National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 9: Pipelining II

CS10014 Computer Organization

Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

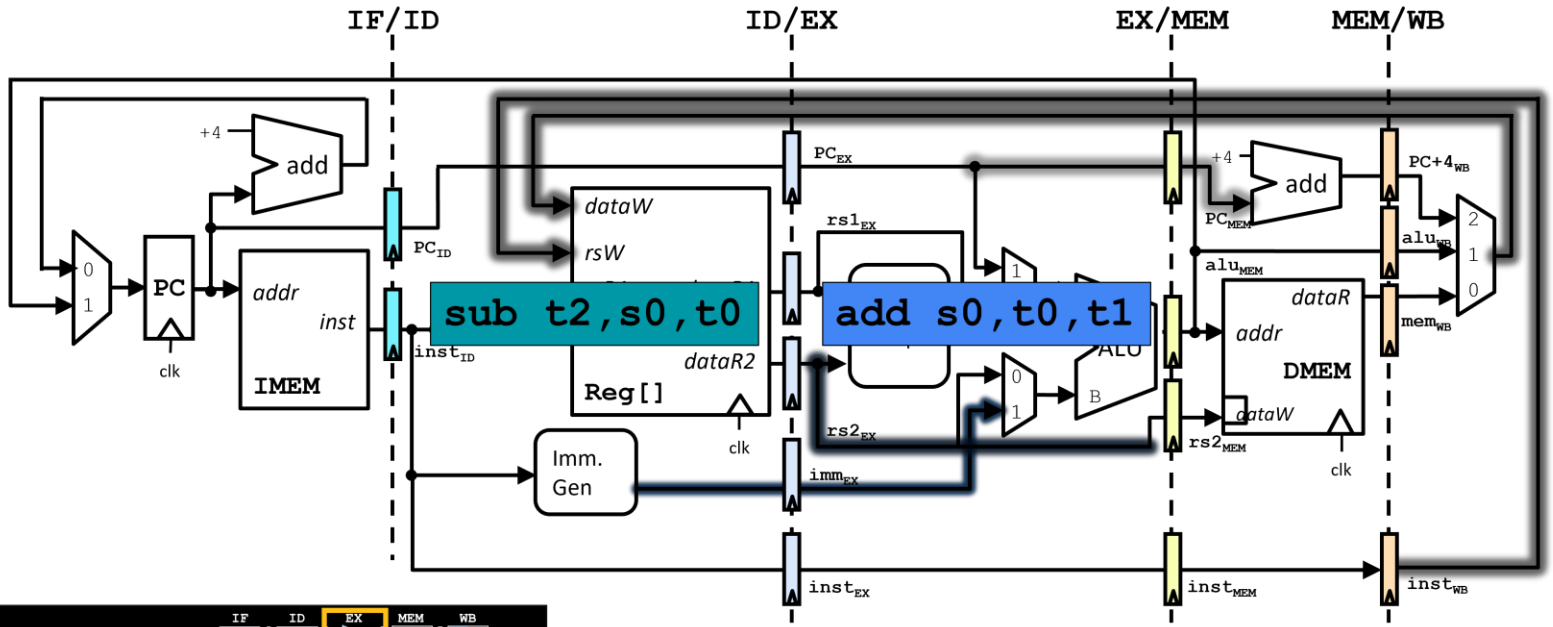
Outline

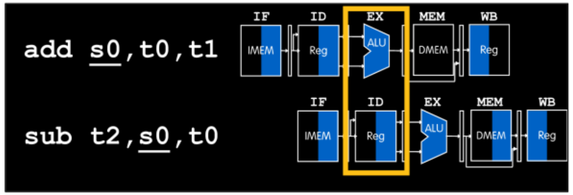
- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

Problems for Pipelining CPUs (1/2)

- Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazard:
    - Occurs when multiple instructions compete for access to a single physical resource
  - Data hazard:
    - Instructions have data dependency
    - Need to wait for previous instruction to complete its data read/write
  - Control hazard:
    - Flow of execution depends on previous instruction

Problems for Pipelining CPUs (2/2)





Pipeline registers allow multiple instructions to execute in separate stages.

Outline

- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

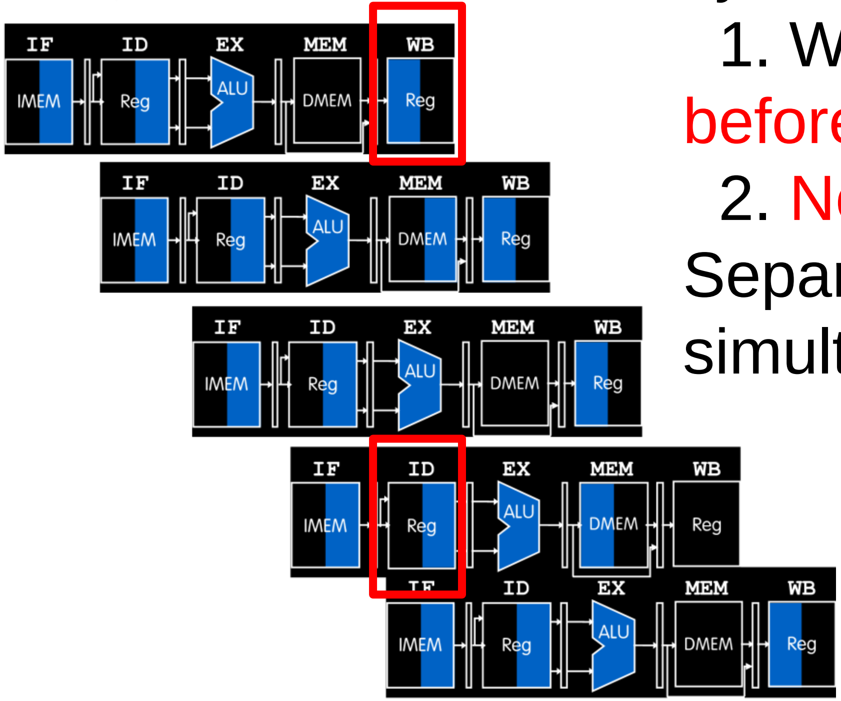
Data Hazard (1/10)

- **Data hazard**
  - Instructions have data dependency
  - Need to wait for previous instruction to complete its data read/write
  - Occurs when an instruction reads a register before a previous instruction has finished writing to that register
- Three cases to consider
  - Register access
  - ALU result
  - Load data hazard

Data Hazard: REG (1/2)

- **Register Access**

```
add t0,t1,t2
lw  t0,8(t3)
or  t3,t4,t5
sw  t0,4(t3)
sll t6,t0,t3
```



The same register is written and read in one cycle:

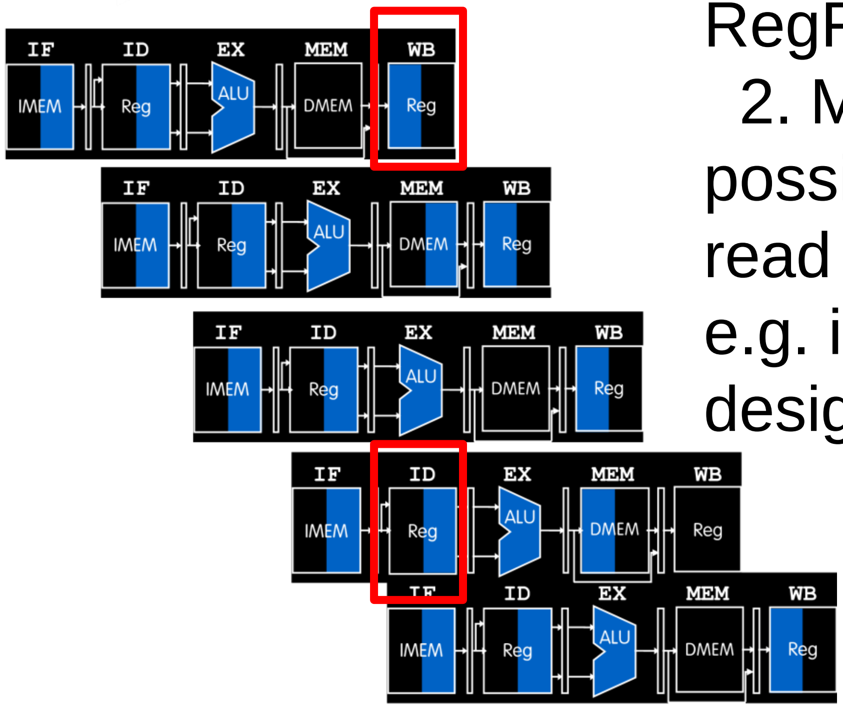
1. WB must write value before ID reads new value
2. No structural hazard – Separate ports allows simultaneous R/W

National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Data Hazard:REG (2/2)

- **Register Access**

```
add t0,t1,t2

lw t0,8(t3)

or t3,t4,t5

sw t0,4(t3)

sll t6,t0,t3
```



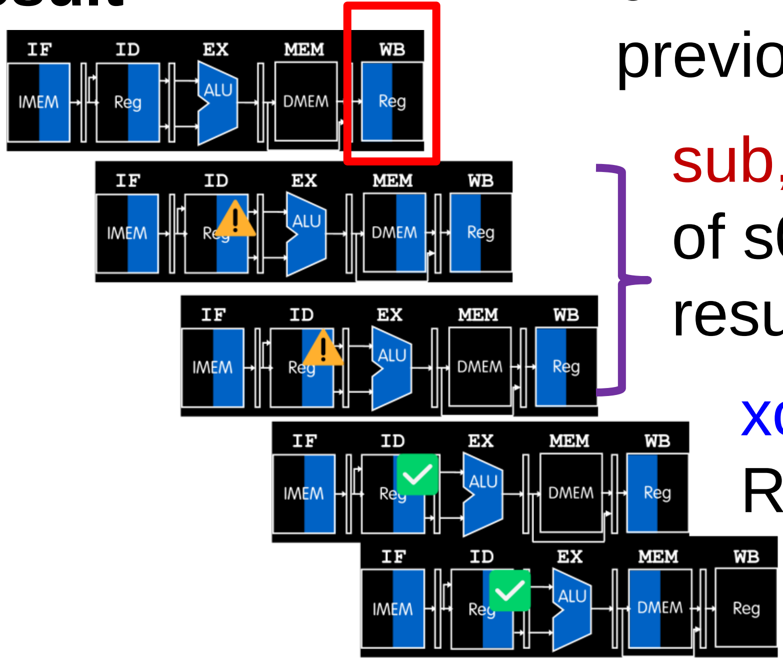
Solution: RegFile HW should write-then-read in the same cycle

1. Exploit high speed of RegFile (100 ps + 100 ps)
2. Might not always be possible to write-then-read in the same cycle., e.g. in high-frequency designs

Data Hazard: ALU (1/6)

- **ALU Result**

```
add s0,t0,t1
sub t2,s0,t0
or t6,s0,t3
xor t5,t1,s0
sw s0,4(t4)
```



s0 value	5	5	5	5	5/9	9	9	9	9

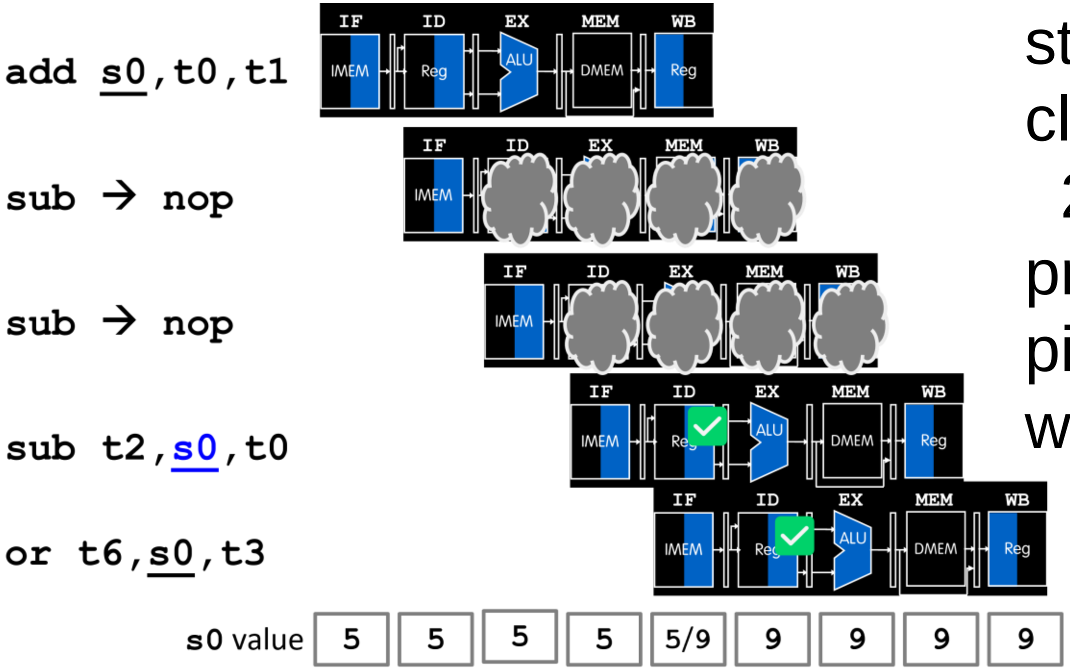
Problem: Instruction depends on WB's RegFile write from previous instruction

sub, or's ID reads old value of s0 and calculates wrong result

xor gets the right value; RegFile is write-then-read

Data Hazard: ALU(2/6)

- **ALU solution 1: Stalling**

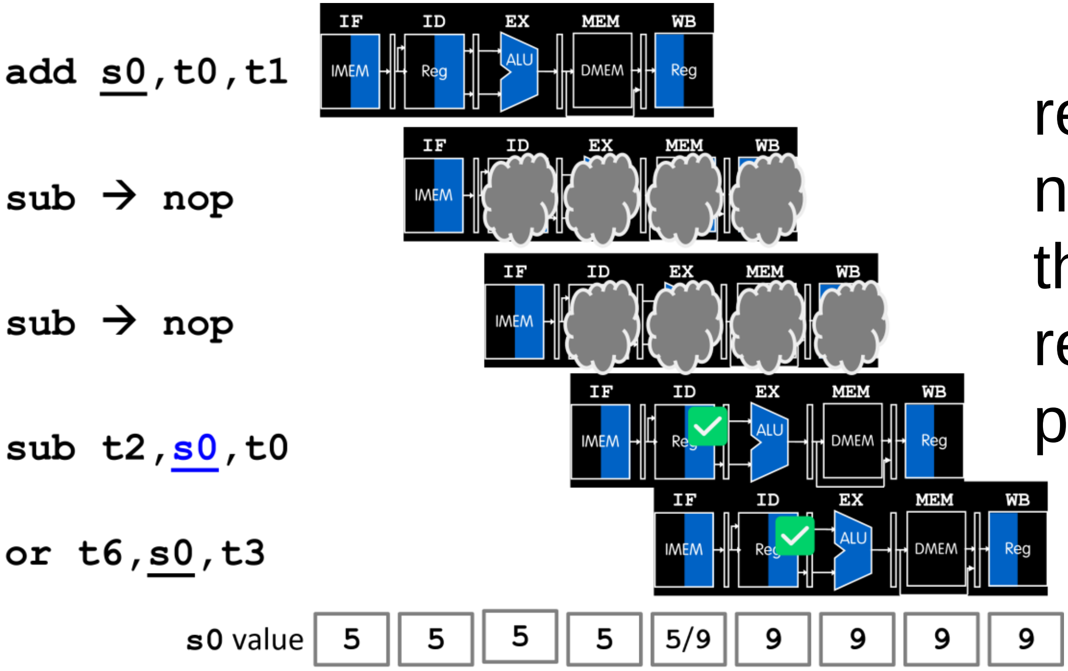


"Bubble" to effectively nop:

1. Affected pipeline stages do nothing during clock cycles
2. Stall all stages by preventing PC, IF/ID pipeline register from writing

Data Hazard: ALU (3/6)

- **ALU solution 1: Stalling**

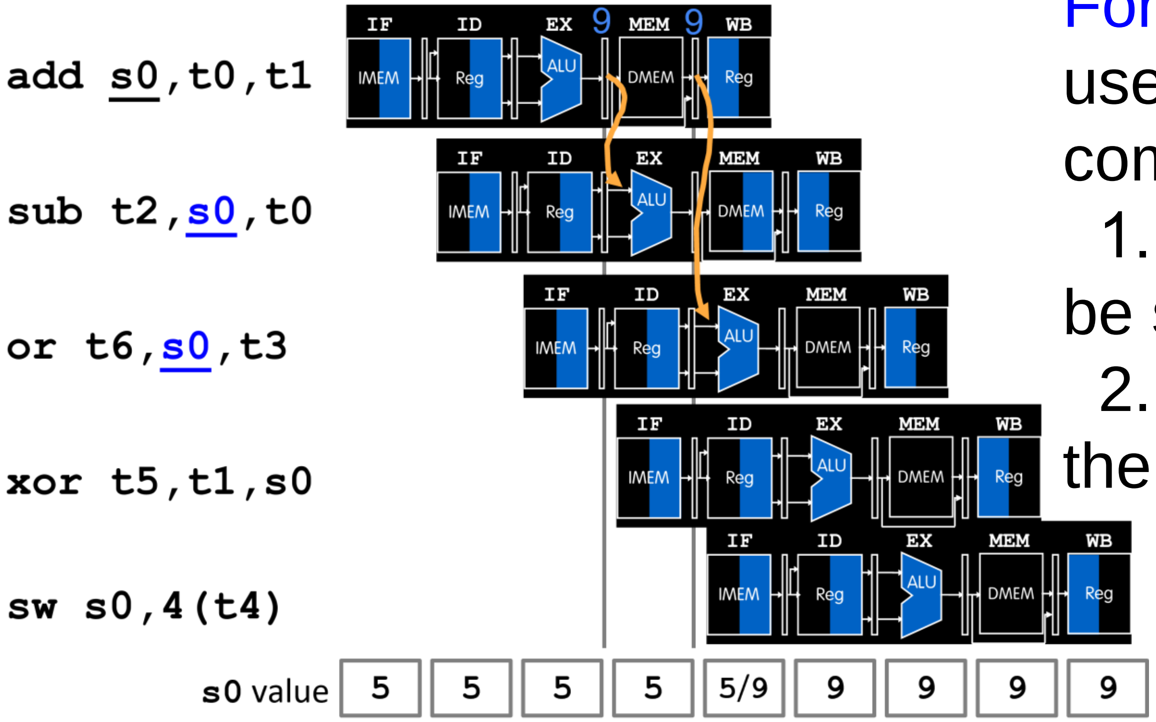


Stalls reduce performance

1. Compiler could rearrange code/insert nops to avoid hazard (and therefore stalls), but this requires knowledge of the pipeline structure

Data Hazard: ALU (4/6)

- **ALU solution 2: Forwarding**



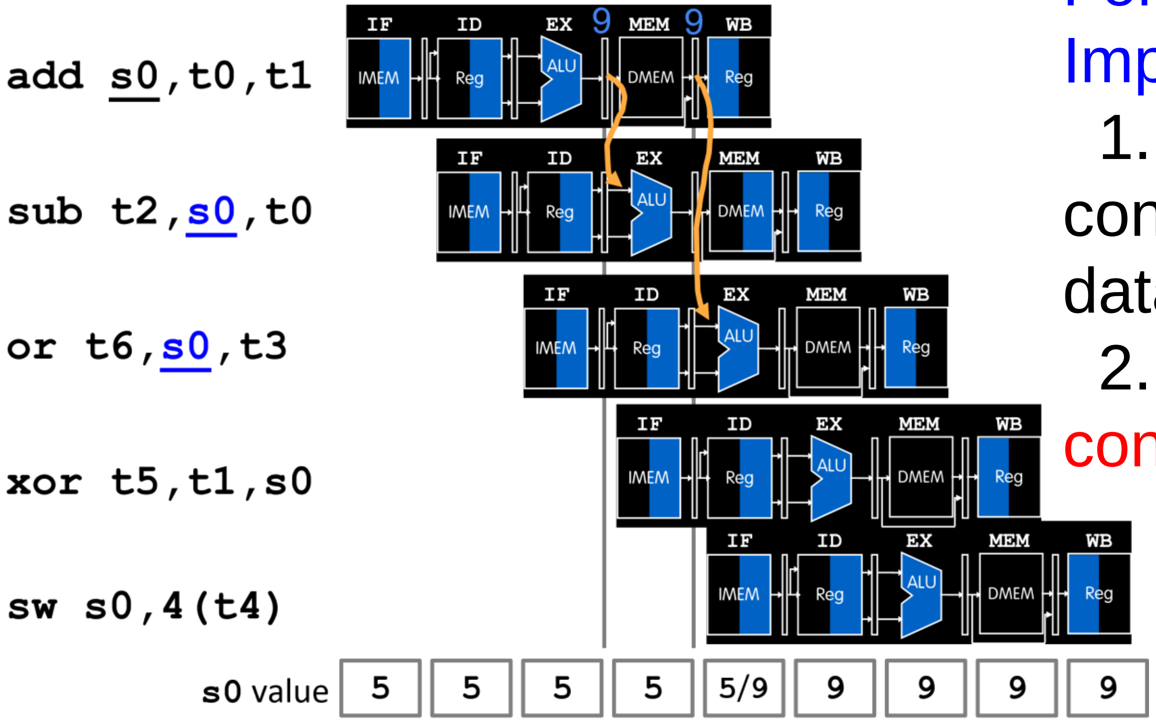
Forwarding (bypassing) uses the result when it is computed

1. Don't wait for value to be stored in RegFile
2. Grab operand from the pipeline stage

Data Hazard: ALU (5/6)

- **ALU solution 2: Forwarding**

```
add s0,t0,t1
sub t2,s0,t0
or t6,s0,t3
xor t5,t1,s0
sw s0,4(t4)
```

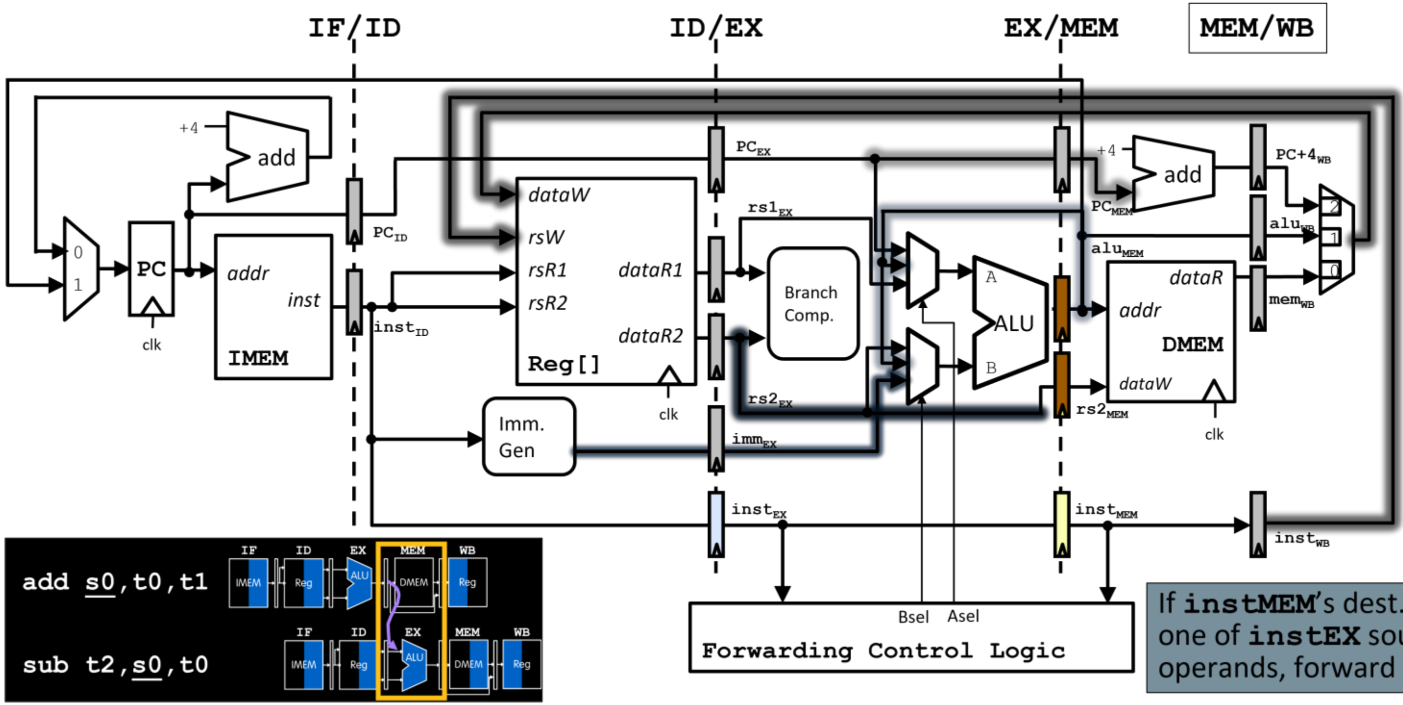


Forwarding (bypassing) Implementation:

1. Make extra connections in the datapath
2. Also add forwarding control logic

Data Hazard: ALU (6/6)

- **Forwarding EX output**



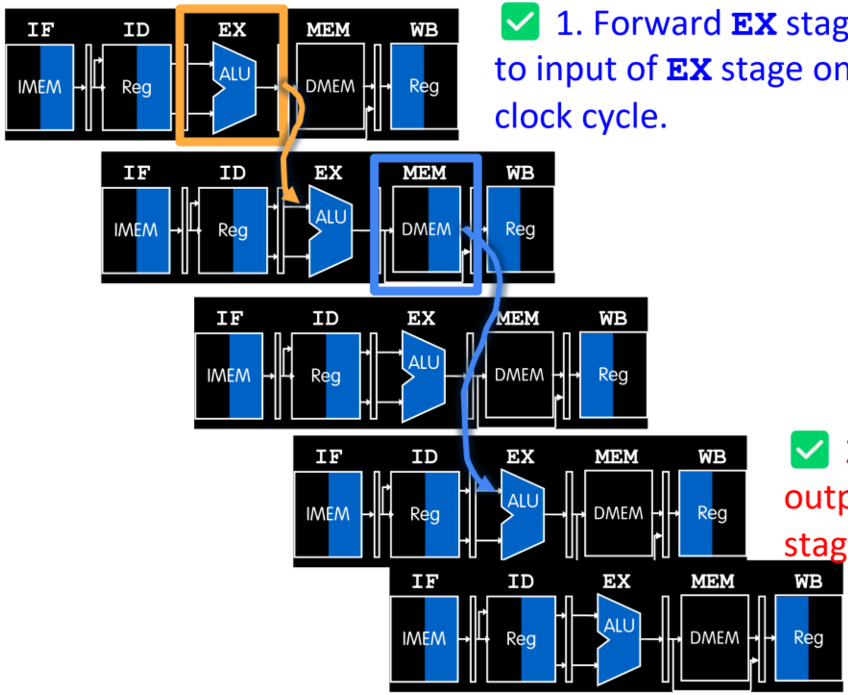
If **instMEM**'s dest. is one of **instEX** source operands, forward it.

Data Hazard: Load (1/8)

- **Forwarding cannot fix all data hazards**

time →

```
add s0,t1,t2

lw s1,8(s0)

or t3,s1,t1

and t4,s1,t2

sll t0,t1,t2
```



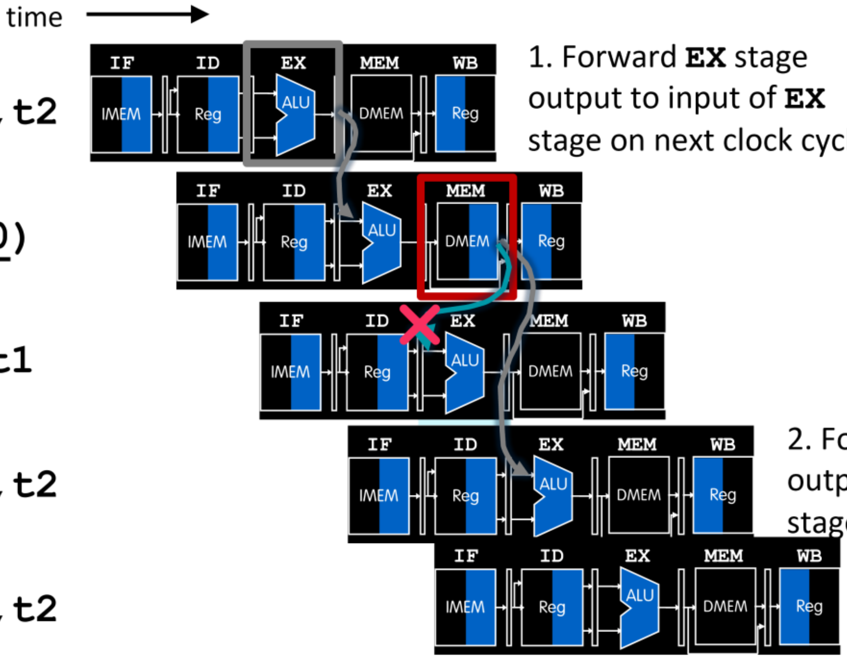
✓ 1. Forward **EX** stage output to input of **EX** stage on next clock cycle.

✓ 2. Forward **MEM** stage output to input of **EX** stage on next clock cycle.

Data Hazard: Load (2/8)

- **Forwarding cannot fix all data hazards**

```
add s0,t1,t2
lw s1,8(s0)
or t3,s1,t1
and t4,s1,t2
sll t0,t1,t2
```



1. Forward **EX** stage output to input of **EX** stage on next clock cycle.

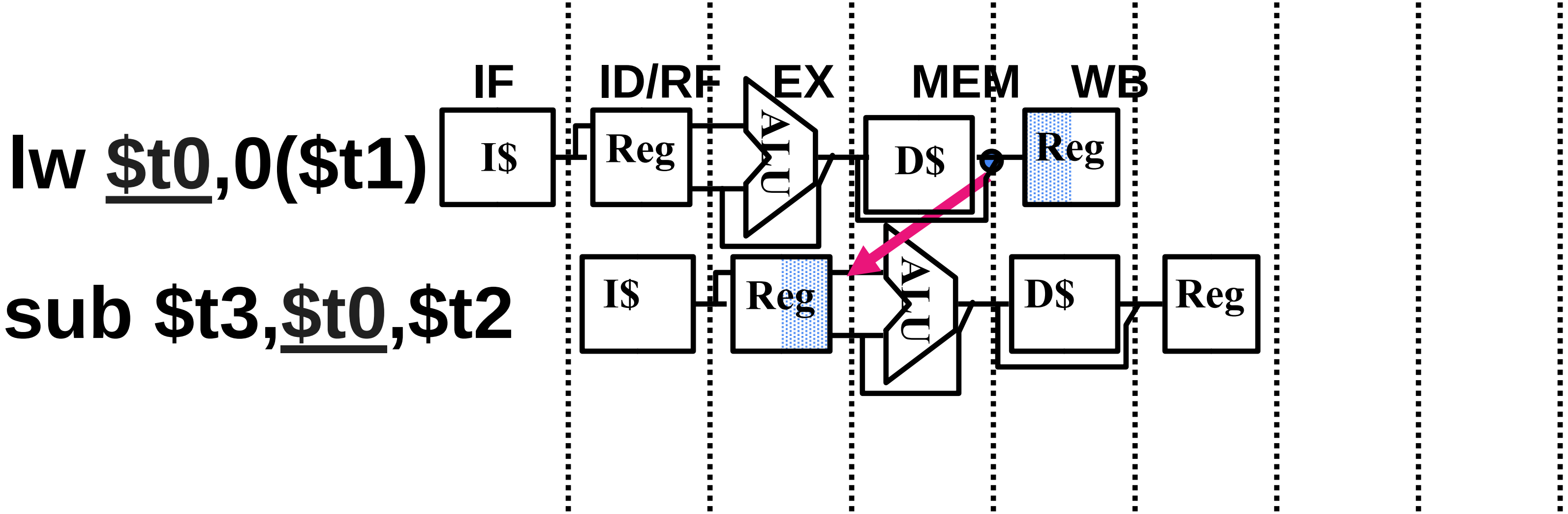
⚠ 3. **MEM** stage (**lw**)'s output needed as **EX** stage (**or**)'s input *in the same clock cycle.*

2. Forward **MEM** stage output to input of **EX** stage on next clock cycle.

Loads can result in an unavoidable pipeline stall.

Data Hazard: Load (3/8)

- **Forwarding cannot fix all data hazards**
  - Must stall instruction dependent on load, then forward (more hardware)



Data Hazard: Load (4/8)

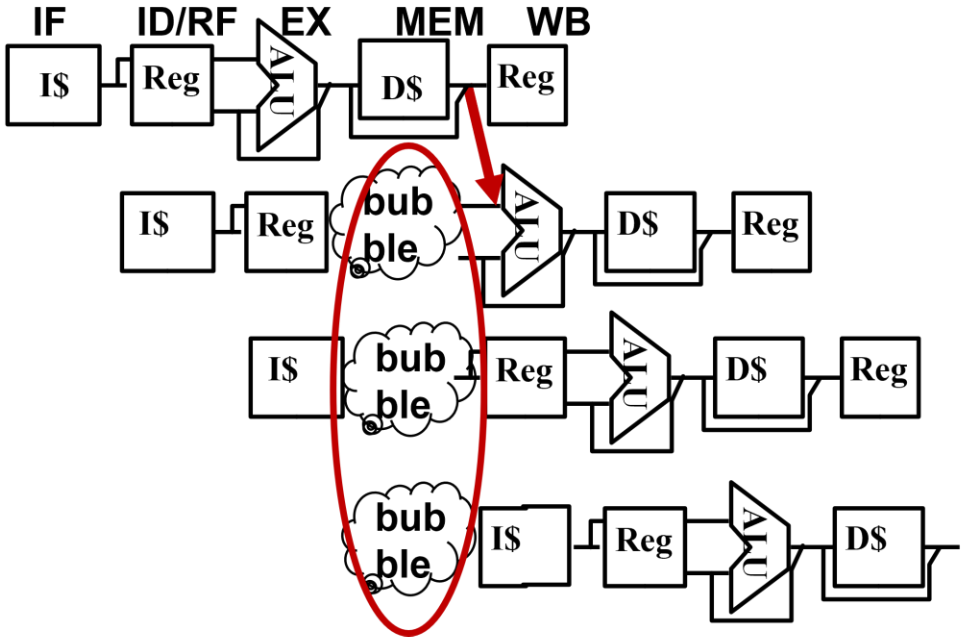
- **Hardware stalls pipeline**
  - Called “interlock”

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Data Hazard: Load (5/8)

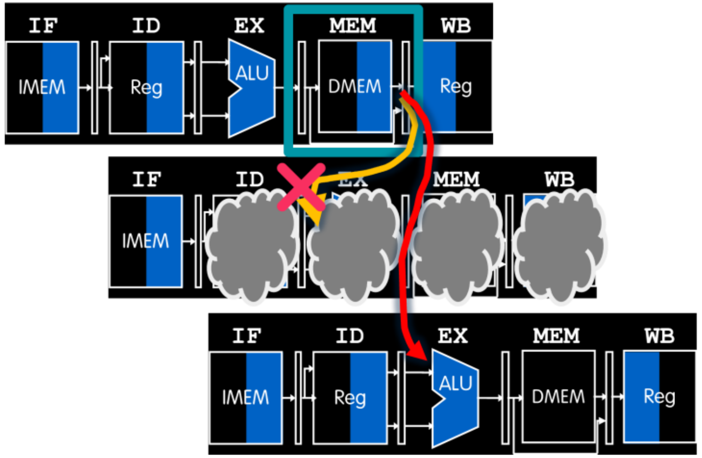
- The instruction slot after a load is called load delay slot
- If this instruction uses the result of load
  - The hardware must stall for one cycle (plus forwarding)
  - This results in performance loss!

```
lw s1,8(s0)
```

Load delay slot:

```
or → nop
```

```
or t3,s1,t1
```

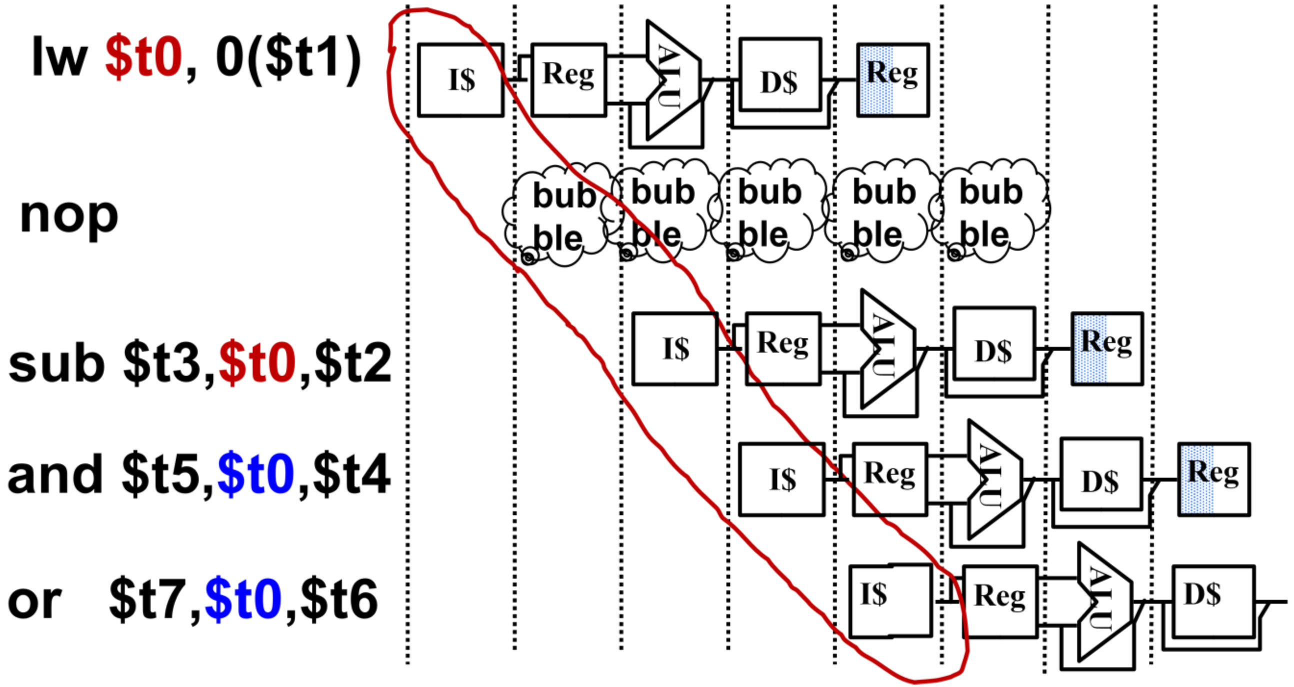


⚠ **MEM** stage (**lw**)'s output needed as **EX** stage (**or**)'s input *in the same clock cycle*.

Forwarding sends data to *the next clock cycle*. Cannot go backwards in time!

Data Hazard: Load (6/8)

- Stall is equivalent to “nop”



Data Hazard: Load (7/8)

- Code scheduling: Fix data hazard using the compiler
  - In the delay slot, put an instruction unrelated to the load result
  - -> No performance loss!

C Code

```
A[3] = A[0] + A[1];
A[4] = A[0] + A[2];
```

Code scheduling:
With knowledge of the underlying CPU pipeline, the compiler reorders code to improve performance.

⚠ Simple compilation
(9 cycles for 7 instructions)

```
lw    t1, 0(t0)
lw    t2, 4(t0)
add   t3, t1, t2
sw    t3, 12(t0)
lw    t4, 8(t0)
add   t5, t1, t4
sw    t5, 16(t0)
```

Stall & forward! (+1 cycle)

(+1 cycle)

✓ Alternative
(7 cycles):

```
lw    t1, 0(t0)
lw    t2, 4(t0)
lw    t4, 8(t0)
add   t3, t1, t2
sw    t3, 12(t0)
add   t5, t1, t4
sw    t5, 16(t0)
```

Forward!
(+0 cycle)

Data Hazard: Load (8/8)

- Instruction slot after a load is called "load delay slot"
- If the instruction uses the result of the "**LOAD**"
    - The hardware interlock will stall it for one cycle
- If the compiler puts an unrelated instruction in that slot
    - No stall
    - Letting the hardware stall the instruction in the delay slot is equivalent to putting a NOP in the slot

Takeaway Questions

- Assume a program executed in a processor
  - Branch: 20%, load: 20%, store: 10%, others: 50%
  - 50% of loads are followed by dependent instruction
    - Require 1 cycle stall (i.e. instruction of 1 nop)
- What is the CPI of such a program in this processor?

Takeaway Questions

- As before
  - Branch: 20%, load: 20%, store: 10%, others: 50%
- Hardware interlocks: same as software interlock
  - 20% of instructions require 1 cycle stall (i.e. insertion of 1 nop)
  - 5% of instructions require 2 cycle stall (i.e. insertion of 2 nops)
- What is the CPI?

Takeaway Questions

- As before
  - Branch: 20%, load: 20%, store: 10%, others: 50%
- Hardware interlocks: same as software interlock
  - 20% of instructions require 1 cycle stall (i.e. insertion of 1 nop)
  - 5% of instructions require 2 cycle stall (i.e. insertion of 2 nops)
- What is the CPI?
  - CPI = 1 + 0.2 * 1 + 0.05 * 2 = 1.3
  - In software, # instructions would increase 30%
  - In hardware, # instructions stays at 1, but CPI would increase 30%

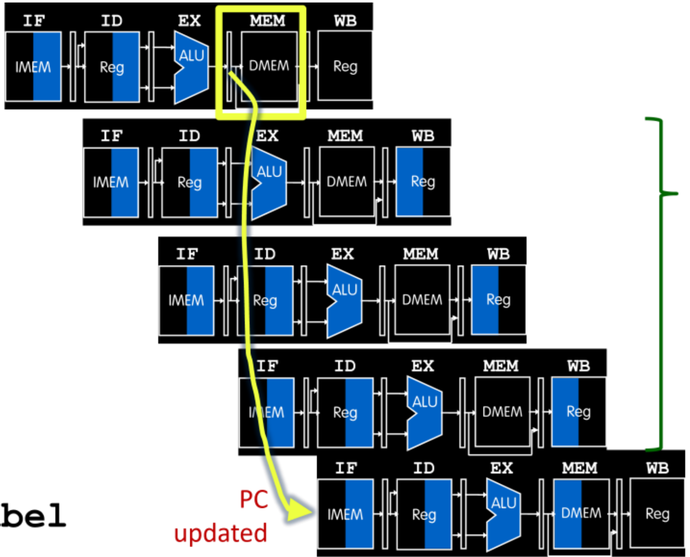
Outline

- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

Control Hazard (1/10)

- Control hazard (conditional branch) occurs when the instruction fetched may not be the one needed
  - For example, if the “beq” branch is **taken**

```
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x70 sw s0,8(t3) # Label
```



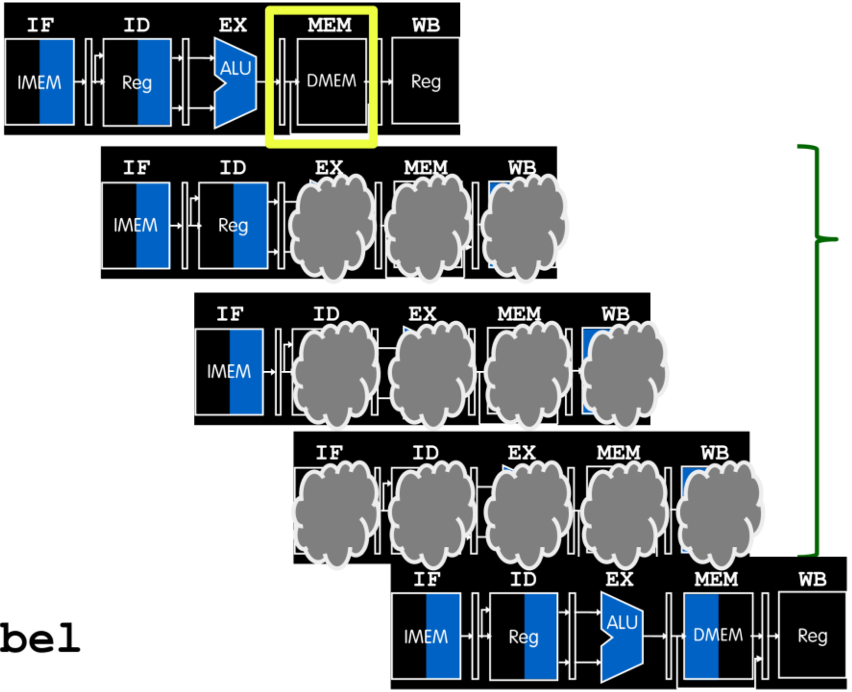
Instruction execution starts before branch outcome is known!

Correct instruction starts executing

Control Hazard (2/10)

- Kill instructions after branch (if taken)

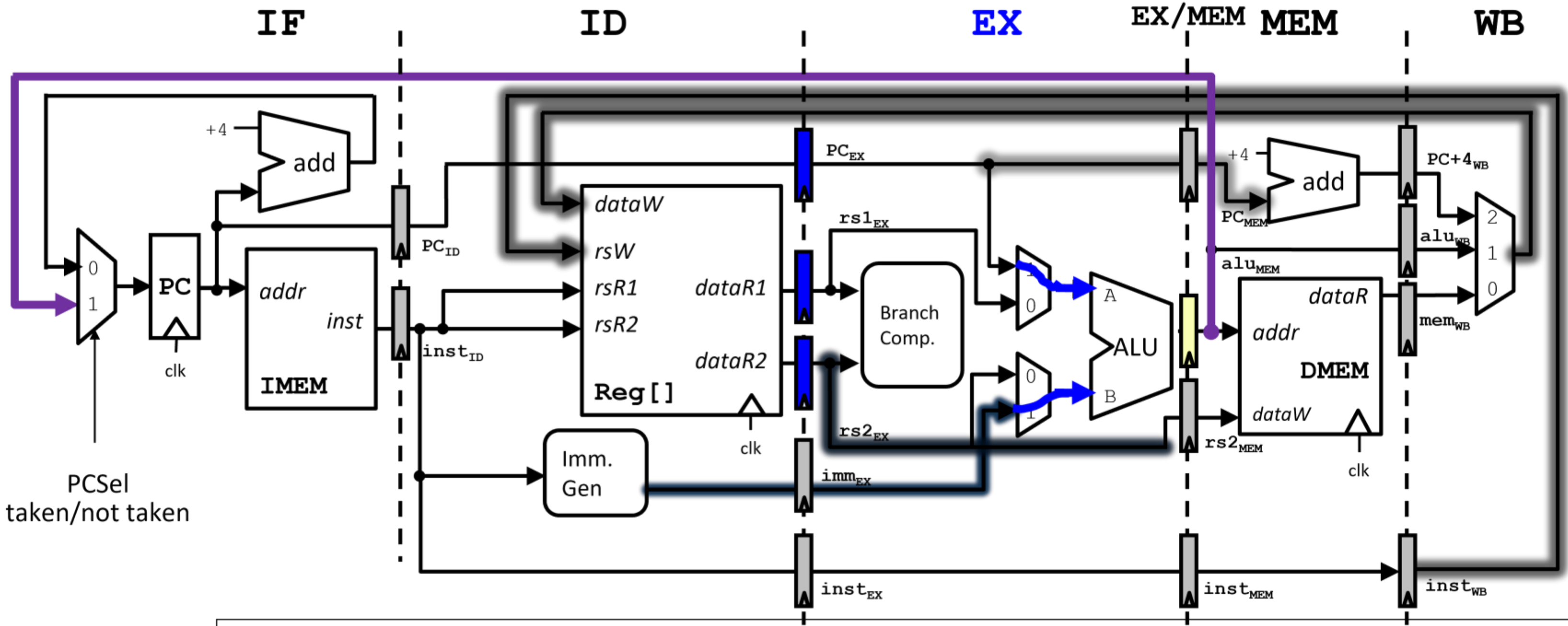
```
0x40 beq t0,t1,Label

0x44 sub t2,s0,t0

0x48 or t6,s0,t3

0x4c xor t5,t1,s0

0x70 sw s0,8(t3) # Label
```



Flush pipeline by converting incorrect instructions to nops.

PC updated, correct instruction loaded

Control Hazard (3/10)



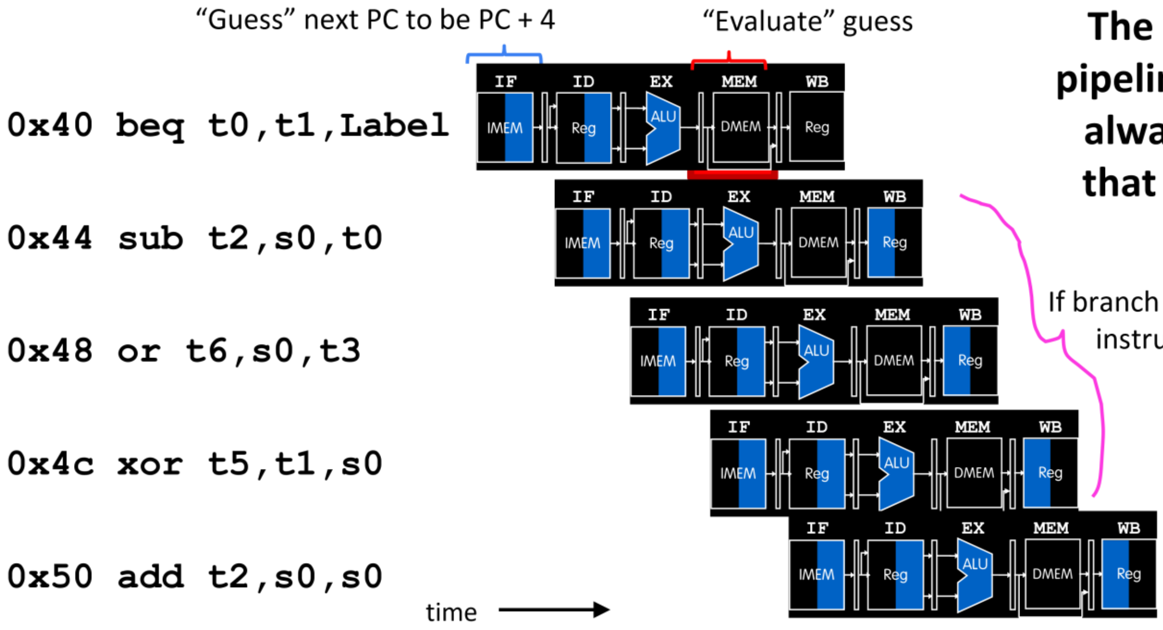
- In **MEM** stage: **EX/MEM** pipeline reg. feeds **IF** stage MUX. PCSel control is set.
- On the next clock cycle in the **IF** stage, PC updates, and the correct instruction is fetched.; fetches the correct instruction.

Control Hazard (4/10)

- **Branch prediction reduces penalties**
  - Every taken branch in the RV32I pipeline costs 3 clock cycles
  - Note if branch is not taken, then pipeline is not stalled
  - The correct instructions are correctly fetched sequentially after the branch instruction
- **We can improve the CPU performance on average through branch prediction**
  - Early in the pipeline, guess which way branches will go
  - Flush pipeline if branch prediction was incorrect

Control Hazard (5/10)

- Naïve predictor: Don't take branch



The simple RV32I pipeline effectively always "predicts" that branches are *not taken*.

If branch not taken, correct instructions are already executed.

Penalty for *incorrect* prediction is still 3 clock cycles! Branch prediction tries to improve *average performance*.

Control Hazard (6/10)

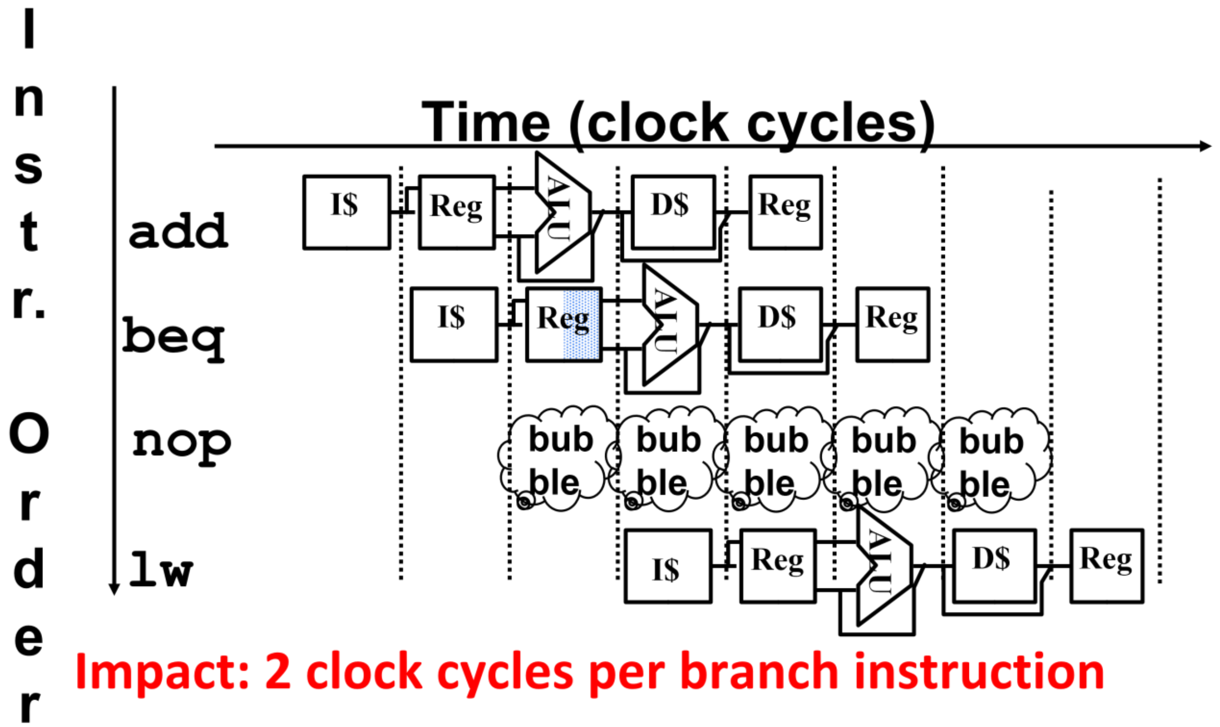
- We put branch decision-making hardware **in ALU stage**
  - Therefore, two more instructions after the branch will always be fetched, whether or not the branch is taken
- **Desired functionality of a branch**
  - If we do not take the branch, don’t waste any time and continue executing normally
  - If we take the branch, don’t execute any instructions after the branch, just go to the desired label

Control Hazard (7/10)

- **Initial Solution: Stall until decision is made**
    - Insert “no-op” instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles)
    - **Drawback**
        - Seems wasteful, particularly when the branch is not taken
        - Branches take 3 clock cycles each (assuming comparator is put in ALU stage)

Control Hazard (8/10)

- User inserting no-op instruction

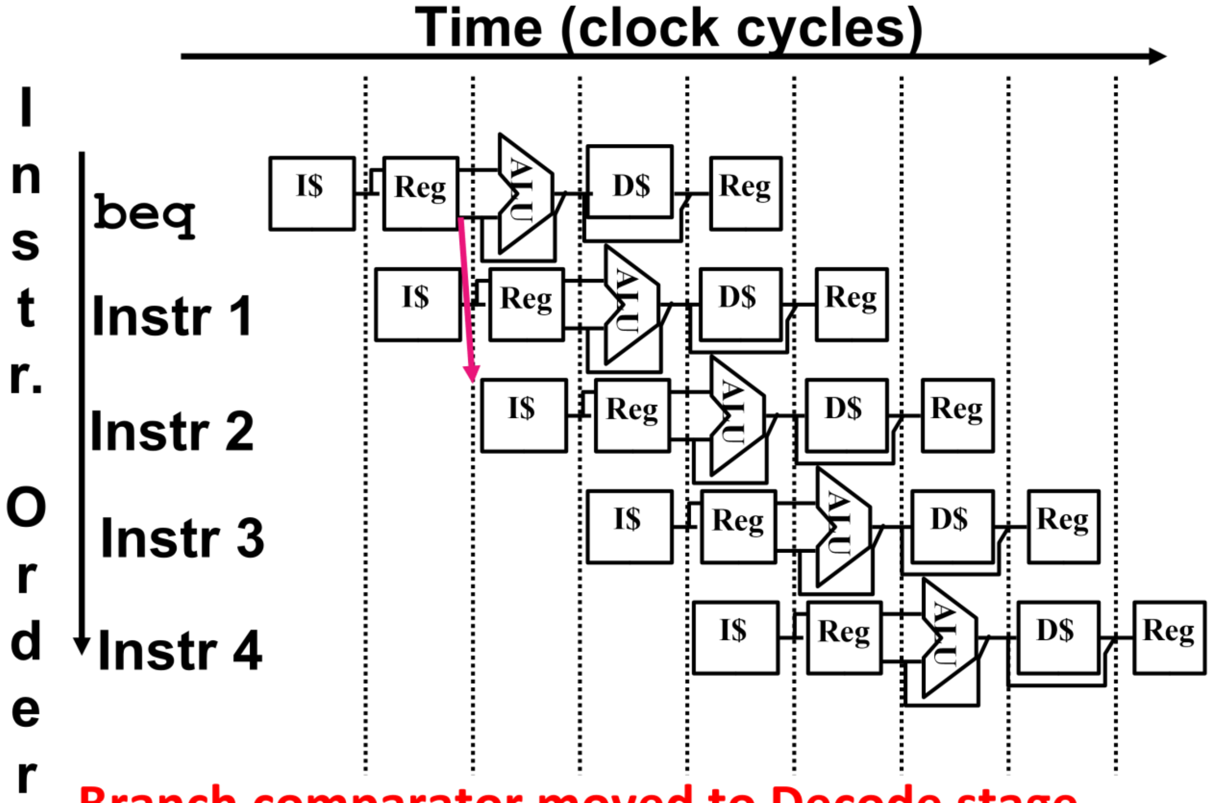


Impact: 2 clock cycles per branch instruction

Control Hazard (9/10)

- **Optimization #1**
    - Insert special branch comparator in Stage 2
    - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
    - Benefit
        - Since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is need

Control Hazard (10/10)



Branch comparator moved to Decode stage.

Outline

- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

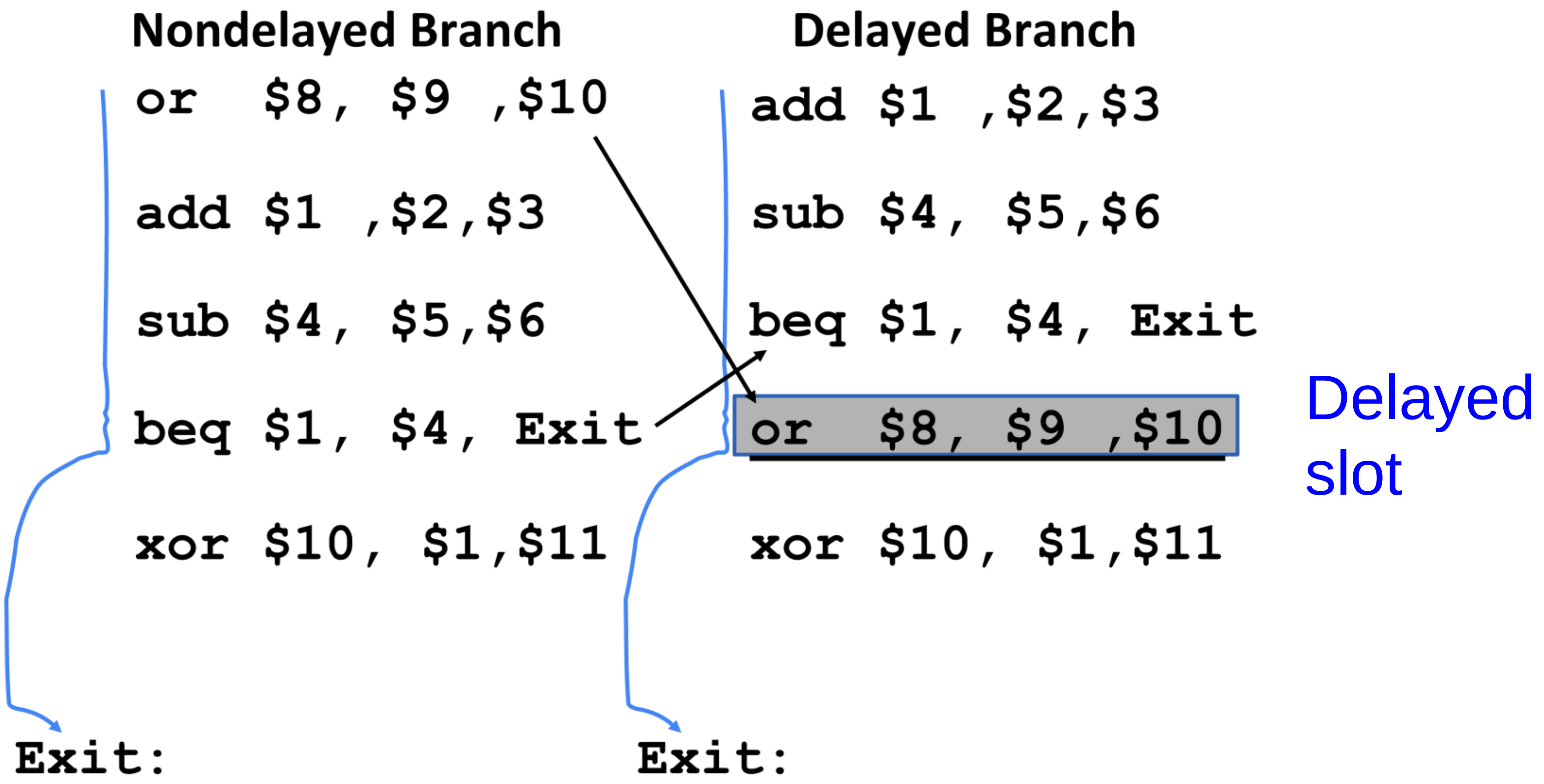
Delayed Branch Slot (1/3)

- **Optimization #2: Delayed Branch Slot**
  - **Old definition:**
    - if we take the branch, none of the instructions after the branch get execute by accident
  - **New definition:**
    - Whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)

Delayed Branch Slot (2/3)

- **Optimization #2: Delayed Branch Slot**
  - We always execution instruction after branch
  - Worst-case:
    - Can always put a no-op in the branch-delay slot
  - Better case:
    - Can find an instruction before the branch which can be placed in the branch-delay slot without affecting flow of the program
    - The compiler must be smart to find instructions to do this

Delayed Branch Slot (3/3)



Nondelayed Branch

```
or  $8, $9 ,$10

add $1 ,$2,$3

sub $4, $5,$6

beq $1, $4, Exit

xor $10, $1,$11

Exit:
```

Delayed Branch

```
add $1 ,$2,$3

sub $4, $5,$6

beq $1, $4, Exit

or  $8, $9 ,$10

xor $10, $1,$11

Exit:
```

Outline

- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

Branch Prediction (1/4)

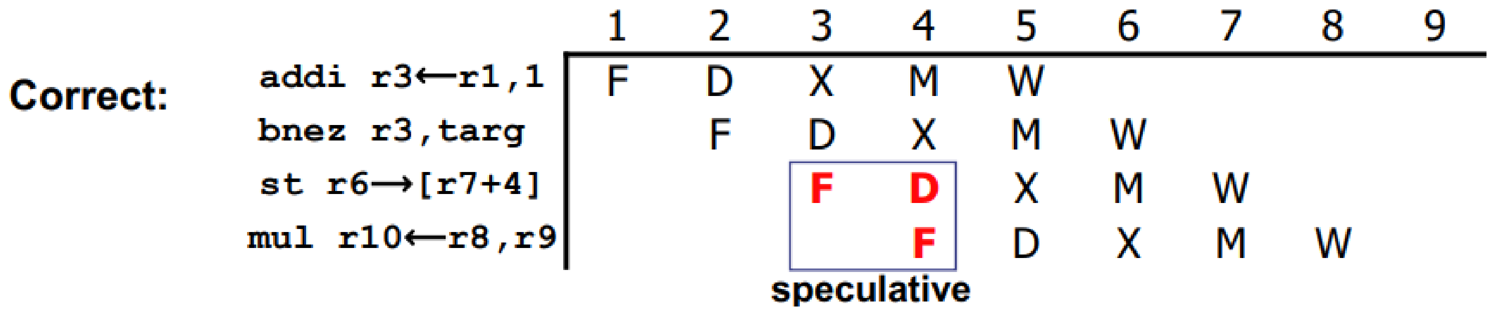
- **When to perform branch prediction?**
  - Option #1: During decode
    - Look at instruction opcode to determine branch instructions
    - Can calculate next PC from instruction (for PC-relative branches)
    - One cycle “mis-fetch” penalty even if branch predictor is correct
  - Option #2: During fetch?
    - How do we do that?
      - Branch predictor

Branch Prediction (2/4)

- **Speculative execution**
  - Execute before all parameters known with certainty
  - Correct speculation
    - Avoid stall, improve performance
  - Incorrect speculation (mis-speculation)
    - Must abort/flush/squash incorrect instructions
    - Must undo incorrect changes
  - Control speculation
    - Are these the correct instructions to execute next?

Branch Prediction (3/4)

- **Branch recovery**
  - What to do when branch is actually taken
    - Instruction that are in F and D 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



Correct:

	1	2	3	4	5	6	7	8	9
addi r3←r1,1	F	D	X	M	W				
bnez r3,targ		F	D	X	M	W			
st r6→[r7+4]			F	D	X	M	W		
mul r10←r8,r9				F	D	X	M	W	

speculative

Branch Prediction (4/4)

- **Mis-speculation recovery**
  - What to do on wrong guess
    - Branch resolves in X (EXEC.) stage
    - Younger insts (in F, D) haven't changed permanent state
    - Flush insts currently in D and X

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

Takeaway Questions

- Assume that
  - **Branch: 20%,** load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken
  - What is the CPI?

Takeaway Questions

- Assume that
  - **Branch: 20%,** load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken
  - What is the CPI?
    - CPI = 1 + 20% * 75% *2 = 1.3
    - Branches cause 30% slowdown
      - Worse with deeper pipelines, why?
      - Can we do better than assuming branch is not taken?

Takeaway Questions

- Assume that
  - **Branch: 20%,** load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken
  - Dynamic branch prediction
    - Branches predicted with 95% accuracy
    - What is the CPI?

Takeaway Questions

- Assume that
  - **Branch: 20%,** load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken
  - Dynamic branch prediction
    - Branches predicted with 95% accuracy
    - What is the CPI?
      - CPI = 1 + 20% * 5% * 2 = 1.02

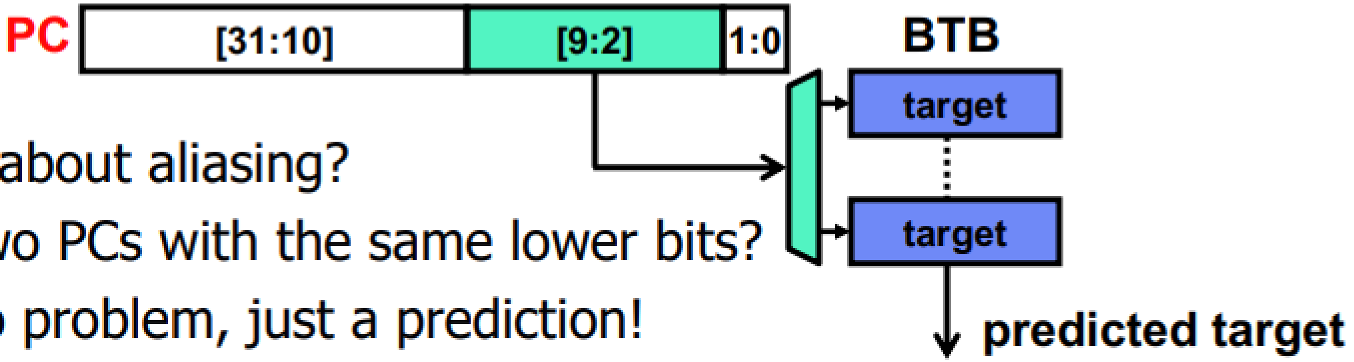
Outline

- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

Branch Target Buffer (1/6)

- **Learn from past, predict the future**
  - Record the past in a hardware structure
- **Branch target buffer (BTB)**
  - “guess” the future PC based on past behavior
  - Last time the branch X was taken, it went to address “Y”
  - So, in the future, if address X is fetched, fetch address Y next
  - PC indexes table of bits target addresses
  - Essentially: branch will go to the same place it went last time

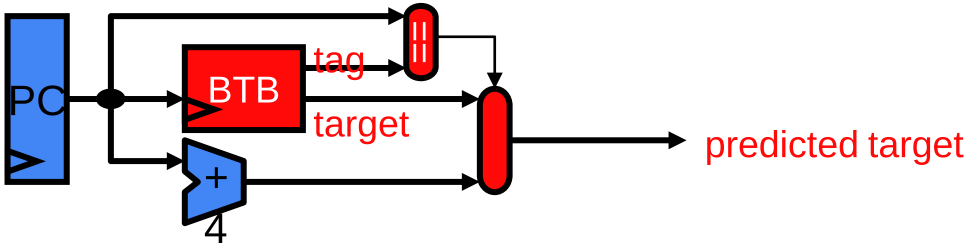
Branch Target Buffer (2/6)



- What about aliasing?
  - Two PCs with the same lower bits?
  - No problem, just a prediction!

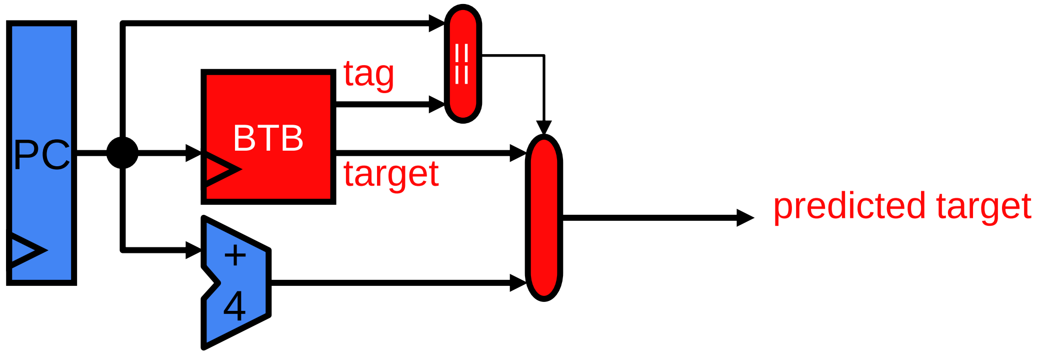
Branch Target Buffer (3/6)

- At fetch, how does inst know it's a branch & should read BTB?
  - All insts access BTB in parallel with instruction fetch
- **Key idea: use BTB to predict which insts are branches**
  - Implement by "tagging" each entry with its corresponding PC
  - Update BTB on every taken branch inst, record target PC
    - BTB[PC].tag = PC, BTB[PC].target = target of branch



Branch Target Buffer (4/6)

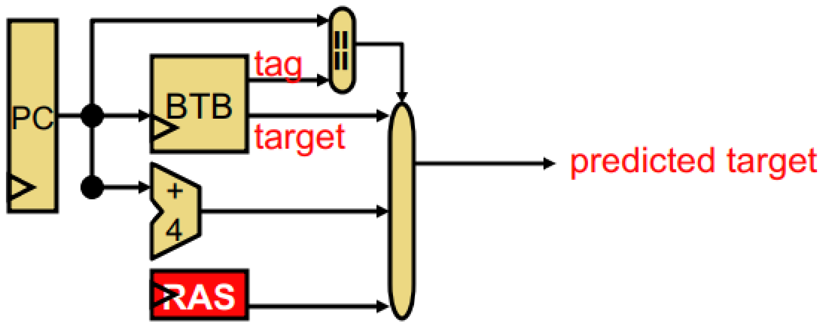
- All insts access at Fetch stage in parallel with Imem
  - Check for tag match, signifies inst at that PC is a branch
  - Predicted PC = (BTB[PC].tag == PC) ? BTB[PC].target: PC + 4



Branch Target Buffer (5/6)

- Why does a BTB work?
  - Because most control instructions use direct targets
  - Target encoded in inst itself -> same “taken” target every time
- What about indirect targets?
  - Target held in a register -> can be different each time
  - Two indirect calls
    - Dynamically linked functions (DLLs): target always the same
    - Dynamically dispatched (virtual) functions: hard but uncommon
  - Two indirect unconditional jumps
    - Switches, function returns

Branch Target Buffer (6/6)



- **Return Address Stack (RAS)**
  - Call instructions?
    - RAS[TopOfStack++] = PC + 4
  - Return instructions? Predicted-target = RAS[--TopOfStack]
  - Q: How can you tell if an inst is a call/return before decoding it?
    - Ans: another predictor (or put them in BTB marked as "return")
    - Or pre-decoded bits in inst memory, written when first executed

Outline

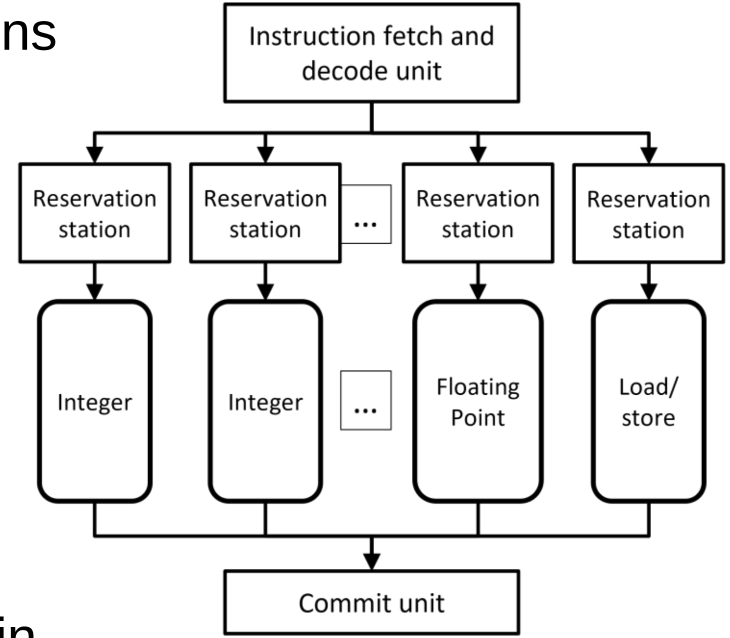
- Data Hazard
- Control Hazard
- Delay Branch Slot
- Branch Prediction
- Branch Target Buffer
- Superscalar processor

Superscalar Processors (1/2)

- **How to further increase processor performance?**
  - **Increase clock rate**
    - Limited by technology and power dissipation
  - **Increase pipeline depth**
    - “Overlap” instruction execution through deeper pipeline, e.g. 10 or 15 stages
      - Less work per stage -> shorter clock cycle/lower power
      - But more potential for all three types of hazards! (more stalling -> CPI > 1)
  - **Design a “superscalar” processor**

Superscalar Processors (2/2)

- **Superscalar processor**
  - Multiple-issue: start multiple instructions per clock cycle
    - Multiple execution units execute instructions in parallel
    - Each execution unit has its own pipeline
    - CPI < 1: multiple instructions completed per clock cycle
  - **Dynamic "out-of-order" execution**
    - Reorder instructions dynamically in HW to reduce impact of hazards



Conclusion

- Pipeline challenge is hazards
  - Forwarding helps with many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance
  - Superscalar
  - Out-of-order execution