pipelining: forwarding/...

last time

dividing single-cycle processor into pipeline stages

critical paths and clock cycle lengths

- paths from rising edge event to next rising edge event
- not double-counting register delays
- slowest path determines cycle time

hazards and dependencies

- dependency = instruction needs another result
- hazard = pipeline handles dependency wrong (w/o help)

data hazards: reading data before written

solving hazards with stalling: hardware inserts nops

forwarding

- use about-to-be-written value in place of read

quiz Q1

critical path = longest path

selected candidates:

- PC → instr mem → PC incr → PC
  - 120 + 20 ps + small reg delay
- PC → instr mem → pipeline registers
  - 120 ps + small reg delay
- pipeline registers → register read → ALU → memory write
  - 100 + 200 + 150 + small reg delay
- pipeline registers → register read → ALU → memory read → register write
  - 100 + 200 + 120 + 100 + small reg delay = 520 + small reg delay

quiz Q3

3000 ps with 1 stage

with 8 stages, ideal case:

- $3000/8 = 375$ ps stages + very tiny register delays

with 8 stages, worst case:

- one stages does almost 3000 ps of work + register delays
- could be *worse* than original processor

told in problem there is a performance improvement

375,3000

quiz Q4

pipeline registers from FD (first stage) to EMW (second stage)

✓ opcode field

second stage needs to know what to do with memory, etc.

✓ one or more register indices

to implement writing to those registers

the result of an ALU operation

not computed in fetch or decode

✓ the values of registers read from the register file

computed by decode

needed (at least sometimes) by all later stages

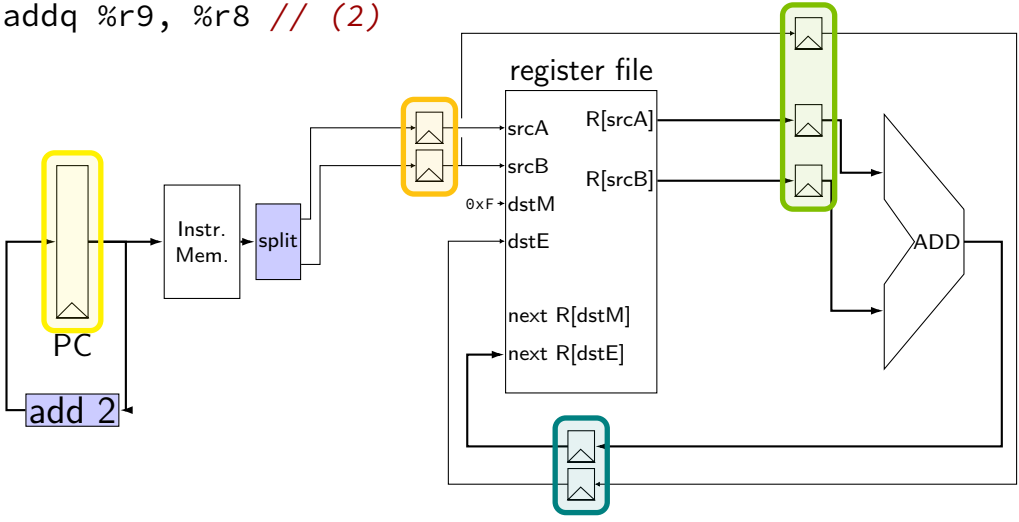
quiz Q5

stages: F / DE / MW

```
cycle                1   2   3   4   5   6   7

addq %r8, %r9        F   DE  MW
addq %r10, %r9           F   x   DE  MW
addq %r8, %r10               F   DE  MW
addq %r8, %r9                    F   DE  MW
```

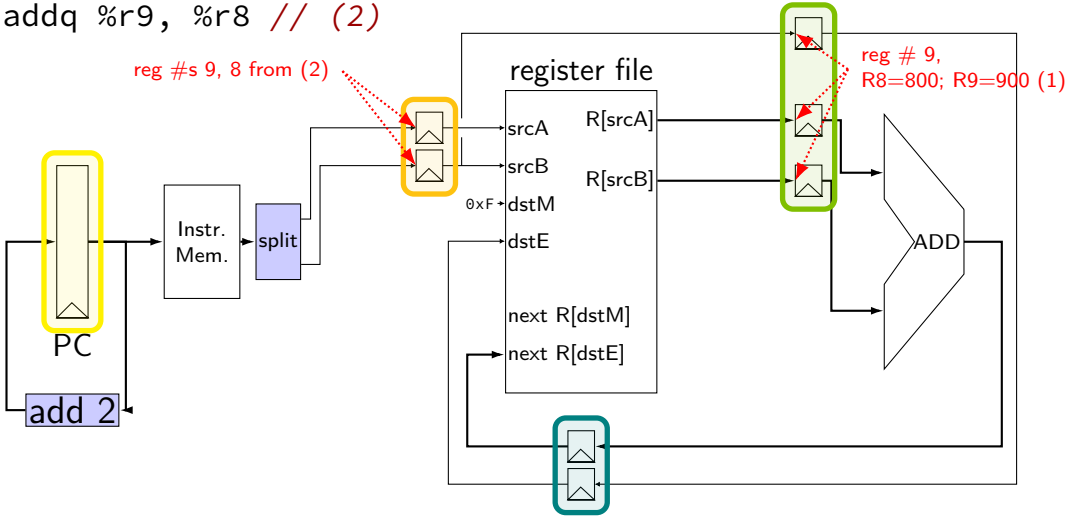
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



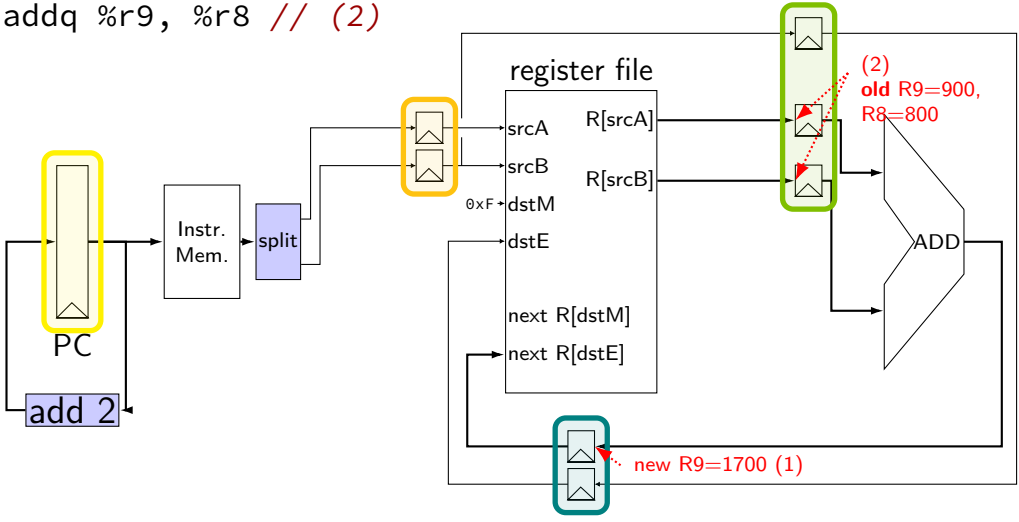
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



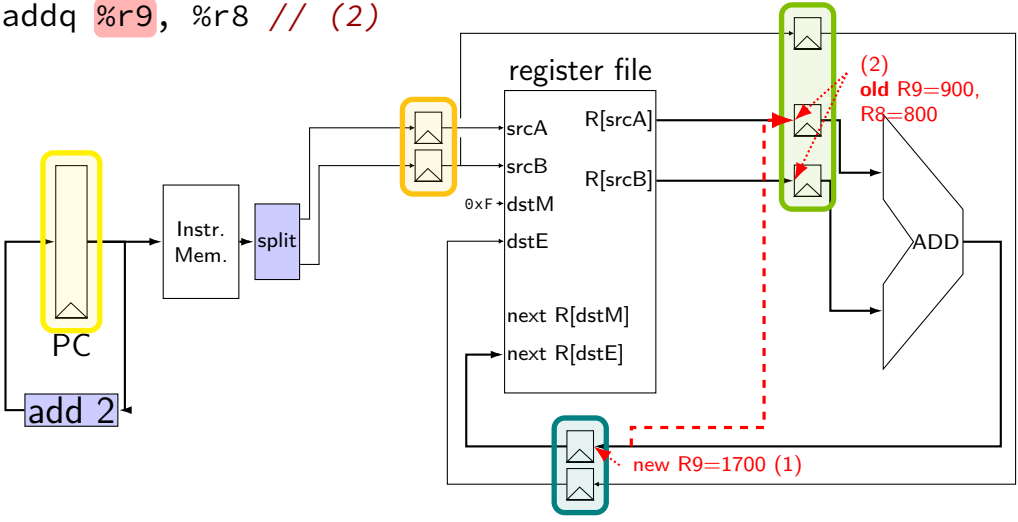
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



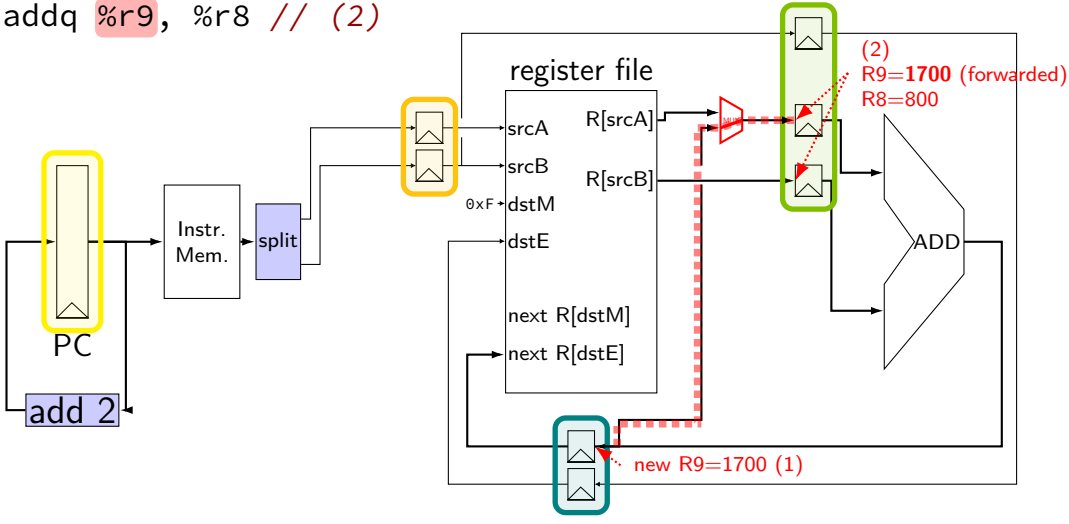
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



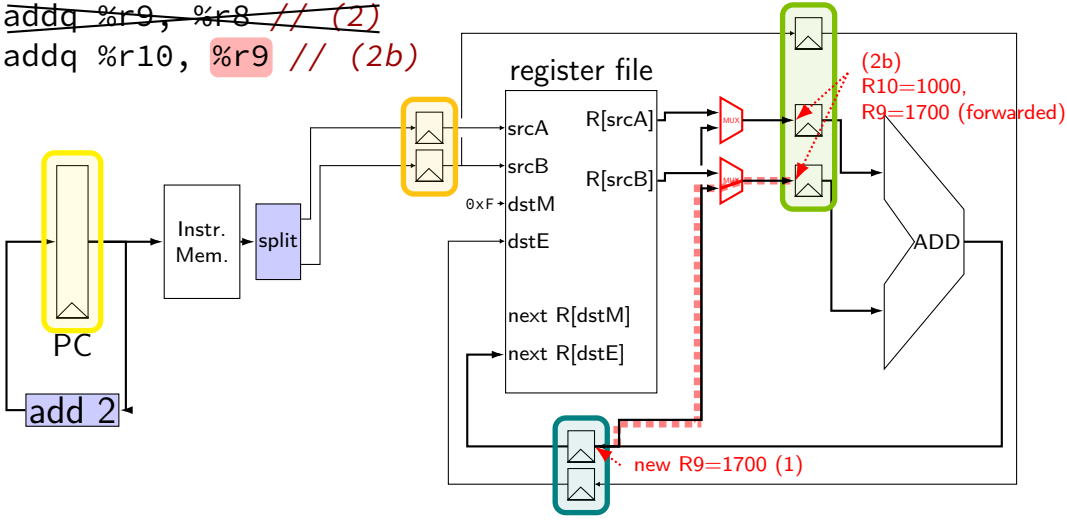
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



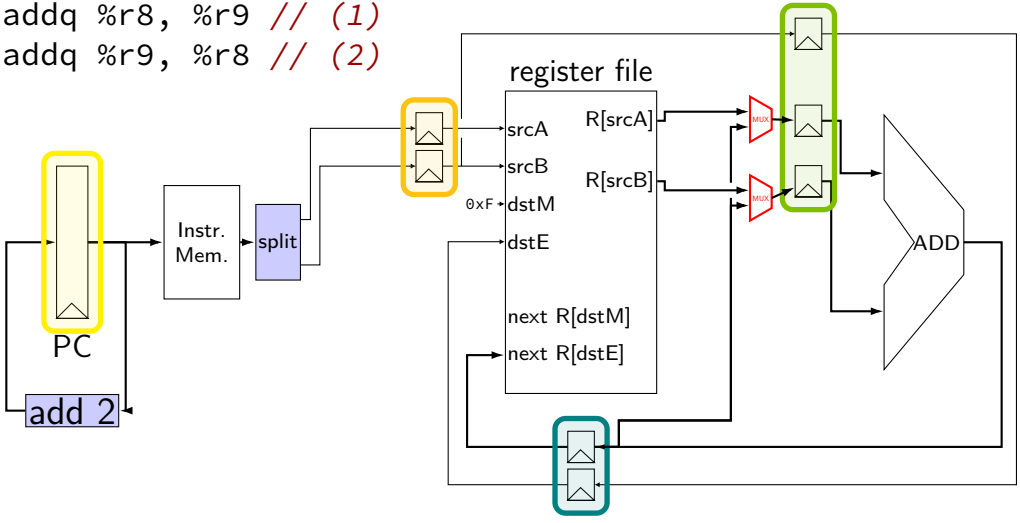
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
addq %r10, %r9 // (2b)
```



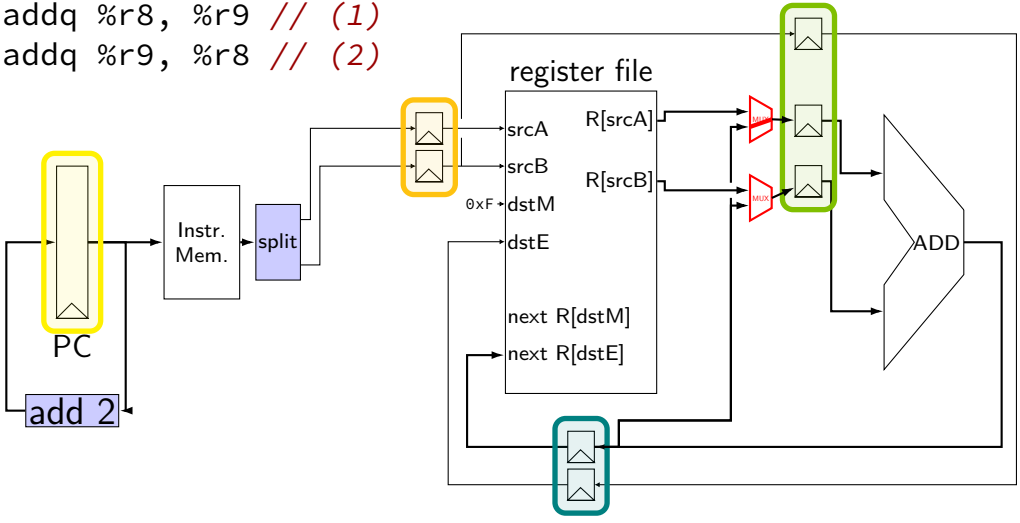
forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



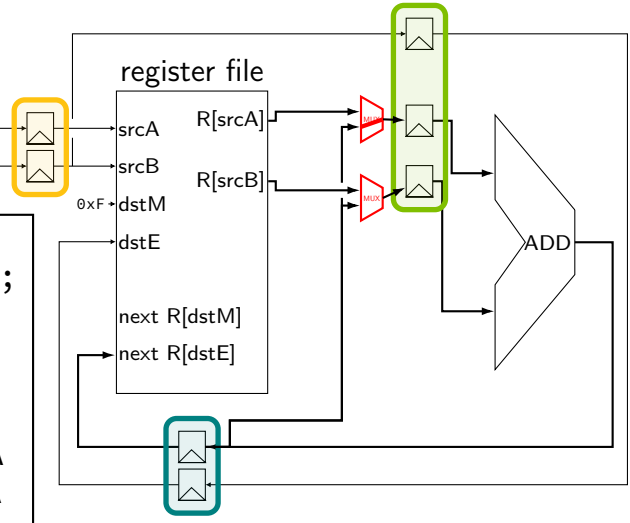
forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



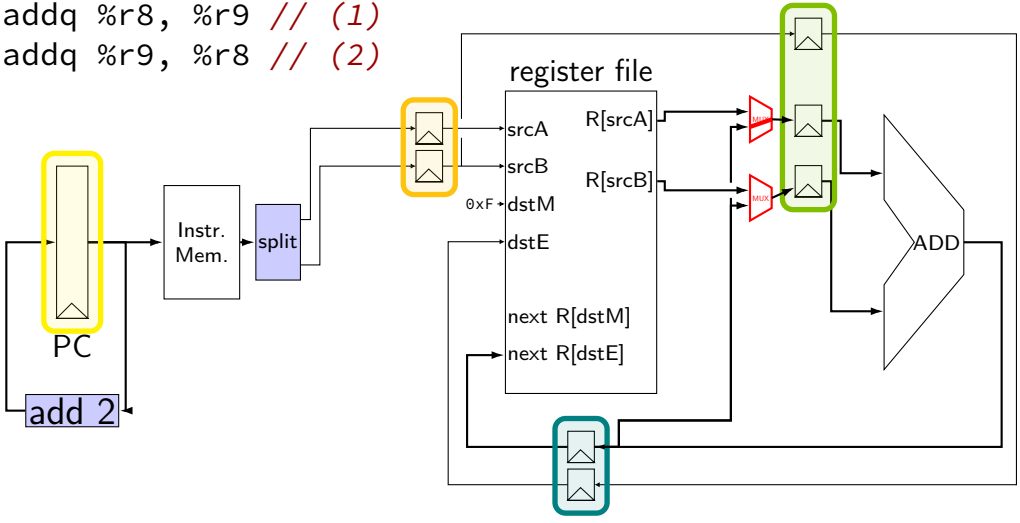

```
d_valA= [
   condition : e_valE;
   1 : reg_outputA;
];
```

What could `condition` be?

a. `W_rA == reg_srcA`
b. `W_dstE == reg_srcA`
c. `e_dstE == reg_srcA`
d. `d_rB == reg_srcA`
e. something else

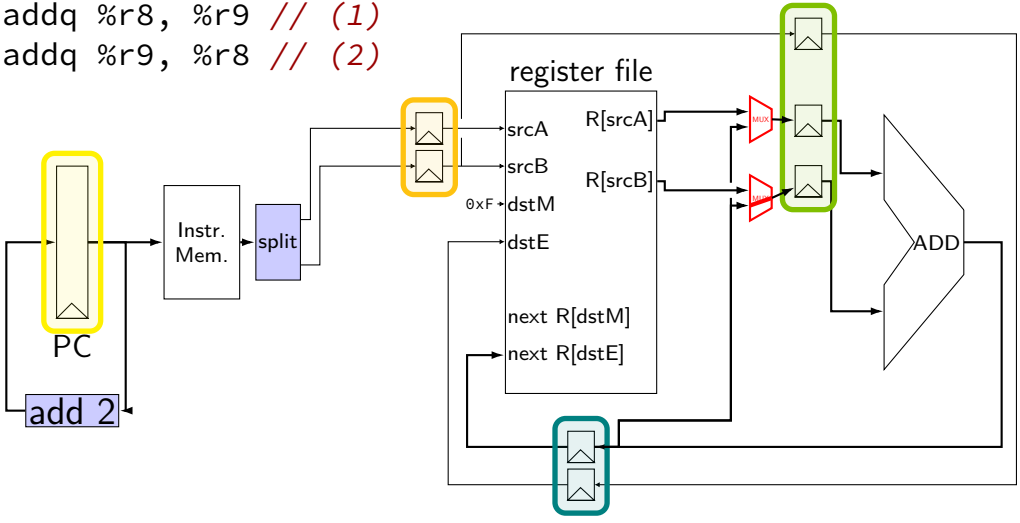
forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```

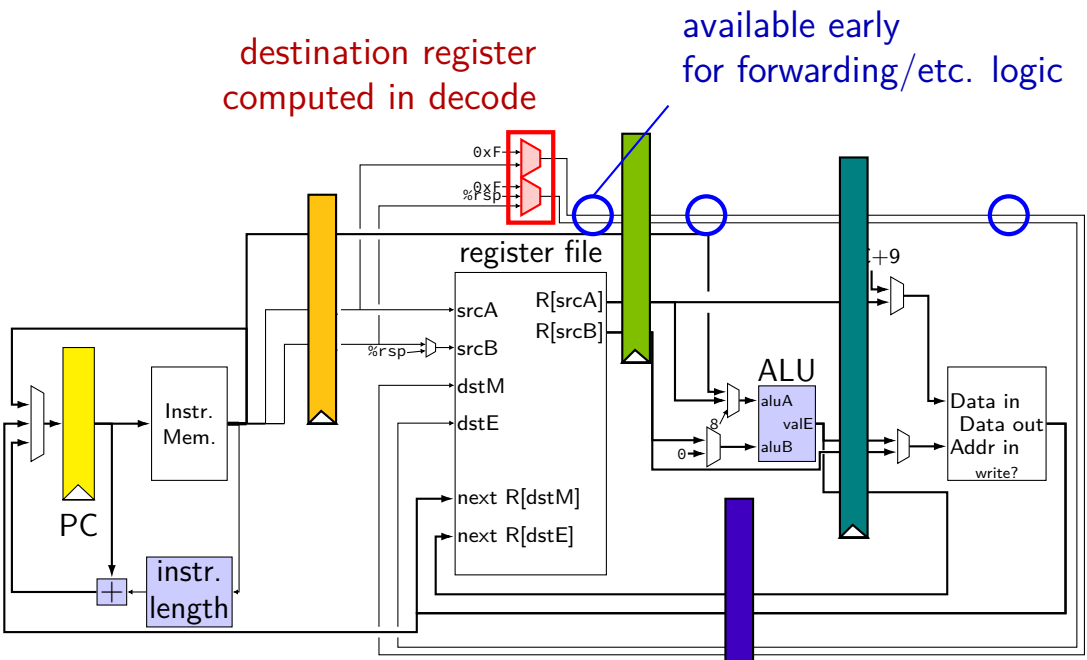


forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



computing destinations early



textbook convention on destinations

dstE/dstM computed mostly in decode

- passed through pipeline as d_dstE, e_dstE, ...

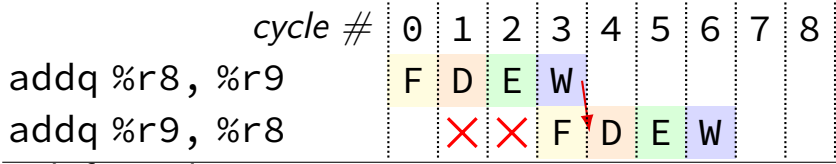
valE/valM only set to value to be stored in dstE/dstM

- passed through pipeline as e_valE, m_valE, ...

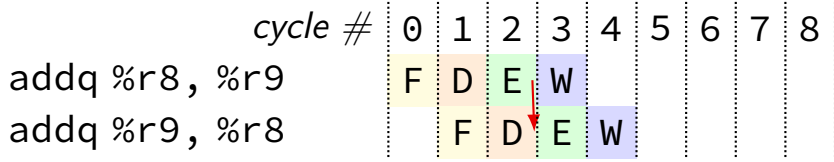
simplifies forwarding/stalling logic

stalling versus forwarding (1)

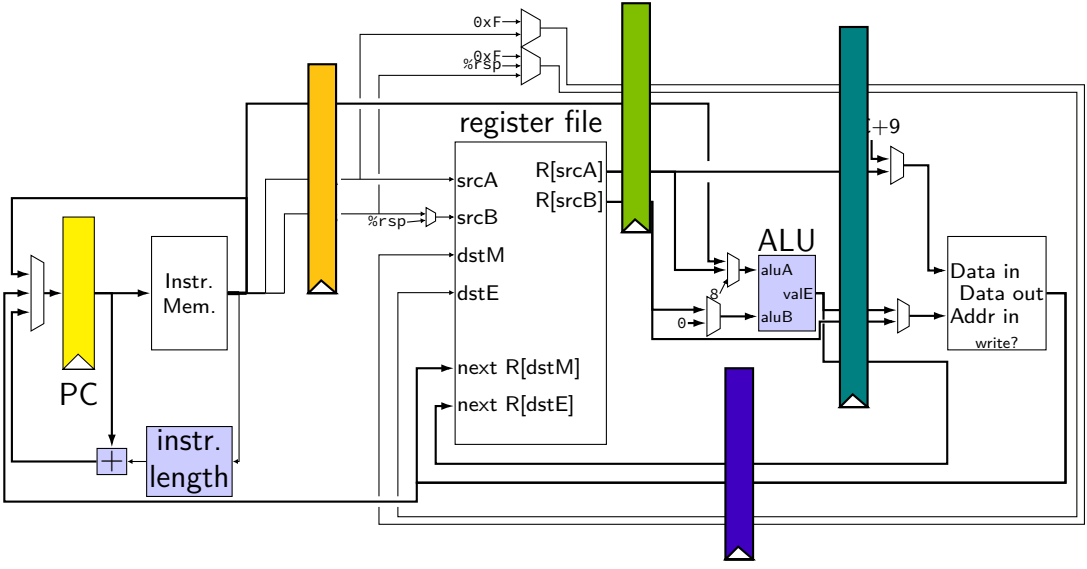
with stalling:



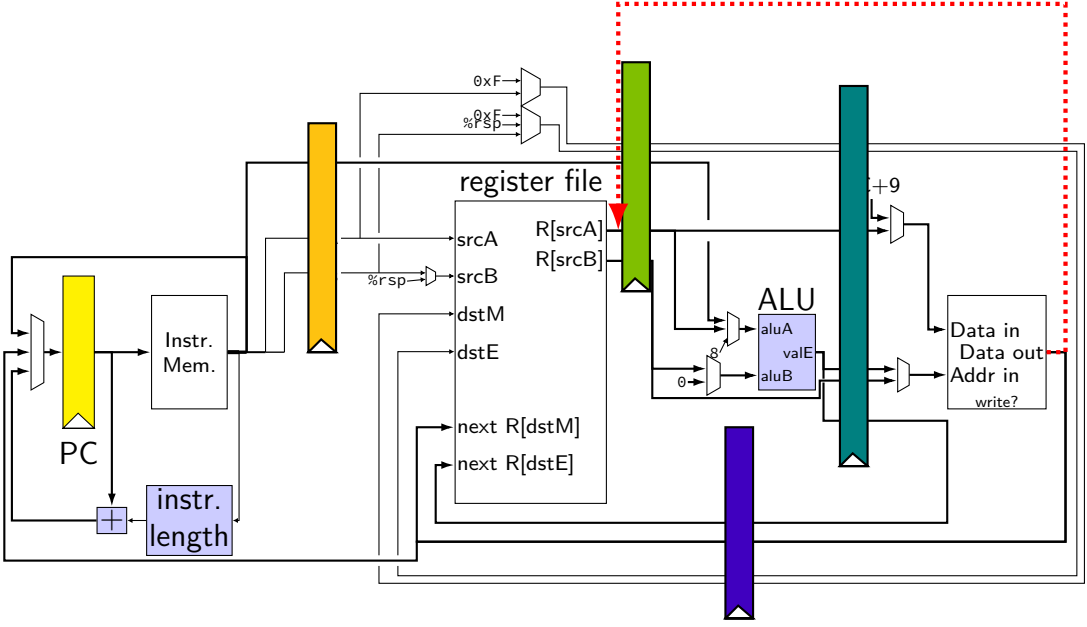
with forwarding:



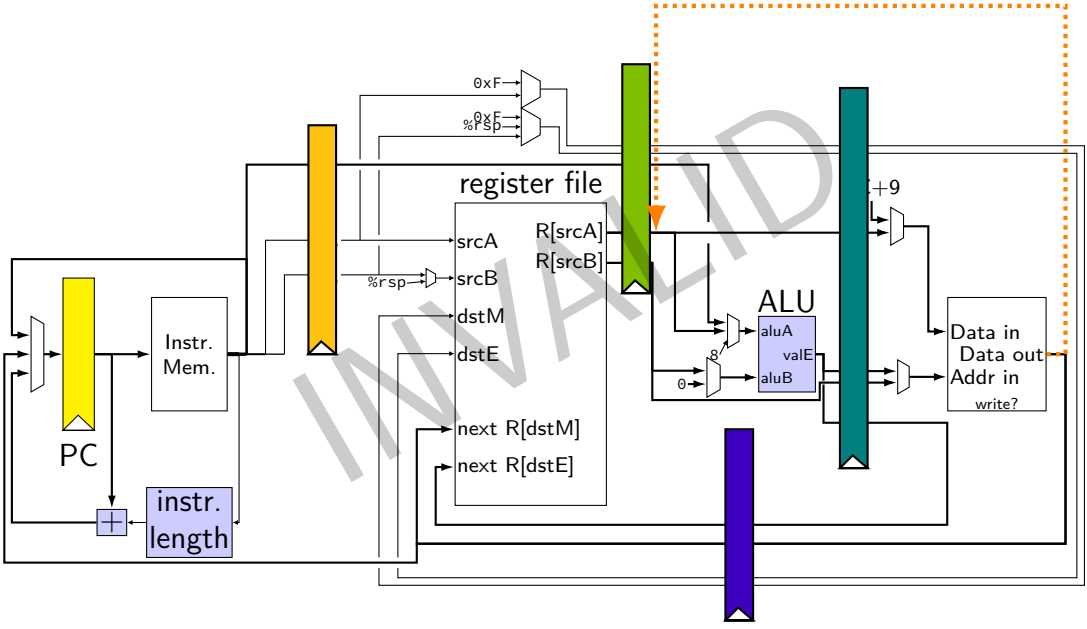
more possible forwarding



more possible forwarding

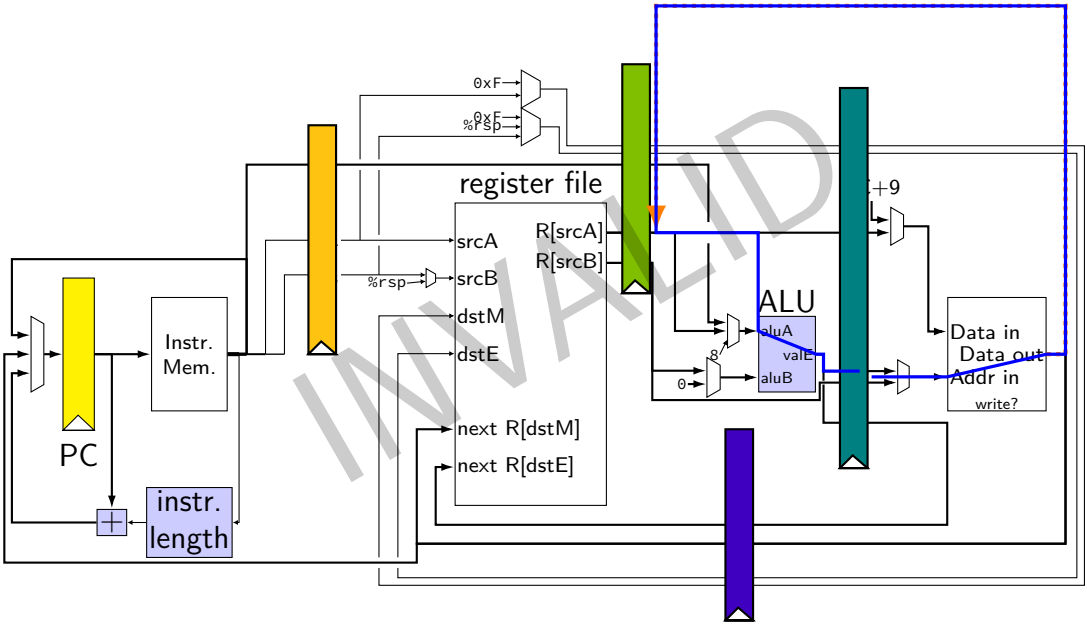


INVALID forwarding path



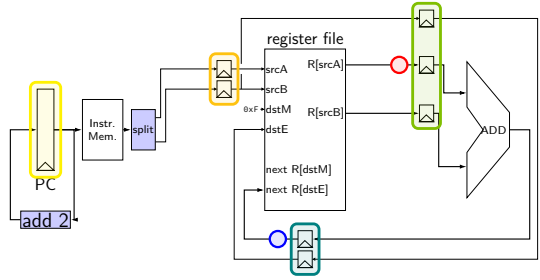
INVALID forwarding path

oops, extra long critical path!



forwarding more?

location of values during cycle 3:



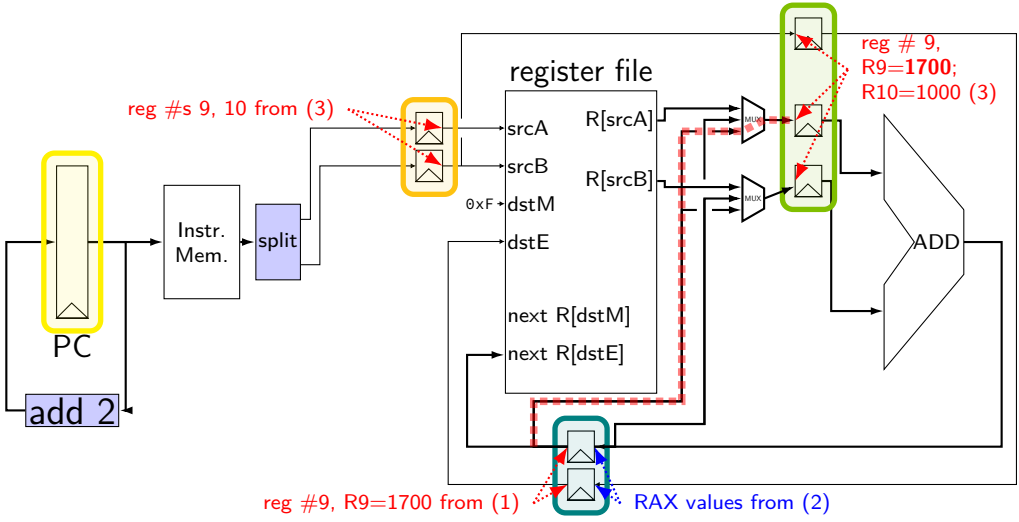

```
// initially %rax = 0,
//           %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %rax, %rax
addq %r9, %r10
```

	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	0x4	0	0	800	900	9		
3		9	10	0	0	0	1700	9
4				900	1000	10	0	0
5							1900	10

should be 1700

forwarding two stages?

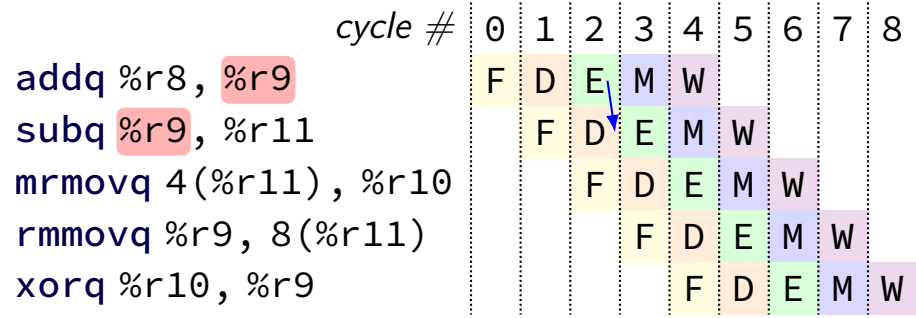
```
addq %r8, %r9   // (1) R9 <- R8 (800) + R9 (900)
addq %rax, %rax // (2)
addq %r9, %r10  // (3) R10 <- R9 (1700) + R10 (1000)
```



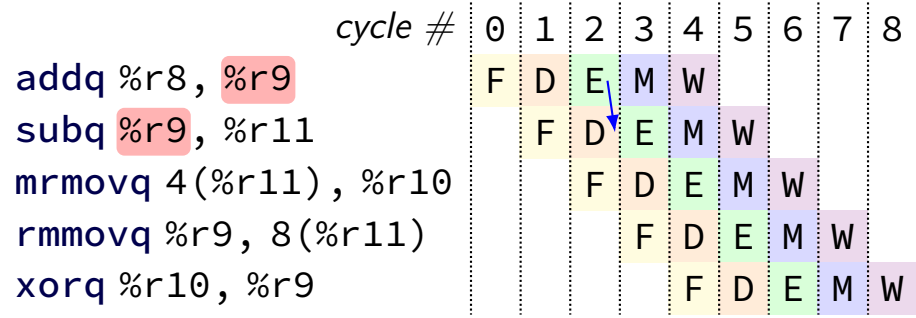
some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

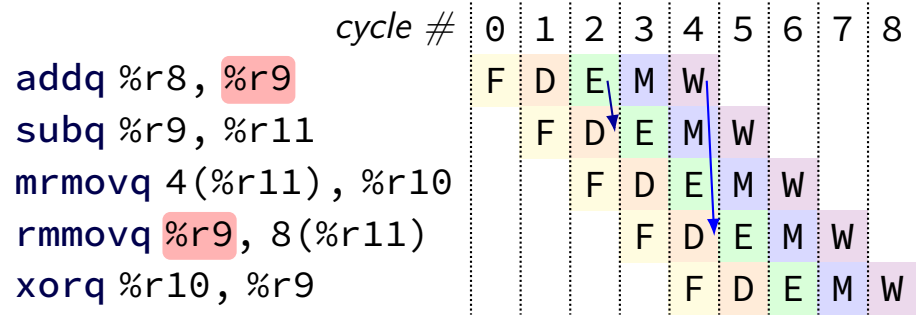
some forwarding paths



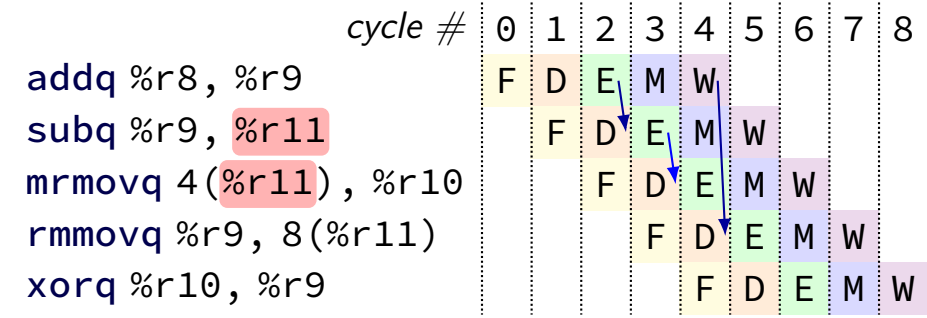
some forwarding paths



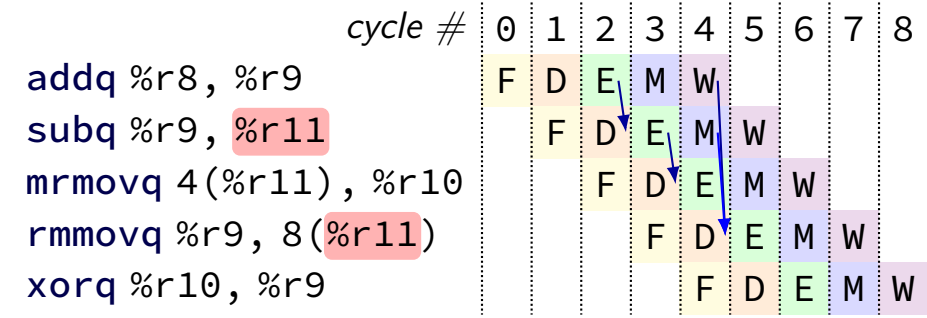
some forwarding paths



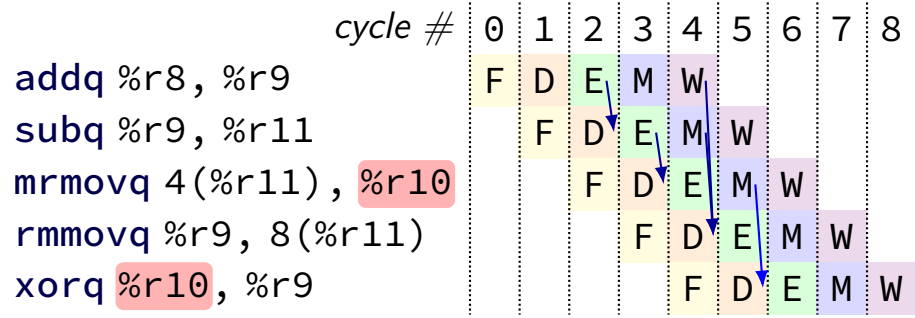
some forwarding paths



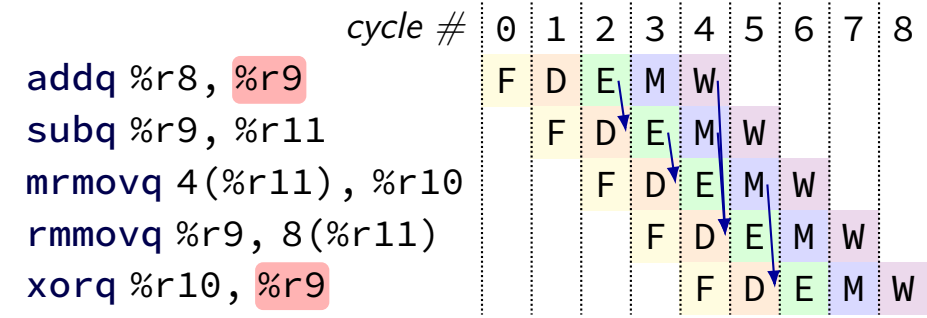
some forwarding paths



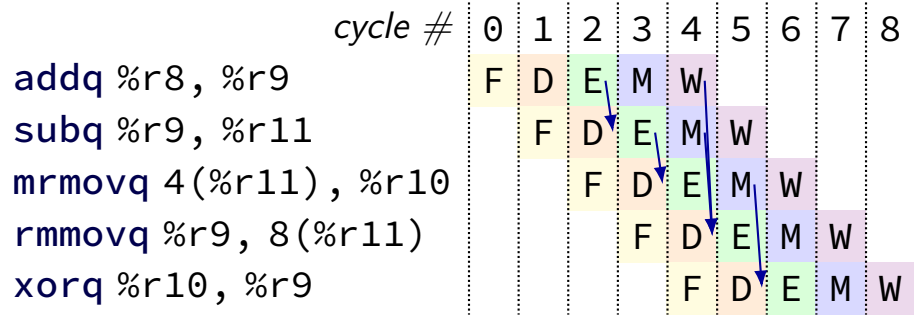
some forwarding paths



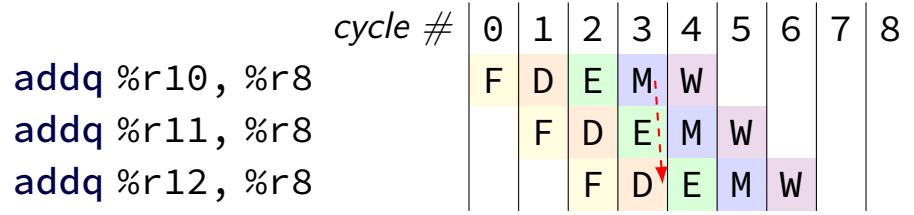
some forwarding paths



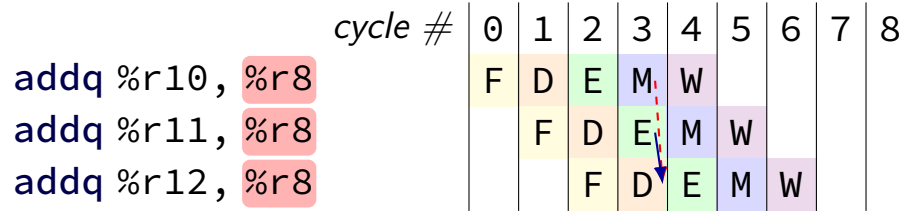
some forwarding paths



multiple forwarding paths (1)



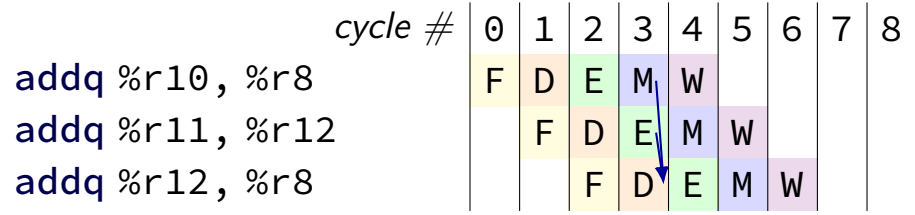
multiple forwarding paths (1)



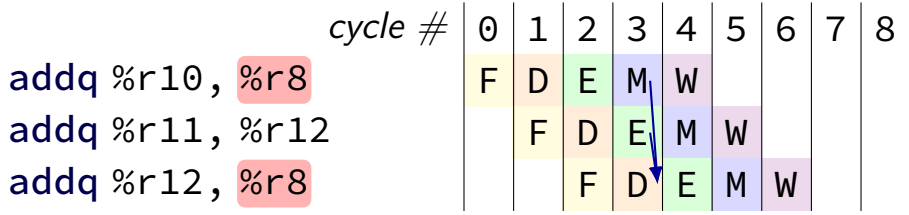
multiple forwarding HCL (1)

```
/* decode output: valA */
d_valA = [
    ...
    reg_srcA == e_dstE : e_valE;
        /* forward from end of execute */

    reg_srcA == m_dstE : m_valE;
        /* forward from end of memory */

    ...
    1 : reg_outputA;
];
```

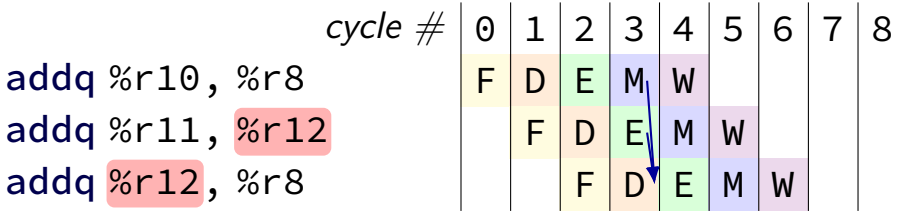
multiple forwarding paths (2)



multiple forwarding paths (2)



multiple forwarding paths (2)



multiple forwarding HCL (2)

```
d_valA = [
    ...
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
...
d_valB = [
    ...
    reg_srcB == m_dstE : m_valE;
    ...
    1 : reg_outputB;
];
```

exercise: forwarding paths

	cycle # 0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r8, %r10`		F	D	E	M	W			
`xorq %r8, %r9`			F	D	E	M	W		
`andq %r9, %r8`				F	D	E	M	W	

in subq, %r8 is _____ addq.

in xorq, %r9 is _____ addq.

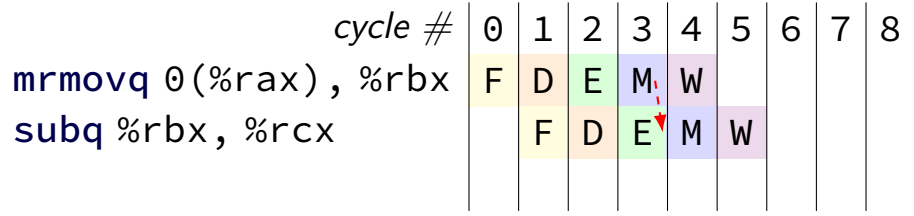
in andq, %r9 is _____ addq.

in andq, %r9 is _____ xorq.

A: not forwarded from

B-D: forwarded to decode from {execute,memory,writeback} stage of

unsolved problem



combine stalling and forwarding to resolve hazard

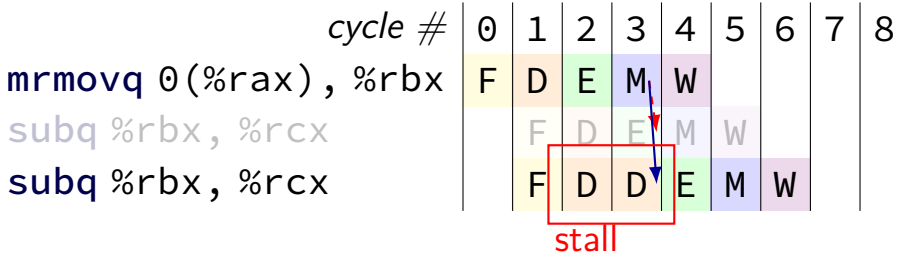
assumption in diagram: hazard detected in subq's decode stage

(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

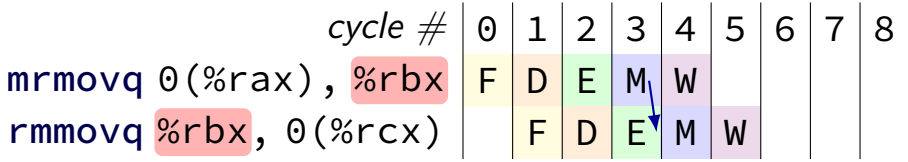
assumption in diagram: hazard detected in `subq`'s decode stage

(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

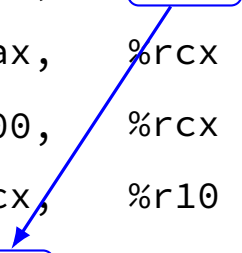
ex.: dependencies and hazards (1)

```
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)

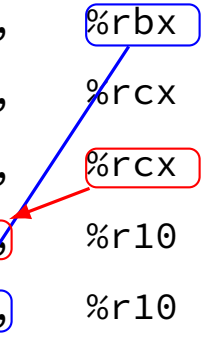
```
addq    %rax,   %rbx
subq    %rax,   %rcx
irmovq  $100,   %rcx
addq    %rcx,   %r10
addq    %rbx,   %r10
```



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

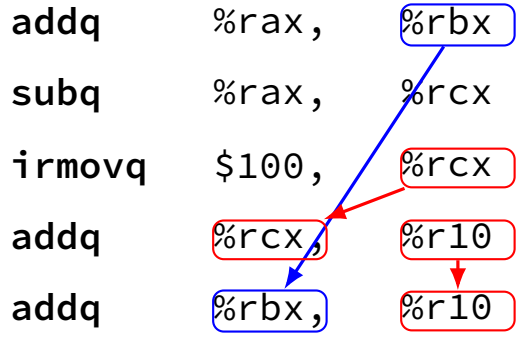
ex.: dependencies and hazards (1)

```
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)



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

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`									

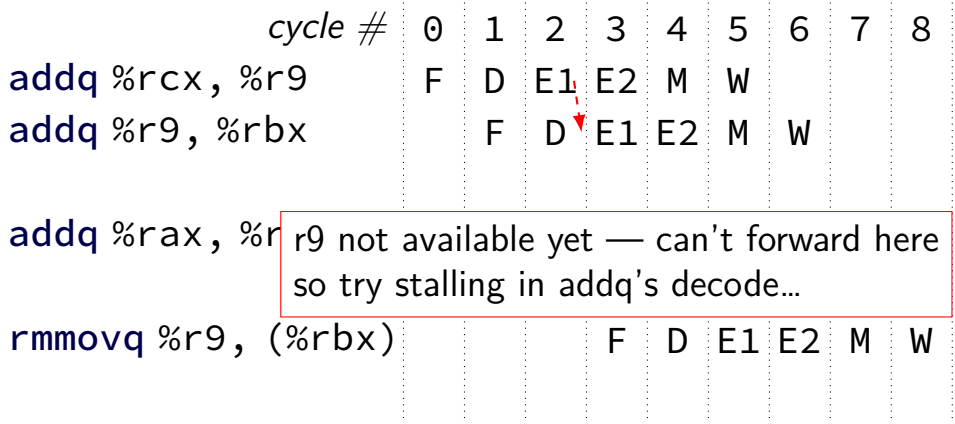
exercise: different pipeline

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

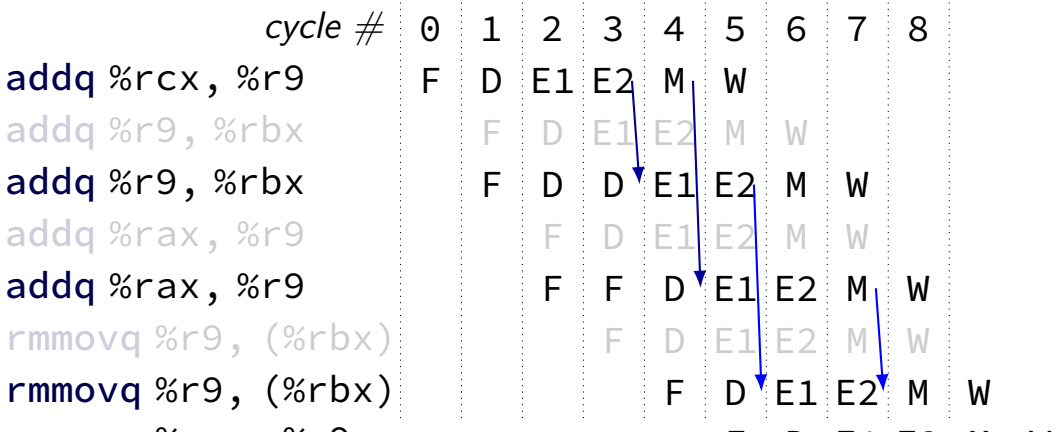
exercise: different pipeline

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/Z							
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

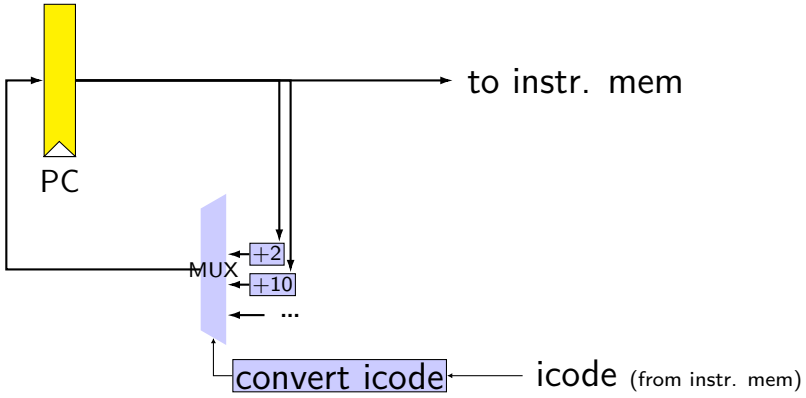
```
addq %r8, %r9
// insert two nops
je   0xFFFF
addq %r10, %r11
```

	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*								
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

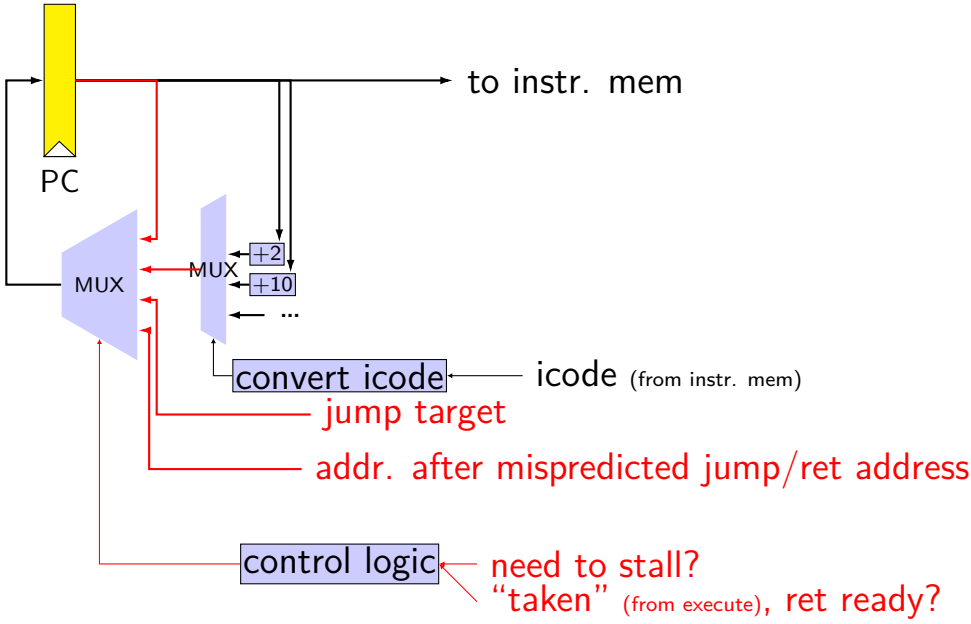
execute je instruction (use SF/ZF)

backup slides

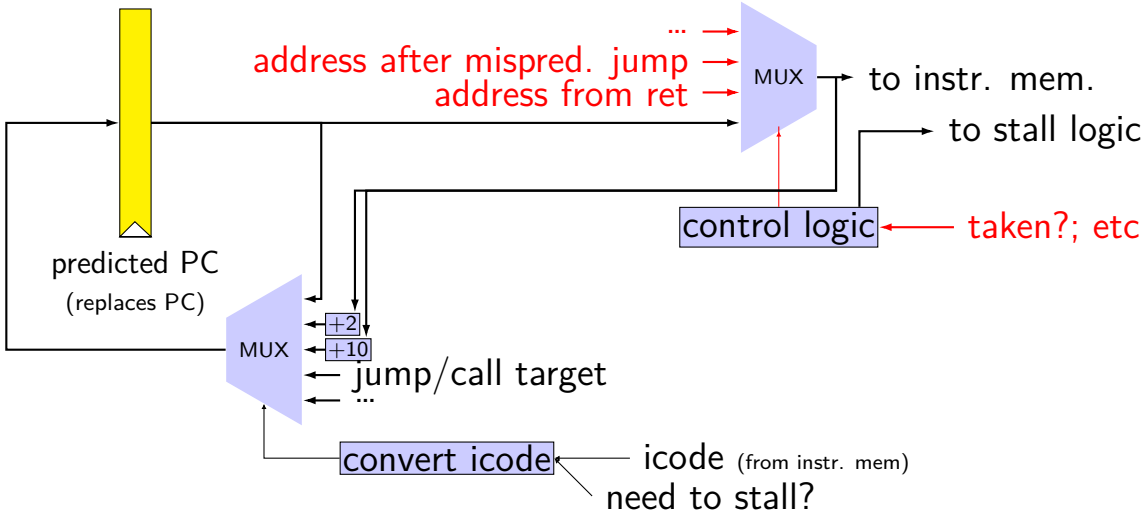
PC update (adding prediction, stall)



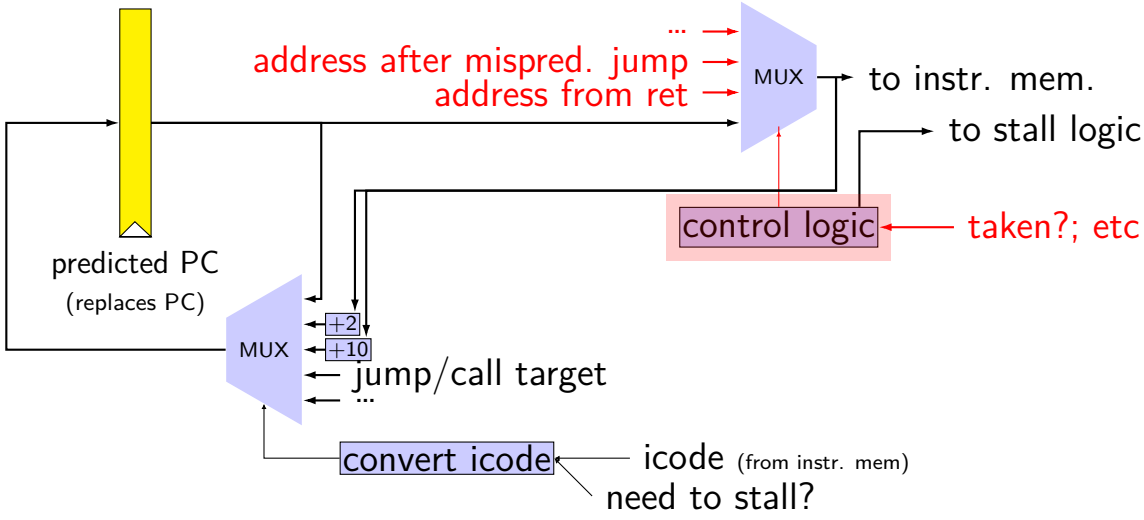
PC update (adding prediction, stall)



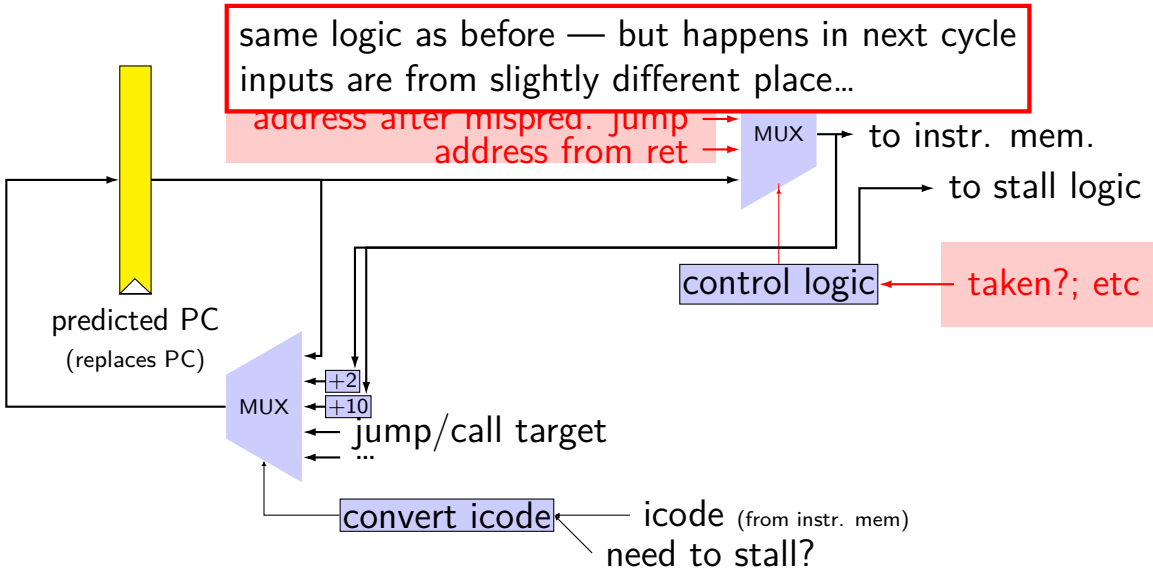
PC update (rearranged)



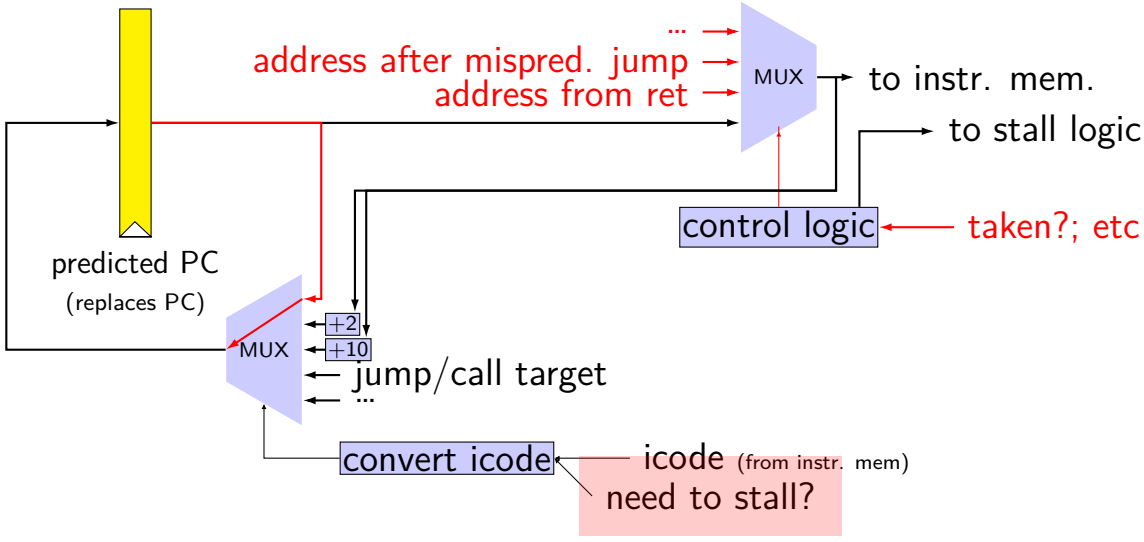
PC update (rearranged)



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

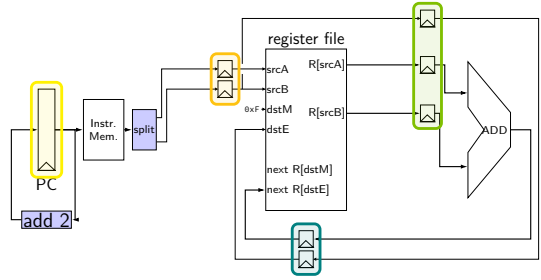
- correct PC at beginning or end of cycle?
- still some time in cycle to do so...

maybe easier to think about branch prediction this way?

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

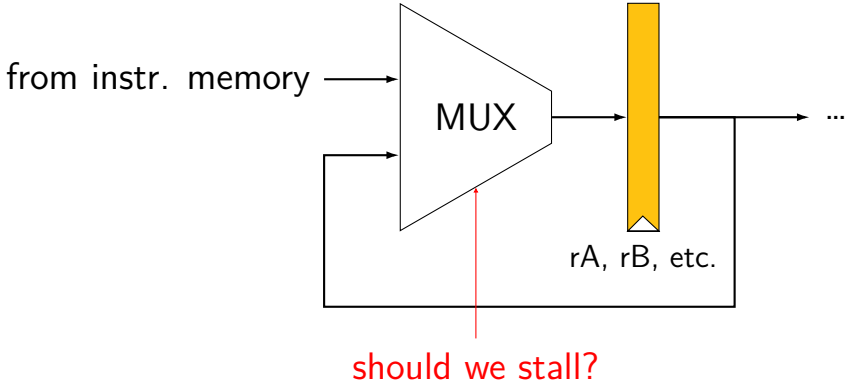
add up everything but add 2 (critical (slowest) path)

pipelining: 1 instruction per 200 ps + pipeline register delays

slowest path through stage + pipeline register delays

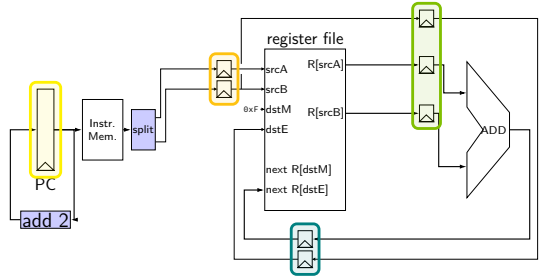
latency: 800 ps + pipeline register delays (4 cycles)

fetch/decode logic — advance or not



exercise

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



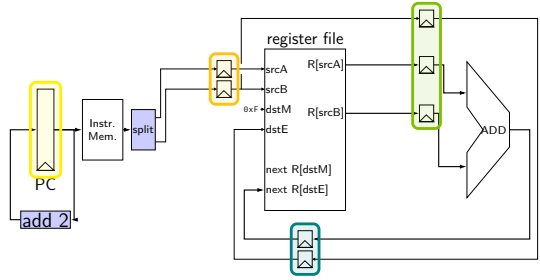
pipeline register delay: 10ps

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



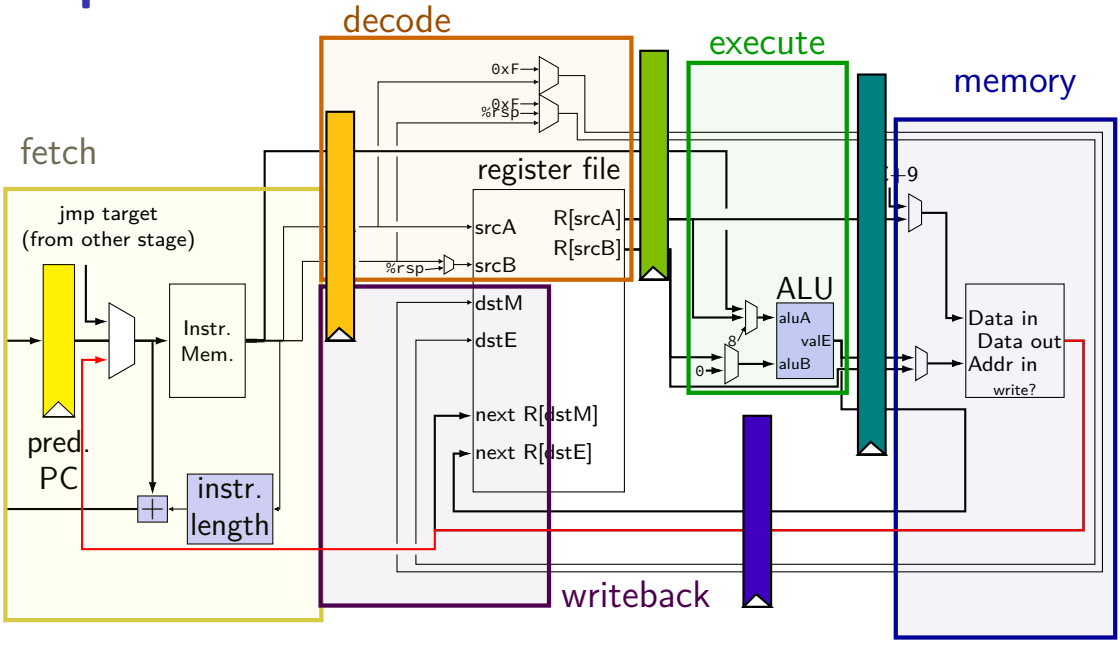
pipeline register delay: 10ps

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

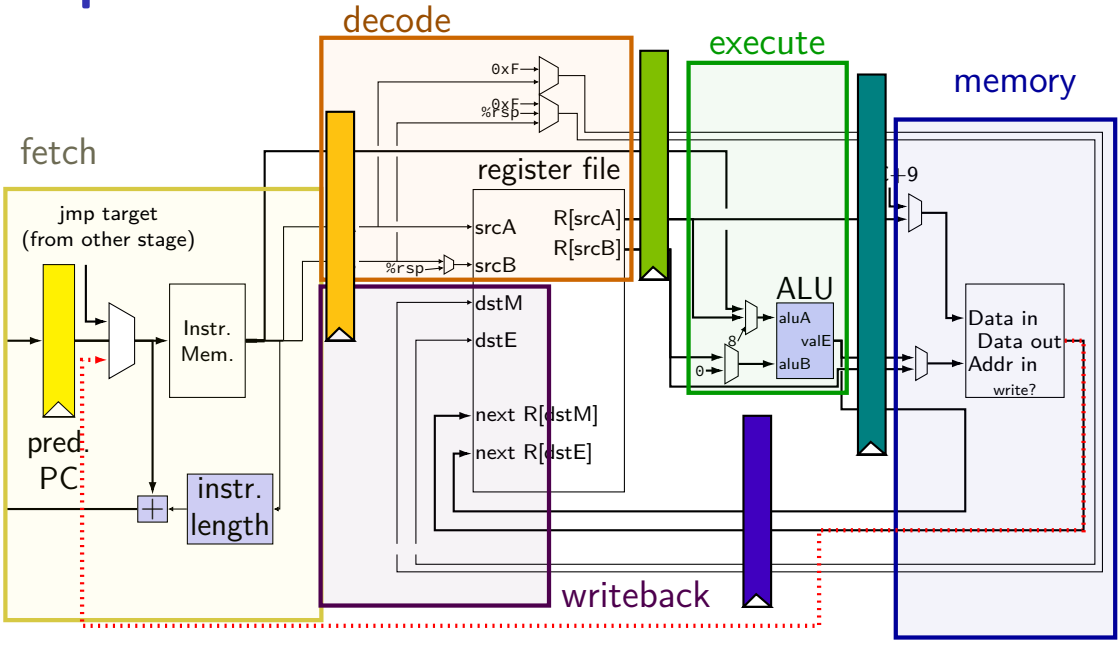
A. 2.00x **B.** 1.70x to 1.99x
C. 1.60x to 1.69x **D.** 1.50x to 1.59x
E. less than 1.50x

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

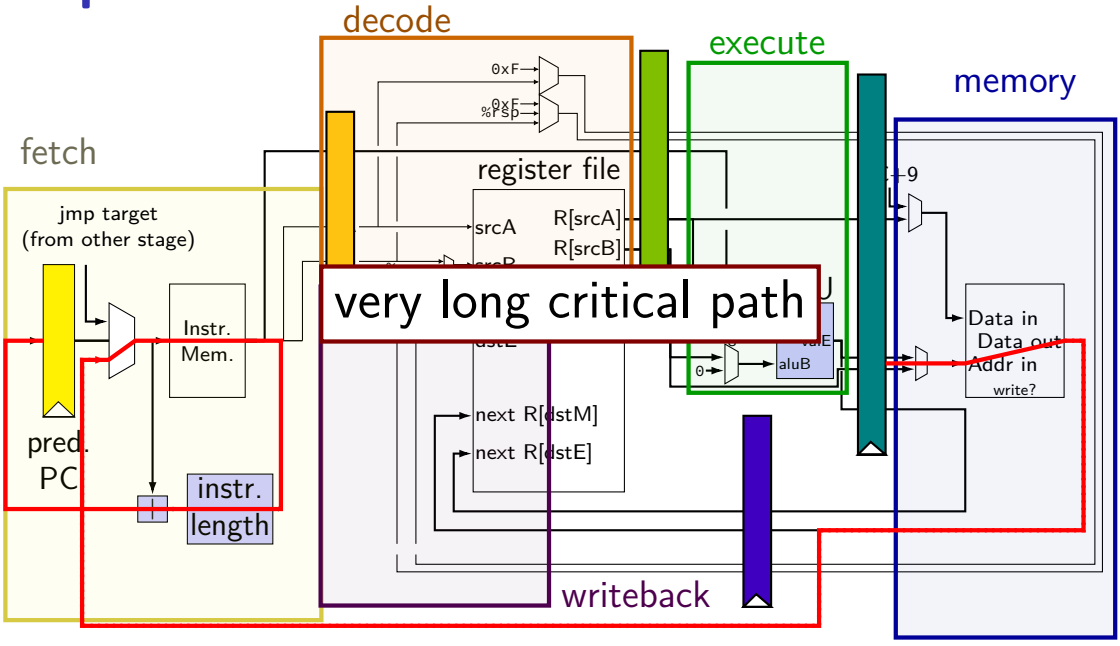
ret paths


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

revisiting data hazards

stalling worked

but very unsatisfying — wait 2 extra cycles to use anything?!

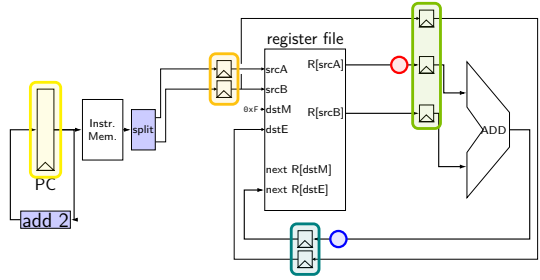
...or more with 5-stage pipeline

observation: value ready before it would be needed

(just not stored in a way that let's us get it)

motivation

location of values during cycle 2:



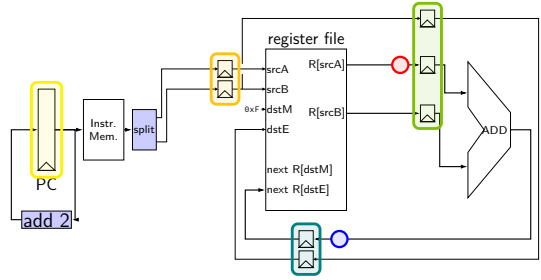

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	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

location of values during cycle 2:




```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	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