Основы проектирования аппаратных ускорителей систем искусственного интеллекта

Лекция 4 – Продолжение Verilog

План лекции

- Параметры
- Проектирование базовых арифметических блоков
- Generate блоки

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
```

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
```

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
```

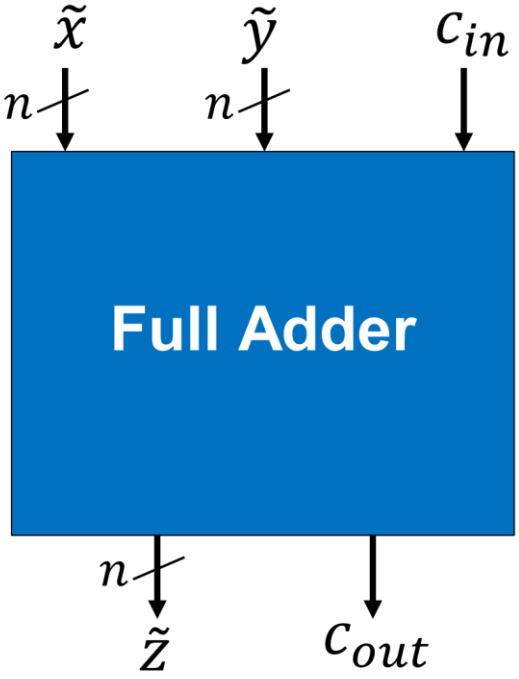
Параметры

```
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 #(.Width(3)) r («arguments»)`
  - без указания параметров — тогда подставляется значение по умолчанию
    `register r («arguments»)`

Параметрический сумматор

Сумматором (Adder) называют вычислительный блок, реализующий сложение двух n-разрядных двоичных чисел $x = (x_{n-1}, \dots, x_0)$ и $y = (y_{n-1}, \dots, y_0)$. Результат сложения часто представляют в виде n-разрядного двоичного числа $z = (z_{n-1}, \dots, z_0)$ и дополнительного однобитного двоичного числа c_{out}, которое сигнализирует о возникновении бита переноса (переполнения), то есть ситуации, когда результирующая сумма является $(n + 1)$-битным числом.



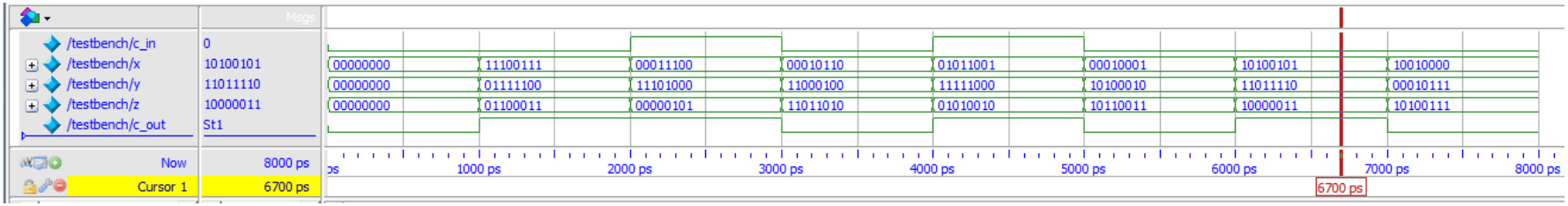
Стоит отметить, что часто сумматор имеет дополнительный вход c_{in}, который используется для учета возможного переноса, возникающего в других арифметических блоках. Наличие такого входа позволяет строить более сложные арифметические блоки из более простых.

Параметрический сумматор

```
module adder
#(
    parameter WIDTH = 8
)
(
    input  [WIDTH - 1:0] x, y,
    input  carry_in,
    output [WIDTH - 1:0] z,
    output carry_out
);
    assign {carry_out, z} = x + y + carry_in;
endmodule
```

```
Transcript
# Top level modules:
#       testbench
# End time: 14:58:01 on Oct 22,2018, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# vsim work.testbench
# Start time: 14:58:01 on Oct 22,2018
# Loading work.testbench
# Loading work.adder
# Time: 0, x: 00000000, y: 00000000, c_in: 0, z: 00000000, c_out: 0
# Time: 1, x: 11100111, y: 01111100, c_in: 0, z: 01100011, c_out: 1
# Time: 2, x: 00011100, y: 11101000, c_in: 1, z: 00000101, c_out: 1
# Time: 3, x: 00010110, y: 11000100, c_in: 0, z: 11011010, c_out: 0
# Time: 4, x: 01011001, y: 11111000, c_in: 1, z: 01010010, c_out: 1
# Time: 5, x: 00010001, y: 10100010, c_in: 0, z: 10110011, c_out: 0
# Time: 6, x: 10100101, y: 11011110, c_in: 0, z: 10000011, c_out: 1
# Time: 7, x: 10010000, y: 00010111, c_in: 0, z: 10100111, c_out: 0
# ** Note: $stop    : ../testbench.v(27)
#    Time: 8 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at ../testbench.v line 27
# 0 ps
# 8400 ps
```

Листинг 5.1 – Поведенческое описание сумматора

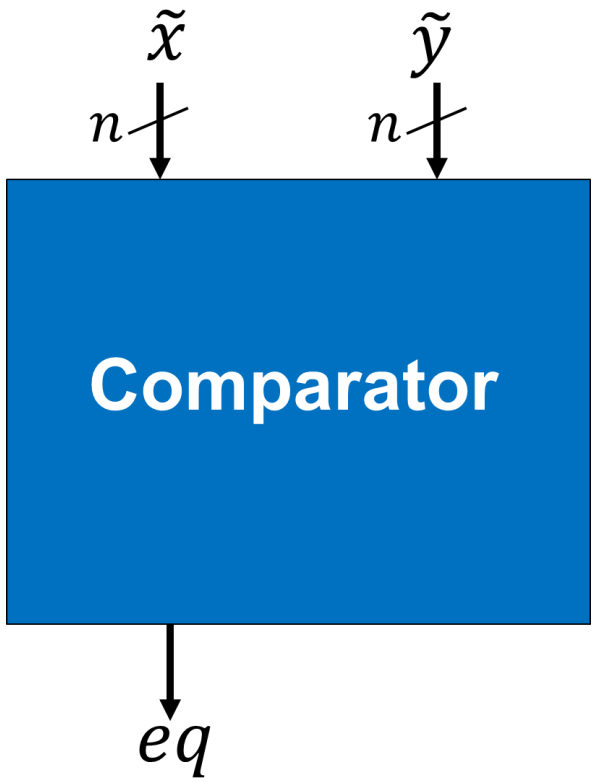


Компаратор

Компараторами (Comparators) называют различные логические устройства, предназначенные для сравнения чисел между собой.

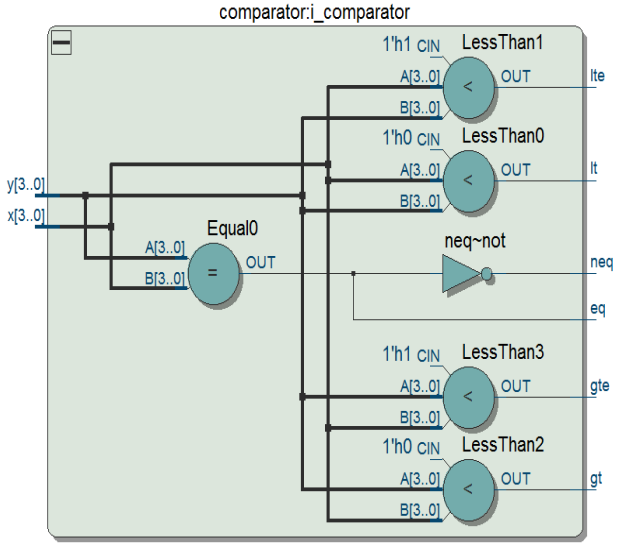
Чаще всего входами компаратора являются две **n**-разрядные шины, а выходом является сигнальный однобитный выход, который содержит единицу, если входные аргументы удовлетворяют выбранному соотношению сравнения, и ноль – в ином случае.

При этом входные аргументы трактуются как целые числа со знаком или без знака, а в качестве отношения сравнения является сравнение на равенство, строгое неравенство или нестрогое неравенство аргументов.

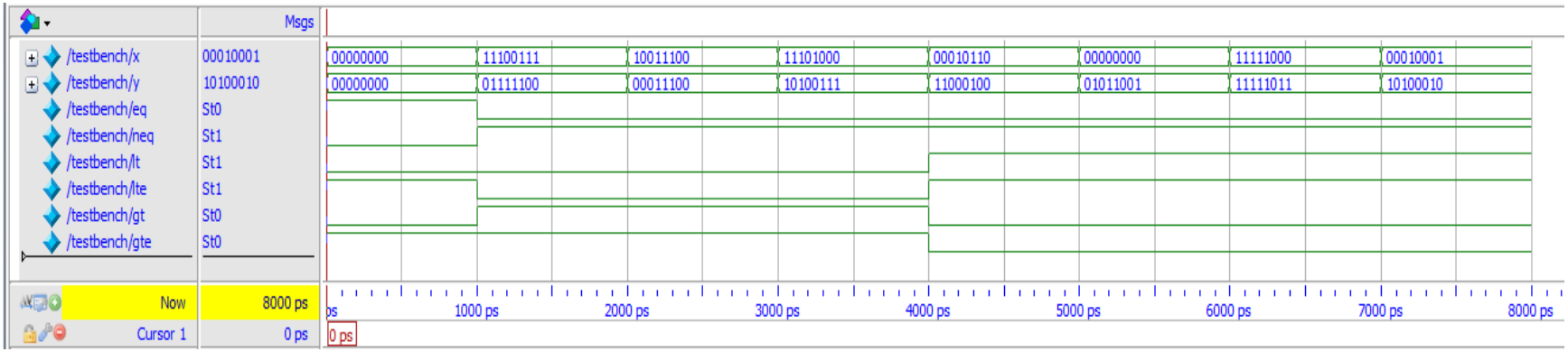


Компаратор

```
module comparator
#(
    parameter WIDTH = 8
)
(
    input  [WIDTH - 1:0] x, y,
    output eq, neq, lt, lte, gt, gte
);
    assign eq  = (x == y);
    assign neq = (x != y);
    assign lt  = (x <  y);
    assign lte = (x <= y);
    assign gt  = (x >  y);
    assign gte = (x >= y);
endmodule
```



Листинг 5.1 – Поведенческое описание компаратора



Устройство сдвига

Устройствами сдвига (shifters) и устройствами циклического сдвига (barrel shifters, rotators) являются логические блоки, представляющие собой сдвиг в адресации элементов некоторой шины данных на заданное число позиций.

Пусть $x = (x_{n-1}, \dots, x_0)$ – некоторая шина данных и s – количество позиций, на которые требуется осуществить сдвиг, тогда выходом устройства сдвига является шина данных $z = (z_{n-1}, \dots, z_0)$, где

$$\boldsymbol{z_i} = \begin{cases} \boldsymbol{x_{i+s}}, \text{если } \boldsymbol{i < n - s}; \\ \boldsymbol{0}, \quad \text{в ином случае}; \end{cases}$$

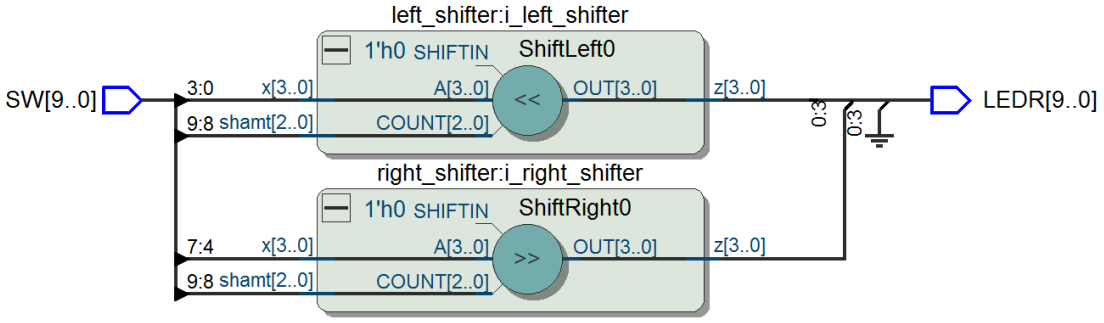
в случае, если сдвиг осуществляется вправо. Если сдвиг осуществляется влево, то шина данных z определяется следующим образом:

$$\boldsymbol{z_i} = \begin{cases} \boldsymbol{x_{i-s}}, \text{если } \boldsymbol{i > s - 1}; \\ \boldsymbol{0}, \quad \text{в ином случае}. \end{cases}$$

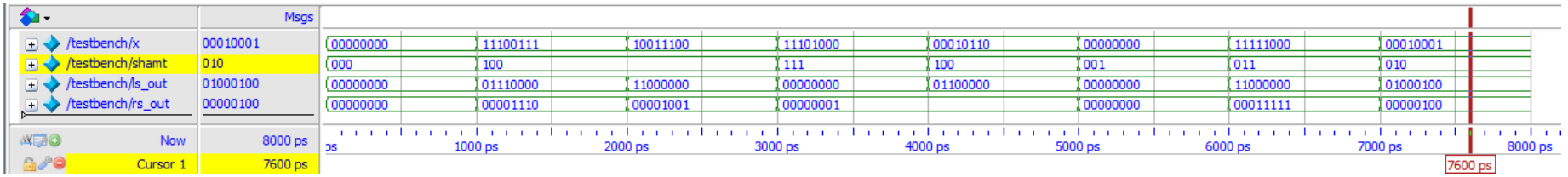
Операции сдвига вправо и влево могут быть интерпретированы как арифметические операции целочисленного деления и умножения на 2^s, соответственно.

Устройство сдвига

```
module left_shifter
#(
    parameter WIDTH = 8,
    parameter SHIFT = 3
)
(
    input  [WIDTH - 1:0] x,
    input  [SHIFT - 1:0] shamt,
    output [WIDTH - 1:0] z
);
    assign z = x << shamt;
endmodule
```



Листинг 5.1 – Поведенческое описание блока сдвига влево



Устройство циклического сдвига

В свою очередь устройства циклического сдвига интерпретируют индексы шины данных как замкнутые в кольцо, то есть следующим индексом после $(n-1)$ является индекс 0. Таким образом, сдвиг индексов происходит по модулю длины шины данных.

Формально результирующую шину данных $z = (z_{n-1}, \dots, z_0)$ устройства циклического сдвига на s позиций можно задать следующим образом:

$$\boldsymbol{z_i = x_{(i+s) \bmod n}},$$

в том случае, когда сдвиг осуществляется вправо.

В случае, когда сдвиг осуществляется влево, шина данных z определяется следующим образом:

$$\boldsymbol{z_i = x_{(i-s) \bmod n}}.$$

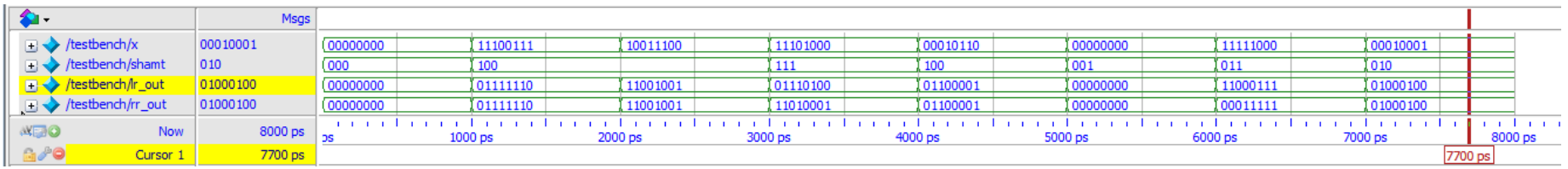
Устройство циклического сдвига

```
module left_rotator
#(
    parameter WIDTH = 8,
    parameter SHIFT = 3
)
(
    input  [WIDTH - 1:0] x,
    input  [SHIFT - 1:0] shamt,
    output [WIDTH - 1:0] z
);

    wire [2 * WIDTH - 1:0] temp;
    assign temp = {x, x} << shamt;
    assign z = temp[2 * WIDTH - 1: WIDTH];
endmodule
```

Листинг 5.1 – Устройство циклического сдвига влево

```
module right_rotator
#(
    parameter WIDTH = 8,
    parameter SHIFT = 3
)
(
    input  [WIDTH - 1:0] x,
    input  [SHIFT - 1:0] shamt,
    output [WIDTH - 1:0] z
);

    wire [2 * WIDTH - 1:0] temp;
    assign temp = {x, x} >> shamt;
    assign z = temp[WIDTH - 1: 0];
endmodule
```

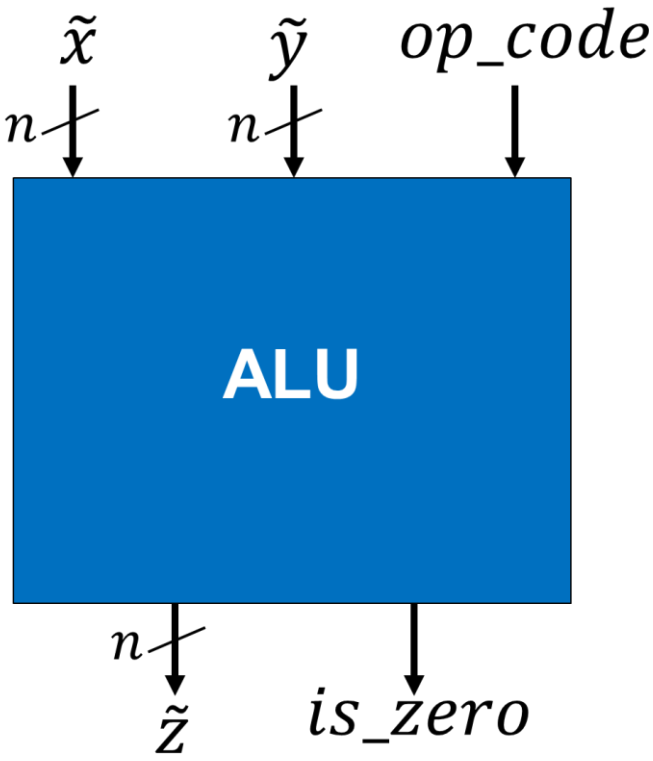
Листинг 5.1 – Устройство циклического сдвига вправо



Арифметико-логическое устройство

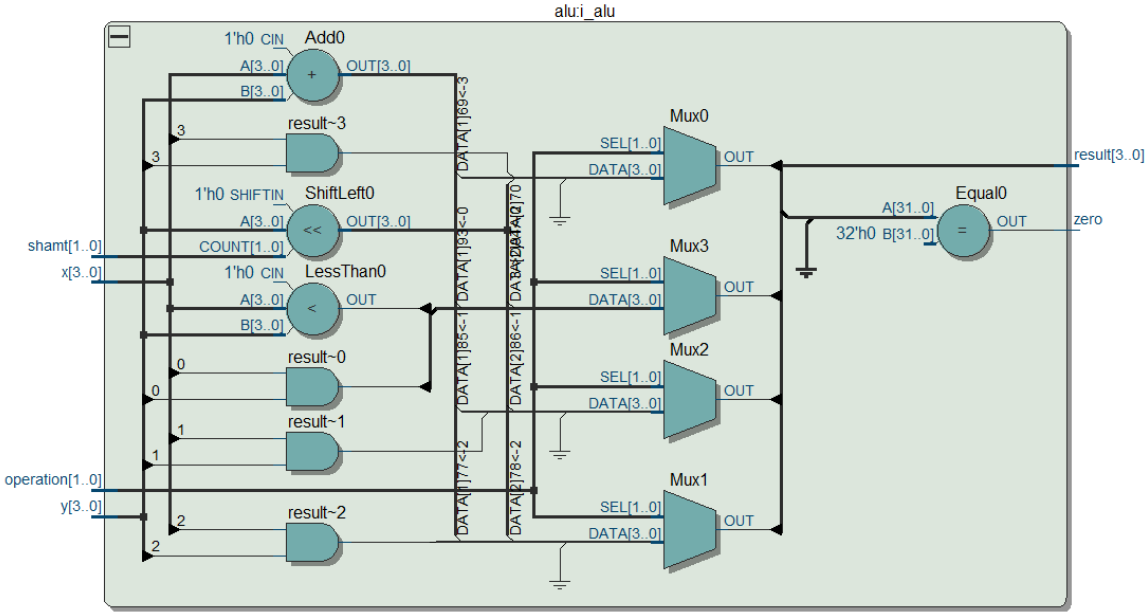
Арифметико-логическое устройство (**АЛУ**) представляет собой блок, позволяющий выполнять ряд арифметических и логических операций над заданным количеством аргументов. Конкретный набор операций, количество операндов и разрядность у поддерживаемых операций может очень сильно варьироваться, в зависимости от типа устройства, для которого разрабатывается **АЛУ**.

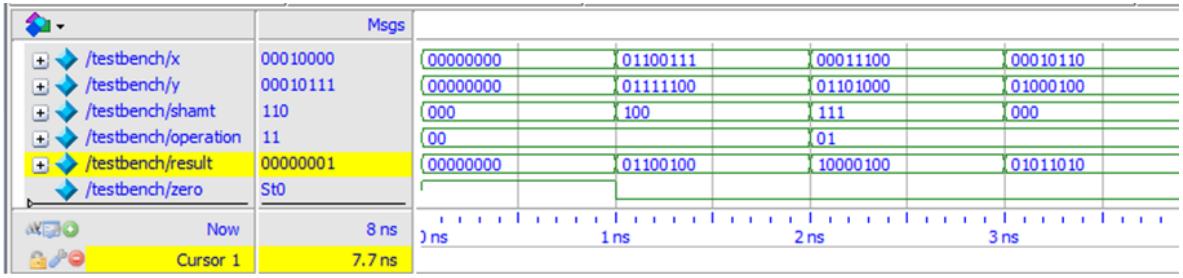
Кроме того, очень часто **АЛУ** имеет ряд дополнительных выходов, которые передают специальные сигналы-индикаторы (флаги), которые обращаются в единицу при наступлении определенного события (например, при переполнении результата или обращении результата в ноль).



Арифметико-логическое устройство

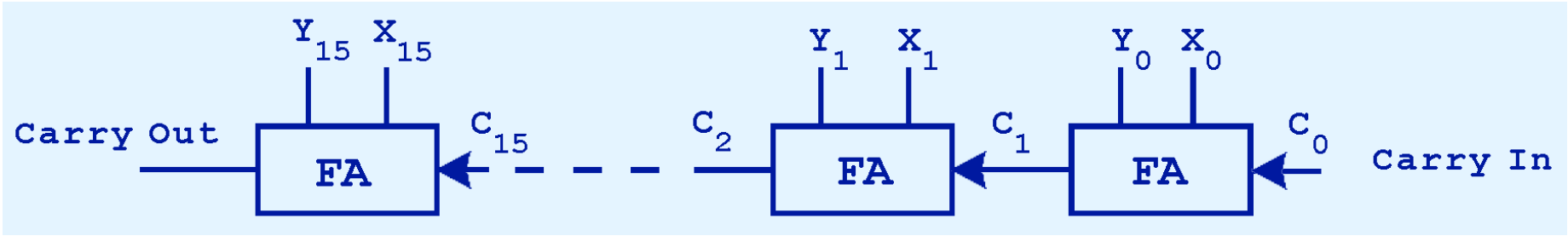
```
//ALU commands
`define ALU_AND 2'b00
`define ALU_ADD 2'b01
`define ALU_SLL 2'b10
`define ALU_SLT 2'b11

module alu
#(
    parameter WIDTH = 4,
    parameter SHIFT = 2
)
(
    input       [WIDTH - 1:0] x, y,
    input       [SHIFT - 1:0] shamt,
    input       [ 1:0]        operation,
    output                    zero,
    output reg  [WIDTH - 1:0] result
);

    always @ (*) begin
        case (operation)
            `ALU_AND : result = x & y;
            `ALU_ADD : result = x + y;
            `ALU_SLL : result = y << shamt;
            `ALU_SLT : result = (x < y) ? 1 : 0;
        endcase
    end

    //Flags
    assign zero = (result == 0);
endmodule
```





Generate блок

- Часто возникает необходимость описать множество однотипных блоков. Например, сумматор с каскадным переносом



- Для таких случае в языке Verilog есть конструкция generate

Generate блок - синтаксис

- Задается ключевыми словами

```
generate … endgenerate
```

```
generate

    genvar i;

    for (i = 0; i <= WIDTH - 1; i = i + 1)
        begin : stage
            //generated statements
        end
endgenerate
```

Generate блок - синтаксис

- Переменных, которые используются в цикле генерации, объявляются при помощи `genvar`
- Можно объявить несколько переменных

```
generate

    genvar i;

    for (i = 0; i <= WIDTH - 1; i = i + 1)
        begin : stage
            //generated statements
        end
endgenerate
```

Generate блок - синтаксис

- Основной блок в цикле генерации должен быть именованным

```
generate

    genvar i;

    for (i = 0; i <= WIDTH - 1; i = i + 1)
        begin : stage
            //generated statements
        end
endgenerate
```

Generate блок - синтаксис

- Внутри основного блока можно использовать:
  - непрерывное присваивание
  - объявление экземпляров модулей
  - объявление **`always`**-блоков
  - объявление **`generate`**-блоков
- Гибкость генерации можно обеспечить использованием условного оператора и оператора **`case`**

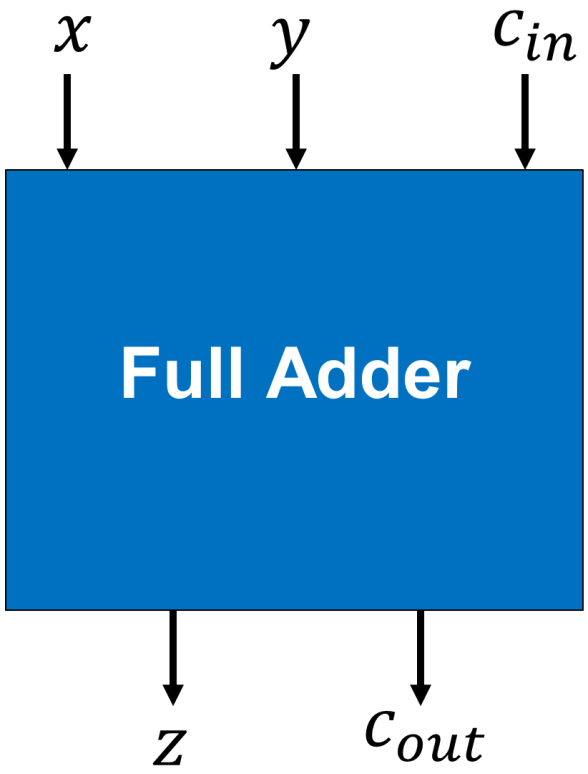
Generate блок - пример

- Сумматор с каскадным переносом

```
generate

    genvar i;

    for (i = 0; i <= WIDTH - 1; i = i + 1)
    begin : stage
        case(i)
          0       : assign {c[i],z[i]}   = x[i]+y[i]+c_in;
          WIDTH -1: assign {c_out,z[i]} = x[i]+y[i]+c[i-1];
          default : assign {c[i],z[i]}   = x[i]+y[i]+c[i-1];
    end
endgenerate
```

Полный сумматор

```
module full_adder_structural
(
    input wire  x, y, carry_in,
    output wire z, carry_out
);
    wire [2:0] t;

    xor(t[0], x, y);
    xor(z, t[0], carry_in);
    and(t[1], x, y);
    and(t[2], t[0], carry_in);
    or (carry_out, t[1], t[2]);

endmodule
```



Каскадное соединение сумматоров

```
module ripple_carry_adder
# (
    parameter WIDTH = 8
)
(
    input                   carry_in,
    input  [WIDTH - 1 : 0] x,
    input  [WIDTH - 1 : 0] y,
    output [WIDTH - 1 : 0] z,
    output                  carry_out
);
    wire [WIDTH : 0] carry;

    assign carry[0] = carry_in;

    generate
            genvar i;
            for (i = 0; i <= WIDTH - 1; i = i + 1)
        begin : stage
            full_adder FA
            (
                .x        (x    [i]    ),
                .y        (y    [i]    ),
                .z        (z    [i]    ),
                .carry_in (carry[i]    ),
                .carry_out(carry[i + 1])
            );
        end
    endgenerate

    assign carry_out = carry[WIDTH];

endmodule
```