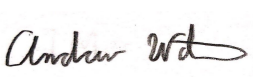
THE MIPS CPU

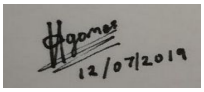
— AN ECE 485 PROJECT —

ANDREW WOLTMAN
&
CLIVE GOMES

I acknowledge all works including figures, codes and writings belong to me and/or persons who are referenced. I understand if any similarity in the code, comments, customized program behavior, report writings and/or figures are found, both the helper (original work) and the requestor (duplicated/modified work) will be called for academic disciplinary action.

SIGNATURES





Abstract: The goal of this document is to provide detailed implementation and documentation of a custom 32 bit Reduced Instruction Set Computer (RISC) processor. The write-up first touches upon the overall design of the datapath. After introducing the datapath, each component in the datapath is introduced. Descriptions and analyses (using test benches to produce simulation results) of each of these components were included. The document then goes on to explain how each of the components are added one at a time until the processor is fully put together. It goes on to explain how it can be used to implement a full set of instructions. The document ends by briefly discussing the key takeaways from this implementation, along with recommendations for improvements in future designs.

Keywords: CPU ; R-Type ; I-Type ; J-Type ; datapath ; RISK ; MIPS

Introduction

The CPU, known as the central processing unit, performs arithmetic, logic, controlling and I/O operations that are specified by a set of instructions. The MIPS Instruction Set is one of the most common ones in use, and has been the basis of this class. This Instruction Set supports three types of instructions:

1. R-Type Instructions: These are the ones that make use of three registers
2. I-Type Instructions: These use two register and an immediate value
3. J-type Instructions: These do not use any registers.

A CPU that is able to run the MIPS instruction set is comprised of : (1) a memory component; (2) an instruction register component; (3) a registers block; (4) one or more control units; (5) an arithmetic logic unit (ALU); and (6) multiple multiplexores.

The CPU is the brain of every computer, it runs all of the different programs that are assembled and stored in main memory. A CPU is designed by first identifying the instruction set that it will need to run. Once done, the three main components of the CPU — the Instruction Register, Registers and the ALU — need to be designed first. With just these three components, one can implement a processor that when given an instruction can decode and execute it in a single cycle. Next, the memory, memory controller, PC and control unit can be introduced to build a multi cycle CPU.

What's Included?

This project includes the implementation of a CPU that supports a RISC instruction set (MIPS). To support this implementation, seven components, the Instruction Register, Memory (Cache) associated with a Memory Controller, Registers Block, ALU, Control Unit, and the PC Unit (or PCU), were implemented. The reduced instruction set this CPU supports include the following instructions:

1. Load Word → eg. lw $t1, 0x4($t0)
2. Add → add $t2, $t3, $t4
3. Subtract → sub $t1, $t5, $t7
4. logical AND → and $t3, $t3, $t4
5. logical OR → or $t3, $t4, $t2
6. Jump → j LABEL
7. Branch if equal → beq $t1, $t2, LABEL
8. Add immediate → addi $t3, $t2, 0x4
9. Store Word → sw $t7, 0x0($t1)
10. Set on Less Than → slt $t3, $t2, $t1
11. Logical NOR → nor $t2, $t7, $t5
12. Logical AND immediate → andi $t2, $t5, 0xFFE
13. Logical OR immediate → ori $t3, $t3, 0x1

Overall Design

The MIPS datapath shown in the figure below was used as the base for designing the CPU in this project. To an extent, the basic components were the same, though slight modifications were made to each, and new components were added. This was done in order to keep the design as simple as possible since the project only required compatibility with a limited number of instructions rather than entire MIPS Instruction Set Architecture.

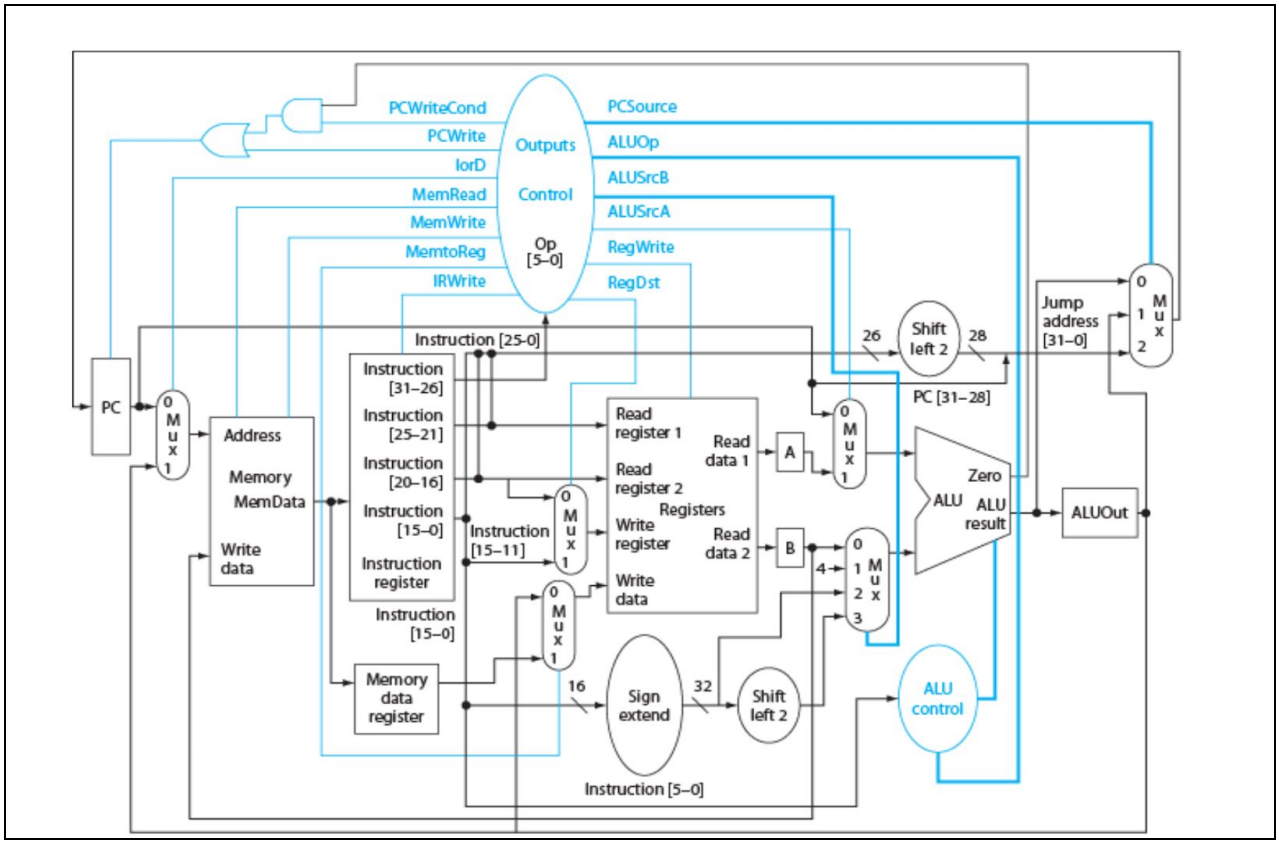


Figure 1: MIPS Multicycle Datapath

Instruction and data memory were grouped into a single entity, similar to the MIPS datapath above. A simple on-board memory (Cache) — comprising of an array of variables with read/write access -- was implemented to be able to store program instructions and data. A Memory Controller unit was also implemented to route data between the CPU and this Cache. Note, however, that the term "cache" here is just used to represent an on-board memory space owned by the CPU; the cache built here doesn't exhibit the properties of a typical "cache".

The instruction register (IR) was modified to also perform the instruction decode (ID) stage of execution. This was possible due to a limited instruction set, as well as the fact that there was no overlap between any of the opcodes of func codes of the instructions. Accordingly, using a simple if-else block, the opcodes and func codes were mapped to a four-bit value that determined the instruction being executed. The table listing these values is given in the appropriate section later. Another change to the IR was that it now also performed the sign-extend operation; this was done to limit the total number of components in the CPU.

Next, the registers block and the ALU were not changed drastically from the MIPS design. The only change was that all the external MUXs were combined into these two entities. The control signals for these MUXs were also routed to the register block and ALU. The ALU

Control Unit was also not implemented in this project. Instead, it's operation was incorporated into the single Main Control Unit (MCU).

Finally, all PC update operations — including the jump, branch, as well as the default PC+4 operation — were performed by a newly designed component, the PC Unit, or PCU. Accordingly, this entity receives inputs from the IR and the PCU itself (the old PC value), as well as control signals from the MCU.

With that, all components of the CPU have been mentioned. A schematic for the entire datapath is given below.

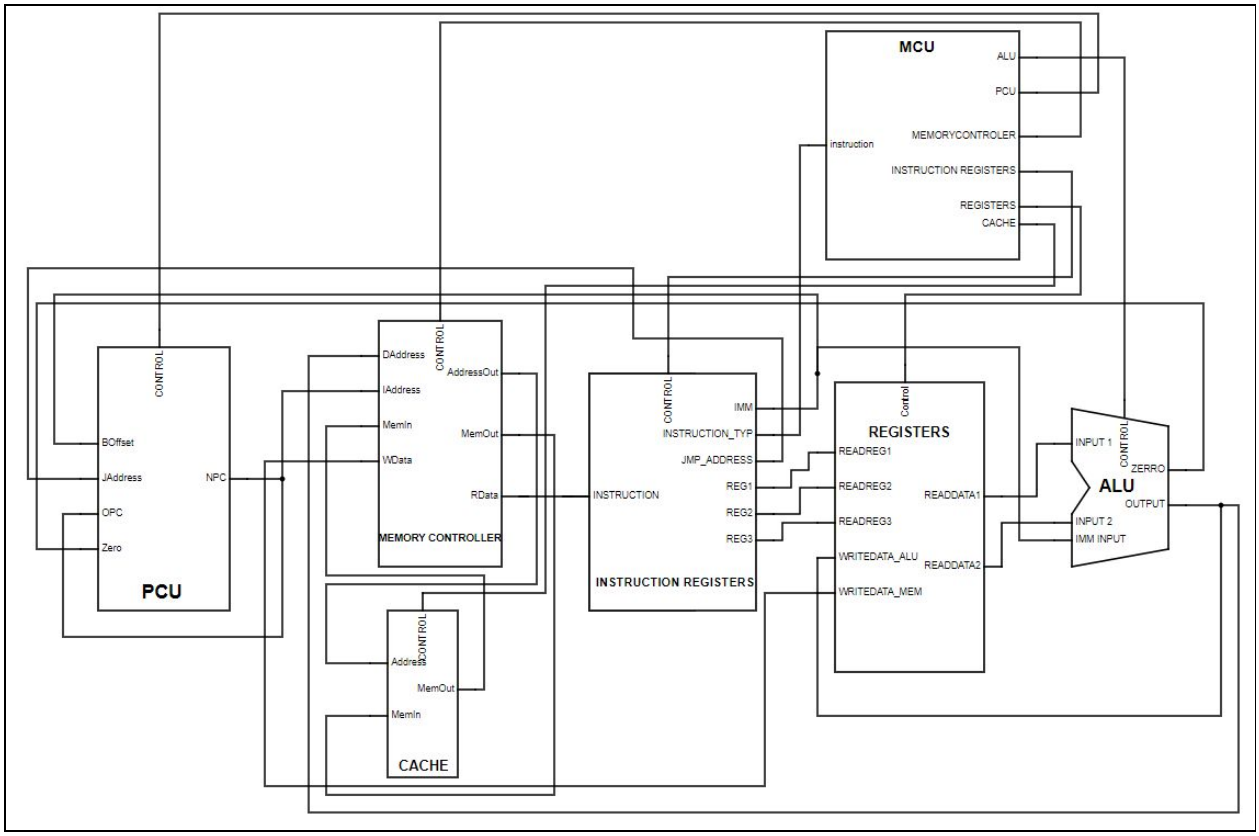


Figure 2: Rough Schematic of Proposed Datapath

Building The Components

Now that the overall design — in terms of the major components and their functions — has been described, this section gives detailed explanations of how each of these components were built, also including tests and analyses for each, starting with the primary component of a CPU — the ALU.

ALU

Description

The 32 bit ALU was designed to perform the arithmetic and logical operations performed by the CPU. This block uses the following 2 control signals:

1. ALUSrc → ALUSrc is used to select the correct value to be used in the calculation of OUTPUT and ZERO. This value can either be INPUT_IMM or INPUT_B.
2. ALUOp → ALUOp is then used to select the correct operation to be performed (ADD, SUBTRACT, etc.).

The ZERO output is always calculated and used in the PCU. The schematic for the ALU is given below:

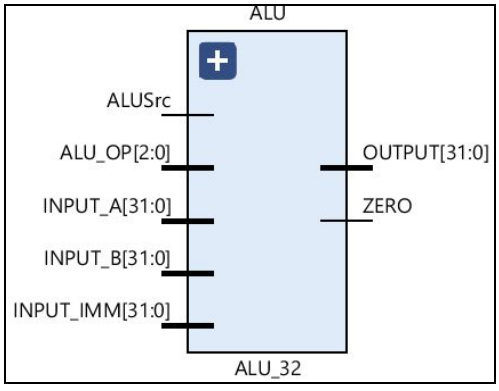


Figure 3: ALU Block View

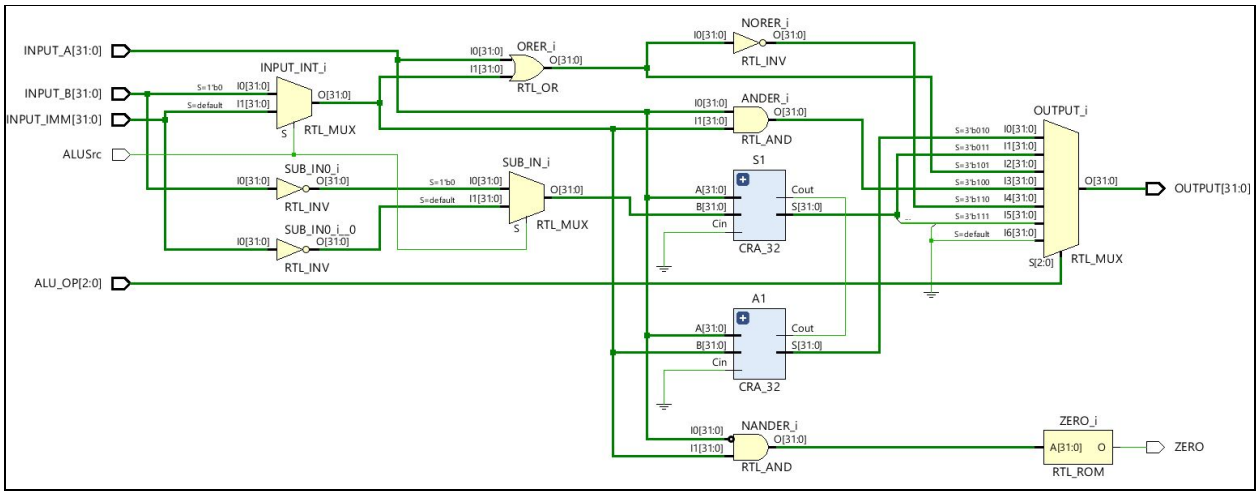


Figure 4: ALU Internal View

Testing

To test the ALU, a simple testbench program that tests each of the possible outputs (ADD, SUBTRACT, OR, AND, NOR, SLT and OTHERs) using both INPUT_B and INPUT_IMM was created. After every call to "wait", the source is changed, and every two such "wait" calls, the function is changed. A screenshot of the testbench code is given below.

```
stimulus: process
begin INPUT_A <= "10000110001100011000110000010001"; -- INPUT A WILL REMAIN THE SAME FOR ALL TESTS
      INPUT_B <= "00000000000000000000000000000000"; -- INPUT B WILL REMAIN THE SAME FOR ALL TESTS
      INPUT_IMM <= "01111001110011100111001111101110"; -- IMM INPUT WILL REMAIN THE SAME FOR ALL TESTS
      ALU_OP <= "010"; -- ADD
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns; -- WAIT
      ALU_OP <= "011"; -- SUBTRACT
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns; -- WAIT
      ALU_OP <= "101"; -- OR
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns; -- WAIT
      ALU_OP <= "100"; -- AND
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns; -- WAIT
      ALU_OP <= "110"; -- NOR
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns; -- WAIT
      ALU_OP <= "111"; -- SLT
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns; -- WAIT
      ALU_OP <= "000"; -- OTHER
      ALUSrc <= '0'; -- INPUT B IS USED
      wait for 50ns; -- WAIT
      ALUSrc <= '1'; -- IMM INPUT IS USED
      wait for 50ns;
  wait;
end process;
```

Figure 5: ALU Test Bench

Results & Analyses

On simulating the ALU using the given testbench, the following output was obtained.

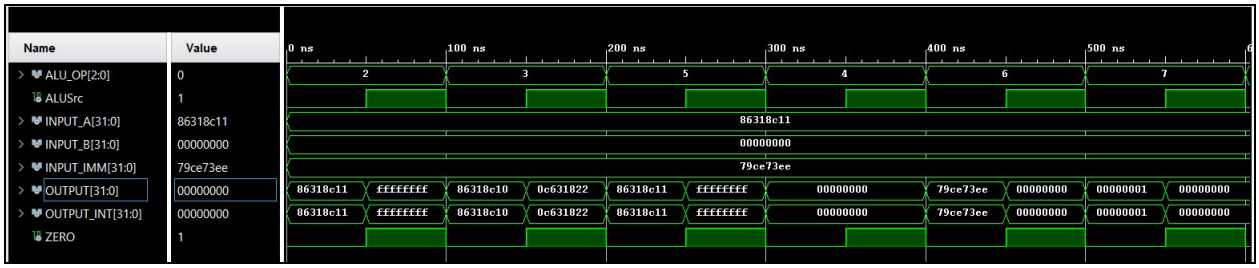


Figure 6: ALU Simulation Output

On observing the waveform above, the following results are obtained:

INPUT_A = 0x86318C11, INPUT_B = 0x00000000, INPUT_IMM = 0xFFFFFFFF

1. ALU_OP = 2, ALUSrc = 0, INPUT_A + INPUT_B => 0x86318C11, ZERO =>0

2. ALU_OP = 2, ALUSrc = 1, INPUT_A + INPUT_IMM => 0xFFFFFFFF, ZERO =>1
3. ALU_OP = 3, ALUSrc = 0, INPUT_A - INPUT_B => 0x86318C11, ZERO =>0
4. ALU_OP = 3, ALUSrc = 1, INPUT_A - INPUT_IMM => 0x0C318C10, ZERO =>1
5. ALU_OP = 5, ALUSrc = 0, INPUT_A | INPUT_B => 0x86318C11, ZERO =>0
6. ALU_OP = 5, ALUSrc = 1, INPUT_A | INPUT_IMM => 0xFFFFFFFF, ZERO =>1
7. ALU_OP = 4, ALUSrc = 0, INPUT_A & INPUT_B => 0x00000000, ZERO =>0
8. ALU_OP = 4, ALUSrc = 1, INPUT_A & INPUT_IMM => 0x00000000, ZERO =>1
9. ALU_OP = 6, ALUSrc = 0, INPUT_A nor INPUT_B => 0x79CE73ee, ZERO =>0
10. ALU_OP = 6, ALUSrc = 1, INPUT_A nor INPUT_IMM => 0x00000000, ZERO =>1
11. ALU_OP = 7, ALUSrc = 0, INPUT_A slt INPUT_B => 0x00000001, ZERO =>0
12. ALU_OP = 7, ALUSrc = 1, INPUT_A slt INPUT_IMM => 0x00000000, ZERO =>1

It is clear that all the above results are exactly as expected. Therefore, the ALU operates successfully!

Instruction Register

Description

The Instruction Register was designed to allow the CPU to parse an instruction and hold these values until the next instruction is fetched. This was achieved by using a single control signal — IRWrite.

IRWrite is used to simply inform the Instruction Register that there is a new instruction that needs to be parsed. This signal is also used so that the outputs — IMM, INSTRUCT_TYP, JMP_ADDRESS, REG1, REG2 and REG3 — are held until the next instruction is fetched. This function is achieved by the use of a latch (a register in VHDL). The schematic for the Instruction Register is given below:

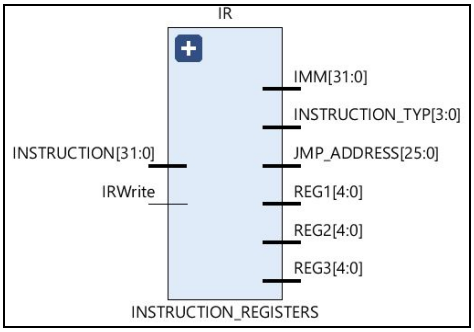


Figure 7: Instruction Register Block View

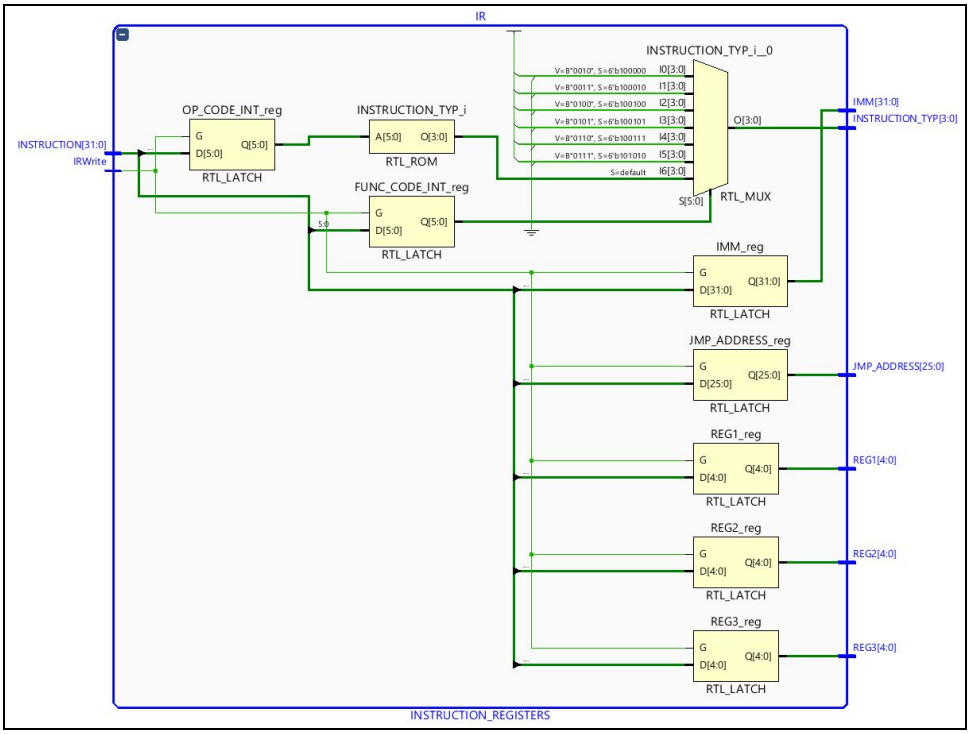


Figure 8: Instruction Register Internal View

Testing

To test the Instruction Register, a simple testbench program was created to verify the use of IRWrite as select for the latch. The testbench first inputs an instruction to the IR, and then sets the latch value to one.; this tests whether the values hold when the IRWrite is flipped to zero and instruction is changed, without updating the outputs. Then, when IRWrite is flipped to one, the outputs are updated. A screenshot of the testbench code is given below.

```
stimulus: process
begin
        IRWrite <= '1'; -- WRITE INSTURCTION TO BE PARSED
        INSTRUCTION <= "10000110001100011000110000100001"; -- INSTRUCTION TO BE PARSED
        wait for 50 ns;
        IRWrite <= '0'; -- DO NOT PARSE NEW INSTRUCTION
        INSTRUCTION <= "11111111111111111111111111111111"; -- CHANGE IN INSTRUCTION TO BE PARSED
        wait for 50 ns;
        IRWrite <= '1'; -- PARSE THE NEW INSTURCTION
        wait for 50 ns;

  wait;
end process;
```

Figure 9: Instruction Register Test Bench

Results & Analyses

On simulating the Instruction Register using the given testbench, the following output was obtained.

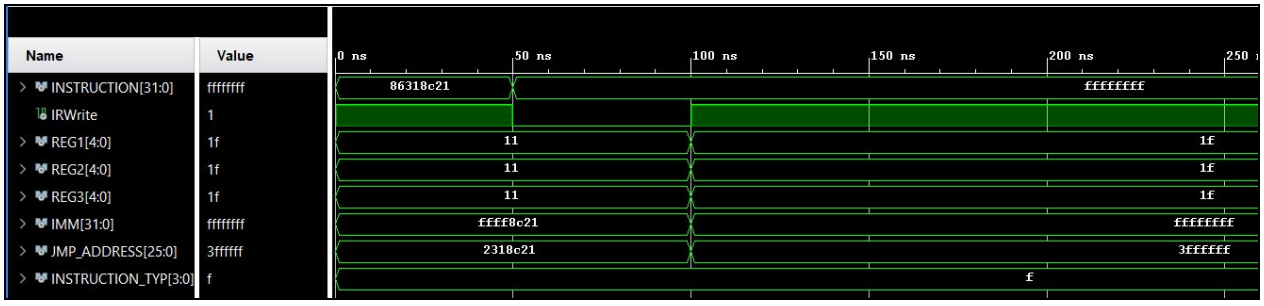


Figure 10: Instruction Register Simulation Output

As can be seen above, The first instruction 0x86318C21 is initially loaded and IRWrite is set to '1'. This means that the instruction will be parsed and it is. Then at 50ns IRWrite is set to '0' and instruction is changed to 0xFFFFFFFF. This is doing what is should, holding the parsed values from the previous instruction because IRWrite is '0' meaning do not update the parsed values. Then at 100ns only IRWrite is updated to '1', allowing the instruction 0xFFFFFFFF to be parsed and outputs to be updated. Clearly, this functionality is being done properly.

Registers Block

Description

The Registers Block was designed to allow the CPU to hold values on which operations are being performed, without having to read to and write from main memory — which typically takes more clock cycles, thus slowing down the CPU operation. The Registers Block designed here also performs the task of selecting the correct indices of registers to read from. This was achieved by using three control signals.

1. MemToReg → MemToReg is used to select between WRITEDATA_MEM and WRITEDATA_ALU values to write.
2. REGWRITE→ This control signal is used to select when to write back to a register.
3. RegDst → Finally, RegDst is used to select which register the value is to be written to.

The Registers Block is used for both selecting data to be passed forward to the ALU and Writing back to registers. The schematic for this component is given below.

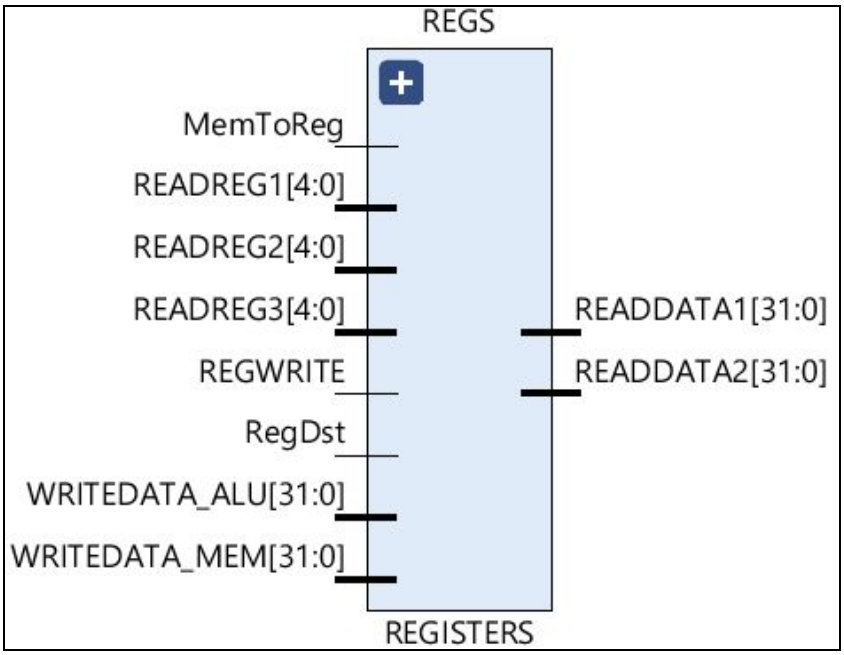


Figure 11: Registers Block Schematic

Testing

To test the Registers Block, a simple testbench program was created to verify the use of the MemToReg, REGWRITE and RegDst control signals. The first test was to hold REGWRITE to '0' and then update both MemToReg and RegDst, thus verifying that READDATA2 does not update. Then REGWRITE is held to '1' so that READDATA2 changed from WRITEDATA_ALU to WRITEDATA_MEM. A screenshot of the testbench code is given below.

```
stimulus: process
begin READREG1 <= "10001"; -- REGISTER 1 VALUE HELD FOR WHOLE TEST
      READREG2 <= "10000"; -- REGISTER 2 VALUE HELD FOR WHOLE TEST
      READREG3 <= "00000"; -- REGISTER 3 INITIAL VALUE
      WRITEDATA_ALU <= "01111001110011100111001111101110"; -- ALU VALUE HELD FOR WHOLE TEST
      WRITEDATA_MEM <= "11111111111111111111111111111111"; -- MEM VALUE HELD FOR WHOLE TEST
      RegDst <= '0'; -- WRITE DATA FROM MEMORY
      REGWRITE <= '0'; -- WRITE TO A REGISTER
      MemToReg <= '0'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      wait for 50ns; -- WAIT
      MemToReg <= '1'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      READREG3 <= "00001"; --UPDATE REGISTER 3 VALUE
      wait for 50ns; -- WAIT
      MemToReg <= '0'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      RegDst <= '1'; -- WRITE DATA FROM ALU
      READREG3 <= "00010"; --UPDATE REGISTER 3 VALUE
      wait for 50ns; -- WAIT
      MemToReg <= '1'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      READREG3 <= "00011"; --UPDATE REGISTER 3 VALUE
      wait for 50ns; -- WAIT
      RegDst <= '0'; -- WRITE DATA FROM MEMORY
      REGWRITE <= '1'; -- DO NOT WRITE TO A REGISTER
      MemToReg <= '0'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      wait for 50ns; -- WAIT
      MemToReg <= '1'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      READREG3 <= "00001"; --UPDATE REGISTER 3 VALUE
      wait for 50ns; -- WAIT
      RegDst <= '1'; -- WRITE DATA FROM ALU
      MemToReg <= '0'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      READREG3 <= "00010"; --UPDATE REGISTER 3 VALUE
      wait for 50ns; -- WAIT
      MemToReg <= '1'; -- WRITE DATA FROM MEMORY TO REGISTER 2
      READREG3 <= "00011"; --UPDATE REGISTER 3 VALUE
      wait for 50ns;
end process;
```

Figure 12: Registers Block Test Bench

Results & Analyses

On simulating the Register using the given testbench, the following output was obtained.

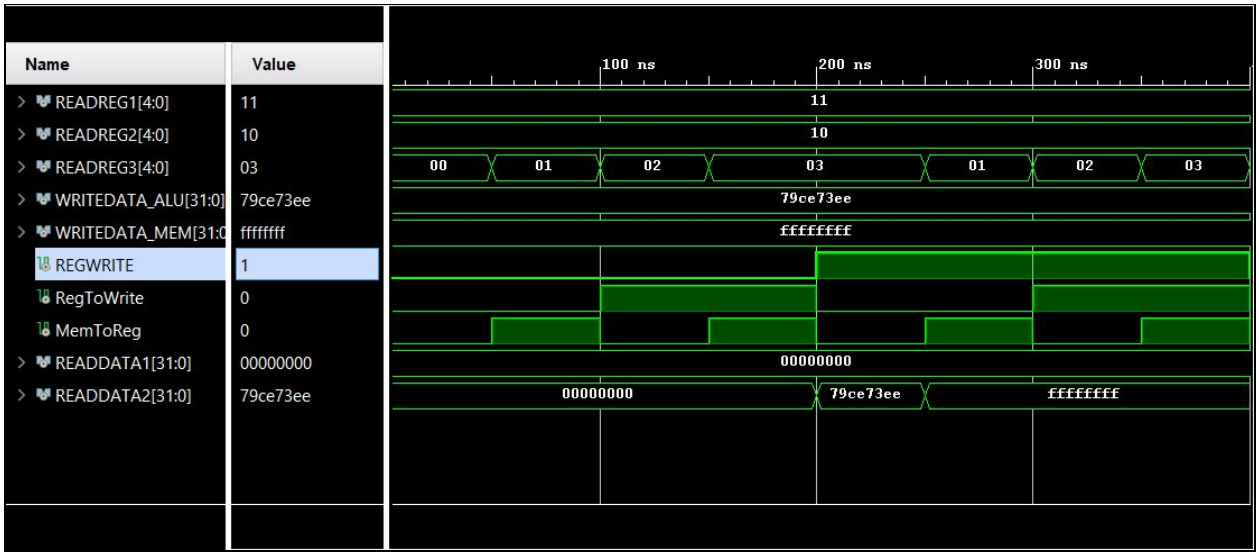


Figure 13: Registers Block Simulation Results

As can be seen above, READDATA2 does not get updated until after REGWRITE is set to '1'. READDATA2 is then updated to the WRITEDATA_ALU value, after which MemToReg is flipped to '1' and READDATA2 changes to the WRITEDATA_MEM value. Clearly, this functionality is being done properly.

Cache

Description

A simple block of memory was designed to test the functionality of the CPU. This block, hereafter called the "Cache", uses a simple read/write mechanism. To enable this, an "Address" input, as well as 2 control signals — MemRead & MemWrite — were added. Two buses, one for receiving input data (MemIn) and one for outputting data (MemOut) were also used. The schematic for the Cache is given below.

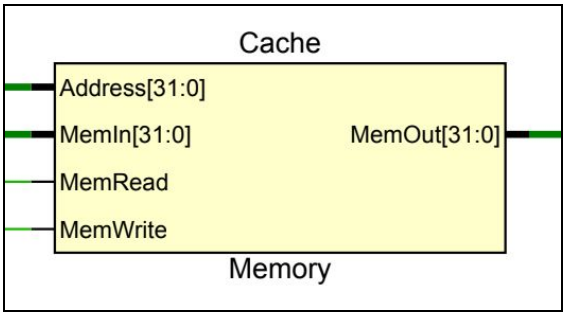


Figure 14: Cache Schematic

Internally, the Cache contains an array of 32-bit STD_LOGIC_VECTOR with integer index. Accordingly, each line of memory is 32-bit long and therefore stores a whole word. This was done because, in the MIPS architecture, everything is 32-bit long. Also, this project did not

require implementing instructions such as lb, lh, or other such instructions that access byte or halfword sized memory.

Based on the address inputted to the Cache, the data of the appropriate memory location is continuously outputted. On the other hand, the data on the MemIn bus is only written to memory if the MemWrite signal is enabled; the MemRead signal similarly controls the updating of the data on the MemOut bus. Since a single line in the memory is 32-bit long rather than the traditional 8-bit structure, the address inputted to the Cache is first divided by 4 and then converted into an integer, before using it as an index for reading or writing data. Also, since the size of the Cache is extremely limited as compared to the required 2^{32} addresses accessible by a 32-bit PC, the inputted address is modded by 1024 (which is the size of the implemented Cache), which effectively loops the memory locations around the Cache block.

Testing

To test the Cache, a simple testbench program that, first, writes data to memory, and then reads back the same memory location to test whether the write process was successful. A screenshot of the testbench code is given below.

```
stimulus: process
begin

  MemIn <= x"12345678"; -- Data to Write
  Address <= x"00000004"; -- Memory Address
  MemRead <= '0'; -- MemRead Disabled
  MemWrite <= '0'; -- MemWrite Disabled

  wait for 10 ns; -- Wait

  MemWrite <= '1'; -- Write to Memory

  wait for 10 ns; -- Wait

  MemWrite <= '0'; -- MemWrite Disabled
  MemRead <= '1'; -- Read from Memory

  wait for 10 ns; -- Wait

  MemRead <= '0'; -- MemRead Disabled

  wait;
end process;
```

Figure 15: Cache Test Bench

Results & Analyses

On simulating the Cache using the given testbench, the following output was obtained.

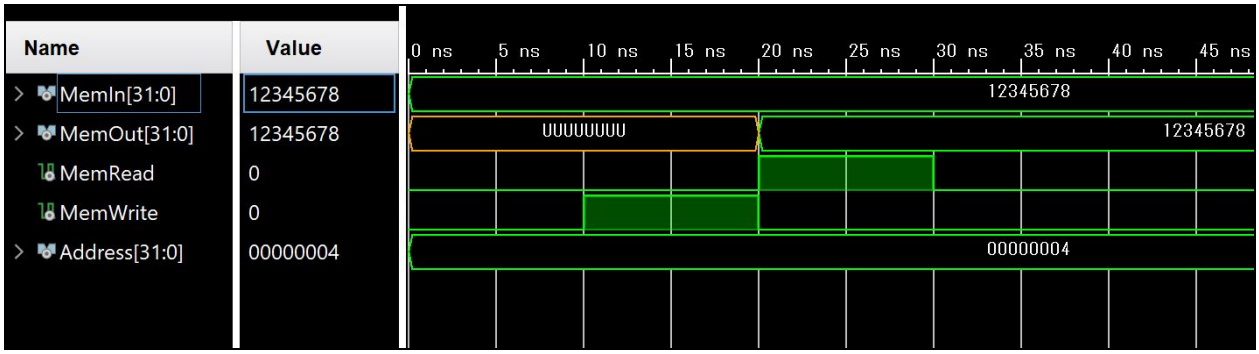


Figure 16: Cache Simulation Results A

As can be seen above, the data to be written to memory, i.e. 0x12345678, is inputted on the MemIn bus. At 10 ns, MemWrite is enabled, and data is written to memory location 0x4, which is inputted on the Address bus. Finally, at 20 ns, MemRead is enabled (with MemWrite disabled) and 0x12345678 is outputted on the MemOut bus. This clearly shows that data was successfully written to the appropriate location.

The same test was performed again, but this time, the MemWrite signal was never enabled. The output obtained is given below.

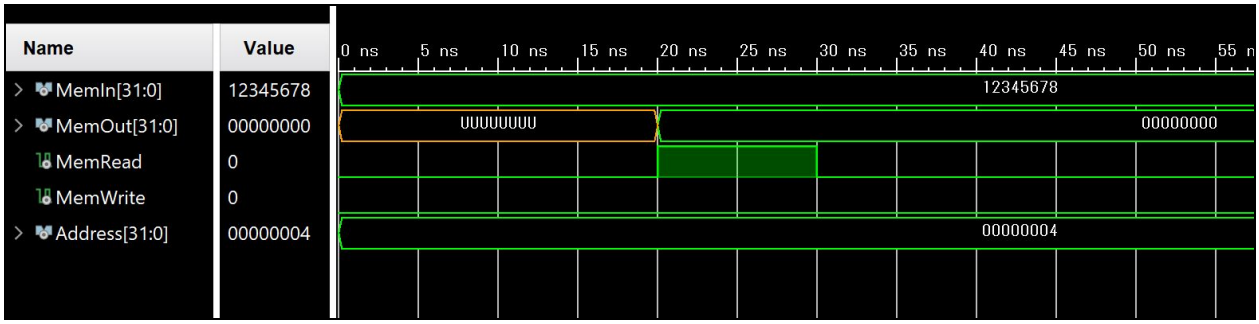


Figure 17: Cache Simulation Results B

Clearly, the inputted memory did not overwrite the memory located -- it was not intended to do so in this test -- and the default memory value, i.e. 0x0, is outputted on the MemOut bus. These two tests confirm that the Cache block works exactly as required by the project.

Memory Controller

Description

To connect the Cache with other components in the CPU, a Memory Controller unit was created. It's main purpose is to route appropriate data and memory addresses, as well as the control signals from the MCU, to the Cache such that it can function as required. The schematic for the Memory Controller is given below.

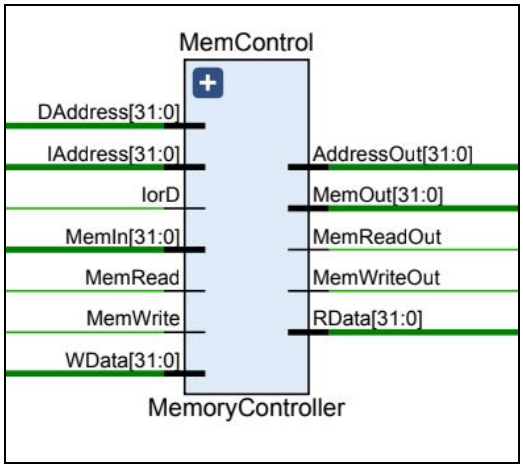


Figure 18: MemControl Block View

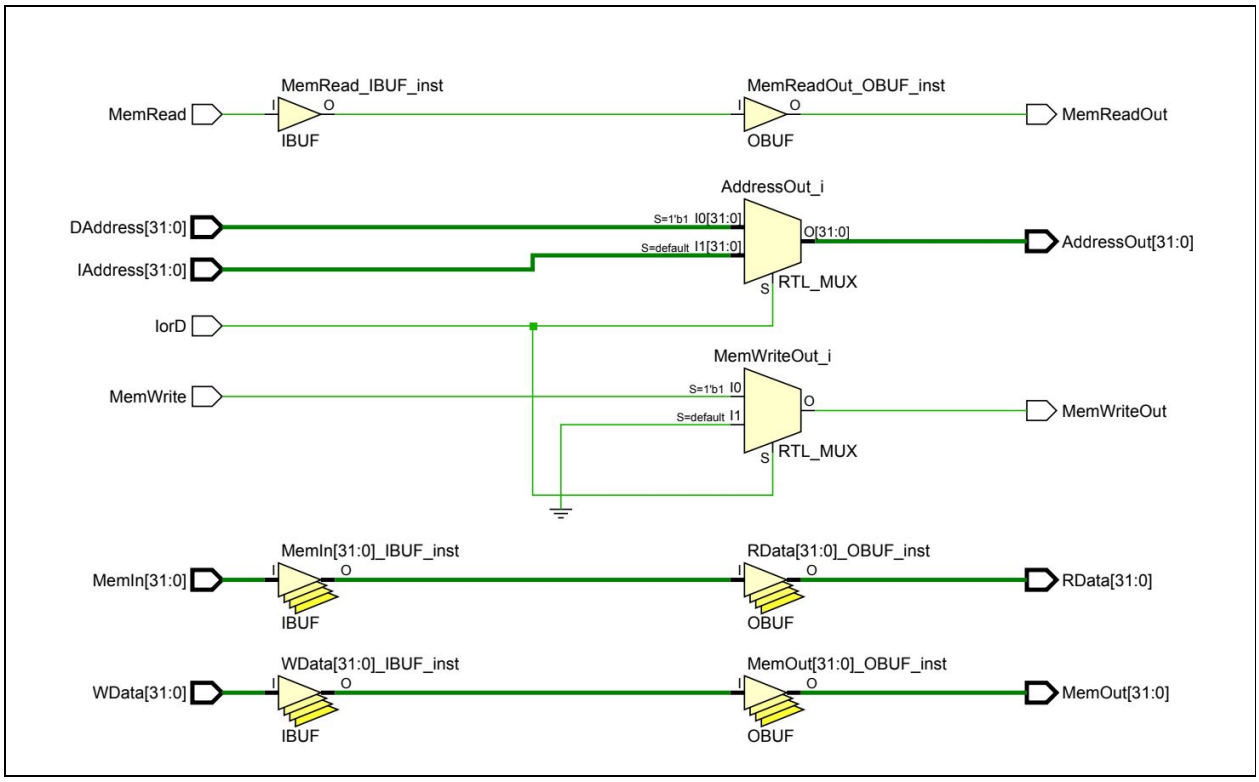


Figure 19: MemControl Internal View

There are two sources of Addresses available to the Memory Controller. Firstly, the Memory Controller receives the updated PC from the PCU though the IAddress bus. This address -- forwarded to the Cache on the AddressOut bus -- is used to fetch instructions

from memory, which is outputted on the RData bus. The Memory Controller also receives an address from the ALU, the DAddress, which is the result of the addition of the register content and 32-bit sign-extended immediate value in the lw & sw instructions. This address is, therefore, used to read data from memory, which is also outputted on the RData bus. During the execution of a sw instruction, the data to be written to memory is inputted on the WData bus. Finally, the 2 of the 3 control signals, i.e. MemRead & MemWrite, were forwarded to the Cache block on the MemReadOut & MemWriteOut outputs. The IorD control signal was used to prevent data to be written to memory when using the IAddress -- MemWriteOut was always disabled when using IAddress, regardless of the MemWrite control signal from the MCU.

Testing

To ensure that the Memory Controller routes appropriate data to the Cache, all possible scenarios -- 4 in this case -- were tested. These test cases, along with their testbench codes, are given below.

1. Reading an instruction from memory

```
stimulus: process
begin

  -- Reading an Instruction

  IAddress <= x"00000000"; -- Initial Address
  MemRead <= '0'; -- MemRead Disabled
  MemWrite <= '0'; -- MemWrite Disabled
  IorD <= '0'; -- Instruction

  wait for 10 ns; -- Wait

  MemIn <= x"98765432"; -- Data Being Read
  MemRead <= '1'; -- Read from Memory
  IAddress <= x"00000004"; -- Instruction Memory Location

  wait for 10 ns; -- Wait
  MemRead <= '0'; -- MemRead Disabled

  wait;
end process;
```

Figure 20: MemControl Test Bench A

2. Writing an instruction to memory

```
stimulus: process
begin

  -- Writing to Instruction Memory

  IAddress <= x"00000000"; -- Initial Address
  MemRead <= '0'; -- MemRead Disabled
  MemWrite <= '0'; -- MemWrite Disabled
  IorD <= '0'; -- Instruction

  wait for 10 ns; -- Wait

  WData <= x"12345678"; -- Data Being Written
  MemWrite <= '1'; -- Write to Memory
  IAddress <= x"00000004"; -- Memory Location

  wait for 10 ns; -- Wait
  MemWrite <= '0'; -- MemWrite Disabled

  wait;
end process;
```

Figure 21: MemControl Test Bench B

3. Reading data from memory

```
stimulus: process
begin

  -- Reading Data

  DAddress <= x"00000000"; -- Initial Address
  MemRead <= '0'; -- MemRead Disabled
  MemWrite <= '0'; -- MemWrite Disabled
  IorD <= '0'; -- Instruction

  wait for 10 ns; -- Wait

  IorD <= '1'; -- Data
  MemIn <= x"98765432"; -- Data Being Read
  MemRead <= '1'; -- Read from Memory
  DAddress <= x"00000004"; -- Data Memory Location

  wait for 10 ns; -- Wait
  MemRead <= '0'; -- MemRead Disabled

  wait;
end process;
```

Figure 22: MemControl Test Bench C

4. Writing Data to memory

```
stimulus: process
begin

  -- Writing Data

  DAddress <= x"00000000"; -- Initial Address
  MemRead <= '0'; -- MemRead Disabled
  MemWrite <= '0'; -- MemWrite Disabled
  IorD <= '0'; -- Instruction

  wait for 10 ns; -- Wait

  IorD <= '1'; -- Data
  WData <= x"12345678"; -- Data Being Written
  MemWrite <= '1'; -- Write to Memory
  DAddress <= x"00000004"; -- Data Memory Location

  wait for 10 ns; -- Wait
  MemWrite <= '0'; -- MemWrite Disabled

  wait;
end process;
```

Figure 23: MemControl Test Bench D

Results & Analyses

The outputs obtained on simulating the Memory Controller using each of the four testbenches are given below. Analyses of these results are also included.

1. Reading an instruction from memory

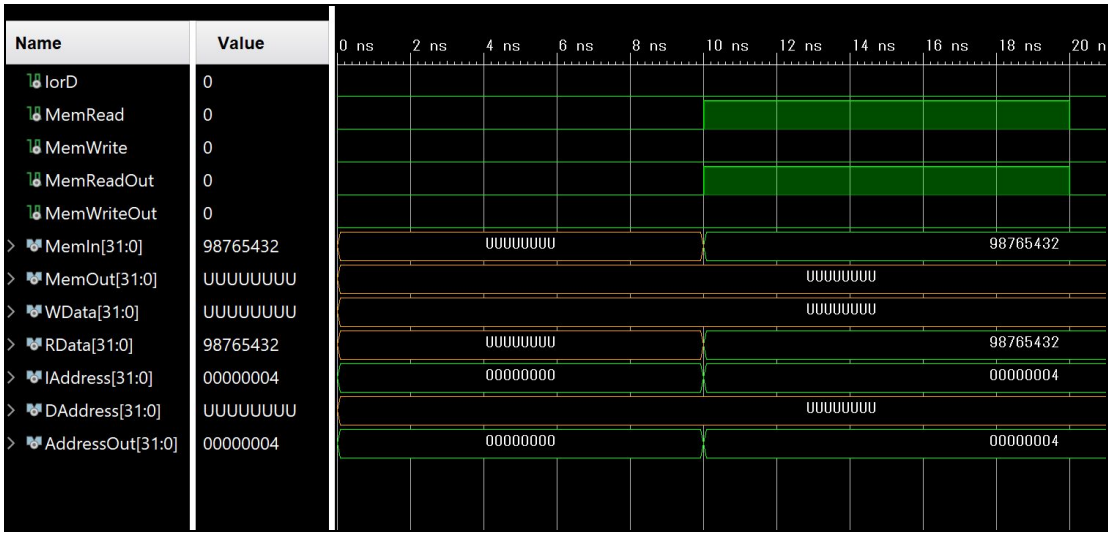


Figure 24: MemControl Simulation Results A

To mimic the behavior of receiving an input instruction from the Cache, the instruction 0x98765432 is loaded onto the MemIn bus. IAddress is given the value 0x4, which isn't really relevant here as there aren't actual connections to the Cache; This address is also outputted on the AddressOut bus, which would, in the actual CPU, go to the Cache. IorD is 0, which represents an instruction access. At 10 ns, MemRead, and consequently MemReadOut, is enabled, and clearly, the instruction 0x98765432 is outputted on the RData. Therefore, an instruction was successfully read from memory.

2. Writing an instruction to memory

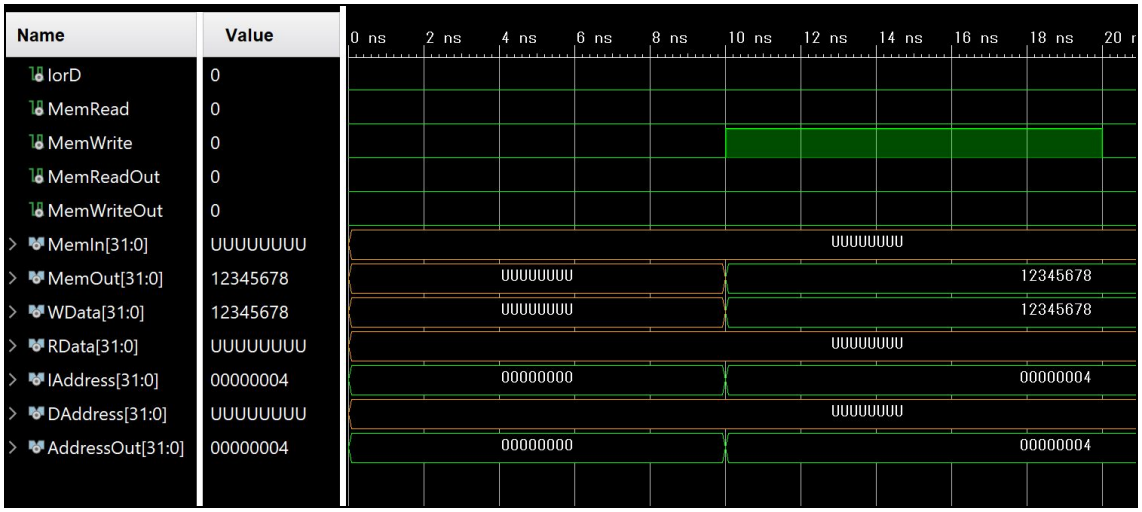


Figure 25: MemControl Simulation Results B

The instruction to be written to memory, i.e. 0x12345678, is loaded onto the WData bus. IAddress is given the value 0x4, which is also outputted on the AddressOut bus. IorD is 0, which represents an instruction access. At 10 ns, MemWrite is enabled. However, since no data must be written to instruction memory, MemWriteOut is still disabled, as can be seen in the output above. Note that the instruction to be written to memory from WData is still forwarded to the Cache block through the MemOut bus, but this instruction will not be written to memory since MemWriteOut is disabled. Therefore, the Memory Controller successfully prevents writing data to instruction memory.

3. Reading data from memory

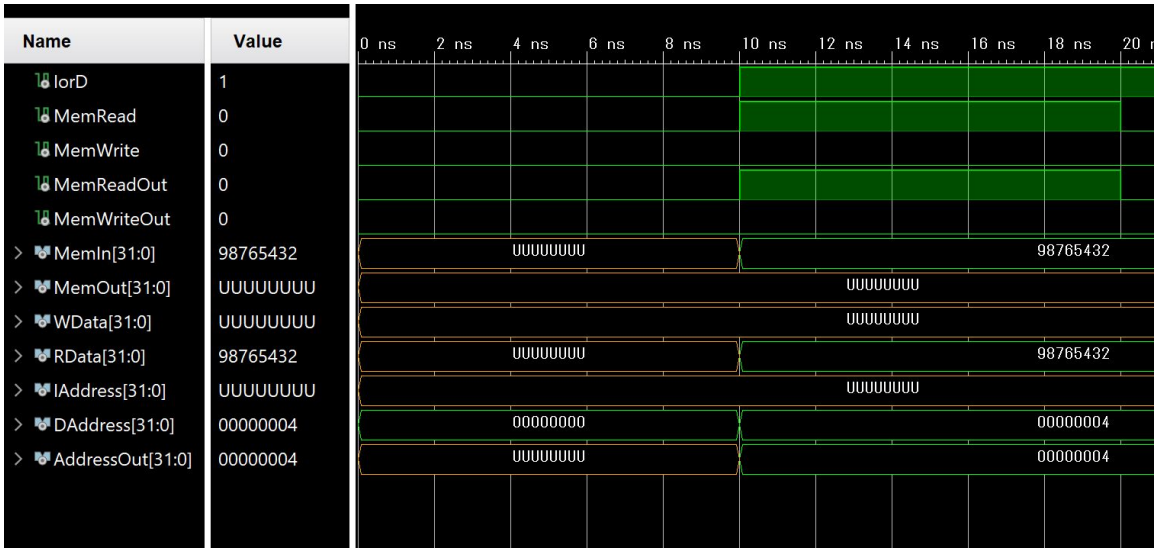


Figure 26: MemControl SImulation Results C

As in the first test case, an input data from the Cache, i.e. 0x98765432, is loaded onto the MemIn bus. DAddress is given the value 0x4, which is also outputted on the AddressOut bus. IorD is 1, which represents a data access. At 10 ns, MemRead, and consequently MemReadOut, is enabled, and clearly, the data 0x98765432 is outputted on the RData. Therefore, data was successfully read from memory.

4. Writing data to memory

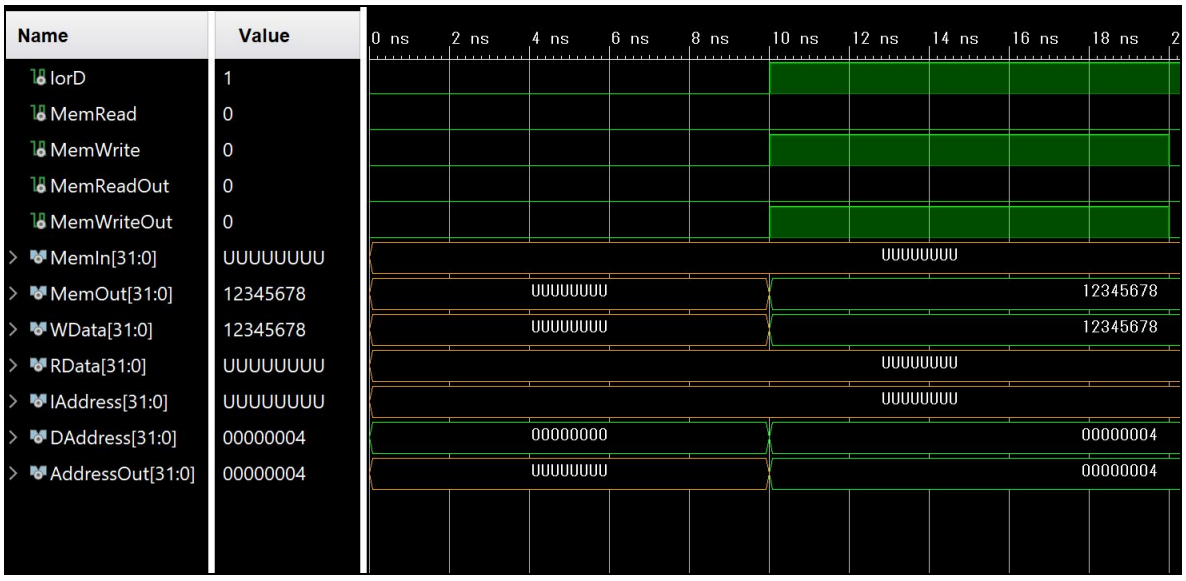


Figure 27: MemControl SImulation Results D

The data to be written to memory, i.e. 0x12345678, is loaded onto the WData bus. DAddress is given the value 0x4, which is also outputted on the AddressOut bus. IorD is 1, which represents a data access. At 10 ns, MemWrite, and consequently MemWriteOut, is enabled. The data to be written to memory from WData also forwarded to the Cache block through the MemOut bus. Therefore, the Memory Controller successfully writes data to memory.

As seen above, the Memory Controller has successfully passed all required tests, and is, therefore, in perfect working condition.

PCU

Description

The PCU was designed to perform the task of updating the current PC value to the new PC, based on the instruction being executed. The schematic for this component is given below.

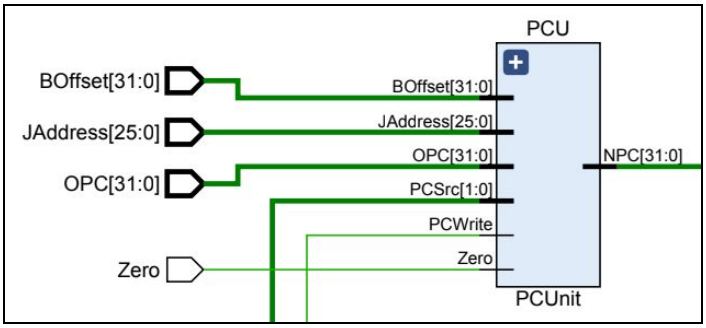


Figure 28: PCU Block View

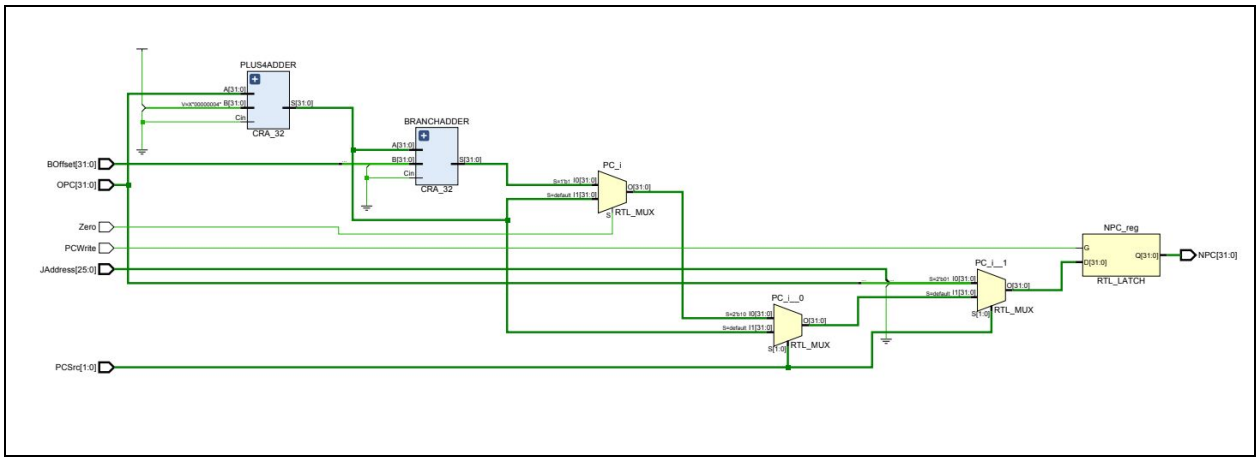


Figure 29: PCU Internal View

The PCU consists of two adders: one takes the current PC value (OPC) and adds 4 to it while the other uses the 32-bit sign-extended branch offset (BOffset) from the IR, which is first left-shifted by 2 bits and then added to the PC+4 result. The 26-bit jump address (JAddress) from the IR is left-shifted by 2 bits, after which the upper 4 bits (31:28) of the PC+4 result is appended to the left of this value. Depending on the PCSrc control signal from the MCU, as well as the Zero output of the ALU (used to decide whether or not to branch in a beq instruction), the appropriate value of the new PC is evaluated and stored in a variable (RTL latch), which is only outputted over the NPC bus when the PCWrite control signal from the MCU is enabled. The different values for the PCSrc control signal are given below. The default operation is PC + 4.

PCSrc	Zero	NPC Operation
00	X	PC + 4
01	X	Jump
10	0	PC + 4
10	1	Branch
Other	X	PC + 4

Testing

To test the PCU, a sequence of all kinds of operations was evaluated. This included a regular PC+4 update, a jump to a certain address, a beq instruction with branch not taken (Zero=1), and finally, a beq instruction with branch taken (Zero=1). The testbench is given below.

```
OPC <= x"00000000"; -- Initial PC
JAddress <= "10000000000000000000000001"; -- Jump to 0x08000004
BOffset <= x"0000003f"; -- Branch to 0x000000fc
PCSrc <= "00"; -- NPC = PC + 4
PCWrite <= '0'; -- PCWrite Disabled
Zero <= '0'; -- Branch Condition = False

wait for 10 ns; -- Delay

PCSrc <= "00"; -- Perform Regular PC + 4
wait for 10 ns; -- Delay
PCWrite <= '1'; -- Update NPC
wait for 10 ns; -- Delay

PCWrite <= '0'; -- PCWrite Disabled
PCSrc <= "01"; -- Perform Jump
wait for 10 ns; -- Delay
PCWrite <= '1'; -- Update NPC
wait for 10 ns; -- Delay

PCWrite <= '0'; -- PCWrite Disabled
PCSrc <= "10"; -- Perform Branch
Zero <= '0'; -- Branch Not Taken
wait for 10 ns; -- Delay
PCWrite <= '1'; -- Update NPC
wait for 10 ns; -- Delay

PCWrite <= '0'; -- PCWrite Disabled
PCSrc <= "10"; -- Perform Branch
Zero <= '1'; -- Branch Taken
wait for 10 ns; -- Delay
PCWrite <= '1'; -- Update NPC

wait;
```

Figure 30: PCU Test Bench

Results & Analyses

The following output was obtained on simulating the PCU using the given testbench.

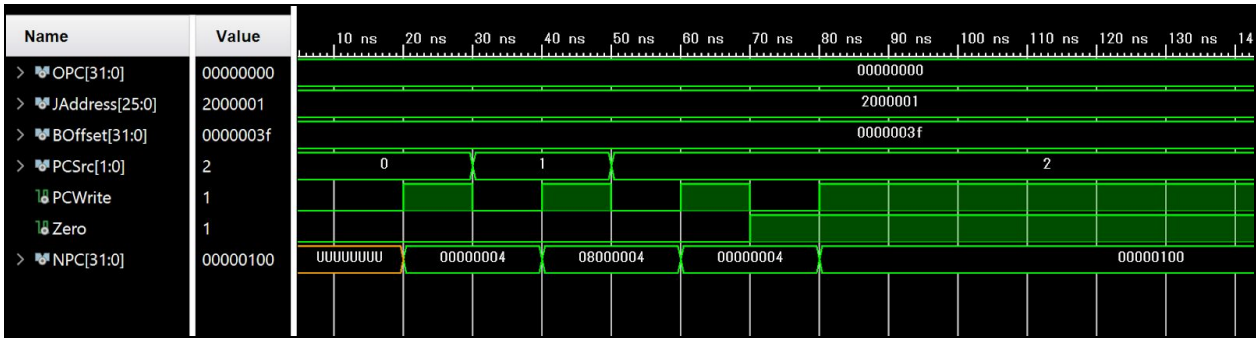


Figure 31: PCU Simulations Results

The initial PC ix 0x0 (which is the default value). After the first PCWrite at 20 ns, the updated PC+4 value, i.e. 0x4, is clearly seen on the NPC.

Next, at 40 ns, the PC is updated to the jump address. The 26 bit jump address from the IR is 0x02000001, which is left-shifted by 2 bits to give 0x08000004, and then the upper 4 bits of PC+4, which are all 0's, is appended to the left of this value. Accordingly, the jump address, i.e. 0x08000004, is seen on the NPC.

At 60 ns, the beq instruction with branch not taken (Zero = 0) is executed. Accordingly, the regular PC+4 operation is performed, and NPC has the value 0x4.

Finally, at 80 ns, the beq instruction with branch taken (Zero = 1) is executed. This time, the branch address is calculated by adding the 32-bit sign-extended BOffset from the IR, i.e. 0x3f — by first left-shifting it 2 bits to get 0xfc — to the PC+4 result, i.e. 0x4. The NPC value is accordingly 0x100.

Therefore, the PCU has been able to successfully update the PC in all different cases. Accordingly, this component seems to work perfectly.

MCU

Description

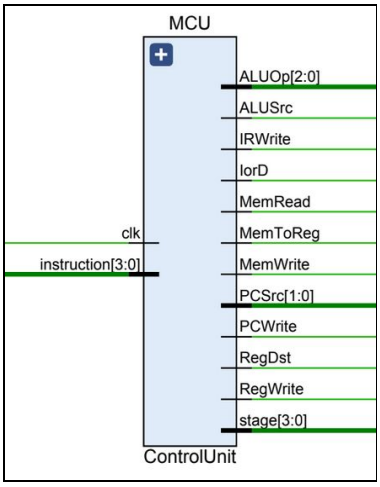


Figure 32: MCU Schematic

The MCU controls the entire functioning of the CPU. Using the clock input, it synchronizes all components. The execution of any given instruction is broken down into a fixed number of stages, with each stage running in a single clock cycle. An output displaying the number of the stage being executed in a particular clock cycle is added to the MCU in order to enable efficient testing. The different possible stages are as follows.

1. IF: MemControl receives next instruction address from PCU and fetches a word from memory at that address.
2. ID: The instruction register decodes the different parts of the instruction.
3. REG: The Registers block outputs read register values.
4. ALU: The ALU performs the required computation.
5. MEM: MemControl receives data address from the ALU and fetches/writes a word from/to that memory location.
6. WB: The Registers block receives data from the ALU or MemControl and writes it to the appropriate register.
7. PC: The NPC is computed depending on the instruction.
8. PCW: PC is updated.

The first two stages, i.e. IF and ID, are common to every instruction. At the end of the ID stage, the MCU receives the 4-bit decoded instruction from the IR, and is now able to send out appropriate control signals to all of the components, depending on the instruction being executed. Brief descriptions of these different control signals are given below.

1. IorD: Represents whether the memory operation pertains data or an instruction. IorD = 0 implies an instruction, while IorD = 1, data.
2. MemRead: Enables data to be read from memory if asserted.
3. MemWrite: Enables data to be written to memory if asserted.
4. IRWrite: IR register receives a new instruction when this control signal is asserted.
5. RegDst: Represents whether the write register index is taken from bits 20:16 (RegDst = 0) or bits 15:11 (RegDst = 1) of the instruction.
6. MemToReg: Represents whether data to be written to a register comes from the ALU output (MemToReg = 0) or from memory (MemToReg = 1).
7. RegWrite: Enables data to be written to appropriate register if asserted.
8. ALUSrc: Represents whether ALU input #2 is the content of read register 2 (ALUSrc = 0) or the 32-bit sign-extended immediate value (ALUSrc = 1).
9. ALUOp: Represents the operation to be performed by ALU (refer to ALU section).
10. PCSrc: Represents the operation to be performed by the PCU (refer to PCU section).
11. PCWrite: Enables PC to be updated when asserted.

There are 6 different types of stage cycles based on the Instruction Set supported by the CPU. A 7th stage cycle, for an unidentified instruction, is also present. Each of these are extensively described below.

1. *Jump Instruction:*

Stage 1 — IF

The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) & MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the memory address represented by the MCU and outputs it to the IR.

Stage 2 — ID
The IR decodes the instruction. The lower 26 bits of the instruction are forwarded to the PCU (JAddress). The instruction type (= 0000) is also forwarded to the MCU.

Stage 3 — PC
The PCU computes the destination address by appending the four most significant bits of the current PC to the left of JAddress and two 0's to the right. The PCU receives the PCSrc (= 01) control signal from the MCU and accordingly computes the new PC value

Stage 4 — PCW
The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

2. *Branch Instruction:*

Stage 1 — IF
The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) & MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the memory address represented by the MCU and outputs it to the IR.

Stage 2 — ID
The IR decodes the instruction. The two register indices are forwarded to the Registers block, while the sign-extended 32-bit branch offset is sent to the PCU. The instruction type (= 0001) is also forwarded to the MCU.

Stage 3 — REG
The Registers block outputs the contents of the two read registers, given by the register indices from IR, to the ALU.

Stage 4 — ALU
The ALU receives inputs from the Registers block, as well as ALUSrc (=0) & ALUOp (= 011) control signals from the MCU. It's "Zero" output is forwarded to the PCU.

Stage 5 — PC
The PCU computes the destination address by adding the 32-bit sign-extended BOffset to the current PC. It receives the PCSrc (= 10) control signal from the MCU and accordingly computes the new PC value.

Stage 6 — PCW
The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

3. *R-Type Instruction (add, sub, and, or, nor, slt):*

Stage 1 — IF
The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) & MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the memory address represented by the MCU and outputs it to the IR.

Stage 2 — ID
The IR decodes the instruction. The three register indices are forwarded to the Registers block. The instruction type (= 0010) is also forwarded to the MCU.

Stage 3 — REG
The Registers block receives RegDst (=1) control signal from the MCU and outputs the contents of the two read registers, given by the register indices from IR, to the ALU. The RegWrite signal is currently disabled (= 0).

Stage 4 — ALU
The ALU receives inputs from the Registers block, as well as ALUSrc (=0) & ALUOp (depends on instruction) control signals from the MCU. The result of the ADD operation is forwarded to the Registers block.

Stage 5 — WB
The Registers block receives the MemToReg (=0) and MemWrite (=1) control signals from the MCU and accordingly writes the addition result to the appropriate register.

Stage 6 — PC
The PCU receives the PCSrc (= 00) control signal from the MCU and accordingly computes the new PC value of PC + 4.

Stage 7 — PCW
The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

4. *I-Type Instruction (addi, andi, ori):*

Stage 1 — IF
The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) &
MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the
memory address represented by the MCU and outputs it to the IR.

Stage 2 — ID
The IR decodes the instruction. The two register indices are forwarded to the Registers block, and the 16 least significant bits (15-0) of the instruction are sign-extended and forwarded to the ALU. The instruction type (= 1010) is also forwarded to the MCU.

Stage 3 — REG
The Registers block receives RegDst (=0) control signal from the MCU and outputs the contents of the read register, given by the register index from IR, to the ALU. The RegWrite signal is currently disabled (= 0).

Stage 4 — ALU
The ALU receives inputs from the Registers block and IR, as well as ALUSrc (=1) & ALUOp (depends on instruction) control signals from the MCU. The result of the ADD Imm operation is forwarded to the Registers block.

Stage 5 — WB
The Registers block receives the MemOrReg (=0) and MemWrite (=1) control signals from the MCU and accordingly writes the immediate addition result to the appropriate register.

Stage 6 — PC
The PCU receives the PCSrc (= 00) control signal from the MCU and accordingly computes the new PC value of PC + 4.

Stage 7 -- PCW
The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

5. *Store Word Instruction:*

Stage 1 — IF
The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) & MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the memory address represented by the MCU and outputs it to the IR.

Stage 2 — ID
The IR decodes the instruction. The two register indices are forwarded to the Registers block, and the 16 least significant bits (15-0) of the instruction are sign-extended and forwarded to the ALU. The instruction type (= 1000) is also forwarded to the MCU.

Stage 3 — REG
The Registers block outputs the contents of the two read registers given by the register indices from IR -- the destination register contents to the ALU and the source register contents to the MemControl (WData).

Stage 4 — ALU

The ALU receives inputs from the Registers block and IR, as well as ALUSrc (=1) & ALUOp (=010) control signals from the MCU. The result of the ADD Imm operation is forwarded to the MemControl (DAddress).

Stage 5 — MEM

The MemControl receives DAddress from the ALU, DataIn from the Registers block, and IorD (=1), MemRead (=0) & MemWrite (=1) control signals from the MCU. The data (from the source register) is accordingly written to the appropriate memory address.

Stage 6 — PC

The PCU receives the PCSrc (= 00) control signal from the MCU and accordingly computes the new PC value of PC + 4.

Stage 7 — PCW

The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

6. *Load Word Instruction:*

Stage 1 — IF

The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) & MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the memory address represented by the MCU and outputs it to the IR.

Stage 2 — ID

The IR decodes the instruction. The two register indices are forwarded to the Registers block, and the 16 least significant bits (15-0) of the instruction are sign-extended and forwarded to the ALU. The instruction type (= 1001) is also forwarded to the MCU.

Stage 3 — REG

The Registers block outputs the contents of the two read registers given by the register indices from IR -- the destination register contents to the ALU and the source register contents to the MemControl (DataIn).

Stage 4 — ALU

The ALU receives inputs from the Registers block and IR, as well as ALUSrc (=1) & ALUOp (=010) control signals from the MCU. The result of the ADD Imm operation is forwarded to the MemControl (DAddress).

Stage 5 — MEM
The MemControl receives DAddress from the ALU, DataIn from the Registers block, and IorD (=1), MemRead (=1) & MemWrite (=0) control signals from the MCU. The data is read from the appropriate memory address and is sent to the Registers block.

Stage 6 — WB
The Registers block receives the MemOrReg (=1) and MemWrite (=1) control signals from the MCU and accordingly writes the data retrieved from memory to the appropriate register.

Stage 7 — PC
The PCU receives the PCSrc (= 00) control signal from the MCU and accordingly computes the new PC value of PC + 4.

Stage 8 — PCW
The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

7. *Unidentified Instruction:*

Stage 1 — IF
The NPC is inputted to the MemControl from the PCU along with IorD (= 0), MemRead (= 1) & MemWrite (= 0) control signals from the MCU. The MemController fetches the contents of the memory address represented by the MCU and outputs it to the IR.

Stage 2 -- ID
The IR decodes the instruction. The lower 26 bits of the instruction are forwarded to the PCU (JAddress). The instruction type (= 0000) is also forwarded to the MCU.

Stage 3 — PC
The PCU receives the PCSrc (= 00) control signal from the MCU and accordingly computes the new PC value of PC + 4.

Stage 4 — PCW
The PCU receives PCWrite (=1) control signal from the MCU and accordingly updates the PC.

Testing

Each of the 7 mentioned stage cycles were tested using a simple testbench with two inputs: (1) a CLK, with a period of 10 ns; and (2) a 4-bit instruction value. The only change made to the testbench during successive tests was the instruction number. The testbench code is given below.

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  instruction <= "0000"; -- instruction #
  wait for 100 ns; -- wait

  stop_the_clock <= true; -- end clock
  wait;
end process;
```

Figure 33: MCU Test Bench

Results & Analyses

The outputs for each of the tests conducted for the various instructions and corresponding stage cycles are given below. Detailed analyses of the results are also included.

1. *Jump Instruction:*

Figure 34: MCU Jump Instruction Simulation Results

- In stage 1 (IF), MemRead = 1. This enables MemControl to read an instruction from memory.
- In stage 2 (ID), IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (PC), the MCU sends the appropriate PCSrc = 01 signal to the PCU since this is a jump instruction. Accordingly, the PCU uses the 26 jump address bits it received from the IR and the current PC to compute the NPC.
- In stage 4 (PCW), PCWrite = 1, which updates current PC to NPC.

2. *Branch Instruction:*

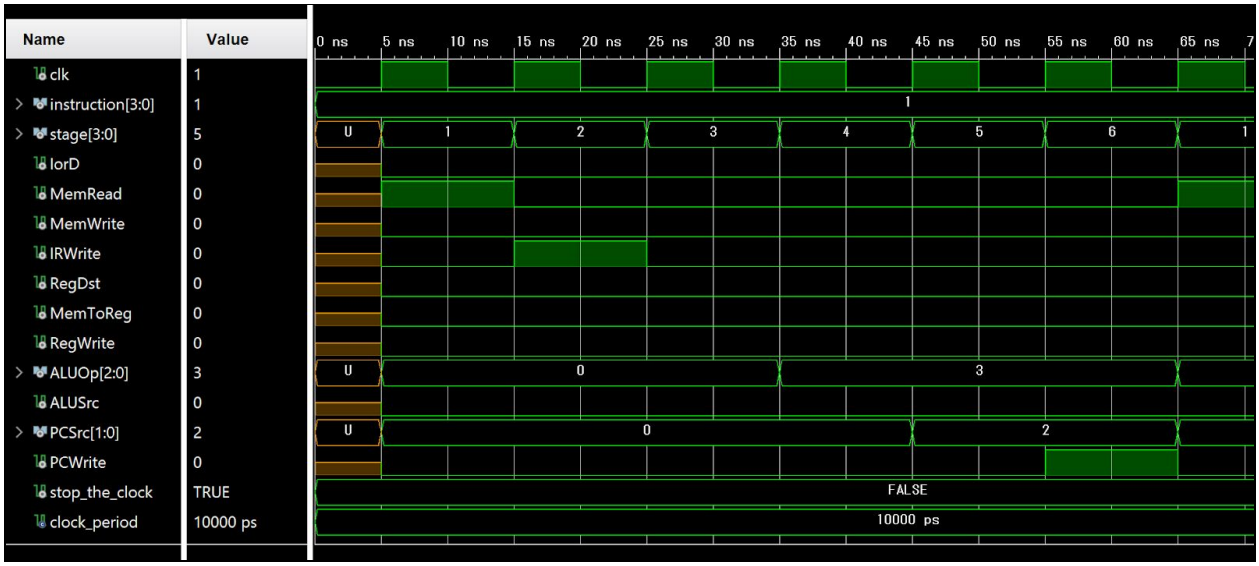


Figure 35: MCU Branch Instruction Simulation Results

- In stage 1 (IF), MemRead = 1. This enables MemControl to read an instruction from memory.
- In stage 2 (ID), IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (REG), the MCU waits for the Registers block to output appropriate register contents.
- In stage 4 (ALU), the MCU sends ALUOp = 011 (subtraction) to the ALU, thus allowing it to output the Zero result to the PCU. ALUSrc = 0 since the register content is used.
- In stage 5 (PC), the MCU sends the appropriate PCSrc = 10 signal to the PCU since this is a branch instruction. Accordingly, the PCU uses the 32-bit sign-extended immediate value from the IR, the Zero result from the ALU, and the current PC, to compute the NPC.
- In stage 6 (PCW), PCWrite = 1, which updates current PC to NPC.

3. R-Type Instruction (add, sub, and, or, nor, slt):

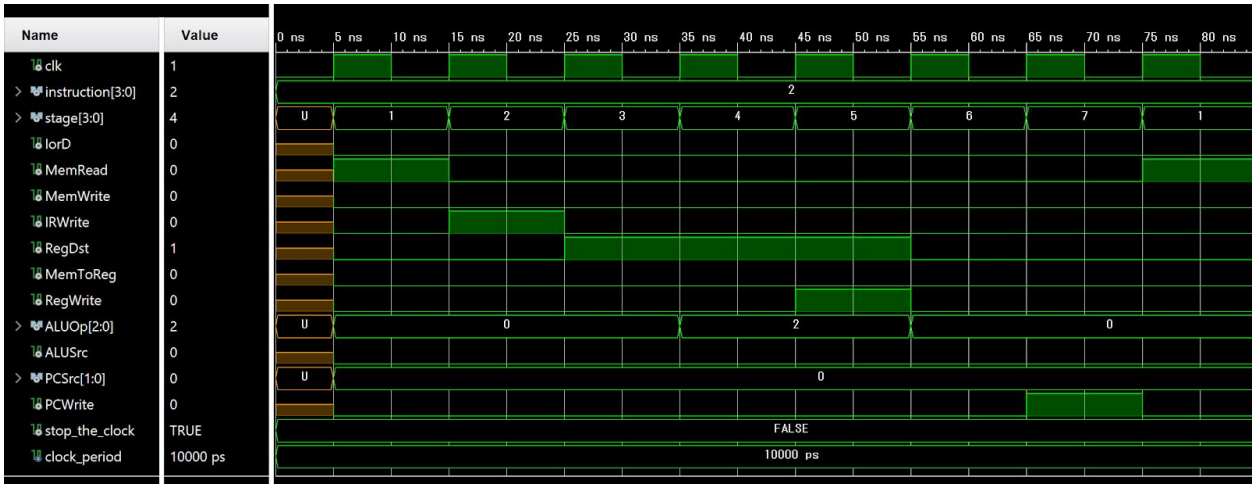


Figure 36: MCU Add Instruction Simulation Results

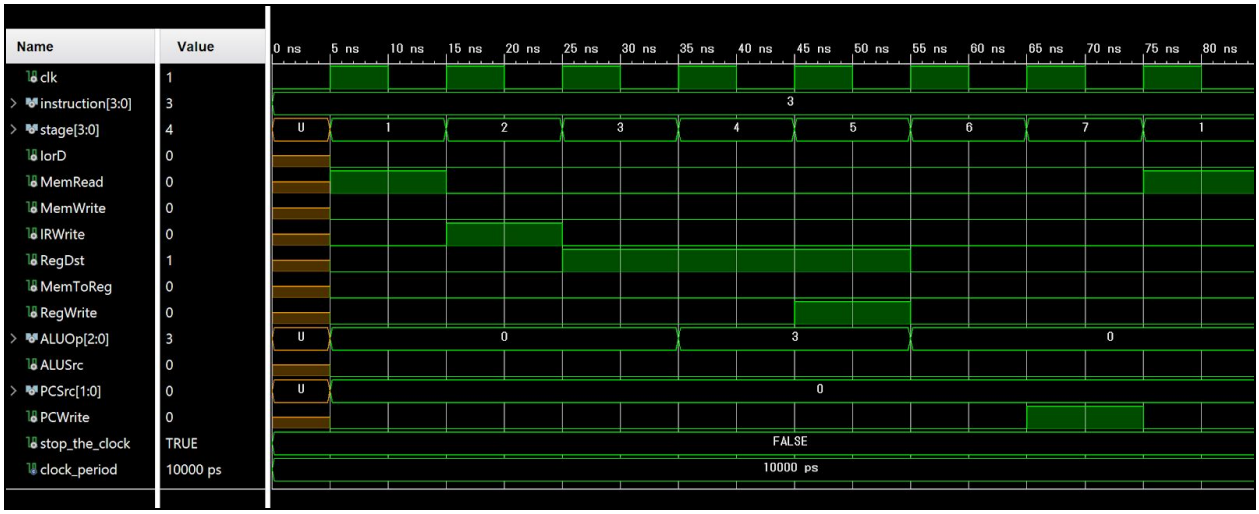


Figure 37: MCU Subtract Instruction Simulation Results

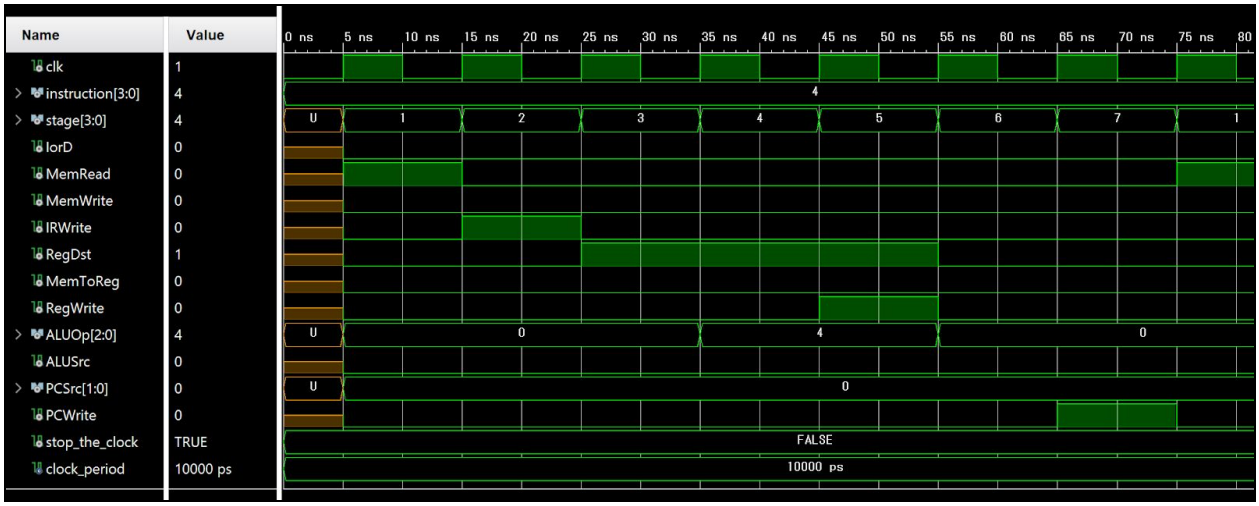


Figure 38: MCU AND Instruction Simulation Results

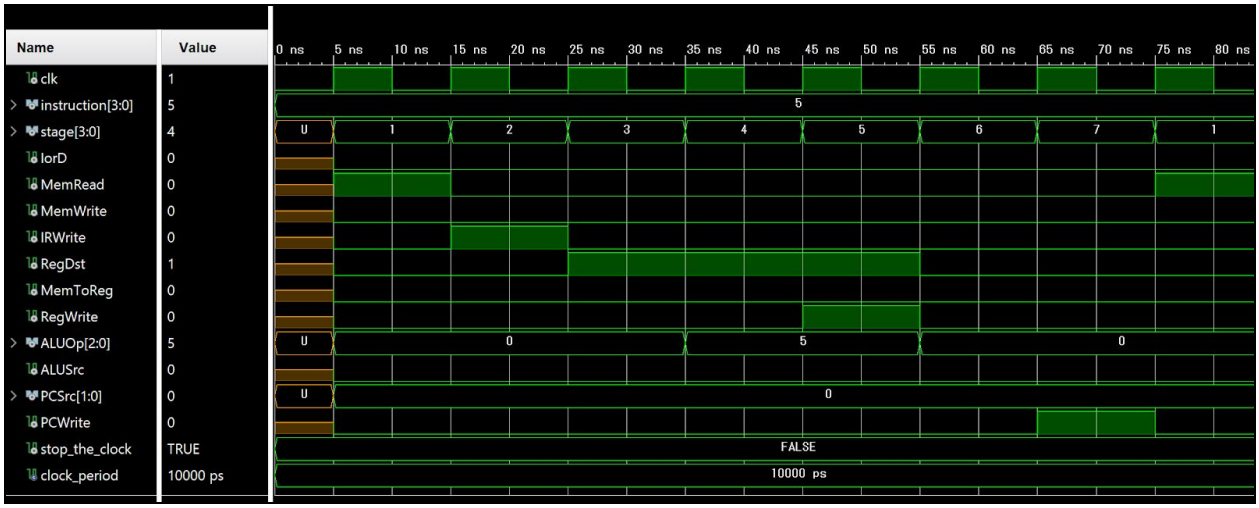


Figure 39: MCU OR Instruction Simulation Results

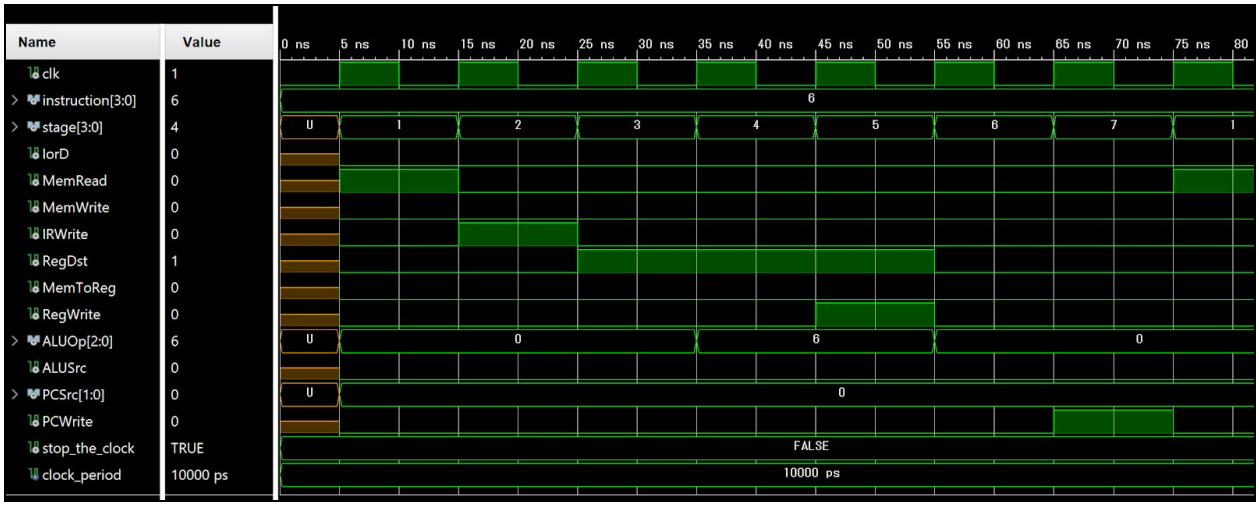


Figure 40: MCU NOR Instruction Simulation Results

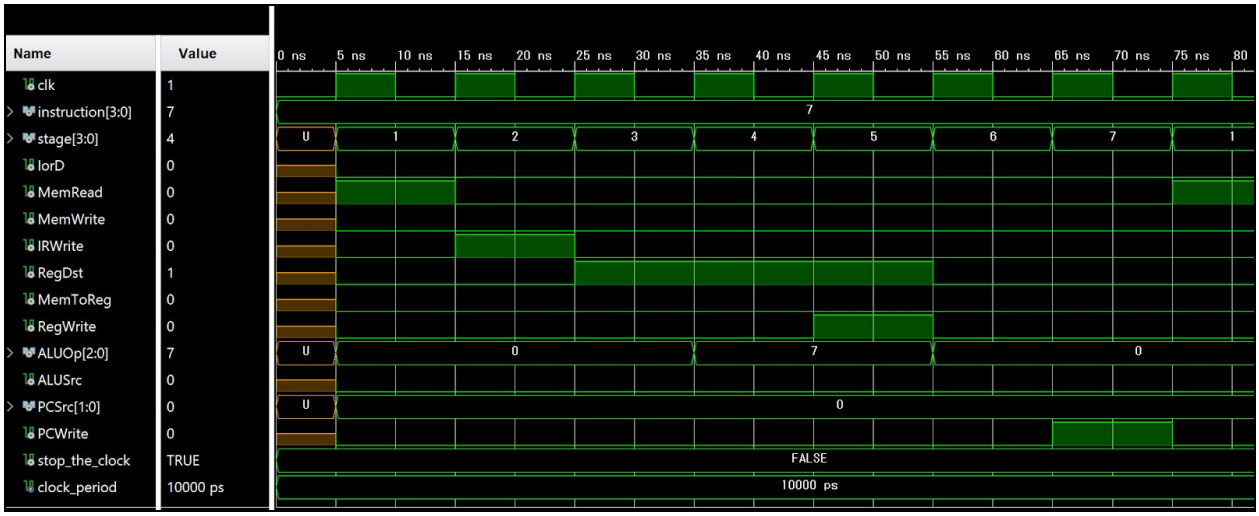


Figure 41: MCU SLT Instruction Simulation Results

- In stage 1 (IF) for all instructions, MemRead = 1. This enables MemControl to read an instruction from memory.
- In stage 2 (ID) for all instructions, IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (REG) for all instructions, the MCU waits for the Registers block to output appropriate register contents. MCU also sets RegDst = 1 and MemToReg = 0 since this is an R-type instruction.
- In stage 4 (ALU), the MCU sends ALUOp =
    - 010 (addition) for add
    - 011 (subtraction) for sub
    - 100 (AND operation) for and
    - 101 (OR operation) for or
    - 110 (NOR operation) for nor
    - 111 (Set On Less Than) for slt

  … to the ALU, thus allowing it to perform the appropriate task. ALUSrc = 0 since the register content is used.
- In stage 5 (WB) for all instructions, RegWrite = 1, which effectively writes the ALU output to the appropriate register.
- In stage 6 (PC) for all instructions, the MCU sends the appropriate PCSrc = 00 signal to the PCU since this is an R-Type instruction. Accordingly, the PCU increments current PC by 4.
- In stage 7 (PCW) for all instructions, PCWrite = 1, which updates current PC to NPC.

4. I-Type Instruction (addi, andi, ori):

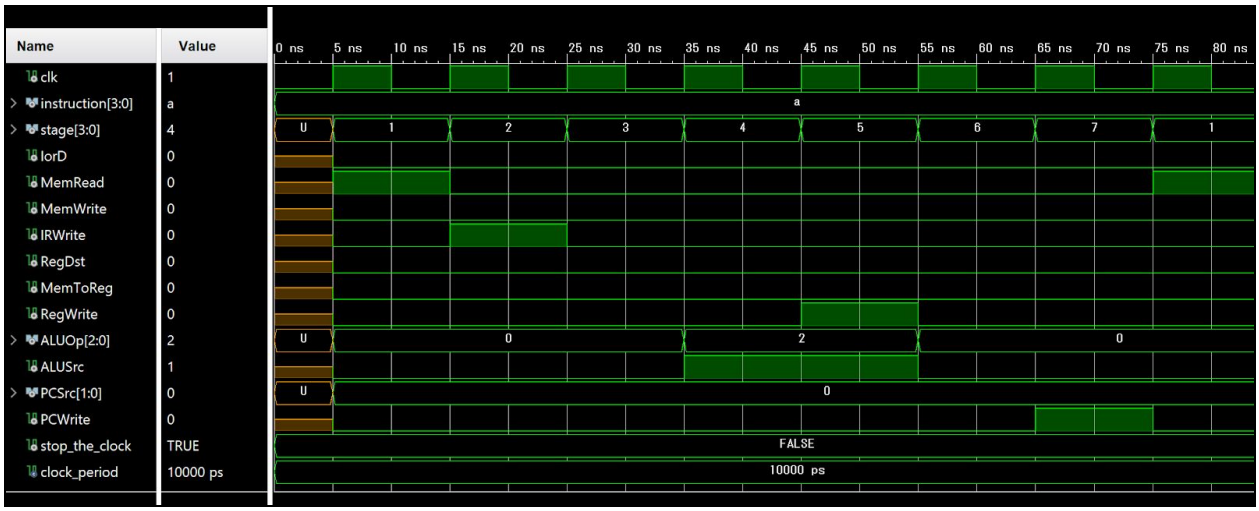


Figure 42: MCU ADD Immediate Instruction Simulation Results

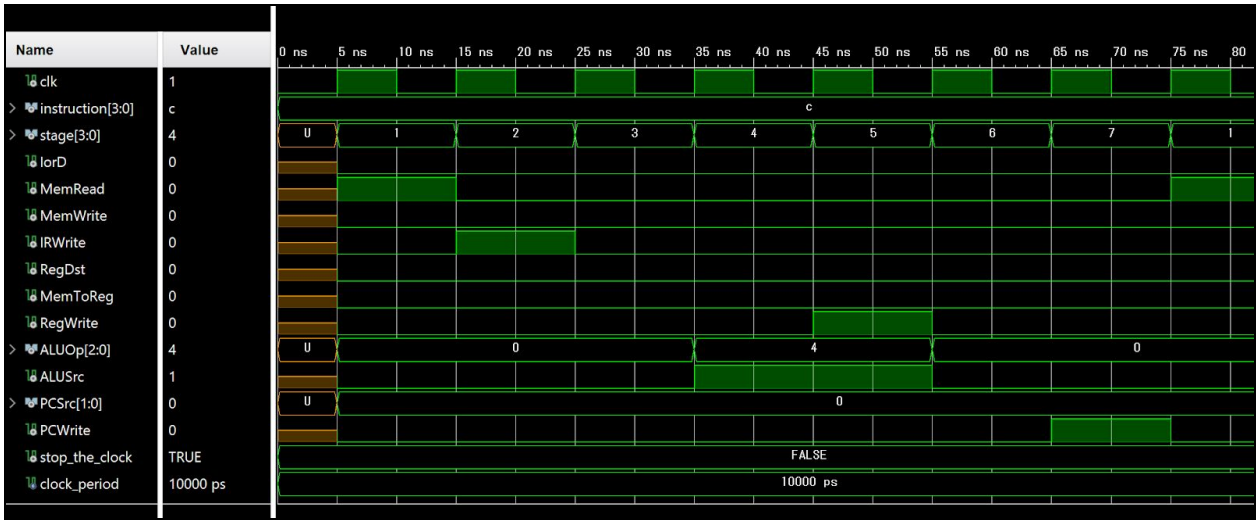


Figure 43: MCU AND Immediate Instruction Simulation Results

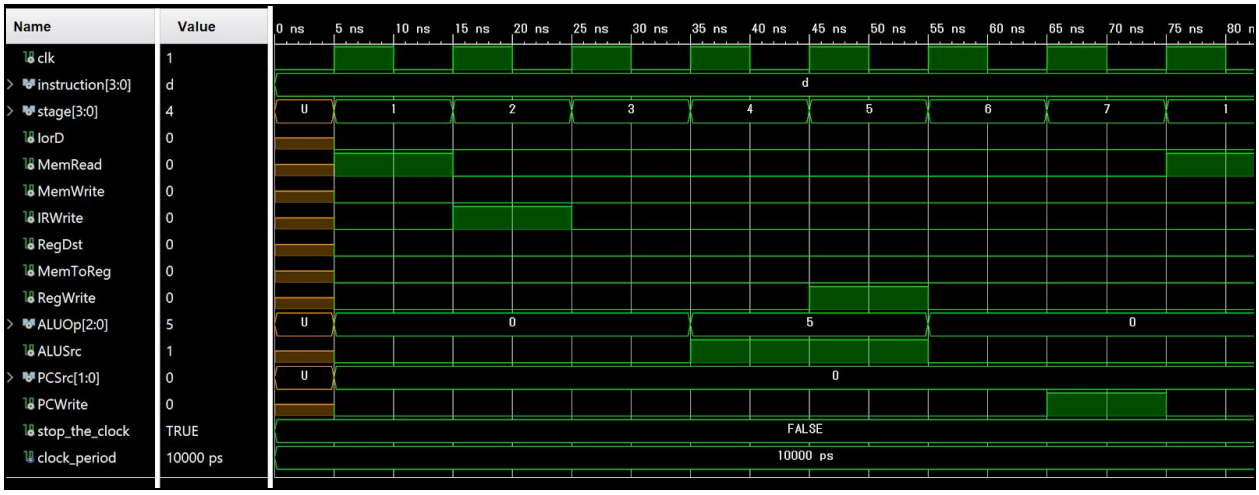


Figure 44: MCU OR Immediate Instruction Simulation Results

- In stage 1 (IF) for all instructions, MemRead = 1. This enables MemControl to read an instruction from memory.
- In stage 2 (ID) for all instructions, IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (REG) for all instructions, the MCU waits for the Registers block to output appropriate register contents. MCU also sets RegDst = 0 and MemToReg = 0 since this is an I-type instruction that writes to a register.
- In stage 4 (ALU), the MCU sends ALUOp =
    - 010 (addition) for addi
    - 100 (AND operation) for andi
    - 101 (OR operation) for ori

  … to the ALU, thus allowing it to perform the appropriate task. ALUSrc = 1 since the 32-bit sign-extended immediate value is used.
- In stage 5 (WB) for all instructions, RegWrite = 1, which effectively writes the ALU output to the appropriate register.
- In stage 6 (PC) for all instructions, the MCU sends the appropriate PCSrc = 00 signal to the PCU since this is an R-Type instruction. Accordingly, the PCU increments current PC by 4.
- In stage 7 (PCW) for all instructions, PCWrite = 1, which updates current PC to NPC.

5. *Store Word Instruction:*

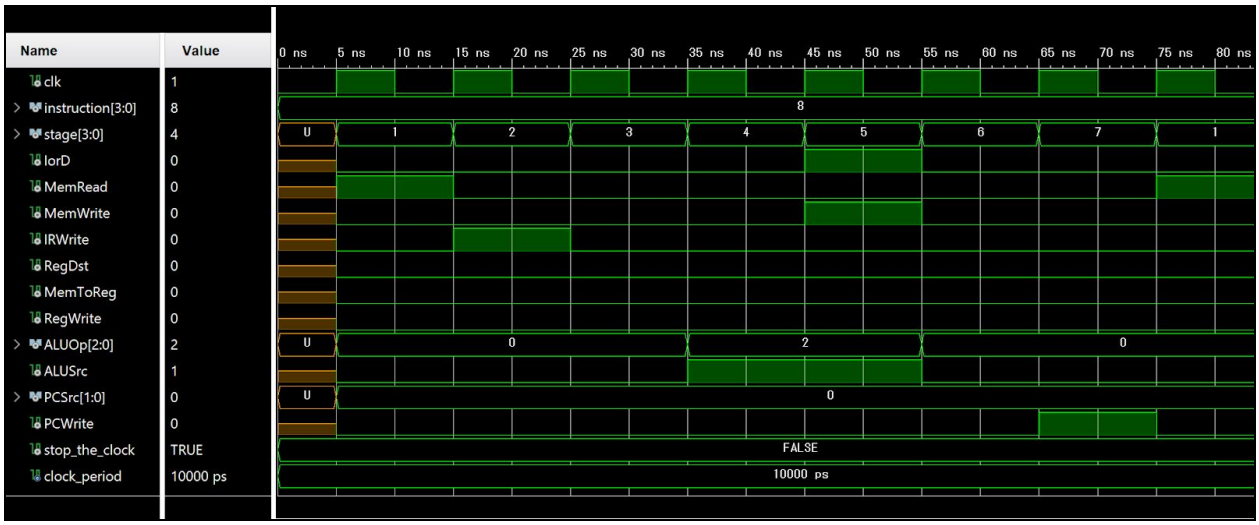


Figure 45: MCU Store Word Instruction Simulation Results

- In stage 1 (IF), MemRead = 1. This enables MemControl to read an instruction from memory.

- In stage 2 (ID), IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (REG), the MCU waits for the Registers block to output appropriate register contents. MCU also sets RegDst = 0 and MemToReg = 0 since this is a sw instruction.
- In stage 4 (ALU), the MCU sends ALUOp = 010 (addition) to the ALU, thus allowing it to add the offset to the base address from register. ALUSrc = 1 since the 32-bit sign-extended immediate value is used.
- In stage 5 (MEM), the MCU sends IorD = 1 and MemWrite = 1 to MemControl, thus writing the contents of read register 2 at the appropriate memory address.
- In stage 6 (PC), the MCU sends the appropriate PCSrc = 00 signal to the PCU since this is an R-Type instruction. Accordingly, the PCU increments current PC by 4.
- In stage 7 (PCW), PCWrite = 1, which updates current PC to NPC.

6. *Load Word Instruction:*

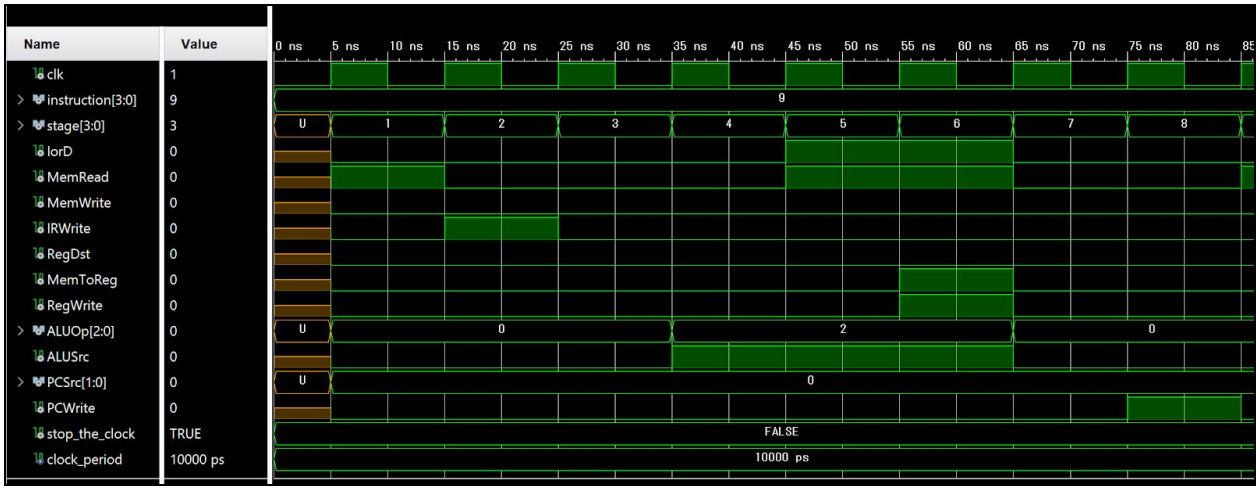


Figure 46: MCU Load Word Instruction Simulation Results

- In stage 1 (IF), MemRead = 1. This enables MemControl to read an instruction from memory.
- In stage 2 (ID), IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (REG), the MCU waits for the Registers block to output appropriate register contents.
- In stage 4 (ALU), the MCU sends ALUOp = 010 (addition) to the ALU, thus allowing it to add the offset to the base address from register. ALUSrc = 1 since the 32-bit sign-extended immediate value is used.
- In stage 5 (MEM), the MCU sends IorD = 1 and MemRead = 1 to MemControl, thus reading a word at the appropriate memory address.

- In stage 6 (WB), the MCU sends MemToReg = 1 and RegWrite = 1, which effectively writes the data loaded from memory to the appropriate register.
- In stage 7 (PC), the MCU sends the appropriate PCSrc = 00 signal to the PCU since this is an R-Type instruction. Accordingly, the PCU increments current PC by 4.
- In stage 8 (PCW), PCWrite = 1, which updates current PC to NPC.

7. *Unidentified Instruction:*

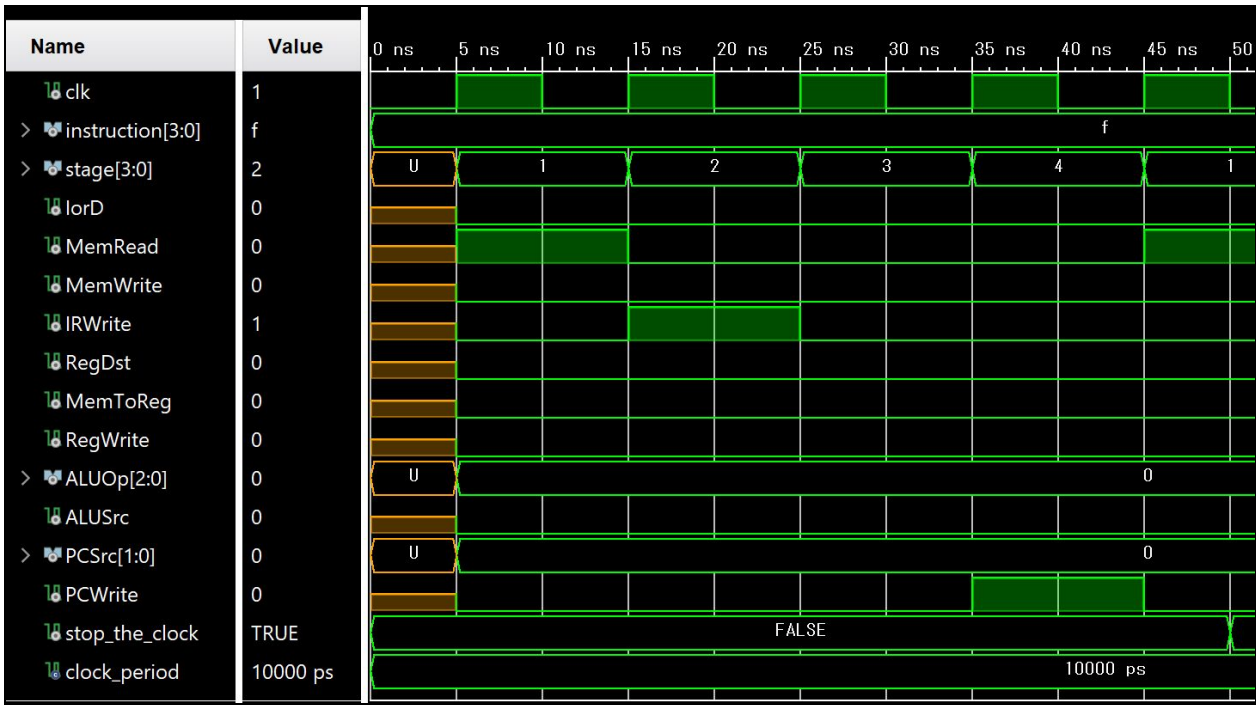


Figure 47: MCU Unidentified Instruction Simulation Results

- In stage 1 (IF), MemRead = 1. This enables MemControl to read an instruction from memory.
- In stage 2 (ID), IRWrite = 1. This enables this fetched instruction to be sent into the IR, which then decodes the instruction and parses its different parts. This is also when the 16-bit immediate value is sign-extended to 32 bits.
- In stage 3 (PC), the MCU sends the appropriate PCSrc = 00 signal to the PCU since this is an invalid instruction. Accordingly, the PCU increments current PC by 4.
- In stage 4 (PCW), PCWrite = 1, which updates current PC to NPC.

As seen from the above tests, the MCU is successfully able to provide accurate control signals to the components of the CPU, thereby enabling a smooth and synchronized execution of the MIPS instructions.

Putting Em Together

With each component working perfectly on their own, the only thing that remains is to put them all together. Starting with the Main Control Unit as a base, each component was added one at a time, to ensure that there were no timing issues when the components ran parallelly.

Starting With The MCU...

To start building the CPU, a new VHDL source file named "CPU" was created. The behavioral entity MCU created earlier was added to this CPU. The only input to the CPU is the CLK signal, which is inputted into the MCU entity (as it needs it to synchronize all CPU components). There is also an output "stage" from the CPU that exists purely for testing purposes — by looking at the stages outputted for each clock cycle, the viewer can tell, to some extent, if the appropriate instructions are being executed (as different types of instructions have different number of stages). The schematic for this base CPU containing just the MCU is given below. Since this model only contains the MCU unit, which was already shown to work perfectly in the previous section, it was not tested again.

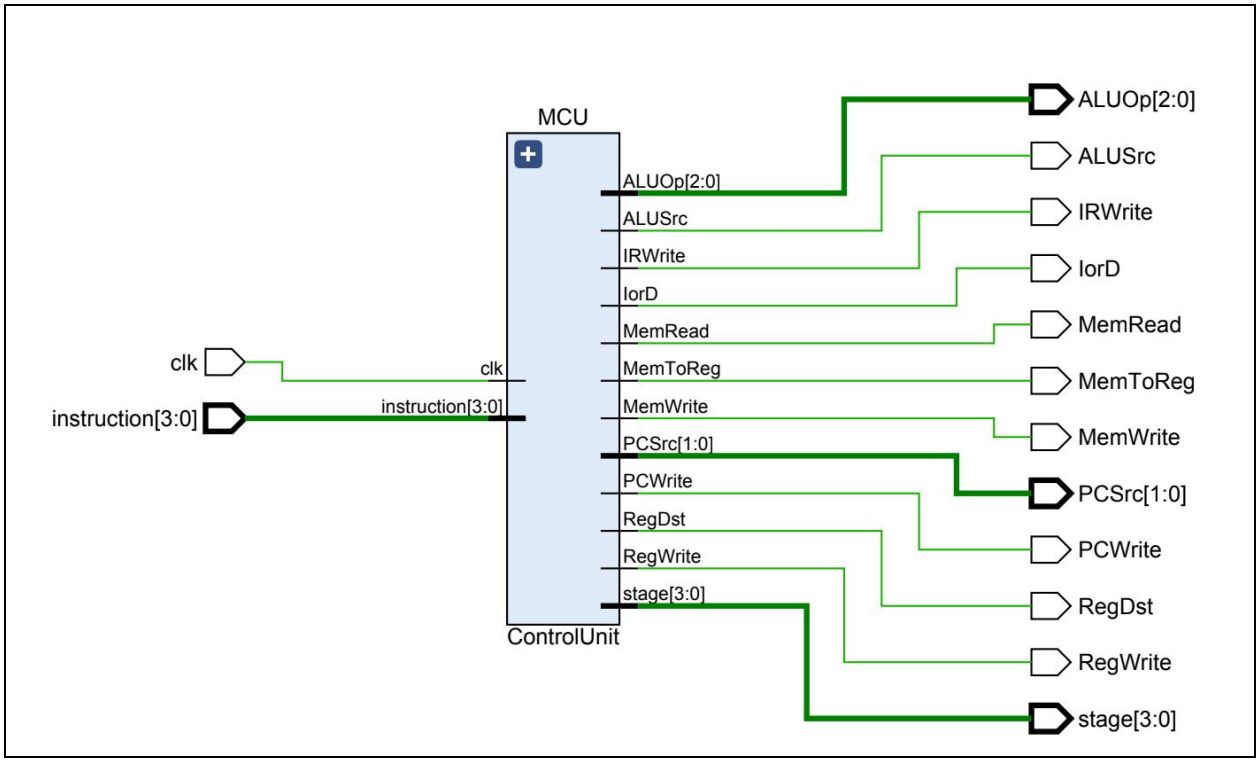


Figure 48: CPU Schematic — Adding The MCU

Adding The PCU…

Going in the sequence of regular execution, it makes sense to add the PCU component to the CPU since any program would need a PC to start at. Accordingly, this behavioral entity was added onto the CPU model and connections were made between it and the MCU. The revised schematic is given below.

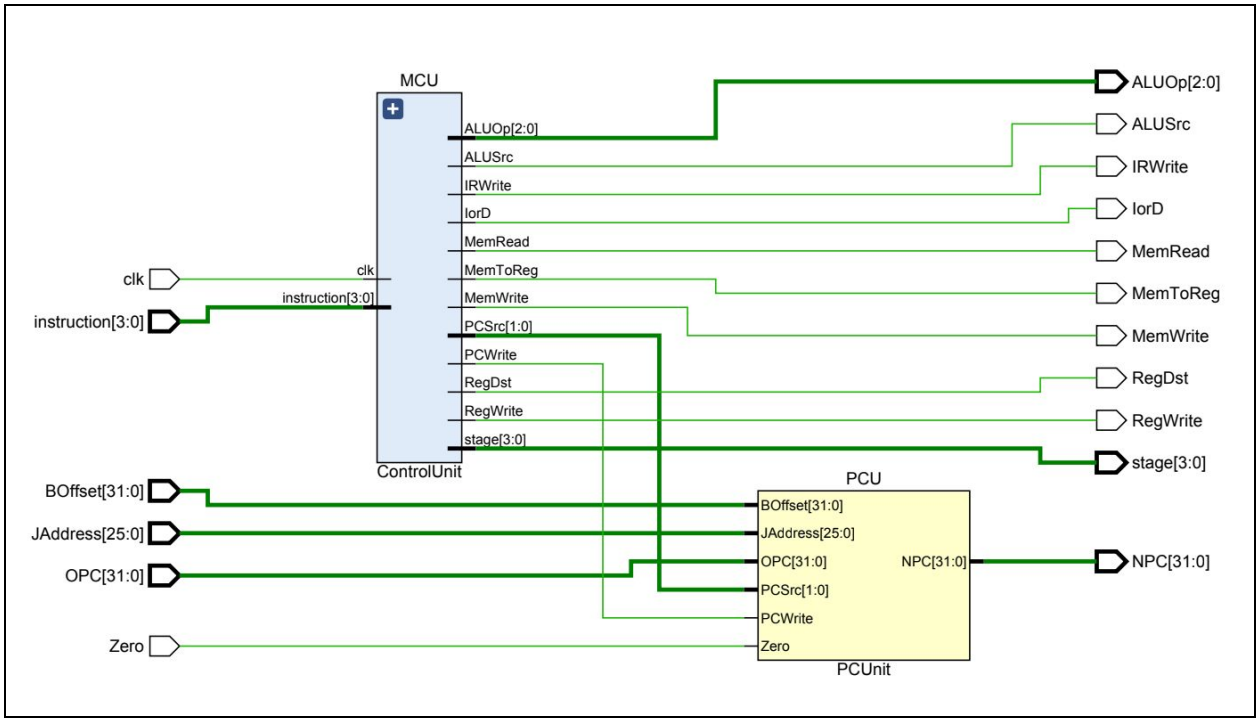


Figure 49: CPU Schematic — Adding The PCU

The main function of the PCU is to update the value of the current PC based on the instruction being executed. In our limited instruction set, there are only 3 different kinds of PC updates possible:

1. PC → PC + 4 ~~~ for add, ori, lw, sw, etc.
2. PC → (PC+4) + (Branch Offset << 2) ~~~ for beq
3. PC → PC(31:28) + (26-Bit Jump Address << 2) ~~ for j

The test bench given below was used to test all these cases. Here, the CPU executes the following series of instructions one after the other:

1. *j instruction*
2. *beq instruction (branch not taken)*
3. *beq instruction (branch taken)*
4. *add instruction*
5. *sw instruction*
6. *lw instruction*

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock
  OPC <= x"00000000"; -- Initial PC
  JAddress <= "10000000000000000000000001"; -- Jump to 0x08000004
  BOffset <= x"0000003f"; -- Branch to 0x000000fc
  Zero <= '0'; -- Branch Condition = False

  instruction <= "0000"; -- j instruction
  wait for 45 ns; -- wait

  instruction <= "0001"; -- beq instruction
  wait for 64 ns; -- wait

  instruction <= "0001"; -- beq instruction
  Zero <= '1'; -- Branch Condition = True
  wait for 64 ns; -- wait

  instruction <= "0010"; -- add instruction
  wait for 64 ns; -- wait

  instruction <= "1000"; -- sw instruction
  wait for 64 ns; -- wait

  instruction <= "1001"; -- lw instruction
  wait for 80 ns; -- wait

  stop_the_clock <= true; -- end clock


  wait;
end process;
```

Figure 50: CPU Test Bench — Adding The PCU

The results of this test have been discussed below.

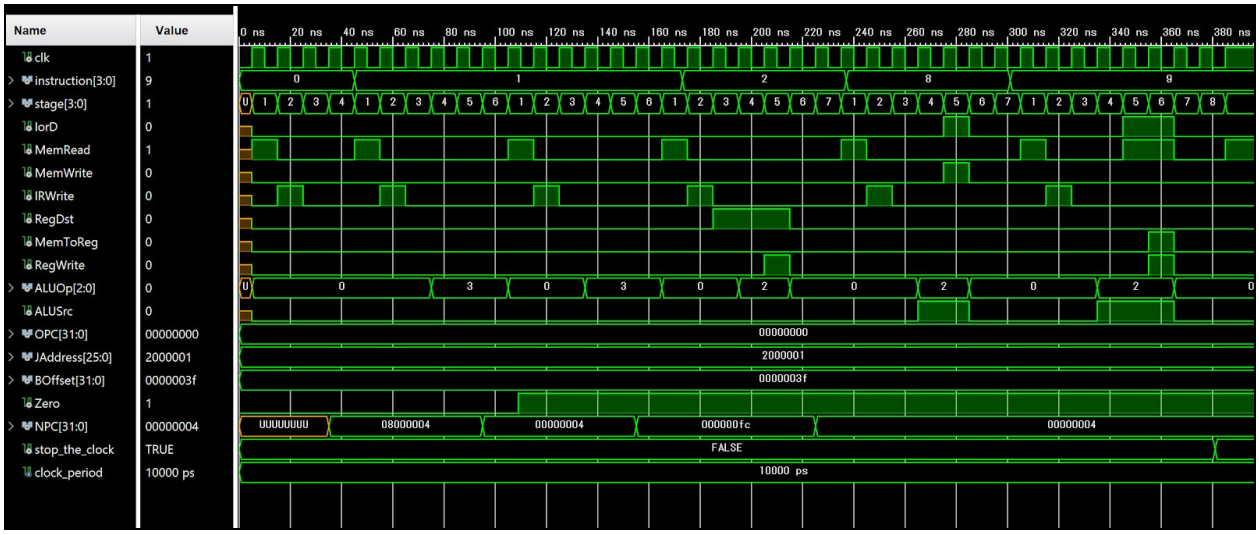


Figure 51: CPU Simulation Result — Adding The PCU

1. *j instruction*
   Initially, NPC is undefined. This is because the CPU starts running at the first rising edge of the clock, which is at about 5 ns. The j instruction takes 4 cycles to execute. The 26-Bit Jump Address inputted to the PCU was 0x200000001 as can be seen in the above figure. The upper 4 bits of PC are 0000, which is then added to the Jump Address left-shifted 2 places, giving the resulting address of 0x80000004. As can be seen at the end of the 4th cycle of the jump instruction (at around 40 ns), the value of NPC after the jump instruction was 0x80000004, which is exactly what we predicted. Therefore, the jump instruction works perfectly in terms of PC update.

2. *beq instruction (branch not taken)*
   The next instruction, which is the beq with branch not taken, starts right after the 4th cycle of the jump instruction at about 40 ns. Note that the current PC at the start of this second instruction is still 0x0 even though NPC was updated because the NPC output has not yet been connected back to the current PC input to make these initial test cases easier to work with; effectively, every instruction starts at PC = 0x0. Branch takes 6 cycles, which ends at around 100 ns. Since branch was not taken in this case, a regular PC + 4 update was performed which can be seen in the figure above; NPC is now 0x4.

3. *beq instruction (branch taken)*
   This time branch was taken. The branch offset input to the PCU was 0x3f as can be seen in the above image. When left-shifted 2 places, this value becomes 0xfc, which is added to current PC (i.e. 0x0) to give 0xfc. This result is also correct, as can be seen at the end of the 6th cycle of the beq instruction at about 160 ns.

4. *add/sw/lw instruction*
   The next three instructions all use the regular PC+4 update. Accordingly, their final NPC should be 0x4 as can be seen starting at about 220 ns.

From the above results, it is clear that our current MCU+PCU combination works perfectly without any timing or other issues.

Adding The Memory Controller...

Now that we have a PC, we can use it to fetch instructions from Memory. But before doing that, we need to add in the MemControl component to the CPU. After doing so, our schematic now looks as follows.

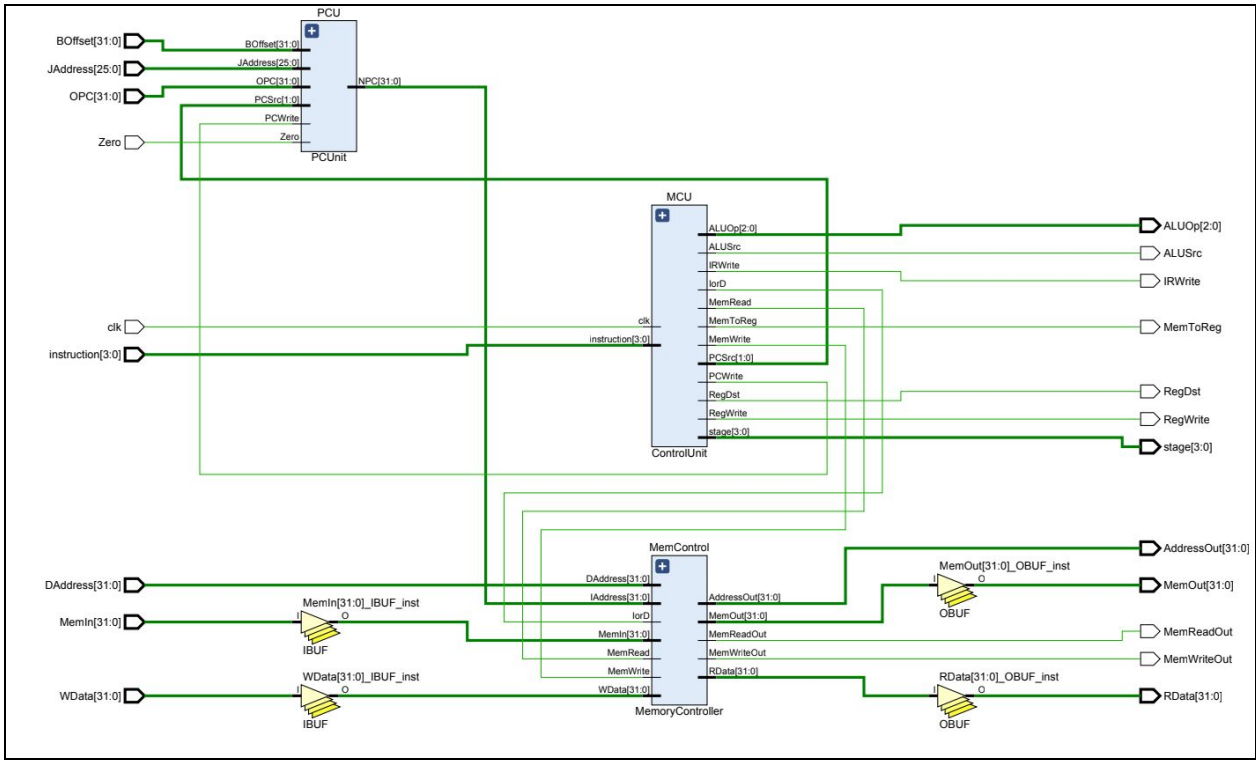


Figure 52: CPU Schematic — Adding The Memory Controller

Since we haven't yet added the actual memory to store data, the only thing we can test at the moment is whether the MemControl unit successfully forwards data and control signals between the CPU and Cache. Accordingly, the following two tests were performed.

1. LW Test → To check read operations

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  instruction <= "1001"; -- lw instruction
  OPC <= x"00000000"; -- Initial Address
  MemIn <= x"12345678"; -- Data Being Read

  wait for 100 ns; -- Wait

  stop_the_clock <= true; -- end clock


  wait;
end process;
```

Figure 53: CPU Test Bench (LW Test) — Adding The Memory Controller

2. SW Test → To check write operations

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  instruction <= "1000"; -- sw instruction
  OPC <= x"00000000"; -- Initial Address
  WData <= x"12345678"; -- Data Being Written

  wait for 100 ns; -- Wait

  stop_the_clock <= true; -- end clock


  wait;
end process;
```

Figure 54: CPU Test Bench (SW Test) — Adding The Memory Controller

Each of these two test benches was simulated and results were recorded. These, along with analyses, are given below. Only those results that are relevant at this point are discussed.

1. LW Test

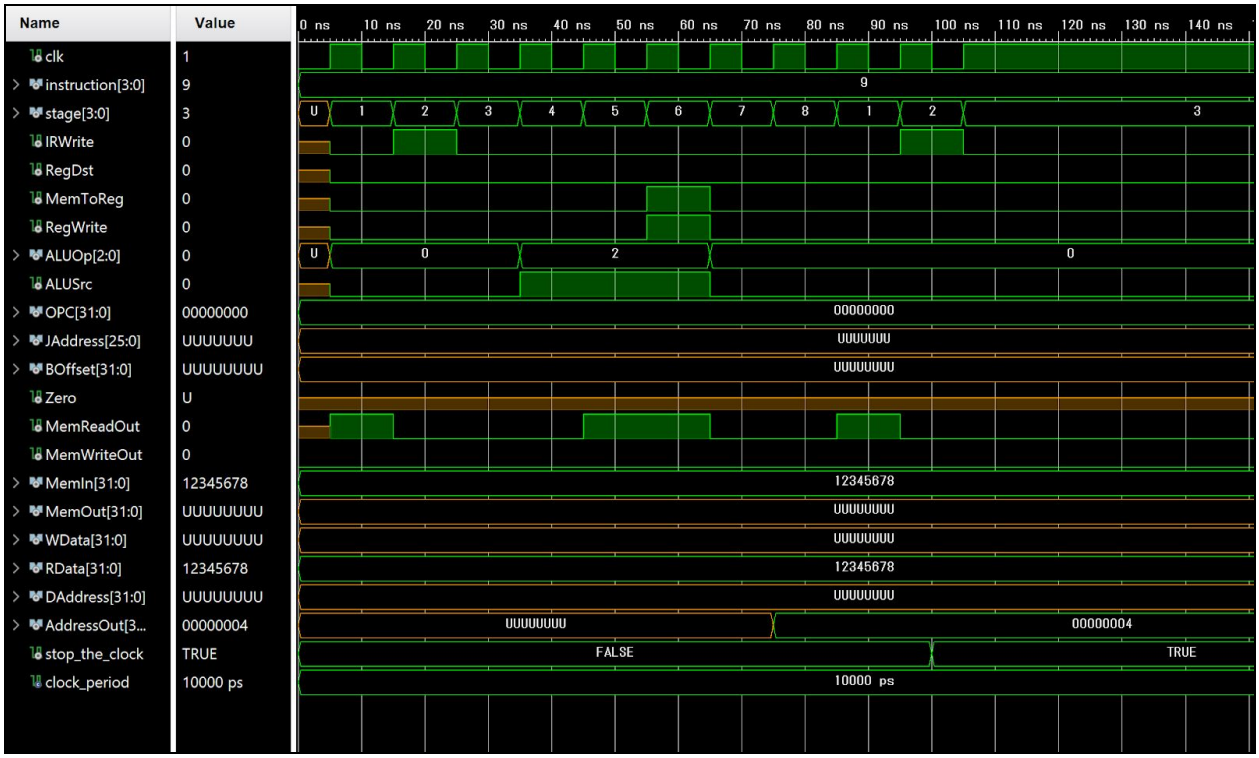


Figure 55: CPU Simulation Results (LW Test) — Adding The Memory Controller

In the above figure, one can see that MemReadOut = 1 for cycles 2 (IF) 5 (MEM) & 6 (WB), which means that the MemControl unit is successfully forwarded the MemRead signal to the Cache. The Address is also successfully forwarding via AddressOut (0x4), as can be seen

starting at clock cycle 8 (when address is actually valid as a result of PC update). Finally, MemIn, i.e data received from the Cache, which has the value 0x12345678, is also sent to the CPU on the RData bus. These are the only three forwarding tasks the MemControl needs to perform in a read operation, and they all work well.

2. SW Test

Figure 56: CPU Simulation Results (SW Test) — Adding The Memory Controller

This test is the polar opposite of the previous one, except that the address forwarded on the AddressOut is still 0x4 (clock cycle 7), which is what we would expect. One can also see that MemWriteOut = 1 for 5 (MEM), which means that the MemControl unit is successfully forwarding the MemWrite signal to the Cache. Finally, WData, i.e data to be written to memory, which has the value 0x12345678, is also sent to the Cache on the MemOut bus. Again, as before, these are the only forwarding tasks the MemControl needs to perform.

Clearly, there aren't any issues after adding the MemControl unit to the base CPU, but no errors were expected anyway as the only task performed by the MemControl is that of forwarding data to and from the Cache, which is trivial and less prone to errors.

Adding The Cache...

Finally, we can hook up the Cache with the MemControl unit and actually test read/write operations to memory. But before that, here is the updated schematic for the CPU.

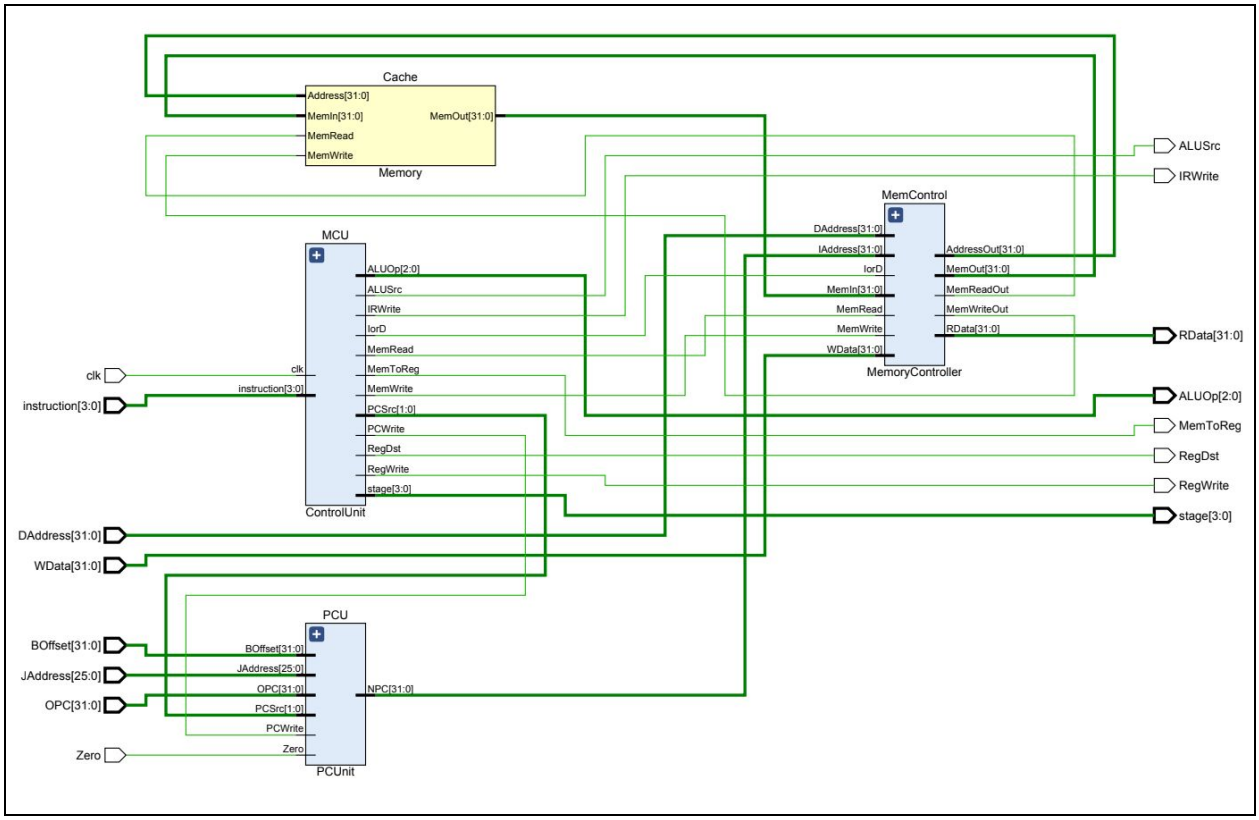


Figure 57: CPU Schematic — Adding The Cache

Using this new CPU, a single test was performed in which data was first written to memory followed by a read operation to that same memory location to ensure that the data was actually written. Thus, both read and write operations were tested simultaneously. No distinction was made between reading/writing instructions vs data as the previous tests conducted (in the Cache section) have already provided sufficient evidence that the current Cache implementation only allows a write to Data memory. Therefore, this was not tested again. The test bench code used is given below.

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  instruction <= "1000"; -- sw instruction
  OPC <= x"00000000"; -- Initial Address
  WData <= x"12345678"; -- Data Being Written
  DAddress <= x"0000000c"; -- Address to Store Data
  wait for 80 ns; -- Wait

  instruction <= "1001"; -- lw instruction
  wait for 100 ns; -- Wait

  stop_the_clock <= true; -- end clock

  wait;
end process;
```

Figure 58: CPU Test Bench — Adding The Cache

The simulation results were as follows:

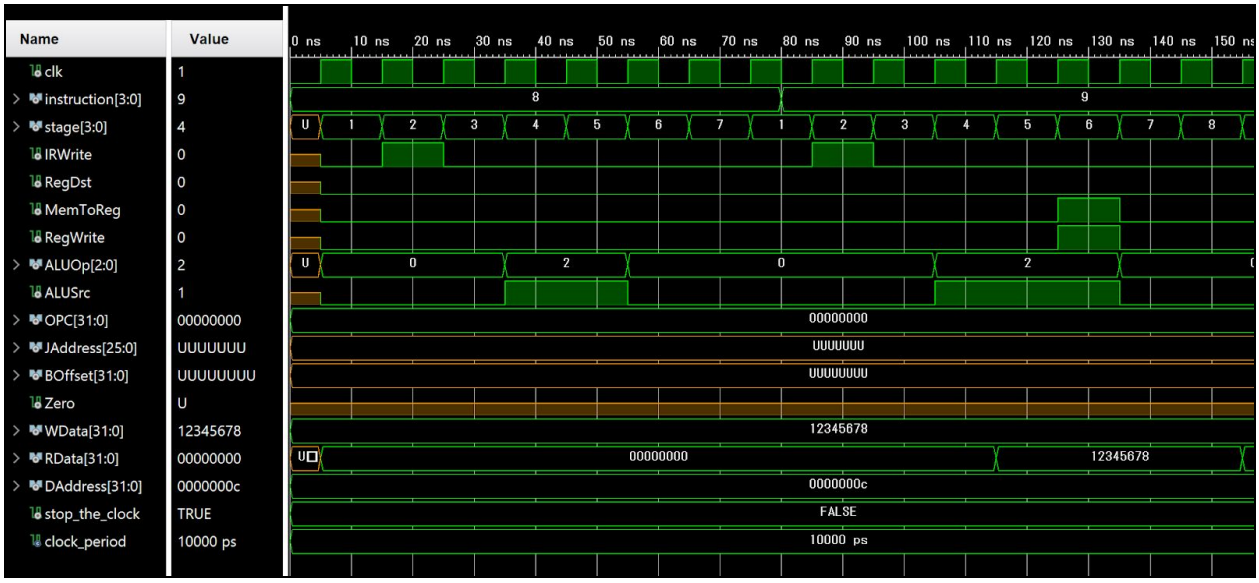


Figure 59: CPU Simulation Results — Adding The Cache

First thing to note here is that this is a sw operation (which takes 7 cycles) followed by a lw instruction (which takes 8 cycles). This can be easily verified by looking at the number of cycles for which each instruction ran — 7 cycles followed by 8 (from the figure). Initially, WData = 0x12345678, which is the data we want to write to memory, and RData = 0x0 (data received from memory). We work by assuming that the sw instruction was successful in writing the data to memory, and then all we have to do to verify that is to check the data read by the lw instruction. This read is done in the MEM stage, i.e. cycle 5 of the lw instruction. Accordingly, one can see that at around 115 ns, RData changed from 0x0 to 0x12345678. Therefore, data was successfully written to and read from memory; our CPU works perfectly thus far.

Adding The Instruction Register...

Having a properly functioning mechanism to read instructions from memory, the next step is to actually decode them. Accordingly, the IR was the next component added to the CPU (see the updated schematic below).

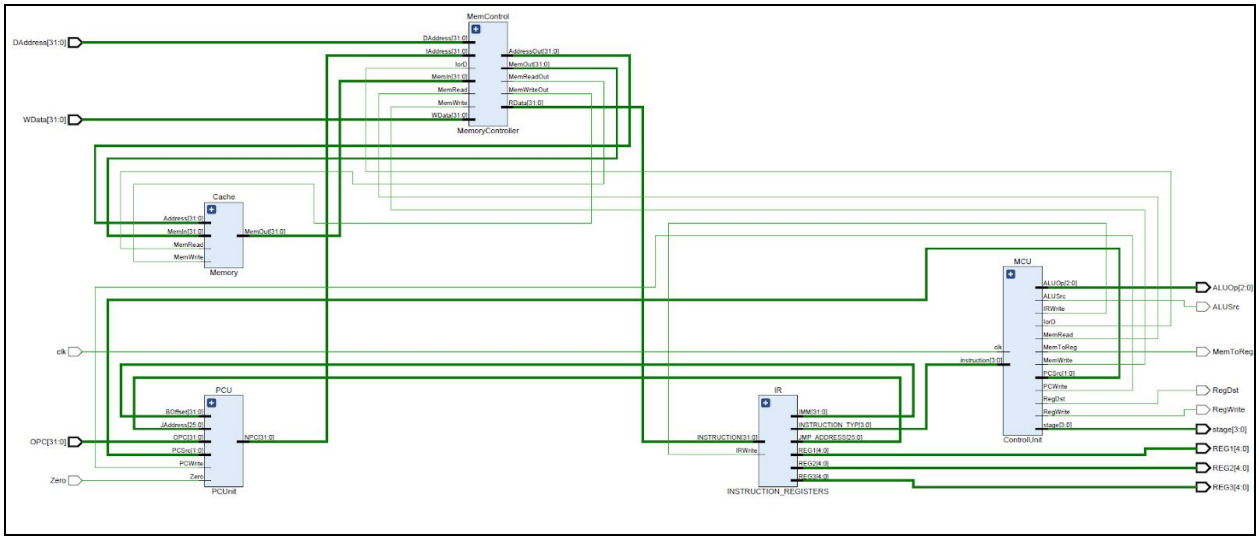


Figure 60: CPU Schematic — Adding The Instruction Register

Since the IR computes the exact same arguments (branch offset, read register indices, etc.) for all instructions, regardless of whether or not they use it, testing a single instruction here should suffice. Accordingly, a simple test bench executing the j instruction is given below.

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  OPC <= x"00000000"; -- Initial Address
  -- Jump instruction is stored at addresses 0x0, 0x4 & 0x8
  wait for 200 ns; -- Wait

  stop_the_clock <= true; -- end clock

  wait;
end process;
```

Figure 61: CPU Test Bench — Adding The Instruction Register

On simulating the above VHDL code, the following output was obtained:

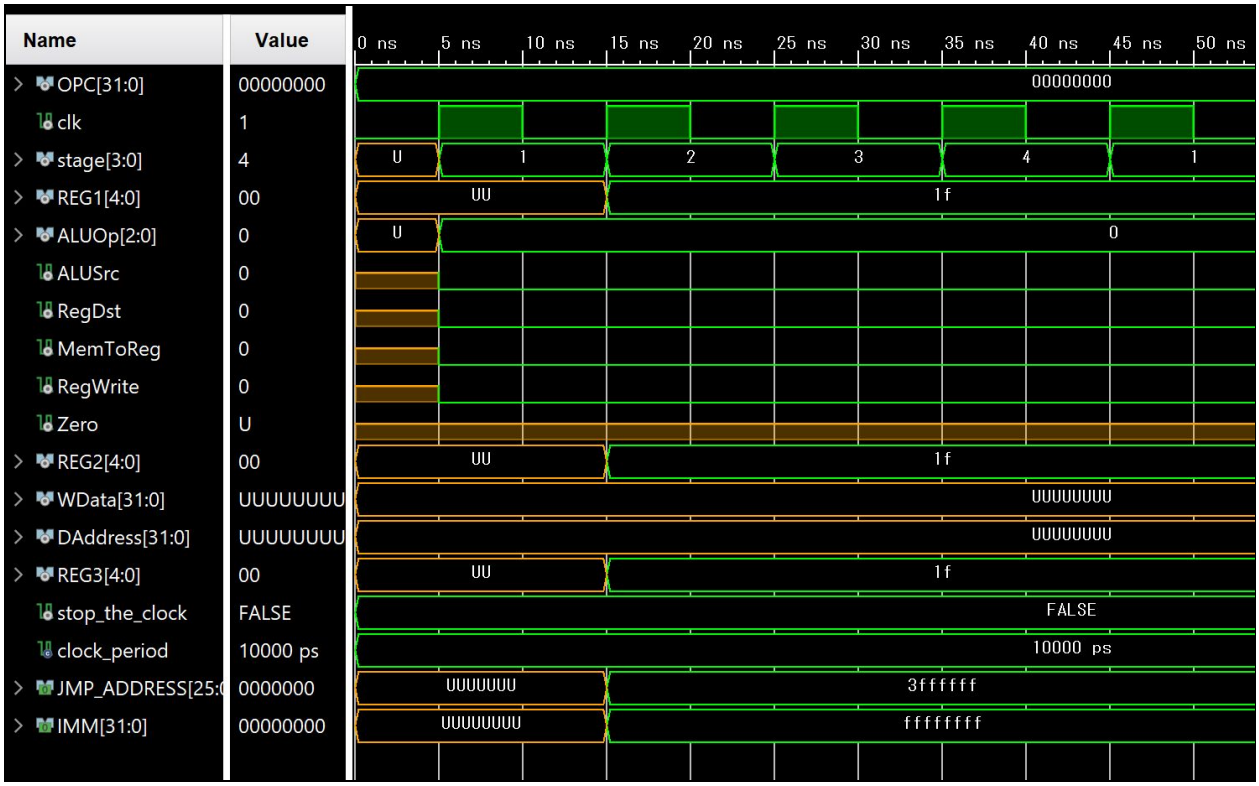


Figure 62: CPU Simulation Results — Adding The Instruction Register

The jump instruction loaded from memory is given by the following machine code:
00001011111111111111111111111111

Based on the above machine code instruction, you would expect the following results:

- instruction = 0000 (the value chosen for the j instruction)
- REG1 = 11111 or 0x1f
- REG2 = 11111 or 0x1f
- REG3 = 11111 or 0x1f
- JMP_ADDRESS = 11111111111111111111111111 = 0x3fffffff
- IMM = 0xffffffff

Confirm from the figure above that all these results are correct.
The last thing to test in the IR is the functioning of the IRWrite control signal. This signal is asserted in the ID stage, i.e. cycle 2 of the j instruction. Notice that the values of all the outputs mentioned above are undefined before this point and are accurate after. This shows that ID took place only after IRWrite was asserted. Therefore, our CPU is still working perfectly.

Adding The Registers Block...

With only two components left to be added, the next one in the sequence is the Registers Block. Here is the new schematic:

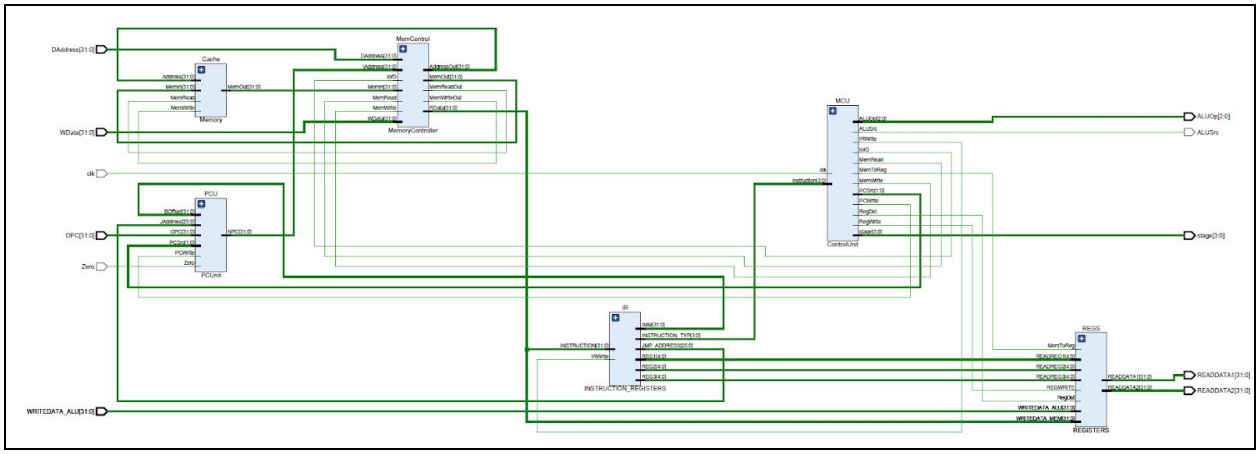


Figure 63: CPU Schematic — Adding The Registers Block

Two things need to be tested in terms of operation of the Registers block: output of register contents and writing to registers. Both of these were performed in a single test given by the following test code:

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  OPC <= x"00000000"; -- Initial Address
  DAddress <= x"00000044"; -- Memory Address to LW from (Contents = 0xffffffff)

  -- LW instruction is stored at addresses 0x0, 0x4 & 0x8
  wait for 200 ns; -- Wait

  stop_the_clock <= true; -- end clock

  wait;
end process;
```

Figure 64: CPU Test Bench — Adding The Registers Block

The following simulation results were obtained:

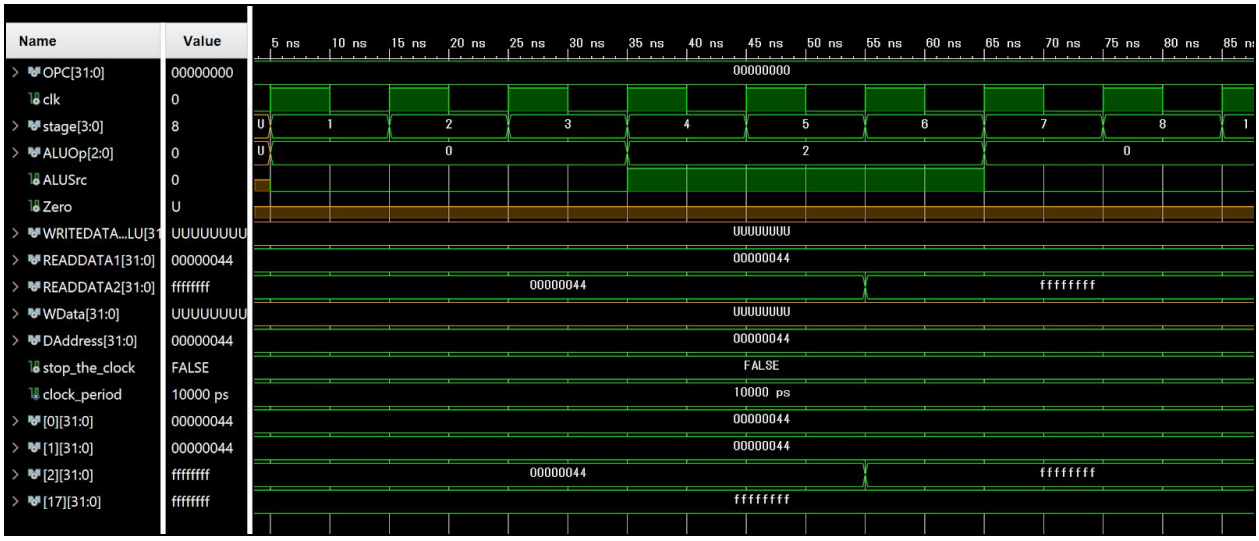


Figure 65: CPU Simulation Results — Adding The Registers Block

The instruction used for this test, stored at memory location 0x0, was a lw instruction. The machine code for this instruction is given by the following:
10001100001000100000000000000000
Here, base register index = 1, destination register index = 2, Offset = 0x0.

Since the ALU isn't added to the CPU yet, the sum of the base register contents and offset was manually inputted into the MemControl unit. This address was 0x44, which is given by memory location 17 (decimal). Note that the memory designed here uses word-sized blocks and so memory location 17 is the 17th word not byte.

The data stored at this memory location is 0xffffffff, as can be seen from the above figure; this is the data that needs to be loaded in register 2. One can see that register 2 does have the value 0xffffffff at the end of the instruction's execution. Therefore, the CPU has passed the first test.

The second thing that needs to be checked, as mentioned before, are the register contents outputted by the Registers block. Verify that the contents of register 1 (0x44) and register 2 (initially 0x44 and then 0xffffffff) are properly outputted on the READDATA1 and READDATA2 registers. Therefore, this operation works fine too. So it is clear that the Registers block is fully functional.

Adding The ALU...

Finally, the ALU is now connected to the CPU. As this is the last component, adding it effectively loops the entire CPU, leaving only the OPC input, the CLK signal and the stage output. This can be seen in the schematic below.

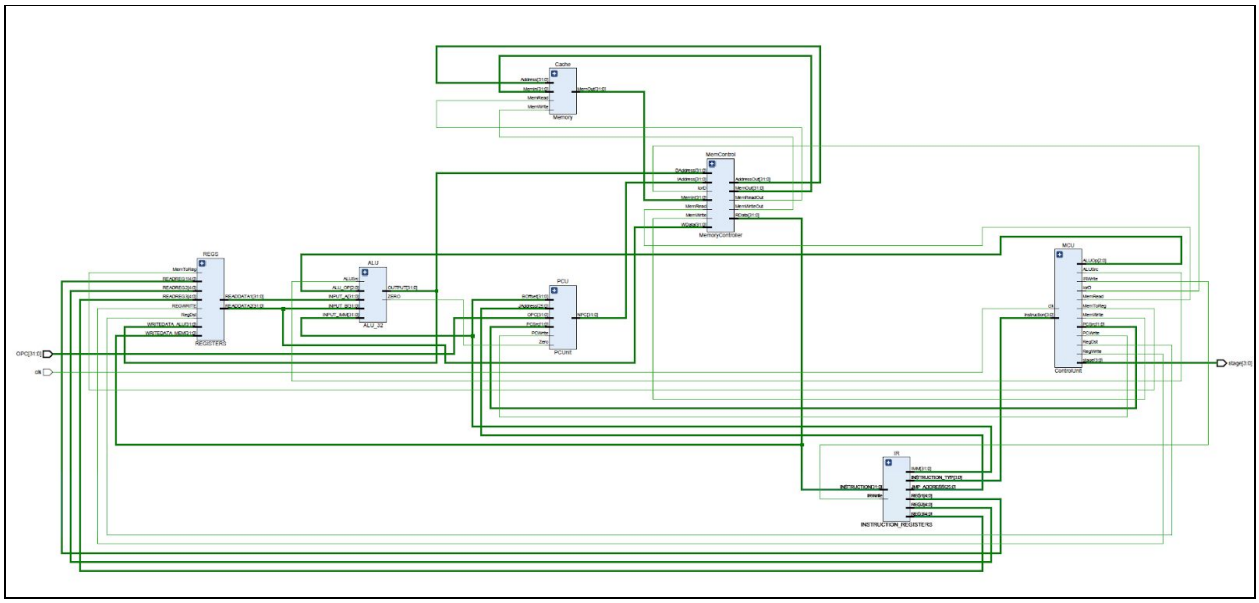


Figure 66: CPU Schematic — Adding The ALU

The current version of the CPU is fully capable of executing any instruction from the supported Instruction Set. However, it can only execute one instruction at a time since the OPC input to the PCU has not been connected yet (for reasons explained later), which means that PC is never updated and is always 0x0 NPC is computed and outputted, but it is not written back to the PC register in the PCU. Accordingly, only the instruction at memory location 0x0 can be executed.

To test that the CPU works fine, a test bench that executes a single add instruction is used; tests for other instructions will be done in a later section. The test bench code is given below.

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  OPC <= x"00000000"; -- start address of add instruction

  wait for 1000 ns;-- for 105 ns; -- Wait

  stop_the_clock <= true; -- end clock

  wait; -- Wait Indwfinitely
end process;
```

Figure 67: CPU Test Bench — Adding The ALU

The simulation results were as follows:

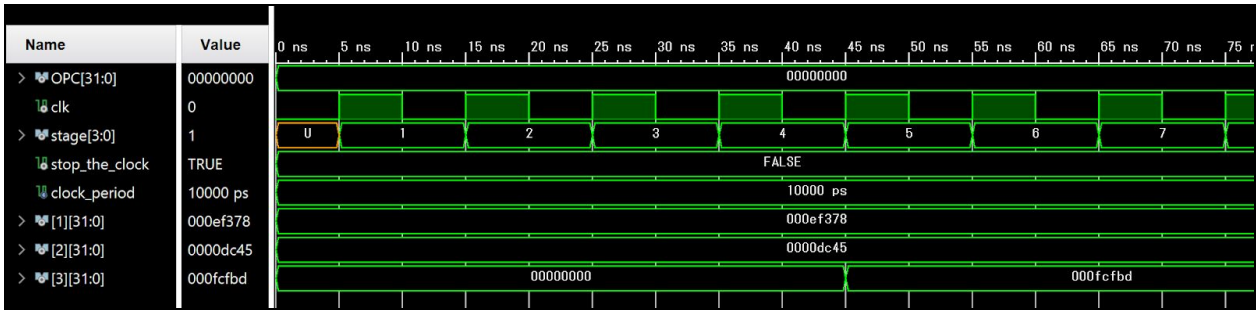


Figure 68: CPU Simulation Results — Adding The ALU

This simple test performs an addition of two numbers stored in register 1 & 2 respectively and writes the result back to register 3.

From the figure above, one can see that initially, register 1 = 0x000ef378, register 2 = 0x0000dc45, and register 3 = 0x00000000. If we add the contents registers 1 & 2, we get the result 0x000fcfbd. Notice that register 3 does contain this exact value at the end of the add operation. Therefore, the CPU is still fully functional.

Looping The Loop...

The final connection that remains is the one from the NPC to the OPC input of the PCU. By adding this connection, instructions will now be able to run from sequential memory locations. Recall that, earlier, every instruction started at PC = 0x0 due to lack of this connection.

The reason this was put off until the end was that adding the connection early on seems to give a weird bug; when calling the jump instruction, the target address computed by the PCU always had 4 most significant bits as undefined. The reason for this was the fact that since NPC is connected to OPC, this forms a circular loop with no start or end. Accordingly, there was no initial value for PC, and though the lower 28 bits had definite values, which were taken from the jump instruction, the upper 4 bits were undefined.

To fix this issue, a register (VHDL variable) was added to the PCU to buffer the output to NPC. By adding this register, it was now possible to initialize the PC to 0x0 by setting the initial value of this register to 0x0. Thus, the problem was fixed, and the final schematic for the CPU is as follows:

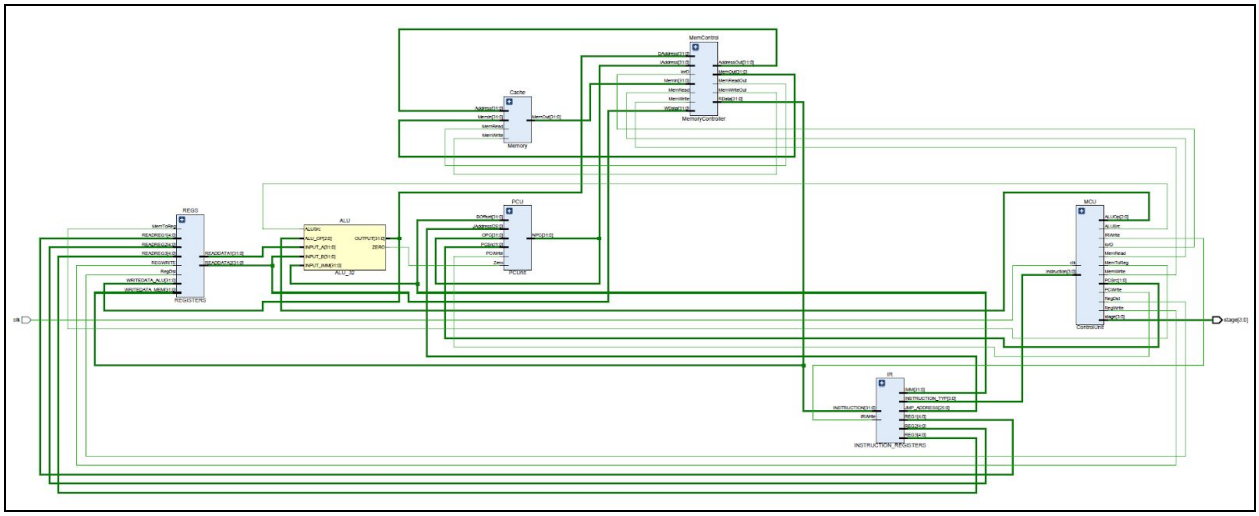


Figure 69: CPU Final Schematic

To test this, a stream of instructions were executed. As the only difference made by adding this new connection was that consecutive instructions in memory can now be executed, this was the only thing that was tested here. The test bench, given below, simply executed a lw instruction, followed by an add, and then a j instruction, which were stored at memory locations 0x0, 0x4, and 0x8 respectively.

```
stimulus: process
begin

  stop_the_clock <= false; -- start clock

  wait for 1000 ns;-- for 1000 ns; -- Wait

  stop_the_clock <= true; -- end clock

  wait; -- Wait Indwfinitely
end process;
```

Figure 70: CPU Final Test Bench

The following simulation results were obtained:

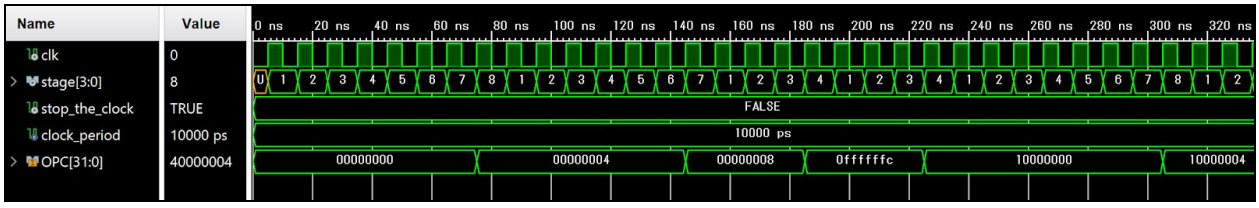


Figure 71: CPU Final Simulation Results

From the figure, one can see that the first instruction takes 8 clock cycles to execute; in the supported Instruction Set, lw is the only instruction that runs for 8 cycles, and so we know that the correct instruction has been executed. Note, however, that the next instruction takes 7 cycles (add instruction), and the one after that (j instruction) takes 4 cycles. Also, all instructions after that take 4 cycles each since they are all invalid — 4 cycles is the minimum any instruction can take in this design (IF + ID + PC + PCW). The main thing here is that the same instruction is not being executed repeatedly; consecutive instructions in memory are being executed and the PC is being updated correctly (refer to the figure). Therefore, this final connection also works just as expected.

A Full-Fledged MIPS Program

It's time to test a complete MIPS program using a bunch of different instructions to perform a specific task. In this simple program, two word-sized numbers are loaded from memory into registers and their output is written back to a third memory location.

→ *MIPS Code:*

0x00000000	lw $t1, 0($t0)	# Load first number
0x00000004	lw $t2, 4($t0)	# Load second number
0x00000008	add $v0, $t2, $t1	# Add the two numbers
0x0000000C	sw $v0, 8($t0)	# Save it to memory

→ *Setup:*

The only thing required to be performed in the test bench here is the starting and stopping of the CLK. Other than that, initial values need to be loaded into registers and memory. Registers, by default, were initialized to 0, though $t0 was loaded with the value 0x44, which is the address to the first of two consecutive numbers in memory.

The following values were stored in memory. Note that, here, "Word Number" indicates the data's integer position in the memory with block size of a word (4 bytes).

Instructions		Data	
Address	Value	Address (Word Number)	Value
0x00000000	0x8D090000	0x00000044 (17)	8
0x00000004	0x8D0A0004	0x00000048 (18)	9
0x00000008	0x012A1020	0x0000004C (19)	0
0x0000000C	0xAD020008		

All other memory locations contain 0x0.

→ *Simulation Results:*

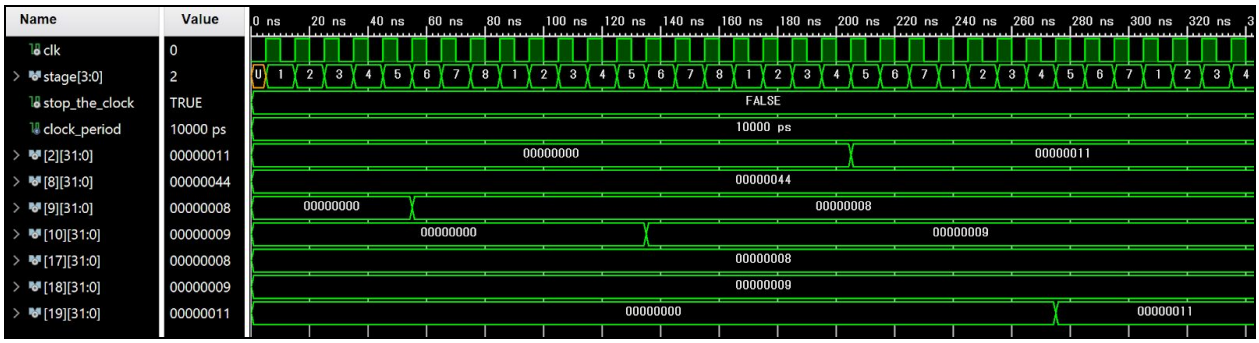


Figure 72: Full-Fledged Program Simulation Results

As can be seen from the above figure, register 8 ($t0) initially contains the base address 0x44 and the memory locations 17 & 18 contain 0x8 & 0x9 respectively. Register 2 ($v0) & 10 ($t2), as well as memory location 19 initially contain 0x0.

After the first lw instruction, at around 60 ns, notice that the contents of register 9 ($t1) changed from 0x0 to 0x8, which was stored in memory location 17 ((0x44 + 0)/4). Similarly, after the second lw instruction at around 130 ns, register 10 ($t2) gets the value 0x9 from memory location 18 ((0x44 + 4)/4).

After the add instruction, at around 200 ns, register 2 ($v0) gets the value 0x11, which is the sum of 0x8 & 0x9. Finally, after the sw instruction at around 270 ns, memory location 19 ((0x44 + 8)/4) gets the value 0x11 from register 2 ($v0).

As can be seen above, the program executed flawlessly. This provides conclusive proof that the designed CPU is able to execute any of the instructions from the supported Instruction Set, and can be used to create programs with any level of complexity.

Conclusions and Future Scope

As it is clear from the obtained results, we have successfully built a 32 bit RISC CPU capable of running a set of 13 different instructions. There were a few issues along the way as the student gained valuable knowledge about VHDL programming. The things that gave the most trouble was connecting the components and getting them to work together with the CLK. With enough practice, these issues should not arise in the future.

By the end of implementation, students gained a good deal of familiarity with writing VHDL code and how to implement a CPU. In future, the CPU can be optimized; a few suggestions include: (1) using a CLA (Carry Look Ahead) adder instead of the existing ripple-carry one, which would enable the ALU & PCU to be faster in computing their outputs; (2) implementing pipelining to execute multiple instructions simultaneously, thus increasing throughput; and (3) adding exception handling features to allow for a more robust implementation of the CPU. As for right now, each instruction must completely finish before the next can not start. In a pipelined-CPU, there is an inherent need for hazard detection because multiple instructions executing simultaneously could now be trying to access a data that has not been updated (data hazard) or trying to use the same component at the same time (structural hazard). The data hazard issue could later be minimized by allowing other instructions to use values after they have been executed in the ALU but before being put into the correct register. Clearly, this is out of the scope of the current project, but future developers may assess the benefits inherent in such strategies.

References

1. Digi-Key's Scheme-it Tool

   Link: https://www.digikey.com/schemeit/

   Description: This is an online design tool that allows the user to draw flowcharts, circuits, and many other types of diagrams. The Datapath figure in the introduction was drawn using this tool.

2. Test Bench Generator

   Link: https://www.doulos.com/knowhow/perl/testbench_creation/

   Description: This is an online software used to generate a test-bench skeleton for a user's VHDL code, thereby reducing the unnecessary effort (of basically copying over signal definitions, library-inclusion statements, etc.) on the part of the user. The test-benches created for this project used this tool to provide the basic starting point.

3. Project 2 Description Document

   Description: This is the project outline given by the professor. The datapath image for the mips processor was used from there.

4. Computer Organization and Design, The Hardware/Software Interface
   By: David A. Patterson and John L. Hennessy

   Description: This is the book used for the class. It was used for all our conceptual definitions within this paper. It was also to help guide our construction of the data path for our RISK processor

Appendix

A. 1-Bit Full Adder

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: Full_Adder_1 - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Full_Adder_1 is
--This is the port map for the Full Adder
	Port ( A : in STD_LOGIC;
	B : in STD_LOGIC;
	Cin : in STD_LOGIC;
	S : out STD_LOGIC;
	Cout : out STD_LOGIC);
end Full_Adder_1;

architecture Behavioral of Full_Adder_1 is

begin
--This is the gate equivalent of the Full Adder
S <= A xor B xor Cin;
Cout <= (A and B) or (A and Cin) or (B and Cin);
end Behavioral;
```

B. 4-Bit Full Adder

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: CRA_4 - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CRA_4 is
--This is the port map for the Full Adder
	Port ( A : in STD_LOGIC_VECTOR(3 downto 0);
```

```
        B : in STD_LOGIC_VECTOR(3 downto 0);
        Cin : in STD_LOGIC;
        S : out STD_LOGIC_VECTOR(3 downto 0);
        Cout : out STD_LOGIC);
end CRA_4;

architecture Behavioral of CRA_4 is
--This is for the structural model, to create the component for the 4 bit CRA to made from.
component Full_Adder_1
        Port(A : in STD_LOGIC;
        B : in STD_LOGIC;
        Cin : in STD_LOGIC;
        S : out STD_LOGIC;
        Cout : out STD_LOGIC);
end component;
--These are the signals used to connect the carry bit to the next adder and Cout is the
Carry bit of the 4 bit Adder
        signal car1,car2,car3: STD_LOGIC;
begin
--Creates 4 Full Adders and breaking down the input of 3 downto 0 into bit strings for each
Full Adder, Cout for
--the 4 bit CRA is the Carry bit of only the 4th Full Adder
A1: Full_Adder_1 port map (A(0), B(0), Cin, S(0), car1);
A2: Full_Adder_1 port map (A(1), B(1), car1, S(1), car2);
A3: Full_Adder_1 port map (A(2), B(2), car2, S(2), car3);
A4: Full_Adder_1 port map (A(3), B(3), car3, S(3), Cout);

end Behavioral;
```

C. 32-Bit Full Adder

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: CRA_32 - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CRA_32 is
--This is the port map for the Full Adder
        Port ( A : in STD_LOGIC_VECTOR(31 downto 0);
        B : in STD_LOGIC_VECTOR(31 downto 0);
        Cin : in STD_LOGIC;
        S : out STD_LOGIC_VECTOR(31 downto 0);
        Cout : out STD_LOGIC);
end CRA_32;
```

```
architecture Behavioral of CRA_32 is
--This is for the structural model, to create the component for the 32 bit CRA to made from.
component CRA_4
--Port map for the 4 bit adder
	Port ( A : in STD_LOGIC_VECTOR(3 downto 0);
	B : in STD_LOGIC_VECTOR(3 downto 0);
	Cin : in STD_LOGIC;
	S : out STD_LOGIC_VECTOR(3 downto 0);
	Cout : out STD_LOGIC);
end component;
--These are the signals used to connect the carry bit to the next adder and Cout is the
Carry bit of the 32 bit Adder
	signal car1,car2,car3,car4,car5,car6,car7: STD_LOGIC;
begin
--Creates 8 4 bit CRA adders and breaking down the input of 31 downto 0 into 4 bit strings
for each 4 bit CRA, Cout for
--the 32 bit CRA is the Carry bit of only the 8th 4 bit CRA
A1: CRA_4 port map (A(3 downto 0), B(3 downto 0), Cin, S(3 downto 0), car1);
A2: CRA_4 port map (A(7 downto 4), B(7 downto 4), car1, S(7 downto 4), car2);
A3: CRA_4 port map (A(11 downto 8), B(11 downto 8), car2, S(11 downto 8), car3);
A4: CRA_4 port map (A(15 downto 12), B(15 downto 12), car3, S(15 downto 12), car4);
A5: CRA_4 port map (A(19 downto 16), B(19 downto 16), car4, S(19 downto 16), car5);
A6: CRA_4 port map (A(23 downto 20), B(23 downto 20), car5, S(23 downto 20), car6);
A7: CRA_4 port map (A(27 downto 24), B(27 downto 24), car6, S(27 downto 24), car7);
A8: CRA_4 port map (A(31 downto 28), B(31 downto 28), car7, S(31 downto 28), Cout);

end Behavioral;
```

D. ALU

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: ALU_32 - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity ALU_32 is
	Port ( ALU_OP : in STD_LOGIC_VECTOR(2 downto 0) := "010";
	ALUSrc : in STD_LOGIC := '0';
	INPUT_A : in STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
	INPUT_B : in STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
	INPUT_IMM : in STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
	OUTPUT : out STD_LOGIC_VECTOR(31 downto 0);
	ZERO : out STD_LOGIC);
end ALU_32;
```

```
architecture Behavioral of ALU_32 is

component CRA_32
	--Port map for the 4 bit adder
	Port ( A : in STD_LOGIC_VECTOR(31 downto 0);
		B : in STD_LOGIC_VECTOR(31 downto 0);
		Cin : in STD_LOGIC ;
		S : out STD_LOGIC_VECTOR(31 downto 0);
		Cout : out STD_LOGIC);
end component;

	signal Cin,Cout : STD_LOGIC;
	signal ADD_OUT , SUB_OUT , ORER,NANDER , ANDER , NORER , INPUT_INT
, SLT_OUT: STD_LOGIC_VECTOR(31 downto 0);
	signal SUB_IN , ZERO_VAL: STD_LOGIC_VECTOR(31 downto 0);
	signal ZERO_OUT: STD_LOGIC;


begin
	Cin <= '0';
	with ALUSrc select
	SUB_IN <= not(INPUT_B) when '0',
		not(INPUT_IMM) when others;
	with ALUSrc select
	INPUT_INT <= INPUT_B when '0',
		INPUT_IMM when others;
	A1: CRA_32 port map (A => INPUT_A(31 downto 0), B => INPUT_INT(31 downto
0), Cin => Cin, S => ADD_OUT(31 downto 0), Cout => Cout);

	S1: CRA_32 port map (A => INPUT_A(31 downto 0), B => SUB_IN, Cin => Cin, S
=> SUB_OUT(31 downto 0), Cout => Cout);
	ZERO_VAL <= "00000000000000000000000000000000";
	ORER <= INPUT_A or INPUT_INT;
	NANDER <= not(INPUT_A) and INPUT_INT;
	NORER <= INPUT_A nor INPUT_INT;
	ANDER <= INPUT_A and INPUT_INT;
	SLT_OUT <= "0000000000000000000000000000000" & SUB_OUT(31);



	with ALU_OP select
	OUTPUT <=   ADD_OUT when "010",
		SUB_OUT when "011",
		ORER when "101",
		ANDER when "100",
		NORER when "110",
		SLT_OUT when "111",
		"00000000000000000000000000000000" when others;


	with NANDER select
	ZERO <= '1' when "00000000000000000000000000000000",
```

```
                    '0' when others;

end Behavioral;
```

Test Bench:

```
--------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: ALU_32_tb - Behavioral
-- Programmer: Andrew Woltman
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity ALU_32_tb is
end;

architecture bench of ALU_32_tb is

  component ALU_32
          Port ( ALU_OP : in STD_LOGIC_VECTOR(2 downto 0);
                    ALUSrc : in STD_LOGIC;
                    INPUT_A : in STD_LOGIC_VECTOR(31 downto 0);
                    INPUT_B : in STD_LOGIC_VECTOR(31 downto 0);
                    INPUT_IMM : in STD_LOGIC_VECTOR(31 downto 0);
                    OUTPUT : out STD_LOGIC_VECTOR(31 downto 0);
                    ZERO : out STD_LOGIC);
  end component;

  signal ALU_OP: STD_LOGIC_VECTOR(2 downto 0);
  signal ALUSrc: STD_LOGIC;
  signal INPUT_A: STD_LOGIC_VECTOR(31 downto 0);
  signal INPUT_B: STD_LOGIC_VECTOR(31 downto 0);
  signal INPUT_IMM: STD_LOGIC_VECTOR(31 downto 0);
  signal OUTPUT: STD_LOGIC_VECTOR(31 downto 0);
  signal ZERO: STD_LOGIC;

begin

  uut: ALU_32 port map ( ALU_OP      => ALU_OP,
                    ALUSrc          => ALUSrc,
                    INPUT_A         => INPUT_A,
                    INPUT_B         => INPUT_B,
                    INPUT_IMM  => INPUT_IMM,
                    OUTPUT          => OUTPUT,
                    ZERO            => ZERO );
```

```
stimulus: process
begin

        INPUT_A <= "10000110001100011000110000010001";
        INPUT_B <= "00000000000000000000000000000000";
        INPUT_IMM <= "01111001110011100111001111101110";
        ALU_OP <= "010";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;
        ALU_OP <= "011";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;
        ALU_OP <= "101";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;
        ALU_OP <= "100";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;
        ALU_OP <= "110";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;
        ALU_OP <= "111";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;
        ALU_OP <= "000";
        ALUSrc <= '0';
        wait for 50ns;
        ALUSrc <= '1';
        wait for 50ns;

        wait;
 end process;

end;
```

E. Registers Block

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: REGISTERS - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity REGISTERS is
	Port ( READREG1 : in STD_LOGIC_VECTOR(4 downto 0) := "00000";
	READREG2 : in STD_LOGIC_VECTOR(4 downto 0) := "00000";
	READREG3 : in STD_LOGIC_VECTOR(4 downto 0) := "00000";
	WRITEDATA_MEM : in STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
	WRITEDATA_ALU : in STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
	REGWRITE : in STD_LOGIC := '0';
	RegDst : in STD_LOGIC := '0';
	MemToReg : in STD_LOGIC:= '0';
	READDATA1 : out STD_LOGIC_VECTOR(31 downto 0);
	READDATA2 : out STD_LOGIC_VECTOR(31 downto 0));
end REGISTERS;

architecture Behavioral of REGISTERS is

type registersArray is array(0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
signal readarray:registersArray :=
(x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000
",x"00000000",

x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000"
,x"00000000",

x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000"
,x"00000000",

x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000",x"00000000"
,x"00000000");
begin
	READDATA1 <= readarray(to_integer(unsigned(READREG1)));
	READDATA2 <= readarray(to_integer(unsigned(READREG2)));

	writeProcess:
process(RegDst,REGWRITE,MemToReg,WRITEDATA_ALU,WRITEDATA_MEM,READR
EG2,READREG3) is
	begin
	if (REGWRITE = '1') then
		 if (RegDst = '1') then
		if (MemToReg = '0') then
		readarray(to_integer(unsigned(READREG3))) <= WRITEDATA_ALU;
		else
		readarray(to_integer(unsigned(READREG3))) <= WRITEDATA_MEM;
		end if;
		else
```

```
                    if (MemToReg = '0') then
                    readarray(to_integer(unsigned(READREG2))) <= WRITEDATA_ALU;
                    else
                    readarray(to_integer(unsigned(READREG2))) <= WRITEDATA_MEM;
                    end if;
                     end if;
          end if;
   end process;

end Behavioral;
```

Test Bench:

```
--------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: REGISTERS_tb - Behavioral
-- Programmer: Andrew Woltman
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity REGISTERS_tb is
end;

architecture bench of REGISTERS_tb is

 component REGISTERS
          Port ( READREG1 : in STD_LOGIC_VECTOR(4 downto 0);
          READREG2 : in STD_LOGIC_VECTOR(4 downto 0);
          READREG3 : in STD_LOGIC_VECTOR(4 downto 0);
          WRITEDATA_ALU : in STD_LOGIC_VECTOR(31 downto 0);
          WRITEDATA_MEM : in STD_LOGIC_VECTOR(31 downto 0);
          REGWRITE : in STD_LOGIC;
          RegDst : in STD_LOGIC;
          MemToReg : in STD_LOGIC;
          READDATA1 : out STD_LOGIC_VECTOR(31 downto 0);
          READDATA2 : out STD_LOGIC_VECTOR(31 downto 0));
 end component;

 signal READREG1: STD_LOGIC_VECTOR(4 downto 0);
 signal READREG2: STD_LOGIC_VECTOR(4 downto 0);
 signal READREG3: STD_LOGIC_VECTOR(4 downto 0);
 signal WRITEDATA_ALU: STD_LOGIC_VECTOR(31 downto 0);
 signal WRITEDATA_MEM: STD_LOGIC_VECTOR(31 downto 0);
 signal REGWRITE: STD_LOGIC;
 signal RegDst : STD_LOGIC;
 signal MemToReg: STD_LOGIC;
 signal READDATA1: STD_LOGIC_VECTOR(31 downto 0);
 signal READDATA2: STD_LOGIC_VECTOR(31 downto 0);
```

```
begin

 uut: REGISTERS port map ( READREG1  => READREG1,
                           READREG2  => READREG2,
                           READREG3  => READREG3,
                           WRITEDATA_ALU => WRITEDATA_ALU,
                           WRITEDATA_MEM => WRITEDATA_MEM,
                           REGWRITE  => REGWRITE,
                           RegDst => RegDst,
                           MemToReg   => MemToReg,
                           READDATA1 => READDATA1,
                           READDATA2 => READDATA2);

 stimulus: process
 begin

          READREG1 <= "10001";
          READREG2 <= "10000";
          READREG3 <= "00000";
          WRITEDATA_ALU <= "01111001110011100111001111101110";
          WRITEDATA_MEM <= "11111111111111111111111111111111";
          RegDst <= '0';
          REGWRITE <= '0';
          MemToReg <= '0';
          wait for 50ns;
          RegDst <= '0';
          REGWRITE <= '0';
          MemToReg <= '1';
          READREG3 <= "00001";
          wait for 50ns;
          RegDst <= '1';
          REGWRITE <= '0';
          MemToReg <= '0';
          READREG3 <= "00010";
          wait for 50ns;
          RegDst <= '1';
          REGWRITE <= '0';
          MemToReg <= '1';
          READREG3 <= "00011";
          wait for 50ns;
          RegDst <= '0';
          REGWRITE <= '1';
          MemToReg <= '0';
          wait for 50ns;
          RegDst <= '0';
          REGWRITE <= '1';
          MemToReg <= '1';
          READREG3 <= "00001";
          wait for 50ns;
          RegDst <= '1';
          REGWRITE <= '1';
          MemToReg <= '0';
```

```
        READREG3 <= "00010";
        wait for 50ns;
        RegDst <= '1';
        REGWRITE <= '1';
        MemToReg <= '1';
        READREG3 <= "00011";
        wait for 50ns;
  end process;
end;
```

F. Instruction Register

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: INSTRUCTION_REGISTERS - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity INSTRUCTION_REGISTERS is
        Port ( INSTRUCTION : in STD_LOGIC_VECTOR(31 downto 0) := x"00000000";
        IRWrite : in STD_LOGIC := '0';
        REG1 : out STD_LOGIC_VECTOR(4 downto 0);
        REG2 : out STD_LOGIC_VECTOR(4 downto 0);
        REG3 : out STD_LOGIC_VECTOR(4 downto 0);
        IMM : out STD_LOGIC_VECTOR(31 downto 0);
        JMP_ADDRESS : out STD_LOGIC_VECTOR(25 downto 0);
        INSTRUCTION_TYP : out STD_LOGIC_VECTOR(3 downto 0));
end INSTRUCTION_REGISTERS;

architecture Behavioral of INSTRUCTION_REGISTERS is

signal OP_CODE_INT : STD_LOGIC_VECTOR(5 downto 0);
signal FUNC_CODE_INT : STD_LOGIC_VECTOR(5 downto 0);

signal IMM_INT : STD_LOGIC_VECTOR(31 downto 0);
begin
-- This updates the Instruction when a new instruction is given
        writeProcess: process(IRWrite) is
        begin
                if (IRWrite = '1') then
                REG1 <= INSTRUCTION(25 downto 21);
                REG2 <= INSTRUCTION(20 downto 16);
                REG3 <= INSTRUCTION(15 downto 11);
                IMM <=  std_logic_vector(resize(signed(INSTRUCTION(15 downto 0)),
32));
```

```
                JMP_ADDRESS <= INSTRUCTION(25 downto 0);
                OP_CODE_INT <= INSTRUCTION(31 downto 26);
                FUNC_CODE_INT <= INSTRUCTION(5 downto 0);
                end if;
        end process;

        process (OP_CODE_INT, FUNC_CODE_INT) is
        begin
        case OP_CODE_INT is
                when "000010" => -- J
                        INSTRUCTION_TYP <= "0000";
                when "000100" => -- BEQ
                        INSTRUCTION_TYP <= "0001";
                when "001000" => -- ADDI
                        INSTRUCTION_TYP <= "1010";
                when "001100" => -- ANDI
                        INSTRUCTION_TYP <= "1100";
                when "001101" => -- ORI
                        INSTRUCTION_TYP <= "1101";
                when "101011" => -- SW
                        INSTRUCTION_TYP <= "1000";
                when "100011" => -- LW
                        INSTRUCTION_TYP <= "1001";
                when others =>
                        INSTRUCTION_TYP <= "1111";
                end case;

                case FUNC_CODE_INT is
                when "100000" => -- ADD
                        INSTRUCTION_TYP <= "0010";
                when "100010" => -- SUB
                        INSTRUCTION_TYP <= "0011";
                when "100100" => -- AND
                        INSTRUCTION_TYP <= "0100";
                when "100101" => -- OR
                        INSTRUCTION_TYP <= "0101";
                when "100111" => -- NOR
                        INSTRUCTION_TYP <= "0110";
                when "101010" => -- SLT
                        INSTRUCTION_TYP <= "0111";
                when others =>
                        null;
                end case;
        end process;

end Behavioral;
```

Test Bench:

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

```
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: INSTRUCTION_REGISTERS_tb - Behavioral
-- Programmer: Andrew Woltman
----------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity INSTRUCTION_REGISTERS_tb is
end;

architecture bench of INSTRUCTION_REGISTERS_tb is

  component INSTRUCTION_REGISTERS
        Port ( INSTRUCTION : in STD_LOGIC_VECTOR(31 downto 0);
        IRWrite : in STD_LOGIC := '0';
        REG1 : out STD_LOGIC_VECTOR(4 downto 0);
        REG2 : out STD_LOGIC_VECTOR(4 downto 0);
        REG3 : out STD_LOGIC_VECTOR(4 downto 0);
        IMM : out STD_LOGIC_VECTOR(31 downto 0);
        JMP_ADDRESS : out STD_LOGIC_VECTOR(25 downto 0);
        INSTRUCTION_TYP : out STD_LOGIC_VECTOR(3 downto 0));
  end component;

  signal INSTRUCTION: STD_LOGIC_VECTOR(31 downto 0);
  signal IRWrite: STD_LOGIC;
  signal REG1: STD_LOGIC_VECTOR(4 downto 0);
  signal REG2: STD_LOGIC_VECTOR(4 downto 0);
  signal REG3: STD_LOGIC_VECTOR(4 downto 0);
  signal IMM: STD_LOGIC_VECTOR(31 downto 0);
  signal JMP_ADDRESS: STD_LOGIC_VECTOR(25 downto 0);
  signal INSTRUCTION_TYP : STD_LOGIC_VECTOR(3 downto 0);

begin

  uut: INSTRUCTION_REGISTERS port map ( INSTRUCTION => INSTRUCTION,
                                IRWrite         => IRWrite,
                                INSTRUCTION_TYP => INSTRUCTION_TYP,
                                REG1            => REG1,
                                REG2            => REG2,
                                REG3            => REG3,
                                IMM             => IMM,
                                JMP_ADDRESS => JMP_ADDRESS);

  stimulus: process
  begin
        IRWrite <= '1';
        INSTRUCTION <= "10000110001100011000110000100001";
        wait for 50 ns;
        IRWrite <= '0';
        INSTRUCTION <= "11111111111111111111111111111111";
        wait for 50 ns;
        IRWrite <= '1';
```

```
        wait for 50 ns;

        wait;
  end process;


end;
```

G. Memory Controller

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: ControlUnit_tb - Behavioral
-- Programmer: Clive Gomes
----------------------------------------------------------------------------
-- Module Name: MemoryController - Behavioral
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MemoryController is
  Port (IorD : in STD_LOGIC; -- Control Signal for Instruction Vs Data
        MemRead : in STD_LOGIC; -- Control Signal to Read from Memory
        MemWrite : in STD_LOGIC; -- Control Signal to Write to Memory
        MemReadOut : out STD_LOGIC; -- Output to Forward MemRead to Memory
        MemWriteOut : out STD_LOGIC; -- Output to Forward MemWrite to Memory
        MemIn : in STD_LOGIC_VECTOR (31 downto 0); -- Data Input From Memory
        MemOut : out STD_LOGIC_VECTOR (31 downto 0); -- Data Output to Memory
        WData : in STD_LOGIC_VECTOR (31 downto 0); -- Data to Write to Memory from
CPU
        RData : out STD_LOGIC_VECTOR (31 downto 0); -- Data Read Sent to CPU
        IAddress : in STD_LOGIC_VECTOR (31 downto 0); -- Instruction Address
        DAddress : in STD_LOGIC_VECTOR (31 downto 0); -- Data Address
        AddressOut : out STD_LOGIC_VECTOR (31 downto 0)); -- Output to Forward
Address to Memory
end MemoryController;

architecture Behavioral of MemoryController is
begin

-- Forward Appropriate Address to Memory
process (IAddress, DAddress, IorD)
begin
  if IorD = '1' then -- Data?
        AddressOut <= DAddress; -- Data Address
  else -- Instruction?
        AddressOut <= IAddress; -- Instruction Address
  end if;
end process;
```

```
-- Forward MemRead & MemWrit to Memory
process (MemRead, MemWrite, IorD)
begin
  MemReadOut <= MemRead; -- Forward MemRead
  -- Forward MemWrite Only If Writing to Data Memory
  if IorD = '1' then MemWriteOut <= MemWrite;
  else MemWriteOut <= '0';
  end if;
end process;

RData <= MemIn; -- Forward Data Read from Memory to CPU
MemOut <= WData; -- Forward Data to Write to Memory from CPU

end Behavioral;
```

Test Bench:

```
--------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: MemoryController_tb - Behavioral
-- Programmer: Clive Gomes
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity MemoryController_tb is
end;

architecture bench of MemoryController_tb is

  component MemoryController
          Port ( IorD : in STD_LOGIC; -- Control Signal for Instruction Vs Data
          MemRead : in STD_LOGIC; -- Control Signal to Read from Memory
          MemWrite : in STD_LOGIC; -- Control Signal to Write to Memory
          MemReadOut : out STD_LOGIC; -- Output to Forward MemRead to Memory
          MemWriteOut : out STD_LOGIC; -- Output to Forward MemWrite to Memory
          MemIn : in STD_LOGIC_VECTOR (31 downto 0); -- Data Input From Memory
          MemOut : out STD_LOGIC_VECTOR (31 downto 0); -- Data Output to Memory
          WData : in STD_LOGIC_VECTOR (31 downto 0); -- Data to Write to Memory from
CPU
          RData : out STD_LOGIC_VECTOR (31 downto 0); -- Data Read Sent to CPU
          IAddress : in STD_LOGIC_VECTOR (31 downto 0); -- Instruction Address
          DAddress : in STD_LOGIC_VECTOR (31 downto 0); -- Data Address
          AddressOut : out STD_LOGIC_VECTOR (31 downto 0)); -- Output to Forward
Address to Memory
  end component;

  signal IorD: STD_LOGIC; -- Control Signal for Instruction Vs Data
```

```
  signal MemRead: STD_LOGIC; -- Control Signal to Read from Memory
  signal MemWrite: STD_LOGIC; -- Control Signal to Write to Memory
  signal MemReadOut: STD_LOGIC; -- Output to Forward MemRead to Memory
  signal MemWriteOut: STD_LOGIC; -- Output to Forward MemWrite to Memory
  signal MemIn: STD_LOGIC_VECTOR (31 downto 0); -- Data Input From Memory
  signal MemOut: STD_LOGIC_VECTOR (31 downto 0); -- Data Output to Memory
  signal WData: STD_LOGIC_VECTOR (31 downto 0); -- Data to Write to Memory from
CPU
  signal RData: STD_LOGIC_VECTOR (31 downto 0); -- Data Read Sent to CPU
  signal IAddress: STD_LOGIC_VECTOR (31 downto 0); -- Instruction Address
  signal DAddress: STD_LOGIC_VECTOR (31 downto 0); -- Data Address
  signal AddressOut: STD_LOGIC_VECTOR (31 downto 0); -- Output to Forward Address
to Memory

begin

  -- Mapping Ports to Signals
  uut: MemoryController port map ( IorD         => IorD,
                                   MemRead      => MemRead,
                                   MemWrite     => MemWrite,
                                   MemReadOut   => MemReadOut,
                                   MemWriteOut  => MemWriteOut,
                                   MemIn        => MemIn,
                                   MemOut       => MemOut,
                                   WData        => WData,
                                   RData        => RData,
                                   IAddress     => IAddress,
                                   DAddress     => DAddress,
                                   AddressOut   => AddressOut );

  -- Start of Test Bench
  stimulus: process
  begin

        -- Writing to Instruction Memory

        IAddress <= x"00000000"; -- Initial Address
        MemRead <= '0'; -- MemRead Disabled
        MemWrite <= '0'; -- MemWrite Disabled
        IorD <= '0'; -- Instruction

        wait for 10 ns; -- Wait

        WData <= x"12345678"; -- Data Being Written
        MemWrite <= '1'; -- Write to Memory
        IAddress <= x"00000004"; -- Memory Location

        wait for 10 ns; -- Wait
        MemWrite <= '0'; -- MemWrite Disabled

        wait; -- Wait Indefinitely
  end process;
  -- End of Test Bench
```

```
end;
```

H. Cache

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: Memory - Behavioral
-- Programmer: Clive Gomes
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Memory is
  Port ( MemIn : in STD_LOGIC_VECTOR (31 downto 0); -- Data Input from Memory
         MemOut : out STD_LOGIC_VECTOR (31 downto 0); -- Data Output to CPU
         MemRead : in STD_LOGIC; -- Control Signal to Read from Memory
         MemWrite : in STD_LOGIC; -- Control Signal to Write to Memory
         Address : in STD_LOGIC_VECTOR (31 downto 0)); -- Memory Address
end Memory;

architecture Behavioral of Memory is
         type MEMORY is array (1023 downto 0) of STD_LOGIC_VECTOR(31 downto 0);  --
Array of 1024 x 32 bit memory locations

         -- Initializing Instructions in Memory
         signal MEMORY1024x32 : MEMORY := (others => x"00000000");
begin

-- Read from Memory
process (MemRead)
begin
         if MemRead = '1' then -- Read Only if Enabled
         MemOut <= MEMORY1024x32(to_integer(unsigned(Address(31 downto 2)) mod
1024)); -- Read from Appropriate Block of Memory
         end if;
end process;

-- Write to Memory
process (MemWrite)
begin
         if MemWrite = '1' then -- Write Only if Enabled
         MEMORY1024x32(to_integer(unsigned(Address(31 downto 2))) mod 1024) <=
MemIn; -- Write to Appropriate Block of Memory
         end if;
end process;
```

```
end Behavioral;
```

Test Bench:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: Memory_tb - Behavioral
-- Programmer: Clive Gomes
----------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity Memory_tb is
end;

architecture bench of Memory_tb is

  component Memory
        Port ( MemIn : in STD_LOGIC_VECTOR (31 downto 0); -- Data Input from
Memory
        MemOut : out STD_LOGIC_VECTOR (31 downto 0); -- Data Output to CPU
        MemRead : in STD_LOGIC; -- Control Signal to Read from Memory
        MemWrite : in STD_LOGIC; -- Control Signal to Write to Memory
        Address : in STD_LOGIC_VECTOR (31 downto 0)); -- Memory Address
  end component;

  signal MemIn: STD_LOGIC_VECTOR (31 downto 0); -- Data Input from Memory
  signal MemOut: STD_LOGIC_VECTOR (31 downto 0); -- Data Output to CPU
  signal MemRead: STD_LOGIC; -- Control Signal to Read from Memory
  signal MemWrite: STD_LOGIC; -- Control Signal to Write to Memory
  signal Address: STD_LOGIC_VECTOR (31 downto 0); -- Memory Address

begin

  -- Mapping Ports to Signals
  uut: Memory port map ( MemIn          => MemIn,
                         MemOut   => MemOut,
                         MemRead  => MemRead,
                         MemWrite => MemWrite,
                         Address  => Address );

  -- Start of Test Bench
  stimulus: process
  begin

        MemIn <= x"12345678"; -- Data to Write
        Address <= x"00000004"; -- Memory Address
        MemRead <= '0'; -- MemRead Disabled
        MemWrite <= '0'; -- MemWrite Disabled
```

```
        wait for 10 ns; -- Wait

        MemWrite <= '1'; -- Write to Memory

        wait for 10 ns; -- Wait

        MemWrite <= '0'; -- MemWrite Disabled
        MemRead <= '1'; -- Read from Memory

        wait for 10 ns; -- Wait

        MemRead <= '0'; -- MemRead Disabled

        wait; -- Wait Indefinitely
  end process;
  -- End of Test Bench

end;
```

I. PCU

Source Code:

```
----------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: PCUnit - Behavioral
-- Programmer: Clive Gomes
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity PCUnit is
   Port ( OPC : in STD_LOGIC_VECTOR (31 downto 0); -- Current PC Inputted In
          JAddress : in STD_LOGIC_VECTOR (25 downto 0); -- 26-Bit Jump Address from
Instruction (Bits 25:0)
          BOffset : in STD_LOGIC_VECTOR (31 downto 0); -- 32-Bit Sign-Extended Branch
Offset
          PCSrc : in STD_LOGIC_VECTOR (1 downto 0); -- Control Signal to Decide How To
Compute NPC
          PCWrite : in STD_LOGIC; -- Control Signal to Update PC to NPC
          Zero : in STD_LOGIC; -- Zero Result From ALU
          NPC : out STD_LOGIC_VECTOR (31 downto 0)); -- NPC Output
end PCUnit;

architecture Behavioral of PCUnit is

          signal PC : STD_LOGIC_VECTOR (31 downto 0) := x"00000000"; -- Signal for NPC
          signal BranchOffset : STD_LOGIC_VECTOR (31 downto 0) := x"00000000"; --
Signal for Left-Shifted-2 Branch Offset
```

```
	signal P4OUT : STD_LOGIC_VECTOR (31 downto 0) := x"00000000"; -- Output
Signal for PC+4 Adder
	signal BRANCHOUT : STD_LOGIC_VECTOR (31 downto 0) := x"00000000"; --
Output Signal for Branch Adder

	-- Full Adder Component
	component CRA_32
	Port ( A : in STD_LOGIC_VECTOR(31 downto 0);
		B : in STD_LOGIC_VECTOR(31 downto 0);
		Cin : in STD_LOGIC;
		S : out STD_LOGIC_VECTOR(31 downto 0);
		Cout : out STD_LOGIC);
	end component;

begin
	-- PC + 4 Adder
	PLUS4ADDER: CRA_32 port map (
	A => OPC(31 downto 0), -- Current PC
	B => x"00000004",	-- + 4
	Cin => '0',
	S => P4OUT(31 downto 0));
	-- Branch Adder
	BRANCHADDER: CRA_32 port map (
	A => OPC(31 downto 0),		-- Current PC
	B => BranchOffset(31 downto 0),-- + Branch Offset
	Cin => '0',
	S => BRANCHOUT(31 downto 0));

  BranchOffset <= BOffset(29 downto 0) & "00"; -- Left-Shift-2 Branch Offset

-- Compute NPC
process (OPC, JAddress, PCSrc, Zero, P4OUT, BRANCHOUT) is
begin
  if PCSrc = "01" then -- j instruction
	PC <= OPC(31 downto 28) & JAddress(25 downto 0) & "00"; -- Compute Jump
Address
  elsif PCSrc = "10" then -- beq instruction
	-- Branch Only if Zero Result of ALU = 1
	if Zero = '1' then
	PC <= BRANCHOUT; -- Branch Address
	else
	PC <= P4OUT; -- PC + 4
	end if;
  else -- For Any Other Instruction
	PC <= P4OUT; --PC + 4
  end if;
end process;

-- Update PC to NPC
process (PCWrite)
  variable PCHold : STD_LOGIC_VECTOR (31 downto 0) := x"00000000"; -- Buffer for NPC
begin
  -- Output NPC Only if PCWrite Asserted
```

```
  if PCWrite = '1' then
          PCHold := PC;
  end if;
  NPC <= PCHold;
end process;

end Behavioral;
```

Test Bench:

```
--------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: PCUnit_tb - Behavioral
-- Programmer: Clive Gomes
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity PCUnit_tb is
end;

architecture bench of PCUnit_tb is

  component PCUnit
          Port ( OPC : in STD_LOGIC_VECTOR (31 downto 0); -- Current PC Inputted In
          JAddress : in STD_LOGIC_VECTOR (25 downto 0); -- 26-Bit Jump Address from
Instruction (Bits 25:0)
          BOffset : in STD_LOGIC_VECTOR (31 downto 0); -- 32-Bit Sign-Extended Branch
Offset
          PCSrc : in STD_LOGIC_VECTOR (1 downto 0); -- Control Signal to Decide How To
Compute NPC
          PCWrite : in STD_LOGIC; -- Control Signal to Update PC to NPC
          Zero : in STD_LOGIC; -- Zero Result From ALU
          NPC : out STD_LOGIC_VECTOR (31 downto 0)); -- NPC Output
  end component;

  signal OPC: STD_LOGIC_VECTOR (31 downto 0); -- Current PC Inputted In
  signal JAddress: STD_LOGIC_VECTOR (25 downto 0); -- 26-Bit Jump Address from
Instruction (Bits 25:0)
  signal BOffset: STD_LOGIC_VECTOR (31 downto 0); -- 32-Bit Sign-Extended Branch
Offset
  signal PCSrc: STD_LOGIC_VECTOR (1 downto 0); -- Control Signal to Decide How To
Compute NPC
  signal PCWrite: STD_LOGIC; -- Control Signal to Update PC to NPC
  signal Zero: STD_LOGIC; -- Zero Result From ALU
  signal NPC: STD_LOGIC_VECTOR (31 downto 0); -- NPC Output

begin
```

```
-- Mapping Ports to Signals
uut: PCUnit port map ( OPC  => OPC,
                       JAddress => JAddress,
                       BOffset  => BOffset,
                       PCSrc  => PCSrc,
                       PCWrite  => PCWrite,
                       Zero     => Zero,
                       NPC     => NPC );

-- Start of Test Bench
stimulus: process
begin

        OPC <= x"00000000"; -- Initial PC
        JAddress <= "10000000000000000000000001"; -- Jump to 0x08000004
        BOffset <= x"0000003f"; -- Branch to 0x000000fc
        PCSrc <= "00"; -- NPC = PC + 4
        PCWrite <= '0'; -- PCWrite Disabled
        Zero <= '0'; -- Branch Condition = False

        wait for 10 ns; -- Delay

        PCSrc <= "00"; -- Perform Regular PC + 4
        PCWrite <= '1'; -- Update NPC

        wait for 10 ns; -- Delay
        PCWrite <= '0'; -- PCWrite Disabled
        PCSrc <= "00"; -- Perform Regular PC + 4
        wait for 10 ns; -- Delay

        PCSrc <= "01"; -- Perform Jump
        PCWrite <= '1'; -- Update NPC

        wait for 10 ns; -- Delay
        PCWrite <= '0'; -- PCWrite Disabled
        PCSrc <= "00"; -- Perform Regular PC + 4
        wait for 10 ns; -- Delay

        PCSrc <= "10"; -- Perform Branch
        Zero <= '0'; -- Branch Not Taken
        PCWrite <= '1'; -- Update NPC

        wait for 10 ns; -- Delay
        PCWrite <= '0'; -- PCWrite Disabled
        PCSrc <= "00"; -- Perform Regular PC + 4
        wait for 10 ns; -- Delay

        PCSrc <= "10"; -- Perform Branch
        Zero <= '1'; -- Branch Taken
        PCWrite <= '1'; -- Update NPC

        wait; -- Wait Indefinitely
end process;
```

```
 -- End of Test Bench

end;
```

J. MCU

Source Code:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: ControlUnit - Behavioral
-- Programmer: Clive Gomes
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ControlUnit is
	Port ( clk : in STD_LOGIC; -- Clock Input
	instruction : in STD_LOGIC_VECTOR(3 downto 0); -- Decoded Instruction from IR
	stage : out STD_LOGIC_VECTOR(3 downto 0); -- Outputs Current Stage (Only For
Testing)
	IorD : out STD_LOGIC; -- Control Signal for Instruction Vs Data
	MemRead : out STD_LOGIC; -- Control Signal to Read from Memory
	MemWrite : out STD_LOGIC; -- Control Signal to Write to Memory
	IRWrite : out STD_LOGIC; -- Control Signal for Updating Instruction in IR
	RegDst : out STD_LOGIC; -- MUX Select Signal to Choose Destination Register
Index
	MemToReg : out STD_LOGIC; -- MUX Select Signal to Choose ALU Output Vs
Data from Memory for Register Write
	RegWrite : out STD_LOGIC; -- Control Signal for Writing to Register
	ALUOp : out STD_LOGIC_VECTOR(2 downto 0); -- Control Signal to Select ALU
Operation
	ALUSrc : out STD_LOGIC; -- MUX Select Signal to Choose Register Contents Vs
Imm Value for ALU Input
	PCSrc : out STD_LOGIC_VECTOR(1 downto 0); -- Control Signal for PC Update
Operation
	PCWrite : out STD_LOGIC); -- Control Signal to Update PC to NPC
end ControlUnit;

architecture Behavioral of ControlUnit is
	signal instr : STD_LOGIC_VECTOR (3 downto 0) := "1111"; -- Signal to Hold
Decoded Instruction (Default -- Invalid Instruction)
begin
	instr <= instruction; -- Get Decoded Instruction from IR

process(clk) is
	variable stageCount : integer := 0; -- Stage Counter
	variable stageEnd : integer  := 4; -- Stage Counter Limit
begin
```

```
-- Perform Operation On Every Rising Edge of Clock
        if (rising_edge(clk)) then

        stageCount := stageCount + 1; -- Increment Stage Count

        -- First 2 Stages Common to All Instructions
        case stageCount is
        when 1 => -- IF: Fetch Instruction from Memory
        IorD <= '0'; -- Instruction
        MemRead <= '1'; -- MemRead Asserted
        MemWrite <= '0';
        IRWrite <= '0';
        RegDst <= '0';
        MemToReg <= '0';
        RegWrite <= '0';
        ALUOp <= "000";
        ALUSrc <= '0';
        PCSrc <= "00";
        PCWrite <= '0';
        when 2 => -- ID: Decode Instruction
        IorD <= '0';
        MemRead <= '0';
        MemWrite <= '0';
        IRWrite <= '1'; -- Send Instruction to IR for Decoding
        RegDst <= '0';
        MemToReg <= '0';
        RegWrite <= '0';
        ALUOp <= "000";
        ALUSrc <= '0';
        PCSrc <= "00";
        PCWrite <= '0';
        when others=> -- Others Clause
        null;
        end case;

        -- Remaining Stages Depending on Decoded Instruction
        case instr is
        when "0000" => -- j instruction
        stageEnd := 4; -- Jump Takes 4 Stages for Execution
        case stageCount is
        when 3 =>null; -- PC: Compute NPC
        IorD <= '0';
        MemRead <= '0';
        MemWrite <= '0';
        IRWrite <= '0';
        RegDst <= '0';
        MemToReg <= '0';
        RegWrite <= '0';
        ALUSrc <= '0';
        ALUOp <= "000";
        PCSrc <= "01"; -- PCU Computes Jump Address
        PCWrite <= '0';
```

```
when 4 =>null; -- PCW: Update PC to NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUSrc <= '0';
ALUOp <= "000";
PCSrc <= "01";
PCWrite <= '1'; -- Write NPC to Output
when others=> -- Others Clause
null;
end case;

when "0001" => -- beq instruction
stageEnd := 6; -- Branch On Equal Takes 6 Stages
case stageCount is
when 3 => -- REG: Wait for Registers Block to Output Register Contents
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "000";
ALUSrc <= '0';
PCSrc <= "00";
PCWrite <= '0';
when 4 => -- ALU: ALU Performs Operation
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "011"; -- Subtract
ALUSrc <= '0'; -- Use Register Content as ALU Input #2
PCSrc <= "00";
PCWrite <= '0';
when 5 =>  -- PC: Compute NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "011";
ALUSrc <= '0';
```

```
            PCSrc <= "10"; -- PC Computes Branch Address
            PCWrite <= '0';
            when 6 =>null; -- PCW: Update PC to NPC
            IorD <= '0';
            MemRead <= '0';
            MemWrite <= '0';
            IRWrite <= '0';
            RegDst <= '0';
            MemToReg <= '0';
            RegWrite <= '0';
            ALUSrc <= '0';
            ALUOp <= "011";
            PCSrc <= "10";
            PCWrite <= '1'; -- Write NPC to Output
            when others=> -- Others Clause
            null;
            end case;

            when "0010" | "0011" | "0100" | "0101" | "0110" | "0111" | "1010" | "1100" |
"1101" => -- add/sub/and/or/nor/slt/addi/andi/ori instruction
            stageEnd := 7; -- R-Type & I-Type Instructions Take 7 stages
            case stageCount is
            when 3 => -- REG: Wait for Registers Block to Output Register Contents
            IorD <= '0';
            MemRead <= '0';
            MemWrite <= '0';
            IRWrite <= '0';
            RegDst <= not instruction(3); -- Set RegDst Based On Instruction
            MemToReg <= '0';
            RegWrite <= '0';
            ALUOp <= "000";
            ALUSrc <= '0';
            PCSrc <= "00";
            PCWrite <= '0';
            when 4 => -- ALU: ALU Performs Operation
            IorD <= '0';
            MemRead <= '0';
            MemWrite <= '0';
            IRWrite <= '0';
            RegDst <= not instruction(3);
            MemToReg <= '0';
            RegWrite <= '0';
            ALUOp <= instruction(2 downto 0); -- Select ALUOp Based on Decoded Instruction
            ALUSrc <= instruction(3); -- Select Imm Value Vs Register Contents as ALU Input
#2 Based On Decoded Instruction
            PCSrc <= "00";
            PCWrite <= '0';
            when 5 => -- WB: Write Data to Register
            IorD <= '0';
            MemRead <= '0';
            MemWrite <= '0';
            IRWrite <= '0';
            RegDst <= not instruction(3);
```

```
MemToReg <= '0'; -- Writing from ALU Output
RegWrite <= '1'; -- Enable Write to Register
ALUOp <= instruction(2 downto 0);
ALUSrc <= instruction(3);
PCSrc <= "00";
PCWrite <= '0';
when 6 =>  -- PC: Compute NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "000";
ALUSrc <= '0';
PCSrc <= "00"; -- PCU Computes PC + 4
PCWrite <= '0';
when 7 =>null; -- PCW: Update PC to NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUSrc <= '0';
ALUOp <= "000";
PCSrc <= "00";
PCWrite <= '1'; -- Write NPC to Output
when others=> -- Others Clause
null;
end case;

when "1000" => -- sw instruction
stageEnd := 7;
case stageCount is
when 3 => -- REG: Wait for Registers Block to Output Register Contents
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "000";
ALUSrc <= '0';
PCSrc <= "00";
PCWrite <= '0';
when 4 => -- ALU: ALU Performs Operation
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
```

```
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "010"; -- Add
ALUSrc <= '1'; -- Use Imm Value as ALU Input #2
PCSrc <= "00";
PCWrite <= '0';
when 5 => -- MEM: Memory Access
IorD <= '1';
MemRead <= '0';
MemWrite <= '1'; -- Enable Write to Memory
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "010";
ALUSrc <= '1';
PCSrc <= "00";
PCWrite <= '0';
when 6 => -- PC: Compute NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "000";
ALUSrc <= '0';
PCSrc <= "00"; -- PCU Computes PC + 4
PCWrite <= '0';
when 7 =>null; -- PCW: Update PC to NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUSrc <= '0';
ALUOp <= "000";
PCSrc <= "00";
PCWrite <= '1'; -- Write NPC to Output
when others=> -- Others Clause
null;
end case;

when "1001" => -- lw instruction
stageEnd := 8;
case stageCount is
when 3 => -- REG: Wait for Registers Block to Output Register Contents
IorD <= '0';
```

```
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0'; -- RegDst for lw instruction
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "000";
ALUSrc <= '0';
PCSrc <= "00";
PCWrite <= '0';
when 4 => -- ALU: ALU Performs Operation
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "010"; -- Add
ALUSrc <= '1'; -- Use Imm Value as ALU Input #2
PCSrc <= "00";
PCWrite <= '0';
when 5 => -- MEM: Memory Access
IorD <= '1';
MemRead <= '1'; -- Enable Memory Read
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUOp <= "010";
ALUSrc <= '1';
PCSrc <= "00";
PCWrite <= '0';
when 6 => -- WB: Write Data to Register
IorD <= '1';
MemRead <= '1';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '1'; -- Writing from Memory
RegWrite <= '1'; -- Enable Write to Register
ALUOp <= "010";
ALUSrc <= '1';
PCSrc <= "00";
PCWrite <= '0';
when 7 => -- PC: Compute NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
```

```
RegWrite <= '0';
ALUOp <= "000";
ALUSrc <= '0';
PCSrc <= "00"; -- PCU Computes PC + 4
PCWrite <= '0';
when 8 =>null; -- PCW: Update PC to NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUSrc <= '0';
ALUOp <= "000";
PCSrc <= "00";
PCWrite <= '1'; -- Write NPC to Output
when others=> -- Others Clause
null;
end case;

when others => -- Invalid Instruction
stageEnd := 4;
case stageCount is
when 3 =>null; -- PC: Compute NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUSrc <= '0';
ALUOp <= "000";
PCSrc <= "00"; -- PCU Computes PC + 4
PCWrite <= '0';
when 4 =>null; -- PCW: Update PC to NPC
IorD <= '0';
MemRead <= '0';
MemWrite <= '0';
IRWrite <= '0';
RegDst <= '0';
MemToReg <= '0';
RegWrite <= '0';
ALUSrc <= '0';
ALUOp <= "000";
PCSrc <= "00";
PCWrite <= '1'; -- Write NPC to Output
when others=> -- Others Clause
null;
end case;
end case;
```

```
        stage <= STD_LOGIC_VECTOR(to_signed(stageCount, 4)); -- Output Current
Stage

        -- Restart Stage Count at End of Instruction Execution
        if stageCount >= stageEnd then
        stageCount := 0;
        end if;

        end if;
end process;

end Behavioral;
```

Test Bench:

```
----------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: ControlUnit_tb - Behavioral
-- Programmer: Clive Gomes
----------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity ControlUnit_tb is
end;

architecture bench of ControlUnit_tb is

  component ControlUnit
        Port (  clk : in STD_LOGIC; -- Clock Input
        instruction : in STD_LOGIC_VECTOR(3 downto 0); -- Decoded Instruction from IR
        stage : out STD_LOGIC_VECTOR(3 downto 0); -- Outputs Current Stage (Only For
Testing)
        IorD : out STD_LOGIC; -- Control Signal for Instruction Vs Data
        MemRead : out STD_LOGIC; -- Control Signal to Read from Memory
        MemWrite : out STD_LOGIC; -- Control Signal to Write to Memory
        IRWrite : out STD_LOGIC; -- Control Signal for Updating Instruction in IR
        RegDst : out STD_LOGIC; -- MUX Select Signal to Choose Destination Register
Index
        MemToReg : out STD_LOGIC; -- MUX Select Signal to Choose ALU Output Vs
Data from Memory for Register Write
        RegWrite : out STD_LOGIC; -- Control Signal for Writing to Register
        ALUOp : out STD_LOGIC_VECTOR(2 downto 0); -- Control Signal to Select ALU
Operation
        ALUSrc : out STD_LOGIC; -- MUX Select Signal to Choose Register Contents Vs
Imm Value for ALU Input
        PCSrc : out STD_LOGIC_VECTOR(1 downto 0); -- Control Signal for PC Update
Operation
        PCWrite : out STD_LOGIC); -- Control Signal to Update PC to NPC
```

```
  end component;

  signal clk: STD_LOGIC;  -- Clock Input
  signal instruction: STD_LOGIC_VECTOR(3 downto 0); -- Decoded Instruction from IR
  signal stage: STD_LOGIC_VECTOR(3 downto 0); -- Outputs Current Stage (Only For
Testing)
  signal IorD: STD_LOGIC; -- Control Signal for Instruction Vs Data
  signal MemRead: STD_LOGIC; -- Control Signal to Read from Memory
  signal MemWrite: STD_LOGIC; -- Control Signal to Write to Memory
  signal IRWrite: STD_LOGIC; -- Control Signal for Updating Instruction in IR
  signal RegDst: STD_LOGIC; -- MUX Select Signal to Choose Destination Register Index
  signal MemToReg: STD_LOGIC; -- Control Signal to Select ALU Operation
  signal RegWrite: STD_LOGIC; -- Control Signal for Writing to Register
  signal ALUOp: STD_LOGIC_VECTOR(2 downto 0); -- Control Signal to Select ALU
Operation
  signal ALUSrc: STD_LOGIC; -- MUX Select Signal to Choose Register Contents Vs Imm
Value for ALU Input
  signal PCSrc: STD_LOGIC_VECTOR(1 downto 0); -- Control Signal for PC Update
Operation
  signal PCWrite: STD_LOGIC; -- Control Signal to Update PC to NPC

  constant clock_period: time := 10 ns; -- Clock Cycle Time
  signal stop_the_clock: boolean; -- Clock On/Off Flag

begin

  -- Mapping Ports to Signals
  uut: ControlUnit port map ( clk         => clk,
                              instruction => instruction,
                              stage    => stage,
                              IorD     => IorD,
                              MemRead     => MemRead,
                              MemWrite    => MemWrite,
                              IRWrite     => IRWrite,
                              RegDst      => RegDst,
                              MemToReg    => MemToReg,
                              RegWrite    => RegWrite,
                              ALUOp       => ALUOp,
                              ALUSrc      => ALUSrc,
                              PCSrc       => PCSrc,
                              PCWrite     => PCWrite );

  -- Start of Test Bench
  stimulus: process
  begin

          stop_the_clock <= false; -- start clock

          instruction <= "1111"; -- instruction #
          wait for 50 ns; -- wait

          stop_the_clock <= true; -- end clock
```

```
        wait; -- Wait Indefinitely
  end process;
  -- End of Test Bench

  -- Generate Clock
  clocking: process
  begin
        while not stop_the_clock loop
        clk <= '0', '1' after clock_period / 2; -- Switch Value of CLK every half period
        wait for clock_period;
        end loop;
        wait;
  end process;

end;
```

K. CPU

Source Code:

```
--------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: CPU - Behavioral
-- Programmer: Clive Gomes
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CPU is
        Port ( clk : in STD_LOGIC; -- Clock Input
        stage : out STD_LOGIC_VECTOR(3 downto 0)); -- Outputs Current Stage (Only
For Testing)
end CPU;

architecture Behavioral of CPU is
  component ControlUnit
        Port ( clk : in STD_LOGIC; -- Clock Input
        instruction : in STD_LOGIC_VECTOR(3 downto 0); -- Decoded Instruction from IR
        stage : out STD_LOGIC_VECTOR(3 downto 0); -- Outputs Current Stage (Only For
Testing)
        IorD : out STD_LOGIC; -- Control Signal for Instruction Vs Data
        MemRead : out STD_LOGIC; -- Control Signal to Read from Memory
        MemWrite : out STD_LOGIC; -- Control Signal to Write to Memory
        IRWrite : out STD_LOGIC; -- Control Signal for Updating Instruction in IR
        RegDst : out STD_LOGIC; -- MUX Select Signal to Choose Destination Register
Index
        MemToReg : out STD_LOGIC; -- MUX Select Signal to Choose ALU Output Vs
Data from Memory for Register Write
        RegWrite : out STD_LOGIC; -- Control Signal for Writing to Register
        ALUOp : out STD_LOGIC_VECTOR(2 downto 0); -- Control Signal to Select ALU
```

```
Operation
        ALUSrc : out STD_LOGIC; -- MUX Select Signal to Choose Register Contents Vs
Imm Value for ALU Input
        PCSrc : out STD_LOGIC_VECTOR(1 downto 0); -- Control Signal for PC Update
Operation
        PCWrite : out STD_LOGIC); -- Control Signal to Update PC to NPC
 end component;

 component PCUnit
        Port ( OPC : in STD_LOGIC_VECTOR (31 downto 0); -- Current PC Inputted In
        JAddress : in STD_LOGIC_VECTOR (25 downto 0); -- 26-Bit Jump Address from
Instruction (Bits 25:0)
        BOffset : in STD_LOGIC_VECTOR (31 downto 0); -- 32-Bit Sign-Extended Branch
Offset
        PCSrc : in STD_LOGIC_VECTOR (1 downto 0); -- Control Signal to Decide How To
Compute NPC
        PCWrite : in STD_LOGIC; -- Control Signal to Update PC to NPC
        Zero : in STD_LOGIC; -- Zero Result From ALU
        NPC : out STD_LOGIC_VECTOR (31 downto 0)); -- NPC Output
 end component;

 component MemoryController
        Port (IorD : in STD_LOGIC; -- Control Signal for Instruction Vs Data
        MemRead : in STD_LOGIC; -- Control Signal to Read from Memory
        MemWrite : in STD_LOGIC; -- Control Signal to Write to Memory
        MemReadOut : out STD_LOGIC; -- Output to Forward MemRead to Memory
        MemWriteOut : out STD_LOGIC; -- Output to Forward MemWrite to Memory
        MemIn : in STD_LOGIC_VECTOR (31 downto 0); -- Data Input From Memory
        MemOut : out STD_LOGIC_VECTOR (31 downto 0); -- Data Output to Memory
        WData : in STD_LOGIC_VECTOR (31 downto 0); -- Data to Write to Memory from
CPU
        RData : out STD_LOGIC_VECTOR (31 downto 0); -- Data Read Sent to CPU
        IAddress : in STD_LOGIC_VECTOR (31 downto 0); -- Instruction Address
        DAddress : in STD_LOGIC_VECTOR (31 downto 0); -- Data Address
        AddressOut : out STD_LOGIC_VECTOR (31 downto 0)); -- Output to Forwarde
Address to Memory
 end component;

 component Memory
        Port ( MemIn : in STD_LOGIC_VECTOR (31 downto 0); -- Data Input from
Memory
        MemOut : out STD_LOGIC_VECTOR (31 downto 0); -- Data Output to CPU
        MemRead : in STD_LOGIC; -- Control Signal to Read from Memory
        MemWrite : in STD_LOGIC; -- Control Signal to Write to Memory
        Address : in STD_LOGIC_VECTOR (31 downto 0)); -- Memory Address
 end component;


 component INSTRUCTION_REGISTERS
        Port ( INSTRUCTION : in STD_LOGIC_VECTOR(31 downto 0);
        IRWrite : in STD_LOGIC := '0';
        REG1 : out STD_LOGIC_VECTOR(4 downto 0);
        REG2 : out STD_LOGIC_VECTOR(4 downto 0);
```

```
        REG3 : out STD_LOGIC_VECTOR(4 downto 0);
        IMM : out STD_LOGIC_VECTOR(31 downto 0);
        JMP_ADDRESS : out STD_LOGIC_VECTOR(25 downto 0);
        INSTRUCTION_TYP : out STD_LOGIC_VECTOR(3 downto 0));
  end component;

  component REGISTERS
        Port ( READREG1 : in STD_LOGIC_VECTOR(4 downto 0);
        READREG2 : in STD_LOGIC_VECTOR(4 downto 0);
        READREG3 : in STD_LOGIC_VECTOR(4 downto 0);
        WRITEDATA_ALU : in STD_LOGIC_VECTOR(31 downto 0);
        WRITEDATA_MEM : in STD_LOGIC_VECTOR(31 downto 0);
        REGWRITE : in STD_LOGIC;
        RegDst : in STD_LOGIC;
        MemToReg : in STD_LOGIC;
        READDATA1 : out STD_LOGIC_VECTOR(31 downto 0);
        READDATA2 : out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  component ALU_32
        Port ( ALU_OP : in STD_LOGIC_VECTOR(2 downto 0);
                ALUSrc : in STD_LOGIC;
                INPUT_A : in STD_LOGIC_VECTOR(31 downto 0);
                INPUT_B : in STD_LOGIC_VECTOR(31 downto 0);
                INPUT_IMM : in STD_LOGIC_VECTOR(31 downto 0);
                OUTPUT : out STD_LOGIC_VECTOR(31 downto 0);
                ZERO : out STD_LOGIC);
  end component;

  signal PCSrc_MCU: STD_LOGIC_VECTOR (1 downto 0); -- PCSrc Output from MCU
  signal PCSrc_PCU: STD_LOGIC_VECTOR (1 downto 0); -- PCSrc Input to PCU
  signal PCWrite_MCU: STD_LOGIC; -- PCWRite Output from MCU
  signal PCWrite_PCU: STD_LOGIC; -- PCWrite Input to PCU

  signal IorD_MCU: STD_LOGIC; -- IorD Output from MCU
  signal IorD_MemControl: STD_LOGIC; -- IorD Input to MemControl
  signal MemRead_MCU: STD_LOGIC; -- MemRead Output from MCU
  signal MemRead_MemControl: STD_LOGIC; -- MemRead Input to MemControl
  signal MemWrite_MCU: STD_LOGIC; -- MemWrite Output from MCU
  signal MemWrite_MemControl: STD_LOGIC; -- MemWrite Input to MemControl

  signal IAddress : STD_LOGIC_VECTOR (31 downto 0); -- Instruction Address
  signal DAddress : STD_LOGIC_VECTOR(31 downto 0); -- Data Address

  signal MemOut_Cache: STD_LOGIC_VECTOR (31 downto 0); -- Data Output from
MemControl
  signal MemIn_Cache: STD_LOGIC_VECTOR (31 downto 0); -- Data Input to Cache
  signal MemOut_MemControl: STD_LOGIC_VECTOR (31 downto 0); -- Data Output from
Cache
  signal MemIn_MemControl: STD_LOGIC_VECTOR (31 downto 0); -- Data Input to
MemControl

  signal MemReadOut : STD_LOGIC; -- MemRead Output from MemControl
```

```
  signal MemRead_Cache: STD_LOGIC; -- MemRead Input to Cache
  signal MemWriteOut : STD_LOGIC; -- MemWrite Output from MemControl
  signal MemWrite_Cache: STD_LOGIC; -- MemWrite Input to Cache
  signal AddressOut : STD_LOGIC_VECTOR (31 downto 0); -- Address Output from
MemControl
  signal Address_Cache: STD_LOGIC_VECTOR (31 downto 0); -- Address Input to Cache

  signal RData_MemControl: STD_LOGIC_VECTOR(31 downto 0); -- Memory Data Output
from MemControl
  signal INSTRUCTION_IR: STD_LOGIC_VECTOR(31 downto 0); -- Instruction Input to
IR
  signal IRWrite_MCU: STD_LOGIC; -- IRWrite Output from MCU
  signal IRWrite_IR: STD_LOGIC; -- IRWrite Input to IR
  signal IMM_IR: STD_LOGIC_VECTOR(31 downto 0); -- 32-Bit Sign-Extended Imm Value
Output from IR
  signal BOffset_PCU: STD_LOGIC_VECTOR(31 downto 0); -- 32 Bit Sign-Extended
Branch Offset Input to PCU
  signal JMP_ADDRESS_IR: STD_LOGIC_VECTOR(25 downto 0); -- 26 Bit Jump Address
Output from IR
  signal JAddress_PCU: STD_LOGIC_VECTOR(25 downto 0); -- 26 Bit Jump Address
Input to PCU
  signal INSTRUCTION_TYP_IR : STD_LOGIC_VECTOR(3 downto 0); -- Decoded
Instruction Output from IR
  signal instruction_MCU: STD_LOGIC_VECTOR(3 downto 0); -- Decoded Instruction
Input to MCU


  signal REG1_IR: STD_LOGIC_VECTOR(4 downto 0); -- Read Register 1 Index Output
from IR
  signal READREG1_REGS: STD_LOGIC_VECTOR(4 downto 0); -- Read Register 1 Index
Input to Registers
  signal REG2_IR: STD_LOGIC_VECTOR(4 downto 0); -- Read Register 2/Write Register 1
Index Output from IR
  signal READREG2_REGS: STD_LOGIC_VECTOR(4 downto 0); -- Read Register 2.Write
Register 1 Index Input to Registers
  signal REG3_IR: STD_LOGIC_VECTOR(4 downto 0); -- Write Register 2 Index Output
from IR
  signal READREG3_REGS: STD_LOGIC_VECTOR(4 downto 0); -- Write Register 2 Index
Input to Registers
  signal RegWrite_MCU: STD_LOGIC; -- RegWrite Output from MCU
  signal REGWRITE_REGS: STD_LOGIC; -- RegWrite Input to Registers
  signal MemToReg_MCU: STD_LOGIC; -- MemToReg Output from MCU
  signal MemToReg_REGS: STD_LOGIC; -- MemToReg Input to Registers
  signal RegDst_MCU : STD_LOGIC; -- RegDst Output from MCU
  signal RegDst_REGS : STD_LOGIC; -- RegDst Input to Registers

  signal ALUOp_MCU: STD_LOGIC_VECTOR(2 downto 0); -- ALUOp Output from MCU
  signal ALU_OP_ALU: STD_LOGIC_VECTOR(2 downto 0); -- ALUOp Input to ALU
  signal ALUSrc_MCU: STD_LOGIC; -- ALUSrc Output from MCU
  signal ALUSrc_ALU: STD_LOGIC; -- ALUSrc Input to ALU
  signal READDATA1_REGS: STD_LOGIC_VECTOR(31 downto 0); -- Read Register 1
Contents Output from Registers
  signal INPUT_A_ALU: STD_LOGIC_VECTOR(31 downto 0); -- Read Register 1 Contents
```

```
Input to ALU
  signal READDATA2_REGS: STD_LOGIC_VECTOR(31 downto 0); -- Read Register 2
Contents Output from Registers
  signal INPUT_B_ALU: STD_LOGIC_VECTOR(31 downto 0); -- Read Register 2 Contents
Input to ALU
  signal INPUT_IMM_ALU: STD_LOGIC_VECTOR(31 downto 0); -- Imm Value Input to
ALU
  signal OUTPUT_ALU: STD_LOGIC_VECTOR(31 downto 0); -- ALU Output
  signal Zero_PCU: STD_LOGIC; -- Zero Result Input to PCU
  signal ZERO_ALU: STD_LOGIC; -- Zero Result Output from ALU

  signal WRITEDATA_ALU_REGS : STD_LOGIC_VECTOR(31 downto 0); -- Write Data
from ALU Input to Registers
  signal WData_MemControl: STD_LOGIC_VECTOR(31 downto 0); -- Write Data from
Memory Output from MemControl
  signal WRITEDATA_MEM_REGS: STD_LOGIC_VECTOR(31 downto 0); -- Write Data
from Memory Input to Registers

  signal OPC_PCU : STD_LOGIC_VECTOR (31 downto 0); -- Current PC Input to PCU
  signal NPC_PCU : STD_LOGIC_VECTOR (31 downto 0); -- NPC Output from PCU

begin

  -- Mapping Appropriate Ports
  MCU : ControlUnit port map ( clk      => clk,
                               instruction => instruction_MCU,
                               stage    => stage,
                               IorD     => IorD_MCU,
                               MemRead      => MemRead_MCU,
                               MemWrite     => MemWrite_MCU,
                               IRWrite      => IRWrite_MCU,
                               RegDst       => RegDst_MCU,
                               MemToReg     => MemToReg_MCU,
                               RegWrite     => RegWrite_MCU,
                               ALUOp        => ALUOp_MCU,
                               ALUSrc       => ALUSrc_MCU,
                               PCSrc        => PCSrc_MCU,
                               PCWrite      => PCWrite_MCU );

  -- Mapping Appropriate Ports
  PCU : PCUnit port map ( OPC          => OPC_PCU,
                    JAddress => JAddress_PCU,
                    BOffset  => BOffset_PCU,
                    PCSrc    => PCSrc_PCU,
                    PCWrite  => PCWrite_PCU,
                    Zero     => Zero_PCU,
                    NPC      => NPC_PCU );

  -- Mapping Appropriate Ports
  MemControl : MemoryController port map ( IorD      => IorD_MemControl,
                                     MemRead      => MemRead_MemControl,
                                     MemWrite     => MemWrite_MemControl,
                                     MemReadOut   => MemReadOut,
```

```
                                MemWriteOut => MemWriteOut,
                                MemIn       => MemIn_MemControl,
                                MemOut      => MemOut_MemControl,
                                WData       => WData_MemControl,
                                RData       => RData_MemControl,
                                IAddress    => IAddress,
                                DAddress    => DAddress,
                                AddressOut  => AddressOut );

-- Mapping Appropriate Ports
Cache : Memory port map ( MemIn    => MemIn_Cache,
                          MemOut   => MemOut_Cache,
                          MemRead  => MemRead_Cache,
                          MemWrite => MemWrite_Cache,
                          Address  => Address_Cache );

-- Mapping Appropriate Ports
IR : INSTRUCTION_REGISTERS port map ( INSTRUCTION => INSTRUCTION_IR,
                                      IRWrite         => IRWrite_IR,
                                      INSTRUCTION_TYP => INSTRUCTION_TYP_IR,
                                      REG1            => REG1_IR,
                                      REG2            => REG2_IR,
                                      REG3            => REG3_IR,
                                      IMM             => IMM_IR,
                                      JMP_ADDRESS => JMP_ADDRESS_IR);

-- Mapping Appropriate Ports
REGS: REGISTERS port map ( READREG1  => READREG1_REGS,
                           READREG2  => READREG2_REGS,
                           READREG3  => READREG3_REGS,
                           WRITEDATA_ALU => WRITEDATA_ALU_REGS,
                           WRITEDATA_MEM => WRITEDATA_MEM_REGS,
                           REGWRITE  => REGWRITE_REGS,
                           RegDst => RegDst_REGS,
                           MemToReg   => MemToReg_REGS,
                           READDATA1 => READDATA1_REGS,
                           READDATA2 => READDATA2_REGS);

-- Mapping Appropriate Ports
ALU : ALU_32 port map ( ALU_OP    => ALU_OP_ALU,
                        ALUSrc    => ALUSrc_ALU,
                        INPUT_A   => INPUT_A_ALU,
                        INPUT_B   => INPUT_B_ALU,
                        INPUT_IMM => INPUT_IMM_ALU,
                        OUTPUT    => OUTPUT_ALU,
                        ZERO      => ZERO_ALU );

-- Wiring Connections

PCSrc_PCU <= PCSrc_MCU; -- MPU Sends PCSrc to PCU
PCWrite_PCU <= PCWrite_MCU; -- MPU Sends PCWrite to PCU
IorD_MemControl <= IorD_MCU; -- MPU Sends IorD to MemControl
MemRead_MemControl <= MemRead_MCU; -- MPU Sends MemRead to MemControl
```

```
  MemWrite_MemControl <= MemWrite_MCU; -- MPU Sends MemRead to MemControl

  IAddress <= NPC_PCU; -- PCU Sends Instruction Address to MemControl
  OPC_PCU <= NPC_PCU; -- Loop Back PCU NPC Output to Current Pc Input

  MemIn_Cache <= MemOut_MemControl; -- MemControl Sends Data to Cache
  MemIn_MemControl <= MemOut_Cache; -- Cache Sends Data to MemControl
  MemRead_Cache <= MemReadOut; -- MemControl Sends MemRead to Cache
  MemWrite_Cache <= MemWriteOut; -- MemControl Sends MemWrite to Cache
  Address_Cache <= AddressOut; -- MemControl Sends Address to Cache

  INSTRUCTION_IR <= RData_MemControl; -- MemControl Sends Instruction Word from
Memory to IR
  IRWrite_IR <= IRWrite_MCU; -- MCU Sends IRWrite to IR
  BOffset_PCU <= IMM_IR; -- IR Sends 32-Bit Sign-Extended Branch Offset to PCU
  JAddress_PCU <= JMP_ADDRESS_IR; -- IR Sends 26-Bit Jump Address to IR
  instruction_MCU <= INSTRUCTION_TYP_IR; -- IR Sends Decoded Instruction to MCU


  READREG1_REGS <= REG1_IR; -- IR Sends Read Register 1 Index to Registers
  READREG2_REGS <= REG2_IR; -- IR Sends Read Register 2/Write Register 1 Index to
Registers
  READREG3_REGS <= REG3_IR; -- IR Sends Write Register 2 Index to Registers
  WRITEDATA_MEM_REGS <= RData_MemControl; -- MemControl Sends Data from
Memory to Registers
  REGWRITE_REGS <= RegWrite_MCU; -- MCU Sends RegWrite to Registers
  MemToReg_REGS <= MemToReg_MCU; -- MCU Sends MemToReg to Registers
  RegDst_REGS <= RegDst_MCU; -- MCU Sends RegDst to Registers

  ALU_OP_ALU <= ALUOp_MCU; -- MCU Sends ALUOP to ALU
  ALUSrc_ALU <= ALUSrc_MCU; -- MCU Sends ALUSrc to ALU
  INPUT_A_ALU <= READDATA1_REGS; -- Registers Sends Read Register 1 Contents to
ALU
  INPUT_B_ALU <= READDATA2_REGS; -- Registers Sends Read Register 2 Contents to
ALU
  INPUT_IMM_ALU <= IMM_IR; -- Registers Sends Imm Value to ALU
  Zero_PCU <= ZERO_ALU; -- ALU Sends Zero Result to PCU
  DAddress <= OUTPUT_ALU; -- ALU Sends Operation Result to MemControl as Data
Address
  WRITEDATA_ALU_REGS <= OUTPUT_ALU; -- ALU Sends Operation Result to
Registers for Register Write
  WData_MemControl <= READDATA2_REGS; -- Registers Sends Read Register 2
Contents to MemControl for Memory Write

end Behavioral;
```

Test Bench:

```
--------------------------------------------------------------------------------
-- Project Name: ECE 485 Final Project - Building A CPU
-- Module Name: CPU_tb - Behavioral
-- Programmer: Clive Gomes
--------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity CPU_tb is
end;

architecture bench of CPU_tb is

  component CPU
        Port ( clk : in STD_LOGIC; -- Clock Input
        stage : out STD_LOGIC_VECTOR(3 downto 0)); -- Outputs Current Stage (Only
For Testing)
  end component;

  signal clk: STD_LOGIC; -- Clock Input
  signal stage: STD_LOGIC_VECTOR(3 downto 0); -- Outputs Current Stage (Only For
Testing)

  constant clock_period: time := 10 ns; -- Clock Cycle Time
  signal stop_the_clock: boolean; -- Clock On/Off Flag

begin

  -- Mapping Ports to Signals
  uut: CPU port map ( clk   => clk,
                      stage => stage );

  -- Start of Test Bench

  stimulus: process
  begin

        stop_the_clock <= false; -- start clock

        wait for 1000 ns;-- for 1000 ns; -- Wait

        stop_the_clock <= true; -- end clock

        wait; -- Wait Indefinitely
  end process;

  -- End Test Bench

  -- Generate Clock
  clocking: process
  begin
        while not stop_the_clock loop
```

```
        clk <= '0', '1' after clock_period / 2; -- Switch Value of CLK every half period
        wait for clock_period;
        end loop;
        wait;
  end process;

end;
```