

## Lab 2: MIPS Single Cycle and Multi Cycle Processors

---

### Introduction

In this lab, you will have an interactive experience with a simple MIPS single cycle and a simple MIPS multi cycle processor. For the experiments, simulated processors written in System Verilog is utilized. The “simple” processors mean that only few instructions are implemented. To interact with the simulated processors and run experiments we will use the ModelSim simulation tool. ModelSim is multi-language HDL simulation environment developed by Mentor Graphics and supported by Intel through their FPGA packages. This lab will help you go through ModelSim and simulate the simple MIPS single cycle and simple MIPS multi cycle processors. This lab also assumes that you have adequate knowledge in System Verilog HDL, test bench functions, MIPS assembly instructions, MIPS single cycle microarchitecture, and MIPS multi cycle microarchitecture. After completing the guided experiments in this lab, you are required to do the designated homework.

### 1. ModelSim Software Environment

#### Objectives

In this part, you will go through the how to download, set-up, and familiarize ModelSim software interface and learn the basic setup and functionalities that you will need for simulating the MIPS single cycle processor.

#### A. Download and Set-up

>>To download ModelSim installer please go to this cloud storage link (RTA Lab NAS Storage):

<http://gofile.me/4s9TK/2a6nfMhW4>

>>Or you can navigate through the Intel page to download but you are required to sign-up to download the needed set-up files:

[https://fpgasoftware.intel.com/19.4/?edition=pro&product=modelsim\\_ae#tabs-2](https://fpgasoftware.intel.com/19.4/?edition=pro&product=modelsim_ae#tabs-2)

#### B. Starting with ModelSim

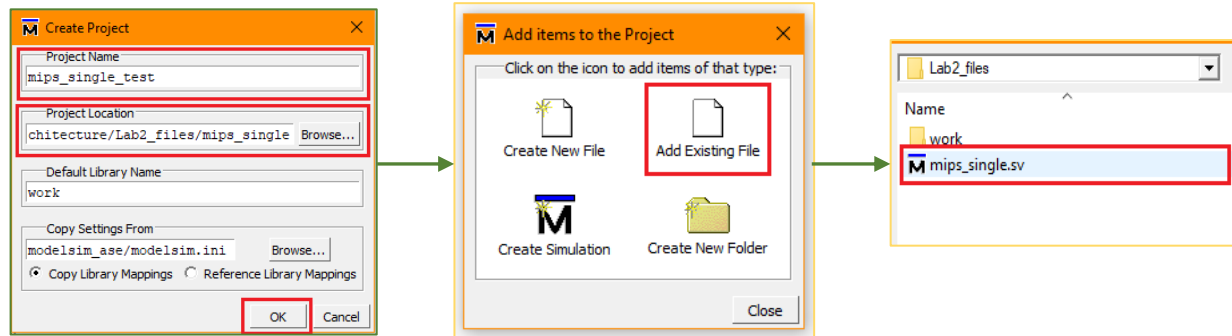
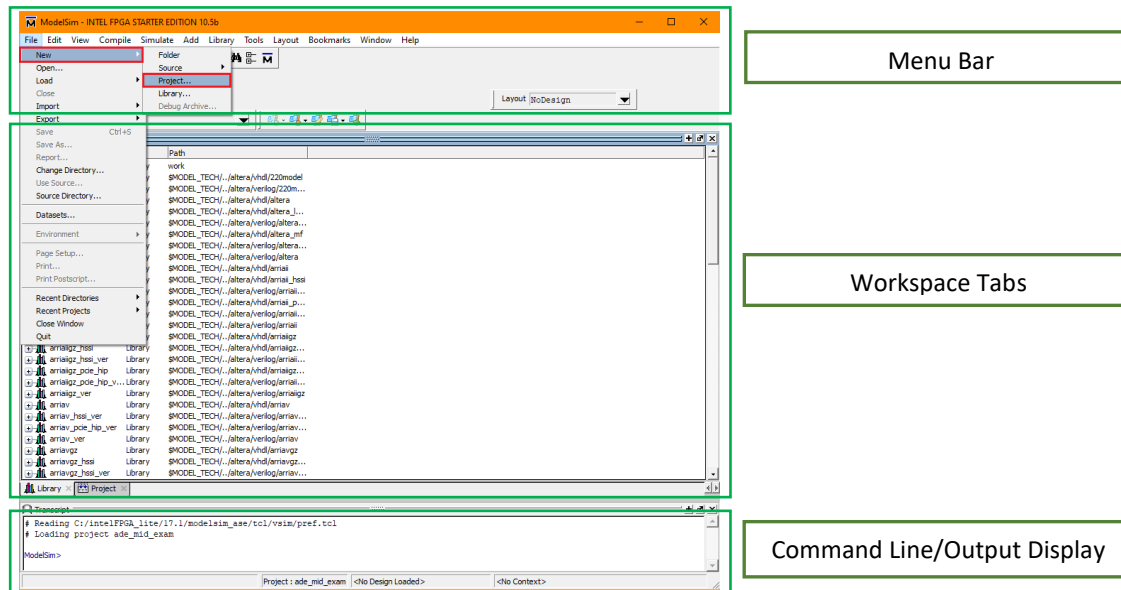
>>Before launching ModelSim, create a folder in your directory named “Lab2\_files” and inside that folder create another folder called “mips\_single\_test”.



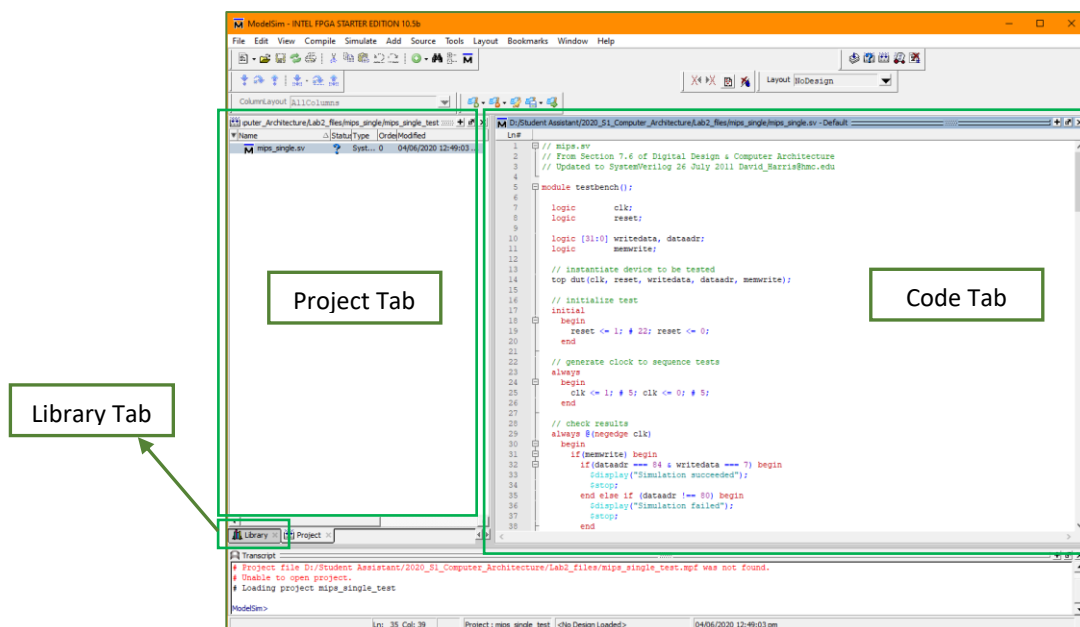
>>Launch ModelSim, then create a new project: File->New->Project

- Specify the project name as “mips\_single-test”; specify the project location to the mips\_single\_test folder that you created then press->OK.
- Then “Add items to the project” window will prompt then select “Add existing File” then->Browse.
- Select the “mips\_single.sv” file that is included in this lab experiment.

## Computer Architecture Laboratory Lab 2

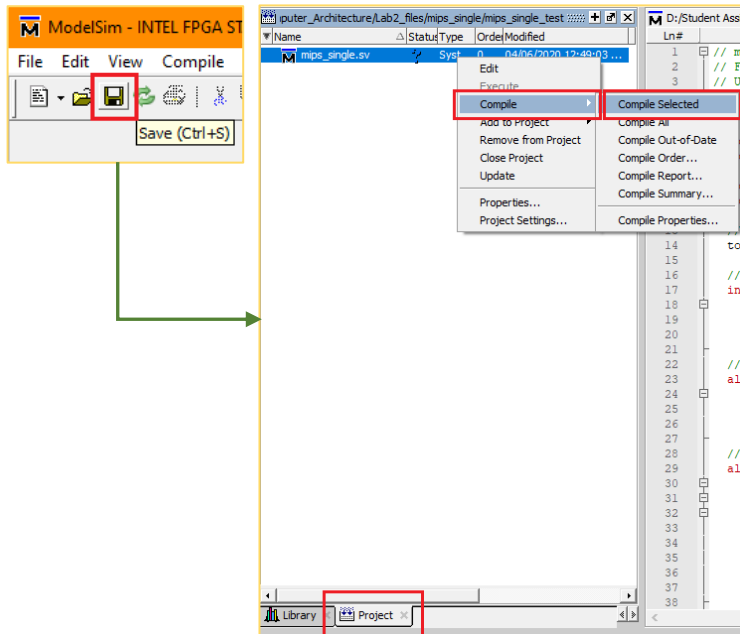


>>After creating a new project, the main editing workspace of ModelSim will appear.



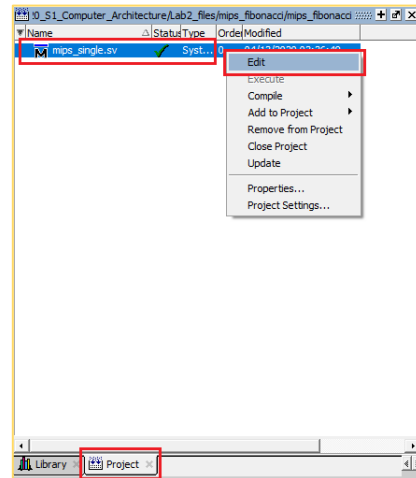
### C. Compiling a Project

>>To compile a project, first save the project then select the Project tab and right click the project named mips\_single.sv->Compile->Compile Selected.

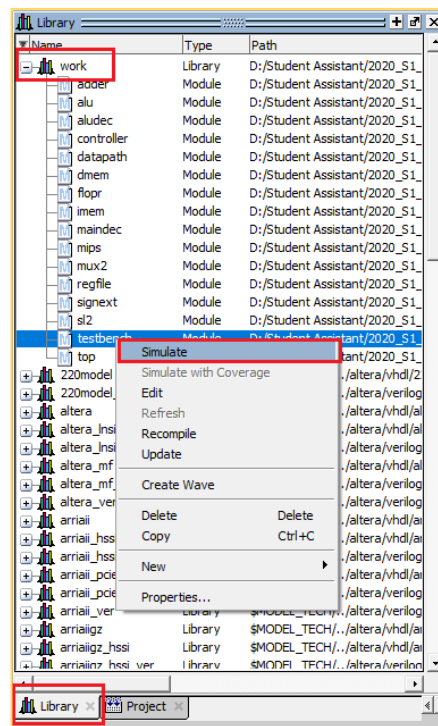


#### Additional Information:

If the "Code Tab" did not appear. Right click "mips\_single.sv" and click Edit.



>>To start simulation mode click the Library tab then expand the work node and right click testbench then->Simulate.



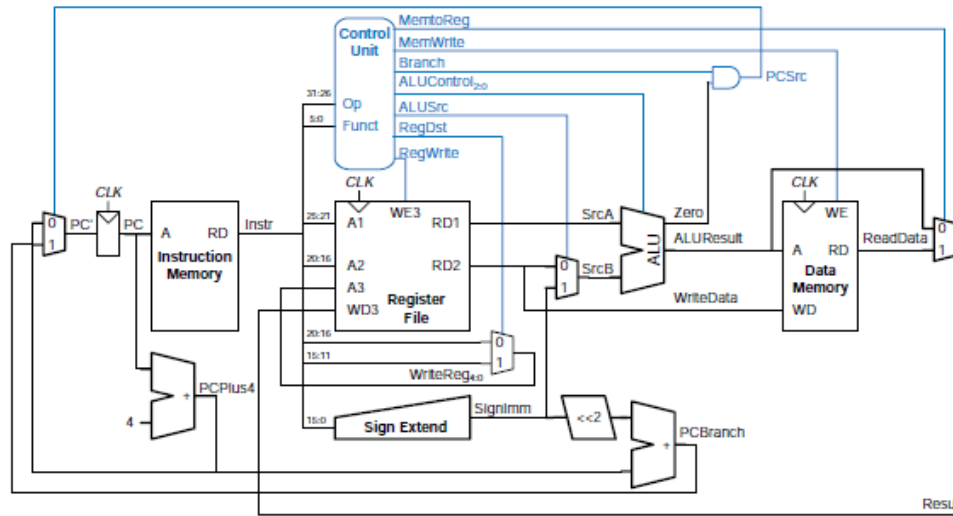
## 2. Simulating MIPS Single Cycle Processor

### Objectives

In this part, you will go through the simulation process in ModelSim and interact with the simple MIPS single cycle processor.

### Preliminaries:

>>For your reference:



### Complete MIPS Single Cycle Processor

(Source: Harris & Harris, "Digital Design and Computer Architecture MIPS Edition", Chapter 7, Complete Single-Cycle MIPS Processor)

>>The available instructions in the simple implementation of the MIPS Single Cycle Processor:

```
always_comb
case(op)
6'b000000: controls <= 9'b110000010; // RIYPE
6'b100011: controls <= 9'b101001000; // LW
6'b101011: controls <= 9'b001010000; // SW
6'b000100: controls <= 9'b000100001; // BEQ
6'b001000: controls <= 9'b101000000; // ADDI
6'b000010: controls <= 9'b000000100; // J
default: controls <= 9'bxxxxxxx; // illegal op
endcase
endmodule

module aludec(input logic [5:0] funct,
input logic [1:0] aluop,
output logic [2:0] alucontrol);

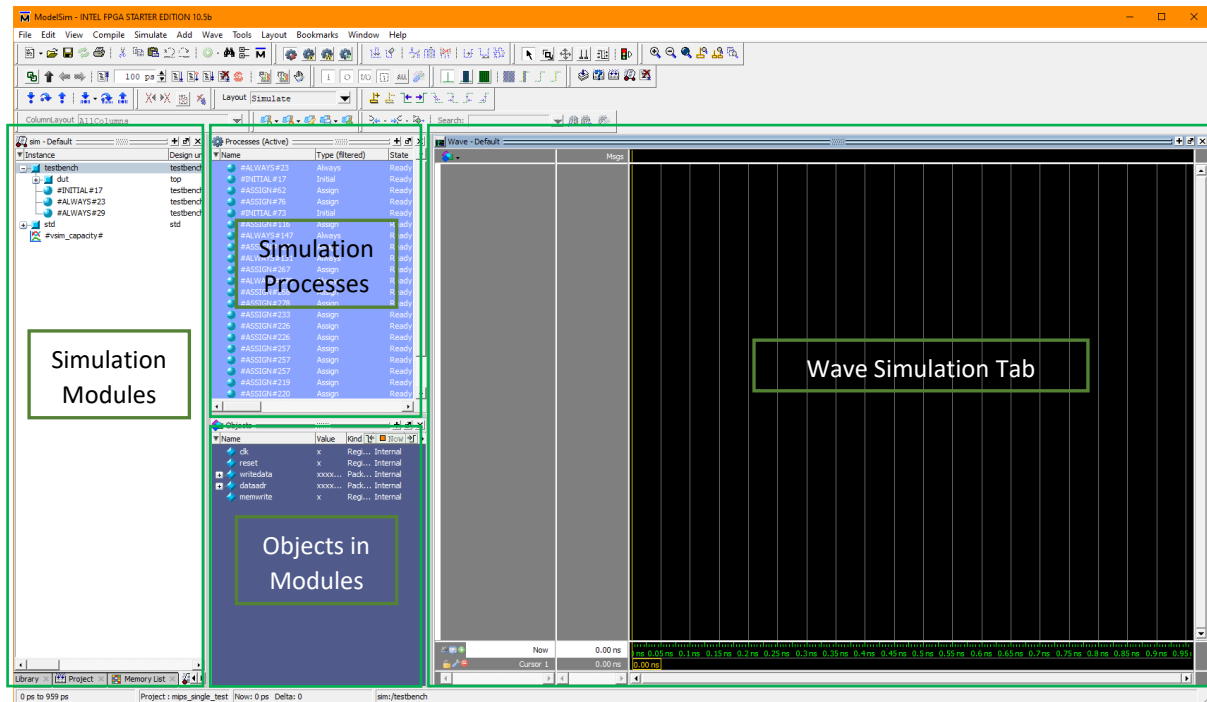
always_comb
case(aluop)
2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
2'b01: alucontrol <= 3'b110; // sub (for beq)
default: case(funct) // R-type instructions
6'b100000: alucontrol <= 3'b010; // add
6'b100010: alucontrol <= 3'b110; // sub
6'b100100: alucontrol <= 3'b000; // and
6'b100101: alucontrol <= 3'b001; // or
6'b101010: alucontrol <= 3'b111; // slt
default: alucontrol <= 3'bxxx; // ???
endcase
endcase
endmodule
```

### Note:

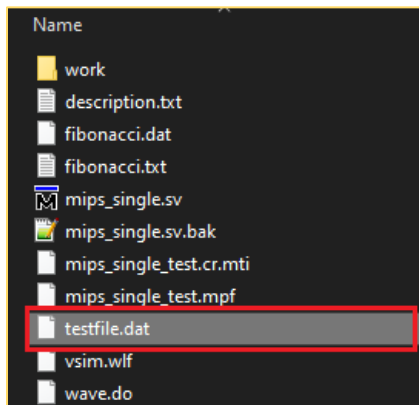
Instructions that are currently not on this list are not yet implemented and will not be executed by the processor.

## A. ModelSim Simulation Window

>>Upon clicking simulation, a new simulation window will appear.



>>Before proceeding with the simulation, make sure the file “testfile.dat” is inside the “mips\_single\_test” folder.



### Note:

The “testfile.dat” contains the machine code that will be loaded into the instructions memory. It contains the machine code instructions that are needed to run in the processor. This code is the default test code to ensure that simple MIPS single cycle processor is working properly. **Please see the “description.txt” file to see the description of “testfile.dat”.**

```
module imem(input logic [5:0] a,
            output logic [31:0] rd);

    logic [31:0] RAM[63:0];

    initial
        $readmemh("testfile.dat",RAM);

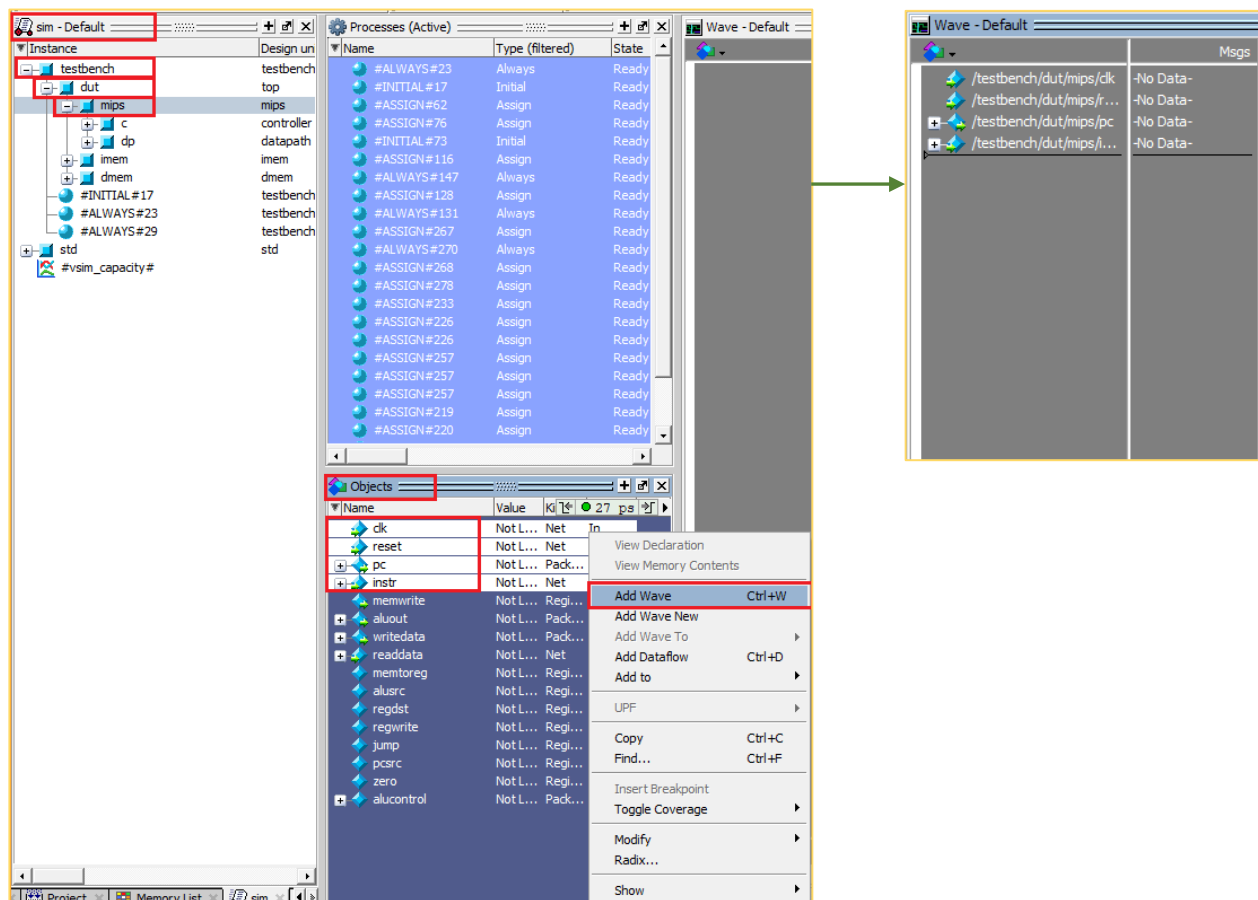
    assign rd = RAM[a]; // word aligned
endmodule
```

## B. Adding Variables to Monitor

>>In this simulation we need to add the following variables to monitor:

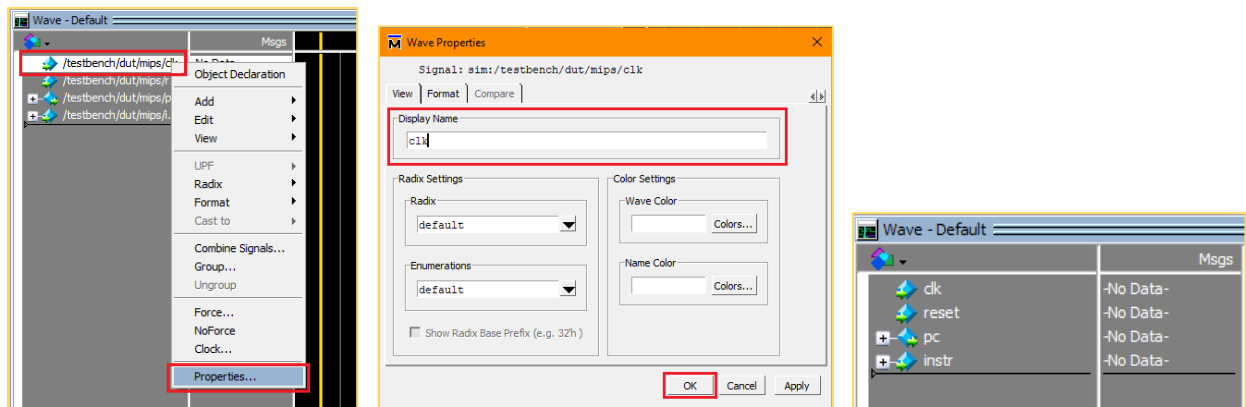
- clock
- reset (\*\*optional)
- program counter
- current instruction
- instructions memory
- data memory
- register file

>>To add the variables to monitor (**clock, reset, program counter, current instruction**), click on the “Simulation Modules” tab->expand “testbench” node, then expand “dut” node and click on “mips” node. Then, go to the “Objects” tab then select the indicated objects (**clk, reset, pc, instr**) and right click->Add wave. Those variables will appear in the “Wave Simulation” tab.

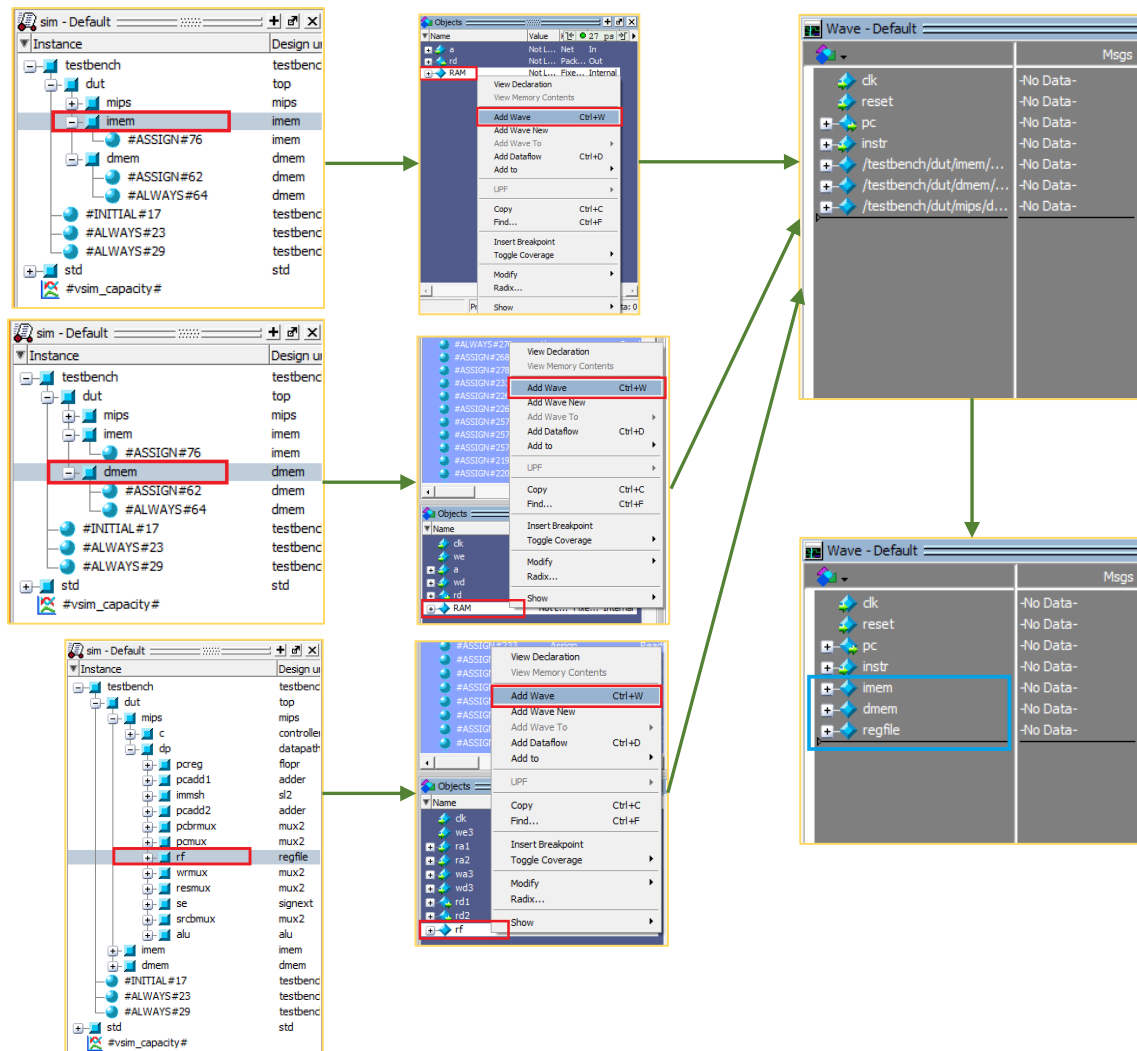


## Computer Architecture Laboratory Lab 2

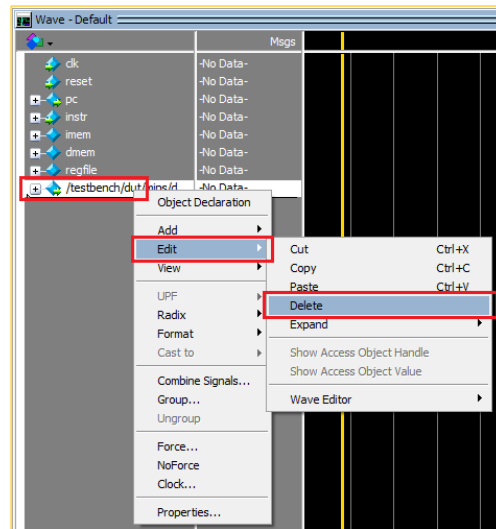
>>After adding the first four variables (**clk**, **reset**, **pc**, **instr**), we have to give it a display name because the default naming looks confusing. Select each added variables in the “Wave Simulation” tab then right click and go to Properties. Go to the “Display Name and give it the name accordingly as follows:



>>Let's add the remaining variables (**imem**, **dmem**, **regfile**). Then rename them accordingly as follows:



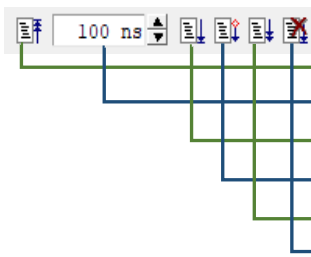
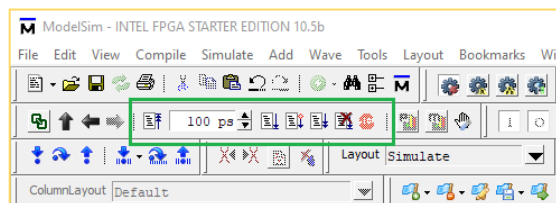
**Additional Information:** If you accidentally add some variables that you don't intend to monitor, you can remove these variables from the "Wave Simulation" tab by right clicking then Edit->Delete.



### C. Single Cycle Test Simulation

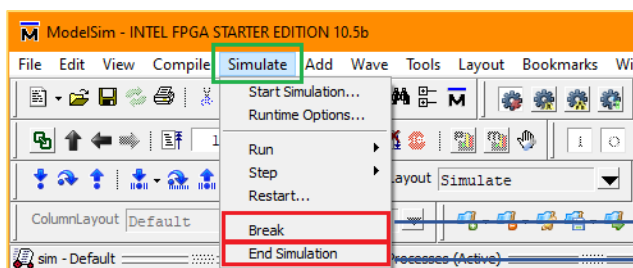
>>After adding all the variables to monitor, we can finally start the simulation. First, familiarize the icons and options in the simulation toolbar or in the dropdown menu.

#### Simulation Toolbar



- Restart Simulation – This will restart simulation and load default variables
- Adjust Running Length for Run Simulation
- Run Simulation – Run the simulation according to the running length
- Continue Run – Continue previous simulation command
- Run All – Run simulation according Input Wave Pattern length
- Break – Interrupt long running simulation

#### Dropdown Menu

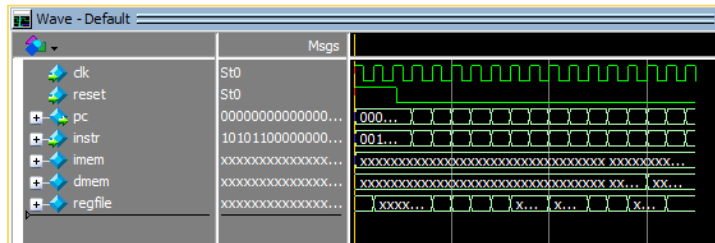


- Interrupt long running simulation
- Interrupt current simulation and closes the simulation window (revert back to coding window).

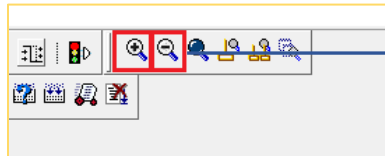


## Computer Architecture Laboratory Lab 2

>>Let's start the simulation. Click the "Run -All" icon in the Simulation Toolbar or go to the dropdown menu: Simulate->Run->Run -All.

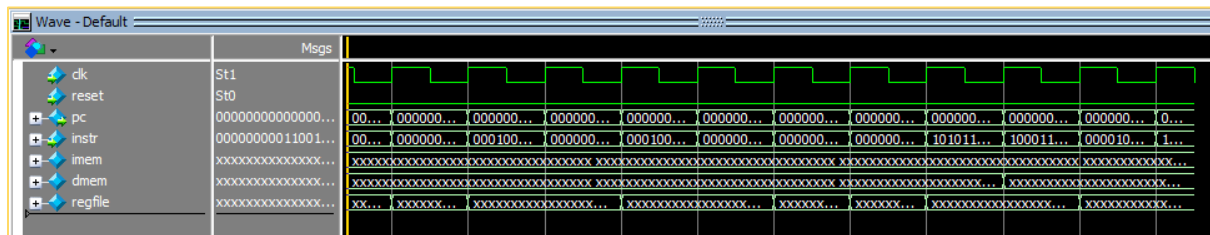


The output waveform should look like this. But, we need to adjust some settings so that we can better analyze it.



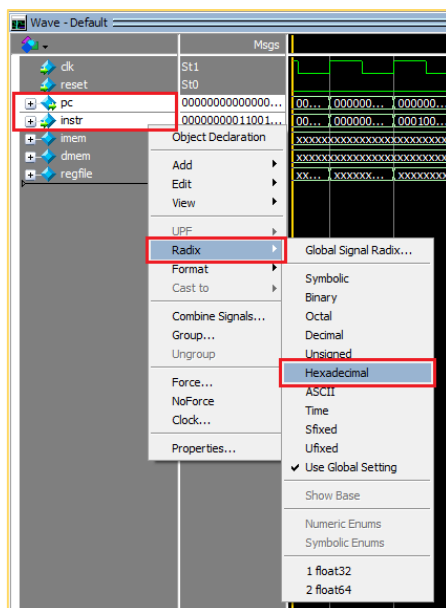
→ Zoom In and Zoom Out

>>Zooming In the output waveform should look like this (it should look better than the one above):



>>The display of the output values may seem a bit confusing because it is binary. Now, let's change the radix of the display. Some output values might be more convenient to be displayed in hexadecimal while others will be much better if decimal.

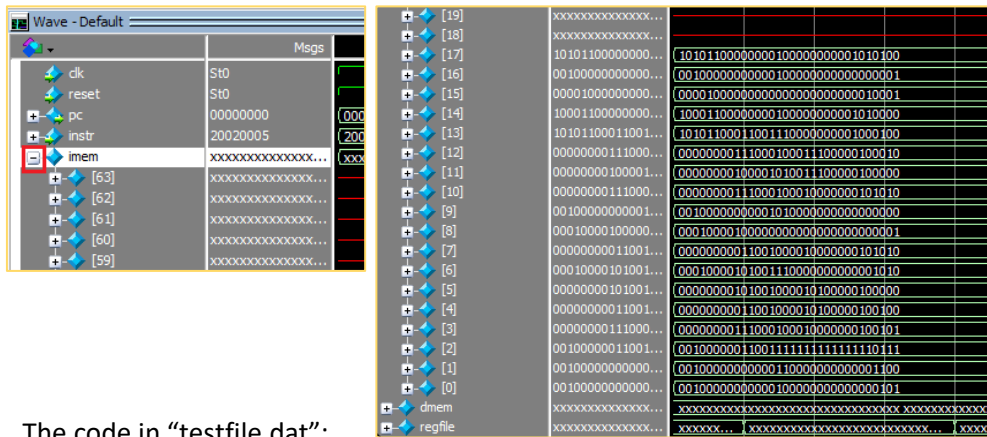
>>Select **pc** and **instr** then **change the radix to Hexadecimal**: right click->Radix->Hexadecimal.



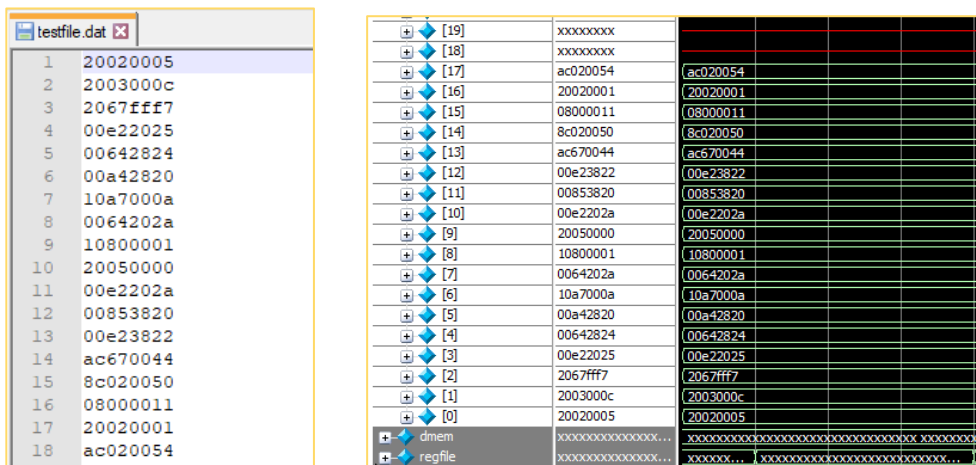
The display values of **pc** and **instr** should look like this instead of the binary display from the figure above.

00...	00000014	00000018	0000001c	00000020	00000028	0000002c
00...	00a42820	10a7000a	0064202a	10800001	00e2202a	00853820

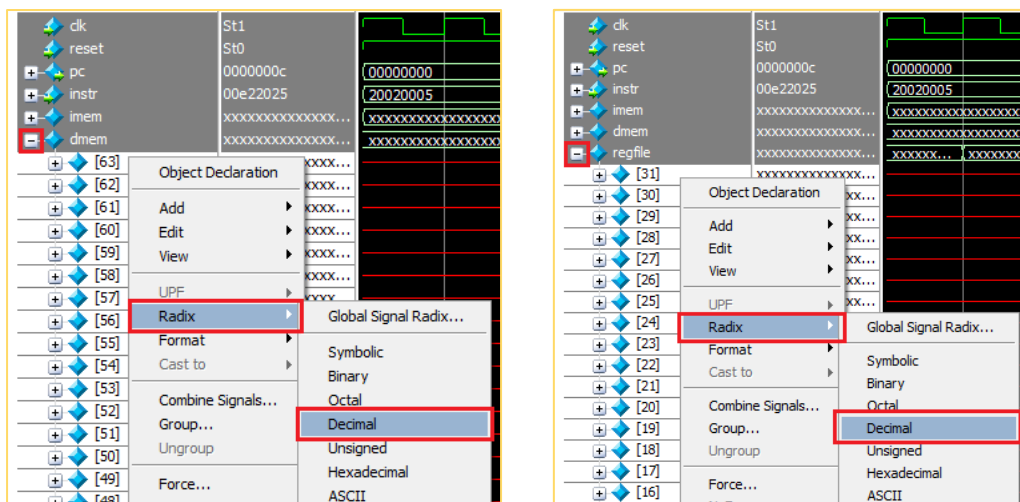
>>Expand the **imem** variable then click [63]->Shift->click [0] to select all node variables from [0] to [63] (You need to scroll down because the variables from [0] to [63] may not fit the current window). Then **change the radix to Hexadecimal**.



The code in "testfile.dat":

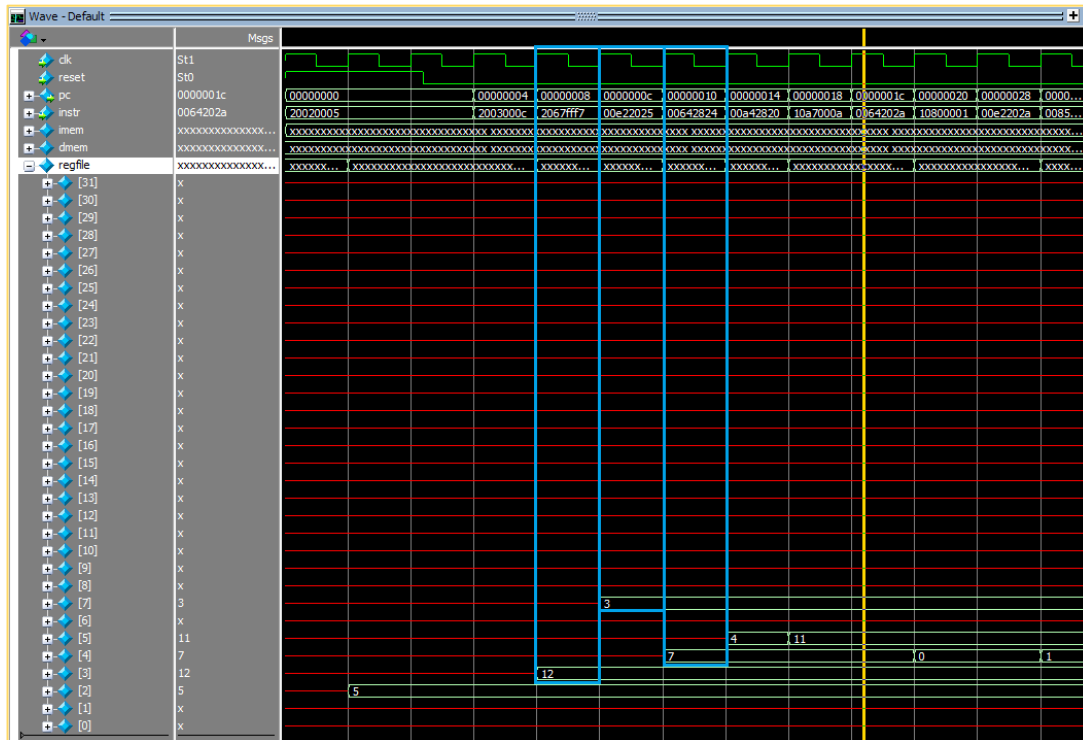


>>For **dmem** and **regfile**, do the same steps from the steps above (expand node, select the words from [0] to [n], and change the radix) then **change the radix to Decimal**.

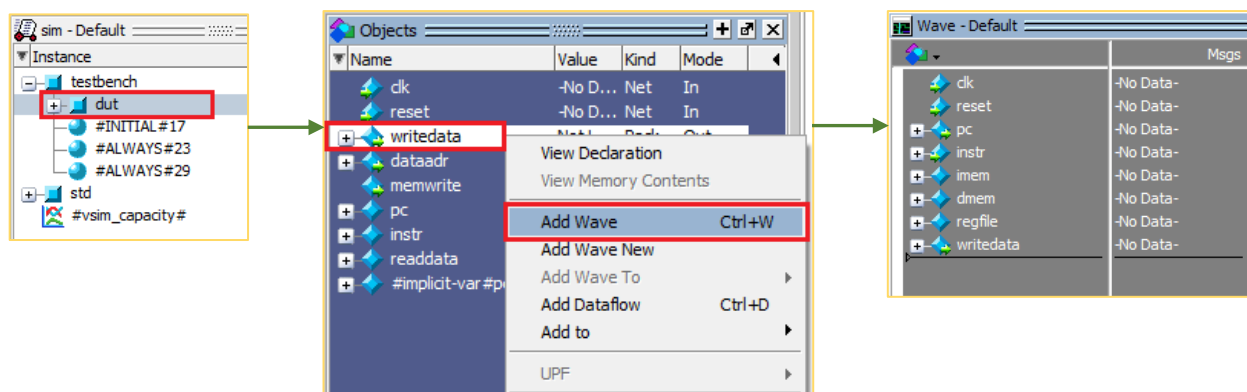


## Computer Architecture Laboratory Lab 2

>>Since we are simulating a MIPS single cycle processor. Notice that the instructions are executed at a single clock cycle and also the same with the program counter update ( $pc_{n+1} = pc_n + 4$ ).



>>Now, to wrap up this test simulation. Let's review the success condition from the test bench. Notice that in the test bench, the simulation succeeds when data **writedata** == 7. To show this value, let's add the writedata variable to the wave simulation. Reset the simulation and go to the "Simulation Modules" tab then click the "dut" node. Then, in the "Objects" tab add the "**writedata**" to the wave and rename it. Change the radix of writedata variable to Decimal. Run the simulation again and the last value in **writedata** should be 7. Now we can guarantee that our simple MIPS single cycle processor is working properly.



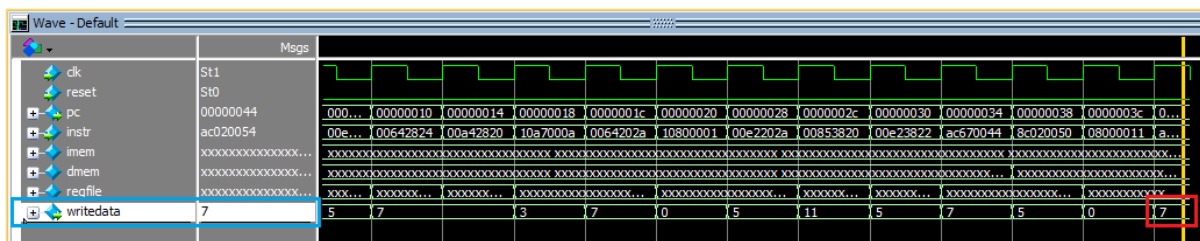
Check Results part in the test bench:

```

28 // check results
29 always @(negedge clk)
30 begin
31     if(memwrite) begin
32         if(dataadr == 84 & writedata == 7) begin
33             $display("Simulation succeeded");
34             $stop;
35         end else if (dataadr != 80) begin
36             $display("Simulation failed");
37             $stop;
38         end
39     end
40 end

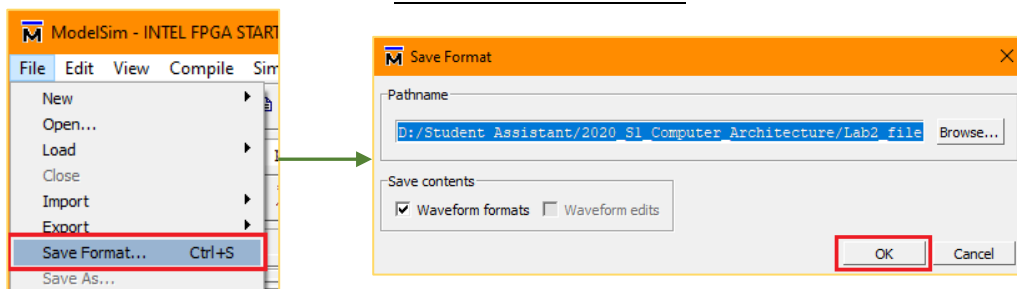
```

Output of writedata variable:

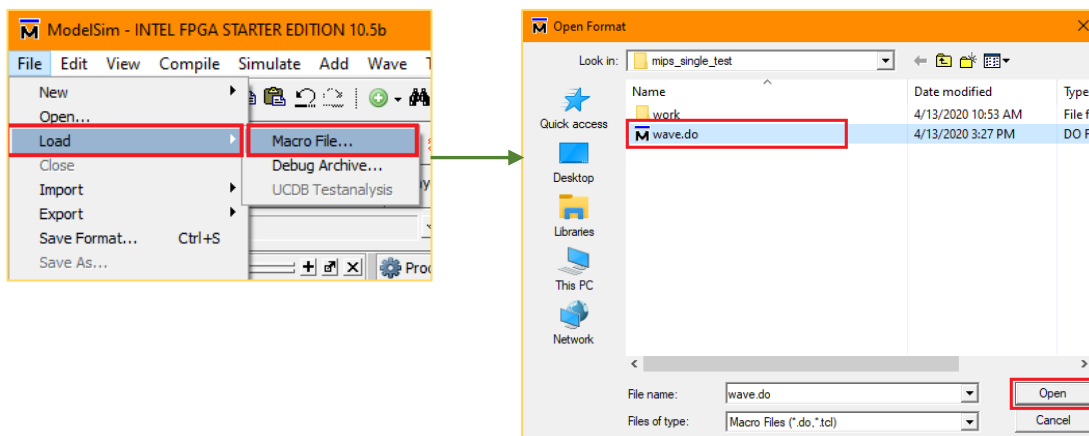


**Additional Information:** To save the settings of your simulations so you will not need to setup every time you rerun it. Go to File->Save Format; a "Save Format" window will appear and press OK. Then, to load the format, go to File->Load->Macro File; an "Open Format" window will appear and select "wave.do" file then click Open.

Save Simulation Format:



Load Simulation Format:

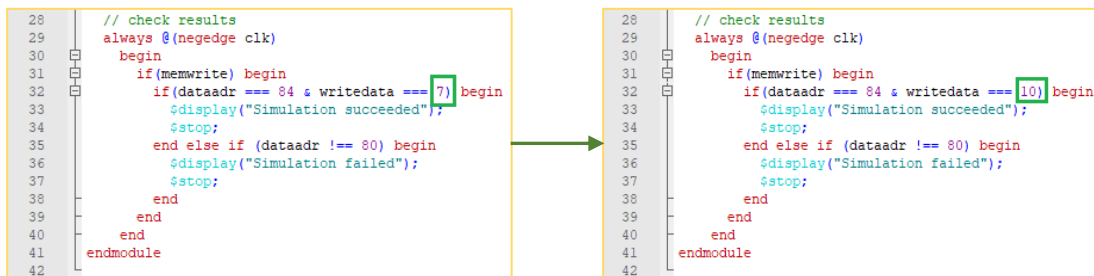


### D. Fibonacci Simulation in MIPS Single Cycle

By now, we are already confident that our simple MIPS single cycle processor is working properly. Let's simulate a rather fun and intuitive code; we will run a Fibonacci sequence generator (it is assumed that you already know the concept of Fibonacci numbers). Create a new ModelSim project and follow the steps below (*please refer through the steps in sections 1-B to 2-C*):

- In the "Lab2\_files" folder create a folder named "mips\_fibonacci". Then, copy and paste the "mips\_single.sv" file. Also, copy and paste "fibonacci.dat" which is the code that we will have to run. (*Optional: copy and paste "fibonacci.txt", the description for the Fibonacci machine code*)
- Create a new project file, name it "mips\_fibonacci" and then point the "Project Location" to the "mips\_fibonacci" folder. Then, Add Existing File and add the mips\_single.sv. [Refer to section 1-B]
- Point to the project tab and compile the "mips\_single.sv". [Refer to section 1-C]
- Go to the "Code Tab" (if it is not showing refer to section 1-C how to open it) then edit the System Verilog code as follows so we can run the Fibonacci sequence generator code:

a) Go to the "check results" part of the "testbench" and replace number 7 with number 10 in the "writedata" If statement.



```

28 // check results
29 always @(negedge clk)
30 begin
31   if(memwrite) begin
32     if(dataadr == 84 & writedata == 7) begin
33       $display("Simulation succeeded");
34       $stop;
35     end else if (dataadr != 80) begin
36       $display("Simulation failed");
37       $stop;
38     end
39   end
40 end
41 endmodule
42

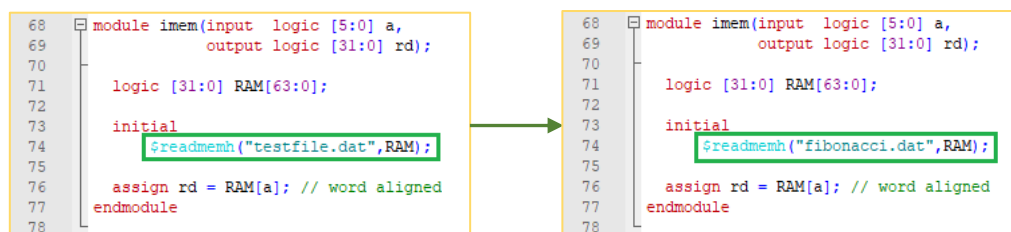
```

```

28 // check results
29 always @(negedge clk)
30 begin
31   if(memwrite) begin
32     if(dataadr == 84 & writedata == 10) begin
33       $display("Simulation succeeded");
34       $stop;
35     end else if (dataadr != 80) begin
36       $display("Simulation failed");
37       $stop;
38     end
39   end
40 end
41 endmodule
42

```

b) Go to the "module imem" part of the System Verilog code and replace "testfile.dat" with "fibonacci.dat".



```

68 module imem(input logic [5:0] a,
69             output logic [31:0] rd);
70
71   logic [31:0] RAM[63:0];
72
73   initial
74     $readmemh("testfile.dat", RAM);
75
76   assign rd = RAM[a]; // word aligned
77 endmodule
78

```

```

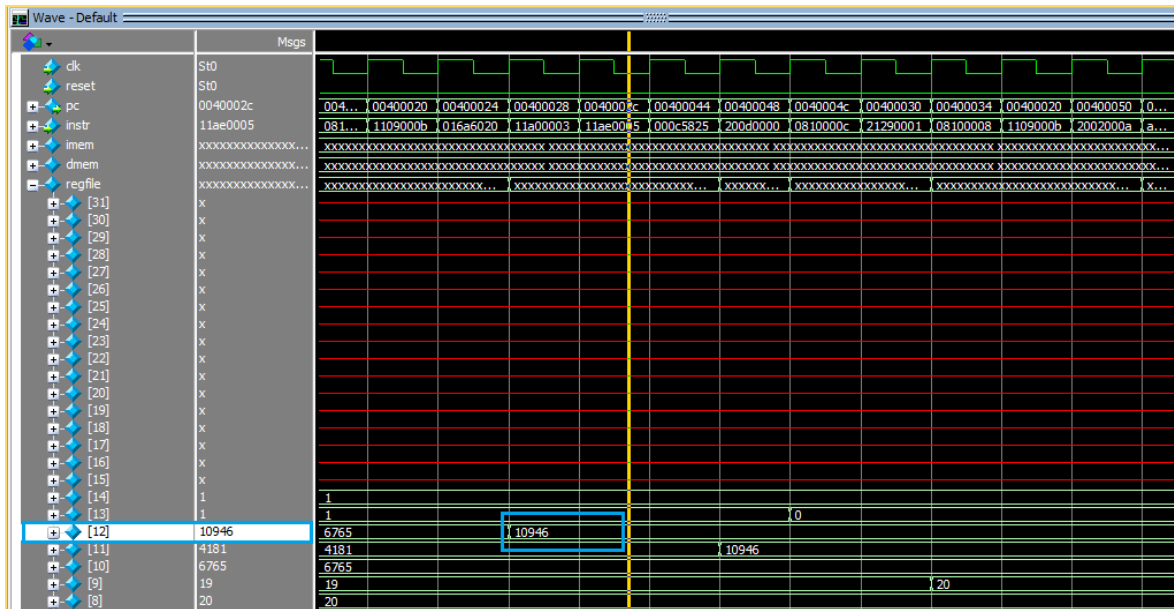
68 module imem(input logic [5:0] a,
69             output logic [31:0] rd);
70
71   logic [31:0] RAM[63:0];
72
73   initial
74     $readmemh("fibonacci.dat", RAM);
75
76   assign rd = RAM[a]; // word aligned
77 endmodule
78

```

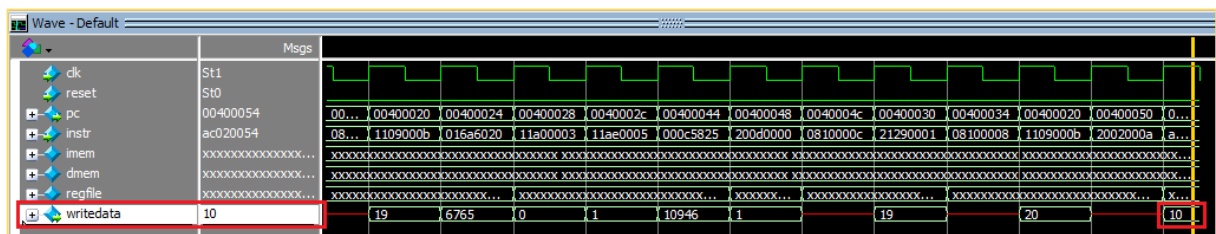
- After editing the System Verilog code, **save the changes**. Click the save icon for go to File->Save. Then, **recompile the project**. [Refer to section 1-C]
- After recompiling the project, go to the "Library" tab then expand work, select "testbench" and Simulate. [Refer to section 1-C]
- When the Simulation Window appears, add the variables to monitor: clock, reset (optional), program counter, current instruction, instructions memory, data memory, register file, and writedata [Refer to the last part of section 2-C for writedata]. Follow the same procedure in section 2-B for adding the main variables. Or you can do the **easier way**: copy and paste the "wave.do" from "mips\_single\_test" folder to the "mips\_fibonacci" folder then load the macro file in the current wave simulation. [Refer to the additional information in section 2-B]

## Computer Architecture Laboratory Lab 2

- **[\*\*Do this if you did not go for the “easier way”]** After adding the simulation variables Change the radix of each variables according to “change the radix” part in section 2-C.
- Then run the simulation (Run -All). The designated Fibonacci output in this code is register \$12 or \$t4 or in the simulation register [12]. This Fibonacci sequence generator generates up to  $f[21]=10946$ .



- You can check the generated Fibonacci sequence by scrolling through the outputs in register \$12. The Fibonacci sequence up to  $f[21]$ : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946.
- To wrap up this simulation, the last value in the **writedata** variable should be 10.



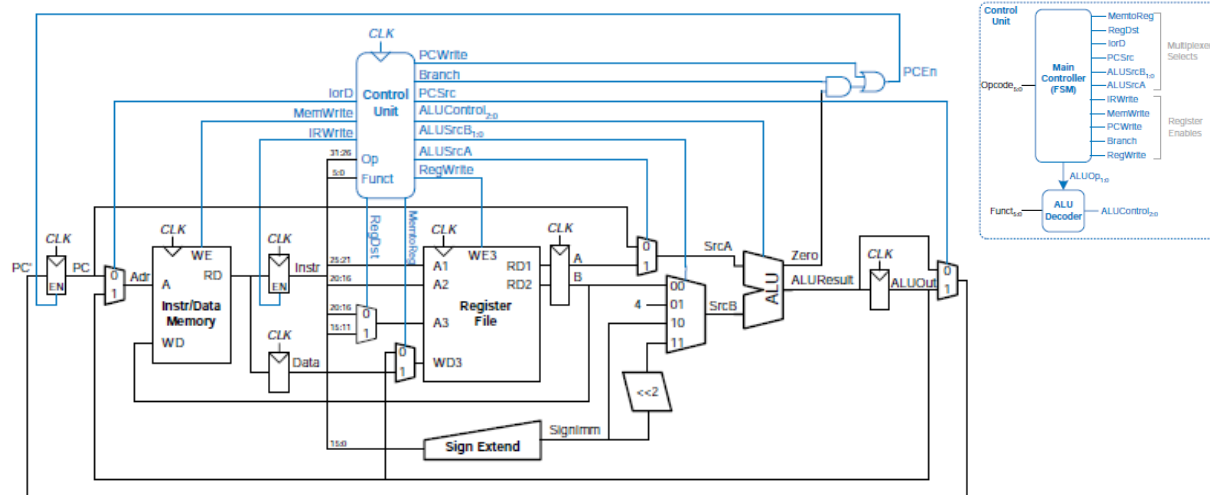
### 3. Simulating MIPS Multi Cycle Processor

#### Objectives

In this part, you will go through the simulation process in ModelSim and interact with the simple MIPS multi cycle processor.

#### Preliminaries:

>>For your reference:



#### Complete MIPS Multi Cycle Processor

(Source: Harris & Harris, "Digital Design and Computer Architecture MIPS Edition", Chapter 7, Complete Multi-Cycle MIPS Processor)

>>The available instructions in the simple implementation of the MIPS Multi Cycle Processor:

```
parameter LW      = 6'b100011;
parameter SW      = 6'b101011;
parameter RTYPE   = 6'b000000;
parameter BEQ     = 6'b000100;
parameter ADDI    = 6'b001000;
parameter J       = 6'b000010;
```

```
// Opcode for lw
// Opcode for sw
// Opcode for R-type
// Opcode for beq
// Opcode for addi
// Opcode for j
```

```
logic [3:0] state, nextstate;
logic [14:0] controls;
```

```
always_comb
case(aluop)
2'b00: alucontrol <= 3'b010; // add
2'b01: alucontrol <= 3'b110; // sub
default: case(funcnt) // RTYPE
6'b100000: alucontrol <= 3'b010; // ADD
6'b100010: alucontrol <= 3'b110; // SUB
6'b100100: alucontrol <= 3'b000; // AND
6'b100101: alucontrol <= 3'b001; // OR
6'b101010: alucontrol <= 3'b111; // SLT
default: alucontrol <= 3'bxxx; // ???
endcase
endcase
```

#### Note:

Instructions that are currently not on this list are not yet implemented and will not be executed by the processor.

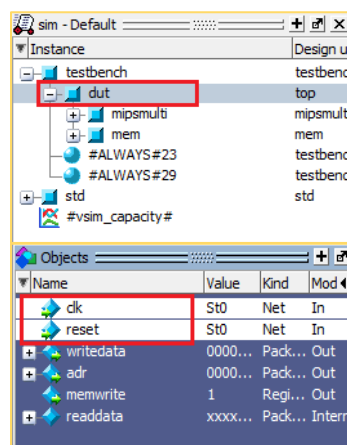


## A. Multi Cycle Test Simulation

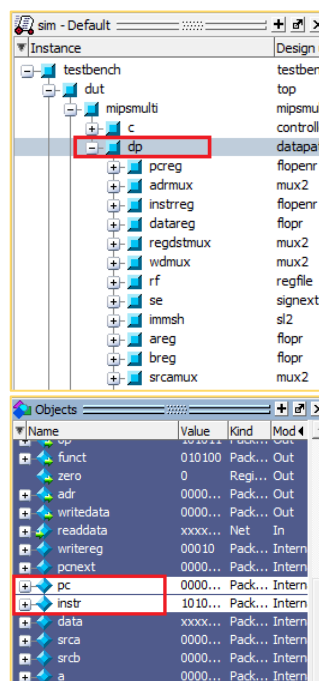
>>Now we have to run the same test simulation on the simple MIPS multi cycle processor as we did on the single cycle implementation.

- Create a folder named “**mips\_multi\_test**” folder in the “Lab2\_files” directory. Then, copy and paste the “**mips\_multi.sv**” file. Also, copy and paste the same “**testfile.dat**” that we used earlier.
- Follow the same procedure in sections 1-B to 2-C: create a new project named “mips\_multi\_test”, compile the code, and start the simulation.
- We need to add the following variables to monitor: clock (**clk**), reset (**reset**), program counter (**pc**), current instruction (**instr**), instructions/data memory (**mem**), register file (**regfile**), and writedata (**writedata**). Notice in this architecture, the instructions memory and data memory are not separate (refer to the complete MIPS multi cycle processor figure above) compared in the single cycle implementation. Note that the locations of these variables are different from the single cycle implementation because of the difference in the data path architecture. Refer to the figures below to locate each variables then add it on the wave simulation:

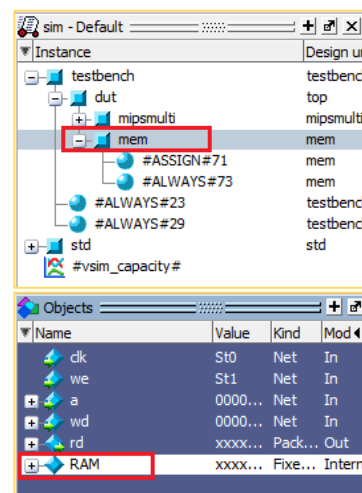
### clk and reset:



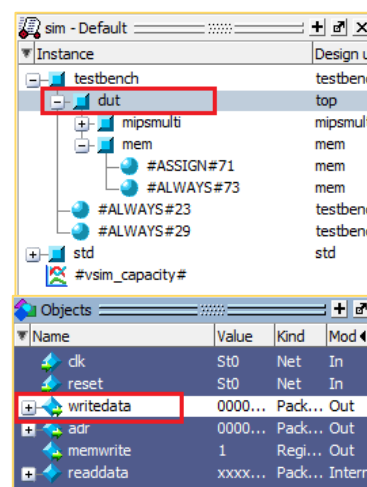
### pc and instr:



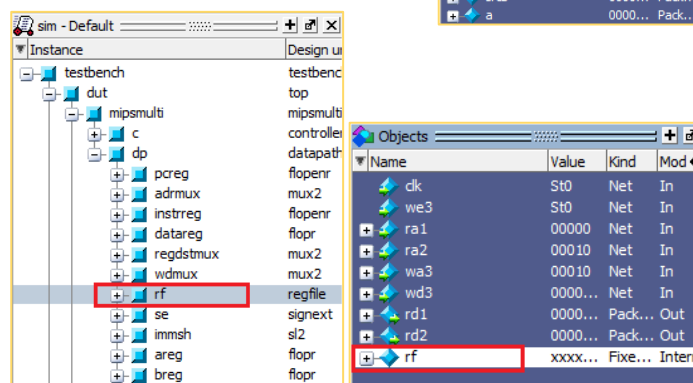
### mem:



### writedata:



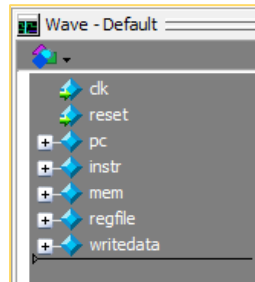
### regfile:



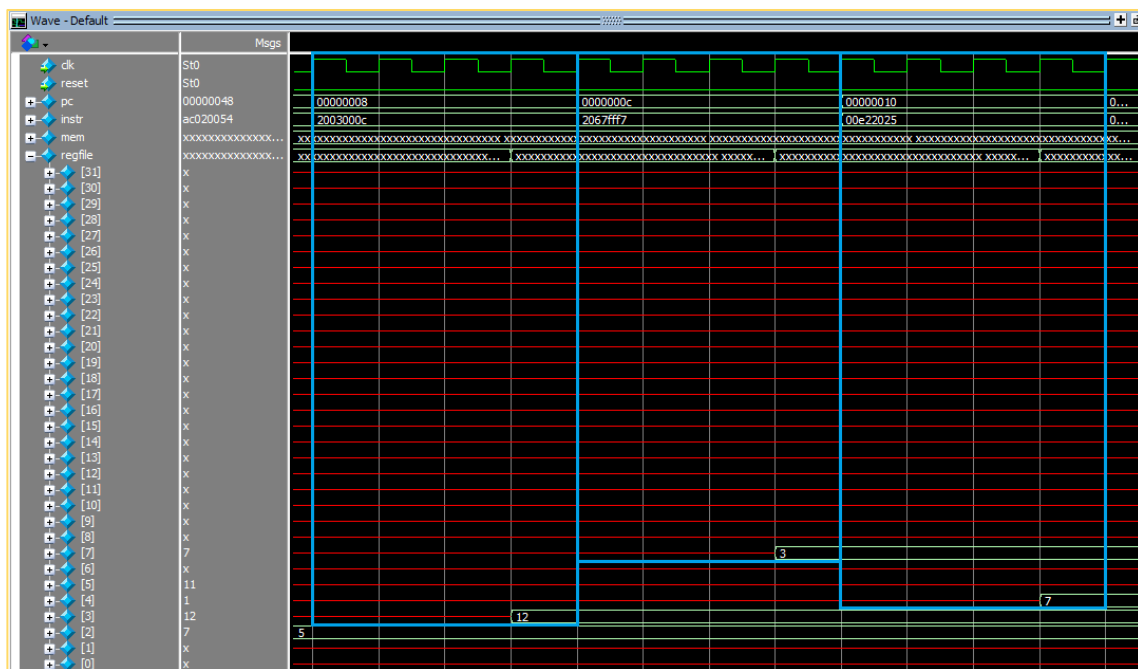


## Computer Architecture Laboratory Lab 2

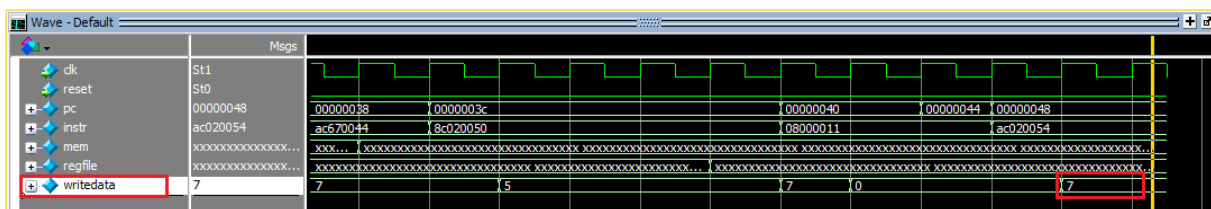
- After adding the variables and then naming them correspondingly, this is how it should look in the “wave Simulation” tab:



- Change the radix of each variables accordingly: **pc**, **instr**, and **mem[0]** to **mem[63]** -> **Hexadecimal**, **regfile[0]** to **regfile[31]**, and **writedata** -> **Decimal**. Then, save the variable settings so we can load it later during the simulation of Fibonacci sequence generator: File->Save Format->OK.
- Then run the simulation (Run -All). Notice that, since we are now simulating MIPS multi cycle processor, the instructions are executed at multiple clock cycles.



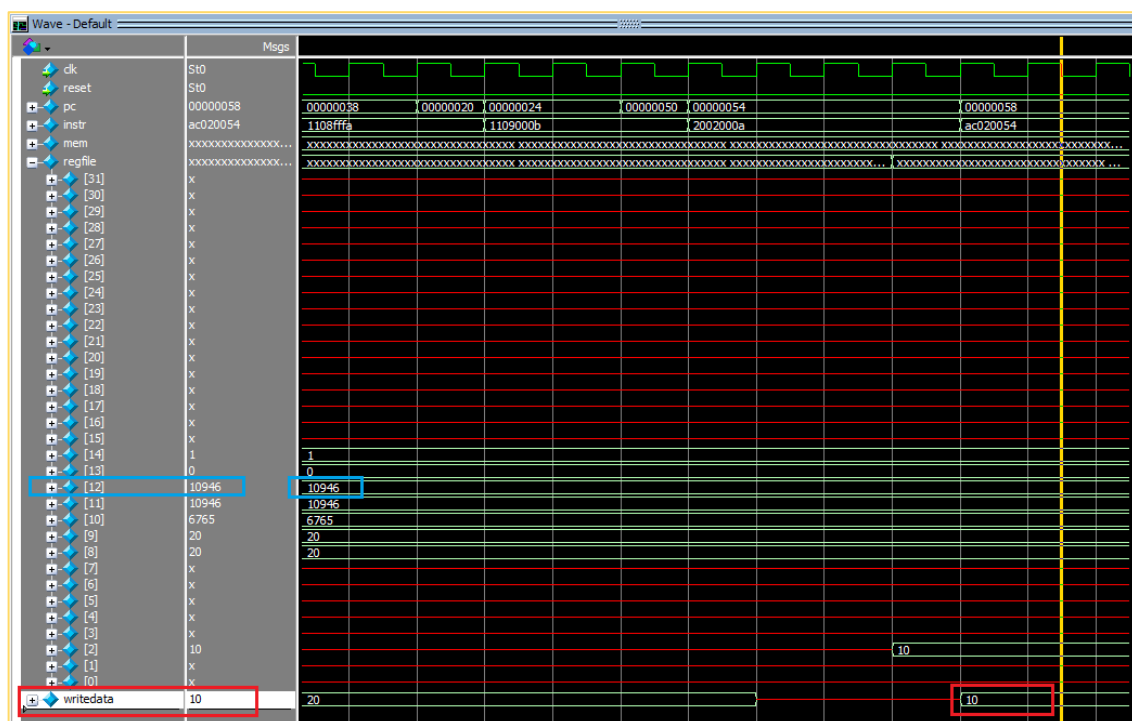
- To wrap up this simulation, the last value in the writedata variable should be 7 as indicated in the success condition of the test bench.



### B. Fibonacci Simulation in MIPS Multi Cycle

>>Now, let's run the Fibonacci sequence generator on the simple MIPS multi cycle processor.

- Create a folder named “**multi\_fibonacci**” in the “Lab2\_files” directory. Then, copy and paste the “**mips\_multi.sv**” file and also the “**fibonacci.dat**” file that contains the code.
- Follow the same procedure in sections 1-B to create a new project named “multi\_fibonacci”.
- Edit the System Verilog code as follows:
  - a) Go to the “check results” part of the “testbench” and replace number 7 with number 10 in the “writedata” If statement. [Refer to section 2-D]
  - b) Go to the “module mem” part of the System Verilog code and replace “testfile.dat” with “fibonacci.dat”. [Refer to section 2-D]
- Save the file, compile the project, and run the simulation. [Refer to section 1-C]
- To load the simulation variables (**clk**, **reset**, **pc**, **instr**, **mem**, **regfile**, and **writedata**), let’s use the saved macro setting file from section 3-A. **Copy** the “**wave.do**” from the “**mips\_multi\_test**” to the “**multi\_fibonacci**” which is directory of this simulation. This will load the same settings that we set-up earlier in section 3-A. To load the same settings: go to File->Load->Macro File and select “wave.do”.
- Then run the simulation (Run -All). The last value in register \$12 should be the 21<sup>st</sup> Fibonacci number  $f[21]=10946$  and the last value in **writedata** should be 10.



## HOMEWORK: EXTEND SIMPLE MIPS MULTI CYCLE PROCESSOR

Modify the simple MIPS multi cycle (“mips\_multi.sv”) System Verilog code to handle one new instruction: **branch if not equal (bne)**. Copy the machine code from the figure below and create a data file named “fibonacci\_bne.dat” to test the new **bne** instruction.

Machine code:	Assembly code:	Description
20080014	main: addi \$t0, \$0, 20	# set max counter (n+1=20)
20090000	addi \$t1, \$0, 0	# fibonacci initialize counter
200d0000	add \$t5, \$0, 0	# store selector
200e0001	add \$t6, \$0, 1	# store select 1
200a0000	addi \$t2, \$0, 0	# \$t2 <= 0 (fibonacci store 0)
214b0001	addi \$t3, \$t2, 1	# \$t3 <= 1 (fibonacci store 1)
000a6020	add \$t4, \$0, \$t2	# f[0] = 0 (fibonacci output)
000b6020	add \$t4, \$0, \$t3	# f[1] = 1 (fibonacci output)
1109000b	loop: beq \$t0, \$t1, end	# if fibonacci counter == max counter then branch to end
016a6020	add \$t4, \$t3, \$t2	# f[n] (fibonacci output)
11a00003	beq \$t5, \$0, trans1	# select store 1
11ae0005	beq \$t5, \$t6, trans2	# select store 2
21290001	append: addi \$t1, \$t1, 1	# append fibonacci counter
150dffffa	bne \$t0, \$t5, loop	# goto loop
000c5025	store1: or \$t2, \$0, \$t4	# store fibonacci output to \$t2
200d0001	addi \$t5, \$0, 1	# transfer counter set to 1
150dffffb	bne \$t0, \$t5, append	# goto append
000c5825	store2: or \$t3, \$0, \$t4	# \$t3 <= \$t4
200d0000	addi \$t5, \$0, 0	# transfer counter set to 0
150dffff8	bne \$t0, \$t5, append	# goto append
2002000a	end: addi \$v0, \$0, 10	# end program system call
ac020054	sw \$2, 84(\$0)	# write adr 84 = 10 (end simulation condition)

Then, present the following:

1. Show the full modified simple MIPS multi cycle (“mips\_multi.sv”) System Verilog code.
2. Indicate the changes that you made to implement the new instruction.
3. Show the output wave simulation that contains  $f[21]=10946$  in register \$12 and the value 10 in writedata variable.