CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

CS 224 – Spring 2019 – Lab #5

Implementing the MIPS Processor with Pipelined Microarchitecture

---

Preliminary Design Report

Berrak Taşkınsu / 21602054

Section No.: 2

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

b) [13 points] The list of all hazards that can occur in this pipeline. For each hazard, give its type (data or control), its specific name ("compute-use" "load-use", "load-store" "J-type jump", "branch" etc), the pipeline stages that are affected, the solution (forwarding, stalling, flushing, combination of these), and explanation of what, when, how.

A pipeline hazard occurs when an instruction depends on the results of other instructions that haven't been completed yet. If not fixed, hazards cause pipeline to compute the wrong results. There are two type of hazards which are data hazards and control hazards. Data hazards occur when a register source is needed from a later stage of the pipeline before it is written. Control hazards occur when an instruction is fetched without checking if it is the next instruction to be fetched. It is caused by branches and jump instructions.

In this pipeline, both control and data hazards can occur. But all data hazards are handled by the Hazard Unit, according to the given pipeline design.

(1) Data Hazards:

The designed Hazard Unit in this pipeline can handle data hazards "compute-use", "load-use" and "load-store" which are all Read-After-Write hazards.

- **"compute – use":** The Hazard Unit handles "compute-use" by forwarding, since even though the data hasn't been written back to the register file, the result is available in early stages; it is the alu result. Therefore, this alu result will be forwarded to the Execution stage of the current instruction from either the Memory or Write Back stages, in case one of the source registers matches the destination registers of early instructions.

- **"load – use" and "load store":** On the other hand, "load-use" and "load-store" hazards are handled by the combination of forwarding and stalling. Stalling is necessary when lw instruction is present because lw instructruction reads data from Data Memory. So the result is not ready when the following instructions try to use it, and it can not be forwarded since it isn't ready. Therefore, current stage is stalled for all following instructions, in case of a match between the source and previous destination registers. For the next cycle, data is forwarded if a second instruction tries to access the same data.

(2) Control Hazards:

The given design of the Hazard Unit handles data hazards, but control hazards such as "branch" and "J-type jump" hazards may occur.

- **"branch":** Branch hazard can not be handled by tforwarding alone, since it is not only about the decision for a branch but also causes the registers to be overwritten, resulting in a different hazard in Decode Stage. The solution is a combination of stalling, flushing and forwarding.

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

c) [8 points] The logic equations for each signal output by the hazard unit, as a function of the input signals that come to the hazard unit. This hazard unit should handle all the data and control hazards that can occur in your pipeline (listed in b above) so that your pipelined processor computes correctly.

Forwarding Logic to Handle "compute – use" Hazard:

```
if ( ( rsE != 0 ) AND ( rsE == WriteRegM ) AND RegWriteM )
     then ForwardAE = 10
else if ( ( rsE != 0 ) AND ( rsE == WriteRegW ) AND RegWriteM )
     then ForwardAE = 01
else         ForwardAE = 00

if ( ( rtE != 0 ) AND ( rtE == WriteRegM ) AND RegWriteM )
     then ForwardAE = 10
else if ( ( rtE != 0 ) AND ( rtE == WriteRegW ) AND RegWriteM )
     then ForwardAE = 01
else         ForwardAE = 00
```

Stalling Logic to Handle "load – use" and "load – store" Hazards :

```
lwstall = ( ( rsD == rtE ) OR ( rtD == rtE )  ) AND MemtoRegE
StallF = StallD = FlushE = lwstall (*)
```

The following are not included in code for HazardUnit since the gicen design doesn't support them.

Forwarding Logic to Handle Control Hazards:

```
ForwardAD = ( rsD != 0 ) AND ( rsD == WriteRegM ) AND RegWriteM
ForwardBD = ( rtD != 0 ) AND ( rtD == WriteRegM ) AND RegWriteM
```

Stalling Logic to Handle Decode Stage Hazards Caused by Early Branch Pediction:

```
branchstall =
     BranchD AND RegWriteE AND ( ( WriteRegE == rsD ) OR
( WriteRegE == rtD ) )
    OR BranchD AND MemtoRegM AND ( ( WriteRegM == rsD ) OR
( WriteRegM == rtD ) )
StallF = StallD = FlushE = branchstall || lwstall (*)
```

**If branch hazard is handled in the hazard unit, then (**) is used instead of (*).*

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

d) [30 points] You are given a skeleton System Verilog code for your pipelined MIPS processor in the file *PipelinedMIPSProcessorToFillNew.txt*. The places in the code that needs to be modified are shown with comment blocks above them. Fill them and highlight the changes you made in the code in your report. You can use a different text highlight color (do this by hand after getting the printout) for this purpose. You do NOT need to follow the skeleton code point by point. If you think your design is better, you are welcome to try it in your code, as long as your version of the code works, too. Note that this is a design problem and there is no single solution to it but if you feel comfortable with the skeleton code then you can use it.

Modules PipeFtoD, PipeWtoF, mux2, mux4, alu, maindec, aludec, sl2, signext, controller, dmem are not modified. Therefore they are not here. There were no major changes made to the skeleton code but few changes are made, they are highlighted with BLUE, unlike the added code which is in RED.

PipeDtoE:

```
module PipeDtoE(input logic clr, clk, reset, RegWriteD, MemtoRegD, MemWriteD,
            input logic[2:0] AluControlD,
            input logic AluSrcD, RegDstD, BranchD,
            input logic[31:0] RD1D, RD2D,
            input logic[4:0] RsD, RtD, RdD,
            input logic[31:0] SignImmD,
            input logic[31:0] PCPlus4D,
               output logic RegWriteE, MemtoRegE, MemWriteE,
               output logic[2:0] AluControlE,
               output logic AluSrcE, RegDstE, BranchE,
               output logic[31:0] RD1E, RD2E,
               output logic[4:0] RsE, RtE, RdE,
               output logic[31:0] SignImmE,
               output logic[31:0] PCPlus4E);

   always_ff @(posedge clk, posedge reset)begin
      if ( reset || clr ) begin
         RD1E <= 0;
         RD2E <= 0;
         RsE <= 0;
         RtE <= 0;
         RdE <= 0;
         SignImmE <= 0;
         PCPlus4E <= 0;
         RegWriteE <= 0;
         MemtoRegE <= 0;
         MemWriteE <= 0;
         AluControlE <= 3'b000;
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

```
            AluSrcE <= 0;
            RegDstE <= 0;
            BranchE <= 0;
        end
        else begin
            RD1E <= RD1D;
            RD2E <= RD2D;
            RsE <= RsD;
            RtE <= RtD;
            RdE <= RdD;
            SignImmE <= SignImmD;
            PCPlus4E <= PCPlus4D;
            RegWriteE <= RegWriteD;
            MemtoRegE <= MemtoRegD;
            MemWriteE <= MemWriteD;
            AluControlE <= AluControlD;
            AluSrcE <= AluSrcD;
            RegDstE <= RegDstD;
            BranchE <= BranchD;
         end
    end
endmodule
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

PipeMtoW:

```
module PipeMtoW(input logic clk, reset, RegWriteM, MemtoRegM,
            input logic[31:0] ReadDataM, ALUOutM,
            input logic[4:0] WriteRegM,
               output logic RegWriteW, MemtoRegW,
               output logic[31:0] ReadDataW, ALUOutW,
               output logic[4:0] WriteRegW);

   always_ff @(posedge clk, posedge reset) begin
      if ( reset ) begin
         ReadDataW <= 32'h00000000;
         ALUOutW <= 32'h00000000;
         WriteRegW <= 5'b00000;
         RegWriteW <= 0;
         MemtoRegW <= 0;
      end
      else begin
         ReadDataW <= ReadDataM;
         ALUOutW <= ALUOutM;
         WriteRegW <= WriteRegM;
         RegWriteW <= RegWriteM;
         MemtoRegW <= MemtoRegM;
      end
   end
endmodule
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

PipeEtoM:

```
module PipeEtoM(input logic clk, reset, RegWriteE, MemtoRegE, MemWriteE, BranchE, Zero,
                input logic[31:0] ALUOut,
                input logic [31:0] WriteDataE,
                input logic[4:0] WriteRegE,
                input logic[31:0] PCBranchE,
                    output logic RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM,
                    output logic[31:0] ALUOutM,
                    output logic [31:0] WriteDataM,
                    output logic[4:0] WriteRegM,
                    output logic[31:0] PCBranchM);

    always_ff @(posedge clk, posedge reset) begin
        if ( reset  ) begin
            ZeroM <= 0;
            ALUOutM <= 3'b000;
            WriteDataM <= 32'h00000000;
            WriteRegM <= 5'b00000;
            PCBranchM <= 32'h00000000;
            RegWriteM <= 0;
            MemtoRegM <= 0;
            MemWriteM <= 0;
            BranchM <= 0;
        end
        else begin
            ZeroM <= Zero;
            ALUOutM <= ALUOut;
            WriteDataM <= WriteDataE;
            WriteRegM <= WriteRegE;
            PCBranchM <= PCBranchE;
            RegWriteM <= RegWriteE;
            MemtoRegM <= MemtoRegE;
            MemWriteM <= MemWriteE;
            BranchM <= BranchE;
         end
    end
endmodule
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

datapath:

```
module datapath (input  logic clk, reset,
                                input logic [31:0] PCF, instr,
                                input logic RegWriteD, MemtoRegD, MemWriteD,
                                input logic [2:0] AluControlD,
                                input logic AluSrcD, RegDstD, BranchD,
                                   output logic PCSrcM, StallD, StallF,
                                   output logic[31:0] PCBranchM, PCPlus4F, instrD, ALUOutM, ResultW,
WriteDataM );

         logic ForwardAD, ForwardBD,  FlushE;
         logic [1:0] ForwardAE, ForwardBE;
         logic [31:0] PCPlus4D, RD1D, RD2D, SignImmD, PCPlus4E, RD1E, RD2E, SignImmE;
         logic [4:0] RsD, RtD, RdD, RsE, RtE, RdE, WriteRegE, WriteRegM, WriteRegW;
         logic RegWriteE, MemtoRegE, MemWriteE, RegWriteM, MemtoRegM, MemWriteM,
RegWriteW, MemtoRegW;
         logic [2:0] AluControlE;
         logic AluSrcE, RegDstE, BranchE, BranchM;
         logic [31:0] PCBranchE, WriteDataE, ReadDataM, ReadDataW, SrcAE, SrcBE, ALUOut,
ALUOutW, resShifter;
         logic Zero, ZeroM;
         logic [31:0] instr;


         assign PCSrcM = BranchM & ZeroM;
         assign RsD =  instrD[25:21];
         assign RtD =  instrD[20:16];
         assign RdD =  instrD[15:11];
         //assign WriteDataE =  // WRITE YOUR CODE HERE;  This is commented out.

         adder          addr1 ( PCF, 32'd4, PCPlus4F );

         PipeFtoD       ftd ( instr, PCPlus4F, ~StallD, clk, reset, instrD, PCPlus4D );

         regfile        rf ( clk, RegWriteW, instrD[25:21], instrD[20:16], WriteRegW, ResultW,
RD1D, RD2D );

         signext        se ( instrD[15:0], SignImmD );

         PipeDtoE       dte ( FlushE, clk, reset, RegWriteD, MemtoRegD, MemWriteD,
AluControlD, AluSrcD, RegDstD, BranchD, RD1D, RD2D, RsD, RtD, RdD, SignImmD,
PCPlus4D,
RegWriteE, MemtoRegE, MemWriteE, AluControlE, AluSrcE, RegDstE, BranchE, RD1E, RD2E,
RsE, RtE, RdE, SignImmE, PCPlus4E );
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

```
	mux2 #(5)	mux1 ( RtE, RdE, RegDstE, WriteRegE );

	sl2		sl21 ( SignImmE, resShifter );

	adder		addr2 ( resShifter, PCPlus4E, PCBranchE );

	mux4 #(32)	mux2 ( RD1E, ResultW, ALUOutM, ALUOutM, ForwardAE, SrcAE );

	mux4 #(32)	mux3 ( RD2E, ResultW, ALUOutM, ALUOutM, ForwardBE, WriteDataE );

	mux2 #(32)	mux4 ( WriteDataE, SignImmE, AluSrcE, SrcBE );

	alu		alu1 ( SrcAE, SrcBE, AluControlE, ALUOut, Zero, reset );

	PipeEtoM	etm ( clk, reset, RegWriteE, MemtoRegE, MemWriteE, BranchE, Zero,
ALUOut, WriteDataE, WriteRegE, PCBranchE, RegWriteM, MemtoRegM, MemWriteM,
BranchM, ZeroM, ALUOutM, WriteDataM, WriteRegM, PCBranchM);

	dmem		dmem1 ( clk, MemWriteM, ALUOutM, WriteDataM, ReadDataM );

	PipeMtoW	mtw ( clk, reset, RegWriteM, MemtoRegM, ReadDataM, ALUOutM,
WriteRegM, RegWriteW, MemtoRegW, ReadDataW, ALUOutW, WriteRegW);

	mux2 #(32)	mux5 ( ReadDataW, ALUOutW, MemtoRegW, ResultW );

	HazardUnit	hu (  RegWriteW, WriteRegW, RegWriteM, MemtoRegM, WriteRegM,
RegWriteE, MemtoRegE, RsE, RtE, RsD, RtD, ForwardAE, ForwardBE, FlushE, StallD, StallF );


endmodule
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

HazardUnit:

```
module HazardUnit( input logic RegWriteW,
            input logic [4:0] WriteRegW,
            input logic RegWriteM,MemToRegM,
            input logic [4:0] WriteRegM,
            input logic RegWriteE,MemtoRegE,
            input logic [4:0] rsE,rtE,
            input logic [4:0] rsD,rtD,
            output logic [1:0] ForwardAE,ForwardBE,
            output logic FlushE,StallD,StallF);

  logic lwstall;
  always_comb begin
      lwstall <= ( ( rsD == rtE ) || ( rtD == rtE ) ) & MemtoRegE;
      StallF <= lwstall;
      StallD <= lwstall;
      FlushE <= lwstall;

      if ( ( rsE != 5'b00000 ) & ( rsE == WriteRegM ) & RegWriteM )      ForwardAE <= 2'b10;
      else if ( ( rsE != 5'b00000 ) & ( rsE == WriteRegW ) & RegWriteW )  ForwardAE <= 2'b01;
      else                                                                ForwardAE <= 2'b00;

      if ( ( rtE != 5'b00000 ) & ( rtE == WriteRegM ) & RegWriteM )      ForwardBE <= 2'b10;
      else if ( ( rtE != 5'b00000 ) & ( rtE == WriteRegW ) & RegWriteW )  ForwardBE <= 2'b01;
      else                                                                ForwardBE <= 2'b00;

  end

endmodule
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

mips:

```
module mips (input  logic        clk, reset,
             output logic[31:0]  PCF,
             input  logic[31:0]  instr,
             output logic[31:0]  aluout, resultW,
             output logic[31:0]  instrOut, WriteDataM,
             output logic StallD, StallF );


    logic memtoreg, zero, alusrc, regdst, regwrite, jump, PCSrcM, branch, memwrite;
    logic [31:0] PCPlus4F, PCm, PCBranchM, instrD;
    logic [2:0] alucontrol;
    assign instrOut = instrD;

    mux2 #(32) pcsrcmux ( PCPlus4F, PCBranchM, PCSrcM, PCm );

    PipeWtoF   wtf ( PCm, ~StallF, clk, reset, PCF);

    datapath      dp ( clk, reset, PCF, instr, regwrite, memtoreg, memwrite, alucontrol, alusrc, regdst,
branch, PCSrcM, StallD, StallF, PCBranchM, PCPlus4F, instrD, aluout, resultW, WriteDataM );

    controller    c ( instr[31:26], instr[5:0], memtoreg, memwrite, alusrc, regdst, regwrite, jump,
alucontrol, branch);

endmodule
```

imem:

```
module imem ( input logic [5:0] addr, output logic [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM
          always_comb
            case ({addr,2'b00})                     // word-aligned fetch
//                address                  instruction
//                -------            -----------
                  8'h00: instr = 32'h20080005;
                  8'h04: instr = 32'h20090006;
                  8'h08: instr = 32'h20040001;
                  8'h0c: instr = 32'h20050002;
                  8'h10: instr = 32'had280000;
                  8'h14: instr = 32'h8d090001;
                  8'h18: instr = 32'h01245020;
                  8'h1c: instr = 32'h01255022;
              default:  instr = {32{1'bx}};        // unknown address
            endcase
endmodule
```

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

e) [9 points] Study the four example programs which respectively have no hazard, compute-use hazard, load-use hazard and branch-hazard. For hazardous programs explicitly show the places where hazards occur and the registers causing the hazard.

- No Hazard

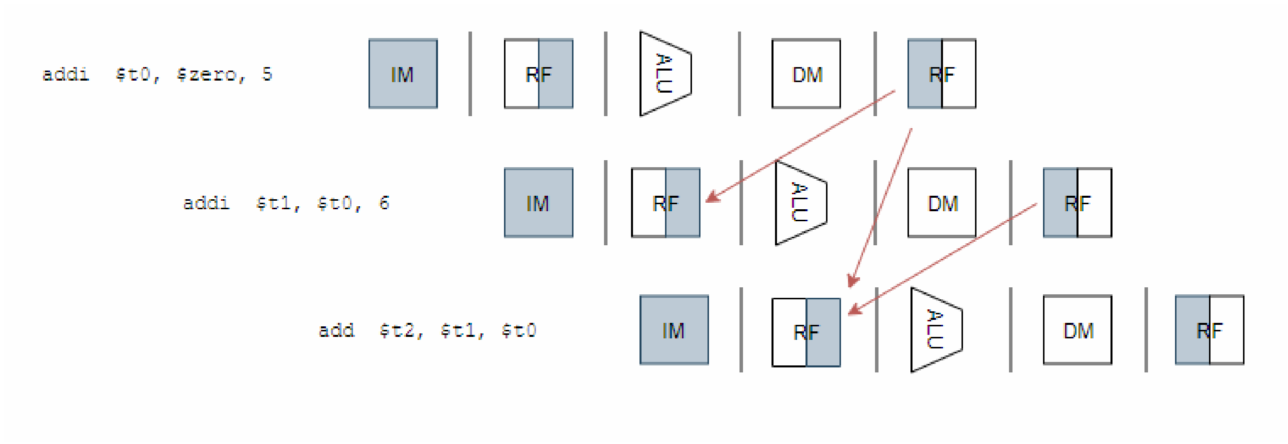
`addi $t0, $zero, 7`	`8'h00: 32'h20080007;`
`addi $t1, $zero, 5`	`8'h04: 32'h20090005;`
`addi $t2, $zero, 0`	`8'h08: 32'h200a0000;`
`addi $t3, $t0, 15`	`8'h0c: 32'h210b000f;`
`add $t2, $t0, $t1`	`8'h10: 32'h01095020;`
`or $t2, $t0, $t1`	`8'h14: 32'h01095025;`
`and $t2, $t0, $t1`	`8'h18: 32'h01095024;`
`sub $t2, $t0, $t1`	`8'h1c: 32'h01095022;`
`slt $t2, $t0, $t1`	`8'h20: 32'h0109502a;`
`sw $t0, 2($t1)`	`8'h24: 32'had280002;`
`lw $t1, 0($t0)`	`8'h28: 32'h8d090000;`
`beq $t0, $zero, 1`	`8'h2c: 32'h1100fff5;`
`addi $t2, $zero, 10`	`8'h30: 32'h200a000a;`
`addi $t1, $zero, 12`	`8'h34: 32'h2009000c;`

No hazard is present in this program. (We may think that there might be a branch hazard, but there isn't. Since $t0 holds the value 7 and since it is not equal to 0, branch is not taken.)

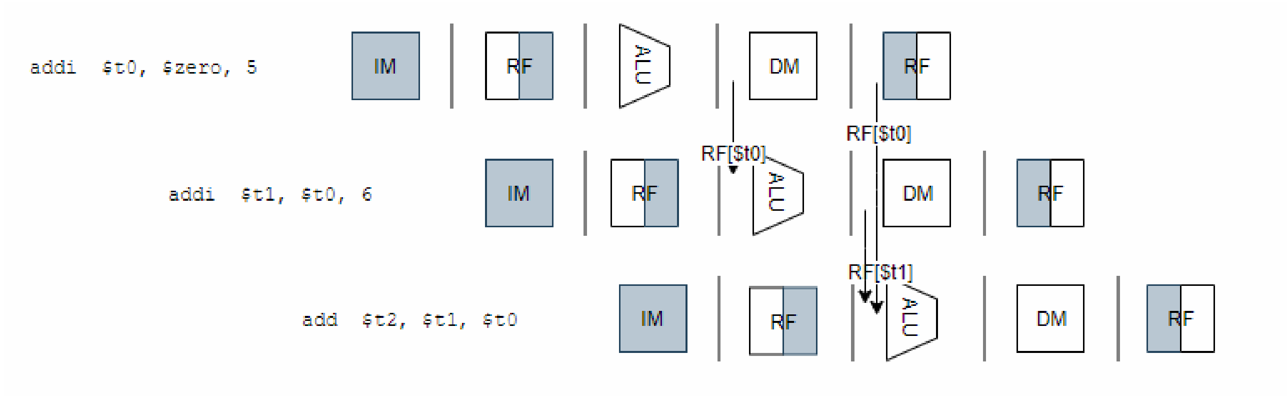
CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054

- Compute-use hazard

addi $t0, $zero, 5	8'h00: 32'h20080005;
addi $t1, $t0, 6	8'h04: 32'h21090006;
add $t2, $t1, $t0	8'h08: 32'h01285020;



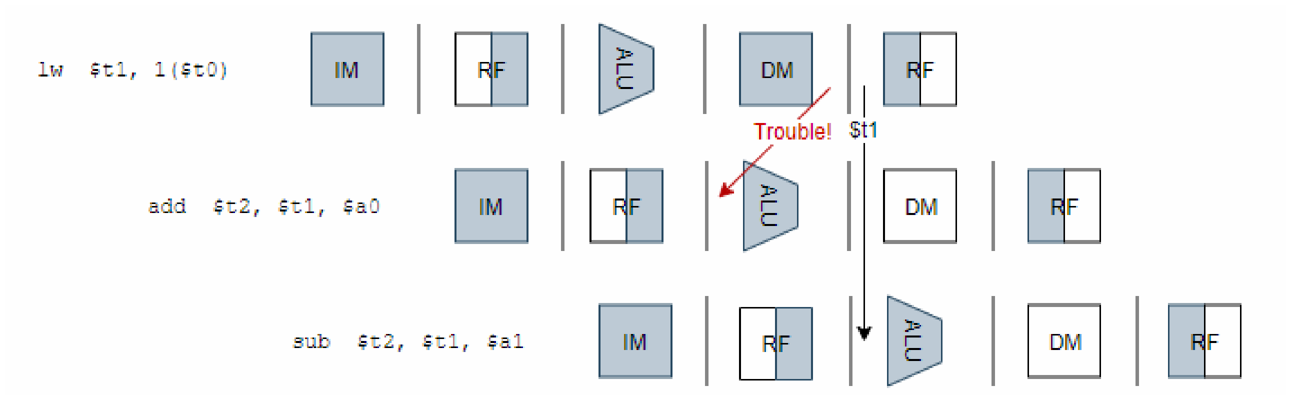
By forwarding ALUOutM in 3rd clock cyle and ResultW and ALUOutM in 4th clock cycle, to Execution Stages of corresponding clock cycles, data hazard is eliminated.



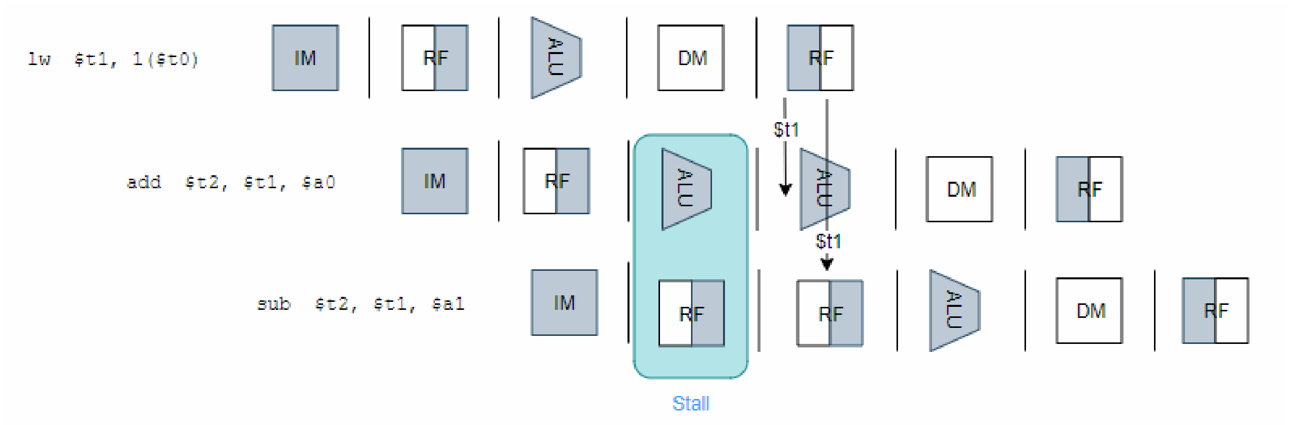
- Load-use hazard

addi $t0, $zero, 5	8'h00: 32'h20080005;
addi $t1, $zero, 6	8'h04: 32'h20090006;
addi $a0, $zero, 1	8'h08: 32'h20040001;
addi $a1, $zero, 2	8'h0c: 32'h20050002;
sw $t0, 0($t1)	8'h10: 32'had280000;
lw $t1, 1($t0)	8'h14: 32'h8d090001;
add $t2, $t1, $a0	8'h18: 32'h01245020;
sub $t2, $t1, $a1	8'h1c: 32'h01255022;

CS 224
Section No.: 2
Spring 2019
Lab 05
Berrak Taşkınsu / 21602054



Here, $t1 is both the destination register of the lw instruction and the source register of both add and sub instructions. Since data is read from data memory, nearly at the end of the lw instruction, forwarding is not an solution on its own. Therefore we need to stall the pipeline, as shown in below.



●● Branch hazard

addi $t1, $zero, 2	8'h00: 32'h20090002;
beq $zero, $zero, 2	8'h04: 32'h10000002;
addi $t1, $zero, 5	8'h08: 32'h20090005;
addi $t1, $t1, 6	8'h0c: 32'h21290006;
addi $t1, $zero, 8	8'h10: 32'h20090008;
addi $a0, $zero, 0	8'h14: 32'h20040000;
addi $a1, $zero, 0	8'h18: 32'h20050000;
sw $t1, 0($zero)	8'h1c: 32'hac090000;

Since 0 = 0, branch will be taken and pc will jump to the 5th instruction, addi $t1, $zero, 8. Clock cycles will be lost, since branch is decided to be taken at the Memory Stage at 5th clock cycle. The solution is early branch resolution, which allows the branch to be decided in the Decode Stage. Yet this improvement results in a different hazard in the Decode Stage, so Hazard Unit should provide both logic if an early branch is taken to reduce the execution time.