Behavioral Model of Processor

Yunlin Xie
San Jose State University
yunlin.xie@sjsu.edu

***Abstract*— This project goes over the process of implementing a behavioral model of a computer system. This computer system has a 256MB memory and 32-bit processor, and it supports the CS147DV Instruction Set taught in our class. This report covers how some individual parts function separately and together.**

I. INTRIDUCTION

The DaVinci v1.0 system is aimed to create a simple simulation of a computer system which includes processor and memory using Verilog code in ModelSim. There are three parts included in the processor: ALU, Register File, and Control Unit. The processor is based on the CS147DV Instruction Set. By doing this project, it helps us understand the individual functionality of each components in the processor and how they work together. We create test files for each part and for the whole Da Vinci system. We use testbench to test each part and the whole system.

II. SYSTEM REQUIREMENTS

This section will cover required parts for the processor and the memory in the DaVinci v1.0 system. The system requirements are the following ones: ALU, register file, control unit, memory, and CS147DV Instruction Set.

A. Arithmetic and Logic Unit (ALU)

The ALU is one of the components in processor. It is responsible for all calculations and logic work. In our system, the main job of our ALU is handling the following nine operations from the 'CS147DV' Instruction Set and one Zero flag.

1) Addition
2) Subtraction
3) Multiplication
4) Shift Logical Right
5) Shift Logical Left
6) Logical AND
7) Logical OR
8) Logical NOR
9) Set Less Than
10) ZERO Flag

B. Register File

The Register File is also a component of the processor. It contains 32 registers, and each register has 32-bit word. The Register File stores the data and results that ALU needs to use. Except the RESET operation, all other operations (READ and WRITE) are executed on the positive edge of the system clock. The READ operation and WRITE operation are set to HIGH-Z and the Register File will be on hold when it is 11 or 00. However, when it is 10 or 01, the READ and WRITE will execute.

C. Control Unit (CU)

The Control Unit is another part of the processor. It is responsible for fetching instruction code, controlling all signals, and directing input and output flows. The Control Unit functions based on a five-state machine. The control unit switches between the following states at positive edge of the clock cycles: Instruction Fetch (IF), Instruction Decode (ID), Execution (EXE), Memory (MEM) and Write Back (WB). All above operations are based on the CS147DV Instruction Set.

D. Memory

The Memory in our DaVinci v1.0 is a memory of 64M. It is addressable for a 32-bit storage at each address. READ and WRITE operations are executed on the positive clock edge, while the Reset operation is executed on the negative edge of the clock system. When READ or WRITE is set to 00 or 11, the DATA_R# will be set to X and the output of the memory will be on HighZ.

E. CS147DV Instruction Set

The CS147DV Instruction Set which provided and taught by Professor Patra has R-type, I-type, and J-type instructions. The instructions are used by the Control Unit, and Control Unit needs to direct the flow of all instructions.

R-type instructions need three registers. Two of them are used to perform the operation, and one register stores the result of it. They also use a 6-bits opcode (operation code), a 5-bits shift amount, and a 6-bits function code. I-type instructions need two registers, one is for storing the result. They also use a 6-bits opcode and a 16-bits immediate value. J-type is mainly dealing with jump and stack operations. They have a 6-bits opcode and a 26-bits address.

III. DESIGN AND IMPLEMENTATION

We use ModelSim as our simulator, we will do the design and implementation for required parts of our system using ModelSim. This section will outline the design and implementation for each part.

A. Arithmetic and Logic Unit (ALU)

As mentioned above, our ALU is designed to solve nine operations and a zero flag. The program will accept two operands and one operation as inputs and return the result as output. Below is the code implementation of the instructions. OP1 and OP2 are operands and we decide which operation to be carried out based on the operation code (OPRN) passed in. The number h20, h22, h2c, and etc., each represents a specific

operation. We also use zero flag to check our output values.

```
always @(OP1 or OP2 or OPRN)
begin
    case (OPRN)
        `ALU_OPRN_WIDTH'h20 : OUT = OP1 + OP2; // addition
        `ALU_OPRN_WIDTH'h22 : OUT = OP1 - OP2; // subtraction
        `ALU_OPRN_WIDTH'h2c : OUT = OP1 * OP2; // multiplaction
        `ALU_OPRN_WIDTH'h02 : OUT = OP1 >> OP2; // shift right
        `ALU_OPRN_WIDTH'h01 : OUT = OP1 << OP2; // shift left
        `ALU_OPRN_WIDTH'h24 : OUT = OP1 & OP2; // and
        `ALU_OPRN_WIDTH'h25 : OUT = OP1 | OP2; // or
        `ALU_OPRN_WIDTH'h27 : OUT = ~(OP1 | OP2); // nor
        `ALU_OPRN_WIDTH'h2a : OUT = OP1 < OP2 ? 1 : 0; // set less than
        default: OUT = `DATA_WIDTH'hxxxxxxxx;

    endcase
end
```

B. Register File

Register File has registers DATA_R1 and DATA_R2 which has the information of location. The information comes from the input of ADDR_R1 and ADDR_R2. ADDR_W is used to transport the input data from DATA_W, and then the data will be written in a register.

```
assign DATA_R1 = ((READ===1'b1)&&(WRITE===1'b0))?data_ret_1:{`DATA_WIDTH{1'bz} };
assign DATA_R2 = ((READ===1'b1)&&(WRITE===1'b0))?data_ret_2:{`DATA_WIDTH{1'bz} };
```

The Reset as mentioned above occurs on the negative edge of the RST, while Read and Write operations execute when the clock is on the positive edge. When WRITE = 1'b0 and READ = 1'b1, the register will read out data. The data will be read in when WRITE = 1'b1 and READ = 1'b0.

```
always @ (negedge RST or posedge CLK)
begin
 if (RST == 1'b0)
 begin
 for(i=0;i<=`DATA_INDEX_LIMIT; i = i + 1)
     reg_32x32[i] = { `DATA_WIDTH{1'b0} };
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read
 begin
      data_ret_1 = reg_32x32[ADDR_R1];
      data_ret_2 = reg_32x32[ADDR_R2];
 end
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write
      reg_32x32[ADDR_W] = DATA_W;
end
end
endmodule
```

C. Control Unit (CU)

The Control Unit is designed to switch between five states progressively every clock cycle. The five states are: Instruction Fetch (IF), Instruction Decode (ID), Execution (EXE), Memory (MEM) and Write Back (WB). After initializing and resetting the state machine, 'state_reg' (the current state register) will be set to 3'bxx and 'next_state' (the next state register) will be set to the next state.

```
//state switching
always@(posedge CLK)
begin

 case(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
 state_reg = next_state;
end
endmodule
```

After done the above procedure, the state machine is supposed to switch between states when the clock is on the positive edge. To implement this, we need to find out and set the correct next state to the 'next_state' variable.

```
// initiation state
initial
begin
   state_reg = 3'bxx;
   next_state = `PROC_FETCH;
end

// reset signal
always@(negedge RST)
begin
   state_reg = 2'bxx;
   next_state = `PROC_FETCH;
end
```

D. Memory

The memory has one port called 'DATA' for both input and output data. When reading from the memory, READ is 1 and WRITE is 0, also ADDR is outputted to DATA. When writing to memory, READ is 0 and WRITE is 1, also ADDR is inputted to DATA. When both READ and WRITE are equal to either 1 or 0, the DATA is on a High-Z state.

```
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read oper
        data_ret =  sram_32x64m[ADDR];
 else if ((READ===1'b0)&&(WRITE===1'b1)) // writ
        sram_32x64m[ADDR] = DATA;
end
```

The Reset occurs when at the negative edge of the system clock cycle, when RST is set to '1'b0'. When reset, 'mem_init_file' is read, and all memory data is set to zeros.

```
always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
    sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
$readmemh(mem_init_file, sram_32x64m);
end
else
```

E. CS147DV Instruction Set

R-TYPE INSTRUCTIONS

The first type of instructions needed to be implemented in the Control Unit is R-type instructions. All R-type instructions have the same opcode h'00. After making sure an instruction is R-type, we need to decide this instruction is which one of the following types: a jump register function, an ALU function using R1 and R2, an ALU function using R1 and shift amount (shamt). We do not need to know which ALU function is it, the 'funct' (function code) will let the ALU know.

```
if(proc_state === `PROC_EXE)
begin
 case (opcode)
 // R-Type
 6'h00 :
 begin
   if(funct === 6'h08)
   begin
    PC_REG = RF_DATA_R1;
   end

   else if(funct === 6'h01 || funct === 6'h02)
   begin
    ALU_OPRN_RET = funct;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = shamt;
   end

   else
   begin
     ALU_OPRN_RET = funct;
     ALU_OP1_RET = RF_DATA_R1;
     ALU_OP2_RET = RF_DATA_R2;
   end
 end
```

In the Write Back (WB) stage of the Control Unit state machine, the R-type instructions will be called again. The same, we need to identify whether an instruction is a R-type instruction first. Then, we need to know whether it is a Jump instruction or other R-type instructions.

```
//R-Type Register WriteBack
6'h00 :
begin
  if(funct === 6'h08)
    PC_REG = RF_DATA_R1;
  else
  begin
    RF_ADDR_W_RET = rd;
    RF_DATA_W_RET = ALU_RESULT;
    RF_WRITE_RET = 1'b1;
  end
end
```

I-TYPE INSTRUCTIONS

Based on the opcode, we decide which I-type we are going to run. Bne (branch if not equal) and beq (branch if equal) are not included in the implementation.

```
  //I-Type
  6'h08 :
  begin
    ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = SIGN_EXTENDED;
  end
  6'h1d :
  begin
    ALU_OPRN_RET =`ALU_OPRN_WIDTH'h2c;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = SIGN_EXTENDED;
  end
  6'h0c :
  begin
    ALU_OPRN_RET = `ALU_OPRN_WIDTH'h24;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = ZERO_EXTENDED;
  end
  6'h0d :
  begin
    ALU_OPRN_RET = `ALU_OPRN_WIDTH'h25;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = ZERO_EXTENDED;
  end
  6'h0a :
  begin
    ALU_OPRN_RET = `ALU_OPRN_WIDTH'h2a;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = SIGN_EXTENDED;
  end
  6'h23 :
  begin
    ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = SIGN_EXTENDED;
  end
  6'h2b :
  begin
    ALU_OPRN_RET =`ALU_OPRN_WIDTH'h20;
    ALU_OP1_RET = RF_DATA_R1;
    ALU_OP2_RET = SIGN_EXTENDED;
  end
```

Some I-type instructions require an immediate value in their functions. So, we need to create ZeroExtended, SignExtended, LUI, and BranchAddress for corresponding operations based on the given immediate value.

```
//Immediate sign extension
SIGN_EXTENDED = {{16{immediate[15]}},immediate};
//Immediate zero extension
ZERO_EXTENDED = {16'h0000, immediate};
//LUI value
LUI = {immediate, 16'h0000};
//Store 32-bit jumpaddress
JUMP_ADDRESS = {6'b0, address};
```

After execution, I-Type instructions that control read and write in memory should be implemented in the Memory state of the Control Unit. There are only two instructions in I-type like this: lw and sw. We distinguish between them based on the given opcode.

```
//LW
6'h23 :
begin
  MEM_ADDR_RET = ALU_RESULT;
  MEM_READ_RET = 1'b1;
end
//SW
6'h2b :
begin
  MEM_ADDR_RET = ALU_RESULT;
  MEM_DATA_RET = RF_DATA_R2;
  MEM_WRITE_RET = 1'b1;
end
```

After done all the above, we are on the Write Back stage. Same thing, we distinguish different operations based on the opcode.

```
//I-Type write back
6'h08 :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = ALU_RESULT;
 RF_WRITE_RET = 1'b1;
end
6'h1d :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = ALU_RESULT;
 RF_WRITE_RET = 1'b1;
end
6'h0c :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = ALU_RESULT;
 RF_WRITE_RET = 1'b1;
end
6'h0d :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = ALU_RESULT;
 RF_WRITE_RET = 1'b1;
end
6'h0f :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = LUI;
 RF_WRITE_RET = 1'b1;
end
6'h0a :
begin
 RF_ADDR_W_RET = rt;
 RF_DATA_W_RET = ALU_RESULT;
 RF_WRITE_RET = 1'b1;
end
6'h04 :
begin
 if(RF_DATA_R1 == RF_DATA_R2)
   PC_REG = PC_REG + SIGN_EXTENDED;
end
6'h05 :
begin
 if(RF_DATA_R1 != RF_DATA_R2)
   PC_REG = PC_REG + SIGN_EXTENDED;
end
6'h23 :
begin
  RF_ADDR_W_RET = rt;
  RF_DATA_W_RET = MEM_DATA;
  RF_WRITE_RET = 1'b1;
end
```

J-TYPE INSTRUCTIONS

There are only four operations in J-type instructions. Push operation is the only one instruction that occur at the execution stage of the Control Unit. The other three instructions occur at the Write Back state. The address of DATA_R1 is by default set to register 0. Based on test cases from the opcode, the Control Unit can know whether it is on the right state.

```
//J-Type
6'h1b :
begin
RF_ADDR_R1_RET = 0;
end
endcase
```

Because the push and pop operations need to read from memory, we need to specify corresponding behavior for them.

```
//PUSH
6'h1b :
begin
  MEM_ADDR_RET = SP_REF;
  MEM_DATA_RET = RF_DATA_R1;
  MEM_WRITE_RET = 1'b1;
  SP_REF = SP_REF - 1;
end
//POP
6'h1c :
begin
  SP_REF = SP_REF + 1;
  MEM_ADDR_RET = SP_REF;
  MEM_READ_RET = 1'b1;
end
```

Except for push instruction, all other J-type instructions are implemented in Write Back state, because they need to write into the registers and or will change the program counter.

```
//J-Type
6'h02 : PC_REG = JUMP_ADDRESS;

6'h03 :
begin
  RF_ADDR_W_RET = 31;
  RF_DATA_W_RET = PC_REG;
  RF_WRITE_RET = 1'b1;
  PC_REG = JUMP_ADDRESS;
end
6'h1c :
begin
  RF_ADDR_W_RET = 0;
  RF_DATA_W_RET = MEM_DATA;
  RF_WRITE_RET = 1'b1;
end
```

IV. TESTING COMPONENTS

In this section, we will test each component using its own test file. We will use the testbench in the ModelSim app to observe the wave and output data if needed. We will cover the following components: Arithmetic and Logic Unit (ALU), Register File (RF), Control Unit (CU), and Memory.

A. Arithmetic and Logic Unit (ALU)

We use the 'alu_tb.v' file to test the ALU. The following image show the results of the test cases we use to test the instructions, we can see this through the Transcripts window in ModelSim. We can also see the input and output data from the testbench wave if we want, but we will not include the testbench here, because the transcripts window is more readable.

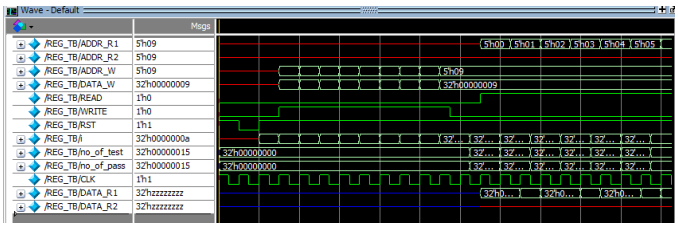
```
M 6> run -all
[TEST] 15 + 3 = 18 , got 18 ... [PASSED]
[TEST] 0 + 10 = 10 , got 10 ... [PASSED]
[TEST] 15 - 5 = 10 , got 10 ... [PASSED]
[TEST] 5 - 5 = 0 , got 0 ... [PASSED]
[TEST] 2 * 7 = 14 , got 14 ... [PASSED]
[TEST] 0 * 3 = 0 , got 0 ... [PASSED]
[TEST] 4 >> 2 = 1 , got 1 ... [PASSED]
[TEST] 4 >> 3 = 0 , got 0 ... [PASSED]
[TEST] 3 << 1 = 6 , got 6 ... [PASSED]
[TEST] 3 << 2 = 12 , got 12 ... [PASSED]
[TEST] 0 & 3 = 0 , got 0 ... [PASSED]
[TEST] 1 & 2 = 0 , got 0 ... [PASSED]
[TEST] 1 | 0 = 1 , got 1 ... [PASSED]
[TEST] 6 | 9 = 15 , got 15 ... [PASSED]
[TEST] 2147483647 ~| 0 = 2147483648 , got 2147483648 ... [PASSED]
[TEST] 6 ~| 9 = 4294967280 , got 4294967280 ... [PASSED]
[TEST] 5 < 10 = 1 , got 1 ... [PASSED]
[TEST] 1 < 1 = 0 , got 0 ... [PASSED]

      Total number of tests          18
      Total number of pass           18

** Note: $stop    : C:/Users/Yunlin/Desktop/cs147/prj_02_source/yu
   Time: 185 ns  Iteration: 0  Instance: /alu_tb
```

B. Register File

When testing the Register File, we first need to make sure the Read and Write can function normally. We use the testbench to observe the wave. We can see the RF functions correctly according to the system clock.



The test file of the RF, 'reg_tb.v', test the Write function by writing DATA_W (0-9) to register address ADDR_W. Before testing Read, the register data output is tested and READ and WRITE are set to 0. Then, we start the Read. We read back register addresses 0-9 from DATA_R1 and DATA_R2.

```
// test of write data with R1 data
for(i=0;i<10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R1 = i;
#10      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;

end

// test of write data with R2 data
for(i=0;i<10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R2 = i;
#10      no_of_test = no_of_test + 1;
        if (DATA_R2 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;

end
```

We can also refer to the transcript window, it shows that all the tests are passed successfully.

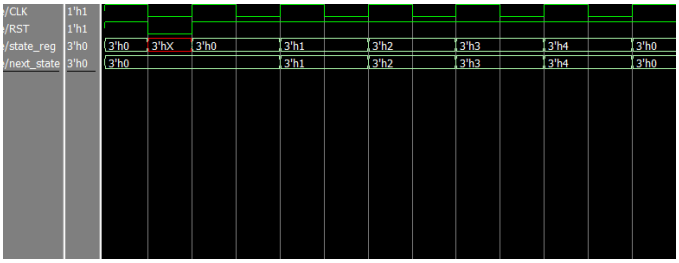
```
VSIM 7> vsim work.REG_TB
# vsim
# Start time: 17:26:04 on Oct 23,2020
# Loading work.REG_TB
# Loading work.CLK_GENERATOR
# Loading work.REGISTER_FILE_32x32
VSIM 8> run -all
#
#       Total number of tests          21
#       Total number of pass           21
#
# ** Note: $stop    : C:/Users/Yunlin/Desktop/cs14
in_prj2/reg_tb.v(106)
#    Time: 445 ns  Iteration: 0  Instance: /REG_TB
```

C. Control Unit (CU)

The Control Unit is made up of five-states state machine. To test this state machine, we want to ensure that the Control Unit is able to change states at the positive edge of the system clock. We also need to make sure the initialization and reset can function normally.

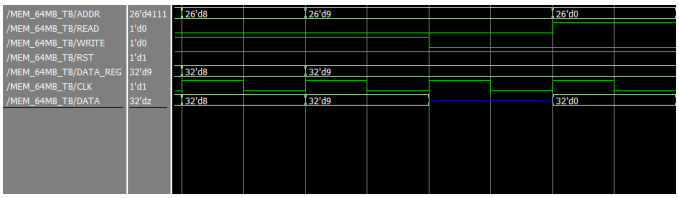
```
if ((READ===1'b1)&&(WRITE===1'b0)) // read oper
        data_ret =  sram_32x64m[ADDR];
else if ((READ===1'b0)&&(WRITE===1'b1)) // writ
        sram_32x64m[ADDR] = DATA;
end
end
```

Using the testbench, we can easily observe the wave and find out that the rest occurs at the negative edge and the state change happens at every positive clock edge.



D. Memory

We want to test Memory, because we need to make sure we read from the correct memory address and write to the correct address. From the wave table below, after reset, we can see that 0-9 are written into memory, and data is also successfully written back out of the memory.

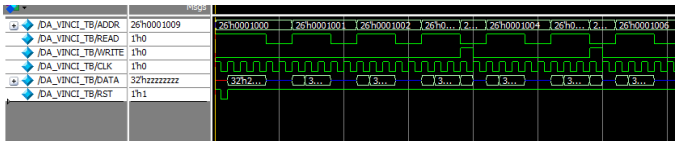


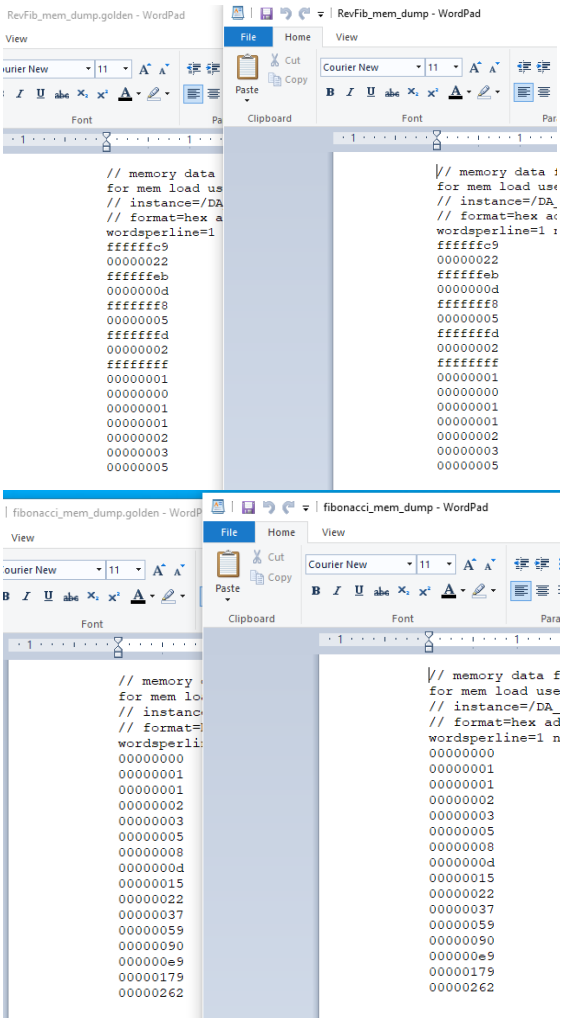
We can also observe the transcript window, it shows us that all the tests are passed successfully.

```
VSIM 3> vsim work.MEM_64MB_TB
# vsim
# Start time: 17:04:23 on Oct 23,2020
# Loading work.MEM_64MB_TB
# Loading work.CLK_GENERATOR
# Loading work.MEMORY_64MB
VSIM 4> run -all
#
#       Total number of tests          27
#       Total number of pass           27
#
# ** Note: $stop    : C:/Users/Yunlin/Desktop/cs147/prj_
#    Time: 405 ns  Iteration: 0  Instance: /MEM_64MB_TB
```

V. SYSTEM TESTING

We use the 'DaVinci_tb.v' file to test our system, and the test is based on CS147DV Instruction Set. We use the RevFib.dat and Fibonacci.dat provided by the Professor to test the out system. We basically run the program, and it will generate a file, we compare that file with the given file provided by the professor. We can also observe the wave from the testbench.





VI. CONCLUSION

By doing this project, I acquire a deep understanding of how a computer system work. The DaVinci v1.0 system presents the connections between the Processor, Memory, Register File, ALU, and Control Unit. By design, implementing, and test all parts separately and as a whole. I learned how to use the Verilog language in ModelSim to create a working microprocessor.

VII. REFERRENCES

[1] M. Morris Mano, and M. D. Ciletti, Digital Design, 4th ed., December 15, 2006.

[2] S. Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, 2nd ed., Prentice Hall PTR, February 21, 2003.

[3] K. Patra, Assignment & Module -- CS-147 Sec 01, Canvas, unpublish

[4] Y. Xie, Project 1 Report – done for CS-147 Sec 01.