Designing and Implementing the CS147DV Instruction Set

Michael Wong
San Jose State University, Department of Computer Science
California, United States of America
michael.j.wong@sjsu.edu

Abstract

In this project, I will explore the steps to design and implement the CS147 Instruction Set. Upon completion of this task, I will create and run testbenches on my design to test its functionality.

Introduction

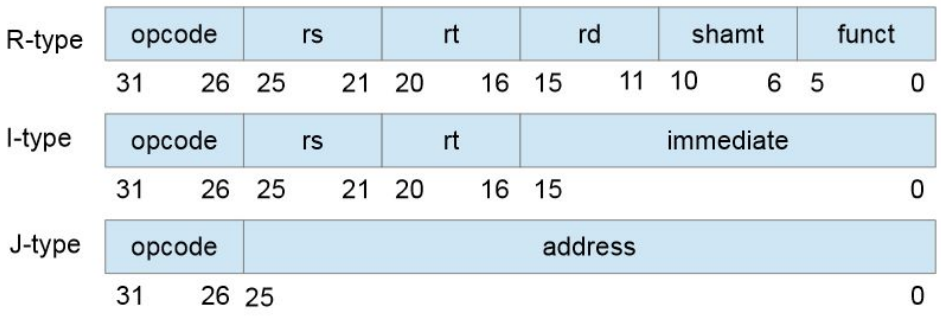
An instruction set is a mutual relationship between software and hardware in regards to a list of operations that a machine can implement. It is a specification of implementations for both hardware and software engineers. The CS147DV Instruction Set is a custom made instruction set used to aid students enrolled in CS 147 in their efforts to understand processors and memory relations. This project will demonstrate how one can create a functional custom instruction set written in Verilog. The instruction set will then be tested for practicality. The purpose of this project is to understand how to design and implement the CS147DV Instruction Set and test its applications.

Requirements for CS147DV

Before I can begin to design and implement the CS147DV Instruction Set, I must understand its fundamentals first. A minimal computer system called DaVinci v1.0 was provided to me to act as the behavioral model supporting the CS147 Instruction Set. The CS147DV Instruction Set can support 3 types of instructions. They are Register Type(R-Type), Immediate Type(I-Type), and Jump Type(J-Type). All three of these instructions have their own format, but the one constant in all of them is the opcode. The opcode is how the instruction set will know what operation is being performed. However, in terms of R-Type instructions, the opcode for all operations are the same, 6'b0. The different operations that an R-Type instruction can perform is signaled by its funct sub-operation. Hence, an opcode of 6'b0 will inform us that it is an R-Type instruction and then funct will tell us which R-Type instruction. For the I-Type instructions, there is an immediate field. The immediate field can be either a zero extension or a sign extension. Determining whether to use sign extension or zero extension is specified in the CS147DV instruction set. Finally, the J-Type instructions use a jump address. The jump address is simply the address where the processor wants to jump forward to. The R and I Type instructions uses register to gather data from the register files in order to execute their respected operations. Details pertaining to the opcodes/funct and operations can be found below:

'CS147DV' Instruction Set

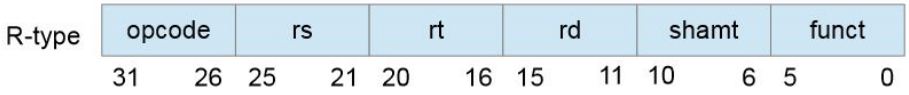
- 3 types of instructions.
  - Register or R type
  - Immediate or I type
  - Jump or J type



'CS147DV' Instruction Set

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>



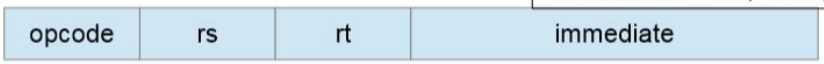
'CS147DV' Instruction Set

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



31 26 25 21 20 16 15 0

'CS147DV' Instruction Set

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

The CS147DV Instruction Set utilizes a state machine to cycle through the steps that the processor takes to perform these tasks. There are five states in this state machine: Instruction Fetch, Instruction Decode, Execution, Access Memory, and Write Back. The state machine will move to the next state every time there is a positive clock signal. The cycle starts with Instruction Fetch, moves down the list and ends with Write Back. If the state machine reaches Write Back, it will cycle back to Instruction Fetch. Upon a reset signal, the state machine will set the current state to Instruction Fetch regardless what state the machine is currently on. Each of these state will play a role in the executing and producing a result from the desired operation. What each state does will be discussed in the designing and implementing section.

Like many other processors, the CS147DV has an ALU. The ALU for this instruction set must have certain operations in order for this set to be applicable. Those operations are addition, subtraction, multiplication, and, or, nor, set less than, shift left logical and shift right logical. In addition to these operations, this particular ALU will have a ZERO flag. The purpose of the ZERO flag is to signal to the processor if the result of the ALU operation is a zero or non-zero

value. The ZERO flag will be a logic 1 if the ALU's result is indeed zero or logic 0 if the result is non-zero.
A register file is needed for the CS147DV instruction set to function correctly. The register file is used to store and transfer data throughout the input and memory. The register file has a 32 bit memory storage for its memory management. The register file should be initialized to have all zeros across all 32 registers. When a negative edge of the reset signal triggers a reset, all 32 register will be set to zero as well. The register file will need to handle read and write signals to the register file addresses. A logic 1 for read and a logic 0 for write indicates a read cycle, whereas a logic 1 for write and a logic 0 for read indicates a write cycle. If both signals are either 0 or 1, then the register file must be implemented to hold the previous value that was stored in the register. In this case, data will not be written or read. What was previously there will remain there until another read or write signal is triggered.

Design and Implementation for CS147DV

The processor consist of the ALU, register file, and the control unit. The design and implementation of the memory model will be explained as I discuss its relevance with the three components mentioned above. To start the memory model is designed to take information generated from the processor and store it into a memory source outside the processor. In terms of this project, the memory source will be data files that contain the instructions to be executed and read from.

The design and implementation for the ALU was simple and easy to execute. This ALU needs to support 9 operations and those operations were given initial flags to signal the different operations such as 'h01 for addition. To implement the other operations, I followed the same structure from the first project.

```
`ALU_OPRN_WIDTH'h01 : OUT = OP1 + OP2;      //addition
```

I would follow this pattern for the rest of the 8 functions until all the operations were covered. These lines of code would be contained inside a case statement. The case statement takes in the OPRN input and selects the desired operation to perform. In addition to performing these functions, the ALU also has a zero flag that will signal if the result of the ALU operation is a zero or not. To implement the zero flag, I initialized the flag to hold a logic zero, stating that the ALU operation is non zero by default. After the case statement, I simply added an if statement to check if the result of the ALU operation is indeed zero.

```
//checks if OUT is equal to 0 and sets ZERO to 1 if so
        if (OUT === 0)
   ZERO = 1'b1;
```

With the case statement managing what operation to perform and checking for a zero result, the design and implementation for the ALU is complete.

The register file starter code that was provided to us was similar to the memory model code that was also given. The design for the register file consisted of data and address inputs along with data outputs. The register file module is meant to behave such as an actual register file component complete with those address and data ports and read and write signals. My first step was to assign registers for the provided input and outputs. Then, the register file needs a place to store the contents of its registers. Similar to the memory model, the register file will also have a memory storage expect, instead of 64 bit, it will be 32 bit to represent a 32 bit register file. Once I have created the memory for the register file, the 32 registers need to contain a void value to hold. In this case, it was selected for the registers to hold an initial value of zero and to be set to zero upon a negative edge of a reset signal. Code below:

```
//intialize the 32 registers as 0
initial
begin
  for (i = 0; i < `REG_INDEX_LIMIT; i = i + 1)
  begin
   mem_32x32[i] = {`DATA_WIDTH{1'b0}};
  end
end

always @ (negedge RST or posedge CLK)
begin
   // TBD: Code for the register file model

   //resets the block on a negative edge of the reset
   if (RST === 1'b0)
   begin
         for (i = 0; i < `REG_INDEX_LIMIT; i = i + 1)
                  begin
          mem_32x32[i] = {`DATA_WIDTH{1'b0}};
                  end
   end
```

With the register file's memory set, I moved on to implement the read and write functions of the component. The register file has a read and write signal that is determined by their respected logical value. In the case of a hold, I checked to see if the register file was given a hold signal. If

this was true, then I set the data registers to hold their previous values. If not, then I assigned them to the registers that will contain what to read. Code shown below:

```
assign DATA_R1 = ((READ===1'b1)&&(WRITE===1'b1) ||
(READ===1'b0)&&(WRITE===1'b0))?{`DATA_WIDTH{1'bz}}:reg_data_r1 ;
assign DATA_R2 = ((READ===1'b1)&&(WRITE===1'b1) ||
(READ===1'b0)&&(WRITE===1'b0))?{`DATA_WIDTH{1'bz}}:reg_data_r2 ;
```

A read operation is used to read the data from the memory addresses and return them through the output data registers. As stated in the requirements section, a read signal is determined when read has a logic 1 and write has a logic 0. To do this, I set the data registers to the corresponding addresses in memory. Here is the verilog code:

```
if (READ === 1'b1 && WRITE === 1'b0)
        begin
        reg_data_r1 = mem_32x32[ADDR_R1];
        reg_data_r2 = mem_32x32[ADDR_R2];
        end
```

A write operation is used to write data into the memory registers of the register files. As stated in the requirements section, a write signal is determined when read has a logic 0 and write has a logic 1. In order to implement write, I have to assign the value of the data to be written to a place in the register file's memory. The location of the register is determined by the write address' value. The code can be found here:

```
else if (READ === 1'b0 && WRITE === 1'b1)
        begin
        mem_32x32[ADDR_W] = DATA_W;
        end
```

The hardest design and implementation for this project was the control unit. The control unit can be split into two separate tasks. The two applications that needed to be implemented are a state machine and generating a control signal. I will first discuss the design and implementation of the state machine. For the control unit I just had to create two registers, state and next_state, to hold information regarding the states for the state machine.
As stated in the requirements, the state machine has 5 states that it must cycle through. CS147DV's state machine starts at Instruction Fetch as its first state. In order to make the state machine start off in the correct spot, I initialized the state to be an unknown 2 bit value, but the next state as Instruction Fetch. This forces a one way path for the state machine to reach

Instruction Fetch. I also used the same approach to set the machine to Instruction Fetch when the negative edge of reset was hit.
With a starting point for the state machine established, there needs to be a way to transverse through the different states of the machine. As mentioned in the requirements, on every positive edge of clock, the state machine will toggle to the next state. To implement this, I created an always statement that will execute when a positive edge of clock is triggered. I used a series of if-else statements to check what state the machine is currently in and then assigned next_state to its proper following state. To toggle, I simply set the state to next_state. With this I was able to create a functioning state machine to toggle through the different stages of the machine. Here's the code:

```
always @ (posedge CLK)
begin
        if (STATE === `PROC_FETCH)
        begin
   next_state = `PROC_DECODE;
        end
        else if (STATE === `PROC_DECODE)
        begin
   next_state = `PROC_EXE;
        end
        else if (STATE === `PROC_EXE)
        begin
        next_state = `PROC_MEM;
        end
        else if (STATE === `PROC_MEM)
        begin
        next_state = `PROC_WB;
        end
        else if (STATE === `PROC_WB)
        begin
        next_state = `PROC_FETCH;
        end

        STATE = next_state;  //changes the state to the next state

end
```

The next portion of the control unit that needed to be designed and implemented was generating the control signal generation. First, I had to connect the outputs to corresponding registers along with internal registers such as program counter, stack pointer, and current instruction. These registers are designed to aid in the instruction set's design, particularly for jumping and stack operations. Each of the states that were described in the state machine play different roles in the execution of certain instructions. Depending on the instructions, the different states will set up, execute and return results for those instructions. I will discuss the different actions that the different states will take during their respective states.
To signal for an Instruction Fetch in the control unit, I was given definitions for these states. To check for the different states, I was given a case statement to check the different states. Each state is handled in the case statement. For the first state, Instruction Fetch, the design is to set the memory control to read and the memory address to the program counter. In addition, the register file's control signals need to be set to hold their previous data. Implementation consisted of setting the memory's address and the signals appropriately. Here's the code:

```
`PROC_FETCH :
  begin
        MEM_ADDR = PC_REG;    //sets memory address to next instruction
        //issues read signal to memory
        MEM_READ = 1'b1;
        MEM_WRITE = 1'b0;
        //makes sure that the register files are holding and no reading is done
        RF_READ = 1'b0;
        RF_WRITE = 1'b0;
  end
```

Instruction Decode is where the setup for all the instructions are being generated. The design for the CS147DV instruction set has three types of instructions, R, I, and J. It is in Instruction Decode where I set up the instruction set for all the instructions. To begin, the current instruction needs to be declared and then configure the register file to the correct set up. The register file needs to be in a read cycle so that it can read what operation to conduct and prepare for. Following the model of the CS147DV instruction set, the registers rt and rs contain the operands to be used for R and I type instructions. Hence, the addresses of the register files will need to contain the data from rt and rs. Lastly, there are constants that can be declared so that future operations that involve I or J type instructions can be completed.

```
`PROC_DECODE :
  begin
        INST_REG = MEM_DATA;    //stores memory read data into current instruction
```

```
        // parse the instruction to decode
        // R-type
        {opcode, rs, rt, rd, shamt, funct} = INST_REG;
        // I-type
        {opcode, rs, rt, immediate } = INST_REG;
        // J-type
        {opcode, address} = INST_REG;

        //helpful setups for sign/zero extension, lui value and the jump address
        SIGN_EXTEBD = {16{immediate[15], immediate}};
        ZERO_EXTEND = {16'b0, immediate};
        LUI = {immediate, 16'b0};
        JUMP_ADDR = {6'b0, address};

        //sets the read address of RF as rs and rt
        RF_ADDR_R1 = rs;
        RF_ADDR_R2 = rt;
        //sets the register file operation to read
        RF_READ = 1'b1;
        RF_WRITE = 1'b0;
        //print_instruction(INST_REG);
    end
```

Execute is the third state of the state machine. This is where, for most of the operations, the work to generate a result is conducted. All the instructions in the CS147 Instruction Set has an opcode to signal which instruction is being conducted. An exception is for the R-Type instructions that have a funct field. All R-Type instructions have the same opcode, 6'b0, but the different operations within the R-Type instruction have unique funct fields to indicate what operation needs to be performed. With this design in mind, I created a case statement to check for opcodes and funct fields. If the opcode signals a 6'b0, then it is a Register Type instruction with further need to check their funct field. Afterwards, simply setup the ALU inputs appropriately. Here's the code:

```
`PROC_EXE :
        begin
    case(opcode)
     6'b0:          //R-Type Instructions
     begin
        case(funct)
```

```
        6'h20 : ALU_OPRN = `ALU_OPRN_WIDTH'h01;          //sets the ALU operation
code to addition
         6'h22 : ALU_OPRN = `ALU_OPRN_WIDTH'h02;         //sets the ALU operation
code to subtraction
         6'h2c : ALU_OPRN = `ALU_OPRN_WIDTH'h03;         //sets the ALU operation
code to multiplication

//continues on with the rest…

        endcase
        //assign inputs for the ALU srl and sll uses the shamt amount as opposed to the data in the
second register file
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = ((funct === 6'h01)||(funct ===6'h02))?shamt:RF_DATA_R2;
    end
//continues on…
```

Next are the I and J Type instructions. Each of the operations they can perform carry unique opcodes, so I don't need to then further check for another field. To implement, I followed the operation format and correlate it with the setup of the code. Providing the inputs for the ALU is done in this state to fulfill their task. It is worth to note that operations such as lui, jump, and jal do not need to be handled in the Execution state since they by nature do not require the assistance of the ALU. Also, I handled the ALU operations for push and pop directly in the Memory Access state since I felt it was simple and unnecessary to go through configuring the ALU to perform that task. Lastly, the push operation needs its first register file address to be set to zero since push stores data to r[0] on the stack. A lot of the code is repetitive, so I will provide only a couple of examples:

```
6'h0c:    //andi
   begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h06;
        ALU_OP1 = RF_DATA_R1;
        ALU_OP2 = ZERO_EXTEND;
   end

6'h04, 6'h05:    //beq or bne, they use the same oprn and ops
   begin
        ALU_OPRN = `ALU_OPRN_WIDTH'h01;
        ALU_OP1 = PC_REG + 1;
```

```
        ALU_OP2 = SIGN_EXTEND;
    end
```

The Memory Access state is when instructions regarding accesses to memory are required. These include load word, store word, push and pop since they are the operations that interact with memory. Depending on the operation that is taking place, a memory read or write signal will be in effect. By default, the memory signals should hold if the four mentioned instructions do not occur. The operations store word and push require a write signal since they are changing data in memory whereas a read signal in needed for load word and pop because they are taking data from memory. The code is similar to each other, so presented below are store word and push. Since these are writing to memory the register tasked to write to memory is initialized so that data can be transferred.

```
6'h2b:                  //sw
        begin
         MEM_ADDR = ALU_RESULT;
         WR_TO_MEM = RF_ADDR_R2;    //enables writing data to memory
         MEM_READ = 1'b0;
         MEM_WRITE = 1'b1;
        end

 6'h1b:                 //push
        begin
         MEM_ADDR = SP_REF;
         WR_TO_MEM = RF_ADDR_R1;    //enables writing data to memory
         MEM_READ = 1'b0;
         MEM_WRITE = 1'b1;
         SP_REF = SP_REF - 1;
        end
```

The final state in the state is the Write Back State. The Write Back State is where writing back to the register file or program counter register is done. At the end of the cycle, the program counter must be incremented by one to update how many programs have been run will the machine is running. Also the memory signals should be set to hold since we do not want to read or write to memory during this stage. Writing to the register files or modifications to the program counter are done here. For many of the operations, the register file's write data will be set to the result of the ALU like addi or mult. For the operations that do not rely on the ALU, the program counter will be updated to contain correct information.

```
6'h08, 6'h1d, 6'h0c, 6'h0d, 6'h0a:    //addi, multi, andi, ori, slti
        begin
                RF_DATA_W = ALU_RESULT;
                RF_ADDR_W = rt;
                RF_READ = 1'b0;
                RF_WRITE = 1'b1;
        end
```

For some operations such as jal and pop, the register file's data or address lines need to be hard set in accordance to their operation specifications.

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

This concludes the design and implementation of the instruction set CS147DV. Now I will proceed to test key modules of my code and then ultimately the entire system.

Testing

Before testing to see if the entire design and implementation of the CS147DV Instruction Set is correct, I took the approach of testing pieces of the design that would then lead up to the testing of the whole system. Testing the whole system will be done in DaVinci_tb.

To test the ALU, I used my ALU tester from the first project. I just modified the tester to test for the zero flag when a result of zero appeared. To do this I added another input and register to hold the value of ZERO.

input [`DATA_INDEX_LIMIT:0] reszero;   //the actual result of the zero value for ALU result
reg [`DATA_INDEX_LIMIT:0] goldenzero;   // expected zero value for ALU result

These two components would be used to hold the values of the expected and actual result for the ALU's zero flag. To test for correctness, I modified the function test_golden declaration from the first project to also check for the zero flag.

test_golden = (res === golden)?1'b1:1'b0 && (reszero === goldenzero)?1'b1:1'b0; //checks to see if the result and the zero value are correct

The output of my test can be found below:
[TEST] 15 + 3 = 18, 0 : got 18, 0 ... [PASSED]
[TEST] 15 - 5 = 10, 0 : got 10, 0 ... [PASSED]
[TEST] 5 - 5 = 0, 1 : got 0, 1 ... [PASSED]
[TEST] 24 * 0 = 0, 1 : got 0, 1 ... [PASSED]
[TEST] 15 >> 4 = 0, 1 : got 0, 1 ... [PASSED]
[TEST] 12 << 3 = 96, 0 : got 96, 0 ... [PASSED]
[TEST] 3 & 5 = 1, 0 : got 1, 0 ... [PASSED]
[TEST] 7 | 9 = 15, 0 : got 15, 0 ... [PASSED]
[TEST] 0 ~| 0 = 4294967295, 0 : got 4294967295, 0 ... [PASSED]
[TEST] 2147483647 ~| 102 = 2147483648, 0 : got 2147483648, 0 ... [PASSED]

Since my ALU passed all of the test cases, I can conclude that my ALU is working properly.

To test the register file, I reviewed the mem_64MB_tb test bench so that I can model the register file's testbench to resemble that tester. The key difference for this testbench is that I do not need to load data externally. I can just write data into the registers and read from those register to test them. Before I can actually test the register file for it's reading and writing operations, I first need to write something into the register so that something can be read and to test its writing functionality. I chose to write integer values into the registers. They were a series of integers ranging from a specified integer to that integer plus 31. For my test I chose a big number like 100 so that I can test that these registers can hold larger values than 1 for example. I used a for loop to write that data into the addresses for a certain range.

```
//write to the registers so that they have something in them, in this case they're ints from
value-31+value
   for(i = value; i < `NUM_OF_REG + value; i = i + 1)
   begin
#10     REG_DATA_W = i;
```

```
        READ = 1'b0;
        WRITE = 1'b1;
#10     ADDR_W = i;
   end
```

Now that there is data in the registers, I can proceed to test its reading and writing tasks. To test for reading, I assigned the read and write signals to hold so that the registers would hold their current data. As implemented in the register file, when both read and write are logic 0, the register file will hold a high z content. Hence, the registers should contain a high z. An if statement is used to check for this condition.

```
READ = 1'b0;
   WRITE = 1'b0;
   no_of_test = no_of_test + 1;
#10    if(DATA_R1 !== {`DATA_WIDTH{1'bz}} || DATA_R2 !== {`DATA_WIDTH{1'bz}} )
        $write("[TEST] Read %1b, Write %1b, expecting 32'hzzzzzzzz, got %8h [FAILED]\n",
READ, WRITE, DATA_W);
   else
        no_of_pass = no_of_pass + 1;
```

The next part of the register file to test is the write function. The test for the write function is to check if the initialized data that was written into the registers are equal to what is expected to be in those registers. I set the register file to read mode so that it can read the contents of the registers. Then, I used an if statement to check if the data in the registers are equal to the expect value. The same for loop that runs though the range of numbers that was used to insert data into the registers was used to get the expected integers.

```
//test writing to the registers
   for(i = value; i < `REG_INDEX_LIMIT + value; i = i + 2)
   begin
        READ = 1'b1;            //set to read so that you can read what's been written
        WRITE = 1'b0;
        ADDR_R1 = i;
        ADDR_R2 = i + 1;
        no_of_test = no_of_test + 1;
#10     if(DATA_R1 !== i || DATA_R2 !== i+1)
        $write("[TEST] Read %1b, Write %1b, expecting %8h, %8h  got %8h, %8h
[FAILED]\n", READ, WRITE, i, i+1, DATA_R1, DATA_R2);
        else
```

```
        no_of_pass = no_of_pass + 1;
    end
```

Once I ran the simulation for the tester, it outputted two lines stating that the register file passed my test conditions. The register file is behaving as expected.

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

I also ran the mem_64_tb testbench that was given to me to reassure that my memory model was functioning correctly. It operates similar to the register file testbench since it was created off the basis of the mem_64_tb. The key difference being that data is being load from outside the source code itself, hench acting as memory. After running the testbench, my memory model passed all it's test cases.

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

After I successfully tested the three testbenches, I can move on to test the entire design and implementation of the CS147DV. Previously stated, a computer system referred as DaVinci v1.0 was provided to me to support the CS147DV Instruction Set. DaVinci v1.0 also comes with a testbench file to assist in testing the CS147DV Instruction Set. Also with the test bench I was provided two .dat files, "fibonacci.dat" and "RevFib.dat." These files contain the information for the tester to run a specified list of operations. A .dat file ending with .golden contains the actual result that you are suppose to get with respect to "fibonacci" or "RevFib." Once the DaVinci testbench gets executed, it creates a .dat file ending with "dump" to store the actual results that the tester reached. These dump files start with their respected titles. Simply checking and comparing the two .dat files will tell me if my implementation of the CS147DV is correct or not. The DaVinci_tb module was able to run successfully for both the fibonacci.dat and RevFib.dat files, but the expected results in the golden file do not equal the results in the dump file.

fibonacci_mem_dump.dat : contents	Fibonaci_mem_dump_golden.dat : contents
00000000 00000000	00000000 00000001

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000	00000001 00000002 00000003 00000005 00000008 0000000d 00000015 00000022 00000037 00000059 00000090 000000e9 00000179 00000262

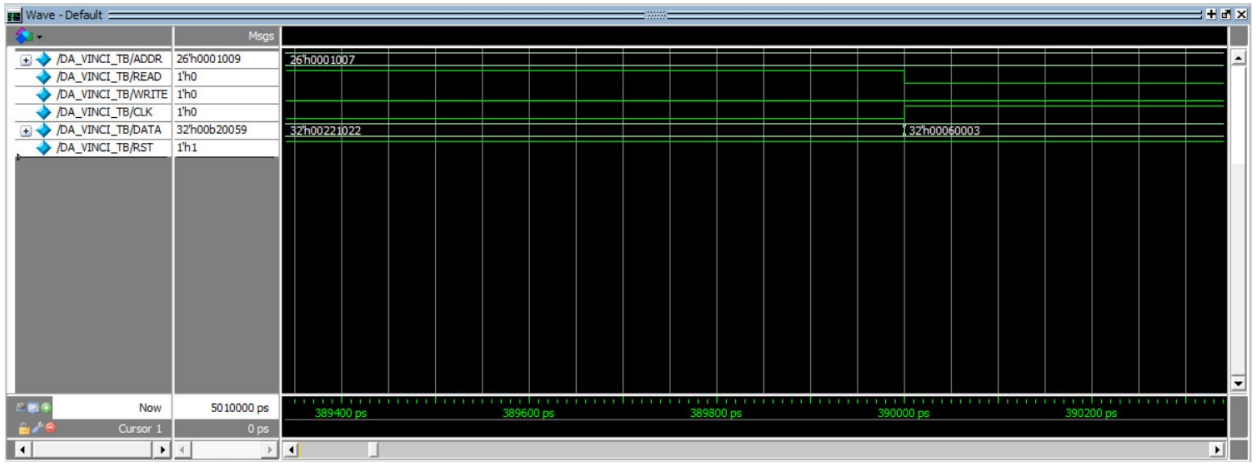
The function that the fibonacci.dat is attempting to do is find the fibonacci sequence in hexadecimal. When I ran my code for fibonacci.dat, I got nothing but zeros. In my debugging process, I discovered that data being passed through the instruction field were not in the correct order.

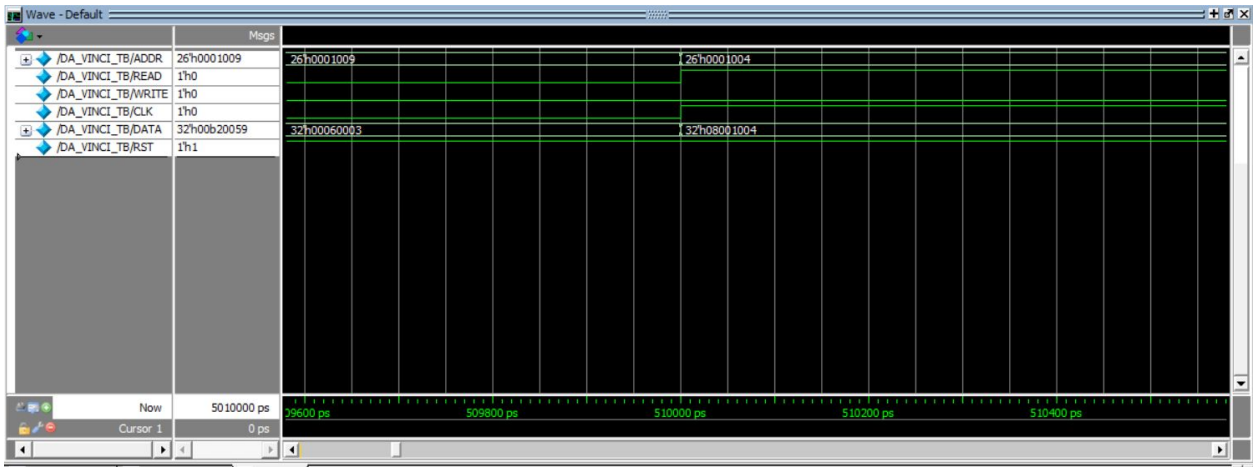
Output from print_instruction	fibonacci.dat
# @ 20ns -> [0X20420001] addi r[02], r[02], 0X0001; # @ 70ns -> [0X3c000100] lui r[00], 0X0100; # @ 120ns -> [0Xac010000] sw r[00], r[01], 0X0000; # @ 170ns -> [0X20000001] addi r[00], r[00], 0X0001; # @ 220ns -> [0Xac020000] sw r[00], r[02], 0X0000; # @ 270ns -> [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;	@0001000 20420001 // addi r[2], r[2], 0x0001; 3C000100 // lui r[0], 0x0100; AC010000 // sw r[1], r[0], 0x0000; 20000001 // loop: addi r[0], r[0], 0x0001; AC020000 // sw r[2], r[0], 0x0000; 20430000 // addi r[3], r[2], 0x0000; 00411020 // add r[2], r[2], r[1]; 20610000 // addi r[1], r[3], 0x0000; 08001003 // jmp loop;

As shown in the table above, my implementation was unable to place data in the correct registers to do the operations. I've checked on my implementations of add, addi, lui, sw and jump. They match the operation format in the instruction set, so I've narrowed it down to the placement of what is being written where. I was unable to find a solution to this problem nor the source for it. I tried switching around the register file's addresses with rt, rs, and rd, but I still receive the same result. I then played around with ordering of code like the location of assigning address' write and address' data. It also can't be the ALU or the register file that was causing the problem since I've tested for their correctness. I deduced that the problem resides in control_unit, but could not find a solution to the problem. The result after running RevFib was at least closer to the expected output.

RevFib_mem_dump.dat contents	RevFib_mem_dump_golden.dat
ff91ffc9	ffffffc9
00440022	00000022
ffd5ffeb	ffffffeb
001a000d	0000000d
ffefffff8	fffffff8
000a0005	00000005
fff9fffd	fffffffd
00040002	00000002
fffdffff	ffffffff
00020001	00000001
00000000	00000000
00020001	00000001
00020001	00000001
00040002	00000002
00060003	00000003
000a0005	00000005

For these cases, they are off by a couple of bits in the 4th and 5th bit position of the hexadecimal results. The data in the addresses are also in a different order like fibonacci. I believe perhaps it has to do with the stack pointer register or there is something causing a shift in the final results. From my testing, I convinced myself that the issue resides in control_unit since ALU, register file, and memory all passed their test cases. Theoretically, I believe I have an applicable design and implementation of the CS147 Instruction Set, but there is clearly something wrong with it. I tried only implementing addi, add, sub, push, sw, and jump, the bare minimum of instructions that the two test files require to run, but still only to run into the same problem. I thought maybe my code was messy and the clutter was causing these outputs, but that was not the reason since it returned the same results. In conclusion, my testing revealed that I was not able to deliver a fully functional instruction set for CS147DV. I also looked at the waveforms to see if a solution could be found there.





Conclusion

In conclusion, I was not able to successfully design and implement a fully functional DaVinci system that would work for the CS147 Instruction Set. I was able to create the building blocks for the instruction set, but failed to deliver its full functionality. I feel that the bug that is preventing me from accomplishing the given task is well hidden and tedious uncover and resolve. Hopefully if I come back to this project I will find that bug and resolve this issue.