32-Bit Processor Simulation – Gate Level Model

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

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

I. Introduction

The goal of the DaVinci v1.0m 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 to be able run the DaVinci v1.0m system.

A. Arithmetic Logic Unit (ALU)

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

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

B. Register File

The Register file is made of 32 registers [0 to 31] that can 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 11 or 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 v1.0m System using logic gates.

III. The Processor Design And Implementation

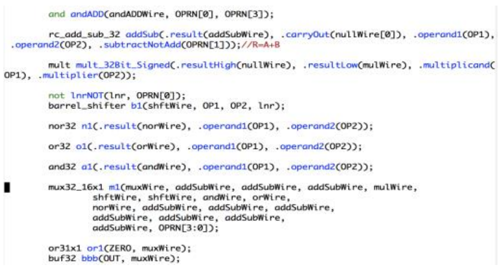
Using the ModelSim simulator, it is simple to program the basic functions of the ALU. This section will outline the design of the ALU using the Verilog Hardware Description Language (VHDL).

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'

B. ALU Operations

There are nine operations that are handled by the ALU. The OPRN is used to select the operation to perform and after the operation, the output, 'OUT' is provided back to the program.

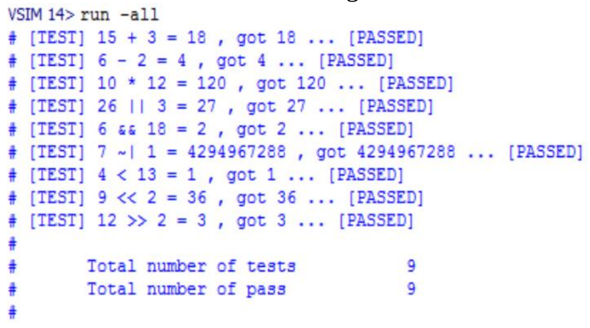


```
        and andADD(andADDWire, OPRN[0], OPRN[3]);

        rc_add_sub_32 addSub(.result(addSubWire), .carryOut(nullWire[0]), .operand1(OP1),
 .operand2(OP2), .subtractNotAdd(OPRN[1]));//R=A+B

        mult mult_32Bit_Signed(.resultHigh(nullWire), .resultLow(mulWire), .multiplicand(
OP1), .multiplier(OP2));

        not lnrNOT(lnr, OPRN[0]);
        barrel_shifter b1(shftWire, OP1, OP2, lnr);

        nor32 n1(.result(norWire), .operand1(OP1), .operand2(OP2));

        or32 o1(.result(orWire), .operand1(OP1), .operand2(OP2));

        and32 a1(.result(andWire), .operand1(OP1), .operand2(OP2));

        mux32_16x1 m1(muxWire, addSubWire, addSubWire, addSubWire, mulWire,
                shftWire, shftWire, andWire, orWire,
                norWire, addSubWire, addSubWire, addSubWire,
                addSubWire, addSubWire, addSubWire,
                addSubWire, OPRN[3:0]);

        or31x1 or1(ZERO, muxWire);
        buf32 bbb(OUT, muxWire);
```

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

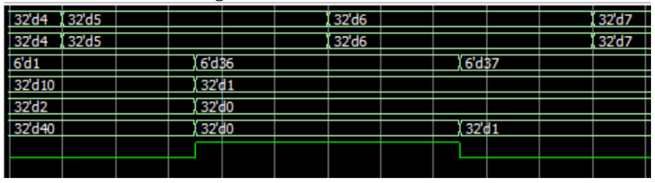


Figure 3.3. Sample ALU Simulation Wavelength

D. *Adder/Subtractor Implementation*

Addition of the bits is done with a half adder. The Half adder implementation uses an XOR gate and an AND gate to get the output.

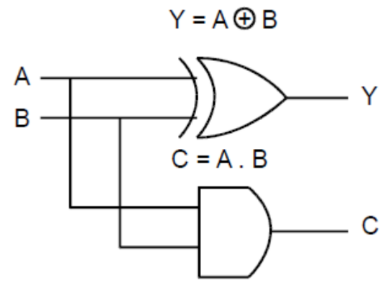


Figure 3.4. Half Adder

A Full Adder is a just two half adders with an OR gate to select the correct Carry Out bit. To implement 32 full adders with the other components.

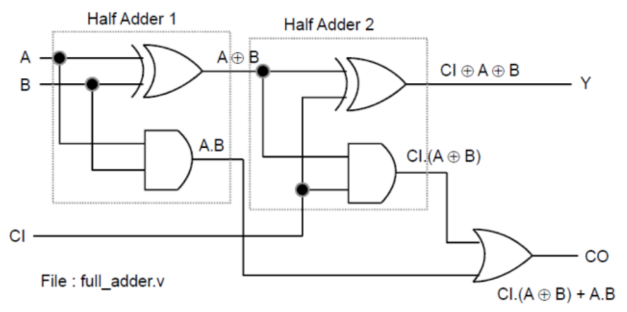


Figure 3.5. Full Adder

To make the circuit compatible with subtraction, we will use a 2’s compliment on the second input bit to simulate subtraction.

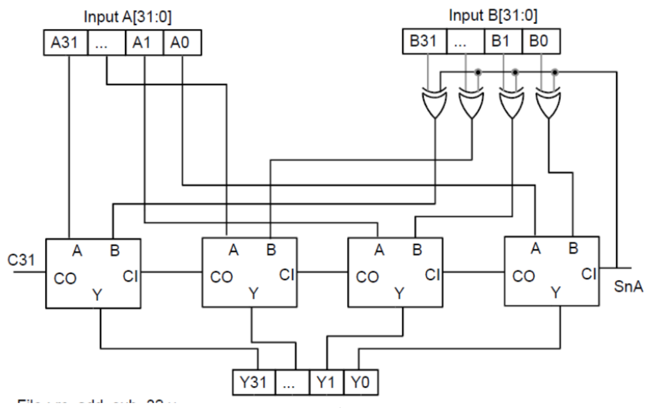


Figure 3.6. Half Adder

E. *Multiplication*

Multiplication through logic gates is done by implementing 32-bit Adders and an AND gates 32 times. The output from the last gate is the Carry Out bit.

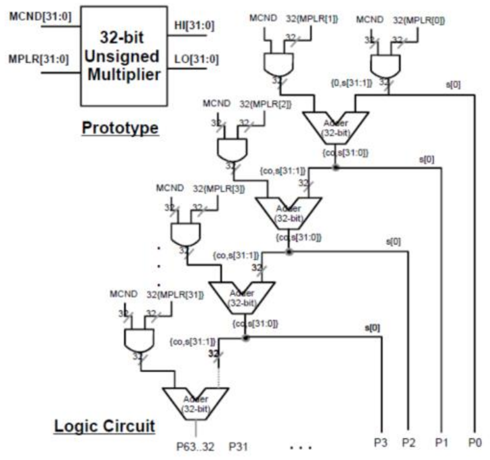


Figure 3.7. Unsigned Multiplication

By contrast, a signed multiplication circuit is the 2’s compliment of a 64-bit adder.

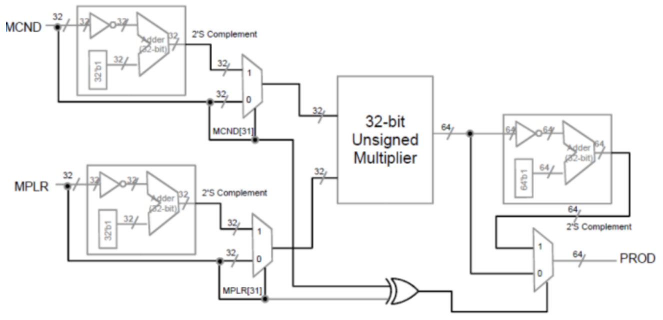


Figure 3.8. Unsigned Multiplication

F. *Shifting*

A barrel shifter is a circuit that shifts a number of bits with logic gates. It’s generally accomplished by using a sequence of multiplexers, where the output of one multiplexer is connected to the next. This was implemented in the program by making a x-bit shifter. The shifter has 32 = 2n where n = the number of control bits and rows. Since our shifter had 5 control bits with 32 rows, that's 160 multiplexers implemented in the program.

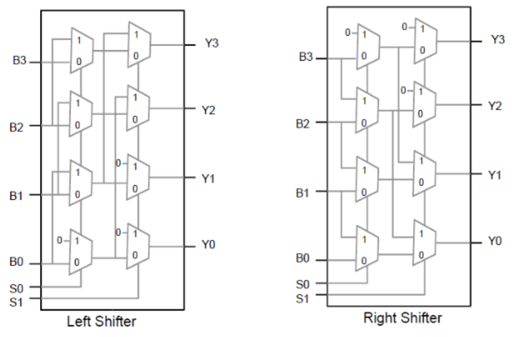


Figure 3.9. Sample Bit Shifter Adder

IV. Register File Design and Implementation

Since this project is an implementation of the DaVinci 1.0m processor, the register file is created using the gate model. The Reigster File can make use of two registers for reading and one other for writing, as well as preforming operations with three registers. The File also includes a reset function that will return it to the initial state.

This implementation of the Register file is made using decoders, logica gates, 32-bit resgisters and multiplexers. Figure 4.1 demonstartes what this looks like.

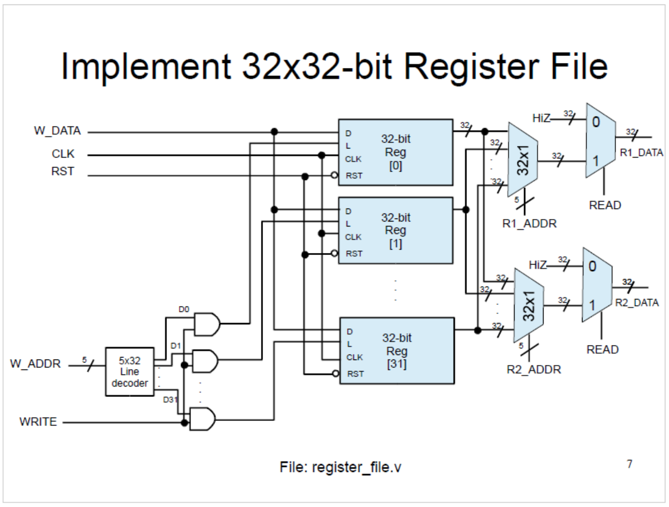


Figure 4.1. Register File Implementation

A. Decoder

The decoder is used to create an active wire based on the input to the wire. It's used in the program to select which registers will receive data from an address signal when combined with an AND gate with the second operand being the write signal. We utilized a 5x32-bit decoder in the program. It was created by using multiple 2x4-bit, 3x8-bit, and 4x16-bit decoders within its modeling.

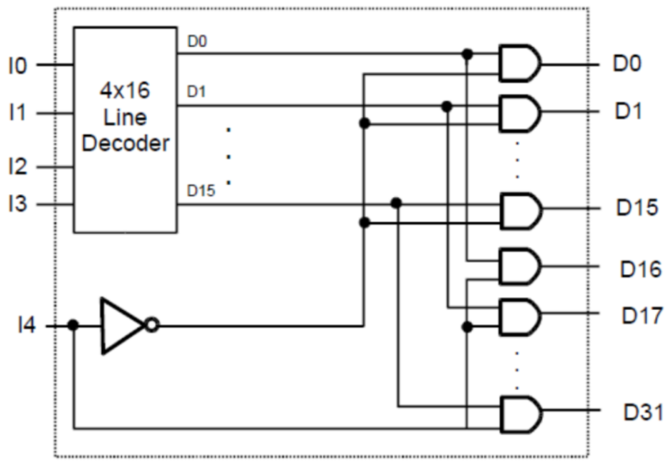


Figure 4.2. Decoder

B. Mutiplexer (Mux)

A multiplexer passes through an input based on the control signal. The most basic form is a 2x1 mux, where two inputs are sent to the mux, and the control signal selects one of them. In this project, we use a single-bit 32x1 and a 32-bit 2x1 mux. Like the decoder, multiple inputs to a multiplexer scale up recursively.

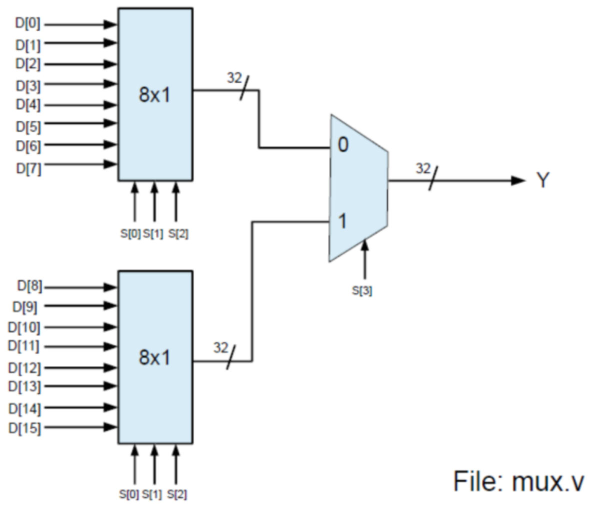


Figure 4.3. Multiplexer

C. 32-Bit Register

A register is the most basic unit of memory. The most basic unit that makes a register is the flip flop. Within a flip flop, we have a D-Latch and an SR-Latch. If we combine many flip-flops together and add them in one place, we have the register file.

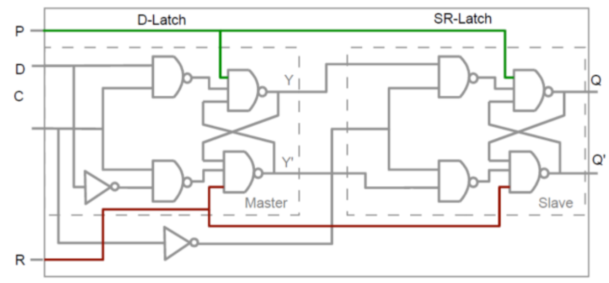


Figure 4.4. Flip-Flop with D/SR-Latches

32x32-bit register make up the Register file from a gate logic perspective. This uses the flip-flip in each register

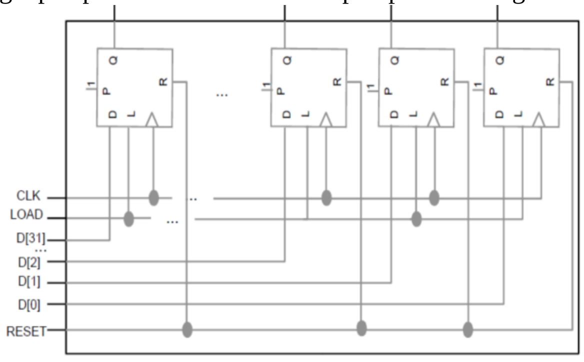


Figure 4.5. Completed Register File

D. Testing

Testing the register file was done using the 'register_file_tb.v' file included with the program three files. Running this produced the wavelengths in Figure 4.6 below this.

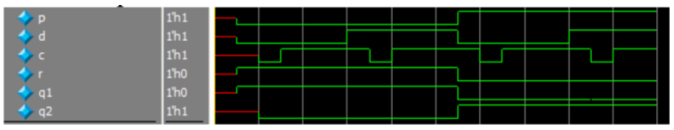


Figure 4.6. Register File Simulation Wavelengths

V. Memory Design and Implementation

There are two differnet components to the memory implemntation. The first part is taken from the Project II implementation of the memory, which is similar to the Register File from Project II, 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_OUT output port. When the opposite is true and READ is 0 and WRITE is 1, ADDR is inputted into the DATA_IN in. The DATA will remain in a High-Z state if READ and WRITE are both set to 0 or 1.

```
always @ (negedge RST or posedge CLK) begin
if (RST === 1'b0) begin
        for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
                sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
        $readmemh(mem_init_file, sram_32x64m);
end else begin
        if ((READ===1'b1)&&(WRITE===1'b0))
                data_ret =  sram_32x64m[ADDR];
        else if ((READ===1'b0)&&(WRITE===1'b1))
                sram_32x64m[ADDR] = DATA_IN;
        end
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.

The second component to the memory is the Memory Wrapper, which uses the same inputs and outputs as the 64M memory module, but only runs at the 'negedge' of the Clock for the 'RST'. After the reset, it set the DATA_OUT register to the DATA wire.

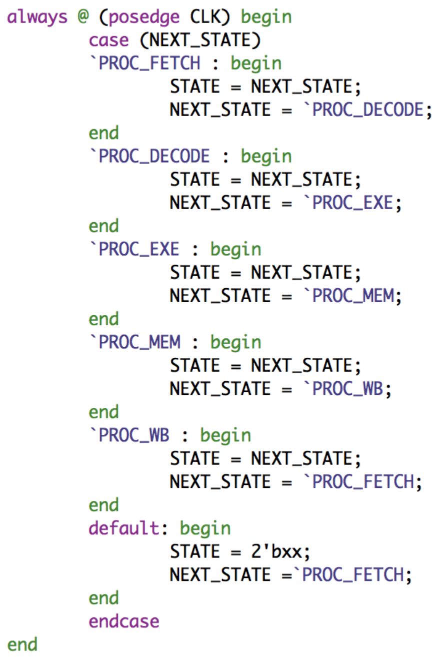
```
always @(negedge RST) begin
        if (RST === 1'b0)
                DATA_OUT = 32'h00000000;
end


always @(DATA) begin
if ((READ===1'b1)&&(WRITE===1'b0))
        DATA_OUT=DATA;
end
```

Figure 5.2. Memory Wrapper Reset / Read

VI. Processor, Implementation, and Testing

A. State Machine Control

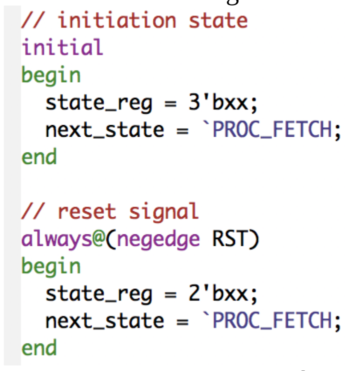
The Control Unit is designed to efficiently switch between the states of the processor. To do this, we will use a state machine. The States will be FETCH, DECODE, EXECUTE, MEMORY, and WRITE BACK. Each clock cycle, the process will go to the next one. When the state machine is initialized and when it's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



```
always @ (posedge CLK) begin
        case (NEXT_STATE)
        `PROC_FETCH : begin
                STATE = NEXT_STATE;
                NEXT_STATE = `PROC_DECODE;
        end
        `PROC_DECODE : begin
                STATE = NEXT_STATE;
                NEXT_STATE = `PROC_EXE;
        end
        `PROC_EXE : begin
                STATE = NEXT_STATE;
                NEXT_STATE = `PROC_MEM;
        end
        `PROC_MEM : begin
                STATE = NEXT_STATE;
                NEXT_STATE = `PROC_WB;
        end
        `PROC_WB : begin
                STATE = NEXT_STATE;
                NEXT_STATE = `PROC_FETCH;
        end
        default: begin
                STATE = 2'bxx;
                NEXT_STATE =`PROC_FETCH;
        end
        endcase
end
```

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.

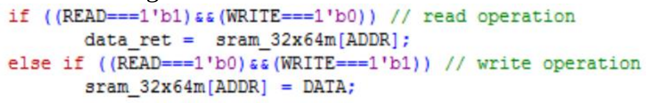


```
// initiation state
initial
begin
  state_reg = 3'bxx;
  next_state = `PROC_FETCH;
end

// 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.

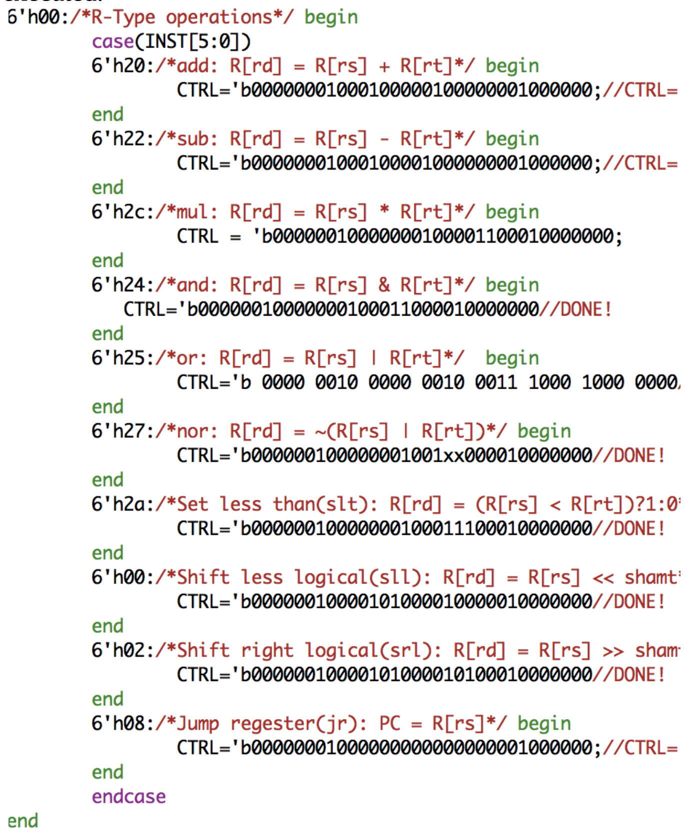


```
if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
```

Figure 6.3. State Machine Initialization / Reset

C. *R-Type Instruction Design*

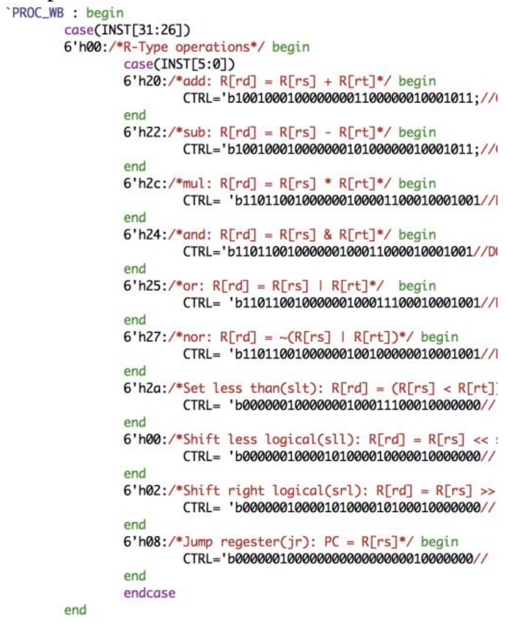
R-type instructions are the simplest to implement in the Control Unit, since all R-Type instructions have the same opcode, h'00. When this opcode is received by the CU, it then differentiates between an ALU function with two data ports set to OP1 and OP2, an ALU function with OP1 and a OP2 set to a shift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



```
6'h00:/*R-Type operations*/ begin
        case(INST[5:0])
        6'h20:/*add: R[rd] = R[rs] + R[rt]*/ begin
                CTRL='b00000001000100000100000001000000;//CTRL=
        end
        6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
                CTRL='b00000001000100001000000001000000;//CTRL=
        end
        6'h2c:/*mul: R[rd] = R[rs] * R[rt]*/ begin
                CTRL = 'b00000010000000010000110001000000;
        end
        6'h24:/*and: R[rd] = R[rs] & R[rt]*/ begin
           CTRL='b00000010000000010001100001000000//DONE!
        end
        6'h25:/*or: R[rd] = R[rs] | R[rt]*/  begin
                CTRL='b 0000 0010 0000 0010 0011 1000 1000 0000
        end
        6'h27:/*nor: R[rd] = ~(R[rs] | R[rt])*/ begin
                CTRL='b0000001000000001001xx000010000000//DONE!
        end
        6'h2a:/*Set less than(slt): R[rd] = (R[rs] < R[rt])?1:0
                CTRL='b00000010000000010001110001000000//DONE!
        end
        6'h00:/*Shift less logical(sll): R[rd] = R[rs] << shamt
                CTRL='b00000010000101000010000010000000//DONE!
        end
        6'h02:/*Shift right logical(srl): R[rd] = R[rs] >> sham
                CTRL='b00000010000101000010100010000000//DONE!
        end
        6'h08:/*Jump regester(jr): PC = R[rs]*/ begin
                CTRL='b00000001000000000000000001000000;//CTRL=
        end
        endcase
end
```

Figure 6.4. 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.

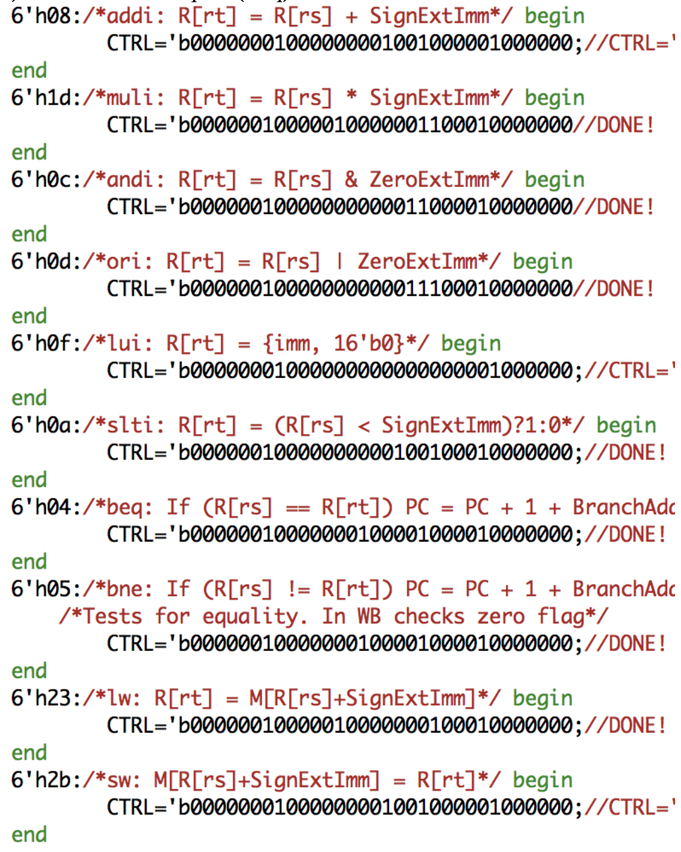


```
`PROC_WB : begin
        case(INST[31:26])
        6'h00:/*R-Type operations*/ begin
                case(INST[5:0])
                6'h20:/*add: R[rd] = R[rs] + R[rt]*/ begin
                        CTRL='b10010001000000001100000010001011;//
                end
                6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
                        CTRL='b10010001000000001010000010001011;//
                end
                6'h2c:/*mul: R[rd] = R[rs] * R[rt]*/ begin
                        CTRL= 'b11011001000000010000110001000100 1//
                end
                6'h24:/*and: R[rd] = R[rs] & R[rt]*/ begin
                        CTRL='b11011001000000010001100001000100 1//D
                end
                6'h25:/*or: R[rd] = R[rs] | R[rt]*/  begin
                        CTRL= 'b11011001000000010001110001000100 1//
                end
                6'h27:/*nor: R[rd] = ~(R[rs] | R[rt])*/ begin
                        CTRL= 'b11011001000000010010000001000100 1//
                end
                6'h2a:/*Set less than(slt): R[rd] = (R[rs] < R[rt]
                        CTRL= 'b00000010000000010001110001000000 0//
                end
                6'h00:/*Shift less logical(sll): R[rd] = R[rs] <<
                        CTRL= 'b00000010000101000010000010000000 0//
                end
                6'h02:/*Shift right logical(srl): R[rd] = R[rs] >>
                        CTRL= 'b00000010000101000010100010000000 0//
                end
                6'h08:/*Jump regester(jr): PC = R[rs]*/ begin
                        CTRL='b00000010000000000000000010000000//
                end
                endcase
        end
```

Figure 6.5. R-Type Write back

D. *I-Type Instruction Design*

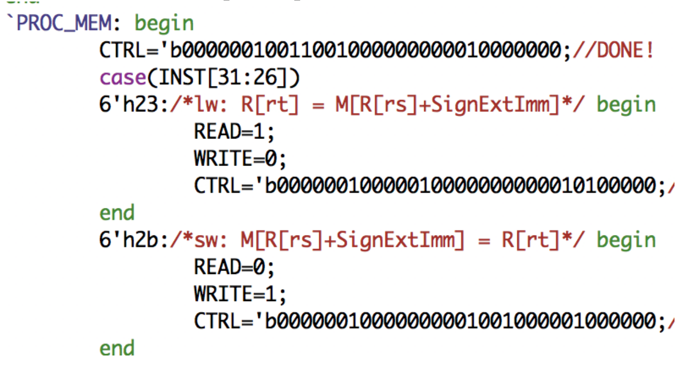
I-Type Instructions we implemented by using cases of the opcode provided to figure out which instruction needed to be executed. The I-Type instructions not included implementation of the Execution Procedure of the CU are the brach if not equal (bne) and branch if equal (beq).



```
    6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
            CTRL='b00000001000000000100100000100000;//CTRL='
    end
    6'h1d:/*muli: R[rt] = R[rs] * SignExtImm*/ begin
            CTRL='b00000001000001000000110001000000O//DONE!
    end
    6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
            CTRL='b00000001000000000001100001000000O//DONE!
    end
    6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
            CTRL='b00000001000000000001110001000000O//DONE!
    end
    6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
            CTRL='b00000001000000000000000001000000;//CTRL='
    end
    6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
            CTRL='b00000001000000000010010001000000O;//DONE!
    end
    6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchAd
            CTRL='b00000001000000010000100001000000O;//DONE!
    end
    6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchAd
        /*Tests for equality. In WB checks zero flag*/
            CTRL='b00000001000000010000100001000000O;//DONE!
    end
    6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
            CTRL='b00000001000001000000001000100000O;//DONE!
    end
    6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
            CTRL='b00000001000000000100100000100000;//CTRL='
    end
```

Figure 6.6. I-Type Instructions

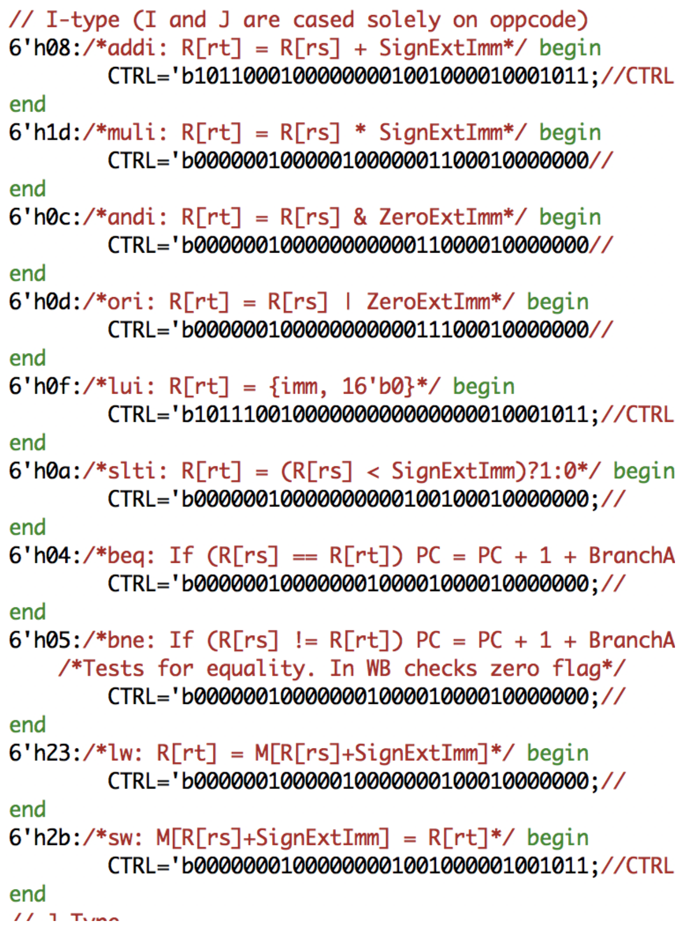
Since some I-Type instruction use an immediate in their functions, a ZeroExtended, a SignExtended, a LUI, and a BrachAddress were made. After execution, the instructions that control reading and writing to the memory, lw and sw, are implemented in the Memory procedural state of the CU. The correct instruction of the two above is determined by, once again, case tests of the opcode provided.



```
    `PROC_MEM: begin
            CTRL='b00000010011001000000000000100000000;//DONE!
            case(INST[31:26])
            6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
                    READ=1;
                    WRITE=0;
                    CTRL='b00000010000010000000000000101000000;/
            end
            6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
                    READ=0;
                    WRITE=1;
                    CTRL='b00000010000000000100100000100000;/
            end
```

Figure 6.7. Load Word / Store Word Instructions

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

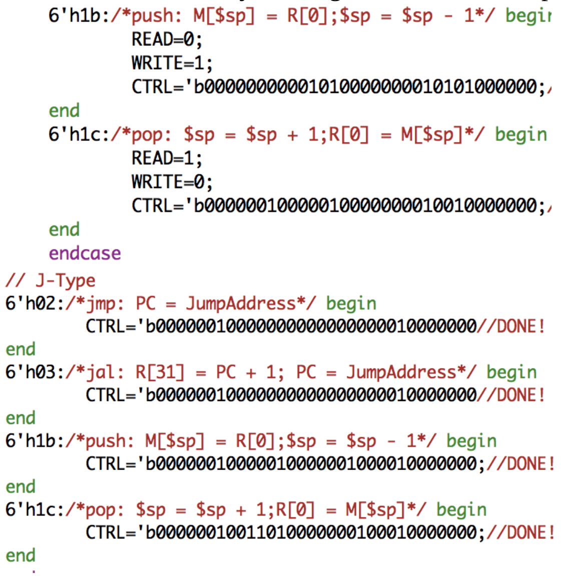


```
// I-type (I and J are cased solely on oppcode)
6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
        CTRL='b10110001000000000100100001000101 1;//CTRL
end
6'h1d:/*muli: R[rt] = R[rs] * SignExtImm*/ begin
        CTRL='b00000001000001000000110001000000O//
end
6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
        CTRL='b00000001000000000001100001000000O//
end
6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
        CTRL='b00000001000000000001110001000000O//
end
6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
        CTRL='b10111001000000000000000001000101 1;//CTRL
end
6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
        CTRL='b00000001000000000010010001000000O;//
end
6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchA
        CTRL='b00000001000000010000100001000000O;//
end
6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchA
    /*Tests for equality. In WB checks zero flag*/
        CTRL='b00000001000000010000100001000000O;//
end
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
        CTRL='b00000001000001000000001000100000O;//
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
        CTRL='b00000000100000000010010000001001011;//CTRL
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.



```
    6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begi
            READ=0;
            WRITE=1;
            CTRL='b0000000000010100000000010101000000;/
    end
    6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
            READ=1;
            WRITE=0;
            CTRL='b0000000100000100000000010010000000;/
    end
    endcase
// J-Type
6'h02:/*jmp: PC = JumpAddress*/ begin
        CTRL='b0000000100000000000000000010000000//DONE!
end
6'h03:/*jal: R[31] = PC + 1; PC = JumpAddress*/ begin
        CTRL='b0000000100000000000000000010000000//DONE!
end
6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
        CTRL='b0000000100000100000001000010000000;//DONE!
end
6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
        CTRL='b0000000100110100000000100010000000;//DONE!
end
```

Figure 6.9. J-Type Instructions

VII. CONCULSION

The DaVinci v1.0m system offers an in-depth model for creating a processor based up on the 'CS147DV' instruction set. This project differed from the processor from Project II by making usage of the logic gates, which was more challenging than a behavioral model. The system demonstrates the connections between the Processor, Memory, Register File, ALU, Control Unit, and logic gates. By building the main parts of the processor, we learned how to efficiently use the Verilog language to create a working microprocessor.