Behavior Model and Simulation of a 32-Bit Processor With DaVinci v1.0 System

Xinken Zheng
CS 147 Section: 01
Zhengxinken@gmail.com

***Abstract*— This report serve the purpose of the creation to the simulation of a DaVinci v1.0 system processor. The model of the simulating is done by supporting the use the 'CS147DV' Instruction Set, a 32x64M Memory module, a Register File, and a Arithmetic Logic Unit (ALU) as well as a Control Unit (CU). This paper below will demonstrates all the parts itself and finally how these individual parts together give the functionality to the processor**

I. Intro

The goal of this project is to show how one can implement a processor with DaVinci v1.0 system using Verilog program language and the simulation tool of ModelSim. This processor serve as a simple guideline of what an actual processor is like in Verilog language. The DaVinci v1.0 system is made up of Register File, memory models, control unit, and an ALU. Test bench are created for each separate components to ensure it is working correctly. The test bench files are 'alu_tb.v' 'da_vinci_tb.v' 'mem_64MB_tb.v' 'register_file_tb.v'. These test bench are all written using Verilog language.

II. Requirements for system

The following will show all of the parts of the DaVinci v1.0 System processor and the memory that are needed to run.

A. Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit is the most important component of the DaVinci v1.0 system. The ALU handles the calculation logic work with the given nine operations in the 'CS147DV' Instruction Set provide by professor Patra with the addition of a Zero flag for the system. The Arithmetic Logic Unit for this project remains the same from project 1.

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

B. Register File

There will be a total of 32 registers in the register file, starting at 0 to 31. The size of each register will be 32-bit word. Reset operation is designed to execute when the clock is on negative edge, and read and write operation will be done on the positive edge. When 11 or 00 is present, read and write operations is in High-Z and register file will be on hold. And read and write can be executed when it is 01 or 10.

C. Memory

Total memory capacity for the memory module in this project is 64M, it is addressable for a 32-bit storage at each address. During negative edge of the clock, reset operations are being executed, and during positive edge of the clock, all other operations are executed. When read and write are set to 11 or 00, Data is set to X, or don't care.

D. Control Unit (CU)

The control unit is responsible for the processor signal and directs the flow of input and output . It is done by a five-state machine. At the positive edge of the clock, the cycle changes between Fetch(IF), Instruction Decode(ID),Execution(EXE), Memory(MEM) and write back(WB). The Control units follows the instruction set "CS147/DV" for its decisions.

E. CS147DV Instruction Set

Professor Patra from San Jose State University helps provide the R-Type, I-Type, and J-Type instructions. The instructions will be mainly used by the control unit to make sure they are working correctly for the DaVinci v1.0 System. These three types of instruction all have unique machine codes for different type on instructions.

I. Implementation and design of the ALU

The Davinci v1.0 System will use a 32-bit ALU, the ALU will have a total of 9 instructions with the addition of zero flag which was not added from the previous project. Addition, Subtraction, Multiplication, Shift Logical Right, Shift Logical Left, Logical AND, Logical OR, Logical NOR, Set Less than, and the ZERO System Flag. These are the instructions in the 'CS147DV' instruction set, we will be implementing this with Verilog language.

A. ALU Design

We went over the ALU design over in the previous project. It is very simple, It is done by taking two operands, compute the instruction based on the operation code, and store the result

in OUT, which is the output of the result. For example, 'ALU_OPRN_WIDTH'h0(n)' n is the specific instruction it correlates to. For example, 'h03' corresponds to multiplication while 'h04' corresponds to division. OP1 and OP2 refers to operand 1 and 2. Zero flag checks the output.

B. ALU Operations

The following 9 instructions are coded in 'ALU.V' file. These are the instructions that will be responsible by the ALU with the addition of the ZERO flag.

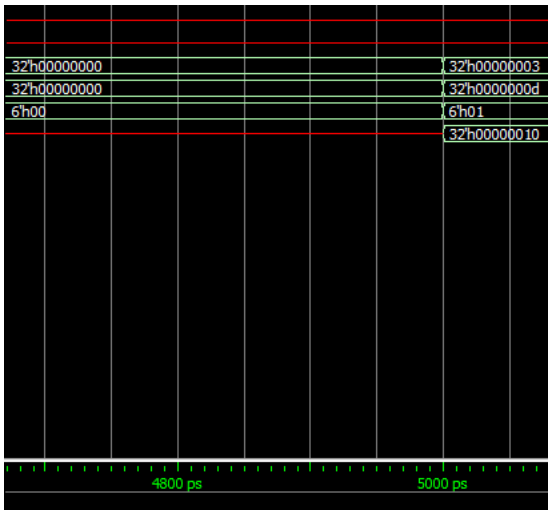
```
always @(OP1 or OP2 or OPRN)
begin
    case (OPRN)
        `ALU_OPRN_WIDTH'h20 : OUT = OP1 + OP2; // Addition
        `ALU_OPRN_WIDTH'h22 : OUT = OP1 - OP2; // Subtraction
        `ALU_OPRN_WIDTH'h2c : OUT = OP1 * OP2; // Multiplaction
        `ALU_OPRN_WIDTH'h02 : OUT = OP1 >> OP2; // Shift Logical 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

'ALU_TB_V' will be the test bench file that will test the ALU to determine if it is running expectedly. Wave graph also provided.

```
# vsim
# Start time: 02:15:31 on Apr 02,2020
# Loading work.alu_tb
# Loading work.ALU
/SIM 15> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 10 - 2 = 8 , got 8 ... [PASSED]
# [TEST] 7 * 5 = 35 , got 35 ... [PASSED]
# [TEST] 21 & 3 = 1 , got 1 ... [PASSED]
# [TEST] 13 | 7 = 15 , got 15 ... [PASSED]
# [TEST] 14 ~| 6 = 4294967281 , got 4294967281 ... [PASSED]
# [TEST] 14 < 2 = 0 , got 0 ... [PASSED]
# [TEST] 5 << 9 = 2560 , got 2560 ... [PASSED]
# [TEST] 11 >> 2 = 2 , got 2 ... [PASSED]
#
#       Total number of tests          9
#       Total number of pass           9
#
# ** Note: $stop    : C:/Users/xinken/Desktop/prj_02_source-1-23/alu_tb.v(119)
#    Time: 95 ns  Iteration: 0  Instance: /alu_tb
# Break in Module alu_tb at C:/Users/xinken/Desktop/prj_02_source-1-23/alu_tb.v line 119
/SIM 16>
```



III. Register File Design and Implementation

The purpose of the Register file is to determine if Read instruction or Write instruction will be executed. It will take the input of both and set DATA_R# to don't care if write and read are 11 or 00. The Register contains Data_R1 and Data_R2 which are data from locations in ADDR_R1 and ADD_R2. Finally, ADDR_W is the address for DATA_W.

```
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}
;
```

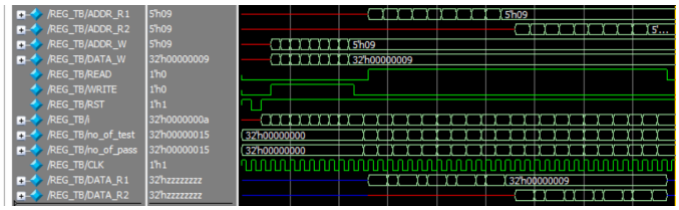
Figure 4.1. Register File Operations

Almost all of the operations will be done when the clock is on the positive edge. The clock will be set to RST when it on the negative edge. Register reads out read and write are the opposite of each other. when Read=1'b1 and write =1'b0, and also when Read=1'b0 and write =1'b1. Register will read out.

```
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;
```

Plenty of test can be done for the functions of *register_file.v*, To accomplish that ,we need to ensure the read and write operation is working properly in the register file, as they need to be stored in the right address. In the wavelength

graph, we can see that indeed read and write operation is working correctly according to the system. Register file will test the function for write by DATA_W 0 to 9 with the matching register addresses from ADDR_W. We can see from the wavelength that both read and write operations will be zero while the registers are being outputted. Then, DATA_R1 and R2 are access to read the data.




```
# /n    Total number of tests         41
#       Total number of pass          41
#
# ** Note: $stop    : C:/Users/xinken/Desktop/CS147_PROJ2_FINAL/register_file_tb.v(86)
#    Time: 1250 ns  Iteration: 0  Instance: /register_file_tb
```

IV. Memory Design with Implementation

The Memory design and implementation for this processor will contain input and output port. Data will be retrieved from *ADDR_MEM* when read operation is set at 1 and write operation is set to 0.the address of the data is being delivered as the output from port *DATA_inout.* And wise versa, if read operation is set at 0 and write operation is set at 1, the present data in the present register will get send to and stored in memory, this is what the function store word does. The data will remain in High-Z mode during read and write operation.

```
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
```

Remember, every process only happen on the positive side of the clock. If the clock is on the negative edge, Reset occurs and gets on the negative edge, and RST becomes ‘1b’0’. This ensure that the data from the memory will be set to zero and give us the ability to read data from ‘mem_init_file”.

```
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);
```

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

A. *Control Unit and State Machine*

The main purpose of the control unit is to direct traffic in the processor, it will help the processor to switch state in an efficient manner. There are five states in an state machine, IF,ID,E,MEM,WB. The control unit will finish one state every clock cycle. Also, the control unit will be in charge of the logic unit and the arithmetic, managing input and output with commanded program instructions.

```
//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;
```

State machine is custom and built to change states only when the clock is at positive. Next_state tells the machine to switch to thr next state of the five states, and will run unless state machine is inactive.

```
// 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
```

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 instruction are identified by an opcode of 0, 6-bits in our case. They are differentiated by function values, which is also 6 bits in our case. R-type instructions only use registers. The following is the format for R-type instructions.

BITS:	31-26	25-21	20-16	15-11	10-6	5-0
	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-FORMAT INSTRUCTION

op - operation
rs - source register
rt - transition register (made this up)
rd - destination register
shamt - shift amount
funct - function

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.

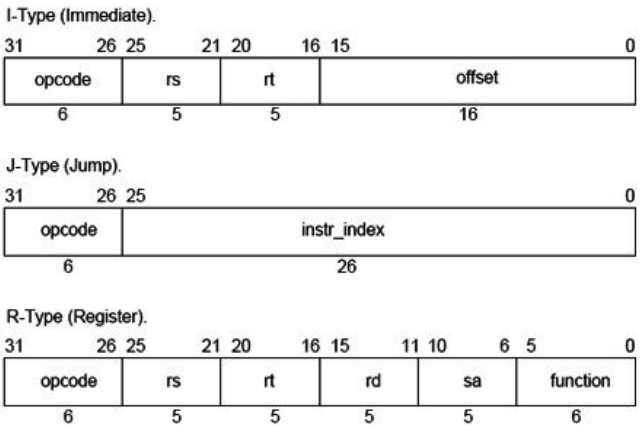
```
if (proc_state === `PROC_EXE)

begin
case (opcode)

6'h00:
begin
if(funct === 6'h01 || funct === 6'h02)
begin
        ALU_OPRN_RET = funct;
        ALU_OP1_RET = RF_DATA_R1;
        ALU_OP2_RET = shamt;

end

else if (funct === 6'h08)
begin
        PC_REG = RF_DATA_R1;
end

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

When these instructions are done, the state machine will switch its state to write back, and then, we will see R-Type instructions will be called again.

```
else if (funct === 6'h08)
begin
        PC_REG = RF_DATA_R1;
end

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

D. Instruction for I-type

I-Type instructions operates on an immediate and register value. It's compositions include a 6-bit opcode, 5-bit for Rs, 5-bit for Rt, and 16-bit immediate value. In I-type instructions, we will figure out what we need to carry out from opcodes. Branch if equal or branch if not equal will not be included.




```
6'h08:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h01;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
6'h1d:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h03;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
6'h0c:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h04;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = ZERO_EXTENDED;
end
6'h0d:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h05;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = ZERO_EXTENDED;
end
6'h0a:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h07;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
6'h23:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h01;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
6'h2b:
begin
ALU_OPRN_RET = `ALU_OPRN_WIDTH'h01;
ALU_OP1_RET = RF_DATA_R1;
ALU_OP2_RET = SIGN_EXTENDED;
end
```

Because some of the I-Type operations uses immediate values, SIGN_EXTENDED, ZERO_EXTENDED, and LUI is

implemented. They will be used in the case when I-Type instructions contain immediate values. We also need to implement load word and store word instructions that are responsible for reading and writing to memory.

```
SIGN_EXTENDED = {{16{immediate[15]}}, immediate};
ZERO_EXTENDED = {16'h0000, immediate};
LUI = {immediate, 16'h0000};
```

```
case (opcode)
6'h23:
begin
MEM_ADDR_RET = ALU_RESULT;
MEM_READ_RET = 1'b1;
end

6'h2b:
begin
MEM_ADDR_RET = ALU_RESULT;
MEM_DATA_RET = RF_DATA_R2;
MEM_WRITE_RET = 1'b1;
end
```

And then, the following code will be executed by write back phrase.

```
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'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
```

E. Instruction for J-type

The last type of instructions are the J-type instructions, or jump type instructions. These instructions require a 26-bit coded address field to specify the target for the jump. They have a 6-bit opcode with a 26 bit target jump address.

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>

14

J-Type instructions are the last thing we will be implementing for the control unit. These instructions will most happen in the write back state in the state machine, while only the push instruction will happen in the Execution state of the control unit. There are four important function for J-type, pop, push, jal and jmp.

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
```

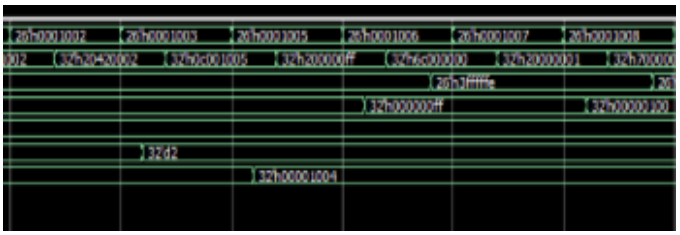
Figure 6.9. J-Type Instructions

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.

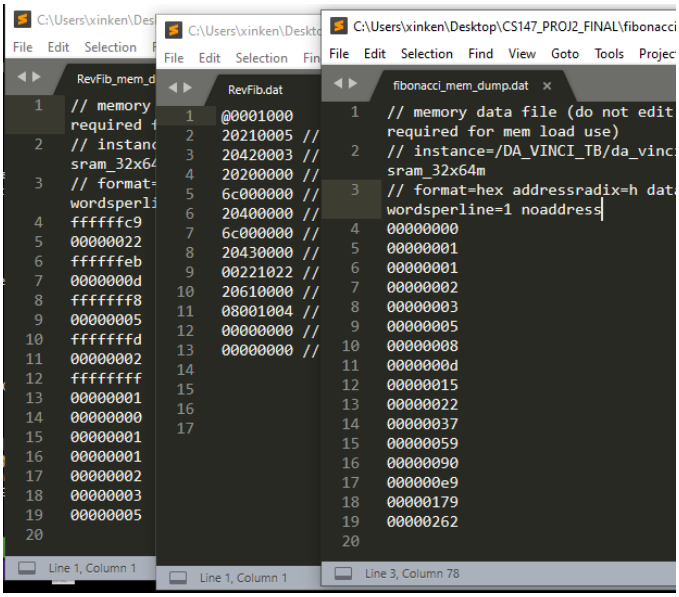
```
6'h02: PC_REG = JUMP_ADDRESS;

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
endcase
```

III. TESTING

At last, we do testing for all of the components in the program, alu,control unit, and register file with ‘davinci_tb.v’. two data files are given to compare the data output with our program. The two files are “RevFib.data’ and ‘Fibonacci.data’. Wavelengths can also be used for read and write test.



Below are the test result I got, comparing it to the data given by ‘xxx.dump.golden’, they are appear to be the same, therefore my program is running correctly.



IV. CONCLUSION

The project was incredibly more challenging than the first one we did, thankfully with the help of my classmates, I was able to understand and do it easier. I learned a lot about processor functionality. I was able to write my own test cases for the processor and testing with waves. It help me get a much deeper understanding of the Verilog language syntax and usage. I am now more familiar with the ModelSim environment and program platform. Overall the project was difficult but rewarding.