

DaVinci v1.0 Processor Simulation

Azael Zamora

San Jose State University

azael.zamora12@gmail.com

Abstract: The objective of this report is to focus on the steps and requirements needed to create a DaVinci v1.0 system processor simulation. The system simulation requires building a 32x64M Memory module, the 'CS147DV' Instruction set, a Register File module, and a modified Arithmetic Logic Unit (ALU) and Control Unit (CU) for the purpose of the simulation. The following sections will describe the individual parts, and also how they come together as whole to run the simulation.

I. INTRODUCTION

The DaVinci v1.0 system processor is comprised of a modified ALU, a memory module, modified Control Unit, and Register File modules. The objective of the system processor is to simulate a processor with memory using the Verilog Hardware Language. The project will help show how the individual components work in the processing system. The processor will then tested on a DaVinci test bench file in Verilog.

II. REQUIREMENTS FOR THE SYSTEM

This part of the paper will cover the system requirements for the processor as well as the memory that is needed in order to be able to run the DaVinci v1.0 system processor for the simulation.

A. The ALU (Arithmetic Logic Unit)

The ALU is a crucial component that is utilized by the processing system. For this project, this component will handle the logic and arithmetic work by being able to complete the nine operations in the 'CS147DV' instruction set that was introduced in Lecture 01, as well as implementing a Zero check flag for the processing system.

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Logical AND (&)
5. Logical NOR (~|)
6. Logical OR (|)
7. Set Less Than (<)
8. Shift Logical Left (<<)
9. Shift Logical Right (>>)
10. ZERO System check

B. Register File for the processor

The register file is comprised of 32 registers from register 0 to register 31, and each register can store a 32-bit word (word addressable). For the purpose of this project, read/write operations will only function in the positive edge of the clock. The reset

function will be implemented in the negative edge of the clock. For the simulation, both read and write will be in HIGH-Z when either 11 or 00 is present, and both will only execute when 01 or 10 is present in the clock.

C. Memory for the processor

The memory module that is provided for this project is a 32x64M memory model that is able to store a 32-bit word in each register that is also known as an address.

D. The Control Unit (CU)

A five-stage state machine will be used in this simulation to decide the corresponding function of the Control Unit. The machine state will change at the positive edge in the clock and will rotate between Fetch, Decode, Execute, Memory, and Write Back for the processor. The control unit will use the instruction set 'CS147DV' to be able to execute its necessary functions.

E. 'CS147DV' Instruction Set

The instruction set 'CS147DV' will be utilized by the processor, which is provided by Professor Patra, a Computer Science professor from San Jose State University. The instruction set provides three different types of instructions which are: R-Type, I-Type, and J-Type. These instructions are necessary for the Control Unit, since it will utilize them to properly implement the simulated process system.

III. DESIGNING AND IMPLEMENTING THE ALU

The implementation of the ALU is relatively easy to program the necessary functions using ModelSim. The following sections will focus on the overall design and implementation of the ALU using Verilog Hardware Language.

A. The ALU Design

The ALU design is very simple, since it's implemented to take in two operators, and an operation and return the result as the output. For this project, the ALU that is built is a 32-bit processor, and each operand is 32-bit, the operand is 6-bit, and the output is 32-bit. All the nine operations that are utilized in this program are defined with 'ALU_WIDTH_OPRN'h0X where the X is used to define the specific operation. In the project, 'h03' is multiplication as an example, while the operands are called 'op1' and 'op2' respectively, and

the output is called ‘golden’ which is the output in this case.

B. The Operations that the ALU handles

For the purpose of this project, nine logical and arithmetic operations will be handled by the ALU, including system check ZERO. As shown in the picture below, once the OPRN has the selected and performed the necessary action, the output labeled ‘OUT’ is utilized to determine the ZERO system check. The image below will show how each of the nine operations is implemented for this project.

```
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 Logical Right
        'ALU_OPRN_WIDTH'h01 : OUT = OP1 << OP2; // Shift Logical 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 That
        default: OUT = 'DATA_WIDTH'hxxxxxxx; //Defaults to the OPRN
    endcase
end

always @(OUT)
begin
```

IMPLEMENTATION OF THE ALU

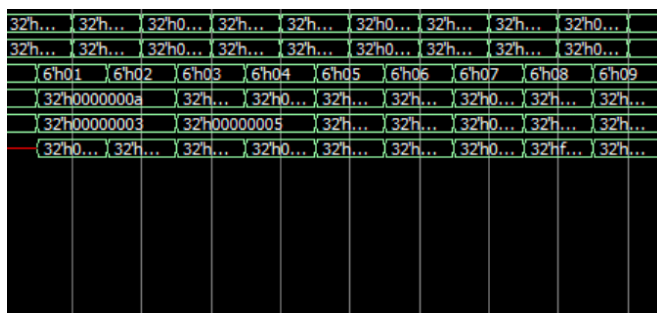
C. Testing the implemented ALU

Utilizing the file ‘alu_tb.v’, we can check if the ALU that has been implemented for this project to determine if the ALU is properly implemented. Each operation that was tested in the file can be presented either as text output, or can be displayed by wavelengths as shown in the following images.

```
[TEST] 15 + 3 = 18 , got 18 ... [PASSED]
[TEST] 15 - 5 = 10 , got 10 ... [PASSED]
[TEST] 5 * 5 = 25 , got 25 ... [PASSED]
[TEST] 10 >> 2 = 2 , got 2 ... [PASSED]
[TEST] 10 << 2 = 40 , got 40 ... [PASSED]
[TEST] 1 & 0 = 0 , got 0 ... [PASSED]
[TEST] 1 | 0 = 1 , got 1 ... [PASSED]
[TEST] 21474836 ~ 0 = 4273492459 , got 4273492459 ... [PASSED]
[TEST] 5 < 10 = 1 , got 1 ... [PASSED]
```

```
Total number of tests      9
Total number of pass      9
```

Text output of ALU_tb



The entire Waveform of the ALU Simulation

IV. IMPLEMENTING THE REGISTER FILE

Implementing the Register File, allows it to determine whether a Read or Write operation is needed, and will also set DATA_R1 and DATA_R2 to X if both read and write operations are 00 or 11. DATA_R1 and DATA_R2 are the two data ports that are needed for the register file, and both ports will contain information from the ADDR_R1 and ADDR_R2 inputs. The input ADDR_W is used to allow DATA_W port to be written to a register.

```
//need to make register do nothing on 00 or 11
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) };
```

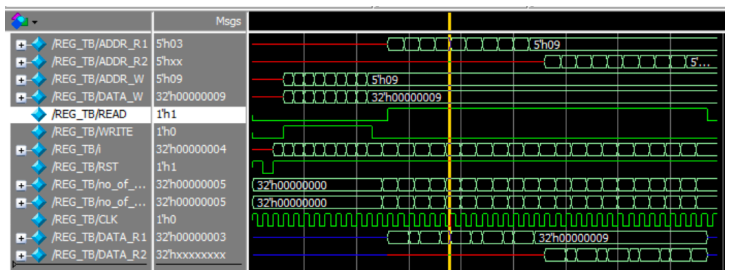
Operations for the Register File

The Reset State will occur when there is a negative edge on the RST, while the other operations and states will occur in the positive edge of CLK. When the inputs READ = 1'b1 and WRITE = 1'b0 the register will read out, and if READ = 1'b0 and WRITE = 1'b1, then the register will read in.

```
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 operation
        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 operation
            reg_32x32[ADDR_W] = DATA_W;
    end
end
```

Conditional check for READ, WRITE operations

In order to make sure that the register file module was implemented properly, a test bench must be created in order to make sure that both read and write operations function properly, and the outputs are stores in their necessary addresses. In the image below, by looking at the wavelengths, you can determine how both read and write operations are functioning based on the system’s clock.



#		
#	Total number of tests	21
#	Total number of pass	21
#		

Both Text and Wavelength output for Register File

V. IMPLEMENTING THE MEMORY

Like the Register File module, memory module has both input and output data port that is named 'DATA'. In the module, READ operation is set to 1 which enables the reading from the memory. The WRITE operation is set to 0, while ADDR, and data address are given for this project, and are used by the DATA inout port as the output. When the READ operation is 0, and the WRITE operation is 1, the ADDR is instead used as the output for the DATA inout. If both READ and WRITE operations are both set to 0 and 1, then the DATA data port will remain in the High-Z state.

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

Implementation of Read/Write for Memory

Furthermore for the implementation of the memory module, the Reset stage will only occur when a negative edge of RST is present, and then the RST will be set to '1'b0'. Once the Reset Stage occurs, all the memory data will then change to 0, and the initialization memory file called 'mem_init_file' will be read instead. The rest of the stages from the processor memory will only happen when Clock has its positive edge present.

```
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
end
```

The implementation of Reset state for Memory

VI. DESIGNING, IMPLEMENTING, AND THE TESTING OF CONTROL UNIT

1. The State Machine Control

For the simulated processor, the Control Unit is implemented so that it can switch to the other different states in the processor. In order to accomplish this, a state machine will need to be utilized for this purpose. The five different states for the processor will be Fetch, Decode, Execute, Memory, and Write Back, as the Control Unit will be rotating between them. As a clock cycle passes, the state operator will switch to the next state. If the state operator is initialized to 'reset', the current state register that is called 'state_reg', is set

to 3'bxx, while 'next_state' register is set to the next corresponding clock state.

```
//state switching
always@(posedge CLK)
begin
  case (STATE) // implements the 5 clock states
    `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
```

Implementation of the state rotation for the control unit

Once the case for STATE is executed, the control unit will proceed to switch states in the positive edge of the Clock cycle. As shown in the image below, the 'next_state' will be set to proper state that is needed, and will proceed to set other proper states, as the machine continues to operate.

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

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

Initialization and reset for the Control Unit

2. Testing the implementation of the State Machine

The state machine can be tested by watching the waveforms. If the waveform is at a positive clock edge, one can check if the machine was implemented properly if it changes to the next corresponding state. The reset state will also need to be taken into account, when a negative clock edge occurs.

3. The Design of the R-Type Instruction

The R-type instructions are relatively easy instructions to implement, since R-types utilize a opcode of value h'00. While the Control Unit receives the opcode, the CU will then proceed to implement the correct ALU function using the two data ports which will be set to OP1 and OP2. The Control Unit will only call the functions of the ALU to be executed, it will focus on whether or not the function works properly.

```

if(proc_state === `PROC_EXE)
begin
case (opcode)
// R-Type Instructions, share same opcode
6'h00 :
begin
if(funcnt === 6'h08)
begin
PC_REG = RF_DATA_R1;
end
//line 149 handles logical right/left shift
else if(funcnt === 6'h01 || funcnt === 6'h02)
begin
ALU_OPRN_RET = funcnt;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = shamt;
end
else
begin
ALU_OPRN_RET = funcnt;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = RF_DATA_R2;
end
end
end
end

```

Implementation of R-Type Instructions

During the Write Back stage, R-Type instructions will be called again in order to write in the needed address.

```

//R-Type write back
6'h00 : //opcode for R-Type instructions
begin
if(funcnt === 6'h08)
PC_REG = RF_DATA_R1;
else
begin
RF_ADDR_W_RET = rd;
// result from the ALU
RF_DATA_W_RET = ALU_RESULT;
RF_WRITE_RET = 1'b1;
end
end
end

```

R-Type write back implementation.

4. The Implementation of the I-Type instruction

Unlike R-Type instructions, which only use the opcode h'00, I-Type instructions that are implemented use variety of opcodes. In order to determine which instruction would be called, cases on opcode were utilized to determine which I-Type instruction is used. As shown in the image below, the following instructions are utilized for the Execution Procedure by the Control Unit.

```

//I-Type
6'h08 : //addi
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end

6'h1d : //mulh
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h2c;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end

6'h0c : //andi
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h24;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = ZERO_EXTENDED;
end
end

```

```

6'h0d : //ori
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h25;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = ZERO_EXTENDED;
end

6'h0a : //slli
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h2a;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end

6'h23 : //load word
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end

6'h2b : //store word
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
end

```

Implementation of I-Type Instructions

Using the instruction set 'CS147DV' for I-Type instructions, some of the instructions use either ZeroExtended, SignExtended, LUI, or a BranchAddress, as their immediate portion of their functions. The image below shows how each of the immediate were implemented.

```

//Immediate sign extension
SIGN_EXTENDED = {{16{immediate[15]}},immediate};

//Immediate zero extension
ZERO_EXTENDED = {16'h0000, immediate};
//LUI value
LUI = {immediate, 16'h0000};

```

Implementation of immediate types

Instructions store word (sw), load word (lw) are utilized during the Memory state of the Control Unit, since these instructions control reading and writing to the memory. Below is an image that shows how store word, and load word were implemented in the program.

```

6'h23 : //load word
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end

6'h2b : //store word
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h20;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
end

```

LW and SW being implemented for Memory

During the Write Back state, cases corresponding to the opcode will be executed.


```

//I-Type
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
end

```

The I-Type write back implementation

5. The Design of the J-Type Instruction

The push operation, one of the four J-Type operations, is only executed in the Execution procedure state of the Control Unit that is designed for this project. The other three J-Type instructions will be executed in the Write Back state unlike the push operation. For this project, J-Type design was implemented through use of running cases on the opcode, as shown in the images below.

```

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

6'h1b : //Push
begin
  MEM_ADDR_RET = SP_REF;
  MEM_DATA_RET = RF_DATA_R1;
  MEM_WRITE_RET = 1'b1;
  SP_REF = SP_REF - 1;
end

6'h1c : //pop
begin
  SP_REF = SP_REF + 1;
  MEM_ADDR_RET = SP_REF;
  MEM_READ_RET = 1'b1;
end

//Store 32-bit jumpaddress
JUMP_ADDRESS = {6'b0, address};

```

J-Type Instruction Implementation

Given that three out of the four J-Type instructions are only implemented in the write back state, it is indicated that the instructions will write to the registers used in the program or the PC_REG is changed as a result.

```

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

6'h03 : //jal
begin
  RF_ADDR_W_RET = 31;
  RF_DATA_W_RET = PC_REG;
  RF_WRITE_RET = 1'b1;
  PC_REG = JUMP_ADDRESS;
end

6'h1c : //pop
begin
  RF_ADDR_W_RET = 0;
  RF_DATA_W_RET = MEM_DATA;
  RF_WRITE_RET = 1'b1;
end

```

Write Back for J_Type Instructions

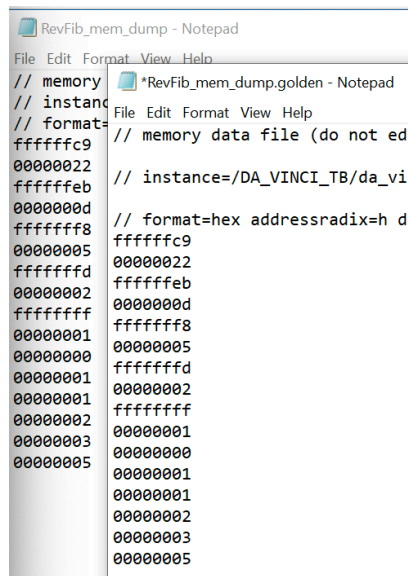
VII. TESTING THE SIMULATED PROCESSOR

To check if our DaVinci v1.0 system processor was correctly implemented, 'DaVinci_tb.v' must be used in the simulator. Utilizing both files 'RevFib.data' and 'Fibonacci.data', one can check both read and write implementations of the Verilog program, while also able to view the wavelengths of where the simulated processor is running.



Wavelength of the DaVinci_TB

Aside from only viewing the wavelengths to see if the processor was correctly implemented, both 'RevFib' data dump, and data.golden files can be used to compare the result to determine whether the processor functioned properly.



```

// memory
// instance
// format=
ffffc9 // memory data file (do not ed
0000022 // instance=/DA_VINCI_TB/da_vi
fffffeb
000000d // format=hex addressradix=h d
fffff8 fffffc9
0000005 0000022
fffffd fffffeb
0000002 000000d
fffff ffffff8
0000001 0000005
0000000 fffffd
0000001 0000002
0000001 ffffff
0000002 0000001
0000003 0000000
0000005 0000001
0000001
0000001
0000002
0000003
0000005

```

Comparison of both files

VIII.

CONCLUSION

To summarize, the DaVinci v.10 system processor is able to offer a behavioral model for simulating a processor that is implemented by the 'CS147DV' instruction set. By completing this project, one is able to see how the system processor is able to show the connections between the ALU, Control Unit, Processor, Memory and the Register File. In short, by completing the project, we were able to build the crucial parts of the processor, as well as learned that the Verilog language can be utilized to create a functioning system processor.