

32-Bit GLM Processor Simulation Model

Xinken Zheng
CS 147
zhengxinken@gmail.com

Abstract— The purpose of this paper is to demonstrate the steps of the creation of DaVinci v1.0m processor and its simulation. This project is built using 'CS147DV' Instruction set, Register file, ALU, Memory module, ALU, and Control Unit. The report below will show every aspects of the implementation and design independently and how they act together for the DaVinci system.

I. INTRODUCTION

The DaVinci system will be a simple example of a real world processor and memory. The project is written in Verilog language. The components included in the DaVinci system are ALU, Memory Module and Control Unit in no specific order. I will be demonstrating the implantation design of each of the parts and test the system.

II. SYSTEM REQUIREMENTS

A. Arithmetic Logic Unit (ALU)

The main part of the DaVinci v1.0m system, it will handle a total of nine operations from the instruction set provided from Professor Patra of SJSU. The nine operations are:

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

B. Register File

There are a total of 32 registers in the DaVinci System.[0-31], each of the register can store a 32-bit word. The register will do different things depending on the state of the clock. When the clock is on the negative edge, reset operations will be executed. While read and write operations will be executed on the positive edge.

C. Memory

There are a total of 64M of memory in this system, this is from adding all 32-bit word at each of the address. When the clock is on negative edge, reset operations are executed on the negative edge of the clock while all of the other operations will be executed when the clock is on positive edge. Data_R# is set to X when read/write are 00 or 11.

D. Control Unit (CU)

The state machine will be the one controlling what fuctions to operate in the Control Unit. There are a total of 5 states in the state machine, INFORMATION FETCH, DECODE, EXECUTION, MEMORY, and WRITE BACK. The states will change when the clock is on the positive edge and cycle between these five states. The control unit of this system will perform according to the Instruction set provided for this project.

III. THE PROCESSOR DESIGN AND IMPLEMENTATION

A. ALU Design

The 32 bit ALU design stays almost the same like the first two projects we did for this semester. It will take in two inputs and one operation depending on the operation code, the calculation then is done and the result will be store in result variable. The operation code depends on the width in

'ALU_OPRN_WIDTH'h(#)

The number in the end will determine the kind of function to be executed. For example, h1 will be for addition and h2 will be for subtraction and so on. Since the ALU for this project will be a 32-bit processor, there will be a 32-bit operand and the output will be 32bit. The output will be stored in OUT.

B. ALU Operations

The ALU will have a total of nine operations, OP1 is operand 1 and OP2 is operand 2, the result after computing is stored in 'OUT'.

```

module ALU(OUT, ZERO, OP1, OP2, OPRN);
// input list
input [DATA_INDEX_LIMIT:0] OP1; // operand 1
input [DATA_INDEX_LIMIT:0] OP2; // operand 2
input [ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

// output list
output [DATA_INDEX_LIMIT:0] OUT; // result of the operation.
output ZERO;

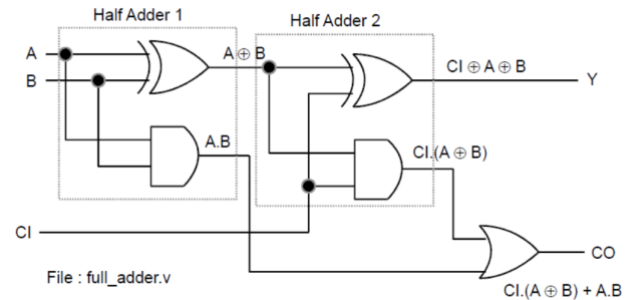
wire [31:0] add_Subwire, shift_wire, mult_wire, and_wire, or_wire, nor_wire, OUT, co;
wire oprnNot, oprnOr, oprnAnd, slt;

and and_1(oprnAnd, OPRN[0], OPRN[3]);
not not_1(oprnNot, OPRN[0]);
or or_1(oprnOr, oprnNot, oprnAnd);

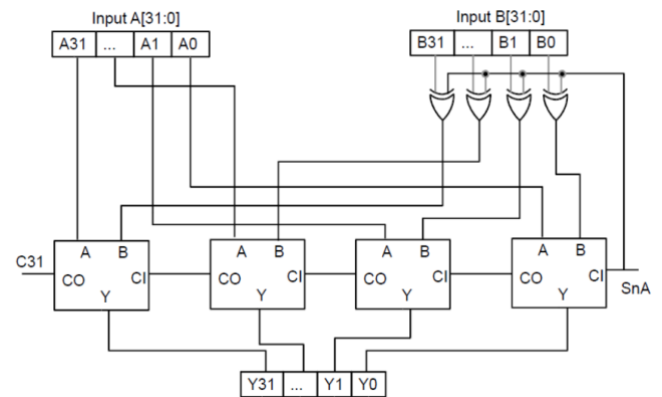
PC_ADD_SUB_32 addsub_1(.Y(add_Subwire), .CO(co[0]), .A(OP1), .B(OP2), .SnA(oprnOr));
buf buf_1(slt, add_Subwire[31]);
MULT32 mult_1(.HI(co), .LO(mult_wire), .A(OP1), .B(OP2));
SHIFT32 shift_1(.Y(shift_wire), .D(OP1), .S(OP2), .LnR(OPRN[0]));
AND32_2x1 and_2(.Y(and_wire), .A(OP1), .B(OP2));
NOR32_2x1 nor_1(.Y(nor_wire), .A(OP1), .B(OP2));
OR32_2x1 or_2(.Y(or_wire), .A(OP1), .B(OP2));

MUX32_16x1 mux_1(.Y(OUT), .I0(add_Subwire), .I1(add_Subwire), .I2(mult_wire),
.I4(shift_wire), .I5(shift_wire), .I6(and_1), .I7(or_wire), .I8(nor_wire),
.I9(slt), .I10(add_Subwire), .I11(add_Subwire), .I12(add_Subwire),
.I13(add_Subwire), .I14(add_Subwire), .I15(add_Subwire), .S(OPRN[3:0]));
OR32x1 or_3(.Y(ZERO), .A(OUT));
endmodule

```

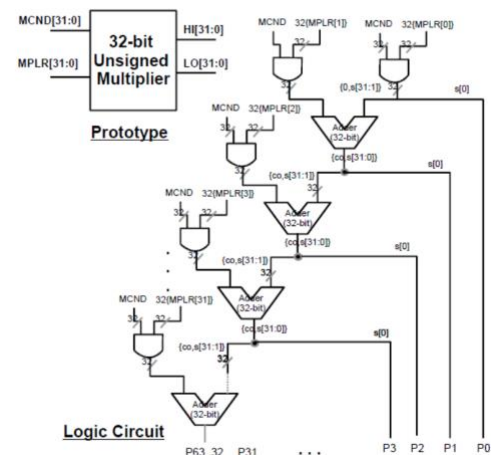


In order for subtraction to work with the circuit, a 2's complement is needed in the second input bit.



E. Multiplication(*)

For multiplication, it is done through the logic gates with a 32-bit adder and an and gate 32 times. The carry out bit is the output of the last gate.



On the other end, circuits for signed multiplication is a 64-bit adder 2's complement.

C. ALU Testing

To ensure the unit is working correctly, we will test bench it with some tests using the testbench file we implemented. Wavelength graph is also available below

```

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
#

```

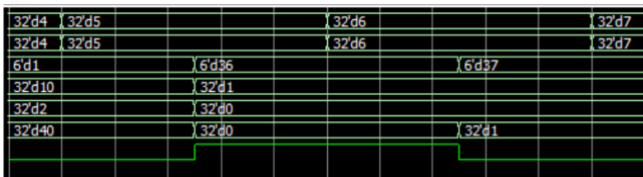
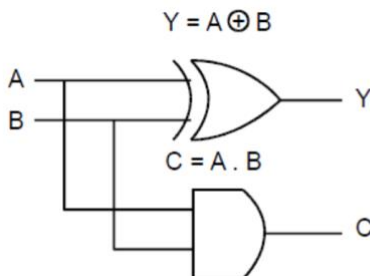
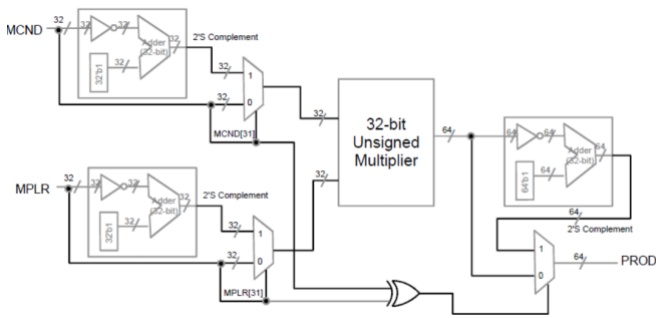


Figure 3.3. Sample ALU Simulation Wavelength

D. Adder(+) and Subtractor(-) design

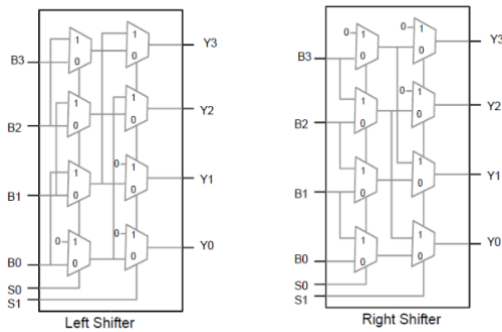
Bitwise addition will be done with a half adder, which will use XOR gate for implementation. The output will be computed using an AND gate. Then for the full adder, we use two half adders with the addition of an OR gate and the correct carry out bit.





F. Barrel Shifter

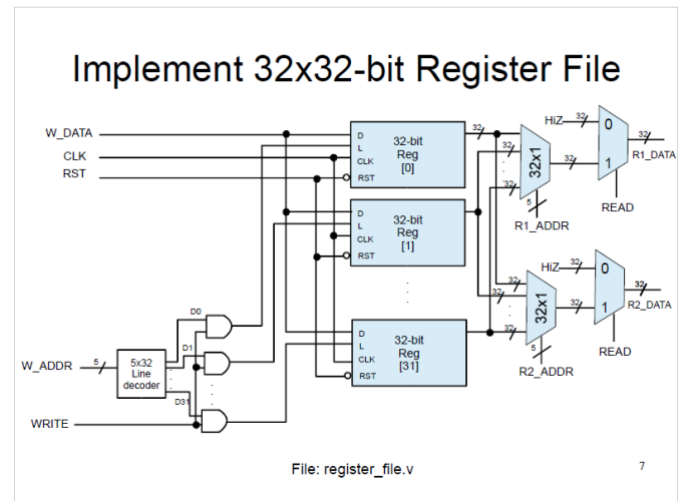
A barrel shifter shifts a word a number of bits in a circuit with logic gates. This is done by using a series of multiplexers. The output of multiplexers is connected one by one. In our program, it was implemented by making a x-bit shifter. The shifter contains $32 = 2n$, as n is the number of control bits. In our case, we have 5 control bits with 32 row. Which mean we have a total of 160 multiplexers implemented in the program.



IV. REGISTER FILE DESIGN AND IMPLEMENTATION

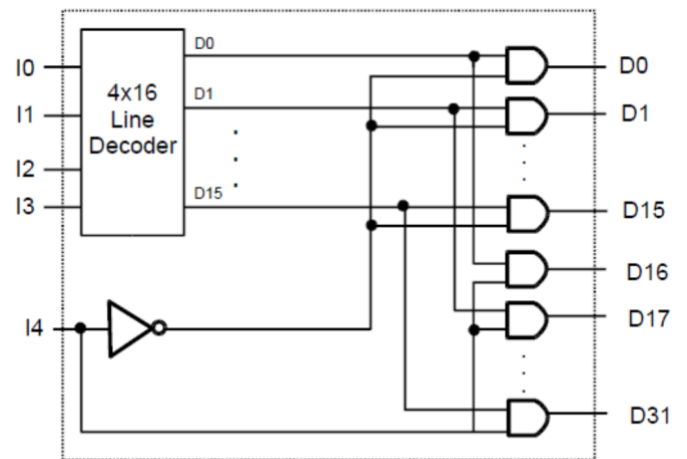
The register file for this project have the ability to allocate two registers for reading and one for writing, as well as three for operations. The register file also has a reset function that return to initial state.

For implementing, we used decoders, logic gates, 32-bit register and multiplexers. The diagram below shows what it is like.



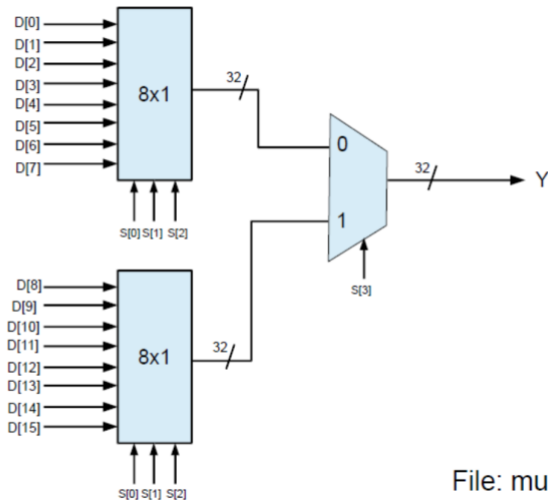
A. Decoder

A decoder is needed for the input of the wire. This is used to choose what register will receive data from address signal, when combined with the second operand with the AND gate being the write signal. We created a 5x32 bit decoder for the program which was implemented using a couple 2x4 bit, 3x8 bit, and 4x16 bit decoders.



B. Multiplexer (Mux)

A multiplexer is a data selector, it selected between several input signals and forward to a single output line. The most basic form of a multiplexer is a 2x1 mux. For our system, we will have a 1-bit 32x1 and 32-bit 2x1 multiplexer. The diagram below shows the diagram for a multiplexer.

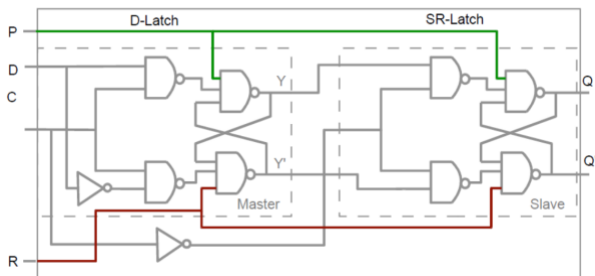


File: mux.v

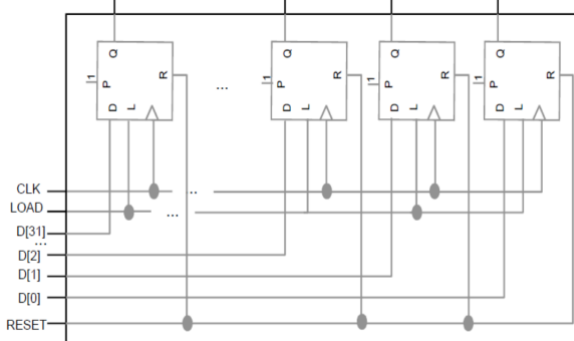
Figure 4.3. Multiplexer

C. 32-Bit Register

A flip flop latch is a circuit that has two stable states and can be used to store state information. It is the least requirement for a register file, and register is the basic unit in memory. Inside a flip-flop, we have a SR-latch and D-Latch. SR latch is made from two NAND gates, D-Latch is used to capture. A register file is made by concluding many flip flop latch together. The diagram below shows flip flop with D/SR latches.

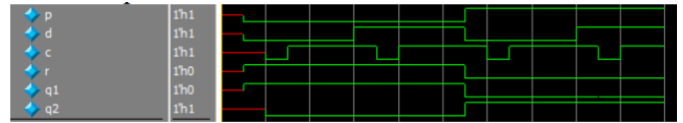


32x32-bit register make up the Register file from a gate logic perspective. This uses the flip-flop in each register



D. Testing

After implementing the register file, we will test it with the test bench file 'register_file_tb.v', this was given by the project starter code and below is the wavelength graph.



V. MEMORY DESIGN AND IMPLEMENTATION

The design and implementation of the memory will have two different parts. The first part was similar to what we did in project 2 on the register file. However, for this time, we will only have one input output port 'DATA'. Read is set to one when reading from memory and write is set to 0. ADDR is the data address and will be given and output the result at DATA_OUT output port. On the other end, when read is 0 and write is set to 1, DATA_IN is inputted with 'ADDR'. DATA stays at HIGH-Z state when read and write are both the same, 0 or 1. Below is the implementation.

```
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
end
```

Reset stage happens when RST is on negative edge and is set to '1b'0'. Data is then changed to 0 after reset and data is being read from file 'mem_init_file'. Any other processes in the memory will only operate when the clock is on positive edge

The second part of the memory we will be designing is the wrapper. The wrapper uses the same input and output with the memory module that is 64M. Under the condition that it only run at the negative edge of the clock when resetting. Data wire is set from data_out register.

```
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
```



```
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= 'b00000010000000100001100010000000;
end
6'h24:/**and: R[rd] = R[rs] & R[rt]*/ begin
CTRL='b0000000100000000100011000010000000//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='b0000000100000001000111000100000000//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] >> shamt
CTRL='b0000001000010100001010000100000000//DONE!
end
6'h08:/*Jump register(jr): PC = R[rs]*/ begin
CTRL='b0000000100000000000000000000000000//CTRL=
end
endcase
```

R-Typed instructions will need to be called again when the state machine is in the WRITE BACK STATE.

```

`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='b1001000100000001100000010001011;//A
end
6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
CTRL='b1001000100000001010000010001011;//A
end
6'h2c:/*mul: R[rd] = R[rs] * R[rt]*/ begin
CTRL='b110110010000000100001100010001001//D
end
6'h24:/*and: R[rd] = R[rs] & R[rt]*/ begin
CTRL='b110110010000000100011000010001001//D
end
6'h25:/*or: R[rd] = R[rs] | R[rt]*/ begin
CTRL='b110110010000000100011100010001001//A
end
6'h27:/*nor: R[rd] = ~(R[rs] | R[rt])*/ begin
CTRL='b11011001000000010010000010001001//A
end
6'h2a:/*Set less than(slt): R[rd] = (R[rs] < R[rt])
CTRL='b00000010000000010001100010000000//
end
6'h08:/*Shift less logical(sll): R[rd] = R[rs] <<
CTRL='b00000010000101000010000010000000//
end
6'h02:/*Shift right logical(srl): R[rd] = R[rs] >>
CTRL='b00000010000101000010100010000000//
end
6'h05:/*Jump register(jr): PC = R[rs]*/ begin
CTRL='b00000010000000000000000000000000//
end
endcase
end

```

C. I-Type Instruction Design

For I-type instructions, we will need to determine what instruction is needed depending on the opcode. There are two instructions that are not part of the CU are branch if not equal and branch if equal.

```

6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
CTRL='b00000001000000001001000001000000;//CTRL='
end
6'h1d:/*mul: R[rt] = R[rs] * SignExtImm*/ begin
CTRL='b00000010000010000001100010000000//DONE!
end
6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
CTRL='b00000010000000000001100001000000//DONE!
end
6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
CTRL='b00000010000000000001110001000000//DONE!
end
6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
CTRL='b00000010000000000000000000000000;//CTRL='
end
6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
CTRL='b0000001000000000000100100010000000;//DONE!
end
6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchAdc
CTRL='b00000010000000010000100001000000;//DONE!
end
6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchAdc
/*Tests for equality. In WB checks zero flag*/
CTRL='b00000010000000010000100001000000;//DONE!
end
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
CTRL='b00000010000010000000010001000000;//DONE!
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
CTRL='b000000100000000100100000100000;//CTRL='
end

```

I-type instructions contains a field for the immediate value, ZeroExtended, signExtended, LUI and Branch address were made. After instruction is executed, load word and store word are implemented in the memory state of the control unit.

```

`PROC_MEM: begin
CTRL='b0000000100110010000000000010000000;//DONE!
case(INST[31:26])
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
READ=1;
WRITE=0;
CTRL='b0000000100000100000000000010100000;
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
READ=0;
WRITE=1;
CTRL='b0000000100000000001001000001000000;
end
end

```

At last, the code below will be executed in the write back state.

```

// I-type (I and J are cased solely on opcode)
6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
CTRL='b101100010000000001001000010001011;//CTRL
end
6'h1d:/*mul: 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='b10111001000000000000000000010001011;//CTRL
end
6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
CTRL='b0000001000000000000100100010000000;//
end
6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchA
CTRL='b00000010000000010000100001000000;//
end
6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchA
/*Tests for equality. In WB checks zero flag*/
CTRL='b00000010000000010000100001000000;//
end
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
CTRL='b00000010000010000000010001000000;//
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
CTRL='b000000010000000001001000001001011;//CTRL
end
// J-Type

```

Figure 6.8. I-Type Write back

D. J-Type Instruction Design

J-type instruction is the last instruction that will be implemented in this program. There are a total of 4 operations in the J-type instruction set. The only one that will take place in the execution state of the control unit I the push operation. And

all of the other operation will take place in the write back state. Data_r1 is 0 by default.

```
6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
    READ=0;
    WRITE=1;
    CTRL='b0000000001010000000010101000000;
end
6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
    READ=1;
    WRITE=0;
    CTRL='b0000001000001000000001001000000;
end
endcase
// J-Type
6'h02:/*jmp: PC = JumpAddress*/ begin
    CTRL='b000000100000000000000001000000//DONE!
end
6'h03:/*jal: R[31] = PC + 1; PC = JumpAddress*/ begin
    CTRL='b000000100000000000000001000000//DONE!
end
6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
    CTRL='b0000001000001000000100001000000;//DONE!
end
6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
    CTRL='b000000100110100000010001000000;//DONE!
end
.
```

VI. CONCLUSION

In conclusion, after completing project 3, our DaVinci system is a better system with the usage of the logic gate. This was a more challenging part compared to what we did. Nonetheless, this gave me a lot of experience with creating a processor through Verilog language. Though the processor does not work perfectly, I am still proud that I have made it this far.