PIPE Hazards

Changelog

Corrections made in this version not in first posting:

27 Mar 2017: slide 12 (RET stall): correct typo of `need_ret_bubble` for `need_ret_stall`

logistical note: HCL2D bug

bug in HCL2D — unnoticed for several semesters

register file writes sometimes happen early

(can *sometimes* read register values before written)

also memory writes (but much harder to notice)

rarely affects processor correctness, but confusing

we will grade with more lenient version

on the last post-quiz

miskeyed some questions

dropped question on ret hazard — no actual clear dependency

pipelining summary

fetch/decode/execute/memory/writeback

add pipeline registers

normal next PC logic in fetch

branch prediction for jXX

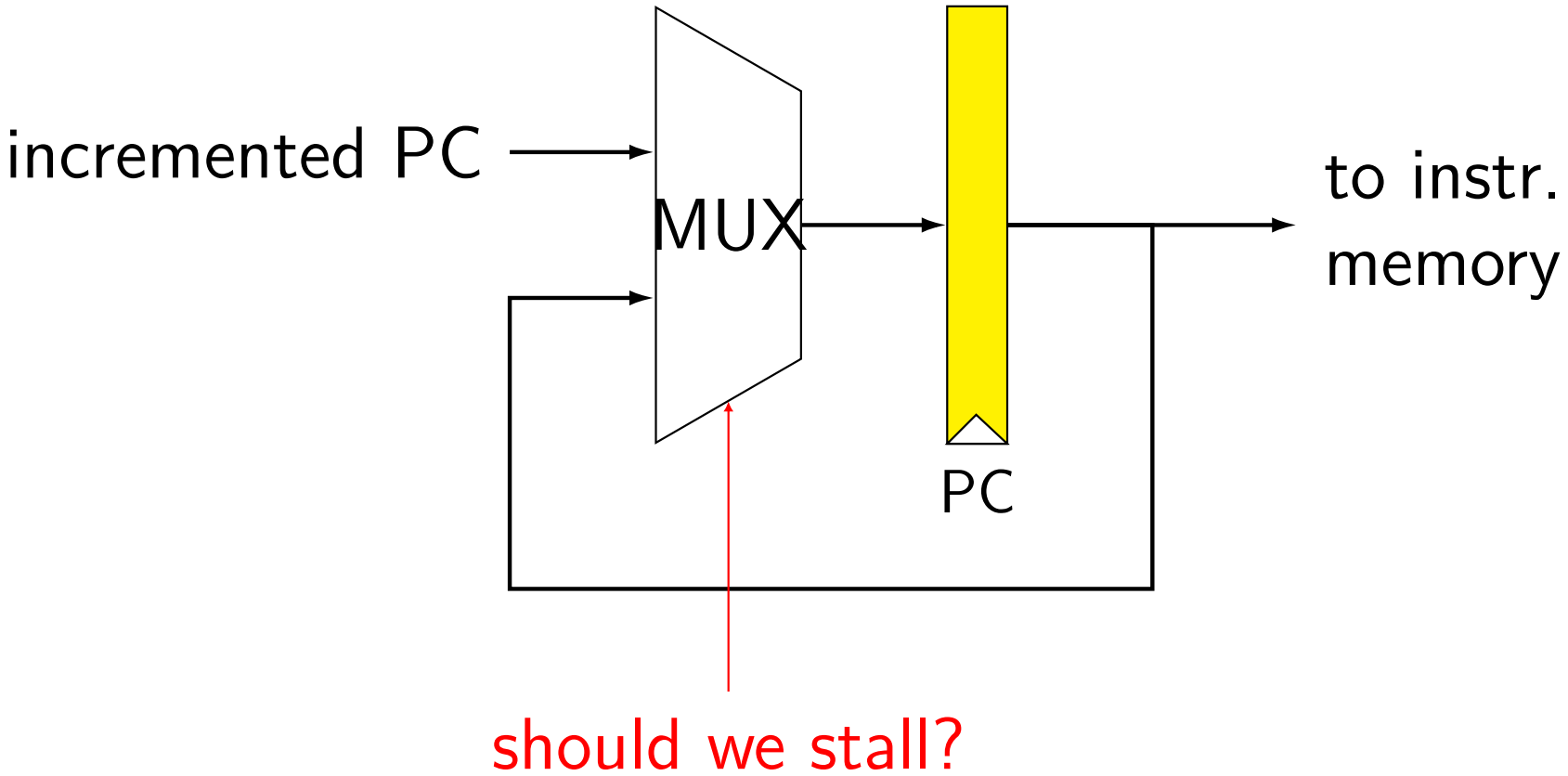
- assume taken; verify before following 'execute' finishes

stalling for ret

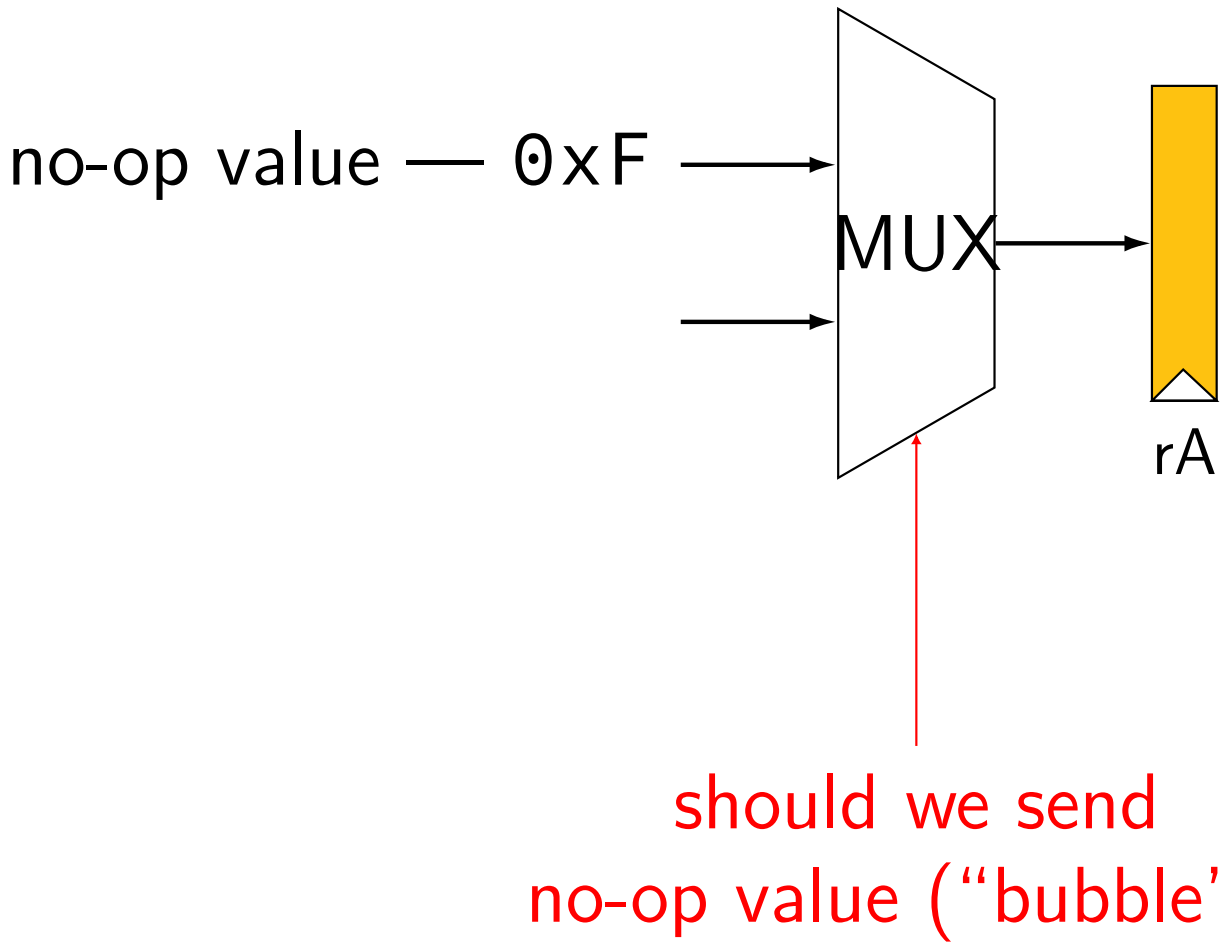
slower due to branch misprediction and stalling

- reason pipeline depth matters

recall: fetch/fetch logic — stall or not



recall: fetch/decode logic — bubble or not



HCL2D signals

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

register bank has three modes

`stall_B`: keep old value for all registers

`bubble_B`: use default value for all registers

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		

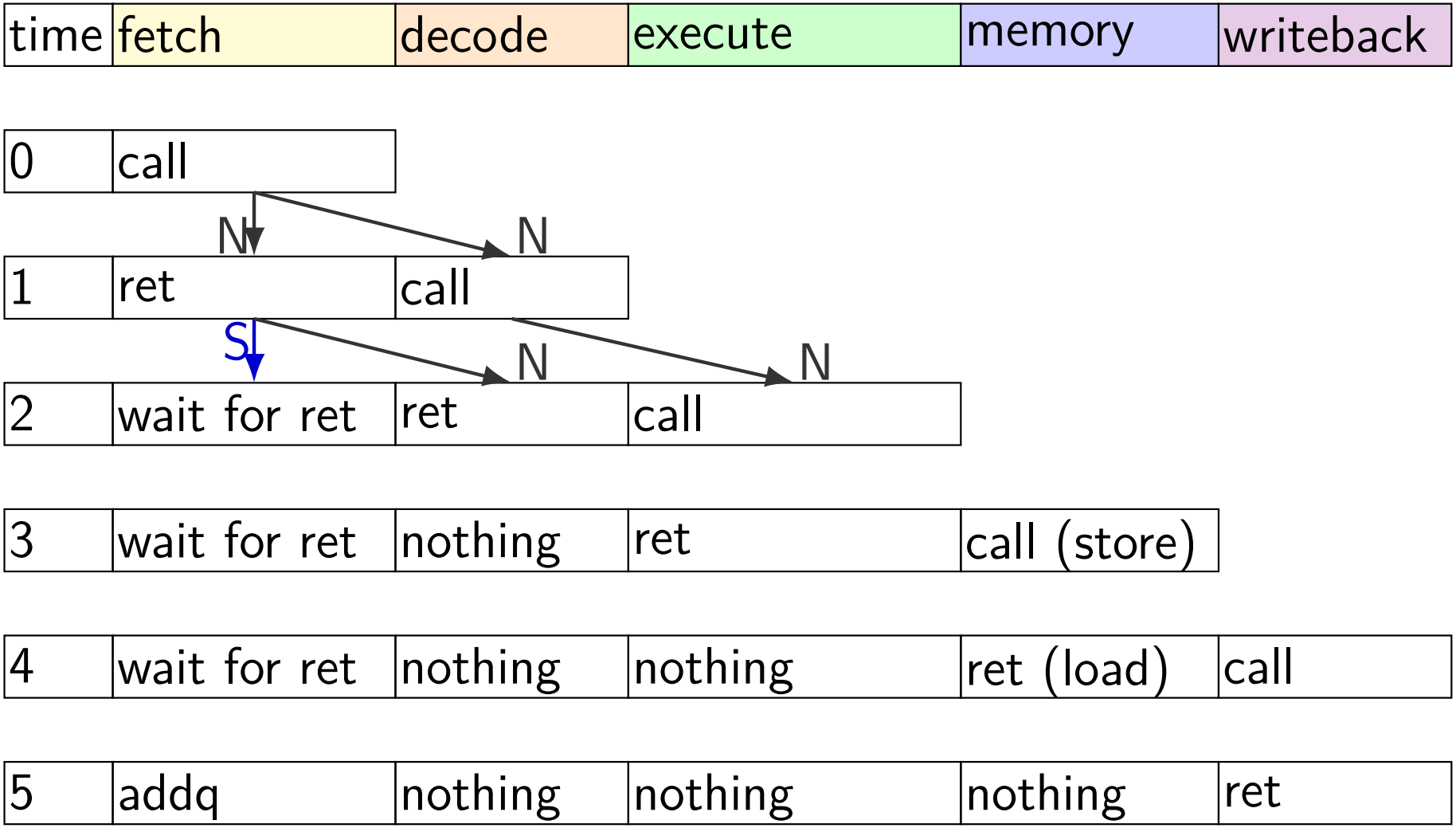
ret stall

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);

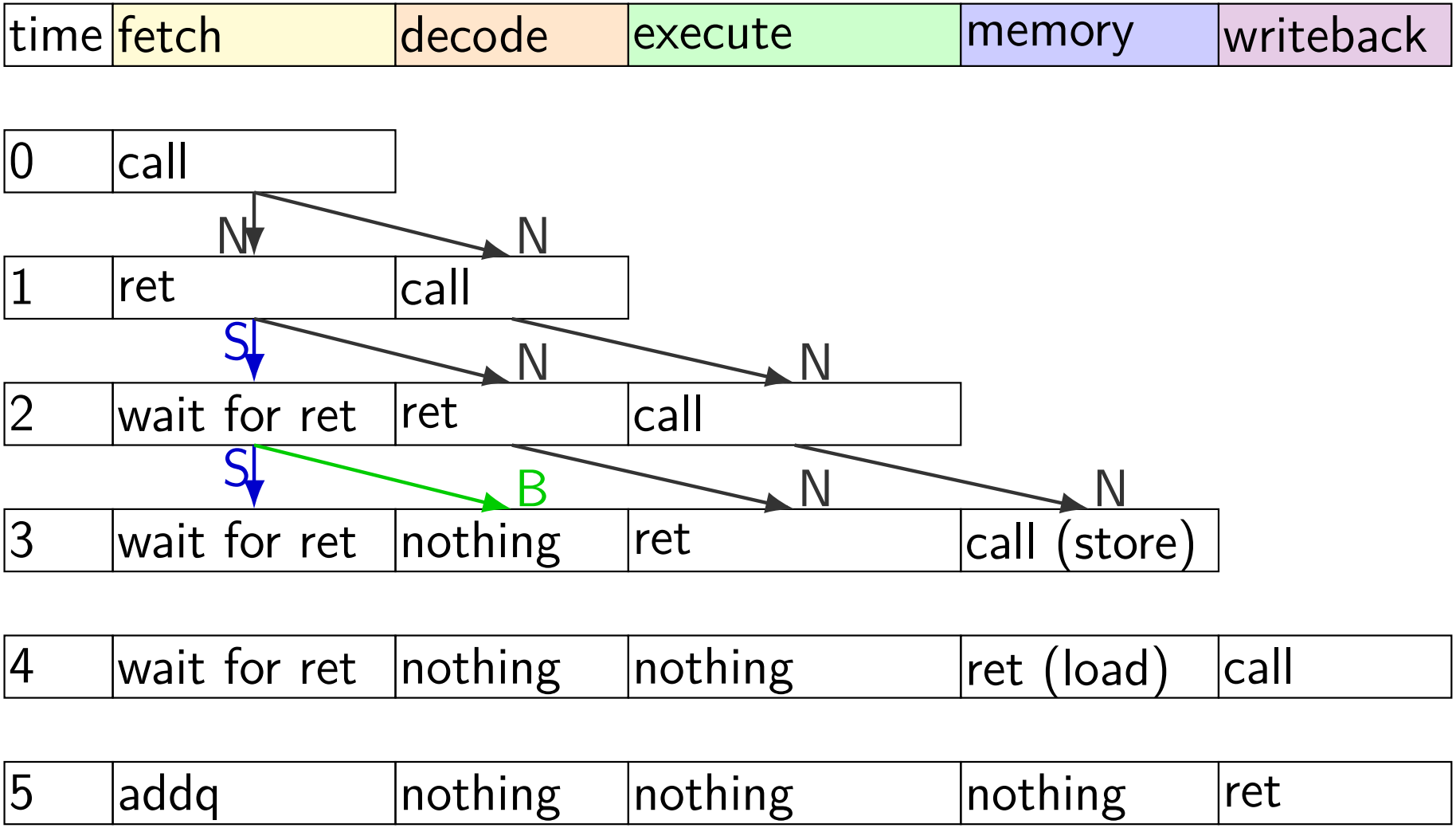
ret stall



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

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

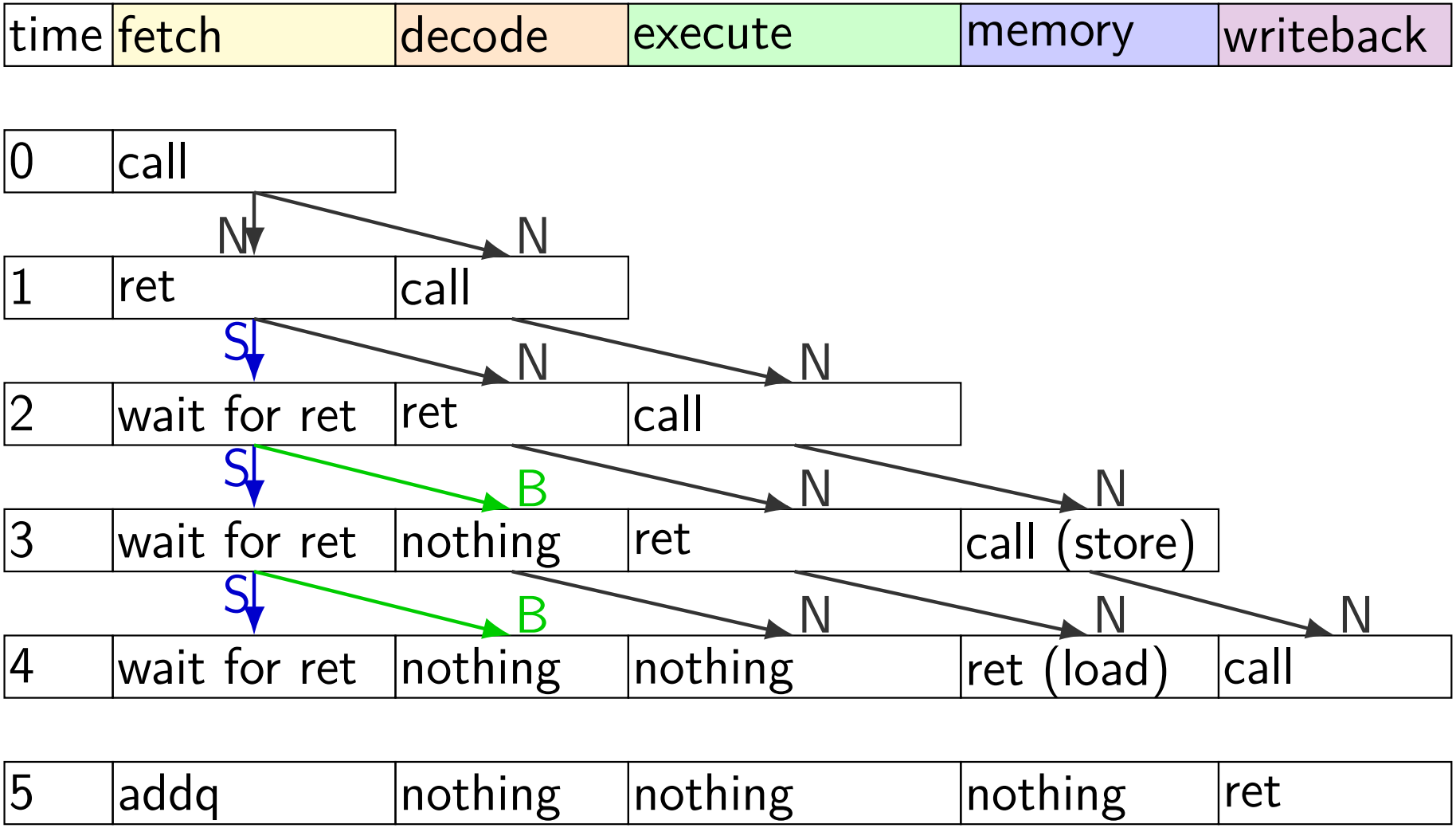
ret stall



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

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

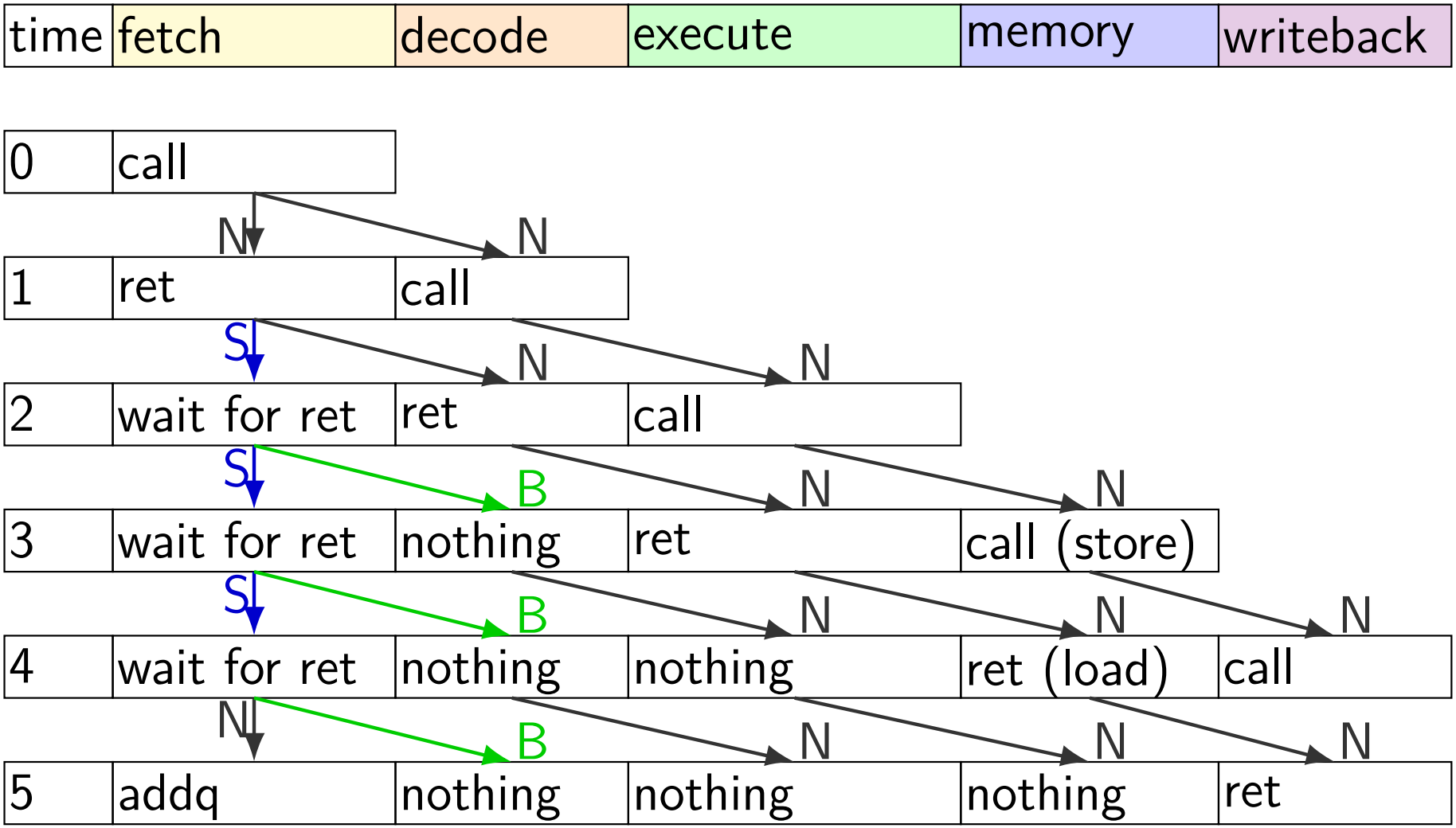
ret stall



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);

HCL2D 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 */ );
```

HCL2D stall example

```
register pP {
    pc : 64 = 0;
    ...
};
wire need_ret_stall : 1;
need_ret_stall = (
    icode_from_imem == RET ||
    D_icode == RET ||
    E_icode == RET
);

stall_P = (
    need_ret_stall ||
    ... /* other cases */
)
```

alternative view

cycle #	0	1	2	3	4	5	6	7	8	
call	F	D	E	M	W					
ret		F	D	E	M	W				
addq			F	F	F	F	D	E	M	W

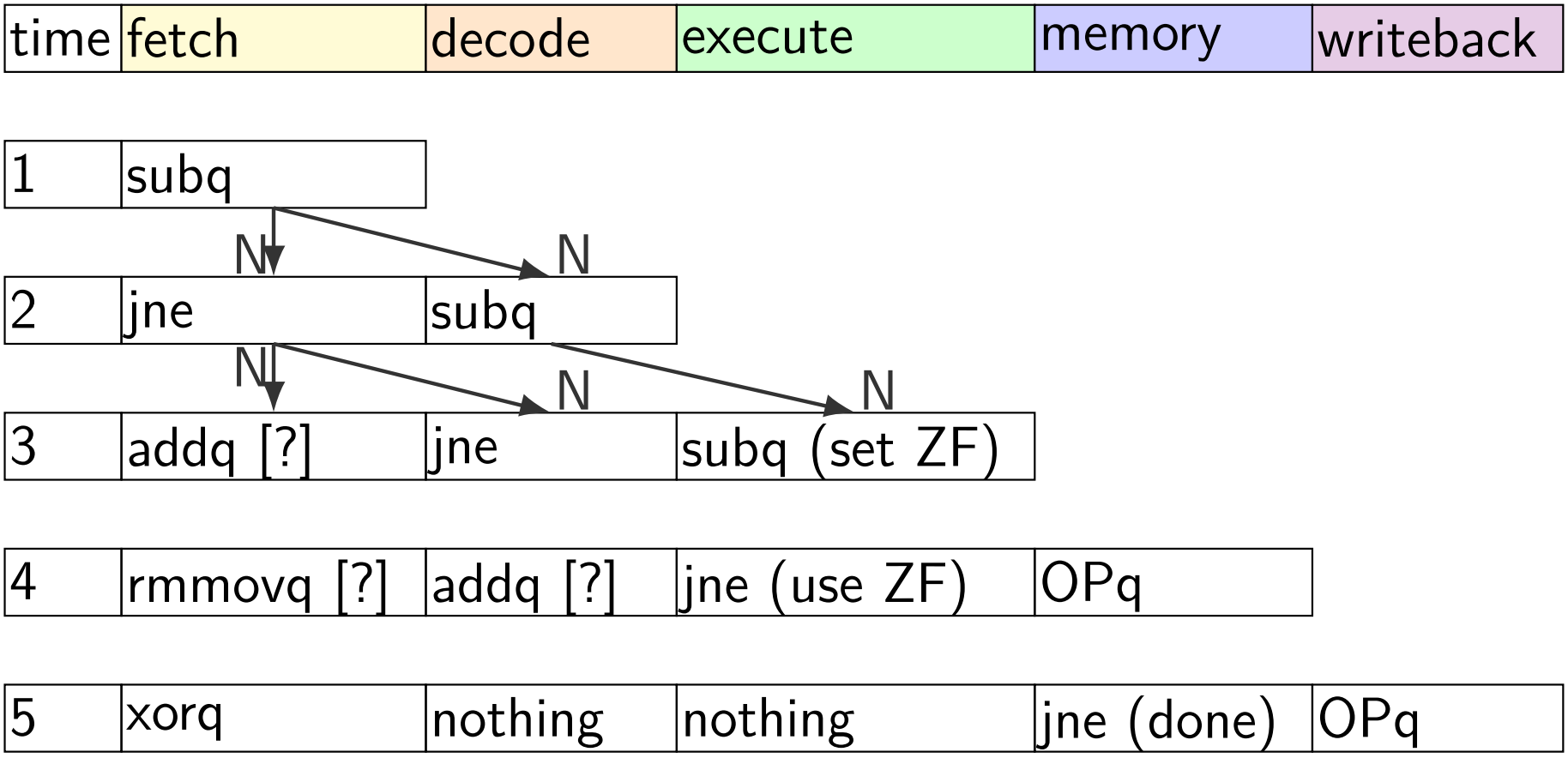
squashing with stall/bubble

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

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

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

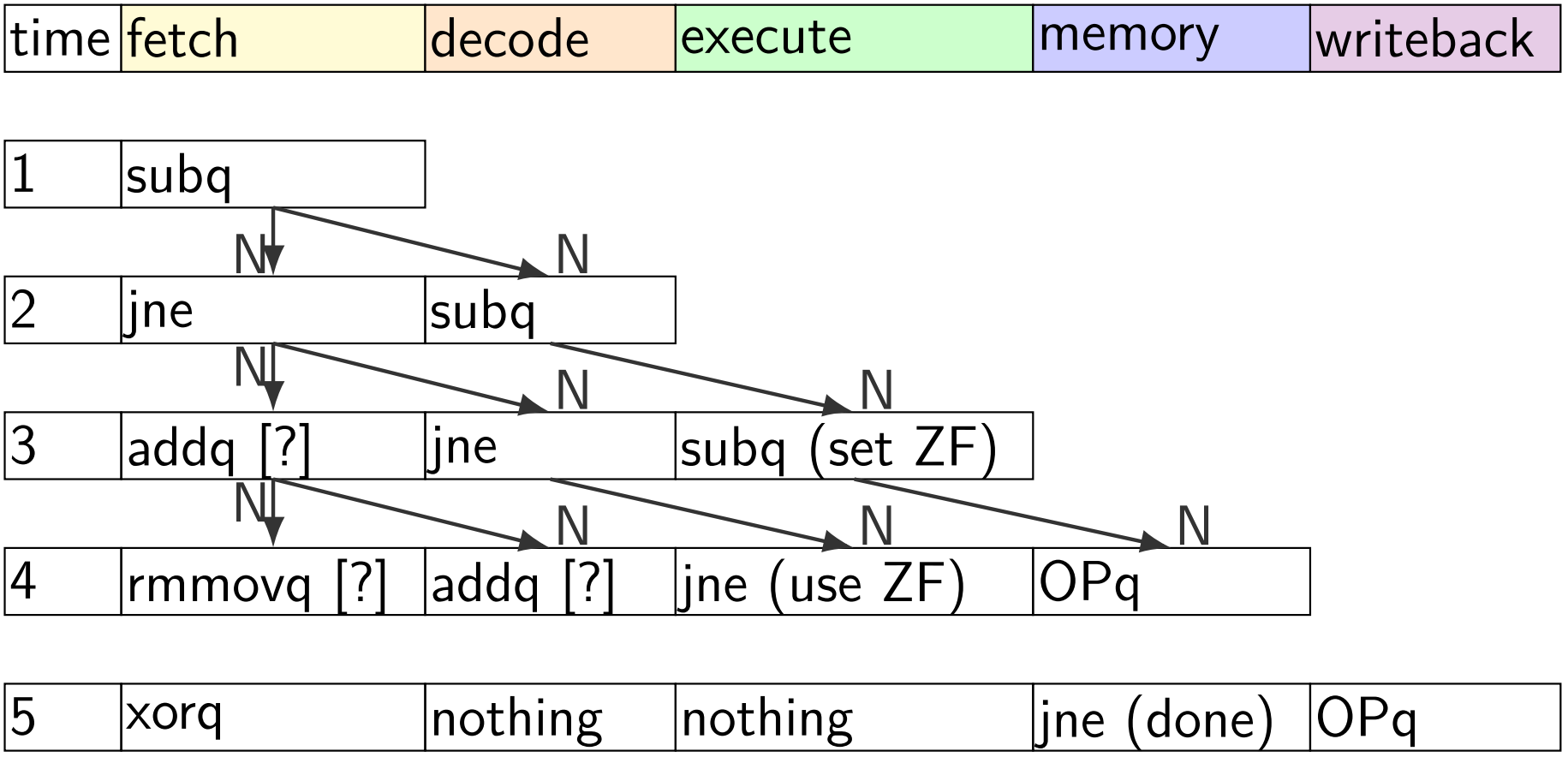
squashing with stall/bubble



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

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

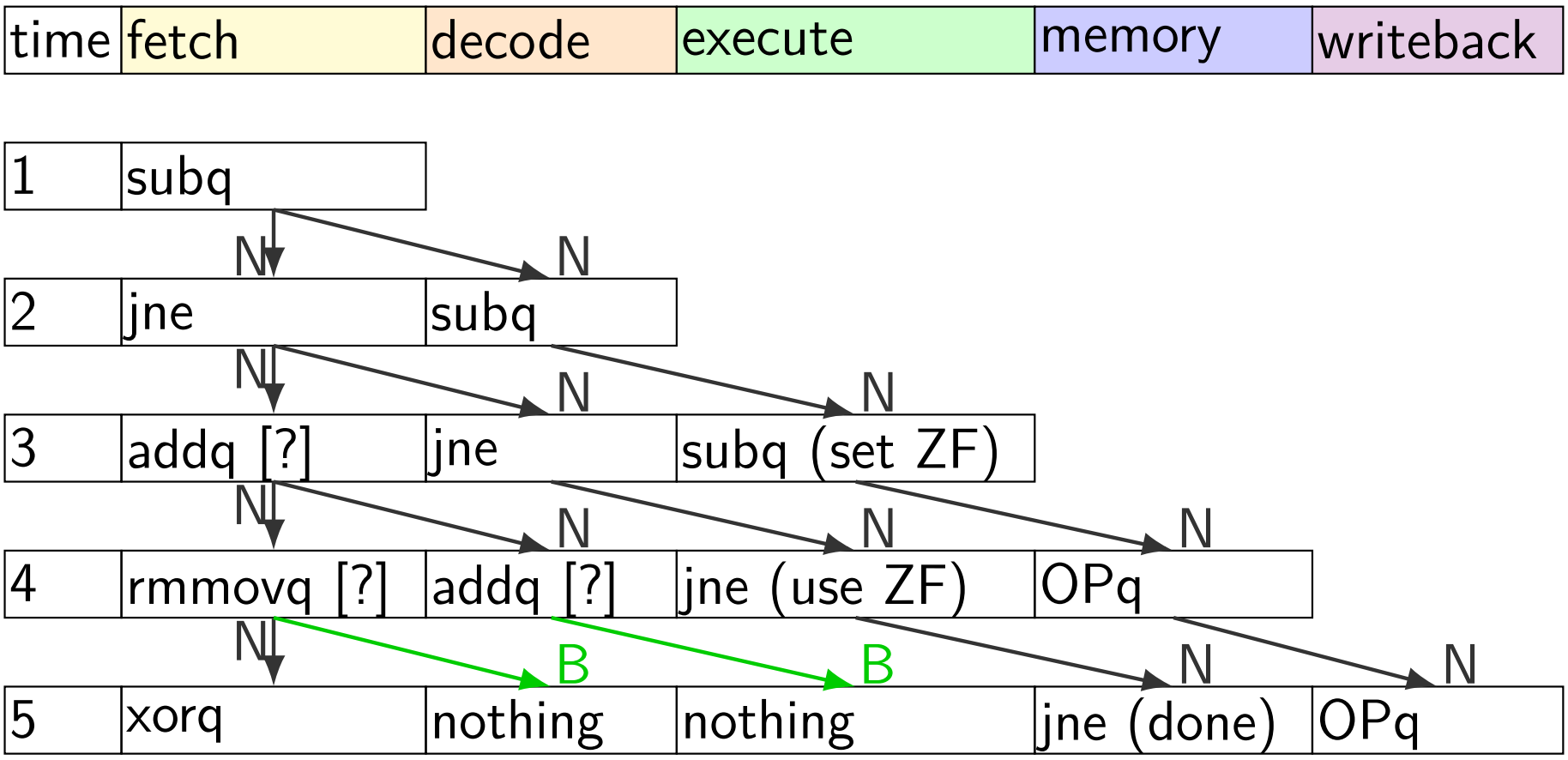
squashing with stall/bubble



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

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

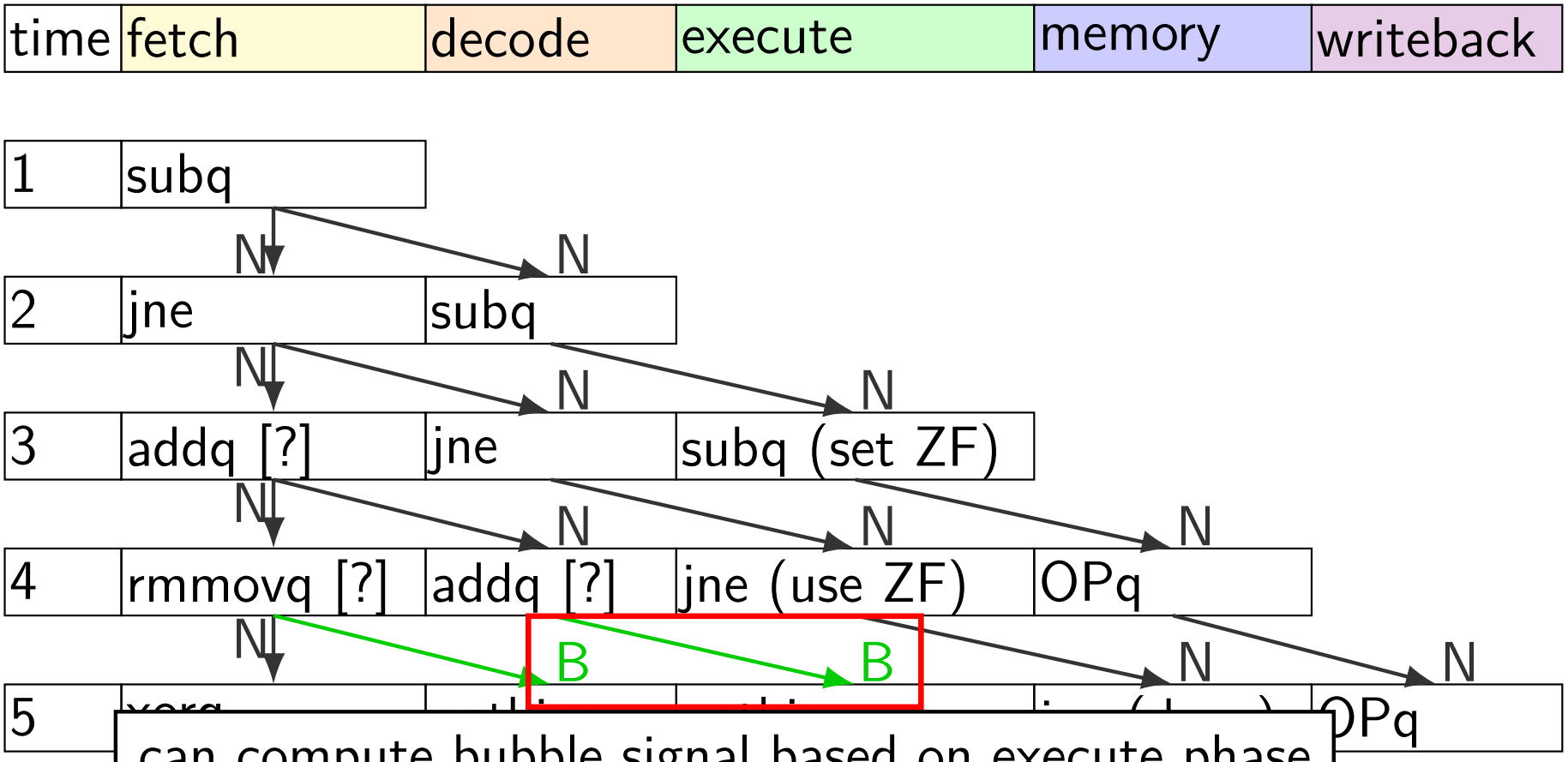
squashing with stall/bubble



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

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

squashing with stall/bubble



can compute bubble signal based on execute phase
won't even start CC write for addq

stal ew value

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

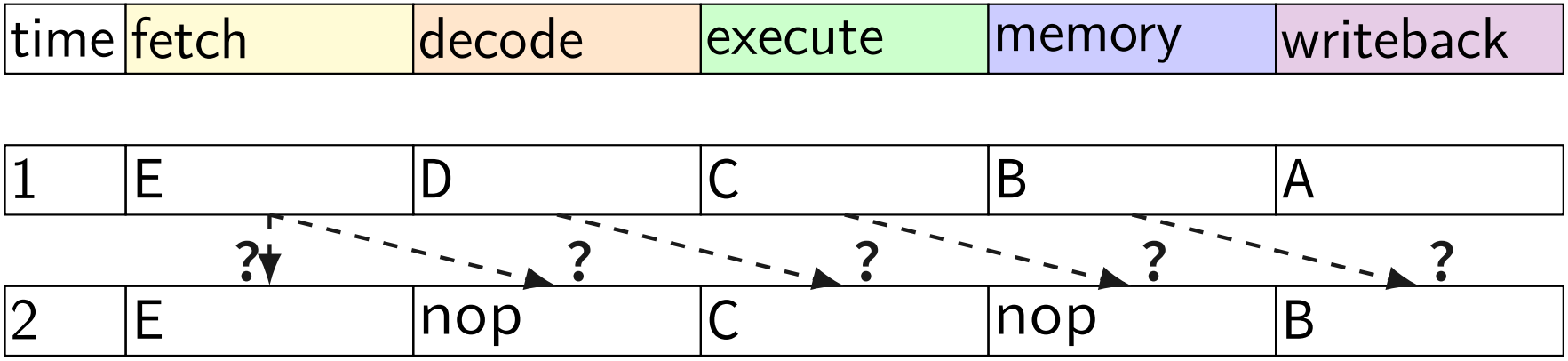
squashing HCL2D

```
just_detected_mispredict =
    e_icode == JXX && !e_branchTaken;
bubble_D = just_detected_mispredict || ...;
bubble_E = just_detected_mispredict || ...;
```

alternative view

cycle #	0	1	2	3	4	5	6	7	8
subq	F	D	E	M	W				
jne		F	D	E	M	W			
addq*			F	D					
rmmovq*				F					
xorq					F	D	E	M	W

exercise: squash + stall (1)

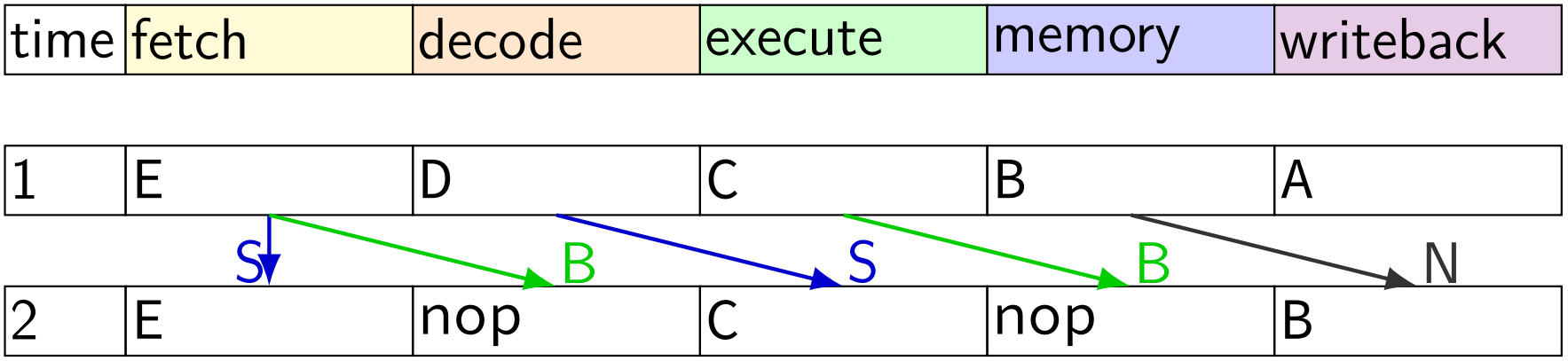


stall (S) = keep old value; normal (N) = use new value

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

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

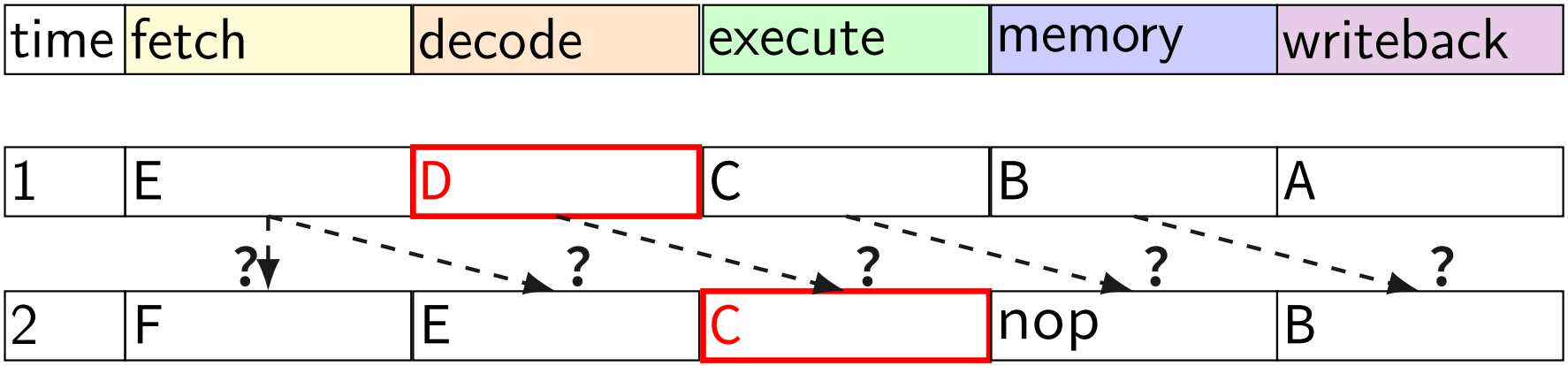
exercise: squash + stall (1)



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

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

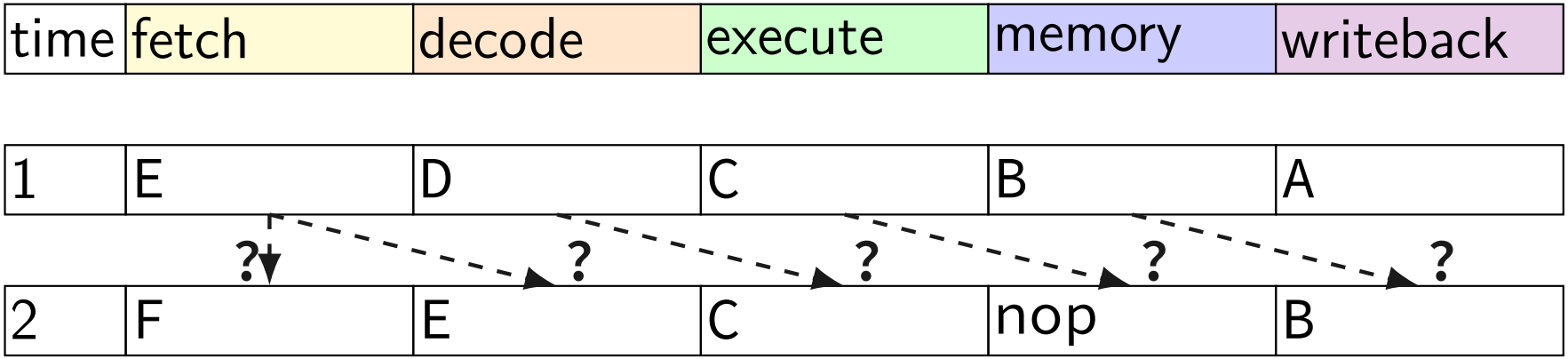
exercise: squash + stall (2)



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

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

exercise: squash + stall (2)

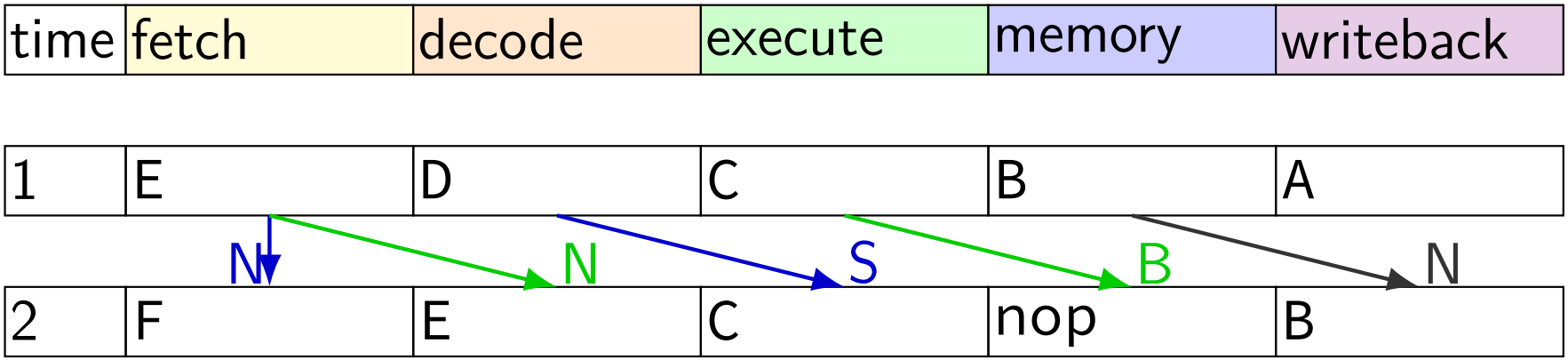


stall (S) = keep old value; normal (N) = use new value

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);

revisiting data hazards

stalling worked

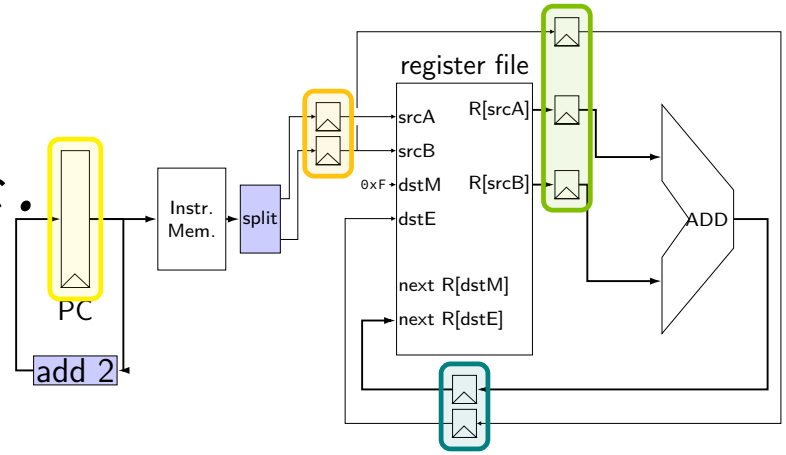
but very unsatisfying — wait 2 extra cycles to use anything?!

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 ...
```

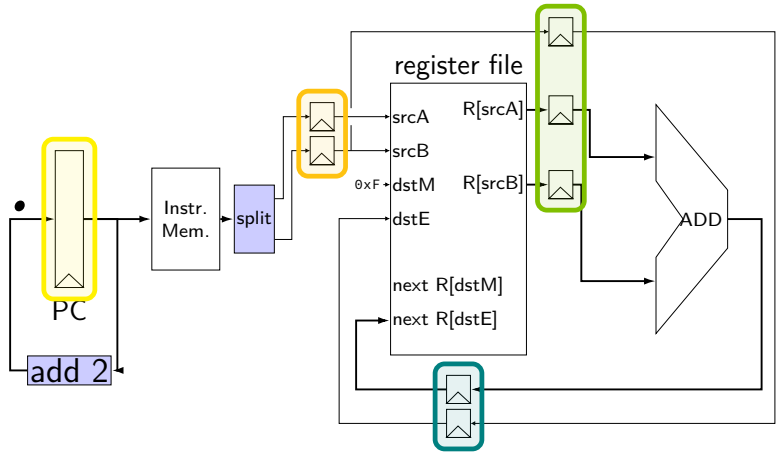


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700

motivation

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

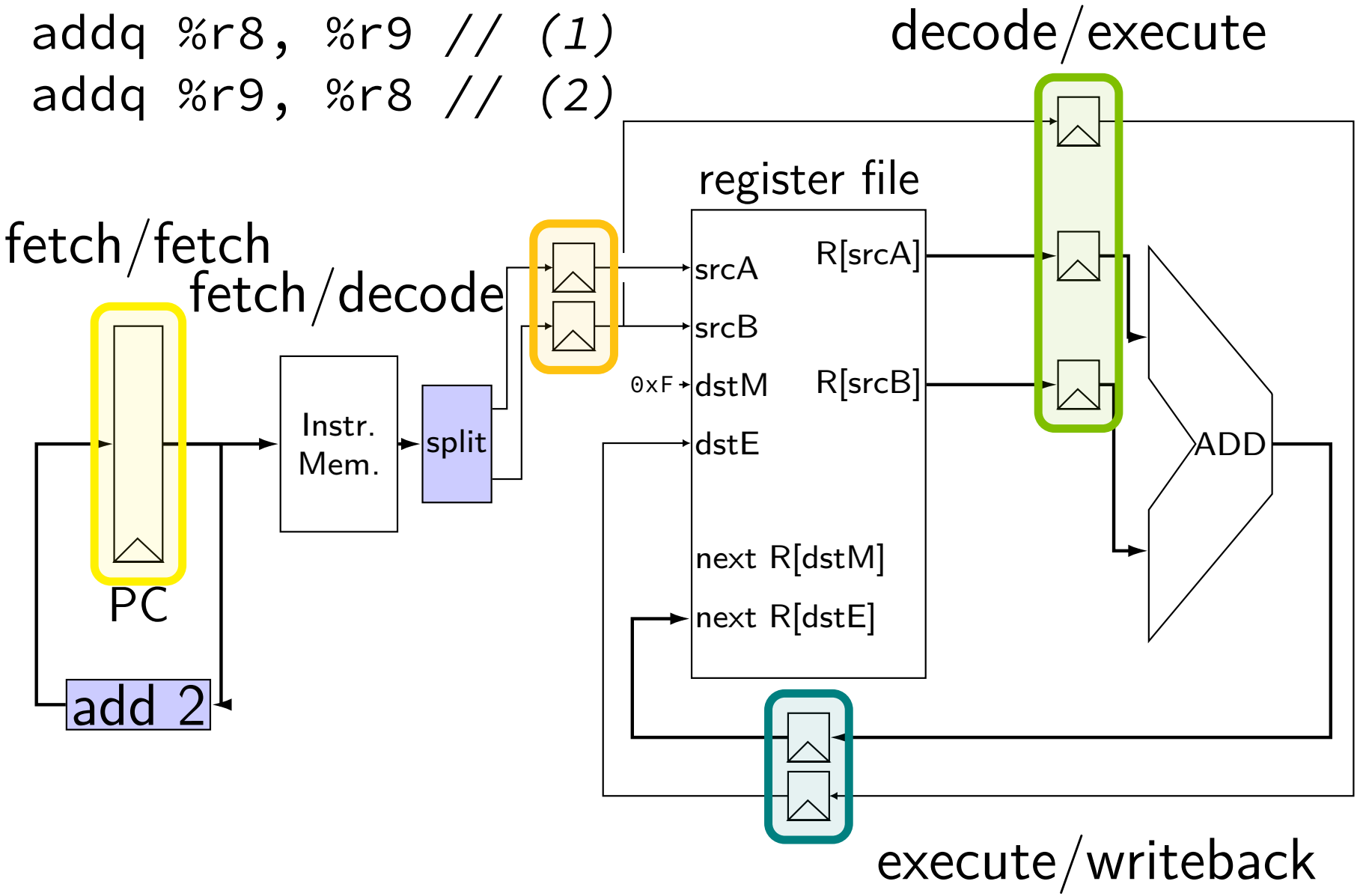


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700

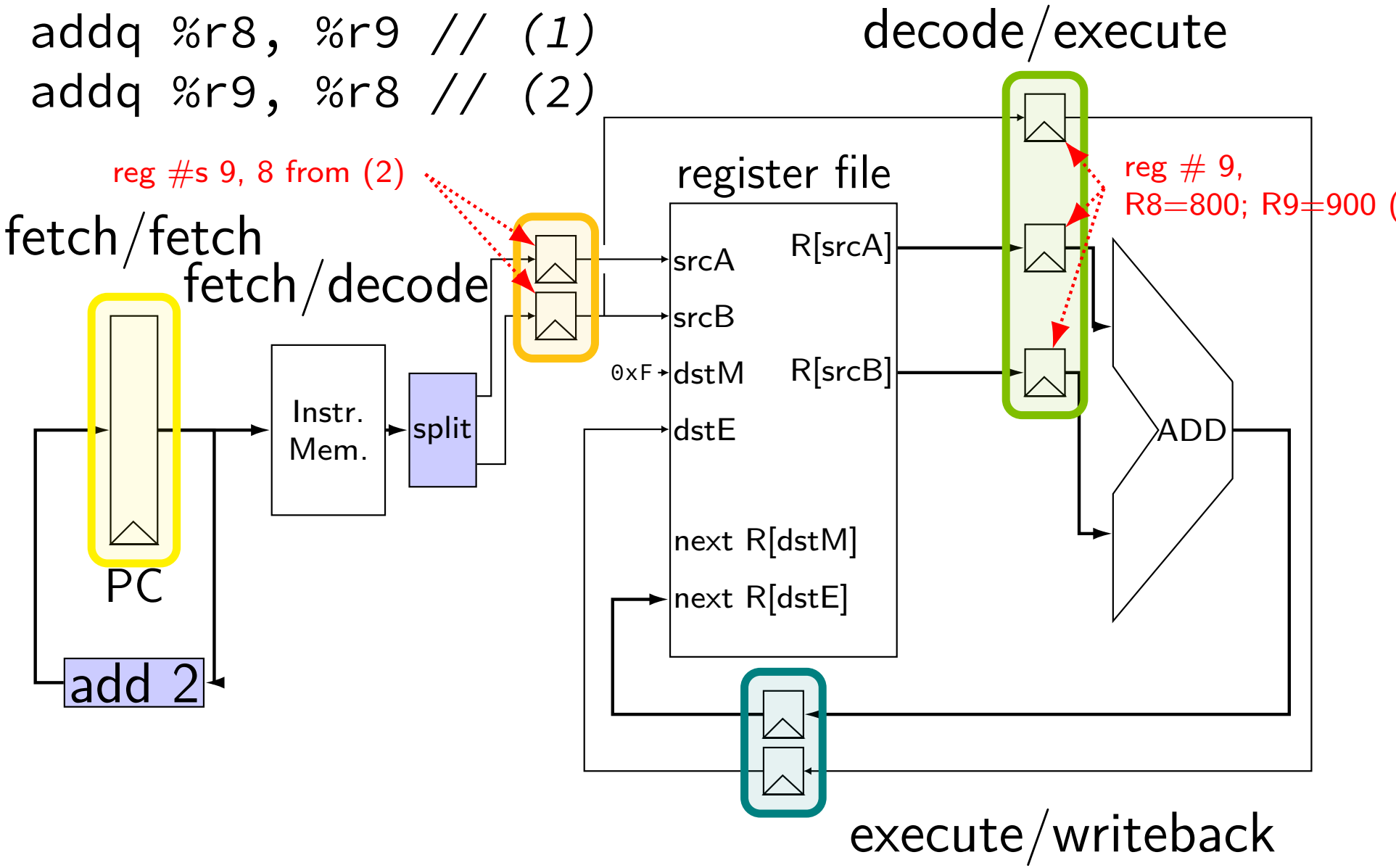
forwarding

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



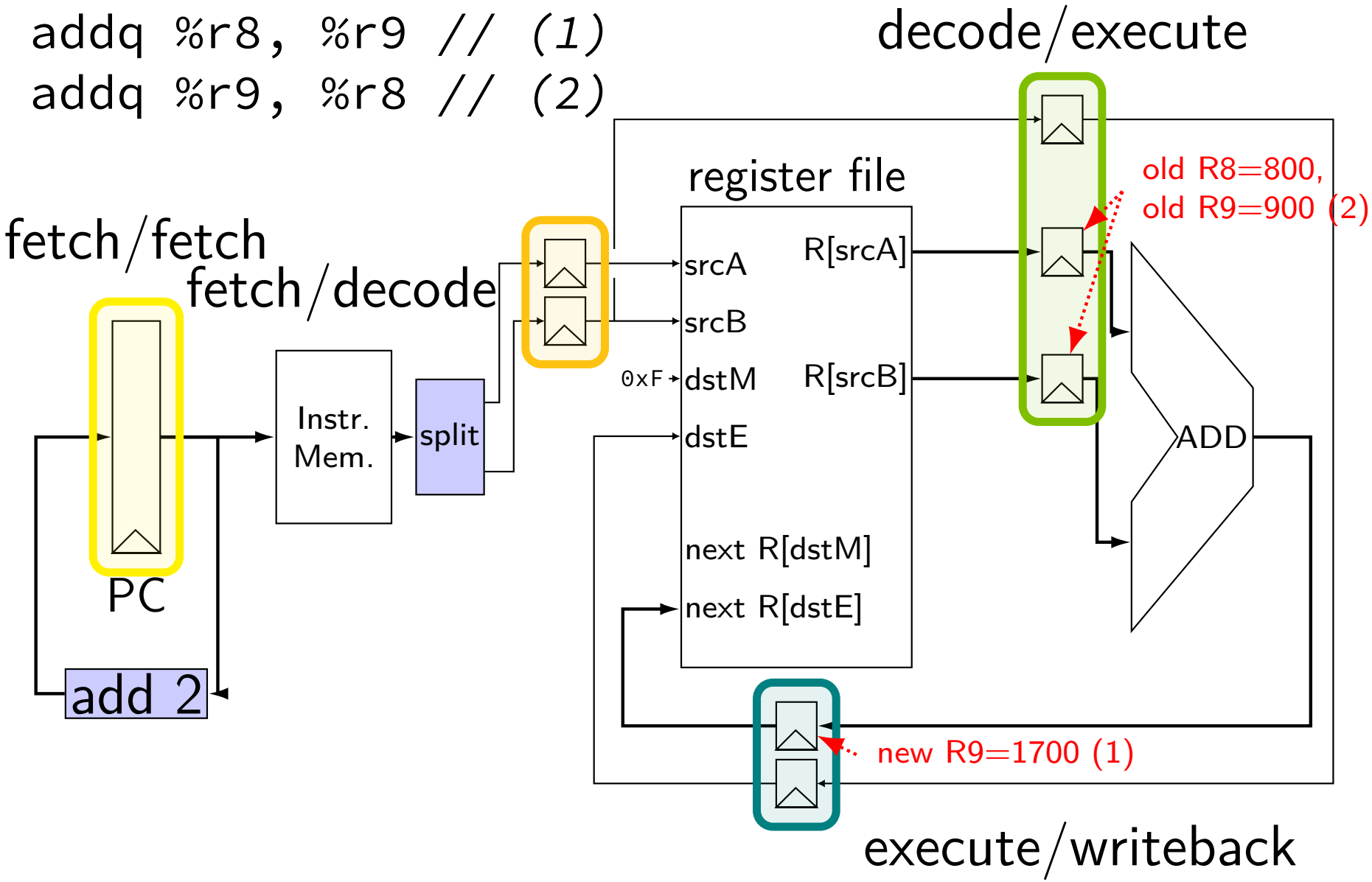
forwarding

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



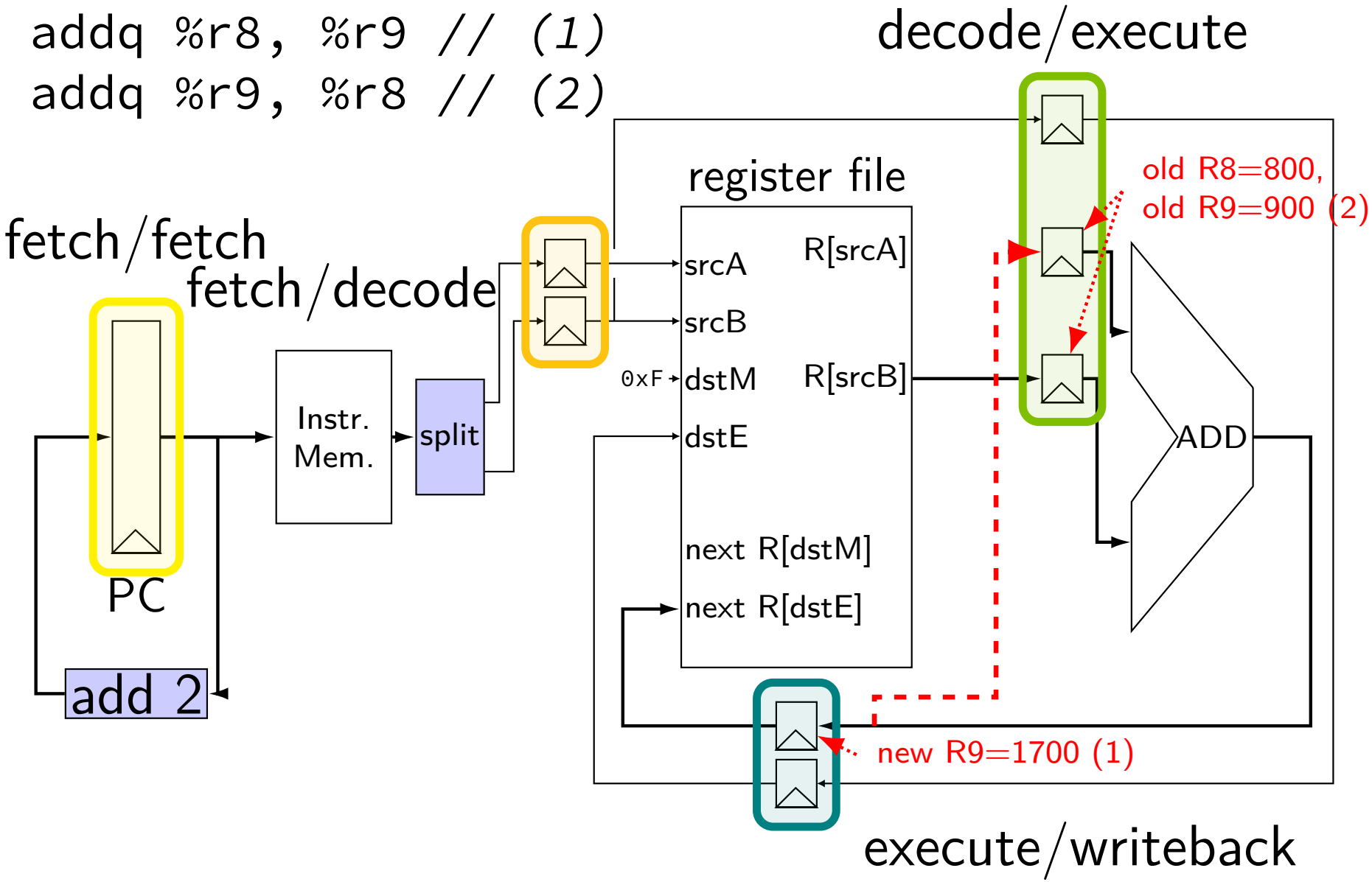
forwarding

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



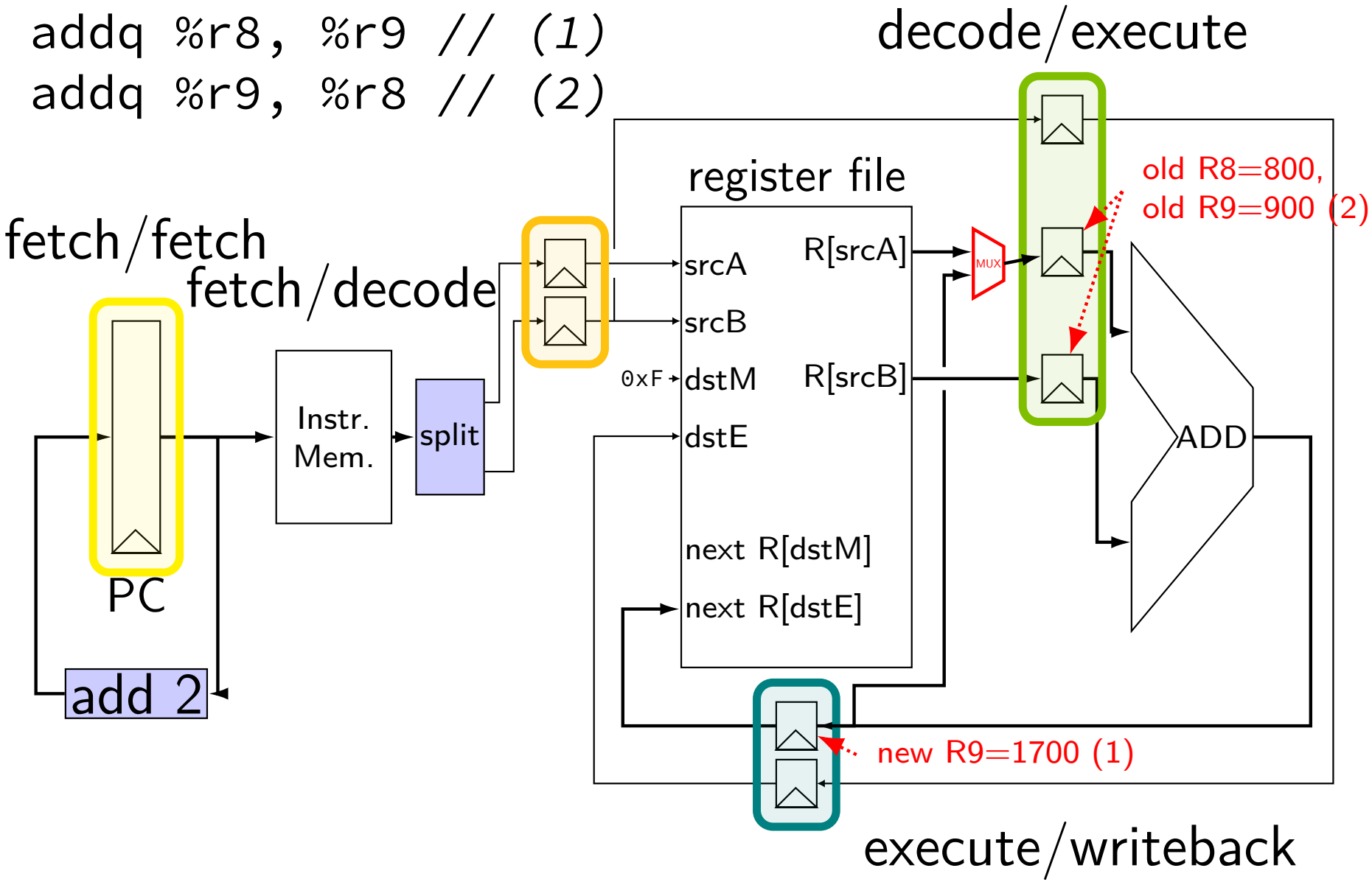
forwarding

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



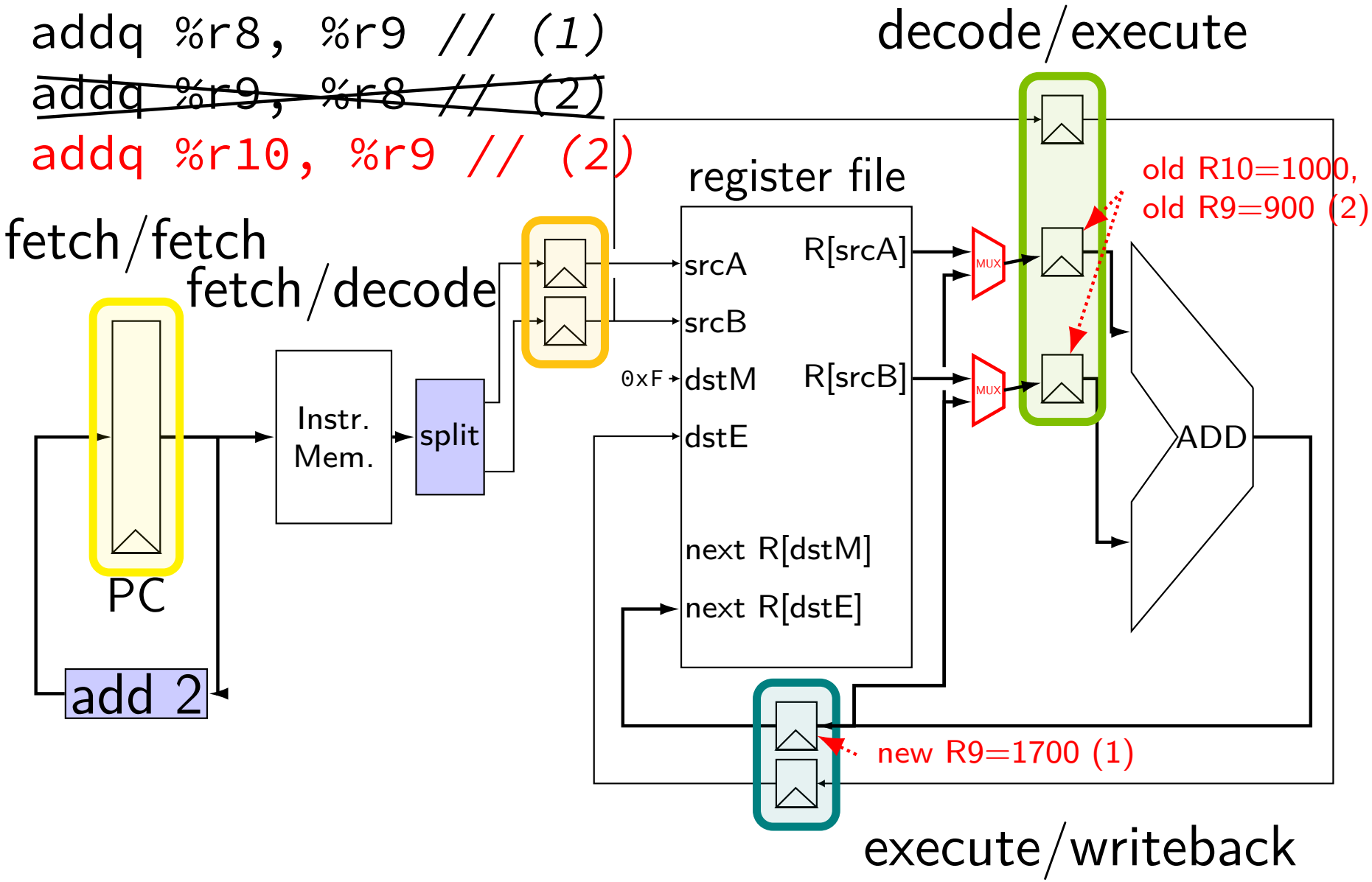
forwarding

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



forwarding

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



Forwarding Idea

read wrong value (e.g. from register)

correct value is already computed

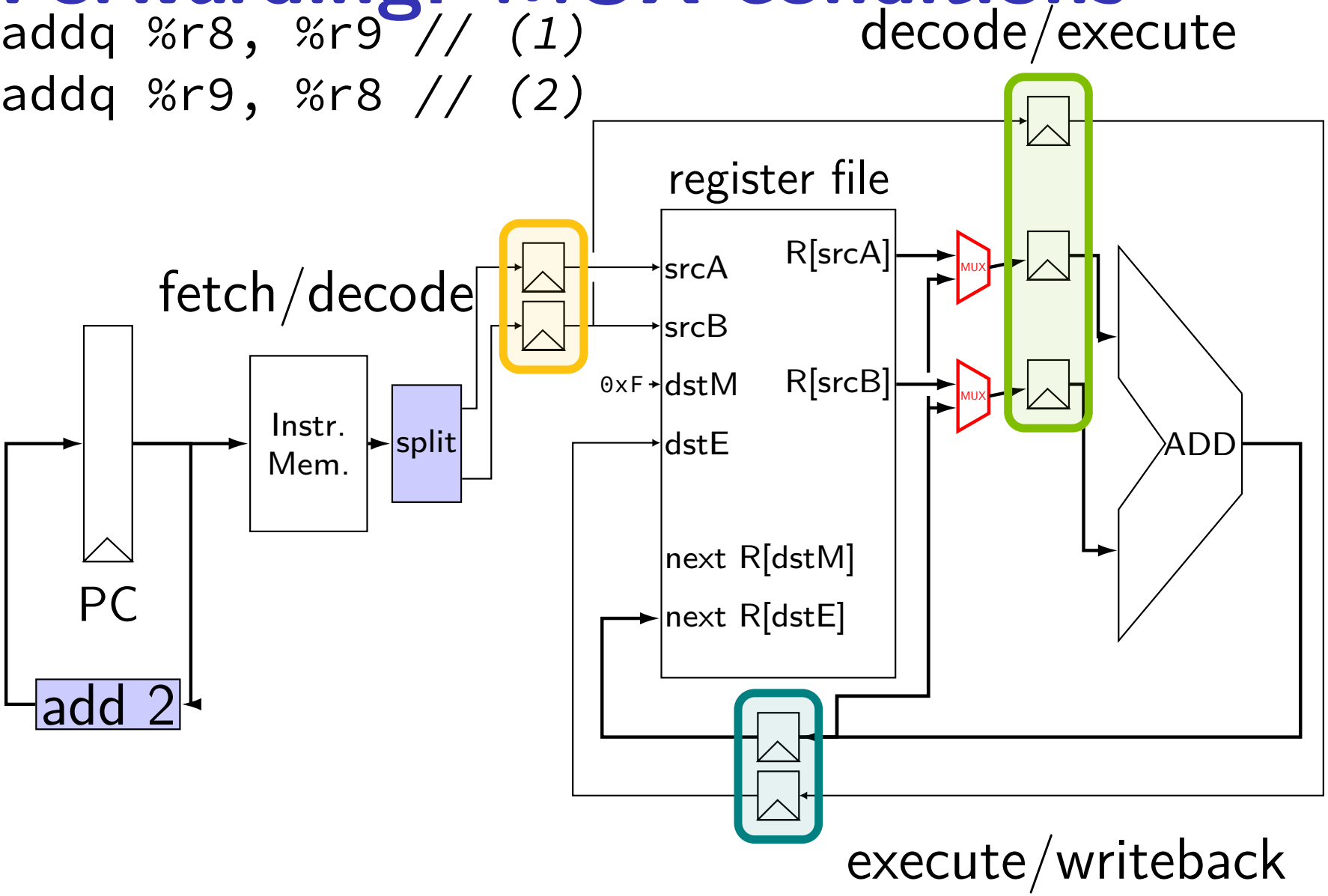
- elsewhere in pipeline
- maybe even after old value was read

substitute from wrong value

- using MUX

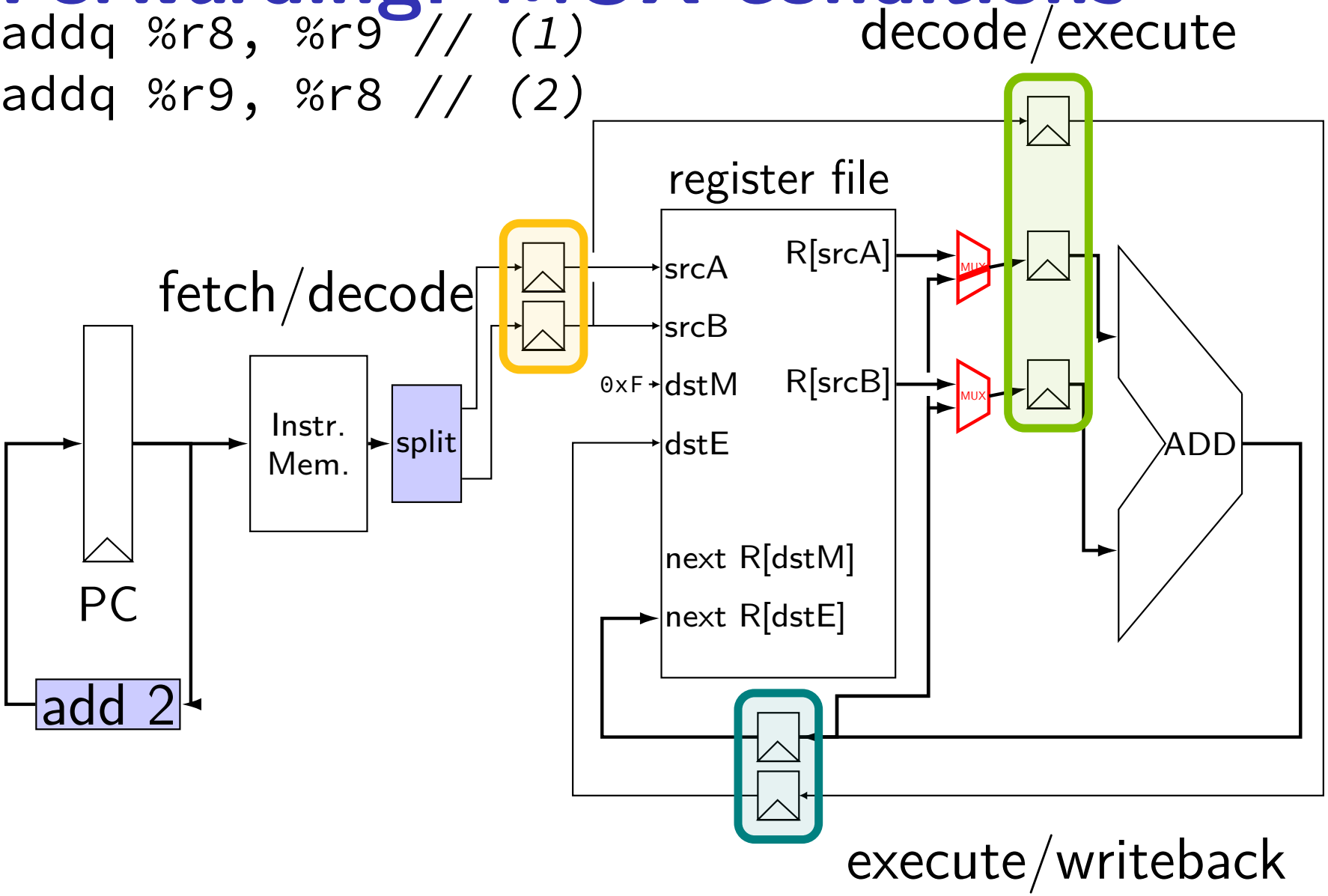
Forwarding: MUX conditions

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



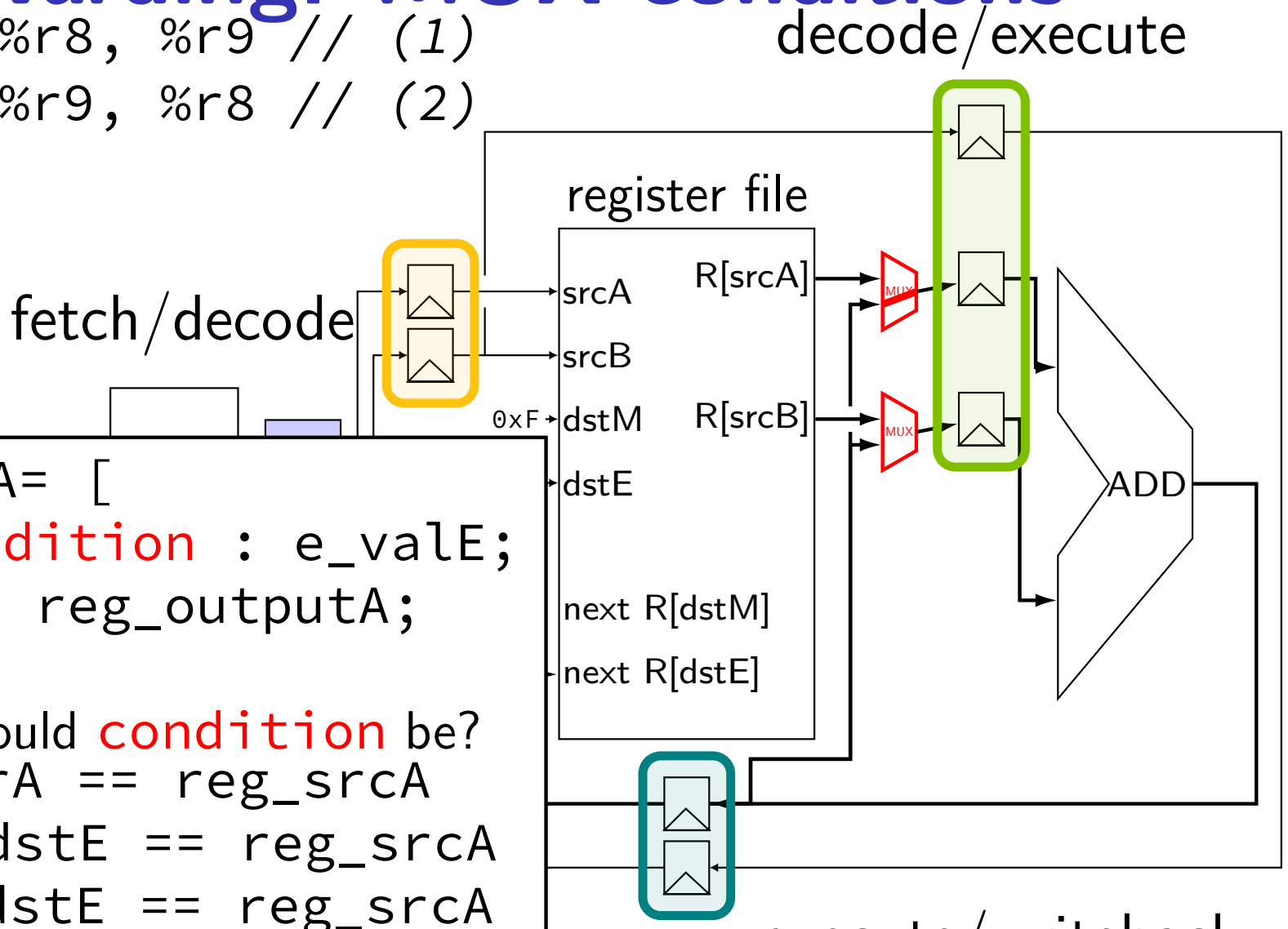
Forwarding: MUX conditions

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



Forwarding: MUX conditions

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



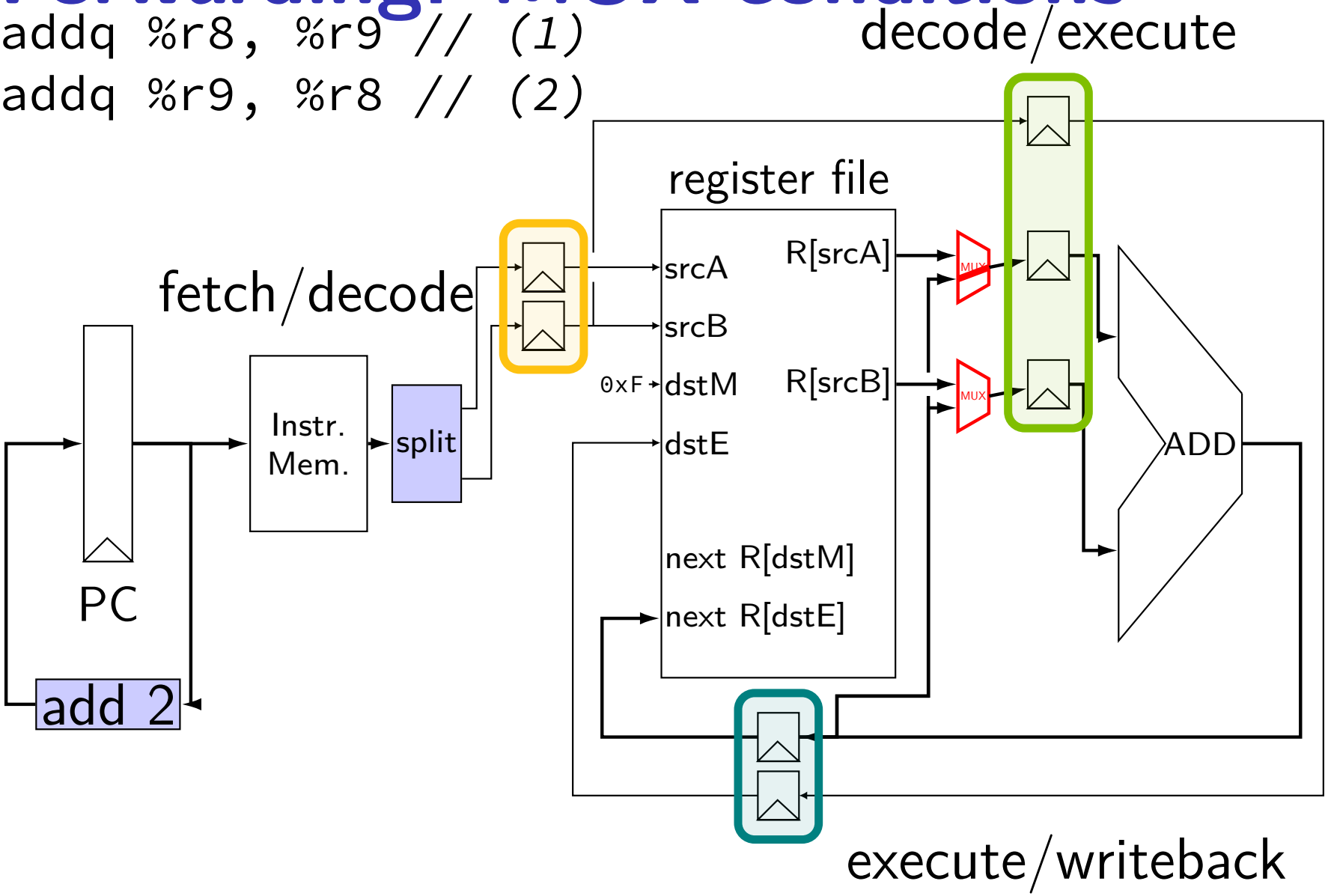

```
d_valA= [
   condition : e_valE;
   1 : reg_outputA;
];
```

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

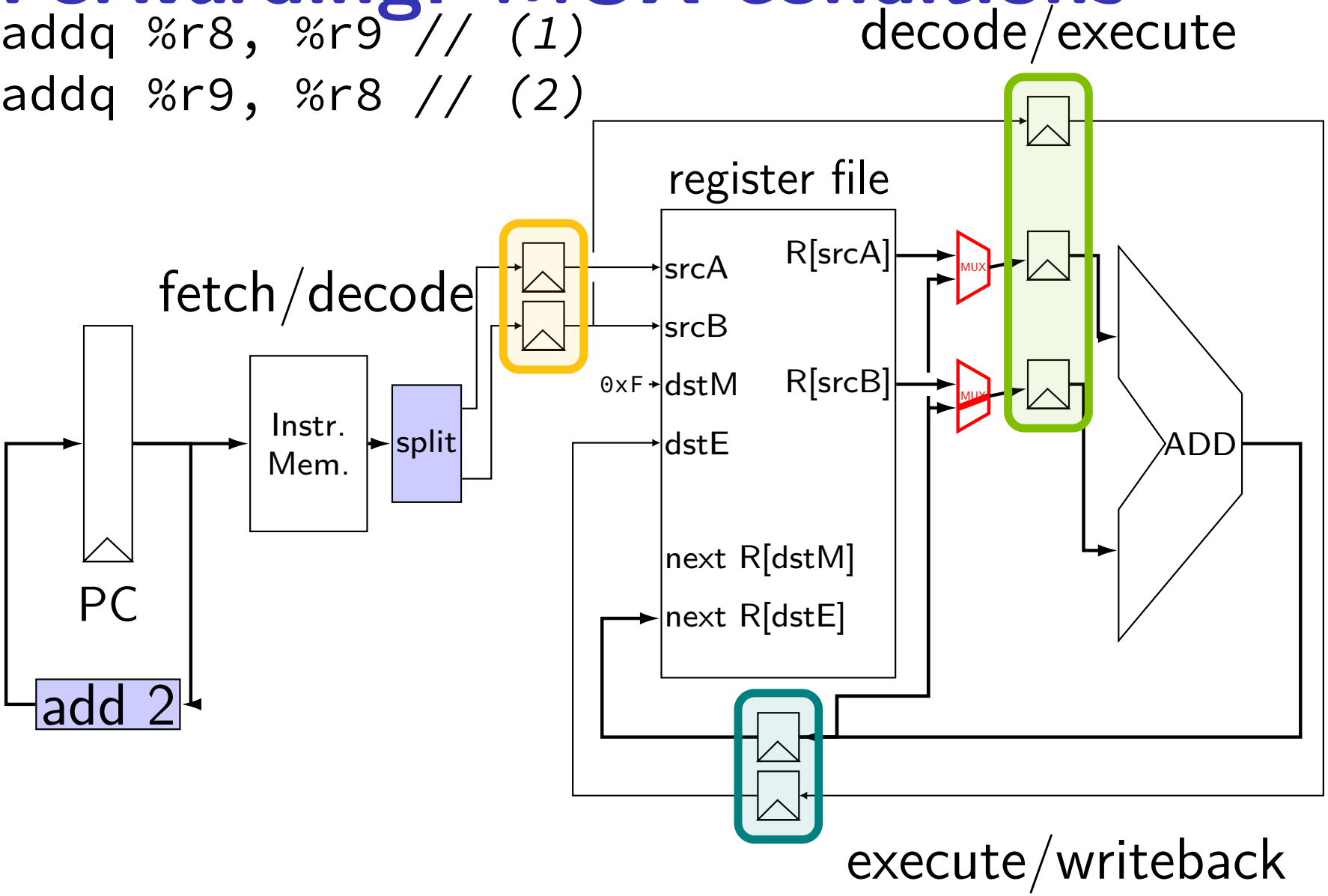
Forwarding: MUX conditions

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

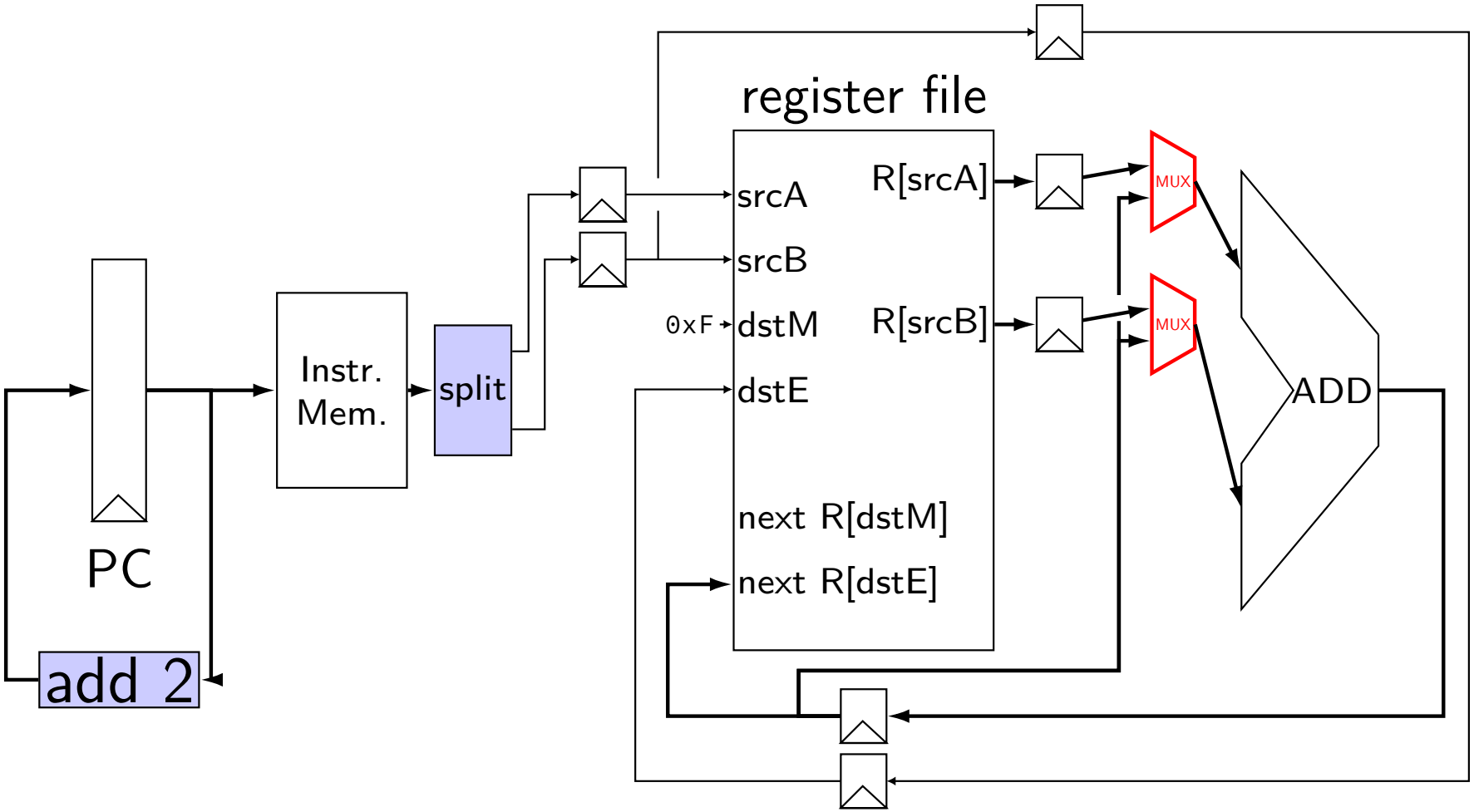


Forwarding: MUX conditions

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



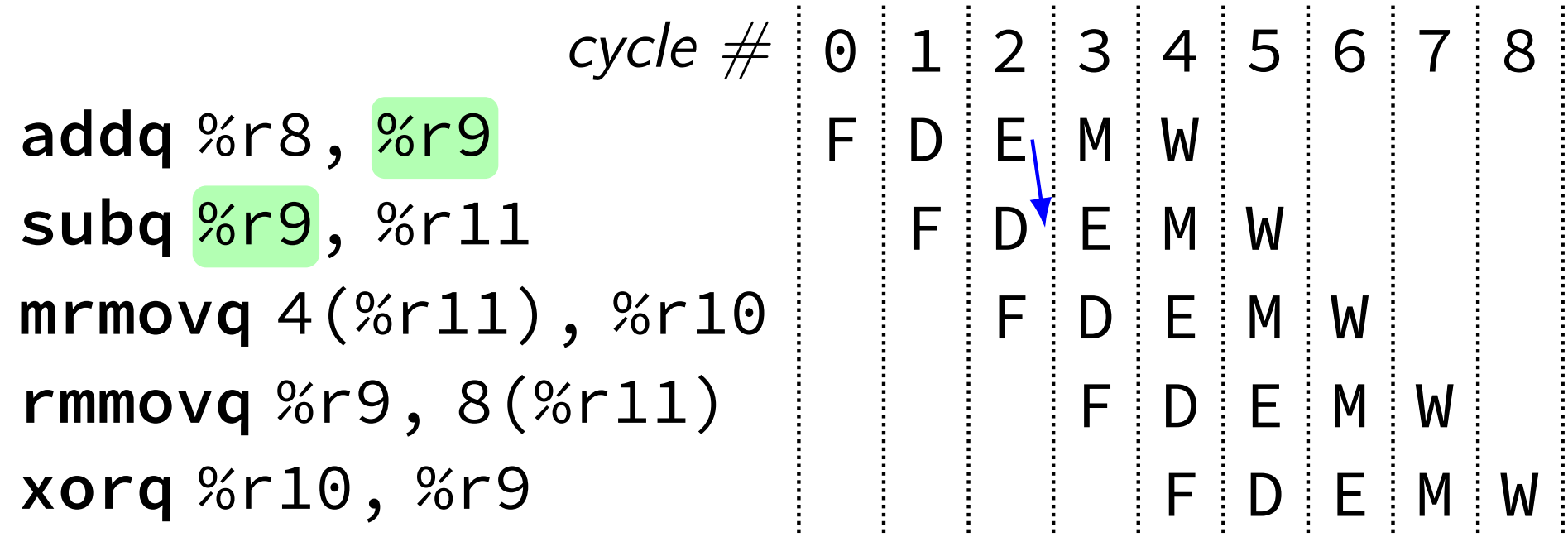
forward alternative



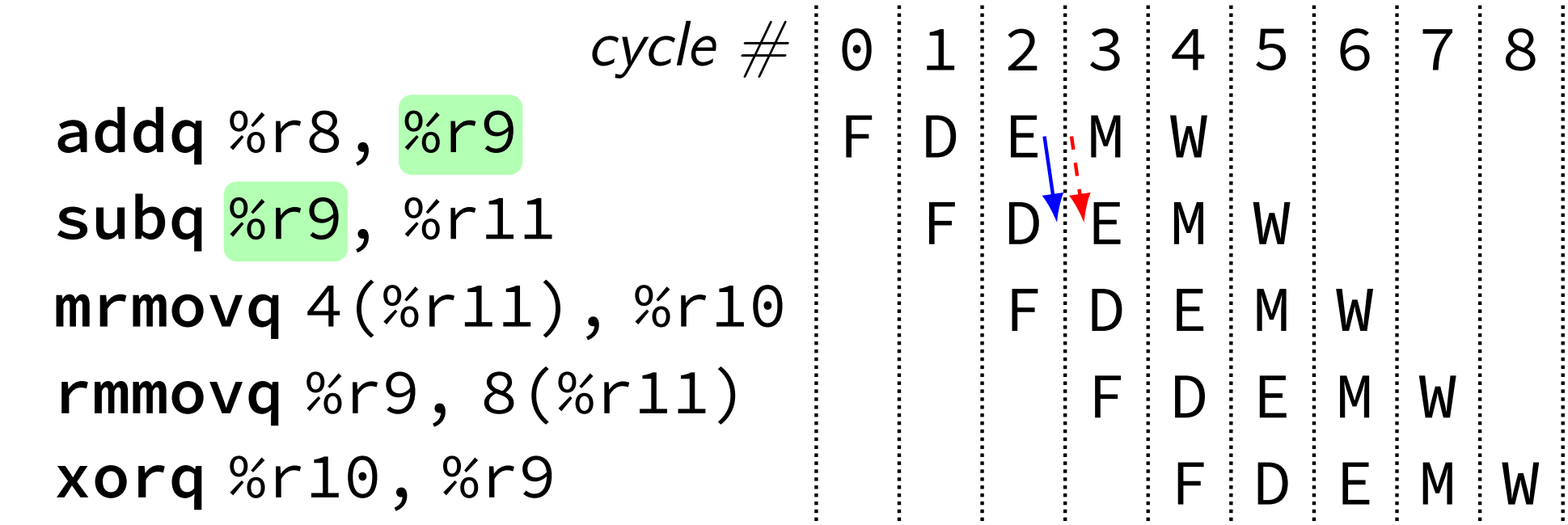
some forwarding paths

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

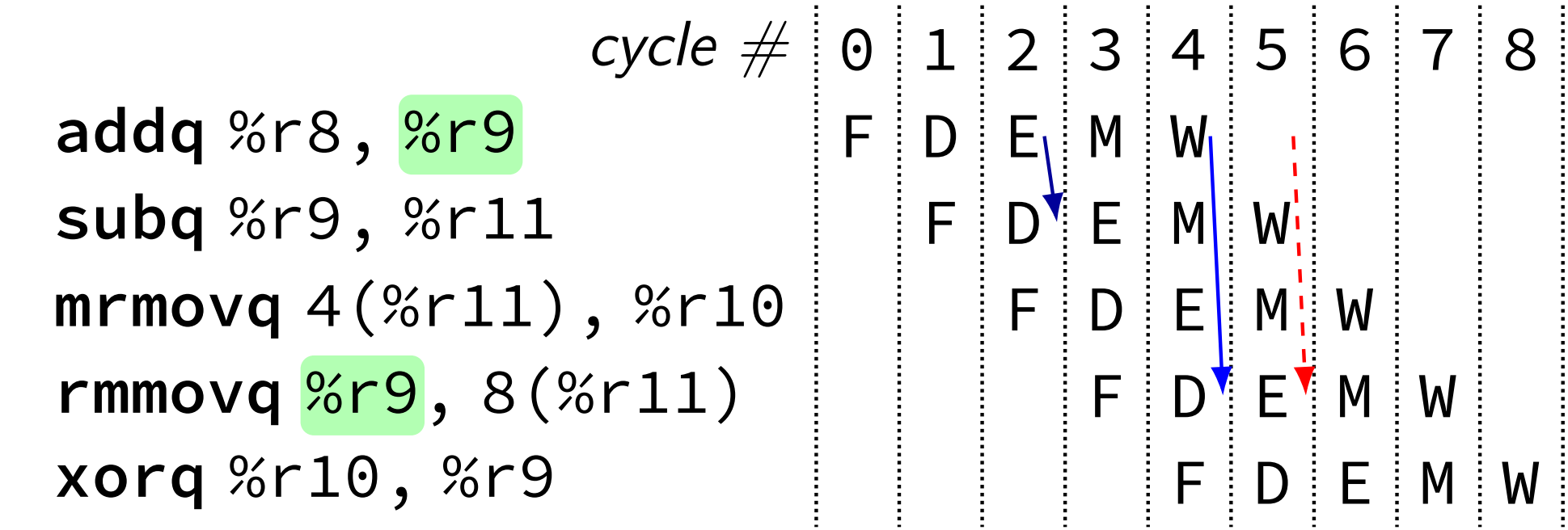
some forwarding paths



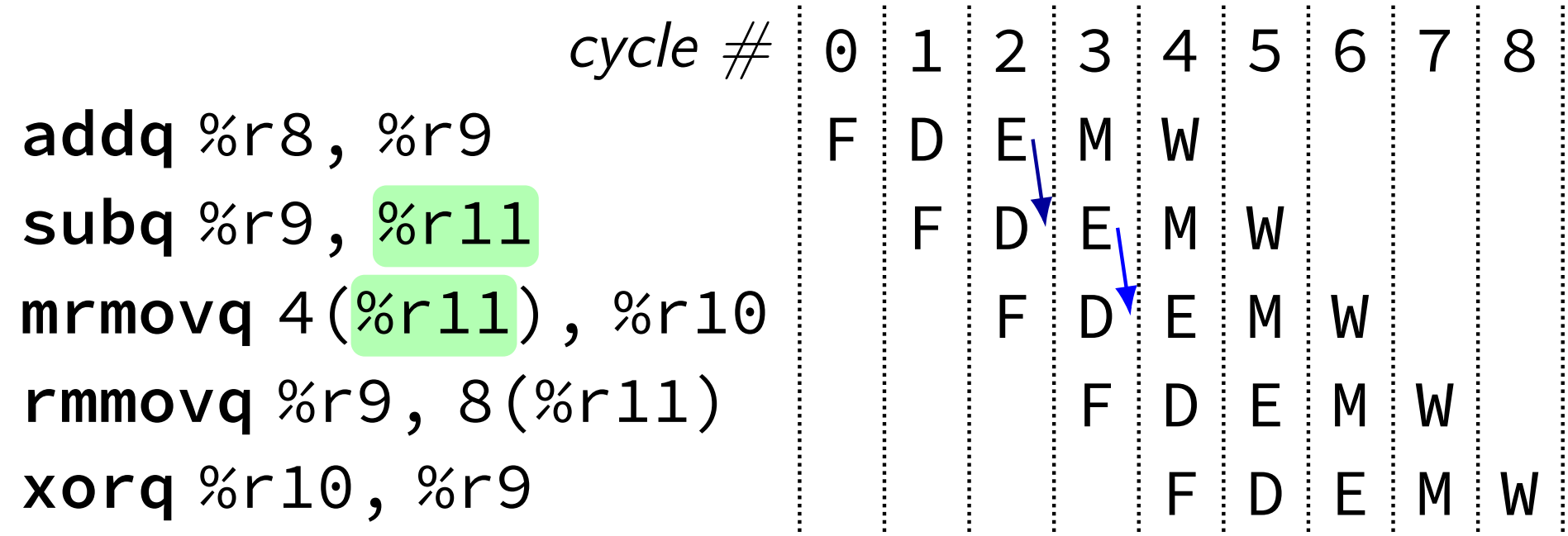
some forwarding paths



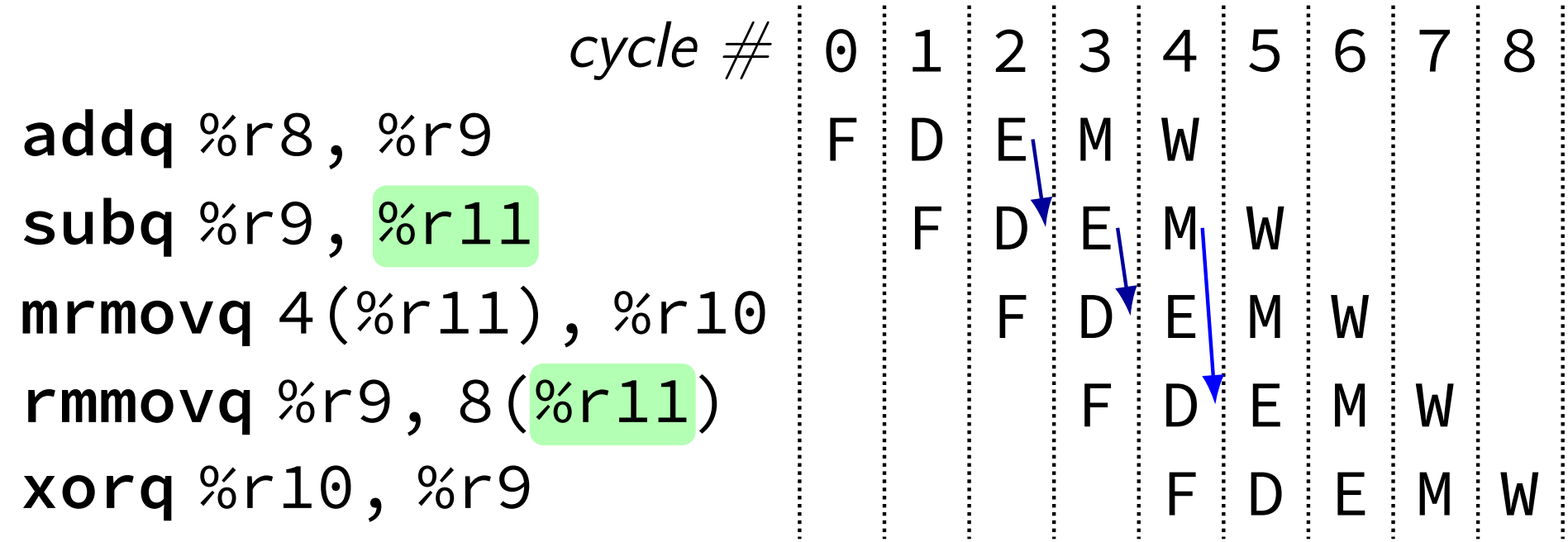
some forwarding paths



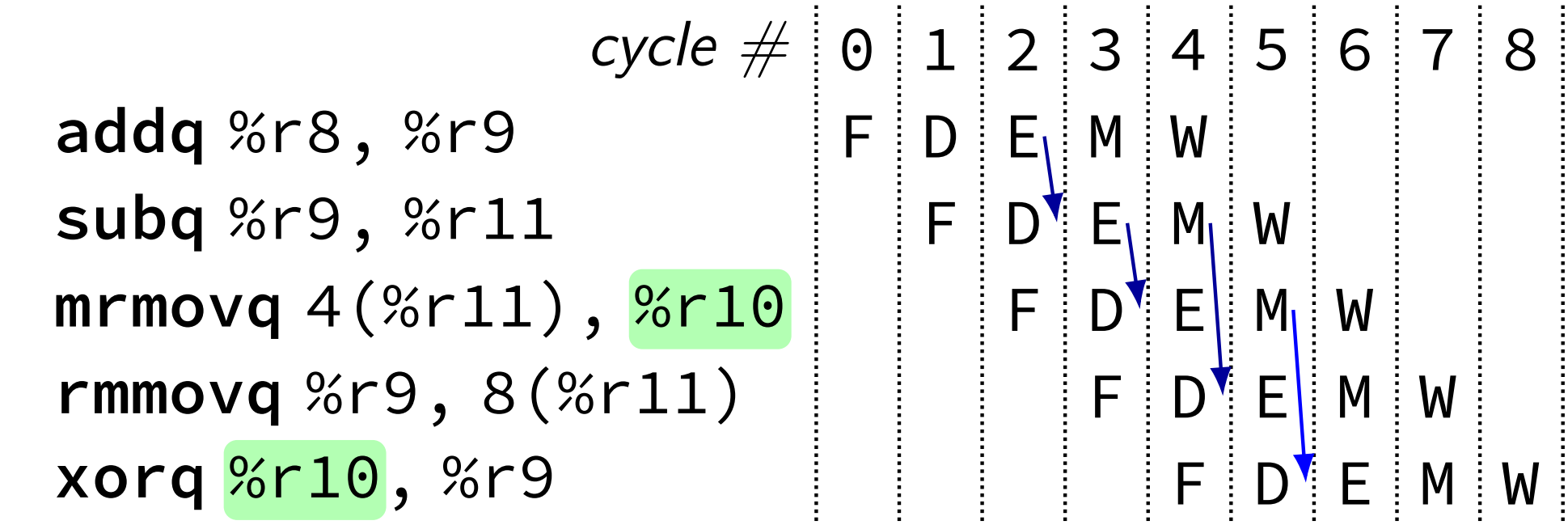
some forwarding paths



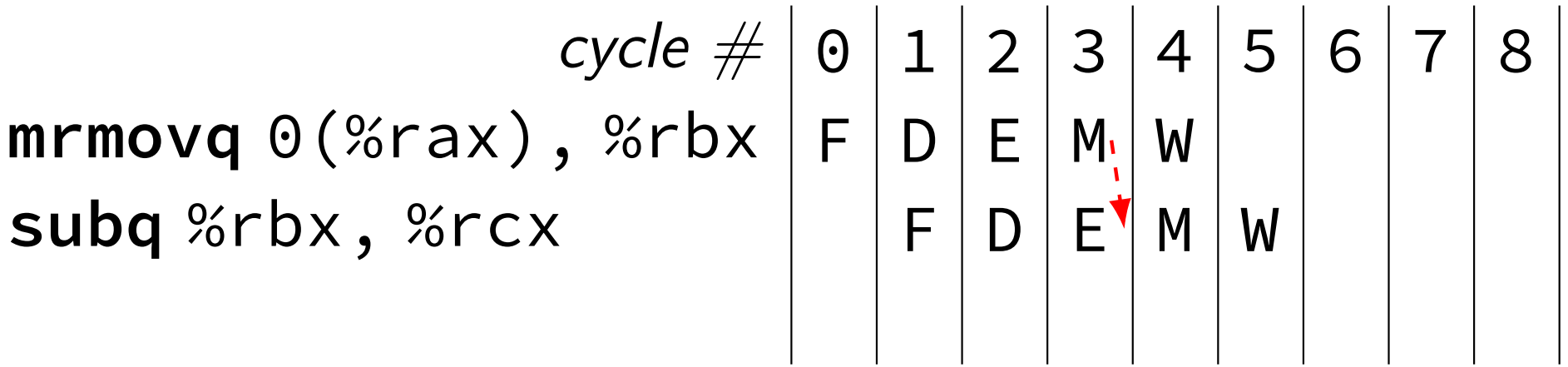
some forwarding paths



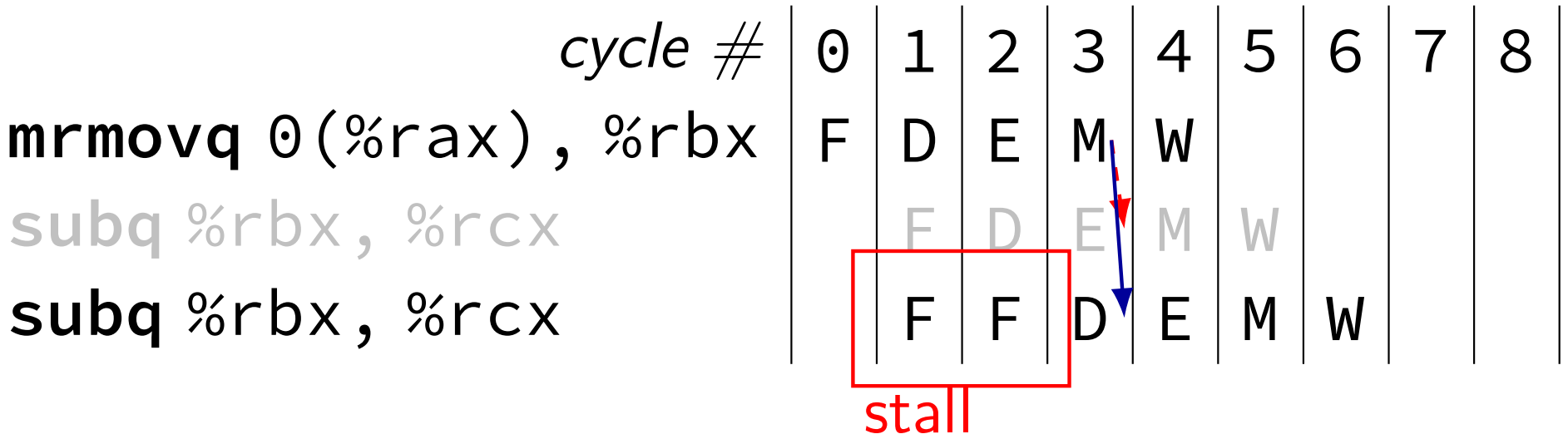
some forwarding paths



unsolved problem



unsolved problem



forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

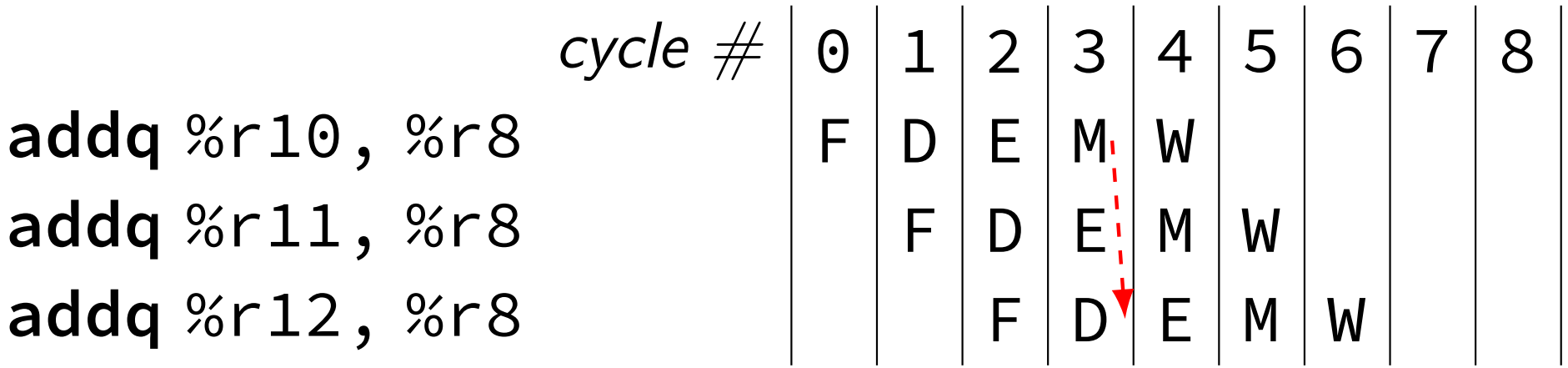
forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

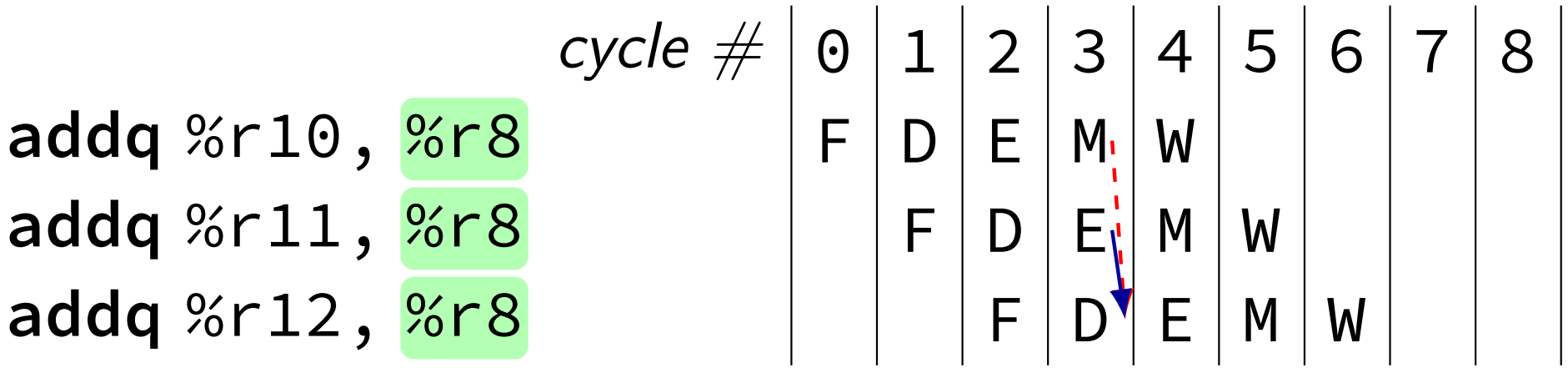
forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

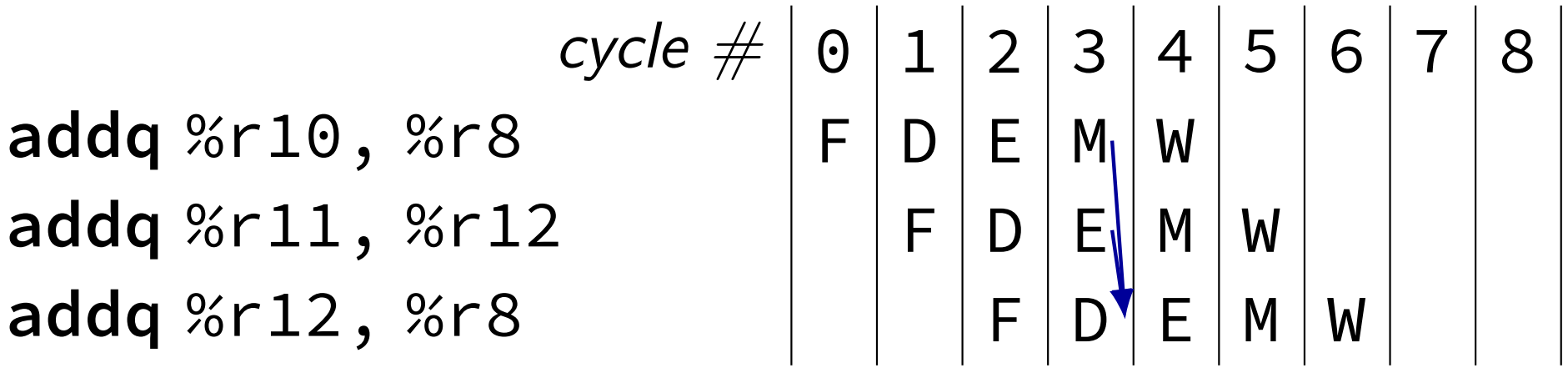
multiple forwarding paths (1)



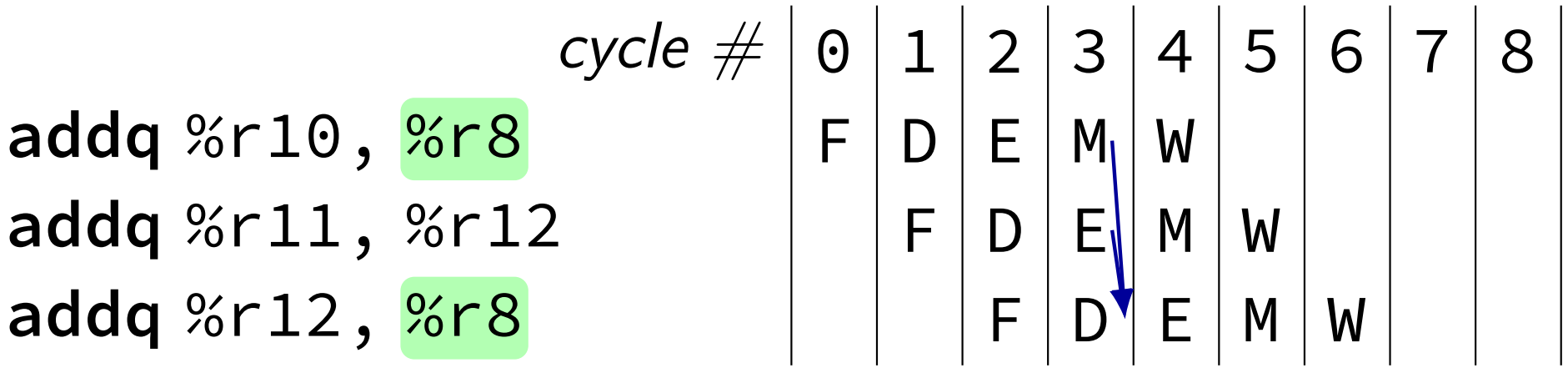
multiple forwarding paths (1)



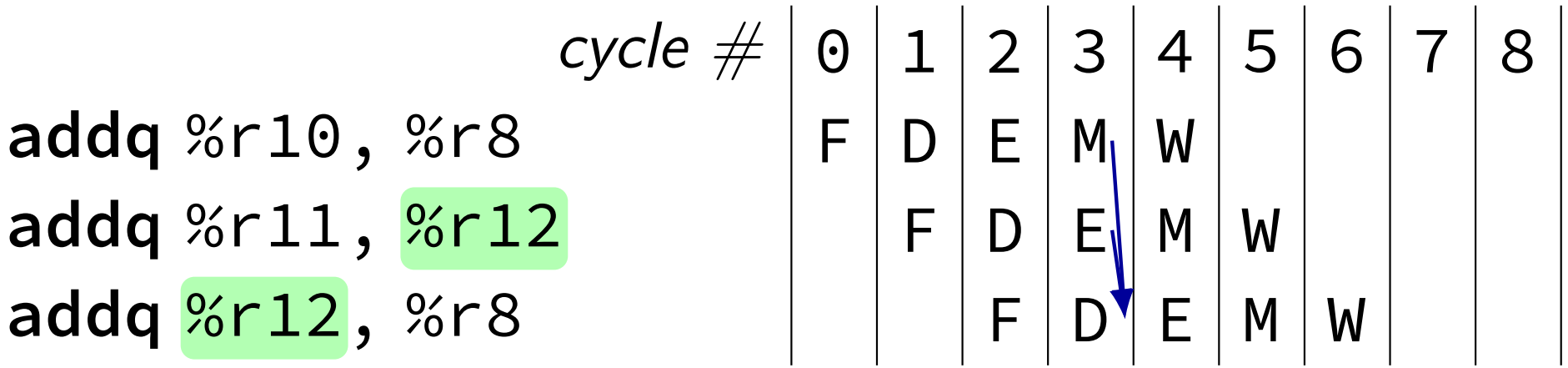
multiple forwarding paths (2)



multiple forwarding paths (2)



multiple forwarding paths (2)



multiple forwarding HCL

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

forwarding after decode

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

forwarding after decode

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

forwarding after decode

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

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

ret control hazard: 3 cycles of stalling

pipelined control costs

how much faster than single-cycle processor?

at most five times faster

depends on HW details:

- how expensive is forwarding logic? (new MUXes on critical path)
- how well balanced are the stages?

depends on what programs we run:

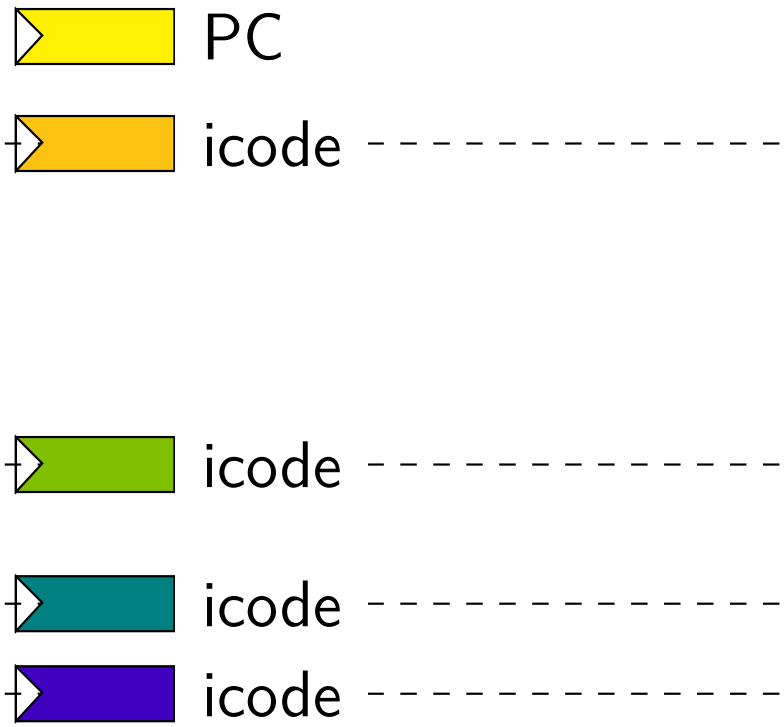
- how many mispredicted jumps?
- how many rets?
- how many load/use hazards?

HCL2D addq unpipelined

```
wire rA : 4, rB : 4, dstE : 4;
wire valA : 64, valB : 64, valE : 64;
register xF {
    pc : 64 = 0;
};
/* Fetch+PC Update*/
pc = F_pc; x_pc = pc + 2;
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode */
reg_srcA = rA; reg_srcB = rB; dstE = rB;
valA = reg_outputA; valB = reg_outputB;
/* Execute */
valE = valA + valB;
/* Writeback */
reg_dstE = dstE; reg_inputE = valE;
```

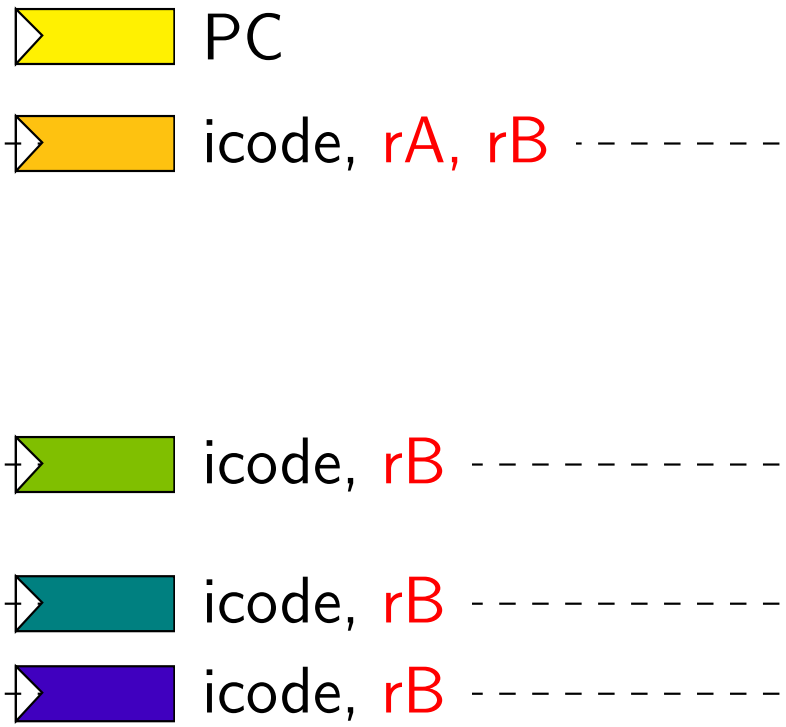
addq pipeline registers

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 $\leftarrow$ valB + valB
memory	
write back	$R[$ rB $] \leftarrow$ valE



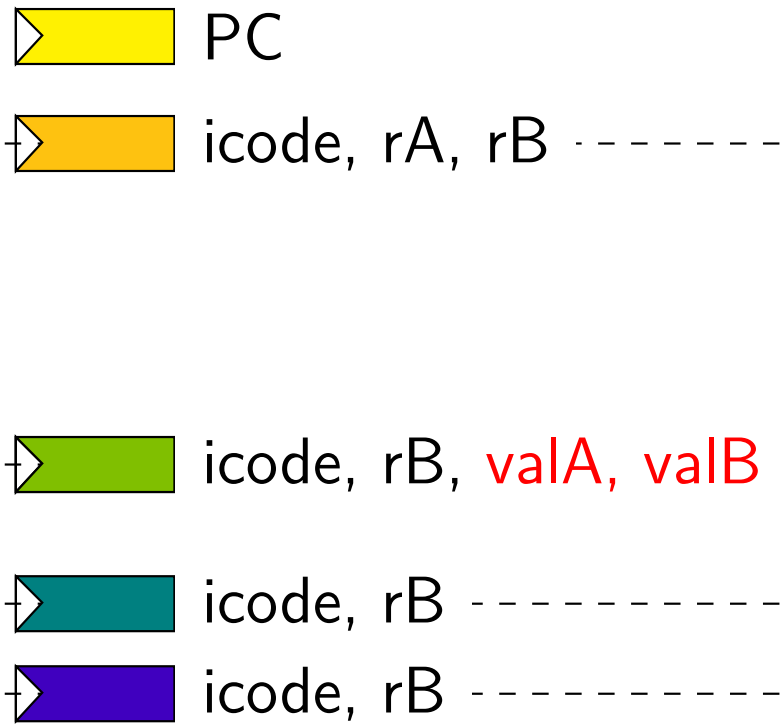
addq pipeline registers

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 $\leftarrow$ valB + valB
memory	
write back	R[rB] $\leftarrow$ valE



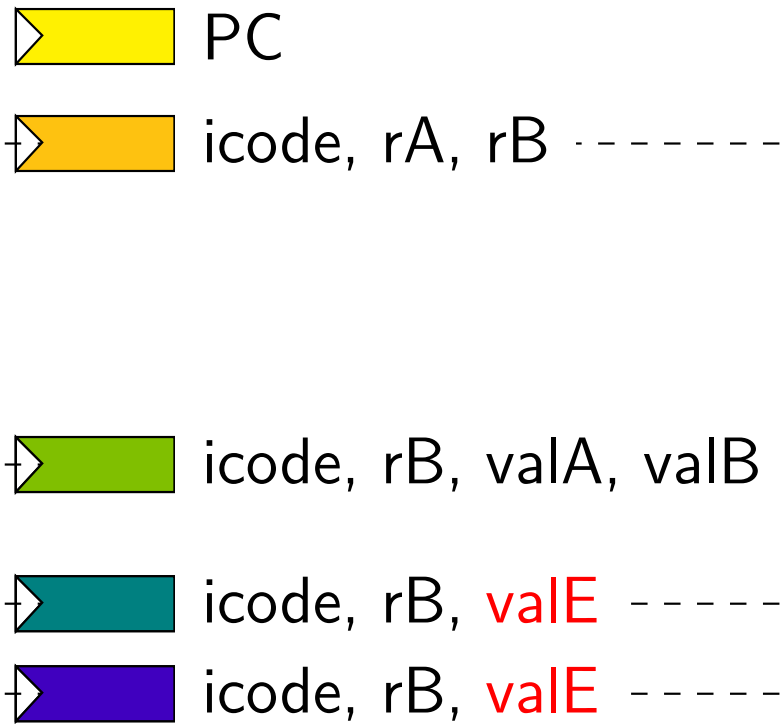
addq pipeline registers

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 $\leftarrow$ valB + valB
memory	
write back	R[rB] $\leftarrow$ valE



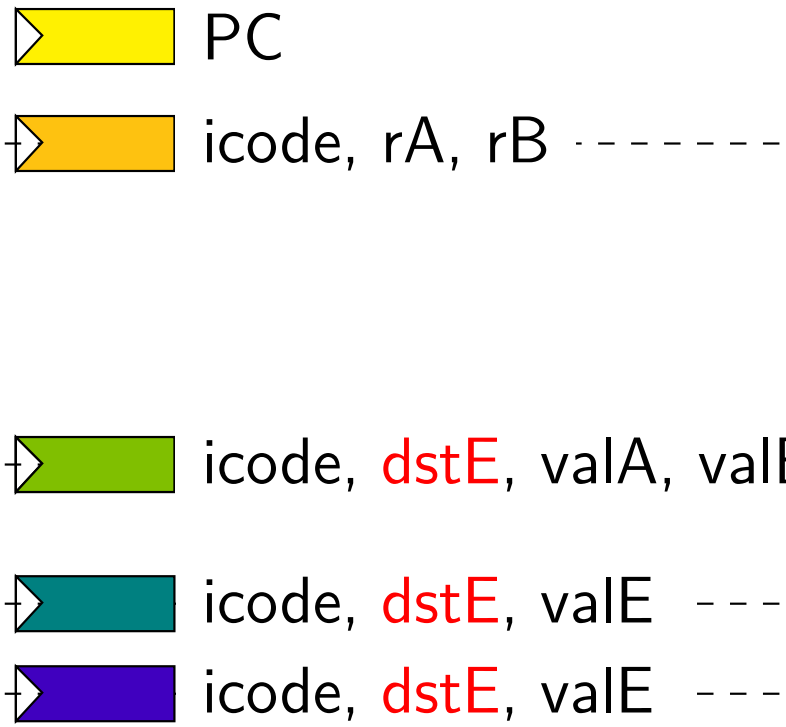
addq pipeline registers

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 $\leftarrow$ valB + valB
memory	
write back	$R[$ rB $] \leftarrow$ valE



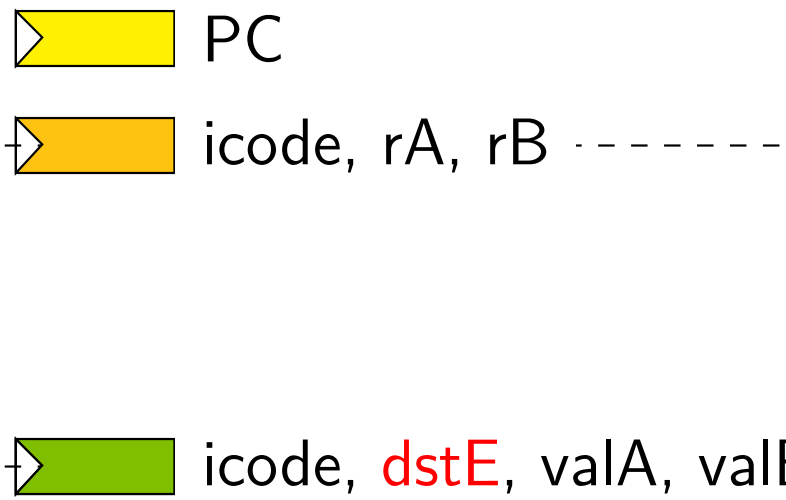
addq pipeline registers

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 $\leftarrow$ valB + 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 $\leftarrow$ valB + valB
write back	R[dstE] $\leftarrow$ valE



redundant with rB + icode
but will make handling data hazards easier

HCL2D pipeline registers

```
register xF {
    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 */
```

HCL2D: Fetch/Decode

unpipelined

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

pipelined

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

HCL2D pipelining debugging: intro

debugging pipelines is consistently one of the biggest sources of difficulty in this class

notably: big drain on TA time

HCL2D pipeline debugging advice

draw a picture of the state of the instructions

get `-d` output

check each stage of the broken instruction

be mindful: there might be a forwarding/hazard handling problem

exercise: stalls and forwarding (1)

```
mrmovq 0(%rax), %rbx
addq %rax, %rcx
subq %rbx, %rcx
rmmovq %rcx, 0(%rax)
```

Are there stalls? Where does forwarding happen?

exercise: stalls and forwarding (2)

```
mrmovq 0(%rax), %rbx
call foo
foo: addq %rbx, %rcx
rmmovq %rcx, 0(%rcx)
ret
```

Are there stalls? Where does forwarding happen?

exercise: stalls and forwarding (3)

```
addq %rax, %rax
jne foo // taken
foo: mrmovq 0(%rax), %rbx
addq %rbx, %rcx
mrmovq 0(%rbx), %rcx
```

Are there stalls? Where does forwarding happen?

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