Pipelining 3 — Hazard Handling

last time

5-stage pipeline
- fetch+PC update / decode / execute / memory / writeback

writing pipeline stages
- read from prior stage's reigsters
- write to next stage's registers
- manipulate components w/in stage (e.g. data memory for memory)
- need a value from prior stage? pass through registers

data hazard — pipeline reads wrong (old) value
- wrong order of reads/writes

control hazard — pipeline doesn't know next instrutcion

resolving hazards with stalling (insert nops)

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	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

e e

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

writeba

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

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

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

can we do better?

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data depe

can't easily read memory early
might be written in previous instruction

can we do better?

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

trick: use values waiting to get to register file

can we do better?

revisiting data hazards

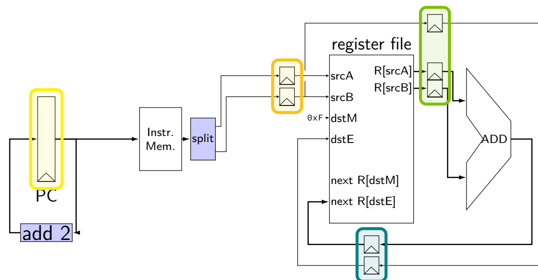
stalling worked

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

observation: value ready before it would be needed
(just not stored in a way that let's us get it)

motivation

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

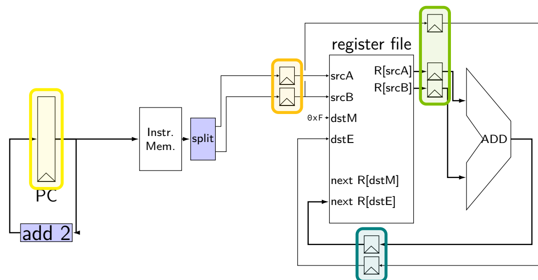


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700

motivation

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

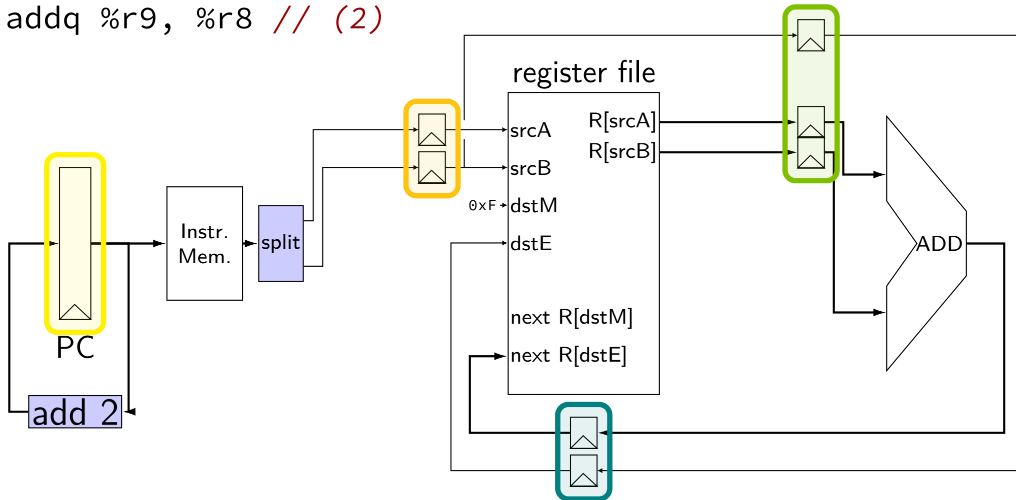


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700

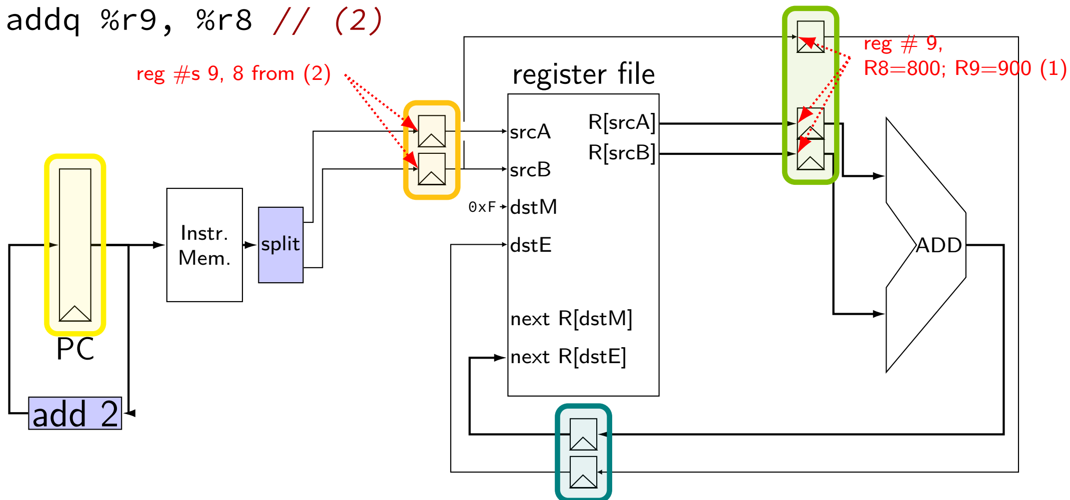
forwarding

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



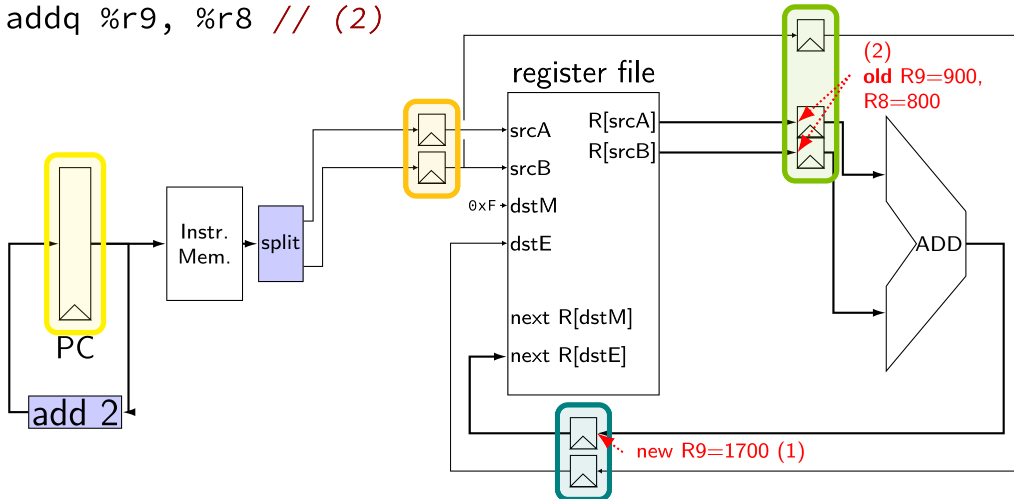
forwarding

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



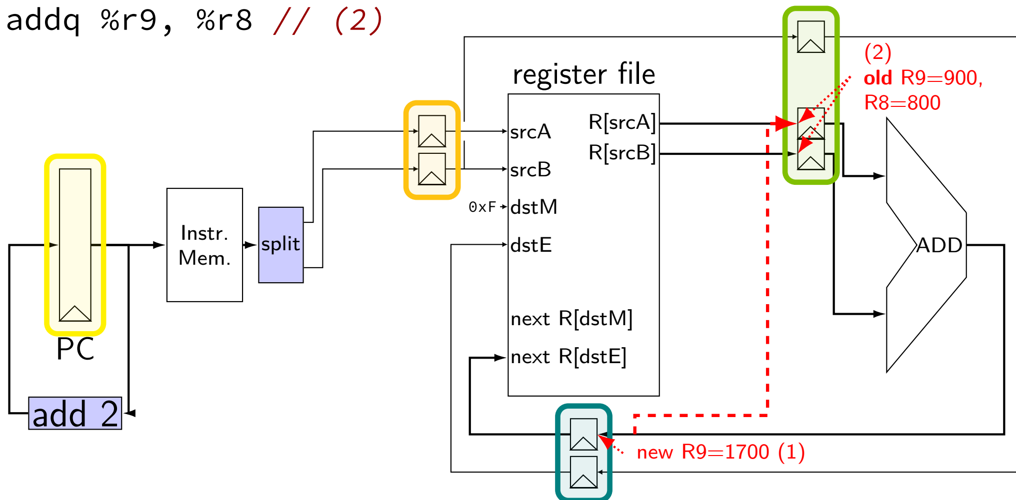
forwarding

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



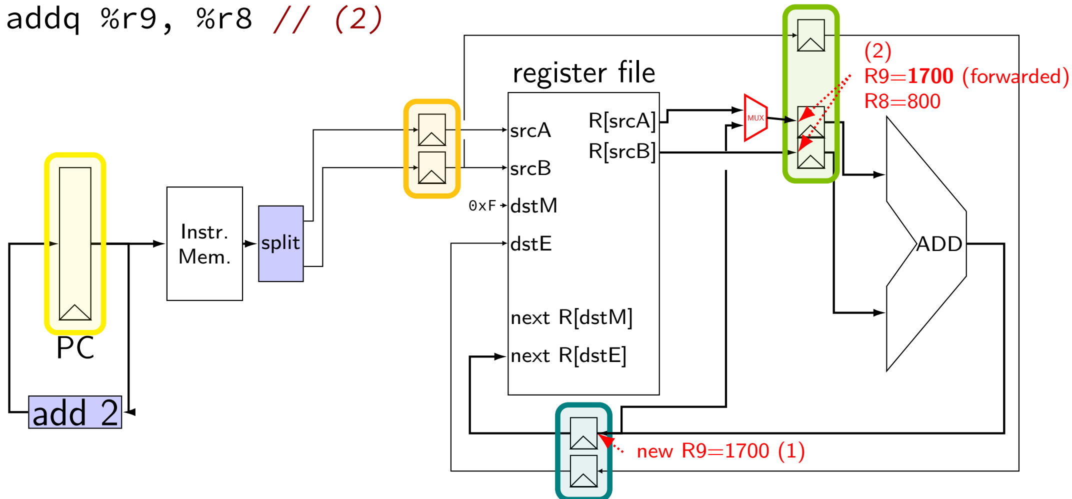
forwarding

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



forwarding

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

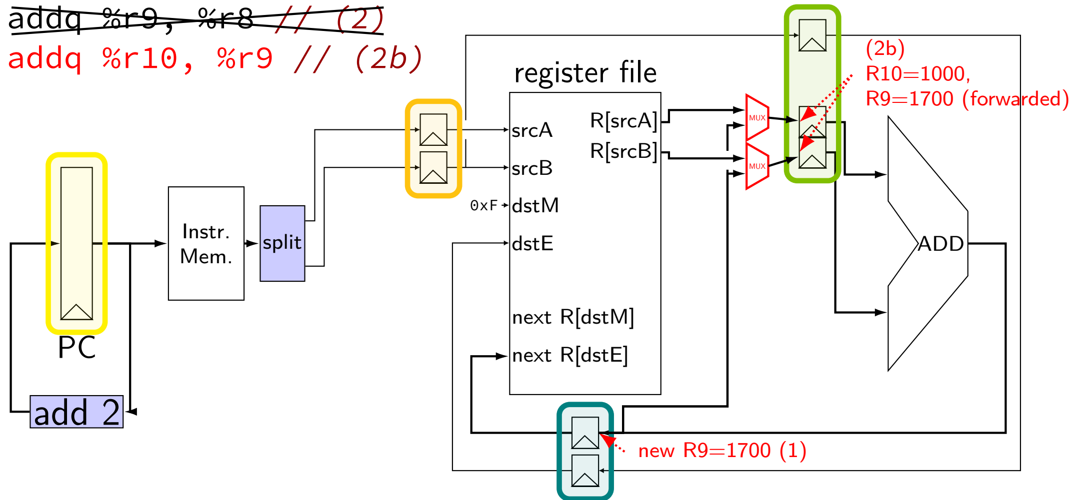


forwarding

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

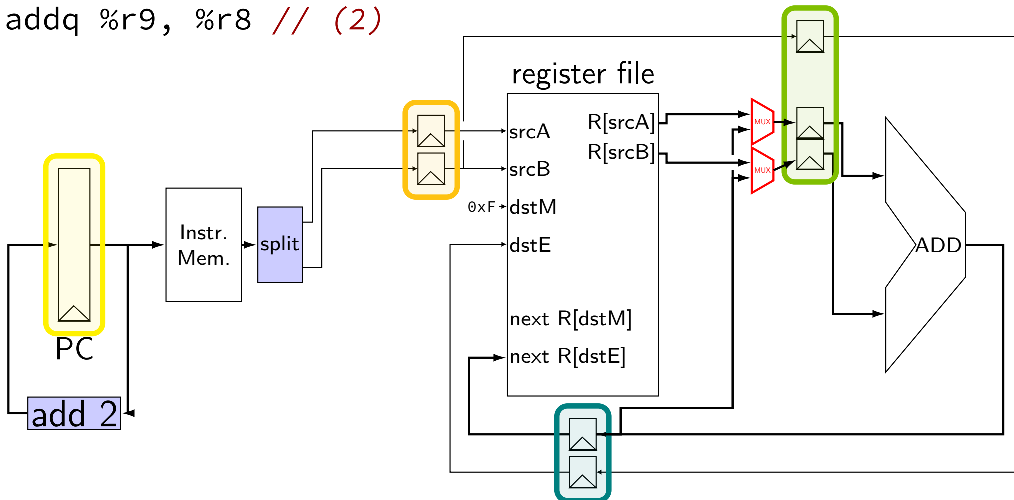
~~addq %r9, %r8 // (2)~~

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



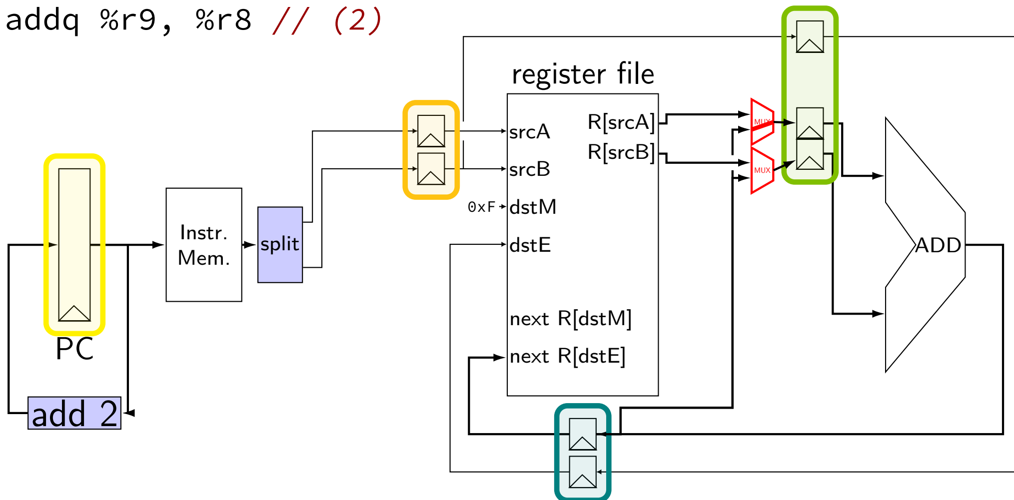
forwarding: MUX conditions

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



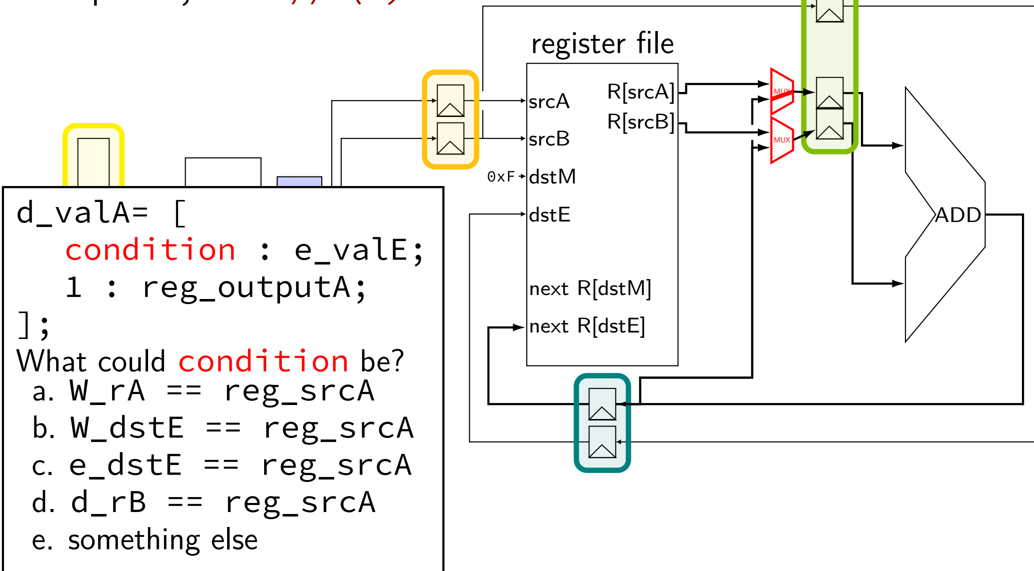
forwarding: MUX conditions

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



forwarding: MUX conditions

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



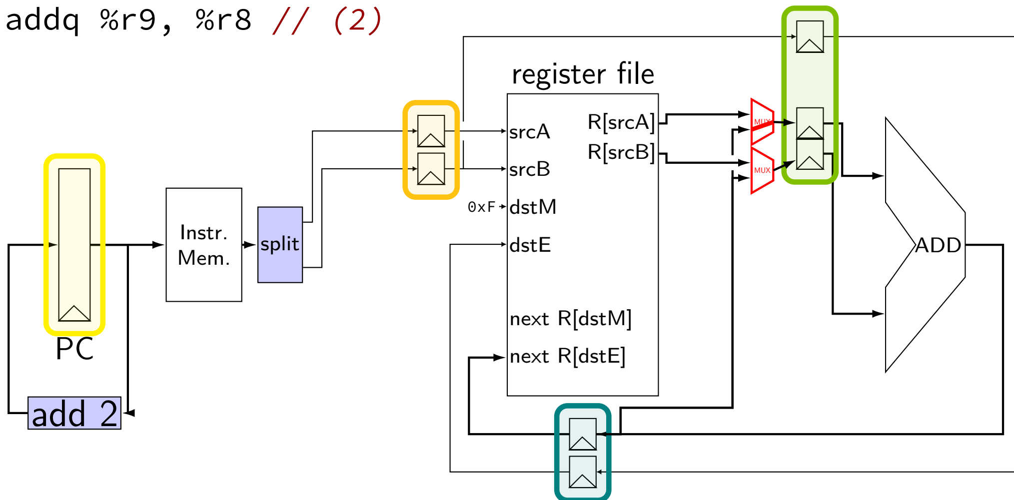

```
d_valA= [
   condition : e_valE;
   1 : reg_outputA;
];
```

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)
```



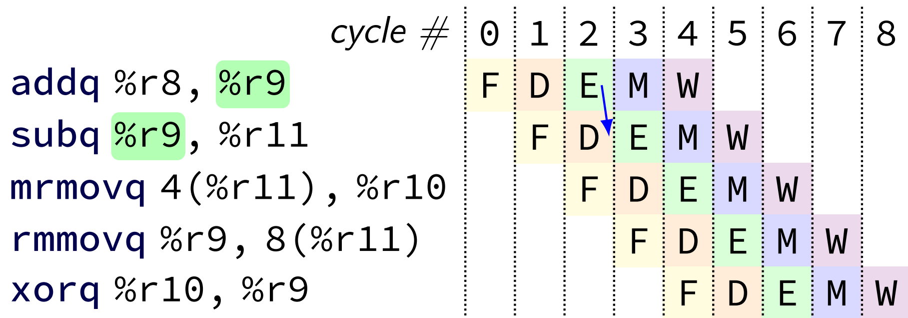
forwarding: MUX conditions

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

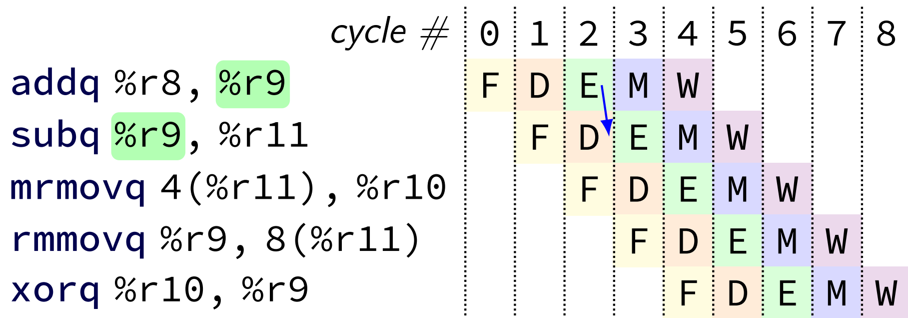
some forwarding paths

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

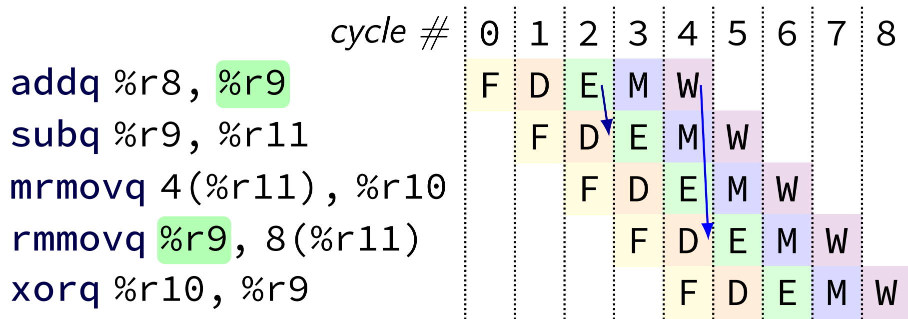
some forwarding paths



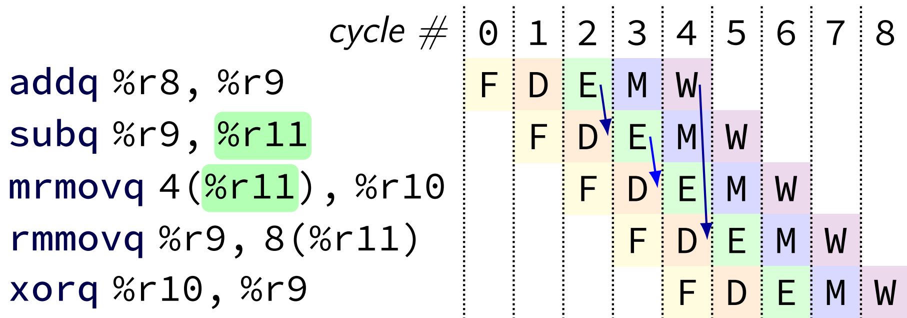
some forwarding paths



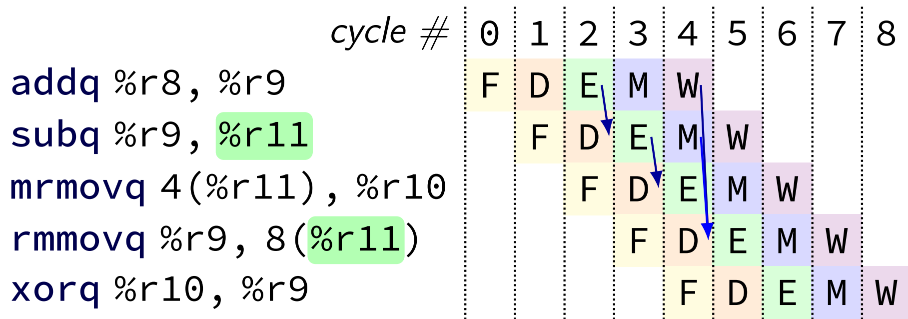
some forwarding paths



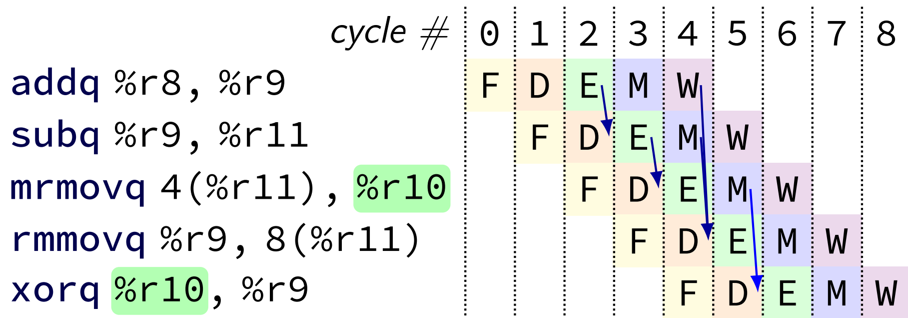
some forwarding paths



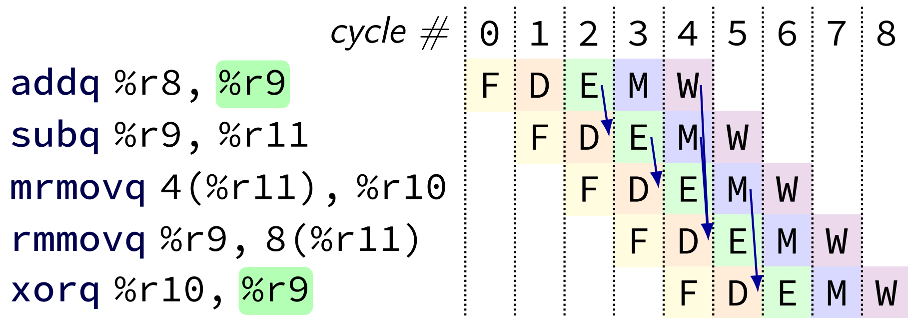
some forwarding paths



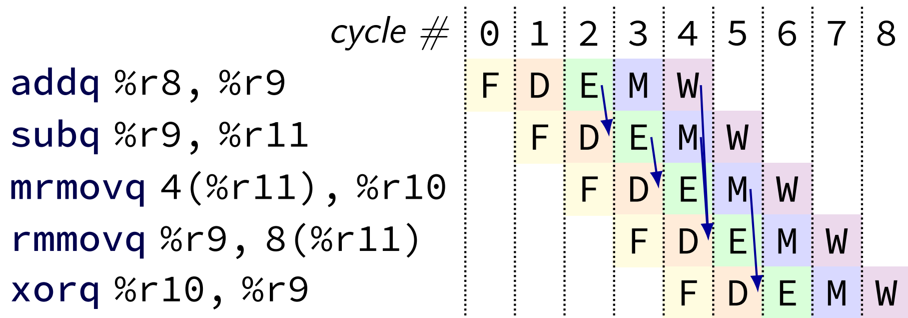
some forwarding paths



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		

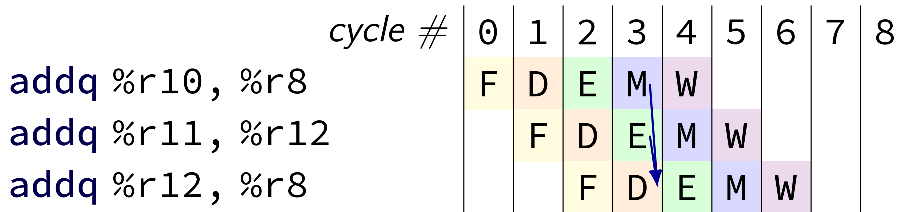
multiple forwarding paths (1)

	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		

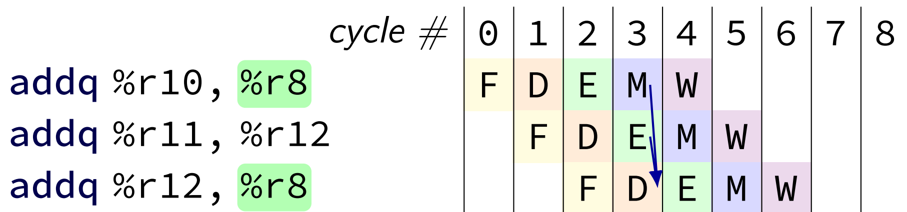
multiple forwarding HCL (1)

```
/* 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;
];
```

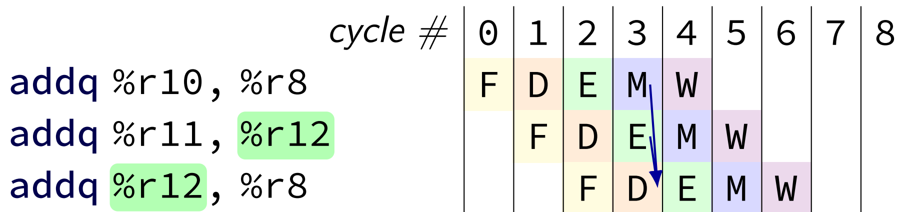
multiple forwarding paths (2)



multiple forwarding paths (2)



multiple forwarding paths (2)



multiple forwarding HCL (2)

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

hazards versus dependencies

dependency — X needs result of instruction Y?

hazard — will it not work in some pipeline?

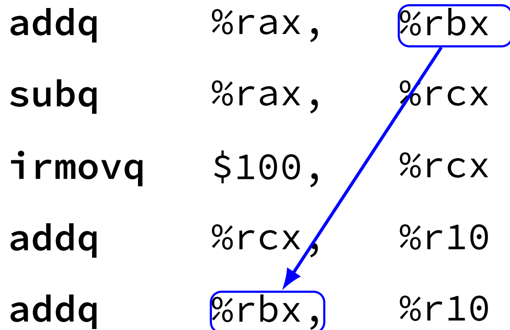
- before extra work is done to "resolve" hazards
- like forwarding or stalling or branch prediction

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 with forwarding?

ex.: dependencies and hazards (1)



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

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?

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 after second execute stage

where does forwarding, stalls occur?

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`									
`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

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

trick: guess and check

up to 3 extra cycles for data dependencies

can we do better?

when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

to "undo" instruction during fetch/decode:

forget everything in pipeline registers

making guesses

```
        subq   %rcx, %rax
        jne    LABEL
        xorq   %r10, %r11
        xorq   %r12, %r13
        ...
LABEL:  addq   %r8, %r9
        rmmovq %r10, 0(%r11)
```

speculate: jne will goto LABEL

right: 2 cycles faster!

wrong: forget before execute finishes

jXX: speculating right

```
         subq %r8, %r8
         jne LABEL
         ...

LABEL:   addq %r8, %r9
         rmmovq %r10, 0(%r11)
         irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	irmovq	rmmovq	addq	jne (done)	OPq

jXX: speculating right

```
        subq %r8, %r8
        jne LABEL
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
        irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	j		
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			
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

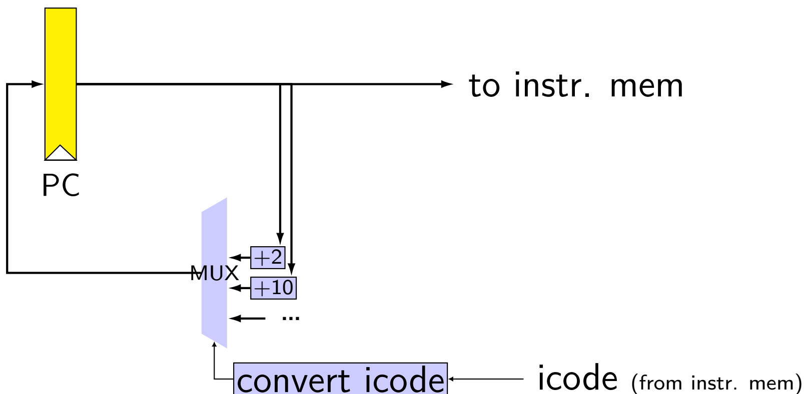
kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
ret	1%	4	4
others	91%	1*	1*

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

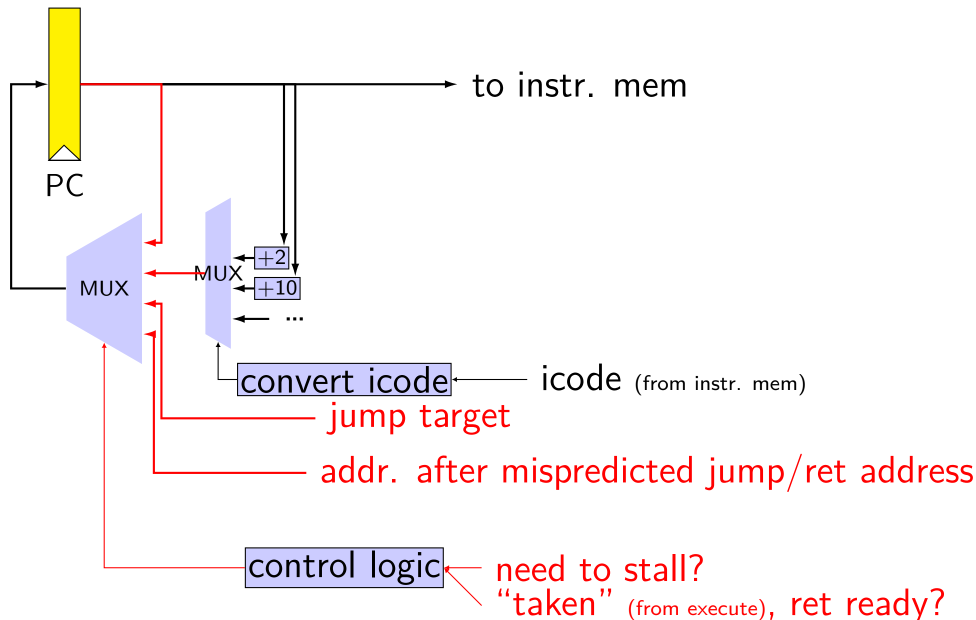
stall: $3 \times .03 + 3 \times .05 + 4 \times .01 + 1 \times .91 =$ 1.19 cylces/instr.

* — ignoring data hazards

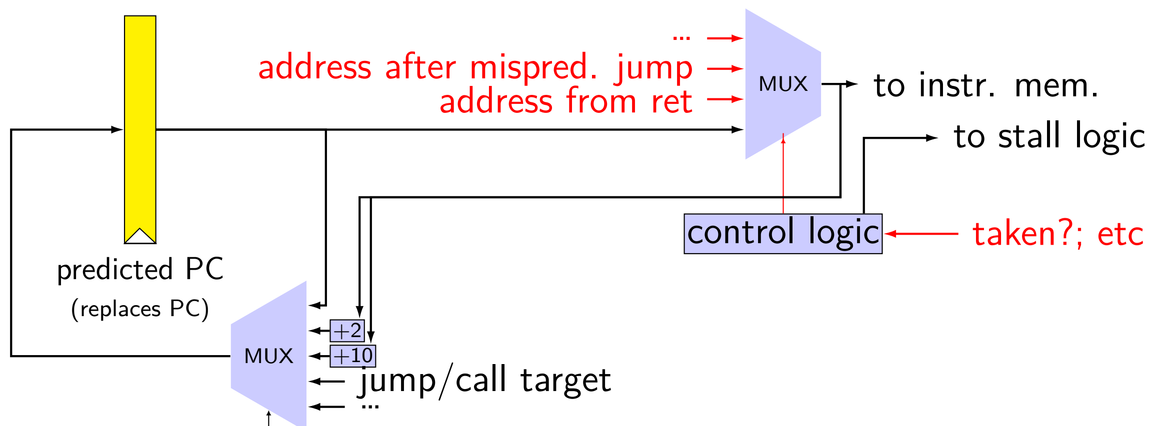
PC update (adding stall)



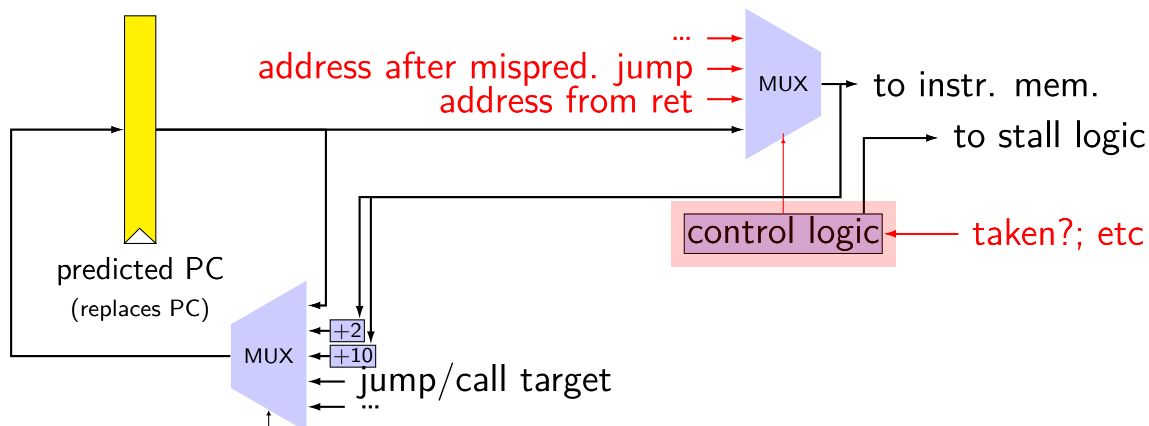
PC update (adding stall)



PC update (rearranged)

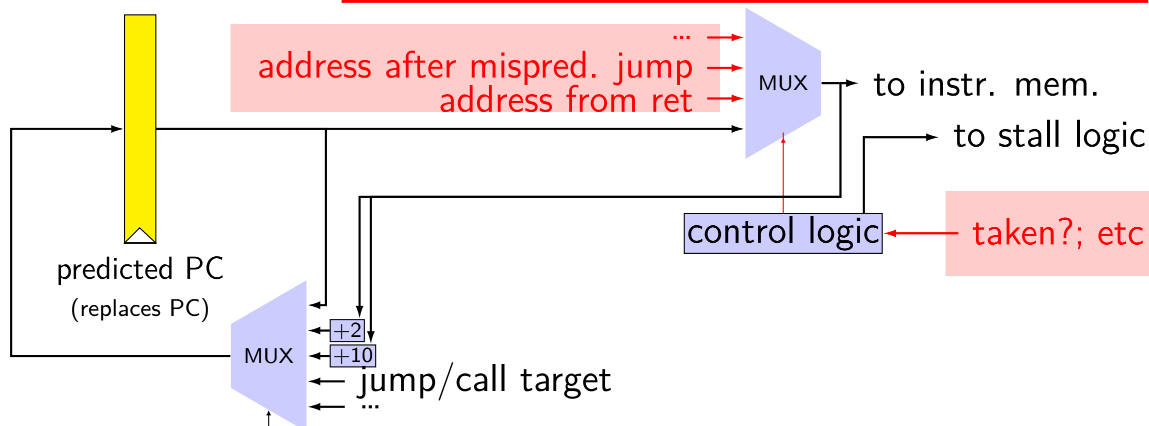


PC update (rearranged)

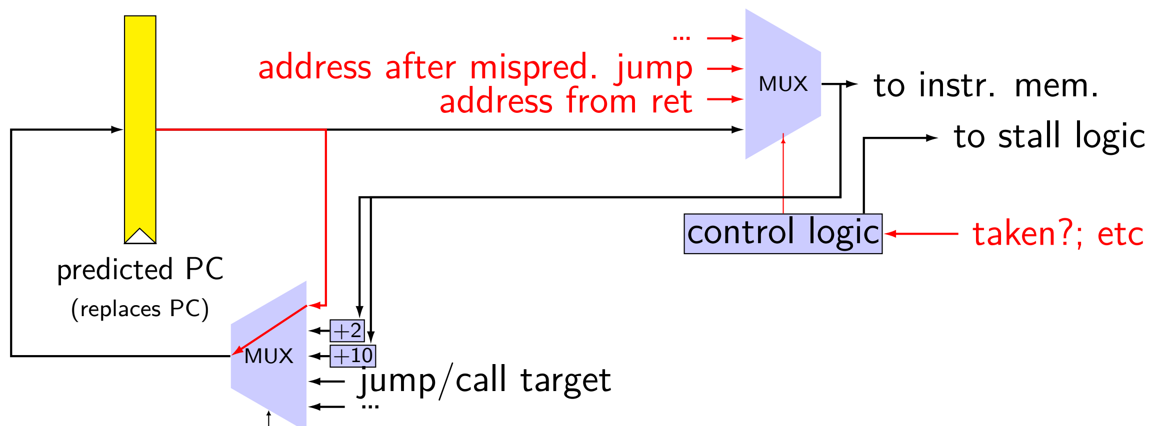


PC update (rearranged)

same logic as before — but happens in next cycle
inputs are from slightly different place...
(e.g. 'taken?' from *execute to memory* registers,
not *execute* directly)



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?

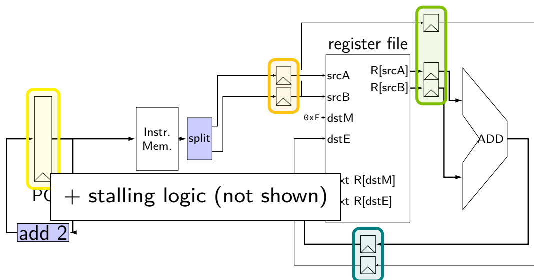
either works
- correct PC at beginning or end of cycle?
- still some time in cycle to do so...

maybe easier to think about branch prediction this way?

backup slides

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

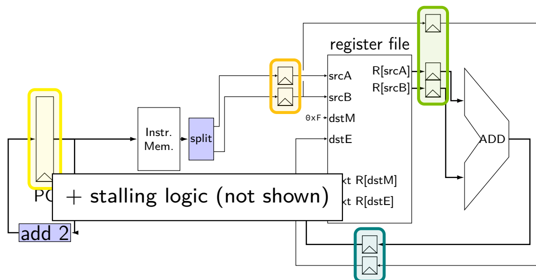
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```

register file
srcA
srcB
0xF dstM
dstE
R[srcA]
R[srcB]
R[dstM]
R[dstE]
Instr. Mem.
split
ADD
PC
add 2
+ 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
```

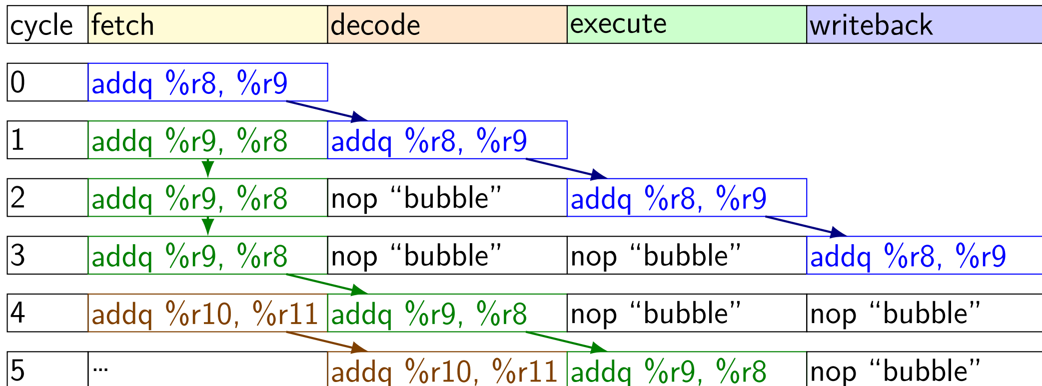


	fetch	fetch→decode		decode→execute			execute→writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2*	8	9					
2	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4	0x4	9	8	---	---	F	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8

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"