```
module top;
   wire         clk, Reset;

   // Added by Smart for memory subsystem
   wire [0:127] I_Cache_Data_To_Mem, D_Cache_Data_To_Mem;
   wire [0:31]  D_Addr, I_Ext_Addr, D_Ext_Addr;
   wire [0:31]  D_Cache_Data_In_New;
   wire [0:31]  D_Cache_Data_Out;
   wire [0:1]   Ext_Interrupt;
   wire [0:31]  IO_Data_In, IO_Data_Out;
   wire         IAlignFaultTrue, DAlignFaultTrue;
   wire         Halt_Final;

   // IF
   wire [0:31]  PCin, PC, BranchPCout, BranchPC, LROut,
                IFIDIRinput, NPC, Instruction, Cache_Data_In_New;
   wire [0:63]  IFIDCTRL, IFIDCTRLtemp;
   wire [0:127] Cache_Data_In;
   wire [0:2]   pcchoice;
      // HOUSER 4/14/97
   wire [0:31]  PredTarget, UA, UTarget,  b_haz_PC;


   // ID
   wire [0:31]  IFIDIR, IFIDPC, IFIDNPC, RA_OUT, RB_OUT, RT_OUT;
   wire [0:4]   RA, RB, RT;
   wire [0:63]  rom1_out, rom2_out, temp3;
   wire [0:4]   EXTADDR;
   wire [0:63]  IDEXCTRL, IDEXCTRLin;
   wire [0:31]  IFIDPredTarget;


   // EX
   wire [0:31]  IDEXIR, IDEXNPC, IDEXPC, IDEXRA, IDEXRB, IDEXRT,
                IDEXRTtemp, IDEXD, SignExt, IDEXSPR;
   wire [0:31]  ALU_RA, ALU_RB, ALUOut, PCOp, NewPC, Out;
   wire [0:31]  EXMEMIR, EXMEMPC, EXMEMOUT, EXMEMNPC,
                MemDataOut, EXMEMRT, EXMEMRTtemp, CTROUT, EXMEMSPR;
   wire [0:63]  EXMEMCTRL, EXMEMCTRLin;
   wire [0:3]   ALUOp;
   wire [0:1]   temp_SelA, SelA;
   wire         SelB, LoadHaz, trap_EX_Inval;
   wire [0:4]   EXWR1, EXWR2;
   wire [0:2]   ASelect, BSelect;
   wire [0:7]   IDEXHAZ, IDEXMEMHAZ;
   wire [0:31]  IDEXPredTarget, SRR0, SRR1, MSR, NewPCtemp;
   wire [0:3]   IDEXSTOHAZ;


   // MEM
   wire [0:31]  MEMWBIR, MEMWBOUT, WBData, CR,
                MEMWBSPR, XEROUT, MEM_WRDATA1, MEM_WRDATA2;
   wire [0:63]  MEMWBCTRL, MEMWBCTRLin;
   wire [0:21]  unused;
   wire [0:1]   SelSPR;
   wire [0:4]   MEMWR1, MEMWR2;
   wire [0:4]   IF_Exc, EX_Exc, M_Exc;
   wire         ID_Exc;
   wire [0:1]   IO_Int;
   wire [0:31]  trap_Handler_Addr, trap_PC_Val;
   wire [0:3]   EXMEMSTOHAZ;


   // WB
   wire [0:4]   WR1, WR2;
   wire [0:31]  WRDATA1, WRDATA2;


   // Control signals
   wire         WB_Load, Logical, Load, Link, WRE1, temp_WRE2, cin, MemToReg,
                byte, MemWr, MemWr_inv, Direct, Unsigned, Shift, Store, Branch, MEMW
RE1,
                SetCTR, SetSO, SetOV, SetCA, ShiftRight, UseCA, SetCR, SelCR,
                ALUSel, msel, PCCond, SetLR, MTSPR, EXMFSPR, MFSPR, temp_SetSO,
                temp_SetOV,temp_SetCR, UseOE, UseRC, highbit_RA,
                highbit_RB, _HoldIFID, _HoldIDEX, _HoldEXMEM, _HoldMEMWB, IFIDBubble
,
                IDEXBubble, EXMEMBubble, MEMWBBubble, not_memLoadHazard, memLoadHaza
rd,
                IDLogical, temp_halt, MEMMFSPR, illegal_instruction_trap;

   initial
      begin
         $recordfile("test.sst"); // signalscan stuff
         $recordvars;
      end

   initial
      begin
         $readmemb("SRA.ROM", alu.maskgen.ROM.mem);
         $readmemb("ROM2", ROM2.mem);
      end // initial begin

   // ************************IF STAGE**************************
   nor5 pcnor (maybe_freeze, memLoadHazard, not_I_Cache_Hit, D_Cache_Req,
               temp_halt, trap_InsertBubble);
   or3$ freezer(FreezePC, branch_hazard, trap_SetPC, maybe_freeze);
   reg32e$ pc (clk, PCin, PC, , Reset, 1’b1, FreezePC);
   adder32 pcplus4 (PC, 32’d4, 1’b0, , NPC);

   // test if it’s a branch
   //   HOUSER 4/14/97
   //           Removed comments around branch identification code


   // bc/bca/bcl/bcla
   comp_6 c9(bc, Instruction[0:5], 6’b010000);
   //bcr & rti
   comp_6 c10IF(bcr1IF, Instruction[0:5], 6’b010011);
   comp_10 bcr_compIF(bcr2IF, Instruction[21:30], 10’d16);
   comp_10 rti_comp(rti2, Instruction[21:30], 10’d17);
   and2$ bcr_andIF(bcrIF, bcr1IF, bcr2IF);
   and2$ rti_and(rti, bcr1IF, rti2);

   or2$ branch_temp_or ( branch_temp, bcrIF, bc );

   and2$ br2(branch, branch_temp, I_Cache_Hit);
   inv1$ br3(not_branch, branch);

   // halt decode
   comp_6 halt_compare(is_halt, Instruction[0:5], 6’h3F);
   and3$ halt_and(temp_halt, is_halt, I_Cache_Hit, not_branch_hazard);


   //   HOUSER 4/16/97
   //   Make bcr_ok = bcr && I_Cache_Hit
   //   and2$ bcr_ok_and ( bcr_ok, bcr, I_Cache_Hit );
```

```
/*   HOUSER 4/15/97  COMMENTED OUT
 // old PC stuff
 mux2_32 bc_or_link_pc(BranchPCout, EXMEMPC, LROut, Link);
 mux2_32 branchpc(PCin, NPC, BranchPCout, TakeBranch);
 */



//   HOUSER: Added 4/14/97   --      branch predictor
BranchPredictorMod brpredict( _HoldIFID, clk, Reset, 1’b1 , PC[24:31],
                              UA[24:31],
                              UTaken, UTarget, UE, PredTaken, PredTarget );


//   Added 4/14/97 By Houser
//           Choosing the new PC value.
//           NEED TO ADD RTI

//   HOUSER 4/16/97
//   Modified to select branch pred. target if
//   branch AND PredTaken
and2$  choose_bp_and ( choose_bp, branch, PredTaken );

mux2_32      b_haz_mux ( b_haz_PC, IDEXNPC, NewPC, TakeBranch );

Choose_PC_Mod cpc ( trap_SetPC, 1’b0, rti, branch_hazard, choose_bp,
                    trap_Handler_Addr, LROut, SRR0, b_haz_PC, PredTarget,
                    NPC, PCin );



//*************************************************//
//                                                 //
//               Smart’s Creation                  //
//                                                 //
//*************************************************//


// Instruction cache
cachetlbseg I_Cache ( PC, I_Ext_Addr,
                      32’h0, Cache_Data_In,
                      Instruction, , 1’b1, , 1’b0,
                      I_Ext_Valid, I_Cache_Hit,
                      , , 1’b1, MSR[17],
                      IProtectViol, IPageFault, IAlignFault,
                      1’b0, , I_Grant, Ext_Reset, clk );

// Data cache
cachetlbseg D_Cache ( EXMEMOUT, D_Ext_Addr,
                      EXMEMRT, Cache_Data_In,
                      MemDataOut, D_Cache_Data_To_Mem,
                      MemWr, Ext_WE, EXMEMCTRL[29],
                      D_Ext_Valid, D_Cache_Hit,
                      IO_Data_In, IO_Data_Out, LoadOrStore, MSR[17],
                      DProtectViol, DPageFault_temp, DAlignFault,
                      Halt, Halt_Final, D_Grant, Ext_Reset, clk );

// External memory and bus
EXTERNAL2 External ( I_Ext_Addr, D_Ext_Addr,
                     128’h0, D_Cache_Data_To_Mem,
                     {Cache_Data_In[96:127], Cache_Data_In[64:95],
                      Cache_Data_In[32:63], Cache_Data_In[0:31]},
                     Ext_WE, I_Cache_Hit, D_Cache_Hit,
                     I_Cache_Req, D_Cache_Req,
                     I_Ext_Valid, D_Ext_Valid,
                     I_Grant, D_Grant,
                     IO_Data_In, IO_Data_Out,
                     1’b0, clk, Ext_Reset, Halt_Final,
                     Ext_Interrupt, 2’b00 );


   // Determine when the caches request the memory bus
   inv1$ IHit_Inv (not_I_Cache_Hit, I_Cache_Hit);
   inv1$ DHit_Inv (not_D_Cache_Hit, D_Cache_Hit);
   or2$ MEM_Load_or_Store ( LoadOrStore, EXMEMCTRL[0], EXMEMCTRL[34] );
   and2$ D_Cache_Request_a ( D_Cache_Req_t, LoadOrStore, not_D_Cache_Hit );
   or2$ D_Cache_Request_o ( D_Cache_Req, D_Cache_Req_t, Halt );
   inv1$ Not_D_Req ( not_D_Cache_Req, D_Cache_Req );
   assign I_Cache_Req = not_I_Cache_Hit;


   // Determine if the data cache’s page fault is real
   and2$ D_Cache_Page_Fault ( DPageFault, DPageFault_temp, LoadOrStore );


   inv1$ Reset_Inv ( Reset, Ext_Reset );


   // ******IF/ID Register******
   // assign _HoldIFID = not_memLoadHazard && ~branch_hazard;

   // Holds the writing of the IF/ID registers if the data
   // cache is waiting for the memory bus
   //   && ~branch_hazard
   //assign _HoldIFID = not_memLoadHazard  && not_D_Cache_Req && ~Halt;
   and3$ _HoldIFID_and ( _HoldIFID, not_memLoadHazard, not_D_Cache_Req, not_Halt );

   //
   //   HOUSER:  4/14/97
   //   Need to change this:
   //           not_I_Cache_Hit || branch_hazard || trap_insert_noop
   //
   //   OLD -- assign IFIDBubble = not_Cache_Hit;
   or5    if_id_bubble_or ( IFIDBubble, temp_halt, not_I_Cache_Hit,
                            branch_hazard, trap_IF_Inval, trap_InsertBubble );


   mux2_32 if_id_bubble(IFIDIRinput, Instruction, 32’d0, IFIDBubble); // inserts bub
ble noop
   reg32e$ if_id_ir(clk, IFIDIRinput, IFIDIR, , Reset, 1’b1, _HoldIFID);
   reg32e$ if_id_pc(clk, PC, IFIDPC, , Reset, 1’b1, _HoldIFID);
   reg32e$ if_id_npc(clk, NPC, IFIDNPC, , Reset, 1’b1, _HoldIFID);
   reg1 halt_reg(clk, temp_halt, IFIDHalt, Reset, _HoldIFID);

   //   HOUSER  4/14/97
   //   New registers for branch predictor
   dffh    if_id_PredTaken ( clk, PredTaken, IFIDPredTaken, , Reset, 1’b1, _HoldIFID
 );
   reg32e$ if_id_PredPC ( clk, PredTarget, IFIDPredTarget, , Reset, 1’b1, _HoldIFID
);
   reg1 valid_fetch(clk, I_Cache_Hit, Valid_Fetch, Reset, _HoldIFID);



   // ************************ID STAGE**************************
   assign RA = IFIDIR[11:15];
   assign RB = IFIDIR[16:20];
   assign RT = IFIDIR[6:10];
   inv1$ clk_inverter(not_clk, clk);
   gprregfile gpr(not_clk, RA, RB, RT, WR1, WR2, RA_OUT, RB_OUT, RT_OUT,
            WRDATA1, WRDATA2, WRE1, WRE2, Reset);
signext16_32 signextend(SignExt, IFIDIR[16:31]);

// gets signals from ROM files
OpcodeDecoder opdecode(IFIDIR[0:5], rom1_out);
extdecoder22_5 ething(IFIDIR[21:30], EXTADDR);
rom64b32w$ ROM2(EXTADDR, 1’b1, rom2_out);
assign temp3 = {rom1_out[1:63], 1’b0};
mux2_64 rommux(IFIDCTRLtemp, {rom1_out[1:63], 1’b0},
               rom2_out[0:63], rom1_out[0]);

// allows initialized IR registers
comp_6 nop(temp_noop, IFIDIR[0:5], 6’d0);
or2$ noop_or(NoOp, temp_noop, IFIDHalt);
// inserts Halt into control word
mux2_64 noopmux(IFIDCTRL, IFIDCTRLtemp[0:63], {62’d0, 1’b1, IFIDHalt}, NoOp);


// MEM hazards
inv1$ i9(not_Load, IDEXCTRL[0]);
inv1$ i10(not_Logical, IFIDCTRL[32]);
assign IDLogical = IFIDCTRL[23];
inv1$ i11(not_SelB, IFIDCTRL[4]);

// RA conflicts
comp_5 c11(memRTequalsRA, EXWR1, IFIDIR[11:15]);
and3$ ac7(memRTRAHazard, EXWRE1, memRTequalsRA, not_Logical);
comp_5 c12(memRAequalsRA, EXWR2, IFIDIR[11:15]);
and3$ ac8(memRARAHazard, EXWRE2, memRAequalsRA, not_Logical);
// RB conflicts
comp_5 c13(memRTequalsRB, EXWR1, IFIDIR[16:20]);
and3$ ac9(memRTRBHazard, EXWRE1, memRTequalsRB, not_SelB);
comp_5 c14(memRAequalsRB, EXWR2, IFIDIR[16:20]);
and3$ ac10(memRARBHazard, EXWRE2, memRAequalsRB, not_SelB);
// RS conflicts
comp_5 c15(memRTequalsRS, EXWR1, IFIDIR[6:10]);
and3$ ac11(memRTRSHazard, EXWRE1, memRTequalsRS, IDLogical);
comp_5 c16(memRAequalsRS, EXWR2, IFIDIR[6:10]);
and3$ ac12(memRARSHazard, EXWRE2, memRAequalsRS, IDLogical);
// Must bubble on a Load
or4$ o10(haz2, memRTRSHazard, memRARSHazard, memRTRAHazard, memRTRBHazard);
and2$ a9(memLoadHazard, haz2, IDEXCTRL[0]);
inv1$ loadinv(not_memLoadHazard, memLoadHazard);
// Store hazards
and3$   store_haz_and1(memRTRTStoreHazard, memRTequalsRS, EXWRE1, IFIDCTRL[34]);
and3$   store_haz_and2(memRTRAStoreHazard, memRAequalsRS, EXWRE2, IFIDCTRL[34]);

// WB Hazards
// RA load conflicts
comp_5 c17(wbRTequalsRA, MEMWR1, IFIDIR[11:15]);
and3$ ac13(wbRTRAHazard, MEMWRE1, wbRTequalsRA, not_Logical);
comp_5 c18(wbRAequalsRA, MEMWR2, IFIDIR[11:15]);
and3$ ac14(wbRARAHazard, MEMWRE2, wbRAequalsRA, not_Logical);
// RB load conflicts
comp_5 c19(wbRTequalsRB, MEMWR1, IFIDIR[16:20]);
and3$ ac15(wbRTRBHazard, MEMWRE1, wbRTequalsRB, not_SelB);
comp_5 c20(wbRAequalsRB, MEMWR2, IFIDIR[16:20]);
and3$ ac16(wbRARBHazard, MEMWRE2, wbRAequalsRB, not_SelB);
// RS load conflicts
comp_5 c21(wbRTequalsRS, MEMWR1, IFIDIR[6:10]);
and3$ ac17(wbRTRSHazard, MEMWRE1, wbRTequalsRS, IDLogical);
comp_5 c22(wbRAequalsRS, MEMWR2, IFIDIR[6:10]);
and3$ ac18(wbRARSHazard, MEMWRE2, wbRAequalsRS, IDLogical);
// Store hazards
and3$   store_haz_and3(wbRTRTStoreHazard, wbRTequalsRS, MEMWRE1, IFIDCTRL[34]);
```

```
   and3$   store_haz_and4(wbRTRAStoreHazard, wbRAequalsRS, MEMWRE2, IFIDCTRL[34]);

   // ILLEGAL INSTRUCTION EXCEPTION
   inv1$ not_valid(illegal_instruction_trap, IFIDCTRL[62]);


   // ******ID/EX Register******
   // assign _HoldIDEX = 1’b1;

   // Holds the writing of the ID/EX registers if the data
   // cache is waiting for the memory bus
   assign _HoldIDEX = not_D_Cache_Req;


   //
   //   HOUSER:  4/14/97
   //   Need to change this:
   //           memLoadHazard || branch_hazard || trap_insert_noop
   //
   //   OLD -- assign IDEXBubble = memLoadHazard;
   or3$    id_ex_bubble_or ( IDEXBubble, memLoadHazard, branch_hazard, trap_ID_Inval
);


   mux2_64 id_ex_bubble2(IDEXCTRLin, IFIDCTRL, {62’d0, 1’b1, temp_halt}, IDEXBubble)
; // inserts bubble to control
   reg64e_pipe id_ex_ctrl(clk, IDEXCTRLin, IDEXCTRL, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_pc(clk, IFIDPC, IDEXPC, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_npc(clk, IFIDNPC, IDEXNPC, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_ir(clk, IFIDIR, IDEXIR, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_RA(clk, RA_OUT, IDEXRA, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_RB(clk, RB_OUT, IDEXRB, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_RT(clk, RT_OUT, IDEXRTtemp, , Reset, 1’b1, _HoldIDEX);
   reg32e$ id_ex_D(clk, SignExt, IDEXD, , Reset, 1’b1, _HoldIDEX);
   reg8e id_ex_EXHaz(clk, {memRTRBHazard, memRARBHazard,
                           memRTRSHazard, memRARSHazard,
                           memRTRAHazard, memRARAHazard,
                           memLoadHazard, 1’b0}, IDEXHAZ, , Reset, 1’b1, _HoldIDEX);
   reg8e id_ex_MEMHaz(clk, {wbRTRBHazard, wbRARBHazard,
                           wbRTRSHazard, wbRARSHazard,
                           wbRTRAHazard, wbRARAHazard,
                           2’b0}, IDEXMEMHAZ, , Reset, 1’b1, _HoldIDEX);
   reg4e id_ex_StoHaz(clk, {memRTRTStoreHazard, memRTRAStoreHazard,
                              wbRTRTStoreHazard, wbRTRAStoreHazard},
                        IDEXSTOHAZ, , Reset, 1’b1, _HoldIDEX);



   //   HOUSER  4/14/97
   //   New registers for branch predictor
   dffh    id_ex_PredTaken ( clk, IFIDPredTaken, IDEXPredTaken, , Reset, 1’b1, _Hold
IDEX );
   reg32e$ id_ex_PredPC ( clk, IFIDPredTarget, IDEXPredTarget, , Reset, 1’b1, _HoldI
DEX );
   //   dffh    id_ex_IsBranch ( clk, IFIDIsBranch, IDEXIsBranch, , Reset, 1’b1, _Ho
ldIDEX );




   // ***********************EX STAGE*************************
   // hard coded logic
   assign Load = IDEXCTRL[0];
   assign Direct = IDEXCTRL[1];
   assign temp_SelA[0:1] = IDEXCTRL[2:3];
   assign SelB = IDEXCTRL[4];
   assign ALUOp[0:3] = IDEXCTRL[5:8];
   assign ALUSel = IDEXCTRL[9];
   assign msel = IDEXCTRL[10];
   assign UseOE = IDEXCTRL[11];
   assign temp_SetSO = IDEXCTRL[13];
   assign temp_SetOV = IDEXCTRL[14];
   assign SetCA = IDEXCTRL[15];
   assign UseCA = IDEXCTRL[16];
   assign cin = IDEXCTRL[17];
   assign ShiftRight = IDEXCTRL[18];
   assign UseRC = IDEXCTRL[12];
   assign Unsigned = IDEXCTRL[19];
   //   assign SetLR = IDEXCTRL[20];
   assign temp_SetCR = IDEXCTRL[21];
   assign SelCR = IDEXCTRL[22];
   assign Link = IDEXCTRL[23];
   assign PCCond = IDEXCTRL[24];
   assign MTSPR = IDEXCTRL[25];
//   assign Branch = IDEXCTRL[35];
   or2$  branch_or (Branch, IDEXCTRL[35], bcr);

   assign EXMFSPR = IDEXCTRL[28];

   assign EXWR1 = IDEXIR[6:10];

   assign EXWR2 = IDEXIR[11:15];

   // check if RA == 0 and it's a load or store, if so, use 0
   comp_5 c0(zero1, IDEXIR[11:15], 5'd0);
   or2$ usezero(UseZero, Load, Store);
   and2$ z3(SelZero, zero1, UseZero);
   mux2_2 z4(SelA, temp_SelA, {1'b0,1'b1}, SelZero);


   // Mux Select A w/ hazards
   ASelector asel(IDEXMEMHAZ, IDEXHAZ, SelA, ASelect);
   mux8_32 muxa(ALU_RA, IDEXRA, 32'h0, IDEXRT, MEM_WRDATA1, MEM_WRDATA2, WRDATA1, WRDATA
2, ,
                ASelect[0], ASelect[1], ASelect[2]);

   // Mux Select B w/ hazards
   BSelector bsel(IDEXMEMHAZ, IDEXHAZ, SelB, BSelect);
   mux8_32 muxb(ALU_RB, IDEXRB, IDEXD, MEM_WRDATA1, MEM_WRDATA2, WRDATA1, WRDATA2,,,
                BSelect[0], BSelect[1], BSelect[2]);

   // selects carry for extended instructions
   mux2$ carrysel(carry, cin, XEROUT[2], UseCA);
   // gets rid of high bits for a cmpl
   mux2$ cmplsel1(highbit_RA, ALU_RA[0], 1'b0, Unsigned);
   mux2$ cmplsel2(highbit_RB, ALU_RB[0], 1'b0, Unsigned);

   // the REAL thing baby
   SuperALU alu(IDEXIR, {highbit_RA, ALU_RA[1:31]}, {highbit_RB, ALU_RB[1:31]},
                IDEXRT, ALUOp, msel, carry, ALUSel, ShiftRight, Out, cout, ovf);

   // determines branch attributes
   Branch_Logic bl(Branch, IDEXIR[6:10], DecCTR, ZeroCTR, OnFalse, OnTrue, UseCTR);

   mux1_32 m7(BCBit, CR, IDEXIR[11:15]);
   inv1$ i77(not_BCBit, BCBit);

   // determine BranchTaken
   and3$ taken_a1 ( take_temp1, OnFalse, not_BCBit, Branch );
   and3$ taken_a2 ( take_temp2, OnTrue, BCBit, Branch );
```

```
   or2$ taken_o1 ( TakeBranch, take_temp1, take_temp2 );

   comp_6 c10(bcr1, IDEXIR[0:5], 6'b010011);
   comp_10 bcr_comp(bcr2, IDEXIR[21:30], 10'd16);
   and2$ bcr_and(bcr, bcr1, bcr2);


   or2$ o1(SetCTR, DecCTR, LoadCTR);
   // branch adder
   or2$ dir(direct_branch, IDEXIR[30], Direct);
   mux2_32 pcrelative(PCOp, IDEXPC, 32'h0, direct_branch);
   adder32 pcadd(PCOp, {IDEXD[0:29],2'b00}, 1'b0, , NewPCtemp);
   and2$ LR_SET_AND ( SetLR, Branch, IDEXIR[31] );

   mux2_32 NewPCMux ( NewPC, NewPCtemp, LROut, bcr );


   //***********SPR***********
   // selects load an SPR
   SPR_Logic spr(IDEXIR[11:15], MTSPR, MFSPR, LoadLR, LoadCTR, LoadXER, FromLR,
                 FromCTR, FromXER);
   encoder4_2 e1({FromLR, FromCTR, FromXER, 1'b0}, SelSPR);
   mux3_32 sprselect(IDEXSPR, LROut, CTROUT, XEROUT, SelSPR[0], SelSPR[1]);

   // checks if it should be checking this
   and2$ a0(temp_SetSO2, UseOE, temp_SetSO);
   and2$ a00(temp_SetOV2, UseOE, temp_SetOV);
   // checks OE bit before setting all of XER
   and2$ a1(SetSO, temp_SetSO2, IDEXIR[21]);
   and2$ a2(SetOV, temp_SetOV2, IDEXIR[21]);
   // XER register
   XER_Register xer(clk, Reset, ovf, cout, IDEXRT, SetSO, SetOV, SetCA, LoadXER, XER
OUT);

   // CTR register
   CTR_Register ctr(clk, Reset, IDEXIR[6:10], IDEXRT, CTROUT, SetCTR, {LoadCTR, DecC
TR});

   // LR register
   LR_Register lr(clk, Reset, IDEXNPC, IDEXRTtemp, LROut, SetLR, LoadLR);

   // should also check if it should be checking this
   and2$ a6(temp_SetCR2, temp_SetCR, UseRC);
   // checks Rc bit before setting CR register
   and2$ a3(SetCR, temp_SetCR2, IDEXIR[31]);
   // CR register
   CR_Register cr(clk, Reset, IDEXRA[0], IDEXRB[0], Out, XEROUT[2], ovf, SetCR,
               SelCR, IDEXIR[6:8], Unsigned, CR);

   // control for write enables
   assign EXWRE1 = IDEXCTRL[30];
   assign temp_WRE2 = IDEXCTRL[31];
   assign WB_Load = IDEXCTRL[0];
   assign MFSPR = IDEXCTRL[28];
   assign Logical = IDEXCTRL[32];
   assign Shift = IDEXCTRL[33];
   assign Store = IDEXCTRL[34];
   // logic to determine when to write RA
   // RA==0
   comp_5 c1(zero, EXWR2, 5'd0);
   inv1$ iwb0(not_zero, zero);
   // RA==RT
   comp_5 c2(match_RT, EXWR1, EXWR2);
   and2$ awb4(temp_match_RT, match_RT, WB_Load);
   inv1$ iwb1(not_match_RT, temp_match_RT);
   // store -> RA
   and3$ awb0(ok_store, Store, temp_WRE2, not_zero);
   // mfspr -> RA
   and2$ awb3(ok_mfspr, MFSPR, temp_WRE2);
   // shift -> RA
   and2$ awb1(ok_shift, Shift, temp_WRE2);
   // load -> RA
   and4$ awb2(ok_load, WB_Load, temp_WRE2, not_zero, not_match_RT);
   or4$ owb1(tempwre2, ok_store, ok_shift, ok_load, ok_mfspr);
   or2$ owb2(EXWRE2, tempwre2, Logical);


   /*   //      HOUSER  4/14/97
    //  Moved branch stuff from MEM stage to here

    mux1_32 m7(BCBit, CR, IDEXIR[11:15]);
    inv1$ i77(not_BCBit, BCBit);
    inv1$ i88(not_IR6, IDEXIR[6]);
    inv1$ i999(not_IR7, IDEXIR[7]);
    and3$ a4(FalseBr, not_IR6, not_IR7, IDEXIR[8]);
    mux2$ m8(BCok, BCBit, not_BCBit, FalseBr);

    //  HOUSER 4/14/97  CHANGED TO MAKE SURE INSTRUCTION IS A BRANCH
    //   and2$ a5(TakeBranch, PCCond, BCok);
   and3$ a5(TakeBranch, PCCond, BCok, Branch);
    */

   //   HOUSER 4/14/97:
   //           If branch is taken, compare target address to predicted target address
   //           Also, check to see if we predicted whether the branch was taken correctl
y

   //   Compare TakeBranch to prediction
   //           b_hazard = NAND ( COMPARE ( TakeBranch, IDEXPredTaken ),
   //                        COMPARE ( NewPC, IDEXPredTarget ) );

   comp_32      branch_pc_compare ( address_correct, NewPC, IDEXPredTarget );
   inv1$        branch_pc_inv1 ( address_correct_bar, address_correct );

   comp_1       branch_taken_compare ( taken_correct, TakeBranch, IDEXPredTaken );
   inv1$        branch_taken_inv ( taken_correct_bar, taken_correct );

   and2$        branch_hazard_detect1 ( temp_branch_hazard, address_correct_bar, TakeBra
nch );
   or2$         branch_hazard_detect2 ( temp_branch_hazard2, temp_branch_hazard,
                                        taken_correct_bar );
   and2$        branch_hazard_detect3 ( branch_hazard, temp_branch_hazard2, Branch );
   inv1$  invert_branch_haz (not_branch_hazard, branch_hazard);



   //   HOUSER 4/14/97
   //           Update the branch predictor if the instruction is a branch
   assign UA = IDEXPC;
   assign UE = Branch;
   assign UTaken = TakeBranch;
   assign UTarget = NewPC;




   // ******EX/MEM Register******
   // assign _HoldEXMEM = 1'b1;

   // Holds the writing of the EX/MEM registers if the data
   // cache is waiting for the memory bus
   assign _HoldEXMEM = not_D_Cache_Req;


   //
   //   HOUSER:  4/14/97
   //
   assign EXMEMBubble = trap_EX_Inval;

   mux3_32 store_haz_ex(IDEXRT, IDEXRTtemp, WRDATA1, WRDATA2, IDEXSTOHAZ[2], IDEXSTO
HAZ[3]);

   mux2_64 ex_mem_bubble(EXMEMCTRLin, IDEXCTRL, {36'd0, IDEXCTRL[36], 27'd0}, EXMEMB
ubble); // inserts bubble to control
   reg64e_pipe ex_mem_ctrl(clk, EXMEMCTRLin, EXMEMCTRL, , Reset, 1'b1, _HoldEXMEM);
   reg32e$ ex_mem_pc(clk, NewPC, EXMEMPC, , Reset, 1'b1, _HoldEXMEM);
   reg32e$ ex_mem_npc(clk, IDEXPC, EXMEMNPC, , Reset, 1'b1, _HoldEXMEM);
   reg32e$ ex_mem_ir(clk, IDEXIR, EXMEMIR, , Reset, 1'b1, _HoldEXMEM);
   reg32e$ ex_mem_out(clk, Out, EXMEMOUT, , Reset, 1'b1, _HoldEXMEM);
   reg32e$ ex_mem_RT(clk, IDEXRT, EXMEMRTtemp, , Reset, 1'b1, _HoldEXMEM);
   reg32e$ ex_mem_SPR(clk, IDEXSPR, EXMEMSPR, , Reset, 1'b1, _HoldEXMEM);
   reg1 ex_mem_WRE2(clk, EXWRE2, MEMWRE2, Reset, _HoldMEMWB);
   reg4e ex_mem_StoHaz(clk, IDEXSTOHAZ, EXMEMSTOHAZ, , Reset, 1'b1, _HoldEXMEM);


   // ***********************MEM STAGE*************************
   // hard coded load logic
   assign MemWr_inv = EXMEMCTRL[26];
   inv1$ idcache0 ( MemWr, MemWr_inv );
   assign byte = EXMEMCTRL[29];
   assign MEMWR1 = EXMEMIR[6:10];
   assign MEMWR2 = EXMEMIR[11:15];
   assign MEMWRE1 = EXMEMCTRL[30];

   /*   HOUSER  4/14/97
    Moved to EX stage

    // execute branch logic - redone by houser
    // moved to EX stage?

    mux1_32 m7(BCBit, CR, EXMEMIR[11:15]);
    inv1$ i77(not_BCBit, BCBit);
    inv1$ i88(not_IR6, EXMEMIR[6]);
    inv1$ i999(not_IR7, EXMEMIR[7]);
    and3$ a4(FalseBr, not_IR6, not_IR7, EXMEMIR[8]);
    mux2$ m8(BCok, BCBit, not_BCBit, FalseBr);
    and2$ a5(TakeBranch, PCCond, BCok);

    */

   mux3_32 store_haz(EXMEMRT, EXMEMRTtemp, WRDATA1, WRDATA2, EXMEMSTOHAZ[0], EXMEMST
OHAZ[1]);

   // same as the wb mux, but for data forwarding - doesn't have wbdata
   mux2_32 memselect(MEM_WRDATA1, EXMEMOUT, EXMEMSPR, EXMEMCTRL[28]);
   assign MEM_WRDATA2 = EXMEMOUT;

   // HOUSER 4/17/97
   // Trap and Interrupt handler

   // IAlignFault = !byte && ( Addr[30] || Addr[31] )

   assign IF_Exc[0] = IPageFault;
   assign IF_Exc[1] = IPageFault;
   assign IF_Exc[2] = IProtectViol;
   assign IF_Exc[3] = 1’b0;
   assign IF_Exc[4] = IAlignFaultTrue;
   and2$ IAlignFault_a ( IAlignFaultTrue, IAlignFault, I_Cache_Hit );

   assign ID_Exc = illegal_instruction_trap;

   // DAlignFault = !byte && ( Addr[30] || Addr[31] )
   assign M_Exc[0] = DPageFault;
   assign M_Exc[1] = DPageFault;
   assign M_Exc[2] = DProtectViol;
   assign M_Exc[3] = 1’b0;
   assign M_Exc[4] = DAlignFaultTrue;
   and2$ DAlignFault_a ( DAlignFaultTrue, DAlignFault, LoadOrStore );

   assign IO_Int[0] = Ext_Interrupt[0];
   assign IO_Int[1] = Ext_Interrupt[1];

   trap_interrupt_M  trapMod ( clk, Reset,
                               1’d1, PC, IFIDPC, EXMEMPC, IF_Exc, ID_Exc,
                               M_Exc, IO_Int,
                               trap_SaveState, trap_IF_Inval, trap_ID_Inval,
                               trap_EX_Inval, trap_M_Inval,
                               trap_InsertBubble, trap_SetPC, trap_PC_Val,
                               trap_Handler_Addr );


   // Set SRR0 and SRR1 if interrupt
   reg32e$ SRR0_reg ( clk, trap_PC_Val,  SRR0, , Reset, 1’b1, trap_SaveState );
   reg32e$ MSR_reg ( clk, SRR1, MSR, , 1’b1, Reset, rti );
   reg32e$ SRR1_reg ( clk, MSR, SRR1, , 1’b1, Reset, trap_SaveState );

   // ******MEM/WB Register******
   assign _HoldMEMWB = not_D_Cache_Req;
   //   assign MEMWBBubble = 1’b0;
   //   assign MEMWBBubble = not_D_Cache_Hit && EXMEMCTRL[0] && trap_M_Inval;
   and3$ MEMWBBubble_and ( MEMWBBubble, not_D_Cache_Hit, EXMEMCTRL[0], trap_M_Inval );

   mux2_64 mem_wb_bubble(MEMWBCTRLin, EXMEMCTRL,
                  {36’d0, EXMEMCTRL[36], 27’d0}, MEMWBBubble); // inserts bubble to cont
rol
   reg64e_pipe mem_wb_ctrl(clk, MEMWBCTRLin, MEMWBCTRL, , Reset, 1’b1, _HoldMEMWB);
   reg32e$ mem_wb_ir(clk, EXMEMIR, MEMWBIR, , Reset, 1’b1, _HoldMEMWB);
   // loaded data
   reg32e$ mem_wb_data(clk, MemDataOut, WBData, , Reset, 1’b1, _HoldMEMWB);
   // alu data
   reg32e$ mem_wb_out(clk, EXMEMOUT, MEMWBOUT, , Reset, 1’b1, _HoldMEMWB);
   // special register
   reg32e$ mem_wb_SPR(clk, EXMEMSPR, MEMWBSPR, , Reset, 1’b1, _HoldMEMWB);
   // uses precalculated WRE2
   reg1 mem_wb_WRE2(clk, MEMWRE2, WRE2, Reset, _HoldMEMWB);


   // ************************WB STAGE**************************
   assign MemToReg = MEMWBCTRL[27];
   assign MEMMFSPR = MEMWBCTRL[28];
   assign WRE1 = MEMWBCTRL[30];
   assign WR1 = MEMWBIR[6:10];
   assign WR2 = MEMWBIR[11:15];
   assign Halt = MEMWBCTRL[63];
   inv1$ Halt_inv ( not_Halt, Halt );

   mux4_32 wbselect(WRDATA1, WBData, /*IN1*/ , MEMWBOUT, MEMWBSPR ,MemToReg, MEMMFSPR);

   assign WRDATA2 = MEMWBOUT;
```

```
   endmodule // top
```

```
// 256 byte, 2-way set associative, 4-word block cache
module cache256b2a4w( addr_in, addr_out, data_in, data_mem_in,
                      data_out, data_mem_out, write_en, mem_we,
                      write_size, valid_in_t, valid_out,
                      io_datain, io_dataout, cache_active,
                      flush, halt, grant, reset, clk );
   input [0:31]   addr_in, data_in, io_datain;
   input [0:127]  data_mem_in;
   input          write_en, write_size, valid_in_t, cache_active,
                  flush, grant, reset, clk;
   output [0:31]  addr_out, data_out, io_dataout;
   output [0:127] data_mem_out;
   output         mem_we, valid_out, halt;

   wire [0:31]    addr_out_temp, tag0_out, tag1_out,
                  comp0_operand, comp1_operand;
   wire [0:127]   data0_out, data1_out, data_in_temp, data_mem_out_temp;
   wire [0:1]     write_size_temp;
   wire [0:31]    data_out_t;
   wire [0:2]     index, index_temp;
   wire [0:3]     offset, offset_temp;

   // Cache flush wires
   wire [0:7]     flush_count;
   wire [0:31]    flush_addr, flush_tag;
   wire [0:2]     flush_index;
   wire [0:3]     flush_offset;
   wire           flush_set, flush_we, flush_valid_in, flush_valid_in_temp;


   // 32-bit addresses, 1-word data in and data out
   //  valid bit: 0 if cache data not valid
   //             1 if cache data valid
   //  dirty bit: 0 if cache line clean
   //             1 if cache line modified on write
   //    LRU bit: 0 if most-recently used
   //             1 if least-recently used (replace first)
   // write_size:
   //         not used:    00 - write quadword
   //         not used:    01 - write doubleword
   //             0 - write word
   //             1 - write byte
   //   write_en: active low
   //     mem_we: active high
   // Need to use clk signal to time how long the WE’s are active
   //   only one clock cycle

   ///////////////////
   //
   //  Flush logic
   //

   // Counter for index and set
   nand2$ flush_en_nand ( flush_en_pre, flush_we, flush_valid_in );
   or2$ flush_en_clr_and ( flush_en_clr, flush_we_inv, flush_we_delay );

   inv1$ flush_en_clr_inv ( flush_we_inv, flush_we );
   buffer$ flush_en_clr_b0 ( flush_we_delay0, flush_we );
   buffer$ flush_en_clr_b1 ( flush_we_delay1, flush_we_delay0 );
   buffer$ flush_en_clr_b2 ( flush_we_delay2, flush_we_delay1 );
   buffer$ flush_en_clr_b3 ( flush_we_delay, flush_we_delay2 );


   dff$ flush_en_dff ( clk_inv, 1’b0, flush_en_hold, , flush_en_clr, flush_en_pre );
   or4$ flush_en_or ( flush_en_temp, flush_valid_in, flush_en_hold,
                      flush_tag_invalid, flush_tag_not_dirty );
   and2$ flush_en_and1 ( flush_en, flush_en_temp, flush );

   syn_cntr8$ flush_counter ( clk, 1’b1, 8’b0, flush_en, 1’b1,
                              reset, 1’b1, , flush_count);

   assign halt = flush_count[3];
   assign flush_index = flush_count[4:6];
   assign flush_offset = 4’b0;
   assign flush_set = flush_count[7];

   // Figure out the address to write to
   mux2_32 flush_tag_mux ( flush_tag, tag0_out, tag1_out, flush_set );
   assign flush_addr = { flush_tag[0:24], flush_index, flush_offset };

   // Select the address and data
   mux2_32 flush_addr_mux ( addr_out, addr_out_temp, flush_addr, flush );

   // Memory write enable
   mux2$ flush_tag_valid_mux ( flush_tag_valid, tag0_valid_out,
                                 tag1_valid_out, flush_set );
   inv1$ flush_tag_valid_inv ( flush_tag_invalid, flush_tag_valid );
   mux2$ flush_tag_dirty_mux ( flush_tag_dirty, tag0_dirty_out,
                               tag1_dirty_out, flush_set );
   inv1$ flush_tag_dirty_inv ( flush_tag_not_dirty, flush_tag_dirty );
   and2$ flush_we_and ( flush_we_dff_in, flush_tag_valid, flush_tag_dirty );

   inv1$ flush_we_clr_inv ( flush_we_clr_temp, valid_in_t );
   and2$ flush_we_clr_and ( flush_we_clr, flush_we_clr_temp, reset_inv );

   dff$ flush_we_dff ( clk_inv, flush_we_dff_in, flush_we0, , flush_we_clr, 1’b1 );
   buffer$ flush_we_b0 ( flush_we1, flush_we0 );
   buffer$ flush_we_b1 ( flush_we2, flush_we1 );
   buffer$ flush_we_b2 ( flush_we3, flush_we2 );
   buffer$ flush_we_b3 ( flush_we4, flush_we3 );
   buffer$ flush_we_b4 ( flush_we5, flush_we4 );
   buffer$ flush_we_b5 ( flush_we, flush_we5 );

   // Select the correct inputs when a flush is requested
   mux2_3 index_select_mux ( index, index_temp, flush_index, flush );
   or2$ offset_select_or ( flush_or_writeback, flush, mem_we_addr );
   mux2_4 offset_select_mux ( offset, offset_temp, flush_offset, flush_or_writeback
);
   mux2$ mem_we_select_mux ( mem_we, mem_we_temp, flush_we, flush );

   // Get the valid_in signal
   and2$ flush_valid_in_and ( flush_valid_in, valid_in_t, flush );

   //
   //
   ///////////////////


   ///////////////////
   //
   //  I/O logic
   //

   // Redirect the valid_in signal if I/O
   and3$ activate_cache ( valid_in, valid_in_t, cache_active, grant );

   // Redirect the valid_out signal if I/O
   inv1$ io_or_data ( io, cache_active );
   and2$ io_valid_a ( io_valid, io, valid_in );
   mux2$ io_valid_out ( valid_out, valid_out_t, io_valid, io );
```

```
// Redirect the incoming data word
//   demux2_32 io_data_in_d ( data_in, data_in_t, io_datain, io );
assign io_datain = data_in;

// Redirect the outgoing data word
mux2_32 io_data_out_m ( data_out, data_out_t, io_dataout, io );

// Redirect memory write enable
inv1$ io_mem_we_i ( write_en_inv, write_en );
mux2$ io_mem_we ( mem_we_temp, mem_we_t, write_en_inv, io );

//
//
////////////////////


// Determine if the data is coming from the CPU or memory
assign data_in_temp = data_mem_in;


assign index_temp = addr_in[25:27];
assign offset_temp = addr_in[28:31];


// Tag and LRU bits
tagblock tag0 ( index, addr_in[0:24], tag0_out, tag0_we,
                tag0_valid_in, tag0_valid_out, tag0_valid_we,
                tag0_dirty_in, tag0_dirty_out, tag0_dirty_we,
                tag0_we_stable, reset, clk );
tagblock tag1 ( index, addr_in[0:24], tag1_out, tag1_we,
                tag1_valid_in, tag1_valid_out, tag1_valid_we,
                tag1_dirty_in, tag1_dirty_out, tag1_dirty_we,
                tag1_we_stable, reset, clk );
bitblock lru_bits ( index, lru_in, lru_out, lru_we,
                    reset, clk_inv );


// Data bits
datablock data0 ( index, offset, data_in, data_in_temp, data0_out,
                  data0_we, write_size_temp, reset, clk );
datablock data1 ( index, offset, data_in, data_in_temp, data1_out,
                  data1_we, write_size_temp, reset, clk );


// Tag comparisons
compblock compare0 ( comp0_out, valid_in, tag0_valid_out, tag0_out,
                     addr_in, reset, clk );
compblock compare1 ( comp1_out, valid_in, tag1_valid_out, tag1_out,
                     addr_in, reset, clk );

or2$ valid_data ( valid_out_temp, comp0_out, comp1_out );
inv1$ i0_clk ( clk_inv, clk );
inv1$ i1_reset ( reset_inv, reset );
dff$ dff0_stabilize ( clk_inv, valid_out_temp, valid_out_t, ,
                      reset_inv, 1'b1 );


// Select the set to read data from
mux2$ data_out_sel_mux ( data_mem_out_sel, set, flush_set, flush );
mux2_128 data_out_mux ( { data_mem_out[96:127], data_mem_out[64:95],
                          data_mem_out[32:63], data_mem_out[0:31] },
                        data0_out, data1_out, data_mem_out_sel );
assign data_out_t = data_mem_out[96:127];
```

```
   // Cache control logic
   logicblock control ( write_en, write_size, set, comp0_out, comp1_out,
                        valid_out_temp,
                        lru_out, tag0_dirty_out, tag1_dirty_out, dirty_out,
                        valid_in, reset, clk,
                        readhit, readmiss, writehit, writemiss,
                        tag0_we, tag0_valid_in, tag0_valid_we,
                        tag0_dirty_in, tag0_dirty_we,
                        tag1_we, tag1_valid_in, tag1_valid_we,
                        tag1_dirty_in, tag1_dirty_we,
                        data0_we, data1_we,
                        lru_in, lru_we,
                        mem_we_t, mem_we_addr,
                        write_size_temp, cache_active );


   // Determine the address to external memory
   mux4_32 addr_out_mux ( addr_out_temp,
                          {  addr_in[0:24], index, 4'h0 },
                          {  addr_in[0:24], index, 4'h0 },
                          { tag0_out[0:24], index, 4'h0 },
                          { tag1_out[0:24], index, 4'h0 },
                          mem_we_addr, set );

endmodule // cache256b2a4w


module compblock( comp, valid, tag_valid, tag, addr, reset, clk );
   input        valid, tag_valid;
   input [0:31] tag, addr;
   input        reset, clk;
   output       comp;

   // Invert reset
   inv1$ i0_reset ( reset_inv, reset );

   // Compare tag bits with address
   comp_26 c0_comp ( comp_temp, { 1'b1, addr[0:24] }, {tag_valid, tag[0:24] } );

   // Latch the result to ensure stable output
   latch$ l0_comp ( comp_temp, clk, comp, , reset_inv, 1'b1 );

endmodule // compblock


module tagblock( index, tag_in, tag_out, tag_we,
                 valid_in, valid_out, valid_we,
                 dirty_in, dirty_out, dirty_we,
                 tag_we_stable, reset, clk );
   input [0:2]   index;
   input [0:24]  tag_in;
   input         valid_in, dirty_in;
   input         tag_we, valid_we, dirty_we;
   input         reset, clk;
   output [0:31] tag_out;
   output        tag_we_stable, valid_out, dirty_out;

   // Invert reset
   //inv1$ i0_reset ( reset_inv, reset );

   // 28-bits per cache line = 4 + 8*3 = 4*8
   bitblock valid_bits ( index, valid_in, valid_out, valid_we, reset, clk );
   bitblock dirty_bits ( index, dirty_in, dirty_out, dirty_we, reset, clk );
   // Stabilize write enables for the RAM's
   // @posedge, clk 1 --> force to 1 -> 1 on clr, 0 on pre
   // @negedge, clk 0 --> latch d -> 1 on clr, 1 on pre
   inv1$ i0_stabilize ( clk_inv, clk );
   inv1$ i1_stabilize ( clk_inv2, clk_inv );
   buffer$ b0_stabilize ( clk_buf1, clk_inv2 );
   buffer$ b1_stabilize ( clk_buf2, clk_buf1 );
   buffer$ b2_stabilize ( clk_buf3, clk_buf2 );
   buffer$ b3_stabilize ( clk_buf4, clk_buf3 );
   buffer$ b4_stabilize ( clk_buf_final, clk_buf4 );
   or2$ o0_stabilize ( clk_pre, clk_inv, clk_buf2 );

   xor2$ x0_stabilize ( tag_we_mask, clk, clk_buf_final );
   inv1$ i2_stabilize ( tag_we_mask_inv, tag_we_mask );

   dff$ dff0_stabilize ( clk_inv, tag_we, tag_we_stable_temp, tag_we_stable_inv,
                         1'b1, clk_pre );

   or3$ o1_stabilize ( tag_we_stable, tag_we_stable_temp, tag_we_mask_inv, clk );


   // Rams
   ram16b8w$ tag0 ( index, tag_in[0:15], 1'b0,
                    tag_we_stable, tag_we_stable, tag_out[0:15] );
   ram16b8w$ tag1 ( index, {tag_in[16:24], 7'b0}, 1'b0,
                    tag_we_stable, tag_we_stable, tag_out[16:31] );

endmodule // tagblock


module bitblock( index, in, out, we, reset, clk );
   input [0:2] index;
   input       in, we, reset, clk;
   output      out;

   wire [0:7]  lru_debug, bitblock_debug;

   // Invert reset
   inv1$ i0_reset ( reset_inv0, reset );
   inv1$ i1_reset ( reset_inv1, reset );

   // Invert clock
   inv1$ i0_clock ( clk_inv0, clk );
   inv1$ i1_clock ( clk_inv1, clk );

   // Buffer input
   buffer$ b0_in ( in0, in );
   buffer$ b1_in ( in1, in );

   demux8 d0 ( we, we0, we1, we2, we3, we4, we5, we6, we7,
               index[0], index[1], index[2] );

   // Registers
   dffh df0 ( clk_inv0, in0, out0, , reset_inv0, 1'b1, we0 );
   dffh df1 ( clk_inv0, in0, out1, , reset_inv0, 1'b1, we1 );
   dffh df2 ( clk_inv0, in0, out2, , reset_inv0, 1'b1, we2 );
   dffh df3 ( clk_inv0, in0, out3, , reset_inv0, 1'b1, we3 );
   dffh df4 ( clk_inv1, in1, out4, , reset_inv1, 1'b1, we4 );
   dffh df5 ( clk_inv1, in1, out5, , reset_inv1, 1'b1, we5 );
   dffh df6 ( clk_inv1, in1, out6, , reset_inv1, 1'b1, we6 );
   dffh df7 ( clk_inv1, in1, out7, , reset_inv1, 1'b1, we7 );

   assign bitblock_debug = { out0, out1, out2, out3, out4, out5, out6, out7 };

   mux8 m0 ( out, out0, out1, out2, out3, out4, out5, out6, out7,
             index[0], index[1], index[2] );

endmodule // bitblock


module datablock( index, offset, in_word, in, out, we, write_size, reset, clk );
   input [0:2]    index;
   input [0:3]    offset;
   input [0:31]   in_word;
   input [0:127]  in;
   input          we, reset, clk;
   input [0:1]    write_size;
   output [0:127] out;

   wire [0:127]   mux_in, ram_in, ram_in_0, ram_in_1;
   wire [0:15]    off1, off2, off3;
   wire [0:15]    we_temp, we_stable, we_stable_temp,
                  we_byte, wbt, we_word, we_qword;


   // Write enable logic
   ext1_16 wqword ( we_qword, we );
   inv1$ i0 ( we_inv, we );
   demuxinv4_4 wword ( {4{we_inv}}, we_word[0:3], we_word[4:7],
                       we_word[8:11], we_word[12:15], offset[0], offset[1] );
   decoder4_16 wbyte ( offset, , wbt );
   assign we_byte = { wbt[3], wbt[2], wbt[1], wbt[0],
                      wbt[7], wbt[6], wbt[5], wbt[4],
                      wbt[11], wbt[10], wbt[9], wbt[8],
                      wbt[15], wbt[14], wbt[13], wbt[12] };
   and2$ a0_writemask ( mask_sel_0, we_inv, write_size[0] );
   or2$ o0_writemask ( mask_sel_1, we, write_size[1] );
   mux4_16$ write_mask ( we_temp, we_qword, 16'hffff, we_word, we_byte,
                         mask_sel_1, mask_sel_0 );

   and3$ a0 ( write_size_1, write_size[1], write_size[0], we_inv );


   // Write in logic - put the word or byte in the right place
   demux4_32 demux_word ( in_word[0:31], ram_in_0[0:31], ram_in_0[32:63],
                          ram_in_0[64:95], ram_in_0[96:127], offset[0], offset[1] );

   demux16_8 demux_byte ( in_word[24:31],
                          ram_in_1[24:31], ram_in_1[16:23], ram_in_1[8:15], ram_in_1
[0:7],
                          ram_in_1[56:63], ram_in_1[48:55], ram_in_1[40:47], ram_in_
1[32:39],
                          ram_in_1[88:95], ram_in_1[80:87], ram_in_1[72:79], ram_in_
1[64:71],
                          ram_in_1[120:127], ram_in_1[112:119], ram_in_1[104:111], r
am_in_1[96:103],
                          offset[0], offset[1], offset[2], offset[3] );

   mux3_128 ultra_mux ( ram_in, in, ram_in_0, ram_in_1,
                        write_size_1, write_size[0] );


   // Stabilize write enables for the RAM's
   // @posedge, clk 1 --> force to 1 -> 1 on clr, 0 on pre
   // @negedge, clk 0 --> latch d -> 1 on clr, 1 on pre
   inv1$ i0_stabilize ( clk_inv, clk );
   inv1$ i1_stabilize ( clk_inv2, clk_inv );
   buffer$ b0_stabilize ( clk_buf1, clk_inv2 );
   buffer$ b1_stabilize ( clk_buf2, clk_buf1 );
   buffer$ b2_stabilize ( clk_buf3, clk_buf2 );
```

```
buffer$ b3_stabilize ( clk_buf4, clk_buf3 );
buffer$ b4_stabilize ( clk_buf5, clk_buf4 );
buffer$ b5_stabilize ( clk_buf_final, clk_buf5 );
or2$ o0_stabilize ( clk_pre, clk_inv, clk_buf2 );

xor2$ x0_stabilize ( we_mask, clk, clk_buf_final );
inv1$ i2_stabilize ( we_mask_inv, we_mask );

dff16$ dff0_stabilize ( clk_inv, we_temp, we_stable_temp, , 1’b1, clk_pre );

or3$ o1_stabilize ( we_stable[0], we_stable_temp[0], we_mask_inv, clk );
or3$ o2_stabilize ( we_stable[1], we_stable_temp[1], we_mask_inv, clk );
or3$ o3_stabilize ( we_stable[2], we_stable_temp[2], we_mask_inv, clk );
or3$ o4_stabilize ( we_stable[3], we_stable_temp[3], we_mask_inv, clk );
or3$ o5_stabilize ( we_stable[4], we_stable_temp[4], we_mask_inv, clk );
or3$ o6_stabilize ( we_stable[5], we_stable_temp[5], we_mask_inv, clk );
or3$ o7_stabilize ( we_stable[6], we_stable_temp[6], we_mask_inv, clk );
or3$ o8_stabilize ( we_stable[7], we_stable_temp[7], we_mask_inv, clk );
or3$ o9_stabilize ( we_stable[8], we_stable_temp[8], we_mask_inv, clk );
or3$ o10_stabilize ( we_stable[9], we_stable_temp[9], we_mask_inv, clk );
or3$ o11_stabilize ( we_stable[10], we_stable_temp[10], we_mask_inv, clk );
or3$ o12_stabilize ( we_stable[11], we_stable_temp[11], we_mask_inv, clk );
or3$ o13_stabilize ( we_stable[12], we_stable_temp[12], we_mask_inv, clk );
or3$ o14_stabilize ( we_stable[13], we_stable_temp[13], we_mask_inv, clk );
or3$ o15_stabilize ( we_stable[14], we_stable_temp[14], we_mask_inv, clk );
or3$ o16_stabilize ( we_stable[15], we_stable_temp[15], we_mask_inv, clk );


// Rams
ram8b8w$ r0 ( index, ram_in[0:7], 1’b0, we_stable[0], mux_in[0:7] );
ram8b8w$ r1 ( index, ram_in[8:15], 1’b0, we_stable[1], mux_in[8:15] );
ram8b8w$ r2 ( index, ram_in[16:23], 1’b0, we_stable[2], mux_in[16:23] );
ram8b8w$ r3 ( index, ram_in[24:31], 1’b0, we_stable[3], mux_in[24:31] );

ram8b8w$ r4 ( index, ram_in[32:39], 1’b0, we_stable[4], mux_in[32:39] );
ram8b8w$ r5 ( index, ram_in[40:47], 1’b0, we_stable[5], mux_in[40:47] );
ram8b8w$ r6 ( index, ram_in[48:55], 1’b0, we_stable[6], mux_in[48:55] );
ram8b8w$ r7 ( index, ram_in[56:63], 1’b0, we_stable[7], mux_in[56:63] );

ram8b8w$ r8 ( index, ram_in[64:71], 1’b0, we_stable[8], mux_in[64:71] );
ram8b8w$ r9 ( index, ram_in[72:79], 1’b0, we_stable[9], mux_in[72:79] );
ram8b8w$ r10 ( index, ram_in[80:87], 1’b0, we_stable[10], mux_in[80:87] );
ram8b8w$ r11 ( index, ram_in[88:95], 1’b0, we_stable[11], mux_in[88:95] );

ram8b8w$ r12 ( index, ram_in[96:103], 1’b0, we_stable[12], mux_in[96:103] );
ram8b8w$ r13 ( index, ram_in[104:111], 1’b0, we_stable[13], mux_in[104:111] );
ram8b8w$ r14 ( index, ram_in[112:119], 1’b0, we_stable[14], mux_in[112:119] );
ram8b8w$ r15 ( index, ram_in[120:127], 1’b0, we_stable[15], mux_in[120:127] );

// Read out logic - byte or word
adder16$ add1 ( {12’b0, offset}, 16’h1, 1’b0, , off1 );
adder16$ add2 ( {12’b0, offset}, 16’h2, 1’b0, , off2 );
adder16$ add3 ( {12’b0, offset}, 16’h3, 1’b0, , off3 );

mux16_8 m1 ( out[0:7], mux_in[0:7], mux_in[8:15], mux_in[16:23],
             mux_in[24:31], mux_in[32:39], mux_in[40:47], mux_in[48:55],
             mux_in[56:63], mux_in[64:71], mux_in[72:79], mux_in[80:87],
             mux_in[88:95], mux_in[96:103], mux_in[104:111],
             mux_in[112:119], mux_in[120:127],
             offset[0], offset[1], offset[2], offset[3] );

mux16_8 m2 ( out[8:15], mux_in[0:7], mux_in[8:15], mux_in[16:23],
             mux_in[24:31], mux_in[32:39], mux_in[40:47], mux_in[48:55],
             mux_in[56:63], mux_in[64:71], mux_in[72:79], mux_in[80:87],
             mux_in[88:95], mux_in[96:103], mux_in[104:111],
             mux_in[112:119], mux_in[120:127],
             off1[12], off1[13], off1[14], off1[15] );

   mux16_8 m3 ( out[16:23], mux_in[0:7], mux_in[8:15], mux_in[16:23],
                mux_in[24:31], mux_in[32:39], mux_in[40:47], mux_in[48:55],
                mux_in[56:63], mux_in[64:71], mux_in[72:79], mux_in[80:87],
                mux_in[88:95], mux_in[96:103], mux_in[104:111],
                mux_in[112:119], mux_in[120:127],
                off2[12], off2[13], off2[14], off2[15] );

   mux16_8 m4 ( out[24:31], mux_in[0:7], mux_in[8:15], mux_in[16:23],
                mux_in[24:31], mux_in[32:39], mux_in[40:47], mux_in[48:55],
                mux_in[56:63], mux_in[64:71], mux_in[72:79], mux_in[80:87],
                mux_in[88:95], mux_in[96:103], mux_in[104:111],
                mux_in[112:119], mux_in[120:127],
                off3[12], off3[13], off3[14], off3[15] );

   assign out[32:127] = mux_in[32:127];

endmodule // datablock


module logicblock( we, size, set, comp0, comp1, hit, lru,
                   dirty0, dirty1, dirty, valid, reset, clk,
                   readhit, readmiss, writehit, writemiss,
                   tag0_we,
                   tag0_valid_in, tag0_valid_we,
                   tag0_dirty_in, tag0_dirty_we,
                   tag1_we,
                   tag1_valid_in, tag1_valid_we,
                   tag1_dirty_in, tag1_dirty_we,
                   data0_we, data1_we,
                   lru_in, lru_we,
                   mem_we, mem_we_addr,
                   write_size, cache_active );
   input        we, size, comp0, comp1, hit, lru,
                dirty0, dirty1, valid, cache_active, reset, clk;
   output       set, readhit, readmiss, writehit, writemiss;
   output       tag0_we,
                tag0_valid_in, tag0_valid_we,
                tag0_dirty_in, tag0_dirty_we,
                tag1_we,
                tag1_valid_in, tag1_valid_we,
                tag1_dirty_in, tag1_dirty_we,
                data0_we, data1_we,
                lru_in, lru_we,
                mem_we, mem_we_addr, dirty;
   output [0:1] write_size;

   wire [0:31]  Rom_Out;


   // Inverted signals
   // NOTE: some may not be needed, minimize later
   inv1$ i0_we ( we_inv, we );
   inv1$ i1_comp0 ( comp0_inv, comp0 );
   inv1$ i2_comp1 ( comp1_inv, comp1 );
   inv1$ i3_hit ( hit_inv, hit );
   inv1$ i4_lru ( lru_inv, lru );
   inv1$ i5_dirty0 ( dirty0_inv, dirty0 );
   inv1$ i6_dirty1 ( dirty1_inv, dirty1 );
   inv1$ i7_clkinv ( clk_inv, clk );
   inv1$ i8_reset0 ( reset_inv0, reset );
   inv1$ i9_reset1 ( reset_inv1, reset );
   inv1$ i9_reset2 ( reset_inv2, reset );
   inv1$ i10_reset3 ( reset_inv3, reset );
   inv1$ i11_valid ( valid_inv, valid );


   // Is the cache line dirty or clean?
   mux2$ dirty_or_clean ( dirty, dirty0, dirty1, set_in );
   and2$ a0_mem_we ( mem_we_addr, dirty, hit_inv );


   // Which set is being read from or written to
   mux2$ which_set ( set_in, lru, comp1, hit );
   dff$ d0_set ( clk_inv, set_in, set, , reset_inv0, 1'b1 );


   // Read/Write, Hit/Miss
   // For debugging purposes only...remove later
   and2$ a0 ( readhit, we, hit );           // Read hit
   and2$ a1 ( readmiss, we, hit_inv );      // Read miss
   and2$ a2 ( writehit, we_inv, hit );      // Write hit
   and2$ a3 ( writemiss, we_inv, hit_inv ); // Write miss


   // Determine write size
   // 00 - quadword, 10 - word, 11 - byte
   // Only a word or byte if a write hit, otherwise qword
   mux2$ m0_writesize ( write_size[0], 1'b0, 1'b1, writehit );
   mux2$ m1_writesize ( write_size[1], 1'b0, size, writehit );


   // Determine if the cache line is dirty and needs to be
   // written out when replaced
   and2$ a0_writeoutblock ( write_out, hit_inv, dirty );


   // Control logic ROM
   rom32b32w$ ROM_IF ( {we_a, set_a, hit_a, dirty_a, valid_a},
                       1'b1, Rom_Out );
   initial $readmemb("CACHE.CONTROL", ROM_IF.mem);


   // Rename output from ROM and handle reset signal
   assign tag0_we = Rom_Out[0];
   //assign tag0_valid_in = Rom_Out[1];
   //assign tag0_valid_we = Rom_Out[2];
   //assign tag0_dirty_in = Rom_Out[3];
   //assign tag0_dirty_we = Rom_Out[4];
   assign tag1_we = Rom_Out[5];
   //assign tag1_valid_in = Rom_Out[6];
   //assign tag1_valid_we = Rom_Out[7];
   //assign tag1_dirty_in = Rom_Out[8];
   //assign tag1_dirty_we = Rom_Out[9];
   assign data0_we = Rom_Out[10];
   assign data1_we = Rom_Out[11];
   //assign lru_in = Rom_Out[12];
   //assign lru_we = Rom_Out[13];
   //assign mem_we = Rom_Out[14];


   // On reset=0, reset the valid, lru and dirty bits
   mux2$ m0 ( tag0_valid_in, 1'b0, Rom_Out[1], reset_inv0 );
   mux2$ m1 ( tag0_valid_we, 1'b1, Rom_Out[2], reset_inv0 );
   mux2$ m2 ( tag0_dirty_in, 1'b0, Rom_Out[3], reset_inv0 );
   mux2$ m3 ( tag0_dirty_we, 1'b1, Rom_Out[4], reset_inv0 );

   mux2$ m4 ( tag1_valid_in, 1'b0, Rom_Out[6], reset_inv1 );
   mux2$ m5 ( tag1_valid_we, 1'b1, Rom_Out[7], reset_inv1 );
   mux2$ m6 ( tag1_dirty_in, 1'b0, Rom_Out[8], reset_inv1 );
   mux2$ m7 ( tag1_dirty_we, 1'b1, Rom_Out[9], reset_inv1 );

   mux2$ m8 ( lru_in, 1'b0, Rom_Out[12], reset_inv2 );
   mux2$ m9 ( lru_we, 1'b1, Rom_Out[13], reset_inv2 );

   mux2$ m10 ( mem_we0, 1'b0, Rom_Out[14], reset_inv2 );
   and2$ a1_mem_we ( mem_we_temp, mem_we0, mem_we_addr );
   nand2$ n1_mem_we ( we_clr0, valid_a, mem_we );
   and2$ a2_mem_we ( we_clr, we_clr0, reset_inv2 );

   dff$ dff0_stable_mem_we ( clk_inv, mem_we_temp, mem_we1, mem_we_inv, we_clr, 1'b1
 );
   dff$ dff1_stable_mem_we ( clk_inv, 1'b0, valid_xx, , reset_inv2, we_clr0 );

   buffer$ b0_mem_we ( mem_we2, mem_we1 );
   buffer$ b1_mem_we ( mem_we3, mem_we2 );
   buffer$ b2_mem_we ( mem_we4, mem_we3 );
   buffer$ b3_mem_we ( mem_we5, mem_we4 );
   buffer$ b4_mem_we ( mem_we6, mem_we5 );
   buffer$ b5_mem_we ( mem_we, mem_we6 );

   and2$ a0_active ( force_miss, reset_inv3, cache_active );

   mux2$ m11 ( we_a, 1'b1, we, force_miss );
   mux2$ m12 ( set_a, 1'b0, set_in, force_miss );
   mux2$ m13 ( hit_a, 1'b0, hit, force_miss );
   mux2$ m14 ( dirty_a, 1'b0, dirty, force_miss );
   mux2$ m15 ( valid_a, 1'b0, valid_yy, force_miss );
   or2$ orxy ( valid_yy, valid, valid_xx );

endmodule // logicblock
```

```
module EXTERNAL2 (addr0, addr1, data_in0, data_in1, data_out,
                  write_en, hit0, hit1, req0, req1,
                  valid0, valid1, grant0, grant1,
                  io_datain, io_dataout,
                  block_read, clk, reset, halt,
                  interrupt, write_size);

   input [0:31]   addr0, addr1, io_datain;
   input          write_en, halt, block_read, hit0, hit1, req0, req1;
   input [0:127]  data_in0, data_in1;
   output         valid0, valid1, grant0, grant1, clk, reset;
   output [0:127] data_out;
   output [0:31]  io_dataout;
   output [0:1]   interrupt;
   input [0:1]    write_size;

   wire [0:31]    addr;
   wire [0:127]   data_in;


   EXTERNAL External ( addr, data_in, data_out, we_mem, valid, block_read,
                       clk, reset, halt, interrupt, write_size, io_datain,
                       io_dataout );


   // Select the address and data words
   mux2_32 m0_addr ( addr, addr0, addr1, grant1 );
   mux2_128 m0_data ( data_in, data_in0, data_in1, grant1 );


   // Select the valid lines
   and3$ a0_valid ( valid0, valid, grant1_delay_inv, req0 );
   and3$ a1_valid ( valid1, valid, grant1_delay, req1 );


   // Select the memory write enable
   and2$ a0_we ( we_mem0, write_en, grant1 );
   buffer$ b0_we ( we_mem1, we_mem0 );
   buffer$ b1_we ( we_mem2, we_mem1 );
   buffer$ b2_we ( we_mem, we_mem2 );


   // Bus arbitration
   // Request, grant logic -- cache 1 has priority over cache 0
   // Cache 0: Instruction cache, requests bus on a cache miss
   // Cache 1: Data cache, requests bus on a cache miss, but only
   //          when it is a load or store
   inv1$ i0_reset ( reset_inv, reset );

   dffh d0_grant ( clk, d0_in, grant0, grant0_inv, reset_inv, 1'b1, 1'b1 );
   dffh d1_grant ( clk, d1_in, grant1, grant1_inv, reset_inv, 1'b1, 1'b1 );
   buffer$ b0_grant ( grant1_d0, grant1 );
   buffer$ b1_grant ( grant1_d1, grant1_d0 );
   buffer$ b2_grant ( grant1_d2, grant1_d1 );
   buffer$ b3_grant ( grant1_delay, grant1_d2 );
   inv1$ b4_grant ( grant1_delay_inv, grant1_d2 );

   inv1$ i0_req ( req0_inv, req0 );
   inv1$ i1_req ( req1_inv, req1 );
   and2$ a0_req ( x0, grant0, grant1_inv );
   or2$ o0_req ( t0, req1_inv, x0 );
   or3$ o1_req ( t1, req0_inv, grant0_inv, grant1 );
   and2$ a1_req ( d0_in, req0, t0 );
   and2$ a2_req ( d1_in, req1, t1 );

endmodule // EXTERNAL2


// Cache rolled up with a set of segment registers and a TLB
module cachetlbseg( addr_in, addr_out, data_in, data_mem_in,
                    data_out, data_mem_out, write_en, mem_we,
                    write_size, valid_in, valid_out,
                    io_datain, io_dataout, cache_active_in, msr17,
                    exception_protection_violation,
                    exception_page_fault,
                    exception_unaligned_fault,
                    flush, halt, grant, reset, clk );
   input [0:31]   addr_in, data_in, io_datain;
   input [0:127]  data_mem_in;
   input          write_en, write_size, cache_active_in, valid_in, msr17;
   input          flush, grant, reset, clk;
   output [0:31]  addr_out, data_out, io_dataout;
   output [0:127] data_mem_out;
   output         halt, mem_we, valid_out;
   output         exception_protection_violation,
                  exception_page_fault,
                  exception_unaligned_fault;

   wire [0:23]    segment_id;
   wire [0:39]    virtual_addr;
   wire [0:19]    phys_page_number;
   wire [0:31]    phys_addr;


   // Segment registers
   segmentregs4 SEGMENTREGS ( addr_in[0:3], segment_id,
                              exception_protection_violation,
                              msr17, reset, clk );

   // TLB
   assign virtual_addr = { segment_id, addr_in[4:19] };
   tlb5 TLB ( virtual_addr, phys_page_number, write_en,
              exception_page_fault, reset, clk );


   // Cache or I/O
   // Deactivate the cache if an I/O address or external signal
   // deactivates it
   nand2$ data_or_io ( cache_active0, phys_page_number[0], phys_page_number[1] );
   and2$ disable_cache ( cache_active, cache_active0, cache_active_in );


   assign phys_addr = { phys_page_number, addr_in[20:31] };
   cache256b2a4w CACHE ( phys_addr, addr_out, data_in, data_mem_in,
                         data_out, data_mem_out, write_en, mem_we,
                         write_size, valid_in, valid_out,
                         io_datain, io_dataout, cache_active,
                         flush, halt, grant, reset, clk );

   // Set alignment fault
   or2$  unaligned_addr_or ( unaligned_addr, phys_addr[30], phys_addr[31] );
   inv1$ write_size_i ( write_size_inv, write_size );
   and2$ alignment_a ( exception_unaligned_fault, write_size_inv, unaligned_addr );

endmodule // cachetlbseg


// 5-entry translation lookaside buffer
module tlb5( addr_in, addr_out, dirty_we, page_fault, reset, clk );
   input [0:39]  addr_in;
input         dirty_we, reset, clk;
output [0:19] addr_out;
output        page_fault;

wire [0:4]    dirty_temp;
wire [0:2]    dirty_sel;
wire [0:39]   tag0, tag1, tag2, tag3, tag4,
              tag0_out, tag1_out, tag2_out, tag3_out, tag4_out;
wire [0:19]   addr0, addr1, addr2, addr3, addr4,
              addr0_out, addr1_out, addr2_out, addr3_out, addr4_out;


// Invert reset signal
inv1$ i0_reset ( reset_inv0, reset );
inv1$ i1_reset ( reset_inv1, reset );
inv1$ i2_dirty ( dirty_we_inv, dirty_we );


// Valid bits
dffh valid0 ( clk, 1’b1, v0, , 1’b1, reset_inv0, 1’b0 );
dffh valid1 ( clk, 1’b1, v1, , 1’b1, reset_inv0, 1’b0 );
dffh valid2 ( clk, 1’b1, v2, , 1’b1, reset_inv0, 1’b0 );
dffh valid3 ( clk, 1’b1, v3, , 1’b1, reset_inv0, 1’b0 );
dffh valid4 ( clk, 1’b1, v4, , 1’b1, reset_inv0, 1’b0 );


// Dirty bits
pencoder8_3v$ encoder ( 1’b0, { 3’b000, c4, c3, c2, c1, c0 }, dirty_sel, );
demux8 d0_dirty ( 1’b1, dirty_temp[0], dirty_temp[1], dirty_temp[2],
                  dirty_temp[3], dirty_temp[4], , , ,
                  dirty_sel[0], dirty_sel[1], dirty_sel[2] );
dffh dirty0 ( clk, dirty_temp[0], d0, , reset_inv1, 1’b1, dirty_we_inv );
dffh dirty1 ( clk, dirty_temp[1], d1, , reset_inv1, 1’b1, dirty_we_inv );
dffh dirty2 ( clk, dirty_temp[2], d2, , reset_inv1, 1’b1, dirty_we_inv );
dffh dirty3 ( clk, dirty_temp[3], d3, , reset_inv1, 1’b1, dirty_we_inv );
dffh dirty4 ( clk, dirty_temp[4], d4, , reset_inv1, 1’b1, dirty_we_inv );


/*
 * TLB:tag         data
 * 000...00000     000...011  // page 0(virtual) -> page 3(physical)
 * 000...00001     000...000  // page 1(virtual) -> page 0(physical)
 * 000...00110     000...010  // page 6(virtual) -> page 2(physical)
 * 000...0ffff     000...001  // For interrupts and exeptions
 * 000...00010     110...000  // For I/O
 */

// Tags
assign tag0 = 40’b0000000000000000000000000000000000000000;
assign tag1 = 40’b0000000000000000000000000000000000000001;
assign tag2 = 40’b0000000000000000000000000000000000000110;
assign tag3 = 40’b0000000000000000000000001111111111111111;
assign tag4 = 40’b0000000000000000000000000000000000000010;


// Addresses
assign addr0 = 20’b00000000000000000011;
assign addr1 = 20’b00000000000000000000;
assign addr2 = 20’b00000000000000000010;
assign addr3 = 20’b00000000000000000001;
assign addr4 = 20’b11000000000000000000;


// Tag registers
reg40e tagreg0 ( clk, tag0, tag0_out, , 1’b1, 1’b1, reset );
```

```
   reg40e tagreg1 ( clk, tag1, tag1_out, , 1’b1, 1’b1, reset );
   reg40e tagreg2 ( clk, tag2, tag2_out, , 1’b1, 1’b1, reset );
   reg40e tagreg3 ( clk, tag3, tag3_out, , 1’b1, 1’b1, reset );
   reg40e tagreg4 ( clk, tag4, tag4_out, , 1’b1, 1’b1, reset );


   // Address registers
   reg20e addrreg0 ( clk, addr0, addr0_out, , 1’b1, 1’b1, reset );
   reg20e addrreg1 ( clk, addr1, addr1_out, , 1’b1, 1’b1, reset );
   reg20e addrreg2 ( clk, addr2, addr2_out, , 1’b1, 1’b1, reset );
   reg20e addrreg3 ( clk, addr3, addr3_out, , 1’b1, 1’b1, reset );
   reg20e addrreg4 ( clk, addr4, addr4_out, , 1’b1, 1’b1, reset );


   // Comparators
   comp_40 compare0 ( c0, tag0_out, addr_in );
   comp_40 compare1 ( c1, tag1_out, addr_in );
   comp_40 compare2 ( c2, tag2_out, addr_in );
   comp_40 compare3 ( c3, tag3_out, addr_in );
   comp_40 compare4 ( c4, tag4_out, addr_in );
   nor4$ n0 ( temp0, c0, c1, c2, c3 );
   inv1$ i0 ( temp1, c4 );
   and2$ tlb_miss ( page_fault, temp0, temp1 );


   // Determine output
   mux5_20 out_mux ( addr_out, addr0_out, addr1_out,
                     addr2_out, addr3_out, addr4_out,
                     dirty_sel[0], dirty_sel[1], dirty_sel[2] );
endmodule // tlb5


// 4 segment registers
module segmentregs4( addr_in, addr_out, protection_violation, msr17, reset, clk );
   input [0:3]   addr_in;
   input         msr17, reset, clk;
   output [0:23] addr_out;
   output        protection_violation;

   wire [0:23]   sr0, sr1, sr2, sr3,
                 sr0_in, sr1_in, sr2_in, sr3_in;
   wire [0:7]    junk;


   // Invert reset signal
   inv1$ i0_reset ( reset_inv, reset );


   // Protection bits
   dffh protect0 ( clk, 1’b1, p0, , 1’b1, reset_inv, 1’b0 );
   dffh protect1 ( clk, 1’b1, p1, , 1’b1, reset_inv, 1’b0 );
   dffh protect2 ( clk, 1’b1, p2, , 1’b1, reset_inv, 1’b0 );
   dffh protect3 ( clk, 1’b1, p3, , 1’b1, reset_inv, 1’b0 );
   mux4$ protection_mux ( p_temp, p0, p1, p2, p3, addr_in[3], addr_in[2] );
   inv1$ protection_inv ( p_inv, p_temp );
   and2$ protection_viol ( protection_violation, p_inv, msr17 );


   // Segments
   assign sr0_in = 24’b000000000000000000000000;
   assign sr1_in = 24’b000000000000000000000000;
   assign sr2_in = 24’b000000000000000000000000;
   assign sr3_in = 24’b000000000000000000000000;
   // Segment registers
   reg24e segreg0 ( clk, sr0_in, sr0, , 1’b1, 1’b1, reset );
   reg24e segreg1 ( clk, sr1_in, sr1, , 1’b1, 1’b1, reset );
   reg24e segreg2 ( clk, sr2_in, sr2, , 1’b1, 1’b1, reset );
   reg24e segreg3 ( clk, sr3_in, sr3, , 1’b1, 1’b1, reset );


   // Determine output
   mux4_32 out_mux ( {junk, addr_out},
                     {8’h0, sr0}, {8’h0, sr1}, {8’h0, sr2}, {8’h0, sr3},
                     addr_in[2], addr_in[3] );

endmodule // segmentregs4
```

```
module Choose_PC_Mod ( int_s, BCR_s, RTI_s, resBranch_s, predBranch_s,
                       int_a, LR_a, SRR0_a, resBranch_a, predBranch_a,
                       PC_4_a, PC_Out );


   input int_s, BCR_s, RTI_s, resBranch_s, predBranch_s;
   input [0:31] int_a, LR_a, SRR0_a, resBranch_a, predBranch_a, PC_4_a;
   output [0:31] PC_Out;

        /*
         *      Priority:
         *              -       Give priority to signals generated furthest down the pip
e.
         *              -       Signals that are generated in same stage cannot occur si
mult.
         *              -       PC_4 is the lowest priority.
         *
         *              Therefore:
         *                      1) Int_a
         *                      2) resBranch_a,
         *                      3) LR_a, SRR0_a, predBranch_a
         *                      4) PC_4
         */

   //   Invert select wires
   inv1$        int_s_INV ( int_s_BAR, int_s );
   inv1$        resBranch_s_INV ( resBranch_s_BAR, resBranch_s );


   // Int_a_Chosen = Int_s
   inv1$        int_INV ( int_a_NotChosen, int_s );

   // resBranch_a_Chosen =  resBranch_s && ! Int_s;
   nand2$       resBranch_NAND ( resBranch_A_NotChosen, resBranch_s, int_s_BAR );

   // LR_a_Chosen = BCR_s && ! resBranch_s && ! Int_s
   nand3$       LR_NAND ( LR_a_NotChosen, BCR_s, resBranch_s_BAR, int_s_BAR );

   // SRR0_a_Chosen = RTI_s && ! resBranch_s && ! Int_s
   nand3$       SRR0_AND ( SRR0_a_NotChosen, RTI_s, resBranch_s_BAR, int_s_BAR );

   // predBranch_a_Chosen = predBranch_s && ! resBranch_s && ! Int_s
   nand3$       predBranch_AND ( predBranch_a_NotChosen, predBranch_s,
                                 resBranch_s_BAR, int_s_BAR );

   // PC_4_a_Chosen = ! ( int_s || resBranch_s || BCR_s || RTI_s || predBranch_s )
   or5          PC_4_NOR ( PC_4_a_NotChosen, int_s, resBranch_s,
                                   BCR_s, RTI_s, predBranch_s );


   // Tristates to drive PC_Out bus

   // Int_a
   tristate32L  t1 ( int_a_NotChosen, int_a, PC_Out );

   // resBranch_a
   tristate32L  t2 ( resBranch_A_NotChosen, resBranch_a, PC_Out );

   // LR_a
   tristate32L  t3 ( LR_a_NotChosen, LR_a, PC_Out );

   // SRR0_a
   tristate32L t4 ( SRR0_a_NotChosen, SRR0_a, PC_Out );

   // predBranch_a
   tristate32L t5 ( predBranch_a_NotChosen, predBranch_a, PC_Out );

   // PC_4_a
   tristate32L t6 ( PC_4_a_NotChosen, PC_4_a, PC_Out );

endmodule
```

```
/*
 * 2-Bit branch prediction unit
 *
 *    created by:       Matt Houser    3/28/97
 *    last modified by: Matt Houser    3/29/97
 *
 *    Log:
 *       3/28/97  Created Module
 *
 *
 *    Description:
 *
 *       This file contains the key modules that make up the
 *       2-bit branch prediction unit for the M3 processor.
 *
 */


module BranchPredictorMod ( _hold, clk, clr, pre, QA, UA, UTaken,
                         UTarget, inUE, OutTaken, OutTarget );

   input         _hold;
   input         clk;           // Clock
   input         clr, pre;    // Clear and pre
   input  [0:7]  QA;            // Look-up address (PC in IF stage)
   input  [0:7]  UA;            // Update address (PC in stage where we resolve branch)
   input         UTaken;        // Was branch taken? (0 = no, 1 = yes)
   input  [0:31] UTarget;       // Target address of updating instruction
   input         inUE;            // Is an instruction actually updating

   output        OutTaken;      // Prediction (0 = not taken, 1 = taken)
   output [0:31] OutTarget;     // Guess target addr in case this really is a branch

   wire   [0:1]   QCount;

//   Update enabled if instruction in EX stage is
//   a branch, and if pipeline is not stalled

   and2$  UE_and ( UE, inUE, _hold );


/**********************************************************

      Counting

 **********************************************************/

   // CountStore holds the counts for the addresses
   csMod          CS ( clk, QA, UA, UE, UTaken, QCount, clr, pre );
   assign OutTaken =  QCount[0];

/**********************************************************

      Targets

 **********************************************************/

   // TargetStore holds the branch targets
   tsMod          TS ( clk, QA, UA, UTarget, OutTarget, clr, pre, UE );

endmodule



module reg2 (CLK, D, Q, QBAR, CLR, PRE, WE);

   input  CLK, CLR, PRE, WE;
   input  [0:1] D;
   output [0:1] Q, QBAR;

   dffh  dffh1 (CLK, D[0], Q[0], QBAR[0], CLR, PRE, WE);
   dffh  dffh2 (CLK, D[1], Q[1], QBAR[1], CLR, PRE, WE);

endmodule // reg2


module csMod (clk, QA, UA, UE, Taken, QCount, clr, pre );

   input clk, UE, clr, pre, Taken;
   input [0:7] QA, UA;
   output [0:1] QCount;
   wire [0:1] UCount, Data;

   regfile2_32 store ( clk, clr, pre, QA[3:7], UA[3:7], UA[3:7], UE, Data, QCount, U
Count );
   updateMod  up    ( UCount, Taken, Data );

endmodule


module updateMod ( Count, Taken, Out );

   input [0:1] Count;
   input Taken;
   output [0:1] Out;


    inv1$ in1( CountH_BAR, Count[0] );
    inv1$ in2( CountL_BAR, Count[1] );
    inv1$ in3( Taken_BAR, Taken );

    and2$ a1 ( a1out, Count[0], Taken ),
          a2 ( a2out, Count[1], Taken );
    and3$ a3 ( a3out, Count[0], Count[1], Taken_BAR );
    or3$  o1 ( Out[0], a1out, a2out, a3out );

    and2$ a4 ( a4out, CountL_BAR, Taken ),
          a5 ( a5out, Count[0], Taken );
    and3$ a6 ( a6out, Count[0], CountL_BAR, Taken_BAR );
    or3$  o2 ( Out[1], a4out, a5out, a6out );

endmodule


module tsMod ( clk, QA, UA, UTarget, OutTarget, clr, pre, UE );

   input clk, clr, pre;
   input [0:31] UTarget;
   input [0:7] QA, UA;
   input UE;
   output [0:31] OutTarget;

   regfile32_32 r1 ( clk, clr, pre, QA[3:7], UA[3:7], UE, UTarget, OutTarget );
//   reg32e$ r1 ( clk, UTarget, OutTarget, , clr, pre, UE );


endmodule
```

```
/*      Trap and Interrupt Module
 *
 *      Written By:     Matt Houser
 *      Created: 4/9/97
 *
 *      The trap and interrupt handler receives trap signals from the
 *      datapath, and interrupt signals from the External module.
 *      It allows for precise traps by flushing the pipeline before
 *      servicing the trap.
 *
 *      The module doesn't actually reset the PC or SRR0 and SRR1 regs.
 *      Instead, it sends signals telling components in the datapath
 *      to do that.
 *
 *      LIMITATIONS:
 *              At this point, the address for the handler routines is
 *              the same. In reality, this wouldn't work. However,
 *              for the purposes of this class (and their test cases),
 *              it will suffice. I hope to have time to "do the right thing".
 *
 *              Also, I am assuming that there will be no interrupts
 *              while we are servicing a trap or interrupt.
 *              This was OK'ed in class on April 9, 1997 by Prof. Davidson
 *
 *
 *      INPUT:
 *
 *              clk, clr, pre   The usual
 *
 *              IF_Exc                  5 wires signifying exceptions thrown in IF.
 *                                                      0       TLB miss
 *                                                      1       Page fault
 *                                                      2       Protection violation
 *                                                      3       Address translation erro
r
 *                                                      4       Alignment exception
 *
 *              ID_Exc                  Wire signifying illegal instruction in ID
 *
 *              M_Exc                   5 wires signifying exceptions thrown in IF.
 *                                                      0       TLB miss
 *                                                      1       Page fault
 *                                                      2       Protection violation
 *                                                      3       Address translation erro
r
 *                                                      4       Alignment exception
 *
 *              IO_Int                  2 wires for the IO device interrupts
 *
 *
 *      OUTPUT:
 *              SaveState               Tells datapath to set SRR0 and SRR1
 *
 *              IF_Inval                Invalidate the IF instruction
 *
 *              ID_Inval                Invalidate the ID instruction
 *
 *              EX_Inval                Invalidate the EX instruction
 *
 *              M_Inval                 Invalidate the M instruction
 *
 *              InsertBubble    Inserts a bubble in pipe
 *
 *              SetPC                   Tells PC to load handler address
 *
 *              Handler_Addr    Address of exception/interrupt handler routine
 *
 *
 *
 *      What happens when an exception signal is raised?
 *
 *              1) Check CAUSED bits of later pipeline stages than the one
 *                 in which the signal was raised.
 *
 *                If they all are 0 (no exceptions waiting to be handled):
 *
 *                  a) Set CAUSED bit for that stage to 1.
 *                  b) Clear CAUSED bits to the left (earlier stages).
 *                      c) Set that instruction and all to the left invalid.
 *
 *      Interrupt requests are treated similarly. When an interrupt signal
 *      goes high, the following happens:
 *
 *      Check to see if we are waiting to service a trap. If not,
 *      treat just like a trap in IF stage.
 *
 */

module trap_interrupt_M  ( clk, clr, pre, IF_PC, ID_PC, M_PC, IF_Exc, ID_Exc, M_Exc,
 IO_Int,
                           SaveState, IF_Inval, ID_Inval, EX_Inval, M_Inval,
                           InsertBubble, SetPC, savedPC, Handler_Addr );

        input   clk, clr, pre;
        input   [0:31] IF_PC, ID_PC, M_PC;
        input   [0:4]           IF_Exc, M_Exc;
        input                           ID_Exc;
        input   [0:1]           IO_Int;
        output                          SaveState, IF_Inval, ID_Inval, EX_Inval, M_I
nval;
        output                          InsertBubble, SetPC;
        output  [0:31]          Handler_Addr, savedPC;

        wire ID_Sig;
        wire M_Throw, M_Throw_Bar, IF_Throw, ID_Throw;
        wire muxs0, muxs1;
        wire [0:31] PC_mux_out;

        //      Determine which stages of the pipeline have set exception
        //      signals during this clock cycle.

        or5     IF_Sig_Or ( IF_Sig, IF_Exc[0], IF_Exc[1], IF_Exc[2],
                                IF_Exc[3], IF_Exc[4] ) ;

        nor5    IF_Sig_NOR ( IF_Sig_Bar, IF_Exc[0], IF_Exc[1], IF_Exc[2],
                                IF_Exc[3], IF_Exc[4] ) ;

        assign  ID_Sig = ID_Exc;

        inv1$   ID_Sig_INV      ( ID_Sig_Bar, ID_Sig );

        or5             M_SIG_OR        ( M_Sig, M_Exc[0], M_Exc[1], M_Exc[2],
                                                M_Exc[3], M_Exc[4] );

        nor5    M_SIG_NOR       ( M_Sig_Bar, M_Exc[0], M_Exc[1], M_Exc[2],
                                                M_Exc[3], M_Exc[4] );

        or2$    IO_SIG_OR       ( IO_Sig, IO_Int[0], IO_Int[1] );
```

```
//      Decide which stage actually gets thrown

//      M_Throw = M_Sig
assign M_Throw = M_Sig;
assign M_Throw_Bar = M_Sig_Bar;

//      ID_Throw = ID_Sig && !M_Sig
and2$   ID_Throw_AND    ( ID_Throw, ID_Sig, M_Sig_Bar );

//      IF_Throw = IF_Sig && !ID_Sig && !M_Sig
and3$   IF_Throw_AND    ( IF_Throw, IF_Sig, ID_Sig_Bar, M_Sig_Bar );

//      IO_Throw = IO_Sig && !IF_Sig && !ID_Sig && !M_Sig
and4$   IO_Throw_AND    ( IO_Throw, IO_Sig, IF_Sig_Bar, ID_Sig_Bar, M_Sig_Bar );


//      Pipe of 1-bit registers to tell in which stage of the datapath
//      the exception causing interrupt is located.
//
//      Will set the stage AFTER the signal is received from because the
//      register won't get written to until the positive edge of the next
//      cycle.


//      Determine new values for CAUSED registers

        //      ID_Caused_Set = IF_Throw || IO_Throw
or2$    ID_Caused_Set_Or ( ID_Caused_Set, IF_Throw, IO_Throw );

        //      EX_Caused_Set = ( ID_Caused || ID_Throw ) && ! M_Throw
or2$    EX_Caused_Set_Or  ( EX_Caused_Set_Or_Out, ID_Caused, ID_Throw ) ;
and2$   EX_Caused_Set_And ( EX_Caused_Set, M_Throw_Bar, EX_Caused_Set_Or_Out );

        //      M_Caused_Set = EX_Caused && !M_Throw
and2$   M_Caused_Set_And  ( M_Caused_Set, EX_Caused, M_Throw_Bar );


//      Declare the actual registers
dffh    ID_CAUSED_Reg   ( clk, ID_Caused_Set, ID_Caused, ID_Caused_Bar,
                                          clr, pre, 1'b1 ),

                EX_CAUSED_Reg   ( clk, EX_Caused_Set, EX_Caused, EX_Caused_Bar,
                                          clr, pre, 1'b1 ),

                M_CAUSED_Reg    ( clk, M_Caused_Set, M_Caused, M_Caused_Bar,
                                          clr, pre, 1'b1 );


//      Set SaveState output wire:      SaveState = M_Caused || M_Throw;
or2$    SaveState_Or    ( SaveState, M_Caused, M_Throw );

//      SetPC is the same
assign SetPC = SaveState;


//      When we receive an signal for an interrupt or trap, set all instructions
//      in that stage and before to NOOP's

//      IF_Inval = IF_Throw || ID_Throw || M_Throw || IO_Throw
or4$    IF_Inval_Or     ( IF_Inval, IF_Throw, ID_Throw, M_Throw, IO_Throw ) ;

//      ID_Inval = ID_Throw || M_Throw
or2$    ID_Inval_Or ( ID_Inval, ID_Throw, M_Throw );

//      EX_Inval = M_Throw
        assign EX_Inval = M_Throw;

        //      M_Inval = M_Throw
        assign M_Inval = M_Throw;

        //      Save PC that will set SRR0
        assign muxs1 = M_Throw;
        assign muxs0 = ID_Throw;
        mux3_32 pc_mux ( PC_mux_out, IF_PC, ID_PC, M_PC, muxs1, muxs0 );
        or3$    any_throw_or ( any_throw, IF_Throw, ID_Throw, M_Throw );
        reg32e$ savePCreg ( clk, PC_mux_out, savedPC, , clr, 1'b1, any_throw );

        //      When to insert a bubble
        //              A bubble is inserted when there is an instruction in the pip
e that caused
        //              an exception or interrupt, or when an exception in IF or ID
was just thrown
        or5             Bubble_Or       ( InsertBubble, ID_Caused, EX_Caused, IF_Thr
ow, ID_Throw, IO_Throw );


        //      Set Handler address -- for now, it's always 0FFFF000
        assign Handler_Addr = 32'h0FFFF000;

endmodule
```

```
module MaskGenerator(N, MASK, FLIPPED);
   input [0:5]   N;
   output [0:31] MASK;
   output [0:31] FLIPPED;

   rom32b32w$ ROM(N[1:5], 1’b1, MASK);

   assign FLIPPED = {MASK[31], MASK[30], MASK[29], MASK[28], MASK[27], MASK[26], MASK[25
], MASK[24],
                     MASK[23], MASK[22], MASK[21], MASK[20], MASK[19], MASK[18], MASK[17
], MASK[16],
                     MASK[15], MASK[14], MASK[13], MASK[12], MASK[11], MASK[10], MASK[9]
, MASK[8],
                     MASK[7], MASK[6], MASK[5], MASK[4], MASK[3], MASK[2], MASK[1], MASK
[0]};
endmodule // MaskGenerator

// Decodes the 6 bit opcodes and outputs the control
// If bit 0 is set then it needs to use the extended opcode ROM file
module OpcodeDecoder(OPCODE, CTRL);
   input [0:5]   OPCODE;
   output [0:63] CTRL;

   //l
   comp_6 c1(l, OPCODE, 6’b100000);
   //lu
   comp_6 c2(lu, OPCODE, 6’b100001);
   //lbzu
   comp_6 c3(lbzu, OPCODE, 6’b100011);
   //st
   comp_6 c4(st, OPCODE, 6’b100100);
   //stu
   comp_6 c5(stu, OPCODE, 6’b100101);
   //stbu
   comp_6 c6(stbu, OPCODE, 6’b100111);
   //ai
   comp_6 c7(ai, OPCODE, 6’b001100);
   //ai.
   comp_6 c8(airec, OPCODE, 6’b001101);
   //bc/bca/bcl/bcla
   comp_6 c9(bc, OPCODE, 6’b010000);
   //bcr
   comp_6 c10(bcr, OPCODE, 6’b010011);
   //type 31
   comp_6 c11(type31, OPCODE, 6’b011111);
   // halt
   comp_6 c12(halt, OPCODE, 6’b111111);

   assign   CTRL[0] = type31;
   or3$ o1(CTRL[1], l, lu, lbzu);
   assign CTRL[2] = bcr;
   assign CTRL[3:4] = 2’d0;
   nor2$ o2(CTRL[5], bc, bcr);
   assign CTRL[6] = CTRL[5];
   assign CTRL[7:8] = 2’d0;
   assign CTRL[9] = CTRL[5];
   assign CTRL[10] = 1’d0;
   assign CTRL[11] = CTRL[5];
   assign CTRL[12] = 1’d0;
   or2$ o3(CTRL[13], ai, airec);
   assign CTRL[14:15] = 2’d0;
   assign CTRL[16] = CTRL[13];
   assign CTRL[17:21] = 5’d0;
   assign CTRL[22] = airec;
   assign CTRL[23:24] = 2’d0;
   or2$ o4(CTRL[25], bc, bcr);
   assign CTRL[26] = 1’d0;
   or3$ o5(CTRL[27], st, stu, stbu);
   or3$ o6(CTRL[28], CTRL[27], CTRL[13], CTRL[25]);
   assign CTRL[29] = 1’d0;
   assign CTRL[30] = lbzu;
   or2$ o7(CTRL[31], CTRL[1], CTRL[13]);
   or4$ o8(CTRL[32], lu, lbzu, stu, stbu);
   assign CTRL[33] = 1’d0;
   assign CTRL[34] = 1’d0;
   assign CTRL[35] = CTRL[27];
   assign CTRL[36] = bc;
   assign CTRL[37:62] = 26’d0;
   // valid bit
   or11 o10(CTRL[63], l, lu, lbzu, st, stu, stbu, ai, airec, bc, bcr, halt);

endmodule // OpcodeDecoder


module ASelector(MEMHAZ, EXHAZ, SelA, ASelect);
   input [0:7]  MEMHAZ, EXHAZ;
   input [0:1]  SelA;
   output [0:2] ASelect;

   wire [0:9]   value;
   assign   value = {EXHAZ[2], EXHAZ[3], EXHAZ[4], EXHAZ[5],
                     MEMHAZ[2], MEMHAZ[3], MEMHAZ[4], MEMHAZ[5], SelA};

   /* value  =memRTRSHazard, memRARSHazard, memRTRAHazard, memRARAHazard,
    wbRTRSHazard, wbRARSHazard, wbRTRAHazard, wbRARAHazard */

   comp_10 c1(temp1, value, 10’b0000000001);
   comp_10 c2(temp2, value, 10’b0000000010);
   comp_10 c3(temp3, value, 10’b1000000010);
   comp_10 c4(temp4, value, 10’b0100000010);
   comp_10 c5(temp5, value, 10’b0010000000);
   comp_10 c6(temp6, value, 10’b0001000000);
   comp_10 c7(temp7, value, 10’b0000100010);
   comp_10 c8(temp8, value, 10’b0000010010);
   comp_10 c9(temp9, value, 10’b0000001000);
   comp_10 c10(temp10, value, 10’b0000000100);

   or4$ o1(tempor1, temp4, temp6, temp7, temp8);
   or3$ o2(ASelect[0], tempor1, temp9, temp10);

   or4$ o3(tempor2, temp2, temp3, temp5, temp8);
   or2$ o4(ASelect[1], tempor2, temp10);

   or4$ o5(tempor3, temp1, temp3, temp5, temp7);
   or2$ o6(ASelect[2], tempor3, temp9);
endmodule // ASelector


module BSelector(MEMHAZ, EXHAZ, SelB, BSelect);
   input [0:7]  MEMHAZ, EXHAZ;
   input        SelB;
   output [0:2] BSelect;

   wire [0:4]   value;
   assign value = {SelB, EXHAZ[0], EXHAZ[1], MEMHAZ[0], MEMHAZ[1]};

   comp_5 c1(temp1, value , 5’b00000);
   comp_5 c2(temp2, value , 5’b10000);
   comp_5 c3(temp3, value , 5’b01000);
   comp_5 c4(temp4, value , 5’b00100);
   comp_5 c5(temp5, value , 5'b00010);
   comp_5 c6(temp6, value , 5'b00001);

   or2$ o1(BSelect[0], temp5, temp6);
   or2$ o2(BSelect[1], temp3, temp4);
   or3$ o3(BSelect[2], temp2, temp4, temp6);
endmodule // BSelector

// tested
// checked buffers
module XER_Register(clk, Reset, ovf, cout, RT, SetSO, SetOV, SetCA, LoadXER, XEROUT);
   input         clk, Reset, ovf, cout, LoadXER;
   input         SetSO, SetOV, SetCA;
   input [0:31]  RT;
   output [0:31] XEROUT;

   wire [0:31]  temp_in, temp;

   // SetSO - sets SO bit
   // SetOV - sets OV bit
   // SetCA - sets CA bit

   or2$ o1(temp_ovf, ovf, XEROUT[0]); // summarizes overflow

   mux4$ m1(temp_in[0], XEROUT[0], temp_ovf, RT[0], /* IN3 */, SetSO, LoadXER);
   mux4$ m2(temp_in[1], XEROUT[1], ovf, RT[1], /* IN3 */, SetOV, LoadXER);
   mux4$ m3(temp_in[2], XEROUT[2], cout, RT[2], /* IN3 */, SetCA, LoadXER);
   mux2_32 m4(temp, {3'b0, XEROUT[3:31]}, {3'b0, RT[3:31]}, LoadXER);
   assign temp_in[3:31] = temp[3:31];
   or4$ o2(SetXER, SetSO, SetOV, SetCA, LoadXER);
   reg32e$ xer(clk, temp_in, XEROUT, , Reset, 1'b1, SetXER);
endmodule // xer


// tested
// checked buffers
module LR_Register(clk, Reset, NPC, RT, LROut, temp_SetLR, LoadLR);
   input         clk, Reset, temp_SetLR, LoadLR;
   input [0:31]  NPC, RT;
   output [0:31] LROut;

   wire [0:31]   temp_in;

   // LoadLR - selects input from RT
   mux2_32 m1(temp_in, NPC, RT, LoadLR);
   // loads or sets LR
   or2$ o1(SetLR, temp_SetLR, LoadLR);
   // actual register
   reg32e$ LR(clk, temp_in, LROut, , Reset, 1'b1, SetLR);
endmodule // LR_Register


// checked buffers
module SPR_Logic(SPR, MTSPR, MFSPR, LoadLR, LoadCTR, LoadXER, FromLR, FromCTR, FromXER);
   input [0:4] SPR;
   input       MTSPR, MFSPR;
   output      LoadLR, LoadCTR, LoadXER, FromLR, FromXER, FromCTR;

   // XER
   comp_5 c1(temp1, SPR, 5'd1);
   // LR
   comp_5 c2(temp2, SPR, 5'd8);
   // CTR
   comp_5 c3(temp3, SPR, 5'd9);
```

```
   // actual control signals
   and2$ a1(LoadXER, MTSPR, temp1);
   and2$ a2(LoadLR, MTSPR, temp2);
   and2$ a3(LoadCTR, MTSPR, temp3);
      // actual control signals
   and2$ a4(FromXER, MFSPR, temp1);
   and2$ a5(FromLR, MFSPR, temp2);
   and2$ a6(FromCTR, MFSPR, temp3);
endmodule // SPR_Logic


// tested, works beautifully
module Branch_Logic(Branch, BO, DecCTR, ZeroCTR, OnFalse, OnTrue, UseCTR);
   output      DecCTR, ZeroCTR, OnFalse, OnTrue, UseCTR;
   input [0:4] BO;
   input       Branch;


   // dec, ctr!=0 && false
   comp_4 c1(temp1, BO[0:3], 4'd0);

   // dec, ctr=0 && false
   comp_4 c2(temp2, BO[0:3], 4'd1);

   // false
   inv1$ i1(not_BO0, BO[0]);
   inv1$ i2(not_BO1, BO[1]);
   and3$ a1(temp3, not_BO0, not_BO1, BO[2]);

   // dec, ctr!=0 && true
   comp_4 c3(temp4, BO[0:3], 4'd4);

   // dec, ctr=0 && true
   comp_4 c4(temp5, BO[0:3], 4'd5);

   // true
   and3$ a2(temp6, not_BO0, BO[1], BO[2]);

   // dec, ctr!=0
   inv1$ i3(not_BO2, BO[2]);
   inv1$ i4(not_BO3, BO[3]);
   and3$ a3(temp7, BO[0], not_BO2, not_BO3);

   // dec, ctr=0
   and3$ a4(temp8, BO[0], not_BO2, BO[3]);

   //always
   and2$ a5(temp9, BO[0], BO[2]);

   // **** actual control signals *****
   // decrement signal
   or3$ o1(temp_dec1, temp1, temp2, temp4);
   or3$ o2(temp_dec2, temp5, temp7, temp8);
   or2$ o3(temp_DecCTR, temp_dec1, temp_dec2);
   // this determines if the CTR is written so it must be a branch
   and2$ a6(DecCTR, temp_DecCTR, Branch);

   // ctr=0 signal
   or3$ o4(ZeroCTR, temp2, temp5, temp8);

   // UseCTR signal
   nor3$ n1(UseCTR, temp3, temp6, temp9);

   // on false signal
   or4$ o5(OnFalse, temp1, temp2, temp3, temp9);
   // on true signal
   or4$ o8(OnTrue, temp4, temp5, temp6, temp9);

endmodule // Branch_Logic


// tested, works perfect
// checked buffers
module CTR_Register(clk, Set, BO, RT, Out, temp_SetCTR, SelCTR);
   input         clk, Set, temp_SetCTR;
   input [0:1]   SelCTR;
   input [0:4]   BO;
   input [0:31]  RT;
   output [0:31] Out;

   wire [0:31]   temp_out, temp_in, dec_out;

   // BO field is the options from opcode
   // SelCTR function
   // 00 nothing
   // 01 decrement
   // 10 load from RT

   // decrements number
   adder32 add(Out, 32'hFFFFFFFE, 1'b1, , dec_out);
   or2$ o1(SetCTR, temp_SetCTR, SelCTR[0]);
   inv1$ i1(not_SelCTR0, SelCTR[0]);
   and2$ a1(real_SelCTR1, not_SelCTR0, SelCTR[1]);

   // selects CTR source
   mux4_32 m1(temp_in, Out, dec_out, RT, /* IN3 */, SelCTR[0], real_SelCTR1);

   // actual register
   reg32e$ CTR(clk, temp_in, Out, , Set, 1'b1, SetCTR);

endmodule // CTR_Register


// tested, works cool
module CR_Register(clk, Set, highRA, highRB, ALUOUT, CA, SO, SetCR, SelCR, CRfield, Unsi
gned, CR);
   input [0:31]  ALUOUT;
   input         clk, Set, SetCR, SelCR, CA, SO, Unsigned, highRA, highRB;
   input [0:2]   CRfield;
   output [0:31] CR;

   wire [0:3]    crin0, crin1, crin2, crin3, crin4, crin5, crin6, crin7;
   wire [0:2]    CRSel;
   wire [0:3]    CRData;
   wire          LT;

   // SetCR - selects if CR is set
   // SelCR - selects if IR has CR field or not

   // generate 4 fields
   inv1$ i3(not_highRA, highRA);
   inv1$ i4 (not_highRB, highRB);
   inv1$ i5(not_Unsigned, Unsigned);

   // EQUAL
   nor32 n1(temp_EQ, ALUOUT); // result is 0 then they are the same
   and4$ a4(ok1, temp_EQ, highRA, highRB, Unsigned);
   and4$ a5(ok2, temp_EQ, not_highRA, not_highRB, Unsigned);
   and2$ a6(ok3, temp_EQ, not_Unsigned);
   or3$ o3(EQ, ok1, ok2, ok3);

   // LT
   and3$ a3(unsigned_ltcase, Unsigned, not_highRA, highRB);
   and2$ a7(signed_ltcase, not_Unsigned, ALUOUT[0]);
   or2$ o1(LT, unsigned_ltcase, signed_ltcase); // get negative

   // GT
   and3$ a2(unsigned_gtcase, Unsigned, highRA, not_highRB);
   nor2$ n2(temp_GT, LT, EQ); // not LT or EQ so must be GT!
   and2$ a8(signed_gtcase, not_Unsigned, temp_GT);
   or2$ o2(GT, unsigned_gtcase, signed_gtcase);


   assign CRData = {LT, GT, EQ, SO};

//   inv1$ i2(not_EQ, EQ);
//   and2$ a1(GT2, CA, not_EQ); // not gt or equal so must be
// what were these for???
//   inv1$ i1(not_CA, CA);
//   mux2_4 m2(CRData, {LT, GT, EQ, SO}, {not_CA, GT2, EQ, SO}, Unsigned);


   mux2_3 m1(CRSel, 3'b0, CRfield, SelCR);

   // select field to write
   demux8 d1(SetCR, cr0, cr1, cr2, cr3, cr4, cr5, cr6, cr7, CRSel[0], CRSel[1], CRSe
l[2]);

   // save every other old field
   mux2_4 f0(crin0, CR[0:3], CRData, cr0);
   mux2_4 f1(crin1, CR[4:7], CRData, cr1);
   mux2_4 f2(crin2, CR[8:11], CRData, cr2);
   mux2_4 f3(crin3, CR[12:15], CRData, cr3);
   mux2_4 f4(crin4, CR[16:19], CRData, cr4);
   mux2_4 f5(crin5, CR[20:23], CRData, cr5);
   mux2_4 f6(crin6, CR[24:27], CRData, cr6);
   mux2_4 f7(crin7, CR[28:31], CRData, cr7);

   // actual register
   reg32e$ crreg(clk, {crin0, crin1, crin2, crin3, crin4, crin5, crin6, crin7},
                 CR, ,Set, 1'b1, SetCR);
endmodule // Condition_Register


// checked buffers
module SuperALU(IR, temp_ALU_RA, ALU_RB, ALU_RT, ALUOp,
                msel, cin, ALUSel, ShiftRight, Out, cout, ovf);
   input [0:31]  IR, temp_ALU_RA, ALU_RB, ALU_RT;
   input         msel, cin, ShiftRight, ALUSel;
   input [0:3]   ALUOp;
   output        cout, ovf;
   output [0:31] Out;

   wire [0:31]   ShiftOut, ALUOut, temp_ShiftOut, ALU_RA, MASK, SLShiftOut, SRShiftO
ut,
                 SRAShiftOut, RealShiftOut, FLIPPED_MASK, not_FLIPPED_MASK, not_MASK
,
                 SRAMask, temp_ShiftOut2, realMASK, inv_SRAMask, CAOut;

   comp_6 c1(temp1, IR[0:5], 6'b011111);
   comp_9 c2(temp2, IR[22:30], 9'd8);
   and2$ a1(subtract, temp1, temp2);
   inv32en inverter(ALU_RA, temp_ALU_RA, subtract);
   alu32 alu(ALU_RA, ALU_RB, cin, msel, ALUOp, temp_cout, ALUOut, ovf);
   barrel_shift b1(ALU_RT, ALU_RB[27:31], ShiftRight, temp_ShiftOut);
   MaskGenerator maskgen(ALU_RB[26:31], MASK, FLIPPED_MASK);

   // is it an sra?
   comp_10 c4(temp3, IR[21:30], 10’b1100011000);
   and2$ a2(sra, temp1, temp3);

   // masked output for sl
   inv32 inv1(not_FLIPPED_MASK, FLIPPED_MASK);
   and32 a3(SLShiftOut, temp_ShiftOut, not_FLIPPED_MASK);
   // masked output for sr
   inv32 inv(not_MASK, MASK);
   and32 a4(SRShiftOut, temp_ShiftOut, not_MASK);
   // masked output for sra
   and32 a5(realMASK, MASK, {32{ALU_RT[0]}});
   or32 o3(SRAShiftOut, SRShiftOut, realMASK);

   // determine if it was sl or sr or sra (this has the mask)
   mux4_32 m0(temp_ShiftOut2, SLShiftOut, SRShiftOut, /* not used */ ,
              SRAShiftOut, sra, ShiftRight);

   // decides if the output was shifted >32 (uses a mask of 0’s)
   mux2_32 m1(ShiftOut, temp_ShiftOut2, 32’b0, ALU_RB[26]);

   // it’s an sra and >32 (uses a mask of sign bits)
   and2$ a6(sra_and_rb, sra, ALU_RB[26]);
   // makes a mask of 32 sign bits
   mux2_32 m2(SRAMask, 32’d0, 32’hFFFFFFFF, ALU_RT[0]);

   // picks the sra masked value or the other one
   mux2_32 m3(RealShiftOut, ShiftOut, SRAMask, sra_and_rb);

   // generates CA for sra
   inv32 inv_mask(inv_SRAMask, MASK); // inverted mask
   and32 ca_gen_and32(CAOut, RealShiftOut, inv_SRAMask); // ANDed with result
   or32_notbitwise ca_gen(shift_CA, CAOut); // ORed together
   and2$ ca_and(temp_CA, shift_CA, ALU_RT[0]); // ANDed with bit 0 of RS
   // selects this for shifts (written only for sra)
   mux2$ m5(cout, temp_cout, temp_CA, ALUSel);

   // uses shifter output or alu output for the fantastic SuperALU effect
   mux2_32 m4(Out, ALUOut, RealShiftOut, ALUSel);

endmodule // superALU


// checked buffers
module gprregfile(CLK, RE_1, RE_2, RE_3, WR_1, WR_2, OUT1, OUT2, OUT3, WRDATA1,
               WRDATA2, WRE1, WRE2, Reset);
   input CLK, Reset;
   input         WRE1, WRE2;
   input [0:4]   RE_1, RE_2, RE_3;
   wire [0:4]    RE11, RE21, RE31;
   wire [0:4]    RE12, RE22, RE32;
   input [0:4]   WR_1, WR_2;
   wire [0:4]    WR11, WR21;
   wire [0:4]    WR12, WR22;
   output [0:31] OUT1, OUT2, OUT3;
   input [0:31]  WRDATA1, WRDATA2;

   wire [0:31]   OUT1_0, OUT1_1, OUT1_2, OUT1_3, OUT1_4, OUT1_5, OUT1_6, OUT1_7;
   wire [0:31]   OUT2_0, OUT2_1, OUT2_2, OUT2_3, OUT2_4, OUT2_5, OUT2_6, OUT2_7;
   wire [0:31]   OUT3_0, OUT3_1, OUT3_2, OUT3_3, OUT3_4, OUT3_5, OUT3_6, OUT3_7;

   // selects the correct write registers
   demux8 d0(WRE1, en10, en11, en12, en13, en14, en15, en16, en17,
             WR_1[0], WR_1[1], WR_1[2]);

   // disables WRE2 if WR1==WR2
   //   xor2$ x1(sig1, WR1[0], WR2[0]);
   //   xor2$ x2(sig2, WR1[1], WR2[1]);
   //   xor2$ x3(sig3, WR1[2], WR2[2]);
   //   xor2$ x4(sig4, WR1[3], WR2[3]);
   //   xor2$ x5(sig5, WR1[4], WR2[4]);
   //   or2$ o1(sig6, sig1, sig2);
   //   or3$ o2(sig7, sig3, sig4, sig5);
   //   or2$ o3(different, sig6, sig7);
   //   and2$ a1(real_WRE2, different, WRE2);

   buffer$ b0(RE11[3], RE_1[3]);
   buffer$ b1(RE11[4], RE_1[4]);
   buffer$ b2(RE12[3], RE_1[3]);
   buffer$ b3(RE12[4], RE_1[4]);

   buffer$ b4(RE21[3], RE_2[3]);
   buffer$ b5(RE21[4], RE_2[4]);
   buffer$ b6(RE22[3], RE_2[3]);
   buffer$ b7(RE22[4], RE_2[4]);

   buffer$ b8(RE31[3], RE_3[3]);
   buffer$ b9(RE31[4], RE_3[4]);
   buffer$ b10(RE32[3], RE_3[3]);
   buffer$ b11(RE32[4], RE_3[4]);

   buffer$ b12(WR11[3], WR_1[3]);
   buffer$ b13(WR11[4], WR_1[4]);
   buffer$ b14(WR12[3], WR_1[3]);
   buffer$ b15(WR12[4], WR_1[4]);

   buffer$ b16(WR21[3], WR_2[3]);
   buffer$ b17(WR21[4], WR_2[4]);
   buffer$ b18(WR22[3], WR_2[3]);
   buffer$ b19(WR22[4], WR_2[4]);

   demux8 d1(WRE2, en20, en21, en22, en23, en24, en25, en26, en27,
             WR_2[0], WR_2[1], WR_2[2]);
   // make thirty two registers here
   regfile4_32 r0(CLK, RE11[3:4], RE21[3:4], RE31[3:4], WR11[3:4], WR21[3:4],
                  OUT1_0, OUT2_0, OUT3_0, WRDATA1, WRDATA2, en10, en20, Reset);
   regfile4_32 r1(CLK, RE11[3:4], RE21[3:4], RE31[3:4], WR11[3:4], WR21[3:4],
                  OUT1_1, OUT2_1, OUT3_1, WRDATA1, WRDATA2, en11, en21, Reset);
   regfile4_32 r2(CLK, RE11[3:4], RE21[3:4], RE31[3:4], WR11[3:4], WR21[3:4],
                  OUT1_2, OUT2_2, OUT3_2, WRDATA1, WRDATA2, en12, en22, Reset);
   regfile4_32 r3(CLK, RE11[3:4], RE21[3:4], RE31[3:4], WR11[3:4], WR21[3:4],
                  OUT1_3, OUT2_3, OUT3_3, WRDATA1, WRDATA2, en13, en23, Reset);
   regfile4_32 r4(CLK, RE12[3:4], RE22[3:4], RE32[3:4], WR12[3:4], WR22[3:4],
                  OUT1_4, OUT2_4, OUT3_4, WRDATA1, WRDATA2, en14, en24, Reset);
   regfile4_32 r5(CLK, RE12[3:4], RE22[3:4], RE32[3:4], WR12[3:4], WR22[3:4],
                  OUT1_5, OUT2_5, OUT3_5, WRDATA1, WRDATA2, en15, en25, Reset);
   regfile4_32 r6(CLK, RE12[3:4], RE22[3:4], RE32[3:4], WR12[3:4], WR22[3:4],
                  OUT1_6, OUT2_6, OUT3_6, WRDATA1, WRDATA2, en16, en26, Reset);
   regfile4_32 r7(CLK, RE12[3:4], RE22[3:4], RE32[3:4], WR12[3:4], WR22[3:4],
                  OUT1_7, OUT2_7, OUT3_7, WRDATA1, WRDATA2, en17, en27, Reset);

   // output 1 mux
   mux8_32 d2(OUT1, OUT1_0, OUT1_1, OUT1_2, OUT1_3, OUT1_4, OUT1_5,
                OUT1_6, OUT1_7, RE_1[0], RE_1[1], RE_1[2]);

   // output 2 mux
   mux8_32 d3(OUT2, OUT2_0, OUT2_1, OUT2_2, OUT2_3, OUT2_4, OUT2_5,
                OUT2_6, OUT2_7, RE_2[0], RE_2[1], RE_2[2]);

   // output 3 mux
   mux8_32 d4(OUT3, OUT3_0, OUT3_1, OUT3_2, OUT3_3, OUT3_4, OUT3_5,
                OUT3_6, OUT3_7, RE_3[0], RE_3[1], RE_3[2]);
endmodule // regfile


// tested, used in the real register file
// checked buffers
module regfile4_32(CLK, RE1, RE2, RE3, WR1, WR2, OUT1, OUT2, OUT3, WRDATA1,
                   WRDATA2, WRE1, WRE2, Reset);
   input         CLK, Reset;
   input [0:31]  WRDATA1, WRDATA2;
   input [0:1]   RE1, RE2, RE3;
   input [0:1]   WR1, WR2;
   input         WRE1, WRE2;
   output [0:31] OUT1, OUT2, OUT3;

   wire [0:31]   OT0, OT1, OT2, OT3;
   wire [0:31]   IN0, IN1, IN2, IN3;
   wire [0:31]   IN0_1, IN1_1, IN2_1, IN3_1;
   wire [0:31]   IN0_2, IN1_2, IN2_2, IN3_2;

   // demux for WRITE 1
   demux4_32 d1(WRDATA1, IN0_1, IN1_1, IN2_1, IN3_1, WR1[0], WR1[1]);

   // demux for WRITE 2
   demux4_32 d2(WRDATA2, IN0_2, IN1_2, IN2_2, IN3_2, WR2[0], WR2[1]);

   // does the write enables
   demux4 d3(WRE1, en10, en11, en12, en13, WR1[0], WR1[1]);
   demux4 d4(WRE2, en20, en21, en22, en23, WR2[0], WR2[1]);
   or2$ o1(en0, en10, en20);
   or2$ o2(en1, en11, en21);
   or2$ o3(en2, en12, en22);
   or2$ o4(en3, en13, en23);

   // prevents more than one write driving the write data bus
   inv1$ i1(not_WRE1, WRE1);
   inv1$ i2(not_WRE2, WRE2);
   tristate32L ts0(not_WRE1, IN0_1, IN0);
   tristate32L ts1(not_WRE2, IN0_2, IN0);
   tristate32L ts2(not_WRE1, IN1_1, IN1);
   tristate32L ts3(not_WRE2, IN1_2, IN1);
   tristate32L ts4(not_WRE1, IN2_1, IN2);
   tristate32L ts5(not_WRE2, IN2_2, IN2);
   tristate32L ts6(not_WRE1, IN3_1, IN3);
   tristate32L ts7(not_WRE2, IN3_2, IN3);

   // four 32-bit registers
//   latch32 l0(1'b1, IN0, en0, 1'b1, OT0);
//   latch32 l1(1'b1, IN1, en1, 1'b1, OT1);
//   latch32 l2(1'b1, IN2, en2, 1'b1, OT2);
//   latch32 l3(1'b1, IN3, en3, 1'b1, OT3);
   reg32e$ r0(CLK, IN0, OT0, , Reset, 1'b1, en0);
   reg32e$ r1(CLK, IN1, OT1, , Reset, 1'b1, en1);
   reg32e$ r2(CLK, IN2, OT2, , Reset, 1'b1, en2);
   reg32e$ r3(CLK, IN3, OT3, , Reset, 1'b1, en3);

   // mux for RE1
   mux4_32 m1(OUT1, OT0, OT1, OT2, OT3, RE1[0], RE1[1]);

      // mux for RE2
   mux4_32 m2(OUT2, OT0, OT1, OT2, OT3, RE2[0], RE2[1]);

      // mux for RE3
   mux4_32 m3(OUT3, OT0, OT1, OT2, OT3, RE3[0], RE3[1]);
endmodule // regfile4_32

/*
 // soon to be replaced
 module memory(addr1, data1, addr2, data2, wraddr, wrdata, wr);
 input [0:31]   addr1, addr2, wraddr, wrdata;
 input  wr;
 output [0:31] data1, data2;
 reg [0:31]      mem[0:4096];
 reg [0:31]      data1, data2;

 always @(addr1 or addr2)
 begin
 #2 data1 = mem[addr1[0:29]];
 data2 = mem[addr2[0:29]];
 if (wr == 1'b1)
 mem[wraddr[0:29]] = wrdata;
     end // always @ (addr)
 endmodule // memory
 */
```

```
module comp_1(OUT,IN1, IN2);
        output OUT;
        input IN1, IN2;

        nor2$ x0 (temp_OUT0, IN1, IN2);
        and2$ a0 (a_OUT0, IN1, IN2);
        or2$  o0 (OUT, temp_OUT0, a_OUT0);

endmodule





module comp_32(OUT, IN1, IN2);
   output       OUT;
   input [0:31] IN1, IN2;

   xor2$ x0 (temp_OUT0, IN1[0], IN2[0]);
   xor2$ x1 (temp_OUT1, IN1[1], IN2[1]);
   xor2$ x2 (temp_OUT2, IN1[2], IN2[2]);
   xor2$ x3 (temp_OUT3, IN1[3], IN2[3]);
   xor2$ x4 (temp_OUT4, IN1[4], IN2[4]);
   xor2$ x5 (temp_OUT5, IN1[5], IN2[5]);
   xor2$ x6 (temp_OUT6, IN1[6], IN2[6]);
   xor2$ x7 (temp_OUT7, IN1[7], IN2[7]);
   xor2$ x8 (temp_OUT8, IN1[8], IN2[8]);
   xor2$ x9 (temp_OUT9, IN1[9], IN2[9]);
   xor2$ x10 (temp_OUT10, IN1[10], IN2[10]);
   xor2$ x11 (temp_OUT11, IN1[11], IN2[11]);
   xor2$ x12 (temp_OUT12, IN1[12], IN2[12]);
   xor2$ x13 (temp_OUT13, IN1[13], IN2[13]);
   xor2$ x14 (temp_OUT14, IN1[14], IN2[14]);
   xor2$ x15 (temp_OUT15, IN1[15], IN2[15]);
   xor2$ x16 (temp_OUT16, IN1[16], IN2[16]);
   xor2$ x17 (temp_OUT17, IN1[17], IN2[17]);
   xor2$ x18 (temp_OUT18, IN1[18], IN2[18]);
   xor2$ x19 (temp_OUT19, IN1[19], IN2[19]);
   xor2$ x20 (temp_OUT20, IN1[20], IN2[20]);
   xor2$ x21 (temp_OUT21, IN1[21], IN2[21]);
   xor2$ x22 (temp_OUT22, IN1[22], IN2[22]);
   xor2$ x23 (temp_OUT23, IN1[23], IN2[23]);
   xor2$ x24 (temp_OUT24, IN1[24], IN2[24]);
   xor2$ x25 (temp_OUT25, IN1[25], IN2[25]);
   xor2$ x26 (temp_OUT26, IN1[26], IN2[26]);
   xor2$ x27 (temp_OUT27, IN1[27], IN2[27]);
   xor2$ x28 (temp_OUT28, IN1[28], IN2[28]);
   xor2$ x29 (temp_OUT29, IN1[29], IN2[29]);
   xor2$ x30 (temp_OUT30, IN1[30], IN2[30]);
   xor2$ x31 (temp_OUT31, IN1[31], IN2[31]);


   nor4$ n0 (n_OUT0, temp_OUT0, temp_OUT1, temp_OUT2, temp_OUT3);
   nor4$ n1 (n_OUT1, temp_OUT4, temp_OUT5, temp_OUT6, temp_OUT7);
   nor4$ n2 (n_OUT2, temp_OUT8, temp_OUT9, temp_OUT10, temp_OUT11);
   nor4$ n3 (n_OUT3, temp_OUT12, temp_OUT13, temp_OUT14, temp_OUT15);
   nor4$ n4 (n_OUT4, temp_OUT16, temp_OUT17, temp_OUT18, temp_OUT19);
   nor4$ n5 (n_OUT5, temp_OUT20, temp_OUT21, temp_OUT22, temp_OUT23);
   nor4$ n6 (n_OUT6, temp_OUT24, temp_OUT25, temp_OUT26, temp_OUT27);
   nor4$ n7 (n_OUT7, temp_OUT28, temp_OUT29, temp_OUT30, temp_OUT31);


   and4$ a0 (a_OUT0, n_OUT0, n_OUT1, n_OUT2, n_OUT3);
   and4$ a1 (a_OUT1, n_OUT4, n_OUT5, n_OUT6, n_OUT7);

      and2$ a3 (OUT, a_OUT0, a_OUT1);
   endmodule // comp_32
```

```
// buffers checked
module inv16( out, in);
   input [0:15]  in;
   output [0:15] out;

   inv1$ i0 ( out[0], in[0] );
   inv1$ i1 ( out[1], in[1] );
   inv1$ i2 ( out[2], in[2] );
   inv1$ i3 ( out[3], in[3] );
   inv1$ i4 ( out[4], in[4] );
   inv1$ i5 ( out[5], in[5] );
   inv1$ i6 ( out[6], in[6] );
   inv1$ i7 ( out[7], in[7] );
   inv1$ i8 ( out[8], in[8] );
   inv1$ i9 ( out[9], in[9] );
   inv1$ i10 ( out[10], in[10] );
   inv1$ i11 ( out[11], in[11] );
   inv1$ i12 ( out[12], in[12] );
   inv1$ i13 ( out[13], in[13] );
   inv1$ i14 ( out[14], in[14] );
   inv1$ i15 ( out[15], in[15] );
endmodule // inv16


// buffers checked
module inv32en(OUT, IN, EN);
   input [0:31]  IN;
   input         EN;
   output [0:31] OUT;

   wire [0:31] temp_out;

   inv16 i1(temp_out[0:15], IN[0:15]);
   inv16 i2(temp_out[16:31], IN[16:31]);
   mux2_32 m1(OUT, IN, temp_out, EN);
endmodule // inv32

// buffers checked
module inv32(OUT, IN);
   input [0:31]  IN;
   output [0:31] OUT;

   inv16 i1(OUT[0:15], IN[0:15]);
   inv16 i2(OUT[16:31], IN[16:31]);
endmodule // inv32


/*
 // buffers checked
 module or9(OUT, in0, in1, in2, in3, in4, in5, in6, in7, in8);
 input  in0, in1, in2, in3, in4, in5, in6, in7, in8;
 output OUT;

 or3$ o1(temp1, in0, in1, in2);
 or3$ o2(temp2, in3, in4, in5);
 or3$ o3(temp3, in6, in7, in8);
 or3$ o4(OUT, temp1, temp2, temp3);

 endmodule // or9
*/


module or5 ( out, in0, in1, in2, in3, in4 );
        input in0, in1, in2, in3, in4;
        output out;

        or3$    o1 ( o1Out, in0, in1, in2 );
        or2$    o2 ( o2Out, in3, in4 );
        or2$    o3 ( out, o1Out, o2Out );
endmodule // or5


// buffers checked
module or6(OUT, in0, in1, in2, in3, in4, in5);
   input  in0, in1, in2, in3, in4, in5;
   output OUT;

   or3$ o1(temp1, in0, in1, in2);
   or3$ o2(temp2, in3, in4, in5);
   or2$ o4(OUT, temp1, temp2);

endmodule // or6


// buffers checked
module or10(OUT, in0, in1, in2, in3, in4, in5, in6, in7, in8, in9);
   input  in0, in1, in2, in3, in4, in5, in6, in7, in8, in9;
   output OUT;

   or3$ o1(temp1, in0, in1, in2);
   or3$ o2(temp2, in3, in4, in5);
   or4$ o3(temp3, in6, in7, in8, in9);
   or3$ o4(OUT, temp1, temp2, temp3);

endmodule // or10

// buffers checked
module or11(OUT, in0, in1, in2, in3, in4, in5, in6, in7, in8, in9, in10);
   input  in0, in1, in2, in3, in4, in5, in6, in7, in8, in9, in10;
   output OUT;

   or3$ o1(temp1, in0, in1, in2);
   or4$ o2(temp2, in3, in4, in5, in6);
   or4$ o3(temp3, in7, in8, in9, in10);
   or3$ o4(OUT, temp1, temp2, temp3);

endmodule // or11


// buffers checked
module mux2_2(Y, IN0, IN1, S0);
   output [0:1] Y;
   input [0:1]   IN0, IN1;
   input         S0;

   mux2$ m0(Y[0], IN0[0], IN1[0], S0);
   mux2$ m1(Y[1], IN0[1], IN1[1], S0);
endmodule // mux2_2


// buffers checked
module mux2_3(Y, IN0, IN1, S0);
   output [0:2] Y;
   input [0:2]   IN0, IN1;
   input         S0;

   mux2$ m0(Y[0], IN0[0], IN1[0], S0);
   mux2$ m1(Y[1], IN0[1], IN1[1], S0);
   mux2$ m2(Y[2], IN0[2], IN1[2], S0);
endmodule // mux2_3
```

```
// buffers checked
module mux2_4(Y, IN0, IN1, S0);
   output [0:3] Y;
   input [0:3]   IN0, IN1;
   input         S0;

   mux2$ m0(Y[0], IN0[0], IN1[0], S0);
   mux2$ m1(Y[1], IN0[1], IN1[1], S0);
   mux2$ m2(Y[2], IN0[2], IN1[2], S0);
   mux2$ m3(Y[3], IN0[3], IN1[3], S0);
endmodule // mux2_4

/*
 // buffers checked
 module mux2_5(Y, IN0, IN1, S0);
 output [0:4] Y;
 input [0:4]     IN0, IN1;
 input   S0;

 mux2$ m0(Y[0], IN0[0], IN1[0], S0);
 mux2$ m1(Y[1], IN0[1], IN1[1], S0);
 mux2$ m2(Y[2], IN0[2], IN1[2], S0);
 mux2$ m3(Y[3], IN0[3], IN1[3], S0);
 mux2$ m4(Y[4], IN0[4], IN1[4], S0);
 endmodule // mux2_5
 */

// buffers checked
module mux1_32(Y, IN, S);
   output       Y;
   input [0:31] IN;
   input [0:4]  S;

   mux4$ m1(t1, IN[0], IN[1], IN[2], IN[3], S[4], S[3]);
   mux4$ m2(t2, IN[4], IN[5], IN[6], IN[7], S[4], S[3]);
   mux4$ m3(t3, IN[8], IN[9], IN[10], IN[11], S[4], S[3]);
   mux4$ m4(t4, IN[12], IN[13], IN[14], IN[15], S[4], S[3]);
   buffer$ b1(s4, S[4]);
   buffer$ b2(s3, S[3]);
   mux4$ m5(t5, IN[16], IN[17], IN[18], IN[19], s4, s3);
   mux4$ m6(t6, IN[20], IN[21], IN[22], IN[23], s4, s3);
   mux4$ m7(t7, IN[24], IN[25], IN[26], IN[27], s4, s3);
   mux4$ m8(t8, IN[28], IN[29], IN[30], IN[31], s4, s3);
   mux4$ m9(t9, t1, t2, t3, t4, S[2], S[1]);
   mux4$ m10(t10, t5, t6, t7, t8, S[2], S[1]);
   mux2$ m11(Y, t9, t10, S[0]);
endmodule // mux1_32


// buffers checked
module mux2_32(Y, IN0, IN1, S0);
   output [0:31] Y;
   input [0:31]  IN0, IN1;
   input         S0;

   mux2_16$ mux1(Y[0:15], IN0[0:15], IN1[0:15], S0);
   mux2_16$ mux2(Y[16:31], IN0[16:31], IN1[16:31], S0);
endmodule // mux2_32


// buffers checked
module mux2_64(Y, IN0, IN1, S0);
   output [0:63] Y;
   input [0:63]  IN0, IN1;
   input         S0;
   mux2_32 mux1(Y[0:31], IN0[0:31], IN1[0:31], S0);
   mux2_32 mux2(Y[32:63], IN0[32:63], IN1[32:63], S0);
endmodule // mux2_64


// buffers checked
module mux2_128(Y, IN0, IN1, S0);
   output [0:127] Y;
   input [0:127]  IN0, IN1;
   input          S0;
   mux2_64 mux1(Y[0:63], IN0[0:63], IN1[0:63], S0);
   mux2_64 mux2(Y[64:127], IN0[64:127], IN1[64:127], S0);
endmodule // mux2_128


// buffers checked
module mux3_32(Y, IN0, IN1, IN2, S1, S0);
   output [0:31] Y;
   input [0:31]  IN0, IN1, IN2;
   input         S0, S1;

   mux3_16$ mux1(Y[0:15], IN0[0:15], IN1[0:15], IN2[0:15], S0, S1);
   mux3_16$ mux2(Y[16:31], IN0[16:31], IN1[16:31], IN2[16:31], S0, S1);
endmodule // mux3_32


// buffers checked
module mux3_64(Y, IN0, IN1, IN2, S1, S0);
   output [0:63] Y;
   input [0:63]  IN0, IN1, IN2;
   input         S0, S1;

   mux3_32 mux1(Y[0:31], IN0[0:31], IN1[0:31], IN2[0:31], S1, S0);
   mux3_32 mux2(Y[32:63], IN0[32:63], IN1[32:63], IN2[32:63], S1, S0);
endmodule // mux3_64


// buffers checked
module mux3_128(Y, IN0, IN1, IN2, S1, S0);
   output [0:127] Y;
   input [0:127]  IN0, IN1, IN2;
   input          S0, S1;

   mux3_64 mux1(Y[0:63], IN0[0:63], IN1[0:63], IN2[0:63], S1, S0);
   mux3_64 mux2(Y[64:127], IN0[64:127], IN1[64:127], IN2[64:127], S1, S0);
endmodule // mux3_128


// buffers checked
module mux4_32(Y, IN0, IN1, IN2, IN3, S1, S0);
   output [0:31] Y;
   input [0:31]  IN0, IN1, IN2, IN3;
   input         S0, S1;

   inv1$ i2(not_S1, S1);
   inv1$ i3(not_S0, S0);
   nand2$ a4(sel_R0, not_S1, not_S0);
   nand2$ a5(sel_R1, not_S1, S0);
   nand2$ a6(sel_R2, S1, not_S0);
   nand2$ a7(sel_R3, S1, S0);
   tristate32L t4(sel_R0, IN0, Y);
   tristate32L t5(sel_R1, IN1, Y);
   tristate32L t6(sel_R2, IN2, Y);
   tristate32L t7(sel_R3, IN3, Y);
endmodule // mux4_32

/*
 module mux4_64(Y, IN0, IN1, IN2, IN3, S1, S0);
 output [0:63] Y;
 input [0:63]  IN0, IN1, IN2, IN3;
 input   S0, S1;

 mux4_32 m0(Y[0:31], IN0[0:31], IN1[0:31], IN2[0:31], IN3[0:31], S1, S0);
 mux4_32 m1(Y[32:63], IN0[32:63], IN1[32:63], IN2[32:63], IN3[32:63], S1, S0);
 endmodule // mux4_64
 */

// buffers checked
module mux5_20(Y, IN0, IN1, IN2, IN3, IN4, S0, S1, S2);
   output [0:19] Y;
   input [0:19]  IN0, IN1, IN2, IN3, IN4;
   input         S0, S1, S2;

   inv1$ i0(not_S0, S0);
   inv1$ i1(not_S1, S1);
   inv1$ i2(not_S2, S2);
   nand3$ a0(sel_R0, not_S0, not_S1, not_S2);
   nand3$ a1(sel_R1, not_S0, not_S1, S2);
   nand3$ a2(sel_R2, not_S0, S1, not_S2);
   nand3$ a3(sel_R3, not_S0, S1, S2);
   nand3$ a4(sel_R4, S0, not_S1, not_S2);
   tristate20L t0(sel_R0, IN0, Y);
   tristate20L t1(sel_R1, IN1, Y);
   tristate20L t2(sel_R2, IN2, Y);
   tristate20L t3(sel_R3, IN3, Y);
   tristate20L t4(sel_R4, IN4, Y);
endmodule // mux5_20


// buffers checked
module mux8_32(Y, IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, S0, S1, S2);
   input [0:31] IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7;
   input         S2, S1, S0;
   output [0:31]         Y;

   inv1$ i41(not_S0, S0);
   inv1$ i51(not_S1, S1);
   inv1$ i61(not_S2, S2);
   nand3$ a11(sel_R0, not_S0, not_S1, not_S2);
   nand3$ a21(sel_R1, not_S0, not_S1, S2);
   nand3$ a31(sel_R2, not_S0, S1, not_S2);
   nand3$ a41(sel_R3, not_S0, S1, S2);
   nand3$ a51(sel_R4, S0, not_S1, not_S2);
   nand3$ a61(sel_R5, S0, not_S1, S2);
   nand3$ a71(sel_R6, S0, S1, not_S2);
   nand3$ a81(sel_R7, S0, S1, S2);
   tristate32L t11(sel_R0, IN0, Y);
   tristate32L t21(sel_R1, IN1, Y);
   tristate32L t31(sel_R2, IN2, Y);
   tristate32L t41(sel_R3, IN3, Y);
   tristate32L t51(sel_R4, IN4, Y);
   tristate32L t61(sel_R5, IN5, Y);
   tristate32L t71(sel_R6, IN6, Y);
   tristate32L t81(sel_R7, IN7, Y);

endmodule // mux8_32
```

```
module mux16_8(Y, IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7,
               IN8, IN9, IN10, IN11, IN12, IN13, IN14, IN15,
               S0, S1, S2, S3);
   input [0:7]  IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7,
                IN8, IN9, IN10, IN11, IN12, IN13, IN14, IN15;
   input        S3, S2, S1, S0;
   output [0:7] Y;

   inv1$ i41(not_S0, S0);
   inv1$ i51(not_S1, S1);
   inv1$ i61(not_S2, S2);
   inv1$ i71(not_S3, S3);

   nand4$ a01(sel_R0, not_S0, not_S1, not_S2, not_S3);
   nand4$ a11(sel_R1, not_S0, not_S1, not_S2,     S3);
   nand4$ a21(sel_R2, not_S0, not_S1,     S2, not_S3);
   nand4$ a31(sel_R3, not_S0, not_S1,     S2,     S3);
   nand4$ a41(sel_R4, not_S0,     S1, not_S2, not_S3);
   nand4$ a51(sel_R5, not_S0,     S1, not_S2,     S3);
   nand4$ a61(sel_R6, not_S0,     S1,     S2, not_S3);
   nand4$ a71(sel_R7, not_S0,     S1,     S2,     S3);
   nand4$ a81(sel_R8,     S0, not_S1, not_S2, not_S3);
   nand4$ a91(sel_R9,     S0, not_S1, not_S2,     S3);
   nand4$ aa1(sel_Ra,     S0, not_S1,     S2, not_S3);
   nand4$ ab1(sel_Rb,     S0, not_S1,     S2,     S3);
   nand4$ ac1(sel_Rc,     S0,     S1, not_S2, not_S3);
   nand4$ ad1(sel_Rd,     S0,     S1, not_S2,     S3);
   nand4$ ae1(sel_Re,     S0,     S1,     S2, not_S3);
   nand4$ af1(sel_Rf,     S0,     S1,     S2,     S3);

   tristate8L$ t01(sel_R0, IN0, Y);
   tristate8L$ t11(sel_R1, IN1, Y);
   tristate8L$ t21(sel_R2, IN2, Y);
   tristate8L$ t31(sel_R3, IN3, Y);
   tristate8L$ t41(sel_R4, IN4, Y);
   tristate8L$ t51(sel_R5, IN5, Y);
   tristate8L$ t61(sel_R6, IN6, Y);
   tristate8L$ t71(sel_R7, IN7, Y);
   tristate8L$ t81(sel_R8, IN8, Y);
   tristate8L$ t91(sel_R9, IN9, Y);
   tristate8L$ ta1(sel_Ra, IN10, Y);
   tristate8L$ tb1(sel_Rb, IN11, Y);
   tristate8L$ tc1(sel_Rc, IN12, Y);
   tristate8L$ td1(sel_Rd, IN13, Y);
   tristate8L$ te1(sel_Re, IN14, Y);
   tristate8L$ tf1(sel_Rf, IN15, Y);

endmodule // mux16_8


// tested, works well
module decoder4_16 ( SEL, Y, YBAR );
   input [0:3]   SEL;
   output [0:15] Y, YBAR;

   wire [0:15]   Y_temp, YBAR_temp;

   decoder3_8$ d0 ( SEL[1:3], Y_temp[0:7], YBAR_temp[0:7] );
   decoder3_8$ d1 ( SEL[1:3], Y_temp[8:15], YBAR_temp[8:15] );
   mux2_8$ m0 ( {Y[15],Y[14],Y[13],Y[12],Y[11],Y[10],Y[9],Y[8]},
                8'h0, Y_temp[0:7], SEL[0] );
   mux2_8$ m1 ( {Y[7],Y[6],Y[5],Y[4],Y[3],Y[2],Y[1],Y[0]},
                Y_temp[8:15], 8'h0, SEL[0] );
   mux2_8$ m2 ( {YBAR[15],YBAR[14],YBAR[13],YBAR[12],YBAR[11],
                 YBAR[10],YBAR[9],YBAR[8]}, 8'hff, YBAR_temp[0:7], SEL[0] );
   mux2_8$ m3 ( {YBAR[7],YBAR[6],YBAR[5],YBAR[4],YBAR[3],
                YBAR[2],YBAR[1],YBAR[0]}, YBAR_temp[8:15], 8’hff, SEL[0] );

endmodule // decoder4_16


// 1-bit, 8-way multiplexor
// tested, works well
module mux8( Y, IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, S0, S1, S2 );
   input  IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7;
   input  S2, S1, S0;
   output Y;

   inv1$ i41(not_S0, S0);
   inv1$ i51(not_S1, S1);
   inv1$ i61(not_S2, S2);
   nand3$ a11(sel_R0, not_S0, not_S1, not_S2);
   nand3$ a21(sel_R1, not_S0, not_S1, S2);
   nand3$ a31(sel_R2, not_S0, S1, not_S2);
   nand3$ a41(sel_R3, not_S0, S1, S2);
   nand3$ a51(sel_R4, S0, not_S1, not_S2);
   nand3$ a61(sel_R5, S0, not_S1, S2);
   nand3$ a71(sel_R6, S0, S1, not_S2);
   nand3$ a81(sel_R7, S0, S1, S2);
   tristateL$ t11(sel_R0, IN0, Y);
   tristateL$ t21(sel_R1, IN1, Y);
   tristateL$ t31(sel_R2, IN2, Y);
   tristateL$ t41(sel_R3, IN3, Y);
   tristateL$ t51(sel_R4, IN4, Y);
   tristateL$ t61(sel_R5, IN5, Y);
   tristateL$ t71(sel_R6, IN6, Y);
   tristateL$ t81(sel_R7, IN7, Y);

endmodule // mux8


module demux4(in, out0, out1, out2, out3, s0, s1);
   output out0, out1, out2, out3;
   input in, s0, s1;

   inv1$ i1(not_s0, s0);
   inv1$ i2(not_s1, s1);
   and3$ n1(out0, in, not_s0, not_s1);
   and3$ n2(out1, in, not_s0, s1);
   and3$ n3(out2, in, s0, not_s1);
   and3$ n4(out3, in, s0, s1);
endmodule // mux8

/*
 module demux4_4(in, out0, out1, out2, out3, s0, s1);
 output [0:3]   out0, out1, out2, out3;
 input [0:3]    in;
 input  s0, s1;

 demux4 d0 ( in[0], out0[0], out1[0], out2[0], out3[0], s0, s1 );
 demux4 d1 ( in[1], out0[1], out1[1], out2[1], out3[1], s0, s1 );
 demux4 d2 ( in[2], out0[2], out1[2], out2[2], out3[2], s0, s1 );
 demux4 d3 ( in[3], out0[3], out1[3], out2[3], out3[3], s0, s1 );
 endmodule // demux4_4
*/

module demuxinv4(in, out0, out1, out2, out3, s0, s1);
   output out0, out1, out2, out3;
   input  in, s0, s1;
```

```
   inv1$ i1(not_s0, s0);
   inv1$ i2(not_s1, s1);
   nand3$ n1(out0, in, not_s0, not_s1);
   nand3$ n2(out1, in, not_s0, s1);
   nand3$ n3(out2, in, s0, not_s1);
   nand3$ n4(out3, in, s0, s1);
endmodule // demuxinv4


module demuxinv4_4(in, out0, out1, out2, out3, s0, s1);
   output [0:3] out0, out1, out2, out3;
   input [0:3]  in;
   input        s0, s1;

   demuxinv4 d0 ( in[0], out0[0], out1[0], out2[0], out3[0], s0, s1 );
   demuxinv4 d1 ( in[1], out0[1], out1[1], out2[1], out3[1], s0, s1 );
   demuxinv4 d2 ( in[2], out0[2], out1[2], out2[2], out3[2], s0, s1 );
   demuxinv4 d3 ( in[3], out0[3], out1[3], out2[3], out3[3], s0, s1 );
endmodule // demuxinv4_4

/*
 module demux2_32(IN, OUT0, OUT1, S0);
 input [0:31]  IN;
 output [0:31] OUT0, OUT1;
 input   S0;
 wire [0:31]     TEMPOUT;

 inv1$ in0(not_S0, S0);
 tristate32L ts0(    S0, IN, OUT0);
 tristate32L ts1(not_S0, IN, OUT1);
 endmodule // mux2_32
 */

module demux4_32(IN, OUT0, OUT1, OUT2, OUT3, S1, S0);
   input [0:31]  IN;
   output [0:31] OUT0, OUT1, OUT2, OUT3;
   input          S0, S1;
   wire [0:31]    TEMPOUT;

   inv1$ in0(not_S1, S1);
   inv1$ in1(not_S0, S0);
   nand2$ an0(sel_WR0, not_S1, not_S0);
   nand2$ an1(sel_WR1, not_S1, S0);
   nand2$ an2(sel_WR2, S1, not_S0);
   nand2$ an3(sel_WR3, S1, S0);
   tristate32L ts0(sel_WR0, IN, OUT0);
   tristate32L ts1(sel_WR1, IN, OUT1);
   tristate32L ts2(sel_WR2, IN, OUT2);
   tristate32L ts3(sel_WR3, IN, OUT3);
endmodule // mux4_32


// 16-way, 8-bit demux
// fully tested, works well
module demux16_8(IN, OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7,
                 OUT8, OUT9, OUT10, OUT11, OUT12, OUT13, OUT14, OUT15,
                 S0, S1, S2, S3);
   input [0:7]  IN;
   input        S3, S2, S1, S0;
   output [0:7] OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7,
                OUT8, OUT9, OUT10, OUT11, OUT12, OUT13, OUT14, OUT15;

   inv1$ i41(not_S0, S0);
   inv1$ i51(not_S1, S1);
   inv1$ i61(not_S2, S2);
   inv1$ i71(not_S3, S3);

   nand4$ a01(sel_R0, not_S0, not_S1, not_S2, not_S3);
   nand4$ a11(sel_R1, not_S0, not_S1, not_S2,     S3);
   nand4$ a21(sel_R2, not_S0, not_S1,     S2, not_S3);
   nand4$ a31(sel_R3, not_S0, not_S1,     S2,     S3);
   nand4$ a41(sel_R4, not_S0,     S1, not_S2, not_S3);
   nand4$ a51(sel_R5, not_S0,     S1, not_S2,     S3);
   nand4$ a61(sel_R6, not_S0,     S1,     S2, not_S3);
   nand4$ a71(sel_R7, not_S0,     S1,     S2,     S3);
   nand4$ a81(sel_R8,     S0, not_S1, not_S2, not_S3);
   nand4$ a91(sel_R9,     S0, not_S1, not_S2,     S3);
   nand4$ aa1(sel_Ra,     S0, not_S1,     S2, not_S3);
   nand4$ ab1(sel_Rb,     S0, not_S1,     S2,     S3);
   nand4$ ac1(sel_Rc,     S0,     S1, not_S2, not_S3);
   nand4$ ad1(sel_Rd,     S0,     S1, not_S2,     S3);
   nand4$ ae1(sel_Re,     S0,     S1,     S2, not_S3);
   nand4$ af1(sel_Rf,     S0,     S1,     S2,     S3);

   tristate8L$ t01(sel_R0, IN, OUT0);
   tristate8L$ t11(sel_R1, IN, OUT1);
   tristate8L$ t21(sel_R2, IN, OUT2);
   tristate8L$ t31(sel_R3, IN, OUT3);
   tristate8L$ t41(sel_R4, IN, OUT4);
   tristate8L$ t51(sel_R5, IN, OUT5);
   tristate8L$ t61(sel_R6, IN, OUT6);
   tristate8L$ t71(sel_R7, IN, OUT7);
   tristate8L$ t81(sel_R8, IN, OUT8);
   tristate8L$ t91(sel_R9, IN, OUT9);
   tristate8L$ ta1(sel_Ra, IN, OUT10);
   tristate8L$ tb1(sel_Rb, IN, OUT11);
   tristate8L$ tc1(sel_Rc, IN, OUT12);
   tristate8L$ td1(sel_Rd, IN, OUT13);
   tristate8L$ te1(sel_Re, IN, OUT14);
   tristate8L$ tf1(sel_Rf, IN, OUT15);

endmodule // demux16_8


module tristate20L(enbar, in, out);
   input         enbar;
   input [0:19]  in;
   output [0:19] out;

   tristate16L$ t0(enbar, in[0:15], out[0:15]);
   tristateL$ t1(enbar, in[16], out[16]);
   tristateL$ t2(enbar, in[17], out[17]);
   tristateL$ t3(enbar, in[18], out[18]);
   tristateL$ t4(enbar, in[19], out[19]);
endmodule // tristate20L


module tristate32L(enbar, in, out);
   input         enbar;
   input [0:31]  in;
   output [0:31] out;

   tristate16L$ t0(enbar, in[0:15], out[0:15]);
   tristate16L$ t1(enbar, in[16:31], out[16:31]);
endmodule // tristate32L

/*
 module latch32(CLR, D, EN, PRE, Q);
 input   CLR, EN, PRE;
 input [0:31]    D;
 output [0:31] Q;

 latch16$ l0(CLR, D[0:15], EN, PRE, Q[0:15]);
 latch16$ l1(CLR, D[16:31], EN, PRE, Q[16:31]);
 endmodule // latch32
 */

module reg4e(CLK, D, Q, QBAR, CLR, PRE, EN);
   input       CLK, CLR, PRE, EN;
   input [0:3] D;
   wire [0:3]  Din;
   output [0:3] Q, QBAR;

   mux2_4 m1(Din, Q, D, EN);
   dff$ d0(CLK, Din[0], Q[0], QBAR[0], CLR, PRE);
   dff$ d1(CLK, Din[1], Q[1], QBAR[1], CLR, PRE);
   dff$ d2(CLK, Din[2], Q[2], QBAR[2], CLR, PRE);
   dff$ d3(CLK, Din[3], Q[3], QBAR[3], CLR, PRE);
endmodule // reg4e


module reg8e(CLK, D, Q, QBAR, CLR, PRE, EN);
   input       CLK, CLR, PRE, EN;
   input [0:7] D;
   wire [0:7]  Din;
   output [0:7] Q, QBAR;

   mux2_8$ m1(Din, Q, D, EN);
   dff$ d0(CLK, Din[0], Q[0], QBAR[0], CLR, PRE);
   dff$ d1(CLK, Din[1], Q[1], QBAR[1], CLR, PRE);
   dff$ d2(CLK, Din[2], Q[2], QBAR[2], CLR, PRE);
   dff$ d3(CLK, Din[3], Q[3], QBAR[3], CLR, PRE);
   dff$ d4(CLK, Din[4], Q[4], QBAR[4], CLR, PRE);
   dff$ d5(CLK, Din[5], Q[5], QBAR[5], CLR, PRE);
   dff$ d6(CLK, Din[6], Q[6], QBAR[6], CLR, PRE);
   dff$ d7(CLK, Din[7], Q[7], QBAR[7], CLR, PRE);
endmodule // reg8e


module reg20e(CLK, D, Q, QBAR, CLR, PRE, EN);
   input         CLK, CLR, PRE, EN;
   input [0:19]  D;
   wire [0:19]   Din;
   output [0:19] Q, QBAR;

   reg8e r0 (CLK, D[0:7], Q[0:7], QBAR[0:7], CLR, PRE, EN);
   reg8e r1 (CLK, D[8:15], Q[8:15], QBAR[8:15], CLR, PRE, EN);
   reg4e r2 (CLK, D[16:19], Q[16:19], QBAR[16:19], CLR, PRE, EN);
endmodule // reg20e


module reg24e(CLK, D, Q, QBAR, CLR, PRE, EN);
   input         CLK, CLR, PRE, EN;
   input [0:23]  D;
   wire [0:23]   Din;
   output [0:23] Q, QBAR;

   reg8e r0 (CLK, D[0:7], Q[0:7], QBAR[0:7], CLR, PRE, EN);
   reg8e r1 (CLK, D[8:15], Q[8:15], QBAR[8:15], CLR, PRE, EN);
   reg8e r2 (CLK, D[16:23], Q[16:23], QBAR[16:23], CLR, PRE, EN);
endmodule // reg24e


module reg40e(CLK, D, Q, QBAR, CLR, PRE, EN);
   input         CLK, CLR, PRE, EN;
   input [0:39]  D;
   wire [0:39]   Din;
   output [0:39] Q, QBAR;

   reg20e r0 (CLK, D[0:19], Q[0:19], QBAR[0:19], CLR, PRE, EN);
   reg20e r1 (CLK, D[20:39], Q[20:39], QBAR[20:39], CLR, PRE, EN);
endmodule // reg40e


// initializes to a valid control bubble
module reg64e_pipe(CLK, D, Q, QBAR, CLR, PRE, EN);
   input CLK, EN, CLR, PRE;
   input [0:63] D;
   output [0:63] Q;
   output [0:63] QBAR;
   wire          not_used;

   reg64e$ r1(CLK, {D[0:61], 1'b0, D[63]}, {Q[0:61], not_used, Q[63]}, /*QBAR*/, CLR, 1'
b1, EN);
   // sets the PRE instead of the CLR to set the valid bit
   dffh r2(CLK, D[62], Q[62], /*QBAR*/, 1'b1, CLR, EN);
endmodule // reg64e_pipe


module demux8(in, out0, out1, out2, out3, out4, out5, out6, out7, s0, s1, s2);
   output  out0, out1, out2, out3, out4, out5, out6, out7, s0, s1, s2;
   input in;
   inv1$ i1(not_s0, s0);
   inv1$ i2(not_s1, s1);
   inv1$ i3(not_s2, s2);
   and4$ n1(out0, in, not_s0, not_s1, not_s2);
   and4$ n2(out1, in, not_s0, not_s1, s2);
   and4$ n3(out2, in, not_s0, s1, not_s2);
   and4$ n4(out3, in, not_s0, s1, s2);
   and4$ n5(out4, in, s0, not_s1, not_s2);
   and4$ n6(out5, in, s0, not_s1, s2);
   and4$ n7(out6, in, s0, s1, not_s2);
   and4$ n8(out7, in, s0, s1, s2);

endmodule // mux8


module ext1_16(out, in);
   output [0:15] out;
   input in;

   buffer$ b0(in0, in);
   buffer$ b1(in1, in);
   buffer$ b2(in2, in);
   assign out = {{5{in0}}, {5{in1}}, {5{in2}}, in};
endmodule // ext1_16


module signext16_32(out, in);
   output [0:31] out;
   input [0:15]  in;

   buffer$ b0(in0, in[0]);
   buffer$ b1(in1, in[0]);
   buffer$ b2(in2, in[0]);
   assign out[0:15] = {{5{in0}}, {5{in1}}, {5{in2}}, in[0]};
   assign out[16:31] = in;
endmodule // signext16_32
```

```
module adder32(a, b, cin, cout, out);
   input [0:31]  a, b;
   input         cin;
   output        cout;
   output [0:31] out;
   wire          temp;

   adder16$ a1(a[0:15], b[0:15], temp, cout, out[0:15]);
   adder16$ a2(a[16:31], b[16:31], cin, temp, out[16:31]);
endmodule // adder32


module alu32(a, b, cin, m, s, cout, out, ovf);
   input [0:31]  a, b;
   input         cin, m;
   input [0:3]   s;
   output        cout;
   output [0:31] out;
   output        ovf;

   alu16$ a2(a[16:31], b[16:31], cin, m, s, temp, out[16:31]);
   alu16vgp$ a1(a[0:15], b[0:15], temp, m, s, cout, out[0:15], ovf, g, p);
endmodule // alu32


module nor5 ( out, in0, in1, in2, in3, in4 );
        input in0, in1, in2, in3, in4;
        output out;

        or3$    o1 ( o1Out, in0, in1, in2 );
        or2$    o2 ( o2Out, in3, in4 );
        nor2$   n1 ( out, o1Out, o2Out );
endmodule


module nor32(OUT, IN);
   input [0:31] IN;
   output       OUT;

   or4$ n1(o1, IN[0], IN[1], IN[2], IN[3]);
   or4$ n2(o2, IN[4], IN[5], IN[6], IN[7]);
   or4$ n3(o3, IN[8], IN[9], IN[10], IN[11]);
   or4$ n4(o4, IN[12], IN[13], IN[14], IN[15]);
   or4$ n5(o5, IN[16], IN[17], IN[18], IN[19]);
   or4$ n6(o6, IN[20], IN[21], IN[22], IN[23]);
   or4$ n7(o7, IN[24], IN[25], IN[26], IN[27]);
   or4$ n8(o8, IN[28], IN[29], IN[30], IN[31]);
   or4$ n9(out1, o1, o2, o3, o4);
   or4$ n10(out2, o5, o6, o7, o8);
   nor2$ n11(OUT, out1, out2);
endmodule // nor32

module nor22(OUT, IN);
   input [0:21] IN;
   output       OUT;

   or4$ n1(o1, IN[0], IN[1], IN[2], IN[3]);
   or4$ n2(o2, IN[4], IN[5], IN[6], IN[7]);
   or4$ n3(o3, IN[8], IN[9], IN[10], IN[11]);
   or4$ n4(o4, IN[12], IN[13], IN[14], IN[15]);
   or4$ n5(o5, IN[16], IN[17], IN[18], IN[19]);
   or4$ n9(out1, o1, o2, o3, o4);
   or3$ n10(out2, o5, IN[20], IN[21]);
   nor2$ n11(OUT, out1, out2);
endmodule // nor22
```

```
module or32_notbitwise(OUT, IN);
   input [0:31] IN;
   output       OUT;

   or4$ n1(o1, IN[0], IN[1], IN[2], IN[3]);
   or4$ n2(o2, IN[4], IN[5], IN[6], IN[7]);
   or4$ n3(o3, IN[8], IN[9], IN[10], IN[11]);
   or4$ n4(o4, IN[12], IN[13], IN[14], IN[15]);
   or4$ n5(o5, IN[16], IN[17], IN[18], IN[19]);
   or4$ n6(o6, IN[20], IN[21], IN[22], IN[23]);
   or4$ n7(o7, IN[24], IN[25], IN[26], IN[27]);
   or4$ n8(o8, IN[28], IN[29], IN[30], IN[31]);
   or4$ n9(out1, o1, o2, o3, o4);
   or4$ n10(out2, o5, o6, o7, o8);
   or2$ n11(OUT, out1, out2);
endmodule // or32_notbitwise

module or32(OUT, IN1, IN2);
   input [0:31] IN1, IN2;
   output [0:31]        OUT;
   or2$ o1(OUT[0], IN1[0], IN2[0]);
   or2$ o2(OUT[1], IN1[1], IN2[1]);
   or2$ o3(OUT[2], IN1[2], IN2[2]);
   or2$ o4(OUT[3], IN1[3], IN2[3]);
   or2$ o5(OUT[4], IN1[4], IN2[4]);
   or2$ o6(OUT[5], IN1[5], IN2[5]);
   or2$ o7(OUT[6], IN1[6], IN2[6]);
   or2$ o8(OUT[7], IN1[7], IN2[7]);
   or2$ o9(OUT[8], IN1[8], IN2[8]);
   or2$ o10(OUT[9], IN1[9], IN2[9]);
   or2$ o11(OUT[10], IN1[10], IN2[10]);
   or2$ o12(OUT[11], IN1[11], IN2[11]);
   or2$ o13(OUT[12], IN1[12], IN2[12]);
   or2$ o14(OUT[13], IN1[13], IN2[13]);
   or2$ o15(OUT[14], IN1[14], IN2[14]);
   or2$ o16(OUT[15], IN1[15], IN2[15]);
   or2$ o17(OUT[16], IN1[16], IN2[16]);
   or2$ o18(OUT[17], IN1[17], IN2[17]);
   or2$ o19(OUT[18], IN1[18], IN2[18]);
   or2$ o20(OUT[19], IN1[19], IN2[19]);
   or2$ o21(OUT[20], IN1[20], IN2[20]);
   or2$ o22(OUT[21], IN1[21], IN2[21]);
   or2$ o23(OUT[22], IN1[22], IN2[22]);
   or2$ o24(OUT[23], IN1[23], IN2[23]);
   or2$ o25(OUT[24], IN1[24], IN2[24]);
   or2$ o26(OUT[25], IN1[25], IN2[25]);
   or2$ o27(OUT[26], IN1[26], IN2[26]);
   or2$ o28(OUT[27], IN1[27], IN2[27]);
   or2$ o29(OUT[28], IN1[28], IN2[28]);
   or2$ o30(OUT[29], IN1[29], IN2[29]);
   or2$ o31(OUT[30], IN1[30], IN2[30]);
   or2$ o32(OUT[31], IN1[31], IN2[31]);
endmodule // or32


module and32(OUT, IN1, IN2);
   input [0:31] IN1, IN2;
   output [0:31]        OUT;
   and2$ o1(OUT[0], IN1[0], IN2[0]);
   and2$ o2(OUT[1], IN1[1], IN2[1]);
   and2$ o3(OUT[2], IN1[2], IN2[2]);
   and2$ o4(OUT[3], IN1[3], IN2[3]);
   and2$ o5(OUT[4], IN1[4], IN2[4]);
   and2$ o6(OUT[5], IN1[5], IN2[5]);
   and2$ o7(OUT[6], IN1[6], IN2[6]);
   and2$ o8(OUT[7], IN1[7], IN2[7]);
   and2$ o9(OUT[8], IN1[8], IN2[8]);
   and2$ o10(OUT[9], IN1[9], IN2[9]);
   and2$ o11(OUT[10], IN1[10], IN2[10]);
   and2$ o12(OUT[11], IN1[11], IN2[11]);
   and2$ o13(OUT[12], IN1[12], IN2[12]);
   and2$ o14(OUT[13], IN1[13], IN2[13]);
   and2$ o15(OUT[14], IN1[14], IN2[14]);
   and2$ o16(OUT[15], IN1[15], IN2[15]);
   and2$ o17(OUT[16], IN1[16], IN2[16]);
   and2$ o18(OUT[17], IN1[17], IN2[17]);
   and2$ o19(OUT[18], IN1[18], IN2[18]);
   and2$ o20(OUT[19], IN1[19], IN2[19]);
   and2$ o21(OUT[20], IN1[20], IN2[20]);
   and2$ o22(OUT[21], IN1[21], IN2[21]);
   and2$ o23(OUT[22], IN1[22], IN2[22]);
   and2$ o24(OUT[23], IN1[23], IN2[23]);
   and2$ o25(OUT[24], IN1[24], IN2[24]);
   and2$ o26(OUT[25], IN1[25], IN2[25]);
   and2$ o27(OUT[26], IN1[26], IN2[26]);
   and2$ o28(OUT[27], IN1[27], IN2[27]);
   and2$ o29(OUT[28], IN1[28], IN2[28]);
   and2$ o30(OUT[29], IN1[29], IN2[29]);
   and2$ o31(OUT[30], IN1[30], IN2[30]);
   and2$ o32(OUT[31], IN1[31], IN2[31]);
endmodule // and32

/*
 module decoder1_2(OUT1, OUT2, IN1, S0);
 input  IN1, S0;
 output OUT1, OUT2;

 inv1$ i1(not_S0, S0);
 and2$ a1(OUT1, IN1, not_S0);
 and2$ a2(OUT2, IN1, S0);
 endmodule // decoder1_2
 */

module and9(OUT, IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8);
   input IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8;
   output      OUT;

   and3$ a1(t1, IN0, IN1, IN2);
   and3$ a2(t2, IN3, IN4, IN5);
   and3$ a3(t3, IN6, IN7, IN8);

   and3$ a4(OUT, t1, t2, t3);
endmodule // and9


module and10(OUT, IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9);
   input IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9;
   output      OUT;

   and3$ a1(t1, IN0, IN1, IN2);
   and3$ a2(t2, IN3, IN4, IN5);
   and4$ a3(t3, IN6, IN7, IN8, IN9);

   and3$ a4(OUT, t1, t2, t3);
endmodule // and10

module encoder4_2(IN, OUT);
   input [0:3]  IN;
   output [0:1] OUT;

   inv1$ i1(not_IN0, IN[0]);
   inv1$ i2(not_IN1, IN[1]);
   inv1$ i3(not_IN2, IN[2]);
   inv1$ i4(not_IN3, IN[3]);

   and4$ a1(temp4, not_IN0, not_IN1, not_IN2, IN[3]);
   and4$ a2(temp3, not_IN0, not_IN1, IN[2], not_IN3);
   and4$ a3(temp2, not_IN0, IN[1], not_IN2, not_IN3);
   and4$ a4(temp1, IN[0], not_IN1, not_IN2, not_IN3);

   or2$ o1(OUT[0], temp1, temp3);
   or2$ o2(OUT[1], temp2, temp4);
endmodule // encoder4_2

module reg1(CLK, D, Q, CLR, EN);
   input  CLK, D, CLR, EN;
   output Q;
   mux2$ m1(Din, Q, D, EN); // enable selects input or old value
   dff$ d0(CLK, Din, Q, , CLR, 1’b1);
endmodule // reg1

/*
 module reg16e(CLK, Din, Q, QBAR, CLR, PRE,en);
 input   CLK, CLR, PRE, en;
 input [0:15]    Din;
 output [0:15] Q, QBAR;

 reg8e reg1(CLK, Din[0:7], Q[0:7], QBAR[0:7], CLR, PRE, en);
 reg8e reg2(CLK, Din[8:15], Q[8:15], QBAR[8:15], CLR, PRE, en);
 endmodule // reg16e
*/

module extdecoder22_5(IN, OUT);
   input [0:9]  IN;
   output [0:4] OUT;

   inv1$ i1(not_in0, IN[0]);
   inv1$ i2(not_in1, IN[1]);
   inv1$ i3(not_in2, IN[2]);
   inv1$ i4(not_in3, IN[3]);
   inv1$ i5(not_in4, IN[4]);
   inv1$ i6(not_in5, IN[5]);
   inv1$ i7(not_in6, IN[6]);
   inv1$ i8(not_in7, IN[7]);
   inv1$ i9(not_in8, IN[8]);
   inv1$ i10(not_in9, IN[9]);

   nor22 none(noneset, {min1, min2, min3, min4, min5, min6, min7, min8, min9, min10, min
11, min12,
        min13, min14, min15, min16, min17, min18, min19, min20, min21, min22});


   and10 a1(min1, IN[0], IN[1], IN[2], IN[3], IN[4], IN[5], IN[6], IN[7], IN[8], IN[9]);
   and10 a2(min2, not_in0, IN[1], IN[2], IN[3], not_in4, IN[5], not_in6, not_in7, IN[8],
 IN[9]);
   and10 a3(min3, not_in0, IN[1], not_in2, IN[3], not_in4, IN[5], not_in6, not_in7, IN[8
], IN[9]);
   and10 a4(min4, IN[0], IN[1], not_in2, not_in3, not_in4, IN[5], IN[6], not_in7, not_in
8, not_in9);
   and10 a5(min5, IN[0], not_in1, not_in2, not_in3, not_in4, IN[5], IN[6], not_in7, not_
in8, not_in9);

   or6 o1(OUT[0], min1, min2, min3, min4, min5, noneset);

   and9 a6(min6, not_in1, not_in2, not_in3, not_in4, not_in5, IN[6], not_in7, not_in
8, not_in9);
   and10 a7(min7, not_in0, IN[1], IN[2], not_in3, IN[4], IN[5], IN[6], IN[7], not_in
8, not_in9);
   and10 a8(min8, not_in0, IN[1], IN[2], IN[3], not_in4, IN[5], IN[6], IN[7], not_in
8, not_in9);
   and10 a9(min9, not_in0, IN[1], IN[2], not_in3, not_in4, IN[5], IN[6], IN[7], not_
in8, not_in9);
   and10 a10(min10, not_in0, IN[1], IN[2], not_in3, not_in4, IN[5], IN[6], IN[7], no
t_in8, not_in9);
   and10 a11(min11, not_in0, not_in1, not_in2, not_in3, not_in4, IN[5], IN[6], IN[7]
, not_in8, not_in9);
   and10 a12(min12, not_in0, not_in1, not_in2, not_in3, not_in4, IN[5], IN[6], not_i
n7, not_in8, not_in9);
   and10 a13(min13, not_in0, not_in1, not_in2, not_in3, IN[4], not_in5, not_in6, not
_in7, not_in8, not_in9);
   and10 a14(min14, not_in0, not_in1, not_in2, not_in3, not_in4, not_in5, not_in6, n
ot_in7, not_in8, not_in9);

   or10 o2(OUT[1], min6, min7, min8, min9, min10, min11, min12, min13, min14, nonese
t);

   and9 a15(min15, not_in1, IN[2], not_in3, not_in4, not_in5, IN[6], not_in7, IN[8],
 not_in9);
   and9 a16(min16, not_in1, not_in2, not_in3, not_in4, not_in5, IN[6], not_in7, IN[8
], not_in9);
   and10 a17(min17, not_in0, not_in1, IN[2], IN[3], IN[4], IN[5], not_in6, IN[7], IN
[8], IN[9]);
   and10 a18(min18, not_in0, not_in1, IN[2], not_in3, IN[4], IN[5], not_in6, IN[7],
IN[8], IN[9]);

   or10 o3(OUT[2], min14, min15, min1, min16, min17, min8, min12, min13, min18, none
set);

   and10 a19(min19, not_in0, not_in1, not_in2, IN[3], IN[4], IN[5], not_in6, IN[7],
IN[8], IN[9]);
   and10 a20(min20, not_in0, not_in1, IN[2], not_in3, not_in4, IN[5], not_in6, IN[7]
, IN[8], IN[9]);

   or11 o4(OUT[3], min16, min15, min19, min20, min2, min3, min7, min9, min13, min12,
 noneset);

   and10 a21(min21, not_in0, not_in1, not_in2, not_in3, IN[4], IN[5], not_in6, IN[7]
, IN[8], IN[9]);
   and10 a22(min22, not_in0, not_in1, not_in2, not_in3, not_in4, IN[5], not_in6, IN[
7], IN[8], IN[9]);


   or11 o5(OUT[4], min14, min16, min21, min20, min3, min9, min17, min4, min11, min12
, noneset);
endmodule // decoder22_5_behave


// 1-bit edge-triggered D flip-flop with write-enable
module dffh (CLK, D, Q, QBAR, CLR, PRE, WE);

   input  CLK, CLR, PRE, WE;
   input  D;
   output Q, QBAR;

   wire   w0;

   mux2$ m0 (w0, Q, D, WE);
   dff$ d0 (CLK, w0, Q, QBAR, CLR, PRE);
```

```
endmodule // dffh


// 4-bit comparator
module comp_4(OUT, IN1, IN2);
   output      OUT;
   input [0:3] IN1, IN2;

   xor2$ n0(temp_OUT0, IN1[0], IN2[0]);
   xor2$ n1(temp_OUT1, IN1[1], IN2[1]);
   xor2$ n2(temp_OUT2, IN1[2], IN2[2]);
   xor2$ n3(temp_OUT3, IN1[3], IN2[3]);
   nor4$ n5(OUT, temp_OUT0, temp_OUT1, temp_OUT2, temp_OUT3);
endmodule // comp_4


// 5-bit comparator
module comp_5(OUT, IN1, IN2);
   output      OUT;
   input [0:4] IN1, IN2;

   xor2$ n0(temp_OUT0, IN1[0], IN2[0]);
   xor2$ n1(temp_OUT1, IN1[1], IN2[1]);
   xor2$ n2(temp_OUT2, IN1[2], IN2[2]);
   xor2$ n3(temp_OUT3, IN1[3], IN2[3]);
   xor2$ n4(temp_OUT4, IN1[4], IN2[4]);

   or4$ n5(temp_OUT5, temp_OUT0, temp_OUT1, temp_OUT2, temp_OUT3);
   nor2$ n6(OUT, temp_OUT4, temp_OUT5);
endmodule // comp_5


// 6-bit comparator
module comp_6(OUT, IN1, IN2);
   output      OUT;
   input [0:5] IN1, IN2;

   xor2$ n0(temp_OUT0, IN1[0], IN2[0]);
   xor2$ n1(temp_OUT1, IN1[1], IN2[1]);
   xor2$ n2(temp_OUT2, IN1[2], IN2[2]);
   xor2$ n3(temp_OUT3, IN1[3], IN2[3]);
   xor2$ n4(temp_OUT4, IN1[4], IN2[4]);
   xor2$ n5(temp_OUT5, IN1[5], IN2[5]);

   or4$ n6(temp_OUT6, temp_OUT0, temp_OUT1, temp_OUT2, temp_OUT3);
   nor3$ n7(OUT, temp_OUT4, temp_OUT5, temp_OUT6);
endmodule // comp_5


// 9-bit comparator
module comp_9(OUT, IN1, IN2);
   output       OUT;
   input [0:8]  IN1, IN2;

   comp_5 c1(temp1, IN1[0:4], IN2[0:4]);
   comp_4 c2(temp2, IN1[5:8], IN2[5:8]);
   and2$ o1(OUT, temp1, temp2);
endmodule // comp_9


// 10-bit comparator
module comp_10(OUT, IN1, IN2);
   output       OUT;
   input [0:9]  IN1, IN2;

   comp_5 c1(temp1, IN1[0:4], IN2[0:4]);
   comp_5 c2(temp2, IN1[5:9], IN2[5:9]);
   and2$ o1(OUT, temp1, temp2);
endmodule // comp_10


// 26-bit comparator
module comp_26(OUT, IN1, IN2);
   output       OUT;
   input [0:25] IN1, IN2;

   xor2$ x0 (temp_OUT0, IN1[0], IN2[0]);
   xor2$ x1 (temp_OUT1, IN1[1], IN2[1]);
   xor2$ x2 (temp_OUT2, IN1[2], IN2[2]);
   xor2$ x3 (temp_OUT3, IN1[3], IN2[3]);
   xor2$ x4 (temp_OUT4, IN1[4], IN2[4]);
   xor2$ x5 (temp_OUT5, IN1[5], IN2[5]);
   xor2$ x6 (temp_OUT6, IN1[6], IN2[6]);
   xor2$ x7 (temp_OUT7, IN1[7], IN2[7]);
   xor2$ x8 (temp_OUT8, IN1[8], IN2[8]);
   xor2$ x9 (temp_OUT9, IN1[9], IN2[9]);
   xor2$ x10 (temp_OUT10, IN1[10], IN2[10]);
   xor2$ x11 (temp_OUT11, IN1[11], IN2[11]);
   xor2$ x12 (temp_OUT12, IN1[12], IN2[12]);
   xor2$ x13 (temp_OUT13, IN1[13], IN2[13]);
   xor2$ x14 (temp_OUT14, IN1[14], IN2[14]);
   xor2$ x15 (temp_OUT15, IN1[15], IN2[15]);
   xor2$ x16 (temp_OUT16, IN1[16], IN2[16]);
   xor2$ x17 (temp_OUT17, IN1[17], IN2[17]);
   xor2$ x18 (temp_OUT18, IN1[18], IN2[18]);
   xor2$ x19 (temp_OUT19, IN1[19], IN2[19]);
   xor2$ x20 (temp_OUT20, IN1[20], IN2[20]);
   xor2$ x21 (temp_OUT21, IN1[21], IN2[21]);
   xor2$ x22 (temp_OUT22, IN1[22], IN2[22]);
   xor2$ x23 (temp_OUT23, IN1[23], IN2[23]);
   xor2$ x24 (temp_OUT24, IN1[24], IN2[24]);
   xor2$ x25 (temp_OUT25, IN1[25], IN2[25]);

   nor4$ n0 (n_OUT0, temp_OUT0, temp_OUT1, temp_OUT2, temp_OUT3);
   nor4$ n1 (n_OUT1, temp_OUT4, temp_OUT5, temp_OUT6, temp_OUT7);
   nor4$ n2 (n_OUT2, temp_OUT8, temp_OUT9, temp_OUT10, temp_OUT11);
   nor4$ n3 (n_OUT3, temp_OUT12, temp_OUT13, temp_OUT14, temp_OUT15);
   nor4$ n4 (n_OUT4, temp_OUT16, temp_OUT17, temp_OUT18, temp_OUT19);
   nor4$ n5 (n_OUT5, temp_OUT20, temp_OUT21, temp_OUT22, temp_OUT23);
   nor2$ n6 (n_OUT6, temp_OUT24, temp_OUT25);

   and4$ a0 (a_OUT0, n_OUT0, n_OUT1, n_OUT2, n_OUT3);
   and3$ a1 (a_OUT1, n_OUT4, n_OUT5, n_OUT6);

   and2$ a3 (OUT, a_OUT0, a_OUT1);
endmodule // comp_26


module comp_40(OUT, IN1, IN2);
   output       OUT;
   input [0:39] IN1, IN2;

   comp_10 c1(temp1, IN1[0:9], IN2[0:9]);
   comp_10 c2(temp2, IN1[10:19], IN2[10:19]);
   comp_10 c3(temp3, IN1[20:29], IN2[20:29]);
   comp_10 c4(temp4, IN1[30:39], IN2[30:39]);
   and4$ o1(OUT, temp1, temp2, temp3, temp4);
endmodule // comp_40
```

```
module decoder32 ( in_s, out_bar );
   input [0:4] in_s;
   output [0:31] out_bar;

   wire [0:4] not_s, s, inv_s;

   // Get inverse signals for select lines
   inv1$ i0(inv_s[0], s[0]);
   inv1$ i1(inv_s[1], s[1]);
   inv1$ i2(inv_s[2], s[2]);
   inv1$ i3(inv_s[3], s[3]);
   inv1$ i4(inv_s[4], s[4]);

   // Buffer for fan-out
   buffer$ bns0 ( not_s[0], inv_s[0] ),
           bns1 ( not_s[1], inv_s[1] ),
           bns2 ( not_s[2], inv_s[2] ),
           bns3 ( not_s[3], inv_s[3] ),
           bns4 ( not_s[4], inv_s[4] ),
           bs0  ( s[0], in_s[0] ),
           bs1  ( s[1], in_s[1] ),
           bs2  ( s[2], in_s[2] ),
           bs3  ( s[3], in_s[3] ),
           bs4  ( s[4], in_s[4] );

   nand5 a0 ( out_bar[0], not_s[0], not_s[1], not_s[2], not_s[3], not_s[4] ), // 00000
         a1 ( out_bar[1], not_s[0], not_s[1], not_s[2], not_s[3], s[4] ),     // 00001
         a2 ( out_bar[2], not_s[0], not_s[1], not_s[2], s[3], not_s[4] ),     // 00010
         a3 ( out_bar[3], not_s[0], not_s[1], not_s[2], s[3], s[4] ),         // 00011
         a4 ( out_bar[4], not_s[0], not_s[1], s[2], not_s[3], not_s[4] ),     // 00100
         a5 ( out_bar[5], not_s[0], not_s[1], s[2], not_s[3], s[4] ),         // 00101
         a6 ( out_bar[6], not_s[0], not_s[1], s[2], s[3], not_s[4] ),         // 00110
         a7 ( out_bar[7], not_s[0], not_s[1], s[2], s[3], s[4] ),             // 00111
         a8 ( out_bar[8], not_s[0], s[1], not_s[2], not_s[3], not_s[4] ),     // 01000
         a9 ( out_bar[9], not_s[0], s[1], not_s[2], not_s[3], s[4] ),         // 01001
         a10 ( out_bar[10], not_s[0], s[1], not_s[2], s[3], not_s[4] ),       // 01010
         a11 ( out_bar[11], not_s[0], s[1], not_s[2], s[3], s[4] ),           // 01011
         a12 ( out_bar[12], not_s[0], s[1], s[2], not_s[3], not_s[4] ),       // 01100
         a13 ( out_bar[13], not_s[0], s[1], s[2], not_s[3], s[4] ),           // 01101
         a14 ( out_bar[14], not_s[0], s[1], s[2], s[3], not_s[4] ),           // 01110
         a15 ( out_bar[15], not_s[0], s[1], s[2], s[3], s[4] ),               // 01111
         a16 ( out_bar[16], s[0], not_s[1], not_s[2], not_s[3], not_s[4] ),   // 10000
         a17 ( out_bar[17], s[0], not_s[1], not_s[2], not_s[3], s[4] ),       // 10001
         a18 ( out_bar[18], s[0], not_s[1], not_s[2], s[3], not_s[4] ),       // 10010
         a19 ( out_bar[19], s[0], not_s[1], not_s[2], s[3], s[4] ),           // 10011
         a20 ( out_bar[20], s[0], not_s[1], s[2], not_s[3], not_s[4] ),       // 10100
         a21 ( out_bar[21], s[0], not_s[1], s[2], not_s[3], s[4] ),           // 10101
         a22 ( out_bar[22], s[0], not_s[1], s[2], s[3], not_s[4] ),           // 10110
         a23 ( out_bar[23], s[0], not_s[1], s[2], s[3], s[4] ),               // 10111
         a24 ( out_bar[24], s[0], s[1], not_s[2], not_s[3], not_s[4] ),       // 11000
         a25 ( out_bar[25], s[0], s[1], not_s[2], not_s[3], s[4] ),           // 11001
         a26 ( out_bar[26], s[0], s[1], not_s[2], s[3], not_s[4] ),           // 11010
         a27 ( out_bar[27], s[0], s[1], not_s[2], s[3], s[4] ),               // 11011
         a28 ( out_bar[28], s[0], s[1], s[2], not_s[3], not_s[4] ),           // 11100
         a29 ( out_bar[29], s[0], s[1], s[2], not_s[3], s[4] ),               // 11101
         a30 ( out_bar[30], s[0], s[1], s[2], s[3], not_s[4] ),               // 11110
         a31 ( out_bar[31], s[0], s[1], s[2], s[3], s[4] );                   // 11111
endmodule

module nand5 ( out, in0, in1, in2, in3, in4 );
   input in0, in1, in2, in3, in4;
   output out;

   and3$ a0 (a0out, in0, in1, in2);
   and2$ a1 (a1out, in3, in4);
   nand2$ a2 (out, a0out, a1out);

endmodule // nand5

module tristate2L ( en_bar, in, out );
        input en_bar;
        input [0:1] in;
        output [0:1] out;

        tristateL$      t0 ( en_bar, in[0], out[0] );
        tristateL$      t1 ( en_bar, in[1], out[1] );
endmodule // tristate2L


module decoder32e ( in_s, in_en, out );
   input [0:4] in_s;
   input in_en;
   output [0:31] out;

   wire [0:4] not_s, s, inv_s;

   // Get inverse signals for select lines
   inv1$ i0(inv_s[0], s[0]);
   inv1$ i1(inv_s[1], s[1]);
   inv1$ i2(inv_s[2], s[2]);
   inv1$ i3(inv_s[3], s[3]);
   inv1$ i4(inv_s[4], s[4]);

   // Buffer for fan-out
   buffer$ bns0 ( not_s[0], inv_s[0] ),
           bns1 ( not_s[1], inv_s[1] ),
           bns2 ( not_s[2], inv_s[2] ),
           bns3 ( not_s[3], inv_s[3] ),
           bns4 ( not_s[4], inv_s[4] ),
           bs0  ( s[0], in_s[0] ),
           bs1  ( s[1], in_s[1] ),
           bs2  ( s[2], in_s[2] ),
           bs3  ( s[3], in_s[3] ),
           bs4  ( s[4], in_s[4] ),
           ben  ( en, in_en );

   and6 a0 ( out[0], not_s[0], not_s[1], not_s[2], not_s[3], not_s[4], en ),// 00000
        a1 ( out[1], not_s[0], not_s[1], not_s[2], not_s[3], s[4], en ),    // 00001
        a2 ( out[2], not_s[0], not_s[1], not_s[2], s[3], not_s[4], en ),    // 00010
        a3 ( out[3], not_s[0], not_s[1], not_s[2], s[3], s[4], en ),        // 00011
        a4 ( out[4], not_s[0], not_s[1], s[2], not_s[3], not_s[4], en ),    // 00100
        a5 ( out[5], not_s[0], not_s[1], s[2], not_s[3], s[4], en ),        // 00101
        a6 ( out[6], not_s[0], not_s[1], s[2], s[3], not_s[4], en ),        // 00110
        a7 ( out[7], not_s[0], not_s[1], s[2], s[3], s[4], en ),            // 00111
        a8 ( out[8], not_s[0], s[1], not_s[2], not_s[3], not_s[4], en ),    // 01000
        a9 ( out[9], not_s[0], s[1], not_s[2], not_s[3], s[4], en ),        // 01001
        a10 ( out[10], not_s[0], s[1], not_s[2], s[3], not_s[4], en ),      // 01010
        a11 ( out[11], not_s[0], s[1], not_s[2], s[3], s[4], en ),          // 01011
        a12 ( out[12], not_s[0], s[1], s[2], not_s[3], not_s[4], en ),      // 01100
        a13 ( out[13], not_s[0], s[1], s[2], not_s[3], s[4], en ),          // 01101
        a14 ( out[14], not_s[0], s[1], s[2], s[3], not_s[4], en ),          // 01110
        a15 ( out[15], not_s[0], s[1], s[2], s[3], s[4], en ),              // 01111
        a16 ( out[16], s[0], not_s[1], not_s[2], not_s[3], not_s[4], en ),  // 10000
        a17 ( out[17], s[0], not_s[1], not_s[2], not_s[3], s[4], en ),      // 10001
        a18 ( out[18], s[0], not_s[1], not_s[2], s[3], not_s[4], en ),      // 10010
        a19 ( out[19], s[0], not_s[1], not_s[2], s[3], s[4], en ),          // 10011
        a20 ( out[20], s[0], not_s[1], s[2], not_s[3], not_s[4], en ),      // 10100
        a21 ( out[21], s[0], not_s[1], s[2], not_s[3], s[4], en ),          // 10101
        a22 ( out[22], s[0], not_s[1], s[2], s[3], not_s[4], en ),          // 10110
        a23 ( out[23], s[0], not_s[1], s[2], s[3], s[4], en ),              // 10111
        a24 ( out[24], s[0], s[1], not_s[2], not_s[3], not_s[4], en ),      // 11000
        a25 ( out[25], s[0], s[1], not_s[2], not_s[3], s[4], en ),          // 11001
        a26 ( out[26], s[0], s[1], not_s[2], s[3], not_s[4], en ),          // 11010
        a27 ( out[27], s[0], s[1], not_s[2], s[3], s[4], en ),              // 11011
        a28 ( out[28], s[0], s[1], s[2], not_s[3], not_s[4], en ),          // 11100
        a29 ( out[29], s[0], s[1], s[2], not_s[3], s[4], en ),              // 11101
        a30 ( out[30], s[0], s[1], s[2], s[3], not_s[4], en ),              // 11110
        a31 ( out[31], s[0], s[1], s[2], s[3], s[4], en );                  // 11111

endmodule


module regfile2_32 ( clk, clr, pre, R1, R2, W, We, Data, Out1, Out2 );

    input           clk, clr, pre;
    input   [0:4]   R1, R2, W;
    input           We;
    input   [0:1]   Data;

    output  [0:1]   Out1, Out2;

    wire    [0:1] Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7,
                    Q8, Q9, Q10, Q11, Q12, Q13, Q14, Q15,
                    Q16, Q17, Q18, Q19, Q20, Q21, Q22, Q23,
                    Q24, Q25, Q26, Q27, Q28, Q29, Q30, Q31;

    wire    [0:31]  R1_select_bar, R2_select_bar, W_select;

    //  Decoder to determine which register gets to drive R1 bus
    decoder32   dR1 ( R1, R1_select_bar );

    //  Decoder to determine which register gets to drive R2 bus
    decoder32   dR2 ( R2, R2_select_bar );

    //  Decoder to determine which register is We
    decoder32e  dW  ( W, We, W_select );

    //  32 2-bit registers
    reg2    r0 ( clk, Data, Q0, , clr, pre, W_select[0] ),
            r1 ( clk, Data, Q1, , clr, pre, W_select[1] ),
            r2 ( clk, Data, Q2, , clr, pre, W_select[2] ),
            r3 ( clk, Data, Q3, , clr, pre, W_select[3] ),
            r4 ( clk, Data, Q4, , clr, pre, W_select[4] ),
            r5 ( clk, Data, Q5, , clr, pre, W_select[5] ),
            r6 ( clk, Data, Q6, , clr, pre, W_select[6] ),
            r7 ( clk, Data, Q7, , clr, pre, W_select[7] ),
            r8 ( clk, Data, Q8, , clr, pre, W_select[8] ),
            r9 ( clk, Data, Q9, , clr, pre, W_select[9] ),
            r10 ( clk, Data, Q10, , clr, pre, W_select[10] ),
            r11 ( clk, Data, Q11, , clr, pre, W_select[11] ),
            r12 ( clk, Data, Q12, , clr, pre, W_select[12] ),
            r13 ( clk, Data, Q13, , clr, pre, W_select[13] ),
            r14 ( clk, Data, Q14, , clr, pre, W_select[14] ),
            r15 ( clk, Data, Q15, , clr, pre, W_select[15] ),
            r16 ( clk, Data, Q16, , clr, pre, W_select[16] ),
            r17 ( clk, Data, Q17, , clr, pre, W_select[17] ),
            r18 ( clk, Data, Q18, , clr, pre, W_select[18] ),
            r19 ( clk, Data, Q19, , clr, pre, W_select[19] ),
            r20 ( clk, Data, Q20, , clr, pre, W_select[20] ),
            r21 ( clk, Data, Q21, , clr, pre, W_select[21] ),
            r22 ( clk, Data, Q22, , clr, pre, W_select[22] ),
            r23 ( clk, Data, Q23, , clr, pre, W_select[23] ),
            r24 ( clk, Data, Q24, , clr, pre, W_select[24] ),
            r25 ( clk, Data, Q25, , clr, pre, W_select[25] ),
            r26 ( clk, Data, Q26, , clr, pre, W_select[26] ),
            r27 ( clk, Data, Q27, , clr, pre, W_select[27] ),
            r28 ( clk, Data, Q28, , clr, pre, W_select[28] ),
            r29 ( clk, Data, Q29, , clr, pre, W_select[29] ),
            r30 ( clk, Data, Q30, , clr, pre, W_select[30] ),
            r31 ( clk, Data, Q31, , clr, pre, W_select[31] );

    // Tristates to drive the Out1 bus
    tristate2L  tR1_0 ( R1_select_bar[0], Q0, Out1 ),
                tR1_1 ( R1_select_bar[1], Q1, Out1 ),
                tR1_2 ( R1_select_bar[2], Q2, Out1 ),
                tR1_3 ( R1_select_bar[3], Q3, Out1 ),
                tR1_4 ( R1_select_bar[4], Q4, Out1 ),
                tR1_5 ( R1_select_bar[5], Q5, Out1 ),
                tR1_6 ( R1_select_bar[6], Q6, Out1 ),
                tR1_7 ( R1_select_bar[7], Q7, Out1 ),
                tR1_8 ( R1_select_bar[8], Q8, Out1 ),
                tR1_9 ( R1_select_bar[9], Q9, Out1 ),
                tR1_10 ( R1_select_bar[10], Q10, Out1 ),
                tR1_11 ( R1_select_bar[11], Q11, Out1 ),
                tR1_12 ( R1_select_bar[12], Q12, Out1 ),
                tR1_13 ( R1_select_bar[13], Q13, Out1 ),
                tR1_14 ( R1_select_bar[14], Q14, Out1 ),
                tR1_15 ( R1_select_bar[15], Q15, Out1 ),
                tR1_16 ( R1_select_bar[16], Q16, Out1 ),
                tR1_17 ( R1_select_bar[17], Q17, Out1 ),
                tR1_18 ( R1_select_bar[18], Q18, Out1 ),
                tR1_19 ( R1_select_bar[19], Q19, Out1 ),
                tR1_20 ( R1_select_bar[20], Q20, Out1 ),
                tR1_21 ( R1_select_bar[21], Q21, Out1 ),
                tR1_22 ( R1_select_bar[22], Q22, Out1 ),
                tR1_23 ( R1_select_bar[23], Q23, Out1 ),
                tR1_24 ( R1_select_bar[24], Q24, Out1 ),
                tR1_25 ( R1_select_bar[25], Q25, Out1 ),
                tR1_26 ( R1_select_bar[26], Q26, Out1 ),
                tR1_27 ( R1_select_bar[27], Q27, Out1 ),
                tR1_28 ( R1_select_bar[28], Q28, Out1 ),
                tR1_29 ( R1_select_bar[29], Q29, Out1 ),
                tR1_30 ( R1_select_bar[30], Q30, Out1 ),
                tR1_31 ( R1_select_bar[31], Q31, Out1 );

    // Tristates to drive the Out2 bus
    tristate2L  tR2_0 ( R2_select_bar[0], Q0, Out2 ),
                tR2_1 ( R2_select_bar[1], Q1, Out2 ),
                tR2_2 ( R2_select_bar[2], Q2, Out2 ),
                tR2_3 ( R2_select_bar[3], Q3, Out2 ),
                tR2_4 ( R2_select_bar[4], Q4, Out2 ),
                tR2_5 ( R2_select_bar[5], Q5, Out2 ),
                tR2_6 ( R2_select_bar[6], Q6, Out2 ),
                tR2_7 ( R2_select_bar[7], Q7, Out2 ),
                tR2_8 ( R2_select_bar[8], Q8, Out2 ),
                tR2_9 ( R2_select_bar[9], Q9, Out2 ),
                tR2_10 ( R2_select_bar[10], Q10, Out2 ),
                tR2_11 ( R2_select_bar[11], Q11, Out2 ),
                tR2_12 ( R2_select_bar[12], Q12, Out2 ),
                tR2_13 ( R2_select_bar[13], Q13, Out2 ),
                tR2_14 ( R2_select_bar[14], Q14, Out2 ),
                tR2_15 ( R2_select_bar[15], Q15, Out2 ),
                tR2_16 ( R2_select_bar[16], Q16, Out2 ),
                tR2_17 ( R2_select_bar[17], Q17, Out2 ),
                tR2_18 ( R2_select_bar[18], Q18, Out2 ),
                tR2_19 ( R2_select_bar[19], Q19, Out2 ),
                tR2_20 ( R2_select_bar[20], Q20, Out2 ),
                tR2_21 ( R2_select_bar[21], Q21, Out2 ),
                tR2_22 ( R2_select_bar[22], Q22, Out2 ),
                tR2_23 ( R2_select_bar[23], Q23, Out2 ),
                tR2_24 ( R2_select_bar[24], Q24, Out2 ),
                tR2_25 ( R2_select_bar[25], Q25, Out2 ),
                tR2_26 ( R2_select_bar[26], Q26, Out2 ),
                tR2_27 ( R2_select_bar[27], Q27, Out2 ),
                tR2_28 ( R2_select_bar[28], Q28, Out2 ),
                tR2_29 ( R2_select_bar[29], Q29, Out2 ),
                tR2_30 ( R2_select_bar[30], Q30, Out2 ),
                tR2_31 ( R2_select_bar[31], Q31, Out2 );
endmodule


module regfile32_32 ( clk, clr, pre, R, W, We, Data, Out );

    input           clk, clr, pre;
    input   [0:4]   R, W;
    input           We;
    input   [0:31]  Data;

    output  [0:31]  Out;

    wire    [0:31] Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7,
                  Q8, Q9, Q10, Q11, Q12, Q13, Q14, Q15,
                  Q16, Q17, Q18, Q19, Q20, Q21, Q22, Q23,
                  Q24, Q25, Q26, Q27, Q28, Q29, Q30, Q31;

    wire    [0:31]  R_select_bar, W_select;

    //  Decoder to determine which register gets to drive R bus
    decoder32   dR ( R, R_select_bar );

    //  Decoder to determine which register is We
    decoder32e  dW  ( W, We, W_select );

    //  32 2-bit registers
    reg32e$ r0 ( clk, Data, Q0, , clr, pre, W_select[0] ),
            r1 ( clk, Data, Q1, , clr, pre, W_select[1] ),
            r2 ( clk, Data, Q2, , clr, pre, W_select[2] ),
            r3 ( clk, Data, Q3, , clr, pre, W_select[3] ),
            r4 ( clk, Data, Q4, , clr, pre, W_select[4] ),
            r5 ( clk, Data, Q5, , clr, pre, W_select[5] ),
            r6 ( clk, Data, Q6, , clr, pre, W_select[6] ),
            r7 ( clk, Data, Q7, , clr, pre, W_select[7] ),
            r8 ( clk, Data, Q8, , clr, pre, W_select[8] ),
            r9 ( clk, Data, Q9, , clr, pre, W_select[9] ),
            r10 ( clk, Data, Q10, , clr, pre, W_select[10] ),
            r11 ( clk, Data, Q11, , clr, pre, W_select[11] ),
            r12 ( clk, Data, Q12, , clr, pre, W_select[12] ),
            r13 ( clk, Data, Q13, , clr, pre, W_select[13] ),
            r14 ( clk, Data, Q14, , clr, pre, W_select[14] ),
            r15 ( clk, Data, Q15, , clr, pre, W_select[15] ),
            r16 ( clk, Data, Q16, , clr, pre, W_select[16] ),
            r17 ( clk, Data, Q17, , clr, pre, W_select[17] ),
            r18 ( clk, Data, Q18, , clr, pre, W_select[18] ),
            r19 ( clk, Data, Q19, , clr, pre, W_select[19] ),
            r20 ( clk, Data, Q20, , clr, pre, W_select[20] ),
            r21 ( clk, Data, Q21, , clr, pre, W_select[21] ),
            r22 ( clk, Data, Q22, , clr, pre, W_select[22] ),
            r23 ( clk, Data, Q23, , clr, pre, W_select[23] ),
            r24 ( clk, Data, Q24, , clr, pre, W_select[24] ),
            r25 ( clk, Data, Q25, , clr, pre, W_select[25] ),
            r26 ( clk, Data, Q26, , clr, pre, W_select[26] ),
            r27 ( clk, Data, Q27, , clr, pre, W_select[27] ),
            r28 ( clk, Data, Q28, , clr, pre, W_select[28] ),
            r29 ( clk, Data, Q29, , clr, pre, W_select[29] ),
            r30 ( clk, Data, Q30, , clr, pre, W_select[30] ),
            r31 ( clk, Data, Q31, , clr, pre, W_select[31] );

    // Tristates to drive the Out bus
    tristate32L tR_0 ( R_select_bar[0], Q0, Out ),
                tR_1 ( R_select_bar[1], Q1, Out ),
                tR_2 ( R_select_bar[2], Q2, Out ),
                tR_3 ( R_select_bar[3], Q3, Out ),
                tR_4 ( R_select_bar[4], Q4, Out ),
                tR_5 ( R_select_bar[5], Q5, Out ),
                tR_6 ( R_select_bar[6], Q6, Out ),
                tR_7 ( R_select_bar[7], Q7, Out ),
                tR_8 ( R_select_bar[8], Q8, Out ),
                tR_9 ( R_select_bar[9], Q9, Out ),
                tR_10 ( R_select_bar[10], Q10, Out ),
                tR_11 ( R_select_bar[11], Q11, Out ),
                tR_12 ( R_select_bar[12], Q12, Out ),
                tR_13 ( R_select_bar[13], Q13, Out ),
                tR_14 ( R_select_bar[14], Q14, Out ),
                tR_15 ( R_select_bar[15], Q15, Out ),
                tR_16 ( R_select_bar[16], Q16, Out ),
                tR_17 ( R_select_bar[17], Q17, Out ),
                tR_18 ( R_select_bar[18], Q18, Out ),
                tR_19 ( R_select_bar[19], Q19, Out ),
                tR_20 ( R_select_bar[20], Q20, Out ),
                tR_21 ( R_select_bar[21], Q21, Out ),
                tR_22 ( R_select_bar[22], Q22, Out ),
                tR_23 ( R_select_bar[23], Q23, Out ),
                tR_24 ( R_select_bar[24], Q24, Out ),
                tR_25 ( R_select_bar[25], Q25, Out ),
                tR_26 ( R_select_bar[26], Q26, Out ),
                tR_27 ( R_select_bar[27], Q27, Out ),
                tR_28 ( R_select_bar[28], Q28, Out ),
                tR_29 ( R_select_bar[29], Q29, Out ),
                tR_30 ( R_select_bar[30], Q30, Out ),
                tR_31 ( R_select_bar[31], Q31, Out );


 endmodule


module and6 ( out, in0, in1, in2, in3, in4, in5 );
   input in0, in1, in2, in3, in4, in5;
   output out;

   and3$ a0 (a0out, in0, in1, in2);
   and3$ a1 (a1out, in3, in4, in5);
   and2$ a2 (out, a0out, a1out);

endmodule
```