Введение

Современные системы автоматизированного проектирования, анализа и синтеза динамических объектов, управления движением требуют выполнения до нескольких миллиардов операций в секунду. Во многих случаях алгоритмы предполагают многократное повторение относительно простых действий. Значительного повышения эффективности таких приложений по соотношению производительности и стоимости можно достичь, возлагая эти функции на специализированные блоки, реализованные на принципах структурной реализации вычислений. Особенно расширились возможности использования структурного подхода (иначе – аппаратной реализации алгоритмов) в связи с широким внедрением микросхем программируемой логики и последнего достижения в этой области – систем на кристалле [2, 9, 10].

Важнейшим условием эффективного проектирования и быстрой доставки на рынок специализированных средств является применение систем автоматизированного проектирования (САПР), требующих, в свою очередь, эффективных, наглядных, управляемых и контролируемых средств описания проекта. Этим условиям в значительной мере отвечает представление проекта в форме текстового описания алгоритма его функционирования. Поэтому в последние годы проектирование аппаратных средств, базирующееся на языковом представлении в рамках проблемно-ориентированных языков, получает все большее распространение.

Программа на языке проектирования создается для достижения определенной цели. Существует три основных назначения программ: для спецификации объекта, для моделирования поведения объекта, для последующего синтеза объекта в заданном базисе. Одна и та же программа может использоваться как основа всех трех назначений или любого их сочетания.

В настоящем пособие авторы пытались осветить эти аспекты и предложить комплекс практических работ для получения навыков проектирования. Пособие не претендует на систематическое изложение языков проектирования аппаратуры VerilogHDL и SystemVerilog, с которыми читатель может ознакомиться, например, в работах [2, 8, 13, 16, 17]. Здесь основное внимание уделяется вопросам интерпретации базовых конструкций языка при реализации проекта в аппаратуре, равно как представлению типовых дискретных устройств на VerilogHDL, а также современным подходам к тестированию проектов.

Для практического освоения материала настоящего пособия рекомендуется выполнять интерпретацию программ в пакете моделирования. Пособие опирается на пакет QuestaSim v.6.3, общие правила работы с которым имеются в Приложении 1. Для работы обучающемуся следует воспользоваться демо-версией этого пакета, доступной в Интернете. Но возможно и применение других пакетов и других версий, тем более, что работа с другими (более поздними) версиями этого продукта во многом аналогична.

Кроме того, для удобства работы обучающиеся могут воспользоваться трафаретами исходных текстов в электронном виде (исходные файлы), которые являются заготовками программ для реализации индивидуальных заданий по большинству изучаемых тем. Эти трафареты доступны в интернете на сайте СПбГЭТУ «ЛЭТИ» (www.eltech.ru), ссылки на них можно найти на странице: Факультеты / Факультет компьютерных технологий и информатики (ФКТИ) / Кафедра вычислительной техники (ВТ) / Печатные труды / А. Х. Мурсаев, О. И. Буренева «Проектирование цифровых устройств с использованием языков Verilog и SystemVerilog», а также на персональных страницах авторов: ... / Кафедра вычислительной техники (ВТ) / Руководство, состав кафедры / …

Желательно также выполнять загрузку разработанных устройств в какую-либо из общедоступных отладочных плат. Материал практической части пособия ориентирован на использование элементной базы компании Altera и соответствующей САПР Quartus II. Программный пакет Quartus II Web Edition, имеющий ряд ограничений по поддерживаемым семействам ПЛИС, но достаточный для выполнения предусмотренных работ, доступен для скачивания на сайте www.altera.com, а некоторые вопросы его настройки и применения приведены в Приложении 2. В качестве учебных плат могут использоваться любые отладочные наборы с установленными ПЛИС компании Altera, например, производимые фирмами TERASIC (DE0, DE1, DE0-Nano или аналогичные) или Waveshare Electronics Ltd (OpenEPXC).

1. Структура программы на VerilogHDL

Данный раздел посвящен изучению вопросов спецификации объекта и анализу программы на алгоритмическом уровне с использованием моделирования. В дальнейшем, по мере прохождения материала будут представляться различные аспекты интерпретации описания в аппаратуре.

1.1. Модуль, декларация портов и внутренних сигналов модуля

Проект в системе проектирования на основе VerilogHDL представлен совокупностью иерархически связанных текстовых фрагментов, называемых модулями (module).

Типовой текст на языке представляет произвольный набор модулей, причем каждому модулю могут предшествовать директивы компилятора.

Рассмотрим структуру и основные разделы программы-образца, представленной в листинге 1.

Листинг 1.1. Текст программы описания простого логического устройства

```
`Timescale 1ns/100ps

module lab1 (x_0,x_1,x_2,z_0,z_1);
parameter delay=1;
input x_0, x_1, x_2;
output z_0, z_1;
wire x_0, x_1, x_2;
wire y0, y1, y2, y3;
reg z_0, z_1;
always @(y0, y1, y2, y3, x_2)
begin # delay
       z_0 =  y1 || y2 || y3;
       z_1 =  y1 || y0 || x_2;
end
assign #delay
       y1 = y0 && ~x_2,
       y2 = x_0 && x_2,
       y3 = ~x_0 && ~x_2,
       y0=~x_0 && ~x_1;
endmodule
```

Замечание: каждый порт фактически описывается трижды – в интерфейсном списке модуля, при объявлении режима порта (in, out или inout), и, наконец, объявляется тип.

Как и большинство программ на языке VerilogHDL (в дальнейшем для краткости Verilog-программа) эта программа начинается с раздела директив компилятора, в которых задается шаг моделирования и временное разрешение симулятора.

В заголовке программы (после ключевого слова **module**) – объявляется имя проекта, описываемого в данном модуле, имена и типы портов, т.е. данных, подаваемых на входы x_0, x_1, x_2 и формируемых на выходах z_1 и z_0. Если выводов во внешнюю среду не предусмотрено, т.е. описываемый блок внутренне закончен и определен, поле декларации внешних соединений модуля может отсутствовать (см. например, листинг 1.2 и раздел 3). Следом за заголовком декларируются параметры настройки модуля (**parameters**). Внутри модуля параметры настройки рассматриваются как константы, но могут переопределяться при включении такого модуля в иерархический проект. Далее в программе объявляются режимы портов и типы данных, передаваемых через порты.

В дополнение к внешним данным могут быть объявлены объекты, представляющие внутренние промежуточные результаты. Декларация объекта (в том числе порта) предполагает объявление его имени и типа – в рассматриваемом примере это y0, y1, y2 и y3.

Язык VerilogHDL опирается на два базовых типа данных для представления сигналов в проектируемом устройстве – **wire** и **reg**. Переменные этих типов могут представлять однобитовый сигнал или набор однобитовых сигналов (вектор или массив). Физическая разрядность векторных данных задается диапазоном индексов, который записывается в квадратных скобках перед списком имен декларируемых данных (цепей) или после него. Префиксная запись задает неупакованный массив одноразрядных сигналов (линий передачи данных), а постфиксная – упакованный. Если объявляется одноразрядный сигнал, то скобочное выражение опускается (хотя декларация вида **wire [1:1] x** не будет ошибкой).

Типы **wire** и **reg** предполагают четырехзначное представление сигналов при моделировании (‘1’, ‘0’, ‘X’ и ‘Z’). Тип **wire** используется для представления объектов, как бы постоянно отслеживающих изменение аргументов соответствующей операции присваивания, а фактически для представления выходов комбинационных схем и соответствующих связей. Единственным оператором, задающим правило изменения объектов типа **wire**, является оператор **assign**, в кото-

рый вложены одна или несколько операций присваивания данным этого типа.

Данные типа **reg** используются для представления элементов, изменяющих свое состояние при возникновении определенных условий (событий) и способных сохранять состояние до момента возникновения такого же или иного предопределенного события. Операции присваивания значений данным типа **reg** размещаются в «процедурных» операторах **initial** и **always**.

SystemVerilog определяет целый ряд дополнительных типов данных, некоторые из которых в данном пособии будут представляться по мере изложения материала.

Функционирование устройства, приведенного в листинге 1.1, задается операторами присваивания промежуточным и выходным данным значений, являющихся логическими функциями входных данных.

1.2. Непрерывные присваивания и процедурные операторы. Операторы initial и always

Процессы в дискретных устройствах происходят параллельно во времени, причем изменения в различных частях устройства могут происходить асинхронно, относительно независимо и в том числе одновременно. При моделировании такое поведение должно быть воспроизведено с требуемой степенью точности последовательными алгоритмами, реализуемыми в ЭВМ. Для правильного понимания принципов языкового описания и результатов моделирования следует достаточно четко представлять методы, заложенные в подсистемы моделирования САПР.

Исполнение операций присваивания в языке VerilogHDL в зависимости от их локализации может предполагать их параллельное или последовательное исполнение (это в равной степени относится и к процедуре моделирования и поведению сигналов в реальном устройстве). Исполнение параллельных операций присваивания (по терминологии стандарта языка continuous assignment – непрерывное присваивание) инициируется по событийному принципу, а именно такое присваивание исполняется в ответ на изменения любого из аргументов, записанного в правой части соответствующих выражений.

На время исполнения всех операторов, инициированных одним и тем же событием (т.е. изменением общего сигнала), значения аргументов остаются неизменными даже в том случае, если какие-то из таких операторов предполагают изменение аргументов других «одно-

временно» исполняемых операторов. Такое правило исполнения операторов и обеспечивает корректное моделирование параллельно протекающих процессов. В отличие от обычных языков программирования, результаты работы совокупности параллельных операторов и присваиваний не зависят от порядка их записи.

Для сложных последовательных алгоритмов описание исключительно через параллельные операторы оказывается неудобным и не наглядным. Последовательные операторы исполняются друг за другом в порядке записи. Они обязательно записываются внутри составных операторов **initial** и **always**. Оператор **always** определен как параллельный, а в его теле присутствуют только последовательные операторы. Исполнение последовательности, входящей в оператор **always**, инициируется либо по времени, либо по событийному принципу.

При инициализации по времени (обозначается конструкцией: **always** # < выражение времени >) последовательность запускается через указанное модельное время от момента начала моделирования и перезапускается через такое же время после исполнения последнего вложенного оператора.

При событийной инициализации (конструкция: **always** @ <список чувствительности>) в списке чувствительности, называемом также списком инициирующих событий, могут присутствовать имена событий:

– переменные, отнесенные к типу **event** (такие переменные должны быть заранее продекларированы и для них должны быть определены правила вычисления);

– функции от сигналов, возвращающих результат типа **event** (например, **posedge**());

– сигналы.

В последнем случае определено, что событием является любое изменение сигнала, включенного в список чувствительности. Оператор **always** определяет последовательное в порядке записи исполнение входящих (вложенных) в него операторов после возникновения каждого инициирующего события. Если оператор **always** может инициализироваться вследствие нескольких различных событий, то общее условие запуска формируется перечислением всех таких событий, разделяемых ключевым словом **or.**

В представленном в листинге 1.1 примере присутствуют четыре непрерывных (параллельных) присваивания, объединенных в оператор **assign** и составной процедурный оператор **always**. Если, напри-

мер, произойдет одновременное изменение входов x_1 и x_2, то при моделировании сначала будут в произвольном порядке вызваны к исполнению параллельные присваивания. Несмотря на то, что присваивание значения сигналу y0 записано позже его использования в выражении для y1, результат будет правильным, потому что изменения значения y0 вызывает исполнение (возможно, повторное) присваивания значения сигналу y1 при новом значении y0.

Замечание: Использование оператора **always** для вычисления выходных данных в данном случае носит иллюстративный характер. Реально в данном случае предпочтительно представление соответствующих преобразований через оператор **assign.**

В процессе исполнения оператора **always** вложенные присваивания выполняются друг за другом. Однако после первого исполнения оператора, например вызванного изменением x_2, он может быть выполнен повторно из-за изменения y1, и результат может быть модифицирован. При моделировании без учета задержек (delay=0) это изменение будет отнесено к тому же моменту времени, что и изменение, вызванное исполнением оператора в ответ на изменение входов. Говорят, что сигнал изменяется относительно вызвавшего его события с дельта-задержкой (то есть очень малой задержкой) [2, 6]. Это значит, что изменение значения не будет учитываться при исполнении операторов, инициирующее событие для которых произошло перед этим изменением. Таким образом, событийное моделирование обеспечивает адекватное представление параллельно работающих и взаимодействующих компонентов.

Процедурный оператор **initial** безусловно и в первую очередь исполняется при начале моделирования. Если таких операторов в модуле несколько, то порядок их вызова не определен. Проектировщик должен так распределять задержки исполнения вложенных операций, чтобы избежать спорных ситуаций. Оператор **initial** и вложенные в него операторы не подлежат имплементации в реальное устройство, а служат исключительно для целей тестирования и отладки проектов. Обычно здесь описывается временная последовательность для входных сигналов при тестировании, а также средства отображения и анализа результатов тестирования. Вложенные операторы исполняются последовательно в порядки записи. Операции присваивания значений входных портов сопровождаются выражениями останова на заданный интервал. Во время останова оператора **initial** исполняются прочие параллельные операторы, для которых на данный момент модельного

времени создались условия инициализации. Рассмотрим программу, приведенную в листинге 1.2.

Листинг 1.2. Текст программы тестирования логического устройства

```
module lab1_m;
 reg x_0,x_1,x_2, z_0,z_1;
 reg y0, y1, y2, y3;
 reg [2:0] takt;
 initial begin   x_0='b0; x_1='b0;  x_2='b0;
         $monitor( "takt = %d, %b %b1 %b1,%b %b1",
                        takt,x_0,x_1,x_2, z_0,z_1);
         for (takt=0;takt!=8;takt=takt+1)
             begin
             #20 x_0= takt[0];
             x_1= takt[1];
             x_2= takt[2];
             end
         $finish;
         end
 always @(x_0,x_1,x_2)
          begin   y0 =~x_0 && ~x_1 ;
                  y1 = y0 && ~x_2;
                  y2 = x_0 && x_2;
                  y3 = ~x_0 && ~x_2 ;
                  z_0 =  y1 || y2 || y3;
                  z_1 =  y1 ||  y0 || x_2 ;
          end
endmodule
```

Модуль *lab1_m* описывает не только некоторое устройство, предполагаемое к реализации (в данном случае комбинационную логическую схему) и представленное оператором **always**, но и тестовую последовательность, представленную оператором **initial**. Так как запрограммирована генерация входов и отображение результатов «внутри» модуля, список портов модуля отсутствует, что является распространенным приемом при отладке алгоритмов.

В предлагаемом примере в генераторе тестового воздействия после установки начальных (нулевых) значений возмущающих сигналов и запуска процедуры непрерывного отображения процесса моделирования (monitor) выполняется циклическая процедура инкрементального наращивания кода входных данных тестируемого устройства. При этом каждое следующее изменение кода отстоит от

предыдущего на 20 единиц модельного времени. Каждое изменение вызывает к исполнению оператор **always** и последовательное исполнение вложенных в него присваиваний. Формально здесь описана последовательная двухъярусная комбинационная схема, где выходы элементов И поданы на входы сборок ИЛИ. В реальности синтезатор может построить иную схему в зависимости от архитектуры целевой микросхемы.

1.3. Блокирующие и неблокирующие присваивания

В языке VerilogHDL определено два вида процедурных присваиваний – блокирующие и неблокирующие. Блокирующие присваивания обозначаются знаком "=" между именем приемника и выражением, определяющим присваиваемое значение, а неблокирующие сочетанием символов "<=". Различие этих видов состоит в порядке фиксации значения данных при исполнении операторов присваивания.

Блокирующее присваивание запрещает выполнение последующих операций вплоть до фактического присваивания приемнику нового значения, а неблокирующее – разрешает.

С точки зрения поведения из этого следует:

– при блокирующем присваивании приемник принимает новое значение сразу после исполнения оператора присваивания, и это «новое» значение используется в последующих выражениях;

– при неблокирующем присваивании вычисленное значение сохраняется во временном буфере, называемом драйвером сигнала. Окончательное присваивание значения сигналу производится только после исполнения всех операторов программы, вызванных общим событием.

В частности при исполнении неблокирующего присваивания некоторому сигналу этот сигнал сохраняет старое значение до исполнения всех операторов в теле исполняемого оператора **always** или до операции приостанова (# <время>), а также и всех других операторов, инициированных общим событием. Реакция на предсказанное изменение сигнала (новое событие) будет воспроизводиться системой моделирования либо через явно указанный в операции присваивания временной интервал задержки, либо – в случае «нулевой» задержки – после отработки всех событий, предсказанных на этот момент времени ранее выполненными операторами.

В программе (листинг 1.2) все присваивания блокирующие. Значит, после изменения x_0, x_1 или x_2 и инициализации оператора

always переменные y0 – y3 «мгновенно» получат новые значения и именно эти новые значения будут использованы при вычислении z_0 и z_1. Иной результат получится, если для присваивания значения переменным y0 – y3 использовать неблокирующие присваивания. Тогда y0 – y3 получают новые значения только после выполнения всех операций, включенных в **always**, а значит, z_0 и z_1 будут формироваться на основе устаревших данных, то есть моделирование для случая описания схемы без памяти будет неадекватным.

Наиболее существенно указанные особенности проявляются при задании и моделировании элементов с памятью – регистровых схем и автоматов, которые рассматриваются далее в разделах 2 и 4.

1.4. Практикум по теме

Для практического освоения материала настоящего пособия рекомендуется выполнить интерпретацию программ в пакете моделирования, например, QuestaSim v.6. Порядок работы в системе моделирования QuestaSim приведен в Приложении 1. Кроме того, удобно воспользоваться доступными в сети по адресу, указанному во Введении к данному пособию, трафаретами исходных текстов.

Для практикума по разделу 1 представляются два исходных файла: файл *Lab1.v* соответствует программе листинга 1.1, а файл *Lab1m.v* – программе листинга 1.2. Необходимо выполнить следующие действия.

1. Запустить программу QuestaSim, создать проект, включив в него файл *Lab1.v*. Просмотреть в редакторе текст файла *Lab1.v*.

2. Выполнить компиляцию проекта.

3. Запустить процедуру моделирования, вызвав команду *Simulate*. Открыть окна наблюдения *Source*, *Process*, *Signals*, *Wave*, *List*.

4. Сформировать тестовую последовательность, используя команду системы моделирования *Force* (в пределах необходимого для полноты проверки числа комбинаций входных данных).

5. Выполнить моделирование в пошаговом режиме. Обратить внимание на состояния процессов и «скачки» курсора между операторами программы в процессе интерпретации, объяснить, почему это происходит? Наблюдать моменты активизации процессов.

6. Выполнить моделирование в автоматическом режиме. Просмотреть временную диаграмму в окне *Wave* и убедиться в правильности вычисления логических функций.

7. Настроить и просмотреть окно *List*. Обратите внимание на то, что различным временным отметкам соответствует разное число строк в списке (календаре событий). Почему?

8. Добавить в проект файл *Lab1_m,* выполнить компиляцию этого файла и пункты 6 и 7, объявив при этом модуль *Lab1_m* вершиной проекта.

9. Заменить в операторе **always** блокирующие присваивания на неблокирующие. Выполнить моделирование в автоматическом режиме и оценить разницу диаграмм по пп. 9 и 10. Почему возникает различие? Какой вариант является ошибочным.

10. Восстановить *Lab1_m.v* в исходном виде, а затем добавить опции задержки компонентов 5 nS, выполнить компиляцию и моделирование, сравнить результаты с ранее полученными.

1.5. Контрольные вопросы по теме

1. Перечислите основные этапы отладки проекта при использовании пакета QuestaSim.

2. Перечислите основные разделы Verilog-программы.

3. Типы данных **wire** и **reg** – что представляют и какими операторами присваиваются?

4. В чем разница между непрерывными и процедурными операторами?

5. При каких условиях исполняется оператор **always**?

6. В чем разница между блокирующими и неблокирующими присваиваниями?

7. Когда происходит фактическое изменение значения сигнала, определенное в операторе присвоения?

2. Представление комбинационных схем и простых триггерных устройств

2.1. Комбинационные схемы

При реализации Verilog-программ в аппаратуру непрерывные (параллельные) присваивания, а также процедурные (последовательные) присваивания, записанные в теле операторов **always**, список чувствительности которых содержит все аргументы такого оператора, порождают комбинационные логические схемы. В случаях, когда логическая функция относительно проста, удобнее и нагляднее непрерывные присваивания, но в сложных случаях, требующих декомпозиции (например, факторизации), приходится использовать последовательные присваивания, в том числе «вкладывая» их в операторы **if** или **case**.

В практике проектирования используются различные способы задания переключательных функций:

– алгебраический,

– алгоритмический,

– табличный.

Язык VerilogHDL позволяет использовать для описания логики функционирования любую форму, не прибегая к преобразованию форм представления.

Алгебраическая форма представляется операцией присваивания, в правой части которой записывается логическое выражение переключательной функции в обозначениях логических операций принятых в языке С (И – &&, ИЛИ – || , НЕ – ~). При небольшом числе переменных предпочтительно использование непрерывного присваивания. Возможно представление комбинационной схемы в структурной форме как совокупность предопределенных примитивов языка (AND, OR, NOT).

Однако при большом числе переменных используют декомпозицию функции на подформулы. Это также может быть полезно, если отдельные подформулы входят в выражения для нескольких выходных сигналов. При этом преобразования удобно представить в виде последовательности процедурных присваиваний, заключенных в операторе **always**.

Алгоритмическая форма предполагает декомпозицию функции, сводя ее к композиции функций меньшего числа аргументов.

Теоретической основой декомпозиции является разложение Шеннона

$$f(x_1, x_2, \ldots\ x_n) = \tilde{}x_1 f(0, x_2, \ldots\ x_n) \vee x_1 f(1, x_2, \ldots\ x_n) .$$

В языке VerilogHDL такое вычисление может быть представлено фрагментом:

```
if (~x1) z= <подформула,
            полученная из f заменой x1 на нуль>;
  else z=   <подформула,
            полученная из f заменой x1 на единицу>;
```

Полученные подформулы в свою очередь могут быть разложены по следующим аргументам.

Отметим, что в практике формальная декомпозиция, как правило, не требуется. Представление в виде последовательной проверки аргументов с использованием условных операторов часто возникает из самой логики проекта и формируется на этапе алгоритмизации задания. Действительно, обычно проектировщик рассуждает так: «если на определенный вход поступает один уровень сигнала, то надо выполнить одно действие (в том числе анализировать другие входные сигналы), иначе – выполнить другое». Такие рассуждения автоматически порождают алгоритмическое представление через операторы **if**.

При использовании процедурных (то есть вложенных в оператор **always**) форм представления комбинационных схем надо учитывать следующие правила, нарушение которых может привести к неадекватному поведению модели и даже некорректному синтезу:

а) список чувствительности оператора **always** должен включать все аргументы логической функции;

б) результаты промежуточных преобразований должны формироваться блокирующими присваиваниями;

в) набор альтернатив в операторах **if** и **case** должен быть полным, охватывающим все наборы аргументов.

Нарушения этих правил может приводить к включению в устройство нежелательных, так называемых «неожиданных» триггеров. Детали подобных и некоторых других ошибок обсуждаются, в частности, в работах [2, 6].

Если функция задана таблицей, то при ее представлении в языке VerilogHDL можно использовать такие варианты.

1. Непосредственная выборка значения из константного массива. При этом таблица определяется как массив-параметр, каждый элемент которого равен значению выхода на коде, численный эквива-

лент которого соответствует индексу этого элемента. Например, для функции z_0 листинга 1.1 можно записать

```
parameter my_function=8'b 10100101;
```

Значение выхода можно получить, используя оператор

```
assign z_0= my_function [{x_2, x_1, x_0}];
```

Однако такой подход не слишком удобен для описания функций с большим числом переменных.

2. Использование оператора выбора **case** или **casex**.

В случаях, когда число нулевых значений функции на всех наборах существенно превышает число единичных (или наоборот) более компактную запись дает оператор выбора. Достаточно перечислить кодовые комбинации, число значений функции на которых превалирует, а остальные комбинации определить как «прочие» (default). Версия **casex** позволяет объединить варианты, совпадающие только в части входных сигналов (знак x обозначает входы, состояние которых несущественно).

Для рассматриваемого примера можно записать

```
casex ({x_2, x_1, x_0})
   3'b0x0 : z_0<=1;
   3'b1x1 : z_0<=1;
   default  z_0<=0;
endcase;
```

3. Создание примитивов пользователя (UDP – user defined primitive).

Примитив – это модуль с одним выходом, поведение которого задается в табличной форме, причем входы и выходы – однобитовые сигналы. По иерархическому уровню примитив эквивалентен модулю и может включаться в другие проекты по тем же правилам, что и модуль (более подробно вопросы структурного представления проекта рассмотрены в следующем разделе). Здесь ограничимся примером реализации функции z_0 из листинга 1.1.

```
primitive example (z_out, in0, in1, in2);
output z_out;
input in0, in1, in2;
table
// in2, in1, in0 : z_0
   0  ?  0 :  1;
   0  0  1 :  0;
   0  1  1 :  0;
```