MIPS Single-Cycle Final Project

Abdelrahman Ahmed Nabil Ahmed Mahmoud

221000309 - Class(3)

Supervised by:

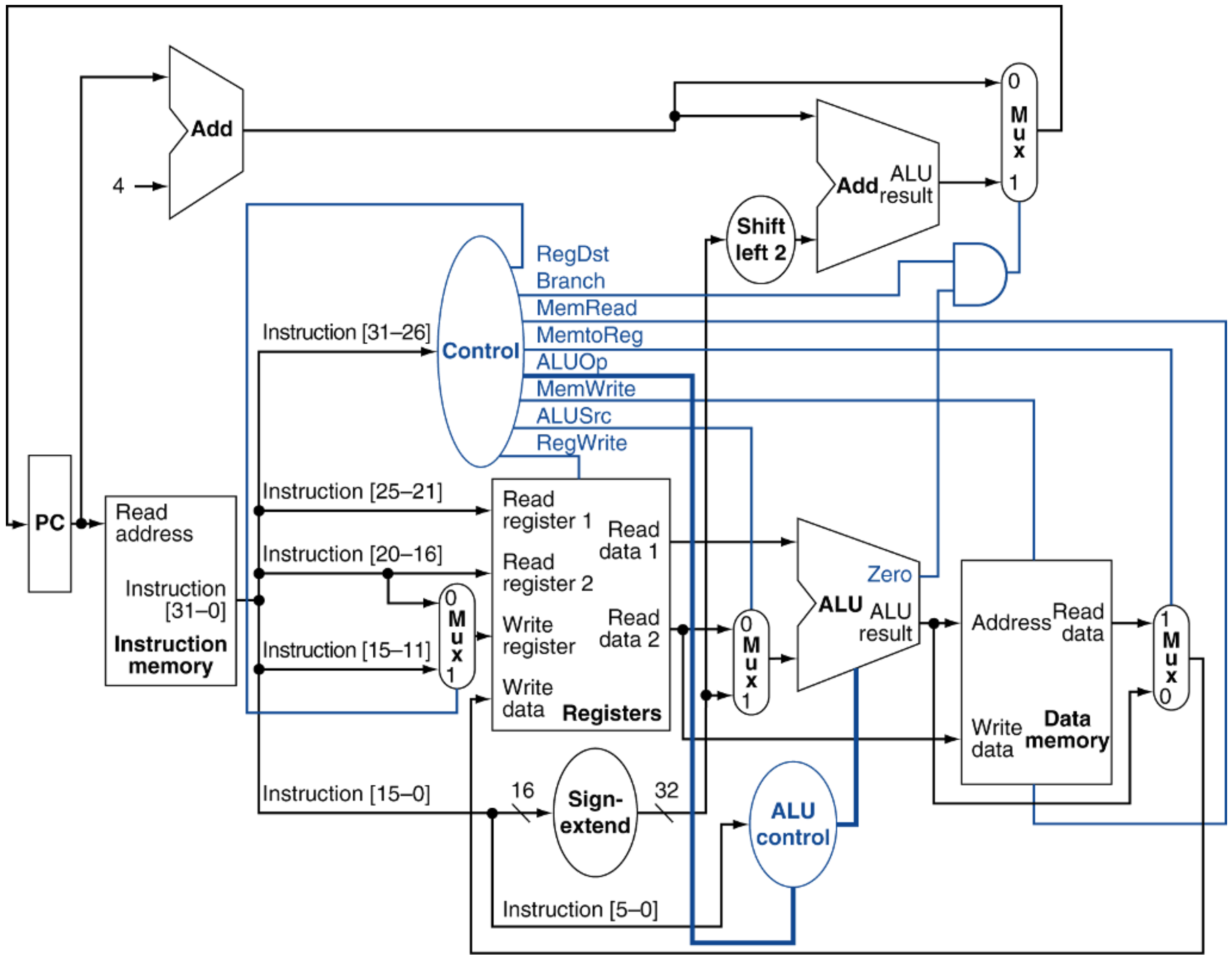
Dr/Marwa ElShenawy

Eng/Rama Hamdy

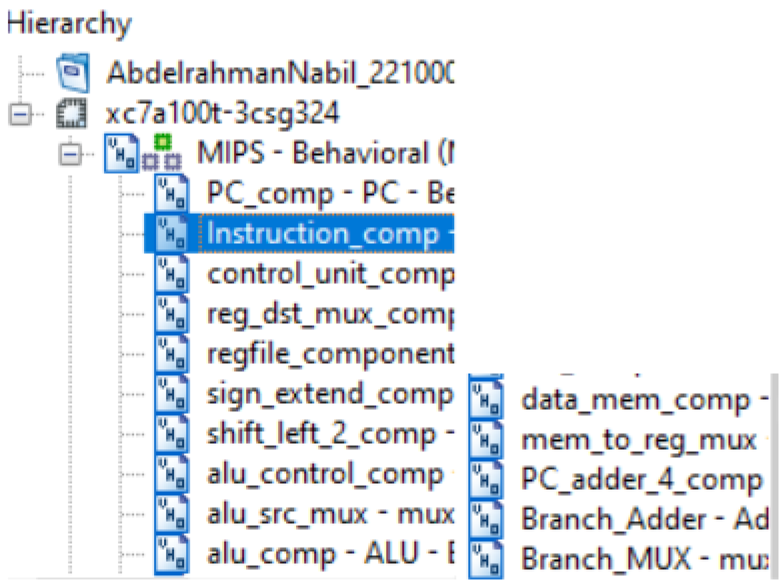
Project Overview

This project tackles VHDL implementation of a simple MIPS processor which executes each instruction in a singular clock cycle.

This processor is designed to execute two main types of instruction:
1- R-Type
2- I-Type



All used components:


Hierarchy
AbdelrahmanNabil_22100(
xc7a100t-3csg324
MIPS - Behavioral (I
PC_comp - PC - Be
Instruction_comp -
control_unit_comp
reg_dst_mux_comp
regfile_component
sign_extend_comp
shift_left_2_comp -
alu_control_comp
alu_src_mux - mux
alu_comp - ALU - I
data_mem_comp -
mem_to_reg_mux
PC_adder_4_comp
Branch_Adder - Ad
Branch_MUX - mu

First and foremost, a common starting point between all instructions is

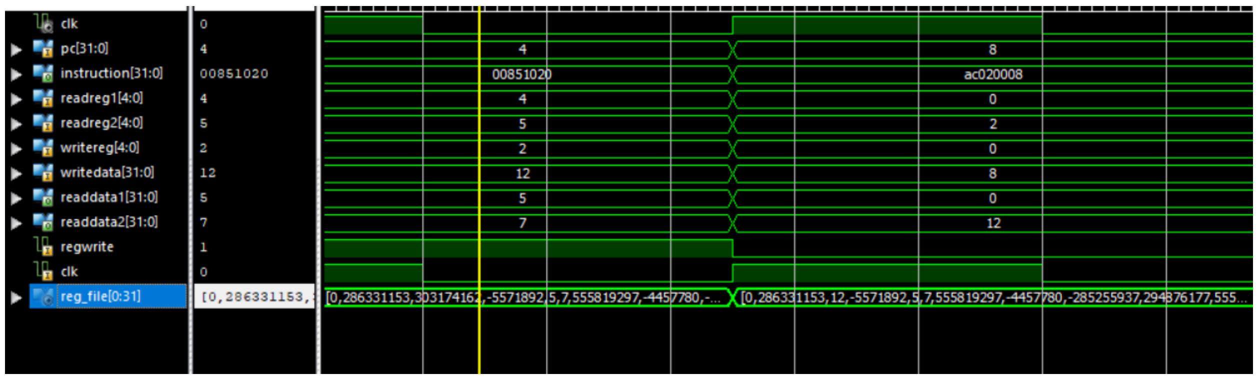
- Fetching the instruction : At the rise of the clock cycle, the PC is passed as input to the instruction memory, where it serves as an address to fetch my instruction. The output is the actual 32 bits that represent the instruction itself.

Second of all the function is decoded to:

- Opcode to decide control signals
- Inputs to the register file(rs,rt,rd)
- Input to sign extend (in case of immediate value)
- Input to ALU Control (function field)

R-Type Instructions

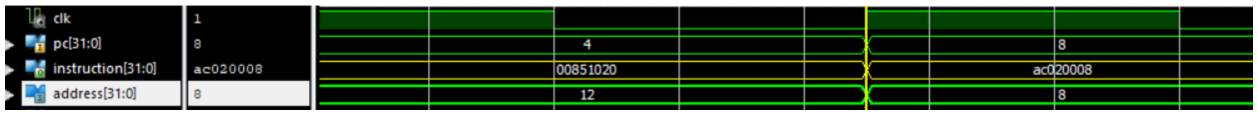
For R-type instructions such as *add*, the following occurs:
1- rs and rt are read registers
2- rd is selected to be the write register from the mux to be the write register as the control signal RegDst is 1
3-rs and rt are inputs to the ALU, where rt is selected and not an immediate value due to the value of the control signal ALUSrc.
4- Finally the result is calculated via the ALU
5- It is selected in the final mux to be the write data, and then it is written back to the register.



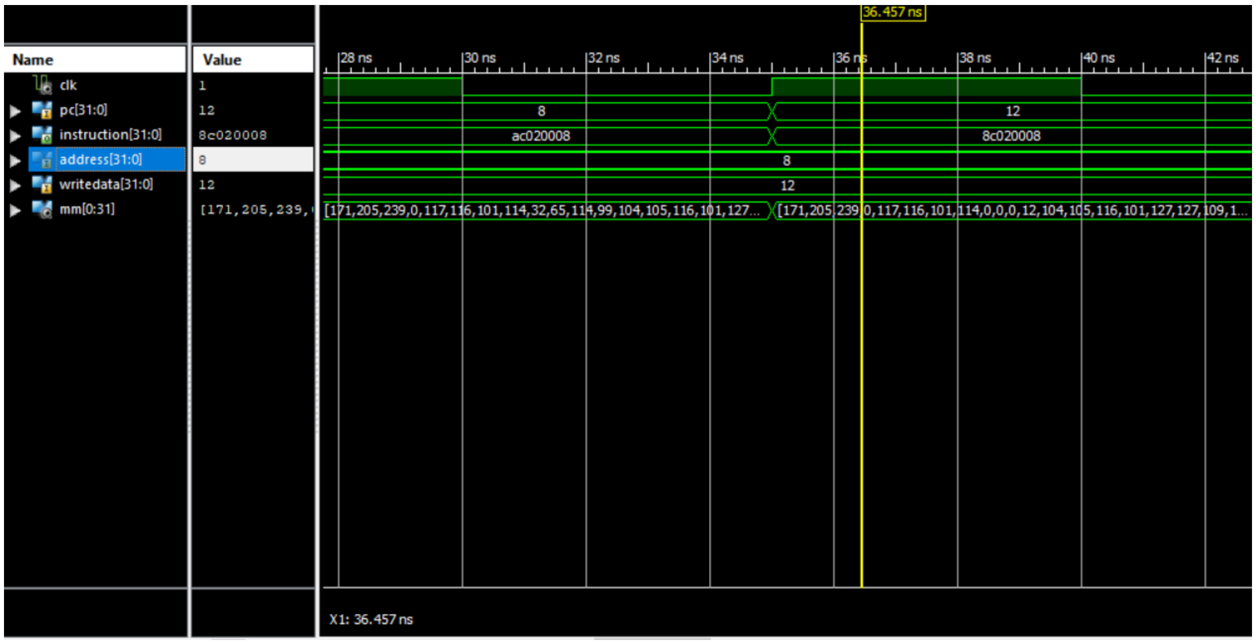
clk 0
pc[31:0] 4 | 4 | 8
instruction[31:0] 00851020 | 00851020 | ac020008
readreg1[4:0] 4 | 4 | 0
readreg2[4:0] 5 | 5 | 2
writereg[4:0] 2 | 2 | 0
writedata[31:0] 12 | 12 | 8
readdata1[31:0] 5 | 5 | 0
readdata2[31:0] 7 | 7 | 12
regwrite 1
clk 0
reg_file[0:31] [0,286331153,303174162,-5571892,5,7,555819297,-4457780,-... | [0,286331153,12,-5571892,5,7,555819297,-4457780,-285255937,294376177,555...

In this screenshot , data from registers (4 and 5) are added and written into register(2). The output is (5+7 = 12) and it is evident in the next clock cycle where index(2) in the register file has in fact changed.

Memory Manipulation Functions

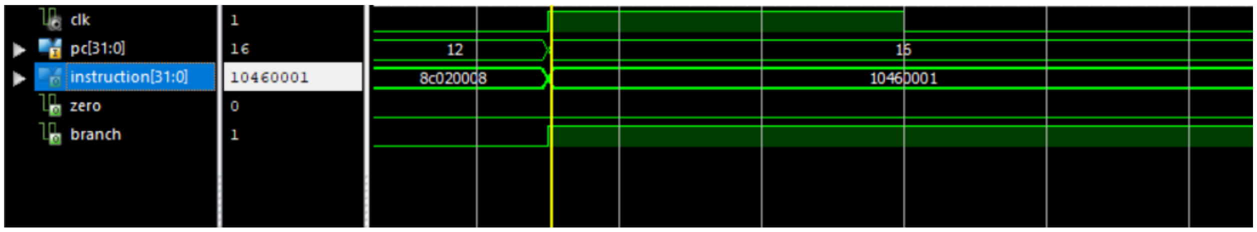


The address is calculated by adding rs to the immediate value. In this case zero register containing zero is added to 8, giving the address of 8.



As can be observed here, the indexes 8 through 11 were used for writing the data, which is because of the byte-addressable memory.

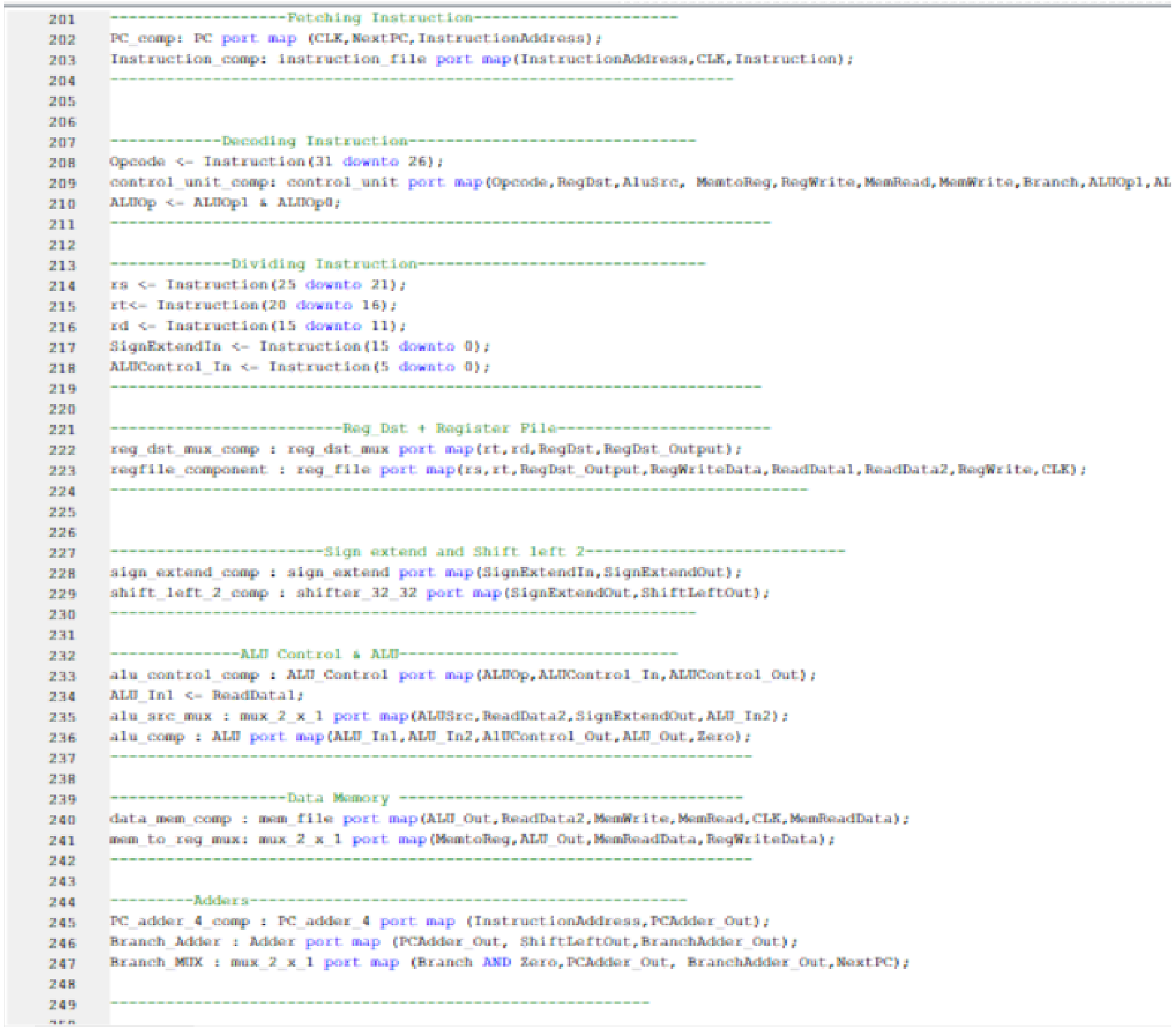
Branch Instructions



For the branch instruction, it was shown that the branch control signal is in fact 0 ; however , since zero flag from ALU isnt 1 the condition isnt fulfilled completely in this case, and no branch occurs. When tested with equal values, it was shown that the PC skipped the next +4 and moved straight to 20 instead of 16.

Mapping Code

Attached here is the picture for the port mapping code in order to define the MIPS datapath, using the signals defined in the MIPS architecture, to connect each component to the other.



```
-------------------Fetching Instruction----------------------
PC_comp: PC port map (CLK,NextPC,InstructionAddress);
Instruction_comp: instruction_file port map(InstructionAddress,CLK,Instruction);
------------------------------------------------------------------



------------Decoding Instruction-------------------------------
Opcode <= Instruction(31 downto 26);
control_unit_comp: control_unit port map(Opcode,RegDst,AluSrc, MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp1,AL
ALUOp <= ALUOp1 & ALUOp0;
----------------------------------------------------------------------

-------------Dividing Instruction-------------------------------
rs <= Instruction(25 downto 21);
rt<= Instruction(20 downto 16);
rd <= Instruction(15 downto 11);
SignExtendIn <= Instruction(15 downto 0);
ALUControl_In <= Instruction(5 downto 0);
---------------------------------------------------------------------

-------------------------Reg_Dst + Register File----------------------
reg_dst_mux_comp : reg_dst_mux port map(rt,rd,RegDst,RegDst_Output);
regfile_component : reg_file port map(rs,rt,RegDst_Output,RegWriteData,ReadData1,ReadData2,RegWrite,CLK);
--------------------------------------------------------------------------



-----------------------Sign extend and Shift left 2---------------------------
sign_extend_comp : sign_extend port map(SignExtendIn,SignExtendOut);
shift_left_2_comp : shifter_32_32 port map(SignExtendOut,ShiftLeftOut);
---------------------------------------------------------------

--------------ALU Control & ALU------------------------------
alu_control_comp : ALU_Control port map(ALUOp,ALUControl_In,ALUControl_Out);
ALU_In1 <= ReadData1;
alu_src_mux : mux_2_x_1 port map(ALUSrc,ReadData2,SignExtendOut,ALU_In2);
alu_comp : ALU port map(ALU_In1,ALU_In2,AlUControl_Out,ALU_Out,Zero);
---------------------------------------------------------------------

-------------------Data Memory -------------------------------------
data_mem_comp : mem_file port map(ALU_Out,ReadData2,MemWrite,MemRead,CLK,MemReadData);
mem_to_reg_mux: mux_2_x_1 port map(MemtoReg,ALU_Out,MemReadData,RegWriteData);
---------------------------------------------------------------------

---------Adders----------------------------------------------
PC_adder_4_comp : PC_adder_4 port map (InstructionAddress,PCAdder_Out);
Branch_Adder : Adder port map (PCAdder_Out, ShiftLeftOut,BranchAdder_Out);
Branch_MUX : mux_2_x_1 port map (Branch AND Zero,PCAdder_Out, BranchAdder_Out,NextPC);

----------------------------------------------------------
```