Языки описания схем

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

Блок 11

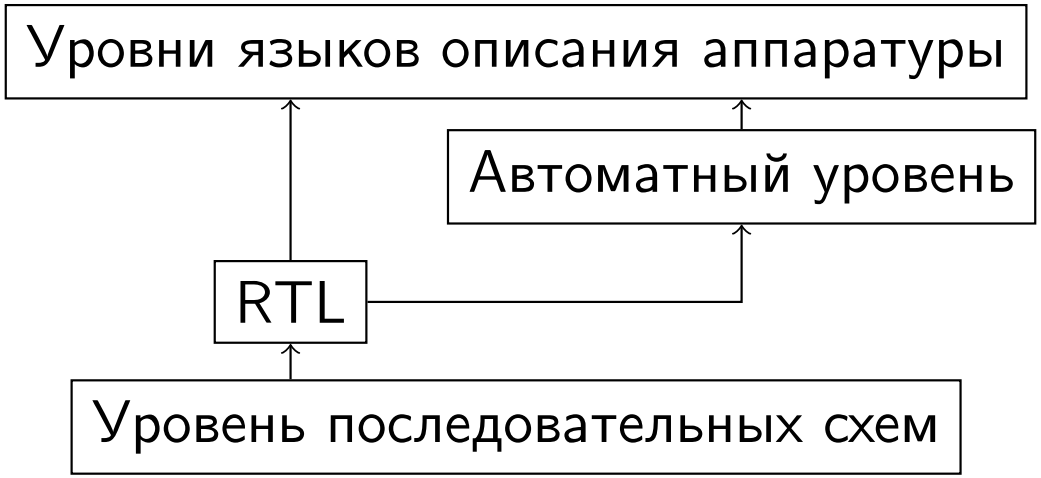
Verilog:
от логических значений
до комбинационных схем

Лектор:
Подымов Владислав Васильевич

E-mail:
valdus@yandex.ru

ВМК МГУ, 2020, осенний семестр

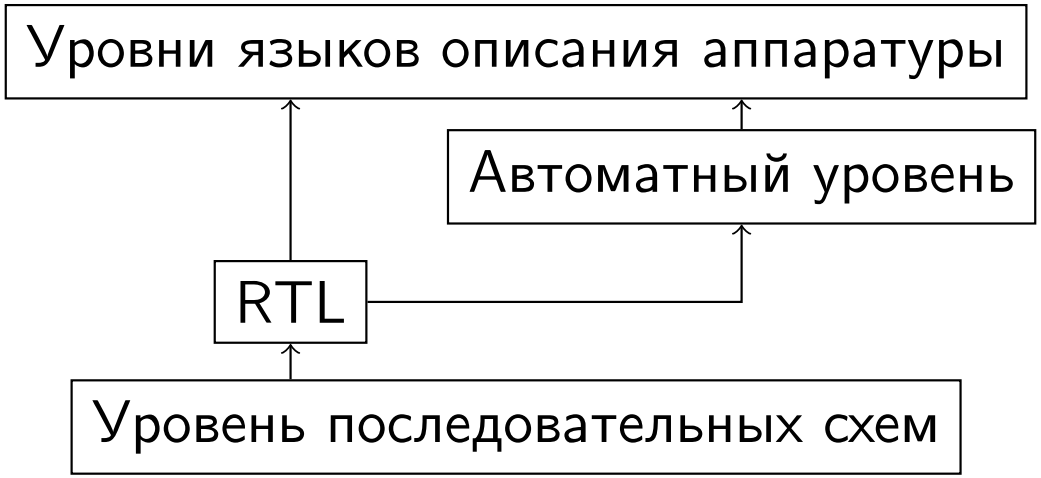
Вступление



Разработка *последовательной схемы* и/или *RTL-описания схемы* — трудоёмкий и неустойчивый к ошибкам процесс:

- имеется *декларативное* описание поведения схемы
- из этого описания *методом пристального взгляда* извлекается основная масса триггеров/регистров
- схема *вручную* дополняется логическими вентилями/булевыми функциями, соединениями и вспомогательными триггерами/регистрами

Вступление



В *языках описания аппаратуры* используются понятия и подходы,
более близкие к декларативному описанию схем
и позволяющие меньше задумываться
о точной расстановке элементов схемы при её разработке

Язык Verilog ($\mathcal{V}$)

Verilog[1] — это один из двух самых популярных на данный момент языков описания цифровых микросхем[2]

Изначально этот язык создавался для программной симуляции схем:

- схема разрабатывается другими средствами
- на языке Verilog описывается **программная** модель, поведение которой приблизительно соответствует поведению схемы
- модель запускается (*как обычная программа*) и выдаёт информацию об изменении значений сигналов во времени и другие отладочные данные

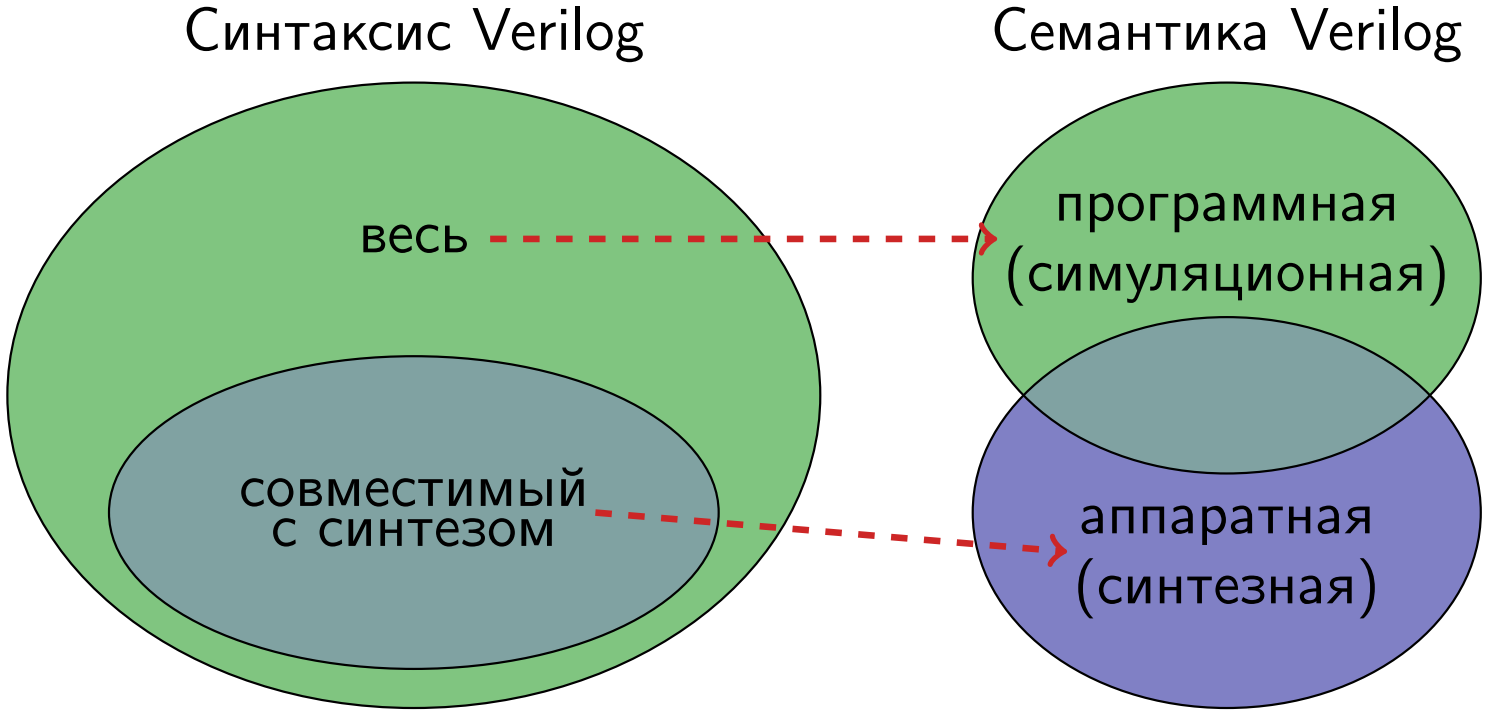
Язык оказался настолько удобным,
что стал повсеместно применяться и для синтеза реальных схем

---

[1] Если считать его расширения и модификации, например, SystemVerilog

[2] Два самых популярных в мире языка описания аппаратуры — Verilog и VHDL
Эти языки похожи, при этом синтаксис Verilog проще, так что остановимся на нём

Язык Verilog ($\mathcal{V}$)



Два главных документа, описывающих $\mathcal{V}$ как

(1) средство симуляции и (2) средство синтеза:

1. IEEE Standard for Verilog Hardware Description Language
   (*в курсе обсуждается версия **2005***)
2. IEEE Standard for Verilog Register Transfer Level Synthesis
   (*в курсе обсуждается версия **2002***)

$\mathcal{V}$ и C/C++

Синтаксис Verilog местами очень похож на синтаксис C/C++[1]

Это сходство позволит избежать долгих объяснений “с ноля”,
но следует иметь в виду, что оно поверхностно:

- Итог сборки кода
  - C/C++: машинный код, выполняемый процессором
  - Verilog: цифровая микросхема (*в том числе и сам процессор*)
- Трактовка переменных:
  - C/C++: последовательно изменяемые области памяти
  - Verilog: выделенные точки микросхемы
- Трактовка выражений и команд:
  - C/C++: команды машинного кода
  - Verilog: наборы логических вентилей и описание поведения подсхем
- ...

[1] И не просто так похож: создатели языка перенесли много синтаксических деталей из C, чтобы язык был *интуитивно понятнее*

$\mathcal{V}$: логические значения

В *C/C++* используются два логических значения:
истина (**true**) и *ложь* (**false**)

В $\mathcal{V}$ используются **четыре** логических значения:

- 1: истина, единица, высокий уровень напряжения
- 0: ложь, ноль, низкий уровень напряжения
- $\mathcal{X}$: неопределённость
- $\mathcal{Z}$: высокий импеданс

Константы, соответствующие этим логическим значениям, записываются так:[1]

```
1'b1    1'b0    1'bx    1'bz
```

[1] Это часть более широкого синтаксиса констант, но об этом позже

$\mathcal{V}$: логические значения $\mathcal{X}$ и $\mathcal{Z}$

Как понимать значение $\mathcal{X}$:

- **Когда задаётся разработчиком**:
  "мне неважно, что будет в этой точке схемы в этот момент"
- **В программной семантике**:
  полноценное значение в семантике выражений с приблизительной трактовкой "невозможно однозначно определить значение"
- **В реальной схеме** есть только конкретные напряжения, и средство синтеза доопределяет $\mathcal{X}$ до 0 или 1 по своему усмотрению

Как понимать значение $\mathcal{Z}$:

- Коротко и огрублённо:
  "эта точка схемы изолирована от входных напряжений 0 и 1"
- Или так:
  "значение в точке не задаётся этой конструкцией языка"

В курсе значение $\mathcal{Z}$ почти не обсуждается из-за сопутствующих технических нагромождений, не особо важных в начале изучения $\mathcal{V}$

Для простоты будем считать, что
если значение $\mathcal{Z}$ не упоминается, то его в языке не существует

$\mathcal{V}$: основные типы данных

Типы данных $\mathcal{V}$ делятся на две категории:

1. Типы соединений (*net data types*)
   - Пример такого типа: `wire` (провод)
2. Типы переменных (*variable data types*)
   - Пример такого типа: `reg`

Соединения и переменные будем называть точками схемы

($\sim$ *переменные* в *C/C++*)

Объявления точек в $\mathcal{V}$ устроены так же,
как и объявления переменных в *C/C++*

- Объявление точек типа `wire` с именами `a`, `b`, `c`:

```
wire a, b, c;
```

- Объявление точек типа `reg` с именами `u`, `v`, `w`:

```
reg u, v, w;
```

$\mathcal{V}$: основные типы данных

В простых случаях (`wire`, `reg`)
каждой точке сопоставляется некоторый *сигнал*,
устройство которого зависит от устройства сигналов в других точках

В аппаратной семантике сигнал определяется так,
как велит уровень абстракции описания схемы

В программной семантике сигнал
принимает *четыре логических значения* и имеет *мгновенные фронты*

Фронты сигнала, *положительные* ($\uparrow$) и *отрицательные* ($\downarrow$),
определяются так:

до \ после	0	$\mathcal{X}$	1
0		$\uparrow^1$	$\uparrow$
$\mathcal{X}$	$\downarrow^1$		$\uparrow^1$
1	$\downarrow$	$\downarrow^1$	

[1] При разработке схемы следует по возможности избегать фронтов, связанных со значением $\mathcal{X}$

$\mathcal{V}$: основные типы данных

Категория типа каждой точки должна соответствовать тому, как точка **синтаксически** используется в коде схемы:

- некоторые конструкции гарантированно не имеют памяти (задают комбинационные схемы), и выходы таких конструкций обязаны быть соединениями
- некоторые конструкции в общем случае имеют память (способны задавать схемы с памятью), и выходы таких конструкций обязаны быть переменными

Замечание: название типа "`reg`" происходит от "`register`" ("регистр"), но это название не очень удачно: точками этого типа можно задавать сигналы **не только** на выходах регистров (и, в частности, триггеров)

Очевидное синтаксическое ограничение, подразумевающееся во всех дальнейших описаниях: каждое логическое значение в каждой точке схемы должно задаваться **не более чем одной** языковой конструкцией[1]

[1] Это неверно в общем случае, но часто верно в синтезируемом коде, так что будем считать это правилом и отдельно обозначать все исключения

$\mathcal{V}$: шины

Объявление *шин* (в терминологии стандарта — векторов) в $\mathcal{V}$:

```
type [msb:lsb] id1, id2, ...;
```

- `type` — *невекторный* тип (`wire`, `reg`, ...)
- `id1, id2, ...` — имена точек
- `msb` и `lsb` — номера́ старшего и младшего разрядов шины
- "`type [msb:lsb]`" — тип той же категории, что и `type`

Примеры:

- Шина проводов `x` ширины 5:

```
wire [4:0] x;
```

- Шина `reg`-ов `y` ширины 3:

```
reg [2:0] y;
```

- То же, что и `y`, но разряды нумеруются с двойки:

```
reg [4:2] z;
```

$\mathcal{V}$: шины

В $\mathcal{V}$ используются **две** арифметических трактовки значения в шине:

- Беззнаковая:
$$(\alpha_{n-1} \dots \alpha_0)_2 = \sum_{i=0}^{n-1} 2^i \cdot \alpha_i$$
- Знаковая (дополнительный код):
$$(\alpha_{n-1} \dots \alpha_0)_2^- = (\overline{\alpha_{n-1}}\alpha_{n-2} \dots \alpha_0)_2 - 2^{n-1}$$

По умолчанию шины полагаются беззнаковыми

Арифметическая трактовка корректна, если в шине содержатся **только** значения 0, 1 — иначе соответствующее число не определено

Одноразрядная точка обычно
отождествляется с шиной точек ширины 1

$\mathcal{V}$: порты

Портами в $\mathcal{V}$ *и в целом в области схемотехники* называются точки схемы, через которые она взаимодействует с окружением (ранее в лекциях это называлось *контактами* и *шинами контактов*)

Сейчас достаточно рассмотреть два вида портов:

- Входные порты, или просто входы — через эти порты сигналы посылаются в схему извне
  - обозначаются ключевым словом `input`
- Выходные порты, или просто выходы — через эти порты сигналы отправляются из схемы вовне
  - обозначаются ключевым словом `output`

$\mathcal{V}$: модули

Модуль в $\mathcal{V}$ — это

- описание (под)схемы
- понятие, аналогичное *классу*/*функции*/*функтору* языка *C/C++*

В модуле содержатся, *в числе прочего*:

- имя (∼ *имя класса/функции*)
- объявление портов (∼ *объявление аргументов функции*)
- тело (∼ *тело функции*)

В теле модуля описывается схема с портами (входами, выходами, ...), задаваемыми модулем

Экземпляры модуля (∼ *объекты класса*) можно вставлять в другие модули в качестве подсхем, задавая соединения портов подсхемы с точками схемы

$\mathcal{V}$: модули

Первый способ объявления модуля:

```
module <имя модуля>(<объявления портов через запятую>);
  <тело модуля>
endmodule
```

Пример: модуль `M` с входными проводами `a`, `b`,
выходной шиной проводов `u` ширины 2 и выходным `reg`-ом `v`

```
module M(input wire a, b,
         output wire [1:0] u, output reg v);
  // тело модуля
endmodule
```

(комментарии в $\mathcal{V}$ устроены так же, как и в C/C++)

$\mathcal{V}$: модули

Второй способ объявления модуля:

```
module <имя модуля>(<имена портов через запятую>);
  <объявления портов>
  <тело модуля>
endmodule
```

Пример: модуль `M` с входными проводами `a`, `b`,
выходной шиной проводов `u` ширины 2 и выходным `reg`-ом `v`

```
module M(a, b, u, v);
  input wire a, b;
  output wire [1:0] u;
  output reg v;
  // тело модуля
endmodule
```

$\mathcal{V}$: модули

Слово `wire` в объявлении портов можно опускать:

```
module M(input /* wire */ a, b,
         output /* wire */ [1:0] u, output reg v);
  // тело модуля
endmodule
```

```
module M(a, b, u, v);
  input /* wire */ a, b;
  output /* wire */ [1:0] u;
  output reg v;
  // тело модуля
endmodule
```

$\mathcal{V}$: модули

Объявление типа и объявление порта **независимы**:

```
module M(input a, b,
         output u, v);
  // тело модуля
  wire [1:0] u;
  reg v;
  // тело модуля
endmodule
```

```
module M(a, b, u, v);
  input a, b;
  output u, v;
  // тело модуля
  wire [1:0] u;
  reg v;
  // тело модуля
endmodule
```

Синтаксическое ограничение: все входы должны быть соединениями

Можно:

```
input wire [1:0] a;
```

Нельзя:

```
input reg [1:0] a;
```

$\mathcal{V}$: непрерывное присваивание (assign)

```
assign x = E;
```

- x — соединение, **не** являющееся входом
- E — комбинационное выражение: выражение, составленное из
  - точек,
  - констант и
  - специальных (комбинационных) операций

Содержательный смысл такой конструкции:
в каждый момент времени
значение в точке x совпадает со значением выражения E

Такой смысл имеют все комбинационные схемы

Пара простых примеров:

- сигнал из y без изменений направляется в соединение x:

```
assign x = y;
```

- значение провода x — сигнал с константным значением 1:

```
assign x = 1'b1;
```

$\mathcal{V}$: константы

Общий способ записи константных значений:

<ширина>'<система счисления><значение>

Системы счисления:

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

Пример: шина ширины 5 со значением 29

```
5'b11101
5'd29
5'o35
5'h1d
```

Упрощённая запись констант:

'<с.с.><значение> = N'<с.с.><значение>,

где N — неспецифицированная ширина не менее 32

<значение> = 'd<значение>

$\mathcal{V}$: список основных комбинационных операций

```
x && y        x || y        !x

x + y   x - y   +x   -x   x * y   x / y   x % y

x == y   x != y   x > y   x >= y   x < y   x <= y

x << y      x >> y      x <<< y      x >>> y

x & y       x | y       x ^ y       ~x

&x      |x      ^x      ~&x      ~|x      ~^x

x ? y : z

x[i]          x[i:j]

$signed(x)    $unsigned(x)
```

Очень похоже на операции языка C/C++ —
это действительно так, но только содержательно и только отчасти

$\mathcal{V}$: аппаратная семантика `assign`

Аппаратная семантика присваивания "`assign x = E;`" — произвольная *комбинационная схема* Σ следующего вида:

- Входы Σ — все точки, используемые в `E`
- Выход Σ — шина `x`
- Программной семантикой
  для каждого набора **булевых** значений $(\alpha_1, \ldots, \alpha_n)$ на входах
  задаётся результат выражения E —
  набор $(\beta_1, \ldots, \beta_k)$ **булевых** значений на выходе
- При *посылке* значений $(\alpha_1, \ldots, \alpha_n)$ на входы Σ
  на выходах *получаются* значения $(\beta_1, \ldots, \beta_k)$

$\mathcal{V}$: расширение и сужение шин

В семантику некоторых комбинационных операций включены механизмы выравнивания ширины аргументов: расширения слишком узких и сужения слишком широких

Сужение всегда устроено одинаково: отбрасываются старшие разряды

$$(x_{n-1}x_{n-2}\ldots x_k x_{k-1}x_{k-2}\ldots x_0) \rightsquigarrow (x_{k-1}x_{k-2}\ldots x_0)$$

Расширение, как правило, относится к одному из двух видов:

1. Беззнаковое: в старшие разряды дописывается 0
$$(x_{k-1}x_{k-2}\ldots x_0) \rightsquigarrow (00\ldots 0 x_{k-1}x_{k-2}\ldots x_0)$$
2. Знаковое: старший разряд "размножается" нужное число раз
$$(x_{k-1}x_{k-2}\ldots x_0) \rightsquigarrow (x_{k-1}x_{k-1}\ldots x_{k-1}x_{k-1}x_{k-2}\ldots x_0),$$
то есть
$$\begin{aligned}
(0x_{k-2}\ldots x_0) &\rightsquigarrow (00\ldots 00x_{k-2}\ldots x_0)\\
(1x_{k-2}\ldots x_0) &\rightsquigarrow (11\ldots 11x_{k-2}\ldots x_0)\\
(\mathcal{X}x_{k-2}\ldots x_0) &\rightsquigarrow (\mathcal{X}\mathcal{X}\ldots\mathcal{X}\mathcal{X}x_{k-2}\ldots x_0)\\
(\mathcal{Z}x_{k-2}\ldots x_0) &\rightsquigarrow (\mathcal{Z}\mathcal{Z}\ldots\mathcal{Z}\mathcal{Z}x_{k-2}\ldots x_0)
\end{aligned}$$

$\mathcal{V}$: знаковость результата выражения

Каждое значение в комбинационном выражении
трактуется либо как *знаковое*, либо как *беззнаковое*

Этим определяются *арифметическая трактовка* значения
и способ *расширения* шины по умолчанию

Язык $\mathcal{V}$ "тяготеет" к беззнаковой трактовке:
согласно стандартной схеме учёта знаковости аргументов,

- результат выполнения операции объявляется беззнаковым $\Leftrightarrow$
  хотя бы один из аргументов беззнаковый
- перед вычислением результата
  знаковость аргументов приравнивается знаковости результата
  - Расширение и сужение шин выполняется после смены знаковости
  - При изменении знаковости аргументов их значения не меняются —
    меняется **только** арифметическая трактовка этих значений

$\mathcal{V}$: комбинационные операции

Логические операции

`x && y` `x || y` `!x`

Программная семантика: результат — беззнаковая шина ширины 1, значение которой определяется согласно таблицам

`x && y`

x \ y	0	$\mathcal{X}$	1
0	0	0	0
$\mathcal{X}$	0	$\mathcal{X}$	$\mathcal{X}$
1	0	$\mathcal{X}$	1

`x || y`

x \ y	0	$\mathcal{X}$	1
0	0	$\mathcal{X}$	1
$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	1
1	1	1	1

`!x`

x	!x
0	1
$\mathcal{X}$	$\mathcal{X}$
1	0

При вычислении результата аргументы сужаются до ширины 1

$\mathcal{V}$: комбинационные операции

Арифметические операции

```
x + y    x - y    +x    -x    x * y    x / y    x % y
```

Программная семантика:

- Результат — естественный для арифметики *с переполнением*
  - Если хотя бы один из аргументов *не определён*,
    то результат — $(\mathcal{X}\mathcal{X}...\mathcal{X})$
- Знаковость аргументов учитывается *стандартно*
- Ширина выражения — это максимум ширины аргументов и
  - если операция является внешней,
    то ширины левой части присваивания
  - иначе — ширины "объемлющего" выражения, если эта ширина задана
- **Ширина результата** равна ширине выражения
  - При вычислении результата
    аргументы расширяются до ширины выражения

$\mathcal{V}$: комбинационные операции

Арифметические отношения:

`x == y` `x != y` `x > y` `x >= y` `x < y` `x <= y`

Программная семантика:

- Результат — беззнаковая шина ширины 1 со следующим значением:
  - если оба числá-аргумента *определены*, то
    - 1, если чи́сла входят в отношение, и
    - 0, если не входят
  - иначе результат — $\mathcal{X}$
- Знаковость аргументов учитывается *стандартно*
  - *Но результат всё равно беззнаковый*
- При вычислении результата
  узкий аргумент расширяется до широкого

$\mathcal{V}$: комбинационные операции

Сдвиговые операции:

`x << y` `x >> y` `x <<< y` `x >>> y`

Программная семантика:

Результат имеет ту же ширину и знаковость, что и `x`

“<<” **и** “<<<”:
сдвиг `x` влево на `y` разрядов с заполнением нолями

“>>”: (*логический сдвиг*)
результат — сдвиг `x` вправо
на `y` разрядов с заполнением нолями

“>>>”: (*арифметический сдвиг*)
результат — сдвиг `x` вправо на `y` разрядов
с заполнением при помощи расширения шины

Если число `y` не определено, то результат — $(\mathcal{X}\mathcal{X}\ldots\mathcal{X})$

$\mathcal{V}$: комбинационные операции

Многобитовые логические операции:

`x & y` `x | y` `x ^ y` `~x`

Программная семантика:

- **Результат:**
  каждый разряд получается из соответствующих разрядов `x` и `y` применением соответствующей операции: (`&&`, `||`, `!=`, `!`)
- **Ширина результата** равна ширине самого широкого аргумента
  - Узкий аргумент расширяется до широкого беззнаково
- Знаковость аргументов учитывается стандартно

$\mathcal{V}$: комбинационные операции

Операции редукции:

`&x` `|x` `^x` `~&x` `~|x` `~^x`

Программная семантика:

Результат — беззнаковая шина ширины 1 со следующим значением:

- Для операции без “~”:
  соответствующая логическая операция (`&&`, `||`, `!=`)
  применяется к паре младших разрядов, и затем,
  итеративно до конца шины,
  к результату предыдущего шага и следующему разряду
- Для операции с “~”:
  отрицание (`!`) результата
  соответствующей операции без “~”

$\mathcal{V}$: комбинационные операции

Тернарный оператор:

```
x ? y : z
```

Программная семантика:

- **Ширина результата** — это максимум ширин `y` и `z`
  - Узкий аргумент `y`/`z` расширяется до широкого беззнаково
- **Результат** зависит от значения условия — выражения "`x == 0`":
  - $1 \Rightarrow$ результат совпадает с `z`
  - $0 \Rightarrow$ результат совпадает с `y`
  - $\mathcal{X} \Rightarrow$ i-й разряд результата — это
    - i-й разряд `y`, если i-е разряды `y` и `z` равны
    - $\mathcal{X}$ иначе
- Знаковость аргументов `y` и `z` учитывается стандартно

$\mathcal{V}$: комбинационные операции

Операции индексации

```
x[i]                    x[i:j]
```

Программная семантика: если `x` — шина, то

- `x[i]` — соединение шины `x` с индексом `i`
- `x[i:j]` — шина (`x[i]` ... `x[j]`)

Операции конкатенкции (слева) и репликации (справа)

```
{x1, x2, ..., xn}          {N{x1, x2, ..., xn}}
```

Программная семантика: если `xi` — шина ширины `ki`, то

- `{x1, x2, ..., xn}` — беззнаковая шина
  `(x1[ki-1]...x1[0] x2[k2-1]...x2[0] ... xn[kn-1]...xn[0])`
- `{N{x1, x2, ..., xn}}` равносильно конкатенации `N` копий шины
  `{x1, x2, ..., xn}`

Замечание: индексация и конкатенация
могут использоваться в левых частях присваиваний

$\mathcal{V}$: особенности работы со знаковостью

Операции изменения знака

`$signed(x)` `$unsigned(x)`

Программная семантика:

- ширина и значение результата
  совпадают с шириной и значением аргумента
- результат трактуется как знаковый (`$signed`)
  или беззнаковый (`$unsigned`)

$\mathcal{V}$: особенности работы со знаковостью

Для всех точек по умолчанию применяется беззнаковая трактовка

Чтобы это изменить, следует добавть слово `signed` в объявление:

```
input signed x;
wire signed y;
reg signed [1:0] z;
```

Константы вида "*<значение>*" по умолчанию трактуются как знаковые

Остальные константы

(*'<с.с.><значение>*, *<ширина>'<с.с.><значение>*)

по умолчанию трактуются как беззнаковые

Знаковые константы с указанием системы счисления выглядят так:

's<с.с.><значение> <ширина>'s<с.с.><значение>

$\mathcal{V}$: примеры (модуль + assign)

```
module M(input x, y, output u, v);
  assign u = x && y || !x && !y;
  assign v = !u;
endmodule
```

Программная семантика модуля:

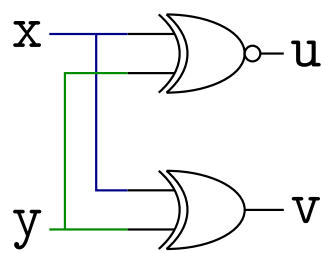
u

x \ y	0	$\mathcal{X}$	1
0	1	$\mathcal{X}$	0
$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$
1	0	$\mathcal{X}$	1

v

x \ y	0	$\mathcal{X}$	1
0	0	$\mathcal{X}$	1
$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$
1	1	$\mathcal{X}$	0

Одна из **аппаратных семантик** модуля:



$\mathcal{V}$: примеры (модуль + assign)

```
module M(input x, y, output u, v);
  assign u = x && y || !x && !y;
  assign v = !u;
endmodule
```

Объявление соединения и assign можно “совмещать”:

```
wire x;
assign x = E;
```
$=$ `wire x = E;`

```
module M(input x, y, output u, v);
  wire tmp1 = x && y;
  wire tmp2 = !x && !y;
  assign u = tmp1 || tmp2;
  assign v = !u;
endmodule
```

$\mathcal{V}$: примеры (модуль + assign)

```
module M(input [1:0] x, input y, output [1:0] u);
  assign {u[0], u[1]} = x ^ {x[0], y};
endmodule
```

Программная семантика модуля:

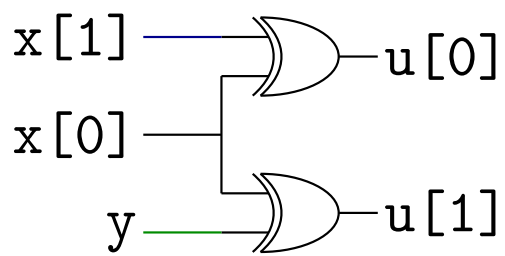
u[0]

x[0] \ x[1]	0	$\mathcal{X}$	1
0	0	$\mathcal{X}$	1
$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$
1	1	$\mathcal{X}$	0

u[1]

x[0] \ y	0	$\mathcal{X}$	1
0	0	$\mathcal{X}$	1
$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$
1	1	$\mathcal{X}$	0

Одна из **аппаратных семантик** модуля:



$\mathcal{V}$: использование подсхем

В теле модуля можно использовать

экземпляры других модулей в качестве подсхем

Синтаксис вставки экземпляра:

```
<имя модуля> <имя экземпляра>
  (<назначения портов через запятую>);
```

Рекомендуемый синтаксис назначения порта:

```
.<имя порта>(<выражение>)
```

Смысл такого назначения порта:

- входной порт: выполняется непрерывное присваивание произвольного `<выражения>` в порт подсхемы
- выходной порт:
  - выполняется непрерывное присваивание порта подсхемы в `<выражение>`
  - допускаются только `<выражения>`, которые разрешено располагать в левой части непрерывного присваивания (“правильно собранные” из переменных, индексаций и конкатенаций)

$\mathcal{V}$: использование подсхем

Порядок назначений портов экземпляра
и порядок объявлений портов соответствующего модуля
не обязаны совпадать при использовании рекомендуемого синтаксиса

Если входной порт экземпляра не назначен,
то на этот вход посылается значение $\mathcal{Z}$[1]

Если выходной порт экземпляра не назначен,
то *ничего страшного не просходит*

[1] Если не уверены, действительно ли так нужно, то старайтесь этого избегать

$\mathcal{V}$: использование подсхем

Пример напоследок: реализация сумматора трёхразрядных чисел[1]

```
module adder_cell(input x, y, cin, output sum, cout);
  assign {cout, sum} = x + y + cin;
endmodule
```

```
module adder(input [2:0] x, y, output [3:0] z);
  adder_cell cell1(.x(x[0]), .y(y[0]), .cin(1'b0),
                   .sum(z[0]), .cout(c1)); // [2]
  adder_cell cell2(.x(x[1]), .y(y[1]), .cin(c1),
                   .sum(z[1]), .cout(c2));
  adder_cell cell3(.x(x[2]), .y(y[2]), .cin(c2),
                   .sum(z[2]), .cout(z[3]));
endmodule
```

[1] Не реализуйте сумматор так!
Это просто демонстрация возможностей языка

[2] **ВНИМАНИЕ!** Каждая необъявленная точка *по определению* имеет тип `wire` — в том числе точки c1 и c2