

pipelining — branches

# last time

forwarding control logic

- look for register index indicating where value will be written
- compare register index to register actually read

forwarding paths

- textbook always forwards to end of decode
- can forward elsewhere, might be useful
- restriction: don't want to make critical path much longer

forward most recent version if multiple options

combining forwarding and stalling

# quiz Q1

	1	2	3	4	5	6						
addq %r8, %rcx	F	D	E	M1	M2	W	7					
mrmovq 8(%rcx), %rax		F	D	E	M1	M2	W	8	9	10		
subq %rcx, %rax			F	D*	D*	D	E	M1	M2	W		

# quiz Q2

	1	2	3	4	5	6	7								
addq %r8, %rcx	F	D	E1	E2	E3	M	W	8	9	10					
mrmovq 8(%rcx), %rax		F	D*	D*	D	E1	E2	E3	M	W	11	12	13	14	
subq %rcx, %rax			F*	F*	F	D*	D*	D*	D	E1	E2	E3	M	W	

# control hazard

```
subq %r8, %r9
```

```
je    0xFFFF
```

```
addq %r10, %r11
```

	fetch		fetch→decode		decode→execute			execute→writeback	
cycle	PC	SF/ZF	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0	0/1							
1	0x2	0/1	8	9					
2	???	0/1	0xF	0xF	800	900	9		

# control hazard

```
subq %r8, %r9
je    0xFFFF
addq %r10, %r11
```

	fetch		fetch→decode		decode→execute			execute→writeback	
cycle	PC	SF/ZF	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0	0/1							
1	0x2	0/1	8	9					
2	???	0/1	0xF	0xF	800	900	9		

0xFFFF if R[8] = R[9]; 0x12 otherwise

# control hazard: stall

```
addq %r8, %r9
```

```
// insert two nops
```

```
je    0xFFFF
```

```
addq %r10, %r11
```

cycle	fetch		fetch→decode		decode→execute			execute→writeback	
	PC	SF/ZF	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0	0/1							
1	0x2*	0/1	8	9					
2	0x2*	0/1	0xF	0xF	800	900	9		
3	0x2	0/0	0xF	0xF	---	---	0xF	1700	9
4	0x10	0/0	0xF	0xF	---	---	0xF	---	0xF
5			10	11	---	---	0xF	---	0xF
6					1000	1100	11	---	0xF

# control hazard: stall

```
addq %r8, %r9
```

```
// insert two nops
```

```
je 0xFFFF
```

```
addq %r10, %r11
```

	fetch	fetch→decode	decode→execute	execute→writeback					
cycle	PC	wait for two cycles for addq to update SF/ZF							
0	0x0	0/1							
1	0x2*	0/1	8	9					
2	0x2*	0/1	0xF	0xF	800	900	9		
3	0x2	0/0	0xF	0xF	---	---	0xF	1700	9
4	0x10	0/0	0xF	0xF	---	---	0xF	---	0xF
5			10	11	---	---	0xF	---	0xF
6					1000	1100	11	---	0xF



# control hazard: stall

```
addq %r8, %r9
```

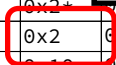
```
// insert two nops
```

```
je 0xFFFF
```

```
addq %r10, %r11
```

cycle	fetch		fetch→decode		decode→execute			execute→writeback	
	PC	SF/ZF	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0	0/1							
1	0x2*								
2	0x2*	0/1	0xF	0xF	800	900	9		
3	0x2	0/0	0xF	0xF	---	---	0xF	1700	9
4	0x10	0/0	0xF	0xF	---	---	0xF	---	0xF
5			10	11	---	---	0xF	---	0xF
6					1000	1100	11	---	0xF

execute je instruction (use SF/ZF)



# stalling for conditional jmps

```
subq %r8, %r8  
je label
```

```
label: irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

# stalling for conditional jmps

```
subq %r8, %r8  
je label
```

```
label: irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

# stalling for conditional jmps

```
subq %r8, %r8  
je label
```

```
label: irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

ZF sent via register

# stalling for conditional jmps

```
subq %r8, %r8  
je label
```

```
label: irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

“taken” sent from execute to fetch

# stalling for ret

```
call empty
addq %r8, %r9
```

```
empty:    ret
```

time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	call (store)	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq	nothing	nothing	nothing	ret

# stalling for ret

```
call empty
addq %r8, %r9
```

```
empty:    ret
```

time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	call (store)	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq	nothing	nothing	nothing	ret

return address stored here

# stalling for ret

```
call empty
addq %r8, %r9
```

```
empty:    ret
```

time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	return address loaded here	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq	nothing	nothing	nothing	ret



# stalling costs

with only stalling:

up to 3 extra cycles for data dependencies

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional `jmp`

## ex.: dependencies and hazards (2)

**mrmovq** 0(%rax) %rbx

**addq** %rbx %rcx

**jne** foo

foo: **addq** %rcx %rdx

**mrmovq** (%rdx) %rcx

where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

## making guesses

```
subq    %rcx, %rax  
jne     LABEL  
xorq    %r10, %r11  
xorq    %r12, %r13
```

...

```
LABEL:  addq    %r8, %r9  
        rmmovq %r10, 0(%r11)
```

speculate: **jne** will goto LABEL

right: 2 cycles faster!

wrong: forget before execute finishes

# when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

# when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

to “undo” instruction during fetch/decode:

forget everything in **pipeline registers**

# jXX: speculating right

```
subq %r8, %r8  
jne LABEL  
...
```

```
LABEL: addq %r8, %r9  
rmmovq %r10, 0(%r11)  
irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	irmovq	rmmovq	addq	jne (done)	OPq

# jXX: speculating right

```
subq %r8, %r8  
jne LABEL  
...
```

```
LABEL: addq %r8, %r9  
rmmovq %r10, 0(%r11)  
irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	j were waiting/nothing		
5	irmovq	rmmovq	addq	jne (done)	OPq

# jXX: speculating wrong

```
subq %r8, %r8
jne LABEL
xorq %r10, %r11
...
```

```
LABEL: addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq



# jXX: speculating wrong

```
subq %r8, %r8
jne LABEL
xorq %r10, %r11
...
```

```
LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	"squash" wrong guesses			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

# jXX: speculating wrong

```
subq %r8, %r8
jne LABEL
xorq %r10, %r11
...
```

```
LABEL: addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne			
4	rmmovq [?]	addq [?]	jne (use 2)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

fetch correct next instruction

# performance

hypothetical instruction mix

kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
ret	1%	4	4
others	91%	1*	1*

# performance

## hypothetical instruction mix

kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
ret	1%	4	4
others	91%	1*	1*

$$\text{predict: } 3 \times .03 + 1 \times .05 + 4 \times .01 + 1 \times .91 =$$

**1.09** cycles/instr.

$$\text{stall: } 3 \times .03 + 3 \times .05 + 4 \times .01 + 1 \times .91 =$$

**1.19** cycles/instr. (1.19 ÷  
1.09 ≈ 1.09x faster)

# exercise: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

cycle # 0 1 2 3 4 5 6 7 8 9 10

(1) **addq** %rcx, %r9

(2) **jne** foo (not taken)

(3) **subq** %rax, %r9

(4) **call** bar

(5) bar: **pushq** %r9

# [solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8	9	10
(1) <b>addq</b> %rcx, %r9		F	D	E	M							
(2) <b>jne</b> foo (not taken)			F	D	E							
(2b) foo: ... (mispred.)				F	D							
(2c) ... (mispred.)					F							
(3) <b>subq</b> %rax, %r9												
(4) <b>call</b> bar												
(5) bar: <b>pushq</b> %r9												

# [solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8	9	10
(1) <b>addq</b> %rcx, %r9		F	D	E	M							
(2) <b>jne</b> foo (not taken)			F	D	E							
(2b) foo: ... (mispred.)				F	D							
(2c) ... (mispred.)					F							
(3) <b>subq</b> %rax, %r9												
(4) <b>call</b> bar												
(5) bar: <b>pushq</b> %r9												

pass ZF/SF

# [solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) <b>addq</b> %rcx, %r9		F	D	E	M	W						
(2) <b>jne</b> foo (not taken)			F	D	E	M	W					
(2b) foo: ... (mispred.)				F	D	E	M	W				
(2c) ... (mispred.)					F	D	E	M	W			
(3) <b>subq</b> %rax, %r9												
(4) <b>call</b> bar												
(5) bar: <b>pushq</b> %r9												

pass ZF/SF

“squashed”



# [solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

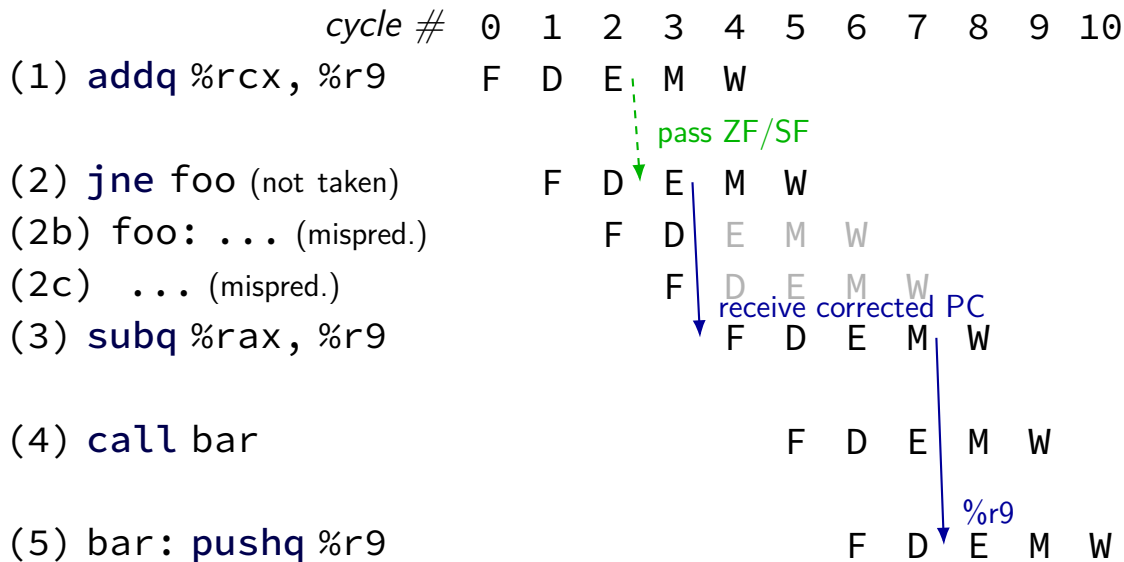
	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) <b>addq</b> %rcx, %r9		F	D	E	M	W						
(2) <b>jne</b> foo (not taken)			F	D	E	M	W					
(2b) foo: ... (mispred.)				F	D	E	M	W				
(2c) ... (mispred.)					F	D	E	M	W			
(3) <b>subq</b> %rax, %r9						F						
(4) <b>call</b> bar												
(5) bar: <b>pushq</b> %r9												

Annotations:

- A green dashed arrow points from the 'E' stage of instruction (1) at cycle 2 to the 'E' stage of instruction (2) at cycle 3, labeled "pass ZF/SF".
- A blue solid arrow points from the 'E' stage of instruction (2) at cycle 3 down to the 'F' stage of instruction (3) at cycle 4, labeled "receive corrected PC".

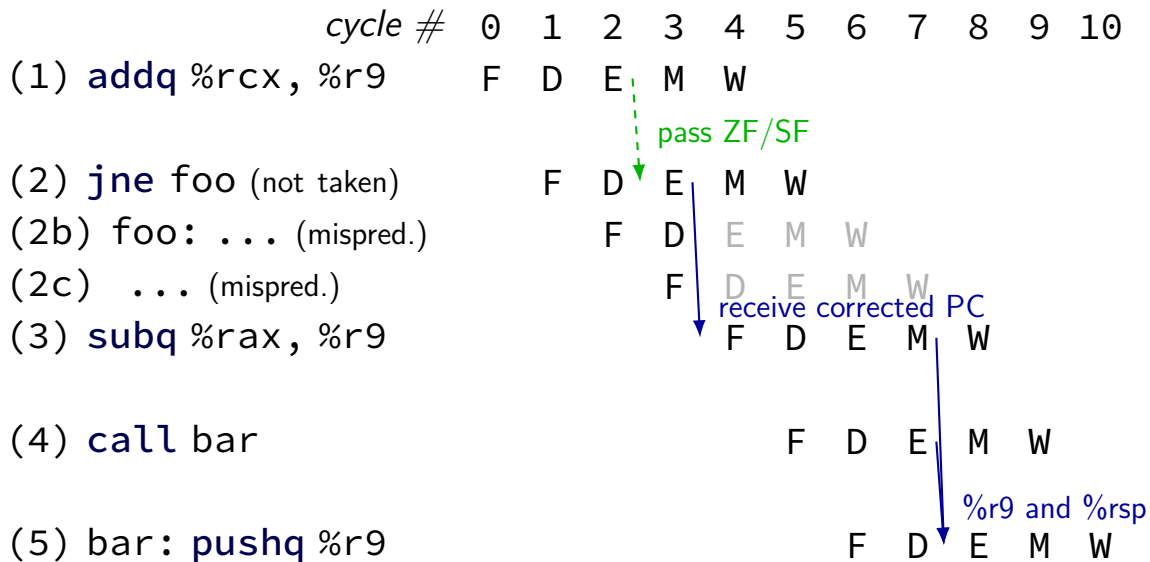
# [solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?



# [solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?



# exercise: with different pipeline

with F/D/E1/E2/M/W

cycle # 0 1 2 3 4 5 6 7 8 9 10

(1) **addq** %rcx, %r9

(2) **jne** foo (not taken)

(3) **subq** %rax, %r9

(4) **call** bar

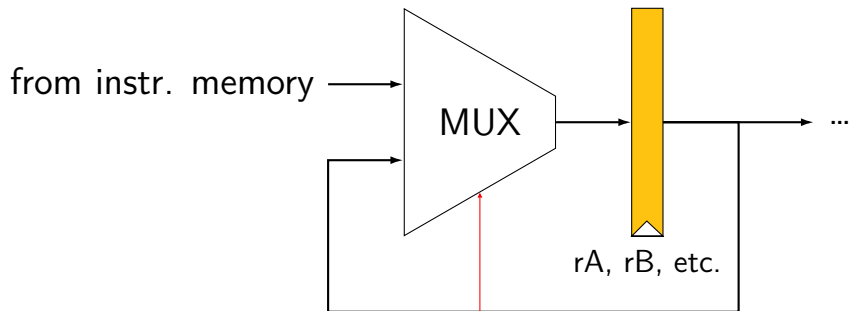
(5) bar: **pushq** %r9

# [solution]: with different pipeline

with F/D/E1/E2/M/W

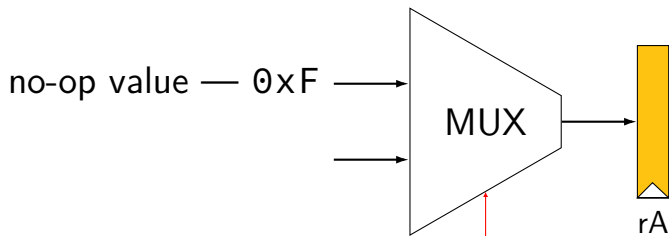
	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) <b>addq</b> %rcx, %r9		F	D	E1	E2	M	W					
(2) <b>jne</b> foo (not taken)			F	D	E1	E2	M	W				
(2b) mispredicted				F	D	E1	E2	M	W			
(2c) mispredicted					F	D	E1	E2	M	W		
(2d) mispredicted						F	D	E1	E2	M	W	
(3) <b>subq</b> %rax, %r9						F	D	E1	E2	M	W	
(4) <b>call</b> bar							F	D	E1	E2	M	W
(5) bar: <b>pushq</b> %r9								F	D*	D	E1	E2

# fetch/decode logic — advance or not



should we stall?

# fetch/decode logic — bubble or not



should we send  
no-op value (“bubble”)?

# HCLRS signals

```
register aB {  
    ...  
}
```

HCLRS: every register bank has these MUXes built-in

`stall_B`: keep **old value** for all registers

register input  $\leftarrow$  register output

pipeline: keep same instruction in this stage next cycle

`bubble_B`: use **default value** for all registers

register input  $\leftarrow$  default value

pipeline: put no-operation in this stage next cycle



# exercise

```
register aB {  
    value : 8 = 0xFF;  
}  
...
```

stall: keep old value bubble: store default value
--

time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	???	1	0
2	0x03	???	0	0
3	0x04	???	0	1
4	0x05	???	0	0
5	0x06	???	0	0
6	0x07	???	1	0
7	0x08	???	1	0
8		???		

# exercise result

```
register aB {  
    value : 8 = 0xFF;  
}
```

...

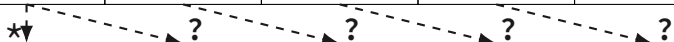
time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	0x01	1	0
2	0x03	0x01	0	0
3	0x04	0x03	0	1
4	0x05	0xFF	0	0
5	0x06	0x05	0	0
6	0x07	0x06	1	0
7	0x08	0x06	1	0
8		0x06		

# exercise: squash + stall (1)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

1	E	D	C	B	A
---	---	---	---	---	---

2	E	nop	C	nop	B
---	---	-----	---	-----	---



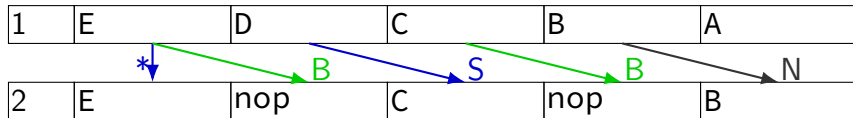
stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

exercise: what are the ?s  
write down your answers,  
then compare with your neighbors

# exercise: squash + stall (1)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

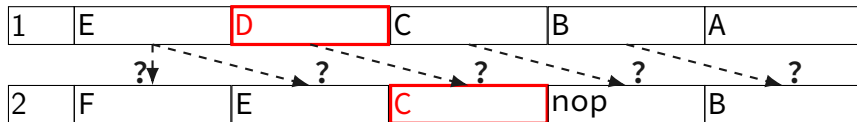


stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

## exercise: squash + stall (2)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

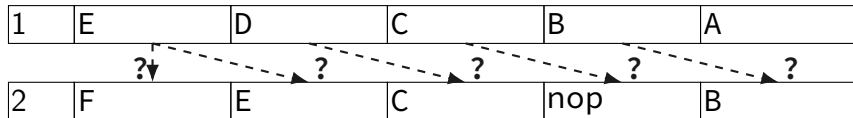


stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

## exercise: squash + stall (2)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

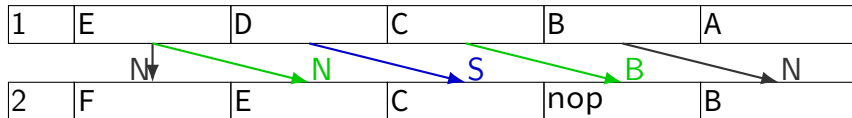


stall (S) = keep old value; normal (N) = use new value  
bubble (B) = use default (no-op);

exercise: what are the ?s  
write down your answers,  
then compare with your neighbors

## exercise: squash + stall (2)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

# implementing stalling + prediction

need to handle updating PC:

- stalling: retry same PC

- prediction: use predicted PC

- misprediction: correct mispredicted PC

need to updating pipeline registers:

- repeat stage in stall: keep same values

- don't go to next stage in stall: insert nop values

- ignore instructions from misprediction: insert nop values



## stalling: bubbles + stall

	cycle #								
	0	1	2	3	4	5	6	7	8
<code>mrmovq 0(%rax), %rbx</code>	F	D	E	M	W				
<code>subq %rbx, %rcx</code>		F	D	D	E	M	W		
inserted nop				E	M	W			
<code>irmovq \$10, %rbx</code>			F	F	D	E	M	W	
...									

need way to keep pipeline register unchanged to repeat a stage

(and to replace instruction with a nop)

## stalling: bubbles + stall

	cycle #									
	0	1	2	3	4	5	6	7	8	
<code>mrmovq 0(%rax), %rbx</code>	F	D	E	M	W					
<code>subq %rbx, %rcx</code>		F	D	D	E	M	W			
inserted nop				E	M	W				
<code>irmovq \$10, %rbx</code>			F	F	D	E	M	W		
...										

keep same instruction in cycle 3  
during cycle 2:  
`stall_D = 1`  
`stall_F = 1` or extra `f_pc MUX`

need way to keep pipeline register unchanged to repeat a stage

(and to replace instruction with a nop)

## stalling: bubbles + stall

	cycle #								
	0	1	2	3	4	5	6	7	8
<code>mrmovq 0(%rax), %rbx</code>	F	D	E	M	W				
<code>subq %rbx, %rcx</code>		F	D	D	E	M	W		
inserted nop				E	M	W			
<code>irmovq \$10, %rbx</code>			F	F	D	E	M	W	
...									

insert nop in cycle 3  
during cycle 2:  
bubble\_E = 1

need way to keep pipeline register unchanged to repeat a stage

(and to replace instruction with a nop)

# jump misprediction: bubbles

`addq %r8, %r9`

`jle target (not taken)`

`target: xorq %rax, %rax (mispredicted)`

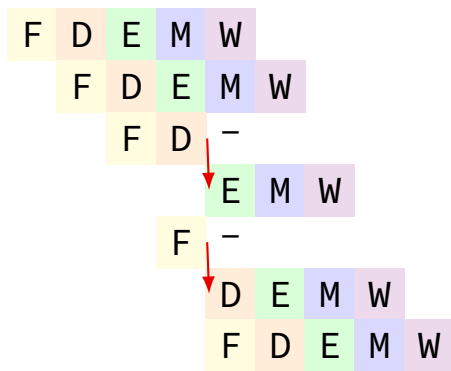
inserted `nop`

`andq %rbx, %rcx (mispredicted)`

inserted `nop`

`subq %r9, %r10 (instr. after jle)`

cycle # 0 1 2 3 4 5 6 7 8

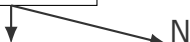


need option: replace instruction with `nop` (“bubble”)

# squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

1	subq				
---	------	--	--	--	--



2	jne	subq			
---	-----	------	--	--	--

3	addq [?]	jne	subq (set ZF)		
---	----------	-----	---------------	--	--

4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
---	------------	----------	--------------	------	--

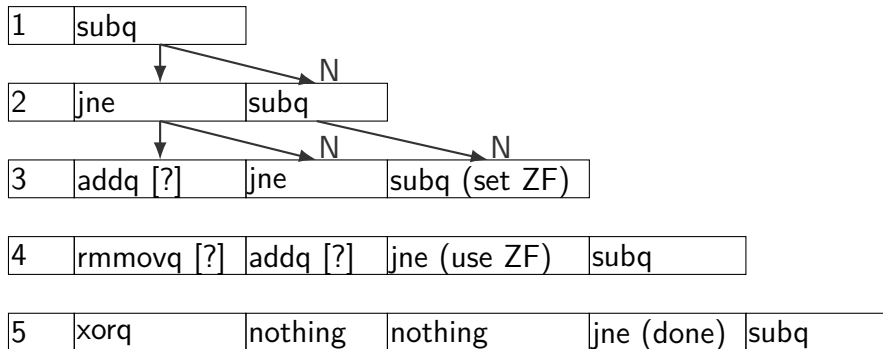
5	xorq	nothing	nothing	jne (done)	subq
---	------	---------	---------	------------	------

stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

# squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

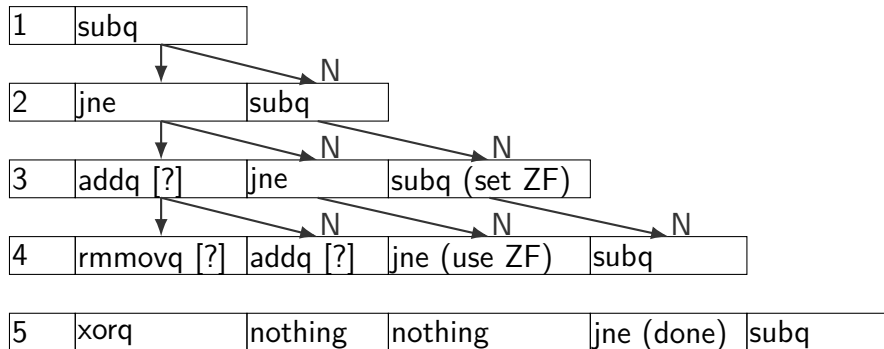


stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

# squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

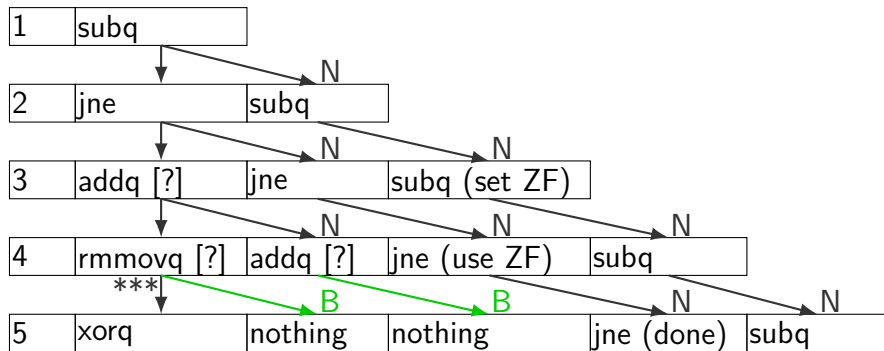


stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

# squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

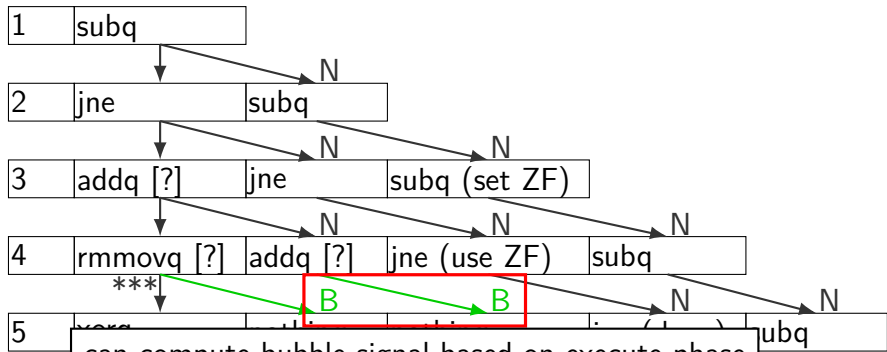
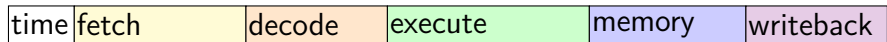


stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);



# squashing with stall/bubble



can compute bubble signal based on execute phase  
stall won't even start CC write for addq  
bubble (B) = use default (no-op);  
new value

## squashing HCLRS

```
just_detected_mispredict =  
    e_icode == JXX && !e_branchTaken;  
bubble_D = just_detected_mispredict || ...;  
bubble_E = just_detected_mispredict || ...;
```

# ret bubbles

addq %r8, %r9

ret

???

inserted nop

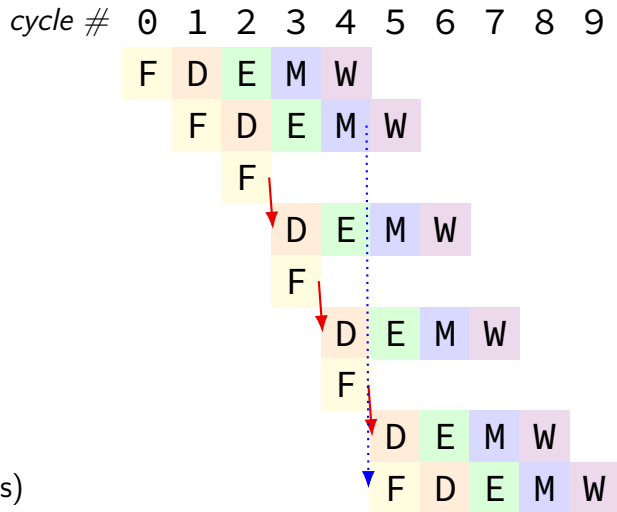
???

inserted nop

???

inserted nop

rrmovq %rax, %r8 (return address)



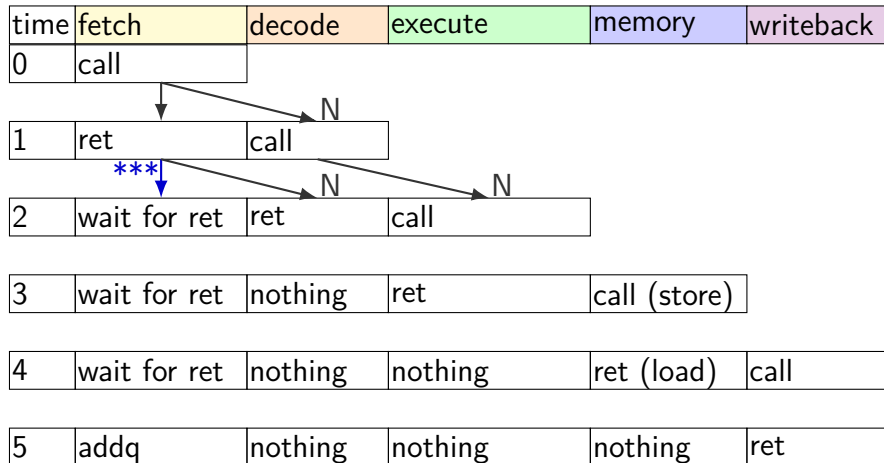
need option: replace instruction with nop (“bubble”)

# ret stall

time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret			
3	wait for ret	nothing	ret		call (store)
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

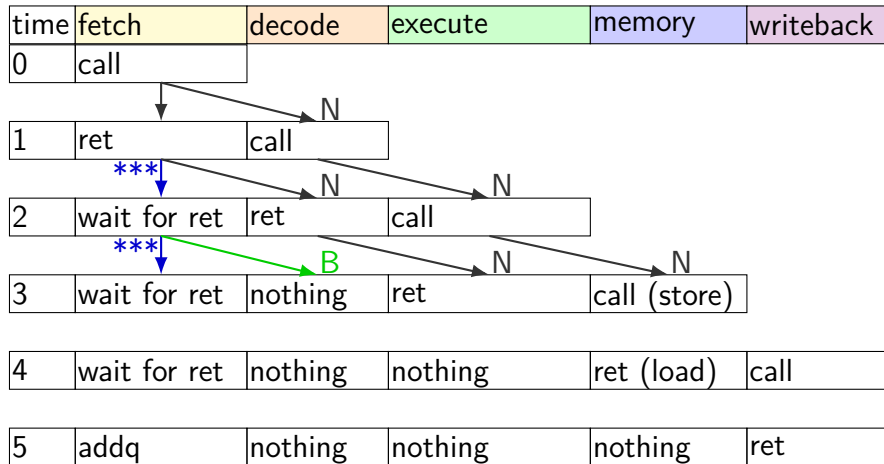
stall (S) = keep old value; normal (N) = use new value  
bubble (B) = use default (no-op);

# ret stall



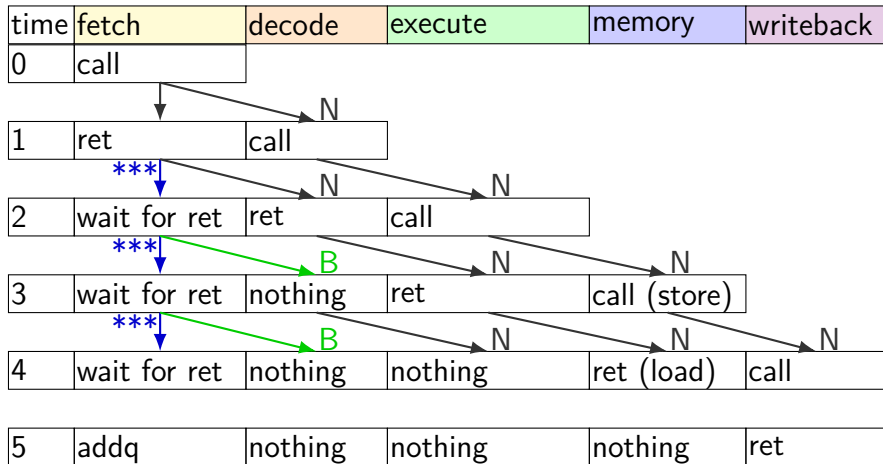
stall (S) = keep old value; normal (N) = use new value  
bubble (B) = use default (no-op);

# ret stall



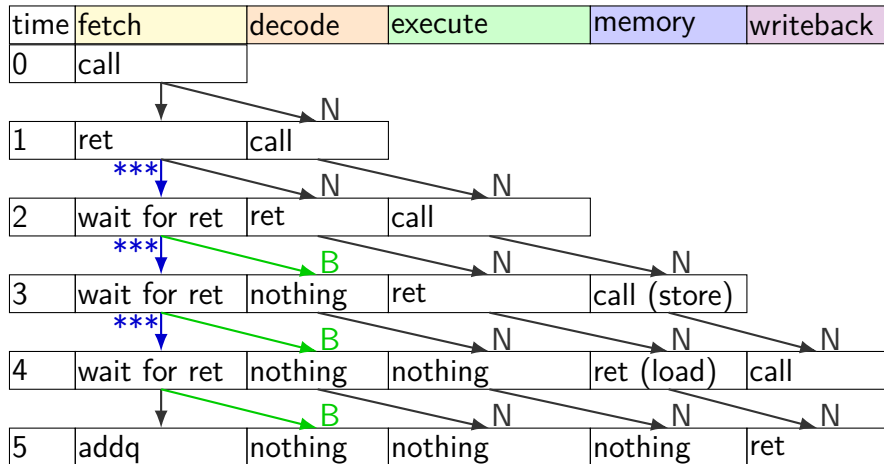
stall (S) = keep old value; normal (N) = use new value  
bubble (B) = use default (no-op);

# ret stall



stall (S) = keep old value; normal (N) = use new value  
 bubble (B) = use default (no-op);

# ret stall



stall (S) = keep old value; normal (N) = use new value  
 bubble (B) = use default (no-op);



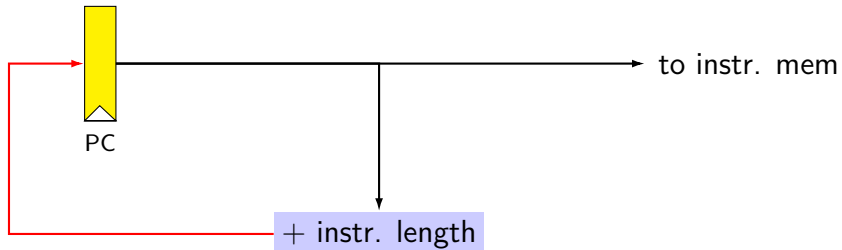
# HCLRS bubble example

```
register fD {
    icode : 4 = NOP;
    rA   : 4 = REG_NONE;
    rB   : 4 = REG_NONE;
    ...
};
wire need_ret_bubble : 1;
need_ret_bubble = ( D_icode == RET ||
                   E_icode == RET ||
                   M_icode == RET );

bubble_D = ( need_ret_bubble ||
             ... /* other cases */ );
```

# building the PC update (one possibility)

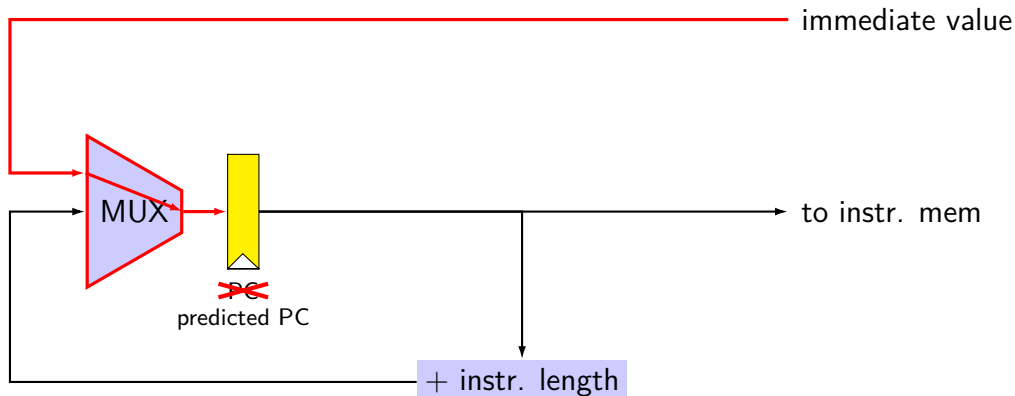
(1) normal case:  $PC \leftarrow PC + \text{instr len}$



# building the PC update (one possibility)

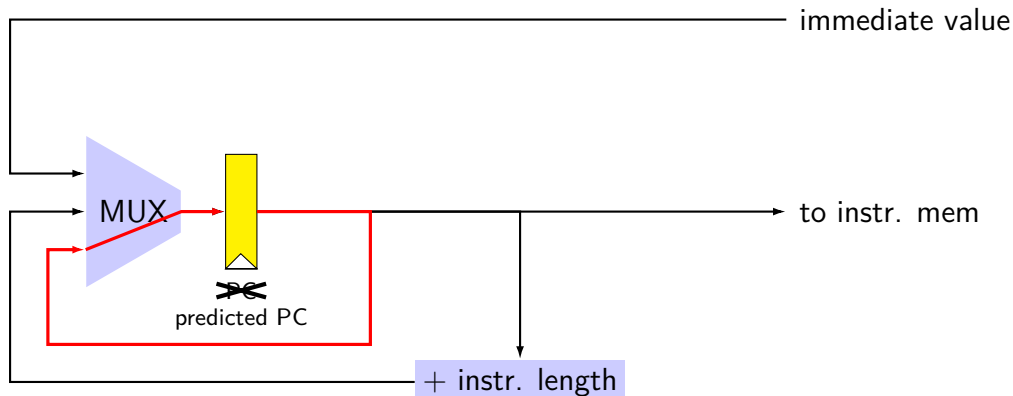
(1) normal case:  $PC \leftarrow PC + \text{instr len}$

(2) immediate: call/jmp, and *prediction* for cond. jumps



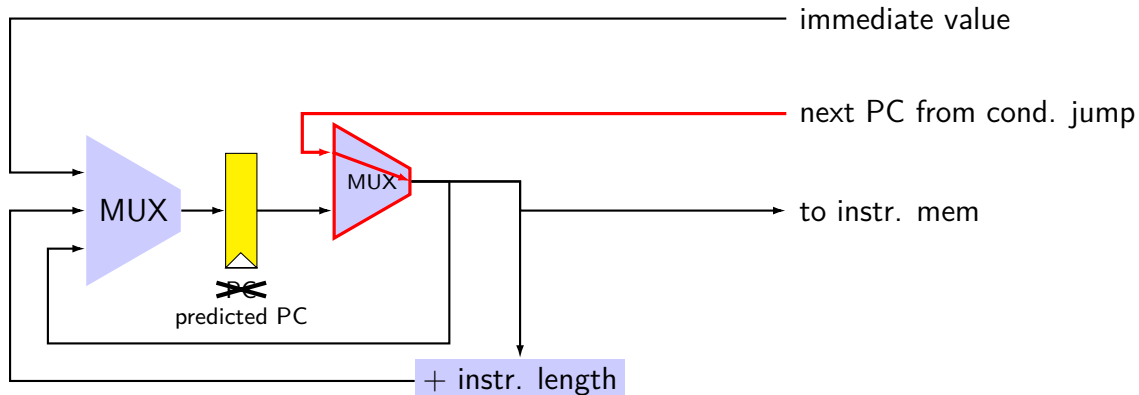
# building the PC update (one possibility)

- (1) normal case:  $PC \leftarrow PC + \text{instr len}$
- (2) immediate: call/jmp, and *prediction* for cond. jumps
- (3) repeat previous PC for stalls (load/use hazard, halt, ret?)



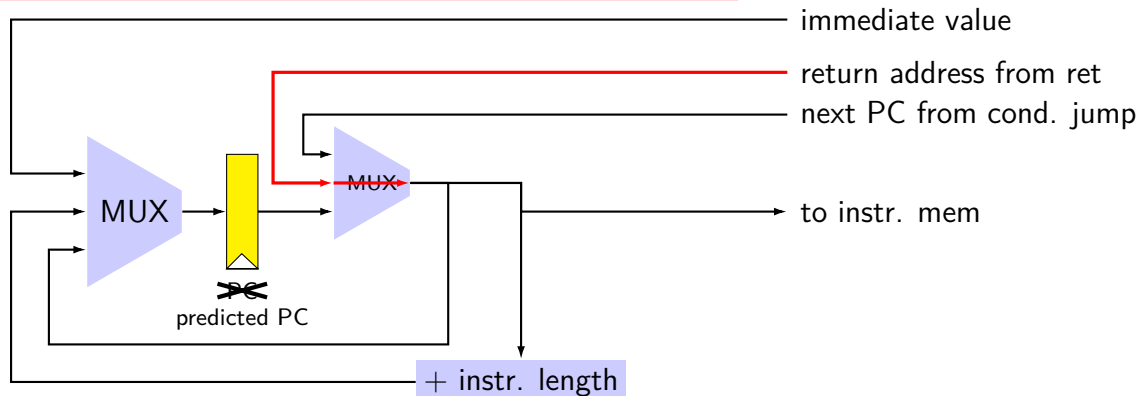
# building the PC update (one possibility)

- (1) normal case:  $PC \leftarrow PC + \text{instr len}$
- (2) immediate: call/jmp, and *prediction* for cond. jumps
- (3) repeat previous PC for stalls (load/use hazard, halt, ret?)
- (4) correct for misprediction of conditional jump



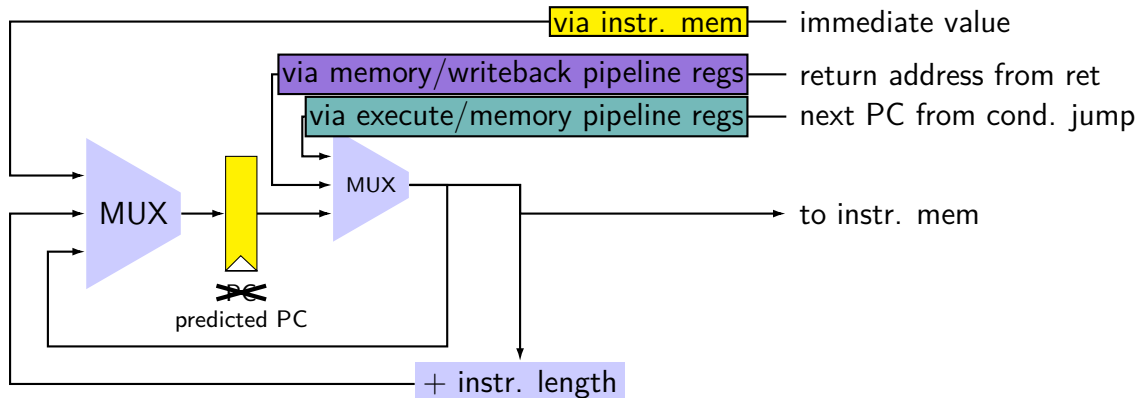
# building the PC update (one possibility)

- (1) normal case:  $PC \leftarrow PC + \text{instr len}$
- (2) immediate: call/jmp, and *prediction* for cond. jumps
- (3) repeat previous PC for stalls (load/use hazard, halt, ret?)
- (4) correct for misprediction of conditional jump
- (5) correct for missing return address for ret



# building the PC update (one possibility)

- (1) normal case:  $PC \leftarrow PC + \text{instr len}$
- (2) immediate: call/jmp, and *prediction* for cond. jumps
- (3) repeat previous PC for stalls (load/use hazard, halt, ret?)
- (4) correct for misprediction of conditional jump
- (5) correct for missing return address for ret



# PC update overview

predict based on instruction length + immediate

override prediction with stalling sometimes

correct when prediction is wrong just before fetching

retrieve corrections from pipeline register outputs for jCC/ret instruction

above is what textbook does

alternative: could instead correct prediction just before setting PC register

retrieve corrections into PC cycle before corrections used

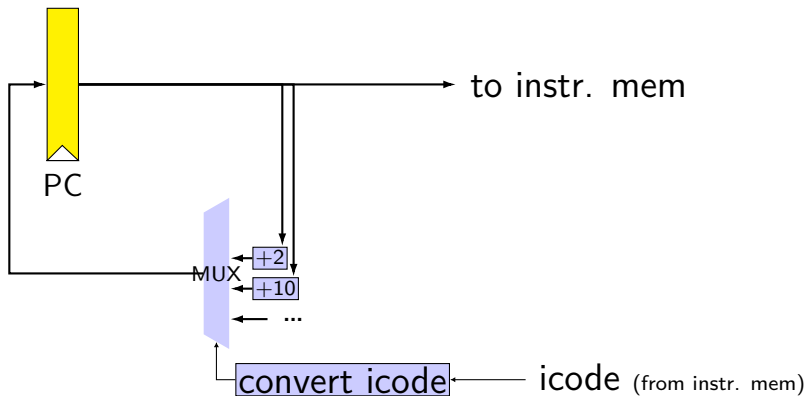
moves logic from beginning-of-fetch to end-of-previous-fetch

I think this is more intuitive, but consistency with textbook is less confusing...

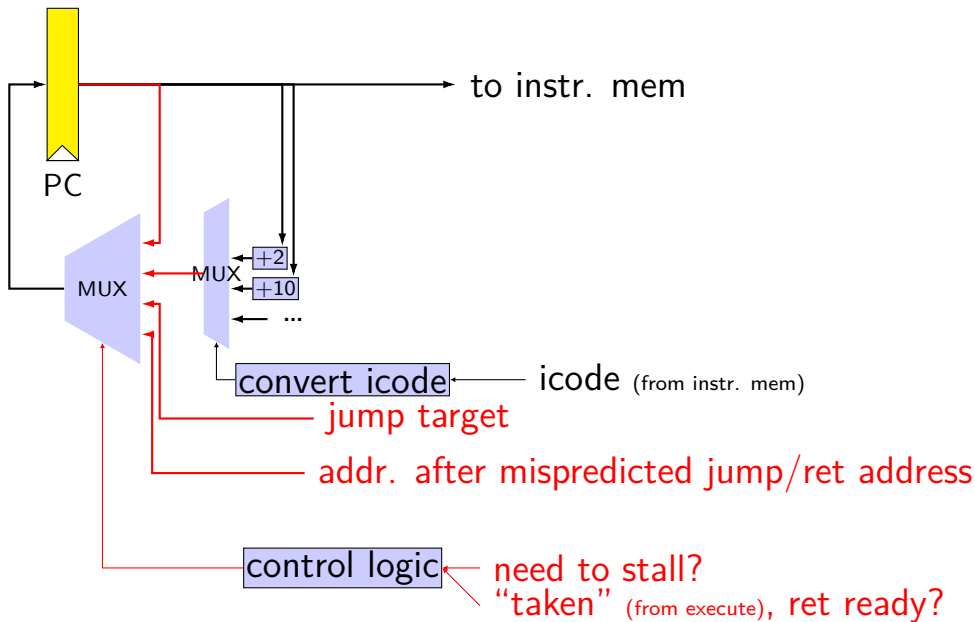


# backup slides

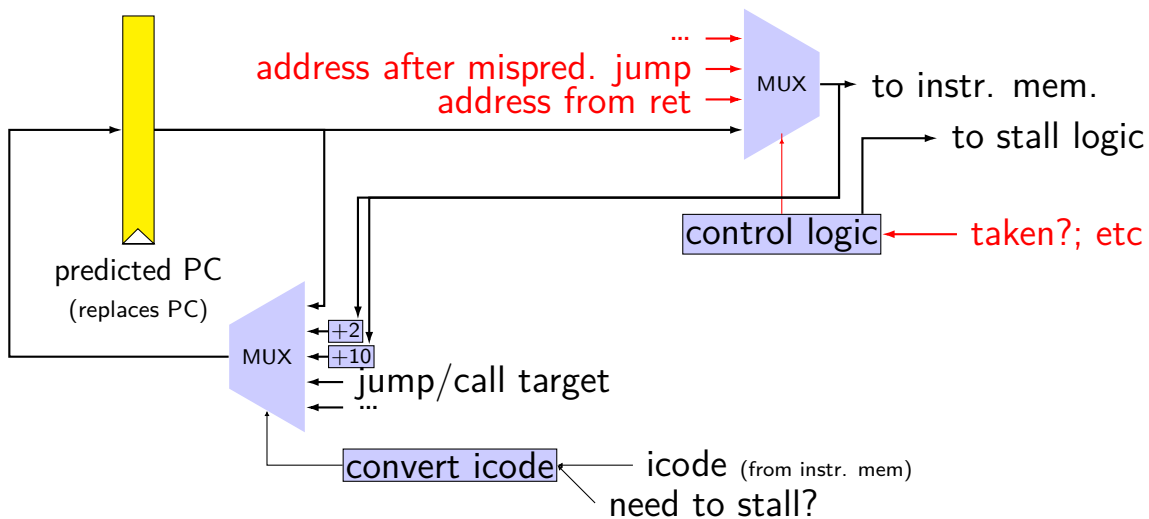
# PC update (adding prediction, stall)



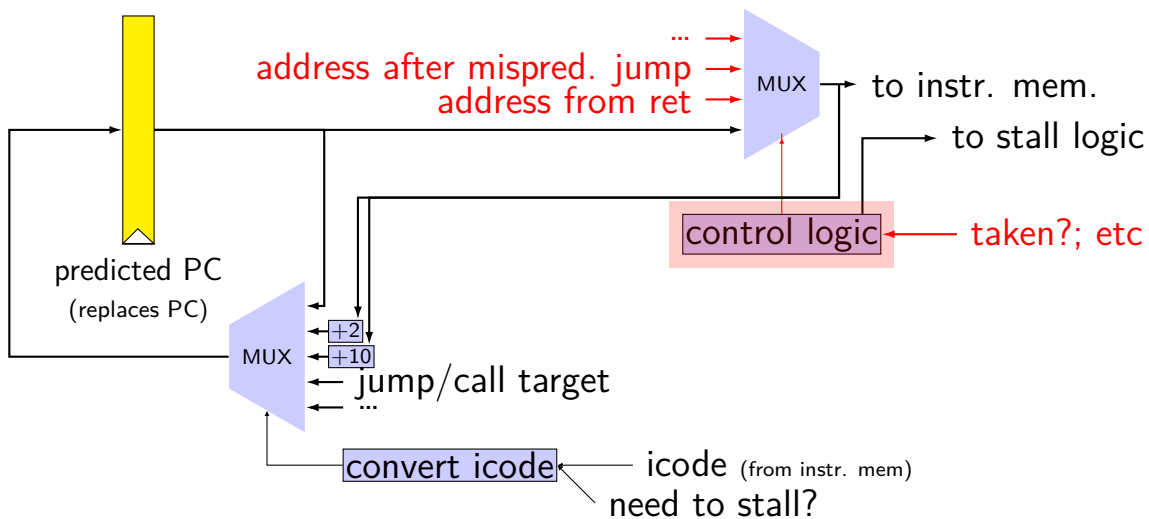
# PC update (adding prediction, stall)



# PC update (rearranged)



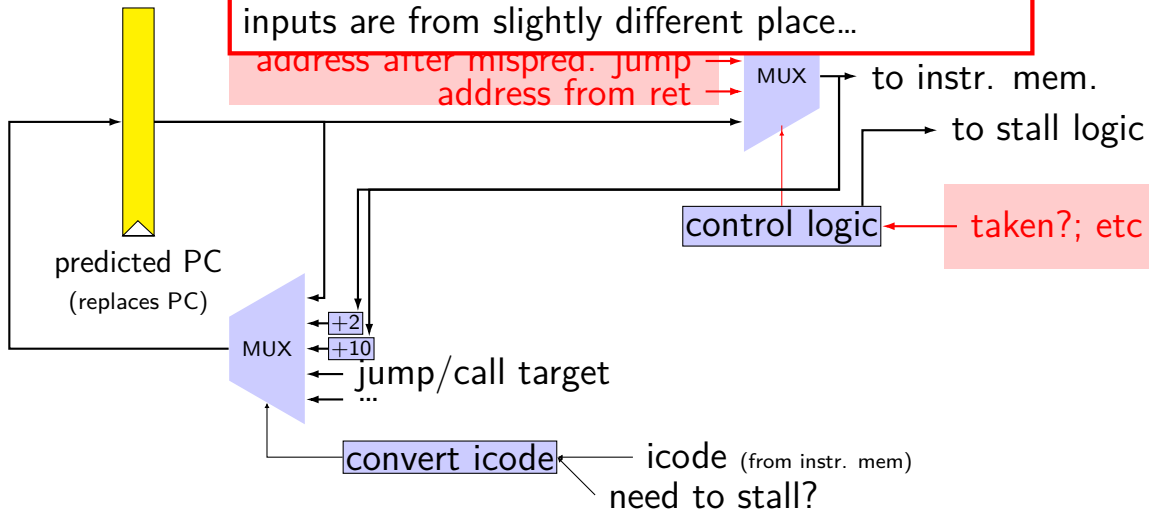
# PC update (rearranged)



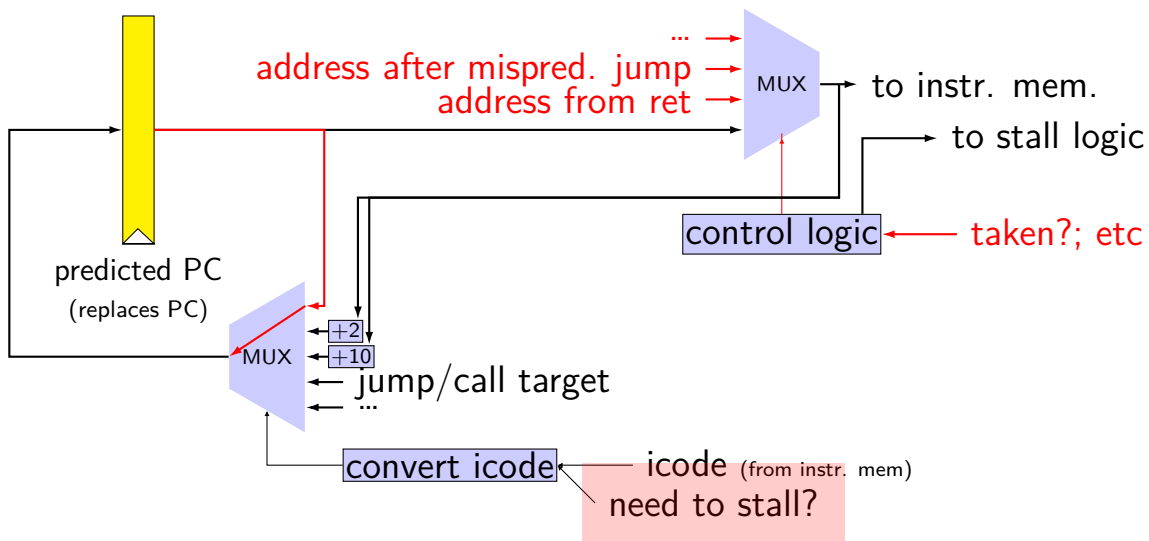
# PC update (rearranged)

same logic as before — but happens in next cycle  
inputs are from slightly different place...

address after mispred. jump  
address from ret



# PC update (rearranged)



## rearranged PC update in HCL

```
/* replacing the PC register: */
register fF {
    predictedPC: 64 = 0;
}

/* actual input to instruction memory */
pc = [
    conditionCodesSaidNotTaken : jumpValP;
    /* from later in pipeline */
    ...
    1: F_predictedPC;
];
```



# why rearrange PC update?

either works

correct PC at beginning or end of cycle?

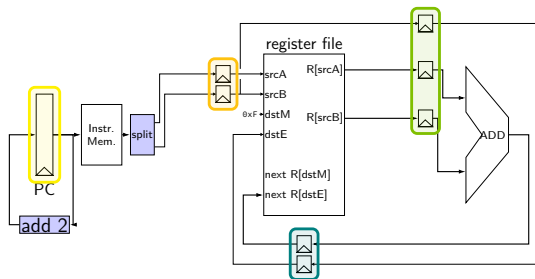
still some time in cycle to do so...

maybe easier to think about branch prediction this way?

# addq processor performance

example delays:

path	time
add 2	80 ps
instruction memory	200 ps
register file read	125 ps
add	100 ps
register file write	125 ps



no pipelining: 1 instruction per 550 ps

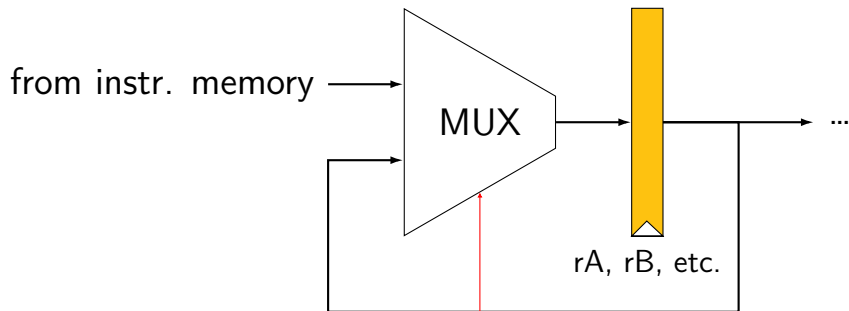
add up everything but add 2 (**critical (slowest) path**)

pipelining: 1 instruction per 200 ps + pipeline register delays

**slowest path through stage** + pipeline register delays

latency: 800 ps + pipeline register delays (4 cycles)

# fetch/decode logic — advance or not



should we stall?

## ex.: dependencies and hazards (1)

`addq %rax, %rbx`

`subq %rax, %rcx`

`irmovq $100, %rcx`

`addq %rcx, %r10`

`addq %rbx, %r10`

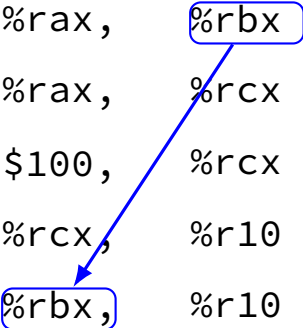
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

## ex.: dependencies and hazards (1)

```
addq    %rax, %rbx
subq    %rax, %rcx
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```



where are dependencies?  
which are hazards in our pipeline?  
which are resolved with forwarding?

## ex.: dependencies and hazards (1)

```
addq    %rax, %rbx
subq    %rax, %rcx
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```

The diagram illustrates data dependencies between instructions. A blue arrow points from the `%rbx` operand of the first instruction (`addq %rax, %rbx`) to the `%rbx` operand of the fifth instruction (`addq %rbx, %r10`). A red arrow points from the `%rcx` operand of the third instruction (`irmovq $100, %rcx`) to the `%rcx` operand of the fourth instruction (`addq %rcx, %r10`).

where are dependencies?  
which are hazards in our pipeline?  
which are resolved with forwarding?

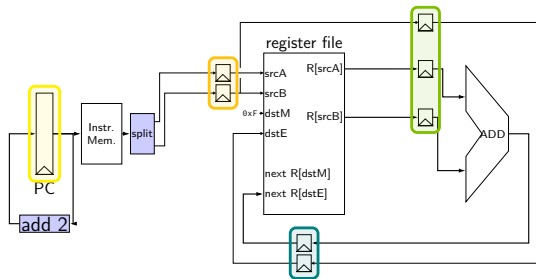
## ex.: dependencies and hazards (1)

addq	%rax,	<span style="border: 1px solid blue; border-radius: 10px; padding: 2px;">%rbx</span>
subq	%rax,	%rcx
irmovq	\$100,	<span style="border: 1px solid red; border-radius: 10px; padding: 2px;">%rcx</span>
addq	<span style="border: 1px solid red; border-radius: 10px; padding: 2px;">%rcx,</span>	<span style="border: 1px solid red; border-radius: 10px; padding: 2px;">%r10</span>
addq	<span style="border: 1px solid blue; border-radius: 10px; padding: 2px;">%rbx,</span>	<span style="border: 1px solid red; border-radius: 10px; padding: 2px;">%r10</span>

where are dependencies?  
which are hazards in our pipeline?  
which are resolved with forwarding?

# exercise

path	time
add 2	50 ps
instruction memory	200 ps
register file read	125 ps
add	100 ps
register file write	125 ps



pipeline register delay: 10ps

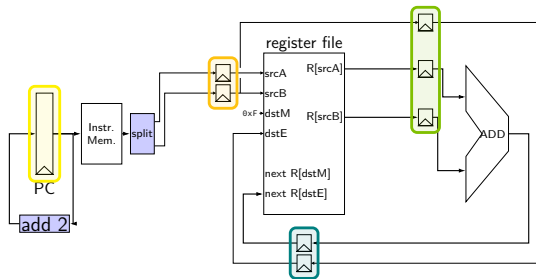
how will throughput improve if we **double the speed of the instruction memory**?

- A.** 2.00x
- B.** 1.70x to 1.99x
- C.** 1.60x to 1.69x
- D.** 1.50x to 1.59x
- E.** less than 1.50x



# exercise

path	time
add 2	50 ps
instruction memory	200 ps
register file read	125 ps
add	100 ps
register file write	125 ps



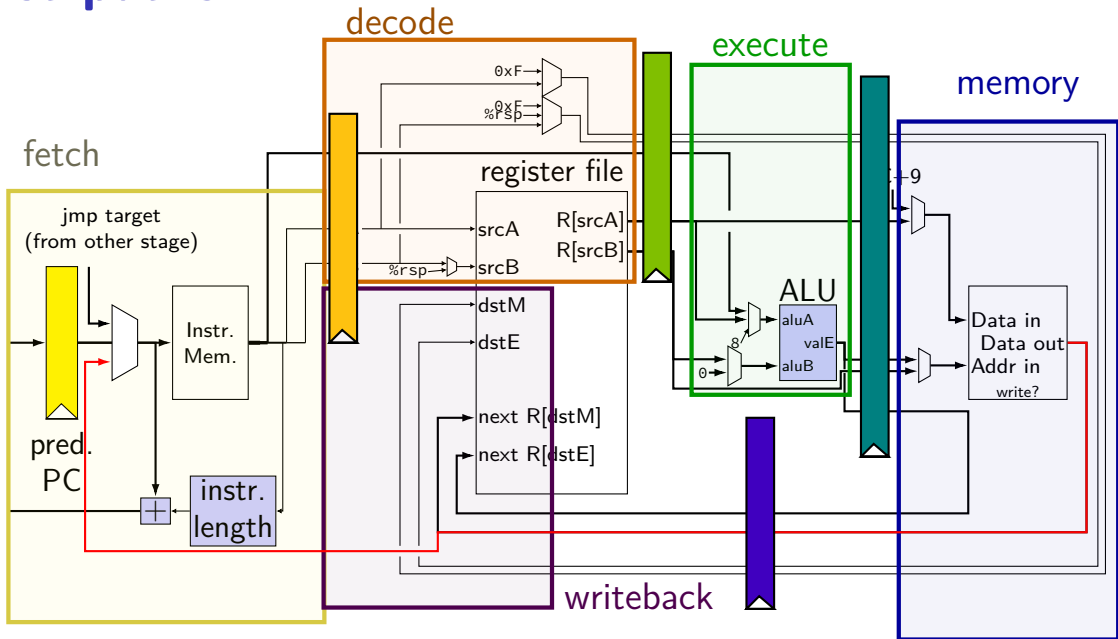
pipeline register delay: 10ps

how will throughput improve if we **double the speed of the instruction memory?**

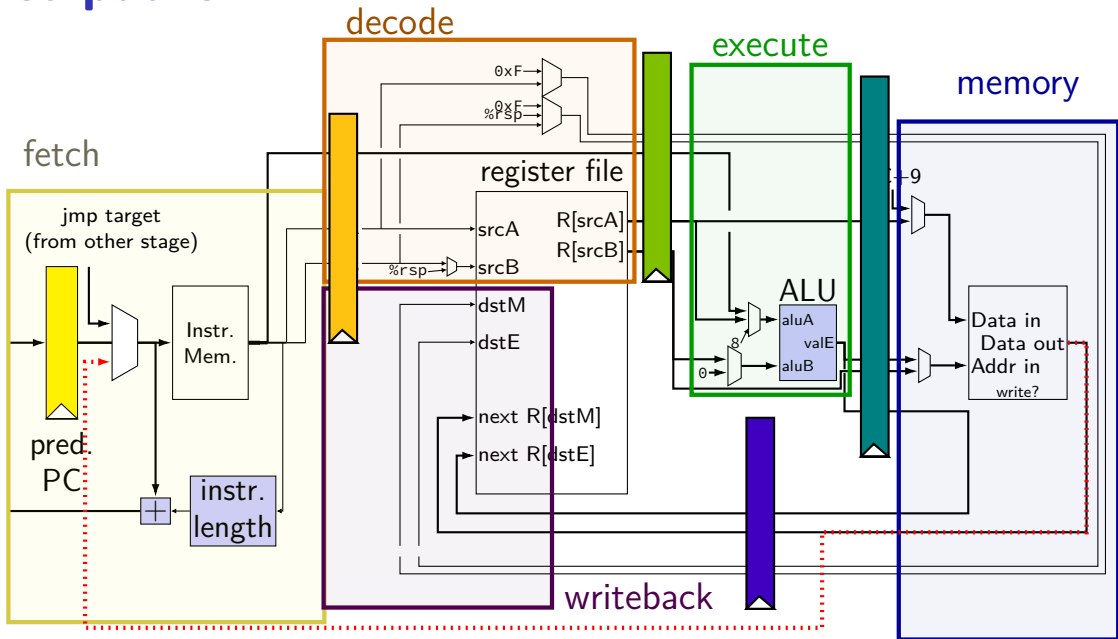
- A.** 2.00x
- B.** 1.70x to 1.99x
- C.** 1.60x to 1.69x
- D.** 1.50x to 1.59x
- E.** less than 1.50x

$$\frac{1}{135} \div \frac{1}{210} = 1.56x \text{ — D}$$

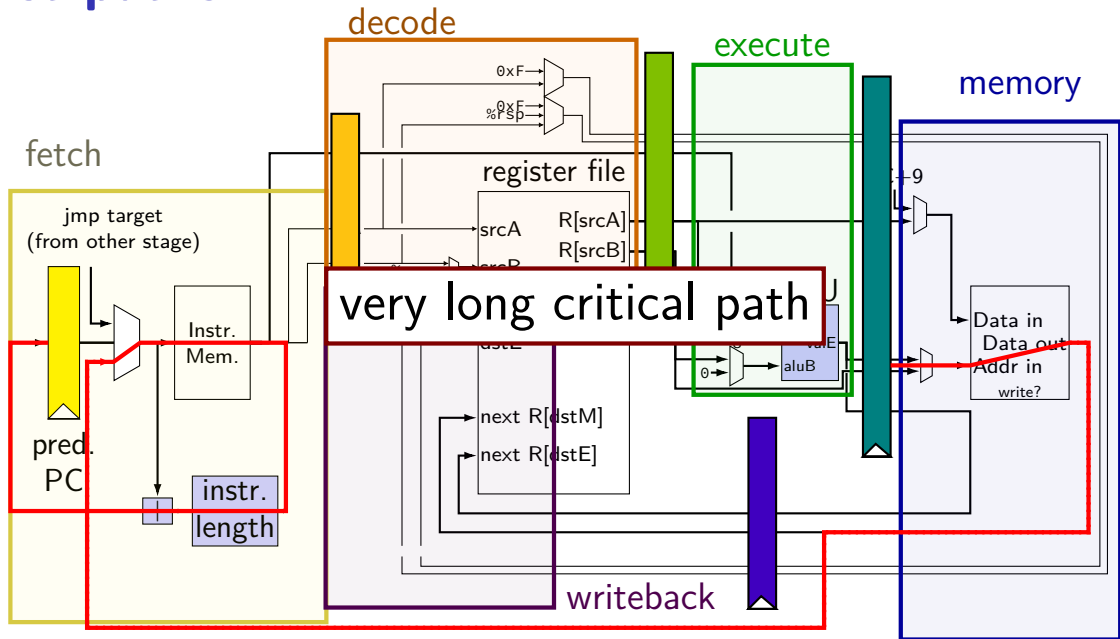
# ret paths



# ret paths



# ret paths



# revisiting data hazards

stalling worked

but very unsatisfying — wait 2 extra cycles to use anything?!  
...or more with 5-stage pipeline

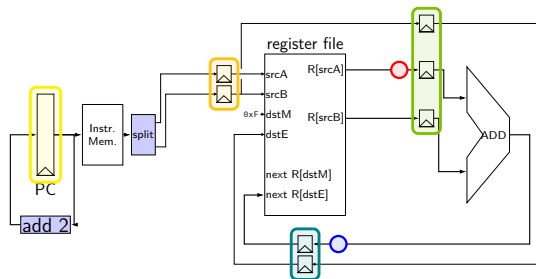
observation: **value** ready before it would be needed  
(just not stored in a way that let's us get it)

# motivation

*// initially %r8 = 800,  
// %r9 = 900, etc.*

```
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

location of values during cycle 2:



	fetch	fetch/decode	decode/execute			execute/writeback		
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

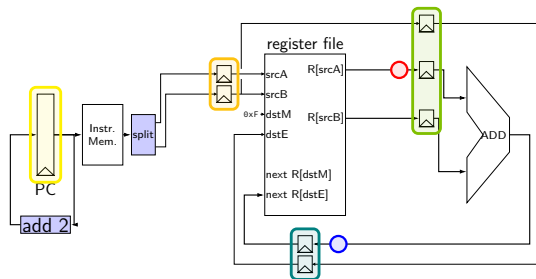
should be 1700

# motivation

*// initially %r8 = 800,  
// %r9 = 900, etc.*

```
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

location of values during cycle 2:



	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700