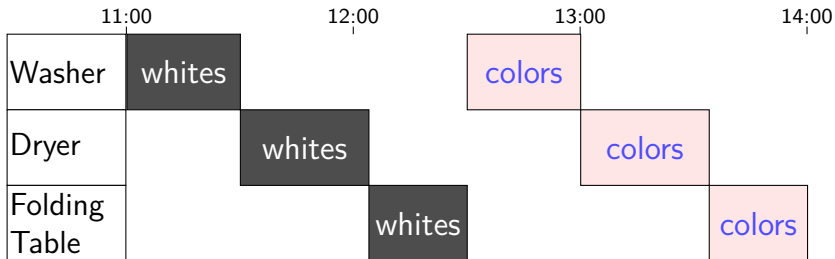
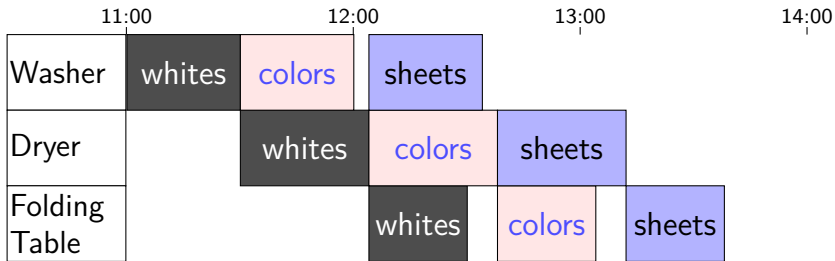
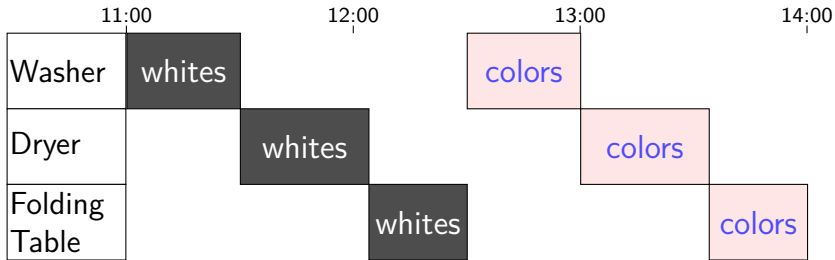


Pipelining (part 1)

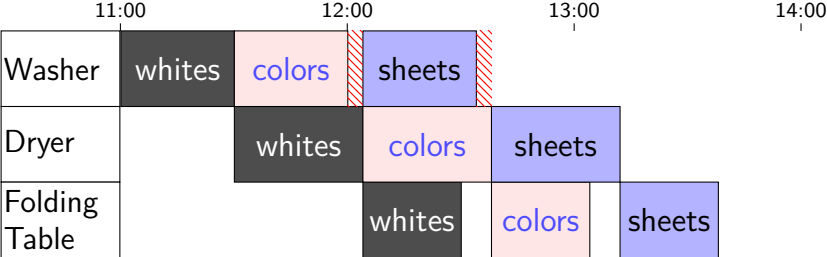
Human pipeline: laundry



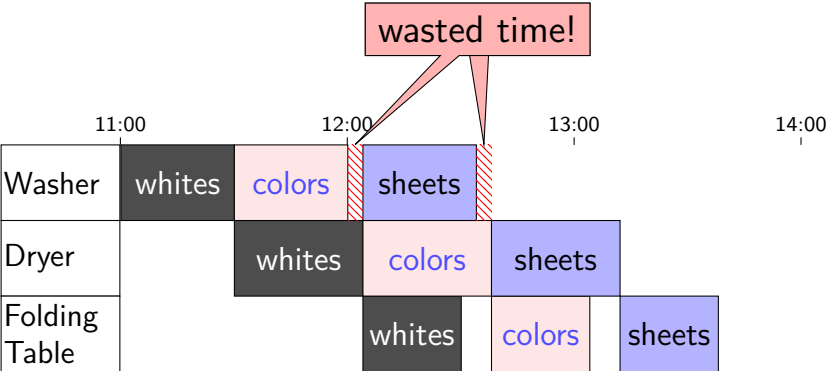
Human pipeline: laundry



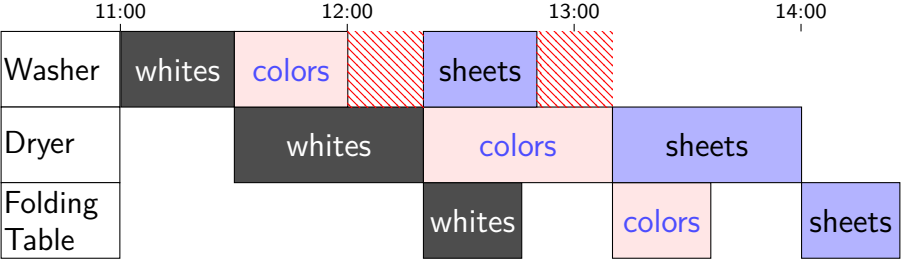
Waste (1)



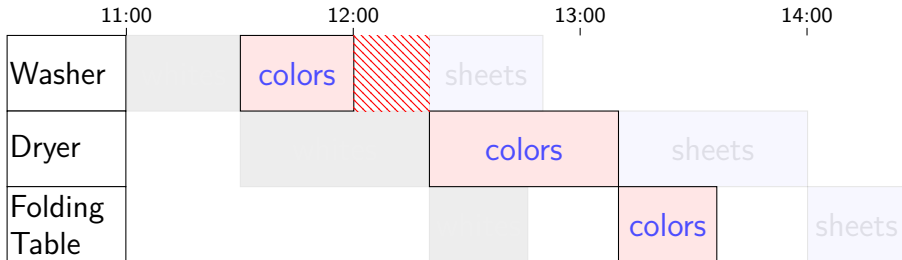
Waste (1)



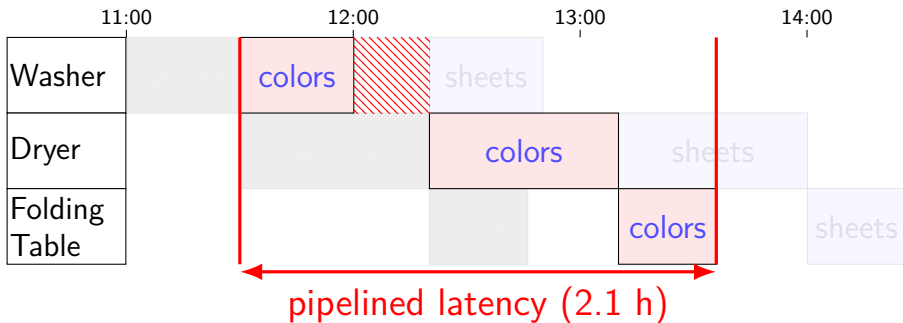
Waste (2)



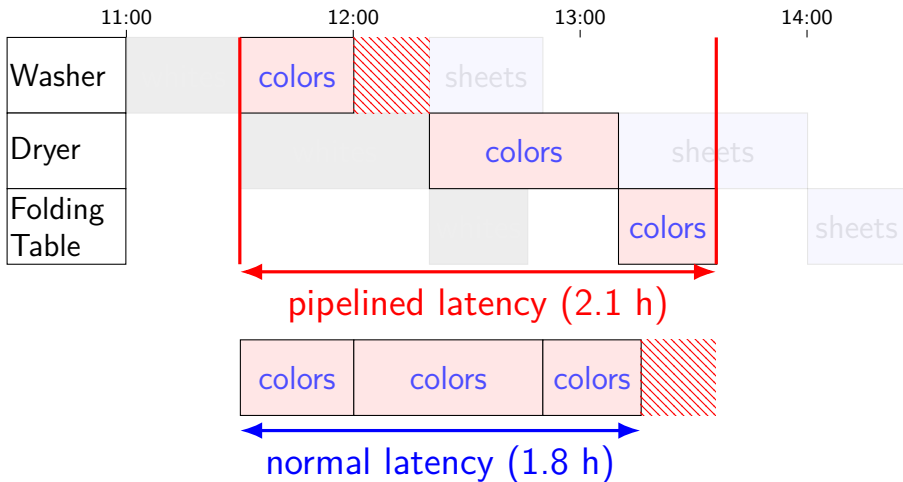
Latency — Time for One



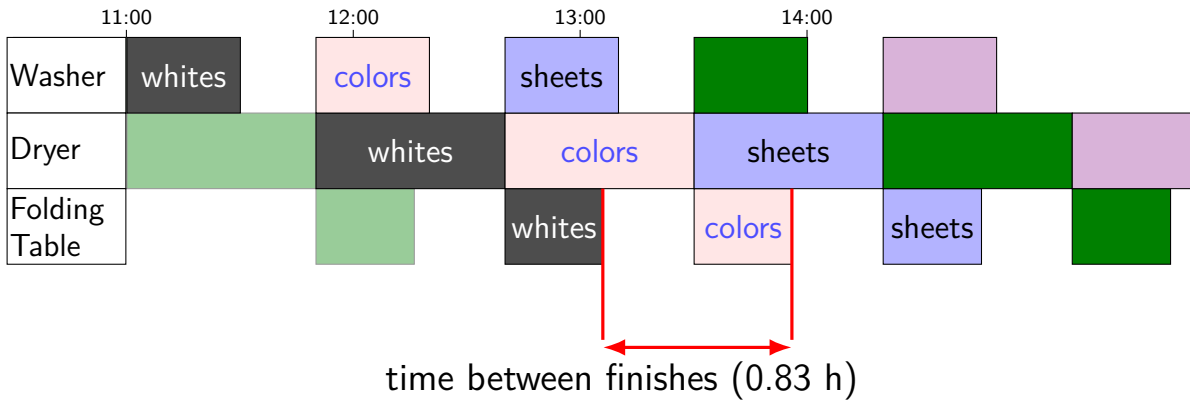
Latency — Time for One



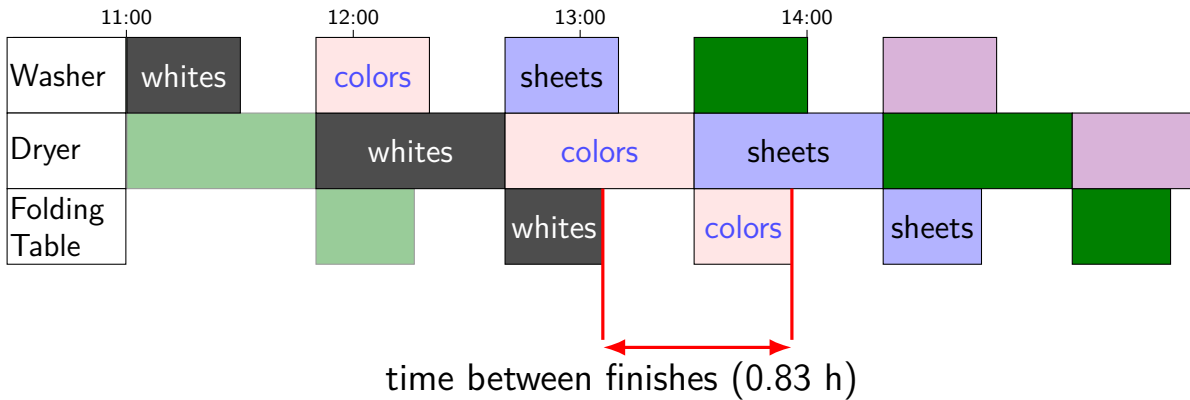
Latency — Time for One



Throughput — Rate of Many

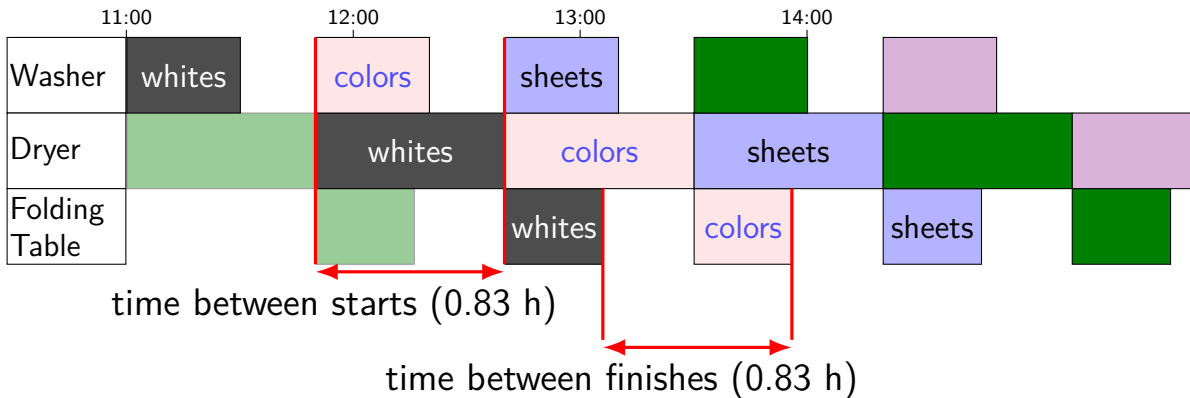


Throughput — Rate of Many



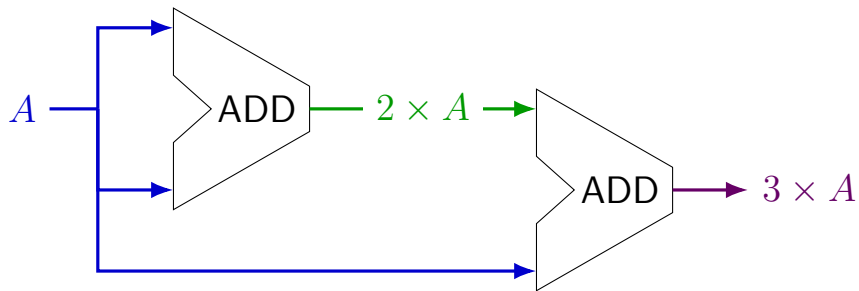
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many

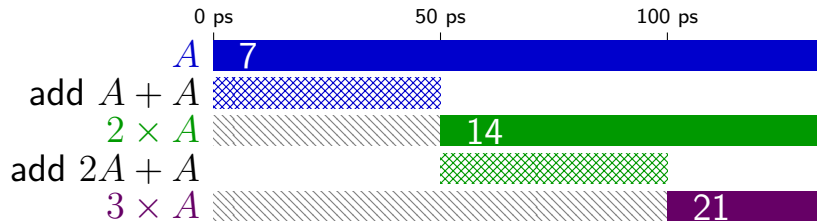
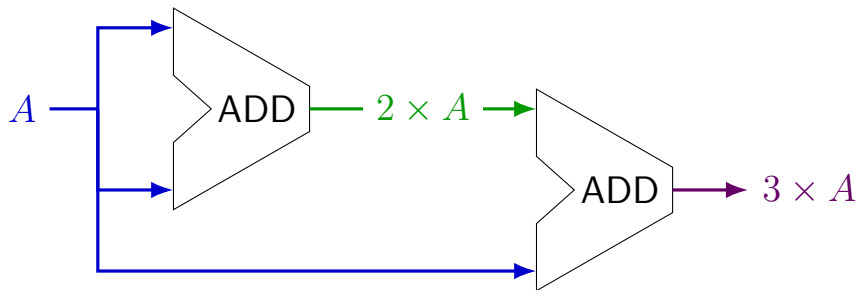


$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

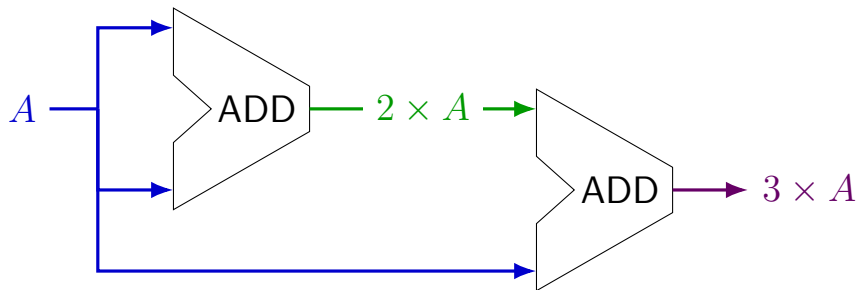
times three circuit



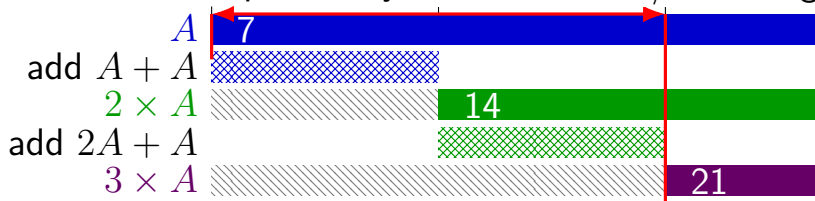
times three circuit



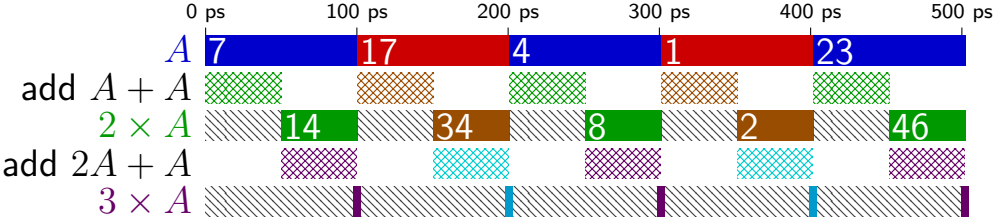
times three circuit



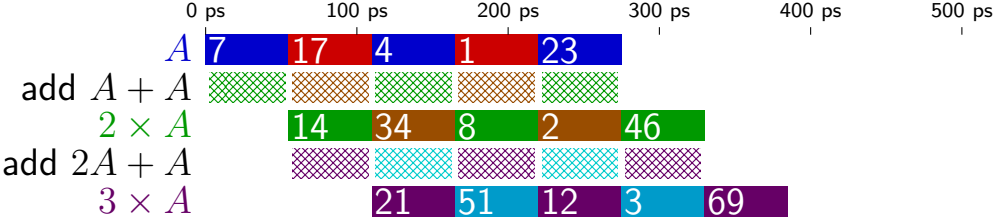
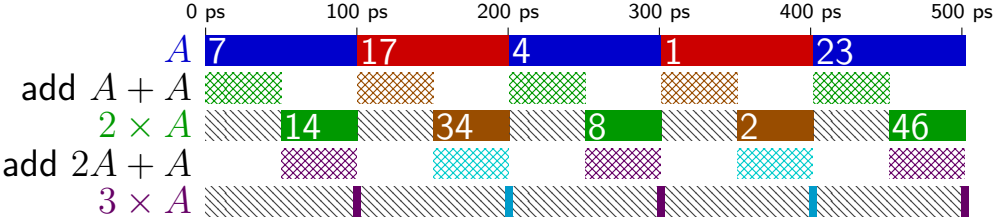
100 ps latency \implies 10 results/ns throughput



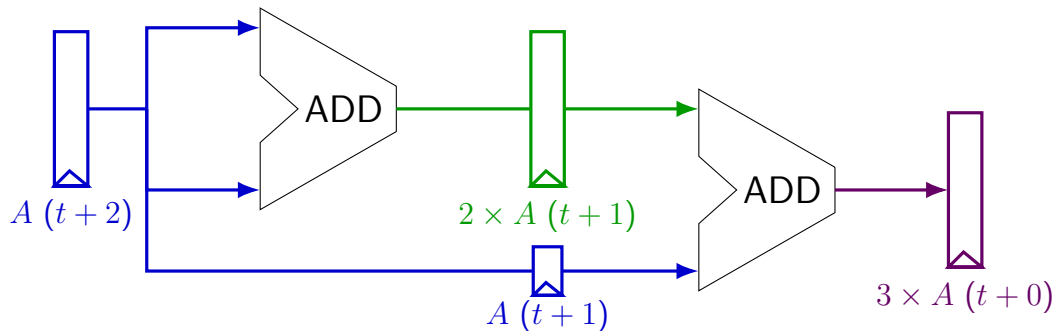
times three and repeat



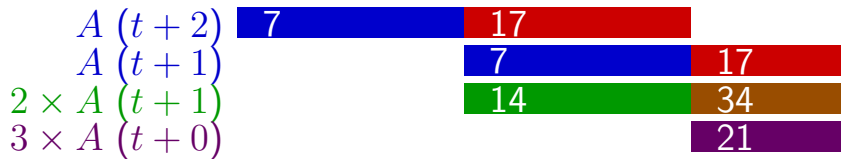
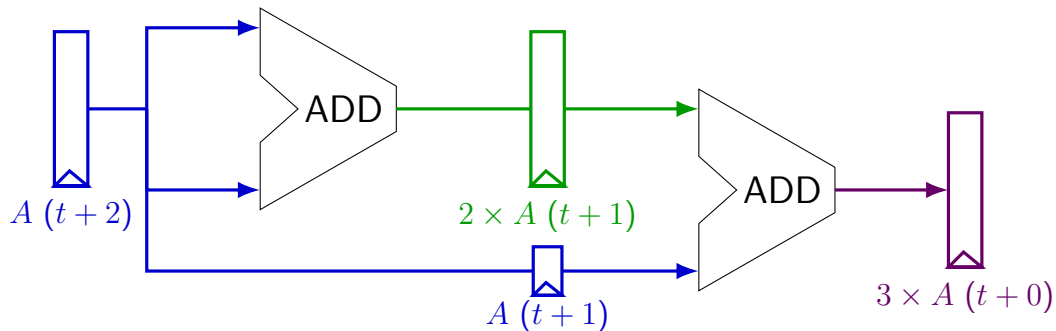
times three and repeat



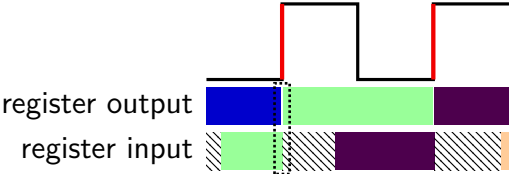
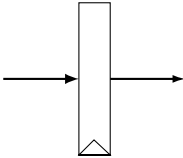
pipelined times three



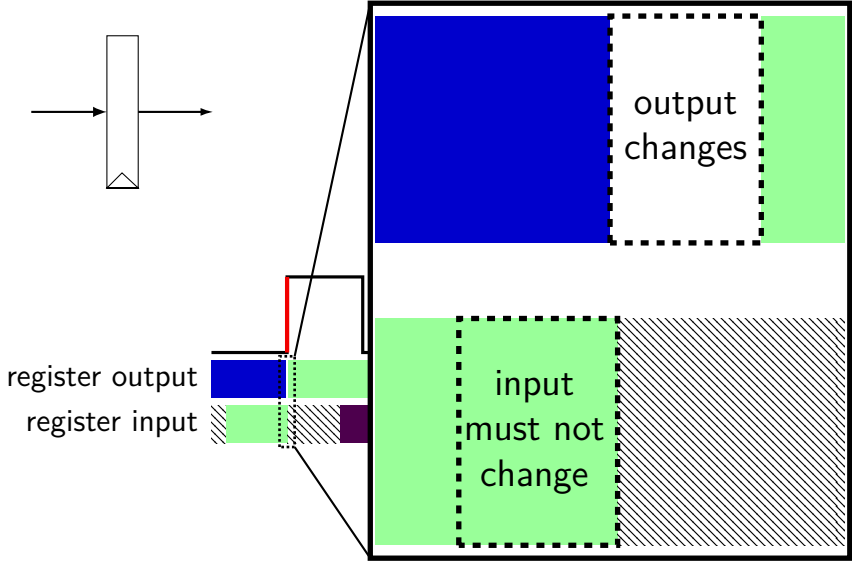
pipelined times three



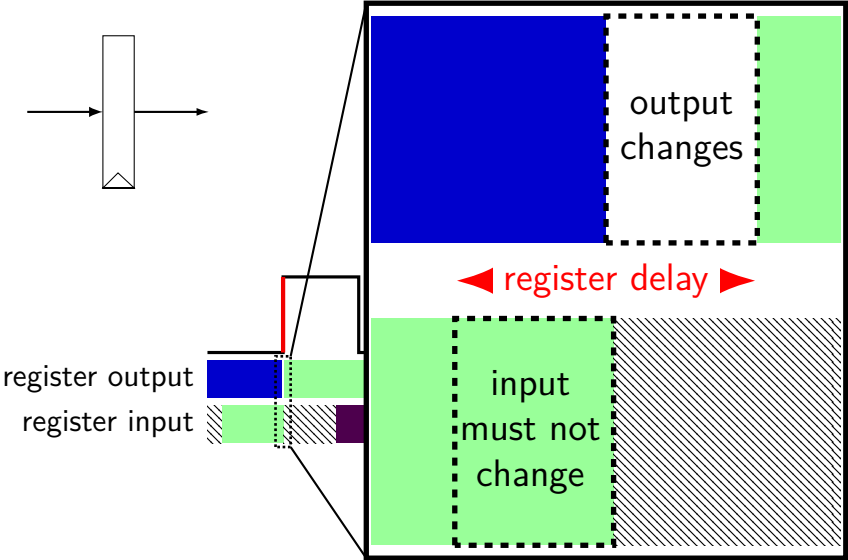
register tolerances



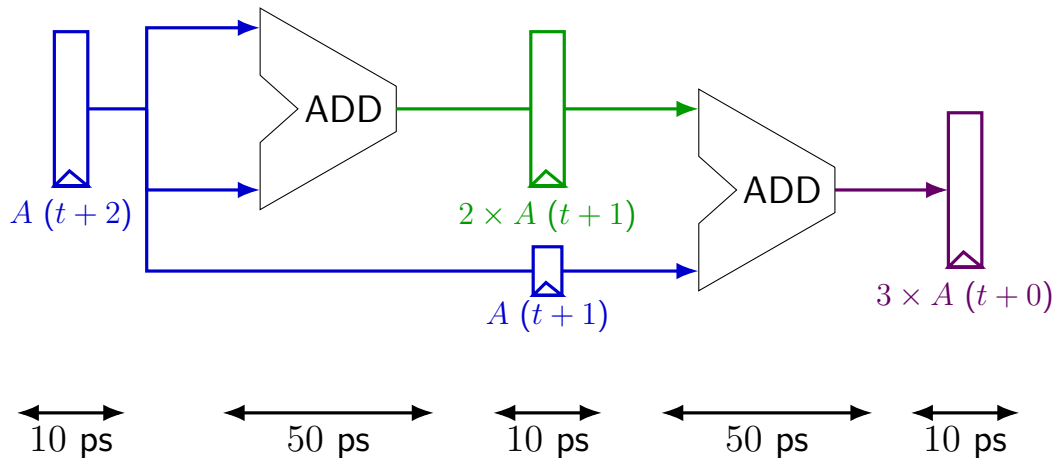
register tolerances



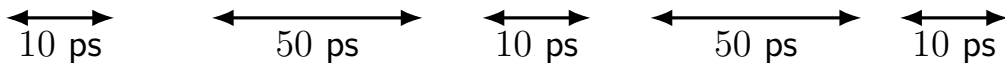
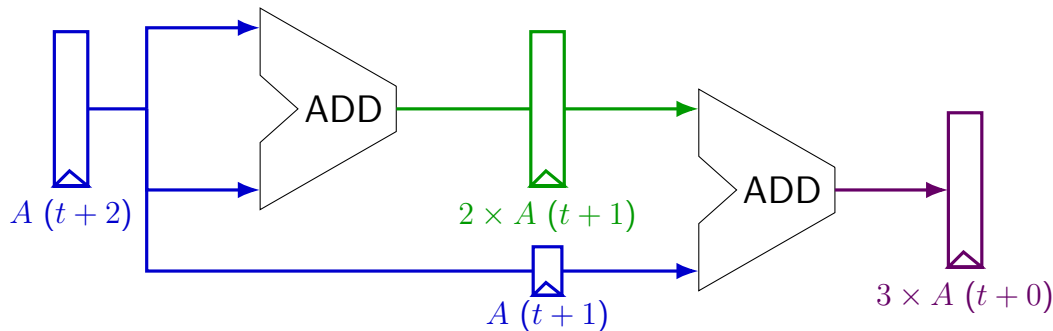
register tolerances



times three pipeline timing

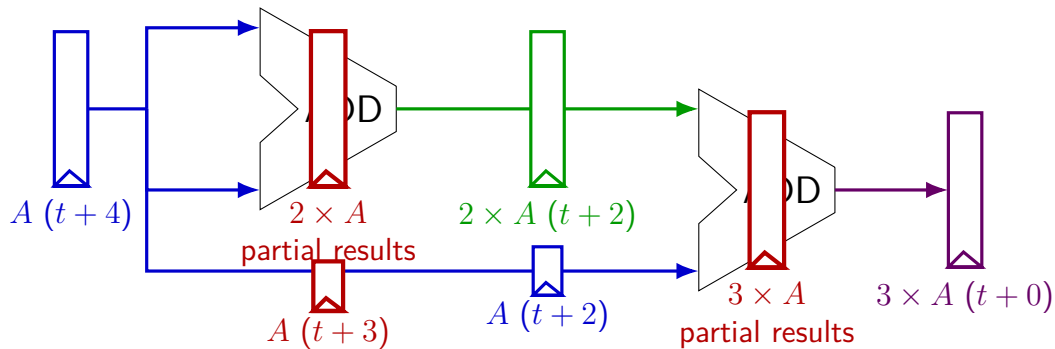


times three pipeline timing

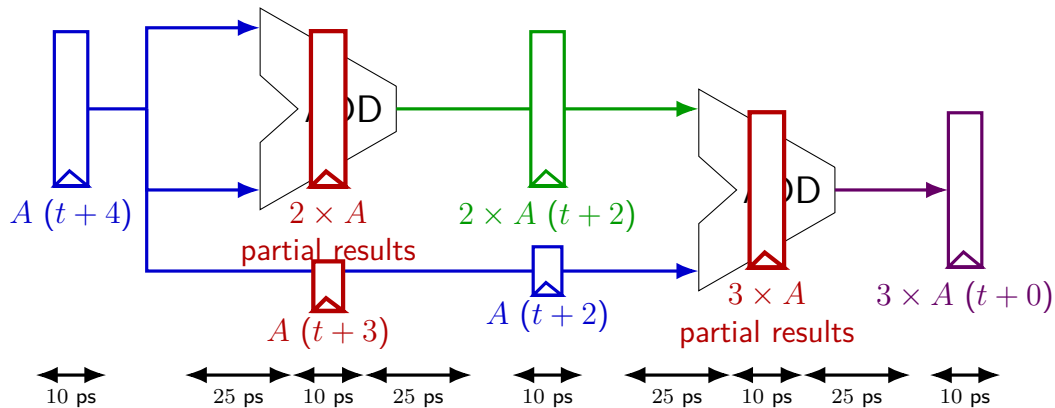


throughput: $\frac{1}{60 \text{ ps}} \approx 16 \text{ G operations/sec}$

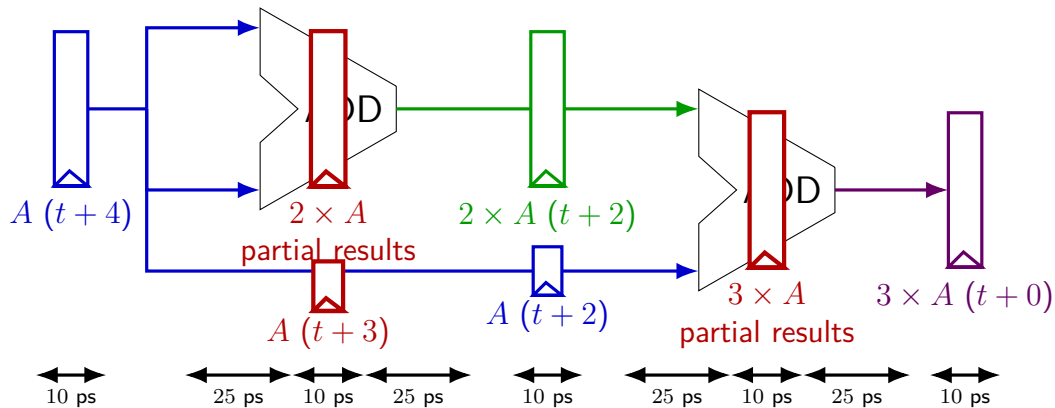
deeper pipeline



deeper pipeline

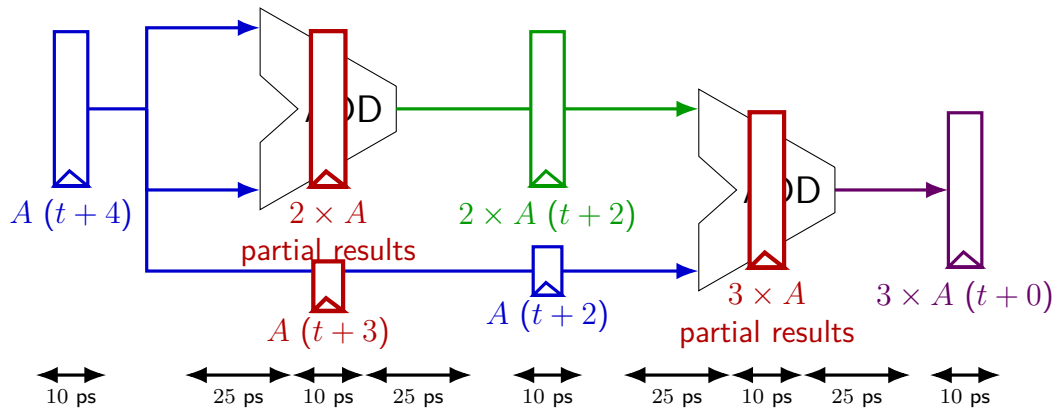


deeper pipeline



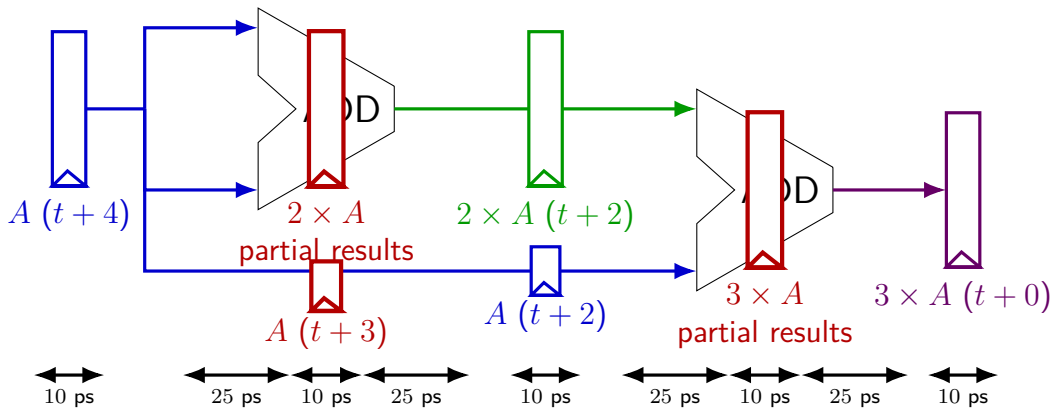
exercise: throughput now?

deeper pipeline



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

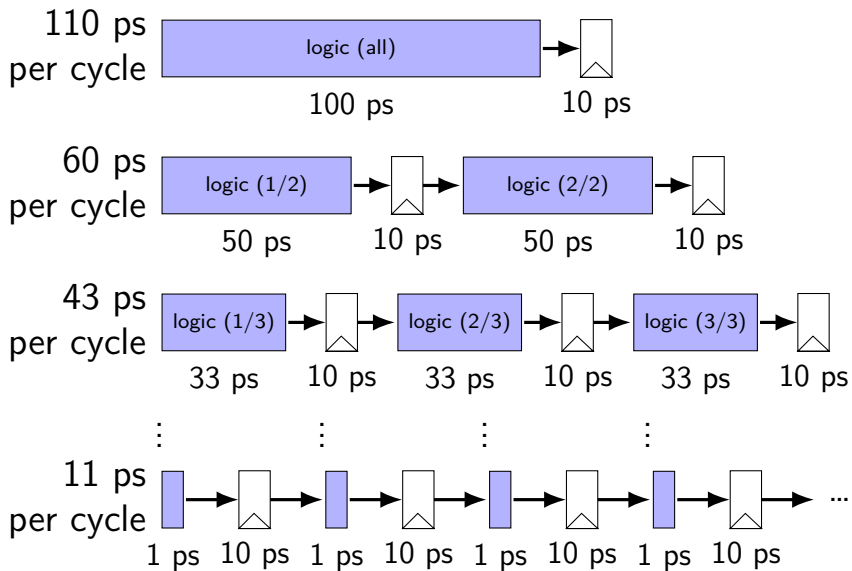
deeper pipeline



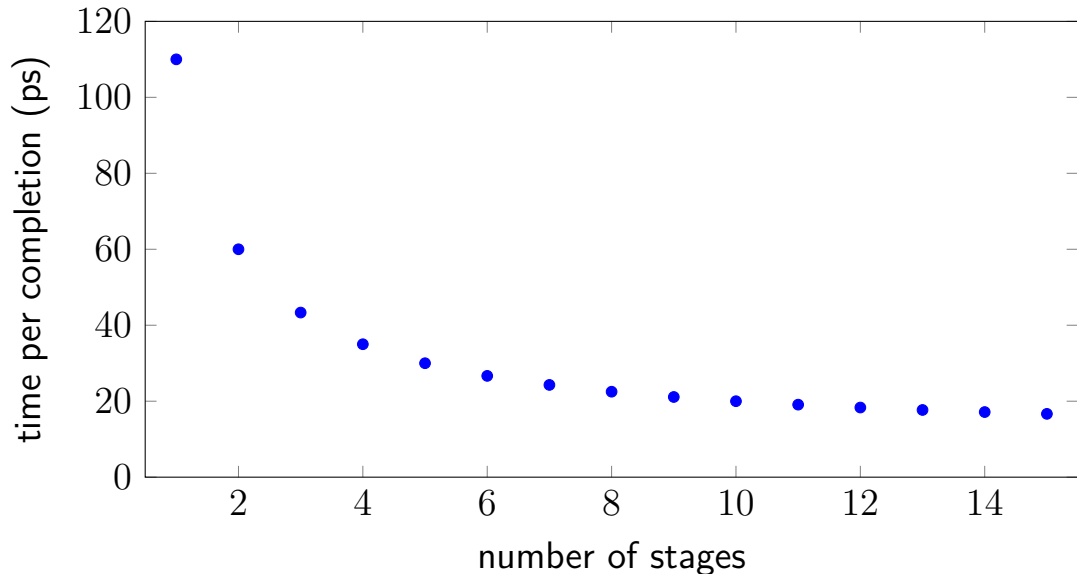
Problem: How much faster can we get?

Problem: Can we even do this?

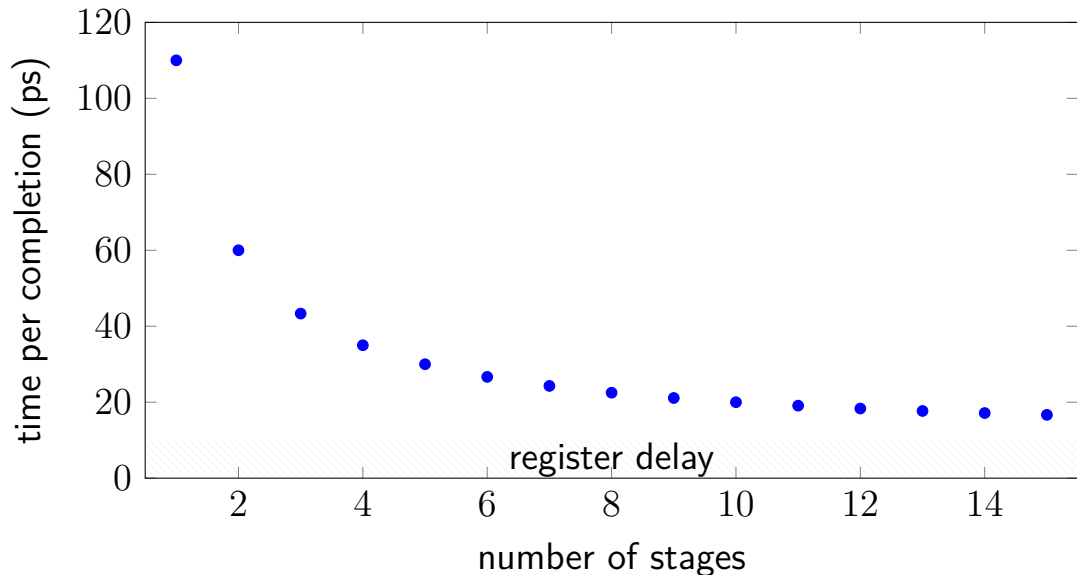
diminishing returns: register delays



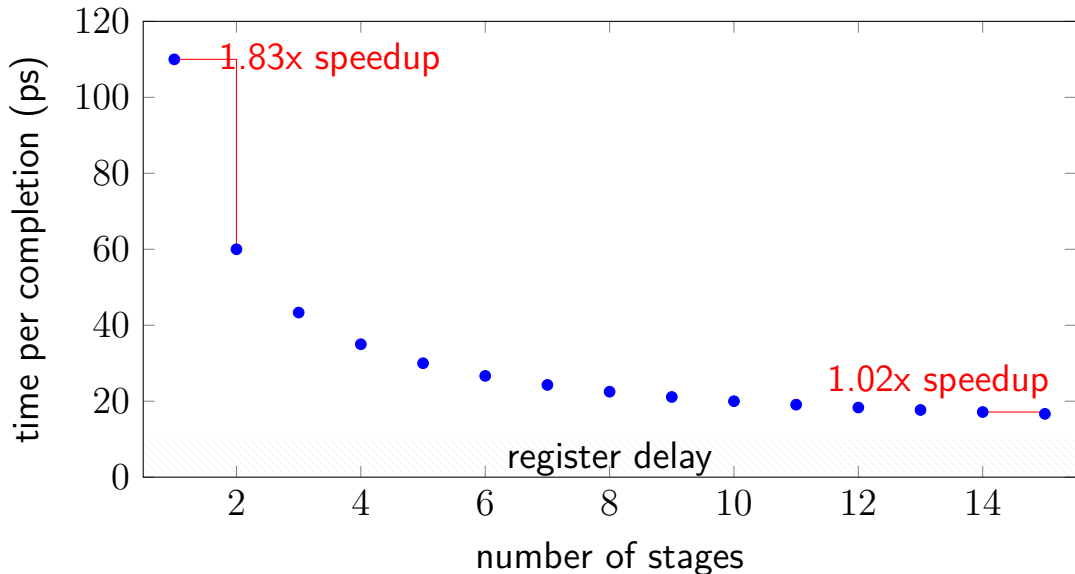
diminishing returns: register delays



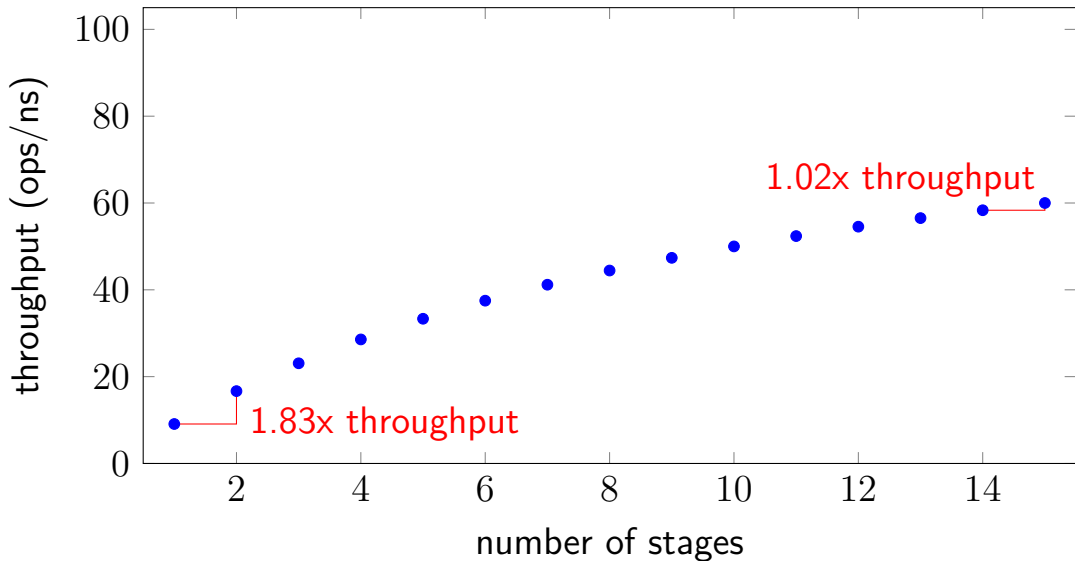
diminishing returns: register delays



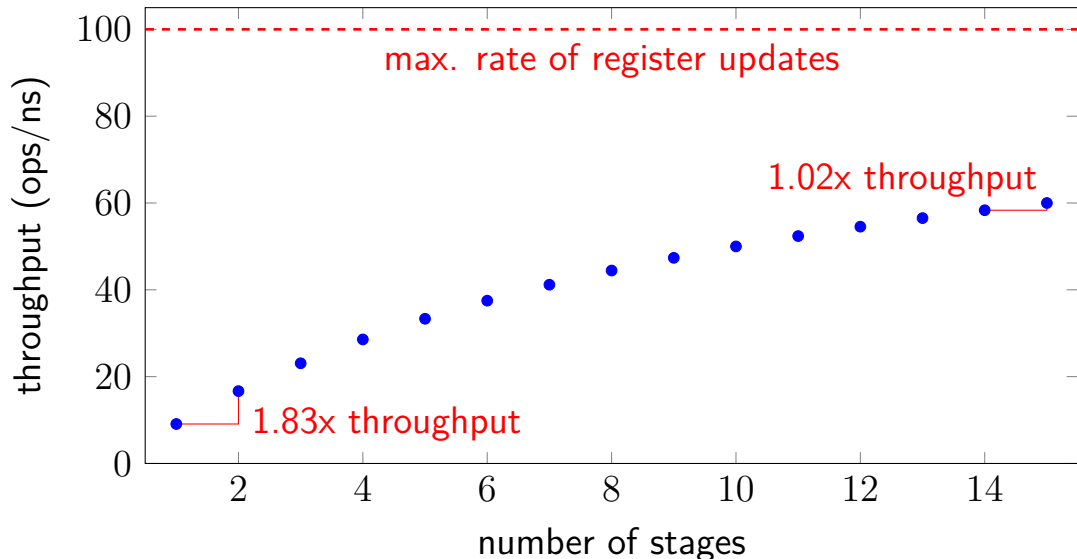
diminishing returns: register delays



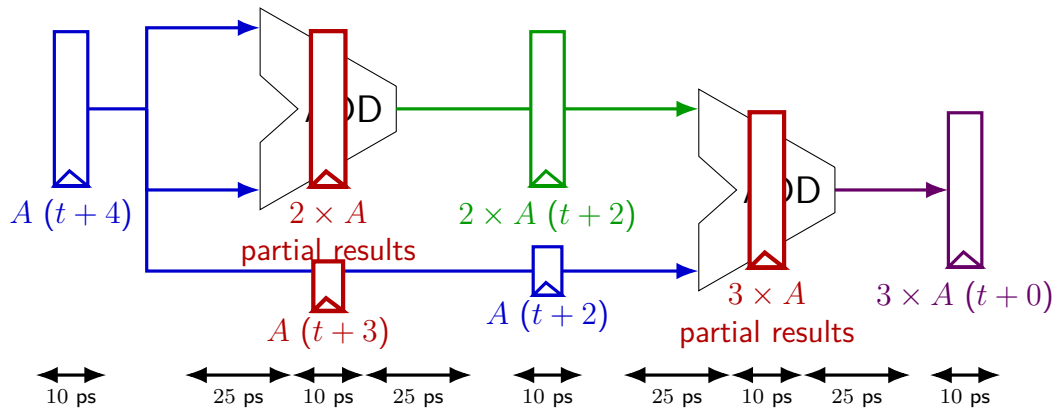
diminishing returns: register delays



diminishing returns: register delays



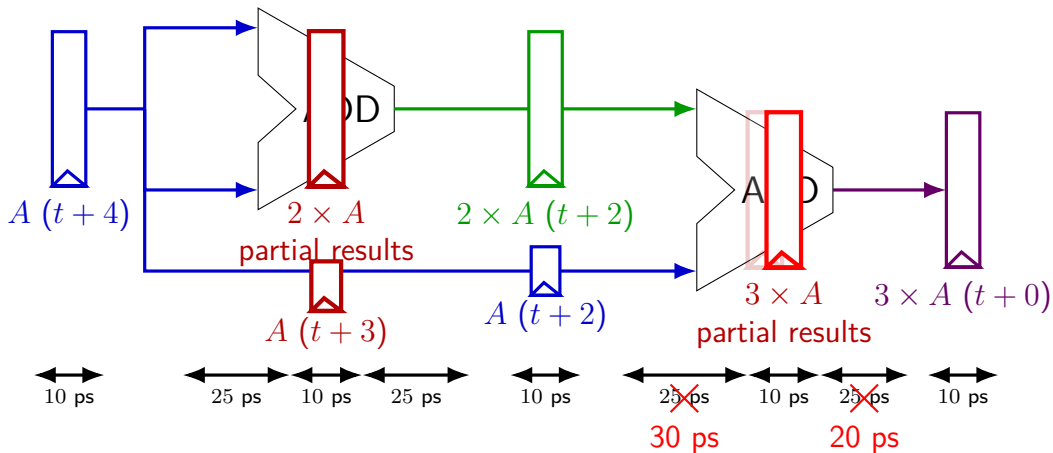
deeper pipeline



Problem: How much faster can we get?

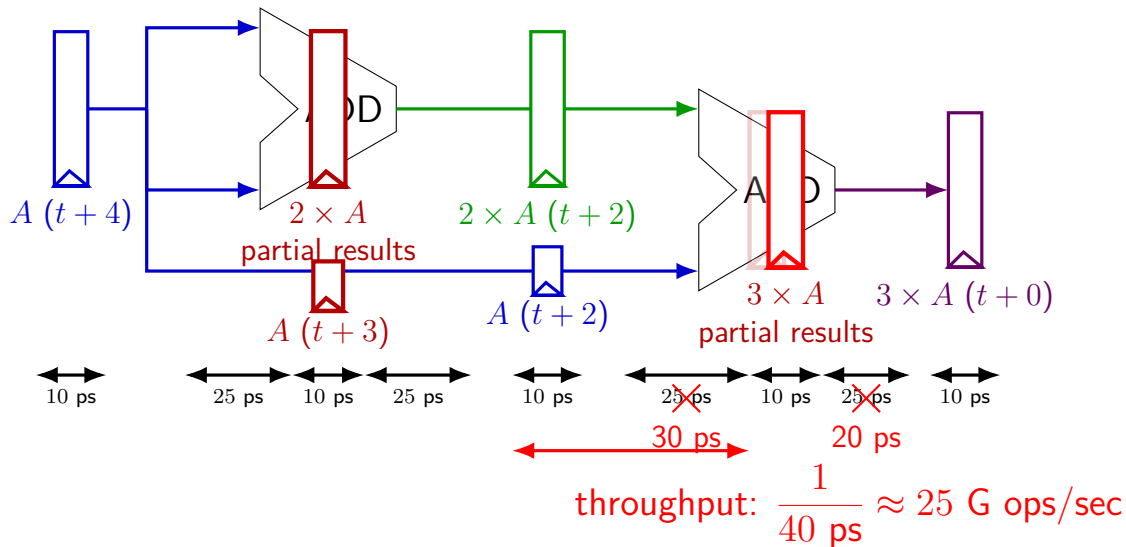
Problem: **Can we even do this?**

deeper pipeline



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

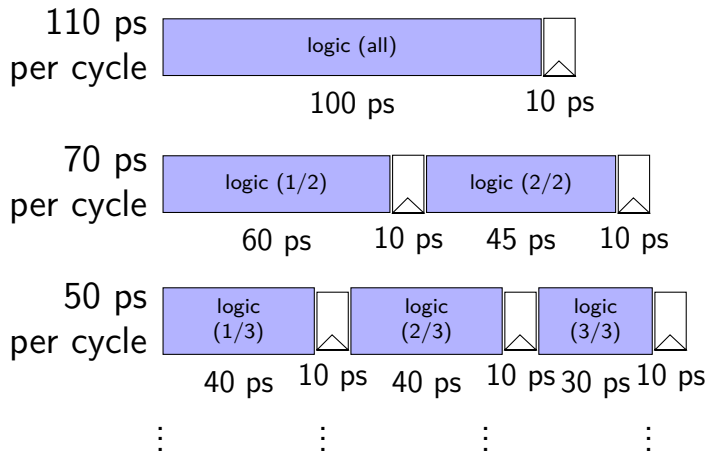
deeper pipeline



diminishing returns: uneven split

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

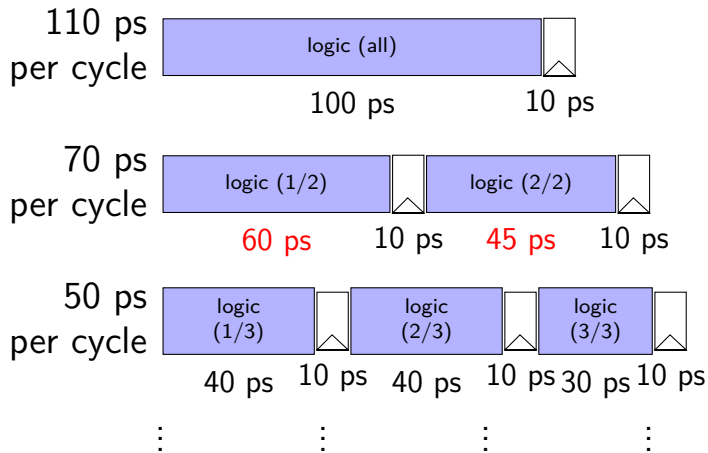
Probably not...



diminishing returns: uneven split

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

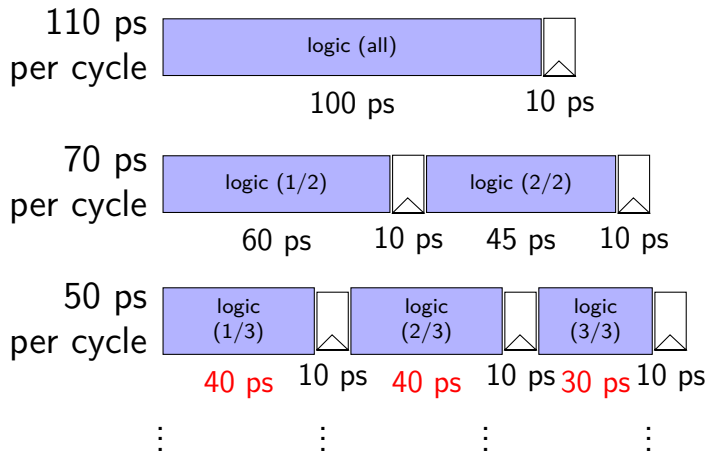
Probably not...



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

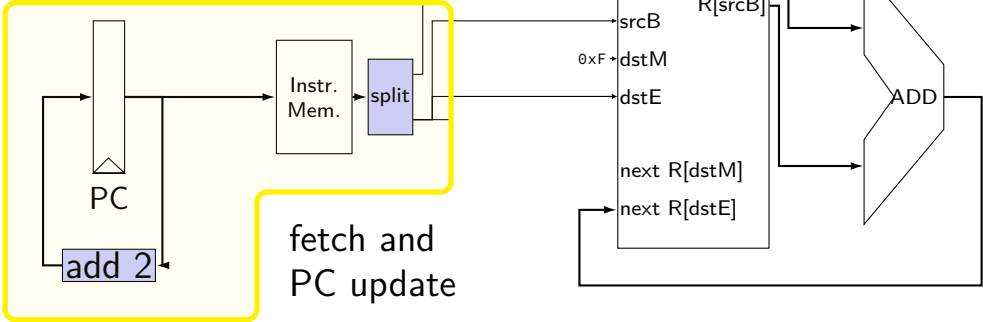
Execute: arithmetic (ALU)

Memory: read/write data memory

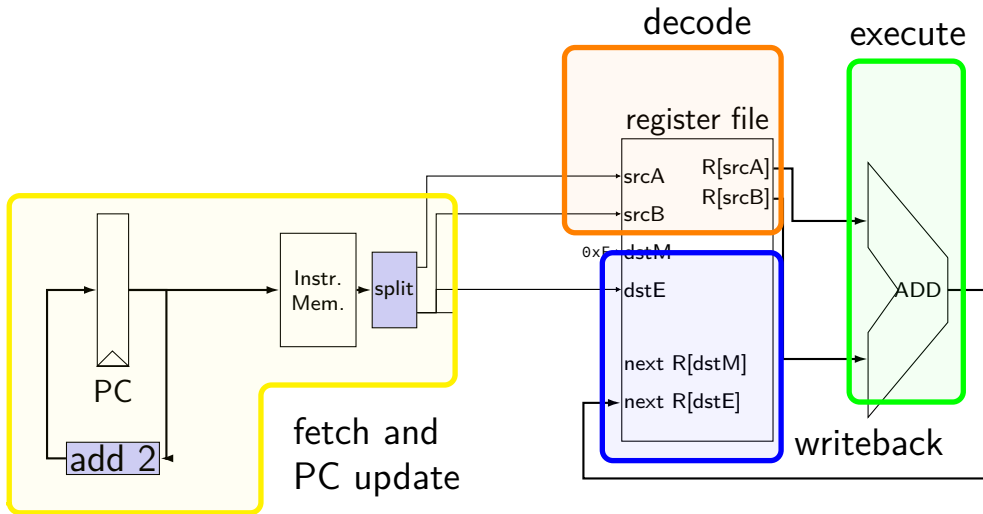
Writeback: write register file

5 stages
one instruction in each
compute next to start immediately

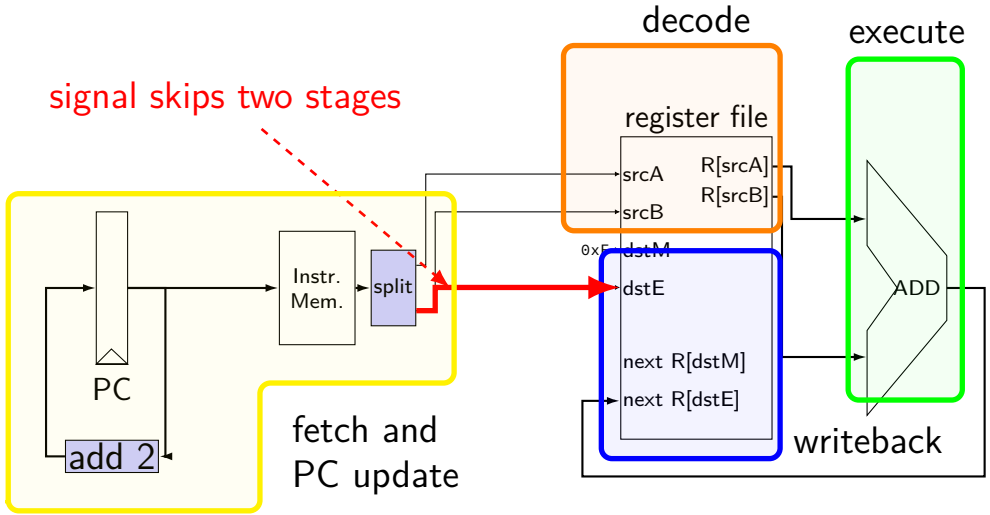
addq CPU



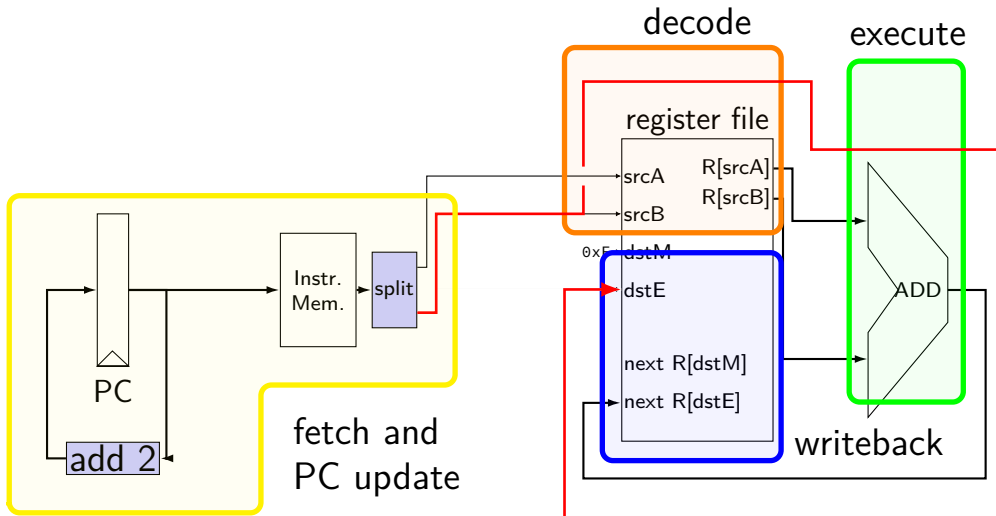
addq CPU



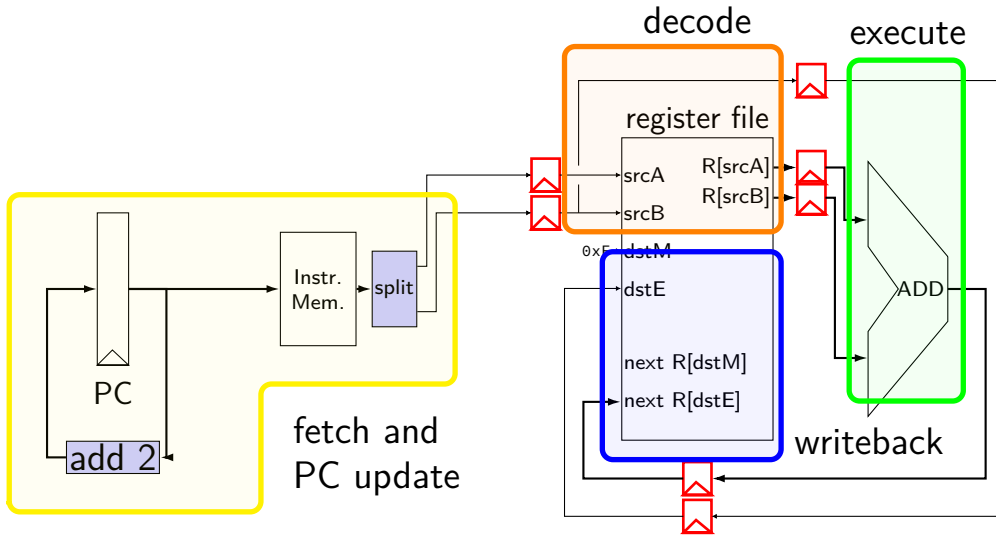
addq CPU



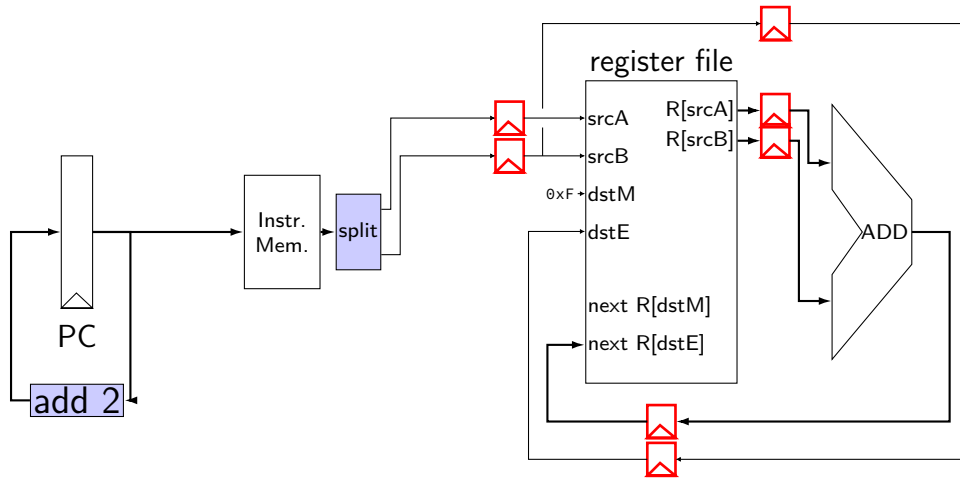
addq CPU



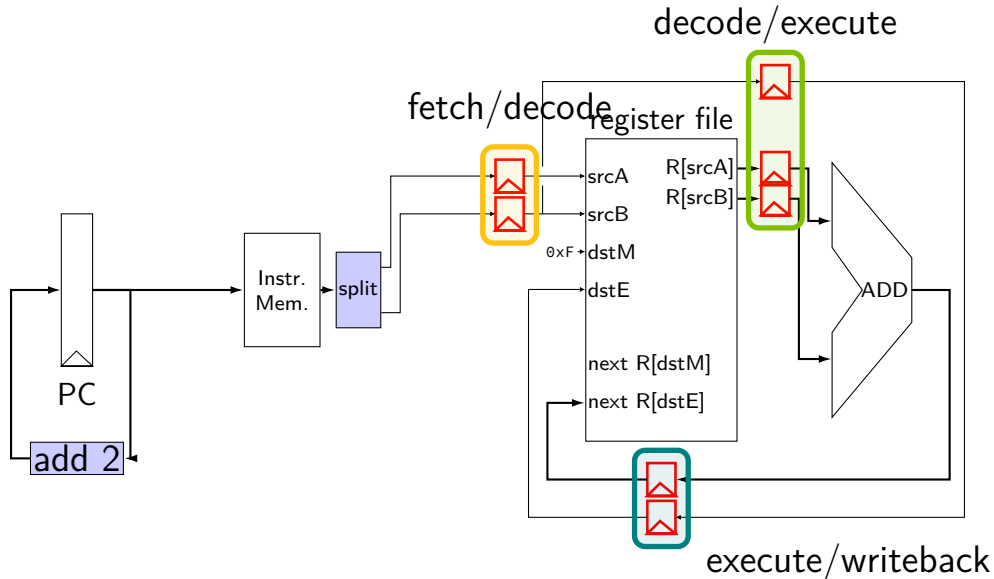
pipelined addq processor



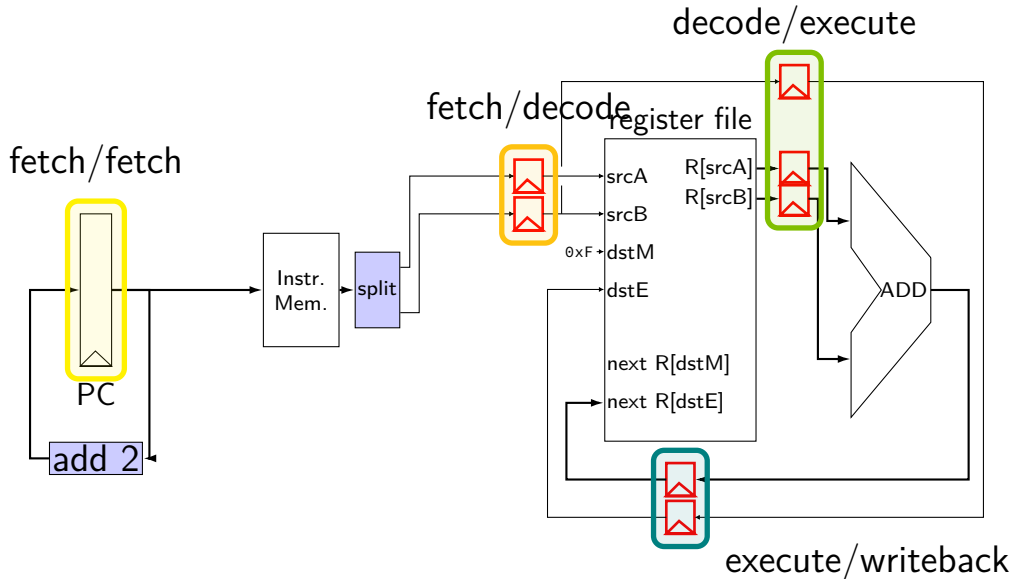
pipelined addq processor



pipelined addq processor



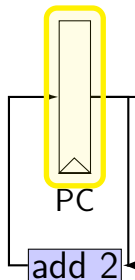
pipelined addq processor



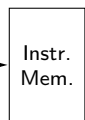
addq execution

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

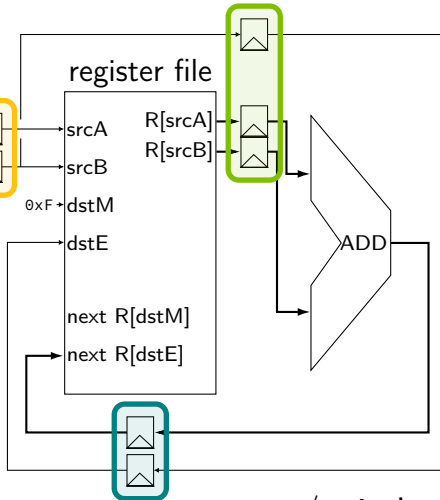
fetch/fetch



fetch/decode



decode/execute

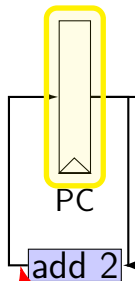


execute/writeback

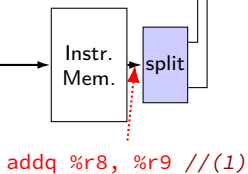
addq execution

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

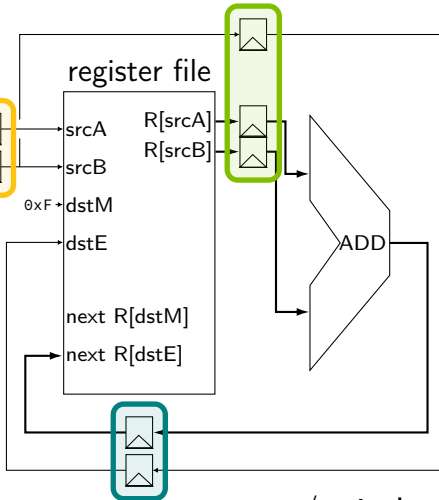
fetch/fetch



fetch/decode



decode/execute

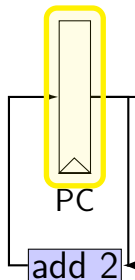


execute/writeback

addq execution

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

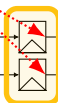
fetch/fetch



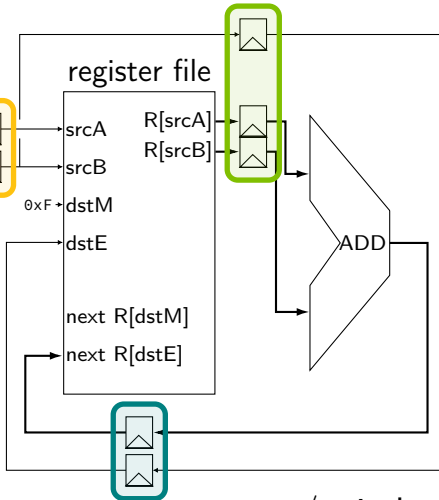
reg #s 8, 9 from (1)

fetch/decode

addq %r10, %r11 // (2)



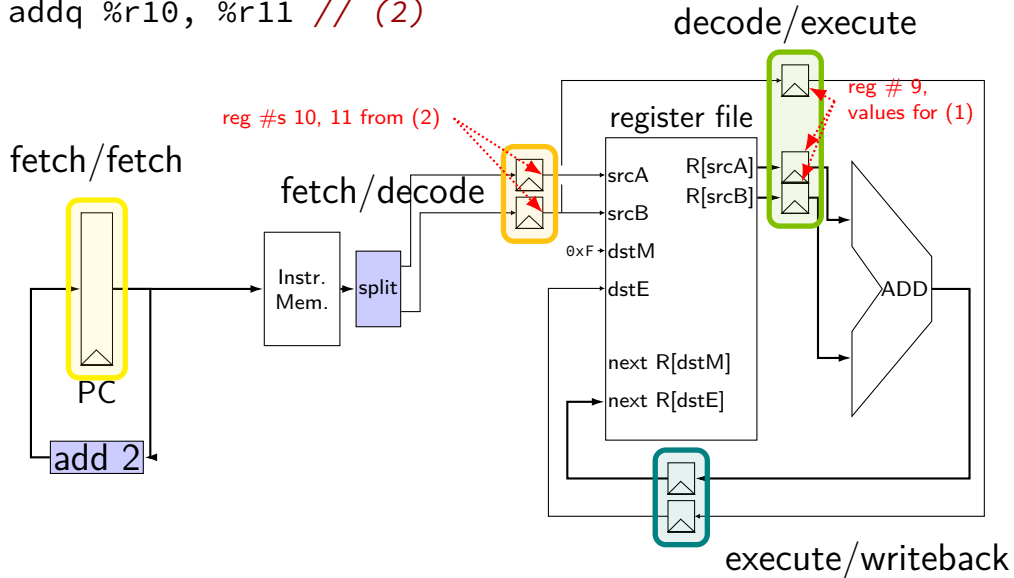
decode/execute



execute/writeback

addq execution

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

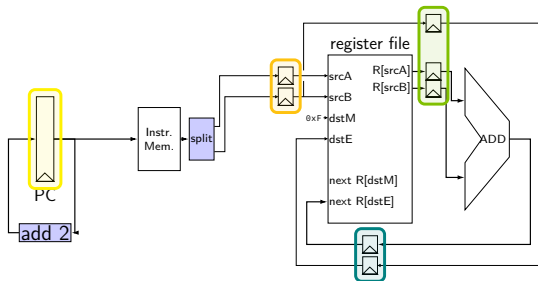


addq processor timing

```
// 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

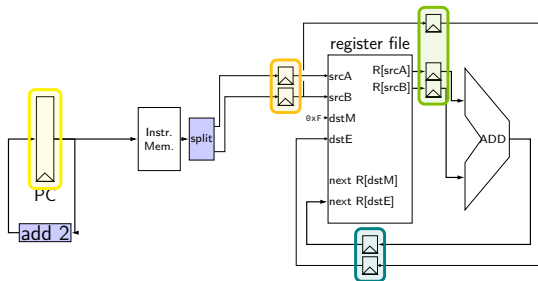


addq processor timing

```
// 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

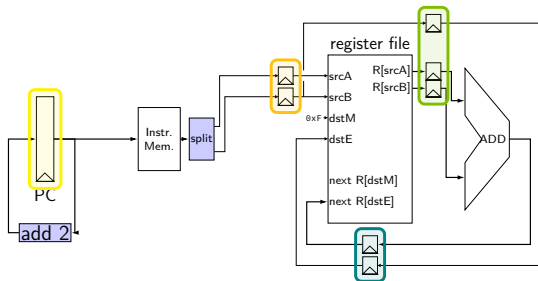


addq processor timing

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

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

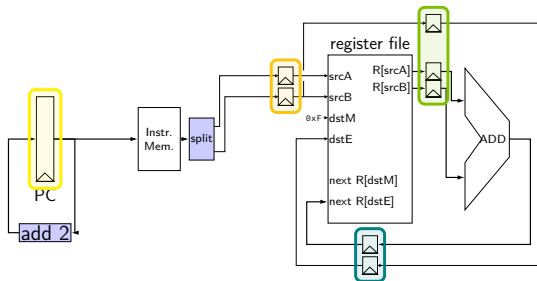
	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8



addq processor timing

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

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

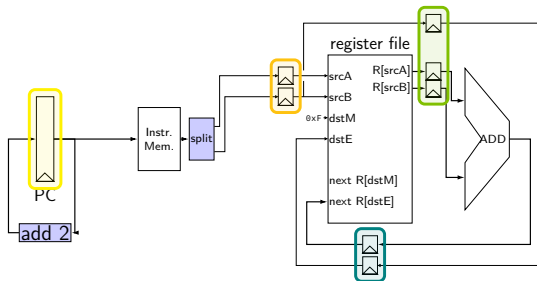


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor timing

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

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

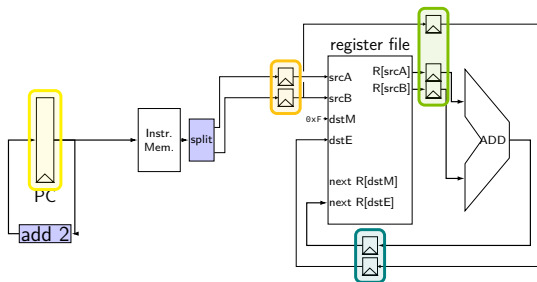


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

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)