Основы проектирования аппаратных ускорителей систем искусственного интеллекта

Лекция 3 – Введение в Verilog

План лекции

- Языки описания схем (hardware description languages)
- Введение в язык Verilog
- Основы тестирования и симуляции схем

Языки описания аппаратного обеспечения

- Hardware Description Language (HDL) – язык описания структуры и функционирования аппаратного обеспечения цифровой системы на различных уровнях абстракции.
- Основные HDL:
  - Verilog
  - SystemVerilog
  - VHDL
  - SystemC
  - …
- http://en.wikipedia.org/wiki/Hardware_description_language

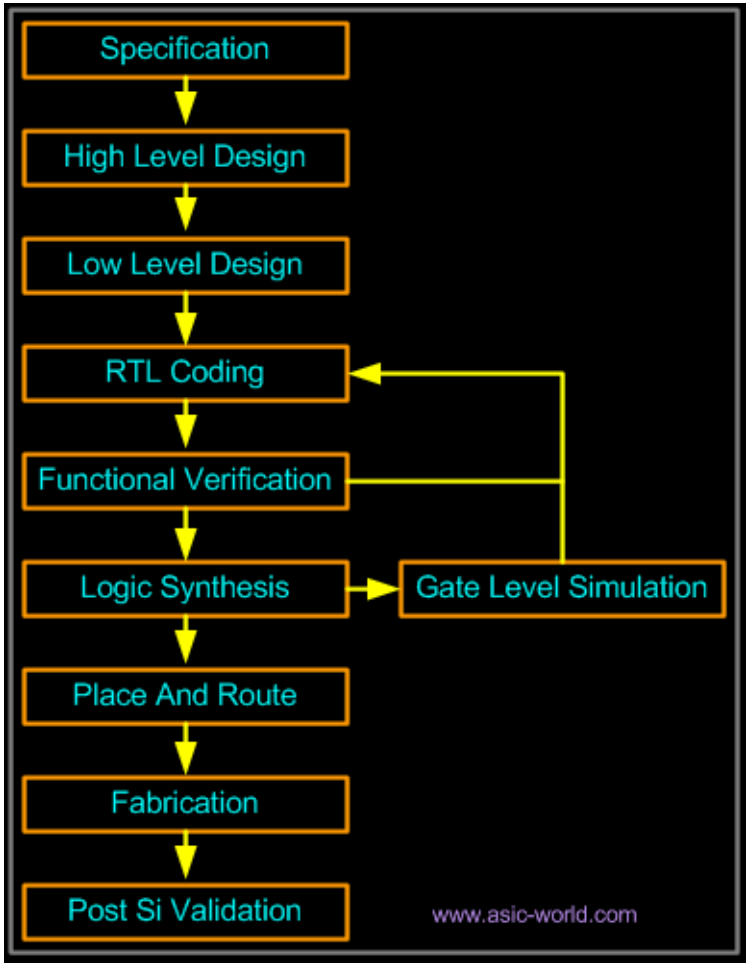
Стили разработки аппаратного обеспечения

- Снизу-вверх
  - Сначала отдельно проектируются базовые элементы, а потом из простых элементов проектируются (собираются) более сложные элементы.
- Сверху-вниз
  - Сначала проектируются системы более высокого уровня, а потом рекурсивно проектируются их компоненты более низкого уровня.

Основные уровни абстракции аппаратного обеспечения

- Поведенческий уровень (Behavioral Level)
  - Описание устройства при помощи набора последовательных алгоритмов, работающих одновременно и согласованно.
- Уровень регистровых передач (Register-Transfer Level)
  - Описание устройства при помощи заданного набора операций преобразования значений, хранящихся в регистрах. По сути устройство описывается в виде иерархии взаимосвязанных автоматов.
- Схемный (функциональный) уровень (Gate Level)
  - Описание устройства при помощи схемы из функциональных элементов в заданной библиотеке элементов.

Упрощенный маршрут проектирования



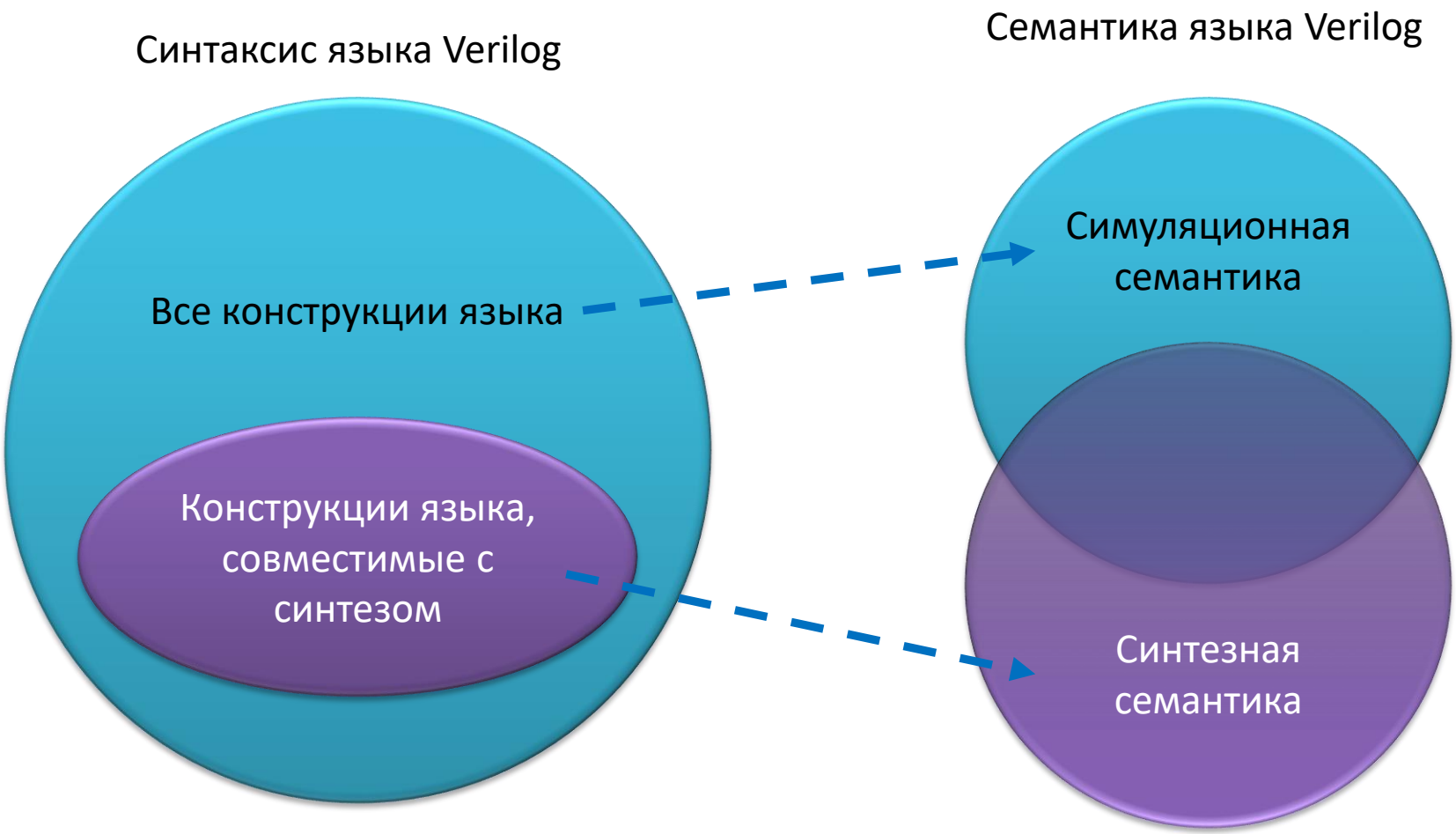
Язык Verilog

- Модули и шины
- Структурное описание модулей
- Функциональное описание модулей
- Блокирующее и неблокирующее присваивание

Язык Verilog

- Verilog – один из самых популярных на данный момент языков описания цифровых интегральных схем
- Изначально этот язык создавался для **моделирования работы** интегральной схемы (**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 использоваться и для **синтеза** реальных схем

Структура языка Verilog



Аналогии с языками программирования

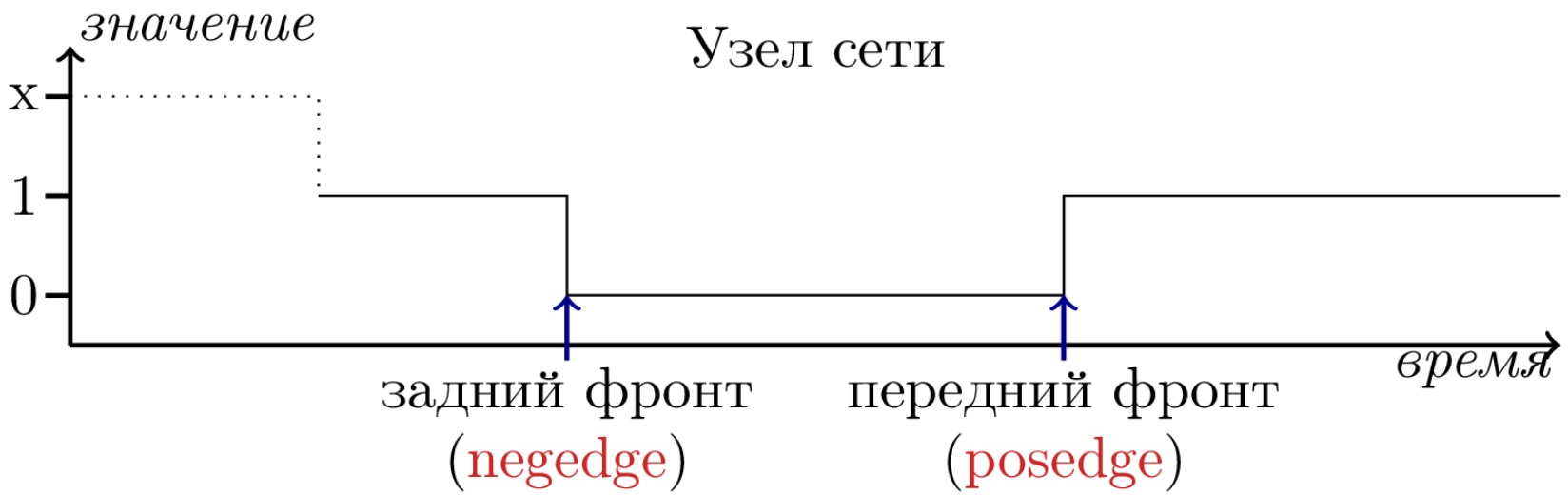
- Некоторые конструкции в синтаксисе языка Verilog очень похожи на конструкции языка C/C++
- Это сходство является поверхностным
- Некоторые детали семантики схожих конструкций этих языков совпадают, но на этом сходство заканчивается:
  - Программа, написанная на языке C/C++, транслируется в машинный код, последовательно выполняемый процессором
  - Схема, спроектированная на языке Verilog, транслируется в
    - Программу, моделирующую изменение сигналов во времени, если используется симуляционная семантика
    - Последовательную схему и затем в реальную цифровую схему, если используется синтезная семантика
- Аналогии с конструкциями языка C/C++ будут отмечены специальным цветом

Логические значения

- 0 — логический ноль
- 1 — логическая единица
- x — неопределённое значение
  - в реальной схеме x означает «0 или 1, а что именно, неизвестно/неважно»
- z — состояние высокого импеданса
  - в симуляционной семантике z j\очень похоже на x
  - в реальной схеме z означает, что мы не можем управлять потенциалом в заданном узле
  - в ближайшее время это значение нам не понадобится

Логические значения

- **Что мы строим:** схему, между узлами которой в реальном времени передаются логические значения



- 0 и 1 — это конкретные **уровни напряжения**
- x — это **абстракция**: уровень напряжения соответствующий неизвестному логическому значению

Основные типы данных

- Типы данных языка Verilog делятся на две основных 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/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 специального типа, то провода, переменные и другие именованные объекты будем называть **узлами** (в схеме)

Основные типы данных

- С точки зрения симуляционной семантики, типы `wire` и `reg` различаются только тем, в каких конструкциях разрешено использовать узлы этих типов:
  - `reg` – при задании узлов, «имеющих память», то есть способных сохранять значение и непрерывно выдавать последнее сохранённое значение
  - `wire` – при задании узлов, «не имеющих память», то есть непрерывно выдающих результат применения булевой функции к аргументам – значениям в других узлах

Основные типы данных

- С точки зрения синтезной семантики:
  - `wire` – это провод, соединяющий элементы схемы (логические вентили и ячейки памяти)
  - `reg` – это либо элемент памяти (регистр, защелка, триггер и т.д.), либо провод, в зависимости от функционала спроектированной схемы
- Хотя «`reg`» расшифровывается как «`register`», следует иметь в виду, что регистры в последовательных схемах и переменные типа `reg` – это не одно и то же

Основные типы данных - шины

- Можно объявлять не только одиночные провода и переменных, но и шины (в стандарте языка - векторы):

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

  - `type` – это тип (например, wire или reg)
  - `x` – это имя объявляемой шины
  - `msb` и `lsb` – целые числа, индекс старшего и младшего битов
- Например,
  - `wire [5:0] x;` – объявление шины `x` ширины 6, биты которой нумеруются от 0 до 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 отрицательными)

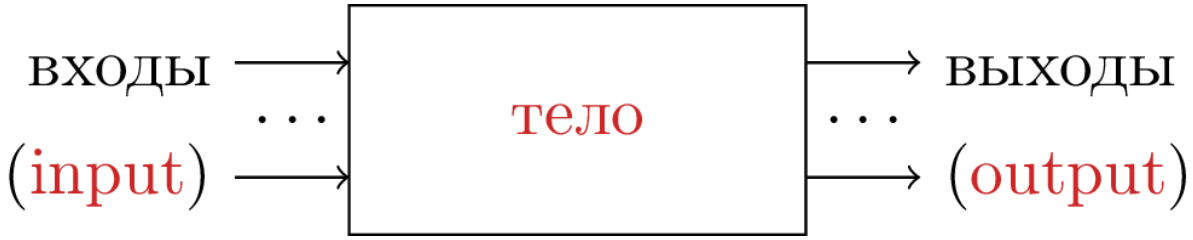
Основные типы данных - шины

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

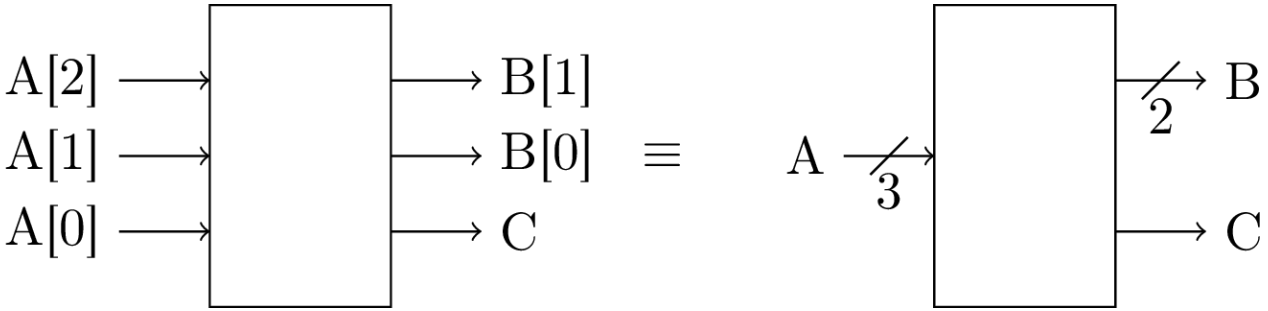
- Совокупность значений шины трактуется по умолчанию как без знаковое целое число:
  - Если `(x[5] x[4] x[3] x[2] x[1] x[0]) = (010111)`, то в шине `x` хранится число 23
  - Если `(z[-1] z[0] z[1]) = (110)`, то в шине `z` хранится число 6
- Язык Verilog содержит немало арифметических операций над шинами, 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 числа – старший бит)

Модули

- «Строительный блок» языка Verilog – модуль



- Входы и выходы могут быть проводами и переменными (`wire` и `reg`), а также могут быть объявлены как шины (`[7:0] bus`)



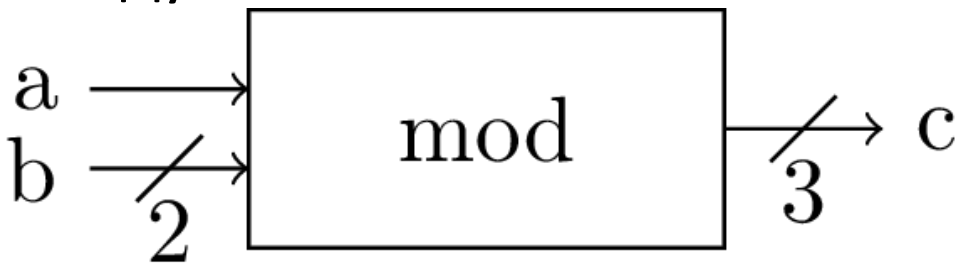
Модули

- Модуль – это описание некоторой схемы, которая имеет заданный функционал, то есть определенным образом преобразует входные сигналы в выходные
- Понятие модуля наиболее близко к понятию «класса» и иногда «функции» и «функтора» в языке C/C++
- Модуль состоит из имени, портов и тела
  - Тело модуля – это описание того, как функционирует модуль
  - Порт – это имя, обозначающий узел, значение которого доступно извне модуля (аргумент функции)
- Для нас достаточно рассмотреть два вида портов:

входы (`input`) и выходы (`output`)

Определение модуля

- Лучше всего описывать модуль в отдельном файле с расширением .v и названием, совпадающим с названием модуля

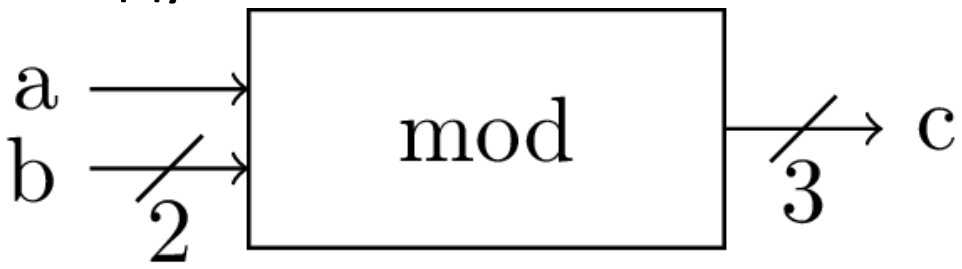


Файл `mod.v`: (первый вариант)

```
module mod(a, b, c);
  input wire a;
  input wire [1:0] b;
  output reg [2:0] c;

  // description
endmodule
// EMPTY LINE!
```

Определение модуля

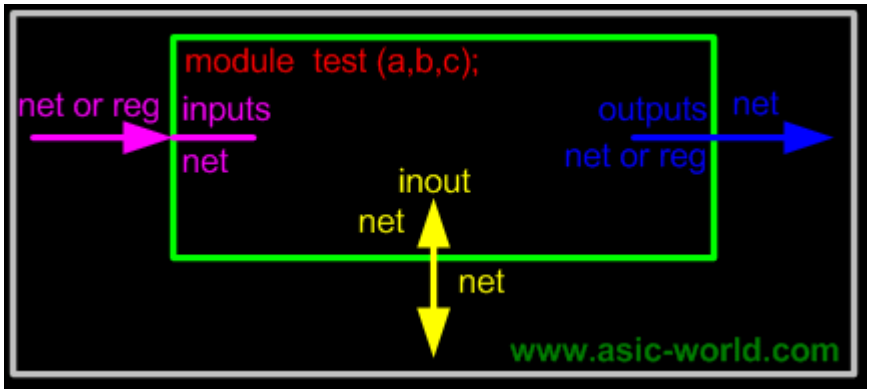
- Лучше всего описывать модуль в отдельном файле с расширением .v и названием, совпадающим с названием модуля



Файл `mod.v`: (второй вариант)

```
module mod(
  input wire a,
  input wire [1:0] b,
  output reg [2:0] c
);
  // description
endmodule
// EMPTY LINE!
```

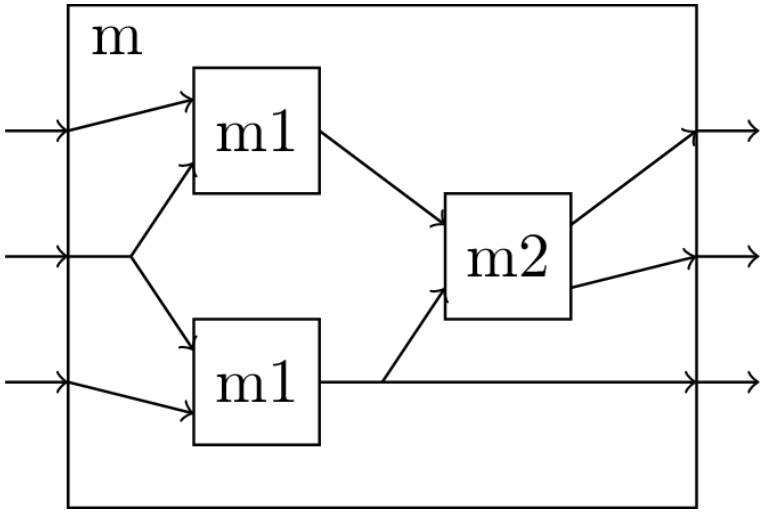
Правила соединения портов



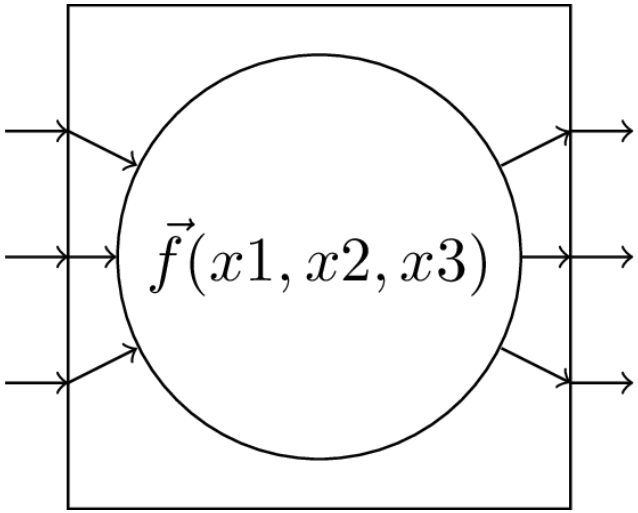
- Входы блока: внутри блока входы могут быть только проводами (`wire`), а вне блока могут быть как проводами, так и переменными(`reg`)
- Выходы блока: внутри блока выходы могут быть как проводами, так и переменными, а вне блока только проводами
- По умолчанию все порты модуля имеют тип `wire`

Способы описания модуля

- Обычно различают два подхода к описанию модуля:
  - структурный: явно описать экземпляры (instances) модулей и связи между ними
  - функциональный: без явного описания структуры задать поведение (реализуемые функции) модулей и связи между ними

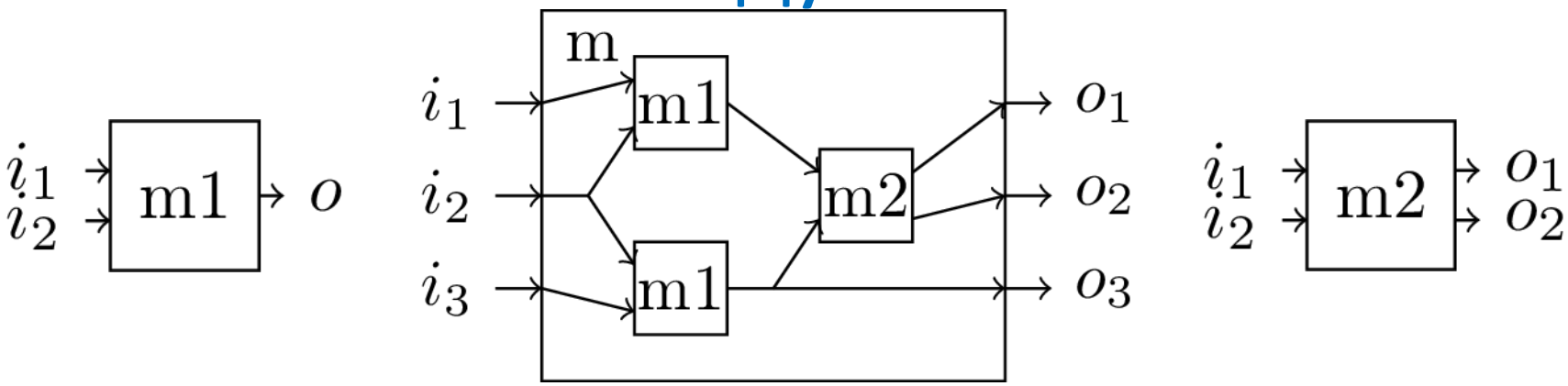


Структурное описание



Функциональное описание

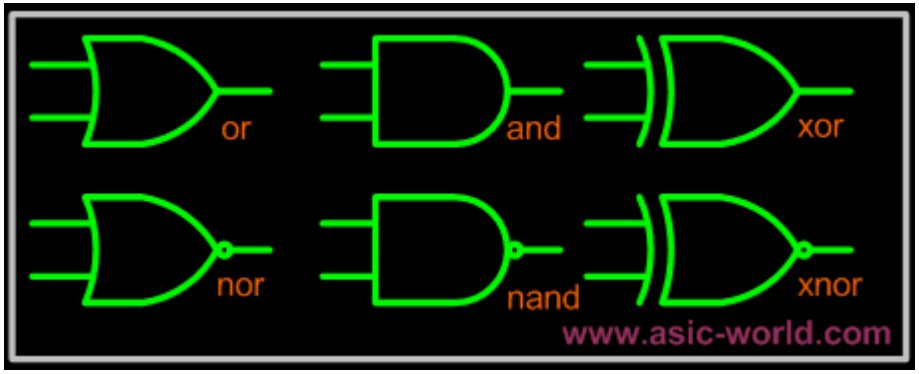
Структурное описание – экземпляры модулей




```
module m(input i1, i2, i3, output o1, o2, o3);
  wire w;
  m1 upleft(.i1(i1), .i2(i2), .o(w));
  m1 downleft(.i1(i2), .i2(i3), .o(o3));
  m2 right(.i1(w), .i2(o3), .o1(o1), .o2(o2));
endmodule
```

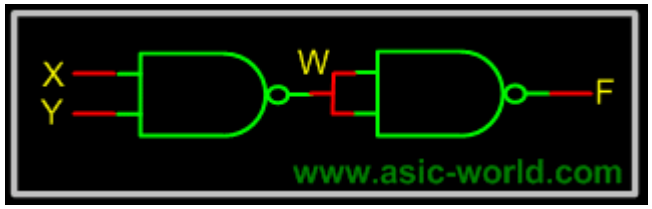
Все встречающиеся в описании имена (в том числе провода: wire) должны быть определены перед первым использованием

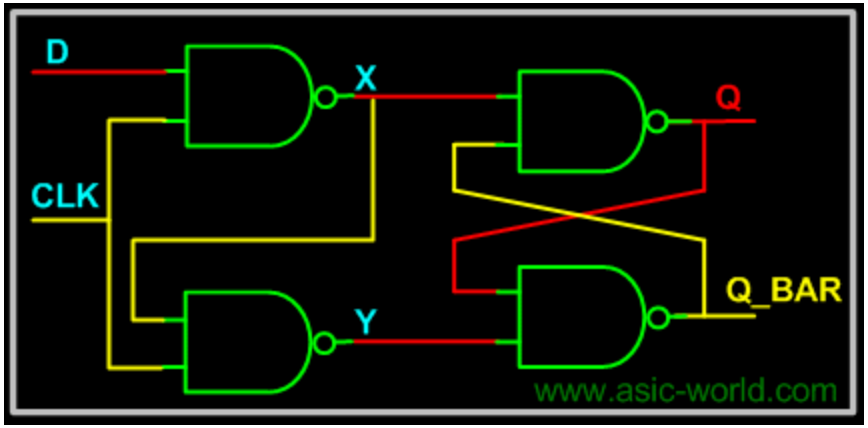
Структурное описание – примитивы




```
module m(…);
  …
  not(o1,i1);
  or(o2,i1,…,in);
  and(o3,i1,…,in);
  …
endmodule
```

Структурное описание – примеры

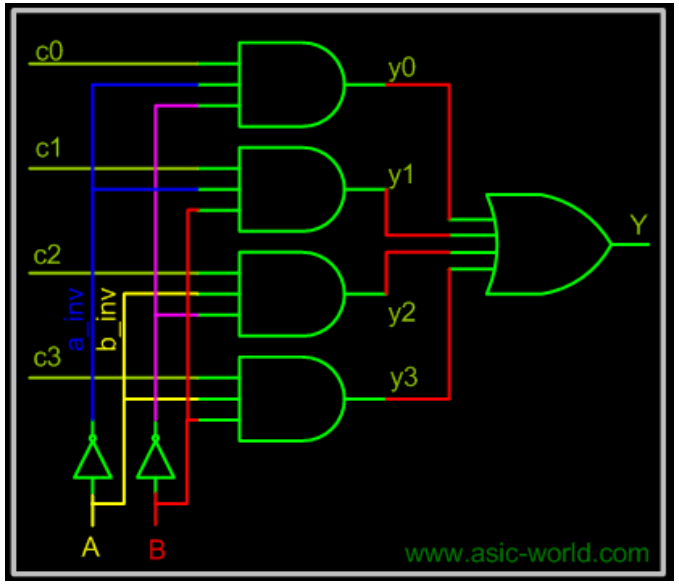





```
module and_from_nand();
  reg  X, Y;
  wire F, W;

  nand U1(W, X, Y);
  nand U2(F, W, W);

endmodule
```

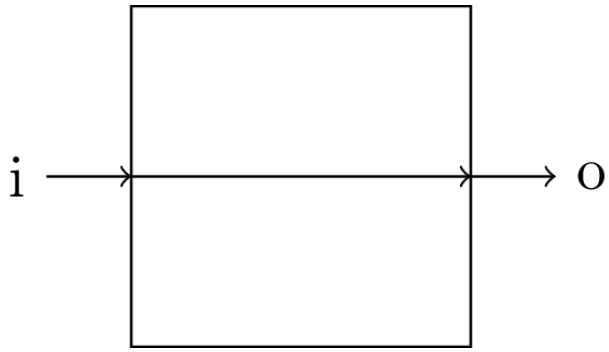
```
module dff_from_nandm();
  wire Q, Q_bar;
  reg  D, CLK;

  nand U1(X, D, CLK);
  nand U2(Y, X, CLK);
  nand U3(Q, Q_bar, X);
  nand U4(Q_bar, Q, Y);

endmodule
```

Структурное описание – примеры




```
module mux_from_gates();
  reg A, B;
  reg c0, c1, c2, c3;

  wire Y;
  wire y0, y1, y2, y3;
  wire a_inv, b_inv;

  not(a_inv, A);
  not(b_inv, B);

  and(y0, c0, a_inv, b_inv);
  and(y1, c1, a_inv, B);
  and(y2, c2, A, b_inv);
  and(y3, c3, A, B);

  or(Y, y0, y1, y2, y3);
endmodule
```

Непрерывное присваивание




```
module trivial(input i, output o);
  assign o = i;
endmodule

assign «провод» = «выражение»;
```

В любой момент времени (с некоторой задержкой при изменении значения) на проводе должно быть значение выражения

Выражения

Что можно использовать при написании выражений:

- логические операции
  - например, a && b — логическое И
- арифметические операции
  - например, a + b — это сложение двух чисел одинаковой битности с переполнением
- побитовые операции
  - например, a & b — это побитовое И двух битовых массивов одинаковой длины
- отношения
  - например, a < b возвращает логическую 1, если число, двоичная запись которого есть a, меньше такового для b, и логический 0 иначе

Выражения

Что можно использовать при написании выражений:

- Конкатенации
  - например, {a, b} — битовый массив, составленный из a и b
- редукции
  - например, &a — логическая 1, если все биты a — единицы, и логический 0 иначе
- условия
  - например, cond ? a : b работает как в C++; cond должно иметь логическое значение, а a и b должны иметь одинаковое число бит
- константы
  - например, 0 — это логический ноль, а 5’b00110 — пятибитная двоичная запись числа 6

(полный список операций можно найти в интернете)

Операторы Verilog и их приоритеты

Verilog Operator	Name	Functional Group
[]	bit-select or part-select	
()	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift
>= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

Функциональное описание always-блок

- Он выглядит так:

```
always @(a or posedge b or negedge c)
  // statement
```

- В аргументе перечисляются места (например, провода), при изменении сигнала в которых должно производиться какое-то действие
- В данном случае:
  - при изменении логического значения в a,
  - а также когда в b возникает передний фронт,
  - а также когда в c возникает задний фронт
- Действие перезаписывает значения сигналов модуля
- После выполнения действия получившиеся значения сохраняются в проводах до следующего выполнения блока

Функциональное описание always-блок

- Он выглядит так:

```
always @(a or posedge b or negedge c)
  begin
    //sequence of statements
  end
```

- Действий можно задавать много, и тогда их обычным образом нужно соединить в составное действие

Функциональное описание always-блок

- Он выглядит так:

```
always @(a, posedge b, negedge c)
  begin
    //sequence of statements
  end
```

- В какой-то момент разработчики стандарта Verilog поняли, что “or” писать неудобно, так что разрешили вместо него ставить запятую
- Какие же действия можно писать в always-блоке?

Блокирующие присваивания

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

- Последовательно делается следующее:
  - в b выставляется начальное значение из c
  - в a выставляется изменённое значение из b
  - в c выставляется изменённое значение из a
- Блокирующее присваивание моделирует последовательное выполнение команд: пока присваивание не выполнено, следующие команды не выполняются (но в конечном итоге строится схема, просто она имеет хитрую структуру с блоками памяти)

Неблокирующие присваивания

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

- Одновременно делается следующее:
  - в b выставляется начальное значение из c
  - в a выставляется начальное значение из b
  - в c выставляется начальное значение из a
- Вообще говоря, одновременности не бывает, но в реальной схеме эти действия будут выполнены близко по времени, и блоки памяти будут организованы так, чтобы выставлялись именно начальные значения

Функциональное описание – управляющие конструкции

```
if(cond) stmt;
else stmt;

case(a)
  3'b000: stmt;
  3'b010: stmt;
  3'b011: stmt;
  default: stmt;
endcase
```

- Условные инструкции тоже можно писать
- Как и инструкцию switch-case
- Они интерпретируются обычным образом (примерно как в C++)

Функциональное описание – примеры

```
module register3(
  input load, reset, clock,
  input [2:0] in,
  output reg [2:0] out
  );

  always @(posedge clock, negedge reset)
    if(~reset) out <= 0;
    else if(~load) out <= in;
endmodule
```

Функциональное описание – примеры

```
module adder(
  input  [7:0] a, b,
  output [7:0] sum
  );

  assign sum = a + b;
endmodule
```

Функциональное описание initial-блок

- Он выглядит так:

```
initial
  begin
    //sequence of statements
  end
```

- В основном используется симулятором, но может быть использован синтезатором для определения начальных значений различных элементов проектируемого устройства
- Выполняет заданный набор в начальный момент времени.
- Выполняется один раз

Структура «программы» на языке Verilog

- «Программа» на языке Verilog представляет собой набор модулей, которые могут описывать различные элементы проектируемой системы от простых функциональных элементов до всей системы целиком
- Модуль определяет свои порты ввода и вывода, которые описывают его входные и выходные соединения (сигналы)
- В модуле могут быть определены дополнительные переменные
- Тело модуля может состоять из следующих элементов:
  - Выражения initial, которые могут инициализировать полюса типа reg
  - Выражения непрерывного присваивания, которые определяют элементы комбинационной логики
  - Выражения always, которые определяют элементы комбинационной и последовательной логики
  - Экземпляры других модулей, которые реализуют определенные ранее модули

Важные замечания

- Модуль языка Verilog НЕ является функцией, как в языках программирования. Модуль можно считать специфическим видом класса.
  - Функция – это фрагмент кода, который можно вызвать
  - Модуль определяет сущность с заданной функциональностью. Его **можно использовать**, но **нельзя вызвать**.
  - Каждый раз когда создается экземпляр модуля, ему в соответствие будет ставиться набор реальных логических элементов, которые будут его реализовывать в интегральной схеме.
- Verilog является языком описания схем, а не языком программирования
  - В итоге мы **описываем** то, что хотим спроектировать

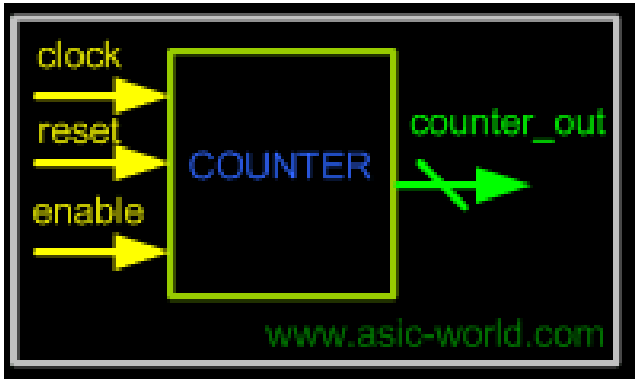
Главный модуль

- Так как при описании 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 является функция с именем `main`.
- Язык Verilog в этом плане более гибкий:
  - главный модуль (top-level entity) – это модуль, экземпляры которого не используются в телах других модулей рассматриваемой совокупности
  - главный модулей может быть несколько, если не требуется синтез одной конкретной схемы
- Никаких специальный требований к виду главного модуля не предъявляется:
  - главный модуль – это такой же модуль, как и все остальные

Язык Verilog – Hello world!

```
1  module hello_world;
2    initial begin
3      $display ("Hello world!");
4      #10 $finish;
5    end
6  endmodule
```

Пример моделирования на языке Verilog – 4-х битовый счетчик



- Входы
  - Clock – вход тактового генератора
  - Reset – сброс значения счетчика (сброс при «1»)
  - Enable – включение счетчика (при «1» счетчик работает)
- Выходы
  - Counter_out – значение счетчика
- Функция
  - Счетчик увеличивается на 1-цу за каждый «такт» тактового генератора

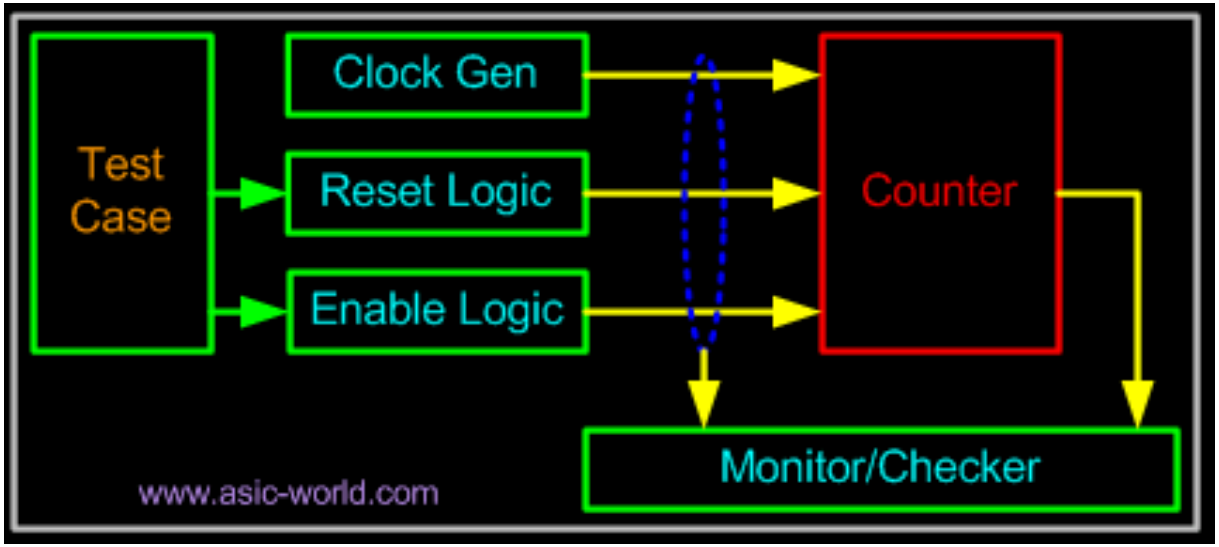
Пример моделирования на языке Verilog – 4-х битовый счетчик

```
1  module counter (clock, reset, enable, counter_out);
2
3    input clock;
4    input reset;
5    input enable;
6
7    output [3:0] counter_out;
8
9    wire clock;
10   wire reset;
11   wire enable;
12
13   reg [3:0] counter_out;
14
```

Пример моделирования на языке Verilog – 4-х битовый счетчик

```
15    always @ (posedge clock)
16      begin: COUNTER
17        if (reset == 1'b1) begin
18          counter_out <= #1 4'b0000;
19        end
20        else if (enable == 1'b1) begin
21          counter_out <= #1 counter_out + 1;
22        end
23      end
24  endmodule
```

Пример моделирования на языке Verilog – тестирование



Пример моделирования на языке Verilog – тестирование

```
1  `include "counter.sv"
2
3  module counter_tb();
4    reg clock, reset, enable;
5    wire [3:0] counter_out;
```

Пример моделирования на языке Verilog – тестирование

```
7   initial begin
8     $dumpfile("dump.vcd");
9     $dumpvars(1);
10
11    $display ("time\t clk\t reset\t enable\t counter");
12    $monitor ("%g\t %b\t %b\t %b\t %b",
13              $time,
14              clock,
15              reset,
16              enable,
17              counter_out
18              );
19
```

Пример моделирования на языке Verilog – тестирование

```
20      clock = 1;
21      reset = 0;
22      enable = 0;
23      #5 reset = 1;
24      #10 reset = 0;
25      #10 enable = 1;
26      #100 enable = 0;
27      #5 $finish;
28    end
29
```

Пример моделирования на языке Verilog – тестирование

```
30    always begin
31      #5 clock =~ clock;
32    end
33
34    counter test_counter (
35      clock,
36      reset,
37      enable,
38      counter_out
39    );
40
41  endmodule
```