CMPE 110: Computer Architecture

Week 4
Pipelining III

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Jose Renau, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

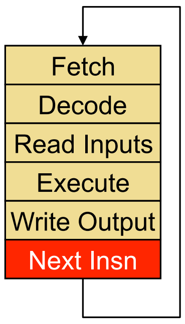
Reminder

- Quiz 1 posted
  - Due on Wednesday (Oct. 12) midnight on eCommons
  - Questions also posted on class Google website

Memory Addressing

- **Addressing mode:** way of specifying address
- Examples
  - **Displacement (based):** address = [R2+immed], e.g., #20(R2)
  - **Index-base:** address = [R2+R3]
  - **Memory-indirect:** address =[mem[R2]]
  - **Auto-increment:** address=[R2], R2= R2+1
  - **Auto-indexing:** address =[R2+immed], R2=R2+immed
  - **Scaled (scale-indexed):** address =[R2+R3*immed1+immed2]
  - **PC-relative:** address =[PC+imm]

Control Transfers



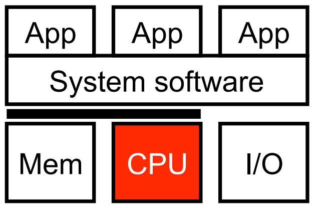
- Default next-PC = PC + sizeof(current insn)
  - Branches and jumps can change that
- **Computing targets**: where to jump to
  - For all branches and jumps
  - PC-relative: e.g., J offset for branches and jumps with function
  - Absolute: e.g., J L3 for function calls
  - Register indirect: e.g., JR R5 for returns, switches & dynamic calls

```
L3:
   addu  R7, R4, R3
   lw  R7, (R7)
   addu  R8, R5, R3
    J L3
   bne R3, R6, -4
```

- **Testing conditions**: whether to jump or not
  - Use registers & separate branch insns (MIPS)

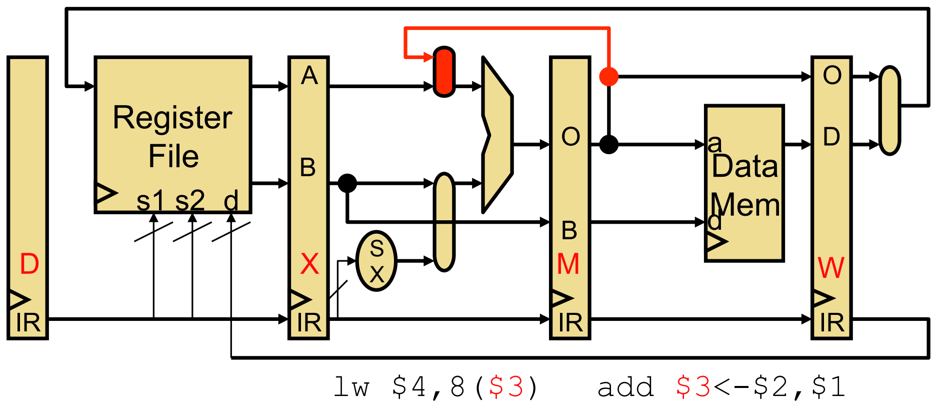
    `e.g., bnez R2,target`

Pipelining



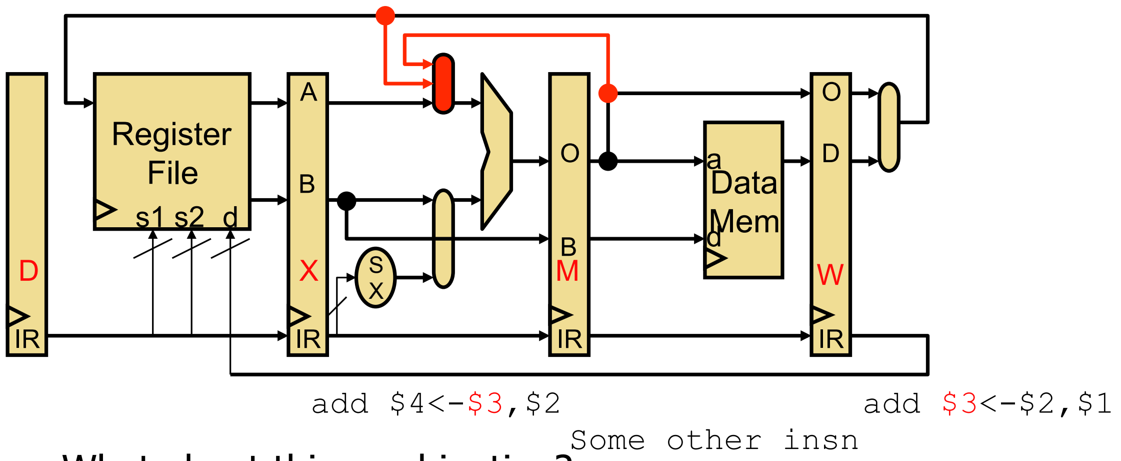
- Single-cycle datapaths vs. pipelined datapath
  - Basic pipelining: F, D, X, M, W
  - Base CPI = 1
  - Pipeline diagram (table)
- Data hazards
  - **Stalling**
  - Bypassing (forwarding)
- Structural hazards
  - **Stalling**
  - Add more hardware resources
- Multi-cycle operations
- Control hazards
  - Branch prediction

Review: Bypassing (forwarding)



- **Bypassing**
  - Reading a value from an intermediate (micro-architectural) source
  - Not waiting until it is available from primary source
  - Here, we are bypassing the register file
  - Also called **forwarding**
  - This example is an **MX** bypass

Review: Bypassing (forwarding)



- What about this combination?
  - Add another bypass path and MUX (multiplexor) input
  - This one is a **WX** bypass

Review: Pipeline Diagrams with Bypassing

- If bypass exists, "from"/"to" stages execute in same cycle
  - Example: MX bypass

	1	2	3	4	5	6	7	8	9	10
add r1<-r2,r3	F	D	X	**M**	W					
sub r2<-r1,r4		F	D	**X**	M	W				

  - Example: WX bypass

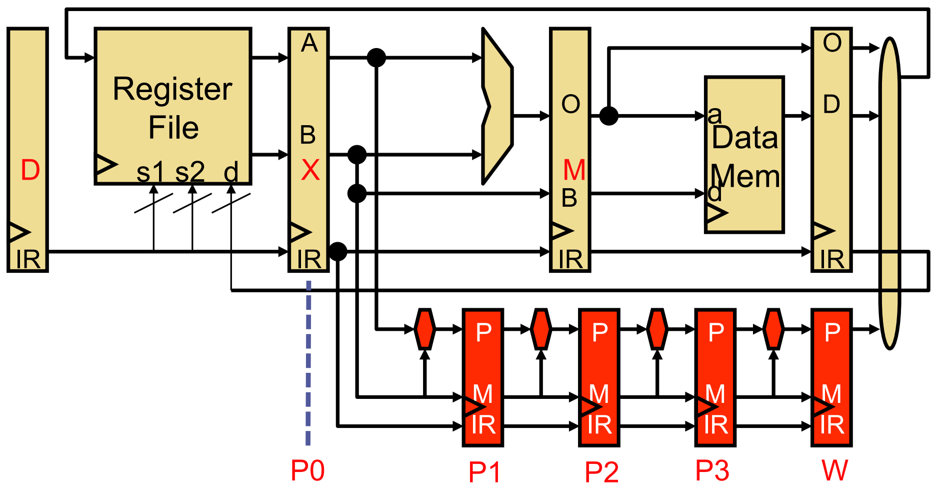
	1	2	3	4	5	6	7	8	9	10
add r1<-r2,r3	F	D	X	M	W					
ld r1,[r7+4]		F	D	X	M	**W**				
sub r2<-r1,r4			F	D	D	**X**	M	W		

  - Example: WM bypass

	1	2	3	4	5	6	7	8	9	10
add r1<-r2,r3	F	D	X	M	**W**					
sw r1,[r4+8]		F	D	X	**M**	W				

  - Can you think of a code example that uses the WM bypass?

Review: Multi-cycle operations



- Multiplier itself is often pipelined, what does this mean?
  - Product/multiplicand register/ALUs/latches replicated
  - Can start different multiply operations in consecutive cycles
  - **But still takes 4 cycles to generate output value**

Review: Structural Hazard

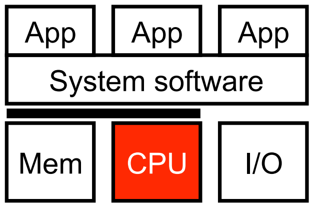
- What about…
  - Two instructions trying to write register file in same cycle?
  - Structural hazard!
- Must prevent:

	1	2	3	4	5	6	7	8	9
`mul $4<-$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $6<-$1,1`		F	D	X	M	W			
`add $5<-$6,$10`			F	D	X	M	**W**		

- Solution? stall the subsequent instruction

	1	2	3	4	5	6	7	8	9
`mul $4<-$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $6<-$1,1`		F	D	X	M	W			
`add $5<-$6,$10`			F	D	**d***	X	M	**W**	

Today



- Single-cycle datapaths vs. pipelined datapath
  - Basic pipelining: F, D, X, M, W
  - Base CPI = 1
  - Pipeline diagram (table)
- Data hazards
  - **Stalling**
  - Bypassing (forwarding)
- Structural hazards
  - **Stalling**
  - Add more hardware resources
- Multi-cycle operations
- Control hazards
  - Branch prediction

More Multiplier Nasties

- What about…
  - Mis-ordered writes to the same register
  - Software thinks **`add`** gets **`$4`** from **`addi`**, actually gets it from **`mul`**

	1	2	3	4	5	6	7	8	9
`mul $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $4,$1,1`		F	D	X	M	**W**			
…									
…									
`add $10,$4,$6`									

- Common? Not for a 4-cycle multiply with 5-stage pipeline
  - More common with deeper pipelines
  - In any case, must be correct

Corrected Pipeline Diagram

- With the correct stall logic
  - Prevent mis-ordered writes to the same register
  - Why two cycles of delay?

	1	2	3	4	5	6	7	8	9
`mul $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $4,$1,1`		F	D	**d***	**d***	X	M	**W**	
…									
…									
`add $10,$4,$6`									

- **Multi-cycle operations complicate pipeline logic**

Pipelined Functional Units

- Almost all multi-cycle functional units are pipelined
  - Each operation takes *N* cycles
  - But can start initiate a new (independent) operation every cycle
  - Requires internal latching and some hardware replication
  - + A cheaper way to improve throughput than multiple non-pipelined units

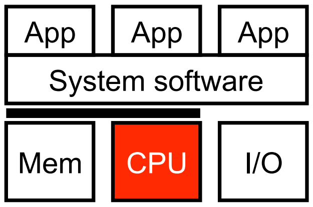
	1	2	3	4	5	6	7	8	9	10	11
`mulf f0,f1,f2`	F	D	E0	E1	E2	E3	W				
`mulf f3,f4,f5`		F	D	E0	E1	E2	E3	W			

- One exception: int/FP divide: difficult to pipeline and not worth it

	1	2	3	4	5	6	7	8	9	10	11
`divf f0,f1,f2`	F	D	E	E	E	E	W				
`divf f3,f4,f5`		F	D	**s***	**s***	**s***	E	E	E	E	W

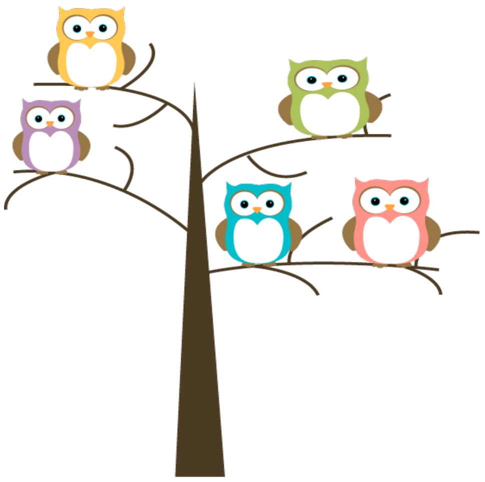
- **s*** = stall for structural hazard

Where are we?



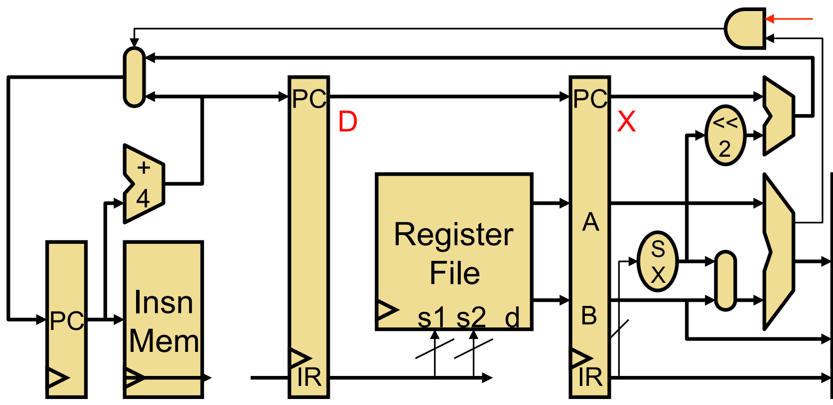
- Single-cycle datapaths vs. pipelined datapath
  - Basic pipelining: F, D, X, M, W
  - Base CPI = 1
  - Pipeline diagram (table)
- Data hazards
  - **Stalling**
  - Bypassing (forwarding)
- Structural hazards
  - **Stalling**
  - Add more hardware resources
- Multi-cycle operations
- Control hazards
  - Branch prediction

```
bnez r3,target
```



Control Dependences and Branch Prediction

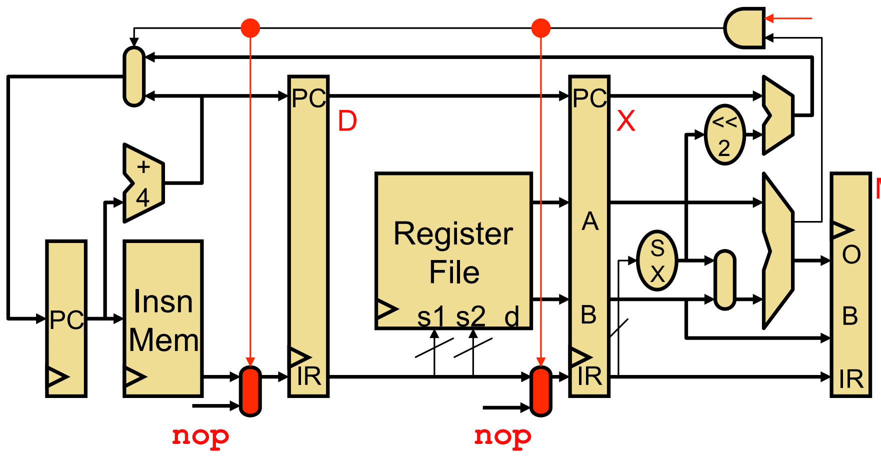
What About Branches?



```
bnez r3,target   F D X M W              F D X M W
The next inst    F F F D X M W            F D X M W
```

- Could just stall to wait for branch outcome (two-cycle penalty)
- **Branch speculation:** speculatively fetch some instruction before branch outcome is known
  - Default: assume "**not-taken**" (at fetch, can't tell it's a branch)

Branch Recovery



```
F D X M   W
~~F D X~~ M   W
      F   D
```

- Branch recovery: what to do when branch is actually taken
  - Insns that will be written into D and X are wrong
  - Flush them, i.e., replace them with **nops**
  + They haven't written permanent state yet (regfile, DMem)
  – Two cycle penalty for taken branches

Branch Speculation and Recovery

Speculation:

	1	2	3	4	5	6	7	8	9
`addi r1,1➔r3`	F	D	X	M	W				
`bnez r3,targ`		F	D	X	M	W			
`st r6➔[r7+4]`			**F**	**D**	X	M	W		
`mul r8,r9➔r10`				**F**	D	P0	P1	P2	…

speculative

- **Mis-speculation recovery**: what to do on wrong guess
  - Not too painful in a short, in-order pipeline
  - Branch resolves in X
  - + Younger insns (in F, D) haven't changed permanent state
  - **On next cycle, flush** insns in D and X **2 cycle overhead**

Recovery:

	1	2	3	4	5	6	7	8	9
`addi r1,1➔r3`	F	D	X	M	W				
`bnez r3,targ`		F	D	**X**	M	W			
~~`st r6➔[r7+4]`~~			**F**	**D**	--	--	--		
~~`mul r8,r9➔r10`~~				**F**	--	--	--	--	
`targ:add r4,r5➔r4`					**F**	D	X	M	W

Branch Performance

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - Speculation: not-taken
  - But actually, **75% of branches are taken**

- CPI = ? (assuming $CPI_{base} = 1$)
- CPI = 1 + 20% * 75% * 2 = 1.3
- **Branches cause 30% slowdown**
  - Worse with deeper pipelines (higher misprediction penalty)

- Can we do better than assuming branch is not taken?

Big Idea: Speculative Execution

- **Speculation**: “risky transactions on chance of profit”

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    - + Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    - – Must abort/flush/squash incorrect insns
    - – Must undo incorrect changes (recover pre-speculation state)

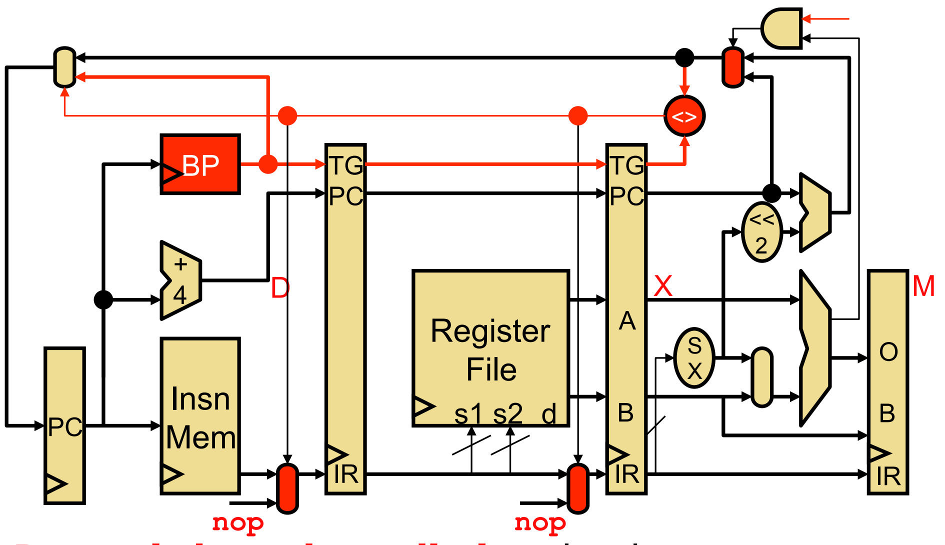
- **“Control speculation”**:
  - speculation aimed at control hazards

Control Speculation Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is PC+length_in_bytes (e.g., PC+4 for 32-bit instrucitons)
  - Can actively guess other targets: **dynamic branch prediction**

- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated insns
    - Hopefully haven’t modified permanent state (Regfile, DMem)
    - + Happens naturally in in-order 5-stage pipeline

Dynamic Branch Prediction



- **Dynamic branch prediction**: hardware guesses outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction**