CS152/252A
Spring 2026

Pipelining
Discussion 2

1 Iron Law

1.1 For each term in the Iron Law, give at least three techniques that improve that term.

Instructions/program:

(a) Better algorithms

(b) Better compiler optimizations

(c) Different programming languages

(d) Complex instructions (RISC to CISC)

(e) Remove branch delay slots

Cycles/instruction:

(a) From multi-cycle implementation to pipelined implementation

(b) Optimize microcode

(c) Bypassing (forwarding)

(d) Adding branch delay slots

(e) Decoupling

(f) Branch prediction

(g) Prefetching

Time/cycle:

(a) Shorten critical path

(b) Minimizing logic depth; less logic complexity; less wire congestion

(c) Deeper pipelining

(d) Better VLSI technology

(e) CISC to RISC

(f) Different clock domains

1.2 Explain how each term changes given the proposed modification (increase / decrease / no effect).

(a) [**Quiz 1, 2011**] In a classic RISC pipeline, modify the ISA (and thus the microarchitecture) to use hardware interlocking instead of software interlocking for both branch delay slots and load-use delay slots.

Instructions/program: Decrease (fewer NOPs)

Cycles/instruction: Increase (fewer instructions flushed from pipeline)

Time/cycle: Increase (more control logic - high fan-out stall signals)

(b) [**Quiz 1, 2013**] Remove hardware floating-point instructions and instead use software subroutines for floating-point arithmetic

Instructions/program: Increase (need more instructions to emulate floating-point operations)

Cycles/instruction: Decrease (integer operations usually lower latency)

Time/cycle: No effect (FPU is usually pipelined to not be on the critical path)
OR
Decreases (less logic)

2 Pipelining

2.1 Data Hazards: Identify all potential hazards

```
ADDI   x1, x0, 4
SW     x1, 8(x2)
SLLI   x3, x1, 1
ADD    x2, x1, x3
LW     x1, 0(x3)
SUB    x1, x0, x0
```

RAW	WAR	WAW
`addi -> sw`	`sw -> add`	`addi -> lw`

2.2 Structural Hazards

(a) Are there any structural hazards in a classic 5-stage RISC pipeline?

No, hardware resources are not shared between pipeline stages

(b) Are there any structural hazards in the single-bus microcoded machine?

The shared bus

(c) What modifications to a RISC pipeline may introduce a structural hazard?

- Single-ported memory (combining IF and MEM)
- Functional units / RF write ports in superscalar implementation
- Multi-cycle unpipelined functional units / memory (iterative multiplier/divider, blocking cache)

2.3 What does the following code do? How many iterations does it run?

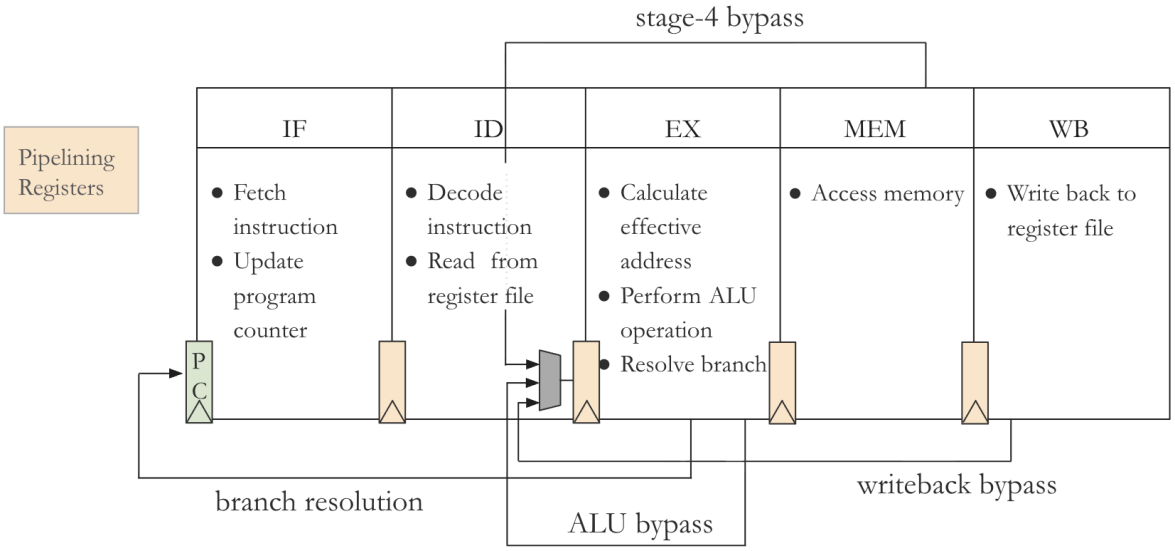
```
      ADDI  x2, x0, 0x700
LOOP: ADD   x1, x2, x0
      LW    x2, 4(x2)
      BNE   x2, x0, LOOP
```

Memory Address	**Memory Value**
0x400	0x000
0x404	0xD40
…	…
0x700	0x9F0
0x704	0x400
…	…
0x9F0	0x400
0x9F4	0x000
…	…
0xD40	0x000
0xD44	0x9F0

This code traverses a linked list to find a pointer to the last node and saves that address in `x1`. It runs for 4 iterations.

1st iteration: We load from the the address `0x704` and save the value `0x400` in `x2`.
2nd iteration: We copy the `0x400` to `x1` and load from `0x404` and obtain `0xD40`.
3rd iteration: We copy the `0xD40` to `x1` and dereferencing `0xD44` results in the value `0x9F0` being saved in `x2`.
4th iteration: the `0x9F0` is copied from `x2` to `x1`, and we load from the address `0x9F4` to get the value `0x000`. Now that `x2 == x0`, we exit the loop, with `x1 == 0x9F0` in the end.

2.4 Using the code from the previous question, fill out the pipeline diagram for the following pipeline, assuming that branches are always predicted not taken.



- What is the CPI for the given code sequence?

  (27 - 5 + 1)/13 = 23/13
  The 27 comes from the total number of cycles, the −5 comes from the pipelining of instructions, and the +1 is to account for the first cycle.

- Consider splitting MEM into two stages, M1 and M2. How does the CPI change?

  (23 + 4)/13 = 27/13
  We add 4 because there are 4 `lw x2 4(x2)` instructions. For each of these instructions, the following `bne` instruction must spin for one extra cycle as it waits for the `lw` instruction to reach the `WB` stage to receive the correct `x2` value and evaluate the branch.

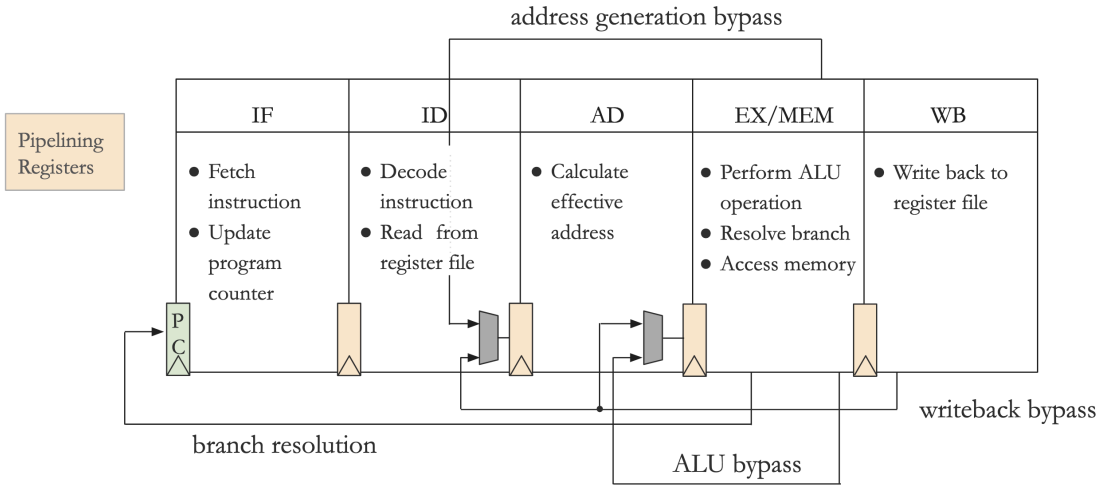
- What is the CPI if the BNE is always correctly predicted?

  (23 - 3*2)/13 = 17/13
  The first 3 `bne` each mispredict and wastes 2 cycles executing the wrong instruction. We can save `2*3=6` cycles from our original cycle count of 23 if we always correctly predict these branches.

Q2.4: Load-Use Interlock (LUI) pipeline

2.4	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
addi	F	D	X	M	W																									
(1) add		F	D	X	M	W																								
lw			F	D	X	M	W																							
bne				F	D	D	X	M	W																					
-					F	F	D	-	-	-																				
-							F	-	-	-	-																			
(2) add								F	D	X	M	W																		
lw									F	D	X	M	W																	
bne										F	D	D	X	M	W															
-											F	F	D	-	-	-														
-													F	-	-	-	-													
(3) add														F	D	X	M	W												
lw															F	D	X	M	W											
bne																F	D	D	X	M	W									
-																	F	F	D	-	-	-								
-																			F	-	-	-	-							
(4) add																				F	D	X	M	W						
lw																					F	D	X	M	W					
bne																						F	D	D	X	M	W			

2.5 Using the same code block, Fill out the pipeline diagram for the following pipeline, assuming that branches are always predicted not taken. [M. Golden and T. Mudge, "A comparison of two pipeline organizations", 1994]



- What is the CPI for the given code sequence?

  (26-5+1)/13 = 22/13
  The 26 comes from the total number of cycles, the −5 comes from the pipelining of instructions, and the +1 is to account for the first cycle.

- Consider splitting `EX/MEM` into two stages, `M1` and `EX/M2`. How does the CPI change?

  (23+3)/13 = 26/13
  We only add 3 in this case because there are 3 `bne` instructions where we stall as we wait for the branch to evaluate. By adding the `M1` stage before the `EX` stage where the branch is resolved, we need to wait an extra cycle for each of the mispredicted branches.

- What is the CPI if `bne` is always correctly predicted?

  (22-3*3)/13 = 13/13 = 1
  We can remove all stalls from mispredicted branches, and we no longer have any delays in our pipeline.

2.6 Suppose that the "load-use interlock" (LUI) pipeline from Q2.4 meets timing at 1 GHz. What is the minimum frequency at which the "address-generation interlock" (AGI) pipeline from Q2.5 performs better on the given code sequence, assuming perfect branch prediction?

We can use the Iron Law to determine the time/cycle of the AGI pipeline. Both pipelines have 13 instructions/program, but the LUI CPI is 23/13 while the AGI CPI with perfect branch prediction is 13/13. We can setup an equivalence of $\frac{23}{13} * \frac{1}{10^9} = \frac{13}{13} * \frac{1}{f_{\text{AGI}}}$. Solving this equality, we determine the minimum frequency $f_{\text{AGI}} = 565$ MHz.

2.7 Under what circumstances might you consider using the AGI pipeline design over the LUI pipeline?

- Sufficiently accurate branch prediction
- Multiple memory stages without increasing load-use delay

Q2.5: Address Generation Interlock (AGI) pipeline

2.5	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
addi	F	D	A	X	W																									
(1) add		F	D	A	X	W																								
lw			F	D	A	X	W																							
bne				F	D	A	X	W																						
-					F	D	A	-																						
-						F	D	-																						
-							F	-																						
(2) add								F	D	A	X	W																		
lw									F	D	A	X	W																	
bne										F	D	A	X	W																
-											F	D	A	-																
-												F	D	-																
-													F	-																
(3) add														F	D	A	X	W												
lw															F	D	A	X	W											
bne																F	D	A	X	W										
-																	F	D	A	-										
-																		F	D	-										
-																			F	-										
(4) add																				F	D	A	X	W						
lw																					F	D	A	X	W					
bne																						F	D	A	X	W				

3 Exceptions

3.1 Why are precise exceptions useful?

3.2 Why might one not want to always implement precise exceptions?

(a) Introduces hardware complexity

(b) Specific application does not need to or cannot recover from fatal exceptions