

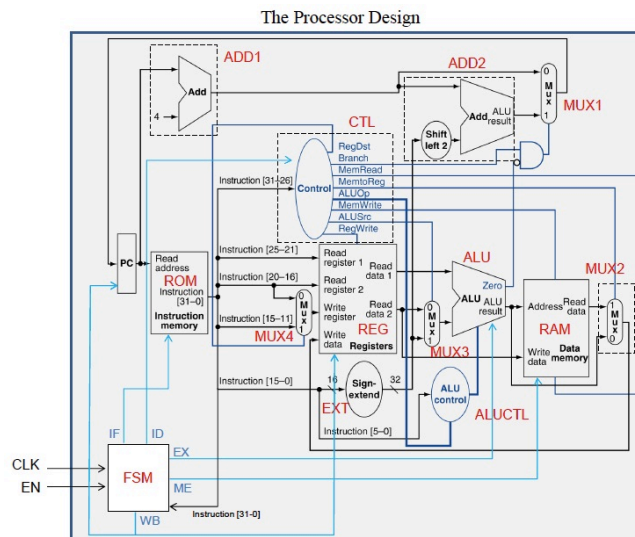
City University of New York
City College



Department of Computer Science, C.S
CSC 343-FG Computer Organization Laboratory
NAC 7/118

Professor Zheng Peng

***Lab 5: 32-bit MIPS processor using
VHDL***



Group 7: Ahmed Abdelrahman,
Anthony Shalagin,
Mohammad Zilon

Due date: 05/28/2017

Semester: Spring 2017

I. Overview

In this lab, the goal is to implement a single cycle 32-bit MIPS processor using VHDL that supports several instructions including addu, addi, addiu, bne, and sw. This lab will allow us to have a greater understanding of the MIPS assembly language, VHDL, and the MIPS processor design. We will run our processor on the provided assembly file, Fibonacci.asm, which will use the above-mentioned instructions to write 10 Fibonacci numbers into the data memory.

Note that addi and addiu perform the same operation except that addi produces a “trap” on overflow in the actual MIPS processor. However, in this design, we will implement addi and addiu in the same way without producing the trap.

II. The Modules

The processor consists of several modules that are combined to make the processor work. The following are the modules used in the processor and their implementations:

1. State Machine Module

This module is the brain of the processor. It takes in the processor’s inputs, which are the clock signal and the enable signal. The enable signal is what enables the processor to start working, and the clock signal is used to trigger different stages and determine the order of operations. It also takes in the instruction fetched as an input, which allows it to track when to stop the application. If the instruction is 0x00000000, then it knows that’s when to stop. The state machine keeps track of what stage we are currently on. There are five stages for each instruction: instruction fetch, instruction decode, execution, memory access, and the write back stage. The state machine starts with the instruction fetch stage and updates the stage at each positive-triggering clock edge. Based on the stage, it produces the correct output signal to activate other parts of the processor. During the instruction fetch stage, it activates/enables the instruction memory module. During the instruction decode stage, it activates the control module. During the execution stage, it activates the ALU module. During the memory access stage, it activates the data memory module. During the write back stage, it activates the register module and PC module.

```

architecture FSM_arch of FSM is
    type state_type is (InF, ID, EX, ME, WB); -- used InF instead of IF since IF is a keyword
    signal state : state_type := InF;
begin
    process(I_FSM_CLK)
    begin
        -- if FSM is enabled, instruction is not 0's
        if I_FSM_EN = '1' and I_FSM_INST /= (I_FSM_INST'range => '0') then
            -- and if it's the rising edge, produce correct output signals based on current state
            if rising_edge(I_FSM_CLK) then
                case state is
                    when InF =>
                        O_FSM_IF <= '1';
                        O_FSM_ID <= '0';
                        O_FSM_EX <= '0';
                        O_FSM_ME <= '0';
                        O_FSM_WB <= '0';
                        state <= ID;
                    when ID =>
                        O_FSM_IF <= '0';
                        O_FSM_ID <= '1';
                        state <= EX;
                    when EX =>
                        O_FSM_ID <= '0';
                        O_FSM_EX <= '1';
                        state <= ME;
                    when ME =>
                        O_FSM_EX <= '0';
                        O_FSM_ME <= '1';
                        state <= WB;
                    when WB =>
                        O_FSM_ME <= '0';
                        O_FSM_WB <= '1';
                        state <= InF;
                end case;
            end if;
        end if;
    end process;
end FSM_arch;

```

2. PC Module

The PC, which stands for program counter, is a register that stores the instruction address that needs to be fetched. It takes in two inputs which are the new PC and a signal to enable the module. The enabling signal comes from the state machine module. During the write back stage, the state machine module enables the PC module, and the PC module outputs the new PC, which is its input. Therefore, it serves to hold and output the address of the new instruction that needs to be fetched. It also helps at the start of the program, as its default output value is 0x00000000, which enables the processor to fetch the first instruction at the start of the program.

```

entity PC is
    Port ( I_PC_UPDATE : in  STD_LOGIC;
          I_PC         : in  STD_LOGIC_VECTOR (31 downto 0);
          O_PC         : out STD_LOGIC_VECTOR (31 downto 0) := x"00000000");
    -- O_PC is initialized to output 0 to fetch the first instruction
end PC;

architecture PC_arch of PC is
begin
    process(I_PC_UPDATE)
    begin
        if I_PC_UPDATE = '1' then
            O_PC <= I_PC;
        end if;
    end process;
end PC_arch;

```

3. Instruction Memory Module

The instruction memory (ROM) module is responsible for holding 256 bytes of instruction memory. It will initially read a file that contains the instructions encoded in binary and place those instructions inside the instruction memory. The ROM module takes in two inputs; the enable signal and the address. When the ROM is enabled by the state machine at the instruction fetch stage, it will take the address, which comes from PC, and will output the instruction at that address.

```

-- Call the function to initialize the ROM
signal rom : rom_type := init_rom("Fibonacci.bin");
begin
  process(I_ROM_EN)
    variable index : integer; -- index (address) to access the ROM data
  begin
    if I_ROM_EN = '1' then
      index := to_integer(unsigned(I_ROM_ADDR));
      O_ROM_DATA <= rom(index+3) & rom(index+2) & rom(index+1) & rom(index);
    end if;
  end process;
end ROM_arch;

```

4. Control Module

The control module is responsible for generating the control signals based on the opcode of the instructions. Therefore, the control module has two inputs; the enable signal and the first six bits of the instruction, which is the opcode. The state machine enables the control module during the instruction decoding stage, and then the control module takes the opcode, from the output of the instruction memory module, and produces the correct output signals. The control output signals are *RegDst* which determines which register should be the write register in the register module, *Branch* which determines if we should branch or not, *MemRead* which determines if we should read from the data memory module, *MemtoReg* which determines which data (from the ALU module or the data memory module) should be written back in the register module, *ALUOp* which helps the ALU control module determine which operation the ALU module should perform, *MemWrite* which determines if we should write to the data memory module, *ALUSrc* which determines which source (from the sign extension module or from the register module) should be used as the input to the ALU, and *RegWrite* which determines if we should write back to the register module or not.

```

architecture Control_arch of Control is
begin
  process(I_CTL_EN)
    begin
      if I_CTL_EN = '1' then
        -- addu
        if I_CTL_INST = "000000" then
          O_CTL_RegDst <= '1';
          O_CTL_Branch <= '0';
          O_CTL_MemRead <= '0';
          O_CTL_MemtoReg <= '0';
          O_CTL_ALUOp <= "10";
          O_CTL_MemWrite <= '0';
          O_CTL_ALUSrc <= '0';
          O_CTL_RegWrite <= '1';
        -- addi or addiu
        elsif I_CTL_INST = "001000" or I_CTL_INST = "001001" then
          O_CTL_RegDst <= '0';
          O_CTL_Branch <= '0';
          O_CTL_MemRead <= '0';
          O_CTL_MemtoReg <= '0';
          O_CTL_ALUOp <= "00";
          O_CTL_MemWrite <= '0';
          O_CTL_ALUSrc <= '1';
          O_CTL_RegWrite <= '1';
        -- bne
        elsif I_CTL_INST = "000101" then
          O_CTL_RegDst <= '0';
          O_CTL_Branch <= '1';
          O_CTL_MemRead <= '0';
          O_CTL_MemtoReg <= '0';
          O_CTL_ALUOp <= "01";
          O_CTL_MemWrite <= '0';
          O_CTL_ALUSrc <= '0';
          O_CTL_RegWrite <= '0';
        -- sw
        elsif I_CTL_INST = "101011" then
          O_CTL_RegDst <= '0';
          O_CTL_Branch <= '0';
          O_CTL_MemRead <= '0';
          O_CTL_MemtoReg <= '0';
          O_CTL_ALUOp <= "00";
          O_CTL_MemWrite <= '1';
          O_CTL_ALUSrc <= '1';
          O_CTL_RegWrite <= '0';
        end if;
      end if;
    end process;
  end Control_arch;

```

5. Register Module

The register module includes 32 registers with each register being 32-bit in width. It takes in an *enable signal* which comes from the state machine module and is 1 during the write back stage, a *write enable signal* which comes from the control module and is 1 if the instruction writes back to a register, a *register source* which is the address of the first read register, a *register target* which is the address of the second read register and could be the address of the write register depending on the RegDst signal coming from the control module, a *register destination* which could be the address of the write register depending on RegDst, and the *write data* which is the data that could be written into the write register. The write data comes from either the data memory module or the ALU module, and is decided by the control module's output signal MemtoReg. The register module always outputs the data read from both the register source and register target inputs. However, it only writes back to the write register when both the enable signal and the write enable signal are 1.

```
architecture REG_arch of REG is
    -- Creating 32 32-bit registers initialized to 0
    type data_array is array (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
    signal reg: data_array := (others => (others => '0'));
begin
    -- access and output the data in the registers based on the input address/register number
    O_REG_DATA_A <= reg(to_integer(unsigned(I_REG_SEL_RS)));
    O_REG_DATA_B <= reg(to_integer(unsigned(I_REG_SEL_RT)));
    process (I_REG_EN)
    begin
        -- if the register module is enabled and the write enable signal is 1
        if I_REG_EN = '1' and I_REG_WE = '1' then
            -- then write to the register at the given input address
            reg(to_integer(unsigned(I_REG_SEL_RD))) <= I_REG_DATA_RD;
        end if;
    end process;
end REG_arch;
```

6. Sign Extension Module

The sign extension module simply performs a signed extension on the immediate number encoded in the I-format and branch instructions. It takes a 16-bit input and outputs a 32-bit signed extended version. The 32-bit upper half is either all 0's or all 1's depending on the most significant bit of the 16-bit input. The lower half of the 32-bit output is the same as the input.

```
architecture EXT_arch of EXT is
begin
    -- upper half of output is equal to the sign bit (MSB of input)
    O_EXT_32(31 downto 16) <= (others => I_EXT_16(15));
    O_EXT_32(15 downto 0) <= I_EXT_16;
end EXT_arch;
```

7. ALU Control Module

The ALU control module determines the ALU operation. It takes in two inputs; the *ALUOp* signal which comes from the control module and the *funct* code which comes from the instruction memory module. Based on the ALUOp and the funct, the ALU control module produces the corresponding ALU control signal that determines which operation the ALU should perform.

```

architecture ALUCTR_arch of ALUCTR is
begin
  process(I_ALU_OP, I_ALU_FUNCT)
  begin
    -- if I-type instruction, then add
    if I_ALU_OP = "00" then
      O_ALU_CTR <= "0010";

    -- if branch instruction, then subtract
    elsif I_ALU_OP = "01" then
      O_ALU_CTR <= "0110";

    -- if R-type instruction, then check funct
    elsif I_ALU_OP = "10" then
      if I_ALU_FUNCT = "100001" then -- if addu, then add
        O_ALU_CTR <= "0010";
      end if;
    end if;
  end process;
end ALUCTR_arch;

```

8. ALU Module

In this implementation, the ALU is expected to be able to perform addition and subtraction. The ALU module takes in the *enable* signal which comes from the state machine and is 1 during the execution stage, the *ALU control* signal which comes from the ALU control module and determines which operation to perform (0010 for addition and 0110 for subtraction), and two inputs to perform the operation on. The first input is always the first data read from the register module; however, the second input can be either the second data read from the register module or the sign extension output depending on the ALUSrc signal from the control module. The ALU module outputs the results of the operation as well as a 1-bit signal that is 1 if the output is zero. This is used in the branch instruction.

```

architecture ALU_arch of ALU is
  signal ALU_OUTPUT : STD_LOGIC_VECTOR (31 downto 0);
begin
  process (I_ALU_EN)
  begin
    if I_ALU_EN = '1' then
      -- Add
      if I_ALU_CTL = "0010" then
        ALU_OUTPUT <= STD_LOGIC_VECTOR(unsigned(I_ALU_A) + unsigned(I_ALU_B));
      -- Subtract
      elsif I_ALU_CTL = "0110" then
        ALU_OUTPUT <= STD_LOGIC_VECTOR(unsigned(I_ALU_A) - unsigned(I_ALU_B));
      end if;
    end if;
  end process;
  O_ALU_Zero <= '1' when ALU_OUTPUT = (ALU_OUTPUT'range => '0') else '0';
  O_ALU_Out <= ALU_OUTPUT;
end ALU_arch;

```

9. Data Memory Module

The module is expected to have 256 bytes of data memory. The data memory module takes in an *enable* signal which comes from the state machine module and is 1 during the memory access stage, a *read enable* signal which comes from the control module and it allows us to read from the data memory based on the *input address*, a *write enable* signal which allows us to write the *input data* into the corresponding data memory based on the input address. When the data memory module is enabled, it checks to see whether we are allowed to read or write. If we can read, then the data memory module outputs the read data from the input address. If we can write, then the module will write the input data into the data memory based on the input address given. We note that the system data memory range is 0x00002000 – 0x000020FF. However, in our implementation, the data memory range is 0x00000000 – 0x000000FF. So, only the last 8-bits contribute to the address.

```

architecture RAM_arch of RAM is
    -- Create the 256 bytes of data memory initialized to 0's.
    -- Note: Little Endian is used
    type ram_type is array (0 to 255) of STD_LOGIC_VECTOR(7 downto 0);
    signal ram : ram_type := (others => (others => '0'));
begin
    process(I_RAM_EN)
        variable index : integer;
    begin
        if I_RAM_EN = '1' then
            index := to_integer(unsigned(I_RAM_ADDR(7 downto 0))); -- only last 8 bits contribute to the address
            if I_RAM_RE = '1' then
                O_RAM_DATA <= ram(index+3) & ram(index+2) & ram(index+1) & ram(index);
            elsif I_RAM_WE = '1' then
                ram(index) <= I_RAM_DATA(7 downto 0);
                ram(index+1) <= I_RAM_DATA(15 downto 8);
                ram(index+2) <= I_RAM_DATA(23 downto 16);
                ram(index+3) <= I_RAM_DATA(31 downto 24);
            end if;
        end if;
    end process;
end RAM_arch;

```

10. Adder Module (1)

This adder module increments the PC value by 4. It takes the output of the PC module, which is the current instruction address. Then, it adds 4 to it and outputs the result. This allows us to update the PC to fetch the next instruction.

```

architecture ADD1_arch of ADD1 is
begin
    O_ADD1_Out <= STD_LOGIC_VECTOR(unsigned(I_ADD1_A) + 4);
end ADD1_arch;

```

11. Adder Module (2)

This adder module computes the branch target address. It takes in the output of the adder 1 module, which is PC+4, and then it adds the output of the sign extension multiplied by 4. So, the immediate of the branch instruction is first sign extended, then it is passed to the adder 2 module which shifts it by 2 bits to the left, then adds that to PC+4, and that gives us the branch target address. $(PC+4 + \text{signExtend}[\text{immediate}] * 4)$

```

architecture ADD2_arch of ADD2 is
    signal I_ADD2_B_buffer : STD_LOGIC_VECTOR (31 downto 0);
begin
    I_ADD2_B_buffer <= STD_LOGIC_VECTOR(shift_left(unsigned(I_ADD2_B), 2));
    O_ADD2_Out <= STD_LOGIC_VECTOR(unsigned(I_ADD2_A) + unsigned(I_ADD2_B_buffer));
end ADD2_arch;

```

12. Multiplexers

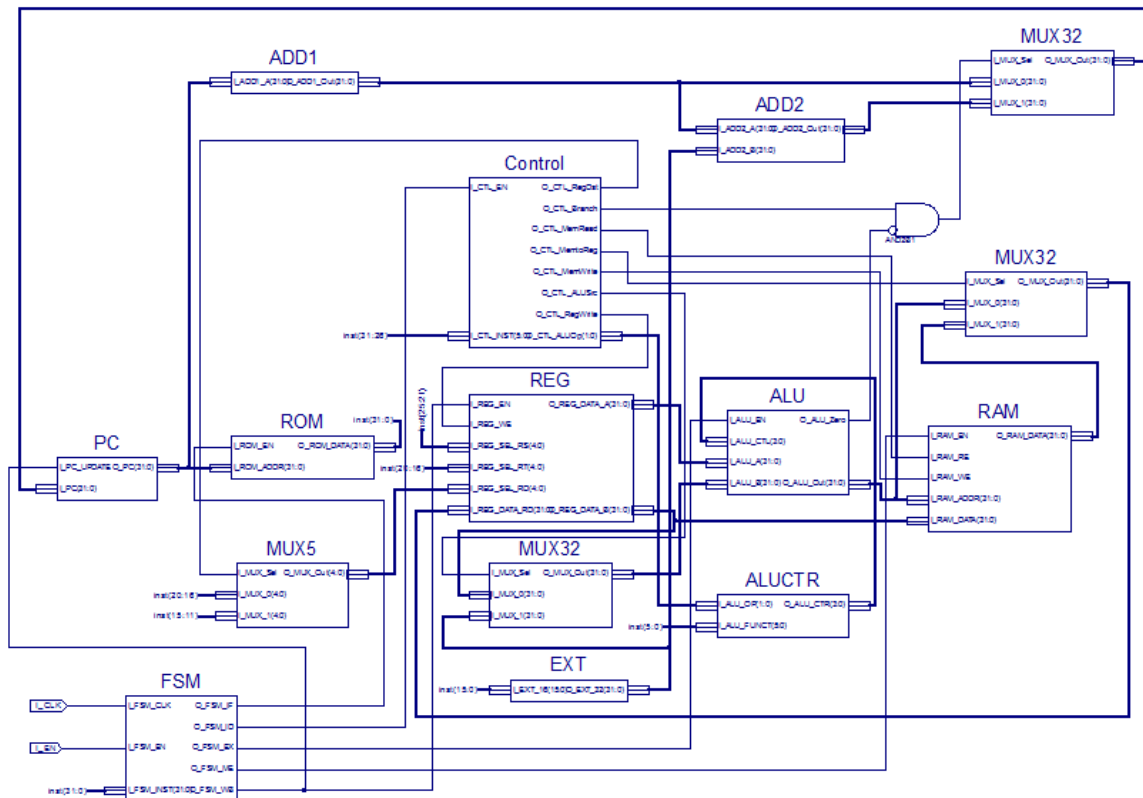
There are four 2-1 multiplexers. Their basic implementation is the same. The output is I_MUX_0 if the I_MUX_Sel is 0, or the output is I_MUX_1 if I_MUX_Sel is 1. We created two different implementations; MUX5 and MUX32. The only difference is the width of the input buses. MUX5 takes in two 5-bits inputs while MUX32 takes in two 32-bits inputs. The multiplexers are used 4 times in the processor to pick a value based on certain control signals. The following are the four cases where the multiplexers were used:

1. Based on RegDst, we pick either the register target or the register destination for the write register input of the Register module.
2. Based on ALUSrc, we pick either the second read data output from the register module or the output of the sign extension to be the input of the ALU.
3. Based on MemtoReg, we either pick either the ALU output or the read data from the data memory module to be written back in the register module.
4. Based on zero signal output from the ALU and the branch control signal from the control module, we either update PC with the branch target address or just PC+4.

```
architecture MUX5_arch of MUX5 is
begin
    O_MUX_Out <= I_MUX_0 when I_MUX_Sel = '0' else I_MUX_1;
end MUX5_arch;
```

III. The Top Module

By using the VHDL implementation of each of those modules, we can use Xilinx to create a schematic of each of those modules. Then, we can connect those modules together in a schematic file using wires to create the top module. The following shows how the top module was implemented in Xilinx.



IV. Testing

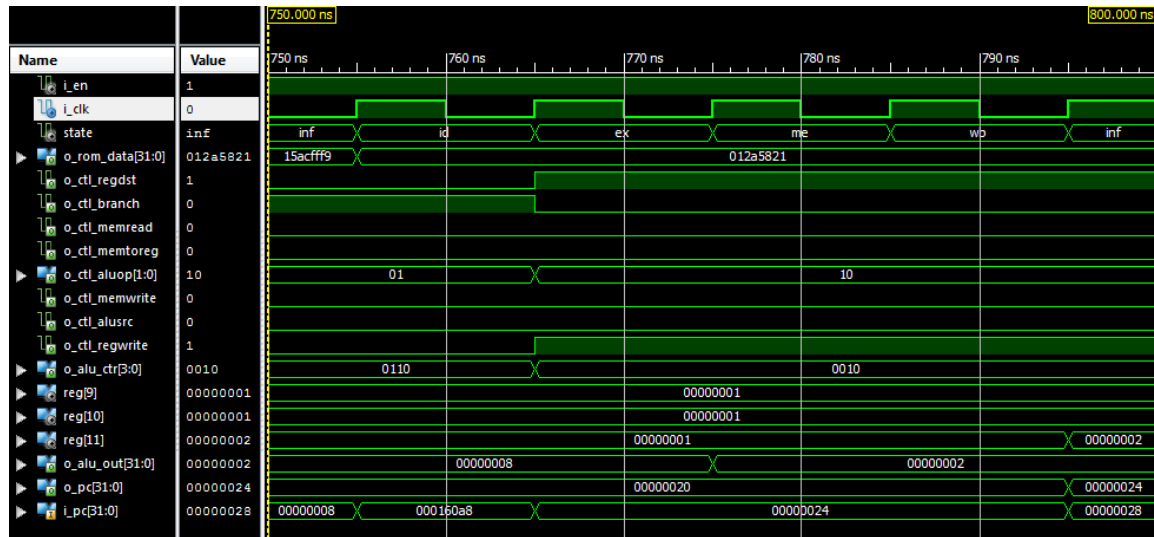
So now that we have created our top module, we decided to test the processor against the five instructions that it is designed to support, and we observed if the waveforms match the expected signal values at each stage.

1. addu \$t3, \$t1, \$t2

This instruction does the following $\$t3 (\$11) = \$t1 (\$9) + \$t2 (\$10)$. The following table shows what the control signals and signals associated with this instruction should be equal to. We will compare the values in this table with the waveform from our simulation. Note that we are examining this instruction during the second iteration of the loop.

Signal	Value	Note
Instruction	0x012a5821	This is the instruction encoding
RegDst	1	Writing to the destination register as it's R-type instruction
Branch	0	Not a branch instruction
MemRead	0	Not reading from the data memory
MemtoReg	0	We pick the ALU result to be written back in the register
ALUOp	10	This is the ALUOp for R-type instructions
MemWrite	0	Not writing to the data memory
ALUSrc	0	Adding the value in the register not from sign extension
RegWrite	1	Enabled since we write back to the register
ALU_CTR	0010	Addition operation for the ALU
ALU_Out	0x00000002	Result of the addition
reg[9]	0x00000001	The value in the first register which is \$t1
reg[10]	0x00000001	The value in the second register which is \$t2
reg[11]	0x00000001	The initial value of the third register which is \$t3
reg[11]*	0x00000002	The value of \$t3 after the write back stage
O_PC	0x00000020	Current instruction address before write back stage
I_PC	0x00000024	Next instruction address before write back stage

The following is the waveform output from our simulation to this instruction.



We can observe that the values in the waveform match our values in the table. During the instruction fetch stage (750 ns – 760 ns), we fetch the correct instruction which is 0x012a5821 from the correct O_PC which is 0x00000020. During the instruction decoding stage (760 ns – 770 ns), the correct control signals are generated. During the execution stage (770 ns – 780 ns), the ALU output is generated and we can observe it changes to the correct value of 0x00000002. During the memory access stage (780 ns – 790 ns), nothing happens as we’re not dealing with memory. During the write back stage (790 ns – 800 ns), the PC gets updated but also reg[11]’s value changes to 0x00000002 which is the ALU output and the value that \$t3 should take after the instruction is executed. Therefore, this instruction works correctly with our MIPS processor.

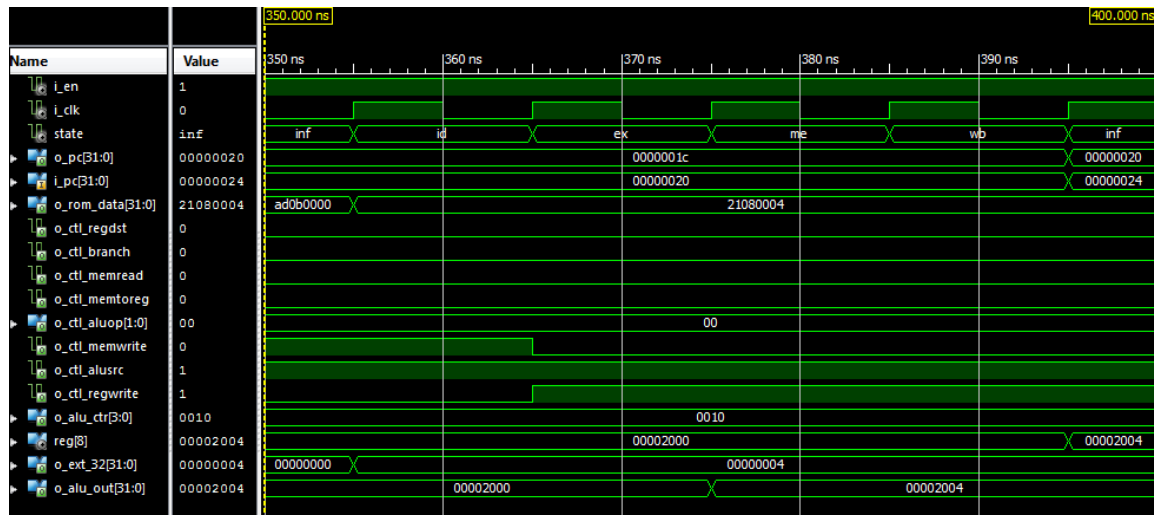
2. addi \$t0, \$t0, 4

This instruction was executed right before the loop started. This instruction accomplishes the following: $\$t0 (\$8) = \$t0 + 4$. The following table shows all the control signals and other signals associated with this instruction.

Signal	Value	Note
Instruction	0x21080004	This is the instruction encoding
RegDst	0	Writing to the target register
Branch	0	Not a branch instruction
MemRead	0	Not reading from memory
MemtoReg	0	We pick the ALU result to be written back in the register
ALUOp	00	This is the ALUOp for I-type instructions
MemWrite	0	Not writing to the memory
ALUSrc	1	Sign extension output is the input to the ALU
RegWrite	1	Enabled since we write back to the register
ALU_CTR	0010	Addition operation for the ALU

O_EXT	0x00000004	Positive 4. Sign extended with 0's
reg[8]	0x00002000	The initial value in \$t0
reg[8]*	0x00002004	The value in \$t0 after the write back stage
ALU_Out	0x00002004	Result of the addition $0x00002000 + 0x00000004$
O_PC	0x0000001c	Current instruction address before write back stage
I_PC	0x00000020	Next instruction address before write back stage

The following is the waveform output from our simulation to this instruction.



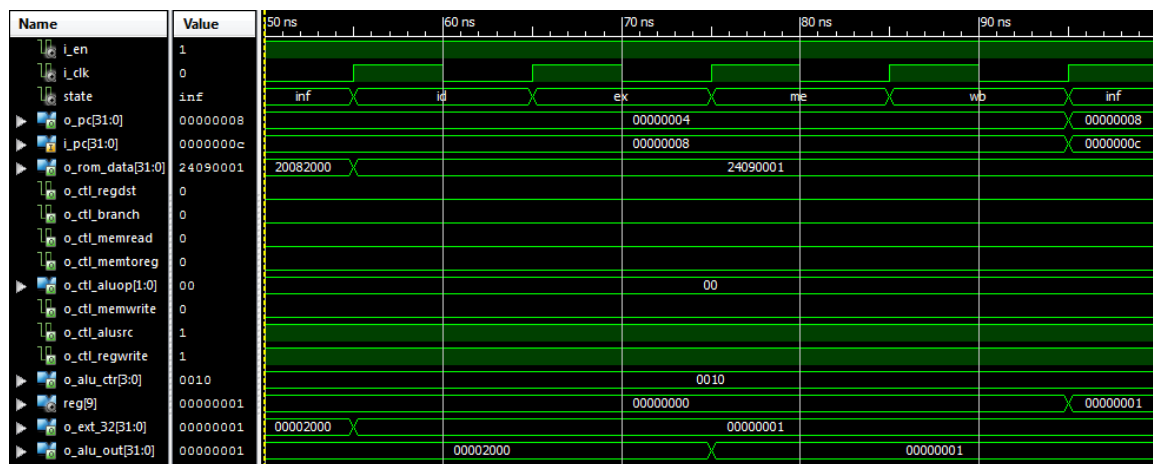
We can observe that the values in the waveform match the values in the table. During the instruction fetch stage (350 ns – 360 ns), the correct instruction is fetched which is 0x21080004 from the O_PC, which is the instruction address 0x0000001c. During the instruction decoding stage (360 ns – 370 ns), the correct control signals are generated. During the execution stage (370 ns – 380 ns), the initial value in \$t0 is added with the sign extended immediate to give the correct ALU_Out, $0x00002000 + 0x00000004 = 0x00002004$. During the memory access stage (380 ns – 390 ns), nothing happens as we're not dealing with memory. During the write back stage (390 ns – 400 ns), the value in register \$t0 is written so we can see it changes to the output of the ALU, 0x00002004.

3. addiu \$t1, \$0, 1

This is the second instruction in the program. Note that this was originally written as a pseudoinstruction which is `li $t1, 1`. However, the MARS compiler changes it to this basic `addiu` instruction. This instruction does the following: $\$t1 (\$9) = \$0 + 1 = 1$. The following table shows the control signals as well as any other signal associated with this instruction.

Signal	Value	Note
Instruction	0x24090001	This is the instruction encoding
RegDst	0	Writing to the target register
Branch	0	Not a branch instruction
MemRead	0	Not reading from memory
MemtoReg	0	We pick the ALU result to be written back in the register
ALUOp	00	This is the ALUOp for I-type instructions
MemWrite	0	Not writing to the memory
ALUSrc	1	Sign extension output is the input to the ALU
RegWrite	1	Enabled since we write back to the register
ALU_CTR	0010	Addition operation for the ALU
O_EXT	0x00000001	Positive 1. Sign extended with 0's
reg[9]	0x00000000	The initial value in \$t1
reg[9]*	0x00000001	The value in \$t1 after the write back stage
ALU_Out	0x00000001	Result of the addition 0x00000000 + 0x00000001
O_PC	0x00000004	Current instruction address before write back stage
I_PC	0x00000008	Next instruction address before write back stage

The following is the waveform output from our simulation to this instruction.



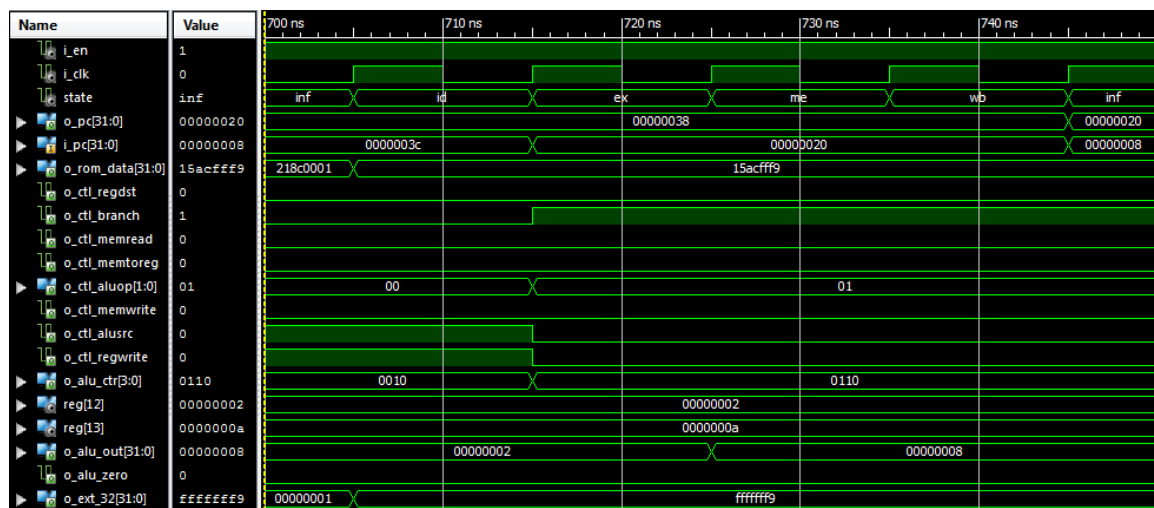
We can observe that the values in the waveform match our values in the table, which shows that our MIPS processor supports this instruction. During the instruction fetch stage (50 ns – 60 ns), the correct instruction 0x24090001 is fetched from O_PC, 0x00000004, which is the address of the instruction. During the instruction decoding stage (60 ns – 70 ns), the control signals are correctly generated. During the execution stage (70 ns – 80 ns), the ALU adds the sign extension output which is 0x00000001 and the initial value of \$9 which is 0x00000000, and as we see the ALU_Out is updated to 0x00000001. During the memory access stage (80 ns – 90 ns), nothing occurs, as we are not dealing with the memory. During the write back stage (90 ns – 100 ns), the ALU_Out is written into \$9, and we can observe that its value changes to 0x00000001 as expected.

4. bne \$t5, \$t4, loop

We will be examining the first occurrence of the bne instruction in the first iteration of the loop. The bne does the following: if $((\$t5 - \$t4) \neq 0)$, then go back 6 instruction (7 instructions from the next instruction). The following table shows all the control signals and other values associated with this instruction.

Signal	Value	Note
Instruction	0x15acfff9	This is the instruction encoding
RegDst	0	Writing to the target register although it doesn't matter
Branch	1	It is a branch instruction
MemRead	0	Not reading from memory
MemtoReg	0	Picked 0 although it doesn't matter since we don't write back
ALUOp	01	This is the ALUOp for branch instruction
MemWrite	0	Not writing to the memory
ALUSrc	0	The target register value is the input to the ALU
RegWrite	0	Not writing back to the register
ALU_CTR	0110	Subtraction operation for the ALU
O_EXT	0xffffffff9	-7. Sign extended with 1's since it's negative
reg[12]	0x00000002	The initial value in \$t4
reg[13]	0x0000000a	The initial value in \$t5
ALU_Out	0x00000008	Result of the subtraction
ALU_Zero	0	ALU_Out is not 0. So we branch
O_PC	0x00000038	Current instruction address before write back stage
I_PC	0x0000003c	Next instruction address
I_PC*	0x00000020	Next instruction address is modified since the ALU_Zero is 0. So we have to branch. The branch target address is $0x0000003c + (4)(0xffffffff9) = 0x00000020$

The following is the waveform output from our simulation to this instruction.



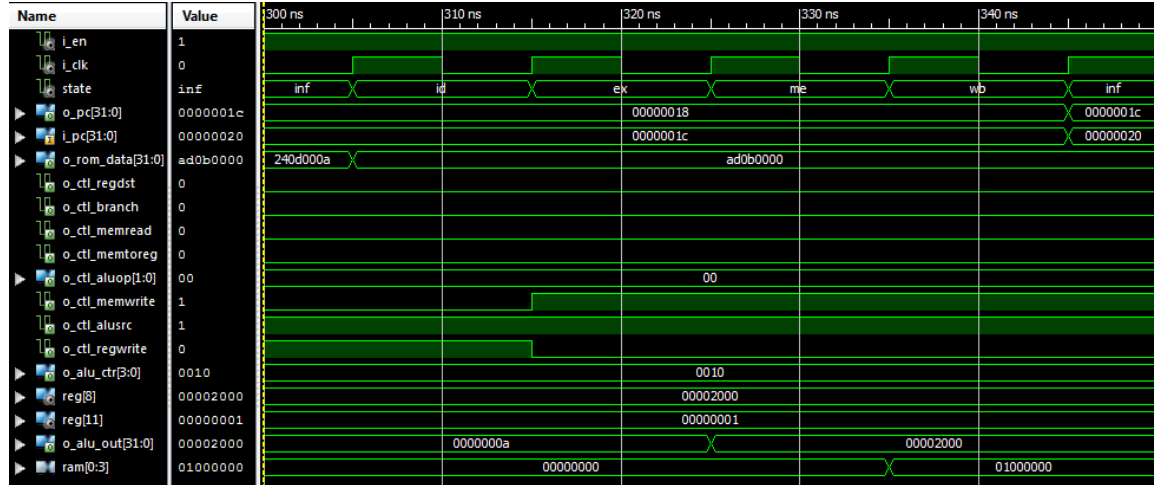
We can observe that the waveform's values match our table's values. During the instruction fetch stage (700 ns – 710 ns), the correct instruction 0x15acff9 is fetched from the O_PC address 0x00000038. During the instruction decoding stage (710 ns – 720 ns), the control signals are correctly generated. During the execution stage (720 ns – 730 ns), the ALU subtracts $\$13 - \$12 = 0x0000000a - 0x00000002 = 0x00000008$. We can observe the ALU_Zero is 0 as expected so we branch and we can see that I_PC changes value from PC+4 which is 0x0000003c to the branch target address 0x00000020. During the memory access (730 ns – 740 ns) and the write back stage (740 ns – 750 ns), nothing occurs since we don't write back or deal with memory in this instruction.

5. sw \$t3, 0(\$t0)

We will be examining the first occurrence of the sw instruction. It saves the value in \$t3 (\$11) in the data memory at the address \$t0 (\$8) + 0. The following table shows the control signals and all other values associated with the instruction.

Signal	Value	Note
Instruction	0xad0b0000	This is the instruction encoding
RegDst	0	Writing to the target register although it doesn't matter
Branch	0	Not a branch instruction
MemRead	0	Not reading from memory
MemtoReg	0	Picked 0 although it doesn't matter since we don't write back
ALUOp	00	This is the ALUOp for I-type instructions
MemWrite	1	We are writing to the memory
ALUSrc	1	The sign extended immediate is the input to the ALU
RegWrite	0	Not writing back to the register
ALU_CTR	0010	Addition operation for the ALU
reg[8]	0x00002000	The initial value in \$t0
reg[11]	0x00000001	The initial value in \$t3
ALU_Out	0x00002000	Result of the addition $0x00002000 + 0x00000000$
ram[0:3]	0x01000000	The value in the RAM after write back. It is little endian
O_PC	0x00000018	Current instruction address (before write back stage)
I_PC	0x0000001c	Next instruction address (before write back stage)

The following is the waveform output from our simulation to this instruction.



We can observe that the waveform's values match our table's values, which shows that our MIPS processor supports the sw instruction. During the instruction fetch stage (300 ns – 310 ns), the instruction 0xad0b0000 is fetched correctly from the correct O_PC address 0x00000018. During the instruction decode stage (310 ns – 320 ns), the correct control signals are generated. During the execution stage (320 ns – 330 ns), the ALU_Out value changes to the correct value, which is 0x00002000. During the memory access stage (330 ns – 340 ns), we can observe that the RAM changes to the correct value which is 0x01000000 at the given address. 0x00002000 is the address 0 for the RAM, and since it is little endian, we're writing the number 1 at the lower byte.

V. Remarks

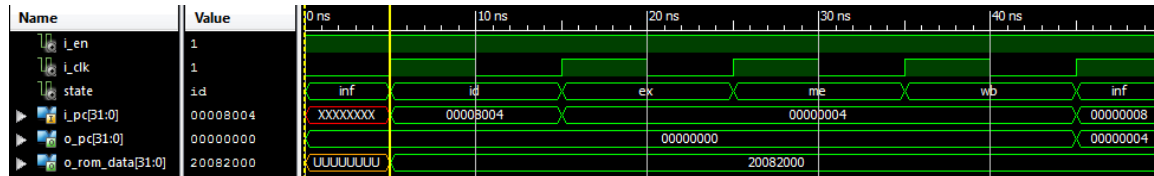
There are four main remarks that we needed to make sure are present for the processor to work correctly.

First, we needed to make sure that the Fibonacci.bin file is inside the directory where our project exists, so that it can locate the file and read the binary instructions and save them into the instruction memory.

Second, we needed to make sure that the simulation runs for the proper amount of time. We set the simulation runtime to 4000 ns. The program consists of 7 instructions inside the loop that runs for 9 iterations, plus 8 instructions before the loop. Therefore, the total number of instructions is $8 + 7 \times 9 = 71$ instructions. Each instruction requires 5 clock cycles. We set the clock period to be 10 ns. Therefore, the total running time of the program is $71 \times 5 \times 10 = 3550$ ns.

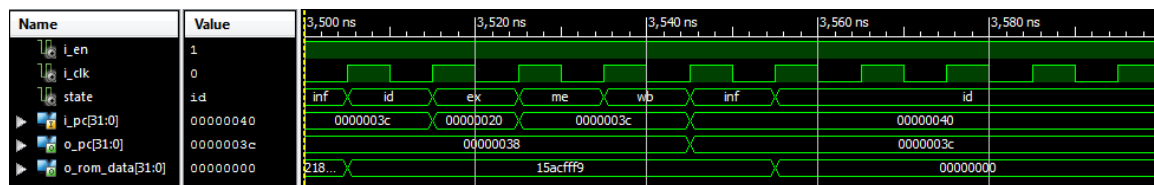
Third, we needed to determine how the program is started. The state machine module is the brain of the processor, and takes in the clock and enable signals. Therefore, we always wanted to initialize the state as the first state, which is the instruction fetch stage.

However, when the program is started and the instruction fetch stage is activated, the instruction memory module will not be able to output the first instruction, as there is no input address from PC. Therefore, we decided to set the default output value of PC to 0x00000000 for the instruction memory module to output the first instruction correctly. The following waveform shows how the program starts.



We can observe that before the first positive-triggering edge, the state is already set to instruction fetch, and the O_PC is already set to 0x00000000. Then at the first triggering edge, the correct instruction is fetched which is 0x20082000.

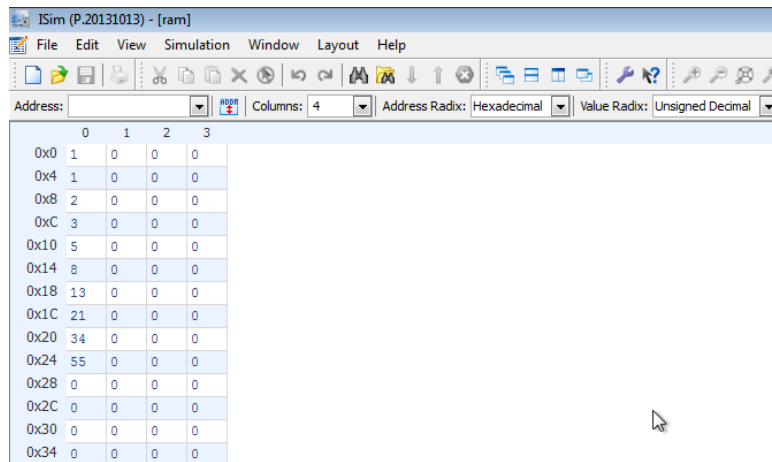
Fourth, we needed to determine how the program stops working. The state machine module is what's responsible for activating all the other modules; therefore, it should be the one responsible for stopping the program. We connected the instruction fetched as an input to the state machine module, and if the instruction fetched is 0x00000000, then we know that's when to stop the program, and we should not update the state or change any of the output values. The following shows the waveform at the last instruction.



Between 3500 ns and 3550 ns, we can observe the last instruction of the program. Then at the next positive-triggering edge that happens at 3555 ns, the instruction 0x00000000 is fetched, and then the stage does not change and nothing happens during the following triggering edges, which shows that the program stopped.

VI. Conclusion

In conclusion, we can see that the processor generated all the correct signals for the five instructions supported. It was also able to read the binary instructions file correctly, as well as know how to start and stop program. To finally check if the program has executed correctly on our MIPS processor, we checked the data memory in our simulation and the following were the results



	0	1	2	3
0x0	1	0	0	0
0x4	1	0	0	0
0x8	2	0	0	0
0xC	3	0	0	0
0x10	5	0	0	0
0x14	8	0	0	0
0x18	13	0	0	0
0x1C	21	0	0	0
0x20	34	0	0	0
0x24	55	0	0	0
0x28	0	0	0	0
0x2C	0	0	0	0
0x30	0	0	0	0
0x34	0	0	0	0

As we can observe, the 10 Fibonacci numbers have been successfully written into the data memory in little Endian format. This proves that our MIPS processor functions properly.