|  |  |
| --- | --- |
| ДИСЦИЛИНА | **Схемотехника устройств компьютерных систем Часть 2** |
|  |  |
| ИНСТИТУТ | **ИТ** |
| КАФЕДРА | **вычислительной техники** |
|  |  |
| ВИД УЧЕБНОГО | **Лекция** |
| МАТЕРИАЛА |  |
| ПРЕПОДАВАТЕЛЬ | **Тарасов И.Е.** |
|  |  |
| СЕМЕСТР | 3 |
|  |  |

5. Реализация основных арифметических функций.

5.1. Содержание раздела

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

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

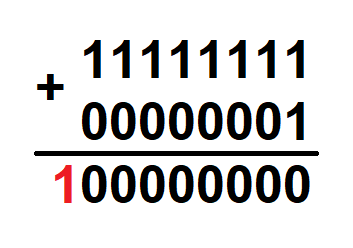
Для более сложных операций, например, трансцендентных функций, существует множество вариантов реализации. В зависимости от требований по быстродействию, точности и размеру, могут быть применены разные подходы к реализации таких функций.

5.2. Сумматор и его логическая функция

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

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

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



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

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

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

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

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

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

**1** 0000001

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

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

5.3. Особенности описания сумматора

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

0 + 0 = 00

0 + 1 = 01

1 + 0 = 01

1 + 1 = 10 (1 переносится в следующий разряд)

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

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

module sum(

input [31:0] a,

input [31:0] b,

output [31:0] c

);

assign c = a + b;

endmodule

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

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

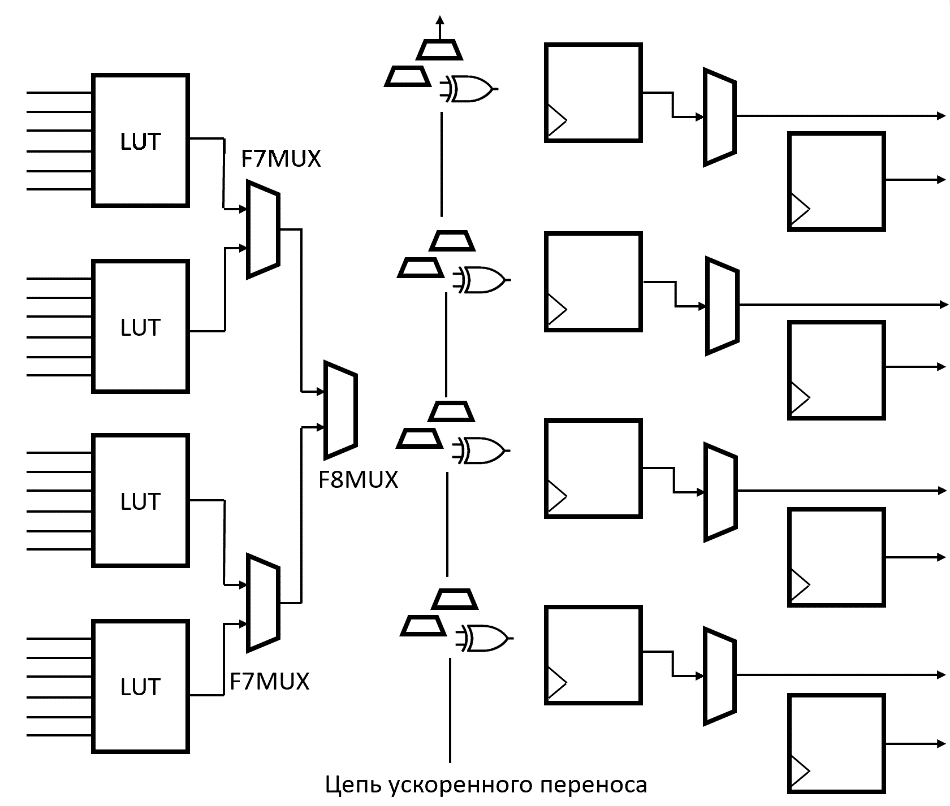
Для схемы в СБИС существует множество вариантов реализации многоразрядного сумматора. Рассмотренный пример, когда при сложении получается бит переноса, который используется следующим разрядом, является самым простым и называется ripple carry. В схемотехнике существует множество подходов, различающихся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

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

Следует обращать внимание, что некоторые схемы сумматора предусматривают многотактную работу. Такие схемы не могут быть автоматически установлены вместо сумматора, описываемого оператором assign, поскольку в этом случае подразумевается, что результат получается комбинационно (т.е. после подачи операндов результат появляется на выходе просто с некоторой задержкой, без необходимости подавать такты синхронизации).

5.4. Цепи ускоренного переноса

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

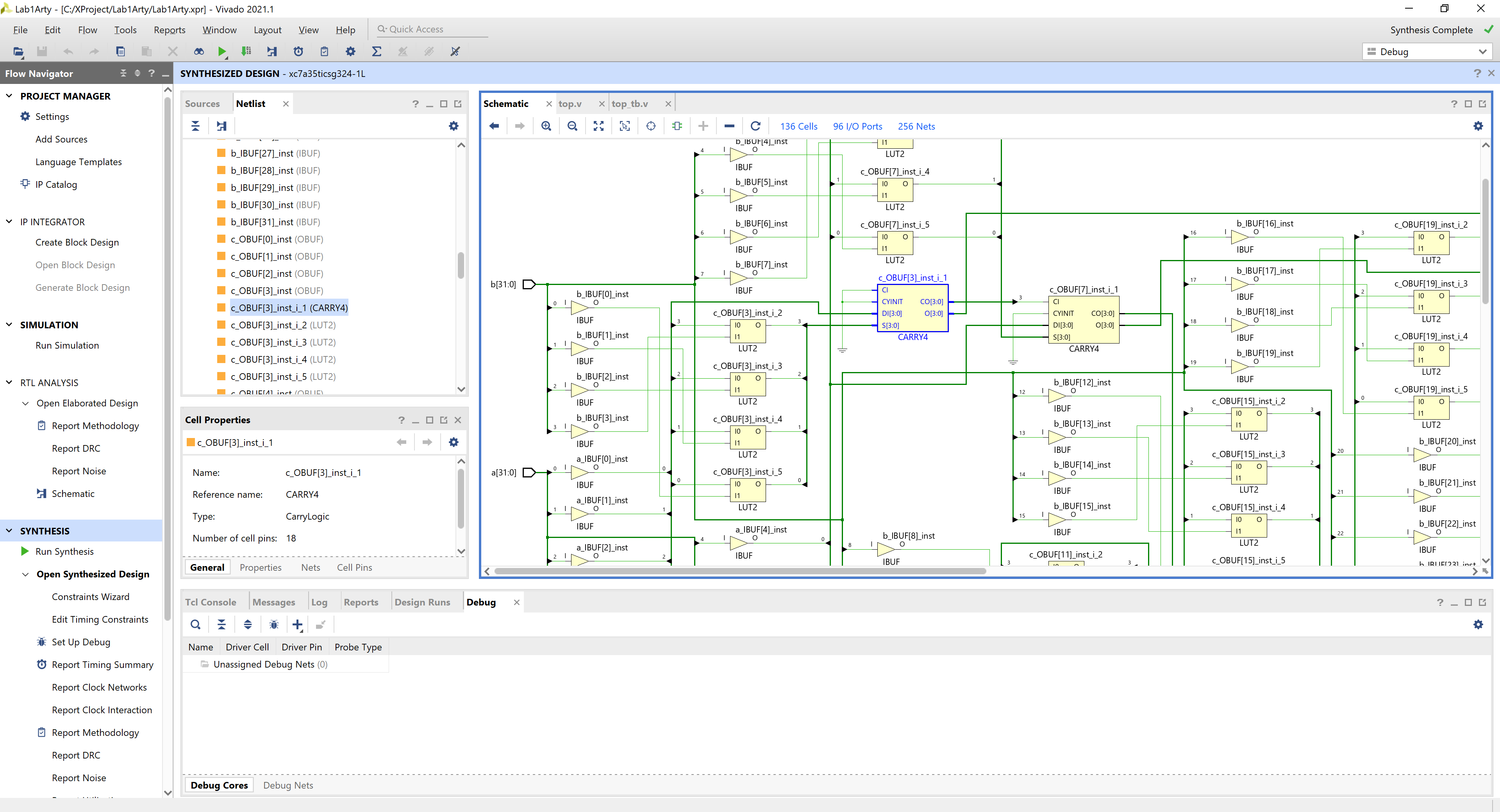


*Рисунок 5.2 Иллюстрация к реализации сложения чисел в дополнительной двоичной арифметике*

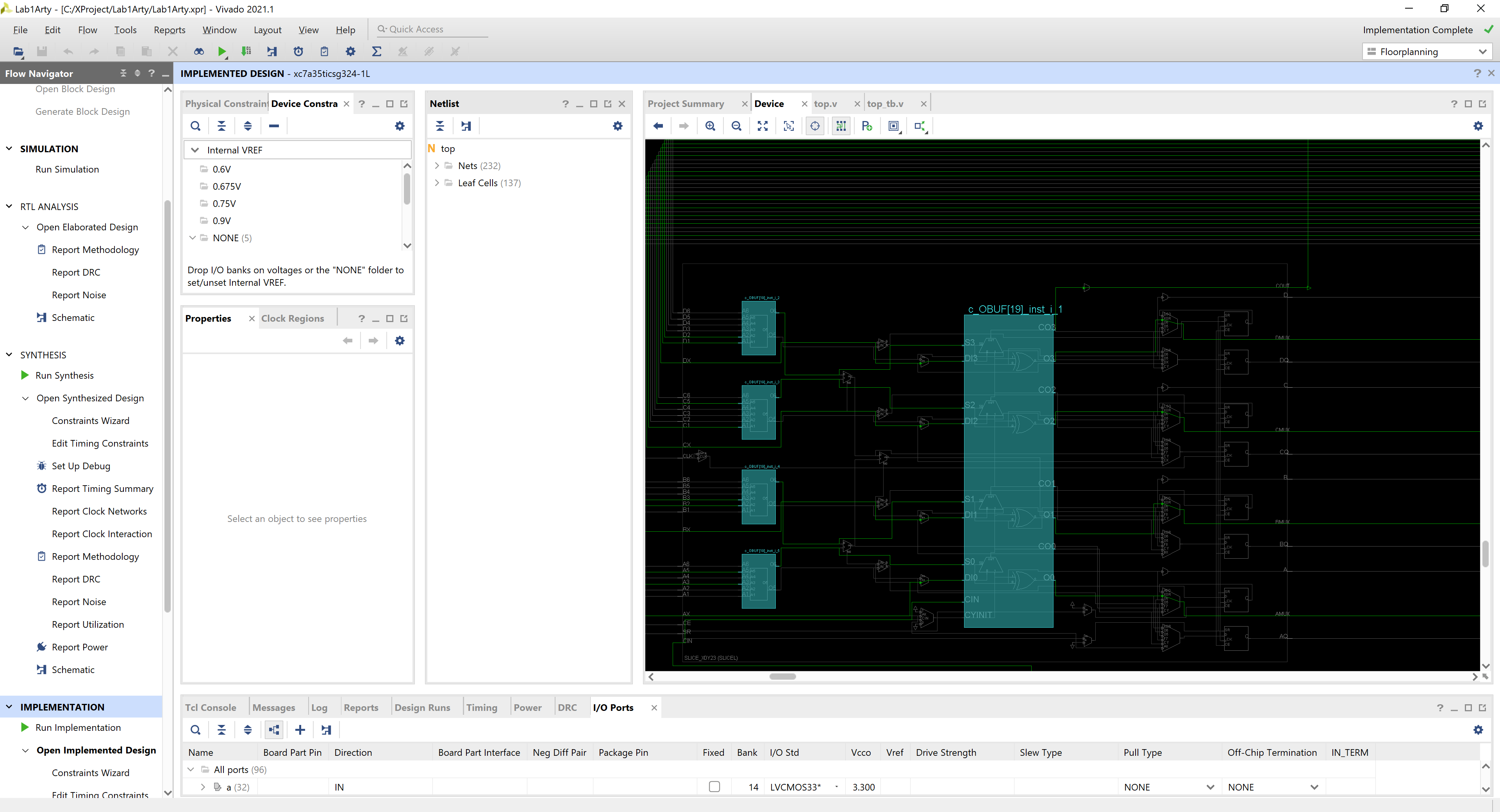
Показанные цепи будут автоматически применены по мере возможности. Именно поэтому рекомендацией по реализации сумматора является использование обычного оператора +, без попыток перейти к описанию сумматора с помощью логических функций.

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

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



*Рисунок 5.3 Реализация сумматора в ПЛИС*



*Рисунок 5.4 Реализация сумматора в топологическом представлении ПЛИС*

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

5.5. Вычитание. Смена знака.

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

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

a – b = a + (-b)

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

Это несложно проверить, например, для числа 1.

00000001 → 11111110 → 11111111

11111111 → 00000000 → 00000001

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

5.6. Умножение.

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

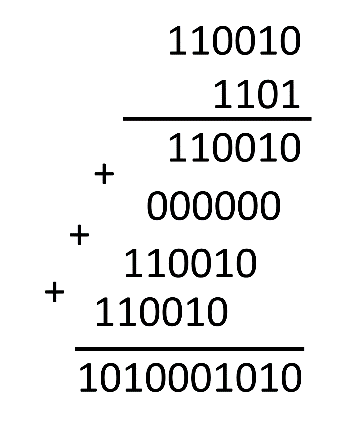
0 \* 0 = 0

0 \* 1 = 0

1 \* 0 = 0

1 \* 1 = 1

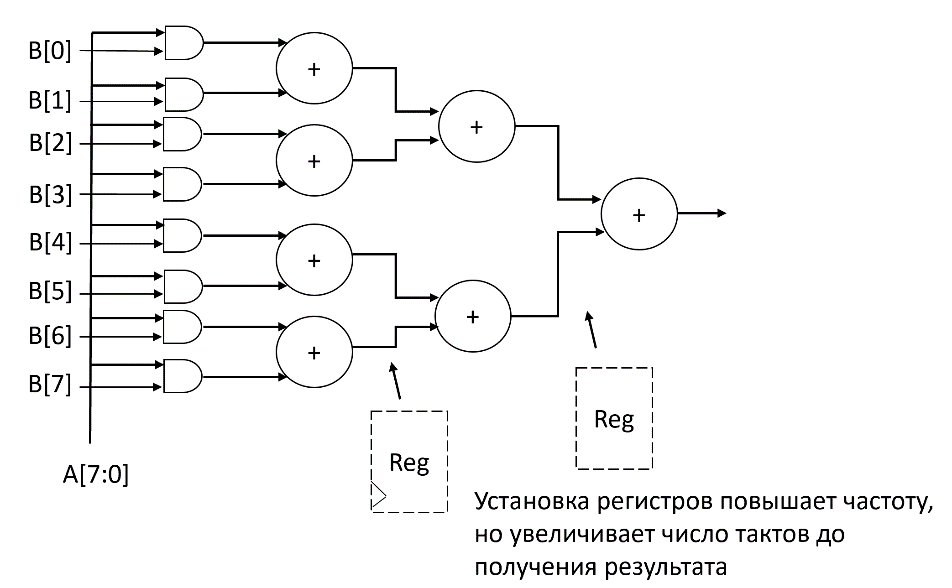
Можно видеть, что умножение однобитных чисел выполняется простым элементом И. Для многоразрядных чисел это правило несущественно изменяется – умножение любого числа на 0 дает в результате 0, а умножение на 1 – само это число.



*Рисунок 5.5 Умножение двоичных чисел в столбик*

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

Если бы сложение производилось не попарно, а последовательно, цепочка содержала бы 7 сумматоров и задержка распространения сигнала была бы больше. Дерево сумматоров позволяет реализовать умножение с log2(N) слоями сумматоров.



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

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

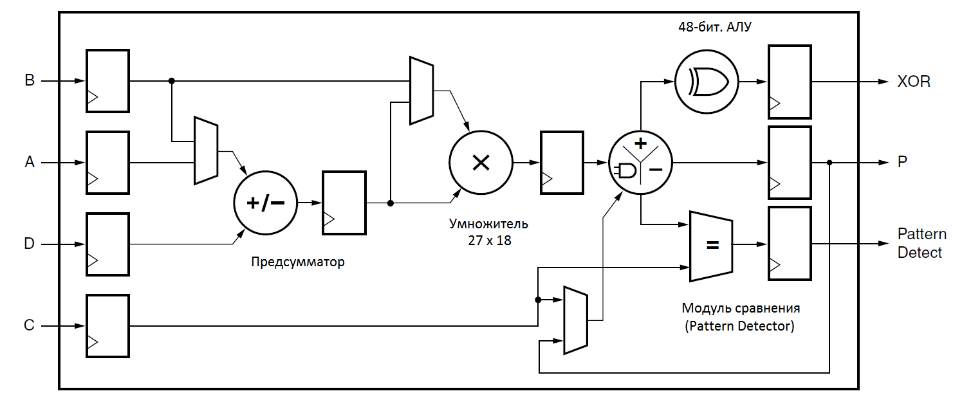
Применять ли конвейеризацию для умножения – неоднозначный вопрос. Если установить регистры после каждого слоя сумматоров, можно существенно повысить тактовую частоту этой схемы. Однако если результат умножения будет получаться через 3-4 или больше тактов, другие компоненты схемы могут простаивать. Это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

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

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

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

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



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

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

assign m = a \* b;

Также можно синтезировать умножение с накоплением:

assign sum = sum + k \* x;

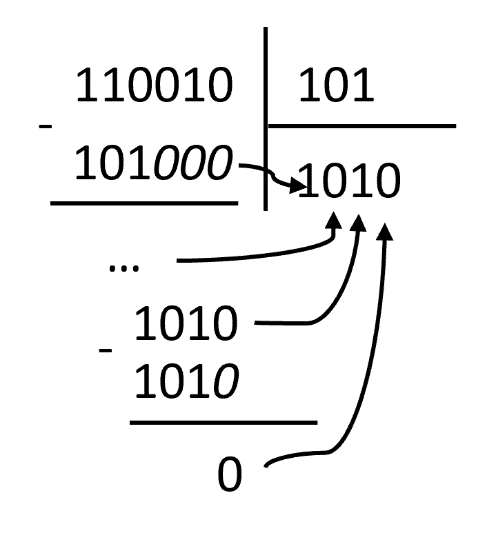
Дополнительные настройки синтезатора могут отключать использование секций DSP. Если в отчете синтезатора эти секции отсутствуют, необходимо проверить, установлено ли их использование для конкретного проекта.

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

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

5.7. Деление

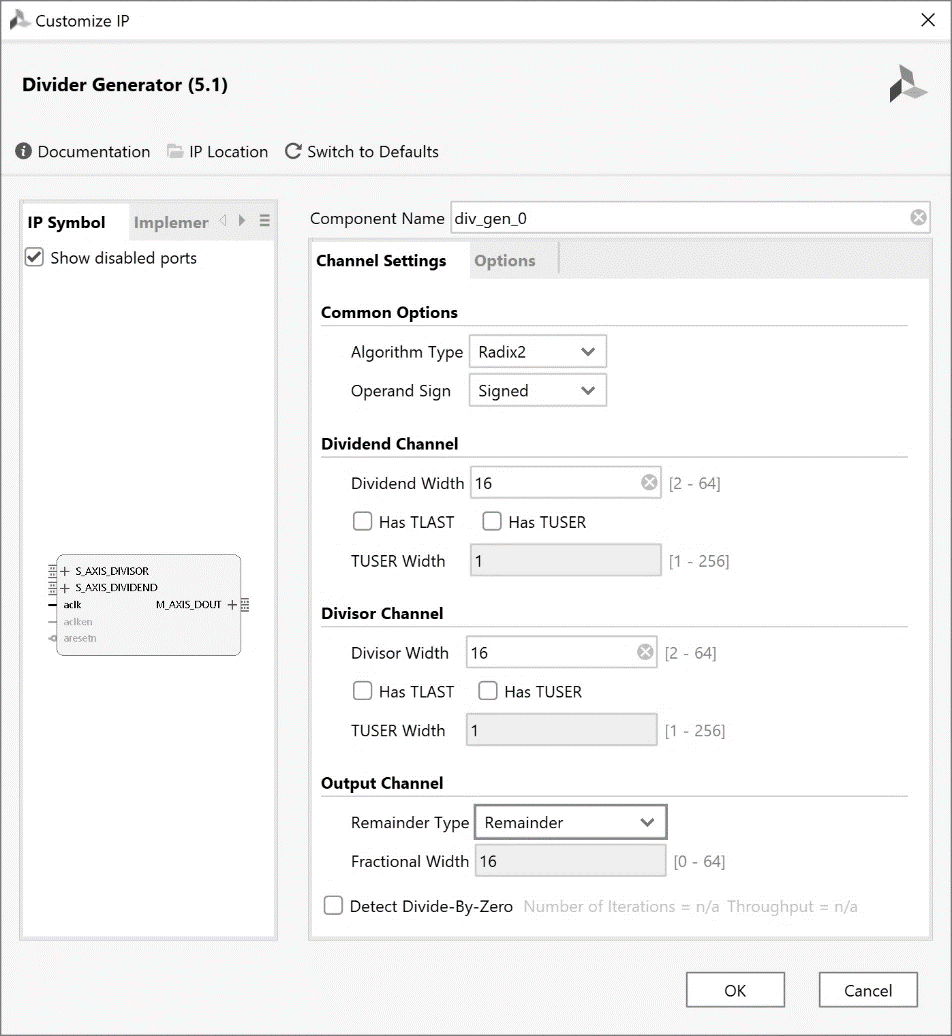
Деление иллюстрируется алгоритмом деления в столбик, как показано на рис. 5.8. В отличие от умножения в столбик, где можно рассчитывать произведения параллельно, при делении удобнее реализовать последовательный алгоритм, поскольку делимое на каждом шаге уменьшается или остается неизменным, в зависимости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



*Рисунок 5.8 Деление двоичных чисел в столбик*

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

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



*Рисунок 5.9 Генератор IP-ядер для деления двоичных чисел*

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

5.8. Числа с фиксированной точкой

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

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

При показанной нумерации разрядов двоичное число 10001.010 будет переведено в десятичный вид как 1\*16 + 1\*1 + 1 \* 1/4 = 17,25. Двоичная точка отделяет три младших разряда, поэтому число, показанное в примере, может обозначаться как 5.3, где 5 – количество разрядов до двоичной точки, а 3 – количество разрядов после нее.

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

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

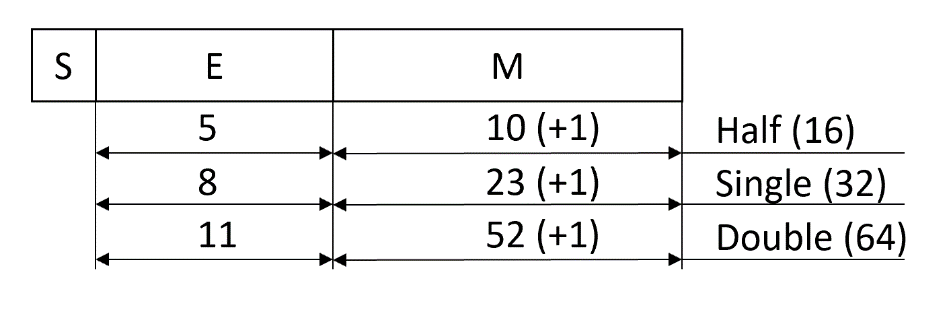
Двоичная точка сама по себе никак не представляется в аппаратуре. Разработчик самостоятельно приписывает разрядам двоичные веса, и именно разработчик ответственен за интерпретацию результата.

5.9. Числа с плавающей точкой.

Формат с фиксированной точкой дает возможность представлять дробные числа, однако у него имеется недостаток, схожий с недостатком целых чисел. Рассмотрим сложение чисел 27 и 2-7. Первое число выглядит в двоичном представлении как 10000000, а второе, с учетом разрядов после двоичной точки, как 0.00000001. Чтобы иметь возможность сложить эти числа, их необходимо записать в одинаковом формате, т.е. оба числа должны иметь 8 разрядов до точки, и 8 после нее. Поэтому для фиксированной точки возникает проблема увеличения разрядности, если необходимо представлять числа, существенно отличающимися по величине. С учетом того, что в технике числа могут иметь диапазон от величин микромира до величин, описывающих характеристики Вселенной, разрядность числа с фиксированной точкой, способного представить любую величину, которая могла бы иметь практический смысл, очень велика. Принимая приближенно, что 3 десятичных разряда (103=1000) соответствуют 10 двоичным (210=1024), и рассматривая числа от постоянной Планка до числа атомов во Вселенной, можно получить разницу десятичных порядков на уровне 110-120, что эквивалентно примерно 400 разрядам двоичного числа.

Вместо резервирования избыточного количества разрядов можно записать только старшие значащие разряды двоичного представления, и отдельно указать, какому положению соответствует старший разряд полученной записи. Например, в записи 0.00000101 старшие нули могут быть отброшены, т.к. не являются значащими. Чтобы понять, сколько разрядов было отброшено, необходимо отдельно записать это число.

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



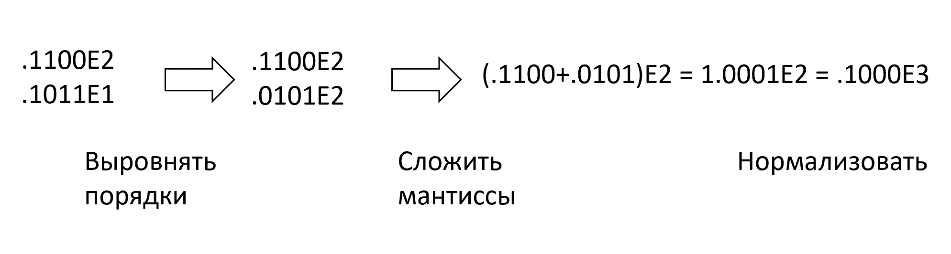
*Рисунок 5.10 Форматы чисел с плавающей точкой*

Например, для формата single порядок записывается в 8 разрядах, а мантисса – в 23. Однако с учетом того, что мантисса записывается в нормализованном виде, т.е. ее старший значащий разряд всегда равен 1, этот разряд можно не хранить. Поэтому к 23 разрядам, отведенным для хранения мантиссы, следует добавить еще один. Такой подход соблюдается для всех показанных форматов представления чисел. Считается, что мантисса лежит в диапазоне от ½ до 1. Если смещение двоичной точки не требуется, то порядок принимается равным половине возможного диапазона, т.е. для числа одинарной точности с 8-разрядным порядком отсутствие смещения представляется числом 127.

При сложении и вычитании чисел с плавающей точкой необходимо сначала совместить положение точки. Для этого мантисса числа, порядок которого меньше, сдвигается вправо на величину разницы порядков (E2-E1). После этого можно складывать или вычитать мантиссы.

Однако после сложения мантисс может оказаться, что результат не помещается в разряды, отведенные для хранения мантиссы. В этом случае необходимо выполнить нормализацию, т.е. привести мантиссу к диапазону ½..1.

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

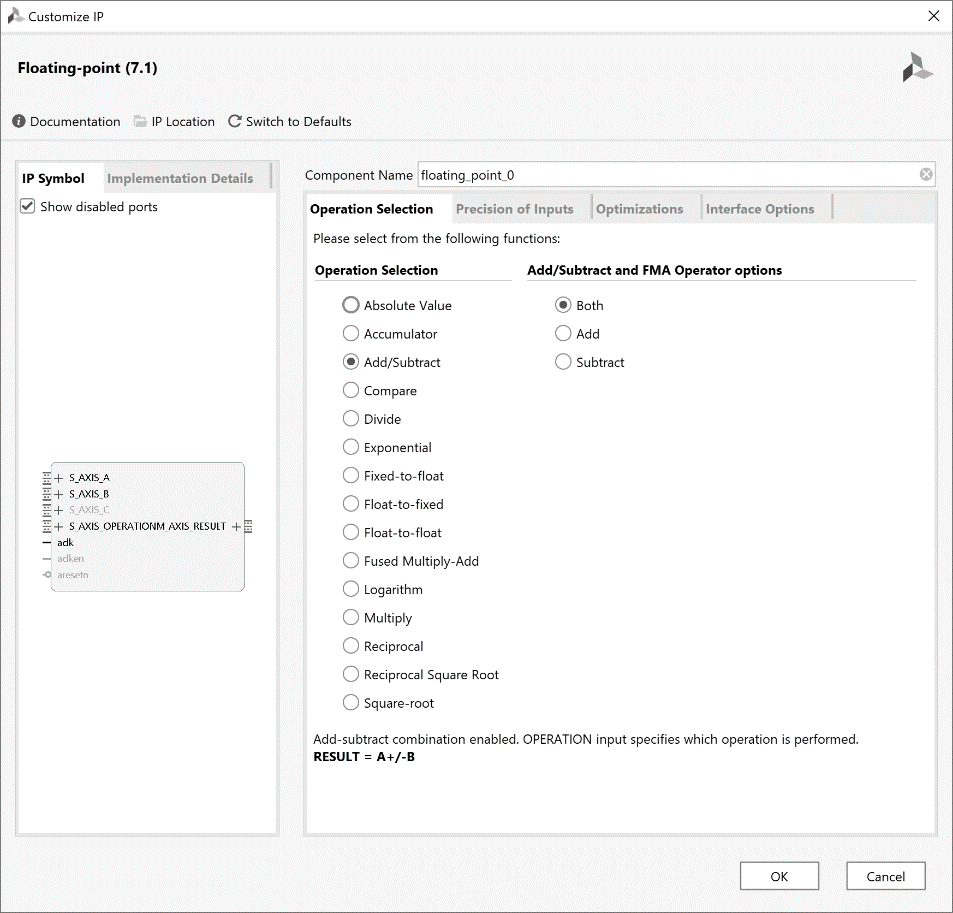


*Рисунок 5.11 Последовательность операций при сложении чисел в формате с плавающей точкой*

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

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

Числа с плавающей точкой и операции с ними описаны в широко используемом формате IEEE754. Этот формат поддерживается большинством процессоров, и содержит также информацию о правилах округления результата и хранения некоторых специальных величин – например, «бесконечность» и «не-число» (NaN, Not a Number). В силу широкой распространенности этих операций существует множество реализаций модулей для работы с числами с плавающей точкой. Их самостоятельная реализация чаще всего нецелесообразна из-за наличия оптимизированных IP-ядер, имеющих хорошие характеристики производительности и учитывающие особенности архитектуры ПЛИС. На рис. 5.12. показано диалоговое окно настройки IP-ядра для работы с числами с плавающей точкой. Это ядро поставляется в составе САПР Vivado и имеет широкий диапазон настроек, включая базовые арифметические операции, преобразования форматов и др. Поддерживаются как показанные на рис. 5.11 основные форматы представления чисел, так и формат, настраиваемый пользователем, в котором можно указать собственные значения разрядности мантиссы и порядка.



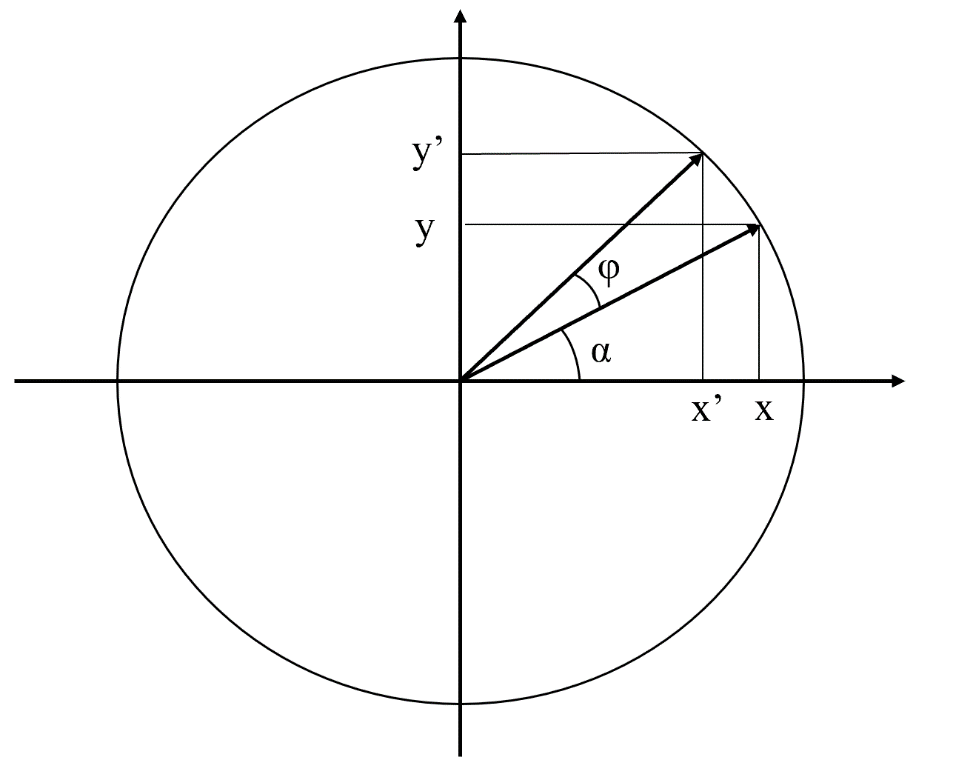
*Рисунок 5.12 Генератор IP-ядер для выполнения операций с плавающей точкой*

Можно указать важную причину для использования чисел с плавающей точкой – большой диапазон возможных значений для обрабатываемых чисел, который сложно записать в целочисленном формате или формате с фиксированной точкой. Например, такие формулы, как вычисление энергии электрона E = e ∙ U или энергии кванта E = ℏ ∙ ν оперируют со значениями, существенно различающимся по значению десятичного порядка. Заряд электрона равен 1,6 ∙ 10-19 Кл, а постоянная Планка ℏ = 6,626 ∙ 10−34Дж ∙ с, тогда как разность потенциалов и особенно частота электромагнитной волны существенно больше. Подобные формулы, очевидно, требуют чисел с плавающей точкой для представления отдельных составляющих. При умножении чисел с плавающей точкой не происходит потеря точности, так как мантиссы можно перемножать без необходимости их взаимного сдвига.

5.10. Вычисление трансцендентных функций на базе алгоритма CORDIC.

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

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



*Рисунок 5.13 Иллюстрация к вычислению трансцендентных функций*

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

x’ = cos(α + φ) = cos α ∙ cos φ – sin α ∙ sin φ

y’ = sin(α + φ) = sin α ∙ cos φ + cos α ∙ sin φ

Вынесем за скобку cos φ:

x’ = cos φ (x ∙ 1 – y ∙ tg φ)

y’ = cos φ (y ∙ 1 + x ∙ tg φ)

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

На первый взгляд, полученные выражения ничего принципиально не изменили. Однако условимся, что поворот будет производиться только на такие углы, для которых tg φ равен целой степени двойки (т.е. 1, ½. ¼, 1/8 и т.д.). В этом случае умножение x ∙ tg φ равносильно сдвигу x на 0, 1, 2, 3… позиции вправо. Аналогично выполняется вычисление y ∙ tg φ. Таким образом, умножение на синус и косинус в алгоритме оказалось заменено на умножение на 1 (благодаря вынесению за скобки cos x) и умножение на степени двойки, которое может быть выполнено сдвигом.

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

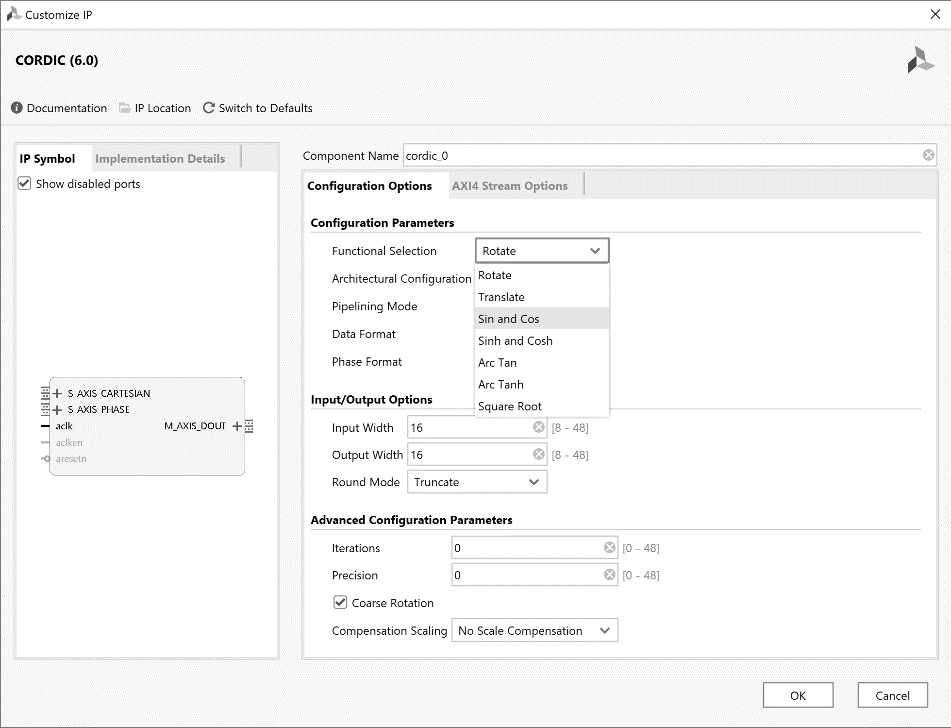
x’ = cos(α – φ) = cos α ∙ cos φ + sin α ∙ sin φ

y’ = sin(α – φ) = sin α ∙ cos φ – cos α ∙ sin φ

Имея таблицу углов, и проинициализировав x = 1, y = 0, можно повернуть угол φ максимально близко к искомому углу φ0, заданному в качестве аргумента. Если после очередного поворота φ > φ0, то на следующей итерации алгоритма поворот происходит на отрицательный угол. Таким образом, методом последовательного приближения φ устремляется φ0, а его координаты x, y естественным образом представляют косинус и синус угла φ.

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

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



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

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

Вычисление экспоненты базируется на формулах гиперболических функций:

sinh x = (ex – e–x) / 2

cosh x = (ex + e–x) / 2

Сложив левые и правые части этих уравнений, получаем:

sinh x + cosh x = ex

cosh x – sinh x = e–x

Вычисление гиперболических функций может быть произведено с помощью алгоритма CORDIC с соответствующим оптимизированным IP-ядром. Однако особенностью этого алгоритма при вычислении гиперболических функций является ограниченный диапазон возможных аргументов. Дело в том, что вычисление этих функций происходит путем замены знаков в формулах поворота угла, в результате чего конец вектора движется по гиперболической кривой с уравнением x2 – y2 = 1. При слишком больших значениях аргумента вектор, проведенный под таким углом, не пересечет гиперболическую кривую, и результат последовательных поворотов вектора окажется некорректным.

Для решения этой проблемы следует воспользоваться свойством степенных функций:

ea+b = ea ∙ eb

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

5.11. Табличная реализация функций.

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

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

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

sin(α + β) = sin α  ∙ cos β + cos α  ∙ sin β

cos(α + β) = cos α ∙  cos β – sin α  ∙ sin β

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

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

5.12. Выводы по разделу

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

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

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

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

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

Контрольные вопросы:

1. Как выглядит таблица истинности для сумматора одноразрядных операндов?

2. Что такое дополнительное двоичное представление числа? Как изменить знак такого числа?

3. Сколько сумматоров необходимо в модуле умножения для 8-разрядных операндов? Для 16-разрядных?

4. Как выполнить операцию деления для двоичных чисел?

5. Можно ли выполнить умножение целых чисел последовательно?

6. Как повернуть вектор, представленный в алгебраическом виде, на заданный угол? Как это используется в алгоритме CORDIC?

7. Какие форматы чисел удобнее для представления следующих величин (необходимо обосновать выбор между целочисленным, фиксированной точкой и плавающей точкой):

- температура воздуха в помещении;

- напряжение аккумулятора;

- суммарная энергия, прошедшая через трансформаторную подстанцию;

- механическое напряжение в элементе конструкции при расчете в САПР.