32-Bit Processor Simulation - Behavioral Model

Craig Huff
CS 147 Section 02
San Jose State University
craig.huff@sjsu.edu

***Abstract*— This paper covers the creation of the DaVinci v1.0 system processor simulation. The simulation is built using the 'CS147DV' Instruction Set, a 32x64m Memory module, a Register File, and a custom Arithmetic Logic Unit (ALU) and Control Unit (CU). The sections below will describe the parts independently and then how they act together as a whole to run the DaVinci v1.0 system.**

I. Introduction

The goal of the DaVinci v1.0 system is to create a simple representation of a processor and memory in the Verilog language. The processor is made up of an ALU, Register File, Control Unit, and memory models. The project demonstrates how all the components work together in a processor. Testing is then done on the DaVinci system.

II. System Requirements

This section will outline the system requirements for the processor and the memory in order to run the DaVinci v1.0 system.

A. Arithmetic Logic Unit (ALU)

The ALU is the major component of the DaVinci v1.0 system. The ALU handles the logic work by handling the nine operations in the 'CS147DV' Instruction Set and a Zero flag for the system

- Addition
- Subtraction
- Multiplication
- Shift Logical Right
- Shift Logical Left
- Logical AND
- Logical OR
- Logical NOR
- Set Less Than
- ZERO System Flag

B. Register File

The Register file is made of 32 registers [0 to 31] that are able to store a 32-bit word. The Reset operation is executed on the negative edge of the clock, while the read and write operations are executed on the positive edge. The READ and WRITE operations will remain in HIGH-Z when 11or 00 is present, and will execute only when 10 or 01 is present.

C. Memory

The Memory Module is 64M of memory that stores a 32-bit word at each address. Reset operations are executed on the negative edge of the clock, and all other operations are executed on the positive edge of the clock. DATA_R# is set to 'don't care' or X when READ or WRITE are set to 00 or 11.

D. Control Unit (CU)

The function of the Control Unit is decided by a five-step state machine. The state changes at the positive edge of the clock and cycles between FETCH, DECODE, EXECUTE, MEMORY, AND WRITE BACK. The control unit executes its decisions according to the Instruction Set, 'CS147DV'.

E. CS147DV Instruction Set

This instruction set is provided by Professor Patra of San Jose State University and provides instructions for R-Type, I-Type, and J-Type instructions. The instructions are used primarily by the Control Unit to correctly implement the DaVinci v1.0 System

III. ALU Design and Implementation

Using the ModelSim simulator, it is simple to program the basic functions of the ALU. This section will outline the design of the ALU using the Verilog Hardware Description Language (VHDL).

A. ALU Design

The design of the ALU is simple. It's designed to take two operands and one operation as the input and return the result of the process as the output. The ALU built in this project is a 32-bit processor, which makes each of the operands 32-bit, the operand 6-bit, and the output as 32-bit. Each operation implemented in the program is defined with the name 'ALU_OPRN_WIDTH'h0(#) where the number sign(#) defines the specific operation. For example, 'h22 is Subtraction. The operands have the names 'OP1' and 'OP2'. The resulting output is referred to as 'OUT'. The 'ZERO' flag will check the output.

B. ALU Operations

There are nine operations that are handled by the ALU plus the ZERO system flag. The OPRN is used to select the operation to perform and after the operation, the output, 'OUT' is checked to determine the ZERO flag.

```
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'hxxxxxxxx; //Defaults to the OPRN

endcase
end

always @(OUT)
begin
    ZERO = OUT == 0 ? 1 : 0;
end
```

Figure 3.1. ALU Operations

C. *ALU Testing*

The ALU was tested using the 'alu_tb.v' file. This test determines if the ALU is functioning properly. The operations test displayed by wavelengths in Figure 5.11 can also be represented in text as is shown in figure 5.10.

```
VSIM 14> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 6 - 2 = 4 , got 4 ... [PASSED]
# [TEST] 10 * 12 = 120 , got 120 ... [PASSED]
# [TEST] 26 || 3 = 27 , got 27 ... [PASSED]
# [TEST] 6 && 18 = 2 , got 2 ... [PASSED]
# [TEST] 7 ~| 1 = 4294967288 , got 4294967288 ... [PASSED]
# [TEST] 4 < 13 = 1 , got 1 ... [PASSED]
# [TEST] 9 << 2 = 36 , got 36 ... [PASSED]
# [TEST] 12 >> 2 = 3 , got 3 ... [PASSED]
#
#       Total number of tests          9
#       Total number of pass           9
#
```

Figure 3.2. ALU Test Pass

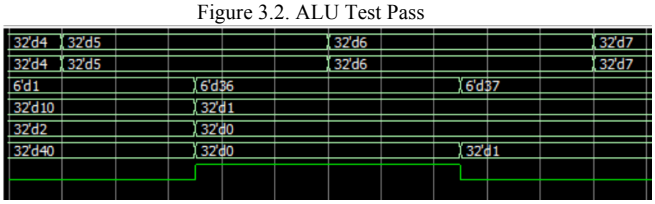

Figure 3.3. ALU Simulation Wavelength

IV. Register File Design and Implementation

The Register File determines whether to read or write using the READ and Write inputs and will set the DATA_R# to X if read and write are 00 or 11. The Register is made of two data ports, Data_R1 and Data_R2, which are given information from the register locations contained in ADDR_R1 and ADD_R2 inputs. The data input port DATA_W can be written to a register using the ADDR_W input.

```
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}
};
```

Figure 4.1. Register File Operations

The Reset is set to occur on the negative edge of the RST, and all other operations will occur on the positive edge of the CLK. The register will read out when READ = 1'b1 and WRITE = 1'b0 and it will read in when READ = 1'b0 and WRITE = 1'b1.

```
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 op
 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 op
      reg_32x32[ADDR_W] = DATA_W;
```

Figure 4.2. Register File Operations

To test the Register File, we must make sure that the read and write functions work correctly and are stored in the right register addresses. Using the wavelengths, in the figure below it can be seen how the READ and WRITE selection is working according to the system's Clock. The test case used for the register file will test the WRITE function by DATA_W 0 through 9 to corresponding register addresses ADDR_W. After, READ and WRITE will begin both set to 0 while to registers are outputted, upon which the data is then read from the registers DATA_R1 and DATA_R2.

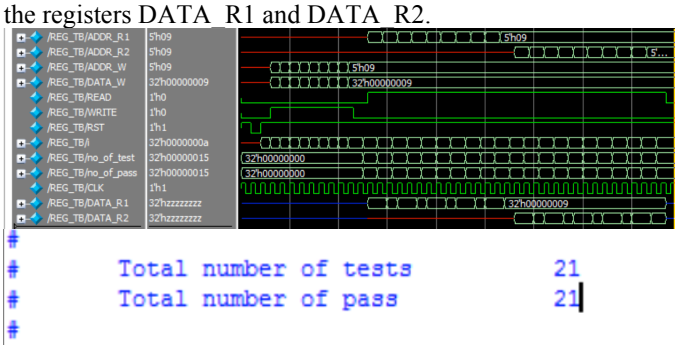

Figure 4.3. Register File Simulation Wavelengths and Pass

V. Memory Design and Implementation

The Memory is similar to the Register File, but it only has one input and output data port called 'DATA'. To read from the memory, READ is set to 1, WRITE is set to 0, and the data address, ADDR, is give and outputted to the DATA inout port. When the opposite is true and READ is 0 and WRITE is 1, ADDR is inputted into the DATA inout. The DATA will remain in a High-Z state if READ and WRITE are both set to 0 or 1.

```
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
```

Figure 5.1. Memory Read \ Write Operations

The Reset is set to occur when there is a negative edge of the RST and the RST is set to '1b'0'. When the reset occurs, all the memory data is changed to 0 and the memory initialization file, 'mem_init_file' is read. All other processes by the memory occur on the positive edge of the Clock.

```
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);
```

Figure 5.2. Memory Reset / Initialization

VI. Control Unit (State Machine) Design, Implementation, and Testing

A. State Machine Control

The Control Unit is designed to efficiently switch between the states of the processor. To do this, we will use a state machine. The States will be FETCH, DECODE, EXECUTE, MEMORY, and WRITE BACK. Each clock cycle, the process will go to the next one. When the state machine is initialized and when it's reset, the current state register, 'state_reg', is set to 3'bxx and the 'next_state' register is set to next state.

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

case(STATE)
  `PROC_FETCH : next_state = `PROC_DECODE;
  `PROC_DECODE : next_state = `PROC_EXE;
  `PROC_EXE : next_state = `PROC_MEM;
  `PROC_MEM : next_state = `PROC_WB;
  `PROC_WB : next_state = `PROC_FETCH;
endcase
state_reg = next_state;
```

Figure 6.1. Next State Machine

After this, the machine is designed to switch states at the positive edge of the Clock cycle. This process is done by setting the 'next_state' to be the appropriate state to follow. This will continue as long as the state machine is active.

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

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

Figure 6.2. State Machine Initialization / Reset

B. State Machine Testing

Testing the State machine is a simple process. By watching the waveforms change, we can verify that at each positive clock edge, the state machine will change to the next state. We also need to look at the reset state, to ensure it occurs at the negative clock edge.

```
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;
```

C. R-Type Instruction Design

R-type instructions are the simplest to implement in the Control Unit, since all R-Type instructions have the same opcode, h'00. When this opcode is received by the CU, it then differentiates between an ALU function with two data ports set to OP1 and OP2, an ALU function with OP1 and a OP2 set to a shift amount, or the jump register function. The CU is not concerned with the functions of the ALU, it just calls them to be executed.

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

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

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

Figure 6.3. R-Type Instructions

Upon completion of the next phases of the State Machine, the R-Type instructions will be called again in the WRITE BACK phase.

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

Figure 6.4. R-Type Write back

D. I-Type Instruction Design

I-Type Instructions we implemented by using cases of the opcode provided to figure out which instruction needed to be executed. The I-Type instructions not included implementation

of the Execution Procedure of the CU are the brach if not equal (bne) and branch if equal (beq).

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

Figure 6.5. I-Type Instructions

Since some I-Type instruction use an immediate in their functions, a ZeroExtended, a SignExtended, a LUI, and a BrachAddress were made. These are called upon in the correct instance using the code given in the instruction phase.

```
//Immediate sign extension
SIGN_EXTENDED = {{16{immediate[15]}},immediate};
//Immediate zero extension
ZERO_EXTENDED = {16'h0000, immediate};
//LUI value
LUI = {immediate, 16'h0000};
```

Figure 6.6. Sign Extensions / Immediate Values

After execution, the instructions that control reading and writing to the memory, lw and sw, are implemented in the Memory procedural state of the CU. The correct instruction of the two above is determined by, once again, case tests of the opcode provided.

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

Figure 6.7. Load Word / Store Word Instructions

Finally, the write back phase will execute the code below using the correct opcode.

```
//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
```

Figure 6.8. I-Type Write back

E. J-Type Instruction Design

Finally, J-Type operations are the last to be implemented. Of the four operations, the push operation is the only one to take place in the Execution procedural state of the CU, while the others will take place in the write back state. By default,

the address of DATA_R1 is set to register 0. The control unit is aware of this state by running test cases on the opcode.

```
//J-Type
6'h1b :
begin
RF_ADDR_R1_RET = 0;
end
endcase
//Store 32-bit jumpaddress
JUMP_ADDRESS = {6'b0, address};
//PUSH
6'h1b :
begin
  MEM_ADDR_RET = SP_REF;
  MEM_DATA_RET = RF_DATA_R1;
  MEM_WRITE_RET = 1'b1;
  SP_REF = SP_REF - 1;
end
//POP
6'h1c :
begin
  SP_REF = SP_REF + 1;
  MEM_ADDR_RET = SP_REF;
  MEM_READ_RET = 1'b1;
end
```

Figure 6.9. J-Type Instructions

Since most of the J-Type instructions are implemented in the write back phase, this means that they all either write to registers in the program or they are changing the PC_REG in the program.

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

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

Figure 6.9. J-Type Write back

VII. TESTING

Testing was done by running the 'DaVinci_tb.v' in the simulator. Using the RevFib.data and Fibonacci.data, we are able to test the read and write capabilites of the program and see the wavelengths of where the program is running.

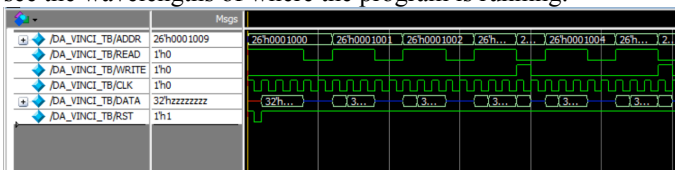

Figure 7.1. DaVinci_TB Test Wavelength

By comparing the results of the data dump to the data.golden file, we can tell where the processor ran correctly or incorrectly.

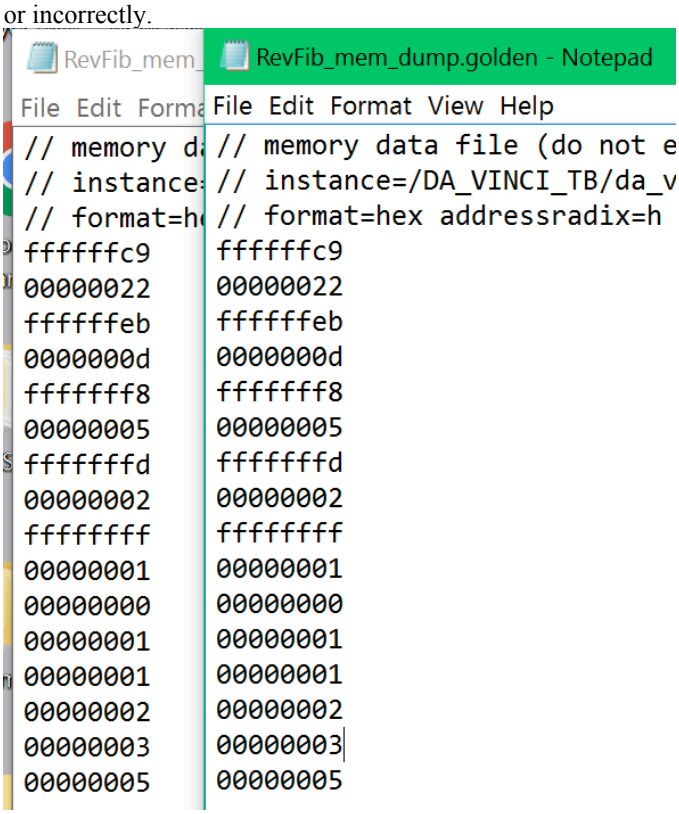

Figure 7.2. Actual RevFib Output (Left) vs. Expected Output (Right)

VIII. CONCLUSION

The DaVinci v1.0 system offers an in-depth model for creating a processor based up on the 'CS147DV' instruction set. The system demonstrates the connections between the Processor, Memory, Register File, ALU, and Control Unit. By building the main parts of the processor, we learned how to efficiently use the Verilog language to create a working microprocessor.