ECE 485
Computer Organization & Design
Project 2: MIPS CPU Design and Implementation
(Multi-cycle)

By: Annabella Chong (A20386586)
& Leah Kosmin (A20371866)

Due Date: 12/07/19

Acknowledgment: *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.*

Name:
Annabella Chong

Signature:
Annabella Chong

Leah Kosmin

I. ABSTRACT

This report covers the design of a custom 32-bit RISC processor containing an instruction set from MIPS. The goal is to implement a multi cycle datapath, using VHDL, which consists of multiple components that are part of the RISC processor. Three types of instructions must be supported: R-type, I-type and J-type.

II. INTRODUCTION

The objective of this lab is to design a MIPS version made up of a multi-cycle datapath consisting of five cycles: (1) Instruction Fetch, (2) Instruction Decode, (3) Execute, (4) Memory, and (5) Write back. At a first glance, designing a processor seems very complicated. Afterall, the program must be able to execute MIPS instructions like the ones that were taught in class. But similar to project I where a 32-bit full adder was implemented using smaller adders, the complex datapath can be designed using simpler components and then connecting them via port mapping in VHDL.

OpCode [31:26]	Function Field [5:0]	Instruction	Operation (example)
100011	-	lw	lw $t1, 100($t2)
000000	100000	add	add $s1, $t2, $s7
000000	100010	sub	sub $t1, $t3, $t8
000000	100100	and	and $t1, $t2, $t3
000000	100101	or	or $s1, $s2, $s3
000010	-	j	j 100
000100	-	beq	beq $t1, $t2, 100
001000	-	addi	addi $t4, $t2, 100

Table 1: Required MIPS Instruction Set

Table 1 displays the required instructions that the processor must enable. For every instruction type, the datapath differs as R-type, I-type, and J-type instructions must be supported.

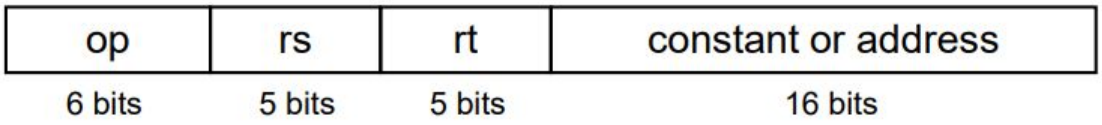
- **Description of R-Type**

An R-type instruction consists of an op code, rs (first source register), rt (second source register), rd (destination register), shamt(shift amount) and funct (function code). An op code is used to determine which instruction will be executed, while funct is used to extend the op code essentially. The instructions add, sub, and, or are classified as R-type instructions.

- **Description of I-Type**

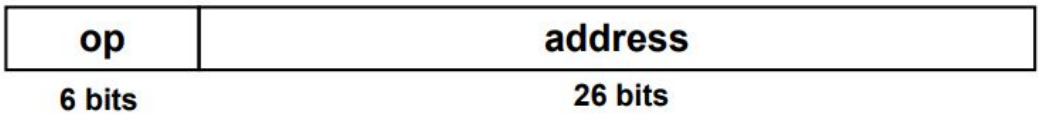
Similar to the R-type instruction and J-type has an op, rs and rt. However, for the 16 least significant bits, a constant or address is encoded. This is because for instructions like

addi, and immediate value is added to rs, and place in rt. The instructions addi, lw, beq are I-type instructions.



- **Description of J-type**

  For a J-type instruction, the only fields are an op code and an address.The instruction j, which stands for jump, will automatically jump to whatever address is specified in the field.



III. SYSTEM DESIGN

As mentioned in the introduction, designing a multicycle datapath can be a huge task and very challenging. Many details must be managed properly in order to obtain accurate results. Instead of creating one huge VHDL file attempting to design the datapath, it is better to break the path into its subcomponents, similar to what was done for the 32-bit adder. By dividing the datapath into subcomponents, each component can be individually tested in order to reduce the chances of error. Appropriate signals had to be used in order to ensure that correct actions would be taken for each instruction.

Once the components are created and verified to be working correctly, the components can be linked properly using port mapping and knowledge on how the datapath works. Listed below is the datapath and the component descriptions for the project.

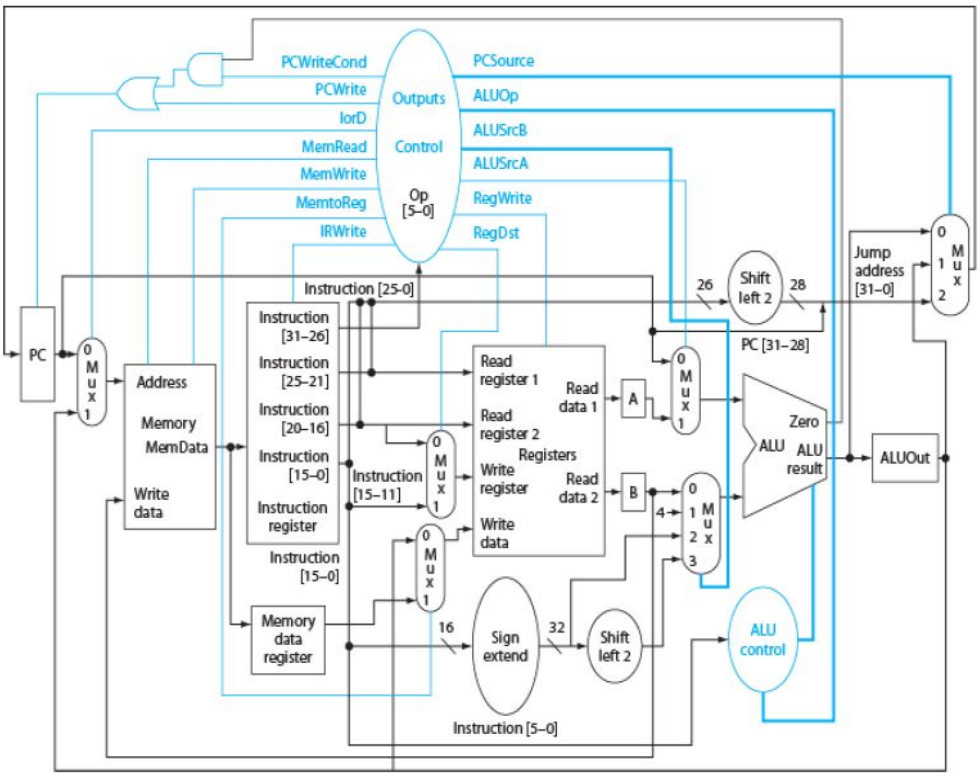


Figure 1: Datapath

- **Description of MUX**

  As shown in figure 1, the datapath requires multiple muxes. For this project, a 2-input and 4-input mux were designed. The mux simple takes either 2 or 4 inputs, each input being 32-bits, and outputs only one of the inputs. The way the output is determined is using a 1-bit select signal from the Control Unit, allowing the appropriate output to pass. Although there is also a 3-input mux in the datapath, the 4-input mux could just be reused by allowing one of the inputs to be empty.

- **Description of Register**

  The register (referred to as "r.vhdl"), is a basic component that can be used to represent the register, but also be utilized as other parts such as the PC. The register consists of two inputs, an enable and D, which is 32-bits. Simply, when enable is set to "1" from the Control Unit, the output Q obtains the value of D. In the datapath, the register will be used for representing the program counter and ALUOut.

- **Description of Sign Extend**

  For certain instructions, a sign extension must be done such that an input of 16-bits gets outputted at 32-bits. The way sign extension works is if the most significant bit is a "0" or

"1", then the input gets sign extended with sixteen zeros or ones. This guarantees that the sign of the original value is not lost.

- **Description of Shift Left 2**
  The shift left two takes an input of 32-bits and moves every bit to the left by two places, replacing the right remaining spots with zeros. This was done by anding the input with "00". On the right side of figure 1, there is a shift left two that deals with sign extension as well, and this was dealt with by manipulating the sizing in the datapath itself.

- **Description of Memory**
  The main responsibilities of memory are reading and writing. During reading, data is grabbed from a specific memory address and placed into a register. During writing, data is placed in a predefined memory address. The inputs required are then the address, the data to be written if the instruction is a write, and the signal indicating if the instruction requires reading or writing. If the write signal is "1", the inputted data will be written into memory at the given address. If the read signal is "1", the data obtained from memory will be placed into an output variable representing the register receiving the data.

- **Description of Instruction Register**
  When looking at the figure 1, the instruction register displays multiple outputs based on a range of bits for the instruction. As described in the introduction, the three instruction types have multiple fields. Instruction register essentially parses the instructions appropriately to diagnose which instruction type is being used, the opcode, the registers, and so on. This is simply done by assigning appropriate variable names to the bit ranges of each instruction and MIPS makes this simple to do since in both R and I-type instructions, the fields bit ranges do not drastically differ from one another. For example, rs is in the same bit range for both R-type and I-typ instructions.

- **Description of Register File**
  The register file is an array of processor registers in the CPU. For a 32-bit RISC implementation, there are 32 registers in total. After receiving instructions from the instruction register, the register file decodes the instructions into three inputs: Read Register 1, Read Register 2, and Write Register. Read Registers 1 and 2 simply become the outputs Read Data 1&2. While Write Register tells the processor where to write the data.

- **Description of ALU**

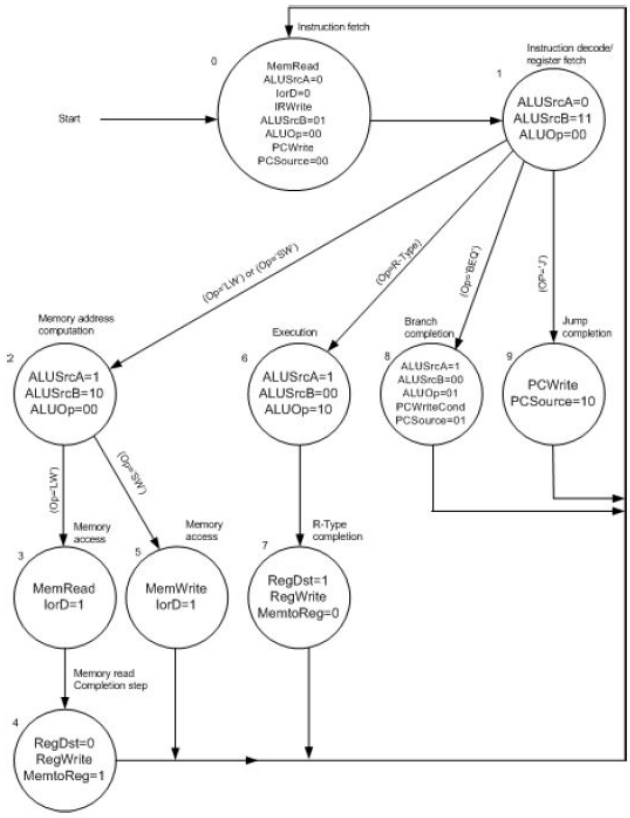
  The ALU is essentially the calculator of the datapath. The ALU performs addition, subtraction, and, or, etc. The zero flag indicates whether or not the two inputs are equal to one another. Naturally, since the possibility of overflow or carryover is possible, and carryin and carryout is required as well. In this project, the 32-bit adder from project 1 had to be utilized for the “add” function. The adder was also used to implement “subtract”.

- **Description of ALU Control**

  The ALU Control is the intermediate step between the main control unit and the ALU. It essentially tells the ALU which operation to perform. The ALU Control uses an opcode and func obtained from the Control Unit and instruction register in order to determine what operation to perform. It then passes the information to the ALU, telling it what to do.

- **Description of Control Unit**

  The control unit was coded as a finite state machine. Each stage in the FSM is representative of the 5 steps for the MIPS datapath- instruction fetch (IF), instruction decode and register fetch (ID), execution and memory address computation and branch completion (EX), memory access or R-type instruction completion (MEM) and memory read completion (WB). These stages are then broken down into the control signals required for the different types of MIPS instructions. For this project, the following types of instructions were taken into account- R-type, LW, SW, jumps, BEQ and add instructions. This is represented by the figure below.



The Control Unit is extremely important to the datapath. The control unit is what sends the appropriate signals to all the different components, telling them what to do. The control unit includes 13 signals that are issued at different times depending on the instruction. The logic of the control unit is based on what's called a finite state machine (FSM).

The state start at zero, then automatically to state one. From there, it selects the next appropriate state in order to send out the suitable signals at each state. The transitioning between the states occurs at every clock cycle while being instructed by the opcode. In total, there are 10 different states that are available depending on the instruction and such.

- **Description of Datapath**

  Finally, once all of the components have been created and tested, the datapath could be implemented. Since the various gritty details were mostly dealt with while creating the individual components, all that had to be done in the datapath was linking all the parts together. This was still complicated, as one had to be conscious, making sure the inputs and outputs matched properly along with their corresponding signals.

After port mapping all the components, every cycle was able to use the necessary resources. Instruction Fetch utilizes PC, memory, instruction register, and ALU. Instruction decode uses the instruction register and register file. Execute uses ALUout and memory. And Write Back uses memory data register, ALUout and the register file.

- **Design Overview**

  Once the components have all been designed and tested, they are all mapped together in the datapath. The datapath.vhdl will essentially instantiate all the components that were created and then port map all the components. Every input and output value will be a signal, and that signal value must remain consistent for the datapath to work properly. The clock will also be used to synchronize the cycles through each instruction.

IV. SIMULATION RESULTS & DISCUSSION

1. 2-input Mux

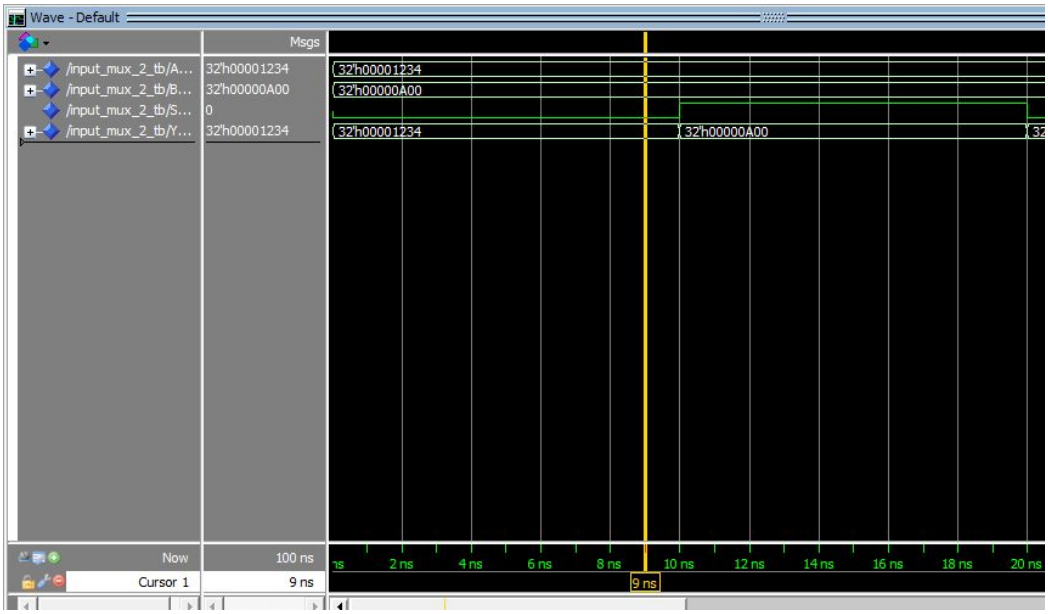


Figure 2: 2-input Mux Test Simulation - 1

Figure 2 displays the simulation results after testing the 2-input mux. For every test case, a delay of 10 ns was used. So for two tests, a complete run was 20 ns. In this case, input A was 0x00001234 and B was 0x00000A00 with a select equal to 0. With a 0 select, input A should be outputted. As displayed in the screenshot, the output was 0x0001234 which is what was expected.

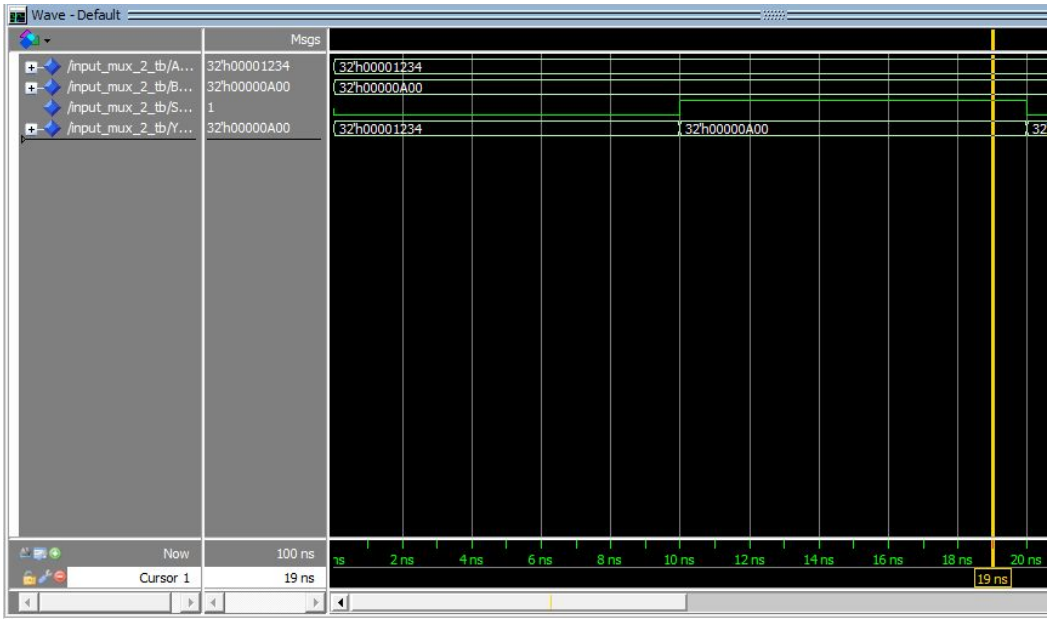


Figure 3: 2-input Mux Test Simulation - 2

For the second test, a select of 1 was used instead with the same A and B values from test 1. This time, the output should be the value for B, which is what is obtained and shown in figure 3. The mux design is simple and works well.

2. 4-input Mux

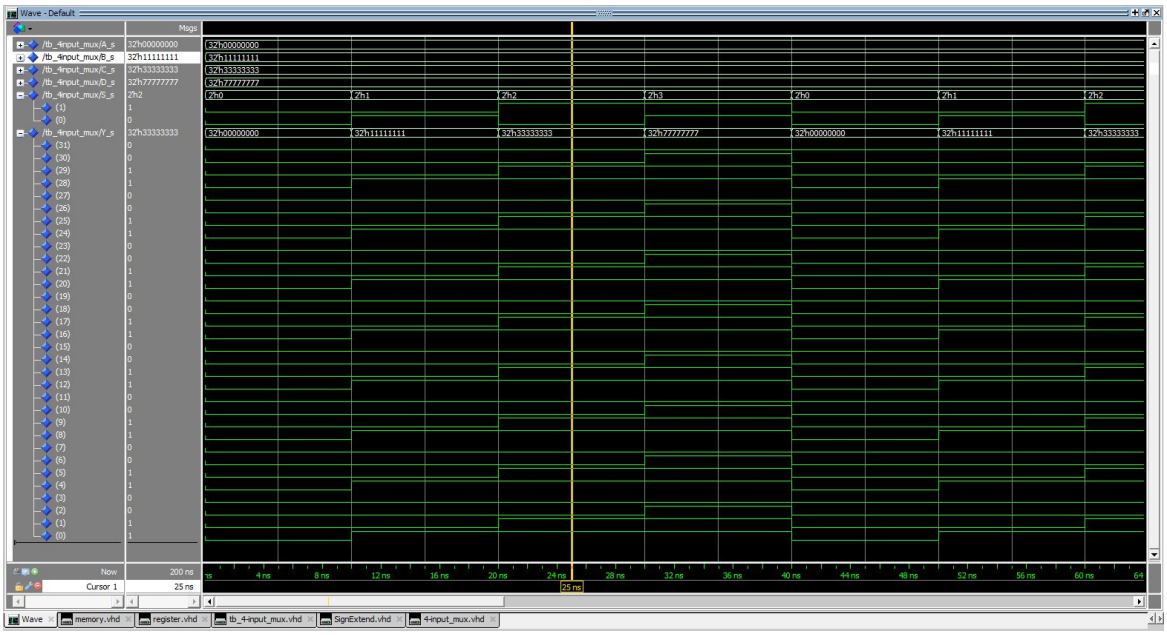


Figure 4: 4-Input Mux Test Simulation

The inputs to the 4-input mux include four 32-bit options (A,B,C,D) the select signal and one 32-bit output (Y).

In figure 4, the following values were utilized:

A= 0x0000 0000, B= 0x1111 1111, C= 0x 3333 3333, D= 0x 7777 7777
Each option was sequentially chosen for the test cases, and it can be observed that the mux functions as intended.

3. Sign Extend

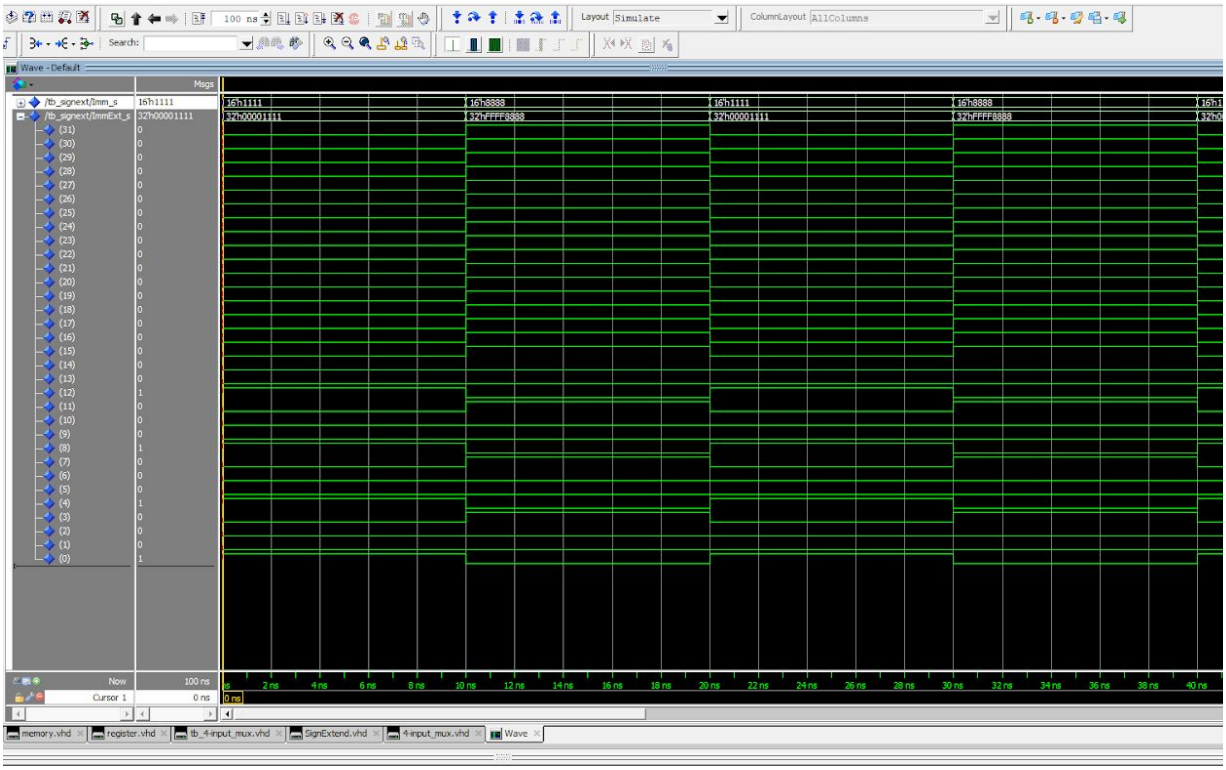


Figure 5: Sign Extend Test Simulation

Figure 5 displays the result of simulation the sign extension code. The input to the immediate field was 0x1111, followed by 0x8888 thus verifying the functionality of sign extend.

4. Shift Left 2

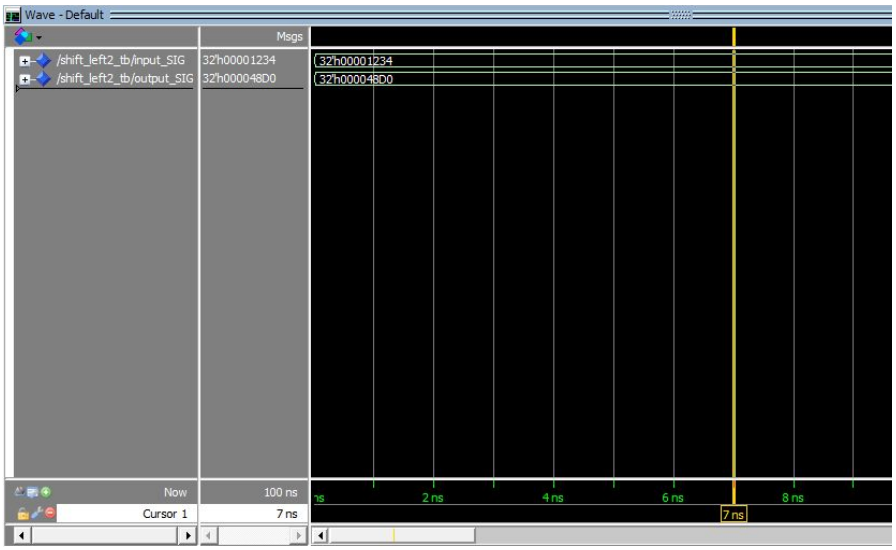


Figure 6: Shift Left 2 Test Simulation

As shown in figure 6, the input value was 0x00001234. When shifted left by 2, the result should be 0x000048D0 which is what is outputted.

5. Instruction Register

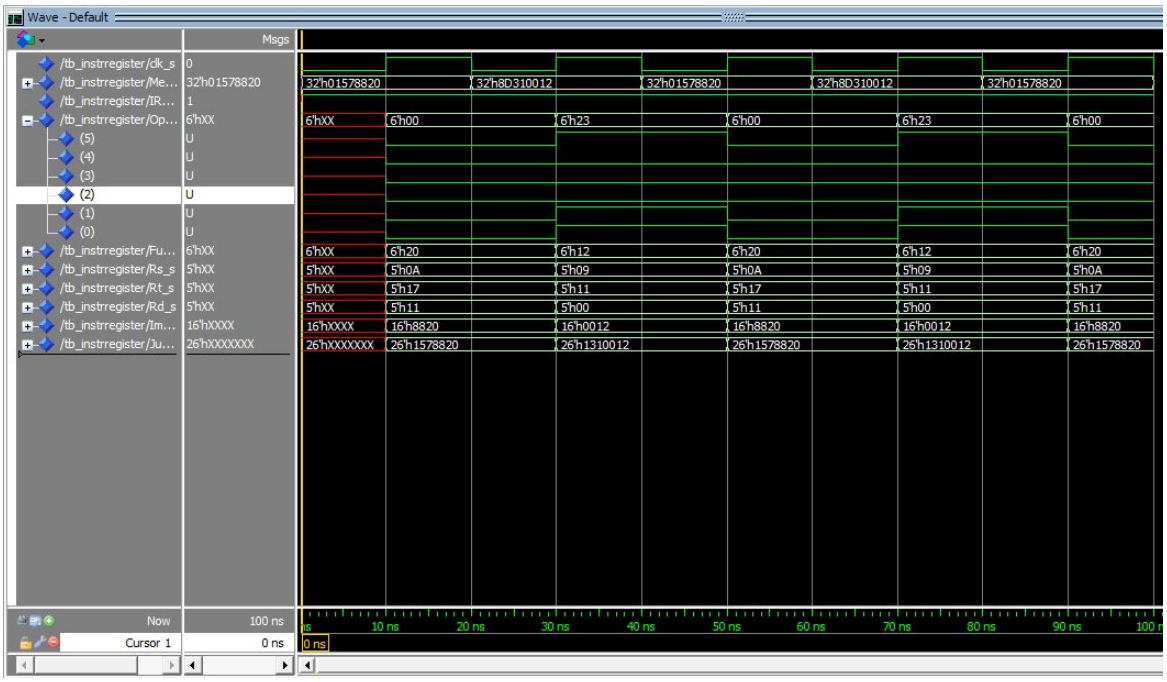


Figure 7: Instruction Register Test Simulation

The instruction register receives the instruction from memory for decoding and is then sent to the control unit, registers and ALU control. Its contents are updated with every clock cycle.

In the above test bench, the instructions used were add $s1, $t2, $s7 and lw $s1, 12($t1). As can be observed, the data is held for one clock cycle before it takes on the values of the next instruction.

6. Memory

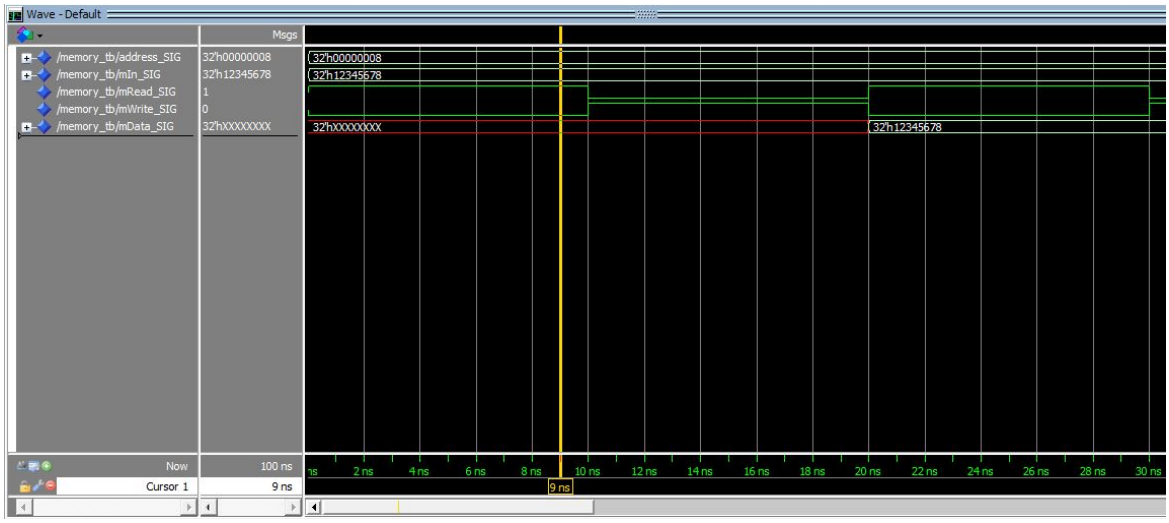


Figure 8: Memory Test Simulation - 1

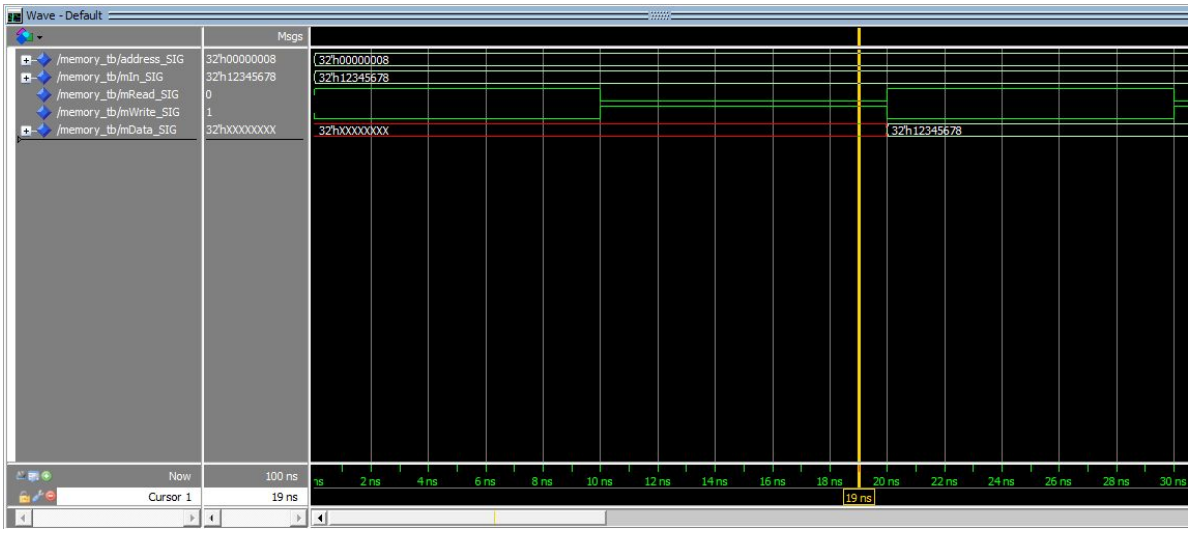


Figure 9: Memory Test Simulation - 2

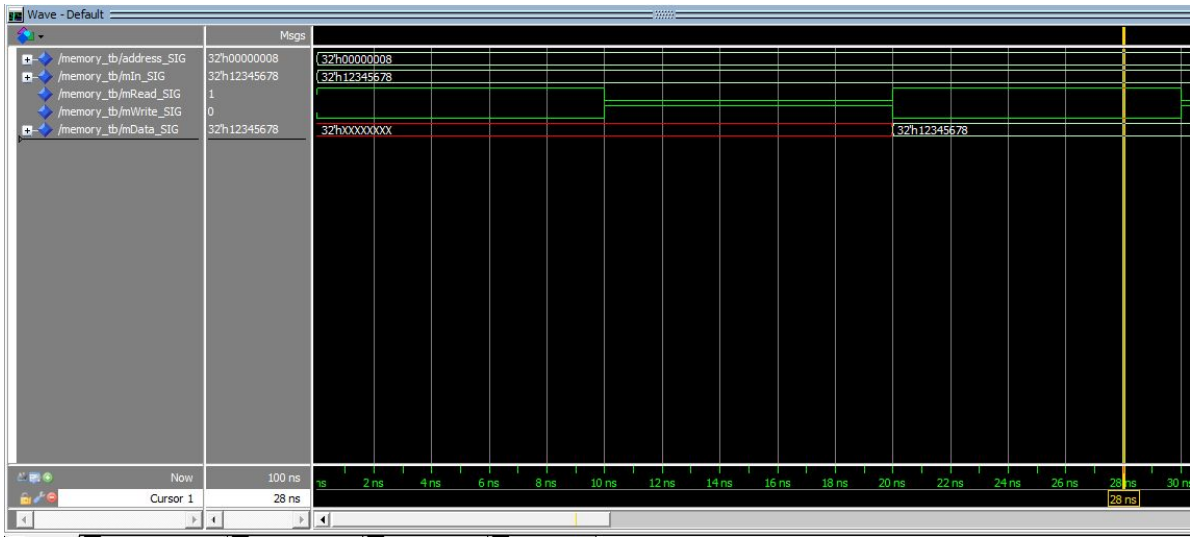


Figure 10: Memory Test Simulation - 3

In order to test that memory was able to read and write properly, tests were performed. The inputted values were as follows:
address = 0x00000008, memData = 0x12345678, read signal = 1, write signal = 0.

In the first 10 ns, illustrated by figure 8, the read signal is 1, and since no data has been inputted into memory yet, there should be no data outputted. This is proved by the output signal being a bunch of Xs, representing lack of assignment.

The second test perform between 10ns and 20ns was writing. The write signal was changed to 1, with the read signal 0. Once again, no data should be expected to output since reading is not being done.

Lastly, in the third test the read signal was adjusted to 1. Now, the outputted data displays 0x12345678 which means the variable memData was written into memory properly.

While designing memory, some adjustments had to be made. The array created representing memory had to be sized down to 2^9 due to a larger size resulting in the simulation crashing. The address had to also be converted into an integer

7. Memory Data Register

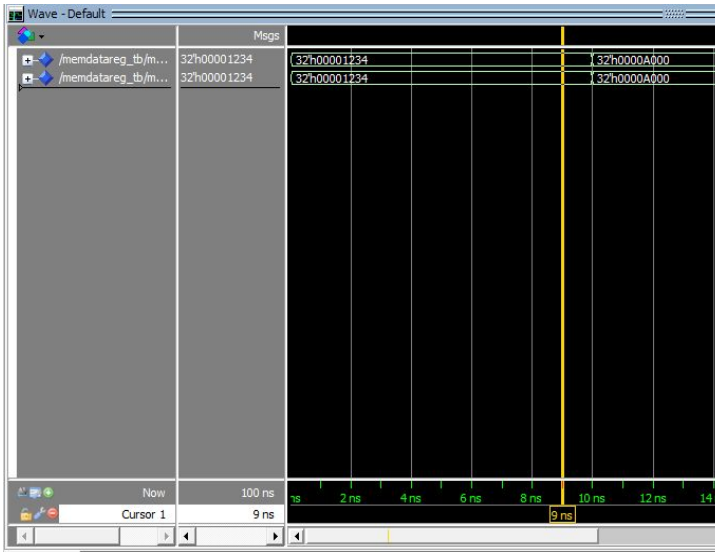


Figure 11: Memory Data Register Simulation - 1

Figures 9 display the simulations for the memory data register component. Simply, the input should equal the output. When input=0x00001234, the output=0x0001234.

8. Temporary Registers

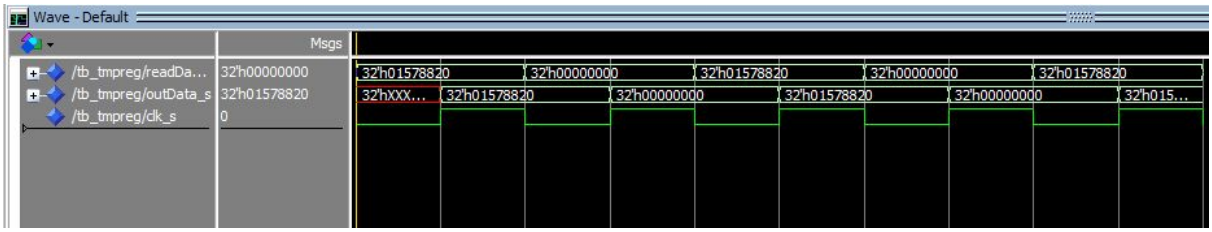


Figure 12: Temporary Registers Test Bench

Temporary Registers were used for register outputs A and B, and ALUOut to store data in between clock cycles. It can be observed from the simulation results that this was achieved.

9. Arithmetic Logic Unit

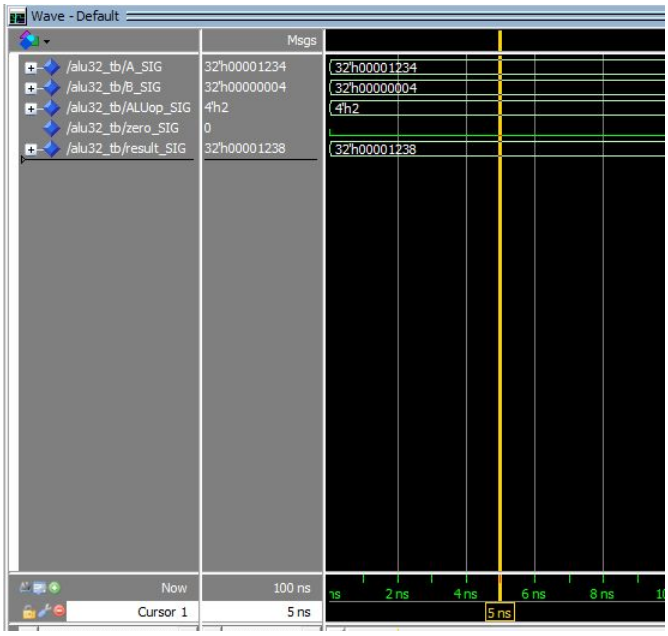


Figure 13: ALU Simulation Test - add

Displayed in the figure is the ALU simulation performing add. The inputs are: A=0x00001234, B=0x00000004, op=0x2
The expected output would be 0x00001238, which is the value obtained.

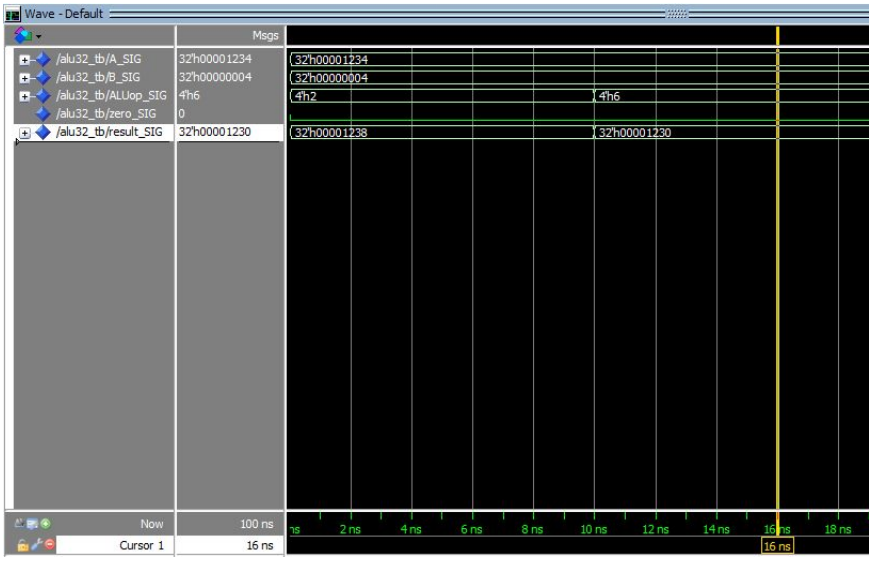


Figure 14: ALU Simulation Test - sub

The figure shows the ALU performing subtract. The inputs are the same, but op=0x6. The expected output is 0x00001230 which is the value shown in the graph.

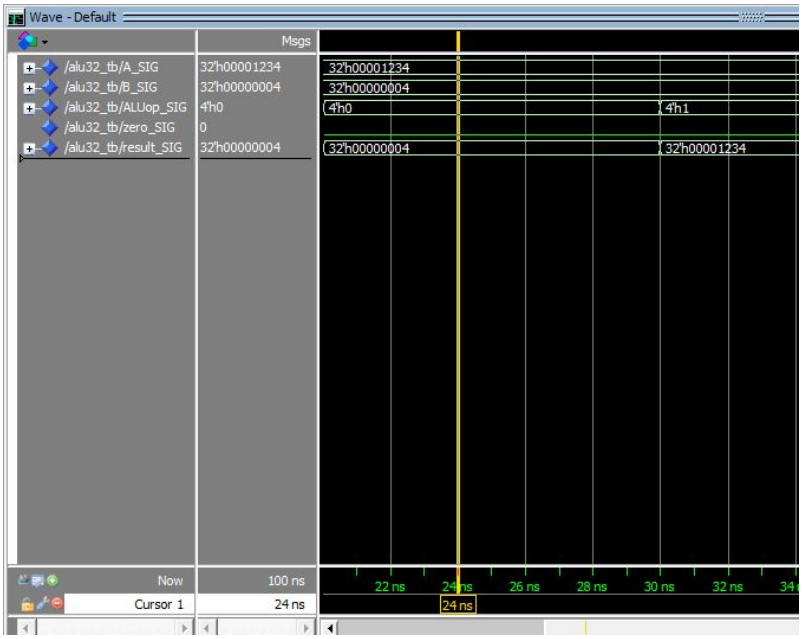


Figure 15: ALU Simulation Test - and

The figure shows the ALU performing and. The inputs are the same, but op=0x0. The expected output is 0x00000004 which is the output.

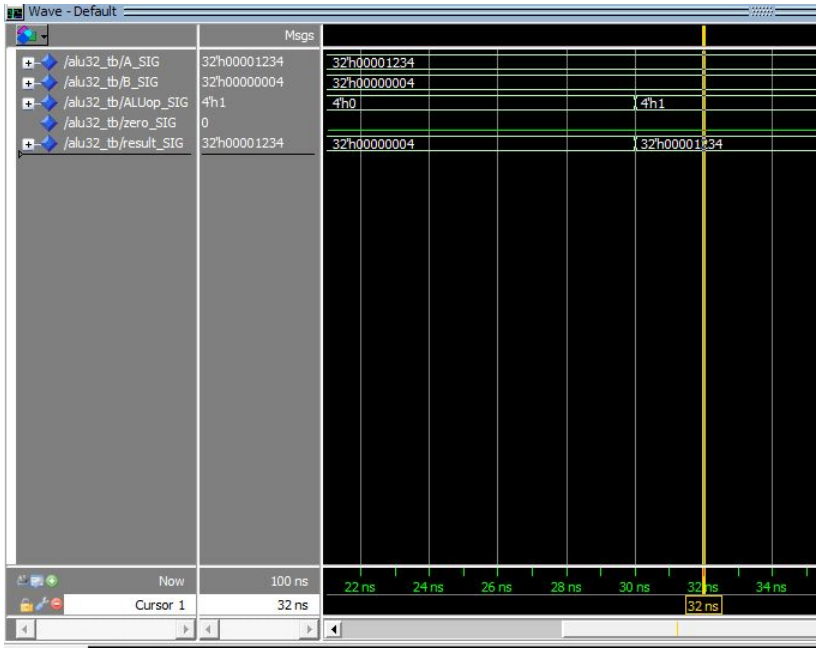


Figure 16: ALU Simulation Test - or

The figure shows the ALU performing subtract. The inputs are the same, but op=0x1. The expected output is 0x00001234 which is the value shown in the graph.

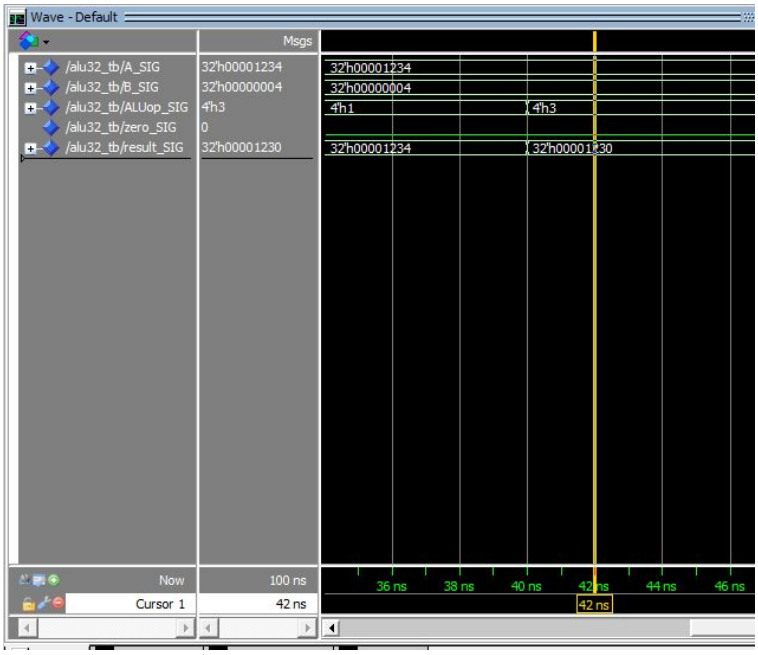


Figure 17: ALU Simulation Test - addi

The figure shows the ALU performing addi. The inputs are the same, but op=0x3. The logic being performed for an addi is xoring A and B so the expected result is 0x0001230 as show.

10. Control Unit

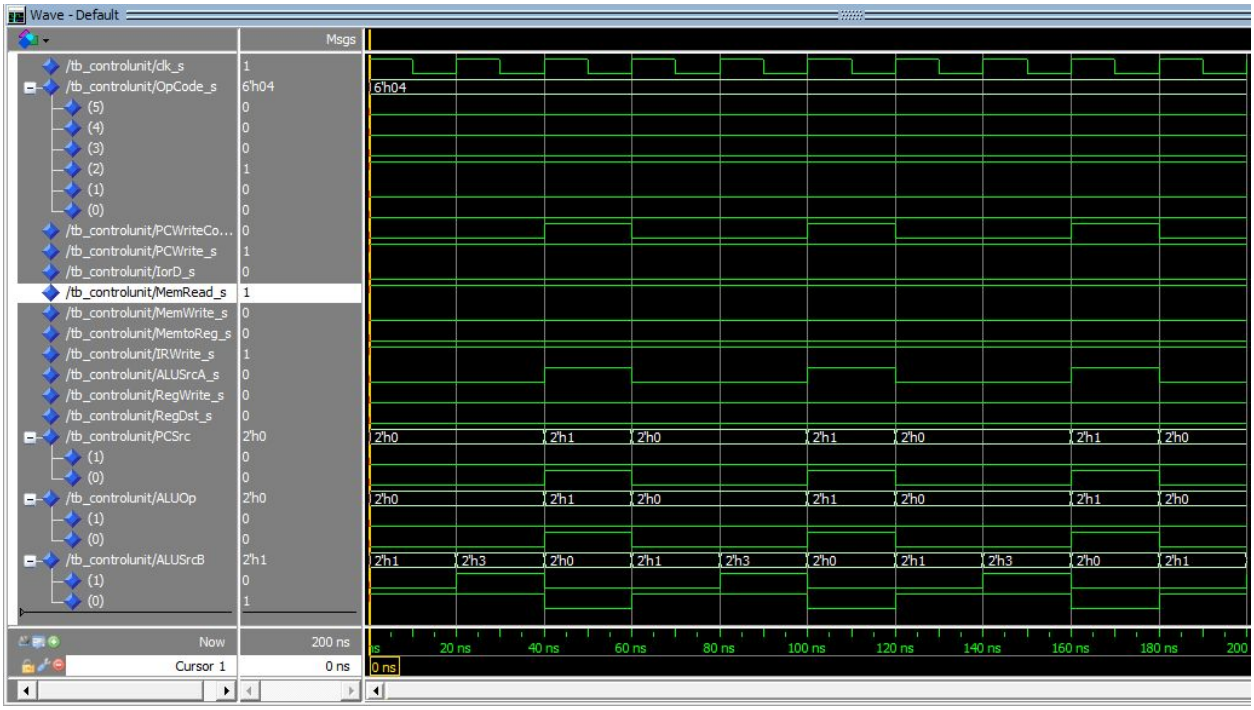


Figure 18: Control Unit Test bench Simulation

As figure 18 shows, the control unit was sending out the correct values for instruction BEQ. To verify that the system is transitioning into its appropriate next stages on the rising clock egde, the following testing was done.

11. MIPS Testing

Figure 19: BEQ, R-type and Jump State Transitions

The results for beq, R-type instructions and the jump state transitions are displayed. The values make sense with what is expected.

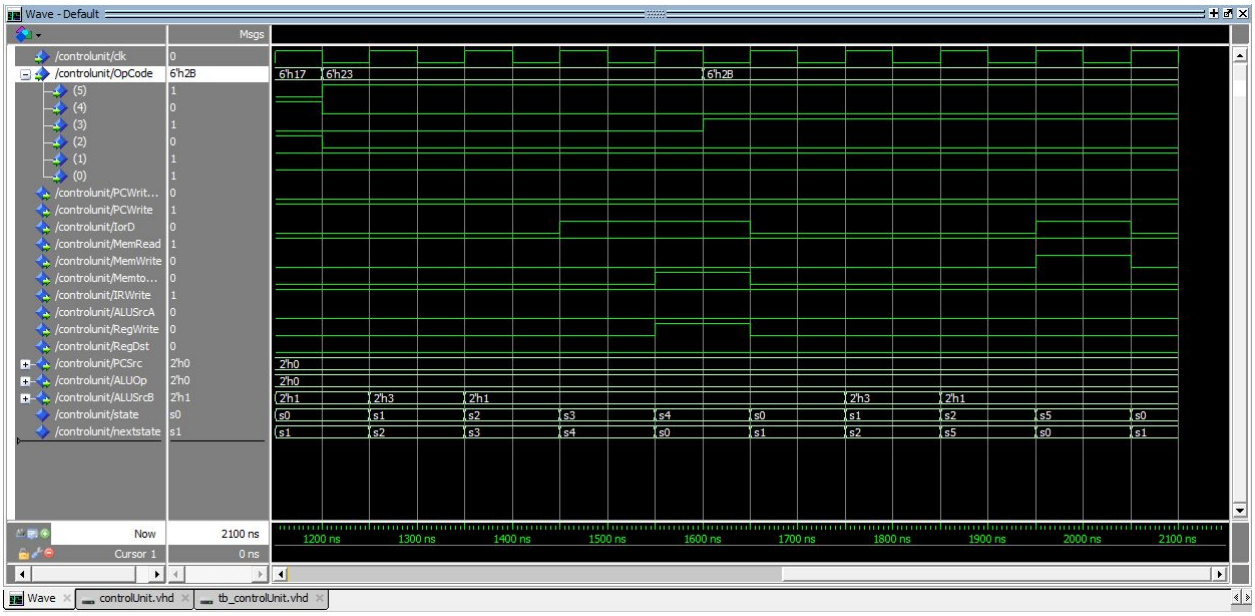


Figure 20: LW, SW State Transitions

The results from figure 20 show the testing of the lw and sw transitions which are as expected.

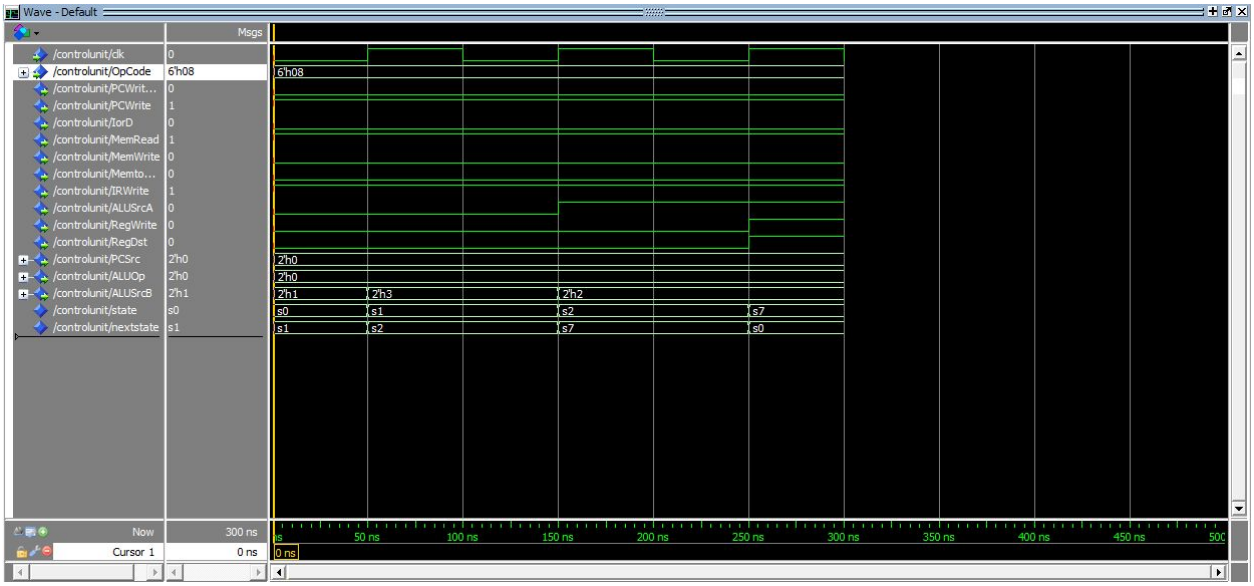


Figure 21: Addi State Transitions

Figure 21 shows the addi transitions, also as expected.

For the next figure, The program will cycle through the states depending on the type of instruction, with some instructions completing sooner than others. State transitions occur on the rising edge of the clock.

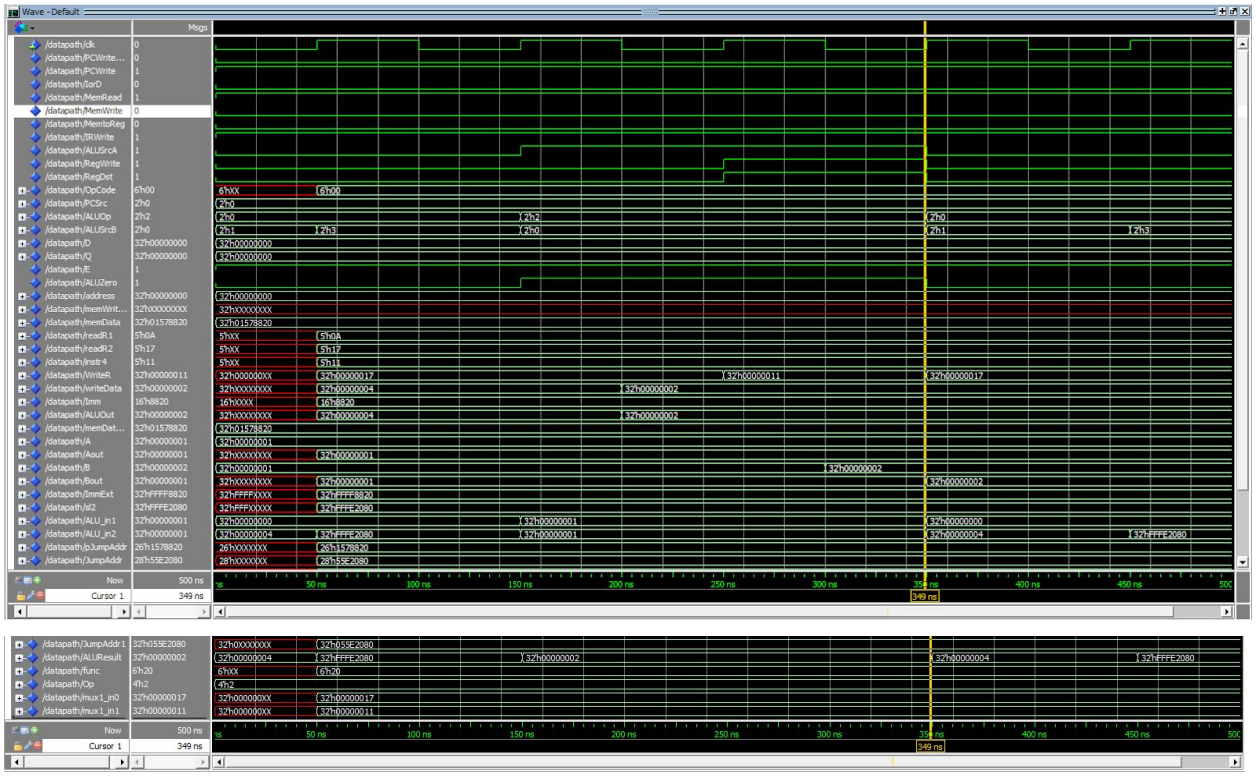
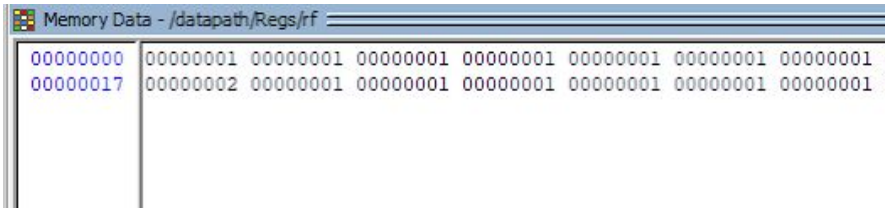


Figure 22: Add instruction Simulation



The contents of memory at address 0x0000 0000 was 0x01578820 (add $s1, $t2, $s7) while the contents of the registers were filled with 0x0000 0001. As can be observed from the simulation, the R-type instruction was executed in 3 clock cycles, and the s1 register has an updated value.

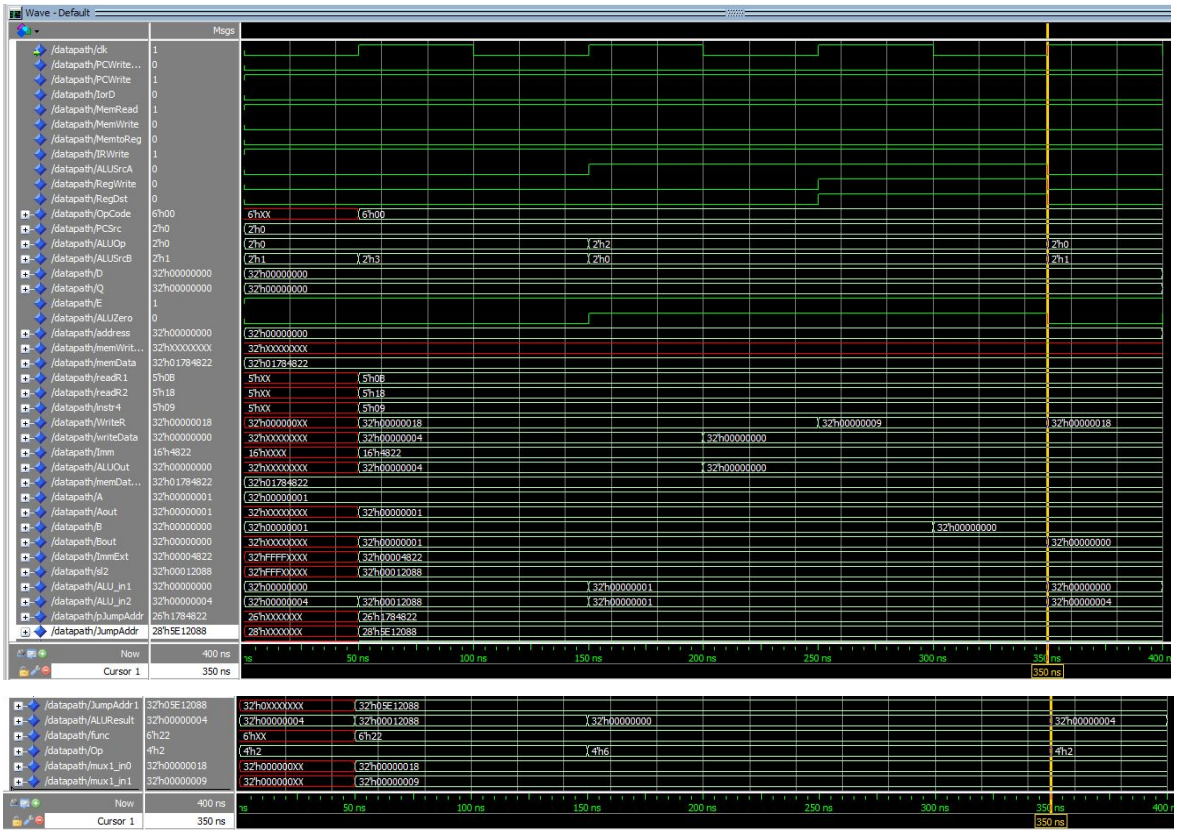


Figure 23: Sub Instruction Simulation

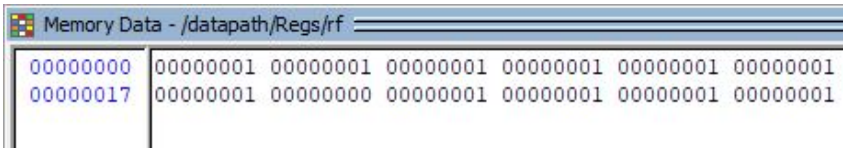


Figure 24: Register Values

Similar to before, the contents of memory at address 0x0000 0000 was 0x01784822 (sub $t1, $t3, $t8) while the contents of the registers were filled with 0x0000 0001. As can be observed from the simulation, the R-type instruction was executed in 3 clock cycles, and the $t1 register has an updated value.

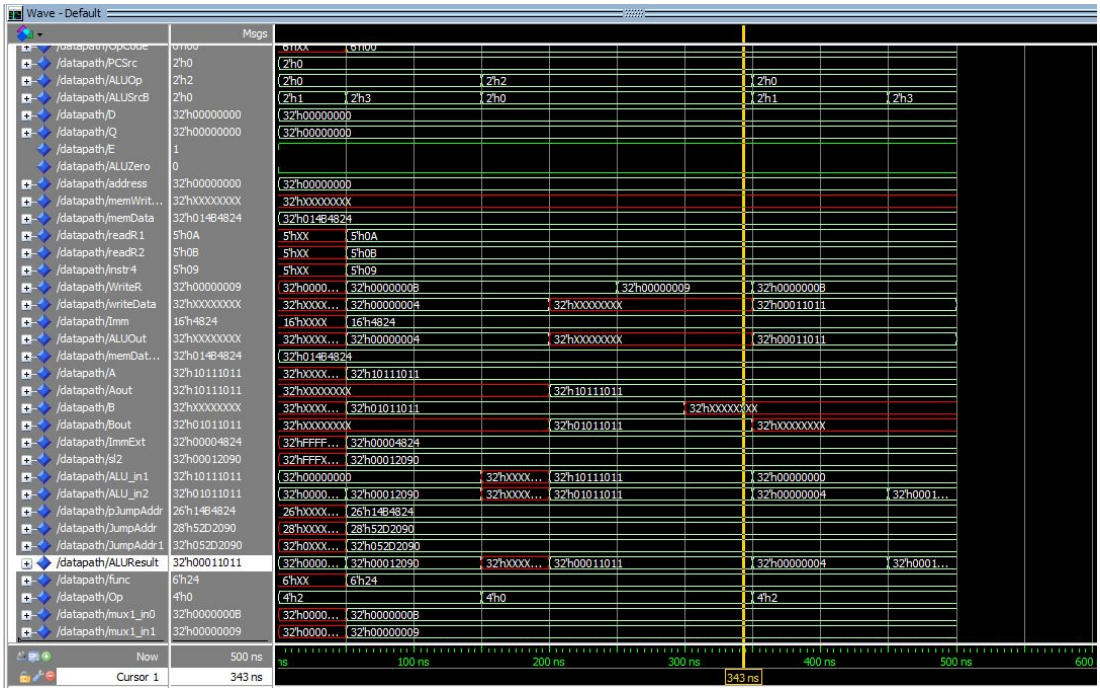


Figure 25: And Instruction Simulation

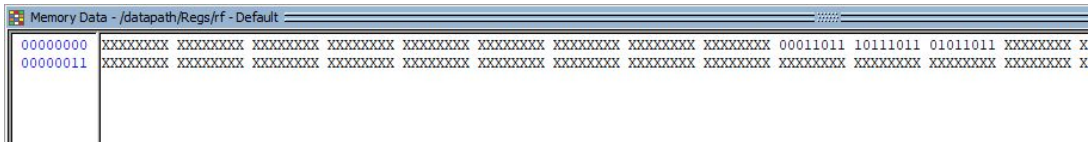


Figure 26: Register Values

For the And instruction simulation above, the instruction and $t1, $t2, $t3 was executed. Registers $t2 and $t3 were loaded with values 0x10111011 and 0x01011011 respectively. The result stored in $t1 is expected.

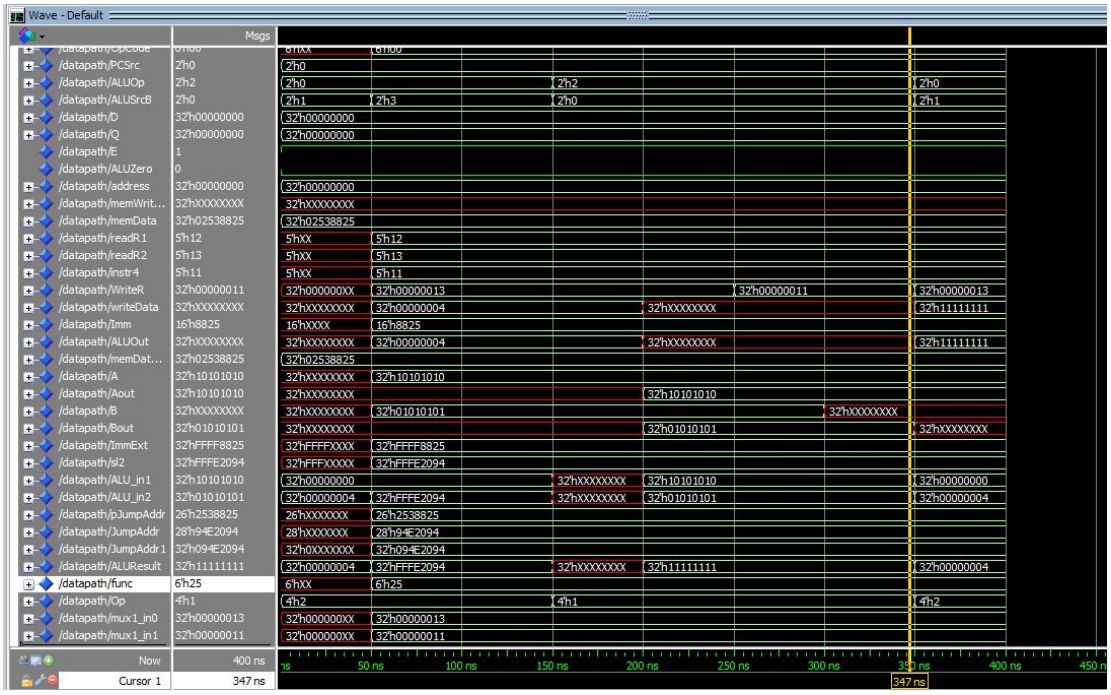


Figure 27: Or Instruction Simulation

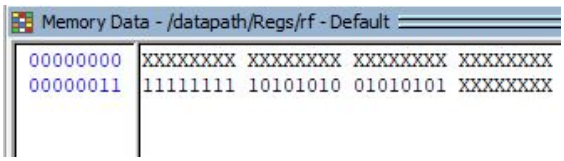


Figure 28: Register Values

For the Or instruction simulation above, the instruction or $s1, $s2, $s3 was executed. Registers $s2 and $s3 were loaded with values 0x10101010 and 0x01010101 respectively. The result stored in register $s1 is expected.

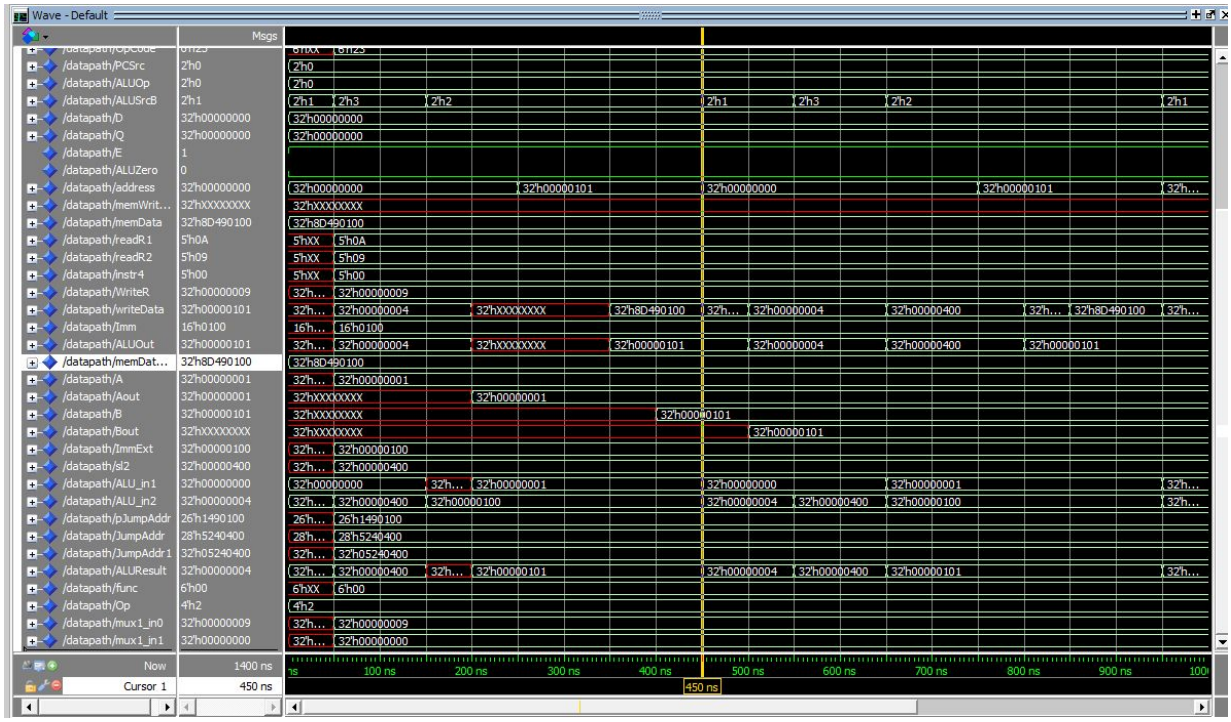


Figure 29: LW Instruction Simulation

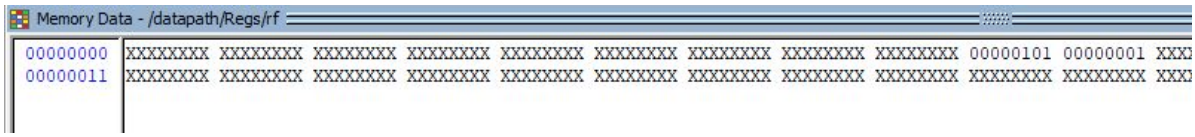


Figure 30: Register Values

For the LW simulation, the instruction lw $t1, 100($t2) was executed. Register $t2 was loaded with the value 0x00000001. The updated register values are shown above.

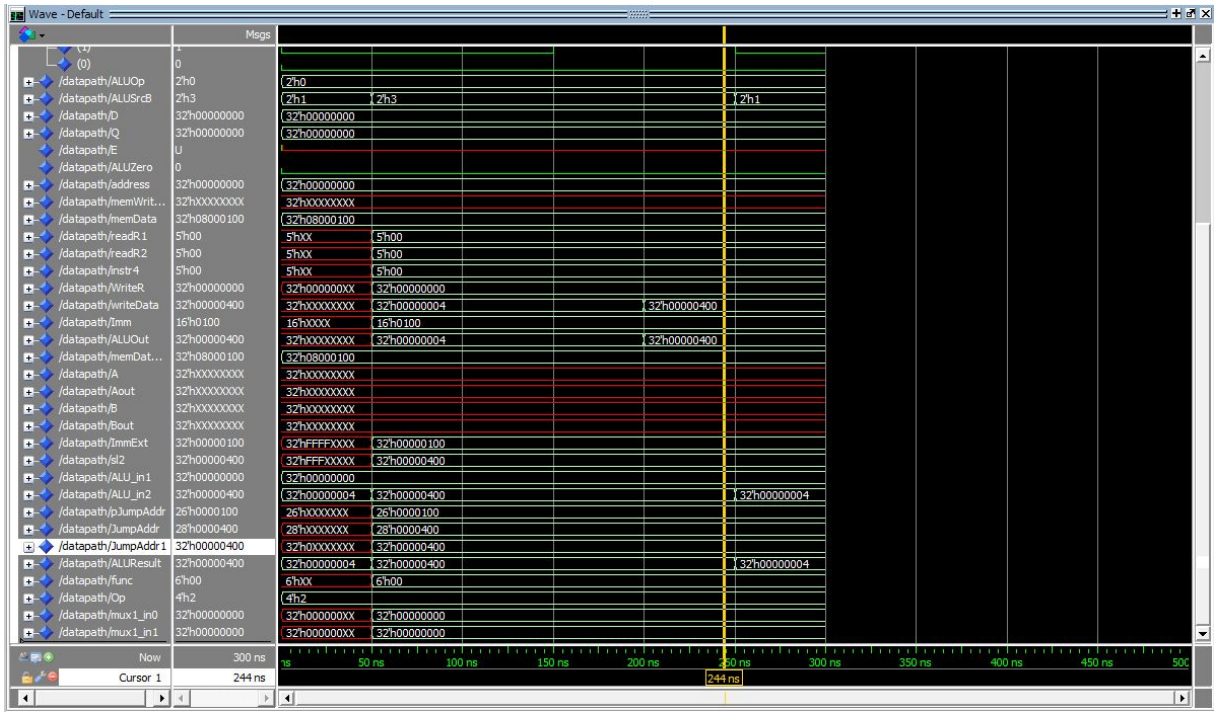


Figure 31: Jump Instruction Simulation

For the simulation above, j 100 was executed. The jump address signal shows the correct destination address.

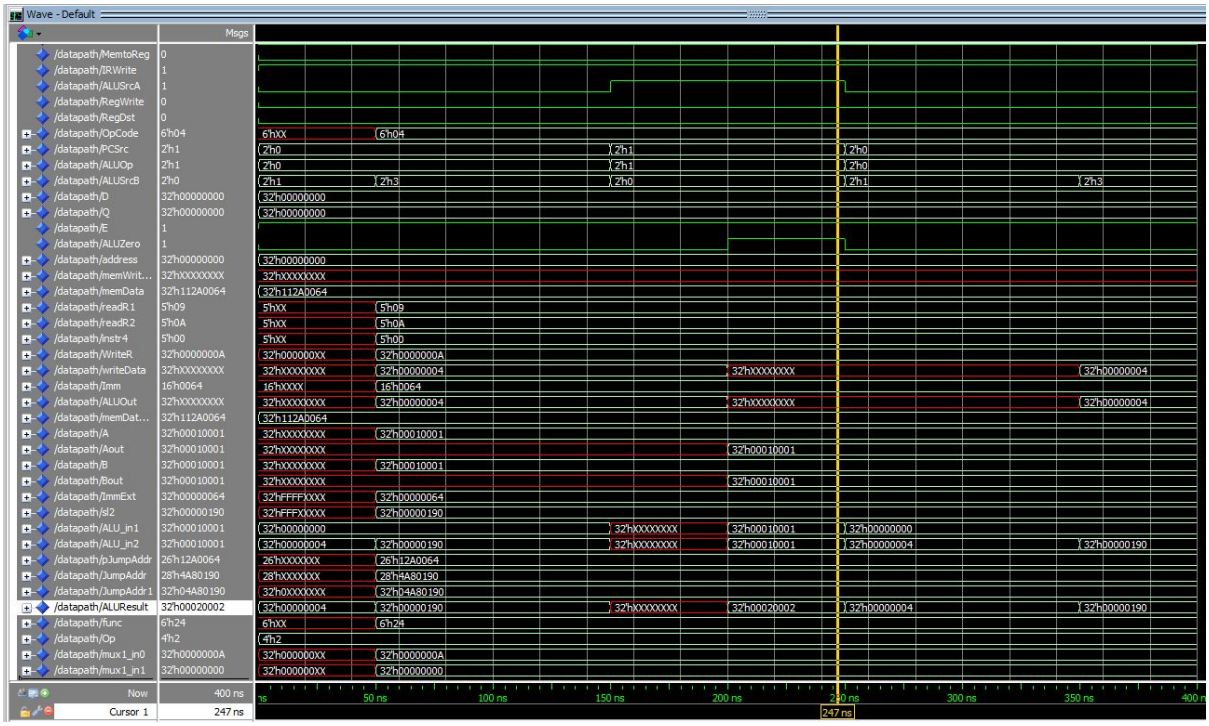


Figure 32: BEQ Instruction Simulation

For the BEQ simulation, the instruction beq $t1, $t2, 100 was executed.

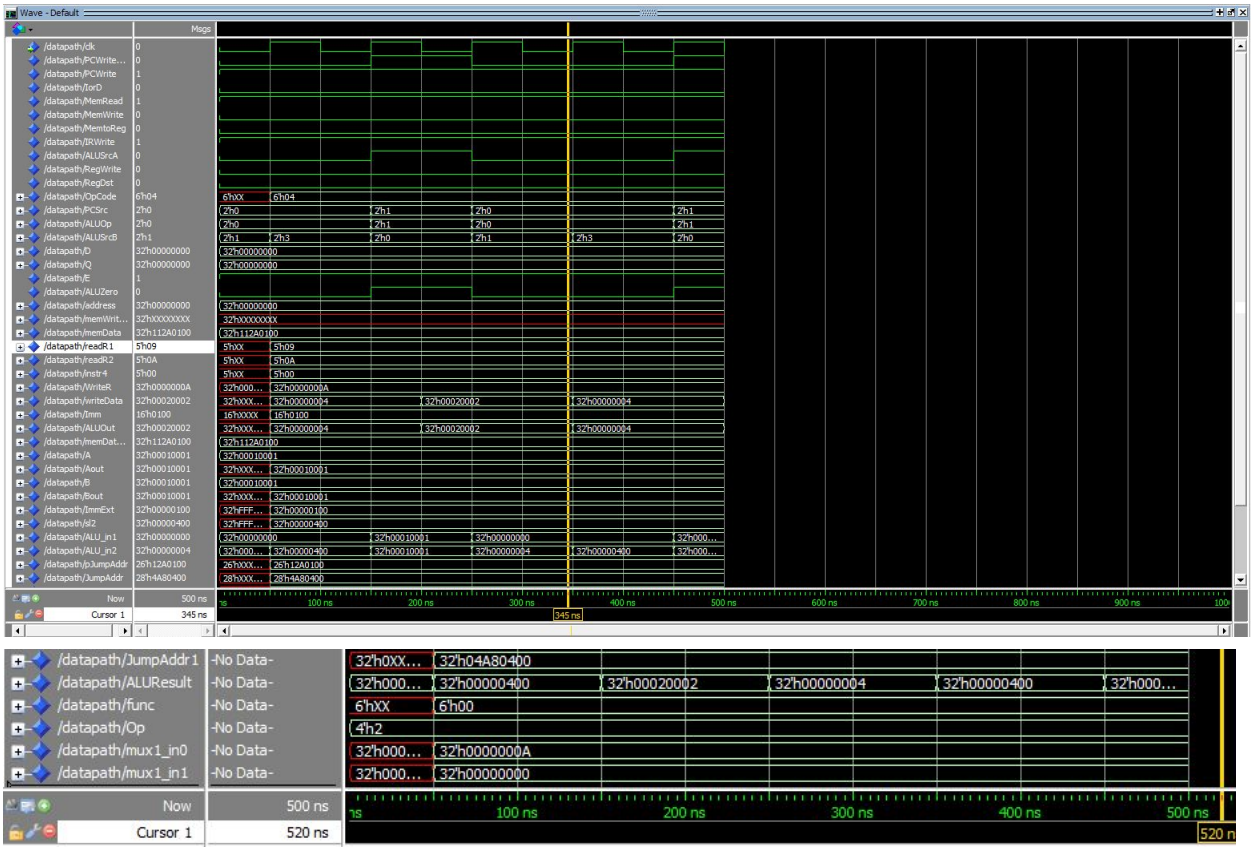


Figure 33: BEQ Simulation

For this simulation, the output was incorrect.

Loaded values: 00010001 into registers

Instr: 214C0100

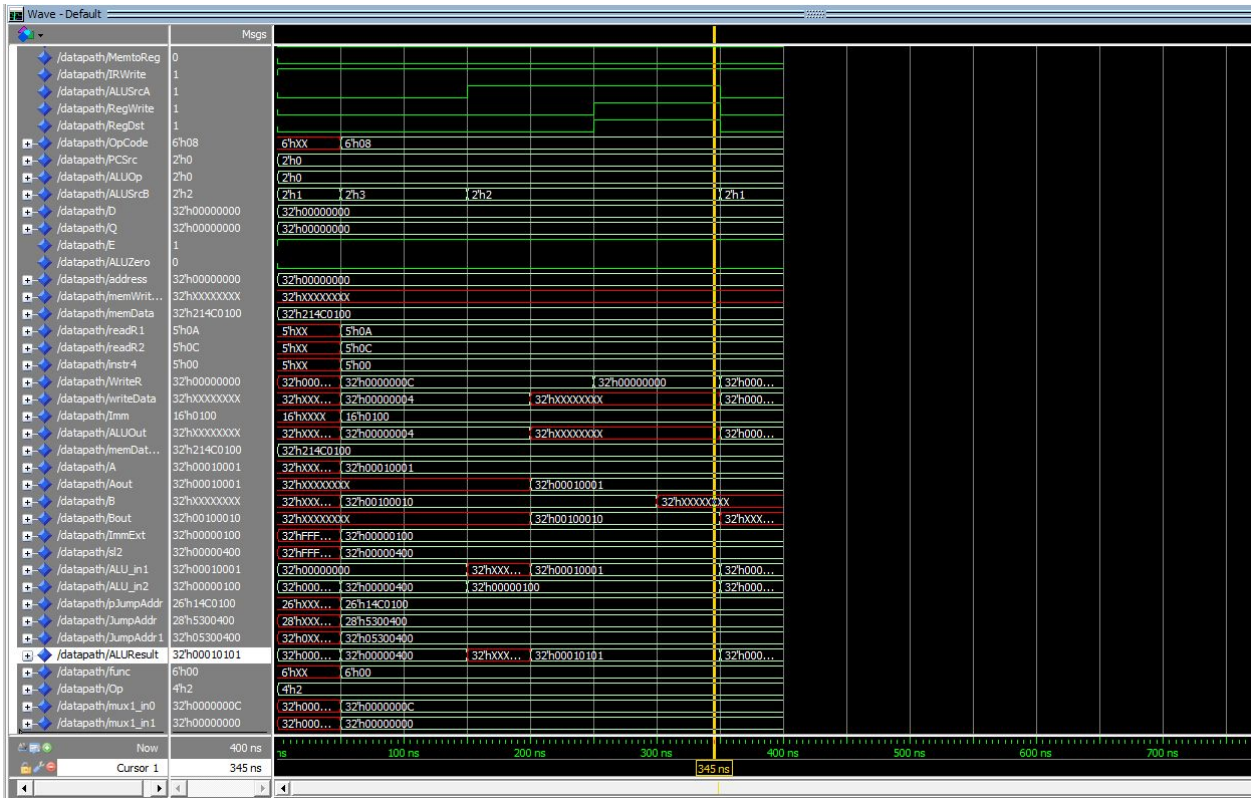
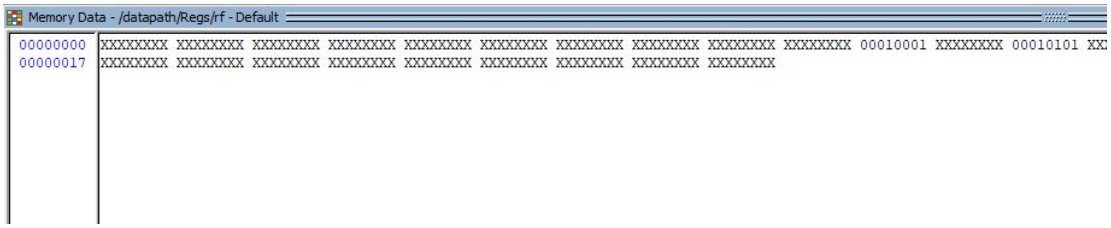


Figure 32: Addi instruction addi $t4, $t2, 100



For figure 32, the expected output was obtained when testing addi.

Discussion

Reflecting back on the design of the datapath, there are a few things that could have been improved for optimization and easier testing. Although the goal was to simplify the datapath as much as possible, the datapath turned out to be quite complex and difficult to work with.

Due to the various components that were created, keeping track of all of them could become tedious. Certain components did not have to be created. For example, sign extending and shift left 2 could have been done directly in the datapath. Being able to remove components simplifies the vhdl, making it more readable. Although creating more components is not necessarily wrong, it is not as practical when working with a large number of components during design.

In the design of the control unit, each of the instruction register were assigned appropriate variable names. Although this was meant to make the integration into the datapath simpler, in some ways it became more complicated when having to keep track of the various names. If the instructions had been kept more generalize, they could have simply been called by their bit ranges, although one would have to make sure to keep track of what bit ranges decode into what information.

Another improvement could have been variable naming. When working with a small number of components, not as good naming conventions may be easier to keep track of. But when trying to portmap a large number of components, while keeping track of signals and such, it would have helped tremendously to have more descriptive names. Often, a large chunk of time would go into debugging simply because the wrong variable was being referenced.

Overall, finding ways to simplify the design would have been very helpful. It would make the code more readable, and easier to debug, instead of staring at the code for a large amount of time trying to understand what a certain variable represents. If the code had been simpler, it might have been possible to get a fully working datapath as well, since it would have been easier to check the logic every step of the way.

V. CONCLUSION

All in all, the project was an excellent experience in utilizing all the knowledge learned over the course of the semester. Students were able to work on a very impressive project, even if it did not work perfectly. The key to complicated designs is simplifying them as much as possible. Even complex things can be made easier with the right type of thinking and design. Although the datapath did not work perfectly, the subcomponents tested correctly.

VII. REFERENCES

[1] Mentor Graphics. *ModelSim Tutorial:* Software Version 10.3a

The software used for this project was ModelSim. The tutorial broke down how to use the software, such as compiling and simulating.

[2] Peter J. Ashenden. *VHDL Tutorial*

VHDL is similar to many other programming languages. However, learning different syntax can be challenging initially. The tutorial broke down defining entities, behavioral and structural models with examples.

[3] Won Jae-Yi. *ECE 485 Fall 2019 Lecture Slides*

This project could not have been completed without the complex knowledge learned over the course of the semester. Understand the basics to the difficulty of datapaths and whatnot were essential to completing this project.

[4] *Tutorial - Using Modelsim for Simulation, for Beginners.* http://www.nanland.com
Although the VHDL tutorial provided information on syntax and concepts, it did not contain much information regarding writing testbench code. This tutorial walks through an example of automating testing with an AND gate, making it easy to translate into this project.

VIII. PROJECT CONTRIBUTIONS

1.) Bella

I worked on the 4-input mux, instruction register, control unit, sign extend and temporary registers, writing both the source and testbench code. Leah and I both worked on the datapath together.

I did the final simulations for the datapath, worked on debugging the system and sometimes its wrongly coded components. I also helped with writing the system design and describing the screenshots for which I'm responsible.

2.) Leah

I was responsible for implementing the 2-input mux, memory, ALU, Shift left 2, reg, and memDataReg. I wrote the source code and the testbench code for them. I performed test simulations and took screenshots. My teammate and I, Bella, both worked on the datapath code together.

I then worked on the abstract, introduction, system design, and conclusion. Bella and I both worked on the simulation results and I did the discussion. We both worked on the descriptions of each screenshot. I also completed the list of references.

APPENDIX

- **2-input Mux Code**

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


--- A and B are the inputs to the mux. Y is the output. S is the select bit
entity input_mux_2 is
   Port(    A:     in STD_LOGIC_VECTOR(31 downto 0);
            B:    in STD_LOGIC_VECTOR(31 downto 0);
```

```
        S:    in STD_LOGIC;
        Y:    out STD_LOGIC_VECTOR(31 downto 0));
end input_mux_2;


--- If S == 0, select A. Else, select B
architecture Behavioral of input_mux_2 is
begin

   Y <= A when S = '0' else B;

end Behavioral;
```

- **2-input Mux Test Bench Code**

```
use IEEE.STD_LOGIC_1164.ALL;

entity input_mux_2_tb is
end input_mux_2_tb;

--- define signals which will contain desired input values. Ouput values remain empty.
architecture Behavioral of input_mux_2_tb is
   signal A_SIG    : STD_LOGIC_VECTOR(31 downto 0);
   signal B_SIG    : STD_LOGIC_VECTOR(31 downto 0);
   signal S_SIG : STD_LOGIC;
   signal Y_SIG    : STD_LOGIC_VECTOR(31 downto 0);

component input_mux_2 is
   Port (A:    in STD_LOGIC_VECTOR(31 downto 0);
         B:    in STD_LOGIC_VECTOR(31 downto 0);
         S:    in STD_LOGIC;
         Y:    out STD_LOGIC_VECTOR(31 downto 0));

end component;

begin
mux2_INST : input_mux_2
   port map (
         A => A_SIG,
         B => B_SIG,
         S => S_SIG,
         Y => Y_SIG);

process is
begin
   A_SIG <= x"00001234";
   B_SIG <= x"00000A00";
   S_SIG <= '0';

   wait for 10 ns;

   A_SIG <= x"00001234";
   B_SIG <= x"00000A00";
   S_SIG <= '1';
```

```
   wait for 10 ns;

end process;

end Behavioral;
```

- **4-input Mux Code**

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

-- instantiation of 4 input MUX; S: select signal
entity input_mux_4 is
        Port(   A,B,C,D:        in STD_LOGIC_VECTOR(31 downto 0);
                S:              in STD_LOGIC_VECTOR(1 downto 0);
                Y:              out STD_LOGIC_VECTOR(31 downto 0));
end input_mux_4;

--- If S == 00, select A else S= 01 select B, S= 10 Select C; S=11 Select D
architecture behaviour of input_mux_4 is
begin
        Y <= A when S = "00" else
            B when S = "01" else
            C when S = "10" else
            D;

end behaviour;
```

- **4-input Mux Test Bench Code**

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

entity tb_4input_mux is
end tb_4input_mux;

architecture test of tb_4input_mux is
        component input_mux_4 is
                port( A,B,C,D:  in STD_LOGIC_VECTOR(31 downto 0);
                S:              in STD_LOGIC_VECTOR(1 downto 0);
                Y:              out STD_LOGIC_VECTOR(31 downto 0));
         end component;

        for U1: input_mux_4 use entity WORK.input_mux_4(behaviour);

        --INPUTS
        signal A_s, B_s, C_s, D_s: std_logic_vector(31 downto 0);
        signal S_s: std_logic_vector(1 downto 0);
        --OUTPUT
        signal Y_s: std_logic_vector(31 downto 0);
```

```
	begin

	U1: input_mux_4 port map (A_s, B_s, C_s, D_s, S_s, Y_s);
		process
		begin

		A_s <= "00000000000000000000000000000000";
		B_s <= "00010001000100010001000100010001";
		C_s <= "00110011001100110011001100110011";
		D_s <= "01110111011101110111011101110111";

		-- Test Case 1
			S_s <= "00";
			wait for 10 ns;

		-- Test Case 2
			S_s <= "01";
			wait for 10 ns;

		-- Test Case 3
			S_s <= "10";
			wait for 10 ns;

		-- Test Case 4
			S_s <= "11";
			wait for 10 ns;

		end process;
end test;
```

- **ALU Code**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.all;


-- define inputs A and B to be 32 bits
-- input ALUop is 4 bits
-- output result is 32 bits
entity alu32 is
	Port( A: in STD_LOGIC_VECTOR(31 downto 0);
	    B: in STD_LOGIC_VECTOR(31 downto 0);
	    ALUop: in STD_LOGIC_VECTOR(3 downto 0);
	    zero: out STD_LOGIC;
	    result: out STD_LOGIC_VECTOR(31 downto 0));
```

```
end alu32;

architecture struct of alu32 is
	component adder_32 is
		Port (A:		in STD_LOGIC_VECTOR(31 downto 0);
		    B:		in STD_LOGIC_VECTOR(31 downto 0);
		    carryin:	in STD_LOGIC;
		    sum:		out STD_LOGIC_VECTOR(31 downto 0);
		    carryout:	out STD_LOGIC);
end component;

	signal add_result: STD_LOGIC_VECTOR(31 downto 0);
	signal sub_result: STD_LOGIC_VECTOR(31 downto 0);
	signal neg_B:	STD_LOGIC_VECTOR(31 downto 0);

begin

	neg_B <= std_logic_vector(unsigned(not B)+1);

	ADD: adder_32 port map(A => A, B => B, carryin => '0', sum => add_result, carryout => open);
	SUB: adder_32 port map(A, neg_B, '0', sub_result, open);

	result <=A and B when(ALUop = "0000") else
			A or B when(ALUop = "0001") else
			add_result when(ALUop = "0010") else
			A xor B when(ALUop = "0011") else
			sub_result when(ALUop = "0110") ;

	zero <=		'1' when unsigned(sub_result) = 0 else --if A=B, sub_result = 0
			'0';


end struct;
```

- **ALU Code Test Bench**

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

entity alu32_tb is
end alu32_tb;

architecture Behavioral of alu32_tb is
	signal A_SIG: STD_LOGIC_VECTOR(31 downto 0);
	signal B_SIG: STD_LOGIC_VECTOR(31 downto 0);
	signal ALUop_SIG: STD_LOGIC_VECTOR(3 downto 0);
	signal zero_SIG: STD_LOGIC;
	signal result_SIG: STD_LOGIC_VECTOR(31 downto 0);
```

```
component alu32 is
	Port( A: in STD_LOGIC_VECTOR(31 downto 0);
	    B: in STD_LOGIC_VECTOR(31 downto 0);
	    ALUop: in STD_LOGIC_VECTOR(3 downto 0);
	    zero: out STD_LOGIC;
	    result: out STD_LOGIC_VECTOR(31 downto 0)
	);

end component;

begin alu32_INST: alu32
	port map(
		A => A_SIG,
		B => B_SIG,
		ALUop => ALUop_SIG,
		zero => zero_SIG,
		result => result_SIG);

process is
begin
	-- add
	A_SIG <= x"00001234";
	B_SIG <= x"00000004";
	ALUop_SIG <= "0010";
	wait for 10ns;

	--sub
	A_SIG <= x"00001234";
	B_SIG <= x"00000004";
	ALUop_SIG <= "0110";
	wait for 10ns;
	-- and
	A_SIG <= x"00001234";
	B_SIG <= x"00000004";
	ALUop_SIG <= "0000";
	wait for 10ns;

	--or
	A_SIG <= x"00001234";
	B_SIG <= x"00000004";
	ALUop_SIG <= "0001";
	wait for 10ns;

	-- addi
	A_SIG <= x"00001234";
	B_SIG <= x"00000004";
	ALUop_SIG <= "0011";
```

```
        wait for 10ns;


end process;
end Behavioral;
```

- ## ALU Control Code

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

entity aluControl is
        Port(   func:           in STD_LOGIC_VECTOR(5 downto 0);
                ALUOp:          in STD_LOGIC_VECTOR(1 downto 0);
                Op:             out STD_LOGIC_VECTOR(3 downto 0));
end aluControl;

architecture behavior of aluControl is
begin
        Op <= "0010" when ALUOp = "10" and func = "100000" else
                "0110" when ALUOp = "10" and func = "100010" else
                "0000" when ALUOp = "10" and func = "100100" else
                "0001" when ALUOp = "10" and func = "100101" else
                "0000";
end behavior;
```

- ## Control Unit Code

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

-- instantiation of 4 input MUX; S: select signal
entity controlUnit is
--generic(ClkFreqHz : integer);
        Port(   clk:        in std_logic;
                OpCode:         in STD_LOGIC_VECTOR(5 downto 0);
                PCWriteCond: out STD_LOGIC;
                PCWrite: out STD_LOGIC;
                IorD: out STD_LOGIC;
                MemRead: out STD_LOGIC;
                MemWrite: out STD_LOGIC;
                MemtoReg: out STD_LOGIC;
                IRWrite: out STD_LOGIC;
                ALUSrcA: out STD_LOGIC;
                RegWrite: out STD_LOGIC;
                RegDst: out STD_LOGIC;
                PCSrc: out STD_LOGIC_VECTOR(1 downto 0);
                ALUOp: out STD_LOGIC_VECTOR(1 downto 0);
                ALUSrcB: out STD_LOGIC_VECTOR(1 downto 0));
```

```
end controlUnit;

architecture behaviour of controlUnit is
	type control_state is ( s0, s1, s2, s3, s4, s5, s6, s7, s8, s9);
	signal state: control_state := s0;
	signal nextstate: control_state;

begin
	Process1: process (clk)
	begin
		if (rising_edge(clk)) then
			state <= nextstate;
		else
			null;
		end if;
	end process;

	Process2: process (state, nextstate, OpCode)
	begin
		nextstate <= state; --when no case is satisfied

		case state is
		when s0 =>
			ALUSrcA <= '0';
			ALUSrcB <= "01";
			ALUOp <= "00" ;
			RegWrite <= '1';
			RegDst <= '1';
			MemtoReg <= '0';
			MemRead<= '1';
			MemWrite <= '0';
			IRWrite <= '1';
			IorD <= '0';
			PCWriteCond <= '0';
			PCWrite <= '1';
			PCSrc <= "00";
			nextstate <= s1;

		when s1 =>
			ALUSrcA <= '0';
			ALUSrcB <= "01";
			ALUOp <= "00" ;
			if (Opcode = "000000") then		-- R-type
				nextstate <= s6;
			elsif (Opcode = "100011") then	-- LW
				nextstate <= s2;
			elsif (OpCode = "101011") then -- SW
				nextstate <= s2;
```

```
            elsif (Opcode = "000100") then   -- BEQ
                        nextstate <= s8;
            elsif (Opcode = "000010") then   -- Jump
                        nextstate <= s9;
            else                                        -- addi
                        nextstate <= s2;
            end if;

when s2 =>
            ALUSrcA <= '0';
            ALUSrcB <= "01";
            ALUOp <= "00" ;
            if (Opcode = "100011") then           -- LW
                        nextstate <= s3;
            else
                        nextstate <= s5;    --SW
            end if;
when s3 =>
            MemRead <= '1';
            IorD <= '1';
            nextstate <= s4;
when s4 =>
            RegDst <= '0';
            RegWrite<= '1';
            MemtoReg<= '1';
            nextstate <= s0;
when s5 =>
            MemWrite <= '1';
            IorD <= '1';
            nextstate <= s0;

when s6 =>
            ALUSrcA <= '1';
            ALUSrcB <= "00";
            ALUOp <= "10";
            nextstate <= s7;

when s7 =>
            RegWrite <= '1';
            RegDst <= '1';
            MemtoReg <= '0';
            nextstate <= s0;
when s8 =>
            ALUSrcA <= '1';
            ALUSrcB <= "00";
            ALUOp <= "01" ;
            PCWriteCond <= '1'; --if ALUzero asserted
            PCSrc <= "01";
```

```
                nextstate <= s0;
            when s9 =>
                PCWrite <= '1';
                PCSrc <= "10";
                nextstate <= s0;

            end case;
        end process;

end architecture;
```

- **Control Unit Test Bench Code**

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

entity tb_controlUnit is
end tb_controlUnit;

architecture test of tb_controlUnit is
        component controlUnit is
            port(   clk:        in std_logic;
                OpCode:             in STD_LOGIC_VECTOR(5 downto 0);
                PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, ALUSrcA,
RegWrite, RegDst: out STD_LOGIC;
                PCSrc, ALUOp, ALUSrcB: out STD_LOGIC_VECTOR(1 downto 0));
        end component;


        for U1: controlUnit use entity WORK.controlUnit(behaviour);

        --INPUT
        signal OpCode_s: STD_LOGIC_VECTOR(5 downto 0);
        signal clk_s: std_logic := '0';
        constant clk_period : time := 20 ns;
        constant high_time: time:= clk_period/2;
        constant low_time: time:= clk_period - high_time;
        --OUTPUTs
        signal PCWriteCond_s, PCWrite_s, IorD_s, MemRead_s, MemWrite_s, MemtoReg_s, IRWrite_s,
ALUSrcA_s, RegWrite_s, RegDst_s: STD_LOGIC;
        signal PCSrc, ALUOp, ALUSrcB: STD_LOGIC_VECTOR(1 downto 0);

        begin
        U1: controlUnit port map (clk_s, OpCode_s, PCWriteCond_s, PCWrite_s, IorD_s, MemRead_s,
MemWrite_s, MemtoReg_s, IRWrite_s, ALUSrcA_s, RegWrite_s, RegDst_s, PCSrc, ALUOp, ALUSrcB);
            process
            begin
```

```
		-- Test Case 1
				OpCode_s <= "000100"; --
				clk_s <= '1';
				wait for high_time;
				clk_s <= '0';
				wait for low_time;
		--Test Case 2
			--OpCode_s <= "100011"; --LW
			--clk_s <= '0';
			--reset_s <= '0';
			--wait for clk_period/10;
			--clk_s <= '1';
			--wait for clk_period/10;
		--Test Case 4
			--OpCode_s <= "000010";
			--clk_s <= '0';
			--reset_s <= '0';
			--wait for 10ns;
		--Test Case 5
			--OpCode_s <= "001000";
			--clk_s <= '1';
			--reset_s <= '0';
			--wait for 10ns;
		--Test Case 6
			--OpCode_s <= "011000";
			--clk_s <= '1';
			--reset_s <= '0';
			--wait for 10ns;

		end process;
end test;
```

- **Instruction Register Code**

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

entity instrRegister is
   Port(
	clk:	in std_logic;
	MemData:	in STD_LOGIC_VECTOR(31 downto 0);
	IRWrite:	in STD_LOGIC;
	OpCode:		out STD_LOGIC_VECTOR(5 downto 0);
	Func:	out STD_LOGIC_VECTOR(5 downto 0);
	Rs:	out STD_LOGIC_VECTOR(4 downto 0);
	Rt:	out STD_LOGIC_VECTOR(4 downto 0);
```

```
            Rd:       out STD_LOGIC_VECTOR(4 downto 0);
            Imm:      out STD_LOGIC_VECTOR(15 downto 0);
            JumpAddr:    out STD_LOGIC_VECTOR(25 downto 0));

end instrRegister;

architecture behaviour of instrRegister is
begin

   process(Memdata, IRWrite, clk)
   begin
            if (IRWrite= '1') and (rising_edge(clk)) then
                        OpCode <= MemData(31 downto 26); --opcode
                        Rs <= Memdata (25 downto 21); --rs
                                    Rt <= Memdata (20 downto 16); --rt
                        Rd <= Memdata (15 downto 11); --rd
                        Imm <= Memdata (15 downto 0);  -- imm
                        Func <= Memdata (5 downto 0);   --func
                        JumpAddr <= Memdata (25 downto 0);  --jump
            end if;
   end process;
end behaviour;
```

- **Instruction Register Testbench Code**

```
use ieee.std_logic_1164.all;

entity tb_instrRegister is
end tb_instrRegister;

architecture test of tb_instrRegister is
   component instrRegister is
            port(
-- figure out how to integrate clk
            clk:        in std_logic;
            MemData:    in STD_LOGIC_VECTOR(31 downto 0);
            IRWrite:    in STD_LOGIC;
            OpCode, Func:    out STD_LOGIC_VECTOR(5 downto 0);
            Rs, Rt, Rd:    out STD_LOGIC_VECTOR(4 downto 0);
            Imm:      out STD_LOGIC_VECTOR(15 downto 0);
            JumpAddr:    out STD_LOGIC_VECTOR(25 downto 0));
    end component;

   for U1: instrRegister use entity WORK.instrRegister(behaviour);

   --INPUTS
   signal clk_s:       std_logic := '0';
   signal MemData_s: std_logic_vector(31 downto 0);
```

```
   signal IRWrite_s: std_logic;
   constant clk_period : time := 20 ns;
   constant high_time: time:= clk_period/2;
   --OUTPUT
   signal OpCode_s, Func_s: std_logic_vector(5 downto 0);
   signal Rs_s, Rt_s, Rd_s: std_logic_vector(4 downto 0);
   signal Imm_s: std_logic_vector(15 downto 0);
   signal JumpAddr_s: std_logic_vector(25 downto 0);

   begin
   clk_s <= not clk_s after high_time;
   U1: instrRegister port map ( clk_s, MemData_s, IRWrite_s, OpCode_s, Func_s, Rs_s, Rt_s, Rd_s, Imm_s,
JumpAddr_s);
            process
            begin

            -- Test Case 1
                     IRWrite_s <= '1';
                     MemData_s <= x"01578820";     --add $s1, $t2, $s7
                     wait for clk_period;
            -- Test Case 2
                     IRWrite_s <= '1';
                     MemData_s <= x"8D310012";     -- lw $s1, 12($t1)
                     wait for clk_period;

            end process;
end test;
```

- **MemDataReg Code**

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

-- entity declaration. memData is the input from memory component. In/out is 32 bits.
entity memDataReg is
            Port ( memData:  in STD_LOGIC_VECTOR(31 downto 0);
                        memDataOut: out STD_LOGIC_VECTOR(31 downto 0));

end entity memDataReg;

-- input becomes the output
architecture behavior of memDataReg is
begin
            memDataOut <= memData;

end architecture behavior;
```

- **MemDataReg TestBench Code**

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

entity memDataReg_tb is
end memDataReg_tb;

--- define signals which will contain desired input values. Output remains empty
architecture Behavioral of memDataReg_tb is
	signal memData_SIG:	STD_LOGIC_VECTOR(31 downto 0);
	signal memDataOut_SIG:	STD_LOGIC_VECTOR(31 downto 0);

component memDataReg is
	Port ( memData:  in STD_LOGIC_VECTOR(31 downto 0);
		memDataOut: out STD_LOGIC_VECTOR(31 downto 0));
end component;

begin memDataReg_INST: memDataReg
	port map (
		memData => memData_SIG,
		memDataOut => memDataOut_SIG);

process is
begin
	memData_SIG <= x"00001234";
	wait for 10ns;

	memData_SIG <= x"0000A000";
	wait for 10ns;

end process;
end Behavioral;
```

- **Memory Code**

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


-- memory entity declaration. memRead/memWrite are 1 bit control signals
entity memory is
	Port ( address:	in STD_LOGIC_VECTOR(31 downto 0);
		memWriteIn: in STD_LOGIC_VECTOR(31 downto 0);
		memRead:	in STD_LOGIC;
		memWrite:	in STD_LOGIC;
		memData:	out STD_LOGIC_VECTOR(31 downto 0));
end entity memory;
```

```
architecture behavior of memory is
type memory_type is array(0 to 511) of STD_LOGIC_VECTOR(31 downto 0);  -- 2^9 because much bigger would
crash
signal memory: memory_type;
begin
-- mem write responsible for writing to desired address if signal is 1
	MEM_WRITE: process(memWrite)
	begin
		if(memWrite = '1') then
			memory(conv_integer(Address)) <= memWriteIn;
		end if;
	end process;

-- mem read responsible for reading data from desired address and placing in memOut
	MEM_READ: process(memRead)
	begin
		if(memRead = '1') then
			memData <= memory(conv_integer(Address));
		end if;
	end process;


end architecture behavior;
```

- **Memory Code TestBench**

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

entity memory_tb is
end memory_tb;

--- define signals which will contain desired input values. Output remains empty
architecture Behavioral of memory_tb is
	signal address_SIG:	STD_LOGIC_VECTOR(31 downto 0);
	signal mIn_SIG:  STD_LOGIC_VECTOR(31 downto 0);
	signal mRead_SIG:	STD_LOGIC;
	signal mWrite_SIG:	STD_LOGIC;
	signal mData_SIG:	STD_LOGIC_VECTOR(31 downto 0);

component memory is
	Port (  address:    in STD_LOGIC_VECTOR(31 downto 0);
		memWriteIn: in STD_LOGIC_VECTOR(31 downto 0);
		memRead:	in STD_LOGIC;
		memWrite:	in STD_LOGIC;
		memData:	out STD_LOGIC_VECTOR(31 downto 0));
```

```
end component;

begin
mem_INST: memory
        port map (
                address => address_SIG,
                memWriteIn => mIn_SIG,
                memRead => mRead_SIG,
                memWrite => mWrite_SIG,
                memData => mData_SIG);
process is
begin
                -- read to confirm that nothing is in address
                address_SIG <= x"00000008";
                mIn_SIG <= x"12345678";
                mRead_SIG <= '1';
                mWrite_SIG <= '0';

                wait for 10ns;

                -- write into address
                address_SIG <= x"00000008";
                mIn_SIG <= x"12345678";
                mRead_SIG <= '0';
                mWrite_SIG <= '1';

                wait for 10ns;

                -- read into address to confirm writing worked in next 10 ns

end process;
end Behavioral;
```

- **Register File Code**

```
-- library declaration
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--Register File entity declaration
entity reg is
        port (
                clk : in std_logic;
                R1 : in std_logic_vector (4 downto 0);
                R2 : in std_logic_vector (4 downto 0);
                WriteReg : in std_logic_vector (4 downto 0);
                WriteData : in std_logic_vector (31 downto 0);
```

```
		RegWrite: in std_logic;
		ReadData1 : out std_logic_vector (31 downto 0);
		ReadData2 : out std_logic_vector (31 downto 0)
	);
end entity reg;

--Behavioral Model. For array use 32*32 size
architecture behavior of reg is
type r_type is array (0 to 31) of std_logic_vector (31 downto 0);
signal r: r_type;

begin
	Write2Reg: process(RegWrite) -- Write to register
	begin
		if (RegWrite = '1') then
			r(conv_integer(WriteReg)) <=  WriteData;
		end if;
	end process;

	ReadR1: process(clk, R1) -- Read register number 1
	begin
		ReadData1 <= r(conv_integer(R1));
	end process;

	ReadR2: process(clk, R2) -- Read register number 2
	begin
		ReadData2 <= r(conv_integer(R2));
	end process;
end architecture behavior;
```

- **Reg Code**

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

entity r is
	Port (	enable: in STD_LOGIC;
		D: in STD_LOGIC_VECTOR(31 downto 0);
		Q: out STD_LOGIC_VECTOR(31 downto 0));

end r;

architecture Behavioral of r is
begin
	Q <= D when enable = '1';

end Behavioral;
```

- **Temporary Register Code**

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

entity tmpReg is
   Port( clk :    in std_logic;
         readData: in STD_LOGIC_VECTOR(31 downto 0);
         outData:  out STD_LOGIC_VECTOR(31 downto 0));

end tmpReg;

architecture behaviour of tmpReg is

   constant clk_period : time := 100 ns;
   constant high_time: time:= clk_period/2;

begin
   process
   begin
         wait for high_time;
         outData <= readData;
         wait for clk_period;
   end process
```

- **Temporary Register Testbench**

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

entity tb_tmpReg is
end tb_tmpReg;

architecture test of tb_tmpReg is
   component tmpReg is
         port(     clk :    in std_logic;
                   readData: in STD_LOGIC_VECTOR(31 downto 0);
                            outData:  out STD_LOGIC_VECTOR(31 downto 0));
    end component;

   for U1: tmpReg use entity WORK.tmpReg(behaviour);

   signal readData_s:  std_logic_vector(31 downto 0);
   signal outData_s:   std_logic_vector(31 downto 0);
   signal clk_s:     std_logic := '0';
   constant clk_period : time := 100 ns;
   constant high_time: time:= clk_period/2;

   begin
```

```
   clk_s <= not clk_s after high_time;
   U1: tmpReg port map (clk_s, readData_s, outData_s);
          process
          begin
                -- Test Case 1
                readData_s <= x"01578820";
                wait for clk_period;
                readData_s <= x"00000000";
                wait for clk_period;
          end process;
end test;
```

- **Shift Left 2 Code**

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

-- shift left 2 takes in an input of 32 bits and returns an output of 32 bits
entity shift_left2 is
          Port ( input:         in STD_LOGIC_VECTOR(31 downto 0);
               output:  out STD_LOGIC_VECTOR(31 downto 0));
end entity shift_left2;

architecture Behavioral of shift_left2 is
begin
          output <= input (29 downto 0) & "00";

end architecture Behavioral;
```

- **Shift Left 2 Test Bench Code**

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

entity shift_left2_tb is
end shift_left2_tb;

architecture Behavioral of shift_left2_tb is
          signal input_SIG  : STD_LOGIC_VECTOR(31 downto 0);
          signal output_SIG              : STD_LOGIC_VECTOR(31 downto 0);

component shift_left2 is
          Port ( input:         in STD_LOGIC_VECTOR(31 downto 0);
               output:  out STD_LOGIC_VECTOR(31 downto 0));
end component;

begin sl2_INST: shift_left2
```

```
	port map (
		input => input_SIG,
		output => output_SIG);

process is
begin
	input_SIG <= x"00001234";

	wait for 10ns;

	input_SIG <= x"ABCD0000";

	wait for 10ns;

end process;
end Behavioral;
```

- **Shift Left Jump Code**

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

-- shift left 2 takes in an input of 32 bits and returns an output of 32 bits
entity shiftleft_j is
	Port ( input:	in STD_LOGIC_VECTOR(25 downto 0);
	       output:  out STD_LOGIC_VECTOR(27 downto 0));
end entity shiftleft_j;

architecture Behavioral of shiftleft_j is
begin
	output <= input (25 downto 0) & "00";

end architecture Behavioral;
```

- **Sign Extend Code**

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

-- instantiation of 4 input MUX; S: select signal
entity signExt is
	Port(	Imm:		in STD_LOGIC_VECTOR(15 downto 0);
		ImmExt:		out STD_LOGIC_VECTOR(31 downto 0));
end signExt;

architecture behaviour of signExt is
begin
process(Imm)
begin
	if (Imm(15)= '0') then
```

```
            ImmExt(31 downto 16) <= "0000000000000000";
            ImmExt(15 downto 0)<= Imm;
        else
            ImmExt(31 downto 16) <= "1111111111111111";
            ImmExt(15 downto 0)<= Imm;
        end if;

end process;

end behaviour;
```

- **Sign Extend Test Bench Code**

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

entity tb_signExt is
end tb_signExt;

architecture test of tb_SignExt is
        component signExt is
            port(   Imm:        in STD_LOGIC_VECTOR(15 downto 0);
                    ImmExt:     out STD_LOGIC_VECTOR(31 downto 0));
        end component;

        for U1: signExt use entity WORK.signExt(behaviour);

        --INPUTS
        signal Imm_s: std_logic_vector(15 downto 0);
        --OUTPUT
        signal ImmExt_s: std_logic_vector(31 downto 0);

        begin
        U1: signExt port map (Imm_s, ImmExt_s);
            process
            begin

            -- Test Case 1
                Imm_s <= "0001000100010001";
                wait for 10 ns;
            -- Test Case 2
                Imm_s <= "1000100010001000";
                wait for 10 ns;

            end process;

end test;
```

- **1-Bit Full-Adder Source Code:**

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

-- one bit adder, defines intputs and outputs. Each is 1 bit
entity adder is
	Port (	A:		in STD_LOGIC;
		B:		in STD_LOGIC;
		carryin:	in STD_LOGIC;
		sum:		out STD_LOGIC;
		carryout:	out STD_LOGIC);
end adder;

-- define what adder will do. Behavioral. Based off gate logic.
architecture Behavioral of adder is
begin
	sum <= A xor B xor carryin;

	carryout <= ((A xor B) and carryin) or (A and B);
end Behavioral;
```

- **4-Bit Full-Adder Source Code:**

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

-- one bit adder, defines intputs and outputs. A, B, and sum will be 4 bits. Carryin/out remain as 1 bit.
entity adder_4 is
	Port (	A:		in STD_LOGIC_VECTOR(3 downto 0);
		B:		in STD_LOGIC_VECTOR(3 downto 0);
		carryin:	in STD_LOGIC;
		sum:		out STD_LOGIC_VECTOR(3 downto 0);
		carryout:	out STD_LOGIC);
end adder_4;

architecture struct of adder_4 is
	-- Component instantiation list. Utilize 1-bit adder to design 4-bit adder.
	component adder is
		Port( A:	in STD_LOGIC;
		      B:	in STD_LOGIC;
		      carryin:		in STD_LOGIC;
		      sum:	out STD_LOGIC;
		      carryout:	out STD_LOGIC);
end component;
--3 signals b/c carryout of first 1-bit adder is the carryout of the second 1-bit adder and so on
	signal C1, C2, C3: STD_LOGIC;

begin
-- map hardware components. Structural Model.
```

```
	Add_0: adder port map(A(0), B(0), carryin, sum(0), C1);
	Add_1: adder port map(A(1), B(1), C1, sum(1), C2);
	Add_2: adder port map(A(2), B(2), C2, sum(2), C3);
	Add_3: adder port map(A(3), B(3), C3, sum(3), carryout);
end struct;
```

- **32-Bit Full-Adder Source Code:**

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

-- one bit adder, defines intputs and outputs. A, B, and sum will be 32 bits. Carryin/out remain as 1.
entity adder_32 is
	Port (	A:		in STD_LOGIC_VECTOR(31 downto 0);
		B:		in STD_LOGIC_VECTOR(31 downto 0);
		carryin:	in STD_LOGIC;
		sum:		out STD_LOGIC_VECTOR(31 downto 0);
		carryout:	out STD_LOGIC);
end adder_32;

architecture struct of adder_32 is
	-- Component instantiation list. Utilize 4-bit adders.
	component adder_4 is
		Port (	A:		in STD_LOGIC_VECTOR(3 downto 0);
			B:		in STD_LOGIC_VECTOR(3 downto 0);
			carryin:	in STD_LOGIC;
			sum:		out STD_LOGIC_VECTOR(3 downto 0);
			carryout:	out STD_LOGIC);
	end component;
-- use 7 signals b/c carryout of first 4-bit adder is the carryout of the second 4-bit adder and so on

	signal C : STD_LOGIC_VECTOR(7 downto 1);

-- Use 8, 4-bit adders to create 32-bit adder
begin
	Add_0: adder_4 port map(A(3 downto 0), B(3 downto 0), carryin, sum(3 downto 0), C(1));
	Add_1: adder_4 port map(A(7 downto 4), B(7 downto 4), C(1), sum(7 downto 4), C(2));
	Add_2: adder_4 port map(A(11 downto 8), B(11 downto 8), C(2), sum(11 downto 8), C(3));
	Add_3: adder_4 port map(A(15 downto 12), B(15 downto 12), C(3), sum(15 downto 12), C(4));
	Add_4: adder_4 port map(A(19 downto 16), B(19 downto 16), C(4), sum(19 downto 16), C(5));
	Add_5: adder_4 port map(A(23 downto 20), B(23 downto 20), C(5), sum(23 downto 20), C(6));
	Add_6: adder_4 port map(A(27 downto 24), B(27 downto 24), C(6), sum(27 downto 24), C(7));
	Add_7: adder_4 port map(A(31 downto 28), B(31 downto 28), C(7), sum(31 downto 28), carryout);
end struct;
```

- **Datapath Code**

```
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;


entity datapath is
   Port(    clk:    in std_logic
);
end datapath;

architecture behaviour of datapath is
   component reg is
          port ( clk : in std_logic;
                    R1 : in std_logic_vector (4 downto 0);
                    R2 : in std_logic_vector (4 downto 0);
                    WriteReg : in std_logic_vector (4 downto 0);
                    WriteData : in std_logic_vector (31 downto 0);
                    RegWrite: in std_logic;
                    ReadData1 : out std_logic_vector (31 downto 0);
                    ReadData2 : out std_logic_vector (31 downto 0));
   end component;

   component instrRegister is
          port(    clk:          in std_logic;
                    MemData:    in STD_LOGIC_VECTOR(31 downto 0);
                    IRWrite:    in STD_LOGIC;
                    OpCode, Func:    out STD_LOGIC_VECTOR(5 downto 0);
                    Rs, Rt, Rd:    out STD_LOGIC_VECTOR(4 downto 0);
                    Imm:     out STD_LOGIC_VECTOR(15 downto 0);
                    JumpAddr:    out STD_LOGIC_VECTOR(25 downto 0));

   end component;

   component input_mux_2 is
          port (A:    in STD_LOGIC_VECTOR(31 downto 0);
                   B:    in STD_LOGIC_VECTOR(31 downto 0);
                   S:     in STD_LOGIC;
                   Y:    out STD_LOGIC_VECTOR(31 downto 0));

   end component;

   component input_mux_4 is
          port(    A,B,C,D:     in STD_LOGIC_VECTOR(31 downto 0);
                    S:          in STD_LOGIC_VECTOR(1 downto 0);
                    Y:          out STD_LOGIC_VECTOR(31 downto 0));
   end component;

   component controlUnit is
          port(    clk:    in std_logic;
                    OpCode:     in STD_LOGIC_VECTOR(5 downto 0);
```

```
                PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite, ALUSrcA,
RegWrite, RegDst: out STD_LOGIC;
                PCSrc, ALUOp, ALUSrcB: out STD_LOGIC_VECTOR(1 downto 0));
    end component;

    component alu32 is
        Port(    A: in STD_LOGIC_VECTOR(31 downto 0);
                        B: in STD_LOGIC_VECTOR(31 downto 0);
                        ALUop: in STD_LOGIC_VECTOR(3 downto 0);
                        zero: out STD_LOGIC;
                        result: out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    component signExt is
        Port(    Imm:        in STD_LOGIC_VECTOR(15 downto 0);
                ImmExt:             out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    component memory is
        Port (    address:    in STD_LOGIC_VECTOR(31 downto 0);
                memWriteIn: in STD_LOGIC_VECTOR(31 downto 0);
                memRead:    in STD_LOGIC;
                memWrite:    in STD_LOGIC;
                memData:    out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    component memDataReg is
        Port ( memData:    in STD_LOGIC_VECTOR(31 downto 0);
        memDataOut: out STD_LOGIC_VECTOR(31 downto 0));

    end component;

    component tmpReg is
        Port(     clk :     in std_logic;
                readData: in STD_LOGIC_VECTOR(31 downto 0);
                        outData:  out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    component shift_left2 is
        Port ( input:    in STD_LOGIC_VECTOR(31 downto 0);
                        output:  out STD_LOGIC_VECTOR(31 downto 0));
    end component;

    component r is
        Port (    enable: in STD_LOGIC;
                D: in STD_LOGIC_VECTOR(31 downto 0);
                Q: out STD_LOGIC_VECTOR(31 downto 0));
    end component;
```

```
component aluControl is
        Port(    func:        in STD_LOGIC_VECTOR(5 downto 0);
                    ALUOp:              in STD_LOGIC_VECTOR(1 downto 0);
                    Op:        out STD_LOGIC_VECTOR(3 downto 0));
end component;

component shiftleft_j is
        Port(    input:    in STD_LOGIC_VECTOR(25 downto 0);
                                output:  out STD_LOGIC_VECTOR(27 downto 0));
end component;

signal PCWriteCond: STD_LOGIC;
signal PCWrite: STD_LOGIC;
signal IorD: STD_LOGIC;
signal MemRead: STD_LOGIC;
signal MemWrite: STD_LOGIC;
signal MemtoReg: STD_LOGIC;
signal IRWrite: STD_LOGIC;
signal ALUSrcA: STD_LOGIC;
signal RegWrite: STD_LOGIC;
signal RegDst:  STD_LOGIC;
signal OpCode: STD_LOGIC_VECTOR(5 downto 0);
signal PCSrc, ALUOp, ALUSrcB: STD_LOGIC_VECTOR(1 downto 0);
signal D,Q: std_logic_VECTOR(31 downto 0) ;
signal E: std_logic;
signal ALUZero: std_logic;
signal address, memWriteIn, memData : std_logic_VECTOR(31 downto 0);
signal readR1, readR2: std_logic_vector(4 downto 0);
signal instr4 : std_logic_vector(4 downto 0);

signal WriteR: std_logic_vector(31 downto 0);
signal writeData: std_logic_vector(31 downto 0);
signal Imm: std_logic_vector(15 downto 0);
signal ALUOut, memDataOut: std_logic_vector(31 downto 0);
signal A, Aout, B, Bout: std_logic_vector(31 downto 0);
signal ImmExt: std_logic_vector(31 downto 0);
signal sl2: std_logic_vector(31 downto 0);
signal ALU_in1, ALU_in2 : std_logic_vector(31 downto 0);
signal pJumpAddr : std_logic_vector(25 downto 0);
signal JumpAddr : std_logic_vector(27 downto 0);
signal JumpAddr1 : std_logic_vector(31 downto 0);
signal ALUResult: std_logic_vector(31 downto 0);
signal func: std_logic_vector(5 downto 0);
signal Op: std_logic_vector(3 downto 0);
signal mux1_in0, mux1_in1 : std_logic_vector(31 downto 0);
```

```
begin

   controlU: controlUnit port map ( clk, OpCode, PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg,
IRWrite, ALUSrcA, RegWrite, RegDst,
            PCSrc, ALUOp, ALUSrcB );
   E <= (ALUZero and PCWriteCond) or PCWrite;
   PC : r port map (E, D, Q);
   Mux0 : input_mux_2 port map (Q, ALUResult, IorD, Address);
   -- memory
   Memory1: memory port map (address, memWriteIn, memRead, memWrite, memData);
   instructReg: instrRegister port map (clk, memData, IRWrite, OpCode, func, readR1, readR2, instr4, Imm,
pJumpAddr);
   MemDataReg1 : memDataReg port map ( memData, memDataOut);

   mux1_in0 <= std_logic_vector(resize(unsigned('0'&readR2),32)); --resizing from
   mux1_in1 <= std_logic_vector(resize(unsigned('0'&instr4),32));

   Mux1 : input_mux_2 port map (mux1_in0, mux1_in1, RegDst, WriteR);
   Mux2 : input_mux_2 port map (ALUOut, memDataOut, MemtoReg, writeData);
   Regs: reg port map ( clk, readR1, readR2, WriteR(4 downto 0), writeData, RegWrite, A,B);
   signExtnd: signExt port map (Imm, ImmExt);
   RegA : tmpReg port map (clk, A, Aout);
   RegB: tmpReg port map (clk, B, Bout);
   shiftleft2:  shift_left2 port map (ImmExt, sl2) ;
   Mux1A : input_mux_2 port map (Q, Aout, ALUSrcA, ALU_in1);
   Mux4 : input_mux_4 port map (Bout, x"00000004", ImmExt, sl2, ALUSrcB, ALU_in2);
   shiftleft2J:  shiftleft_j port map (pJumpAddr, JumpAddr);

   JumpAddr1 <= Q(31 downto 28) & JumpAddr(27 downto 0);

   Mux4B : input_mux_4 port map (ALUResult, ALUOut, JumpAddr1, x"00000000", PCSrc, D);
   ALUCon : aluControl port map (func, ALUOp, Op);
   ALU: alu32 port map (alu_in1, alu_in2, op, ALUzero, ALUResult) ;
   ALUOut1: tmpReg port map (clk, ALUResult, ALUOut);

end behaviour;
```