last time

hazard — pipelined processor does the wrong thing w/o changes

- caused by dependencies between instructions
- which dependencies cause hazards? dpeends on pipeline

data hazard — reading outdated value from register

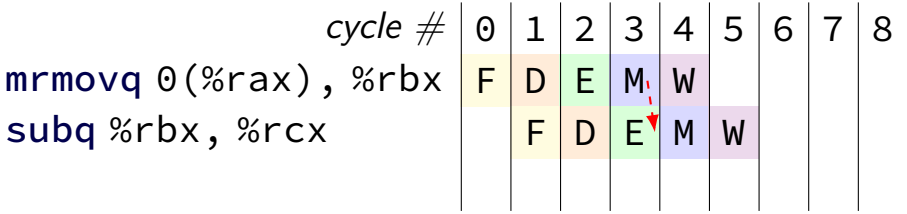
stalling — add nops (delay instruction with hazard) to resolve hazard

forwarding to resolve data hazards

- in data hazard, value to be written often computed in future stage
- add wire from future stage to after register read
- use MUX to select between value from register file and future stage value

logistics: no lab next week / quiz due Thursday

unsolved problem



combine stalling and forwarding to resolve hazard

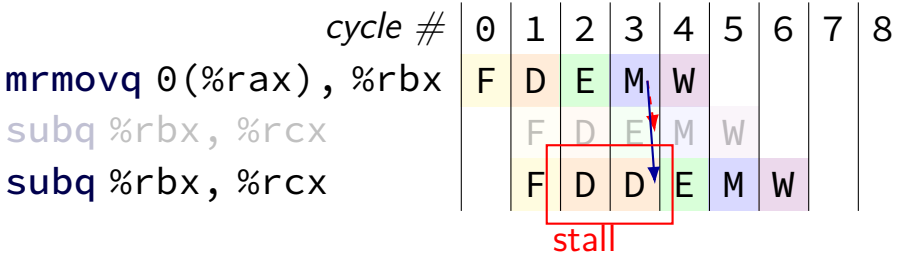
assumption in diagram: hazard detected in `subq`'s decode stage

(since easier than detecting it in fetch stage)

typically what you'll implement

intuition: try to forward, but detect that it won't work

unsolved problem



combine stalling and forwarding to resolve hazard

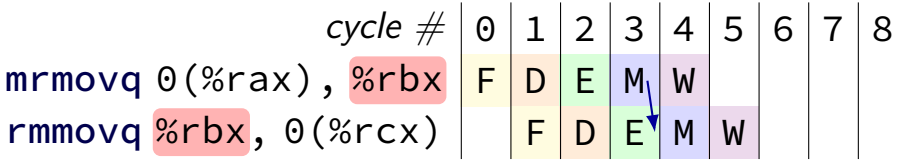
assumption in diagram: hazard detected in `subq`'s decode stage

(since easier than detecting it in fetch stage)

typically what you'll implement

intuition: try to forward, but detect that it won't work

solveable problem



common for real processors to do this
but our textbook only forwards to the end of decode

aside: forwarding timings

forwarding: adds MUXes for forwarding to critical path

- might slightly increase cycle time, considered acceptable

should not add much more to critical path:

- example: can't use value read from memory in ALU in same cycle

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage   // 5 stage
addq %rax, %r8      //           // W
subq %rax, %r9      // W         // M
xorq %rax, %r10     // EM        // E
andq %r8,  %r11     // D         // D
```

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage   // 5 stage
addq %rax, %r8      //           // W
subq %rax, %r9      // W         // M
xorq %rax, %r10     // EM        // E
andq %r8,  %r11     // D         // D
```

addq/andq is hazard with 5-stage pipeline

addq/andq is **not** a hazard with 4-stage pipeline

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

result only available near end of second execute stage

where does forwarding, stalls occur?

	cycle #	0	1	2	3	4	5	6	7	8
(1) `addq %rcx, %r9`		F	D	E1	E2	M	W			
(2) `addq %r9, %rbx`										
(3) `addq %rax, %r9`										
(4) `rmmovq %r9, (%rbx)`										
(5) `rrmovq %rcx, %r9`										

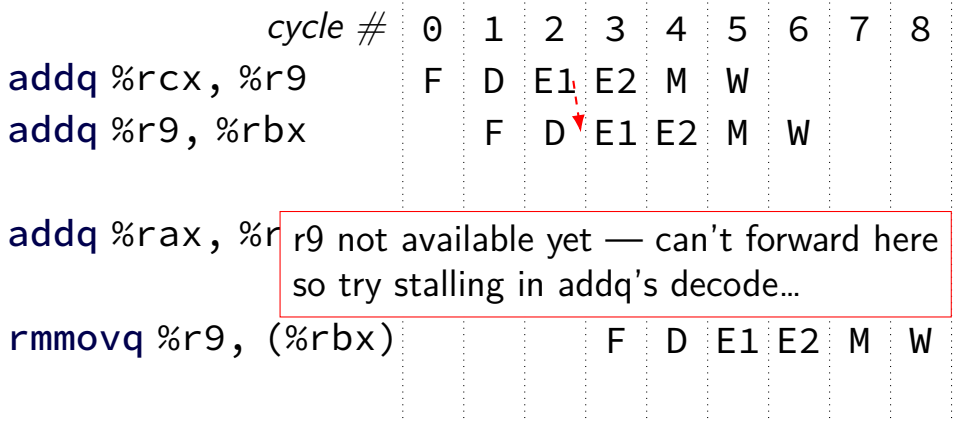
exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`	F	D	E1	E2	M	W			
`addq %r9, %rbx`									
`addq %rax, %r9`									
`rmmovq %r9, (%rbx)`									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W



exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

	cycle #	0	1	2	3	4	5	6	7	8	
`addq %rcx, %r9`		F	D	E1	E2	M	W				
`addq %r9, %rbx`			F	D	E1	E2	M	W			
`addq %r9, %rbx`			F	D	D	E1	E2	M	W		
`addq %rax, %r9`		after stalling once, now we can forward									
`addq %rax, %r9`				F	F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`					F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`						F	D	E1	E2	M	W

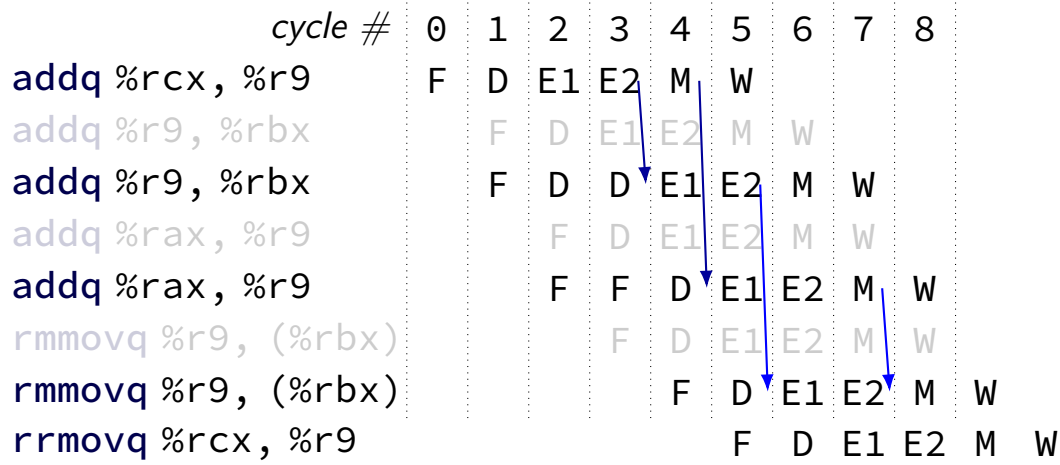
exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
`addq %rcx, %r9`	F	D	E1	E2	M	W				
`addq %r9, %rbx`		F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	D	E1	E2	M	W		
`addq %rax, %r9`			F	D	E1	E2	M	W		
`addq %rax, %r9`			F	F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`				F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`					F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/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
```

	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	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	S							
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

wait for two cycles for addq to update SF/ZF

control hazard: stall

```
addq %r8, %r9
// insert two nops
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*								
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)

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?

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

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

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

stalling costs

with only stalling:

up to 3 extra cycles for data dependencies

extra 3 cycles (total 4) f

trick: use values waiting to get to register file

extra 2 cycles (total 3) for conditional jmp

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

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

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

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 [?]	a			
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

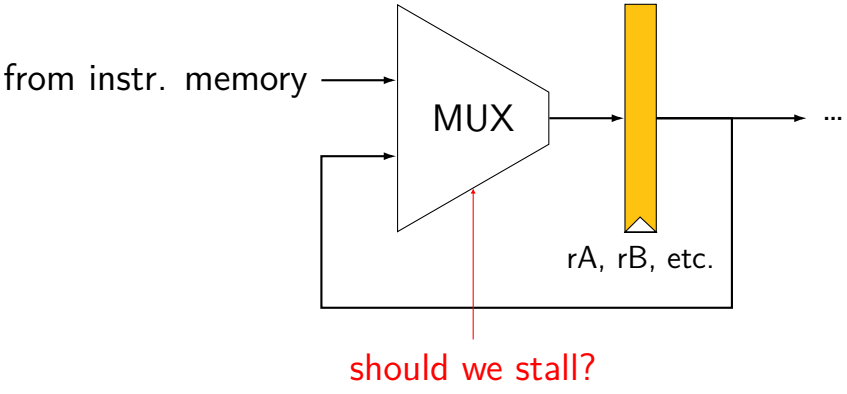
kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
ret	1%	4	4
others	91%	1*	1*

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

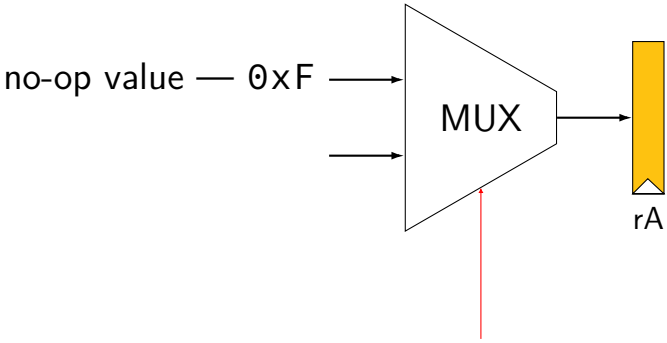
stall: $3 \times .03 + 3 \times .05 + 4 \times .01 + 1 \times .91 =$ 1.19 cylces/instr. ($1.19 \div 1.09 \approx 1.09$x faster)

* — ignoring data hazards

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		

backup slides

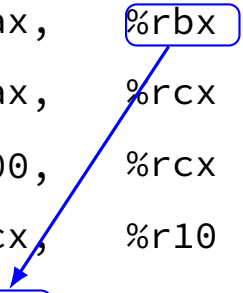
ex.: dependencies and hazards (1)

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

where are dependencies?
which are hazards in our pipeline?

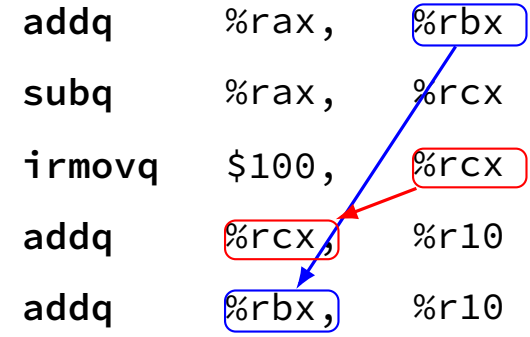
ex.: dependencies and hazards (1)

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



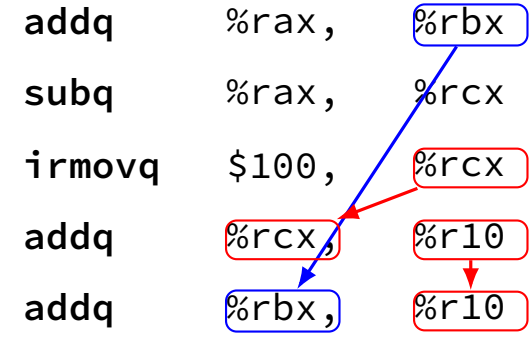
where are dependencies?
which are hazards in our pipeline?

ex.: dependencies and hazards (1)



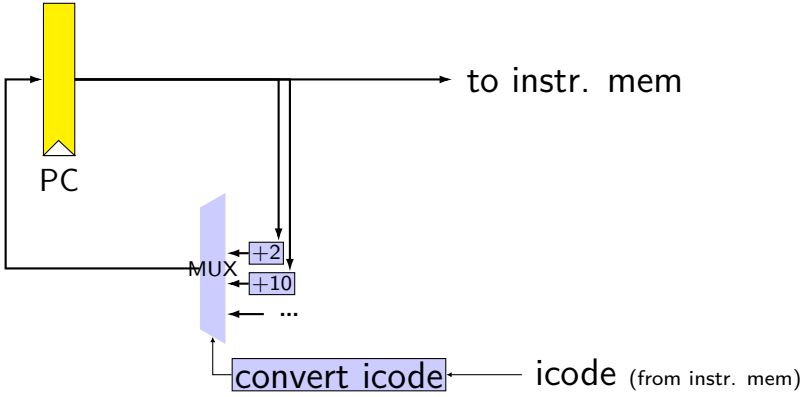
where are dependencies?
which are hazards in our pipeline?

ex.: dependencies and hazards (1)

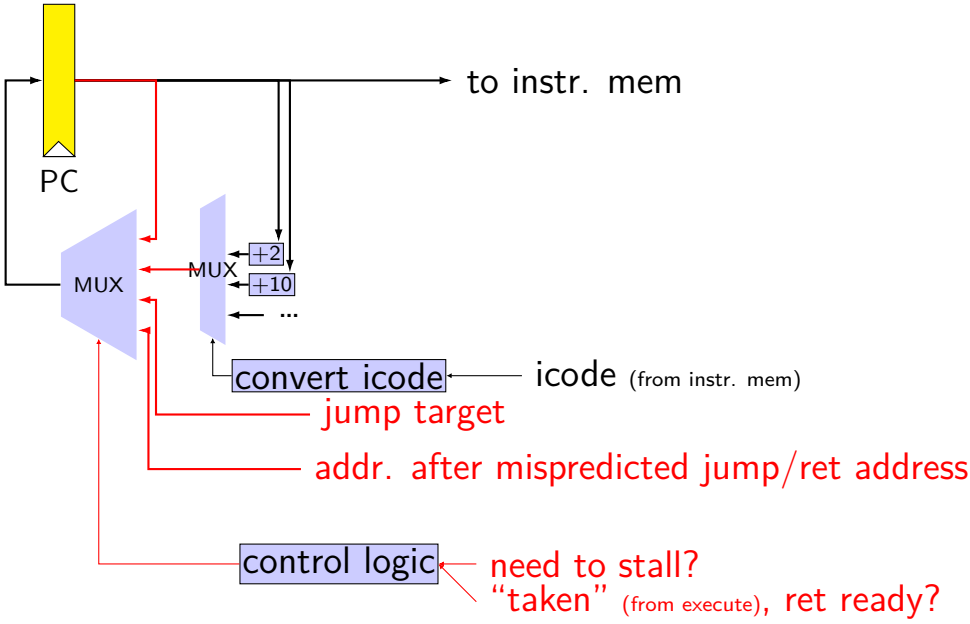


where are dependencies?
which are hazards in our pipeline?

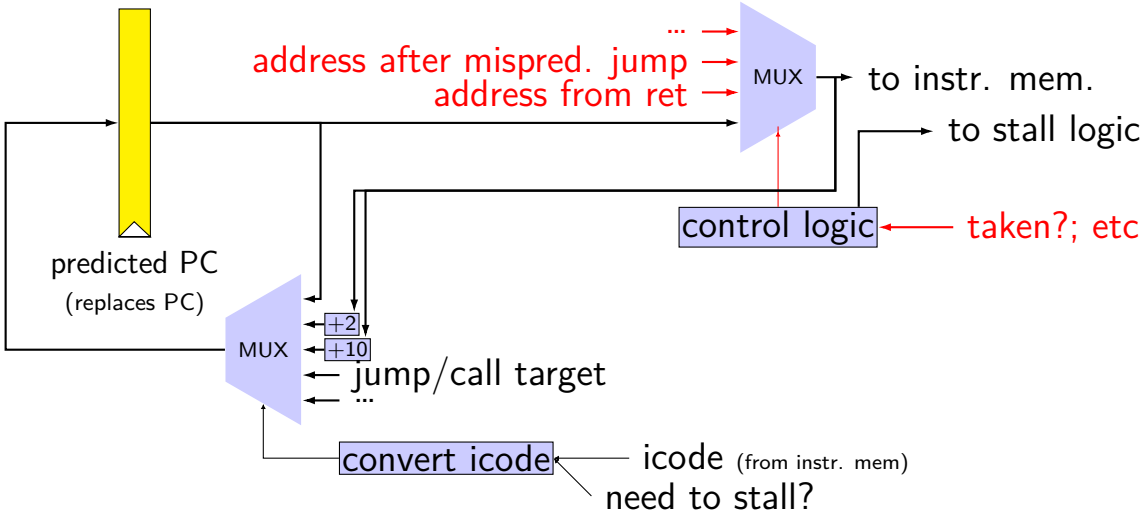
PC update (adding prediction, stall)



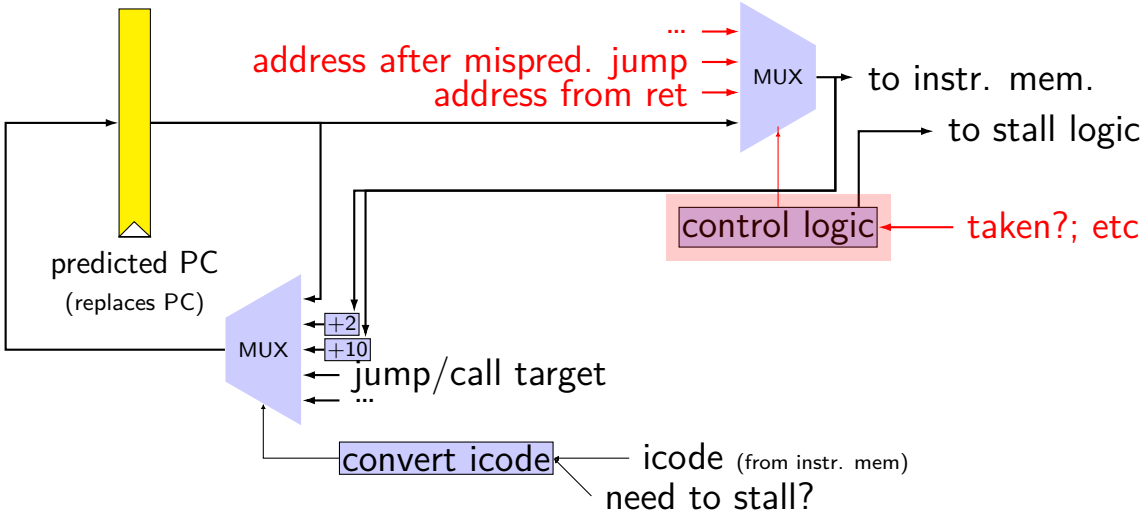
PC update (adding prediction, stall)



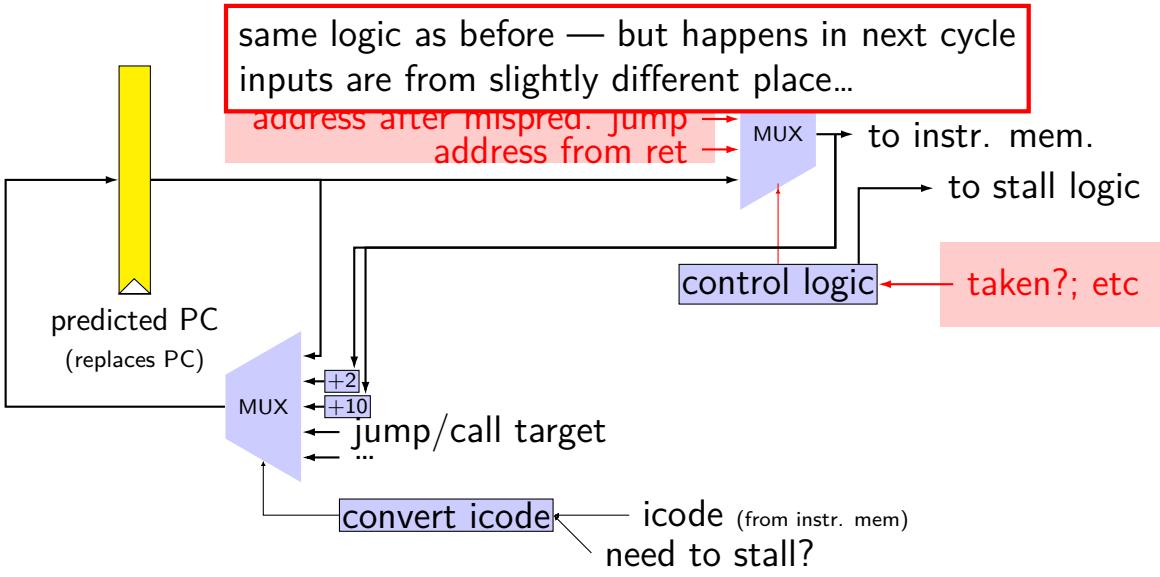
PC update (rearranged)



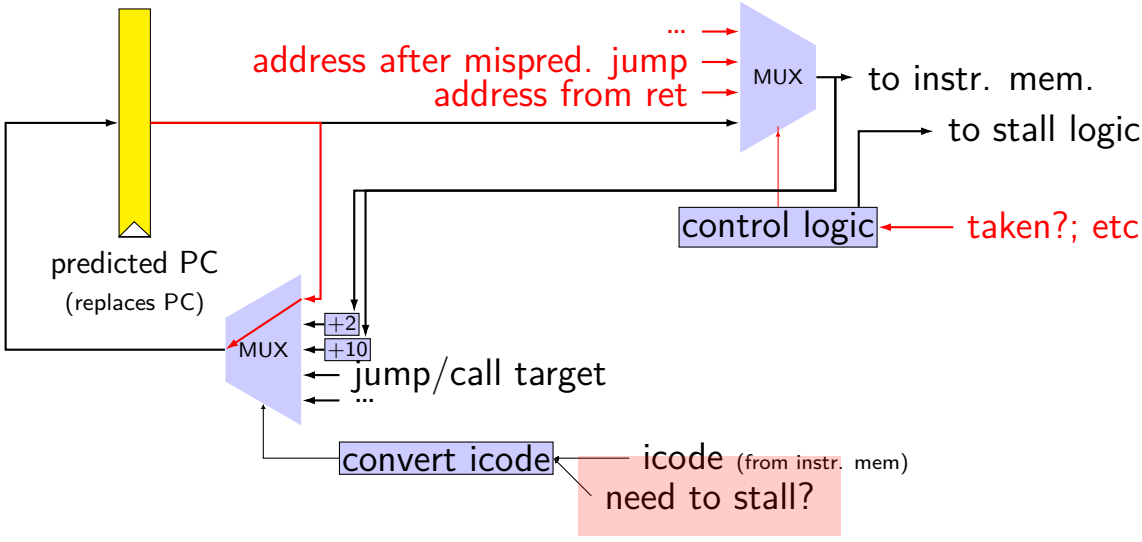
PC update (rearranged)



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;
];
```

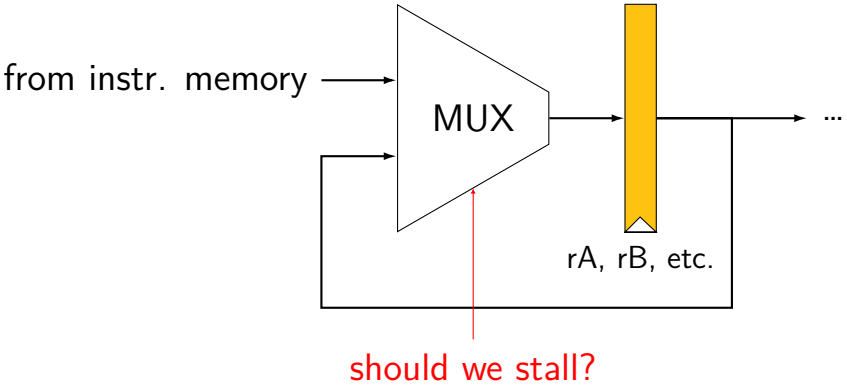
why rearrange PC update?

either works

- correct PC at beginning or end of cycle?
- still some time in cycle to do so...

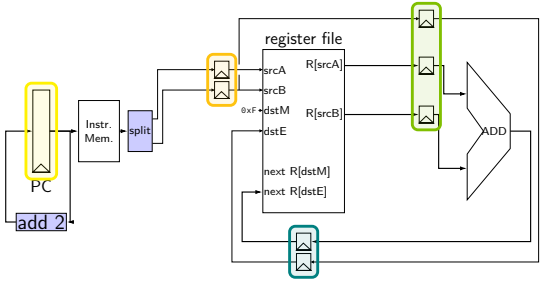
maybe easier to think about branch prediction this way?

fetch/decode logic — advance or not



exercise

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



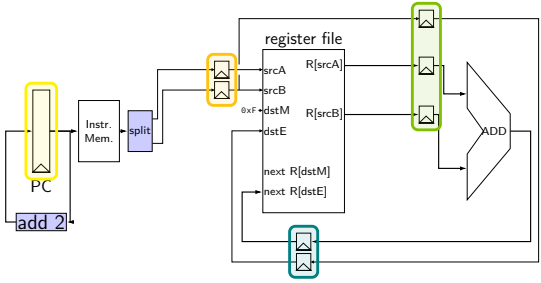
pipeline register delay: 10ps

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

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