Лабораторная работа № 2. Арифметико-логическое устройство

1. **Цель работы:**

a) Изучение новых конструкций языка Verilog (SystemVerilog)
b) Реализация схемы АЛУ
c) Написание теста для проверки АЛУ

2. **Краткие теоретические сведения**

Арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ) – это устройство, которое принимает на входе два N-разрядных операнда и, в зависимости от управляющего сигнала, производит на ними те или иные действия и выдаёт N-разрядный результат.

АЛУ имеет входы операндов A и B, входы выбора операций F, выход результата Y. Основой АЛУ служит сумматор, схема которого дополнена логикой, расширяющей функциональные возможности АЛУ и обеспечивающей его перестройку с одной операции на другую. На рисунке приведена схема АЛУ, выполняющего семь операций.

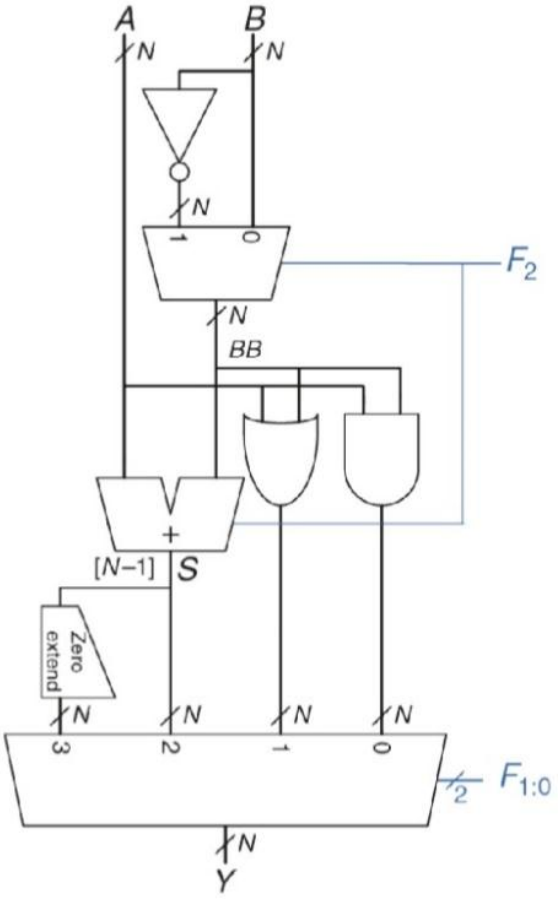

Рисунок 1 – Схема АЛУ

Функции, задаваемые селектором, приведены в таблице 1.

Таблица 1 – Функции селектора

F[2:0]	Функция
000	A & B
001	A | B
010	A + B
011	не используется
100	A & ~B
101	A | ~B
110	A – B
111	SLT*

*SLT(set if less than) – это функция которая выдаёт 1 если A < B, и 0 во всех остальных случаях.

Основные конструкции Verilog (SystemVerilog)

В языке Verilog (SysytemVerilog) можно объединять провода в шины.

Например, строка

```
wire [9:0] a;
```

объявляет десятиразрядную шину. Из шины можно выбрать некоторый диапазон битов и назначить другой шине с таким же количеством битов:

```
wire [3:0] b = a[8:5];
```

Для записи целых чисел (констант) используется следующий синтаксис:

N’BValue

где N это разрядность числа, B – система счисления, в которой это число записано, Value – само число.

Например, число 41232_{10} мы можем записать следующим образом (в шестнадцатеричной системе счисления):

```
4’hA110
```

Внутри констант можно использовать символ «нижнее подчёркивание» для удобного разделения разрядов. Например, запись `8’b0010_0001` будет эквивалентна записи `8’b00100001`.

Блок always

Для удобства описания схем язык Verilog имеет процедурные блоки «always».

Для описания процедурного блока используется следующий синтаксис:

always @(<sensitivity_list>) <statements>

Где <sensitivity_list> – это список всех входных сигналов, к которым чувствителен блок. Т.е. оператор выполняется только тогда, когда случается событие, заданное в списке чувствительности.

Такую запись можно прочитать следующим образом: "Всегда выполнять выражения <statements> при изменении сигналов, описанных в списке чувствительности <sensitivity list>".

Операторы always можно использовать для создания триггеров, защелок или комбинационной логики в зависимости от списка чувствительности и

оператора. Из-за подобной гибкости языка при синтезе аппаратных блоков можно непреднамеренно получить нежелательную конфигурацию.

Во избежание таких ошибок в SystemVerilog введены операторы always_ff, always_latch и always_comb. Оператор always_ff ведет себя так же, как always, но используется только тогда, когда подразумевается синтез триггеров, и позволяет инструментальной среде в противном случае выдавать предупреждение. Оператор always_comb, соответственно, используется только тогда, когда необходимо синтезировать комбинационную логику.

Типы присвоений

В языке Verilog (Sysytem Verilog) существует три типа присвоений – непрерывное (постоянное), блокирующее, неблокирующее.

Непрерывное (постоянное) присвоение всегда разворачивается в комбинационную логику. Непрерывное присваивание может встречаться в операторе assign. Левой частью всегда является сигнал, правой — выражение, использующее любые другие сигналы.

Пример, приведённый в листинге 1 синтезируется в схему, изображенную на рисунке 2.

Листинг 1 – Использование непрерывного присваивания

```
wire a,b;
assign a=~b;
```

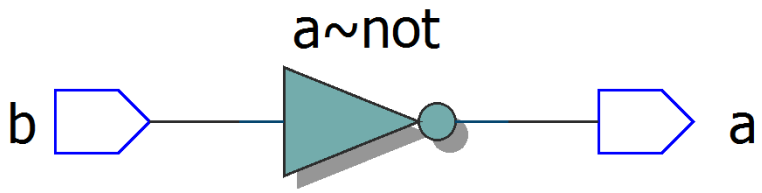

Рисунок 2 – Схема, синтезируемая по коду из листинга 1

Такой тип присваиваний применяется вне процедурных блоков.

Внутри процедурных блоков применяются блокирующие (blocking, =) и неблокирующие (nonblocking, <=) присваивания (вместе называемые процедурными).

Все операторы неблокирующего присваивания внутри одного блока always «выполняются» одновременно, а условия, определяющие произойдут присваивания или нет, определяются заранее. К моменту присваивания, обычно это фронт тактирующего сигнала, все используемые в выражениях сигналы должны иметь установившиеся значения. В противном случае результат выполнения операции может быть непредсказуемым.

При использовании блокирующего присваивания операторы будут «выполняться» последовательно.

Часто использование блокирующих и неблокирующих присваиваний могут приводить к синтезу одинаковых схем.

Ниже рассмотрена разница между процедурными присваиваниями на примерах.

Пример в листинге 2 демонстрирует применение блокирующих присваиваний. Схема, получаемая при синтезе данного кода, приведена на рисунке 3.

Листинг 2 – Использование блокирующих присваиваний

```
module block_module1(
input logic clk,
inout logic x,
inout logic y
);

always @(posedge clk) begin
x = ~x;
y = ~y;
end
```

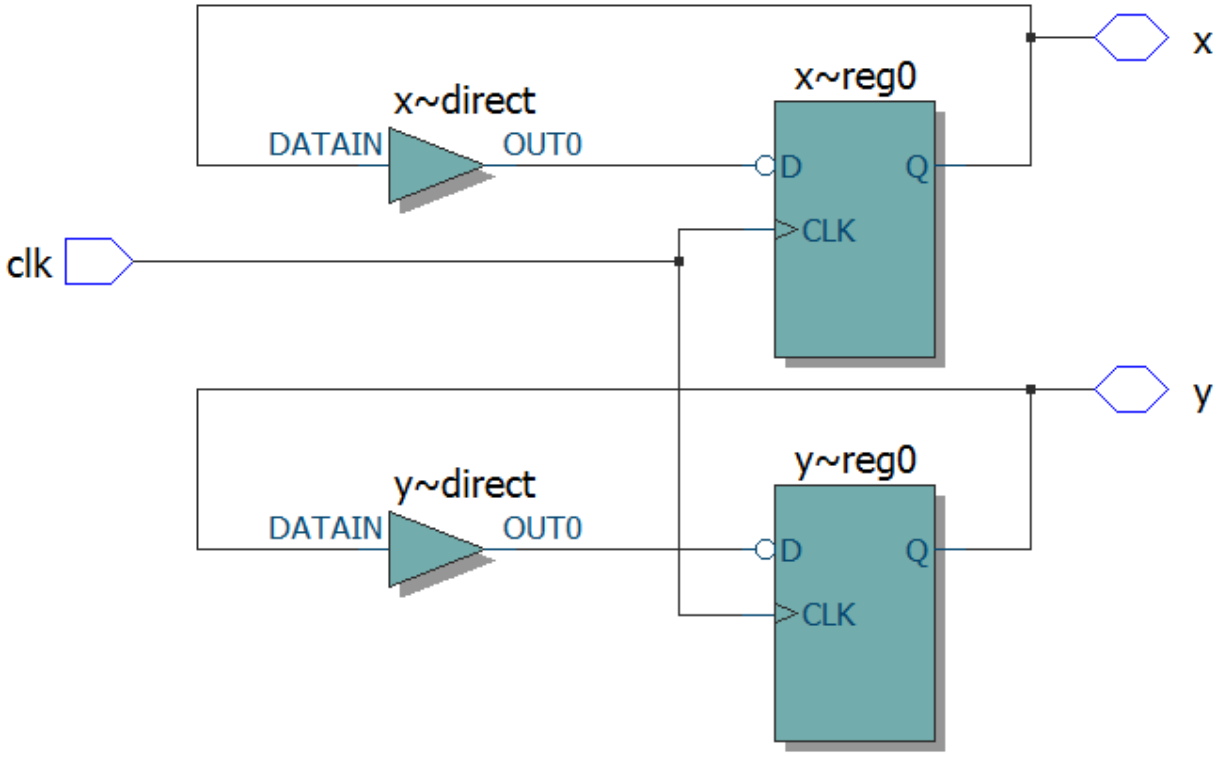

Рисунок 3 – Схема, синтезируемая по коду из листинга 2

Если же в данном коде использовать неблокирующие присваивания, то синтезируется та же самая схема (можно легко убедиться в этом, самостоятельно синтезировав данные схемы).

Листинг 3 – Использование неблокирующих присваиваний

```
module nonblock_module1(
input logic clk,
inout logic x,
inout logic y
);
```

```
always @(posedge clk) begin
x <= ~x;
y <= ~y;
end

endmodule
```

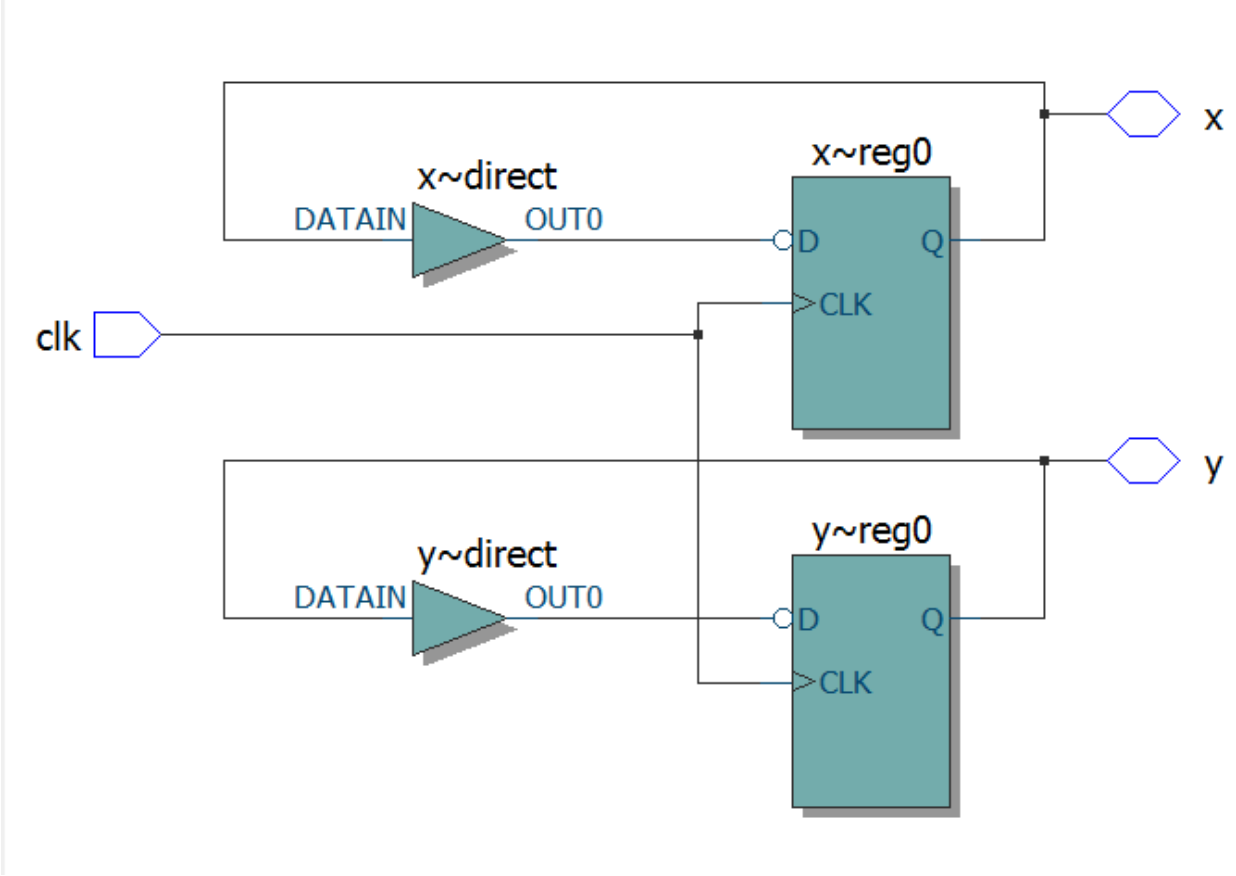

Рисунок 4 – Схема, синтезируемая по коду из листинга 3

Разница между присваиваниями становится видна тогда, когда сигналы зависят друг от друга.

Например, код, приведённый в листинге 4, синтезируется в схему, показанную на рисунке 5.

Листинг 4 – Использование блокирующих присваиваний (второй вариант)

```
module block_module2(
input logic clk,
inout logic x,
inout logic y
);

always @(posedge clk) begin
x = ~x;
y = x;
end
```

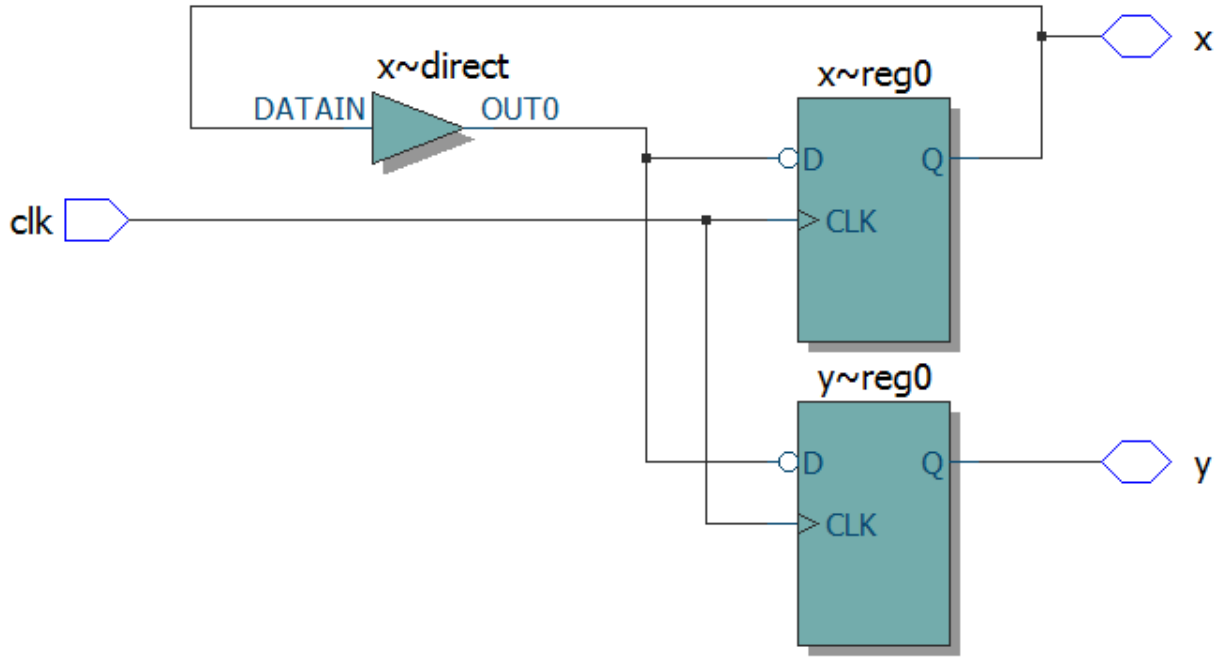

Рисунок 5 - Схема, синтезируемая по коду из листинга 4

Если же в данном коде использовать неблокирующие присваивания, то синтезируется совершенно другая схема (листинг 5, рисунок 6).

Листинг 5 – Использование неблокирующих присваиваний (второй вариант)

```
module nonblock_module2(
input logic clk,
inout logic x,
inout logic y
);

always @(posedge clk) begin
x <= ~x;
y <= x;
end

endmodule
```

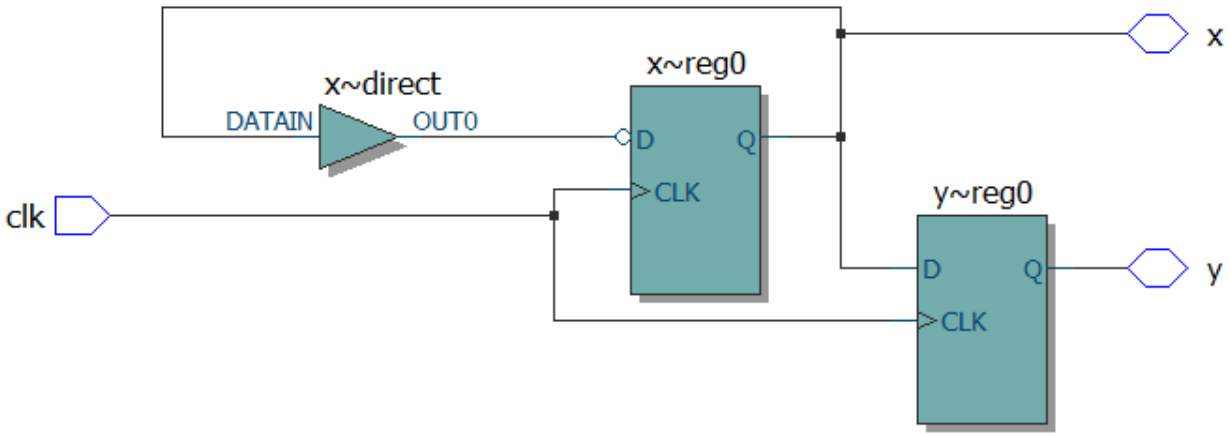

Рисунок 6 - Схема, синтезируемая по коду из листинга 5

Тестирование

Для проверки схемы можно использовать среду тестирования с самопроверкой, для этого используется оператор assert.

```
a = 0; b = 0; c = 0; #10;
assert (y === 1) else $error("000 failed.");
```

Оператор assert в SystemVerilog проверяет, истинно ли указанное условие. Если нет, то выполняется оператор else. Системная процедура $error в операторе else печатает сообщение об ошибке с указанием нарушенного условия.

В SystemVerilog сравнение с помощью == и != работает для сигналов, которые не принимают значения x и z. Среда тестирования использует операторы === и !== для сравнений на равенство и неравенство соответственно, потому что эти операторы работают также и с операндами, значения которых могут быть x или z.

3. Задание

1. Спроектируйте на языке System Verilog АЛУ, показанное на приведенной схеме.
2. Незаконченный прототип такого АЛУ:

```
module alu(
input logic [3:0]operandA, [3:0]operandB,
input logic [2:0]funcsel,
output logic [3:0]result
);

logic [3:0] maybeinvertedB;
assign maybeinvertedB = ???

logic [3:0] sum;
assign sum = ???

always_comb
   case (funcsel[1:0])
      2'b00: result <= ???
      2'b01: result <= ???
      2'b10: result <= ???
      2'b11: result <= ???
   endcase

endmodule
```

Закончите строки со знаками вопроса.

3. Напишите на языке System Verilog модуль, тестирующий работу АЛУ (тестбенч). Модуль сам должен сигнализировать о наличии ошибок в работе АЛУ.

4. Частично тестирующий модуль приведён ниже. Дополните его теми наборами входных и выходных значений (тесткейсами), которых там не хватает.

```
module testbench_alu();
   logic [3:0] operandA, operandB;
   logic [2:0] funcsel;
   logic [3:0] alu_result;

   alu  dut(operandA, operandB, funcsel, alu_result);

   initial begin
      operandA = 4'b0011;
      operandB = 4'b0101;
      funcsel = 3'b000;
      #10
      assert (alu_result === 4'b0001) else $error("A & B failed.");


      funcsel = 3'b001;
      #10
      assert (alu_result === (operandA | operandB)) else $error("A | B failed.");

      operandA = 9;
      operandB = 2;
      funcsel = 3'b010;
      #10
      assert (alu_result === (9+2)) else $error("9 + 2 failed.");
```

```
        operandB = 11;
        funcsel = 3'b010;
        #10
        assert (alu_result === (9+11-16)) else $error("9 + 11 failed.");

        funcsel = 3'b111;
        #10
        assert (alu_result === 1) else $error("9 < 11 failed.");
        $stop;
    end
endmodule
```

5. Добавьте в схему АЛУ дополнительный одноразрядный выход zero, который устанавливается в единицу если все разряды N-разрядного выхода равны 0.
6. Дополните тестбенч новыми тесткейсами, чтобы протестировать доработанную схему АЛУ.