Языки описания схем

(mk.cs.msu.ru → Лекционные курсы → Языки описания схем)

Блок 13
Verilog:
Синтез
Симуляция, совместимая с синтезом

лектор:
Подымов Владислав Васильевич

e-mail:
valdus@yandex.ru

Осень 2018

(V) Стандарт синтеза

Не всё, что написано на языке Verilog, может быть транслировано в последовательную схему

Существует отдельный стандарт языка, относящийся к синтезу цифровых схем и принципам симуляции для “правильного” программного тестирования дизайна:

“IEEE Standard
for Verilog Register Transfer Level Synthesis”

В этом блоке обсуждается часть стандарта, относящаяся к описанной ранее части синтаксиса языка

(V) Стандарт синтеза

Конструкции языка Verilog с точки зрения стандарта синтеза делятся на три категории:

1. поддерживаемые
   - если дизайн состоит только из поддерживаемых конструкций, то ему однозначно соответствует последовательная схема
   - соответствующая схема согласно маршруту проектирования может быть оптимизирована с сохранением функционала

(V) Стандарт синтеза

Конструкции языка Verilog с точки зрения стандарта синтеза делятся на три категории:

2. игнорируемые
   - средство синтеза игнорирует такие конструкции (*расценивает их как комментарии*)
   - игнорируемые конструкции дописываются в дизайн для организации полноценной функциональной симуляции без изменения синтезируемой схемы

(V) Стандарт синтеза

Конструкции языка Verilog с точки зрения стандарта синтеза делятся на три категории:

3. неподдерживаемые
   - чтобы синтез завершился успешно, все неподдерживаемые конструкции должны быть составными частями игнорируемых
   - это означает, что такие конструкции можно писать вне игнорируемых только в файлах, используемых исключительно для функциональной симуляции

(V) Стандарт синтеза

Коротко о том, что поддерживается, игнорируется и не поддерживается в языке Verilog:

- игнорируются все конструкции, имеющие исключительно последовательную программную семантику:
  - контроль времени (`#N`)
  - *системные задачи*, в число которых входят все команды, начинающиеся с "`$`", упомянутые в блоке 9
  - инициализация (*кроме исключительного случая, здесь не обсуждаемого: описание ROM как двумерного массива с блоком инициализации*)
- поддерживаются конструкции, которые можно (*особенным образом*) однозначно трактовать как описание значений на выходах комбинационных схем и регистров
  - об этом подробнее рассказывается дальше
- не поддерживаются многие другие конструкции:
  - числа с плавающей точкой (`$realtime`, ...)
  - безусловный always-блок
  - почти все конструкции, не упомянутые далее

Поддерживаемые конструкции: комбинационная логика

Точку, являющуюся выходом комбинационной схемы, можно описать тремя разными способами:

1. при помощи непрерывного присваивания
2. при помощи присваивания в объявлении
3. при помощи always-блока

Основной пример, который будет использоваться для иллюстрации этих трёх способов описания, — сумматор:

требуется описать комбинационную схему,
выход которой — шина `z` ширины 3,
на которую непрерывно выводится
сумма чисел в шинах `x`, `y` ширины 2

Поддерживаемые конструкции: комбинационная логика

Непрерывное присваивание

```
wire [2:0] z;
assign z = x + y;
```

Синтезируемая схема будет содержать комбинационную подсхему сумматора, входы которой соединены (*реальными*) проводами с контактами, которым соответствуют имена `x`, `y`

Имя `z` транслируется в три выходных контакта подсхемы сумматора

Присваивание при объявлении

```
wire [2:0] z = x + y;
```

Эта запись эквивалентна объявлению с последующим непрерывным присваиванием (*в верхней половине слайда*)

Поддерживаемые конструкции: комбинационная логика

always-блок

```
reg [2:0] z;
always @(x,y) z = x + y;
```

Можно легко убедиться, что в этих двух строчках сказано

"как бы ни изменялись числа `x`, `y`, всегда выводи в `z` их сумму"

Это в точности соответствует функционалу комбинационной схемы сумматора

В общем случае always-блок может описывать комбинационную схему, в выходные контакты которой транслируется **несколько** точек

Чтобы по always-блоку синтезировалась комбинационная схема, необходимо, чтобы в теле блока описывался комбинационный функционал, то есть чтобы память, которую подразумевает тип переменной, оказалась фиктивной

Поддерживаемые конструкции: комбинационная логика

Чтобы по always-блоку синтезировалась комбинационная схема с выходными контактами, соответствующими точкам $x_1, \ldots x_k$, которую, кроме того, можно корректно просимулировать, достаточно делать так:

- не писать `posedge` и `negedge` в списке отслеживания (`watchlist`, `event list`) always-блока
- включать в список отслеживания все входы предполагаемой комбинационной схемы, либо писать такой список: `@(*)`
  - `@(*)` — это сокращение для `@(<`*имена всех переменных, встречающихся в выражениях присваиваний, ветвлений и выборов*`>)`

Поддерживаемые конструкции: комбинационная логика

Чтобы по always-блоку синтезировалась комбинационная схема с выходными контактами, соответствующими точкам $x_1, \ldots x_k$, которую, кроме того, можно корректно просимулировать, достаточно делать так:

- убедиться, что для **каждого** набора логических значений списка отслеживания новое значение **каждой** переменной x_i определено и **не зависит** от старых значений $x_1, \ldots, x_k$
- перезаписывать значение переменной либо только блокирующими, либо только неблокирующими присваиваниями
  - нельзя "перемешивать" блокирующие и неблокирующие присваивания — стандарт это явно запрещает
  - можно использовать только неблокирующие присваивания — стандарт явно этого не запрещает — но семантика этих присваиваний лучше подходит для описания триггеров, так что лучше их избегать

Поддерживаемые конструкции: комбинационная логика

Более нетривиальный пример

```
wire [1:0] z;
wire u;
reg x, y;
always @(*) begin
  x = 0;
  y = 0;
  case(z)
  0: x = u;
  1: y = u;
  2: begin x = 1; y = x; end
  endcase
end
```

И как это работает?

Поддерживаемые конструкции: триггеры

Немного напоминаний

Триггер — это элемент цифровой схемы, который

- имеет внутренние состояния (память)
- имеет комбинационный функционал относительно входов и состояния
- дискретно изменяет состояние в зависимости от входных сигналов

Триггер — это довольно общее понятие, но на практике триггер обычно может быть представлен как **D-защёлка** или **D-триггер**, возможно **с дополнительными управляющими входами**, на входы которых “навешены” дополнительные комбинационные схемы

Поддерживаемые конструкции: триггеры

Напоминание: D-защёлка имеет информационный вход `x`, управляющий вход `en`, выход `y` и работает так:

- если `en = 1`, то на `y` непрерывно пересылается значение `x`
- если `en = 0`, то на `y` напрерывно пересылается значение, сохранённое во время последнего защёлкивания: заднего фронта сигнала `en`

Пример поддерживаемой реализации D-защёлки:

```
wire en, x;
reg y;
always @(*)
  if(en) y <= x;
```

Поддерживаемые конструкции: триггеры

Условия, при которых по `always`-блоку синтезируется набор D-защёлок, обёрнутых комбинационными схемами, с функционалом, описанном в блоке:

- список отслеживания — `@(*)` (или эквивалентная развёрнутая запись)
- все присваивания блока являются неблокирующими
- для каждого предполагаемого выхода защёлки существует сценарий работы блока, при котором этот выход сохраняет старое значение

Пример поддерживаемой реализации набора защёлок:

```
wire en, x, a;
reg [1:0] y; reg z;
always @(*) begin
  if(en && x) begin y <= 2; z <= a; end
  if(~x) y <= 0;
end
```

Поддерживаемые конструкции: триггеры

Контрпример

```
wire en, x;
reg y;
always @(*)
  if(en) y <= x;
  else y <= 0;
```

В `always`-блоке нет сценариев выполнения, согласно которым в `y` не изменяется

Если бы это была защёлка, то она бы никогда не защёлкивалась

Неожиданно always-блок оказался поддерживаемым **комбинационным** блоком

Поддерживаемые конструкции: триггеры

D-триггер имеет информационный вход `x`, тактовый вход `clk` и выход `y` и работает так:

- по переднему фронту `clk` сохраняется значение `x`
- последнее сохранённое значение непрерывно пересылается в `y`

Пример поддерживаемой реализации D-триггера:

```
wire clk, x;
reg y;
always @(posedge clk)
  y <= x;
```

Поддерживаемые конструкции: триггеры

Условия, при которых по `always`-блоку синтезируется набор D-триггеров, обёрнутых комбинационными схемами, с функционалом, описанным в блоке:

- в списке отслеживания присутствует **ровно** одна однобитовая точка (*имя общего тактового входа*), предварённая словом `posedge` или `negedge`
- все присваивания блока являются неблокирующими

Пример:

```
wire clk, x, a, b;
reg [1,0] y;
reg z;
always @(posedge clk) begin
  if(a && b) begin y <= 2; z <= 1; end
  if(~b) y <= a;
end
```

Поддерживаемые конструкции: триггеры

D-триггер может иметь дополнительные входы, и обычно это входы одного из двух типов:

- асинхронные: состояние изменяется особенным образом по заданному фронту (*переднему или заднему*) такого входа
- синхронные: по фронту **тактового** сигнала состояние изменяется с учётом текущего значения на входе

Изменение функционала триггера при добавлении синхронного входа можно представить как “навешивание” комбинационной схемы на входы и выходы

Пример поддерживаемой реализации D-триггера с синхронным сбросом:

```
wire clk, rst, x;
reg y;
always @(posedge clk)
  if(rst) y <= 0;
  else y <= x;
```

Поддерживаемые конструкции: триггеры

Триггеры с дополнительными **асинхронными** управляющими входами устроены сложнее, и поддерживаемые конструкции для синтеза таких триггеров более хитры

Но сначала пример поддерживаемой реализации D-триггера с асинхронным сбросом:

```
wire clk, rst, x;
reg y;
always @(posedge clk, posedge rst)
  if(rst) y <= 0;
  else y <= x;
```

Поддерживаемые конструкции: триггеры

Условия, при которых по `always`-блоку синтезируется набор D-триггеров с дополнительными асинхронными входами, обёрнутых комбинационными схемами, с функционалом, описанным в блоке:

- каждая точка, перечисленная в списке отслеживания, является однобитовой и предварена словом `posedge` или `negedge`
- тело блока имеет вид
  `if(`c_1`)` S_1 `else if... else if(`c_k`)` S_k `else` S_{k+1}, где
  - $c_1, \ldots, c_k$ — условия, взаимно однозначно соответствующие отслеживаемым точкам, кроме тактовой
  - если отслеживается `posedge x`, то соответствующее условие c_i имеет вид "`x`", иначе "`~x`"
  - S_i, $1 \leq i \leq k+1$ — команда, все присваивания которой являются неблокирующими

Поддерживаемые конструкции: триггеры

Пример поддерживаемого описания набора триггеров с дополнительными асинхронными входами:

```
wire clk, set, rst, x;
reg [1:0] y;
always @(posedge clk, negedge set, posedge rst)
  if(rst) y <= 2;
  else if(~set) y[1] <= 1;
  else y <= x;
```

А как выглядит эта последовательная схема?

Значения `x` и `z`

Значение `x`

- трактуется средством синтеза как любое логическое значение (0, 1) на выбор средства синтеза (*don't care value*)
- можно использовать явно в специальной команде выбора `casex` (*её в слайдах не было, но она есть*)
- нельзя использовать явно ни в каких других командах и выражениях

Принципы использования значения `z` выходят за рамки понятий, рассмотренных в курсе для последовательных схем, поэтому явное использование этого значения в рамках курса не рекомендуется

Другие возможности синтеза

В стандарте описано намного больше допустимых конструкций, включая

- описание блоков (*адресованной*) памяти
- атрибуты команд (*как и в C/C++, подсказывающие средству синтеза семантические особенности блока, позволяющие "правильно" его синтезировать*)
- директивы (*как директивы компилятору C/C++, в том числе макроопределения*)
- ...

Всё это можно использовать, предварительно внимательно почитав документацию

Заключение

Чтобы меньше задумываться над конкретными деталями стандарта, при разработке схемы на языке Verilog достаточно в “спорных” местах задаваться вопросом

а можно ли представить себе последовательную схему с описанным функционалом?

Если несложно представить себе разбиение функционала на триггеры/регистры и комбинационную часть, значит, с высокой вероятностью схема будет синтезирована корректно