Gate Level Model of Davinci v1.0

SHUANG PAN

San Jose State University

shuang.pan@sjsu.edu

***Abstract*—The project is to use ModelSim Student Edition to implement to write gate level model of a bare minimum computer system Davinci v1.0, which it supports the instruction set cs 147DV that is taught by Professor. Patra in the class. The followings are the objectives of this lab:**

1) Implement a mixed model of a computer system with a 32-bit processor and 256MB memory. The processor supports the instruction set CS147DV. There will be two major parts in the implementation of processor.
    1. Mixed level data path implementation
        i. Gate level ALU
        ii. Gate level register file
        iii. Behavioral level memory
        iv. Gate level data path
    2. Behavioral control circuit implementation
2) Implement test for Davinci v1.0m.

Before start implement all big components of the processor, we need to build some tiny circuit components which will be used to construct ALU, register file, and then data path.

1. Half Adder

The half adder has two inputs A and B, one XOR gate and one AND gate, one-bit output Y and one carryout bit C. The half adder doesn't have the carry-in bit which is used in practice.

Y = A XOR B
C = A. B

The following figure 1.1 is the digital circuit of the half adder.

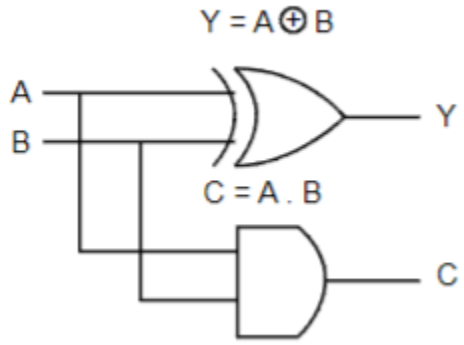


Figure 1.1 Half adder

```
module HALF_ADDER(Y,C,A,B);
output Y,C;
input A,B;

xor inst1(Y, A, B);
and inst2(C, A, B);

endmodule
```

Figure 1.2 Implementation of half adder

The above figure 1.2 is the implementation of the half adder. It uses the XOR and AND operator to get and C.

To test the half adder, four tests are used to give different values to A and B (figure 1.3).

```
module half_adder_tb;
reg A, B;
wire Y, C;

HALF_ADDER hs_inst_1(.Y(Y), .C(C), .A(A), .B(B));

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

#5 ;
end

endmodule
```

Figure 1.3 Implementation of half adder testbench

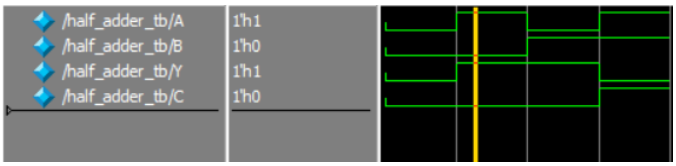


Figure 1,4 Half adder testbench waveform

The above figure 1.4 shows the results. When A is 0 and B is 0, Y is 0 and C is 0. When A is 1 and B is 0, Y is 1 and C is 0. When A is 0 and B is 1, Y is 1 and C is 0. When both A and B are 0, Y is 0 and C is 1.

2. Full Adder

After that, we build the full added which is constructed by two half adders. The following figure 2.1 the digital circuit of the full adder.

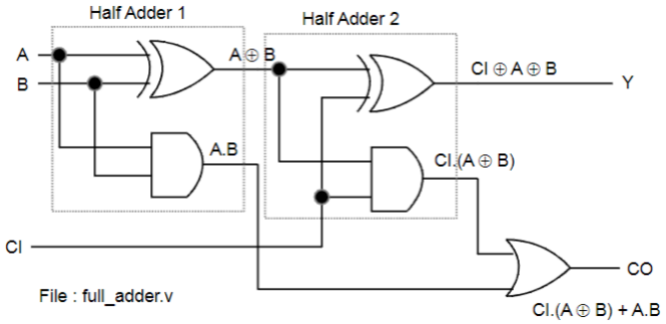


Figure 2.1 Full Adder

The full adder has three inputs which are A, B and Ci and two output which are Y and CO.

The following formula is relation between input and output.

$$Y = CI \oplus (A \oplus B)$$
$$CO = CI.(A \oplus B) + A.B$$

The following figure 2.2 is the implementation of full adder.

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

wire halsum, halco, ha2co;

HALF_ADDER ha1(.Y(halsum), .C(halco), .A(A), .B(B));
HALF_ADDER ha2(.Y(S), .C(ha2co), .A(halsum), .B(CI));
or inst1(CO, halco, ha2co);

endmodule
```

Figure 2.2 Implementation of full adder

The output halsum of the first adder and CI will be the input of the second half adder. The output CO will be the or operation between carryout bit of the first adder and the AND result between CI and the output halsum of the first half adder.

To test the full adder, the testbench is created as the following figure 2.3.

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

FULL_ADDER ha_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=1; CI=0;
#5 A=1; B=1; CI=0;
#5 A=0; B=0; CI=1;
#5 A=1; B=0; CI=1;
#5 A=0; B=1; CI=1;
#5 A=1; B=1; CI=1;
#5;
end

endmodule
```

Figure 2.3 Implementation of full adder

All eight test cases for A, B, and C are created. The following figure 2.4 is the waveform of results.

Figure 2.4 Full adder testbench

When CI is 0, the result is the same with the half adder. When Ci is 1,
A = 0, B = 0; Y = 1, CO = 0

A = 1, B = 0; Y = 0, CO = 1
A = 0; A = 1; Y = 0, CO = 1
A = 1; B = 1; Y = 1, CO = 1

3. Adder and Subtractor

The following figure 3.1 is the 32bits adder. It has 32 full adders and two 32bits inputs. The outputs are one 32bits bits number. Besides, it has the carry-in bit 0 and the carryout bit.

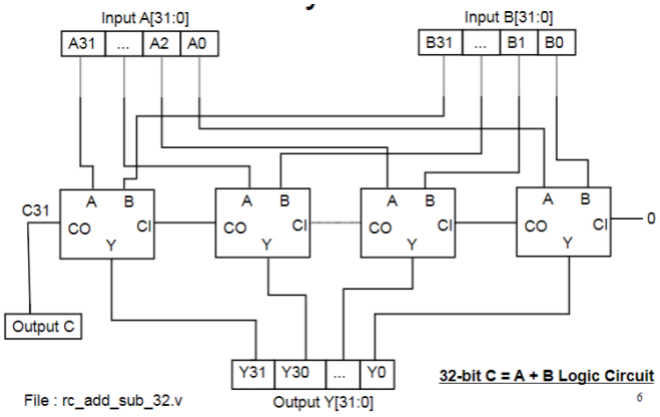


Figure 3.1 32bits adder

The following figure 3.2 is 32bits adder and subtractor. It uses SnA to control the adder and subtractor. When SnA is 0, it is the adder because the number will keep the same after the xor operation with 0. When SnA is 1, the input the two's complement of the original number which means the operation will become the subtraction.

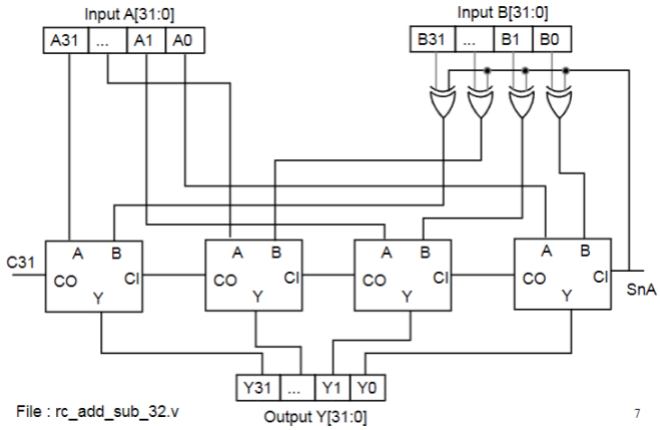


Figure 3.2 32bits Adder and Subtractor

For repeating structure at gate level, we use *generate* and *endgenerate* reversed word and put the loop with name in the middle so that it will not cause problems to run the loop.

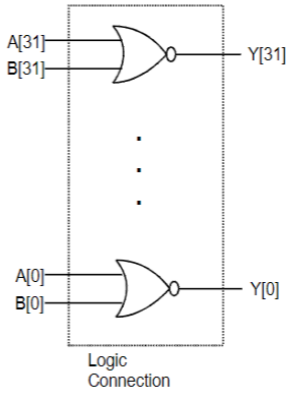


Figure 3 Repeating structure at gate level*

The following figure 3.3 is the implementation of 32bits adder and subtractor.

```
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 [31:0] xorResult;
wire [31:0] subCarry;

genvar i;
genvar j;

generate
for(j = 0; j < 32; j = j + 1)
begin : xor_loop
  xor xor_inst(xorResult[j], SnA, B[j]);
end
endgenerate

FULL_ADDER inst1(.S(Y[0]), .CO(subCarry[0]), .A(A[0]), .B(xorResult[0]), .CI(SnA));

generate
for(i = 1; i < 31; i = i + 1)
begin : rc_add_sub_32_loop
  FULL_ADDER full_adder(.S(Y[i]), .CO(subCarry[i]), .A(A[i]), .B(xorResult[i]), .CI(subCarry[i - 1]));
end
endgenerate

FULL_ADDER inst2(.S(Y[31]), .CO(CO), .A(A[31]), .B(xorResult[31]), .CI(subCarry[30]));

endmodule
```

Figure 3.3 Implementation of 32bits adder and subtractor

At the beginning, it performs the XOR operation for one 32bits number for adder or subtractor. For the operation, the carry-in bit will be SnA and it is separated from the loop. After that, the loop will run 30 times, for every full adder, it will use the carryout bit from the last full adder as the carry-in bit. The final operation will be separated from the loop because it will return the last bit of Y and the CO bit.

The following figure 3.4 is the 64bits adder and subtractor. It can use two 32bits adder and subtractor or 64 full adders. The following figure uses 64 full adders.

```
module RC_ADD_SUB_64(Y, CO, A, B, SnA);
// output list
output [63:0] Y;
output CO;
// input list
input [63:0] A;
input [63:0] B;
input SnA;

wire [63:0] xorResult;
wire [63:0] subCarry;

genvar i;
genvar j;

generate
for(j = 0; j < 64; j = j + 1)
begin : xor_loop
  xor xor_inst(xorResult[j], SnA, B[j]);
end
endgenerate

FULL_ADDER inst1(.S(Y[0]), .CO(subCarry[0]), .A(A[0]), .B(xorResult[0]), .CI(SnA));

generate
for(i = 1; i < 63; i = i + 1)
begin : rc_add_sub_64_loop
  FULL_ADDER full_adder(.S(Y[i]), .CO(subCarry[i]), .A(A[i]), .B(xorResult[i]), .CI(subCarry[i - 1]));
end
endgenerate

FULL_ADDER inst2(.S(Y[63]), .CO(CO), .A(A[63]), .B(xorResult[63]), .CI(subCarry[62]));


endmodule
```

Figure 3.4 64bits adder and subtractor

The logic is the same with the 32bits one.
To test the adder and subtractor, some numbers will be used to test the operation (figure 3.5).

```
module rc_add_sub_32_tb;
reg SnA;
reg SnA2;
reg [`DATA_INDEX_LIMIT:0] A;
reg [`DATA_INDEX_LIMIT:0] B;
wire [31:0] S;
wire CO;
wire CO2;

reg [63:0] C;
reg [63:0] D;
wire [63:0] S_64bits;

RC_ADD_SUB_32 inst(.Y(S), .CO(CO), .A(A), .B(B), .SnA(SnA));
RC_ADD_SUB_64 inst1(.Y(S_64bits), .CO(CO2), .A(C), .B(D), .SnA(SnA2));

initial
begin
A = 'h12345678; B = 'h87654321; SnA = 0;
#5 A = 'h12345678; B = 'h87654321; SnA = 1;
#5 C = 'h1234567812345678; D = 'h8765432187654321; SnA2 = 0;
#5 C = 'h1234567812345678; D = 'h8765432187654321; SnA2 = 1;
#5;
end

endmodule
```

Figure 3.5 Adder and subtractor testbench

To test the 32bits one, two 32-bits numbers A and B and SnA which is either zero or one will be used. To test the 64bits one, two 64-bits numbers C and D and SnA which is either zero or one will be used. The following waveform is the result for those operation (figure 3.6).

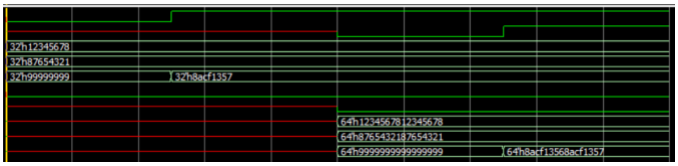


Figure 3.6 Implementation of adder and subtractor

A = 32'h12345678
B = 32'h87654321
A + B = 32'h999999999
A – B = 32'h8acf1357

C = 64'h1234567812345678
D = 64'h8765432187654321
C + D = 64'h9999999999999999
C – D = 64'h8acf13568acf1357

4. Mux

A multiplexer is used to select the digital signal you want from multiples signals. The most basic multiplexer is 1-bit 2x1 mux (figure 4.1).

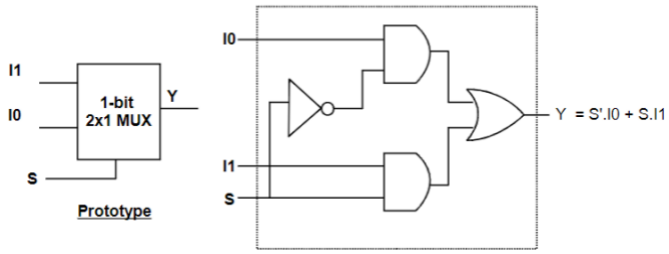


Figure 4.1 1-bit 2x1 mux

It has three inputs. I1 and I0 are selected by signal S. The circuit has one inverter, two and gates, and one or gate.

Output Y = S'I0 + S.I1

The following figure 4.2 is the implementation of 1-bit 2x1 mux.

```
module MUX1_2x1(Y,I0, I1, S);
//output list
output Y;
//input list
input I0, I1, S;

wire subresult1;
wire subresult2;
wire subresult3;
and inst(subresult1, S, I1);
not inst1(subresult2, S);
and inst2(subresult3, subresult2, I0);
or  inst3(Y, subresult3, subresult1);

endmodule
```

Figure 4.2 Implementation of 1-bit 2x1 mux

As described above, the intermediate results from not gate and two and gates are stored. Finally, the output will be decided using or gate.

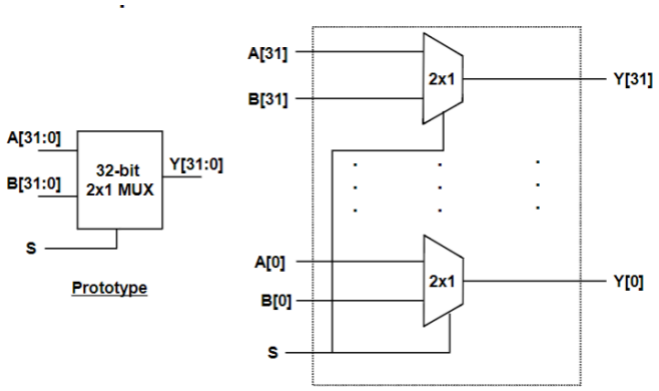


Figure 4.3 32bits 2x1 mux

The above figure is the 32-bits 2x1 mux. It has 32 1-bit 2x1 mux, two 32-bits input A and B, and selection input S, and a 32-bits output.

```
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;

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

Figure 4.4 Implementation of 32-bits 2x1 mux

The above is the implementation of 32-bits 2x1 mux. It is the repeating structure at gate level, so the same method will be used as mentioned above.

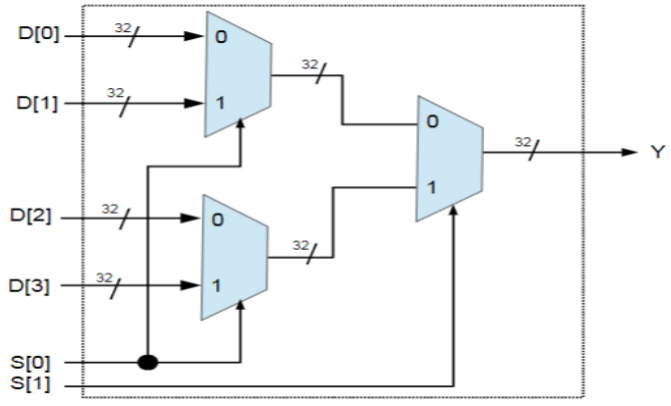


Figure 4.5 32-bit 4x1 mux

The above diagram is the 32-bit 4x1 mux. It has four inputs D[3:0]. Three 32-bit 2x1 mux will be used. The output Y will be one of D[3:0] according to S.

```
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] result1;
wire [31:0] result2;

MUX32_2x1 mux32_2x1one(result1, I0, I1, S[0]);
MUX32_2x1 mux32_2x1two(result2, I2, I3, S[0]);
MUX32_2x1 mux32_2x1three(Y, result1, result2, S[1]);

endmodule
```

Figure 4.6 Implementation of 32-bit 4x1 mux

The first two 32-bit 2x1 mux will select two inputs from D[3:0] according to S[0]. The third 32-bit 2x1 mux will select the final result from two subresults according to S[1].

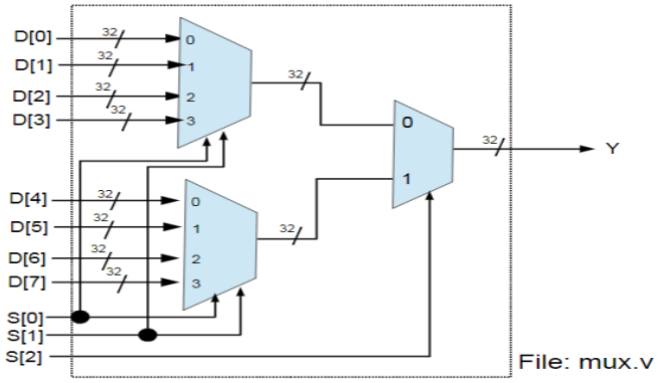


Figure 4.7 32-bit 8x1 mux

The above diagram is the 32-bit 8x1 mux. It has eight inputs D[7:0]. Two 32-bit 4x1 mux and one 32-bit 2x1 mux will be used. The output Y will be one of D[7:0] according the S.

```
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;

wire [31:0] result1;
wire [31:0] result2;

MUX32_4x1 mux32_4x1one(result1, I0, I1, I2, I3, S[1:0]);
MUX32_4x1 mux32_4x1two(result2, I4, I5, I6, I7, S[1:0]);
MUX32_2x1 mux32_2x1one(Y, result1, result2, S[2]);

endmodule
```

Figure 4.8 Implementation of 32-bit 8x1 mux

The two 32-bit 4x1 mux will select two inputs from D[7:0] according to S[1:0]. The 32-bit 2x1 mux will select the final result from two subresults according to S[2].

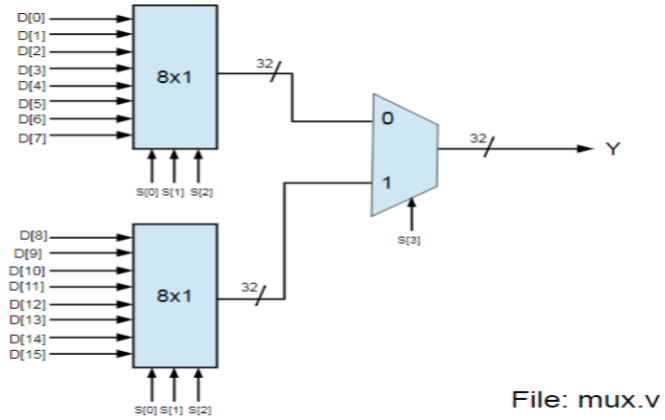


Figure 4.9 32-bit 16x1 mux

The above diagram is the 32-bit 16x1 mux. It has sixteen inputs D[15:0]. Two 32-bit 8x1 mux and one 32-bit 2x1 mux will be used. The output Y will be one of D[15:0] according the S.

```
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;

wire [31:0] result1;
wire [31:0] result2;

MUX32_8x1 mux32_8x1one(result1, I0, I1, I2, I3, I4, I5, I6, I7, S[2:0]);
MUX32_8x1 mux32_8x1two(result2, I8, I9, I10, I11, I12, I13, I14, I15, S[2:0]);
MUX32_2x1 mux32_2x1(Y, result1, result2, S[3]);

endmodule
```

Figure 4.10 32-bit 16x1 mux

The two 32-bit 8x1 mux will select two inputs from D[15:0] according to S[2:0]. The 32-bit 2x1 mux will select the final result from two subresults according to S[3].

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

wire [31:0] result1;
wire [31:0] result2;

MUX32_16x1 mux32_16x1one(result1, I0, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13, I14, I15, S[3:0]);
MUX32_16x1 mux32_16x1two(result2, I16, I17, I18, I19, I20, I21, I22, I23, I24, I25, I26, I27, I28, I29, I30, I31, S[3:0]);
MUX32_2x1 mux32_2x1(Y, result1, result2, S[4]);
endmodule
```

Figure 4.11 Implementation of 32-bit 32x1 mux

The above figure is the implementation of 32-bit 32x1 mux. It has thirty two inputs D[31:0]. Two 32-bit 16x1 mux and one 32-bit 2x1 mux will be used. The output Y will be one of D[31:0] according the S. The two 32-bit x1 mux will select two inputs from D[31:0] according to S[3:0]. The 32-bit 2x1 mux will select the final result from two subresults according to S[4].

5. Mult

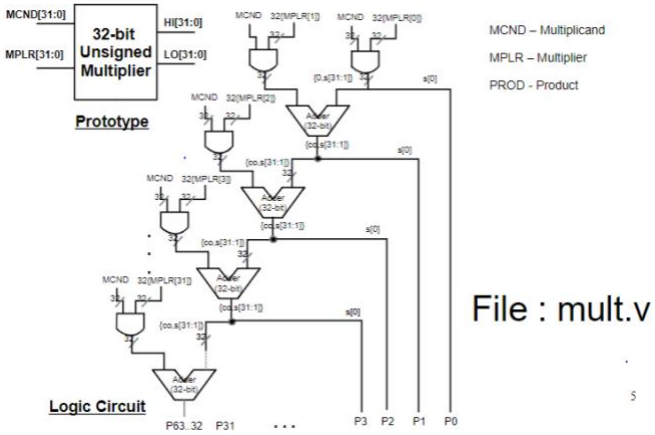


Figure 5.1 32-bits Unsigned Multiplier

After building the 32-bits and adder, we can construct the 32-bits unsigned multiplier using 32-bits AND gate and adder. The multiplier has 32 32-bits AND gate and 31 adders. It has two inputs which are one 32-bits MCND and one 32-bits MPLR and two outputs which are 32-bits HI and 32-bits LO. HI and LO consist the 64-bits number.

```
module MULT32_U(HI, LO, A, B);
// output list
output [31:0] HI;
output [31:0] LO;
// input list
input [31:0] A;
input [31:0] B;

wire [31:0] result1;
wire CO [31:0];
AND32_2x1 and1(result1, A, {32{B[0]}});

wire [31:0] sub_value [31:0];
buf32 buf1(sub_value[0], result1);
buf buf2(LO[0], sub_value[0][0]);
buf buf3(CO[0], 0);

genvar i;
generate
  for(i = 1; i <= 31; i = i + 1)
  begin : add_loop
    wire [31:0] and_result;
    wire [31:0] add_result;
    AND32_2x1 and3(and_result, A, {32{B[i]}});
    RC_ADD_SUB_32 addX(add_result, CO[i], and_result, {CO[i - 1], sub_value[i - 1][31:1]}, 1'b0);
    buf32 buf4(sub_value[i], add_result);
    buf buf2(LO[i], sub_value[i][0]);
  end
endgenerate
    buf32 buf5(HI, {CO[31], sub_value[31][31:1]});

endmodule
```

Figure 5.2 Implementation of 32-bits multiplier

The above figure 5.2 is the implementation of 32-bits multiplier. At the beginning, a 32-bits carryout and 32 32-bits immediate results are defined. Before starting the loop, the first lower bit is sub_value[0][0], and the sub_value[0] is the immediate result after the 32 bits and operations. The 32-bits AND gate will be implemented in the logic_32bits section at the following. In the loop, we need a AND result and a add result which will be used for calculation. Firstly, the AND result will be computed between multiplicand and the ith bit of the multiplier and then do the addition between and_result and the concatenation of 0 and sub_value[i - 1][31:0]. After that, we store the add_result of the current loop in the sub_value[i] using buf32 which will be implemented in the section of logci_32bits. Then, the current carryout bit will be saved. After finishing the loop, the last lower bit is also saved. Finally, the concatenation of 0 and sub_value[31][31:1] will be HI part.

After that, we need to implement the signed multiplication circuit. The following figure 5.3 is the circuit the signed multiplication according to unsigned multiplication.

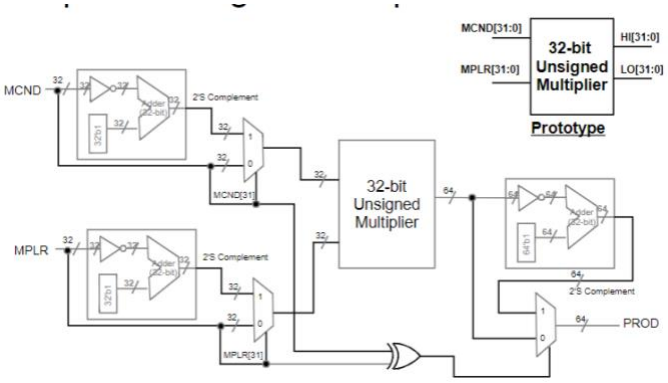


Figure 5.3 Signed multiplication circuit

At the beginning, we need to convert the multiplicand and multiplier to its 2's complement form. The 32-bits 2's complement will be implemented in the logic section at the following. The we applying 32-bits 2x1 mux to select the positive one according to the 31th bit of the multiplicand and multiplier. After doing the 32-bit multiplication, the result will be selected between the result and its 2's complement according the xor result of to the 31th bit of the multiplicand and multiplier. It both of them are positive or negative. The result will be positive. Otherwise, the result be negative.

The following figure 5.4 is the implementation of signed multiplication circuit.

```
module MULT32(HI, LO, A, B);
// output list
output [31:0] HI;
output [31:0] LO;
// input list
input [31:0] A;
input [31:0] B;
wire xor_result;

wire [31:0] A_2S;
wire [31:0] B_2S;

wire [31:0] result1;
wire [31:0] result2;
wire [31:0] result3_high;
wire [31:0] result3_low;

wire [31:0] result3_high_2S;
wire [31:0] result3_low_2S;
TWOSCOMP32 twos1(A_2S, A);
TWOSCOMP32 twos2(B_2S, B);

MUX32_2x1 mux32one(result1, A, A_2S, A[31]);
MUX32_2x1 mux32two(result2, B, B_2S, B[31]);

MULT32_U mult(result3_high, result3_low, result1, result2);

TWOSCOMP32 twos3(result3_high_2S, result3_high);
TWOSCOMP32 twos4(result3_low_2S, result3_low);

xor xor1(xor_result, A[31], B[31]);

MUX32_2x1 mux32three(HI, result3_high, result3_high_2S, xor_result);
MUX32_2x1 mux32four(LO, result3_low, result3_low_2S, xor_result);
endmodule
```

Figure 5.4 Implementation of signed multiplication

The steps are exactly the same as I mentioned above. Firstly, do the 2's complement for both number and select the positive one. After the multiplication, selected the result according to the xor result between the 31th bit of the multiplicand and multiplier.

To test the program, I design the testbench like the following figure 5.5.

```
module mult_tb;
reg signed [31:0] b = 'h12345678;
reg signed [31:0] c = 'h87654321;

wire signed [63:0] a = b * c;

reg [31:0] A;
reg [31:0] B;
wire [31:0] HIGH;
wire [31:0] LOW;

reg [31:0] C;
reg [31:0] D;
wire [31:0] HIGH_sign;
wire [31:0] LOW_sign;

MULT32_U mul32(.HI(HIGH), .LO(LOW), .A(A), .B(B));
MULT32 mul32two(.HI(HIGH_sign), .LO(LOW_sign), .A(C), .B(D));

initial
begin
A = 'h12345678; B = 'h87654321;
#5 A = 'hffffffff; B = 'hffffffff;
#5 C = 'hffffffff; D = 'hffffffff;
#5 C = 'h12345678; D = 'h87654321;
#5 C = 'h12345678; D = 'h12345678;
#5 C = 'h0000000a; D = 'h00000009;
#5 ;
end

endmodule
```

Figure 5.5 Multiplication testbench

Different values will be used for unsigned and signed multiplication.
The following figure 5.6 is the waveform which show the result.

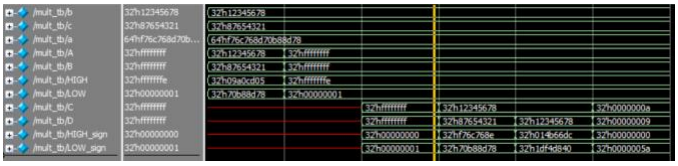


Figure 5.6 Waveform of multiplication testbench

A = 'h12345678; B = 'h87654321;
Result: HI:32'h09a0cd05 LO:32'h70b88d78
A = 'hffffffff; B = 'hffffffff;
Result: HI:32'hfffffffe LO: 32'h00000001
C = 'hffffffff; D = 'hffffffff;
Result: HI_signed: 32'h00000000
LW_signed: 32'h00000001
C = 'h12345678; D = 'h87654321;
Result:
HI_signed: 32'hf76c768e
LO_signed: 32'h70b88d78
C = 'h12345678; D = 'h12345678;
Result:
HI_signed: 32'h012b66dc
LO_signed: 32'h1df4d840
C = 'h0000000a; D = 'h00000009;
HI_signed: 32'h00000000
LO_signed: 32'h0000005a

6. Barrel Shifter

The following figure 6.1 is the prototype of the barrel shifter. The input is a 32-bit number D, a 5-bit shift amount, and a LnR control signal. The output is a 32-bit number after shifting.

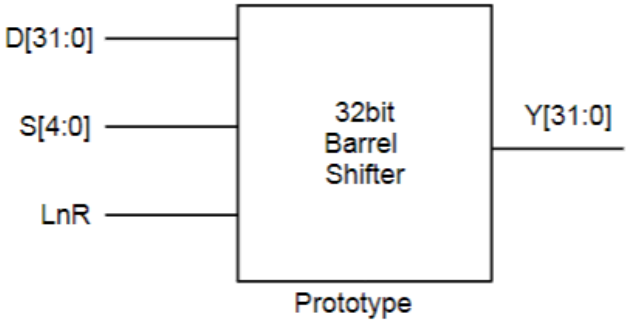


Figure 6.1 Barrel Shifter Prototype

The following figure is a 4-bit barrel shifter. The input is a 4-bit number and a 2-bit shift amount. It is can be extended to a 32-bit barrel shifter. Then there will be 5 columns total. The first column will shift one bit if S0 is 1, then 2, 4, 8, 16.

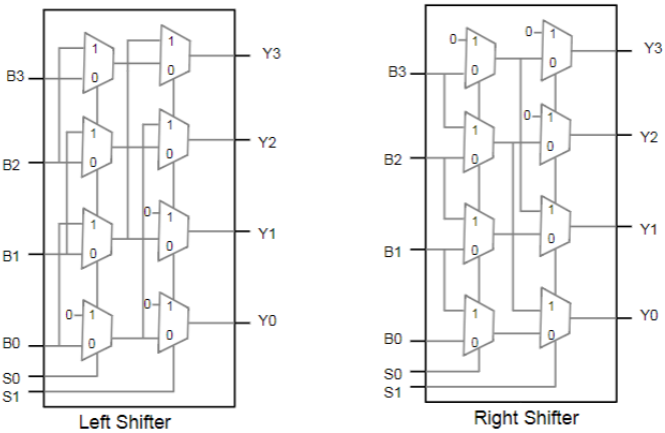


Figure 6.2 4-bit left and right barrel shifter

The following figure is the overview of the 32-bit barrel shifter. It will do both the right and left shift. Then the Y will be selected by LnR signal using a 32-bit 2x1 mux and the S will 32bits in reality.

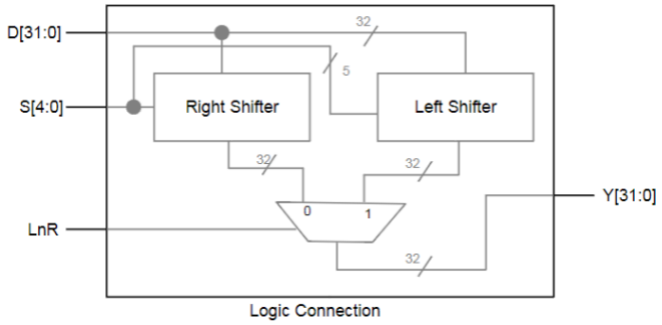


Figure 6.3 Overview of 32-bit barrel shifter

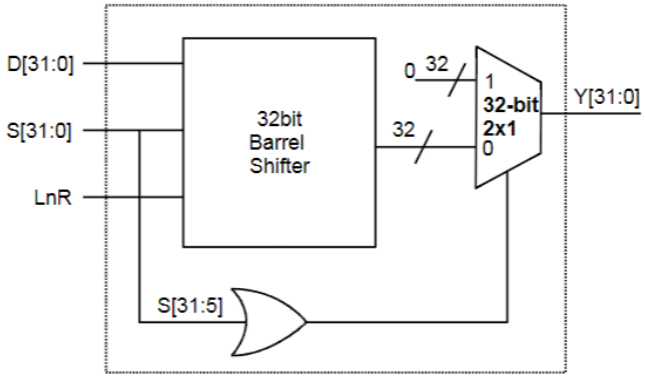


Figure 6.3 Overview of 32-bit barrel shifter*

The following figure 6.4, 6.5 6.6 and 6.7 are the implementation of 32-bit barrel shifter.

```
module BARREL_SHIFTER32(Y,D,S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;
input LnR;

wire [31:0] toRight;
wire [31:0] toLeft;
SHIFT32_R shift_right(toRight, D, S);
SHIFT32_L shift_left(toLeft, D, S);

MUX32_2x1 mux32_2xlone(Y, toLeft, toRight, LnR);

endmodule
```

Figure 6.4 Select 0 or shifting result

In the above figure, the instantiation of BARREL_SSHIFTER32 is created at the beginning, it will return the result after shifting. Since S is 32bits, the result will be 0 if there is a 1 in S[31:5] like the figure 6.3* above. It happens because all bits are shifted out. Finally, the result will be chosen between 0 and shift_out using 32bits_2x1 mux.

```
module BARREL_SHIFTER32(Y,D,S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;
input LnR;

wire [31:0] toRight;
wire [31:0] toLeft;
SHIFT32_R shift_right(toRight, D, S);
SHIFT32_L shift_left(toLeft, D, S);

MUX32_2x1 mux32_2xlone(Y, toLeft, toRight, LnR);

endmodule
```

Figure 6.5 Decision of shifting right or left

In the figure 6.5, it will do both of shifting left and right, and then select the result according the LnR signal using a 32bits_2x1 mux.

```
module SHIFT32_R(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

wire [31:0] out [5:0];

genvar i, j, m, k;
generate
        for (i = 0; i <= 31; i = i + 1)
        begin : first_loop
                if(i <= 30)
                begin
                  MUX1_2x1 mux_2xlone(.Y(out[0][i]), .I1(D[i + 1]), .I0(D[i]), .S(S[0]));
                end
                else
                begin
                  MUX1_2x1 mux_2xltwo(.Y(out[0][i]), .I1(1'b0), .I0(D[i]), .S(S[0]));
                end
        end
        for (m = 0; m <= 2; m = m + 1)
        begin : inner_loop
                for (k = 0; k <= 31; k = k + 1)
                begin : inner_inner_loop
                  if(k < 32 - 2**(m + 1))
                  begin
                    MUX1_2x1 mux_2xlthree(.Y(out[m + 1][k]), .I1(out[m][k + 2**(m + 1)]), .I0(out[m][k]), .S(S[m + 1]));
                  end
                  else
                  begin
                    MUX1_2x1 mux_2xlfour(.Y(out[m + 1][k]), .I1(1'b0), .I0(out[m][k]), .S(S[m + 1]));
                  end
                end
        end

        for (j = 0; j <= 31; j = j + 1)
        begin : last_loop
                if(j < 16)
                begin
                  MUX1_2x1 mux_2xlfive(.Y(Y[j]), .I1(out[3][j + 16]), .I0(out[3][j]), .S(S[4]));
                end
                else
                begin
                  MUX1_2x1 mux_2xlsix(.Y(Y[j]), .I1(1'b0), .I0(out[3][j]), .S(S[4]));
                end
        end
endgenerate

endmodule
```

Figure 6.6 Right Shifter

The above figure is the implementation of the right shifter. There will be three loops in the process of shifting. The first loop will do the shifting for the first column in the shifter diagram. If s[0] is 1, it will import a zero for the first mux; otherwise, it will set the current bit i value to bit i + 1 value. All bits after this column shifting will be stored in the out[0][i] one be one. The second loop is to iterate the three columns in the diagram. For each column, the shifting bit will be 2^1, 2^2 and 2^3. If S[i] is 1 it will be do the same operation as the first loop; otherwise, it will keep same and move to the next

column. The operation will be done by mux1_2x1 which select bit one by one. The last loop will be output loop. If S[i] is 1, it will import 16 zeros; otherwise, it will keep same. Then the result will be the output.

```
module SHIFT32_LR(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

wire [31:0] out [5:0];

genvar i, j, m, k;
generate
        for (i = 0; i <= 31; i = i + 1)
        begin : first_loop
                if(i >= 1)
                begin
                  MUX1_2x1 mux_2x1one(.Y(out[0][i]), .I1(D[i - 1]), .I0(D[i]), .S(S[0]));
                end
                else
                begin
                  MUX1_2x1 mux_2x1two(.Y(out[0][i]), .I1(1'b0), .I0(D[i]), .S(S[0]));
                end
        end
        for (m = 0; m <= 2; m = m + 1)
        begin : inner_loop
                for (k = 0; k <= 31; k = k + 1)
                begin : inner_inner_loop
                  if(k >= 2**(m + 1))
                  begin
                    MUX1_2x1 mux_2x1three(.Y(out[m + 1][k]), .I1(out[m][k - 2**(m + 1)]), .I0(out[m][k]), .S(S[m + 1]));
                  end
                  else
                  begin
                    MUX1_2x1 mux_2x1four(.Y(out[m + 1][k]), .I1(1'b0), .I0(out[m][k]), .S(S[m + 1]));
                  end
                end
        end

        for (j = 0; j <= 31; j = j + 1)
        begin : last_loop
                if(j >= 16)
                begin
                  MUX1_2x1 mux_2x1five(.Y(Y[j]), .I1(out[3][j - 16]), .I0(out[3][j]), .S(S[4]));
                end
                else
                begin
                  MUX1_2x1 mux_2x1six(.Y(Y[j]), .I1(1'b0), .I0(out[3][j]), .S(S[4]));
                end
        end

endgenerate

endmodule
```

Figure 6.7 Left Shifter

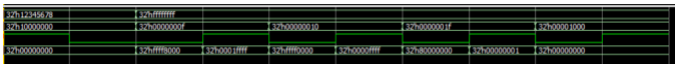
The figure 6.7 is the implementation of left shifter. it will do the same operation except for direction with the right shifter.

To the 32 bits barrel shifter, the implementation is at the following figure 6.8. It will test the shifter with different number, shift amount and LnR signal.

```
module barrel_shifter_tb;


reg [31:0] D;
reg [31:0] S;
reg LnR;
wire [31:0] Y;

SHIFT32 shifter(Y,D,S, LnR);

initial
begin
D = 'h12345678; S = 'h10000000; LnR = 1;
#5 D = 'h12345678; S = 'h10000000; LnR = 0;
#5 D = 'hffffffff; S = 'h0000000f; LnR = 0;
#5 D = 'hffffffff; S = 'h0000000f; LnR = 1;
#5 D = 'hffffffff; S = 'h00000010; LnR = 0;
#5 D = 'hffffffff; S = 'h00000010; LnR = 1;
#5 D = 'hffffffff; S = 'h0000001f; LnR = 0;
#5 D = 'hffffffff; S = 'h0000001f; LnR = 1;
#5 D = 'hffffffff; S = 'h00001000; LnR = 0;
#5 D = 'hffffffff; S = 'h00001000; LnR = 1;
#5 ;
end

endmodule
```

Figure 6.8 32 bits barrel shifter testbench

The following figure 6.9 will be the waveform of the result.



D = 'h12345678; S = 'h10000000; LnR = 1;
Result: 32'h0
D = 'h12345678; S = 'h10000000; LnR = 0;
Result: 32'h0
D = 'hffffffff; S = 'h0000000f; LnR = 0;
Result: 32'hffff8000
D = 'hffffffff; S = 'h0000000f; LnR = 1;
Result: 32'h0001ffff
D = 'hffffffff; S = 'h00000010; LnR = 0;
Result: 32'hffff0000
D = 'hffffffff; S = 'h00000010; LnR = 1;
Result: 32'h0000ffff
D = 'hffffffff; S = 'h0000001f; LnR = 0;
Result: 32'h80000000
D = 'hffffffff; S = 'h0000001f; LnR = 1;
Result: 32'h00000001
D = 'hffffffff; S = 'h00001000; LnR = 0;
Result: 32'h0
D = 'hffffffff; S = 'h00001000; LnR = 1;
Result: 32'h0

7. logic

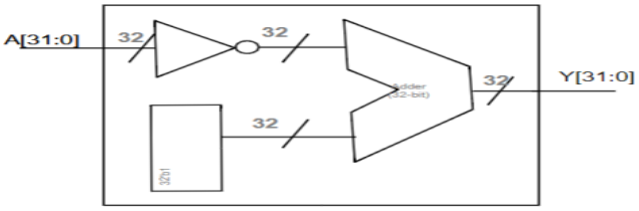


Figure 7.1 32-bit two's complement

The above figure is the interface of converting a number to its two's complement. It has a 32-bit input, the inverter which negate the number. After that, use the 32-bit adder to the addiction between the negative result and 1. The result will be output Y.

```
module TWOSCOMP32(Y,A);
//output list
output [31:0] Y;
//input list
input [31:0] A;

wire [31:0] not_result;
INV32_1x1 not1(not_result, A);
wire [31:0] one = 32'b1;
wire useless;

RC_ADD_SUB_32 add(Y, useless, one, not_result, 1'b0);

endmodule
```

Figure 7.2 Implementation of 32-bit two's complement

The figure is the implementation of 32-bit two's complement. The INV32_1x1 will invert all bits and get the not_result. After that, the 32-bit adder is to calculate the two's complement.

```
module TWOSCOMP64(Y,A);
//output list
output [63:0] Y;
//input list
input [63:0] A;

wire [63:0] not_result;
INV32_1x1 not1(not_result[31:0], A[31:0]);
INV32_1x1 not2(not_result[63:32], A[63:32]);
wire [63:0] one = 63'b1;
wire useless;

RC_ADD_SUB_64 sub_add(Y, useless, one, not_result, 1'b0);

endmodule
```

Figure 7.3 Implementation of 64-bit two's complement

The 64-bit two's complement is similar with the 32-bit one. It uses two 32-bit inverter to invert all 64 bits and add 1 to the result to get the output Y.

It is time to start to build the register file. The first digital component is SR latch.

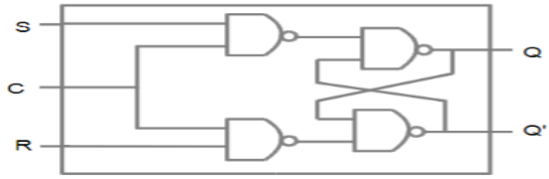


Figure 7.4 SR latch

The SR latch has three inputs S, R, and C. C is to control the data flow. It has four NAND gate and two outputs Q and Qbar.

```
module SR_LATCH(Q,Qbar, S, R, C, nP, nR);
input S, R, C;
input nP, nR;
output Q,Qbar;

wire subresult1;
wire subresult2;

nand n1(subresult1, C, S);
nand n2(subresult2, C, R);

nand n3(Q, subresult1, nP, Qbar);
nand n4(Qbar, subresult2, nR, Q);

endmodule
```

Figure 7.5 Implementation of SR latch

To implement the SR latch, it firstly use two NAND gate to get the subresults and then make the Qbar as one of the inputs for the upper NAND gate and Q as one of the inputs for the lower NAND gate in the diagram. If C is 0, the data will not change. If C is 1 and S and R are not 1 at the same time. The data will be determined by S and R.

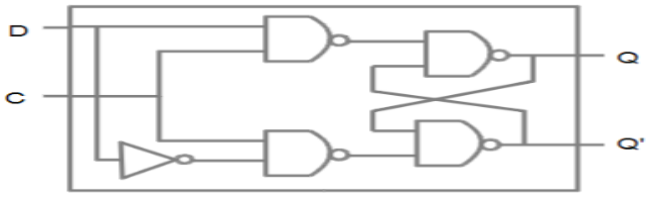


Figure 7.6 D latch

```
module D_LATCH(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

wire subresult1;
wire subresult2;
wire notD;

not not1(notD, D);

nand nand1(subresult1, C, D);
nand nand2(subresult2, notD, C);

nand nand3(Q, subresult1, nP, Qbar);
nand nand4(Qbar, subresult2, nR, Q);

endmodule
```

Figure 7.7 Implementation of D latch

The only difference between SR latch and D latch is the input S and R. D latch use one D input and negative D to control data conflict. It means "two inputs" will not be both of zero or one.

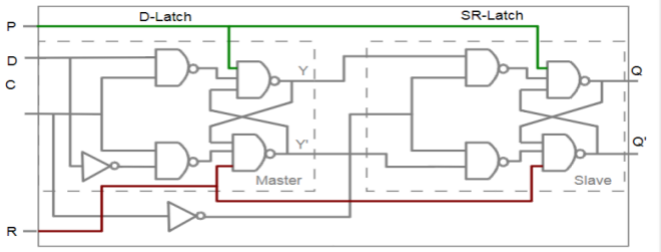


Figure 7.8 D flip flop

The bit flip flop on positive edges has one D latch and one SR latch. The SR is connected with D latch. The output of SR latch will be the inputs of D latch. Besides, it will D-latch signal and SR-latch signal to control the output of one-bit flip flop.

```
module D_FF(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;
wire sub_q;
wire sub_qbar;
wire notC;

not not1(notC, C);

D_LATCH d_latch(sub_q, sub_qbar, D, C, nP, nR);

SR_LATCH sr_latch(Q,Qbar, sub_q, sub_qbar, notC, nP, nR);

endmodule
```

Figure 7.9 Implementation of D flip flop

The figure 7.9 is the implementation of D flip flop. The sub_q. sub_qbar and negative C will be the output of SR latch.

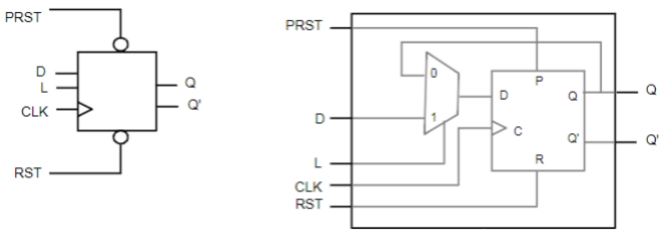


Figure 7.10 1-bit register

The register consists of 1 D flip flop and one 1-bit 2x1 mux The input D will be chosen from Q and D depending on the input L (keep data or change data).

```
module REG1(Q, Qbar, D, L, C, nP, nR);
input D, C, L;
input nP, nR;
output Q,Qbar;
wire Y;

MUX1_2x1 mux(Y, Q, D, L);

D_FF ff(Q, Qbar, Y, C, nP, nR);

endmodule
```

Figure 7.11 Implementation of 1-bit register

To implement the 1-bit register, select the input using 1-bit 2x1 mux firstly, and then implement the D flip flop to get the Q and Qbar (Qbar will be useless).

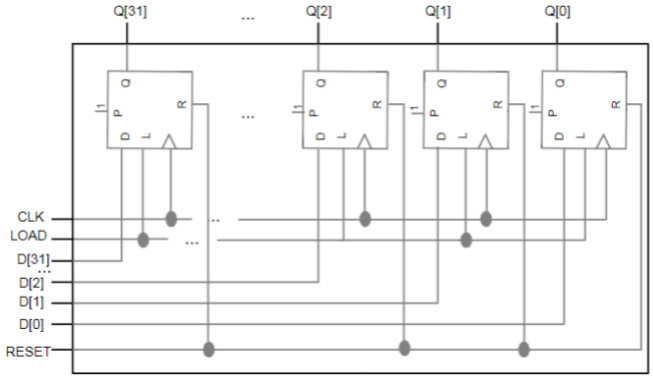


Figure 7.12 32-bit register

The 32-bit register consist 32 one-bit registers. The preset set signal will be 1 all time. Besides that, everything will be the same with the 1-bit register.

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

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

wire [31:0] useless;

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

endmodule
```

Figure 7.13 Implementation of 32-bit register

To implement the 32-bit register, the context will be in the loop in the *generate* and calculated one bit by one bit.

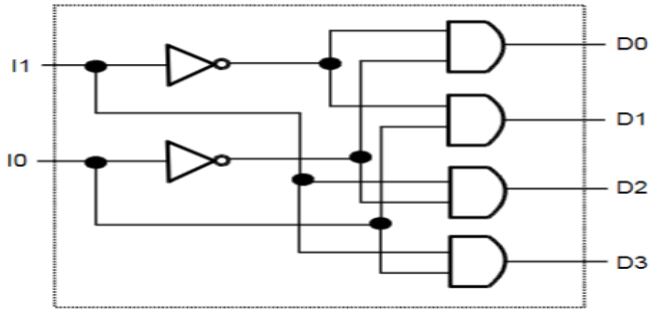


Figure 7.14 2-to-4 lines decoder

The 2-to-4 line decoder will convert to input to four output. It will use inverter for each input and do the combination between them. 2x2=4 will be outputs.

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

wire not0;
wire not1;

not NOT0(not0, I[0]);
not NOT1(not1, I[1]);

and and1(D[0], not0, not1);
and and2(D[1], not1, I[0]);
and and3(D[2], not0, I[1]);
and and4(D[3], I[0], I[1]);

endmodule
```

Figure 7.15 Implementation of 2-to-4 line decoder

As mentioned above, the implementation will use two NOT gates firstly, and then use four AND gates to do the combination between them.

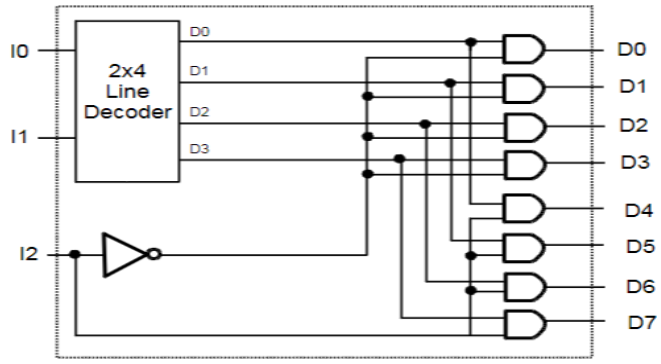


Figure 7.16 3-to-8 line decoder

The 3-to-8 line decoder will use one 2x4 line decoder and 1 inverter. It has three inputs and eight outputs. The 2x4 decoder will have four outputs. Then doing the combination with I2 and NOT I2 will produce 8 outputs.

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

wire [3:0] decoder2x4;
wire notI2;

not not1(notI2, I[2]);
DECODER_2x4 decode(decoder2x4, I[1:0]);

genvar i;
generate
for(i = 0; i < 8; i = i + 1)
begin : loop_3x8
  if(i < 4)
  begin
    and and1(D[i], decoder2x4[i], notI2);
  end
  else
  begin
    and and2(D[i], decoder2x4[i - 4], I[2]);
  end
end
endgenerate

endmodule
```

Figure 7.17 Implementation of 3-to-8 decoder

As mentioned above, the implementation will use NOT operation for I2 and then apply 2x4 decoder for I[1:0]. To implement the rest of the system, it will use a loop. From D0 to D3, the outputs from 2x4 will do combination with I2. For the rest of outputs, it will do the combination with NOT I2.

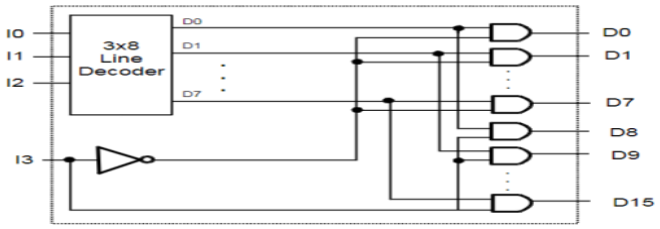


Figure 7.18 4-to-16 line decoder

The 4x16 line decoder will use one 3x8 line decoder and 1 inverter. It has four inputs and sixteen outputs. The 3x8 decoder will have eight outputs. Then doing the combination with I3 and NOT I3 will produce 16 outputs.

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

wire [7:0] decoder3x8;
wire notI3;

DECODER_3x8 decode(decoder3x8, I[2:0]);
not not1(notI3, I[3]);

genvar i;
generate
for(i = 0; i < 16; i = i + 1)
begin : loop_4x16
  if(i < 8)
  begin
    and and1(D[i], decoder3x8[i], notI3);
  end
  else
  begin
    and and2(D[i], decoder3x8[i - 8], I[3]);
  end
end
endgenerate

endmodule
```

Figure 7.19 Implementation of 4-to-16 line decoder

As mentioned above, the implementation will use NOT operation for I3 and then apply 3x8 decoder for I[2:0]. To implement the rest of the system, it will use a loop. From D0 to D7, the outputs from 3x8 will do combination with I3. For the rest of outputs, it will do the combination with NOT I3.

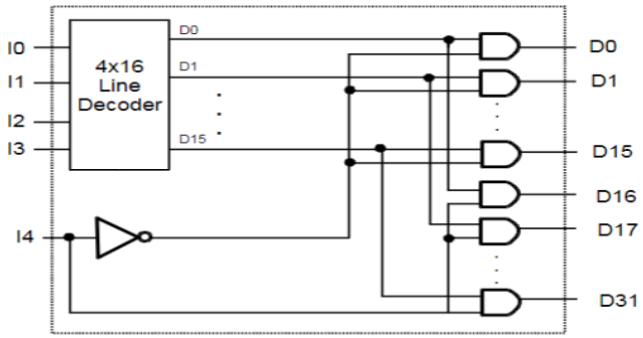


Figure 7.20 5-to-32 line decoder

The 5x32 line decoder will use one 4x16 line decoder and 1 inverter. It has 5 inputs and 32 outputs. The 4x16 decoder will have 16 outputs. Then doing the combination with I4 and NOT I4 will produce 32 outputs.

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

wire [15:0] decoder4x16;
wire notI4;

DECODER_4x16 decode(decoder4x16, I[3:0]);
not not1(notI4, I[4]);

genvar i;
generate
for(i = 0; i < 32; i = i + 1)
begin : loop_5x32
  if(i < 16)
  begin
    and and1(D[i], decoder4x16[i], notI4);
  end
  else
  begin
    and and2(D[i], decoder4x16[i - 16], I[4]);
  end
end
endgenerate

endmodule
```

Figure 7.21 Implementation of 5-to-32 bit decoder

As mentioned above, the implementation will use NOT operation for I4 and then apply 4x16 decoder for I[3:0]. To implement the rest of the system, it will use a loop. From D0 to D15, the outputs from 4x16 will do combination with I4. For the rest of outputs, it will do the combination with NOT I4.
The decoder and flip flop will be tested after building the register file.

8. logic_32bits

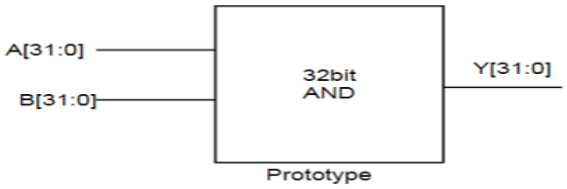


Figure 8.1 32-bit AND

The figure 8.1 32-bit AND will be implemented by 32 AND gates which have logic connection. It has two inputs A and B. The result will be evaluated one bit by one bit through 2x1 AND. The following figure 8.2 is the implementation of 32-bit AND.

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

genvar i;
generate
for(i = 0; i < 32; i = i + 1)
begin : and_loop
  and inst(Y[i], A[i], B[i]);
end
endgenerate

endmodule
```

Figure 8.2 Implementation of 32-bit AND

It uses the loop in the generate keyword and set the output one by one through 2x1 AND gate.

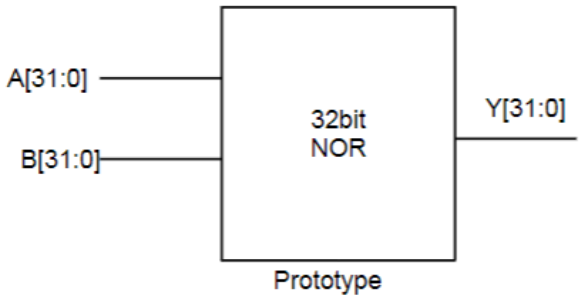


Figure 8.3 32-bit OR

The figure 8.3 32-bit NOR will be implemented by 32 NOR gates which have logic connection. It has two inputs A and B. The result will be evaluated one bit by one bit through 2x1 NOR. The following figure 8.4 is the implementation of 32-bit NOR.

```
module NOR32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

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

endmodule
```

Figure 8.4 Implementation of 32-bit NOR

It uses the loop in the generate keyword and set the output one by one through 2x1 NOR gate.

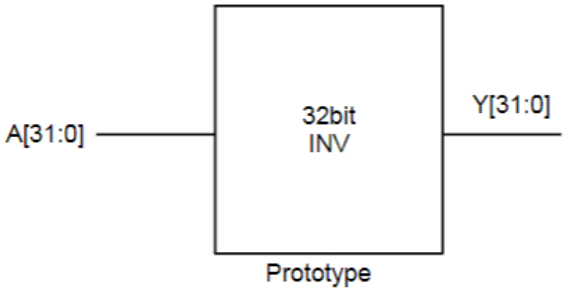


Figure 8.5 32-bit inverter

The figure 8.5 32-bit NOR will be implemented by 32 INV gates which have logic connection. It has one input A. The result will be evaluated one bit by one bit through 1x1 INV. The following figure 8.6 is the implementation of 32-bit INV.

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

genvar i;
generate
for(i = 0; i < 32; i = i + 1)
begin : not_loop
  not inst(Y[i], A[i]);
end
endgenerate

endmodule
```

Figure 8.6 Implementation of 32-bit inverter

It uses the loop in the generate keyword and set the output one by one through 2x1 INV gate.

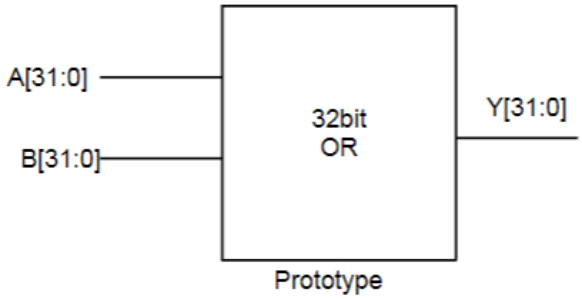


Figure 8.7 32-bit OR

The figure 8.7 32-bit OR will be implemented by 32-bit AND and 32-bit INV. It has two inputs A and B. The following figure 8.8 is the implementation of 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] nor_result;

NOR32_2x1(nor_result, A, B);
INV32_1x1(Y, nor_result);

endmodule
```

Figure 8.8 Implementation of 32-bit OR

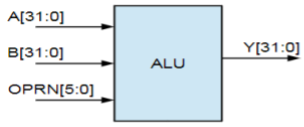
In the implementation, the nor_result will be gotten first through the NOR32_2x1. After, invert the nor_result and get the output Y.

```
module buf32(Output, A);
 input [31:0] A;
 output [31:0] Output;

 genvar i;
generate
 for(i = 0; i < 32; i = i + 1)
begin : buf_loop
   buf inst(Output[i], A[i]);
end
endgenerate
 endmodule

module xor32(Output, A, B);
 input [31:0] A;
 input [31:0] B;
 output [31:0] Output;

 genvar i;
generate
 for(i = 0; i < 32; i = i + 1)
begin : xor_loop
   xor inst(Output[i], A[i], B[i]);
end
endgenerate
 endmodule
```

Figure 8.9 Buf32 and XOR32

The above figure 8.9 are the other two 32-bit gates which are buf32 and XOR32. They will be used in the ALU and the operation of multiplication. They have the same logic with AND32. The only difference is gate.

9. ALU



TYPE	OPRN CODE
add	xx0001
sub	xx0010
mul	xx0011
Shift R	xx0100
Shift L	xx0101
and	xx0110
or	xx0111
nor	xx1000
slt	xx1001

Figure 9.1 Interface of ALU

The figure 9.1 shows the interface of ALU. It has two 32 operands A and B, one operation code for different operation which is 6 bits, and one 32-bit output.

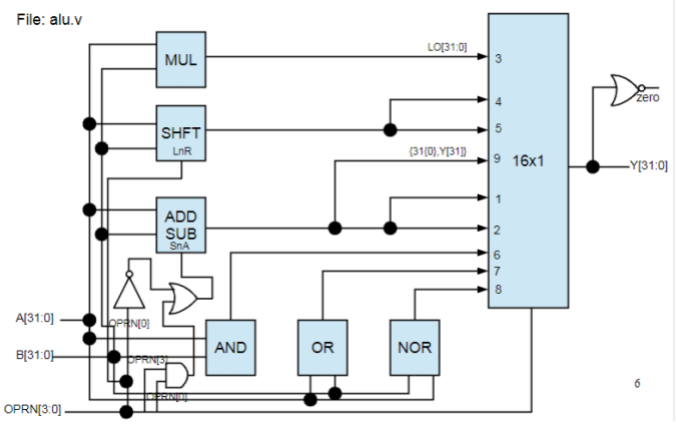


Figure 9.2 Digital circuit of ALU

The figure 9.2 is the digital circuit. It uses all electronic devices that we designed at the above sections. It will do different operations according to the oprn. Finally, it will select the final result through the 32-bit 16x1 mux. The outputs have one 32-bit result and one ZERO flag.

```
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] addSubWire;
        wire [31:0] shftWire;
        wire [31:0] mulWire;
        wire [31:0] andWire;
        wire [31:0] orWire;
        wire [31:0] norWire;
        // TBD
        wire [31:0] nullWire;
        wire [31:0] muxWire;
        wire lnr;
        wire andADDWire;
        wire orADDWire;

        and andADD(andADDWire, OPRN[0], OPRN[3]);
//Y, CO, A, B, SnA
        RC_ADD_SUB_32 addSub(.Y(addSubWire), .CO(nullWire[0]), .A(OP1), .B(OP2), .SnA(OPRN[1]));//R=A+B
```

```
        MULT32 mult_32Bit_Signed(.HI(nullWire), .LO(mulWire), .A(OP1), .B(OP2));

        not lnrNOT(lnr, OPRN[0]);
        SHIFT32 bl(shftWire, OP1, OP2, lnr);

        NOR32_2x1 nl(.Y(norWire), .A(OP1), .B(OP2));

        OR32_2x1 ol(.Y(orWire), .A(OP1), .B(OP2));

        AND32_2x1 al(.Y(andWire), .A(OP1), .B(OP2));

        MUX32_16x1 ml(muxWire, addSubWire, addSubWire, addSubWire, mulWire,
                shftWire, shftWire, andWire, orWire,
                norWire, addSubWire, addSubWire, addSubWire,
                addSubWire, addSubWire, addSubWire,
                addSubWire, OPRN[3:0]);

        nor31x1 or1(ZERO, muxWire);
        buf32 bbb(OUT, muxWire);
endmodule
```

Figure 9.3 Implementation of ALU

The implementation of ALU is very straightforward. It will create instantiation of different module and do the all calculation once the inputs or oprn is changed. The different results will be selected by 32-bit 16x1 mux according OPRN[3:0]. Finally, the ZERO flag will be set and the final result to be chosen using nor31x1 and buf32 modules.

```
// test 15 + 3 = 18
#5  op1_reg=15;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 14 - 5 = 9
#5  op1_reg=14;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 15 - 15 = 0
#5  op1_reg=15;
    op2_reg=15;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 10 * 9 = 90
#5  op1_reg=10;
    op2_reg=9;
    oprn_reg=`ALU_OPRN_WIDTH'h03;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 16 >> 2 = 4
#5  op1_reg=16;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h04;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 7 << 3 = 56
#5  op1_reg=7;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h05;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 3 & 5 = 1
#5  op1_reg=3;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h06;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 10 | 6 = 12
#5  op1_reg=10;
    op2_reg=6;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test ~(1 | 4) = 4294967290
#5  op1_reg=1;
    op2_reg=4;
    oprn_reg=`ALU_OPRN_WIDTH'h08;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));

//test 6 > 5 = 0
#5  op1_reg=6;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h09;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net, zero));
```

Figure 9.3 Testbench for ALU*

To test the program, we use all the test cease as the above figure.

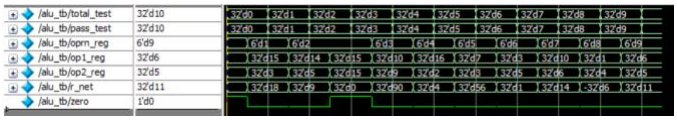


Figure 9.4 Waveform of ALU

The above figure 9.4 is the waveform of the ALU result.

```
# [TEST] 15 + 3 = 18 , got 18, zero_flag = 0... [PASSED]
# [TEST] 14 - 5 = 9 , got 9, zero_flag = 0... [PASSED]
# [TEST] 15 - 15 = 0 , got 0, zero_flag = 1... [PASSED]
# [TEST] 10 * 9 = 90 , got 90, zero_flag = 0... [PASSED]
# [TEST] 16 >> 2 = 4 , got 4, zero_flag = 0... [PASSED]
# [TEST] 7 << 3 = 56 , got 56, zero_flag = 0... [PASSED]
# [TEST] 3 & 5 = 1 , got 1, zero_flag = 0... [PASSED]
# [TEST] 10 | 6 = 14 , got 14, zero_flag = 0... [PASSED]
# [TEST] 1 ~| 4 = 4294967290 , got 4294967290, zero_flag = 0... [PASSED]
# [TEST] 6 < 5 = 0 , got 0, zero_flag = 0... [PASSED]
#
#       Total number of tests          10
#       Total number of pass           10
#
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/alu_tb.v(127)
#    Time: 105 ns  Iteration: 0  Instance: /alu_tb
```

Figure 9.5 Transcript output of ALU

The above figure 9.5 is the text output when the testbench is simulated. All test cases work fine, so the ALU is functioning correctly.

10. Register file

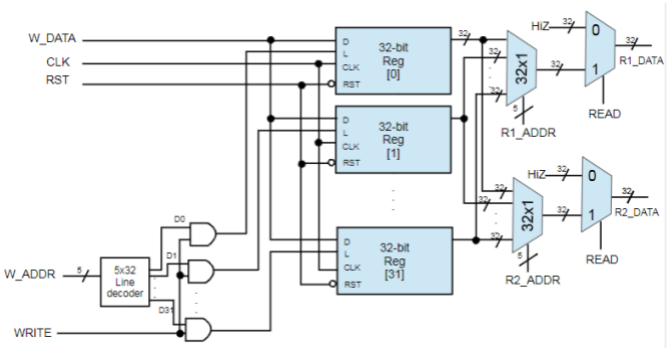


Figure 10.1 Digital circuit of 32x32 register file

The register file 5 inputs which W_data, CLK, RST, W_ADDR, WRITE. W_data is the data which is to be written to one of the registers depending on the write signal and W_ADDR. The R1_DATA and R2_DATA will be read depending on the READ signal and R1_ADDR and R2_ADDR which are used to select the register output using 32-bit 32x1 mux. If READ signal is 1 and WRITE signal is 0, the register file will be in READ mode. If READ signal is 0 and WRITE signal is 1, the register file will be in WRITE mode. Otherwise, it will be in High Z.

```
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;

wire [31:0] decoder5x32;
wire [31:0] load5x32;
wire [31:0] reg32x32 [31:0];

wire [31:0] X;
wire [31:0] Y;

DECODER_5x32 decoder(decoder5x32, ADDR_W);
```

```
genvar i, j;
generate
for(i = 0; i < 32; i = i + 1)
begin : load_loop
  and and1(load5x32[i], WRITE, decoder5x32[i]);
end

for(j = 0; j < 32; j = j + 1)
begin : reg_loop
  REG32 reg32(reg32x32[j], DATA_W, load5x32[j], CLK, RST);
end
endgenerate

MUX32_32x1 mux1(X, reg32x32[0], reg32x32[1], reg32x32[2], reg32x32[3], reg32x32[4], reg32x32[5], reg32x32[6], reg32x32[7],
                   reg32x32[8], reg32x32[9], reg32x32[10], reg32x32[11], reg32x32[12], reg32x32[13], reg32x32[14], reg32x32[15],
                   reg32x32[16], reg32x32[17], reg32x32[18], reg32x32[19], reg32x32[20], reg32x32[21], reg32x32[22], reg32x32[23],
                   reg32x32[24], reg32x32[25], reg32x32[26], reg32x32[27], reg32x32[28], reg32x32[29], reg32x32[30], reg32x32[31], ADDR_R1);

MUX32_32x1 mux2(Y, reg32x32[0], reg32x32[1], reg32x32[2], reg32x32[3], reg32x32[4], reg32x32[5], reg32x32[6], reg32x32[7],
                   reg32x32[8], reg32x32[9], reg32x32[10], reg32x32[11], reg32x32[12], reg32x32[13], reg32x32[14], reg32x32[15],
                   reg32x32[16], reg32x32[17], reg32x32[18], reg32x32[19], reg32x32[20], reg32x32[21], reg32x32[22], reg32x32[23],
                   reg32x32[24], reg32x32[25], reg32x32[26], reg32x32[27], reg32x32[28], reg32x32[29], reg32x32[30], reg32x32[31], ADDR_R2);

MUX32_2x1 mux3(DATA_R1, { `DATA_WIDTH{1'bx}}, X, READ);
MUX32_2x1 mux4(DATA_R2, { `DATA_WIDTH{1'bx}}, Y, READ);
endmodule
```

Figure 10.2 Implementation of 32x32 register file

The above figure is to implement the 32x32 register file. It defines decoder5x32 to store all address, load5x32 to store the result after the AND operation. After that, it will select the register output from 32 registers using R1_ADDR, R2_ADDR and two 32-bit 32x1 mux. Finally, if READ signal, the output will be High Z; otherwise, the data will be read from registers.

The following part is to store the 32x32 register file. At the same time, it will test the component we built in the above sections.

```
// Register Instance
REGISTER_FILE_32x32 reg_inst(.DATA_R1(DATA_R1), .DATA_R2(DATA_R2), .ADDR_R1(ADDR_R1), .ADDR_R2(ADDR_R2),
                             .DATA_W(DATA_W), .ADDR_W(ADDR_W), .READ(READ), .WRITE(WRITE), .CLK(CLK), .RST(RST));

initial
begin
RST = 1'b1;
READ = 1'b0;
WRITE = 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 = 1;i < 10; i = i + 1)
begin
#10   DATA_W = i; READ = 1'b0; WRITE = 1'b1; ADDR_W = i;
end
```

Figure 10.3 Initialization block

At the beginning, we initialize the READ and WRITE signal to 0 and all tests and passed tests to 0. In the initialization block, data is written at the specific address by using a loop. Data is from one to ten and address is also from one to ten.

```
//read cycle
#10   READ=1'b0; WRITE=1'b0;
#5    no_of_test = no_of_test + 1;
      if (DATA_R1 !== {`DATA_WIDTH{1'bz}} && DATA_R2 !== {`DATA_WIDTH{1'bz}})
        $write("[TEST] Read %1b, Write %1b, expecting 32'hzzzzzzzz, got %8h [FAILED]\n", READ, WRITE, DATA_R1);
      else
        no_of_pass  = no_of_pass + 1;

//test write data from address R1
for(i = 1; i < 10; i = i + 1)
begin
#5      READ=1'b1; WRITE=1'b0; ADDR_R1 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R1 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R1);
        else
            no_of_pass  = no_of_pass + 1;
end
#20
//test write data from address R2
for(i = 1; i < 10; i = i + 1)
begin
#5      READ = 1'b1; WRITE = 1'b0; ADDR_R2 = i;
#5      no_of_test = no_of_test + 1;
        if (DATA_R2 !== i)
            $write("[TEST] Read %1b, Write %1b, expecting %8h, got %8h [FAILED]\n", READ, WRITE, i, DATA_R2);
        else
            no_of_pass  = no_of_pass + 1;
end

#10    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;
```

Figure 10.4 Tests block

The test cases (figure 10.4) include three parts. The first part is read cycle. It is to test the high Z when both READ and WRITE signal are zeros. After that, the first for loop is to test data which is written in the register from read address R1. The second loop is to test data which is written in the register from read address R2.

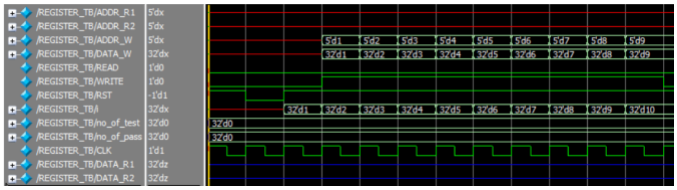


Figure 10.5 Waveform of initialization block

The figure 10.5 show the integer i is from 1 to 10. The write address and written data are from 1 to 9. It is the initialization block.

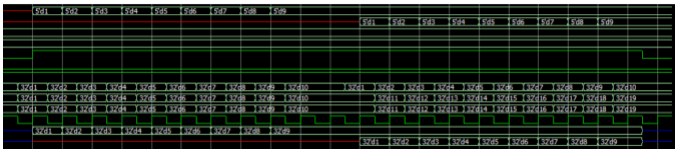


Figure 10.6 Waveform of the test block

In the figure 10.6, the blue line shows that the DATA_R1 and DATA_R2 are at high Z state which means the first part is passed. Then both of the ARRD_R1 and DATA_R1 are from 1 to 9. Integer i is from one to 10. Since the loop only run 9 times, the second part passed. After that, both of the ARRD_R2 and DATA_R are from 1 to 9. Integer i is from one to 10. Since the loop only run 9 times, the third part passed.

```
VSIM 3> run -all
#
#       Total number of tests          19
#       Total number of pass           19
#
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/register_tb.v(105)
#    Time: 345 ns  Iteration: 0  Instance: /REGISTER_TB
# Break in Module REGISTER_TB at C:/Users/pansh/Downloads/prj_02/register_tb.v line 105
```

Figure 10.7 Total tests and passed tests

The above figure 10.7 show total number of tests and total number of passed tests. There is one test for high Z in the if statement. In two for loops, total number of tests = 9 + 9 = 18. Thus, all tests are passed.

11. Data Path

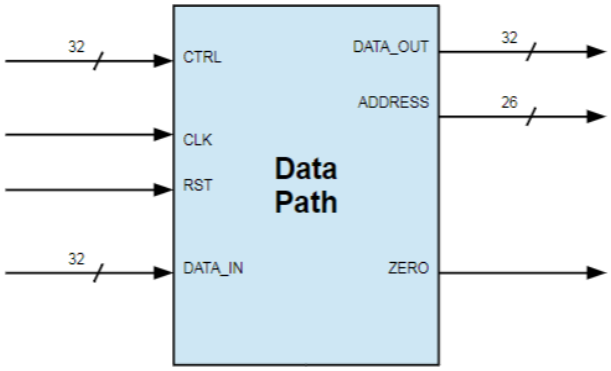


Figure 11.1 Prototype of Data Path

The data path has one 32-bit CTRL to control the data flow (some bits will not be used), CLK, RST, and 32-bit instruction to be analyzed. Besides, it has one 32-bit data output, one 26-bit ADDRESS, and one ZERO flag.

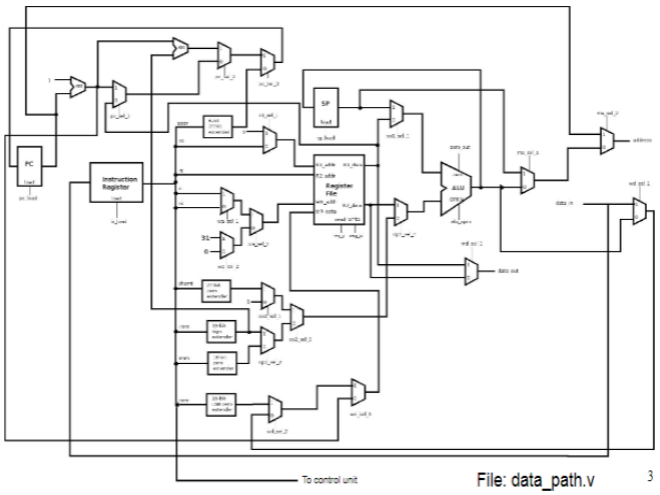


Figure 11.2 Complete Data Path

The complete data path shows the data flow of the one-line assembly code in computation (CS147DV instruction set). The data path contains everything besides memory file. It includes R-type, J-type and I-type instructions. The details will be explained at the following implementation.

```
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;

wire [31:0] pc_load, pc_sel_1, pc_sel_2, pc_sel_3;
wire [31:0] pc_subresult2, pc_subresult1;
wire [31:0] wa_sel_1, wa_sel_2, wa_sel_3;
wire [31:0] op2_sel_1, op2_sel_2, op2_sel_3, op2_sel_4;
wire [31:0] wd_sel_1, wd_sel_2, wd_sel_3;
wire [31:0] r1_sel_1;
wire [31:0] sp_load;
wire [31:0] R1_DATA, R2_DATA, alu_result;
wire [31:0] op1_sel_1;
wire [31:0] ma_sel_1;
wire [5:0] xxx;
wire CO, CO1;

        REG32_PC reg1(pc_load, pc_sel_3, CTRL[0], CLK, RST);

        RC_ADD_SUB_32 AnS1(pc_subresult1, CO, pc_load, 32'b1, 1'b0);
        RC_ADD_SUB_32 AnS2(pc_subresult2, CO1, pc_subresult1, {{16{INSTRUCTION[15]}}, INSTRUCTION[15:0]}, 1'b0);

        MUX32_2x1 mux1(pc_sel_1, R1_DATA, pc_subresult1, CTRL[1]);
        MUX32_2x1 mux2(pc_sel_2, pc_sel_1, pc_subresult2, CTRL[2]);
        MUX32_2x1 mux3(pc_sel_3, {{6'b0}, INSTRUCTION[25:0]}, pc_sel_2, CTRL[3]);

      REG32 ir_register(INSTRUCTION, DATA_IN, CTRL[4], CLK, RST);

      MUX32_2x1 mux4(wa_sel_1, {{27{1'b0}}, INSTRUCTION[15:11]}, {{27{1'b0}}, INSTRUCTION[20:16]}, CTRL[10]);
      MUX32_2x1 mux5(wa_sel_2, 32'b0, 32'd31, CTRL[11]);
      MUX32_2x1 mux6(wa_sel_3, wa_sel_2, wa_sel_1, CTRL[12]);

      MUX32_2x1 mux7(op2_sel_1, 32'b1, {{27'b0}, INSTRUCTION[10:6]}, CTRL[18]);
      MUX32_2x1 mux8(op2_sel_2, {{16'b0}, INSTRUCTION[15:0]}, {{16{INSTRUCTION[15]}}, INSTRUCTION[15:0]}, CTRL[19]);
      MUX32_2x1 mux9(op2_sel_3, op2_sel_2, op2_sel_1, CTRL[20]);
      MUX32_2x1 mux10(op2_sel_4, op2_sel_3, R2_DATA, CTRL[21]);

      MUX32_2x1 mux11(wd_sel_1, alu_result, DATA_IN, CTRL[13]);
      MUX32_2x1 mux12(wd_sel_2, wd_sel_1, {INSTRUCTION[15:0], {16'b0}}, CTRL[14]);
      MUX32_2x1 mux13(wd_sel_3, pc_subresult1, wd_sel_2, CTRL[15]);

      MUX32_2x1 mux14(r1_sel_1, {{27{1'b0}}, INSTRUCTION[25:21]}, 32'b0, CTRL[7]);

      REG32_SP sp_inst(sp_load, alu_result, CTRL[16], CLK, RST);

      REGISTER_FILE_32x32 reg_file(R1_DATA, R2_DATA, r1_sel_1[4:0], INSTRUCTION[20:16], wd_sel_3, wa_sel_3[4:0], CTRL[9], CTRL[8], CLK, RST);

      MUX32_2x1 mux15(op1_sel_1, R1_DATA, sp_load, CTRL[17]);

      ALU alu(alu_result, ZERO, op1_sel_1, op2_sel_4, CTRL[27:22]);

      MUX32_2x1 mux16(DATA_OUT, R2_DATA, R1_DATA, CTRL[30]);

      MUX32_2x1 mux17(ma_sel_1, alu_result, sp_load, CTRL[28]);
      MUX32_2x1 mux18({xxx, ADDR}, ma_sel_1, pc_load, CTRL[29]);

endmodule
```

Figure 10.4 Implementation of Data Path

At the beginning, all intermediate results will be created according to different signals for using in different operation. After, we start from the PC (program counter). By using the 32-bit PC register (it is similar with Reg32, and the only difference is that it checks the INST_START_ADDR and exchange the location of 1'b1 and RESET for one-bit register), it will get the output and it is stored in pc_load which is control by CTRL[0].

After that, two operations of the adder will be performed. The results will be used to select the next instruction. Then we start to select the next instruction. Mux1 is to choose the result between R1_DATA and pc_subresult1 which is the pc_load + 1, and the result is controlled by CTRL[1]. The result is stored in pc_sel_1. Mux2 is to select the result between pc_sel_1 and the result from second adder operation and it is control by CTRL[1]. Mux3 is to selected the next instruction between pc_sel_2 and

ZERO EXTENSION of INSTRUCTION[25:0] and it's controlled by CTRL[3].

Analyzing the data_in through the instruction register will be performed and it is controlled by CTRL[4]. Mux4, mux5 and mux6 is to select the WR_addr of the register file. Mux4 is to select the result from rt and rd and it is controlled by CTRL[10]. The result will be stored in wa_sel_1. Mux5 is to select the result from 0 and 31 by CTRL[11] and the result is stored in wa_sel_2. Mux6 is to select the WR_addr between wa_sel_1 and wa_sel_2 by CTRL[12].

After that, one operand of ALU will be selected. Mux7 is to select the result between 1 and 27-bit zero extension of shamt by CTRL[18]. The result is saved in op2_sel_1. Mux8 is to the result between 16-bit signed extension imm and 16-bit zero extension imm by CTRL[19]. The result is stored in op2_sel_2. Mux9 is to select a result between op2_sel1 and op2_sel2 by CTRL[20] and the result is stored in op2_sel_3. Mux10 is to select the one operand for ALU between R2_DATA and op2_sel_3 by CTRL[21].

After that, the WR_data is selected. Mux11 is to select wd_sel_1 between alu_result and data_in which will be written in the register file by CTRL[13]. Mux12 is to select wd_sel_2 between wd_sel_1 and 16-bit LSB zero extension imm. Mux13 is to select WR_data between pc_subresult1 and wd_sel_2 by CTRL[15].

At the following, mux14 is to select the R1_addr between rs and 0 by CTRL[7]. After that, stack pointer will be loaded through REG32_SP and alu_result by CTRL[16] and the result will be stored in sp_load. Mux15 is to select the other operand of ALU between op1_sel_1 and R1_data by CTRL[17]. Then ALU start the computation. Two operands are op1_sel_1 and op2_sel_1. Oprn is CTRL[27:22]. The output will be stored in alu_result and set zero flag.

After that, mux16 is to select DATA_OUT between R2_DATA and R1_DATA by CTRL[30]. Mux17 is to select ma_sel_1 between alu_result and sp_load by CTRL[28]. Finally, mux18 is to select ADDR between ma_sel1 and pc_load by CTRL[29].

11. Control Unit

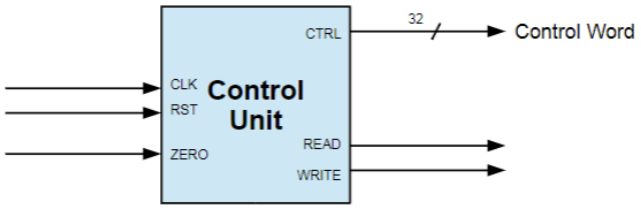


Figure 11.1 Prototype of Control Unit

The control unit has three inputs which are CLK, RST and ZERO from ALU. Besides, it will issue the READ and WRITE signal and a Control Word which will be used in the data path.

The control unit has five stages, and they are PROC_FETCH, PROC_DECODE, PROC_EXE, PROC_MEM and PROC_WB. Different operations will be performed at the different stages. The figure 11.2 is the implementation of the state machine.

```
input CLK, RST;
// list of outputs
output [2:0] STATE;
reg [2:0] state;
reg [2:0] next_state;

assign STATE = state;

initial
begin
  state = 3'bxx;
  next_state = `PROC_FETCH;
end

always @ (negedge RST)
begin
  state = 3'bxx;
  next_state = `PROC_FETCH;
end

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

end

endmodule;
```

Figure 11.2 State machine

Two internal registers will be used to represent the current state and next state of the state machine. At every positive edge of CLK, the next state will be assigned to the current state and the next state will has a new state according to the logic.

The followings will talk about the details of control unit.

```
module CONTROL_UNIT(CTRL, READ, WRITE, ZERO, INSTRUCTION, CLK, RST);
// Output signals
output [`CTRL_WIDTH_INDEX_LIMIT:0]  CTRL;
output READ, WRITE;

// input signals
input ZERO, CLK, RST;
input [`DATA_INDEX_LIMIT:0] INSTRUCTION;

reg [`CTRL_WIDTH_INDEX_LIMIT:0] CTRL;

wire [2:0] proc_state;
PROC_SM state_machine(.STATE(proc_state),.CLK(CLK),.RST(RST));

reg [5:0]   opcode;
reg [4:0]   rs;
reg [4:0]   rt;
reg [4:0]   rd;
reg [4:0]   shamt;
reg [5:0]   funct;
reg [15:0]  imm;
reg [25:0]  addr;

assign READ = CTRL[5]; // MEM_R
assign WRITE = CTRL[6]; // MEM_W
```

Figure 11.3 Setup of control unit

Several registers are created and will be used for the decomposition of INSTRUCTION. READ signal and WRITE signal of memory will be control by CTRL[5] and CTRL[6].

```
always @ (proc_state)
begin
    case(proc_state)
        `PROC_FETCH:
        begin
          CTRL = 32'b00100000000000000000000000100000; // 0010 0000 0000 0000 0000 0000 0010 0000
        end
        `PROC_DECODE:
        begin
          CTRL = 32'b00000000000000000000001000010000; // 0000 0000 0000 0000 0000 0010 0001 0000
        end
        `PROC_EXE:
```

Figure 11.4 Fetch and decode

In the fetch stage, the READ signal for memory and pc_load signal will be set to 1 for reading instruction from memory and issue the starting address through program counter. After that, it will be execution time of ALU.

```
`PROC_EXE:
begin
  print_instruction(INSTRUCTION);
  {opcode, rs, rt, rd, shamt, funct} = INSTRUCTION; //R type
  {opcode, rs, rt, imm} = INSTRUCTION; //I type
  {opcode, addr}= INSTRUCTION; // J type
```

Figure 11.5 Decompose

It will decompose the INSTRUCTION for different types of instruction (R, I, J).

```
case(opcode)
// R Type
6'h00 : begin
  case(funct)
      6'h20:
      //add
      begin
        CTRL = 32'b00000000011000000000001000000000; // 0000 0000 0110 0000 0000 0010 0000 0000
      end
      6'h22:
      //sub
      begin
        CTRL = 32'b00000000010100000000001000000000; // 0000 0000 1010 0000 0000 0010 0000 0000
      end
      6'h2c:
      //mul
      begin
        CTRL = 32'b00000000111000000000001000000000; // 0000 0000 1110 0000 0000 0010 0000 0000
      end
      6'h24:
      //and
      begin
        CTRL = 32'b00000000110100000000001000000000; // 0000 0001 1010 0000 0000 0010 0000 0000
      end
      6'h25:
      //or
      begin
        CTRL = 32'b00000000111100000000001000000000; // 0000 0001 1110 0000 0000 0010 0000 0000
      end
      6'h27:
      //nor
      begin
        CTRL = 32'b00000001000100000000001000000000; // 0000 0010 0010 0000 0000 0010 0000 0000
      end
      6'h2a:
      //slt
      begin
        CTRL = 32'b00000001001100000000001000000000; // 0000 0010 0110 0000 0000 0010 0000 0000
      end
      6'h02:
      //srl
      begin
        CTRL = 32'b00000000100010100000000001000000000; // 0000 0001 0001 0100 0000 0010 0000 0000
      end
      6'h01:
      //sll
      begin
        CTRL = 32'b00000000101010100000000001000000000; // 0000 0001 0101 0100 0000 0010 0000 0000
      end
      6'h08:
      //jr
      begin
        CTRL = 32'b00000000000000000000000001000000000; // 0000 0000 0000 0000 0000 0010 0000 0000
      end
  endcase
end
```

Figure 11.6 Execution of R type instructions

For different R type of instructions, there will be different signal. For all R type instrucftions, READ signal (CTRL[9]) will be set to 1 for register read. CTRL[27:22] is operation code.

add: CTRL[22] is set to 1 for add operation. CTRL[21] is set to one to get the data from register file for one operand of ALU. Addition happens between [rs] and [rt]. After that, for sub, mul, and, or, nor and slt operation, they will keep same besides except for different oprn code for ALU.

For slt and srl operation, the second operand will be from shamt. Thus, CTRL[18] and CTRL[20] will be set to 1 to get the 32-bit zero extension shamt. Of course, there will different oprn code for ALU. Finally, jr operation is not related to ALU, so only register READ signal is set to 1.

```
// I-type
6'h08 :
//addi
begin
  CTRL = 32'b00000000001001000000000001000000000; // 0000 0000 0100 1000 0000 0010 0000 0000
end
6'h1d :
//multi
begin
  CTRL = 32'b00000000011001000000000001000000000; // 0000 0000 1100 1000 0000 0010 0000 0000
end
6'h0c :
//andi
begin
  CTRL = 32'b00000000110000000000000001000000000; // 0000 0001 1000 0000 0000 0010 0000 0000
end
6'h0d :
//ori
begin
  CTRL = 32'b00000000111000000000000001000000000; // 0000 0001 1100 0000 0000 0010 0000 0000
end
6'h0f:
//lui
begin
  CTRL = 32'b00000000000000000000001011000000000; // 0000 0000 0000 0000 0001 0110 0000 0000
end
6'h0a :
//slti
begin
  CTRL = 32'b00000001001001000000000001000000000; // 0000 0010 0100 1000 0000 0010 0000 0000
end
6'h04 :
//beq
begin
  CTRL = 32'b00000000101000000000000001000000000; // 0000 0000 1010 0000 0000 0010 0000 0000
end
6'h05 :
//bne
begin
  CTRL = 32'b00000000101000000000001000000000; // 0000 0000 1010 0000 0000 0010 0000 0000
6'h23 :
//lw
begin
  CTRL = 32'b00000000001001000000001000000000; // 0000 0000 0100 1000 0000 0010 0000 0000
end
6'h2b :
//sw
begin
  CTRL = 32'b00000000001001000000000001000000000; // 0000 0000 0100 1000 0000 0010 0000 0000
end
6'h1b :
// push
begin
  CTRL = 32'b00010000010010010000000101000000000; // 0001 0000 1001 0010 0000 0010 1000 0000
end
6'h1c :
// pop
begin
  CTRL = 32'b00000000001010010000000000000000000; // 0000 0000 0101 0010 0000 0000 0000 0000
end
```

Figure 11.7 I type in execution stage

Register READ signal will be still 1. For addi, multi, andi, ori, slti, lw and sw operation, different oprn code will be set to 1 for different operations from CTRL[27:22]. CTRL[19] will be set to 1 for slti, addi, multi, lw and sw to the 16-bit signed extension imm, otherwise, it will be 16-bit zero extension imm. For lui operation, CTRL[10] and CTRL[12] will be set to 1 to load the rt to WR_addr. For beq and bne, CTRL[21] is set to 1 for getting the R2_data from the register file. For push operation, R1_addr will be set to 0 and the first operand will be data from R1_data. The second operand will be 1. For pop operation, CTRL[16] is set to 1 to get the value from stack pointer and second operand will be 1.

```
// J type
6'h02:
//jmp
begin
  CTRL = 32'b00000000000000000000000000000000;
end
6'h03:
//jal
begin
  CTRL = 32'b00000000000000000000101000000000; // 0000 0000 0000 0000 0000 1010 0000 0000
end
6'h1b:
// push
begin
  CTRL = 32'b00010000100100100000001010000000; // 0001 0000 1001 0010 0000 0010 1000 0000
end
6'h1c:
// pop
begin
  CTRL = 32'b00000000001011010000000100000000; // 0000 0000 0101 1010 0000 0010 0000 0000
end
```

Figure 11.8 J type instruction in execution stage

For jmp operation, all bits will be 0. For jal, WR_addr will be set to 31 by setting CTRL[11] to 1. For push operation, memory write signal, op1_sel_1, and ma_sel_1 will be set 1 to get the address. For pop operation, CTRL[17], CTRL[19] and CTRL[20] are set to 1 to issue the first operand and WR_data.

After that, it will be memory stage.

```
`PROC_MEM:
begin
  case(opcode)
    6'h1b :
    // push
    begin
      CTRL = 32'b01010000100100100000001011000000; // 0101 0000 1001 0010 0000 0010 1100 0000
    end
    6'h1c :
    // pop
    begin
      CTRL = 32'b00000000001010010001000000100000; // 0000 0000 0101 0010 0010 0000 0010 0000
    end
    6'h23 :
    // lw
    begin
      CTRL = 32'b00000000001001000000101100100000; // 0000 0000 0100 1000 0001 0110 0010 0000
    end
    6'h2b :
    // sw
    begin
      CTRL = 32'b00000000001001000000001001000000; // 0000 0000 0100 1000 0000 0010 0100 0000
    end
  endcase
```

Figure 11.9 Memory stage

For the push operation, memory signal will be set to one (CTRL[6]). R1_addr will be set to 0 (CTRL[7]). CTRL[17], CTRL[19] and CTRL[20] are set to 1 to issue the first operand and WR_data. Finally, CTRL[30] is set to one to get R1_data as data_out and CTRL[28] is set to 1 to get addr from stack pointer. For pop operation, memory read signal will be set to 1 to pop out the data from stack, data_in will be used to set the WR_data. CTRL[17] and CTRL[20] is set to 1 to set operands. For lw and sw operations, they are similar. Lw to load word from memory, and sw is to write data to the memory. They have the same op2 and oprn code. For lw, rt will be the WR_addr

After that, it will be write_back stage for R-type, I type and J type operation. All operations will set CTRL 0, 1, 3 to 1 for issuing next address pc + 1 except for jr.

```
// R type
6'h00 :
begin
  case(funct)
      6'h20:
      //Add
      begin
        CTRL = 32'b00000000011000001001001100001011; // 0000 0000 0110 0000 1001 0011 0000 1011
      end
      6'h22:
      //Sub
      begin
        CTRL = 32'b00000000101000001001001100001011; // 0000 0000 1010 0000 1001 0011 0000 1011
      end
      6'h2c:
      //mult
      begin
        CTRL = 32'b00000000111000001001001100001011; // 0000 0000 1110 0000 1001 0011 0000 1011
      end
      6'h24:
      //and
      begin
        CTRL = 32'b00000000110100001001001100001011; // 0000 0001 1010 0000 1001 0011 0000 1011
      end
      6'h25:
      //or
      begin
        CTRL = 32'b00000000111100001001001100001011; // 0000 0001 1110 0000 1001 0011 0000 1011
      end
      6'h27:
      //nor
      begin
        CTRL = 32'b00000001000100001001001100001011; // 0000 0010 0010 0000 1001 0011 0000 1011
      end
      6'h2a:
      //slt
      begin
        CTRL = 32'b00000001001100001001001100001011; // 0000 0010 0110 0000 1001 0011 0000 1011
      end
```

Figure 11.20 R type one at write_back stage

All operations in the above figure will set register R and W to 1. CTRL[12] is set to one to set the WR_addr to rd. CTRL[15] is set to one to select the alu_result for WR_data. CTRL[21] is set to 1 to get R2_data. Beside, each operation will their own oprn code.

```
6'h02:
//srl
begin
  CTRL = 32'b00000001000101001001001100001011; // 0000 0001 0001 0100 1001 0011 0000 1011
end
6'h01:
//sll
begin
  CTRL = 32'b00000001010101001001001100001011; // 0000 0001 0101 0100 1001 0011 0000 1011
end
6'h08:
// jr
begin
  CTRL = 32'b00000000000000000000001000001001; // 0000 0000 0000 0000 0000 0010 0000 1001
end
```

Figure 11.21 R type two at write_back stage

For srl and sll operation, it will set CTRL[18] and CTRL[20] to 1 for getting the shift amount (shamt).

For jr operation, it will set CTRL[3] and CTRL[0] to 1 to use R1_data for jumping and register READ signal is set to 1 for reading data.

```
// I type
6'h08 :
//addi
begin
  CTRL=  32'b00000000001001000100101111000001011; // 0000 0000 0100 1000 1001 0111 0000 1011
end
6'h1d :
//multi
begin
  CTRL = 32'b00000000111001000100101111000001011; // 0000 0000 1100 1000 1001 0111 0000 1011
end
6'h0c :
//andi
begin
  CTRL = 32'b00000001100000001001011100001011; // 0000 0001 1000 0000 1001 0111 0000 1011
end
6'h0d :
//ori
begin
  CTRL = 32'b00000001110000001001011100001011; // 0000 0001 1100 0000 1001 0111 0000 1011
end
6'h0f :
//lui
begin
  CTRL = 32'b00000000000000001101010100001011; // 0000 0000 0000 0000 1101 0101 0000 1011
end
6'h0a :
//slti
begin
  CTRL = 32'b00000001001001000100101111000001011; // 0000 0010 0100 1000 1001 0111 0000 1011
end
```

Figure 11.22 I-type write_back

Besides lui, the above operations difference is that it uses rt as the WR_addr and use immediate values.

```
//slti
begin
  CTRL = 32'b00000001001001000100101111000001011; // 0000 0010 0100 1000 1001 0111 0000 1011
end
6'h04 :
//beq
begin
  if(ZERO ===1)
  begin
    CTRL = 32'b00000000101000000001011000001101; // 0000 0000 1010 0000 0001 0110 0000 1101
  end
  else
  begin
    CTRL = 32'b00000000101000000001011000001011; // 0000 0000 1010 0000 0001 0110 0000 1011
  end
end
6'h05 :
//bne
begin
  if(ZERO ===0)
  begin
    CTRL = 32'b00000000101000000001011000001101; // 0000 0000 1010 0000 0001 0110 0000 1101
  end
  else
  begin
    CTRL = 32'b00000000101000000001010100001011; // 0000 0000 1010 0000 0001 0101 0000 1011
  end
end
6'h23 :
//lw
begin
  CTRL = 32'b00000000001001000101101100001011; // 0000 0000 0100 1000 1011 0111 0000 1011
end
6'h2b :
//sw
begin
  CTRL = 32'b00000000001001000000000000001011; // 0000 0000 0100 1000 0000 0000 0000 1011
end
```

Figure 11.23 I-type part 2 write_back

For slti and operation, it will use the 16-bit LSB zero extension imm. For beq and bne operation, it will check the zero flag from alu and then do similar operations to write the corresponding information to the register.

If (R[rs] == R[rt])
  PC = PC + 1 + BranchAddress

If (R[rs] != R[rt])
  PC = PC + 1 + BranchAddress

For lw and sw operation, the following operation will be performed.

R[rt] = M[R[rs]+SignExtImm]

M[R[rs]+SignExtImm] = R[rt]

```
// jmp
begin
  CTRL = 32'h00000000000000000000100000000001; // 0000 0000 0000 0000 0000 1000 0000 0001
end
6'h03 :
//jal
begin
  CTRL = 32'b00000000000000000000100100000001; // 0000 0000 0000 0000 0000 1001 0000 0001
end
6'h1c :
//pop
begin
  CTRL = 32'b00000000001011011101000110000101l; // 0000 0000 0101 1011 1010 0011 0000 1011
end
6'h1b:
// push
begin
  CTRL = 32'b00010000100100110000001010001011; // 0001 0000 1001 0011 0000 0010 1000 1011
end
```

Figure 11.24 J-type write_back

Finally, jmp operation will set CTRL[11] to 1 for to select 31 as WR_addr. For jal operation, it will set CTRL[8] to 1 for writing data to register. Push and pop operations are related. The following operations will be performed.

M[$sp] = R[0]
$sp = $sp - 1

$sp = $sp + 1
R[0] = M[$sp]

The operation will issue the WR_addr to 0. Pop will write to data address and push don't write data to register.

12. Memory

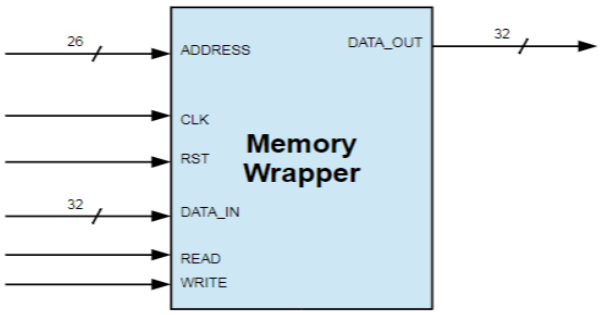


Figure 12.1 Prototype of Memory Wrapper

The memory wrapper module will have one 26 input data and one 32-bit output data. Beside, it has READ and WRITE signal, CLK, RST and a 32-bit data to be written.

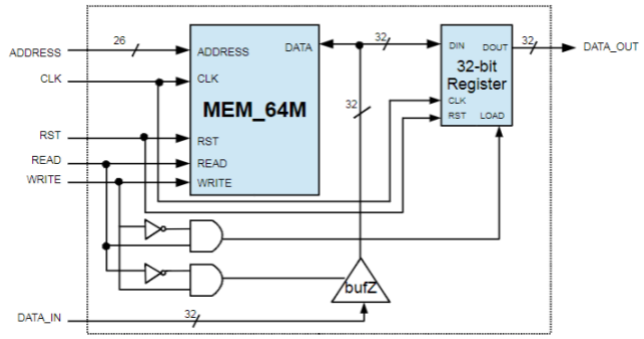


Figure 12.2 Memory Wrapper

From the digital circuit of memory wrapper, we can see that it use inverter, AND gate and bufZ to control load signal of internal register and data out. The whole program is given, and it is completed in behavioral model (figure 12.3). It has the same logic of the one in project II.

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

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

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

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

initial
begin
DATA_OUT = 32'h00000000;
end

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


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

module MEMORY_64MB(DATA, READ, WRITE, ADDR, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// input ports
input READ, WRITE, CLK, RST;
input [`ADDRESS_INDEX_LIMIT:0] ADDR;
// inout ports
inout [`DATA_INDEX_LIMIT:0] DATA;

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x64m [0:`MEM_INDEX_LIMIT]; // memory storage
integer i; // index for reset operation

reg [`DATA_INDEX_LIMIT:0] data_ret; // return data register

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

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
    sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
$readmemh(mem_init_file, sram_32x64m);
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
end
end
endmodule
```

Figure 12.3 Implementation of Memory system

13. Processor

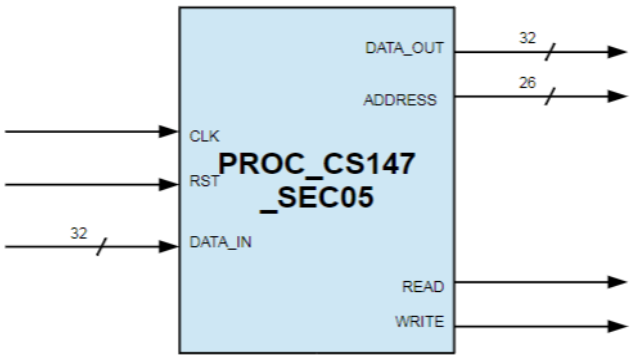


Figure 13.1 Prototype of Processor

The processor will have one 32-bit input, one 32-bit output, and one 26-bit address. It will also issue the READ and WRITE signal.

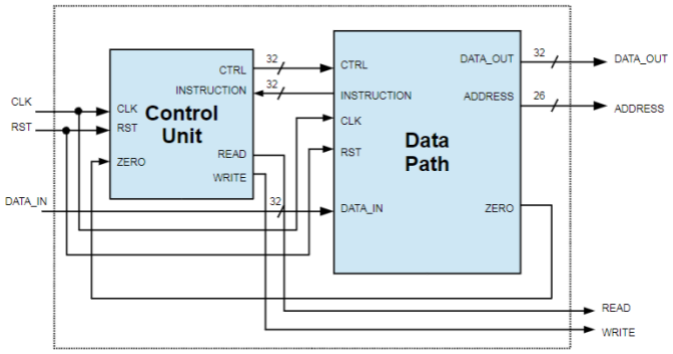


Figure 13.3 Processor

The processor consists control unit and data path. It is implemented at the following figure 1.

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

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

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

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

endmodule;
```

Figure 13.3 Processor

The processor has the instance of the control unit, data path. It has all wires to connect different components. Besides, it has CLK and RST input signal. Also, it has one data bus and READ, WRITE, and ADDR as output.

14. Davinci v1.0

The Davinci v 1.0 is created by combining the processor and the memory system. The following figure 14.1 show the interface of Davinci v 1.0.

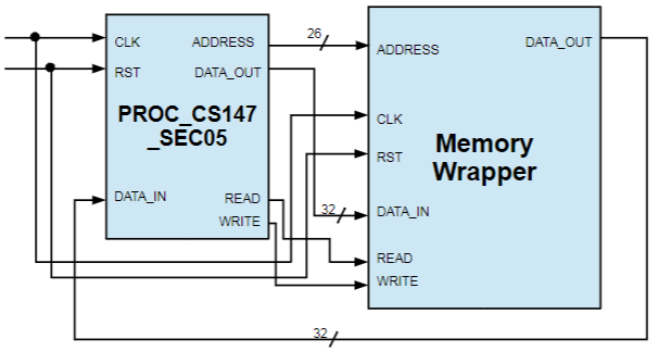


Figure 14.1 Davinci v1.0

The following figure 14.2 is the implementation of the Davinci v 1.0. It creates the instances of processor and the memory system. Besides, it has a parameter about the file reading and writing.

```
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;
```

Figure 14.2 Implementation of Davinci v1.0

To test the whole system, three files are created and passed to the system. By comparing the dump file and the golden file, we can see whether the Davinci v1.0 works correctly.

```
`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;

 // 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";
 defparam da_vinci_inst.mem_init_file = "tests.dat";
DA_VINCI da_vinci_inst(.DATA(DATA), .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", da_vinci_inst.memory_inst.sram_32x64m, 'h03fffff0, 'h03ffffff);
       //$writememh("fibonacci_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h01000000, 'h0100000f);
       $writememh("tests_mem_dump.dat", da_vinci_inst.memory_inst.sram_32x64m, 'h01000000, 'h0100000f);
       $stop;

end
endmodule;
```

Figure 14.3 Davinci v1.0 testbench

The figure 14.3 is the Davinci v1.0 testbench. It will create the instance of Davinci v1.0. There are three files total, so the file is passed to the system one by one through memory model. While execute the instruction in the file, some data will be written to the dump file.

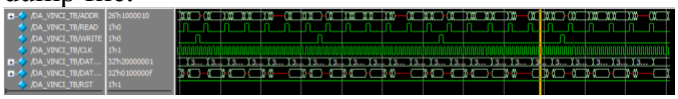


Figure 14.4 Waveform of testbench

In the figure 14.4, it is the waveform when the tests.dat was passed into the system. The second part is the whole waveform. The first part is the one after zooming in. The first part of the figure shows the address in increasing by 1 and the data is changing all the time as the instructions are implemented.

```
# @       30ns -> [0X20420001] addi   r[02], r[02], 0X0001;
# @       80ns -> [0X3c000100] lui    r[00], 0X0100;
# @      130ns -> [0Xac010000] sw     r[00], r[01], 0X0000;
# @      180ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @      230ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @      280ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @      330ns -> [0X00411020] add    r[02], r[01], r[02];
# @      380ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @      430ns -> [0X08001003] jmp    0X0001003;
# @      480ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @      530ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @      580ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @      630ns -> [0X00411020] add    r[02], r[01], r[02];
# @      680ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @      730ns -> [0X08001003] jmp    0X0001003;
# @      780ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @      830ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @      880ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @      930ns -> [0X00411020] add    r[02], r[01], r[02];
# @      980ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     1030ns -> [0X08001003] jmp    0X0001003;
# @     1080ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     1130ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     1180ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     1230ns -> [0X00411020] add    r[02], r[01], r[02];
# @     1280ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     1330ns -> [0X08001003] jmp    0X0001003;
# @     1380ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     1430ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     1480ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     1530ns -> [0X00411020] add    r[02], r[01], r[02];
# @     1580ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     1630ns -> [0X08001003] jmp    0X0001003;
# @     1380ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     1430ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     1480ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     1530ns -> [0X00411020] add    r[02], r[01], r[02];
# @     1580ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     1630ns -> [0X08001003] jmp    0X0001003;
# @     1680ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     1730ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     1780ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     1830ns -> [0X00411020] add    r[02], r[01], r[02];
# @     1880ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     1930ns -> [0X08001003] jmp    0X0001003;
# @     1980ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     2030ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     2080ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     2130ns -> [0X00411020] add    r[02], r[01], r[02];
# @     2180ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     2230ns -> [0X08001003] jmp    0X0001003;
# @     2280ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     2330ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     2380ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     2430ns -> [0X00411020] add    r[02], r[01], r[02];
# @     2480ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     2530ns -> [0X08001003] jmp    0X0001003;
# @     2580ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     2630ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     2680ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     2730ns -> [0X00411020] add    r[02], r[01], r[02];
# @     2780ns -> [0X20610000] addi   r[03], r[01], 0X0000;
# @     2830ns -> [0X08001003] jmp    0X0001003;
# @     2880ns -> [0X20000001] addi   r[00], r[00], 0X0001;
# @     2930ns -> [0Xac020000] sw     r[00], r[02], 0X0000;
# @     2980ns -> [0X20430000] addi   r[02], r[03], 0X0000;
# @     3030ns -> [0X00411020] add    r[02], r[01], r[02];
# @     3080ns -> [0X20610000] addi   r[03], r[01], 0X0000;
```

```
# @   3130ns -> [0X08001003] jmp   0X0001003;
# @   3180ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   3230ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   3280ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   3330ns -> [0X00411020] add  r[02], r[01], r[02];
# @   3380ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   3430ns -> [0X08001003] jmp   0X0001003;
# @   3480ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   3530ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   3580ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   3630ns -> [0X00411020] add  r[02], r[01], r[02];
# @   3680ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   3730ns -> [0X08001003] jmp   0X0001003;
# @   3780ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   3830ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   3880ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   3930ns -> [0X00411020] add  r[02], r[01], r[02];
# @   3980ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   4030ns -> [0X08001003] jmp   0X0001003;
# @   4080ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   4130ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   4180ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   4230ns -> [0X00411020] add  r[02], r[01], r[02];
# @   4280ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   4330ns -> [0X08001003] jmp   0X0001003;
# @   4380ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   4430ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   4480ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   4530ns -> [0X00411020] add  r[02], r[01], r[02];
# @   4580ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   4630ns -> [0X08001003] jmp   0X0001003;
# @   4680ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# @   4730ns -> [0Xac020000] sw    r[00], r[02], 0X0000;
# @   4780ns -> [0X20430000] addi  r[02], r[03], 0X0000;
# @   4830ns -> [0X00411020] add  r[02], r[01], r[02];
# @   4880ns -> [0X20610000] addi  r[03], r[01], 0X0000;
# @   4930ns -> [0X08001003] jmp   0X0001003;
# @   4980ns -> [0X20000001] addi  r[00], r[00], 0X0001;
# ** Note: $stop    : C:/Users/pansh/Downloads/prj_02/da_vinci_tb.v(56)
#    Time: 5010 ns  Iteration: 0  Instance: /DA_VINCI_TB
```

Figure 14.5 Text output of the testbench

There will be text output which is print_instruction in the process of run the whole program in the file. We can see all lines are working correctly.

fibonacci - Notepad

File Edit Format View Help

```
@0001000
20420001 //          addi r[2], r[2], 0x0001;
3C000100 //          lui  r[0], 0x0100;
AC010000 //          sw   r[1], r[0], 0x0000;
20000001 // loop:    addi r[0], r[0], 0x0001;
AC020000 //          sw   r[2], r[0], 0x0000;
20430000 //          addi r[3], r[2], 0x0000;
00411020 //          add  r[2], r[2], r[1];
20610000 //          addi r[1], r[3], 0x0000;
08001003 //          jmp  loop;
```

fibonacci_mem_dump - Notepad

File Edit Format View Help

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

fibonacci_mem_dump.golden - Notepad

File Edit Format View Help

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

Figure 14.6 fibonacci

The first file is the one who calculate the Fibonacci. According to the assembly code, they are converted to the machine code. The program is to calculate the fibonacci numbers. After passing the file to the system, the result will be written into the fibonacci.mem_dump.dat. Compared with the golden file, we can see it is the same. Thus, the test is passed.

tests - Notepad

File Edit Format View Help

```
@0001000
20420010 //          addi r[2], r[2], 0x0010;
3C000100 //          lui  r[0], 0x0100;
AC010000 //          sw   r[1], r[0], 0x0000;
20000001 // loop:    addi r[0], r[0], 0x0001;
AC020000 //          sw   r[2], r[0], 0x0000;
20430000 //          addi r[3], r[2], 0x0000;
00411022 //          sub  r[2], r[2], r[1];
20610000 //          addi r[1], r[3], 0x0000;
08001003 //          jmp  loop;
```

tests_mem_dump - Notepad

File Edit Format View Help

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000010
00000010
00000000
ffffffff0
ffffffff0
00000000
00000010
00000010
00000000
ffffffff0
ffffffff0
00000000
00000010
00000010
00000000
ffffffff0
ffffffff0
00000000
00000010
00000010
00000000
```

tests_mem_dump_golden - Notepad

File Edit Format View Help

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000010
00000010
00000000
ffffffff0
ffffffff0
00000000
00000010
00000010
00000000
ffffffff0
ffffffff0
00000000
00000010
00000010
00000000
```

Figure 14.7 tests

The figure 14.7, we set the first data to 00000010; after that, the subtraction will be implemented in the for loop. The program will produce a circular pattern. The result is written into the test_mem_dump.dat. Compared with the test_mem_dump.dat, it is the same. Thus, it is passed.

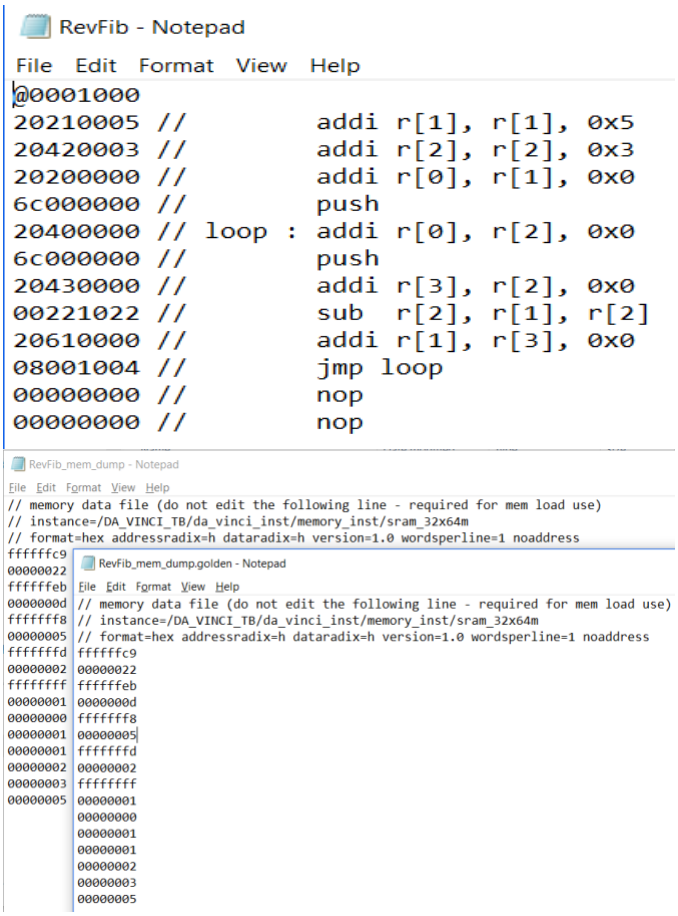


Figure 14.8 RevFib

The first test is almost the same with the RevFibonacci numbers. The only difference is that the initialization numbers. The result is written to the Revfib_mem_dump.dat. Compared with the golden file, it is the same. Thus, it is passed.

15. Conclusion

In project3, I have a better understanding on the Verilog programming language in the process of designing and implemented the Davinci v1.0. The most important thing that I learned is that how the computer works in the gate level. I learned different function of different component including memory, register, control unit, and ALU. I know how the data is transferred from one component to the other component through data path. The control unit is key to handle the different signal and data path. I know what operations will be done in different stage by using state machine. Overall, this project is very complex and stressful, but I have much deeper comprehension about how hardware works exactly.