**Lab5 Preliminary**

CS224

Section No: 1

Fall 2019

Lab No. 5

Fatih Sevban Uyanık

21602486

b) **[10 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.

**compute-use:** This hazard is a data type of hazard. If there are subsequent instructions and if these instructions are dependent on each other because of a register value, then there is a compute-use hazard or RAW(read and write hazard) available. In this case, one instruction writes a register and subsequent instructions read this specific register. For example

18 --> add $t0, $t2, $t3

1c --> and $s0, $t0, $t1

In this example, the instruction in 18 writes to the register $t0 and the instruction 1c reads the register $t0. In this hazard, the affected stage is alu because the computed value in instruction 18 could not be written to register $t0. Hence, the register $t0 does not contain the computed value in instruction 18 yet and a wrong value is taken in instruction 1c.

**load-use:** This hazard is a data type of hazard and cannot be resolved as easy as RAW or compute-use data hazards. This hazard is encoutered in lw instructions. The lw instruction is not able to read the data from memory until the end of memory stage. Because of this reason, the computed value cannot be bypassed to the subsequent instructions execute stage. This problem is named as having two-cycle latency. The subsequent or next instruction is not able to read the computed value at the first two subsequent cycles. For example:

18 --> lw $a0, 10($zero)

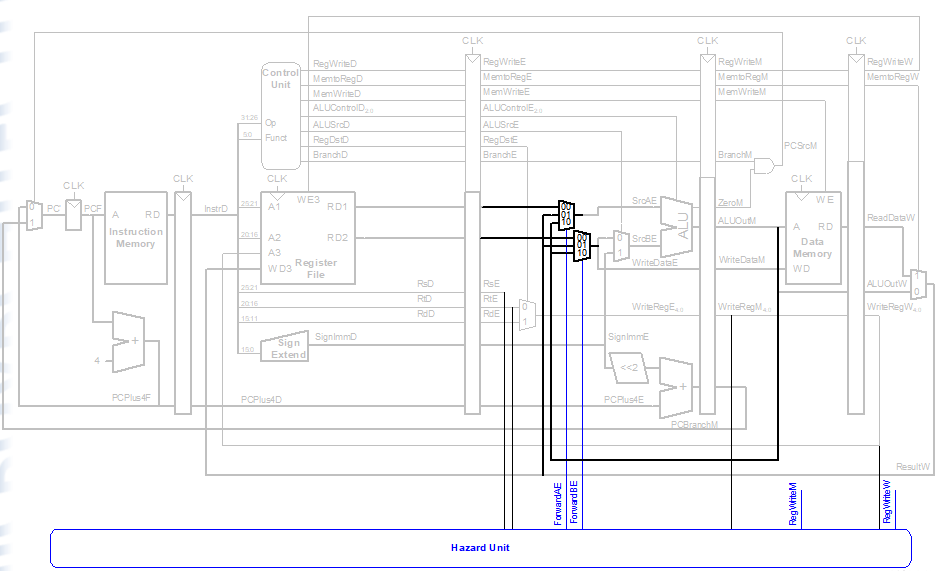
1c --> add $s0, $a0, $t1

In this case, the instruction in 18 reads from memory and writes to the register $a0 and the instruction 1c reads the register $a0. In this hazard, the affected stage is also alu because the retrieved value from memory in instruction 18 could not be written to register $t0. Hence, the register $t0 does not contain the retrieved value in instruction 18 yet and a wrong value is taken in instruction 1c. In this case, 2 cycles are needed to get the value from previous memory stage.

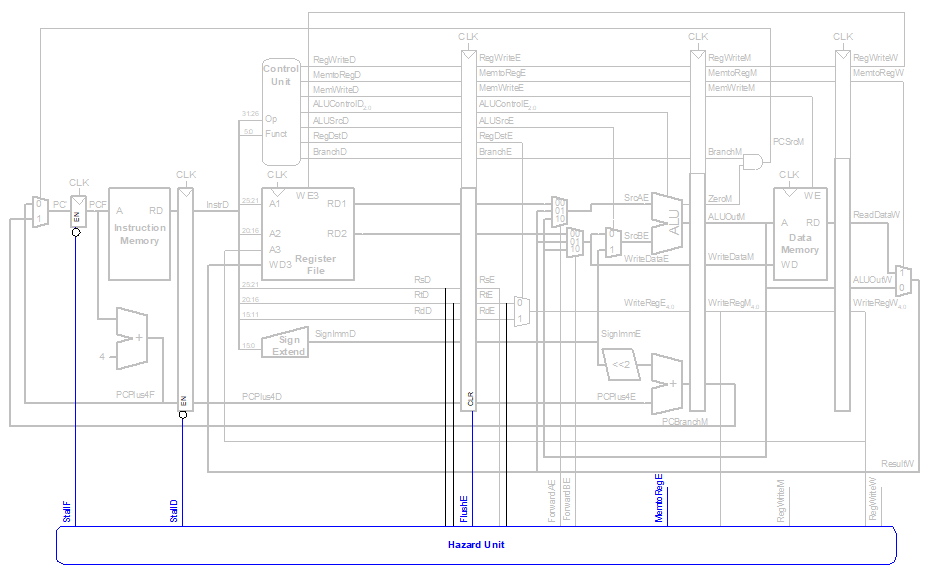
**branch:** This hazard is a control type of hazard. This hazard is encountered in beq instructions. The pipeline processor is not capable of determining which instruction to fetch in the instruction memory. The reason behind it is that the branch decision could not be done while the subsequent instruction is fetched. In this type of hazard, the stage of fetch is affected because in this stage, the new instruction will be determined through pc register.

c) **[10 points]** For each hazard, give the solution (forwarding, stalling, flushing, combination of these), and explanation of what, when, how.

**compute-use:** Forwarding would resolve this issue. In this hazard, as mentioned, alu stage was affected because of reading the wrong value. This happenned because the compuıted value in the previuos stage was not writed yet in to the register file. By the courtesy of forwarding, the computed value in the previous stage is forwarded or conducted directly to the alu stage without waiting until the value is written to register file. The modification that needs to be done to the processor is that adding two multiplexers in front of the alu. This modification would enable to get values from register file, memory stage and write back stages. Hence, we can directly forward data from these stages. In addition, a hazard unit is needs to be added to the processor as well because this unit detects whether there is a compute-use hazard and directs from which stage the data needs to be forwarded ehich are memory and write back stages. The following figure shows the modifications that is done.



**load-use:** For load-use hazards, the solution is stalling. Stalling enables to hold the operations until the desired value is available. The operations that are holded while stalling are fetch and decode. While stalling, the pipelined processor repeats the fetch and decode stages and the unneccessary stages are acting such as a bubble. These bubbles are also acting like nope instructions. The hazard unit is responsible for detecting load-use hazards and three connections are added to hazard unit. These are StallF, StallD and FlushE. Stall signals are responsible for making the bubbles or stalling and FlushE is responsible for flushing the current or invalid execution stage. The following figure shows the modification that needs to be done to the processor:



**Branch:** The branch decision is made in the memory stage. That’s why, in the fetch stage, it is not known which instruction to fetch. Hence, one solution is to use stalling. Because there are stages between fetch and memory stages, three stages needs to be stalled but this approach is fairly inefficcient because it degrades the performance of the pipelined processor. Another way of dealing with control hazards are making predictions about whether the branch is taken or not. In case of making false predictions, the unneccessary performed instructions need to be flushed. Because the branch instruction is determined three cycles later, the wrong predicted three instructions need to be flushed or discarded. This event is called branch misprediction penalty. However, the branch misprediction penalty can be decreased futher to 1 when the branch decision could be moved from memory stage to decode stage. To achieve this, an equality comparator is needed in the decode stage such that it checks the operands and make the branch decision earlier. That way, there would be a much more efficient processor and the branch misprediction penalty would decrease to one. However, this modification results in a compute-use hazard which needs to be solved by adding multipleexers in front of the equality comparator module. The forwarded data will come from the write back stage and all hazards would be resolved in this way. The following figure indicates the changes that needs to be done to the processor:



After adding muxes in front of the equality comparator, we get the following figure:



d) **[10 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) so that your pipelined processor computes correctly.

// compute-use logic

if((rsE != 0)&&(rsE == WriteRegM)&&RegWriteM)

ForwardAE = 2'b10;

else if((rsE != 0)&&(rsE == WriteRegW)&&RegWriteW)

ForwardAE = 2'b01;

else ForwardAE = 2'b00;

if((rtE != 0)&&(rtE == WriteRegM)&& RegWriteM)

ForwardBE = 2'b10;

else if((rtE != 0)&&(rtE == WriteRegW)&&RegWriteW)

ForwardBE = 2'b01;

else ForwardBE = 2'b00;

// load-use

lwstall = ((rsD==rtE) || (rtD==rtE)) && MemtoRegE;

// branch logic

ForwardAD = (rsD != 0) && (rsD == WriteRegM) && RegWriteM;

ForwardBD = (rtD != 0) && (rtD == WriteRegM) && RegWriteM;

branchstall = (BranchD && RegWriteE && (WriteRegE == rsD || WriteRegE == rtD)) ||

(BranchD && MemToRegM && (WriteRegM == rsD || WriteRegM == rtD ))

// setting flush and stall variables.

StallF = (lwstall || branchstall);

StallD = (lwstall || branchstall);

FlushE = (lwstall || branchstall);

e) **[15 points]** Write small test programs, in MIPS assembly, that will show whether the pipelined processor is working or not. Each of your test programs should be designed to catch problems, if there are any, in the execution of MIPS instructions in your pipelined machine. Write:

* A test program with no hazards (to verify that there are no problems with the connections in your pipeline etc.)
* A test program that has one type of hazard, and another, and another...

In the end, have at least 4 test programs (testing at least 3 hazards) with their machine code (in hex).

// 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;

// As it is stated the code block above, there is no hazard.

// Compute-use hazard

addi $t0, $zero, 5 8'h00: 32'h20080005;

addi $t1, $t0, 6 8'h04: 32'h21090006; // $to is tried to use but the correct value is not written.

add $t2, $t1, $t0 8'h08: 32'h01285020; // $to and $t1 are tried to use but the correct values are not written.

// 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;// $t1 is tried to use but the correct value is not written yet

sub $t2, $t1, $a1 8'h1c: 32'h01255022; // $t1 is tried to use but the correct value is not written yet

// Branch hazard

addi $t1, $zero, 2 8'h00: 32'h20090002;

beq $zero, $zero, 2 8'h04: 32'h10000002;//beq has not decided the result yet, but new fetch is initialized.

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;