Mixed Model of Basic Computer System

Ryan Tran
San Jose State University
ryan.l.tran@sjsu.edu

***Abstract* – This report details the design and implementation of the bare minimum computer system DaVinci v1.0m which supports the CS147DV instruction set.**

Index Terms – gate, mixed, model, computer, system.

I. Introduction

The goal of this project is to implement a mixed model, both behavioral and gate level, of a computer system consisting of a 32-bit processor and 256MB memory, called DaVinci v1.0m.

II. System Requirements

The following section describes the various parts of the DaVinci v1.0m system in more detail.

A. CS147DV Instruction Set

Professor Patra created this instruction set for his CS147 class at San Jose State University. It is used in the ALU and CU to determine which operations to perform.

B. Arithmetic Logic Unit (ALU)

The ALU is the first of the three components of a processor. The ALU performs all arithmetic and logic operations for the computer system. In addition, the ALU also sets a zero flag, which is set to 1 when an operation results in a zero result and is set to 0 when an operation results in a non-zero result.

C. Register File (RF)

The RF is the second of the three components of a processor. The RF has dual read and single write capabilities using its 32 registers, each of 32 bits. Initially, all registers are defaulted to the value 0.

D. Memory

The memory for the DaVinci v1.0 system is 32-bit accessible with 64M addresses, equating to a total of 256MB of memory. It is similar to the RF, except it is single read and single write, using a single inout port for read and write operations.

E. Control Unit (CU)

The CU is the third of the three components of a processor. The CU uses a five-state state machine to synchronize the operations of the processor. Based on the current state and the operation code from the CS147DV instruction set, the CU performs the necessary operations.

F. Data Path

The data path controls the way data flows. It works together with the control unit to form the processor. Its components include the ALU, RF, instruction register, program counter, and stack pointer.

G. Processor

The processor is comprised of the control unit and data path. It receives data from the memory and clock and reset signals. It outputs to the memory an address and data for writing along with a read and write signal.

H. System Implementation

The entire DaVinci v1.0m system is comprised of the processor and the memory wrapper. This is the highest level of the system. It only requires two inputs, the clock signal and the reset signal.

III. ALU Design, Implementation, and Testing

The ALU receives three inputs: “OP1” and “OP2” which are 32-bit operands and “OPRN” which is a 6-bit operation code. The ALU has two outputs: “OUT” which is a 32-bit result from the performed operation and “ZERO” which is a 1-bit flag that indicates whether the performed operation resulted to zero. The ALU is comprised of multiple subparts: binary ripple carry adder/subtractor, signed multiplier, shifter, and logic gates, all of which are 32-bit. The following is the digital circuit diagram, implementation, and test bench results for the ALU:

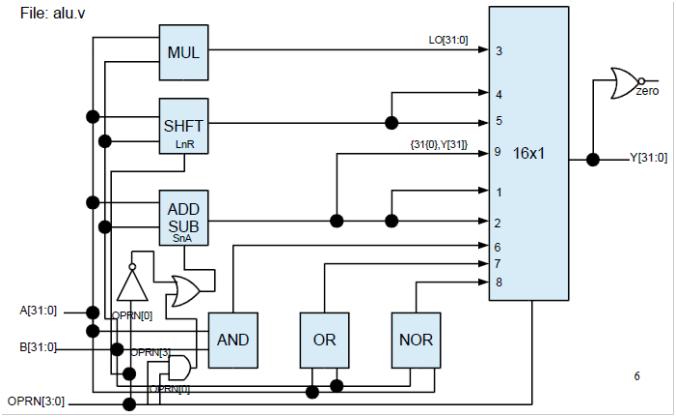

Image 3.1 ALU digital circuit diagram

```
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 [`DATA_INDEX_LIMIT:0] unused;

wire [`DATA_INDEX_LIMIT:0] add_sub_res;
wire not_oprn0;
wire and_oprn3_oprn0;
wire SnA;
not not_oprn0_inst(not_oprn0, OPRN[0]);
and and_oprn3_oprn0_inst(and_oprn3_oprn0, OPRN[3], OPRN[0]);
or or_SnA_inst(SnA, not_oprn0, and_oprn3_oprn0);
RC_ADD_SUB_32 rc_add_sub_32_inst(.Y(add_sub_res), .CO(unused[0]), .A(OP1), .B(OP2), .SnA(SnA));

wire [`DATA_INDEX_LIMIT:0] mul_res;
MULT32 mult32_inst(.HI(unused), .LO(mul_res), .A(OP1), .B(OP2));

wire [`DATA_INDEX_LIMIT:0] shift_res;
SHIFT32 shift32_inst(.Y(shift_res), .D(OP1), .S(OP2), .LnR(OPRN[0]));

wire [`DATA_INDEX_LIMIT:0] and_res;
AND32_2x1 and32_2x1_inst(.Y(and_res), .A(OP1), .B(OP2));

wire [`DATA_INDEX_LIMIT:0] or_res;
OR32_2x1 or32_2x1_inst(.Y(or_res), .A(OP1), .B(OP2));

wire [`DATA_INDEX_LIMIT:0] nor_res;
NOR32_2x1 nor32_2x1_inst(.Y(nor_res), .A(OP1), .B(OP2));

MUX32_16x1 mux32_16x1_inst(.Y(OUT), .I0(unused), .I1(add_sub_res), .I2(add_sub_res), .I3(mul_res),
                           .I4(shift_res), .I5(shift_res), .I6(and_res), .I7(or_res),
                           .I8(nor_res), .I9({31'b0, add_sub_res[31]}), .I10(unused), .I11(unused),
                           .I12(unused), .I13(unused), .I14(unused), .I15(unused), .S(OPRN[3:0]));

wire [31:0] zero_res;
or or_init_inst(zero_res[0], OUT[0], OUT[0]);
genvar i;
generate
    for (i = 1; i < 31; i = i + 1)
    begin : or_gen_loop
        or or_inst(zero_res[i], OUT[i], zero_res[i - 1]);
    end
endgenerate
wire not_zero;
or or_end_inst(not_zero, OUT[31], zero_res[30]);
not not_inst(ZERO, not_zero);
```

Image 3.2 Implementation of ALU

```
# [TEST] 3 + 4 = 7 , got 7 ... [PASSED]
# [TEST] 20 - 15 = 5 , got 5 ... [PASSED]
# [TEST] 8 * 4 = 32 , got 32 ... [PASSED]
# [TEST] 8 >> 2 = 2 , got 2 ... [PASSED]
# [TEST] 4 << 4 = 64 , got 64 ... [PASSED]
# [TEST] 5 & 10 = 0 , got 0 ... [PASSED]
# [TEST] 10 | 20 = 30 , got 30 ... [PASSED]
# [TEST] 3 ~| 6 = 4294967288 , got 4294967288 ... [PASSED]
# [TEST] 5 < 9 = 1 , got 1 ... [PASSED]
#
#       Total number of tests           9
#       Total number of pass            9
```

Image 3.3 ALU test bench transcript

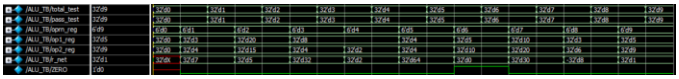

Image 3.4 ALU test bench waveform

A. *Binary Ripple Carry Adder/Subtractor*

The binary ripple carry adder/subtractor is implemented by chaining 32 full adders. The last carry out signal is the carry out signalf or the entire binary ripple carry adder. The following is the digital circuit diagram, implementation, and test bench results for the binary ripple carry adder/subtractor:

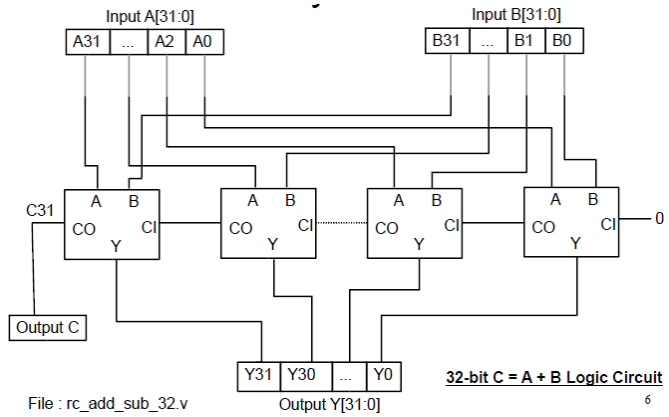

Image 3.5 Binary ripple carry adder/subtractor digital circuit diagram

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

wire [`DATA_INDEX_LIMIT:0] XORs;
wire [`DATA_WIDTH:0] CIs;

assign CO = CIs[`DATA_WIDTH];
assign CIs[0] = SnA;

genvar i;
generate
    for (i = 0; i < `DATA_WIDTH; i = i + 1)
    begin : fa_32_gen_loop
        xor xor_inst(XORs[i], B[i], SnA);
        FULL_ADDER fa_inst(.S(Y[i]), .CO(CIs[i + 1]), .A(A[i]), .B(XORs[i]), .CI(CIs[i]));
    end
endgenerate

endmodule
```

Image 3.6 Implementation of binary ripple carry adder/subtractor

```
# A:          0 B:          0 SnA:0 Y:          0 CO:0
# A:         32 B:         16 SnA:1 Y:         16 CO:1
# A:         32 B:         16 SnA:0 Y:         48 CO:0
# A:         16 B:         32 SnA:1 Y:4294967280 CO:0
# A:         16 B:         32 SnA:0 Y:         48 CO:0
# A:2147483648 B:2147483648 SnA:0 Y:          0 CO:1
# A:2147483649 B:2147483649 SnA:0 Y:          2 CO:1
```

Image 3.7 Binary ripple carry adder/subtractor test bench transcript

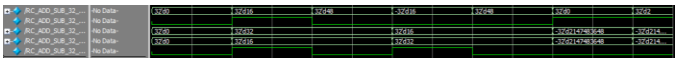

Image 3.8 Binary ripple carry adder/subtractor test bench waveform

A1. *Full Adder*

The full adder is implemented by combining two half adders. The following is the digital circuit diagram, implementation, and test bench results for the full adder:

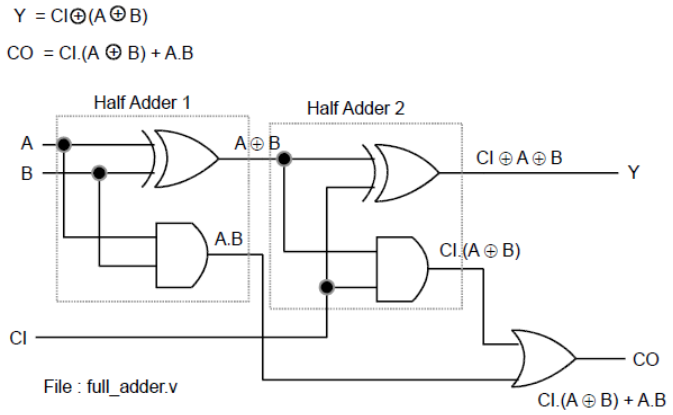

Image 3.9 Full adder digital circuit diagram

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

wire Y_1, C_1, C_2;

HALF_ADDER ha_inst_1(.Y(Y_1), .C(C_1), .A(A), .B(B));
HALF_ADDER ha_inst_2(.Y(S), .C(C_2), .A(Y_1), .B(CI));

or or_inst(CO, C_2, C_1);

endmodule
```

Image 3.10 Implementation of full adder

```
# A:0 B:0 CI:0 S:0 CO:0
# A:0 B:0 CI:1 S:1 CO:0
# A:0 B:1 CI:0 S:1 CO:0
# A:0 B:1 CI:1 S:0 CO:1
# A:1 B:0 CI:0 S:1 CO:0
# A:1 B:0 CI:1 S:0 CO:1
# A:1 B:1 CI:0 S:0 CO:1
# A:1 B:1 CI:1 S:1 CO:1
```

Image 3.11 Full adder test bench transcript

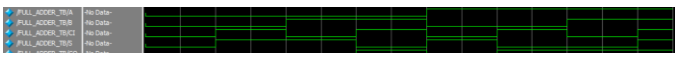

Image 3.12 Full adder test bench waveform

A2. Half Adder

The half adder is implemented using simply an xor gate and an and gate. The two inputs are passed through both the xor gate and the and gate to obtain respectively the main result and the carry out. The following is the digital circuit diagram, implementation, and test bench results for the half adder:

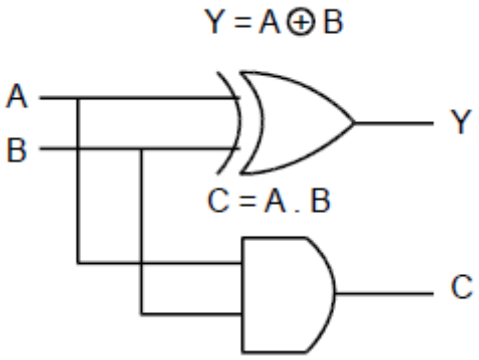

Image 3.13 Half adder digital circuit diagram

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

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

endmodule
```

Image 3.14 Implementation of half adder

```
# A:0 B:0 Y:0 C:0
# A:1 B:0 Y:1 C:0
# A:0 B:1 Y:1 C:0
# A:1 B:1 Y:0 C:1
```

Image 3.15 Half adder test bench transcript

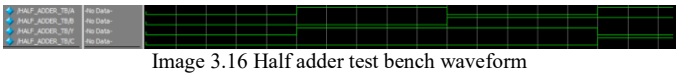

Image 3.16 Half adder test bench waveform

B. Signed Multiplier

The signed multiplier is implemented using an unsigned multiplier, multiplexers, and two's complements. Both the multiplicand and the multiplier are passed through a two's complement circuit to retrieve its corresponding two's complement. Then they are passed through multiplexers with their most significant bit as a selection signal in order to determine whether to use the signed or unsigned version of the multiplicand and multiplier. Then, they are passed through the unsigned multiplier. The unsigned multiplier returns a 64-bit unsigned result, which is passed through another two's complement and another multiplexer, with a selection signal of an xor between the multiplicand and the multiplier's most significant bits, to determine the final result, signed or unsigned. The following is the digital circuit diagram, implementation, and test bench results for the signed multiplier:

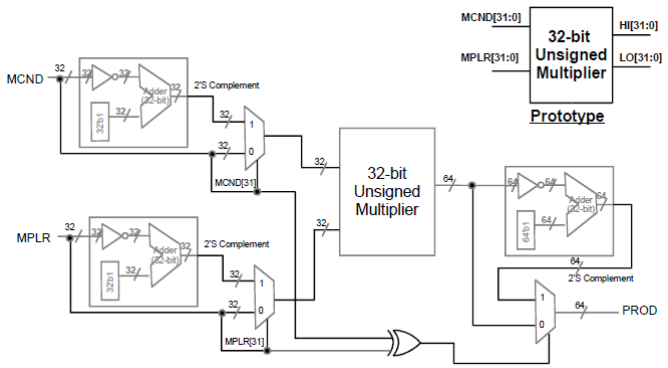

Image 3.17 Signed multiplier digital circuit diagram

```
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;

// Two's complement multiplicand and multiplier
wire [31:0] twoscomp32_mcnd;
wire [31:0] twoscomp32_mplr;
TWOSCOMP32 twoscomp32_inst_mcnd(.Y(twoscomp32_mcnd), .A(A));
TWOSCOMP32 twoscomp32_inst_mplr(.Y(twoscomp32_mplr), .A(B));

// Mux multiplicand and multiplier to choose complement or not
wire [31:0] mux32_2x1_mcnd;
wire [31:0] mux32_2x1_mplr;
MUX32_2x1 mux32_2x1_inst_mcnd(.Y(mux32_2x1_mcnd), .I0(A), .I1(twoscomp32_mcnd), .S(A[31]));
MUX32_2x1 mux32_2x1_inst_mplr(.Y(mux32_2x1_mplr), .I0(B), .I1(twoscomp32_mplr), .S(B[31]));

// Unsigned multiplication
wire [63:0] mult32_u_res;
MULT32_U mult32_u_inst(.HI(mult32_u_res[63:32]), .LO(mult32_u_res[31:0]), .A(mux32_2x1_mcnd), .B(mux32_2x1_mplr));

// Two's complement of unsigned multiplication
wire [63:0] twoscomp64_res;
TWOSCOMP64 twoscomp64_inst(.Y(twoscomp64_res), .A(mult32_u_res));

// XOR to choose signed or unsigned
wire xorRes;
xor xor_inst(xorRes, A[31], B[31]);

// Mux to choose complement or not
wire [63:0] mux64_2x1_res;
MUX64_2x1 mux64_2x1_inst(.Y(mux64_2x1_res), .I0(mult32_u_res), .I1(twoscomp64_res), .S(xorRes));

// Buffer to output
BUF32 buf32_inst_HI(.Y(HI), .A(mux64_2x1_res[63:32]));
BUF32 buf32_inst_LO(.Y(LO), .A(mux64_2x1_res[31:0]));

endmodule
```

Image 3.18 Implementation of signed multiplier

```
# A:          0 B:          0 HI:          0 LO:          0
# A:          4 B:          8 HI:          0 LO:         32
# A:          8 B:          4 HI:          0 LO:         32
# A:      65536 B:      65536 HI:          1 LO:          0
# A:      65535 B:      65535 HI:          0 LO:4294836225
# A:4294967295 B:4294967295 HI:          0 LO:          1
# A:4294967292 B:          8 HI:4294967295 LO:4294967264
# A:          8 B:4294967292 HI:4294967295 LO:4294967264
# A:4294901760 B:      65536 HI:4294967295 LO:          0
# A:      65535 B:4294901761 HI:4294967295 LO:     131071
# A:          1 B:          1 HI:          0 LO:          1
```

Image 3.19 Signed multiplier test bench transcript

Image 3.20 Signed multiplier test bench waveform

B1. Unsigned Multiplier

The unsigned multiplier is implemented using 31 binary ripple carry adder/subtractors along with 32 and gates. This is a combinational implementation of the unsigned multiplier, where at each addition step, the least significant bit of the addition result is recorded in the result. The first "level" has two and gates while the rest have one and gate and one ripple carry adder/subtractor. Though we are using an adder/subtractor, we only use the addition functionality. The following is the digital circuit diagram, implementation, and test bench results for the unsigned multiplier:

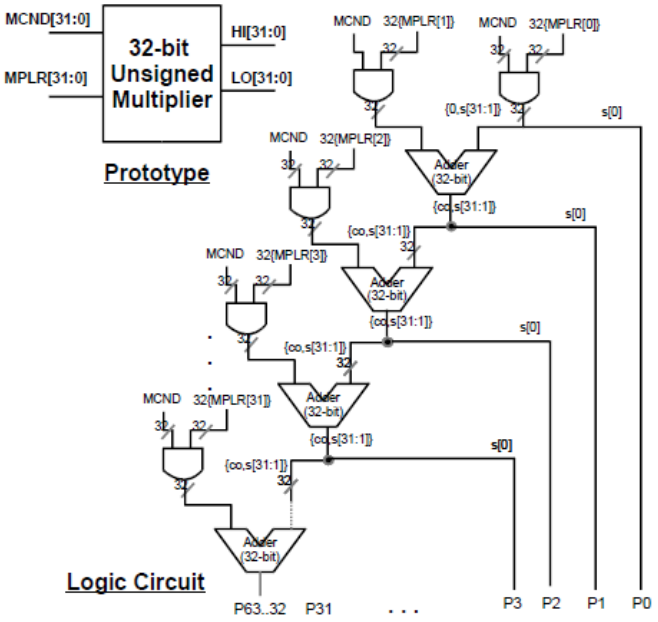

Image 3.21 Unsigned multiplier digital circuit diagram

```
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 COs [31:0];
wire [31:0] res [31:0];

AND32_2x1 and32_2x1_inst(res[0], A, {32{B[0]}});
buf buf_inst_1(COs[0], 1'b0);
buf buf_inst_2(LO[0], res[0][0]);

genvar i;
generate
    for (i = 1; i < 32; i = i + 1)
    begin : adder_31_gen_loop
        wire [31:0] and_res;
        AND32_2x1 and32_2x1_inst(and_res, A, {32{B[i]}});
        RC_ADD_SUB_32 rc_add_sub_32_inst(.Y(res[i]), .CO(COs[i]), .A(and_res), .B({COs[i - 1], res[i - 1][31:1]}), .SnA(1'b0));
        buf buf_inst(LO[i], res[i][0]);
    end
endgenerate
BUF32 buf32_inst(.Y(HI), .A({COs[31], res[31][31:1]}));

endmodule
```

Image 3.22 Implementation of unsigned multiplier

```
# A:          0 B:          0 HI:          0 LO:          0
# A:          4 B:          8 HI:          0 LO:         32
# A:          8 B:          4 HI:          0 LO:         32
# A:      65536 B:      65536 HI:          1 LO:          0
# A:      65535 B:      65535 HI:          0 LO:4294836225
# A:4294967295 B:4294967295 HI:4294967294 LO:          1
```

Image 3.23 Unsigned multiplier test bench transcript

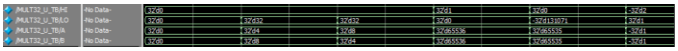

Image 3.24 Unsigned multiplier test bench waveform

B2. Two's Complement

The two's complement is implemented very simply with an inverter and an adder. The two's complement is found by inverting the bits and adding one, so the implementation is a 32-bit inverter which is added with a 32-bit value of 1 through a ripply carry adder. The 64-bit version simply use a 64-bit inverter and a 64-bit ripple carry adder, but the concept is the exact same. The following is the digital circuit diagram and implementation for the two's complement:

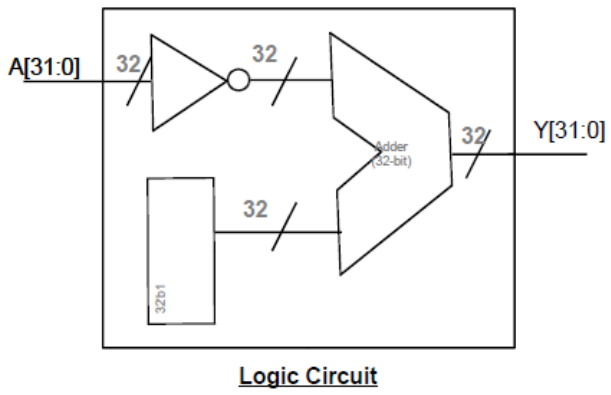

Image 3.25 Two's complement digital circuit diagram

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

wire [31:0] inv_res;
wire CO;

INV32_1x1 inv32_1x1_inst(inv_res, A);
RC_ADD_SUB_32 rc_add_sub_32_inst(.Y(Y), .CO(CO), .A(inv_res), .B(32'b1), .SnA(1'b0));

endmodule
```

Image 3.26 Implementation of two's complement

```
# A:00000000000000000000000000000000 Y:00000000000000000000000000000000
# A:00000000000000000000000000000001 Y:11111111111111111111111111111111
# A:00000000000000000000000000100000 Y:11111111111111111111111111100000
# A:11111111111111111111111111111111 Y:00000000000000000000000000000001
# A:11111111111111111111111111111111 Y:00000000000000000000000000000001
```

Image 3.27 Two's complement test bench transcript

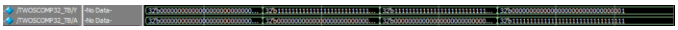

Image 3.28 Two's complement test bench waveform

B3. Multiplexer

The base 1-bit, 2x1 multiplexer is implemented using basic logic gates: one inverter, two ands, and one or. The following is the digital circuit diagram, implementation, and test bench results for the 1-bit, 2x1 multiplexer:

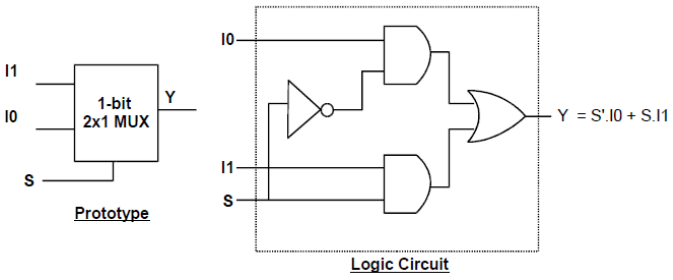

Image 3.29 1-bit, 2x1 multiplexer digital circuit diagram

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

wire s_not, and_1, and_2;

not not_inst(s_not, S);
and and_inst_1(and_1, I0, s_not);
and and_inst_2(and_2, I1, S);
or or_inst(Y, and_1, and_2);

endmodule
```

Image 3.30 Implementation of 1-bit, 2x1 multiplexer

```
# I0:0 I1:0 S:0 Y:0
# I0:0 I1:0 S:1 Y:0
# I0:0 I1:1 S:0 Y:0
# I0:0 I1:1 S:1 Y:1
# I0:1 I1:0 S:0 Y:1
# I0:1 I1:0 S:1 Y:0
# I0:1 I1:1 S:0 Y:1
# I0:1 I1:1 S:1 Y:1
```

Image 3.31 1-bit, 2x1 multiplexer test bench transcript

Image 3.32 1-bit, 2x1 multiplexer test bench waveform

The 32-bit, 2x1 multiplexer is simply 32, 1-bit 2x1 multiplexers connected together. The following is the digital circuit diagram, implementation, and test bench results for the 32-bit, 2x1 multiplexer:

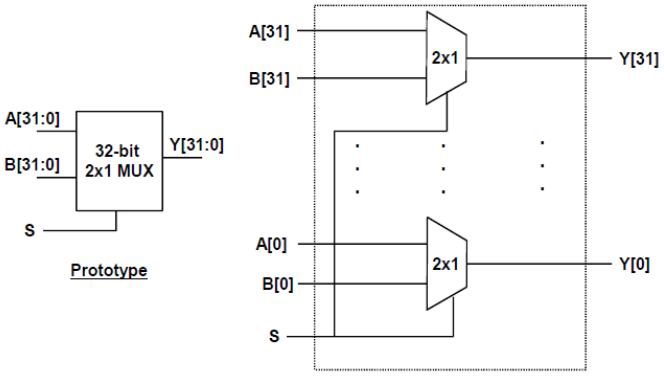

Image 3.33 32-bit, 2x1 multiplexer digital circuit diagram

```
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 : mux1_2x1_gen_loop
        MUX1_2x1 mux1_2x1_inst(Y[i], I0[i], I1[i], S);
    end
endgenerate

endmodule
```

Image 3.34 Implementation of 32-bit, 2x1 multiplexer

```
# I0:          0 I1:          0 S:0 Y:          0
# I0:4294967294 I1:4294967295 S:1 Y:4294967295
# I0:4294967294 I1:4294967295 S:0 Y:4294967294
```

Image 3.35 32-bit, 2x1 multiplexer transcript

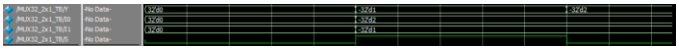

Image 3.36 32-bit, 2x1 multiplexer waveform

Subsequent larger input multiplexers are implemented utilizing smaller multiplexers. The following is the digital circuit diagram, implementation, and test bench results for the 32-bit, 32x1 multiplexer:

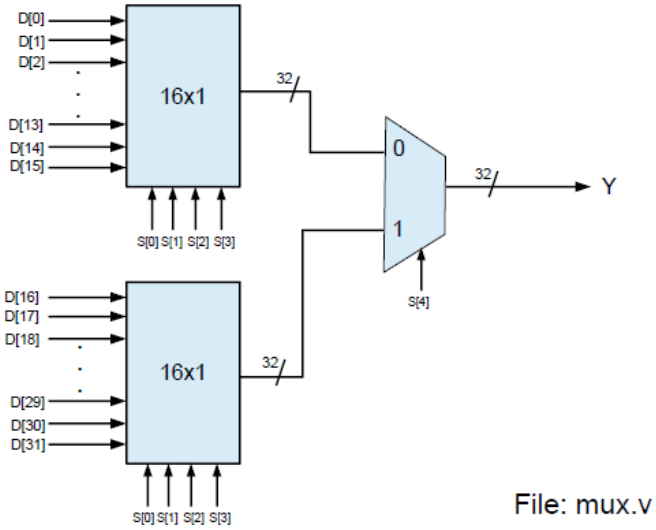

Image 3.37 32-bit, 32x1 multiplexer digital circuit diagram

```
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] mux32_16x1_res_1;
wire [31:0] mux32_16x1_res_2;

MUX32_16x1 mux32_16x1_inst_1(.Y(mux32_16x1_res_1), .I0(I0), .I1(I1), .I2(I2), .I3(I3),
                             .I4(I4), .I5(I5), .I6(I6), .I7(I7),
                             .I8(I8), .I9(I9), .I10(I10), .I11(I11),
                             .I12(I12), .I13(I13), .I14(I14), .I15(I15), .S(S[3:0]));
MUX32_16x1 mux32_16x1_inst_2(.Y(mux32_16x1_res_2), .I0(I16), .I1(I17), .I2(I18), .I3(I19),
                             .I4(I20), .I5(I21), .I6(I22), .I7(I23),
                             .I8(I24), .I9(I25), .I10(I26), .I11(I27),
                             .I12(I28), .I13(I29), .I14(I30), .I15(I31), .S(S[3:0]));
MUX32_2x1 mux32_2x1_inst(.Y(Y), .I0(mux32_16x1_res_1), .I1(mux32_16x1_res_2), .S(S[4]));

endmodule
```

Image 3.38 Implementation of 32-bit, 32x1 multiplexer

```
# I0:4294967264 I1:4294967265 I2:4294967266 I3:4294967267 I4:4294967268 I5:4294967269 I6:4294967270 I7:4294967271
8 I15:4294967279 I16:4294967280 I17:4294967281 I18:4294967282 I19:4294967283 I20:4294967284 I21:4294967285 I22:42
:4294967293 I30:4294967294 I31:4294967295 S: 0 Y:4294967264
# I0:4294967264 I1:4294967265 I2:4294967266 I3:4294967267 I4:4294967268 I5:4294967269 I6:4294967270 I7:4294967271
8 I15:4294967279 I16:4294967280 I17:4294967281 I18:4294967282 I19:4294967283 I20:4294967284 I21:4294967285 I22:42
:4294967293 I30:4294967294 I31:4294967295 S: 1 Y:4294967265
# I0:4294967264 I1:4294967265 I2:4294967266 I3:4294967267 I4:4294967268 I5:4294967269 I6:4294967270 I7:4294967271
8 I15:4294967279 I16:4294967280 I17:4294967281 I18:4294967282 I19:4294967283 I20:4294967284 I21:4294967285 I22:42
:4294967293 I30:4294967294 I31:4294967295 S: 2 Y:4294967266
```

Image 3.39 32-bit, 32x1 multiplexer test bench transcript

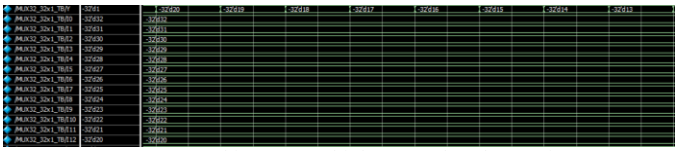

Image 3.40 32-bit, 32x1 multiplexer test bench waveform

C. *Shifter*

The shifter is implemented using a barrel shifter and a multiplexer. The following is the digital circuit diagram, implementation, and test bench results for the shifter:

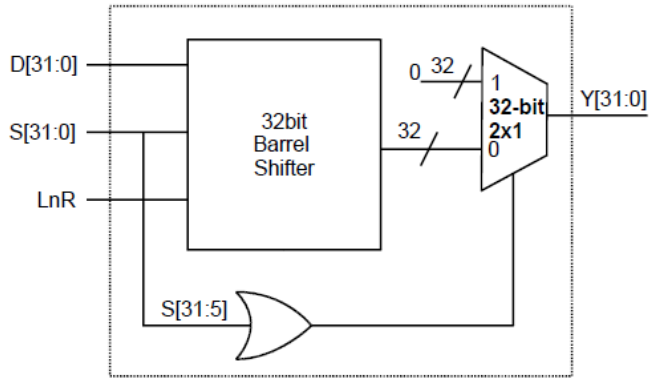

Image 3.41 Shifter digital circuit diagram

```
module SHIFT32(Y, D, S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [31:0] S;
input LnR;

wire [31:0] barrel_shifter32_res;

wire [31:5] or_res;
or or_init_inst(or_res[5], S[5], S[5]);
genvar i;
generate
    for (i = 6; i <= 31; i = i + 1)
    begin : or_gen_loop
        or or_inst(or_res[i], S[i], or_res[i - 1]);
    end
endgenerate

BARREL_SHIFTER32 barrel_shifter32_inst(.Y(barrel_shifter32_res), .D(D), .S(S[4:0]), .LnR(LnR));
MUX32_2x1 mux32_2x1_inst(.Y(Y), .I0(barrel_shifter32_res), .I1(32'b0), .S(or_res[31]));

endmodule
```

Image 3.42 Implementation of shifter

```
# D:          0 S:          0 LnR:0 Y:          0
# D:          2 S:          4 LnR:1 Y:         32
# D:          5 S:          5 LnR:1 Y:        160
# D:         13 S:          3 LnR:1 Y:        104
# D:2147483648 S:          1 LnR:1 Y:          0
# D:2147483648 S:          3 LnR:1 Y:          0
# D:         64 S:          4 LnR:0 Y:          4
# D:        100 S:          5 LnR:0 Y:          3
# D:        123 S:          3 LnR:0 Y:         15
# D:          1 S:          1 LnR:0 Y:          0
# D:          1 S:          3 LnR:0 Y:          0
# D:          1 S:         31 LnR:1 Y:2147483648
# D:          1 S:         32 LnR:1 Y:          0
# D:          1 S:         33 LnR:1 Y:          0
# D:          1 S:         35 LnR:1 Y:          0
```

Image 3.43 Shifter test bench transcript

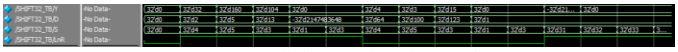

Image 3.44 Shifter test bench waveform

C1. 32-Bit Barrel Shifter

The 32-bit barrel shifter is implemented using a 32-bit right shifter, a 32-bit left shifter, and a 32-bit, 2x1 multiplexer. Depending on the select signal, the multiplexer will output either the right or the left shifted result. The following is the digital circuit diagram, implementation, and test bench results for the 32-bit barrel shifter:

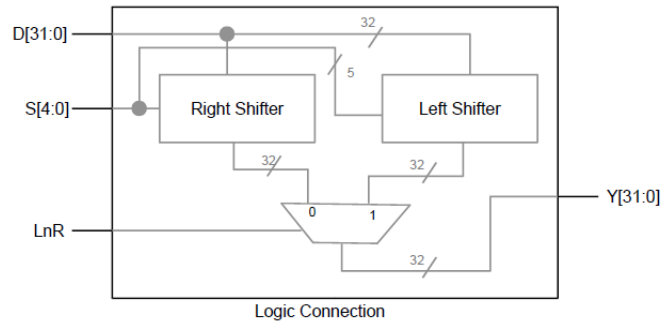

Image 3.45 32-bit barrel shifter digital circuit diagram

```
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] shift32_l_res;
wire [31:0] shift32_r_res;

SHIFT32_L shift32_l_inst(.Y(shift32_l_res), .D(D), .S(S));
SHIFT32_R shift32_r_inst(.Y(shift32_r_res), .D(D), .S(S));
MUX32_2x1 mux32_2x1_inst(.Y(Y), .I0(shift32_r_res), .I1(shift32_l_res), .S(LnR));

endmodule
```

Image 3.46 Implementation of 32-bit barrel shifter

```
# D:          0 S: 0 LnR:0 Y:          0
# D:          2 S: 4 LnR:1 Y:         32
# D:          5 S: 5 LnR:1 Y:        160
# D:         13 S: 3 LnR:1 Y:        104
# D:2147483648 S: 1 LnR:1 Y:          0
# D:2147483648 S: 3 LnR:1 Y:          0
# D:         64 S: 4 LnR:0 Y:          4
# D:        100 S: 5 LnR:0 Y:          3
# D:        123 S: 3 LnR:0 Y:         15
# D:          1 S: 1 LnR:0 Y:          0
# D:          1 S: 3 LnR:0 Y:          0
```

Image 3.47 32-bit barrel shifter test bench transcript

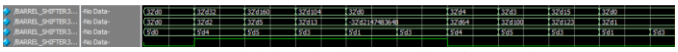

Image 3.48 32-bit barrel shifter test bench waveform

C2. 32-Bit Right Shifter

The 32-bit right shifter is implemented purely using 160 1-bit, 2x1 multiplexers. At each "level" the multiplexers output to the next level multiplexer shifted over by the corresponding number of bits for that level. The following is the digital circuit diagram (actually 4-bit, but the concept extends to 32-bit), implementation, and test bench results for the 32-bit right shifter:

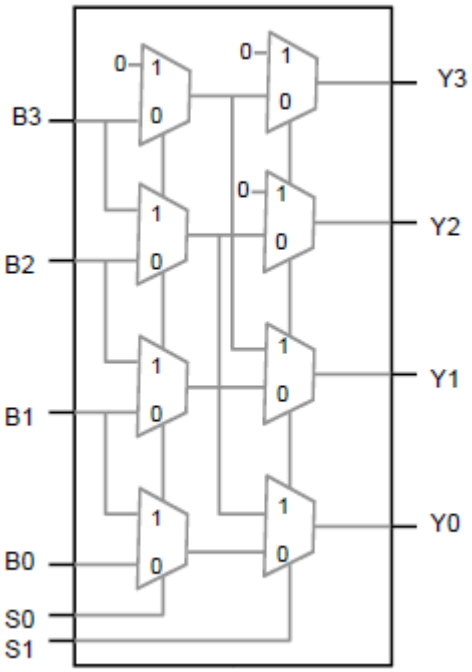

Image 3.49 4-bit right shifter digital circuit diagram

```
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] res [3:0];

genvar i, j;
generate
    for (i = 0; i < 32; i = i + 1)
    begin : first_mux_gen_loop
        if (i == 31)
        begin
            MUX1_2x1 mux_inst_1(.Y(res[0][i]), .I0(D[i]), .I1(1'b0), .S(S[0]));
        end
        else
        begin
            MUX1_2x1 mux_inst_2(.Y(res[0][i]), .I0(D[i]), .I1(D[i + 1]), .S(S[0]));
        end
    end

    for (i = 1; i < 4; i = i + 1)
    begin : semi_inner_mux_gen_loop
        for (j = 0; j < 32; j = j + 1)
        begin : inner_mux_gen_loop
            if (j > 31 - (2 ** i))
            begin
                MUX1_2x1 mux_inst_3(.Y(res[i][j]), .I0(res[i - 1][j]), .I1(1'b0), .S(S[i]));
            end
            else
            begin
                MUX1_2x1 mux_inst_4(.Y(res[i][j]), .I0(res[i - 1][j]), .I1(res[i - 1][j + 2 ** i]), .S(S[i]));
            end
        end
    end

    for (i = 0; i < 32; i = i + 1)
    begin : last_mux_gen_loop
        if (i > 15)
        begin
            MUX1_2x1 mux_inst_5(.Y(Y[i]), .I0(res[3][i]), .I1(1'b0), .S(S[4]));
        end
        else
        begin
            MUX1_2x1 mux_inst_6(.Y(Y[i]), .I0(res[3][i]), .I1(res[3][i + 16]), .S(S[4]));
        end
    end
endgenerate

endmodule
```

Image 3.50 Implementation for 32-bit right shifter

```
# D:          0 S: 0 Y:          0
# D:         64 S: 4 Y:          4
# D:        100 S: 5 Y:          3
# D:        123 S: 3 Y:         15
# D:          1 S: 1 Y:          0
# D:          1 S: 3 Y:          0
```

Image 3.51 32-bit right shifter test bench transcript

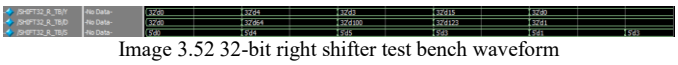

Image 3.52 32-bit right shifter test bench waveform

C3. 32-Bit Left Shifter

The 32-bit left shifter is nearly the same as the 32-bit right shifter, just in the opposite direction. The following is the digital circuit diagram (actually 4-bit, but the concept extends to 32-bit), implementation, and test bench results for the 32-bit left shifter:

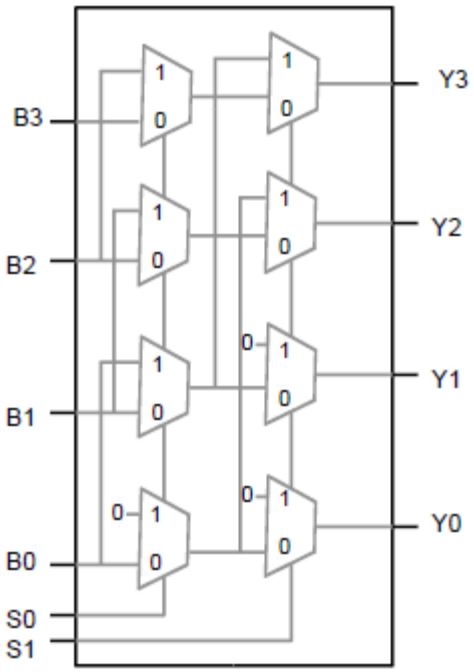

Image 3.53 32-bit left shifter digital circuit diagram

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

wire [31:0] res [3:0];

genvar i, j;
generate
    for (i = 0; i < 32; i = i + 1)
    begin : first_mux_gen_loop
        if (i == 0)
        begin
            MUX1_2x1 mux_inst_1(.Y(res[0][i]), .I0(D[i]), .I1(1'b0), .S(S[0]));
        end
        else
        begin
            MUX1_2x1 mux_inst_2(.Y(res[0][i]), .I0(D[i]), .I1(D[i - 1]), .S(S[0]));
        end
    end

    for (i = 1; i < 4; i = i + 1)
    begin : semi_inner_mux_gen_loop
        for (j = 0; j < 32; j = j + 1)
        begin : inner_mux_gen_loop
            if (j < 2 ** i)
            begin
                MUX1_2x1 mux_inst_3(.Y(res[i][j]), .I0(res[i - 1][j]), .I1(1'b0), .S(S[i]));
            end
            else
            begin
                MUX1_2x1 mux_inst_4(.Y(res[i][j]), .I0(res[i - 1][j]), .I1(res[i - 1][j - 2 ** i]), .S(S[i]));
            end
        end
    end

    for (i = 0; i < 32; i = i + 1)
    begin : last_mux_gen_loop
        if (i < 16)
        begin
            MUX1_2x1 mux_inst_5(.Y(Y[i]), .I0(res[3][i]), .I1(1'b0), .S(S[4]));
        end
        else
        begin
            MUX1_2x1 mux_inst_6(.Y(Y[i]), .I0(res[3][i]), .I1(res[3][i - 16]), .S(S[4]));
        end
    end
endgenerate

endmodule
```

Image 3.54 Implementation of 32-bit left shifter

```
# D:          0 S: 0 Y:          0
# D:          2 S: 4 Y:         32
# D:          5 S: 5 Y:        160
# D:         13 S: 3 Y:        104
# D:2147483648 S: 1 Y:          0
# D:2147483648 S: 3 Y:          0
```

Image 3.55 32-bit left shifter test bench transcript

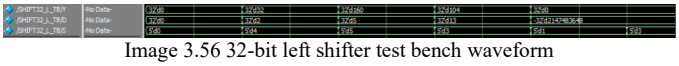

Image 3.56 32-bit left shifter test bench waveform

D. *32-Bit Logic Gates*

The 32-bit logic gates, nor, and, inverter, and or, are very simple. They are implemented through connecting 32 of their respective logic gates together. An exception is the or gate, which could either be implemented the same way, or through connecting a 32-bit nor followed by a 32-bit inverter. The following are the implementations for the 32-bit logic gates:

```
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_32_gen_loop
        nor nor_inst(Y[i], A[i], B[i]);
    end
endgenerate

endmodule
```

Image 3.57 Implementation for 32-bit nor gate

```
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_32_gen_loop
        and and_inst(Y[i], A[i], B[i]);
    end
endgenerate

endmodule
```

Image 3.58 Implementation for 32-bit and gate

```
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_32_gen_loop
        not not_inst(Y[i], A[i]);
    end
endgenerate

endmodule
```

Image 3.59 Implementation for 32-bit inverter

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

wire [31:0] nor_res;

NOR32_2x1 nor32_2x1_inst(.Y(nor_res), .A(A), .B(B));
INV32_1x1 inv32_1x1_inst(.Y(Y), .A(nor_res));

endmodule
```

Image 3.60 Implementation for 32-bit or gate

IV. RF Design, Implementation, and Testing

The RF receives eight inputs: “READ,” “WRITE,” “CLK,” and “RST” which are 1-bit flags indicating respectively the read or write type, the clock signal, and the reset signal, “DATA_W” which is a 32-bit data this is to be written to the RF, and “ADDR_R1,” “ADDR_R2,” and “ADDR_W” which are 5-bit addresses, two of which are for reading and one for writing respectively. The RF has two outputs: “DATA_R1” and “DATA_R2” which are 32-bit data returned when requesting a read operation from the RF. As previously mentioned, all registers in the RF are initially set to value 0. The RF is comprised of 32 32-bit registers, a 5x32 line decoder, two 32-bit 32x1 multiplexers, and two 32-bit 2x1 multiplexers. The line decoder along used alongside and gates and the write signal determine which register among the 32 registers to write data to, if any. The register file is dual read, so it has one 32-bit 32x1 multiplexer to output two read addresses at the same time. The two 32-bit 2x1 multiplexers decide whether to output hi-Z or the actual register data, depending on the read or write status of the register file. The register file is positive edge triggered and resets on a reset signal of zero. The following is the digital circuit diagram, implementation, and test bench results for the RF:

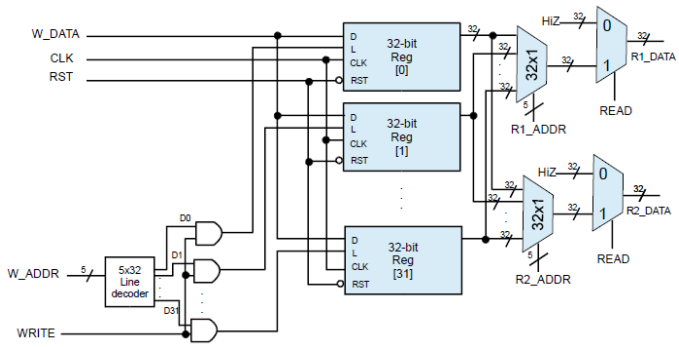

Image 4.1 RF digital circuit diagram

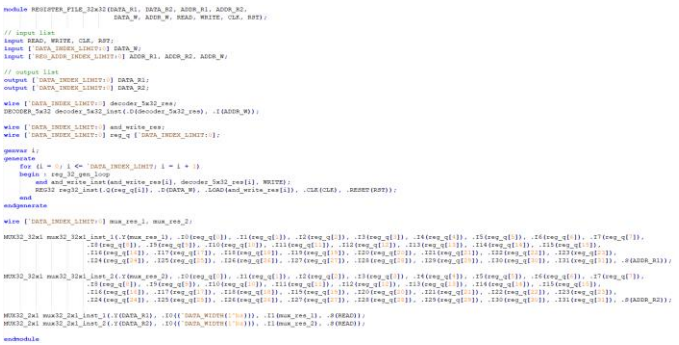

Image 4.2 Implementation of RF

```
#
#          Total number of tests          32
#          Total number of pass           32
#
```

Image 4.3 RF test bench transcript

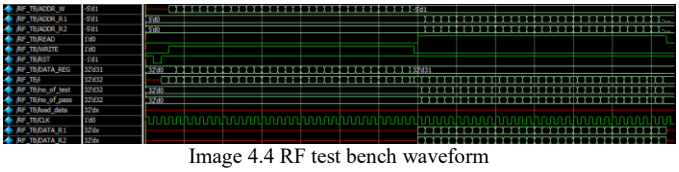

Image 4.4 RF test bench waveform

A. *32-Bit Register*

The 32-bit register is implemented using 32 1-bit registers. It does not have preset capabilities, is positive edge triggered, and is reset on zero reset signal. The following is the digital circuit diagram, implementation, and test bench results for the 32-bit register:

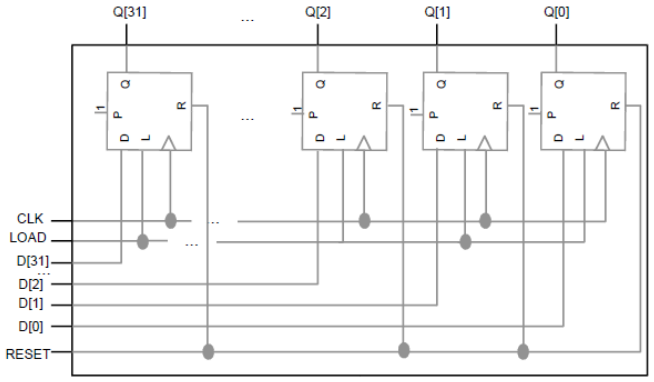

Image 4.5 32-bit register digital circuit diagram

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

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

wire unused;

genvar i;
generate
    for (i = 0; i < 32; i = i + 1)
    begin : reg1_32_gen_loop
        REG1 reg1_inst(.Q(Q[i]), .Qbar(unused), .D(D[i]), .L(LOAD), .C(CLK), .nP(1'b1), .nR(RESET));
    end
endgenerate

endmodule
```

Image 4.6 Implementation for 32-bit register

```
# D:4294967295 LOAD:1 CLK:0 RESET:1 Q:          x
# D:4294967295 LOAD:1 CLK:1 RESET:1 Q:4294967295
# D:          x LOAD:1 CLK:0 RESET:1 Q:4294967295
# D:          x LOAD:1 CLK:1 RESET:0 Q:          0
# D:4294967295 LOAD:1 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:1 CLK:1 RESET:1 Q:4294967295
# D:          0 LOAD:1 CLK:0 RESET:1 Q:4294967295
# D:          0 LOAD:1 CLK:1 RESET:1 Q:          0
# D:          x LOAD:1 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:0 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:0 CLK:1 RESET:1 Q:          0
# D:          x LOAD:0 CLK:0 RESET:1 Q:          0
# D:          x LOAD:0 CLK:1 RESET:0 Q:          0
# D:4294967295 LOAD:0 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:0 CLK:1 RESET:1 Q:          0
# D:          0 LOAD:0 CLK:0 RESET:1 Q:          0
# D:          0 LOAD:0 CLK:1 RESET:1 Q:          0
# D:          x LOAD:0 CLK:0 RESET:1 Q:          0
```

Image 4.7 32-bit register test bench transcript

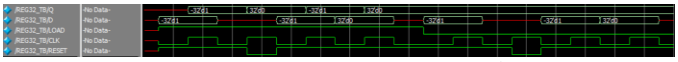

Image 4.8 32-bit register test bench waveform

A1. 1-Bit Register

The 1-bit register is implemented using a D-flipflop and a multiplexer. This register has the ability to not always load everything that is inputted because it has a load signal. The register only saves a new value when the load signal is on, otherwise the same value is looped back into the register. The following is the digital circuit diagram, implementation, and test bench results for the 1-bit register:

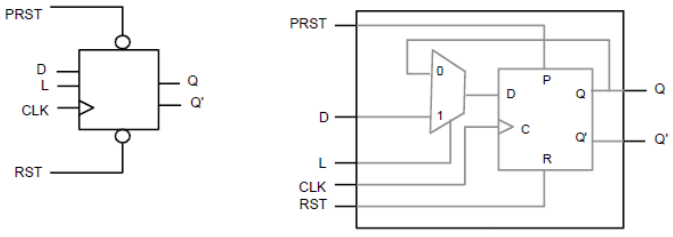

Image 4.9 1-bit register digital circuit diagram

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

wire mux_res;
wire d_ff_q;

MUX1_2x1 mux1_2x1_inst(.Y(mux_res), .I0(d_ff_q), .I1(D), .S(L));
D_FF d_ff_inst(.Q(d_ff_q), .Qbar(Qbar), .D(mux_res), .C(C), .nP(nP), .nR(nR));

buf buf_inst(Q, d_ff_q);

endmodule
```

Image 4.10 Implementation of 1-bit register

```
# D:1 C:0 L:1 nP:1 nR:1 Q:x Qbar:x
# D:1 C:1 L:1 nP:1 nR:1 Q:1 Qbar:0
# D:x C:0 L:1 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:0 L:1 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:1 L:1 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 L:1 nP:1 nR:1 Q:0 Qbar:1
# D:x C:1 L:1 nP:0 nR:1 Q:1 Qbar:0
# D:x C:1 L:1 nP:1 nR:0 Q:0 Qbar:1
# D:1 C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:1 C:1 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:0 C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:0 C:1 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:1 L:0 nP:0 nR:1 Q:1 Qbar:0
# D:x C:1 L:0 nP:1 nR:0 Q:0 Qbar:1
```

Image 4.11 1-bit register test bench transcript

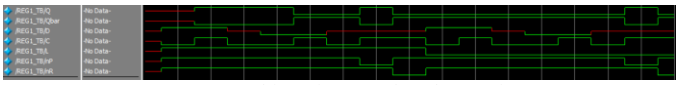

Image 4.12 1-bit register test bench waveform

A2. D-Flipflop

The D-flipflop is implemented using a D-latch and an SR-latch. The D-latch is the master latch and the SR-latch is the slave latch. The following is the digital circuit diagram, implementation, and test bench results for the D-flipflop:

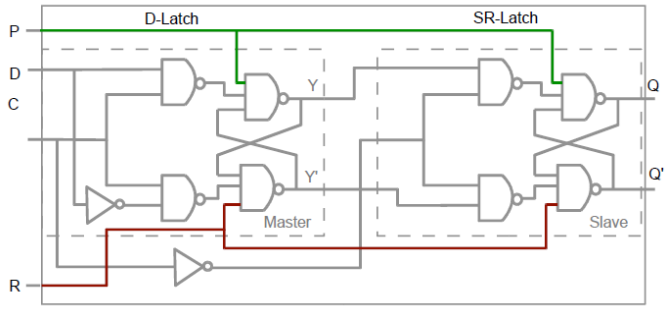

Image 4.13 D-Flipflop digital circuit diagram

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

wire Y, Ybar;
wire not_c;

not not_inst(not_c, C);

D_LATCH d_latch_inst(.Q(Y), .Qbar(Ybar), .D(D), .C(not_c), .nP(nP), .nR(nR));
SR_LATCH sr_latch_inst(.Q(Q), .Qbar(Qbar), .S(Y), .R(Ybar), .C(C), .nP(nP), .nR(nR));

endmodule
```

Image 4.14 Implementation of D-Flipflop

```
# D:1 C:0 nP:1 nR:1 Q:x Qbar:x
# D:1 C:1 nP:1 nR:1 Q:1 Qbar:0
# D:x C:0 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:0 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:1 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:1 nP:0 nR:1 Q:1 Qbar:0
# D:x C:1 nP:1 nR:0 Q:0 Qbar:1
```

Image 4.15 D-Flipflop test bench transcript

Image 4.16 D-Flipflop test bench waveform

A3. D-Latch

The D-latch is implemented using an SR-latch, just using an inverter to invert the input signal instead of having to receive two separate inputs like the SR-latch. The following is the digital circuit diagram, implementation, and test bench results for the D-latch:

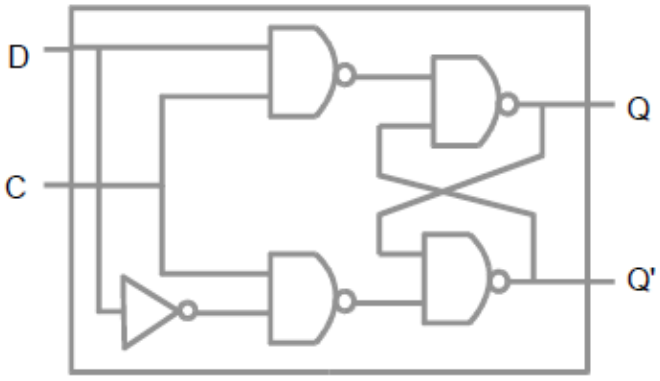

Image 4.17 D-latch digital circuit diagram

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

wire not_d;

not not_inst(not_d, D);

SR_LATCH sr_latch_inst(.Q(Q), .Qbar(Qbar), .S(D), .R(not_d), .C(C), .nP(nP), .nR(nR));

endmodule
```

Image 4.18 Implementation of D-latch

```
# D:1 C:1 nP:1 nR:1 Q:1 Qbar:0
# D:x C:0 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:1 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:1 nP:0 nR:1 Q:1 Qbar:x
# D:x C:1 nP:1 nR:0 Q:x Qbar:1
```

Image 4.19 D-latch test bench transcript

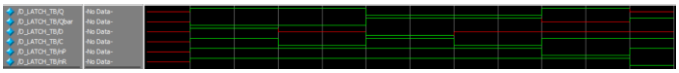

Image 4.20 D-latch test bench waveform

A4. SR-Latch

The SR-latch is implemented using four nand gates. It receives two data inputs and one control signal. The following is the digital circuit diagram, implementation, and test bench results for the SR-latch:

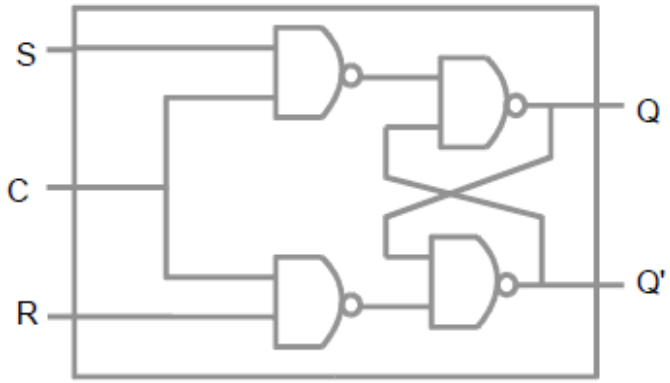

Image 4.21 SR-latch digital circuit diagram

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

wire nand_sc_res;
wire nand_rc_res;
wire nand_sQbar_res;
wire nand_rq_res;

nand nand_sc_inst(nand_sc_res, S, C);
nand nand_rc_inst(nand_rc_res, R, C);

NAND_3x1 nand_3x1_sQbar_inst(.Y(nand_sQbar_res), .A(nP), .B(nand_sc_res), .C(nand_rq_res));
NAND_3x1 nand_3x1_rq_inst(.Y(nand_rq_res), .A(nR), .B(nand_rc_res), .C(nand_sQbar_res));

buf buf_inst_1(Q, nand_sQbar_res);
buf buf_inst_2(Qbar, nand_rq_res);

endmodule
```

Image 4.22 Implementation of SR-latch

```
# S:1 R:0 C:1 nP:1 nR:1 Q:1 Qbar:0
# S:x R:x C:0 nP:1 nR:1 Q:1 Qbar:0
# S:0 R:0 C:1 nP:1 nR:1 Q:1 Qbar:0
# S:0 R:1 C:1 nP:1 nR:1 Q:0 Qbar:1
# S:x R:x C:0 nP:1 nR:1 Q:0 Qbar:1
# S:0 R:0 C:1 nP:1 nR:1 Q:0 Qbar:1
# S:x R:x C:x nP:0 nR:1 Q:1 Qbar:x
# S:x R:x C:x nP:1 nR:0 Q:x Qbar:1
```

Image 4.23 SR-latch test bench transcript

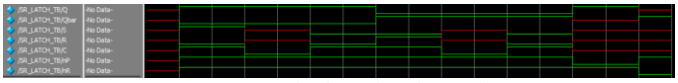

Image 4.24 SR-latch test bench waveform

B. Line Decoder

The base 2x4 line decoder is implemented with four and gates and two inverters. The following is the digital circuit diagram, implementation, and test bench results for the 2x4 line decoder:

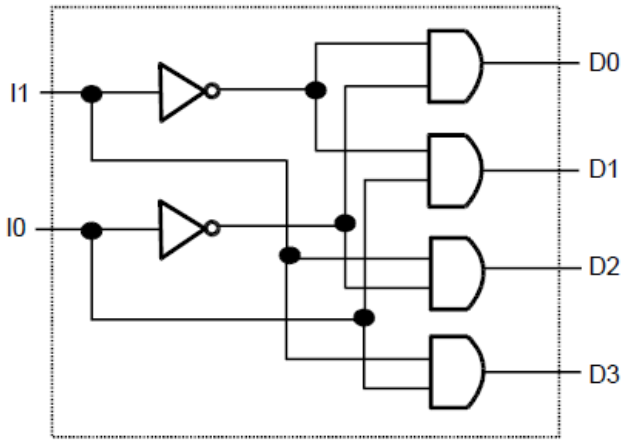

Image 4.25 2x4 line decoder digital circuit diagram

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

wire [1:0] I_not;

not not_inst_1(I_not[0], I[0]);
not not_inst_2(I_not[1], I[1]);

and and_inst_1(D[0], I_not[1], I_not[0]);
and and_inst_2(D[1], I_not[1], I[0]);
and and_inst_3(D[2], I[1], I_not[0]);
and and_inst_4(D[3], I[1], I[0]);

endmodule
```

Image 4.26 Implementation of 2x4 line decoder

```
# I:0 D:0001
# I:1 D:0010
# I:2 D:0100
# I:3 D:1000
```

Image 4.27 2x4 line decoder test bench transcript

Image 4.28 2x4 line decoder test bench waveform

Subsequent larger line decoders are implemented using a smaller decoder, an invert, and the corresponding number of and gates for the number outputs. The following is the digital

circuit diagram, implementation, and test bench results for the 5x32 line decoder:

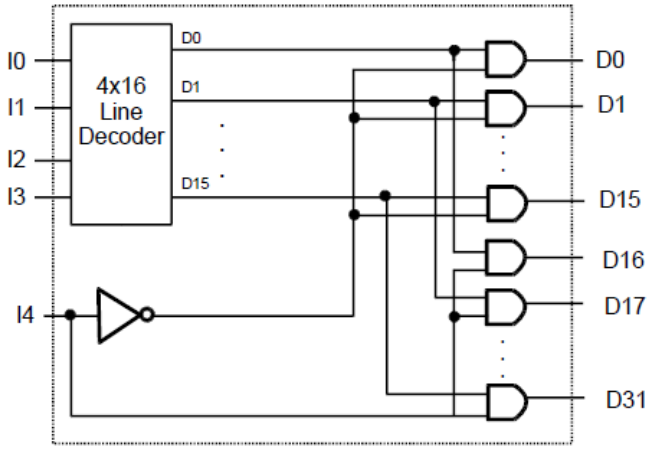

Image 4.29 5x32 line decoder digital circuit diagram

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

wire I4_not;
not not_inst(I4_not, I[4]);

wire [15:0] decoder_4x16_res;
DECODER_4x16 decoder_4x16_inst(.D(decoder_4x16_res), .I(I[3:0]));

genvar i;
generate
    for (i = 0; i < 16; i = i + 1)
    begin : and_32_gen_loop
        and and_inst_1(D[i], decoder_4x16_res[i], I4_not);
        and and_inst_2(D[i + 16], decoder_4x16_res[i], I[4]);
    end
endgenerate

endmodule
```

Image 4.30 Implementation of 5x32 line decoder

```
# I: 0 D:00000000000000000000000000000001
# I: 1 D:00000000000000000000000000000010
# I: 2 D:00000000000000000000000000000100
# I: 3 D:00000000000000000000000000001000
# I: 4 D:00000000000000000000000000010000
# I: 5 D:00000000000000000000000000100000
# I: 6 D:00000000000000000000000001000000
# I: 7 D:00000000000000000000000010000000
# I: 8 D:00000000000000000000000100000000
# I: 9 D:00000000000000000000001000000000
# I:10 D:00000000000000000000010000000000
# I:11 D:00000000000000000000100000000000
# I:12 D:00000000000000000001000000000000
# I:13 D:00000000000000000010000000000000
# I:14 D:00000000000000000100000000000000
# I:15 D:00000000000000001000000000000000
# I:16 D:00000000000000010000000000000000
# I:17 D:00000000000000100000000000000000
# I:18 D:00000000000001000000000000000000
# I:19 D:00000000000010000000000000000000
# I:20 D:00000000000100000000000000000000
# I:21 D:00000000001000000000000000000000
# I:22 D:00000000010000000000000000000000
# I:23 D:00000000100000000000000000000000
# I:24 D:00000001000000000000000000000000
# I:25 D:00000010000000000000000000000000
# I:26 D:00000100000000000000000000000000
# I:27 D:00001000000000000000000000000000
# I:28 D:00010000000000000000000000000000
# I:29 D:00100000000000000000000000000000
# I:30 D:01000000000000000000000000000000
# I:31 D:10000000000000000000000000000000
```

Image 4.31 5x32 line decoder test bench transcript

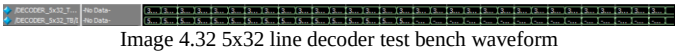

Image 4.32 5x32 line decoder test bench waveform

V. Memory Design, Implementation, and Testing

The memory contains more memory than the RF and is single read instead of dual read. It receives five inputs: “READ,” “WRITE,” “CLK,” and “RST” which are 1-bit flags respectively indicating the read or write operation, the clock signal, and the reset signal and “ADDR” which is a 26-bit address for reading or writing. It also has one 32-bit inout port, “DATA,” for transferring data that is read or to be written. The actual 256MB memory is represented by “sram_32x64m,” which is an array of about 64M elements each of 32-bit size. The memory is then wrapped with a memory wrapper with separate input and output ports instead of the memory’s one inout port. The following is the digital circuit diagram, implementation, and test bench results for the memory wrapper and 64MB memory:

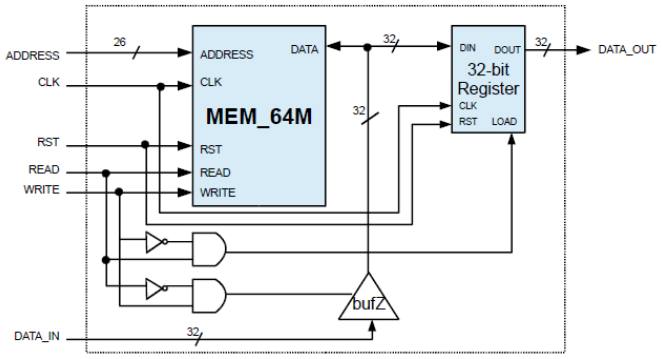

Image 5.1 Memory wrapper digital circuit diagram

```
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

endmodule
```

Image 5.2 Implementation of memory wrapper

```
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
```

Image 5.3 Implementation of 64MB memory

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

Image 5.4 64MB memory test bench transcript

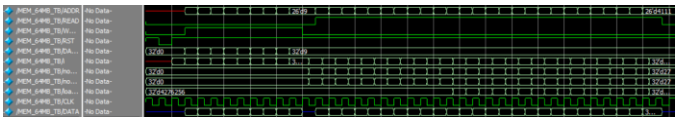

Image 5.5 64MB memory test bench waveform

VI. CU Design, Implementation, and Testing

The CU, using a five-state state machine, sends a control word to the data path to execute the current stage of the instruction cycle. It also sends signals for reading and writing to and from the memory. The CU does not have an independent test bench, as it must work together with the data path and memory in order to yield results. The following is the implementation for the CU:

```
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;
reg READ, WRITE;
reg [`DATA_INDEX_LIMIT:0] INSTR;

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

always @ (proc_state)
begin
    if (proc_state === `PROC_FETCH)
    begin
        READ = 1'b1;
        WRITE = 1'b0;
        CTRL = `CTRL_WIDTH'h00200000;
    end

    else if (proc_state === `PROC_DECODE)
    begin
        INSTR = INSTRUCTION;
        if (INSTR[31:26] === 6'h1b) // push
        begin
            CTRL = `CTRL_WIDTH'h00000070;
        end
        else if (INSTR[31:26] === 6'h0f || INSTR[31:26] === 6'h02 ||
                 INSTR[31:26] === 6'h03 || INSTR[31:26] === 6'h1c) // lui, jmp, jal, pop
        begin
            CTRL = `CTRL_WIDTH'h00000010;
        end
        else
        begin
            CTRL = `CTRL_WIDTH'h00000050; // everything else
        end
    end
```

Image 6.1 Implementation of CU (fetch and decode)

```
else if (proc_state === `PROC_EXE)
begin
    /* R-Type */
    if (INSTR[31:26] === 6'h00)
    begin
        if (INSTR[5:0] === 6'h20) // add
        begin
            CTRL = `CTRL_WIDTH'h00006000;
        end
        else if (INSTR[5:0] === 6'h22) // sub
        begin
            CTRL = `CTRL_WIDTH'h0000A000;
        end
        else if (INSTR[5:0] === 6'h2c) // mul
        begin
            CTRL = `CTRL_WIDTH'h0000E000;
        end
        else if (INSTR[5:0] === 6'h24) // and
        begin
            CTRL = `CTRL_WIDTH'h0001A000;
        end
        else if (INSTR[5:0] === 6'h25) // or
        begin
            CTRL = `CTRL_WIDTH'h0001E000;
        end
        else if (INSTR[5:0] === 6'h27) // nor
        begin
            CTRL = `CTRL_WIDTH'h00022000;
        end
        else if (INSTR[5:0] === 6'h2a) // slt
        begin
            CTRL = `CTRL_WIDTH'h00026000;
        end
        else if (INSTR[5:0] === 6'h01) // sll
        begin
            CTRL = `CTRL_WIDTH'h00015400;
        end
        else if (INSTR[5:0] === 6'h02) // srl
        begin
            CTRL = `CTRL_WIDTH'h00011400;
        end
        else if (INSTR[5:0] === 6'h08) // jr
        begin
            CTRL = `CTRL_WIDTH'h00000000;
        end
    end
```

Image 6.2 Implemenation of CU (execute)

```
else // I-Type and J-Type
begin
    /* I-Type */
    if (INSTR[31:26] === 6'h08) // addi
    begin
        CTRL = `CTRL_WIDTH'h00004800;
    end
    else if (INSTR[31:26] === 6'h1d) // muli
    begin
        CTRL = `CTRL_WIDTH'h0000C800;
    end
    else if (INSTR[31:26] === 6'h0c) // andi
    begin
        CTRL = `CTRL_WIDTH'h00018000;
    end
    else if (INSTR[31:26] === 6'h0d) // ori
    begin
        CTRL = `CTRL_WIDTH'h0001C000;
    end
    else if (INSTR[31:26] === 6'h0f) // lui
    begin
        CTRL = `CTRL_WIDTH'h00000000;
    end
    else if (INSTR[31:26] === 6'h0a) // stli
    begin
        CTRL = `CTRL_WIDTH'h00024800;
    end
    else if (INSTR[31:26] === 6'h04) // beq
    begin
        CTRL = `CTRL_WIDTH'h0000A000;
    end
    else if (INSTR[31:26] === 6'h05) // bne
    begin
        CTRL = `CTRL_WIDTH'h0000A000;
    end
    else if (INSTR[31:26] === 6'h23) // lw
    begin
        CTRL = `CTRL_WIDTH'h00004800;
    end
    else if (INSTR[31:26] === 6'h2b) // sw
    begin
        CTRL = `CTRL_WIDTH'h00004800;
    end
```

Image 6.3 Implementation of CU (excute)

```
    /* J-Type */
    else if (INSTR[31:26] === 6'h02) // jmp
    begin
        CTRL = `CTRL_WIDTH'h00000000;
    end
    else if (INSTR[31:26] === 6'h03) // jal
    begin
        CTRL = `CTRL_WIDTH'h00000000;
    end
    else if (INSTR[31:26] === 6'h1b) // push
    begin
        CTRL = `CTRL_WIDTH'h00009200;
    end
    else if (INSTR[31:26] === 6'h1c) // pop
    begin
        CTRL = `CTRL_WIDTH'h00005300;
    end
```

Image 6.4 Implementation of CU (execute)

```
else if (proc_state === `PROC_MEM)
begin
    /* R-Type */
    if (INSTR[31:26] === 6'h00)
    begin
        if (INSTR[5:0] === 6'h20) // add
        begin
            CTRL = `CTRL_WIDTH'h00006000;
        end
        else if (INSTR[5:0] === 6'h22) // sub
        begin
            CTRL = `CTRL_WIDTH'h0000A000;
        end
        else if (INSTR[5:0] === 6'h2c) // mul
        begin
            CTRL = `CTRL_WIDTH'h0000E000;
        end
        else if (INSTR[5:0] === 6'h24) // and
        begin
            CTRL = `CTRL_WIDTH'h0001A000;
        end
        else if (INSTR[5:0] === 6'h25) // or
        begin
            CTRL = `CTRL_WIDTH'h0001E000;
        end
        else if (INSTR[5:0] === 6'h27) // nor
        begin
            CTRL = `CTRL_WIDTH'h00022000;
        end
        else if (INSTR[5:0] === 6'h2a) // slt
        begin
            CTRL = `CTRL_WIDTH'h00026000;
        end
        else if (INSTR[5:0] === 6'h01) // sll
        begin
            CTRL = `CTRL_WIDTH'h00015400;
        end
        else if (INSTR[5:0] === 6'h02) // srl
        begin
            CTRL = `CTRL_WIDTH'h00011400;
        end
        else if (INSTR[5:0] === 6'h08) // jr
        begin
            CTRL = `CTRL_WIDTH'h00000000;
        end
    end
```

Image 6.5 Implementation of CU (memory)

```
else // I-Type and J-Type
begin
    /* I-Type */
    if (INSTR[31:26] === 6'h08) // addi
    begin
        CTRL = `CTRL_WIDTH'h00004800;
    end
    else if (INSTR[31:26] === 6'h1d) // muli
    begin
        CTRL = `CTRL_WIDTH'h0000C800;
    end
    else if (INSTR[31:26] === 6'h0c) // andi
    begin
        CTRL = `CTRL_WIDTH'h00018000;
    end
    else if (INSTR[31:26] === 6'h0d) // ori
    begin
        CTRL = `CTRL_WIDTH'h0001C000;
    end
    else if (INSTR[31:26] === 6'h0f) // lui
    begin
        CTRL = `CTRL_WIDTH'h00000000;
    end
    else if (INSTR[31:26] === 6'h0a) // stli
    begin
        CTRL = `CTRL_WIDTH'h00024800;
    end
    else if (INSTR[31:26] === 6'h04) // beq
    begin
        CTRL = `CTRL_WIDTH'h0000A000;
    end
    else if (INSTR[31:26] === 6'h05) // bne
    begin
        CTRL = `CTRL_WIDTH'h0000A000;
    end
    else if (INSTR[31:26] === 6'h23) // lw
    begin
        READ = 1'b1;
        WRITE = 1'b0;
        CTRL = `CTRL_WIDTH'h00004800;
    end
    else if (INSTR[31:26] === 6'h2b) // sw
    begin
        READ = 1'b0;
        WRITE = 1'b1;
        CTRL = `CTRL_WIDTH'h00004800;
    end
```

Image 6.6 Implementation of CU (memory)

```
    /* J-Type */
    else if (INSTR[31:26] === 6'h02) // jmp
    begin
        CTRL = `CTRL_WIDTH'h00000000;
    end
    else if (INSTR[31:26] === 6'h03) // jal
    begin
        CTRL = `CTRL_WIDTH'h00000000;
    end
    else if (INSTR[31:26] === 6'h1b) // push
    begin
        READ = 1'b0;
        WRITE = 1'b1;
        CTRL = `CTRL_WIDTH'h00509200;
    end
    else if (INSTR[31:26] === 6'h1c) // pop
    begin
        READ = 1'b1;
        WRITE = 1'b0;
        CTRL = `CTRL_WIDTH'h00100000;
    end
```

Image 6.7 Implementation of CU (memory)

```
else if (proc_state === `PROC_WB)
begin
    /* R-Type */
    if (INSTR[31:26] === 6'h00)
    begin
        if (INSTR[5:0] === 6'h20) // add
        begin
            CTRL = `CTRL_WIDTH'h1200608B;
        end
        else if (INSTR[5:0] === 6'h22) // sub
        begin
            CTRL = `CTRL_WIDTH'h1200A08B;
        end
        else if (INSTR[5:0] === 6'h2c) // mul
        begin
            CTRL = `CTRL_WIDTH'h1200E08B;
        end
        else if (INSTR[5:0] === 6'h24) // and
        begin
            CTRL = `CTRL_WIDTH'h1201A08B;
        end
        else if (INSTR[5:0] === 6'h25) // or
        begin
            CTRL = `CTRL_WIDTH'h1201E08B;
        end
        else if (INSTR[5:0] === 6'h27) // nor
        begin
            CTRL = `CTRL_WIDTH'h1202208B;
        end
        else if (INSTR[5:0] === 6'h2a) // slt
        begin
            CTRL = `CTRL_WIDTH'h1202608B;
        end
        else if (INSTR[5:0] === 6'h01) // sll
        begin
            CTRL = `CTRL_WIDTH'h1201548B;
        end
        else if (INSTR[5:0] === 6'h02) // srl
        begin
            CTRL = `CTRL_WIDTH'h1201148B;
        end
        else if (INSTR[5:0] === 6'h08) // jr
        begin
            CTRL = `CTRL_WIDTH'h00000009;
        end
    end
```

Image 6.8 Implementation of CU (write back)

```
    if (INSTR[31:26] === 6'h08) // addi
    begin
        CTRL = `CTRL_WIDTH'h1600488B;
    end
    else if (INSTR[31:26] === 6'h1d) // muli
    begin
        CTRL = `CTRL_WIDTH'h1600C88B;
    end
    else if (INSTR[31:26] === 6'h0c) // andi
    begin
        CTRL = `CTRL_WIDTH'h1601808B;
    end
    else if (INSTR[31:26] === 6'h0d) // ori
    begin
        CTRL = `CTRL_WIDTH'h1601C08B;
    end
    else if (INSTR[31:26] === 6'h0f) // lui
    begin
        CTRL = `CTRL_WIDTH'h1700008B;
    end
    else if (INSTR[31:26] === 6'h0a) // stli
    begin
        CTRL = `CTRL_WIDTH'h1602088B;
    end
    else if (INSTR[31:26] === 6'h04) // beq
    begin
        if (ZERO === 1'b0)
        begin
            CTRL = `CTRL_WIDTH'h0000A00D;
        end
        else
        begin
            CTRL = `CTRL_WIDTH'h0000A00B;
        end
    end
    else if (INSTR[31:26] === 6'h05) // bne
    begin
        if (ZERO !== 1'b0)
        begin
            CTRL = `CTRL_WIDTH'h0000A00D;
        end
        else
        begin
            CTRL = `CTRL_WIDTH'h0000A00B;
        end
    end
    else if (INSTR[31:26] === 6'h23) // lw
    begin
        CTRL = `CTRL_WIDTH'h1680008B;
    end
    else if (INSTR[31:26] === 6'h2b) // sw
    begin
        CTRL = `CTRL_WIDTH'h0000000B;
    end
```

Image 6.9 Implementation of CU (write back)

```
/* J-Type */
else if (INSTR[31:26] === 6'h02) // jmp
begin
    CTRL = `CTRL_WIDTH'h00000001;
end
else if (INSTR[31:26] === 6'h03) // jal
begin
    CTRL = `CTRL_WIDTH'h08000081;
end
else if (INSTR[31:26] === 6'h1b) // push
begin
    CTRL = `CTRL_WIDTH'h0000930B;
end
else if (INSTR[31:26] === 6'h1c) // pop
begin
    CTRL = `CTRL_WIDTH'h0280008B;
end
```

Image 6.10 Implementation of CU (write back)

A. *State Machine*

The CU uses a five-state state machine to cycle between the instruction cycle: fetch (“PROC_FETCH”), decode (“PROC_DECODE”), execute (“PROC_EXE”), memory (“PROC_MEM”), and write back (“PROC_WB”). The state machine receives two inputs: “CLK” and “RST” which are 1-bit signals representing the clock signal and reset signal respectively. The state machine has one output “STATE” which is a 3-bit indication of the current state. The current state (“STATE”) and next state (“next_state”) are both represented internally with 3-bit registers. The same state machine is used as in the complete behavioral model, DaVinci v1.0. The following is the implementation for the state machine:

```
module PROC_SM(STATE, CLK, RST);
// list of inputs
input CLK, RST;
// list of outputs
output [2:0] STATE;

// Registers for current and next states
reg [2:0] STATE;
reg [2:0] next_state;

// Initial state is unknown and next state is fetch
initial
begin
    STATE = 3'bxxx;
    next_state = `PROC_FETCH;
end

// On negative edge of reset, state is unknown and next state is fetch
always @ (negedge RST)
begin
    STATE = 3'bxxx;
    next_state = `PROC_FETCH;
end

// On positive edge of clock, change states
always @ (posedge CLK)
begin
    STATE = next_state;

    case (STATE)
        `PROC_FETCH: next_state = `PROC_DECODE;
        `PROC_DECODE: next_state = `PROC_EXE;
        `PROC_EXE: next_state = `PROC_MEM;
        `PROC_MEM: next_state = `PROC_WB;
        `PROC_WB: next_state = `PROC_FETCH;
    endcase
end

endmodule
```

Image 6.11 Implementation of state machine

B. *Control Instructions*

Control instructions are derived by analyzing the data path and determining which control signals need to be activated during which state of the instruction cycle. Each control signal can be assigned to each data path component arbitrarily, but I followed the convention Professor Patra used in the Lecture 11 microinstruction tutorial video. The following are the derivations of the control words:

INSTRUCTION: add rd, rs, rt | R[rd] = R[rs] + R[rt]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	0		0		0		0		1	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	1	B
CTRL[2]	pc_sel_2	0		0		0		0		0	
CTRL[3]	pc_sel_3	0		0		0		0		1	
CTRL[4]	ir_load	0		1		0		0		0	
CTRL[5]	r1_sel_1	0	0	0	5	0	0	0	0	0	8
CTRL[6]	reg_r	0		1		0		0		0	
CTRL[7]	reg_w	0		0		0		0		1	
CTRL[8]	sp_load	0		0		0		0		0	
CTRL[9]	op1_sel_1	0	0	0	0	0	0	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	1	6	1	6	1	6
CTRL[14]	alu_oprn[0]	0		0		1		1		1	
CTRL[15]	alu_oprn[1]	0		0		0		0		0	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	1	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	1	2
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	1	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00006000		0x00006000		0x1200608B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.12 Control derivation for add

INSTRUCTION: sub rd, rs, rt | R[rd] = R[rs] - R[rt]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	0		0		0		0		1	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	1	B
CTRL[2]	pc_sel_2	0		0		0		0		0	
CTRL[3]	pc_sel_3	0		0		0		0		1	
CTRL[4]	ir_load	0		1		0		0		0	
CTRL[5]	r1_sel_1	0	0	0	5	0	0	0	0	0	8
CTRL[6]	reg_r	0		1		0		0		0	
CTRL[7]	reg_w	0		0		0		0		1	
CTRL[8]	sp_load	0		0		0		0		0	
CTRL[9]	op1_sel_1	0	0	0	0	0	0	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	1	A	1	A	1	A
CTRL[14]	alu_oprn[0]	0		0		0		0		0	
CTRL[15]	alu_oprn[1]	0		0		1		1		1	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	1	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	1	2
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	1	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0000A000		0x0000A000		0x1200A08B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.13 Control derivation for sub

INSTRUCTION: mul rd, rs, rt | R[rd] = R[rs] * R[rt]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	**1**	E	**1**	E	**1**	E
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**1**		**1**		**1**	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		**0**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0000E000		0x0000E000		0x1200E08B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.14 Control derivation for mul

INSTRUCTION: srl rd, rs, shamt | R[rd] = R[rs] >> shamt

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	4	**0**	4	**0**	4
CTRL[10]	op2_sel_1	0		0		**1**		**1**		**1**	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		**1**		**1**		**1**	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[14]	alu_oprn[0]	0		0		**0**		**0**		**0**	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		**0**	
CTRL[16]	alu_oprn[2]	0		0		**1**		**1**		**1**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00011400		0x00011400		0x1201148B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.15 Control derivation for srl

INSTRUCTION: sll rd, rs, shamt | R[rd] = R[rs] << shamt

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	4	**0**	4	**0**	4
CTRL[10]	op2_sel_1	0		0		**1**		**1**		**1**	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		**1**		**1**		**1**	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	5	**0**	5	**0**	5
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		**0**	
CTRL[16]	alu_oprn[2]	0		0		**1**		**1**		**1**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00015400		0x00015400		0x1201548B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.16 Control derivation for sll

INSTRUCTION: and rd, rs, rt | R[rd] = R[rs] & R[rt]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	**1**	A	**1**	A	**1**	A
CTRL[14]	alu_oprn[0]	0		0		**0**		**0**		**0**	
CTRL[15]	alu_oprn[1]	0		0		**1**		**1**		**1**	
CTRL[16]	alu_oprn[2]	0		0		**1**		**1**		**1**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0001A000		0x0001A000		0x1201A08B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.17 Control derivation for and

INSTRUCTION: or rd, rs, rt | R[rd] = R[rs] | R[rt]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	**1**	E	**1**	E	**1**	E
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**1**		**1**		**1**	
CTRL[16]	alu_oprn[2]	0		0		**1**		**1**		**1**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0001E000		0x0001E000		0x1201E08B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.18 Control derivation for or

INSTRUCTION: nor rd, rs, rt | R[rd] = ~(R[rs] | R[rt])

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	**1**	2	**1**	2	**1**	2
CTRL[14]	alu_oprn[0]	0		0		**0**		**0**		**0**	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		**0**	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		**0**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**1**	2	**1**	2	**1**	2
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00022000		0x00022000		0x1202208B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.19 Control derivation for nor

INSTRUCTION: slt rd, rs, rt | R[rd] = (R[rs] < R[rt]) ? 1 : 0

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	**1**	6	**1**	6	**1**	6
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		**0**	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		**0**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**1**	2	**1**	2	**1**	2
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	2
CTRL[26]	wa_sel_1	0		0		0		0		**0**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00026000		0x00026000		0x1202608B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.20 Control derivation for slt

INSTRUCTION: jr rd, rs, rt | PC = R[rs]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**0**	9
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	0
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**0**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	0	0	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	0	0	0	0	0	0
CTRL[14]	alu_oprn[0]	0		0		0		0		0	
CTRL[15]	alu_oprn[1]	0		0		0		0		0	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	0	0
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00000000		0x00000000		0x00000009	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.21 Control derivation for jr

INSTRUCTION: addi rt, rs, imm | R[rt] = R[rs] + SignExtImm

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	8	**0**	8	**0**	8
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**1**		**1**		**1**	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		**0**	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	4	**0**	4	**0**	4
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		**0**	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		**0**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	6
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00004800		0x00004800		0x1604488B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.22 Control derivation for addi

INSTRUCTION: muli rt, rs, imm | R[rt] = R[rs] * SignExtImm

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**1**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	8	**0**	8	**0**	8
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**1**		**1**		**1**	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		**0**	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	C	**0**	C	**0**	C
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		**0**	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		**0**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	6
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0000C800		0x0000C800		0x1600C88B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.23 Control derivation for muli

INSTRUCTION: andi rt, rs, imm | R[rt] = R[rs] & ZeroExtImm

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**0**		**0**		**0**	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		**0**	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	8	**0**	8	**0**	8
CTRL[14]	alu_oprn[0]	0		0		**0**		**0**		**0**	
CTRL[15]	alu_oprn[1]	0		0		**1**		**1**		**1**	
CTRL[16]	alu_oprn[2]	0		0		**1**		**1**		**1**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	6
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00018000		0x00018000		0x1601808B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.24 Control derivation for andi

INSTRUCTION: ori rt, rs, imm | R[rt] = R[rs] | ZeroExtImm

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**0**		**0**		**0**	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		**0**	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	C	**0**	C	**0**	C
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		**1**	
CTRL[15]	alu_oprn[1]	0		0		**1**		**1**		**1**	
CTRL[16]	alu_oprn[2]	0		0		**1**		**1**		**1**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	1	**0**	1	**0**	1
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		**0**	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	6
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0001C000		0x0001C000		0x1601C08B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.25 Control derivation for ori

INSTRUCTION: lui rt, rs, imm | R[rt] = {imm, 16'b0}

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	0	1	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**0**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	0	0	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	0	0	0	0	0	0
CTRL[14]	alu_oprn[0]	0		0		0		0		0	
CTRL[15]	alu_oprn[1]	0		0		0		0		0	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		**1**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	7
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000010		0x00000000		0x00000000		0x1700008B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.26 Control derivation for lui

INSTRUCTION: slti rt, rs, imm | R[rt] = (R[rs] < SignExtImm) ? 1 : 0

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	8	**0**	8	**0**	8
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**1**		**1**		0	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		0	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	4	**0**	4	0	0
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		0	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		0	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	**1**	2	**1**	2	**1**	2
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**0**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	6
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	**1**	1
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00024800		0x00024800		0x1602088B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.27 Control derivation for slti

INSTRUCTION: beq or bne rt, rs, imm | if (R[rs] == or != R[rt]) PC = PC + 1 + SignExtImm

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX	If false: WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	0	D	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**1**		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	0	0	0
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**0**		**0**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	0	**0**	0	**0**	0	**0**	0
CTRL[10]	op2_sel_1	0		0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	**1**	A	**1**	A	**1**	A	**1**	A
CTRL[14]	alu_oprn[0]	0		0		**0**		**0**		0		0	
CTRL[15]	alu_oprn[1]	0		0		**1**		**1**		**1**		**1**	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		**0**		**0**	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	0	**0**	0	**0**	0	**0**	0
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		**0**		**0**	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	0	0	0	0
CTRL[26]	wa_sel_1	0		0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		0		0		0	
	WRITE	0		0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x0000A000		0x0000A000		0x0000A00D		0x0000A00B	
	STAGE	IF		ID/RF		EXE		MEM		WB		WB	

Image 6.28 Control derivation for beq and bne

INSTRUCTION: lw rt, rs, imm | R[rt] = M[R[rs] + SignExtImm]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	8
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**1**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	8	**0**	8	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**1**		**1**		0	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		0	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	4	**0**	4	0	0
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		0	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		0	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	0	**0**	0	0	0
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		0	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		**1**	
CTRL[24]	wd_sel_2	0		0		0		0		**0**	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	**1**	6
CTRL[26]	wa_sel_1	0		0		0		0		**1**	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	1	0	**1**	1
	READ	1		0		0		1		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000050		0x00004800		0x00004800		0x1680008B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.29 Control derivation for lw

INSTRUCTION: sw rt, rs, imm | M[R[rs] + SignExtImm] = R[rt]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	**1**	B
CTRL[2]	pc_sel_2	0		0		0		0		**0**	
CTRL[3]	pc_sel_3	0		0		0		0		**1**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	**0**	5	0	0	0	0	0	0
CTRL[6]	reg_r	**0**		**1**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**0**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	**0**	8	**0**	8	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		**1**		**1**		0	
CTRL[12]	op2_sel_3	0		0		**0**		**0**		0	
CTRL[13]	op2_sel_4	0	0	0	0	**0**	4	**0**	4	0	0
CTRL[14]	alu_oprn[0]	0		0		**1**		**1**		0	
CTRL[15]	alu_oprn[1]	0		0		**0**		**0**		0	
CTRL[16]	alu_oprn[2]	0		0		**0**		**0**		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	**0**	0	**0**	0	0	0
CTRL[18]	alu_oprn[4]	0		0		**0**		**0**		0	
CTRL[19]	alu_oprn[5]	0		0		**0**		**0**		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	0	0
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		0		0	
	WRITE	0		0		0		1		0	
	CODE	0x00200000		0x00000050		0x00004800		0x00004800		0x0000000B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.30 Control derivation for sw

INSTRUCTION: jmp address | PC = JumpAddress(6'b0, address)

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	**0**		**0**		**0**		**0**		**1**	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	0	1
CTRL[2]	pc_sel_2	0		0		0		0		0	
CTRL[3]	pc_sel_3	0		0		0		0		**0**	
CTRL[4]	ir_load	**0**		**1**		**0**		**0**		**0**	
CTRL[5]	r1_sel_1	0	0	0	1	0	0	0	0	0	0
CTRL[6]	reg_r	**0**		**0**		**0**		**0**		**0**	
CTRL[7]	reg_w	**0**		**0**		**0**		**0**		**0**	
CTRL[8]	sp_load	**0**		**0**		**0**		**0**		**0**	
CTRL[9]	op1_sel_1	0	0	0	0	0	0	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	0	0	0	0	0	0
CTRL[14]	alu_oprn[0]	0		0		0		0		0	
CTRL[15]	alu_oprn[1]	0		0		0		0		0	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	**1**	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	0	0
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000010		0x00000000		0x00000000		0x00000001	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.31 Control derivation for jmp

INSTRUCTION: jal address | R[31] = PC + 1 and PC = JumpAddress(6'b0, address)

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	0		0		0		0		1	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	0	1
CTRL[2]	pc_sel_2	0		0		0		0		0	
CTRL[3]	pc_sel_3	0		0		0		0		0	
CTRL[4]	ir_load	0		1		0		0		0	
CTRL[5]	r1_sel_1	0	0	0	1	0	0	0	0	0	8
CTRL[6]	reg_r	0		0		0		0		0	
CTRL[7]	reg_w	0		0		0		0		1	
CTRL[8]	sp_load	0		0		0		0		0	
CTRL[9]	op1_sel_1	0	0	0	0	0	0	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		0		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	0	0	0	0	0	0
CTRL[14]	alu_oprn[0]	0		0		0		0		0	
CTRL[15]	alu_oprn[1]	0		0		0		0		0	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		0		0	
CTRL[21]	ma_sel_2	1	2	0	0	0	0	0	0	0	0
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	0	8
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		1	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		0		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000010		0x00000000		0x00000000		0x08000081	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.32 Control derivation for jal

INSTRUCTION: push address | M[sp] = R[0] and sp = sp - 1

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	0		0		0		0		1	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	1	B
CTRL[2]	pc_sel_2	0		0		0		0		0	
CTRL[3]	pc_sel_3	0		0		0		0		1	
CTRL[4]	ir_load	0		1		0		0		0	
CTRL[5]	r1_sel_1	0	0	1	7	0	0	0	0	0	0
CTRL[6]	reg_r	0		1		0		0		0	
CTRL[7]	reg_w	0		0		0		0		0	
CTRL[8]	sp_load	0		0		0		0		1	
CTRL[9]	op1_sel_1	0	0	0	0	1	2	1	2	1	3
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		1		0		1	
CTRL[13]	op2_sel_4	0	0	0	0	0	9	0	9	0	9
CTRL[14]	alu_oprn[0]	0		0		0		0		0	
CTRL[15]	alu_oprn[1]	0		0		1		1		1	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		1		0	
CTRL[21]	ma_sel_2	1	2	0	0	0	0	0	5	0	0
CTRL[22]	md_sel_1	0		0		0		1		0	
CTRL[23]	wd_sel_1	0		0		0		0		0	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	0	0
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		0		0	
	WRITE	0		0		0		1		0	
	CODE	0x00200000		0x00000070		0x00009200		0x00509200		0x0000930B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.33 Control derivation for push

INSTRUCTION: pop address | sp = sp + 1 and R[0] = M[sp]

CONTROL	SIGNAL	IF	HEX	ID/RF	HEX	EXE	HEX	MEM	HEX	WB	HEX
CTRL[0]	pc_load	0		0		0		0		1	
CTRL[1]	pc_sel_1	0	0	0	0	0	0	0	0	1	B
CTRL[2]	pc_sel_2	0		0		0		0		0	
CTRL[3]	pc_sel_3	0		0		0		0		1	
CTRL[4]	ir_load	0		1		0		0		0	
CTRL[5]	r1_sel_1	0	0	0	1	0	0	0	0	0	8
CTRL[6]	reg_r	0		0		0		0		0	
CTRL[7]	reg_w	0		0		0		0		1	
CTRL[8]	sp_load	0		0		1		0		0	
CTRL[9]	op1_sel_1	0	0	0	0	1	3	0	0	0	0
CTRL[10]	op2_sel_1	0		0		0		0		0	
CTRL[11]	op2_sel_2	0		0		0		0		0	
CTRL[12]	op2_sel_3	0		0		1		0		0	
CTRL[13]	op2_sel_4	0	0	0	0	0	5	0	0	0	0
CTRL[14]	alu_oprn[0]	0		0		1		0		0	
CTRL[15]	alu_oprn[1]	0		0		0		0		0	
CTRL[16]	alu_oprn[2]	0		0		0		0		0	
CTRL[17]	alu_oprn[3]	0	0	0	0	0	0	0	0	0	0
CTRL[18]	alu_oprn[4]	0		0		0		0		0	
CTRL[19]	alu_oprn[5]	0		0		0		0		0	
CTRL[20]	ma_sel_1	0		0		0		1		0	
CTRL[21]	ma_sel_2	1	2	0	0	0	0	0	1	0	8
CTRL[22]	md_sel_1	0		0		0		0		0	
CTRL[23]	wd_sel_1	0		0		0		0		1	
CTRL[24]	wd_sel_2	0		0		0		0		0	
CTRL[25]	wd_sel_3	0	0	0	0	0	0	0	0	1	2
CTRL[26]	wa_sel_1	0		0		0		0		0	
CTRL[27]	wa_sel_2	0		0		0		0		0	
CTRL[28]	wa_sel_3	0	0	0	0	0	0	0	0	0	0
	READ	1		0		0		1		0	
	WRITE	0		0		0		0		0	
	CODE	0x00200000		0x00000010		0x00005300		0x00100000		0x0280008B	
	STAGE	IF		ID/RF		EXE		MEM		WB	

Image 6.34 Control derivation for pop

VII. Data Path Design, Implementation, and Testing

The data path is responsible for the flow of data in the system. It works in unison with the control unit to form the processor. It receives a 32-bit control word, 32-bit input data, 1-bit clock signal, and a 1-bit reset signal. It outputs a 26-bit address, 32-bit instruction, 32-bit output data, and a 1-bit zero signal. It connects the ALU, RF, instruction register, program counter, and stack pointer through a system of many multiplexers. The data path does not have an independent test bench because it requires the control unit to function properly. It will be tested as part of the complete system test. The following is the digital circuit diagram and implementation for the data path:

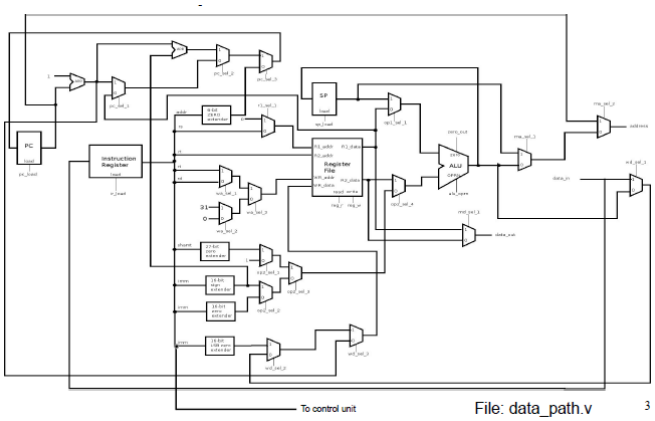

Image 7.1 Data path digital circuit diagram

```
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 [`DATA_INDEX_LIMIT:0] R1_data, R2_data, alu_out, add_res_1, add_res_2,
                           op1_sel_1, op2_sel_1, op2_sel_2, op2_sel_3, op2_sel_4,
                           wd_sel_1, wd_sel_2, wd_sel_3, ma_sel_2, ma_sel_1,
                           pc_out, ir_res, sp_res, pc_sel_1, pc_sel_2, pc_sel_3;
wire [4:0] r1_sel_1, wa_sel_1, wa_sel_2, wa_sel_3;
wire unused;

BUF32 buf_inst(.Y(INSTRUCTION), .A(DATA_IN));

RC_ADD_SUB_32 add_1(.Y(add_res_1), .CO(unused), .A(32'b1), .B(pc_out), .SnA(1'b0));
RC_ADD_SUB_32 add_2(.Y(add_res_2), .CO(unused), .A(add_res_1), .B({{16{ir_res[15]}}, ir_res[15:0]}), .SnA(1'b0));

defparam pc_inst.PATTERN = `INST_START_ADDR;
REG32_PP pc_inst(.Q(pc_out), .D(pc_sel_3), .LOAD(CTRL[0]), .CLK(CLK), .RESET(RST));

MUX32_2x1 mux_pc_sel_1(.Y(pc_sel_1), .I0(R1_data), .I1(add_res_1), .S(CTRL[1]));
MUX32_2x1 mux_pc_sel_2(.Y(pc_sel_2), .I0(pc_sel_1), .I1(add_res_2), .S(CTRL[2]));
MUX32_2x1 mux_pc_sel_3(.Y(pc_sel_3), .I0({6'b0, ir_res[25:0]}), .I1(pc_sel_2), .S(CTRL[3]));

REG32 ir_inst(.Q(ir_res), .D(DATA_IN), .LOAD(CTRL[4]), .CLK(CLK), .RESET(RST));

MUX5_2x1 mux_r1_sel_1(.Y(r1_sel_1), .I0(ir_res[25:21]), .I1(5'b00000), .S(CTRL[5]));

REGISTER_FILE_32x32_BEHAVIORAL rf_32x32_inst(.DATA_R1(R1_data), .DATA_R2(R2_data), .ADDR_R1(r1_sel_1), .ADDR_R2(ir_res[20:16]),
                                             .DATA_W(wd_sel_3), .ADDR_W(wa_sel_3), .READ(CTRL[6]), .WRITE(CTRL[7]), .CLK(CLK), .RST(RST));

defparam sp_inst.PATTERN = `INIT_STACK_POINTER;
REG32_PP sp_inst(.Q(sp_res), .D(alu_out), .LOAD(CTRL[8]), .CLK(CLK), .RESET(RST));

MUX32_2x1 mux_op1_sel_1(.Y(op1_sel_1), .I0(R1_data), .I1(sp_res), .S(CTRL[9]));

MUX32_2x1 mux_op2_sel_1(.Y(op2_sel_1), .I0(32'b1), .I1({{27'b0, ir_res[10:6]}}), .S(CTRL[10]));
MUX32_2x1 mux_op2_sel_2(.Y(op2_sel_2), .I0({16'b0, ir_res[15:0]}), .I1({{16{ir_res[15]}}, ir_res[15:0]}), .S(CTRL[11]));
MUX32_2x1 mux_op2_sel_3(.Y(op2_sel_3), .I0(op2_sel_2), .I1(op2_sel_1), .S(CTRL[12]));
MUX32_2x1 mux_op2_sel_4(.Y(op2_sel_4), .I0(op2_sel_2), .I1(R2_data), .S(CTRL[13]));

ALU alu_inst(.OUT(alu_out), .ZERO(ZERO), .OP1(op1_sel_1), .OP2(op2_sel_4), .OPRN(CTRL[19:14]));

MUX32_2x1 mux_ma_sel_1(.Y(ma_sel_1), .I0(alu_out), .I1(sp_res), .S(CTRL[20]));
MUX32_2x1 mux_ma_sel_2(.Y(ADDR), .I0(ma_sel_1), .I1(pc_out), .S(CTRL[21]));

MUX32_2x1 mux_md_sel_1(.Y(DATA_OUT), .I0(R2_data), .I1(R1_data), .S(CTRL[22]));

MUX32_2x1 mux_wd_sel_1(.Y(wd_sel_1), .I0(alu_out), .I1(DATA_IN), .S(CTRL[23]));
MUX32_2x1 mux_wd_sel_2(.Y(wd_sel_2), .I0(wd_sel_1), .I1({{ir_res[15:0]}, 16'b0}), .S(CTRL[24]));
MUX32_2x1 mux_wd_sel_3(.Y(wd_sel_3), .I0(add_res_1), .I1(wd_sel_2), .S(CTRL[25]));

MUX5_2x1 mux_wa_sel_1(.Y(wa_sel_1), .I0(ir_res[15:11]), .I1(ir_res[20:16]), .S(CTRL[26]));
MUX5_2x1 mux_wa_sel_2(.Y(wa_sel_2), .I0(5'b00000), .I1(5'b11111), .S(CTRL[27]));
MUX5_2x1 mux_wa_sel_3(.Y(wa_sel_3), .I0(wa_sel_2), .I1(wa_sel_1), .S(CTRL[28]));

endmodule
```

Image 7.2 Implementation of data path

VIII. Processor Design, Implementation, and Testing

The processor is comprised of the control unit and the data path. The processor receives a 32-bit input data from the memory, 1-bit clock signal, and a 1-bit reset signal. It outputs a 26-bit address, 32-bit output data, 1-bit read signal, and a 1-bit write signal. Again, the processor does not have its own test bench and will be tested as part of the entire system test

bench. The following is the digital circuit diagram and implementation for the processor:

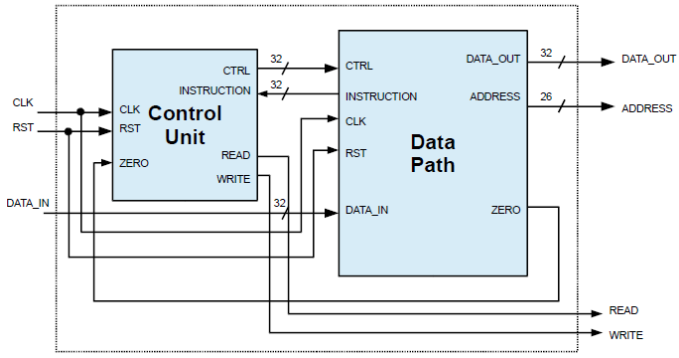

Image 8.1 Processor digital circuit diagram

```
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;
```

Image 8.2 Implementation of processor

IX. System Implementation Design, Implementation, and Testing

The DaVinci v1.0m system implementation is comprised of the processor and the memory wrapper. The testing is done using the provided Fibonacci and RevFib tests. The following is the digital circuit diagram, implementation, and test bench results for the DaVinci v1.0m system:

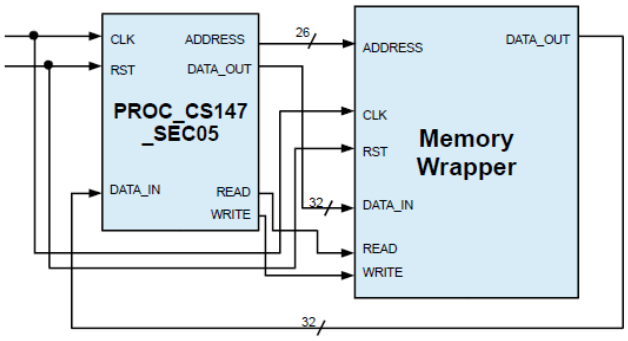

Image 9.1 DaVinci v1.0m digital circuit diagram

```
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));*/
MEMORY_64MB memory_inst(.DATA_OUT(MEM_DATA_OUT), .DATA_IN(MEM_DATA_IN),
                        .READ(READ), .WRITE(WRITE),
                        .ADDR(ADDR), .CLK(CLK),   .RST(RST));
endmodule;
```

Image 9.2 Implementation of DaVinci v1.0m

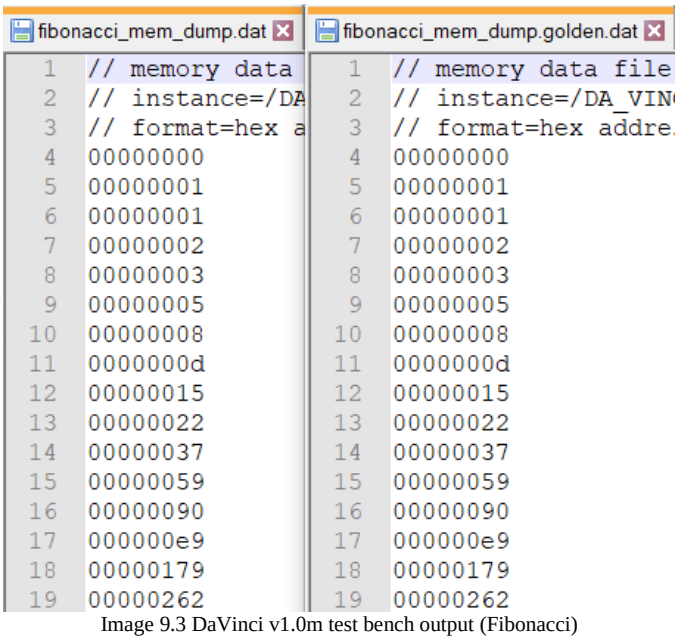

Image 9.3 DaVinci v1.0m test bench output (Fibonacci)

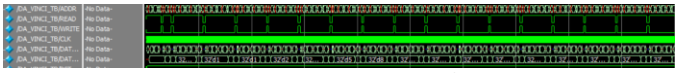

Image 9.4 DaVinci v1.0m test bench waveform (Fibonacci)

RevFib_mem_dump.dat

```
1  // memory data
2  // instance=/DA
3  // format=hex a
4  fffffffc9
5  00000022
6  ffffffeb
7  0000000d
8  fffffff8
9  00000005
10 fffffffd
11 00000002
12 ffffffff
13 00000001
14 00000000
15 00000001
16 00000001
17 00000002
18 00000003
19 00000005
```

RevFib_mem_dump.golden.dat

```
1  // memory data fil
2  // instance=/DA_VI
3  // format=hex addr
4  fffffffc9
5  00000022
6  ffffffeb
7  0000000d
8  fffffff8
9  00000005
10 fffffffd
11 00000002
12 ffffffff
13 00000001
14 00000000
15 00000001
16 00000001
17 00000002
18 00000003
19 00000005
```

Image 9.5 DaVinci v1.0m test bench output (RevFib)

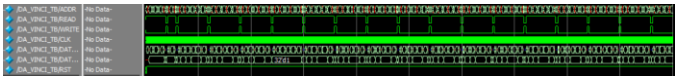

Image 9.6 DaVinci v1.0m test bench waveform (RevFib)

X. Conclusion

This project detailed how to build a simple computer system at the gate level. It included digital circuit diagrams, implementation, and test benches for all the various components of the system. Even though all the individual test benches yielded successful results, the full system integration test with the gate level register file and memory wrapper yielded unsuccessful results. The system passes the full system test when used alongside the behavioral model of the register file and with no memory wrapper and a separate input and output port for the 64MB memory. Due to this observation, it seems every other part of the gate level modeling, including the data path and control unit work as expected, but for an unknown reason, its integration with the gate level register file and memory wrapper fails.