last time

forwarding and forwarding implementation

dependency / hazards

control hazards

quiz Q1/2

```
                    0 1 2 3 4 5 6 7 8 9 A B C D
mrmovq 8(%r8), %r9  F D E W
irmovq $1000, %r8     F D E W
addq %r9, %r8           F x x D E W
pushq %r8                     F x x D E W
subq %rsp, %r10                     F x x D E W
```

quiz Q3

```
                          0  1  2  3  4  5  6  7  8  9 10 11
addq %r8, %r9             F  D1 D2 E1 E2 M1 M2 M3 W
subq %r9, %r10               F  x  D1 D2 E1 E2 M1 M2 M3 W
mrmovq 8(%r9), %r11             F  x  D1 D2 E1 E2 M1 M2 M3 W
```

quiz Q4

"The true result of the branch is determined near the end of the "execute 2" stage, so the correct instruction can be fetched a cycle later."

```
     addq %r8, %r8              F  D1 D2 E1 E2 M1 M2 M3 W
     jle foo                       F  D1 D2 E1 E2 M1 M2 M3 W
     mrmovq 0x1234(%r8), %r12                     F  D1 D2 E1 E2 M1 M2 M3
     halt

foo: subq %r8, %r9                     F  D1 D2 E1 [then removed -- bad gu
     subq %r8, %r9                        F  D1 D2 [then removed -- bad gu
     subq %r8, %r9                           F  D1 [then removed -- bad gu
     subq %r8, %r9
     subq %r8, %r9
```

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				
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

"squash" wrong guesses

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 [?]	j			
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*

* — ignoring data hazards

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*

predict: $3 \times .03 + 1 \times .05 + 4 \times .01 + 1 \times .91 =$ 1.09 cycles/instr.

stall: $3 \times .03 + 3 \times .05 + 4 \times .01 + 1 \times .91 =$ 1.19 cylces/instr. ($1.19 \div 1.09 \approx 1.09$x faster)

* — ignoring data hazards

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`

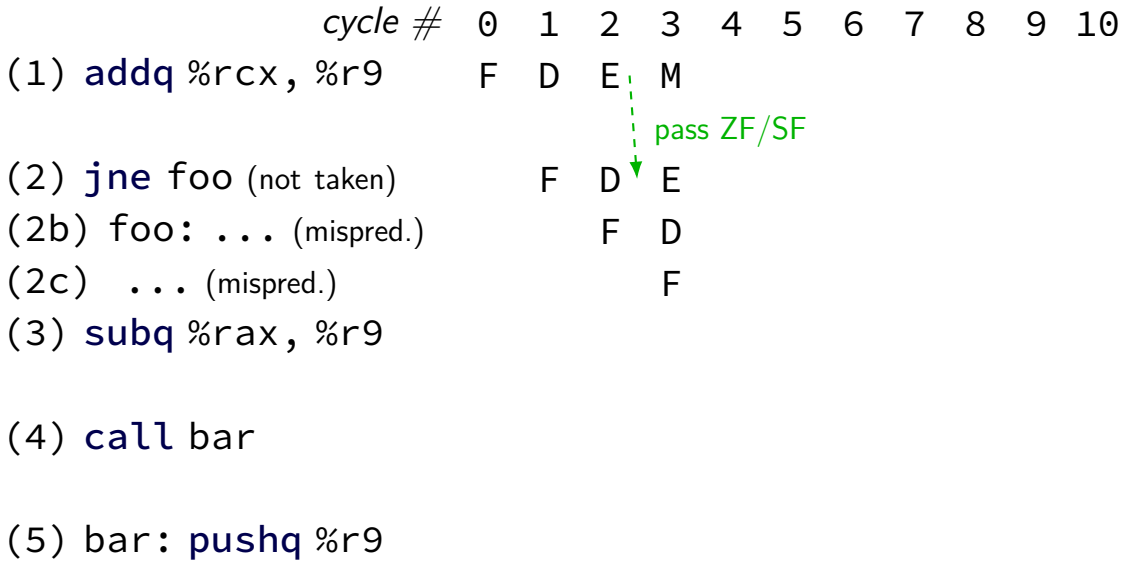
[solution]: control hazard timing+forwarding?

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

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

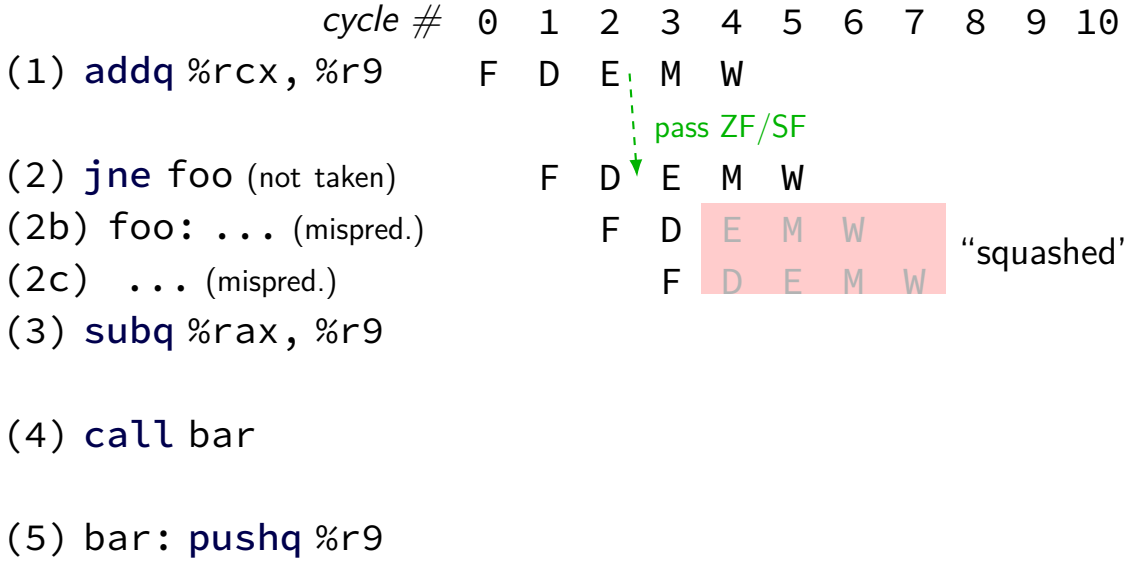
[solution]: control hazard timing+forwarding?

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



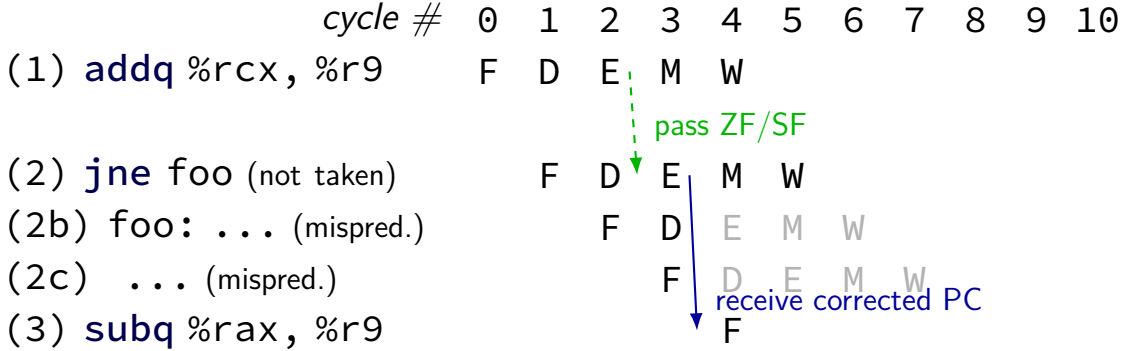
[solution]: control hazard timing+forwarding?

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



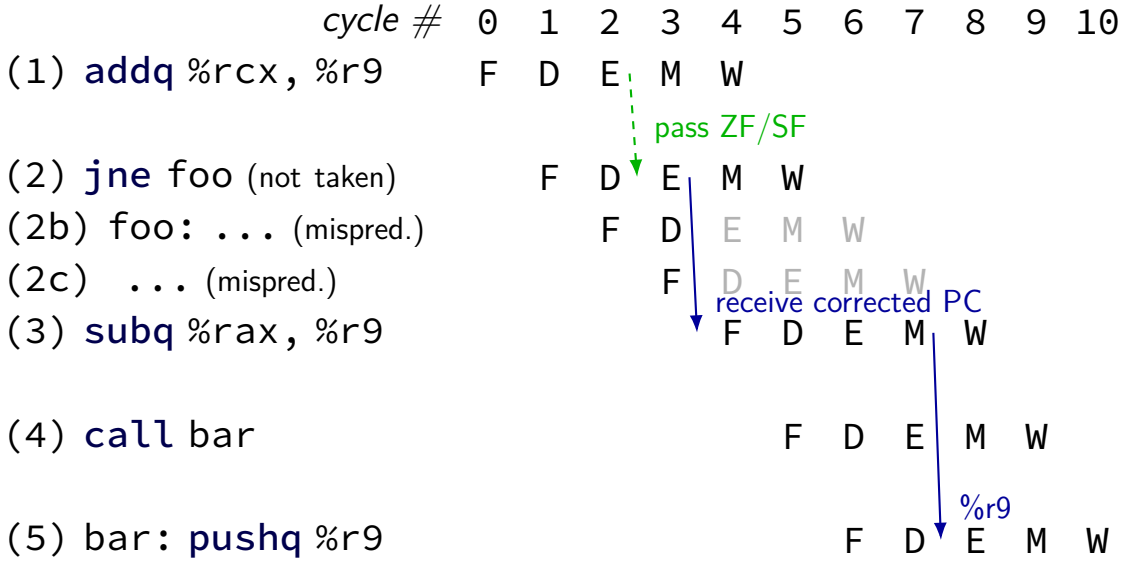
[solution]: control hazard timing+forwarding?

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



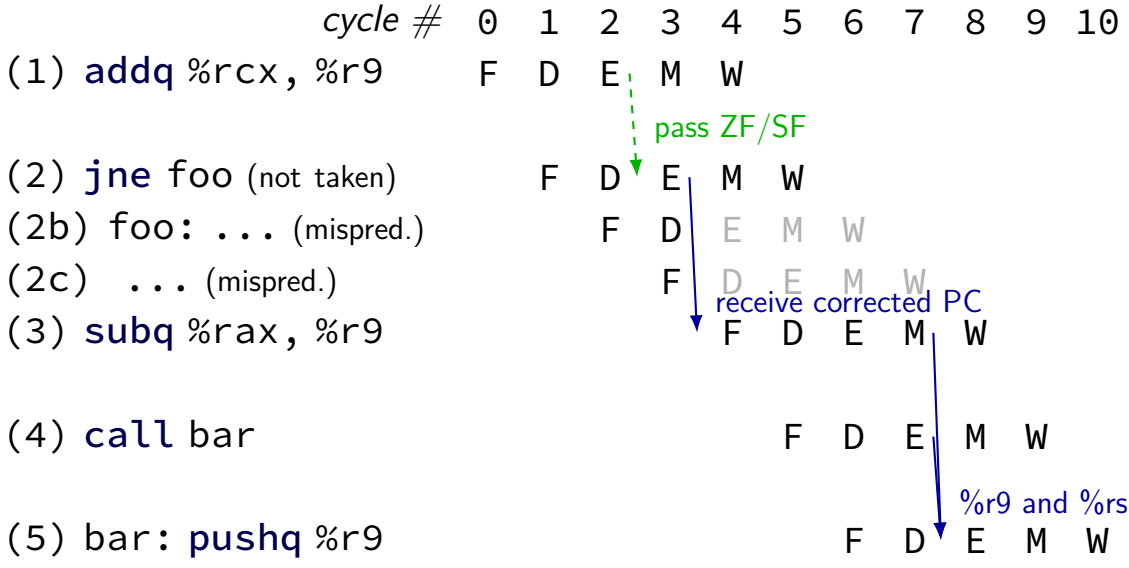
[solution]: control hazard timing+forwarding?

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

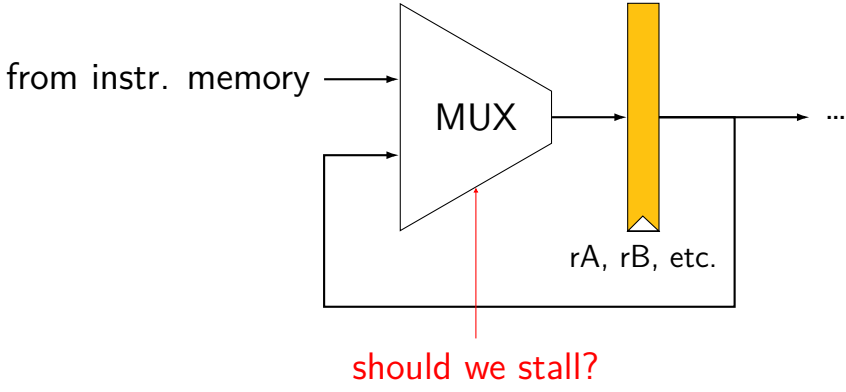
with F/D/E1/E2/M/W

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`											

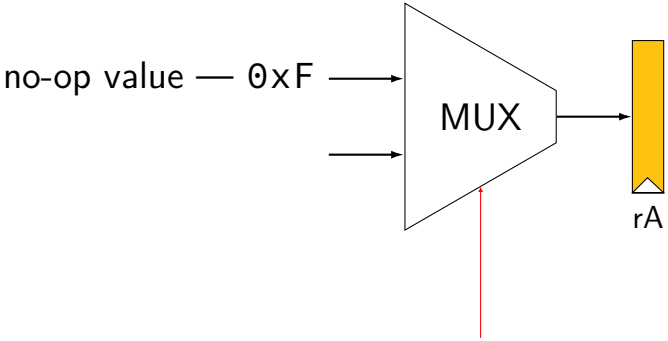
[solution]: with different pipeline

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

fetch/decode logic — advance or not



fetch/decode logic — bubble or not



HCLRS signals

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

HCLRS: every register bank has these MUXes built-in

`stall_B`: keep old value for all registers

- register input ← register output
- pipeline: keep same instruction in this stage next cycle

`bubble_B`: use default value for all registers

- register input ← 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

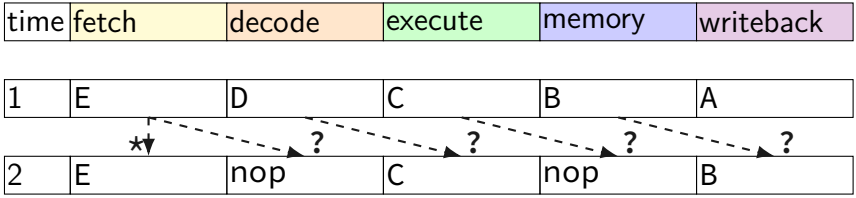
time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	???	1	0
2	0x03	???	0	0
3	0x04	???	0	1
4	0x05	???	0	0
5	0x06	???	0	0
6	0x07	???	1	0
7	0x08	???	1	0
8		???		

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		

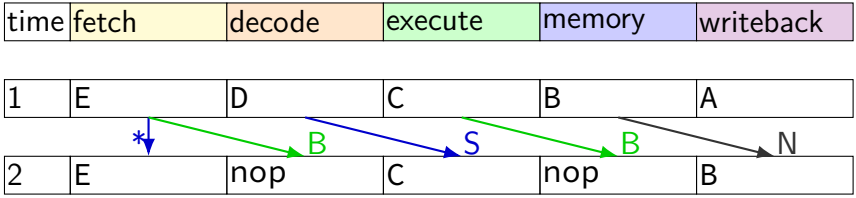
exercise: squash + stall (1)



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

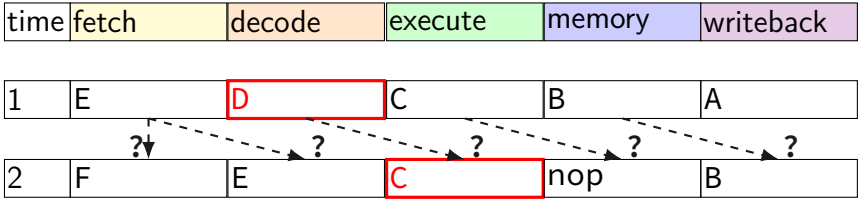
exercise: squash + stall (1)



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

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

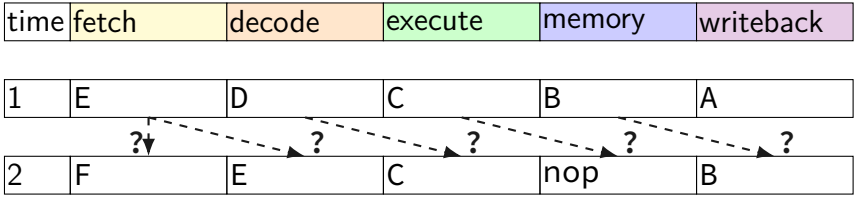
exercise: squash + stall (2)



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

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

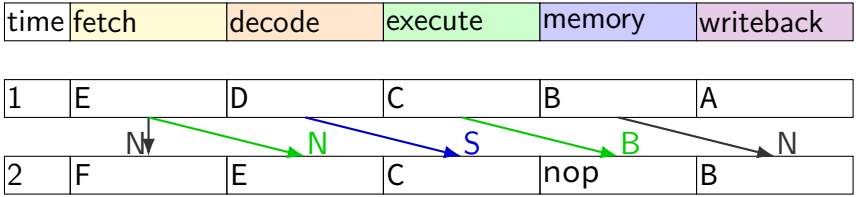
exercise: squash + stall (2)



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)



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

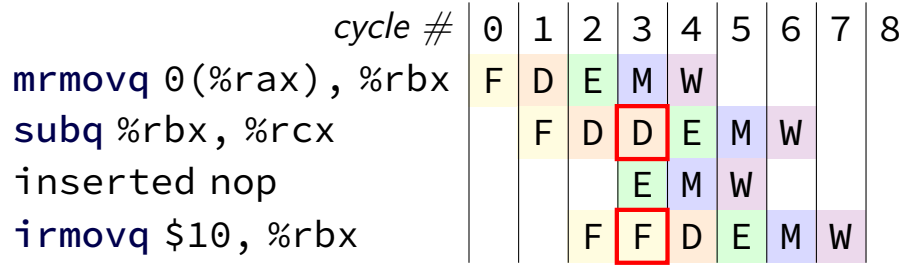
stalling: bubbles + stall

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

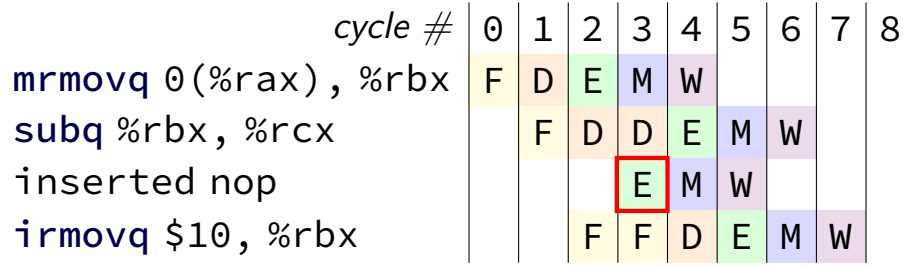


keep same instruction in cycle 3
during cycle 2:
stall_D = 1
stall_F = 1 or extra f_pc MUX

need way to keep pipeline register unchanged to repeat a stage

(*and* to replace instruction with a nop)

stalling: bubbles + stall

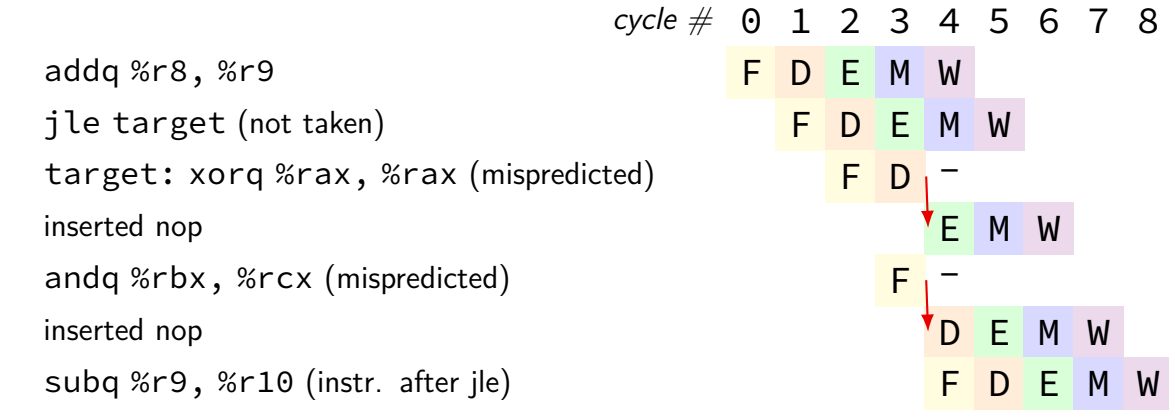


insert nop in cycle 3
during cycle 2:
bubble_E = 1

need way to keep pipeline register unchanged to repeat a stage

(*and* to replace instruction with a nop)

jump misprediction: bubbles



need option: replace instruction with nop ("bubble")

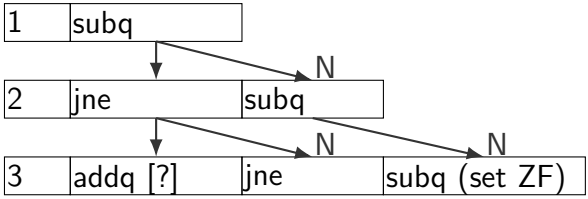
squashing with stall/bubble

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);

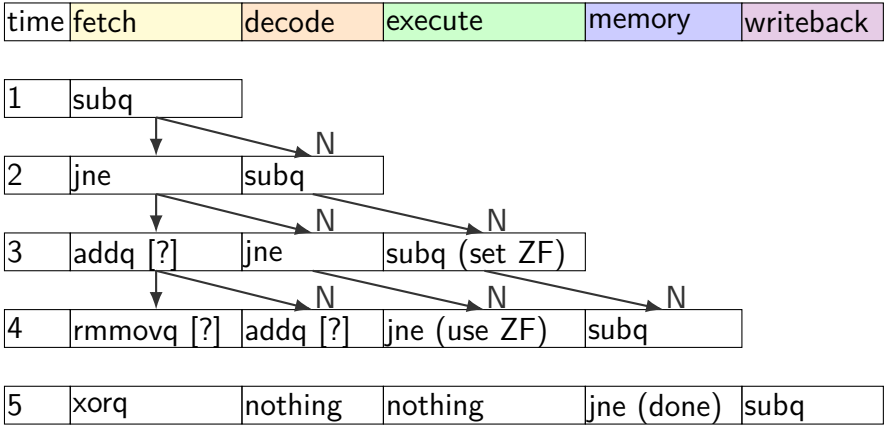
squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
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



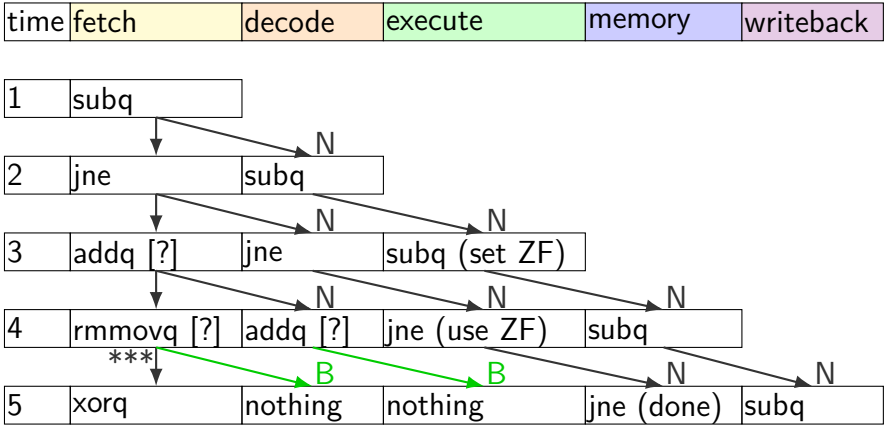
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



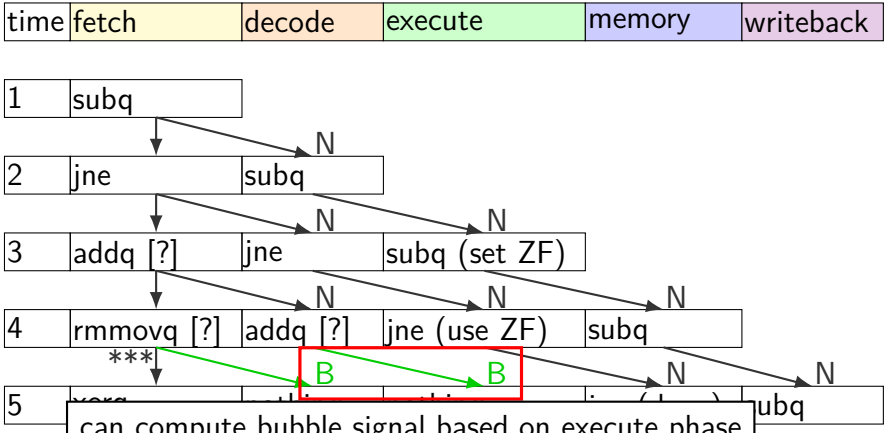
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



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

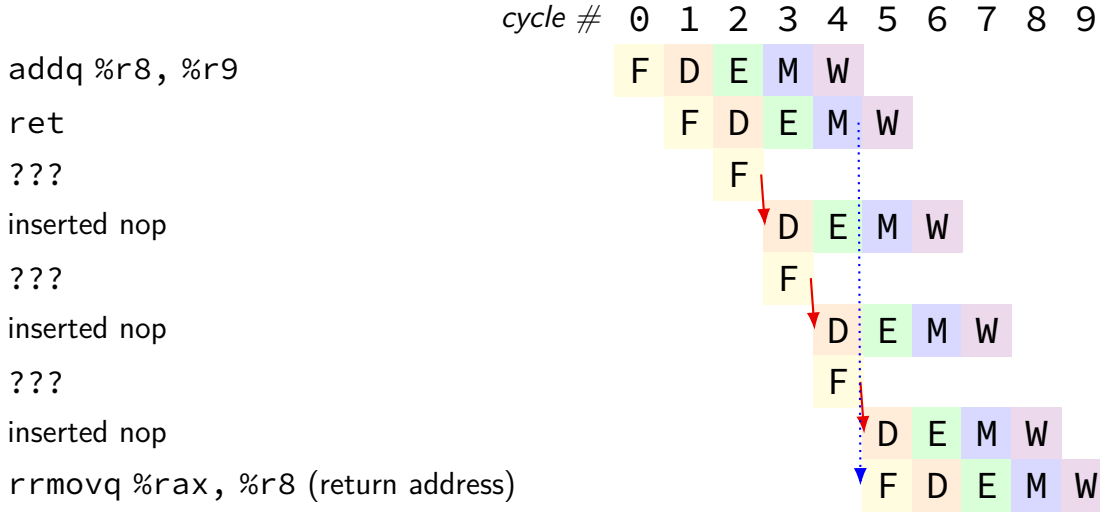
squashing with stall/bubble



squashing HCLRS

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

ret bubbles



need option: replace instruction with nop ("bubble")

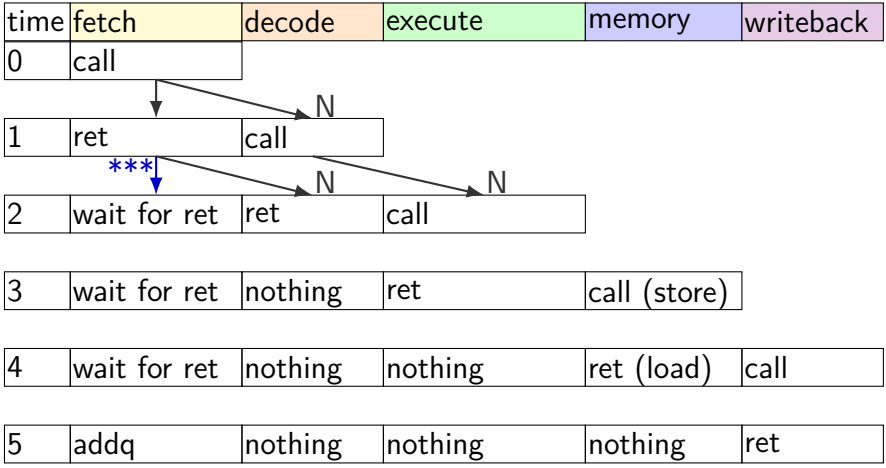
ret stall

time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
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);

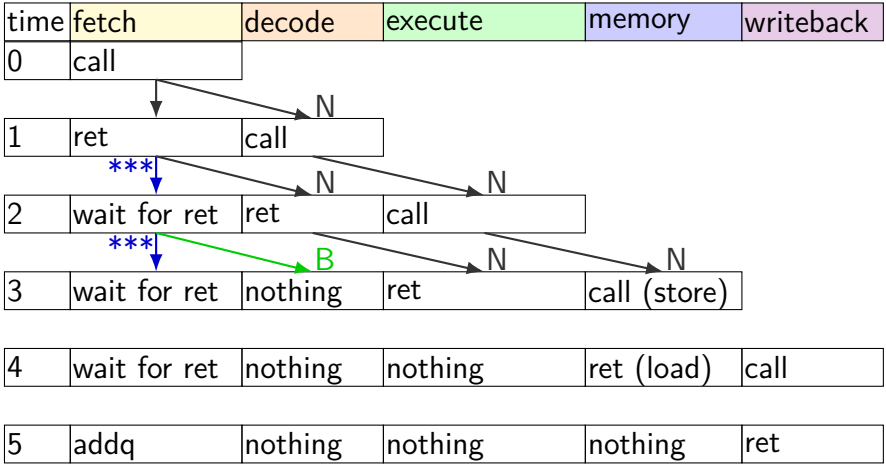
ret stall



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

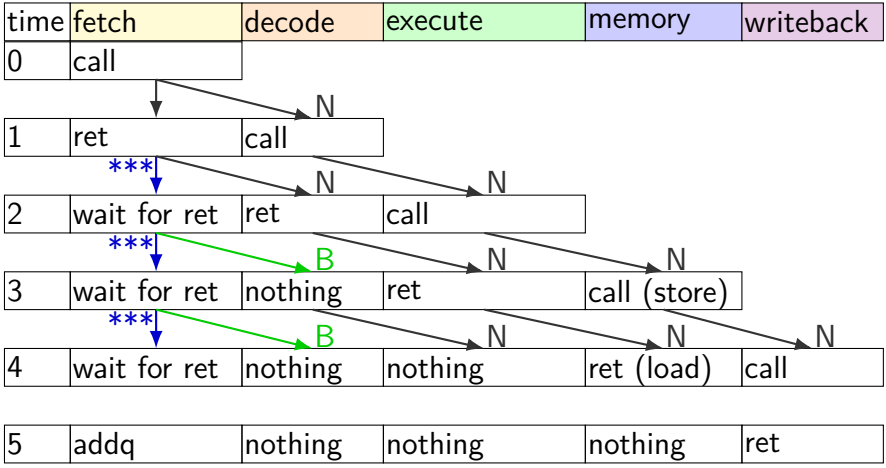
bubble (B) = use default (no-op);

ret stall



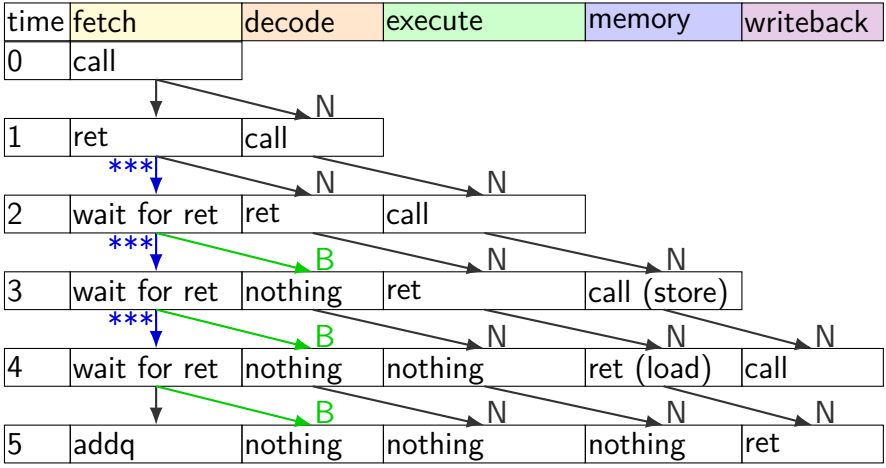
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

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);

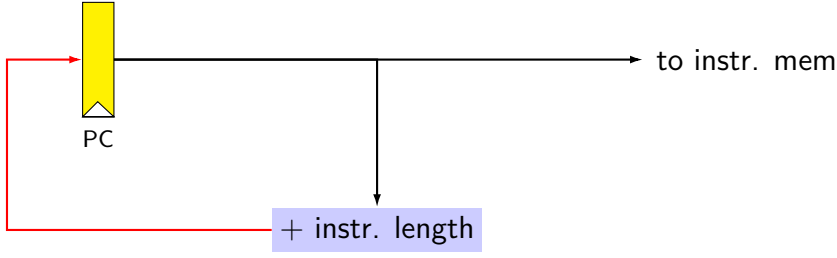
HCLRS bubble example

```
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)

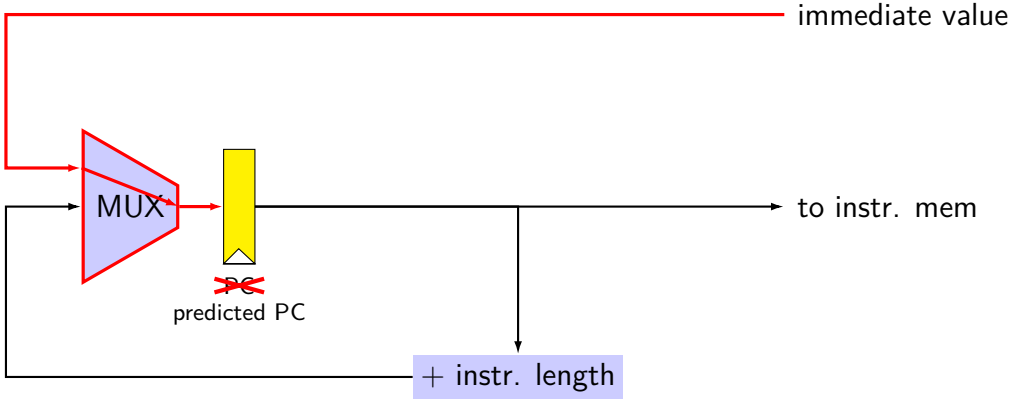
(1) normal case: PC $\leftarrow$ PC + instr len



building the PC update (one possibility)

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

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

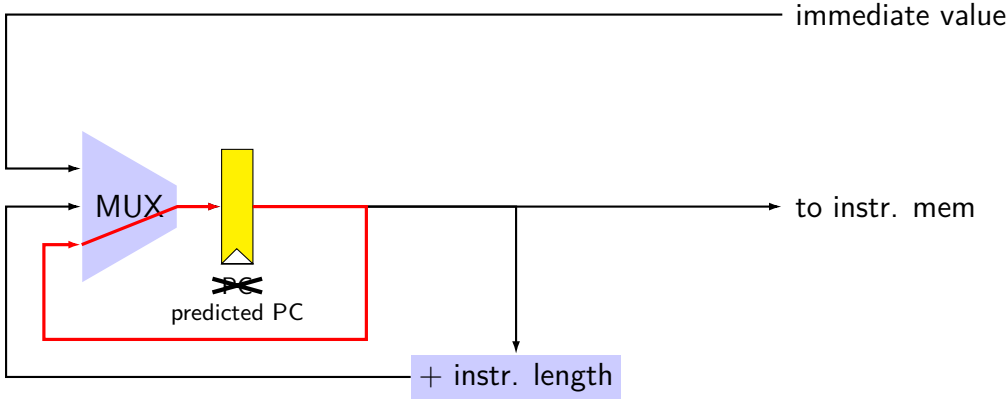


building the PC update (one possibility)

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

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

(3) repeat previous PC for stalls (load/use hazard, halt, ret?)



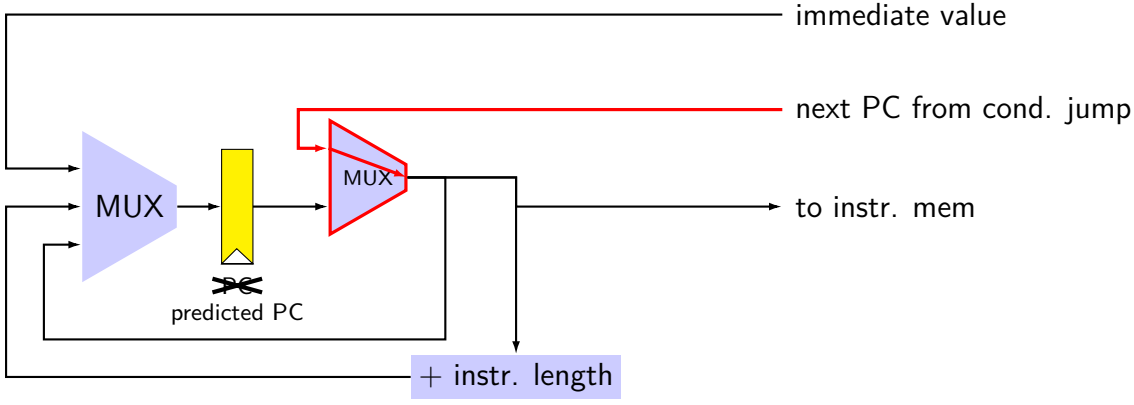
building the PC update (one possibility)

(1) normal case: PC $\leftarrow$ PC + 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)

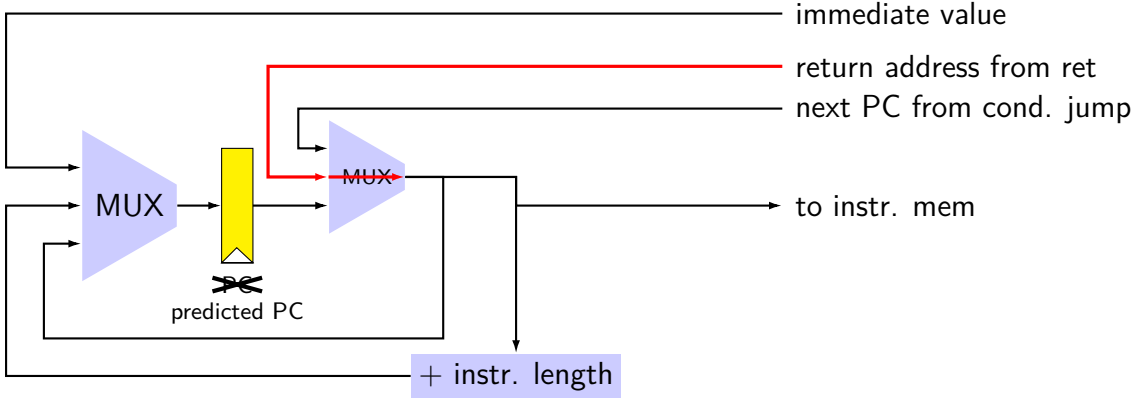
(1) normal case: PC $\leftarrow$ PC + instr len

(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)

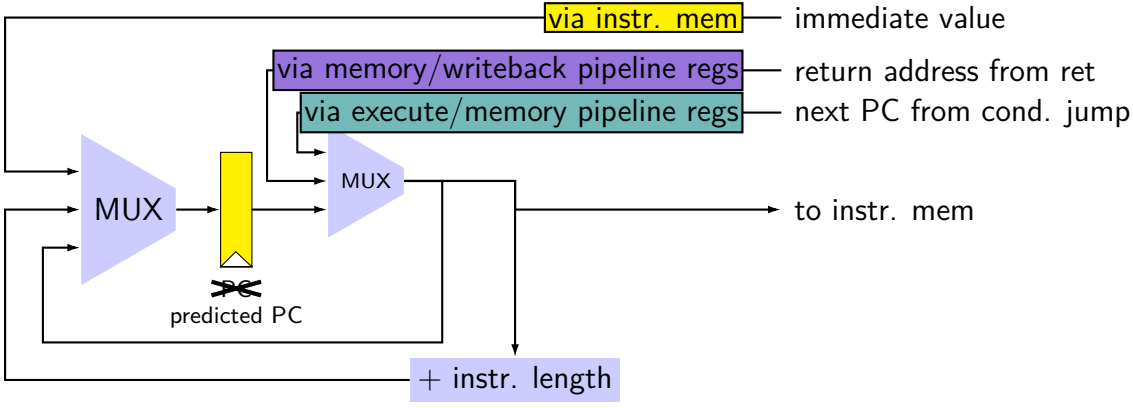
(1) normal case: PC $\leftarrow$ PC + instr len

(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...

after forwarding/prediction

where do we still need to stall?

memory output needed in fetch

- `ret` followed by anything

memory output needed in exceute

- `mrmovq` or `popq` + use
  (in immediatelly following instruction)

overall CPU

5 stage pipeline

1 instruction completes every cycle — except hazards

most data hazards: solved by forwarding

load/use hazard: 1 cycle of stalling

jXX control hazard: branch prediction + squashing

- 2 cycle penalty for misprediction
- (correct misprediction after jXX finishes execute)

ret control hazard: 3 cycles of stalling

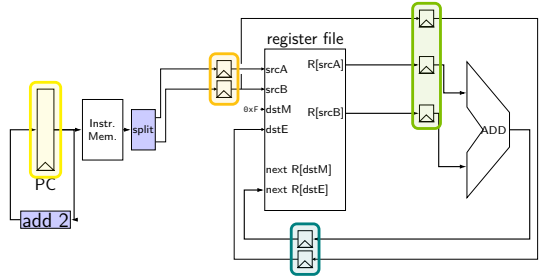
- (fetch next instruction after ret finishes memory)

backup slides

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)

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

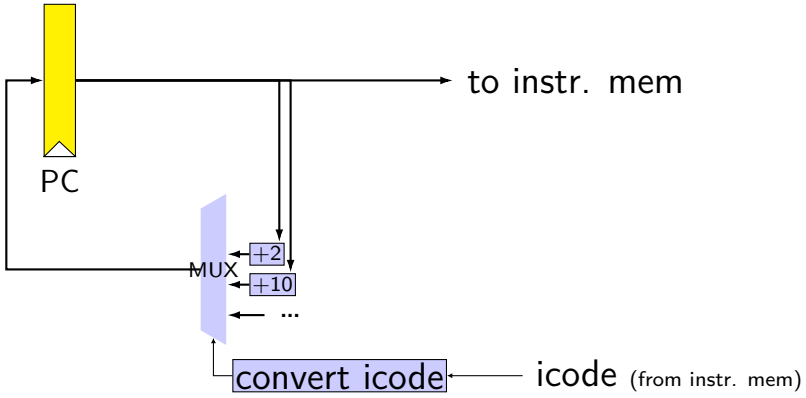
```
        call empty
        addq %r8, %r9

empty:  ret
```

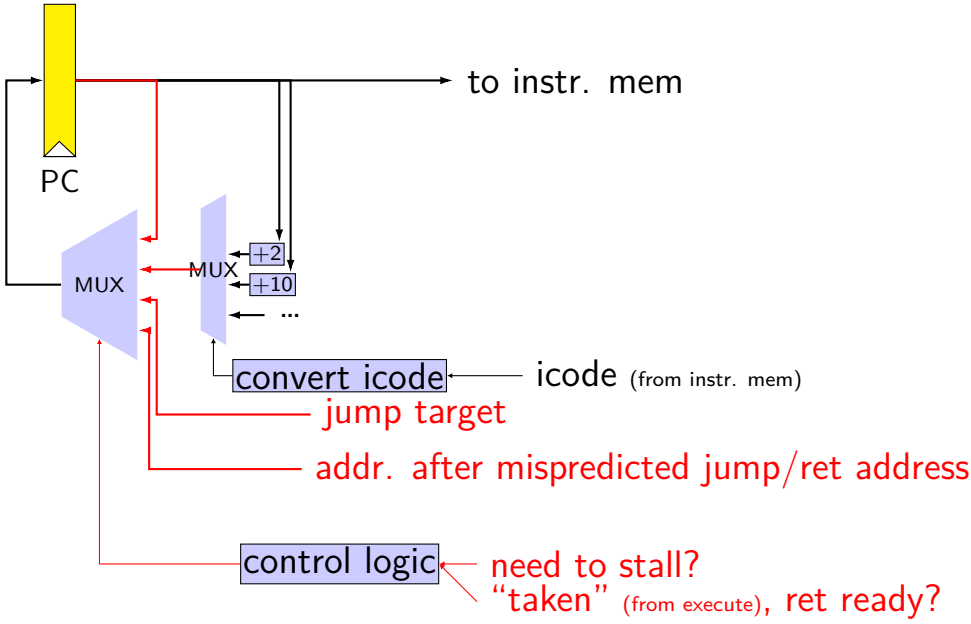
time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	call (store)	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq	nothing	nothing	nothing	ret

return address loaded here

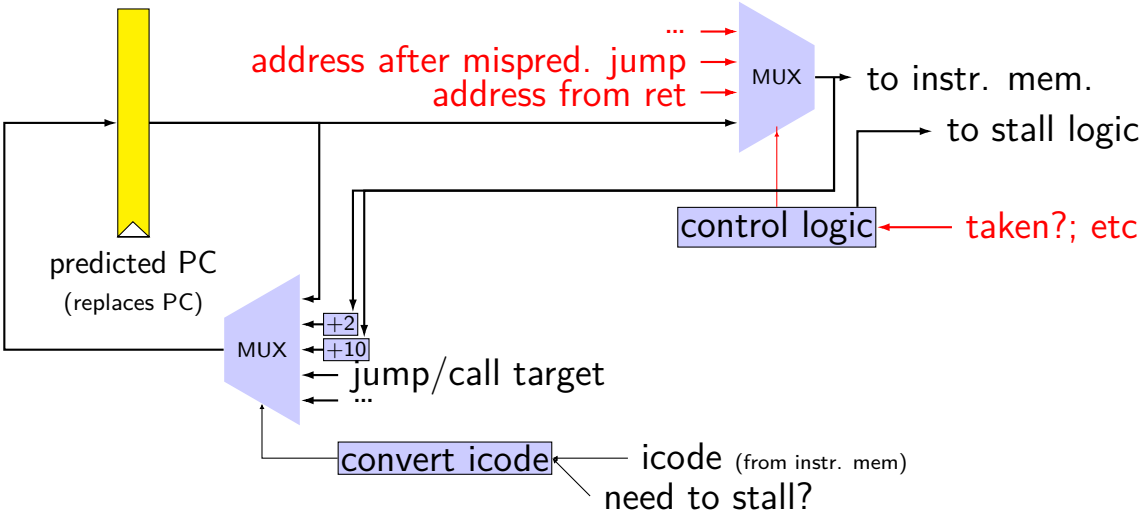
PC update (adding prediction, stall)



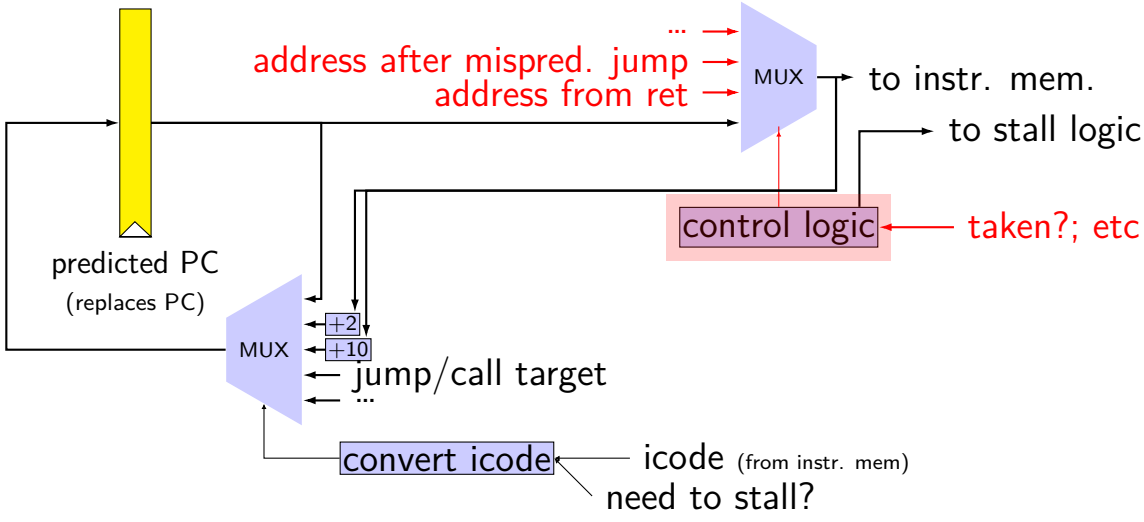
PC update (adding prediction, stall)



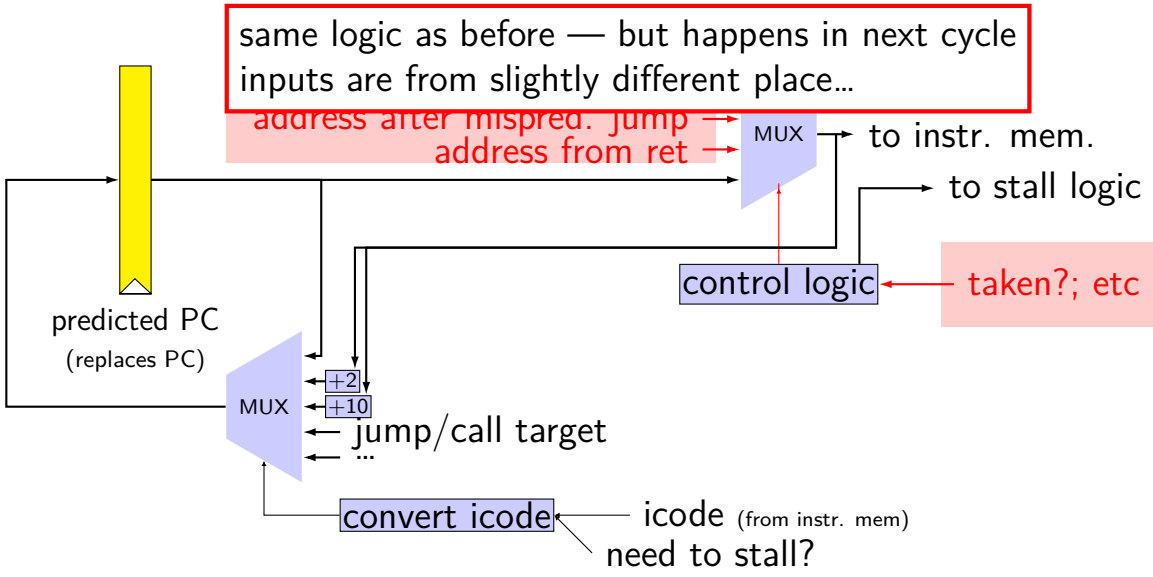
PC update (rearranged)



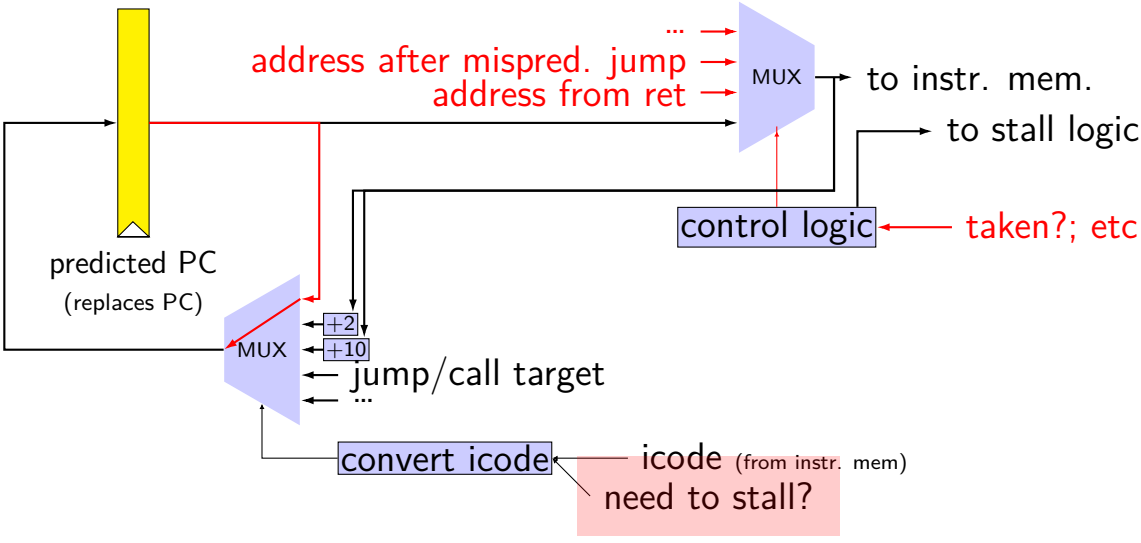
PC update (rearranged)



PC update (rearranged)



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;
];
```

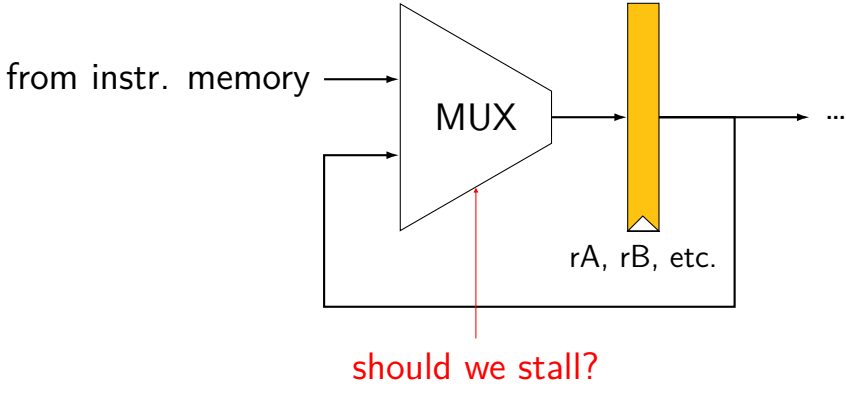
why rearrange PC update?

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?

fetch/decode logic — advance or not



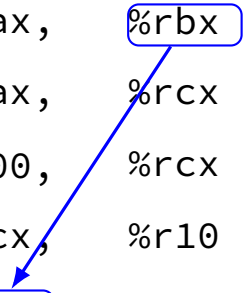
ex.: dependencies and hazards (1)

```
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 wiht forwarding?

ex.: dependencies and hazards (1)

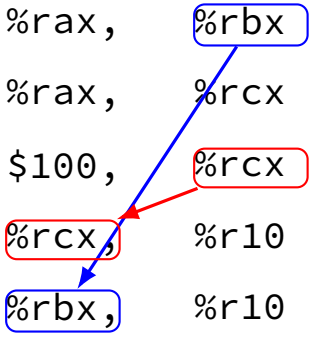
```
addq    %rax,   %rbx
subq    %rax,   %rcx
irmovq  $100,   %rcx
addq    %rcx,   %r10
addq    %rbx,   %r10
```



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

ex.: dependencies and hazards (1)

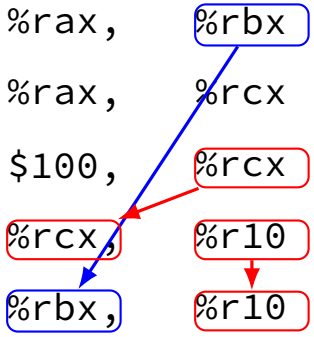
```
addq    %rax, %rbx
subq    %rax, %rcx
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```



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

ex.: dependencies and hazards (1)

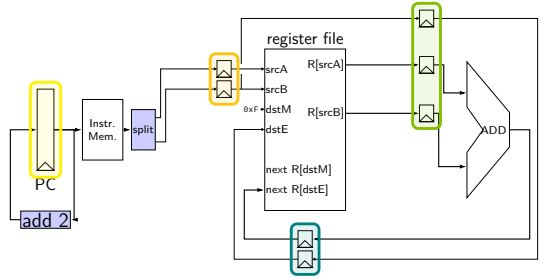
```
addq    %rax, %rbx
subq    %rax, %rcx
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```



where are dependencies?
which are hazards in our pipeline?
which are resolved wiht 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



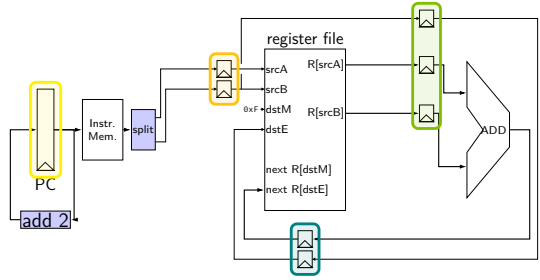
pipeline register delay: 10ps

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



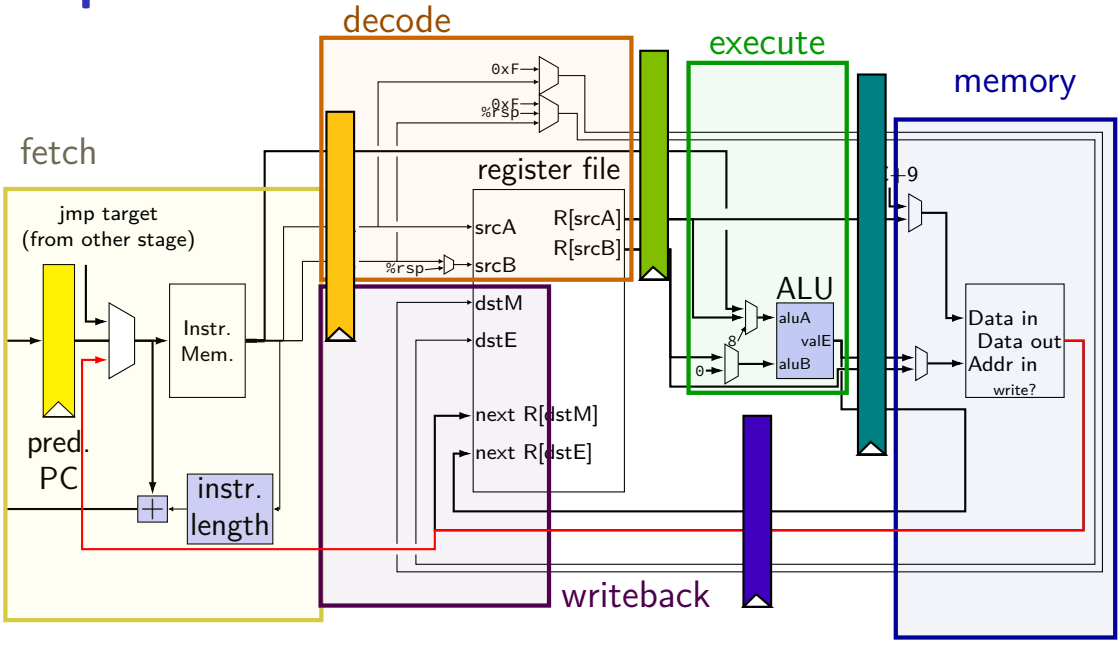
pipeline register delay: 10ps

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

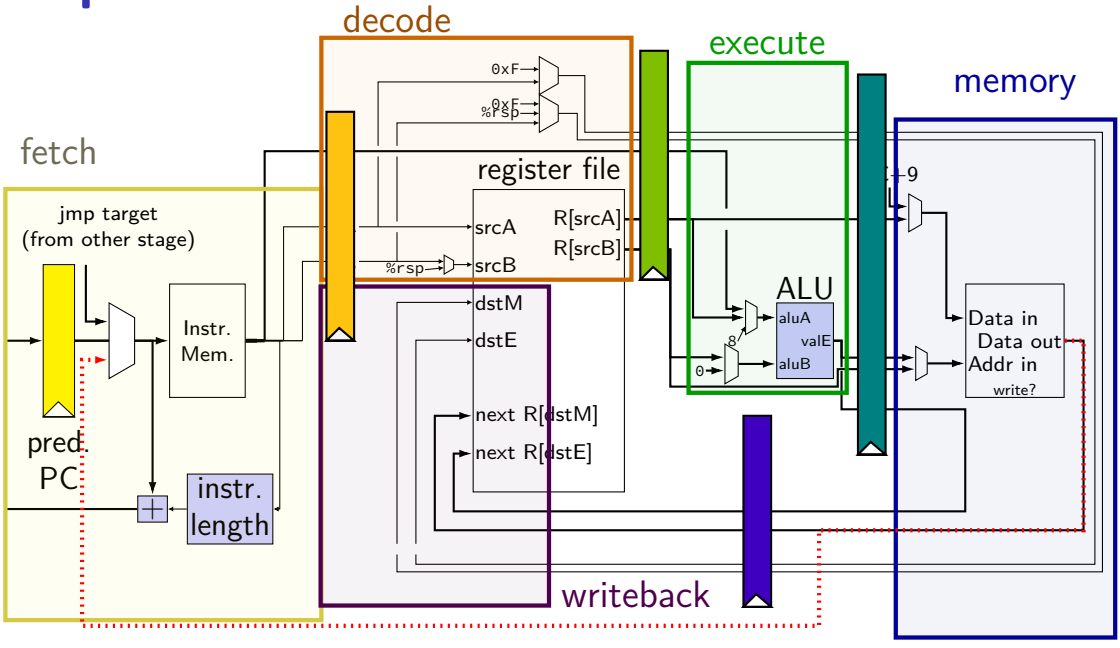
A. 2.00x **B.** 1.70x to 1.99x
C. 1.60x to 1.69x **D.** 1.50x to 1.59x
E. less than 1.50x

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

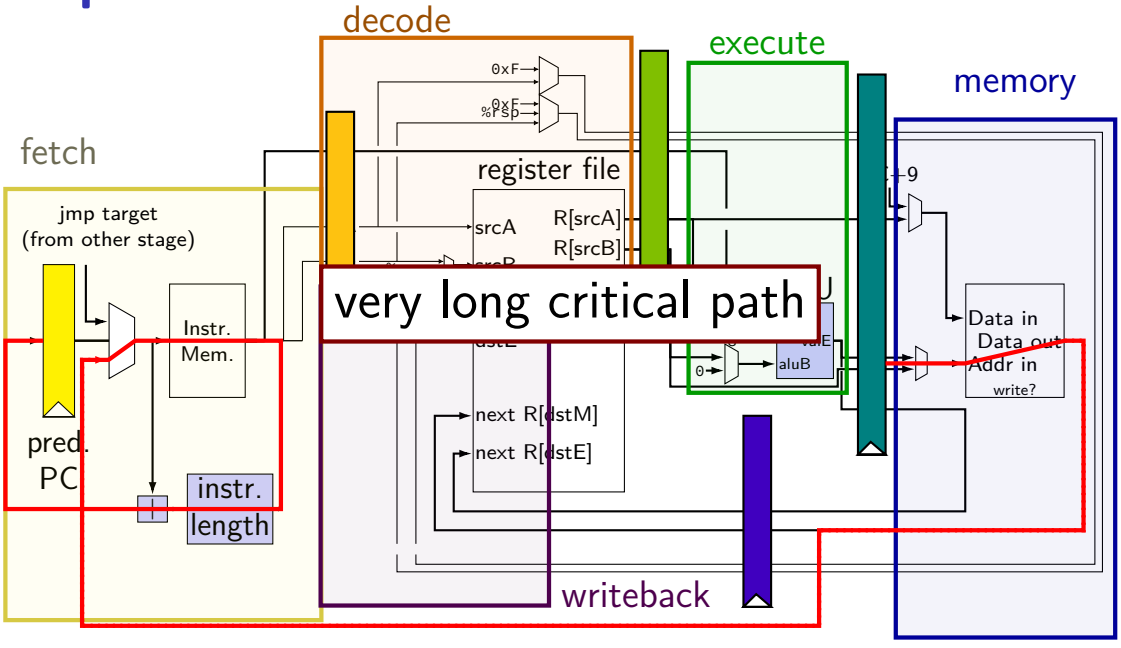
ret paths


decode
execute
memory
fetch
register file
0xF
%rsp
srcA
srcB
R[srcA]
R[srcB]
jmp target
(from other stage)
Instr.
Mem.
dstM
dstE
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
0
+9
Data in
Data out
Addr in
write?
pred.
PC
instr.
length
writeback

ret paths


decode
execute
memory
fetch
register file
0xF
0xF
%rsp
srcA
srcB
%rsp
R[srcA]
R[srcB]
jmp target
(from other stage)
Instr.
Mem.
dstM
dstE
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
0
+9
Data in
Data out
Addr in
write?
pred.
PC
instr.
length
writeback

ret paths


decode
execute
memory
fetch
register file
jmp target
(from other stage)
srcA
R[srcA]
R[srcB]
very long critical path
Instr.
Mem.
Data in
Data out
Addr in
write?
aluB
next R[dstM]
next R[dstE]
pred.
PC
instr.
length
writeback