Extending the Pipelined MIPS Verilog Model

ECE 472 Winter 2010
Ashtyn Moehlenhoff
Torben Rasmussen

Introduction

Pipelining the MIPS processor generates a speed-up by running multiple operations in parallel, instead of waiting for one operation to complete before starting the next. The basic model of the pipelined processor design (given to us) does not allow the jump instruction, nor does it perform data hazard detection or stalling. In order to fully exploit the benefits of the pipelined processor, three things must be added: 1) a jump instruction, 2) data hazard detection and forwarding, and 3) stalling.

Jump Instruction

The simple pipelined processor given to us does not allow jump instructions. Jump instructions are useful in order to move forward/backwards in the program. To add jump, we had to add a jump control signal and add to the control_pipeline.v. We also introduced a new MUX to select either pc+4 or the immediate jump location.

Because the processor is now pipelined, a nop must be inserted by the compiler after every jump instruction to ensure that when the jump is taken, no unwanted instructions are put into the pipeline.

Data Hazard Detection and Forwarding

Because the pipelined processor attempts to run multiple instructions in parallel, the processor must be able to detect dependencies between instructions and handle the dependency. The dependence of one instruction on an earlier instruction that is still in the pipeline is called a data hazard. An example of a data hazard is:

```
add $1, $0, $0;     IF    ID    EX    MEM   WB
add $2, $1, $1;           IF    ID    EX    MEM   WB
```

In this example you can see that the second instruction depends on the first. The register $1 will not be written back until after the fifth stage of the pipeline, but the second instruction requires the value of $1 for its EX stage. In order to solve this problem, we can forward the result of the EX stage from the first instruction to the input of the EX stage for the second instruction.

To detect data hazards, logic must be written to test whether an instruction depends on a previous instruction that is still in the pipeline:

```
always @(*)
begin
ForwardA = 2'b00;
ForwardB = 2'b00;

// EX Hazard
if(MEM_RegWrite && (MEM_RegRd != 0) && (MEM_RegRd == EX_rs))
       ForwardA = 2'b10;

if(MEM_RegWrite && (MEM_RegRd != 0) && (MEM_RegRd == EX_rt))
       ForwardB = 2'b10;

  //MEM Hazard
if(WB_RegWrite && (WB_RegRd != 0) && !(MEM_RegWrite && (MEM_RegRd !=0)
&& (MEM_RegRd != EX_rs) && (WB_RegRd == EX_rs)))
       ForwardA = 2'b01;

if(WB_RegWrite && (WB_RegRd != 0) && !(MEM_RegWrite && (MEM_RegRd !=0)
&& (MEM_RegRd != EX_rt) && (WB_RegRd == EX_rt)))
       ForwardB = 2'b01;
end
```

To forward the values, a MUX must be inserted before the EX stage to determine what to use for ALU_a and ALU_b. Here is a table of the control signal for the forwarding MUX and the output:

MUX control (ForwardA/ForwardB)	Output of MUX
00	From register file
01	From previous instruction's EX stage
10	From previous instruction's MEM stage

The two 4-to-1 MUX's we inserted are:

```
mux4 FWAMUX(ForwardA, EX_rd1, WB_wd, MEM_ALUOut, 32'd0, EX_fw_a_out);
mux4 FWBMUX(ForwardB, EX_rd2, WB_wd, MEM_ALUOut, 32'd0, EX_fw_b_out);
```

Stalling

If an instruction depends on a previous load word instruction, the instruction must wait for the word to be loaded from memory before it can continue:

```
lw   $1, 0($4)            IF    ID    EX    MEM   WB
add $2, $1, $1;           IF    ID    -     -     -
                                      EX    MEM   WB
```

Logic to detect load-use hazard:

```
if((EX_MemRead == 1'b1) && ((EX_rt == ID_rs) || (EX_rt == ID_rt))) Stall <= 1;
```

If a load-use hazard is detected, we insert a nop by setting the control signals of the EX stage to 0 and make sure the PC isn't incremented:

```
always @(posedge clk)
begin
       if((EX_MemRead == 1'b1) && ((EX_rt == ID_rs) || (EX_rt == ID_rt)))
       begin
                      Stall          <= 1;
                      EX_ALUOp       <= 0;
                      EX_ALUSrc   <= 0;
                      EX_Branch   <= 0;
                      EX_MemRead  <= 0;
                      EX_MemWrite <= 0;
                      EX_RegWrite <= 0;
                      EX_MemtoReg <= 0;
                      add_increment = 32'd0;
       end
       else
       begin
                      Stall          <= 0;
                      add_increment = 32'd4;
       end
  end
```

To make sure the PC doesn't increment, we use the variable "add_increment": set to 4 for no stall; set to 0 whenever the pipeline needs stalled. The "add_increment" value was then used as the input to the PC add unit:

```
add32          IF_PCADD(IF_pc, add_increment, IF_pc4);
```

We added to the IF/ID pipeline register to stall the instruction and PC in the case of a load-use hazard:

```
else if(Stall)
begin
        ID_instr <= ID_instr;
        ID_pc4   <= ID_pc4;
end
```

Testing

In order to test the code we simulated the following sequence of instructions (in rom32.v):

```
5'd0 : data_out = { 6'd35, 5'd0, 5'd1, 16'd4 };             // lw $1, 4($0)        r1=1
5'd1 : data_out = { 6'd35, 5'd0, 5'd2, 16'd4 };             // lw $2, 4($0)        r2=1
5'd2 : data_out = { 6'd0,  5'd1, 5'd2, 5'd3, 5'd0, 6'd32 }; // add $3, $1, $2  r3=r1+r2
5'd3 : data_out = { 6'd0,  5'd3, 5'd3, 5'd4, 5'd0, 6'd32 }; // add $4, $3, $3  r3=r1+r2
5'd4 : data_out = { 6'd2,  26'd0 };                         // j 0; restart loop
5'd5 : data_out = { 32'd0 };                                // nop
```

The waveform (figure 1) shows a scenario where both types of hazards are encountered and a jump instruction is executed. The pipeline stalls on the first add instruction to wait for the word to be loaded from memory. You can see in our waveform that the "Stall" control signal is set high and the entire pipeline is stalled for one cycle. In the next cycle, the add can execute with the forwarded values from the EX and MEM stages. The second add is dependent on the results from the first add, so the values are forwarded from the output of the previous instruction's EX stage. Finally, the jump instruction is executed and the program restarts.

Synthesis and Place & Route

Synthesis and Place & Route are the next steps (after design) necessary to tapeout a chip. We used the program Design Vision to synthesize our MIPS pipelined processor Verilog code. This schematic can be seen in Figure 2.

The program Encounter was used to route our design. The hardware layout can be seen in Figure 3.

The area, power and timing reports are as follow:

```
****************************************
Report : area
Design : mips_core
Version: B-2008.09-SP4
Date   : Tue Dec  7 12:42:13 2010
****************************************

Library(s) Used:

    tt_1v8_25c (File: /nfs/spectre/u8/vlsi/UMC180/tt_1v8_25c.db)

Number of ports:              267
Number of nets:               827
Number of cells:               19
Number of references:          19

Combinational area:       120711.730838
Noncombinational area:    97137.533005
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:          217849.263843
Total area:                 undefined

***** End Of Report *****
```

```
****************************************
Report : power
        -analysis_effort low
Design : mips_core
Version: B-2008.09-SP4
Date   : Tue Dec  7 12:40:38 2010
****************************************


Library(s) Used:

    tt_1v8_25c (File: /nfs/spectre/u8/vlsi/UMC180/tt_1v8_25c.db)


Operating Conditions: tt_1v8_25c   Library: tt_1v8_25c
Wire Load Model Mode: top


Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   9.4551 mW   (95%)
  Net Switching Power  = 451.2024 uW    (5%)
                         ---------
Total Dynamic Power    =   9.9063 mW  (100%)

Cell Leakage Power     = 726.4437 nW

***** End Of Report *****


****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
        -sort_by group
Design : mips_core
```

```
Version: B-2008.09-SP4
Date   : Tue Dec  7 12:46:37 2010
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: tt_1v8_25c   Library: tt_1v8_25c
Wire Load Model Mode: top

  Startpoint: rnd_pass1/r5_o_reg_3_
              (rising edge-triggered flip-flop)
  Endpoint: wr_en_o[2] (output port)
  Path Group: (none)
  Path Type: max

  Point                                                    Incr       Path
  -----------------------------------------------------------------------------
  rnd_pass1/r5_o_reg_3_/CK (DFFHQX1)                       0.00 #     0.00 r
  rnd_pass1/r5_o_reg_3_/Q (DFFHQX1)                        0.36       0.36 r
  rnd_pass1/r5_o[3] (r5_reg_2)                             0.00       0.36 r
  iforward/fw_alu_rn[3] (forward)                          0.00       0.36 r
  iforward/fw_alu_rt/alu_wr_rn[3] (forward_node_3)         0.00       0.36 r
  iforward/fw_alu_rt/U17/Y (XOR2X1)                        0.20       0.56 r
  iforward/fw_alu_rt/U15/Y (NOR3X1)                        0.05       0.61 f
  iforward/fw_alu_rt/U14/Y (AND4X2)                        0.20       0.81 f
  iforward/fw_alu_rt/U2/Y (NOR2X1)                         0.28       1.09 r
  iforward/fw_alu_rt/mux_fw[1] (forward_node_3)            0.00       1.09 r
  iforward/alu_rt_fw[1] (forward)                          0.00       1.09 r
  iexec_stage/muxb_fw_ctl[1] (exec_stage)                  0.00       1.09 r
  iexec_stage/i_alu_muxb/fw_ctl[1] (alu_muxb)              0.00       1.09 r
  iexec_stage/i_alu_muxb/U103/Y (XOR2X1)                   0.19       1.28 f
  iexec_stage/i_alu_muxb/U102/Y (NAND3BX1)                 0.06       1.34 r
  iexec_stage/i_alu_muxb/U101/Y (MX2X1)                    0.67       2.01 r
  iexec_stage/i_alu_muxb/U100/Y (AOI22X1)                  0.09       2.09 f
  iexec_stage/i_alu_muxb/U95/Y (NAND2X1)                   0.46       2.56 r
  iexec_stage/i_alu_muxb/b_o[0] (alu_muxb)                 0.00       2.56 r
  iexec_stage/MIPS_alu/b[0] (mips_alu)                     0.00       2.56 r
  iexec_stage/MIPS_alu/mips_alu/b[0] (alu)                 0.00       2.56 r
  iexec_stage/MIPS_alu/mips_alu/r92/B[0] (alu_DW01_addsub_0)
                                                           0.00       2.56 r
  iexec_stage/MIPS_alu/mips_alu/r92/U1/Y (XOR2XL)          0.25       2.80 f
  iexec_stage/MIPS_alu/mips_alu/r92/U1_0/CO (ADDFXL)       0.69       3.50 f
  iexec_stage/MIPS_alu/mips_alu/r92/U1_1/CO (ADDFX2)       0.20       3.70 f
  iexec_stage/MIPS_alu/mips_alu/r92/U1_2/CO (ADDFX2)       0.19       3.89 f
```

```
iexec_stage/MIPS_alu/mips_alu/r92/U1_3/CO (ADDFX2)      0.19       4.08 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_4/CO (ADDFX2)      0.19       4.27 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_5/CO (ADDFX2)      0.19       4.46 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_6/CO (ADDFX2)      0.19       4.65 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_7/CO (ADDFX2)      0.19       4.84 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_8/CO (ADDFX2)      0.19       5.02 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_9/CO (ADDFX2)      0.19       5.21 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_10/CO (ADDFX2)     0.19       5.40 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_11/CO (ADDFX2)     0.19       5.59 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_12/CO (ADDFX2)     0.19       5.78 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_13/CO (ADDFX2)     0.19       5.97 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_14/CO (ADDFX2)     0.19       6.16 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_15/CO (ADDFX2)     0.19       6.35 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_16/CO (ADDFX2)     0.19       6.54 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_17/CO (ADDFX2)     0.19       6.73 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_18/CO (ADDFX2)     0.19       6.92 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_19/CO (ADDFX2)     0.19       7.11 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_20/CO (ADDFX2)     0.19       7.30 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_21/CO (ADDFX2)     0.19       7.49 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_22/CO (ADDFX2)     0.19       7.68 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_23/CO (ADDFX2)     0.19       7.87 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_24/CO (ADDFX2)     0.19       8.06 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_25/CO (ADDFX2)     0.19       8.24 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_26/CO (ADDFX2)     0.19       8.43 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_27/CO (ADDFX2)     0.19       8.62 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_28/CO (ADDFX2)     0.19       8.81 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_29/CO (ADDFX2)     0.19       9.00 f
iexec_stage/MIPS_alu/mips_alu/r92/U1_30/S (ADDFX2)      0.21       9.21 r
iexec_stage/MIPS_alu/mips_alu/r92/SUM[30] (alu_DW01_addsub_0)
                                                        0.00       9.21 r
iexec_stage/MIPS_alu/mips_alu/U310/Y (CLKINVX1)         0.06       9.27 f
iexec_stage/MIPS_alu/mips_alu/U309/Y (OAI22X1)          0.14       9.41 r
iexec_stage/MIPS_alu/mips_alu/r86/B[30] (alu_DW01_add_0)
                                                        0.00       9.41 r
iexec_stage/MIPS_alu/mips_alu/r86/U1_30/CO (ADDFX2)     0.35       9.76 r
iexec_stage/MIPS_alu/mips_alu/r86/U1_31/CO (ADDFX2)     0.18       9.94 r
iexec_stage/MIPS_alu/mips_alu/r86/U1_32/Y (XOR3X2)      0.12      10.06 f
iexec_stage/MIPS_alu/mips_alu/r86/SUM[32] (alu_DW01_add_0)
                                                        0.00      10.06 f
iexec_stage/MIPS_alu/mips_alu/U286/Y (AOI221XL)         0.19      10.25 r
iexec_stage/MIPS_alu/mips_alu/U197/Y (NAND2X1)          0.04      10.29 f
iexec_stage/MIPS_alu/mips_alu/alu_out[0] (alu)          0.00      10.29 f
iexec_stage/MIPS_alu/U32/Y (OR2X1)                      0.19      10.48 f
iexec_stage/MIPS_alu/c[0] (mips_alu)                    0.00      10.48 f
```

```
  iexec_stage/alu_ur_o[0] (exec_stage)                    0.00      10.48 f
  MEM_CTL/dmem_addr_i[0] (mem_module)                     0.00      10.48 f
  MEM_CTL/i_mem_addr_ctl/addr_i[0] (mem_addr_ctl)         0.00      10.48 f
  MEM_CTL/i_mem_addr_ctl/U10/Y (MXI2X1)                   0.11      10.59 f
  MEM_CTL/i_mem_addr_ctl/U4/Y (OAI2BB1X1)                 0.12      10.71 f
  MEM_CTL/i_mem_addr_ctl/wr_en[2] (mem_addr_ctl)          0.00      10.71 f
  MEM_CTL/Zz_wr_en[2] (mem_module)                        0.00      10.71 f
  wr_en_o[2] (out)                                        0.00      10.71 f
  data arrival time                                                 10.71
  ---------------------------------------------------------------------------
  (Path is unconstrained)

***** End Of Report *****
```

Conclusion

The group spent approximately 20 hours total on this project. The most valuable part was learning the intricacies of hazard detection on a pipelined MIPS processor. The least valuable part was creating the hardware design for the part, simple because it was pre-scripted for us.

In the future it may be valuable to limit the amount of starter code that teams are given. It would promote creativity, and would waste less time learning how to interpret others' code.

Appendix

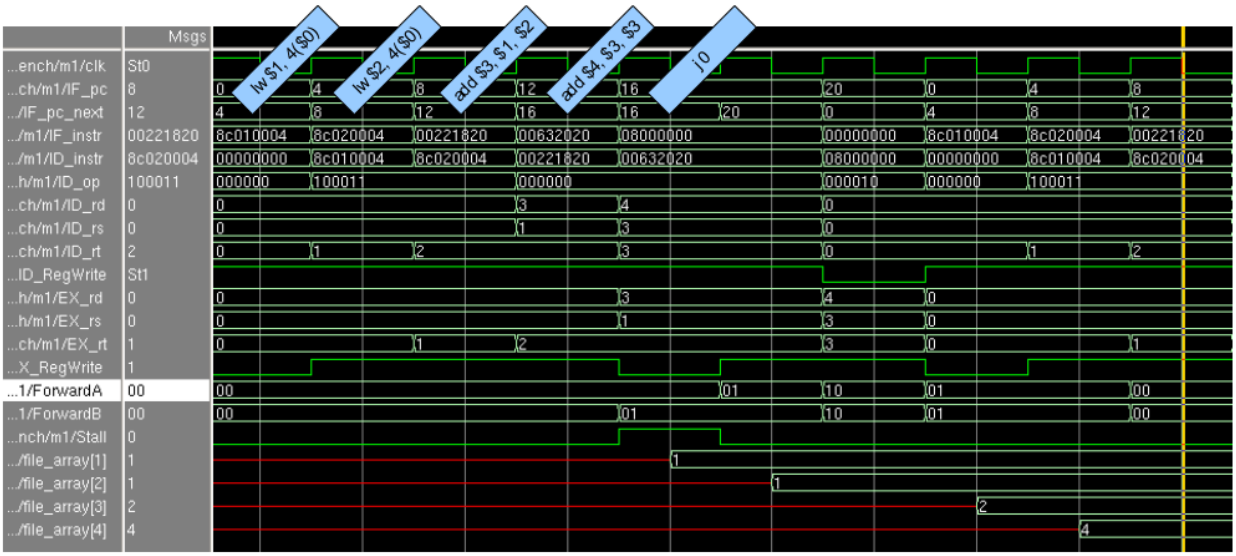

Figure 1

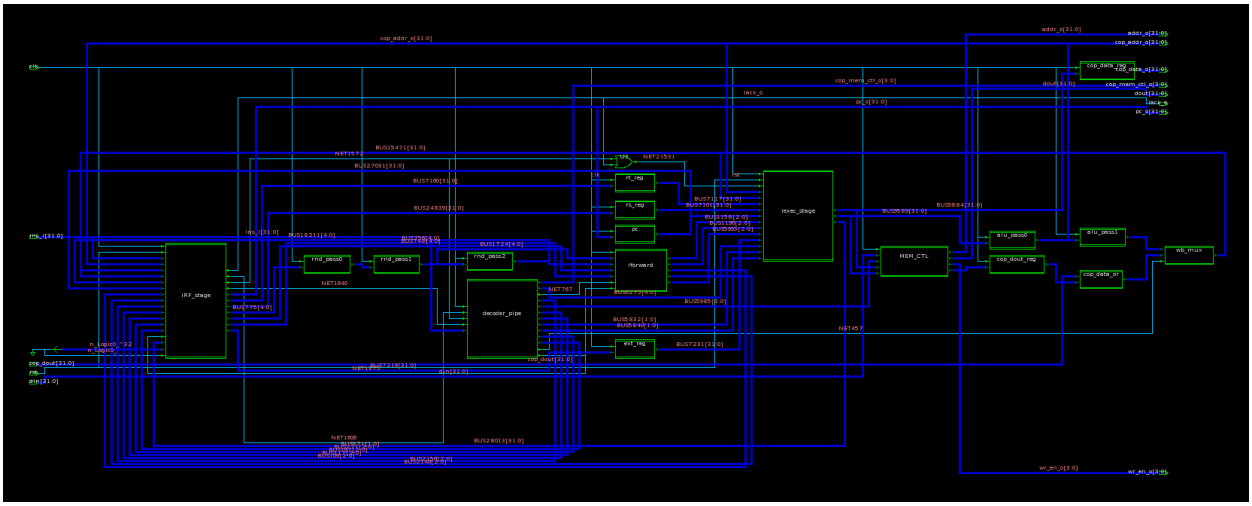

Figure 2

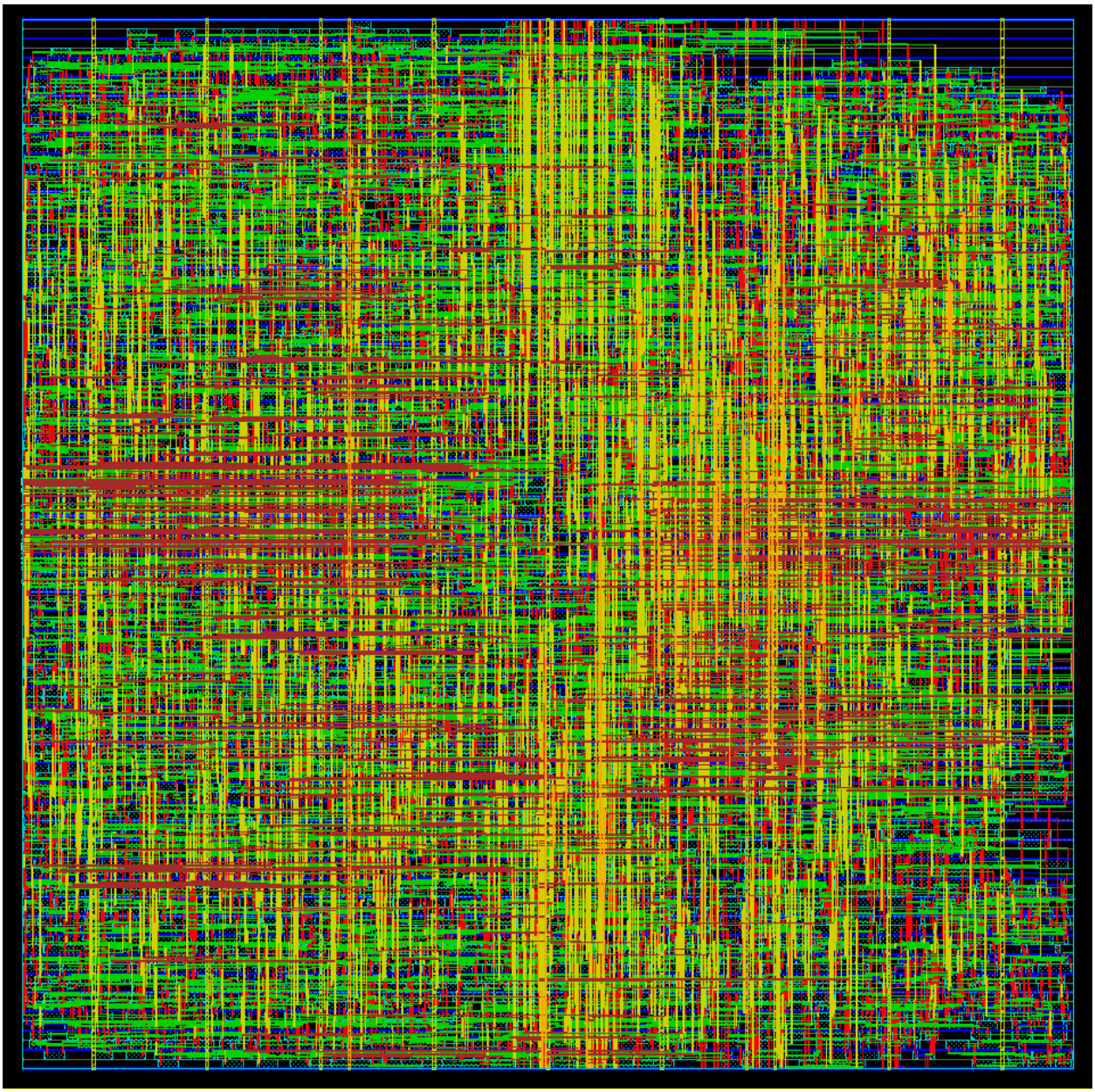

Figure 3

```
module mips_pipeline(clk, reset);
input clk, reset;

    // *************************************************************************
    //                            Signal Declarations                          
    // *************************************************************************

    // IF Signal Declarations

    wire [31:0] IF_instr, IF_pc, IF_pc_next, IF_pc4;

    // ID Signal Declarations

    reg [31:0] ID_instr, ID_pc4;  // pipeline register values from EX

    wire [5:0] ID_op, ID_funct;
    wire [4:0] ID_rs, ID_rt, ID_rd;
    wire [15:0] ID_immed;
    wire [25:0] ID_jump_offset;
    wire [31:0] ID_extend, ID_rd1, ID_rd2, ID_jump_in, ID_jump_offset_shifted;
    wire [31:0] stall_mux_out;

    assign ID_op = ID_instr[31:26];
    assign ID_rs = ID_instr[25:21];
    assign ID_rt = ID_instr[20:16];
    assign ID_rd = ID_instr[15:11];
    assign ID_immed = ID_instr[15:0];
    assign ID_jump_offset = ID_instr[25:0];

    wire ID_RegWrite, ID_Branch, ID_RegDst, ID_MemtoReg,  // ID Control signals
         ID_MemRead, ID_MemWrite, ID_ALUSrc, ID_Jump;
    wire [1:0] ID_ALUOp;

    // EX Signals

    reg  [31:0] EX_pc4, EX_extend, EX_rd1, EX_rd2;
    wire [31:0]  EX_offset, EX_btgt, EX_alub, EX_ALUOut;
    reg  [4:0]  EX_rt, EX_rd, EX_rs;
    wire [4:0]  EX_RegRd;
    wire [5:0] EX_funct;

    reg  EX_RegWrite, EX_Branch, EX_RegDst, EX_MemtoReg,  // EX Control Signals
         EX_MemRead, EX_MemWrite, EX_ALUSrc;

    reg  Stall = 0;
    reg  [31:0] add_increment = 32'd4;

    wire EX_Zero;

    reg  [1:0] EX_ALUOp;
    wire [2:0] EX_Operation;

   // MEM Signals

    wire MEM_PCSrc;

    reg  MEM_RegWrite, MEM_Branch, MEM_MemtoReg,
         MEM_MemRead, MEM_MemWrite, MEM_Zero;

    reg  [31:0] MEM_btgt, MEM_ALUOut, MEM_rd2;
    wire [31:0] MEM_memout;
    reg  [5:0] MEM_RegRd;

    // WB Signals

    reg WB_RegWrite, WB_MemtoReg;  // WB Control Signals

    reg  [31:0] WB_memout, WB_ALUOut;
    wire [31:0] WB_wd;
```



```
    reg  [4:0] WB_RegRd;

    // Forwarding Unit
    reg  [1:0] ForwardA, ForwardB;
    wire [31:0] EX_fw_a_out, EX_fw_b_out;


    // *************************************************************************
    //                                IF Stage                                 
    // *************************************************************************

    // IF Hardware

    reg32                IF_PC(clk, reset, IF_pc_next, IF_pc);

    add32                IF_PCADD(IF_pc, add_increment, IF_pc4);

    mux2 #(32)  IF_PCMUX(MEM_PCSrc, IF_pc4, MEM_btgt, ID_jump_in);

    rom32                IMEM(IF_pc, IF_instr);

    // *************************************************************************
    //                              Lab 4 Code                                 
    // *************************************************************************

    assign ID_jump_offset_shifted = ID_jump_offset << 2;

    mux2 #(32) JMPMUX(ID_Jump, ID_jump_in, ID_jump_offset_shifted, IF_pc_next);

    // *************************************************************************
    //                            END Lab 4 Code                               
    // *************************************************************************


    always @(posedge clk)                    // IF/ID Pipeline Register
    begin
        if (reset)
        begin
            ID_instr <= 0;
            ID_pc4   <= 0;
        end
        else if(Stall)
        begin
          ID_instr <= ID_instr;
          ID_pc4   <= ID_pc4;
        end
        else begin
            ID_instr <= IF_instr;
            ID_pc4   <= IF_pc4;
        end
    end

    // *************************************************************************
    //                                ID Stage                                 
    // *************************************************************************

    reg_file    RFILE(clk, WB_RegWrite, ID_rs, ID_rt, WB_RegRd, ID_rd1, ID_rd2,
WB_wd);

    // sign-extender
    assign ID_extend = { {16{ID_immed[15]}}, ID_immed };

    control_pipeline CTL(.opcode(ID_op), .RegDst(ID_RegDst),
                         .ALUSrc(ID_ALUSrc), .MemtoReg(ID_MemtoReg),
                         .RegWrite(ID_RegWrite), .MemRead(ID_MemRead),
                         .MemWrite(ID_MemWrite), .Branch(ID_Branch),
                         .Jump(ID_Jump), .ALUOp(ID_ALUOp));
```

```
    always @(posedge clk)                    // ID/EX Pipeline Register
    begin
        if (reset)
        begin
            EX_RegDst   <= 0;
                  EX_ALUOp    <= 0;
            EX_ALUSrc   <= 0;
            EX_Branch   <= 0;
            EX_MemRead  <= 0;
            EX_MemWrite <= 0;
            EX_RegWrite <= 0;
            EX_MemtoReg <= 0;

            EX_pc4      <= 0;
            EX_rd1      <= 0;
            EX_rd2      <= 0;
            EX_extend   <= 0;
            EX_rt       <= 0;
            EX_rd       <= 0;
            EX_rs       <= 0;
        end
        else if (Stall)
        begin

            EX_rd       <= EX_rd;
            EX_rs       <= EX_rs;
            EX_rt       <= EX_rt;
            EX_RegWrite <= ID_RegWrite;
        end
        else begin
            EX_RegDst   <= ID_RegDst;
            EX_ALUOp    <= ID_ALUOp;
            EX_ALUSrc   <= ID_ALUSrc;
            EX_Branch   <= ID_Branch;
            EX_MemRead  <= ID_MemRead;
            EX_MemWrite <= ID_MemWrite;
            EX_RegWrite <= ID_RegWrite;
            EX_MemtoReg <= ID_MemtoReg;

            EX_pc4      <= ID_pc4;
            EX_rd1      <= ID_rd1;
            EX_rd2      <= ID_rd2;
            EX_extend   <= ID_extend;
            EX_rt       <= ID_rt;
            EX_rd       <= ID_rd;
            EX_rs       <= ID_rs;
        end
    end

    // ***********************************************************************
    //                              EX Stage
    // ***********************************************************************

    assign EX_offset = EX_extend << 2;

    assign EX_funct = EX_extend[5:0];

    add32               EX_BRADD(EX_pc4, EX_offset, EX_btgt);

    mux4 FWAMUX(ForwardA, EX_rd1, WB_wd, MEM_ALUOut, 32'd0, EX_fw_a_out);
    mux4 FWBMUX(ForwardB, EX_rd2, WB_wd, MEM_ALUOut, 32'd0, EX_fw_b_out);

    mux2 #(32)  ALUMUX(EX_ALUSrc, EX_fw_b_out, EX_extend, EX_alub);

    alu                 EX_ALU(EX_Operation, EX_fw_a_out, EX_alub, EX_ALUOut, EX
_Zero);

    mux2 #(5)   EX_RFMUX(EX_RegDst, EX_rt, EX_rd, EX_RegRd);
```



```
    alu_ctl     EX_ALUCTL(EX_ALUOp, EX_funct, EX_Operation);

    always @(posedge clk)                    // EX/MEM Pipeline Register
        begin
        if (reset)
        begin
            MEM_Branch   <= 0;
            MEM_MemRead  <= 0;
            MEM_MemWrite <= 0;
            MEM_RegWrite <= 0;
            MEM_MemtoReg <= 0;
            MEM_Zero     <= 0;

            MEM_btgt     <= 0;
            MEM_ALUOut   <= 0;
            MEM_rd2      <= 0;
            MEM_RegRd    <= 0;
        end
        else begin
            MEM_Branch   <= EX_Branch;
            MEM_MemRead  <= EX_MemRead;
            MEM_MemWrite <= EX_MemWrite;
            MEM_RegWrite <= EX_RegWrite;
            MEM_MemtoReg <= EX_MemtoReg;
            MEM_Zero     <= EX_Zero;

            MEM_btgt     <= EX_btgt;
            MEM_ALUOut   <= EX_ALUOut;
            MEM_rd2      <= EX_rd2;
            MEM_RegRd    <= EX_RegRd;
        end
    end

    // ***********************************************************************
    //                              MEM Stage
    // ***********************************************************************

    mem32               MEM_DMEM(clk, MEM_MemRead, MEM_MemWrite, MEM_ALUOut, MEM
_rd2, MEM_memout);

    and                 MEM_BR_AND(MEM_PCSrc, MEM_Branch, MEM_Zero);

    always @(posedge clk)                // MEM/WB Pipeline Register
    begin
        if (reset)
        begin
            WB_RegWrite <= 0;
            WB_MemtoReg <= 0;
            WB_ALUOut   <= 0;
            WB_memout   <= 0;
            WB_RegRd    <= 0;
        end
        else begin
            WB_RegWrite <= MEM_RegWrite;
            WB_MemtoReg <= MEM_MemtoReg;
            WB_ALUOut   <= MEM_ALUOut;
            WB_memout   <= MEM_memout;
            WB_RegRd    <= MEM_RegRd;
        end
    end

    // ***********************************************************************
    //                              WB Stage
    // ***********************************************************************
    mux2 #(32)  WB_WRMUX(WB_MemtoReg, WB_ALUOut, WB_memout, WB_wd);


    // ***********************************************************************
    //                          Forwarding Unit
```

```
    // ***********************************************************************
    always @(*)
    begin
      ForwardA = 2'b00;
      ForwardB = 2'b00;

      // EX Hazard
      if(MEM_RegWrite && (MEM_RegRd != 0) && (MEM_RegRd == EX_rs))
        ForwardA = 2'b10;

      if(MEM_RegWrite && (MEM_RegRd != 0) && (MEM_RegRd == EX_rt))
        ForwardB = 2'b10;

      //MEM Hazard
      if(WB_RegWrite && (WB_RegRd != 0) && !(MEM_RegWrite && (MEM_RegRd !=0)
      && (MEM_RegRd != EX_rs) && (WB_RegRd == EX_rs)))
        ForwardA = 2'b01;

      if(WB_RegWrite && (WB_RegRd != 0) && !(MEM_RegWrite && (MEM_RegRd !=0)
      && (MEM_RegRd != EX_rt) && (WB_RegRd == EX_rt)))
        ForwardB = 2'b01;
    end

    // ***********************************************************************
    //                            Stalling Unit
    // ***********************************************************************
    always @(posedge clk)
    begin
    if((EX_MemRead == 1'b1) && ((EX_rt == ID_rs) || (EX_rt == ID_rt)))
          begin
            Stall       <= 1;
            EX_ALUOp    <= 0;
            EX_ALUSrc   <= 0;
            EX_Branch   <= 0;
            EX_MemRead  <= 0;
            EX_MemWrite <= 0;
            EX_RegWrite <= 0;
            EX_MemtoReg <= 0;
            add_increment = 32'd0;
          end
        else
          begin
            Stall        <= 0;
            add_increment = 32'd4;
          end
      end
endmodule
```

```
//------------------------------------------------------------------------
// Title         : MIPS Single-Cycle Control Unit
// Project       : ECE 313 - Computer Organization
//------------------------------------------------------------------------
// File          : control_single.v
// Author        : John Nestor  <nestorj@lafayette.edu>
// Organization  : Lafayette College
//
// Created       : November 2002
// Last modified : 7 January 2005
//------------------------------------------------------------------------
// Description :
//   Control unit for the MIPS pipelined  processor implementation described
//   Section 6.3 of "Computer Organization and Design, 3rd ed."
//   by David Patterson & John Hennessey, Morgan Kaufmann, 2004 (COD3e).
//
//   It implements the function specified in Figure 6.25 on p. 401 of COD3e.
//
//------------------------------------------------------------------------

module control_pipeline(opcode, RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, Mem
Write, Branch, Jump, ALUOp);
    input [5:0] opcode;
    output RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump;
    output [1:0] ALUOp;
    reg    RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump;
    reg    [1:0] ALUOp;

    parameter R_FORMAT = 6'd0;
    parameter LW = 6'd35;
    parameter SW = 6'd43;
    parameter BEQ = 6'd4;
    parameter J = 6'd2;

    always @(opcode)
    begin
        case (opcode)
          R_FORMAT :
          begin
              RegDst=1'b1; ALUSrc=1'b0; MemtoReg=1'b0; RegWrite=1'b1; MemRead=1'
b0;
              MemWrite=1'b0; Branch=1'b0; ALUOp = 2'b10; Jump = 1'b0;
          end
          LW :
          begin
              RegDst=1'b0; ALUSrc=1'b1; MemtoReg=1'b1; RegWrite=1'b1; MemRead=1'
b1;
              MemWrite=1'b0; Branch=1'b0; ALUOp = 2'b00; Jump = 1'b0;
          end
          SW :
          begin
              RegDst=1'bx; ALUSrc=1'b1; MemtoReg=1'bx; RegWrite=1'b0; MemRead=1'
b0;
              MemWrite=1'b1; Branch=1'b0; ALUOp = 2'b00; Jump = 1'b0;
          end
          BEQ :
          begin
              RegDst=1'bx; ALUSrc=1'b0; MemtoReg=1'bx; RegWrite=1'b0; MemRead=1'
b0;
              MemWrite=1'b0; Branch=1'b1; ALUOp = 2'b01; Jump = 1'b0;
          end
          J   :
          begin
              RegDst=1'bx; ALUSrc=1'b0; MemtoReg=1'bx; RegWrite=1'b0; MemRead=1'
b0;
              MemWrite=1'b0; Branch=1'b0; ALUOp = 2'bxx; Jump = 1'b1;
          end

          default
```



```
          begin
              $display("control_single unimplemented opcode %d", opcode);
              RegDst=1'b0; ALUSrc=1'b0; MemtoReg=1'b0; RegWrite=1'b0; MemRead=1'
b0;
              MemWrite=1'b0; Branch=1'b0; ALUOp = 2'b00; Jump = 1'b0;
          end
        endcase
    end
endmodule
```

```
//    A simple 32-bit by 32-word read-only memory model
//    ECE 313 Fall 2002


//  11/21/02 - modified to access up to 64 words JN

module rom32(address, data_out);
  input  [31:0] address;
  output [31:0] data_out;
  reg    [31:0] data_out;

  parameter BASE_ADDRESS = 25'd0; // address that applies to this memory

  wire [5:0] mem_offset;
  wire address_select;

  assign mem_offset = address[7:2];  // drop 2 LSBs to get word offset

  assign address_select = (address[31:8] == BASE_ADDRESS);  // address decoding

  always @(address_select or mem_offset)

  begin
    if ((address % 4) != 0) $display($time, " rom32 error: unaligned address %d", address)
;
    if (address_select == 1)
    begin
      case (mem_offset)

          // ADD test:
          /*
          5'd0 : data_out = { 6'd0,  5'd0, 5'd0, 5'd2, 5'd0, 6'd32 };//add $2, $
0, $0;
          5'd1 : data_out = { 6'd0,  5'd2, 5'd2, 5'd3, 5'd0, 6'd32 };//add $3, $
2, $2;
          5'd2 : data_out = { 6'd0,  5'd3, 5'd3, 5'd4, 5'd0, 6'd32 };//add $4, $
3, $3;
          */


          5'd0 : data_out = { 6'd35, 5'd0, 5'd1, 16'd4 };              // lw $1,
 4($0)    r1=1
          5'd1 : data_out = { 6'd35, 5'd0, 5'd2, 16'd4 };              // lw $2,
 4($0)    r2=1
          5'd2 : data_out = { 6'd0,  5'd1, 5'd2, 5'd3, 5'd0, 6'd32 };  // add $3
, $1, $2  r3=r1+r2
          5'd3 : data_out = { 6'd0,  5'd3, 5'd3, 5'd4, 5'd0, 6'd32 };  // add $4
, $3, $3  r3=r1+r2
          5'd4 : data_out = { 6'd2,  26'd0 };                          // j 0; r
estart  loop
          5'd5 : data_out = { 32'd0 };                                 // nop



        // 5'd3 : data_out = { 6'd0,  5'd0, 5'd0, 5'd4, 5'd0, 6'd32 };  //add $
2, $0, $0;
          // 5'd4 : data_out = { 6'd0,  5'd0, 5'd0, 5'd5, 5'd0, 6'd32 };  //add $
2, $0, $0;
          // 5'd5 : data_out = { 6'd0,  5'd0, 5'd0, 5'd6, 5'd0, 6'd32 };  //add $
2, $0, $0;
          // 5'd6 : data_out = { 6'd0,  5'd0, 5'd0, 5'd7, 5'd0, 6'd32 };  //add $
2, $0, $0;
          // 5'd7 : data_out = { 6'd0,  5'd0, 5'd0, 5'd8, 5'd0, 6'd32 };  //add $
2, $0, $0;


          //5'd0 : data_out = { 6'd0,  5'd0, 5'd0, 5'd2, 5'd0, 6'd32 };  // add
$2, $0, $0  r2=0+0
          //5'd0 : data_out = { 6'd35, 5'd0, 5'd2, 16'd4 };              // lw $
```



```
2, 4($0)    r2=1
          //5'd1 : data_out = { 6'd0,  5'd0, 5'd2, 5'd3, 5'd0, 6'd32 };  // add
$3, $0, $2  r3=0+r2
          //5'd2 : data_out = { 32'd0 };                                 // nop
          //5'd3 : data_out = { 32'd0 };                                 // nop
          //5'd4 : data_out = { 32'd0 };                                 // nop
          //5'd5 : data_out = { 6'd2,  26'd0 };                          // j 0;
 restart  loop
          //5'd6 : data_out = { 32'd0 };                                 // nop
          /*
          5'd0 : data_out = { 6'd35, 5'd0, 5'd2, 16'd4 };              // lw $2,
 4($0)    r2=1
          5'd1 : data_out = { 6'd35, 5'd0, 5'd3, 16'd8 };              // lw $3,
 8($0)    r3=2
          5'd2 : data_out = { 6'd35, 5'd0, 5'd4, 16'd20 };             // lw $4,
 20($0)  r4=5
          //5'd3 : data_out = { 6'd2, 26'd0 }; //j 0; restart  loop

          5'd3 : data_out = { 6'd0, 5'd0, 5'd0, 5'd5, 5'd0, 6'd32 };  // add $5,
 $0, $0  r5=0
          5'd4 : data_out = 0; // no-op to stall until add is done
          5'd5 : data_out = 0; // no-op to stall until add is done
          5'd6 : data_out = 0; // no-op to stall until add is done
          5'd7 : data_out = { 6'd0, 5'd5, 5'd2, 5'd5, 5'd0, 6'd32 };  // add $5,
 $5, $1  r5 = r6 + 1
          5'd8 : data_out = 0; // no-op to stall until add is done
          5'd9 : data_out = 0; // no-op to stall until add is done
          5'd10 : data_out = 0; // no-op to stall until add is done

          5'd11 : data_out = { 6'd0, 5'd4, 5'd5, 5'd6, 5'd0, 6'd42 };  // slt $6
, $4, $5  is $5 > 54?
          5'd12 : data_out = 0; // no-op to stall until slt is done
          5'd13 : data_out = 0; // no-op to stall until slt is done
          5'd14 : data_out = 0; // no-op to stall until slt is done

          5'd15 : data_out = { 6'd4, 5'd6, 5'd0, -16'd9 };             // beq $6
, $zero, -9  if not, go back 2
          5'd16 : data_out = 32'b0; // no-op after branch
          5'd17 : data_out = 32'b0; // no-op after branch
          5'd18 : data_out = 32'b0; // no-op after branch
          5'd19 : data_out = { 6'd43, 5'd0, 5'd5, 16'd0 };             // MEM[0]
 = $5
          5'd20 : data_out = { 6'd4, 5'd0, 5'd0, -16'd18 };            // beq $0
, $0, -18 restart loop at word 3
          // add more cases here as desired
          */
          default data_out = 32'hxxxx;

      endcase

      $display($time, " reading data: rom32[%h] => %h", address, data_out);

    end
  end
endmodule
```

```
//------------------------------------------------------------------------
// Title         : Register File (32 32-bit registers)
// Project       : ECE 313 - Computer Organization
//------------------------------------------------------------------------
// File          : reg_file.v
// Author        : John Nestor  <nestorj@lafayette.edu>
// Organization  : Lafayette College
//
// Created       : October 2002
// Last modified : 7 January 2005
//------------------------------------------------------------------------
// Description :
//   Behavioral model of the register file  used in the implementations of the M
IPS
//   processor subset described in Ch. 5-6 of "Computer Organization and Design,
 3rd ed."
//   by David Patterson & John Hennessey, Morgan Kaufmann, 2004 (COD3e).
//
//   It implements the function specified in Fig 5-7 on p. 295 of COD3e.
//
//------------------------------------------------------------------------

module reg_file(clk, RegWrite, RN1, RN2, WN, RD1, RD2, WD);
  input clk;
  input RegWrite;
  input [4:0] RN1, RN2, WN;
  input [31:0] WD;
  output [31:0] RD1, RD2;

  reg [31:0] RD1, RD2;
  reg [31:0] file_array [31:1];

  always @(RN1 or file_array[RN1])
  begin
    if (RN1 == 0) RD1 = 32'd0;
    else RD1 = file_array[RN1];
    $display($time, " reg_file[%d] => %d (Port 1)", RN1, RD1);
  end

  always @(RN2 or file_array[RN2])
  begin
    if (RN2 == 0) RD2 = 32'd0;
    else RD2 = file_array[RN2];
    $display($time, " reg_file[%d] => %d (Port 2)", RN2, RD2);
  end

  always @(posedge clk)
    if (RegWrite && (WN != 0))
    begin
      file_array[WN] <= WD;
      $display($time, " reg_file[%d] <= %d (Write)", WN, WD);
    end
endmodule
```

```
//4-to-1 MUX://
module mux4(sel, a, b, c, d, y);
  input [31:0] a,b,c,d;
  input [1:0] sel;
  output [31:0] y;
  reg [31:0] y;

  always@(*)
  case (sel)
    2'd0:  y=a;
    2'd1:  y=b;
    2'd2:  y=c;
    2'd3:  y=d;
  default: $display("Error in selection");
  endcase
endmodule
```