

Computer System Behavioral Model Implementation

James Nguyen, Computer Scientist, San Jose State University

Abstract—This report will cover how to implement a computer system behavioral model which will be known as DaVinci v1.0. The computer system will be constructed through Verilog Hardware Description Language, tested through the ModelSim Simulation, and will include different basic arithmetic and logical operations. This report will explain how to get started on the project and load the files into the Model Sim Simulator, the required knowledge and functional specifications of the computer system, the design and implementation the different modules within DaVinci v1.0, and the testing of the different operations of these distinct modules and the computer system.

I. INTRODUCTION

Every code that programmers write to solve worldly issues, whether it be in Java or C++, are always generated into assembly code which is read by a computer system to smoothly and efficiently run the code. These unique codes that are created by the programmers would be very useless if there is no computer system as the assembly code that is generated would not be able to be read and executed.

This project will show the background processes and exemplify how the assembly code created by programmers are able to be deciphered by a computer system and be smoothly be executed. The main objective of the project is to implement a computer system, that contains a 32-bit processor and 256-megabyte memory, known as DaVinci v1.0 through Hardware Description Language or HDL. Furthermore, another objectives include being able to implement a way to test the implemented full computer system as well as the different modules within the computer system through HDL and to simulate and analyze the signal waveforms passed by the created computer system from the simulation of the test bench.

These implementations will be written through Verilog, a popular hardware description language, and will be compiled and run by ModelSim, a simulation environment for hardware description languages developed by Mentor Graphics, in which the Verilog language will be passed through.

II. GETTING STARTED

In order to get started with the project, there are several distinct steps that must be taken first. This section will provide these different directions to begin and effectively tackle the project.

A. Downloading ModelSim Simulator

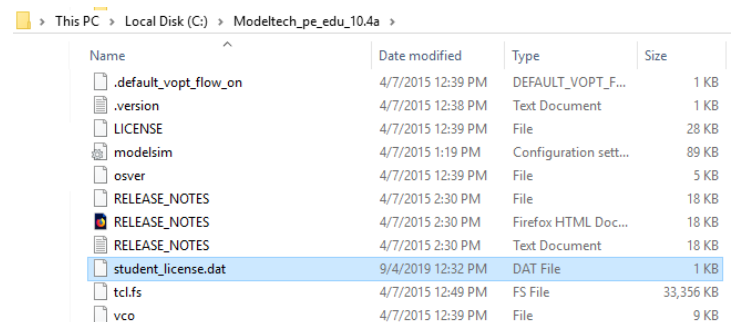
This project will be tested and ran through the ModelSim simulator by Mentor Graphics in order to generate an HDL simulation environment in which an arithmetic logic unit can be developed through the Verilog code that will be passed through. Furthermore, other simulations are costly while the

ModelSim has a Student Edition which is free for academic purposes.

The simulation, ModelSim, can be downloaded for free at: https://www.mentor.com/company/higher_ed/modelsim-student-edition

Once on the link, click the button labeled “Download Student Edition”, which the site will lead you to a form in which you will enter your email as well as other contact information to access the download link. Once the executable downloaded file is saved to the drive, open the executable file and follow the prompted instructions.

Once the installation is finished, a website will appear with a form that must be filled with your information. Once the form is filled, another email will be sent to your inbox with a file named ‘student_license.dat’. The file will need to be saved and placed into the top-level installation directory of ModelSim PE Student Edition. This can be seen in Figure 1.

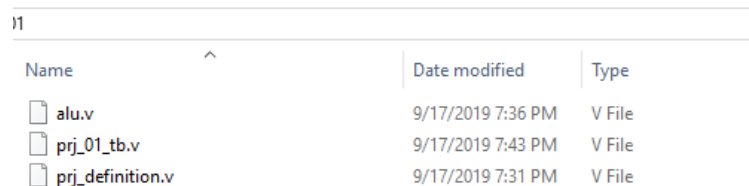


This PC > Local Disk (C:) > Modeltech_pe_edu_10.4a >				
Name	Date modified	Type	Size	
.default_vopt_flow_on	4/7/2015 12:39 PM	DEFAULT_VOPT_F...	1 KB	
.version	4/7/2015 12:38 PM	Text Document	1 KB	
LICENSE	4/7/2015 12:39 PM	File	28 KB	
modelsim	4/7/2015 1:19 PM	Configuration sett...	89 KB	
osver	4/7/2015 12:39 PM	File	5 KB	
RELEASE_NOTES	4/7/2015 2:30 PM	File	18 KB	
RELEASE_NOTES	4/7/2015 2:30 PM	Firefox HTML Doc...	18 KB	
RELEASE_NOTES	4/7/2015 2:30 PM	Text Document	18 KB	
student_license.dat	9/4/2019 12:32 PM	DAT File	1 KB	
tclfs	4/7/2015 12:49 PM	FS File	33,356 KB	
vco	4/7/2015 12:39 PM	File	9 KB	

Fig. 1. Loading the student_license.dat file into the top-level directory

B. Loading the Files & Creating the Project

The required files in order to start the project will be in a downloadable zipped folder which can be found at the following link: <https://sjsu.instructure.com/courses/1323423/files/54548243/download?wrap=1>.



11			
Name	Date modified	Type	
alu.v	9/17/2019 7:36 PM	V File	
prj_01_tb.v	9/17/2019 7:43 PM	V File	
prj_definition.v	9/17/2019 7:31 PM	V File	

Fig. 2. Required Files to start the Project

The unzipped folder should have ten Verilog files which will must be opened in the ModelSim Simulator. The Verilog files 'alu_tb.v' and 'register_file_tb.v' will need to also be created making that a total of twelve Verilog files. Each Verilog file has their own specific operation and includes starter code to help get started on the project.

The 'alu.v' file will be used to implement the arithmetic and logic unit. The 'alu_tb.v' file be used to test the ALU module that is created to ensure that the unit is working properly.

The 'clk_gen.v' file will be used to create a clock generator that the computer system will be running on.

The 'control_unit.v' file will be used to synchronize the different operations of the processor.

The da_vinci.v' will be used to create the full computer system that includes many different and unique modules. There is also the 'da_vinci_tb.v' file which will be used to test the DaVinci v1.0 computer system.

The 'memory.v' file will be used to create a 64-megabyte memory module and will be effectively test through the 'memory_64MB_tb.v' file.

The 'prj_definition.v' file will be utilized to define the bits of the different data that will be used to implement the computer system.

The 'processor.v' file will be effectively utilized to create a 32-bit processor that is able to read an instruction set known as 'CS147DV'.

Finally, the 'register_file.v' file will be used to generate a register file for the processor to use. The register file will be very similar to the memory file and will be tested through the 'register_file_tb.v' file.

Throughout the project, only the 'alu.v', 'alu_tb.v', 'register_file.v', 'register_file_tb.v', 'control_unit.v', and 'da_vinci_tb.v' will only need to be edited as the other files has the sufficient code needed to implement the computer system and should not be altered.

In order to these files, first open the ModelSim application. There will be a window that opens labeled as "IMPORTANT Information". On the bottom of the screen click on the 'Jumpstart' button as seen in figure 3. If the window cannot be seen, click on 'File' then 'New' and then on "Project", and skip the next instruction given.

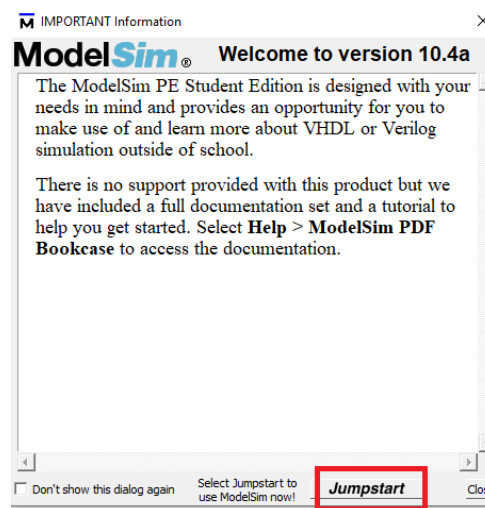


Fig. 3. Location of the Jumpstart Button.

Once the Jumpstart button is pressed, another window will pop up in which the 'Create a Project' link should be pressed on. Give the project the name, 'CS-147-PROJ2', and then choose a location for the project that can easily be accessed later.

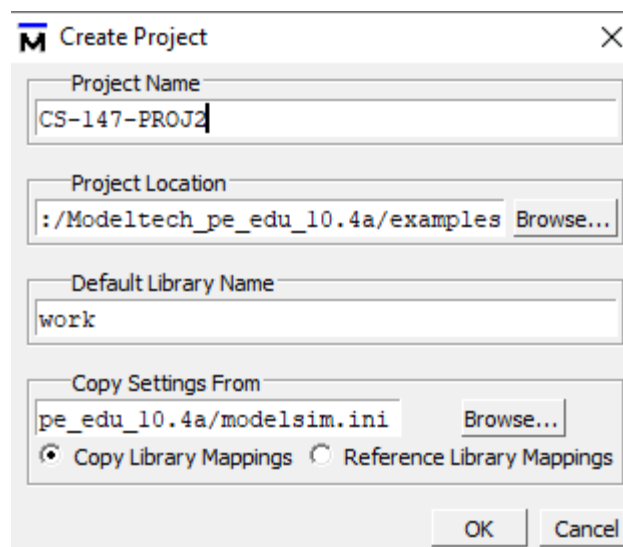


Fig. 4. Create a Project Window

Once the Project is created another window will pop up giving options, as seen in figure 5, on how files will be added to the project. In this case, since there are starter files, the 'Add Existing File' button should be clicked

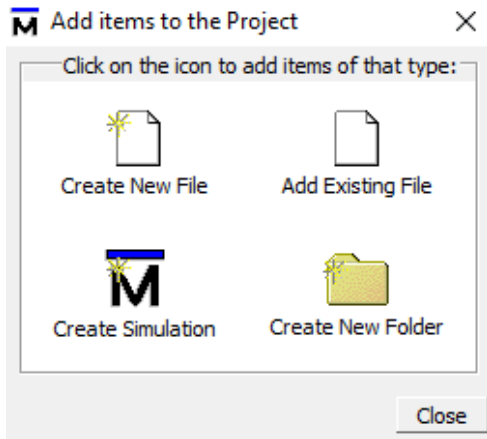


Fig. 5. Window the Pops Up After Creating the Project

After the button is pressed, there will be a 'Add file to Project' window in which the browse button should be pressed. Then, the three files that were downloaded should be selected.

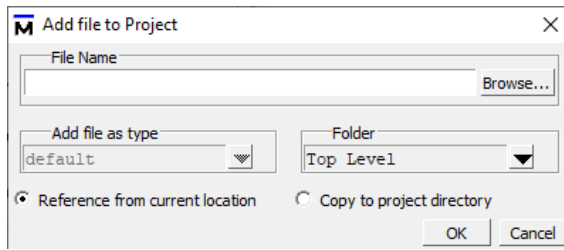


Fig. 6. 'Add file to Project' Window

These files will be added to a tab called 'Project', and to access these files, simply double click on the files which will lead to a window popping up with the Verilog code.

III. REQUIREMENTS OF COMPUTER SYSTEM

Before implementing and designing the computer system, DaVinci V1.0, there is required knowledge that surrounds a computer system that one must understand in order to successfully create the module.

This section will describe exactly what is and the different aspects surrounding a computer system and the requirements needed for the computer system to run smoothly and effectively.

A. Knowledge of CS147 DV Instruction Set Requirement

The Computer System that will be implemented throughout this project will support an instruction set known as CS147DV. The Instruction Set is created and provided by Professor Kaushik Patra of San Jose State University. This instruction set will contain three different types of instruction sets: R-type, I-type, and J-type instructions.

R or Register type instructions usually concern three registers and the main operations will utilize those three registers only. In Figure 7, it can be comprehended that R-type instructions are split into six different sections with each section containing a certain number of bits. The opcode will

be the 0x00 if the instruction is an R-type. While rs, rt, and rd will represent the different register numbers. The shamt section will be the shift amount of the operation and the funct section will be the sub operation of the opcode.

Name	Mnemonic	Format	Operation	OpCode /funct
Addition	add	R	$R[rd] = R[rs] + R[rt]$	0x00 / 0x20
Subtraction	sub	R	$R[rd] = R[rs] - R[rt]$	0x00 / 0x22
Multiplication	mul	R	$R[rd] = R[rs] * R[rt]$	0x00 / 0x2c
Logical AND	and	R	$R[rd] = R[rs] \& R[rt]$	0x00 / 0x24
Logical OR	or	R	$R[rd] = R[rs] R[rt]$	0x00 / 0x25
Logical NOR	nor	R	$R[rd] = \sim(R[rs] R[rt])$	0x00 / 0x27
Set less than	slt	R	$R[rd] = (R[rs] < R[rt]) ? 1:0$	0x00 / 0x2a
Shift left logical	sll	R	$R[rd] = R[rs] \ll \text{shamt}$	0x00 / 0x01
Shift right logical	srl	R	$R[rd] = R[rs] \gg \text{shamt}$	0x00 / 0x02
Jump Register	jrr	R	$PC = R[rs]$	0x00 / 0x08

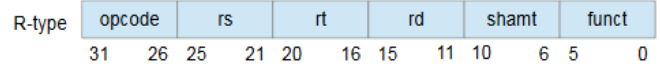


Fig. 7. R-type Instruction Format^[1]

I or Immediate type instructions will concern two different registers and a one immediate data which is a direct data sent by the users' instructions. The Immediate numbers will either be sign extended or zero extended based on the operation being performed. I-type instructions will be split into four different sections. The opcode in I-type instructions will vary based on the operation being performed. There will be two sections for two registers, rs and rt, and another section for the immediate amount.

Name	Mnemonic	Format	Operation	OpCode
Addition immediate	addi	I	$R[rt] = R[rs] + \text{SignExtImm}$	0x08
Multiplication immediate	muli	I	$R[rt] = R[rs] * \text{SignExtImm}$	0x1d
Logical AND immediate	andi	I	$R[rt] = R[rs] \& \text{ZeroExtImm}$	0x0c
Logical OR immediate	ori	I	$R[rt] = R[rs] \text{ZeroExtImm}$	0x0d
Load upper immediate	lui	I	$R[rt] = \{imm, 16'b0\}$	0x0f
Set less than immediate	slti	I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1:0$	0x0a
Branch on equal	beq	I	If $(R[rs] == R[rt])$ $PC = PC + 1 + \text{BranchAddress}$	0x04
Branch on not equal	bne	I	If $(R[rs] != R[rt])$ $PC = PC + 1 + \text{BranchAddress}$	0x05
Load word	lw	I	$R[rt] = M[R[rs] + \text{SignExtImm}]$	0x23
Store word	sw	I	$M[R[rs] + \text{SignExtImm}] = R[rt]$	0x2b

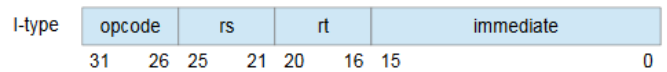


Fig. 8. I-type Instruction Format^[1]

J or Jump type instructions will contain two sections. One section is for the opcode while there is a 26-bit section for the address that the operation will jump to.

Name	Mnemonic	Format	Operation	OpCode
Jump to address	jmp	J	$PC = \text{JumpAddress}$	0x02
Jump and Link	jal	J	$R[31] = PC + 1; PC = \text{JumpAddress}$	0x03
Push to Stack	push	J	$M[\$sp] = R[0]$ $\$sp = \$sp - 1$	0x1b
Pop from Stack	pop	J	$\$sp = \$sp + 1$ $R[0] = M[\$sp]$	0x1c

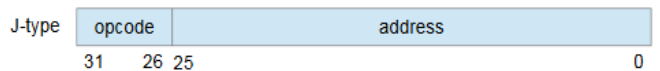


Fig. 9. J-type Instruction Format^[1]

This CS147DV Instruction set will be effectively be used throughout the this project as the computer system that

will be implemented will be able to read all of the instructions of CS147DV.

B. Arithmetic and Logic Unit Requirements

An ALU or arithmetic and logic unit is a significant digital circuit part of the Central Processing Unit (CPU). The ALU is used by the control unit within the CPU so it is required to perform logical operations such as Bitwise AND, OR, and XOR as well as arithmetic operations which includes additions, subtraction, multiplication, and division. Any complex problems that require mathematical or logical operation that arise within the computer will call upon the ALU to efficiently provide solutions.

These complex problems are broken down into basic two operand operations. For example, $(J+K*Y-Z)$ will be broken down into three main operations by the ALU:

- Result 1: $K*Y$
- Result 2: $J + \text{Result1}$
- Result 3: $\text{Result2} - Z$

As a result, the ALU is required to have three specific input and one specific output:

- Input 1: this input will be one of the operand within the the arithmetic problem
- Input 2: this input will be the other operand within the the arithmetic problem
- Input 3: this input will be the opcode which is the code that tells the ALU what kind of operation should be performed onto the other two operand inputs. This input is usually sent in by the Control Unit of the CPU. Furthermore, this input can tell the ALU to either perform an arithmetic operation such as add or logical operation such as Bitwise AND.
- Output 1: this output will be the result of performing the operation onto the the two operands.

Since the ALU is a piece of computer architecture, the unit's inputs will have a specified width depending on the scenario. In Figure 12, the ALU can be represented through this symbol in computer architecture. In this scenario, the ALU logical block has two 32 bit-width operators, a 6 bits opcode, and a 32-bit width result.

Throughout the project the ALU will be used to perform addition, subtraction, multiplication, shift right, shift left, Bitwise AND, Bitwise OR, Bitwise NOR, and set less than operations as these are the only operations that CS147 instruction set contains.

The ALU is also required to include a ZERO output which will be 1 if the output of the ALU is 0.

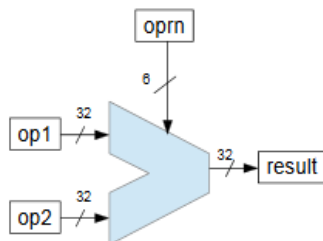


Fig. 10. ALU Logical Block^[1]

C. Register File Requirements

A register file is a unit that is usually part of the processor which is used to store data. The register file will contain 32 registers and allow the processor to quickly store data and read data to efficiently compute different operations.

The register file will contain eight different inputs and two outputs. There will be two different inputs to specify the address in the register file that is being read. There is also a 5-bit input for the address that will the register file can write to and another 32-bit input for the data that will be written into the address. There are also two outputs for the data that is read from the two different inputs with the two addresses. Furthermore, there is 1-bit input signals for read, write, reset, and clock. The register file will only read when write is set to 0 and read is set to 1, and will only write when read is set to 0 and write is set to 1.

D. Memory Requirements

The memory in this project will be a 256 MB and will be double word addressable. The memory, unlike the register file, is a separate unit from the processor and has much more space since the memory is meant to be kept and not to stay temporary.

The memory has five main inputs and one in-output. The memory has 1-bit input signals for read, write, clock, and reset. Similarly to the register file, the memory will only read when write is set to 0 and read is set to 1, and vice versa. The memory will also have a 32-bit in-output for the data which will return data from the inputted address if the signal is on read, otherwise the memory will send the data to the 10-bit inputted address if the signal is on write.

E. Control Unit Requirements

Another requirement of creating a computer system is the Control Unit. The Control Unit will be in charge on controlling all the processes with the processor, and making sure all the modules are synchronized with each other. The Control Unit will move the data between the arithmetic and logic unit, memory, and register file to perform the different instructions. The Control will hold two distinct types of registers, one for the program counter which will hold the memory address to fetch the different instructions, and another register known as the instruction register to hold the current instructions.

The Control Unit will have five different states for the unit to be able to read and execute the instructions successfully:

- **Instruction Fetch:** In this state, the control unit will fetch the instruction that is pointed from the program counter. Since this is based on the CS147DV instruction set the program counter at the beginning will be 'h0001000 and will increase per each instruction.

- **Instruction Decode:** During the state, the control unit will decode the instruction and comprehend the action that the instruction is trying to do.
- **Execution:** In this state, the control unit then executes the instructions and will use the arithmetic and logic unit if needed.
- **Memory Access:** Within this state, the control unit will access the memory if needed to read or write data from the memory.
- **Write Back:** During the state, the control unit will write back to the memory or register file if necessary.

The work between the distinct modules through the Control Unit can effectively be seen in Figure 11. As a result, it can be discerned that the Control Unit is one of the most important modules within a computer system.

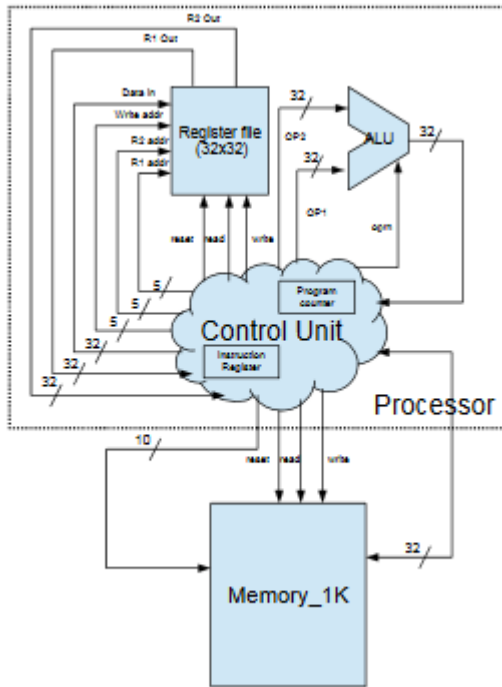


Fig. 11. Control Unit Diagram^[2]

After the Write Back state, the Control Unit will continuously go back to the Instruction Fetch state after the Write Back State. This loop will continue until there is a processor instruction to halt or if the power turns off.

F. Clock System Requirements

The Clock System of the computer system is a signal that is constantly flipping between 0 and 1. By doing this there is a synchronized pattern for the control unit to follow and switch between the different states. This allows all the modules to easily connect with one another and thus being able to make a processor.

IV. DESIGN AND IMPLEMENTATION OF ALU

With the starter code, it is given that the operands will have a 32-bit width, the opcode will be 6 bits wide, and the

result will be in 32 bits as specified in the 'prj_definition.v' file. This will be the same as the representation in Figure 12.

Moreover, the starter code further gives that the operands will be known as 'OP1' and the 'OP2', the opcode will be set as 'oprn', and the output will be known as 'OUT'. There is also an output known as 'ZERO' which checks if 'OUT' is equal to 0.

```
input [`DATA_INDEX_LIMIT:0] OP1; // operand 1
input [`DATA_INDEX_LIMIT:0] OP2; // operand 2
input [`ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

// output list
output [`DATA_INDEX_LIMIT:0] OUT; // result of the operation.
output ZERO;
```

Fig. 12. Starting Code Specifying Input & Output Names

Therefore, to complete the ALU module, there will be different operands for nine different operations due to limitations of the project. These operand numbers will be based on the function number of the R-type instructions to make it easy for the Control Unit to process. The operation and its corresponding opcode can effectively be seen in table 1.

TABLE I. OPERATION & CORRESPONDING OPRN

OPRN(Hex)	Operation
0x20	Addition
0x22	Subtraction
0x2c	Multiplication
0x02	Shift Right
0x01	Shift Left
0x24	AND
0x25	OR
0x27	NOR
0x2a	SLT
---	OUT = h'xxxxxxxx

Furthermore, case(OPRN) will be used as there are nine different cases that will occur for the nine different opcodes. The implementation of case is as below in Figure 13.

```
always @ (OP1 or OP2 or OPRN)
begin
    case (OPRN)
```

Fig. 13. Implementation of Case at the Beginning

The following sections will describe the design for each operation and how the designs will effectively be implemented as Verilog Code.

A. Addition

If 'OPRN' is equal to h'01 since the opcode is in hexadecimal 1, then addition will be performed. In this case OP1 will be added with OP2 and will returned into result.

```
`ALU_OPRN_WIDTH'h20: OUT = OP1 +OP2;
```

Fig. 14. Implementation of Addition

B. Subtraction

If 'OPRN' is equal to h'02, then subtraction case will be performed. In this case OP2 will be subtracted from OP1 and will be returned into result.

```
`ALU_OPRN_WIDTH'h22: OUT = OP1 - OP2;
```

Fig. 15. Implementation of Subtraction

C. Multiplication

If 'OPRN' is equal to h'03, then the multiplication case will be performed. In this case OP1 will multiply with OP2 and will be returned into result.

```
`ALU_OPRN_WIDTH'h2c: OUT = OP1*OP2;
```

Fig. 16. Implementation of Multiplication

D. Shift Right

If 'OPRN' is equal to h'04, then shift right case will be performed. In this case OP1 will be shifted right in bit wise by the amount specified in OP2 and will be returned into result.

```
`ALU_OPRN_WIDTH'h02: OUT = OP1 >> OP2;
```

Fig. 17. Implementation of Shift Right

E. Shift Left

If 'OPRN' is equal to h'05, then shift left case will be performed. In this case OP1 will be shifted left in bit wise by the amount specified in OP2 and will be returned into result.

```
`ALU_OPRN_WIDTH'h01: OUT = OP1 << OP2;
```

Fig. 18. Implementation of Shift Left

F. AND Logical Operation

If 'OPRN' is equal to h'06, then AND case will be performed. In this case, we will be using '&' to represent logical AND which be performed between the OP1 and OP2 and stored into the result.

```
`ALU_OPRN_WIDTH'h24: OUT = OP1 & OP2;
```

Fig. 19. Implementation of AND

G. OR Logical Operation

If 'OPRN' is equal to h'07, then OR case will be performed. In this case, we will be using '|' to represent logical OR which be performed between the OP1 and OP2 and stored into the result.

```
`ALU_OPRN_WIDTH'h25: OUT = OP1 | OP2;
```

Fig. 20. Implementation of OR

H. NOR Logical Operation

If 'OPRN' is equal to h'08, then NOR case will be performed. In this case, since there is no operation in Verilog to symbolize NOR, AND, and NOT symbols will be used to express the equivalent of nor. Since in NOR, the output is 1 if the op1 and op2 are 0, the equation $\sim OP1 \& \sim OP2$ can be used to express the relationship.

```
`ALU_OPRN_WIDTH'h27: OUT = ~(OP1 | OP2);
```

Fig. 21. Implementation of NOR

I. SLT Operation

If 'OPRN' is equal to h'09, then set on less than case will be performed. In this case, the symbol '<' will be placed

between OP1 and OP2 to test whether OP1 is truly less than OP2. If OP1 is less than OP2, 1 is returned in result, otherwise 0 is returned.

```
`ALU_OPRN_WIDTH'h2a: OUT = OP1 < OP2;
```

Fig. 22. Implementation of SLT

J. ZERO Output

After the operations are performed to compute the output of the Arithmetic and Logic Unit, the 'OUT' register should be tested to discern whether the OUT is equal to 0. This can effectively be done with a simple if then statement.

```
always @(OUT)
begin
    if (OUT == 0)
        ZERO = 1;
    else
        ZERO = 0;
end
```

Fig. 23. Implementation of ZERO output

With these implemetations, the arithmetic and logic unit should be complete as it has the basic operations that will be performed if an operand code is passed through the unit.

V. DESIGN AND IMPLEMENTATION OF REGISTER FILE

Through the register file Verilog file, the inputs and outputs are already given. The given inputs for the register file is READ, WRITE, CLK, RST, DATA_W, ADDR_R1, ADDR_R2, ADDR_W will be sent in, while DATA_R1 and DATA_R2 will be the outputs that are being sent out.

To begin coding the register file module, the module assigns DATA_R1 and DATA_R2 to be equal to 1'bz if the READ and WRITE signals are not 0 and 1, 1 and 0 respectively.

```
assign DATA_R1 = ((READ==1'b1)&&(WRITE==1'b0)) ? data_return1:({`DATA_WIDTH{1'bz}});
assign DATA_R2 = ((READ==1'b1)&&(WRITE==1'b0)) ? data_return2:({`DATA_WIDTH{1'bz}});
```

Fig. 24. Register Read and Write Signal Implementation

Furthermore a 32x32 memory storage is created for the different amount of registers within the register file. The register file will at the beginning will set the all the registers to 0. On the negative edge of the Reset signal, the register file is also reset, and all the registers will be set to 0. This can effectively be implemented through a for loop.

```
initial
begin
    for(i=0;i<=`REG_INDEX_LIMIT; i = i + 1)
        reg_32x32m[i] = { `DATA_WIDTH{1'b0}};
    end

    always @ (negedge RST or posedge CLK)
    begin
        if (RST === 1'b0)
        begin
            for(i=0;i<=`REG_INDEX_LIMIT; i = i + 1)
                reg_32x32m[i] = { `DATA_WIDTH{1'b0}};
            end
        end
```

Fig. 25. Implemetnation of resetting of Register File

The Register File will include read and write operations which can be effectively be implemented through an if statement. The if statement will test if the READ and WRITE signals are 0 and 1, or 1 and 0 respectively. If READ and Write are 0 and 1 respectively, then the memory storage in the register file with the location of ADDR_W is set to DATA_W. If the READ and WRITE signals is 1 and 0 respectively, then data on the memory location with ADDR_R1 and ADDR_R2 will be set to DATA_R1 and DATA_R2.

```
begin
  if ((READ==1'b1)&&(WRITE==1'b0)) // read operation
    data_return1 = reg_32x32m[ADDR_R1];
    data_return2 = reg_32x32m[ADDR_R2];
  if ((READ==1'b0)&&(WRITE==1'b1))// write operation
    reg_32x32m[ADDR_W] = DATA_W;
end
```

Fig. 26. Implemetnation of Reading and Writing of Register File

VI. DESGIN AND IMPLEMENTATION OF CONTROL UNIT

With the implementation of the Arithmetic and Logic Unit and Register File finished, the Control Unit will be the last module that needs to be finished since the other modules such as the memory and clock generator is all ready done with the starter code. This section will explain the different steps that occurred to implement the Control Unit which is the most important system of the processing computer system.

A. State Machine

As shown before, the Control Unit contains five different states, and in order to simulate the computer system, the Control Unit must be switched between these different states. With this program, there will be a state machine to switch the Control Unit between the different states.

The state machine will keep track of the current track as well as the next state and will switch to the next state. The table below shows the state name in the program, their definitions, and their next state.

TABLE II. STATES AND CORRESPONDING DEFINITIONS

Program State Name	Definition	Next State
`PROC_FETCH	Instruction Fetch	`PROC_DECODE
`PROC_DECODE	Instruction Decode	`PROC_EXE
`PROC_EXE	Execution	`PROC_MEM
`PROC_MEM	Memory Access	`PROC_WB
`PROC_WB	Write Back	`PROC_FETCH

To start off the state machine, the current state is set to 3'bxxx and the next state is set to `PROC_FETCH and this is also the case when the RST is on the negative edge.

```
initial
begin
  current_state = 3'bxx;
  next_state = `PROC_FETCH;
end

always @(negedge RST)
begin
  current_state = 3'bxx; //0 is PROC_FETCH
  next_state = `PROC_FETCH;
end
```

Fig. 27. Implemetnation of Start of State Machine and Restart State Machine.

After this is implemented, the control unit must also be implemented to change the state at the positive edge of the Clock System. This can simply be done by first changing the current state to the next state and then using a case system to determine the next state. This can effectively be seen in Figure 28.

```
//switching states
always @(posedge CLK)
begin
  current_state = next_state;
end
//switching next states when current states change
always @(current_state)
begin
  case(current_state)
    `PROC_FETCH : next_state = `PROC_DECODE;
    `PROC_DECODE : next_state = `PROC_EXE;
    `PROC_EXE : next_state = `PROC_MEM;
    `PROC_MEM : next_state = `PROC_WB;
    `PROC_WB : next_state = `PROC_FETCH;
  endcase
end
```

Fig. 28. Implemetnation of Changing State

With the State Machine created, the Control Unit can be implemented to read different types of instructions. These different of instructions, however, will go through the same even in the first two states.

When the state is equal to `PROC_FETCH, the memory address is set to the pc register and the memory is set to read the instruction at that specific location. The read signals will be set to both 0s.

Furthermore, when the state is at `PROC_DECODE, the instruction register will save the instruction, and then will parse the instructions for register, immediate, and jump type instructions, in order to decode the different sections of the instruction. The control unit will also calculate the sign extended of the immediate number as well as the zero extended immediate of the immediate number. The LUI and jump address is calculated as well. During this state, the register file will set its addresses to the rs and rt registers and the register file will then read from the registers.

```

if(proc_state == `PROC_DECODE)
begin
INST_REG = MEM_DATA;
//R-type
(opcode, rs, rt, rd, shamt, funct) = INST_REG;
// I-type
(opcode, rs, rt, immediate) = INST_REG;
// J-type
(opcode, address) = INST_REG; |

signExtendedImmediate = {{16{immediate[15]}},immediate} ;
zeroExtendedImmediate = {{16'h0,immediate}} ;
lui = {immediate, 16'h0};
jumpAddress = {6'b0, address};

RF_ADDR_R1_REG = rs;
RF_ADDR_R2_REG = rt;
RF_READ_REG = 1'h1;
end

```

Fig. 29. Implementation of `PROC_DECODE state

After the two states, there will be different actions and events taken depending on the type of the instruction within the further states. The following sections will explain these different that will occur based on the instruction type.

B. R-Type Instruction Design and Implementation

The first type of instruction that is read and executed by the Control Unit are R or Register type instructions. The instructions will go through distinct actions depending on the state is in, starting with the `PROC_EXE.

1) `PROC_EXE

During this state, if the function code of the instruction is equal to 0x08, then the pc register is set to the data within register s. Furthermore, if the function code is 0x01 or 0x02, the function will enter the shift amount and register into the ALU to shift. Otherwise, the control unit will add both data read from the register file into the ALU as OP1 and OP2 and set the operand as the function code.

```

//handle r-type
6'h0 :
begin
if(funct === 6'h08)
begin
PC_REG = RF_DATA_R1;
end
if(funct === 6'h01 || funct === 6'h02)
begin
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = shamt;
ALU_OPRN_REG = funct;
end
else
begin
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = RF_DATA_R2;
ALU_OPRN_REG = funct;
end
end
end

```

Fig. 30. Implementation of `PROC_EXE for R-Type Instructions

2) `PROC_MEM

Since the register type instructions only involve the register file, nothing will occur within this state as it involves the memory.

3) `PROC_WB

During this state, unless the function code is equal to 0x08, the control unit will write back to register d with the

result from the ALU. If the function code is equal to 0x08, the pc register is set to the data within register s.

```

6'h00:
begin
if(funct === 6'h08)
PC_REG = RF_DATA_R1;
else
begin
RF_ADDR_W_REG = rd;
RF_DATA_W_REG = ALU_RESULT;
RF_WRITE_REG = 1'b1;
end
end

```

Fig. 31. Implementation of `PROC_WB for R-Type Instructions

C. I-Type Instruction Design and Implementation

The next type of instruction that will be executed will be immediate type instructions. The instructions will go through different actions as well events depending on the state it is in.

1) `PROC_EXE

During this state, in the multiplication, addition, and set less than operations, register s is loaded into the ALU as well as the sign extended of the immediate. The ALU will then load the operand code depending on the operation. For the AND and OR operations, the ALU is loaded with register s and the zero extended of the immediate. For load word and store word operations register s and the sign extended of the immediate is added together through the ALU to determine the memory address to be accessed.

```

//handle i-type
6'h08 :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h20;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = signExtendedImmediate;
end
6'h1d :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h2c;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = signExtendedImmediate;
end
6'h0c :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h24;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = zeroExtendedImmediate;
end
6'h0d :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h25;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = zeroExtendedImmediate;
end
6'h0a :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h2a;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = signExtendedImmediate;
end
6'h23 :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h20;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = signExtendedImmediate;
end
6'h2b :
begin
ALU_OPRN_REG = `ALU_OPRN_WIDTH'h20;
ALU_OP1_REG = RF_DATA_R1;
ALU_OP2_REG = signExtendedImmediate;
end

```

Fig. 32. Implementation of `PROC_EXE for I-Type Instructions

2) `PROC_MEM

During this state, the memory reads the data on the address that is equal to the ALU Result if the opcode is for load word. Furthermore, the memory will write the data based on the address that is equal to ALU Result if the opcode is for store word.

```
//handle i type instruction
6'h23:
begin
    MEM_ADDR_REG = ALU_RESULT;
    MEM_WRITE_REG = 1'b0;
end

6'h2b:
begin
    MEM_ADDR_REG = ALU_RESULT;
    MEM_DATA_REG = RF_DATA_R2;
    MEM_READ_REG = 1'b0;
end

end
```

Fig. 33. Implementation of `PROC_MEM for I-Type Instructions

3) `PROC_WB

During this state, for addition, multiplication, AND, OR, and set less than operations, the ALU result is sent into register t. Furthermore, for lui, register t is set to the LUI. For branch on equal and branch on not equal, the PC is increased by the amount of the branch address or the sign extended immediate number if it meets the certain conditions.

```
//handle I-type
6'h08:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = ALU_RESULT;
    RF_WRITE_REG = 1'b1;
end

6'h0c:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = ALU_RESULT;
    RF_WRITE_REG = 1'b1;
end

6'h0d:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = ALU_RESULT;
    RF_WRITE_REG = 1'b1;
end

6'h1d:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = ALU_RESULT;
    RF_WRITE_REG = 1'b1;
end

6'h0f:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = lui;
    RF_WRITE_REG = 1'b1;
end

6'h0a:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = ALU_RESULT;
    RF_WRITE_REG = 1'b1;
end

6'h04:
begin
    if(RF_DATA_R1 == RF_DATA_R2)
    begin
        PC_REG = PC_REG + signExtendedImmediate;
    end
end
```

```
6'h05:
begin
    if(RF_DATA_R1 != RF_DATA_R2)
    begin
        PC_REG = PC_REG + signExtendedImmediate;
    end
end

6'h23:
begin
    RF_ADDR_W_REG = rt;
    RF_DATA_W_REG = MEM_DATA;
    RF_WRITE_REG = 1'b1;
end
```

Fig. 34. Implementation of `PROC_WB for I-type Instruction

D. J-Type Instruction Design and Implementation

The last type of instruction that will be executed will be the jump type instruction. The instructions will go through different actions depending on the state as well.

1) `PROC_EXE

In this state the action will only occur for J-type instructions here if the operation code is for push. If the opcode is for push, the register address is set to 0.

```
//handle j-type push
6'h1b :
begin
    RF_ADDR_R1_REG = 0;
end
```

Fig. 35. Implementation of `PROC_EXE for J-type Instruction

2) `PROC_MEM

In this state, the memory address will be read and written for the push and pop operation. The stack pointer will be increased or decreased depending on the operation.

```
//handle j type
6'h1b:
begin
    MEM_ADDR_REG = SP_REF;
    MEM_DATA_REG = RF_DATA_R1;
    MEM_READ_REG = 1'b0;
    SP_REF = SP_REF - 1;
end

6'h1c:
begin
    SP_REF = SP_REF + 1;
    MEM_ADDR_REG = SP_REF;
    MEM_WRITE_REG = 1'b0;
end
```

Fig. 36. Implementation of `PROC_MEM for J-type Instruction

3) `PROC_WB

During this state, the control unit will set the pc register to the jump address if the operation code is for jump. The control unit will also do the same if the operation code is for jump and link but will also set register 31 to the PC register. The control unit will also write to register 0 the data that is read from the memory if the operation is pop.

Fig. 37. Implementaation of `PROC_WB for J-type Instruction

VII. TEST IMPLEMENTATION OF ALU

After implementation of the ALU in Verilog, there must be some sort of testing that must occur to verify that the ALU is running smoothly and correctly.

In order to fully test the ALU, a test bench file can be used to run ALU and pass inputs and outputs to authenticate if the accuracy of the ALU. Using the 'alu_tb.v' file, Verilog code can be written to compare the answers.

The following sections will describe how the testbench was effectively used to verify if the ALU is accurately providing the correct answers.

A. Testing Using ‘Golden’

Using the golden function provided by the test bench, one can effectively set the golden to the expected answer and compare the answer to those that come out of the ALU.

```
reg [DATA_INDEX_LIMIT:0] golden; // expected result
begin
    $write("[TEST] %0d ", op1);
    case (oprn)
        `ALU_OPRN_WIDTH'h20 : begin $write("+ "); golden = op1 + op2; end
        `ALU_OPRN_WIDTH'h22 : begin $write("- "); golden = op1 - op2; end
        `ALU_OPRN_WIDTH'h2c : begin $write("* "); golden = op1 * op2; end
        `ALU_OPRN_WIDTH'h02 : begin $write(">> "); golden = op1 >> op2; end
        `ALU_OPRN_WIDTH'h01 : begin $write("<< "); golden = op1 << op2; end
        `ALU_OPRN_WIDTH'h24 : begin $write("& "); golden = op1 & op2; end
        `ALU_OPRN_WIDTH'h25 : begin $write("& "); golden = op1 & op2; end
        `ALU_OPRN_WIDTH'h27 : begin $write("nor "); golden = ~(op1 | op2); end
        `ALU_OPRN_WIDTH'h2a : begin $write("< "); golden = op1 < op2; end

        default: begin $write("? "); golden = `DATA_WIDTH'hx; end
    endcase
end
```

Fig. 38. Golden Function Used to Find Expected Values

To test these, fourteen different functions were called upon the ALU module. One of the function calls can be seen below in Figure 25. These were used to test if the golden was equivalent to the answer given by the ALU module.

```

// test 15 + 3 = 18
#5  op1_reg=15;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
                  test_golden(op1_reg,op2_reg,oprn_reg,r_net));

```

Fig. 39. Golden Function Used to Find Expected Values

After running the simulation, the following was shown on the transcript window which shows that the golden answers were exactly the same as those passed by the ALU module.

```
# [TEST] 15 + 3 = 18, got 18 ... ZERO is 0 [PASSED]
# [TEST] 15 - 5 = 10, got 10 ... ZERO is 0 [PASSED]
# [TEST] 15 + 5 = 20, got 20 ... ZERO is 0 [PASSED]
# [TEST] 20 * 10 = 200, got 200 ... ZERO is 0 [PASSED]
# [TEST] 20 >> 1 = 10, got 10 ... ZERO is 0 [PASSED]
# [TEST] 20 << 1 = 40, got 40 ... ZERO is 0 [PASSED]
# [TEST] 1 & 0 = 0, got 0 ... ZERO is 1 [PASSED]
# [TEST] 1 & 1 = 1, got 1 ... ZERO is 0 [PASSED]
# [TEST] 1 | 0 = 1, got 1 ... ZERO is 0 [PASSED]
# [TEST] 0 | 0 = 0, got 0 ... ZERO is 1 [PASSED]
# [TEST] 0 nor 0 = 4294967295, got 4294967295 ... ZERO is 0 [PASSED]
# [TEST] 2147483647 nor 2147483647 = 2147483648, got 2147483648 ... ZERO is 0 [PASSED]
# [TEST] 1 slt 10 = 1, got 1 ... ZERO is 0 [PASSED]
# [TEST] 10 slt 1 = 0, got 0 ... ZERO is 1 [PASSED]
#
#
# Total number of tests      14
# Total number of pass      14
```

Fig. 40. Golden Function Used to Find Expected Values

B. ALU Wave Testing

Another way one can test is by viewing the waves formed when the simulations occur on the operands, operator, result to confirm if the answers are as expected.

Using the waves, we can verify if the ALU is working properly. Through Figure 41, it can be concluded that the ALU is successfully working as all the waves are correct.



Fig. 41. Wave Window of ALU Testing

VIII. TEST IMPLEMENTATION OF RESGISTER FILE

To test the Register file in Verilog, a test bench file can be used to run the Register File. Using the 'register_file_tb.v' file, Verilog code can be written to see if the register file is read and writing correctly.

The following section will demonstrate how the testbench was effectively used to verify if the ALU is accurately providing the correct answers.

A. Testing Using Reading and Writing

In order to test the register file, the test bench will first write data into the register file by using a for loop and inserting a number into the register corresponding to the index to the register file. The test bench will then go through the register file and read each register to make sure the data is actually written using an if statement to compare the data. The test bench will also test if the data will return a 1b'z if the read and write signals are equal to 0 and 0 respectively.

```

// Start the operation
#10 RST=1'b0;
#10 RST=1'b1;
// Write cycle
for(i=1;i<10; i = i + 1)
begin
    DATA_W=i; READ=1'b0; WRITE=1'b1; ADDR_W = i;
end

// Read Cycle
#10 READ=1'b0; WRITE=1'b0;
#5 no_of_test = no_of_test + 1;
if (DATA_ret1 != ('DATA_WIDTH(1'b2)') && DATA_ret2 != ('DATA_WIDTH(1'b2)'))
    $write("([TEST] Read %1b, Write %1b, expecting 32'hzzzzzzzz, got %8h and %8h [FAILED]\n",
else
    no_of_pass = no_of_pass + 1;

// test of write data
for(i=0;i<10; i = i + 1)
begin
    #5 READ=1'b1; WRITE=1'b0; ADDR_R1 = i; ADDR_R2 = i;
    no_of_test = no_of_test + 1;
    if (DATA_ret1 != i && DATA_ret2 !=i)
        $write("([TEST] Read %1b, Write %1b, expecting %8h, got %8h and %8h[FAILED]\n",
    else
        no_of_pass = no_of_pass + 1;
end

```

Fig. 42. Wave Window of ALU Testing

The results will be printed if everything is passed as seen in Figure 43.

```

Total number of tests      11
Total number of pass      11

```

Fig. 43. Test Results of Register File

B. Testing Using Waves

Another way to test the register file is by reading the waves and making sure the signals, input, and outputs are all correct. The waves can effectively be seen in Figure 44 which shows that the reading and writing waves are synchronized with the clock cycle and returns the correct data that is needed.

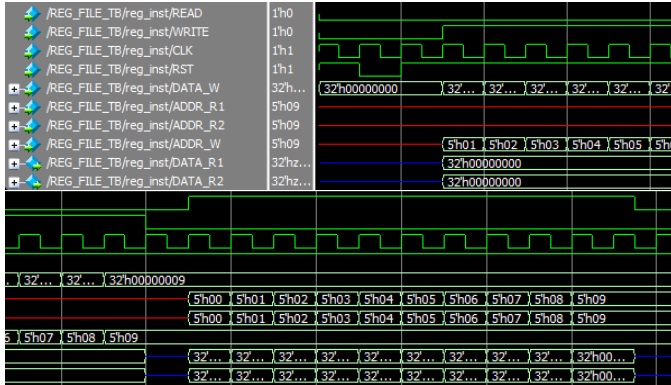


Fig. 44. Wave Window of ALU Testing

IX. TEST IMPLEMENTATION OF COMPUTER SYSTEM

Now that the Control Unit, ALU, and Register File is complete, the whole computer system can finally be tested. The testing will be done using the 'Da_Vinci_tb.v' file. The testbench provides code to test the RevFib.dat and Fibonacci.dat test file that was given with the starter code. Along with the starter code, there is also two golden files to compare with the memory dump files given by each of the data files.

The two files will run through the test bench to be compared with the golden files to check if it matches. As seen in Figure 45, the memory dumps match the golden memory dumps which means the processor is running correctly.

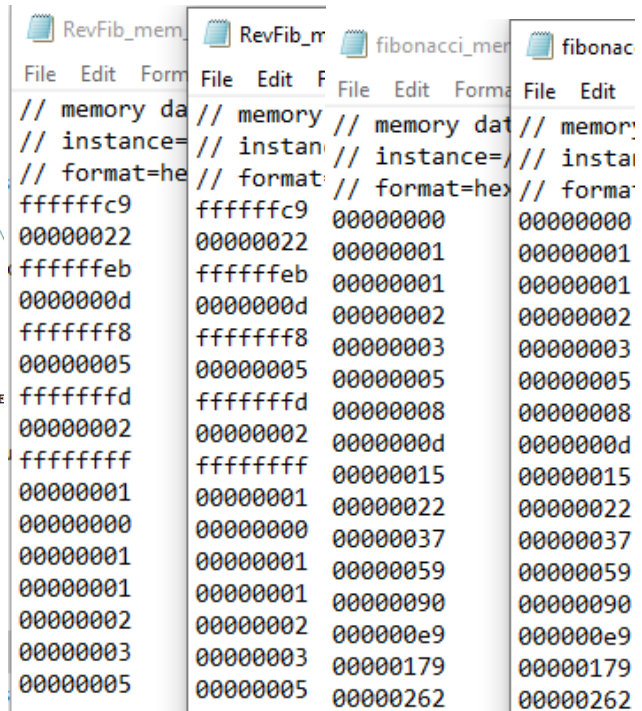


Fig. 45. Golden Testing of Computer System Simulation

Furthermore test files were created to test all of the instruction set as the two data files that were provided didn't run all of the instructions. There were also test files tested that was provided by a classmate which tests every CS147DV operation. There was also a golden file which provided the outputs matched as seen below and the memory files effectively matched.

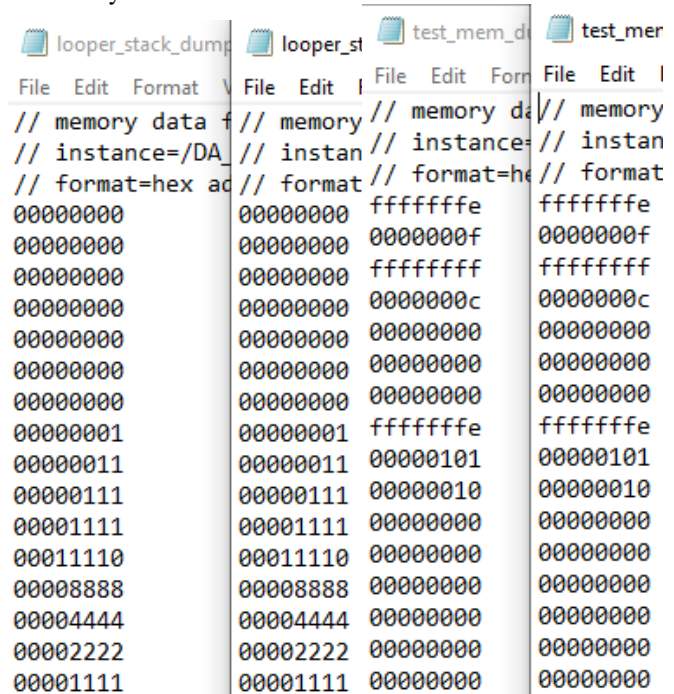


Fig. 46. Golden Testing of Computer System Simulation From Classmate

Not only that, the different test files can be tested through the waves and it was noted that the register file, memory file, arithmetic and logic unit were able to run smoothly. The portions of each distinct wave can be seen below for each of the tests.

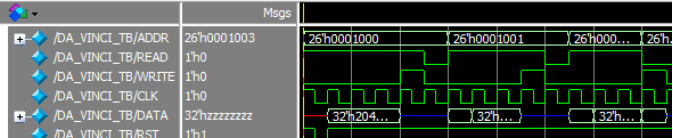


Fig. 47. Waves for Fibonacci Testing

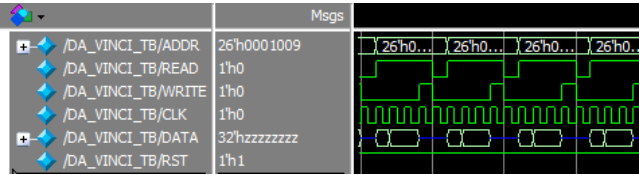


Fig. 48. Waves for RevFibonacci Testing

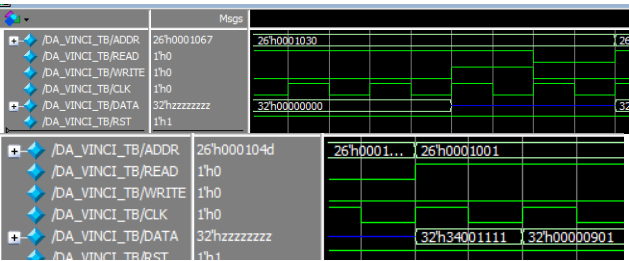


Fig. 49. Waves for Testing of all CS 147 DV

X. CONCLUSION

With the implementation of the computer system behavioral model, one is truly able to understand what occurs in the background in the computer system when a computer reads a specific code and executes it throughout its processor.

Through the project, one fundamentally learns and understand the role of the different roles of the unit in a computer system and how significant each unit is in deciphering instructions for a computer to run. This project also taught one how to use Verilog Coding, and the different aspects of the ModelSim such as the use of wave forms, and how these wave forms play an important role in reveal the underlying relationships between the different units in a computer system. The project also shows how important a computer system is and how it plays a large role in everyday technology.

.Now, whenever, we are coding in any type of language, we will be able to under what exactly the computer is doing as we execute the code and run it through the computer!

REFERENCES

[1] K. Patra. CS 147. Class Lecture, Topic: “Computer System, Instruction Set, ALU.” San Jose State University, San Jose, CA, 2014.
[2] K. Patra. CS 147. Class Lecture, Topic: “Clock, Memory, Controller, Von Neumann Arch, System SW” San Jose State University, San Jose, CA, 2014.