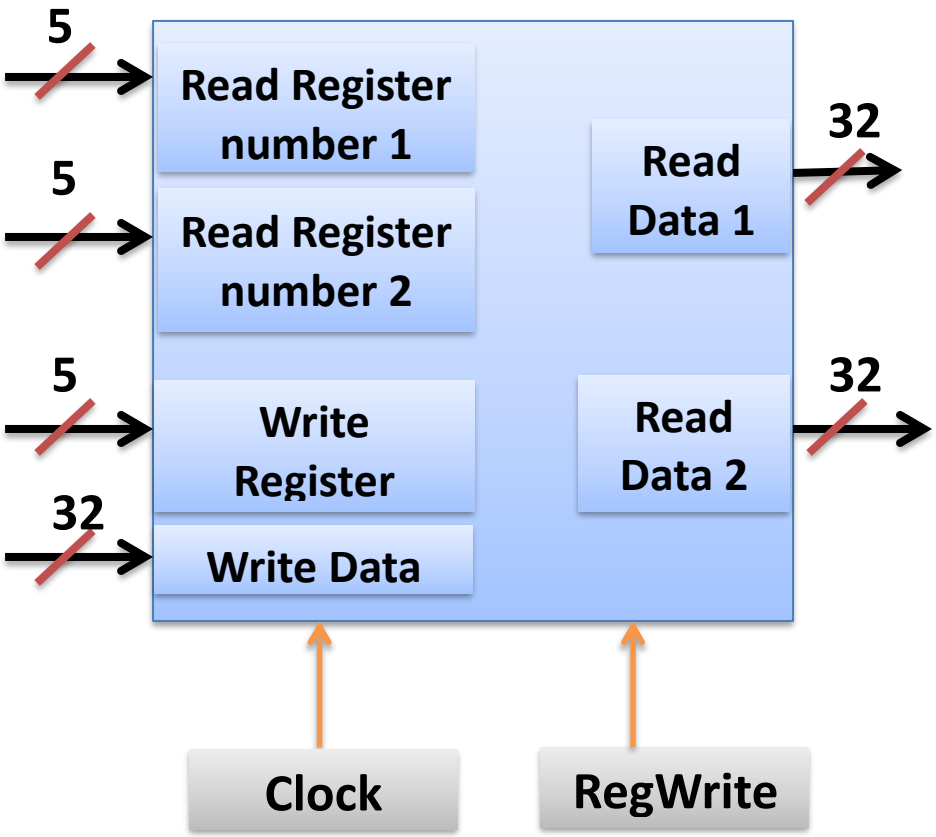
Lab 2: Implementation of Register File and Partial Datapath

A register file is one of the important components of the MIPS data path. It can read two registers and write in to one register. The MIPS register file contains total of 32 32-bit registers. Hence 5-bits are used to specify the register numbers that are to be read or written.

Register Read: Register file always outputs the contents of the register corresponding to read register numbers specified. **Reading a register is not dependent on any other signals.**
Register Write: Register writes are controlled by a control signal **RegWrite**. Additionally the register file has a **clock signal**. The write should happen if **RegWrite signal is made 1 and if there is positive edge of clock**.



Exercise 2.1 Implement MIPS register file with above specifications using Behavioral modeling. Test the Register file using Testbench.

Hint: Registers are similar to memory. So 32 32-bit registers is like 32 memory locations with 32 bits each.
Syntax to read from memory is:
RegData = RegMemory[RegAddress]; (if used inside always block)

assign RegData = RegMemory[RegAddress]; (if used outside always block)
Where **RegData** is the data read from address location specified by **RegAddress**.

Syntax to write in to memory location is:
RegMemory[RegAddress] = NewData; (if used inside always block), where **NewData** is written in to the **RegMemory** to address location specified by **RegAddress.** Since we define RegMemory as Reg type, the above assignment is not done outside always block.

You can use the module definition shown below for register file. Please note that the following is just the module definition which specifies input, output ports. **You have to describe the behavior of Register file to make the design complete.** Also if any of the outputs are to be assigned inside the always block then you have to additionally define those outputs as **reg**.

```
module Register_file(
    input [4:0] Read_Reg_Num_1,
    input [4:0] Read_Reg_Num_2,
    input [4:0] Write_Reg_Num,
    input [31:0] Write_Data,
    output [31:0] Read_Data_1,
    output [31:0] Read_Data_2,
    input RegWrite,
    input clk
    );
 //Add reset as one more input to initialize the register file with some default values
//Define 32, 32 bit RegMemory here
// your behavioral design comes here
```

Testbench:
You can test your Register file with the following test pattern. (below code comes after instantiation of the test module). The test bench contains three initial blocks one for generation of signals to check writing in to register 0 (when Register_Num is 00000) and register 1(Register_Num is 00001), Second initial block to read register 0 and register 1 and register 2. The third initial block is used to generate the clock. If you have reset as input, you have to add one more initial block where the reset (active low) is low for some time initially and then changed to 1.

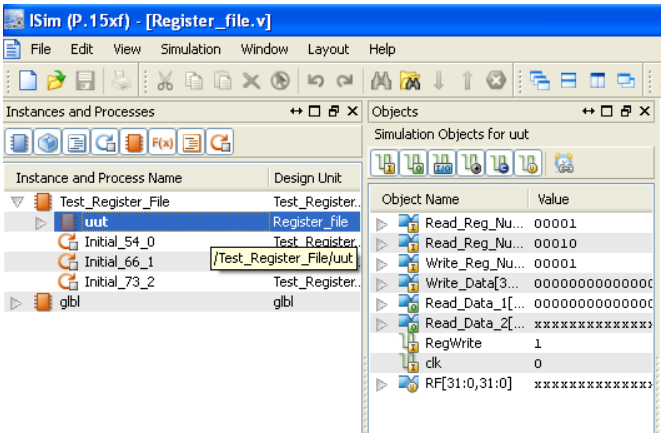
```
//Write Data in to registers
   initial begin
   RegWrite = 0;
   #15;
   RegWrite = 1;   Write_Data = 20;   Write_Reg_Num = 0;
   #10;
   RegWrite = 1;   Write_Data = 30;   Write_Reg_Num = 1;
   #10;
   RegWrite = 1;   Write_Data = 30;   Write_Reg_Num = 1;
   #10;
   end

//Clock generation
initial begin
clk =0;
repeat (8)
#10 clk = ~ clk;    #10 $finish;
end

//Reading Registers
initial begin
 #10   Read_Reg_Num_1 = 0;      Read_Reg_Num_2 = 0;
 #15   Read_Reg_Num_1 = 0;      Read_Reg_Num_2 = 1;
 #10   Read_Reg_Num_1 = 1;      Read_Reg_Num_2 = 2;
end
endmodule
```

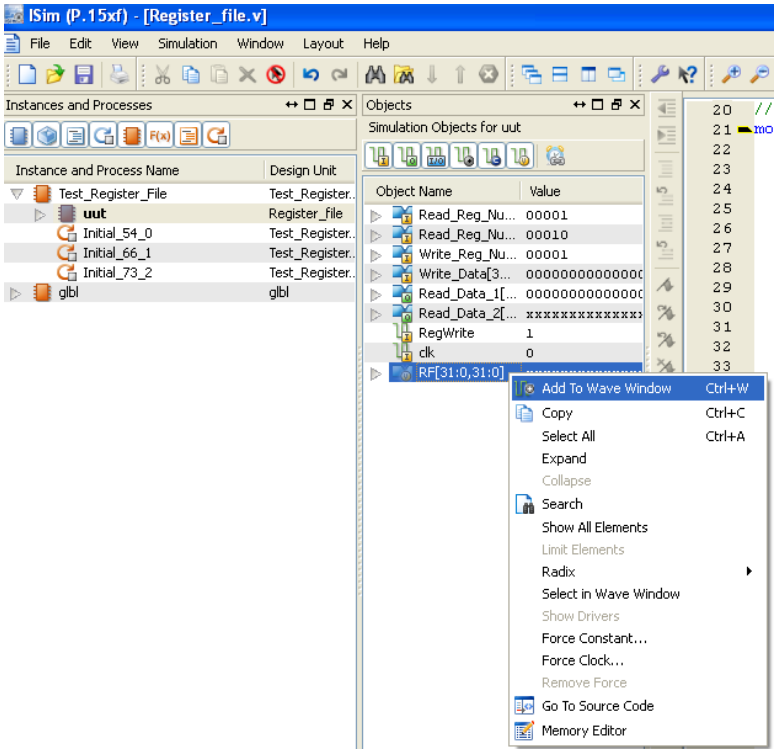
For easy viewing of the testbench waveforms, you can change the view in simulation window to decimal/Hexadecimal mode. Right click on all the input output signals select Radix and then click on signed decimal/Hexadecimal.

Additional Information:

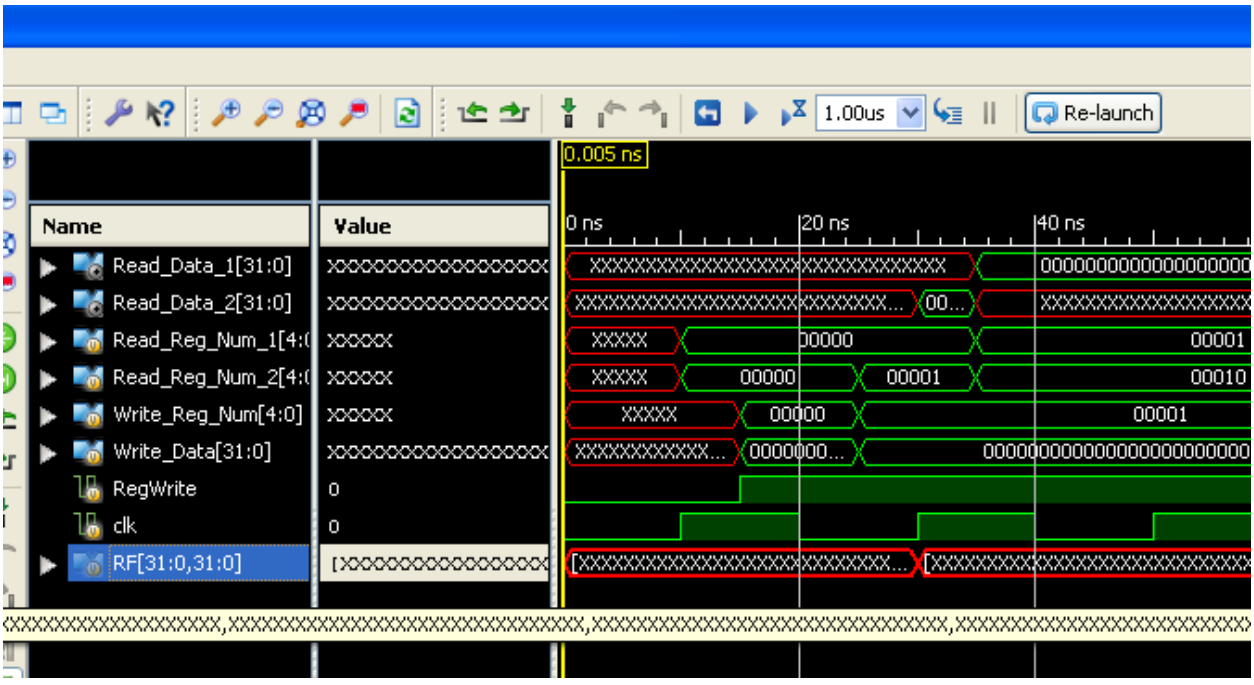
For verifying the design you might have to monitor the internal signals which could be defined as wires or Reg. In the above Register file design **RegMemory** will be defined as reg and is neither input nor output. The steps for adding internal signal (in this case RegMemory) into the wave form window is shown below.

1. To check the internal signals of a module (e.g. Register File), Click on the corresponding module in the ISim window (after you run the test bench) and then right click → Add to Wave window.





2. Then Click on Re-launch Icon. Click 'No' when prompted to save the waveform. This will run the testbench with the newly added internal signal.



Q2.1. Did you get any errors during the check syntax of design file or when simulating the testbench? If yes then, how did you solve the error?

Answer: No error faced; perfect simulation achieved.

Q2.2. Copy the image of final working code of the register file here.

Answer:

```
1  `timescale 1ns / 1ps
2  module Register_file(
3      input [4:0] Read_Reg_Num_1,
4      input [4:0] Read_Reg_Num_2,
5      input [4:0] Write_Reg_Num,
6      input [31:0] Write_Data,
7      output [31:0] Read_Data_1,
8      output [31:0] Read_Data_2,
9      input RegWrite,
10     input clk
11     );
12
13     reg [31:0] RegMemory [31:0];
14
15     assign Read_Data_1 = RegMemory[Read_Reg_Num_1];
16     assign Read_Data_2 = RegMemory[Read_Reg_Num_2];
17
18     always @(posedge clk) begin
19         if(RegWrite == 1'b1) RegMemory[Write_Reg_Num] = Write_Data;
20     end
21 endmodule
```

Q2.3. What changes will you make to the Register_File module if you want to give an option for preloading the register file with some default values?

Answer: We can give a *reset* signal to the Register File, such that when reset is 0 (active low), some default values will get loaded into the Register File.

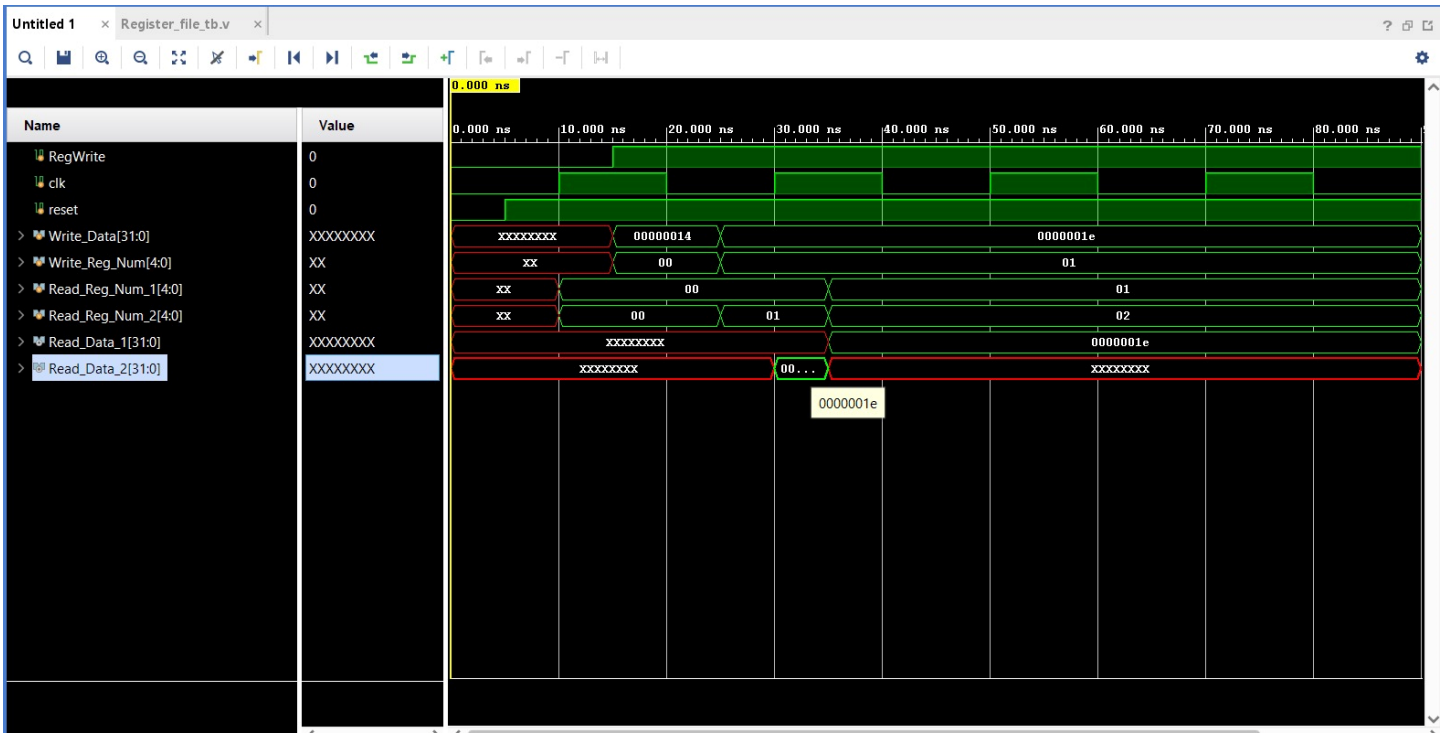
Q2.4. Modify the Register_File module by including reset signal as input. Add an always block to your register file. This always block checks for reset signal and initializes the register memory (RegMemory) with some default values?

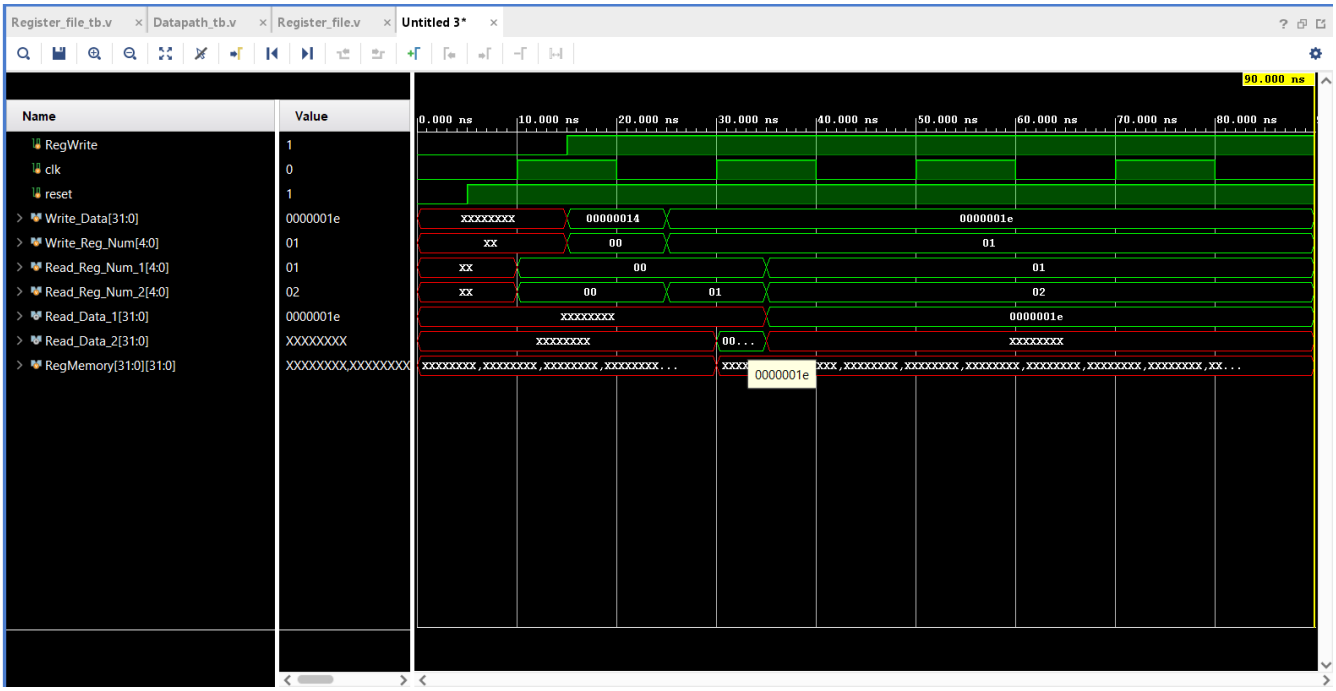
Answer:

```
`timescale 1ns / 1ps
module Register_file(
    input [4:0] Read_Reg_Num_1,
    input [4:0] Read_Reg_Num_2,
    input [4:0] Write_Reg_Num,
    input [31:0] Write_Data,
    output [31:0] Read_Data_1,
    output [31:0] Read_Data_2,
    input RegWrite,
    input clk,
    input reset
    );

    reg [31:0] RegMemory [31:0];

    assign Read_Data_1 = RegMemory[Read_Reg_Num_1];
    assign Read_Data_2 = RegMemory[Read_Reg_Num_2];

    always @(posedge clk) begin
        if(RegWrite == 1'b1) RegMemory[Write_Reg_Num] = Write_Data;
    end

    always @(*) begin
    if(reset == 1'b0) begin
         RegMemory[7] = 32'd56;
         RegMemory[3] = 32'd16;
         RegMemory[4] = 32'd69;
         RegMemory[6] = 32'd10;
         end
    end
endmodule
```

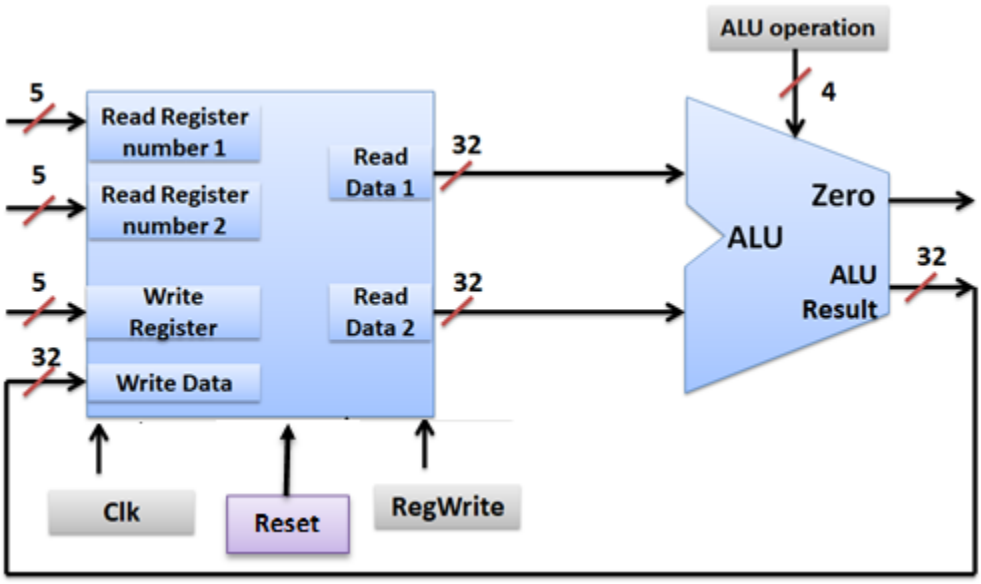
Q2.5. Copy the image of waveform window that is generated for the given Testbench? Waveforms should be in decimal/ hexadecimal view.

Answer:





Exercise 2.2 Implement partial Datapath of a MIPS like processor (Shown below). This Datapath has Read Register number 1, Read Register number 2, Write Register, ALU Control lines (ALU Operation) bits, Clk, Reset, and RegWrite as inputs. This Datapath has only one output i.e. Zero. To implement this Datapath first create another module called Datapath. Instantiate (similar to calling a function) the Register file module and ALU Module (Reuse the ALU which was implemented in Lab 1) in Datapath module. Make internal connections as per the Datapath given. (e.g. ReadData1 is connected to first input of ALU). Please note that all intermediate signals should be defined as wire with appropriate bit widths.



Q2.6. Copy the image of Verilog code of the above Datapath.

Answer:

```
`timescale 1ns / 1ps
module Datapath(
    input [4:0] Read_Reg_Num_1,
    input [4:0] Read_Reg_Num_2,
    input [4:0] Write_Reg,
    input [3:0] ALUControl,
    input clk,
    input reset,
    input RegWrite,
    output reg ALUZero
    );

    wire [31:0] input_1, input_2;
    wire [31:0] ALUResult;

    Register_file ins_reg_fil ( Read_Reg_Num_1, Read_Reg_Num_2, Write_Reg, ALUResult, input_1, input_2, RegWrite, clk, reset);
    alu ins_alu (input_1, input_2, ALUControl, ALUZero, ALUResult);

endmodule
```

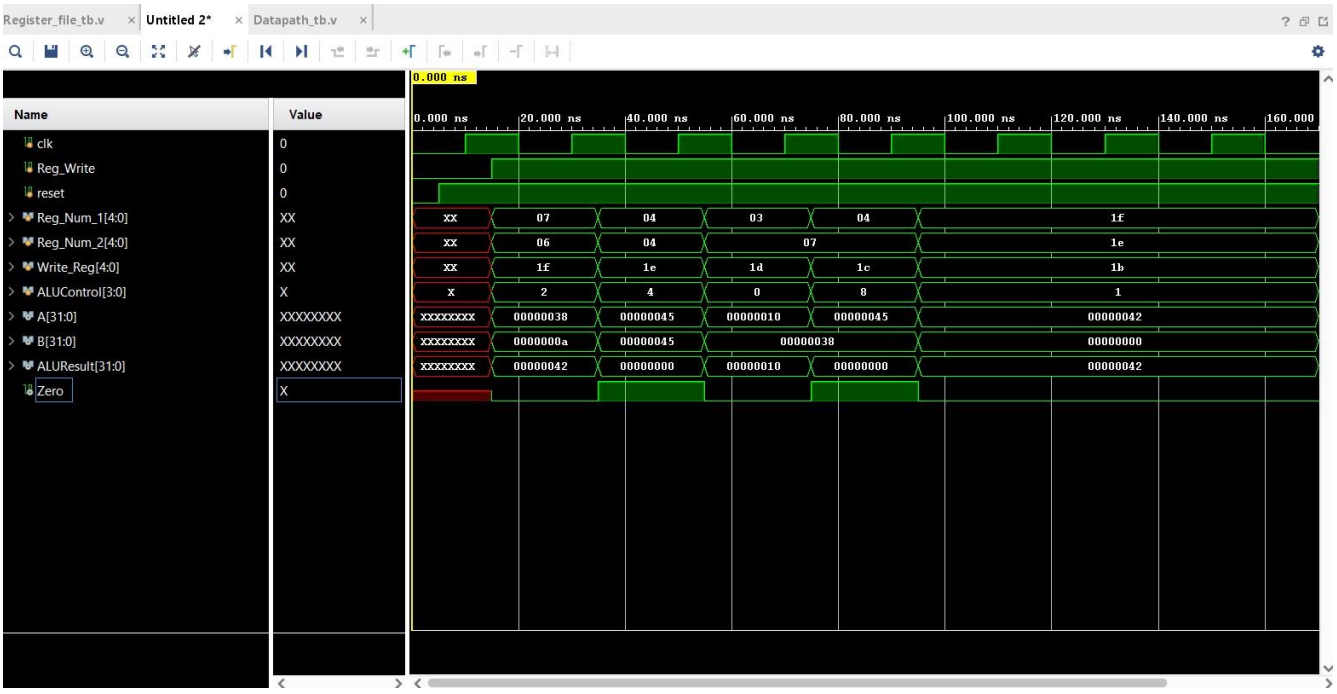
Q2.7. Write an appropriate test bench to test the above data path. Copy the test bench below (Hint: You should supply new register numbers after the positive edge of Clk or before positive edge of Clk to avoid confustion)

Answer:

```
`timescale 1ns / 1ps
module Datapath_tb(
    );
    reg clk, Reg_Write, reset;
    reg [4:0] Reg_Num_1, Reg_Num_2, Write_Reg;
    reg [3:0] ALUControl;
    wire [31:0] ALUResult;
    wire [31:0] A, B;
    wire Zero;

    Register_file ins1 (Reg_Num_1, Reg_Num_2, Write_Reg, ALUResult, A, B, Reg_Write, clk, reset);
    alu ins2 (A, B, ALUControl, Zero, ALUResult);

    initial begin
    clk = 0;
    repeat (16)
    #10 clk = ~clk; #10 $finish;
    end

    initial begin
    Reg_Write = 0; #15; Reg_Write = 1;
    Reg_Num_1 = 7; Reg_Num_2 = 6; Write_Reg = 31; ALUControl = 4'b0010; #20;
    Reg_Num_1 = 4; Reg_Num_2 = 4; Write_Reg = 30; ALUControl = 4'b0100; #20;
    Reg_Num_1 = 3; Reg_Num_2 = 7; Write_Reg = 29; ALUControl = 4'b0000; #20;
    Reg_Num_1 = 4; Reg_Num_2 = 7; Write_Reg = 28; ALUControl = 4'b1000; #20;
    Reg_Num_1 = 31; Reg_Num_2 = 30; Write_Reg = 27; ALUControl = 4'b0001; #20;
    end

    initial begin
    reset = 0; #5;
    reset = 1;
    end

endmodule
```

Q2.8. Simulate the above test bench and paste the waveforms below.

Answer:



Q2.9. List the concepts you learnt from this experiment (Conclusions/Observations)

Answer: I implemented a partial data path of a MIPS processor by instantiating two modules, Register_File and ALU. I connected both the modules and ran the simulation under testbench environment. By giving the reset signal, the Register File was initialized with some default values. After reading these register values and performing ALU operations according to ALU control lines, the result was stored back in the write register by the Register File itself, in memory. Therefore, I learnt how to create a partial data path flow using Register File by reading and writing into the memory, and a simple ALU unit in Verilog.