ECE 485 Project 2
Design and Implementation of a MIPS CPU with a Multicycle Datapath

Muntaser Khan – A20252490

Subash Luitel – A20275605

Instructor: Dr. Suresh Borkar

ECE 485-01

Due Date: 11/06/13

Executive Summary

In this project report we discuss the design and implementation of the project by making certain assumptions. Then we go over the output result and compare it with the hand-calculated measurement. The output results are explained and the codes covering the CPU are attached in the appendix.

Introduction

In this project, we design a custom RISC processor which is basically a stripped down MIPS processor. We get more practical hands-on approach to computer architecture design problems. The
processor we designed is a 32-bit version of the MIPS processor and the instruction set will be a small subset of the actual MIPS ISA. We implemented the multicycle datapath version of the processor utilizing the VHDL hardware descriptive language. The processor must support the three types of instruction formats of R, I, and J (beq), along with store word and load word. The table below summarizes the instructions that have to be provided:

Table I: Core MIPS Instruction Set to be Designed (with example)

OpCode [31 : 26]	Function Field [5 : 0]	Instruction	Operation
100011	--	lw	lw $t3, 200($s2)
101011	--	sw	sw $t4, 100($t3)
000000	100000	add	add $s3, $t2, $s2
000100	--	beq	beq $s5, $s2, 500
		(Custom set)	

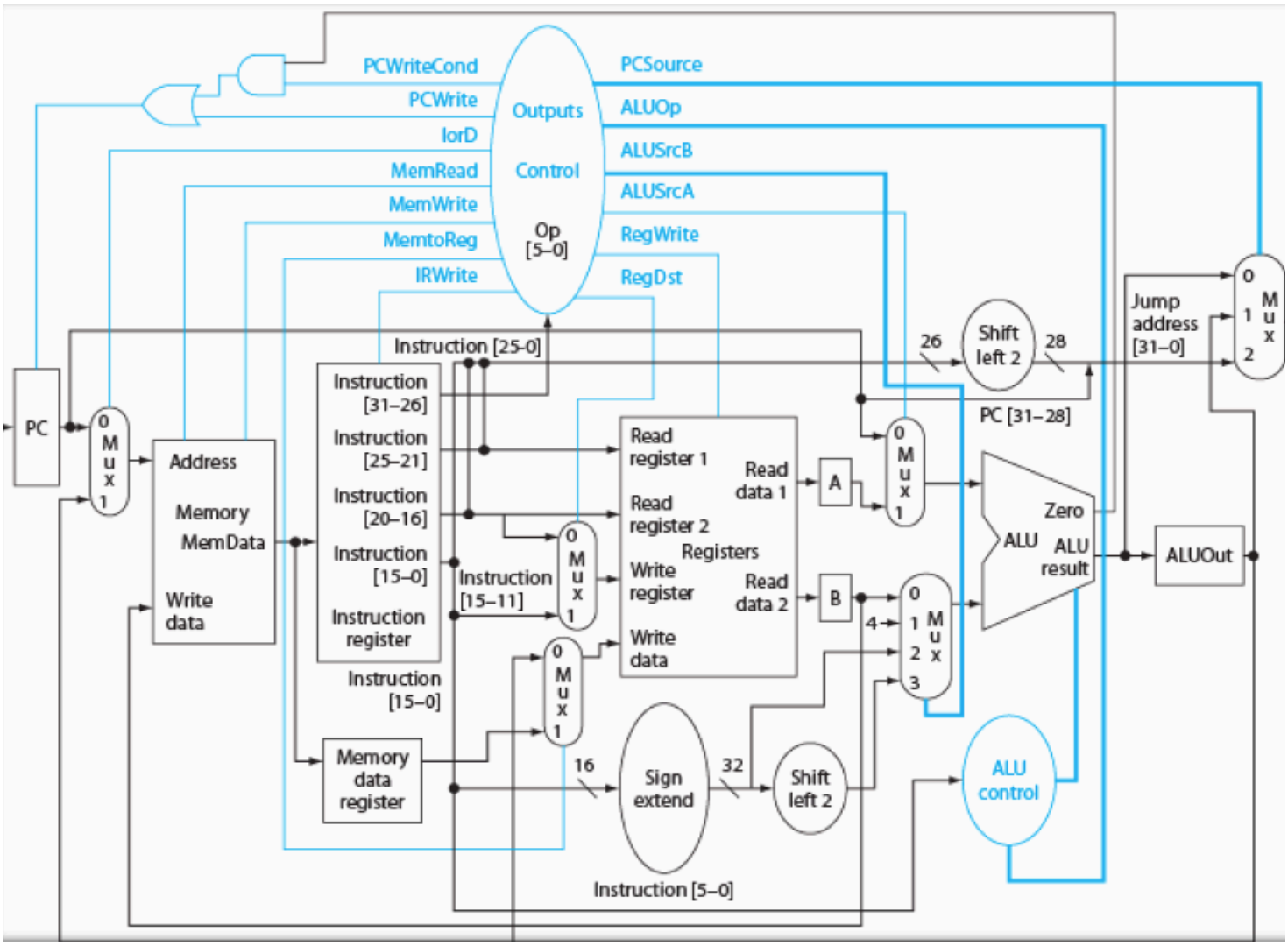
The custom set is chosen from the following ten options. We decided to use set 0.

The total set you need to design is the core set as above + a custom set designated for you as follows.
Student ID ending in:

1. BNE, ANDI
2. NAND, BNE
3. SUBI, ANDI
4. BNE, ORI
5. NAND, ORI
6. OR, ANDI
7. BNE, SUBI
8. NAND, ANDI
9. ORI, SRL
0. SUBI, ORI

Design

The group used the following design for the processor and data path:



Instruction	Instruction w/Reg	PC
lw $t3, 100($s2)	lw $11,200($18)	1000
sw $t4, 100($t3)	sw $12,100($11)	1004
add $s3, $t2, $s2	add $19,$10,$18	1008
beq $s5, $s2, 500	beq $21,$18,0[$500-0x0000100C]	100C
ori $t5,$t6,10	ori $13,$14,10	1010
SUBI $t3,$t4,10	SUBI $11,$12,10	1014

We start with PC(Program counter) in the multicycle datapath. We use PC starting from $1000 to $1014 in the test where we implement four type of instructions.

We have 2 type of instructions. The multi-cycle steps are:

1. Instruction Fetch
   IR=Memory[PC]
   PC=PC+4
   Here we send PC to memory as the address ,read instruction from memory ,write instruction into IR for use on next cycle and increment PC by 4. It uses ALU in this first cycle and sets control signals to send PC and constant 4 to ALU

2. Instruction Decode
   Here we decode and complete Register File Read

   A = Reg[IR[25-21]];

   B = Reg[IR[20-16]];

   ALUOut = PC + (sign-extend(IR[15-0]) << 2);

3. Execution
   - Memory reference:

   ALUOut = A + sign-extend(IR[15-0]);

   - Arithmetic-logical instruction:

   ALUOut = A op B;

   - Branch:

   if (A == B) PC = ALUOut;

4. Memory/ Completion
   Memory reference:

   MDR = Memory[ALUOut]; (load) or

   Memory[ALUOut] = B; (store)

   - Arithmetic-logical instruction:

   Reg[IR[15-11]] = ALUOut;

5. Read completion
   Load operation:

   Reg[IR[20-16]] = MDR;

OpCode[31:2	Rs[25:21]	Rt[20:16]	Rd[15:10]	shamt[10:6	function[5:0	
OpCode[31:2	Rs[25:21]	Rt[20:16]	imm[15:			He
OpCode[31:2	Adress[25:					x
(35)10001	(18)10010	(11)01011	(200)0000000011001000			0X8E4B00C8
101011	(11)01011	(12)01100	(100)0000000001100100			0XAD6C0064
(0)000000	(10)01010	(18)10010	(19)10011	(0)00000	(32)10000	0X01524C10
(4)000100	(21)10101	(18)10010	(125)0000000001111101			0X12B2007D
001101	(13)01101	(14)01110	(10)0000000000001010			0X35AE000A
111110	01100	01011	same			0X35AE000A

So, to design, we have to keep in mind the following facts:

- Instructions always do the first two steps
- Branch can finish in the third step Ff
- Arithmetic-logical can finish in the fourth step Ff
- Stores can finish in the fourth step
- Loads finish in the fifth step

Instruction Number of cycles

Branch : 3

Arithmetic-logical(add, subI): 4

Stores: 4

Loads: 5

There are 19 registers possible for Rs, Rt and Rd. One kind of register is always the same and another kind of register changes from time to time.

Mux Design:

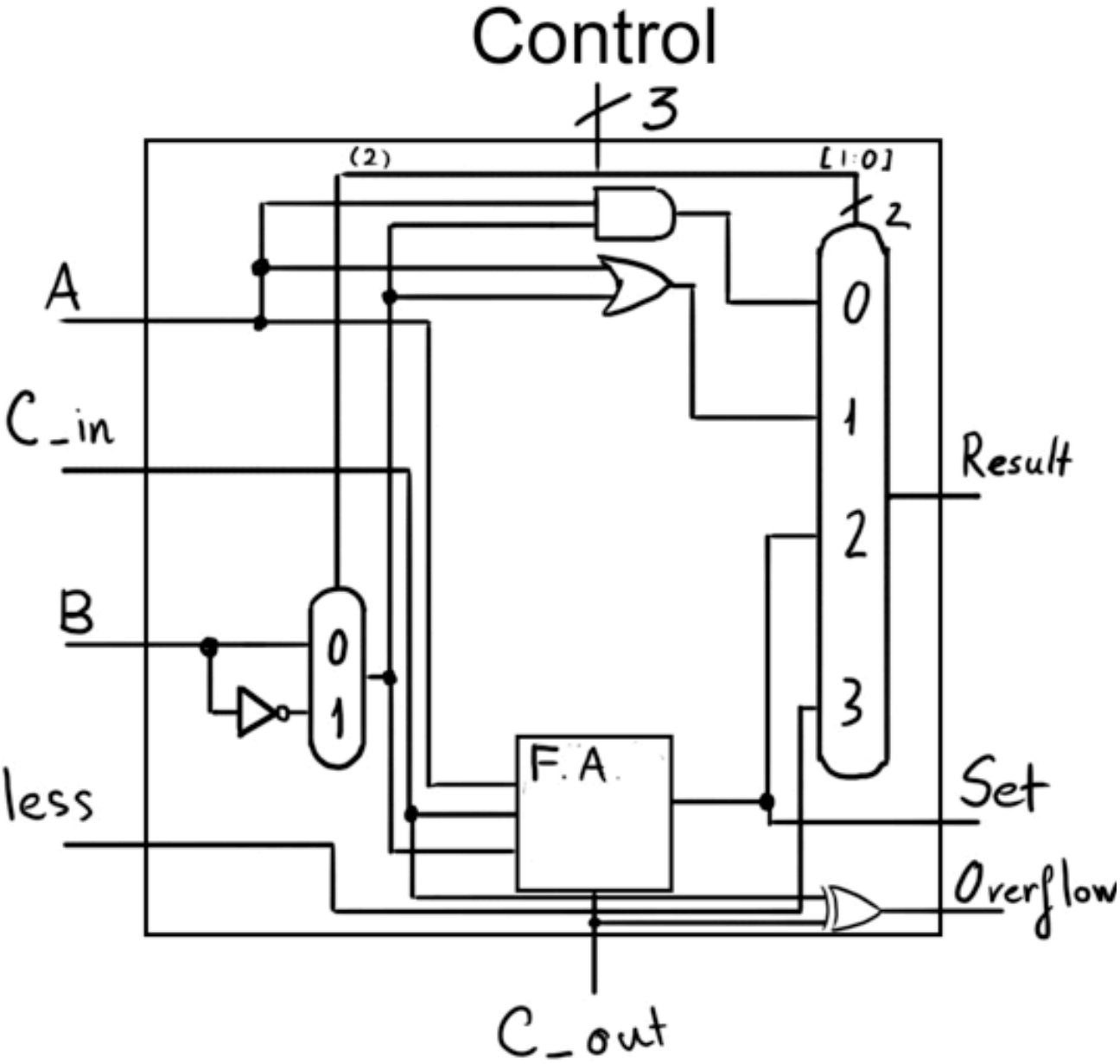
a. To decode instruction (find if destination is rt or rd for lw/add instruction), we need a mux taking 2 input 5 bit information. We have a 1bit control signal to check if it is “I-type” or “R-type”.
b. We need mux taking 4 input 32 bit information as shown in the diagram.

The list of instruction sets are given below:

lw	rt, address	100011	Rs	Rt	Offset [16 bits]		
sw	rt, address	101011	Rs	Rt	Offset [16 bits]		
add	rd, rs, rt	000000	Rs	Rt	Rd	00000	100000
beq	rs, rt, label	000100	Rs	Rt	Offset [16 bits]		
ori	rt, rs, imm	001101	Rs	Rt	Imm [16 bits]		
Subi	rt,rs,imm	111110	Rs	Rt	Imm [16 bits]		

ALU Design:

Single-bit ALU design will be like below:



The ALU performs add and subi. The inputs are 32 bit size and are unsigned. The result of the instruction is written to the output in the Memory Modify step or Write Back step for load instruction. Opcode helps to determine specific instruction. Overflow detection and carry support are avoided for complexity.

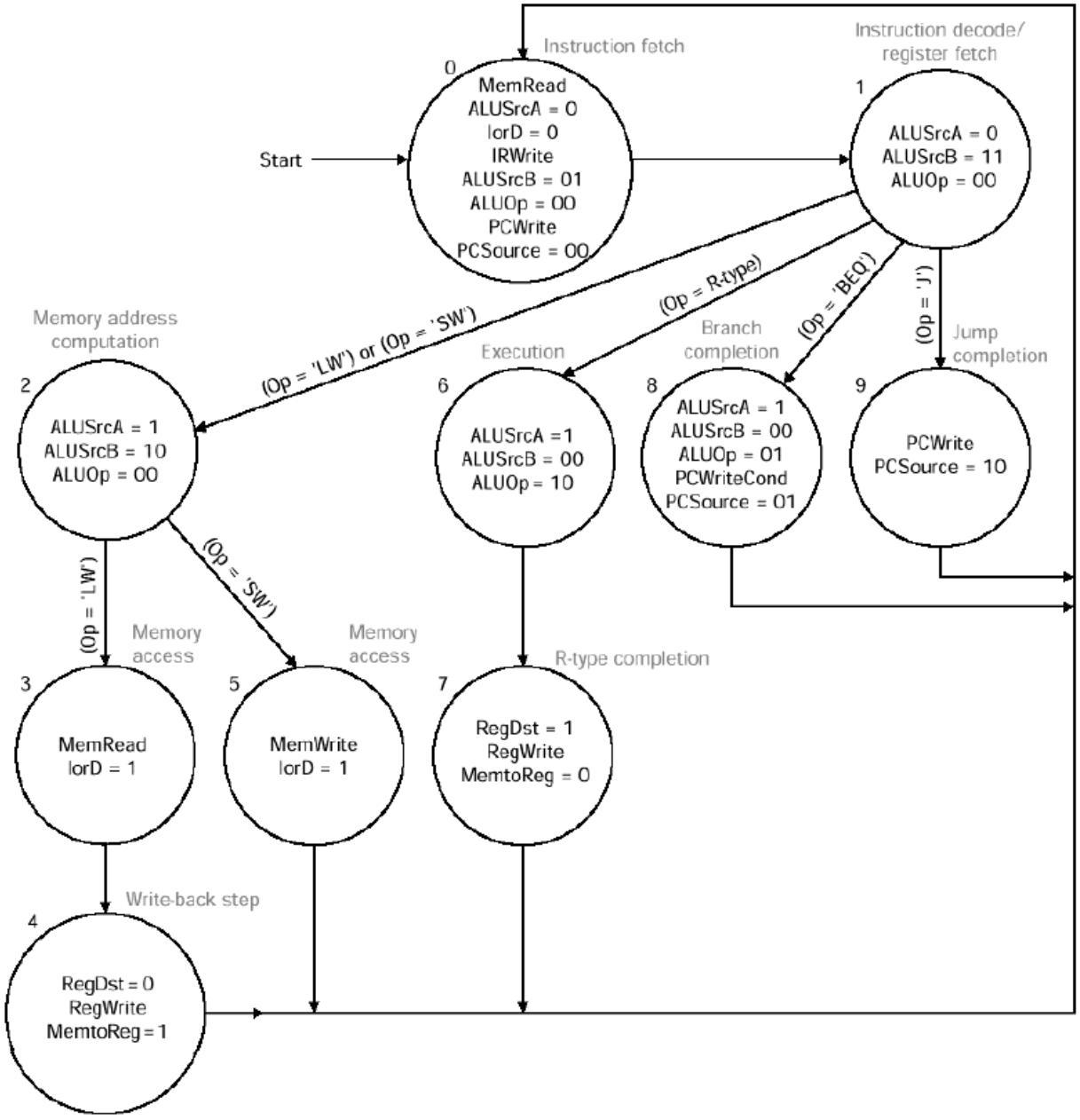
Memory:

The memory is 32 bit wide. The data is read from or written to the memory through the data bus. The control signal signals if data is to be written or read. The memory is byte addressable and the data word is usually divided into four bytes and is written accordingly.

Control:

Only instructions don't determine control signals, finite state machines are necessary for control signals. For the complexity, we won't perform this part in the design.

Different steps in the datapath are necessary to understand for us as they differ with different instructions. A detailed representation of the control signals via state diagram is shown below:



Implementation

We successfully designed the processor in order to support 3 types of instruction formats. The memory is assumed to be word addressable consisting of 32 bits. We have written the testbench too to design the processor. The designed processor can carry out all the necessary instruction specified.

ALU implementation:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is
  port ( INPUT_1: in std_logic_vector(31 downto 0);
                        INPUT_2: in std_logic_vector(31 downto 0);
                        CONTROL: in std_logic_vector(2 downto 0);
                        B_OUT: out std_logic;
                        F_OUT: out std_logic_vector(31 downto 0));
end entity ALU;

architecture behave of ALU is
begin
 F_OUT <=  std_logic_vector(unsigned(INPUT_1) + unsigned(INPUT_2)) when (CONTROL = "000")
       else
        std_logic_vector(unsigned(INPUT_1) - unsigned(INPUT_2)) when (CONTROL = "001")
       else
        INPUT_1 OR INPUT_2 when (CONTROL = "010") --or
       else
        INPUT_1 AND INPUT_2 when (CONTROL = "011") --and
       else
        x"FFFFFFFF";

 -- Assuming 1 for Branch and 0 for no branch
 B_OUT <=  '1' when (CONTROL = "100" OR CONTROL = "101") else
       '0';
end architecture behave;
```

Discussion: So, there are 2 inputs, 2-bit control signal and 2 outputs (1 for branch). Based on the control signal(100 or 101), we detect if we have to branch or not.

Datapath Implementation:

```
--Combination of Instruction Memory, Instruction Decode,
--Register, and ALU and misc. MUX's and Sign extenders, Data Memory, and pc generation
--Final version of the Datapath of the 32-bit processor
library ieee;
use ieee.std_logic_1164.all;

entity DATAPATH is
          port (PCIN: in std_logic_vector(31 downto 0);
                       PCOUT: out std_logic_vector(31 downto 0);
                       DATAOUT: out std_logic_vector(31 downto 0);
end entity DATAPATH;
```

Here we introduce the entity datapath which has 1 PC input, 1 PC output and 1 data output.

```
architecture behave of DATAPATH is

--Temp Signals used as connections between components
signal DFEED: std_logic_vector(31 downto 0);
signal DRS: std_logic_vector(4 downto 0);
signal DRT: std_logic_vector(4 downto 0);
signal DRD: std_logic_vector(4 downto 0);
signal DIMM: std_logic_vector(15 downto 0);
```

```
signal DSHMT: std_logic_vector(4 downto 0);
signal DFUNC: std_logic_vector(5 downto 0);
signal DTYPE: std_logic_vector(1 downto 0);
signal DM2OUT: std_logic_vector(4 downto 0);
signal DSE5OUT: std_logic_vector(31 downto 0);
signal DSE16OUT: std_logic_vector(31 downto 0);
signal DRD1: std_logic_vector(31 downto 0);
signal DRD2: std_logic_vector(31 downto 0);
signal DCTRLOUT: std_logic_vector(5 downto 0);
signal DADATA: std_logic_vector(31 downto 0);
signal DBDATA: std_logic_vector(31 downto 0);
signal DBRA: std_logic;
signal DFDATA: std_logic_vector(31 downto 0);
signal DMEMRD: std_logic_vector(31 downto 0);
signal DREGWRITE: std_logic_vector(0 downto 0);
signal DREGVALUE: std_logic_vector(31 downto 0);
```

Then we combine the signals for all the components throughout the processor.

```
component Instruction_Memory
        port (PC: in std_logic_vector(31 downto 0);                          --32-bit instruction
                        INSTRUCTION: out std_logic_vector(31 downto 0)); --32-bit instruction(hardcoded in)
end component;
```

The instruction memory component has 32 bit input and output

```
component ID
        port ( FEED: in std_logic_vector(31 downto 0);
      OPCODE: out std_logic_vector (5 downto 0);
                RS: out std_logic_vector(4 downto 0);
                RT: out std_logic_vector(4 downto 0);
                RD: out std_logic_vector(4 downto 0);
                SHMT: out std_logic_vector(4 downto 0);
                FUNC: out std_logic_vector(5 downto 0);
                IMM: out std_logic_vector(15 downto 0));
end component;
```

Then we decode the instruction and the port includes Rs, Rt, RD, shift amount, function code and immediate value.

```
component DRegister
        port (RR1: in std_logic_vector(4 downto 0);          --5-bit Read Reg. 1
                RR2: in std_logic_vector(4 downto 0);                --5-bit Read Reg. 2
                WR: in std_logic_vector(4 downto 0);                 --5-bit Write Register
                WD: in std_logic_vector(31 downto 0);                --32-bit Write Data
                Control: in std_logic_vector(5 downto 0);    --6-bit Opcode/Function
                RD1: out std_logic_vector(31 downto 0);      --32-bit output1
                RD2: out std_logic_vector(31 downto 0);      --32-bit output2
                CTRL_OUT: out std_logic_vector(5 downto 0)); --Control Output to pass through the control values for alu
operation
end component;

component ALU
  port ( INPUT_1: in std_logic_vector(31 downto 0);
                INPUT_2: in std_logic_vector(31 downto 0);
                CONTROL: in std_logic_vector(2 downto 0);
                B_OUT: out std_logic;
                F_OUT: out std_logic_vector(31 downto 0));
end component;

component Data_Memory
  port (ADDR: in std_logic_vector(31 downto 0);        --32-bit Address location
          WD: in std_logic_vector(31 downto 0);                --32-bit data
                Control: in std_logic_vector(5 downto 0);    --LW for Read SW for Write
                RD: out std_logic_vector(31 downto 0));      --32-bit data at specific address, only going to output when lw
end component;
```

```
component Mux2
	port (ZERO: in std_logic_vector(4 downto 0);
		ONE: in std_logic_vector(4 downto 0);
		CTRL: in std_logic_vector(1 downto 0);
		OUTPUT: out std_logic_vector(4 downto 0));
end component;

component MuxIn32
	port (ZERO: in std_logic_vector(31 downto 0);
		ONE: in std_logic_vector(31 downto 0);
		CTRL: in std_logic_vector(1 downto 0);
		OUTPUT: out std_logic_vector(31 downto 0));
end component;

--component Mux2_32_1                                                    --Mux for Write Data
	--port (ZERO: in std_logic_vector(31 downto 0);          --Zero Input for 32-bit mux
		-- ONE: in std_logic_vector(31 downto 0);          --One Input for 32-bit mux
		--CTRL: in std_logic_vector(5 downto 0);           --Control signal
		--OUTPUT: out std_logic_vector(31 downto 0));      --32-bit output
-- end component;

component Mux3                                                         --Mux for A_Data of
ALU
	port (ZERO: in std_logic_vector(31 downto 0);
		ONE: in std_logic_vector(31 downto 0);
		TWO: in std_logic_vector(31 downto 0);
		CONTROL: in std_logic_vector(1 downto 0);
		OUTPUT: out std_logic_vector(31 downto 0));
end component;

component Sign_Extend_5                                                --Takes 5 bit number and
truncattes with 0
	port (INPUT: in std_logic_vector(4 downto 0);          --5-bit Input
		OUTPUT: out std_logic_vector(31 downto 0));    --32-bit ouput
end component;

component Sign_Extend_16                                               --Takes 16 bit number and trancates
with 0
	port (INPUT: in std_logic_vector(15 downto 0);         --16-bit Input
		OUTPUT: out std_logic_vector(31 downto 0));    --32-bit output
end component;

component PC_Calc
  port (PC_IN: in std_logic_vector(31 downto 0);     --32-bit PC
		IMM: in std_logic_vector(31 downto 0);                 --32-bit Immidiate value
		Branch: in std_logic;                                  --Branch Control Signal from ALU
		Control: in std_logic_vector(5 downto 0);      --Control signal from Instruction Mem
		X_Type: in std_logic_vector(1 downto 0);       --Type of instruction: Going to be used for Jump Instruction "01"
		PC_OUT: out std_logic_vector(31 downto 0));    --32-bit PC output
end component;
begin

Instruction_Mem: Instruction_Memory port map(PC => aPC,
                                             INSTRUCTION => TEMP_INST);

Decode: Instruction_Decode port map(INST => TEMP_INST,
                                    RS => TEMP_RS,
                                    RT => TEMP_RT,
                                    RD => TEMP_RD,
                                    IMM => TEMP_IMM,
                                    SHMT => TEMP_SHMT,
                                    FUNCT => TEMP_FUNCT,
```

```
                                                    X_TYPE => TEMP_TYPE);

Rd_Select: Mux_2 port map(ZERO => TEMP_RT,
                          ONE => TEMP_RD,
                          CTRL => TEMP_TYPE,
                          OUTPUT => TEMP_M2OUT);

Reg_Data: Register_Data port map(RR1 => TEMP_RS,
                                 RR2 => TEMP_RT,
                                 WR => TEMP_M2OUT,
                                 WD => TEMP_Reg_Value,
                                 Control => TEMP_FUNCT,
                                 RD1 => TEMP_RD1,
                                 RD2 => TEMP_RD2,
                                 CTRL_OUT => TEMP_CTRL_OUT);

Shmt_Extend: Sign_Extend_5 port map(INPUT => TEMP_SHMT,
                                    OUTPUT => TEMP_SE5OUT);

Imm_Extend: Sign_Extend_16 port map(INPUT => TEMP_IMM,
                                    OUTPUT => TEMP_SE16OUT);


A_Data_Select: Mux_3 port map(ZERO => TEMP_RD1,
                              ONE => TEMP_SE16OUT,
                              TWO => TEMP_SE5OUT,
                              CTRL => TEMP_FUNCT,
                              OUTPUT => TEMP_A_DATA);


B_Data_Select: Mux2_32 port map(ZERO => TEMP_RD2,
                                ONE =>  TEMP_SE16OUT,
                                CTRL => TEMP_FUNCT,
                                OUTPUT => TEMP_B_DATA);

ALU: ALU_32_Bit port map(A_DATA => TEMP_A_DATA,
                         B_DATA => TEMP_B_DATA,
                         CONTROL => TEMP_FUNCT,
                         BRANCH => TEMP_BRANCH,
                         F_DATA => TEMP_F_DATA);

Data_Mem: Data_Memory port map(ADDR => TEMP_F_DATA,
                               WD => TEMP_RD2,
                               Control => TEMP_FUNCT,
                               RD => TEMP_MEM_RD);


Write_DataMux: Mux2_32_1 port map(ZERO => TEMP_MEM_RD,
                                  ONE => TEMP_F_DATA,
                                  CTRL => TEMP_FUNCT,
                                  OUTPUT => TEMP_Reg_Value);

PC: PC_Calc port map(PC_IN =>aPC,
                     IMM => TEMP_SE16OUT,
                     Branch => TEMP_BRANCH,
                     Control => TEMP_FUNCT,
                     X_TYPE => TEMP_TYPE,
                     PC_OUT => aPC_OUT);

        aF_DATA <= TEMP_F_DATA;
        aTEMP_TYPE <= TEMP_TYPE;
end architecture behave;
```

VI. Results

We implemented the example instructions from the project document

lw $t3, 200($s2)

Binary representation:

(35)10001	(18)10010	(11)01011	(200)0000000011001000	0X8E4B00C8

Initial condition:

```
PC          = 1000
Memdata     = 8E4B00C8
```

The instruction load word is meant to load the word stored at 200($18)(With the initialized values this turns out to be memory location $146 which has the value $1357 stored in it) and store it in Reg 11. In this case we can see in the picture attached that the new value of Register 11 was changed to $1357. It can also be seen that the PC out value shows that it should be a sequential output, incrementing the PC value to PC+4 to a value of $1004.

sw t4,100($t3) or sw $12,100($11)

The instruction Store word is meant to store the word at Reg 12 in the memory location designated by 100($11) (With initialized values, this memory location is $2F4). It can be seen in the picture that the value of Reg 12 is now the same as the value at memory location $2f4.Store word also increments the PC value sequentially to PC+4 to a value of $1008.

add $t3,$t2,$s2 or add $11,$10,$18

The instruction Add adds the value of Reg 10 and Reg 18, and puts it Reg 11. The initial values of Reg 10 was $130 and Reg 18 was $120, thus when added results in $250. The PC value is incremented sequentially to PC+4 to a value of $100C.

beq $s5,$s2,500 or beq $21,$18,500[$500-$100C]

The instruction branch if equal will branch to the branch address (which is calculated by taking the branch address, subtracting the current PC, and the adding that to the PC value) if the value in Reg 21 and Reg 18 are equal. In this case the value of Reg 13 is $120 and Reg 18 is $150, so they are not equal thus resulting in just a sequential increment of the PC to PC+4 to a value of $1010.

ori $t5,$t6,10 or ori $13,$14,10

The instruction Immediate Or takes the logical OR between Reg 14 and the immediate value 10 and places the result in Reg 13. In this case the value of Reg 14 is $140, when this is OR'ed with 10 the result is $14A. This can be seen as the result placed in Reg 13 in the results picture. The PC is also sequentially incremented to PC+4 to a value of $1014.

Conclusions

The output results and simulations were mostly correct. The team had a clear idea how each type of instruction would operate with the help of book reference and it helped to finish the coding step by step. The component declarations and set up procedure are discussed heavily in the report. Also the test bench results are explained. The PC values in each step of the programs are shown and instruction type procedures are shown too. The group became better in working with VHDL to design complex processors.