Bilkent University – CS 224 Computer Organization

Design Report | Lab #5 | Section #2 | Defne Betül Çiftci | 21802635

12.05.2020

b) 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”, “branch” etc.), the pipeline stages that are affected.

Data hazards

Compute-use hazard: (Specifically a Read-after-write (RAW) hazard) It happens when one instruction **computes** a register and subsequent instructions read this register before it is written; so the instruction reads the previous register instead of the updated register.

ADD **R1**, R2, R3	IF	ID	EX	MEM	**WB**	
AND R7, **R1**, R8		IF	**ID**	EX	MEM	WB

So, the pipeline stages affected are WriteBack and Decode stages.

Load-use hazard: (Specifically a Read-after-write (RAW) hazard) It happens when one instruction **writes** a register and subsequent instructions read this register before it is written; so the instruction reads the previous register instead of the updated register.

LW **R2**, 20(R1)	IF	ID	EX	MEM	**WB**	
AND R7, **R2**, R8		IF	**ID**	EX	MEM	WB

So, the pipeline stages affected are WriteBack and Decode stages.

Load-store hazard: When data is *loading* into a register and another instruction wants to store the register when it is not written yet.

Compute-store hazard: When data is being *computed* into a register and the next instruction wants to store the register when it is not written yet.

Control hazards

Branch hazard: The pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched.

c) [15 points] For each hazard, give the solution (forwarding, stalling, flushing, combination of these), and explanation of what, when, how.

Data hazards

There are two solutions to *compute-use hazards.* The first solution is to reorder the instructions, however there are other solutions for when reordering is not possible to eliminate the hazards.

One possible solution is **forwarding.** We don’t wait for the result to be stored in a register and forward the results whenever they are ready. This requires extra connections in the datapath.

In this case, the result of the add is already decided after the Execution stage, so this value can then be forwarded to the AND instruction’s Execution stage.

ADD **R1**, R2, R3	IF	ID	**EX** R1	MEM	**WB**	
AND R7, **R1**, R8		IF	**ID**	**EX**	MEM	WB

Another solution for this kind of problem would be **stalling**. For this type of problem:

ADD **R1**, R2, R3	IF	ID	EX	MEM	**WB**		
OR R9, R2, R3		IF	ID	EX	MEM	WB	
AND R7, **R1**, R8			IF	**ID**	EX	MEM	WB

In which case this type of solution would be appropriate:

ADD **R1**, R2, R3	IF	ID	EX	MEM	**WB**			
OR R9, R2, R3		IF	ID	EX	MEM	WB		
stalling			*bubble*	*Bubble*	*bubble*	*bubble*	*bubble*	
AND R7, **R1**, R8				IF	**ID**	EX	MEM	WB

One disadvantage of stalling solution is that it wastes cycles.

To solve the problem with the example given in *load-use hazards*, **we cannot use forwarding for load-use hazards, so we stall**:

LW **R2,** 20(R1)	IF	ID	EX	MEM	**WB**			
stalling		bubble	bubble	Bubble	Bubble	bubble		
stalling			bubble	Bubble	Bubble	bubble	bubble	
AND R7, **R2**, R8				IF	**ID**	EX	MEM	WB

Solutions to *load-store* and *compute-store* hazards are similar to the solutions of load-use and compute-use hazards, respectively; we stall for load-store hazards and we can either forward or stall for compute-store hazards. An example of forwarding for compute-store hazards is as follows.

ADD **R1**, R2, R3	IF	ID	EX	MEM R1	**WB**	
SW **R1**, 20(R8)		IF	ID	EX	MEM	WB

For *load-store hazards*, **we cannot use forwarding for load-use hazards, so we stall**:

LW **R2,** 20(R1)	IF	ID	EX	MEM	**WB**	
stalling		bubble	bubble	bubble	bubble	bubble

SW **R2**, 20(R8)				IF	ID	EX	MEM	WB

Control hazards

Branch hazard:

For beq instruction, the branch is not determined until the 4th stage of the pipeline and instructions after the branch are executed before the branch occurs. These instructions must be flushed if branch happens – so we need to predict the branch result. Branch misprediction penalty (which is the number of the instructions flushed) could be reduced by determining the branch earlier.

One solution is to stall the pipeline until the branch decision is made.

beq R1, R2, 40	IF	ID	EX	**MEM**	WB				
Stall		Bubble	bubble	bubble	Bubble				
Stall			bubble	bubble	Bubble				
Stall				bubble	Bubble				
…									
…									
…									
slt R2, R1, R0					IF	ID	EX	MEM	WB

Because the decision is made in the Memory stage, the pipeline would have to be stalled for three cycles, which is wasteful for system performance.

An example of the solution of branch hazards by assuming the branch is not taken:

beq R1, R2, 40	IF	ID	EX	MEM	WB				
and R7, R3, R4		IF	ID	EX	MEM	WB			
or R5, R8, R10			IF	ID	EX	MEM	WB		
sub R9, R8, R10				IF	ID	EX	MEM	WB	
…									
…									
…									
slt R2, R1, R0					IF	ID	EX	MEM	WB

Instructions with blue color are flushed, which is wasteful because this means that we wasted 3 clock cycles. Assuming the branch is not taken is fine when we are correct, but if it is taken then it is wasteful.

With the introduction of early branch prediction (we look at the equation of rs and rt before the Execution (ALU) stage):

beq R1, R2, 40	IF	ID	EX	MEM	WB		
and R7, R3, R4		IF	ID	EX	MEM	WB	
…							
…							
…							
slt R2, R1, R0			IF	ID	EX	MEM	WB

Now we waste only 1 clock cycle by flushing.

Early branch decision hardware introduces a new RAW data hazard. Specifically, if one of the source operands for the branch was computed by a previous instruction and has not yet been written into the register file (compute-use hazard), the branch will read the wrong operand value from the register file. As before, we can solve the data hazard by *forwarding* the correct value if it is available or by *stalling* the pipeline until the data is ready.

d) [15 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.

If a result is in the:

- Writeback stage, it will be written in the first half of the cycle and read during the second half, so no hazard exists.
- in the Memory stage, it can be forwarded to the equality comparator. (we add new multiplexers here)
- Execute stage or in the Memory stage, stalling must be done in the Decode stage.

The function of the *Decode stage forwarding logic* is given below.

ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM

ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM

Execution stage forwarding:

```
if(rsE != 0) AND (rsE == WriteRegM) && RegWriteM)
    ForwardAE =2'b10
else if((RsE != 0) AND (RsE == WriteRegW) AND RegWriteW))
    ForwardAE = 2'b01
else
```

```
        ForwardAE = 2‘b00


    if( (RtE != 0) AND (RtE == WriteRegM) AND RegWriteM))
        ForwardBE =  2'b10
    else if((RtE != 0) && (RtE == WriteRegW) && RegWriteW)
        ForwardBE = 2'b01
    else
        ForwardBE = 2‘b00
```

What we do with the following logic equation is stall detection. Branch decision is made in the Decode stage with this implementation (early prediction). If there is a dependency on an ALU or on Memory stage, until the sources are ready it is stalled.

```
branchstall = (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD))
          OR (BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM ==
rtD))
```

Stalling could also be possible due to a load or a branch hazard, and we need to take this into account as well:

```
    StallF = StallD = FlushE = (lwstall OR branchstall)
```