Языки описания схем

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

Блок 11

Verilog:
программная симуляция схем

Лектор:
Подымов Владислав Васильевич

E-mail:
valdus@yandex.ru

ВМК МГУ, 2019, осенний семестр

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 позапускать 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 перевыпуск 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 перевыпустить 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 производится отладка

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, дёшево, 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 обуждается *небольшая часть основ* программной симуляции в $\mathcal{V}$

$\mathcal{V}$: подробнее о программной семантике

Состояние данных схемы в программной семантике состоит из:

- значений всех точек
  - значение точки — это набор из n логических значений, где n — ширина точки
- текущего времени: числа с плавающей точкой

В процессе выполнения схема по особым правилам

- порождает и обрабатывает **события**
- увеличивает текущее время

Событие состоит из

- времени выполнения: числа с плавающей точкой — и
- действия

Пример действия: “значение переменной заменяется на ...”

$\mathcal{V}$: подробнее о программной семантике

Общее устройство симуляции:

Начало: текущее время 0, значения всех разрядов всех переменных — $\mathtt{x}$, значения всех разрядов всех соединений — $\mathtt{z}$

Шаг симуляции:

- выполняются языковые конструкции,
  которые должны выполниться в текущий момент
  - этими конструкциями могут породиться события
    для текущего и последующих моментов
- обрабатываются события,
  которые должны обработаться в текущий момент
  - это может привести к выполнению языковых конструкций
- если все конструкции выполнились и все события обработаны,
  то текущее время увеличивается
  - до времени ближайшего события, если оно грядёт
  - если грядущих событий нет, то симуляция завершается

$\mathcal{V}$: тестирующий модуль

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

Обсудим механизмы и особенности симуляции в таком контексте:

реализован модуль на языке $\mathcal{V}$,

и хочется посмотреть, как этот модуль 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 в переменных задаётся сценарий выполнения

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

$\mathcal{V}$: инициализация

Переменные, *как и в C/C++*, можно инициализировать:

```
type id = E;
```

Семантика:

- инициализация выполняется в момент времени 0
- вычисляется значение `E`
- порождается и немедленно обрабатывается событие "присвоить переменной вычисленное значение" в текущий момент времени
  - будем записывать это коротко:
    значение "немедленно записывается" в `x`

В коде $\mathcal{V}$ можно определять процедуры: конструкции, выполняющиеся в заданные моменты времени и порождающие события при выполнении

Пример такой конструкции — непрерывное присваивание

Семантика непрерывного присваивания "`assign x = E;`": в начале симуляции и каждый раз, когда изменяются значения аргументов `E`, выражение `E` вычисляется и немедленно записывается в `x`

$\mathcal{V}$: инициализация; процедурные команды

Другой пример процедуры — начальная процедура:

```
initial <команда>
```

Семантика: <команда> выполняется в момент времени 0

Примеры команд:

- Блокирующее присваивание: `x = E;`
  Семантика:
  вычислить значение E и немедленно записать результат в x
- Неблокирующее присваивание: `x <= E;`
  Семантика: вычислить значение E и записать результат в x
  в текущий момент, но после обработки
  остальных событий текущего момента
- Составная команда: `begin <последовательность команд> end`
  Семантика: немедленно последовательно выполнить
  все команды между begin и end
- Принудительно завершить симуляцию: `$finish;`

$\mathcal{V}$: контроль временны́х задержек

Перед каждой командой, а также внутри некоторых команд можно добавить запись `#N`, где `N` — число

Общий смысл этой записи: “подождать `N` единиц времени”

Типовые применения:

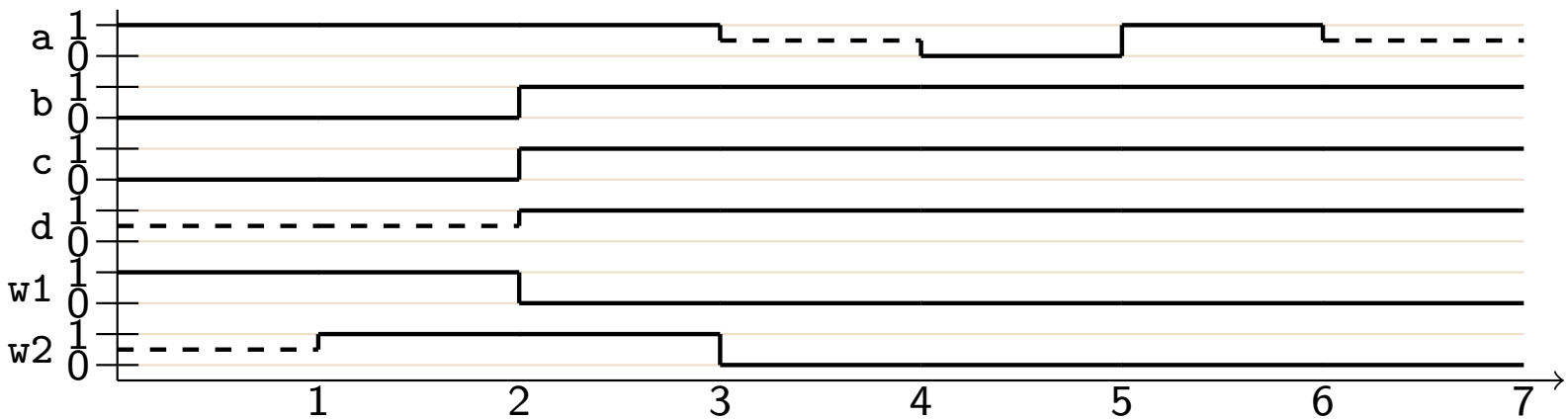
- `assign #N x = E;` : присваивать переменной `x` значение `E` **через N единиц времени после его изменения** (*то есть пересылать E в x с задержкой N*)
- `#N <команда>` : перед выполнением `<команды>` “заморозить” процедуру на `N` единиц времени
- `x = #N E;` : вычислить выражение `E`, заморозить процедуру на `N` единиц времени, и затем немедленно записать вычисленный результат в `x`
- `x <= #N E;` : вычислить выражение `E` и записать вычисленное значение в `x` спустя `N` единиц времени после всех событий этого грядущего момента, но не замораживать выполнение процедуры

$\mathcal{V}$: пример

Совмещаем знания в большом бессмысленном примере:

```
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
```

Как это выполняется:



$\mathcal{V}$: постоянная процедура

Синтаксис постоянной процедуры:

```
always <команда>
```

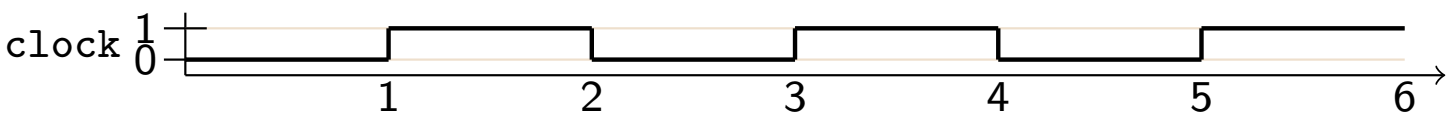
Семантика: <команда> выполняется всегда, то есть:

- <команда> выполняется в начале симуляции
- после каждого выполнения <команда> выполняется снова

Процедура корректна только в том случае, если <команда>
в каждый момент времени выполняется лишь конечное число раз

Пример: моделирование тактового сигнала

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



$\mathcal{V}$: компиляция и отладочный вывод

Программная симуляция кода на языке $\mathcal{V}$ — это
самое обычное выполнение самой обычной программы (симулятора)

Устройство симулятора объяснялось в самом начале:

- данные — значения всех точек и число с плавающей точкой, обозначающее текущее время
- данные преобразуются согласно событиям,
  и время периодически увеличивается

Исполняемый файл симулятора можно собрать из исходного кода любым подходящим компилятором

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

$\mathcal{V}$: компиляция и отладочный вывод

Утилита `iverilog` во многом похожа на *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,,y,,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$
```

$\mathcal{V}$: компиляция и 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 выполяются в заданные моменты времени симуляции согласно выполнению процедур

$\mathcal{V}$: компиляция и отладочный вывод

Примеры таких команд:

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

Вспомогательные выражения:

- `$realtime`: возвращает текущее время
  как 64-битное *число с плавающей точкой*
- `$time`: возвращает текущее время
  как 64-битное *целое число* (*с округлением*)
- `$stime`: возвращает текущее время как 32-битное *целое число*

$\mathcal{V}$: компиляция и отладочный вывод

Пример: наблюдение за переменными

```
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$
```

$\mathcal{V}$: получение симуляционной трассы

Используя следующие две процедурные команды, можно легко получить трассу выполнения схемы в отдельном файле:

- `$dumpfile("file")`:
  открыть файл `file` для записи трассы и начать запись
- `$dumpvars(level, objlist)`: начать 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 модулей требуются только непосредственно содержащиеся в них точки

Трасса записывается в особом текстовом формате (VCD), описанном в стандарте языка, и содержит информацию об

- иерархии отслеживаемых модулей и точек и
- изменении значений отслеживаемых точек

$\mathcal{V}$: получение симуляционной трассы

Ничто не запрещает совместить команды отладочного вывода и извлечения трассы:

```
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,,y,,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$
```

$\mathcal{V}$: получение симуляционной трассы

Ничто не запрещает совместить команды отладочного вывода и извлечения трассы:

```
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$
```

$\mathcal{V}$: получение симуляционной трассы

И если отладочного вывода недостаточно, чтобы понять, как выполняется схема, то из сохранённой трассы можно получить диаграмму отслеживаемых сигналов при помощи любого подходящего средства (например, `gtkwave`):

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