# Simulation of 32-Bit Processor at Gate Level

Azael Zamora
San Jose State University
azael.zamora12@gmail.com

***Abstract:* The objective of this paper is to cover DaVinci v1.0m simulator processor. The simulated processor is constructed utilizing the 'CS147DV' Instruction Set, a Control Unit, 32x64m memory module, register file, an Arithmetic Logic Unit, while utilizing logic gates to implement the processor. The following sections of the paper will cover each component individually, afterwards it will describe how they function together in order to implement the DaVinci processor system.**

## I. Introduction

For this project, the simulated processor utilizes the following components: ALU, memory models, Control Unit, and a Register File. The main objective of the DaVinci v1.0m system is to be able to create a simulated processor with memory in the Verilog hardware language. This project will demonstrate how the components required, will work together in the processor. Afterwards, to make sure that all the components were correctly implemented together, testing will be executed on the DaVinci system.

## II. The Requirements for the System

The following section will describe the components that are required in to properly implement that simulated DaVinci v1.0m processor.

### *A. The ALU (Arithmetic Logic Unit)*

The ALU is a crucial component for the implementation of the DaVinci processor. The following nine operations from the provided 'CS147DV' Instruction Set, will be executed logically by the ALU.

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

### *B. The Register File*

For the purpose of this project, the register file is comprised of 32 registers, each register capable of storing a 32-bit word. Throughout this project, the Reset operation will be executed when there is a negative edge on the clock for the processor. While the negative edge will be where the Reset operation will be executed, on the positive edge of the clock, both read and write operations will be executed instead. When either 11 or 00 is present in the simulation, both read and write will remain the HIGH_Z state, but will execute if either 10 or 01 Is present during the simulation.

### *C. The Memory*

For this project, the memory module is a 64M of memory that is capable of storing 32-bit word at each address. For this memory module, only the reset operation will be executed in the negative edge of the clock, while all other operations in the processor will only execute within the positive edge of the clock. For this implementation, DATA_R# will be set contain a 'don't care' or X value when either Read or Write at at 00 or 11.

### *D. The Control Unit (CU)*

The main purpose of the Control Unit is to act like a five-step state machine that is determined by five different stage changes. The following state changes of the clock will cycle throughout the positive edge of the clock: Fetch, Decode, Execute, Memory, and Write Back. Utilizing the instruction set 'CS147DV', the control unit will execute the decisions based on the instruction set.

### *E. The CS147DV Instruction Set*

The 'CS147DV' Instruction Set is an instruction set that is provided by Professor Kaushik Patra of San Jose State University which provides the following three instruction types: R-Type, J-Type, and I-Type. The Control Unit primarily utilizes the instructions in order to be implemented by the DaVinci processor system, while utilizing logic gates for the project.

## III. Implementing and Designing the Processor

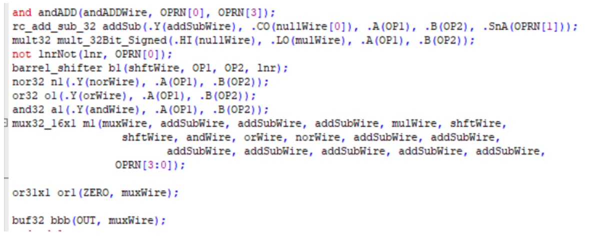
Utilizing the ModelSim simulator, it is relatively a simple process to program the needed logical operations for the ALU. The following section will provide the implementation of the Arithmetic Logic Unit using the Verilog Hardware Language.

### *A. The ALU Design*

The ALU design is very simple, since it's designed to take in two operators, and an operation and return the result as the output. For this project, the ALU that is built is a 32-bit processor, and each operand is 32-bit, the operand is 6-bit, and the output is 32-bit. All the nine operations that are utilized in this program are defined with 'ALU_WIDTH_OPRN'h0X where the X is used to define the specific operation. In the project, 'h03' is multiplication as an example, while the operants are called 'OP1' and 'OP2' respectively, and the output is referred to as 'OUT' which is the output in this case.

## B. *The Operations that the ALU handles*

For the purpose of this project, nine operations are handled by the ALU. The following operations will be shown along with their implementations.

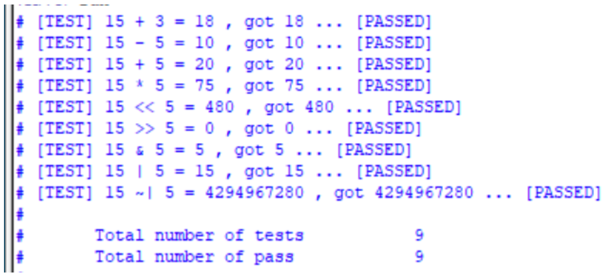


```
and andADD(andADDWire, OPRN[0], OPRN[3]);
rc_add_sub_32 addSub(.Y(addSubWire), .CO(nullWire[0]), .A(OP1), .B(OP2), .SnA(OPRN[1]));
mult32 mult_32Bit_Signed(.HI(nullWire), .LO(mulWire), .A(OP1), .B(OP2));
not lnrNot(lnr, OPRN[0]);
barrel_shifter bl(shftWire, OP1, OP2, lnr);
nor32 nl(.Y(norWire), .A(OP1), .B(OP2));
or32 ol(.Y(orWire), .A(OP1), .B(OP2));
and32 al(.Y(andWire), .A(OP1), .B(OP2));
mux32_16x1 ml(muxWire, addSubWire, addSubWire, addSubWire, mulWire, shftWire,
                shftWire, andWire, orWire, norWire, addSubWire, addSubWire,
                      addSubWire, addSubWire, addSubWire, addSubWire, addSubWire,
               OPRN[3:0]);

or31x1 orl(ZERO, muxWire);

buf32 bbb(OUT, muxWire);
```

***Implementing the ALU Operations***

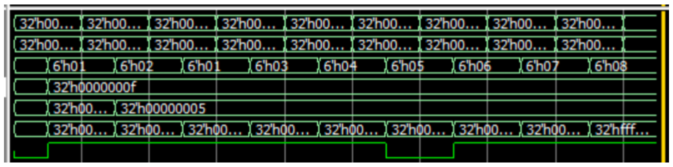
## C. *Testing of the implemented ALU*

Utilizing the bench test file 'alu_tb.v', we can make sure that the ALU was properly implemented for this project. The test file will provide a test case for each operation that needs to be correctly implemented. The following images displays the wavelength and text results of the operations' test.



```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 15 * 5 = 75 , got 75 ... [PASSED]
# [TEST] 15 << 5 = 480 , got 480 ... [PASSED]
# [TEST] 15 >> 5 = 0 , got 0 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 5 = 4294967280 , got 4294967280 ... [PASSED]
#
#       Total number of tests          9
#       Total number of pass           9
```

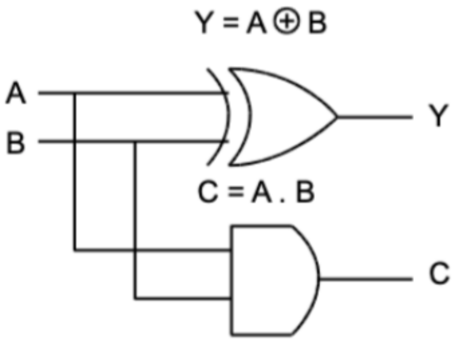
***ALU Text Pass***



***ALU Wavelength simulation***

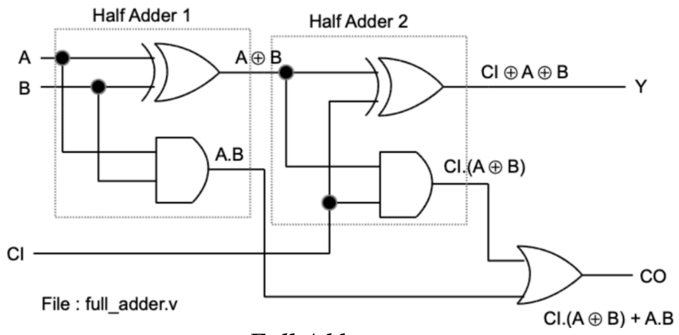
## D. *Implementing the Adder/Subtracter at the gate level*

In order to accomplish addition with bits, it is done through the usage of a half adder. For this project, the half adder is implemented at the logic level utilizing an AND gate and an XOR gate in order to get the output. The following image provides the implementation of the half adder using logic gates.



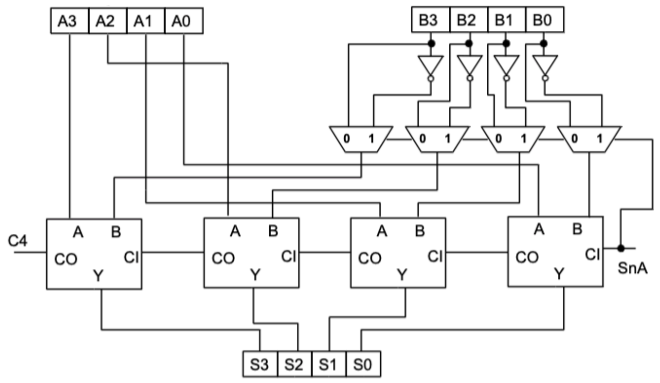
***Half Adder***

When implementing two half adders with the addition of an OR gate to correctly select the CO (Carry Out) bit results in a full adder. For the scope of this project, 32 full adders will need to be implemented with the other required components. The diagram below shows how the a 1-bit full adder is implemented at the gate level.



***Full Adder***

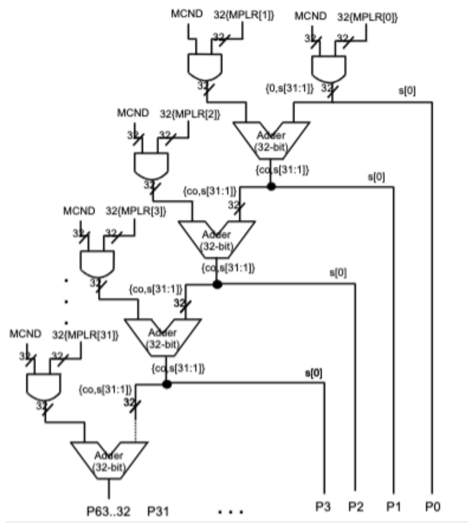
In order to to make the full adder circuit be compatible to handle subtraction, 2's compliment on the second bit input will need to implemented in order to simulate the process of subtraction.



***Ripple Carry Adder-Subtractor***

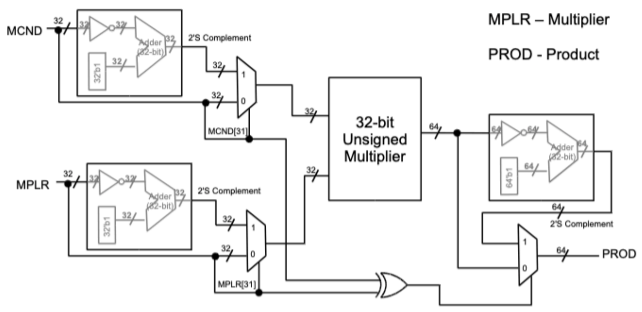
### E. *Implementation of multiplication at the gate level*

The unsigned multiplication operation at the gate level is accomplished by implementing 32-bit Full adders and also implementing AND gates 32 times. The final gate from the output is the CO (Carry Out) bit.



***Unsigned Multiplication***

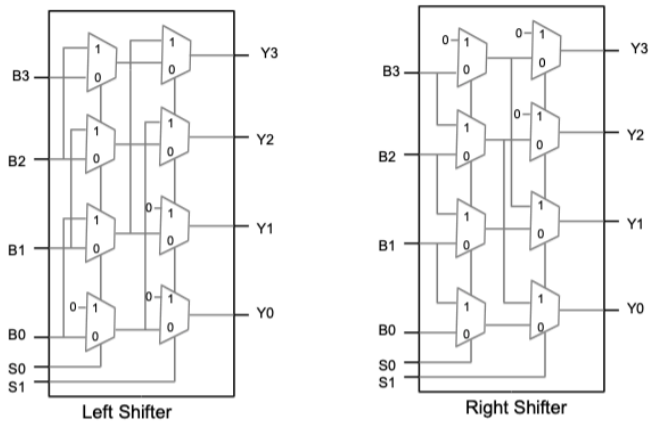
Similarly, the signed multiplication circuit implements 2's compliment on a 64-bit adder in order to allow positive and negative numbers.



***Signed Multiplication***

### F. *Shifting*

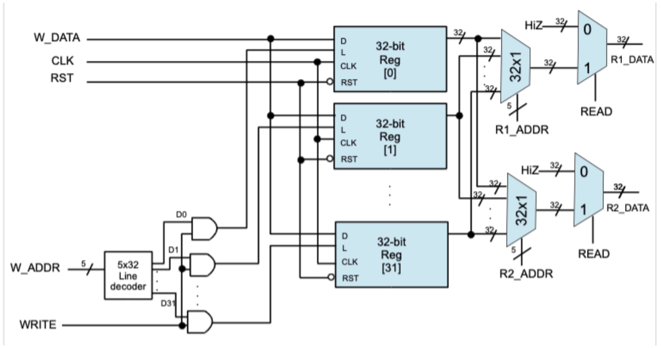
For this project, a barrel shifter was required which is a circuit capable of shifting a certain number of bits while utilizing logic gates. Ideally the function of barrel shifter is accomplished by utilizing multiplexers in which the output of a multiplexer is connected to the next multiplexer. For this project, a barrel shifter was implemented by making it be able to shift n-bits. For the simulated processor, the shifter has 5 control bits, which results in 32 rows, meaning that a total of 160 multiplexers are implemented for the project. The image below shows two types of shifters that are referred to as 'Left shifter' and 'Right shifter'.



***Shifter Adder***

## IV. Implementing and designing the Register File

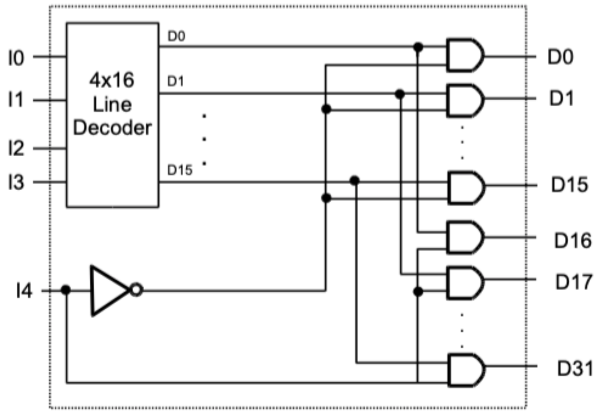
Since a simulated processor was implemented for this project, a register file must also be implemented at the gate level. The register file that is required for the processor utilizes two registers for reading, while it only utilizes a register for writing, and also capable of using three other registers for operations. The register file also implemented reset functionality into to return the register file back to its initial state. To implement the register file, it is comprised of decoders, 32-bit registers, multiplexers, and also logical gates.



***Implementing the Register File***

### A. *Decoder*

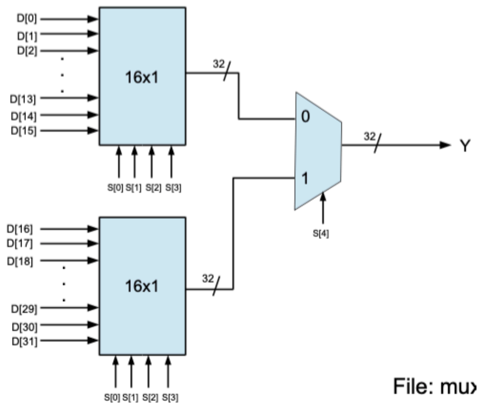
A decoder is utilized to make an active wire determined by the input to the wire. The decoder is used by the register file to determine which register will receive data from the address signal when joined with an AND gate, with the second operand of the decoder being the write signal. For this project we utilized a 5x32-bit decoder that was created by a multitude of 4x16-bit, 3x8-bit, 2x4-bit decoders in the modeling design of the decoder.



***Decoder***

## B. Multiplexer

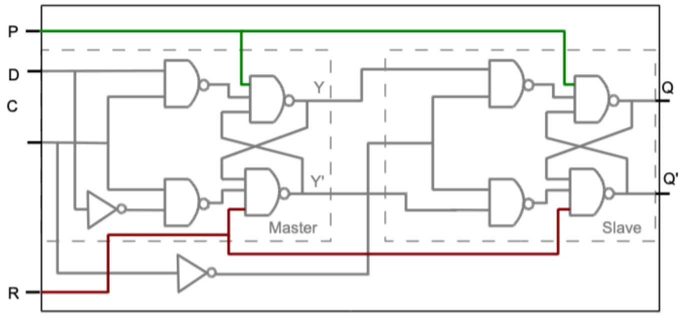
Multiplexer is a circuit that allows an input to pass, based on the control signal which determines which input goes through. The most simple form of a multiplexer is a 2x1 in which two inputs are received by the multiplexer and the circuit must then select one of the inputs. For the purpose of the project, a 1-bit 32x1 and 32-bit 2x1 multiplexers were implemented for this project. Similarly to a decoder, multiple inputs to a multiplexer will scale it up recursively.



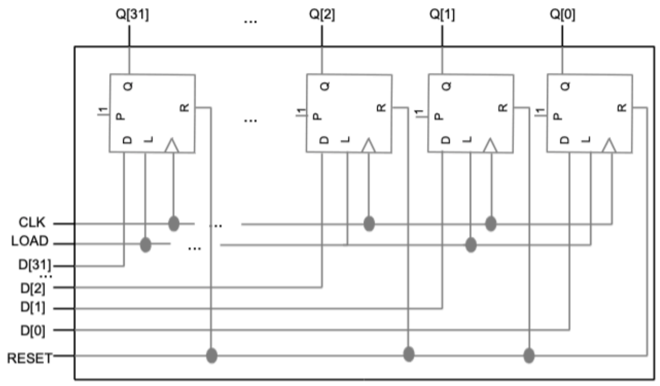
***Multiplexer***

## C. The 32-Bit Register

The simplest unit of memory is a register. The component that primarily makes up a register is called a flip flop component. In a flip flop, the component contains two kinds of latches, D-Latch and an SR-Latch. Combining a bunch of flip flops in one place, results in having the register file.



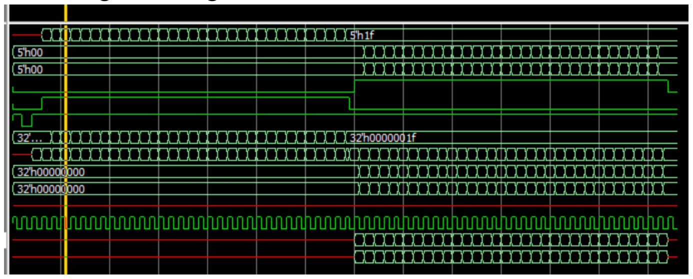
***Flip-Flop with both D and SR-Latches***



***Fully implemented Register File***

## D. Testing

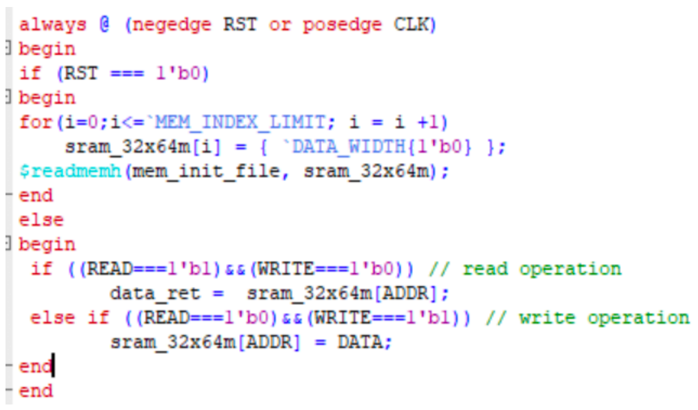
The implemented register file was tested utilizing the file 'register_file_tb.v' along with three other files provided for this project. The test file produced the following wavelengths in the simulation.



*SIMULATION OF THE REGISTER FILE WAVELENGTH*

## V. DESIGNING AND IMPLEMENTING THE MEMORY

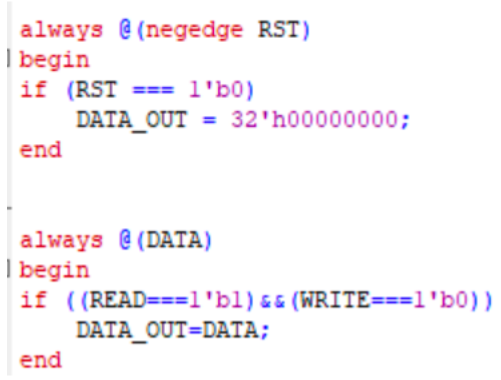
Like the Register File module, memory module has both input and output data port that is named 'DATA'. In the module, READ operation is set to 1 which enables the reading from the memory. The WRITE operation is set to 0, while ADDR and data address are given for this project, and are used by the DATA_OUT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



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

***Memory Operations of Read and Write***

Furthermore for the implementation of the memory module, the Reset stage will only occur when a negative edge of RST is present, and then the RST will be set to ‘1’b0’. Once the Reset Stage occurs, all the memory data will then change to 0, and the initialization memory file called ‘mem_init_file’ will be read instead. The rest of the stages from the processor memory will only happen when Clock has its positive edge present. The other component of the memory is called ‘Memory Wrapper’ which utilizes the same inputs/outputs from the 64M memory module, but with the exception that it only functions on the ‘negedge’ of the clock for the ‘RST’. Once reset occurs, the DATA_OUT register is set to DATA wire.



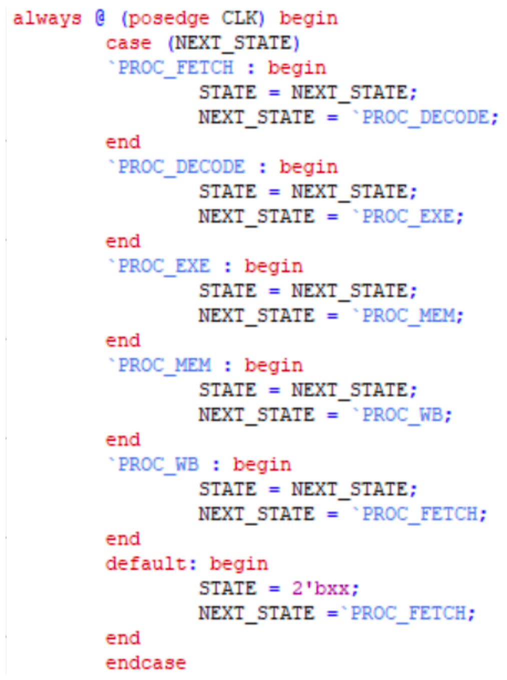
```
always @(negedge RST)
begin
if (RST === 1'b0)
    DATA_OUT = 32'h00000000;
end

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

***Reset and Read for Memory Wrapper***

## VI. IMPLEMENTING, TESTING OF THE PROCESSOR

### *A. The State Machine Control*

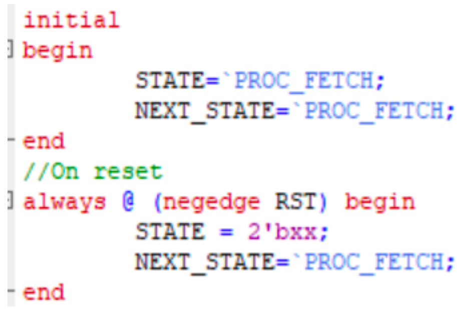
For the simulated processor, the Control Unit is implemented so that it can switch to the other different states in the processor. In order to accomplish this, a state machine will need to be utilized for this purpose. The five distinct states for the processor will be Fetch, Decode, Execute, Memory, and Write Back, as the Control Unit will be rotating between them. As a clock cycle passes, the state operator will switch to the next state. If the state operator is initialized to ‘reset’ , the current state register that is called ‘state_reg’, is set to 3’bxx, while ‘next_state’ register is set to the next corresponding clock state.



```
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
```

***Implementing Next State Machine***

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

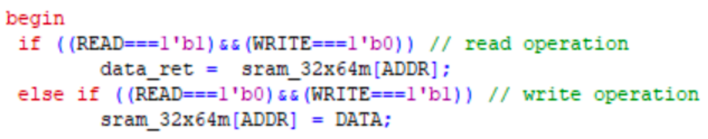


```
initial
begin
        STATE=`PROC_FETCH;
        NEXT_STATE=`PROC_FETCH;
end
//On reset
always @ (negedge RST) begin
        STATE = 2'bxx;
        NEXT_STATE=`PROC_FETCH;
end
```

**Start/Reset of State Machine**

### *B. Testing the State Machine*

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

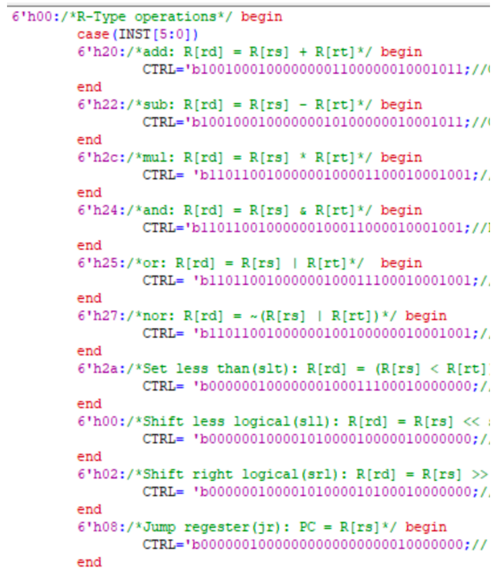


```
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
```

**Start and Reset of the State Machine**

## *C. Implementing the R-Type Instruction*

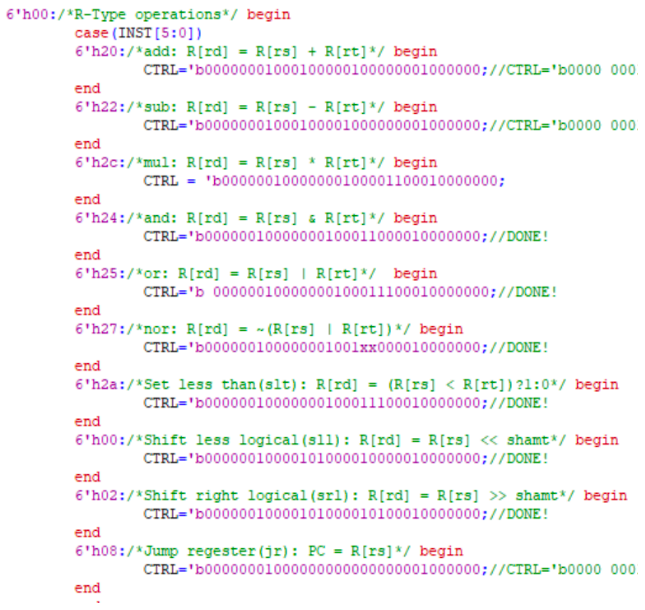
The R-type instructions are relatively easy instructions to implement, since R-types utilize a opcode of value h'00. While the Control Unit receives the opcode, the CU will then proceed to implement the correct ALU function using the two data ports which will be set to OP1 and OP2. The Control Unit will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



```
6'h00:/*R-Type operations*/ begin
        case(INST[5:0])
        6'h20:/*add: R[rd] = R[rs] + R[rt]*/ begin
                CTRL='b10010001000000000110000000100010 11;//
        end
        6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
                CTRL='b10010001000000001010000000100010 11;//
        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='b11011001000000010001100000100010 01;//
        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= 'b11011001000000010010000000100010 01;/
        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= 'b00000010000101000010000001000000 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
```

***Implementation of R-Type Instructions***

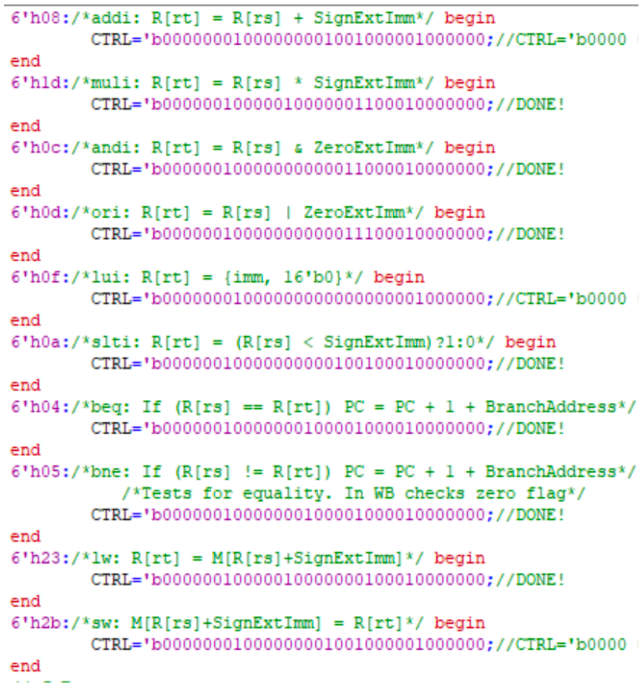
During WriteBack stage, R-Type instructions are called again in order to write in the needed address.



```
6'h00:/*R-Type operations*/ begin
        case(INST[5:0])
        6'h20:/*add: R[rd] = R[rs] + R[rt]*/ begin
                CTRL='b00000001000100000100000001000000;//CTRL='b0000 000
        end
        6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
                CTRL='b00000001000100001000000001000000;//CTRL='b0000 000
        end
        6'h2c:/*mul: R[rd] = R[rs] * R[rt]*/ begin
                CTRL = 'b00000010000000010000110001000000 0;
        end
        6'h24:/*and: R[rd] = R[rs] & R[rt]*/ begin
                CTRL='b00000010000000010001100000100000 00;//DONE!
        end
        6'h25:/*or: R[rd] = R[rs] | R[rt]*/  begin
                CTRL='b 00000010000000010001110001000000 0;//DONE!
        end
        6'h27:/*nor: R[rd] = ~(R[rs] | R[rt])*/ begin
                CTRL='b00000010000000010 01xx000010000000;//DONE!
        end
        6'h2a:/*Set less than(slt): R[rd] = (R[rs] < R[rt])?1:0*/ begin
                CTRL='b00000010000000010001110001000000 0;//DONE!
        end
        6'h00:/*Shift less logical(sll): R[rd] = R[rs] << shamt*/ begin
                CTRL='b00000010000101000010000001000000 0;//DONE!
        end
        6'h02:/*Shift right logical(srl): R[rd] = R[rs] >> shamt*/ begin
                CTRL='b00000010000101000010100010000000 0;//DONE!
        end
        6'h08:/*Jump regester(jr): PC = R[rs]*/ begin
                CTRL='b00000001000000000000000001000000;//CTRL='b0000 000
        end
```

## *D. Implementing the I-Type Instruction*

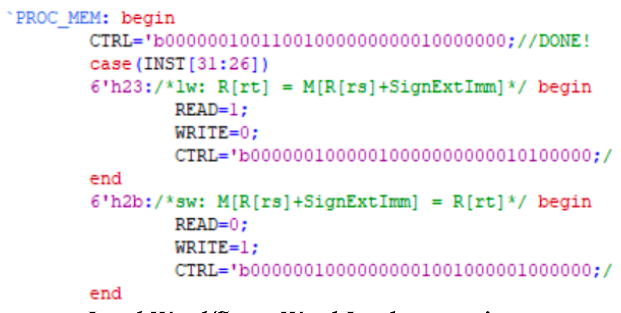
Unlike R-Type instructions, which only use opcode h'00, I-Type instructions that are implemented use variety of opcodes. In order to determine which instruction would be called, cases on opcode were utilized to determine which I-Type instruction is used. As shown in the image below, the following instructions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



```
6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
        CTRL='b00000001000000000100100000010000 00;//CTRL='b0000
end
6'h1d:/*muli: R[rt] = R[rs] * SignExtImm*/ begin
        CTRL='b00000010000010000000110001000000 0;//DONE!
end
6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
        CTRL='b00000010000000000001100001000000 0;//DONE!
end
6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
        CTRL='b00000010000000000001110001000000 0;//DONE!
end
6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
        CTRL='b00000001000000000000000001000000;//CTRL='b0000
end
6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
        CTRL='b00000010000000000010010001000000 0;//DONE!
end
6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchAddress*/
        CTRL='b00000010000000010000100001000000 0;//DONE!
end
6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress*/
            /*Tests for equality. In WB checks zero flag*/
        CTRL='b00000010000000010000100001000000 0;//DONE!
end
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
        CTRL='b00000010000010000000100010000000 0;//DONE!
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
        CTRL='b00000001000000000100100000010000 00;//CTRL='b0000
end
```

***Implementation of I-Type Instructions***

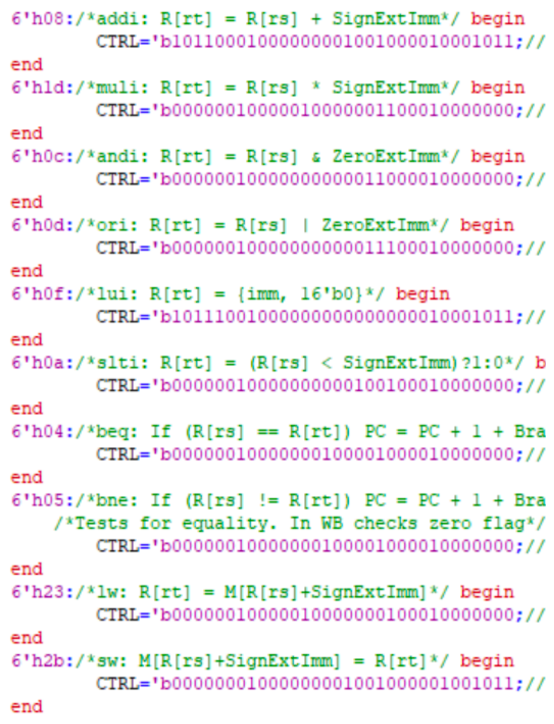
Using the instruction set 'CS147DV' for I-Type instructions, some of the instructions use either ZeroExtended, SignExtended, LUI, or a BranchAddress, as their immediate portion of their functions. The image below shows how each of the immediate were implemented.



```
`PROC_MEM: begin
        CTRL='b00000010011001000000000000100000 00;//DONE!
        case(INST[31:26])
        6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
                READ=1;
                WRITE=0;
                CTRL='b00000010000010000000000000101000 00;/
        end
        6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
                READ=0;
                WRITE=1;
                CTRL='b00000010000000000100100000010000 00;/
        end
```

***Load Word/Store Word Implementation***

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

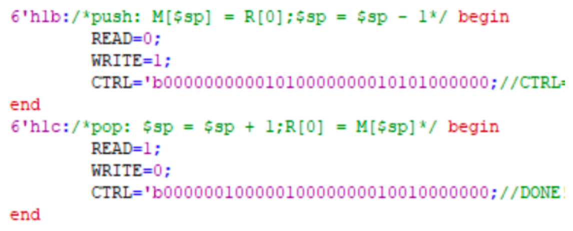


```
6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
        CTRL='b10110001000000001001000010001011;//
end
6'h1d:/*muli: R[rt] = R[rs] * SignExtImm*/ begin
        CTRL='b00000010000010000001100010000000;//
end
6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
        CTRL='b00000010000000000001100001000000;//
end
6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
        CTRL='b00000010000000000001110001000000;//
end
6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
        CTRL='b10111001000000000000000010001011;//
end
6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ b
        CTRL='b00000010000000000010010001000000;//
end
6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + Bra
        CTRL='b00000010000000100001000010000000;//
end
6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + Bra
    /*Tests for equality. In WB checks zero flag*/
        CTRL='b00000010000000100001000010000000;//
end
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
        CTRL='b00000010000010000000100010000000;//
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
        CTRL='b00000001000000001001000001001011;//
end
```

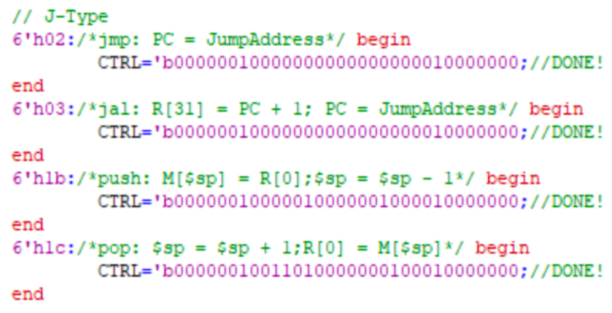
**I-Type Instruction WriteBack**

### E. *Implementing J-Type Instruction*

The push operation, one of the four J-Type operations, is only executed in the Execution procedure state of the Control Unit that is designed for this project. The other three J-Type instructions will be executed during Write Back state unlike the push operation. For this project, J-Type design was is implemented through use of running cases on the opcode by the control unit, as shown in the images below since the DATA_R1 address is set to register 0 by default.



```
6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
        READ=0;
        WRITE=1;
        CTRL='b00000000000101000000000101010000000;//CTRL=
end
6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
        READ=1;
        WRITE=0;
        CTRL='b00000010000010000000001001000000;//DONE
end
```



```
// J-Type
6'h02:/*jmp: PC = JumpAddress*/ begin
        CTRL='b00000010000000000000000010000000;//DONE!
end
6'h03:/*jal: R[31] = PC + 1; PC = JumpAddress*/ begin
        CTRL='b00000010000000000000000010000000;//DONE!
end
6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
        CTRL='b00000010000010000000100001000000;//DONE!
end
6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
        CTRL='b00000010011010000000100010000000;//DONE!
end
```

***J-Type Instruction Implementation***

## VII. Conclusion

To summarize, the DaVinci v1.0m processor system allowed insight in how to implement a simulated processor utilizing the 'CS147DV' instruction set that was provided. The simulated system showed how the required components: Processor, ALU, Memory, Control Unit, Register File, and logic units came together in order to properly implement the system. Overall, the project allowed me to utilize the Verilog Hardware Language to properly implement a microprocessor.