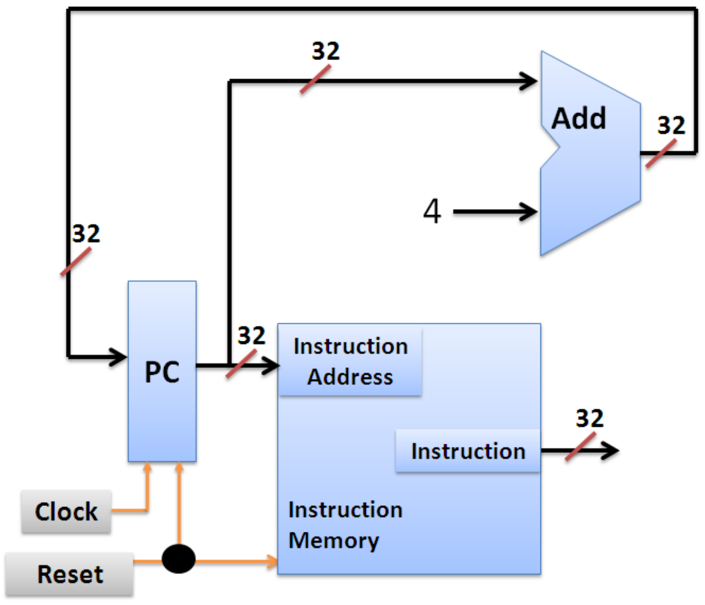
# Lab 1: Design of Instruction Fetch and ALU

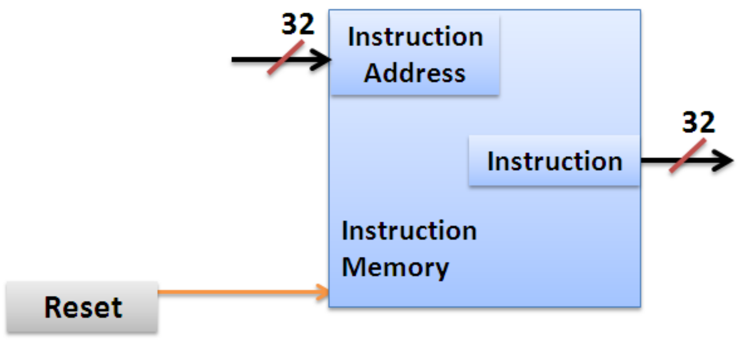
## <u>Exercise 1.1: Implement Instruction Fetch Unit in Verilog.</u>

The aim of this exercise is to implement the Behavioral model for instruction fetch unit. Instruction fetch is the first stage of any processor. The instruction fetch unit for MIPS processor consists of three main units: (1) A 32-bit program counter (PC) register also called Instruction register which holds the address of instruction that is to be fetched. (2) A byte addressable **BigEndian** Instruction Memory which accepts a 32-bit address and gives as output a 32-bit instruction code. (3) An adder to increment the contents of PC to point to next instruction.  The instruction fetch unit contains two inputs a clock and a reset. When reset becomes zero PC should be initialized to 0 and the Instruction memory should be initialized with specific values. When the reset is not zero the instruction fetch unit should output a 32-bit instruction code corresponding to the address in PC at positive edge of clock. The PC should be incremented to point to next instruction after each clock cycle. The figure below shows the block level diagram of the instruction fetch unit.



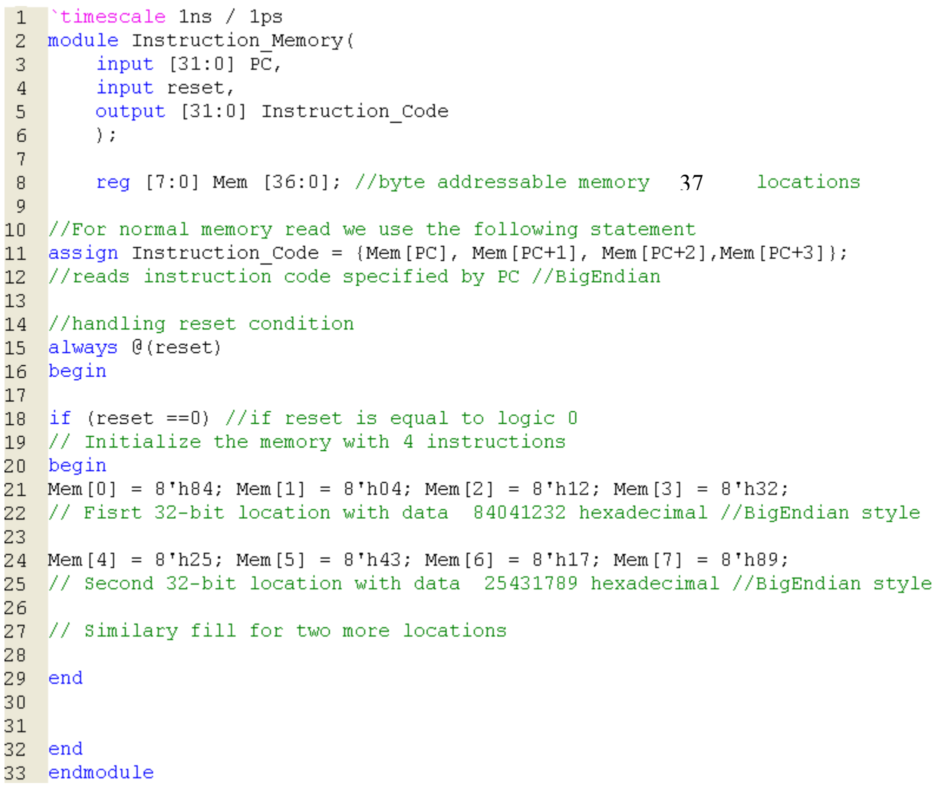
The instruction fetch is implemented in stages. The first stage is to implement the Instruction memory.

### <u>Exercise 1.1.1: Implement Instruction memory in Verilog.</u>



The instruction memory has two inputs a 32-bit Input coming from PC and a 1 bit reset. It has one 32-bit output indicating the output instruction code. According to the specifications when reset is logic 0 the Instruction memory should be initialized with specific data. This initialization necessary to write the instruction codes into the memory. When reset is logic 1 the instruction memory should output the 32-bit instruction code corresponding to the 32-bit input address. The partial code for the Instruction memory is shown below.  Please read the comments for a better understanding of the design.

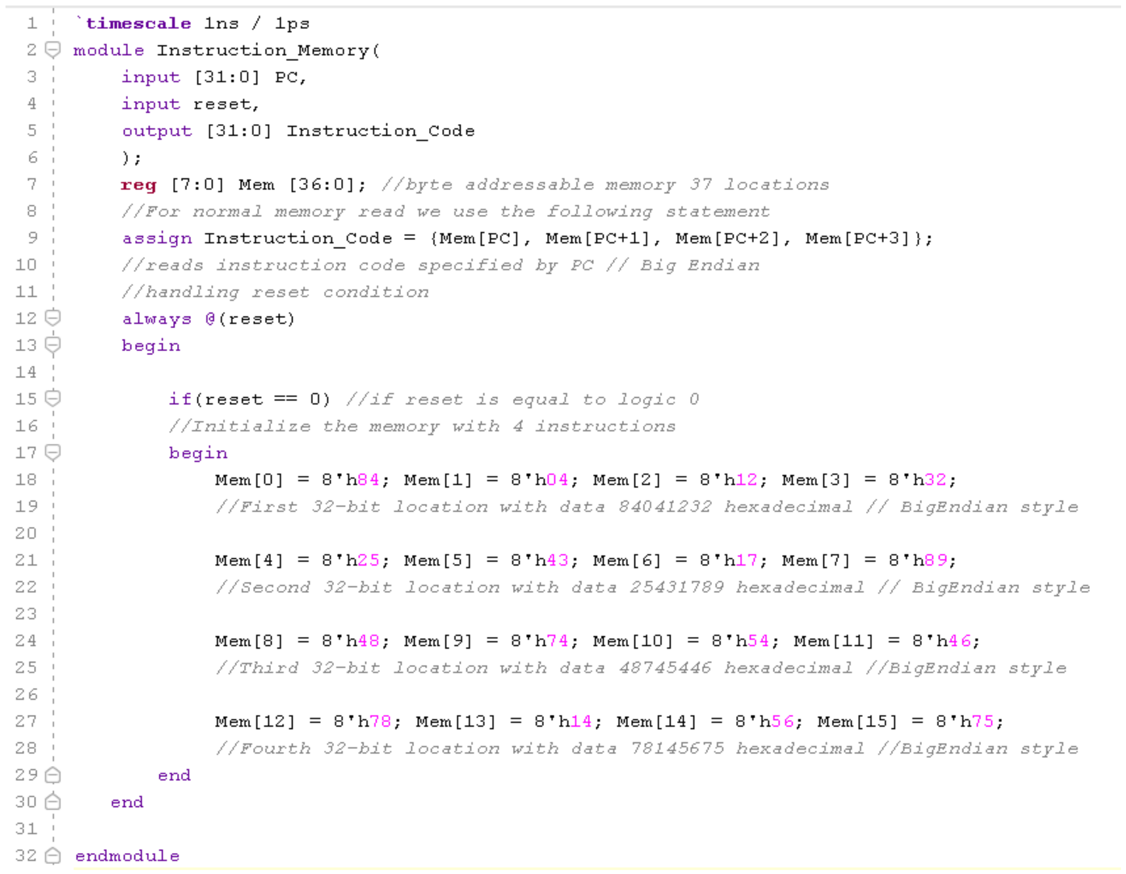
**Partial code: Instruction_Memory.v**



```
`timescale 1ns / 1ps
module Instruction_Memory(
    input [31:0] PC,
    input reset,
    output [31:0] Instruction_Code
    );

    reg [7:0] Mem [36:0]; //byte addressable memory  37    locations

//For normal memory read we use the following statement
assign Instruction_Code = {Mem[PC], Mem[PC+1], Mem[PC+2],Mem[PC+3]};
//reads instruction code specified by PC //BigEndian

//handling reset condition
always @(reset)
begin

if (reset ==0) //if reset is equal to logic 0
// Initialize the memory with 4 instructions
begin
Mem[0] = 8'h84; Mem[1] = 8'h04; Mem[2] = 8'h12; Mem[3] = 8'h32;
// Fisrt 32-bit location with data  84041232 hexadecimal //BigEndian style

Mem[4] = 8'h25; Mem[5] = 8'h43; Mem[6] = 8'h17; Mem[7] = 8'h89;
// Second 32-bit location with data  25431789 hexadecimal //BigEndian style

// Similary fill for two more locations

end


end
endmodule
```

**reg [7:0] Mem [36:0]; defines byte addressable memory with 37 locations.**
**Copy the image of completed Instruction memory module?**

Answer:   The screenshot is pasted on the next page.

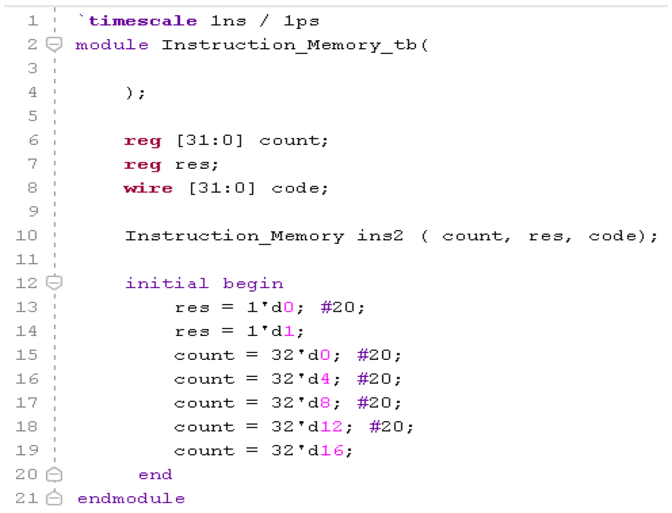


```verilog
`timescale 1ns / 1ps
module Instruction_Memory(
    input [31:0] PC,
    input reset,
    output [31:0] Instruction_Code
    );
    reg [7:0] Mem [36:0]; //byte addressable memory 37 locations
    //For normal memory read we use the following statement
    assign Instruction_Code = {Mem[PC], Mem[PC+1], Mem[PC+2], Mem[PC+3]};
    //reads instruction code specified by PC // Big Endian
    //handling reset condition
    always @(reset)
    begin

        if(reset == 0) //if reset is equal to logic 0
        //Initialize the memory with 4 instructions
        begin
            Mem[0] = 8'h84; Mem[1] = 8'h04; Mem[2] = 8'h12; Mem[3] = 8'h32;
            //First 32-bit location with data 84041232 hexadecimal // BigEndian style

            Mem[4] = 8'h25; Mem[5] = 8'h43; Mem[6] = 8'h17; Mem[7] = 8'h89;
            //Second 32-bit location with data 25431789 hexadecimal // BigEndian style

            Mem[8] = 8'h48; Mem[9] = 8'h74; Mem[10] = 8'h54; Mem[11] = 8'h46;
            //Third 32-bit location with data 48745446 hexadecimal //BigEndian style

            Mem[12] = 8'h78; Mem[13] = 8'h14; Mem[14] = 8'h56; Mem[15] = 8'h75;
            //Fourth 32-bit location with data 78145675 hexadecimal //BigEndian style
       end
    end

endmodule
```

**<u>Exercise 1.1.2 Write the TestBench to test the functionality of the Instruction Memory Module. (As part of your testbench enable reset initially and then give different values of PC)</u>**

**Copy the image of Testbench code?**

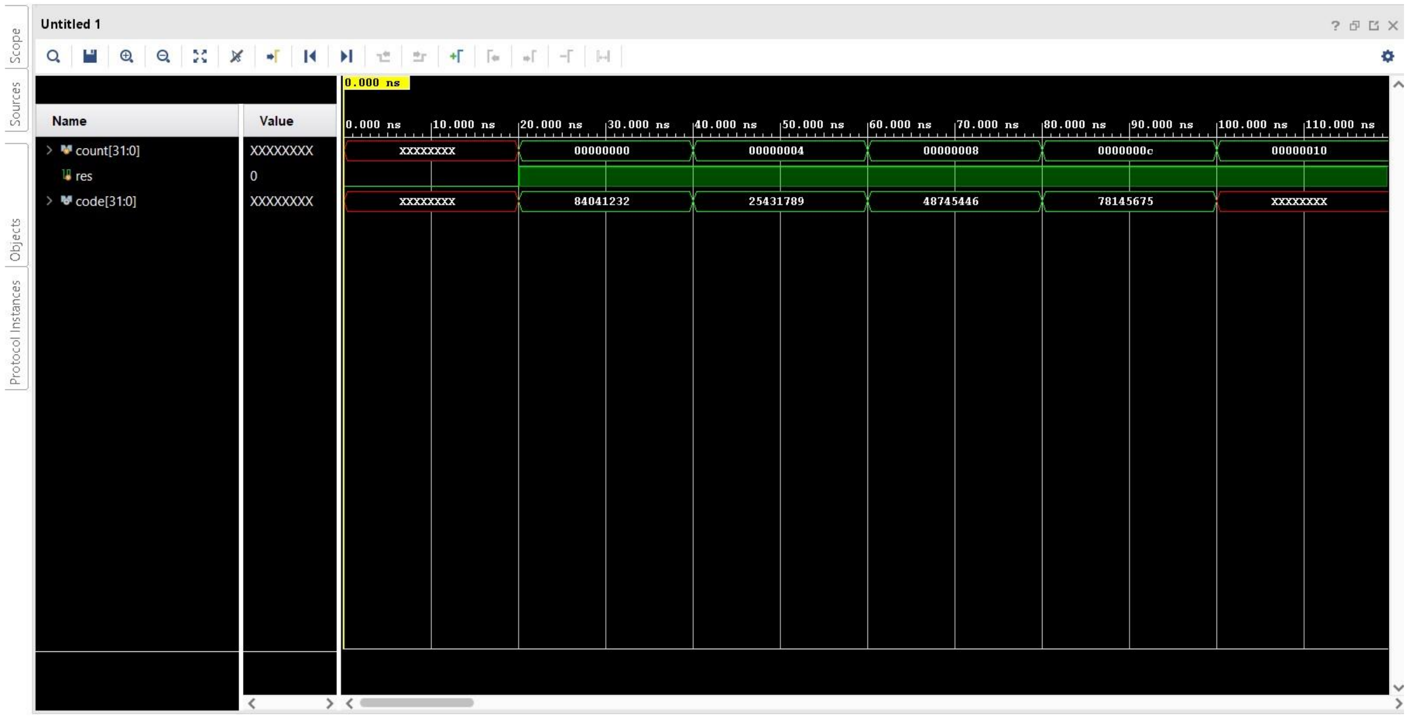
Answer:



```verilog
`timescale 1ns / 1ps
module Instruction_Memory_tb(

    );

    reg [31:0] count;
    reg res;
    wire [31:0] code;

    Instruction_Memory ins2 ( count, res, code);

    initial begin
        res = 1'd0; #20;
        res = 1'd1;
        count = 32'd0; #20;
        count = 32'd4; #20;
        count = 32'd8; #20;
        count = 32'd12; #20;
        count = 32'd16;
     end
endmodule
```

**Copy the image of waveform window that is generated for your Testbench? (Change display radix to Hexadecimal)**

**Answer:**



**What changes will you make to the line 11 of the Instruction_Memory module if the memory is of LittleEndian type?**

Answer: *assign Instruction_Code = {Mem[PC+3], Mem[PC+2], Mem[PC+1], Mem[PC]};*

**What changes will you make to the line 21 of the Instruction_Memory module if the memory is of LittleEndian type?**

Answer: *Mem[0] = 8'h32; Mem[1] = 8'h12; Mem[2] = 8'h04; Mem[3] = 8'h84;*

**The data read out from Instruction memory is 32-bits. Then what is the reason for defining Mem as [7:0] Mem [No. of locations] instead of [31:0] Mem [No. of locations]**

Answer: *The instruction memory of MIPS has a byte addressable memory, meaning that it supports accessing no more than individual bytes in the memory, or 8-bits. Each address identifies a single byte of storage.*

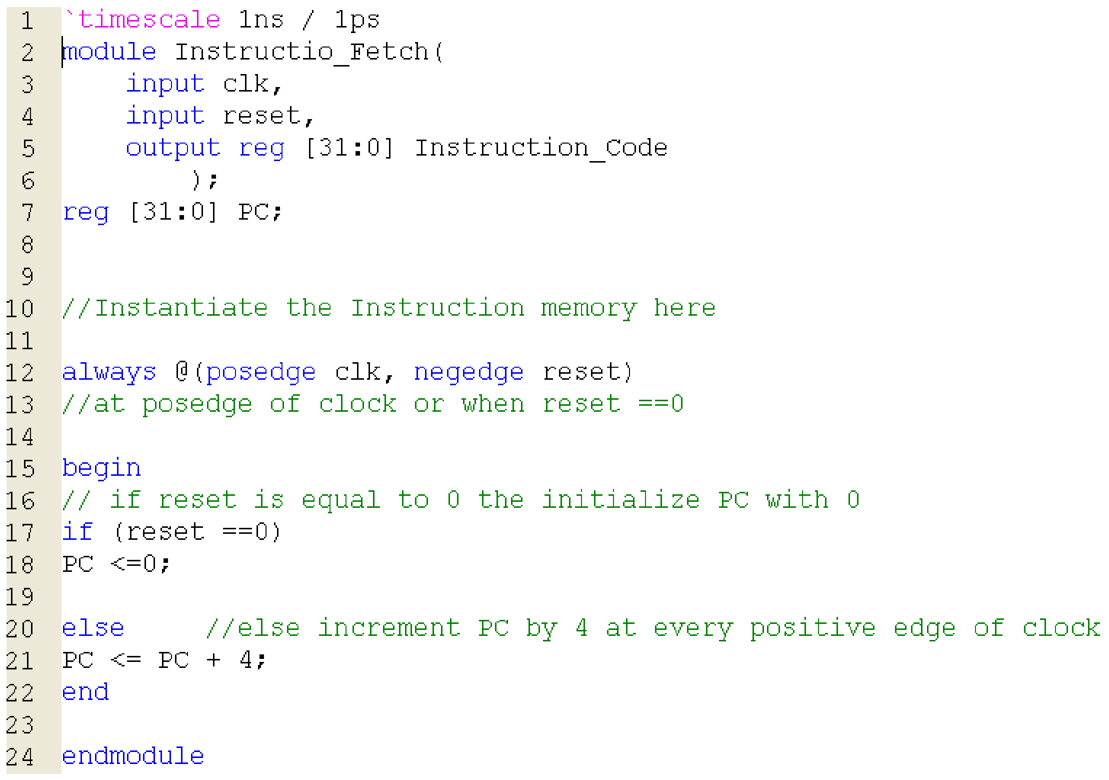
**Find out and list other ways of initializing the memory.**

Answer: *Verilog allows to initialize memory from a text file with either hex or binary values. The respective commands are:*

- `$readmemh`
- `$readmemb`
- `Use file IO system task, such as $fscanf and $fopen`

**<u>Exercise 1.1.3 Implement and test (using test bench) the Instruction fetch unit by instantiating the Instruction memory block. (As part of Instruction fetch test bench enable reset initially and then generate continuous clock).</u>**

The instruction fetch unit has clock and reset pins as inputs and Instruction code as output. Internally it has a PC register which holds the address of current instruction. It also has an adder to compute PC + 4. The partial code for instruction fetch unit (without instantiation of instruction memory) is shown below.



```
`timescale 1ns / 1ps
module Instructio_Fetch(
    input clk,
    input reset,
    output reg [31:0] Instruction_Code
        );
reg [31:0] PC;


//Instantiate the Instruction memory here

always @(posedge clk, negedge reset)
//at posedge of clock or when reset ==0

begin
// if reset is equal to 0 the initialize PC with 0
if (reset ==0)
PC <=0;

else     //else increment PC by 4 at every positive edge of clock
PC <= PC + 4;
end

endmodule
```

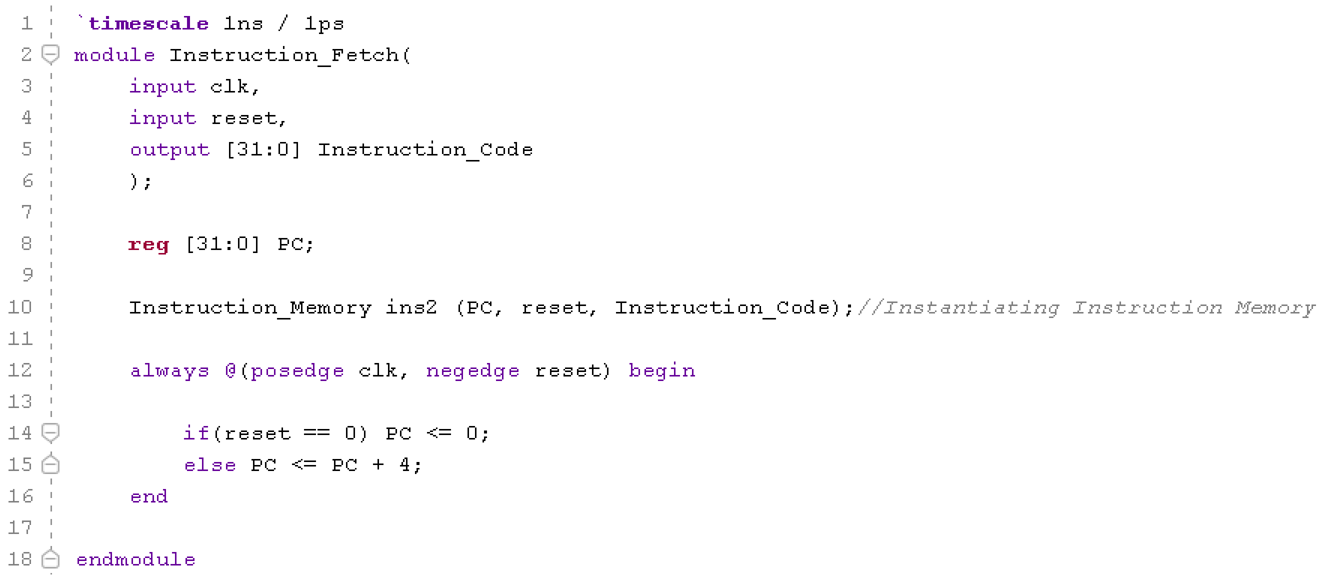
**There is an error in the code above. What is the error and what should be done to solve this error?**

Answer: *Instruction_Code is declared as a **reg** type variable, whereas in the Instruction memory module, this was declared as a **wire** type variable. Similar error comes with PC variable, where it is declared as a **wire** in Instruction memory module, but declared as a **reg** here. Any variables inside a* *procedural block and involved in the left-hand assignment must be declared as a* ***reg*** *type variable. Otherwise, the variable should be declared as a* ***wire*** *type.*

*The error can be fixed by declaring Instruction_Code variable as a wire type (remove* ***reg*** *in line 5) and by declaring PC as a* ***reg*** *type variable in Instruction_Memory module.*

**Copy the image of completed Instruction fetch module?**

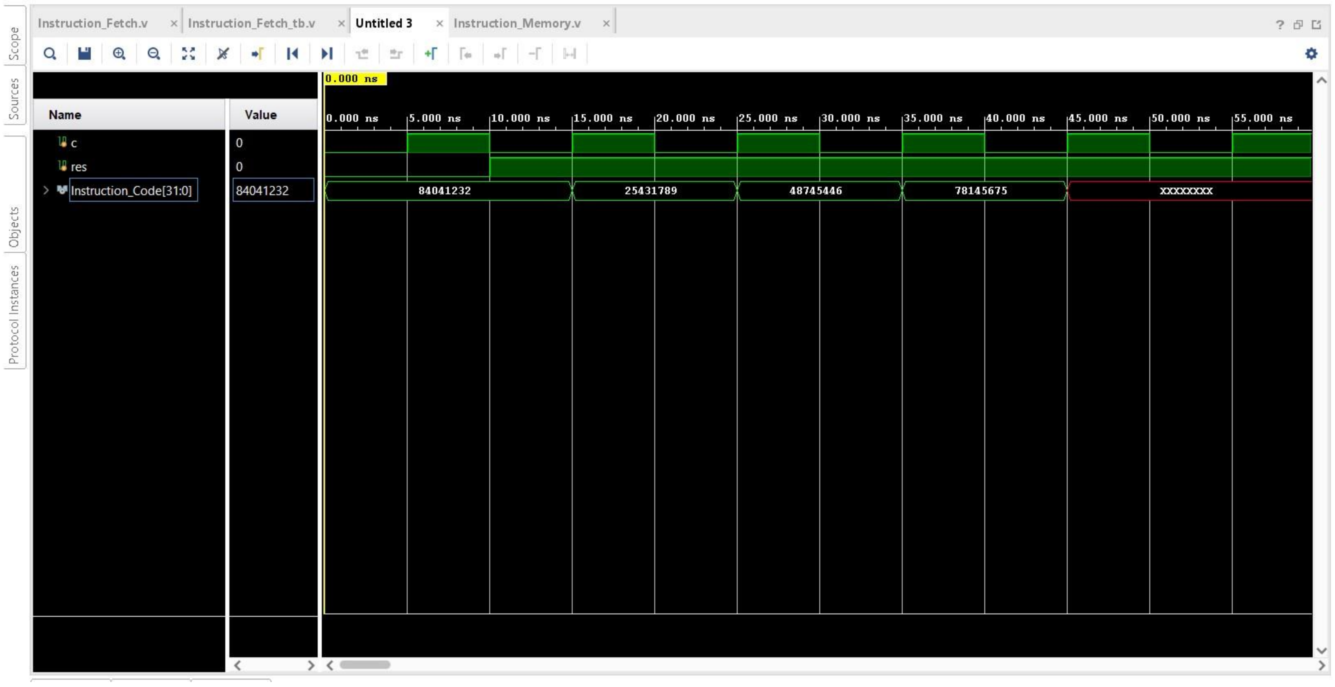
**Answer:**



```
1   `timescale 1ns / 1ps
2   module Instruction_Fetch(
3       input clk,
4       input reset,
5       output [31:0] Instruction_Code
6       );
7
8       reg [31:0] PC;
9
10      Instruction_Memory ins2 (PC, reset, Instruction_Code);//Instantiating Instruction Memory
11
12      always @(posedge clk, negedge reset) begin
13
14          if(reset == 0) PC <= 0;
15          else PC <= PC + 4;
16      end
17
18  endmodule
```

**Copy the image of waveform window that is generated for your Testbench? (Change display radix to Hexadecimal).**

**Answer:**



| Name | Value |
|---|---|
| c | 0 |
| res | 0 |
| Instruction_Code[31:0] | 84041232 |

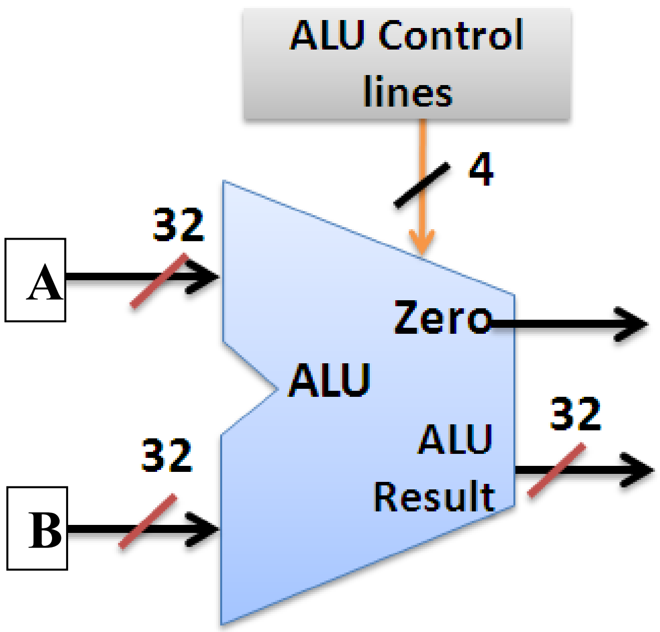
Instruction_Code[31:0]: 84041232, 25431789, 48745446, 78145675, XXXXXXXX

**Exercise 1.2 Design of simple ALU.**

ALU (Arithmetic and Logical unit) is an important component of any processor. ALU performs different arithmetic and logical operations depending on the control lines. In this section you will be implementing a simple ALU for a given set of specifications.

**ALU Specifications:**

This ALU has two 32-bit operands and 4 control lines as inputs. It has two outputs a 32-bit ALU result and a Zero indicator which becomes logic 1 if and only if the 32-bit ALU result is Zero. The ALU has to perform different functions according to the value of 4 control lines. Block diagram of ALU along with the mapping of control lines to functions performed is shown below.



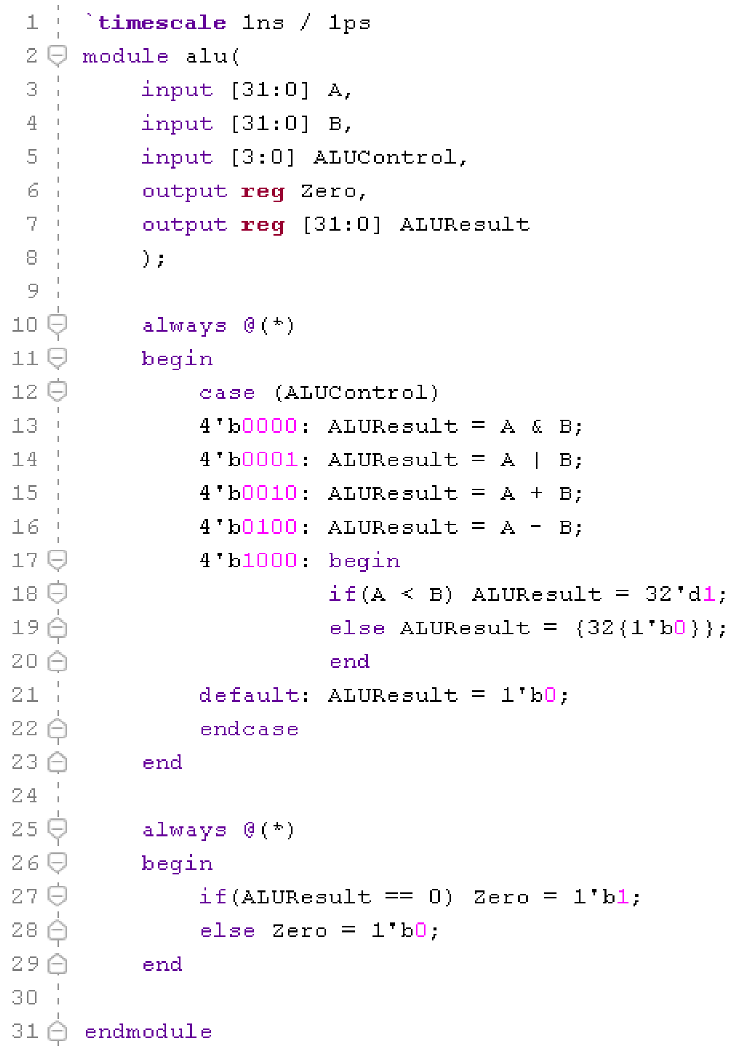
| ALU Control lines | Function |
|---|---|
| 0000 | Bitwise-AND |
| 0001 | Bitwise-OR |
| 0010 | Add (A+B) |
| 0100 | Subtract (A-B) |
| 1000 | Set on less than |

*Set on less than (ALU Result = 1 if A<B else ALU Result =0)

**Implement the ALU for above specifications in Verilog using behavioral modeling (Hint: Use Case statement).**

**Paste the image of your Verilog code.**

Answer: Screenshot is pasted on the next page.



```
`timescale 1ns / 1ps
module alu(
    input [31:0] A,
    input [31:0] B,
    input [3:0] ALUControl,
    output reg Zero,
    output reg [31:0] ALUResult
    );

    always @(*)
    begin
        case (ALUControl)
        4'b0000: ALUResult = A & B;
        4'b0001: ALUResult = A | B;
        4'b0010: ALUResult = A + B;
        4'b0100: ALUResult = A - B;
        4'b1000: begin
                 if(A < B) ALUResult = 32'd1;
                 else ALUResult = {32{1'b0}};
                 end
        default: ALUResult = 1'b0;
        endcase
    end

    always @(*)
    begin
        if(ALUResult == 0) Zero = 1'b1;
        else Zero = 1'b0;
    end

endmodule
```

**<u>Write the test bench to test the ALU. Your test bench should have 7 different test patterns as mentioned below. (Assume test pattern changes after every 20 time units)</u>**

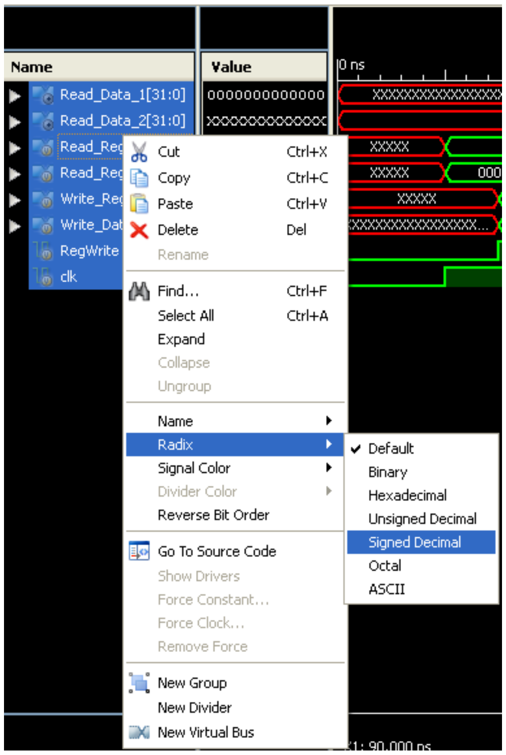
Test case 1: A = 23, B = 42, ALUContol = 4`b0000.
Test case 2: A = 23, B = 42, ALUContol = 4`b0001.
Test case 3: A = 23, B = 42, ALUContol = 4`b0010.
Test case 4: A = 23, B = 42, ALUContol = 4`b0100.
Test case 5: A = 23, B = 42, ALUContol = 4`b1000.
Test case 6: A = 42, B = 23, ALUContol = 4`b1000.
Test case 7: A = 42, B = 23, ALUContol = 4`b0100.

**Paste the image of your test bench with above test cases.**

Answer: Screenshot is pasted on the next page.

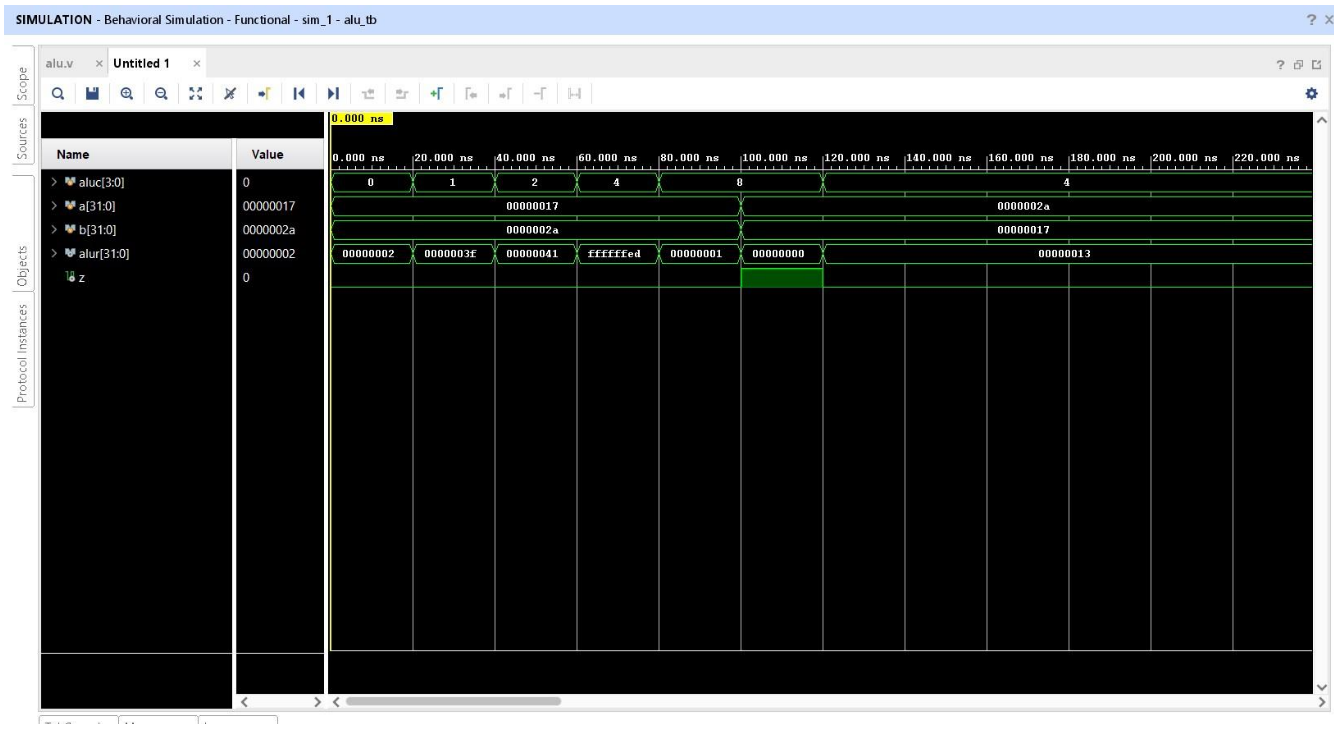
```
`timescale 1ns / 1ps
module alu_tb(

    );

    reg [3:0] aluc;
    reg [31:0] a, b;
    wire [31:0] alur;
    wire z;

    alu ins1 ( a, b, aluc, z, alur );

    initial begin
    a = 32'd23; b = 32'd42; aluc = 4'b0000; #20; // Test case 1
    a = 32'd23; b = 32'd42; aluc = 4'b0001; #20; // Test case 2
    a = 32'd23; b = 32'd42; aluc = 4'b0010; #20; // Test case 3
    a = 32'd23; b = 32'd42; aluc = 4'b0100; #20; // Test case 4
    a = 32'd23; b = 32'd42; aluc = 4'b1000; #20; // Test case 5
    a = 32'd42; b = 32'd23; aluc = 4'b1000; #20; // Test case 6
    a = 32'd42; b = 32'd23; aluc = 4'b0100; #20; // Test case 7
   end

endmodule
```

For easy viewing of the testbench waveforms where signals are multi bit, you can change the view in simulation window to decimal/hexadecimal mode. Right click on all the input output signals select Radix and then click on signed decimal/hexadecimal.



**Copy the image of waveform window that is generated for your Testbench? (Change display radix to Hexadecimal).**

**Answer:**



SIMULATION - Behavioral Simulation - Functional - sim_1 - alu_tb

| Name | Value |
|---|---|
| aluc[3:0] | 0 |
| a[31:0] | 00000017 |
| b[31:0] | 0000002a |
| alur[31:0] | 00000002 |
| z | 0 |

**How many different functions can be implemented by ALU with 4-control lines?**

Answer: *4 control lines mean 4-bits, that means $2^4$ = 16 different functions can be implemented.*