CSE 30321 – Lecture 13/14 – In Class Handout

For the sequence of instructions shown below, show how they would progress through the pipeline.

For all of these problems:

- Stalls are indicated by placing the code of the stage where the hazard would be discovered in the succeeding square
- We will assume a standard 5 stage pipeline
  - (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, M = Memory Access, WB = Write Back)
- Assume that each stage of the pipeline takes just 1 clock cycle to finish.

Example 1:

- Assume that forwarding **HAS NOT** been implemented
- Assume that you **CANNOT** read and write a register in the same clock cycle

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Add $5, $3, $4	**IF**	**ID**	**EX**	**M**	**WB**												
Add $6, $5, $7		**IF**	**ID**	**ID**	**ID**	**ID**	**EX**	**M**	**W**	**Add must wait until $5 written by previous add; reads $5 in ID stage**							
LW $7, 0($6)			**IF**	**IF**	**IF**	**IF**	**ID**	**ID**	**ID**	**ID**	**EX**	**M**	**WB**	**LW stalled by prior add; needs $6 too – reads in CC #10**			
SUB $1, $2, $3							**IF**	**IF**	**IF**	**IF**	**ID**	**EX**	**M**	**WB**	**Pipeline full so SUB can't go**		
Add $9, $7, $8											**IF**	**ID**	**ID**	**ID**	**EX**	**M**	**WB**

(Last add must wait for $7 from LW)

Example 2:

- Let's do the same problem as before, but now assume that **forwarding HAS been implemented**
- Assume that you CANNOT read and write from the same register in the register file in the same clock cycle

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Add $5, $3, $4	**IF**	**ID**	**EX**	**M**	**WB**								**Data to be loaded into $5 available at end of CC 3 / Beginning of CC 4**				
Add $6, $5, $7		**IF**	**ID**	**EX**	**M**	**WB**							**Add gets data for $5 directly from output of ALU**				
LW $7, 0($6)			**IF**	**ID**	**EX**	**M**	**WB**						**Lw gets $6 data directly from output of ALU**				
SUB $1, $2, $3				**IF**	**ID**	**EX**	**M**	**WB**					**No dependencies on any other instructions**				
Add $9, $7, $8					**IF**	**ID**	**EX**	**M**	**WB**				**Add gets $7 data from latch between memory and writeback stage (its an input to ALU)**				

(Ability to forward eliminates *all* stalls!)

Example 3:

- Like Example 2, assume that **forwarding HAS been implemented**
- Assume that you **CAN** read and write a register in the same clock cycle

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Add $1, $6, $9	**IF**	**ID**	**EX**	**M**	**WB**												
Add $6, $2, $4		**IF**	**ID**	**EX**	**M**	**WB**											
LW $7, 0($6)			**IF**	**ID**	**EX**	**M**	**WB**					**LW instruction producing result that will be stored in $7; even with forwarding must stall; data not available until end of CC #6 and needed at beginning of CC #6**					
SUB $1, $7, $8				**IF**	**ID**	**ID**	**EX**	**M**	**WB**								
Add $9, $1, $8					**IF**	**IF**	**ID**	**EX**	**M**	**WB**							

Example 4:

- Assume that **forwarding HAS been implemented**
- We will stall if we encounter a branch instruction
- Branches or Jumps are resolved after the EX stage.
- Assume that register $2 has the value of 0 and $3 has the value of 0

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LW $1, 4($9)	**IF**	**ID**	**EX**	**M**	**W**												
Add $4, $1, $9		**IF**	**ID**	**ID**	**EX**	**M**	**W**							**Add gets data from lw forwarding**			
Sub $7, $4, $9			**IF**	**IF**	**ID**	**EX**	**M**	**WB**						**Sub gets data from add forwarding**			
BEQ $2, $3, X					**IF**	**ID**	**EX**							**BEQ must still wait to enter the pipeline**			
Add $9, $8, $7																	
And $4, $5, $5																	
X: Add $4, $5, $9								**IF**	**ID**	**EX**	**M**	**WB**		**Can't start Add until after BEQ finishes executing (comparing)**			

Example 5:

- Assume that **forwarding HAS been implemented**
- We will predict that any branch instruction is **NOT TAKEN**
- Branches or Jumps are resolved after the EX stage.
- Assume that register $2 has the value of 0 and $3 has the value of 0

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LW $1, 4($9)	**IF**	**ID**	**EX**	**M**	**W**												
Add $4, $1, $9		**IF**	**ID**	**ID**	**EX**	**M**	**W**										
Sub $7, $4, $9			**IF**	**IF**	**ID**	**EX**	**M**	**WB**									
BEQ $2, $3, X					**IF**	**ID**	**EX**										
Add $9, $8, $7						**IF**	**ID**					**Add and And start down pipeline; however, they would not change state until CC 10 and 11. They never get this far so there is no harm done. We can kill them and restart the next add instruction.**					
And $4, $5, $5							**IF**										
X: Add $4, $5, $9								**IF**	**ID**	**EX**	**M**	**W**					

Example 6:

- Assume that **forwarding HAS been implemented**
- We will predict that any branch instruction is **TAKEN**
- Branches or jumps are resolved after the EX stage.
- Assume that register $2 has the value of 0 and $3 has the value of 0

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LW $1, 4($9)	**IF**	**ID**	**EX**	**M**	**W**												
Add $4, $1, $9		**IF**	**ID**	**ID**	**EX**	**M**	**W**										
Sub $7, $4, $9			**IF**	**IF**	**ID**	**EX**	**M**	**WB**									
BEQ $2, $3, X					**IF**	**ID**	**EX**										
Add $9, $8, $7																	
And $4, $5, $5																	
X: Add $4, $5, $9						**IF**	**ID**	**EX**	**M**	**W**							

This is the best situation – the last add instruction finishes 2 CC's earlier.

Example 7:
For the sequence of instructions shown below, show how they would progress through the pipeline.

Part 1:

- Assume that **forwarding HAS been implemented**
- We will predict that any branch instruction is **NOT TAKEN**
- Branches or Jumps are resolved after the EX stage.
- Assume that register \$8 *does not equal* \$1 for the 1st Beq instruction
- Assume that register \$17 *does equal* \$26 for the 2nd Beq instruction

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SUB \$1, \$2, \$3	F	D	E	M	W												
Add \$8, \$9, \$10		F	D	E	M	W											
Beq \$1, \$8, X			F	D	E	M	W				Technically, nothing done, but can think of instruction as progressing through pipeline						
Lw \$7, 0(\$20)				F	D	E	M	W									
Add \$11, \$7, \$12					F	D	D	E	M	W							
Sw \$11, 0(\$24)						F	F	D	E	M	W						
X: Addi \$17, \$17, 1								F	D	E	M	W					
Beq \$17, \$26, Y									F	D	E	M	W				
Sub \$5, \$6, \$7										F	D						
Or \$8, \$5, \$5											F						
Y: Addi \$17, \$17, 1												F	D	E	M	W	
Sw \$17, 0(\$10)													F	D	E	M	W
SUB \$1, \$2, \$3														F	D	E	…
Add \$8, \$9, \$10															F	D	…

Part 2:

- (i) Assume that this sequence of code is executed 100 times. How many cycles does the pipelined implementation take?
- (ii) How many cycles would this code take in a multi-cycle implementation?

- From Part 1, you can see that it takes 17 clock cycles to execute 12 instructions.
- However, we can start the next "iteration" in clock cycle 14. Therefore, it *really* only takes 13 cycles for each iteration and 17 CCs for the last one.
- Therefore, iterations 1 through 99 take 13 CCs each
    - (13 x 99 = 1287 CCs)
- Iteration 100 takes 17 CCs
- Therefore 1287 CCs + 17 CCs = 1304 CCs

- For the multi-cycle implementation, we have:
    - 9 instructions that take 4 CCs
    - 2 instructions that take 3 CCs
    - 1 instruction that takes 5 CCs
- Therefore, each "iteration" takes: (9x4) + (2x3) + (1x5) = 36 + 6 + 5 = 47 CCs
- If there are 100 iterations, then 4700 CCs are required

Pipelining gives us a speed up of 4700 / 1304 = 3.6 for this implemention

- Little to no extra HW is needed!