Лекция 2

Программируемые логические интегральные схемы

Осень 2016

План лекции

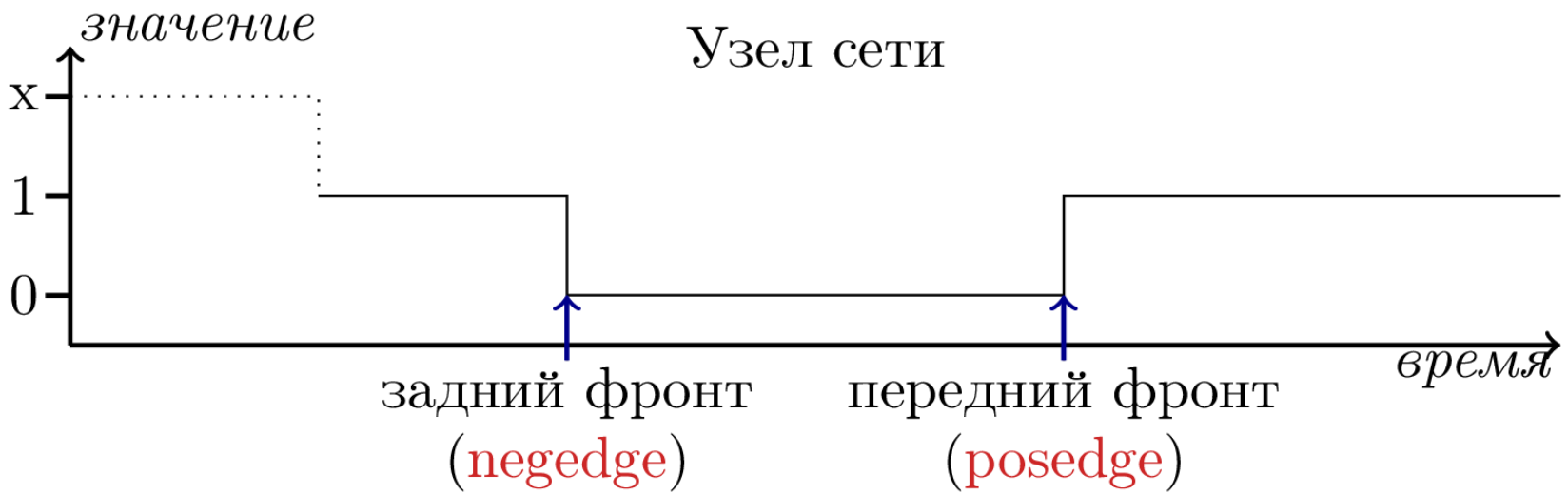
- Модули и шины
- Структурное описание модулей
- Функциональное описание модулей
- Блокирующее и неблокирующее присваивание
- Параметризованные модули

Логические значения

- 0 — логический ноль
- 1 — логическая единица
- x — неопределённое значение
  - например, в неинициализированном регистре
- z — состояние высокого импеданса
  - оно нужно в основном для управления шиной, к которой подключено много независимых устройств
  - в ближайшее время это значение нам не понадобится

Логические значения

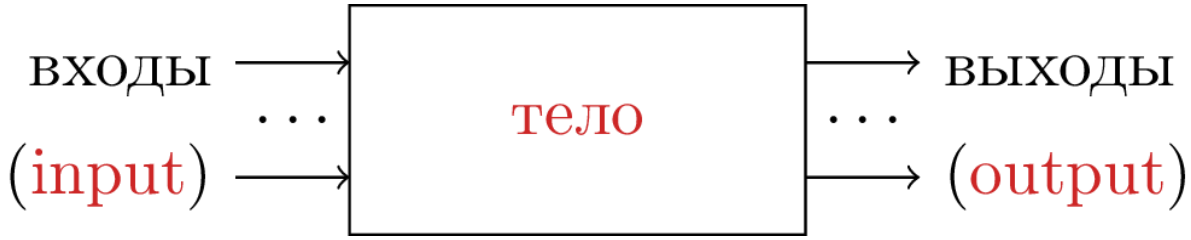
- **Что мы строим:** схему, между узлами которой в реальном времени передаются логические значения



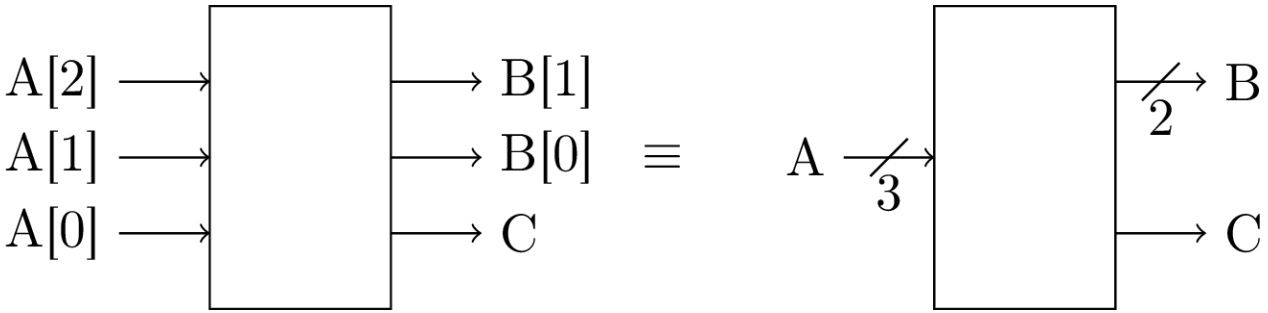
- 0 и 1 — это конкретные **уровни напряжения**
- x — это **абстракция**: уровень напряжения соответствующий неизвестному логическому значению

Verilog: модули и шины

- «Строительный блок» языка Verilog – модуль

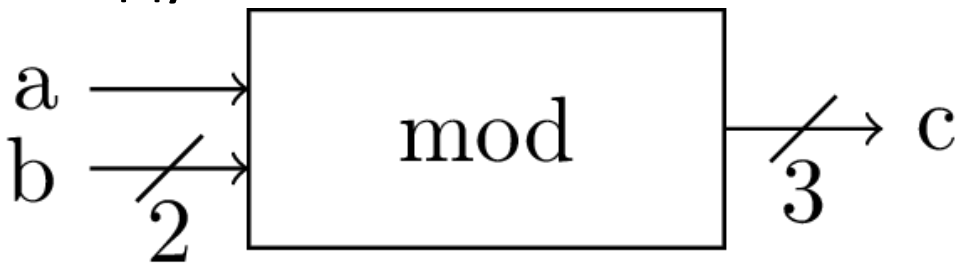


- Входы и выходы могут быть проводами и регистрами (wire и reg), а также могут быть объявлены как массивы ([7:0] bus, {a,b,c})
- Массив проводов – это шина:



Verilog: определение модуля

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

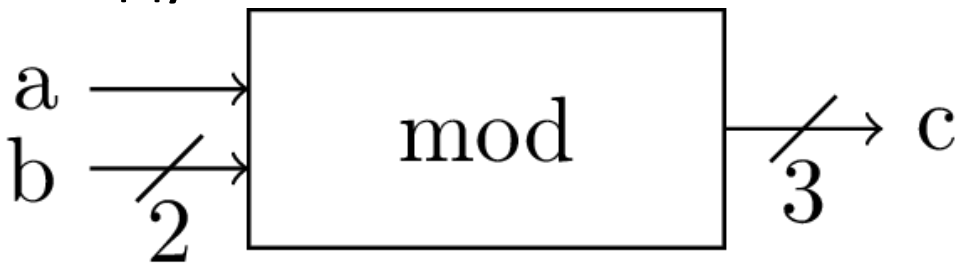


Файл `mod.v`: (первый вариант)

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

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

Verilog: определение модуля

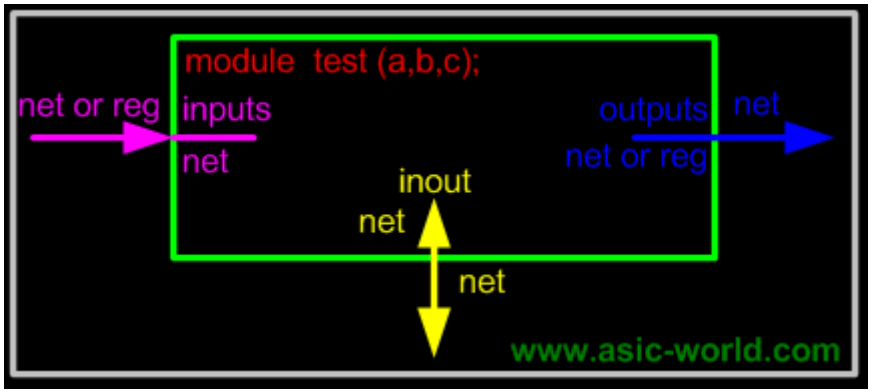
- Лучше всего описывать модуль в отдельном файле с расширением .v и названием, совпадающим с названием модуля



Файл `mod.v`: (второй вариант)

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

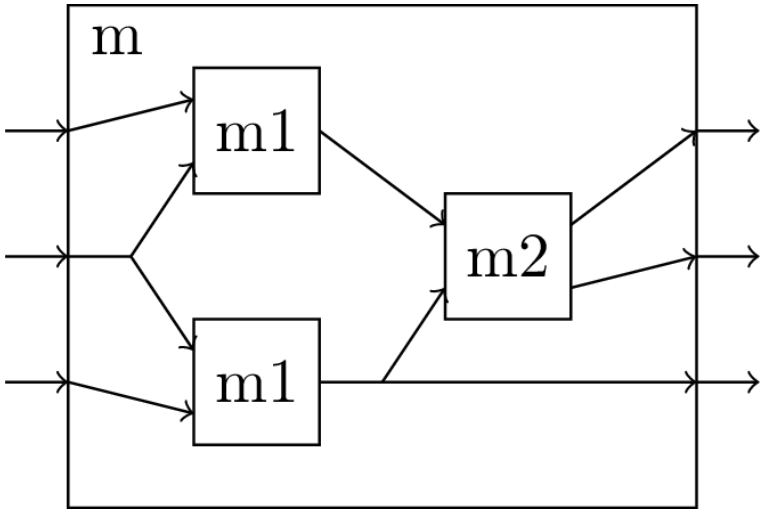
Verilog: правила соединения портов



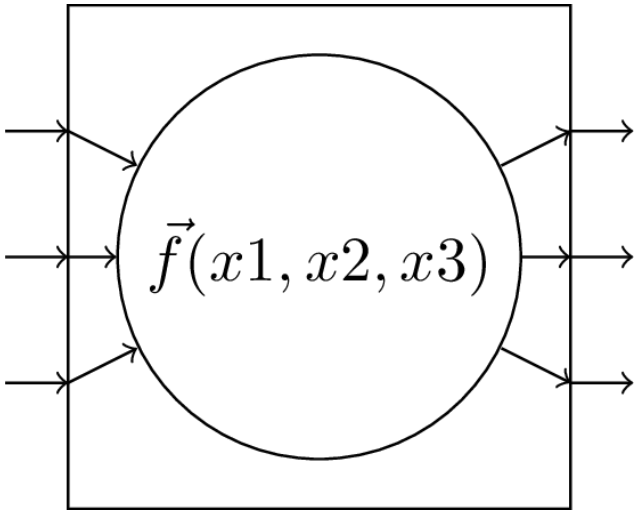
- Входы блока: внутри блока входы могут быть только проводами (wire), а вне блока могут быть как проводами, так и регистрами(reg)
- Выходы блока: внутри блока выходы могут быть как проводами, так и регистрами, а вне блока только проводами
- Смешанные порты: внутри и вне блока могут быть только проводами

Verilog: 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

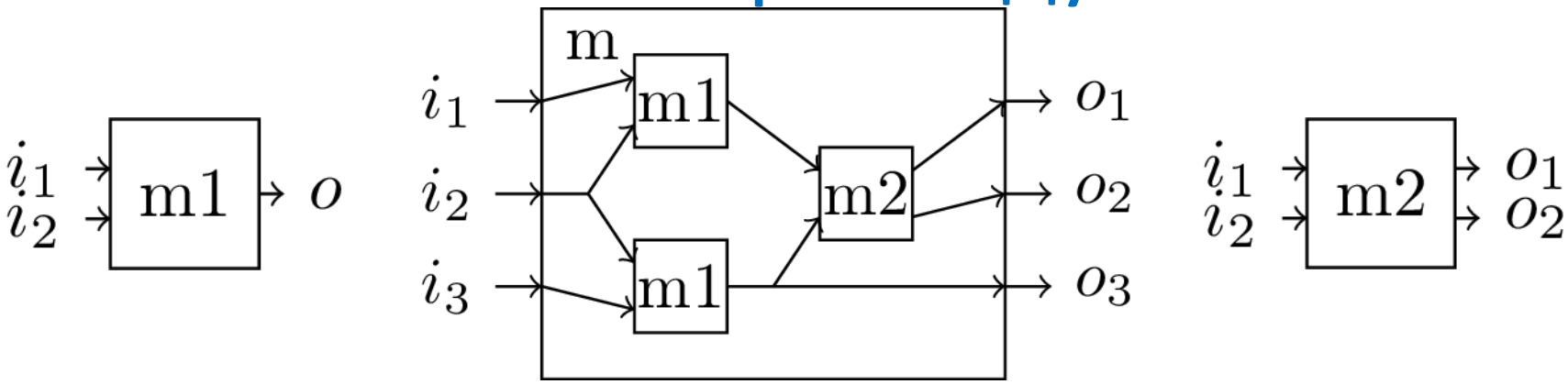


Структурное описание



Функциональное описание

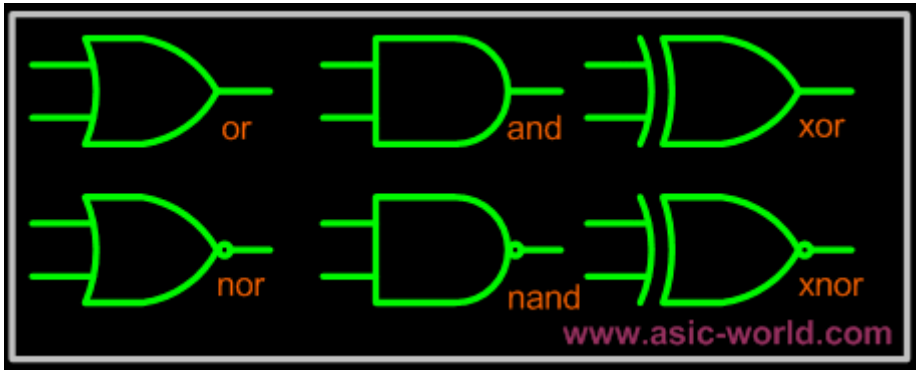
Verilog: структурное описание – экземпляры модулей




```
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
```

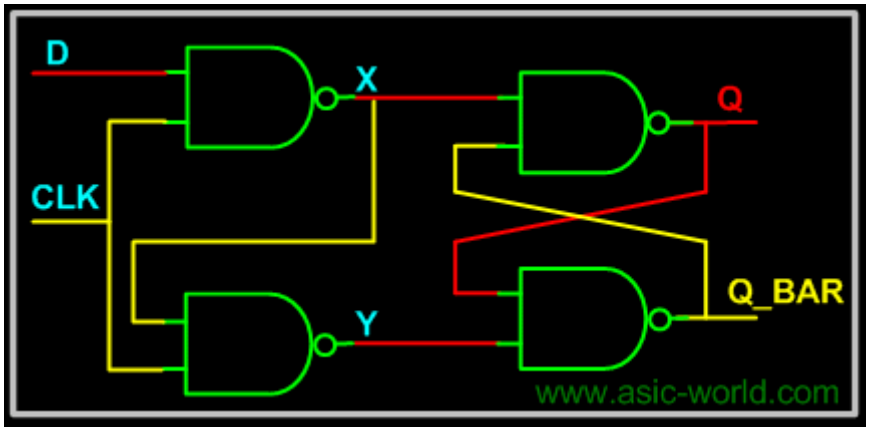
Все встречающиеся в описании имена (в том числе провода: wire) должны быть определены перед первым использованием

Verilog: структурное описание – примитивы



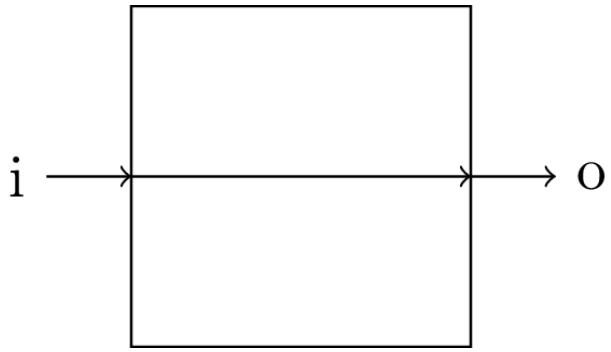

```
module m(…);
   …
   not(o1,i1);
   or(o2,i1,…,in);
   and(o3,i1,…,in);
   …
endmodule
```

Verilog: структурное описание – пример




```
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
```

Verilog: структурное описание – 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

Verilog: 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

Verilog: 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)

Verilog: 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

Verilog: 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

Verilog: 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?

Verilog: функциональное описание – 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)

Verilog: функциональное описание – 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

Verilog: регистры(переменные)

- При выставлении сигналов в схеме могут понадобиться дополнительные (неявные) ячейки памяти
- Чтобы компилятор имел возможность распознать такие места и по необходимости синтезировать дополнительную память, в Verilog вводится понятие регистра или переменной (терминология менялась в стандарте)
- Всё, что появляется в присваиваниях (=, <=) слева, должно быть объявлено как переменная:

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

- Всё остальное может быть объявлено переменной

Verilog: регистры(переменные)

- Имя не может одновременно быть переменной и проводом
- В некоторых случаях (например, при встрече в левой части непрерывного присваивания) имя не может быть переменной
- Все входы и выходы являются проводами по умолчанию
- Все входы обязаны быть проводами
- Выходы можно определять как переменные: достаточно
  - дописать в начале модуля `reg «имя выхода»;` или
  - при определении выхода написать

```
output reg «имя выхода»
```

вместо

```
output «имя выхода»
```

Verilog: функциональное описание – условные переходы

```
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++)

Verilog: 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
```

Verilog: функциональное описание – примеры

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

  assign sum = a + b;
endmodule
```

Verilog: параметры

Иногда бывает нужно написать несколько невероятно похожих, но всё же разных модулей

```
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

module register5(
  input load, reset, clock,
  input [4:0] in,
  output reg [4:0] out);

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

Verilog: параметры

- Чтобы описать сразу всё разнообразие модулей, отличающихся только какими-то константными значениями (например, регистры — размером шины), достаточно описать один модуль с соответствующими параметрами:

```
module register
  #(parameter Width = 5)
  ( input load, reset, clock,
    input [Width-1:0] in,
    output reg [Width-1:0] out
  );
  always @(posedge clock, negedge reset)
    if(~reset) out <= 0;
    else if(~load) out <= in;
endmodule
```

Verilog: параметры

- Чтобы описать сразу всё разнообразие модулей, отличающихся только какими-то константными значениями (например, регистры — размером шины), достаточно описать один модуль с соответствующими параметрами:
  Или так:

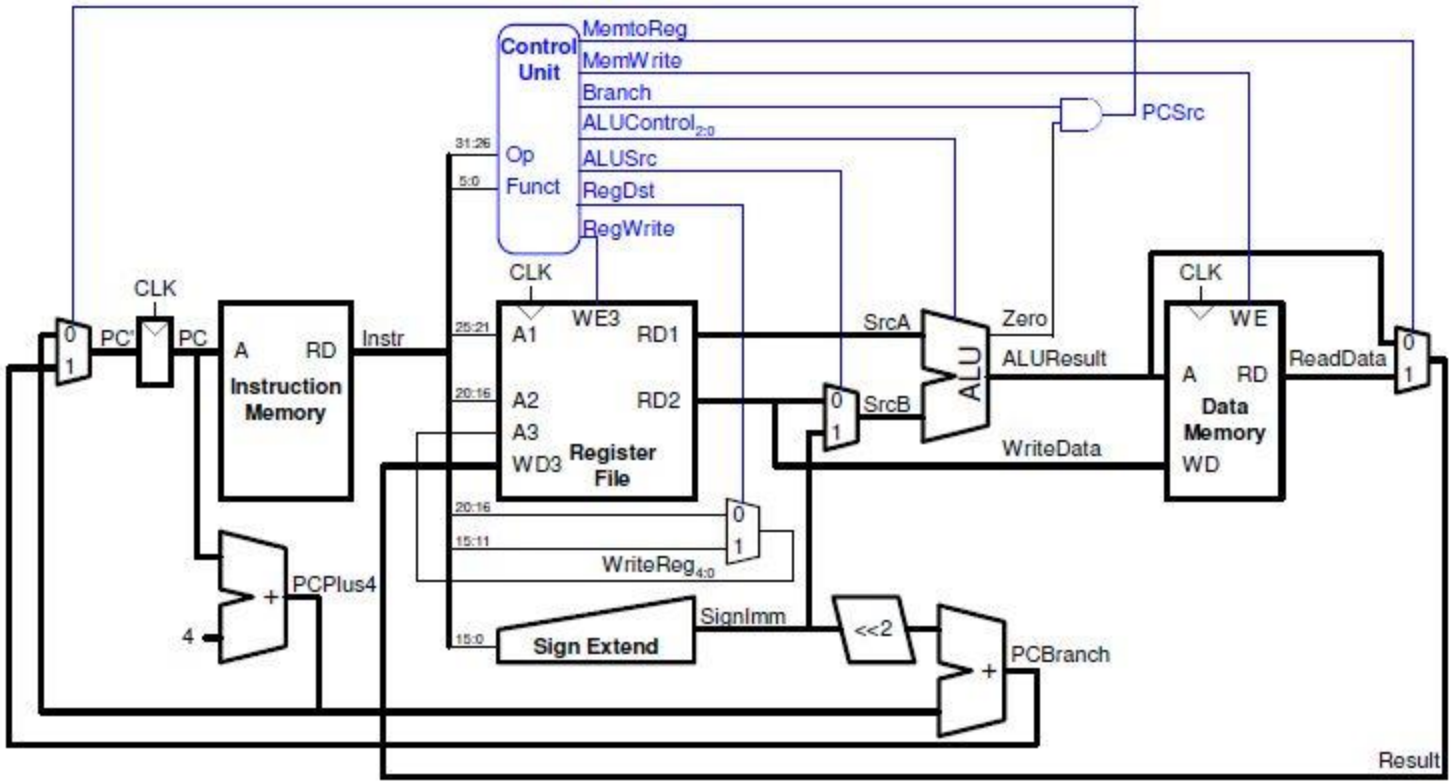
```
module register(load, reset, clock, in , out);
  parameter Width = 5;
  input load, reset, clock;
  input [Width-1:0] in;
  output reg [Width-1:0] out;

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

Verilog: параметры

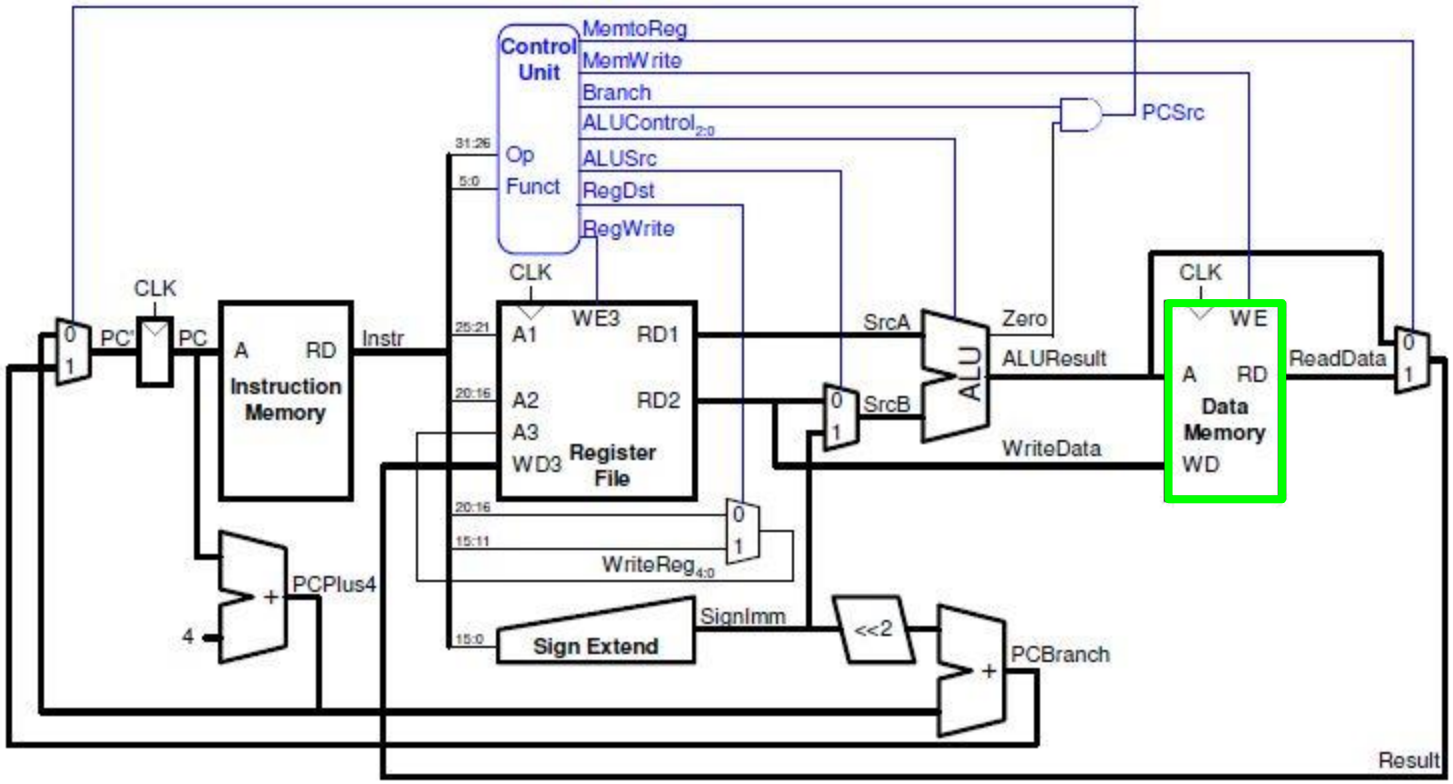
```
parameter Width = 5;
```

- Параметр можно писать вместо числа почти везде в модуле (нельзя — в константах на месте размера)
- Значение параметра по умолчанию указывается при его определении (здесь — 5)
- Экземпляр параметризованного модуля может быть вызван двумя способами:
  - с явным указанием параметров (указание параметров — такое же, как и входов-выходов)
    `register r #(.Width(3)) («arguments»)`
  - без указания параметров — тогда подставляется значение по умолчанию
    `register r («arguments»)`

Verilog: реальные примеры



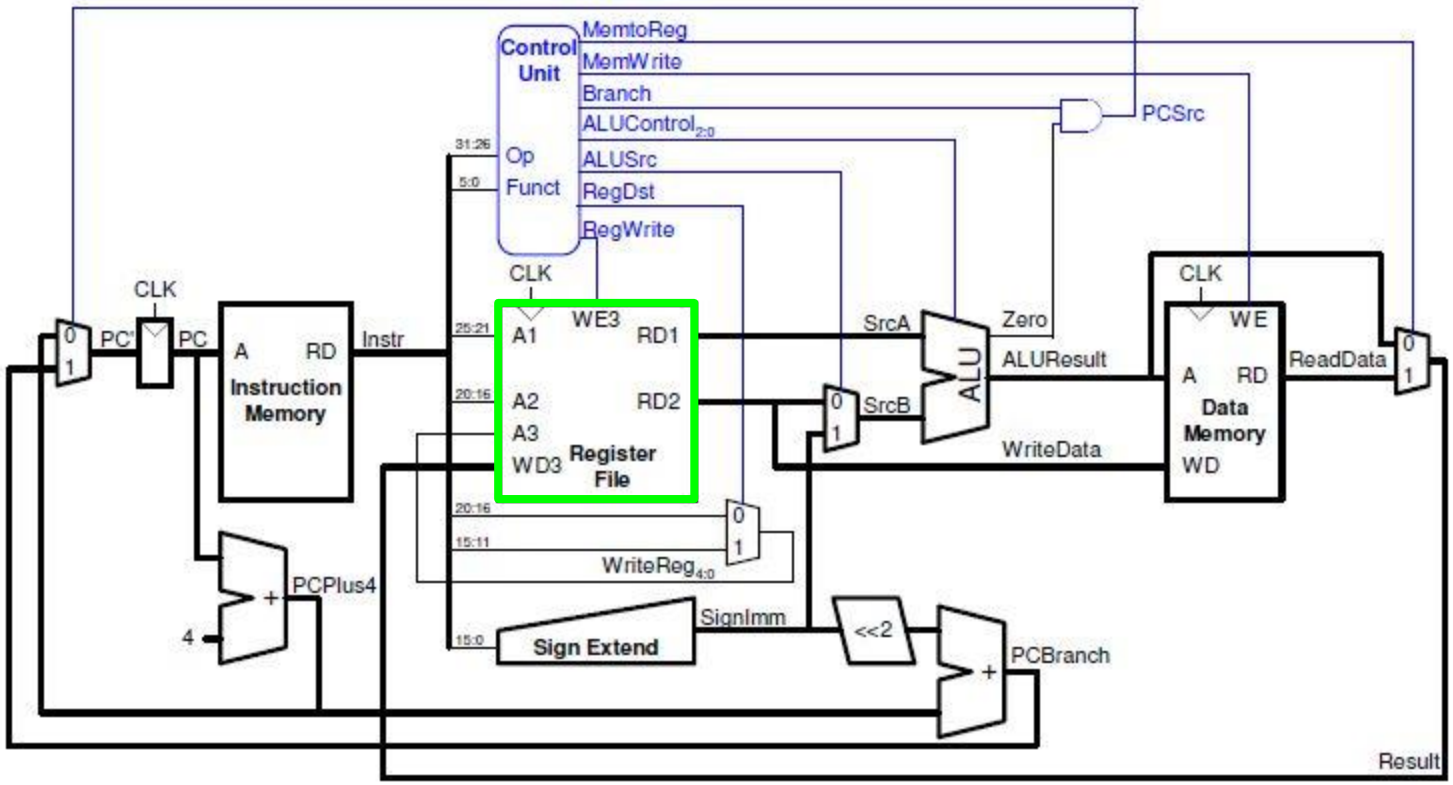
Verilog: реальные примеры – оперативная память



Verilog: реальные примеры – оперативная память

```
// ram.v
// RAM for the single-cycle processor
module ram #(parameter N = 6, M = 32)
            (input  wire          clk,
             input  wire          we,
             input  wire [N-1:0]  adr,
             input  wire [M-1:0]  din,
             output reg  [M-1:0]  dout);

  reg [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we) mem[adr] <= din;

  assign dout = mem[adr];
endmodule
```

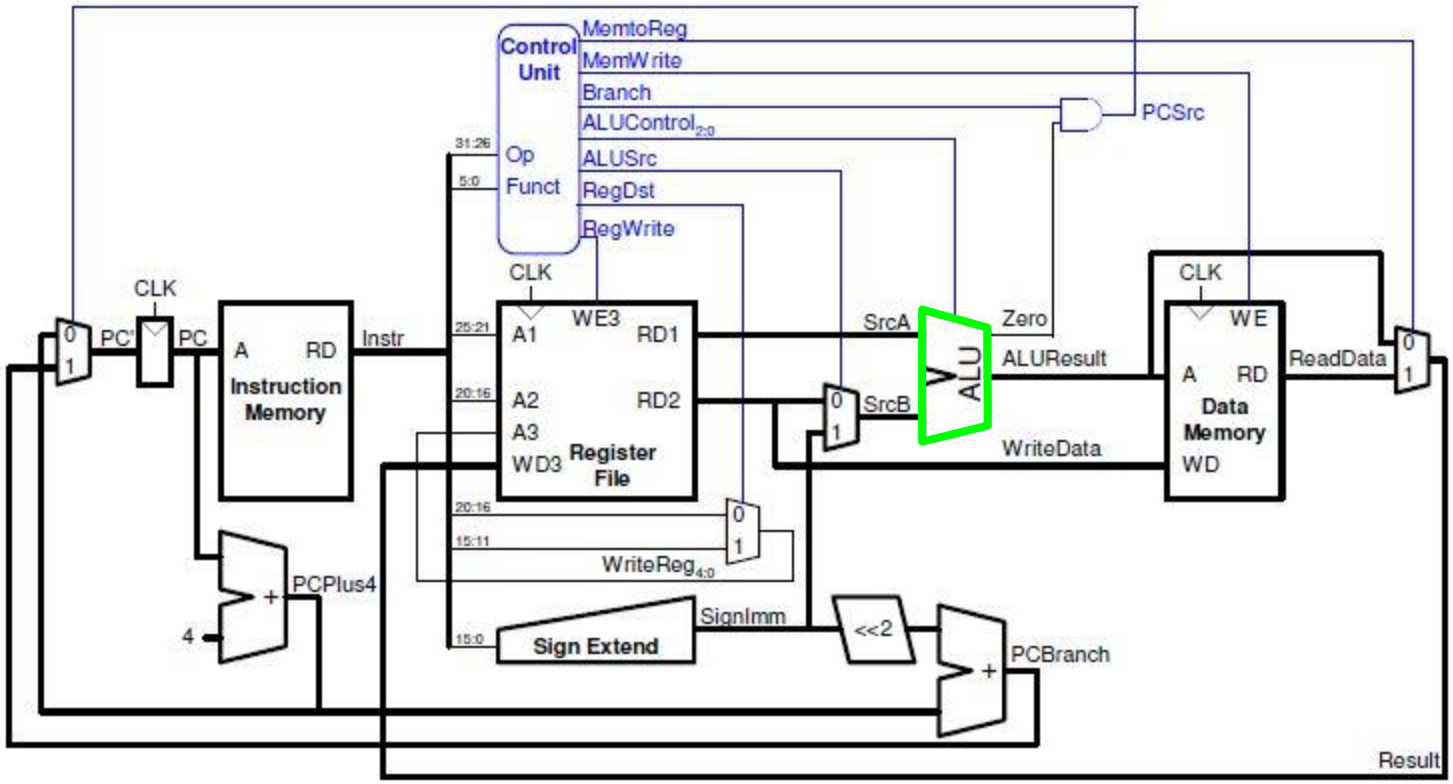
Verilog: реальные примеры – регистровый файл



Verilog: реальные примеры – регистровый файл

```
// regfile.v
// Register file for the single-cycle processor
module regfile(input  wire        clk,
               input  wire        we3,
               input  wire [4:0]  ra1, ra2, wa3,
               input  wire [31:0] wd3,
               output reg  [31:0] rd1, rd2);

  reg [31:0] rf[31:0];

  always @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

Verilog: реальные примеры – арифметико-логическое устройство



Verilog: реальные примеры – арифметико-логическое устройство

```
// alu.v
// ALU for the single-cycle processor
module alu(input  wire [31:0] a, b,
           input  wire [2:0]  alucontrol,
           output reg [31:0] result,
           output reg        zero);

  reg [31:0] condinvb, sum;
  assign condinvb = alucontrol[2] ? ~b : b;
  assign sum = a + condinvb + alucontrol[2];
  always @(*)
    case (alucontrol[1:0])
      2'b00: result = a & b;
      2'b01: result = a | b;
      2'b10: result = sum;
      2'b11: result = sum[31];
    endcase
  assign zero = (result == 32'b0);
endmodule
```