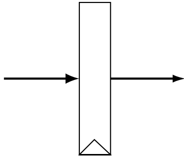
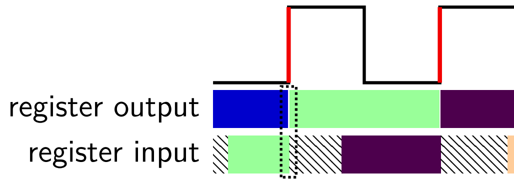
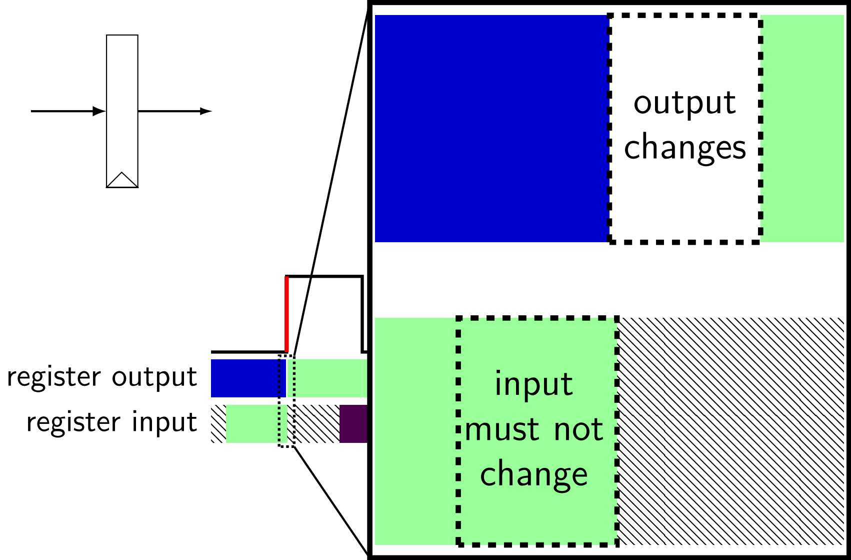
last time

register tolerances

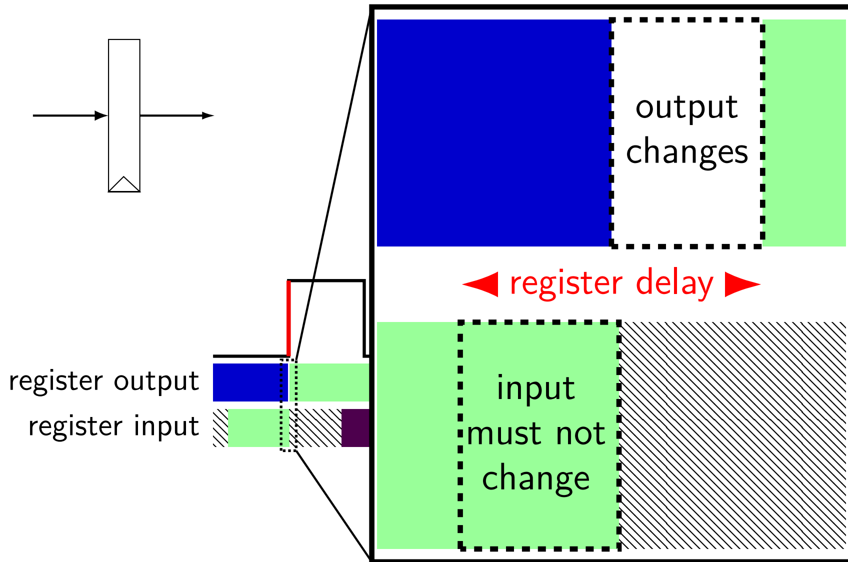




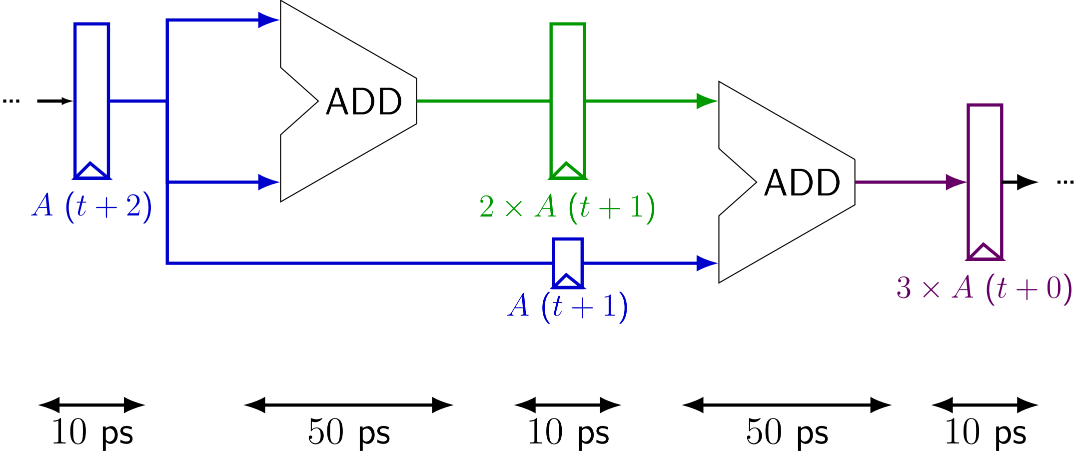
register tolerances



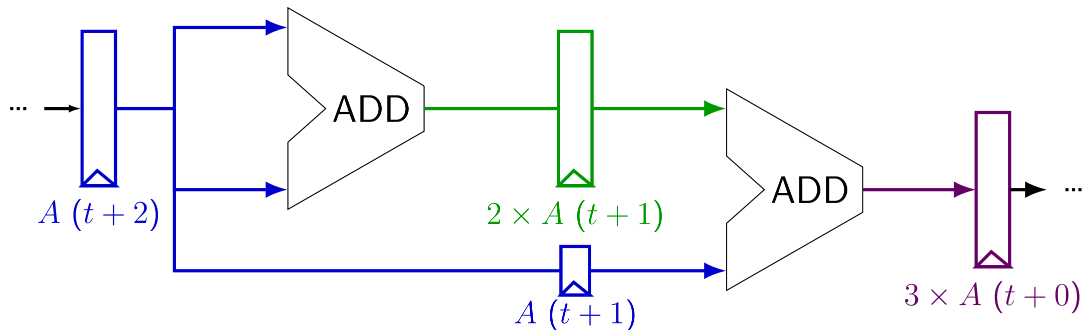
register tolerances



times three pipeline timing



times three pipeline timing

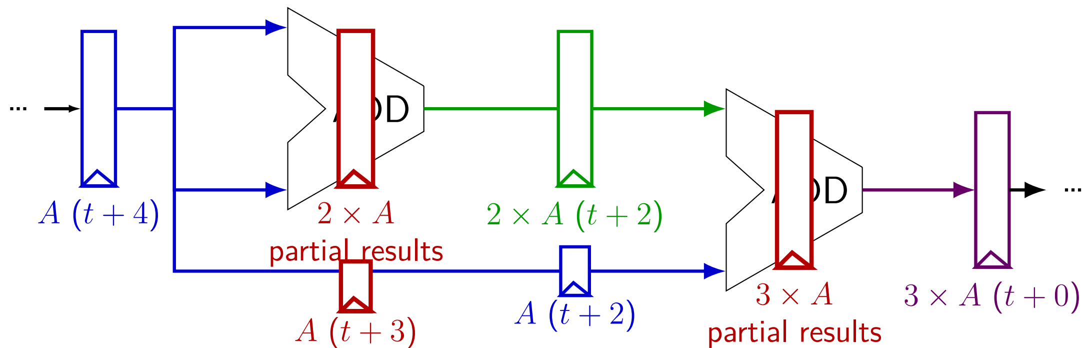


10 ps 50 ps 10 ps 50 ps 10 ps

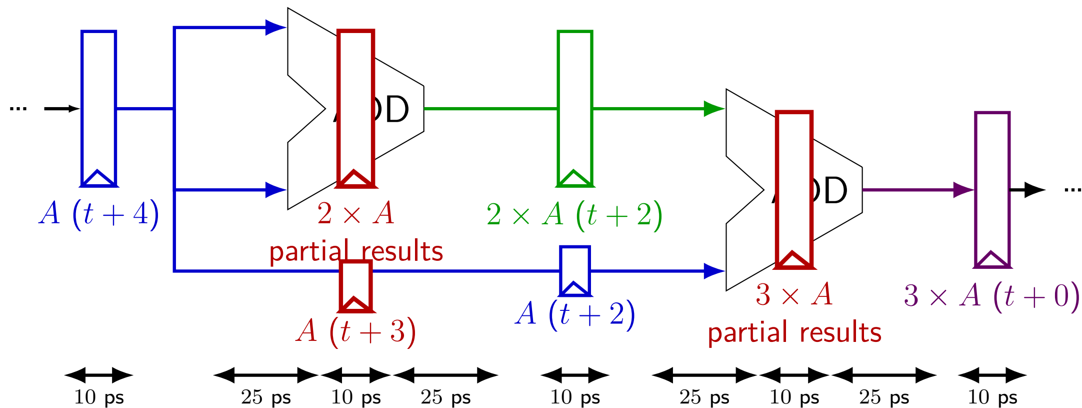
exercise: minimum clock cycle time:

A. 50 ps B. 60 ps C. 65 ps D. 70 ps E. 130 ps

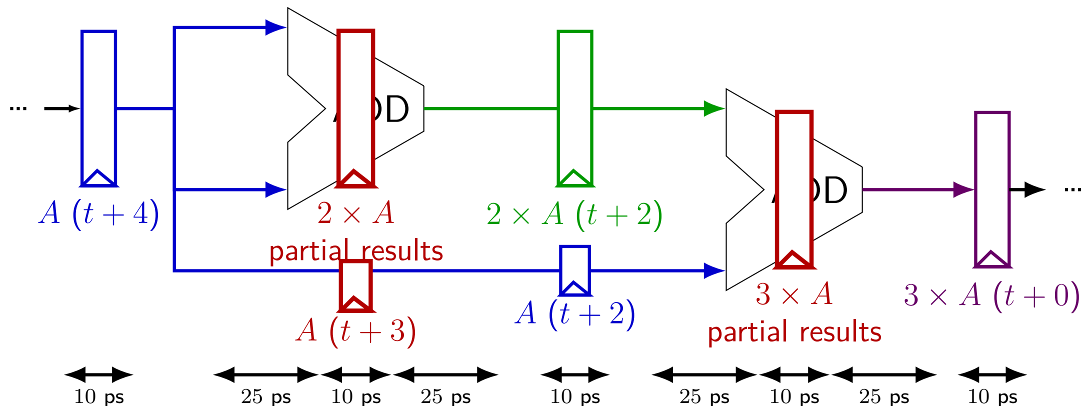
deeper pipeline



deeper pipeline



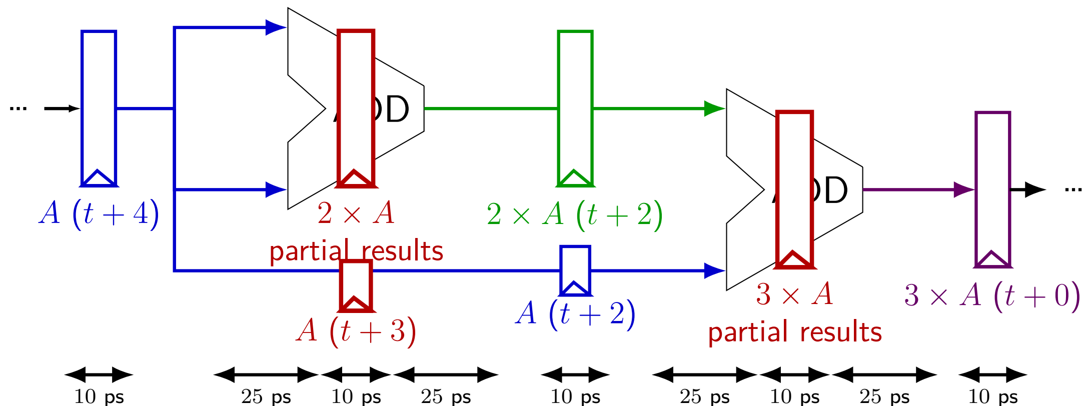
deeper pipeline



Problem: How much faster can we get?

Problem: Can we even do this?

deeper pipeline

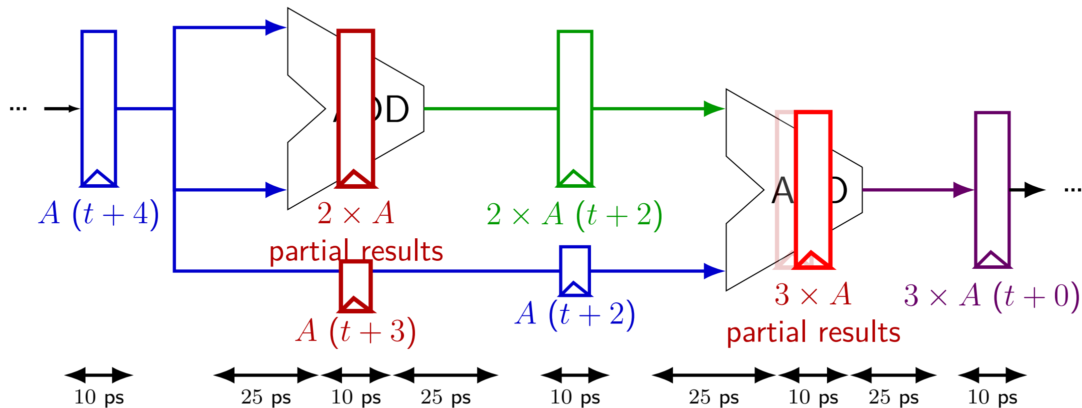


exercise: throughput now?

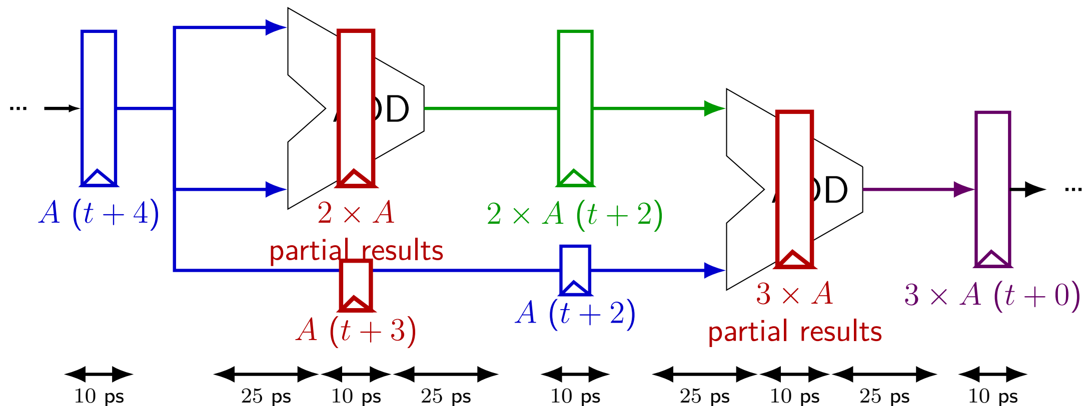
A. 1/(25 ps) B. 1/(30 ps)

C. 1/(35 ps) D. something else

deeper pipeline



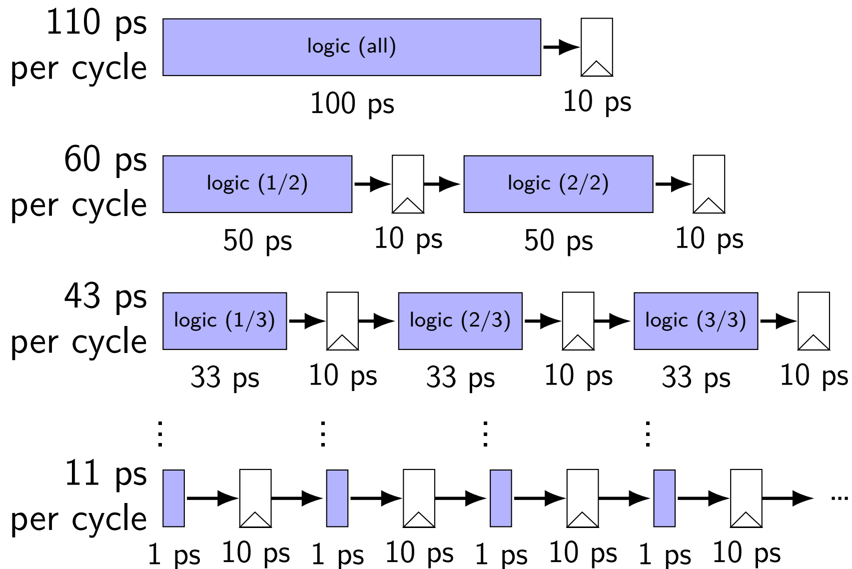
deeper pipeline



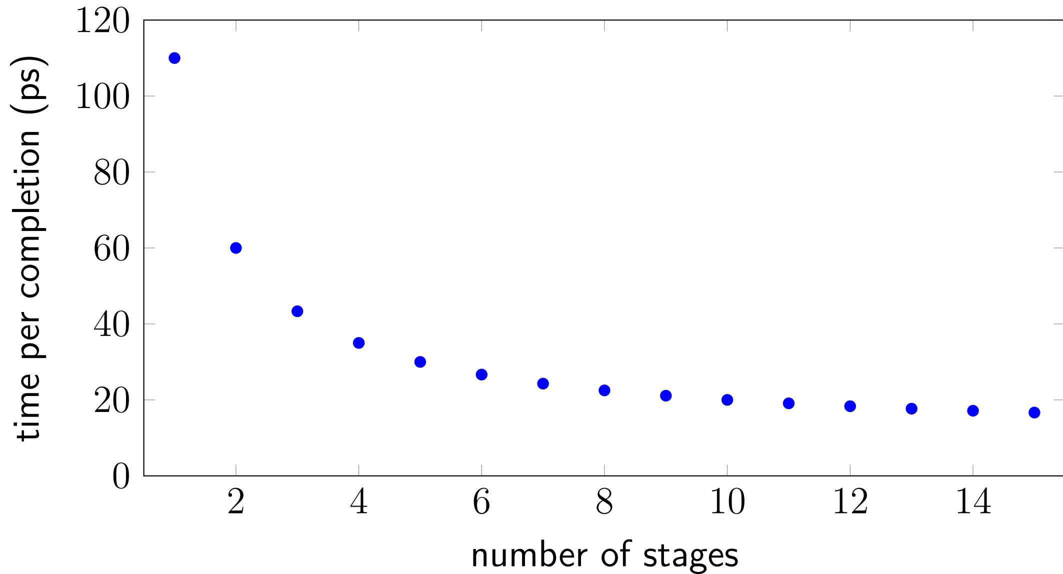
Problem: How much faster can we get?

Problem: Can we even do this?

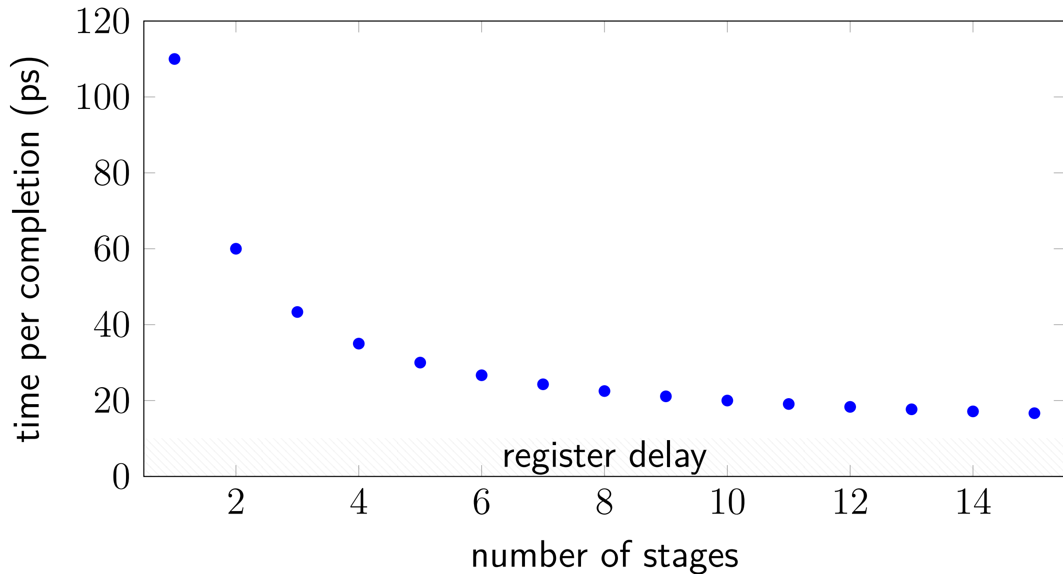
diminishing returns: register delays



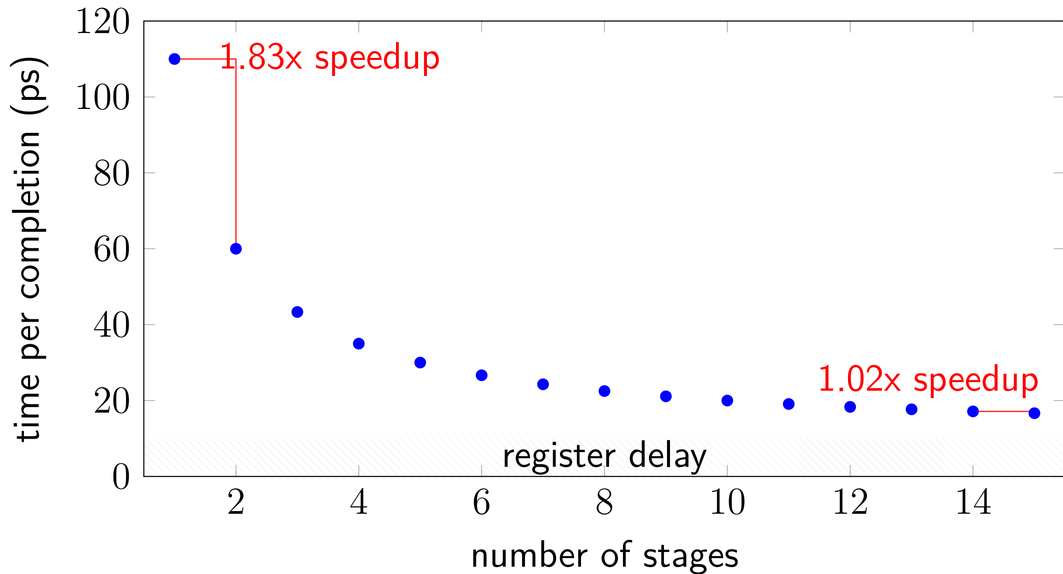
diminishing returns: register delays



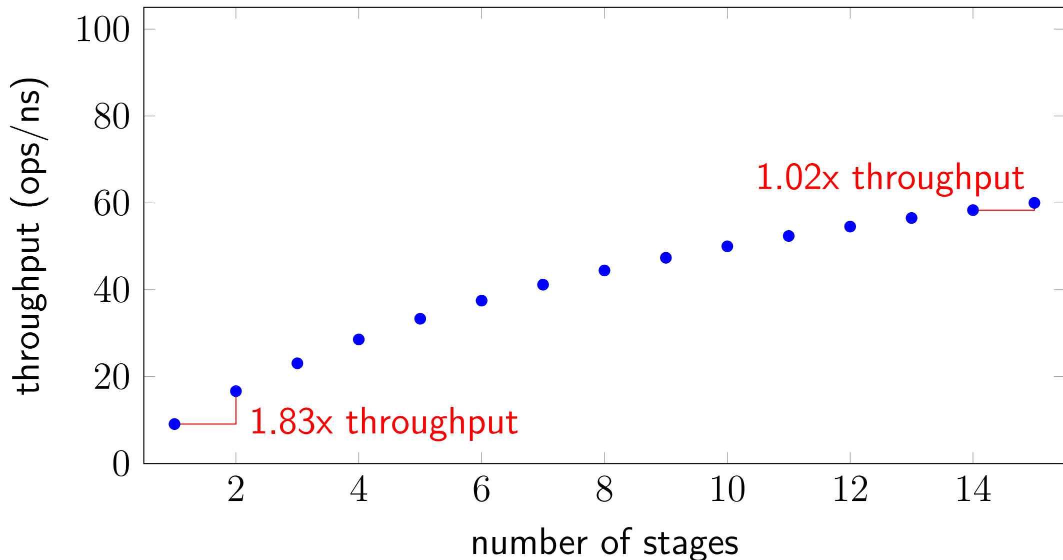
diminishing returns: register delays



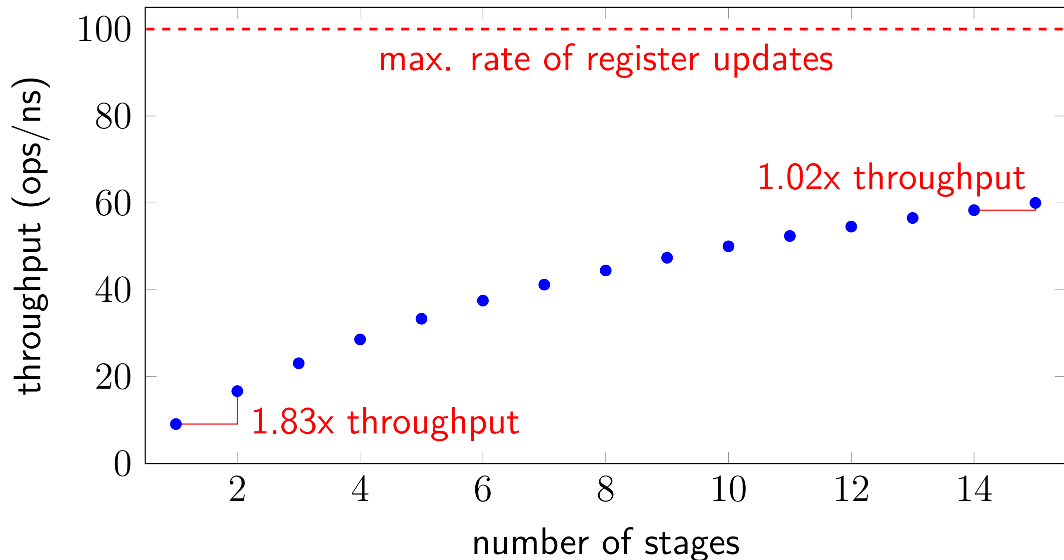
diminishing returns: register delays



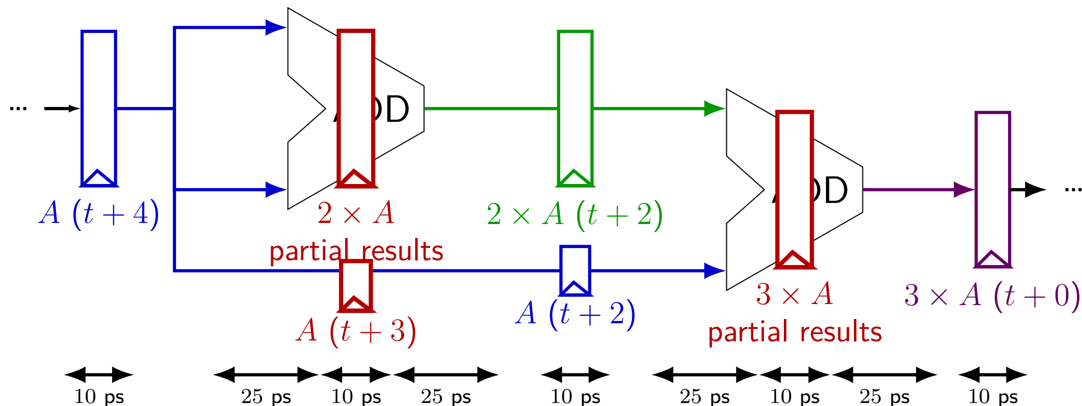
diminishing returns: register delays



diminishing returns: register delays



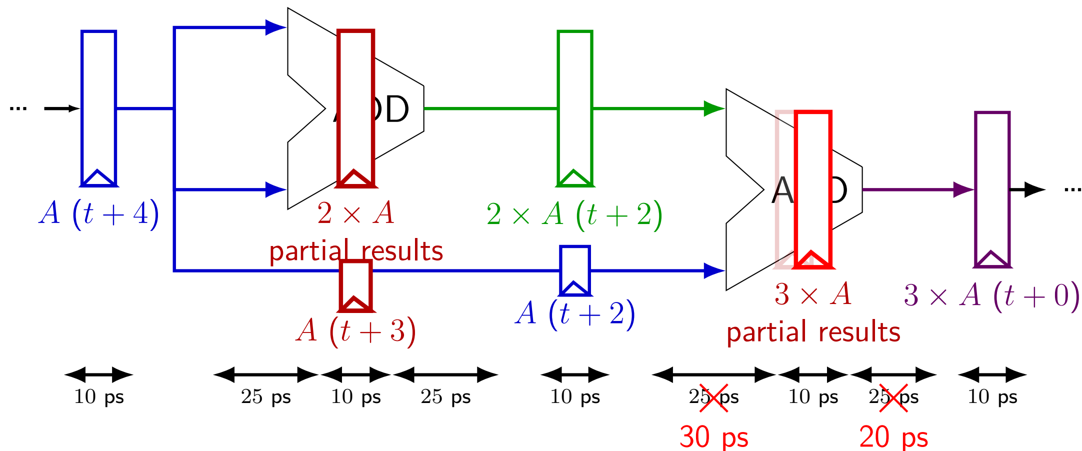
deeper pipeline



Problem: How much faster can we get?

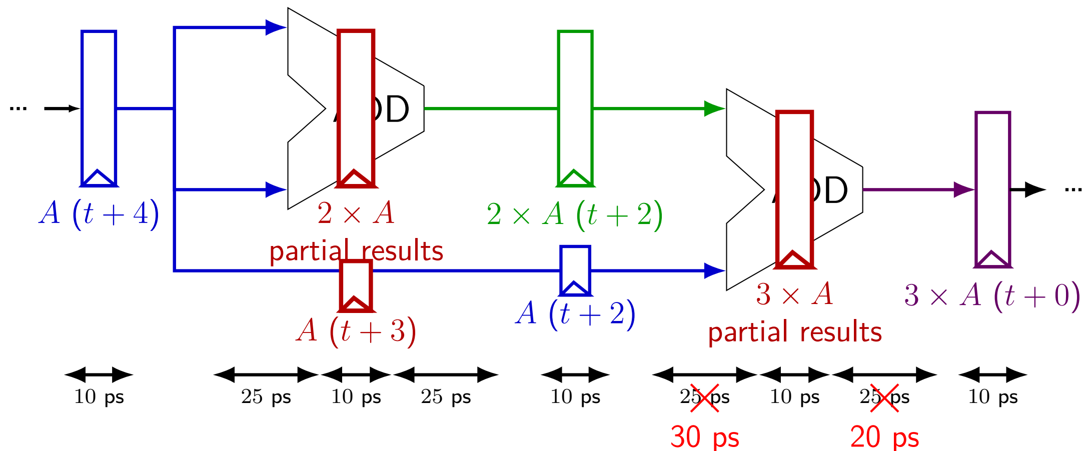
Problem: Can we even do this?

deeper pipeline



exercise: throughput now? (didn't split second add evenly)

deeper pipeline

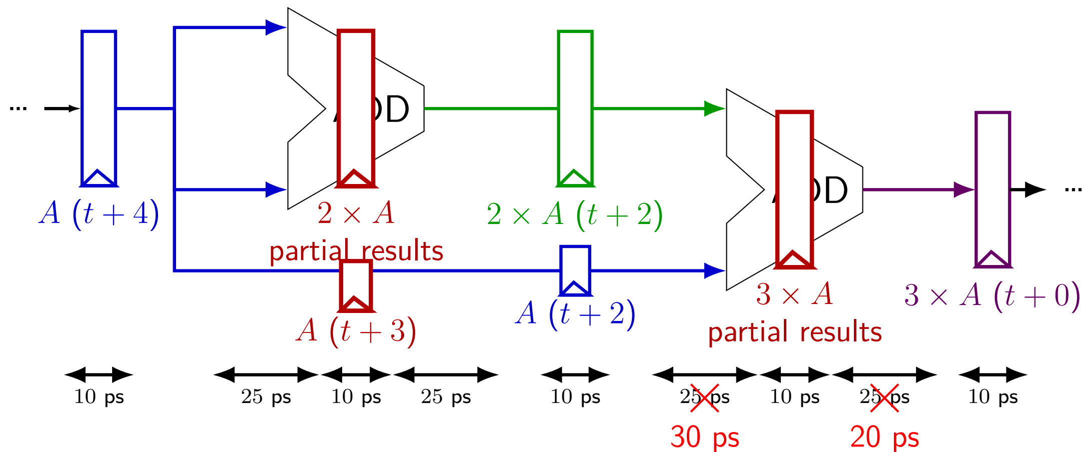


exercise: throughput now? (didn't split second add evenly)

A. 1/(25 ps) B. 1/(30 ps)

C. 1/(35 ps) D. 1/(40 ps) E. something else

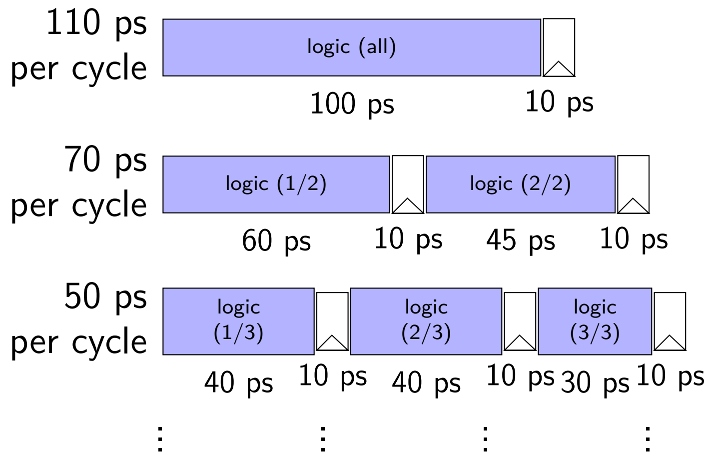
deeper pipeline



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

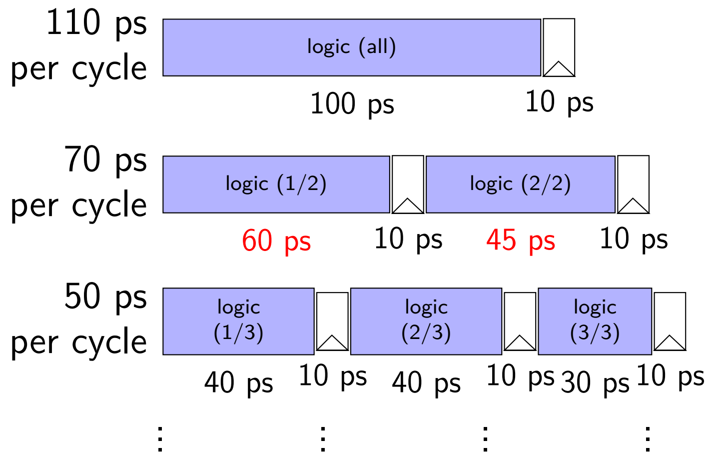
Probably not...



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

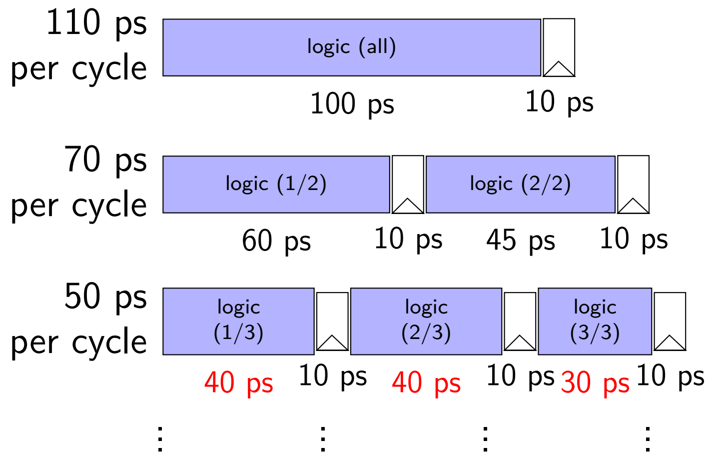
Probably not...



diminishing returns: uneven split

Can we split up some logic (e.g. adder) arbitrarily?

Probably not...



textbook SEQ ‘stages’

conceptual order only

Fetch: read instruction memory

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

PC Update: write PC register

textbook SEQ ‘stages’

conceptual order only

Fetch: read instruction memory

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

PC Update: write PC register

writes happen at end of cycle

textbook SEQ 'stages'

conceptual order only

Fetch: read instruction memory

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

PC Update: write PC register

reads — "magic"
like combinatorial logic
as values available

textbook stages

~~conceptual order only~~ pipeline stages

Fetch/PC Update: read instruction memory;
compute next PC

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

textbook stages

~~conceptual order only~~ pipeline stages

Fetch/PC Update: read instruction memory;
compute next PC

Decode: read register file

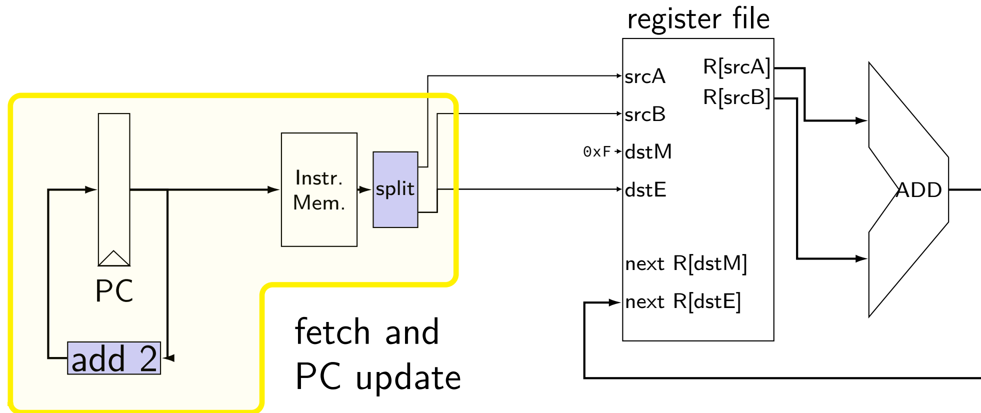
Execute: arithmetic (ALU)

Memory: read/write data memory

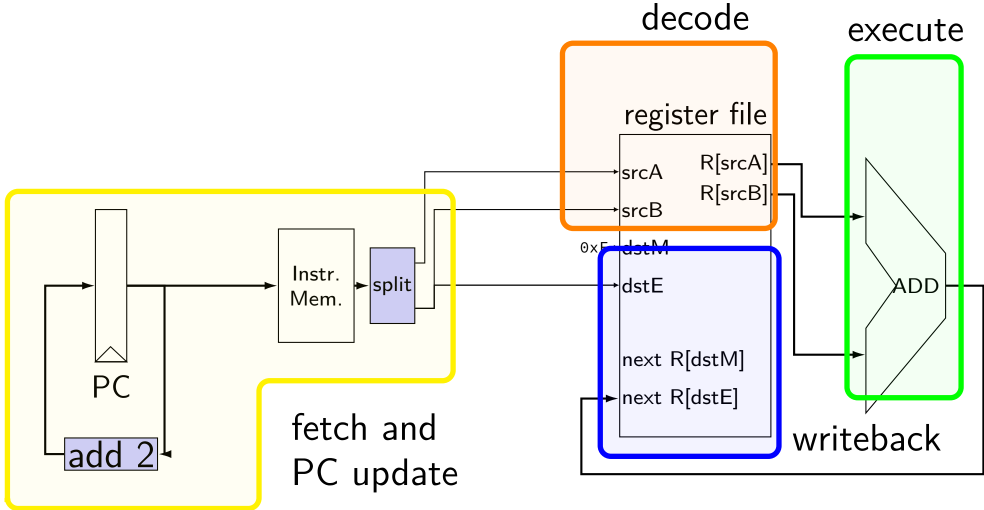
Writeback: write register file

5 stages
one instruction in each
compute next to start immediatelly

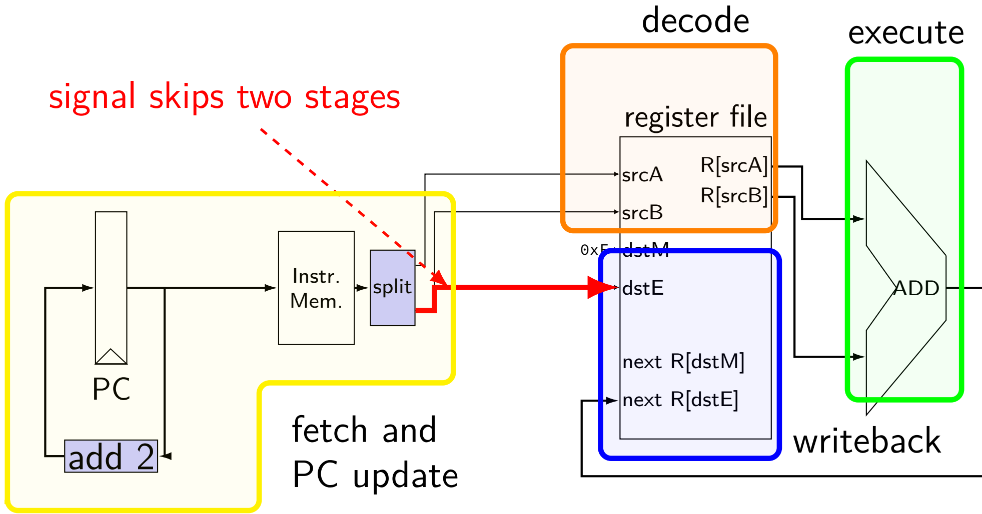
addq CPU



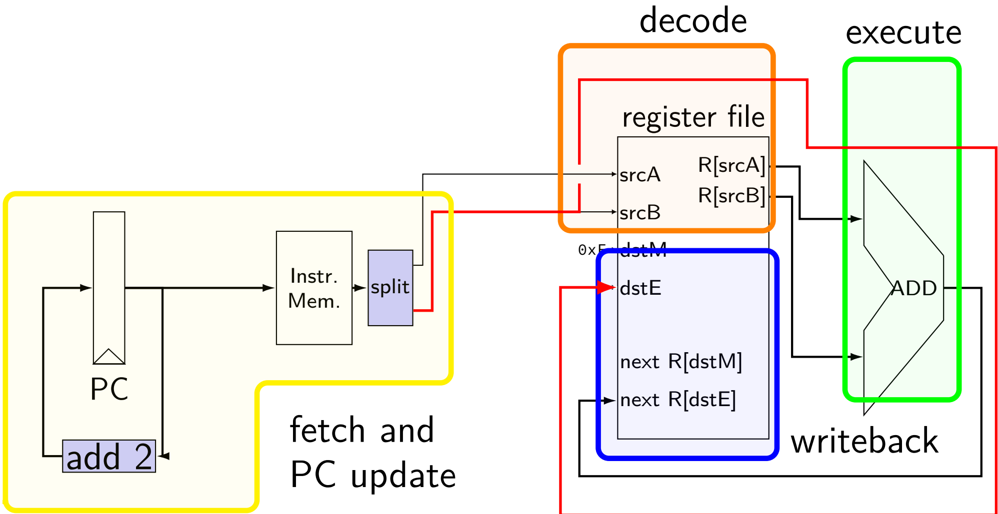
addq CPU



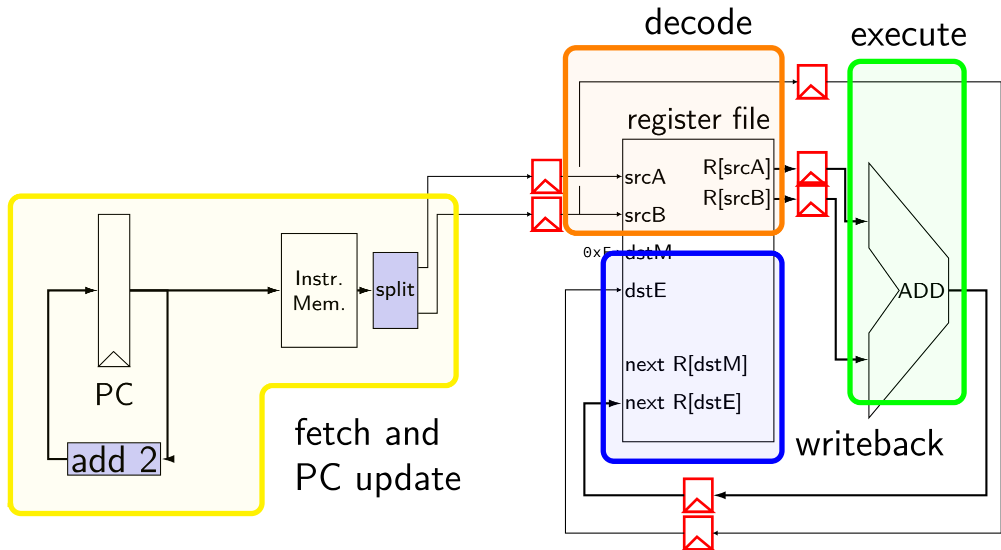
addq CPU



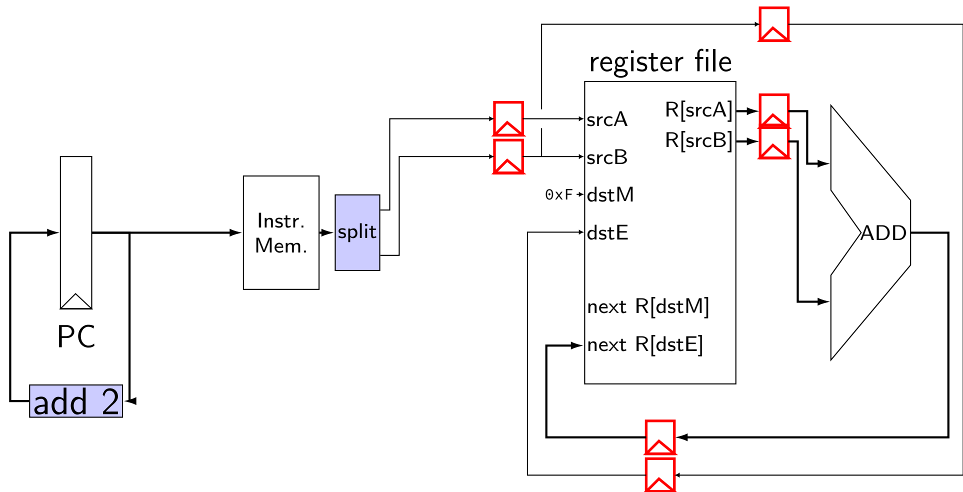
addq CPU



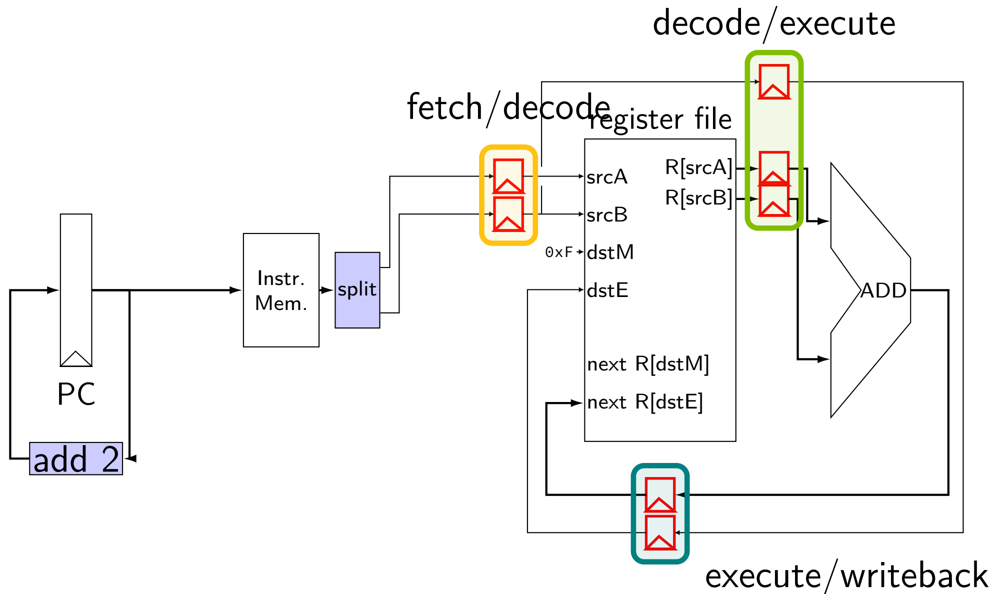
pipelined addq processor



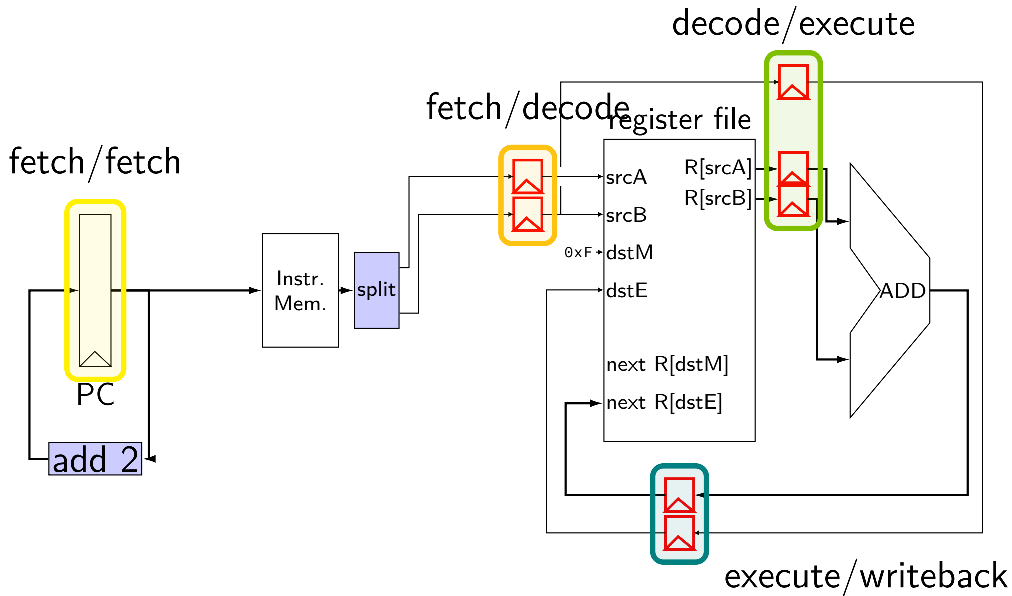
pipelined addq processor



pipelined addq processor

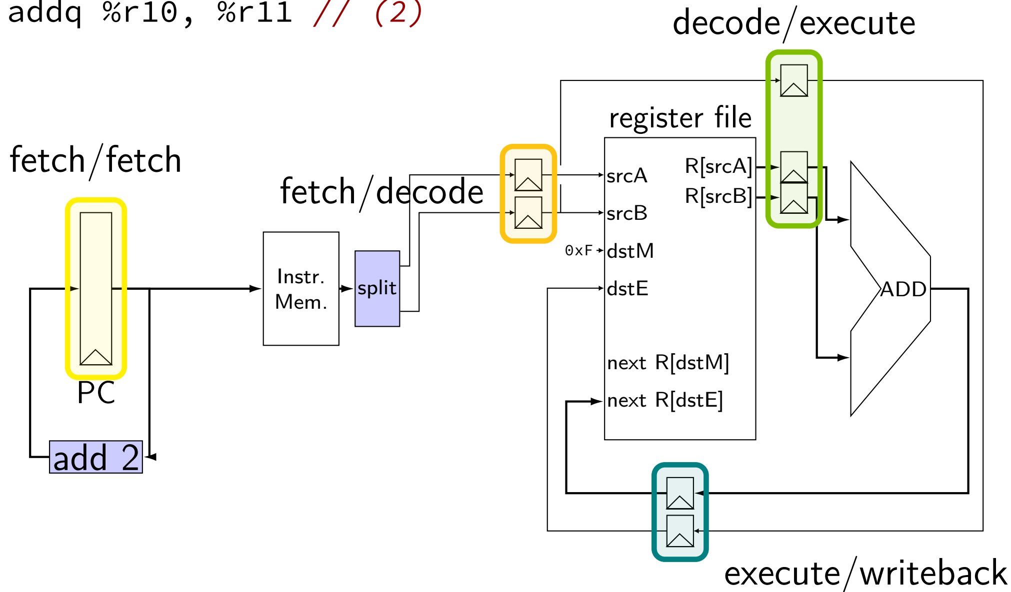


pipelined addq processor



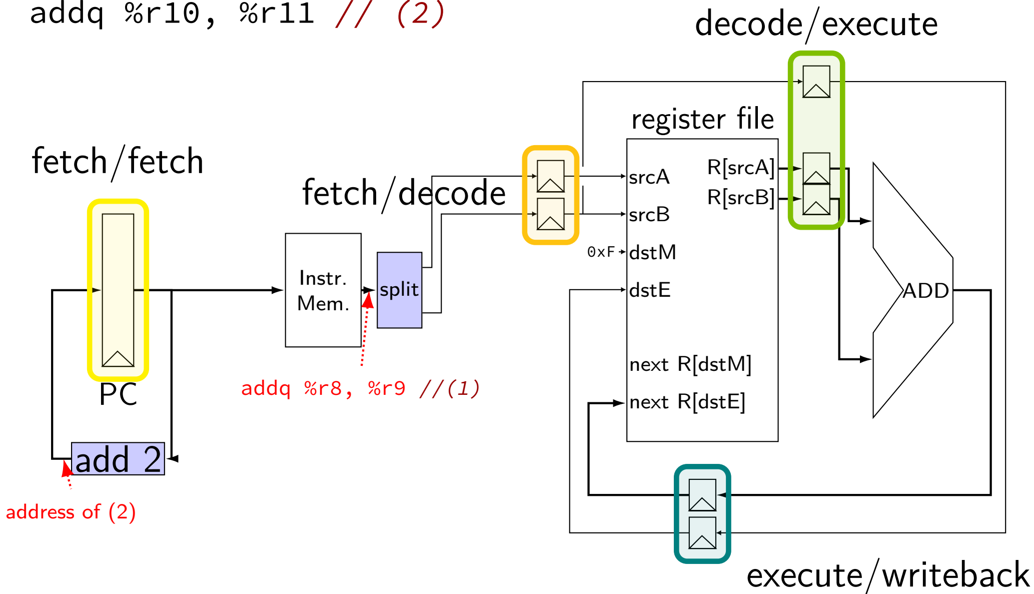
addq execution

```
addq %r8, %r9   // (1)
addq %r10, %r11 // (2)
```



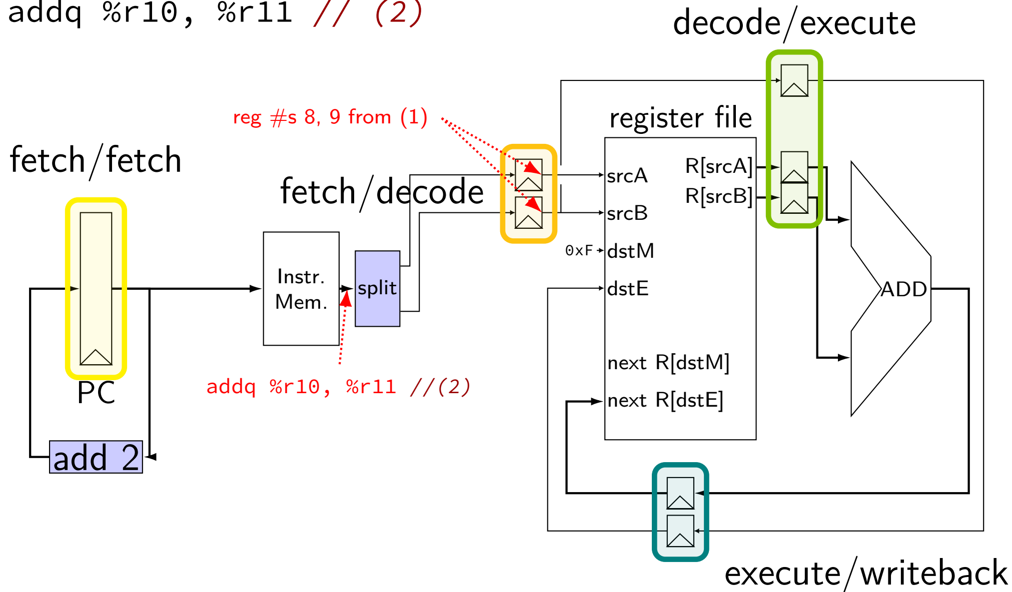
addq execution

```
addq %r8, %r9  // (1)
addq %r10, %r11 // (2)
```



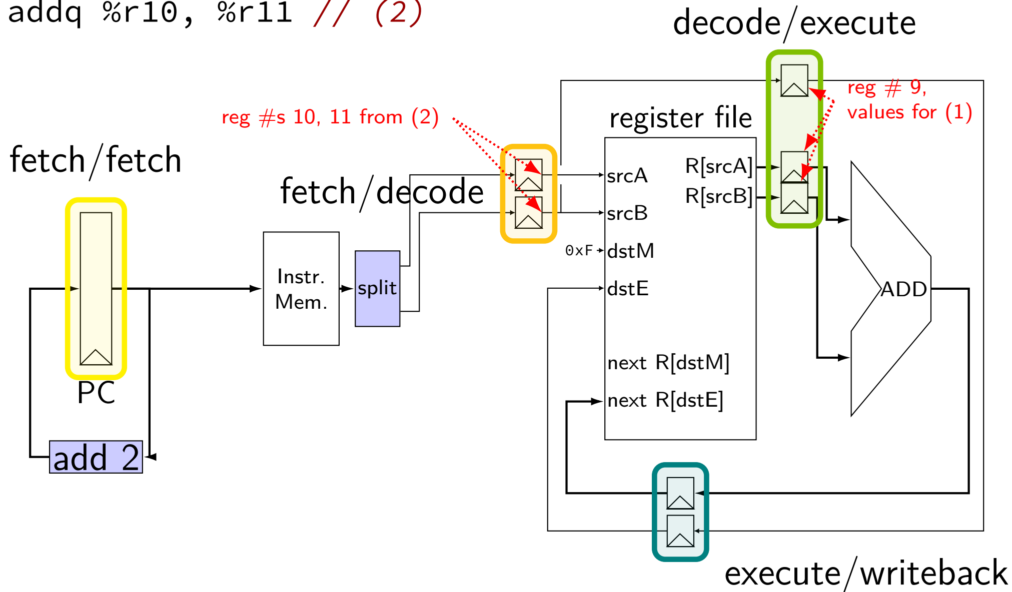
addq execution

```
addq %r8, %r9   // (1)
addq %r10, %r11 // (2)
```



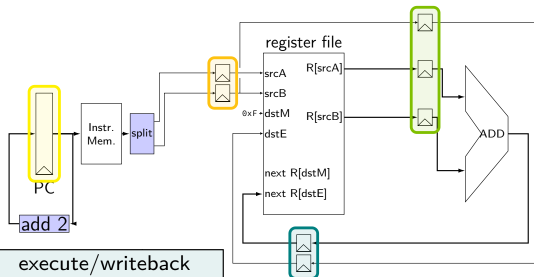
addq execution

```
addq %r8, %r9  // (1)
addq %r10, %r11 // (2)
```



addq processor timing

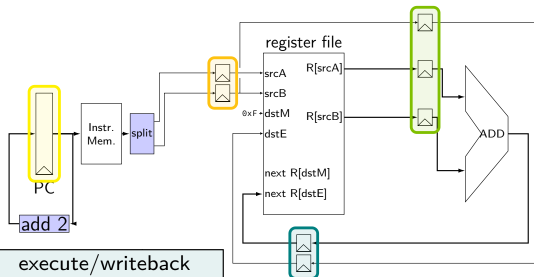
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor timing

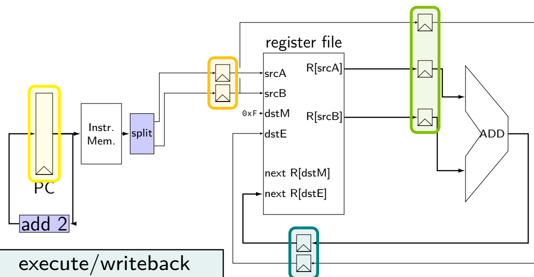
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor timing

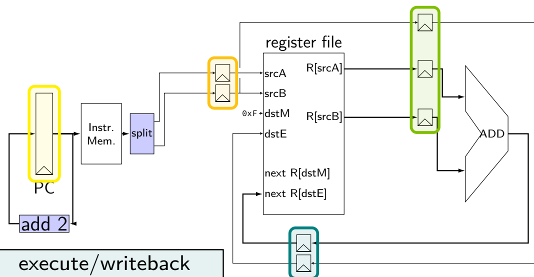
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor timing

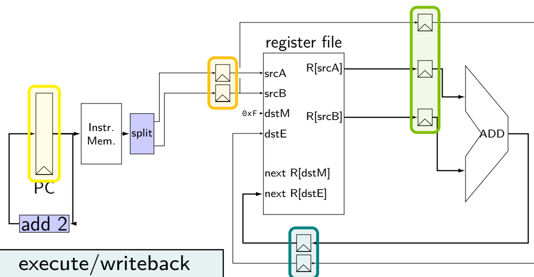
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor timing

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

critical path

every path from state output to state input needs enough time

output — may change on rising edge of clock

input — must be stable sufficiently before rising edge of clock

critical path: slowest of all these paths — determines cycle time

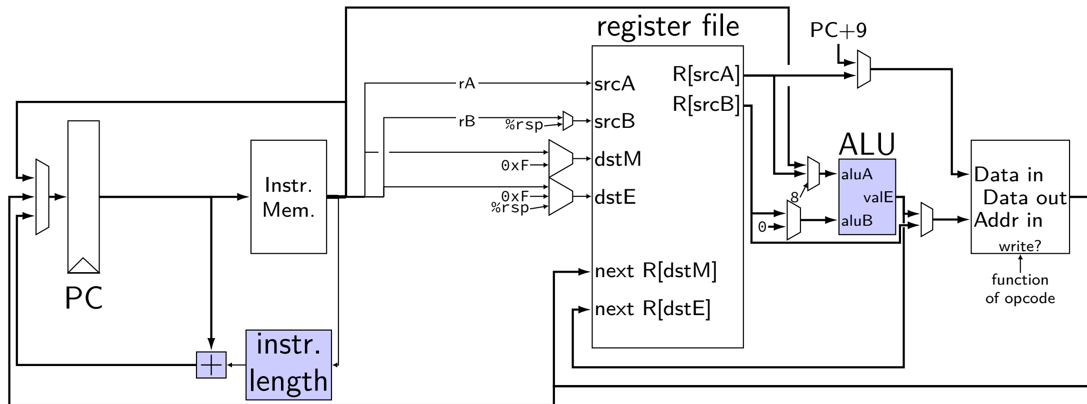
times three: slowest stage ended up mattering

have to choose *one* clock cycle length

can't vary clock depending on what instruction is running

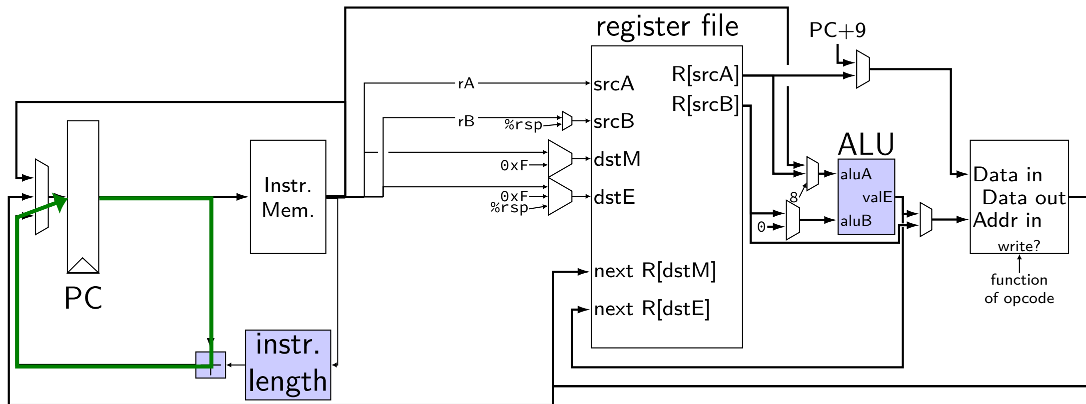
matters with or without pipelining

SEQ paths



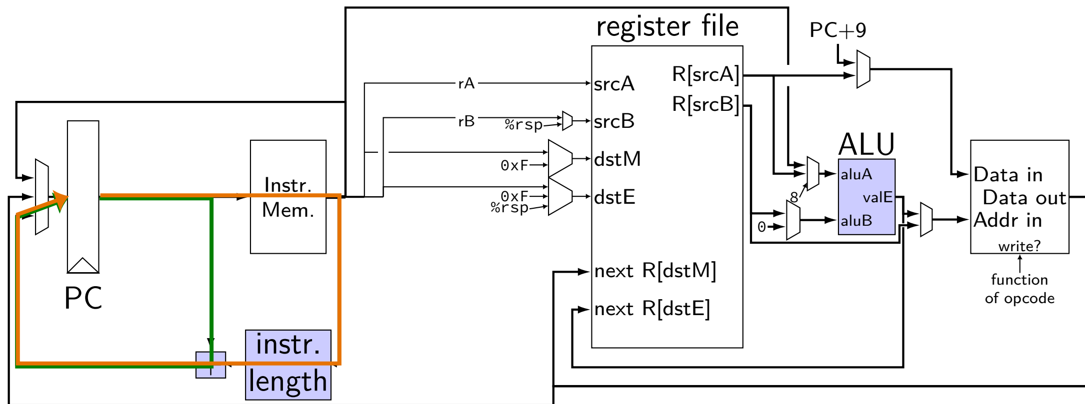
SEQ paths

path 1: 25 picoseconds



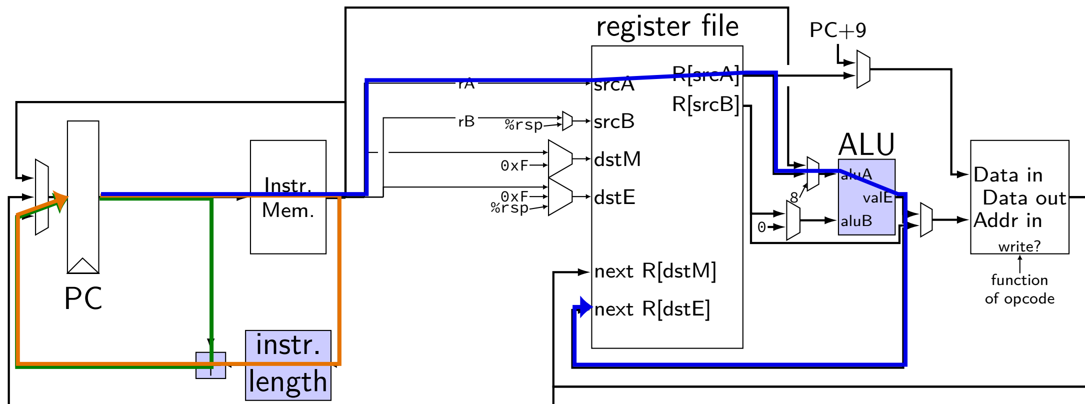
SEQ paths

path 1: 25 picoseconds   path 2: 50 picoseconds



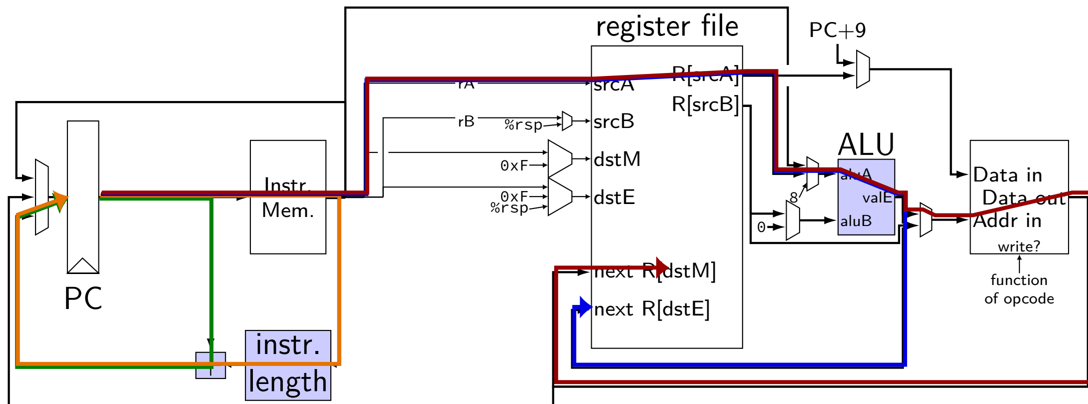
SEQ paths

path 1: 25 picoseconds
path 2: 50 picoseconds
path 3: 400 picoseconds



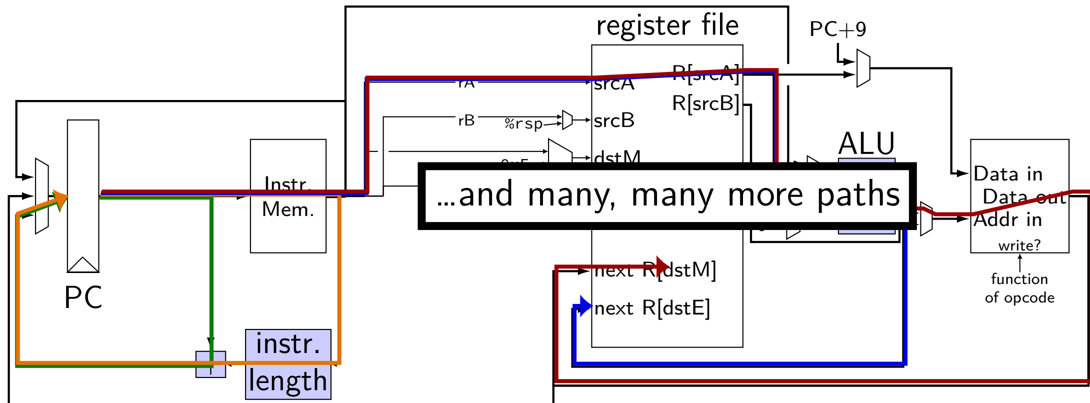
SEQ paths

path 1: 25 picoseconds
path 2: 50 picoseconds
path 3: 400 picoseconds
path 4: 900 picoseconds
...
...

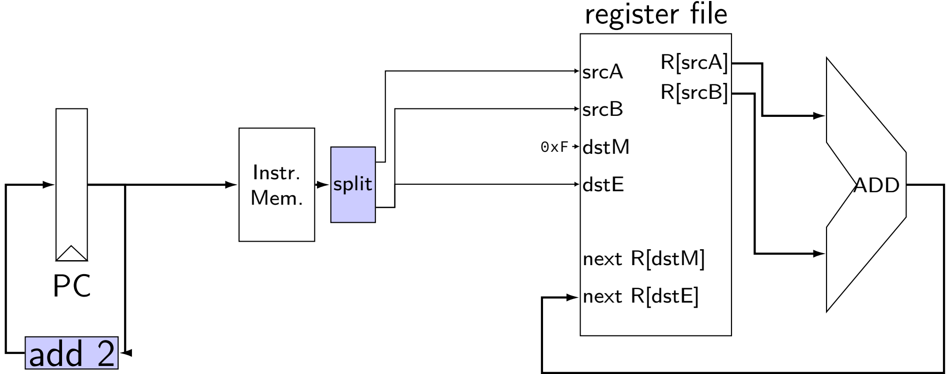


SEQ paths

path 1: 25 picoseconds
path 2: 50 picoseconds
path 3: 400 picoseconds
path 4: 900 picoseconds
...
...

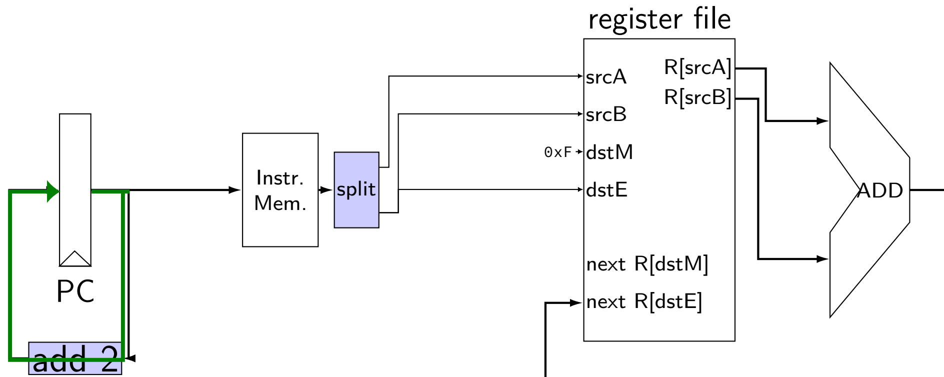


sequential addq paths



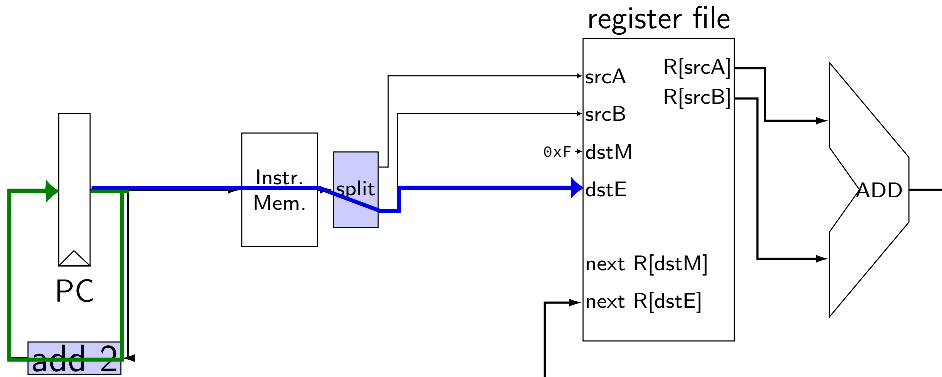
sequential addq paths

path 1: 25 picoseconds



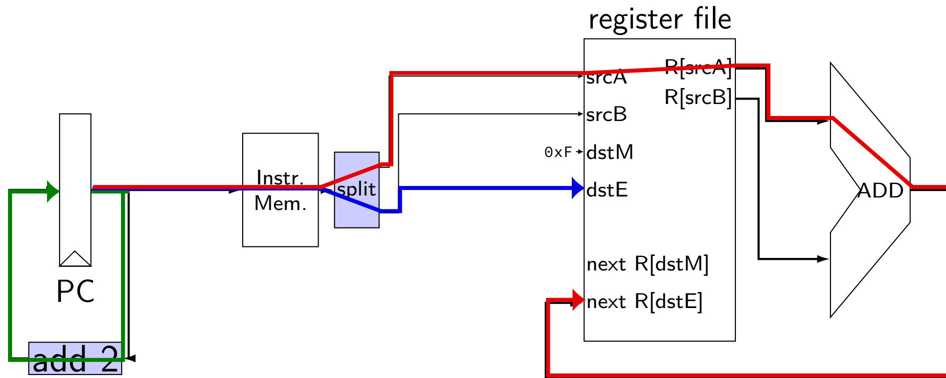
sequential addq paths

path 1: 25 picoseconds
path 2: 375 picoseconds



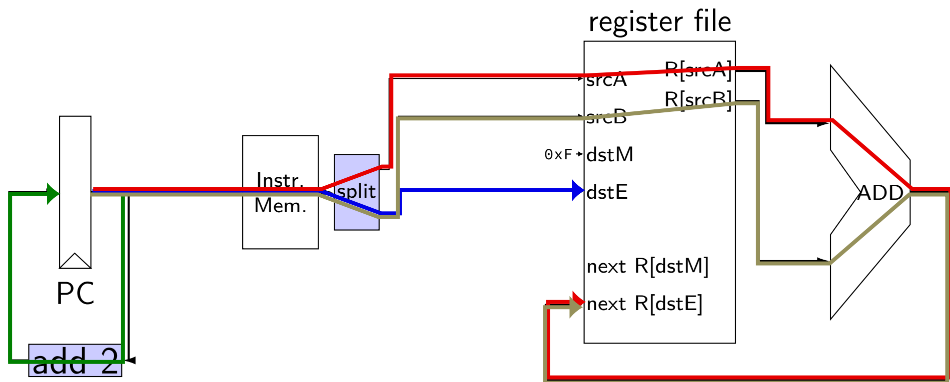
sequential addq paths

path 1: 25 picoseconds
path 2: 375 picoseconds
path 3: 500 picoseconds



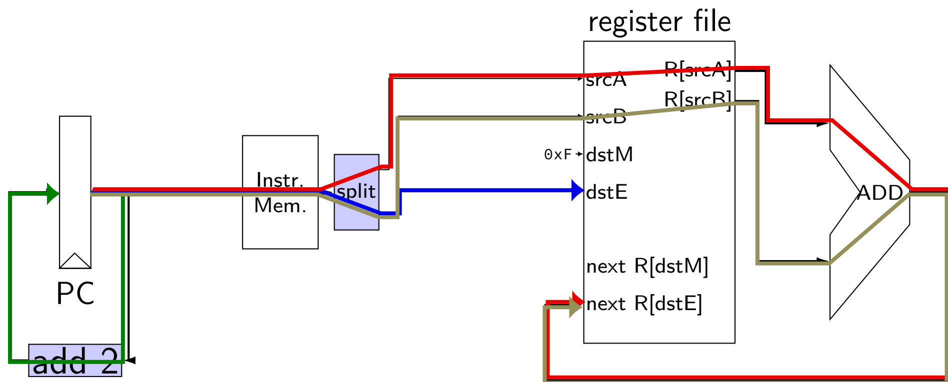
sequential addq paths

path 1: 25 picoseconds
path 2: 375 picoseconds
path 3: 500 picoseconds
path 4: 500 picoseconds

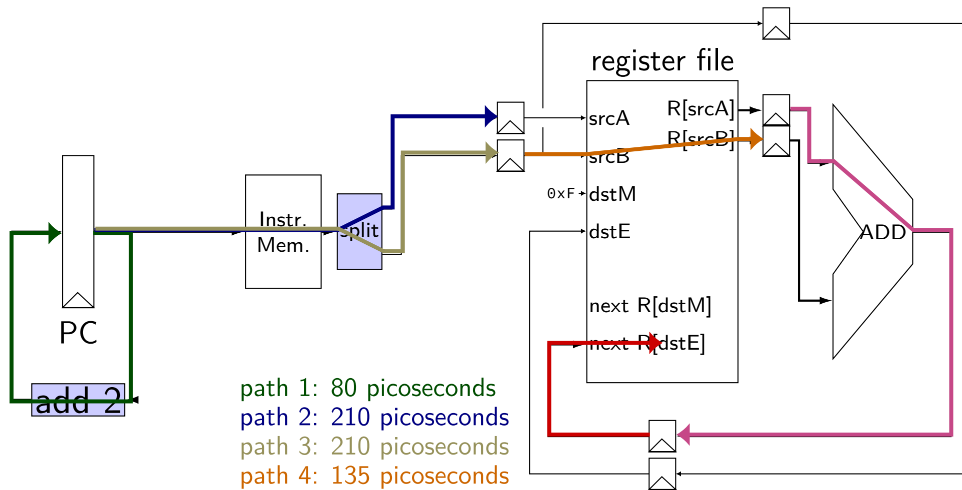


sequential addq paths

path 1: 25 picoseconds
path 2: 375 picoseconds
path 3: **500 picoseconds**
path 4: **500 picoseconds**
overall cycle time: **500 picoseconds** (longest path)

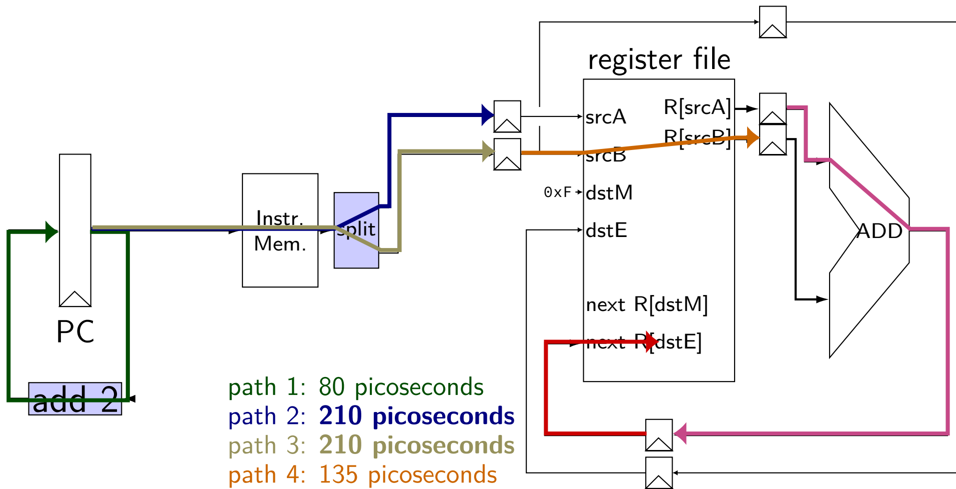


pipelined addq paths



path 1: 80 picoseconds
path 2: 210 picoseconds
path 3: 210 picoseconds
path 4: 135 picoseconds
path 5: 110 picoseconds
path 6: 135 picoseconds
…
overall cycle time: 210 picoseconds

pipelined addq paths

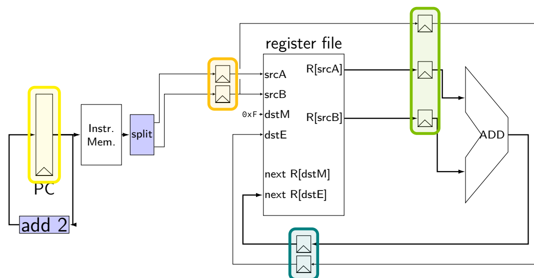


path 1: 80 picoseconds
path 2: **210 picoseconds**
path 3: **210 picoseconds**
path 4: 135 picoseconds
path 5: 110 picoseconds
path 6: 135 picoseconds
...
overall cycle time: **210 picoseconds**

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)

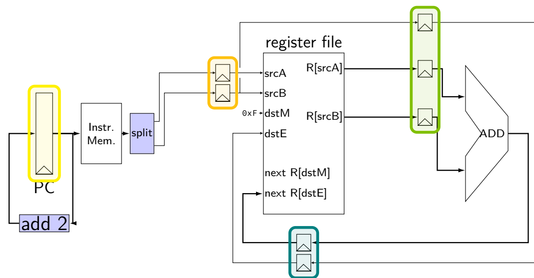
pipelining: 1 instruction per 200 ps + pipeline register delays

- slowest path through stage + pipeline register delays
- latency: 800 ps + pipeline register delays (4 cycles)

addq processor timing exercise 1

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



exercise 1: when instruction 1 stores its result in %rbx, what did instruction 3 *just complete*?

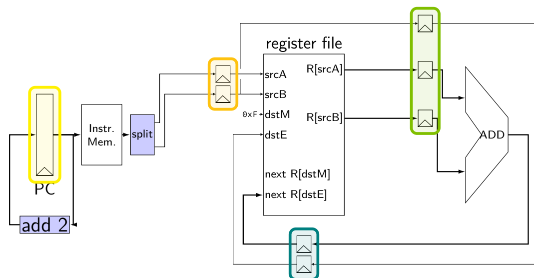
```
addq %rax, %rbx   /* 1 */
addq %rcx, %rdx
addq %r8, %r9     /* 3 */
```

A. fetch **B.** decode (register read) **C.** execute
D. writeback **E.** nothing (it's not running)

addq processor performance exercise 2

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



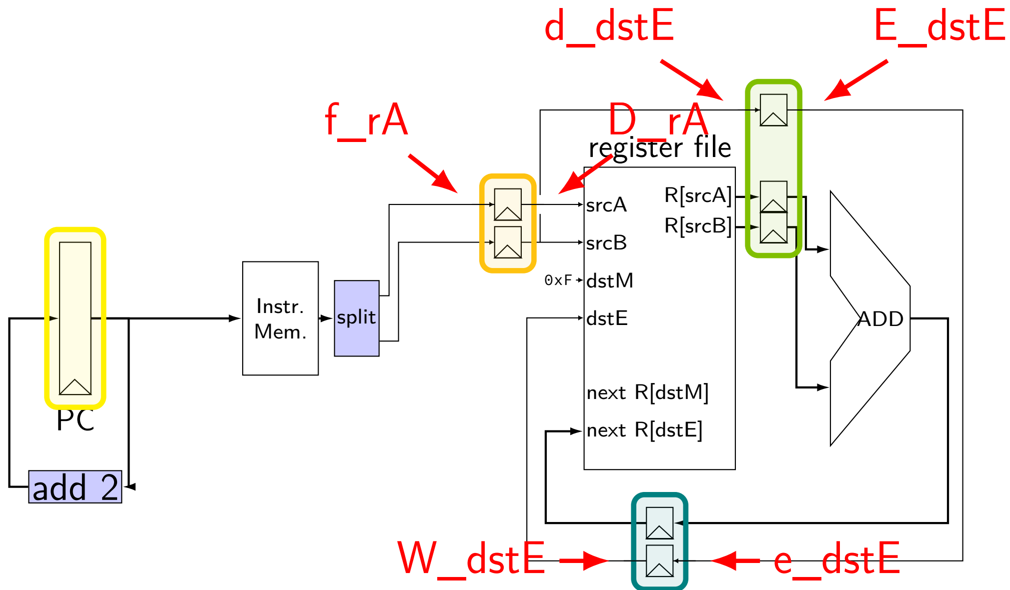
now, cycle time = 200 ps + register delays per cycle

exercise 2:

suppose we combine add and register file read together — new cycle time?

A. 200 ps + reg delay
B. 225 ps + reg delay
C. 250 ps + reg delay
D. 280 ps + reg delay
E. 325 ps + reg delay
E. something else

pipeline register naming convention



pipeline register naming convention

f — fetch sends values here

D — decode receives values here

d — decode sends values here

…

addq HCL

```
...
/* f: from fetch */
f_rA = i10bytes[12..16];
f_rB = i10bytes[8..12];

/* fetch to decode */
/* f_rA -> D_rA, etc. */
register fD {
    rA : 4 = REG_NONE;
    rB : 4 = REG_NONE;
}
```

```
/* D: to decode
   d: from decode */
d_dstE = D_rB;
/* use register file: */
reg_srcA = D_rA;
d_valA = reg_outputA;
...

/* decode to execute */
register dE {
    dstE : 4 = REG_NONE;
    valA : 64 = 0;
    valB : 64 = 0;
}

...
```

addq fetch/decode

unpipelined

```
/* Fetch+PC Update*/
pc = P_pc;
p_pc = pc + 2;
rA = i10bytes[12..16];
rB = i10bytes[8..12];
/* Decode */
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
valA = reg_outputA;
valB = reg_outputB;
```

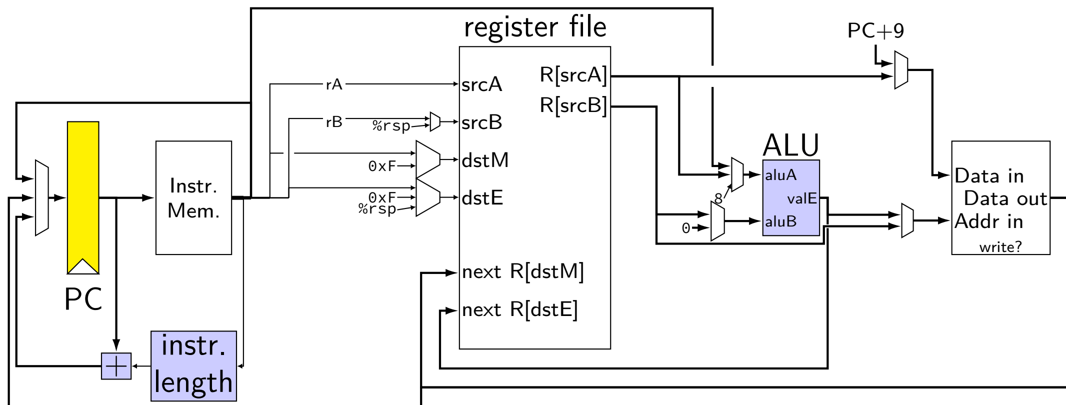
pipelined

```
/* Fetch+PC Update*/
pc = P_pc;
p_pc = pc + 2;
f_rA = i10bytes[12..16];
f_rB = i10bytes[8..12];
/* Decode */
reg_srcA = D_rA;
reg_srcB = D_rB;
d_dstE = D_rB;
d_valA = reg_outputA;
d_valB = reg_outputB;
```

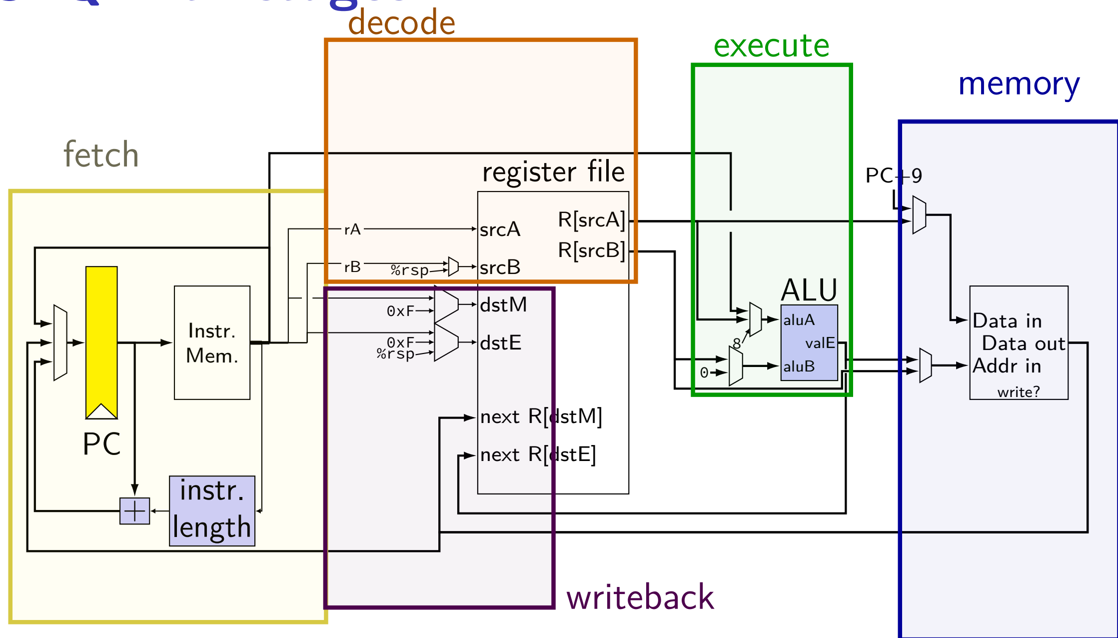
addq pipeline registers

```
register pP {
    pc : 64 = 0;
};
/* Fetch+PC Update*/
register fD {
    rA : 4 = REG_NONE; rB : 4 = REG_NONE;
};
/* Decode */
register dE {
    valA : 64 = 0; valB : 64 = 0; dstE : 4 = REG_NONE;
}
/* Execute */
register eW {
    valE : 64 = 0; dstE : 4 = REG_NONE;
}
/* Writeback */
```

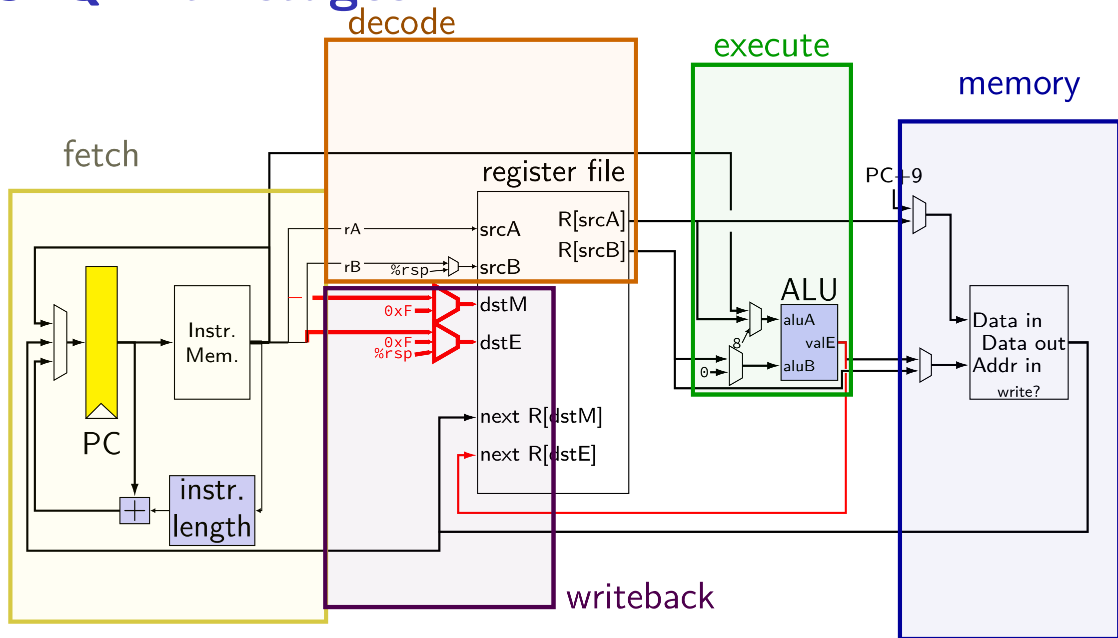
SEQ without stages



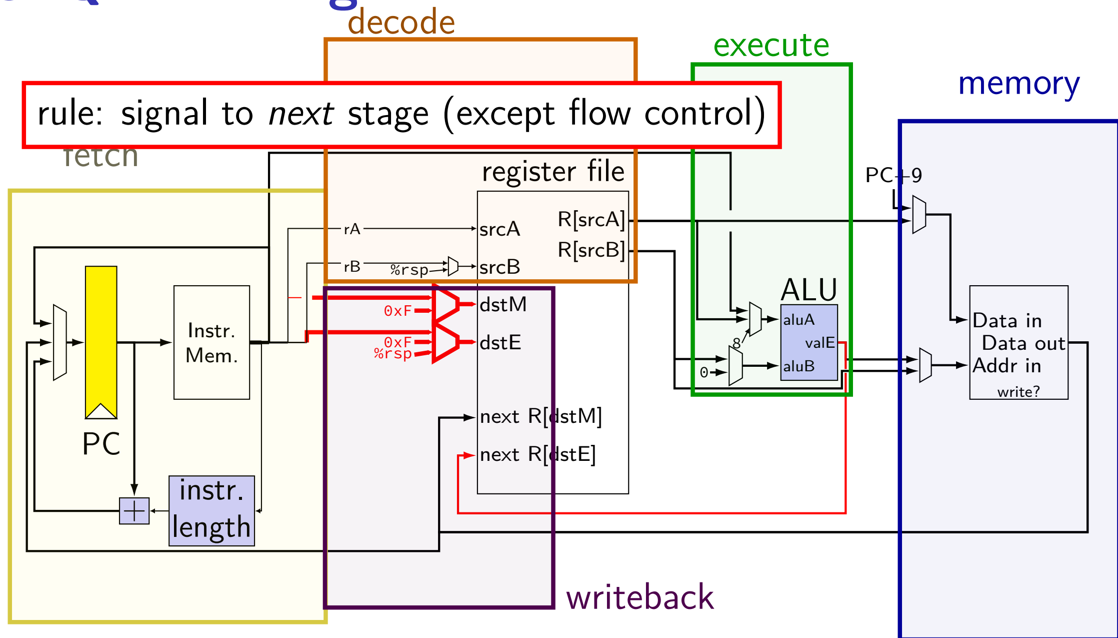
SEQ with stages



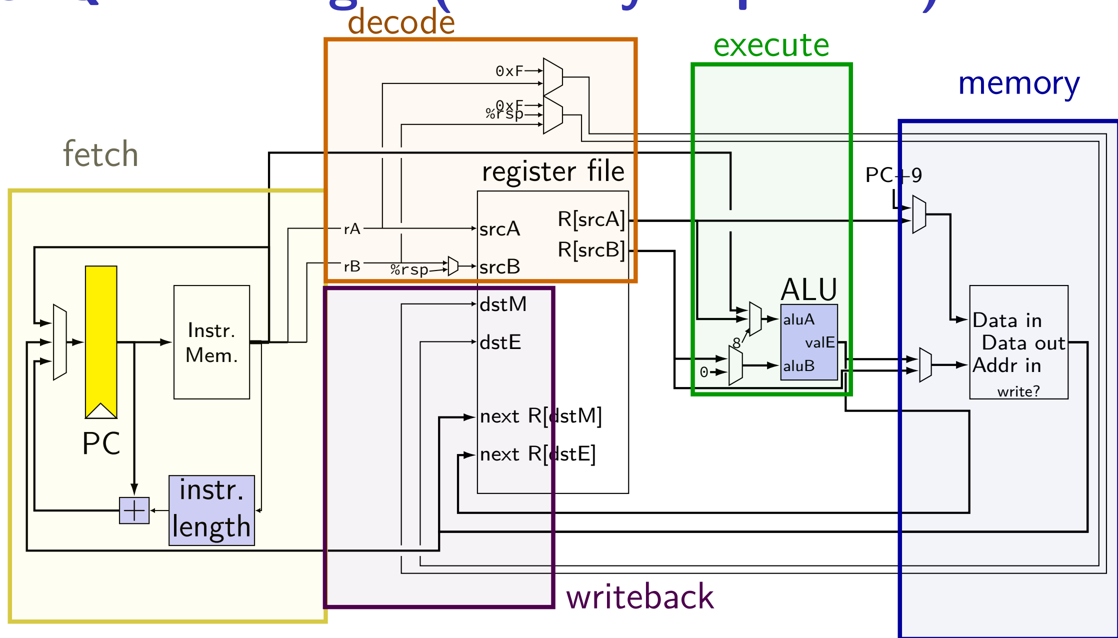
SEQ with stages



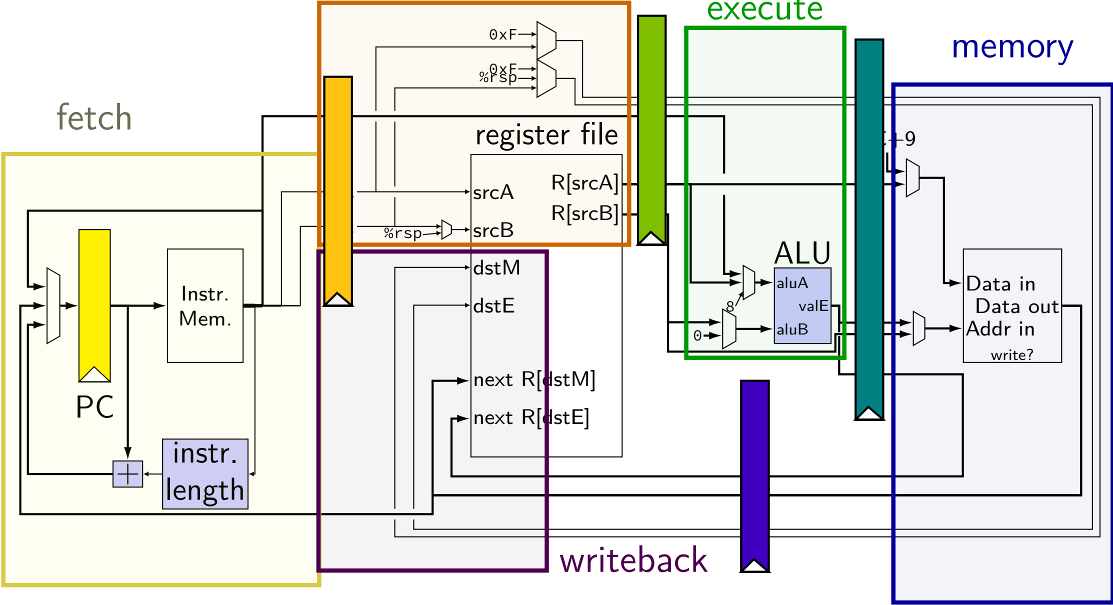
SEQ with stages



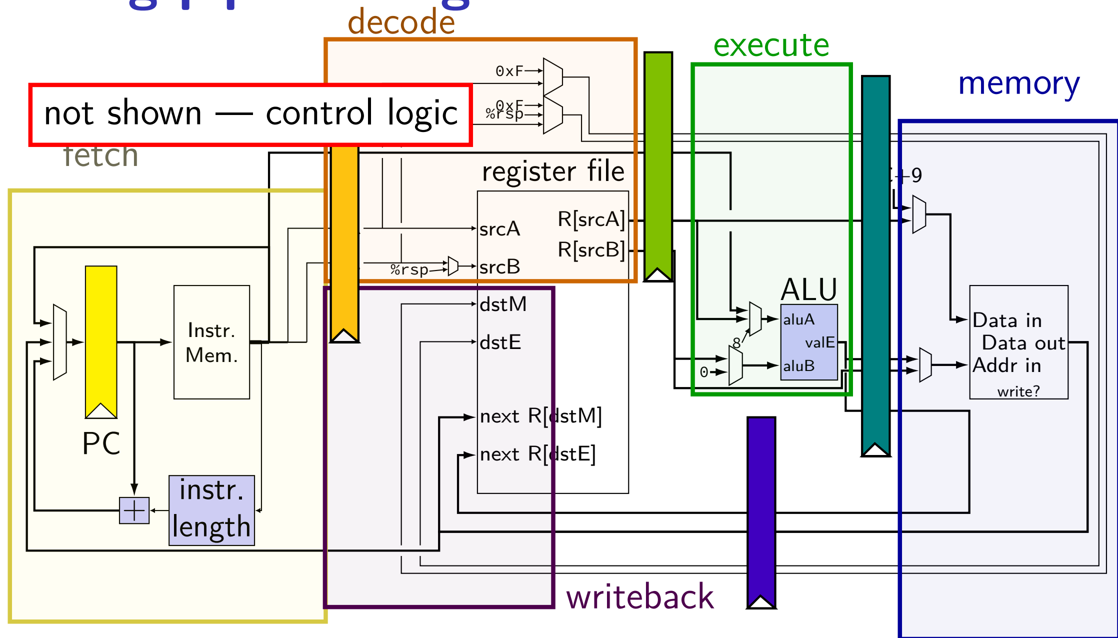
SEQ with stages (actually sequential)



adding pipeline registers



adding pipeline registers



passing values in pipeline

read prior stage's outputs

- e.g. decode: get from fetch via pipeline registers (`D_icode`, …)

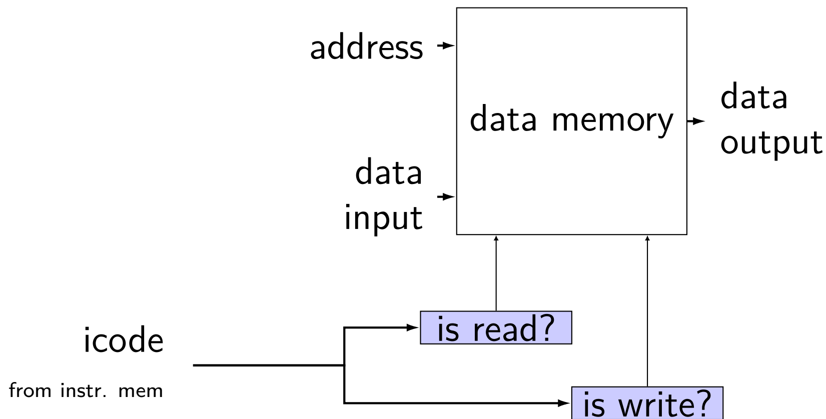
send inputs for next stage

- e.g. decode: send to execute via pipeline registers (`d_icode`, …)

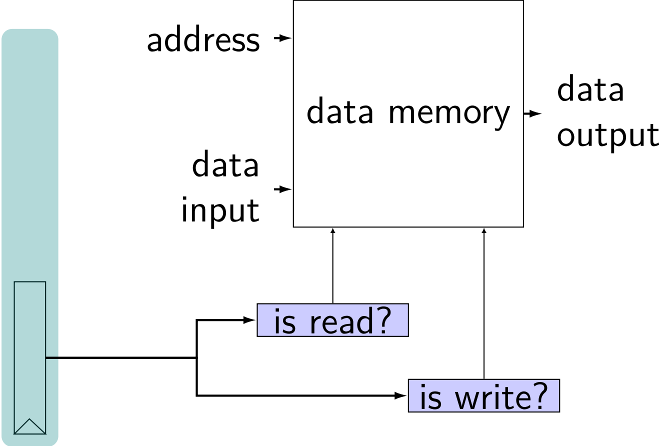
exceptions: deliberate sharing between instructions

- via register file/memory/etc.
- via control flow instructions

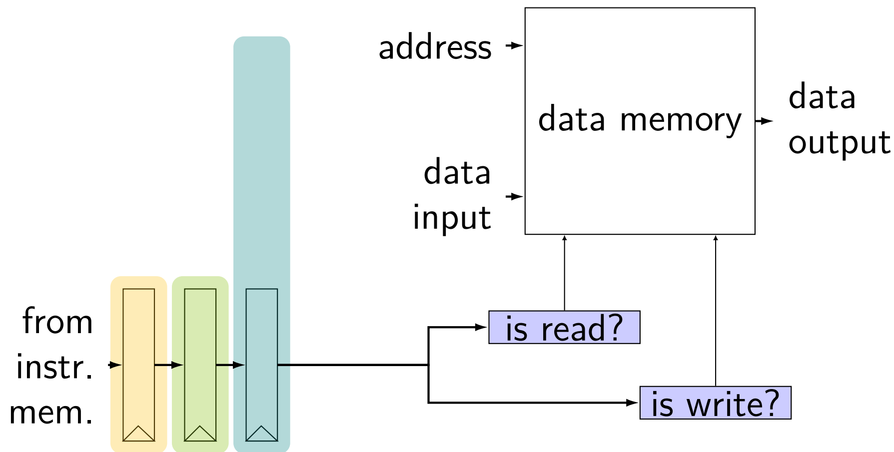
memory read/write logic



memory read/write logic



memory read/write logic



memory read/write: SEQ code

```
icode = i10bytes[4..8];
mem_readbit = [
    icode == MRMOVQ || ...: 1;
    0;
];
```

memory read/write: PIPE code

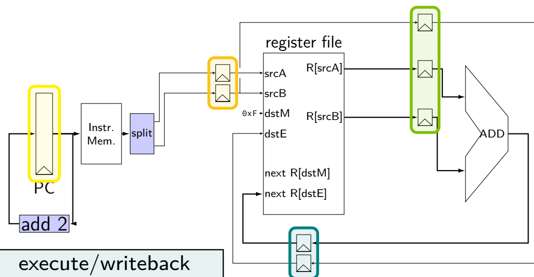
```
f_icode = i10bytes[4..8];
register fD { /* and dE and eM and mW */
    icode : 4 = NOP;
}
d_icode = D_icode;
...
e_icode = E_icode;
mem_readbit = [
    M_icode == MRMOVQ || ...: 1;
    0;
];
```

memory read/write: PIPE code

```
f_icode = i10bytes[4..8];
register fD { /* and dE and eM and mW */
    icode : 4 = NOP;
}
d_icode = D_icode;
...
e_icode = E_icode;
mem_readbit = [
    M_icode == MRMOVQ || ...: 1;
    0;
];
```

addq processor: data hazard

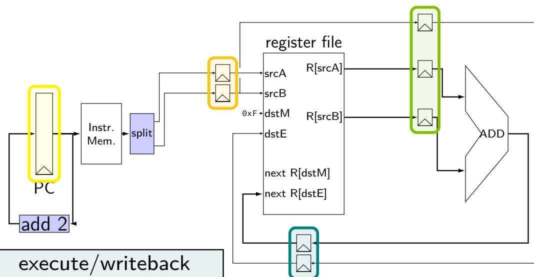
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```



	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

addq processor: data hazard

```
// 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

data hazard

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

step#	pipeline implementation	ISA specification
1	read r8, r9 for (1)	read r8, r9 for (1)
2	read r9, r8 for (2)	write r9 for (1)
3	write r9 for (1)	read r9, r8 for (2)
4	write r8 for (2)	write r8 ror (2)

pipeline reads older value...

instead of value ISA says was just written

data hazard compiler solution

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

one solution: change the ISA

- all addqs take effect three instructions later

make it compiler's job

problem: recompile everytime processor changes?

data hazard hardware solution

```
addq %r8, %r9
// hardware inserts: nop
// hardware inserts: nop
addq %r9, %r8
```

how about hardware add nops?

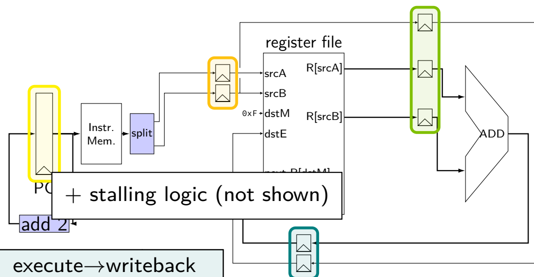
called stalling

extra logic:

- sometimes don't change PC
- sometimes put do-nothing values in pipeline registers

addq processor: data hazard stall

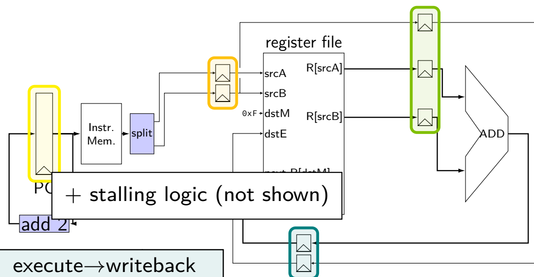
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4	0x4	9	8	---	---	F	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8

addq processor: data hazard stall

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4	0x4	9	8	---	---	F	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8

addq processor: data hazard stall

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```

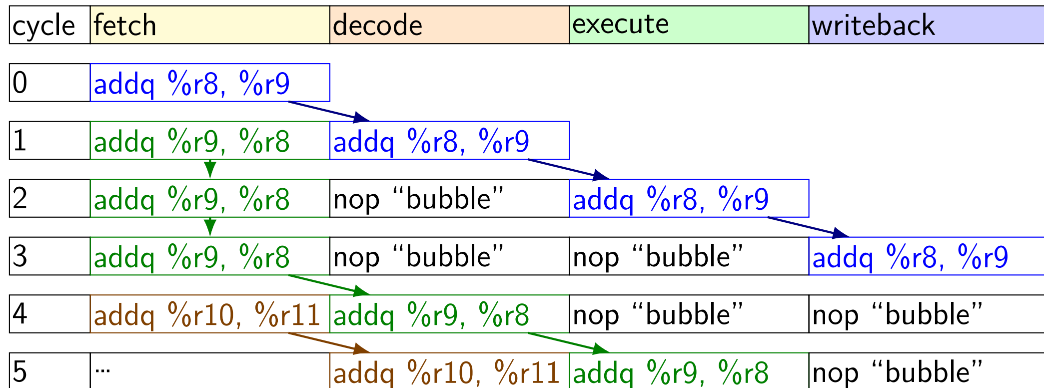
+ stalling logic (not shown)

	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	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4	0x4	9	8	---	---	F	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8

R[9] written during cycle 3; read during cycle 4

addq stall

```
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



cycle	fetch	decode	execute	writeback
0	addq %r8, %r9			
1	addq %r9, %r8	addq %r8, %r9		
2	addq %r9, %r8	nop “bubble”	addq %r8, %r9	
3	addq %r9, %r8	nop “bubble”	nop “bubble”	addq %r8, %r9
4	addq %r10, %r11	addq %r9, %r8	nop “bubble”	nop “bubble”
5	…	addq %r10, %r11	addq %r9, %r8	nop “bubble”

hazards versus dependencies

dependency — X needs result of instruction Y?

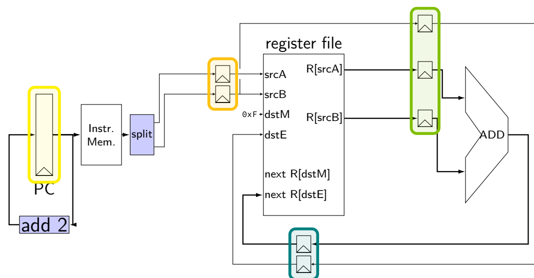
- has potential for being messed up by pipeline
- (since part of X may run before Y finishes)

hazard — will it not work in some pipeline?

- before extra work is done to “resolve” hazards
- multiple kinds: so far, *data hazards*

data hazard exericse

```
addq %r8, %r9
addq %r10, %r11
addq %r9, %r8
addq %r11, %r10
```



to resolve data hazards with stalling, how many stalls are needed?

hint: complete timeline

instr	cycle: 0	1	2	3	4	5	6	7
addq R8, R9	F	D	E	W				
addq R10, R11		F						
addq R9, R8								
addq R11, R10								

data hazard exericse solution

(not on this version of the slides)

revisiting data hazards

stalling worked

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

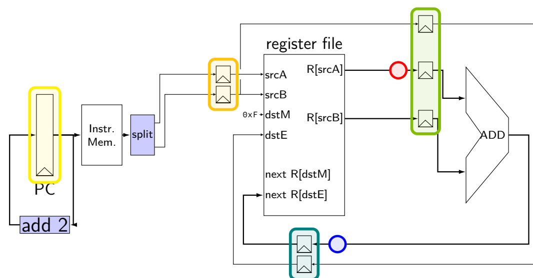
...or more with 5-stage pipeline

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:



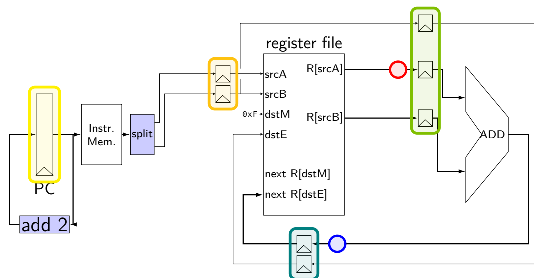

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	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:



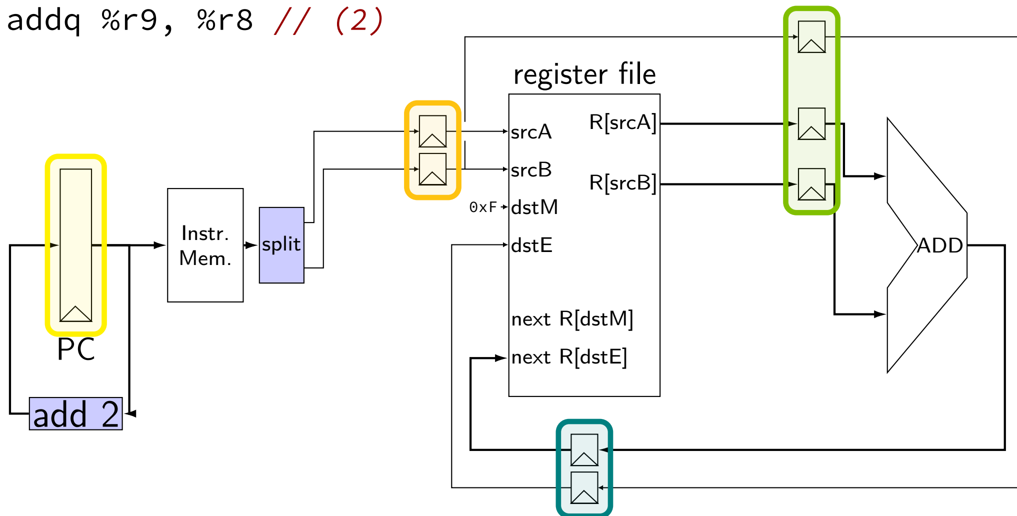

```
// initially %r8 = 800,
//            %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```

	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

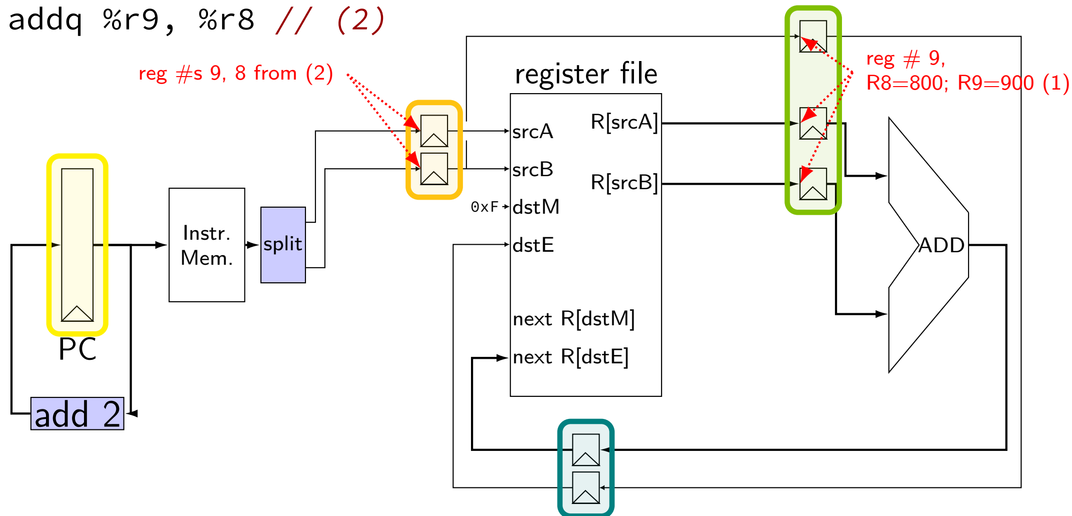
forwarding

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



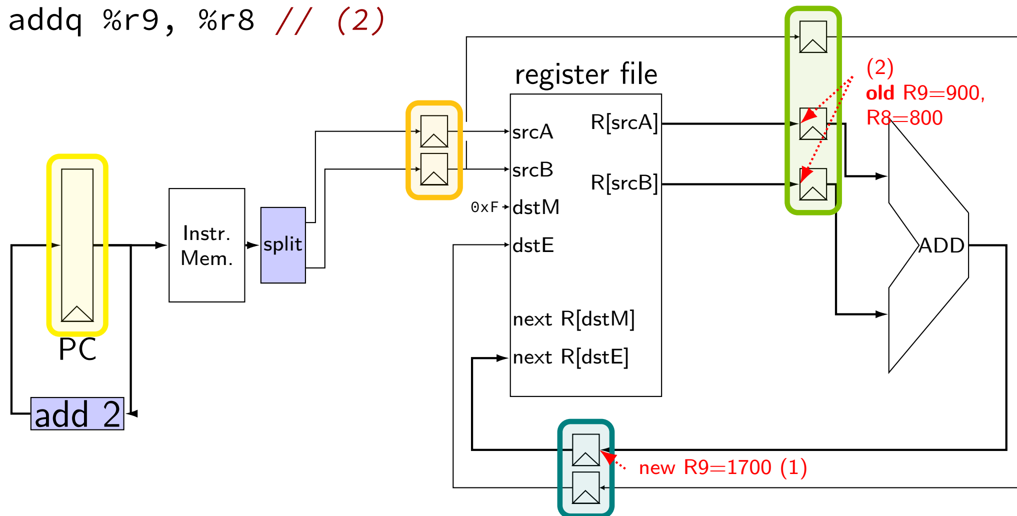
forwarding

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



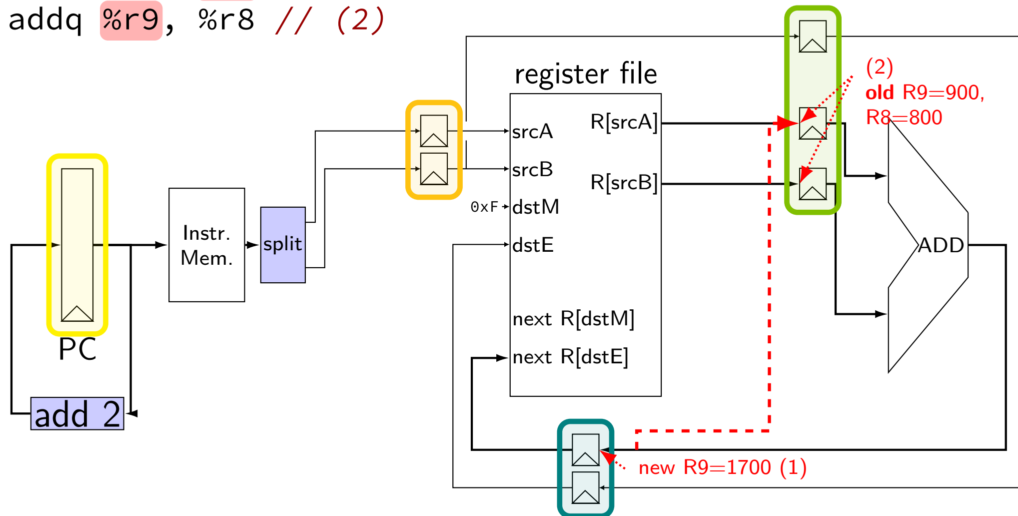
forwarding

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



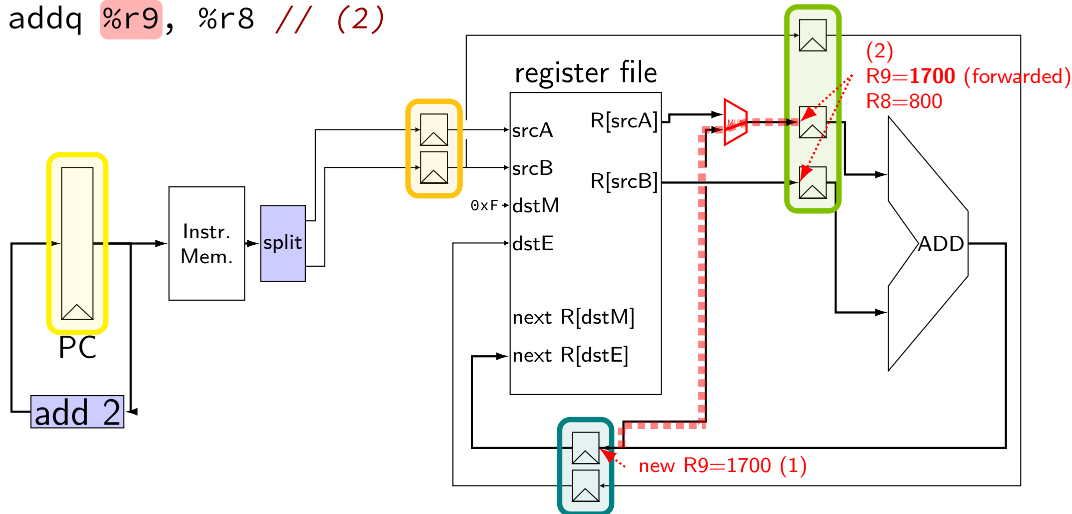
forwarding

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



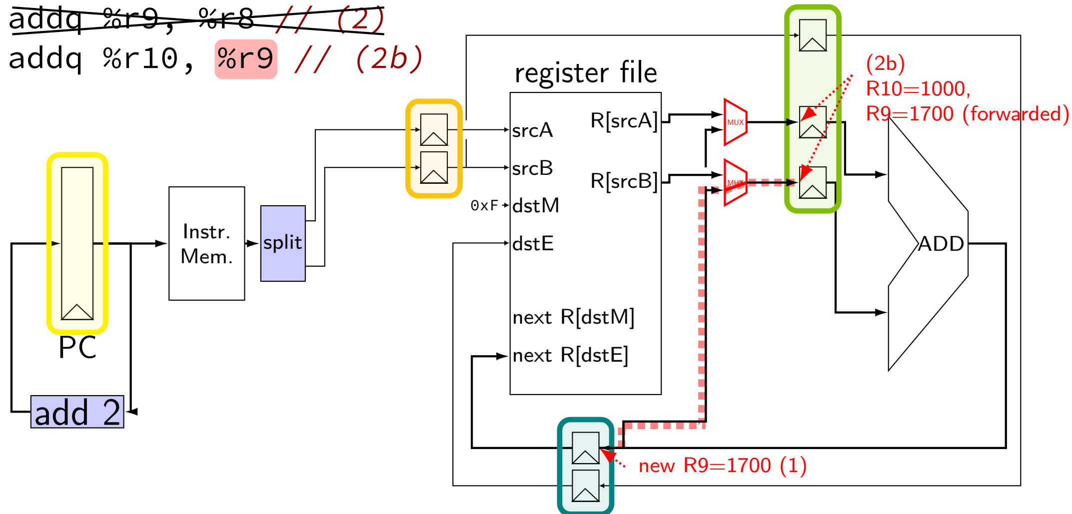
forwarding

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



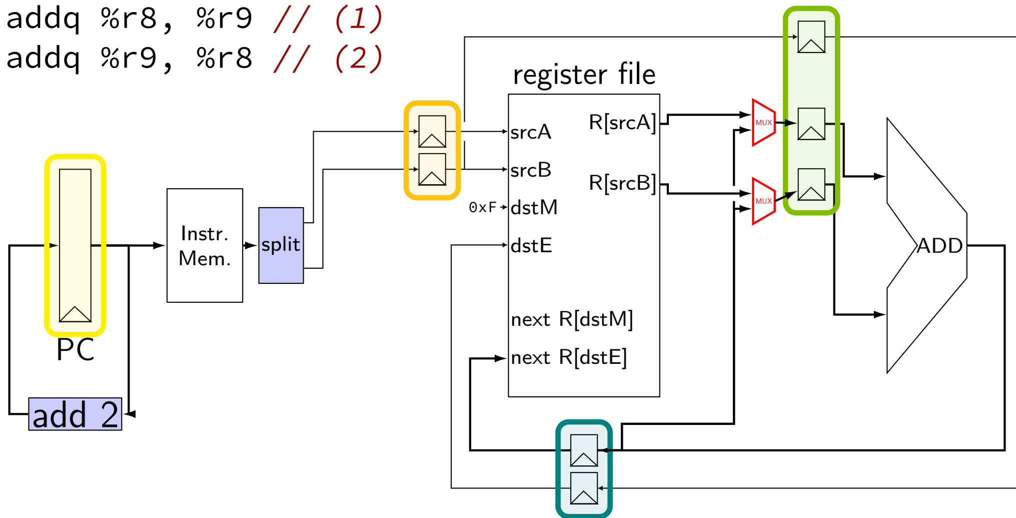
forwarding

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



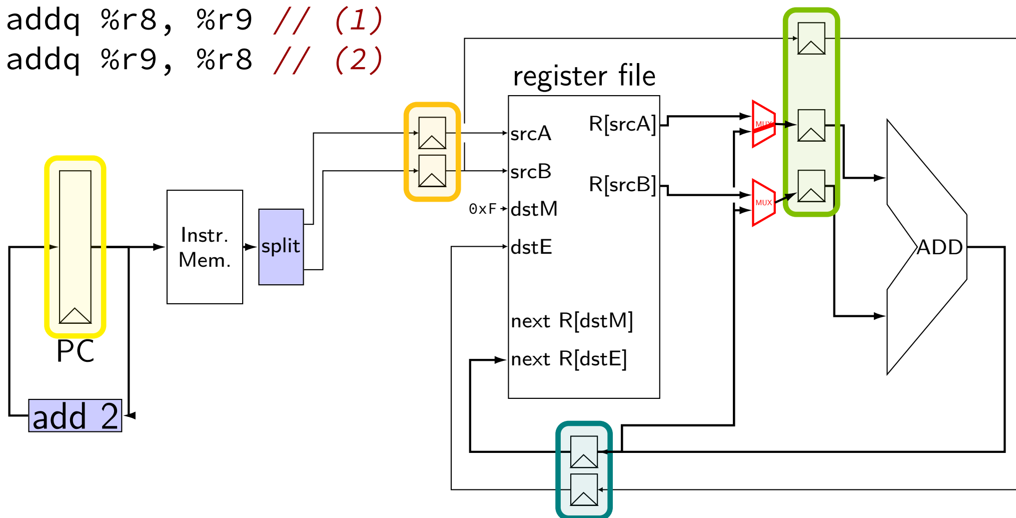
forwarding: MUX conditions

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



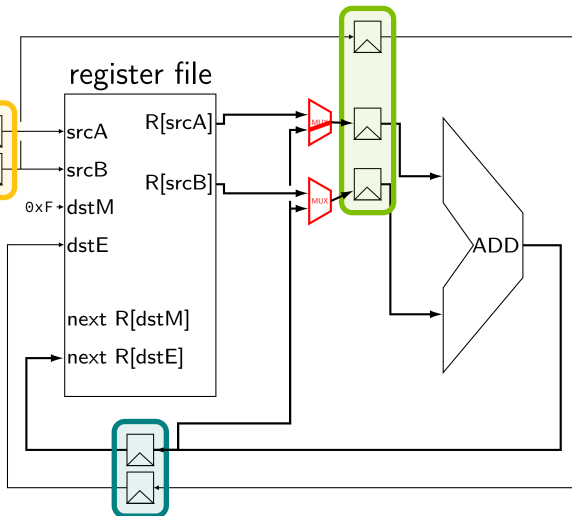
forwarding: MUX conditions

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



forwarding: MUX conditions

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



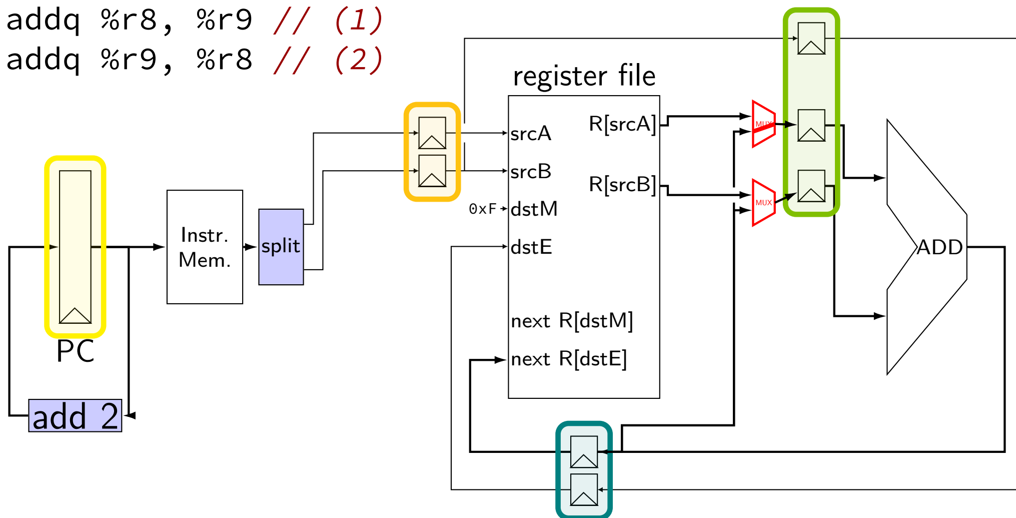

```
d_valA= [
   condition : e_valE;
   1 : reg_outputA;
];
```

What could condition be?

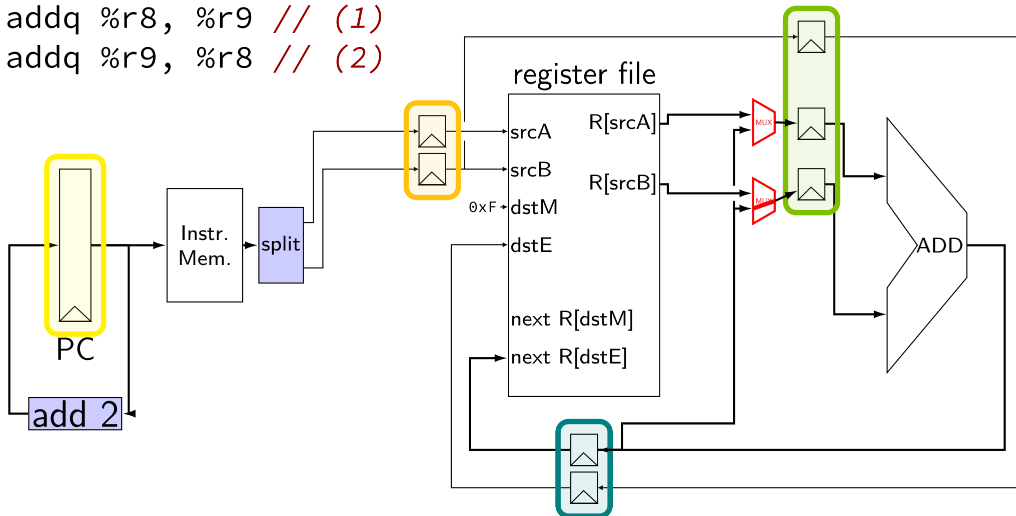
a. W_rA == reg_srcA
b. W_dstE == reg_srcA
c. e_dstE == reg_srcA
d. d_rB == reg_srcA
e. something else

forwarding: MUX conditions

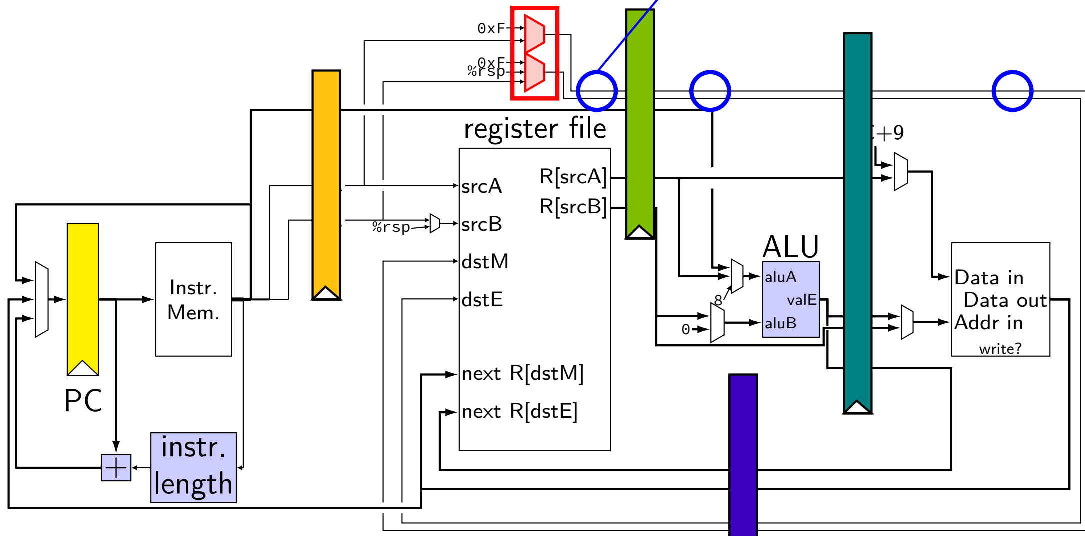
```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



forwarding: MUX conditions



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:

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	W					
`addq %r9, %r8`		✕	✕	F	D	E	W		

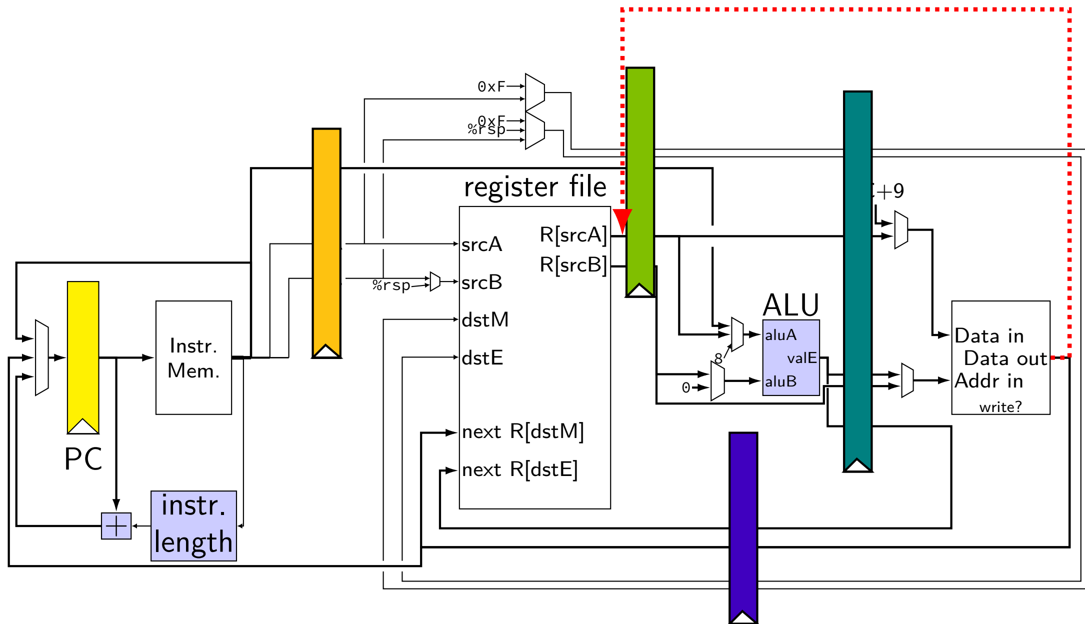
with forwarding:

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

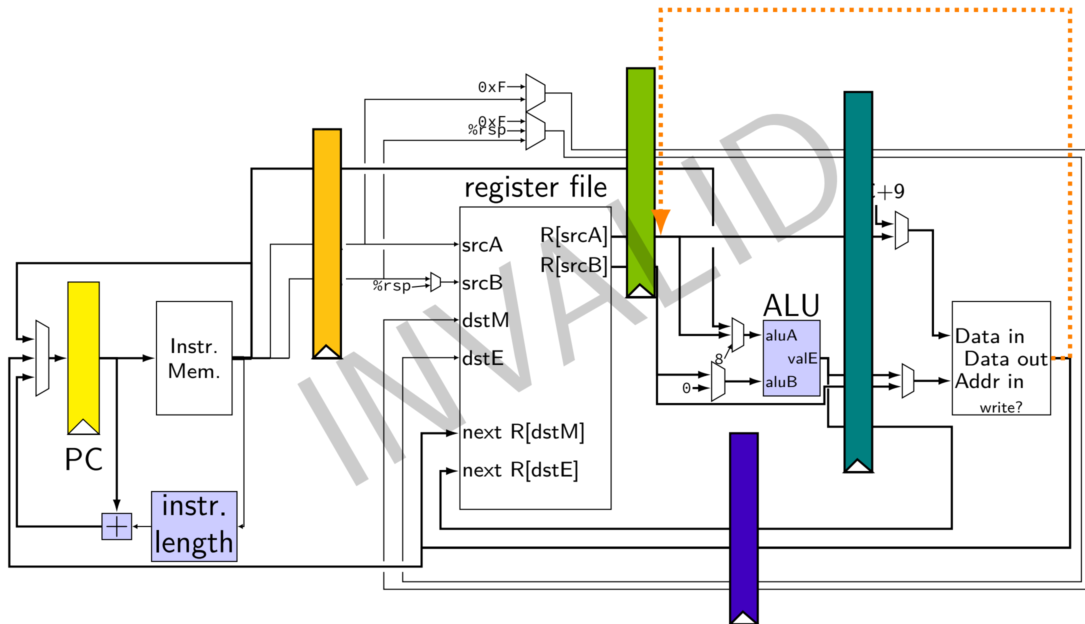
more possible forwarding

0xF
0xF
%rsp
register file
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
%rsp
Instr.
Mem.
PC
instr.
length
ALU
aluA
valE
aluB
8
0
+9
Data in
Data out
Addr in
write?

more possible forwarding

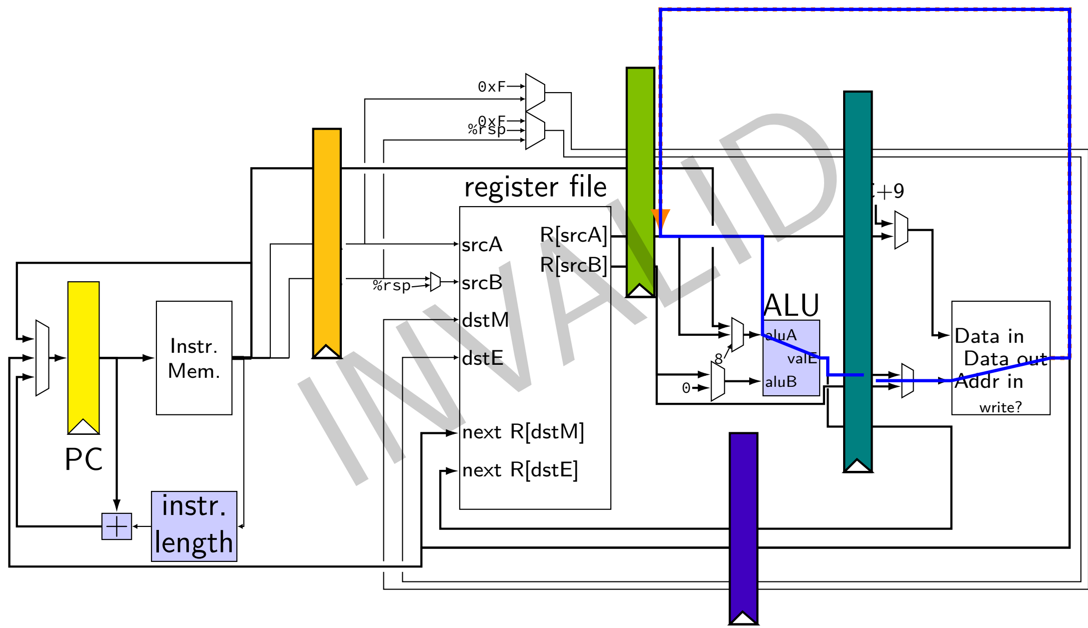


INVALID forwarding path



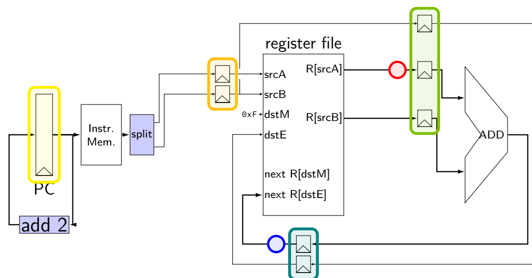
INVALID forwarding path

oops, extra long critical path!



forwarding more?

location of values during cycle 3:



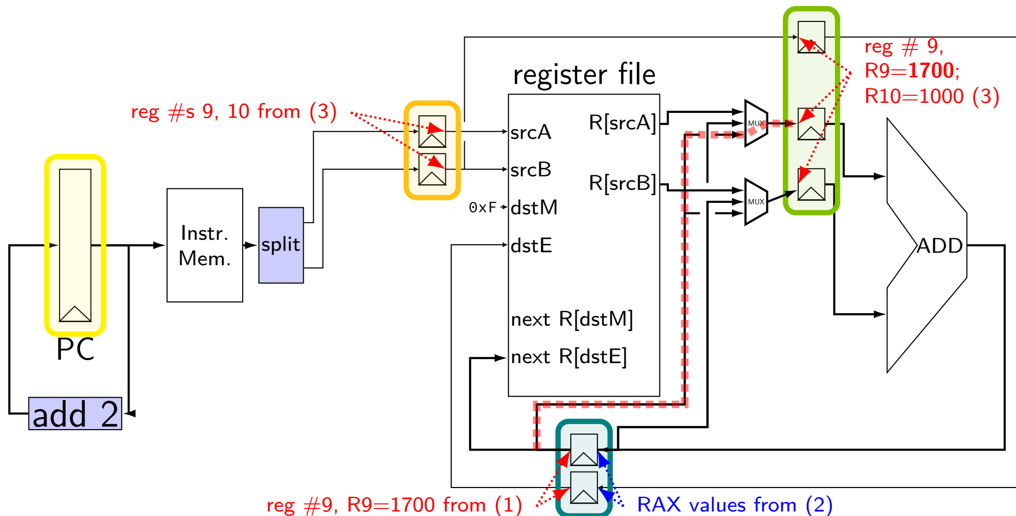

```
// initially %rax = 0,
//           %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %rax, %rax
addq %r9, %r10
```

	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?

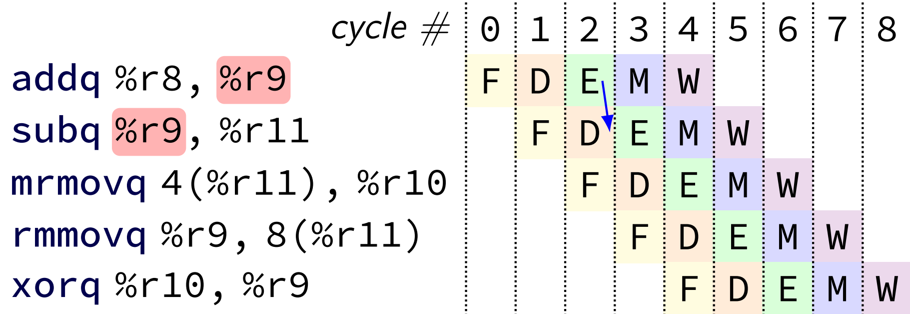
```
addq %r8, %r9    // (1) R9 <- R8 (800) + R9 (900)
addq %rax, %rax  // (2)
addq %r9, %r10   // (3) R10 <- R9 (1700) + R10 (1000)
```



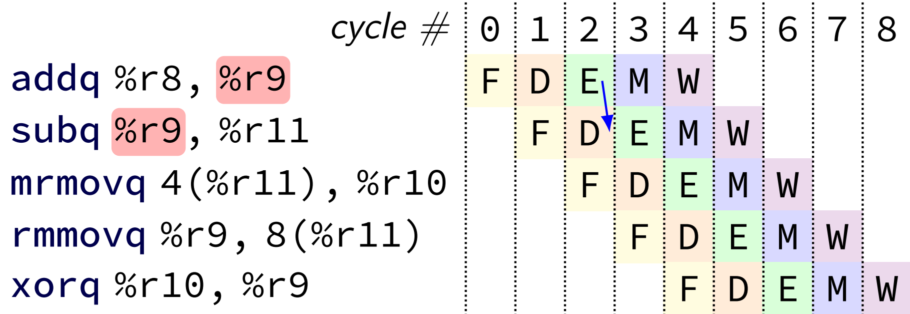
some forwarding paths

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

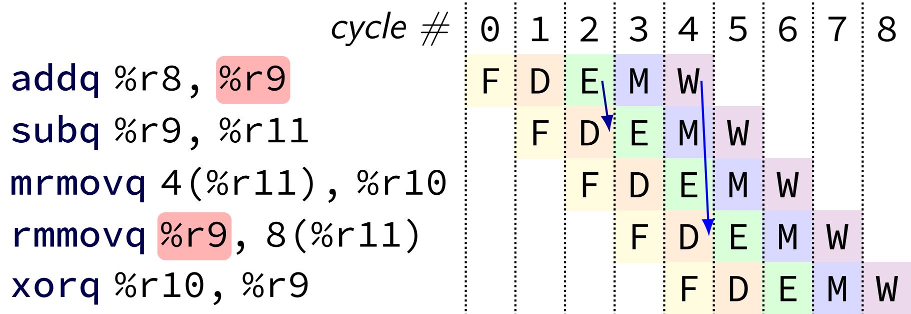
some forwarding paths



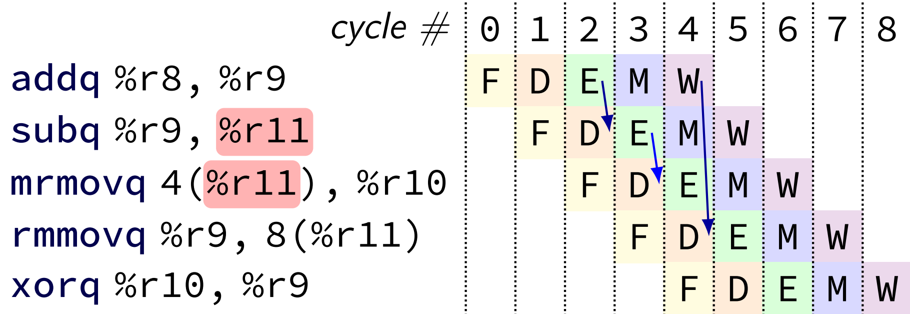
some forwarding paths



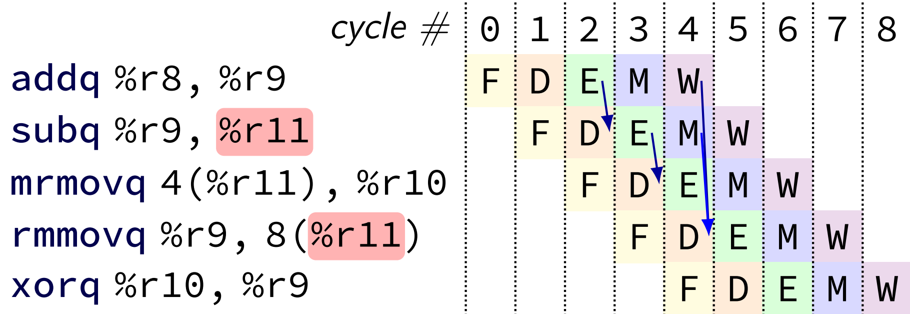
some forwarding paths



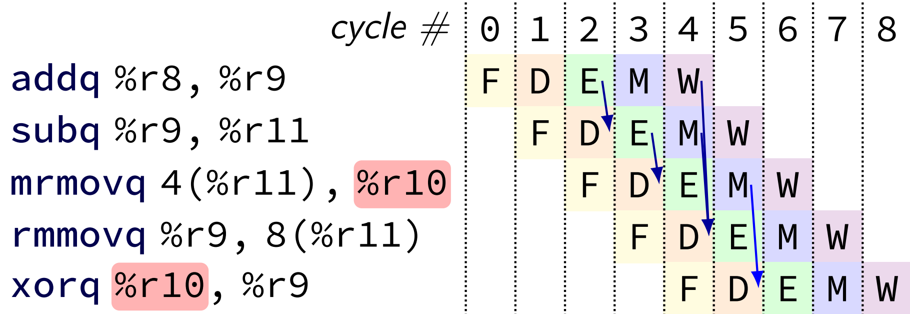
some forwarding paths



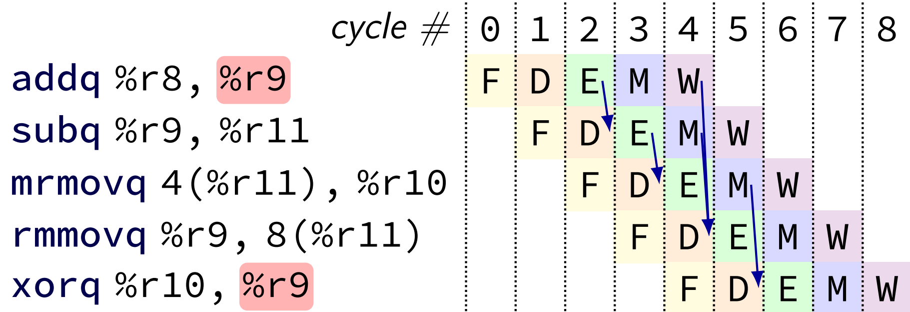
some forwarding paths



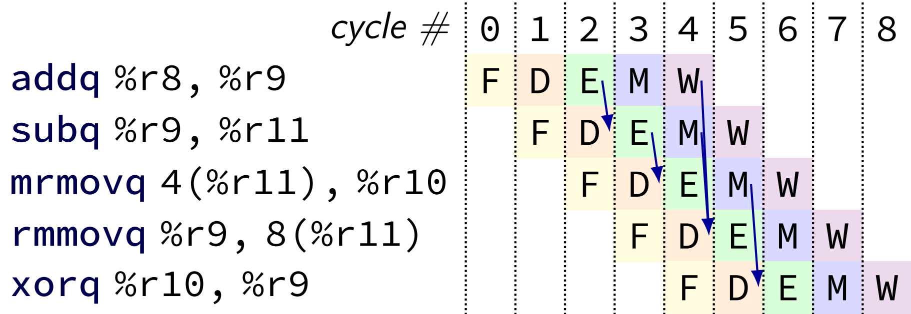
some forwarding paths



some forwarding paths



some forwarding paths



multiple forwarding paths (1)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r8`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

multiple forwarding paths (1)

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`		F	D	E	M	W				
`addq %r11, %r8`			F	D	E	M	W			
`addq %r12, %r8`				F	D	E	M	W		

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)

	cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8		F	D	E	M	W				
addq %r11, %r12			F	D	E	M	W			
addq %r12, %r8				F	D	E	M	W		

multiple forwarding paths (2)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r12`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

multiple forwarding paths (2)

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`		F	D	E	M	W				
`addq %r11, %r12`			F	D	E	M	W			
`addq %r12, %r8`				F	D	E	M	W		

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.

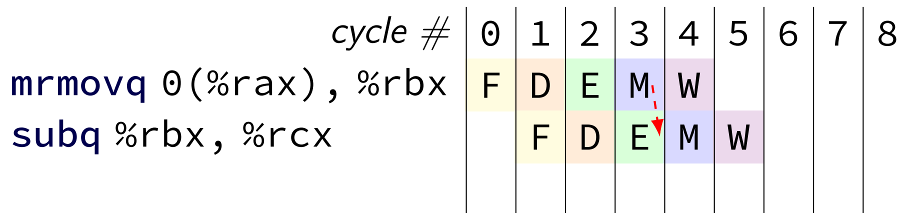
in andq, %r9 is ________ addq.

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

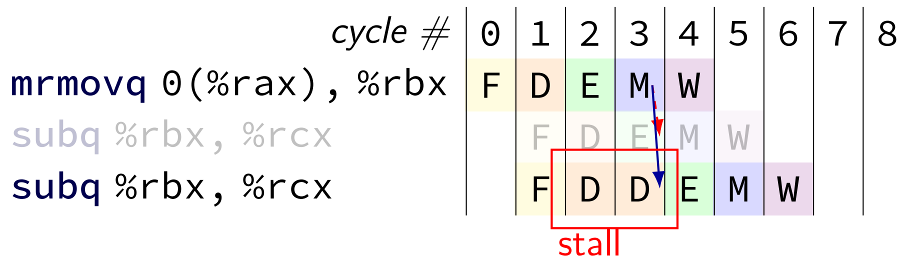
assumption in diagram: hazard detected in `subq`'s decode stage

(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

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

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

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 %rax, %r9`			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

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

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	
`rrmovq %rcx, %r9`						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
```

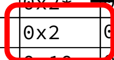
	fetch		fetch→decode		decode→execute			execute→writeback	
cycle	PC								
0	0x0	0/1							
1	0x2*	0/1	8	9					
2	0x2*	0/1	0xF	0xF	800	900	9		
3	0x2	0/0	0xF	0xF	---	---	0xF	1700	9
4	0x10	0/0	0xF	0xF	---	---	0xF	---	0xF
5			10	11	---	---	0xF	---	0xF
6					1000	1100	11	---	0xF

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	[illegible]/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 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		
5	irmovq	rmmovq	addq	jne (done)	OPq

were waiting/nothing

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	[illegible]		
4	rmmovq [?]	addq [?]	jne (use ZF)	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

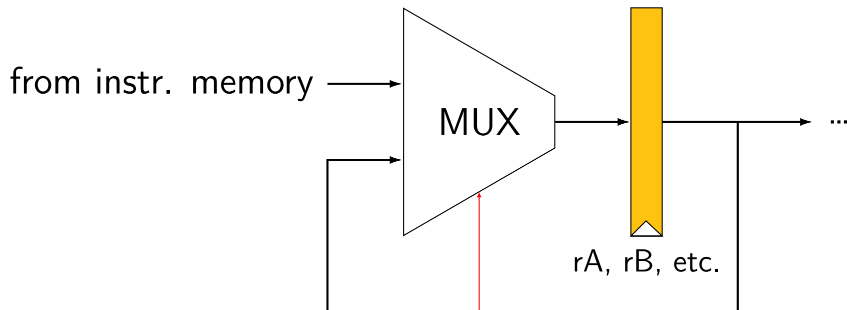
performance

hypothetical instruction mix

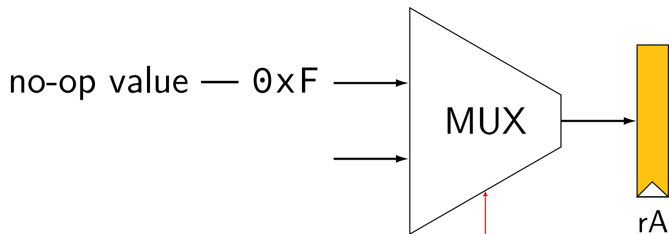
kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
ret	1%	4	4
others	91%	1*	1*

* — ignoring data hazards

fetch/decode logic — advance or not



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 ← 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		???		

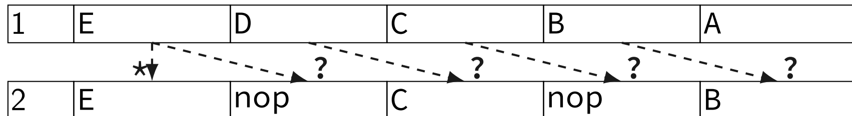
exercise result

```
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)

time	fetch	decode	execute	memory	writeback

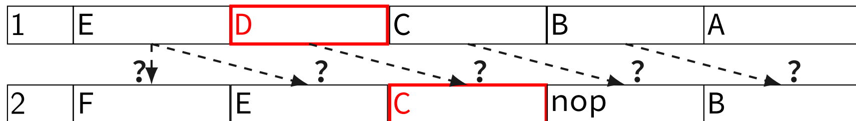


stall (S) = keep old value; normal (N) = use new value

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

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

exercise: squash + stall (2)

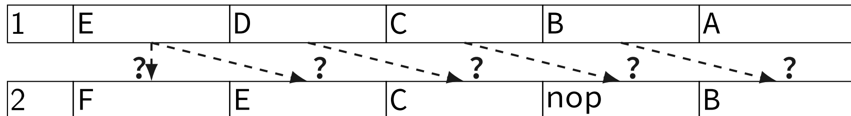


stall (S) = keep old value; normal (N) = use new value

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

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

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	

...

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
`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	
…									

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)

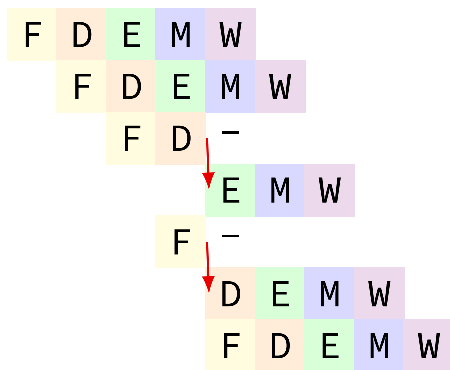
`target: xorq %rax, %rax` (mispredicted)

inserted nop

`andq %rbx, %rcx` (mispredicted)

inserted nop

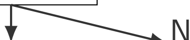
`subq %r9, %r10` (instr. after jle)



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

squashing with stall/bubble

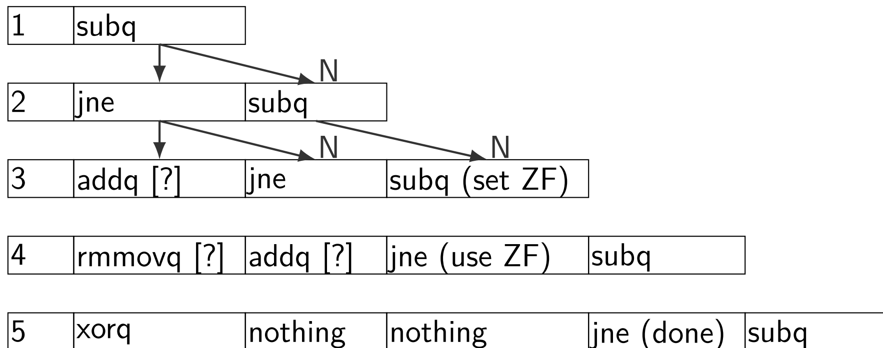
time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq



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

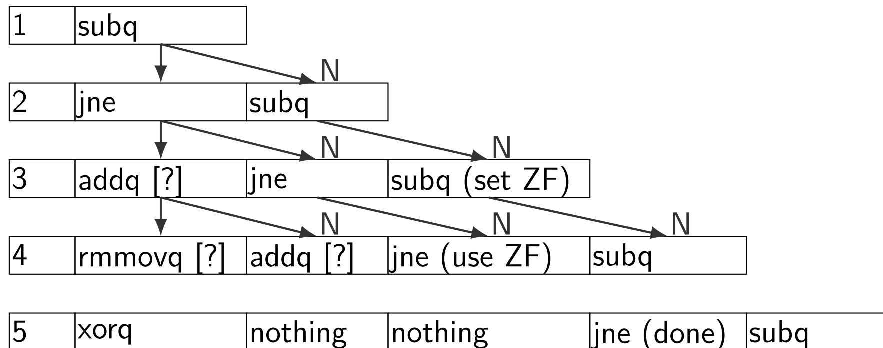
bubble (B) = use default (no-op);

squashing with stall/bubble



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

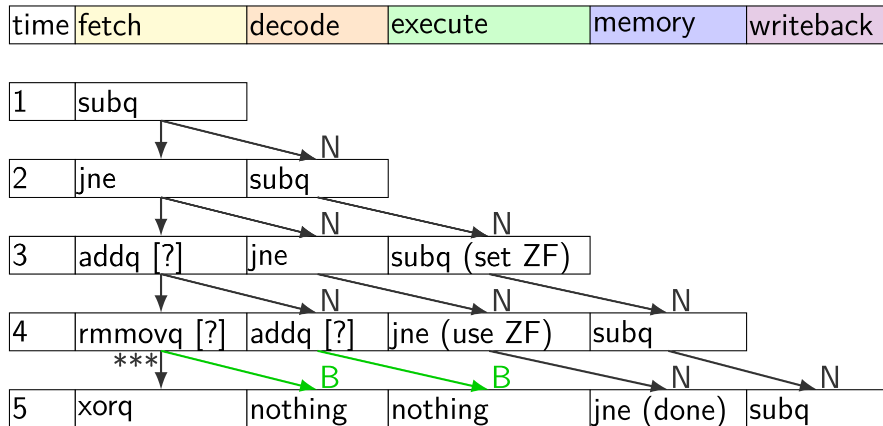
squashing with stall/bubble



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

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

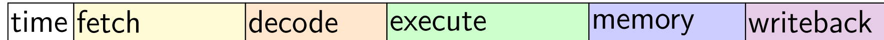
squashing with stall/bubble

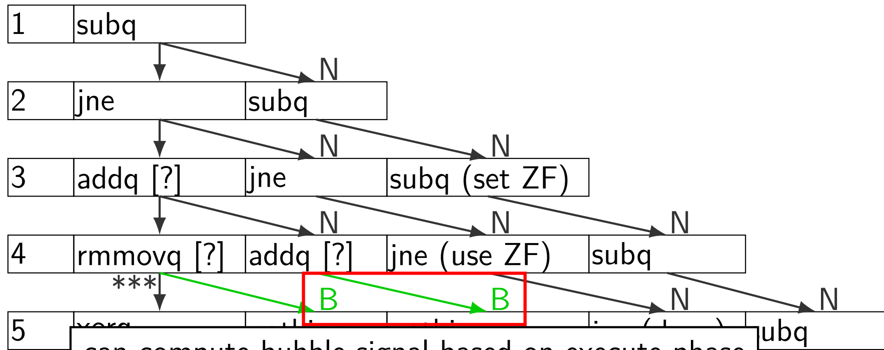


stall (S) = keep old value; normal (N) = use new value

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

squashing with stall/bubble





stal … ew value

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

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

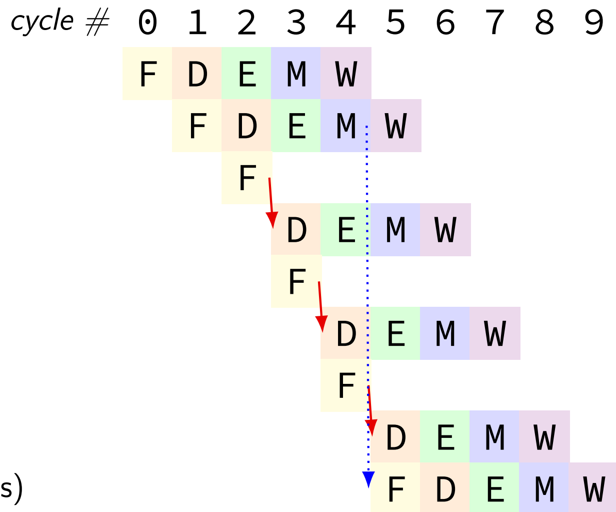
`???`

inserted nop

`???`

inserted nop

`rrmovq %rax, %r8` (return address)



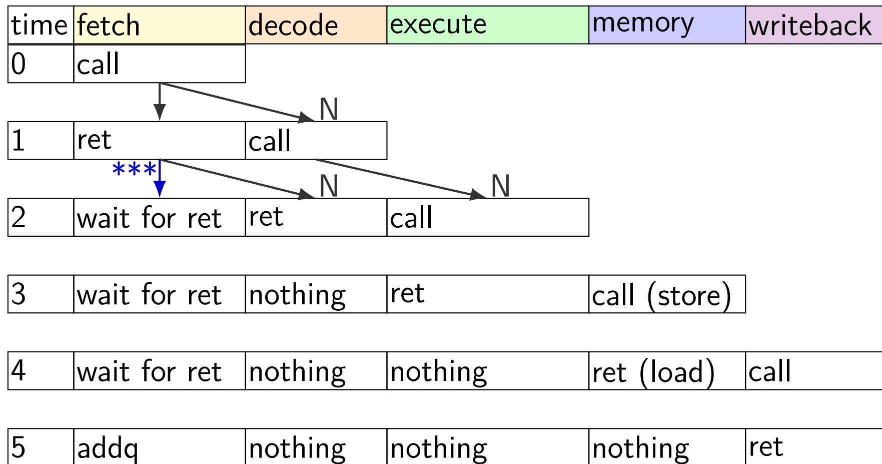
need option: replace instruction with nop (“bubble”)

ret stall

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

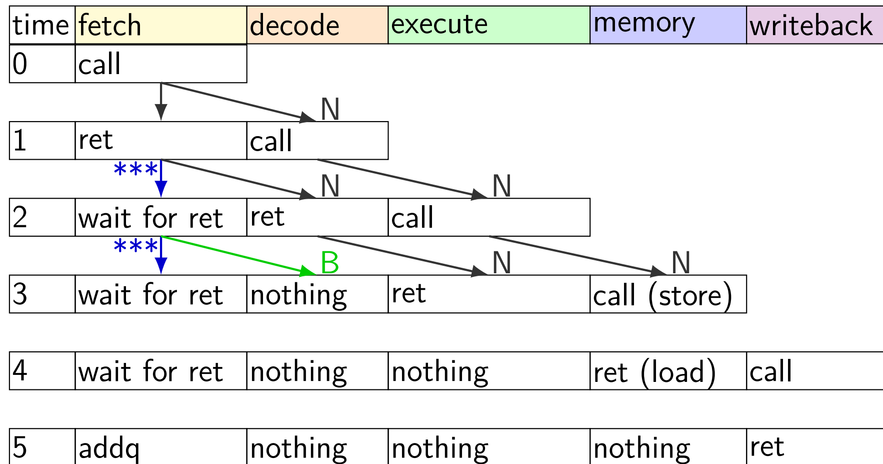
ret stall



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

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

ret stall

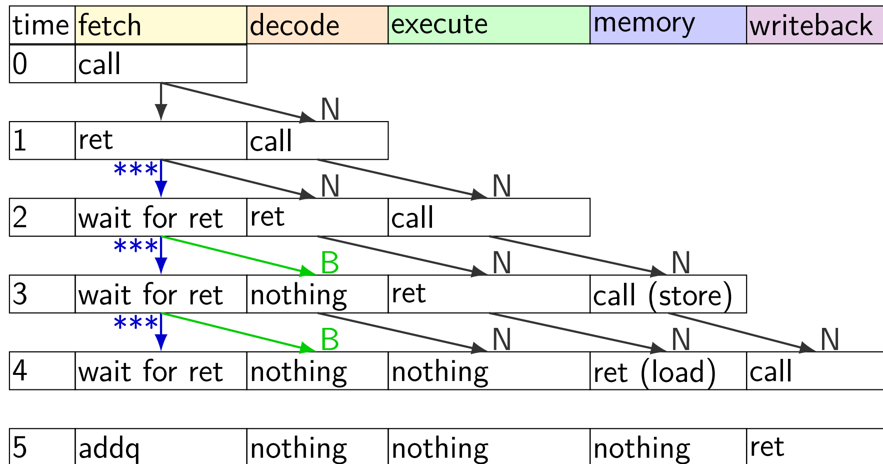


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

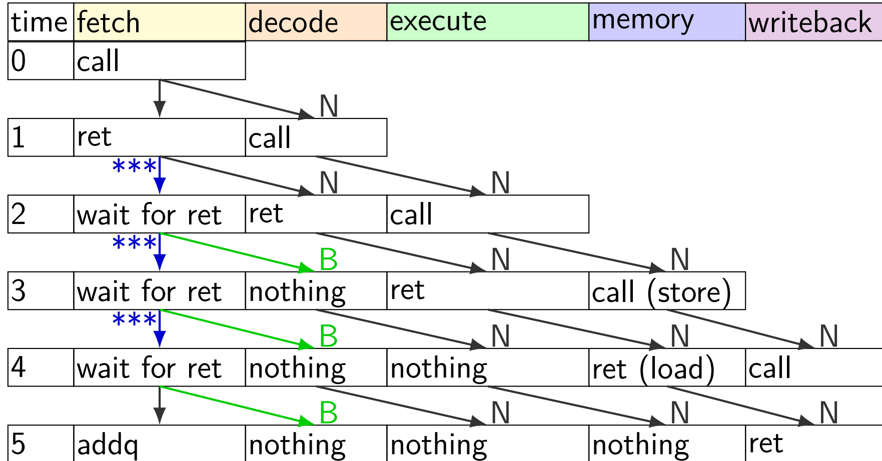
ret stall



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);

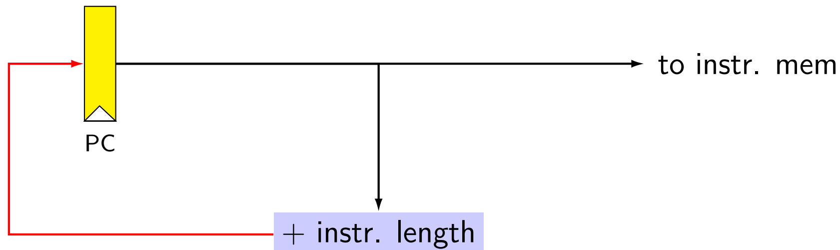
HCLRS bubble example

```
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)

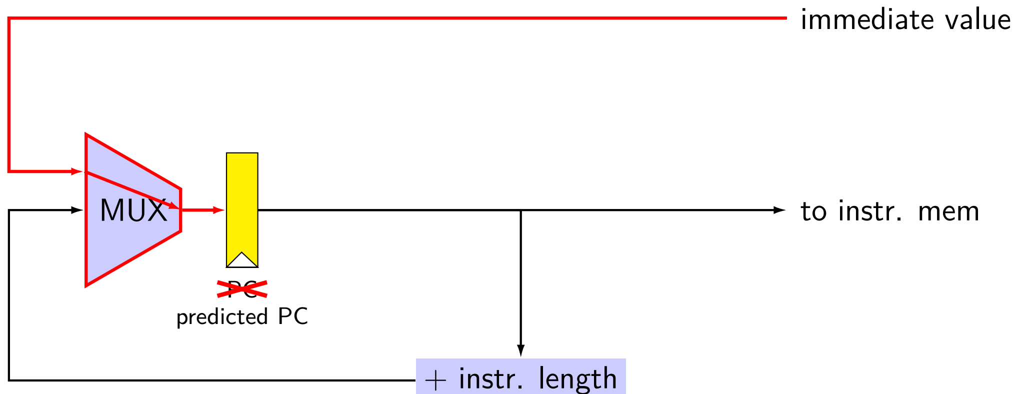
(1) normal case: PC $\leftarrow$ PC + instr len



building the PC update (one possibility)

(1) normal case: PC ← PC + instr len

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

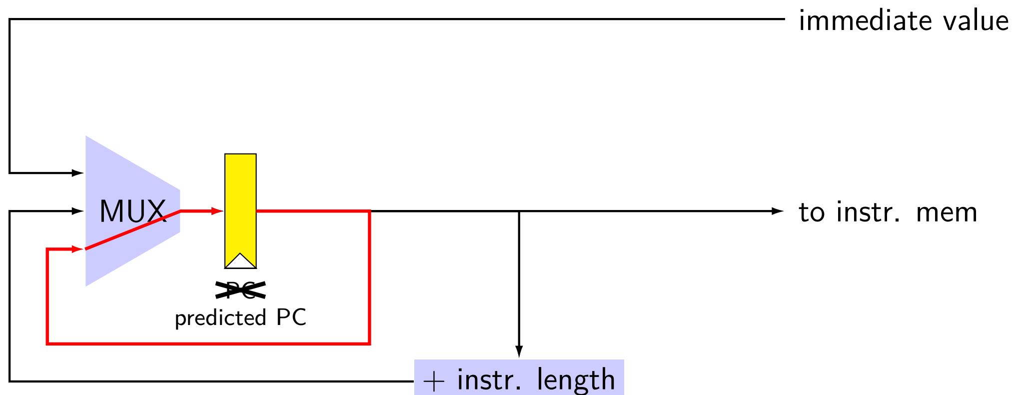


building the PC update (one possibility)

(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?)



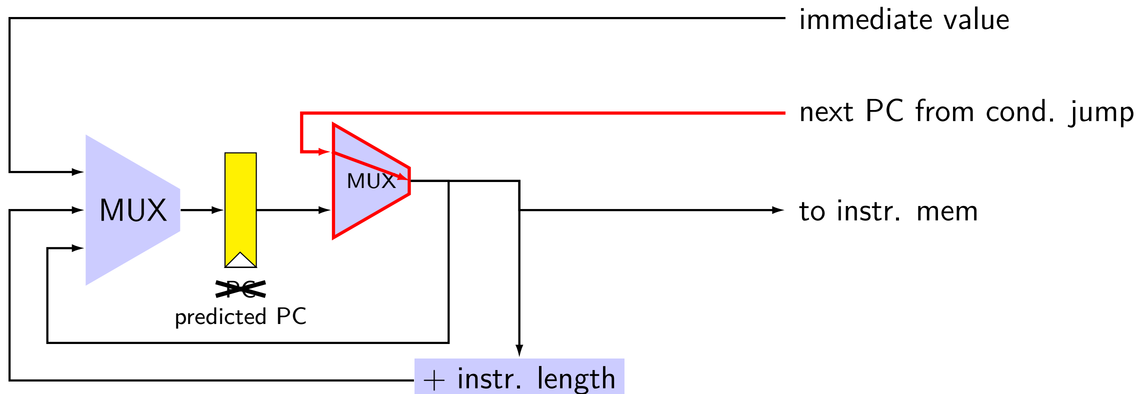
building the PC update (one possibility)

(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)

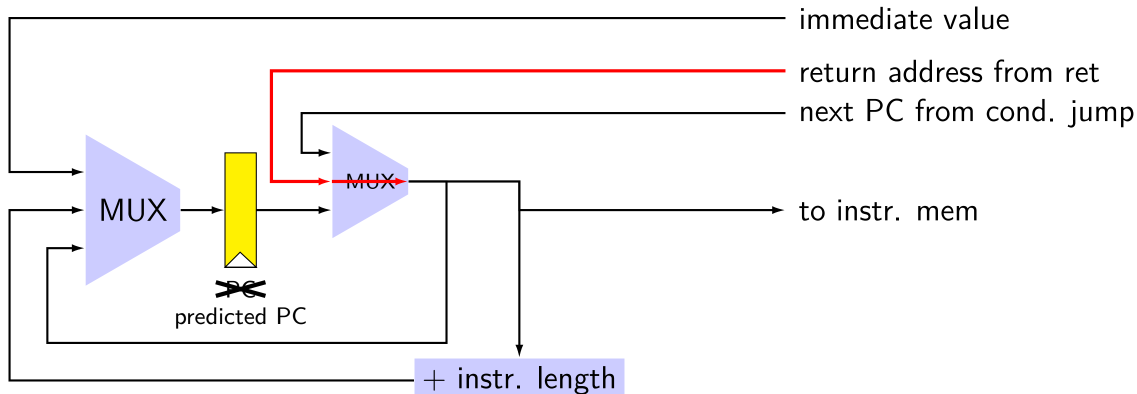
(1) normal case: PC ← PC + instr len

(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)

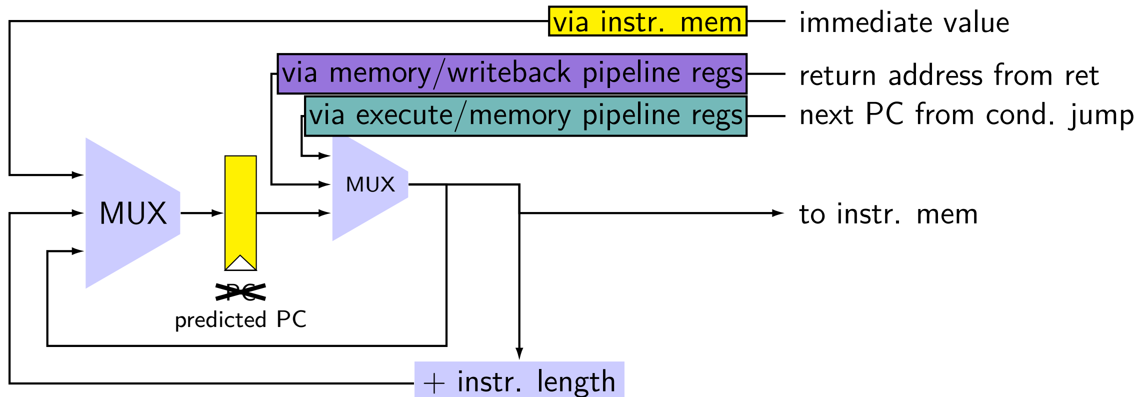
(1) normal case: PC $\leftarrow$ PC + instr len

(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

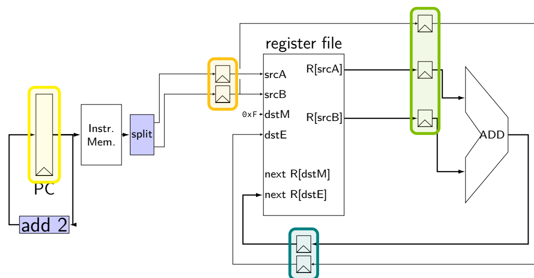
- (fetch next instruction after ret finishes memory)

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

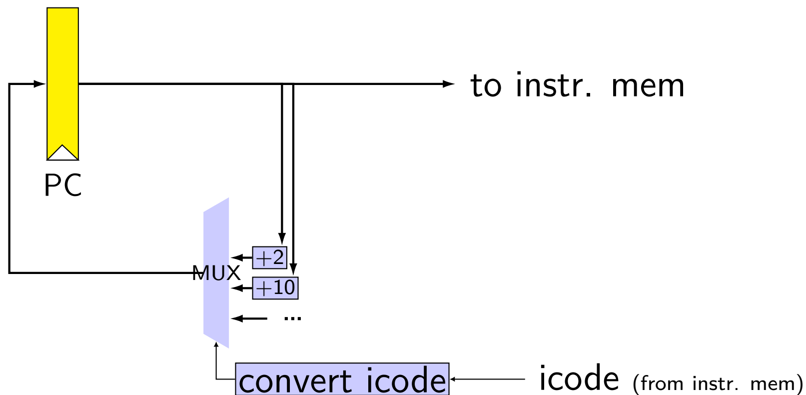
```
            call empty
            addq %r8, %r9

empty:      ret
```

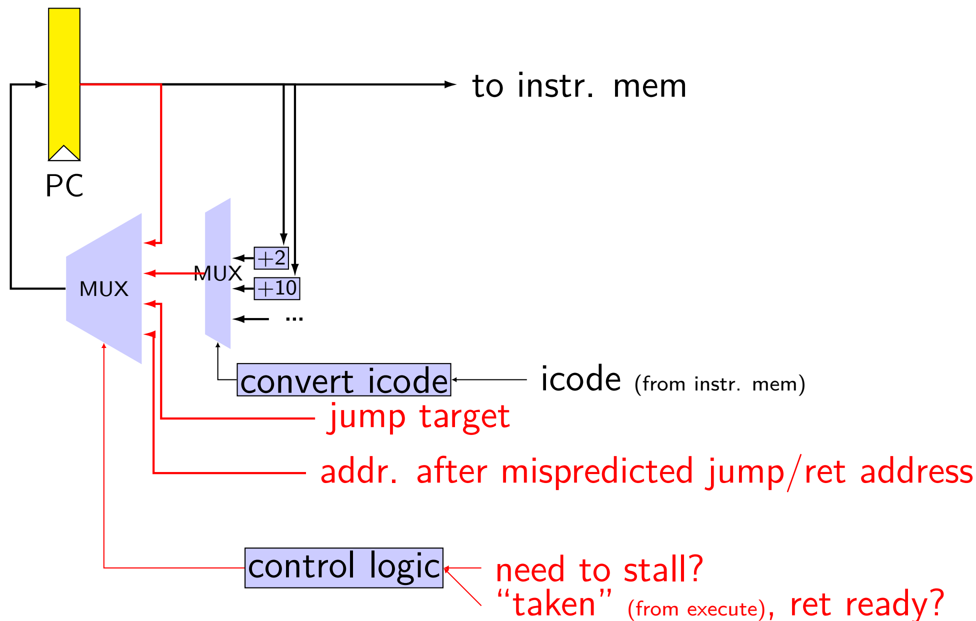
time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	call (store)	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq	nothing	nothing	nothing	ret

return address loaded here

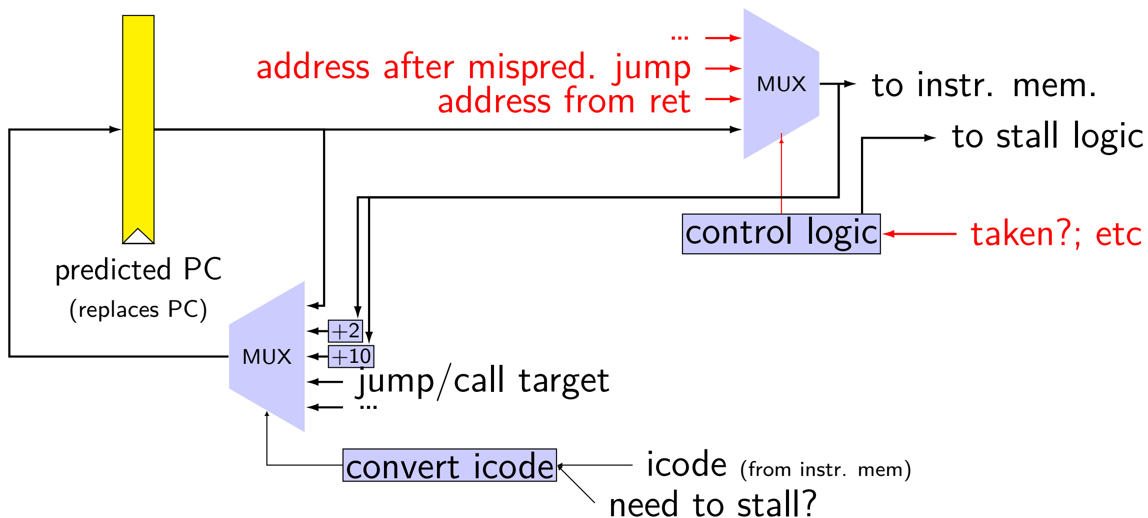
PC update (adding prediction, stall)



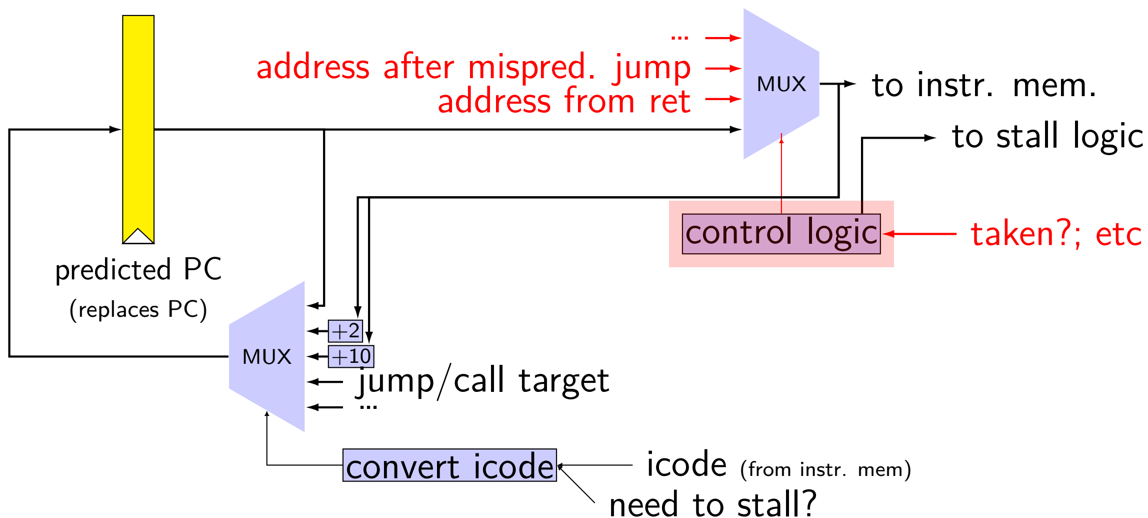
PC update (adding prediction, stall)



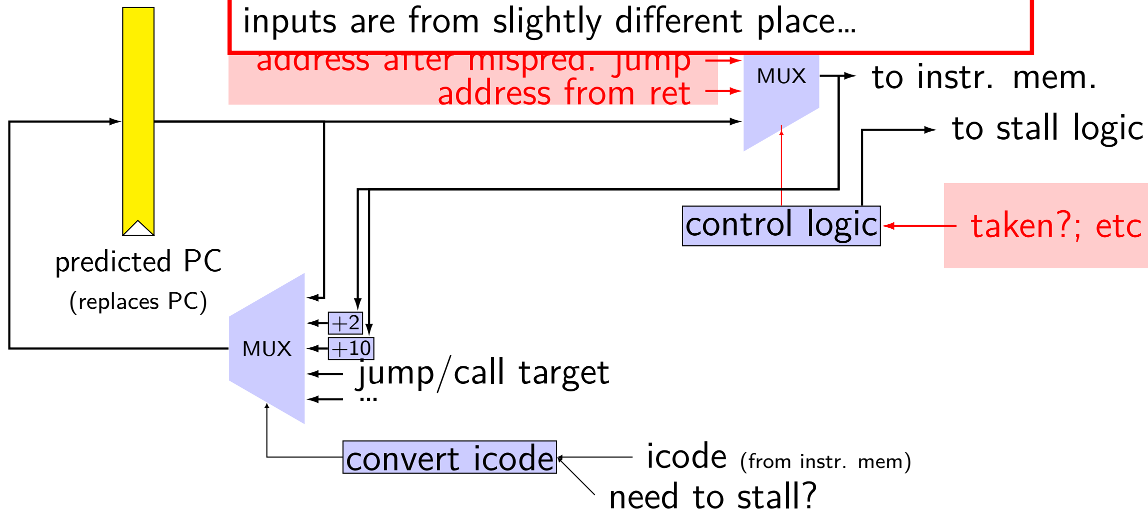
PC update (rearranged)



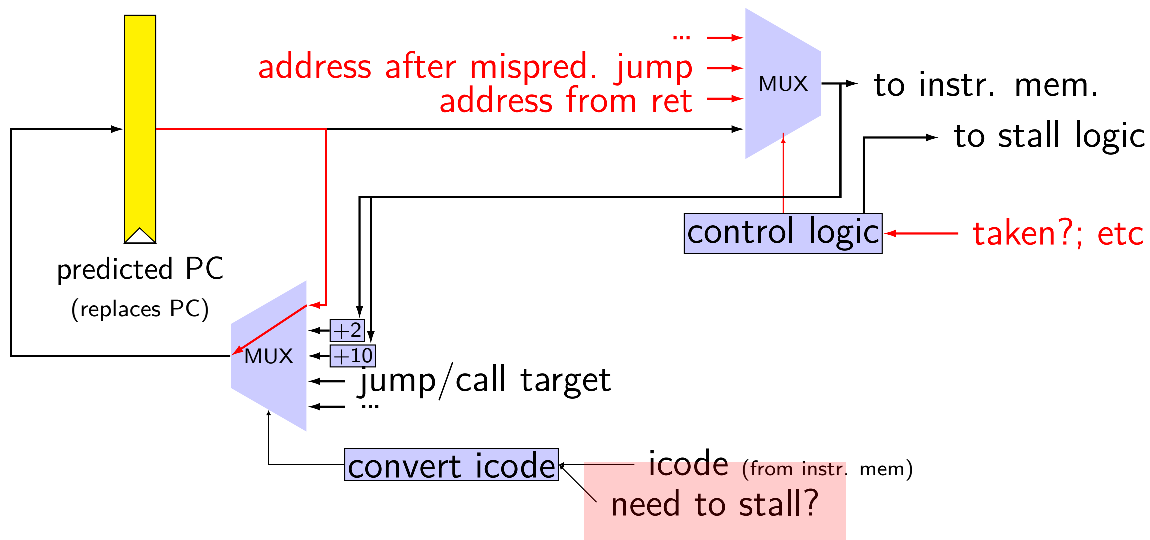
PC update (rearranged)



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;
];
```

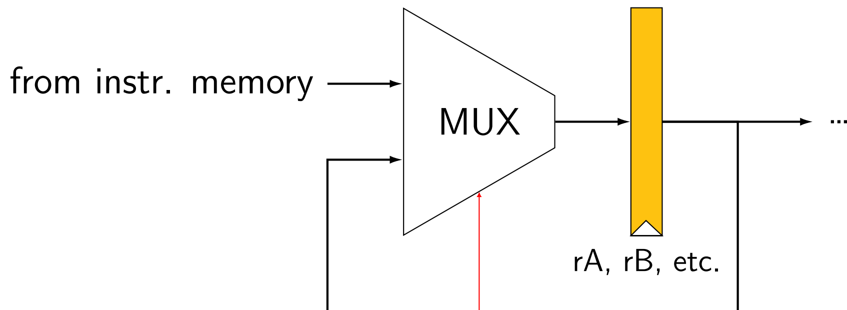
why rearrange PC update?

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

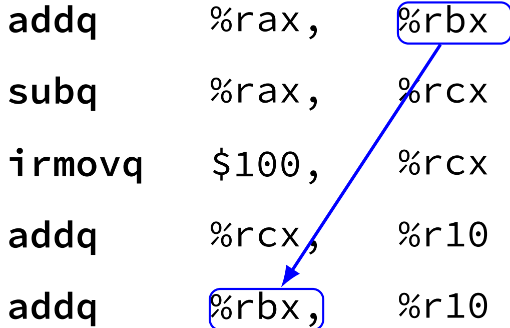


ex.: dependencies and hazards (1)

```
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)



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?

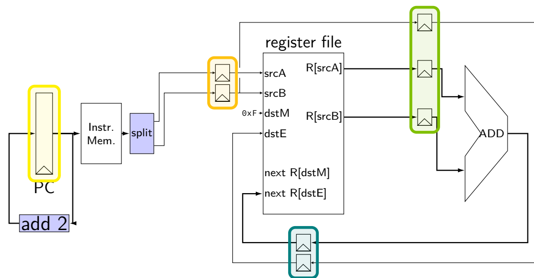
ex.: dependencies and hazards (1)

```
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



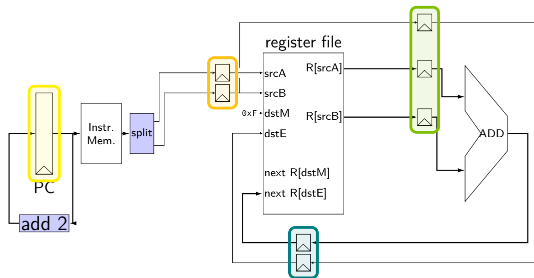
pipeline register delay: 10ps

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

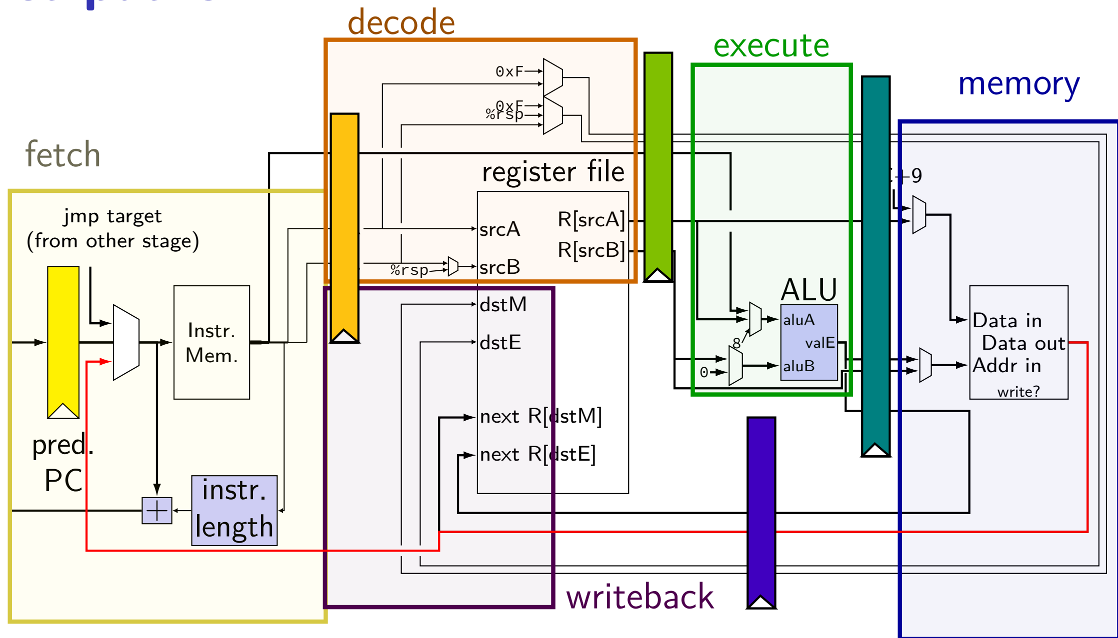


pipeline register delay: 10ps

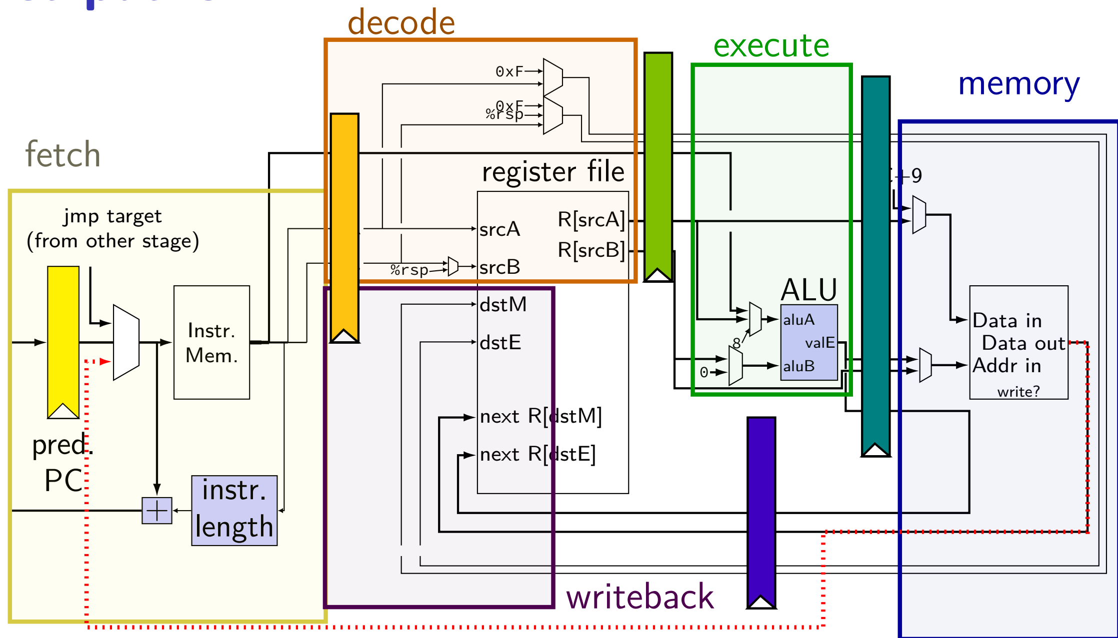
how will throughput improve if we double the speed of the instruction memory?

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

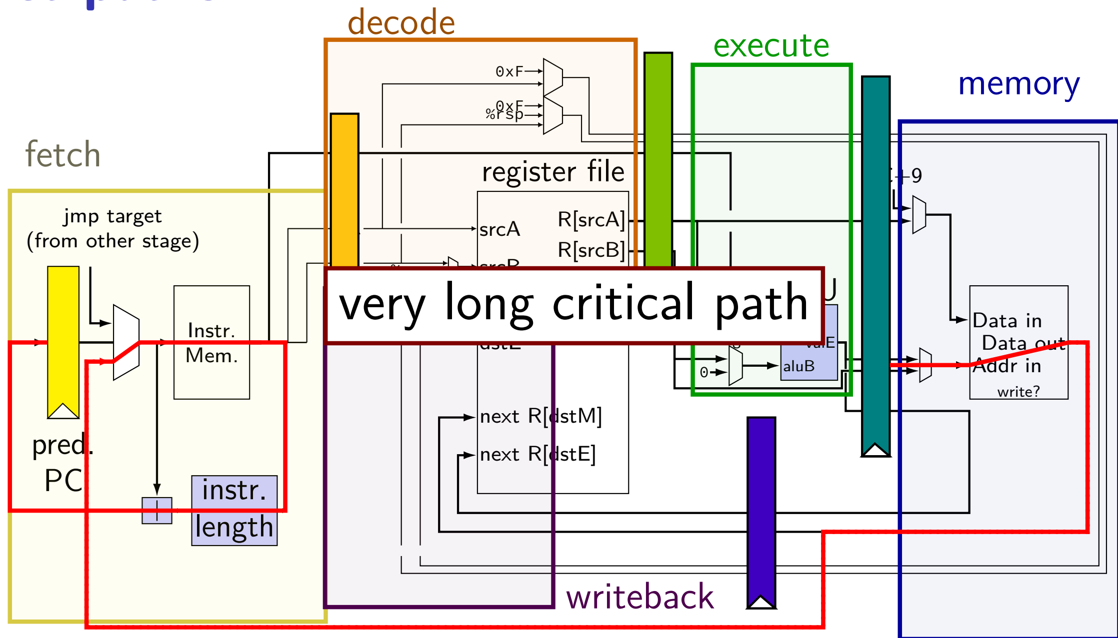
ret paths



ret paths



ret paths



backup slides

backup slides