



# last time

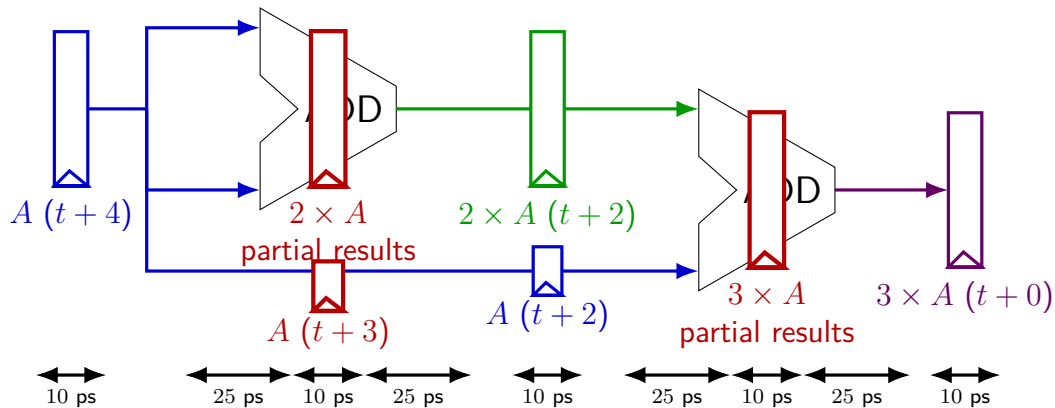
## pipelining idea

assembly line of circuit “stages”  
registers hold value for stage to work  
on rising edge — take next value

## diminishing returns from register delays

stage time = computation + register delays  
stage time determines cycle time

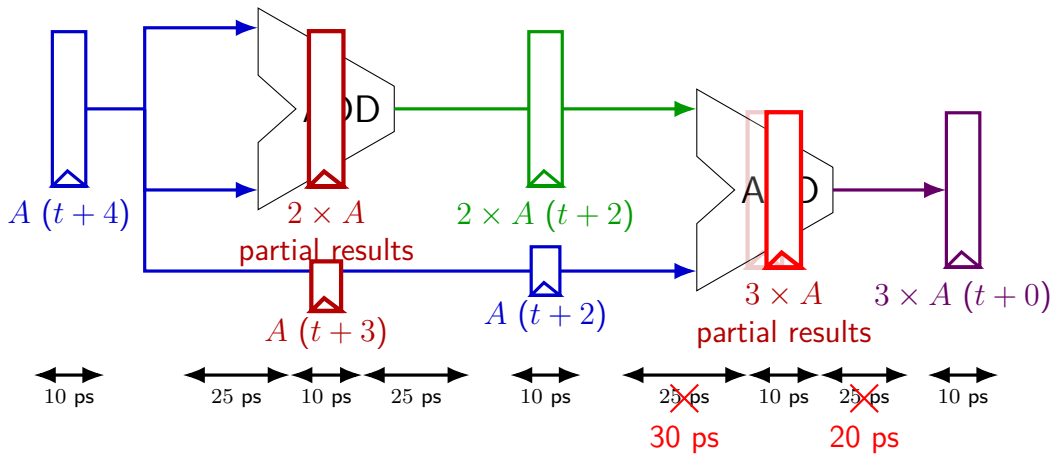
# deeper pipeline



Problem: How much faster can we get?

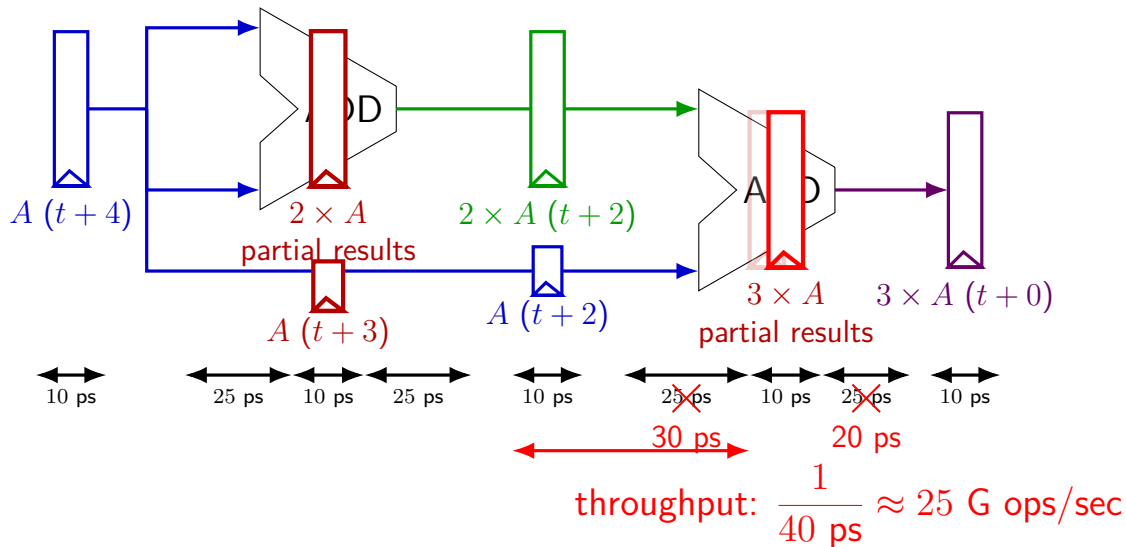
Problem: **Can we even do this?**

# deeper pipeline



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

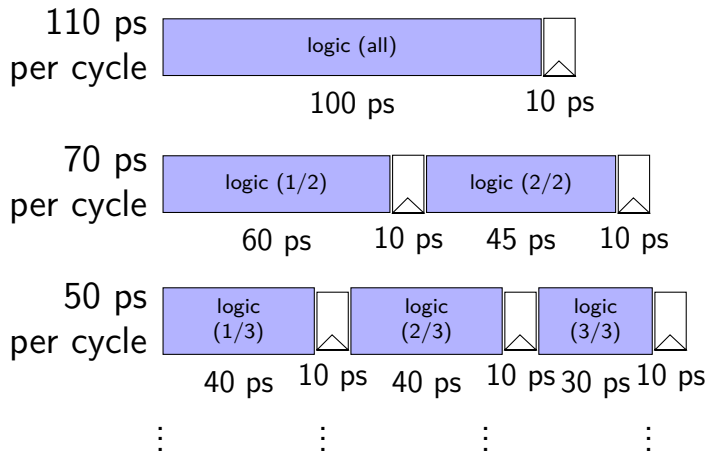
# deeper pipeline



# diminishing returns: uneven split

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

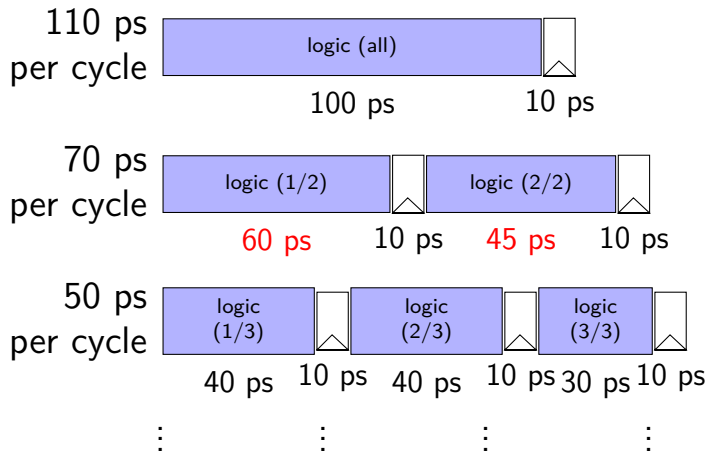
Probably not...



# diminishing returns: uneven split

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

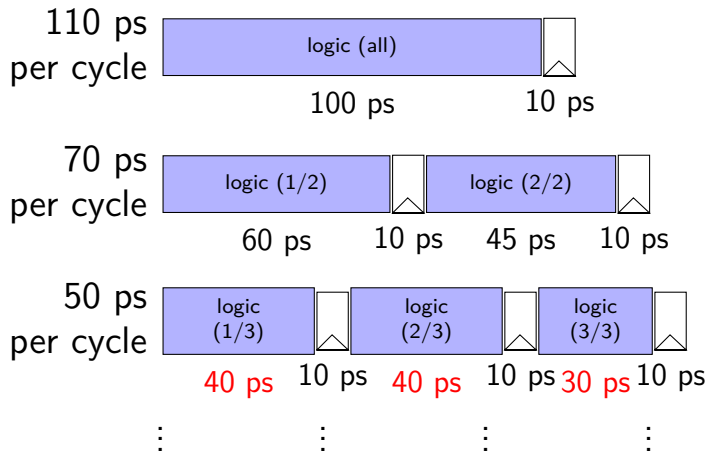
Probably not...



# 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

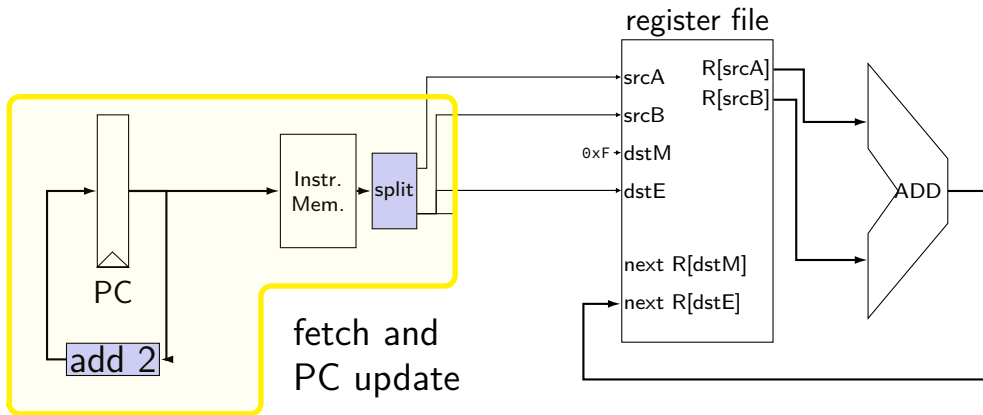
Writeback: write register file

5 stages

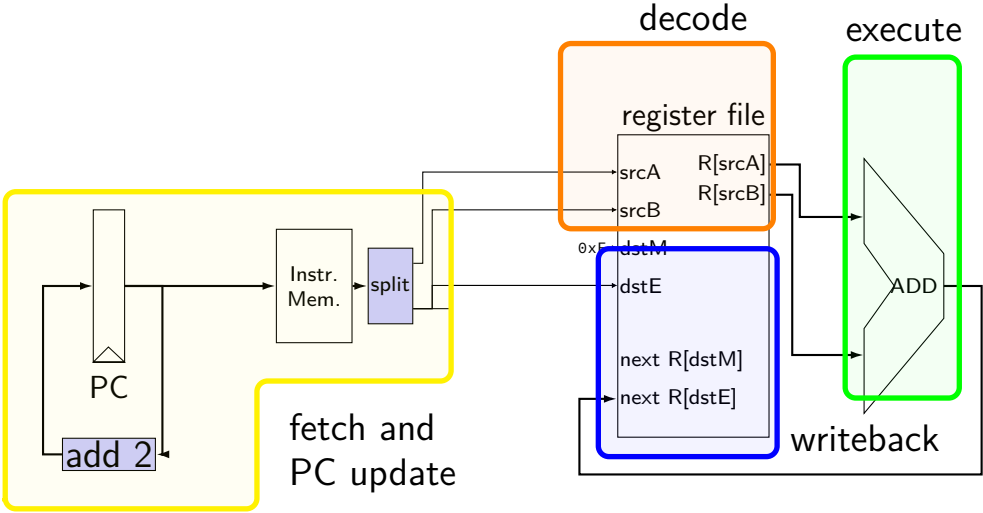
one instruction in each

compute next to start immediately

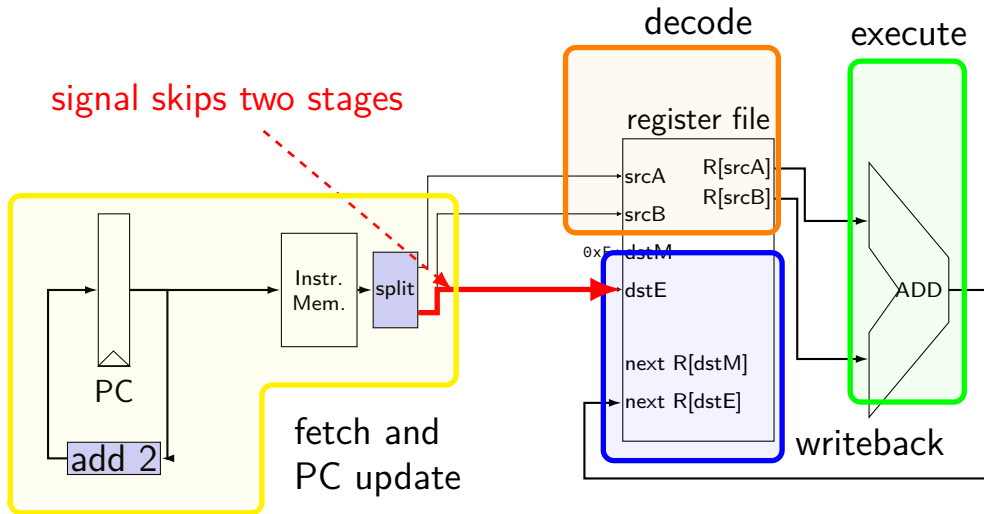
# addq CPU



# addq CPU

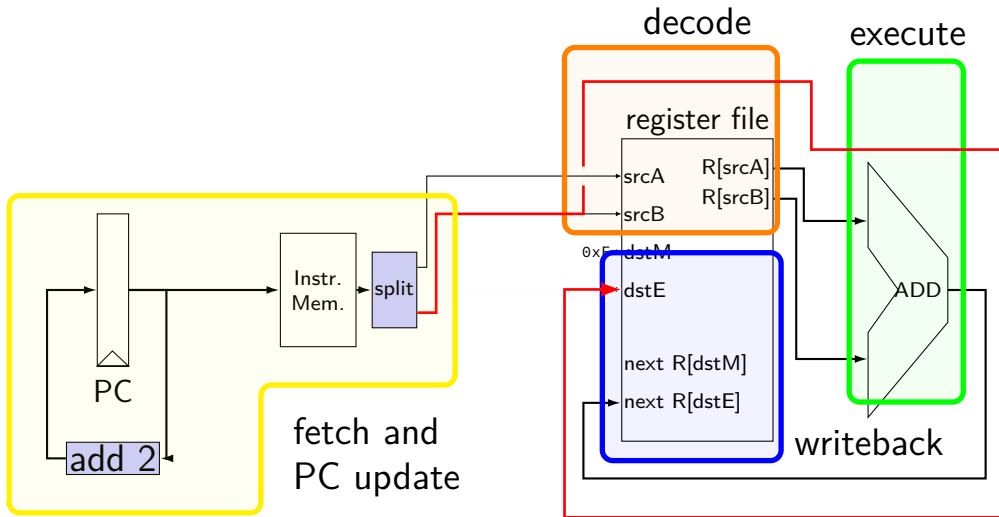


# addq CPU

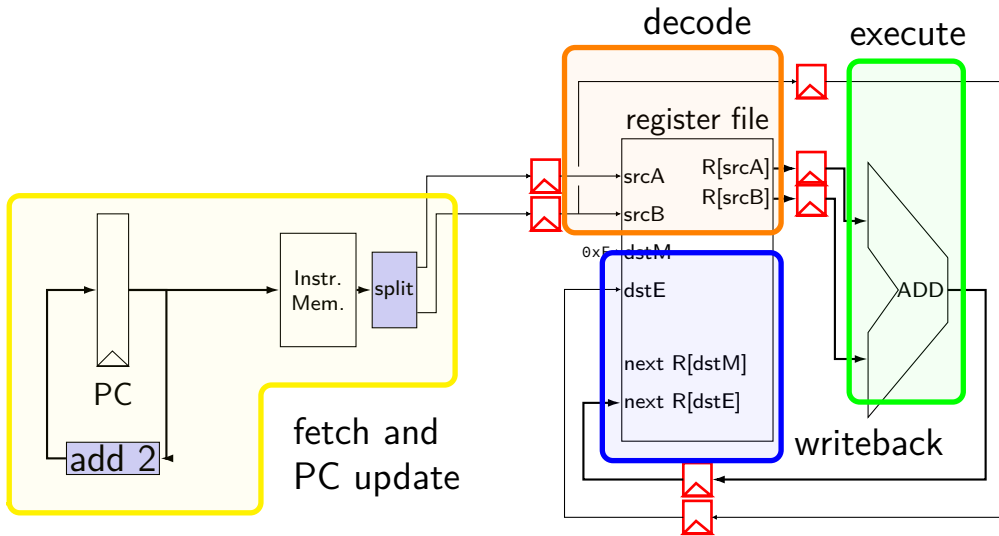




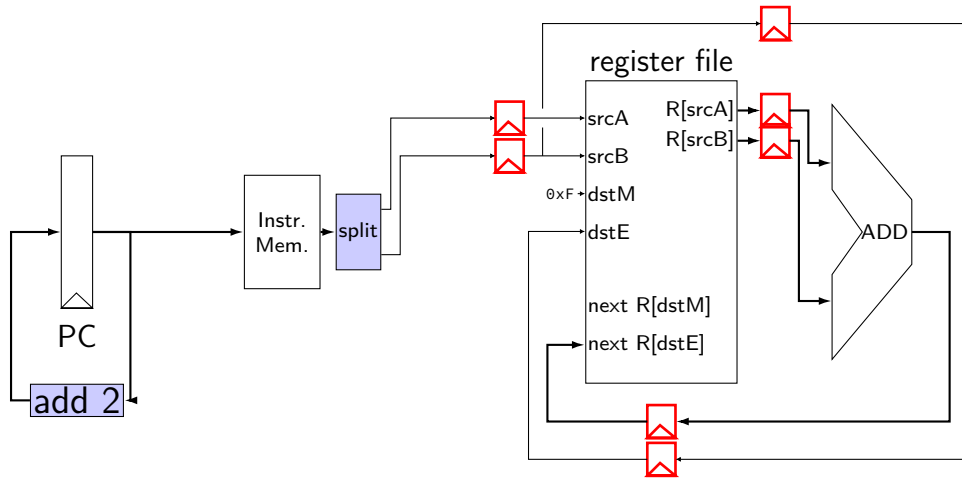
# addq CPU



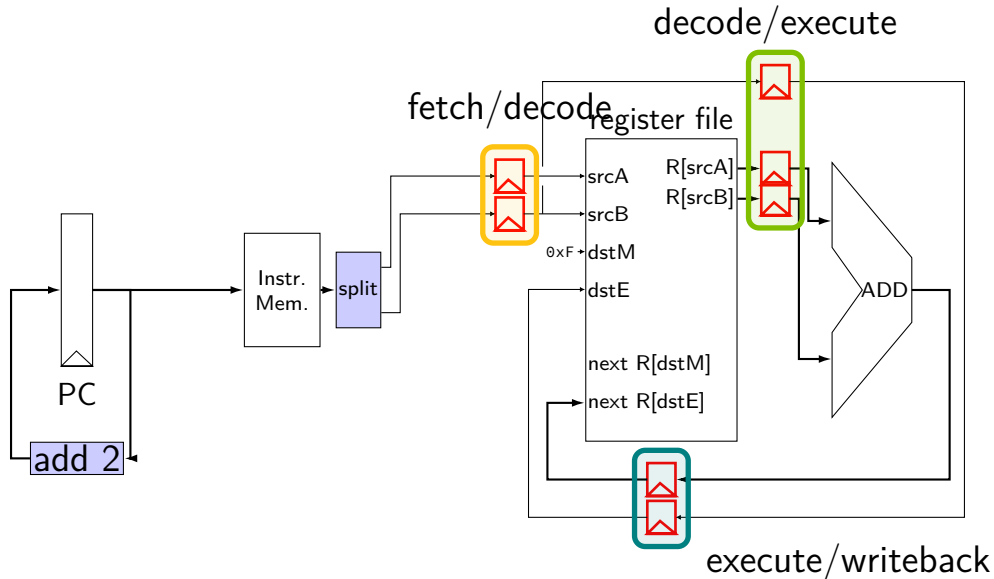
# pipelined addq processor



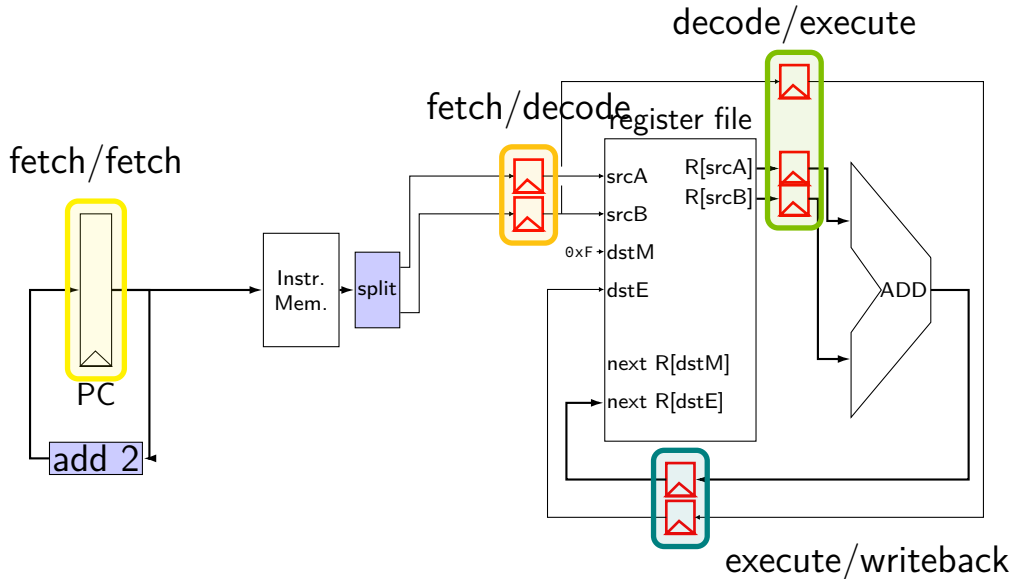
# pipelined addq processor



# pipelined addq processor



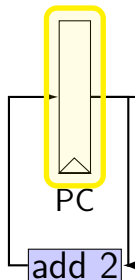
# pipelined addq processor



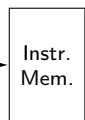
# addq execution

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

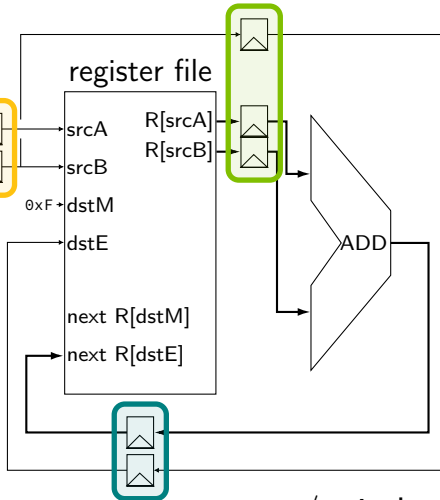
fetch/fetch



fetch/decode



decode/execute

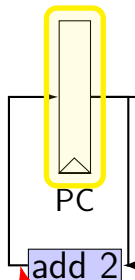


execute/writeback

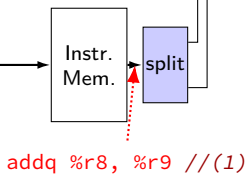
# addq execution

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

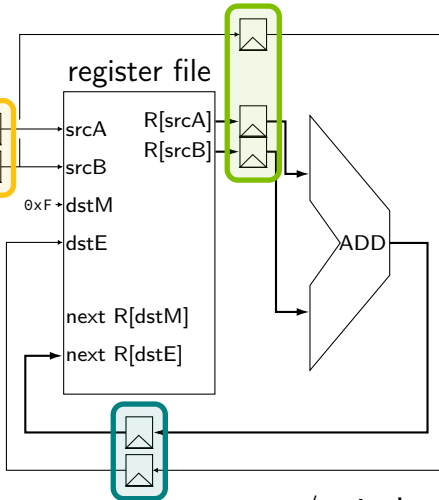
fetch/fetch



fetch/decode



decode/execute



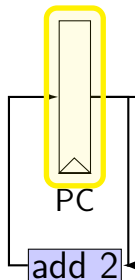
execute/writeback

address of (2)

# addq execution

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

fetch/fetch



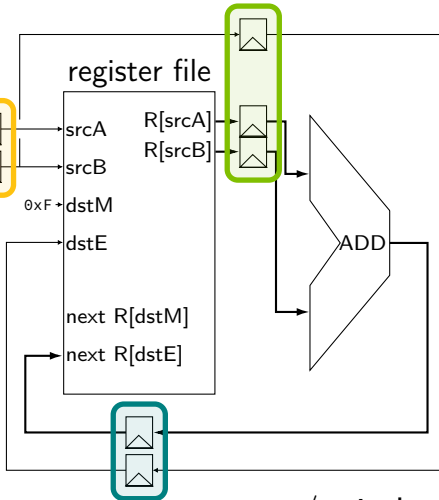
reg #s 8, 9 from (1)

fetch/decode

addq %r10, %r11 // (2)



decode/execute

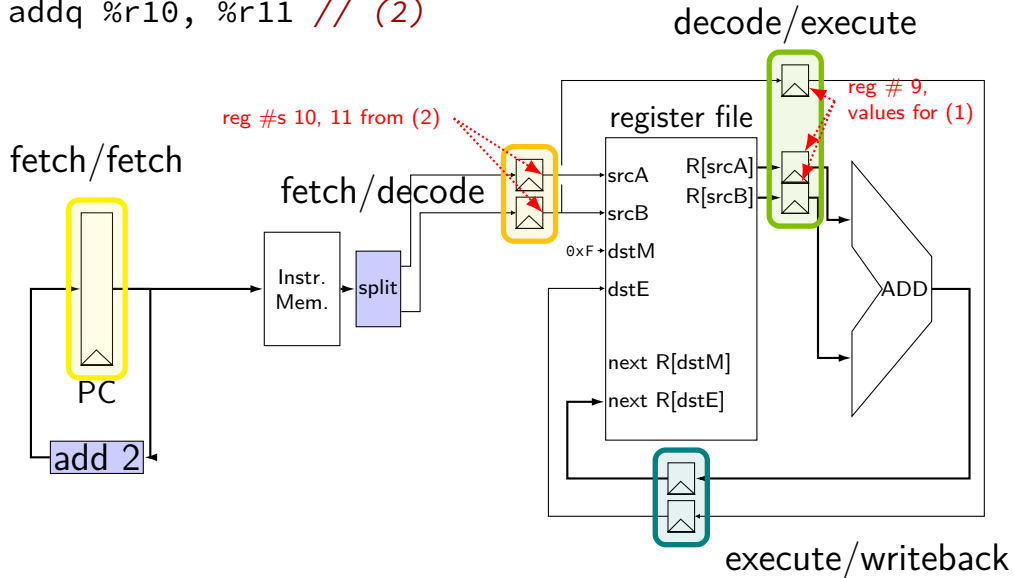


execute/writeback



# addq execution

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

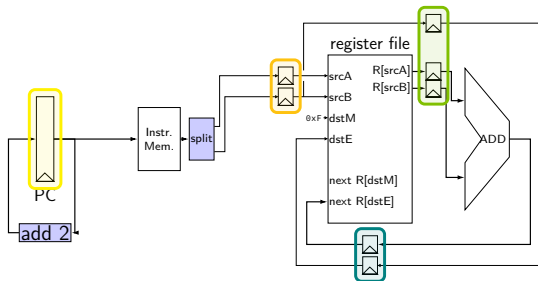


# addq processor timing

```
// 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

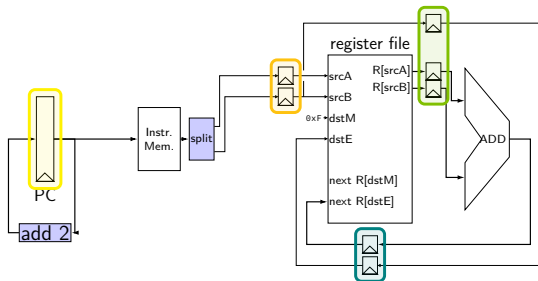


# addq processor timing

```
// 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

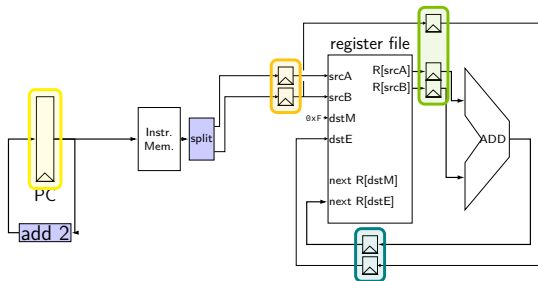


# addq processor timing

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

```
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```

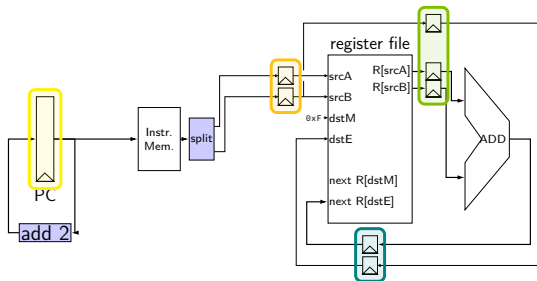
	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8



# addq processor timing

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

```
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```

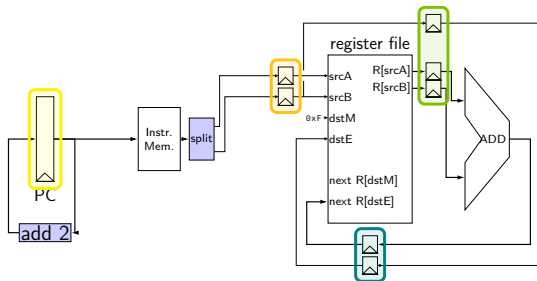


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

# addq processor timing

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

```
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```

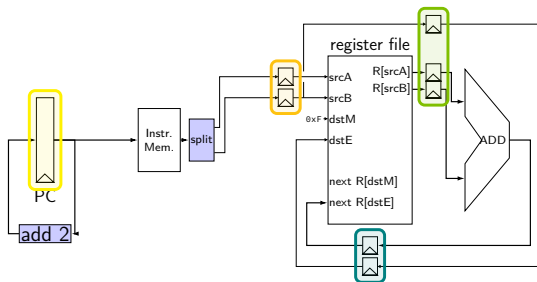


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

# 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

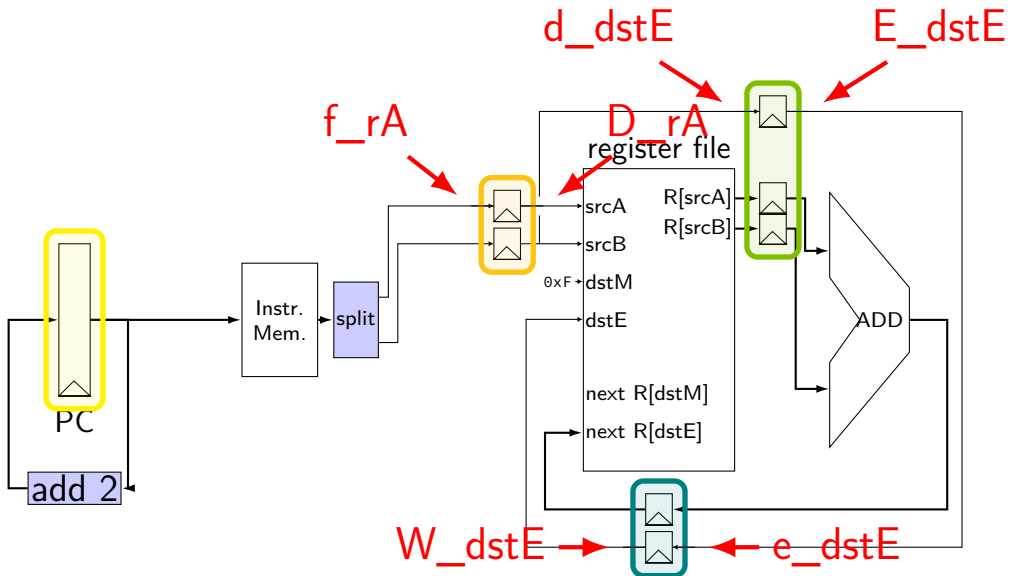
add up everything but add 2 (critical (slowest) path)

pipelining: 1 instruction per 200 ps + pipeline register delays

slowest path through stage + pipeline register delays

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

# 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;
```

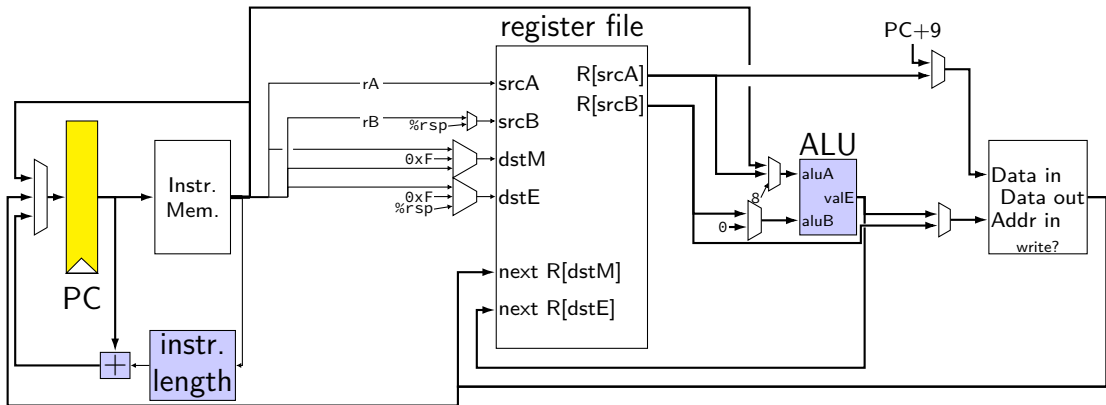
pipelined

```
/* 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;
```

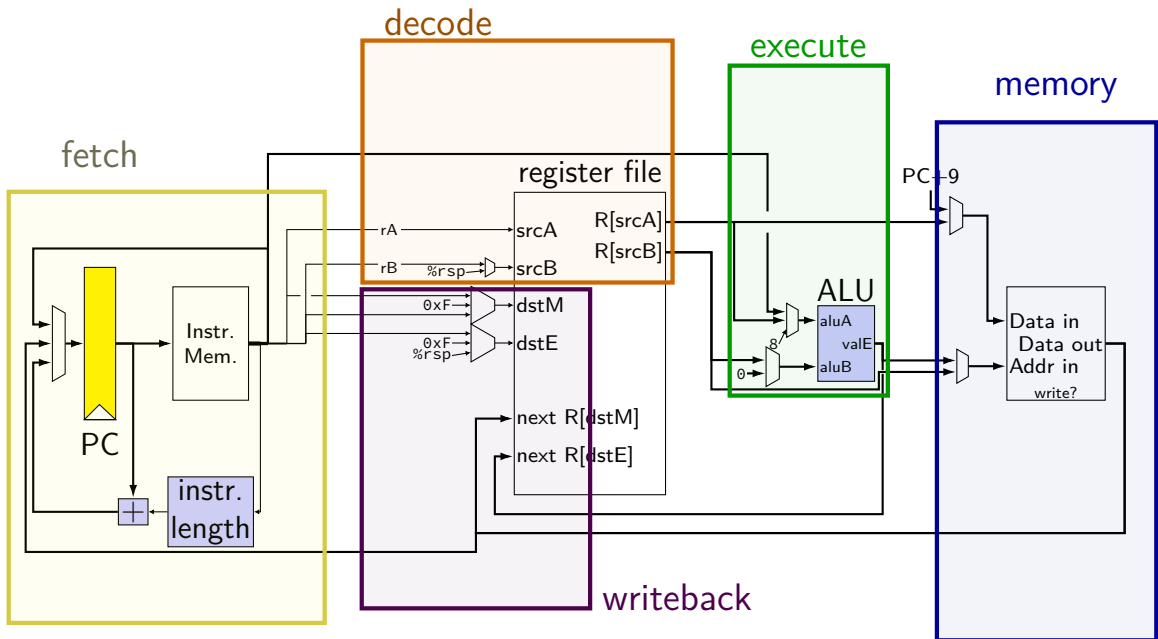
# addq pipeline registers

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

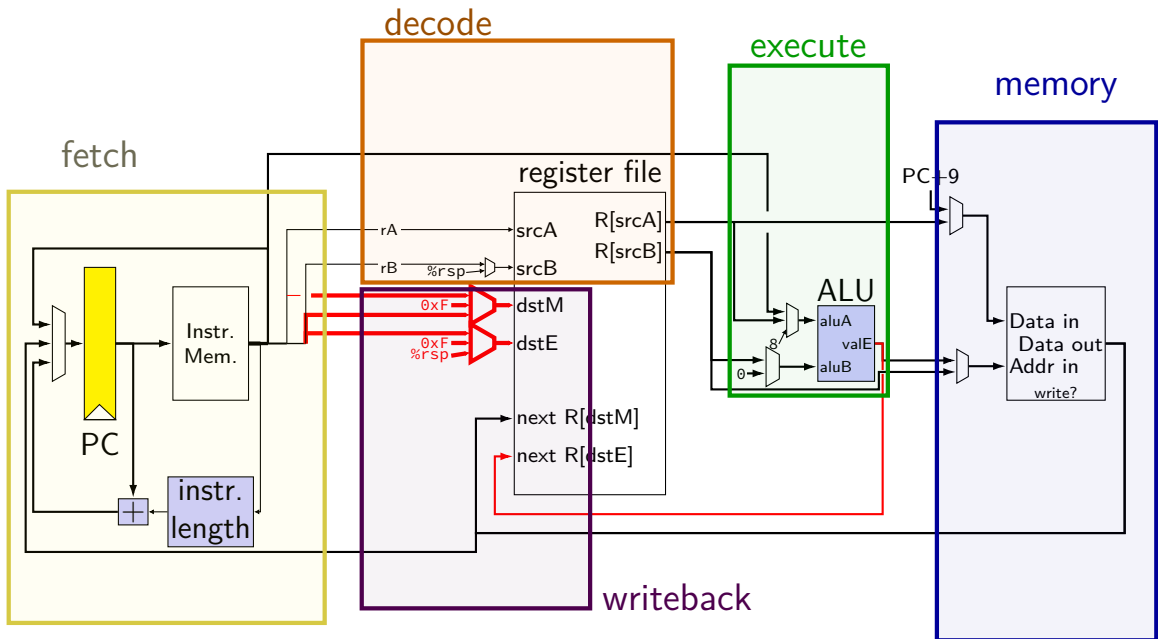
# SEQ without stages



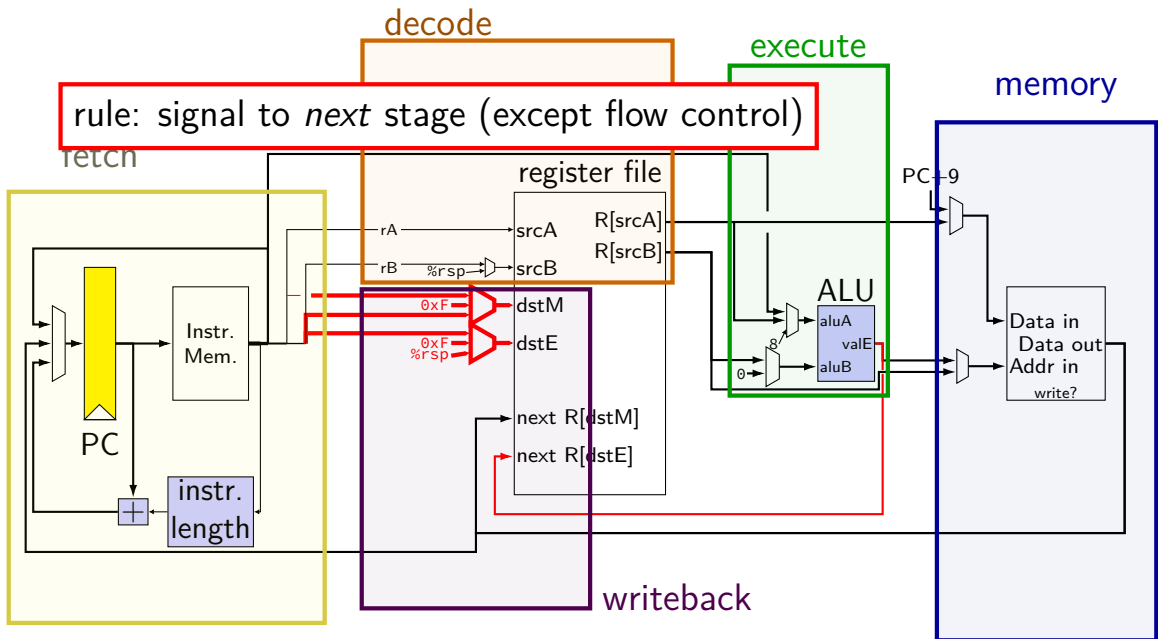
# SEQ with stages



# SEQ with stages

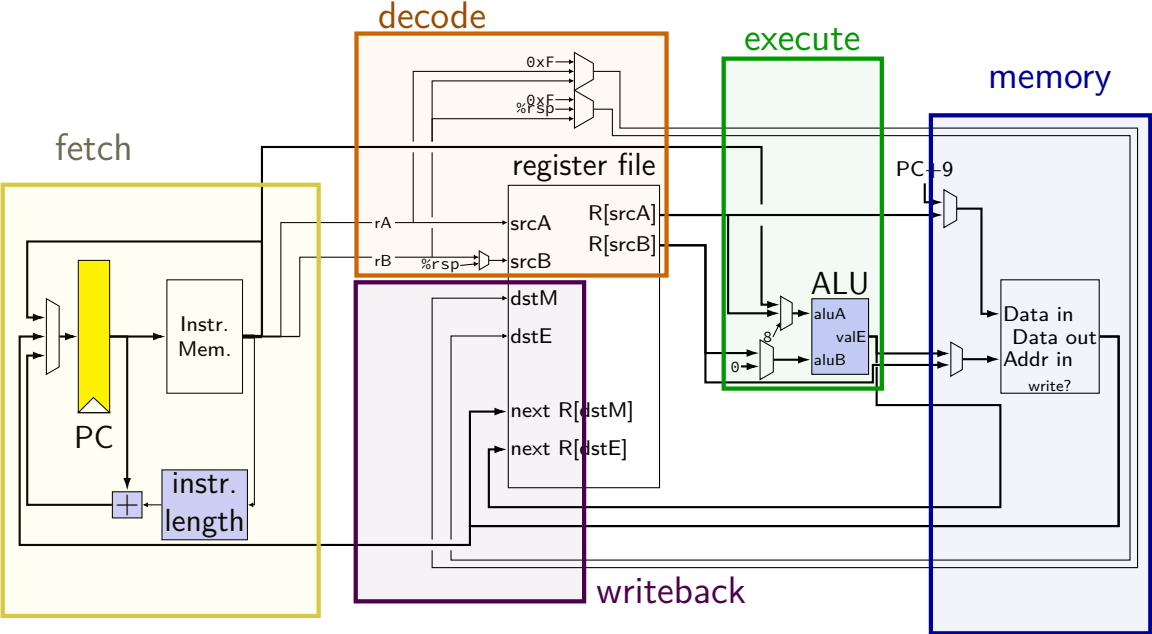


# SEQ with stages

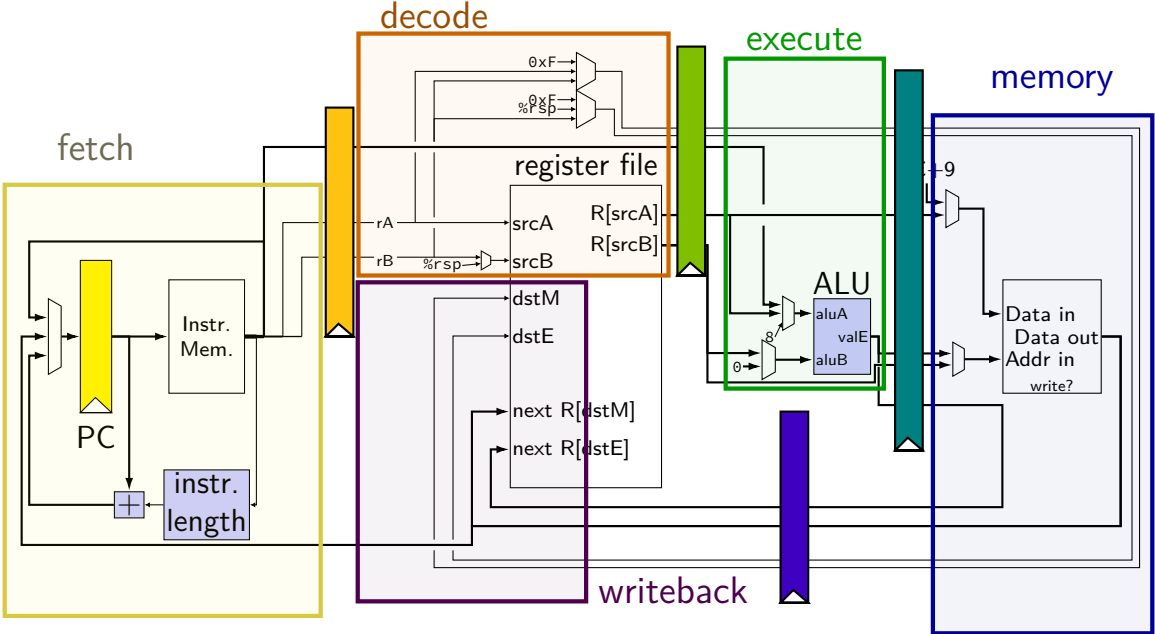




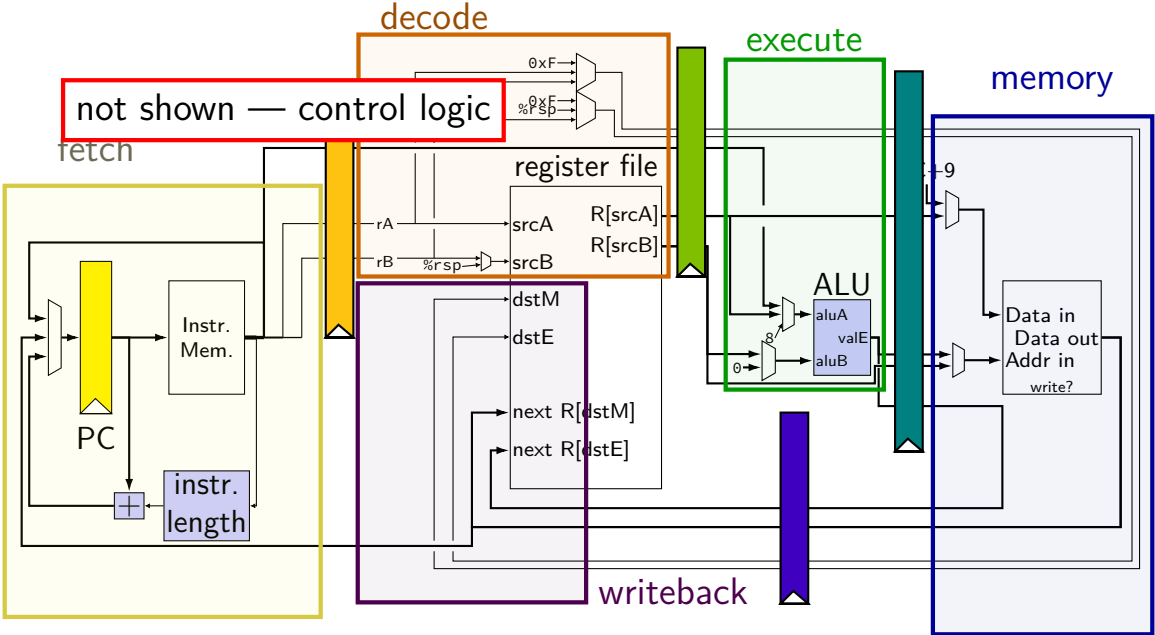
# SEQ with stages (actually sequential)



# 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**

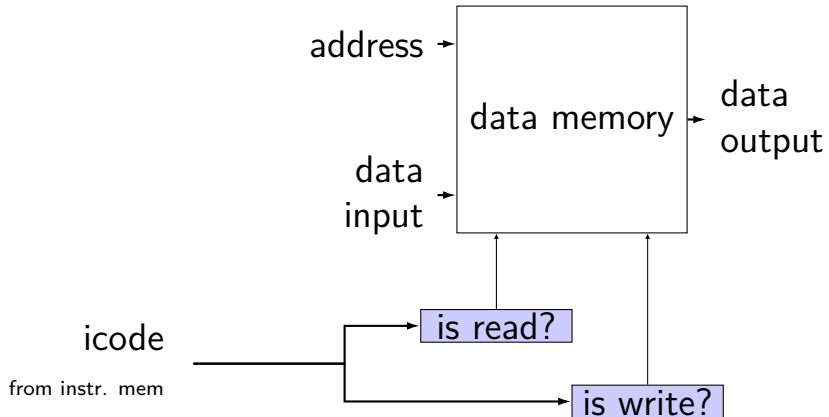
e.g. decode: send to execute via pipeline registers (d\_icode, ...)

exceptions: **deliberate sharing** between instructions

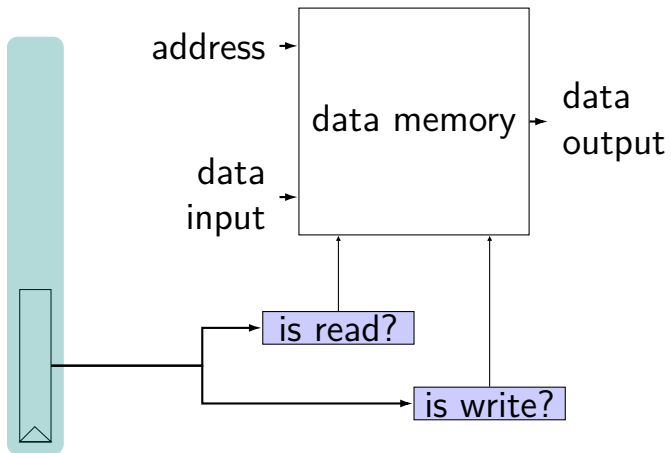
via register file/memory/etc.

via control flow instructions

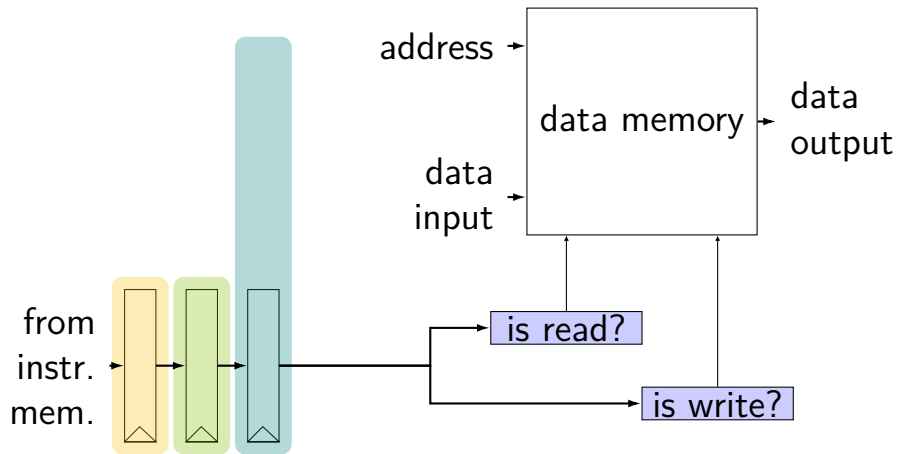
# memory read/write logic



# 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;
];
```

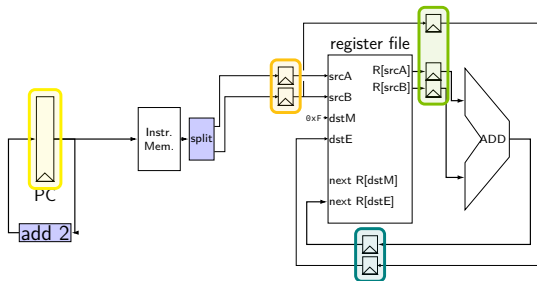
## memory read/write: PIPE code

```
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.
```

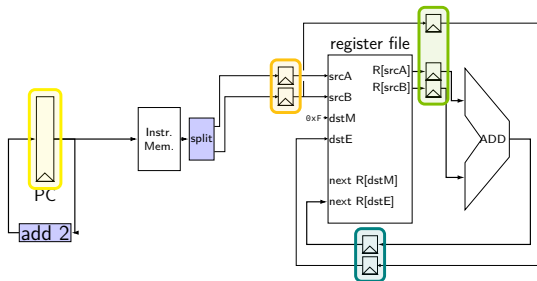
```
addq %r8, %r9
addq %r9, %r8
addq ...
addq ...
```



	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 for (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**

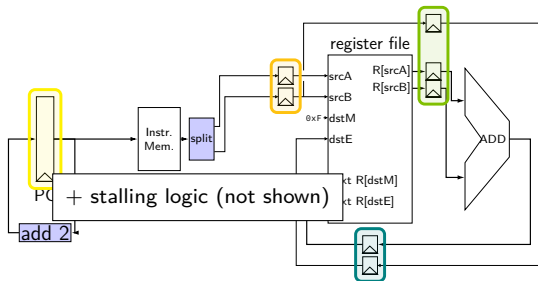
extra logic:

- 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
```

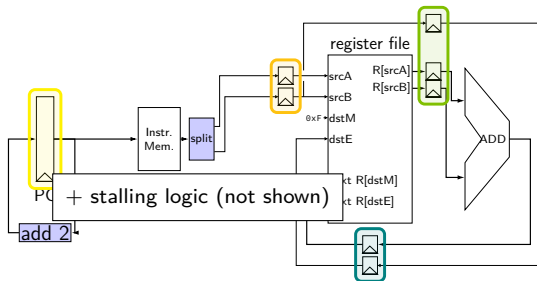


	fetch	fetch→decode		decode→execute			execute→writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2*	8	9					
2	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4	0x4	9	8	---	---	F	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8



# addq processor: data hazard stall

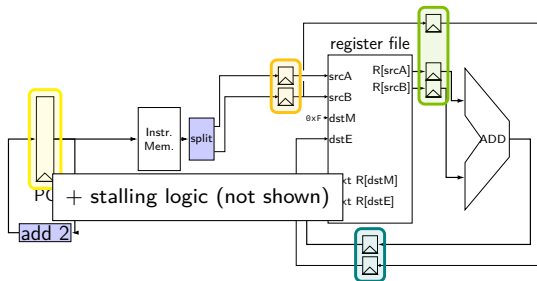
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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
```

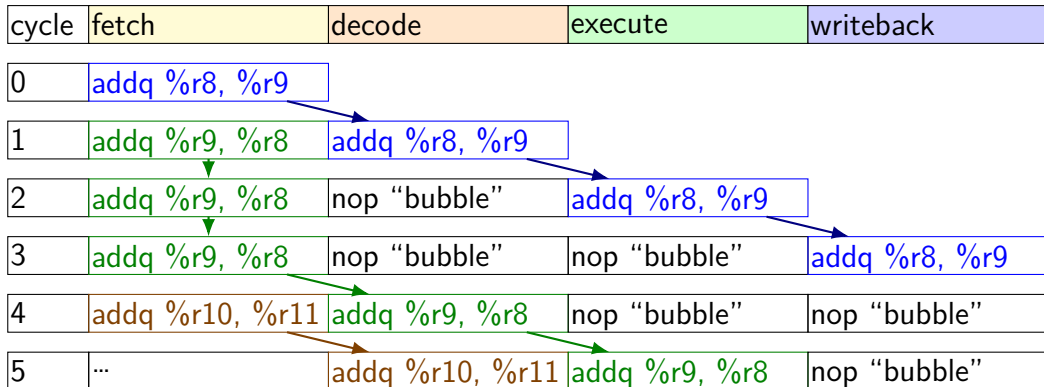


	fetch	fetch→decode		decode→execute			execute→writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2*	8	9					
2	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4	0x4	9	8	---	---	F	---	F
5		10	11	1700	800	8	---	F
6				1000	1100	11	2500	8

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

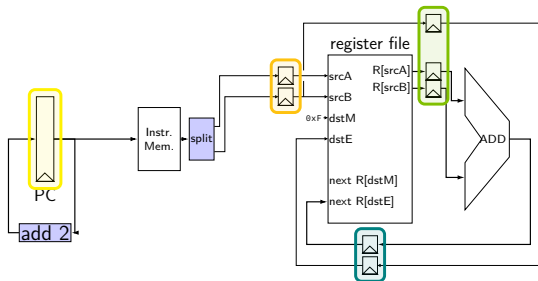
# addq stall

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



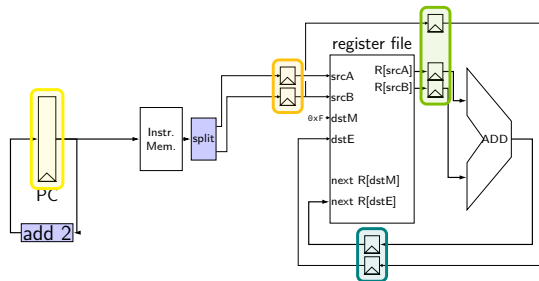
# hazard exercise

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



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

# hazard exercise solution



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

```
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
```

cycle	fetch		fetch→decode		decode→execute			execute→writeback	
	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	wait for two cycles for addq to update SF/ZF							
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
```

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

**execute je instruction (use SF/ZF)**

# 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

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

writeback

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 dep

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

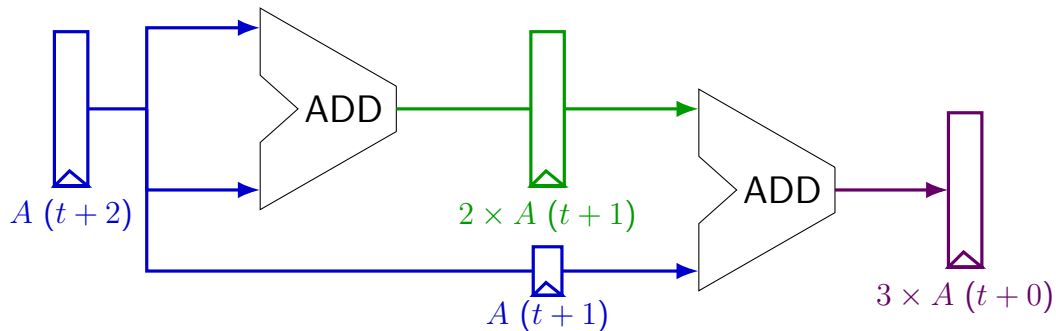
up to 3 extra cycles for data dependencies

trick: use values waiting to get to register file

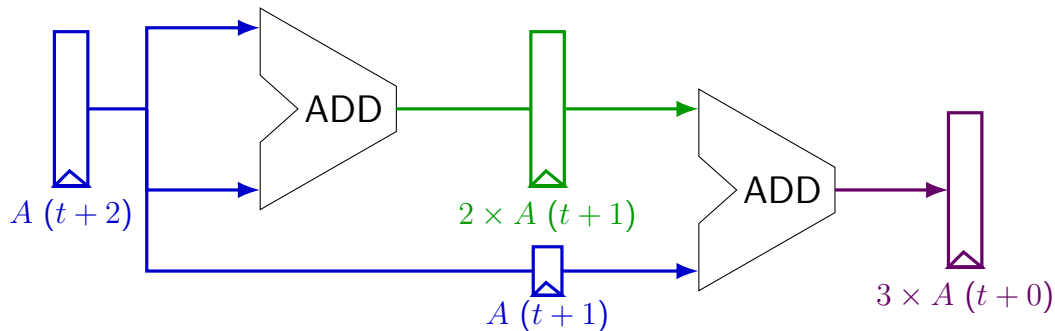
can we do better?

# backup slides

# pipelined times three



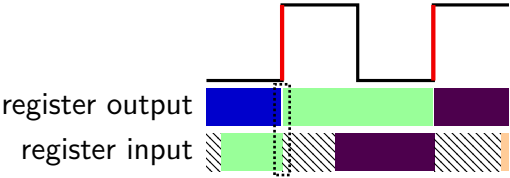
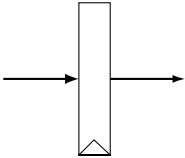
# pipelined times three



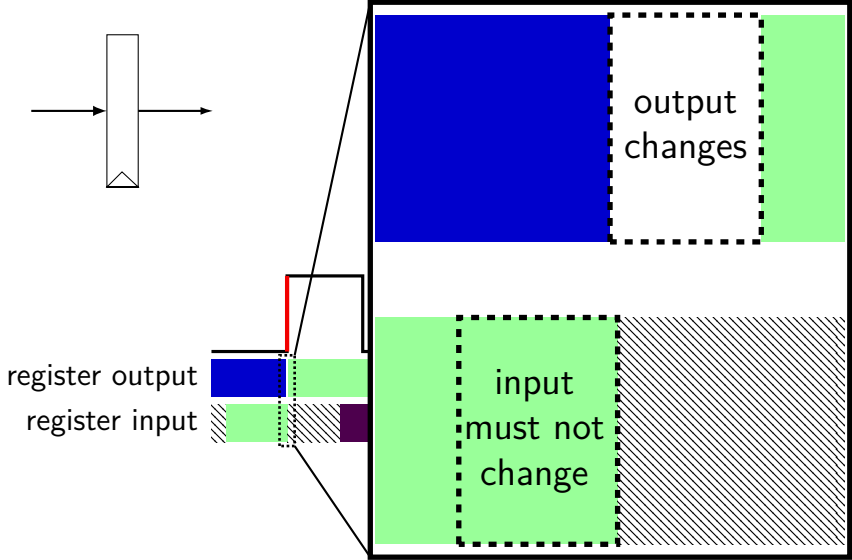
$A(t+2)$	7	17
$A(t+1)$	7	17
$2 \times A(t+1)$	14	34
$3 \times A(t+0)$		21



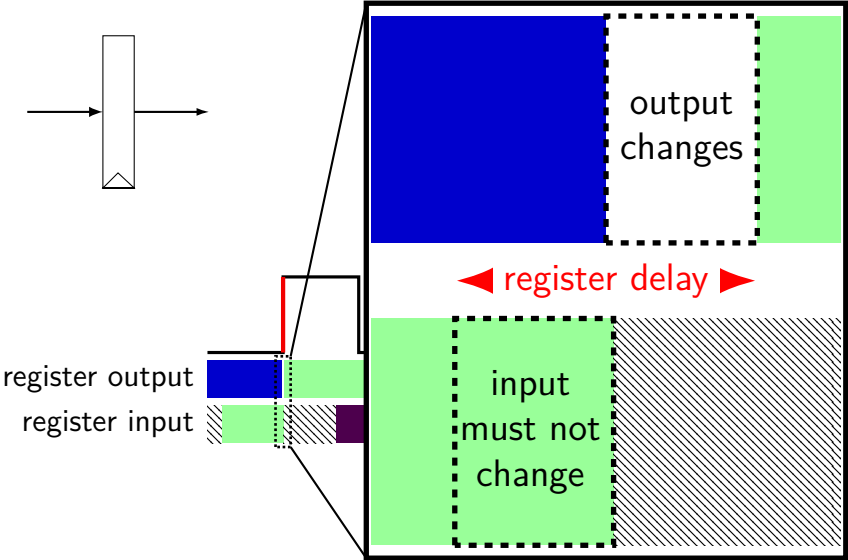
# register tolerances



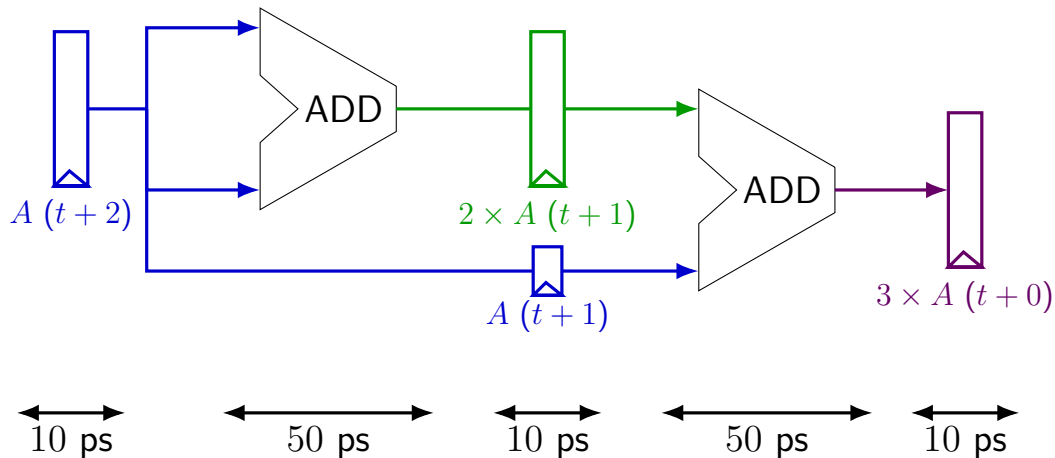
# register tolerances



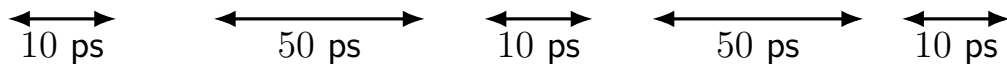
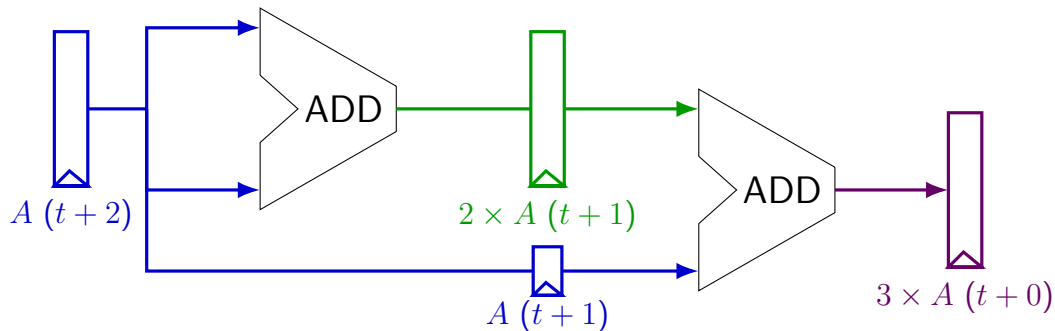
# register tolerances



# times three pipeline timing

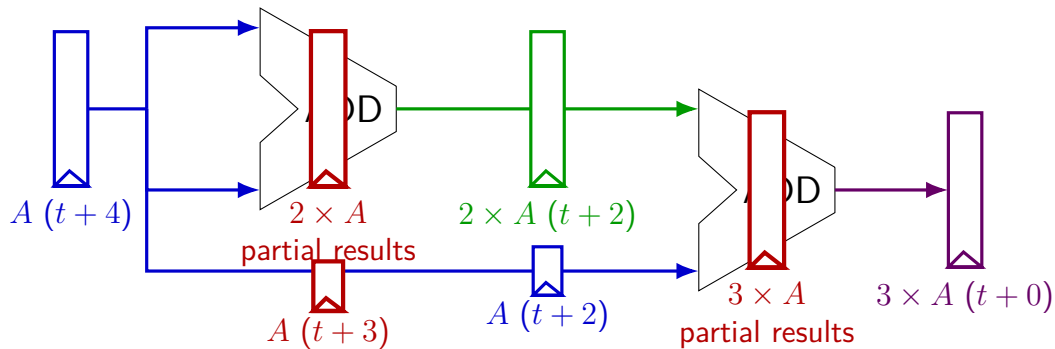


# times three pipeline timing

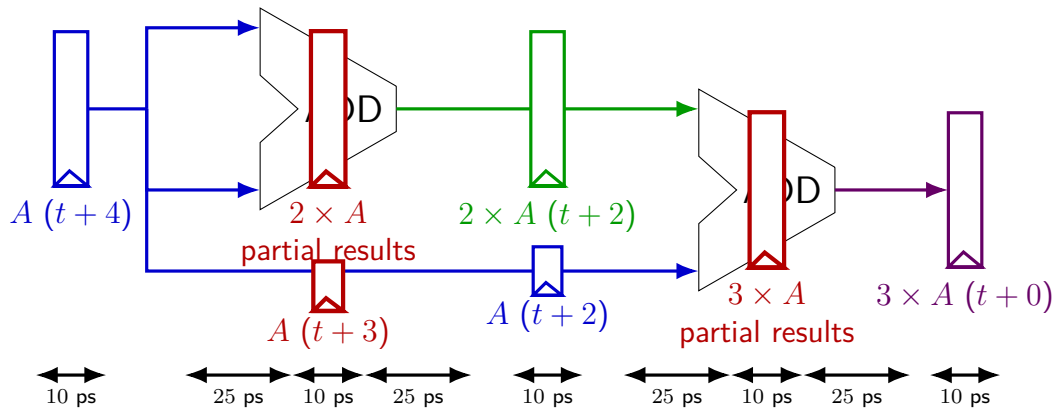


throughput:  $\frac{1}{60 \text{ ps}} \approx 16 \text{ G operations/sec}$

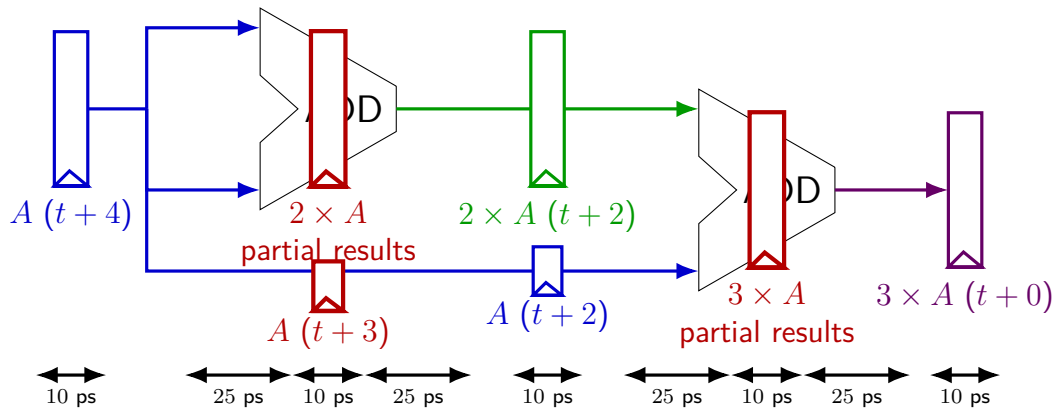
# deeper pipeline



# deeper pipeline



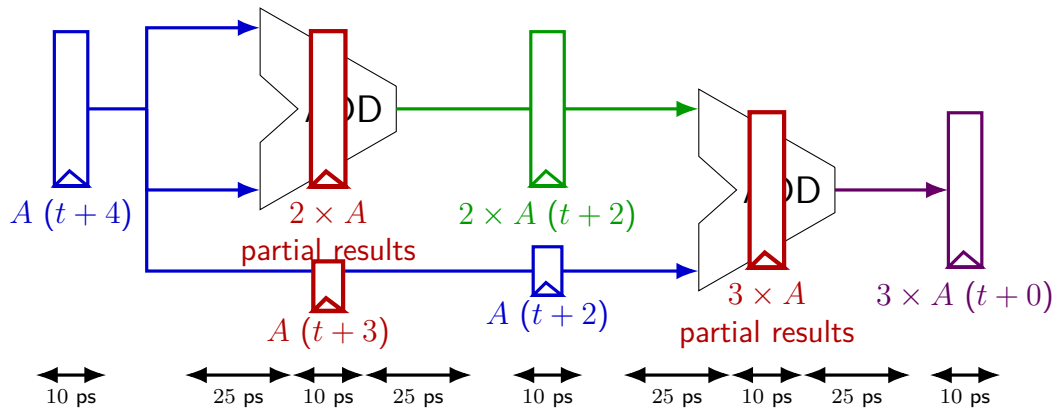
# deeper pipeline



exercise: throughput now?

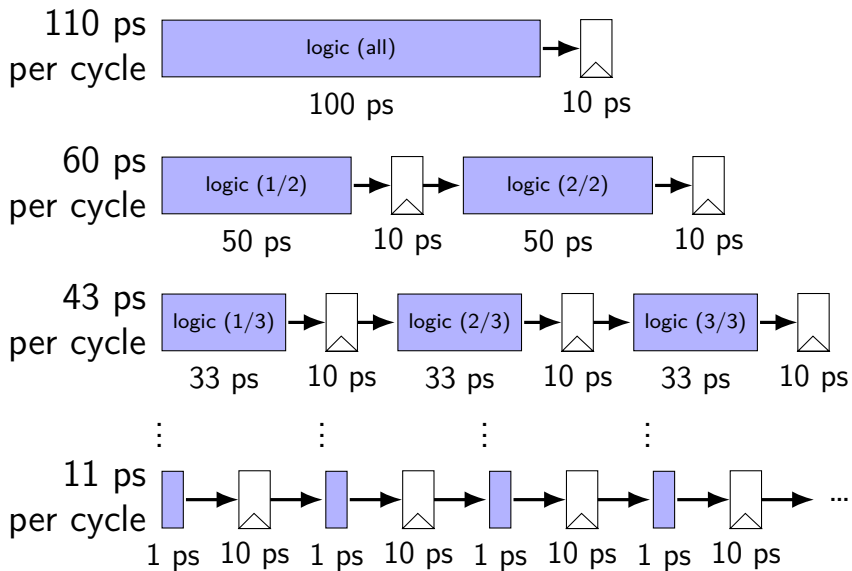


# deeper pipeline

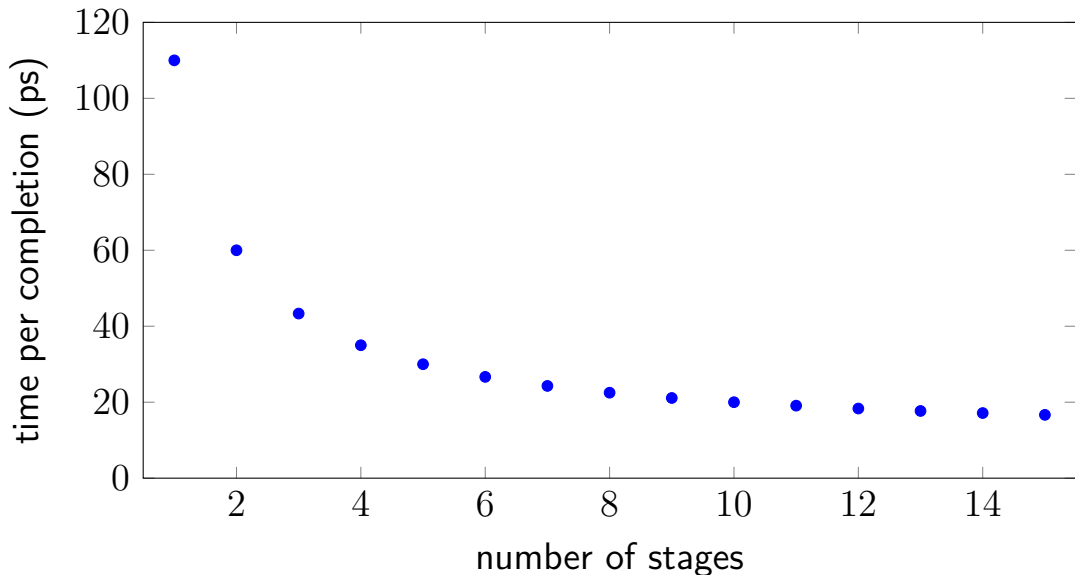


throughput:  $\frac{1}{35 \text{ ps}} \approx 28 \text{ G ops/sec}$

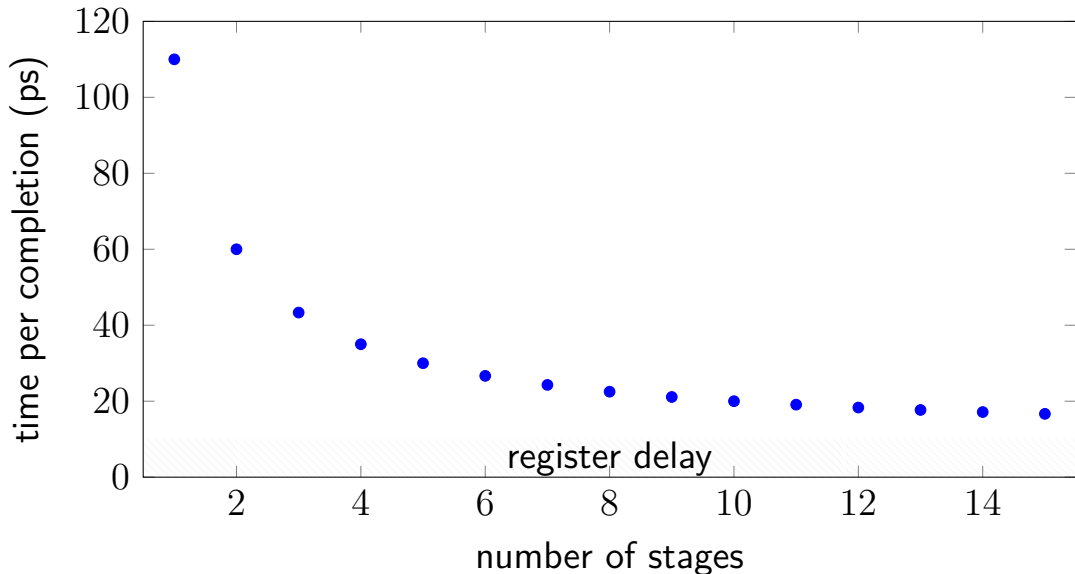
# diminishing returns: register delays



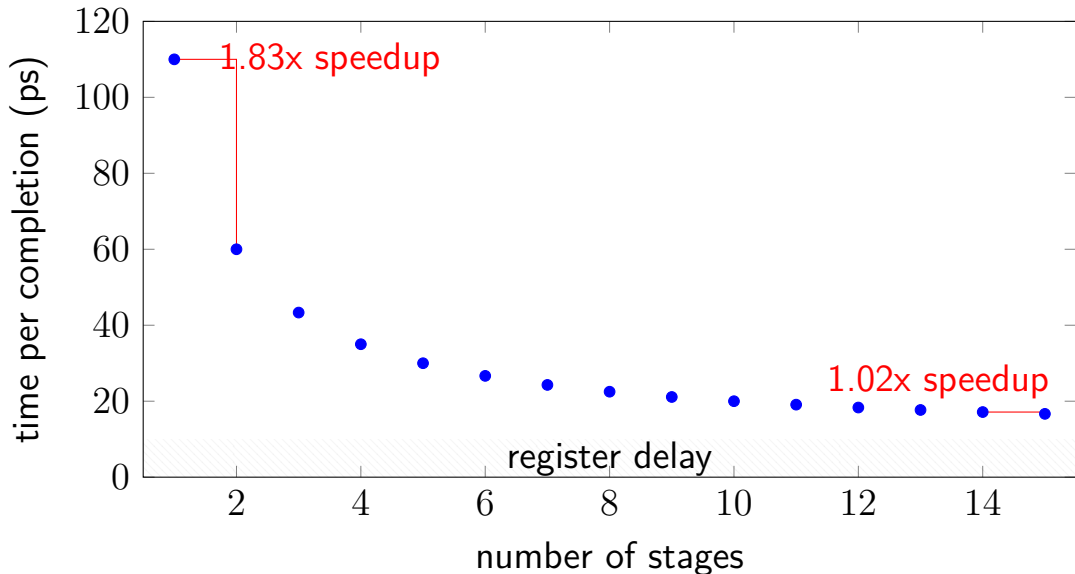
## diminishing returns: register delays



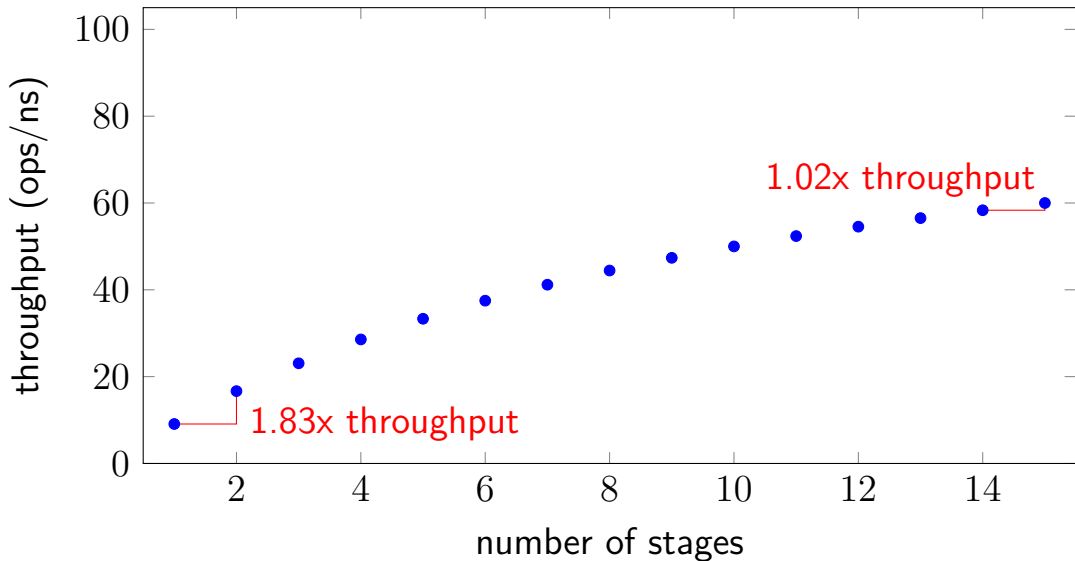
# diminishing returns: register delays



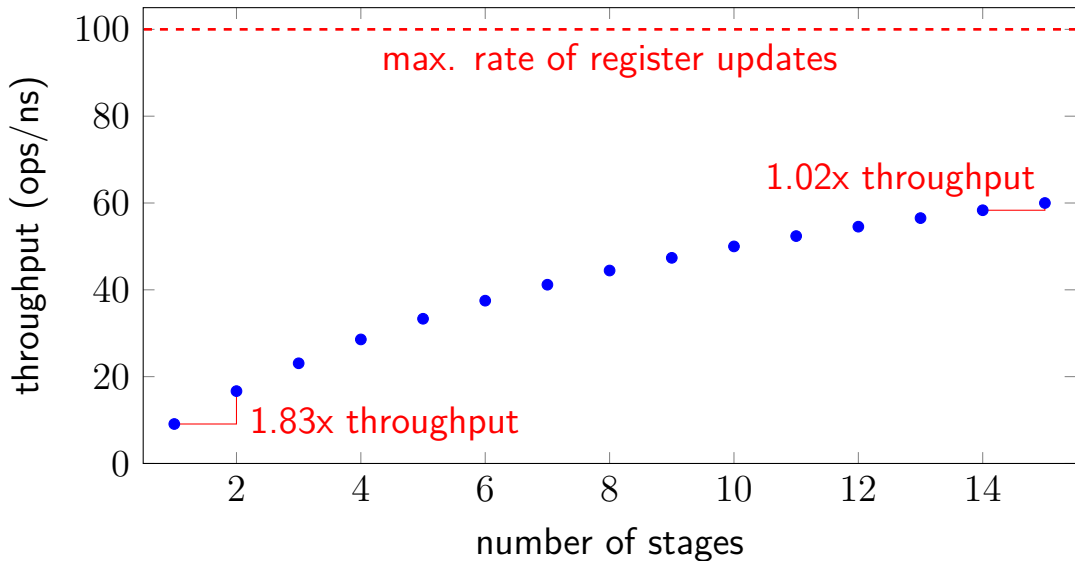
# diminishing returns: register delays



# diminishing returns: register delays

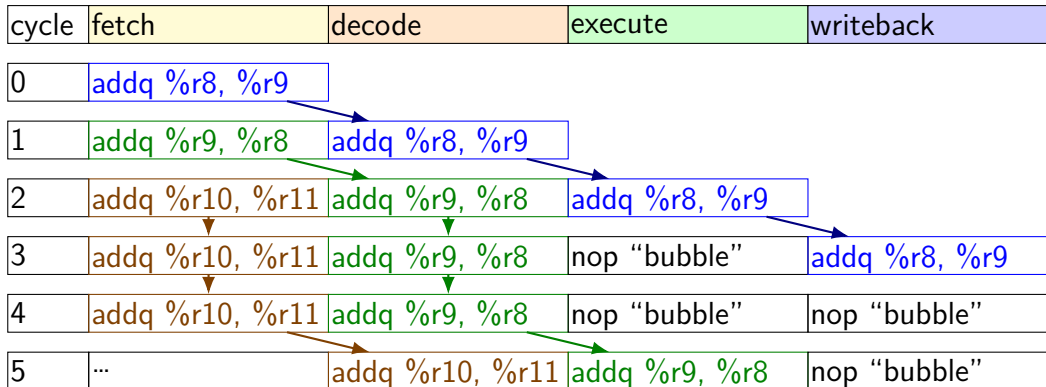


# diminishing returns: register delays



# addq stall (alternative)

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

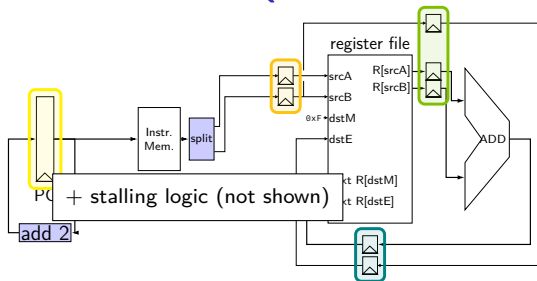






# addq processor: data hazard stall (alternative)

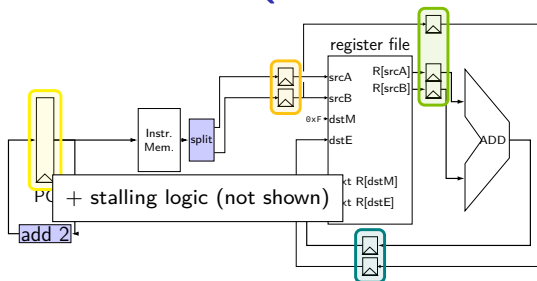
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
addq %r10, %r11
```



	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