Project 3 – Gate Level Implementation of DaVinci 1.0

Catalina Lamboglia
San Jose State University
catalina.lamboglia@sjsu.edu

Introduction

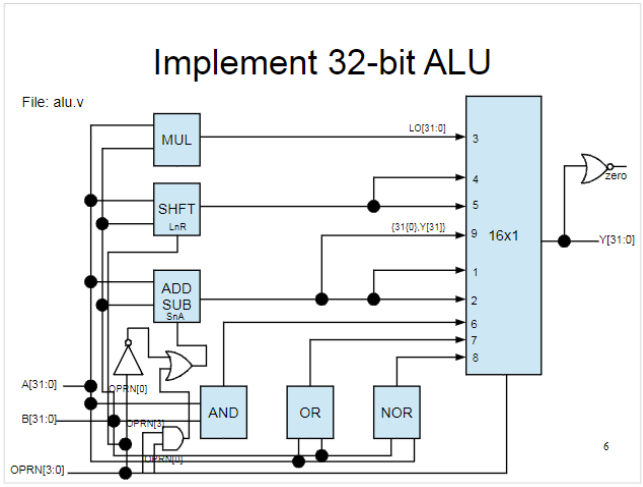
This report will cover the entire assembly, implementation, and testing of the DaVinci 1.0 System. This implementation will be done at the gate level.

STRUCTURE:

1. Assembly of ALU and Testing
2. Register File Assembly and Testing
3. Control unit design
4. Memory unit
5. Quick discussion of entire system assembly and testing
6. *Conclusion*

I. ASSEMBLY OF ALU AND TESTING

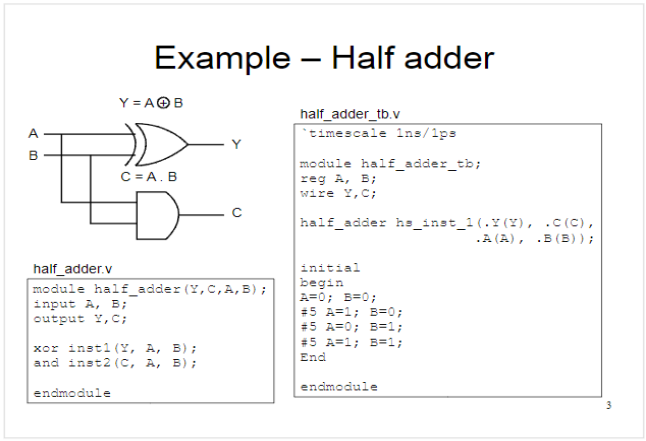
(All subsequent images of circuits/schematics are credited to Kaushik Patra, unless otherwise stated. Format will be as follows "Kaushik Patra, Lab #, Slide #". These labs and slides can be found within the CS147 canvas course shell respective to his class and your section. Images of code and waveforms are generated by me, not all of the code is mine, some is prewritten by Kaushik Patra. Essentially, if it's an output/input declaration or comment header/module parameter listing/declaration it was written by Patra.)



(Kaushik Patra, Lab 14, Slide 3)

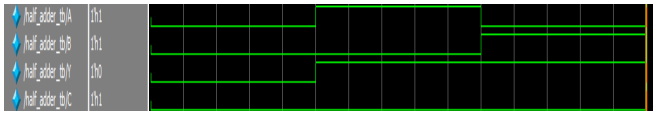
Above, the gate level ALU implementation is shown. The ALU requires a multiplier, shifter, adder/subtractor, 32 bit 2x1 and, or, and nor components for its operations. The primary function of the ALU is to provide the system with arithmetic and logic operations. CS147DV's first nine operations are handled by the ALU.

A. The Adder/Subtractor



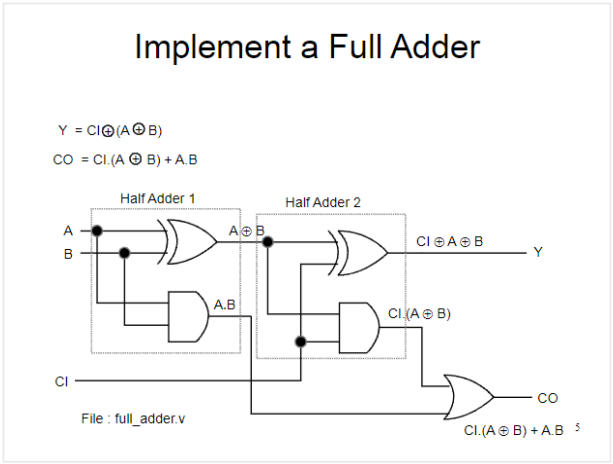
(Kaushik Patra, Lab 11, Slide 2)

The half-adder's implementation was provided by the professor as well as the testing. The xor gate provides the logic for bit addition while the and provides the carried out bit.



Test cases:
(A+B=Y+CO)
0+0=0+0
1+0=1+0
0+1=1+0
1+1=0+1

Next is to assemble a full adder using two half-adders.



(Kaushik Patra, Lab 11, Slide 3)

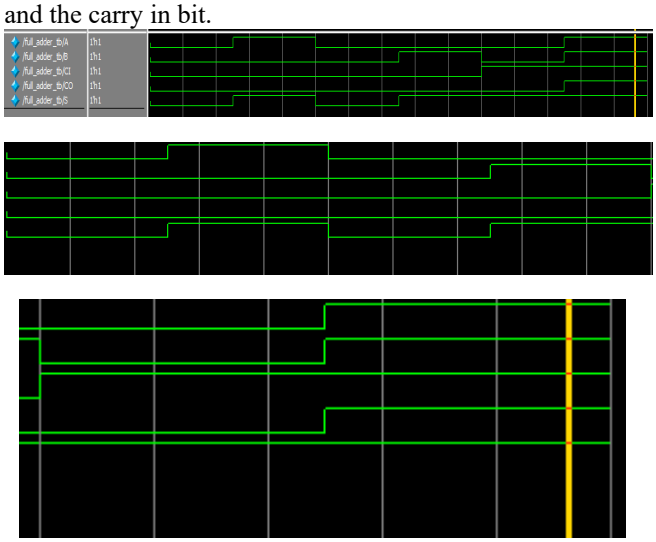
The full adder allows for a complete bit addition with a carry in bit and a carry out from the completed bit addition.

```
// Name: full_adder.v
// Module: FULL_ADDER
//
// Output: S : Sum
//         CO : Carry Out
//
// Input: A : Bit 1
//        B : Bit 2
//        CI : Carry In
//
// Notes: 1-bit full adder implementaiton.
//
//
// Revision History:
//
// Version      Date            Who             email                   note
//-----------------------------------------------------------------------------------------
//  1.0     Sep 10, 2014        Kaushik Patra   kpatra@sjsu.edu         Initial creation
//-----------------------------------------------------------------------------------------
`include "prj_definition.v"

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

wire C, Y, S;
//wire CI, CO;
wire tempC, tempY;
assign S = Y;
HALF_ADDER ha_inst_1(.Y(tempY), .C(tempC), .A(A), .B(B));
HALF_ADDER ha_inst_2(.Y(Y), .C(C), .A(tempY), .B(CI));

or inst1(CO, C, tempC);

endmodule;
```

```
`timescale 1ns/1ps

module full_adder_tb;
reg A, B, CI;
wire CO, S;

// module FULL_ADDER(S, CO, A, B, CI);
FULL_ADDER fa_inst_1(.S(S), .CO(CO), .A(A), .B(B), .CI(CI));

initial
begin
A=0; B=0; CI=0;
#5 A=1; B=0; CI=0;
#5 A=0; B=0; CI=0;
#5 A=0; B=1; CI=0;
#5 A=0; B=0; CI=1;
#5 A=1; B=1; CI=1;
#5 ;
end

endmodule
```

(My implementation and testing. Unless otherwise stated, the code is mine. Header comments are credited to Kaushik Patra as well as the initial set up for module names, parameters, and output/input/wires.)

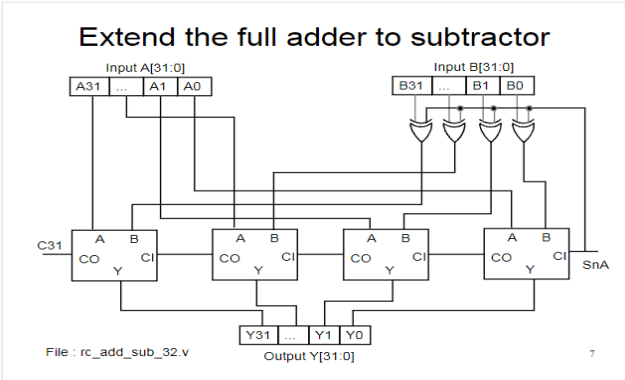
To test, all you must do is drive the two addition operands and the carry in bit.



Line in waveform order (top to bottom): A, B, CI, CO, S

Test cases:
(A+B+CI=S+CO)
0+0 +0 = 0+0
1+0+0=1+0
0+0+0=0+0
0+1+0=1+0
0+0+1=1+0
1+1+1=1+1

Now that you have a working full adder, you can put them in sequence to generate an adder/subtractor of any bit length. For our purposes, we are only making 32-bit length addition/subtraction operations so you will need 32 full adders in sequence as follows.



(Kaushik Patra, Lab 11, Slide 4)

You can do the intermediary process of simply making an adder (the same as above, sans the xors for every B input bit.) but for my implementation I simply made the full adder/subtractor all at once and tested it using SnA.

```
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;

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



wire B0;
wire B1;
wire B2;
wire B3;
wire B4;
wire B5;
wire B6;
wire B7;
wire B8;
wire B9;
wire B10;
wire B11;
wire B12;
wire B13;
wire B14;
wire B15;
```

```
wire B30;
wire B31;

wire CO0;
wire CO1;
wire CO2;
wire CO3;
wire CO4;
wire CO5;
wire CO6;
wire CO7;
wire CO8;
wire CO9;
wire CO10;
wire CO11;
wire CO12;
wire CO13;
wire CO14;
wire CO15;
wire CO16;
wire CO17;
wire CO18;
wire CO19;
wire CO20;
wire CO21;
wire CO22;
wire CO23;
wire CO24;
wire CO25;
wire CO26;
wire CO27;
wire CO28;
wire CO29;
wire CO30;
wire CO31;
```

You can simply make these within a for-loop but my method of code output/generation was scripting with java so I have 32 of the B wires (B0....B31), this will be a pattern throughout my implementations. Essentially, if these wire segments (the implementation of the large sequences of wires) are absent then assume they are generated. I will say when I omit them and a statement such as “implement 32 wire (name)” or some such when this occurs. I will adopt a simpler method of this later by using 32 bit wires and manipulating each bit separately later.

```
xor xor0(B0, SnA, B[0]);
xor xor1(B1, SnA, B[1]);
xor xor2(B2, SnA, B[2]);
xor xor3(B3, SnA, B[3]);
xor xor4(B4, SnA, B[4]);
xor xor5(B5, SnA, B[5]);
xor xor6(B6, SnA, B[6]);
xor xor7(B7, SnA, B[7]);
xor xor8(B8, SnA, B[8]);
xor xor9(B9, SnA, B[9]);
xor xor10(B10, SnA, B[10]);
xor xor11(B11, SnA, B[11]);
xor xor12(B12, SnA, B[12]);
xor xor13(B13, SnA, B[13]);
xor xor14(B14, SnA, B[14]);
xor xor15(B15, SnA, B[15]);
xor xor16(B16, SnA, B[16]);
xor xor17(B17, SnA, B[17]);
xor xor18(B18, SnA, B[18]);
xor xor19(B19, SnA, B[19]);
xor xor20(B20, SnA, B[20]);
xor xor21(B21, SnA, B[21]);
xor xor22(B22, SnA, B[22]);
xor xor23(B23, SnA, B[23]);
xor xor24(B24, SnA, B[24]);
xor xor25(B25, SnA, B[25]);
xor xor26(B26, SnA, B[26]);
xor xor27(B27, SnA, B[27]);
xor xor28(B28, SnA, B[28]);
xor xor29(B29, SnA, B[29]);
xor xor30(B30, SnA, B[30]);
xor xor31(B31, SnA, B[31]);
```

The generation of the “negative” bits if designated by SnA.

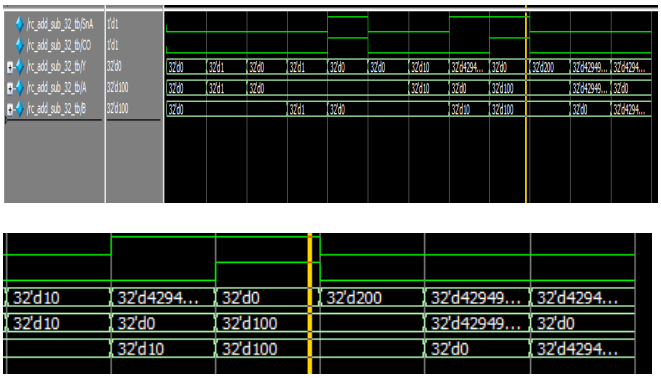
```
415  FULL_ADDER fa_inst_0(.S(Y[0]), .CO(CO0), .A(A[0]), .B(B0), .CI(SnA));
416  FULL_ADDER fa_inst_1(.S(Y[1]), .CO(CO1), .A(A[1]), .B(B1), .CI(CO0));
417  FULL_ADDER fa_inst_2(.S(Y[2]), .CO(CO2), .A(A[2]), .B(B2), .CI(CO1));
418  FULL_ADDER fa_inst_3(.S(Y[3]), .CO(CO3), .A(A[3]), .B(B3), .CI(CO2));
419  FULL_ADDER fa_inst_4(.S(Y[4]), .CO(CO4), .A(A[4]), .B(B4), .CI(CO3));
420  FULL_ADDER fa_inst_5(.S(Y[5]), .CO(CO5), .A(A[5]), .B(B5), .CI(CO4));
421  FULL_ADDER fa_inst_6(.S(Y[6]), .CO(CO6), .A(A[6]), .B(B6), .CI(CO5));
422  FULL_ADDER fa_inst_7(.S(Y[7]), .CO(CO7), .A(A[7]), .B(B7), .CI(CO6));
423  FULL_ADDER fa_inst_8(.S(Y[8]), .CO(CO8), .A(A[8]), .B(B8), .CI(CO7));
424  FULL_ADDER fa_inst_9(.S(Y[9]), .CO(CO9), .A(A[9]), .B(B9), .CI(CO8));
425  FULL_ADDER fa_inst_10(.S(Y[10]), .CO(CO10), .A(A[10]), .B(B10), .CI(CO9));
426  FULL_ADDER fa_inst_11(.S(Y[11]), .CO(CO11), .A(A[11]), .B(B11), .CI(CO10));
427  FULL_ADDER fa_inst_12(.S(Y[12]), .CO(CO12), .A(A[12]), .B(B12), .CI(CO11));
428  FULL_ADDER fa_inst_13(.S(Y[13]), .CO(CO13), .A(A[13]), .B(B13), .CI(CO12));
429  FULL_ADDER fa_inst_14(.S(Y[14]), .CO(CO14), .A(A[14]), .B(B14), .CI(CO13));
430  FULL_ADDER fa_inst_15(.S(Y[15]), .CO(CO15), .A(A[15]), .B(B15), .CI(CO14));
431  FULL_ADDER fa_inst_16(.S(Y[16]), .CO(CO16), .A(A[16]), .B(B16), .CI(CO15));
432  FULL_ADDER fa_inst_17(.S(Y[17]), .CO(CO17), .A(A[17]), .B(B17), .CI(CO16));
433  FULL_ADDER fa_inst_18(.S(Y[18]), .CO(CO18), .A(A[18]), .B(B18), .CI(CO17));
434  FULL_ADDER fa_inst_19(.S(Y[19]), .CO(CO19), .A(A[19]), .B(B19), .CI(CO18));
435  FULL_ADDER fa_inst_20(.S(Y[20]), .CO(CO20), .A(A[20]), .B(B20), .CI(CO19));
436  FULL_ADDER fa_inst_21(.S(Y[21]), .CO(CO21), .A(A[21]), .B(B21), .CI(CO20));
437  FULL_ADDER fa_inst_22(.S(Y[22]), .CO(CO22), .A(A[22]), .B(B22), .CI(CO21));
438  FULL_ADDER fa_inst_23(.S(Y[23]), .CO(CO23), .A(A[23]), .B(B23), .CI(CO22));
439  FULL_ADDER fa_inst_24(.S(Y[24]), .CO(CO24), .A(A[24]), .B(B24), .CI(CO23));
440  FULL_ADDER fa_inst_25(.S(Y[25]), .CO(CO25), .A(A[25]), .B(B25), .CI(CO24));
441  FULL_ADDER fa_inst_26(.S(Y[26]), .CO(CO26), .A(A[26]), .B(B26), .CI(CO25));
442  FULL_ADDER fa_inst_27(.S(Y[27]), .CO(CO27), .A(A[27]), .B(B27), .CI(CO26));
443  FULL_ADDER fa_inst_28(.S(Y[28]), .CO(CO28), .A(A[28]), .B(B28), .CI(CO27));
444  FULL_ADDER fa_inst_29(.S(Y[29]), .CO(CO29), .A(A[29]), .B(B29), .CI(CO28));
445  FULL_ADDER fa_inst_30(.S(Y[30]), .CO(CO30), .A(A[30]), .B(B30), .CI(CO29));
446  FULL_ADDER fa_inst_31(.S(Y[31]), .CO(CO31), .A(A[31]), .B(B31), .CI(CO30));
447
448  assign CO = CO31;
449
450  endmodule
```

The actual addition/subtraction process.

Testing:

```
1   `timescale 1ns/1ps
2   `include "prj_definition.v"
3
4   module rc_add_sub_32_tb;
5   reg SnA;
6   wire CO, S;
7   wire [31:0] Y;
8
9   reg [31:0] A;
10  reg [31:0] B;
11
12  // module RC_ADD_SUB_32(Y, CO, A, B, SnA);
13  RC_ADD_SUB_32 fa_inst_1(.Y(Y), .CO(CO), .A(A), .B(B), .SnA(SnA));
14
15  initial
16  begin
17
18  A=0; B=0; SnA=0;
19  #5 A=1; B=0; SnA=0;
20  #5 A=0; B=0; SnA=0;
21  #5 A=0; B=1; SnA=0;
22  #5 A=0; B=0; SnA=1;
23
24
25  #5 A=0; B=0; SnA=0;
26  #5 A=10; B=0; SnA=0;
27  #5 A=0; B=10; SnA=1;
28  #5 A=100; B=100; SnA=1;
29  #5 A=100; B=100; SnA=0;
30  #5 A=32'b1; B=0; SnA=0;
31  #5 A=0; B=32'b1; SnA=0;
32  #5 ;
33  end
34
35  endmodule
```

When SnA =1 then subtraction occurs.



Waveform line order: SnA, CO, Y, A, B
(Not showing the first cases as they don't illustrate as well.)

Test cases:
(A+B=Y, minus sign => SnA == 1)
0+0=0
1+0=1 (Test A)
0+0=0 (Testing if anything will carry between operations)
0+1=0 (Testing B)
0-0=0 (Checking subtraction)
0+0=0 (Checking if any carry over from subtraction)

10+0=10 (Checking more than one bit on A side)
0-10=-10 (two's complement in the result/image above)
100-100=0
100+100=200
(32 1's in binary)+0=(32 1's in binary)
0+(32 1's in binary)=(32 1's in binary)

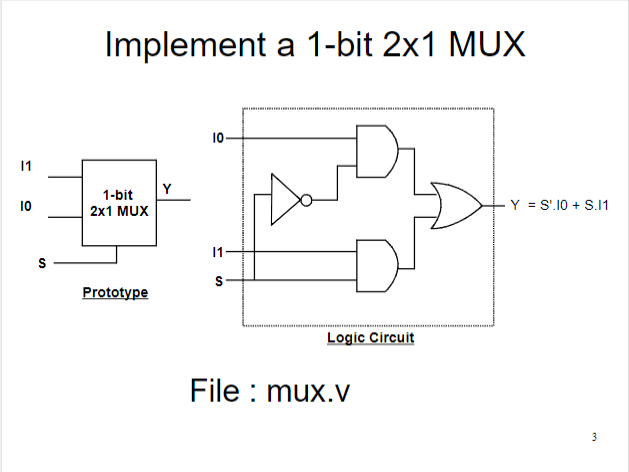
B. Multiplier

The multiplier will be using multiplexers, adders, 32-bit ands, and 32-bit two's complement. As far as the labs instructed, we hadn't implemented the 32-bit and so I made my own for this multiplier, it works the same as the one instructed to make later. It is as follows:

```
414  module mult_32x_and(R, A, B);
415  // output list
416  output [31:0] R;
417  // input list
418  input [31:0] A;
419  input [31:0] B;
420
421  //and and0(R[0], A[0], B[0]);
422
423  and and0(R[0], A[0], B[0]);
424  and and1(R[1], A[1], B[1]);
425  and and2(R[2], A[2], B[2]);
426  and and3(R[3], A[3], B[3]);
427  and and4(R[4], A[4], B[4]);
428  and and5(R[5], A[5], B[5]);
429  and and6(R[6], A[6], B[6]);
430  and and7(R[7], A[7], B[7]);
431  and and8(R[8], A[8], B[8]);
432  and and9(R[9], A[9], B[9]);
433  and and10(R[10], A[10], B[10]);
434  and and11(R[11], A[11], B[11]);
435  and and12(R[12], A[12], B[12]);
436  and and13(R[13], A[13], B[13]);
437  and and14(R[14], A[14], B[14]);
438  and and15(R[15], A[15], B[15]);
439  and and16(R[16], A[16], B[16]);
440  and and17(R[17], A[17], B[17]);
441  and and18(R[18], A[18], B[18]);
442  and and19(R[19], A[19], B[19]);
443  and and20(R[20], A[20], B[20]);
444  and and21(R[21], A[21], B[21]);
445  and and22(R[22], A[22], B[22]);
446  and and23(R[23], A[23], B[23]);
447  and and24(R[24], A[24], B[24]);
448  and and25(R[25], A[25], B[25]);
449  and and26(R[26], A[26], B[26]);
```

```
449  and and26(R[26], A[26], B[26]);
450  and and27(R[27], A[27], B[27]);
451  and and28(R[28], A[28], B[28]);
452  and and29(R[29], A[29], B[29]);
453  and and30(R[30], A[30], B[30]);
454  and and31(R[31], A[31], B[31]);
455
456  endmodule
```

First we must begin constructing the multiplexers. The basic operation of a multiplexer is to allow you to select between different inputs by picking the selection bit that refers to the number of that input. (i.e. If you want input 16 set the selection value to 16 and the multiplexer will output the input from 16.)



(Kaushik Patra, Lab 12, Slide 2)

```
174  // 1-bit mux
175  module MUX1_2x1(Y,I0, I1, S);
176  //output list
177  output Y;
178  //input list
179  input I0, I1, S;
180
181  wire tempI0, tempI1;
182  wire invS;
183
184  assign invS = ~S;
185
186  and inst1(tempI0, invS, I0);
187  and inst2(tempI1, S, I1);
188  or inst3(Y, tempI1, tempI0);
189
190
191  endmodule
```

Testing:

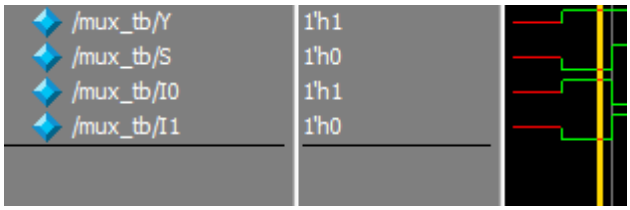
For any of the multiplexers, you simply have to be able to select any of the inputs. For the 2x1 simply check if you can select from I0 and I1. (My mux_tb.v tests many things so I will show only what is relevant.)

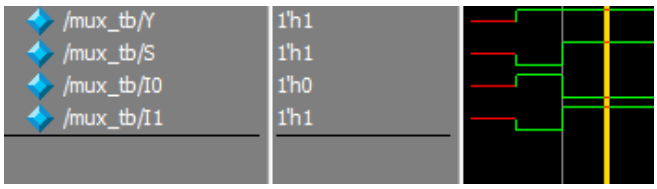
```
wire Y;
wire [31:0] Y_32;
wire [31:0] Y_16x1;

reg [31:0] I0_32;
reg [31:0] I1_32;
reg S;
reg I0, I1;
```

```
MUX1_2x1 mux1(.Y(Y), .I0(I0), .I1(I1), .S(S));
```

(If any registers aren't shown/accidentally get ommited, assume they are made. Rule of thumb: if you need to input the value then make a register, if it's an output then make it a wire all of the appropriate bit length.)

Waveform:

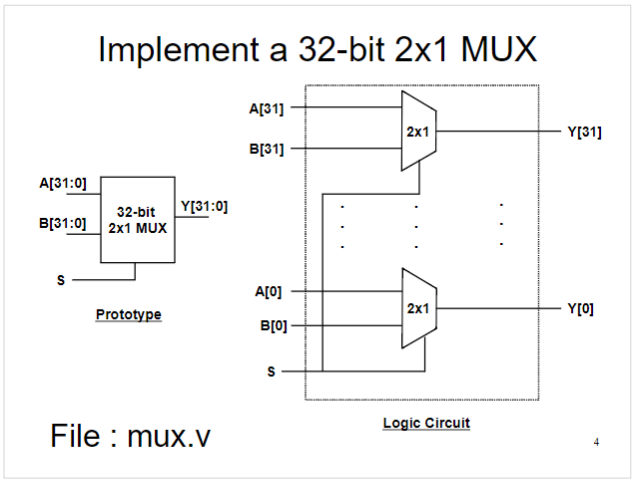





```
#5 ;

I0=1; I1=0; I0_32 = 50; I1_32 = 25; S=0; S_16x1=0;
#5 I0=0; I1=1; I0_32 = 50; I1_32 = 25; S=1; S_16x1=1;
```

Test cases:
I0 set to 1, I1 set to 0, S=0, Y=1
I0 set to 0, I1 set to 1, S=1, Y=1

Now we need to expand out multiplexer to 32 bits so put 32 1bit 2x1's in series:



(Kaushik Patra, Lab 12, Slide 2)

```
// 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;
```

```
MUX1_2x1 mux_0(.Y(Y[0]), .I0(I0[0]), .I1(I1[0]), .S(S));
MUX1_2x1 mux_1(.Y(Y[1]), .I0(I0[1]), .I1(I1[1]), .S(S));
MUX1_2x1 mux_2(.Y(Y[2]), .I0(I0[2]), .I1(I1[2]), .S(S));
MUX1_2x1 mux_3(.Y(Y[3]), .I0(I0[3]), .I1(I1[3]), .S(S));
MUX1_2x1 mux_4(.Y(Y[4]), .I0(I0[4]), .I1(I1[4]), .S(S));
MUX1_2x1 mux_5(.Y(Y[5]), .I0(I0[5]), .I1(I1[5]), .S(S));
MUX1_2x1 mux_6(.Y(Y[6]), .I0(I0[6]), .I1(I1[6]), .S(S));
MUX1_2x1 mux_7(.Y(Y[7]), .I0(I0[7]), .I1(I1[7]), .S(S));
MUX1_2x1 mux_8(.Y(Y[8]), .I0(I0[8]), .I1(I1[8]), .S(S));
MUX1_2x1 mux_9(.Y(Y[9]), .I0(I0[9]), .I1(I1[9]), .S(S));
MUX1_2x1 mux_10(.Y(Y[10]), .I0(I0[10]), .I1(I1[10]), .S(S));
MUX1_2x1 mux_11(.Y(Y[11]), .I0(I0[11]), .I1(I1[11]), .S(S));
MUX1_2x1 mux_12(.Y(Y[12]), .I0(I0[12]), .I1(I1[12]), .S(S));
MUX1_2x1 mux_13(.Y(Y[13]), .I0(I0[13]), .I1(I1[13]), .S(S));
MUX1_2x1 mux_14(.Y(Y[14]), .I0(I0[14]), .I1(I1[14]), .S(S));
MUX1_2x1 mux_15(.Y(Y[15]), .I0(I0[15]), .I1(I1[15]), .S(S));
MUX1_2x1 mux_16(.Y(Y[16]), .I0(I0[16]), .I1(I1[16]), .S(S));
MUX1_2x1 mux_17(.Y(Y[17]), .I0(I0[17]), .I1(I1[17]), .S(S));
MUX1_2x1 mux_18(.Y(Y[18]), .I0(I0[18]), .I1(I1[18]), .S(S));
MUX1_2x1 mux_19(.Y(Y[19]), .I0(I0[19]), .I1(I1[19]), .S(S));
MUX1_2x1 mux_20(.Y(Y[20]), .I0(I0[20]), .I1(I1[20]), .S(S));
MUX1_2x1 mux_21(.Y(Y[21]), .I0(I0[21]), .I1(I1[21]), .S(S));
MUX1_2x1 mux_22(.Y(Y[22]), .I0(I0[22]), .I1(I1[22]), .S(S));
MUX1_2x1 mux_23(.Y(Y[23]), .I0(I0[23]), .I1(I1[23]), .S(S));
MUX1_2x1 mux_24(.Y(Y[24]), .I0(I0[24]), .I1(I1[24]), .S(S));
MUX1_2x1 mux_25(.Y(Y[25]), .I0(I0[25]), .I1(I1[25]), .S(S));
MUX1_2x1 mux_26(.Y(Y[26]), .I0(I0[26]), .I1(I1[26]), .S(S));
MUX1_2x1 mux_27(.Y(Y[27]), .I0(I0[27]), .I1(I1[27]), .S(S));
MUX1_2x1 mux_28(.Y(Y[28]), .I0(I0[28]), .I1(I1[28]), .S(S));
MUX1_2x1 mux_29(.Y(Y[29]), .I0(I0[29]), .I1(I1[29]), .S(S));
MUX1_2x1 mux_30(.Y(Y[30]), .I0(I0[30]), .I1(I1[30]), .S(S));
MUX1_2x1 mux_31(.Y(Y[31]), .I0(I0[31]), .I1(I1[31]), .S(S));

endmodule
```

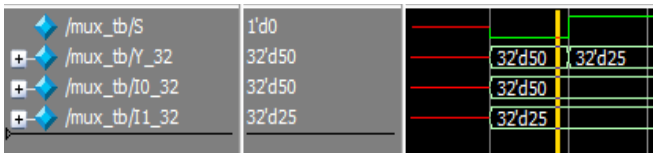
(*Side note*: You've probably noticed the implementation method I've done has every single connection “manually” made. Do not do it manually. Simply make a simple script or for loop to output the code to text and copy paste. As above you can see the index of the loop is the particular bit that needs to be accessed etc. So, using above as an example, you can do “MUX1_2x1 mux_(i)(.Y(Y[(i)]), .I0(I0[(i)]), .I1(I1[(i)]), .S(S));” where “(i)” is replaced with however you set the string to be outputted as the current index.)

Testing:

```
wire Y;
wire [31:0] Y_32;
wire [31:0] Y_16x1;

reg [31:0] I0_32;
reg [31:0] I1_32;
reg S;
reg I0, I1;
```

```
MUX32_2x1 mux2(.Y(Y_32), .I0(I0_32), .I1(I1_32), .S(S));
```

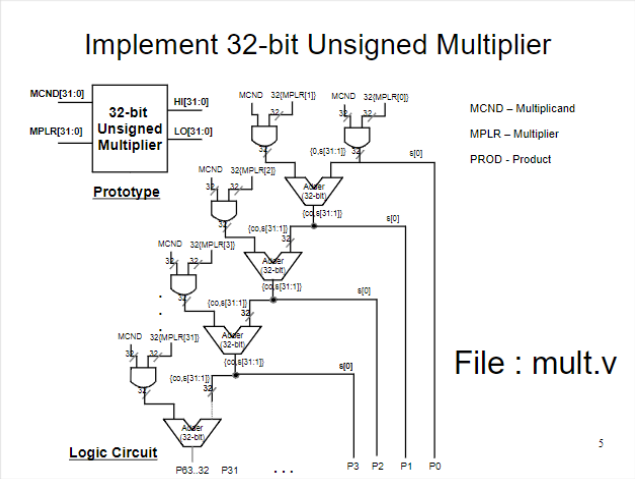
```
#5 ;

I0=1; I1=0; I0_32 = 50; I1_32 = 25; S=0; S_16x1=0;
#5 I0=0; I1=1; I0_32 = 50; I1_32 = 25; S=1; S_16x1=1;
```



Test cases:
I0 = 50, I1=25, S=0, Y=50
I0 = 50, I1=25, S=1, Y=25
(It isn't really necessary to test every bit in this case as we know the 1 bit 2x1 works, but you can do that if you're skeptical.)

Now we can properly begin making the multiplier.

The basic function of a multiplier is to replicate actual multiplication.
We will start with unsigned multiplication.



We have the 32 bit adder and and gates made already. Simply remember that the first carry in is 0 and follow the schematic in all subsequent cases.

```
67  module MULT32_U(HI, LO, A, B);
68  // output list
69  output [31:0] HI;
70  output [31:0] LO;
71  // input list
72  input [31:0] A;
73  input [31:0] B;
```

```
80  wire [31:0] B_ext0 = {32{B[0]}};
81  wire [31:0] B_ext1 = {32{B[1]}};
82  wire [31:0] B_ext2 = {32{B[2]}};
83  wire [31:0] B_ext3 = {32{B[3]}};
84  wire [31:0] B_ext4 = {32{B[4]}};
85  wire [31:0] B_ext5 = {32{B[5]}};
86  wire [31:0] B_ext6 = {32{B[6]}};
87  wire [31:0] B_ext7 = {32{B[7]}};
88  wire [31:0] B_ext8 = {32{B[8]}};
89  wire [31:0] B_ext9 = {32{B[9]}};
90  wire [31:0] B_ext10 = {32{B[10]}};
91  wire [31:0] B_ext11 = {32{B[11]}};
92  wire [31:0] B_ext12 = {32{B[12]}};
93  wire [31:0] B_ext13 = {32{B[13]}};
94  wire [31:0] B_ext14 = {32{B[14]}};
95  wire [31:0] B_ext15 = {32{B[15]}};
96  wire [31:0] B_ext16 = {32{B[16]}};
97  wire [31:0] B_ext17 = {32{B[17]}};
98  wire [31:0] B_ext18 = {32{B[18]}};
99  wire [31:0] B_ext19 = {32{B[19]}};
100 wire [31:0] B_ext20 = {32{B[20]}};
101 wire [31:0] B_ext21 = {32{B[21]}};
102 wire [31:0] B_ext22 = {32{B[22]}};
103 wire [31:0] B_ext23 = {32{B[23]}};
104 wire [31:0] B_ext24 = {32{B[24]}};
105 wire [31:0] B_ext25 = {32{B[25]}};
106 wire [31:0] B_ext26 = {32{B[26]}};
107 wire [31:0] B_ext27 = {32{B[27]}};
108 wire [31:0] B_ext28 = {32{B[28]}};
109 wire [31:0] B_ext29 = {32{B[29]}};
110 wire [31:0] B_ext30 = {32{B[30]}};
111 wire [31:0] B_ext31 = {32{B[31]}};
```

```
113 wire [31:0] andResult0;
114 wire [31:0] andResult1;
115 wire [31:0] andResult2;
116 wire [31:0] andResult3;
117 wire [31:0] andResult4;
118 wire [31:0] andResult5;
119 wire [31:0] andResult6;
120 wire [31:0] andResult7;
121 wire [31:0] andResult8;
122 wire [31:0] andResult9;
123 wire [31:0] andResult10;
124 wire [31:0] andResult11;
125 wire [31:0] andResult12;
126 wire [31:0] andResult13;
127 wire [31:0] andResult14;
128 wire [31:0] andResult15;
129 wire [31:0] andResult16;
130 wire [31:0] andResult17;
131 wire [31:0] andResult18;
132 wire [31:0] andResult19;
133 wire [31:0] andResult20;
134 wire [31:0] andResult21;
135 wire [31:0] andResult22;
136 wire [31:0] andResult23;
137 wire [31:0] andResult24;
138 wire [31:0] andResult25;
139 wire [31:0] andResult26;
140 wire [31:0] andResult27;
141 wire [31:0] andResult28;
142 wire [31:0] andResult29;
143 wire [31:0] andResult30;
144 wire [31:0] andResult31;
```

These are for the 32 and operations.

```
146 mult_32x_and and0(andResult0, A, B_ext0);
147 mult_32x_and and1(andResult1, A, B_ext1);
148 mult_32x_and and2(andResult2, A, B_ext2);
149 mult_32x_and and3(andResult3, A, B_ext3);
150 mult_32x_and and4(andResult4, A, B_ext4);
151 mult_32x_and and5(andResult5, A, B_ext5);
152 mult_32x_and and6(andResult6, A, B_ext6);
153 mult_32x_and and7(andResult7, A, B_ext7);
154 mult_32x_and and8(andResult8, A, B_ext8);
155 mult_32x_and and9(andResult9, A, B_ext9);
156 mult_32x_and and10(andResult10, A, B_ext10);
157 mult_32x_and and11(andResult11, A, B_ext11);
158 mult_32x_and and12(andResult12, A, B_ext12);
159 mult_32x_and and13(andResult13, A, B_ext13);
160 mult_32x_and and14(andResult14, A, B_ext14);
161 mult_32x_and and15(andResult15, A, B_ext15);
162 mult_32x_and and16(andResult16, A, B_ext16);
163 mult_32x_and and17(andResult17, A, B_ext17);
164 mult_32x_and and18(andResult18, A, B_ext18);
165 mult_32x_and and19(andResult19, A, B_ext19);
166 mult_32x_and and20(andResult20, A, B_ext20);
167 mult_32x_and and21(andResult21, A, B_ext21);
168 mult_32x_and and22(andResult22, A, B_ext22);
169 mult_32x_and and23(andResult23, A, B_ext23);
170 mult_32x_and and24(andResult24, A, B_ext24);
171 mult_32x_and and25(andResult25, A, B_ext25);
172 mult_32x_and and26(andResult26, A, B_ext26);
173 mult_32x_and and27(andResult27, A, B_ext27);
174 mult_32x_and and28(andResult28, A, B_ext28);
175 mult_32x_and and29(andResult29, A, B_ext29);
176 mult_32x_and and30(andResult30, A, B_ext30);
177 mult_32x_and and31(andResult31, A, B_ext31);
```

```
wire [31:0] adderRight23;
wire [31:0] adderRight24;
wire [31:0] adderRight25;
wire [31:0] adderRight26;
wire [31:0] adderRight27;
wire [31:0] adderRight28;
wire [31:0] adderRight29;
wire [31:0] adderRight30;
wire [31:0] adderRight31;

wire [31:0] adderResult0;
wire [31:0] adderResult1;
wire [31:0] adderResult2;
wire [31:0] adderResult3;
wire [31:0] adderResult4;
wire [31:0] adderResult5;
```

((32 of both these [adderResult stops at index 30], all with the end of the name as their index... Saving space.))

```
wire adderCO0;
wire adderCO1;
wire adderCO2;
wire adderCO3;
wire adderCO4;
wire adderCO5;
wire adderCO6;
wire adderCO7;
wire adderCO8;
wire adderCO9;
wire adderCO10;
wire adderCO11;
wire adderCO12;
wire adderCO13;
wire adderCO14;
wire adderCO15;
wire adderCO16;
wire adderCO17;
wire adderCO18;
wire adderCO19;
wire adderCO20;
wire adderCO21;
wire adderCO22;
wire adderCO23;
wire adderCO24;
wire adderCO25;
wire adderCO26;
wire adderCO27;
wire adderCO28;
wire adderCO29;
wire adderCO30;
wire adderCO31;
```

Next segment will be plain text as it's easier to read that :

assign LO[0] = andResult0[0];

//module RC_ADD_SUB_32(Y, CO, A, B, SnA); template
//RC_ADD_SUB_32 fa_inst_1(.Y(Y), .CO(CO), .A(A), .B(B), .SnA(SnA));

assign adderRight0 = {1'b0,andResult0[31:1]}; // first case/adder only

RC_ADD_SUB_32 adder_0(.Y(adderResult0), .CO(adderCO0), .A(adderRight0), .B(andResult1), .SnA(1'b0));
assign LO[1] = adderResult0[0];

assign adderRight1 = {adderCO0, adderResult0[31:1]};
RC_ADD_SUB_32 adder_1(.Y(adderResult1), .CO(adderCO1), .A(adderRight1), .B(andResult2), .SnA(1'b0));
assign LO[2] = adderResult1[0];

assign adderRight2 = {adderCO1, adderResult1[31:1]};
RC_ADD_SUB_32 adder_2(.Y(adderResult2), .CO(adderCO2), .A(adderRight2), .B(andResult3), .SnA(1'b0));
assign LO[3] = adderResult2[0];

assign adderRight3 = {adderCO2, adderResult2[31:1]};
RC_ADD_SUB_32 adder_3(.Y(adderResult3), .CO(adderCO3), .A(adderRight3), .B(andResult4), .SnA(1'b0));
assign LO[4] = adderResult3[0];

assign adderRight4 = {adderCO3, adderResult3[31:1]};
RC_ADD_SUB_32 adder_4(.Y(adderResult4), .CO(adderCO4), .A(adderRight4), .B(andResult5), .SnA(1'b0));
assign LO[5] = adderResult4[0];

assign adderRight5 = {adderCO4, adderResult4[31:1]};
RC_ADD_SUB_32 adder_5(.Y(adderResult5), .CO(adderCO5), .A(adderRight5), .B(andResult6), .SnA(1'b0));
assign LO[6] = adderResult5[0];

assign adderRight6 = {adderCO5, adderResult5[31:1]};
RC_ADD_SUB_32 adder_6(.Y(adderResult6), .CO(adderCO6), .A(adderRight6), .B(andResult7), .SnA(1'b0));
assign LO[7] = adderResult6[0];

assign adderRight7 = {adderCO6, adderResult6[31:1]};
RC_ADD_SUB_32 adder_7(.Y(adderResult7), .CO(adderCO7), .A(adderRight7), .B(andResult8), .SnA(1'b0));
assign LO[8] = adderResult7[0];

assign adderRight8 = {adderCO7, adderResult7[31:1]};
RC_ADD_SUB_32 adder_8(.Y(adderResult8), .CO(adderCO8), .A(adderRight8), .B(andResult9), .SnA(1'b0));
assign LO[9] = adderResult8[0];

assign adderRight9 = {adderCO8, adderResult8[31:1]};
RC_ADD_SUB_32 adder_9(.Y(adderResult9), .CO(adderCO9), .A(adderRight9), .B(andResult10), .SnA(1'b0));
assign LO[10] = adderResult9[0];

assign adderRight10 = {adderCO9, adderResult9[31:1]};
RC_ADD_SUB_32 adder_10(.Y(adderResult10), .CO(adderCO10), .A(adderRight10), .B(andResult11), .SnA(1'b0));
assign LO[11] = adderResult10[0];

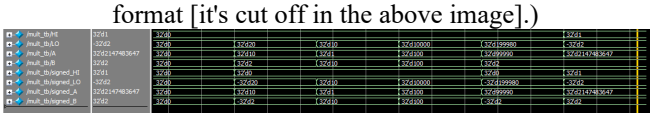
```
assign adderRight11 = {adderCO10, adderResult10[31:1]};
RC_ADD_SUB_32 adder_11(.Y(adderResult11),
.CO(adderCO11), .A(adderRight11), .B(andResult12),
.SnA(1'b0));
assign LO[12] = adderResult11[0];

assign adderRight12 = {adderCO11, adderResult11[31:1]};
RC_ADD_SUB_32 adder_12(.Y(adderResult12),
.CO(adderCO12), .A(adderRight12), .B(andResult13),
.SnA(1'b0));
assign LO[13] = adderResult12[0];

assign adderRight13 = {adderCO12, adderResult12[31:1]};
RC_ADD_SUB_32 adder_13(.Y(adderResult13),
.CO(adderCO13), .A(adderRight13), .B(andResult14),
.SnA(1'b0));
assign LO[14] = adderResult13[0];

assign adderRight14 = {adderCO13, adderResult13[31:1]};
RC_ADD_SUB_32 adder_14(.Y(adderResult14),
.CO(adderCO14), .A(adderRight14), .B(andResult15),
.SnA(1'b0));
assign LO[15] = adderResult14[0];

assign adderRight15 = {adderCO14, adderResult14[31:1]};
RC_ADD_SUB_32 adder_15(.Y(adderResult15),
.CO(adderCO15), .A(adderRight15), .B(andResult16),
.SnA(1'b0));
assign LO[16] = adderResult15[0];

assign adderRight16 = {adderCO15, adderResult15[31:1]};
RC_ADD_SUB_32 adder_16(.Y(adderResult16),
.CO(adderCO16), .A(adderRight16), .B(andResult17),
.SnA(1'b0));
assign LO[17] = adderResult16[0];

assign adderRight17 = {adderCO16, adderResult16[31:1]};
RC_ADD_SUB_32 adder_17(.Y(adderResult17),
.CO(adderCO17), .A(adderRight17), .B(andResult18),
.SnA(1'b0));
assign LO[18] = adderResult17[0];

assign adderRight18 = {adderCO17, adderResult17[31:1]};
RC_ADD_SUB_32 adder_18(.Y(adderResult18),
.CO(adderCO18), .A(adderRight18), .B(andResult19),
.SnA(1'b0));
assign LO[19] = adderResult18[0];

assign adderRight19 = {adderCO18, adderResult18[31:1]};
RC_ADD_SUB_32 adder_19(.Y(adderResult19),
.CO(adderCO19), .A(adderRight19), .B(andResult20),
.SnA(1'b0));
assign LO[20] = adderResult19[0];

assign adderRight20 = {adderCO19, adderResult19[31:1]};
RC_ADD_SUB_32 adder_20(.Y(adderResult20),
.CO(adderCO20), .A(adderRight20), .B(andResult21),
.SnA(1'b0));
assign LO[21] = adderResult20[0];

assign adderRight21 = {adderCO20, adderResult20[31:1]};
RC_ADD_SUB_32 adder_21(.Y(adderResult21),
.CO(adderCO21), .A(adderRight21), .B(andResult22),
.SnA(1'b0));
assign LO[22] = adderResult21[0];

assign adderRight22 = {adderCO21, adderResult21[31:1]};
RC_ADD_SUB_32 adder_22(.Y(adderResult22),
.CO(adderCO22), .A(adderRight22), .B(andResult23),
.SnA(1'b0));
assign LO[23] = adderResult22[0];

assign adderRight23 = {adderCO22, adderResult22[31:1]};
RC_ADD_SUB_32 adder_23(.Y(adderResult23),
.CO(adderCO23), .A(adderRight23), .B(andResult24),
.SnA(1'b0));
assign LO[24] = adderResult23[0];

assign adderRight24 = {adderCO23, adderResult23[31:1]};
RC_ADD_SUB_32 adder_24(.Y(adderResult24),
.CO(adderCO24), .A(adderRight24), .B(andResult25),
.SnA(1'b0));
assign LO[25] = adderResult24[0];

assign adderRight25 = {adderCO24, adderResult24[31:1]};
RC_ADD_SUB_32 adder_25(.Y(adderResult25),
.CO(adderCO25), .A(adderRight25), .B(andResult26),
.SnA(1'b0));
assign LO[26] = adderResult25[0];

assign adderRight26 = {adderCO25, adderResult25[31:1]};
RC_ADD_SUB_32 adder_26(.Y(adderResult26),
.CO(adderCO26), .A(adderRight26), .B(andResult27),
.SnA(1'b0));
assign LO[27] = adderResult26[0];

assign adderRight27 = {adderCO26, adderResult26[31:1]};
RC_ADD_SUB_32 adder_27(.Y(adderResult27),
.CO(adderCO27), .A(adderRight27), .B(andResult28),
.SnA(1'b0));
assign LO[28] = adderResult27[0];

assign adderRight28 = {adderCO27, adderResult27[31:1]};
RC_ADD_SUB_32 adder_28(.Y(adderResult28),
.CO(adderCO28), .A(adderRight28), .B(andResult29),
.SnA(1'b0));
assign LO[29] = adderResult28[0];

assign adderRight29 = {adderCO28, adderResult28[31:1]};
RC_ADD_SUB_32 adder_29(.Y(adderResult29),
.CO(adderCO29), .A(adderRight29), .B(andResult30),
.SnA(1'b0));
assign LO[30] = adderResult29[0];

assign adderRight30 = {adderCO29, adderResult29[31:1]};
RC_ADD_SUB_32 adder_30(.Y(adderResult30),
.CO(adderCO30), .A(adderRight30), .B(andResult31),
.SnA(1'b0));
assign LO[31] = adderResult30[0];
assign HI = adderResult30;

endmodule
```

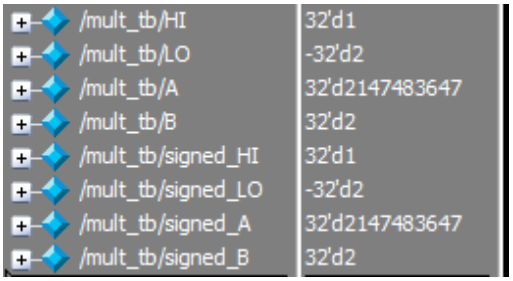
Testing:

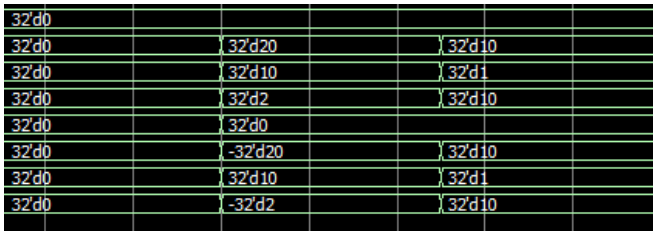
```
module mult_tb;
//reg SnA;
//wire CO, S;
wire [31:0] HI;
wire [31:0] LO;

reg [31:0] A;
reg [31:0] B;

wire [31:0] signed_HI;
wire [31:0] signed_LO;

reg [31:0] signed_A;
reg [31:0] signed_B;

// module MULT32_U(HI, LO, A, B);
// module MULT32(HI, LO, A, B);
MULT32_U mult(.HI(HI), .LO(LO), .A(A), .B(B));
MULT32 signedMult(.HI(signed_HI), .LO(signed_LO), .A(signed_A), .B(signed_B));
```

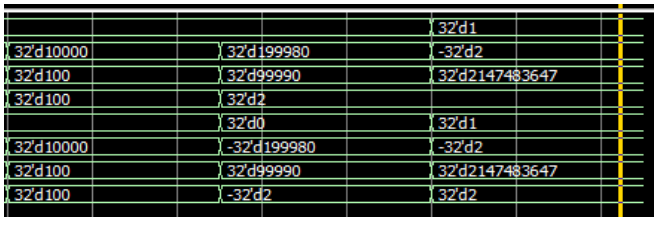
```
24      initial
25      begin
26      #5 A=0; B=0; signed_A=0; signed_B=0;
27      #5 A=10; B=2; signed_A=10; signed_B=-2;
28      #5 A=1; B=10; signed_A=1; signed_B=10;
29      #5 A=100; B=100; signed_A=100; signed_B=100;
30      #5 A=99990; B=2; signed_A=99990; signed_B=-2;
31      #5 A=32'b11111111111111111111111111111111; B=2; si
32      #5 ;
33      end
34
35      endmodule
```

#5 A=32'b11111111111111111111111111111111; B=2; signed_A=32'b11111111111111111111111111111111; signed_B=2;
(Line 31 as plain text since it's long and hard to see in image format [it's cut off in the above image].)





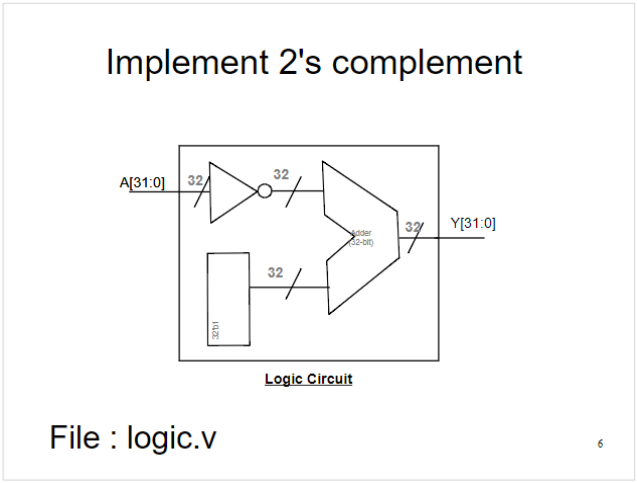




Test cases (Including the signed multiplier, will show implementation after):
(AxB=LO (HI is 0 unless stated otherwise))
Unsigned:0x0=0 Signed: 0x0=0
Unsigned:10x2=20 Signed: 10x-2=-20
Unsigned:1x10=10 Signed: 1x10=10
Unsigned:100x100=10000 Signed: 100x100=10000
Unsigned:99990x2=199980 Signed: 99990x-2=-199980
Unsigned:(32x1 bit binary)x2=2147483647 (HI = 1)
Signed: (32x1 bit binary)x-2=-2147483647 (HI = 1)

(Also, we can see that two's complement works through this test so I will not be testing that module. If you do wan't to test it just put any number into it and verify that the output is the correct number in binary. (i.e. put in 2 and check that you get [31 1's and a 0 in binary].))

Implementation of two's complement:
The basic operation is to flip the bits of your operand and then add 1 to it.

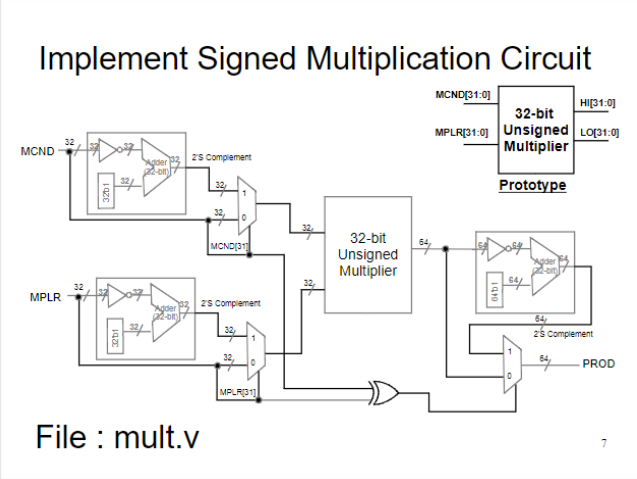


(Kaushik Patra, Lab 12, Slide 3)

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

// TBD
wire [31:0] notA;
wire [31:0] B = 32'b1;
wire SnA = 1'b0;
wire CO;
assign notA = ~A;

RC_ADD_SUB_32 fa_inst_1(.Y(Y), .CO(CO), .A(notA), .B(B), .SnA(SnA));

endmodule
```

We also need a 64 bit variant so:
These will be used for the signed multiplier.

```
// 64-bit two's complement
module TWOSCOMP64(Y,A);
//output list
output [63:0] Y;
//input list
input [63:0] A;

// TBD
wire [63:0] notA;
wire [63:0] B = 64'b1;
wire SnA = 1'b0;
wire CO;
assign notA = ~A;

RC_ADD_SUB_32 fa_inst_1(.Y(Y[31:0]), .CO(CO), .A(notA[31:0]), .B(B[31:0]), .SnA(SnA));
RC_ADD_SUB_32 fa_inst_2(.Y(Y[63:32]), .CO(CO), .A(notA[63:32]), .B(B[63:32]), .SnA(SnA));
endmodule
```



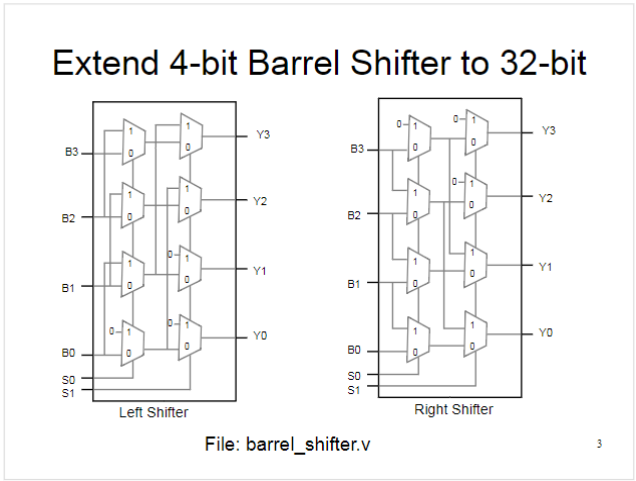
(Kaushik Patra, Lab 12, Slide 4)

```
22  module MULT32(HI, LO, A, B);
23  // output list
24  output [31:0] HI;
25  output [31:0] LO;
26  // input list
27  input [31:0] A;
28  input [31:0] B;
29
30  // TBD
31  wire [31:0] A2;
32  wire [31:0] B2;
33
34  wire [31:0] selectedA;
35  wire [31:0] selectedB;
36
37  wire [31:0] tempHI;
38  wire [31:0] tempLO;
39
40  wire [31:0] HI2;
41  wire [31:0] LO2;
42
43  wire [63:0] intermediateResult;
44  wire resultSelect;
45
46  //module TWOSCOMP32(Y,A);
47  TWOSCOMP32 compA(.Y(A2), .A(A));
48  TWOSCOMP32 compB(.Y(B2), .A(B));
```

```
49
50  //module MUX32_2x1(Y, I0, I1, S);
51  MUX32_2x1 mux_mcnd(.Y(selectedA), .I0(A), .I1(A2), .S(A[31]));
52  MUX32_2x1 mux_mplr(.Y(selectedB), .I0(B), .I1(B2), .S(B[31]));
53
54  //module MULT32_U(HI, LO, A, B);
55  MULT32_U mult(.HI(tempHI), .LO(tempLO), .A(selectedA), .B(selectedB));
56
57  TWOSCOMP32 compHI(.Y(HI2), .A(tempHI));
58  TWOSCOMP32 compLO(.Y(LO2), .A(tempLO));
59
60  xor xor0(resultSelect, A[31], B[31]);
61
62  MUX32_2x1 mux_first_64(.Y(LO), .I0(tempLO), .I1(LO2), .S(resultSelect));
63  MUX32_2x1 mux_second_64(.Y(HI), .I0(tempHI), .I1(HI2), .S(resultSelect));
64
65  endmodule
```

(Testing has already been shown. Refer to the unsigned multiplier's testing section.)

C. Barrel Shifter



(Kaushik Patra, Lab 13, Slide 2)

To extend the left and right shifter to 32 bits you need 5 total shifts (5 vertical rows of muxs, 32 muxs per row, with the row's shifts being 1, 2, 4, 8, 16.) The selection bit for the first row is 0 up to 4 for the fifth row.

```
435  wire firstShift0;
436  wire firstShift1;
437  wire firstShift2;
438  wire firstShift3;
439  wire firstShift4;
440  wire firstShift5;
441  wire firstShift6;
442  wire firstShift7;
443  wire firstShift8;
444  wire firstShift9;
445  wire firstShift10;
446  wire firstShift11;
447  wire firstShift12;
448  wire firstShift13;
449  wire firstShift14;
450  wire firstShift15;
451  wire firstShift16;
452  wire firstShift17;
453  wire firstShift18;
454  wire firstShift19;
455  wire firstShift20;
456  wire firstShift21;
457  wire firstShift22;
458  wire firstShift23;
459  wire firstShift24;
460  wire firstShift25;
461  wire firstShift26;
462  wire firstShift27;
463  wire firstShift28;
464  wire firstShift29;
465  wire firstShift30;
466  wire firstShift31;
```

Both left and right shifters have similar sets of wires like above. There is a set of firstShift0->31, secondShift0->31, third...fourthShift0->31. For the fifth shift simply save the shifted bit to the output wire, Y.

Right shifter:

```
wire fourthShift27;
wire fourthShift28;
wire fourthShift29;
wire fourthShift30;
wire fourthShift31;

wire zeroBit;
assign zeroBit = 1'b0;
```

```
MUX1_2x1 mux_first_0(.Y(firstShift0), .I0(D[0]), .I1(D[1]), .S(S[0]));
MUX1_2x1 mux_first_1(.Y(firstShift1), .I0(D[1]), .I1(D[2]), .S(S[0]));
MUX1_2x1 mux_first_2(.Y(firstShift2), .I0(D[2]), .I1(D[3]), .S(S[0]));
MUX1_2x1 mux_first_3(.Y(firstShift3), .I0(D[3]), .I1(D[4]), .S(S[0]));
MUX1_2x1 mux_first_4(.Y(firstShift4), .I0(D[4]), .I1(D[5]), .S(S[0]));
MUX1_2x1 mux_first_5(.Y(firstShift5), .I0(D[5]), .I1(D[6]), .S(S[0]));
MUX1_2x1 mux_first_6(.Y(firstShift6), .I0(D[6]), .I1(D[7]), .S(S[0]));
MUX1_2x1 mux_first_7(.Y(firstShift7), .I0(D[7]), .I1(D[8]), .S(S[0]));
MUX1_2x1 mux_first_8(.Y(firstShift8), .I0(D[8]), .I1(D[9]), .S(S[0]));
MUX1_2x1 mux_first_9(.Y(firstShift9), .I0(D[9]), .I1(D[10]), .S(S[0]));
MUX1_2x1 mux_first_10(.Y(firstShift10), .I0(D[10]), .I1(D[11]), .S(S[0]));
MUX1_2x1 mux_first_11(.Y(firstShift11), .I0(D[11]), .I1(D[12]), .S(S[0]));
MUX1_2x1 mux_first_12(.Y(firstShift12), .I0(D[12]), .I1(D[13]), .S(S[0]));
MUX1_2x1 mux_first_13(.Y(firstShift13), .I0(D[13]), .I1(D[14]), .S(S[0]));
MUX1_2x1 mux_first_14(.Y(firstShift14), .I0(D[14]), .I1(D[15]), .S(S[0]));
MUX1_2x1 mux_first_15(.Y(firstShift15), .I0(D[15]), .I1(D[16]), .S(S[0]));
MUX1_2x1 mux_first_16(.Y(firstShift16), .I0(D[16]), .I1(D[17]), .S(S[0]));
MUX1_2x1 mux_first_17(.Y(firstShift17), .I0(D[17]), .I1(D[18]), .S(S[0]));
MUX1_2x1 mux_first_18(.Y(firstShift18), .I0(D[18]), .I1(D[19]), .S(S[0]));
MUX1_2x1 mux_first_19(.Y(firstShift19), .I0(D[19]), .I1(D[20]), .S(S[0]));
MUX1_2x1 mux_first_20(.Y(firstShift20), .I0(D[20]), .I1(D[21]), .S(S[0]));
MUX1_2x1 mux_first_21(.Y(firstShift21), .I0(D[21]), .I1(D[22]), .S(S[0]));
MUX1_2x1 mux_first_22(.Y(firstShift22), .I0(D[22]), .I1(D[23]), .S(S[0]));
MUX1_2x1 mux_first_23(.Y(firstShift23), .I0(D[23]), .I1(D[24]), .S(S[0]));
MUX1_2x1 mux_first_24(.Y(firstShift24), .I0(D[24]), .I1(D[25]), .S(S[0]));
MUX1_2x1 mux_first_25(.Y(firstShift25), .I0(D[25]), .I1(D[26]), .S(S[0]));
MUX1_2x1 mux_first_26(.Y(firstShift26), .I0(D[26]), .I1(D[27]), .S(S[0]));
MUX1_2x1 mux_first_27(.Y(firstShift27), .I0(D[27]), .I1(D[28]), .S(S[0]));
MUX1_2x1 mux_first_28(.Y(firstShift28), .I0(D[28]), .I1(D[29]), .S(S[0]));
MUX1_2x1 mux_first_29(.Y(firstShift29), .I0(D[29]), .I1(D[30]), .S(S[0]));
MUX1_2x1 mux_first_30(.Y(firstShift30), .I0(D[30]), .I1(D[31]), .S(S[0]));
MUX1_2x1 mux_first_31(.Y(firstShift31), .I0(D[31]), .I1(zeroBit), .S(S[0]));
```

```
MUX1_2x1 mux_second_0(.Y(secondShift0), .I0(firstShift0), .I1(firstShift2), .S(S[1]));
MUX1_2x1 mux_second_1(.Y(secondShift1), .I0(firstShift1), .I1(firstShift3), .S(S[1]));
MUX1_2x1 mux_second_2(.Y(secondShift2), .I0(firstShift2), .I1(firstShift4), .S(S[1]));
MUX1_2x1 mux_second_3(.Y(secondShift3), .I0(firstShift3), .I1(firstShift5), .S(S[1]));
MUX1_2x1 mux_second_4(.Y(secondShift4), .I0(firstShift4), .I1(firstShift6), .S(S[1]));
MUX1_2x1 mux_second_5(.Y(secondShift5), .I0(firstShift5), .I1(firstShift7), .S(S[1]));
MUX1_2x1 mux_second_6(.Y(secondShift6), .I0(firstShift6), .I1(firstShift8), .S(S[1]));
MUX1_2x1 mux_second_7(.Y(secondShift7), .I0(firstShift7), .I1(firstShift9), .S(S[1]));
MUX1_2x1 mux_second_8(.Y(secondShift8), .I0(firstShift8), .I1(firstShift10), .S(S[1]));
MUX1_2x1 mux_second_9(.Y(secondShift9), .I0(firstShift9), .I1(firstShift11), .S(S[1]));
MUX1_2x1 mux_second_10(.Y(secondShift10), .I0(firstShift10), .I1(firstShift12), .S(S[1]));
MUX1_2x1 mux_second_11(.Y(secondShift11), .I0(firstShift11), .I1(firstShift13), .S(S[1]));
MUX1_2x1 mux_second_12(.Y(secondShift12), .I0(firstShift12), .I1(firstShift14), .S(S[1]));
MUX1_2x1 mux_second_13(.Y(secondShift13), .I0(firstShift13), .I1(firstShift15), .S(S[1]));
MUX1_2x1 mux_second_14(.Y(secondShift14), .I0(firstShift14), .I1(firstShift16), .S(S[1]));
MUX1_2x1 mux_second_15(.Y(secondShift15), .I0(firstShift15), .I1(firstShift17), .S(S[1]));
MUX1_2x1 mux_second_16(.Y(secondShift16), .I0(firstShift16), .I1(firstShift18), .S(S[1]));
MUX1_2x1 mux_second_17(.Y(secondShift17), .I0(firstShift17), .I1(firstShift19), .S(S[1]));
MUX1_2x1 mux_second_18(.Y(secondShift18), .I0(firstShift18), .I1(firstShift20), .S(S[1]));
MUX1_2x1 mux_second_19(.Y(secondShift19), .I0(firstShift19), .I1(firstShift21), .S(S[1]));
MUX1_2x1 mux_second_20(.Y(secondShift20), .I0(firstShift20), .I1(firstShift22), .S(S[1]));
MUX1_2x1 mux_second_21(.Y(secondShift21), .I0(firstShift21), .I1(firstShift23), .S(S[1]));
MUX1_2x1 mux_second_22(.Y(secondShift22), .I0(firstShift22), .I1(firstShift24), .S(S[1]));
MUX1_2x1 mux_second_23(.Y(secondShift23), .I0(firstShift23), .I1(firstShift25), .S(S[1]));
MUX1_2x1 mux_second_24(.Y(secondShift24), .I0(firstShift24), .I1(firstShift26), .S(S[1]));
MUX1_2x1 mux_second_25(.Y(secondShift25), .I0(firstShift25), .I1(firstShift27), .S(S[1]));
MUX1_2x1 mux_second_26(.Y(secondShift26), .I0(firstShift26), .I1(firstShift28), .S(S[1]));
MUX1_2x1 mux_second_27(.Y(secondShift27), .I0(firstShift27), .I1(firstShift29), .S(S[1]));
MUX1_2x1 mux_second_28(.Y(secondShift28), .I0(firstShift28), .I1(firstShift30), .S(S[1]));
MUX1_2x1 mux_second_29(.Y(secondShift29), .I0(firstShift29), .I1(firstShift31), .S(S[1]));
MUX1_2x1 mux_second_30(.Y(secondShift30), .I0(firstShift30), .I1(zeroBit), .S(S[1]));
MUX1_2x1 mux_second_31(.Y(secondShift31), .I0(firstShift31), .I1(zeroBit), .S(S[1]));
```

```
MUX1_2x1 mux_third_0(.Y(thirdShift0), .I0(secondShift0), .I1(secondShift4), .S(S[2]));
MUX1_2x1 mux_third_1(.Y(thirdShift1), .I0(secondShift1), .I1(secondShift5), .S(S[2]));
MUX1_2x1 mux_third_2(.Y(thirdShift2), .I0(secondShift2), .I1(secondShift6), .S(S[2]));
MUX1_2x1 mux_third_3(.Y(thirdShift3), .I0(secondShift3), .I1(secondShift7), .S(S[2]));
MUX1_2x1 mux_third_4(.Y(thirdShift4), .I0(secondShift4), .I1(secondShift8), .S(S[2]));
MUX1_2x1 mux_third_5(.Y(thirdShift5), .I0(secondShift5), .I1(secondShift9), .S(S[2]));
MUX1_2x1 mux_third_6(.Y(thirdShift6), .I0(secondShift6), .I1(secondShift10), .S(S[2]));
MUX1_2x1 mux_third_7(.Y(thirdShift7), .I0(secondShift7), .I1(secondShift11), .S(S[2]));
MUX1_2x1 mux_third_8(.Y(thirdShift8), .I0(secondShift8), .I1(secondShift12), .S(S[2]));
MUX1_2x1 mux_third_9(.Y(thirdShift9), .I0(secondShift9), .I1(secondShift13), .S(S[2]));
MUX1_2x1 mux_third_10(.Y(thirdShift10), .I0(secondShift10), .I1(secondShift14), .S(S[2]));
MUX1_2x1 mux_third_11(.Y(thirdShift11), .I0(secondShift11), .I1(secondShift15), .S(S[2]));
MUX1_2x1 mux_third_12(.Y(thirdShift12), .I0(secondShift12), .I1(secondShift16), .S(S[2]));
MUX1_2x1 mux_third_13(.Y(thirdShift13), .I0(secondShift13), .I1(secondShift17), .S(S[2]));
MUX1_2x1 mux_third_14(.Y(thirdShift14), .I0(secondShift14), .I1(secondShift18), .S(S[2]));
MUX1_2x1 mux_third_15(.Y(thirdShift15), .I0(secondShift15), .I1(secondShift19), .S(S[2]));
MUX1_2x1 mux_third_16(.Y(thirdShift16), .I0(secondShift16), .I1(secondShift20), .S(S[2]));
MUX1_2x1 mux_third_17(.Y(thirdShift17), .I0(secondShift17), .I1(secondShift21), .S(S[2]));
MUX1_2x1 mux_third_18(.Y(thirdShift18), .I0(secondShift18), .I1(secondShift22), .S(S[2]));
MUX1_2x1 mux_third_19(.Y(thirdShift19), .I0(secondShift19), .I1(secondShift23), .S(S[2]));
MUX1_2x1 mux_third_20(.Y(thirdShift20), .I0(secondShift20), .I1(secondShift24), .S(S[2]));
MUX1_2x1 mux_third_21(.Y(thirdShift21), .I0(secondShift21), .I1(secondShift25), .S(S[2]));
MUX1_2x1 mux_third_22(.Y(thirdShift22), .I0(secondShift22), .I1(secondShift26), .S(S[2]));
MUX1_2x1 mux_third_23(.Y(thirdShift23), .I0(secondShift23), .I1(secondShift27), .S(S[2]));
MUX1_2x1 mux_third_24(.Y(thirdShift24), .I0(secondShift24), .I1(secondShift28), .S(S[2]));
MUX1_2x1 mux_third_25(.Y(thirdShift25), .I0(secondShift25), .I1(secondShift29), .S(S[2]));
MUX1_2x1 mux_third_26(.Y(thirdShift26), .I0(secondShift26), .I1(secondShift30), .S(S[2]));
MUX1_2x1 mux_third_27(.Y(thirdShift27), .I0(secondShift27), .I1(secondShift31), .S(S[2]));
MUX1_2x1 mux_third_28(.Y(thirdShift28), .I0(secondShift28), .I1(zeroBit), .S(S[2])); // fourth
MUX1_2x1 mux_third_29(.Y(thirdShift29), .I0(secondShift29), .I1(zeroBit), .S(S[2]));
MUX1_2x1 mux_third_30(.Y(thirdShift30), .I0(secondShift30), .I1(zeroBit), .S(S[2]));
MUX1_2x1 mux_third_31(.Y(thirdShift31), .I0(secondShift31), .I1(zeroBit), .S(S[2]));
```

```
MUX1_2x1 mux_fourth_0(.Y(fourthShift0), .I0(thirdShift0), .I1(thirdShift8), .S(S[3]));
MUX1_2x1 mux_fourth_1(.Y(fourthShift1), .I0(thirdShift1), .I1(thirdShift9), .S(S[3]));
MUX1_2x1 mux_fourth_2(.Y(fourthShift2), .I0(thirdShift2), .I1(thirdShift10), .S(S[3]));
MUX1_2x1 mux_fourth_3(.Y(fourthShift3), .I0(thirdShift3), .I1(thirdShift11), .S(S[3]));
MUX1_2x1 mux_fourth_4(.Y(fourthShift4), .I0(thirdShift4), .I1(thirdShift12), .S(S[3]));
MUX1_2x1 mux_fourth_5(.Y(fourthShift5), .I0(thirdShift5), .I1(thirdShift13), .S(S[3]));
MUX1_2x1 mux_fourth_6(.Y(fourthShift6), .I0(thirdShift6), .I1(thirdShift14), .S(S[3]));
MUX1_2x1 mux_fourth_7(.Y(fourthShift7), .I0(thirdShift7), .I1(thirdShift15), .S(S[3]));
MUX1_2x1 mux_fourth_8(.Y(fourthShift8), .I0(thirdShift8), .I1(thirdShift16), .S(S[3]));
MUX1_2x1 mux_fourth_9(.Y(fourthShift9), .I0(thirdShift9), .I1(thirdShift17), .S(S[3]));
MUX1_2x1 mux_fourth_10(.Y(fourthShift10), .I0(thirdShift10), .I1(thirdShift18), .S(S[3]));
MUX1_2x1 mux_fourth_11(.Y(fourthShift11), .I0(thirdShift11), .I1(thirdShift19), .S(S[3]));
MUX1_2x1 mux_fourth_12(.Y(fourthShift12), .I0(thirdShift12), .I1(thirdShift20), .S(S[3]));
MUX1_2x1 mux_fourth_13(.Y(fourthShift13), .I0(thirdShift13), .I1(thirdShift21), .S(S[3]));
MUX1_2x1 mux_fourth_14(.Y(fourthShift14), .I0(thirdShift14), .I1(thirdShift22), .S(S[3]));
MUX1_2x1 mux_fourth_15(.Y(fourthShift15), .I0(thirdShift15), .I1(thirdShift23), .S(S[3]));
MUX1_2x1 mux_fourth_16(.Y(fourthShift16), .I0(thirdShift16), .I1(thirdShift24), .S(S[3]));
MUX1_2x1 mux_fourth_17(.Y(fourthShift17), .I0(thirdShift17), .I1(thirdShift25), .S(S[3]));
MUX1_2x1 mux_fourth_18(.Y(fourthShift18), .I0(thirdShift18), .I1(thirdShift26), .S(S[3]));
MUX1_2x1 mux_fourth_19(.Y(fourthShift19), .I0(thirdShift19), .I1(thirdShift27), .S(S[3]));
MUX1_2x1 mux_fourth_20(.Y(fourthShift20), .I0(thirdShift20), .I1(thirdShift28), .S(S[3]));
MUX1_2x1 mux_fourth_21(.Y(fourthShift21), .I0(thirdShift21), .I1(thirdShift29), .S(S[3]));
MUX1_2x1 mux_fourth_22(.Y(fourthShift22), .I0(thirdShift22), .I1(thirdShift30), .S(S[3]));
MUX1_2x1 mux_fourth_23(.Y(fourthShift23), .I0(thirdShift23), .I1(thirdShift31), .S(S[3]));
MUX1_2x1 mux_fourth_24(.Y(fourthShift24), .I0(thirdShift24), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_25(.Y(fourthShift25), .I0(thirdShift25), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_26(.Y(fourthShift26), .I0(thirdShift26), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_27(.Y(fourthShift27), .I0(thirdShift27), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_28(.Y(fourthShift28), .I0(thirdShift28), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_29(.Y(fourthShift29), .I0(thirdShift29), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_30(.Y(fourthShift30), .I0(thirdShift30), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_31(.Y(fourthShift31), .I0(thirdShift31), .I1(zeroBit), .S(S[3]));
```

```
MUX1_2x1 mux_fifth_0(.Y(Y[0]), .I0(fourthShift0), .I1(fourthShift16), .S(S[4]));
MUX1_2x1 mux_fifth_1(.Y(Y[1]), .I0(fourthShift1), .I1(fourthShift17), .S(S[4]));
MUX1_2x1 mux_fifth_2(.Y(Y[2]), .I0(fourthShift2), .I1(fourthShift18), .S(S[4]));
MUX1_2x1 mux_fifth_3(.Y(Y[3]), .I0(fourthShift3), .I1(fourthShift19), .S(S[4]));
MUX1_2x1 mux_fifth_4(.Y(Y[4]), .I0(fourthShift4), .I1(fourthShift20), .S(S[4]));
MUX1_2x1 mux_fifth_5(.Y(Y[5]), .I0(fourthShift5), .I1(fourthShift21), .S(S[4]));
MUX1_2x1 mux_fifth_6(.Y(Y[6]), .I0(fourthShift6), .I1(fourthShift22), .S(S[4]));
MUX1_2x1 mux_fifth_7(.Y(Y[7]), .I0(fourthShift7), .I1(fourthShift23), .S(S[4]));
MUX1_2x1 mux_fifth_8(.Y(Y[8]), .I0(fourthShift8), .I1(fourthShift24), .S(S[4]));
MUX1_2x1 mux_fifth_9(.Y(Y[9]), .I0(fourthShift9), .I1(fourthShift25), .S(S[4]));
MUX1_2x1 mux_fifth_10(.Y(Y[10]), .I0(fourthShift10), .I1(fourthShift26), .S(S[4]));
MUX1_2x1 mux_fifth_11(.Y(Y[11]), .I0(fourthShift11), .I1(fourthShift27), .S(S[4]));
MUX1_2x1 mux_fifth_12(.Y(Y[12]), .I0(fourthShift12), .I1(fourthShift28), .S(S[4]));
MUX1_2x1 mux_fifth_13(.Y(Y[13]), .I0(fourthShift13), .I1(fourthShift29), .S(S[4]));
MUX1_2x1 mux_fifth_14(.Y(Y[14]), .I0(fourthShift14), .I1(fourthShift30), .S(S[4]));
MUX1_2x1 mux_fifth_15(.Y(Y[15]), .I0(fourthShift15), .I1(fourthShift31), .S(S[4]));
MUX1_2x1 mux_fifth_16(.Y(Y[16]), .I0(fourthShift16), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_17(.Y(Y[17]), .I0(fourthShift17), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_18(.Y(Y[18]), .I0(fourthShift18), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_19(.Y(Y[19]), .I0(fourthShift19), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_20(.Y(Y[20]), .I0(fourthShift20), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_21(.Y(Y[21]), .I0(fourthShift21), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_22(.Y(Y[22]), .I0(fourthShift22), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_23(.Y(Y[23]), .I0(fourthShift23), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_24(.Y(Y[24]), .I0(fourthShift24), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_25(.Y(Y[25]), .I0(fourthShift25), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_26(.Y(Y[26]), .I0(fourthShift26), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_27(.Y(Y[27]), .I0(fourthShift27), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_28(.Y(Y[28]), .I0(fourthShift28), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_29(.Y(Y[29]), .I0(fourthShift29), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_30(.Y(Y[30]), .I0(fourthShift30), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_31(.Y(Y[31]), .I0(fourthShift31), .I1(zeroBit), .S(S[4]));

endmodule
```

(I'll show all the testing for the shifters later as I use the same cases for each and the testing is all on the same file.)

Left Shifter:

```
// Left shifter
module SHIFT32_L(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

// TBD

wire firstShift0;
wire firstShift1;
wire firstShift2;
wire firstShift3;
wire firstShift4;
wire firstShift5;
wire firstShift6;
wire firstShift7;
wire firstShift8;
wire firstShift9;
wire firstShift10;
wire firstShift11;
wire firstShift12;
wire firstShift13;
```

Once again, it uses the same wire set up as the right shifter. It also shares the zero bit wire.

```
MUX1_2x1 mux_first_0(.Y(firstShift0), .I0(D[0]), .I1(zeroBit), .S(S[0]));
MUX1_2x1 mux_first_1(.Y(firstShift1), .I0(D[1]), .I1(D[0]), .S(S[0]));
MUX1_2x1 mux_first_2(.Y(firstShift2), .I0(D[2]), .I1(D[1]), .S(S[0]));
MUX1_2x1 mux_first_3(.Y(firstShift3), .I0(D[3]), .I1(D[2]), .S(S[0]));
MUX1_2x1 mux_first_4(.Y(firstShift4), .I0(D[4]), .I1(D[3]), .S(S[0]));
MUX1_2x1 mux_first_5(.Y(firstShift5), .I0(D[5]), .I1(D[4]), .S(S[0]));
MUX1_2x1 mux_first_6(.Y(firstShift6), .I0(D[6]), .I1(D[5]), .S(S[0]));
MUX1_2x1 mux_first_7(.Y(firstShift7), .I0(D[7]), .I1(D[6]), .S(S[0]));
MUX1_2x1 mux_first_8(.Y(firstShift8), .I0(D[8]), .I1(D[7]), .S(S[0]));
MUX1_2x1 mux_first_9(.Y(firstShift9), .I0(D[9]), .I1(D[8]), .S(S[0]));
MUX1_2x1 mux_first_10(.Y(firstShift10), .I0(D[10]), .I1(D[9]), .S(S[0]));
MUX1_2x1 mux_first_11(.Y(firstShift11), .I0(D[11]), .I1(D[10]), .S(S[0]));
MUX1_2x1 mux_first_12(.Y(firstShift12), .I0(D[12]), .I1(D[11]), .S(S[0]));
MUX1_2x1 mux_first_13(.Y(firstShift13), .I0(D[13]), .I1(D[12]), .S(S[0]));
MUX1_2x1 mux_first_14(.Y(firstShift14), .I0(D[14]), .I1(D[13]), .S(S[0]));
MUX1_2x1 mux_first_15(.Y(firstShift15), .I0(D[15]), .I1(D[14]), .S(S[0]));
MUX1_2x1 mux_first_16(.Y(firstShift16), .I0(D[16]), .I1(D[15]), .S(S[0]));
MUX1_2x1 mux_first_17(.Y(firstShift17), .I0(D[17]), .I1(D[16]), .S(S[0]));
MUX1_2x1 mux_first_18(.Y(firstShift18), .I0(D[18]), .I1(D[17]), .S(S[0]));
MUX1_2x1 mux_first_19(.Y(firstShift19), .I0(D[19]), .I1(D[18]), .S(S[0]));
MUX1_2x1 mux_first_20(.Y(firstShift20), .I0(D[20]), .I1(D[19]), .S(S[0]));
MUX1_2x1 mux_first_21(.Y(firstShift21), .I0(D[21]), .I1(D[20]), .S(S[0]));
MUX1_2x1 mux_first_22(.Y(firstShift22), .I0(D[22]), .I1(D[21]), .S(S[0]));
MUX1_2x1 mux_first_23(.Y(firstShift23), .I0(D[23]), .I1(D[22]), .S(S[0]));
MUX1_2x1 mux_first_24(.Y(firstShift24), .I0(D[24]), .I1(D[23]), .S(S[0]));
MUX1_2x1 mux_first_25(.Y(firstShift25), .I0(D[25]), .I1(D[24]), .S(S[0]));
MUX1_2x1 mux_first_26(.Y(firstShift26), .I0(D[26]), .I1(D[25]), .S(S[0]));
MUX1_2x1 mux_first_27(.Y(firstShift27), .I0(D[27]), .I1(D[26]), .S(S[0]));
MUX1_2x1 mux_first_28(.Y(firstShift28), .I0(D[28]), .I1(D[27]), .S(S[0]));
MUX1_2x1 mux_first_29(.Y(firstShift29), .I0(D[29]), .I1(D[28]), .S(S[0]));
MUX1_2x1 mux_first_30(.Y(firstShift30), .I0(D[30]), .I1(D[29]), .S(S[0]));
MUX1_2x1 mux_first_31(.Y(firstShift31), .I0(D[31]), .I1(D[30]), .S(S[0]));
```

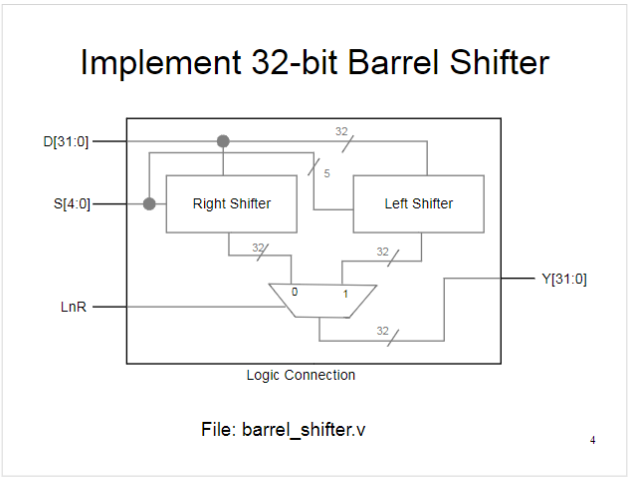
```
MUX1_2x1 mux_second_0(.Y(secondShift0), .I0(firstShift0), .I1(zeroBit), .S(S[1]));
MUX1_2x1 mux_second_1(.Y(secondShift1), .I0(firstShift1), .I1(zeroBit), .S(S[1]));
MUX1_2x1 mux_second_2(.Y(secondShift2), .I0(firstShift2), .I1(firstShift0), .S(S[1]));
MUX1_2x1 mux_second_3(.Y(secondShift3), .I0(firstShift3), .I1(firstShift1), .S(S[1]));
MUX1_2x1 mux_second_4(.Y(secondShift4), .I0(firstShift4), .I1(firstShift2), .S(S[1]));
MUX1_2x1 mux_second_5(.Y(secondShift5), .I0(firstShift5), .I1(firstShift3), .S(S[1]));
MUX1_2x1 mux_second_6(.Y(secondShift6), .I0(firstShift6), .I1(firstShift4), .S(S[1]));
MUX1_2x1 mux_second_7(.Y(secondShift7), .I0(firstShift7), .I1(firstShift5), .S(S[1]));
MUX1_2x1 mux_second_8(.Y(secondShift8), .I0(firstShift8), .I1(firstShift6), .S(S[1]));
MUX1_2x1 mux_second_9(.Y(secondShift9), .I0(firstShift9), .I1(firstShift7), .S(S[1]));
MUX1_2x1 mux_second_10(.Y(secondShift10), .I0(firstShift10), .I1(firstShift8), .S(S[1]));
MUX1_2x1 mux_second_11(.Y(secondShift11), .I0(firstShift11), .I1(firstShift9), .S(S[1]));
MUX1_2x1 mux_second_12(.Y(secondShift12), .I0(firstShift12), .I1(firstShift10), .S(S[1]));
MUX1_2x1 mux_second_13(.Y(secondShift13), .I0(firstShift13), .I1(firstShift11), .S(S[1]));
MUX1_2x1 mux_second_14(.Y(secondShift14), .I0(firstShift14), .I1(firstShift12), .S(S[1]));
MUX1_2x1 mux_second_15(.Y(secondShift15), .I0(firstShift15), .I1(firstShift13), .S(S[1]));
MUX1_2x1 mux_second_16(.Y(secondShift16), .I0(firstShift16), .I1(firstShift14), .S(S[1]));
MUX1_2x1 mux_second_17(.Y(secondShift17), .I0(firstShift17), .I1(firstShift15), .S(S[1]));
MUX1_2x1 mux_second_18(.Y(secondShift18), .I0(firstShift18), .I1(firstShift16), .S(S[1]));
MUX1_2x1 mux_second_19(.Y(secondShift19), .I0(firstShift19), .I1(firstShift17), .S(S[1]));
MUX1_2x1 mux_second_20(.Y(secondShift20), .I0(firstShift20), .I1(firstShift18), .S(S[1]));
MUX1_2x1 mux_second_21(.Y(secondShift21), .I0(firstShift21), .I1(firstShift19), .S(S[1]));
MUX1_2x1 mux_second_22(.Y(secondShift22), .I0(firstShift22), .I1(firstShift20), .S(S[1]));
MUX1_2x1 mux_second_23(.Y(secondShift23), .I0(firstShift23), .I1(firstShift21), .S(S[1]));
MUX1_2x1 mux_second_24(.Y(secondShift24), .I0(firstShift24), .I1(firstShift22), .S(S[1]));
MUX1_2x1 mux_second_25(.Y(secondShift25), .I0(firstShift25), .I1(firstShift23), .S(S[1]));
MUX1_2x1 mux_second_26(.Y(secondShift26), .I0(firstShift26), .I1(firstShift24), .S(S[1]));
MUX1_2x1 mux_second_27(.Y(secondShift27), .I0(firstShift27), .I1(firstShift25), .S(S[1]));
MUX1_2x1 mux_second_28(.Y(secondShift28), .I0(firstShift28), .I1(firstShift26), .S(S[1]));
MUX1_2x1 mux_second_29(.Y(secondShift29), .I0(firstShift29), .I1(firstShift27), .S(S[1]));
MUX1_2x1 mux_second_30(.Y(secondShift30), .I0(firstShift30), .I1(firstShift28), .S(S[1]));
MUX1_2x1 mux_second_31(.Y(secondShift31), .I0(firstShift31), .I1(firstShift29), .S(S[1]));
```

```
MUX1_2x1 mux_third_0(.Y(thirdShift0), .I0(secondShift0), .I1(zeroBit), .S(S[2]));
MUX1_2x1 mux_third_1(.Y(thirdShift1), .I0(secondShift1), .I1(zeroBit), .S(S[2]));
MUX1_2x1 mux_third_2(.Y(thirdShift2), .I0(secondShift2), .I1(zeroBit), .S(S[2]));
MUX1_2x1 mux_third_3(.Y(thirdShift3), .I0(secondShift3), .I1(zeroBit), .S(S[2]));
MUX1_2x1 mux_third_4(.Y(thirdShift4), .I0(secondShift4), .I1(secondShift0), .S(S[2]));
MUX1_2x1 mux_third_5(.Y(thirdShift5), .I0(secondShift5), .I1(secondShift1), .S(S[2]));
MUX1_2x1 mux_third_6(.Y(thirdShift6), .I0(secondShift6), .I1(secondShift2), .S(S[2]));
MUX1_2x1 mux_third_7(.Y(thirdShift7), .I0(secondShift7), .I1(secondShift3), .S(S[2]));
MUX1_2x1 mux_third_8(.Y(thirdShift8), .I0(secondShift8), .I1(secondShift4), .S(S[2]));
MUX1_2x1 mux_third_9(.Y(thirdShift9), .I0(secondShift9), .I1(secondShift5), .S(S[2]));
MUX1_2x1 mux_third_10(.Y(thirdShift10), .I0(secondShift10), .I1(secondShift6), .S(S[2]));
MUX1_2x1 mux_third_11(.Y(thirdShift11), .I0(secondShift11), .I1(secondShift7), .S(S[2]));
MUX1_2x1 mux_third_12(.Y(thirdShift12), .I0(secondShift12), .I1(secondShift8), .S(S[2]));
MUX1_2x1 mux_third_13(.Y(thirdShift13), .I0(secondShift13), .I1(secondShift9), .S(S[2]));
MUX1_2x1 mux_third_14(.Y(thirdShift14), .I0(secondShift14), .I1(secondShift10), .S(S[2]));
MUX1_2x1 mux_third_15(.Y(thirdShift15), .I0(secondShift15), .I1(secondShift11), .S(S[2]));
MUX1_2x1 mux_third_16(.Y(thirdShift16), .I0(secondShift16), .I1(secondShift12), .S(S[2]));
MUX1_2x1 mux_third_17(.Y(thirdShift17), .I0(secondShift17), .I1(secondShift13), .S(S[2]));
MUX1_2x1 mux_third_18(.Y(thirdShift18), .I0(secondShift18), .I1(secondShift14), .S(S[2]));
MUX1_2x1 mux_third_19(.Y(thirdShift19), .I0(secondShift19), .I1(secondShift15), .S(S[2]));
MUX1_2x1 mux_third_20(.Y(thirdShift20), .I0(secondShift20), .I1(secondShift16), .S(S[2]));
MUX1_2x1 mux_third_21(.Y(thirdShift21), .I0(secondShift21), .I1(secondShift17), .S(S[2]));
MUX1_2x1 mux_third_22(.Y(thirdShift22), .I0(secondShift22), .I1(secondShift18), .S(S[2]));
MUX1_2x1 mux_third_23(.Y(thirdShift23), .I0(secondShift23), .I1(secondShift19), .S(S[2]));
MUX1_2x1 mux_third_24(.Y(thirdShift24), .I0(secondShift24), .I1(secondShift20), .S(S[2]));
MUX1_2x1 mux_third_25(.Y(thirdShift25), .I0(secondShift25), .I1(secondShift21), .S(S[2]));
MUX1_2x1 mux_third_26(.Y(thirdShift26), .I0(secondShift26), .I1(secondShift22), .S(S[2]));
MUX1_2x1 mux_third_27(.Y(thirdShift27), .I0(secondShift27), .I1(secondShift23), .S(S[2]));
MUX1_2x1 mux_third_28(.Y(thirdShift28), .I0(secondShift28), .I1(secondShift24), .S(S[2]));
MUX1_2x1 mux_third_29(.Y(thirdShift29), .I0(secondShift29), .I1(secondShift25), .S(S[2]));
MUX1_2x1 mux_third_30(.Y(thirdShift30), .I0(secondShift30), .I1(secondShift26), .S(S[2]));
MUX1_2x1 mux_third_31(.Y(thirdShift31), .I0(secondShift31), .I1(secondShift27), .S(S[2]));
```

```
MUX1_2x1 mux_fourth_0(.Y(fourthShift0), .I0(thirdShift0), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_1(.Y(fourthShift1), .I0(thirdShift1), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_2(.Y(fourthShift2), .I0(thirdShift2), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_3(.Y(fourthShift3), .I0(thirdShift3), .I1(zeroBit), .S(S[3])); //4th
MUX1_2x1 mux_fourth_4(.Y(fourthShift4), .I0(thirdShift4), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_5(.Y(fourthShift5), .I0(thirdShift5), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_6(.Y(fourthShift6), .I0(thirdShift6), .I1(zeroBit), .S(S[3]));
MUX1_2x1 mux_fourth_7(.Y(fourthShift7), .I0(thirdShift7), .I1(zeroBit), .S(S[3])); //8th
MUX1_2x1 mux_fourth_8(.Y(fourthShift8), .I0(thirdShift8), .I1(thirdShift0), .S(S[3]));
MUX1_2x1 mux_fourth_9(.Y(fourthShift9), .I0(thirdShift9), .I1(thirdShift1), .S(S[3]));
MUX1_2x1 mux_fourth_10(.Y(fourthShift10), .I0(thirdShift10), .I1(thirdShift2), .S(S[3]));
MUX1_2x1 mux_fourth_11(.Y(fourthShift11), .I0(thirdShift11), .I1(thirdShift3), .S(S[3]));
MUX1_2x1 mux_fourth_12(.Y(fourthShift12), .I0(thirdShift12), .I1(thirdShift4), .S(S[3]));
MUX1_2x1 mux_fourth_13(.Y(fourthShift13), .I0(thirdShift13), .I1(thirdShift5), .S(S[3]));
MUX1_2x1 mux_fourth_14(.Y(fourthShift14), .I0(thirdShift14), .I1(thirdShift6), .S(S[3]));
MUX1_2x1 mux_fourth_15(.Y(fourthShift15), .I0(thirdShift15), .I1(thirdShift7), .S(S[3]));
MUX1_2x1 mux_fourth_16(.Y(fourthShift16), .I0(thirdShift16), .I1(thirdShift8), .S(S[3]));
MUX1_2x1 mux_fourth_17(.Y(fourthShift17), .I0(thirdShift17), .I1(thirdShift9), .S(S[3]));
MUX1_2x1 mux_fourth_18(.Y(fourthShift18), .I0(thirdShift18), .I1(thirdShift10), .S(S[3]));
MUX1_2x1 mux_fourth_19(.Y(fourthShift19), .I0(thirdShift19), .I1(thirdShift11), .S(S[3]));
MUX1_2x1 mux_fourth_20(.Y(fourthShift20), .I0(thirdShift20), .I1(thirdShift12), .S(S[3]));
MUX1_2x1 mux_fourth_21(.Y(fourthShift21), .I0(thirdShift21), .I1(thirdShift13), .S(S[3]));
MUX1_2x1 mux_fourth_22(.Y(fourthShift22), .I0(thirdShift22), .I1(thirdShift14), .S(S[3]));
MUX1_2x1 mux_fourth_23(.Y(fourthShift23), .I0(thirdShift23), .I1(thirdShift15), .S(S[3]));
MUX1_2x1 mux_fourth_24(.Y(fourthShift24), .I0(thirdShift24), .I1(thirdShift16), .S(S[3]));
MUX1_2x1 mux_fourth_25(.Y(fourthShift25), .I0(thirdShift25), .I1(thirdShift17), .S(S[3]));
MUX1_2x1 mux_fourth_26(.Y(fourthShift26), .I0(thirdShift26), .I1(thirdShift18), .S(S[3]));
MUX1_2x1 mux_fourth_27(.Y(fourthShift27), .I0(thirdShift27), .I1(thirdShift19), .S(S[3]));
MUX1_2x1 mux_fourth_28(.Y(fourthShift28), .I0(thirdShift28), .I1(thirdShift20), .S(S[3]));
MUX1_2x1 mux_fourth_29(.Y(fourthShift29), .I0(thirdShift29), .I1(thirdShift21), .S(S[3]));
MUX1_2x1 mux_fourth_30(.Y(fourthShift30), .I0(thirdShift30), .I1(thirdShift22), .S(S[3]));
MUX1_2x1 mux_fourth_31(.Y(fourthShift31), .I0(thirdShift31), .I1(thirdShift23), .S(S[3]));
```

```
MUX1_2x1 mux_fifth_0(.Y(Y[0]), .I0(fourthShift0), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_1(.Y(Y[1]), .I0(fourthShift1), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_2(.Y(Y[2]), .I0(fourthShift2), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_3(.Y(Y[3]), .I0(fourthShift3), .I1(zeroBit), .S(S[4])); // 4
MUX1_2x1 mux_fifth_4(.Y(Y[4]), .I0(fourthShift4), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_5(.Y(Y[5]), .I0(fourthShift5), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_6(.Y(Y[6]), .I0(fourthShift6), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_7(.Y(Y[7]), .I0(fourthShift7), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_8(.Y(Y[8]), .I0(fourthShift8), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_9(.Y(Y[9]), .I0(fourthShift9), .I1(zeroBit), .S(S[4])); // 10
MUX1_2x1 mux_fifth_10(.Y(Y[10]), .I0(fourthShift10), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_11(.Y(Y[11]), .I0(fourthShift11), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_12(.Y(Y[12]), .I0(fourthShift12), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_13(.Y(Y[13]), .I0(fourthShift13), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_14(.Y(Y[14]), .I0(fourthShift14), .I1(zeroBit), .S(S[4]));
MUX1_2x1 mux_fifth_15(.Y(Y[15]), .I0(fourthShift15), .I1(zeroBit), .S(S[4])); //16
MUX1_2x1 mux_fifth_16(.Y(Y[16]), .I0(fourthShift16), .I1(fourthShift0), .S(S[4]));
MUX1_2x1 mux_fifth_17(.Y(Y[17]), .I0(fourthShift17), .I1(fourthShift1), .S(S[4]));
MUX1_2x1 mux_fifth_18(.Y(Y[18]), .I0(fourthShift18), .I1(fourthShift2), .S(S[4]));
MUX1_2x1 mux_fifth_19(.Y(Y[19]), .I0(fourthShift19), .I1(fourthShift3), .S(S[4]));
MUX1_2x1 mux_fifth_20(.Y(Y[20]), .I0(fourthShift20), .I1(fourthShift4), .S(S[4]));
MUX1_2x1 mux_fifth_21(.Y(Y[21]), .I0(fourthShift21), .I1(fourthShift5), .S(S[4]));
MUX1_2x1 mux_fifth_22(.Y(Y[22]), .I0(fourthShift22), .I1(fourthShift6), .S(S[4]));
MUX1_2x1 mux_fifth_23(.Y(Y[23]), .I0(fourthShift23), .I1(fourthShift7), .S(S[4]));
MUX1_2x1 mux_fifth_24(.Y(Y[24]), .I0(fourthShift24), .I1(fourthShift8), .S(S[4]));
MUX1_2x1 mux_fifth_25(.Y(Y[25]), .I0(fourthShift25), .I1(fourthShift9), .S(S[4]));
MUX1_2x1 mux_fifth_26(.Y(Y[26]), .I0(fourthShift26), .I1(fourthShift10), .S(S[4]));
MUX1_2x1 mux_fifth_27(.Y(Y[27]), .I0(fourthShift27), .I1(fourthShift11), .S(S[4]));
MUX1_2x1 mux_fifth_28(.Y(Y[28]), .I0(fourthShift28), .I1(fourthShift12), .S(S[4]));
MUX1_2x1 mux_fifth_29(.Y(Y[29]), .I0(fourthShift29), .I1(fourthShift13), .S(S[4]));
MUX1_2x1 mux_fifth_30(.Y(Y[30]), .I0(fourthShift30), .I1(fourthShift14), .S(S[4]));
MUX1_2x1 mux_fifth_31(.Y(Y[31]), .I0(fourthShift31), .I1(fourthShift15), .S(S[4]));


endmodule
```

Barrel shifter (that combines left and right):

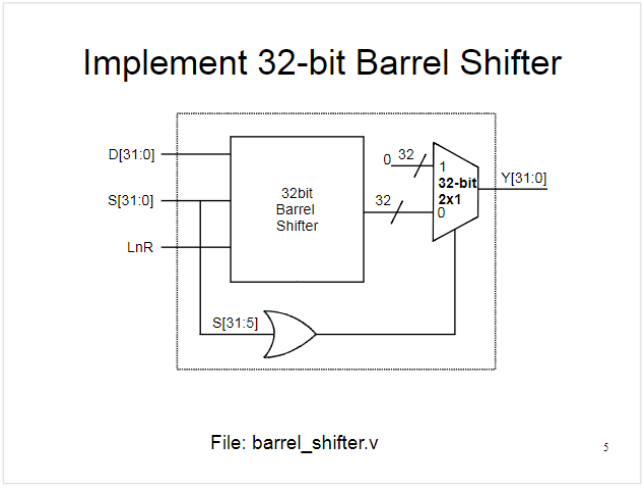


(Kaushik Patra, Lab 13, Slide 3)

When LnR is 0 then shift right, else if its 1, shift left.

```
// 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;

// TBD

wire [31:0] Y_R;
wire [31:0] Y_L;


//module SHIFT32_L(Y,D,S);
SHIFT32_L shift_left(.Y(Y_L), .D(D), .S(S));
SHIFT32_R shift_right(.Y(Y_R), .D(D), .S(S));

MUX32_2x1 mux(.Y(Y), .I0(Y_R), .I1(Y_L), .S(LnR));

endmodule
```

Barrel shifter which can detect if you're shifting more than 5 times:



(Kaushik Patra, Lab 13, Slide 3)

The point of this shifter is that if you're shifting by more than 32 times you'll just be returned 0 anyway.

```
// 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 sel;
wire [31:0] res;

wire or_res_5;
wire or_res_6;
wire or_res_7;
wire or_res_8;
wire or_res_9;
wire or_res_10;
wire or_res_11;
wire or_res_12;
wire or_res_13;
wire or_res_14;
wire or_res_15;
wire or_res_16;
wire or_res_17;
wire or_res_18;
wire or_res_19;
wire or_res_20;
wire or_res_21;
wire or_res_22;
wire or_res_23;
wire or_res_24;
wire or_res_25;
wire or_res_26;
wire or_res_27;
wire or_res_28;
wire or_res_29;
```

```
56  // TBD
57  //module BARREL_SHIFTER32(Y,D,S, LnR);
58  BARREL_SHIFTER32 shifter(.Y(res), .D(D), .S(S[4:0]), .LnR(LnR));
59
60  or or5(or_res_5, S[5], S[6]);
61  or or6(or_res_6, or_res_5, S[7]);
62  or or7(or_res_7, or_res_6, S[8]);
63  or or8(or_res_8, or_res_7, S[9]);
64  or or9(or_res_9, or_res_8, S[10]);
65  or or10(or_res_10, or_res_9, S[11]);
66  or or11(or_res_11, or_res_10, S[12]);
67  or or12(or_res_12, or_res_11, S[13]);
68  or or13(or_res_13, or_res_12, S[14]);
69  or or14(or_res_14, or_res_13, S[15]);
70  or or15(or_res_15, or_res_14, S[16]);
71  or or16(or_res_16, or_res_15, S[17]);
72  or or17(or_res_17, or_res_16, S[18]);
73  or or18(or_res_18, or_res_17, S[19]);
74  or or19(or_res_19, or_res_18, S[20]);
75  or or20(or_res_20, or_res_19, S[21]);
76  or or21(or_res_21, or_res_20, S[22]);
77  or or22(or_res_22, or_res_21, S[23]);
78  or or23(or_res_23, or_res_22, S[24]);
79  or or24(or_res_24, or_res_23, S[25]);
80  or or25(or_res_25, or_res_24, S[26]);
81  or or26(or_res_26, or_res_25, S[27]);
82  or or27(or_res_27, or_res_26, S[28]);
83  or or28(or_res_28, or_res_27, S[29]);
84  or or29(or_res_29, or_res_28, S[30]);
85  or or30(sel, or_res_29, S[31]);
86
87  MUX32_2x1 mux(.Y(Y), .I0(res), .I1(zero_32), .S(sel));
88
89  endmodule
```

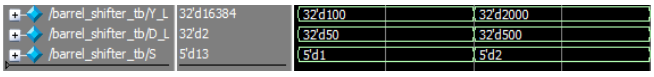
The ors are making S[31:5] into 1 bit. If any of them are 1 then the output should be 0.

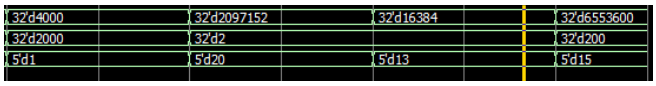
Testing (all the shifters):

```
`timescale 1ns/1ps

module barrel_shifter_tb;
wire [31:0] Y_R;
wire [31:0] Y_L;
wire [31:0] Y_Shifter;
wire [31:0] Y_32;

reg [31:0] D_L;
reg [31:0] D_R;
reg [31:0] D_S;
reg [31:0] D_32;
reg [4:0] S;
reg [31:0] S_32;
reg LnR; // if == 0 then shift right


//module SHIFT32_L(Y,D,S);
//module SHIFT32(Y,D,S, LnR);
//module BARREL_SHIFTER32(Y,D,S, LnR);
SHIFT32_L shift_left(.Y(Y_L), .D(D_L), .S(S));
SHIFT32_R shift_right(.Y(Y_R), .D(D_R), .S(S));
BARREL_SHIFTER32 shifter(.Y(Y_Shifter), .D(D_S), .S(S), .LnR(LnR));
SHIFT32 shifter32(.Y(Y_32), .D(D_32), .S(S_32), .LnR(LnR));

initial
begin
D_R=50; D_L=50; D_S = 50; D_32 = 50; S=1; S_32 = 1; LnR = 0;
#10 D_R=500; D_L=500; D_S = 500; D_32 = 500; S=2; S_32 = 2; LnR = 0;
#10 D_R=2000; D_L=2000; D_S = 2000; D_32 = 2000; S=1; S_32 = 1; LnR = 1;
#10 D_R=2; D_L=2; D_S = 2; D_32 = 2; S=20; S_32 = 20; LnR = 1;
#10 D_R=2; D_L=2; D_S = 2; D_32 = 2; S=45; S_32 = 45; LnR = 0;
#10 D_R=200; D_L=200; D_S = 200; D_32 = 200; S=15; S_32 = 15; LnR = 0;
#5 ;
end

endmodule
```

Test cases (shared by all):
D=50, Shift =1, LnR =0
D=500, Shift =2 ,LnR =0
D=2000, Shift =1, LnR =1
D=2, Shift =20, LnR =1
D=2, Shift =45, LnR =0
D=200, Shift =15, LnR =0

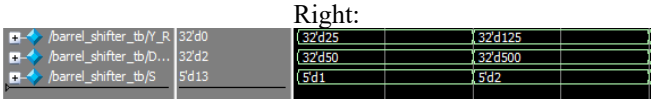
Left:



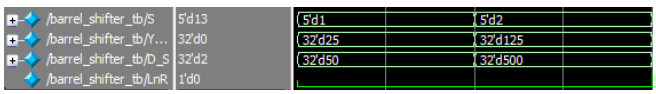


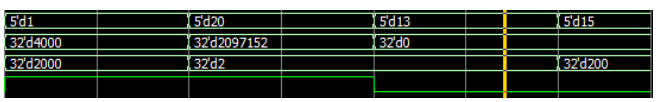
(For the 5 bit shift registers 45 isn't supported so this case can be ignored. That means every shifter except SHIFTER32.)

Right:

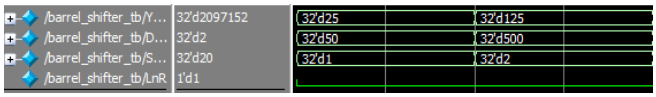


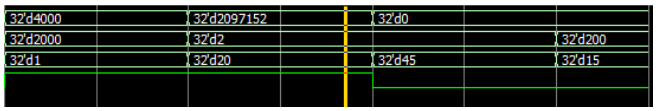
Shifter with options between L and R:





32 bit Shifter:





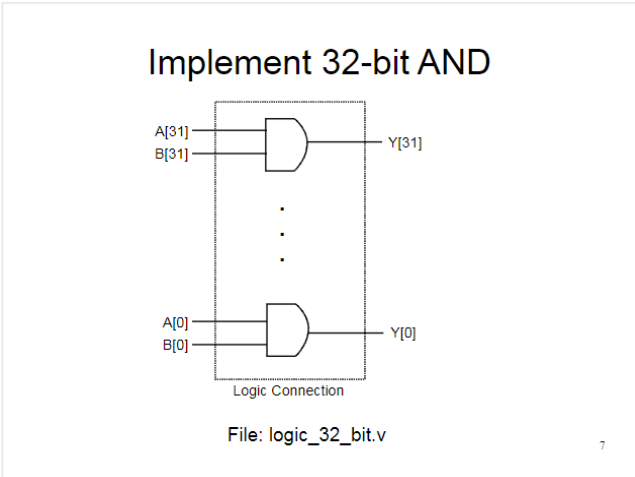
Results:
Left:
50<<1=100
500<<2=2000
2000<<1=4000
2<<20=2097152
2<<45=((Invalid result based on design))
200<<15=6553600

Right:
50>>1=25
500>>2=125
2000>>1=1000
2>>20=0
2>>45=0
200>>15=0

L and R Shifter:
50>>1=25
500>>2=125
2000<<1=4000
2<<20=2097152
2>>45=0
200>>15=0

32Bit Shifter:
50>>1=25
500>>2=125
2000<<1=4000
2<<20=2097152
2>>45=0
200>>15=0

D. 32 bit AND, NOR, INV, OR



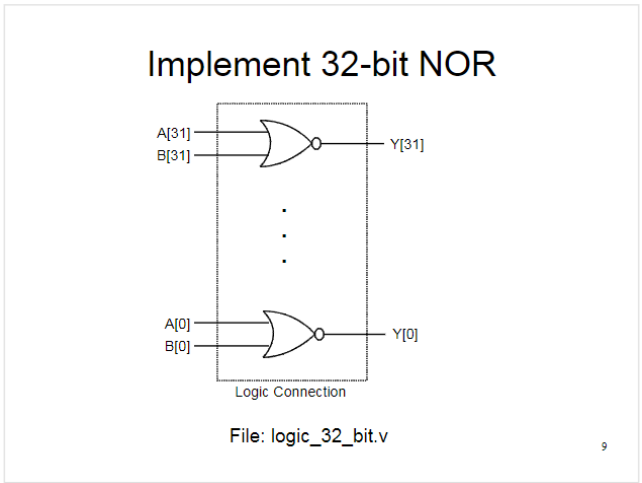
(Kaushik Patra, Lab 13, Slide 4)

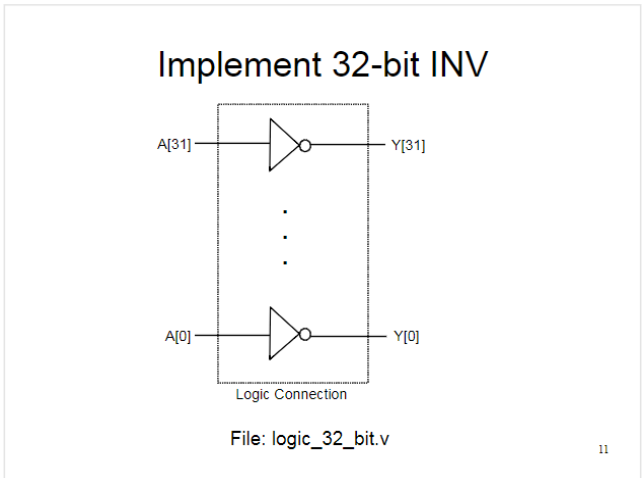
```
// 32-bit AND
module AND32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

// TBD

and and0(Y[0], A[0], B[0]);
and and1(Y[1], A[1], B[1]);
and and2(Y[2], A[2], B[2]);
and and3(Y[3], A[3], B[3]);
and and4(Y[4], A[4], B[4]);
and and5(Y[5], A[5], B[5]);
and and6(Y[6], A[6], B[6]);
and and7(Y[7], A[7], B[7]);
and and8(Y[8], A[8], B[8]);
and and9(Y[9], A[9], B[9]);
and and10(Y[10], A[10], B[10]);
and and11(Y[11], A[11], B[11]);
and and12(Y[12], A[12], B[12]);
and and13(Y[13], A[13], B[13]);
and and14(Y[14], A[14], B[14]);
and and15(Y[15], A[15], B[15]);
and and16(Y[16], A[16], B[16]);
and and17(Y[17], A[17], B[17]);
and and18(Y[18], A[18], B[18]);
and and19(Y[19], A[19], B[19]);
and and20(Y[20], A[20], B[20]);
and and21(Y[21], A[21], B[21]);
and and22(Y[22], A[22], B[22]);
and and23(Y[23], A[23], B[23]);
and and24(Y[24], A[24], B[24]);
and and25(Y[25], A[25], B[25]);
and and26(Y[26], A[26], B[26]);
and and27(Y[27], A[27], B[27]);
and and28(Y[28], A[28], B[28]);
and and29(Y[29], A[29], B[29]);
and and30(Y[30], A[30], B[30]);
and and31(Y[31], A[31], B[31]);

endmodule
```

(Kaushik Patra, Lab 13, Slide 5)

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

// TBD
nor nor0(Y[0], A[0], B[0]);
nor nor1(Y[1], A[1], B[1]);
nor nor2(Y[2], A[2], B[2]);
nor nor3(Y[3], A[3], B[3]);
nor nor4(Y[4], A[4], B[4]);
nor nor5(Y[5], A[5], B[5]);
nor nor6(Y[6], A[6], B[6]);
nor nor7(Y[7], A[7], B[7]);
nor nor8(Y[8], A[8], B[8]);
nor nor9(Y[9], A[9], B[9]);
nor nor10(Y[10], A[10], B[10]);
nor nor11(Y[11], A[11], B[11]);
nor nor12(Y[12], A[12], B[12]);
nor nor13(Y[13], A[13], B[13]);
nor nor14(Y[14], A[14], B[14]);
nor nor15(Y[15], A[15], B[15]);
nor nor16(Y[16], A[16], B[16]);
nor nor17(Y[17], A[17], B[17]);
nor nor18(Y[18], A[18], B[18]);
nor nor19(Y[19], A[19], B[19]);
nor nor20(Y[20], A[20], B[20]);
nor nor21(Y[21], A[21], B[21]);
nor nor22(Y[22], A[22], B[22]);
nor nor23(Y[23], A[23], B[23]);
nor nor24(Y[24], A[24], B[24]);
nor nor25(Y[25], A[25], B[25]);
nor nor26(Y[26], A[26], B[26]);
nor nor27(Y[27], A[27], B[27]);
nor nor28(Y[28], A[28], B[28]);
nor nor29(Y[29], A[29], B[29]);
nor nor30(Y[30], A[30], B[30]);
nor nor31(Y[31], A[31], B[31]);

endmodule
```

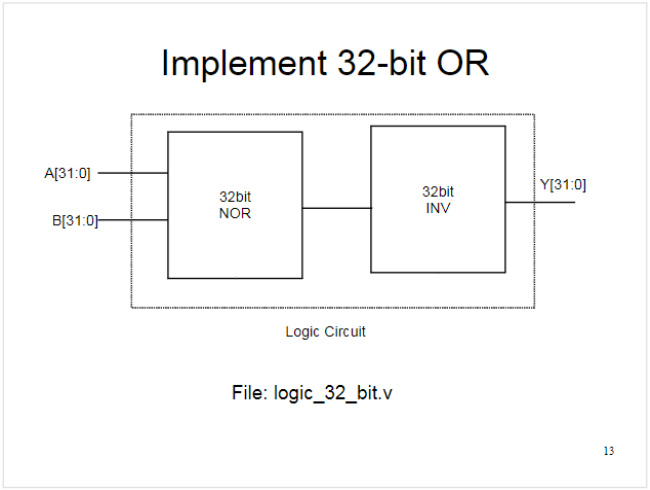
(The testing is short so I will do it all together once again.)





(Kaushik Patra, Lab 13, Slide 6)

```
// 32-bit inverter
module INV32_1x1(Y,A);
//output
output [31:0] Y;
//input
input [31:0] A;

// TBD
assign Y = ~A;

endmodule
```



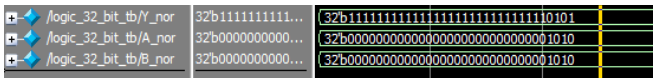
(Kaushik Patra, Lab 13, Slide 7)

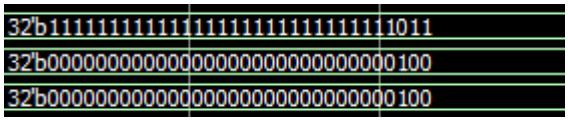
```
// 32-bit OR
module OR32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

wire [31:0] tempY;

NOR32_2x1 nor_inst(.Y(tempY),.A(A),.B(B));
INV32_1x1 inv(.Y(Y), .A(tempY));

endmodule
```

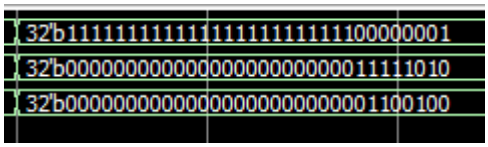
Testing of 32 bit logic gates:

```
`timescale 1ns/1ps

module logic_32_bit_tb;
wire [31:0] Y_nor;
wire [31:0] Y_and;
wire [31:0] Y_inv;
wire [31:0] Y_or;

reg [31:0] A_nor;
reg [31:0] A_and;
reg [31:0] A_inv;
reg [31:0] A_or;

reg [31:0] B_nor;
reg [31:0] B_and;
reg [31:0] B_or;

//module NOR32_2x1(Y,A,B);
//module AND32_2x1(Y,A,B);
//module INV32_1x1(Y,A);
//module OR32_2x1(Y,A,B);
NOR32_2x1 nor_inst(.Y(Y_nor), .A(A_nor), .B(B_nor));
AND32_2x1 and_inst(.Y(Y_and), .A(A_and), .B(B_and));
INV32_1x1 inv_inst(.Y(Y_inv), .A(A_inv));
OR32_2x1  or_inst(.Y(Y_or), .A(A_or), .B(B_or));

initial
begin
A_nor=10; B_nor=10; A_and=10; B_and=10; A_inv=10; A_or=10; B_or=10;
#15 A_nor=4; B_nor=4; A_and=4; B_and=4; A_inv=4;  A_or=4; B_or=4;
#15 A_nor=250; B_nor=100; A_and=1; B_and=10; A_inv=0; A_or=100; B_or=10;
#15 ;
end

endmodule
```

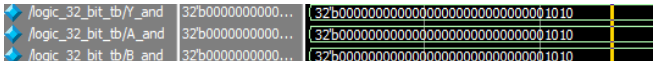
NOR:

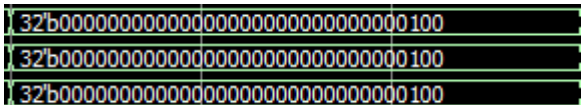


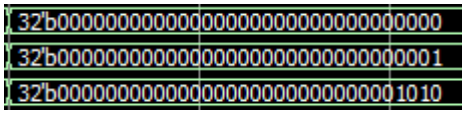




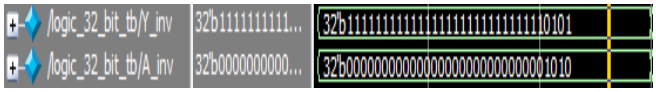
AND:

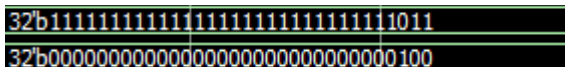


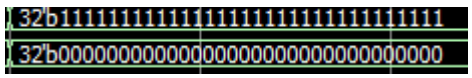




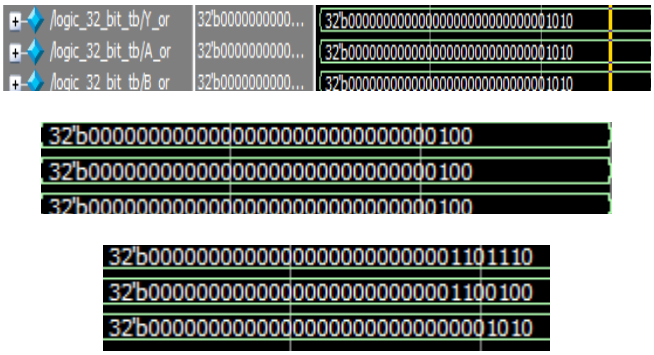
INV:





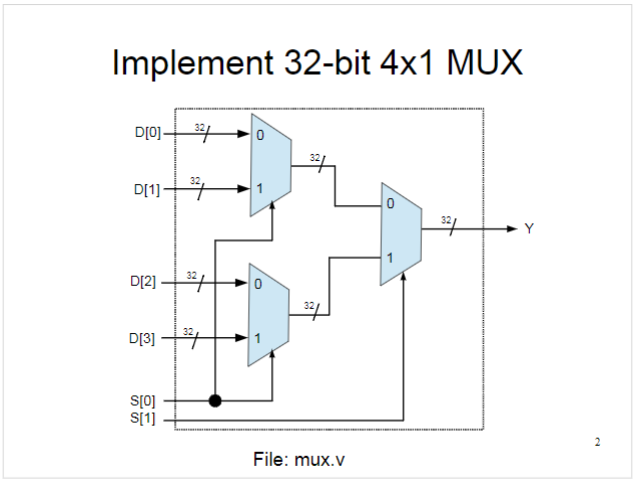


OR:



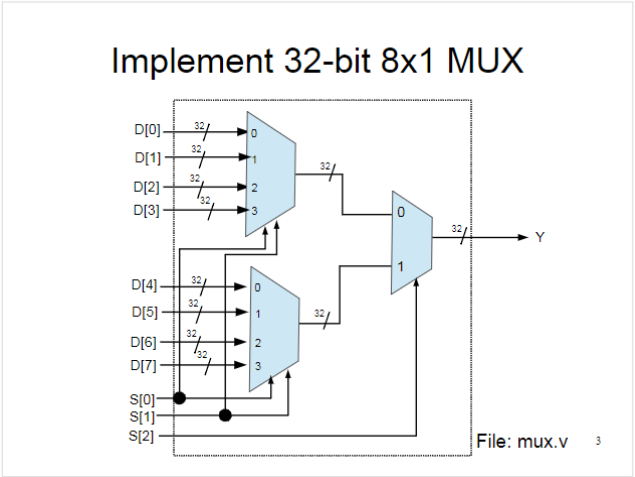
In each of these cases, you can see that the proper bitwise operations are correct/resulting in the correct values.

E. Multiplexers (forming a 16x1 mux)



(Kaushik Patra, Lab 14, Slide 1)

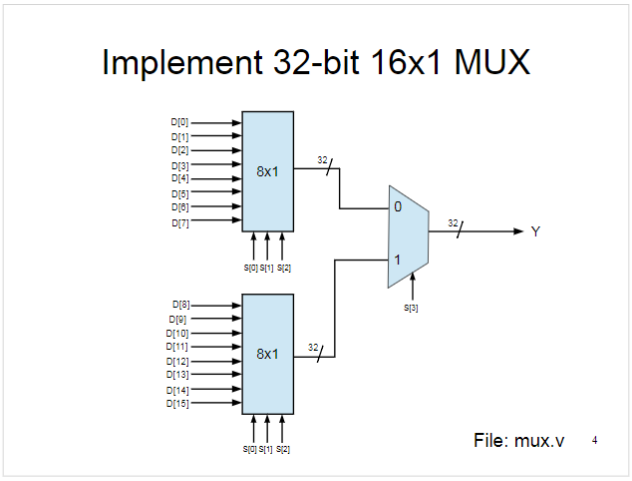
Each subsequent mux will rely on previous ones so they are easily instantiated. You should test them as you go but I will show the testing after each's implementation.




```
// 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_Y;
wire [31:0] mux2_Y;

// TBD
//module MUX32_2x1(Y, I0, I1, S);
MUX32_2x1 mux1(.Y(mux1_Y), .I0(I0), .I1(I1), .S(S[0]));
MUX32_2x1 mux2(.Y(mux2_Y), .I0(I2), .I1(I3), .S(S[0]));

MUX32_2x1 mux3(.Y(Y), .I0(mux1_Y), .I1(mux2_Y), .S(S[1]));

endmodule
```

(Kaushik Patra, Lab 14, Slide 2)

```
// 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;

// TBD
wire [31:0] mux1_Y;
wire [31:0] mux2_Y;

//module MUX32_4x1(Y, I0, I1, I2, I3, S);

MUX32_4x1 mux1(.Y(mux1_Y), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .S(S[1:0]));
MUX32_4x1 mux2(.Y(mux2_Y), .I0(I4), .I1(I5), .I2(I6), .I3(I7), .S(S[1:0]));

MUX32_2x1 mux3(.Y(Y), .I0(mux1_Y), .I1(mux2_Y), .S(S[2]));

endmodule
```



(Kaushik Patra, Lab 14, Slide 2)

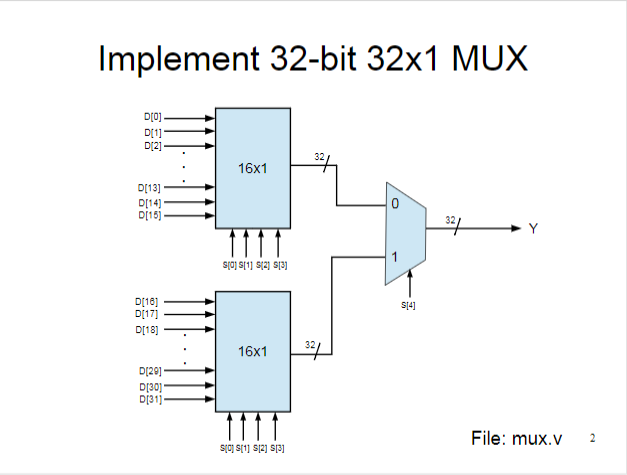
```
// 32-bit 16x1 mux
module MUX32_16x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                  I8, I9, I10, I11, I12, I13, I14, I15, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [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;
```

```
// TBD
wire [31:0] mux1_Y;
wire [31:0] mux2_Y;

//module MUX32_8x1(Y, I0, I1, I2, I3, I4, I5, I6, I7, S);
MUX32_8x1 mux1(.Y(mux1_Y), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .I4(I4), .I5(I5), .I6(I6), .I7(I7), .S(S[2:0]));
MUX32_8x1 mux2(.Y(mux2_Y), .I0(I8), .I1(I9), .I2(I10), .I3(I11), .I4(I12), .I5(I13), .I6(I14), .I7(I15), .S(S[2:0]));

MUX32_2x1 mux3(.Y(Y), .I0(mux1_Y), .I1(mux2_Y), .S(S[3]));

endmodule
```

Plain text is as follows, as it is hard to read:

```
// 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;

// TBD
wire [31:0] mux1_Y;
wire [31:0] mux2_Y;

//module MUX32_8x1(Y, I0, I1, I2, I3, I4, I5, I6, I7, S);
MUX32_8x1 mux1(.Y(mux1_Y), .I0(I0), .I1(I1), .I2(I2),
.I3(I3), .I4(I4), .I5(I5), .I6(I6), .I7(I7), .S(S[2:0]));

MUX32_8x1 mux2(.Y(mux2_Y), .I0(I8), .I1(I9), .I2(I10),
.I3(I11), .I4(I12), .I5(I13), .I6(I14), .I7(I15), .S(S[2:0]));

MUX32_2x1 mux3(.Y(Y), .I0(mux1_Y), .I1(mux2_Y),
.S(S[3]));

endmodule
```

The 32 bit 32x1 isn't used until later, but we will be using it for testing so I will show it now.



(Kaushik Patra, Lab 16, Slide 1)

```
// 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;

// TBD
wire [31:0] mux1_res;
wire [31:0] mux2_res;
MUX32_16x1 mux_16_1(.Y(mux1_res), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .I4(I4), .I5(I5),
                    .I6(I6), .I7(I7), .I8(I8), .I9(I9), .I10(I10), .I11(I11),
                    .I12(I12), .I13(I13), .I14(I14), .I15(I15), .S(S[3:0]));
MUX32_16x1 mux_16_2(.Y(mux2_res), .I0(I16), .I1(I17), .I2(I18), .I3(I19), .I4(I20), .I5(I21),
                    .I6(I22), .I7(I23), .I8(I24), .I9(I25), .I10(I26), .I11(I27),
                    .I12(I28), .I13(I29), .I14(I30), .I15(I31), .S(S[3:0]));
MUX32_2x1 mux3(.Y(Y), .I0(mux1_res), .I1(mux2_res), .S(S[4]));

endmodule
```

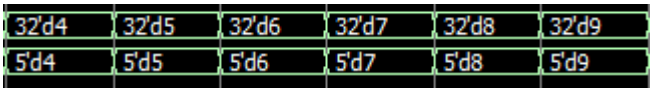
Testing:

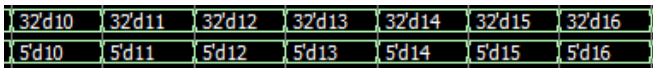
Essentially, you need to be able to put something in every input and then read from those inputs. I would test every multiplexer but since they are telescoping in nature (each one refers to an instance of previous ones,) I can simply test the highest “order” or mux. If you encounter problems you should test them one by one, but I will only show testing of the 32 bit 16x1 mux.

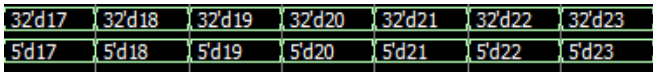
```
`timescale 1ns/1ps
`include "prj_definition.v"

module mux_tb;
//reg SnA;
//wire CO;//, S;
//wire [31:0] Y;
wire Y;
wire [31:0] Y_32;
wire [31:0] Y_16x1;

reg [31:0] I0_32;
reg [31:0] I1_32;
reg S;
reg I0, I1;

reg [31:0] I0_16x1;
reg [31:0] I1_16x1;
reg [31:0] I2_16x1;
reg [31:0] I3_16x1;
reg [31:0] I4_16x1;
reg [31:0] I5_16x1;
reg [31:0] I6_16x1;
reg [31:0] I7_16x1;
reg [31:0] I8_16x1;
reg [31:0] I9_16x1;
reg [31:0] I10_16x1;
reg [31:0] I11_16x1;
reg [31:0] I12_16x1;
reg [31:0] I13_16x1;
reg [31:0] I14_16x1;
reg [31:0] I15_16x1;

reg [31:0] I16_16x1;
reg [31:0] I17_16x1;
reg [31:0] I18_16x1;
reg [31:0] I19_16x1;
reg [31:0] I20_16x1;
reg [31:0] I21_16x1;
reg [31:0] I22_16x1;
reg [31:0] I23_16x1;
reg [31:0] I24_16x1;
reg [31:0] I25_16x1;
reg [31:0] I26_16x1;
reg [31:0] I27_16x1;
reg [31:0] I28_16x1;
reg [31:0] I29_16x1;
reg [31:0] I30_16x1;
reg [31:0] I31_16x1;

reg [3:0] S_16x1;
reg [4:0] S_32x1;

// module MUX32_2x1(Y, I0, I1, S);
// module MUX1_2x1(Y,I0, I1, S);
MUX1_2x1 mux1(.Y(Y), .I0(I0), .I1(I1), .S(S));
MUX32_2x1 mux2(.Y(Y_32), .I0(I0_32), .I1(I1_32), .S(S));
//MUX32_16x1 mux_16(.Y(Y_16x1), .I0(I0_16x1),
.I1(I1_16x1), .I2(I2_16x1), .I3(I3_16x1), .I4(I4_16x1),
.I5(I5_16x1),
//            .I6(I6_16x1), .I7(I7_16x1), .I8(I8_16x1),
.I9(I9_16x1), .I10(I10_16x1), .I11(I11_16x1),
//            .I12(I12_16x1), .I13(I13_16x1),
.I14(I14_16x1), .I15(I15_16x1), .S(S_16x1));

MUX32_32x1 mux_32(.Y(Y_16x1), .I0(I0_16x1),
.I1(I1_16x1), .I2(I2_16x1), .I3(I3_16x1), .I4(I4_16x1),
.I5(I5_16x1), .I6(I6_16x1), .I7(I7_16x1),
            .I8(I8_16x1), .I9(I9_16x1), .I10(I10_16x1),
.I11(I11_16x1), .I12(I12_16x1), .I13(I13_16x1),
.I14(I14_16x1), .I15(I15_16x1),
            .I16(I16_16x1), .I17(I17_16x1),
.I18(I18_16x1), .I19(I19_16x1), .I20(I20_16x1),
.I21(I21_16x1), .I22(I22_16x1), .I23(I23_16x1),
            .I24(I24_16x1), .I25(I25_16x1),
.I26(I26_16x1), .I27(I27_16x1), .I28(I28_16x1),
.I29(I29_16x1), .I30(I30_16x1), .I31(I31_16x1),
.S(S_32x1));

 initial
 begin

 I0_16x1=0;
 I1_16x1=1;
 I2_16x1=2;
 I3_16x1=3;
 I4_16x1=4;
 I5_16x1=5;
 I6_16x1=6;
 I7_16x1=7;
 I8_16x1=8;
 I9_16x1=9;
 I10_16x1=10;
 I11_16x1=11;
 I12_16x1=12;
 I13_16x1=13;
 I14_16x1=14;
 I15_16x1=15;
 I16_16x1=16;
 I17_16x1=17;
 I18_16x1=18;
 I19_16x1=19;
 I20_16x1=20;
 I21_16x1=21;
 I22_16x1=22;
 I23_16x1=23;
 I24_16x1=24;
 I25_16x1=25;
 I26_16x1=26;
 I27_16x1=27;
 I28_16x1=28;
 I29_16x1=29;
 I30_16x1=30;
 I31_16x1=31;
```

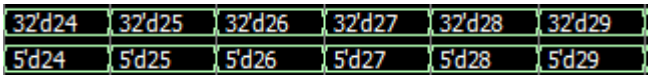
```
#5 S_32x1 = 0 ;
#5 S_32x1 = 1 ;
#5 S_32x1 = 2 ;
#5 S_32x1 = 3 ;
#5 S_32x1 = 4 ;
#5 S_32x1 = 5 ;
#5 S_32x1 = 6 ;
#5 S_32x1 = 7 ;
#5 S_32x1 = 8 ;
#5 S_32x1 = 9 ;
#5 S_32x1 = 10 ;
#5 S_32x1 = 11 ;
#5 S_32x1 = 12 ;
#5 S_32x1 = 13 ;
#5 S_32x1 = 14 ;
#5 S_32x1 = 15 ;
#5 S_32x1 = 16 ;
#5 S_32x1 = 17 ;
#5 S_32x1 = 18 ;
#5 S_32x1 = 19 ;
#5 S_32x1 = 20 ;
#5 S_32x1 = 21 ;
#5 S_32x1 = 22 ;
#5 S_32x1 = 23 ;
#5 S_32x1 = 24 ;
#5 S_32x1 = 25 ;
#5 S_32x1 = 26 ;
#5 S_32x1 = 27 ;
#5 S_32x1 = 28 ;
#5 S_32x1 = 29 ;
#5 S_32x1 = 30 ;
#5 S_32x1 = 31 ;
end
```

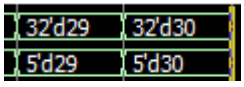
Since each input simply holds the value of its index and we're incrementing by one each time, I am omitting the input registers from the waveform.

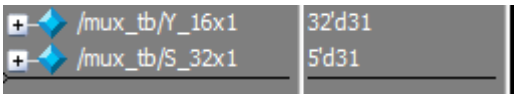






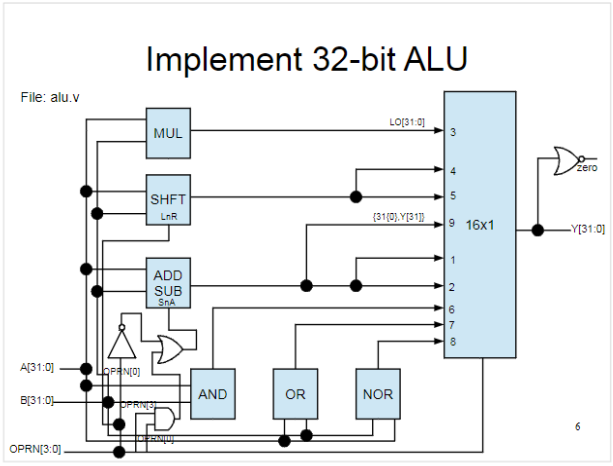






The timeline is on the last test case and you can see that the value of 31 (32nd input) exists.

F. *Assembling the ALU*



(Kaushik Patra, Lab 14, Slide 3)

```
`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 [`ALU_OPRN_INDEX_LIMIT:0] OPRN; // operation code

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

wire [31:0] nor_res;
wire [31:0] or_res;
wire [31:0] and_res;
wire [31:0] add_sub_res;
wire [31:0] shifter_res;
wire [31:0] mul_res;
wire [31:0] mul_high;
wire [31:0] add_sub_2;
wire CO;

wire [31:0] n; //place holder, n for nothing

//making SnA
wire one_bit_and_res;
wire inv_res;
wire SnA;
```

```
wire nor_res_0;
wire nor_res_1;
wire nor_res_2;
wire nor_res_3;
wire nor_res_4;
wire nor_res_5;
wire nor_res_6;
wire nor_res_7;
wire nor_res_8;
wire nor_res_9;
wire nor_res_10;
wire nor_res_11;
wire nor_res_12;
wire nor_res_13;
wire nor_res_14;
wire nor_res_15;
wire nor_res_16;
wire nor_res_17;
wire nor_res_18;
wire nor_res_19;
wire nor_res_20;
wire nor_res_21;
wire nor_res_22;
wire nor_res_23;
wire nor_res_24;
wire nor_res_25;
wire nor_res_26;
wire nor_res_27;
wire nor_res_28;
wire nor_res_29;
wire nor_res_30;
```

Making SnA:

```
and and_1(one_bit_and_res, OPRN[3], OPRN[0]);
assign inv_res = ~OPRN[0];
or or_1(SnA, one_bit_and_res, inv_res);
```

```
MULT32 mult(.HI(mul_high), .LO(mul_res), .A(OP1), .B(OP2));
SHIFT32 shifter(.Y(shifter_res), .D(OP1), .S(OP2), .LnR(OPRN[0]));
RC_ADD_SUB_32 add_sub(.Y(add_sub_res), .CO(CO), .A(OP1), .B(OP2), .SnA(SnA));

NOR32_2x1 nor_inst(.Y(nor_res), .A(OP1), .B(OP2));
AND32_2x1 and_inst(.Y(and_res), .A(OP1), .B(OP2));
OR32_2x1  or_inst(.Y(or_res), .A(OP1), .B(OP2));
```

```
assign add_sub_2 = {{31{1'b0}},add_sub_res[31]};
```

(plain text for mux):

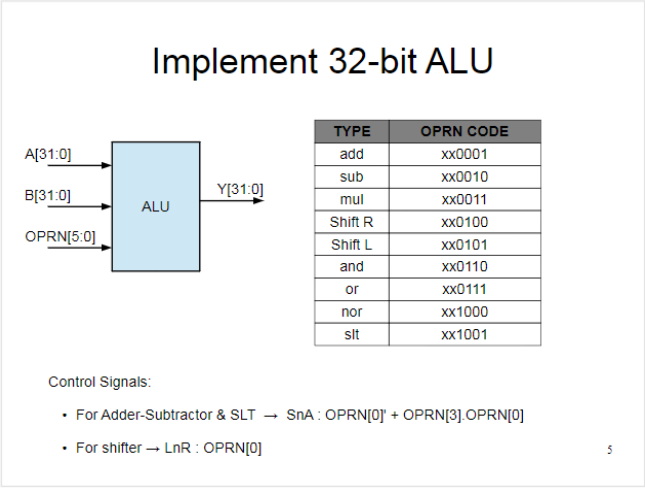
MUX32_16x1 mux(.Y(OUT), .I0(n), .I1(add_sub_res), .I2(add_sub_res), .I3(mul_res), .I4(shifter_res), .I5(shifter_res), .I6(and_res), .I7(or_res), .I8(nor_res), .I9(add_sub_2), .I10(n), .I11(n), .I12(n), .I13(n), .I14(n), .I15(n), .S(OPRN[3:0]));

Turning result into 1 bit and checking if any bit is 1. If all bits are 0 then set ZERO to 1.

```
//nor nor_1(nor_res_1, nor_res_0, OUT[2]);
//entire lower block was nors before
or nor0(nor_res_0, OUT[0], OUT[1]);
or nor1(nor_res_1, nor_res_0, OUT[2]);
or nor2(nor_res_2, nor_res_1, OUT[3]);
or nor3(nor_res_3, nor_res_2, OUT[4]);
or nor4(nor_res_4, nor_res_3, OUT[5]);
or nor5(nor_res_5, nor_res_4, OUT[6]);
or nor6(nor_res_6, nor_res_5, OUT[7]);
or nor7(nor_res_7, nor_res_6, OUT[8]);
or nor8(nor_res_8, nor_res_7, OUT[9]);
or nor9(nor_res_9, nor_res_8, OUT[10]);
or nor10(nor_res_10, nor_res_9, OUT[11]);
or nor11(nor_res_11, nor_res_10, OUT[12]);
or nor12(nor_res_12, nor_res_11, OUT[13]);
or nor13(nor_res_13, nor_res_12, OUT[14]);
or nor14(nor_res_14, nor_res_13, OUT[15]);
or nor15(nor_res_15, nor_res_14, OUT[16]);
or nor16(nor_res_16, nor_res_15, OUT[17]);
or nor17(nor_res_17, nor_res_16, OUT[18]);
or nor18(nor_res_18, nor_res_17, OUT[19]);
or nor19(nor_res_19, nor_res_18, OUT[20]);
or nor20(nor_res_20, nor_res_19, OUT[21]);
or nor21(nor_res_21, nor_res_20, OUT[22]);
or nor22(nor_res_22, nor_res_21, OUT[23]);
or nor23(nor_res_23, nor_res_22, OUT[24]);
or nor24(nor_res_24, nor_res_23, OUT[25]);
or nor25(nor_res_25, nor_res_24, OUT[26]);
or nor26(nor_res_26, nor_res_25, OUT[27]);
or nor27(nor_res_27, nor_res_26, OUT[28]);
or nor28(nor_res_28, nor_res_27, OUT[29]);
or nor29(nor_res_29, nor_res_28, OUT[30]);
or nor30(nor_res_30, nor_res_29, OUT[31]);

assign ZERO = ~nor_res_30; // just not the result bit
endmodule
```

Testing:

You need to check that every operation works properly.

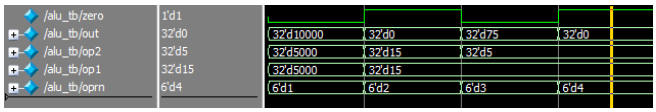


(Kaushik Patra, Lab 14, Slide 3)

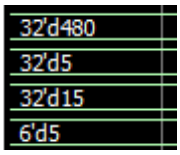
Simply drive OPRN with the corresponding opcodes and verify that A(operation)B is correct.

```
`timescale 1ns/1ps
`include "prj_definition.v"

module alu_tb;
wire zero;
wire [31:0] out;

reg [31:0] op2;
reg [31:0] op1;
reg [`ALU_OPRN_INDEX_LIMIT:0] oprn;

// module ALU(OUT, ZERO, OP1, OP2, OPRN);
ALU alu(.OUT(out), .ZERO(zero), .OP1(op1), .OP2(op2), .OPRN(oprn));

initial
begin

op1 = 5000; op2 = 5000; oprn = 1;    // add
#5 op1 = 15; op2 = 15; oprn = 2; // sub
#5 op1 = 15; op2 = 5; oprn = 3; // mul
#5 op1 = 15; op2 = 5; oprn = 4; // shift R
#5 op1 = 15; op2 = 5; oprn = 5; // shift L
#5 op1 = 15; op2 = 5; oprn = 6; // and
#5 op1 = 15; op2 = 5; oprn = 7; // or
#5 op1 = 15; op2 = 5; oprn = 8; // nor
#5 op1 = 15; op2 = 5; oprn = 9; // slt
#5 ;
end

endmodule
```

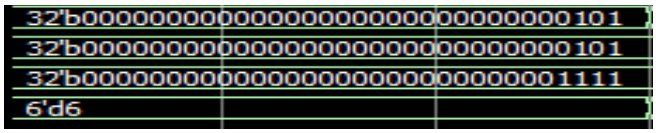
Test cases:
5000+5000=10000
15-15= 0 (ZERO == 1)
15x5= 75
15>>5= 0 (ZERO == 1)
15<<5= 480
(and, or, and nor are better seen in the waveform)
15<5=0



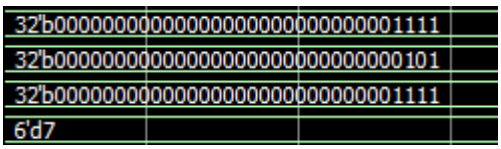
shift left:



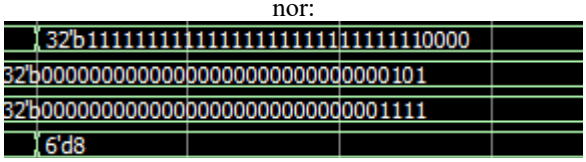
and:



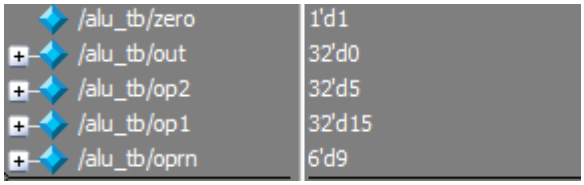
or:



nor:

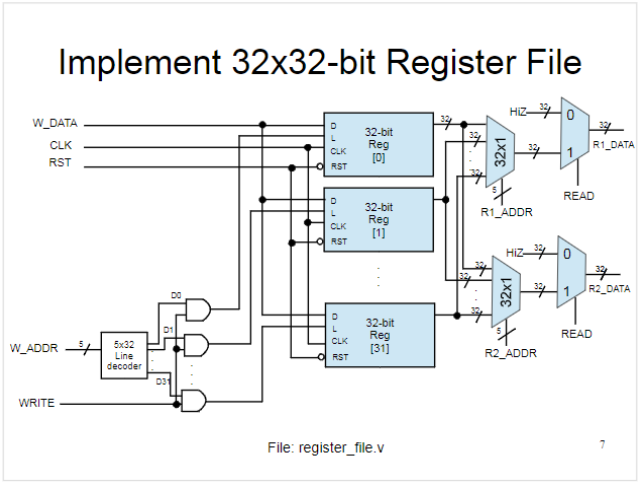


slt:



With the ALU working, we now move on to the register file.

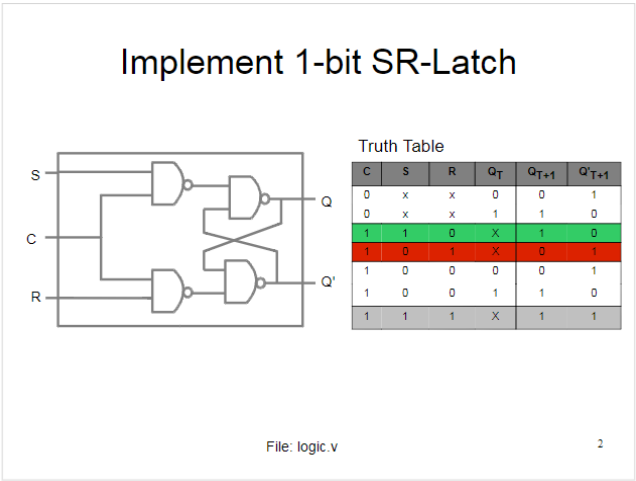
II. Register File Assembly and Testing



(Kaushik Patra, Lab 16, Slide 4)

The register file will act as the temporary storage used for operations and other intermediary storage/processes. We need a 5x32 line decoder (this is similar to the multiplexers in the telescoping nature so it can also be tested at the topmost level to test all of them. You, of course, may want to test every step of the way) and 32bit registers. At this point the 32x1 muxs would be new/should be done but I have already covered them in the ALU's mux section.

A. Registers



(Kaushik Patra, Lab 15, Slide 1)

```
// 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;

// TBD
wire SRes;
wire RRes;

nand nand1(SRes, S, C);
nand nand2(RRes, C, R);

nand nand3(Q, Qbar, SRes);
nand nand4(Qbar, Q, RRes);

endmodule
```

Testing: (I will not be doing it all at once as there are many different things with different purposes within the test file)

```
`timescale 1ns/1ps
`include "prj_definition.v"

module logic_tb;
reg S, R, C;
reg nP, nR;
wire Q,Qbar;
wire D_Latch_Q, D_Latch_Qbar;
reg D_Latch_D, D_Latch_C;
wire FF_Q, FF_Qbar;
reg FF_D, FF_C, FF_nP, FF_nR;
reg reg_L;
wire reg_Q;

wire [31:0] D;
reg [4:0] I;
//wire [3:0] D;
//reg [1:0] I;

wire [31:0] Q_32;

reg CLK, LOAD;
reg [31:0] D_32;
reg RESET;
```

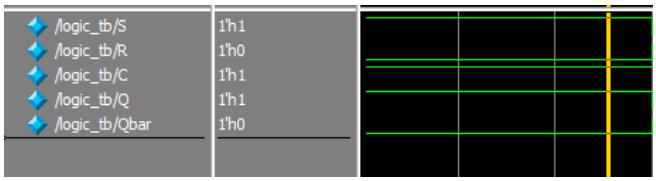
```
//module REG32(Q, D, LOAD, CLK, RESET);
//module SR_LATCH(Q, Qbar, S, R, C, nP, nR);
//module D_LATCH(Q, Qbar, D, C, nP, nR);
//module D_FF(Q, Qbar, D, C, nP, nR);
//module REG1(Q, Qbar, D, L, C, nP, nR);
SR_LATCH sr_latch(.Q(Q), .Qbar(Qbar), .S(S), .R(R), .C(C), .nP(nP), .nR(nR));
D_LATCH d_latch(.Q(D_Latch_Q), .Qbar(D_Latch_Qbar), .D(D_Latch_D), .C(D_Latch_C), .nP(nP), .nR(nR));
D_FF flip_flop(.Q(FF_Q), .Qbar(FF_Qbar), .D(FF_D), .C(FF_C), .nP(FF_nP), .nR(FF_nR));

DECODER_5x32 decoder_5x32(.D(D),.I(I));

//DECODER_2x4 D_2x4(.D(D[3:0]), .I(I[1:0]));
//REG1 single_reg(.Q(reg_Q), .Qbar(FF_Qbar), .D(FF_D), .L(reg_L), .C(FF_C), .nP(FF_nP), .nR(FF_nR));
//REG32 reg_32(.Q(Q_32), .D(D_32), .LOAD(LOAD), .CLK(CLK), .RESET(RESET));
```

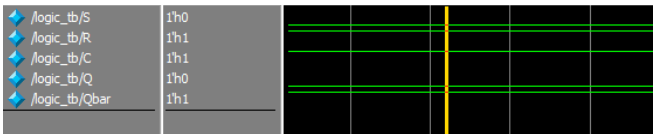
Things are commented out/in as needed.

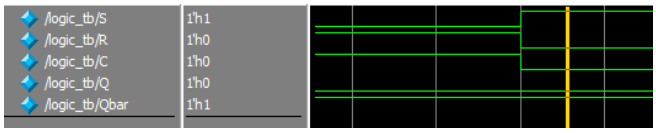
```
#5 S=1; C=1; R=0; // D
#5 S=0; C=1; R=1; //D
#5 S=1; C=0; R=0; //D
//#5 S=0; C=1; R=0;
#5 ;//S=0; C=1; R=0;
```

I did not bother with implementing nP or nR for this module, I simply handled that within the flip flop.

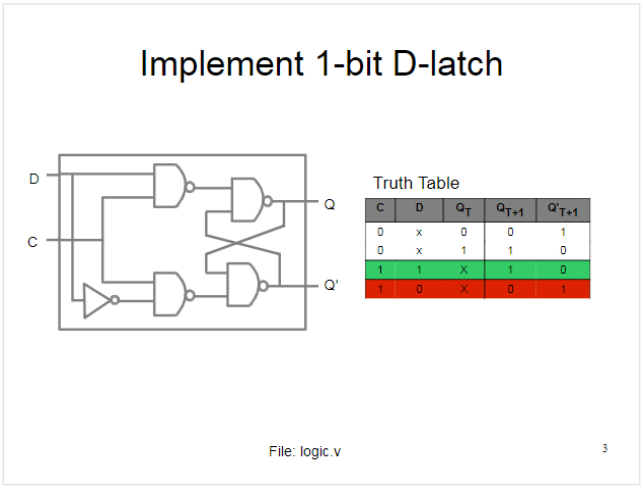
Waveform:







All we have to test is if S will set Q to 1 when C is 1 as well as test if Q will equal zero when C == 1, S==0, and R ==1. Every other condition is a race condition or we don't care about it.



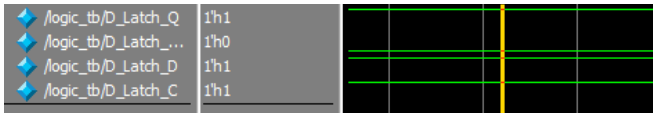
(Kaushik Patra, Lab 15, Slide 2)

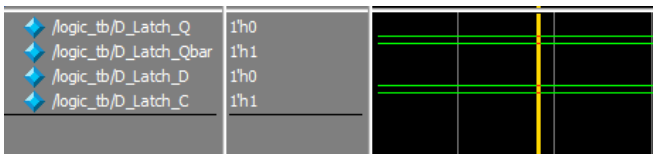
```
// 1 bit D latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module D_LATCH(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

// TBD
wire DRes;
wire RRes;
wire R;
assign R = ~D;

nand nand1(DRes, D, C);
nand nand2(RRes, C, R);

nand nand3(Q, Qbar, DRes);
nand nand4(Qbar, Q, RRes);

endmodule
```

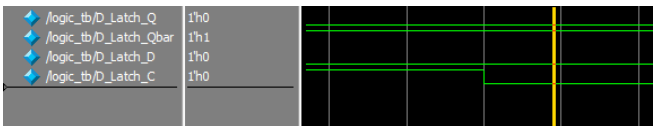
The bottom of the schematic will be the “R” side as in the flip-flop and SR latch.
(Slight modification to test was made, changes are as such:)

```
/*
#5 S=1; C=1; R=0;
#5 S=0; C=1; R=1;
#5 S=1; C=0; R=0; */

#5  D_Latch_C=1; D_Latch_D=1;
#5  D_Latch_C=1; D_Latch_D=0;
#5  D_Latch_C=0; D_Latch_D=0;

//#5 FF_D=1; FF_C=1; FF_nR=0; FF_nP=0; reg_L =1;
//#5 FF_D=0; FF_C=1; FF_nR=0; FF_nP=1; reg_L=1;
//#5 FF_D=1; FF_C=0; FF_nR=1; FF_nP=1; reg_L =0;
//#5 S=0; C=1; R=0;
#5 ;//S=0; C=1; R=0;
```

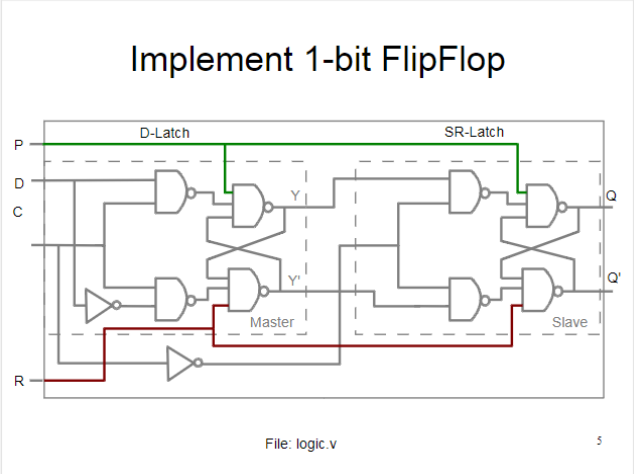
Test cases shown above
Waveform:







The only cases we need this time are:
C==1, D==1, Q ==1
and
C==1, D==0, Q==0



(Kaushik Patra, Lab 15, Slide 3)

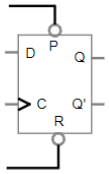
((You should actually properly use the SR-Latch and D-Latch that were made by taking advantage of the extra parameters nP and nR and a 3 input nand. I wasn't aware of the three input nand and didn't notice the extra parameters. So, I re-implemented the SR-Latch and D-Latch in my Flip-Flop. Regardless, my instantiation works.))

```
// 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;

// TBD
wire DRes;
wire CRes;
wire PRes;
wire RRes;
wire Dnot;
wire Cnot;
wire D_Latch_Q, D_Latch_QBar;
assign Dnot = ~D;
assign Cnot = ~C;
```

```
wire SR_DRes, SR_CRes;
wire SR_PRes, SR_RRes;

nand nand1(DRes, D, C);
nand nand2(CRes, C, Dnot);

and and1(PRes, DRes, nP);
nand nand3(D_Latch_Q, D_Latch_Qbar, PRes);
and and2(RRes, nR, CRes);
nand nand4(D_Latch_Qbar, D_Latch_Q, RRes);

nand nand5(SR_DRes, D_Latch_Q, Cnot);
nand nand6(SR_CRes, D_Latch_Qbar, Cnot);

and and3(SR_PRes, SR_DRes, nP);
nand nand7(Q, SR_PRes, Qbar);
and and4(SR_RRes, SR_CRes, nR);
nand nand8(Qbar, SR_RRes, Q);


endmodule
```

Testing:
Truth table:

Implement 1-bit FlipFlop

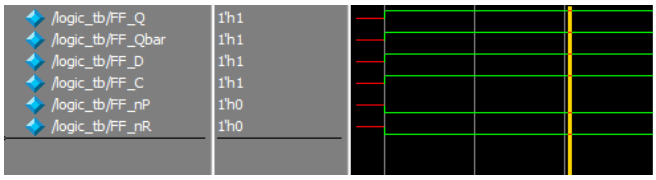


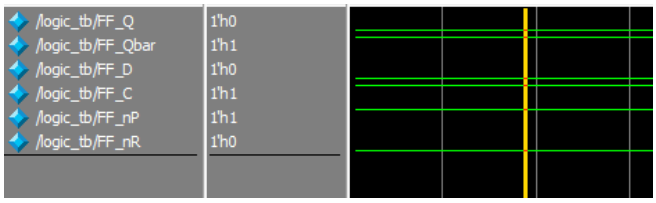
C	D	P	R	Q_t	Q_{t+1}
x	x	0	0	x	?
x	x	0	1	x	1
x	x	1	0	x	0
0	x	1	1	0	0
0	x	1	1	1	1
1	0	1	1	x	0
1	1	1	1	x	1

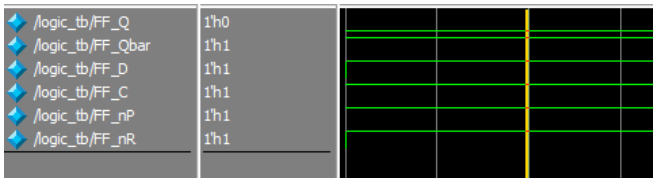
File: logic.v 4

(Kaushik Patra, Lab 15, Slide 2)

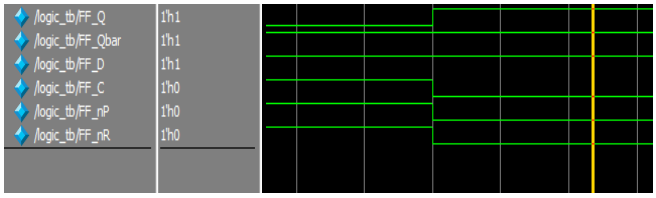
```
#5 FF_D=1; FF_C=1; FF_nR=0; FF_nP=0; reg_L =1;
#5 FF_D=0; FF_C=1; FF_nR=0; FF_nP=1; reg_L =1;
#5 FF_D=1; FF_C=1; FF_nR=1; FF_nP=1; reg_L =0;
#5 FF_D=1; FF_C=0; FF_nR=0; FF_nP=0; reg_L =1;

#5 ;
```





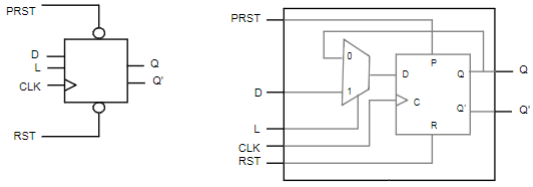


(Race condition)



1 bit Register:

Implement 1-bit Register



File: logic.v 6

(Kaushik Patra, Lab 15, Slide 3)

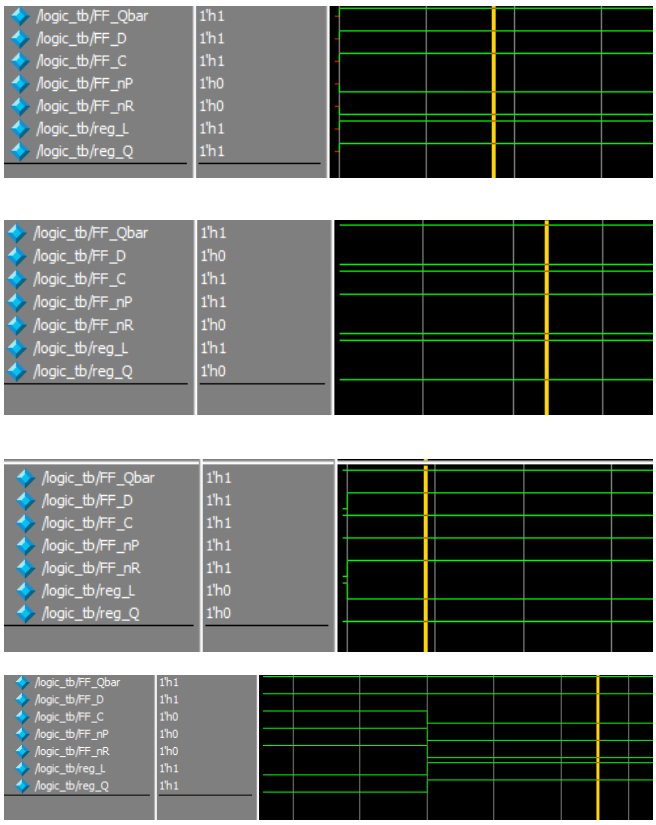
```
// 1 bit register +ve edge,
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module REG1(Q, Qbar, D, L, C, nP, nR);
input D, C, L;
input nP, nR;
output Q,Qbar;

// TBD
wire selectedD;

MUX1_2x1 mux(.Y(selectedD), .I0(Q), .I1(D), .S(L));
D_FF flip_flop(.Q(Q), .Qbar(Qbar), .D(selectedD), .C(C), .nP(nP), .nR(nR));

endmodule
```

```
//DECODER_2x4 D_2x4(.D(D[3:0]), .I(I[1:0]));
REG1 single_reg(.Q(reg_Q), .Qbar(FF_Qbar), .D(FF_D), .L(reg_L), .C(FF_C), .nP(FF_nP), .nR(FF_nR));
//REG32 reg_32(.Q(Q_32), .D(D_32), .LOAD(LOAD), .CLK(CLK), .RESET(RESET));
```

REG1 single_reg(.Q(reg_Q), .Qbar(FF_Qbar), .D(FF_D), .L(reg_L), .C(FF_C), .nP(FF_nP), .nR(FF_nR));

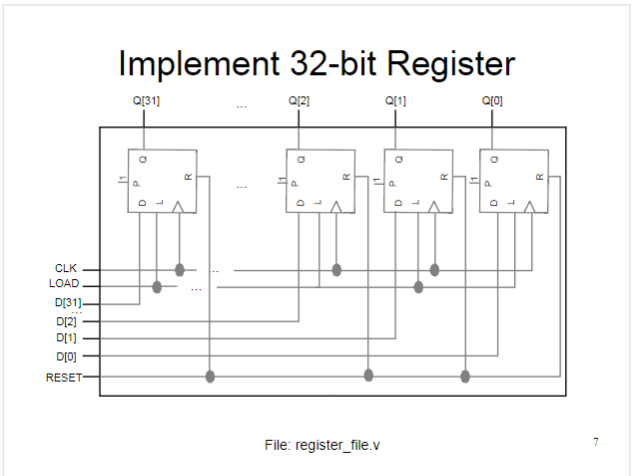
```
#5 FF_D=1; FF_C=1; FF_nR=0; FF_nP=0; reg_L =1;
#5 FF_D=0; FF_C=1; FF_nR=0; FF_nP=1; reg_L =1;
#5 FF_D=1; FF_C=1; FF_nR=1; FF_nP=1; reg_L =0;
#5 FF_D=1; FF_C=0; FF_nR=0; FF_nP=0; reg_L =1;

#5 ;
```

Same test cases since the only change is a mux was added to make a register.



Essentially, so far, we have just been constructing the ability to store one bit of data and output it (as Q). Now we can create 32 registers in series to form a 32 bit register

32-bit Register:



(Kaushik Patra, Lab 14, Slide 4)

```
// 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;

// TBD
wire one;
wire [31:0] Qbar;
assign one = 1'b1;
```

```
REG1 reg0(.Q(Q[0]), .Qbar(Qbar[0]), .D(D[0]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg1(.Q(Q[1]), .Qbar(Qbar[1]), .D(D[1]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg2(.Q(Q[2]), .Qbar(Qbar[2]), .D(D[2]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg3(.Q(Q[3]), .Qbar(Qbar[3]), .D(D[3]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg4(.Q(Q[4]), .Qbar(Qbar[4]), .D(D[4]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg5(.Q(Q[5]), .Qbar(Qbar[5]), .D(D[5]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg6(.Q(Q[6]), .Qbar(Qbar[6]), .D(D[6]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg7(.Q(Q[7]), .Qbar(Qbar[7]), .D(D[7]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg8(.Q(Q[8]), .Qbar(Qbar[8]), .D(D[8]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg9(.Q(Q[9]), .Qbar(Qbar[9]), .D(D[9]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg10(.Q(Q[10]), .Qbar(Qbar[10]), .D(D[10]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg11(.Q(Q[11]), .Qbar(Qbar[11]), .D(D[11]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg12(.Q(Q[12]), .Qbar(Qbar[12]), .D(D[12]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg13(.Q(Q[13]), .Qbar(Qbar[13]), .D(D[13]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg14(.Q(Q[14]), .Qbar(Qbar[14]), .D(D[14]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg15(.Q(Q[15]), .Qbar(Qbar[15]), .D(D[15]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg16(.Q(Q[16]), .Qbar(Qbar[16]), .D(D[16]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg17(.Q(Q[17]), .Qbar(Qbar[17]), .D(D[17]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg18(.Q(Q[18]), .Qbar(Qbar[18]), .D(D[18]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg19(.Q(Q[19]), .Qbar(Qbar[19]), .D(D[19]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg20(.Q(Q[20]), .Qbar(Qbar[20]), .D(D[20]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg21(.Q(Q[21]), .Qbar(Qbar[21]), .D(D[21]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg22(.Q(Q[22]), .Qbar(Qbar[22]), .D(D[22]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg23(.Q(Q[23]), .Qbar(Qbar[23]), .D(D[23]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg24(.Q(Q[24]), .Qbar(Qbar[24]), .D(D[24]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg25(.Q(Q[25]), .Qbar(Qbar[25]), .D(D[25]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg26(.Q(Q[26]), .Qbar(Qbar[26]), .D(D[26]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg27(.Q(Q[27]), .Qbar(Qbar[27]), .D(D[27]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg28(.Q(Q[28]), .Qbar(Qbar[28]), .D(D[28]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg29(.Q(Q[29]), .Qbar(Qbar[29]), .D(D[29]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg30(.Q(Q[30]), .Qbar(Qbar[30]), .D(D[30]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg31(.Q(Q[31]), .Qbar(Qbar[31]), .D(D[31]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));

endmodule
```

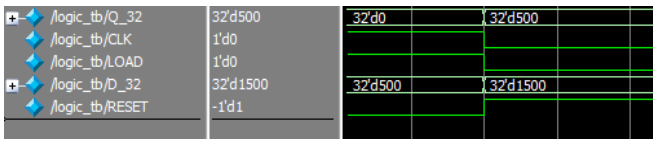
In case it's hard to see:
Sample plain text:

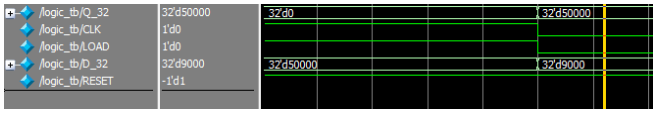
REG1 reg0(.Q(Q[0]), .Qbar(Qbar[0]), .D(D[0]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg1(.Q(Q[1]), .Qbar(Qbar[1]), .D(D[1]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg2(.Q(Q[2]), .Qbar(Qbar[2]), .D(D[2]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg3(.Q(Q[3]), .Qbar(Qbar[3]), .D(D[3]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg4(.Q(Q[4]), .Qbar(Qbar[4]), .D(D[4]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg5(.Q(Q[5]), .Qbar(Qbar[5]), .D(D[5]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
REG1 reg6(.Q(Q[6]), .Qbar(Qbar[6]), .D(D[6]), .L(LOAD), .C(CLK), .nP(one), .nR(RESET));
//32 of these

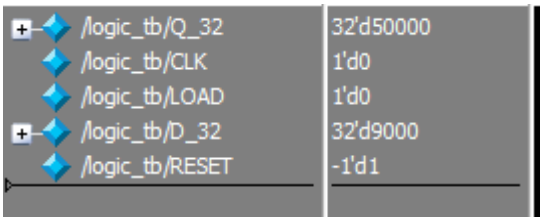
Testing:

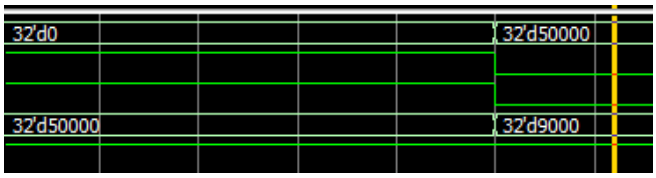
REG32 reg_32(.Q(Q_32), .D(D_32), .LOAD(LOAD), .CLK(CLK), .RESET(RESET)); //module instance

```
initial
begin

#15 LOAD = 0; CLK=1; D_32 = 50;  RESET = 1;
#15 LOAD = 0; CLK=0; D_32 = 50;  RESET = 1;
#15 LOAD = 1; CLK=1; D_32 = 500;  RESET = 0;
#15 LOAD = 0; CLK=0; D_32 = 1500;  RESET = 1;
#15 LOAD = 0; CLK=1; D_32 = 50000000;  RESET = 1;

#15 LOAD = 0; CLK=1; D_32 = 50; RESET = 1;
#15 LOAD = 1; CLK=0; D_32 = 50; RESET = 0;
#15 LOAD = 1; CLK=1; D_32 = 50000; RESET = 1;
#15 LOAD = 0; CLK=0; D_32 = 9000; RESET = 1;
```



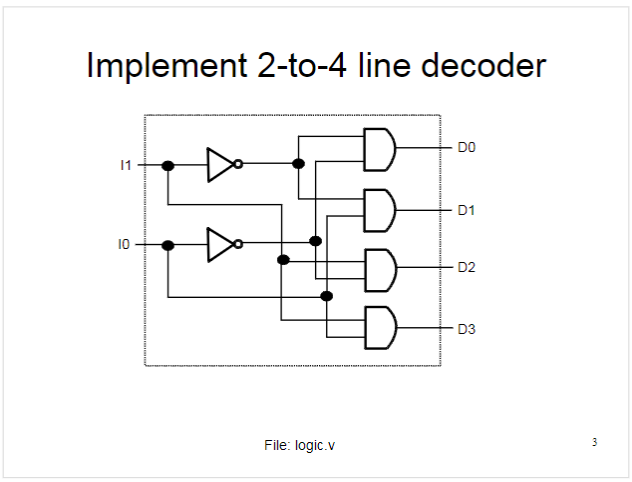






My test doesn't have a real clock so I'm only showing some examples of the value being updated on the positive edge of the clock.
For more testing of the 32 bit registers we can check the register_file_tb.v later. (It actually has a timing issue on the 32nd bit but more on that later.)

Line decoders:

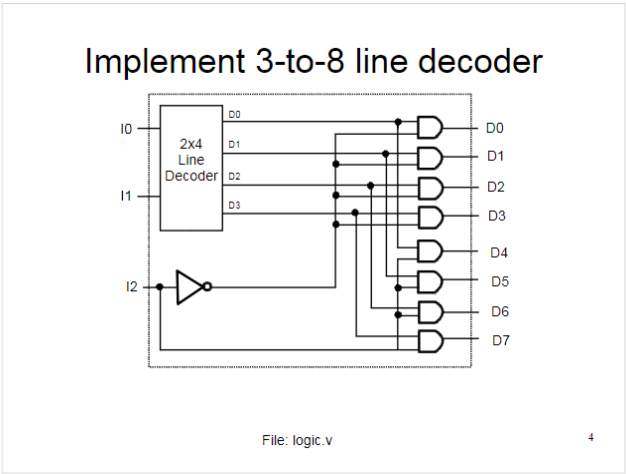


(Kaushik Patra, Lab 16, Slide 2)

A line decoder simply gives every single combination of bits. For our purpose, we will be using it to select which register file should be written into. We need a 5x32 line decoder but to create a 5x32 we will make intermediaries to make the job much simpler.

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

wire I0_not, I1_not;
assign I0_not = ~I[0];
assign I1_not = ~I[1];


// TBD
and and0(D[0], I0_not, I1_not);
and and1(D[1], I[0], I1_not);
and and2(D[2], I0_not, I[1]);
and and3(D[3], I[0], I[1]);


endmodule
```

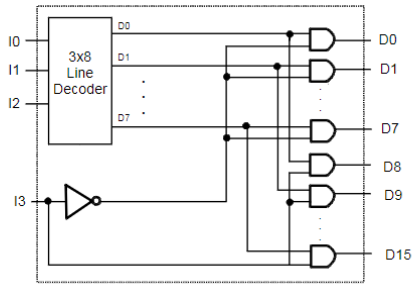
Due to the telescoping nature of the decoder, I will only show the testing of the 5x32 decoder. So, the testing section will come later.



(Kaushik Patra, Lab 16, Slide 2)

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

//TBD
wire [3:0] tempD;
wire I2_Not;
assign I2_Not = ~I[2];
DECODER_2x4 D_2x4(.D(tempD),.I(I[1:0]));

and and0(D[0], tempD[0], I2_Not);
and and1(D[1], tempD[1], I2_Not);
and and2(D[2], tempD[2], I2_Not);
and and3(D[3], tempD[3], I2_Not);
and and4(D[4], tempD[0], I[2]);
and and5(D[5], tempD[1], I[2]);
and and6(D[6], tempD[2], I[2]);
and and7(D[7], tempD[3], I[2]);

endmodule
```

Implement 4-to-16 line decoder

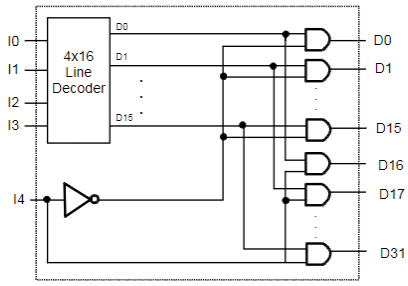


File: logic.v 5

(Kaushik Patra, Lab 16, Slide 3)

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

// TBD
wire [7:0] tempD;
wire I3_Not;
assign I3_Not = ~I[3];
DECODER_3x8 D_3x8(.D(tempD), .I(I[2:0]));

and and0(D[0], tempD[0], I3_Not);
and and1(D[1], tempD[1], I3_Not);
and and2(D[2], tempD[2], I3_Not);
and and3(D[3], tempD[3], I3_Not);
and and4(D[4], tempD[4], I3_Not);
and and5(D[5], tempD[5], I3_Not);
and and6(D[6], tempD[6], I3_Not);
and and7(D[7], tempD[7], I3_Not);
and and8(D[8], tempD[0], I[3]);
and and9(D[9], tempD[1], I[3]);
and and10(D[10], tempD[2], I[3]);
and and11(D[11], tempD[3], I[3]);
and and12(D[12], tempD[4], I[3]);
and and13(D[13], tempD[5], I[3]);
and and14(D[14], tempD[6], I[3]);
and and15(D[15], tempD[7], I[3]);

endmodule
```

Implement 5-to-32 line decoder



File: logic.v 6

(Kaushik Patra, Lab 16, Slide 3)

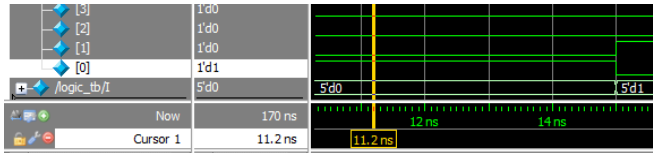
```
// 5x32 Line decoder
module DECODER_5x32(D,I);
// output
output [31:0] D;
// input
input [4:0] I;

// TBD
wire [15:0] tempD;
wire I4_Not;
assign I4_Not = ~I[4];
DECODER_4x16 D_4x8(.D(tempD),.I(I[3:0]));

and and0(D[0], tempD[0], I4_Not);
and and1(D[1], tempD[1], I4_Not);
and and2(D[2], tempD[2], I4_Not);
and and3(D[3], tempD[3], I4_Not);
and and4(D[4], tempD[4], I4_Not);
and and5(D[5], tempD[5], I4_Not);
and and6(D[6], tempD[6], I4_Not);
and and7(D[7], tempD[7], I4_Not);
and and8(D[8], tempD[8], I4_Not);
and and9(D[9], tempD[9], I4_Not);
and and10(D[10], tempD[10], I4_Not);
and and11(D[11], tempD[11], I4_Not);
and and12(D[12], tempD[12], I4_Not);
and and13(D[13], tempD[13], I4_Not);
and and14(D[14], tempD[14], I4_Not);
and and15(D[15], tempD[15], I4_Not);
and and16(D[16], tempD[0], I[4]);
and and17(D[17], tempD[1], I[4]);
and and18(D[18], tempD[2], I[4]);
and and19(D[19], tempD[3], I[4]);
and and20(D[20], tempD[4], I[4]);
and and21(D[21], tempD[5], I[4]);
and and22(D[22], tempD[6], I[4]);
and and23(D[23], tempD[7], I[4]);
and and24(D[24], tempD[8], I[4]);
and and25(D[25], tempD[9], I[4]);
and and26(D[26], tempD[10], I[4]);
and and27(D[27], tempD[11], I[4]);
and and28(D[28], tempD[12], I[4]);
and and29(D[29], tempD[13], I[4]);
and and30(D[30], tempD[14], I[4]);
and and31(D[31], tempD[15], I[4]);

endmodule
```

Testing:
DECODER_5x32 decoder_5x32(.D(D),.I(I)); //module instance

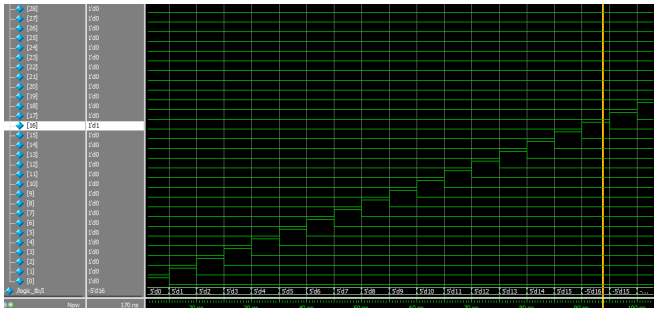
All we need to do to test is check if the bit selected by I becomes 1.

```
83    #5 I = 0 ;
84    #5 I = 1 ;
85    #5 I = 2 ;
86    #5 I = 3 ;
87    #5 I = 4 ;
88    #5 I = 5 ;
89    #5 I = 6 ;
90    #5 I = 7 ;
91    #5 I = 8 ;
92    #5 I = 9 ;
93    #5 I = 10 ;
94    #5 I = 11 ;
95    #5 I = 12 ;
96    #5 I = 13 ;
97    #5 I = 14 ;
98    #5 I = 15 ;
```

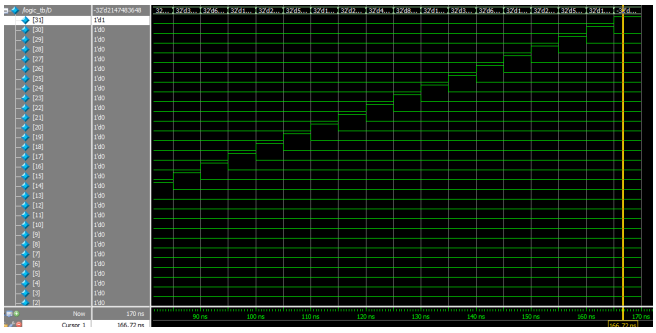
This continues until I == 31. I have cut off the remaining instances.



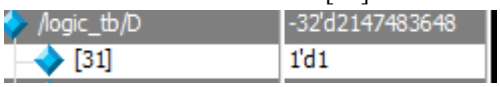
Here we can see that D[0] == 1 when I ==0.
I will only show D's bit changing as the image would be too large and hard to see if I try to capture I as well.



Here we can see an overview of the the bit changing at every step.



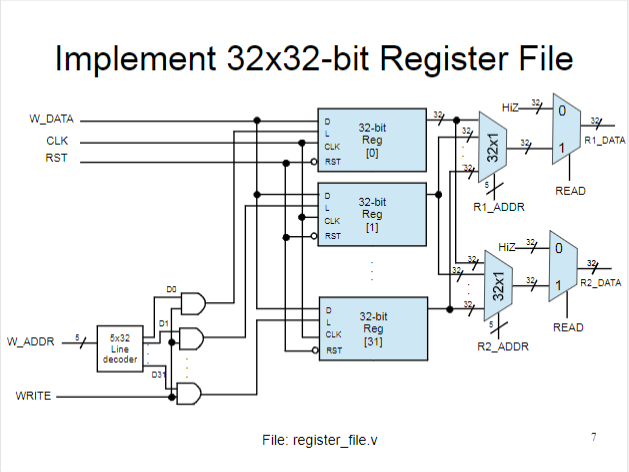
Here is the climb to D[31].



With this we can see that every sub-instance of the decoder works as every possible bit can be activated. (For those recreating this and still skeptical: simply drive the sub-

instances of the decoders in a similar way to convince yourself.)

Register File:



Now we have every subcomponent created so all we have to do is assemble and make intermediate variables.

```
`include "prj_definition.v"

// This is going to be +ve edge clock triggered register file.
// Reset on RST=0
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 list
output [`DATA_INDEX_LIMIT:0] DATA_R1;
output [`DATA_INDEX_LIMIT:0] DATA_R2;

// TBD
wire [31:0] L;
wire [31:0] D;
```

```
wire [31:0] Data_0;
wire [31:0] Data_1;
wire [31:0] Data_2;
wire [31:0] Data_3;
wire [31:0] Data_4;
wire [31:0] Data_5;
wire [31:0] Data_6;
wire [31:0] Data_7;
wire [31:0] Data_8;
wire [31:0] Data_9;
wire [31:0] Data_10;
wire [31:0] Data_11;
wire [31:0] Data_12;
wire [31:0] Data_13;
wire [31:0] Data_14;
wire [31:0] Data_15;
wire [31:0] Data_16;
wire [31:0] Data_17;
wire [31:0] Data_18;
wire [31:0] Data_19;
wire [31:0] Data_20;
wire [31:0] Data_21;
wire [31:0] Data_22;
wire [31:0] Data_23;
wire [31:0] Data_24;
wire [31:0] Data_25;
wire [31:0] Data_26;
wire [31:0] Data_27;
wire [31:0] Data_28;
wire [31:0] Data_29;
wire [31:0] Data_30;
wire [31:0] Data_31;
```

```
wire [31:0] Read_Data_0;
wire [31:0] Read_Data_1;

wire [31:0] HiZ;
assign HiZ = 32'bZ;

//module REG32(Q, D, LOAD, CLK, RESET);
DECODER_5x32 decoder_5x32(.D(D),.I(ADDR_W));
```

```
and and0(L[0], D[0], WRITE);
and and1(L[1], D[1], WRITE);
and and2(L[2], D[2], WRITE);
and and3(L[3], D[3], WRITE);
and and4(L[4], D[4], WRITE);
and and5(L[5], D[5], WRITE);
and and6(L[6], D[6], WRITE);
and and7(L[7], D[7], WRITE);
and and8(L[8], D[8], WRITE);
and and9(L[9], D[9], WRITE);
and and10(L[10], D[10], WRITE);
and and11(L[11], D[11], WRITE);
and and12(L[12], D[12], WRITE);
and and13(L[13], D[13], WRITE);
and and14(L[14], D[14], WRITE);
and and15(L[15], D[15], WRITE);
and and16(L[16], D[16], WRITE);
and and17(L[17], D[17], WRITE);
and and18(L[18], D[18], WRITE);
and and19(L[19], D[19], WRITE);
and and20(L[20], D[20], WRITE);
and and21(L[21], D[21], WRITE);
and and22(L[22], D[22], WRITE);
and and23(L[23], D[23], WRITE);
and and24(L[24], D[24], WRITE);
and and25(L[25], D[25], WRITE);
and and26(L[26], D[26], WRITE);
and and27(L[27], D[27], WRITE);
and and28(L[28], D[28], WRITE);
and and29(L[29], D[29], WRITE);
and and30(L[30], D[30], WRITE);
and and31(L[31], D[31], WRITE);
```

```
REG32 reg_0(.Q(Data_0), .D(DATA_W), .LOAD(L[0]), .CLK(CLK), .RESET(RST));
REG32 reg_1(.Q(Data_1), .D(DATA_W), .LOAD(L[1]), .CLK(CLK), .RESET(RST));
REG32 reg_2(.Q(Data_2), .D(DATA_W), .LOAD(L[2]), .CLK(CLK), .RESET(RST));
REG32 reg_3(.Q(Data_3), .D(DATA_W), .LOAD(L[3]), .CLK(CLK), .RESET(RST));
REG32 reg_4(.Q(Data_4), .D(DATA_W), .LOAD(L[4]), .CLK(CLK), .RESET(RST));
REG32 reg_5(.Q(Data_5), .D(DATA_W), .LOAD(L[5]), .CLK(CLK), .RESET(RST));
REG32 reg_6(.Q(Data_6), .D(DATA_W), .LOAD(L[6]), .CLK(CLK), .RESET(RST));
REG32 reg_7(.Q(Data_7), .D(DATA_W), .LOAD(L[7]), .CLK(CLK), .RESET(RST));
REG32 reg_8(.Q(Data_8), .D(DATA_W), .LOAD(L[8]), .CLK(CLK), .RESET(RST));
REG32 reg_9(.Q(Data_9), .D(DATA_W), .LOAD(L[9]), .CLK(CLK), .RESET(RST));
REG32 reg_10(.Q(Data_10), .D(DATA_W), .LOAD(L[10]), .CLK(CLK), .RESET(RST));
REG32 reg_11(.Q(Data_11), .D(DATA_W), .LOAD(L[11]), .CLK(CLK), .RESET(RST));
REG32 reg_12(.Q(Data_12), .D(DATA_W), .LOAD(L[12]), .CLK(CLK), .RESET(RST));
REG32 reg_13(.Q(Data_13), .D(DATA_W), .LOAD(L[13]), .CLK(CLK), .RESET(RST));
REG32 reg_14(.Q(Data_14), .D(DATA_W), .LOAD(L[14]), .CLK(CLK), .RESET(RST));
REG32 reg_15(.Q(Data_15), .D(DATA_W), .LOAD(L[15]), .CLK(CLK), .RESET(RST));
REG32 reg_16(.Q(Data_16), .D(DATA_W), .LOAD(L[16]), .CLK(CLK), .RESET(RST));
REG32 reg_17(.Q(Data_17), .D(DATA_W), .LOAD(L[17]), .CLK(CLK), .RESET(RST));
REG32 reg_18(.Q(Data_18), .D(DATA_W), .LOAD(L[18]), .CLK(CLK), .RESET(RST));
REG32 reg_19(.Q(Data_19), .D(DATA_W), .LOAD(L[19]), .CLK(CLK), .RESET(RST));
REG32 reg_20(.Q(Data_20), .D(DATA_W), .LOAD(L[20]), .CLK(CLK), .RESET(RST));
REG32 reg_21(.Q(Data_21), .D(DATA_W), .LOAD(L[21]), .CLK(CLK), .RESET(RST));
REG32 reg_22(.Q(Data_22), .D(DATA_W), .LOAD(L[22]), .CLK(CLK), .RESET(RST));
REG32 reg_23(.Q(Data_23), .D(DATA_W), .LOAD(L[23]), .CLK(CLK), .RESET(RST));
REG32 reg_24(.Q(Data_24), .D(DATA_W), .LOAD(L[24]), .CLK(CLK), .RESET(RST));
REG32 reg_25(.Q(Data_25), .D(DATA_W), .LOAD(L[25]), .CLK(CLK), .RESET(RST));
REG32 reg_26(.Q(Data_26), .D(DATA_W), .LOAD(L[26]), .CLK(CLK), .RESET(RST));
REG32 reg_27(.Q(Data_27), .D(DATA_W), .LOAD(L[27]), .CLK(CLK), .RESET(RST));
REG32 reg_28(.Q(Data_28), .D(DATA_W), .LOAD(L[28]), .CLK(CLK), .RESET(RST));
REG32 reg_29(.Q(Data_29), .D(DATA_W), .LOAD(L[29]), .CLK(CLK), .RESET(RST));
REG32 reg_30(.Q(Data_30), .D(DATA_W), .LOAD(L[30]), .CLK(CLK), .RESET(RST));
REG32 reg_31(.Q(Data_31), .D(DATA_W), .LOAD(L[31]), .CLK(CLK), .RESET(RST));
```

The following muxs are long so I'll post the plain text:

```
MUX32_32x1 mux_32_0(.Y(Read_Data_0), .I0(Data_0),
.I1(Data_1), .I2(Data_2), .I3(Data_3), .I4(Data_4),
.I5(Data_5), .I6(Data_6), .I7(Data_7),
                .I8(Data_8), .I9(Data_9), .I10(Data_10),
.I11(Data_11), .I12(Data_12), .I13(Data_13), .I14(Data_14),
.I15(Data_15),
                .I16(Data_16), .I17(Data_17), .I18(Data_18),
.I19(Data_19), .I20(Data_20), .I21(Data_21), .I22(Data_22),
.I23(Data_23),
                .I24(Data_24), .I25(Data_25), .I26(Data_26),
.I27(Data_27), .I28(Data_28), .I29(Data_29), .I30(Data_30),
.I31(Data_31), .S(ADDR_R1));

MUX32_32x1 mux_32_1(.Y(Read_Data_1), .I0(Data_0),
.I1(Data_1), .I2(Data_2), .I3(Data_3), .I4(Data_4),
.I5(Data_5), .I6(Data_6), .I7(Data_7),
                .I8(Data_8), .I9(Data_9), .I10(Data_10),
.I11(Data_11), .I12(Data_12), .I13(Data_13), .I14(Data_14),
.I15(Data_15),
                .I16(Data_16), .I17(Data_17), .I18(Data_18),
.I19(Data_19), .I20(Data_20), .I21(Data_21), .I22(Data_22),
.I23(Data_23),
                .I24(Data_24), .I25(Data_25), .I26(Data_26),
.I27(Data_27), .I28(Data_28), .I29(Data_29), .I30(Data_30),
.I31(Data_31), .S(ADDR_R2));
```

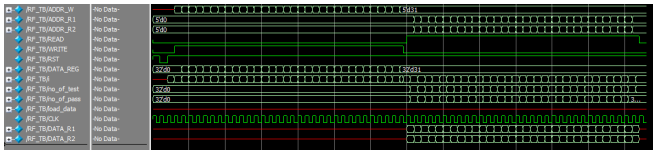
```
MUX32_2x1 mux0(.Y(DATA_R1), .I0(HiZ), .I1(Read_Data_0), .S(READ));
MUX32_2x1 mux1(.Y(DATA_R2), .I0(HiZ), .I1(Read_Data_1), .S(READ));

endmodule
```

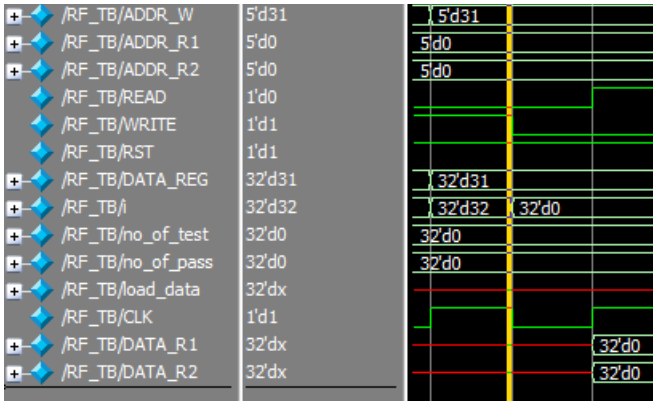
The testbench was created by Kaushik Patra and no further code posted will be from me. (Due to lack of time, I have only gotten as far as the register_file.v implementation and the control unit will only be discussed from a theoretical point of view.)

Testing:
The strategy is simply to load data into the register file and then read it back. The data written in is the same as the number of the index.

```
`include "prj_definition.v"
module RF_TB;
// Storage list
reg [`REG_ADDR_INDEX_LIMIT:0] ADDR_W;
reg [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1;
reg [`REG_ADDR_INDEX_LIMIT:0] ADDR_R2;
// reset
reg READ, WRITE, RST;
// data register
reg [`DATA_INDEX_LIMIT:0] DATA_REG;
integer i; // index for memory operation
integer no_of_test, no_of_pass;
integer load_data;

// wire lists
wire  CLK;
wire [`DATA_INDEX_LIMIT:0] DATA_R1;
wire [`DATA_INDEX_LIMIT:0] DATA_R2;

// Clock generator instance
CLK_GENERATOR clk_gen_inst(.CLK(CLK));

// 64MB memory instance
REGISTER_FILE_32x32 ref_32x32_inst(.DATA_R1(DATA_R1), .DATA_R2(DATA_R2), .ADDR_R1(ADDR_R1),
                                   .ADDR_R2(ADDR_R2), .DATA_W(DATA_REG), .ADDR_W(ADDR_W),
                                   .READ(READ), .WRITE(WRITE), .CLK(CLK), .RST(RST));
```

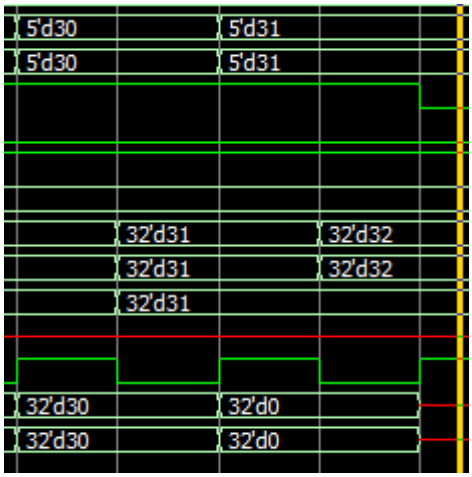
```
initial
begin
RST=1'b1;
READ=1'b0;
WRITE=1'b0;
DATA_REG = {`DATA_WIDTH{1'b0} };
ADDR_R1 = {`DATA_WIDTH{1'b0} };
ADDR_R2 = {`DATA_WIDTH{1'b0} };
no_of_test = 0;
no_of_pass = 0;

// Start the operation
#10    RST=1'b0;
#10    RST=1'b1;
// Write cycle
for(i=0;i<32; i = i + 1)
begin
#10     DATA_REG=i; READ=1'b0; WRITE=1'b1; ADDR_W = i;
end


#5 READ=1'b0; WRITE=1'b0;
// test of write data
for(i=0;i<32; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R1 = i; ADDR_R2 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i)
            $write("[TEST @ %0dns] Read %1b, Write %1b, expect
        else
            no_of_pass  = no_of_pass + 1;
end


#5    READ=1'b0; WRITE=1'b0; // No op

#10 $write("\n");
    $write("\tTotal number of tests %d\n", no_of_test);
    $write("\tTotal number of pass  %d\n", no_of_pass);
    $write("\n");
    $stop;

end
endmodule;
```





Once bit 32 is being written to the operation changes and 0 ends up being what remains within the 32nd register.

```
# [TEST @ 665ns] Read 1, Write 0, expecting 0000001f, got 00000000 [FAILED]
#
#       Total number of tests          32
#       Total number of pass           31
#
# ** Note: $stop    : C:/Modelsim Projects/Project 3/register_file_tb.v(88)
#    Time: 680 ns  Iteration: 0  Instance: /RF_TB
```

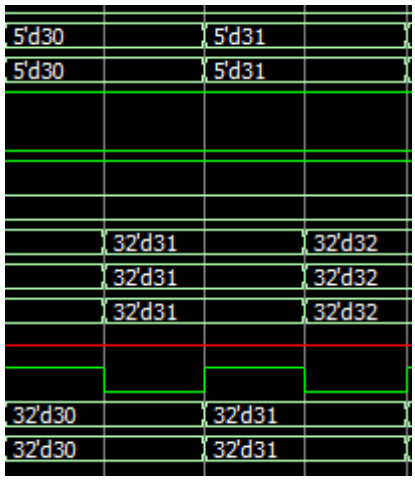


Bit 32 as 0.

If I change the value of the for loops to 33 instead of 32 then 32 is written into but of course a new error pops up (the 33rd doesnt exist obviously).

```
for(i=0;i<33; i = i + 1)
begin
#10     DATA_REG=i; READ=1'
end

#5 READ=1'b0; WRITE=1'b0;
// test of write data
for(i=0;i<33; i = i + 1)
begin
```

```
[TEST @ 685ns] Read 1, Write 0, expecting 00000020, got 00000000 [FAILED]

      Total number of tests          33
      Total number of pass           32

** Note: $stop    : C:/Modelsim Projects/Project 3/register_file_tb.v(88)
   Time: 700 ns  Iteration: 0  Instance: /RF_TB
```

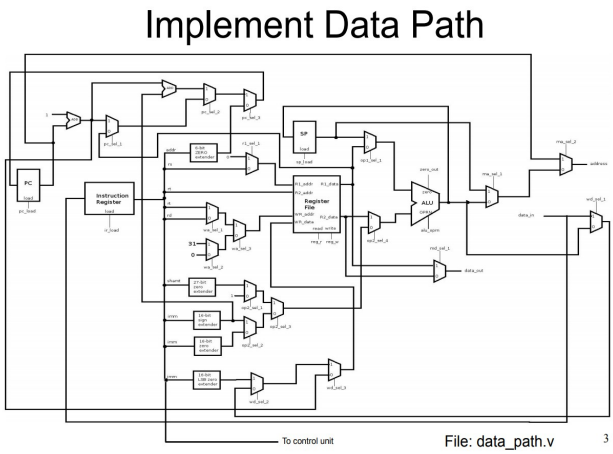


32 has been written into.

So the issue is either a strange timing issue on my part or an error with the test bench. Regardless, every register can be read from and written to, therefore my register file works as intended (for the most part).

Next would be the control unit. I have yet to begin it; therefore, I will outline what should be done, not what I have done.

III. Control Unit Design



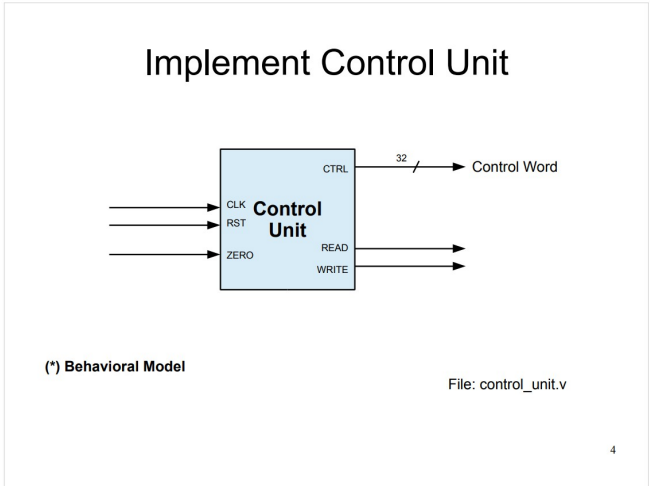
(Kaushik Patra, Lab 17, Slide 2)

(Please get the file from Canvas instead, it's very large and can't be completely captured here. Lab 17 under the Lab Module.)

The data path sends all signals to the appropriate/shown segments. Any line pointing in from text into a module is from the data path. The data path is arbitrary and designed by you.

```
`include "prj_definition.v"
module DATA_PATH(DATA_OUT, ADDR, ZERO, INSTRUCTION, DATA_IN, CTRL, CLK, RST);

// output list
output [`ADDRESS_INDEX_LIMIT:0] ADDR;
output ZERO;
output [`DATA_INDEX_LIMIT:0] DATA_OUT, INSTRUCTION;

// input list
input [`CTRL_WIDTH_INDEX_LIMIT:0]  CTRL;
input CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_IN;

// TBD

endmodule
```

You would take CTRL and choose which bits do what and pass those down the data path. The DATA_IN and DATA_OUT are separate from the CTRL so when those appear on the above solution to the data path then use those instead of the CTRL bits.

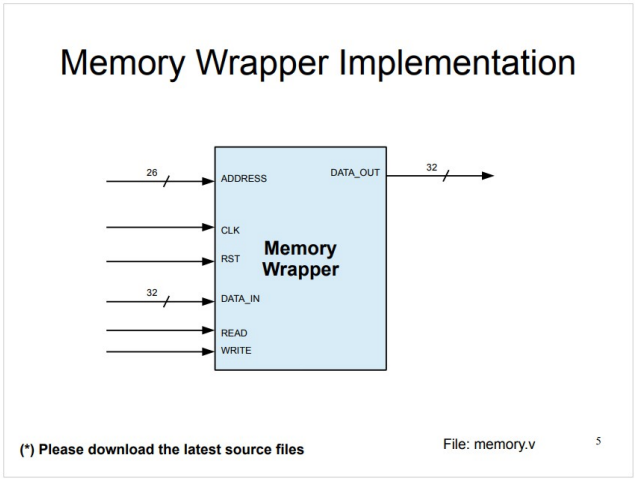


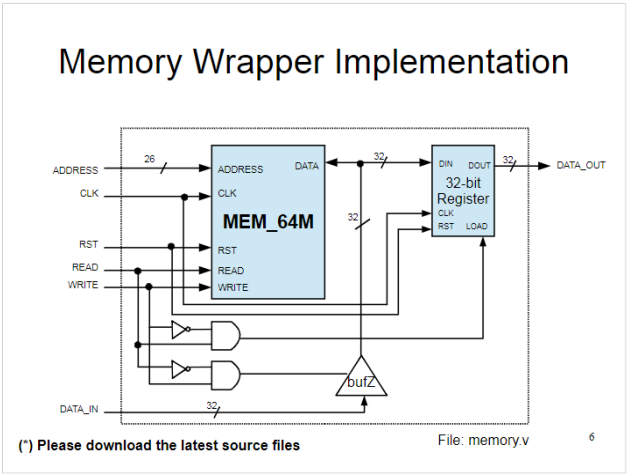
(Kaushik Patra, Lab 17, Slide 2)

The datapath works in conjunction with the control unit so your instantiation depends entirely on the data path instantiation. Essentially, you want to take care of the ZERO flag, run the clock, and decide when to read or write.

IV. Memory Unit

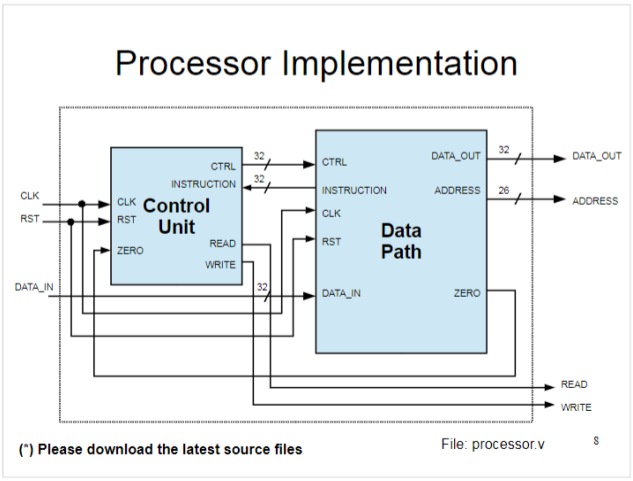
The memory acts as the "permanent" storage for the system.





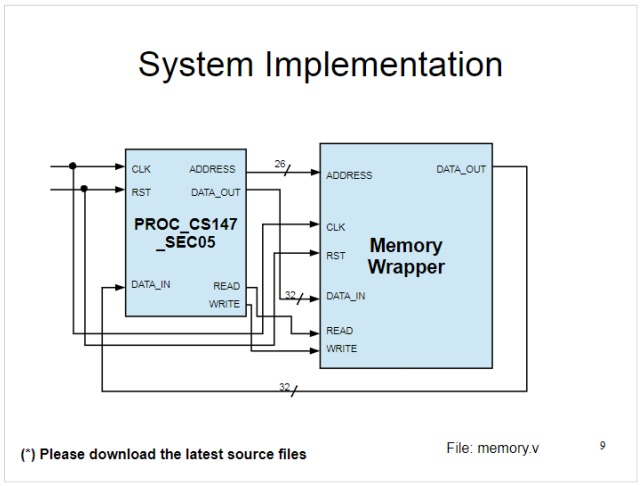
(Kaushik Patra, Lab 17, Slide 3)

V. Quick discussion of entire system assembly and testing



(Kaushik Patra, Lab 17, Slide 4)

Once the datapath and control unit have been made we can simply make the appropriate connections then have a processor. With this processor made the system just needs memory at this point.



(Kaushik Patra, Lab 17, Slide 5)

The clock and reset should be run seperately by the da_vinci.v instantiation.

```
`include "prj_definition.v"
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;
```

This is already premade by Kaushik Patra, as well as the test bench, so simply running da_vinci_tb.v will test your entire system.

```
`include "prj_definition.v"
module DA_VINCI_TB;
// output list
wire [`ADDRESS_INDEX_LIMIT:0] ADDR;
wire READ, WRITE, CLK;
// inout list
wire [`DATA_INDEX_LIMIT:0] DATA_OUT, DATA_IN;

// reset
reg RST;

// Clock generator instance
CLK_GENERATOR clk_gen_inst(.CLK(CLK));

// DA_VINCI v1.0 instance
defparam da_vinci_inst.mem_init_file = "fibonacci.dat";
//defparam da_vinci_inst.mem_init_file = "RevFib.dat";
DA_VINCI da_vinci_inst(.MEM_DATA_OUT(DATA_OUT), .MEM_DATA_IN(DATA_IN),
                       .ADDR(ADDR), .READ(READ),
                       .WRITE(WRITE), .CLK(CLK), .RST(RST));
```

```
initial
begin
RST=1'b1;
#5 RST=1'b0;
#5 RST=1'b1;

// TBD: rest of the test code goes here.

//# 20 $stop;
#5000  //$writememh("RevFib_mem_dump.dat",
       $writememh("fibonacci_mem_dump.dat'
       $stop;

end
endmodule;
```

The cut off lines:

#5000 //$writememh("RevFib_mem_dump.dat", da_vinci_inst.memory_inst.memory_inst.sram_32x64m, 'h03fffff0, 'h03ffffff);

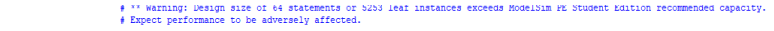
$writememh("fibonacci_mem_dump.dat", da_vinci_inst.memory_inst.memory_inst.sram_32x64m, 'h01000000, 'h0100000f);

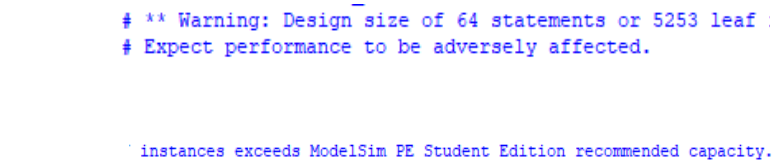
$stop;

VI. Conclusion

The entire system could not be tested as I've run out of time for the project, but everything I have written is fully functioning. With the exception of the register file, which could be a performance issue due to the student version's leaf limitations (will attach a readout below, it has to be a form of timing error/delay), most components work. We have made the ALU, which required an adder, multiplier, multiplexers, barrel shifter, 32 bit AND, NOR, OR, and NOT gates. We have made a register file, which need line decoders, multiplexers, registers, and multiplexers. Every small component and module has been made aside from those shown from Lab 17.

A probable cause for the time delay:





** Warning: Design size of 64 statements or 5253 leaf instances exceeds ModelSim PE Student Edition recommended capacity.

Expect performance to be adversely affected.