логике по выходу), и AND с парой инверторов на входах (в инверсной логике по входам).

Комбинированные вентили, образующие базис сами по себе, используются как базовые элементы тех или иных технологических платформ. В качестве базового элемента семейства транзисторно-транзисторной логики (ТТЛ) выбран, к примеру, вентиль NAND. Суть в том, что налаживание производства этого единственного компонента полностью закрывает вопрос об элементной базе для реализации любой комбинаторной логики.

В свое время была популярна технологическая концепция базового матричного кристалла (БМК), в ячейках которого находится, скажем, 10^6 базовых вентилей. Потребителю остается только разработать металлизацию – топологию проводных соединений выходов этих вентилей со входами с тем, чтобы получить нужную функциональность. В наше время металлизация стала частью единого технологического

процесса производства кристаллов. Поэтому концепция БМК стала не столь актуальной. Библиотеки современных технологических процессов включают вентили всех возможных типов.

* * * **Logicly** * * *

⋆ Разработайте схемы, реализующие функции NOT, AND и OR в базисе NAND и в базисе NOR.

⋆ Реализуйте сумматор по модулю два на четырех вентилях NAND:

$$z = x \oplus y = x'y + y'x = \Big(((xy)'x)'((xy)'y)'\Big)'$$

⋆ Реализуйте отрицание сумматора по модулю два (NOT-XOR) на четырех вентилях NOR:

$$z = (x \oplus y)' = (x' + y)(x + y') = \Big(((x + y)' + x)' + ((x + y)' + y)'\Big)'$$

⋆ Откройте модель **Comb/impl**, в которой реализована микросхема логической функции «импликация» $A \to B$ (A влечет B):

$$A \to B = A' + B = (AB')'.$$

Изучите её таблицу истинности.

Это основная функция математической логики, которая формализует дедуктивный вывод – суждение от общего к частному:

$$A \;\parallel\; (A \to B) \quad \Rightarrow \quad B$$

Если истинно A и A влечет B, то истинно B.

⋆ Удостоверьтесь в том, что импликация составляет базис булевой алгебры, реализовав NOT, OR и AND из элементов $A \to B$:

$$A' = (A \to 0); \quad A + B = (A \mapsto 0) \to B; \quad AB = (A \to (B \to 0)) \to 0$$

⋆ Проверьте, что $(A \to (B \to 0))$ – это NAND.

Для специалистов по математической логике подобные формулы легко читаемы. Дело привычки.

2.7. Линейные нормальные формы

Множество из двух элементов 0 и 1 с умножением xy (AND) и сложением по модулю два $x \oplus y = x'y + y'x$ (XOR) – это простейшая из возможных модель аксиоматики поля – поле F_2. В ней дотошно выполняются все свойства сложения и умножения, привычные по каждодневной работе с вещественными числами. Обе операции коммутативны и ассоциативны, есть нейтральные элементы по умножению ($x1 = x$) и сложению ($x \oplus 0 = x$). Умножение дистрибутивно относительно сложения $z(x \oplus y) = zx \oplus zy$. Единственная особенность в том, что сумма (по модулю два) двух единиц равна нулю: $1 \oplus 1 = 0$. F_2 – это простое поле характеристики два.

Интерпретация пары 0,1 как элементов поля превносит в булеву традиционные категории линейной алгебры. Множество двоичных n-блоков $\bar{x} = (x_0, \ldots, x_{n-1})$ становится не просто набором вершин гиперкуба, а n-мерным линейным пространством

F_2^n над полем F_2 с покоординатными операциями умножения на скаляр $\alpha \in F_2$, $\alpha\bar{x} = (\alpha x_0, \ldots, \alpha x_{n-1})$ и сложения,

$$\bar{x} \oplus \bar{y} = (x_0 \oplus y_0, \ldots, x_{n-1} \oplus y_{n-1}).$$

Естественным образом выделяется класс линейных булевых функций – линейных комбинаций переменных x_j с двоичными коэффициентами $\alpha_j \in F_2$:

$$y = f(\bar{x}) = \bigoplus_{j=0}^{n-1} \alpha_j x^j$$

Линейная функция реализует линейное отображение пространства F_2^n в одномерное пространство F_2. Ядро этого отображения – множество $\bar{x}$, таких что $f(\bar{x}) = 0$, – является $(n-1)$-мерным подпространством в F_2^n размерности $(n-1)$ и насчитывает 2^{n-1} элементов. Так что любая нетривиальная (не тождественно нулевая) линейная функция уравновешена – принимает равное число единичных и нулевых значений.

Тождество $1 \oplus x = 1'x + 1x' = x'$ позволяет выразить отрицание через сумму по модулю два. Это делает операции сложения (XOR) и перемножения (AND) базисом булевой алгебры. Чтобы представить произвольную функцию в этом базисе, достаточно в конъюнктивной нормальной форме заменить сложения $(+)$ на $(\oplus)$ (когда среди слагаемых не более одной единицы, эти сложения тождественны), а отрицания x' заменить на $(1 \oplus x)$. После раскрытия скобок и приведения подобных получится некоторый многочлен. К примеру,

$$x' + xy' + xyz' = (1 \oplus x) \oplus x(1 \oplus y) \oplus xy(1 \oplus z) = 1 \oplus xyz.$$

Таким образом, любая булева функция представима многочленом от своих переменных. Это и есть её линейная нормальная форма. Общий вид её таков:

$$f(\bar{x}) = \bigoplus_{\bar{j} \in J} \alpha_{\bar{j}} \bar{x}_{\bar{j}}.$$

Здесь J – множество всех подмножеств набора индексов $(0, 1, \ldots, n-1)$, включая пустое подмножество. Оно насчитывает ровно 2^n элементов. Каждому из поднаборов $\bar{j} \in J, \bar{j} = (j_1, \ldots j_k)$ отвечает произведение $\bar{x}_{\bar{j}} = x_{j_1} \ldots x_{j_k}$ k-переменных. Пустому набору индексов отвечает произведение нуля переменных, равное 1.

Линейная нормальная форма реализует множество всех булевых функций от n-переменных как 2^n-мерное линейное пространство над F_2 с базисом из всех возможных произведений k переменных, $0 \leq k \leq n$. Этот базис известен как базис Жегалкина. Все булевы функции представимы двоичными линейными комбинациями его элементов – то есть многочленами от n-переменных. Степень этого многочлена называют степенью булевой функции. Функции степени 0 – это константы $f(\bar{x}) = 0$, $f(\bar{x}) = 1$. Линейные функции – это булевы функции степени 1.

Для приложений, связанных с криптографией, важно представление о булевой функции со случайными равновероятными значениями. Из общих соображений ясно, что при инвертировании любой из входных переменных истинно случайная функция должна изменять или сохранять значение с равной вероятностью – если по вершинам гиперкуба случайно разбросать значения $0/1$, то в соседних вершинах равновероятно окажутся одинаковые или противоположные значения.

Известен класс бент-функций, удовлетворяющих этому требованию. Функция $f(\bar{x})$ является бент-функцией, если ее градиент вдоль любого направления $\bar{v} = (v_0, \ldots, v_{n-1})$

$$\bigtriangledown_{\bar{v}} f(\bar{x}) = f(\bar{x}) \oplus f(\bar{x} \oplus \bar{v})$$

уравновешен – принимает равное число 2^{n-1} нулевых и единичных значений. Пусть в векторе направления $\bar{v}$ всего одна единица в позиции k. Условие уравновешенности градиента вдоль этого направления означает, что при инвертировании x_k функция меняет значение 2^{n-1} раз (когда градиент равен 1) и не меняет значение в столько же раз.

Характеризация класса бент-функций – это актуальная и нерешенная проблема булевой алгебры. Известно, что они существуют только при четных n и среди них есть функции степени 2.

*** **Logicly** ***

⋆ Смоделируйте бент-функцию от четырех переменных.

$$y = xy \oplus uv.$$

Проверьте, что ее градиент вдоль любого из четырех направлений $\bar{v} = (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)$ уравновешен. Докажите это. Докажите, что класс бент-функций замкнут относительно сдвигов: если $f(\bar{x})$ – бент, то таковой же является и $f(\bar{x} + \bar{v})$ при любом $\bar{v}$.

---

3. Библиотечные комбинаторные блоки

Цифровые устройства – это чрезвычайно большие системы, насчитывающие миллионы вентилей. Как и всякие сложные системы, они устроены иерархически. Из вентилей создаются относительно простые функциональные блоки с концептуально определенным поведением. Из них складываются блоки посложнее, потом еще более сложные, и так далее. Находясь на верхнем, прикладном уровне этой иерархии, сложно представить себе, что за способностью устройства демонстрировать полезное поведение скрывается хорошо организованная работа огромного числа элементарных логических вентилей. Редактируя текст на компьютере, Вы даже не имеет представления об их существовании. Наоборот, находясь на нижнем уровне иерархии – уровне вентилей, почти невозможно уяснить целесообразность скрытую за хаотическим с виду мельтешением нулей и единиц.

Ключевую роль в построении иерархии играет способность выделять функциональные блоки, обладающие ясными поведенческими описаниями типа – какие входы, какие выходы, что умеет. При наличии такого описания, внутреннее устройство блока уходит на второй план. Многие, имеющие массовое применение блоки удостаиваются включению в библиотеки. Некоторые из ставших библиотечными комбинаторных блоков и обсуждаются ниже.

3.1. Дешифраторы

Дешифраторы – это комбинаторные функциональные блоки, которые решают простейшую задачу выбора одного объекта из многих.

Имеется 2^n объектов, рис. 9а. Каждому присвоен номер от 0 до $2^n - 1$. Требуется выбрать один из них, послав ему активный высокий (единичный) сигнал выбора s (*select*). Прочие объекты остаются не выбранными – получают пассивный низкий сигнал. Дешифратр выбирает объект, номер которого задан ему двоичным позиционным кодом на адресной шине $a_0, \ldots, a_{n-1}$. На n-разрядной шине можно представить номера от 0 до $2^n - 1$. Соответственно, полный n-разрядный дешифратор может выбрать один из 2^n объектов.

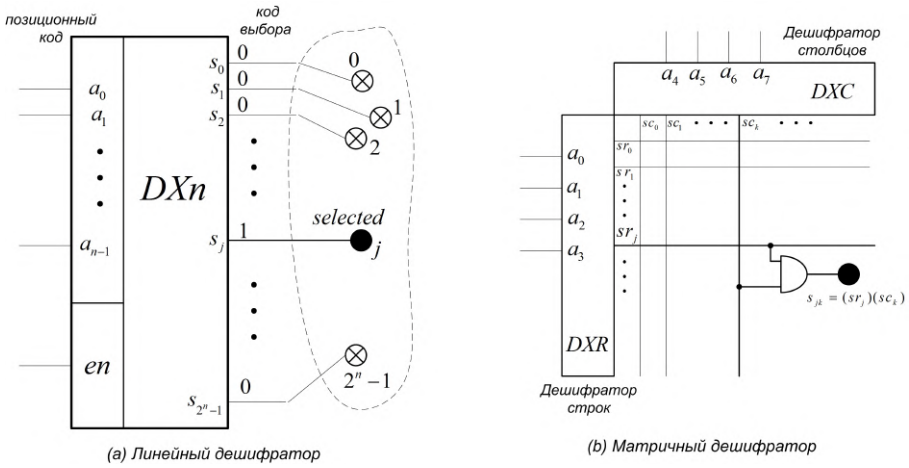


Рис. 9. Дешифраторы

Удобно считать, что дешифратор реализует преобразование двоичного позиционного кода адреса в код выбора – 2^n-разрядный двоичный код, в котором допустимо существование только одной активной единицы.

Внутреннее устройстово дешифратора элементарно. В нем попросту реализованы все 2^n термов нормальной конъюнктивной формы от n адресных переменных. Можно реализовать и термы дизъюнктивной формы. Сигналы выбора станут тогда активными низкими – все единицы, кроме одного нуля на выбранном выходе.

Как правило, дешифраторы снабжают дополнительным входом разрешения en (*enable*). При наличии активного разрешающего сигнала на входе en дешифратор исправно выполняет функцию выбора. В отсутствие разрешения он блокирован – не выбирает никого.

Дешифратор быстро становится сложным в реализации. Уже при $n = 10$ он насчитывает $2^{10} = 1024$ вентиля AND на 10 входов каждый. Значительный выигрыш в сложности достигается в двухкоординатных (матричных) дешифраторах, рис. 9b. Шина адреса делится на две части. Одна часть используется для выбора строк матрицы – сигналы rs (*raw select*), вторая – для выбора столбцов – cs (*column select*). Находящийся в узле матрицы объект получает сигнал выбора с выхода локального вентиля AND, то есть когда он выбран и по строке и по столбцу.

Массовая профессия дешифраторов – быть дешифраторами адреса в микросхемах памяти, содержащих миллионы одинаковых ячеек. Именно дешифратор (чаще всего, матричный) и раздает сигналы выбора ячейкам, номера которых задаются извне на адресных входах микросхемы.

Располагая n-разрядным дешифратором, можно реализовать любую булеву функцию от n переменных – достаточно объединить нужные выходы по логике OR, то есть достроить нормальную форму. Но это не слишком хорошая идея. Много реализованных в дешифраторе термов окажутся не востребованными. Идея оправдывает себя, когда нужно реализовать сразу несколько функций. Термы, не востребованные в одной из них, оказываются востребованными в других.

* * * **Logicly** * * *

⋆ Откройте модель **Comb/dx**, в которой реализованы микросхемы простых де-

шифраторов с $n = 1, 2$. Вникнете в их работу, изучите внутренние структуры. Обратите внимание, что 1-входовый дешифратор без входа разрешения – это просто инвертор.

⋆ По аналогии с каскадной реализацией 2-входового дешифратора из трех 1-входовых, постройте схему дешифратора на 3 входа.

⋆ Скопируйте внутренность дешифратора DX2 в отдельный файл через карман, и преобразуйте его из конъюнктивной формы в дизъюнктивную.

⋆ На 2-входовом дешифраторе реализуйте сумматор по модулю два: $y = a_0 \oplus a_1$.

⋆ Изучите модель **Comb/seg7** преобразователя 4-разрядного позиционного кода в семи-сегментный (см. рис. 8), реализованного на полном 4-разрядном дешифраторе.

3.2. Обращение дешифратора

Этот комбинаторный блок преобразует входной код выбора (единственная единица на одном из входов) в позиционный код номера выбранной линии. Он интересен как пример ситуации, когда частичная определенность функции дает максимальное упрощение ее реализации в нормальной форме. К примеру, к терму $x'_0x_1x'_2$, опознающему комбинацию $(0, 1, 0)$ на входах, можно прибавить термы, принимающие единичные значения на запрещенных входах с более чем с одной единицей так, чтобы получился квадрат. Объединение термов по сторонам квадрата упрощает сумму до 1-терма x_1:

$$x'_0x_1x'_2 = x'_0x_1x'_2 + x_0x_1x'_2 + x'_0x_1x_2 + x_0x_1x_2 = x_1x'_2 + x_1x_2 = x_1.$$

Подобным образом, любой n-терм, опознающий код с единственной единицей, можно «доукомлектовать» до $(n-1)$-мерного гиперкуба и упростить до 1-терма – переменной. Так что в нормальной форме реализации обращенного дешифратора полностью исключается слой конъюнктеров. Остаются только объединения переменных по логике OR.

$***$ **Logicly** $***$

⋆ Откройте модель **Comb/bdx**. Проанализируйте реализацию 4-входового обращенного дешифратора $BDX2$. Дополнительный выход on – это антипод входа en дешифратора. Он сигнализирует о наличии единицы на одном из входов (кто-то выбран). Проанализируйте каскадную реализацию 8-входового обращенного дешифратора $BDX3$ из двух блоков $BDX2$.

⋆ Реализуйте 8-входовый обращенный дешифратор $BDX3$ напрямую – в минимизированной нормальной форме (из вентилей OR).

3.3. Операции с битовыми масками

Пара полезных комбинаторных задач касается битовых масок - n-разрядных кодов $(1, 1, \ldots, 1, 0, 0, \ldots, 0)$, содержащих $0 \le k < n$ подряд идущих единиц. Число k называют весом маски.

Первая – это формирование маски с весом, заданным двоичным позиционным кодом числа k. Вторая, обратная, – это подсчет числа единиц в маске.

Вторая задача возникает в параллельных аналого-цифровых преобразователях (АЦП), организованных по схеме рис. 10. На оси напряжений U расставлен ряд эквидистантных порогов $1, 2, 3, \ldots$, с которыми параллельно сравнивается напряжение U_{ad}, подлежащее преобразованию в цифровой код. Связанные с каждым из порогов устройства сравнения (компараторы), выдают 1, когда U_{ad} превышает порог, и 0 –

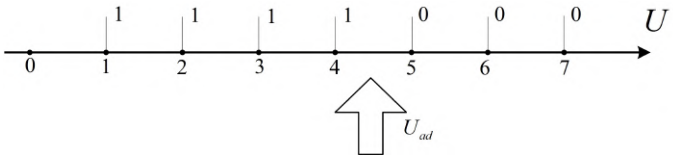


Рис. 10. Параллельный АЦП

иначе. Сырым выходным кодом такого преобразователя оказывается битовая маска, вес которой и характеризует текущее значение U_{ad}. Требуется оценить число единиц в ней.

* * * **Logicly** * * *

⋆ Откройте **Comb/mask**. Блок $DMSK3$ формирует оценку веса маски в столбце $(x_0, x_1, \ldots, x_6)$. Вникните в его внутренне устройство. Слой сумматоров по модулю два формирует градиент битовой маски – единицу на границе 1/0. Обращенный дешифратор $BDX3$ преобразует положение этой единицы в позиционный код на выходе.

⋆ Проанализируйте несложную логику работы 2-разрядного генератора маски $MASK2$. Уясните принцип каскадирования двух генераторов $MASK2$ для построения генератора $MASK3$. По аналогии постройте генератор $MASK4$.

3.4. Коммутационная логика

Аппарат булевой алгебры адекватен и для описания релейно-контактных схем, конструируемых из электромеханических блоков. Привнесенные из теории релейно-контактных схем категории и составляют концептуальную основу комбинаторной коммутационной логики.

Базовый примитив релейно-контактной логики – это электромагнитное реле, ключ на рис. 11, который может либо соединить вход x с выходом y (сигнал en – высокий, ток в обмотке создает магнитное поле, контакт притянут и замкнут), либо разорвать это соединение. Его комбинаторным эквивалентом – логическим ключом может быть вентиль AND, один из входов которого (x) интерпретируется как вход сигнала, а второй (en) – как вход управления. Когда на входе en активный высокий сигнал, ключ замкнут и сигнал проходит на выход – $y = x$. При $en = 0$ ключ разомкнут, $y = 0$ и не зависит от x.

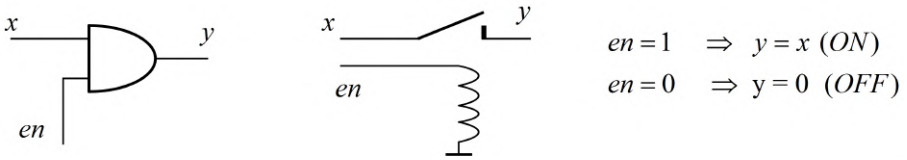


Рис. 11. Релейно-контактная аналогия AND

* * * **Logicly** * * *

⋆ Откройте файл **Comb/key**. Изучите работу вентиля AND в роли логического ключа.

⋆ Примените в схеме ключа иные вентили – OR, NAND, NOR. Научитесь бегло отвечать на вопросы: какой уровень на входе en открывает ключ данного типа, какой уровень присутствует на его выходе в закрытом состоянии, какой ключ инвертирует сигнал x, а какой – нет.

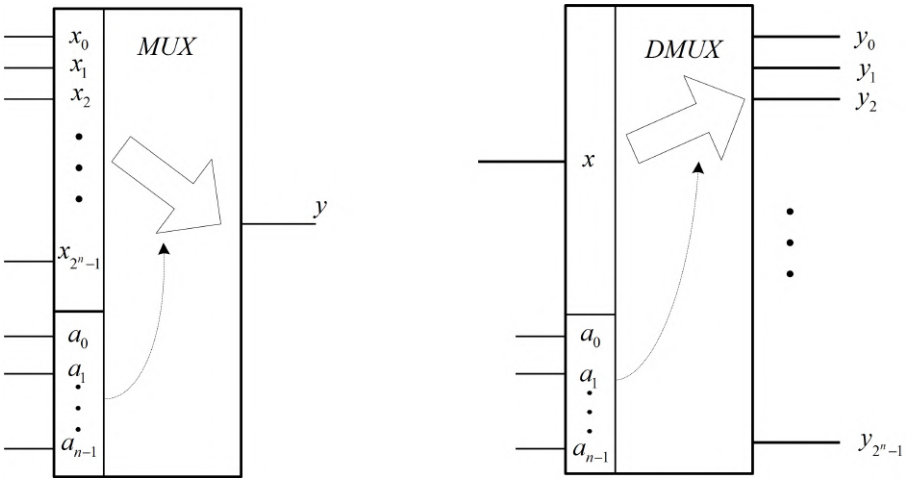


Рис. 12. Мультиплексор и демультиплексор

Показанные на рис. 12 комбинаторные блоки – это логические аналоги многопозиционного электромеханического переключателя, шагового искателя. Мультиплексор MUX (слева) подключает на выход y один из входов x_j, номер j которого задан позиционным кодом на адресной шине $(a_0, \ldots, a_{n-1})$. Демультиплексор $DMUX$ переключает единственный вход x на один из выходов y_j. Каждый из них – это не более как линейка логических ключей, один из которых открывается сигналом выбора с дешифратора адреса.

Мультиплексор с n-разрядной адресной шиной позволяет реализовать любую булеву функцию $f(a_0, \ldots, a_{n-1})$ от n переменных – достаточно перенести на входы x_j столбец значений 0/1 из таблицы истинности.

*** **Logicly** ***

⋆ Откройте файл **Comb/mux**. Изучите работу мультиплексоров и демультиплексоров на 2 и 4 входа.

⋆ На 4-входовом мультиплексоре $MUX2$ реализуйте сумматор по модулю два $y = a_0 \oplus a_1$. Константы 0 и 1 взять из библиотеки.

⋆ Добавьте в 2-входовый мультиплексор $MUX1$ вход разрешения en, так чтобы при $en = 0$ ни один из входов не был подключен к выходу.

⋆ Из трех элементов $MUX1$ постройте мультиплексор $MUX2$ на 4 входа.

⋆ Постройте мультиплексор $MUX1$ на трех вентилях NAND и одном инверторе, на трех вентилях NOR и одном инверторе.

3.5. Комбинаторные сдвигатели

Распространены комбинаторные блоки, именуемые барабанными сдвигателями (*barrel shifter*). Они реализуют сдвиг n-разрядного двоичного кода на число пози-

ций k, $0 \le k < n$, заданное позиционным кодом. Название барабанный подразумевает циклический сдвиг, но встречаются и не циклические сдвигатели.

Их реализации интересны как пример использования схемы Горнера эффективного вычисления степени x^k. Вместо того, чтобы умножать на x много раз, можно представить показатель k двоичным позиционным кодом $k = \sum k_j 2^j$, вычислить все факторы x^{2^j} со степенями двойки в показателе по схеме $x^2 = xx$, $x^4 = x^2x^2$, $x^8 = x^4x^4, \ldots$, а затем перемножить те из них, которые входят в разложение k с единичными коэффициентами k_j.

Барабанный сдвигатель – это каскад блоков, каждый из которых, в зависимости от управляющего сигнала a, либо пропускает код напрямую ($a = 0$), либо сдвигает его на $2^k = 1, 2, 4, 8, \ldots$ позиций ($a = 1$). Комбинируя управляющие сигналы можно гибко управлять итоговым сдвигом.

*** **Logicly** ***

⋆ Откройте файл **Comb/bsh8** с 8-разрядным барабанным сдвигателем, который «умеет» циклически сдвигать код на шине $\bar{x} = (x_7, \ldots, x_0)$ на число позиций от 0 до 7, заданное на входах (a_0, a_1, a_2). Вникните в его структуру.

Внутри обнаруживается три управляемые сдвигателя на 1, 2 и 4 позиции. Их внутренности однотипны – содержат по 8 двухвходовых мультиплексоров. Различие только в проводных связях.

⋆ Откройте файл **Comb/mux3** с заготовкой мультиплексора, три входа которого d, sr, sl предназначены для режимов прямого прохождения – d, сдвига влево – sl и вправо – sr. Разработайте на ней схему реверсивного сдвигателя, умеющего циклически сдвигать 4-разрядный входной код на 0,1,2,3 позиции вправо и влево.

⋆ Откройте файл **Comb/bsh8x2** со схемой нециклического сдвигателя, который сдвигает вправо код на шине $\bar{a} = (a_7, \ldots, a_0)$, принимая в освободившиеся старшие разряды биты из младших разрядов шины $\bar{b} = (b_7, \ldots, b_0)$. Вникните в его внутреннюю структуру. Коды на шинах $\bar{a}$ и $\bar{b}$ вначале сдвигаются циклически. Результат «собирается» потом из старших разряд $\bar{b}$ и младших разрядов $\bar{a}$ по битовой маске, которая формируется блоком $MASK3$.

---

4. Комбинаторная арифметика

Сердцевину комбинаторной логики составляют реализации арифметических операций над числами, представленными двоичным позиционным кодом.

4.1. Полусумматоры и инкременторы

Все начинается с полусумматора – комбинаторного блока hADD, который суммирует два бита a и b и формирует бит результата сложения $s = a \oplus b$ и бит переноса $c = ab$ в старший разряд. Антипод – это вычитающий полусумматор hSUB, который вычитает бит b из бита a и формирует результат $s = a \oplus b$ и перенос $c = a'b$ – заем из старшего разряда.

*** **Logicly** ***

⋆ Откройте файл **Math/hSUM**. Изучите реализацию и работу полусумматоров hADD и hSUB. Посмотрите их таблицы истинности.

⋆ Изучите устройство комбинированного блока hAS, который «умеет» складывать или вычитать в зависимости от сигнала на входе *sub* (*subtract*). В него добавлен мультиплексор MUX2, который включает один из двух режимов формирования переноса.

⋆ Блок hASNAND отличается тем, что сумматор по модулю два реализован в нем на вентилях NAND. Вникните в его структуру.

Использование мультиплексоров для коммутации логических связей – это универсальный прием организации архитектуры многорежимных комбинаторных блоков. Обратите на него внимание.

На полусумматорах строятся инкременторы (INC) и декременторы (DEC) – которые прибавляют единицу к позиционному коду на входе , или вычитают её.

*** **Logicly** ***

⋆ Изучите организацию двухрежимного инкрементора/декрементора на управляемых полусумматорах – файл **Math/INC**.

⋆ Изучите внутренность блока INC4. Реализация инкрементора доведена в ней до уровня вентилей. Присутствует слой сумматоров по модуля два, которые суммируют входные биты с битами переносов. Переносы же формируются линейкой вентилей AND. Эта линейка образует цепь сквозного переноса (*ripple carry*). Единица переноса распространяется по ней слева направо, достигая входов сумматоров. Вентили AND выступают в роли логических ключей, каждый из которых может заблокировать распространение переноса. Первый же нуль во входном коде выключает ключ, размыкая цепь. До входа данного сумматора перенос доходит только при условии, что во всех предыдущих (младших) разрядах входного блока присутствуют единицы.

Цепь сквозного переноса – это лестничная реализация многовходового вентиля AND. Она не слишком удачная в плане быстродействия. Задержка распространения переноса растет в ней линейно с увеличением числа разрядов.

⋆ В быстром инкременторе FINC4 с параллельным переносом для каждого из разрядов реализован отдельный многовходовый вентиль AND, который вычисляет перенос непосредственно по битам входного кода. Изучите его структуру.

⋆ Взяв за основу внутренность блока INC4, превратите его в декрементор.

4.2. Полные сумматоры

Полный сумматор ADD реализует арифметическое сложение не двух, а трех битов – двух разрядных битов a и b из позиционных кодов слагаемых и бита c переноса из младшего разряда. Он формирует результат сложения $s = a \oplus b \oplus c$ и перенос в старший разряд

$$C = ab'c + a'bc + abc' + abc = (a \oplus b)c + ab = ab + ac + bc.$$

При сложении трех битов возникает как максимум один перенос: $1+1+1 = 3 = 1+2$, а это единичный результат и перенос 2 в старший разряд.

Его антипод – вычитатель SUB – арифметически вычитает из бита a бит b и перенос c. Получается $s = a \oplus b \oplus c$ и

$$C = a'b'c + abc + a'bc' + a'bc = (a \oplus b)'c + a'b = a'b + a'c + bc.$$

Примечательно, что блоки ADD и SUB формируют результат s одинаково и отличаются только логикой формирования переноса. Результат s формируется трехвходовым сумматором по модулю два, который не поддается минимизации по Карно.

* * * **Logicly** * * *

⋆ Откройте файл **Math/ADD1**. Изучите реализации полного сумматора ADD и вычитателя SUB на двух полусумматорах/вычитателях.

⋆ Блоки AC1, AC2 содержат два варианта реализации логики переноса для сумматора. Первый хорош тем, что задействованный в нем сумматор по модулю два может быть позаимствован из реализации вычисления суммы s. Второй вариант – это минимизированная нормальная форма. Блоки SC1, SC2 содержат те же два варианта для вычитателя. Изучите эти реализации, докажите справедливость соответствующих булевых формул.

⋆ Реализуйте двухрежимный сумматор/вычитатель со входом управления sub. Нужный для этого мультиплексор имеется в библиотеке.

Многоразрядные сумматоры/вычитатели реализуются как линейки одноразрядных с цепями сквозного переноса – перенос, генерируемый в предыдущем разряде поступает на вход переноса следующего.

На самом деле n-разрядные сумматор/вычитатель реализует операции сложения и вычитания в модульном кольце Z_N по модулю $N = 2^n$. Перенос в несуществующий разряд номер n, равно как и заем из него игнорируются. Факт приведения результата по модулю $N = 2^n$ регистрируется как наличие переноса C из старшего разряда. Этот перенос служит признаком арифметического переполнения в беззнаковой арифметике.

Возникновение переполнений в 4-разрядном сумматоре поясняет рис. 13. Представимые в четырех разрядах числа от 0 до 15 равномерно распределены по кругу. Прибавлению к числу A числа B отвечает движение по этому кругу из точки A на B шагов против часовой стрелки. Вычитанию – движение по часовой стрелке. Переполнение происходит всякий раз, когда при этом движении пересекается «красная линия» – линия беззнакового арифметического переполнения C.

* * * **Logicly** * * *

⋆ Откройте файл **Math/ADD4**. Изучите структуру 4-разрядных сумматора и вычитателя со сквозным переносом. Глядя на рис. 13, проверьте их работу на нескольких примерах. Уясните логику формирования сигналов C арифметического переполнения при сложении и вычитании.

⋆ Должно быть ясно, что обе схемы регулярно продолжаются вправо до любого числа разрядов.

Часто бывает нужно просто сравнить два числа, не выполняя их вычитание. С задачей справляется комбинаторный компаратор. Он принимает на входы позиционные коды чисел A и B, сравнивает их, и сообщает о результате сравнения одним из трех сигналов – $A = B$, $A > B$ или $A < B$. Сравнение проводится побитово, начиная со старших разрядов. Как только обнаруживается, что пара битов в некотором разряде отличается, потребность в дальнейшем сравнении отпадает. Результат уже известен – либо $A > B$, либо $A < B$.

* * * **Logicly** * * *

⋆ Откройте файл **Math/COMP**. Изучите логику работы одноразрядного компаратора CMP1. Он принимает пару разрядных битов a, b и сигнал eq (*equal*), который

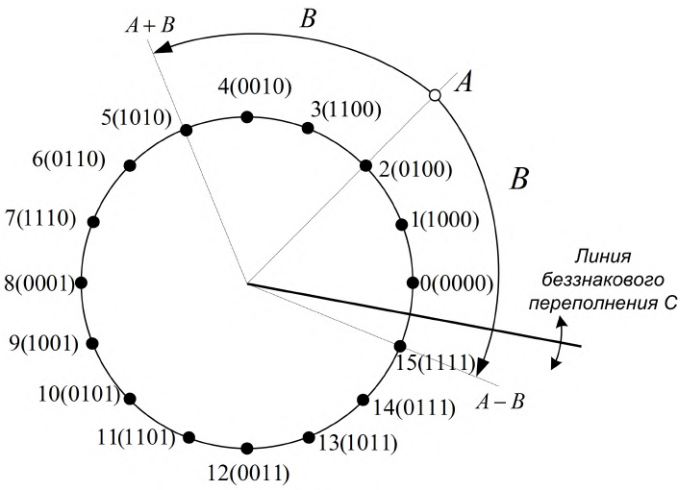


Рис. 13. Арифметические переполнения в беззнаковой арифметике

сигнализирует о факте равенства битов в предыдущем, более старшем разряде. Если биты в старшем разряде не равны ($eq = 0$), работа компаратора блокируется. При $eq = 1$ он активируется, сравнивает биты a, b и устанавливает один из выходных сигналов gt ($a > b$), lt ($a < b$) или eq ($a = b$).

⋆ Загляните внутрь 4-разрядного CMP4 компаратора и разберитесь в логике его работы.

4.3. Знаковая арифметика

В тех же n-разрядах, в которых представляются положительный целые числа от 0 до $2^n - 1$, можно кодировать и числа со знаком. Для представления знаковых чисел общепринят **двоичный дополнительный код**. Положительные числа представляются в нем обычным позиционным кодом, отрицательные – дополнениями до единицы несуществующего n-го разряда:

$$-A = 2^n - A.$$

Число 2^n различных n-разрядных кодов четно. Число нуль особое – не имеет знака. Так что для кодирования остальных чисел остается нечетное число $2^n - 1$ вариантов. Поэтому, кодировка, которая представляла бы столько же положительных чисел, сколько и отрицательных, невозможна. Граница между положительными и отрицательными числами проводится по значению старшего разряда, который объявляется знаковым S (sign). Диапазон положительных чисел ограничивается для этого числами от 1 до $2^{n-1} - 1$, коды которых имеют нуль в старшем знаковом разряде. Тогда получается, что диапазон отрицательных чисел от $-1 = 2^n - 1$ до $-2^{n-1} = 2^n - 2^{n-1} = 2^{n-1}$ содержит на одно число больше. Зато знаковый разряд кодов всех отрицательных чисел оказывается единичным. Пример знаковой кодировки для 4 разрядов дан на рис. 14.

Существует эффективный алгоритм изменения знака числа в двоично-дополнительной кодировке. В его основе лежит простое тождество:

$$-A = 2^n - A = (2^n - 1) - A + 1 = A' + 1.$$

Суть в том, что двоичный код числа $2^n - 1$ содержит единицы во всех разрядах, а поэтому код разности $A' = (2^n - 1) - A$ – это в точности поразрядное отрицание кода числа A. Чтобы обратить знак числа (операция NEG), нужно применить к его коду порязрядное NOT, а затем прибавить единицу.

* * * **Logicly** * * *

⋆ Откройте файл **Math/NEG**. Изучите реализацию 4-разрядного инвертора двоично-дополнительного кода. Проверьте, что отличные от 0 и $2^{n-1} = -8$ числа образуют пары $A \leftrightarrow -A$, рис. 14, а числа 0 и - 8 инвариантны относительно изменения знака.

⋆ При попытке изменения знака у не имеющего положительного двойника числа - 8 устанавливается флаг OVF (*overflow*) арифметического переполнения в знаковой арифметике. Объясните логику его формирования.

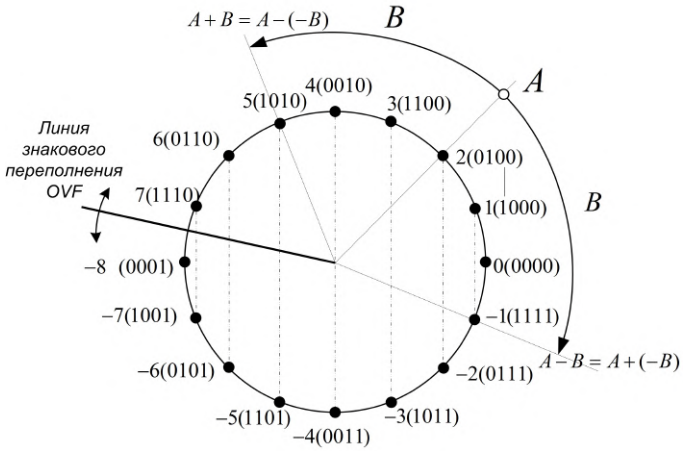


Рис. 14. Арифметические переполнения в знаковой арифметике

Важно, что для сложения/вычитания знаковых чисел в двоично-дополнительной кодировке не требуются какие-либо особые сумматоры/вычитатели. Комбинаторные блоки сложения/вычитания на самом деле реализуют операции над абстрактными позиционными кодами A и B, не интересуясь там, как мы их интерпретируем. При знаковой интерпретации, рис. 14, движению от точки A на B шагов против часовой стрелки отвечает прибавление положительного числа B, или вычитание отрицательного числа $-B$. Движение по часовой стрелке – это либо вычитание положительного B, либо прибавление отрицательного $-B$.

Если при этом движении переходится «красная черта» – линия OVF на рисунке, происходит арифметическое переполнение в знаковой арифметике (*overflow*). Его регистрируют по факту изменения старшего знакового бита S (*sign*), вызванного тем, что перенос в старший разряд и перенос из него отличаются. На круге есть еще один интервал, где изменяется знаковый бит – это переходы между 0 и -1. Но там значения переносов в старший разряд и из него совпадают.

* * * **Logicly** * * *

⋆ Откройте файл **Math/SADD** с 4-разрядными сумматором и вычитателем, ко-

торые адаптированы к работе в знаковой арифметике добавлением флагов знака S и арифметического переполнения OVF. Глядя на круг на рис. 14, поупражняйтесь в знаковых сложении и вычитании.

⋆ Попытайтесь сформулировать условия арифметических переполнений при сложении и вычитании. При сложении переполнения возможны только когда знаки слагаемых совпадают, при вычитании – когда они противоположны.

Изменение знака числа связано с очень небольшими накладными расходами – поразрядное отрицание и прибавление единицы: $-B = B' + 1$. К тому же, сумматор со входом переноса способен выполнять прибавление единицы «налету», без дополнительных затрат. Это позволяет забыть про вычитатели, сведя вычитание B к сложению с $-B$.

*** **Logicly** ***

⋆ Многофункциональное арифметическое устройство в файле **Math/ALU** управляется двумя сигналами A-B и B-A и «умеет» сложить $A + B$, вычесть $A - B$ или $B - A$, обратить знак числа ($0 - A$ или $0 - B$). Все это оно делает как в беззнаковой, так и в знаковой арифметике, формируя как флаг беззнакового переполнения C, так и флаг знакового переполнения OVF с флагом знака S. Построено же оно на одном сумматоре, на входы которого «навешены» блоки инвертирования, которые либо пропускают данные напрямую, либо, при $not = 1$, инвертируют их. Поиграйте с ним, проверив его работу во всех режимах.

⋆ Обратите внимание на логику формирования флага C беззнакового переполнения. Объясните назначение сумматора по модулю два (Выполняемое сумматором «налету» обращение знака ($-B = 2^n - B$) неизменно генерирует лишний перенос.)

4.4. Сумматоры с ускоренным переносом

Сумматор со сквозным (*ripple*) переносом содержит комбинаторный путь, длина которого линейно растет с увеличением разрядности n. Путь распространения переноса от входа младшего, нулевого разряда до выхода старшего разряда номер $(n-1)$ может включать до n промежутков. При большой разрядности это существенно ограничивает быстродействие. Схемы сумматоров с ускоренным переносом решают эту проблему.

В основе техник ускоренного переноса лежит оригинальная интерпретация банальной формулы для переноса c_{n+1} на выходе полного сумматора номер $n \geq 0$ через разрядные биты слагаемых a_n, b_n и перенос c_n на входе:

$$c_{n+1} = c_n(a'_n b_n \oplus a_n b'_n) + a_n b_n = c_n p_n + g_n; \quad p_n = a'_n b_n \oplus a_n b'_n, \quad g_n = a_n b_n.$$

Интерпретация эта расщепляет перенос c_{n+1} на два флага – флаг распространения p_n (*propagation*) и флаг g_n генерации (*generation*). Обнаруживается, что перенос на выходе полного сумматора появляется по двум взаимно исключающим причинам: он может быть либо порожден в данном сумматоре (g_n), либо распространится через него ($c_n p_n$) со входа переноса. Существенно то, что флаги p_n, g_n вычисляются в каждом данном разряде локально – по битам a_n, b_n независимо от каких-либо переносов.

В итоге оказывается, что биты переноса для всех (кроме нулевого) разрядов сумматора могут быть вычислены рекуррентно:

$$c_1 = c_0 p_0 + g_0; \quad c_2 = c_1 p_1 + g_1; \quad c_3 = c_2 p_2 + g_2; \quad \ldots$$

Последовательное исключение переносов в правых частях дает:

$$c_1 = c_0 p_0 + g_0,$$

$$c_2 = c_0 p_0 p_1 + g_0 p_1 + g_1,$$

$$c_3 = c_0 p_0 p_1 p_2 + g_0 p_1 p_2 + g_1 p_2 + g_2,$$

$$c_4 = c_0 p_0 p_1 p_2 p_3 + g_0 p_1 p_2 p_3 + g_1 p_2 p_3 + g_2 p_3 + g_3$$

и так далее.

Все усложняющиеся формулы этой последовательности имеют отчетливую смысловую интерпретацию: к примеру, перенос c_3 на выходе второго разряда может появиться из-за распространения переноса c_0 на входе через разряды $0, 1, 2$ ($c_0 p_0 p_1 p_2$), или из-за распространения переноса g_0, порожденного в разряде 0 , через разряды $1, 2$ ($g_0 p_1 p_2$), или из-за распространения g_1 через разряд 2 ($g_1 p_2$). Наконец, это перенос может быть порожден в разряде 2 (g_2). Примечательно, что все эти четыре альтернативы взаимно исключают друг друга – если в каком-то разряде порождается перенос – $g = 1$, то он через него не распространяется – $p = 0$.

* * * **Logicly** * * *

⋆ Откройте файл **Math/FADD**. Изучите структуру и работу одноразрядного полного сумматора FADD, который формирует пару флагов p и g вместо одного сигнала переноса. Оцените длины комбинаторных путей от входов до выходов в числе вентилей AND, OR.

⋆ Изучите реализацию 4-разрядного сумматора, который построен из четырех блоков FADD и схемы ускоренного переноса FAST_CARRY, которая принимает сигналы p, g со всех разрядов и вычисляет переносы для них по вышеприведенным формулам.

Быстродействие подобного сумматора определяется скоростью вычисления переносов по формулам, которые прогрессивно усложняются с ростом разрядности. Обычная практика состоит в том, чтобы строить многоразрядные сумматоры по схеме сквозного переноса из относительно мало разрядных секций с ускоренным переносом.

Структура формул ускоренного переноса предоставляет возможность строить многоразрядные сумматоры по схеме удвоения, когда из двух одноразрядных полных сумматоров с флагами p, g строится 2-разрядный с такими же флагами, из двух 2-разрядных – 4-разрядный и так далее.

Формула для переноса c_2 из старшего разряда 2-разрядного сумматора приводится к виду

$$c_2 = c_0 p_0 p_1 + g_0 p_1 + g_1 = c_0 p_{01} + g_{01}; \quad p_{01} = p_0 p_1, \quad g_{01} = g_0 p_1 + g_1,$$

где p_{01} и g_{01} имеют смысл флагов распространения переноса через двухразрядный сумматор и генерации переноса в нем. Если ввести аналогичные флаги для разрядов 2 и 3

$$p_{23} = p_2 p_3, \quad g_{23} = g_2 p_3 + g_3,$$

формулу для переноса из 4-разрядного счетчика удается привести к

$$c_4 = c_0 p_0 p_1 p_2 p_3 + g_0 p_1 p_2 p_3 + g_1 p_2 p_3 + g_2 p_3 + g_3 = c_0 p_{0123} + g_{0123},$$

где

$$p_{0123} = p_0 p_1 p_2 p_3 = p_{01} p_{23},$$

$$g_{0123} = g_0p_1p_2p_3 + g_1p_2p_3 + g_2p_3 + g_3 = g_{01}p_{23} + g_{23}$$

имеют смысл флагов распространения и генерации для 4-разрядов. На следующем шаге удвоения найдем

$$c_8 = c_0p_{01234567} + g_{01234567}$$

где

$$p_{01234567} = p_{0123}p_{4567}, \quad g_{01234567} = g_{0123}p_{4567} + g_{4567}$$

и так далее.

* * * **Logicly** * * *

⋆ Откройте файл **Math/ADD2n**, в котором по схеме удвоения реализованы сумматоры разрядностей 2 и 4. Изучив внутренности блоков, уясните логику удвоения.

⋆ Выполните следующий шаг удвоения, построив 8-разрядный сумматор из пары 4-разрядных.

4.5. Комбинаторные перемножители

Умножение числа $A = \sum_{j=0}^{n-1} a_j 2^j$ на степень двойки 2^k сводится к сдвигу позиционного кода $(a_{n-1}, \ldots, a_0)$ этого числа на k позиций влево – в сторону старших разрядов: $2^k A = (a_{n-1}, \ldots, a_0) \ll k$. Это позволяет реализовать умножение A на $B = \sum_{k=0}^{n-1} b_k 2^k$ как сложение сдвинутых на k позиций копий кода A с двоичными весами b_k:

$$BA = \sum_{k=0}^{n-1} b_k 2^k \Big(\sum_{j=0}^{n-1} a_j 2^j \Big) = \sum_{k=0}^{n-1} b_k (2^k A) = \sum_{k=0}^{n-1} b_k ((a_{n-1}, \ldots, a_0) \ll k)$$

Для представления произведения двух n-разрядных чисел необходимо (и достаточно) $2n$ двоичных разрядов – максимально возможное значение произведения $(2^n - 1)(2^n - 1) = 2^{2n} - 2^{n+1} + 1$ всегда «умещается» в $2n$ разрядов.

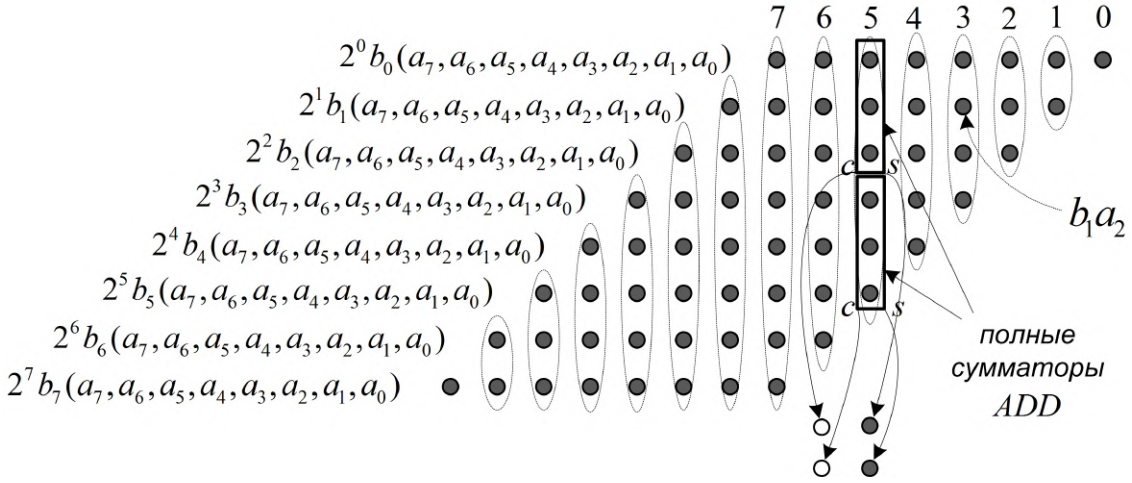


Рис. 15. Схема умножения 8-разрядных числе

Чтобы перемножить два n-разрядные числа, нужно вычислить n^2 1-разрядных произведений $b_k a_j$ (логика AND), сдвинуть k-строки получившегося прямоугольного

битового массива на k позиций влево, как показано на рис. 15, и сложить биты получившейся ромбовидной структуры по столбцам с учетом возможности многократных переносов в старшие разряды.

Оптимизация перемножителей связана с придумыванием эффективных схем выполнения этих сложений, которые требуют наименьшего числа вентилей или минимизируют длины комбинаторных путей. Банальная схема попарного сложения строк не всегда оказывается наилучшей. Экономию дает применение трехвходовых полных сумматоров ADD для сложения троек битов в столбце, рис. 15. Каждый такой сумматор заменяет три бита данного разряда на один бит суммы s и добавляет бит переноса c в следующий разряд. На рисунке два полные сумматора заменяют шесть битов разряда 5 на два бита сумм и два бита переносов. Остается сложить два бита в разряде 5 (полусумматор) и 9 битов в разряде 6 (три полных сумматора).

Перемножители разрядности $2n$ строят из n-разрядных по схеме удвоения. Позиционные коды $2n$-разрядных чисел A, B представляются суммами кодов в старших (h) и младших (l) половинках: $A = 2^n A_h + A_l$, $B = 2^n B_h + B_l$. Их можно перемножить по схеме

$$AB = 2^{2n} A_h B_h + 2^n (A_h B_l + A_l B_h) + A_l B_l.$$

Для этого потребуется четыре n-разрядные перемножителя и сумматор.

Известен алгоритм Карацубы, который позволяет сократить число перемножителей до трех. Вычисляется произведение сумм

$$(A_h + A_l)(B_h + B_l) = A_h B_h + (A_h B_l + A_l B_h) + A_l B_l.$$

При уже вычисленных произведениях $A_h B_h$, $A_l B_l$ это позволяет найти требуемую сумму произведений $(A_h B_l + A_l B_h)$ вычитанием:

$$A_h B_l + A_l B_h = (A_h + A_l)(B_h + B_l) - A_h B_h - A_l B_l.$$

Вычисления по алгоритму Карацубы приводят к довольно замысловатой структуре сложений. Они оказывается эффективными только при высоких разрядностях, когда сложность сэкономленного перемножителя оправдывает усложнение сумматора.

*** **Logicly** ***

⋆ Откройте файл **Math/MULT**. Изучите реализацию 2-разрядного перемножителя MUL2 на двух полусумматорах hADD. Исследуйте реализацию 3-разрядного перемножителя MUL3. Нарисуйте для него граф сложения девяти 1-разрядных произведений $a_j b_k$. Подумайте над иным вариантом структуры этого дерева. Нельзя ли снизить число полных и полусумматоров ?

⋆ Откройте файл **Math/MULT4** с реализацией 4-разрядного перемножителя из 2-разрядных по схеме удвоения. Вникните в схему реализации сложений. Продумайте, во что выльется попытка сэкономить один перемножитель по алгоритму Карацубы.

Отсюда один шаг до реализации 8-разрядного перемножителя на рис. 3, с которого все и начиналось.

5. Регистровая логика

Структура многих алгоритмов предполагает наличие циклов, когда реализуемые некотором функциональным блоком операции выполняются повторно с различными наборами входных данных. Среди входных данных, используемых на данном проходе

цикла, могут быть и такие, которые вычислены тем же блоком на предыдущих проходах, или зависят от результатов предыдущих вычислений. То новое, что привносят циклы в организацию вычислений – это обратные связи, когда на вход комбинаторного блока может быть подано нечто, зависящее от его выходов.

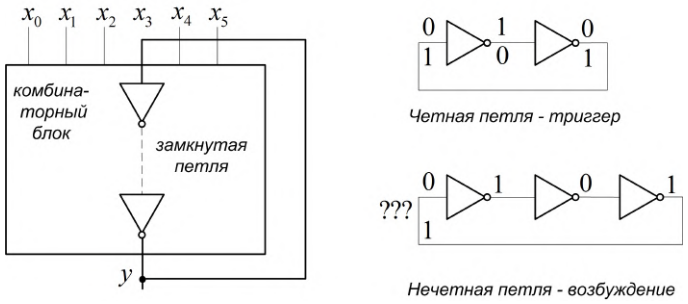


Рис. 16. Комбинаторная петля

В комбинаторной логике обратные связи категорически не допустимы. Попытка ввести их приводит к неразрешимым проблемам. Пусть, например, на вход x_3 комбинаторного блока на рис. 16 «возвращен» его выходной сигнал y. Результатом может стать образование замкнутой петли, содержащей некоторое число инверторов. Если это число четно, петля оказывается триггером с двумя устойчивыми состояниями – обе комбинации $(1, 0)$ и $(0, 1)$ значений на выходах пары инверторов на рисунке в равной мере логически допустимы. Состояние петли предсказать невозможно. Когда же число инверторов нечетно, логически допустимых состояний в петле не оказывается вовсе. Физически такая петля становится самовозбуждающимся мультивибратором.

Проблему обратных связей решает переход к регистровой логике (RTL – *Register Transfer Logic*). Достигается это одной, но радикальной мерой – непрерывное физическое время, в котором «живет» комбинаторная логика, заменяется на дискретное.

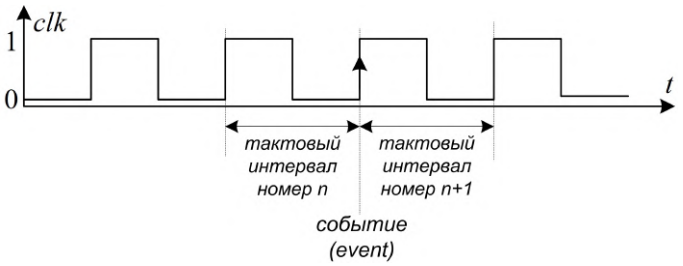


Рис. 17. Дискретное время

Дискретное время вводится единым для всей системы тактовым сигналом *clk* (*clock*), рис. 17, фронты которого – переходы $0 \to 1$ – это события (*events*), отделяющие один тактовый интервал от другого. Длительность тактового интервала T или, что то же самое, тактовая частота $F = \frac{1}{T}$ задают скорость хода часов дискретного времени – быстродействие RTL-логики.

Привязку работы цифровой логики к дискретному времени обеспечивает специальный логический примитив, именуемый динамическим d-триггером, рис. 18. Этот примитив содержит в себе элемент памяти, способный хранить один бит $q = 0/1$. На рисунке этот элемент изображает переключатель на два положения. Текущее значе-

ние хранимого в памяти бита – состояние триггера – и демонстрируется на его прямом q и инверсном q' выходах.

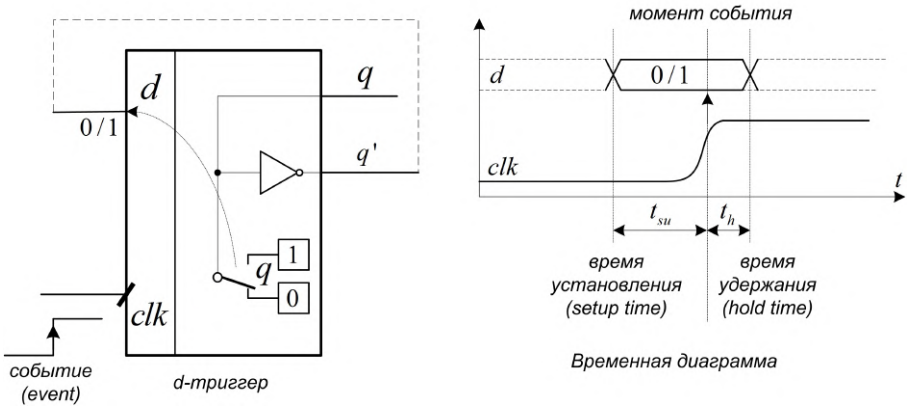


Рис. 18. Динамический d-триггер

В момент, когда на тактовом входе clk наступает событие, триггер «бросает взгляд» на свой d-вход, фиксирует там некоторое состояние – 0 или 1, и сохраняет его во внутренней памяти – переводит переключатель в соответствующее положение. В момент события информация с d-входа принимается в триггер, сохраняется в нем и начинает демонстрироваться на выходах.

Включение d-триггера в проблемную комбинаторную петлю разрывает ее, поскольку его d-вход не связан q-выходом непосредственно. Это и снимает проблемы с обратными связями в регистровой, синхронной логике. К примеру, показанная на рисунке связь инверсного выхода триггера с d-входом самовозбуждающейся петли не порождает. По каждому событию на clk триггер штатно принимает в себя инверсию своего собственного состояния, что вызывает последовательность переключений $q = 0 \to 1 \to 0 \to 1 \to \ldots$

Для нормального срабатывания физического d-триггера, рис. 18, требуется, чтобы логическое состояние на его d-входе стабилизировалось за некоторое время до наступления события, и сохранялось в течение некоторого времени после. Два основные временные параметра триггера, характеризующие его быстродействие, – это время установления t_{su} (*setup time*) и время удержания t_h (*hold time*).

Технически же d-триггеры строятся как все те же комбинаторные логические структуры с четными петлями обратной связи, обладающими двумя стационарными состояниями. Так что на самом деле регистровая логика снимает проблему обратных связей ровно тем, что упрятывает её внутрь триггеров.

Присутствие петель внутри триггеров делает их состояния после включения питания неопределенными. Это вынуждает вводить в их схемы дополнительные входы асинхронной начальной установки. Присутствие активного сигнала на входе clr (*clear*) устанавливает и удерживает триггер в состоянии 0, а на входе pre (*preset*) – в состоянии 1. Триггер выполняет свою штатную синхронную работу только когда оба сигнала на входах асинхронной установки пассивны.

Ключевую роль в понимании регистровой логики играет концепция конечного автомата, рис. 19. Линейка из m d-триггеров с объединенными входами тактирования clk составляет m-разрядный регистр с векторным входом $\bar{d} = (d_0, \ldots, d_{m-1})$ и векторным же выходом $\bar{q} = (q_0, \ldots, q_{m-1})$. Внутреннее состояние регистра $\bar{q}_n$, явленное на его выходах – это текущее состояние автомата (состояние на данном тактовом

интервале, или, в бытовом смысле, состояние сегодня). Оно поступает на входы комбинаторной логики $\bar{d}_n = F(\bar{q}_n, \bar{x})$, которая, с учетом внешних сигналов $\bar{x}$, вычисляет вектор $\bar{d}_n$ значений на d-входах. Это вектор состояния автомата в будущем, то есть завтра.

Текущий тактовый интервал закончится событием на входе clk, сменой сегодня на завтра. По этому событию вектор будущего $\bar{d}_n$ примется в регистр и станет новым настоящим: $\bar{q}_{n+1} = \bar{d}_n$. А комбинаторная логика приступит к вычислению нового будущего $\bar{d}_{n+1} = \bar{q}_{n+2}$ на послезавтра. Автомат оказывается динамической системой, состояние которого эволюционирует в дискретном времени: $\bar{q}_n \to \bar{q}_{n+1} \to \bar{q}_{n+2} \to \ldots$

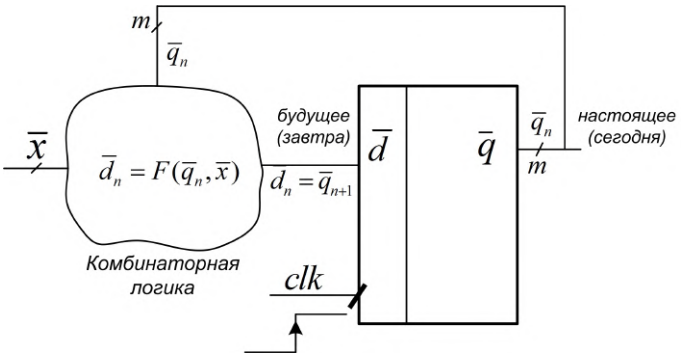


Рис. 19. Конечный автомат

После очередного события, когда состояние автомата изменилось, комбинаторной логике выдается очередное задание на вычисление вектора будущего. Завершить его выполнение требуется к моменту следующего события, то есть в течение тактового интервала. Быстродействие комбинаторной логики и определяет предельную частоту тактирования автомата. Регистровая логика не только снимает трудности с обратными связями, но и решает проблему учета временных задержек срабатывания комбинаторных блоков. Правда, решается она несколько унифицировано – всем без разбора выделяется одно и то же время на выполнение работы – тактовый интервал.

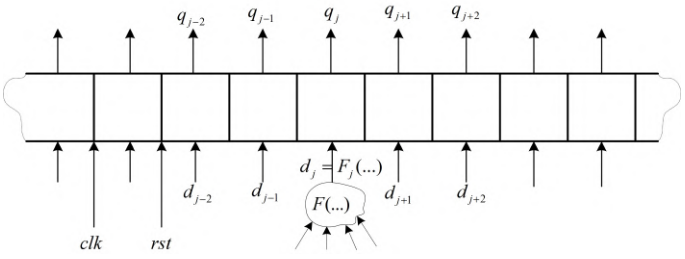


Рис. 20. RTL система

В целом, парадигму разработки синхронных (регистровых) цифровых систем можно представить как задачу программирования для спецвычислителя, показанного на рис. 20. Вам предоставлена неограниченная в обе стороны линейка d-триггеров (ячеек памяти) с единым входом тактирования clk и входом начальной установки rst. Вы можете по своему усмотрению определить набор начальных состояний триггеров, который установится в них перед началом работы по активному уровню сигнала rst. Кроме того, и это главное, предоставляется возможность навесить на каждый

из входов d_j любой комбинаторный блок от любого числа переменных, среди которых могут быть и выходные переменные q_j. Располагая этими ресурсами, требуется создать что-нибудь полезное.

Класс систем, которые допускают реализацию этими структурами невообразимо широк. Он заведомо включает все алгоритмически вычислимые по Тьюрингу функции, поскольку сама машина Тьюринга может быть смоделирована в этой структуре. В ней можно разработать ядро любого процессора и заняться потом программированием в его системе команд. А можно реализовать конвейерные структуры, реализующие алгоритмы цифровой обработки вообще и цифровой фильтрации в частности.

*** **Logicly** ***

⋆ Откройте файл **Regs/dff** с одиночным d-триггером DP. Изучите его работу. Что бывает, если на входы clr и pre поданы два активные единичные сигнала одновременно? Это состояние обычно рассматривают как запрещенное. Соедините выход q' с d-входом. Изучите результат.

⋆ Изучите структуру 4-разрядного регистра RG4 со входом сброса clr. Убедитесь в том, что по каждому событию на clk (нарастающему фронту) он принимает в себя данные в шины $\bar{d} = (d_3, d_2, d_1, d_0)$.

⋆ Изучите работу 4-разрядного регистра RG4EN с дополнительным входом разрешения en ($enable$). В отсутствие разрешения ($en = 0$) триггер блокирован – сохраняет свое состояние.

Посмотрите на его внутреннюю структуру. На каждый из d-входов добавлен мультиплексор MUX, один из входов которого ($x0$) «намертво» подключен к выходу триггера. Получается, что в запрещенном состоянии триггер на каждом такте принимает в себя свое собственное состояние, в разрешенном – данные с внешней шины $\bar{d}$.

Мультиплексор, размножающий d-вход на несколько – это типовое решение для организации нескольких режимов работы. Обратите на него внимание.

---

6. Регистровые примитивы

Набор битов, хранящихся в ячейках неограниченного регистра на рис. 20, – это глобальное состояние RTL-системы. Оно может насчитывать миллионы битов. Достижение системой нужной цели обеспечивается эволюцией этого состояния в дискретном времени, по тактам.

Уяснение логики работы такой системы становится возможным только за счет придания смысловых интерпретаций отдельным группам битов глобального состояния - регистрам конечной разрядности. В первую очередь выделяются регистры, состояния которых не связаны с логикой работы, а просто хранят обрабатываемые данные. Значения других битов определяют состав операций, предпринимаемых системой на данном такте. Именно они и входят в её текущее состояние.

При такой этой интерпретации система распадается на набор регистровых примитивов и конечный автомат управления ими. Регистровые примитивы – это «рабочие лошадки» системы. Каждый из них хранит в себе некоторый блок данных и «умеет» к концу данного такта (моменту события) выполнить над ним одну из оговоренного набора элементарных операций. Какую именно, зависит от полученного задания. Задание выдается автоматом управления в виде набора сигналов выбора режима работы.

Автомат же управления – это «руководящий орган» системы. В каждом данном состоянии (на данном такте), он раздает задания всем подчиненным регистровым

примитивам и, с учетом результатов их выполнения, переходит в то или иное следующее состояние, в котором раздается новый набор заданий, и так далее. Целесообразное поведение системы в целом достигается как результат правильно организованной коллективной работы многих участников.

Самая общая схема организации многорежимного регистрового примитива показана на рис. 21.

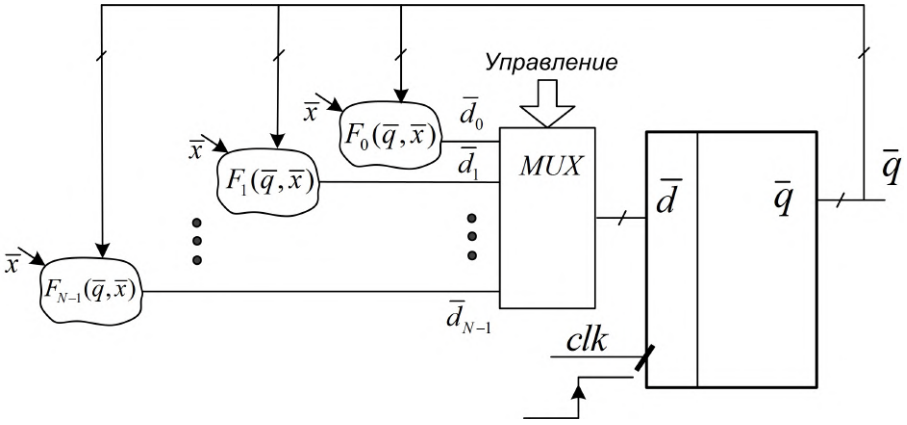


Рис. 21. Многофункциональный примитив

Это n-разрядный регистр, входная $\bar{d}$-шина которого «размножена» на N направлений $(\bar{d}_0, \ldots, \bar{d}_{N-1})$ мультиплексором MUX. К входным шинам мультиплексора подключены индивидуальные комбинаторные блоки, которые параллельно формируют N разных вариантов будущего – $\bar{d}_j = F_j(\bar{x}, \bar{q})$. Какой их них реализуется, определяет мультиплексор, на адресную шину которого приходят сигналы управления. Структура эта отнюдь не является императивом. Сигналы управления могут поступать и непосредственно на входы комбинаторных блоков, определяя один из возможных режимов их функционирования. Здесь важна общая идея.

Типовые регистровые примитивы выполняют довольно простые операции – сохраняют код неизменным, принимают его с параллельной входной шины, сдвигают влево или вправо в различных вариантах (сдвиговые регистры), прибавляют или вычитают единицу (счетчики), прибавляют код с параллельной шины (накапливающие сумматоры).

6.1. Сдвиговые регистры

Это самые простые регистровые примитивы из всех возможных. Они не содержат накакой комбинаторной логики вовсе. Чтобы сдвинуть или вообще переставить местами биты хранящегося в регистре кода, нужно просто должным образом «перепутать» провода, подключающие $\bar{q}$-выходы к $\bar{d}$-входам.

Важное применение сдвиговых регистров связано с преобразованием параллельного кода в последовательный и наоборот. Данные принимаются в регистр с параллельной шины а затем последовательно, бит за битом выталкиваются на выход сдвигами. Или данные бит за битом заносятся в сдвиговый регистр, а затем параллельно считываются с разрядных выходов. Находится и масса других полезных применений.

*** **Logicly** ***

⋆ Откройте файл **Regs/sr4** с регистром левого сдвига. После сброса в нуль по входу clr он сдвигает содержимое в сторону старших разрядов, принимая на вход последовательных данных бит din. Изучите его работу. Более простого регистрового примитива придумать невозможно.

⋆ Изменив связи, преобразуйте его в регистр левого циклического сдвига, регистр сдвига вправо со входом din, регистр правого циклического сдвига.

Мультиплексированием связей примитивный регистр сдвига превращается в многорежимный примитив, способный выполнять целый набор различных функций.

*** **Logicly** ***

⋆ Откройте файл **Regs/srlr** с реверсивным сдвиговым регистром с синхронной установкой состояния. Регистр управляется двумя сигналами. При активном высоком уровне сигнала L (Line) включается режим синхронной установки – по событию на clk принимается код с шины $(d_3, \ldots, d_0)$. При $L = 0$ выполняются сдвиги. Направление сдвига определяется сигналом $Left$. При $Left = 0$ код сдвигается на одну позицию вправо. В освобождающийся разряд q_3 заносится бит со входа SRI (*shift right input*). При $Left = 1$ реализуется левый сдвиг, в младший разряд $q0$ заносится бит со входа SLI (*shift left input*). Активный высокий сигнал clr асинхронно сбрасывает регистр в нуль.

Изучите работу регистра во всех режимах – сброс, прием с шины, правый и левый сдвиги. Ознакомьтесь с его структурой. Регистр построен на четырех триггерах DSH, d-входы которых размножены на три направления двумя мультиплекторами. Первый управляется сигналом L и подключает на d-вход бит d с внешней шины. Второй организует входы sr (*shift right*) и sl (*shift left*) для правого и левого сдвигов.

⋆ Реализуйте режимы правого ($q_0 \rightarrow SRI$) и левого ($q_3 \rightarrow SLI$) циклических сдвигов. Чтобы проводник не уходил под изображение регистра, стоит использовать промежуточные буферы.

⋆ Организуйте режим правого арифметического сдвига в знаковой арифметике – деления на два. Значение знакового разряда q_3 должно сохраняться при сдвиге. Это называют сдвигом с расширением знака (*sign extend*).

⋆ Реализуйте генератор псевдослучайной последовательности на сдвиговом регистре с обратной связью. Для этого установите режим левого сдвига и подайте на вход SLI сигнал $SLI = q_3 \oplus q_2$. Требуемый сумматор по модулю два возьмите из библиотеки. Стартовав с начального состояния $(0, 0, 0, 1)$, зафиксируйте последовательность состояний. Оцените ее период. Проделайте то же для $SLI = q_3 \oplus q_0$. Изучите последовательности состояний при $SLI = q_3 \oplus q_1$ (вместо одного цикла длиной 15 получается три цикла по 5 состояний в каждом). регистровый

6.2. Счетчики

Умение считать на пальцах – самое фундаментальное интеллектуальное достижение человечества. Не удивительно, что счетчики стали самыми распространенными регистровыми примитивами. Логически же это чуть более сложные структуры, чем сдвиговые регистры. Будущее формируется в них не совсем тривиальной комбинаторной логикой – инкрементарами или декрементарами.

* * * **Logicly** * * *

⋆ Откройте файл **Regs/cnt**. Как видно, инкрементирующий счетчик – это примитив, состоящий из регистра и инкрементора. Инкрементор принимает текущее содержимое регистра, прибавляет к нему единицу и подает результат на d-входы в качестве будущего содержимого. По каждому событию, фронту сигнала clk, будущее становится настоящим, а инкрементор приступает к прибавлению очередной единицы. Получается, что содержимое регистра увеличивается на единицу на каждом такте. То есть счетчик ведет счет числа тактов.

Полюбуйтесь на работу счетчика. Загляните во внутренности инкрементора и регистра. Там все знакомо.

⋆ Замена инкрементора на декрементор дает вычитающий, декрементрирующий счетчик. Заглянув внутрь блоков, вспомните, чем декрементор отличается от инкрементора.

⋆ Подумайте над тем, как сделать счетчик реверсивным, «научив его» считать как вперед, так и назад.

⋆ Откройте файл **Regs/cntud**, в котором реализован реверсивный счетчик со входом разрешения en. При $en = 1$ он «умеет» считать как вперед ($Up = 1$), так и назад ($Up = 0$). При $en = 0$ он сохраняет свое состояние. Изучите его работу во всех режимах. Вникните в реализацию многофункционального комбинаторного блока ID.

Примечательная особенность n-разрядного счетчика как регистрового примитива состоит в том, что все его 2^n состояний «намотаны» на один цикл максимально возможного периода. Одно из распространенных применений счетчиков – это деление частоты. При частоте тактового сигнала f, частота сигналов на разрядных выходах счетчика составляет $\frac{f}{2}, \frac{f}{4}, \frac{f}{8}, \ldots, \frac{f}{2^n}$. Когда нужно поделить частоту на число M, отличное от степени двойки, применяют счетчики с укороченным циклом, орбита которых «урезана» до M состояний.

* * * **Logicly** * * *

⋆ Откройте файл **Regs/cntm**. Примененный здесь инкрементор INCR отличается входом res синхронной установки нуля. При $res = 1$ все выходы инкрементора – нулевые. Изучите реализацию этого блока. Внутри находится стандартный инкрементор с ускоренным переносом, на выходы которого установлены логические ключи AND, которые выключаются при $res = 1$.

⋆ Убедитесь в том, что это счетчик декадный ($M = 10$)– пробегает цикл состояний от 0 до 9, измерив период сигнала RES в тактах. Изменив связи 4-входового вентиля OR, реализуйте счет по модулям $M = 3, 5, 13$.

⋆ Подключив на вход res выход eq компаратора CMP, реализуйте счет по модулю M, произвольно заданному на разрядных входах $(m_0, \ldots, m_3)$. Как все это работает при изменении M от 0 до 15 ?

Для упрощения реализации счетчиков существует специальный тип триггера – счетный T-триггер. Идея, стоящая за его структурой банальна. Она состоит в том, чтобы упрятать входящий в состав 1-разрядного инкрементора (полусумматора) сумматор по модулю два во внутренность триггера. Комбинаторная логика счетчика упрощается тогда до логики формирования переносов.

*** **Logicly** ***

⋆ Откройте файл **Regs/tff**. Изучите структуру и работу T-триггера. Это объект, который «умеет» либо переключаться, при $T = 1$, либо не переключаться. Вход T имеет смысл переноса из младшего разряда.

⋆ Изучите структуру счетчика со сквозным переносом на T-триггерах. Над линейкой триггеров «висит» цепь распространения переноса en. Первый же разрядный нуль перекрывает ее, блокируя ключ AND. Если вы «работаете» счетчике T-триггером n-го разряда, то вы получаете разрешение переключиться – перенос, если никто до вас не блокировал его. А это бывает только когда во всех младших разрядах имеются единицы.

⋆ Изучите отличия реализации счетчика с параллельным переносом.

⋆ Взяв за основу внутренности счетчиков (скопировать через карман), модифицируйте их, чтобы получить вычитающие счетчики. Реализуйте реверсивный счетчик. Нужный для этого мультиплексор имеется в библиотеке.

В отошедшей на второй план двухтактной регистровой логике, которая использовала не один, а два противофазные сигнала тактирования, базовым типом динамического триггера был JK-триггер с двумя входами управления – J и K. С переходом на отнотактную логику триггер этот не был забыт ввиду некоторых полезных особенностей. JK-триггеры строят из d-триггеров с комбинаторной логикой на входе.

По данному событию (фронту на тактовом входе) JK-триггер способен: сохранить свое состояние – $J = K = 0$, установиться в единичное состояние – $J = 1$, $K = 0$, установиться в нуль – $J = 0$, $K = 1$, и, наконец, переключиться в противоположное состояние (счетный режим) – $J = K = 1$.

*** **Logicly** ***

⋆ Откройте файл **Regs/jkff**. Изучите поведение и внутреннюю структуру JK-триггера. По своему, она гениальна. Внутри имеется мультиплексор, входные ключи которого управляются сигналами j и k. При $j = k = 0$ на d вход подключен прямой выход триггера q – режим хранения. При $j = k = 1$ – инверсный выход – счетный режим. При $j = 0, k = 1$ оба ключа закрыты – на d-входе нуль. Наконец, и это самое интересное, при $j = 1, k = 0$ оба ключа открыты. Один проводит на вход d прямой выход q, второй – инверсный. На выходе объединяющего эти пути вентиля OR всегда единица.

6.3. Накапливающий сумматор

Этот примитив отличается от счетчика тем, что на каждом такте к содержимому регистра добавляется не единица, а число B, заданное на входной параллельной шине. Делается это не инкрементором, а полноценным многоразрядным сумматором. Накапливающий сумматор выполняет функцию интегрирования входного сигнала, присутствующего на шине B.

Про постоянном значении B на выходе n-разрядного накапливающего сумматора, тактируемого с частотой f, присутствует пилообразный сигнал с частотой $\frac{f}{2^n}B$, которая может изменяться в широких пределах с шагом $\frac{f}{2^n}$. Скажем, при $f = 100$ МГц 32-разрядный накапливающий сумматор позволяет устанавливать частоту с точностью до 0.023 Гц.

Для понимания сути полезно считать, что внутри накапливающего сумматора присутствует истинный пилообразный сигнал непрерывного времени $x(t) = (Bt) \bmod 2^n$, но на выход предъявляется не он сам, а лишь поток его выборочных значений $x_j = x(t)|_{t=jT}$, взятых с шагом T, равным длительности такта, и округленных до целого числа. Когда период пилы не кратен T, на разные её периоды может приходится не одинаковое количество выборок.

Главное применение накапливающих сумматоров – это формирование местной фазы во всевозможных системах фазовой синхронизации.

*** **Logicly** ***

⋆ Откройте файл **Regs/accum** со схемой 8-разрядного накапливающего сумматора с 6-разрядным входом B установки частоты переполнений. Разберитесь в её структуре. Оцените период (в тактах) пилообразного сигнала на выходе при $B = 2^5 = 32$. Насколько изменится среднее значение этого периода при $B = 2^5 + 2 = 34$. На каком по счету периоде проявится первое сокращение числа выборок не периоде, вызванное добавлением двойки.

6.4. Игры с тактированием

Часто оказывается, что на данном тактовом интервале регистровому примитиву нечего делать – он должен просто пропустить такт, сохранив свое состояние. Прямое вмешательство в цепи тактирования в регистровой логике считается крайне нежелательным, а на некоторых платформах оно просто не возможно. Легальное решение – это применение регистров со входами разрешения en (*enable*). В отсутствие разрешения, при $en = 0$, такие регистры сохраняют свои состояния – такт пропускается.

Входы разрешения полезны и когда нужно организовывать регистровую подсистему с пониженной в k раз частотой тактирования. На входы разрешения ее регистров подается сигнал, который активируется только на каждом k-ом такте. Для формирования таких сигналов в ход идут счетчики.

*** **Logicly** ***

⋆ Откройте файл **Regs/denff** со 1-разрядным регистром со входом разрешения en – триггером. Он превращен в счетный триггер с тем, чтобы можно было просто регистрировать факты срабатывания. Все остальное – это разные варианты формирования сигнала разрешения.

Счетчик CNT2 с дешифратором DX2 на выходе – это типовое решение для формирования сигналов разрешения, возникающих один раз на периоде счетчика. Подключая ко входу разрешения разные выходы дешифратора, можно гибко управлять положением точки выдачи разрешения на этом периоде. Попробуйте разные варианты подключения.

⋆ Если объединить некоторые из выходов дешифратора по логике OR, можно организовать произвольную картину распределения точек разрешения по периоду счетчика. Попробуйте организовать разрешения в точках s_1, s_2.

⋆ Подключив ко входу res сброса счетчика выход s2 дешифратора, реализуйте деление частоты тактирования на 3 (подключить сигнал res на вход разрешения). Это универсальный способ деления частоты на произвольное целое M.

⋆ В тех случаях, когда не хотят связываться с дешифратором, применяют схемы регистрации перепадов. Блок PND на схеме выдает импульс длиной в один такт

при обнаружении на входе d положительного pd (*positive differnce*) или отрицательного nd (*negative differnce*) перепадов. Ознакомьтесь с его внутренней структурой. Посмотрите, что получается если сигналы pd, nd подать на вход разрешения.

6.5. Регистровые автоматы

На следующем уровне иерархии регистровых систем находятся регистровые автоматы. Эти структуры уже могут претендовать на роль по настоящему динамических систем. Они способны эволюционировать в дискретном времени и достигают конечного результата не к концу текущего тактового интервала, а по завершении некоторого процесса, продолжающегося в течение нескольких таков.

Мозговой центр такой системы, рис. 22, – это автомат с конечным числом внутренних состояний s. В его подчинении находится штат регистровых примитивов, способных выполнять те или иные наборы элементарных операций. Находясь в данном состоянии, автомат раздает примитивам конкретный набор заданий. Количество различных наборов как раз и определяет минимально необходимое число состояний автомата. Анализируя результаты выполнения задний, автомат принимает решение о том, каким будет его состояние на следующем такте. Архитектура автомата определяется списком состояний, функцией, преобразующей состояния в наборы заданий, и правилами перехода из одного состояния в другое. Цель его работы – организовать совместную деятельность коллектива подчиненных для наиболее эффективного решения глобальной задачи.

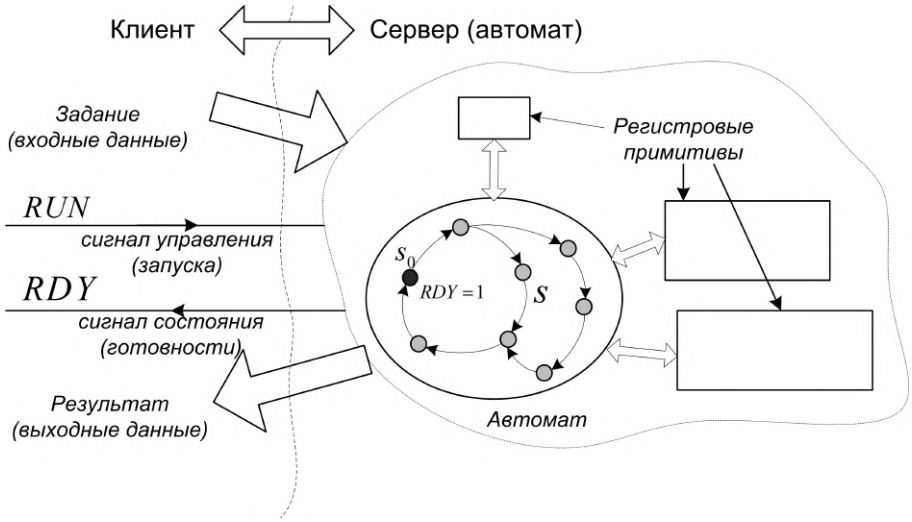


Рис. 22. Регистровый автомат

Реализуемые автоматами процессы по большей части цикличны – имеют начало, течение и конец. В таких случаях выделяется особое начальное состояние $s0$, в котором автомат не делает ничего, а просто ожидает задания от клиента – автомата верхнего уровня, пользующегося его услугами. При разработке таких, выступающих серверами автоматов требуется четко определить протокол взаимодействия клиент-сервер.

Итогом накопленного громадного опыта разработки клиент-серверных архитектур стала выработка принципа рукопожатий (*handshaking*), суть которого состоит в том, что в ходе взаимодействия клиент и сервер обмениваются парой сигналов, имеющих

смысл сигнала готовности сервера RDY (*ready*) и сигнала RUN запуска операции со стороны клиента. Сигналы это могут называться иначе и вообще не быть электрическими, а пересылаться голубиной почтой. Это не меняет сути.

Выглядит же все примитивно. Находясь в пассивном состоянии $s0$ автомат сообщает об этом сигналом готовности RDY. Клиент подготавливает задание в виде набора данных на входных шинах сервера и, удостоверившись в его готовности, запускает операцию по RUN=1. Сервер выходит из состояния $s0$ и приступает к выполнению задания. Сигнал RDY снимается. По завершении работы, сервер возвращается в состояние $s0$, сигнал готовности устанавливается вновь. Это служит сообщением клиенту о том, что результаты заказанной операции готовы и уже находятся на выходных шинах сервера.

Необходимо, чтобы клиент успел снять сигнал запуска RUN до момента завершения операции – сигнала RDY. Иначе операция будет запущена повторно. Чтобы снять это проблему, договариваются, что клиент должен снять сигнал RUN после оповещения RDY=0 о начале выполнения операции, а сервер не имеет право установить RDY=1, не дождавшись этого события. Это соглашение и завершает формулировку протокола рукопожатий, рис. 23.

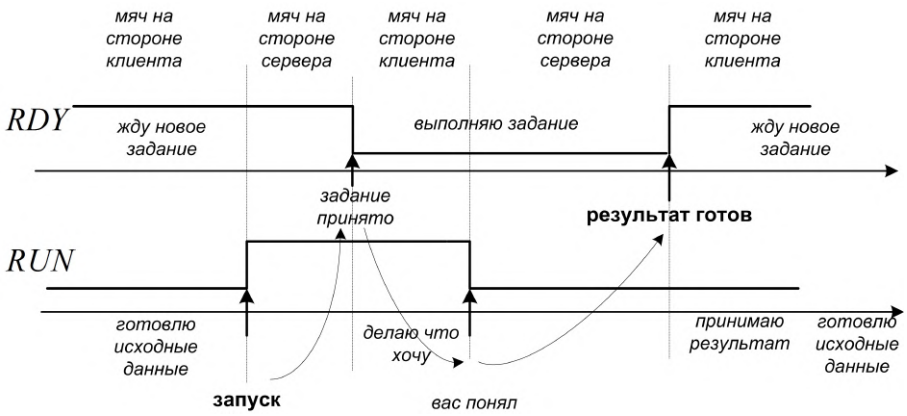


Рис. 23. Протокол рукопожатий

Действуя по этому протоколу, клиент и сервер обмениваются событиями – фронтами на линиях RDY и RUN. При каждом таком событии инициатива подобно теннисному мячу, переходит с одной стороны на другую. Обладающая инициативой (мячом) сторона может длить это состояние сколько угодно, выполняя те или иные нужные ей действия в своем темпе. Покончив с этим, она передает инициативу другой стороне, демонстрируя фронт на своей выходной линии. Когда инициатива упущена, сторона не имеет права предпринимать никаких активных действий на шинах, пока ни будет получено разрешение от противоположной стороны.

Протокол рукопожатий выглядит как нечто примитивное. И это так и есть. Его значение не в сложности, а в том, что он является итогом тщательного обдумывания проблем, возникающих при организации взаимодействия клиент-сервер. Будучи однажды сформулированным, он экономит массу ментальной энергии при выполнении конкретных разработок. Задумав реализовать подобный обмен с чистого листа, вы в конечном итоге придете к этому протоколу, преодолев по пути множество подводных камней, не бросающихся в глаза при поверхностном взгляде.

Глядя на рис. 23, можно сформулировать пару ценных выводов относительно логики работы любого регистрового автомата. Автомат должен демонстрировать

на выходе RDY факт пребывания в пассивном состоянии $s0$, выходить из этого состояния по условию $(RDY)(RUN) = 1$, и возвращаться в него по условию $(RDY')(RUN')(DONE) = 1$, где $DONE$ – некоторый флаг завершения операции, формируемый внутри автомата.

Разработка регистрового автомата – творческая задача, не предполагающая единственного решения. Прежде всего, следует определиться с рабочей силой – составом регистровых примитивов и форматами сигналов управления ими. Составив план совместных действий примитивов для достижения конечной цели, определимся с минимальным числом состояний s_j автомата. Далее придется закодировать состояния s_j двоичными кодами $\bar{q}_j$ в некотором регистре состояния, реализовать набор функций $F_k(\bar{q}_j)$, раздающих сигналы выбора операции примитиву номер k в состоянии s_j, и реализовать функции перехода $\bar{d} = F(\bar{q}_j, COND)$, которые вычисляют будущее состояние $\bar{d}$ по текущему состоянию $\bar{q}_j$ и некоторому набору условий $COND$. Именно в задачах этого рода возникает насущная потребность в минимизации булевых функций. Упрощение достигается, когда число состояний невелико. Редкая удача, когда удается обойтись всего двумя состояниями – пассивным s_0 и еще одним рабочим s_1. Оба они кодируются тогда битами 0 и 1 в 1-разрядном регистре.

Биты выбора режимов регистровых примитивов можно не вычислять по коду состояния, а просто считывать из банка памяти с состоянием в качестве входа адреса. Если ещё завести банк памяти, из которого код следующего состояния считывается по адресу, состоящему из битов текущего состояния и флагов выполнения операций, то окажется, что разработка автомата свелась к разработке микропрограммы для спецвычислителя – заполнению банков памяти «правильными» двоичными кодами. Обнаруживается, что дистанция между конструированием автоматов и программированием не так уж велика.

* * * **Logicly** * * *

⋆ Откройте файл **Regs/mulauto** с автоматом умножения 4-разрядных чисел $A = (a_3, \ldots, a_0)$ и $B = (b_3, \ldots, b_0)$. Нажмите кнопку начальной установки clr, чтобы «привести его в чувство».

Логика работы автомата проста. Исходные данные принимаются в регистры RegA и RegB. На каждом такте анализируется младший разряд регистра B (флаг LSB), и при LSB=1 код из RegA добавляется в регистр аккумулятор ACC. По событию на clk код в RegB сдвигается на единицу вправо, а в RegA – на единицу влево. Все завершается, как только в RegB не останется ни одной единицы – флаг EMP. Результат умножения виден на выходе аккумулятора.

⋆ Ознакомьтесь со структурой регистровых примитивов. Регистр RegB «умеет» принять код с параллельной шины (при $l = 1$) и сдвигать его вправо. Он формирует флаги LSB – младший разряд и EMP – все нули. Регистр RegA принимает данные с шины (при $l = 1$) и сдвигает их влево. Аккумулятор ACC сбрасывается в нуль при $rst = 1$ и прибавляет к содержимому данные из RegA при $en = 1$. При $en = 0$ и $rst = 0$ в нем ничего не происходит. Поинтересуйтесь их реализацией.

⋆ Одноразрядный регистр состояния построен на счетном триггере, который изменяет свое состояние по сигналу разрешения en. Проанализируйте логику его работы. Она проста и классична – сброс готовности RDY по условию $(RDY)(RUN)$ и установка по $(RDY')(RUN')(EMP)$.

⋆ Потренируйтесь в умножении – поучительное занятие для начинающих, если отслеживать не только результат, но и сигналы управления. Чтобы не забывать снимать сигнал RUN после снятия RDY, соедините выход RDY со входом RUN на уровне

клиента. Автомат станет перезапускаться автоматически. Чем определяется число тактов на умножение ?

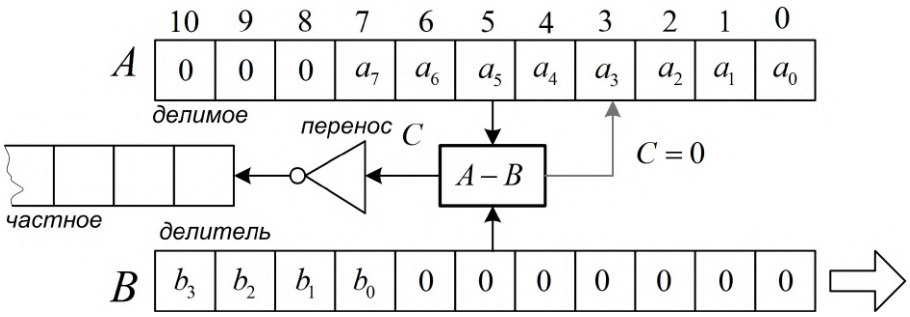


Рис. 24. Алгоритм деления

Исследуемый далее автомат реализует алгоритм деления 8-разрядного числа A (делимого) не 4-разрядное число B (делитель). По сути это привычное деление «в столбик». Коды делимого и делителя заносятся в 11-разрядные регистры как показано на рис. 24. На каждом такте вычисляется разность $A - B$. Если вычитание проходит успешно, без переноса C из старшего разряда, результат заносится в регистр A. При наличии переноса результат игнорируется, содержимое A остается неизменным. Содержимое регистра B сдвигается вправо, одновременно в регистр частного левым сдвигом заносится инверсия бита переноса. В итоге в регистре частного оказывается 8-разрядный результат, а в регистре A – 4-разрядный код остатка.

* * * **Logicly** * * *

⋆ Откройте файл **Regs/divauto** с автоматом деления. Выполните его начальную установку, нажав кнопку clr. Ознакомьтесь с режимами работы регистровых примитивов.

Регистр RegB либо принимает код с шины (при $l = 1$), либо сдвигает его вправо (при $sh = 1$), либо пребывает в режиме хранения. Примитив RegA либо принимает в свой внутренний регистр код делимого (при $l = 1$), либо, при $sub = 1$, заносит в него вычисленную разность $A - B$. По результату вычисления разности устанавливается флаг переполнения C. При $l = 1$ регистр частного RegC обнуляется, а в его младший разряд заносится единица – будущий флаг завершения операции. При $sli = 1$ выполняется левый сдвиг. Поинтересуйтесь внутренней структурой примитив.

В роли 1-разрядного регистра состояния этого автомата выступает старший разряд RegC. При начальной установке по *clr* в нем устанавливается единица. Она и служит сигналом готовности RDY. При запуске, по условию (RUN)(RDY)=1 формируется сигнал L, который вызывает инициализацию автомата – делимое и делитель заносятся в регистры RegA, RegB, а в регистре RegC устанавливается единица младшего разряда. Вызванное инициализацией снятие сигнала RDY активирует сигнал WORK, который переводит все примитивы в основной рабочий режим. Через 8 тактов сдвига единица младшего разряда в RegC достигает старшего. Это и вызывает установку сигнала готовности RDY. Цикл завершается.

Роль сигнала LOCK' вторична. Он просто предотвращает аномальное завершение работы автомата при RUN=1. Появление единичного бита в предпоследнем разряде q7 регистра RegC при условии (RUN)(RDY')=1 устанавливает LOCK'=0, а это блоки-

рует работу автомата за один такт до завершения цикла. Если автоматом управлять по протоколу, не забывая снять RUN после снятия RDY, сигнал этот никогда не активируется.

⋆ Уясните логику работы автомата. Обратите внимание на логику блокировки по условию LOCK'=0. Потренируйтесь в делении. На этом этапе можно установить режим перезапуска, подключив выход RDY ко входу RUN на стороне клиента.

⋆ Совершенный автомат деления должен формировать флаг исключения Ex (*exeption*) при делении на нуль. Выясните, как наш автомат ведет себя в этой ситуации. Попытайтесь усовершенствовать его логику так, чтобы при делении на нуль он устанавливал флаг Ex и останавливался.

---

7. А что там, внутри триггеров

Главное, что находится внутри триггера – это элемент памяти, способный хранить бит, пребывая в одном из двух стационарных состояний – нулевом или единичном. В триггерах, как и во всех структурах статической оперативной памяти, за хранение бита отвечают четные логические петли из пары соединенных кольцом инверторов. Чтобы такая петля стала полноценным триггером, нужно лишь добавить средства управления, позволяющие переключать петлю из одного состояния в другое.

Триггеры со всевозможными схемами управления представлены во всех технологических платформах. И строятся они не из вентилей, а чаще всего непосредственно из транзисторов, то есть не самом низовом уровне технологии. Тем не менее, знакомство с классическими приемами построения триггеров из вентилей полезно, поскольку дает достаточно адекватное представление об общих принципах их организации.

7.1. RS-триггер

Все начинается с простейшего RS-триггера с двумя входами управления – входом R (*Reset*) установки нуля и входом S (*Set*) установки единицы.

*** **Logicly** ***

⋆ Взгляните на RS-триггер в файле **FFs/rs**.

Когда на обоих входах управления s', r' присутствуют пассивные единицы, оба замкнутые в кольцо вентиля NAND работают как инверторы, образуя четную петлю с двумя стационарными состояниями $q = 0$ и $q = 1$. Это режим хранения. Присутствие активного нуля на входе s' триггер воспринимает как приказ переключиться в единичное состояние и немедленно исполняет его. Нуль на входе r' – это приказ встать в нулевое состояние. Вот и вся логика управления.

Состояние $s' = r' = 0$, которое означает выдачу сразу двух противоположных приказов, трактуется как запрещенное. Если его тем не менее установить, ничего ужасного не происходит. Триггер «немного сходит с ума» и демонстрирует единицы как на прямом, так q и на инверсном q' выходах.

Реальная проблема возникает лишь при переходе из запрещенного состояния $s' = r' = 0$ в режим хранения $s' = r' = 1$. Этот переход запускает процесс, известный как логическая гонка. В момент перехода оба вентиля обнаруживают у себя на входах по две единицы. Каждый из них пытается поставить на своем выходе предписанный логикой NAND нуль. Проблема в том, что обоим им это сделать не удается. Кто поставит нуль первым, тот и не позволит сделать это другому. Получается, что состояние триггера после логической гонки непредсказуемо.

⋆ Поупражняйтесь в управлении триггером, чтобы привыкнуть к его нраву.

⋆ Проведите эксперимент по наблюдению логической гонки. В каком состоянии оказывается ваш триггер после неё ?

⋆ Реализуйте триггер на двух вентилях NOR. Чем отличается логика его работы.

7.2. Добавляем вход разрешения

RS-триггер асинхронен – не имеет ни малейшего представления о существовании дискретного времени. Первая попытка довести до него факт существования тактового сигнала предпринимается установкой на входы управления логических ключей, управляемых сигналом разрешения en. При $en = 0$ входы управления оказываются отключенными. Триггер пребывает в режиме хранения.

$***$ **Logicly** $***$

⋆ Откройте файл **FFs/rsen**. Поупражняйтесь в управлении этим триггером, пока логика управления ни станет прозрачной.

⋆ Постройте аналогичную схему на четырех вентилях NOR. В чем отличие логики управления ?

Соединение входов s, r RS-триггера через инвертор приводит к чрезвычайно полезному примитиву, известному как защелка (*latch*). При $en = 1$ защелка прозрачна. Состояние на выходе q повторяет сигнал на d-входе. Переход en из единицы в нуль вызывает «защелкивание» последнего достигнутого состояния на q. Защелкнутое состояние сохраняется в триггере, пока сигнал en ни станет снова единичным.

$***$ **Logicly** $***$

⋆ Откройте файл **FFs/latch**. Поупражняйтесь в управлении защелкой.

⋆ Обратите внимание, что защелка еще не является полноценным триггером. Состояние на ее d-выходе изменяется не по событию (фронту) на входе en, а при наличии не нем высокого уровня. Попытка соединить инверсный выход q' с d-входом, чтобы организовать счетный режим, закончится самовозбуждением. Проверьте это.

⋆ Постройте аналогичную схему на вентилях NOR. В чем отличие логики управления ?

Защелки обязаны своей популярностью массовостью типовой операции приёма данных с параллельной шины в регистр. Данные выдаются на шину на некоторое время и их присутствие сопровождается сигналом записи den (*data enabled*). Если этот сигнал подать на вход en защелки, данные с d-входов окажутся сохраненными в ней до следующего сигнала den. Прозрачность защелки при $en = 1$ оказывается в этой ситуации полезным свойством – данные появляются на q-выходах защелки одновременно с их появлением на d-входах, существенно раньше того, как произойдет из защелкивание.

7.3. Двухтактные триггеры ведущий-ведомый

Исторически первые в полном смысле динамические структуры, способные изменять свое состояние только по событию (фронту), строились по принципу ведущий (*master*) – ведомый (*slave*). Заложенная в эти архитектуры идея лежит на поверхности. Чтобы избавиться от нежелательной прозрачности защелки, состоящей в том, что изменение на входе может изменять выход непосредственно, нужно поставить две защелки одну за другой тандемом и выдавать им разрешения «противофазно» –

когда одна разрешена, другая запрещена. Это исключает возможность прямого прохождения со входа на выход.

* * * **Logicly** * * *

⋆ Откройте файл **FFs/rs2t**. Разберитесь в структуре и работе двухтактного RS-триггера.

Высокий уровень тактового сигнала clk разрешает установку ведущего (*master*) триггера по входам s, r. Изменение состояния ведомого (*slave*) триггера при этом запрещено. Событие – переход из единицы в нуль на входе clk (срез тактового сигнала) блокирует состояние ведущего триггера и открывает ведомый. Защелкнутое в ведущем триггере состояние переписывается в ведомый и появляется на выходе q.

Для двухтактной логики характерно противофазное тактирование ведущего и ведомого триггеров, при котором каждый из них оказывается открытыми лишь в течение половины тактового интервала. Тем не менее, и в двухтактных регистровых автоматах на срабатывание комбинаторной логики по прежнему отводится целый такт – от одного среза тактового импульса до другого. Вычисление будущего состояния начинается сразу же после среза, как только текущее состояние стабилизируется на выходе триггера. Правда, принять результат этого вычисления триггер оказывается способным лишь спустя половину тактового интервала, когда на входе clk появится высокий уровень. Прием результата в ведущий триггер длится в течение всего следующего полутакта в режиме прозрачности до его защелкивания по срезу на clk.

⋆ Откройте файл **FFs/d2t** с двухтактным d-триггером. Поупражняйтесь в управлении им. Убедитесь в том, что этот триггер без проблем переводится в счетный режим соединением инверсного выхода q' с d-входом.

⋆ Откройте файл **FFs/jk2t** с двухтактным JK-триггером. Изучите его работу при всех четырех режимах на входах j, k.

Это самый распространенный тип триггера в двухтактной регистровой логике. По существу, это RS-триггер, входы set и $reset$ которого превращены во входы j и k добавлением перекрестных связей с выходов q' и q. В результате запрещенное для RS-триггера состояние $s = r = 1$ оказалось задействованным под счетный режим $j = k = 1$.

7.4. Однофазный динамический триггер

Двухтактная регистровая логика была оттеснена на второй план с появлением динамических структур с однофазным тактированием. Эти структуры требуют меньшего числа вентилей, но более сложны в понимании, поскольку идея их функционирования не столь тривиальна.

* * * **Logicly** * * *

⋆ Откройте файл **FFs/d1t** со схемой однофазного d-триггера. Чтобы «привести его в чувство», щелкните пару раз по входу clk.

Разумно начать с упрощенной схемы – первое приближение. Налицо тандем из одного выходного – ведомого и двух входных – ведущих RS-триггеров на вентилях NAND.

При низком уровне на входе тактирования clk ведомый триггер заведомо находится в режиме хранения (уровни 1 на обоих входах установки). При изменении же сигнала на входе d в нем просто срабатывает цепочка из двух инверторов $d{-}!d - d$.

При этом один из ведущих триггеров неизменно оказывается в запрещенном состоянии – две единицы на выходах. Какой именно, зависит от уровня на d-входе.

Тот из пары ведущих триггеров, который находится в «правильном» состоянии, просто не узнает о поступлении положительного перепада на вход clk, поскольку на другом входе соответствующего вентиля $NAND$ уже присутствует 0. У того же, который находится в запрещенном состоянии, при переходе clk в единицу на входе $NAND$ образуется две единицы. На выходе появляется нуль, что и обеспечивает установку ведомого триггера в состояние согласно действующему уровню на d.

После прохождения фронта, при $clk = 1$, ведомый триггер заведомо находится в режиме установки – присутствует нуль на одном из его входов s', r'. Если это вход установки нуля r', вход d оказывается заблокированным. Сигнал с него внутрь триггера не проходит вовсе. Если же это вход установки единицы s', сигнал с d-входа проходит внутрь. И это приводит к проблеме. Если сигнал на входе d перевести в нуль, схема перейдет в «дикое» состояние с запрещенной комбинацией $s' = r' = 0$ на входах установки мастера. На изменения на d она уже больше не реагирует, и выводится из этого состояния только по снятию clk.

⋆ Приведите триггер в «дикое» состояние. Для этого при $clk = 0$ установите $d = 1$ и примите эту единицу с d-входа в триггер, организовав фронт на clk. Теперь, не снимая $clk = 1$ обнулите d. Полюбуйтесь на результат с $q = q' = 1$. Попробуйте вывести схему из этого состояния. Объясните, почему добавление всего одной дополнительной связи решает эту проблему.

⋆ Откройте файл **FFs/d1tclr** со схемой триггера с добавленным входом асинхронного сброса clr. Разберитесь в логике её работы.

Добавить входы асинхронных установок в однофазный d-триггер не так просто, как кажется. При $clk = 0$ все просто. Ведомый триггер находится в режиме хранения и достаточно переключить в нужное состояние только его. Когда же $clk = 1$, ведомый триггера находится в режиме установки со стороны ведущих. Поэтому, приходится переключать не только ведомый, но и оба ведущие.

⋆ Чтобы удостовериться в понимании логики асинхронного сброса, добавьте в схему триггера вход асинхронной установки единицы pre. Ответ можно подсмотреть в файле **FFs/dff** со схемой полноценного однотактного динамического триггера со входами асинхронных установок.

---

8. Шины коллективного пользования

Выход обычного логического вентиля всегда находится в одном из двух возможных состояний – нулевом или единичным. Соединить два такие выхода проводником невозможно – в ситуации, когда один вентиль захочет поставить на своем выходе высокий уровень – единицу, а второй низкий уровень – нуль, произойдет логический конфликт, электрически подобный короткому замыканию шины питания на землю.

И все же возможность подключения множества выходов к общей шине коллективного пользования имеется. Ее предоставляет специальный примитив – буфер с тремя выходными состояниями. Два из них – это обычные логические 0 и 1. А третье – это новое z-состояние с высоким выходным импедансом. Когда выход находится в z-состоянии, он подобен никуда не подключенному проводнику. Такой трехстабильный буфер непременно имеет вход управления oe (*output enable*). Когда он запрещен ($oe = 0$), выход находится в третьем состоянии. При $oe = 1$ буфер открывается и передает на выход логическое состояние со своего входа.

* * * **Logicly** * * *

⋆ Откройте **Zbuf/zst**. Исследуйте работу трехстабильного буфера. Пребывание выхода в третьем z-состоянии изображается особым цветом – *high impedance signal color*.

Выходы двух или несколько таких буферов можно подключить к общей шине BUS (примитив BUS на схеме следует воспринимать как соединение проводников в точку). При таком подключении буферы приобретают статус абонентов, использующих общий ресурс (шину) в режиме разделения времени. На каждом данном временном интервале право подключения к шине может быть предоставлено только одному из них. Это достигается соблюдением логики арбитража на входах *en* разрешения буферов – в каждый данный момент разрешение может быть выдано не более чем одному из абонентов шины.

⋆ Поиграйте с шиной с двумя абонентами на схеме. Разрешите выход на шину одному и другому из них. Посмотрите, что происходит, когда абоненты начинают конфликтовать на шине.

Когда к шине коллективного пользования не подключен ни один из абонентов, логическое состояние на ней не определено. Это создает проблемы при подаче сигнала с шины на вход какого-либо вентиля. Проблема решается подключением шины к питанию (*Pull Up*) или земле (*Pull Down*) через подтягивающий резистор. Это обеспечивает присутствие на ней единичного или нулевого логического уровня по умолчанию.

⋆ Посмотрите на работу шин с *Pull Up* и *Pull Down* подтягивающими резисторами.

Самое ходовое применение трехстабильных буферов – это мультиплексирование множества выходов на общую шину. Важный пример – микросхема памяти, которая содержит, скажем, 2^{20} 8-разрядных (байтовых) ячеек. Каждая из ячеек имеет 8 выходов данных. Все они подключаются к линиям единой 8-разрядной шины данных микросхемы через трехстабильные буферы, которым раздаются разрешения *oe* сигналами выбора с выходов дешифратора адреса.

* * * **Logicly** * * *

⋆ Изучите схему мультиплексора в файле **Zbuf/zmux**, который подключает четыре сигнала x к общей шине BUS под управлением дешифратора адреса DX2 со входом разрешения.

Любая микропроцессорная система строится вокруг двунаправленной трехстабильной шины данных DBUS той или иной разрядности, по которой центральный процессор системы (CPU) либо считывает данные из регистра (порта ввода/вывода или ячейки памяти), либо записывает их в регистр.

* * * **Logicly** * * *

⋆ Откройте файл **Zbuf/cpubus** с моделью обмена данными между процессором CPU и двумя порами ввода/вывода по 2-разрядной двунаправленной шине данных. Линии шины подтянуты вверх *Pull Up*-резисторами. Сбросьте все регистры модели в нуль по *clr*.

⋆ Вникните в структуру модели. Порт считывания RD «умеет» передать на шину два входные бита d_0, d_1 по сигналу считывания rd (*read*). Порт записи WR принимает

данные с шины в регистры-защелки по сигналу записи wr ($write$). На стороне CPU работает комбинированный регистр RDWR, который «умеет» либо передать хранящиеся во внутреннем регистре данные на шину по сигналу rd, либо принять данные с шины в этот регистр по сигналу wr. Состояние внутреннего регистра CPU можно наблюдать на вспомогательных выходах d_0, d_1.

$\star$ Поупражняйтесь в организации процессов передачи данных из порта RD во внутренний регистр CPU (сигнал считывания rd), и из регистра CPU в порт WR (сигнал записи wr). Выясните, можно ли выдать сигналы записи и считывания одновременно.

Для полноты картины в модели не хватает только шины адреса, по которой процессор мог бы выбирать (адресовать) один из множества участвующих в обмене портов.
