Behavioral Model of a Computer System

NADA EL ZEINI
San Jose State University
nada.elzeini@sjsu.edu

***Abstract*— This report primarily deals with writing behavioral model of a bare minimum computer system DaVinci v1.0 supporting instruction set 'CS147DV'. Using Verilog, this fully functional system integrating 32- bit processor and 256MB memory (double word (32-bit) addressable 64M address) will be implemented and tested. To achieve this goal, multiple tasks will be accomplished, as described in this report.**

1) An ALU will be implemented with ZERO flag and tested.

2) A register file implementation will be done and test bench will be created similarly to the memory modeling.

3) Control unit state machine will be completed and running.

4) R-type, I-type, J-type instructions will be implemented and unit tested.

5) Full testing of DaVinci v1.0 with small program with the use of 'CS147DV' instruction set.

I. SYSTEM REQUIREMENTS

Basically, to implement the DaVinci v.1.0 system a representation (with Verilog as the hardware description language) of a processor and memory is required. The processor is a summation of a control unit (CU) , register file (RF) , and arithmetic & logic unit (ALU). Therefore, its implementation will have to integrate those 3 components. Data flow occurs in this way: Register file to ALU as 2 operands (op1 and op2), then ALU to register file as the result. Afterwards, from register file to memory to store the result to register file again to load value from memory. This section will list the requirements for the processor and memory to have a correct data flow.

A. Register File (RF)

Register file model is very similar to the memory model. It differs in size (much smaller than memory) and usage of 2 output data bus and 1 input data bus dedicated for read and write operation, instead of 1 bi-directional data bus used in memory. This file is of 32 registers ([0:31]). Like in the memory file, the reset (RST) operation is on the negative edge of the clock while read/write operations occur on the positive edge. Also, combinations of 00 and 11 of read/write signals will make the RF hold the previous value of the previously read data; not hiZ since none of the IOs are bi-directional.

B. Arithmetic & Logic Unit (ALU)

The ALU is an essential building block of a computer processor, it provides a fundamental functionality of a computer. This digital circuit is used to handle arithmetic (addition, subtraction, multiplication...) and logic operations (like AND and OR). These nine operations (set up previously in project 1) of the 'CS147DV' instruction set will be handled by the ALU in addition to 'tZERO' flag:

- Addition

- Subtraction
- Multiplication
- Shift Right
- Shift Left
- Bitwise AND
- Bitwise OR
- Bitwise NOR
- Set Less Than
- ‘ZERO’ Flag

C. Memory

The memory will have the following properties:

- 64M storing 32-bit word at each address
- Reset on negative edge of reset signal, while all other operations will be done on positive edge (synchronous).
- Read operation at read=1 and write=0
- Write operation at read=0 and write=1
- Any other combination of read/write signal will result in hiZ state of data port ‘data’.

D. Control Unit (CU)

The control unit is responsible for directing an operation between the RF, ALU and memory. It basically controls the processor to execute the ‘CS147DV’ instructions this project requires, following a 5-steps cycle changing states between: FETCH, DECODE, EXECUTE, MEMORY, and WRITE BACK.

E. Instruction Set

‘CS147DV” instruction set introduced in lecture 1 of CS147 class section 01 of Spring 2019 will be used. The 3 types of instructions Register (or R type), which allows 3 registers rs,rt and rd (destination) with a shift amount, each of 5 bits and ‘funct’ of 6 bits to specify operation (later seen); Immediate (or I type) which has only 2 registers (rt as destination and source) and 16-bit immediate (sign or zero extension might be used) and the last type is J type which consist of 6 bits of opcode as the rest of the instruction types , with a 26-bit

II. DESIGN & IMPLEMENTATION OF ALU

A. Design & Operations of ALU

The design process is simple: the ALU is treated as a module and the functions of the ALU are described using Verilog. The previously mentioned 9 foundational operations will be handled by the ALU. These operations implemented in the ‘alu.v’ file (Figure 2.1). Each operation will begin with ‘ALU_OPRN_WIDTH’h0#’ where ALU_OPRN_WIDTH is already defined to be of 6- bits in ‘prj_definition.v’ (op-code width) and where # will be replaced by the number specified for that particular operation; in a way that when calling operation 7 for example, ’h07’ will be used to refer to that operation (Bitwise OR corresponds to 7 as in the listed operations earlier). Note that when the operation, input value, or the opcode are not defined and unknown values of ‘x’ will appear indicating failure of operation. The operands are ‘op1’ and ‘op2’. The result will be in ‘OUT’.

Additionally, an extension of functionality is done to set a ‘ZERO’ output to 1 when the current ALU operation is zero and to 0 otherwise (figure 2.1). The ‘ZERO’ flag will check if the output is 0 or not.

```
always @(OP1 or OP2 or OPRN)
begin
// TBD - Code for the ALU
case (OPRN)
`ALU_OPRN_WIDTH'h01: OUT = OP1 + OP2; // integer add
`ALU_OPRN_WIDTH'h02: OUT = OP1 - OP2; //sub
`ALU_OPRN_WIDTH'h03: OUT = OP1 * OP2; // mul
`ALU_OPRN_WIDTH'h04: OUT = OP1 >> OP2; // integer shift right
`ALU_OPRN_WIDTH'h05: OUT = OP1 << OP2; // integer shift left
`ALU_OPRN_WIDTH'h06: OUT = OP1 & OP2; //bitwise AND
`ALU_OPRN_WIDTH'h07: OUT = OP1 | OP2; //OR
`ALU_OPRN_WIDTH'h08: OUT = ~(OP1 | OP2); //NOR
`ALU_OPRN_WIDTH'h09: OUT = OP1 < OP2; //set less than

default: OUT = `DATA_WIDTH'hxxxxxxxx;
endcase
//now extend functionality to set the "ZERO" output
//which is set to 1 if the result of the current ALU operation is zero
//it is set to 0 otherwise
// use of ternary operator

ZERO = (OUT === 0) ? 1 : 0;

end
```

Figure 2.1: ALU Operations

B. Testing of the ALU

A test bench file 'alu_tb.v' will assess the functionality of the ALU implemented. Each operations is tested using different numbers and multiple times. In this project, 20 operations were executed (figure 2.2 and figure 2.3).

```
VSIM 3> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 0 - 0 = 0 , got 0 ... [PASSED]
# [TEST] 5 * 10 = 50 , got 50 ... [PASSED]
# [TEST] 9 * 4 = 36 , got 36 ... [PASSED]
# [TEST] 0 * 8 = 0 , got 0 ... [PASSED]
# [TEST] 8 >> 3 = 1 , got 1 ... [PASSED]
# [TEST] 1 >> 1 = 0 , got 0 ... [PASSED]
# [TEST] 8 << 3 = 64 , got 64 ... [PASSED]
# [TEST] 2 << 1 = 4 , got 4 ... [PASSED]
# [TEST] 15 & 15 = 15 , got 15 ... [PASSED]
# [TEST] 1 & 15 = 1 , got 1 ... [PASSED]
# [TEST] 0 | 0 = 0 , got 0 ... [PASSED]
# [TEST] 15 | 15 = 15 , got 15 ... [PASSED]
# [TEST] 1 | 5 = 5 , got 5 ... [PASSED]
# [TEST] 1 ~| 4 = 4294967290 , got 4294967290 ... [PASSED]
# [TEST] 2 ~| 0 = 4294967293 , got 4294967293 ... [PASSED]
# [TEST] 9 < 1 = 0 , got 0 ... [PASSED]
# [TEST] 6 < 7 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          20
#       Total number of pass           20
#
# ** Note: $stop    : C:/Users/Nada/Downloads/Project 2/prj_02/
#    Time: 205 ns  Iteration: 0  Instance: /alu_tb
# Break in Module alu_tb at C:/Users/Nada/Downloads/Project 2/p
```

Figure 2.2: Transcript output of ALU testing

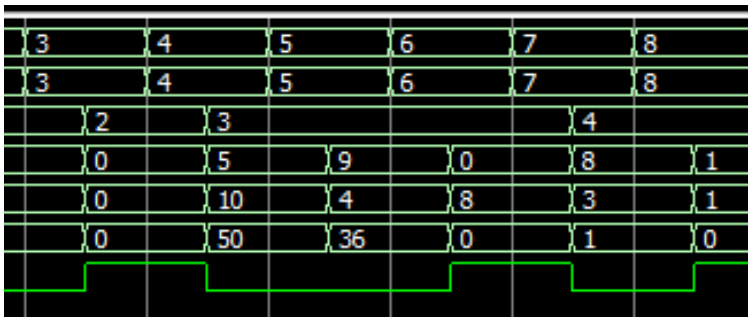

Figure 2.3: Wavelength output of ALU

III. DESIGN & IMPLEMENTATION OF MEMORY

This section will explain the modeling of a 256 MB memory with 32-bit word line, meaning we have 64M address to access the data from the memory. Figure 3.1 shows the ports required.

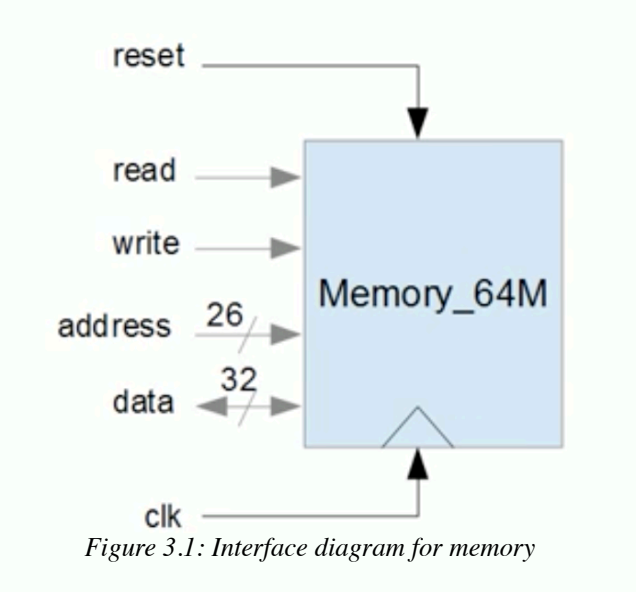

Figure 3.1: Interface diagram for memory

First of all, memory will be modeled as an array of vectors of registers (array of 26M 32-bit registers). The memory has to basically support 2 fundamental operations: Read and Write. Briefly, to explain these two operations: putting data (storing a value) into memory is referred to as 'writing' whereas getting data out is called 'reading' (reading a previously stored data). The memory doesn't distinguish between 'instruction '

and ‘data’ in terms of storage, they’re both considered to be ‘DATA’; a sequence of bit values.

All ports of the memory are inputs to the exception of ‘DATA’ which is bi-directional (InOut); so both read and write operations use the same data bus, out when it’s a READ (ADDR outputted) and in when it’s a WRITE. A return data register ‘ret_data’ is used for the read operations, in order to set the ‘DATA’ in it (assign statement). When both operations are called simultaneously, the’DATA’ will remain in HiZ state as this Verilog code shows:

- reg [`DATA_INDEX_LIMIT:0] data_ret;
- ((READ===1'b1)&&(WRITE===1'b0))? data_ret:{`DATA_WIDTH{1'bz} };

The following code provides the READ and WRITE operations (figure 3.2).

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

Figure 3.2: Read & Write operations

The ‘reset’ signal or ‘RST’ is to clear the memory content and set addresses (or indexes) back to zero. This reset operation in the memory model will be done at the negative edge of the ‘RST’ signal as mentioned in the requirements section (figure 3.3).

```
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
```

Figure 3.3: Reset Initialization

To test if the memory is working properly we use a test bench where the assignment of the data line is different such as the ‘ASSIGN’ statement will drive the data port during the WRITE operation whereas inside the memory, it drives the data port during the READ operation.

The transcript output shows that all tests were correctly done (figure 3.4) and the waveform output clearly proves that the memory is working fine and the simulation is successful (figure 3.5). In the waveform output, it is clear that when there is a 00 READ/WRITE combination the DATA port is in HiZ indicated by the blue signal.

```
# vsim -gui
# Start time: 01:33:36 on Mar 07,2019
# Loading work.MEM_64MB_TB
# Loading work.CLK_GENERATOR
# Loading work.MEMORY_64MB
add wave -position insertpoint  \
sim:/MEM_64MB_TB/ADDR \
sim:/MEM_64MB_TB/READ \
sim:/MEM_64MB_TB/WRITE \
sim:/MEM_64MB_TB/RST \
sim:/MEM_64MB_TB/CLK \
sim:/MEM_64MB_TB/DATA
VSIM 3> run -all
#
#       Total number of tests          27
#       Total number of pass           27
#
# ** Note: $stop    : C:/Users/Nada/Downloads/Pr
```

Figure 3.4: Output transcript of memory testing

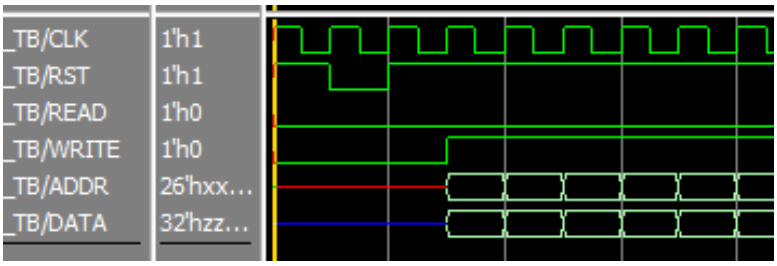

Figure 3.5: Waveform output of memory (partial view)

IV. DESIGN & TESTING OF REGISTER FILE

A. Register File Design

To be implemented is a 32 bit word accessible dual read register file. A brief definition of register file is "a state element that consists of a set of registers that can be read or written by supplying a register number to be accessed" (Computer Organization and Design) and in this case it will have 32 registers. Register file has two output data buses and one input bus dedicated for read and write operations. READ is done at READ=1 and WRTIE=0 while a WRITE operation is done at READ=0 and WRITE=1. Combinations 00 and 11 will result of an X value at DATA_R*. Logically, two data outports are used: DATA_R1 and DATA_R2 corresponding to the content at the respective register locations ADDR_R1 and ADDR_2. Additionally, data at input port DATA_W can be written at ADDR_W, which is evidently the address of the memory location to be written.

Since none of the IOs are bi-directional for register file there is no need for HiZ. However, with 'assign' statements; that ensure that data read is returned correctly (setting DATA_R* to the value of a return data register data_ret_R*) for READ operation (Figure 4.1), along with the logic of the if-else statement that will set the conditions of READ and WRITE signals (Figure 4.2) , 'x' will be the value at DATA_R* when both READ and WRITE are 0 or 1. Also, as stated earlier in the report, an essential part of this implementation is that reset is set to occur at the negative edge of reset RST signal, while rest of the operations will be done at the positive edge of the clock CLK signal (also shown in Figure 4.2).

B. Register File Testing

```
initial
begin
for(i=0; i<`DATA_INDEX_LIMIT ; i = i + 1)
 reg_32x32[i]={`DATA_WIDTH{1'b0}}; //REPLICATION
end


// d) register block  is always reset on negative ed
// same memory.v code
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

// start reading
else
begin
   if ((READ===1'b1)&&(WRITE===1'b0)) // read operat
begin
        data_ret_R1 =  reg_32x32[ADDR_R1]; // read t
        data_ret_R2 =  reg_32x32[ADDR_R2];
end
   else if ((READ===1'b0)&&(WRITE===1'b1)) // write
reg_32x32[ADDR_W] = DATA_W; // f ) on write request
end
end
// do not handle 00 or 11 for such configuration the
endmodule
```

Figure 4.2:Register File Operations

In order to test the register file a register file test bench is created similar to the memory test bench (with some noticeable changes of course). This test bench will determine if the read and write operations are being conducted correctly and if the results are being returned in the corresponding registers. There is no need to load the register file externally as done during the memory testing so to

make it simple, some data will be written in all register location (specifically numbers from 0 to 9) and will be read back. The test bench for the register file is programmed to return a transcript output of the number of passes of the 21 data that will be tested to read/write (0 to 9 twice for DATA_R* and a 00 combination test results in 21). A write cycle is first tested for DATA_W; programmed using a for loop. The wavelength output for the 'reg_tb' shows that the data was written successfully (0-9) at address ADDR_W (Figure 4.3 and 4.5).

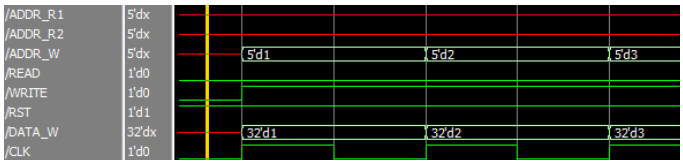

Figure 4.3: Waveform output of Register File Testing (partial)

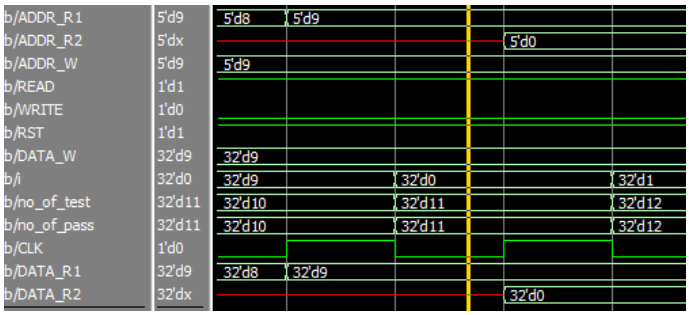

Figure 4.4: Waveform output for Register File (partial)

Afterwards, as the write cycle is over READ and WRITE are set to 00 and addresses of the memory location ADDR_R1 and ADDR_R2 are to be read for DATA_R1 and DATA_R2 respectfully (one register at a time). As the waveform shows, each 0-9 are read back from DATA_R1 first and then from DATA_R2 (Figure 4.4 and 4.5). The transcript output (Figure 4.6) as well as the wavelength output are both proof that the register file was correctly implemented.

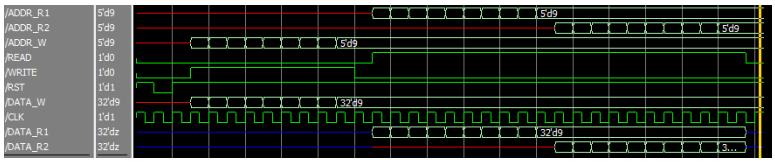

Figure 4.5: Waveform Output for Register File Testing

```
VSIM 201> run -all
#
#       Total number of tests          21
#       Total number of pass           21
#
# ** Note: $stop    : C:/Users/Nada/Downloads/
#    Time: 345 ns  Iteration: 0  Instance: /re
# Break in Module reg_tb at C:/Users/Nada/Down
```

Figure 4.6: Transcript Output of Register File Testing

V. DESIGN AND IMPLEMENTATION OF PROCESSOR

A fully functional processor supporting instruction set of 'CS147DV' is an outcome expected from this project. The 'processor.v' file should implement the processor integrating CU, RF and ALU. The implementation of these three local instances will ensure that the processor will function correctly. This section will focus on the implementation of the control unit which will

result in the fully functional system integrating memory and the processor aimed for in this project. The ‘control_unit.v’ has various inputs and outputs (Figure 5.1) and an InOut ‘MEM_DATA’ the is the data to be read from memory at the specified address (to be declared in Verilog using an intermediate return register ‘MEM_DATA_RET’).

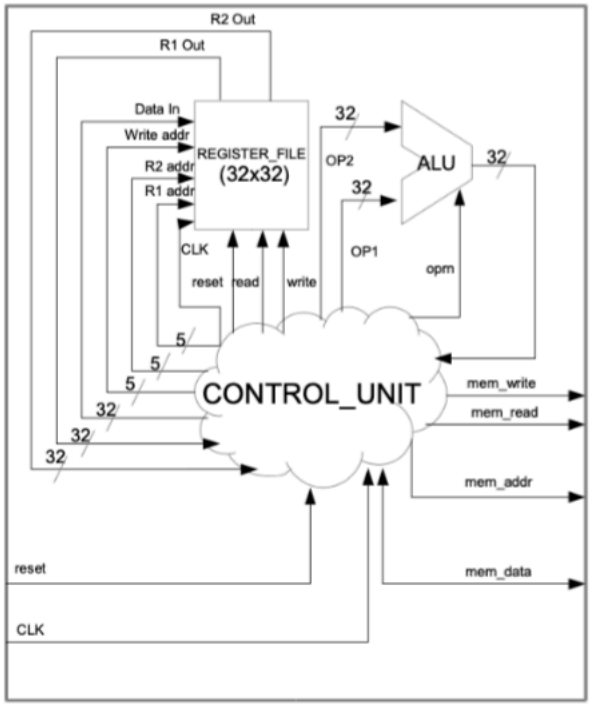

Figure 5.1: Simple Processor Schematic

A. *State Machine Control*

In ‘control_unit.v’ a module will be used to control the state machine:’ PROC_SM(STATE,CLK,RST)’. The CU should be able to cycle through five states: FETCH, DECODE, EXECUTE, MEMORY ACCESS & WRITE BACK as well as produce the required control signal at each state. As long as the system is running, there should be a regular incrementation of state from one clock cycle to another. The state machine should be reset on the negative edge of the RST, and at each positive edge of the clock CLK begin assigning the current state value ‘STATE’ with the the next value denoted ‘next_state’ in the corresponding Verilog code (Figure 5.2).

In order to begin switching between states, an initialization of states has to be done such as at both ‘reset’ and ‘initial’, next_state is set to `PROC_FETCH and the the STATE is set to 2 it unknown (2’bxx) (Figure 5.2).

```
module PROC_SM(STATE,CLK,RST);
// list of inputs
input CLK, RST;
// list of outputs
output [2:0] STATE;

// state machine implementatic
//define state and next_state
reg [2:0] STATE;
reg [2:0] next_state;

initial
begin
STATE= 2'bxx;
 next_state= `PROC_FETCH;
end

//reset signal handling

always @ (negedge RST)
begin
STATE= 2'bxx;
 next_state=`PROC_FETCH;
end
```

Figure 5.2: Initialization of State Machine

Using a case statement, at the positive edge of the clock, state switching begins as shown in

Figure 5.3. At the default case, the state and next state are set as in initialization.

```
to determine next state*/
//state switching
always @ (posedge CLK)

begin
        case(next_state)
        `PROC_FETCH :
begin
STATE=next_state;
next_state = `PROC_DECODE;
end
        `PROC_DECODE :
begin
STATE=next_state;
next_state= `PROC_EXE;
end
        `PROC_EXE :
begin
STATE=next_state;
next_state= `PROC_MEM;
end
        `PROC_MEM : begin
STATE=next_state;
next_state= `PROC_WB;
end
        `PROC_WB : begin
STATE=next_state;
next_state= `PROC_FETCH;
end
        default :
begin
STATE=2'bxx;
next_state= `PROC_FETCH;
end
endcase

end
```

Figure 5.3: State Switching for State Machine

B. Main Control Unit Module Design

At the FETCH state, the memory is prepared to read (memory control for read operation), the register file control is set on hold (00 or 11 for READ/WRITE signals). The memory address for the next instruction is set at this stage and the instruction will be fetched from the address set in the program counter PC_REG (holds program counter value) (figure 5.4).

```
//initialize
initial
begin
PC_REG = `INST_START_ADDR; //32'h00001000 from prj_definition
SP_REF = `INIT_STACK_POINTER; // 32'h03ffffff from prj_definition
end
/* Always at the change of processor state
set the control signal and address /data
signal values properly for memory, register file
and ALU */
always @ (proc_state)
begin
// Code for the control unit model
case(proc_state)

        `PROC_FETCH : begin

                        MEM_ADDR = PC_REG; //Set memory address to program counter
                        MEM_READ = 1; MEM_WRITE =0; //memory control for read operation
                        RF_READ = 1; RF_WRITE = 1; //Also set the register file control to h
                      end
```

Figure 5.4: Initialization and 'FETCH' state

Next is the DECODE state; at this point, the register file content is accessed and fetched; the read addresses of RF are set to 'rs' and 'rt' field values along with logical RF operation set to reading. MEM_DATA which is the memory read data is stored into INST_REG register intended to store the current instruction. The corresponding code will show how the instruction is parsed into the required fields for the three types of instructions. Sign extended 'IMM_SIGN_EXT' and zero extended 'IMM_ZERO_EXT' values of immediate are set, as well as LUI value for I-type instruction as shown in the Verilog code (Figure 5.5).

```
`PROC_DECODE : begin
                INST_REG = MEM_DATA ;// Store the memory read c
                // parse the instruction
                // R-type
                {opcode, rs, rt, rd, shamt, funct} = INST_REG;
                // I-type
                {opcode, rs, rt, immediate } = INST_REG;
                // J-type
                {opcode, address} = INST_REG;
                //calculate and store sign extended value of im
                IMM_SIGN_EXT = {{16{immediate[15]}},immediate};
                // same for zero extension
                IMM_ZERO_EXT = {{16{1'b0}},immediate};
                //lui value for I type
                LUI= {immediate,{16{1'b0}}};
                //For j-type inst store 32 bit jump address frc
                JumpAddr = {6'b0,address};
                //set the read address of RF as rs and rt field
                RF_ADDR_R1 = rs;
                RF_ADDR_R2 = rt;
                //with RF operation set to reading.
                RF_READ = 1; RF_WRITE = 0;
                print_instruction(INST_REG);
                end
```

Figure 5.5: DECODE state in CONTROL_UNIT Module

'print_instruction' is used to print current instruction fetched; it is very helpful when it comes to the testing process later as it helps identify any errors in the instructions and any failed operations.

At the `PROC_EXE, a case switching opcodes statement is used to set the corresponding operation accordingly (more details in later instructions design subsections).

In `PROC_MEM another case statement switching opcodes is used to be able to perform operations involving memory. By default, memory is set to HighZ. Read and Write signals are set according to the operation wanted to be set.

Write back to Register File or the Program Counter is done at `PROC_WB where RF writing address, data and control are set to write back into RF for the majority of the instructions. The PC register PC_REG is incremented by 1 by default (it is modified accordingly also to certain instructions).

C. R-Type Instruction Design

The opcode for R-type instructions is the same h'00 for every one of that type. The difference is in the 'function'. A choice between using a case block inside the principal case block used to switch between opcodes can be used to switch between functions or a series of if-else statements conditioning different functions, can be made. The 'funct' is associated with the appropriate 'ALU_OPRN' and ALU operands are mainly set to register file data 1 & 2 for the h'00 opcode and changed if needed (to shamt for instance) as seen in figure 5.6.

```
`PROC_EXE : begin
case(opcode)
//R type

6'h00: //addition
begin
ALU_OP1 = RF_DATA_R1;
ALU_OP2 = RF_DATA_R2;
if(funct === 6'h20)
begin   ALU_OPRN='h01;
        end
else if(funct === 6'h22) //substraction
begin
        ALU_OPRN='h02;
        end
else if(funct === 6'h2c) //multiply
begin
        ALU_OPRN='h03;
        end
else if(funct === 6'h24) //bitwise and
begin
        ALU_OPRN='h06;
        end
else if(funct === 6'h25) //bitwise or
begin
         ALU_OPRN='h07;
        end
else if(funct === 6'h27) //logical nor
begin
        ALU_OPRN='h08;

        end
else if(funct === 6'h2a) //set less than
begin
        ALU_OPRN='h09;

        end
else if(funct === 6'h01) //shift left
begin
```

```
        ALU_OP2 = shamt;
         ALU_OPRN='h05;
        end
else if(funct === 6'h02) //shift right
begin
        ALU_OP2 = shamt;
        ALU_OPRN='h04;
        end
 else if(funct === 6'h08)   //jump
 begin
 end
 else
 begin
 $write("Operation of R type not found or no operation was called\n");
 end
```

Figure 5.6: R-type Instructions

When reaching the Write Back stage, R-type instructions are called again (figure 5.7). This can be simply done checking if the 'funct' is referring to a jump instruction, if yes than set PC to R[rs] (operation of jr in CS147DV instruction set); if not then 'rd' is set as the register file address of the memory location to be written and the data to be written at that address is the 'ALU_RESULT' .

```
        `PROC_WB: begin
                /*write back to RF or PC_REG is done h
                //increase PC_REG by 1

                //set RF writing address, data and con
                RF_READ = 0;
                RF_WRITE=1;


case(opcode)

//r type
6'h00:
        begin
                if(funct==6'h08)
                begin
                PC_REG=RF_DATA_R1;
                end
                else
                        begin
                                RF_ADDR_W= rd;
                                RF_DATA_W=ALU_RESULT;

                        end
        end
```

Figure 5.7: R-type Instructions at Write Back

D. I-Type Instruction Design

I-type instructions differ by opcode therefore they are implemented by associating the correct opcode with the appropriate instruction. All of these instructions are set during the EXE stage of the control unit module (figure 5.8) with the exception of branch on equal 'beq' and branch not equal 'bne' which are later implemented in `PROC_WB as in figure 5.9. The immediate zero and sign extension as well as LUI that were set in the `PROC_DECODE earlier are crucial for implementing these instructions.

```
  //I type
  // differs by opcode not by funct like R-type
  6'h08:  //imm addition
  begin

       ALU_OP1 = RF_DATA_R1;
       ALU_OP2 = IMM_SIGN_EXT; ALU_OPRN = 'h01;
  end
  6'h1d: //muli
  begin

       ALU_OP1 = RF_DATA_R1;
       ALU_OP2 = IMM_SIGN_EXT; ALU_OPRN = 'h03;
  end
  6'h0c: //andi
  begin

       ALU_OP1 = RF_DATA_R1;
       ALU_OP2 = IMM_ZERO_EXT; ALU_OPRN = 'h06;
  end
  6'h0d: //ori
  begin

       ALU_OP1 = RF_DATA_R1;
       ALU_OP2 = IMM_ZERO_EXT;
       ALU_OPRN = 'h07;
  end
  6'h0f:
  begin
  end //lui //no need for alu operation
  6'h0a: //set less than imm
  begin

       ALU_OP1 = RF_DATA_R1;
       ALU_OP2 = IMM_SIGN_EXT;
           ALU_OPRN = 'h09;
  end
  6'h23: //load word
  begin
```

```
     ALU_OP1 = RF_DATA_R1;
     ALU_OP2 = IMM_SIGN_EXT;
     ALU_OPRN = 'h01;
end
//05 94 branch beq and bne no need for alu ope
6'h04: //beq
begin
ALU_OP1=RF_DATA_R1;
ALU_OP2=RF_DATA_R2;
ALU_OPRN = 'h02;
end
```

```
6'h05: //bne
begin
ALU_OP1=RF_DATA_R1;
ALU_OP2=RF_DATA_R2;
ALU_OPRN='h02;
end
6'h2b: //store word
begin

     ALU_OP1 = RF_DATA_R1;
     ALU_OP2 = IMM_SIGN_EXT; ALU_OPRN = 'h01;
end //end for I type
```

Figure 5.8: I-type Instructions at EXE

At `PROC_MEM, load word 'lw' and store word 'sw' instructions are implemented (figure 5.10) following instruction operation in figure 5.9

lw	I	R[rt] = M[R[rs]+SignExtImm]	0x23
sw	I	M[R[rs]+SignExtImm] = R[rt]	0x2b

Figure 5.9: Load word and Store word operations

```
PROC_MEM : begin
/ lw push pop instructions are involved
.t set memory to 00 or 11 hiZ
:our memory related operation set mem rea
:ss for the stack operation needs to be s
                MEM_READ=0;
                MEM_WRITE=0; //set to Hi2
                case(opcode)

                //Load Word
                6'h23:
                begin
                MEM_ADDR =ALU_RESULT;
                MEM_READ = 1;
                MEM_WRITE=0;
                RF_ADDR_W=rt;
                RF_DATA_W = MEM_DATA;
                end
                //Store Word
                6'h2b:
                begin
                  MEM_READ = 0;
    MEM_WRITE = 1;
    MEM_ADDR = ALU_RESULT;
    MEM_DATA_RET = RF_DATA_R2;
                end
```

Figure 5.10: I-type instructions at MEM

For WB, regarding I-type instructions, only two 'beq' and 'bne' need to be specified due to their particular operations (obvious in code). The rest can be implemented using a default case as shown in figure 5.9.

```
6'h04: //branch on equal beq
        //beq and bne need to modify th
        begin
        if(RF_DATA_R1 === RF_DATA_R2)
        begin
        ALU_OP1=PC_REG;
        ALU_OP2=IMM_SIGN_EXT;
        ALU_OPRN='h01; //add
        PC_REG=ALU_RESULT;
        end
        end
6'h05: //branch on not equal beq
        begin
        if(RF_DATA_R1 !== RF_DATA_R2)
        begin
        ALU_OP1=PC_REG;
        ALU_OP2=IMM_SIGN_EXT;
        ALU_OPRN='h01; //add
        PC_REG=ALU_RESULT;

default:
begin
RF_ADDR_W=rt;
RF_DATA_W=ALU_RESULT;
end
endcase
```

Figure 5.9: I-type Instructions in WB

E. J-Type Instruction Design

There are only four instructions of type J to implement. 'push' and 'pop' operations are involved in the `PROC_EXE (figure 5.11) where for 'push' operand is set to subtract but add for 'pop'; as well as in the memory access stage `PROC_MEM as 'push' will write to memory while 'pop' will read from it (figure 5.12) and the WB stage (Figure 5.13). As for 'jal' and 'jump', they will be implemented in the WB state (figure 5.13) following their operation in 'CS147DV' Instruction Set.

```
        6'h02: //jmp
        begin

        end
        6'h03: //jal
        begin

        end

        //J-type
        //jmp and jal empty no nee
        6'h1b: //PUSH TO STACK
        begin
            RF_ADDR_R1 = 0;
            ALU_OP1 = SP_REF;
            ALU_OP2=1;
            ALU_OPRN='h02; //sub 1
        end
        6'h1c: //POP FROM STACK
        begin
        RF_ADDR_R1=0;
        ALU_OP1=SP_REF;
        ALU_OP2=1;
        ALU_OPRN='h01 ;//add 1 to
        end

        default:$write("No operati
endcase
```

Figure 5.11: J-type instructions (at EXE)

```
                //Push
                6'h1b:
                begin
                 MEM_READ = 0;
    MEM_WRITE = 1;
    MEM_ADDR = SP_REF;
    MEM_DATA_RET = RF_DATA_R1;
    SP_REF = ALU_RESULT;
                end
                //pop
                6'h1c:
                begin
                  MEM_READ = 1;
    MEM_WRITE = 0;
    SP_REF= ALU_RESULT;
    MEM_ADDR = SP_REF;
    RF_DATA_W = MEM_DATA;
                end
default: $write("");
                endcase
                end
```

Figure 5.12: J-type Instructions (at MEM)

```
//to modify the PC_REG accordi
6'h02: //jump to address jmp
        begin
        PC_REG=JumpAddr-1; //p
        end
6'h03: //jal jump and link
        begin

        RF_ADDR_W=31;
        RF_DATA_W=PC_REG+1; //
        PC_REG=JumpAddr-1;

        end
6'h1b: //push to stack
        begin
        SP_REF=ALU_RESULT;
        end
6'h1c: //pop from stack
        begin
        RF_ADDR_W=0;
        RF_DATA_W=MEM_DATA;

        end
```

Figure 5.13: J-type Instructions (at WB)

VI. TEST STRATEGY AND TEST IMPLEMENTATION

A test bench is used for the testing process for the DA_VINCI system. A schematic of the DA_VINCI system is presented in figure 6.1 to show that an instance of processor and of memory correctly connected with one another is sufficient to implement a bare minimum computer system; a goal this project is aiming for.

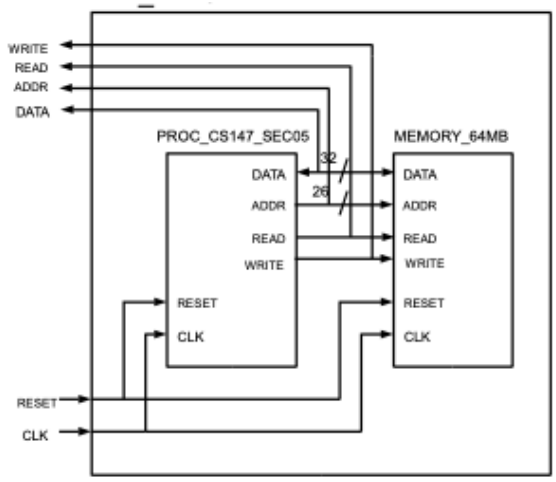

Figure 6.1: DA_VINCI schematic

Now that everything is in place, an external program (machine code), once passed into the da_vinci module (Figure 6.1) will initialize the memory with the test program.

The whole system is tested using three different memory data files (need to modify name in 'da_vinci_tb.v'). The first is basic 'fibonacci.dat' containing various instructions, then we test the reverse fibonacci file, and finally a self-created assembly code sample in 'CS147DV' instruction (figure 6.2) set is tested. This last one includes some instructions which were not tested using the previous two files.

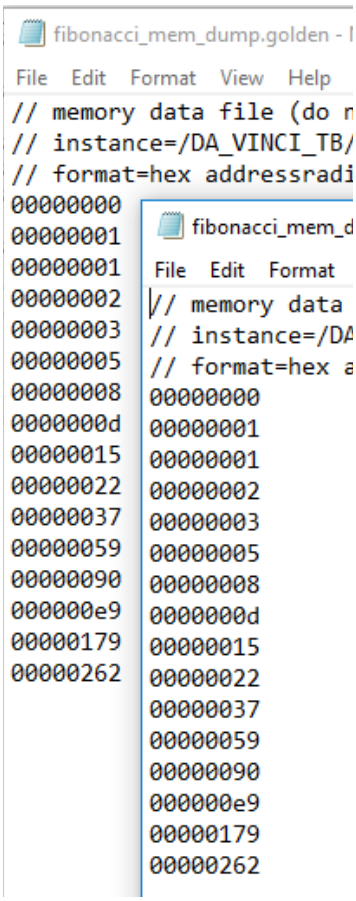

Figure 6.2: Expected result (golden on the left) vs. Test Result (on right) for Fibonacci.dat

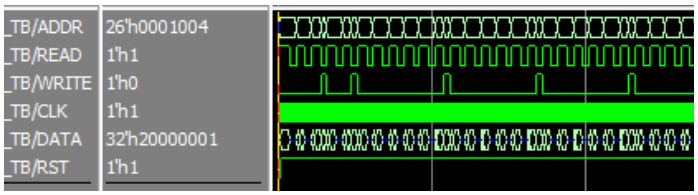

Figure 6.3: Waveform output for tested Fibonacci.dat (partial)

```
# @     20ns -> [0X20420001] addi  r[02], r[02], 0X0001;
# @     70ns -> [0X3c000100] lui   r[00], 0X0100;
# @    120ns -> [0Xac010000] sw    r[00], r[01], 0X0000;
# @    170ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @    220ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @    270ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @    320ns -> [0X00411020] add  r[02], r[01], r[02];
# @    370ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @    420ns -> [0X08001003] jmp   0X0001003;
# @    470ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @    520ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @    570ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @    620ns -> [0X00411020] add  r[02], r[01], r[02];
# @    670ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @    720ns -> [0X08001003] jmp   0X0001003;
# @    770ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @    820ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @    870ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @    920ns -> [0X00411020] add  r[02], r[01], r[02];
# @    970ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   1020ns -> [0X08001003] jmp   0X0001003;
# @   1070ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   1120ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   1170ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   1220ns -> [0X00411020] add  r[02], r[01], r[02];
# @   1270ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   1320ns -> [0X08001003] jmp   0X0001003;
# @   1370ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   1420ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   1470ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   1520ns -> [0X00411020] add  r[02], r[01], r[02];
# @   1570ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   1620ns -> [0X08001003] jmp   0X0001003;
# @   1670ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   1720ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   1770ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   1820ns -> [0X00411020] add  r[02], r[01], r[02];
# @   1870ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   1920ns -> [0X08001003] jmp   0X0001003;
# @   1970ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   2020ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   2070ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   2120ns -> [0X00411020] add  r[02], r[01], r[02];
# @   2170ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   2220ns -> [0X08001003] jmp   0X0001003;
# @   2270ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   2320ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   2370ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   2420ns -> [0X00411020] add  r[02], r[01], r[02];
# @   2470ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   2520ns -> [0X08001003] jmp   0X0001003;
# @   2570ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   2620ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   2670ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   2720ns -> [0X00411020] add  r[02], r[01], r[02];
# @   2770ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   2820ns -> [0X08001003] jmp   0X0001003;
# @   2870ns -> [0X20000001] addi  r[00], r[00], 0X0001;
```

Figure 6.6: Transcript output for tested Fibonacci.dat using 'print_instruction'(partial)

Comparisons between the resulted dump file and the expected outcome are shown respectively in figures 6.2, 6.6 and 6.9. As the relevant data memory section was dump, it is concluded that the program created the right result since dumped file and expected 'golden' outcome are identical. Waveforms can also be a way to test the result (Figures 6.3,6.7,6.10) however it would take time for such large values, and it's is in a way irrelevant when testing can be done using external file as

demonstrated. The ‘print_instruction’ task previously mentioned can help catch any errors with the instructions to be tested (figure 6.6&6.11).

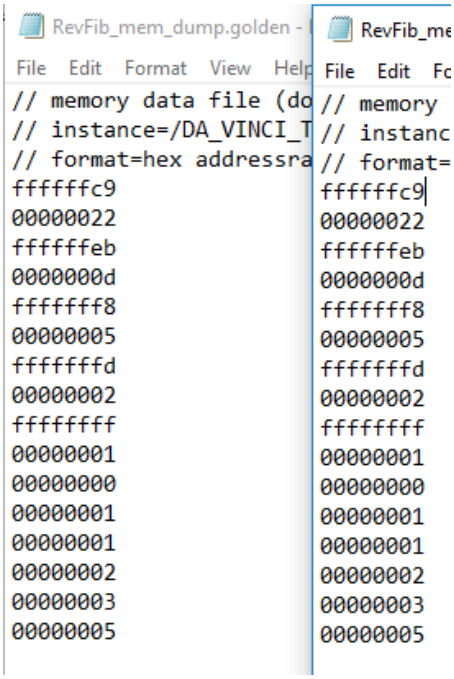

Figure 6.6: Expected result (golden on the left) vs. Test Result (on right) for RevFib.dat

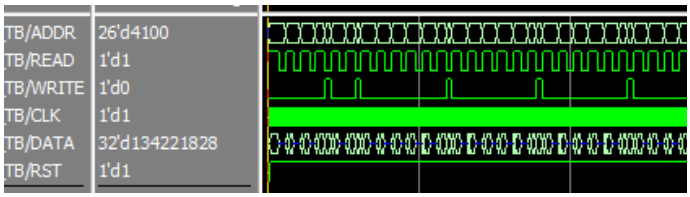

Figure 6.7: Waveform output for tested RevFib.dat (partial)

```
@0001000
20210001 //        addi r[1], r[1], 0x1
20420005 //        addi r[2], r[2], 0x5
00220020 //        add r[1], r[2], r[0] s
6c000000 //        push
00410022 //     sub r[2],r[1],r[0] should
6c000000 //        push
30430009 //     andi r[2],r[3],0x9  r[3]
0060002c //     mul r[3],r[0],r[0] should
6c000000 //     push
0c001001 //     jal 0x001001
6c000000 //     push
```

Figure 6.8: Self-created ‘testing_file.dat’

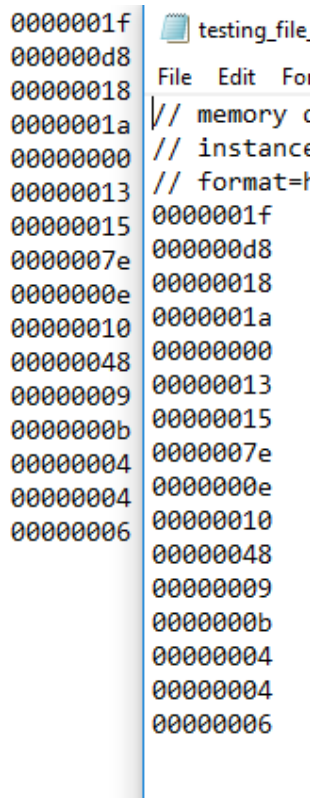

Figure 6.9: Expected result (golden on the left) vs. Test Result (on right) for testing_file.dat

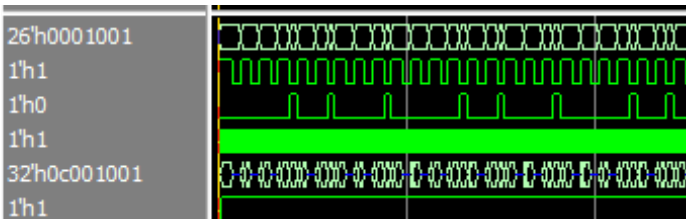

Figure 6.10: Waveform output for tested testing_file.dat (partial for better view)

```
    Time: 0 ps  Iteration: 0  Instance: /DA_VINCI_TB/da_
SIM 128> run -all
 @      20ns -> [0X20210001] addi  r[01], r[01], 0X0001;
 @      70ns -> [0X20420005] addi  r[02], r[02], 0X0005;
 @     120ns -> [0X00220020] add  r[01], r[02], r[00];
 @     170ns -> [0X6c000000] push;
 @     220ns -> [0X00410022] sub  r[02], r[01], r[00];
 @     270ns -> [0X6c000000] push;
 @     320ns -> [0X30430009] andi  r[02], r[03], 0X0009;
 @     370ns -> [0X0060002c] mul  r[03], r[00], r[00];
 @     420ns -> [0X6c000000] push;
 @     470ns -> [0X0c001001] jal   0X0001001;
 @     520ns -> [0X20420005] addi  r[02], r[02], 0X0005;
 @     570ns -> [0X00220020] add  r[01], r[02], r[00];
 @     620ns -> [0X6c000000] push;
 @     670ns -> [0X00410022] sub  r[02], r[01], r[00];
 @     720ns -> [0X6c000000] push;
 @     770ns -> [0X30430009] andi  r[02], r[03], 0X0009;
 @     820ns -> [0X0060002c] mul  r[03], r[00], r[00];
 @     870ns -> [0X6c000000] push;
 @     920ns -> [0X0c001001] jal   0X0001001;
 @     970ns -> [0X20420005] addi  r[02], r[02], 0X0005;
 @    1020ns -> [0X00220020] add  r[01], r[02], r[00];
 @    1070ns -> [0X6c000000] push;
 @    1120ns -> [0X00410022] sub  r[02], r[01], r[00];
 @    1170ns -> [0X6c000000] push;
 @    1220ns -> [0X30430009] andi  r[02], r[03], 0X0009;
 @    1270ns -> [0X0060002c] mul  r[03], r[00], r[00];
 @    1320ns -> [0X6c000000] push;
 @    1370ns -> [0X0c001001] jal   0X0001001;
 @    1420ns -> [0X20420005] addi  r[02], r[02], 0X0005;
 @    1470ns -> [0X00220020] add  r[01], r[02], r[00];
 @    1520ns -> [0X6c000000] push;
 @    1570ns -> [0X00410022] sub  r[02], r[01], r[00];
 @    1620ns -> [0X6c000000] push;
 @    1670ns -> [0X30430009] andi  r[02], r[03], 0X0009;
 @    1720ns -> [0X0060002c] mul  r[03], r[00], r[00];
 @    1770ns -> [0X6c000000] push;
 @    1820ns -> [0X0c001001] jal   0X0001001;
 @    1870ns -> [0X20420005] addi  r[02], r[02], 0X0005;
 @    1920ns -> [0X00220020] add  r[01], r[02], r[00];
 @    1970ns -> [0X6c000000] push;
 @    2020ns -> [0X00410022] sub  r[02], r[01], r[00];
 @    2070ns -> [0X6c000000] push;
```

Figure 6.11: Transcript output for tested testing_file.dat using 'print_instruction'(partial)

VII. CONCLUSION

This project provides a comprehensive model for writing and eventually creating a behavioral model of a bare minimum computer system DaVinci v1.0, particularly one supporting instruction set 'CS147DV' described in class lecture. This project completely emphasized on the importance of organization after having written and built each instance of the computer system and accurately understanding the relations and connections between each of the ALU, Register File, Memory, Control Unit and the Processor. My knowledge on these has significantly increased as the project definitely cleared some misconceptions I had, my Verilog language abilities have been highly improved as well as my usage ability of ModelSim.

VIII. REFERENCES

- Computer Organization and Design (5th edition) by David A. Patterson & John L. Hennessy.
- CS147 Section 1 Lectures SJSU Spring 2019
- CS147 Section 1 lab codes SJSU Spring 2019