

# Языки описания схем

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

## Блок 12

Verilog:  
основы программной симуляции

Лектор:  
**Подымов Владислав Васильевич**

E-mail:

**valdus@yandex.ru**

# Вступление

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

Как проверить, правильно ли реализована эта схема?

Отладка программ — несложный привычный процесс:

- ▶ придумать тестовое покрытие и позапустить программу на нём
- ▶ вставить отладочный вывод в “подозрительные” места
- ▶ запустить отладчик и “наглядно” увидеть, как это работает
- ▶ выпустить программу с ошибками, и если пользователь с ними столкнётся, то исправить и прислать новую версию

Отладка микросхемы — более “неповоротливый” и трудный процесс: в конечном итоге микросхема — это очень маленькое техническое устройство, внутрь которого заглянуть никак нельзя, а можно только посылать сигналы на входы и считывать их с выходов

# Вступление

Ошибки в микросхемах намного критичнее ошибок в программах, и их исправление более трудоёмко:

- ▶ В худшем случае в микросхему, выпущенную как устройство, нельзя внести ни одного изменения
  - ▶ если в ней есть хотя бы одна критичная ошибка, то вся схема выбрасывается
  - ▶ при этом пере выпуск микросхемы и доведение её до конечного пользователя — недешёвое удовольствие
- ▶ Даже в лучшем случае (*например, в программируемых логических интегральных схемах — ПЛИС*) перепрограммировать схему намного труднее, чем пере выпустить программу
  - ▶ с программами всё просто: выложил в сеть новую версию, пользователь её скачал и запустил
  - ▶ скачав новую версию кода схемы, пользователь вынужден будет
    - ▶ внимательно изучить, куда и как положить этот код
    - ▶ аккуратно положить этот код в нужное место: любая ошибка может привести к полной поломке устройства

# Вступление

Отладка (или, как говорят схемотехники, *верификация*) схемы производится на всех этапах проектирования:

- ▶ итоговое устройство “вживую” запускается на тестовых сигналах
- ▶ перед производством микросхемы она запускается на устройствах, способных моделировать поведение произвольных схем (*например, ПЛИС*)
- ▶ перед этим последовательно выполняется синтез схемы на разных уровнях абстракции из исходного высокоуровневого кода, и на каждом уровне отслеживается свой класс ошибок

Каждый следующий этап отладки схемы затратнее предыдущего по времени и финансам, и тем затратнее, чем больше ошибок обнаруживается на этом этапе

# Вступление

Начать отлаживать схему можно и **до** её синтеза: достаточно

- ▶ разработать схему на языке описания аппаратуры и
- ▶ воспроизвести поведение этой схемы в **программной семантике** (то есть выполнить **программную симуляцию** схемы)

**Плюсы** программной симуляции:

быстро, дешево, позволяет исправить ошибки в “логике” схемы

**Минусы** программной симуляции:

- ▶ в ней никак не учитываются физические и технологические особенности схемы (*которыми также могут порождаться ошибки*)
- ▶ даже без учёта этих особенностей программная семантика схемы только *приблизительно* похожа на аппаратную

В курсе обсуждается *небольшая часть основ* программной симуляции в  $\mathcal{V}$

## У: тестирующий модуль

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

### Сквозной пример:

реализован модуль сумматора, и хочется посмотреть, как он работает

Для симуляции схемы обычно создаётся отдельный **тестирующий** модуль (testbench)

В этот модуль вставляется экземпляр тестируемого модуля

К портам экземпляра подключаются *переменные* (ко входам) и *соединения* (к выходам), и отдельно задаётся **сценарий выполнения** схемы: сигналы, соответствующие переменным

```
module test();  
    reg [1:0] x, y;  
    wire [2:0] z;  
    sum testee(.x(x), .y(y), .z(z));  
    // ...  
endmodule
```

## $\mathcal{V}$ : начало симуляции

На каждом шаге симуляции определены, *в числе прочего*:

- ▶ **текущее время  $\tau$** :
  - ▶ строго говоря, это число с плавающей точкой
  - ▶ чтобы не углубляться в механизмы округления значений времени, будем считать, что это **целое число**
- ▶ значения всех точек в текущий момент времени
  - ▶ такие, как обсуждалось в *предыдущем блоке*

В начале симуляции:

- ▶  $\tau = 0$
- ▶ Значение каждой переменной:  $(\mathcal{X}\mathcal{X} \dots \mathcal{X})$
- ▶ Значение каждого соединения:  $(\mathcal{Z}\mathcal{Z} \dots \mathcal{Z})$

## У: сценарий выполнения на примере

```
module test();  
  reg [1:0] x, y;  
  // ...  
  initial begin  
    x = 3;  
    #1 y = 2; x = 2;  
    #2 x = 1;  
    #1 $finish;  
  end  
endmodule
```

`initial` <команда> — начальная процедура:

- ▶ Запускается в начале симуляции
- ▶ Выполняет <команду> и завершается

`begin` <последовательность команд> `end` — составная команда:  
последовательно выполняются <команды> <последовательности>



## У: сценарий выполнения на примере

```
module test();  
  reg [1:0] x, y;  
  // ...  
  initial begin  
    x = 3;  
    #1 y = 2; x = 2;  
    #2 x = 1;  
    #1 $finish;  
  end  
endmodule
```

`x = E;` — **блокирующее присваивание**: вычисляется выражение E, и в тот же момент переменной x присваивается вычисленное значение

`#N` — **задержка** в N единиц времени

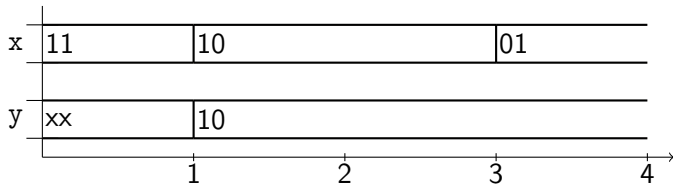
`#N <команда>`: перед выполнением <команды> заморозить процедуру на N единиц времени

`$finish;` — команда принудительного завершения симуляции

## У: сценарий выполнения на примере

```
module test();  
  reg [1:0] x, y;  
  // ...  
  initial begin  
    x = 3;  
    #1 y = 2; x = 2;  
    #2 x = 1;  
    #1 $finish;  
  end  
endmodule
```

Соответствующие сигналы x, y:



## У: события

В ходе симуляции планируются и выполняются **события**, заставляющие симулятор делать что бы то ни было

**Например**,  $x \leftarrow val$  — событие “присвоить точке  $x$  значение  $val$ ”

При **планировании** события определяются его

- ▶ **время** выполнения и
- ▶ **очерёдность** среди всех событий с одинаковым временем

Спланированные события группируются в последовательность **регионов**:

- ▶ в регионе содержатся все события с одинаковым временем выполнения
- ▶ регионы упорядочены по времени выполнения

**Текущим регионом** далее называется регион, события которого спланированы на *текущее время*

## У: события

Ход симуляции:

- ▶ Если текущий регион непуст, то события этого региона выполняются в порядке очерёдности
  - ▶ При выполнении этих событий могут планироваться новые события в текущем и других регионах
- ▶ Если все регионы пусты, то симуляция завершается
- ▶ Если текущий регион пуст и есть хотя бы одно невыполненное событие, то обновляется текущее время : следующий непустой регион становится текущим

При обсуждении очерёдности выполнения событий будем считать, что существует два вида событий:<sup>1</sup>

- ▶ **срочные**: сначала выполняются все такие события региона
- ▶ **отложенные**: выполняются после всех срочных событий региона

---

<sup>1</sup> На самом деле события бывают **активные**, **неактивные**, **отложенные** и **обозревающие**, и виды преобразуются в ходе симуляции.

В объяснениях жертвуем строгостью в пользу “не выпадения в осадок”

## У: процессы

События планируются **процессами**, запускающимися каждый раз при выполнении подходящих условий симуляции<sup>1</sup>

### Примеры процессов:

- ▶ *initial <команда>*: запускается один раз в начале симуляции и выполняет <команду>
- ▶ *assign x = E;*: запускается при каждом изменении значений точек выражения E и выполняется так:
  - ▶ вычисляется значение val выражения E
  - ▶ планируется текущее срочное событие  $x \leftarrow val$
- ▶ *Экземпляр модуля*: эквивалентен набору процессов модуля и набору *непрерывных присваиваний* согласно назначениям портов

Процессы запускаются **параллельно**, и в случае **одновременности** выполняются последовательно в неспецифицированном порядке<sup>2</sup>

---

<sup>1</sup> На самом деле каждый шаг выполнения процесса — это особое событие. Снова жертвуем строгостью в пользу “не выпадения в осадок”

<sup>2</sup> Это обычный вид программного параллелизма: конкурентность (concurrency)

## ∪: процедуры

Процедуры — это процессы, управляющие значениями *переменных*

### Примеры процедур:

- ▶ *Начальная процедура* `initial <команда>`
- ▶ *Постоянная процедура* `always <команда>`: запускается в начале симуляции и выполняет <команду> в бесконечном цикле
  - ▶ *Ограничение на использование*: между обновлениями текущего времени процедурой должно планироваться **конечное** число событий

В ∪ есть возможность *инициализации переменных*, и выглядит это примерно так же, как и в C/C++:

```
reg [2:0] x = <комбинационное выражение>;
```

Инициализация “`reg ... x = E;`” эквивалентна объявлению, совмещённому с подходящей начальной процедурой:

```
reg ... x;  
initial x = E;
```

# ∪: процедуры

## Примеры команд:

- ▶ *begin* <последовательность команд> *end*:  
выполнить <последовательность команд>
- ▶ *\$finish*:  
спланировать текущее срочное событие “завершить симуляцию”
- ▶  $x = E$ ; — *блокирующее присваивание*:
  - ▶ вычислить значение *val* выражения *E*
  - ▶ “заморозить” процедуру  
до выполнения всех текущих срочных событий
  - ▶ спланировать текущее срочное событие  $x \leftarrow val$
- ▶  $x \leq E$  — *неблокирующее присваивание*
  - ▶ вычислить значение *val* выражения *E*
  - ▶ спланировать текущее отложенное событие  $x \leftarrow val$

## У: контроль временных задержек

Перед каждой командой, а также внутри некоторых команд и в некоторых других местах процессов можно дописывать *задержку* “#N”, где N — число<sup>1</sup>

Общий смысл этой записи: поправка  
“здесь должна быть задержка в N единиц времени”  
к семантике соответствующего места кода

---

<sup>1</sup> И здесь тоже жертвуем строгостью:  
задержки бывают разными и применяются намного разнообразнее



# ∪: контроль временных задержек

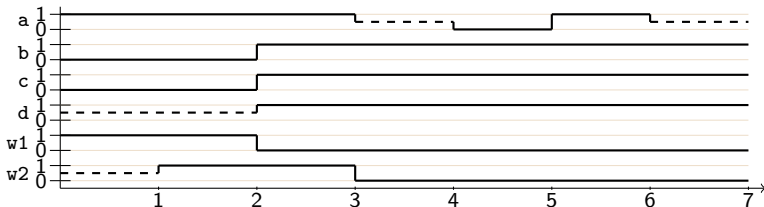
Типовые применения:

- ▶ `#N <команда>`: заморозить процедуру на  $N$  единиц времени, и затем приступить к выполнению команды
- ▶ `x = #N E;`
  - ▶ вычислить значение `val` выражения  $E$
  - ▶ заморозить процедуру на  $N$  единиц времени
  - ▶ выполнить присваивание для (пред)вычисленного значения `val`
- ▶ `x <= #N E;`
  - ▶ вычислить значение `val` выражения  $E$
  - ▶ спланировать отложенное событие  $x \leftarrow val$  со временем (текущее +  $N$ )
- ▶ `assign #N x = E;`: каждый раз, когда изменяются значения точек, используемых в  $E$ ,
  - ▶ вычисляется значение `val` выражения  $E$  и
  - ▶ планируется срочное событие  $x \leftarrow val$  со временем (текущее +  $N$ )

# Большой бессмысленный 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
end
endmodule
```

Как это выполняется:



## ℳ: компиляция кода

Программная симуляция кода на языке ℳ — это самое обычное выполнение самой обычной программы (**симулятора**)

Общее устройство симулятора обсуждалось ранее:

- ▶ данные: текущее время и значения точек
- ▶ поток управления: параллельно (конкурентно) запущенные процессы
- ▶ пошагово планируются/выполняются события и обновляется значение текущего времени

Исполняемый файл симулятора можно собрать из исходного кода любым подходящим компилятором

Далее в примерах используется компилятор Icarus Verilog и соответствующая утилита `iverilog` консоли Linux

## У: КОМПИЛЯЦИЯ КОДА

Утилита `iverilog` используется в целом так же, как и `gcc/g++`:

```
terminal> ls
sum.v test.v
terminal> pygmentize sum.v
module sum(input [1:0] x, input [1:0] y, output [2:0] z);
    assign z = x + y;
endmodule
terminal> pygmentize test.v
module test();
    reg [1:0] x, y;
    wire [2:0] z;
    sum testee(.x(x), .y(y), .z(z));
    initial begin
        x = 3;
        #1 y = 2; x = 2;
        #2 x = 1;
        #1 $finish;
    end
endmodule
terminal> iverilog test.v sum.v
terminal> ls
a.out sum.v test.v
terminal> ./a.out
terminal> █
```

## У: отладочный вывод

Как и любую нормальную программу, симулятор можно заставить выдавать полезную информацию о своей работе (то есть о выполнении схемы)

Например, можно вставить в исходный код команды отладочного вывода, схожие с командой *printf* языка C/C++

Важное отличие от C/C++: эти команды выполняются в заданные моменты времени и планируют события, выполнение которых приводит к желаемому результату

## ∪: отладочный вывод

Примеры таких команд:

- ▶ `$display("format", args...)`: спланировать текущее срочное событие — *полный аналог команды printf языка C/C++*
- ▶ `$strobe("format", args...)`: аналогично `$display`, но планируется отложенное событие, выполняющееся после всех событий того же региона
- ▶ `$monitor("format", args...)`: аналогично `$strobe`, но событие планируется при каждом изменении значений `args`
- ▶ `$monitor(args...)`: аналогично предыдущему пункту, но для “удобочитаемого” формата по умолчанию

Вспомогательные выражения:

- ▶ `$time`: возвращает текущее время как 64-битное целое число
- ▶ `$stime`: возвращает текущее время как 32-битное целое число

## У: ОТЛАДОЧНЫЙ ВЫВОД

Пример: наблюдение за переменными

```
module test();  
    reg clock = 0;  
    always #2 clock = !clock;  
    initial #7 $finish;  
    initial begin  
        $display("time clock");  
        $monitor(" %2d  %1d", $stime, clock);  
    end  
endmodule
```

```
terminal> ls  
test.v  
terminal> iverilog test.v  
terminal> ls  
a.out test.v  
terminal> ./a.out  
time clock  
  0  0  
  2  1  
  4  0  
  6  1  
terminal> █
```

## ∪: визуализация сигналов

Особыми командами можно получить описание сигналов в требуемых (отслеживаемых) точках в особом текстовом формате (VCD, описан в стандарте ∪):

- ▶ `$dumpfile("file")`: (срочное событие) открыть файл `file` для записи сигналов и иерархии модулей
- ▶ `$dumpvars(level, objlist)`: (срочное событие) включить запись информации об изменениях сигналов в файл, открытый командой `$dumpfile`
  - ▶ `objlist`: список отслеживаемых точек и имён экземпляров модулей
    - ▶ *полагается, что в схеме есть один экземпляр модуля тестирования с именем, равным имени модуля*
  - ▶ `level`:
    - ▶ 0, если требуются все точки всей иерархии экземпляров
    - ▶ 1, если для перечисленных экземпляров требуются только непосредственно содержащиеся в них точки

Визуализировать сигналы можно любым подходящим средством — например, GTKWave



## ∇: визуализация сигналов (пример)

```
module test();  
    reg clock = 0;  
    always #2 clock = !clock;  
    initial #7 $finish;  
    initial begin  
        $dumpfile("dump.vcd");  
        $dumpvars(1, test);  
    end  
endmodule
```

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

GTKWave Analyzer v3.3.79 (w)1999-2017 BSI

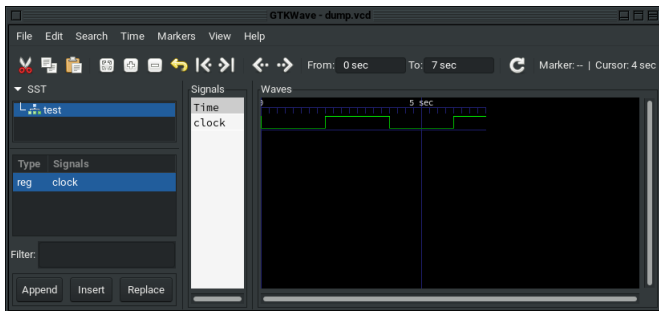
[0] start time.

[7] end time.



## ∇: визуализация сигналов (пример)

```
module test();  
    reg clock = 0;  
    always #2 clock = !clock;  
    initial #7 $finish;  
    initial begin  
        $dumpfile("dump.vcd");  
        $dumpvars(1, test);  
    end  
endmodule
```

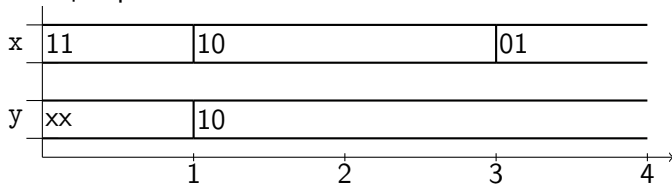


# Сквозной пример от начала и до конца

Тестируемый модуль:

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

Выдумываем сценарий выполнения:



Определяемся с тем, что будет извлекаться из симулятора — например:

- ▶ изменения  $x$ ,  $y$  и  $z$  в консоли
- ▶ красивые картинки изменения  $x$ ,  $y$  и  $z$

## Сквозной пример от начала и до конца

Реализуем соответствующий тестирующий модуль:

```
module test();
reg [1:0] x, y;
wire [2:0] z;
sum testee(.x(x), .y(y), .z(z));
initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, test);
    $display("time x y z");
    $monitor(" %1d %1d %1d %1d", $time, x, y, z);
    x = 3;
    #1 y = 2; x = 2;
    #2 x = 1;
    #1 $finish;
end
endmodule
```

# Сквозной пример от начала и до конца

Компилируем, симулируем, визуализируем:

```
terminal> ls
sum.v test.v
terminal> iverilog test.v sum.v
terminal> ls
a.out sum.v test.v
terminal> ./a.out
VCD info: dumpfile dump.vcd opened for output.
time x y z
  0  3 x x
  1  2 2 4
  3  1 2 3
terminal> ls
a.out dump.vcd sum.v test.v
terminal> gtkwave dump.vcd
```

GTKWave Analyzer v3.3.79 (w)1999-2017 BSI

[0] start time.

[4] end time.



# Сквозной пример от начала и до конца

Компилируем, симулируем, визуализируем:

