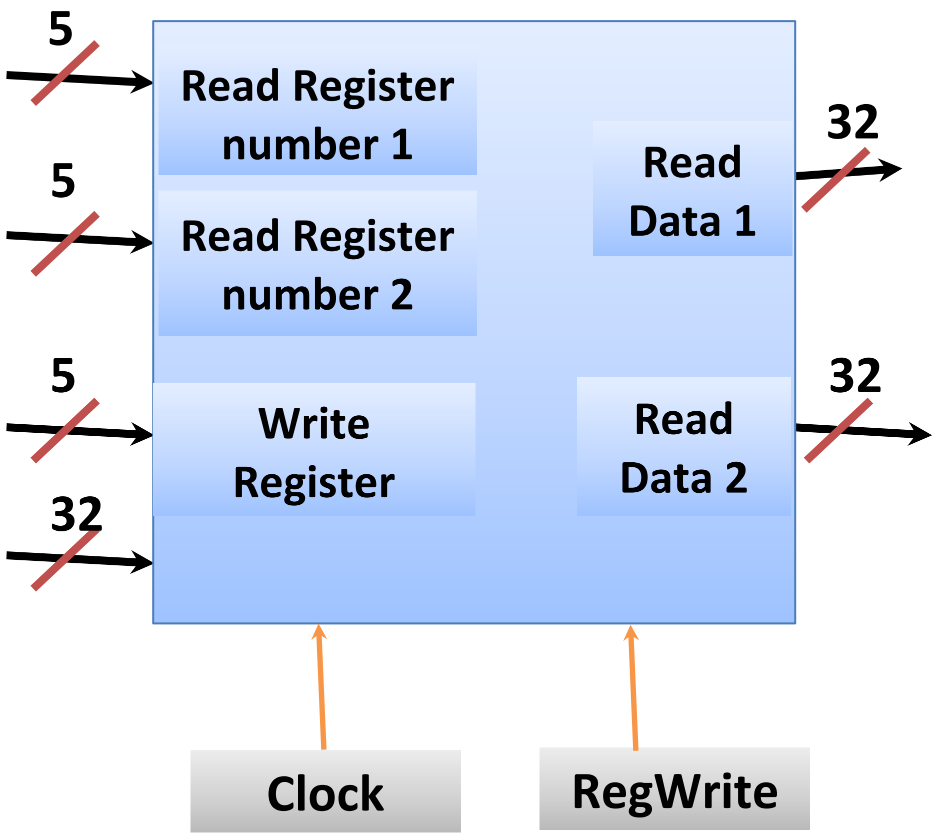
Lab 2: Implementation of Register File and Partial Datapath

A register file is one of the important components of the MIPS data path. It can read two registers and write into one register. The MIPS register file contains a 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: The 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 the **RegWrite signal is made 1 and if there is a positive edge of the clock**.



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

Hint: Registers are similar to memory. So 32 32-bit registers are 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 the address location specified by **RegAddress**.

Syntax to write into memory location is:
RegMemory[RegAddress] = NewData; (if used inside always block), where **NewData** is written into the **RegMemory** to address the 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 the register file. Please note that the following is just the module definition which specifies input, and output ports. **You have to describe the behavior of the 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. (The below code comes after the instantiation of the test module). The test bench contains three initial blocks one for the 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.

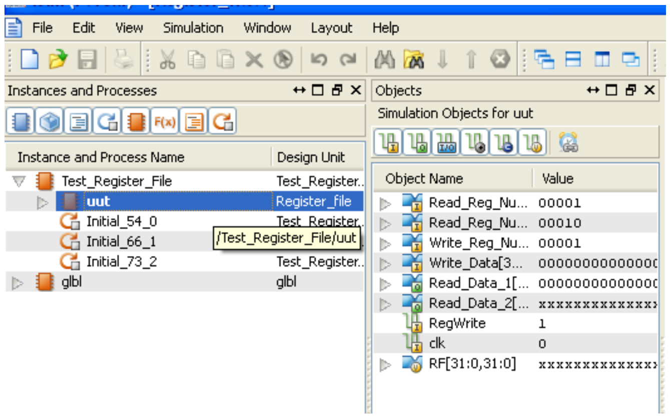
```
//Write Data in to registers
   initial begin
   RegWrite = 0;
   #15;
   RegWrite = 1;  Write_Data = 20;  Write_Reg_Num = 0;
   #10;
   RegWrite = 1;  Write_Data = 30;  Write_Reg_Num = 1;
   #10;
   RegWrite = 1;  Write_Data = 30;  Write_Reg_Num = 1;
   #10;
   end

//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:

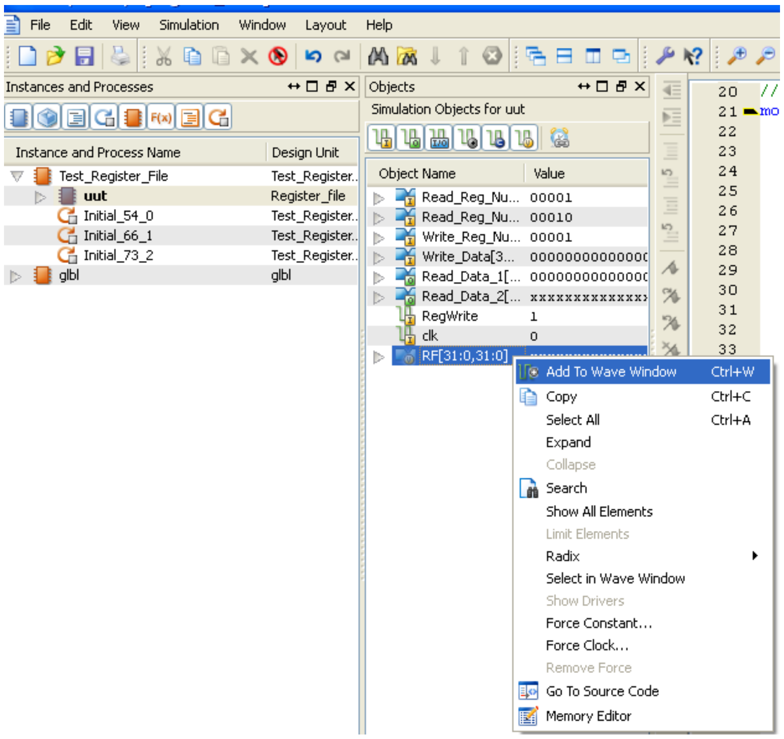
For verifying the design you might have to monitor the internal signals which could be defined as wires or Reg. In the above Register file design **RegMemory** will be defined as reg and is neither input nor output. The steps for adding internal signal (in this case RegMemory) into the wave form window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

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

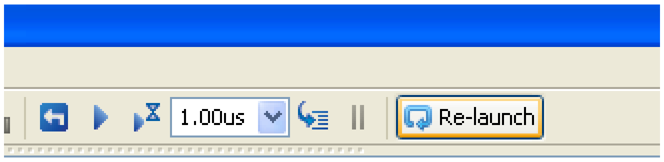


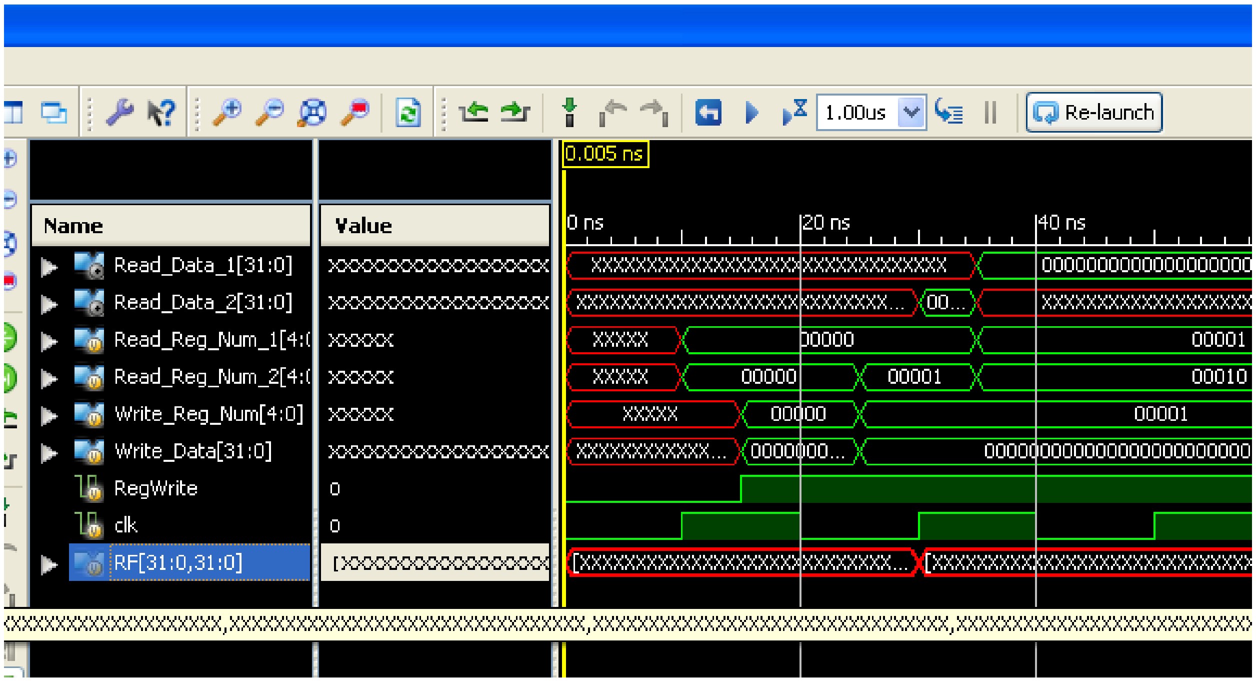
Instance and Process Name	Design Unit
Test_Register_File	Test_Register..
uut	Register_file
Initial_54_0	Test_Register..
Initial_66_1	
Initial_73_2	Test_Register..
glbl	glbl

Object Name	Value
Read_Reg_Nu...	00001
Read_Reg_Nu...	00010
Write_Reg_Nu...	00001
Write_Data[3...	0000000000000
Read_Data_1[...	0000000000000
Read_Data_2[...	xxxxxxxxxxxxx
RegWrite	1
clk	0
RF[31:0,31:0]	xxxxxxxxxxxxx



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.



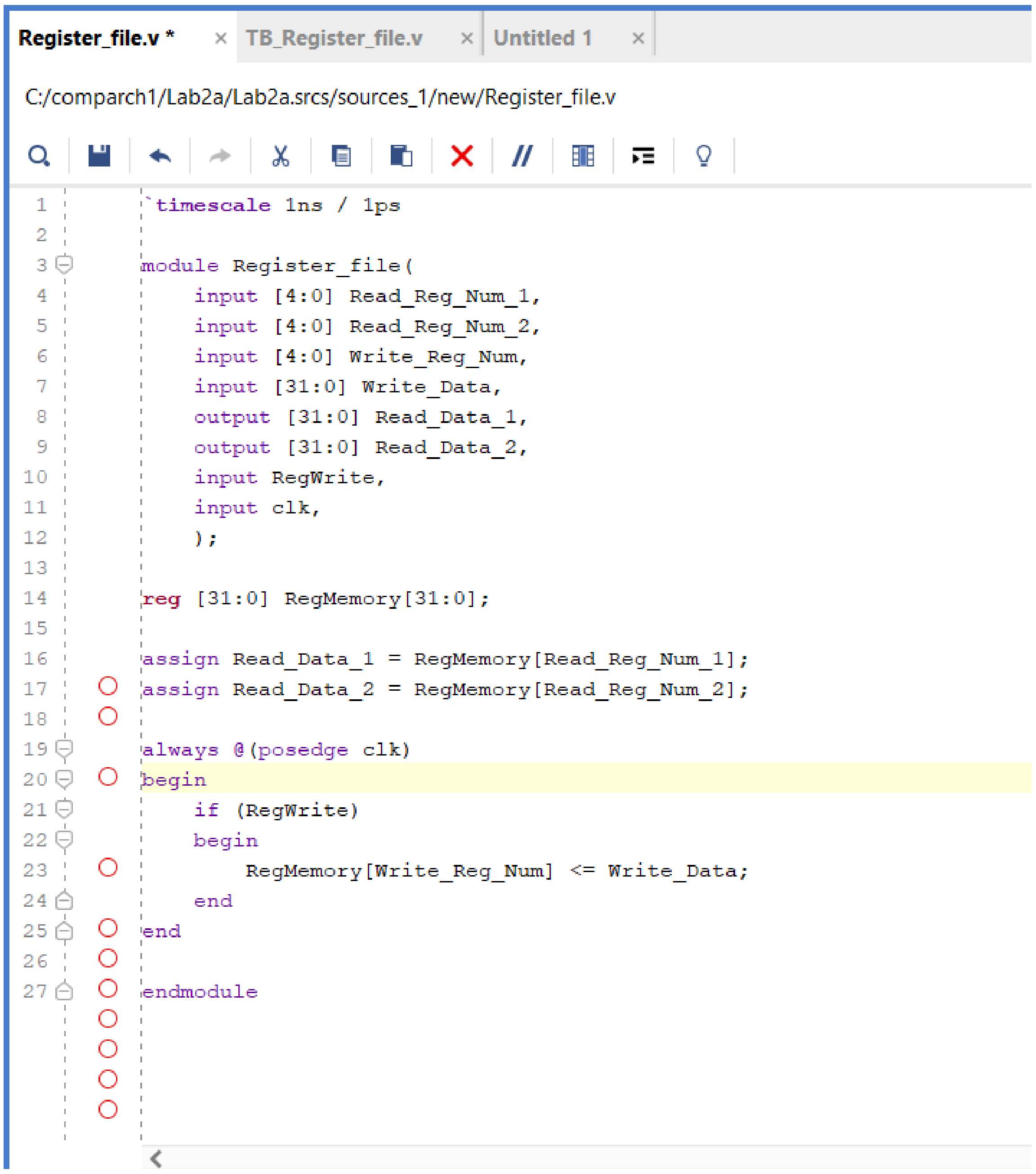


Q2.1. Did you get any errors during the check syntax of the design file or when simulating the testbench? If yes then, how did you solve the error?

Answer: The variables were not declared in the test bench and the module was not instantiated. So, I declared the variables and instantiated the module.

The first write was not reflected in the waveform because the write_data variable was changing before the following positive edge of the clock. Increased clock frequency to resolve this

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



```
`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,
    );

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)
    begin
        RegMemory[Write_Reg_Num] <= Write_Data;
    end
end

endmodule
```

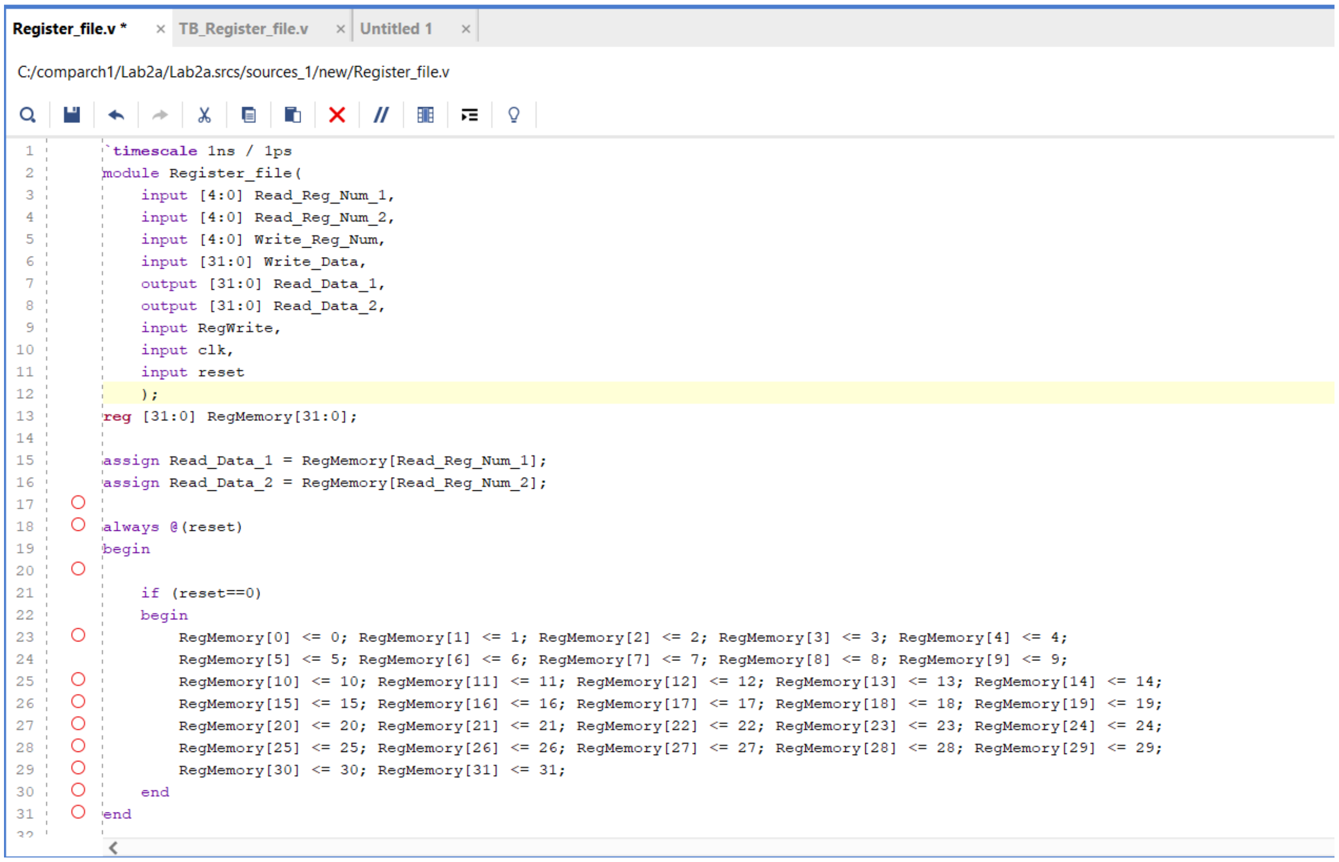
Answer:

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 will add an initial block that initializes the registers with some values.

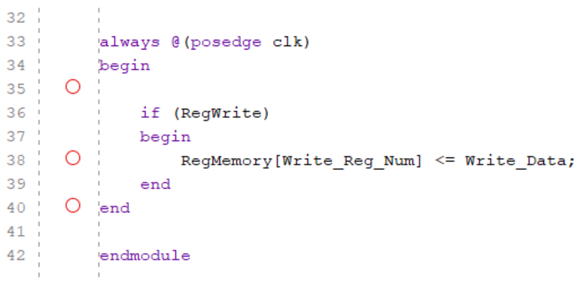
Q2.4. Modify the Register_File module by including a reset signal as input. Add an always block to your register file. This always block checks for reset signal and initializes the register memory (RegMemory) with some default values?

Answer:


Register_file.v * × TB_Register_file.v × Untitled 1 ×

C:/comparch1/Lab2a/Lab2a.srcs/sources_1/new/Register_file.v

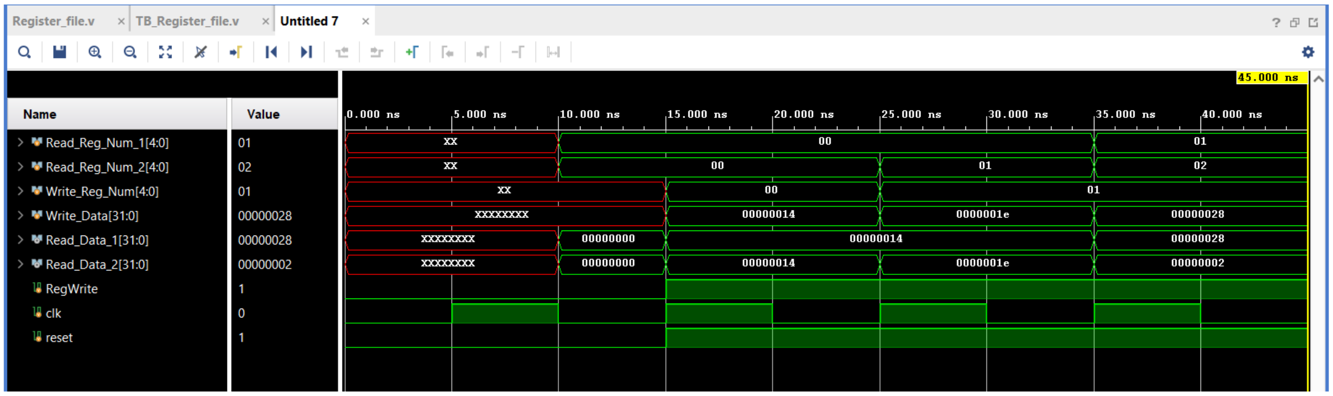
```verilog
`timescale 1ns / 1ps
module Register_file(
    input [4:0] Read_Reg_Num_1,
    input [4:0] Read_Reg_Num_2,
    input [4:0] Write_Reg_Num,
    input [31:0] Write_Data,
    output [31:0] Read_Data_1,
    output [31:0] Read_Data_2,
    input RegWrite,
    input clk,
    input reset
    );
reg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

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

always @(reset)
begin

    if (reset==0)
    begin
        RegMemory[0] <= 0; RegMemory[1] <= 1; RegMemory[2] <= 2; RegMemory[3] <= 3; RegMemory[4] <= 4;
        RegMemory[5] <= 5; RegMemory[6] <= 6; RegMemory[7] <= 7; RegMemory[8] <= 8; RegMemory[9] <= 9;
        RegMemory[10] <= 10; RegMemory[11] <= 11; RegMemory[12] <= 12; RegMemory[13] <= 13; RegMemory[14] <= 14;
        RegMemory[15] <= 15; RegMemory[16] <= 16; RegMemory[17] <= 17; RegMemory[18] <= 18; RegMemory[19] <= 19;
        RegMemory[20] <= 20; RegMemory[21] <= 21; RegMemory[22] <= 22; RegMemory[23] <= 23; RegMemory[24] <= 24;
        RegMemory[25] <= 25; RegMemory[26] <= 26; RegMemory[27] <= 27; RegMemory[28] <= 28; RegMemory[29] <= 29;
        RegMemory[30] <= 30; RegMemory[31] <= 31;
    end
end
```



```verilog

always @(posedge clk)
begin

    if (RegWrite)
    begin
        RegMemory[Write_Reg_Num] <= Write_Data;
    end
end

endmodule
```

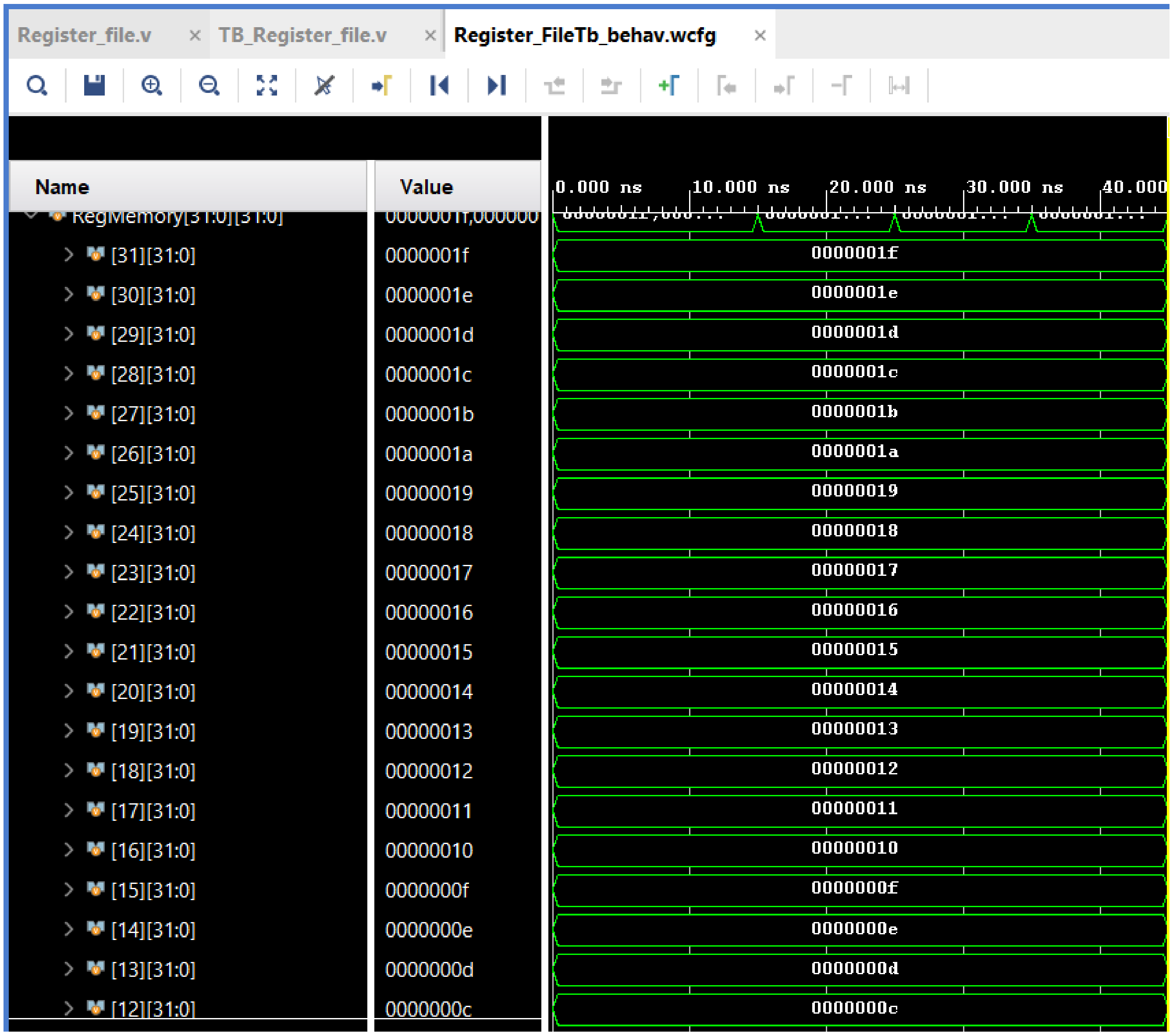
Q2.5. Copy the image of the waveform window that is generated for the given Testbench? Waveforms should be in decimal/ hexadecimal view.

Answer:



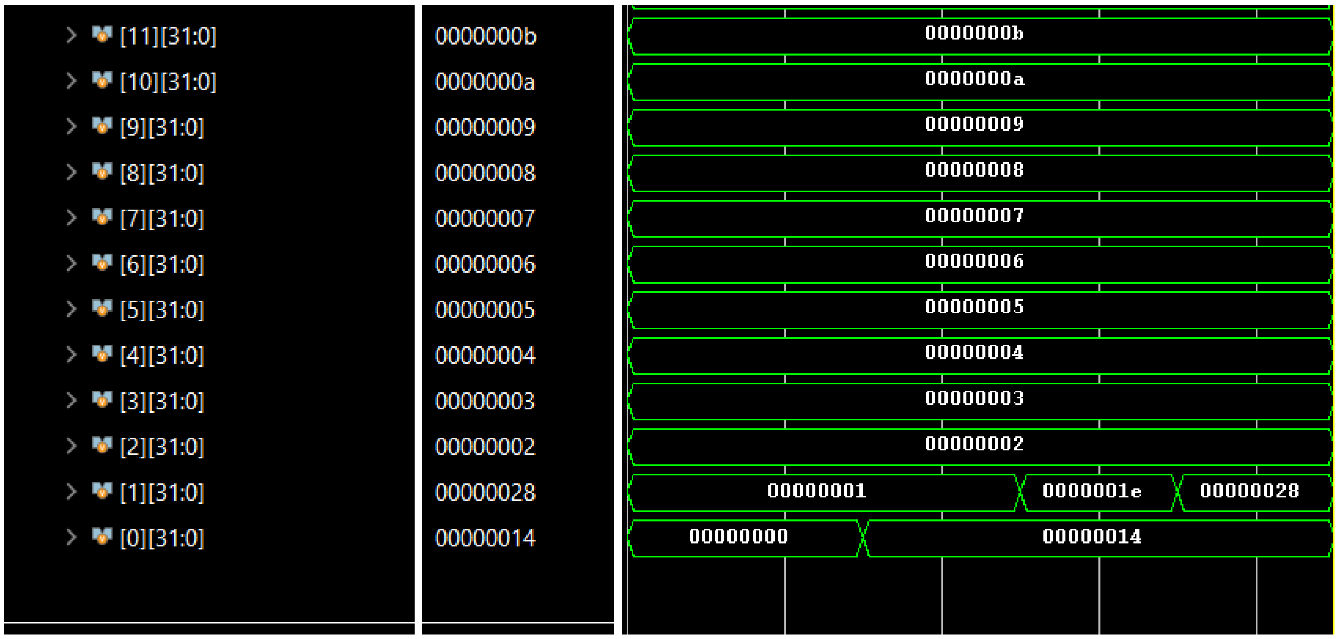
Register_file.v × TB_Register_file.v × Untitled 7 ×

Name	Value
Read_Reg_Num_1[4:0]	01
Read_Reg_Num_2[4:0]	02
Write_Reg_Num[4:0]	01
Write_Data[31:0]	00000028
Read_Data_1[31:0]	00000028
Read_Data_2[31:0]	00000002
RegWrite	1
clk	0
reset	1



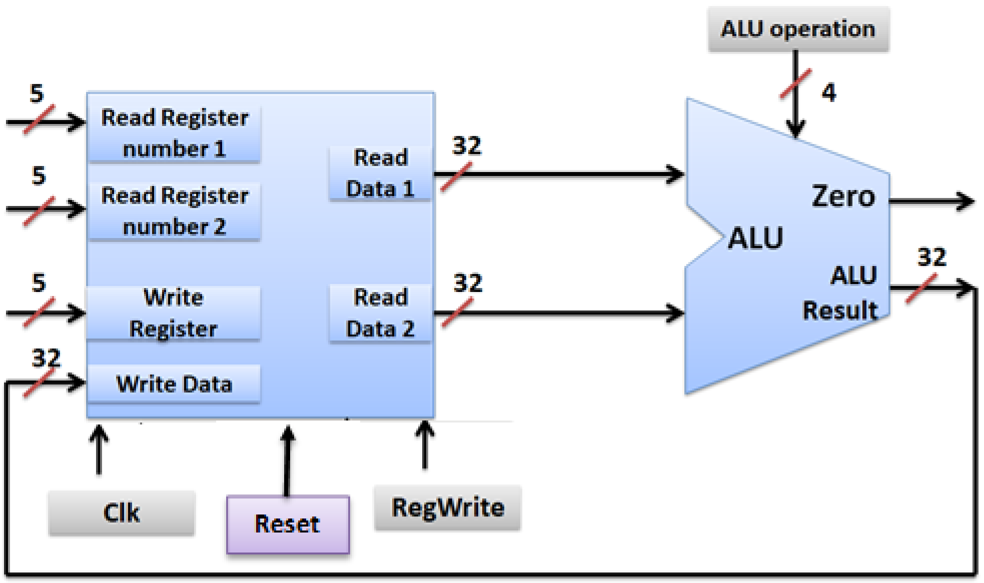
Register_file.v × TB_Register_file.v × Register_FileTb_behav.wcfg ×

Name	Value
RegMemory[31:0][31:0]	0000001f,000000
[31][31:0]	0000001f
[30][31:0]	0000001e
[29][31:0]	0000001d
[28][31:0]	0000001c
[27][31:0]	0000001b
[26][31:0]	0000001a
[25][31:0]	00000019
[24][31:0]	00000018
[23][31:0]	00000017
[22][31:0]	00000016
[21][31:0]	00000015
[20][31:0]	00000014
[19][31:0]	00000013
[18][31:0]	00000012
[17][31:0]	00000011
[16][31:0]	00000010
[15][31:0]	0000000f
[14][31:0]	0000000e
[13][31:0]	0000000d
[12][31:0]	0000000c



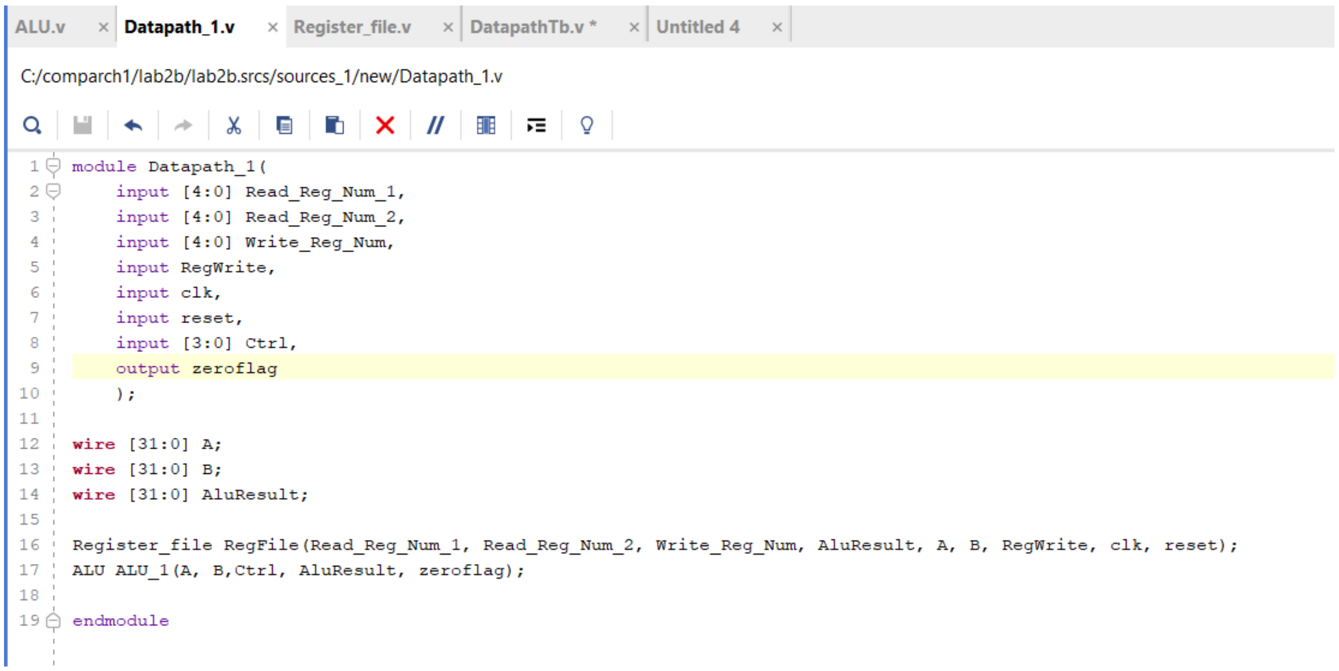
[11][31:0]	0000000b
[10][31:0]	0000000a
[9][31:0]	00000009
[8][31:0]	00000008
[7][31:0]	00000007
[6][31:0]	00000006
[5][31:0]	00000005
[4][31:0]	00000004
[3][31:0]	00000003
[2][31:0]	00000002
[1][31:0]	00000028
[0][31:0]	00000014

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 the Datapath module. Make internal connections as per the Datapath given. (e.g. ReadData1 is connected to the first input of ALU). Please note that all intermediate signals should be defined as wire with appropriate bit widths.



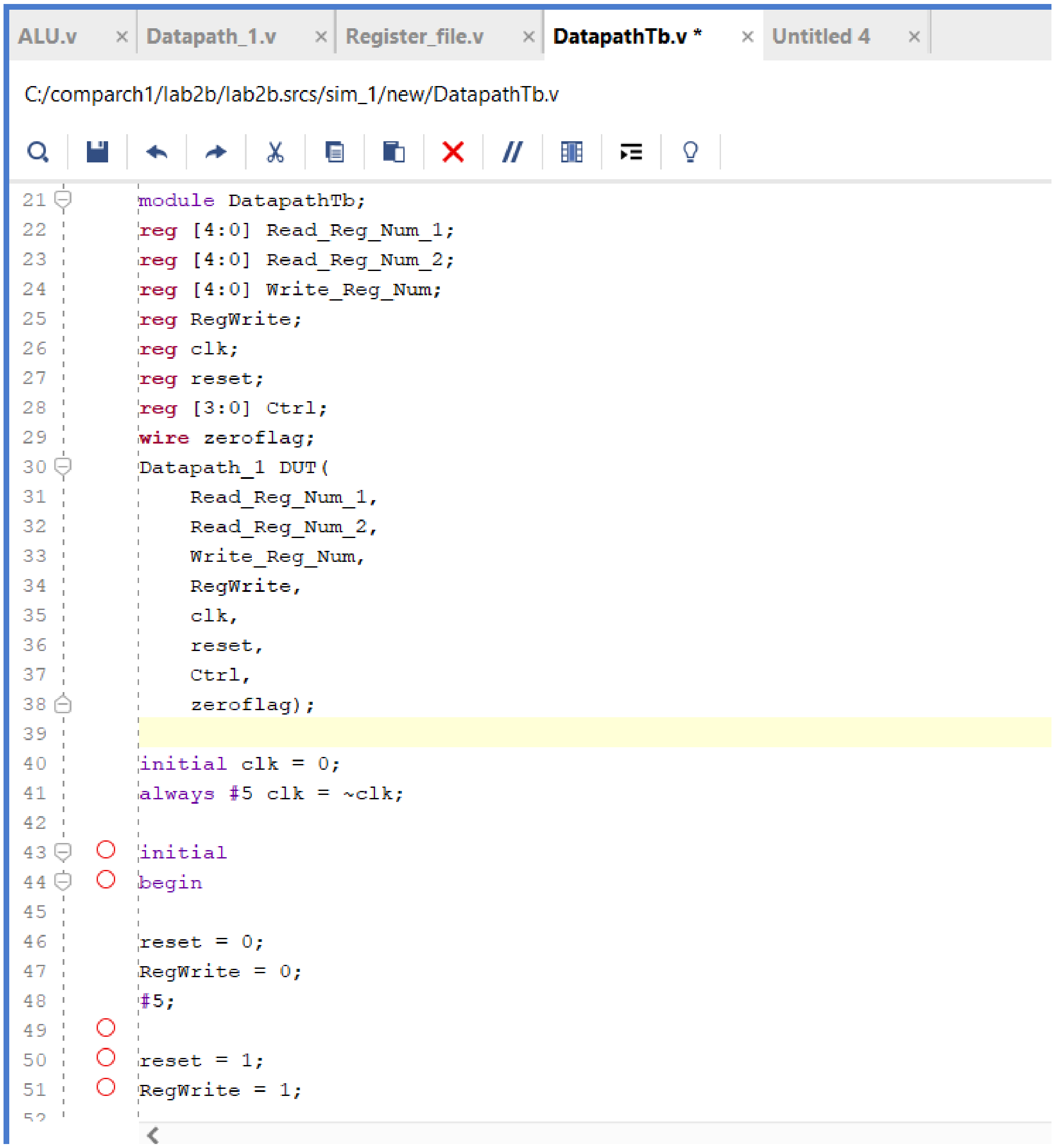
Q2.6. Copy the image of the Verilog code of the above Datapath.

Answer:



```
module Datapath_1(
    input [4:0] Read_Reg_Num_1,
    input [4:0] Read_Reg_Num_2,
    input [4:0] Write_Reg_Num,
    input RegWrite,
    input clk,
    input reset,
    input [3:0] Ctrl,
    output zeroflag
    );

wire [31:0] A;
wire [31:0] B;
wire [31:0] AluResult;

Register_file RegFile(Read_Reg_Num_1, Read_Reg_Num_2, Write_Reg_Num, AluResult, A, B, RegWrite, clk, reset);
ALU ALU_1(A, B,Ctrl, AluResult, zeroflag);

endmodule
```

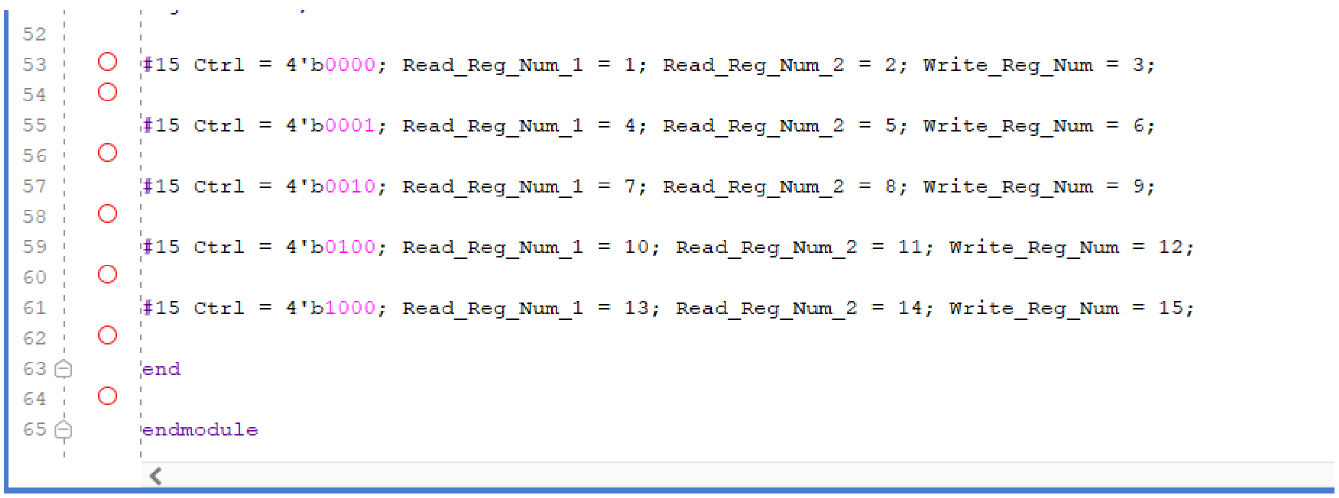
Q2.7. Write an appropriate test bench to test the above data path. Copy the test bench below (Hint: You should supply new register numbers after the positive edge of Clk or before the positive edge of Clk to avoid confusion)


ALU.v × Datapath_1.v × Register_file.v × DatapathTb.v * × Untitled 4 ×

C:/comparch1/lab2b/lab2b.srcs/sim_1/new/DatapathTb.v

```
module DatapathTb;
reg [4:0] Read_Reg_Num_1;
reg [4:0] Read_Reg_Num_2;
reg [4:0] Write_Reg_Num;
reg RegWrite;
reg clk;
reg reset;
reg [3:0] Ctrl;
wire zeroflag;
Datapath_1 DUT(
    Read_Reg_Num_1,
    Read_Reg_Num_2,
    Write_Reg_Num,
    RegWrite,
    clk,
    reset,
    Ctrl,
    zeroflag);

initial clk = 0;
always #5 clk = ~clk;

initial
begin

reset = 0;
RegWrite = 0;
#5;

reset = 1;
RegWrite = 1;
```

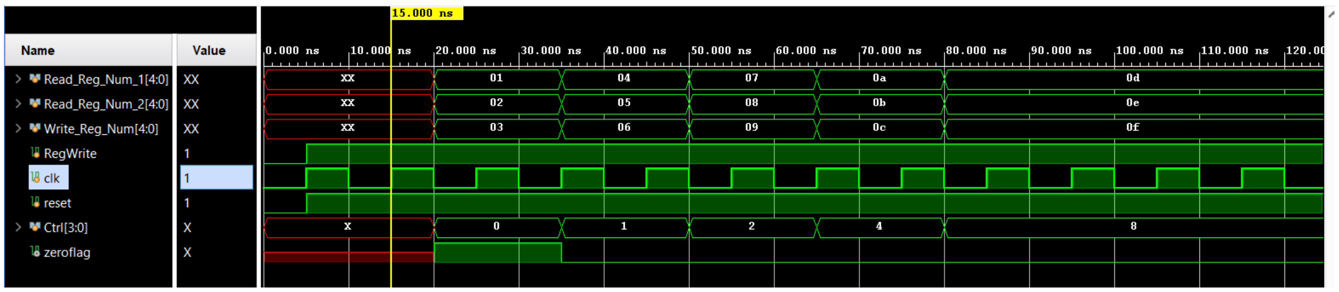
Answer:



```
#15 Ctrl = 4'b0000; Read_Reg_Num_1 = 1; Read_Reg_Num_2 = 2; Write_Reg_Num = 3;

#15 Ctrl = 4'b0001; Read_Reg_Num_1 = 4; Read_Reg_Num_2 = 5; Write_Reg_Num = 6;

#15 Ctrl = 4'b0010; Read_Reg_Num_1 = 7; Read_Reg_Num_2 = 8; Write_Reg_Num = 9;

#15 Ctrl = 4'b0100; Read_Reg_Num_1 = 10; Read_Reg_Num_2 = 11; Write_Reg_Num = 12;

#15 Ctrl = 4'b1000; Read_Reg_Num_1 = 13; Read_Reg_Num_2 = 14; Write_Reg_Num = 15;

end

endmodule
```

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

Answer:

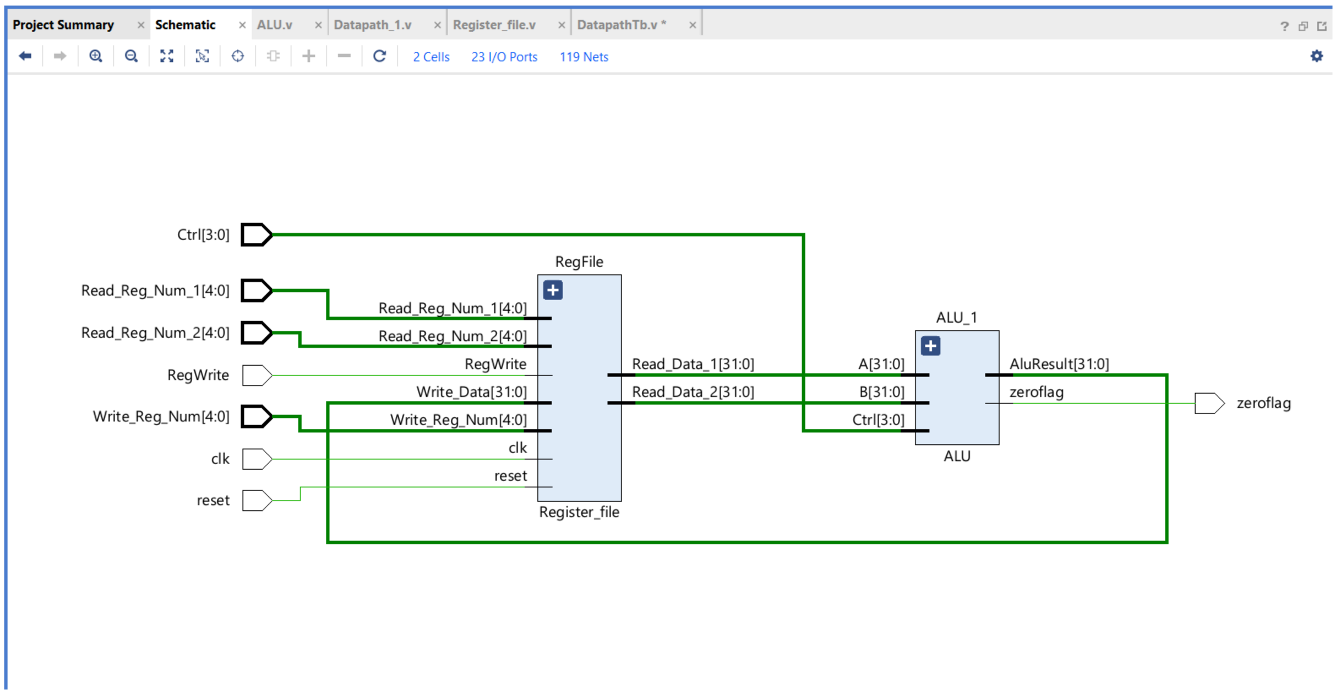


Q2.9. List the concepts you learned from this experiment (Conclusions/Observations)

Answer: Learned to implement Register file and Datapath. Specifically, I learnt how to connect different modules, using multiple procedural blocks and initializing with reset signal

<u>Exercise 2.3 Now Synthesise the above partial Datapath using the Xilinx Vivado</u>

Q2.10. Copy the RTL generated below



Q2.11. Note the observations from the RTL generated

Information from the read register 1 and read register 2 is retrieved as read data 1 and read data 2. Subsequently, these data inputs are directed to A and B inputs of the Arithmetic Logic Unit (ALU). The ALU performs operations based on the Control Signal, and the outcome is stored in ALUResult. Simultaneously, the zero flag is updated to reflect the current state. The resultant ALUResult is looped back and provided as feedback in the Write_Data.