Behavioral Model of a Computer System in Verilog

Joseph Nguyen
San Jose State University
College of Science, Computer Science
joseph.q.nguyen@sjsu.edu

***Abstract*—this report explores the implementation of a computer system in Verilog. The computer system will be able to read and execute a set of given instructions. The system will have many components to it and they will all be put together in ModelSim.**

I. Introduction

The objective of this project is to develop a computer system that would be able to read instructions from the given instruction set called CS147DV. The computer system will contain a 32-bit processor as well as a 256MB memory. We will also have to create a register file, alu, and control unit. The system will be tested in the da_vinci test bench.

II. Requirements for the System

A. Arithmetic Logic Unit with Zero Output

This ALU model will be very similar to the one from the first project. The only difference is that this ALU will have an extra output ZERO. If the result is equal to zero, the output will be 1, otherwise it will be 0.

B. 32 X 32 Register File

The register file that will be used in this project will contain 32 registers each holding 32 bits of data. The register file is very important as it is used to read and write data that is used in the instructions.

C. 64Mb Memory

Similar the register file, the 64 Mb memory will be word addressable, which means that each memory cell will hold 32 bits of data. This memory model is crucial to the computer system since the instruction set will require the access of memory.

D. Control Unit

The control unit is responsible for simulating the steps of the process that takes place in reading and executing an instruction. The steps are as follow: *Instruction Fetch*, *Instruction Decode*, *Execute*, *Memory Access*, and *Write Back.*

E. Processor

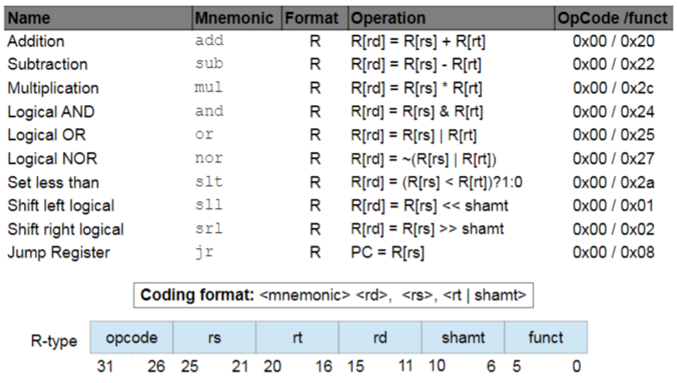
The 32-bit processor model will hold the instances ALU, Register File, and Control Unit. This component is important as it represents the processors that we have studied. It is also a core component of the computer system.

F. DaVinci

In this project, the name of the computer system will be DaVinci. This computer system will hold the instances of the Processor as well as 64 Mb Memory. The computer system will be responsible for allowing the Processor and Memory to work together to process and execute instructions from a DAT file. The instructions will be the bare minimum set called CS147DV.

G. CS147DV Instruction Set

The reason why this instruction set is considered bare minimum is because it has a limited amount of instructions. Actual instructions have many instructions which allow for more flexible programs, but because this instruction set is limited, we can only write using basic instructions. The instruction set will contain the three basic types of intructions R, I and J-type.

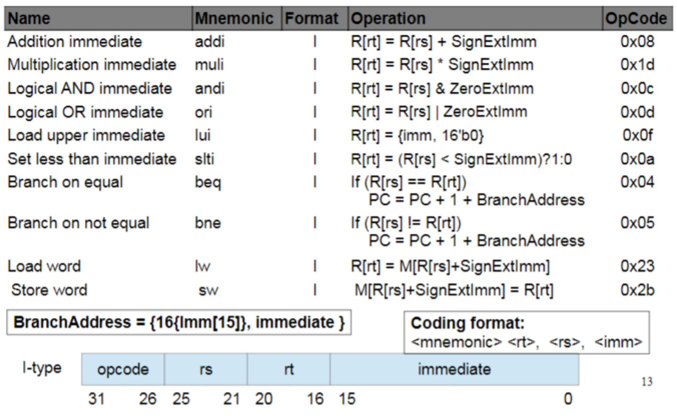


Name	Mnemonic	Format	Operation	OpCode /funct
Addition	add	R	R[rd] = R[rs] + R[rt]	0x00 / 0x20
Subtraction	sub	R	R[rd] = R[rs] - R[rt]	0x00 / 0x22
Multiplication	mul	R	R[rd] = R[rs] * R[rt]	0x00 / 0x2c
Logical AND	and	R	R[rd] = R[rs] & R[rt]	0x00 / 0x24
Logical OR	or	R	R[rd] = R[rs] | R[rt]	0x00 / 0x25
Logical NOR	nor	R	R[rd] = ~(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] << shamt	0x00 / 0x01
Shift right logical	srl	R	R[rd] = R[rs] >> shamt	0x00 / 0x02
Jump Register	jr	R	PC = R[rs]	0x00 / 0x08

Coding format: <mnemonic> <rd>, <rs>, <rt | shamt>

R-type	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0

Fig. 1. CS147DV R-Type Instructions



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

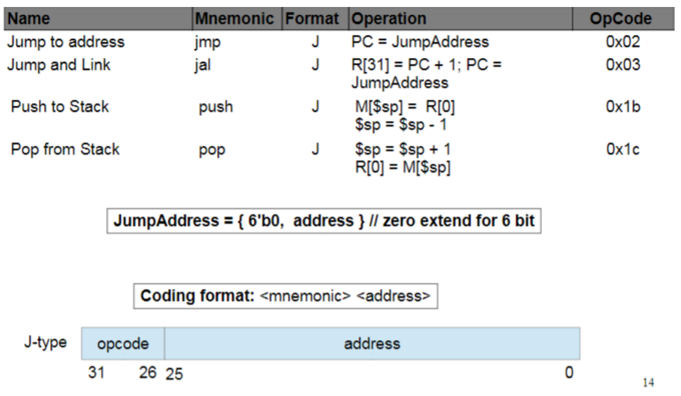
BranchAddress = {16{Imm[15]}, immediate }

Coding format:
<mnemonic> <rt>, <rs>, <imm>

I-type	opcode	rs	rt	immediate
	31 26	25 21	20 16	15 0

13

Fig. 2. CS147DV I-Type Instructions



Name	Mnemonic	Format	Operation	OpCode
Jump to address	jmp	J	PC = JumpAddress	0x02
Jump and Link	jal	J	R[31] = PC + 1; PC = 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

JumpAddress = { 6'b0, address } // zero extend for 6 bit

Coding format: <mnemonic> <address>

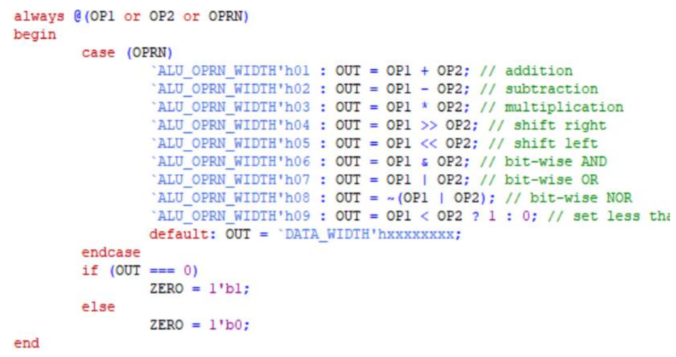
J-type	opcode	address
	31 26	25 0

14

Fig. 3. CS147DV J-Type Instructions

III. Design and Implimentation of ALU

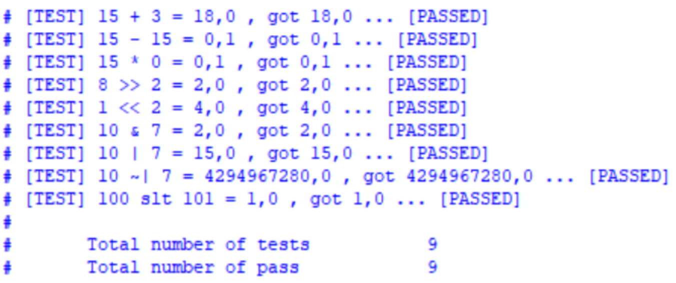
The ALU will take three inputs OP1, OP2, and OPRN. The ALU will have two outputs OUT and ZERO. We will represent this component as a module in ModelSim. Similar to project 1, at the change of OP1, OP2, or OPRN, the results of the operation based on the inputs will be stored in OUT. If OUT is equal to 0, then ZERO will output 1, otherwise 0.



```
always @(OP1 or OP2 or OPRN)
begin
        case (OPRN)
                `ALU_OPRN_WIDTH'h01 : OUT = OP1 + OP2; // addition
                `ALU_OPRN_WIDTH'h02 : OUT = OP1 - OP2; // subtraction
                `ALU_OPRN_WIDTH'h03 : OUT = OP1 * OP2; // multiplication
                `ALU_OPRN_WIDTH'h04 : OUT = OP1 >> OP2; // shift right
                `ALU_OPRN_WIDTH'h05 : OUT = OP1 << OP2; // shift left
                `ALU_OPRN_WIDTH'h06 : OUT = OP1 & OP2; // bit-wise AND
                `ALU_OPRN_WIDTH'h07 : OUT = OP1 | OP2; // bit-wise OR
                `ALU_OPRN_WIDTH'h08 : OUT = ~(OP1 | OP2); // bit-wise NOR
                `ALU_OPRN_WIDTH'h09 : OUT = OP1 < OP2 ? 1 : 0; // set less tha
                default: OUT = `DATA_WIDTH'hxxxxxxxx;
        endcase
        if (OUT === 0)
                ZERO = 1'b1;
        else
                ZERO = 1'b0;
end
```

Fig. 4. The main Verilog code for the module ALU

To test the ALU module, I have modified the testbench from project 1 to include the output for ZERO



```
# [TEST] 15 + 3 = 18,0 , got 18,0 ... [PASSED]
# [TEST] 15 - 15 = 0,1 , got 0,1 ... [PASSED]
# [TEST] 15 * 0 = 0,1 , got 0,1 ... [PASSED]
# [TEST] 8 >> 2 = 2,0 , got 2,0 ... [PASSED]
# [TEST] 1 << 2 = 4,0 , got 4,0 ... [PASSED]
# [TEST] 10 & 7 = 2,0 , got 2,0 ... [PASSED]
# [TEST] 10 | 7 = 15,0 , got 15,0 ... [PASSED]
# [TEST] 10 ~| 7 = 4294967280,0 , got 4294967280,0 ... [PASSED]
# [TEST] 100 slt 101 = 1,0 , got 1,0 ... [PASSED]
#
#       Total number of tests           9
#       Total number of pass            9
```

Fig. 5. Transcipt for ALU_TB

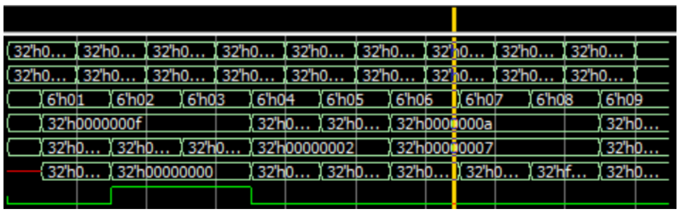
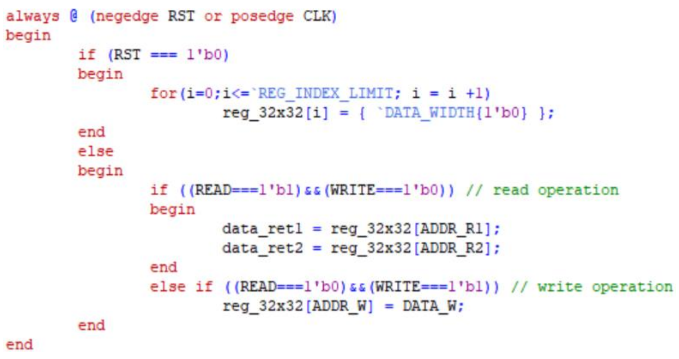


Fig. 6. Wave signals for ALU_TB

IV. Design and Implimentation of REGISTER_FILE_32X32

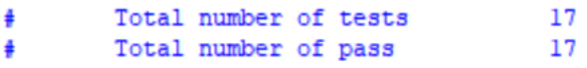
REGISTER_FILE_32X32 will take eight inputs READ, WRITE, CLK, RST, ADDR_R1, ADDR_R2, ADD_W, DATA_W and will give two outputs DATA_R1, DATA_R2. During a read operation, the register file will read the contents of registers at addresses ADDR_R1 and ADDR_R2 and will store the respected values at DATA_R1 and DATA_R2. During a write operation, the register file will store the contents DATA_W in the register at address DATA_W. The reason why this register file is able to read from two registers in parallel is so that it can read from two operands at once for the ALU.



```
always @ (negedge RST or posedge CLK)
begin
        if (RST === 1'b0)
        begin
                for(i=0;i<=`REG_INDEX_LIMIT; i = i +1)
                        reg_32x32[i] = { `DATA_WIDTH{1'b0} };
        end
        else
        begin
                if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
                begin
                        data_ret1 = reg_32x32[ADDR_R1];
                        data_ret2 = reg_32x32[ADDR_R2];
                end
                else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
                        reg_32x32[ADDR_W] = DATA_W;
        end
end
```

Fig. 7. The main Verilog code for the model REGISTER_FILE_32X32

To test the register file, I implemented a testbench that would write data into the register file and then read the same data back and compare them.



```
#       Total number of tests           17
#       Total number of pass            17
```

Fig. 8. Transcipt for REGISTER_FILE_TB

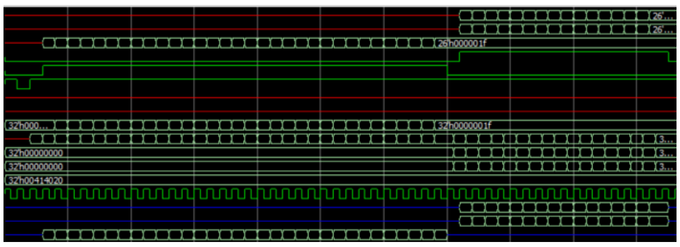
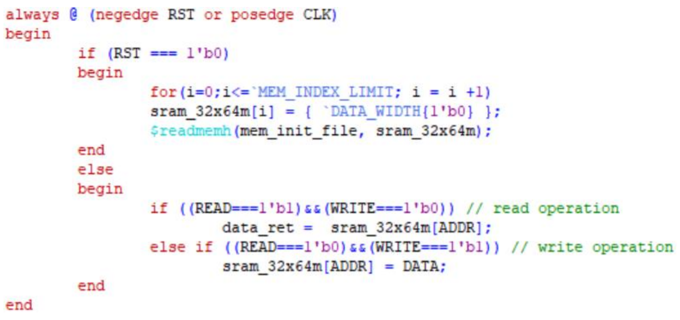


Fig. 9. Wave signals for REGISTER_FILE_TB

V. Design and Implimentation of MEMORY_64MB

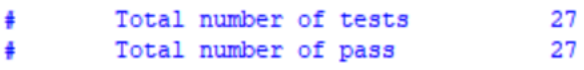
MEMORY_64MB will take five inputs READ, WRITE, CLK, RST, ADDR and will also take an inout DATA. During a read operation, content from the memory address ADDR will be read and stored in DATA. During a write operation, content from DATA will be stored in memory address ADDR.



```
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
        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
```

Fig. 10. Main Verilog code for MEMORY_64MB

To test the memory, a testbench has been given to us. The register file testbench implementation was based of this testbench.



```
#       Total number of tests          27
#       Total number of pass           27
```

Fig. 11. Transcipt for MEM_64MB_TB

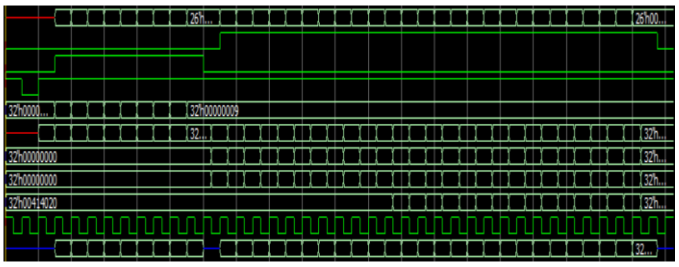


Fig. 12. Wave signals for MEM_64MB_TB

VI. Design and Implimentation of Control_Unit

CONTROL_UNIT will contain a state machine that is responsilbe for changing the state of the instruction process that was introduced earlier. The state machine will be implemented as a module PROC_SM. The state machine takes in two inputs CLK and RST. The state machine defines a state with the next state. The definition is as follows:

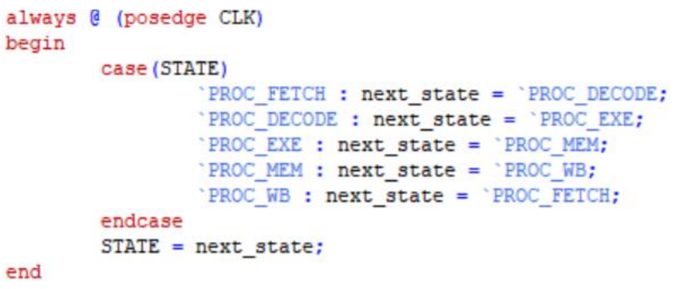
PROC_FETCH $\rightarrow$ *PRO_DECODE*

PROC_DECODE $\rightarrow$ *PROC_EXECUTE*

PROC_EXECUTE $\rightarrow$ *PROC_MEM*

PROC_MEM $\rightarrow$ *PRO_WB*

PROC_WB $\rightarrow$ *PRO_FETCH*



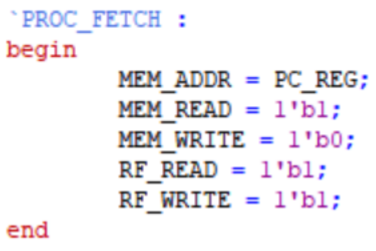
```
always @ (posedge CLK)
begin
        case(STATE)
                `PROC_FETCH : next_state = `PROC_DECODE;
                `PROC_DECODE : next_state = `PROC_EXE;
                `PROC_EXE : next_state = `PROC_MEM;
                `PROC_MEM : next_state = `PROC_WB;
                `PROC_WB : next_state = `PROC_FETCH;
        endcase
        STATE = next_state;
end
```

Fig. 13. Main Verilog code for PROC_SM

After the implementation of the the state machine, we will now implement the module CONTROL_UNIT. This module will take in five inputs ZERO, CLK, RST, RF_DATA_R1, RF_DATA_R2 and ALU_RESULT. It will give outputs RF_DATA_W, RF_ADDR_W, RF_ADDR_R1, RF_ADDR_R2, RF_READ, RF_WRITE, ALU_OP1, ALU_OP2, ALU_OPRN, MEM_ADDR, MEM_READ, MEM_WRITE. It will also take inout signal MEM_DATA. Based on the state of the state machine, the control unit will do specific operations in order to properly parse an instruction and either read or write data.

PROC_FETCH

The first state that every instruction goes through is the fetch stage. In this stage, the memory address for read is set to the program counter. Initially, the program counter is set to start at a location specified by ISA. Memory is then set to read, which means that the instruction to fetch is stored in memory data.

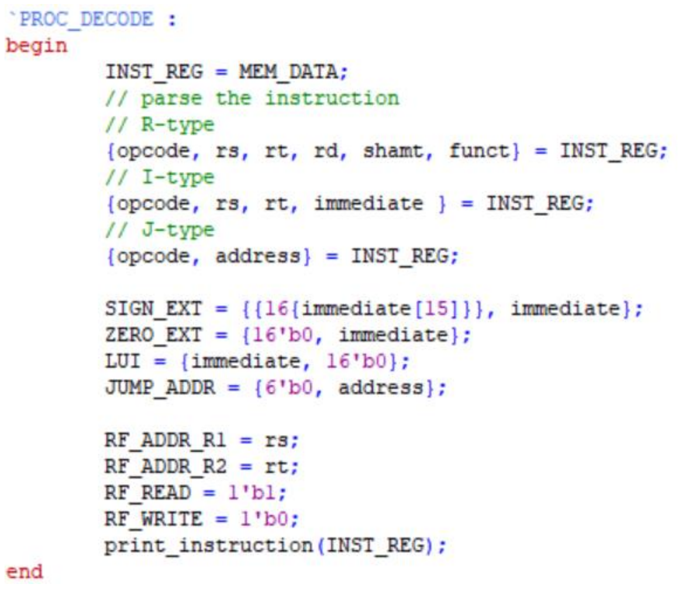


```
`PROC_FETCH :
begin
        MEM_ADDR = PC_REG;
        MEM_READ = 1'b1;
        MEM_WRITE = 1'b0;
        RF_READ = 1'b1;
        RF_WRITE = 1'b1;
end
```

Fig. 14. Verilog code for PROC_FETCH

PROC_DECODE

The second state that every instruction goes through is the decode stage. In this stage, the instruction is parsed into different registers that specify a region of the instruction. Extra registers are defined for specific instructions, for instance, I-Type instructions use a sign extension of the immediate, therefore a register SIGN_EXT = {{16{immediate[15]}}, immediate} is created. The register file will then read the contents of registers at address rs and rt. I also used a print_instruction task to print the parsed instructions.



```
`PROC_DECODE :
begin
        INST_REG = MEM_DATA;
        // 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;

        SIGN_EXT = {{16{immediate[15]}}, immediate};
        ZERO_EXT = {16'b0, immediate};
        LUI = {immediate, 16'b0};
        JUMP_ADDR = {6'b0, address};

        RF_ADDR_R1 = rs;
        RF_ADDR_R2 = rt;
        RF_READ = 1'b1;
        RF_WRITE = 1'b0;
        print_instruction(INST_REG);
end
```

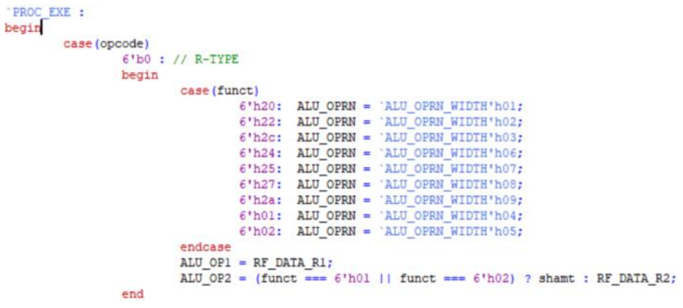
Fig. 15. Verilog code for PROC_DECODE

PROC_EXECUTE

In this stage, the parsed instructions will have the content that was read be placed into the ALU to compute a result. Not all instructions use the ALU such as lui, kmp, and jal. In this case, those instructions will not show up in the stage

R-Type Excute

All R-Type instructions in CS147DV use the ALU except for Jump Register. The rest of the R-Type instructions compute a result from R[rs] and R[rt], with the exception of the shift left and shift right instructions, of which compute a result from R[rs] and shamt

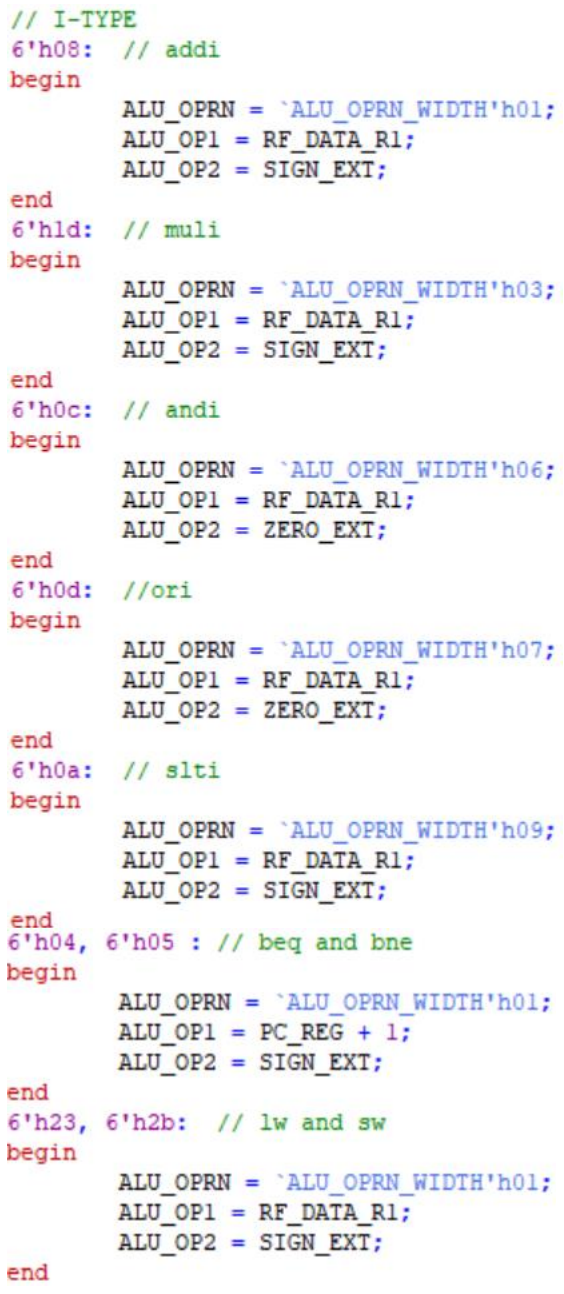


```
`PROC_EXE :
begin
        case(opcode)
                6'b0 : // R-TYPE
                begin
                        case(funct)
                                6'h20:  ALU_OPRN = `ALU_OPRN_WIDTH'h01;
                                6'h22:  ALU_OPRN = `ALU_OPRN_WIDTH'h02;
                                6'h2c:  ALU_OPRN = `ALU_OPRN_WIDTH'h03;
                                6'h24:  ALU_OPRN = `ALU_OPRN_WIDTH'h06;
                                6'h25:  ALU_OPRN = `ALU_OPRN_WIDTH'h07;
                                6'h27:  ALU_OPRN = `ALU_OPRN_WIDTH'h08;
                                6'h2a:  ALU_OPRN = `ALU_OPRN_WIDTH'h09;
                                6'h01:  ALU_OPRN = `ALU_OPRN_WIDTH'h04;
                                6'h02:  ALU_OPRN = `ALU_OPRN_WIDTH'h05;
                        endcase
                        ALU_OP1 = RF_DATA_R1;
                        ALU_OP2 = (funct === 6'h01 || funct === 6'h02) ? shamt : RF_DATA_R2;
                end
```

Fig. 16. Verilog code for PROC_EXECUTE for R-Type case

I-Type Excute

All I-Type instructions in CS147DV use the ALU except for Load upper immediate. Nearly all I-Type instructions use the content from R[rs] as the first operand, with the exception of Branch On Equal and Branch on Not Equal of which use PC + 1. Every I-Type instruction uses either Sign extended immediate or Zero extended immediate as the second operand.

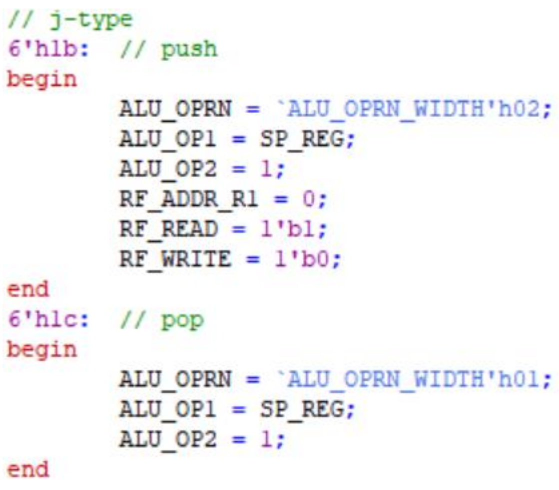


```
// I-TYPE
6'h08:  // addi
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h01;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = SIGN_EXT;
end
6'h1d:  // muli
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h03;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = SIGN_EXT;
end
6'h0c:  // andi
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h06;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = ZERO_EXT;
end
6'h0d:  //ori
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h07;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = ZERO_EXT;
end
6'h0a:  // slti
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h09;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = SIGN_EXT;
end
6'h04, 6'h05 : // beq and bne
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h01;
        ALU_OP1 = PC_REG + 1;
        ALU_OP2 = SIGN_EXT;
end
6'h23, 6'h2b:  // lw and sw
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h01;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = SIGN_EXT;
end
```

Fig. 17. Verilog code for PROC_EXECUTE for I-Type case

J-Type Excute

The only J-Type instructions that use the ALU are Push and Pop. Both instructions use the stack pointer register as the first operand and 1 and the second operand. Push computes and subtraction whereas pop computes an addition. Additionally, the Push instruction needs to also read the content of the first register.

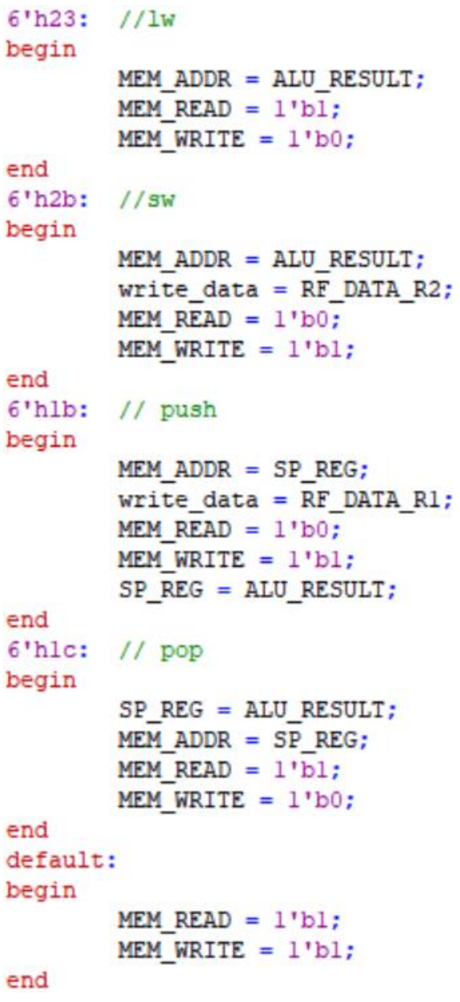


```
// j-type
6'h1b:  // push
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h02;
        ALU_OP1 = SP_REG;
        ALU_OP2 = 1;
        RF_ADDR_R1 = 0;
        RF_READ = 1'b1;
        RF_WRITE = 1'b0;
end
6'h1c:  // pop
begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h01;
        ALU_OP1 = SP_REG;
        ALU_OP2 = 1;
end
```

Fig. 18. Verilog code for PROC_EXECUTE for J-Type case

PROC_MEM

In this stage, memory will be accessed for specific instructions. For this instruction set only Load word, Store word, Push, and Pop will need to access memory. For Load word, the result from the exceute stage will be used as the address to read from in memory. For Store word, the result from the execute stage will be used as the address in memory to write. The data to be written will be the content from R[rt]. For Push, the contents that were read earlier will be written into the stack pointer register, the stack pointer register will then be decremented. For Pop, the stack pointer register is incremented. The address to be read is the stack pointer.



```
6'h23:  //lw
begin
        MEM_ADDR = ALU_RESULT;
        MEM_READ = 1'b1;
        MEM_WRITE = 1'b0;
end
6'h2b:  //sw
begin
        MEM_ADDR = ALU_RESULT;
        write_data = RF_DATA_R2;
        MEM_READ = 1'b0;
        MEM_WRITE = 1'b1;
end
6'h1b:  // push
begin
        MEM_ADDR = SP_REG;
        write_data = RF_DATA_R1;
        MEM_READ = 1'b0;
        MEM_WRITE = 1'b1;
        SP_REG = ALU_RESULT;
end
6'h1c:  // pop
begin
        SP_REG = ALU_RESULT;
        MEM_ADDR = SP_REG;
        MEM_READ = 1'b1;
        MEM_WRITE = 1'b0;
end
default:
begin
        MEM_READ = 1'b1;
        MEM_WRITE = 1'b1;
end
```

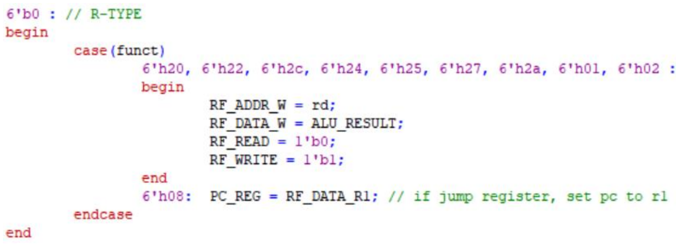
Fig. 19. Verilog code for PROC_MEM

PROC_WB

In this stage, all results have been computed, the only thing to worry about now is to assign the result back to the corresponding registers. This stage is also resposible for updating the program counter

R-Type Write Back

For the R-Type instructions, the Jump register instruction will set the program counter to the content in R[rs]. The rest of the R-Type instructions will write the result from the exceute stage into the address rd.

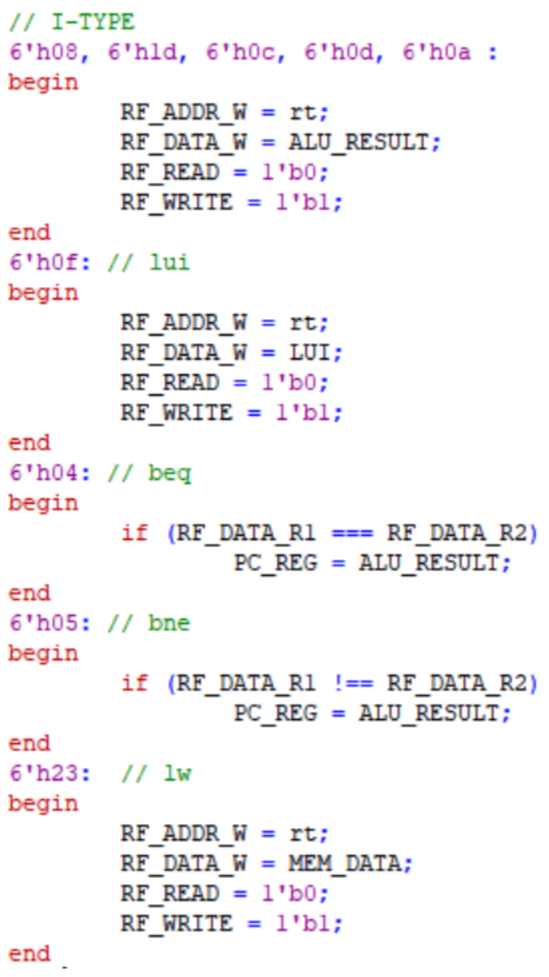


```
6'b0 : // R-TYPE
begin
        case(funct)
                6'h20, 6'h22, 6'h2c, 6'h24, 6'h25, 6'h27, 6'h2a, 6'h01, 6'h02 :
                begin
                        RF_ADDR_W = rd;
                        RF_DATA_W = ALU_RESULT;
                        RF_READ = 1'b0;
                        RF_WRITE = 1'b1;
                end
                6'h08:  PC_REG = RF_DATA_R1; // if jump register, set pc to r1
        endcase
end
```

Fig. 20. Verilog code for PROC_WB for R-Type case

I-Type Write Back

For the I-Type instructions, Addition, Multiplication, AND, OR, and Set less than will write the results from ALU into register at address rt. Load upper immediate will do the same but with LUI instead of ALU result. For Branch on equal and Branch on not equal, the program counter will be set the ALU result if the condition is true. For Load word, the content read from memory will be stored in the register at address rt. Store word has already been finished in the memory stage so it is not needed in write back.

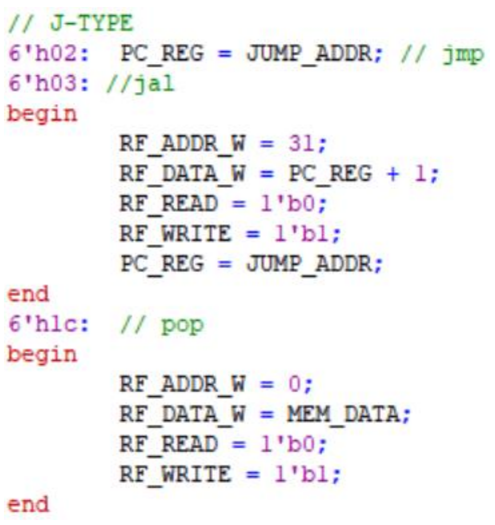


```
// I-TYPE
6'h08, 6'h1d, 6'h0c, 6'h0d, 6'h0a :
begin
        RF_ADDR_W = rt;
        RF_DATA_W = ALU_RESULT;
        RF_READ = 1'b0;
        RF_WRITE = 1'b1;
end
6'h0f: // lui
begin
        RF_ADDR_W = rt;
        RF_DATA_W = LUI;
        RF_READ = 1'b0;
        RF_WRITE = 1'b1;
end
6'h04: // beq
begin
        if (RF_DATA_R1 === RF_DATA_R2)
                PC_REG = ALU_RESULT;
end
6'h05: // bne
begin
        if (RF_DATA_R1 !== RF_DATA_R2)
                PC_REG = ALU_RESULT;
end
6'h23:  // lw
begin
        RF_ADDR_W = rt;
        RF_DATA_W = MEM_DATA;
        RF_READ = 1'b0;
        RF_WRITE = 1'b1;
end
```

Fig. 21. Verilog code for PROC_WB for I-Type case

J-Type Write Back

For the J-Type instructions, Jump to address will simple set program counter to jump address. For Jump and Link R[31] will be set to PC + 1 and PC will also be set to jump address. Push has already been done at memory stage, so it does not need to write back. For Pop, R[0] is set to the memory read from the memory stage.

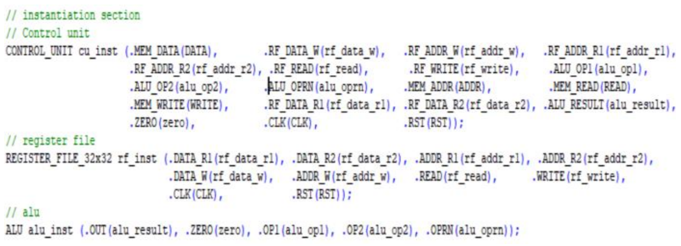


```
// J-TYPE
6'h02:  PC_REG = JUMP_ADDR; // jmp
6'h03: //jal
begin
        RF_ADDR_W = 31;
        RF_DATA_W = PC_REG + 1;
        RF_READ = 1'b0;
        RF_WRITE = 1'b1;
        PC_REG = JUMP_ADDR;
end
6'h1c:  // pop
begin
        RF_ADDR_W = 0;
        RF_DATA_W = MEM_DATA;
        RF_READ = 1'b0;
        RF_WRITE = 1'b1;
end
```

Fig. 22. Verilog code for PROC_WB for J-Type case

VII. Design and Implimentation of Processor

The processor for this project will be implemented as a module PROC_CS147_SEC05. This processor takes in two inputs CLK and RST. The processor gives out three outputs ADDR, READ, and WRITE. The processor has an inout port DATA. It also a collection of wire to connect into the instantiated CONTROL_UNIT, REGISTER_FILE_32X32, and ALU.

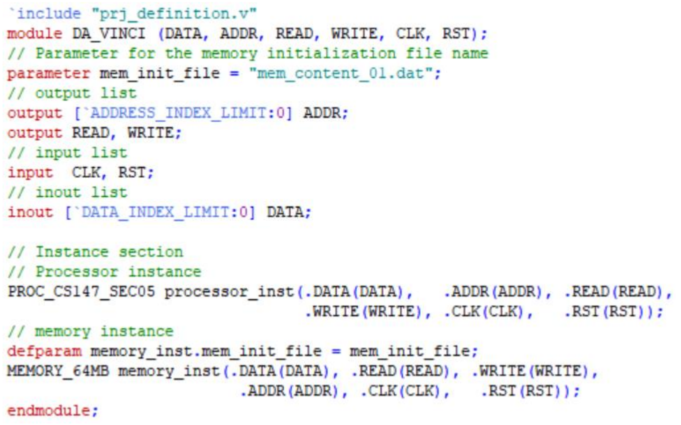


```
// instantiation section
// Control unit
CONTROL_UNIT cu_inst (.MEM_DATA(DATA),        .RF_DATA_W(rf_data_w),   .RF_ADDR_W(rf_addr_w),   .RF_ADDR_R1(rf_addr_r1),
                      .RF_ADDR_R2(rf_addr_r2), .RF_READ(rf_read),       .RF_WRITE(rf_write),     .ALU_OP1(alu_op1),
                      .ALU_OP2(alu_op2),       .ALU_OPRN(alu_oprn),     .MEM_ADDR(ADDR),         .MEM_READ(READ),
                      .MEM_WRITE(WRITE),       .RF_DATA_R1(rf_data_r1), .RF_DATA_R2(rf_data_r2), .ALU_RESULT(alu_result),
                      .ZERO(zero),             .CLK(CLK),               .RST(RST));
// register file
REGISTER_FILE_32x32 rf_inst (.DATA_R1(rf_data_r1), .DATA_R2(rf_data_r2), .ADDR_R1(rf_addr_r1), .ADDR_R2(rf_addr_r2),
                             .DATA_W(rf_data_w),   .ADDR_W(rf_addr_w),   .READ(rf_read),       .WRITE(rf_write),
                             .CLK(CLK),            .RST(RST));
// alu
ALU alu_inst (.OUT(alu_result), .ZERO(zero), .OP1(alu_op1), .OP2(alu_op2), .OPRN(alu_oprn));
```

Fig. 23. The instantiation of control unit, register file, and alu in the processor

VIII. Design and Implimentation of Da_Vinci

The last component of this computer system is the computer system itself. In order to process instructions, DaVinci creates the instances for Memory and Processor. The system takes inputs CLK and RST. It gives out outputs ADDR, READ, and WRITE. Additionally, it takes an inout port for DATA.



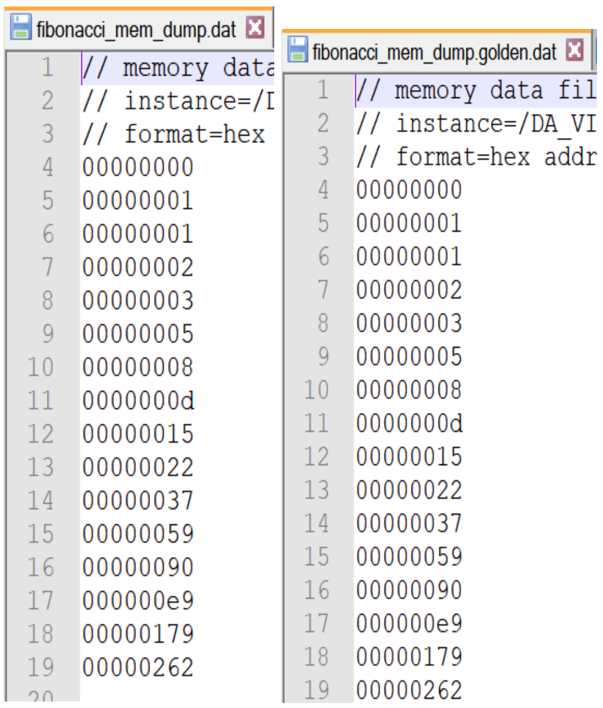
```
`include "prj_definition.v"
module DA_VINCI (DATA, ADDR, READ, WRITE, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output READ, WRITE;
// input list
input  CLK, RST;
// inout list
inout [`DATA_INDEX_LIMIT:0] DATA;

// Instance section
// Processor instance
PROC_CS147_SEC05 processor_inst(.DATA(DATA),   .ADDR(ADDR), .READ(READ),
                                .WRITE(WRITE), .CLK(CLK),   .RST(RST));
// memory instance
defparam memory_inst.mem_init_file = mem_init_file;
MEMORY_64MB memory_inst(.DATA(DATA), .READ(READ), .WRITE(WRITE),
                        .ADDR(ADDR), .CLK(CLK),   .RST(RST));
endmodule;
```

Fig. 24. The Verilog code for module DA_VINCI

IX. TEST STRATEGY AND TEST IMPLEMENTATION

Now that we have completed our computer system, it is time to test it. In the Da_Vinci testbench, we can comment out the test that we do not want. First let us run the testbench using the fibonacci test. The testbench will read from the fibonacci DAT file, which contains the machine code that uses the instructions from CS147DV. This machine code will produce the first few fibonacci numbers. The fibonacci golden dump file shows the expected output, whereas the regular dump file shows the output that was computed from our computer system. After running the testbench, we can compare the dump files to see if they match.



fibonacci_mem_dump.dat

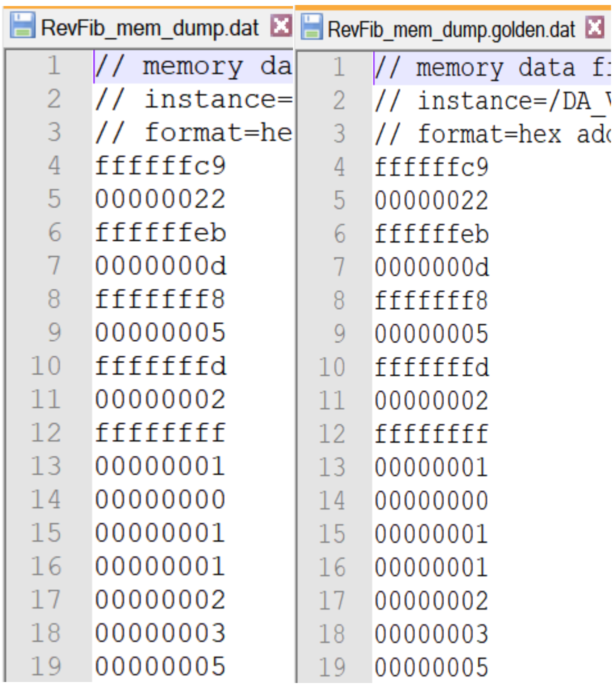
```
// memory data
// instance=/I
// format=hex
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

fibonacci_mem_dump.golden.dat

```
// memory data fil
// instance=/DA_VI
// format=hex addr
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

Fig. 25. The matching resulting and expected fibonacci dump files

To further test our computer system, we can also do the same thing but with the reverse fibonacci file.



RevFib_mem_dump.dat

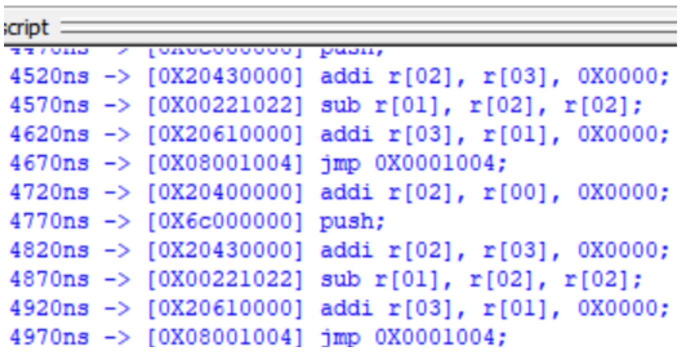
```
// memory da
// instance=
// format=he
ffffffc9
00000022
ffffffeb
0000000d
fffffff8
00000005
fffffffd
00000002
ffffffff
00000001
00000000
00000001
00000001
00000002
00000003
00000005
```

RevFib_mem_dump.golden.dat

```
// memory data f
// instance=/DA_
// format=hex ad
ffffffc9
00000022
ffffffeb
0000000d
fffffff8
00000005
fffffffd
00000002
ffffffff
00000001
00000000
00000001
00000001
00000002
00000003
00000005
```

Fig. 26. The matching resulting and expected reverse fibonacci dump files

When running the testbench, the parsed instructions will also be printed on the transcipt since we used the print instruction in the decode stage.



script

```
4520ns -> [0X20430000] addi r[02], r[03], 0X0000;
4570ns -> [0X00221022] sub r[01], r[02], r[02];
4620ns -> [0X20610000] addi r[03], r[01], 0X0000;
4670ns -> [0X08001004] jmp 0X0001004;
4720ns -> [0X20400000] addi r[02], r[00], 0X0000;
4770ns -> [0X6c000000] push;
4820ns -> [0X20430000] addi r[02], r[03], 0X0000;
4870ns -> [0X00221022] sub r[01], r[02], r[02];
4920ns -> [0X20610000] addi r[03], r[01], 0X0000;
4970ns -> [0X08001004] jmp 0X0001004;
```

Fig. 27. The transcript of the DA_VINCI_TB

X. CONCULSION

This project has taught me about the architechure of a computer system during the process of reading instructions. Now I understand that the computer system takes a lot of back and fouth procedures between the register file and memory in order to just process one information. The computer system that I have created is not perfect. I have only used the tests given and have not been able to try other tests. There are still bugs that could possibly be in the computer system. Further testing needs to be done in order to ensure that the computer system is perfect.