Mixed Model of a Computer System in Verilog

Joseph Nguyen
San Jose State University
College of Science, Computer Science
joseph.q.nguyen@sjsu.edu

***Abstract*—this report explores the implementation of a mixed computer system in Verilog. The computer system will be very similar to the one in Project 2, in which we have a computer system that can read assembly code and either read or data. The only difference is that most of the components will be created using logic gates only.**

I. Introduction

The objective of this project is to understand the deep architecture of a computer system. In order to do so, we must go through the tedious task of creating the most of the system components using only gate level operations. The computer system will do exactly the same task as in Project 2; in fact, the output for the tests will be exactly the same. One thing to note is the computer system in Project was developed in the behavioral level, not the gate level.

II. Requirements for the System

A. *Arithmetic Logic Unit with Zero Output*

This ALU model will behave the same as the one in Project 2. The only difference is that this ALU be created using only logic gates.

B. *32 X 32 Register File*

The register file that will be the same as in Project 2; however, the component will be created using register that are implemented at the gate level using logic gates

C. *64Mb Memory*

Very similar to the memory that was in Project 2, this will not be created in the gate level.

D. *Control Unit*

This control unit will be similar to the one in Project 2. While not being a gate level implementation, the control unit is not exactly the same as in Project 2. Instead of computing data, the control unit will send control signals to the data path.

E. *Data Path*

The data path will be responsible for computing the data that is requested by the instructions. This data path will be purely be made of gates. The data path will take signals from the control to decide what to do with the given instruction.

F. *Processor*

This 32-bit processor will be similar to the one in Project 2. This will not be implemented in the gate level

G. *Various Gate Level Components*

The individual gate level components are crucial to the implementation of the bigger components. These smaller components will include 32-bit Fuller Adder, 32-bit Multiplier, 32-bit Shifter, 32-bit logic gates, and more.

H. *DaVinci*

As with Project 2, the name of this computer system is called DaVinci and it can hold a very small instruction set named CS147DV.

I. *CS147DV Instruction Set*

The same instruction set from Project 2, will have R, I, And J-Type instructions

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>

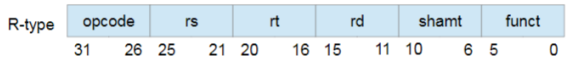


Fig. 1. CS147DV R-Type Instructions

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>

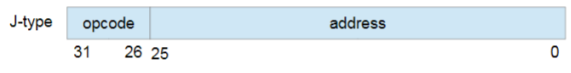
13

Fig. 2. CS147DV I-Type Instructions

Name	Mnemonic	Format	Operation	OpCode
Jump to address	jmp	J	PC = JumpAddress	0x02
Jump and Link	jal	J	R[31] = PC + 1; PC = JumpAddress	0x03
Push to Stack	push	J	M[$sp] = R[0] $sp = $sp - 1	0x1b
Pop from Stack	pop	J	$sp = $sp + 1 R[0] = M[$sp]	0x1c

JumpAddress = { 6'b0, address } // zero extend for 6 bit

Coding format: <mnemonic> <address>



14

Fig. 3. CS147DV J-Type Instructions

III. Design of Ripple Carry Adder/Subtractor

The Ripple Carry Adder/Subtractor is a very important component of the ALU because it is what defines the addition and subtraction operation. It takes two operands and a signal, SnA were 0 is addition and 1 is subtraction. The first step in creating the ripple carry adder is to create the half adder. The half adder takes an XOR and AND gate.

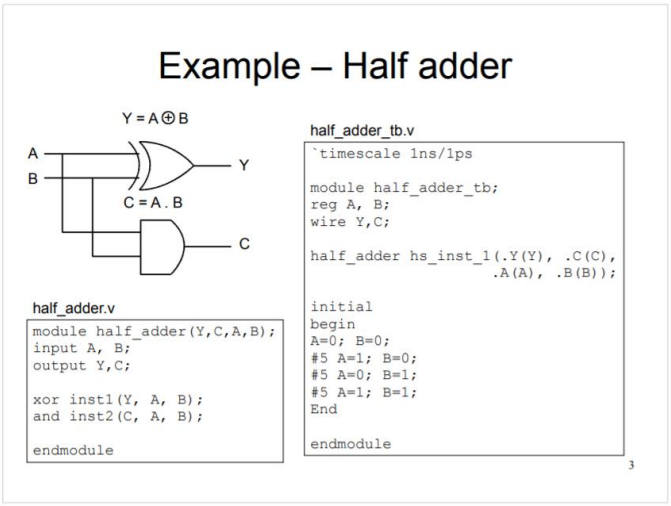


Fig. 4. Gate level implementation of half adder

Next we create the full adder. The full adder takes two half adders and an OR gate.

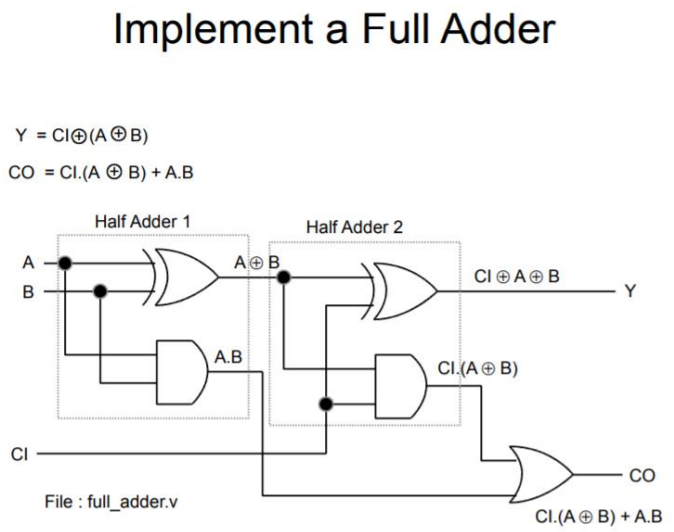


Fig. 5. Gate level implementation of full adder

```
module FULL_ADDER(S, CO, A, B, CI);
    output S, CO;
    input A, B, CI;
    wire H1_Y, H1_C, H2_C;

    HALF_ADDER h1_inst(.Y(H1_Y), .C(H1_C), .A(A), .B(B));

    HALF_ADDER h2_inst(.Y(S), .C(H2_C), .A(H1_Y), .B(CI));

    or or_inst(CO, H1_C, H2_C);
endmodule
```

Fig. 6. Verilog code for full adder

Now, we can create the ripple carry adder/subtracter, it takes 32 fuller adders connected to each other, to aid us in the implementation we will use a 32 iteration generate loop instead of creating 32 full adders. We will also general XOR gates to deal with the SnA signal

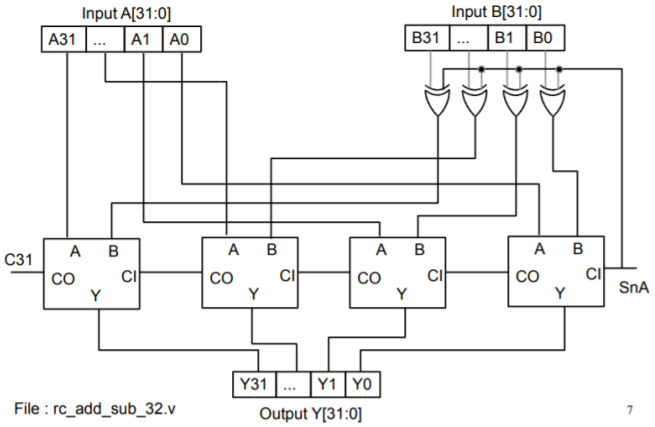


Fig. 7. Gate level implementation of ripple carry adder/subtractor

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

    wire [`DATA_INDEX_LIMIT:0] S;
    wire CO;

    wire [`DATA_INDEX_LIMIT:0] CO_STAGE, B_SnA;

    xor xor_inst(B_SnA[0], B[0], SnA);
    FULL_ADDER fa_inst(S[0], CO_STAGE[0], A[0], B_SnA[0], SnA);
    genvar i;
    generate
        for (i = 1; i < 32; i = i + 1)
        begin
                xor xor_inst(B_SnA[i], B[i], SnA);
                FULL_ADDER fa_inst2(S[i], CO_STAGE[i], A[i], B_SnA[i], CO_STAGE[i - 1]);
        end
    endgenerate
    buf buf_inst(CO, CO_STAGE[31]);
endmodule
```

Fig. 8. Gate level implementation of ripple carry adder/subtractor

IV. Design and Signed Multiplier

The next component is the signed multiplier. This component is important because it defines the multplication operation for the ALU. The first step is to create gate level multiplexors. The multiplexors allow is to choose values based on a signal. To create a one bit multiplexor, we use two AND gates, one NOT and one OR gate.

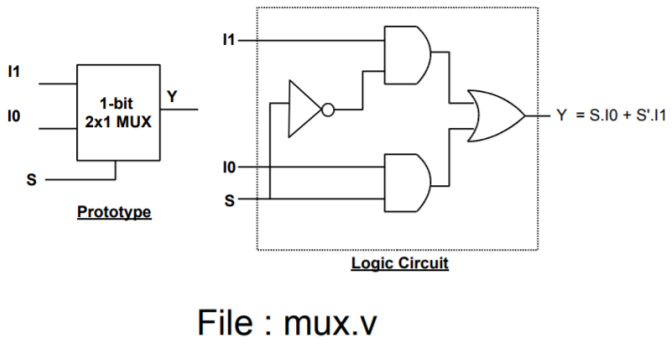
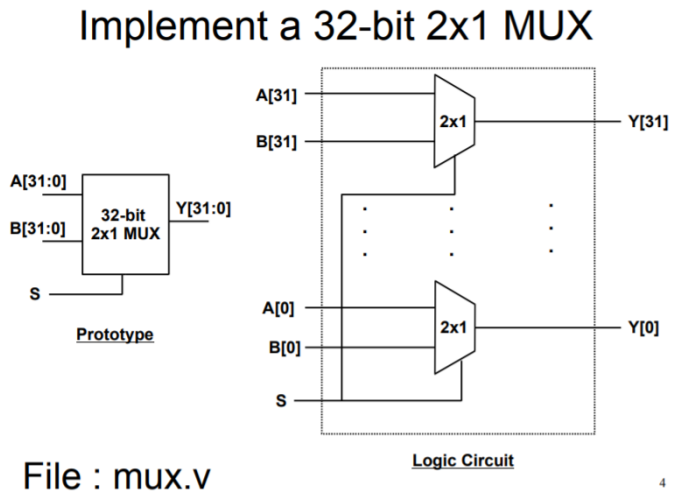


Fig. 9. Gate level implementation of 1-bit mux

```
// 1-bit mux
module MUX1_2x1(Y, I0, I1, S);
    //output list
    output Y;
    //input list
    input I0, I1, S;

    wire S_NOT_AND_I0, S_AND_I1, S_NOT;

    not (S_NOT, S);
    and (S_NOT_AND_I0, S_NOT, I0);
    and (S_AND_I1, S, I1);
    or (Y, S_NOT_AND_I0, S_AND_I1);
endmodule
```

Fig. 10. Verilog code of 1-bit mux

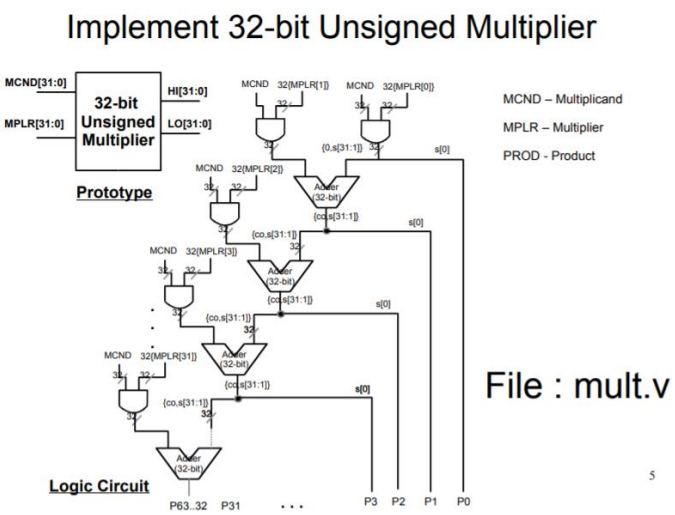
Next, we extend the 1-bit multiplexor to a 32-bit multiplexor.




```
// 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;

    wire [31:0] Y;

    genvar i;
    generate
        for (i = 0; i < 32; i = i + 1)
        begin : mux_loop
            MUX1_2x1 mux(Y[i], I0[i], I1[i], S);
        end
    endgenerate

endmodule
```

Fig. 11. Diagram and Verilog code for gate level 32-bit multplexor

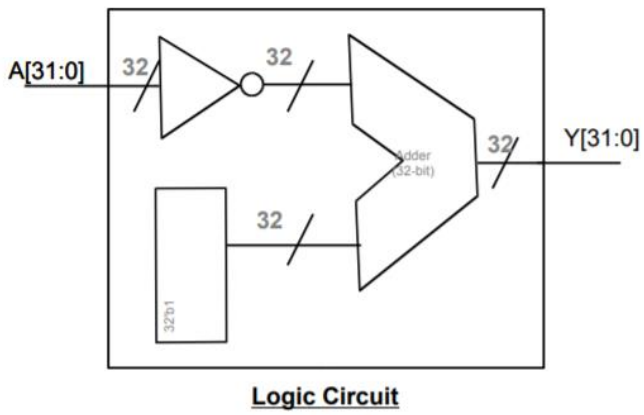
Now, we implement an unsigned 32-bit multiplier that uses multplexors and ripple carry adders.




```
AND32_2x1 and1({AND[0][31:1], LO[0]}, A, {32{B[0]}});
AND32_2x1 and2(AND[1], A, {32{B[1]}});
RC_ADD_SUB_32 rc1({ADDER[0][31:1], LO[1]}, CO[0], AND
genvar i;
generate
    for (i = 2; i < 32; i = i + 1)
    begin : mult_u_loop
        AND32_2x1 and3(AND[i], A, {32{B[i]}});
        RC_ADD_SUB_32 rc2({ADDER[i - 1][31:1], LO[i]}
    end
endgenerate

generate
    for (i = 0; i < 31; i = i + 1)
    begin : buf_loop
        buf buf_inst(HI[i], ADDER[30][i + 1]);
    end
endgenerate
buf buf_inst(HI[31], CO[30]);
```

Fig. 12. Diagram and portion of Verilog code for unsigned mutliplier

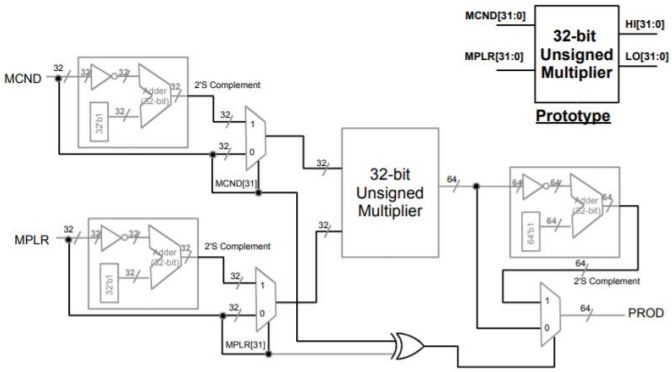
To determine the correct signs of HI and LO outputs of the multiplier, we implement a 2's complement module.




```
// 32-bit two's complement
module TWOSCOMP32(Y,A);
    //output list
    output [31:0] Y;
    //input list
    input [31:0] A;

    wire [31:0] Y;
    wire [31:0] A_INV;
    wire CO;

    INV32_1x1 inv(A_INV, A);
    RC_ADD_SUB_32 adder(Y, CO, A_INV, 32'b1, 1'b0);

endmodule
```

Fig. 13. Diagram and Verilog code for two's complement

With the additional implementation of a 64-bit two's complement module, we can create the signed multiplier by following the diagram.

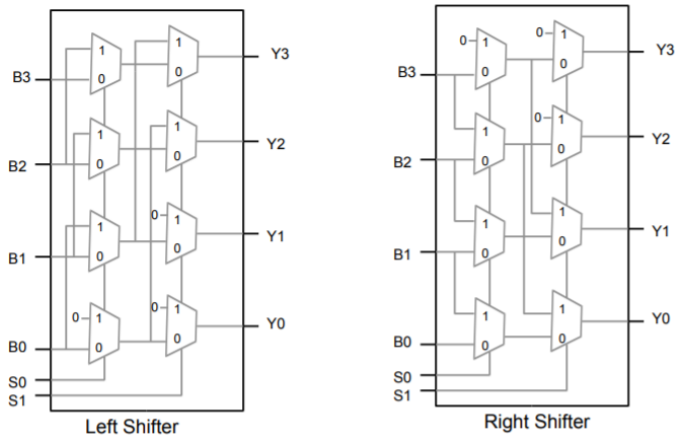



```
TWOSCOMP32 t1(T1, MCND);
MUX32_2x1 m1(M1, MCND, T1, MCND[31]);

TWOSCOMP32 t2(T2, MPLR);
MUX32_2x1 m2(M2, MPLR, T2, MPLR[31]);

MULT32_U mul_u(HI_U, LO_U, M1, M2);
TWOSCOMP64 t3({HI_S, LO_S}, {HI_U, LO_U});

xor xor1(X, MCND[31], MPLR[31]);

MUX64_2x1 mux64({HI, LO}, {HI_U, LO_U}, {HI_S, LO_S}, X);
```

Fig. 14. Diagram and portion of Verilog code for signed multiplier

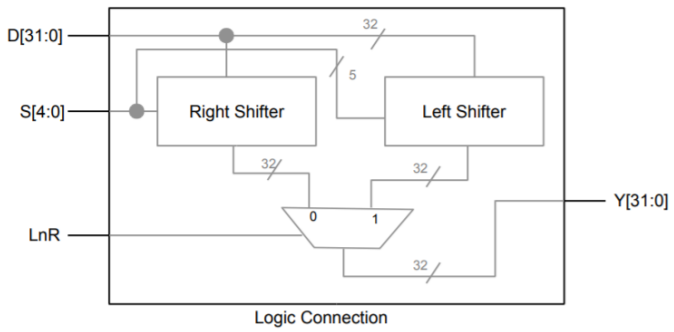
V. Design Barrel Shifter

The next important component is the barrel shifter. The barrel shifter is important for the ALU becaseu it defines the left and right shift operations. To implement this, we must first create the left and right shifters by follwing the diagram and using multiplexors.

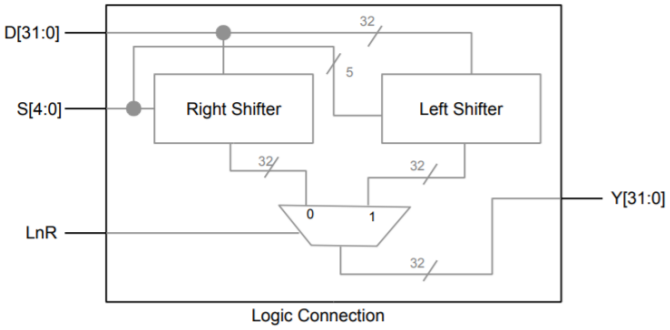



```
module SHIFT32_R(Y,D,S);
    // output list
    output [31:0] Y;
    // input list
    input [31:0] D;
    input [4:0] S;
    // output wire
    wire [31:0] Y;
    // Shift stage outputs
    wire [31:0] S0_OUT, S1_OUT, S2_OUT, S3_OUT;

    MUX32_2x1 S0_mux(S0_OUT, D, {1'b0, D[31:1]}, S[0]);
    MUX32_2x1 s1_mux(S1_OUT, S0_OUT, {2'b0, S0_OUT[31:2]}, S[1]);
    MUX32_2x1 s2_mux(S2_OUT, S1_OUT, {4'b0, S1_OUT[31:4]}, S[2]);
    MUX32_2x1 s3_mux(S3_OUT, S2_OUT, {8'b0, S2_OUT[31:8]}, S[3]);
    MUX32_2x1 s4_mux(Y, S3_OUT, {16'b0, S3_OUT[31:16]}, S[4]);
endmodule

// Left shifter
module SHIFT32_L(Y,D,S);
    // output list
    output [31:0] Y;
    // input list
    input [31:0] D;
    input [4:0] S;
    // output wire
    wire [31:0] Y;
    // Shift stage outputs
    wire [31:0] S0_OUT, S1_OUT, S2_OUT, S3_OUT;

    MUX32_2x1 S0_mux(S0_OUT, D, {D[30:0], 1'b0}, S[0]);
    MUX32_2x1 s1_mux(S1_OUT, S0_OUT, {S0_OUT[29:0] , 2'b0}, S[1]);
    MUX32_2x1 s2_mux(S2_OUT, S1_OUT, {S1_OUT[27:0] , 4'b0}, S[2]);
    MUX32_2x1 s3_mux(S3_OUT, S2_OUT, {S2_OUT[23:0] , 8'b0}, S[3]);
    MUX32_2x1 s4_mux(Y, S3_OUT, {S3_OUT[15:0] , 16'b0}, S[4]);
endmodule
```

Fig. 15. Diagram and Verilog code for left and right shifters



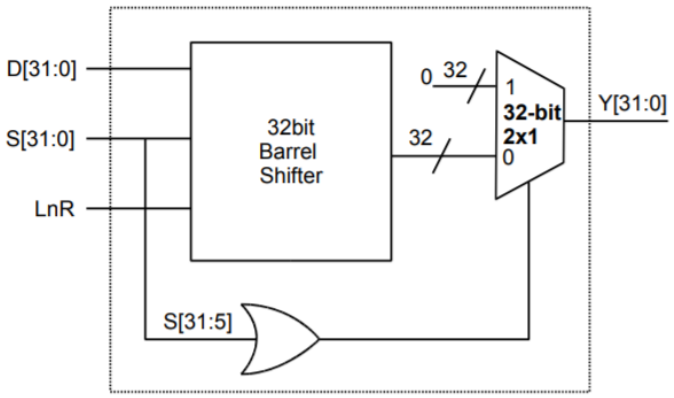
Next, we will implement the 32-bit barrel shifter which computes the left and right shift and chooses which value based on the signal LnR.




```
// Shift with control L or R shift
module BARREL_SHIFTER32(Y,D,S, LnR);
    // output list
    output [31:0] Y;
    // input list
    input [31:0] D;
    input [4:0] S;
    input LnR;

    wire [31:0] L, R, Y;

    SHIFT32_L left_shift(L,D,S);
    SHIFT32_R right_shift(R,D,S);
    MUX32_2x1 mux(Y, R, L, LnR);
endmodule
```

Fig. 16. Diagram and Verilog code for 32-bit barrel shifter

Finally, because the barrel shifter is used by the ALU, the shift amount has to also be 32-bits. To make this barrel shifter adapt to the ALU, we take the first five bits as the shift amount, and use the rest as a ZERO flag.



```
// 32-bit shift amount shifter
module SHIFT32(Y,D,S, LnR);
    // output list
    output [31:0] Y;
    // input list
    input [31:0] D;
    input [31:0] S;
    input LnR;

    wire [31:0] OUT, Y;
    wire OR;

    BARREL_SHIFTER32 bs(OUT,D,S[4:0], LnR);

    or or_inst(OR, S[31:5]);

    MUX32_2x1 mux(Y, OUT, 32'b0, OR);
endmodule
```

Fig. 17. Diagram and Verilog code for ALU 32-bit barrel shifter

VI. Design of 32-Bit Logic Gates

The ALU has 32-bit AND, OR, and NOR operations. To create these, we simple create 32 instances of the corresponding logic gates using a generate loop. Verilog supports one bit logic gates so we can create a gate using the appropriate syntax. In addition to the ALU logic operations, we will create a 32-bit Inverter that we can use for other modules.

```
// 32-bit NOR
module NOR32_2x1(Y,A,B);
    //output
    output [31:0] Y;
    //input
    input [31:0] A;
    input [31:0] B;

    genvar i;
    generate
        for (i = 0; i < 32; i = i + 1)
        begin : nor_loop
            nor nor_inst(Y[i], A[i], B[i]);
        end
    endgenerate

endmodule
```

Fig. 18. Verilog code for 32-bit nor, the other modules will be identical

VII. Design of Multiplexors and Line Decoders

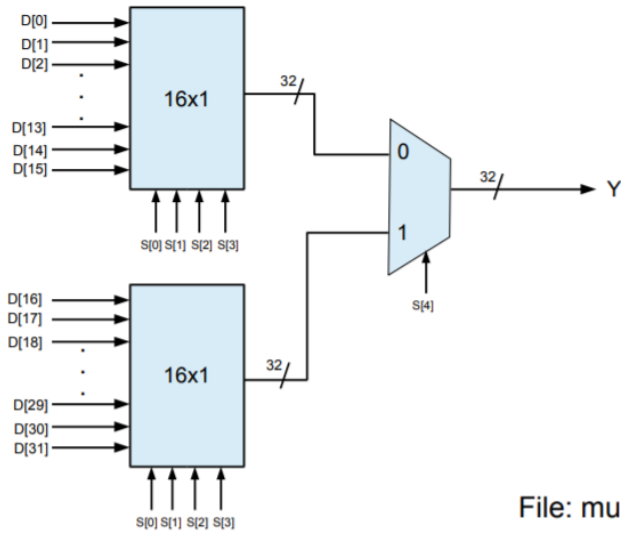
To help us create the other components of the computer system, we must create a 32-bit 32x1 multiplexor as well as a 5-to-32 line decoder. We create them, we start 1-bit and then work our way up.

```
// 32-bit 4x1 mux
module MUX32_4x1(Y, I0, I1, I2, I3, 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 [1:0] S;

    wire [31:0] MUX1;
    wire [31:0] MUX2;

    MUX32_2x1 M1(MUX1, I0, I1, S[0]);
    MUX32_2x1 M2(MUX2, I2, I3, S[0]);

    MUX32_2x1 M3(Y, MUX1, MUX2, S[1]);

endmodule
```

```
// 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, I1, I2, I3, I4, I5, I6, I7;
    input [2:0] S;

    wire [31:0] MUX1;
    wire [31:0] MUX2;

    MUX32_4x1 M1(MUX1, I0, I1, I2, I3, S[1:0]);
    MUX32_4x1 M2(MUX2, I4, I5, I6, I7, S[1:0]);

    MUX32_2x1 M3(Y, MUX1, MUX2, S[2]);

endmodule
```

```
// 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, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13, I14, I15;
    input [3:0] S;

    wire [31:0] MUX1;
    wire [31:0] MUX2;

    MUX32_8x1 M1(MUX1, I0, I1, I2, I3, I4, I5, I6, I7, S[2:0]);
    MUX32_8x1 M2(MUX2, I8, I9, I10, I11, I12, I13, I14, I15, S[2:0]);

    MUX32_2x1 M3(Y, MUX1, MUX2, S[3]);

endmodule
```

Implement 32-bit 32x1 MUX



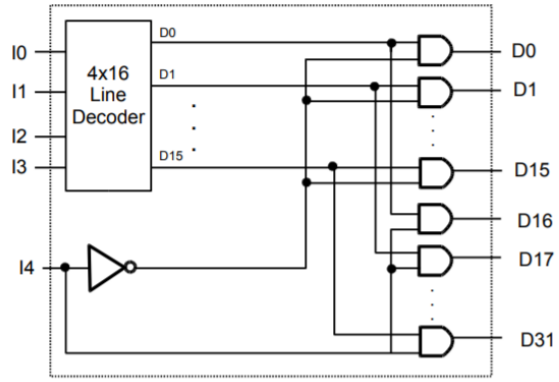
File: mu

```
// 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;

    wire [31:0] MUX1;
    wire [31:0] MUX2;

    MUX32_16x1 M1(MUX1, I0, I1, I2, I3,
                        I4, I5, I6, I7,
                        I8, I9, I10, I11,
                        I12, I13, I14, I15, S[3:0]);
    MUX32_16x1 M2(MUX2, I16, I17, I18, I19,
                        I20, I21, I22, I23,
                        I24, I25, I26, I27,
                        I28, I29, I30, I31, S[3:0]);

    MUX32_2x1 M3(Y, MUX1, MUX2, S[4]);

endmodule
```

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

    wire [3:0] D;
    wire INV1, INV2;

    not n1(INV1, I[1]);
    not n2(INV2, I[0]);

    and a1(D[0], INV1, INV2);
    and a2(D[1], INV1, I[0]);
    and a3(D[2], I[1], INV2);
    and a4(D[3], I[1], I[0]);

endmodule
// 3x8 Line decoder
module DECODER_3x8(D,I);
    // output
    output [7:0] D;
    // input
    input [2:0] I;

    wire [7:0] D;

    wire [3:0] LD; //2x4 line decoder output
    wire INV;

    DECODER_2x4 ld(LD, I[1:0]);
    not n1(INV, I[2]);

    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1)
        begin : loop1
            and a1(D[i], LD[i], INV);
        end
    endgenerate

    generate
        for (i = 4; i < 8; i = i + 1)
        begin : loop2
            and a2(D[i], LD[i - 4], I[2]);
        end
    endgenerate

endmodule
```

```verilog
// 4x16 Line decoder
module DECODER_4x16(D,I);
    // output
    output [15:0] D;
    // input
    input [3:0] I;

    wire [15:0] D;

    wire [7:0] LD;
    wire INV;

    DECODER_3x8 ld(LD, I[2:0]);
    not n1(INV, I[3]);

    genvar i;
    generate
        for (i = 0; i < 8; i = i + 1)
        begin : loop1
            and a1(D[i], LD[i], INV);
        end
    endgenerate

    generate
        for (i = 8; i < 16; i = i + 1)
        begin : loop2
            and a2(D[i], LD[i - 8], I[3]);
        end
    endgenerate
endmodule
```

Implement 5-to-32 line decoder



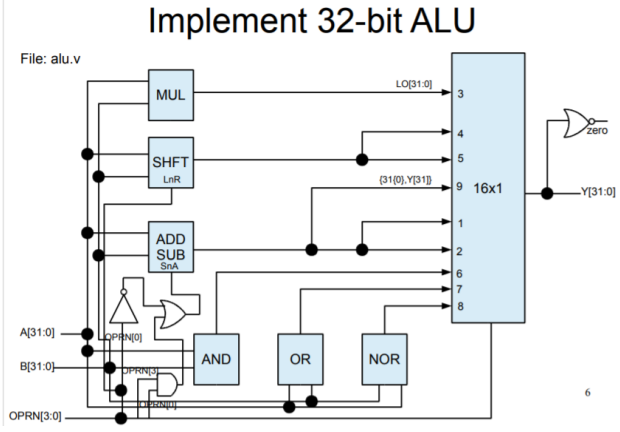

```verilog
// 5x32 Line decoder
module DECODER_5x32(D,I);
    // output
    output [31:0] D;
    // input
    input [4:0] I;

    wire [31:0] D;

    wire [15:0] LD;
    wire INV;

    DECODER_4x16 ld(LD, I[3:0]);
    not n1(INV, I[4]);

    genvar i;
    generate
        for (i = 0; i < 16; i = i + 1)
        begin : loop1
            and a1(D[i], LD[i], INV);
        end
    endgenerate

    generate
        for (i = 16; i < 32; i = i + 1)
        begin : loop2
            and a2(D[i], LD[i - 16], I[4]);
        end
    endgenerate
endmodule
```

Fig. 19. The diagram and Verilog code for multiplexors and decoders

VIII. Design and Implimentation of ALU

The gate level implementation of ALU will take a couple of logic gates and use the ALU components created earlier.

Implement 32-bit ALU



```
MULT32 mul(X, MUL, A, B);

SHIFT32 shift(SHFT, A, B, OPRN[0]); // LnR = OPRN[0]

// SnA
not not_inst(T1, OPRN[0]);
and and_inst(T2, OPRN[3], OPRN[0]);
or or_inst(SnA, T1, T2);

RC_ADD_SUB_32 add_sub(ADD_SUB, XX, A, B, SnA);

AND32_2x1 and_inst2(AND, A, B);
OR32_2x1 or_inst2(OR, A, B);
NOR32_2x1 nor_inst(NOR, A, B);

MUX32_16x1 mux(OUT, X, ADD_SUB, ADD_SUB, MUL, SHFT, SHFT, AND, OR, NOR, {{31

// zero flag
nor nor_inst2(ZERO,
    OUT[0], OUT[1], OUT[2], OUT[3], OUT[4], OUT[5], OUT[6], OUT[7],
    OUT[8], OUT[9], OUT[10], OUT[11], OUT[12], OUT[13], OUT[14], OUT[15],
    OUT[16], OUT[17], OUT[18], OUT[19], OUT[20], OUT[21], OUT[22], OUT[23],
    OUT[24], OUT[25], OUT[26], OUT[27], OUT[28], OUT[29], OUT[30], OUT[31]);
```

Fig. 20. The diagram and portion of Verilog code for gate level ALU

In order to test the ALU, we can use the test bench from Project 2 in order to test.

```
# [TEST] 15 + 3 = 18,0 , got 18,0 ... [PASSED]
# [TEST] 15 - 15 = 0,1 , got 0,1 ... [PASSED]
# [TEST] 15 * 0 = 0,1 , got 0,1 ... [PASSED]
# [TEST] 8 >> 2 = 2,0 , got 2,0 ... [PASSED]
# [TEST] 1 << 2 = 4,0 , got 4,0 ... [PASSED]
# [TEST] 10 & 7 = 2,0 , got 2,0 ... [PASSED]
# [TEST] 10 | 7 = 15,0 , got 15,0 ... [PASSED]
# [TEST] 10 ~| 7 = 4294967280,0 , got 4294967280,0 ... [PASSED]
# [TEST] 100 slt 101 = 1,0 , got 1,0 ... [PASSED]
#
#       Total number of tests           9
#       Total number of pass            9
```

Fig. 21. Transcipt for ALU test bench

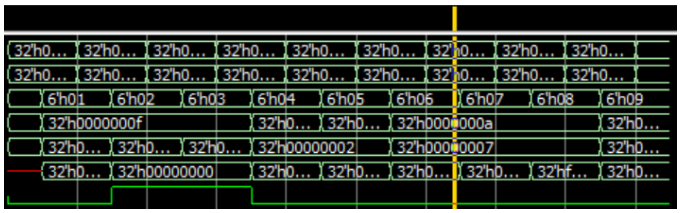


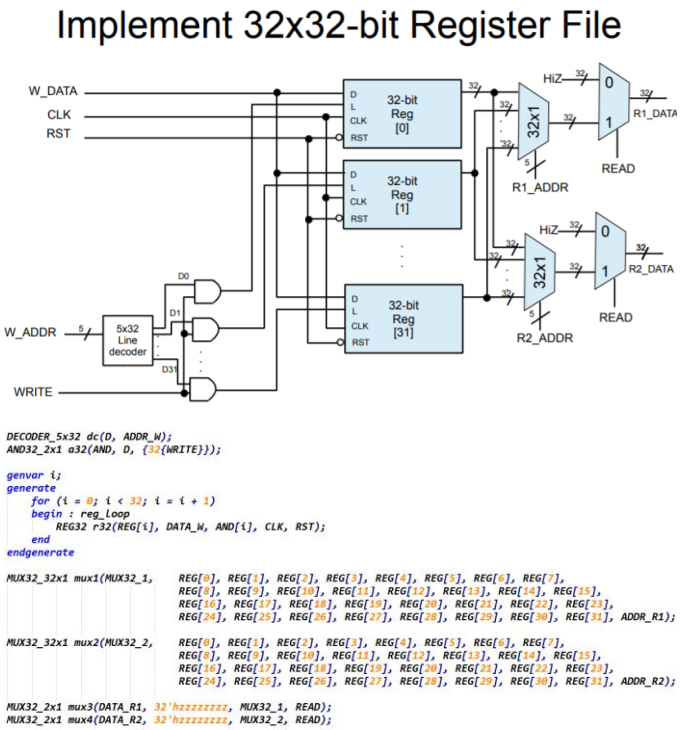
Fig. 22. Wave signals for ALU test bench

IX. Design and Implimentation of Register_File_32X32

The register file will be similar, in terms of functionality, to Project 2's register file but it will be done in the gate level. We will use a 5-to-32 line decoder, multiplexors as well as 32 32-bit registers. The 32-bit registers are created using gate level implementation of SR Latch, D Latch, and D-Flip-Flop.

```
// 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 Q,Qbar;
    wire N1, N2, N3, N4;

    nand nand1(N1, S, C);
    nand nand2(N2, R, C);
    nand nand3(Q, N1, Qbar, nP);
    nand nand4(Qbar, N2, Q, nR);

endmodule

// 1 bit D 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 D_LATCH(Q, Qbar, D, C, nP, nR);
    input D, C;
    input nP, nR;
    output Q,Qbar;

    wire Q, Qbar;

    wire D_INV;
    not not1(D_INV, D);

    SR_LATCH sr(Q, Qbar, D, D_INV, C, nP, nR);

endmodule

// 1 bit flipflop +ve edge,
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module D_FF(Q, Qbar, D, C, nP, nR);
    input D, C;
    input nP, nR;
    output Q,Qbar;

    wire Q, Qbar;
    wire Y, Ybar;

    wire C_INV;
    not not1(C_INV, C);

    D_LATCH d(Y, Ybar, D, C, nP, nR);
    SR_LATCH s(Q, Qbar, Y, Ybar, C_INV, nP, nR);

endmodule

// 1 bit register +ve edge,
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module REG1(Q, Qbar, D, L, C, nP, nR);
    input D, C, L;
    input nP, nR;
    output Q,Qbar;

    wire Q, Qbar;

    wire MUX;
    MUX1_2x1 mux (MUX, Q, D, L);

    D_FF df(Q, Qbar, MUX, C, nP, nR);

endmodule
```

```
// 32-bit registere +ve edge, Reset on RESET=0
module REG32(Q, D, LOAD, CLK, RESET);
    output [31:0] Q;

    input CLK, LOAD;
    input [31:0] D;
    input RESET;

    wire [31:0] Q, X;

    genvar i;
    generate
        for (i = 0; i < 32; i = i + 1)
        begin : reg_loop
            REG1 r(Q[i], X[i], D[i], LOAD, CLK, 1'b1, RESET);
        end
    endgenerate

endmodule
```




```
DECODER_5x32 dc(D, ADDR_W);
AND32_2x1 a32(AND, D, {32{WRITE}});

genvar i;
generate
    for (i = 0; i < 32; i = i + 1)
    begin : reg_loop
        REG32 r32(REG[i], DATA_W, AND[i], CLK, RST);
    end
endgenerate

MUX32_32x1 mux1(MUX32_1,    REG[0], REG[1], REG[2], REG[3], REG[4], REG[5], REG[6], REG[7],
                            REG[8], REG[9], REG[10], REG[11], REG[12], REG[13], REG[14], REG[15],
                            REG[16], REG[17], REG[18], REG[19], REG[20], REG[21], REG[22], REG[23],
                            REG[24], REG[25], REG[26], REG[27], REG[28], REG[29], REG[30], REG[31], ADDR_R1);

MUX32_32x1 mux2(MUX32_2,    REG[0], REG[1], REG[2], REG[3], REG[4], REG[5], REG[6], REG[7],
                            REG[8], REG[9], REG[10], REG[11], REG[12], REG[13], REG[14], REG[15],
                            REG[16], REG[17], REG[18], REG[19], REG[20], REG[21], REG[22], REG[23],
                            REG[24], REG[25], REG[26], REG[27], REG[28], REG[29], REG[30], REG[31], ADDR_R2);

MUX32_2x1 mux3(DATA_R1, 32'hzzzzzzzz, MUX32_1, READ);
MUX32_2x1 mux4(DATA_R2, 32'hzzzzzzzz, MUX32_2, READ);
```

Fig. 23. The diagram and portion of Verilog code for 32x32-bit register file

To test the register file, I used the testbench that was provided and modified it a little. I was able to pass most tests, but I could not get the HiZ to output.

```
[TEST] Read 0, Write 0, expecting 32'hzzzzzzzz, 32'hzzzzzzzz, got xxxxxxxx, xxxxxxxx [FAILED]
[TEST] Read 0, Write 0, expecting 32'hzzzzzzzz, 32'hzzzzzzzz, got xxxxxxxx, xxxxxxxx [FAILED]

      Total number of tests          33
      Total number of pass           32
```

Fig. 24. Transcipt for register file testbench

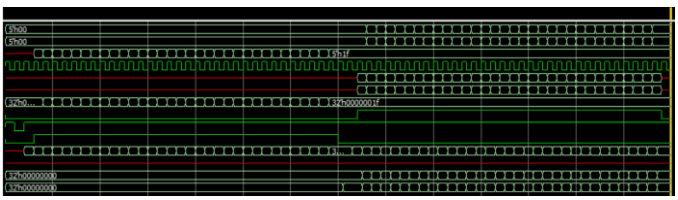


Fig. 25. Wave signals for register file testbench

X. Design and Implimentation of MEMORY

The memory component will be the same as in Project 2, which means it is not created in the gate level. It will take five inputs READ, WRITE, CLK, RST, ADDR and will also take an inout DATA.

```
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 ope
    sram_32x64m[ADDR] = DATA;
end
end
```

Fig. 26. Main Verilog code for memory

We can use the given test bench in order to test and see the memory output.

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

Fig. 27. Transcipt for memory

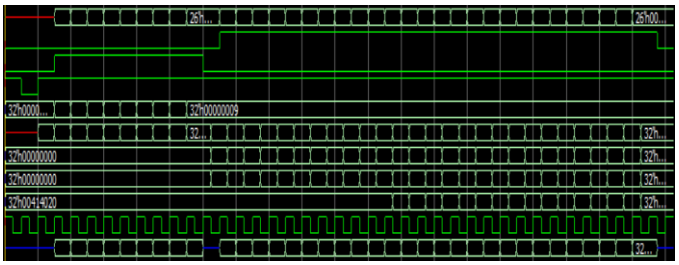
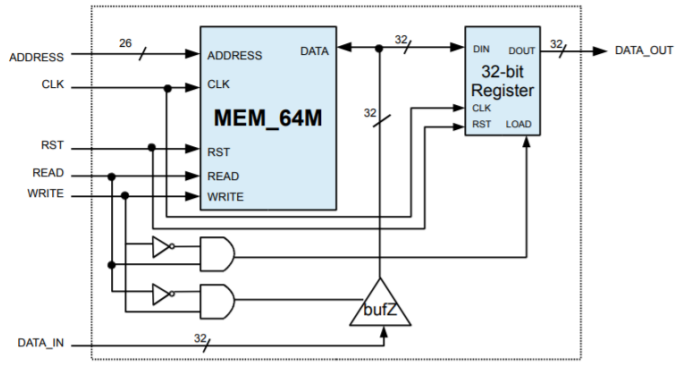


Fig. 28. Wave signals for memory

After creating the 64-bit memory, we must also create a memory wrapper, that connects the memory to a 32-bit register.




```
reg  [`DATA_INDEX_LIMIT:0] DATA_OUT;
wire [`DATA_INDEX_LIMIT:0] DATA;

assign DATA = ((READ===1'b0)&&(WRITE===1'b1))?DATA_IN:{`DATA_WIDTH{1'bz} };

defparam memory_inst.mem_init_file = mem_init_file;
MEMORY_64MB memory_inst(.DATA(DATA), .READ(READ), .WRITE(WRITE),
                        .ADDR(ADDR), .CLK(CLK),   .RST(RST));

initial
begin
DATA_OUT = 32'h00000000;
end

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
```

Fig. 29. Diagram and portion of Verilog code for memeroy wrapper

XI. Design and Implimentation of Control_Unit

The control unit for this project will not be gate level and will be a little different from the control unit in Project 2. The difference is that this control unit does not access the ALU at all. All it will do is output memory read and write signals based on the instruction and also send control signals to the data path based on the instruction. The control unit will get the instruction from the data path. The control unit will also have a state machine that is directly taken from Project 2.

```
always @ (posedge CLK)
begin
    case(STATE)
        `PROC_FETCH : next_state = `PROC_DECODE;
        `PROC_DECODE : next_state = `PROC_EXE;
        `PROC_EXE : next_state = `PROC_MEM;
        `PROC_MEM : next_state = `PROC_WB;
        `PROC_WB : next_state = `PROC_FETCH;
    endcase
    STATE = next_state;
end
```

Fig. 30. Main Verilog code for state machine

For each state in the control unit, there will be a specific control signal generated for each instruction. I was not able to generate the control signal for every instruction because I did not have enough time to. I plan to have each instruction generate a signal that follows the format for Homework 2. For instance, in PROC_FETCH, the control unit must read an instruction from memory, so for all instructions, the same control signal is created and sent to the data path.

```
always @ (posedge CLK)
begin
    case(proc_state)
        `PROC_FETCH :
        begin
            CTRL = 32'h08000020;
            MEM_READ = 1'b1;
            MEM_WRITE = 1'b0;
        end
```

Fig. 31. Starting code for control unit, PROC_FETCH state

XII. Design and Implimentation of Data Path

The data path will take in the control signal from the control unit and an instruction from memory and will generate a path based on the signal. The data path is responsible for compute values from the ALU and do read and write operations between the register file and memory in order to simulate the the process of parsing and excecuting an instruction. We will follow the data path diagram given to us. One important thing to note is that in order to initialize the stack pointer and program counter at gate level, we must create a register that has preset values. I created a module that creates a register that initializes for a VALUE parameter. It does this by creating two registers, one with a zero preset and the other with a one preset. Then it chooses the preset value by using a mutiplexor with the VALUE as the selection input.

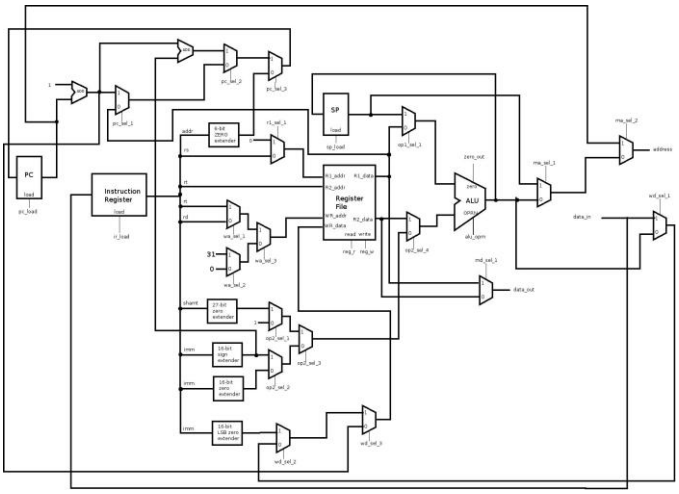


Fig. 32. Complete data path diagram

```
// 32-bit register with preset value
module REG32_PRE(Q, D, LOAD, CLK, RESET, VALUE);
    output [31:0] Q;

    input CLK, LOAD;
    input [31:0] D, VALUE;
    input RESET;

    wire [31:0] Q, X1, X2;
    wire [31:0] B0, B1;

    genvar i;
    generate
        for (i = 0; i < 32; i = i + 1)
        begin : reg_loop
            REG1 r0(B0[i], X1[i], D[i], LOAD, CLK, 1'b1, RESET);
            REG1 r1(B1[i], X2[i], D[i], LOAD, CLK, RESET, 1'b1);
            MUX1_2x1 m(Q[i], B0[i], B1[i], VALUE[i]);
        end
    endgenerate
endmodule
```

Fig. 33. Implementation of preset register

To test the preset register, I created a testbench in logic.v that loads the program counter and increments it every clock cycle. The output wave shows correct values.

```
`timescale 1ns/1ps

module LOGIC_TB;
reg LOAD, C, RST;
wire [31:0] PC, ADD1;
wire x;

REG32_PRE pc(PC, ADD1, LOAD, C, RST, 32'h00001000);
RC_ADD_SUB_32 rc(ADD1, x, PC, 32'b1, 1'b0);


initial
begin
    C = 0; LOAD = 0; RST = 0;
    # 10 LOAD = 1; RST = 1;
    # 10 LOAD = 0;
    # 10 LOAD = 1;
    # 10 LOAD = 0;
    # 10 LOAD = 1;

    #20 $stop;
end

always
begin
    #10 C = ~C;
end

endmodule
```

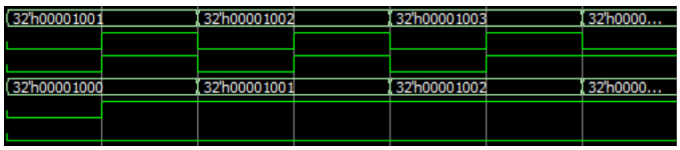


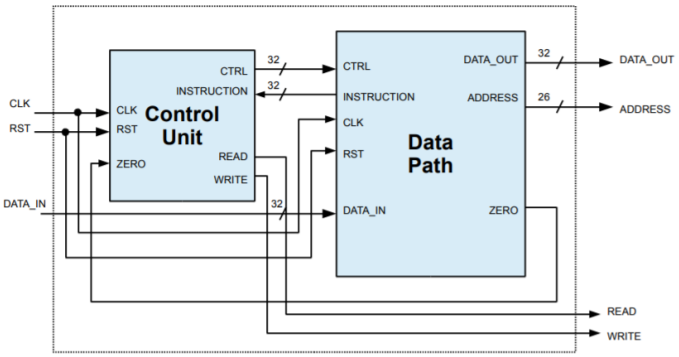
Fig. 34. Verilog code and output wave for preset register testbench

```
FULL_ADDER a1(ADD1, x, PC_LOAD, 1, 0);
FULL_ADDER a2(ADD2, x, ADD1, BIT_16_SIGN_EXT, 0);

REG32_PRE pc(PC_LOAD, PC_SEL_3, CTRL[0], CLK, RST, `INST_START_ADDR);
MUX32_2x1 m2(PC_SEL_1, R1_DATA, ADD1, CTRL[1]);
MUX32_2x1 m3(PC_SEL_2, PC_SEL_1, ADD2, CTRL[2]);
MUX32_2x1 m4(PC_SEL_3, BIT_6_ZERO_EXT, PC_SEL_2, CTRL[3]);

REG32 ir(IR_LOAD, DATA_IN, CTRL[4], CLK, RESET);
BUF32_2x1 buf3({OPCODE, RS, RT, RD, SHAMT, FUNCT, IMM, ADDR}, IR_LOAD);

BUF32_2x1 buf4(BIT_6_ZERO_EXT, {6'b0, ADDR});
BUF32_2x1 buf5(BIT_27_ZERO_EXT,{27'b0, SHAMT});
BUF32_2x1 buf6(BIT_16_SIGN_EXT, {{16{IMM[15]}}, IMM});
BUF32_2x1 buf7(BIT_16_ZERO_EXT, {16'b0, IMM});
BUF32_2x1 buf8(BIT_16_LSB_ZERO_EXT, {IMM, 16'b0});

MUX32_2x1 m6(R1_SEL_1, RS, 1'b0, CTRL[7]);
REGISTER_FILE_32x32 rf(R1_DATA, R2_DATA, R1_SEL_1, RT,
                       WA_SEL_3, WD_SEL_3, CTRL[8], CTRL[9], CLK, RST);
MUX32_2x1 m7(WA_SEL_1, RD, RT, CTRL[10]);
MUX32_2x1 m8(WA_SEL_2, 0, 31, CTRL[11]);
MUX32_2x1 m9(WA_SEL_3, WA_SEL_2, WA_SEL_1, CTRL[12]);
MUX32_2x1 m10(WD_SEL_1, ALU, DATA_IN, CTRL[13]);
MUX32_2x1 m11(WD_SEL_2, WD_SEL_1, BIT_16_LSB_ZERO_EXT, CTRL[14]);
MUX32_2x1 m12(WD_SEL_3, ADD1, WD_SEL_2, CTRL[15]);
REG32_PRE sp(SP_LOAD, ALU, CTRL[16], CLK, RESET, `INIT_STACK_POINTER);
MUX32_2x1 m14(OP1_SEL_1, R1_DATA, SP_LOAD, CTRL[17]);
MUX32_2x1 m15(OP2_SEL_1, 1, BIT_27_ZERO_EXT, CTRL[18]);
MUX32_2x1 m16(OP2_SEL_2, BIT_16_ZERO_EXT, BIT_16_SIGN_EXT, CTRL[19]);
MUX32_2x1 m17(OP2_SEL_3, OP2_SEL_2, OP2_SEL_1, CTRL[20]);
MUX32_2x1 m18(OP2_SEL_4, OP2_SEL_3, R2_DATA, CTRL[21]);
ALU alu(ALU, ZERO, OP2_SEL_4, OP1_SEL_1, CTRL[25:22]);
MUX32_2x1 m19(MA_SEL_1, ALU, SP_LOAD, CTRL[26]);
MUX32_2x1 m20(MA_SEL_2, MA_SEL_1, PC_LOAD, CTRL[27]);
MUX32_2x1 m21(DATA_OUT, R2_DATA, R1_DATA, CTRL[28]);
```

Fig. 35. Portion of Verilog code for data path

XIII. Design and Implimentation of Processor

The processor for this project will be a little different than the one from Project 2. Because the data path already holds the ALU and Register File, the processor will not need to implement them. Instead, the processor will implement control unit and data path. The processor will be responsible for allowing the control unit and data path to communicate to each other.

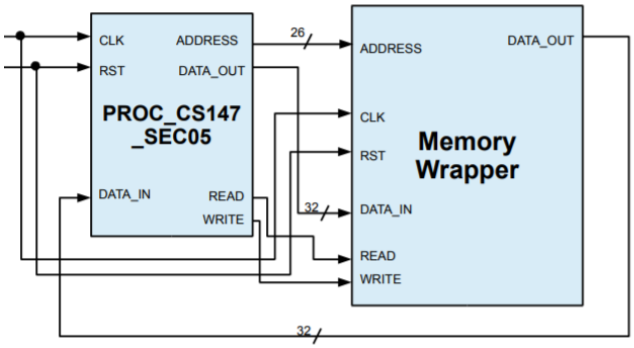



```
module PROC_CS147_SEC05(DATA_OUT, ADDR, DATA_IN, READ, WRITE, CLK, RST);
// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output [`DATA_INDEX_LIMIT:0] DATA_OUT;
output READ, WRITE;
// input list
input  CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_IN;

// net section
wire zero;
wire [`CTRL_WIDTH_INDEX_LIMIT:0] ctrl;
wire [`DATA_INDEX_LIMIT:0] INSTRUCTION;

// instantiation section
// Control unit
CONTROL_UNIT cu_inst (.CTRL(ctrl), .READ(READ), .WRITE(WRITE),
                      .ZERO(zero), .INSTRUCTION(INSTRUCTION),
                      .CLK(CLK),   .RST(RST));

// data path
DATA_PATH    data_path_inst (.DATA_OUT(DATA_OUT),  .INSTRUCTION(INSTRUCTION), .DATA_IN(DATA_IN), .ADDR(ADDR), .ZERO(zero),
                             .CTRL(ctrl),  .CLK(CLK),   .RST(RST));

endmodule;
```

Fig. 36. Diagram and Verilog code for processor

XIV. Design and Implimentation of Da_Vinci

The final component of this project is the whole computer system. This system will hold the memory wrapper and the processor and is responsible for making them communicate with each other.




```
module DA_VINCI (MEM_DATA_OUT, MEM_DATA_IN, ADDR, READ, WRITE, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output [`DATA_INDEX_LIMIT:0] MEM_DATA_OUT, MEM_DATA_IN;
output READ, WRITE;
// input list
input  CLK, RST;

// Instance section
// Processor instanceIN
PROC_CS147_SEC05 processor_inst(.DATA_IN(MEM_DATA_OUT),    .DATA_OUT(MEM_DATA_IN),
                                .ADDR(ADDR), .READ(READ),
                                .WRITE(WRITE), .CLK(CLK),    .RST(RST));
// memory instance
defparam memory_inst.mem_init_file = mem_init_file;
MEMORY_WRAPPER memory_inst(.DATA_OUT(MEM_DATA_OUT), .DATA_IN(MEM_DATA_IN),
                           .READ(READ), .WRITE(WRITE),
                           .ADDR(ADDR), .CLK(CLK),    .RST(RST));
endmodule;
```

Fig. 37. Diagram and Verilog code DaVinci

XV. Test Strategy and Test Implementation

Because I was not able to create the control signals in the control unit, this computer system is incomplete and will not work. However, most of the lower level components were able to create correct output which allows the ALU and Register file to work properly, all I need to do now is create the control signals and make sure that the data path is correct and my computer system should work.

```
# @      30ns -> [0Xzzzzzzzz]
# @      80ns -> [0Xzzzzzzzz]
# @     130ns -> [0Xzzzzzzzz]
# @     180ns -> [0Xzzzzzzzz]
# @     230ns -> [0Xzzzzzzzz]
# @     280ns -> [0Xzzzzzzzz]
# @     330ns -> [0Xzzzzzzzz]
# @     380ns -> [0Xzzzzzzzz]
# @     430ns -> [0Xzzzzzzzz]
# @     480ns -> [0Xzzzzzzzz]
# @     530ns -> [0Xzzzzzzzz]
# @     580ns -> [0Xzzzzzzzz]
```

Fig. 38. Transcript for DaVinci testbench

XVI. Conculsion

Overall, I wish that I was able to have completed to project in time. The project was tedious and stressful, but I was able to learn a lot more about Verilog because of it. The concepts of wire and register values were pretty confusing to me but this project has allowed me to understand it more clearly. I think that it would have been nice if I was able to work with more students to create test benches and generate control signals because that would have made the project funner and more successful. I am glad that I was able to implement an ALU and Register file at the gate level and I hope to finish this implementation one day