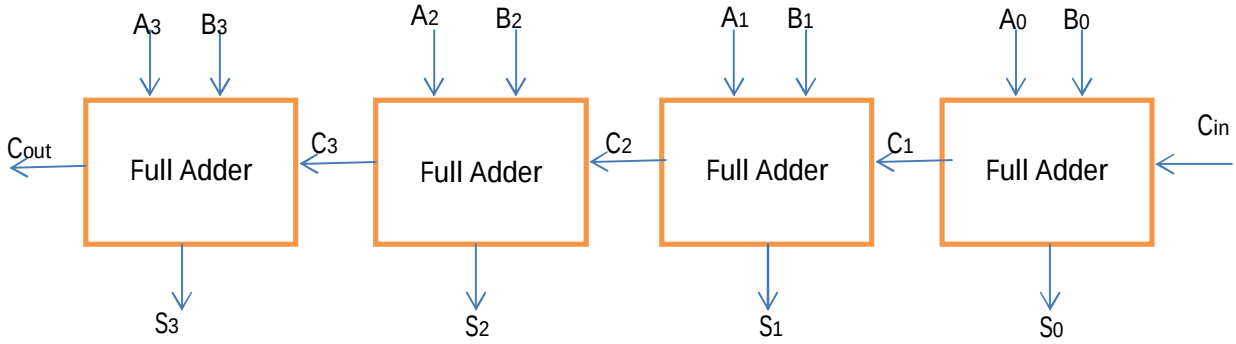
PROBLEM 1

- **Split Register**
  This instruction requires changes to the datapath and control path. The biggest change would be a register file with two write ports (this instruction writes to both $rd and $rt). Additionally, there would have to be a change in the datapath to accommodate writing the bottom half of a register.

- **Bit Equal**
  Presented in discussion session slides
  http://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/discuss5.pdf

- **Replace Under Mask**
  The implementation of this instruction would require a specialized version of barrel shifter and additional comparator logic in the datapath to ensure that it can be accommodated in a MIPS-like pipeline. If the result was written to $rd instead of $rt, then a complete redesign of the ISA would have been required.

PROBLEM 2



➔ Delay for C1 = 2 input AND (8 + 2^2)τ = 12τ + 3 input OR gate (8 + 3^2)τ = 17τ = **29τ**

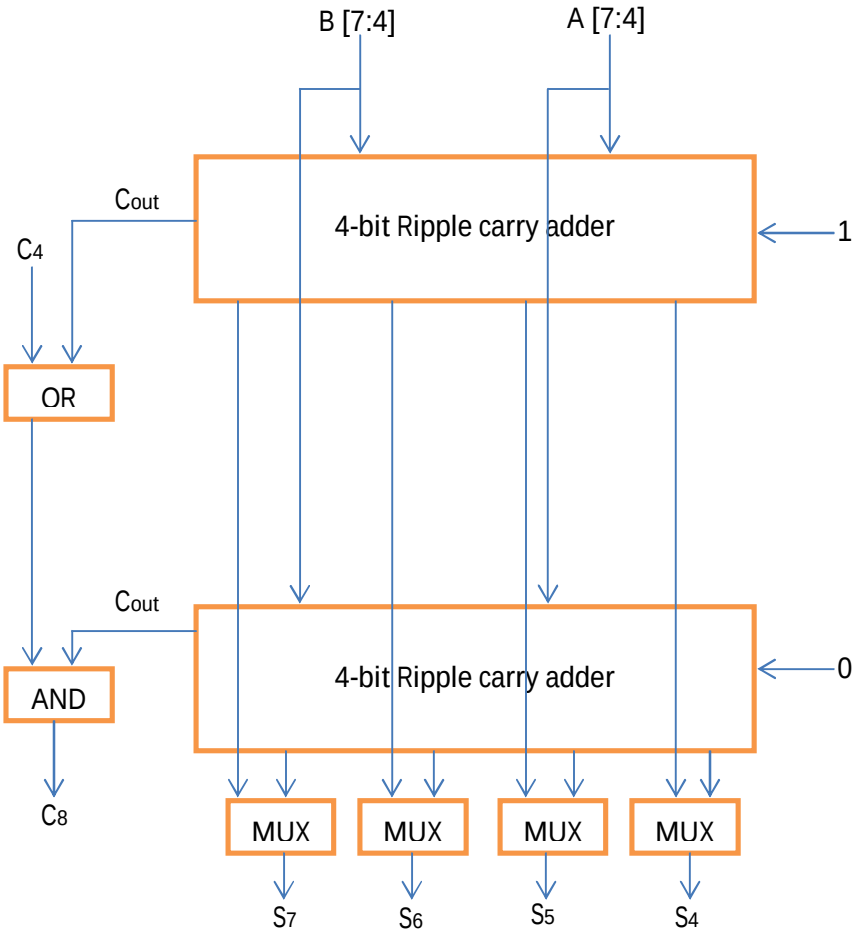
➔ Delay for C2 = delay for C1 + functional delay = 29τ + 29τ = **58τ**

➔ Delay for C3 = delay for C2 + functional delay = 58τ + 29τ = **87τ**

➔ Delay for C4 = delay for C3 + functional delay = 87τ + 29τ = **116τ**

Delay for S5:
C4 arrives at **116τ**. We have to find the critical path for MUX which is the **sel** signal
Functional delay of MUX = (8 + 1^2)τ + (8 + 2^2)τ + (8 + 2^2)τ = **33τ**

Total delay = 116τ + 33τ = 149τ


B [7:4]
A [7:4]
Cout
4-bit Ripple carry adder
1
C4
OR
Cout
4-bit Ripple carry adder
0
AND
C8
MUX
MUX
MUX
MUX
S7
S6
S5
S4

```
PROBLEM 3 - Register File
// Top level rf_hier module - provided to class
/* $Author: karu $ */
/* $LastChangedDate: 2009-03-04 23:09:45 -0600 (Wed, 04 Mar 2009) $ */
/* $Rev: 45 $ */
// YOU SHALL NOT EDIT THIS FILE. ANY CHANGES TO THIS FILE WILL
// RESULT IN ZERO FOR THIS PROBLEM.

module rf_hier (
                // Outputs
                read1data, read2data,
                // Inputs
                read1regsel, read2regsel, writeregsel, writedata, write
                );
   input [2:0] read1regsel;
   input [2:0] read2regsel;
   input [2:0] writeregsel;
   input [15:0] writedata;
   input        write;

   output [15:0] read1data;
   output [15:0] read2data;

   wire          clk, rst;
   wire          err;


   // Ignore err for now
   clkrst clk_generator(.clk(clk), .rst(rst), .err(err) );
   rf rf0(
          // Outputs
          .read1data                    (read1data[15:0]),
          .read2data                    (read2data[15:0]),
          .err                          (err),
          // Inputs
          .clk                          (clk),
          .rst                          (rst),
          .read1regsel                  (read1regsel[2:0]),
          .read2regsel                  (read2regsel[2:0]),
          .writeregsel                  (writeregsel[2:0]),
          .writedata                    (writedata[15:0]),
          .write                        (write));

endmodule
// DUMMY LINE FOR REV CONTROL :4:

// Your code for register file goes in here
/* $Author: karu $ */
/* $LastChangedDate: 2009-03-04 23:09:45 -0600 (Wed, 04 Mar 2009) $ */
/* $Rev: 45 $ */
module rf (
           // Outputs
           read1data, read2data, err,
           // Inputs
           clk, rst, read1regsel, read2regsel, writeregsel, writedata, write
           );
```

```
    // Use of parameter to make RF modifiable later
    parameter WIDTH = 16;

    // define module injputs and outputs
    input clk, rst;
    input [2:0] read1regsel;
    input [2:0] read2regsel;
    input [2:0] writeregsel;
    input [WIDTH-1:0] writedata;
    input        write;

    output [WIDTH-1:0] read1data;
    output [WIDTH-1:0] read2data;
    output        err;

    // Assigning err to 0; can be used for debugging purposes
    assign err = 0; // Should it be there?

    // wires that are local to the module
    wire [WIDTH-1:0] q7,q6,q5,q4,q3,q2,q1,q0;
    wire [7:0] we, awe;

    // instantiating a 3-8 decoder
    decode3_8 deocder (.sel(writeregsel), .Out(we));

    // generating write enable signals
    and2 andgates[7:0] (.in1(we), .in2({8{write}}), .out(awe));

    // individual registers - note that there are 8 such copies
    register #(WIDTH) my_regs7 (.q(q7), .d(writedata), .clk(clk), .rst(rst), .we(awe[7]));
    register #(WIDTH) my_regs6 (.q(q6), .d(writedata), .clk(clk), .rst(rst), .we(awe[6]));
    register #(WIDTH) my_regs5 (.q(q5), .d(writedata), .clk(clk), .rst(rst), .we(awe[5]));
    register #(WIDTH) my_regs4 (.q(q4), .d(writedata), .clk(clk), .rst(rst), .we(awe[4]));
    register #(WIDTH) my_regs3 (.q(q3), .d(writedata), .clk(clk), .rst(rst), .we(awe[3]));
    register #(WIDTH) my_regs2 (.q(q2), .d(writedata), .clk(clk), .rst(rst), .we(awe[2]));
    register #(WIDTH) my_regs1 (.q(q1), .d(writedata), .clk(clk), .rst(rst), .we(awe[1]));
    register #(WIDTH) my_regs0 (.q(q0), .d(writedata), .clk(clk), .rst(rst), .we(awe[0]));

    // instantiate 8:1 MUX for choosing what needs to come at the output read port
    mux8_1 choosefrom8[WIDTH-1:0] (.InA(q0), .InB(q1), .InC(q2), .InD(q3), .InE(q4), .InF(q5), .
    InG(q6), .InH(q7), .S({WIDTH{read1regsel}}), .Out(read1data));
    mux8_1 choosefrom8again[WIDTH-1:0] (.InA(q0), .InB(q1), .InC(q2), .InD(q3), .InE(q4), .InF(
    q5), .InG(q6), .InH(q7), .S({WIDTH{read2regsel}}), .Out(read2data));

endmodule
// DUMMY LINE FOR REV CONTROL :1:

// Code for 3 to 8 decoder
module decode3_8 (sel, Out);

    // define module inputs and outputs
    input [2:0] sel;
    output [7:0] Out;

    // wires that are local to the module
```

```
    wire [2:0] nsel;

    // structural design of a 3 to 8 decoder
    not1 nottedsel[2:0] (.in1(sel), .out(nsel[2:0]));
    nor3 ng0 (.in1(sel[0]), .in2(sel[1]), .in3(sel[2]), .out(Out[0]));
    nor3 ng1 (.in1(nsel[0]), .in2(sel[1]), .in3(sel[2]), .out(Out[1]));
    nor3 ng2 (.in1(sel[0]), .in2(nsel[1]), .in3(sel[2]), .out(Out[2]));
    nor3 ng3 (.in1(nsel[0]), .in2(nsel[1]), .in3(sel[2]), .out(Out[3]));
    nor3 ng4 (.in1(sel[0]), .in2(sel[1]), .in3(nsel[2]), .out(Out[4]));
    nor3 ng5 (.in1(nsel[0]), .in2(sel[1]), .in3(nsel[2]), .out(Out[5]));
    nor3 ng6 (.in1(sel[0]), .in2(nsel[1]), .in3(nsel[2]), .out(Out[6]));
    nor3 ng7 (.in1(nsel[0]), .in2(nsel[1]), .in3(nsel[2]), .out(Out[7]));

endmodule
// end of module

// Code for register
module register (q, d, clk, rst, we);

    // This value can be changed later
    parameter WIDTHreg = 16;

    // define inputs and outputs of the module
    input[WIDTHreg-1:0] q;
    input clk, rst, we;
    output [WIDTHreg-1:0] d;

    // wires that are local to the module
    wire [WIDTHreg-1:0] enabledIn;

    // A combination of a 2:1 MUX and a DFF to hold and update values; for more information
    look into discussion session slides
    mux2_1 mux[WIDTHreg-1:0] (.InA (q), .InB(d), .S({WIDTHreg{we}}), .Out(enabledIn));
    dff file[WIDTHreg-1:0] (.q(q), .d(enabledIn), .clk({WIDTHreg{clk}}), .rst({WIDTHreg{rst}}));

endmodule
// end of module

// Code for 8:1 MUX
module mux8_1(InA, InB, InC, InD, InE, InF, InG, InH, S, Out);

    // define module inputs and outputs
    input  InA;
    input  InB;
    input  InC;
    input  InD;
    input  InE;
    input  InF;
    input  InG;
    input  InH;
    input  [2:0] S;
    output         Out;

    // wires that are local to the module
    wire        result1;
    wire        result2;
```

```
    // Build a 8:1 MUX using two 4:1 MUXes and one 2:1 MUX designed in HW1
    mux4_1      mux1(.InA(InA),     .InB(InB), .InC(InC), .InD(InD),    .S(S[1:0]), .Out(result1
    ));
    mux4_1      mux2(.InA(InE),     .InB(InF), .InC(InG), .InD(InH),     .S(S[1:0]),    .Out(
    result2));
    mux2_1      mux3(.InA(result1), .InB(result2),  .S(S[2]),   .Out(Out));

endmodule
// end of module
```

```
PROBLEM 4 - Saturating Counter
// Top level sc_hier module - provided to class
/* $Author: karu $ */
/* $LastChangedDate: 2009-03-04 23:09:45 -0600 (Wed, 04 Mar 2009) $ */
/* $Rev: 45 $ */
// YOU SHALL NOT EDIT THIS FILE. ANY CHANGES TO THIS FILE WILL
// RESULT IN ZERO FOR THIS PROBLEM.

module sc_hier (/*AUTOARG*/
   // Outputs
   out,
   // Inputs
   ctr_rst
   );


   input ctr_rst;
   output [2:0] out;

   wire         err;
   wire         clk;
   wire         rst;

   clkrst clk_generator(.clk(clk), .rst(rst), .err(err) );
   sc sc0(/*AUTOINST*/
          // Outputs
          .out                          (out[2:0]),
          .err                          (err),
          // Inputs
          .clk                          (clk),
          .rst                          (rst),
          .ctr_rst                      (ctr_rst));

endmodule
// DUMMY LINE FOR REV CONTROL :1:

// Code for saturating counter
module sc( clk, rst, ctr_rst, out, err);

   // define module inputs and outputs
   input clk;
   input rst;
   input ctr_rst;
   output [2:0] out;
   output reg err;

   reg [2:0] nextState;

   // Instantiate DFF
   dff test [2:0](out, nextState, clk, rst);

   // Begin logic for saturating counter
   always@(out, ctr_rst)begin
     case(out)
       3'd0:begin
         nextState=ctr_rst?3'd0:3'd1;
```

```
        err=1'd0;
      end
      3'd1:begin
        nextState=ctr_rst?3'd0:3'd2;
        err=1'd0;
      end
      3'd2:begin
        nextState=ctr_rst?3'd0:3'd3;
        err=1'd0;
      end
      3'd3:begin
        nextState=ctr_rst?3'd0:3'd4;
        err=1'd0;
      end
      3'd4:begin
        nextState=ctr_rst?3'd0:3'd5;
        err=1'd0;
      end
      3'd5:begin
        nextState=ctr_rst?3'd0:3'd5;
        err=1'd0;
      end
      default:begin
        nextState=3'd1;
        err=1'd1;
      end
    endcase
  end
  // end counter logic

endmodule
// end of module
```

```
PROBLEM 5 - FIFO
// Top level fifo_hier module - provided to class
/* $Author: karu $ */
/* $LastChangedDate: 2009-03-04 23:09:45 -0600 (Wed, 04 Mar 2009) $ */
/* $Rev: 45 $ */
// YOU SHALL NOT EDIT THIS FILE. ANY CHANGES TO THIS FILE WILL
// RESULT IN ZERO FOR THIS PROBLEM.

module fifo_hier(/*AUTOARG*/
   // Outputs
   data_out, fifo_empty, fifo_full, data_out_valid,
   // Inputs
   data_in, data_in_valid, pop_fifo
   );

   input [63:0] data_in;
   input        data_in_valid;
   input        pop_fifo;

   output [63:0] data_out;
   output        fifo_empty;
   output        fifo_full;
   output        data_out_valid;

   clkrst clk_generator(.clk(clk),
                        .rst(rst),
                        .err(err) );

   fifo fifo0(/*AUTOINST*/
              // Outputs
              .data_out                 (data_out[63:0]),
              .fifo_empty               (fifo_empty),
              .fifo_full                (fifo_full),
              .data_out_valid           (data_out_valid),
              .err                      (err),
              // Inputs
              .data_in                  (data_in[63:0]),
              .data_in_valid            (data_in_valid),
              .pop_fifo                 (pop_fifo),
              .clk                      (clk),
              .rst                      (rst));

endmodule
// DUMMY LINE FOR REV CONTROL :4:

// Code for FIFO
module fifo(/*AUTOARG*/
   // Outputs
   data_out, fifo_empty, fifo_full, data_out_valid, err,
   // Inputs
   data_in, data_in_valid, pop_fifo, clk, rst
   );

    // define module inputs and outputs
    input [63:0] data_in;
    input        data_in_valid;
```

```
input        pop_fifo;
input        clk;
input        rst;

output [63:0] data_out;
output        fifo_empty;
output        fifo_full;
output        data_out_valid;
output        err;

// wires local to module
wire [63:0] data[3:0],data_flop[3:0];   // use of 2-dimensional array for holding data value
wire [1:0] rd_ptr,wr_ptr;
reg [1:0] rd_nxt, wr_nxt;
wire [2:0] count;
reg [2:0] nxt_count;
wire [63:0] d_out;

// Instantiate DFF for maintaining read and write pointer logic
dff temp0 [1:0](rd_ptr, rd_nxt, clk, rst);
dff temp1 [1:0](wr_ptr, wr_nxt, clk, rst);
dff temp2 [2:0](count, nxt_count, clk, rst);

// Instantiate DFF for maintaining data update logic
dff temp3 [63:0](data_flop[0], data[0],clk,rst);
dff temp4 [63:0](data_flop[1], data[1],clk,rst);
dff temp5 [63:0](data_flop[2], data[2],clk,rst);
dff temp6 [63:0](data_flop[3], data[3],clk,rst);

// Logic for data to be pushed into FIFO
assign data[0] =(data_in_valid & (wr_ptr==2'd0) & (~fifo_full))? data_in:data_flop[0];
assign data[1] =(data_in_valid & (wr_ptr==2'd1) & (~fifo_full))? data_in:data_flop[1];
assign data[2] =(data_in_valid & (wr_ptr==2'd2) & (~fifo_full))? data_in:data_flop[2];
assign data[3] =(data_in_valid & (wr_ptr==2'd3) & (~fifo_full))? data_in:data_flop[3];

// Updating read pointer
always@(*)begin
    case(rd_ptr)
        2'd0:rd_nxt=(pop_fifo & (~fifo_empty))?2'd1:2'd0;
        2'd1:rd_nxt=(pop_fifo & (~fifo_empty))?2'd2:2'd1;
        2'd2:rd_nxt=(pop_fifo & (~fifo_empty))?2'd3:2'd2;
        2'd3:rd_nxt=(pop_fifo & (~fifo_empty))?2'd0:2'd3;
    endcase
end

// Updating WRITE pointer
always@(*)begin
    case(wr_ptr)
        2'd0: wr_nxt=(data_in_valid & (~fifo_full))? 2'd1:2'd0;
        2'd1: wr_nxt=(data_in_valid & (~fifo_full))? 2'd2:2'd1;
        2'd2: wr_nxt=(data_in_valid & (~fifo_full))? 2'd3:2'd2;
        2'd3: wr_nxt=(data_in_valid & (~fifo_full))? 2'd0:2'd3;
    endcase
end

// Maintaining count to see if FIFO is full or empty
```

```
    always @(*)begin
        case(count)
            3'd0: nxt_count=~(data_in_valid^pop_fifo)?3'd0:data_in_valid?3'd1:3'd0;
            3'd1: nxt_count=~(data_in_valid^pop_fifo)?3'd1:data_in_valid?3'd2:3'd0;
            3'd2: nxt_count=~(data_in_valid^pop_fifo)?3'd2:data_in_valid?3'd3:3'd1;
            3'd3: nxt_count=~(data_in_valid^pop_fifo)?3'd3:data_in_valid?3'd4:3'd2;
            3'd4: nxt_count=~(data_in_valid^pop_fifo)?3'd4:data_in_valid?3'd4:3'd3;
            default nxt_count=3'bz;
        endcase
    end

    // deciding logic for FIFO full and empty
    assign data_out_valid=~fifo_empty;
    assign data_out= data_flop[rd_ptr];
    assign fifo_full=(count==3'd4);
    assign fifo_empty=(count==3'd0);

endmodule
// end of module
```