Языки описания схем

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

Блок 8
Verilog:
базовый синтаксис

лектор:
Подымов Владислав Васильевич

e-mail:
valdus@yandex.ru

Осень 2018

(V) Немного о семантике

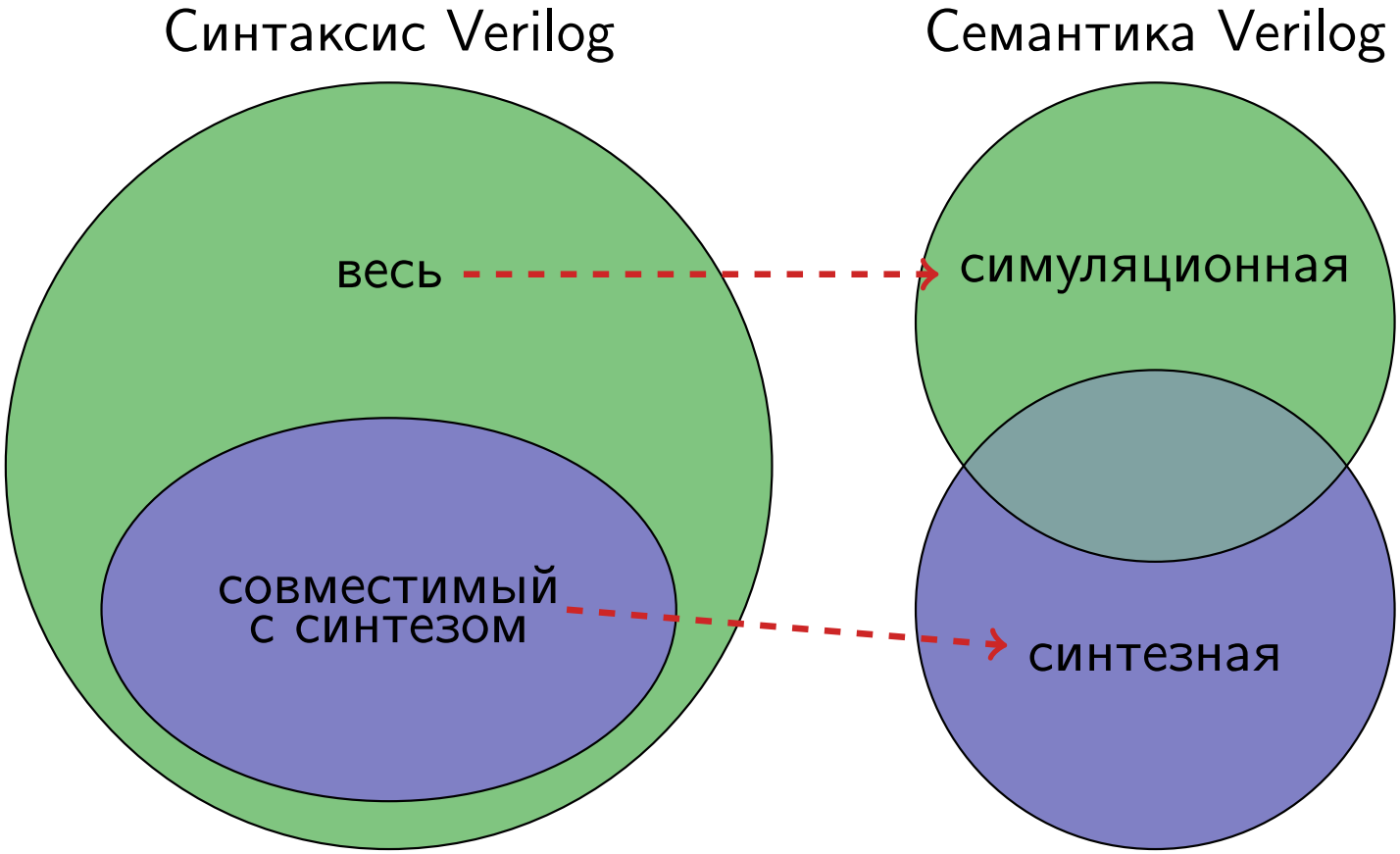
Verilog (*и его расширения/модификации*) — это один из двух самых популярных на данный момент языков описания цифровых интегральных схем (*второй — VHDL*)

Изначально этот язык создавался для симуляции:

- реальная схема разрабатывалась при помощи других средств
- на языке Verilog описывалась модель, поведение которой приблизительно соответствовало схеме
- эта модель компилировалась и запускалась (*как обычная программа*), и в результате работы выдавала информацию о преобразовании логических значений во времени и другой отладочный вывод

Язык оказался настолько удобным, что стал повсеместно применяться и для синтеза реальных схем

(V) Немного о семантике



В этом блоке слайдов обсуждается базовая часть синтаксиса языка, а также семантика на содержательном уровне

(V) Аналогии с C/C++

Синтаксис языка Verilog местами очень похож на синтаксис языка C/C++

Следует иметь в виду, что это сходство поверхностно

Некоторые детали семантики одинаковых конструкций этих языков совпадают, но на этом схожесть заканчивается:

- программа, написанная на C/C++, транслируется в машинный код, последовательно выполняемый процессором
- схема, спроектированная на Verilog, транслируется в
  - программу, моделирующую изменение сигналов во времени, если используется симуляционная семантика
  - последовательную схему и затем реальную цифровую схему, если используется синтезная семантика

Аналогии с конструкциями языка C/C++ будут проводиться *в особом цвете*

(V) Логические значения

В переменных Verilog используются 4 логических значения:

- 0 — это логический ноль
- 1 — это логическая единица
- x — это неизвестное логическое значение
  - в реальной схеме x означает
    “0 или 1, а что именно, неизвестно/неважно”
- z — это состояние высокого импеданса
  - в симуляционной семантике значение z очень похоже на x
  - в реальной схеме z означает, что мы не можем управлять потенциалом в обозначенной точке
  - например, z на контакте может означать
    “если соединить контакт с Vcc или GND,
    то получившаяся цепь будет разомкнута”

Чтобы не путать значения x, z с другими именами, будем выделять их цветом: x, z

(V) Основные типы данных

Типы данных языка делятся на две категории:

- типы соединений, или типы проводов (*net data types*)
  - из этих типов сейчас важен один: `wire`
- типы переменных (*variable data types*)
  - из этих типов сейчас важен один: `reg`

Синтаксис объявления проводов и переменных
аналогичен синтаксису объявления переменных в C:

`wire a, b, c;`	—	объявление проводов с именами `a`, `b`, `c`
`reg x, y, z;`	—	объявление переменных с именами `x`, `y`, `z`

Так как слово “переменная” зарезервировано для объектов особых типов, будем называть провода, переменные и другие именованные объекты точками (в схеме)

(V) Основные типы данных

С точки зрения симуляционной семантики, типы `wire` и `reg` различаются только тем, в каких конструкциях разрешено использовать точки этих типов:

- `reg` — в обозначении точек, “имеющих память”, то есть способных сохранять значение и непрерывно выдавать последнее сохранённое значение
- `wire` — в обозначении точек, “не имеющих памяти”, то есть непрерывно выдающих результат применения булевой функции к аргументам — значениям в других точках

(V) Основные типы данных

С точки зрения синтезной семантики,

- `wire` — это провод, соединяющий элементы схемы — логические вентили и ячейки памяти
- `reg` может быть как элементом с памятью (регистром, триггером, защёлкой, ...), так и соединяющим проводом, в зависимости от функционала спроектированной схемы

Хотя "`reg`" и расшифровывается как "`register`", следует иметь в виду, что регистры в последовательных схемах и переменные типа `reg` — это не одно и то же

(V) Основные типы данных

Можно объявлять не только одиночные провода и переменные, но и шины (в терминологии стандарта языка — векторы):

```
type [msb:lsb] x;
```

- `type` — это тип (например, `wire` или `reg`)
- `x` — это имя объявляемой шины
- `msb` и `lsb` — целые числа,[1] индекс старшего и младшего битов

Например,

- `wire [5:0] x;` — объявление шины `x` ширины 6, полностью соответствующее определениям **блока 5**
- `reg [6:9] y;` — объявление шины ширины 4, содержащей переменные от `y[9]` (младший бит) до `y[6]` (старший бит)
- `reg [-1:1] z;` — допустимое объявление шины ширины 3 (числа могут быть любыми целыми)

---

[1] Более точно, это константные выражения — *то, к чему можно дописать `constexpr` в C++*

(V) Основные типы данных

`wire [5:0] x; reg [-1:1] z;`

Совокупность значений шины трактуется по умолчанию как беззнаковое целое число:

- если `(x[5] x[4] x[3] x[2] x[1] x[0]) = (10111)`, то в шине `x` хранится число 23
- если `(z[-1] z[0] z[1]) = (110)`, то в шине `z` хранится число 6

Язык Verilog содержит немало арифметических операций над шинами, и во всех этих операциях подразумевается такая трактовка совокупностей логических значений

Исключение: некоторые ключевые слова позволяют изменять эту трактовку

Например, если дописать в начало объявления шины слово `signed`, то, *как и в C/C++*, число считается знаковым (в дополнительном коде, знак — старший бит)

(V) Модули

Модуль в Verilog — это

- то, что строго определялось как “модуль” в **блоке 5**
- *понятие, аналогичное “классу” и иногда “функции” и “функтору” в C/C++*

Модуль состоит из имени, портов и тела

Тело — это описание того, как функционирует модуль

Порт — это имя, обозначающее точку, значение которой доступно непосредственно извне модуля (*аргументы функции*)

Сейчас достаточно рассмотреть два вида портов:

входы (`input`) и выходы (`output`)

(V) Модули

Есть два равносильных способа определения модуля, которые можно проиллюстрировать на следующем примере: модуль `M` с входом `x1`, входной шиной `x2` ширины 3, выходом `y1` и выходной шиной `y2` ширины 2

Первый способ, с объявлением портов в аргументах модуля:

```
module M(input x1, input [2:0] x2,
         output y1, output [1:0] y2);
  // тело модуля
endmodule
```

(V) Модули

Есть два равносильных способа определения модуля, которые можно проиллюстрировать на следующем примере: модуль `M` с входом `x1`, входной шиной `x2` ширины 3, выходом `y1` и выходной шиной `y2` ширины 2

Второй способ, с объявлением портов вне аргументов модуля:

```
module M(x1, x2, y1, y2);
  // тело модуля
  input x1;
  input [2:0] x2;
  output y1;
  output [1:0] y2;
  // тело модуля
endmodule
```

(V) Модули

По умолчанию все порты модуля имеют тип `wire`:

```
input x          ≡  input wire x
input [a:b] x    ≡  input wire [a:b] x
output x         ≡  output wire x
output [a:b] x   ≡  output wire [a:b] x
```

Если требуется тип, отличный от `wire`, то этот тип можно указывать при определении порта:

```
input/output type x
input/output type [a:b] x
```

Входные порты обязательно должны быть проводами

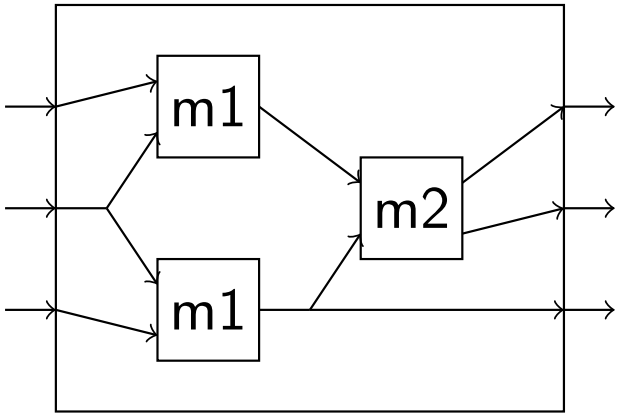
Тип выходных портов может быть любым

(V) Подходы к описанию модуля

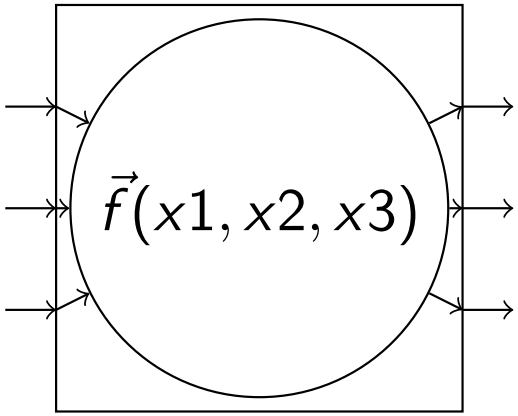
При описании поведения модуля (*в его теле*) используется сочетание двух подходов:

1. Структурный: описываются экземпляры других модулей (*объекты других классов*) и соединения между ними
   - например, описание D-триггера в блоке 4 как особым образом соединённые между собой две D-защёлки и одно отрицание — это структурное описание
2. Функциональный: способ преобразования сигналов описывается при помощи специальных конструкций без явного описания структуры этих конструкций в терминах других модулей и последовательных схем

Структурный подход



Функциональный подход



(V) Главный модуль

Так как при проектировании схемы с применением структурного подхода может описываться много модулей, среди них нужно явно выделить главный — ту самую схему, поведение которой описывается совокупностью модулей

В языке C/C++ главной функцией обязательно является функция с именем `main`. Язык Verilog в этом плане более гибкий:

- главный модуль (*top-level module*) — это модуль, экземпляры которого не используются в телах других модулей рассматриваемой совокупности
- главных модулей может быть несколько, если не требуется синтез одной конкретной схемы

Никаких особых требований к виду главного модуля не предъявляется: главный модуль — это такой же модуль, как и все остальные

(V) Непрерывное присваивание

```
assign x = E;
```

- `x` — это провод или шина проводов
- `E` — это комбинационное выражение: выражение над именами точек, константами и комбинационными операциями

Смысл: в каждый момент времени выставлять в `x` значение выражения `E`

Это присваивание можно трактовать так:

- `E` — это описание комбинационной схемы с единственным выходом или шиной выходов, входы которой — точки, используемые в выражении
- выходом комбинационной схемы назначается точка `x`

Далее описано **всё**, что может встречаться в `E`

Комбинационные операции имеют вид и смысл, максимально приближенный к стандартным операторам в C/C++, поэтому далее явно выписываются только некоторые характерные особенности операций

(V) Примеры комбинационных операций

Логические операции:

`&&` `||` `!`

Аргументы и результат — логические значения
Если результат существенно зависит от входного значения `x`, то результат — `x`
Эти операции можно трактовать как логические вентили И, ИЛИ, НЕ

Побитовые логические операции:

`&` (бинарная) `|` (бинарная) `^` (бинарная) `~`

Аргументы и результат — шины
Ширина аргументов выравнивается до большей нолями, результат имеет ту же ширину
Входное значение `x` в каждом бите учитывается так же, как и в обычных логических операциях

(V) Примеры комбинационных операций

Арифметические операции:

`+` (унарная и бинарная) `-` (унарная и бинарная) `*` `/` `%`

Аргументы и результат — шины логических значений (двоичные записи чисел)

Ширина аргументов выравнивается до большей, по умолчанию (но не всегда) нолями

Ширина результата — достаточная для хранения результата в худшем случае

Если хотя бы один бит входных значений — `x`, то все биты результата — `x`

Эти операции можно трактовать как “классические” схемы: сумматор, вычитатель, умножитель, . . .

(V) Примеры комбинационных операций

Арифметические (и логические) отношения:

`==` `!=` `>` `>=` `<` `<=`

Результат — логическое значение

Остальные свойства — как у арифметических отношений

Операции редукции: (все унарные)

`&` `|` `^` `~&` `~|` `~^`/`^~`

Аргумент — шина

Результат — логическое значение, получаемое при применении логической операции ко всем битам шины

(V) Примеры комбинационных операций

Другие операции

- логические сдвиги `<<`, `>>`
- индексация:
  - `x[i]` — `i`-й бит шины `x`
  - `x[a:b]` — шина, составленная из значений шины `x` от `x[a]` (старшего) до `x[b]` (младшего)
- конкатенация `{x, y, z, ...}`: шина значений, получаемых (от старшего к младшему) выписыванием значений аргументов: аргументы — слева направо, значения — от старшего к младшему
- репликация `{A{x}}`: конкатенация `A` копий шины `x`
- тернарный оператор `a ? x : y`: если логическое значение `a` — правда, то результат — `x`, иначе — `y`

(V) Примеры комбинационных операций

Пример

```
module sum(input [3:0] x,
           input [3:0] y,
           output [4:0] z);
  assign z = x + y;
endmodule
```

Это модуль сумматора 4-битовых чисел, записанный на языке Verilog

(V) Константы

Основной способ записи констант выглядит так:

<ширина>'<система счисления><значение>

Системы счисления:

- `b`: двоичная
- `o`: восьмеричная
- `d`: десятичная
- `h`: шестнадцатеричная

Примеры:

- `5'h1d` — число 29, записанное в пяти битах
- `18'b0100` — число 4, записанное в 18 битах
- `5'b0x00` — шина ширины 5, хранящая `x` в среднем бите и 0 в остальных

(V) Константы

Особые правила

“’<система счисления><значение>”

$\equiv$

“32’<система счисления><значение>”

“<значение>”

$\equiv$

“’d<значение>”

Если фактическая ширина значения больше ширины константы, то старшие биты отбрасываются до нужной ширины

Если фактическая ширина значения меньше ширины константы, то недостающие биты выбираются так:

- если старший бит значения — 0 или 1, то добавляются биты 0
- если старший бит значения — `x`, то добавляются биты `x`
- если старший бит значения — `z`, то добавляются биты `z`

(V) always-блок

`always @(watchlist) statement`

Смысл: каждый раз, когда в системе происходит одно из событий, записанных в `watchlist`, выполнять действие `statement`

Событие `watchlist` — это список следующих конструкций через запятую: (`x` — имя точки)

- `x`: "значение в точке `x` изменилось"
- `posedge x`: "в точке `x` наступил передний фронт"
- `negedge x`: "в точке `x` наступил задний фронт"

Переходы 1→`x` и `x`→0 расцениваются как задние фронты, а переходы 0→`x` и `x`→1 — как передние фронты

Составное действие — это последовательность действий, обрамлённая словами `begin`, `end`

(V) Блокирующее присваивание

$$x = E;$$

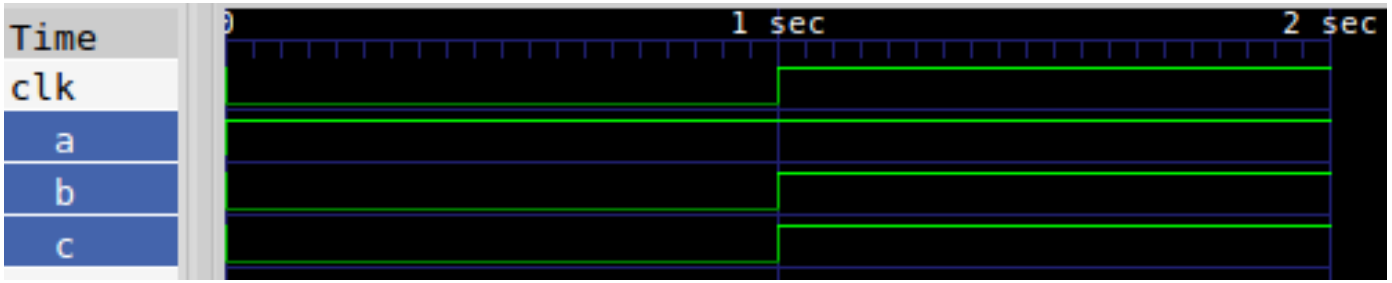
Это одно из действий, которые можно писать в always-блоке

- `x` — переменная
- `E` — комбинационное выражение

Смысл: изменить значение в точке `x` на `E`, и в следующих действиях always-блока использовать новое значение

Пример:

```
always @(posedge clk) begin
  b = a; c = b; a = c;
end
```



(V) Неблокирующее присваивание

x <= E;

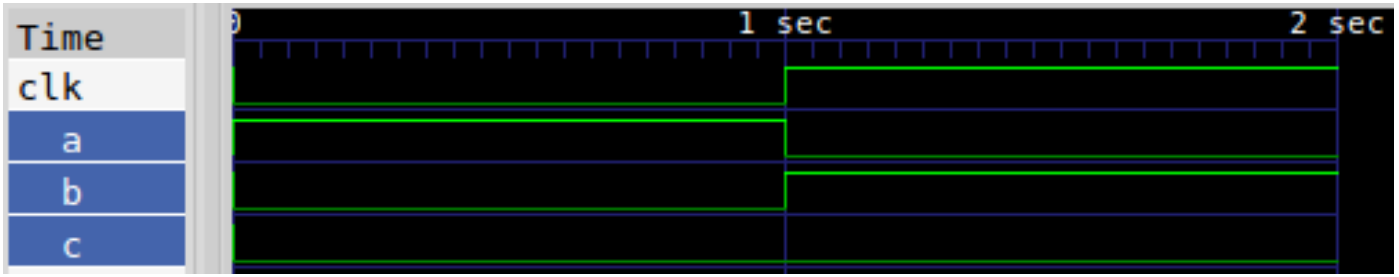
Это одно из действий, которые можно писать в always-блоке

- `x` — переменная
- `E` — комбинационное выражение

Смысл: вычислить значение `E` для текущих значений в точках, и после выполнения остальных одновременных действий изменить значение в точке `x` на `E` на вычисленное

Пример:

```
always @(posedge clk) begin
  b <= a; c <= b; a <= c;
end
```



(V) Ветвление

```
if(E) statement
else statement
```

Это одно из действий, которые можно писать в always-блоке

Значение выражения `E` должен быть однобитовым логическим

Смысл: если для текущих значений в точках верно `E`, то выполнить действие после `then`, а иначе — действие после `else`

Как и в C/C++, ветку "`else`" можно не писать

Значения `x` и `z`, *как правило*, трактуются как 0
(*переход в отрицательную ветку*)

(V) Выбор

```
case (E)
A: statement
B: statement
...
Z: statement
default: statement
endcase
```

Это одно из действий, которые можно писать в always-блоке

`A`, `B`, ..., `Z` — константы

Смысл: вычислить значение выражения `E` для текущих значений в точках и выполнить действие, записанное после соответствующей константы (если такой константы нет, то после `default`; если и `default` нет, то ничего не выполнять)

(V) Экземпляры модулей

Все описанные ранее конструкции — это элементы описания модуля согласно функциональному подходу

Структурный подход — это использование экземпляров модулей в теле другого модуля

Это делается при помощи такой конструкции:

```
<имя модуля> <имя экземпляра>(<соединение портов>);
```

<имя модуля> — это имя модуля, экземпляр которого вставляется в описываемый модуль (*тип объявляемого объекта*)

<имя экземпляра> — это уникальное имя вставляемого экземпляра (*имя объявляемого объекта*)

<соединение портов> — это *аргументы конструктора объекта:* описание того, какие точки схемы являются входами и выходами экземпляра; это список через запятую следующих записей:

```
.<имя порта>(<комбинационное выражение>)
```

(V) Экземпляры модулей

Пример

```
module sum2(input [1:0] x, input [1:0] y, output [2:0] z);
  assign z = x + y;
endmodule

module csum2(input c, input [1:0] x, input [1:0] y,
             output [2:0] z);
  assign z = x + y + c;
endmodule

module sum4(input [3:0] x, input [3:0] y, output [4:0] z);
  wire carry;
  sum2 first(.x(x[1:0]), .y(y[1:0]), .z({carry, z[1:0]}));
  csum2 second(.x(x[3:2]), .y(y[3:2]),
               .c(carry), .z(z[4:2]));
endmodule
```