The City College of New York

A VHDL-based MIPS Architecture Simulation

A CSC343 Lab Report

Contributors:

Jonathan Tran, Naman Pujari, Ranjodh Cheema, Charlie Ding

Computer Organization with Professor Xiang Meng

07/22/2018

Abstract

The hardware description language, VHDL was used to study and simulate the universally common MIPS CPU architecture. All vital CPU components were designed individually and tested in isolated environments. When combined, said vital components worked in conjunction to produce MIPS-expected outputs for common I/R-type instructions, such as **load word**, **store word** and **add.** The successful execution of above commands demonstrated the success of the VHDL code in emulated all key phases of the MIPS architecture: **instruction fetch**, **instruction decode**, **execution**, **memory access**, and **write back**.

Introduction

Background and Relevant Information

MIPS, or *Microprocessor without Interlocked Pipeline Stages*, is a processor architecture that is widely studied in higher levels of education. Though its partial uses are sparse even in generous terms, the educational content it offers is valuable, as it provides a solid understanding into the inner workings of a CPU. Capable of executing **32-bit** instruction, the MIPS CPU architecture is a highly intricate combination of binary logic – it combines essential components such as adders, flip flops, and/or gates in a manner that can solve problems as a single, powerful entity.

A product of a research endeavor originating in 1984, MIPS incorporates several **vital** components, all of which can be categorized into **5** quintessential CPU phases: **IF** (instruction fetch), **ID** (instruction decode), **EX** (execution), **MEM** (memory access), and **WB** (write back). **IF** is responsible for sequentially fetching CPU instructions (triggered by an on-chip clock), **ID** is responsible for decoding the instructions, enabling functionality for specific capabilities of the CPU, **EX** is responsible for all arithmetic, or execution that is required for the instruction, **MEM**, if required, deals with access with memory or RAM, while **WB** writes back all necessary information or outputs back to the CPU register files.

Objective

We aim to use **VHDL** (Very High Speed Integrated Circuit Hardware Description Language) to emulate the inner workings of a MIPS style CPU. In specific, we wish to demonstrate all 5 phases of the MIPS architecture and employ test cases to prove the functionality of the CPU.

Methodology and Implementation

Technology Used

To achieve the objectives of this lab report, the Intel® Quartus® Prime Software, in addition to ModelSim® was used. ModelSim® assisted in visualizing output waves of compiled VHDL files, while Intel® Quartus® Prime® provided an IDE for designing all vital components of the MIPS architecture.

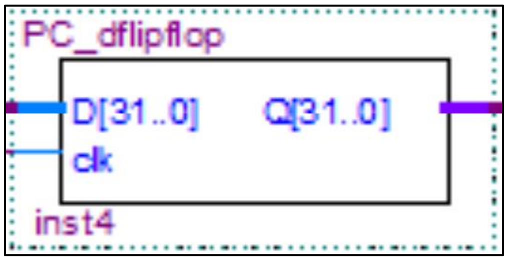
Implementation

As mentioned previously, the simulation was driven mostly by separating the MIPS architecture into 5 clear phases. They will be listed below, along with their vital components, in an organized fashion, the details of each component will be given, in addition to its role in the overall hierarchy of the MIPS architecture.

Instruction Fetch

In the conventional MIPS architecture, Instruction Fetch makes use of 3 components: the **PC** (program counter) which – at its core – is simply a D flip-flop, a 32-bit adder to increment PC value, and an Instruction Memory unit. In accordance to the typical MIPS CPU flow, the Program Counter identifies the specific line of instruction that is required by the computer at a given clock cycle. The Instruction Memory, upon receiving a query from the PC, outputs the instruction in question and feeds it to the next phase of the CPU, **ID** (instruction decode). Upon the completion of each clock cycle, the PC is also updated to the PC + 4, with the help of the 32-bit adder.

Program Counter Specifications and Code

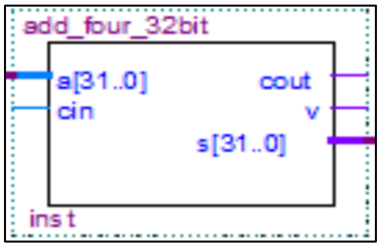


As detailed previously, the program counter is essentially a very large D flip-flop. As such, its value is updated on the bases of its input. Since it is a clock enabled unit (value changes every leading-edge trigger of the clock), one of its definite inputs is a clock signal, while the other is the next PC value (PC + 4). It's outcome, is the current (before clock update) value.

The design of the PC flip flop, in VHDL, is very simple. Upon a leading-edge clock trigger (which is verified by that fact that the clock value has changed into 1), the output takes the value of the input in the previous clock cycle. Here is a .VHD snippet that demonstrates the concepts above:

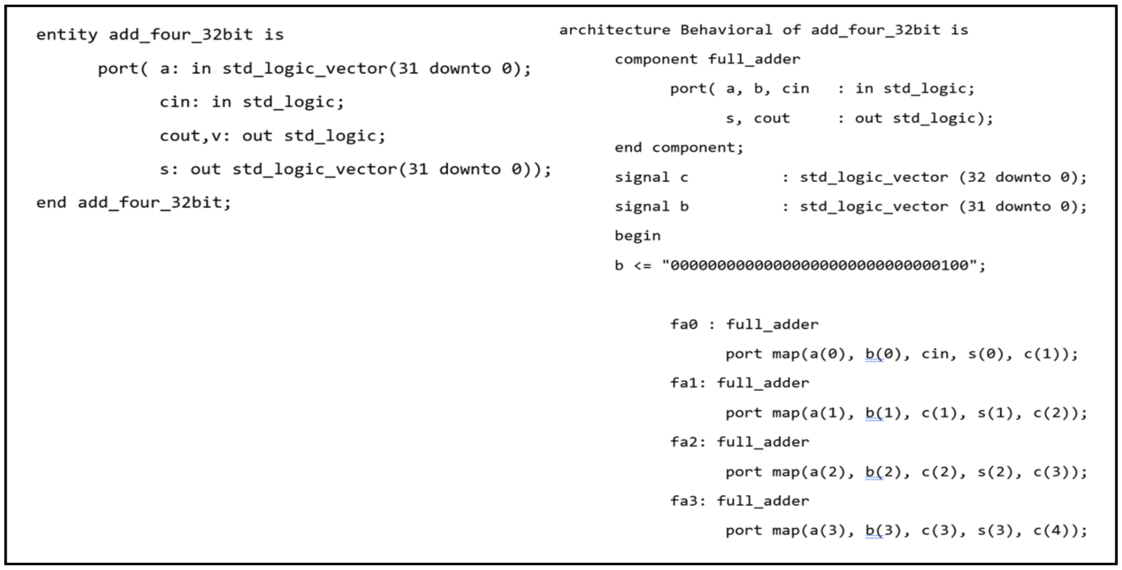
```
entity PC_dflipflop is

Port (D     : in std_logic_vector(31 downto 0);
      clk   : in std_logic;
      Q     : out std_logic_vector(31 downto 0));

end PC_dflipflop;

begin

PROCESS(clk)

        begin

          if((clk'event) and (clk='1')) then

                Q <= D;

          end if;

        end process;

end Behavioral;
```

32-Bit Adder Specifications and Code



This component is an expansion from the more common 1-2 bit full adder, in that it simply contains 32. Looking at the code snippets below, it becomes clear how the 32-bit adder is designed. Since it's purpose is to increment PC by 4, a hardcoded variable with the value of 4 in 32-bit is made available. Every bit is then fed into a 32 separate full adders to produce bit-by-bit results for the value of PC+4.

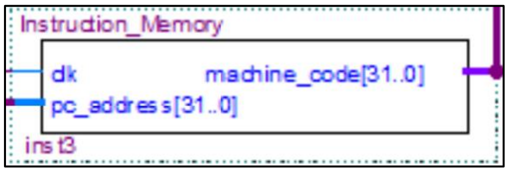
Since this is a not a clock-enabled component, the only inputs for it are, the 32-bit input to be incremented (current PC value), the carry in (any carries while conducting the sum), while the primary outputs are carry out and the 32-bit sum.



```
entity add_four_32bit is
        port( a: in std_logic_vector(31 downto 0);
                cin: in std_logic;
                cout,v: out std_logic;
                s: out std_logic_vector(31 downto 0));
end add_four_32bit;
```

```
architecture Behavioral of add_four_32bit is
    component full_adder
        port( a, b, cin    : in std_logic;
            s, cout       : out std_logic);
    end component;
    signal c            : std_logic_vector (32 downto 0);
    signal b            : std_logic_vector (31 downto 0);
    begin
    b <= "00000000000000000000000000000100";

        fa0 : full_adder
            port map(a(0), b(0), cin, s(0), c(1));
        fa1: full_adder
            port map(a(1), b(1), c(1), s(1), c(2));
        fa2: full_adder
            port map(a(2), b(2), c(2), s(2), c(3));
        fa3: full_adder
            port map(a(3), b(3), c(3), s(3), c(4));
```

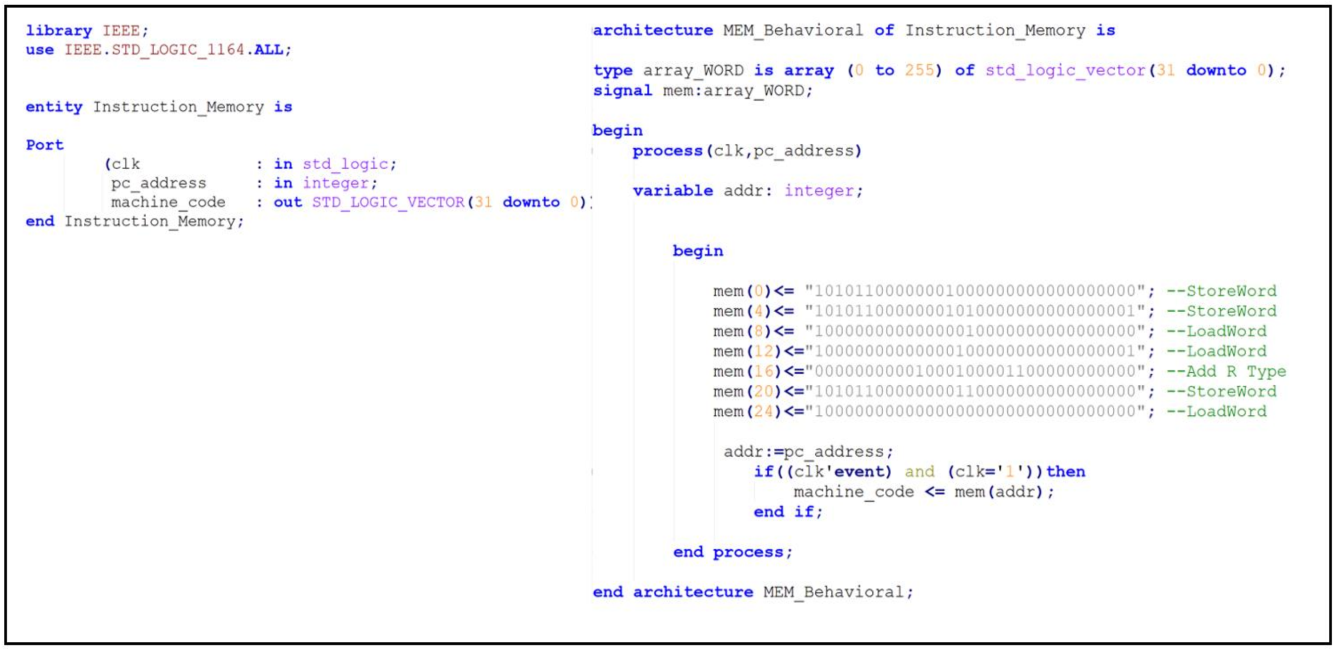
Instruction Memory Specifications and Code



The instruction memory serves to save machine code in a stable, unchanging state. To design a component that acts similarly to the Instruction Memory in the conventional MIPS architecture, a block containing an array of 32-bit instructions was created.

As this component is also clock enabled (to synchronize when the machine code is available as an output of instruction memory), it too as a clock input signal. Along with this, the block accepts the PC address as an input. The output of course is the piece of machine code that the specific input PC address pointed to.

In the code below, the PC address is interpreted as an integer, which is later used as an index to output the appropriate instruction out of an array of 7 test instructions.

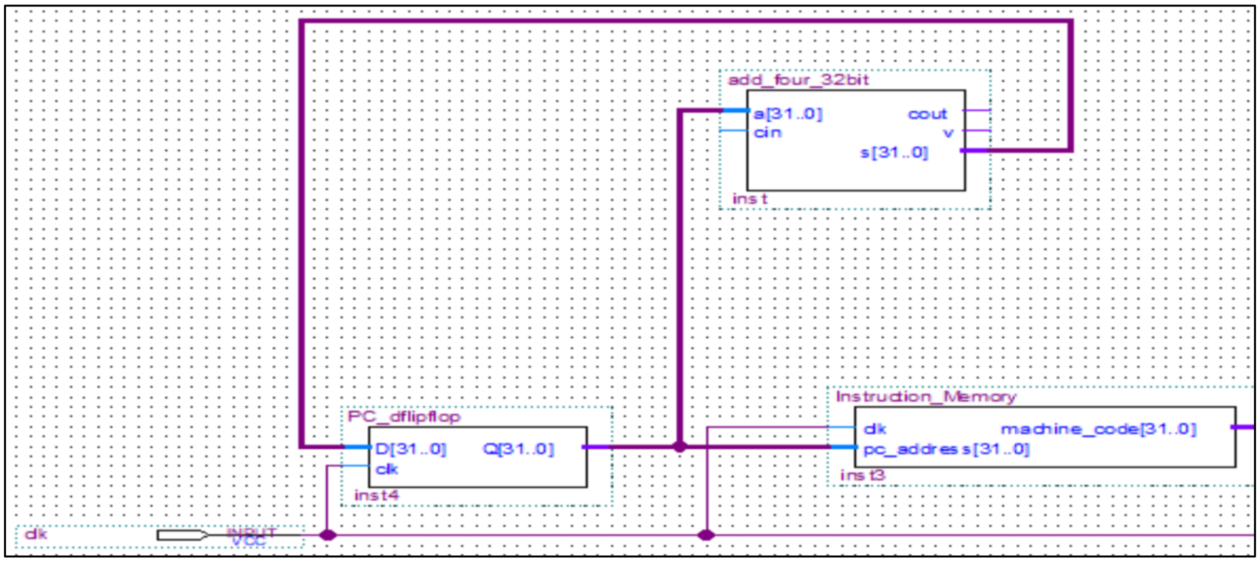


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity Instruction_Memory is

Port
        (clk              : in std_logic;
         pc_address       : in integer;
         machine_code     : out STD_LOGIC_VECTOR(31 downto 0))
end Instruction_Memory;
```

```
architecture MEM_Behavioral of Instruction_Memory is

type array_WORD is array (0 to 255) of std_logic_vector(31 downto 0);
signal mem:array_WORD;

begin
    process(clk,pc_address)

    variable addr: integer;


        begin

            mem(0)<= "10101100000000100000000000000000"; --StoreWord
            mem(4)<= "10101100000000101000000000000001"; --StoreWord
            mem(8)<= "10000000000000100000000000000000"; --LoadWord
            mem(12)<="10000000000000100000000000000001"; --LoadWord
            mem(16)<="00000000001000100001100000000000"; --Add R Type
            mem(20)<="10101100000000110000000000000000"; --StoreWord
            mem(24)<="10000000000000000000000000000000"; --LoadWord

             addr:=pc_address;
                if((clk'event) and (clk='1'))then
                    machine_code <= mem(addr);
                end if;

        end process;

end architecture MEM_Behavioral;
```

Instruction Fetch Overview

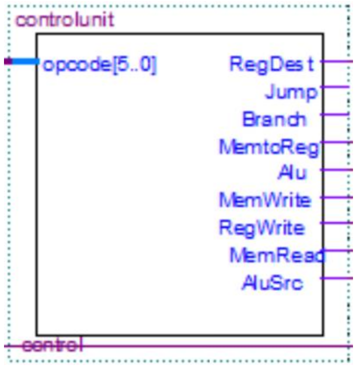
Combining the components above, we obtain a logical flow that allows us to retrieve instructions in a clock-controlled manner. In the schematic below it is easy to notice that the clock triggers the Program Counter, which then causes the Instruction Memory to output an instruction. In the same clock cycle, the Program Counter is also set to update in the next leading-edge trigger.



Instruction Decode

The Instruction Decode phase of the CPU processes what the actual instruction requires for execution. It uses many major components, two of which will be discussed in this section of the report.

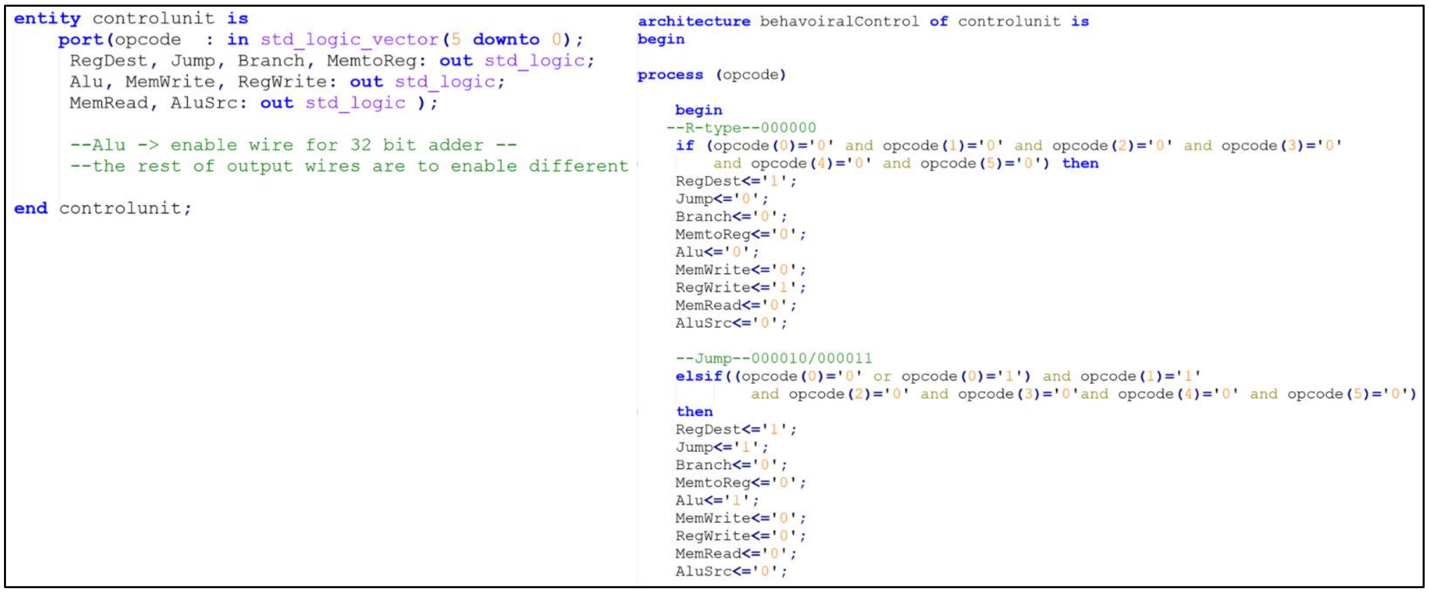
Control Unit Specifications and Code



The Control Unit is perhaps the most important component in the entire CPU, as it practically allows all instructions to function as they are intended to. A decoder, the Control Unit enables certain functionalities of the CPU over the other depending on what kind of input, or ***opcode***, it receives from the instruction.

The one input, hence, is the *opcode*, while the many outputs, *RegDest*, *Jump*, *Branch*, *MemtoReg*, and *MemWrite*, just to name a few enable the various functionalities of the CPU. These outputs physically connect to other components, all across the 5 phases of the CPU.

Here is the code for the Control Unit. Please notice how we have designed a decoder-type logic for this component.

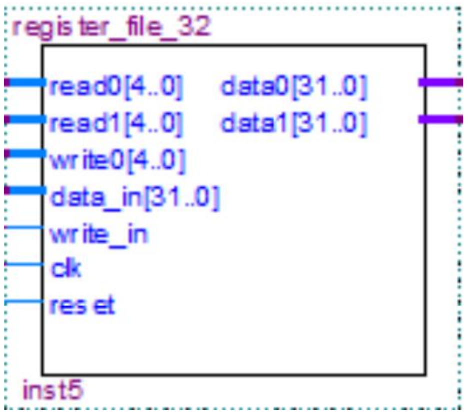


```
entity controlunit is
    port(opcode  : in std_logic_vector(5 downto 0);
     RegDest, Jump, Branch, MemtoReg: out std_logic;
     Alu, MemWrite, RegWrite: out std_logic;
     MemRead, AluSrc: out std_logic );

     --Alu -> enable wire for 32 bit adder --
     --the rest of output wires are to enable different

end controlunit;
```

```
architecture behavoiralControl of controlunit is
begin

process (opcode)

    begin
   --R-type--000000
    if (opcode(0)='0' and opcode(1)='0' and opcode(2)='0' and opcode(3)='0'
        and opcode(4)='0' and opcode(5)='0') then
    RegDest<='1';
    Jump<='0';
    Branch<='0';
    MemtoReg<='0';
    Alu<='0';
    MemWrite<='0';
    RegWrite<='1';
    MemRead<='0';
    AluSrc<='0';

    --Jump--000010/000011
    elsif((opcode(0)='0' or opcode(0)='1') and opcode(1)='1'
            and opcode(2)='0' and opcode(3)='0'and opcode(4)='0' and opcode(5)='0')
    then
    RegDest<='1';
    Jump<='1';
    Branch<='0';
    MemtoReg<='0';
    Alu<='1';
    MemWrite<='0';
    RegWrite<='0';
    MemRead<='0';
    AluSrc<='0';
```

Register File Specifications and Code

The Register File is the on-CPU memory. It is (in terms of hardware speed) extremely fast, and all the arithmetic in the MIPS CPU only deals with data found directly in the Register File.



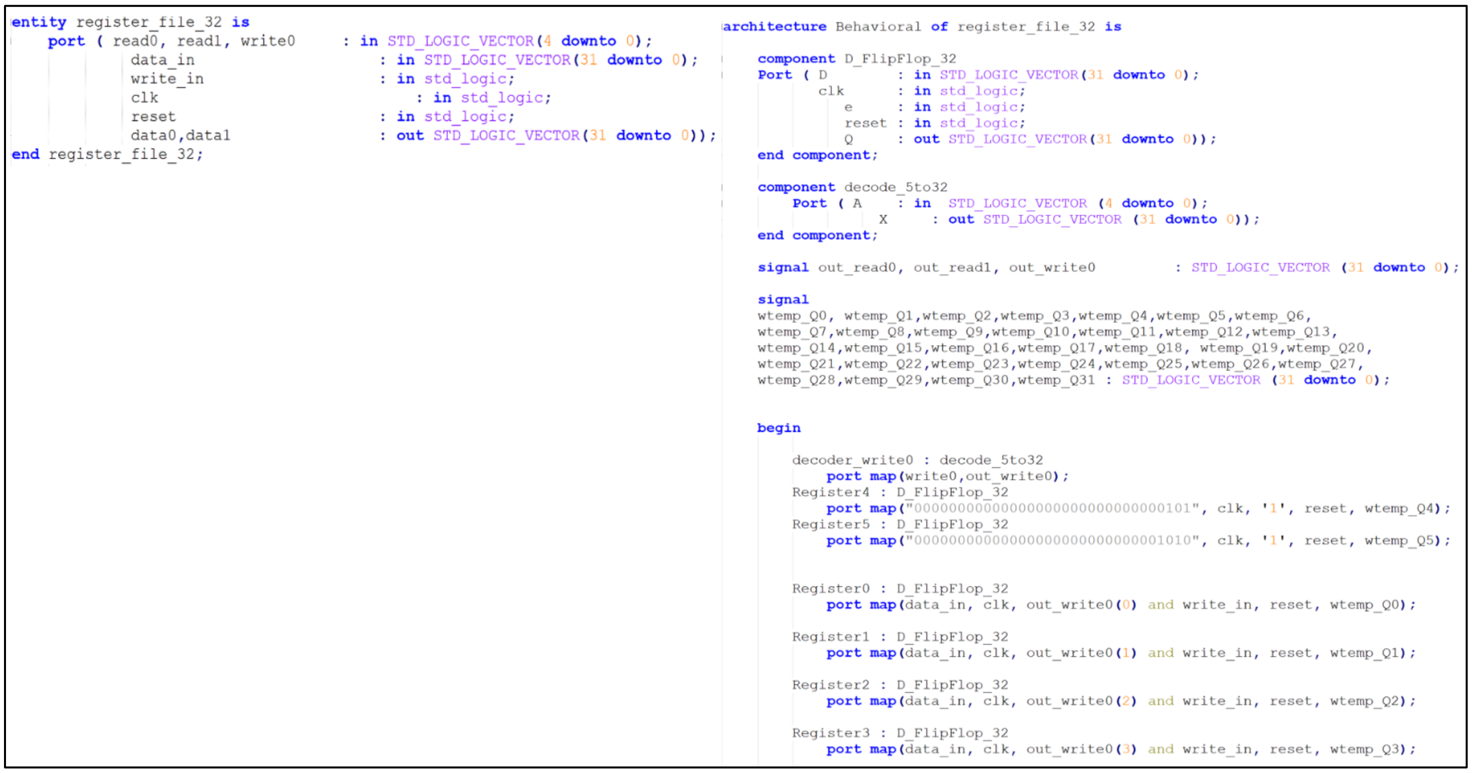
It stores 32 registers, all with the capacity of holding 32 bits each. The inputs of the register file are as follows:

- *read0* – the first register to read from (obtained from instruction code)
- *read1* – the second register to read from
- *write0* – the register to write into (overwrites previously stored data)
- *data_in* – data to be written to the register marked for write
- *write_in* – enabler coming from control unit that allows for the register file to be written into.

Since this is also a clock-enabled component, it has both the *clk* and *reset* inputs.

The outputs of the register file are *data0*, and *data1*, the data retrieved from read register 1 and read register 2 specifically.

The code for the register file is included below.



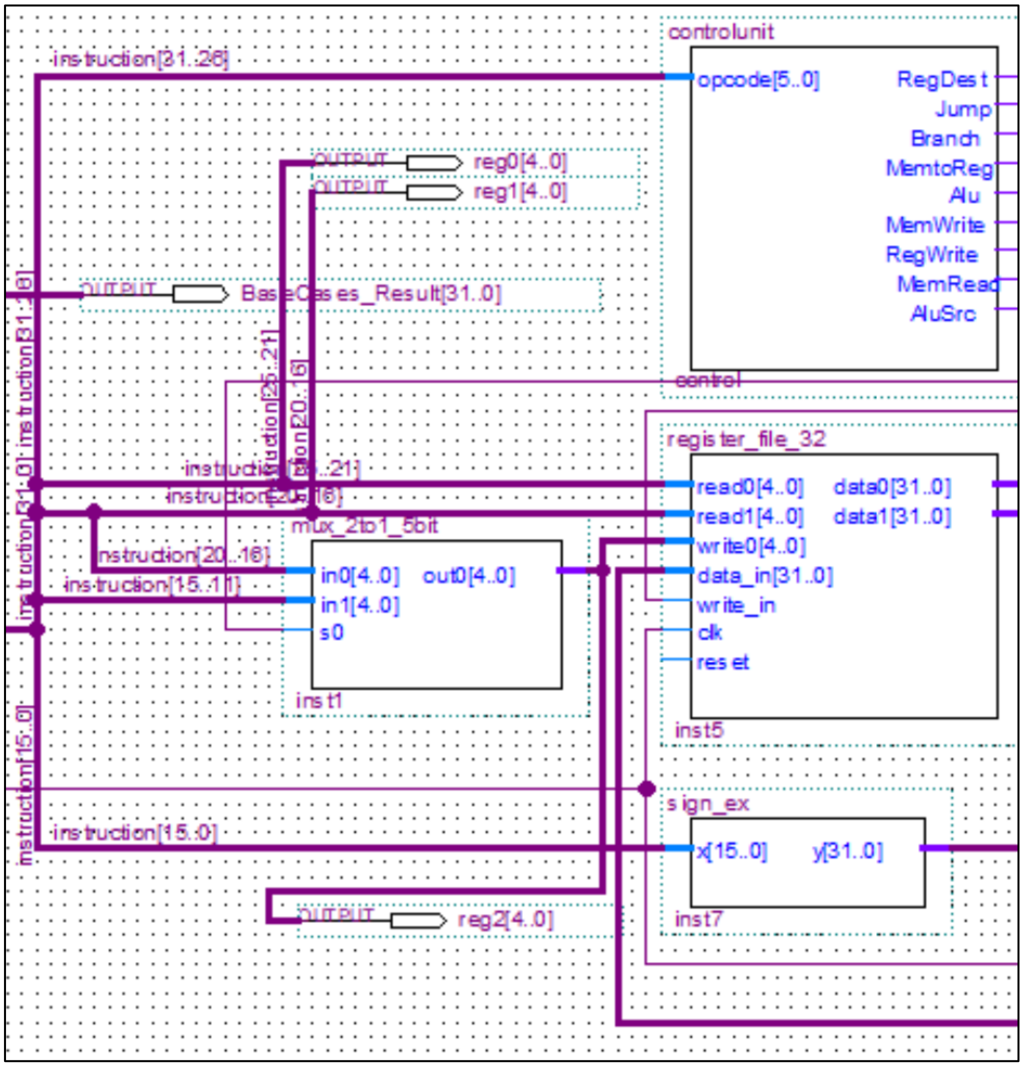
```vhdl
entity register_file_32 is
    port ( read0, read1, write0     : in STD_LOGIC_VECTOR(4 downto 0);
            data_in                     : in STD_LOGIC_VECTOR(31 downto 0);
            write_in                    : in std_logic;
            clk                             : in std_logic;
            reset                       : in std_logic;
            data0,data1                 : out STD_LOGIC_VECTOR(31 downto 0));
end register_file_32;
```

```vhdl
architecture Behavioral of register_file_32 is

    component D_FlipFlop_32
    Port ( D        : in STD_LOGIC_VECTOR(31 downto 0);
           clk      : in std_logic;
             e      : in std_logic;
             reset : in std_logic;
             Q      : out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    component decode_5to32
        Port ( A    : in  STD_LOGIC_VECTOR (4 downto 0);
                  X     : out STD_LOGIC_VECTOR (31 downto 0));
    end component;

    signal out_read0, out_read1, out_write0         : STD_LOGIC_VECTOR (31 downto 0);

    signal
    wtemp_Q0, wtemp_Q1,wtemp_Q2,wtemp_Q3,wtemp_Q4,wtemp_Q5,wtemp_Q6,
    wtemp_Q7,wtemp_Q8,wtemp_Q9,wtemp_Q10,wtemp_Q11,wtemp_Q12,wtemp_Q13,
    wtemp_Q14,wtemp_Q15,wtemp_Q16,wtemp_Q17,wtemp_Q18, wtemp_Q19,wtemp_Q20,
    wtemp_Q21,wtemp_Q22,wtemp_Q23,wtemp_Q24,wtemp_Q25,wtemp_Q26,wtemp_Q27,
    wtemp_Q28,wtemp_Q29,wtemp_Q30,wtemp_Q31 : STD_LOGIC_VECTOR (31 downto 0);


    begin

        decoder_write0 : decode_5to32
            port map(write0,out_write0);
        Register4 : D_FlipFlop_32
            port map("00000000000000000000000000000101", clk, '1', reset, wtemp_Q4);
        Register5 : D_FlipFlop_32
            port map("00000000000000000000000000001010", clk, '1', reset, wtemp_Q5);


        Register0 : D_FlipFlop_32
            port map(data_in, clk, out_write0(0) and write_in, reset, wtemp_Q0);

        Register1 : D_FlipFlop_32
            port map(data_in, clk, out_write0(1) and write_in, reset, wtemp_Q1);

        Register2 : D_FlipFlop_32
            port map(data_in, clk, out_write0(2) and write_in, reset, wtemp_Q2);

        Register3 : D_FlipFlop_32
            port map(data_in, clk, out_write0(3) and write_in, reset, wtemp_Q3);
```

Continued in next page.

Instruction Decode Overview

Here is the complete logical flow of the Instruction Decode phase



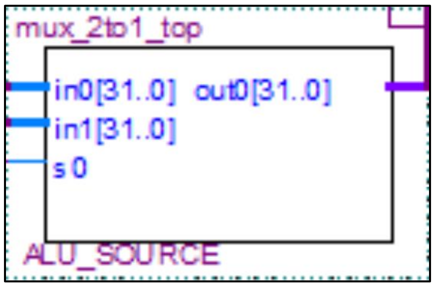
The instruction received from the Instruction Fetch phase is split into 4 distinct parts. The first 6 bits of the machine code are inputted into the Control Unit. It then enables the necessary components for CPU function. 5-bit sequences corresponding to read register 1, 2, and write register are inputted into their appropriate slots in the register file. Read register 2 and any immediate value goes through a MUX to delineate between R-type and I-type instructions. Finally, the last 16 bits (bottom most part of the schematic) goes through a sign extender (to convert the value into 32 bits for any arithmetic), and directly into the **Execution** phase.

<u>Execution</u>

As the name suggests, this phase of the CPU deals with executing the arithmetic involved in machine code. Arithmetic ranging from addition to multiplication and division is supported through clever bitwise logic techniques in commercial/standard MIPS CPUs. Although designing such an execution unit, or ALU is desirable, it suffices (for out case) to design an ALU that supports both addition and subtraction. Most commands involving memory access and Program Counter manipulation relied on addition/subtract more than they do on other arithmetic operations.

Upon receiving all the data that is required for be processed, the ALU, in addition to a MUX choosing between register (R-type) and immediate value (I-type) data, outputs the appropriate result, which is then pipelined into both the Memory Access and Write Back phases.

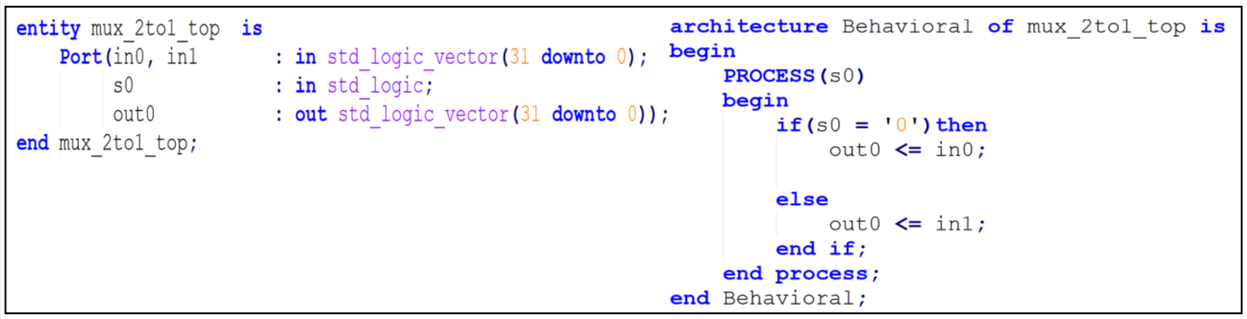
ALU Source Multiplexer Specifications and Code



Located before the second input of then ALU, the ALU-source enabled multiplexer, using its switch-like capabilities, determines whether arithmetic is done between two register data values or one register data value, and one immediate value. In the case of I-type commands such as load/store word and branch, a multiplexer like this is **essential** as it allows the ALU to conduct arithmetic on the **correct** data values.

As can be seen, the three inputs of the multiplexer include – of course – two data inputs, one from the second **read register** from the Instruction Decode phase, and one from the immediate value, after passing through the sign-extend module, also found in the Instruction Decode phase. The third input, and enabler of the multiplexer is a line coming directly from the Control Unit (again, located in ID). The one output of the MUX is fed into the second input of the ALU (to be described), and can either be the R-type register data or I-type immediate data.

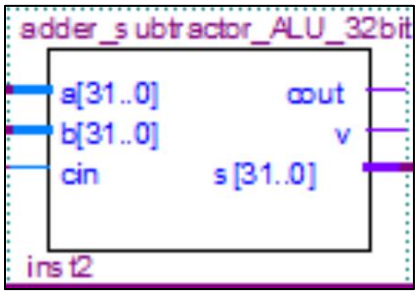
In the code below, `s0`, or the control line simply acts as a switch to alter between R and I-type data.



```
entity mux_2to1_top  is
    Port(in0, in1        : in std_logic_vector(31 downto 0);
         s0              : in std_logic;
         out0            : out std_logic_vector(31 downto 0));
end mux_2to1_top;
```

```
architecture Behavioral of mux_2to1_top is
begin
    PROCESS(s0)
    begin
        if(s0 = '0')then
            out0 <= in0;

        else
            out0 <= in1;
        end if;
    end process;
end Behavioral;
```

ALU Specifications and Code

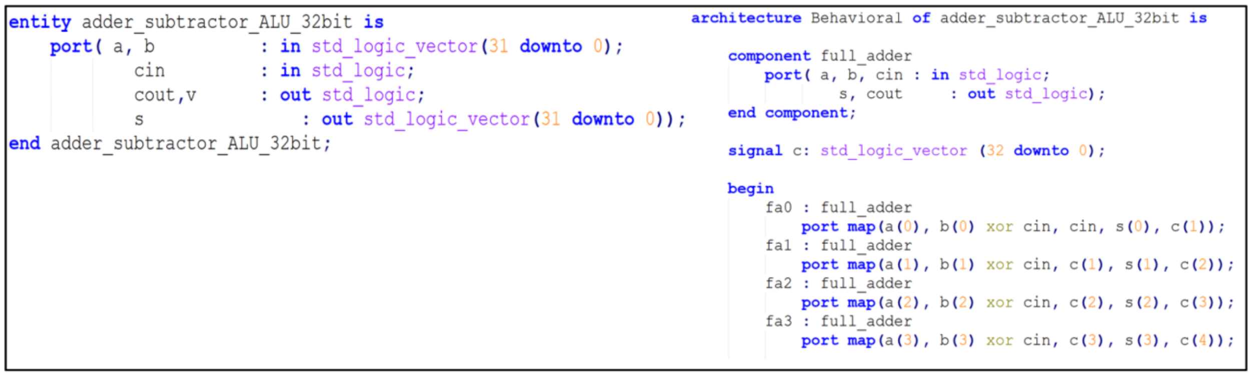


The arithmetic powerhouse of the CPU, the report version of the ALU contains full adders that can be designed to both conduct addition and subtraction. This component, hence, is very similar to the conventional full adder, that performs bitwise arithmetic on long 32-bit sequences of data.

Our VHDL version of the ALU contains 3 primary inputs, two being input *a* and input *b*, the addend and the adder, while the last input being the *carry in*. The two primary outputs of this component are *s*, the sum (this is valid even for subtraction, as *a – b = a + (-b)*) and the *carry out*.

Please note that the input *a* will **always** be the output of register 1 in the Instruction Decode phase. Input *b* can either be the output of register 2 **or** the immediate value (after going through the sign extend) in the Instruction Decode phase. That is the purpose of the multiplexer explained prior to this.

In the code below (which is expectedly similar to that of the adder in Instruction Fetch) notice the use of multiple full adders to conduct both addition and subtraction.

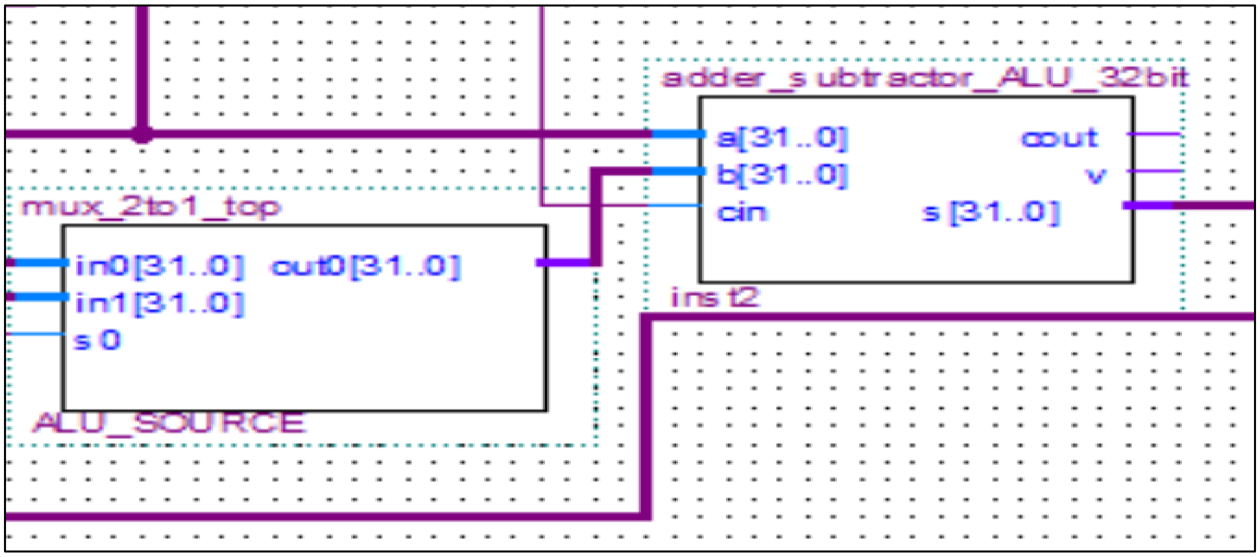


```
entity adder_subtractor_ALU_32bit is
    port( a, b          : in std_logic_vector(31 downto 0);
          cin           : in std_logic;
          cout,v        : out std_logic;
          s                 : out std_logic_vector(31 downto 0));
end adder_subtractor_ALU_32bit;
```

```
architecture Behavioral of adder_subtractor_ALU_32bit is

    component full_adder
        port( a, b, cin : in std_logic;
                s, cout     : out std_logic);
    end component;

    signal c: std_logic_vector (32 downto 0);

    begin
        fa0 : full_adder
            port map(a(0), b(0) xor cin, cin, s(0), c(1));
        fa1 : full_adder
            port map(a(1), b(1) xor cin, c(1), s(1), c(2));
        fa2 : full_adder
            port map(a(2), b(2) xor cin, c(2), s(2), c(3));
        fa3 : full_adder
            port map(a(3), b(3) xor cin, c(3), s(3), c(4));
```

Execution Overview

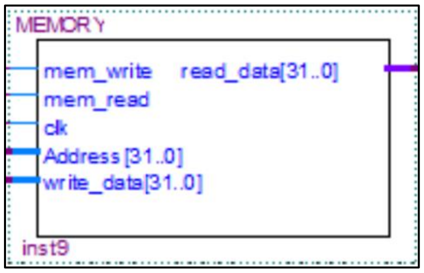
Combining the components above, we obtain a logical flow that allows us to feed in appropriate data into an arithmetic-enabled component named the ALU, and retrieve an output based on its operations. Firstly, the output of the first register in the Instruction Decode phase is fed into first input of the ALU, and secondly, the output of the MUX deciding between register 2 data and immediate data is fed into the second input of the ALU.



Memory Access

This phase of the CPU functionality deals with memory not directly mounted on the chip itself. Rather, it deals with accessing data in a distant location on the hardware, known as **RAM**, or **Main Memory.** The data in Main Memory can be read and write. It can also be accessed at any time during the MIPS CPU's running time. The logical flow of the Memory Access is perhaps the simplest out of all phases, simply because there is only one major component: Main Memory itself.

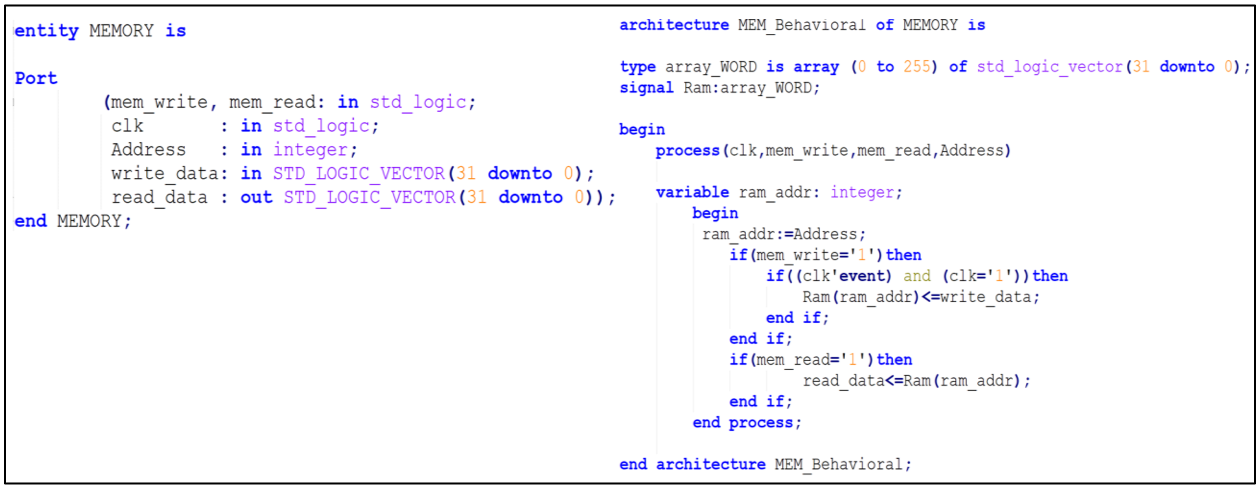
Main Memory Specifications and Code



The Main Memory block is simply an array of words (32-bit sequences) that can be accessed in the form of either read or write. As such, the 2 most important inputs of the block are *address*, which located which 32-bit sequence is to be accessed, and *write_data*, which defines what data is to be written, and to what address. There are also two control pin inputs, *mem_write* and *mem_read*. These two enable the write and read capabilities of Main Memory respectively. Hence, if *mem_write* is enabled, then both *address* and *write_data* must be fed an input. Also, the component **is** clock triggered, so an input for it is also mandatory.

The sole output of Main Memory is *read_data*, which, in consistency with its name, outputs data based on the *address* input. The output of *read_data* has functionality in the Write Back phase, which will be touched on next in this report.

Refer to the code snippet below to understand how Memory stores words for later access. We have not included any hardcoded data into the component intentionally, as we intend to test *store word* instructions on our CPU before *load word* instructions.

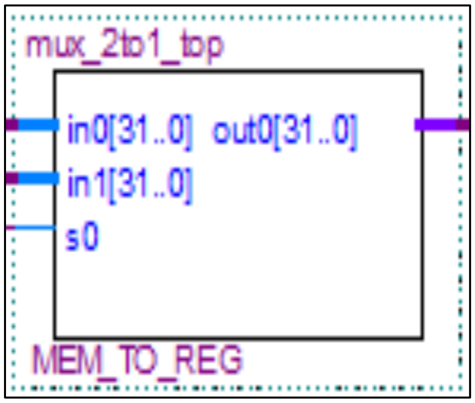


```
entity MEMORY is

Port
        (mem_write, mem_read: in std_logic;
         clk        : in std_logic;
         Address    : in integer;
         write_data: in STD_LOGIC_VECTOR(31 downto 0);
         read_data : out STD_LOGIC_VECTOR(31 downto 0));
end MEMORY;
```

```
architecture MEM_Behavioral of MEMORY is

type array_WORD is array (0 to 255) of std_logic_vector(31 downto 0);
signal Ram:array_WORD;

begin
    process(clk,mem_write,mem_read,Address)

    variable ram_addr: integer;
        begin
         ram_addr:=Address;
           if(mem_write='1')then
               if((clk'event) and (clk='1'))then
                   Ram(ram_addr)<=write_data;
               end if;
           end if;
           if(mem_read='1')then
                   read_data<=Ram(ram_addr);
           end if;
        end process;

end architecture MEM_Behavioral;
```

<u>Write Back</u>

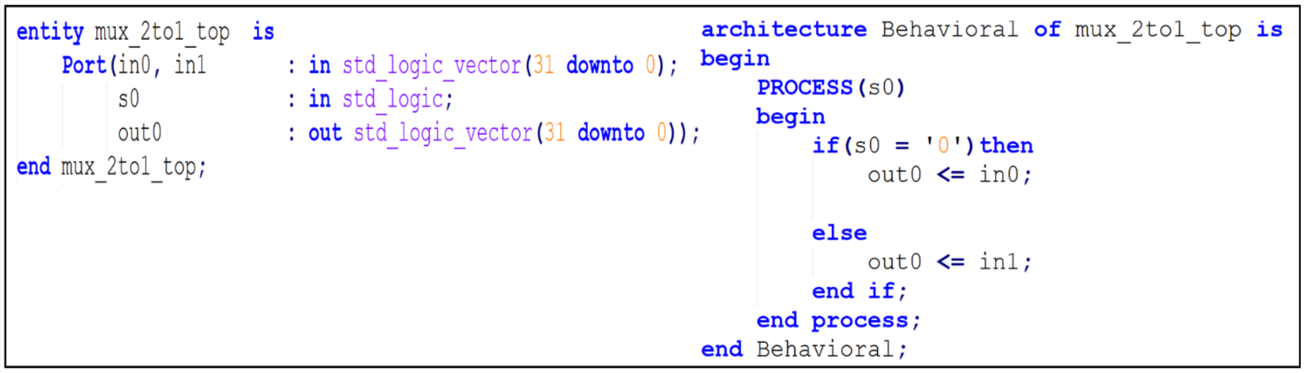
The Write Back phase is just as crucial for the functionality of the CPU as all of the other 4 phases. Most closely related to the Memory Access and Instruction Decode phases, the Write Back phase deals with deliver the end product of the CPU instruction back to the register file (which is the on-chip memory).



As data written back to the register file (going into the *data_in* input of the register file discussed in the Instruction Decode phase), can originate from either Main Memory (in instructions that deal with memory) or directly from the ALU, the only component in this phase is a MUX that can choose from either of these options.

In the block diagram to the left, *in0* is write back data originating from the ALU, while *in1* is write back data originating from Main Memory. *s0* is a control unit input that decides between which of the two to write into the register file.

Please take a look at the code below to get an idea of how this final component of the CPU functions.



```
entity mux_2to1_top  is
    Port(in0, in1        : in std_logic_vector(31 downto 0);
         s0              : in std_logic;
         out0            : out std_logic_vector(31 downto 0));
end mux_2to1_top;
```

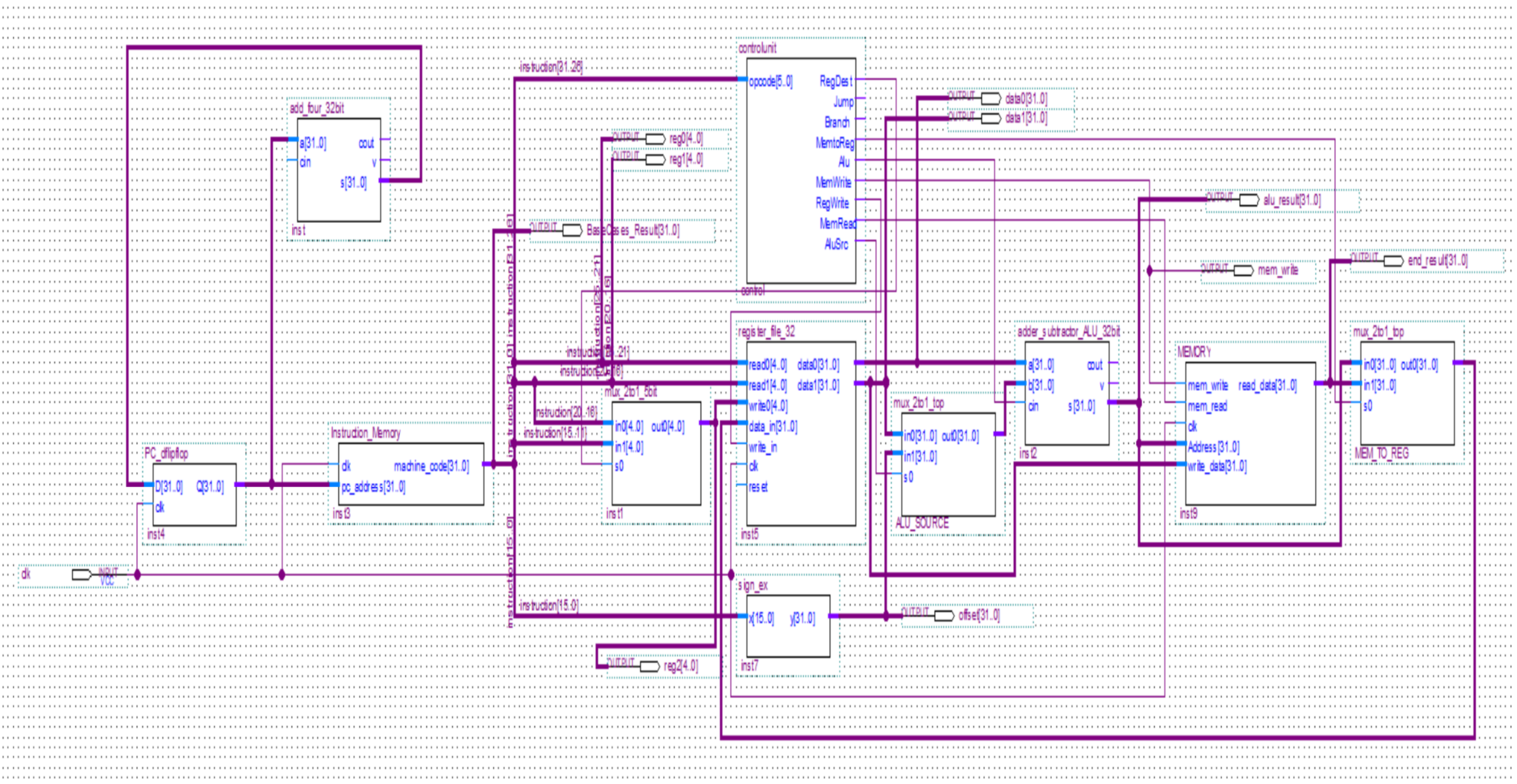
```
architecture Behavioral of mux_2to1_top is
begin
    PROCESS(s0)
    begin
        if(s0 = '0')then
            out0 <= in0;

        else
            out0 <= in1;
        end if;
    end process;
end Behavioral;
```

Testing and Results

MIPS Component Assembly

To set up the test case for the VHDL MIPS CPU, we assembled every component appropriately. Upon completion of the assembly, the architecture of our VHDL CPU looked as such: (picture in next page). The delineations for the 5 phases have not been made, but referring to the component breakdown prior to this section, the phases become clear.

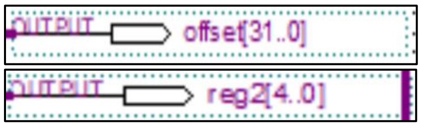
Assembly picture located in next page.



add_four_32bit
a[31..0]
cin
cout
v
s[31..0]
inst
PC_dflipflop
D[31..0]
Q[31..0]
clk
inst4
Instruction_Memory
clk
machine_code[31..0]
pc_address[31..0]
inst3
clk
INPUT
VCC
instruction[31..26]
instruction[25..21]
instruction[20..16]
instruction[15..11]
instruction[15..0]
instruction[31..0]
OUTPUT
BaseCases_Result[31..0]
reg0[4..0]
reg1[4..0]
reg2[4..0]
mux_2to1_5bit
in0[4..0]
out0[4..0]
in1[4..0]
s0
inst1
controlunit
opcode[5..0]
RegDest
Jump
Branch
MemtoReg
Alu
MemWrite
RegWrite
MemRead
AluSrc
control
register_file_32
read0[4..0]
data0[31..0]
read1[4..0]
data1[31..0]
write0[4..0]
data_in[31..0]
write_in
clk
reset
inst5
sign_ex
x[15..0]
y[31..0]
inst7
offset[31..0]
data0[31..0]
data1[31..0]
mux_2to1_top
in0[31..0]
out0[31..0]
in1[31..0]
s0
ALU_SOURCE
adder_subtractor_ALU_32bit
a[31..0]
cout
b[31..0]
v
cin
s[31..0]
inst2
alu_result[31..0]
mem_write
MEMORY
mem_write
read_data[31..0]
mem_read
clk
Address[31..0]
write_data[31..0]
inst9
end_result[31..0]
mux_2to1_top
in0[31..0]
out0[31..0]
in1[31..0]
s0
MEM_TO_REG

Testing Technique

The processes of the MIPS CPU are unreadable for the most part (apart from, of course the write back output). To provide a crucial part of our test bench (which is the ability to analyze the outputs of the CPU at every point), we introduced **test variables,** using existing capabilities in the Quartus® Prime IDE. Schematic-wise, these variables appear as rectangular outputs (image on the right). The data passed through any wire that these variables attach to can be read at every clock cycle and provides us a good way of verifying the legitimacy of our CPU, not just by analyzing its output, but by analyzing its processes at every point.



In our MIPS VHDL assembly, we have included the following test variables:

- `clk` – clock signal oscillating between values of 0 and 1, in intervals in the order of nanoseconds
- `BaseCases_Result` – the instruction memory being processed during the CPU function cycle
- `reg0` – read register #1
- `reg1` – read register #2
- `reg2` – write register
- `offset` – immediate value (for I-Type instructions), if any
- `data0` – data in read register #1
- `data1` – data in read register #2
- `alu_result` – output of ALU after all arithmetic processes have been completed
- `mem_write` – control pin output indicating whether Main Memory will be written into
- `end_result` – data queried from the Main Memory (memory read from Main Memory)

Since we had set up our Instruction Memory in such a way that it was preloaded with 32-bit instructions, it was not necessary to set up input variables in our test bench – the instructions would be taken from the Instruction Memory during an end to end flow of our CPU. Below are the instructions preloaded into memory:

- *store word* instruction =101011 00000 00100 0000000000000000
  - save register 4 into memory address 0
- *store word* instruction = 101011 00000 00101 0000000000000001
  - save register 5 into memory address 1
- *load word* instruction = 100000 00000 00001 0000000000000000
  - memory will have 5 at memory address 0.
  - load 5 in memory address 0 into register 1.
- *load word* instruction = 100000 00000 00010 0000000000000001
  - memory will have 10 at memory address 1.
  - load 10 in memory address 1 into register 2.
- *add* instruction = 000000 00001 00010 00011 00000000000
  - reg1 = 5, reg2 = 10
  - save reg1 + reg2 (15) in register 3.
- *store word* instruction = 101011 00000 00011 0000000000000000
  - save Register 3 (15) into memory address 0.
- *load word* instruction = 100000 00000 00000 0000000000000000
  - memory will have 15 at memory address 0
  - load 15 in memory address 0

Note:

The following register values are *initially* hardcoded:

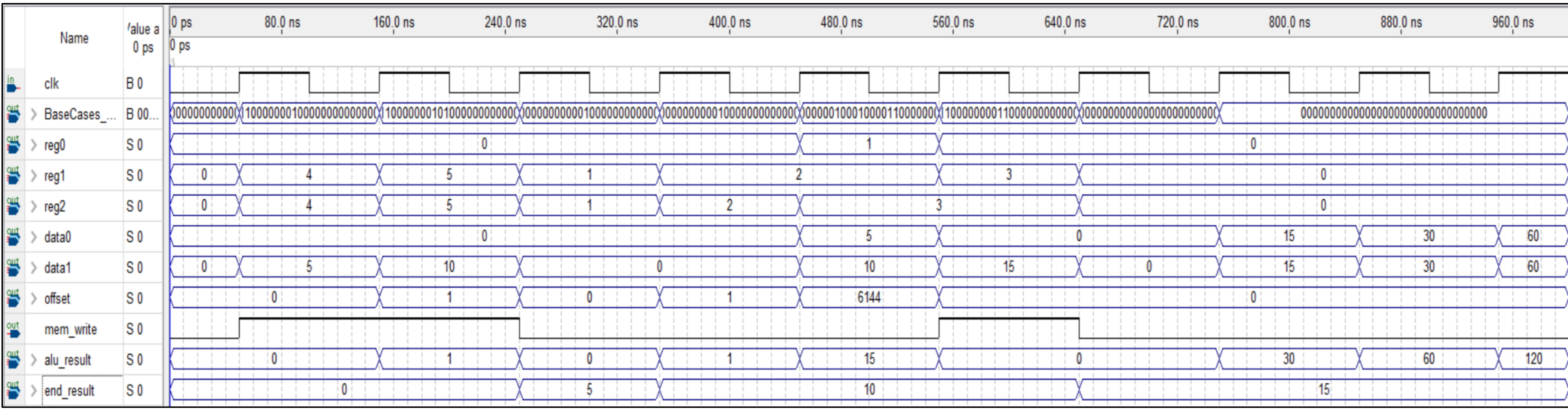
Register 4 → 5
Register 5 → 10
Register 0 → 0

Note how every following instruction deals with information retrieved from the previous one. This was intentional, as we wanted a testing environment that dealt very little with hardcoded variables, and imitated the process of actual MIPS CPUs. It is also very important to note that these instructions are both R-type and I-type, so type compatibility will also be tested for this CPU.

Test Results

The outputs for each of the **test variables** described in the section above are included in the next page:

Test variable outputs located in the next page.

Name
Value at 0 ps
0 ps
80.0 ns
160.0 ns
240.0 ns
320.0 ns
400.0 ns
480.0 ns
560.0 ns
640.0 ns
720.0 ns
800.0 ns
880.0 ns
960.0 ns
clk B 0
BaseCases_... B 00...
0000000000
1000000010000000000000
1000000010100000000000
0000000000100000000000
0000000001000000000000
0000010001000011000000
1000000001100000000000
0000000000000000000000
00000000000000000000000000000000
reg0 S 0
0 1 0
reg1 S 0
0 4 5 1 2 3 0
reg2 S 0
0 4 5 1 2 3 0
data0 S 0
0 5 0 15 30 60
data1 S 0
0 5 10 0 10 15 0 15 30 60
offset S 0
0 1 0 1 6144 0
mem_write S 0
alu_result S 0
0 1 0 1 15 0 30 60 120
end_result S 0
0 5 10 15

Result Analysis

Note that each instruction is delineated by each rising edge trigger in the clock cycle. Analyzing the nature of each **test variable** during the instruction processes can help us verify the legitimacy of the VHDL MIPS CPU. Please refer to the "**Testing Technique**" section to understand what each instruction is meant to do.

Instruction 1 (**between 50ns and 150ns**): the value of *reg0* is 0, as it should be. At the same time, both *reg1* and *reg2* are identified to be register 4, as the value in that register must be stored into memory. We see later in *data1* that the data stored in register 4 is 5, and that the offset value is 0 (which confirms that the memory address will indeed be 0). At the same time, *mem_write* is triggered to a one, which also confirms that something will be written into Main Memory (the value 5, of register 4). Both *alu_result* and *end_result* is 0, since we are accessing memory address 0, and since nothing is being written back, respectively. **Later instructions will confirm that the value 5 was indeed written into memory address 0.**

Instruction 2 (**between 150ns and 250ns**): this is also a similar *store word* instruction. The value of *reg0* again is 0, which it should be. This time the value of *reg1* is 5 as we are writing the value of register 5 (10) into memory. The value of *data1* is 10, which confirms that 10 will be written into memory address 1. The *offset* value is 1, which again proves that the memory address is 1. Also, for the running time of this instruction, *mem_write* is 1. *end_result* is 0 because nothing is written back. **Later instructions will confirm that the value 10 was indeed written into memory address 1.**

Instruction 3 (**between 250ns and 350ns**): this is a *load word* instruction. The value of *reg0* is 0, which must be in the case in this I-type instruction. The value of *reg2* is 1, as we will write Main Memory data into register 1. *data1* is 0 because nothing is stored in the first register yet. The value of *offset* is 0 since we are writing the value in address 0. Also, *mem_write* is 0 because we are reading from Main Memory and not writing. Finally, the *end_result* value is 5, confirming that 5 (the value of register 4 that was stored into memory) was retrieved from memory, and written to register 1. **This confirms that Instruction 1 worked.**

Instruction 4 (**between 350ns and 450ns**): this is also a *load word* instruction. The value of *reg0* is 0, which must be in the case in this I-type instruction. The value of *reg2* is 2, as we will write Main Memory data into register 2. *data1* is 0 because nothing is stored in the second register yet. The value of *offset* is 1 (same with *alu_result)* since we are writing the value in address 1. Also, *mem_write* is 0 because we are reading from Main Memory and not writing. Finally, the *end_result* value is 10, confirming that 10 (the value of register 5 that was stored into memory) was retrieved from memory, and written to register 2. **This confirms that Instruction 2 worked.**

Instruction 5 (**between 450ns and 550ns**): this is an *add* instruction. The value of *reg0* finally changes to 1 as it is the addend register, and the value of *reg1* is 2, as it is the adder. The value of *reg2* is 3, as it is the register we wish to write the ALU result into. *data1* and *data2* are 5 and 10 respectively, and the *offset* value is 6144, which is irrelevant in this case (the value is very high because of the R-type format – we will not use the *offset* value). The value for *mem_write* is expectedly 0 also. Finally, the value of *alu_result* is 15, as it is the answer (5+10) we want to write into register 3. **Note: *end_result* is not 15 only because it is the data coming out of the memory slot (which, between the previous instruction and this one, was not changed). *alu_result* is the variable we care about, and it is expectedly 15.**

Instruction 6 (**between 550ns and 650ns**): this is a *store word* instruction. The value of *reg0* is 0, as it should be. At the same time, *reg1* is 3, as the value in that register must be stored into memory. We see later in *data1* that the data stored in register 3 is 15 (the answer from the previous instruction, and that the offset value is 0 (which confirms that the memory address will indeed be 0). At the same time, *mem_write* is triggered to a one, which also confirms that something will be written into Main Memory (the value 15, of register 3). Both *alu_result* and *end_result* is 0, since we are accessing memory address 0, and since nothing is being written back, respectively. **The next instruction will confirm that the value 15 was indeed written into memory address 0.**

Instruction 7 (**between 650ns and 750ns**): this is a *load word* instruction. By the end of the running time of the instruction, we see that *end_result* has a value of 15, which proves that this instruction works, and that **the VHDL MIPS CPU works.** Since the instruction specifies registers to be 0, we only care about *end_result,* which has the proper value (the CPU accessed the memory address 0 and retrieved the 15 that was written into it in the previous instruction). **This also proves that Instruction 6 worked.**

The analyses of the instruction above prove the proper functioning of our MIPS simulation in VHDL.

Conclusion

We have been successful in emulating a simple MIPS processor entirely built on VHDL. Although the processor is known by many to be rather simplistic, creating a copy of it proved not only to be challenging, but fun. We, as a group, learned a lot about the inner workings of the components that go into building a large and sophisticated handler for simple Boolean logic. In future endeavors, we aim to extend our project to multi core CPUs and hyperthreading.