Языки описания схем

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

Блок 10

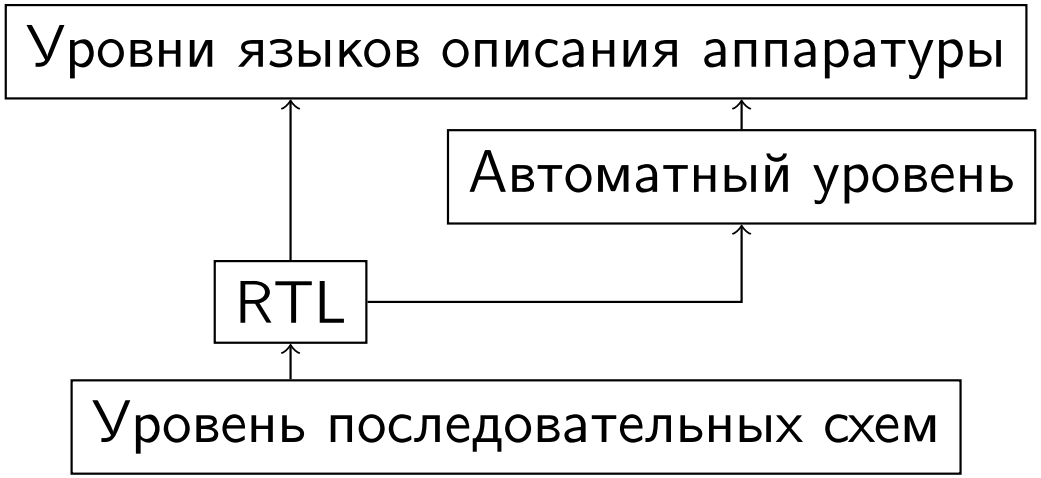
Verilog:
от логических значений
до комбинационных схем

Лектор:
Подымов Владислав Васильевич

E-mail:
valdus@yandex.ru

ВМК МГУ, 2019, осенний семестр

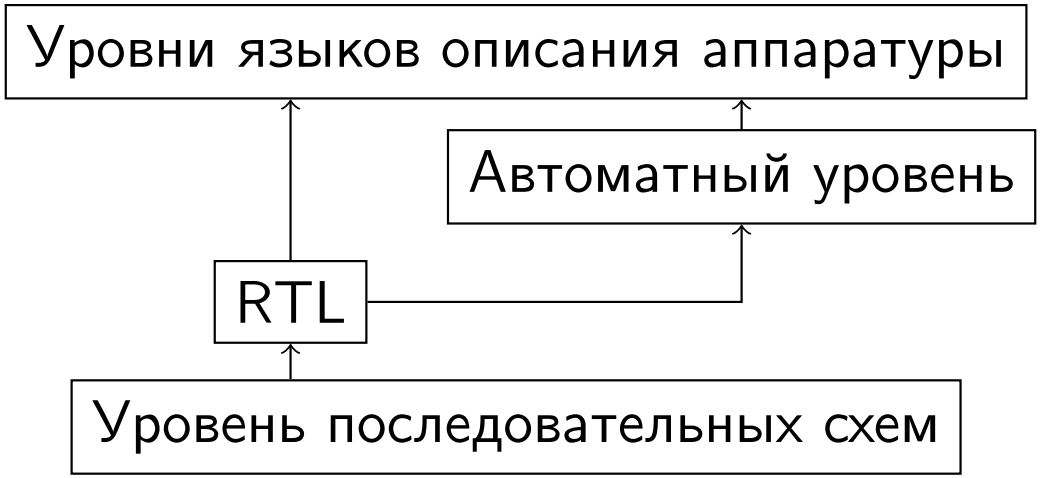
Вступление



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[2] самых популярных на данный момент языков описания цифровых микросхем

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

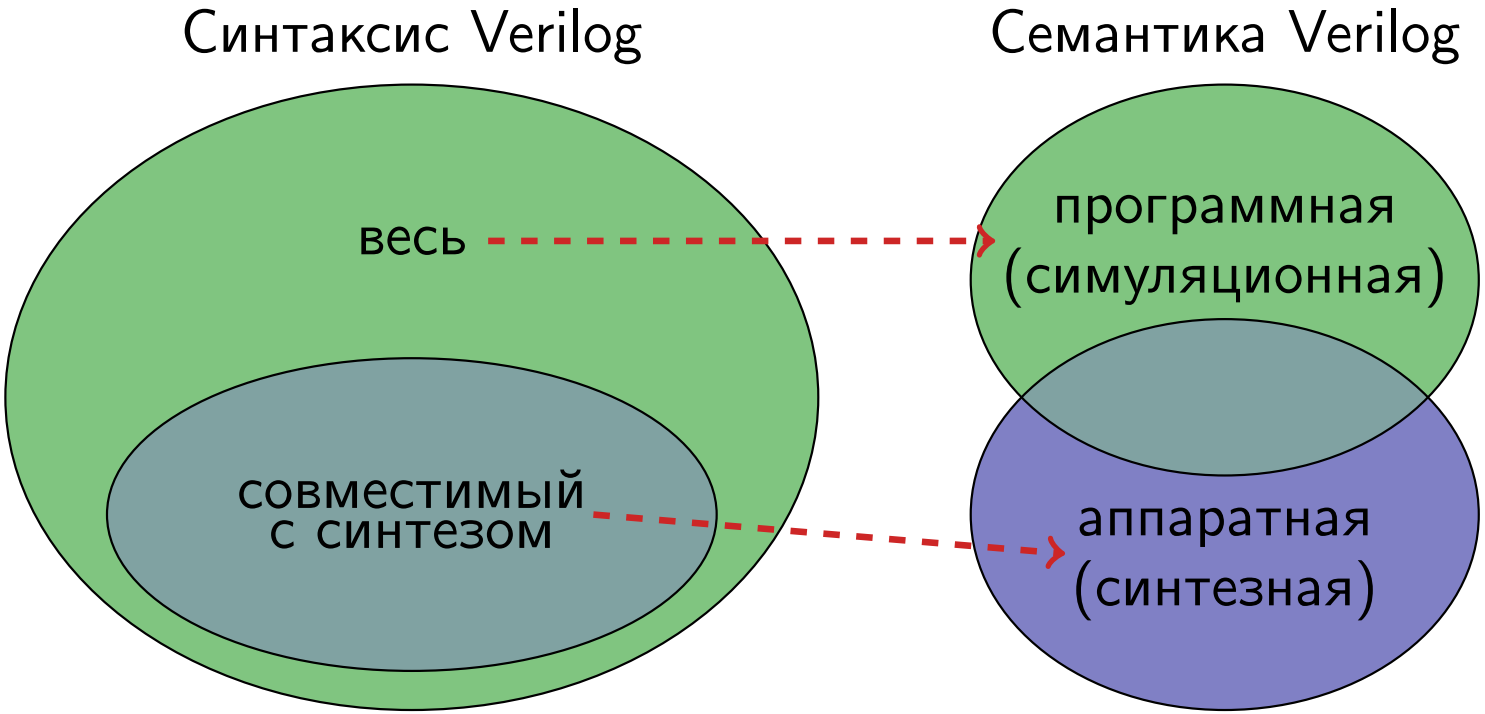
Язык оказался настолько удобным,
что стал повсеместно применяться и для синтеза реальных схем

---

[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
- `x`: неопределённость, любое из значений 0, 1
- `z`: состояние высокого импеданса

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 x и z

Как понимать значение 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 x до 0 или 1 по своему усмотрению

Как понимать значение z:

- Коротко и огрублённо:
  “эта точка схемы изолирована от входных напряжений 0 и 1”
- Чуть более точно:
  “напряжение в соединении задаётся другими точками схемы”

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

$\mathcal{V}$: 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`) значением точки является *сигнал*, и значением в кажый момент времени — одно из логических значений (*на "стыке" значений — фронт*)

В аппаратной семантике определение сигнала задаётся уровнем описания цифровой схемы, от RTL до схемы как реального устройства

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

Фронты сигнала, *передние* (↑) и *задние* (↓), определяются так:[1]

до \ после	0	x	1
0		↑[2]	↑
x	↓[2]		↑[2]
1	↓	↓[2]	

[1] Фронты, связанные со значением `z`, здесь не обсуждаются

[2] При разработке схемы следует по возможности избегать фронтов, связанных со значением `x`

$\mathcal{V}$: основные типы данных

Категория типа 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 триггеров и регистров

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] Для значения z всё немного сложнее, но не будем в это углубляться

$\mathcal{V}$: 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

Значением шины ширины n является **набор** из n цифровых сигналов

Кроме того, в описании преобразования значений сигналов используются две арифметическая трактовки значения шины в заданный момент времени:

1. Набор значений $(\alpha_{n-1} \dots \alpha_0)$ беззнаковой шины соответствует числу $(\alpha_{n-1} \dots \alpha_0)_2 = \sum_{i=0}^{n-1} 2^i \cdot \alpha_i$
2. Набор значений $(\alpha_{n-1} \dots \alpha_0)$ знаковой шины соответствует числу $(\alpha_{n-1} \dots \alpha_0)_2^- = (\overline{\alpha_{n-1}}\alpha_{n-2} \dots \alpha_0)_2 - 2^{n-1}$ (*дополнительный код*)

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 того же типа 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 рассмотрим два вида портов:

- Входы — обозначаются словом `input`
- Выходы — обозначаются словом `output`

Порты в $\mathcal{V}$ имеют тот же смысл, что и в последовательных схемах: это точки схемы, через которые

- ей посылаются сигналы из окружения (входы) и
- из неё сигналы отправляются в окружение (выходы)

$\mathcal{V}$: модули

Модуль в $\mathcal{V}$ — это

- описание подсхемы
- 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, содержащая порты (входы, выходы, ...), задаваемые 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

По умолчанию считается, что тип каждого идентификатора — `wire`:

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

```
module M(a, b, u, v);
  input 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)

Поведение любой комбинационной схемы можно представить так:

в каждый момент времени значения на выходах заданным образом соответствуют значениям на входах в тот же момент

Такое соответствие задаётся непрерывным присваиванием:

assign x = E;

- x — соединение, **не** являющееся входом
- E — комбинационное выражение: выражение, составленное из точек, констант и специальных (комбинационных) операций
- значение в точке x в каждый момент времени равно значению E в тот же момент времени

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;
```

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

Для удобства проектирования схем в семантику $\mathcal{V}$ включены механизмы выравнививания ширины шин

Если шине $\mathtt{x}$ ширины N присваивается значение шины $\mathtt{y}$ ширины M и при этом:

- $N < M$,
  то $(M - N)$ старших битов шины $\mathtt{y}$ игнорируются в присваивании
- $N > M$, то шине $\mathtt{x}$ присваивается значение $\mathtt{y}$, расширенное требуемым числом старших разрядов:
  - если значение $\mathtt{vy}$ старшего разряда $\mathtt{y}$ (знака) — 0, $\mathtt{x}$ или $\mathtt{z}$, то значения добавляемых разрядов равны $\mathtt{vy}$
  - иначе ($\mathtt{vy} = 1$ и) значения всех добавляемых разрядов —
    - 0, если шина $\mathtt{y}$ беззнаковая, и
    - 1, если знаковая

$\mathcal{V}$: 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>`

Константа вида "`<значение>`" трактуется как знаковая, остальные — как беззнаковые

$\mathcal{V}$: 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	x	1
0	0	0	0
x	0	x	x
1	0	x	1

`x || y`

x \ y	0	x	1
0	0	x	1
x	x	x	1
1	1	1	1

`!x`

a	! a
0	1
x	x
1	0

При вычислении результата ширина аргументов сужается до 1

$\mathcal{V}$: комбинационные операции

Арифметические операции

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

Программная семантика:

Знаковость выражения: если хотя бы один аргумент беззнаковый, то выражение беззнаковое, иначе выражение знаковое

- будем называть такой вид знаковости стандартным

Ширина выражения — максимум ширины аргументов и ширины

- левой части присваивания, если операция является внешней
- "объемлющего" выражения, если эта ширина задана

Ширина и знаковость результата равны ширине и знаковости выражения

Знаковость аргументов приравнивается знаковости выражения, и аргументы расширяются до ширины выражения

Результат — естественный для *арифметики с переполнением*

- если хотя бы одно из чисел не определено, то результат — расширенный `1'bx`

$\mathcal{V}$: комбинационные операции

Арифметические отношения:

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

Программная семантика:

Знаковость выражения: стандартная

Знаковость аргументов приравнивается знаковости выражения, и аргумент меньшей ширины расширяется до аргумента большей ширины

Результат — беззнаковая шина ширины 1, значение которой — 1, если аргументы входят в отношение, и 0, если не входят

- если хотя бы одно из чисел не определено, то результат — `1'bx`

$\mathcal{V}$: 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 "<<<": результат — сдвиг `x` влево на `y` разрядов с заполнением нолями

">>", а также ">>>" для беззнакового `x`: результат — сдвиг `x` вправо на `y` разрядов с заполнением нолями (*логический сдвиг*)

">>>" для знакового `x`: результат — сдвиг `x` вправо на `y` разрядов с заполнением знаком (*арифметический сдвиг*)

Если число `y` не определено, то результат — расширенный `1'bx`

$\mathcal{V}$: комбинационные операции

Многобитовые логические операции:

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

Программная семантика:

Аргумент меньшей ширины расширяется до аргумента большей ширины **с заполнением нолями**

Результат — шина той же ширины, каждый бит которой — результат применения соответствующей операции (&&, ||, !=, ==) к соответствующим битам аргументов

Знаковость результата: стандартная

$\mathcal{V}$: комбинационные операции

Операции редукции:

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

Программная семантика:

Результат вычисления каждой операции — беззнаковая шина ширины 1

Результат для операции без "~": соответствующая логическая операция (`&&`, `||`, `!=`) применяется к паре младших разрядов, и затем, итеративно до конца шины, к результату предыдущего шага и следующему разряду

Результат для операции с "~": результат —
отрицание (`!`) результата соответствующей операции без "~"

$\mathcal{V}$: комбинационные операции

Тернарный оператор:

```
x ? y : z
```

Программная семантика:

Знаковость результата: стандартная для аргументов `y`, `z`

Ширина результата: максимум ширин `y`, `z`

Наиболее узкий из аргументов `y`, `z` расширяется до наиболее широкого **с заполнением нолями**

Значение результата:

- если значение "`x == 0`" — 1, то совпадает с `z`
- если значение "`x == 0`" — 0, то совпадает с `y`
- иначе каждый разряд `b` результата определяется так:
  - если соответствующие разряды `y` и `z` равны, то `b` равен этим разрядам
  - иначе значение `b` — `x`

$\mathcal{V}$: комбинационные операции

Операции индексации

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

Программная семантика: если `x` — шина типа `type [msb:lsb]`, то

- результат для "`x[i]`" — беззнаковое значение типа `type`, равное `i`-му разряду шины `x`
- результат для "`x[i:j]`" — шина (x[i] ... x[j]) элементов типа `type` ширины `(i-j+1)`

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

$\mathcal{V}$: комбинационные операции; знаковые точки

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`)

Объявления знаковых точек

Чтобы точка по умолчанию трактовалась как знаковая, достаточно добавить слово `signed` в подходящее место её объявления:

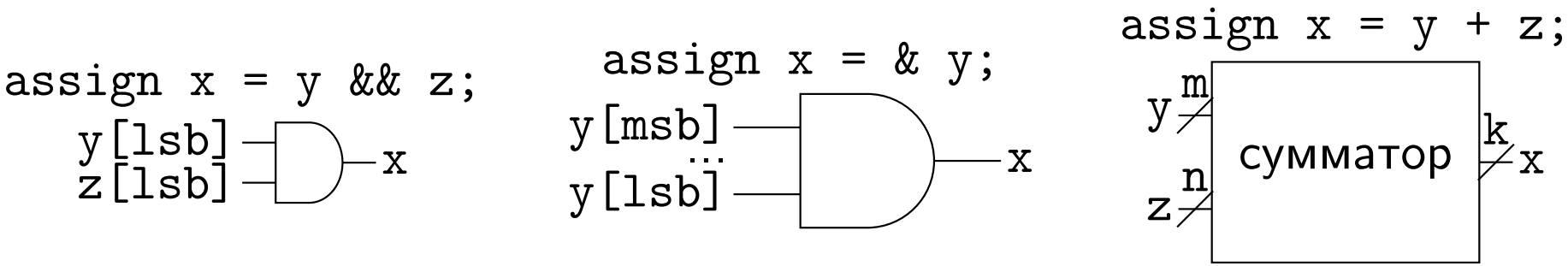
```
input signed x;
wire signed y;
reg signed [1:0] z;
```

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

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

- Входы Σ — шины всех переменных, используемых в `E`
- Выход Σ — шина `x`
- Если для булевых значений $(\alpha_1, \ldots, \alpha_n)$, сгруппированных по шинам и подставленных на места переменных `E`, результат вычисления выражения — $(b_{k-1} \ldots b_0)$ и $b_i \in \{0, 1\}$, то при посылке $(\alpha_1, \ldots, \alpha_n)$ на соответствующие входы Σ на выходе `x[i]` обязательно получается значение b_i

Примеры:



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

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

Программная семантика:

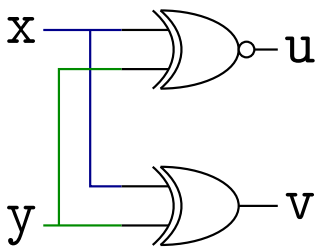
u

x \ y	0	x	1
0	1	x	0
x	x	x	x
1	0	x	1

v

x \ y	0	x	1
0	0	x	1
x	x	x	x
1	1	x	0

Аппаратная семантика:



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

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

Объявление про́вода и непрерывное присваивание 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:

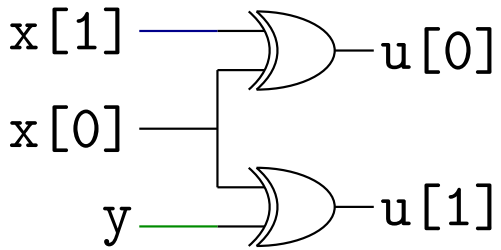
u[0]

x[0] \ x[1]	0	x	1
0	0	x	1
x	x	x	x
1	1	x	0

u[1]

x[0] \ y	0	x	1
0	0	x	1
x	x	x	x
1	1	x	0

Аппаратная семантика:



$\mathcal{V}$: 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 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] *Напоминание*: все точки по умолчанию имеют тип `wire` — в том числе `c1` и `c2`