Архитектура вычислительных систем. Домашнее задание №2. Вариант №1.

Комиссаров Данил Андреевич

March 2025

1 Полный отчет

Первый вариант задания. Арифметико-логическое устройство (ALU).

В отличие от предыдущего задания, где достаточно было школьной логики и курса дискретного анализа, здесь требуется умение работы с Verilog, что среди студентов второго курса не является очень распространённым. Поэтому сначала будет уместно разобраться с языком описания аппаратуры с нуля.

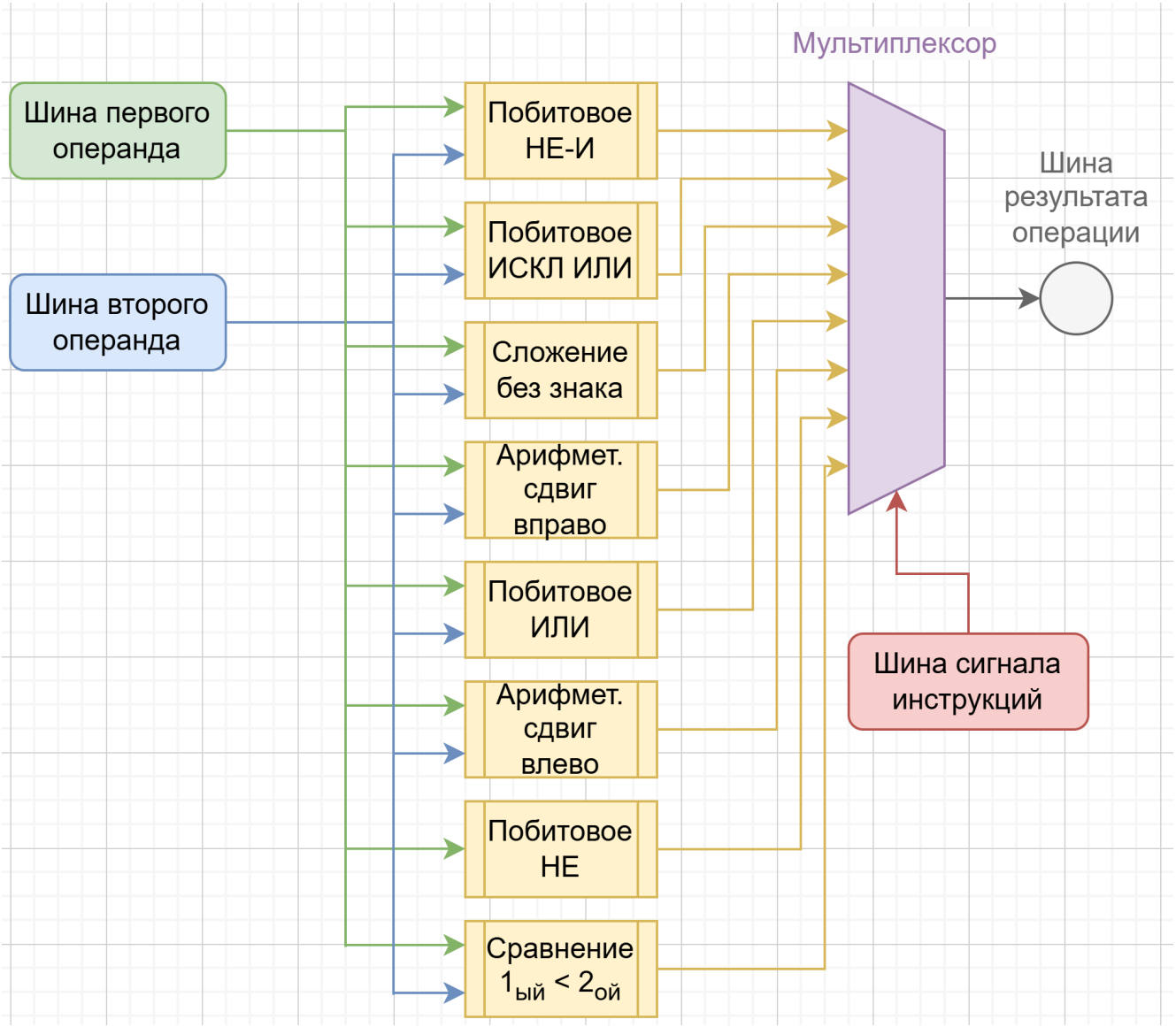
К счастью, это удобно разбирать на примере АЛУ, который и сам состоит из простых элементов: сумматор, компаратор, многобитные логические элементы. Постепенно их создавая, придём к полному пониманию языка.

1.1 Разбор

Что вообще представляет собой так называемый модуль **"alu_register"**? Вообще это некий блок, который под управлением устройства управления служит для выполнения арифметических и логических преобразований над данными, называемыми в этом случае операндами. Разрядность операндов обычно называют размером или длиной машинного слова. То есть из себя АЛУ представляет несколько блоков комбинационной логики, которые и являются преобразованиями над операндами, и блока коммутатора, который соединяет соответствующий блок с выходом в зависимости от сигнала инструкций. В ТЗ описаны следующие команды:

opcode_i	Операция над first_i и second_i
3'b000	Побитовое НЕ-И
3'b001	Побитовое Исключающее ИЛИ
3'b010	Сложение (без знака)
3'b011	Арифметический (знаковый) сдвиг first_i вправо на second_i
3'b100	Побитовое ИЛИ
3'b101	Логический сдвиг first_i влево на second_i
3'b110	Побитовое НЕ для first_i
3'b111	first_i < second_i

Тогда схематично можно так представить себе АЛУ:



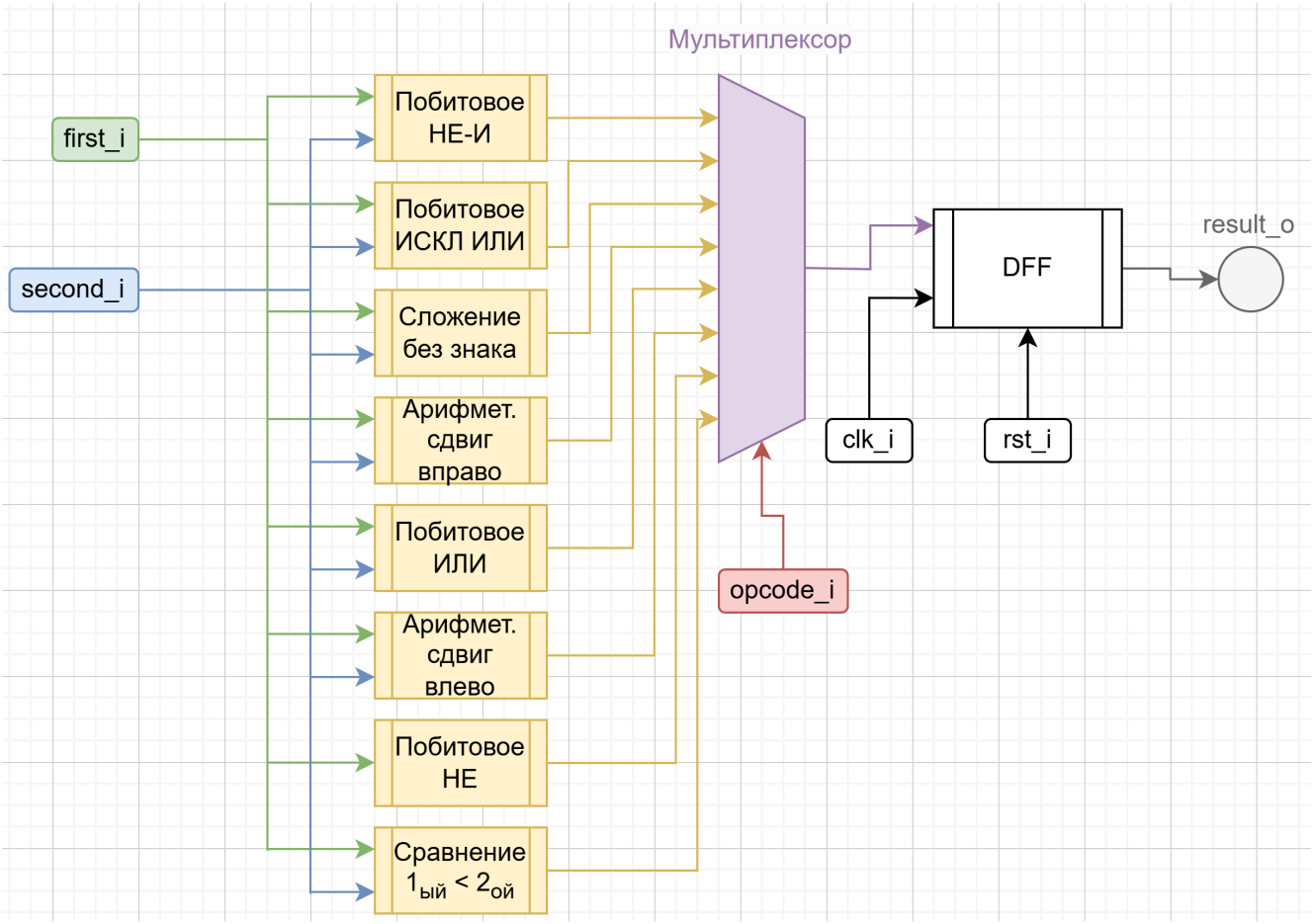
Помимо прочего, вот следующие требования:

- Модуль имеет параметр **WIDTH** для указания ширины операндов
- Модуль имеет следующие порты

Название	Ширина	Направление	Описание
clk_i	1	Input	Тактовый сигнал
rst_i	1	Input	Сигнал сброса
first_i	WIDTH	Input	Шина первого операнда
second_i	WIDTH	Input	Шина второго операнда
opcode_i	3	Input	Шина, кодирующая операцию
result_o	WIDTH	Output	Шина результата операции

- Модуль включает регистр (D-триггер) для сохранения результата операции: Результат операции, исполняемой в текущий такт, сохранен в регистре на следующий такт
    - Шина **result_o** отображает значение, записанное в регистре
    - Регистр сбрасываемый, значение при сбросе: 0
- Модуль тактируется сигналом **clk_i**, тактирование по положительному фронту
- Сброс синхронный, по активному высокому уровню сигнала **rst_i**
- Арифметические операции, в зависимости от значения **opcode_i**

То есть стоило бы переназвать элементы и добавить в принципиальную схему сигнал сброса и тактовый сигнал:



Такова принципиальная схема, она понятна и проста, теперь можно переходить к следующему этапу.

1.2 Verilog

1.2.1 Модули

Программа на Verilog, она же описание схемы, состоит из модулей (module), точнее из экземпляров модулей (module instances). Модуль можно представить как “черный ящик” с торчащими из него проводами — портами (ports). Порты бывают трех типов: входные (input), выходные (output) и двунаправленные (inout). В большинстве случаев используются первые два типа портов. Двунаправленные порты нужны для моделирования двунаправленных шин, на базе выходов с тремя состояниями и открытым стоком. Их мы рассматривать не будем.

Список портов описывается в заголовке модуля. К примеру, рассмотрим этот пустой модуль:

Learning > module.v > ...

```
1   module  blackbox            // module - ключевое слово, blackbox - имя модуля
2   #(
3                               //Здесь могут находиться параметры
4   )
5   (
6       input  a, b, c          // входные порты a,b,c
7       input  [7:0] bus,       // входной порт bus - 8-разрядная шина
8       output  [7:0] bus_out   // выходной порт bus_out, также 8-разрядный
9   );
10                              // Здесь добавляется тело модуля
11
12  endmodule                   // endmodule - конец описания модуля, ключевое слово
```

Рис. 1: \Learning\module.v

В теле модуля описывается его функциональность. Этот модуль пустой, его порты никуда не подключены.

1.2.2 Типы данных

В Verilog существуют два класса типов: типы для моделирования аппаратуры и стандартные арифметические типы данных, скопированные из языка Си. Рассмотрим первый класс, т.к. именно он используется для моделирования сигналов в схеме.

Сигнал может принимать 4 значения:

- 0 – логический ноль, или ложь
- 1 – логическая единица, или истина
- x – неопределенное значение. К примеру, значение регистра в начальный момент симуляции (до сброса или первой записи в регистр)
- z – состояние с высоким импедансом. Чаще всего сигнал принимает это значение, если он никуда не подключен – “обрыв провода”

В большинстве модулей на Verilog используются 2 основных типа данных – **wire** и **reg**. Из названия может показаться, что **wire** моделирует провод, а **reg** – регистр, но, как будет показано далее, это не совсем так. Хотя и в SystemVerilog есть универсальный тип logic, который может использоваться во всех случаях.

1.2.3 Wire

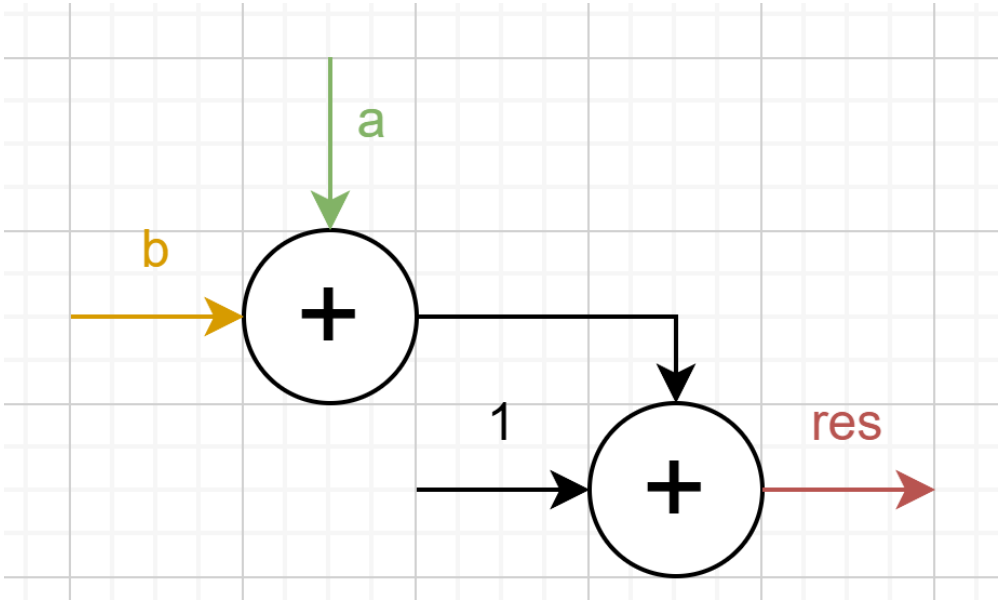
Тип wire служит для моделирования сигналов, которые не могут “хранить” состояние. К примеру, значение на выходе комбинационной схемы полностью определяется значениями на входах. Если значения на входе меняются, меняется и значение на выходе, т.е. состояние не хранится.

Тип **wire** используется вместе с операцией непрерывного присваивания — **assign**. При непрерывном присваивании всякий раз, когда меняется значение переменных в правой части присваивания, обновляется значение переменной в левой части. К примеру, простую комбинационную схему можно описать следующим образом:

Learning > combinatorix.v

```
1   module  combinatorix        // combinatorix - имя модуля
2   #(
3                               // Здесь могут находиться параметры
4   )
5   (
6       input wire [7:0] a,b,   // Входные порты a,b,
7
8       output [7:0] res        // Тип wire устанавливается по-умолчанию, но можно написать
9                               // output wire [7:0] res
10  );
11      assign res = a+b+1;     // Здесь добавляется тело модуля
12
13  endmodule                   // endmodule - конец описания модуля
```

Рис. 2: \Learning\combinatorix.v



Если сигнал не объявлен явно (например, через reg или wire), он автоматически считается wire. То есть действительно сущность **wire** можно принять за некоторый участок проводов.

В коде было использовано так называемое *непрерывное прививание* "$assign \ldots = \ldots$". Это присваивание работает непрерывно, если проводить аналогию, то можно это представить как накоротко соединенный провод. Значение сигнала обновляется мгновенно при изменении любого из входных сигналов в правой части выражения. Целью присваивания всегда является сигнал типа **wire**. Нельзя использовать **assign** для переменных типа **reg**.

assign используется для описания логических операций (AND, OR, сложение и т.д.), мультиплексоров, декодеров и других элементов без памяти.

Арифметические операции

Операция	Обозначение	Комментарий
Сложение	x + y	Знаковые или беззнаковые операнды
Умножение	x * y	Знаковые или беззнаковые операнды
Сравнение	x < y; x == y; x != y; x <= y; x > y; x >= y	Результат 1 бит
Сдвиг влево	x << y	Логический сдвиг x влево на y бит
Сдвиг вправо	x >> y	Логический сдвиг x вправо на y бит
Арифметический сдвиг вправо	x >>> y	Арифметический (знаковый) сдвиг x вправо на y бит

```
module signed_adder #(
  parameter WIDTH = 8
) (
  input  [WIDTH-1:0] first_i,
  input  [WIDTH-1:0] second_i,
  output [WIDTH-1:0] sum_o
);

  wire signed [WIDTH-1:0] first;
  wire signed [WIDTH-1:0] second;

  assign first = first_i;
  assign second = second_i;

  assign sum_o = first + second;

endmodule
```

Кстати о "[ ]" перед именем переменной. Квадратные скобки используются для указания разрядности (битовой ширины) сигнала. Это позволяет создавать многобитные переменные (шины), которые представляют собой группы из нескольких битов.

Скобки [MSB:LSB] задают диапазон битов, где:

- MSB (Most Significant Bit) — номер старшего бита,
- LSB (Least Significant Bit) — номер младшего бита.

Квадратные скобки также используются для доступа к битам ($signal[5]$) или срезам ($signal[3:0]$). Синтаксис примерно как в Python, интуитивен.

1.2.4 Regs

Тип **reg** может хранить значение и используется в процедурных блоках. Процедурный блок в Verilog – процедура, срабатывающая по определенному событию. К примеру, этим событием может быть фронт тактового сигнала или начало симуляции. В процедурных блоках могут использоваться Си-подобные управляющие конструкции:

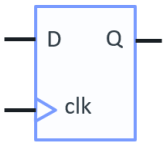
- if... else..
- for
- do... while..
- case

Строка `"always @(posedge clk)"` называется списком чувствительности. Она определяет события, по которым выполняется процедурный блок. Данный блок выполняется по каждому положительному фронту (positive edge) сигнала синхронизации.

Так же здесь можно заметить, что вместо “=” здесь используется “<=”, это называется *неблокирующее присваивание*. Это специальный тип присваивания, используемый для моделирования последовательной логики, применяются внутри блоков ‘‘`always`’’. Если снова проводить аналогию, то неблокирующее прививание можно представить как DFF, к которому подключен тактирующий сигнал `posedge clk`, то есть в моменты работы блока always значению слева единожды присваивается значение справа, и больше никаких ограничений не действует.

Следует обратить внимание, что порядок неблокирующих присваиваний не важен внутри процедурного блока. Рассмотрим тогда следующую ситуацию.

D	clk	Qt-1	Qt
D	0	Q	Q
D	1	Q	Q
D	1→0	Q	Q
D	**0→1**	**Q**	**D**




```
module dff #(
  parameter WIDTH = 8
)(
  input              clk_i,
  input  [WIDTH-1:0] d_i,
  output [WIDTH-1:0] q_o
);

  reg [WIDTH-1:0] d_ff;

  always @(posedge clk_i) begin
    d_ff <= d_i;
  end

  assign q_o = d_ff;

endmodule
```

23

Рис. 3: Картинка из презентации

Learning > overlay.v > ...

```
1   module  overlay
2   #(
3       |   |   |   |   |   |   |   //Здесь могут находиться параметры
4   )
5   (
6       input  a_i, b_i,        // входные порты
7       input  clk_i,           // входной порт - тактовый сигнал
8       output a_o, b_o         // выходные порты
9   );
10      reg a_reg, b_reg;
11      assign a_o = b_reg;
12
13      always @(posedge clk_i) begin // запуск блока при каждом положительном фронте тактового сигнала
14          a_reg <= a_i;
15          b_reg <= a_reg;
16      // и эти две стороки равносильны следующим двум строчкам:
17      //  b_reg <= a_reg;
18      //  a_reg <= a_i;
19      end
20
21  endmodule
```

Рис. 4: \Learning\overlay.v

Здесь я хочу показать, что неблокирующие присваивания можно менять местами. Помимо простых рассуждений, хочу добавить результат обработки программы **Quartus Prime**. Это приложение от **Intel**, которое поддерживает ведение проектов на Verilog, поэтому считаю, что следующий аргумент будет достаточно весомым в области того, как нужно воспринимать код Verilog.

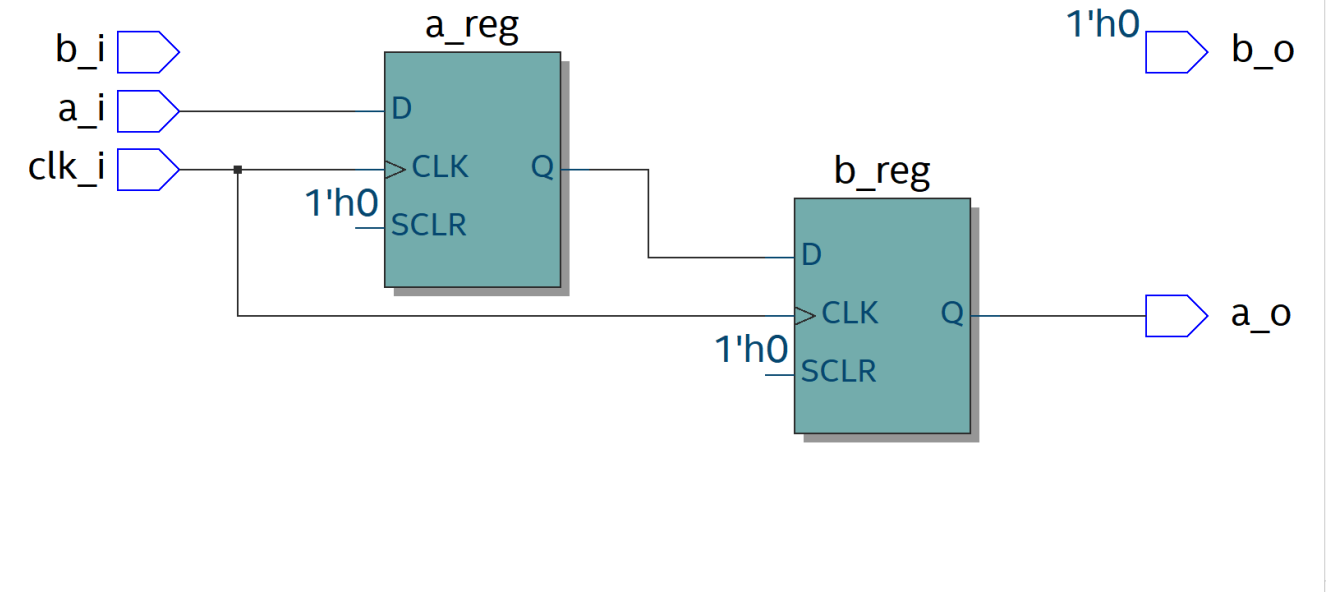


Рис. 5: Результат обработки файла \Learning\overlay.v

Действительно, как и было сказано ранее, неблокирующее присваивание “<=” стоит рассматривать как DFF.

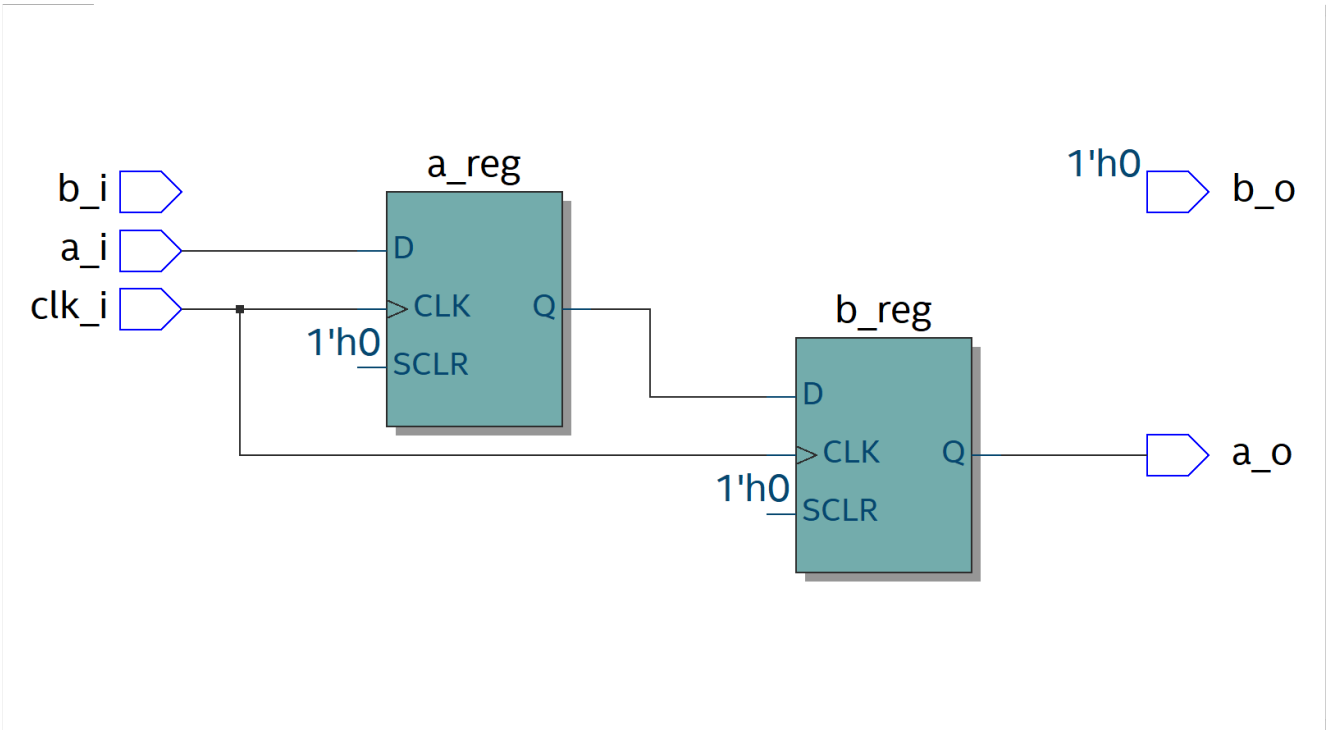


Рис. 6: А это уже результат обработки того же файла, если поменять строки.

Очень интересно. Несмотря на другой код, генерируется та же RTL схема. Хотя бы это уже говорит о том, что взаимное расположение строк не играет роли, но всё же, если генерируется некая схема, почему бы не разобраться в причине, почему же так выходит.

```
always @(posedge clk_i) begin
//    a_reg <= a_i;
//    b_reg <= a_reg;
// и эти две стороки равносиль
  b_reg <= a_reg;
  a_reg <= a_i;
end
```

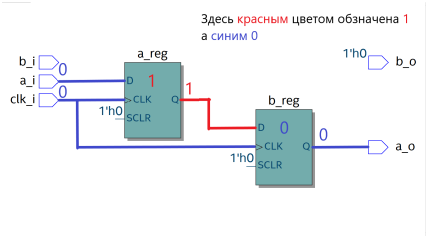


Рис. 7: Шаг 1

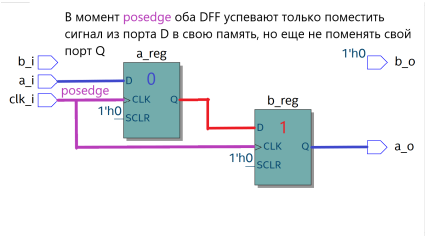


Рис. 8: Шаг 2

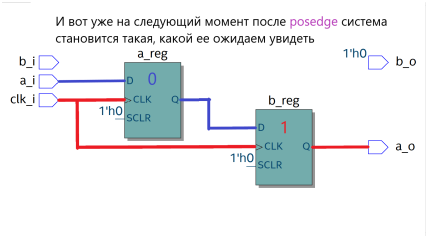


Рис. 9: Шаг 3

Считаю, на этом можно и остановиться в рассмотрении этих тем.

1.2.5 Testbench

Чтобы удостовериться в работоспособности написанного кода, следует его проверить. С этим помогает так называемый **testbench**.

Модули в Verilog можно сохранять, импортировать и использовать в других модулях. Таким образом, суть testbench заключается в том, чтобы проверяемый модуль полностью обернуть другим модулем, соединить соответствующие входные и выходные порты, и манипулируя временем и работой портов, проследить работу первого модуля. Оно же инстанцирование модуля.

Напишем тогда testbench для файла *overlay.h*, чтобы рассмотреть его работу под новым углом.

Learning > testbench.v > ...

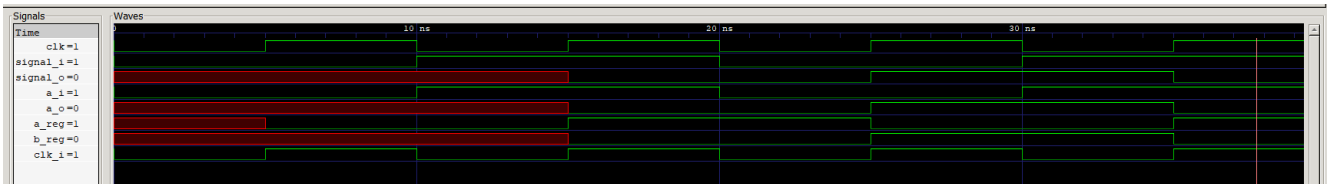
```
1   `timescale 1ns / 100ps
2
3   module testbench();
4       // Тактовый генератор
5       reg clk = 1'b0;
6       always begin
7           #5 clk = ~clk;  // Инверсия каждые 5 наносекунд
8       end
9       // Входной сигнал для подключения к тестируемому модулю
10      reg signal_i = 1'b0;
11      // Выходной сигнал для подключения к тестируемому модулю
12      wire signal_o;
13
14      // Экземпляр тестируемого модуля
15      overlay imported_module(
16  //синтаксис: module_name instance_name
17  /*module_name – Имя модуля которое даем модулю при его объявлении*/
18  /*instance_name – Имя экземпляра. Это уникальное имя, которое вы
19  даете конкретному экземпляру модуля, когда используете его в
20  другом модуле или тестбенче.*/
21          .clk_i(clk),
22          .a_i(signal_i),
23          .a_o(signal_o)
24  //синтаксис: .порт_модуля_который_тестируем(порт_который_описан_в_testbench)
25      );
26
27      // Блок инициализации
28      initial begin
29          $dumpvars;
30          #10;
31          signal_i <= ~signal_i;
32          #10;
33          signal_i <= ~signal_i;
34          #10;
35          signal_i <= ~signal_i;
36          #10;
37          $finish;                        // Завершение симуляции
38      end
39      /*Initial – это процедурный блок, который описывает поведение
40      в начале симуляции. Он позволяет инициализировать переменные и
41      задавать конкретные значения портам дизайна в начале моделирования.
42      Выполнение начального блока начинается в момент времени 0 в
43      симуляции. Он выполняется только один раз во время симуляции и
44      завершается, когда выполнены все содержащиеся в нём операторы
45      */
46  endmodule
```

Рис. 10: \Learning\testbench.v

Впрочем, в файле и на картинке описание достаточно подробное, в дополнительных опи-

саниях нужды не вижу.

Можно запустить при помощи **Makefile** и проверить работу.



Testbench работает как и предполагалось. Супер!

1.3 Сборка

Знаний, описанных выше, достаточно для сбора АЛУ. Можно и приступать.

Пошел генерал с внуком в цирк. До этого никогда в цирке не был. Там бегемот дрессированный по сцене бегает, клоуны валяются, поливаются из ведра, гимнасты на полотнах... Генерал сидит красный весь, прям закипает, а потом как заорет командным голосом:

Для описания каждого из 8-ми операций не будем создавать отдельно 8 модулей, так как эти операции уже описаны в стандартной библиотеке, да и код не охота раздувать.

alu.v > { } alu_register

```
1   module alu_register #(          //название модуля
2       parameter WIDTH = 8         //требуемый параметр параметр шины по ТЗ
3   ) (
4       input wire clk_i,           //переменная тактирования
5       input wire rst_i,           //переменная синхронного сброса
6       input wire [WIDTH-1:0] first_i,     //первый операнд
7       input wire [WIDTH-1:0] second_i,    //второй операнд
8       input wire [2:0] opcode_i,          //вход инструкции
9       output wire [WIDTH-1:0] result_o    //результат работы АЛУ
10  );
```

Здесь описали входные и выходные порты, которые были описаны.

```
12  reg [WIDTH-1:0] result_reg; //промежуточные переменные типа reg, чтобы
13  reg [WIDTH-1:0] alu_result; //можно было работать с ними в always блоке
14
15  always @(*) begin
16      case(opcode_i)
17          3'b000: alu_result <= ~(first_i & second_i);
18          3'b001: alu_result <= first_i ^ second_i;
19          3'b010: alu_result <= first_i + second_i;
20          3'b011: alu_result <= $signed(first_i) >>> second_i;
21          3'b100: alu_result <= first_i | second_i;
22          3'b101: alu_result <= first_i << second_i;
23          3'b110: alu_result <= ~first_i;
24          3'b111: alu_result <= first_i < second_i;
25          default: alu_result <= {WIDTH{1'b0}};
26          //Конкатенация {}: Объединяет битовые последовательности.
27          //{WIDTH{1'b0}}: Создает вектор из WIDTH нулевых бит.
28      endcase
29  end
```

Структура **case** подобна структурам **switch** в других языках программирования, считаю что понимание этого абзаца интуитивно.

Стандартом языка предусмотрены специальные системные функции $ signed() и $ unsigned(). Эти функции по своей сути являются директивами компилятору, они говорят ему, как он должен интерпретировать выражение. При этом разрядность входного выражения не меняется.

$signed – возвращаемое значение знаковое, $unsigned – возвращаемое значение беззнаковое.

```
31  always @(posedge clk_i) begin
32      if (rst_i)
33          result_reg <= {WIDTH{1'b0}};
34      else
35          result_reg <= alu_result;
36  end
37
38  assign result_o = result_reg;
39
40  endmodule
```

Здесь прописываем работу тактового сигнала и сигнала сброса. По ТЗ сказано реализовать синхронный сброс, а значит, приоритет проверки сигнала тактирования выше, и только внутри этого цикла можно установить ветвление. И в конце концов выдаем сигнал регистра на выход АЛУ.

Замечательно. Посмотрим, как его нарисует Quartus.

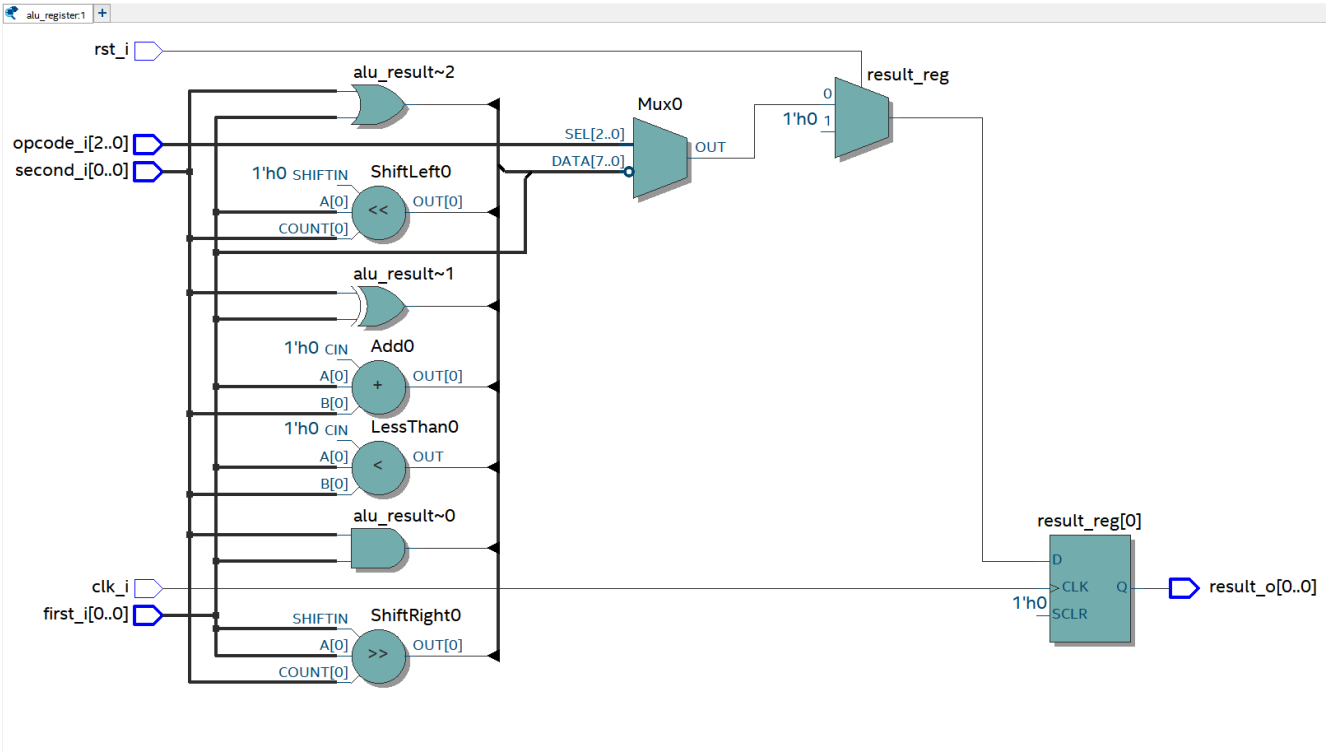
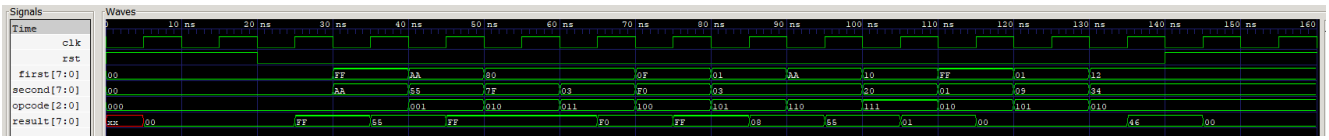


Рис. 11: Схема при WIDTH = 1

Фантастика. Схема соответствует ожиданиям и схожа с принципиальной схемой на стр.2.

Теперь можно написать testbench для проверки модуля, как того требует ТЗ. Ничего интересного в этом коде нет, просто поочередно проверяем функциональность каждого элемента.



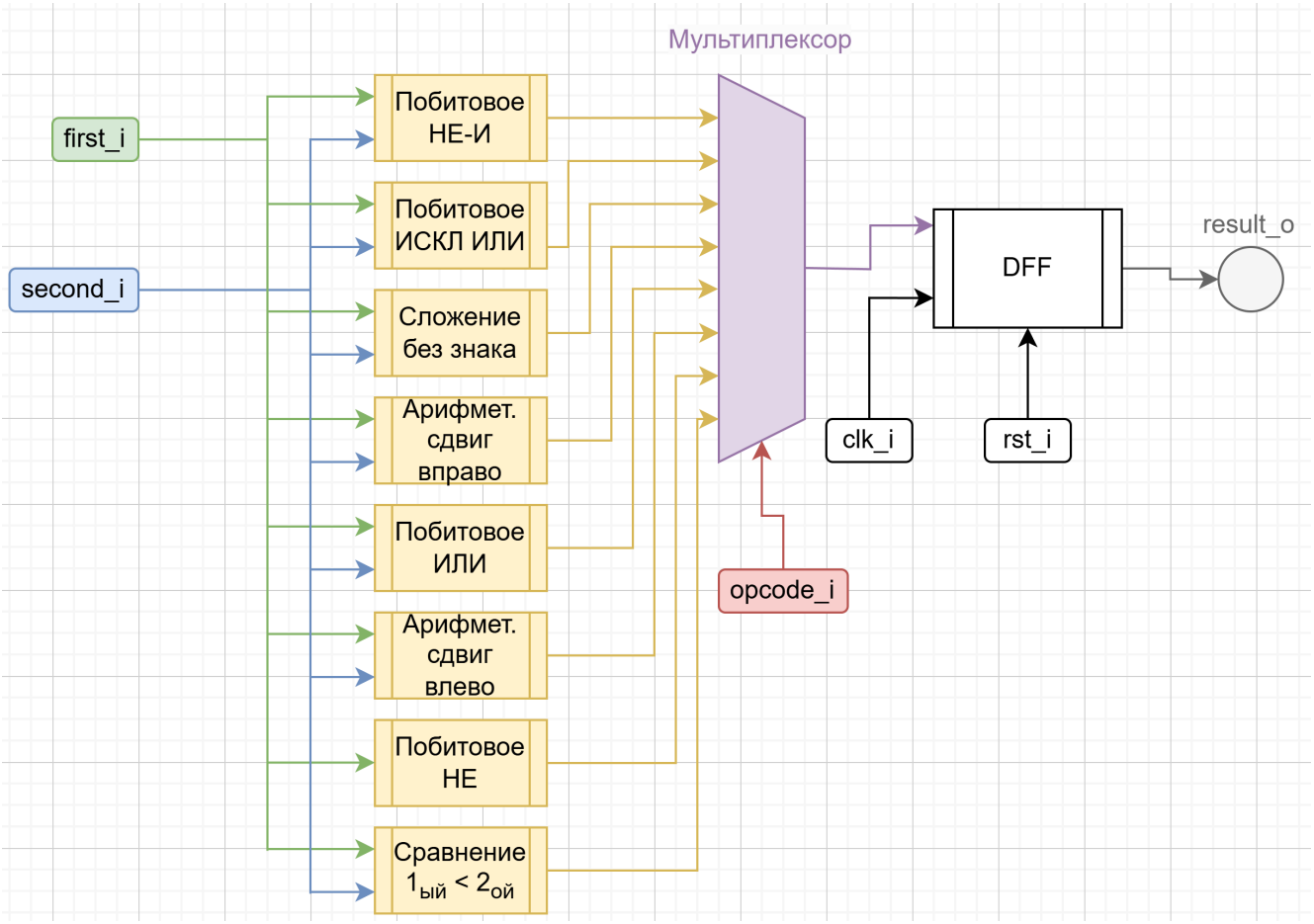
Все системы работают корректно. Результат появляется только по положительному фронту тактирующего сигнала, как и требуется в ТЗ. Сигнал сброса обнуляет выходную шину, но только по фронту тактирующего сигнала, как и требуется в ТЗ. Каждая логическая и арифметическая операция соответствует своему коду и выполняет свои функции, как и требуется в ТЗ.

Когда все аспекты рассмотрены, можно начать написание "формального отчета".

2 Формальный отчет

1. Выполнил: Комиссаров Данил Андреевич.
2. Студент группы Б01-304.
3. Первый вариант задания. Арифметико-логическое устройство (ALU).
4. Контакты: komissarov.da@phystech.edu
5. Модуль alu_register выполняет арифметические и логические операции над двумя входными операндами (first_i, second_i) в соответствии с кодом операции (opcode_i). Результат операции сохраняется в регистр и выводится на порт result_o на следующий такт.

Модуль поддерживает синхронный сброс, обнуляющий регистр при активации сигнала rst_i.



6.

7. Параметр WIDTH
   Описание Разрядность операндов
   Допустимые значения Целое число больше или равно 1

Название	**Ширина**	**Направление**	**Описание**
clk_i	1	Вход	Тактовый сигнал
rst_i	1	Вход	Синхронный сброс (активный уровень — 1)
first_i	WIDTH	Вход	Первый операнд
second_i	WIDTH	Вход	Второй операнд
opcode_i	3	Вход	Код операции
result_o	WIDTH	Выход	Результат операции

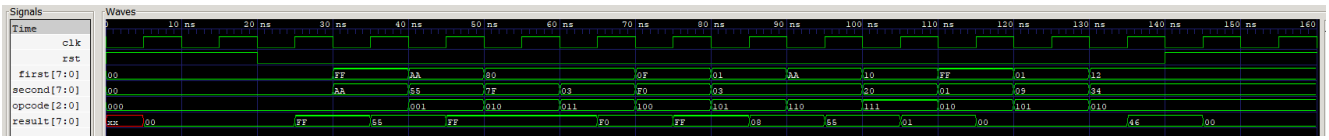
8.

9. Тактирование и сброс
   Тактирование: По положительному фронту сигнала clk_i.
   Сброс: Синхронный (активный уровень — 1).
   При активации rst_i регистр обнуляется на следующем такте.

10. Тестирование Сценарий тестирования:

    1. Сброс: Проверка обнуления регистра при активации rst_i.
       Проверка операций:
    2. Побитовое НЕ-И (opcode_i = 000).
    3. Исключающее ИЛИ (001).
    4. Сложение (010).
    5. Арифметический сдвиг вправо (011).
    6. Побитовое ИЛИ (100).
    7. Логический сдвиг влево (101).
    8. Побитовое НЕ (110).
    9. Сравнение (111).
    10. Граничные случаи:
    11. Переполнение при сложении.
    12. Сдвиг на значение, превышающее разрядность.
    13. Задержка вывода: Убедиться, что результат появляется через такт.
    14. Сброс: Снова обнуление регистра.

11. В папке будет находиться Makefile, чтобы запустить все файлы в папке, введите в консоль *make run*, чтобы отчистить папку от результатов компиляции, введите *make clean*.



12.

На этом пожалуй можно и закончить. Наверное я уделил слишком много внимания несущественным аспектам, в то время, когда некоторые фундаментальные принципы были упущены, но это было интересно.