МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт кибербезопасности и цифровых технологий
Кафедра КБ-6 «Приборы и информационно-измерительные системы»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

по дисциплине Разработка устройств на базе ПЛИС

Студента _____ А. Д. Глухов
дата, подпись — инициалы и фамилия

Группа БПБО-02-20 шифр

Работа защищена _____

Руководитель работы _____ В.П. Орлов
дата, подпись — инициалы и фамилия

МОСКВА 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Важнейший компонент любого персонального компьютера – это микропроцессор, который управляет работой компьютера и выполняет большую часть обработки информации.

В микропроцессорах нашли отражение высокие научно-технические достижения в области физики твердого тела, кристаллографии, радиотехники и электроники, математики и автоматизации, кибернетики и электроники В современном мире трудно найти область техники, где не применялись бы микропроцессоры. Важнейшими из них являются: автоматизация электротехнического оборудования, управление производством, физическое и математическое моделирование, обработка результатов экспериментов, управление приборами и искусственными органами в медицине, обеспечение безопасности дорожного движения.

Целью лабораторной работы является создание однотактного микропроцессора архитектуры MIPS, используя язык описания аппаратуры Verilog. Для создания микропроцессора использовалась система автоматизированного проектирования Vivado компании Xilinx.

1 ОБЩАЯ ИНФОРМАЦИЯ О ЯДРЕ MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) — микропроцессор, разработанный компанией MIPS Computer Systems (в настоящее время MIPS Technologies) в соответствии с концепцией проектирования процессоров RISC (сокращенный набор команд) и впервые реализованный 1985 году. Существует большое количество модификаций этой архитектуры, созданных специально для 3D-моделирования, быстрой обработки чисел с плавающей запятой, многопотоковых вычислений. Различные варианты этих процессоров используется в роутерах, смартфонах, планшетах и игровых консолях.

1.1 НАБОР РЕГИСТРОВ

В архитектуре MIPS определено 32 регистра общего назначения. У каждого регистра есть имя и порядковый номер от 0 до 31.

Название	Номер	Назначение
`$0`	0	Константный нуль
`$at`	1	Временный регистр для нужд ассемблера
`$v0-$v1`	2–3	Возвращаемые функциями значения
`$a0-$a3`	4–7	Аргументы функций
`$t0-$t7`	8–15	Временные переменные
`$s0-$s7`	16–23	Сохраняемые переменные
`$t8-$t9`	24–25	Временные переменные
`$k0-$k1`	26–27	Временные переменные операционной системы (ОС)
`$gp`	28	Глобальный указатель (англ.: global pointer)
`$sp`	29	Указатель стека (англ.: stack pointer)
`$fp`	30	Указатель кадра стека (англ.: frame pointer)
`$ra`	31	Регистр адреса возврата из функции

MIPS обычно хранит переменные в 18 из 32 регистров: **`$s0-$s7`** и **`$t0-$t9`**. Регистры, имена которых начинаются на **`$s`**, называют сохраняемыми (saved) регистрами. В соответствии с соглашением об использовании регистров MIPS эти регистры используются для размещения в них переменных. Сохраняемые регистры имеют особое значение в контексте вызова процедур. Регистры, имена которых

начинаются с **$t**, называют временными (temporary) регистрами. Они используются для хранения временных переменных.

В рамках данной работы будем использовать регистры **$0, $t0-$t7, $s0-$s7**. Двоичные индексы некоторых нужных нам регистров (5 бит на индекс).

Имя	bin	dec	hex
$0	00000	0	0
$t0	01000	8	8
$t1	01001	9	9
$t2	01010	10	A
$t3	01011	11	B
$t4	01100	12	C
$t5	01101	13	D
$t6	01110	14	E
$t7	01111	15	F
$s0	10000	16	10
$s1	10001	17	11
$s2	10010	18	12
$s3	10011	19	13
$s4	10100	20	14
$s5	10101	21	15
$s6	10110	22	16
$s7	10111	23	17

1.2 АРХИТЕКТУРА И МИКРОАХИТЕКТУРА ПРОЦЕССОРА MIPS

Архитектура процессора определяется двумя вещами:

1. Набор команд ассемблера;
2. Архитектурным состоянием.

Архитектурное состояние процессора MIPS определяется содержимым ***счетчика команд*** и 32 видимых программисту регистров, поэтому любой процессор, реализующий архитектуру MIPS, вне зависимости от его микроархитектуры обязан

иметь счетчик команд и ровно 32 регистра. Зная текущее архитектурное состояние, процессор точно знает, какую операцию над какими данными надо выполнить для получения нового архитектурного состояния.

Взаимное расположение регистров, памяти, АЛУ и других строительных блоков, из которых состоит микропроцессор, называют ***микроархитектурой.*** У каждой архитектуры, включая MIPS, может быть много различных микроархитектур, обеспечивающих разное соотношение производительности, цены и сложности. Все они смогут выполнять одни и те же программы, но их внутреннее устройство ***может очень сильно отличаться.***

1.3 СЧЕТЧИК ПРОГРАММЫ

Счетчик программы **(*program counter pc*)** - область памяти 32 бит - содержит адрес команды, которую выполняет процессор на текущий такт. Адрес команды - это адрес 1-го байта команды в памяти инструкций. Т.к. каждая команда занимает 4 байта, корректными значениями счетчика программы могут быть только адреса, которые кратны 4-м - в примере выше, это 0, 4, 8, C (в шестнадцатеричной записи).

1.4 ПАМЯТЬ ИНСТРУКЦИЙ

Память инструкций **(*instruction memory*)** - адресуемая область памяти, хранящая последовательность команд, которые одну за одной выполняет процессор. Адресуемая - это значит, что каждый байт в этой области имеет свой порядковый номер, т.е. адрес - для 32хбитной архитектуры адрес имеет длину 32 бита (4 байта), т.е. память инструкций теоретически может содержать максимально 2^{32} байт. Каждая команда представляет из себя машинный код длиной 32 бита (4 байта - *слово* - *word*) для 32-х битного процессора - это значит, что память инструкций может содержать $2^{32}/4=2^{30}$ команд.

Например, следующая программа на ассемблере MIPS

```
lw $s0, 0 ($0)
```

```
lw $s1, 4 ($0)
add $s2, $s0, $s1
sw $s2, 8 ($0)
```

будет иметь такой вид в памяти инструкций:

```
0x00000000: 10001100 00001001 00000000 00000000
0x00000004: 10001100 00001010 00000000 00000100
0x00000008: 00000001 00101010 01011000 00100000
0x0000000C: 10101100 00001011 00000000 00001000
```

При старте работы процессор сначала выдернет из памяти инструкций первые 4 байта (1-е слово) и выполнит их как команду, потом выдернет 2-е 4 байта (2-е слово) и выполнит их как команду и т.п.

1.5 ПАМЯТЬ ДАННЫХ

Память данных **(*data memory*)** - адресуемая область памяти, хранящая произвольные данные, которыми может оперировать процессор во время выполнения программы. Аналогично памяти инструкций, длина адреса для 32хбитной архитектуры составляет 32 бита, т.е. программа может адресовать максимум 2^32 байт. Байты также логически организованы в слова по 4 байта на слово (для 32хбитной архитектуры).

2 ПОНЯТИЕ ЯЗЫКА АССЕМБЛЕРА И КОНВЕРТАЦИЯ ЕГО В МАШИННЫЙ КОД ПРОЦЕССОРА

Язык ассемблер - это самый низкоуровневый язык программирования - по сути это просто представленный в удобном для чтения и понимания человеком виде машинный код.

Для каждого процессора язык ассемблера свой - подобрать набор команд так, чтобы все их многообразие можно было однозначно закодировать всеми возможными комбинациями значений доступных 32х бит (для 32хбитного процессора) на код машинной команды, и чтобы при этом при помощи этого набора можно было эффективно выполнять программы, транслируемые в ассемблер компиляторами высокоуровневых языков программирования, - задача архитекторов процессора.

Набор и структура команд, которые рассмотрены ниже, выбраны и реализованы таким образом потому, что так посчитали нужным инженеры и архитекторы MIPS Technologies - команды, создававшие архитектуры других процессоров, решили эту же задачу другим образом.

На данной лабораторной работе мы реализавали некоторых команд 32хбитного процессора MIPS: **`add`** (сложение), **`sub`** (вычитание), **`lw`** (загрузка значения из памяти данных), **`sw`** (сохранение значения в память данных), **`addi`** (сложение с константой), **`beq`** (условный переход), **`j`** (безусловный переход) - ниже каждая из них будет разобрана в деталях.

2.1 КОМАНДЫ ТИПА R-TYPE (REGISTER TYPE)

Команды типа R-type имеют 3 операнда, все операнды - адреса регистров, 2 регистра-источника, 1 регистр – назначение.

В 32хбитном процессоре на каждую команду отведено ровно 32 бита. Команды типа R-type разбиты на следующие поля по битам:

op(6 бит)	rs(5 бит)	rt(5 бит)	rd(5 бит)	shamt(5 бит)	funct(6 бит)

rs, rt, rd - операнды - адреса регистров (*'r'* везде от *register*)

op (*opcode*)	**0** для всех операций R-type
funct (*function*)	для **add=32**, для **sub=34**
rs, rt	источник1, источник2 (обозначение выбрано как s - source, а 't' просто идет в алфавите после 's')
rd	назначение (*destination*)
shamt	сдвиг (*ammount of shift*) = 0 для всех операций R-type

2.1.1 КОМАНДА ADD (СЛОЖИТЬ)

Команда **add** складывает значения 2х регистров и кладет результат в 3й регистр. Синтаксис на ассемблере:

```
add rd, rs, rt
```

rd, rs, rt - адреса (имена) регистров - команда **add** должна взять значения из регистров **rs** и **rt**, вычислить сумму **rs+rt** и записать результат в регистр **rd**, например:

```
add $s0, $s1, $s2
```

op(6 бит)	rs(5 бит)	rt(5 бит)	rd(5 бит)	shamt(5 бит)	funct(6 бит)
0	17 ($s1)	18 ($s2)	16 ($s0)	0	32 (add)
000000	10001	10010	10000	00000	10 0000

на практике должно быть выполнено как **$s0=$s1+$s2** - вычислить сумму значений, хранящихся в регистрах **$s1** и **$s2**, и записать результат в регистр **$s0**.

2.1.2 КОМАНДА SUB (ВЫЧЕСТЬ)

Команда **sub** очевидно вычисляет разность значений 2х регистров и кладет результат в 3й регистр. Синтаксис на ассемблере:

```
sub rd, rs, rt
```

rd, rs, rt - адреса (имена) регистров - команда **sub** должна взять значения из регистров **rs** и **rt**, вычислить разность **rs-rt** и записать результат в регистр **rd**, например:

```
sub $s0, $s1, $s2
```

op(6 бит)	rs(5 бит)	rt(5 бит)	rd(5 бит)	shamt(5 бит)	funct(6 бит)
0	17 ($s1)	18 ($s2)	16 ($s0)	0	34 (sub)
000000	10001	10010	10000	00000	10 0010

на практике должно быть выполнено как **$s0=$s1-$s2** - вычислить разность значений регистров **$s1** и **$s2** и записать результат в регистр **$s0**.

2.2 КОМАНДЫ ТИПА I-TYPE (IMMEDIATE TYPE)

Команды типа I-type в отличии от команд типа R-type умеют работать с числовыми константами (*immediates* - надо понимать что-то типа "незамедлительные" значения, т.е. их не нужно ниоткуда получать), которые встроены прямо в код команды.

op (6 бит)	rs (5 бит)	rt (5 бит)	imm (16 бит)

op	код операции: для **lw=35**, для **sw=43**, для **addi=8**, для **beq=4**
rs, rt	адреса регистров-операндов - смысловые роли могут быть разные для разных команд

imm (immediate)	значение операнда-константы

2.2.1 КОМАНДА LW (ЗАГРУЗИТЬ СЛОВО)

Команда **lw** загружает значение из памяти данных в регистр. Синтаксис на ассемблере:

```
lw rt, imm (rs)
```

rt – адрес регистра-назначения, **imm** - константа - адрес загружаемого значения в памяти данных, **rs** - адрес регистра, содержащего значение сдвига для адреса загрузки, например:

```
lw $s0, 4 ($0)
```

сначала вычислит адрес загружаемого значения как значение **imm** + значение, содержащееся в регистре **rs (4+0=4)**, затем считает значение по вычисленному адресу (4) из памяти данных и запишет его в регистр **rt ($s0).**

op (6 бит)	**rs (5 бит)**	**rt (5 бит)**	**imm (16 бит)**
35	**0 ($0)**	**16 ($s0)**	**4**
100011	**00000**	**10000**	**0000000000000100**

2.2.2 КОМАНДА SW (СОХРАНИТЬ СЛОВО)

Команда **sw** записывает (сохраняет) значение из регистра в память данных, синтаксис на ассемблере:

```
sw rt, imm (rs)
```

rt - адрес регистра-источника, **imm** - константа - адрес сохранения значения в памяти данных, **rs** - адрес регистра, содержащего значение сдвига для адреса сохранения, например:

sw $s0, 4 ($0)

сначала вычислит адрес сохранения значения как значение **imm** + значение, содержащееся в регистре **rs** (4+0=4), затем считает значение из регистра **rt ($s0)** и сохранит его в память данных по вычисленному адресу (4).

op (6 бит)	rs (5 бит)	rt (5 бит)	imm (16 бит)
43	0 ($0)	16 ($s0)	4
101011	00000	10000	0000000000000100

2.2.3 КОМАНДА ADDI (СЛОЖИТЬ С КОНСТАНТОЙ)

Команда addi складывает значение регистра с константой и записывает результат в регистр, синтаксис на ассемблере:

addi rt, rs, imm

rt - адрес регистра-назначения, **rs** - адрес регистра, содержащего 1е складываемое значение, **imm** - константа - 2е складываемое значение.

Команда **addi** должна взять значения из регистра rs, вычислить сумму rs+imm и записать результат в регистр **rt**, например:

addi $s0, $s1, 4

на практике должно быть выполнено как **$s0=$s1+4** - вычислить сумму значения, хранящегося в регистре **$s1** со значением константы 4, и записать результат в регистр **$s0**.

op (6 бит)	rs (5 бит)	rt (5 бит)	imm (16 бит)
8	0 ($0)	16 ($s0)	4
001000	00000	10000	0000000000000100

2.2.4 КОМАНДА BEQ (ОТВЕТВИТЬСЯ, ЕСЛИ РАВНО)

Команда beq осуществляет условный переход - перевод счетчика программы в указанное место при выполнении определенного условия.

При помощи команды условного перехода **beq** можно организовывать циклы и блоки проверки условий, синтаксис на ассемблере:

```
beq rs, rt, imm
```

rs, rt - адреса регистров, значения которых сравнивает команда, **imm** - адрес перехода относительно адреса текущей инструкции (в ассемблерной программе задается при помощи специальных меток).

Адрес перехода задается при помощи специальных меток - символьных имен, которые можно назначить на любую строку в ассемблерной программе - транслятор в машинный код сам вычислит нужный адрес перехода, например:

```
label: addi $s0, $0, 4
beq $s0, $s1, label
```

сравнит значения регистров **$s0** и **$s1** и передаст управление инструкции, адрес которой в памяти инструкций равен адресу строки, помеченной меткой "**label**" (те в данном примере это одна строка выше), если значения равны; если значения не равны, то программа продолжит последовательное выполнение со следующей строчки.

2.3 КОМАНДЫ ТИПА J-TYPE (JUMP TYPE)

Команда **j** осуществляет безусловный переход - перевод счетчика программы в указанное место.

Принцип работы полностью аналогичен команде условного перехода **beq**, только переход осуществляется всегда и без всяких условий. Также диапазон перехода относительно текущего положения чуть больше, чем у команды **beq**, т.к. на адрес перехода внутри команды отведено 26 бит, а не 16, синтаксис на ассемблере:

```
j addr
```

addr - адрес перехода относительно адреса текущей инструкции (в ассемблерной программе задается при помощи метки), например:

```
label: addi $s0, $0, 4
j label
```

вернет управление на одну строку выше (в данном случае получим бесконечный цикл).

op (6 бит)	addr (26 бит)
2	4
000010	00000000000000000000000100

3 ОСНОВНЫЕ МОДУЛИ ДИЗАЙНА HDL

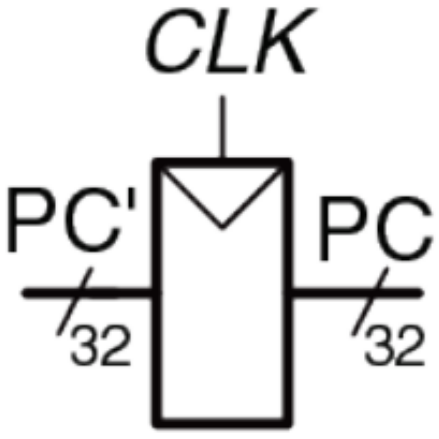
Основные элементы, которые определяют текущее состояние системы - счетчик программы (program counter), файл регистров (register file), память инструкций (instruction memory) и память данных (data memory) - они и станут основными блоками дизайна - определим их в виде соответствующих модулей.

Каждая команда в момент выполнения будет влиять на один или несколько перечисленных модулей одновременно - собственно в том, каким образом команда модифицирует счетчик программы, файл регистров и память данных, и заключается ее основной смысл.

3.1 СЧЕТЧИК ПРОГРАММЫ (PROGRAM COUNTER)

Модуль счетчик программы (*program counter*) позволяет установить следующее значение счетчика программы на каждый такт синхросигнала.

На входе – 1 бит на тактовый сигнал clk (Clock), 1 бит на сигнал reset и 32хбитное значение для следующего значения счетчика программы pc' (program counter next - pc_next). На выходе - текущее значение счетчика программы pc (program counter). Значение на выходе pc меняется на значение входа pc' или на 0, если на reset подается сигал, на каждый такт сигнала clk.



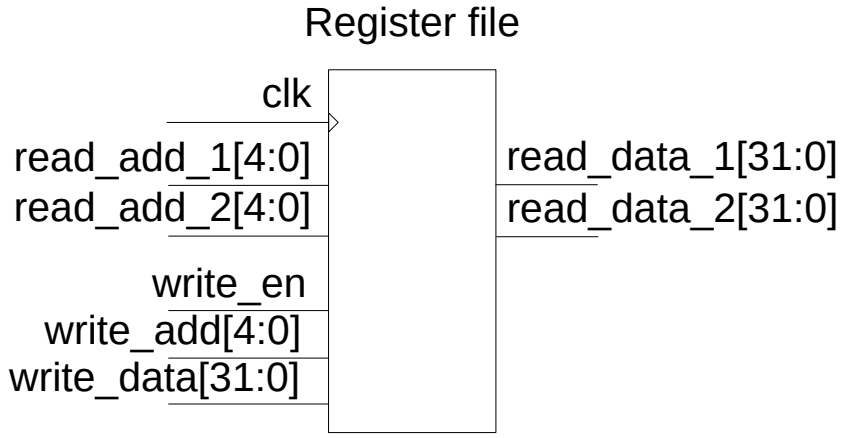
Файл program_counter.v:

```
module pc(
	input wire clk,
	input wire reset,
	input wire [31:0] pc_next,
	output reg [31:0] pc
	);

	always @(posedge clk ) begin // при каждом фронте сигнала clk
		if(reset)	pc <= 0;
		else		pc <= pc_next;
	end

endmodule
```

3.2 ФАЙЛ РЕГИСТРОВ (REGISTER FILE)

Модуль файл регистров (register file) хранит значения внутренних регистров процессора, позволяет получать 32х битное значение регистра по 5ти битному адресу и делать запись 32х битного значения в регистр по 5ти битному адресу.

Модуль устроен так, что читать можно два значения одновременно. Запись делается в один регистр на один такт сигнала Clock.



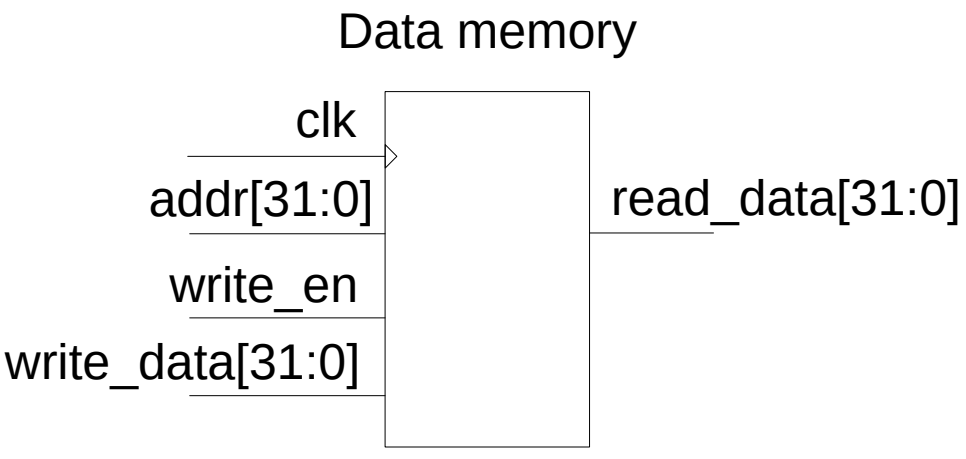
Файл register_file.v:

```
module register_file(
	input clk,
	input [4:0] raddr1,		//A1// адрес регистра для чтения
	input [4:0] raddr2,		//A2// адрес регистра для чтения
	input [4:0] waddr,		//A3// адрес регистра для записи

	input we,			//WE3//сигнал управления
```

```
        input [31:0] wdata,         //WD3//запись в регистр waddr

        output [31:0] rdata1,       // RD1// чтение из регистра raddr1
        output [31:0] rdata2        // RD2// чтение из регистра raddr2

        );

        reg [31:0] regs[31:0];

        always @(posedge clk) begin
                if(we) begin
                        regs[waddr] <= wdata;
                end
        end

        assign rdata1 = raddr1? regs[raddr1]:0;
        assign rdata2 = raddr2? regs[raddr2]:0;
endmodule
```

3.3 ПАМЯТЬ ДАННЫХ (DATA MEMORY)

Модуль память данных (data memory) - RAM (random access memory) - запоминающее устройство с произвольным доступом на чтение и запись. Адресуем память 32х битным указателем 2^32 байт=4096 Мегабайт. Память организована словами (блоками) по 4 байта, загрузка и сохранение осуществляется также словами по 4 байта. С 32х битных адресом получаем 2^32 / 4 = 1'073'741'824 4хбайтовых слов виртуальной памяти в доступном адресном пространстве.

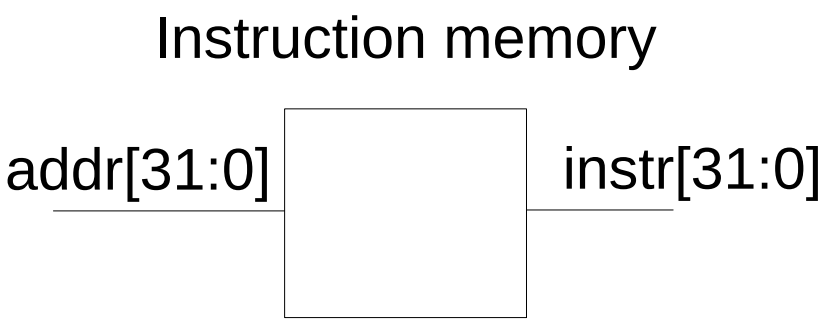


Файл data_memory:

```
module mem #(
        SIZE = 1024
    )(
        input wire clk,
        input wire [31:0] addr,
        input wire we,
        input wire [31:0] data,
        output wire[31:0] out
    );

    reg[31:0] memory [SIZE-1: 0];

    always@(posedge clk) begin
        if(we) begin
            memory[addr] <= data; //'<=' = '='
        end
    end

    assign out = memory[addr];
endmodule
```

3.4 ПАМЯТЬ ИНСТРУКЦИЙ (INSTRUCTION MEMORY)

Модуль память инструкций (instruction memory) - ROM (read-only memory) - память, доступная только для чтения. По логике адресации полная аналогия с памятью данных - 32х битный адрес, выравнивание по 4 слова. Главное отличие от модуля памяти данных - отсутствие интерфейса записи.



Код программы будет представлен в виде двоичных констант с заранее определенными адресами: одна инструкция - одно 4х битное слово.

Например, для такой ассемблерной программы:

```
addi $s0, $0, 96     // Поместить в регистр 96
sw $s0, 0xff ($0)   // Вывести значение регистра на
                    // светодиоды
addi $s1, $0, 255   // Поместить в регистр 255
sw $s1, 0xff ($0)   // Вывести значение регистра на
                    // светодиоды
```

Файл instruction_memory:

```
module im
        #(      parameter SIZE = 128          )

        (
        input wire [31:0] addr,
        output reg [31:0] cmd
        );


        always@(addr) begin

        case (addr)
              // addi $s0, $0, 96
              32'h00000000: cmd <= 32'b001000_00000_10000_00000000_1001_0110;
                                  // op(6 бит), rs(5 бит), rt(5 бит), imm(16 бит)
                                  //          rs=$0,     rt=$s0,    imm=96
              // sw $s0, 0xff ($0)
              32'h00000004: cmd <= 32'b101011_00000_10000_0000_0000_1111_1111;
                                  // op(6 бит), rs(5 бит), rt(5 бит), imm(16 бит)
                                  //          rs=$0,     rt=$s0,    imm=0xff
              // addi $s1, $0, 255
              32'h00000008: cmd <= 32'b001000_00000_10001_00000000_1111_1111;
                                  // op(6 бит), rs(5 бит), rt(5 бит), imm(16 бит)
```

```
                        //          rs=$0,     rt=$s1,    imm=255
                // sw $s1, 0xff ($0)
                32'h0000000C: cmd <= 32'b101011_00000_10001_0000_0000_1111_1111;
                        // op(6 бит), rs(5 бит), rt(5 бит), imm(16 бит)
                        //          rs=$0,     rt=$s1,    imm=0xff
                default: cmd <= 0;
            endcase // case (cmd)
        end
endmodule
```

3.5 КОНТРОЛЛЕР (CONTROLLER)

Контроллер (controller) - Устройство управления формирует управляющие сигналы на основе сигналов на входах opcode, funct. Для упрощения разработки мы поделим устройство управления на две части. Основной дешифратор вычисляет значение большинства выходов на основе поля opcode. Он также формирует двухбитный сигнал ALUOp. Дешифратор АЛУ (ALU decoder) использует ALUOp совместно с полем funct для вычисления состояния ALUControl.

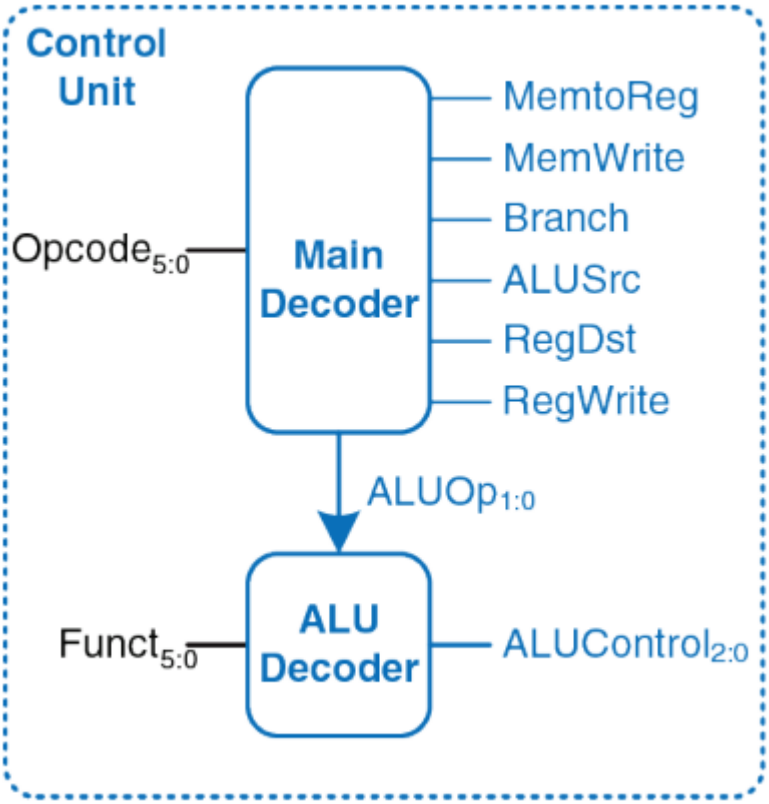


Таблица истинности для основного дешифратора.

Команда	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
Команды типа R	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

Файл controller:

```
`include "main_decoder.v"
`include "alu_decoder.v"

module controller(
input wire [5:0]    opcode,
input wire [5:0]    funct,
output wire         wemem,
output wire         werf,
output wire         rfwasrc,
output wire         memToRf,
output wire         j,
output wire         PCSrc,
output wire         aluSrc,
output wire [3:0]   aluControl
output wire         zero,
output wire         overflow,
);
wire[1:0] aluop;
wire branch;
main_decoder md(.opcode(opcode), .wemem(wemem), .werf(werf), .branch(branch), .rfwasrc(rfwasrc), .memToRf(memToRf), .aluSrc(aluSrc), .aluop(aluop), .j(j));
alu_decoder ad(.funct(funct), .aluop(aluop), .aluControl(aluControl));
assign PCSrc = zero & branch;
endmodule
```

Файл main_decoder.v:

```
module main_decoder(
	input wire [5:0] opcode,

	output reg wemem,
	output reg werf,
	output reg branch,
	output reg rfwasrc,
	output reg memToRf,
	output reg j,
	output reg aluSrc,
	output reg[1:0] aluop,
);
	always @* begin
		case(opcode)
			//lw
			6'b100_011: begin
				werf    <= 1;
				rfwasrc <= 0;
				aluSrc <= 1;
				branch		<= 0;
				wemem		<= 0;
				memToRf <= 1;
				aluop   <= 2'b00;
				j		<= 0;
			end //lw

			//sw
			6'b101_011: begin
				werf    <= 0;
				rfwasrc <= 0;
				aluSrc <= 1;
				branch		<= 0;
				wemem		<= 1;
				memToRf <= 0;
				aluop   <= 2'b00;
				j		<= 0;
			end //sw

			//beq
			6'b000_100: begin
				werf    <= 0;
				rfwasrc <= 0;
				aluSrc <= 0;
				branch		<= 1;
				wemem		<= 0;
				memToRf <= 0;
				aluop   <= 2'b01;
				j		<= 0;
			end //beq

			//j
			6'b000_010: begin
```

```
                werf    <= 0;
                rfwasrc <= 0;
                aluSrc <= 0;
                branch          <= 0;
                wemem           <= 0;
                memToRf <= 0;
                aluop   <= 2'b00;
                j               <= 1;
            end //j

            //addi
            6'b001_000: begin
                werf    <= 1;
                rfwasrc <= 0;
                aluSrc <= 1;
                branch          <= 0;
                wemem           <= 0;
                memToRf <= 0;
                aluop   <= 2'b00;
                j               <= 0;
            end //addi

            //R-type
            6'b000_000: begin
                werf    <= 1;
                rfwasrc <= 1;
                aluSrc <= 0;
                branch          <= 0;
                wemem           <= 0;
                memToRf <= 0;
                aluop   <= 2'b10;
                j               <= 0;
            end //R-type
        endcase
    end
endmodule
```

Файл alu_decoder.v:

```
module alu_decoder(
    input wire [5:0] funct,
    input wire [1:0] aluop,
    output reg [3:0] aluControl
);
    always @* begin
        if (aluop == 2'b00)             aluControl <= 4'b0010;      //add
        else if (aluop == 2'b01)        aluControl <= 4'b0110;      //sub
        else if(aluop == 2'b10) begin
            case(funct)
                //add
                6'b100_000: begin
                    aluControl <= 4'b0010;
```

```
                        end//add

                        //sub
                        6'b100_010: begin
                                aluControl <= 4'b0110;
                        end//sub

                        //and
                        6'b100_100: begin
                                aluControl <= 4'b0000;
                        end//and

                        //or
                        6'b100_000: begin
                                aluControl <= 4'b0001;
                        end//or
                endcase
        end
    end
endmodule
```

3.6 ШИНА ДАННЫХ (DATAPATH)

Шина данных (datapath) - логика путешествия данных между блоками процессора - между файлом регистром и памятью данных, установка счетчика программы в зависимости от значения текущей инструкции.

Файл adder.v:

```
module adder(
        input wire[31:0]        input1;
        input wire[31:0]        input2;
        output wire[31:0]       out;
);
        assign out = input1 + input2;
endmodule
```

Файл mux.v:

```
module mux(
        input wire selector
        input wire [31:0] input1,
        input wire [31:0] input0,
        output wire [31:0] out
)
        assign out = selector ? input1:input0;
endmodule
```

Файл alu.v:

```
module alu(
	input wire [31:0] data1,
	input wire [31:0] data2,
	input wire [4:0] aluControl,
	output reg [31:0] out,
	output wire zero,
	output wire overflow
	);
	assign zero = (out == 0);

	always @* begin
		case(aluControl)
			//AND
			4'b0000: begin
				out <= data1&data2;
			end
			//OR
			4'b0001: begin
				out <= data1|data2;
			end
			//ADD
			4'b0010: begin
				out <= data1+data2;
			end
			4'b0110: begin
				out <= data1-data2;
			end
		endcase
	end
endmodule
```

Файл datapath.v:

```
`include "adder.v"
`include "mux2.v"
`include "register_file.v"
`include "program_counter.v"
`include "alu.v"
/*********************************************
*DATA PATH
*
*ответственный за:
* - вычисление следующей команды для выполнения
* - считывание данных из файла регистров
* - передача данных в АЛУ*
* *********************************************/

module datapath(
	input	wire			clk, reset,
	input	wire [31:0]		instr, memdataout
	//данные из контроллера
	input	wire			werf, rfwasrc, memToRf,
	input	wire			j, PCSrc,
```

```
        input   wire                aluSrc,
        input   wire [3:0]          aluControl,
        input   wire                zero, overflow,

        output  wire [31:0]         pc,
        output  wire [31:0]         aluout,
        output  wire                writeData,
);

wire [4:0]      waddr;
wire [31:0] pcnext, pcplus4, pcjump, pcbranch;
wire [31:0] srca, srcb;
wire [31:0] memaluout;
wire [31:0] signimm;

//вычисление pc_next
pc      pcreg(clk, reset, pcnext, pc); //обновление текущей команды по фронту clk
adder   pcadd1(pc, 32'd4, pcplus4);
adder   pcadd2({signimm[29:0], 2'b00}}, pcplus4, pcbranch);
mux2    pcbrmux(PCSrc, pcbranch, pcplus4, pcbrnext);
mux2    pcjmux(j, {pcplus4[31:28], instr[25:0], 2'b00}, pcbrnext, pcnext);

//логика файла регистров
signext se(inst[15:0], signimm);
mux2    memalutoreg(memToRf, memdataout, aluout, memaluout);
mux2    wrtaddr(rfwasrc, instr[15:11], inst[20:16], waddr);
register_file rf(clk, instr[25:21], instr[20:16], waddr, werf, memaluout srcA, writeData);

//логика АЛУ
mux2    srcbmux(aluSrc, signimm, writeData, srcb);
alu             alu(srca, srcb, aluControl, aluout, zero, overflow);
endmodule
```

3.7 РЕГИСТР СО ЗНАКОВЫМ РАСШИРЕНИЕМ (SIGNEXT)

Регистр со знаковым расширением (signext) используется для смещения команды для его передачи в АЛУ.

Файл signext.v:

```
module signext(
        input  wire [15:0] in,
        output          wire [31:0] out
)
        assign out = {{16{in[15]}}, in};
endmodule
```

3.8 МОДУЛЬ ВЕРХНЕГО УРОВНЯ

Модуль верхнего уровня (top module) - представляет собой конкретный вариант аппаратной системы, которая будет работать на чипе ПЛИС.

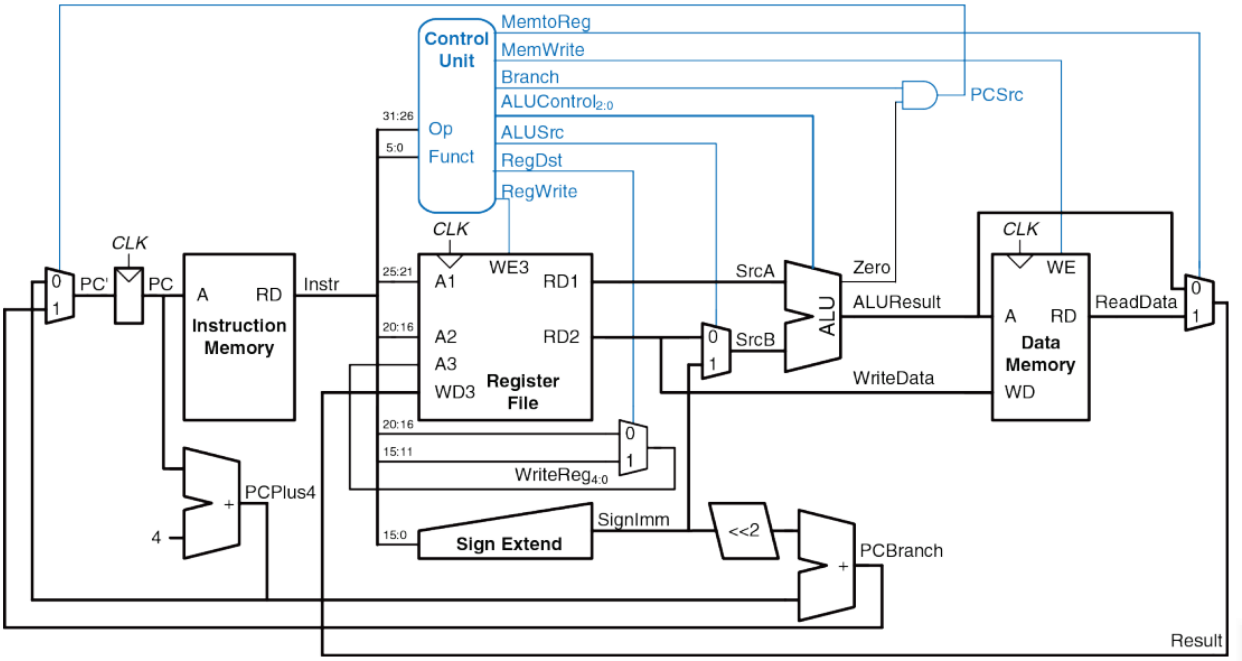
Файл mips.v:

```
`include "controller.v"
`include "datapath.v"

module mips(
input   wire            clk, reset,
input   wire [31:0]     instr, memdataout,

output  wire            wemem,
output  wire [31:0] pc, aluout, writeData,
);
wire            werf;
wire            rfwasrc, memToRf;
wire            j, PCSrc;
wire            aluSrc;
wire [3:0]      aluControl;
wire            zero, overflow;
controller c(instr[31:26], instr[5:0], wemem, werf, rfwasrc, memToRf, j, PCSrc, aluSrc, aluControl, zero, overflow);
datapath d(clk, reset, instr, memdataout, werf, rfwasrc, memToRf, j, PCSrc, aluSrc, aluControl, zero, overflow, pc, aluout, writeData);
endmodule
```

Файл top.v:

```
`include "mips.v"
`include "instruction_memory.v"
`include "data_memory.v"

module top(
input wire                  clk, reset,
output wire         wemem,
output wire [31:0] aluOut, writeData
);
wire [31:0] pc, instr, memdataout,

mips  cpu(clk, reset, instr, memdataout, wemem, pc, aluOut, writeData);
mem  m(clk, aluOut wemem, writeData, memdataout);
im          i(pc, instr);
endmodule
```

СХЕМА РАБОТЫ ОДНОТАКТНОГО ПРОЦЕССОРА MIPS



Законченный однотактный процессор MIPS в виде схемы

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были получены знания о работе процессоров MIPS архитектуры, о языке ассемблера и его конвертации в машинный код.

В программе Vivado были созданы:

- основные блоки, из которых состоит микропроцессор MIPS;
- однотактный процессор MIPS.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Цифровая схемотехника и архитектура компьютера https://microelectronica.pro/wp-content/uploads/books/digital-design-and-computer-architecture-russian-translation.pdf [интернет ресурс] (дата обращения 01.12.2022)

2. О75 Основы HDL Verilog как средства проектирования цифровых устройств: Уч. пос. / Под ред. А.И. Сухопарова. - М.: МИЭТ, 2006. - 136 с.: ил.