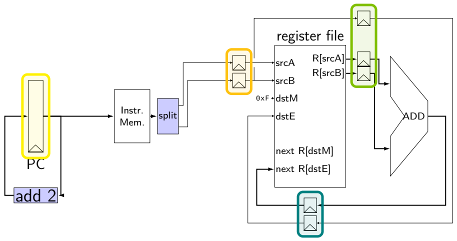
Changelog

Changes made in this version not seen in first lecture:

- 10 October 2017: remove duplication of stall logic slides
- 10 October 2017: slide 6: use d_dstE and reg_dstE instead of dstE, use P_pc and p_pc consistently
- 10 October 2017: move pipeline stages slide after mention of the stall for ret
- 10 October 2017: slide 8: show version without moved wires for dstE/dstM
- 10 October 2017: slide 16: add valA and valB, not valB and valB

addq processor timing

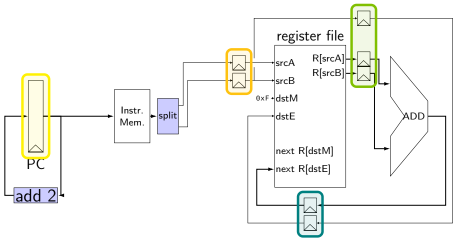
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

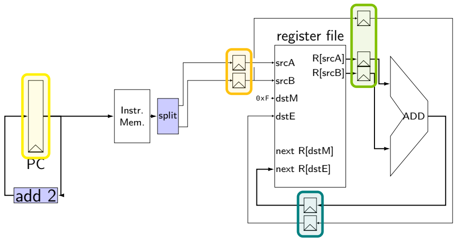
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

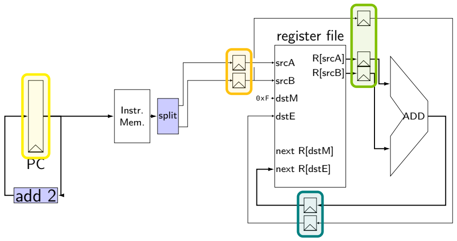
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

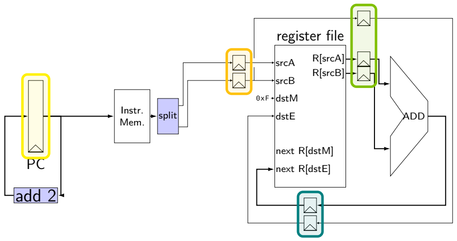
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

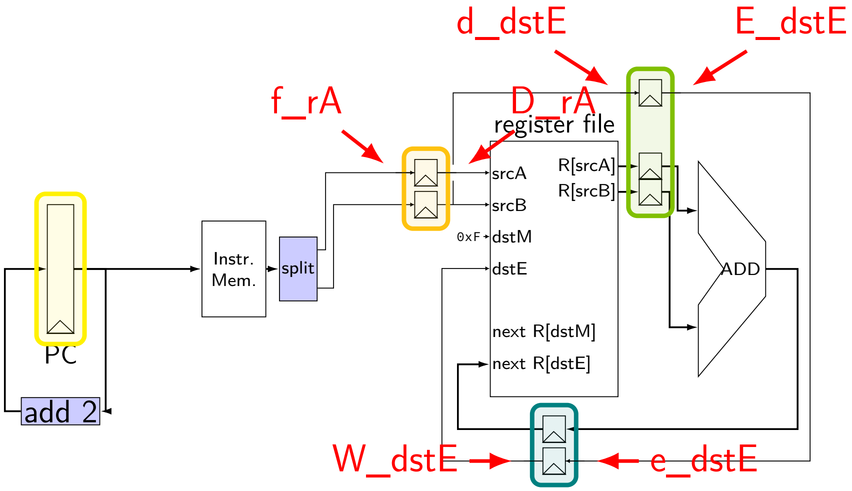
addq processor timing

```
// 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

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[12..16];

/* 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;
```

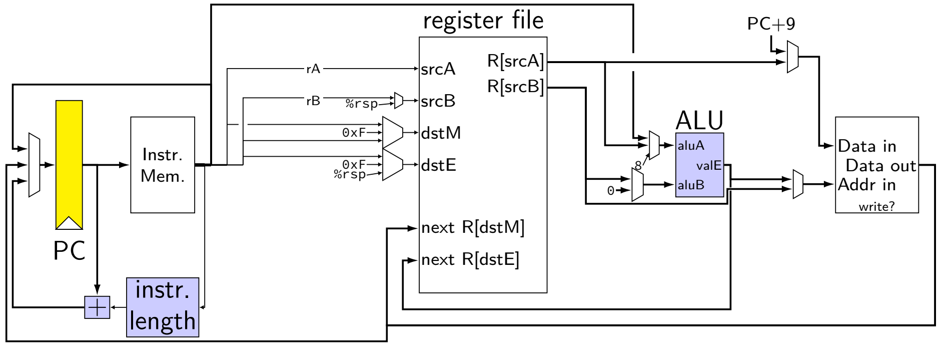
pipelined

```
/* 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;
```

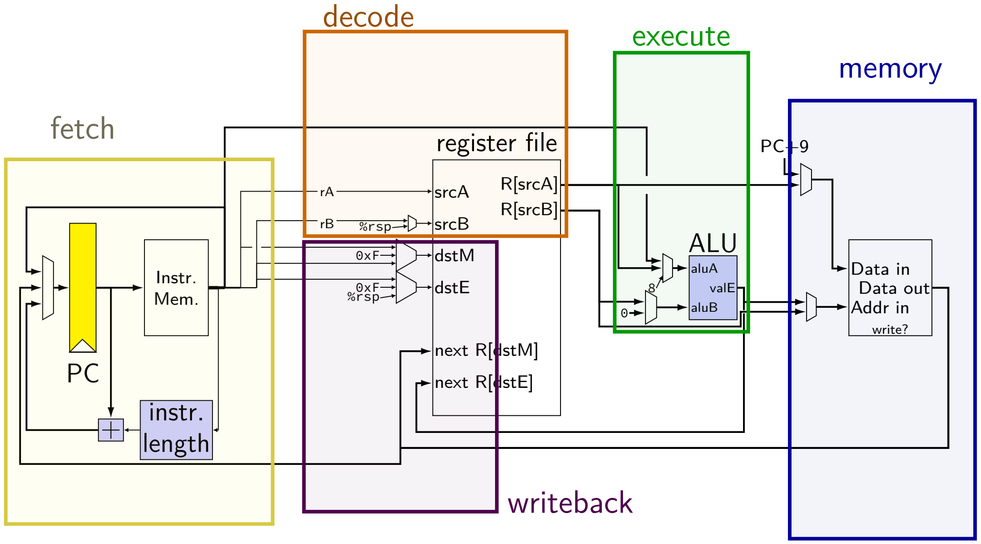
addq pipeline registers

```
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 = E; dstE : 4 = REG_NONE;
}
/* Execute */
register eW {
    valE : 64 = 0; dstE : 4 = REG_NONE;
}
/* Writeback */
```

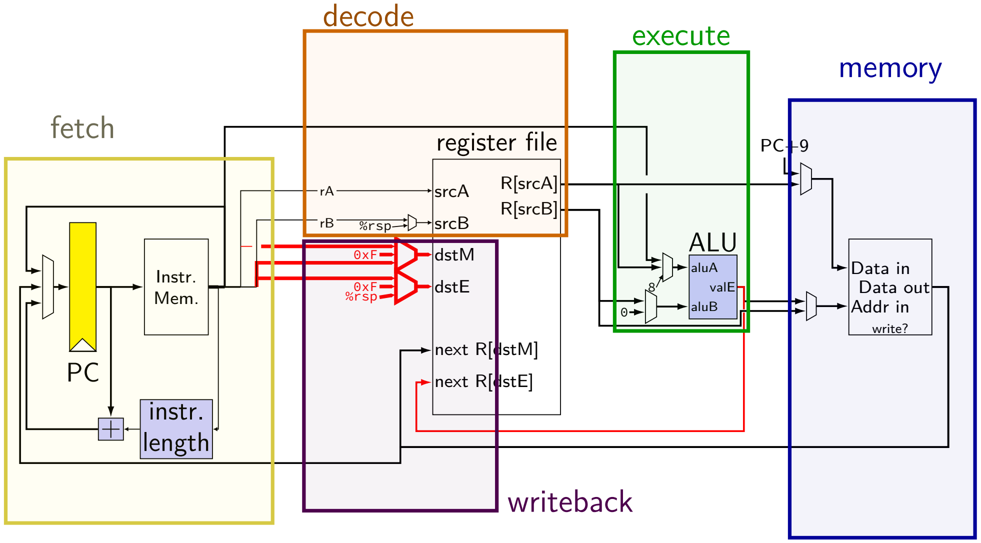
SEQ without stages



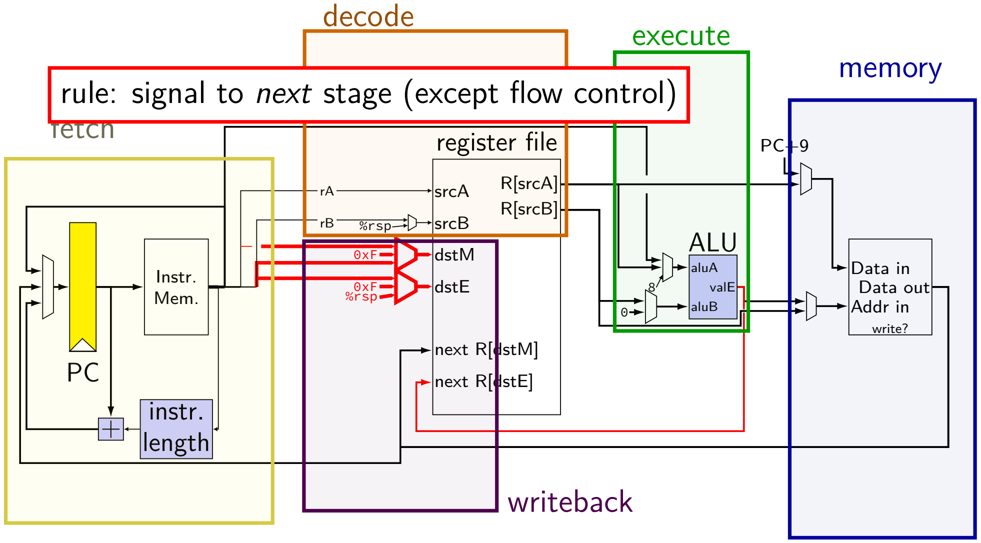
SEQ with stages



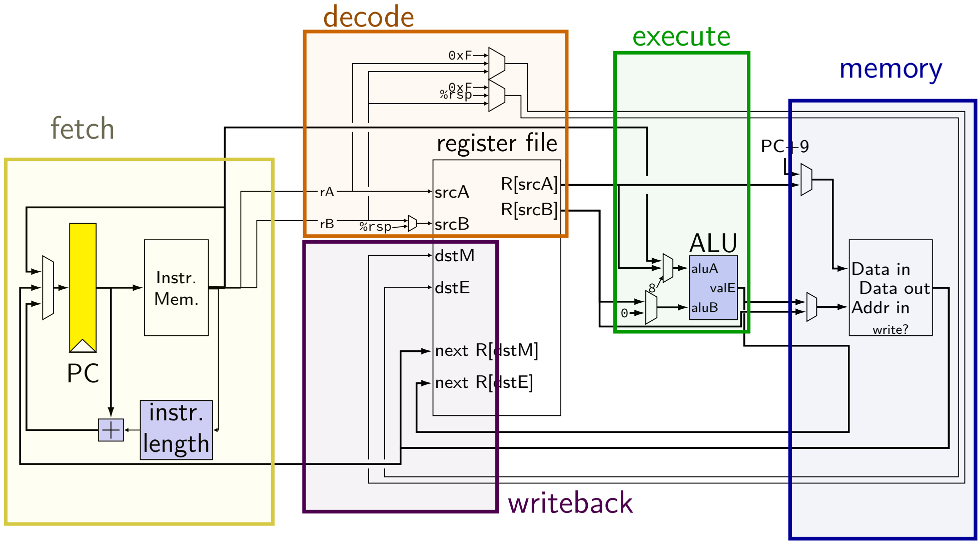
SEQ with stages



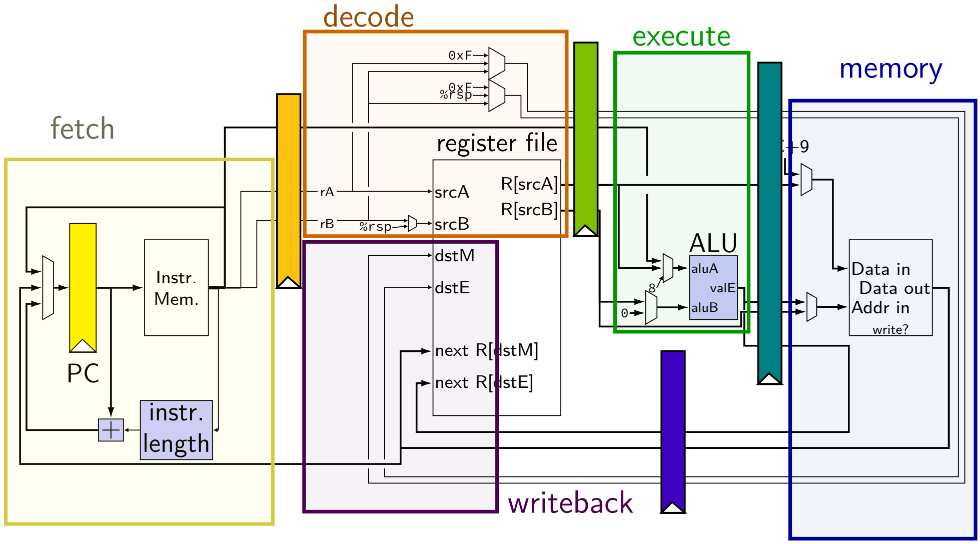
SEQ with stages


rule: signal to *next* stage (except flow control)

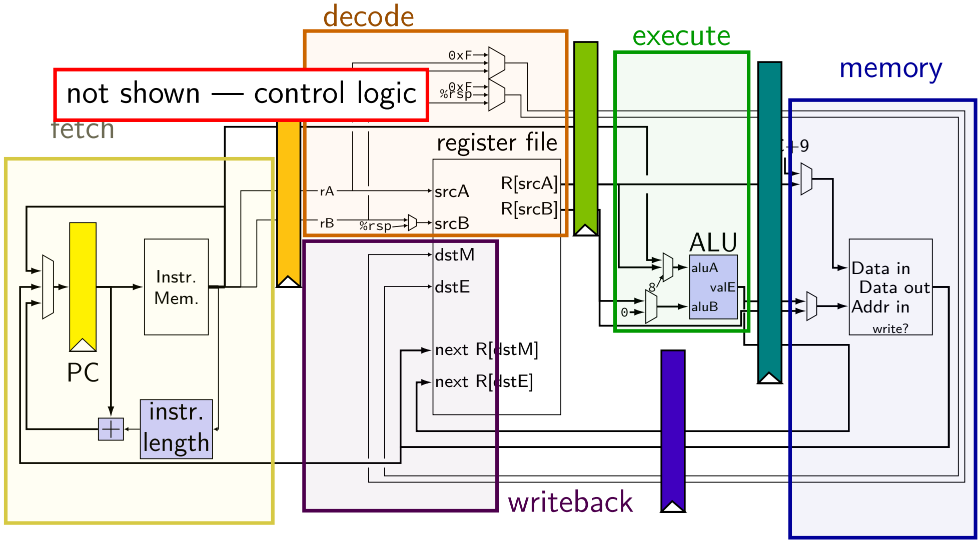
SEQ with stages (actually sequential)



adding pipeline registers



adding pipeline registers



not shown — control logic

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

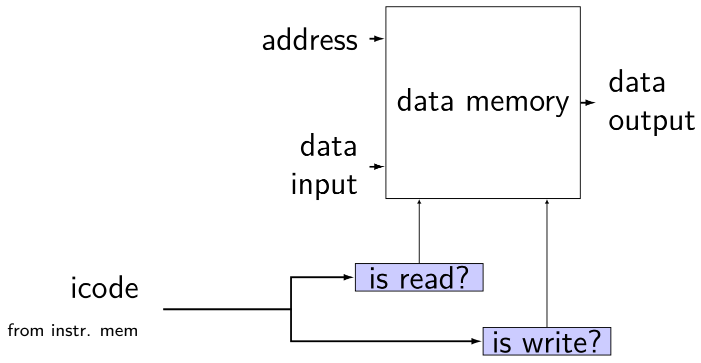
e.g. decode: send to execute via pipeline registers (d_icode, ...)

exceptions: deliberate sharing between instructions

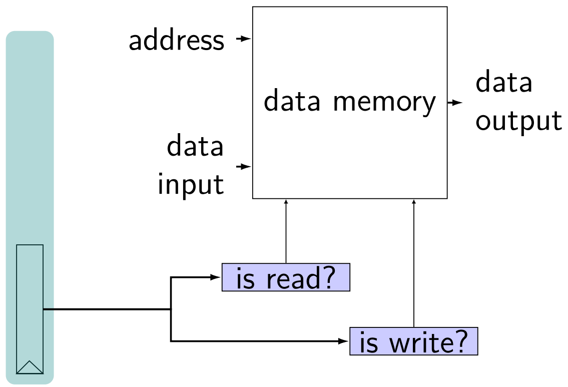
via register file/memory/etc.

via control flow instructions

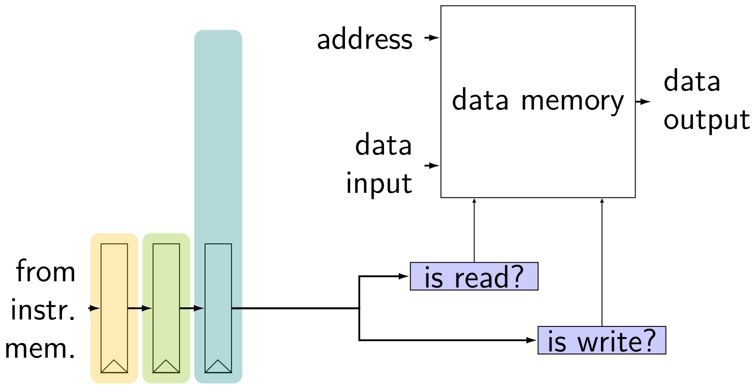
memory read/write logic



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

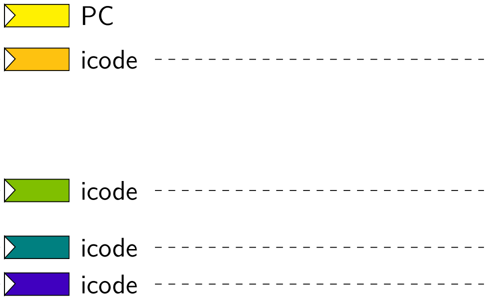
```
f_icode = i10bytes[4..8];
register fD { /* and dE and eM and mW */
    icode : 4 = NOP;
}
d_icode = D_icode
...
e_icode = E_icode;
mem_readbit = [
    M_icode == MRMOVQ || ...: 1;
    0;
];
```

addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun ← M_1[PC] rA : rB ← M_1[PC+1] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← R[rB]

execute	valE ← valA + valB
memory	
write back	R[rB] ← valE

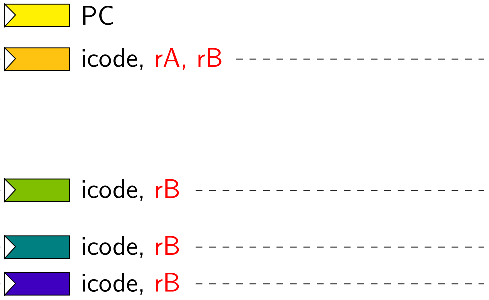


addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun ← M_1[PC] rA : rB ← M_1[PC+1] valP ← PC + 2
PC update	PC ← valP

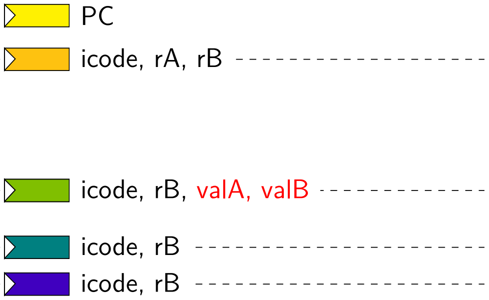
decode	valA ← R[rA]
	valB ← R[rB]

execute	valE ← valA + valB
memory	
write back	R[rB] ← valE



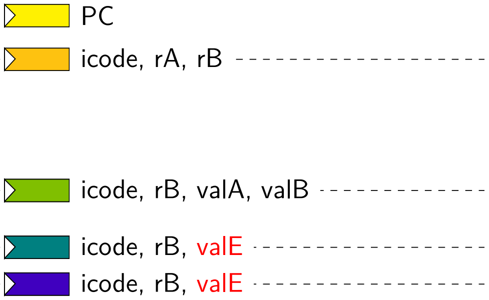
addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] valP $\leftarrow$ PC + 2
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$
execute	valE $\leftarrow$ valA + valB
memory	
write back	$R[$ rB $]$ $\leftarrow$ valE



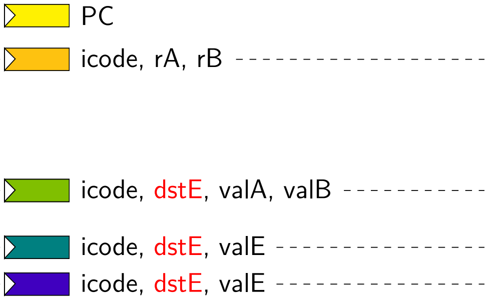
addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] valP $\leftarrow$ PC + 2
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$
execute	valE $\leftarrow$ valA + valB
memory	
write back	$R[$ rB $]$ $\leftarrow$ valE



addq pipeline registers

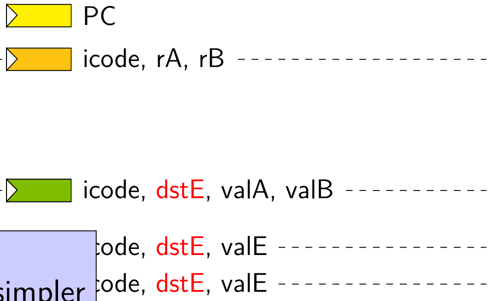
stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] valP $\leftarrow$ PC + 2
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$ dstE $\leftarrow$ rB
execute	valE $\leftarrow$ valA + valB
memory	
write back	$R[$ dstE $]$ $\leftarrow$ valE



addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] valP $\leftarrow$ PC + 2
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$ dstE $\leftarrow$ rB
execute	valE $\leftarrow$ valA + valB
memory	
write back	$R[$ dstE $]$ $\leftarrow$ valE

redundant with rB + icode
but will make implementation simpler

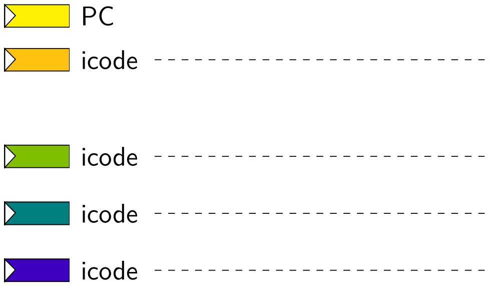


pushq pipeline registers

stage	pushq rA
fetch	icode : ifun ← M_1[PC] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← $R[\%rsp]$

execute	valE ← valB − 8
memory	M[valE] ← valA
write back	$R[\%rsp]$ ← valE



PC

icode

icode

icode

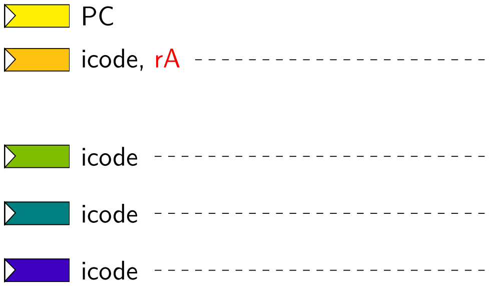
icode

pushq pipeline registers

stage	pushq rA
fetch	icode : ifun ← M_1[PC] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← $R[\%rsp]$

execute	valE ← valB − 8
memory	M[valE] ← valA
write back	$R[\%rsp]$ ← valE



PC

icode, rA

icode

icode

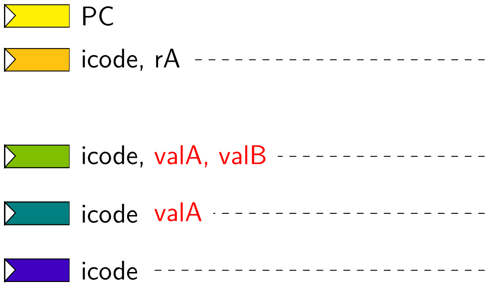
icode

pushq pipeline registers

stage	pushq rA
fetch	icode : ifun ← M_1[PC] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← $R[\%rsp]$

execute	valE ← valB − 8
memory	M[valE] ← valA
write back	$R[\%rsp]$ ← valE



PC

icode, rA

icode, valA, valB

icode valA

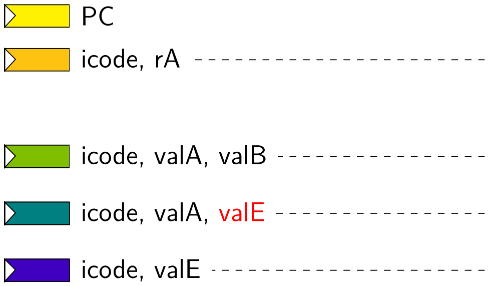
icode

pushq pipeline registers

stage	pushq rA
fetch	icode : ifun ← M_1[PC] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← $R[\%rsp]$

execute	valE ← valB − 8
memory	M[valE] ← valA
write back	$R[\%rsp]$ ← valE



PC

icode, rA

icode, valA, valB

icode, valA, valE

icode, valE

pushq pipeline registers

stage	pushq rA
fetch	icode : ifun ← M_1[PC] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← $R[\%rsp]$
	dstE ← $\%rsp$

execute	valE ← valB − 8
memory	M[valE] ← valA
write back	R[dstE] ← valE

🟨 PC

🟧 icode, rA

🟩 icode, valA, valB, dstE

🟦 icode, valA, valE, dstE

🟪 icode, valE, dstE

pushq pipeline registers

stage	pushq rA
fetch	icode : ifun ← M_1[PC] valP ← PC + 2
PC update	PC ← valP

decode	valA ← R[rA]
	valB ← $R[\%rsp]$
	dstE ← $\%rsp$

execute	valE ← va B − 8
me	
write back	R[dstE] ← valE

🟨 PC

🟧 icode, rA

🟩 icode, valA, valB, dstE

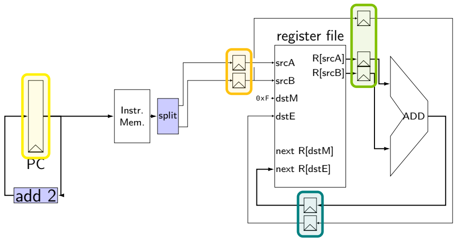
ode, valA, valE, dstE

ode, valE, dstE

redundant with icode
but will make implementation simpler

addq processor: data hazard

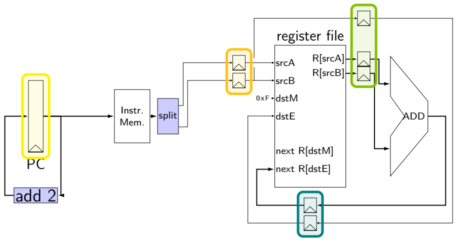
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```



	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

usually not acceptable

data hazard hardware solution

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

how about hardware add nops?

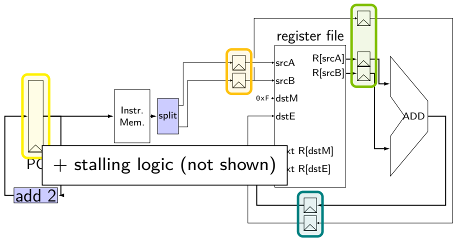
called stalling

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
```

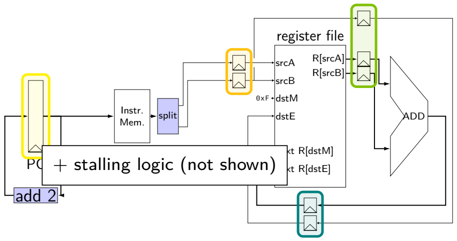


+ stalling logic (not shown)

	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

addq processor: data hazard stall

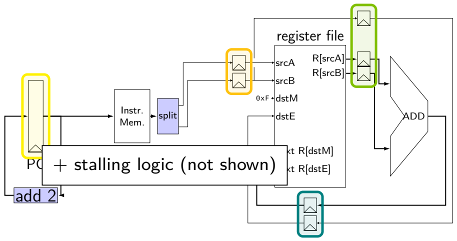
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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

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”

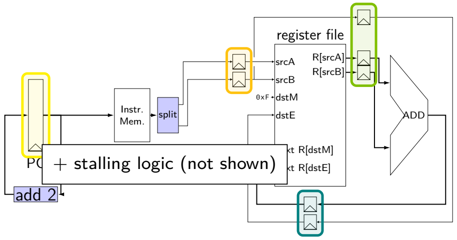
addq stall (alternative)

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

addq processor: data hazard stall (alternative)

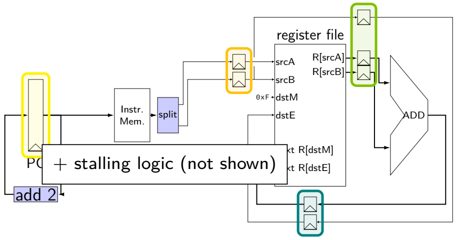
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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	0x4*	9*	8*	800	900	9		
3	0x4*	9*	8*	---	---	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 (alternative)

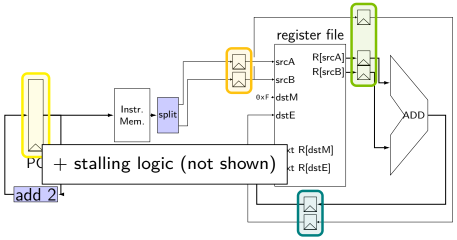
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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	0x4*	9*	8*	800	900	9		
3	0x4*	9*	8*	---	---	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 (alternative)

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

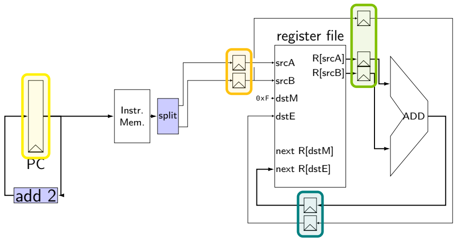


	fetch	fetch→decode		decode→execute			execute→writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4*	9*	8*	800	900	9		
3	0x4*	9*	8*	---	---	F*	1700	9
4	0x4	9	8	---	---	F*	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8

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

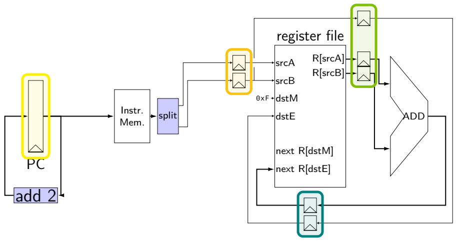
hazard exericse

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



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

hazard exericse solution



cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	W					
`addq %r10, %11`		F	D	E	W				
`addq %r9, %r8`			✗	F	D	E	W		
`addq %r11, %r10`					F	D	E	W	

exercise: pipelining improvement (1)

1% of instructions executed need to stall 4 cycles for hazard

2% stall exactly 3

10% stall exactly 2

15% stall exactly 1

how many cycles per instruction? (compute the mean)

exercise: pipelining improvement (1)

1% of instructions executed need to stall 4 cycles for hazard

2% stall exactly 3

10% stall exactly 2

15% stall exactly 1

how many cycles per instruction? (compute the mean)

$1 + .15 \times 1 + .10 \times 2 + .02 \times 3 + .01 \times 4 = 1.45$

exercise: pipelining improvement (2)

1% of instructions executed need to stall 4 cycles for hazard

2% stall exactly 3

10% stall exactly 2

15% stall exactly 1

how many cycles per instruction? 1.45

original cycle time: 1200 ps; new cycle time: 300 ps

how much better throughput?

exercise: pipelining improvement (2)

1% of instructions executed need to stall 4 cycles for hazard

2% stall exactly 3

10% stall exactly 2

15% stall exactly 1

how many cycles per instruction? 1.45

original cycle time: 1200 ps; new cycle time: 300 ps

how much better throughput?

1 every ($1.45 \times 300 = 435$ ps) versus 1 every 1200 — 2.76 faster

control hazard

```
addq %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

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

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

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

control hazard: stall

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

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

control hazard: stall

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

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

wait for two cycles for addq to update SF/ZF

control hazard: stall

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

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

execute je instruction (use SF/ZF)

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

pipeline stages

fetch — instruction memory, *most* PC computation

decode — reading register file

execute — computation, condition code read/write

memory — memory read/write

writeback — writing register file, writing Stat register

pipeline stages

fetch — instruction memory, *most* PC computation

decode — reading register file

common case: fetch next instruction in next cycle
can't for conditional jump, return

execute — computation, condition code read/write

memory — memory read/write

writeback — writing register file, writing Stat register

pipeline stages

fetch — instruction memory, *most* PC computation

decode — reading register file

execute — computation, condition code read/write

memory — memory read/write

read/write in same stage avoids reading wrong value
get value updated for prior instruction (not earlier/later)

writeback — writing register file, writing Stat register

pipeline stages

fetch — instruction memory, *most* PC computation

decode — reading register file

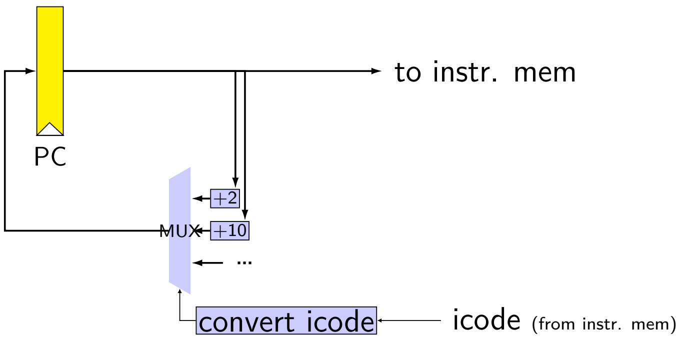
execute — computation, condition code read/write

memory — memory read/write

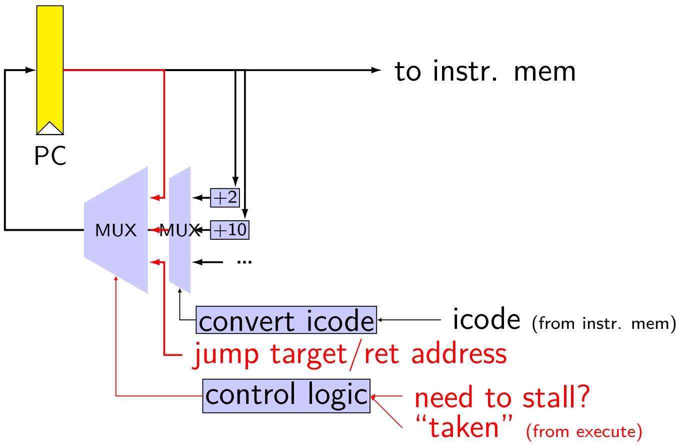
writeback — writing register file, writing Stat register

don't want to halt until everything else is done

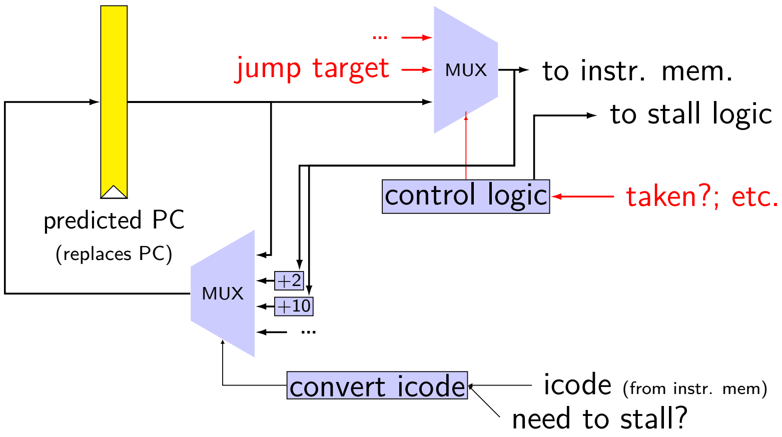
PC update (adding stall)



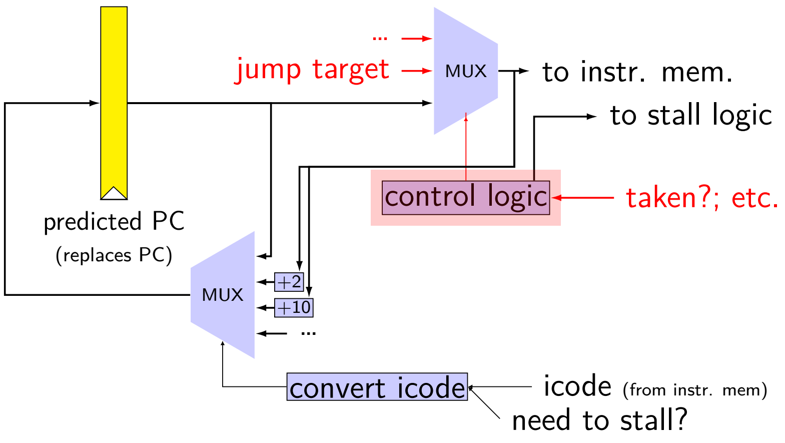
PC update (adding stall)



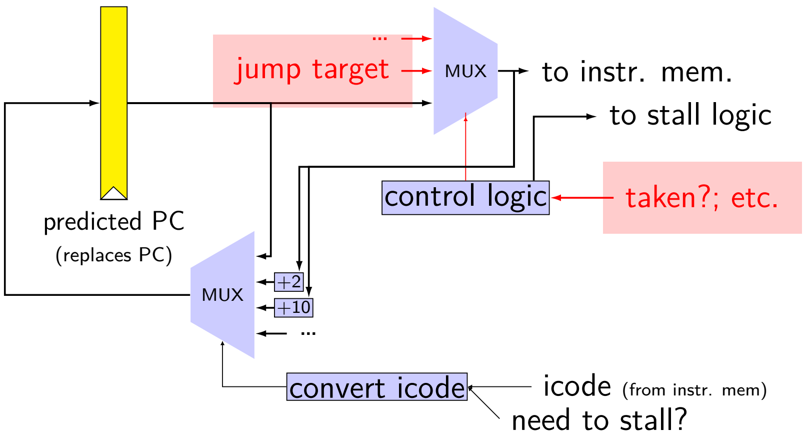
PC update (rearranged)



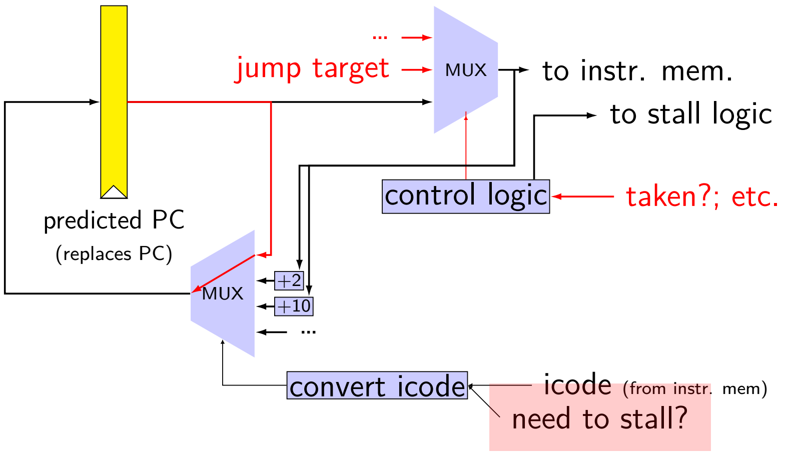
PC update (rearranged)



PC update (rearranged)



PC update (rearranged)



rearranged PC update in HCL

```
/* actual input to instruction memory */
pc = [
    conditionCodesSaidTaken : jumpTarget;
        /* from later in pipeline */
    ...
    1: P_predictedPC; /* a register, replacing PC register */
];
```

stalling for ret

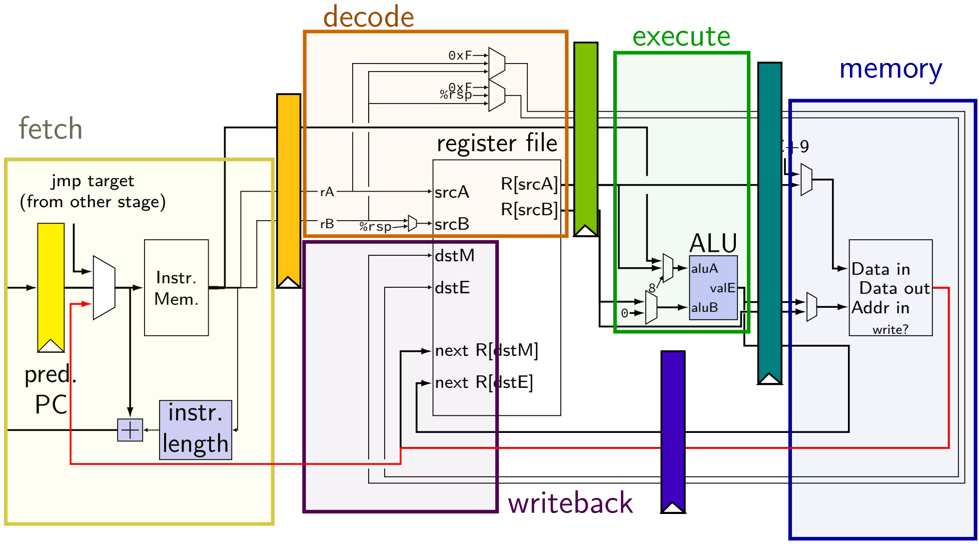
```
        call empty
        addq %r8, %r9

empty:  ret
```

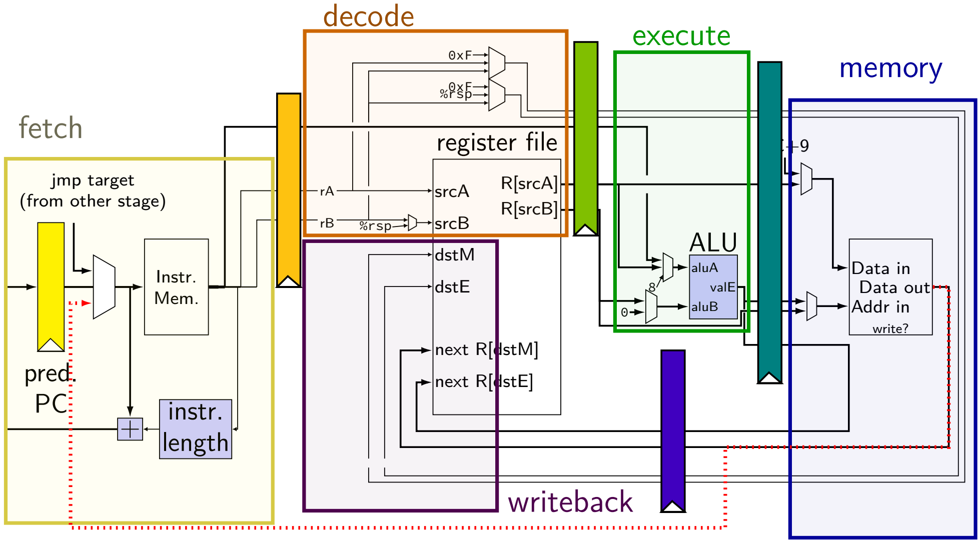
time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	call (store)	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq	nothing	nothing	nothing	ret

why not start addq here?

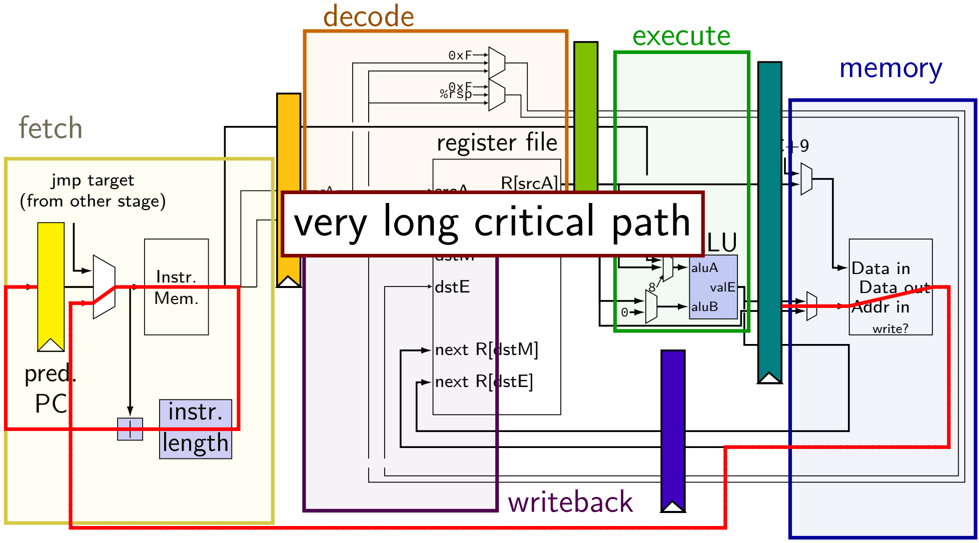
ret paths



ret paths

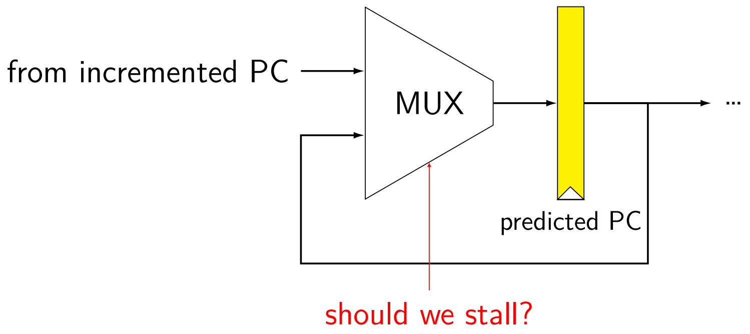


ret paths



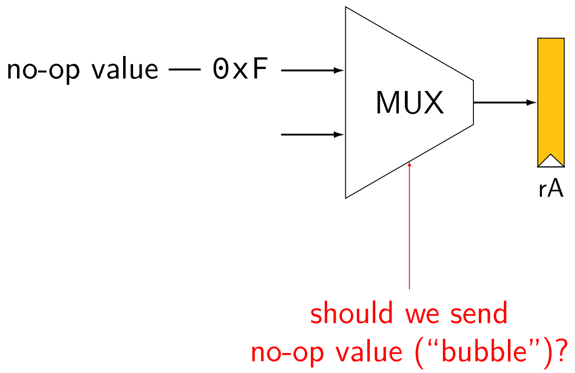
very long critical path

fetch/fetch logic — advance or not



should we stall?

fetch/decode logic — bubble or not



should we send
no-op value ("bubble")?

HCLRS signals

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

HCLRS: every register bank has these MUXes built-in

`stall_B`: keep old value for all registers

- register input → register output

`bubble_B`: use default value for all registers

- register input → default value

exercise

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

stall: keep old value
bubble: store default value

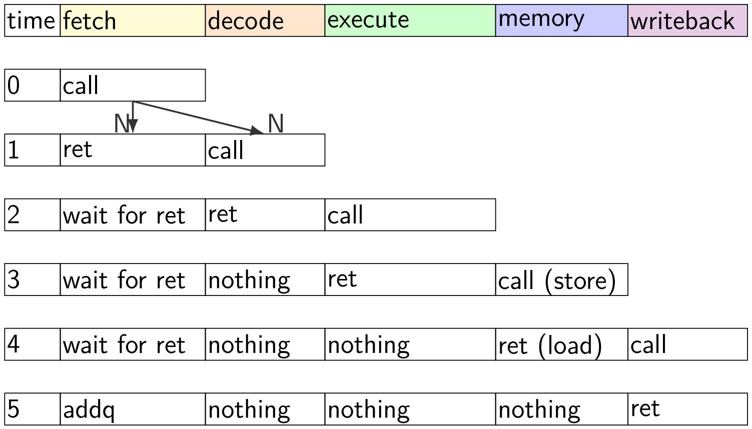
time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	???	1	0
2	0x03	???	0	0
3	0x04	???	0	1
4	0x05	???	0	0
5	0x06	???	0	0
6	0x07	???	1	0
7	0x08	???	1	0
8		???		

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		

ret stall

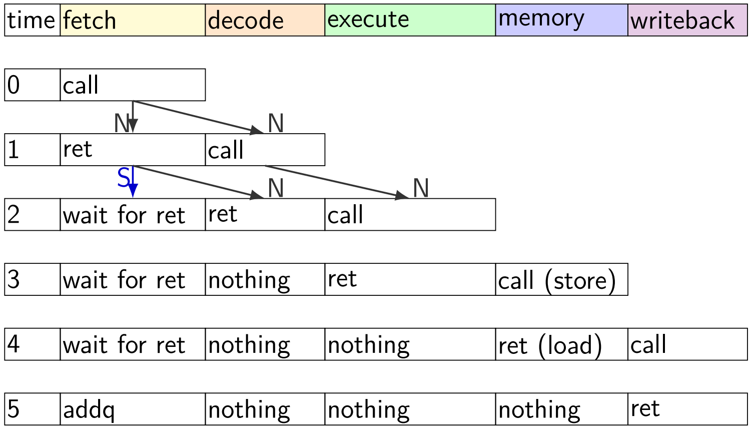


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

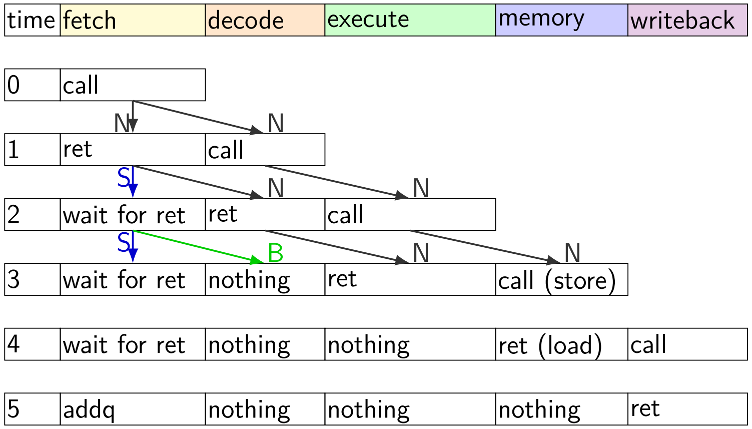


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

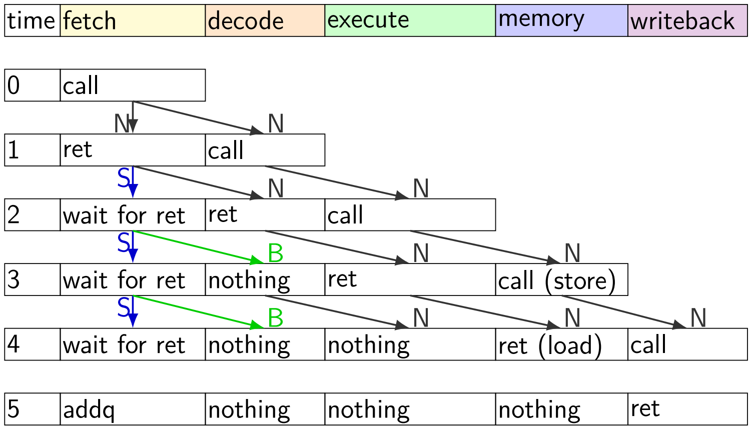


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

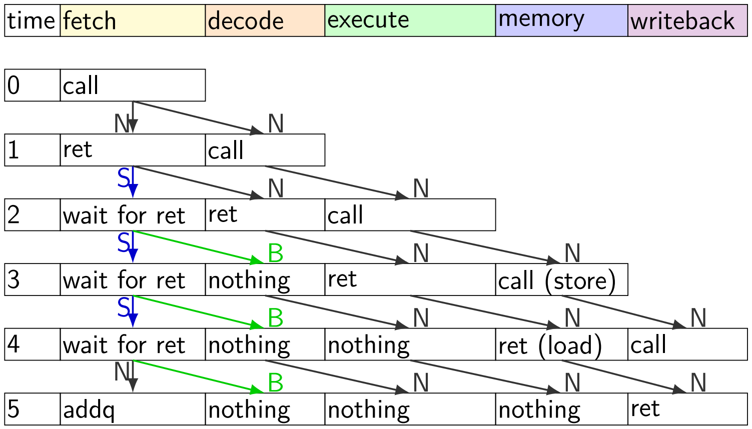


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall



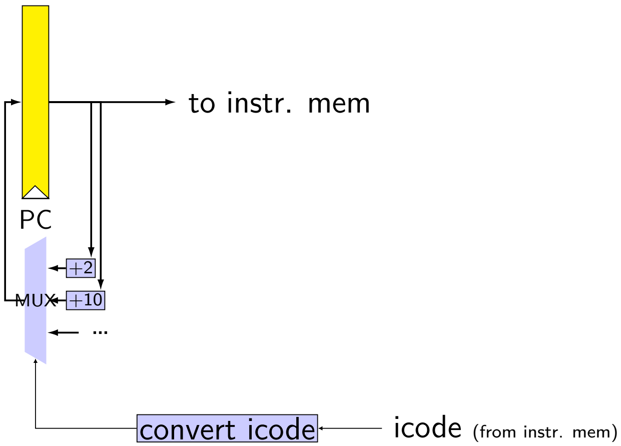
time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

backup slides

PC update from lab



PC update from lab

```
icode = i10bytes[4..8];
p_pc = [
    icode == ADD || ...: P_pc + 2;
    icode == IRMOVQ || ...: P_pc + 10;
    ...
];
```