**Введение (Слайд 2)**

В данной главе продемонстрированы основные схемы в Makerchip IDE. Она дает базовое понимание фундаментальных принципов проектирования цифровой логики и способов ее проектирования с использованием TL-Verilog в Makerchip IDE.

В этой главе вы изучите:

- основные логические элементы;

- способы объединения логических элементов в функции комбинационной логики более высокого порядка, включая мультиплексоры и арифметические схемы;

- синтаксис языка TL-Verilog для выражения комбинационных и арифметических логических функций;

- последовательностную логику и способы ее описания на TL-Verilog;

- способы отладки комбинационных схем в Makerchip, включая использование возможностей визуальной отладки, уникальных для платформы Makerchip.

**Комбинационная логика (Слайд 3)**

**Логические элементы**

В цифровых схемах провода (источники сигналов) стабилизируются в одном из двух напряжений: высоком (VDD) или низком (VSS или земля). Таким образом, провод несет логическое значение, где высокое значение можно рассматривать как 1/истина/включено, а низкое как 0/ложь/выключено и т.д. Это обеспечивает абстракцию для составления логических функций более высокого порядка с предсказуемым поведением.

Логические элементы – это основные строительные блоки для создания логических функций. Таблица ниже описывает основные логические элементы. Их функция определяется таблицами истинности, которые показывают значение выхода X для каждой комбинации входных значений A и B.

Обратите внимание на следующее:

- элементы «И» (AND) и «ИЛИ» (OR) соответствуют их русским значениям;

- маленький кружок (или «пузырек») на выходе некоторых элементов означает инвертированный выход;

- в элементе «XOR», он же «исключающее ИЛИ» и «XNOR», он же «исключающее ИЛИ-НЕ», «исключающее» означает «не оба»;

- логические элементы могут быть соединены в различные комбинации для создания логических функций более высокого порядка, как в приведенной ниже схеме (это схема, известная как «полный сумматор»);

- схемы, состоящие только из логических элементов, называются «комбинационными».

**Синтаксис TL-Verilog: философия (Слайд 4)**

TL-Verilog очень строго определяет синтаксис. Это может вызывать трудности у новичков с их собственными предпочтениями в стиле кодирования. Но эта строгость имеет важное преимущество. В промышленности с кодом работают много людей из разных команд, и такая строгость обеспечивает единообразие. Спецификацию синтаксиса TL-Verilog можно найти в меню «Help» («Помощь») Makerchip, но наиболее важные аспекты будут рассмотрены далее.

Синтаксис TL-Verilog: логические операции

Логические операции в разных отраслях науки обозначаются по-разному. На следующей схеме показаны некоторые из этих математических обозначений, а также операторы TL-Verilog (которые аналогичны операторам Verilog) для основных логических элементов (вентилей).

Для группировки операций, чтобы сформировать более сложные логические функции, можно использовать круглые скобки. Если выражение растягивается на несколько строк, эти строки должны иметь больший отступ, чем первая строка. Выражения всегда должны заканчиваться точкой с запятой. Всегда ставьте пробел до и после «=». Например:

$foo = ( $val1 && $val2) || (! $val1 && ! $val2)

**Синтаксис TL-Verilog: отступы и имена сигналов (Слайд 5)**

В TL-Verilog (внутри блоков кода \TLV) отступы и пробелы имеют смысл. Табуляции (которые не имеют определенного поведения) не допускаются. Каждый уровень отступа равен 3 пробелам (редактор Makerchip помогает придерживаться этим правилам).

Если вы будете придерживаться предложенных в этом курсе имен сигналов, у вас никогда не возникнет проблем. Тем, кто захочет немного отклониться от сценария, следует знать, что TL Verilog накладывает ограничения на имена сигналов. В то время как языки обычно оставляют выбор между camel-case регистром и подчеркиванием на усмотрение кодовых конвенций, TL Verilog принуждает к такому выбору.

Ограничения наименований служат нескольким целям:

- они обеспечивают согласованность;

- они различают типы;

- при обработке TL-Verilog в Verilog автоматически генерируемая логика может использовать имена сигналов Verilog, которые не конфликтуют с именами сигналов, созданными разработчиком.

В частности, имена сигналов TL-Verilog:

- имеют префикс «$»;

- состоят из лексем, разделенных символами подчеркивания, где каждая лексема представляет собой строку символов нижнего регистра, за которыми следуют ноль или более цифр;

начинаются как минимум с двух строчных символов.

Так, например, это корректные варианты наименований переменных:

- $my\_sig;

$val1.

А эти нет:

- $a;

- $Sig (на самом деле это «сигнал состояния», который в этом курсе использоваться не будет);

- $val\_1;

В других местах вы можете встретить термин «pipesignal» относящийся к сигналам в TL Verilog. Различие между сигналами Verilog и сигналами «pipesignals» TL-Verilog в этом курсе не имеет значения, и мы будем просто использовать сокращенный термин «сигналы».

**Арифметическая логика (Слайд 7)**

**Концепция**

В то время как отдельные сигналы (или «биты») хранят одно из двух значений в цифровой схеме, набор из N сигналов (называемых «шиной» или «вектором») позволяет описать до 2^N возможных значений.

Привычный для всех формат представления чисел – в десятичной системе счисления (СС). В десятичной СС мы используем десять цифр (0-9), и когда мы считаем до последней доступной цифры (9), мы возвращаемся к 0 и увеличиваем значение следующего разряда, который равен десяти. Основание десять очень неудобно для цифровой логики. Двоичное основание или любая степень двойки (4, 8, 16) гораздо более естественна. В двоичной или бинарной СС есть цифры 0 и 1. Каждая из них может быть представлена битом. Шестнадцатеричная СС также довольно распространена. В ней используются цифры 0-9 и символы A-F (для чисел от 10 до 15). Одна шестнадцатеричная цифра может быть представлена 4 битами. В таблице ниже показаны значения от нуля до двадцати в десятичной, двоичной и шестнадцатеричной СС.

Языки HDL поддерживают «векторные» сигналы, которые содержат несколько битов. Векторы могут использоваться для представления двоично-кодированных (знаковых или беззнаковых) целых значений. Например, 5-битный вектор может хранить значение 13 в виде битового значения 01101. HDL поддерживают арифметические операции, такие как сложение, которые работают с этими битовыми векторными значениями.

**Синтаксис TL-Verilog (Слайд 8)**

В TL-Verilog наиболее распространенными типами данных являются битовые переменные (которые вы использовали в предыдущей лабораторной работе) и битовые векторы. Вектор объявляется путем указания его битового диапазона следующим образом:

$vect[7:0] = ....;

Битовые диапазоны обычно не используются в правой части выражения. Используется вектор с указанием нужного диапазон битов

В Verilog и TL-Verilog арифметические операции, такие как +, -, \*, /, и % (остаток от деления или деление по модулю) могут использоваться с векторами. Без этих операторов схему сумматора пришлось бы строить путем повторения полной схемы сумматора (которая была рассмотрена ранее) для каждой позиции бита в сумматоре.

**Мультиплексор (Слайд 9)**

**Концепция**

Одной из наиболее важных логических функций является мультиплексор (или MUX), изображенный на слайде.

Мультиплексор осуществляет выбор между двумя или более входами (которые могут быть двоичными значениями, векторами или любыми другими типами данных). Линия выбора определяет вход, который нужно передать на выход. Чаще всего линией выбора является либо двоично-кодированный индекс входа, либо «one-hot» вектор, в котором каждый бит вектора соответствует одному входу (для выбора соответствующего входного значения подается сигнал на один и только один из битов).

MUX, изображенный на слайде, может быть построен из базовых логических вентилей, как показано ниже. Эту реализацию можно описать как «задать выходе X1, если он выбран (по сигналу S == 1) ИЛИ X2 (если S == 0)».

**Синтаксис TL-Verilog (Слайд 10)**

Verilog предоставляет не менее шести способов кодирования мультиплексора, каждый из которых имеет свои преимущества и недостатки. В TL-Verilog предпочтительно использовать тернарный оператор условия (?:), мы будем придерживаться такого подхода на протяжении всего курса. В своей простейшей форме тернарный оператор выглядит следующим образом:«$out = $sel ? $in1 : $in0;»

Это выражение можно интерпретировать так:

«$out соответствует: $in1 если выбран $sel ($sel == 1), иначе $in0 (если $sel == 0)».

Тернарный оператор можно объединить в цепочку для реализации мультиплексоров с более чем двумя входными значениями, из которых нужно выбирать. Сами входы могут быть векторами. Для согласованности мы будем использовать очень специфическое форматирование кода, показанное ниже для четырехканального мультиплексора шириной 8 бит с однократным выбором (здесь сигналы $in0-3 являются 8-битными векторами).

$out[7:0] =

$sel[3]

? $in3 :

$sel[2]

? $in2 :

$sel[1]

? $in1 :

//default

$in0;

Это выражение устанавливает приоритет сверху вниз. Так, если выбран «$sel[3]», то «$in3» будет соединен с выходом независимо от других битов «$sel». Ниже показана его буквальная интерпретация, а также его одноэлементное представление (которое неоднозначно в отношении приоритета).

**Литералы и конкатенация (Слайд 11)**

**Концепция и синтаксис**

Литералы

Выражение: «$foo[7:0] = 6;» определяет «$foo» для хранения постоянного значения 6. В данном случае 6 принудительно разделено на восемь бит присвоением. Часто необходимо явно указать ширину литерала:

«$foo[7:0] = 8'd6;»

Данное выражение явно присваивает «$foo» 8-битное десятичное («d») значение числа 6 («'» – символ одинарной кавычки). Другие способы записать тоже самое:

$foo[7:0] = 8'b110; // 8-битная двоичная шестерка

или

$foo[7:0] = 8'h6; // 8-битная шестерка в шестнадцатеричной СС

Конкатенация

Конкатенация битовых векторов – это объединение двоичных векторов один за другим с образованием вектора большей разрядности. Пример синтаксиса, реализующего операцию конкатенации:

$word[15:0] = {$upper\_byte, $lower\_byte}

**Структура файла и маршрут выполнения (Слайд 12)**

**Концепты**

Исходный файл TL-Verilog сначала обрабатывается макропрепроцессором M4. Именно так была импортирована и потом запущена визуализация калькулятора. Полученный файл TL-Verilog обрабатывается SystemVerilog или Verilog инструментом SandPiper™ компании Redwood EDA. Существует возможность просмотреть, как код обрабатывается в Verilog с помощью SandPiper, открыв «Show Verilog» в меню «E» редактора.

Verilator это симулятор который выполняет симуляцию путем преобразования Verilog в компилируемое описание на С++ и затем выполнение. Этот симулятор запускается для получения данных трассировки, которые можно просмотреть в программе отображения временных диаграмм.

На вкладке LOG отображаются результаты работы всех этих инструментов. Вывод M4 и SandPiper отображается синим цветом, а вывод Verilator и его итоговая симуляция – черным.

Функции TL-Verilog используются для определения логики внутри модуля (System)Verilog. Комментарии к коду ниже объясняют части исходного файла TL-Verilog, структурированные для использования в Makerchip.

Для тех, кто интересуется мотивацией такой структуры файлов, необходимо понять стратегию развития TL-Verilog из Verilog. Конечной целью является создание нового языка моделирования, философски отличного от Verilog во всех отношениях. Это произойдет в течение следующего десятилетия или десятилетий. Пока же сообщество работаем над этим постепенно, опираясь на Verilog в качестве рабочей отправной точки, и с языком TL-Verilog как расширением Verilog. Это наслоение также обеспечивает последовательный и постепенный путь миграции. И по мере развития инструментов всегда можно вернуться к Verilog.

Также следует отметить тот факт, что TL-Verilog – это реализация TL-X на языке Verilog, расширение языка, определенное для использования поверх любого HDL для расширения его возможностей на уровне транзакций. Таким образом, существует возможность миграции с любого поддерживаемого HDL (на момент подготовки этого материала Verilog является единственным поддерживаемым HDL).

Благодаря использованию синтаксиса TL-Verilog только в определениях модулей, инструменты на базе Verilog, которые используются для сшивания связей между модулями, могут оставаться в неведении об использовании TL-Verilog. Внутри модуля Verilog можно использовать другие формы модульности и иерархии, характерные для TL-Verilog.

Весь код, который разрабатывается в этом курсе, находится внутри области \TLV. Теперь вы понимаете, как это связано с инструментами на базе Verilog.

**Последовательная логика (Слайд 13)**

**Концепция тактирования и триггеры**

В последовательной логике вводится понятие тактового сигнала.

Тактовые импульсы проходят через всю схему к триггерам, которые упорядочивают логику. Триггеры бывают разных типов, но самым простым и распространенным является «D-триггер с управлением по переднему фронту сигнала» (в курсе будет рассмотрен именно этот триггер). Они передают значение с входа на выход, но только при приходе переднего фронта тактового сигнала. Триггеры хранят значения выходных сигналов до следующего переднего фронта тактового сигнала.

Хотя существуют триггеры, которые работают по заднему фронту тактового сигнала, разрабатываемые в данном курсе схемы будут работать только по переднему фронту. Кроме того, существуют триггеры, которые включают в себя логические функции. Поскольку в данном курсе будем используются только D-триггеры, в дальнейшем будем называть их просто триггерами.

**Примеры последовательной логики и сигнал сброса (Слайд 14)**

Прежде чем перейти к теоретическим рассуждениям о последовательной логике, изучим схему, которая вычисляет последовательность Фибоначчи. Каждое число в последовательности Фибоначчи является суммой двух предыдущих чисел: 1, 1, 2, 3, 5, 8, 13, ... Эта схема будет вычислять следующее число в последовательности.

С каждым передним фронтом тактового сигнала значения будут распространяться через триггеры, сдвигая один триггер вправо, создавая следующую временную диаграмму.

Данная схема неполная. В ней не хватает входа сброса. Нужен способ инициализировать схему двумя единицами, чтобы начать генерацию последовательности.

В отличие от комбинационных схем, где выходные значения являются исключительно функцией входных значений, последовательные схемы имеют внутреннее состояние. Каждая последовательная схема должна иметь возможность перейти в известное состояние «сброса» («reset»).

Поэтому в каждой такой схеме должен быть сигнал «сброса», который отвечает за это. Схема должна быть спроектирована таким образом, чтобы при подаче сигнала сброса схема стабилизировалась в известном состоянии сброса.

Самый простой подход к обеспечению возможности сброса заключается в том, чтобы при подаче сигнала сброса каждый триггер устанавливался в значение сброса. Эта методология используется некоторыми группами разработчиков, но, если важны площадь и энергопотребление, можно сделать иначе.

В рассматриваемой схеме Фибоначчи достаточно сбросить только первый триггер, как это показано на нижнем рисунке.

**Синтаксис TL-Verilog (Слайд 15)**

В TL-Verilog можно ссылаться на предыдущую и предпоследнюю версии $num как >>1$num и >>2$num. В отличие от RTL, в TL-проектировании нет необходимости назначать их в явном виде. Они неявно доступны для использования.

Поэтому рассматриваемый пример схемы последовательности Фибоначчи можно задать следующим выражением:

$num[31:0] = $reset ? 1 : (>>1$num + >>2$num)