Mixed Model of a Computer System

NADA EL ZEINI
San Jose State University
nada.elzeini@sjsu.edu

***Abstract*— This report reflects the project of implementing a mixed model of a bare minimum system DaVinci v1.0m supporting instruction set CS147DV described is CS 147 class lectures (Spring 2019). The primary difference between this project and the previous one is the gate level modeling usage in this project, and the implementation of various components accordingly. The computer system to be implemented should be of a 32-bit processor and 256MB memory meaning a double-word (32-bit) addressable 64M address. As a result, the outcome expected are a fully functional processor supporting 'CS147DV' instruction set, 256MB memory model with data pre-loading capability, a fully functional system integrating the previous two called 'DaVinci v1.0m' and a full testing of it using a small program. The report's goal is to detail what it took to achieve this implementation.**

I. SYSTEM REQUIREMENTS

Basically, to implement the DaVinci v.1.0m system a representation (with Verilog as the hardware description language) of a processor and memory is required. The processor is a summation of a control unit (CU) , register file (RF) , and arithmetic & logic unit (ALU). Therefore, its implementation will have to integrate those 3 components. Data flow occurs in this way: Register file to ALU as 2 operands (op1 and op2), then ALU to register file as the result. Afterwards, from register file to memory to store the result to register file again to load value from memory as said in project 2. To achieve the implementation of a mixed model of a computer system with a 32-bit processor and 256MB memory the project is done following this flow: first, the processor implementation will have two major parts as described in the project's objectives. The first part will focus on reaching mixed level data path implementation; the schematic of the data path to be done is shown in figure 1.1.

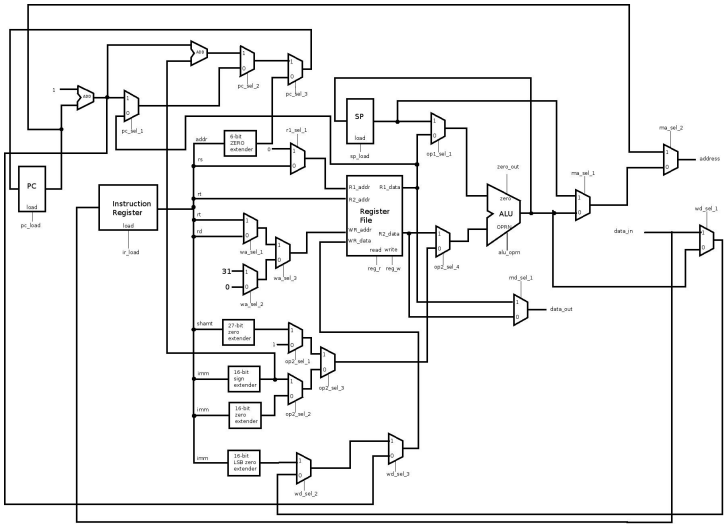


Figure1.1: Complete Datapath

The second part will concentrate on the behavioral control circuit implementation (some of it will be used from the previous project). Following this complete implementation, testing will be done for the system DaVinci v1.0m after having tested its components individually to make the complete testing relatively simpler. The following outlines the components required for the processor and memory.

(a) Instruction Set

'CS147DV" instruction set introduced in lecture 1 of CS147 class section 01 of Spring 2019 will be used. (Figure 1.2).

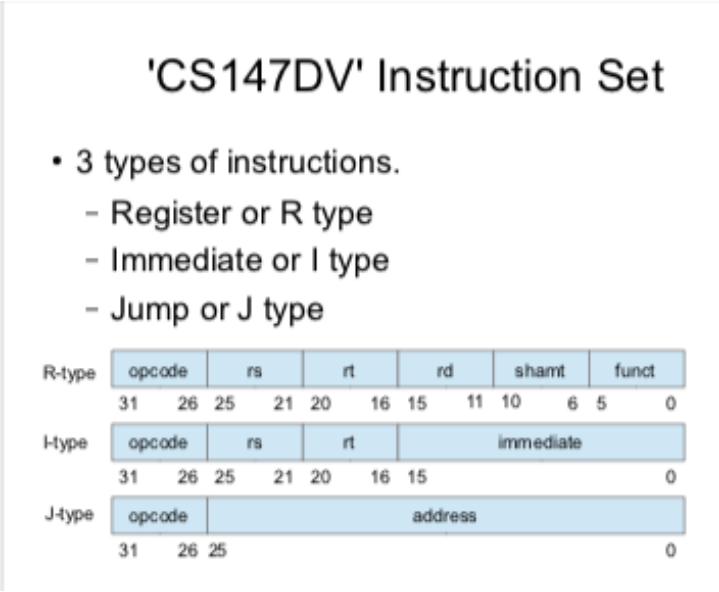


Figure 1.2: 'CS147DV' Instruction Set

The 3 types of instructions: Register (or R type), which allows 3 registers rs, rt and rd (destination) with a shift amount, each of 5 bits and 'funct' of 6 bits to specify operation (Figure 1.3); Immediate (or I type) which has only 2 registers (rt as destination and source) and 16-bit immediate (sign or zero extension might be used) (Figure 1.4) and the last type is J type which consist of 6 bits of opcode as the rest of the instruction types , with a 26-bit address (Figure 1.5).

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>

Figure 1.3: 'CS147DV' R-type Instructions

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>

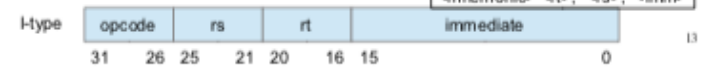


Figure 1.4: 'CS147DV' I-type Instructions

Name	Mnemonic	Format	Operation	OpCod
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>

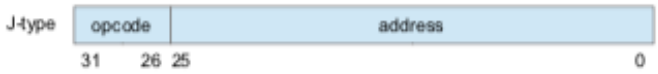


Figure 1.5: 'CS147DV' J-type Instructions

(b) Arithmetic Logic Unit (ALU)

As similarly stated in the previous project, the ALU is an essential building block of a computer processor, it provides a fundamental functionality of a computer. This digital circuit is used to handle arithmetic (addition, subtraction, multiplication...) and logic operations (like AND and OR). These nine operations of the 'CS147DV' instruction set will be handled by the ALU in addition to 'ZERO' flag:

- Addition
- Subtraction

- Multiplication
- Shift Right
- Shift Left
- Bitwise AND
- Bitwise OR
- Bitwise NOR
- Set Less Than
- 'ZERO' Flag

The ALU will be implemented at the gate level as discussed in following sections. The file used is 'alu.v'.

(c) Register File (RF)

Register File will also be done at the gate level with negative edge triggered clock. It consists of 32 registers (index 0 to 31) storing each a 32-bit word. RESET is done at the negative edge of the clock and the READ/WRITE operations are done at the positive edge. Combinations of 00 and 11 of read/write signals will make the RF hold the previous value of the previously read data; not hiZ since none of the IOs are bi-directional. The file used is 'register_file.v'.

(d) Memory

Same as in project 2, The memory will have the following properties:

- 64M storing 32-bit word at each address

- Reset on negative edge of reset signal, while all other operations will be done on positive edge (synchronous).

-Write operation at read=0 and write=1

-Any other combination of read/write signal will result in hiZ state of data port 'data'.

(d) Data Path

As the processor consists of the data path and the control unit, implementing the data path as in Figure 1.1 is a must to get the whole system to function as properly wanted. This implementation will be done at the gate level. Including Program Counter (PC) and Stack Pointer (SP) registers. The file used is 'data_path.v' mainly.

(e) Behavioral Control Circuit (CU)

The control unit is responsible for directing an operation between the RF, ALU and memory. It basically controls the processor to execute the 'CS147DV' instructions this project requires, following a 5-steps cycle changing states between: FETCH, DECODE, EXECUTE, MEMORY, and WRITE BACK (same from project 2 explanation). The control unit implementation will be behavioral as guided since gate level modeling would be too complex. The control unit synchronize operations of the processor, each bit of control signal will be assigned to control one part of data path accordingly (depending on each instruction and the stage).

II. DESIGN & IMPLEMENTATION OF ALU

Various components are needed to implement ALU, the design is the following outline of components. Afterwards, the correct implementation of each is done to produce a working ALU supporting instruction set 'CS147DV'. Figure 2.1 shows the schematic of the 32-bit ALU that is to be done.

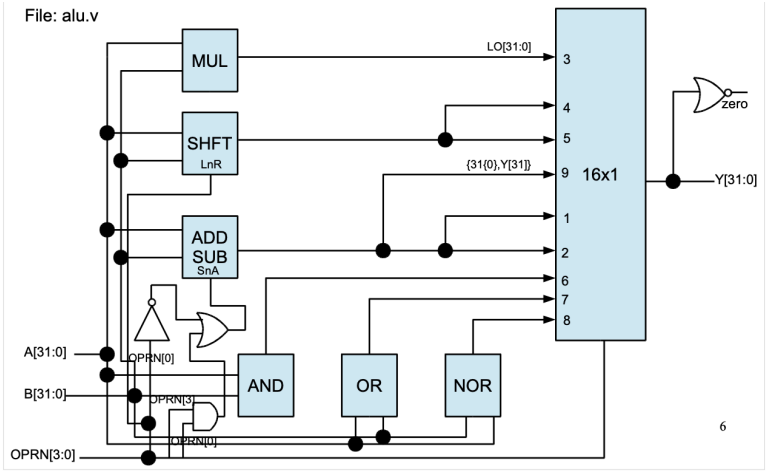


Figure 2.1: 32-bit ALU Schematic

(a) Full Adder

A full adder is implemented using already done half adders following the schematic in figure 2.2. The file used is 'full_adder.v' and the half adder module is done in 'half_adder.v'.

The sum output is 'S' (not Y). Carry-out is 'CO' and Carry-in is 'CI'.

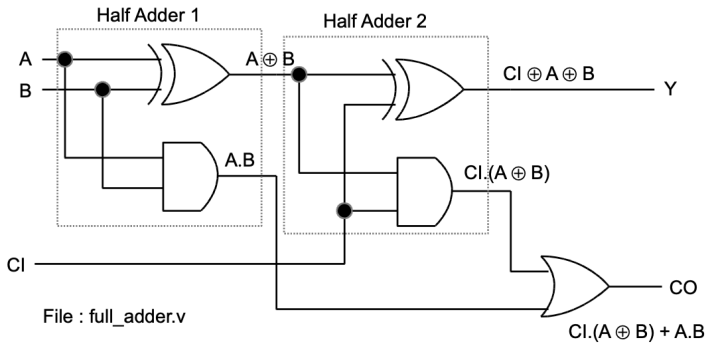


Figure 2.2: Full Adder Schematic

```
module FULL_ADDER(S,CO,A,B, CI);
output S,CO;
input A,B, CI;

wire sum_1, carry_01, carry_02;

HALF_ADDER ha_inst_1(.S(sum_1), .C(carry_01), .A(A), .B(B));
HALF_ADDER ha_inst_2(.S(S), .C(carry_02), .A(sum_1), .B(CI));

or or_inst(CO,carry_01,carry_02);


endmodule
```

Figure 2.3: Full Adder Module

(b) Adder - Subtractor

Using full adder instantiations and 'xor' instantiation, implementation of 32-bit adder/ subtractor is done, The schematic followed to implement it is in figure 2.4. 'Y' is the variable used for the output. The file 'rc_add_sub_32.v' is used (figure 2.5). In the same file, another adder/subtractor is module is done but extended to 64-bit.

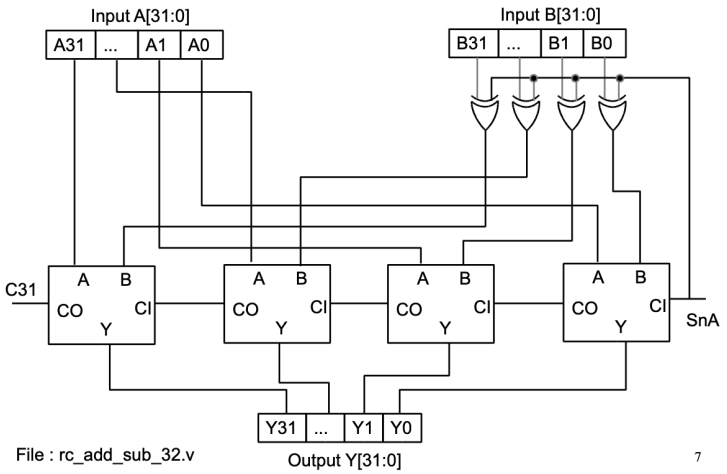


Figure 2.4: 32-bit adder subtractor schematic

```
module RC_ADD_SUB_32(Y, CO, A, B, SnA);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
output CO;
// input list
input [`DATA_INDEX_LIMIT:0] A;
input [`DATA_INDEX_LIMIT:0] B;
input SnA;


//add wires

//wire for xor result
wire [31:0]xor_result;
//wire to connec tfull adders Sna,CO,CI
wire [31:0] next_connect;
genvar i;
generate
for (i=0;i<32;i=i+1)
begin:  rc_add_sub_32_loop
xor xor_inst(xor_result[i],SnA,B[i]);
if(i==0) //if lsb connect snA to fa, input B(xor result)m op1 (input A), next >CO, outp
        begin FULL_ADDER fa(Y[i],next_connect[i],A[i],xor_result[i],SnA);
        end
else if(i==31)
        begin FULL_ADDER fa(Y[i],CO,A[i],xor_result[i],next_connect[i-1]); //CO intsead
        end
else if(i !=0 && i!=31)
        begin FULL_ADDER fa(Y[i],next_connect[i],A[i],xor_result[i],next_connect[i-1]);
        end


end
endgenerate
```

Figure 2.5: 32-bit adder subtractor module

(c) 32-bit 2x1 Multiplexer

To set a 32-bit 2x1 multiplexer, a 1-bit 2x1 multiplexer is needed. The file 'mux.v' will contain every muttiplexer module needed for the project. Figure 2.6 shows the logic circuit of a 1-bit 2x1 MUX.

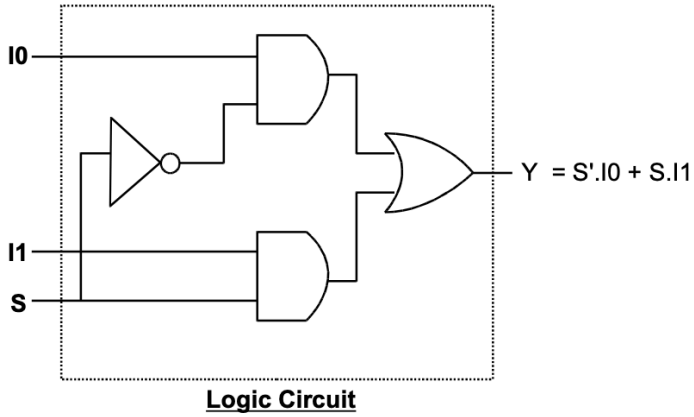


Figure 2.6: 1-bit 2x1 MUX schematic

As seen, one inverter gate, two 2-input AND gates and one 2-input OR gate are used.The gate level modeling code is shown in figure 2.7

```
// 1-bit mux
module MUX1_2x1(Y,I0, I1, S);
//output list
output Y;
//input list
input I0, I1, S;
//S is control
//done first
wire not_output,and_output1,and_output2;
not not_inst(not_output,S);
and and_inst1(and_output1,not_output,I0);
and and_inst2(and_output2,S,I1);
or or_inst(Y,and_output1,and_output2);

endmodule
```

Figure 2.7: 1-bit 2x1 MUX module

Now, we are able to set 2x1 32-bit multiplexer using a generate block (figure 2.8).

```
// 32-bit mux
module MUX32_2x1(Y, I0, I1, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input S;

///done second
genvar i;
generate
for(i=0;i<32;i=i+1)
begin: mux32_2x1_loop
MUX1_2x1 mux32_2x1(Y[i],I0[i],I1[i],S);
end
endgenerate
```

Figure 2.8: 32-bit 2x1 MUX module

(d) 32-bit Logic Gates

In the 'logic_32_bit.v' file, 32-bit NOR, OR, AND, INVERTER(NOT) and BUF will be done for necessary use later on.

Using the generate block and instantiation of the corresponding gate, this is simple to implement.

The figure 2.9 from the lab guides provided shows how this done. To do it for a different logic gate, say for AND all that is needed to be done is to replace the instantiated logic gate with the one wanted so 'and' instead of 'nor' and change the variables names accordingly.

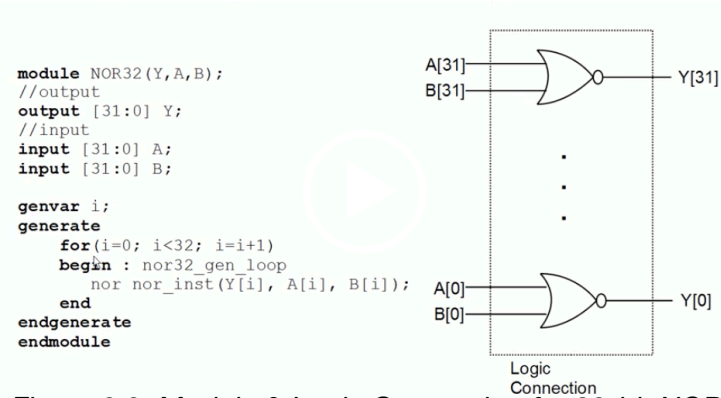


Figure 2.9: Module & Logic Connection for 32-bit NOR

(e) Unsigned Multiplier

For the ALU to support 'CS147DV' instructions a signed multiplier is needed, to accomplish that, an unsigned multiplier will be done first.

This implementation follows the logic circuit in figure 2.10. Using instantiations 32 times of 32-bit adder subtractor already implemented and 32-bit AND the unsigned multiplier is done. LO OR HI IS RESULT??? ??? ???////

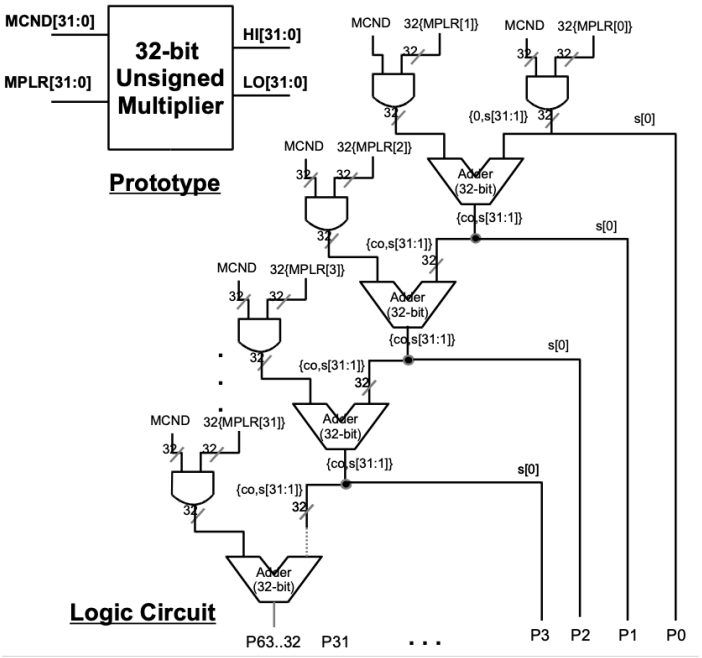


Figure 2.10:Logic Connection for 32-bit Unsigned Multiplier

(f) 2's Complement

In order to set a signed multiplier and for later use also, a 2's complement module is necessary. The implementation is simple, the usage of a 32-bit inverter done in 'logic_32_bit.v' and of a 32-bit adder done in 'rc_add_sub_32.v' are needed and 32-bit '1' as shown in the logic circuit in figure 2.11.

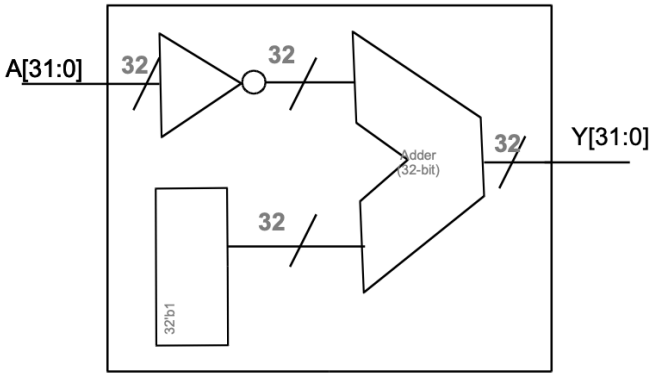


Figure 2.11: Two's Complement Logic Circuit

The 32-bit two's complement implementation will be done in the 'logic.v' file (figure 2.12).

```
module TWOSCOMP32(Y,A);
//output list
output [31:0] Y;
//input list
input [31:0] A;
wire [31:0] not_out;
//add wires
wire nulll;
genvar j;
generate
for(j=0;j<32;j=j+1)
begin : loop_2s_32
not not_inst(not_out[j],A[j]);
end
endgenerate
RC_ADD_SUB_32 adder(Y, nulll ,not_out, 32'b1, 1'b0);
endmodule
```

Figure 2.12:32-bit Two's Complement Module

Similarly, a 64-bit two's complement module is done in the the same file for later use.

(g) Signed Multiplier

Now that unsigned multiplier and 32-bit 2's complement are done, a signed multiplier is done following the schematic in figure 2.13. The implementation is done is the 'mult.v' file. Basically, an instantiation of the unsigned multiplier is needed along two 32-bit 2's complement and one 64-bit two's complement. In addition, 2 32-bit 2x1 multiplexers are used to set in place to select either original value or

complemented value (in case it's negative) depending of the most significant bit.

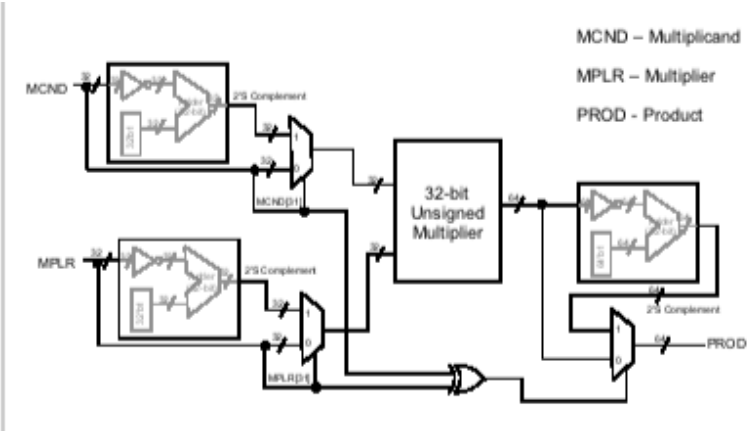


Figure 2.13:Logic Connection for 32-bit Signed

(h) Barrel Shifter

A barrel shifter is a digital circuit that shifts a data word by a certain number of bits without using sequential logic. In fact, it is one form of combination al circuit that shifts or rotated the input data bits by the number of bit positions specified by a value. A 4-bit version of a barrel shifter is known to be of 4 multiplexers with common selections in sequence with 4 more multiplexers such as the output of one mux is connected to the next as shown in the schematic in figure 2.14.

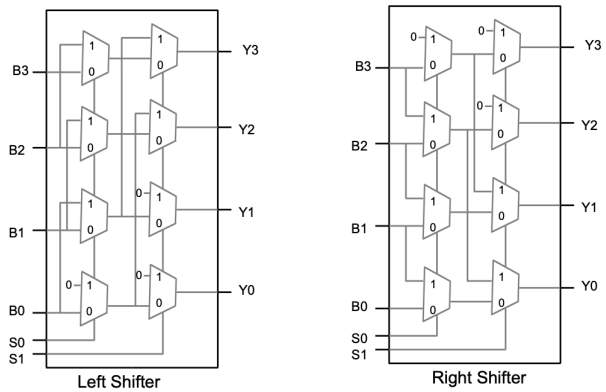


Figure 2.14:Logic Connections for 4-bit Left & Right Barrel Shifters

Modules for a 32-bit right and left shifters will be done then instantiated in 4-bit barrel (BARREL_SHIFTER_32 module) shifter and 32-bit shifter (SHIFT32 module). The last one can be implemented by extending the 4-bit barrel shifter or just instantiating the correct component and logically connect them following the circuit in figure 2.15 and 2.16.

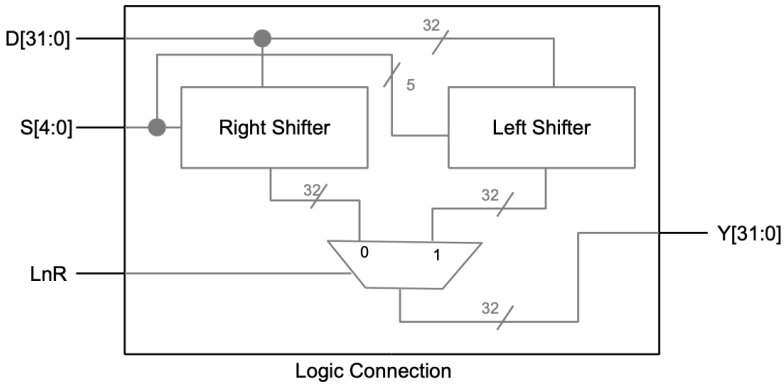


Figure 2.15:Logic Connection for 32-bit barrel shifter

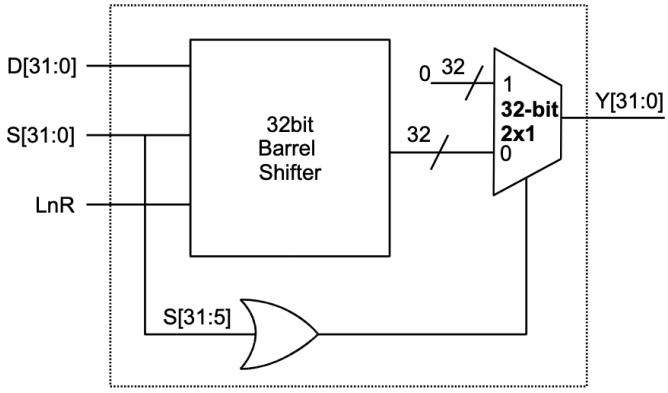


Figure 2.16:Logic Connection for 32-bit barrel shifter

Note that a 27-bit OR gate needs to be implemented (figure 2.17). NEEDED FOR

This implementation as well as barrel shifter components are set in the 'barrel_shifter.v' file.

```
//27 bit or
module OR27_1x1(Y,A);
//output
output  Y;
//input
input [31:0] A;
wire [29:0] wire_out;
genvar i;
generate
for(i=0;i<31;i=i+1)
begin : or_gen_loop
if(i==0)
begin
        or orzero_inst(wire_out[0],A[0],A[1]);
end
else if(i==30)
begin
or or_inst30(Y,wire_out[29],A[31]);
end
else
begin
or or_inst(wire_out[i],wire_out[i-1],A[i+1]);
end


end


endgenerate
endmodule
```

Figure 2.17: 27-bit OR gate module

(i) 32-bit 4x1 MUX

A 4 by 1 multiplexer has the logic connection of three 2x1 multiplexers (figure 2.18). This implementation is done along with the rest of multiplexers in 'mux.v'.

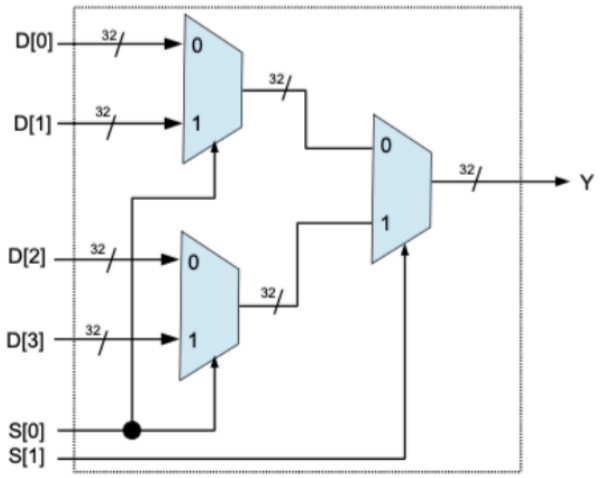


Figure 2.18: 32-bit 4x1 MUX logic circuit

(j) 32-bit 8x1 MUX

In a hierarchical way, 8x1 multiplexer is implemented using two 4x1 multiplexers done earlier and one 2x1 multiplexer both of 32-bit. The logic circuit is shown in figure 2.19. Figure 2.20 is the code done to implement 8x1 mux.

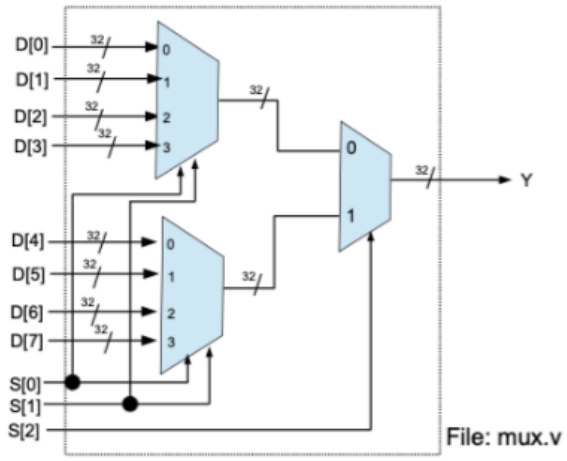


Figure 2.19: 32-bit 8x1 MUX

```
// 32-bit 8x1 mux
module MUX32_8x1(Y, I0, I1, I2, I3, I4, I5, I6, I7, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [31:0] I7;
input [2:0] S;
//done 4th
// need 2 32 bit 4x1 mux and 1 32-bit 2x1
//add wires
wire [31:0] mux4x1_out1,mux4x1_out2;
MUX32_4x1 mux4x1_1(mux4x1_out1,I0,I1,I2,I3,S[1:0]);
MUX32_4x1 mux4x1_2(mux4x1_out2,I4,I5,I6,I7,S[1:0]);
MUX32_2x1 mux2x1(Y,mux4x1_out1,mux4x1_out2,S[2]); //revi

endmodule
```

Figure 2.20: 32-bit 8x1 MUX module Verilog code

(k) 32-bit 16x1 MUX

Similarly for a 16x1 multiplexer, two 8x1 multiplexers can be used along with one 2x1 multiplexer. The Verilog code for this multiplexer is shown in figure 2.21.

```
// 32-bit 16x1 mux
module MUX32_16x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                  I8, I9, I10, I11, I12, I13, I14, I15, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [31:0] I7;
input [31:0] I8;
input [31:0] I9;
input [31:0] I10;
input [31:0] I11;
input [31:0] I12;
input [31:0] I13;
input [31:0] I14;
input [31:0] I15;
input [3:0] S;
//need 2 32-bit 8x1 and 1 32-bit 2x1 mux
//add wires
wire [31:0] mux8x1_out1,mux8x1_out2;
MUX32_8x1 mux8x1_1(mux8x1_out1,I0,I1,I2,I3,I4,I5,I6,I7,S[2:0]);
MUX32_8x1 mux8x1_2(mux8x1_out2,I8,I9,I10,I11,I12,I13,I14,I15,S[2:0]);
MUX32_2x1 mux2x1(Y,mux8x1_out1,mux8x1_out2,S[3]);
endmodule
```

Figure 2.21: 32-bit 16x1 MUX Verilog Code

(l) 32-bit ALU

Now that all the previous components have been implemented, everything is set to implement a 32-bit ALU using gate level modeling. The logic circuit the project is following to implement the ALU is shown in figure 2.1.

Figure 2.23 shows the Verilog code for the implementation. Notice the use of 32-bit buffer to buffer out the final output, after several writing in Verilog I figured this way would avoid confusion and misplacing wires and variables. Also, at this point we need to set a module for the 'Set less than' operation.

```
//
`include "prj_definition.v"
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 [5:0] OPRN; // operation code
// output list
output [`DATA_INDEX_LIMIT:0] OUT; // result of the operation.
output ZERO;
//add wires
wire [31:0] and32_wire;
wire [31:0] or32_wire;
wire [31:0]nor32_wire;
wire[31:0] add_sub_out;
 wire[31:0] co_wire;
wire[31:0] hi_wire;
wire[31:0]lo_wire;
wire or_out,and_out,LnR;
wire[31:0] shifter_out;
wire[31:0] mux_wire;
wire[31:0] slt_out;
///32 bit logic gates
AND32_2x1 and32_inst(and32_wire,OP1,OP2);
OR32_2x1 or32_inst(or32_wire,OP1,OP2);
NOR32_2x1 nor32_inst(nor32_wire,OP1,OP2);
//1 bit logic gates
and and_inst(and_out,OPRN[0],OPRN[3]);
not not_inst(not_out,OPRN[0]);
or or_inst(or_out,and_out,not_out); //noy out is lnr
//adder/sub
RC_ADD_SUB_32 adder_sub(add_sub_out, co_wire[0], OP1, OP2, or_out);
//setless than
SET_LESS_THAN slt(slt_out,OP1,OP2);
//shifter
SHIFT32 barrel_shifter(shifter_out,OP1,OP2,OPRN[0]);
//multiplier
MULT32 multiplier(hi_wire,lo_wire,OP1,OP2);
//16x1 mux
MUX32_16x1 muxx(mux_wire, add_sub_out,add_sub_out,add_sub_out,lo_wire, shifter_out, shifter_out, and32_wire,
 or32_wire,nor32_wire,slt_out, slt_out, slt_out,slt_out,slt_out, slt_out,slt_out, OPRN[3:0]);
//final
NOR32_1x1 nor31(ZERO,mux_wire);
BUF32_1x1 buf_inst(OUT,mux_wire);
endmodule
```

Figure 2.23: Verilog Code for 32-bit ALU

III. DESIGN & IMPLEMENTATION OF REGISTER FILE

Same as done in the ALU implementation, the register file will be implemented using gate level modeling. After implementing the following individual components, the implementation of the register file will follow.

(a) 1-bit SR-Latch

SR-latch will be implemented in accordance to the schematic in figure 3.1 in addition to nR and nP signals referring to 'not reset' (so PRESET) and 'not preset' (so RESET) respectively. Instead of 2 input NAND gates in the second half of the schematic, I decided to use two 3 input NAND gates to make implementation of flip flop simpler later on. Preset is at nR=1 and nP=0, Reset is at nP=1 and nR=0,while normal operation occurs at nP=1 and nR=1. Undefined operation is at nP=0 and nR=0. These combination are common for D-latch and Flip Flop too. Since we want to implement a positive edge triggered flip flop, for the SR-latch, the 'C' control signal should not be inverted as shown in the Verilog

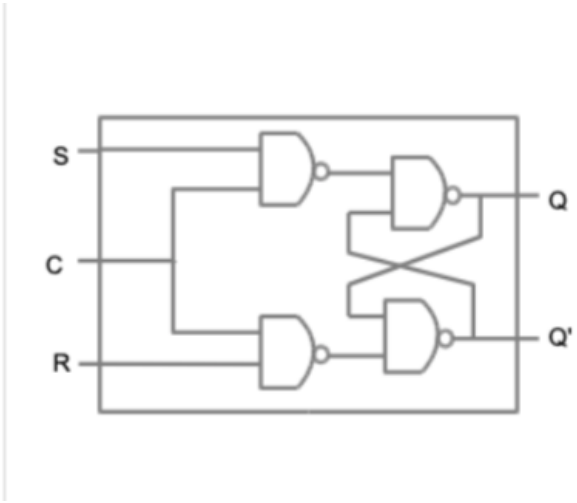


Figure 3.1: Logic Circuit of 1-bit SR-latch

Code. Note that the latches are done in the 'logic.v' file.

```
// 1 bit SR latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module SR_LATCH(Q,Qbar,S,R,C,nP,nR);
input S,R,C;
input nP, nR;
output Q,Qbar;
wire out[9:0];
        not invC(notC, C);
        nand nand1(out[6], S, C);
        nand nand2(out[7], C, R);
        nand nand3(out[8], nR, out[6], out[9]);
        nand nand4(out[9], out[8], out[7], nP);
        buf bufQ(Q, out[8]);
        buf bufQbar(Qbar, out[9]);
endmodule
```

Figure 3.2: Verilog Code for SR-Latch

(b) 1-bit D-Latch

There is an option to use the previously implemented SR-latch to implement the D-latch however to avoid confusion and after failed attempts at trying to set it using Sr latch I chose to implement it following the schematic shown in figure 3.3. It is important to not that since the goal for now is a positive edge triggered flip flop than the logic circuit should in fact have an inverter to complement the control C signal as done in the Verilog Code shown in figure 3.4

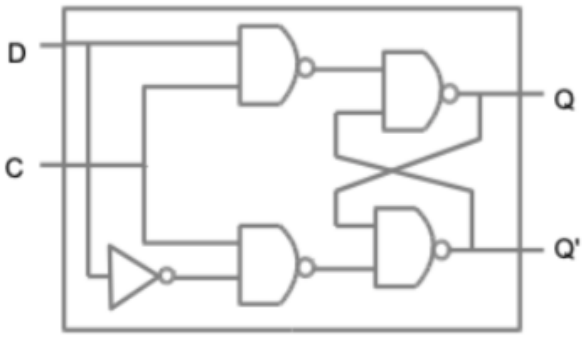


Figure 3.3: Logic Circuit for D-Latch

Additionally, nP and nR inputs are also inferred as done in the SR-latch implementation reason why we used two 3 input NAND gates (figure 3.4).

```
// normal operation nP=1, nR=1
module D_LATCH(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;
wire Dbar;


wire Y,Ybar;
wire [9:0] out;
wire dOut1,dOut2;
        not invD(notD, D);
not invC(notC, C);
        nand nand1(out[2], D, notC);
        nand nand2(out[3], notD, notC);
        nand nand3(dOut1, nR, out[2], dOut2);
        nand nand4(dOut2, dOut1, out[3], nP);
buf b1(Q, dOut1);
buf b2(Qbar, dOut2);
endmodule
```

Figure 3.4: D-Latch Verilog Code

(c) 1-bit Flip Flop

Now SR-latch and D-latch are done we can proceed to the positive edge triggered flip flop implementation. The basic logic circuit for it is in figure 3.5. Notice that the inverters were placed

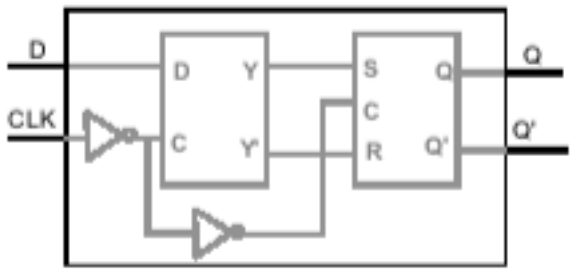


Figure 3.5: Positive Edge Triggered D Flip Flop Logic Circuit

correctly when implementing the Sr and D latches previously which means no inversion is needed at this point. The Verilog implementation is done in figure 3.6.

```
module D_FF(Q, Qbar, D, C,nP, nR);
        input nR, D, C, nP;
        output Q, Qbar;
        wire notD, notC, dout1, dout2;
D_LATCH d1(.Q(dout1), .Qbar(dout2), .D(D), .C(C), .nP(nP), .nR(nR));
SR_LATCH sr1(.Q(Q),.Qbar(Qbar),.S(dout1),.R(dout2),.C(C),.nP(nP),.nR(nR));
endmodule
```

Figure 3.6: Positive Edge Triggered D Flip Flop Verilog Code

(d) 1-bit Register

A 1-bit register is without doubt needed for the register file implementation given its definition. The schematic in figure 3.7 is simple to follow.

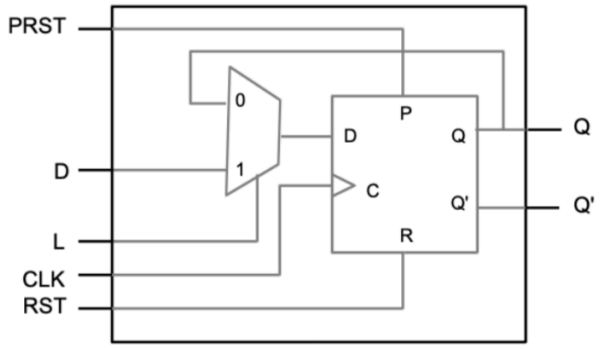


Figure 3.7: 1-bit Register Logic Circuit

Obviously, a 2x1 MUX is needed to select between Data and Preset, the LOAD 'L' signal is the selector. The flip flop that implemented earlier will be instantiated as well. The Verilog code in figure 3.8 shows how the connections were done in parallel to the 1-bit register schematic.

```
// normal operation nP=1, nR=1
module REG1(Q, Qbar, D, L, C, nP, nR);
        input nR, D, L, C, nP;
        output Q, Qbar;
        wire muxOut;
        wire flopOut;
         MUX1_2x1 m(muxOut, ff_out, D, L);

        D_FF b(.Q(ff_out), .Qbar(Qbar), .nR(nR), .D(muxOut), .C(C), .nP(nP));
        buf(Q, ff_out);
endmodule
```

Figure 3.8: 1-bit Register Verilog Code

(e) 32-bit Register

Using a generate block, a 32 bit register is done in parallel to the logic circuit provided in the lab codes for the CS147 Spring 2019 class.

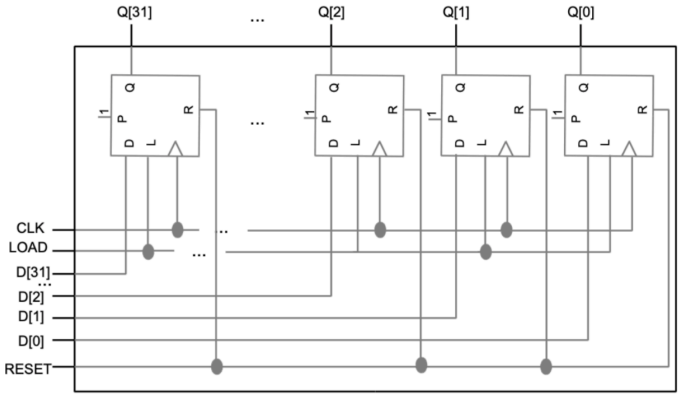


Figure 3.8: 32-bit Register Logic Circuit

The preset is set to 1 so nR 'not Reset' which is preset will be set to 1 (figure 3.9) and the nP will be connected to the RESET signal.

```
module REG32(Q, D,LOAD, CLK, RESET);
        input CLK,LOAD,RESET;
        input [31:0] D;
        output [31:0] Q;
        wire Qbar;
        wire notR;

        genvar i;
        generate
                for (i=0; i<32; i=i+1) begin : bitreg_gen_loop
                        //not n(notR, RESET);
                        REG1 r(.Q(Q[i]), .Qbar(Qbar), .nR(1'b1), .D(D[i]), .L(LOAD), .C(CLK), .
                end
        endgenerate
```

Figure 3.9: Verilog Code for 32-bit register file

(f) 32-bit 32x1 MUX

In the 'mux.v' file, a 32-bit 32x1 multiplexer is done. To do this, 2 instantiations of 32-bit 16x1 multiplexers are used along with a third multiplexer 32-bit of 2 by one.(figure3.10)

```
// 32-bit mux
module MUX32_32x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                     I8, I9, I10, I11, I12, I13, I14, I15,
                     I16, I17, I18, I19, I20, I21, I22, I23,
                     I24, I25, I26, I27, I28, I29, I30, I31, S);
 // output list
 output [31:0] Y;
 //input list
 input [31:0] I0, I1, I2, I3, I4, I5, I6, I7;
 input [31:0] I8, I9, I10, I11, I12, I13, I14, I15;
 input [31:0] I16, I17, I18, I19, I20, I21, I22, I23;
 input [31:0] I24, I25, I26, I27, I28, I29, I30, I31;
 input [4:0] S;

 // done last
 //need 2 32-bit 16x1 mux and 1 32-bit
 //add wires
 wire [31:0] mux16x1_out1,mux16x1_out2;

 MUX32_16x1 mux16x1_1(mux16x1_out1,I0, I1, I2, I3, I4, I5, I6, I7,
                     I8, I9, I10, I11, I12, I13, I14, I15,S[3:0]);
 MUX32_16x1 mux16x1_2(mux16x1_out2, I16, I17, I18, I19, I20, I21, I22, I23,
                     I24, I25, I26, I27, I28, I29, I30, I31,S[3:0]);
 MUX32_2x1 mux2x1(Y,mux16x1_out1,mux16x1_out2,S[4]);


 endmodule
```

Figure 3.10: Verilog Code for 32-bit 32x1 MUX

(g) 2-to-4 Line Decoder

Briefly, a decoder is a combinational circuit with an n-bit binary code applied to the inputs and m-bit binary code expected at the output. Decoders are necessary for the implementation of a register file. Starting with a simple 2-to-4 line decoder, in a hierarchy format we will use these implementations to

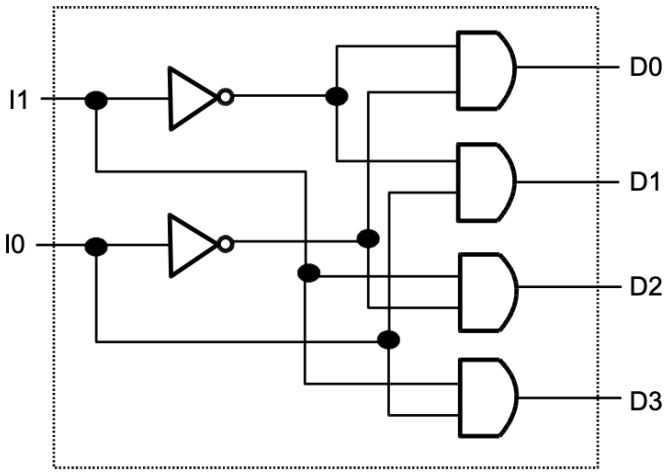


Figure 3.11:Logic Circuit for 2-to-4 line decoder

set the next ones as done for multiplexers. All decoders are done in the 'logic.v' file.

Simply, instantiation of 4 2-input AND gates are needed in addition to 2 inverters(figure 3.11). The simple Verilog code is shown on figure 3.12.

```
// 2x4 Line decoder
module DECODER_2x4(D,I);
// output
output [3:0] D;
// input
input [1:0] I;

wire W[1:0];

not not1(W[0],I[0]);//first inverter
not not2(W[1],I[1]);//second inverter
and and1(D[0],W[0],W[1]);
and and2(D[1],W[1],I[0]);
and and3(D[2],W[0],I[1]);
and and4(D[3],I[0],I[1]);


endmodule
```

Figure3.12: Gate level modeling for 2-to-4 line decoder

(h) 3-to-8 Line Decoder

This decoder will need a 2-to4 line decoder instantiation and using a for loop we instantiate corresponding AND gates as in the 3.13 figure.

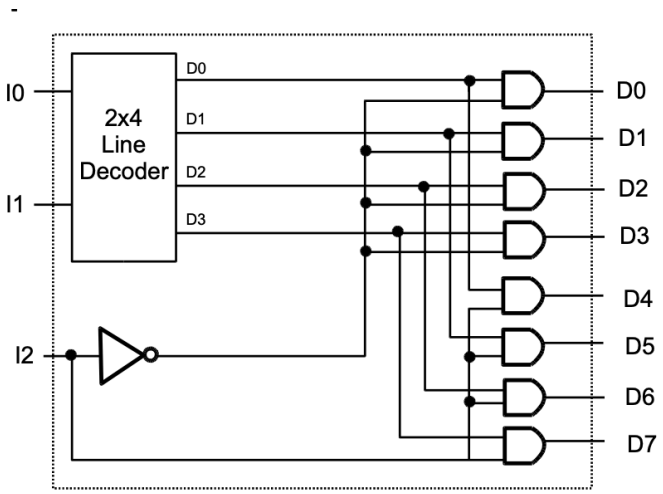


Figure 3.13:Logic Circuit for 3-to-8 line decoder

```
// 3x8 Line decoder
module DECODER_3x8(D,I);
// output
output [7:0] D;
// input
input [2:0] I;
wire [4:0]d;
DECODER_2x4 d2x4(d[3:0],I[1:0]); //leave one out for third input
not inst(d[4],I[2]);
//use generate cause faster
genvar i;
generate
for(i=0;i<4;i=i+1)
begin : decoder_loop
and and1(D[i],d[4],d[i]);
and and2(D[i+4],I[2],d[i]);
end
endgenerate
endmodule
```

Figure3.14: Gate level modeling for 3-to-8 line decoder

Notice in figure 3.14 the loop keeps going till it's less than 4 making perfect accordance with the schematic of this decoder.

(i) 4-to-16 Line Decoder

Similarly, this is done using the previous 3-to-8 decoder instantiation, an inverter along with a for loop that goes till index 'i' is less than 8.

(j) 5-to-32 Line Decoder

This decoder is implemented using the previous 4-to-16 decoder instantiation with an inverter and with a for loop that goes till index 'i' is less than 16 of 2 and gates instantiations each time.

(k) 32x32-bit Register File

At this point, every component needed is done and ready to be used in the register file implementation.

A logic circuit or schematic is always a great reference, as implementation can be done in parallel to it (figure 3.14).

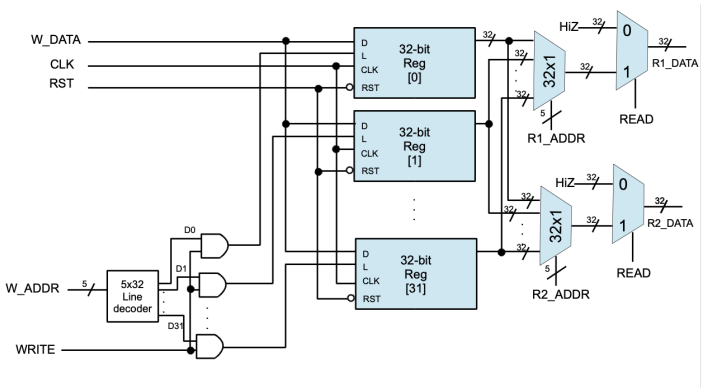


Figure 3.14: Schematic for 32x32 bit register file

Using a generate block we instantiate 32 times a 32-bit register that we implemented lastly. Clock and Reset are common signals. The write data is also common between the 32-bit registers. What is different is the LOAD 'L' input , if the load is 1 then only the register output will be returned. To have only one register 'on' we use the 5x32 decoder, the output from the decoder will only have one bit on, so one L signal is on. To do this, AND gates are needed, 2 inputs each, one for WRITE and the second is for the each of the 1 bit of the decoder output such that when write is 1 and the chosen bit is 1 the AND gate returns 1 turning the LOAD signal on. The rest is clear in the schematic. The Verilog code is shown in figure 3.15.

```
module REGISTER_FILE_32x32(DATA_R1, DATA_R2, ADDR_R1, ADDR_R2,
                           DATA_W, ADDR_W, READ, WRITE, CLK, RST);
// input list
input READ, WRITE, CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_W;
input [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1, ADDR_R2, ADDR_W;
output [`DATA_INDEX_LIMIT:0] DATA_R1;
output [`DATA_INDEX_LIMIT:0] DATA_R2;
wire [31:0] reg_wire [31:0];
wire [31:0] decoder_wire;
wire [31:0] and_wire;
wire [31:0] reg_out1;
wire [31:0] reg_out2;
wire notRST;
DECODER_5x32 decoder(decoder_wire, ADDR_W);
            //not(notRST,RST);
genvar i;
for (i=0; i<32; i=i+1)
begin : reg32_loop
            and and_inst(and_wire[i], decoder_wire[i], WRITE);
             REG32 reg32(.Q(reg_wire[i]),.D(DATA_W) ,.LOAD(and_wire[i]), .CLK(CLK),.RESET(RST));
    end
MUX32_32x1 mux1(reg_out1, reg_wire[0], reg_wire[1], reg_wire[2], reg_wire[3],
reg_wire[4],  reg_wire[5],  reg_wire[6],  reg_wire[7],  reg_wire[8],
reg_wire[9],  reg_wire[10], reg_wire[11], reg_wire[12],reg_wire[13],
reg_wire[14], reg_wire[15], reg_wire[16],reg_wire[17], reg_wire[18],
reg_wire[19], reg_wire[20], reg_wire[21], reg_wire[22], reg_wire[23],
reg_wire[24], reg_wire[25], reg_wire[26], reg_wire[27], reg_wire[28],
reg_wire[29], reg_wire[30], reg_wire[31], ADDR_R1);
    MUX32_32x1 mux2(reg_out2, reg_wire[0], reg_wire[1], reg_wire[2], reg_wire[3],
reg_wire[4],  reg_wire[5],  reg_wire[6],  reg_wire[7],  reg_wire[8],
reg_wire[9],  reg_wire[10], reg_wire[11], reg_wire[12],reg_wire[13],
reg_wire[14], reg_wire[15], reg_wire[16],reg_wire[17], reg_wire[18],
reg_wire[19], reg_wire[20], reg_wire[21], reg_wire[22], reg_wire[23],
reg_wire[24], reg_wire[25], reg_wire[26], reg_wire[27], reg_wire[28],
reg_wire[29], reg_wire[30], reg_wire[31], ADDR_R2);
    MUX32_2x1 mux3(DATA_R1, 32'bZ, reg_out1, READ);
    MUX32_2x1 mux4(DATA_R2, 32'bZ, reg_out2, READ);
endmodule
```

Figure 3.15: Gate Level Modeling for 32x32 bit register file

IV. DESIGN & IMPLEMENTATION OF DATA PATH

The complete data path to be done is shown in figure 1.1. Now that every component needed are done, it is rather simple to implement the data path.

Depending on the component, the signal controlling that component will be connected the to CTRL input of the data path module. A wire for each signal will be set with the correct size.

First of all, the adders were implemented (no control signal is needed since SnA will be zero since it's addition (1 is for substraction). I proceeded with implementing multiplexers in parallel to the data path given in figure 1.1. An ALU, Register File are instantiated. An Instruction register is instantiated using a 32 bit register. PC and SP registers are also implemented using the REG32_PP that was suggested for the project and that is set in the logic.v file. The Verilog code is shown in figure 4.1.

```
//PC REGISTER
defparam PC.PATTERN=32'h00001000;
REG32_PP PC(.Q(pc_load),.D({6'b0,pc_sel_3}),.LOAD(CTRL[12]),.CLK(CLK),.RESET(RST));
//SP
defparam SP.PATTERN=32'h03fffffc;
REG32_PP SP(.Q(sp_load),.D(alu_out),.LOAD(CTRL[23]),.CLK(CLK),.RESET(RST));
```

Figure 4.1: Verilog Code for PC and SP register implementation.

The rest of the code is implemented in the 'data_path.v' file as explained.

V. DESIGN & IMPLEMENTATION OF CONTROL UNIT

The control unit implementation is done in the 'control_unit' file. Using the same design from project 2 all we have to do is replace the assignments of registers and the operations for each instruction at each of the 5 stages by the 'CTRL' signal of 32-bit with '1' for the corresponding place to indicate which component is on at a certain stage for a certain instruction. For intance at the EXE stage for the R-type addition isntruction the 'op2_sel_4' signal should be on so we put 1 at the 8th index in CTRL and 1 at the 21st index since that is the corresponding for the addition ALU operation. To show what the inputs and outputs are the schematic helps clarify

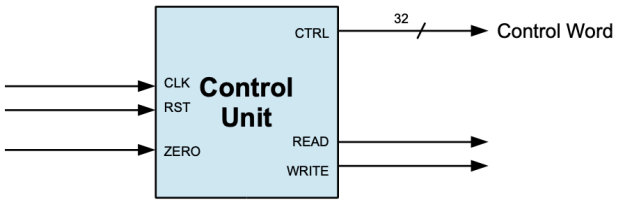


Figure 5.1: Control Unit Schematic

that in figure 5.1. In addition, an INSTRUCTION

input is also set for this module. This input will be parsed into an instruction register at the DECODE stage.

VI.DESIGN & IMPLEMENTATION OF MEMORY

This section will explain the modeling of a 256 MB memory with 32-bit word line, meaning we have 64M address to access the data from the memory. Figure 6.1 shows the ports required. (explanation from project2)

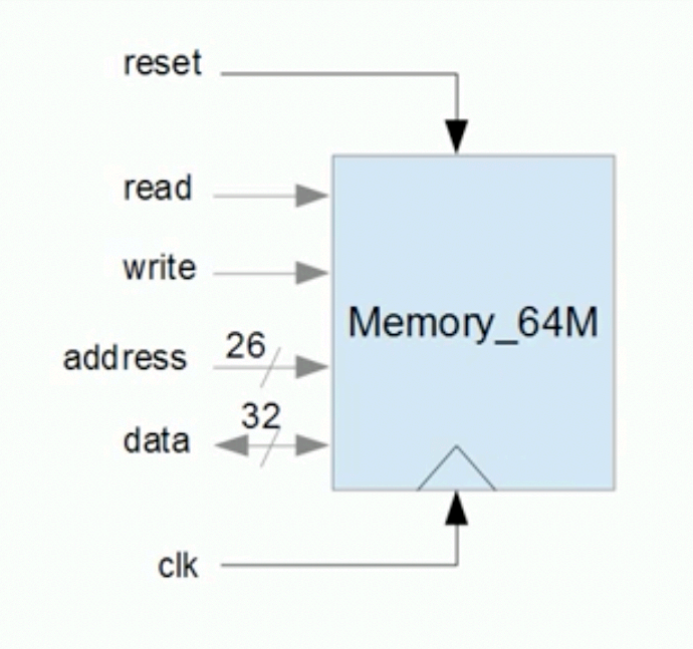


Figure 6.1: Memory Schematic

A memory wrapper implementation was done following the schematic in figure 6.2 provided by lab 17 from CS147 Spring 2019 class.

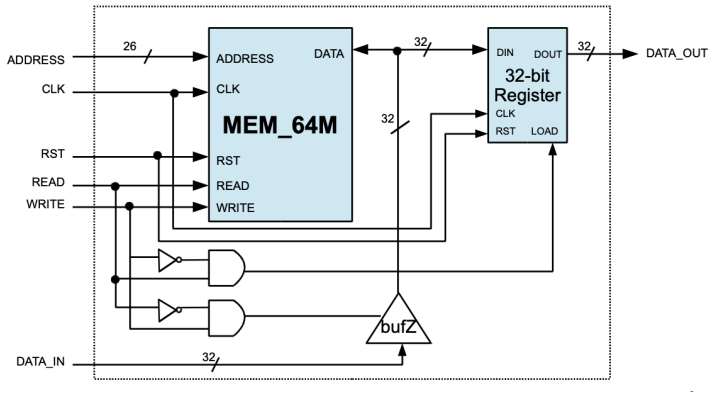


Figure 6.2: Memory Wrapper Implementation Schematic

VII. DESIGN & IMPLEMENTATION OF PROCESSOR

Processor implementation is done following the connections show in its schematic (figure 7.1). A fully functional processor supporting instruction set of 'CS147DV' is an outcome expected from this project. The 'processor.v' file should implement the processor using the control unit and data path implementations. The implementation of these instances will ensure that the processor will function correctly

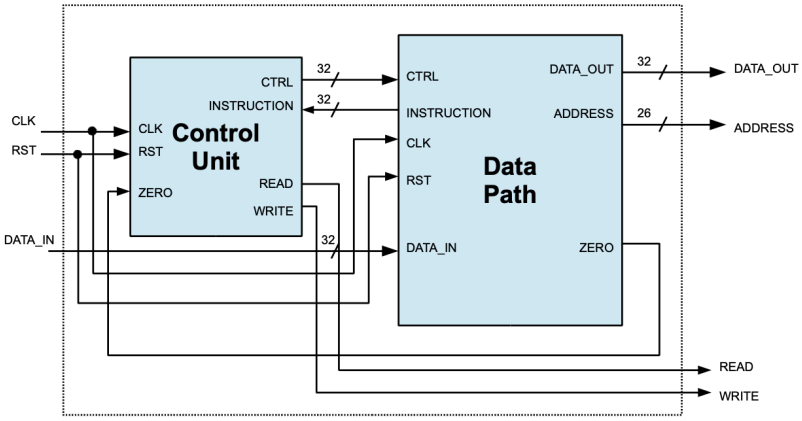


Figure 7.1: Processor schematic

VIII. TEST STRATEGY AND TEST IMPLEMENTATION

In order to make debugging simpler and easier, I tested each relevant component to ensure that it is working properly before moving on to the next one which is dependent on the previous one. For each following section, a test bench was set. Each section will state the name of the test bench file with important details when needed along with a transcript output and waveform output when it makes testing clearer and more accurate.

(a) Adder/Subtractor Testing

TestBench file: 'rc_add_sub_32_tb.v'

Transcript output for testing of 64 bit module of adder subtractor:

```
# Start time: 21:04:26 on May 03,2019
# Loading work.rc_add_sub_32_tb
# Loading work.RC_ADD_SUB_64
# Loading work.FULL_ADDER
# Loading work.HALF_ADDER
VSIM 7> run -all
#
#  A=                  0 SnA=0 B=                  0 result is:                  0  CO is 0
#
#  A=                  0 SnA=1 B=                  0 result is:                  0  CO is 1
#
#  A=                  9 SnA=0 B=                  8 result is:                 17  CO is 0
#
#  A=                  9 SnA=1 B=                  8 result is:                  1  CO is 1
#
#  A=                  5 SnA=1 B=                  5 result is:                  0  CO is 1
#
#  A=                  7 SnA=1 B=                 10 result is:18446744073709551613  CO is 0
```

Transcript output for testing of 32 bit module of adder subtractor:

```
# Start time: 21:07:52 on May 03,2019
# Loading work.rc_add_sub_32_tb
# Loading work.RC_ADD_SUB_32
# Loading work.FULL_ADDER
# Loading work.HALF_ADDER
VSIM 10> run -all
#
#  A=          0 SnA=0 B=          0 result is:          0  CO is 0
#
#  A=          0 SnA=1 B=          0 result is:          0  CO is 1
#
#  A=          9 SnA=0 B=          8 result is:         17  CO is 0
#
#  A=          9 SnA=1 B=          8 result is:          1  CO is 1
#
#  A=          5 SnA=1 B=          5 result is:          0  CO is 1
#
#  A=          5 SnA=0 B=          5 result is:         10  CO is 0
```

(b) Multiplexers Testing

TestBench file: 'mux_tb.v'

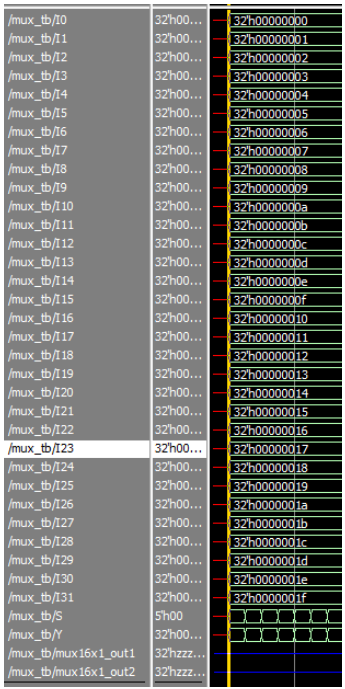
Details: I started testing for 2x1 MUX then 4x1 MUX, the outputs were expected, then I realized that testing the 32x1 MUX will reduce testing time as it would ultimately let me know if the rest of the implemented multiplexers are working properly. So if the testing didn't produce the expected result then at that point I would test each multiplexer individually and see where it went wrong. Luckily, the testing resulted in expected outcome. To test it, I did the following 32 times: I assigned a number for each input corresponding the the input order in a way that I# will

have # assigned to it. That way the result will be the control I assigned.

Transcript output: (partial)

```
# control is:00001 result is:          1
# control is:00010 result is:          2
# control is:00011 result is:          3
# control is:00100 result is:          4
# control is:00101 result is:          5
# control is:00110 result is:          6
# control is:00111 result is:          7
# control is:01000 result is:          8
# control is:01001 result is:          9
# control is:01010 result is:         10
# control is:01011 result is:         11
# control is:01100 result is:         12
# control is:01101 result is:         13
# control is:01110 result is:         14
# control is:01111 result is:         15
# control is:10000 result is:         16
# control is:10001 result is:         17
# control is:10010 result is:         18
# control is:10011 result is:         19
# control is:10100 result is:         20
# control is:10101 result is:         21
# control is:10110 result is:         22
# control is:10111 result is:         23
# control is:11000 result is:         24
# control is:11001 result is:         25
# control is:11010 result is:         26
# control is:11011 result is:         27
# control is:11100 result is:         28
# control is:11101 result is:         29
# control is:11110 result is:         30
# control is:11111 result is:         31
```

Waveform output:



(c) Multiplier Testing

Test Bench File: must_tb.v

Details; LO result are the 32 least significant bits and Hi are the 32 bit most significants bits.

The module of the testing is show below:(partial)

```
wire [31:0] LO;
 MULT32 signed_mult(HI, LO, A, B);
initial
begin
#5 $write("testing signed multiplier\n");
#5 A='b0; B='b0; //can write golden here CHANGE OPERATIONS
#5 $write("A is%d B is%d HI=%b LO=%b\n",A,B,HI,LO);
#5 A='b1; B='b0;
#5 $write("A is%d B is%d HI=%b LO=%b\n",A,B,HI,LO);
#5 A=5; B=3;
#5 $write("A is%d B is%d HI=%b LO=%b\n",A,B,HI,LO);
#5 A='b1; B='b1;
#5 $write("A is%d B is%d HI=%b LO=%b\n",A,B,HI,LO);
#5 A='b11; B='b1;
#5 $write("A is%d B is%d HI=%b LO=%b\n",A,B,HI,LO);
#5 A='hffffffff; B='b1;
#5 $write("A is%d B is%d HI=%d LO=%d\n",A,B,HI,LO);
#5 A='hFFFFFFF7; B='h00000001;
#5 $write("A is%h B is%h HI=%d LO=%d\n",A,B,HI,LO);
#5 A='b11; B='b11;
#5 $write("A is%d B is%d HI=%b LO=%b\n",A,B,HI,LO);
#5 A='hfffffffd;B='hfffffffe;
#5 $write("A is%h B is%h HI=%b LO=%b\n",A,B,HI,LO);
#5 A='hfffffff4;B='h00000002;
#5 $write("A is%h B is%h HI=%b LO=%b\n",A,B,HI,LO);
#5 A='hfffffffb;B='hfffffffe;
#5 $write("A is%h B is%h HI=%b LO=%b\n",A,B,HI,LO);
#5 A='hfffffff6;B='hfffffffb;
#5 $write("A is%h B is%h HI=%b LO=%b\n",A,B,HI,LO);
```

Used different radix to get better idea of result.

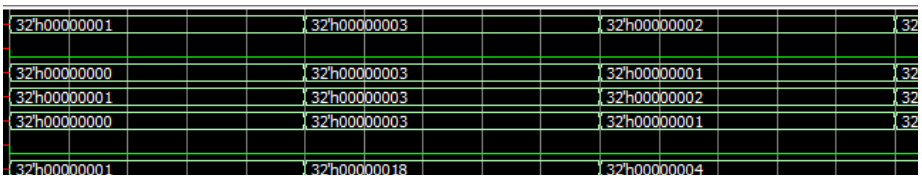
Transcript output: put the Hi first to make result more obvious. When the result is negative sign extension makes HI all 1 as shown for some cases in the output below.

```
VSIM 34> run -all
# testing signed multiplier
# A is         0 B is         0 HI=00000000000000000000000000000000 LO=00000000000000000000000000000000
# A is         1 B is         0 HI=00000000000000000000000000000000 LO=00000000000000000000000000000000
# A is         5 B is         3 HI=00000000000000000000000000000000 LO=00000000000000000000000000001111
# A is         1 B is         1 HI=00000000000000000000000000000000 LO=00000000000000000000000000000001
# A is         3 B is         1 HI=00000000000000000000000000000000 LO=00000000000000000000000000000011
# A is4294967295 B is         1 HI=42949672295 LO=4294967295
# A isfffffff7 B is00000001 HI=4294967295 LO=4294967287
# A is         3 B is         3 HI=00000000000000000000000000000000 LO=00000000000000000000000000001001
# A isffffffff B isfffffffe HI=00000000000000000000000000000000 LO=00000000000000000000000000000010
# A isfffffff4 B is00000002 HI=11111111111111111111111111111111 LO=11111111111111111111111111101000
# A isfffffffb B isfffffffe HI=00000000000000000000000000000000 LO=00000000000000000000000000001010
# A isfffffff6 B isfffffffb HI=00000000000000000000000000000000 LO=00000000000000000000000000110010
VSIM 35>
```

(d) Barrel Shifter Testing

TestBench file:barrel_shifter_tb.v

Waveform output: success (partial)



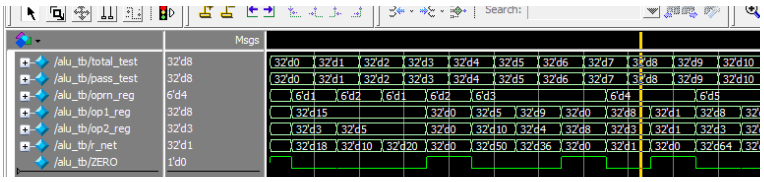
(e) ALU Testing

TestBench file: 'alu_tb.v' same testbench used in previous projects 1 and 2.

Transcript output:

```
VSIM 38> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 0 - 0 = 0 , got 0 ... [PASSED]
# [TEST] 5 * 10 = 50 , got 50 ... [PASSED]
# [TEST] 9 * 4 = 36 , got 36 ... [PASSED]
# [TEST] 0 * 8 = 0 , got 0 ... [PASSED]
# [TEST] 8 >> 3 = 1 , got 1 ... [PASSED]
# [TEST] 1 >> 1 = 0 , got 0 ... [PASSED]
# [TEST] 8 << 3 = 64 , got 64 ... [PASSED]
# [TEST] 2 << 1 = 4 , got 4 ... [PASSED]
# [TEST] 15 & 15 = 15 , got 15 ... [PASSED]
# [TEST] 1 & 15 = 1 , got 1 ... [PASSED]
# [TEST] 0 | 0 = 0 , got 0 ... [PASSED]
# [TEST] 15 | 15 = 15 , got 15 ... [PASSED]
# [TEST] 1 | 5 = 5 , got 5 ... [PASSED]
# [TEST] 1 ~| 4 = 4294967290 , got 4294967290 ... [PASSED]
# [TEST] 2 ~| 0 = 4294967293 , got 4294967293 ... [PASSED]
# [TEST] 9 < 1 = 0 , got 0 ... [PASSED]
# [TEST] 6 < 7 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          20
#       Total number of pass           20
#
```

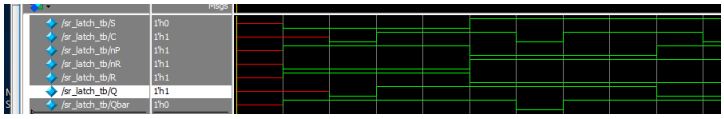
Waveform output: (partial)



(f) Sr Latch Testing

TestBench file: sr_latch_tb.v

Waveform output:



(g) D Latch Testing

TestBench file: d_latch_tb.v

Transcript output is expected:

```
PRESET C = 1 nP=0 nR=1 D=1 Q=0 Qbar=1

PRESET C = 1 nP=0 nR=1 D=0 Q=0 Qbar=1

Normal C = 1 nP=1 nR=1 D=0 Q=0 Qbar=1

Normal C = 1 nP=1 nR=1 D=1 Q=1 Qbar=0
```

(h)Flip Flop Testing

TestBench file: d_ff_tb.v

Transcript output is expected: (all variables manipulated in test bench are shown in transcript output)

```
#
#  SET C = 1 nP=0 nR=1 D=0 Q=0 Qbar=1
#
#  SET C = 1 nP=0 nR=1 D=1 Q=0 Qbar=1
#
# RESET C = 1 nP=1 nR=0 D=0 Q=1 Qbar=0
#
# RESET C = 1 nP=1 nR=0 D=1 Q=1 Qbar=0
#
#  Normal C = 1 nP=1 nR=1 D=0 Q=0 Qbar=1
#
#  Normal C = 1 nP=1 nR=1 D=1 Q=1 Qbar=0
```

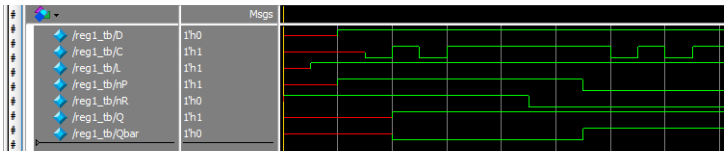
(i) Register Testing

TestBench file: reg1_tb.v

Transcript Output: success in comparison to knowledge from lectures

```
#
# C = 1 L=1 nP=1 nR=1 D=1 Q=1 Qbar=0
#
# C = 1 L=1 nP=0 nR=0 D=1 Q=1 Qbar=1
#
# C = 1 L=1 nP=1 nR=0 D=1 Q=1 Qbar=0
#
# C = 1 L=1 nP=1 nR=0 D=0 Q=1 Qbar=0
```

Waveform output:(partial)



(j) Decoder Testing

TestBench file: decoder_5x32_tb.v

Only tested 5 to 32 line decoder because it includes other implemented decoders

Transcript output: success

```
# I=00000 , D=00000000000000000000000000000001
#
# I=00001 , D=00000000000000000000000000000010
#
# I=00010 , D=00000000000000000000000000000100
#
# I=00011 , D=00000000000000000000000000001000
#
# I=00100 , D=00000000000000000000000000010000
#
# I=00101 , D=00000000000000000000000000100000
#
# I=00110 , D=00000000000000000000000001000000
#
# I=00111 , D=00000000000000000000000010000000
#
# I=01000 , D=00000000000000000000000100000000
#
# I=01001 , D=00000000000000000000001000000000
#
# I=01010 , D=00000000000000000000010000000000
#
# I=01011 , D=00000000000000000000100000000000
#
# I=01100 , D=00000000000000000001000000000000
#
# I=01101 , D=00000000000000000010000000000000
#
# I=01110 , D=00000000000000000100000000000000
#
# I=01111 , D=00000000000000001000000000000000
#
# I=10000 , D=00000000000000010000000000000000
#
# I=11111 , D=10000000000000000000000000000000
```

(k) Register File Testing

TestBench file: reg_tb.v

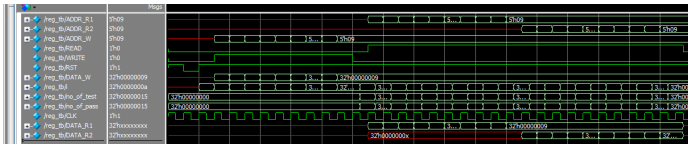
Using the same test bench from project 2.

HiZ is not necessary.

Transcript output: success

```
# Loading work.reg_tb
# ** Warning: Design size of 108 statements or 730
# Expect performance to be adversely affected.
VSIM 5> run -all
#
#       Total number of tests          21
#       Total number of pass           21
#
```

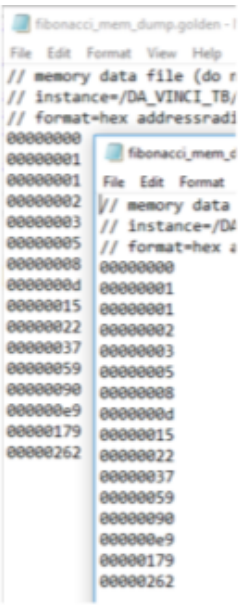
Waveform output:



(l) System Testing

To test the whole system we use the da_vinci_tb.v, the result should. Be dumped into the fibonacci.dat file. We compare the golden expected fibonacci file along with the dumped file.

Expected transcript output:



Due to not having enough time to review every control signal for the control unit implementation, my testing for the whole system failed after multiple attempts. A missing ‘on’ bit is my best assumption of the mistakes made at this point.

XIII. CONCLUSION

This project resulted in the implementation of many important components of a computer system however successful implementation of the system as a whole failed due to inability to implement the control unit successfully due to lack of time to do so. I learned a lot from this project, my Verilog language skills highly increased and my understanding of each component and their relations and gate level connections is much better.

XIV. REFERENCES

- Digital Design (4th Edition) by M. Morris Mano and Michael D. Ciletti .
- - Computer Organization and Design (5th edition) by David A. Patterson & John L. Hennessy.
- - CS147 Lectures and Lab codes SJSU Spring 2019