

Языки описания схем

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

Блок 9

Verilog:

базовые возможности симуляции

лектор:

Подымов Владислав Васильевич

e-mail:

valdus@yandex.ru

Осень 2017

(V) Отладка схем

Чем более нетривиальная схема разработана проектировщиком, тем больше вероятность, что она содержит ошибки

Исправление ошибок в программе — это привычный процесс:

- ▶ отладочный вывод при разработке
- ▶ тестирование
- ▶ отладчик, проигрывающий программу пошагово и показывающий состояния вычислений
- ▶ пусть пользователь сообщает об ошибках, и чем они критичнее, тем быстрее исправятся в новых версиях

(V) Отладка схем

Ошибки в схеме, как правило, критичны:

- ▶ программу можно легко поменять на исправленную
- ▶ “поменять схему” — это
 - ▶ (хороший случай) легко перепрограммировать ПЛИС
 - ▶ (плохой случай) выбросить “железку”, содержащую неисправную схему, и поставить на её место новую

Замена аппаратуры — это трудоёмкий и недешёвый процесс, поэтому отладка схемы — более важный этап, чем отладка программы

(V) Отладка схем

Возможности отладки схемы зависят от используемого маршрута проектирования

Например, можно:

- ▶ тестировать изготовленную схему-прототип
- ▶ анализировать схему на низких уровнях представления, возникающих в маршруте
- ▶ тестировать последовательную схему на ПЛИС
- ▶ анализировать функционал RTL-описания схемы

В курсе рассматривается один способ отладки RTL-описания схемы без привлечения более низких уровней описания

(V) Отладка схем

Самый известный способ отладки RTL-описания схемы — это функциональная симуляция, или программная симуляция

Характерные особенности этого способа:

- ▶ выполнение RTL-описания при симуляции — это последовательность событий (в том числе событий “значение сигнала изменилось”) в контексте увеличивающейся переменной, хранящей текущее время
- ▶ изменение логического значения в точке всегда мгновенно
- ▶ реакция сигналов в точках схемы на происходящие события может быть как мгновенной, так и отложенной с явным указанием длительности задержки
- ▶ RTL-описание преобразуется в программу, последовательно преобразующую логические значения и выполняющую специальные отладочные события (например, отладочный вывод)

(V) Как будем симулировать схемы

Предварительное замечание:

- ▶ дальнейшая инструкция по симуляции схемы подразумевает использование стандартной консоли Linux и конкретного набора утилит
- ▶ эти утилиты можно установить и в Windows и использовать схожим образом
- ▶ другие утилиты не рассматриваются

(V) Как будем симулировать схемы

Шаг 1

Описываем модуль тестирования (testbench) без портов, в котором содержатся

- ▶ экземпляр тестируемого модуля
- ▶ команды, описывающие сценарий работы схемы:
изменение портов тестируемого модуля во времени
- ▶ отладочные команды

```
// тестируемый модуль
module sum(input [1:0] x, input [1:0] y, output [2:0] z);
    assign z = x + y;
endmodule
```

```
// модуль тестирования
module test_sum;
    reg [1:0] x, y;
    wire [2:0] z;
    sum _sum(.x(x), .y(y), .z(z));
    // другие команды
endmodule
```

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

Многие конструкции языка Verilog предназначены **исключительно для симуляции**, так как используют понятия, разумные в контексте работы последовательной программы, но бессмысленные с точки зрения схемы:

- ▶ начало и конец работы
 - ▶ например, инициализация переменных и завершение симуляции
- ▶ несхемные последовательные типы данных и значения этих типов
 - ▶ например, “число с плавающей точкой” и “текущее модельное время”
- ▶ вывод в консоль (в выходной поток)
- ▶ точное указание задержек при распространении сигналов

Средство синтеза, как правило, игнорирует все такие конструкции

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

Инициализация переменных *выглядит так же, как и в C/C++:*

```
reg [3:0] r = 4'b10x1;
```

Неинициализированная переменная имеет значение **x**

Инициальный блок выполняется в нулевой момент времени:

```
initial <команда>
```

Контроль временных задержек осуществляется командой **#N**,
где N — число

Смысл: задержка длительности N между некоторыми
событиями симуляции

Контроль временных задержек может появляться везде, где
разумно понятие “задержки”

\$finish — команда завершения программы симуляции

(V) Как будем симулировать схемы

Шаг 1: контроль временных задержек

`assign #N x = E`

- ▶ пересылать значение E в провод x с задержкой N

`#N` перед командой последовательного блока

- ▶ последовательными, например, являются `always`-блоки и инициальные блоки
- ▶ подождать время N перед выполнением всех следующих команд блока

`x = #N E;` в блокирующем присваивании

- ▶ вычислить значение E
- ▶ подождать время N
- ▶ записать вычисленное значение в x
- ▶ перейти к выполнению следующих команд

`x <= #N E;`

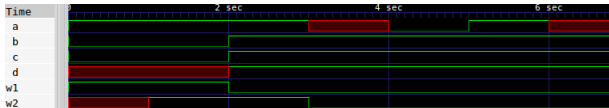
- ▶ вычислить значение E
- ▶ перейти к выполнению следующих команд
- ▶ спустя время N записать вычисленное значение в x

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

Пример:

```
module test;
  reg a = 1, b = 0, c, d;
  wire w1, w2;
  assign w1 = !b;
  assign #1 w2 = !b;
  initial begin
    c = 0; #1 a <= #3 0; b = #1 1; c = a; d = b;
    #1 a = 1'bx; #3 a = 1'bx;
  end
  initial begin
    #5 a = 1; #2 $finish;
  end
endmodule
```



(о том, как получить такие диаграммы сигналов, рассказывается дальше)

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

Безусловный **always**-блок выполняется в нулевой момент времени, и немедленно повторяется после завершения:

`always <команда>`

Безусловный **always**-блок обязан иметь ненулевое время выполнения, то есть содержать команду `#N` перед последовательной командой или внутри блокирующего присваивания

Пример:

```
module test;  
    reg clock = 0;  
    always #1 clock = !clock;  
    initial #6 $finish;  
endmodule
```



(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

`$display("format", args...)` — *полный аналог команды printf в C/C++*

`$strobe("format", args...)` — подождать, пока все одновременные действия выполнятся, и выполнить `$display`

`$monitor("format", args...)` — выполнять `$display` каждый раз, когда изменяется хотя бы одно из значений в `args`

`$monitor(args...)` — выполнить `monitor` для естественного формата; две запятых подряд порождают пробел

`$realtime` — возвращает текущее модельное время как 64-битное *число с плавающей точкой*

`$time` — возвращает текущее модельное время как 64-битное *целое число* (с округлением)

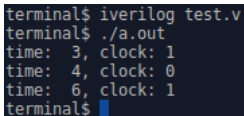
`$stime` — возвращает текущее модельное время как 32-битное *целое число*

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

Пример:

```
module test;  
    reg clock = 0;  
    always #1.5 clock = !clock;  
    initial #6 $finish;  
    initial #3 $monitor("time: %2d, clock: %d", $stime, clock);  
endmodule
```



```
terminal$ iverilog test.v  
terminal$ ./a.out  
time:  3, clock: 1  
time:  4, clock: 0  
time:  6, clock: 1  
terminal$
```

*(о том, что именно произошло в консоли,
рассказывается дальше)*

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

`$dumpfile("file")` — открыть для записи файл `file` и записать служебную информацию, требуемую для обработки диаграмм сигналов

`$dumpvars(level, objlist)` — начать запись диаграмм сигналов в файл, установленный при помощи `$dumpfile`

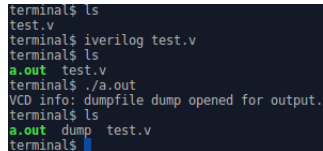
- ▶ `objlist` — это список точек и экземпляров модулей через запятую (имя главного модуля совпадает с именем его единственного экземпляра)
- ▶ `level`: 0, если требуются все точки всех экземпляров иерархии модулей, и 1, если для перечисленных модулей требуются только непосредственно содержащиеся в них точки

(V) Как будем симулировать схемы

Шаг 1: элементы симуляционного синтаксиса

Пример:

```
module test;  
    reg clock = 0;  
    always #1.5 clock = !clock;  
    initial #6 $finish;  
    initial begin  
        $dumpfile("dump");  
        $dumpvars(0,test);  
    end  
endmodule
```



```
terminal$ ls  
test.v  
terminal$ iverilog test.v  
terminal$ ls  
a.out test.v  
terminal$ ./a.out  
VCD info: dumpfile dump opened for output.  
terminal$ ls  
a.out dump test.v  
terminal$
```

Файл dump содержит

- ▶ информацию об иерархии модулей
 - ▶ в этом примере нет иерархии, есть только один экземпляр модуля test
- ▶ диаграммы сигналов во всех точках всех модулей иерархии
 - ▶ здесь это единственная точка clock

(V) Как будем симулировать схемы

Шаг 2

Компилируем модуль тестирования при помощи консольной утилиты iverilog

Использование этой утилиты схоже с использованием gcc/g++

```
terminal$ ls
sum.v test.v
terminal$ cat sum.v
module sum(input [1:0] x, input [1:0] y, output [2:0] z);
    assign z = x + y;
endmodule
terminal$ cat test.v
module test;
    reg [1:0] x = 0, y = 0;
    wire [2:0] z;
    sum sum(.x(x), .y(y), .z(z));
    initial begin
        #1 x = 1; #1 y = 1; #1 $finish;
    end
    initial $monitor(x,,z);
    initial begin
        $dumpfile("dump");
        $dumpvars(0,test);
    end
endmodule
terminal$ iverilog sum.v test.v
terminal$ ls
a.out sum.v test.v
terminal$
```

(V) Как будем симулировать схемы

Шаг 3

Исполняем результат компиляции

```
a.out sum.v test.v
terminal$ ./a.out
VCD info: dumpfile dump opened for output.
0 0 0
1 0 1
1 1 2
terminal$ ls
a.out dump sum.v test.v
terminal$
```

Если цель симуляции — отладочный вывод в консоль, то изучаем то, что вывелось в консоль

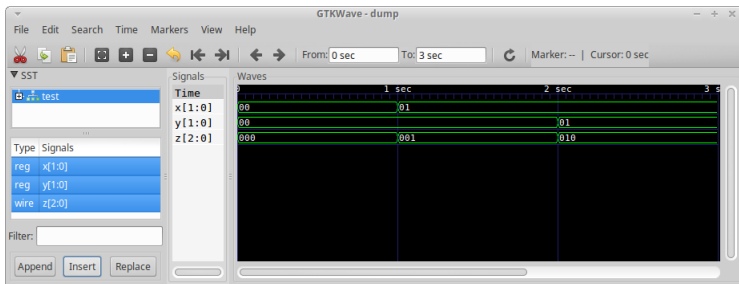
Если цель симуляции — генерация диаграмм сигналов, то переходим к следующему шагу

(V) Как будем симулировать схемы

Шаг 4

Если модуль тестирования содержал команды генерации диаграмм сигналов, то визуализируем полученные диаграммы при помощи утилиты gtkwave

```
terminal$ ls  
a.out dump sum.v test.v  
terminal$ gtkwave dump
```



Конец блока 9