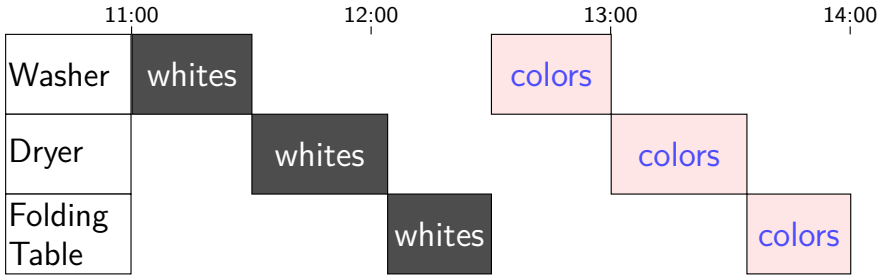
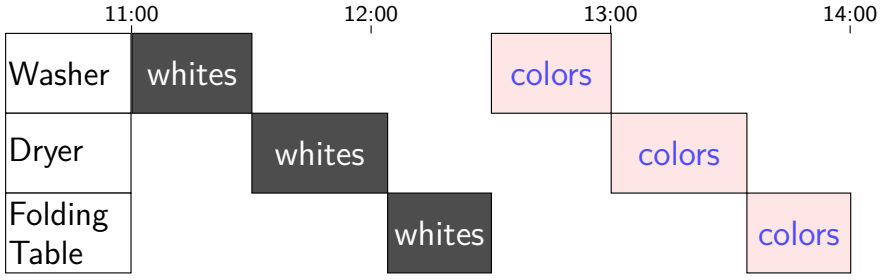
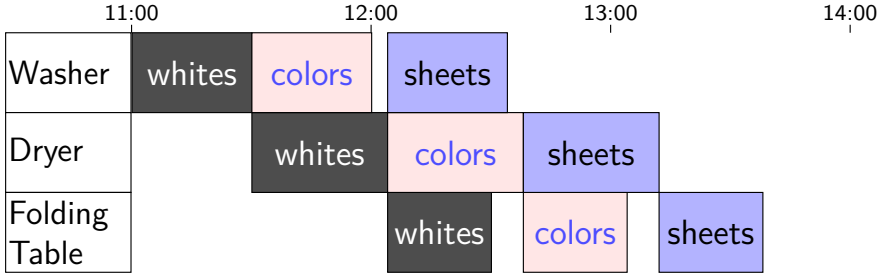
Pipelining

Human pipeline: laundry

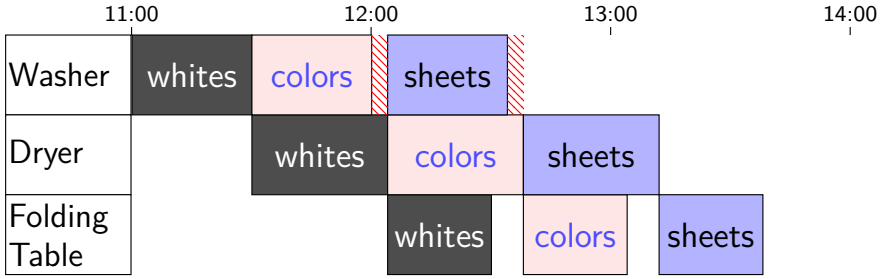


Human pipeline: laundry

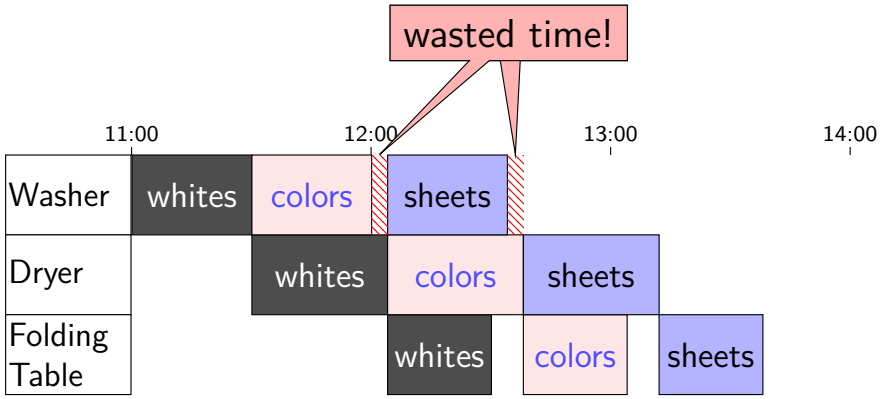




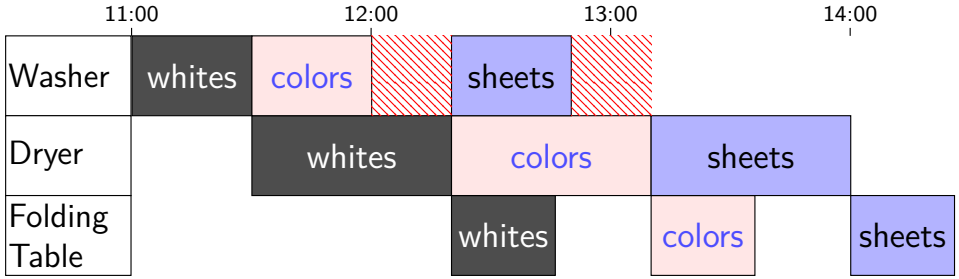
Waste (1)



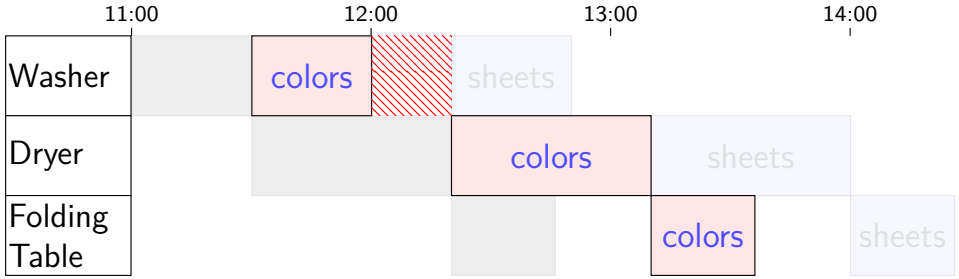
Waste (1)



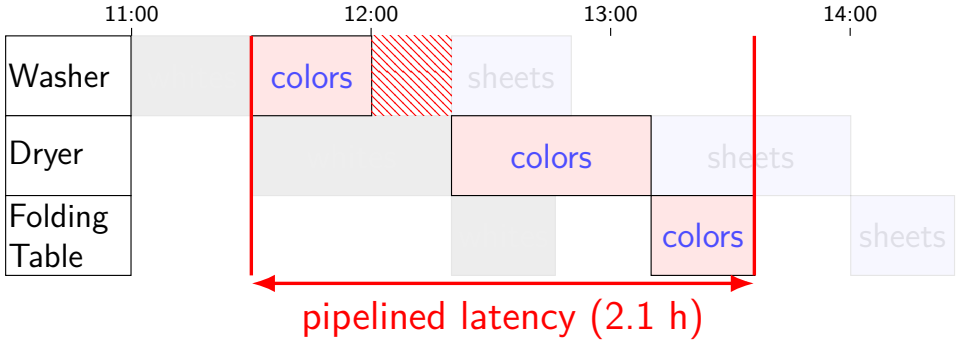
Waste (2)



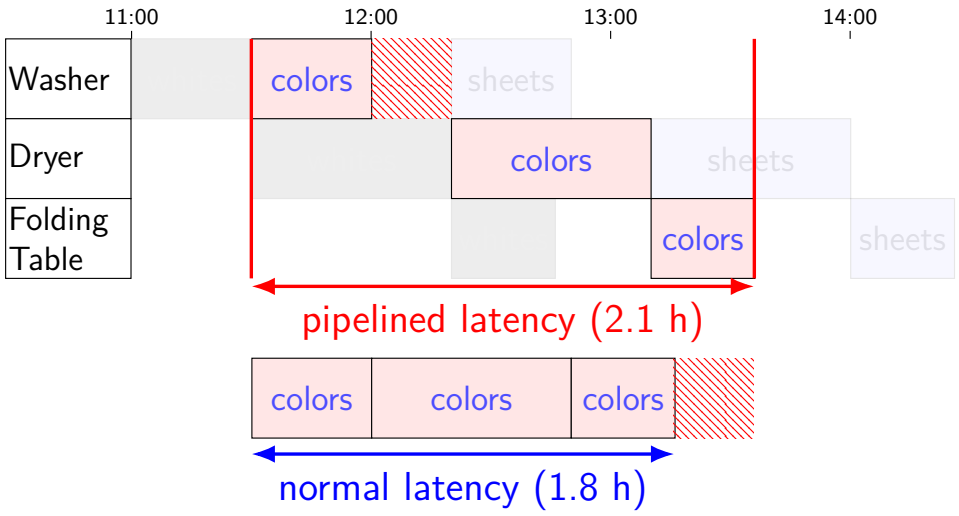
Latency — Time for One



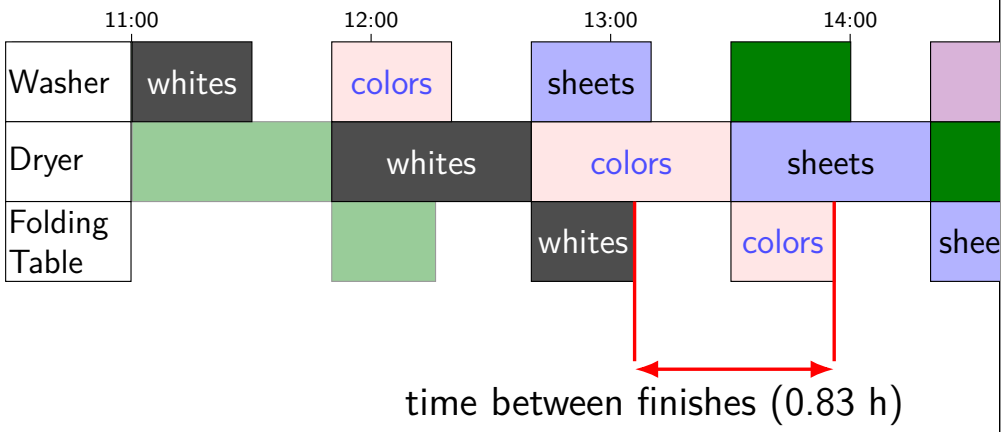
Latency — Time for One



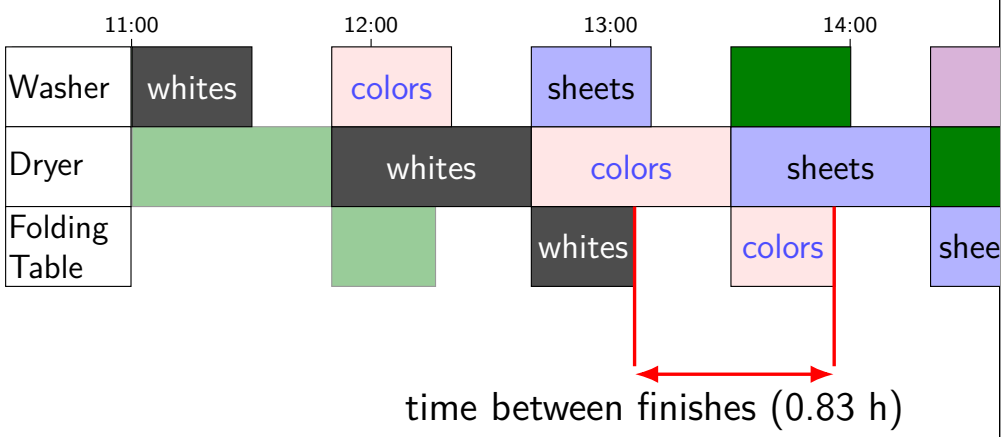
Latency — Time for One



Throughput — Rate of Many

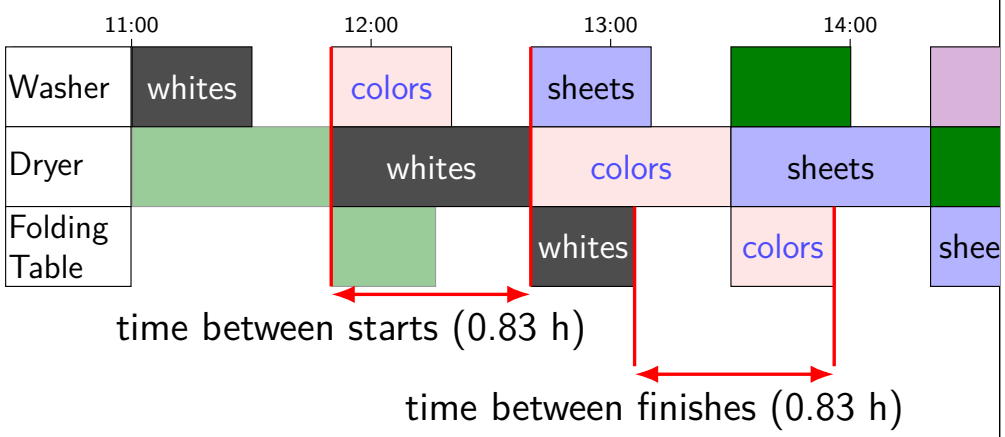


Throughput — Rate of Many



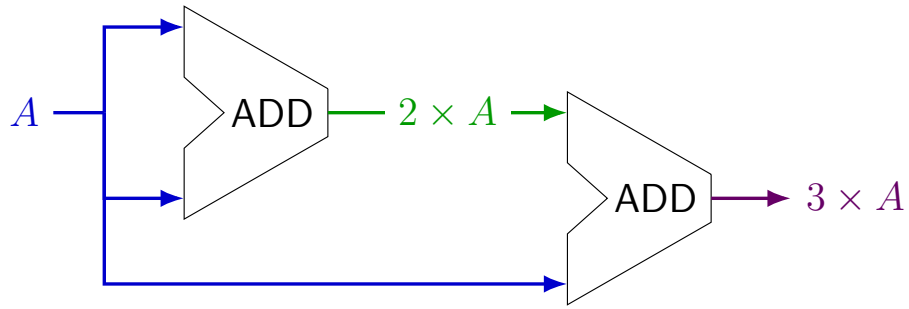
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many

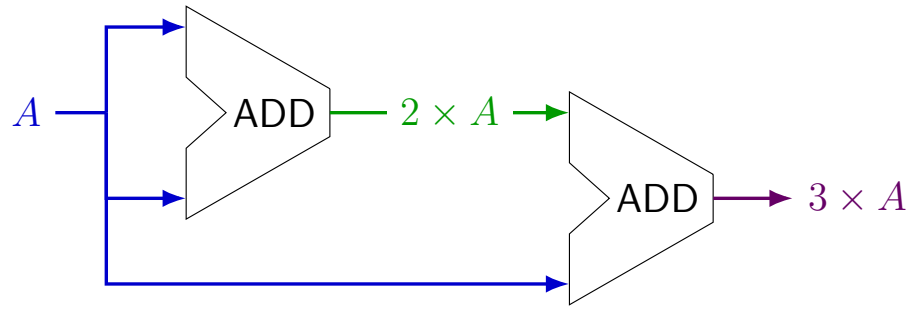


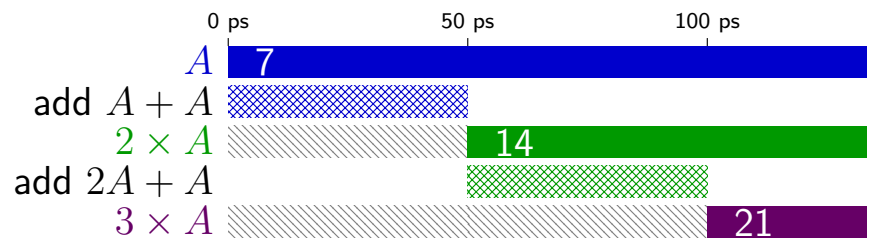
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

times three circuit

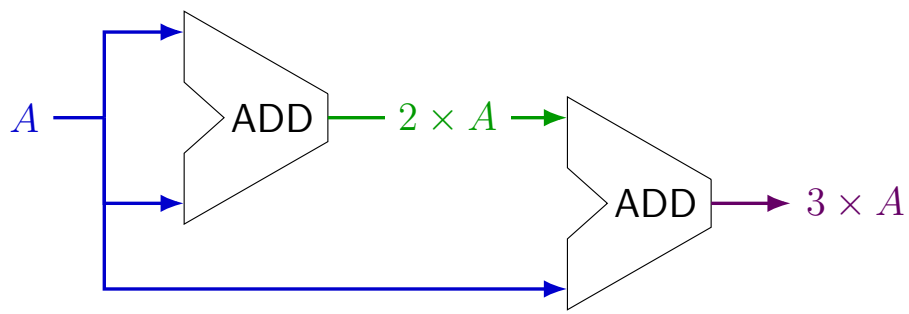


times three circuit

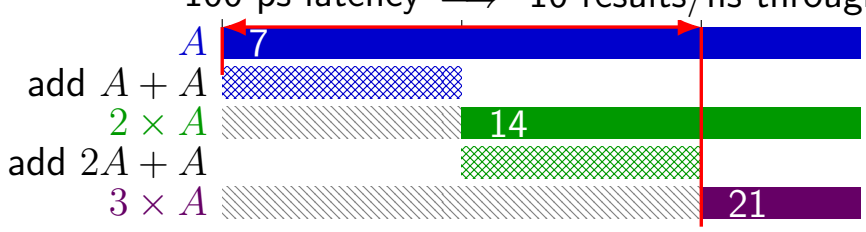




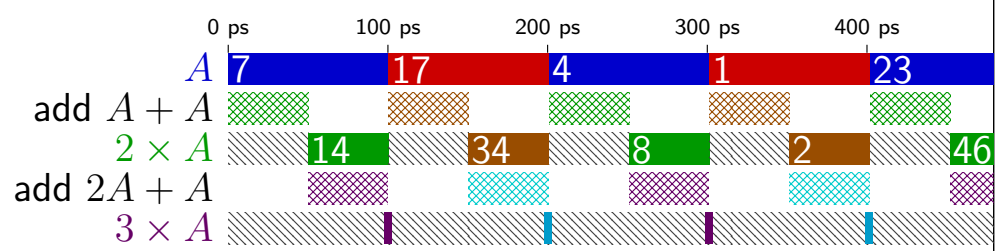
times three circuit



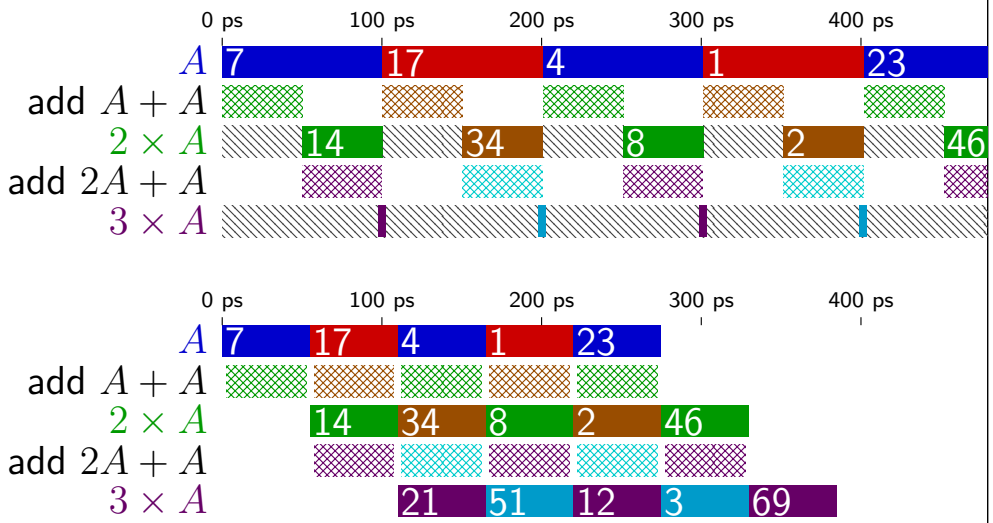
100 ps latency $\Longrightarrow$ 10 results/ns throughput



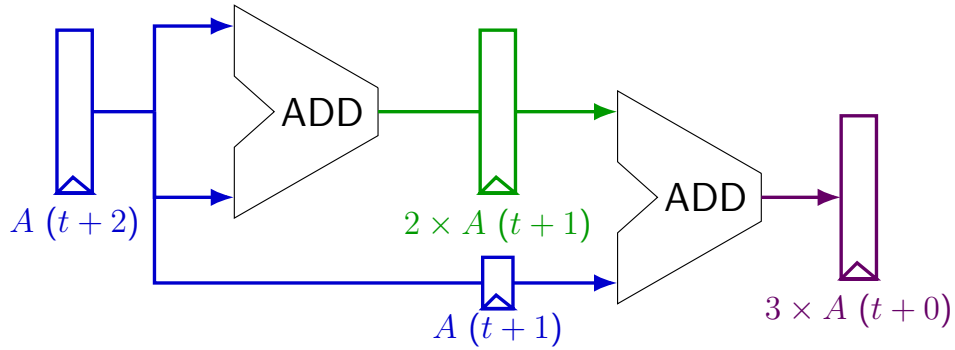
times three and repeat



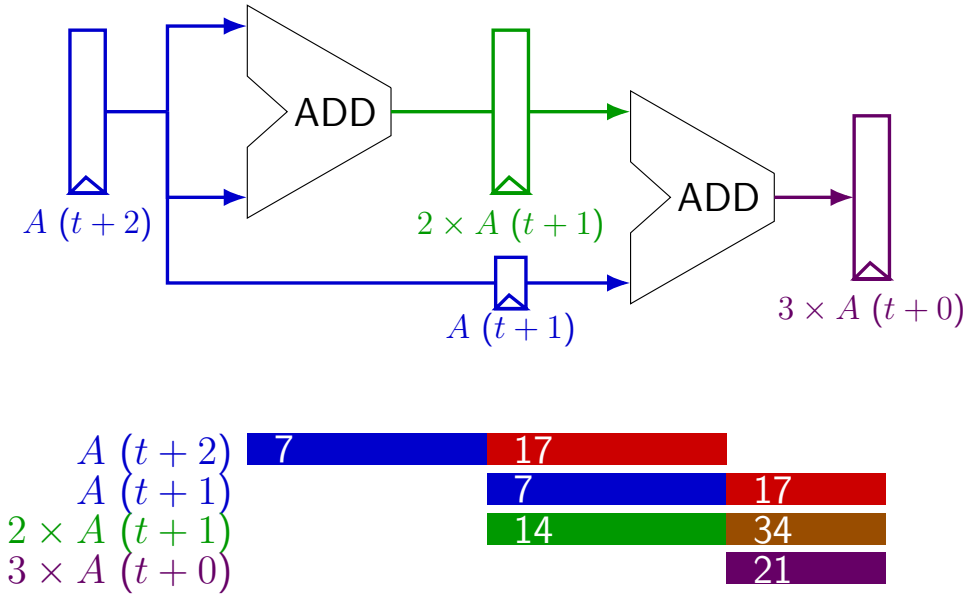
times three and repeat



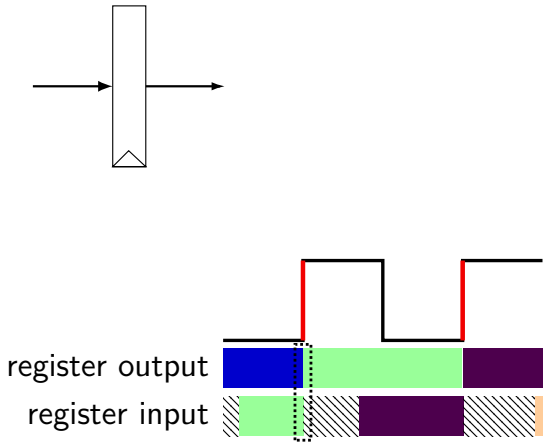
pipelined times three



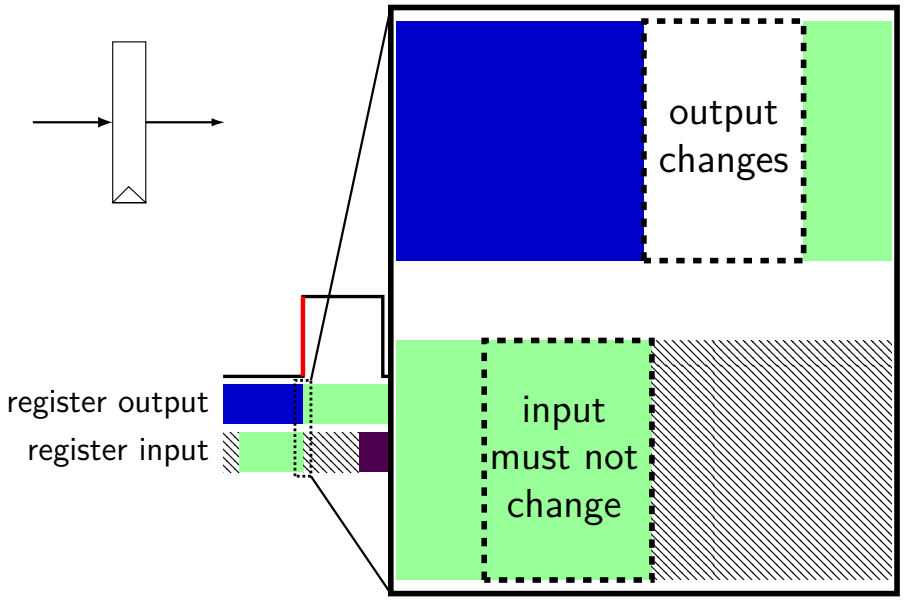
pipelined times three



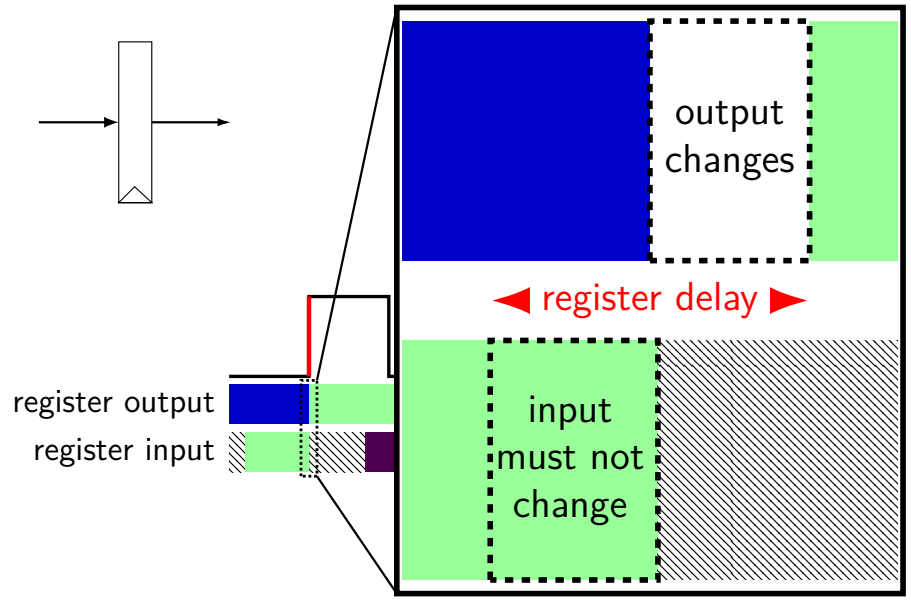
register tolerances



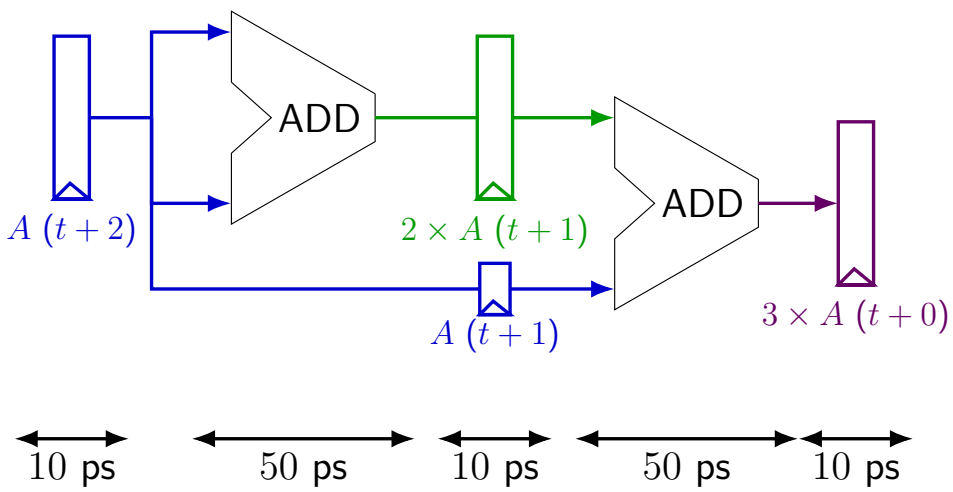
register tolerances



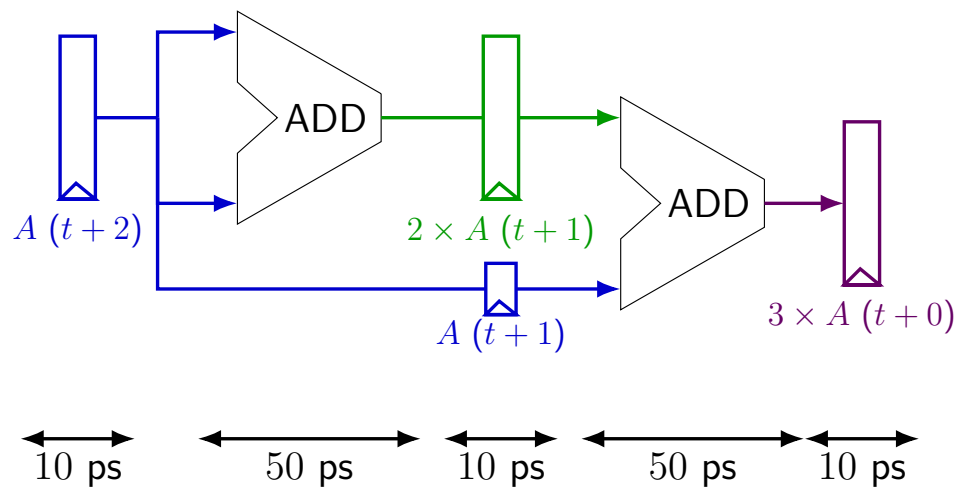
register tolerances



times three pipeline timing

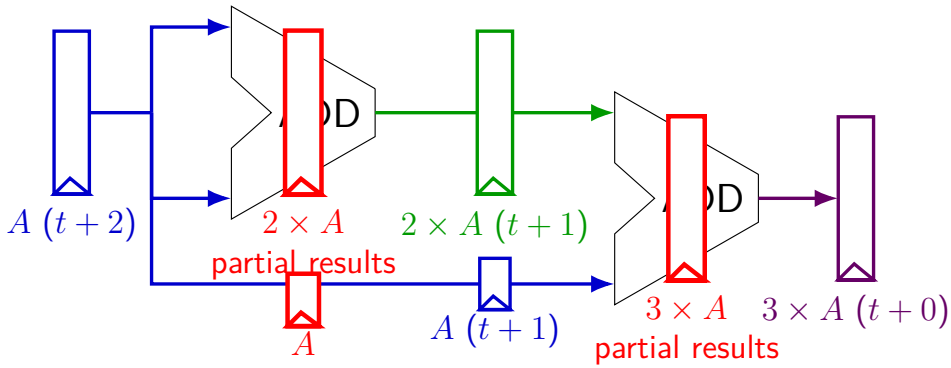


times three pipeline timing

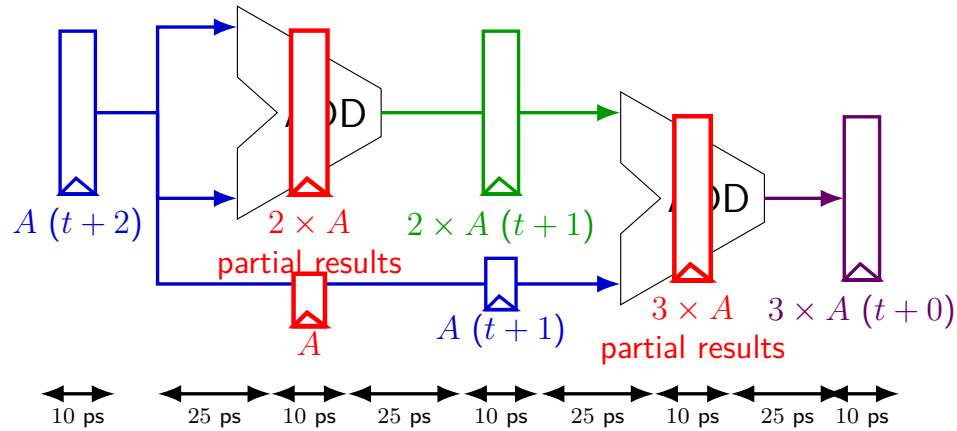


throughput: $\frac{1}{60\ \text{ps}} \approx 16$ G operations/sec

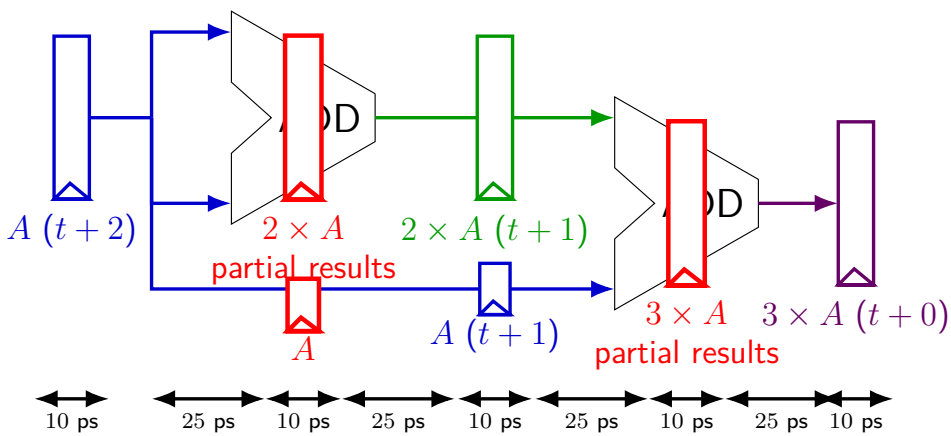
deeper pipeline



deeper pipeline

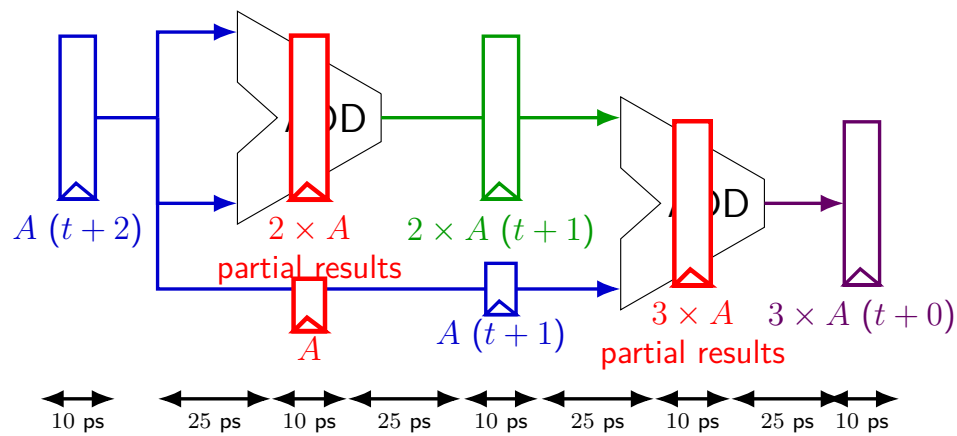


deeper pipeline



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

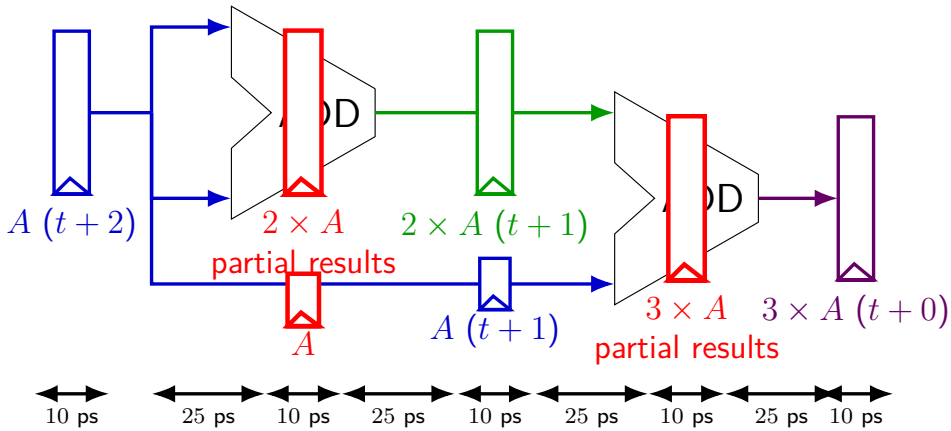
deeper pipeline



Problem: How much faster can we get?

Problem: Can we even do this?

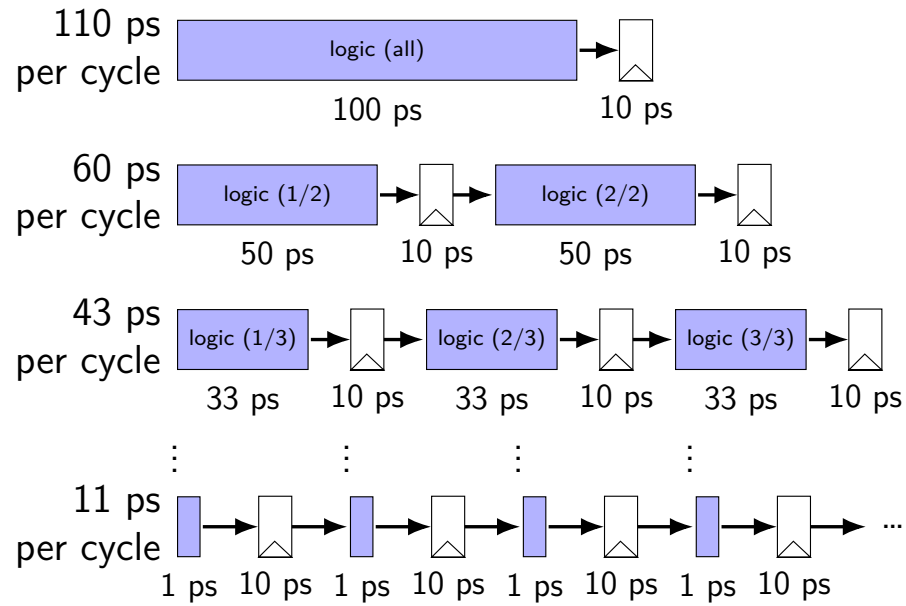
deeper pipeline



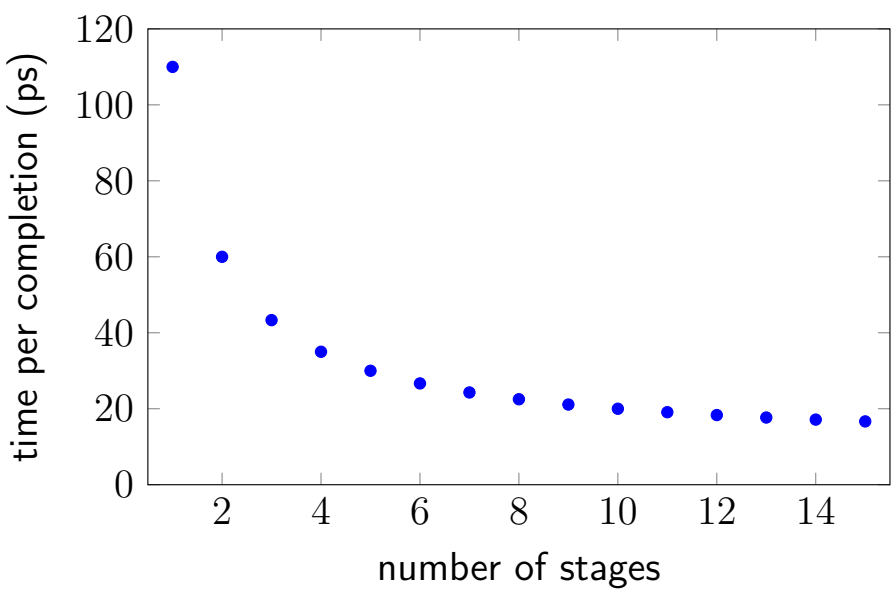
Problem: How much faster can we get?

Problem: Can we even do this?

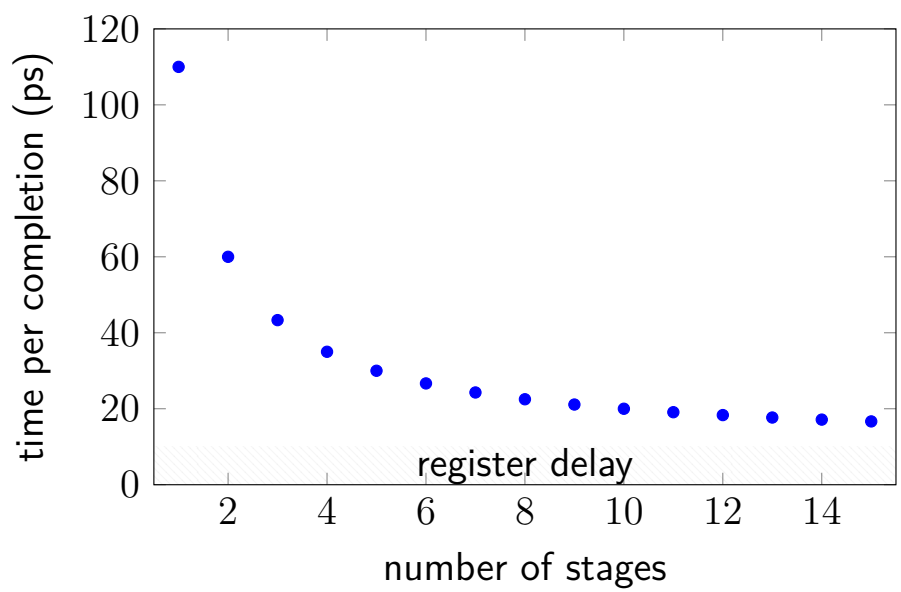
diminishing returns: register delays



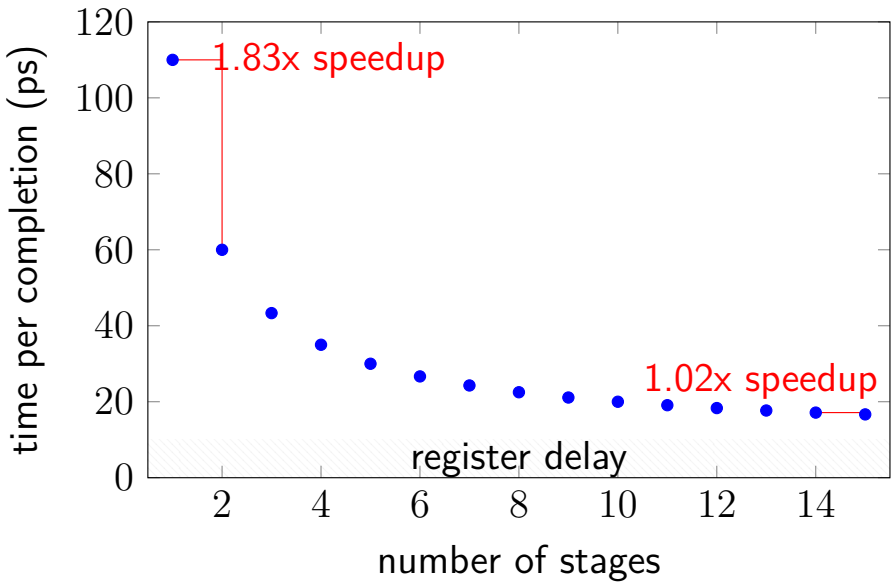
diminishing returns: register delays



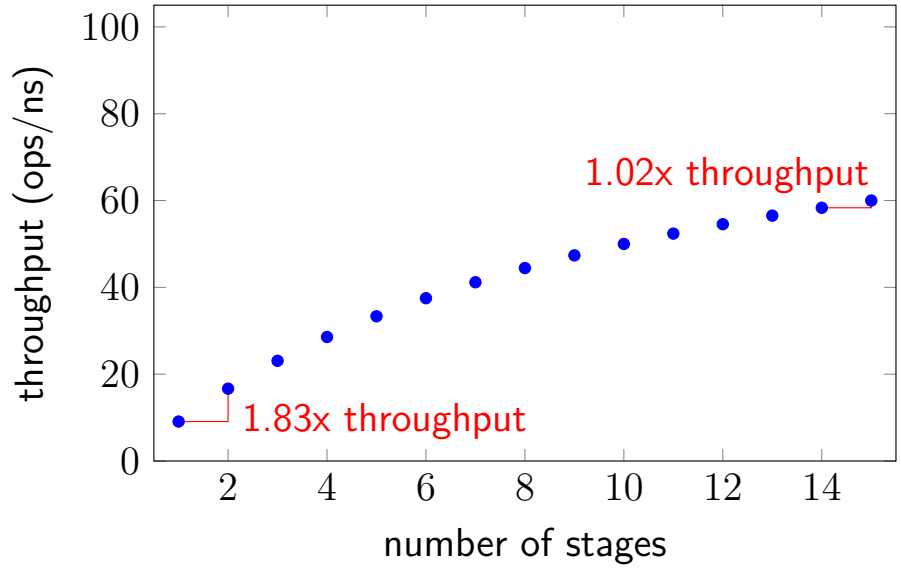
diminishing returns: register delays



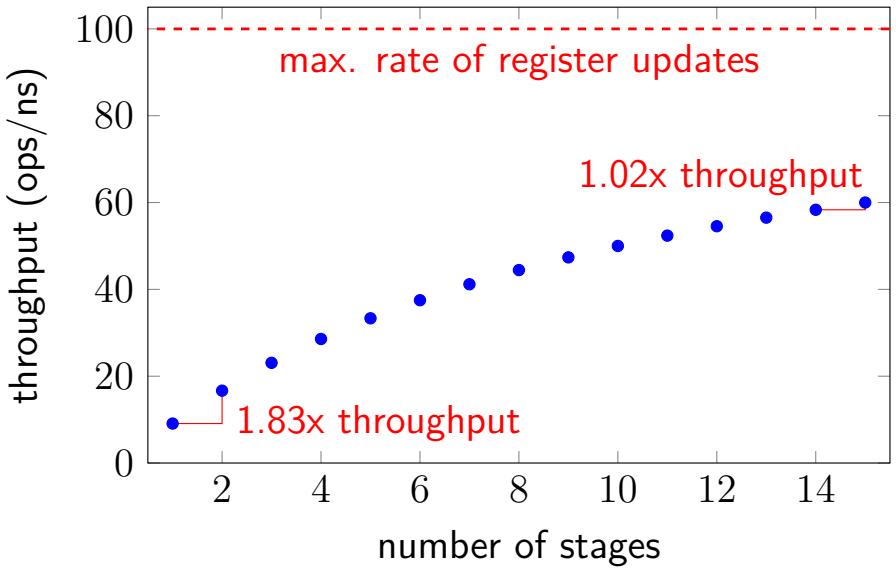
diminishing returns: register delays



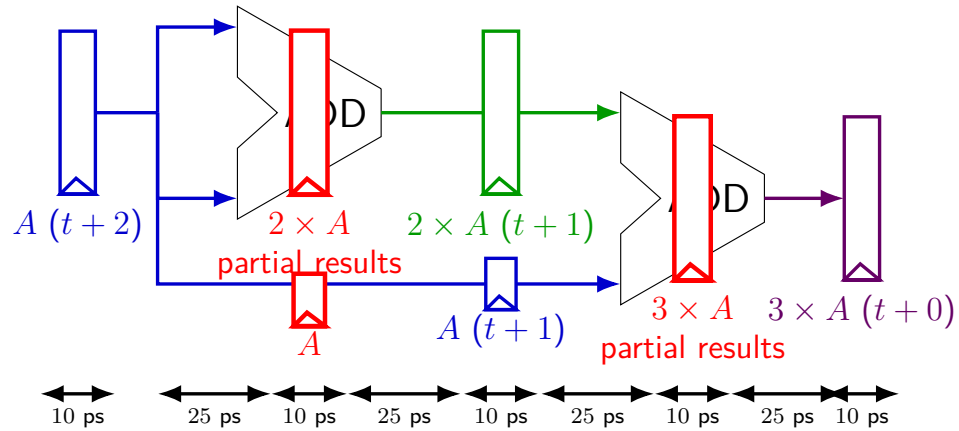
diminishing returns: register delays



diminishing returns: register delays



deeper pipeline



Problem: How much faster can we get?

Problem: Can we even do this?

diminishing returns: uneven split

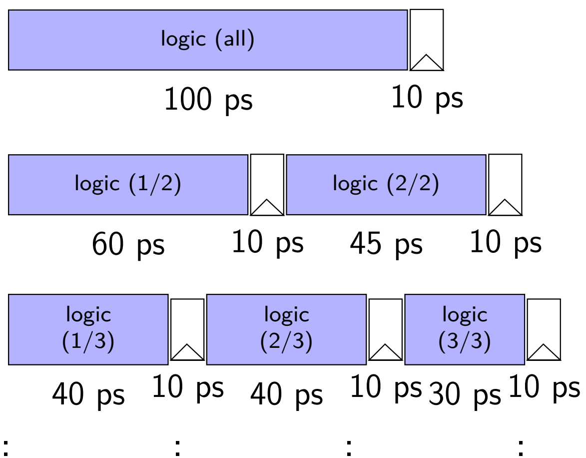
Can we split up some logic (e.g. adder) arbitrarily?

Probably not...

110 ps
per cycle

70 ps
per cycle

50 ps
per cycle



---

diminishing returns: uneven split

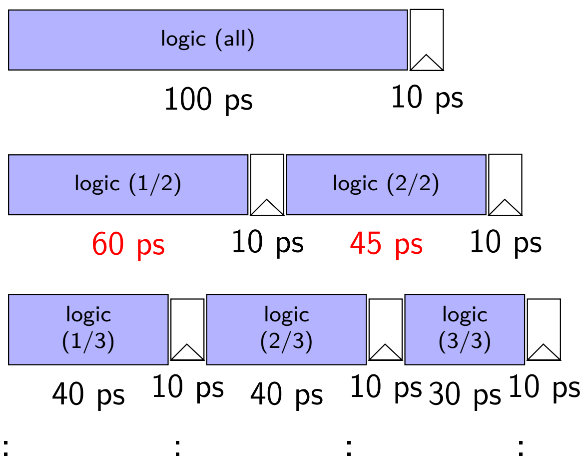
Can we split up some logic (e.g. adder) arbitrarily?

Probably not...

110 ps
per cycle

70 ps
per cycle

50 ps
per cycle



---

diminishing returns: uneven split

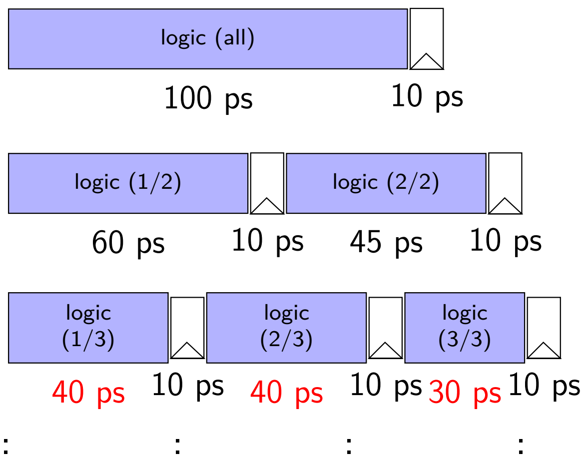
Can we split up some logic (e.g. adder) arbitrarily?

Probably not...

110 ps
per cycle

70 ps
per cycle

50 ps
per 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

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

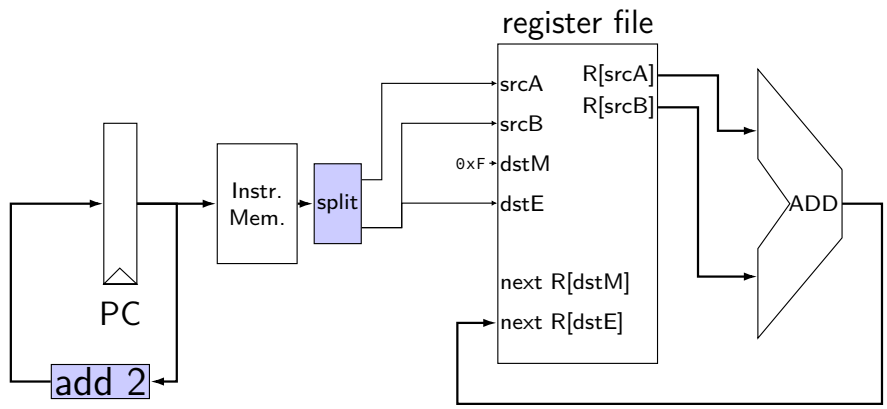
Execute: arithmetic (ALU)

Memory: read/write data memo

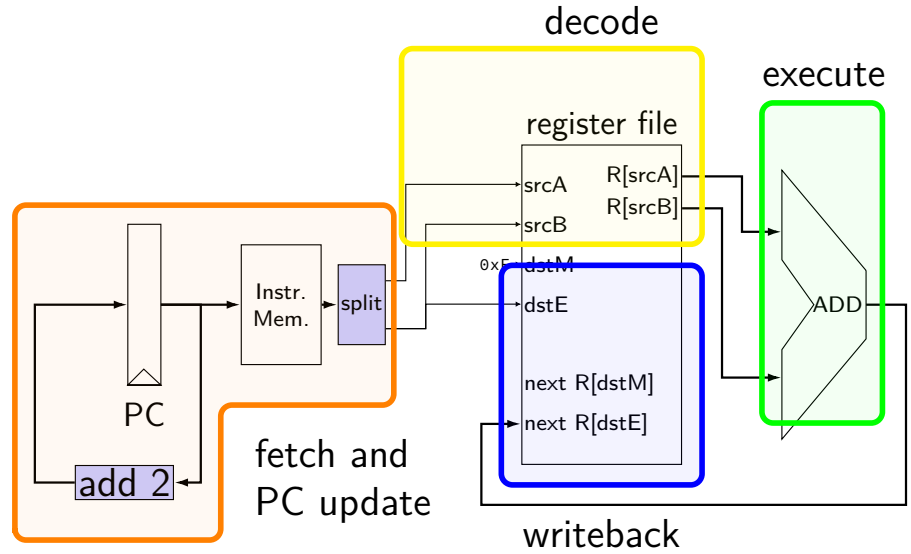
Writeback: write register file

5 stages
one instruction in each
compute next to start
immediatelly

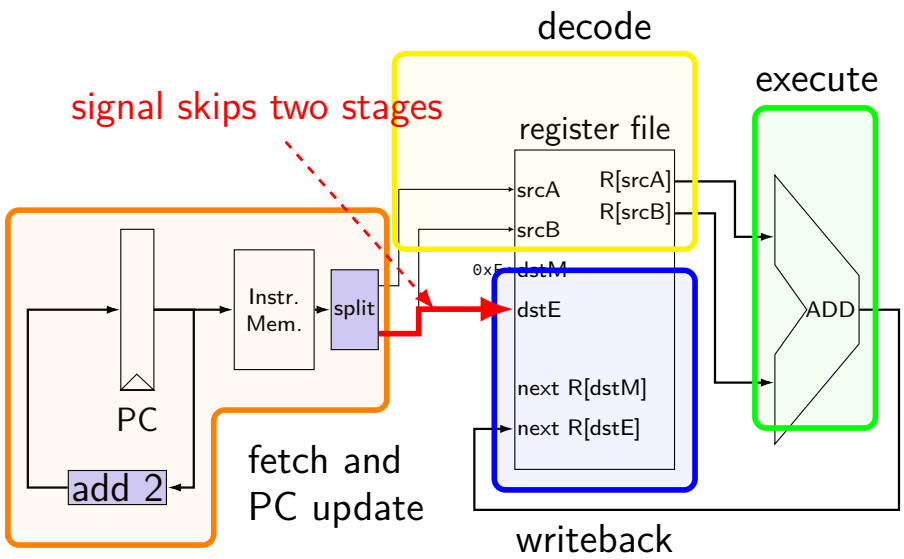
addq processor



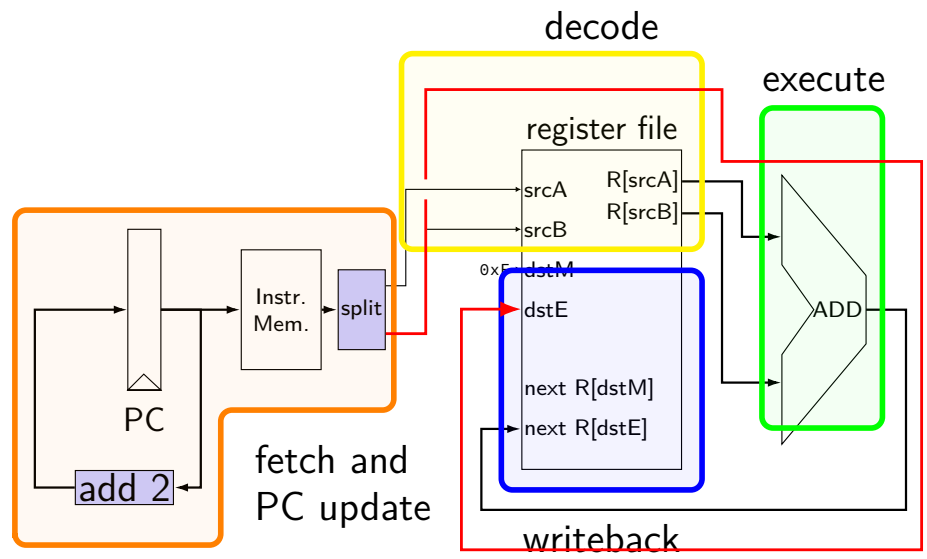
addq processor



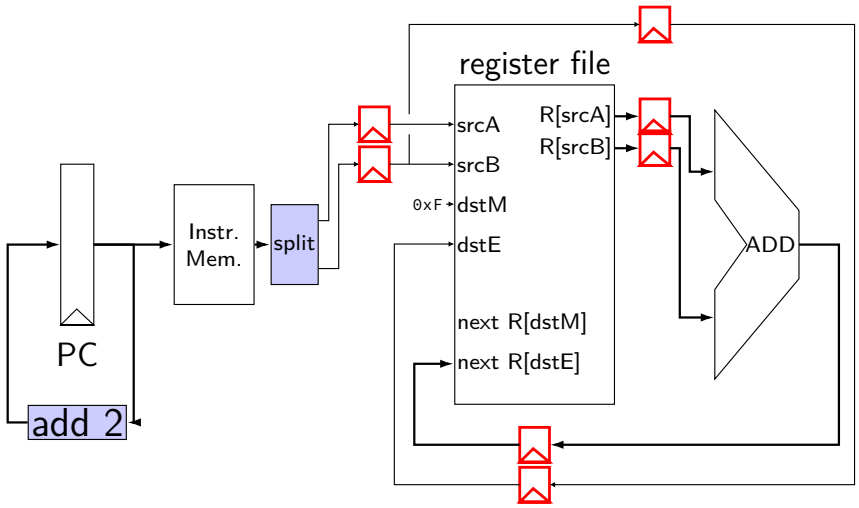
addq processor



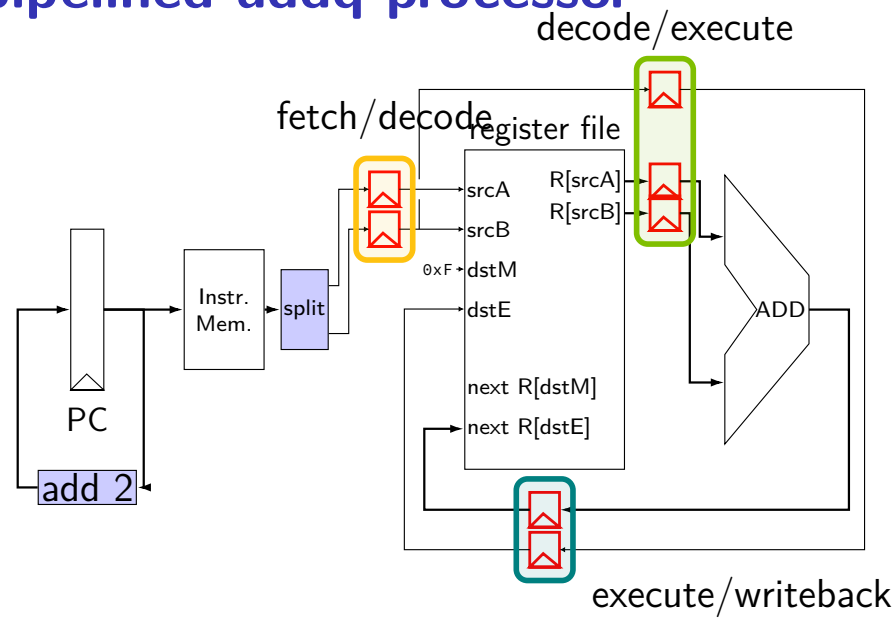
addq processor



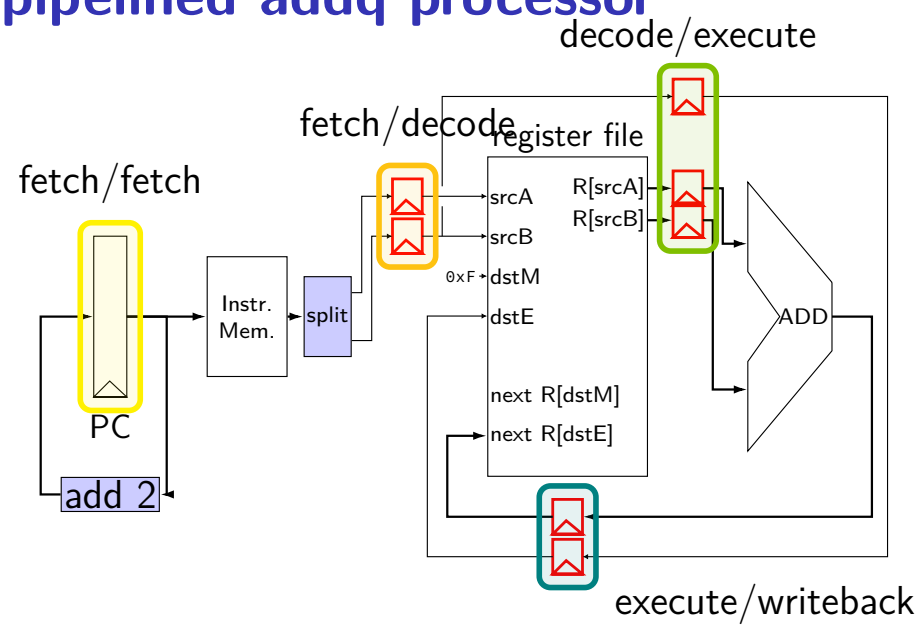
pipelined addq processor



pipelined addq processor

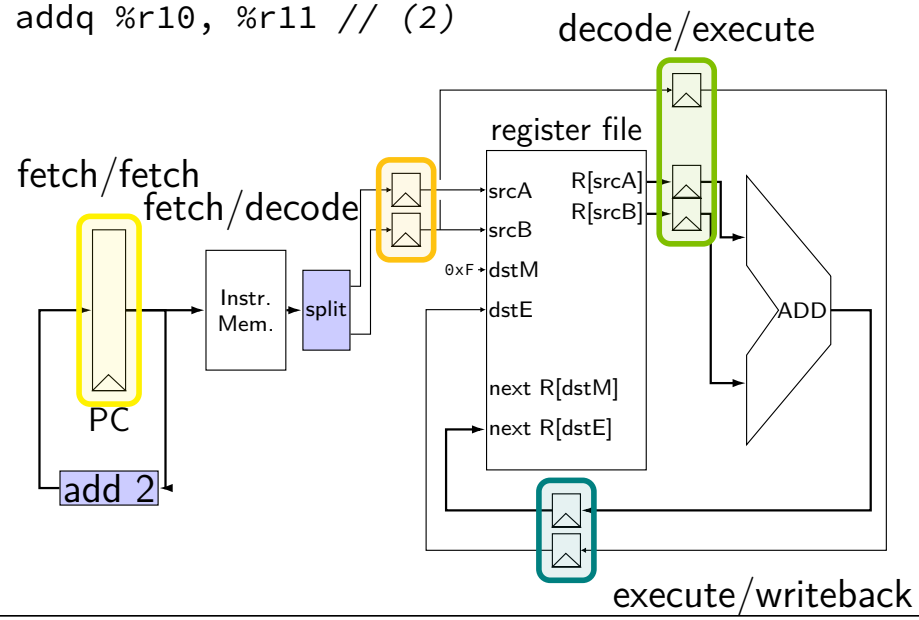


pipelined addq processor



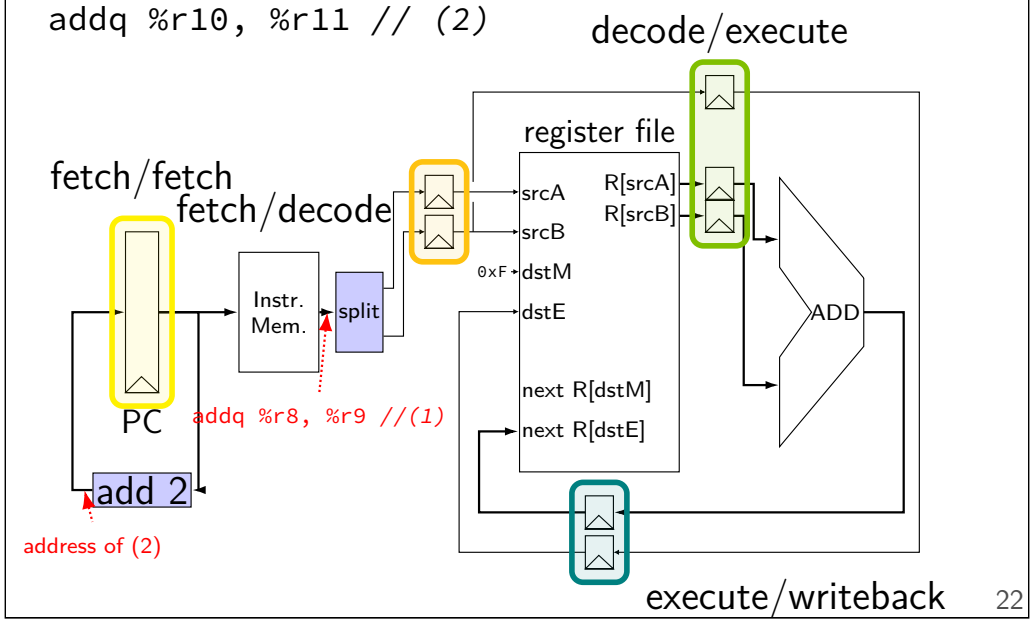
addq execution

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



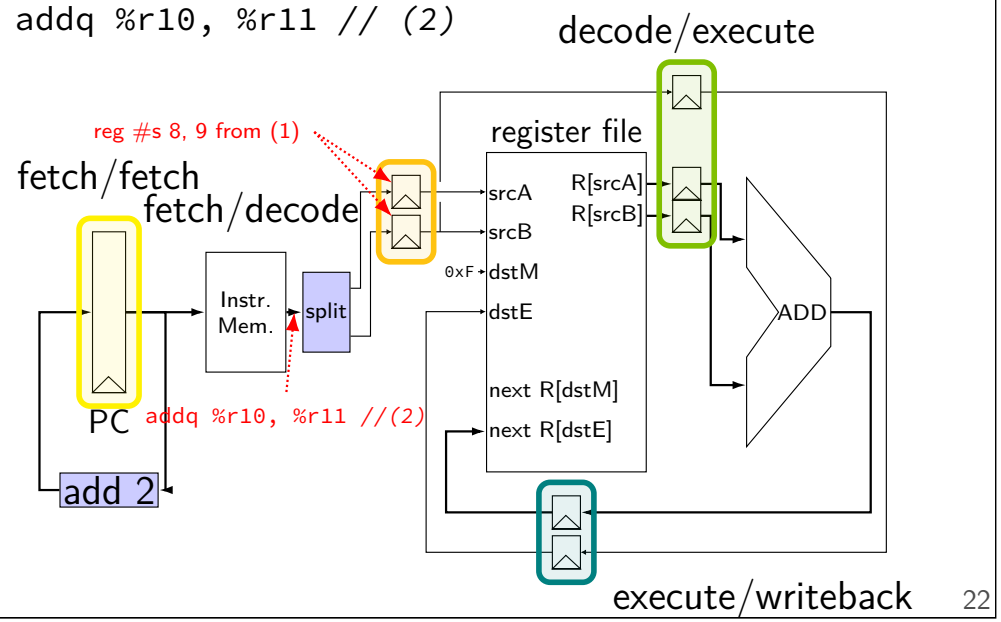
addq execution

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



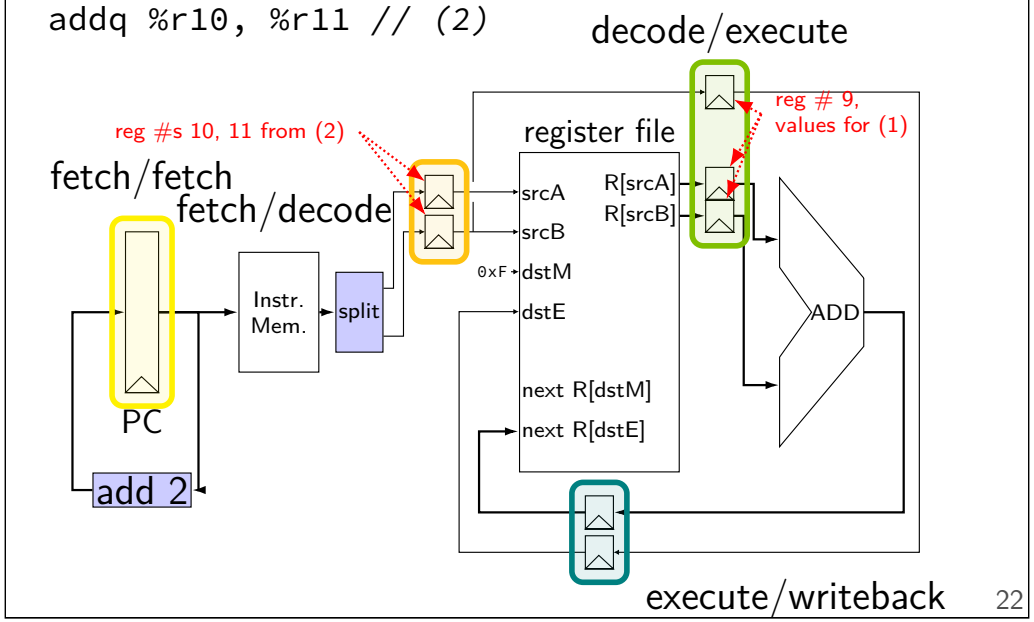
addq execution

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



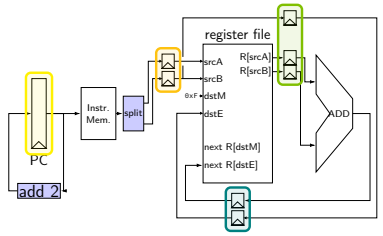
addq execution

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



addq processor timing

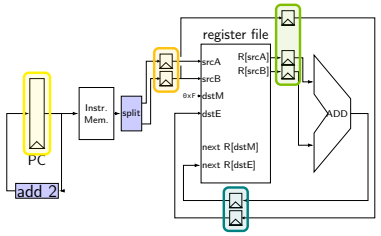
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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
```

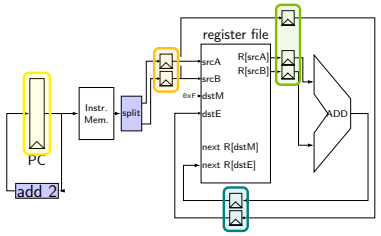


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

---

addq processor timing

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

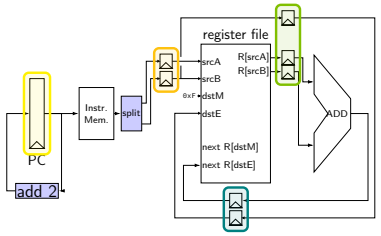


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

---

addq processor timing

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

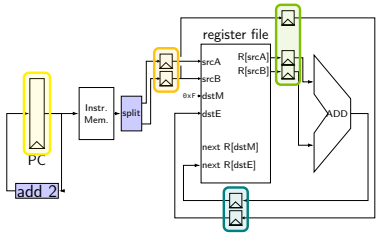


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

---

addq processor timing

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

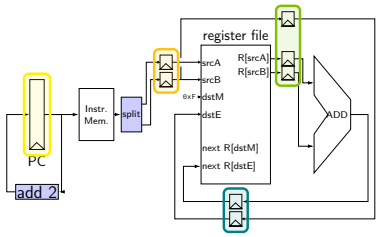


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor performance

example delays:

path	time
add 2 (PC update)	80 ps
instruction memory	200 ps
register file read	150 ps
add	100 ps
register file write	150 ps



no pipelining: 1 instruction per 600 ps

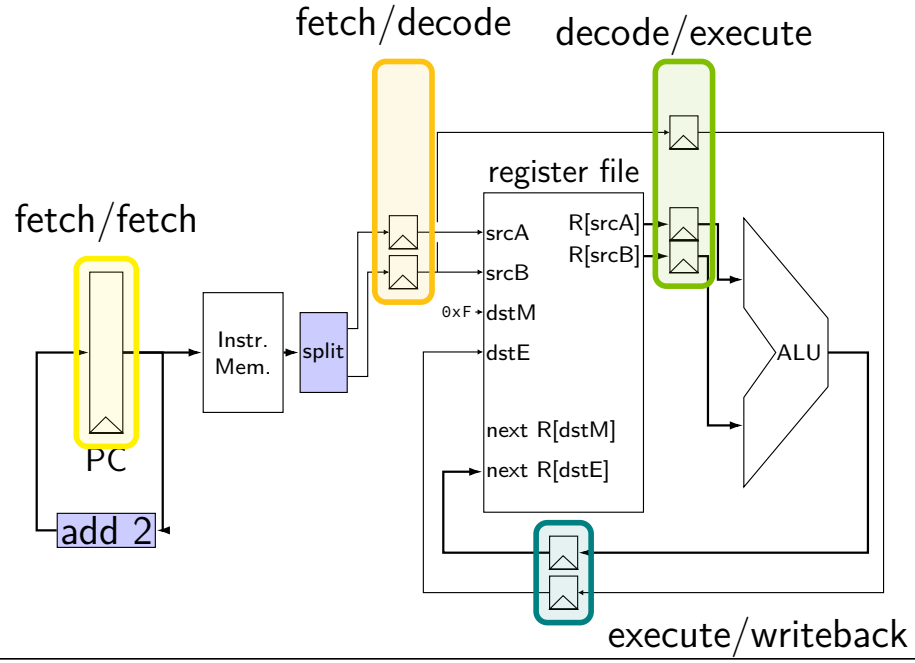
add up everything but add 2 (critical (slowest) path)

pipelining: 1 instruction per 200 ps + register delay

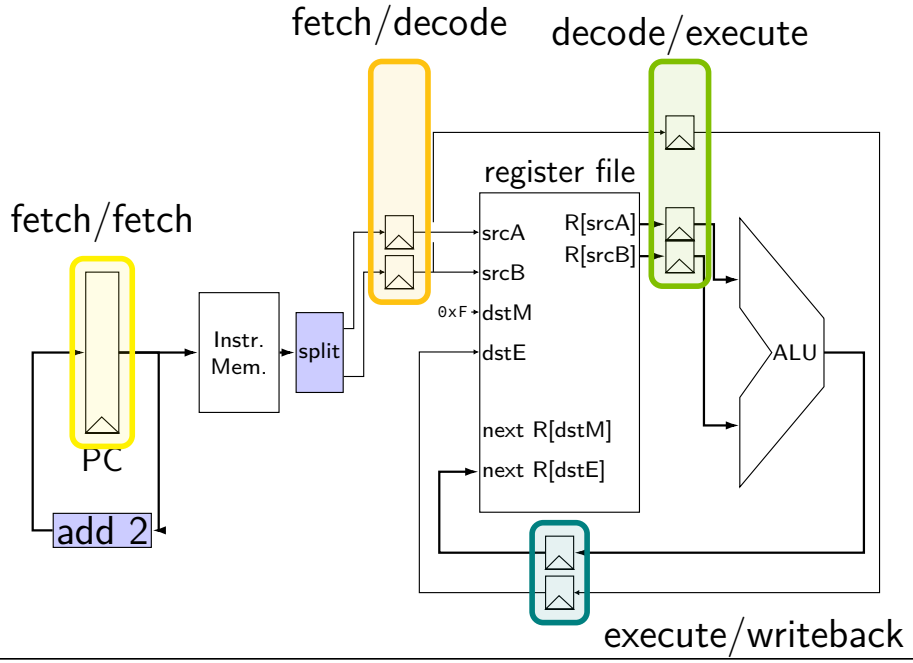
slowest path through stage + register delay

latency: 800 ps + register delay (4 cycles)

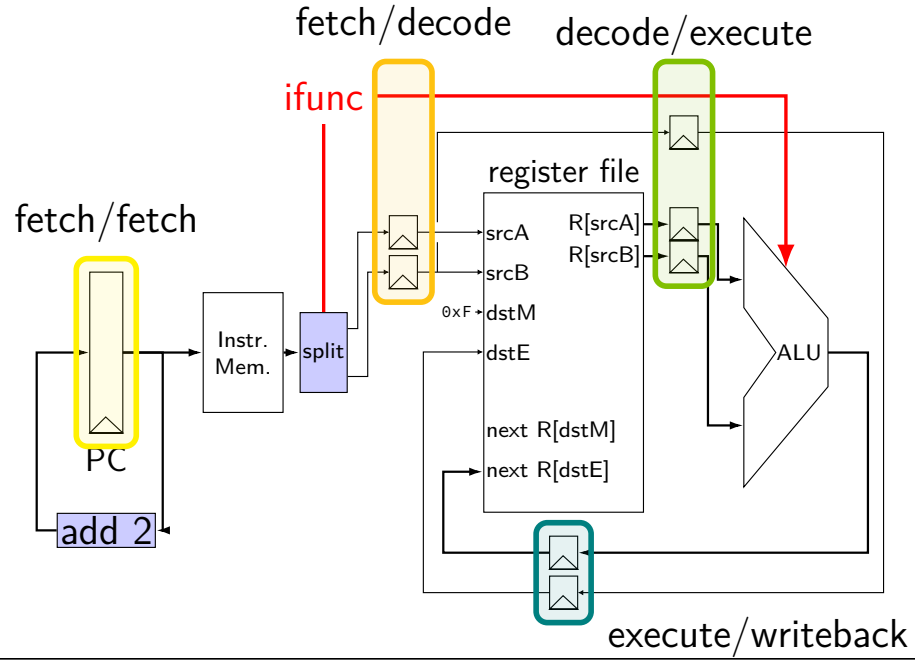
OPq processor



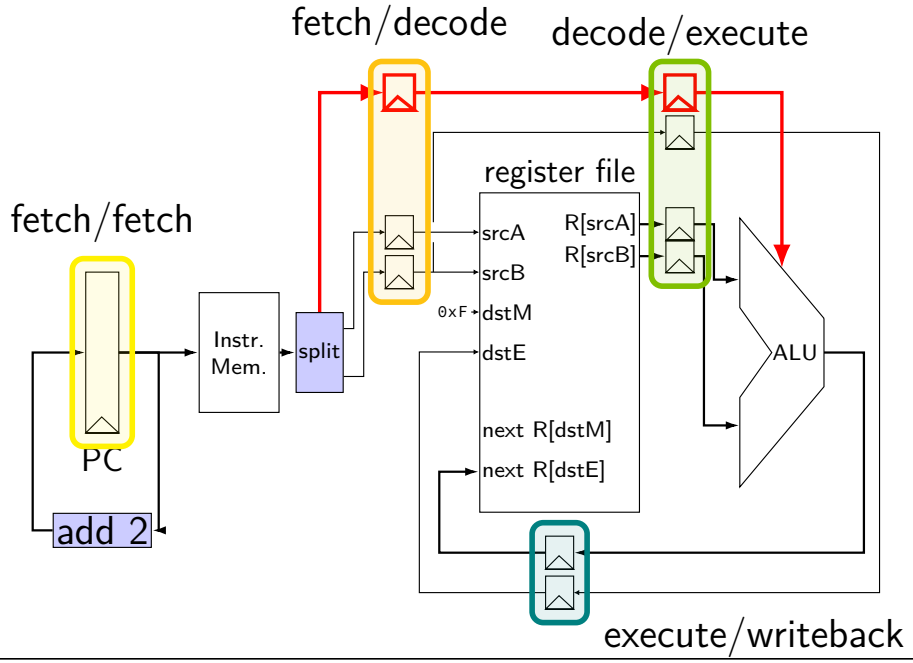
OPq processor



OPq processor

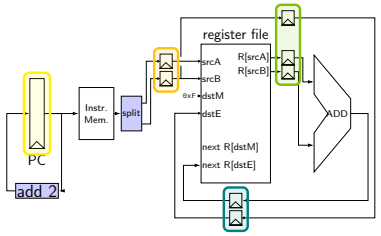


OPq processor



addq processor: data hazard

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



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

should be 1700

data hazard

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

step#	pipeline implementation	ISA specification
1	read r8, r9 for (1)	read r8, r9 for (1)
2	read r9, r8 for (2)	write r9 for (1)
3	write r9 for (1)	read r9, r8 for (2)
4	write r8 for (2)	write r8 ror (2)

pipeline reads older value...

instead of value ISA says was just written

data hazard compiler solution

```
addq %r8, %r9
nop
nop
addq %r9, %r8
```

one solution: change the ISA

all addqs take effect three instructions later

make it compiler's job

usually not acceptable

data hazard hardware solution

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

how about hardware add nops?

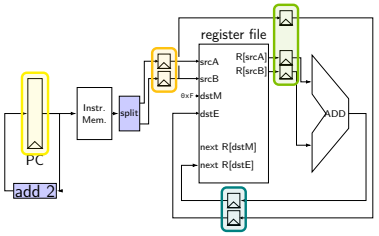
called stalling

extra logic:

- sometimes don't change PC
- sometimes put do-nothing values in pipeline registers

addq processor: data hazard stall

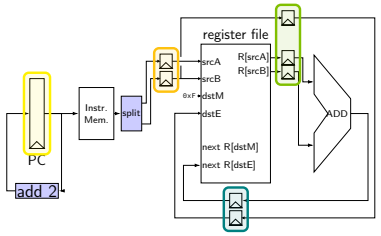
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
```



	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		9	8	---	---	F	---	F
5				1700	800	8	---	F
6							2500	8

addq processor: data hazard stall

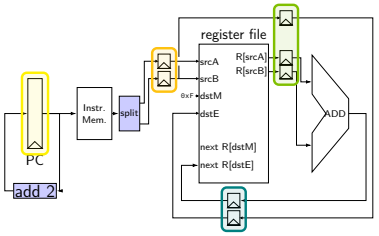
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
```



	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	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4		9	8	---	---	F	---	F
5				1700	800	8	---	F
6							2500	8

addq processor: data hazard stall

```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
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	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4		9	8	---	---	F	---	F
5				1700	800	8	---	F
6							2500	8

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

control hazard

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

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

control hazard

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

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

control hazard: stall

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

wait for two cycles for addq to update SF/ZF

	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	F	F	800	900	9		
3	0x2	0/0	F	F	---	---	F	1700	9
4	0x10	0/0	F	F	---	---	F	---	F
5			10	11	---	---	F	---	F
6					1000	1100	11	---	F

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*	execute je instruction (use SF/ZF)							
2	0x2*	0/1	F	F	800	900	9		
3	0x2	0/0	F	F	---	---	F	1700	9
4	0x10	0/0	F	F	---	---	F	---	F
5			10	11	---	---	F	---	F
6					1000	1100	11	---	F

pipelined Y86 CPU

five stages — fetch+PC update/decode/execute/memory/writeback

one per cycle

need: pipeline registers between stages

need: way of dealing with control hazards

need: way of dealing with data hazards

- stalling + two techniques we'll take about next week

pipelining summary

assembly line for math

- divide into pieces
- each piece in parallel for different instructions

increase throughput but also increase latency

limited by uneven division of work

limited by dependencies ("hazards")

limited by register delays

register operations when stalling

"stall" — write disable; keep old value

"bubble" — write default (no-operation) instead of input

HCL2D will provides these directly

if it didn't — MUX in front of register input