pipelining

last time

Y86-64 single-cycle design 'stages'

textbook convention of E port for ALU + M port for memory

usually values being computed not in use

ALU used for OPq + (non-PC) address computations

special cases:

- reading/writing %rsp for stack instructions
- using old %rsp versus new %rsp for memory access
- writing value of PC + increment for CALL
- read/write enable on data memory

textbook trick: rrmovq+irmovq compute value+0 in ALU

general idea: add/set MUXes for each instruction's needs

quiz Q2

setup: popq is now B0 (first byte), [rA]F (second byte)

want: B[register] (one byte)

change: register comes from new part of instruction (not rA/rB)

we only write to this register

quiz Q3

reading machine code:

- done when PC (address to read from) is available

reading from the register file

- done when register index is available
- register index not available until machine code is

writing memory

- done when rising edge of clock happens
- address + value are setup earlier, but not acted on yet

writing registers

- done when rising edge of clock happens
- register index + value are setup earlier, but not acted on yet

quiz Q4

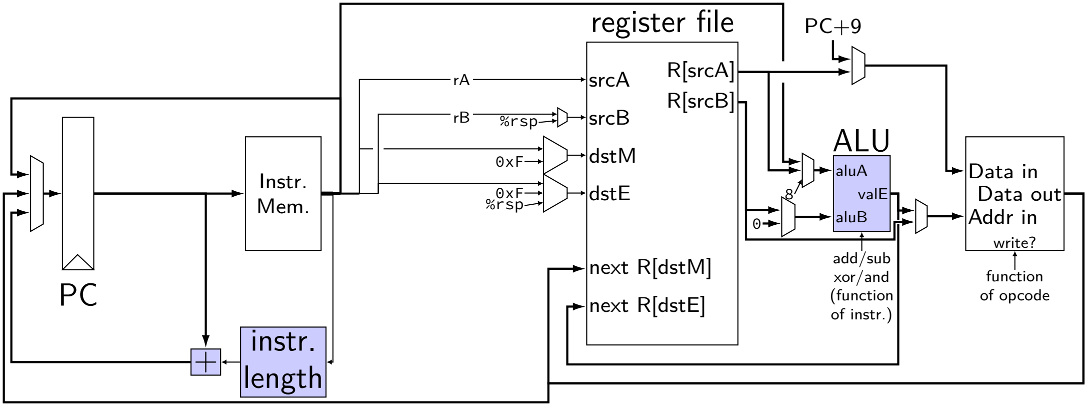
immovq: immediate (constant) to memory move

data memory inputs:

- address `mem_addr`: where to write or read from
- value `mem_input`: what to write (if writing)
- (also `mem_readbit`/`mem_writebit`)

constant is in the machine code

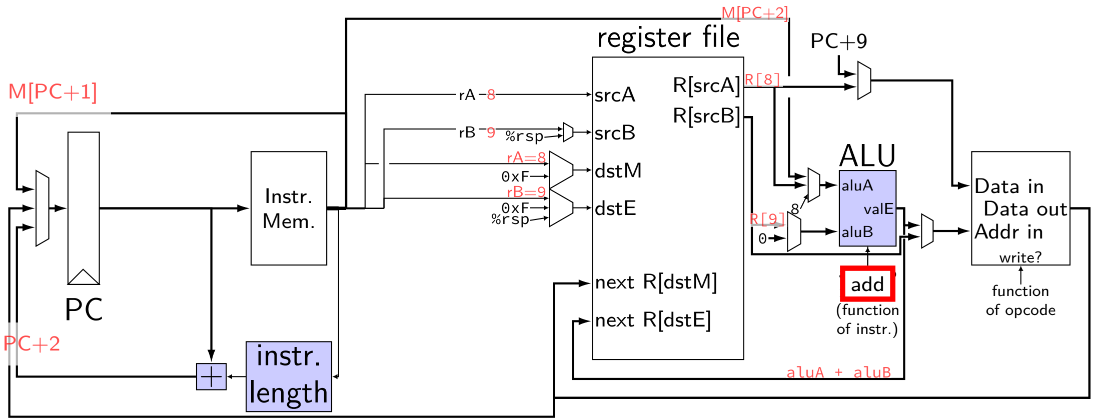
circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...

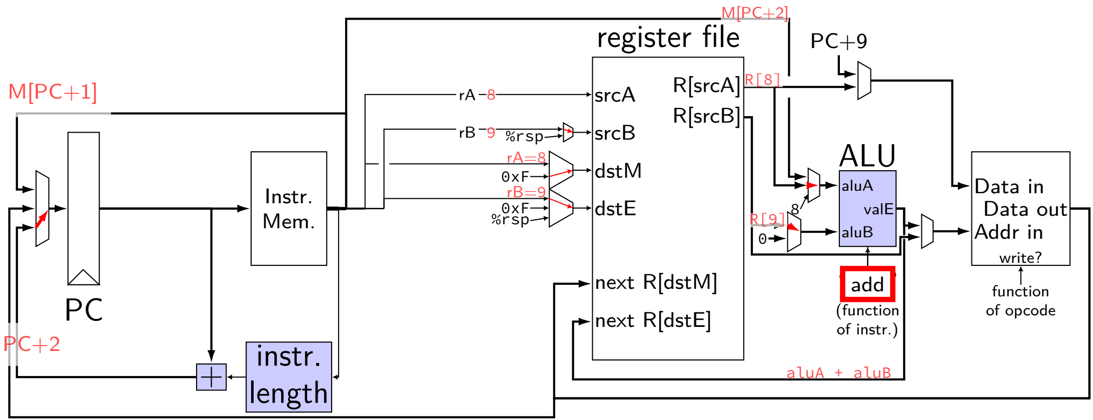
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



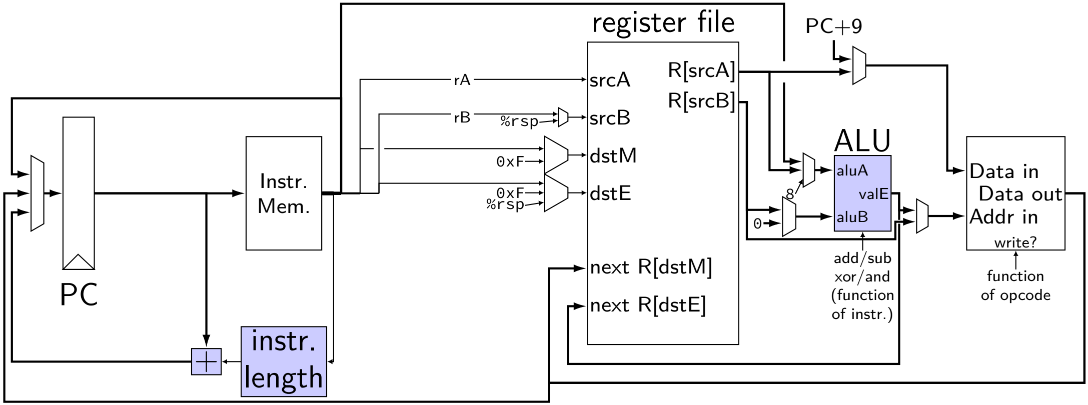
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select when running `addq %r8, %r9`?

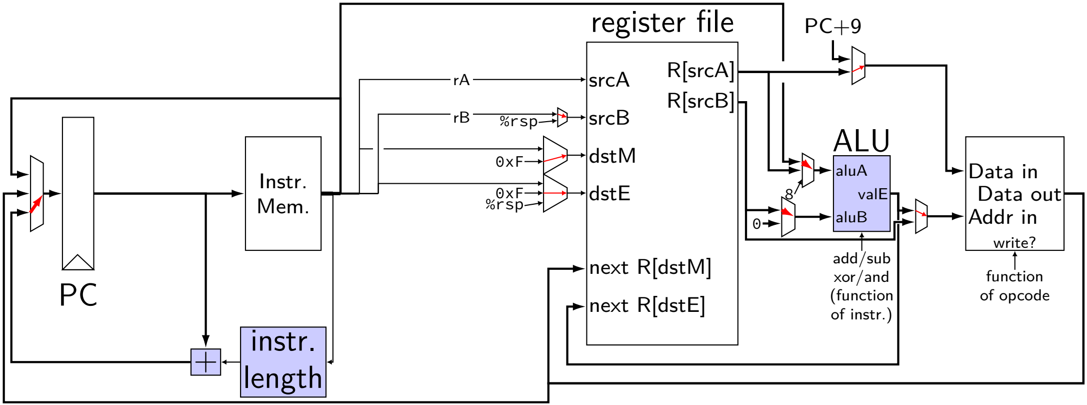
circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...

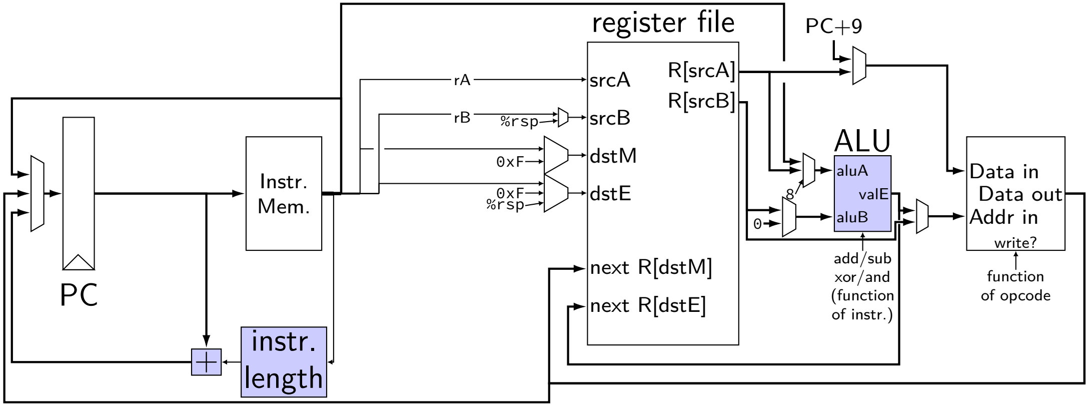
Exercise: what do they select for `rmmovq`?

circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for `rmmovq`?

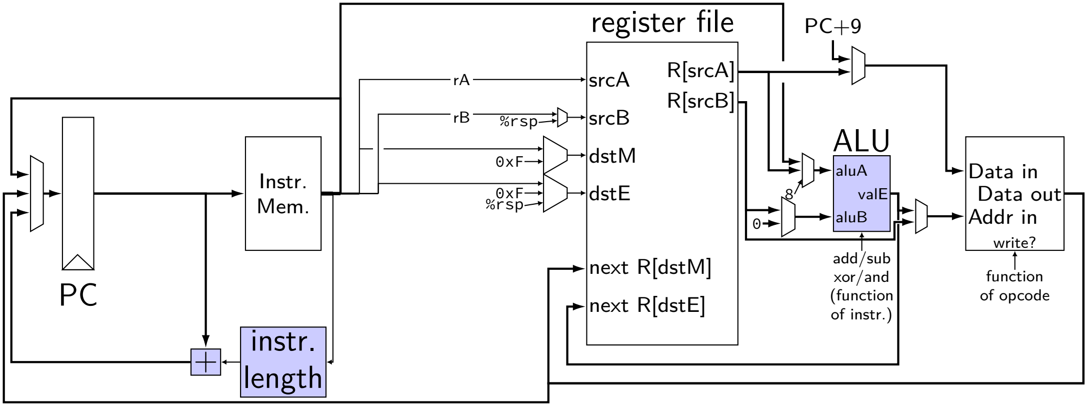
circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …

Exercise: what do they select for `irmovq`?

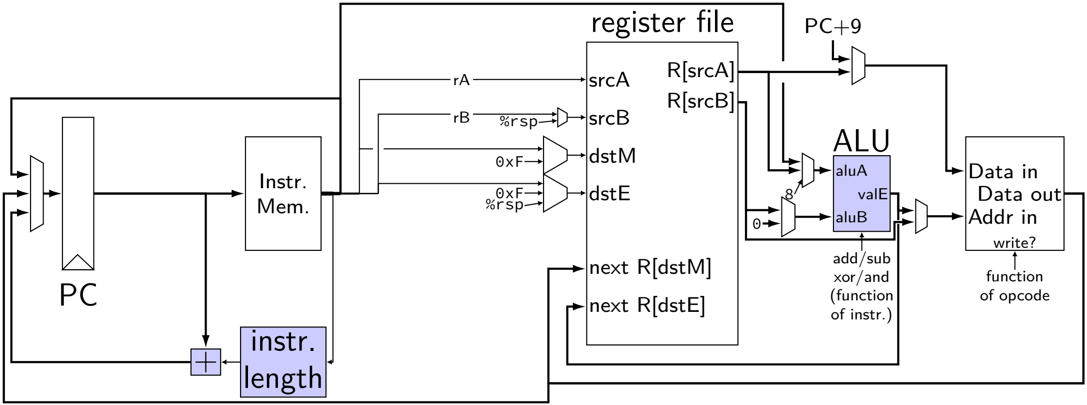
circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …

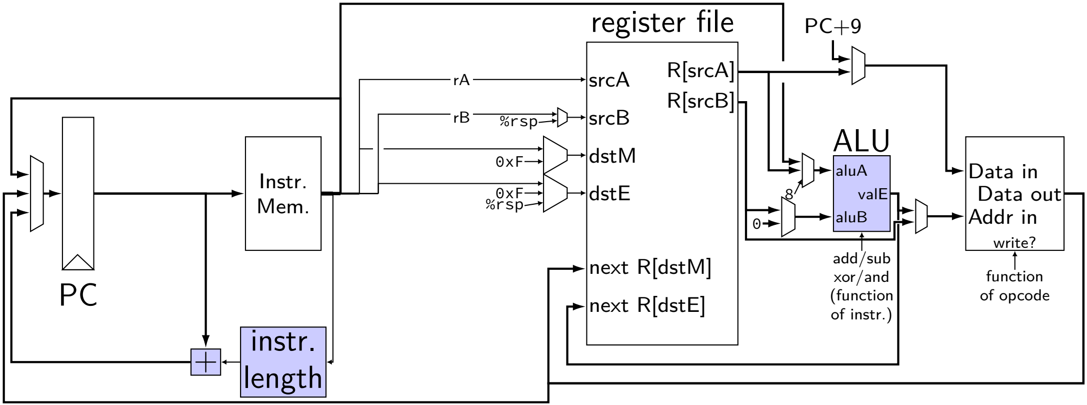
Exercise: what do they select for `mrmovq`?

circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for `jle`?

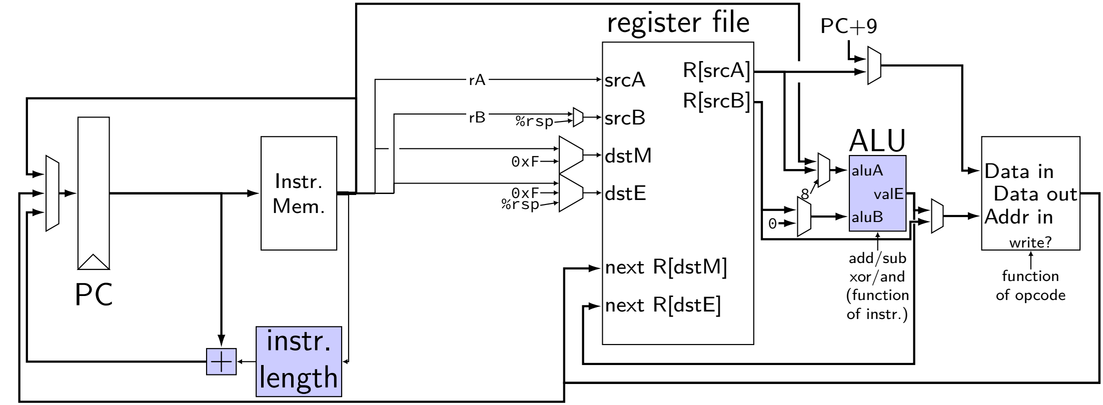
circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …

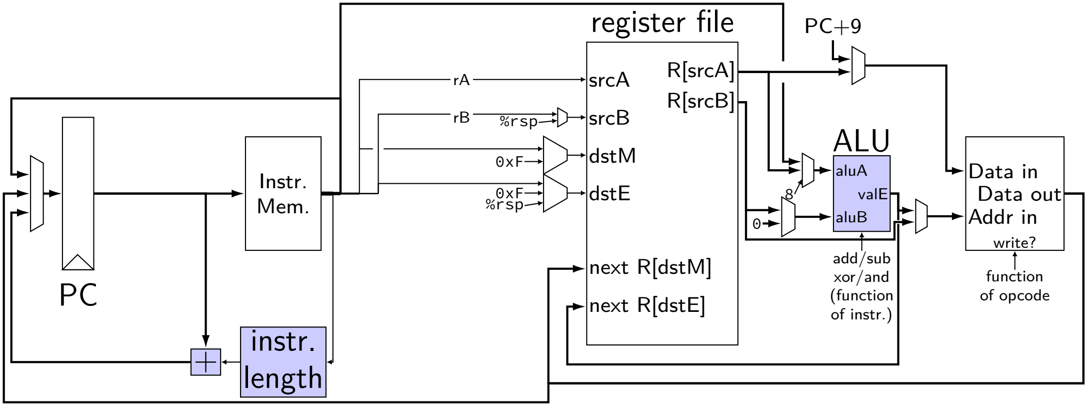
Exercise: what do they select for `cmovle`?

circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for `ret`?

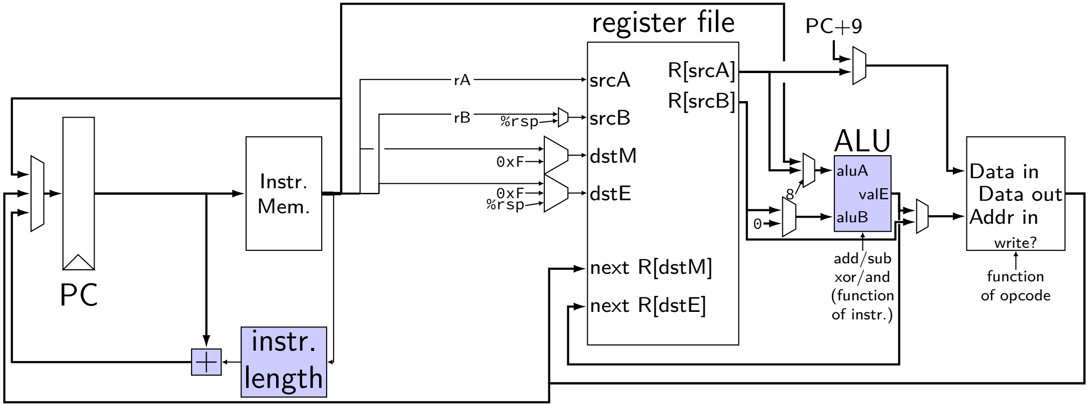
circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...

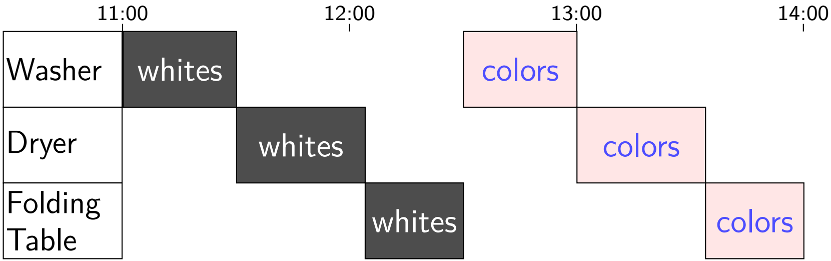
Exercise: what do they select for **popq**?

circuit: setting MUXes



MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `call`?

Human pipeline: laundry



11:00

12:00

13:00

14:00

Washer

Dryer

Folding Table

whites

whites

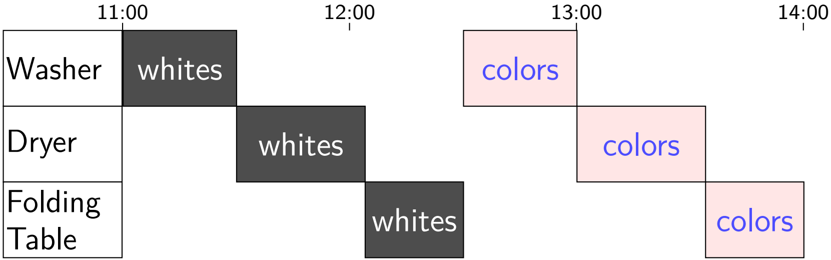
whites

colors

colors

colors

Human pipeline: laundry

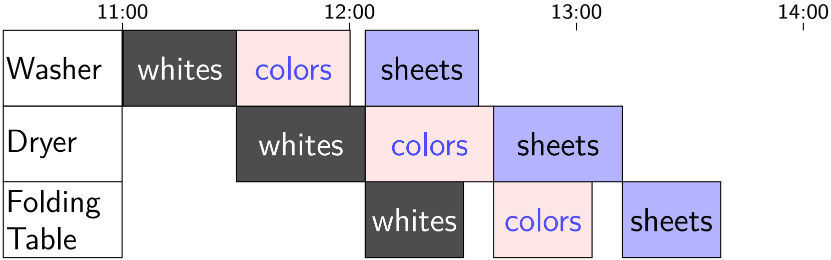


11:00 12:00 13:00 14:00

Washer whites colors

Dryer whites colors

Folding Table whites colors



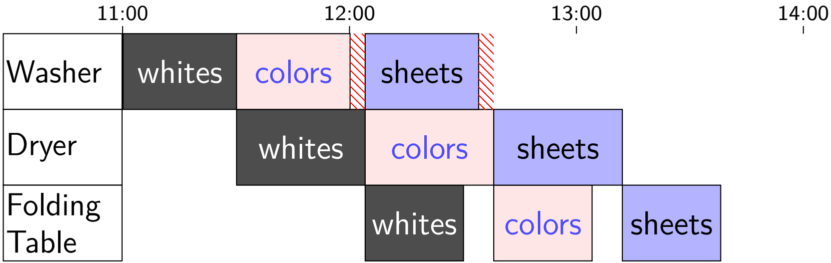
11:00 12:00 13:00 14:00

Washer whites colors sheets

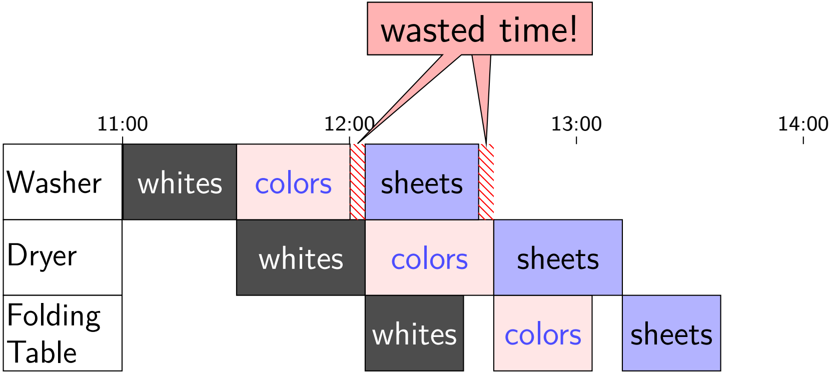
Dryer whites colors sheets

Folding Table whites colors sheets

Waste (1)


11:00
12:00
13:00
14:00
Washer
whites
colors
sheets
Dryer
whites
colors
sheets
Folding Table
whites
colors
sheets

Waste (1)



wasted time!

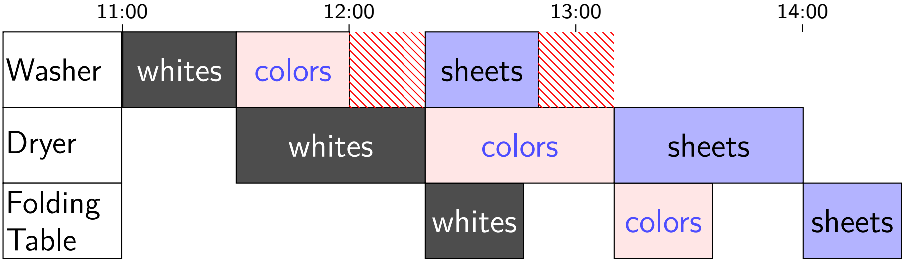
11:00 12:00 13:00 14:00

Washer: whites, colors, sheets

Dryer: whites, colors, sheets

Folding Table: whites, colors, sheets

Waste (2)



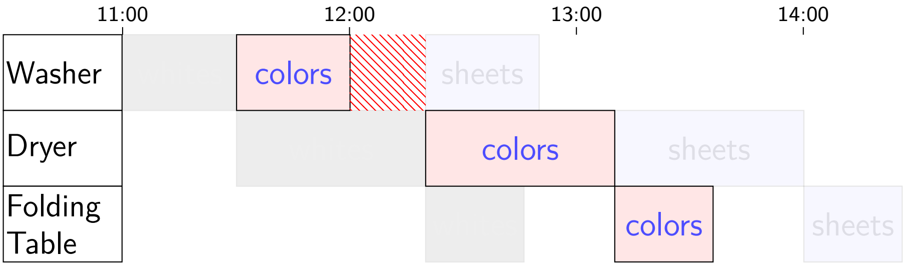
11:00 12:00 13:00 14:00

Washer whites colors sheets

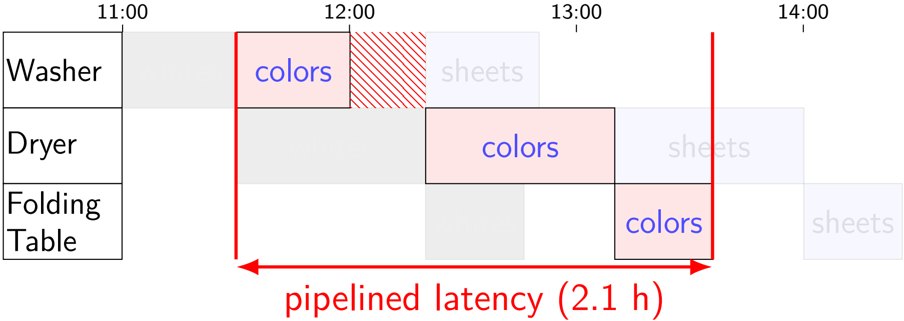
Dryer whites colors sheets

Folding Table whites colors sheets

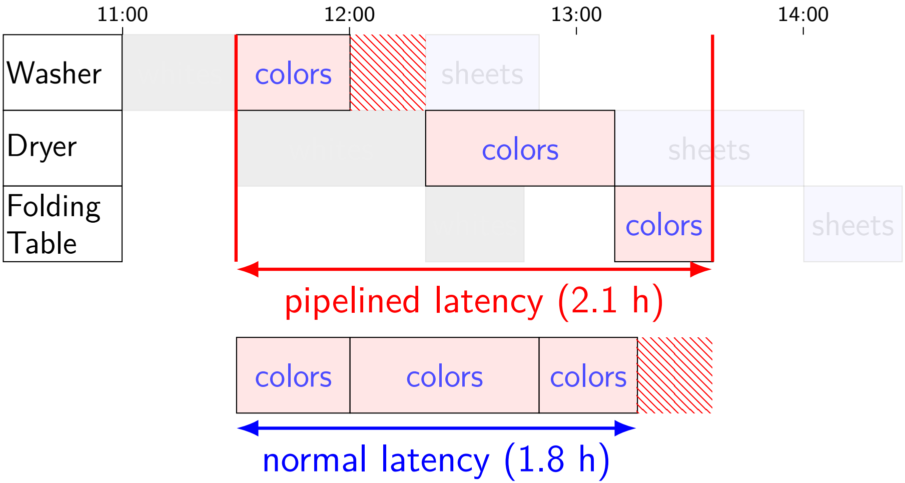
Latency — Time for One



Latency — Time for One



Latency — Time for One



11:00 12:00 13:00 14:00

Washer: whites, colors, sheets

Dryer: whites, colors, sheets

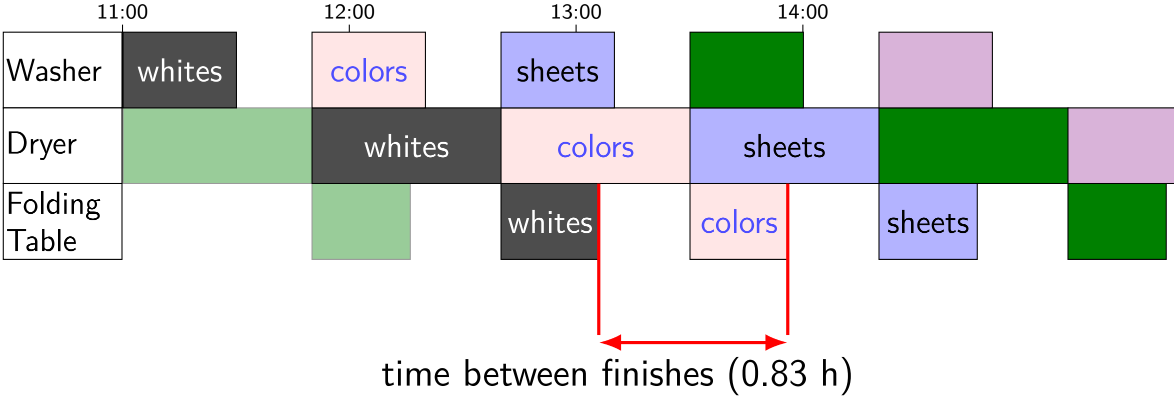
Folding Table: whites, colors, sheets

pipelined latency (2.1 h)

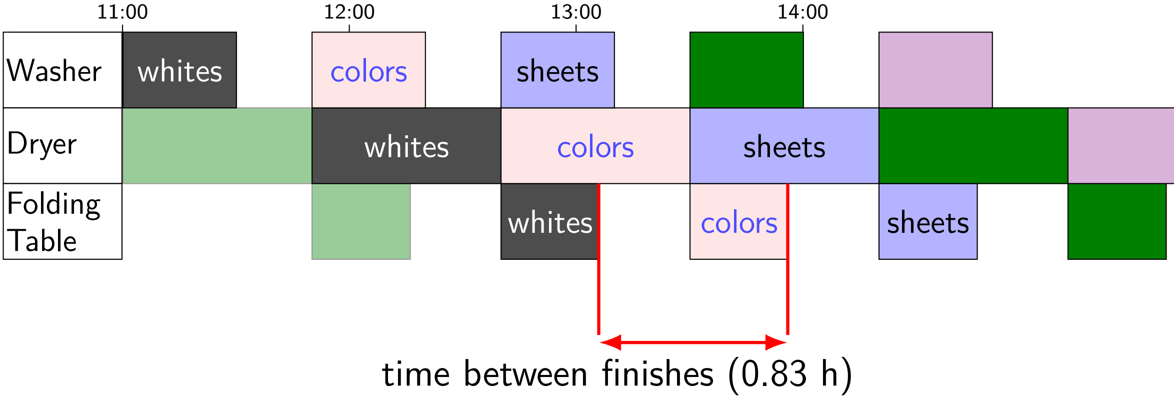
colors colors colors

normal latency (1.8 h)

Throughput — Rate of Many

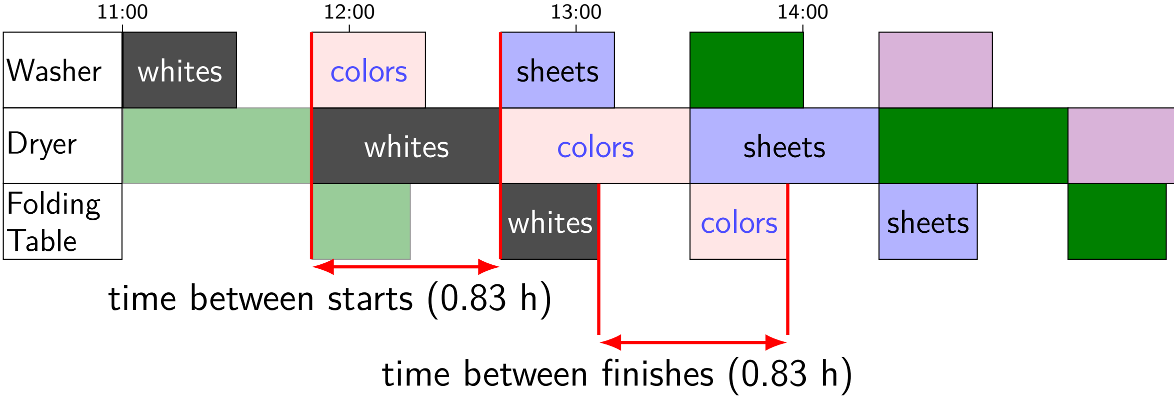


Throughput — Rate of Many



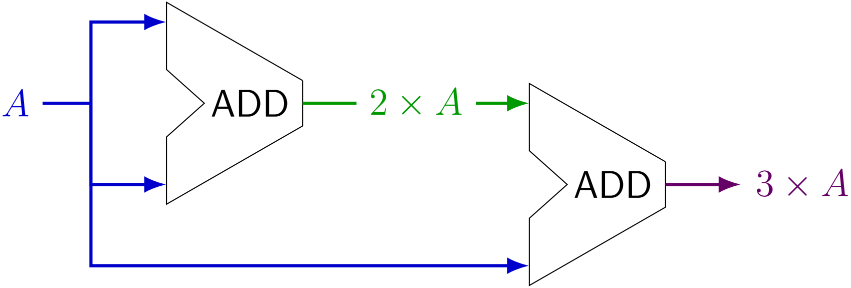
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many

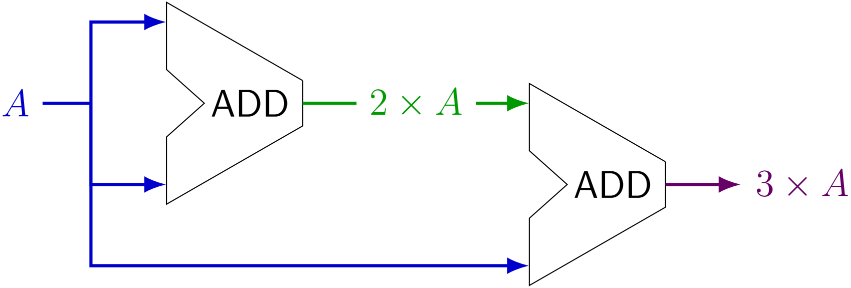


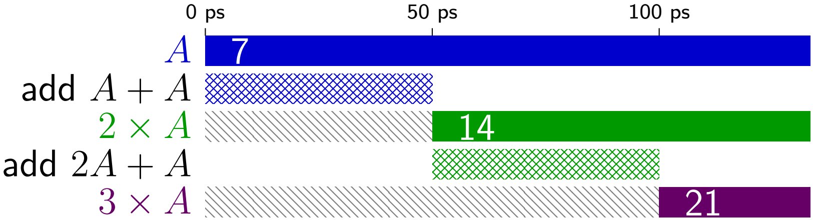
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

times three circuit

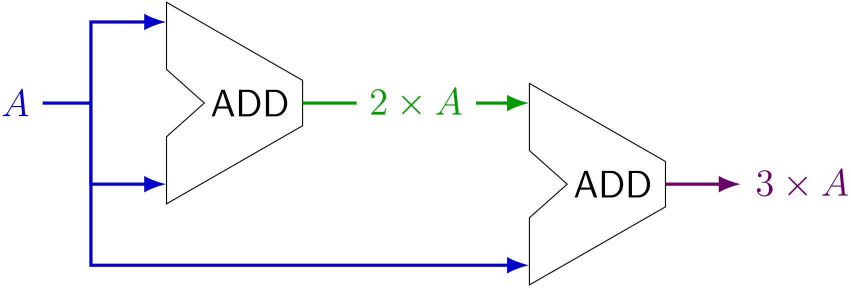


times three circuit

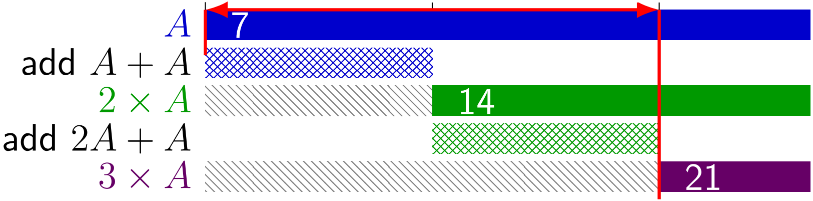




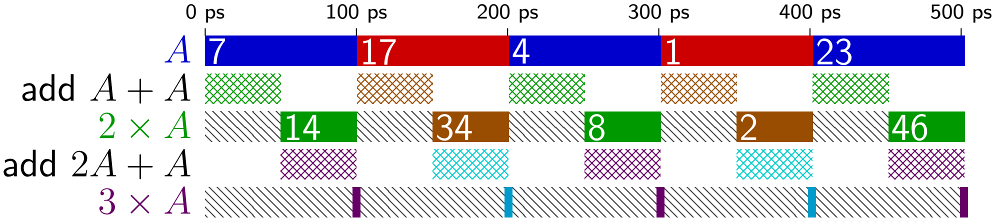
times three circuit



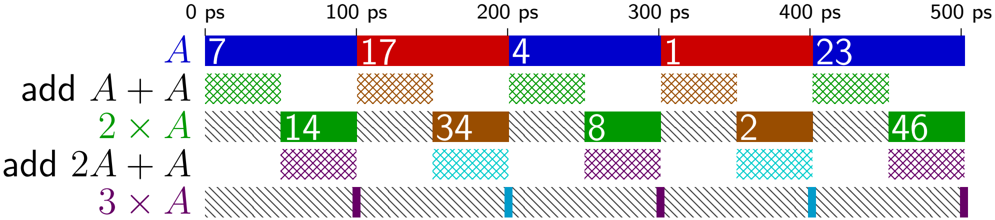
100 ps latency $\implies$ 10 results/ns throughput

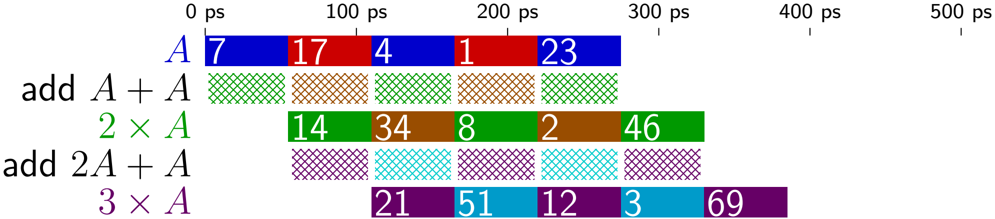


times three and repeat

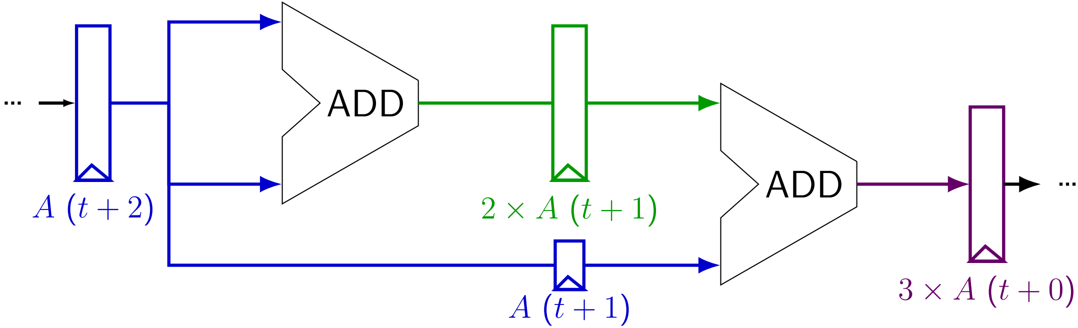


times three and repeat

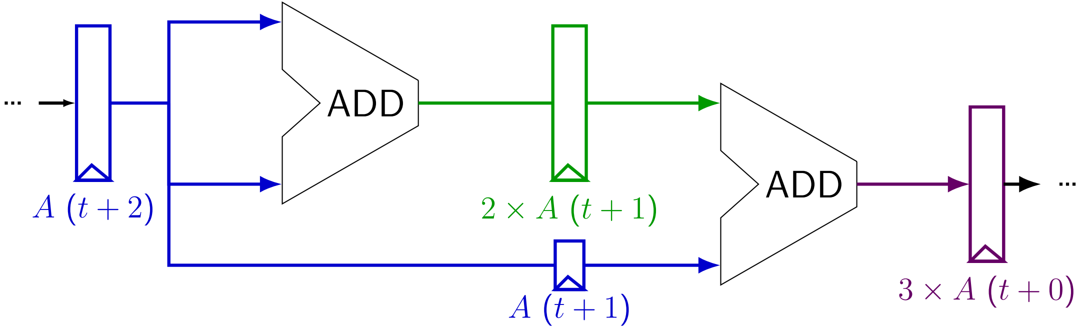




pipelined times three

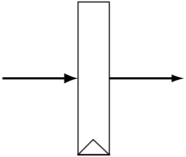


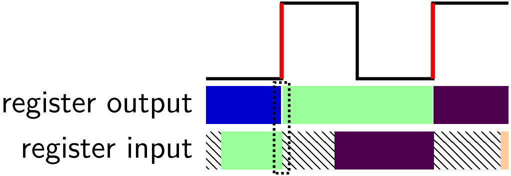
pipelined times three



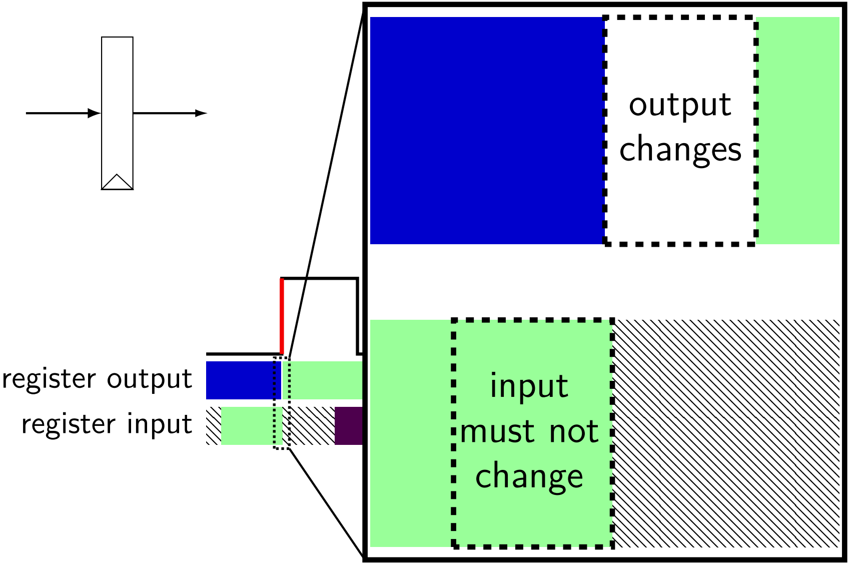
$A\ (t+2)$	7	17	
$A\ (t+1)$		7	17
$2 \times A\ (t+1)$		14	34
$3 \times A\ (t+0)$			21

register tolerances

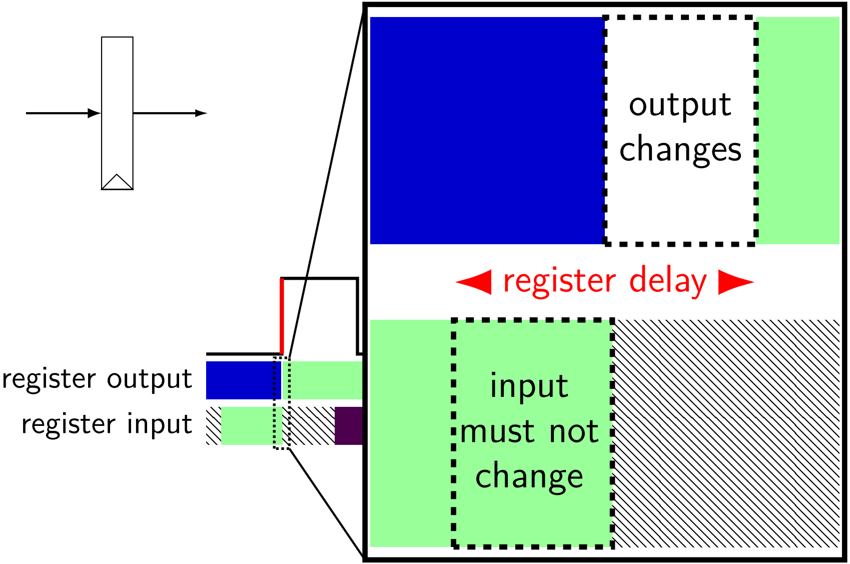




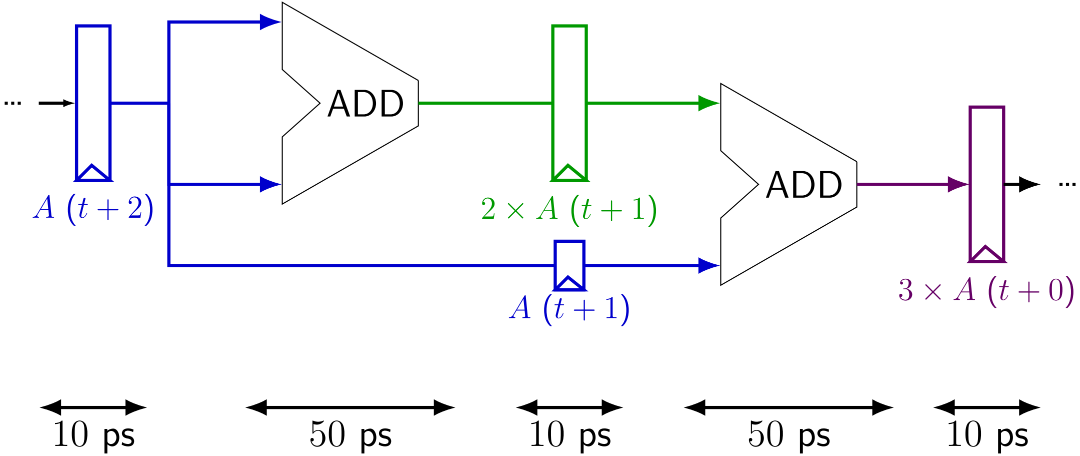
register tolerances



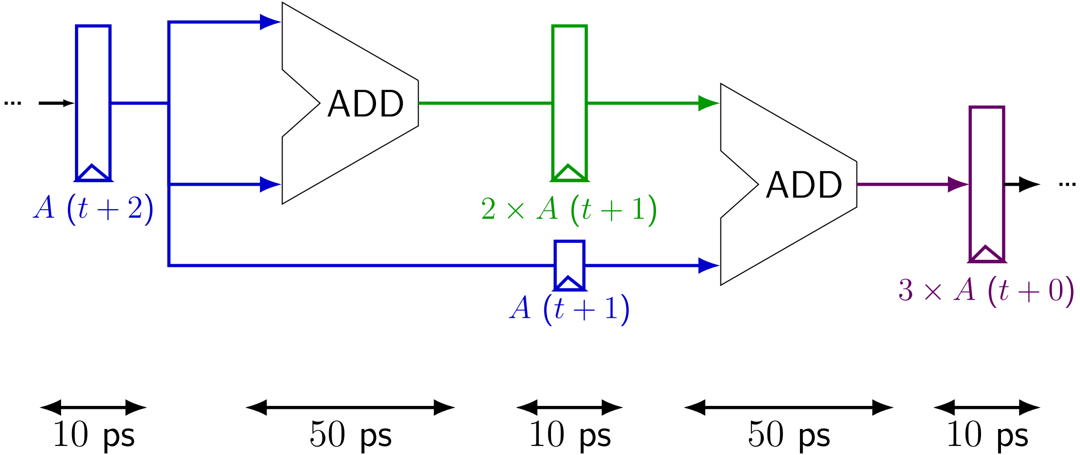
register tolerances



times three pipeline timing



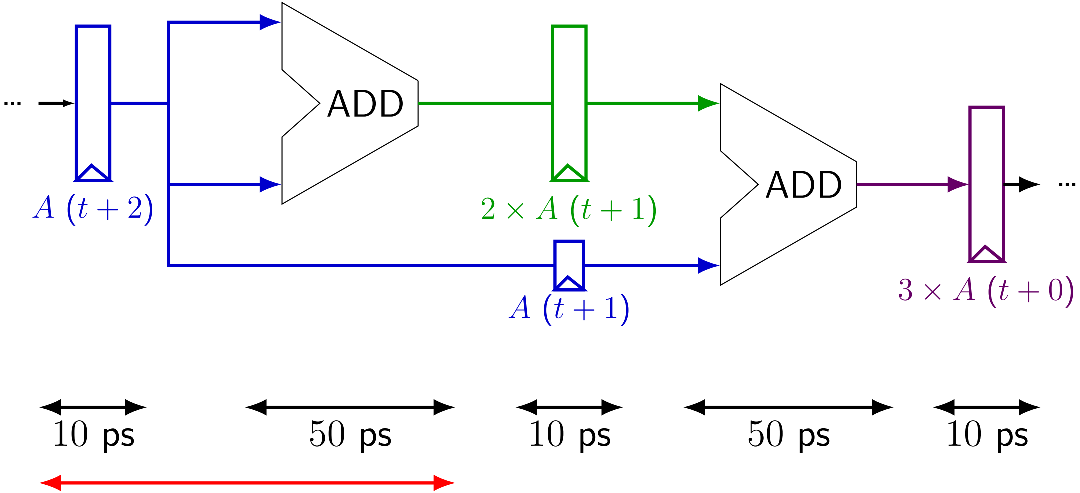
times three pipeline timing



exercise: minimum clock cycle time:

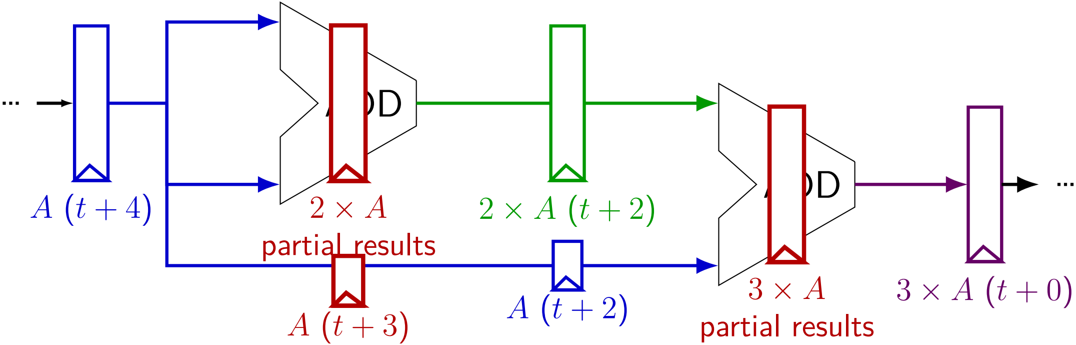
A. 50 ps B. 60 ps C. 65 ps D. 70 ps E. 130 ps

times three pipeline timing

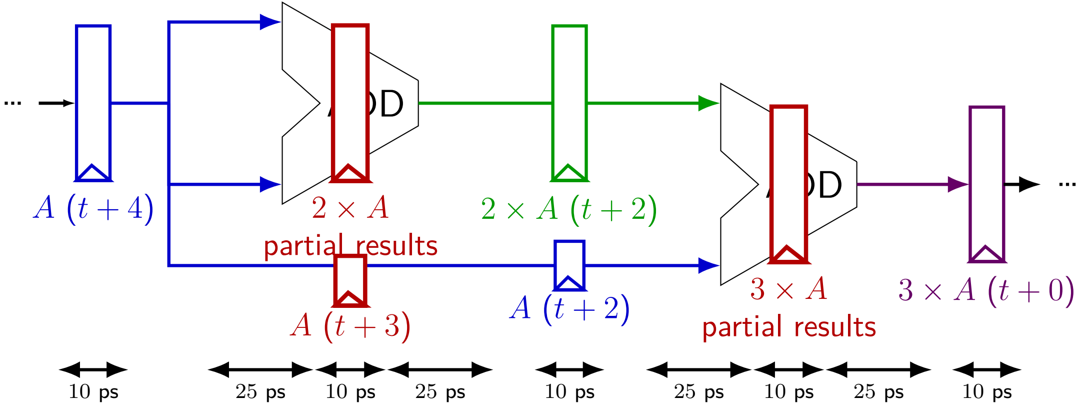


throughput: $\dfrac{1}{60\text{ ps}} \approx 16$ G operations/sec

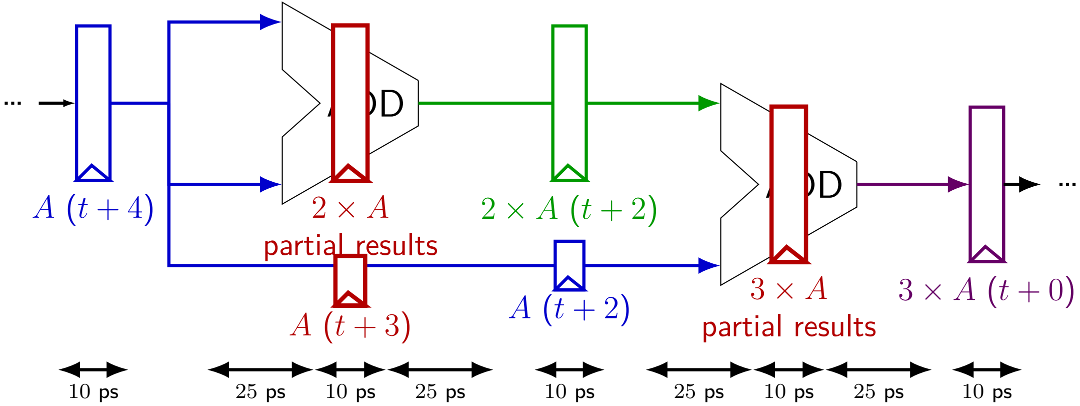
deeper pipeline



deeper pipeline



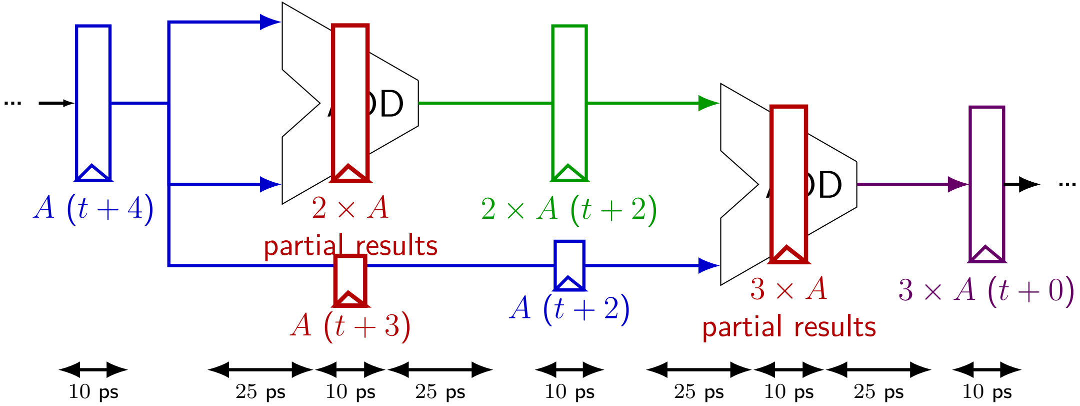
deeper pipeline



Problem: How much faster can we get?

Problem: Can we even do this?

deeper pipeline

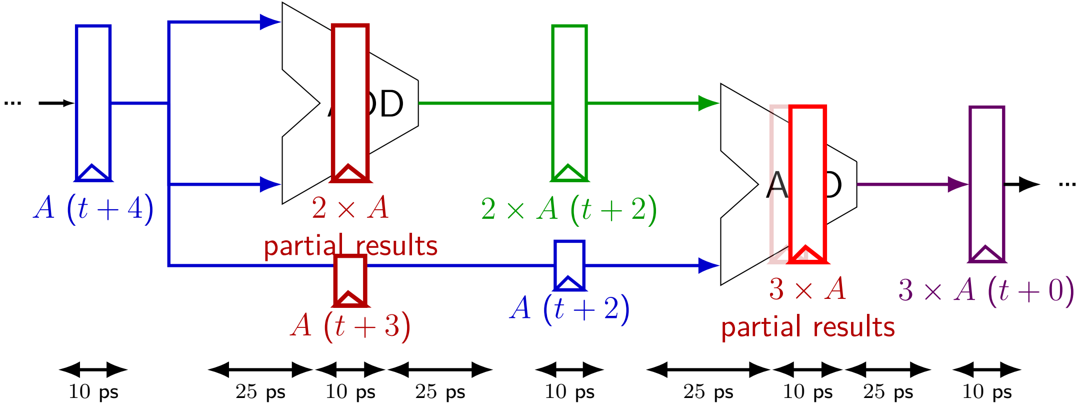


exercise: throughput now?

A. 1/(25 ps) B. 1/(30 ps)

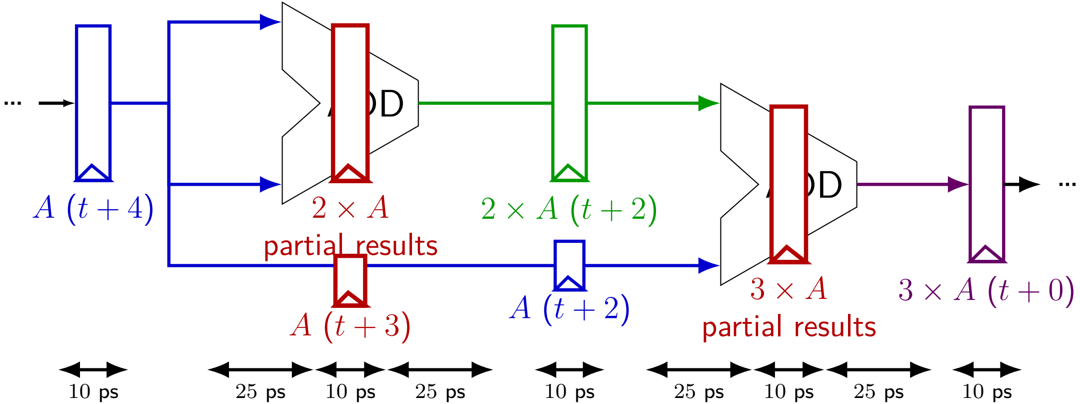
C. 1/(35 ps) D. something else

deeper pipeline



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

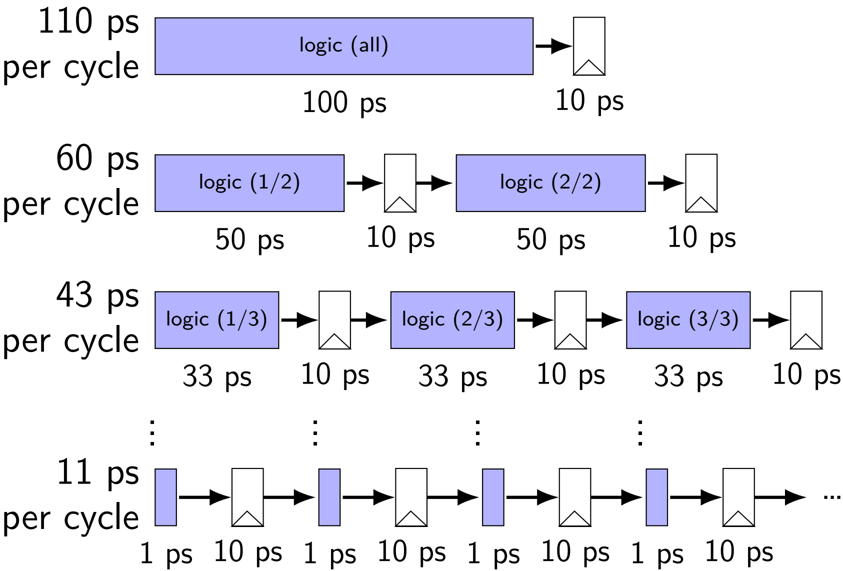
deeper pipeline



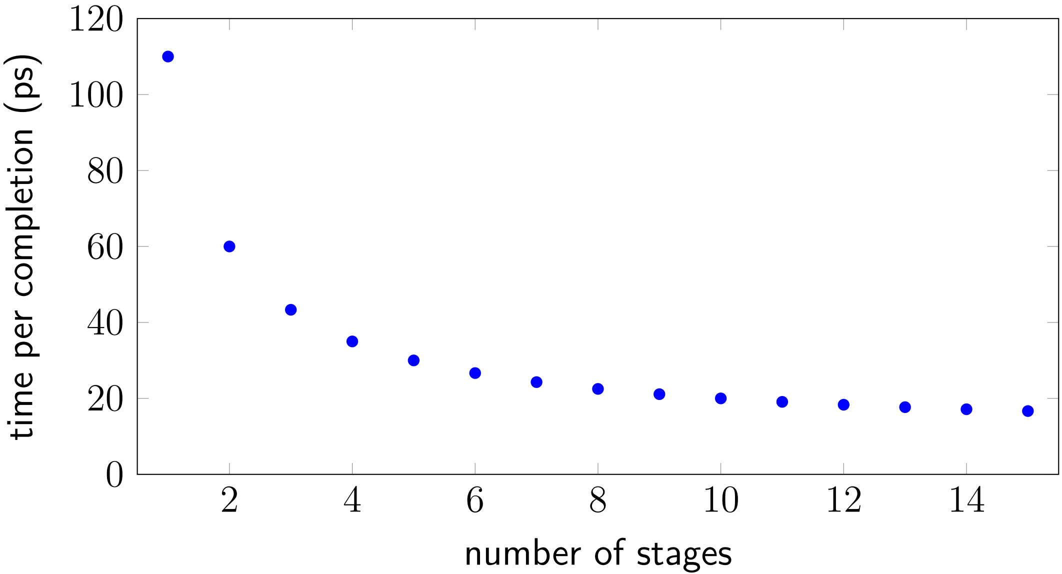
Problem: How much faster can we get?

Problem: Can we even do this?

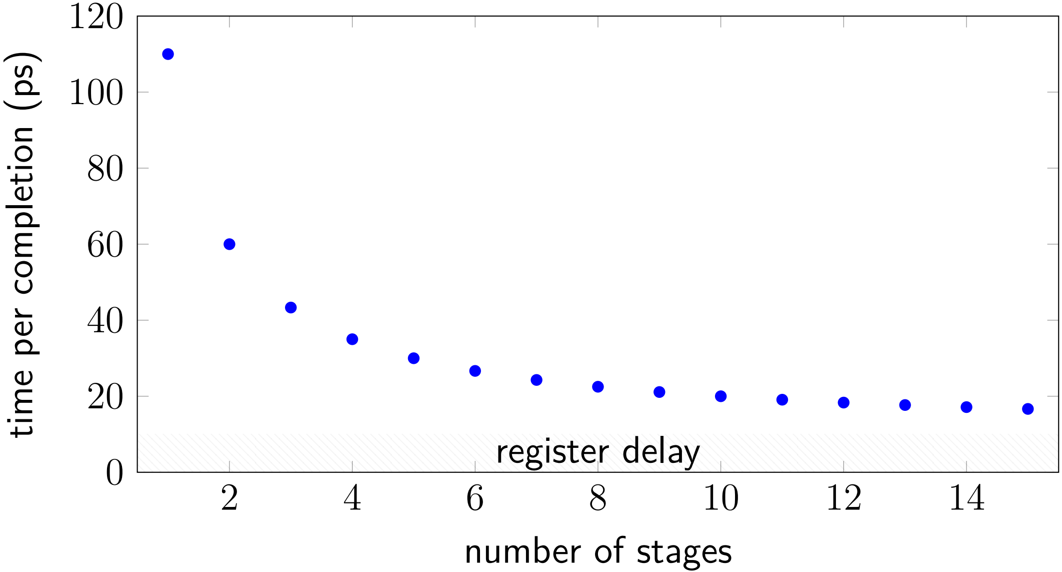
diminishing returns: register delays



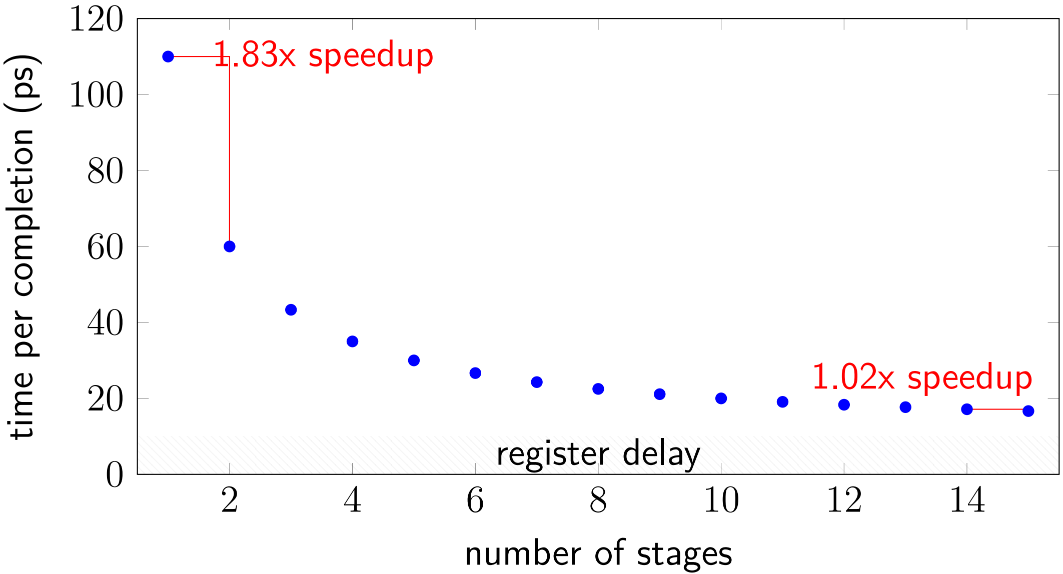
diminishing returns: register delays



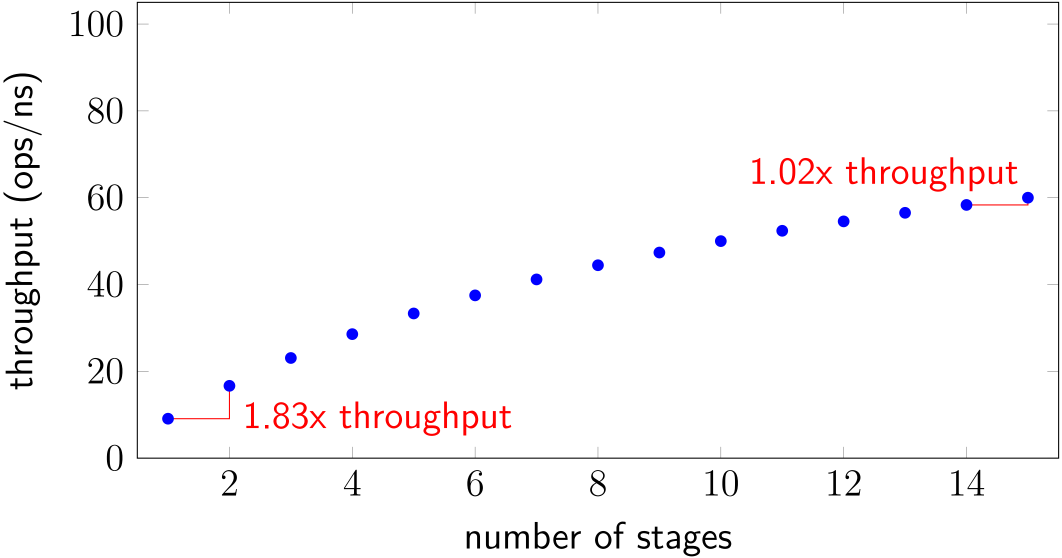
diminishing returns: register delays



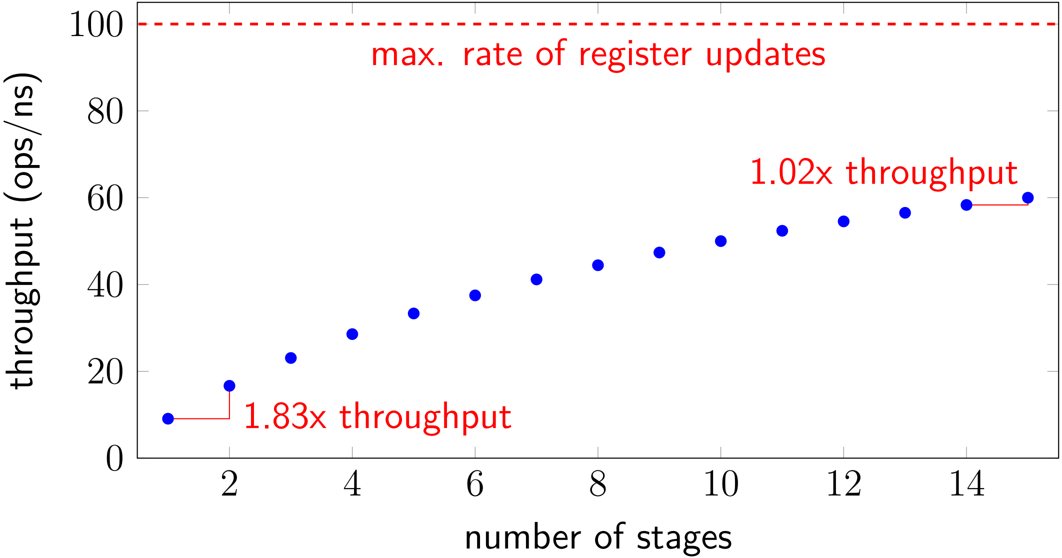
diminishing returns: register delays



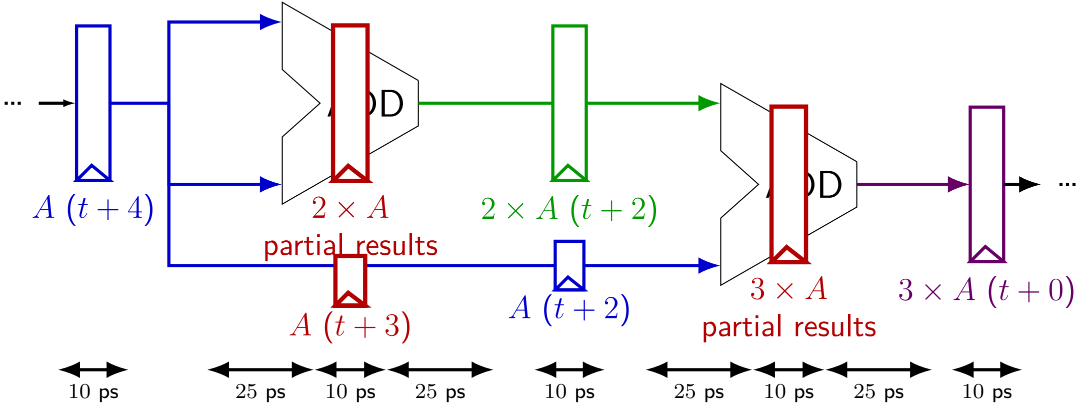
diminishing returns: register delays



diminishing returns: register delays



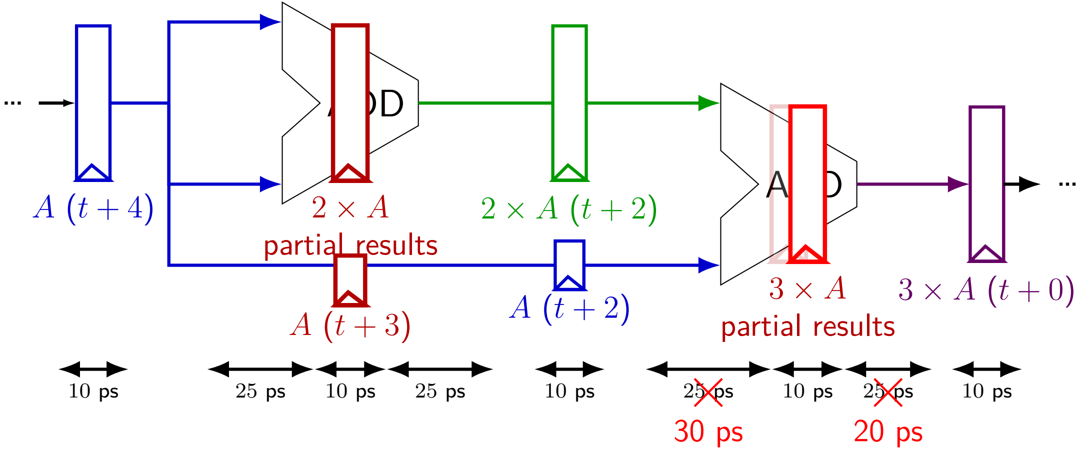
deeper pipeline



Problem: How much faster can we get?

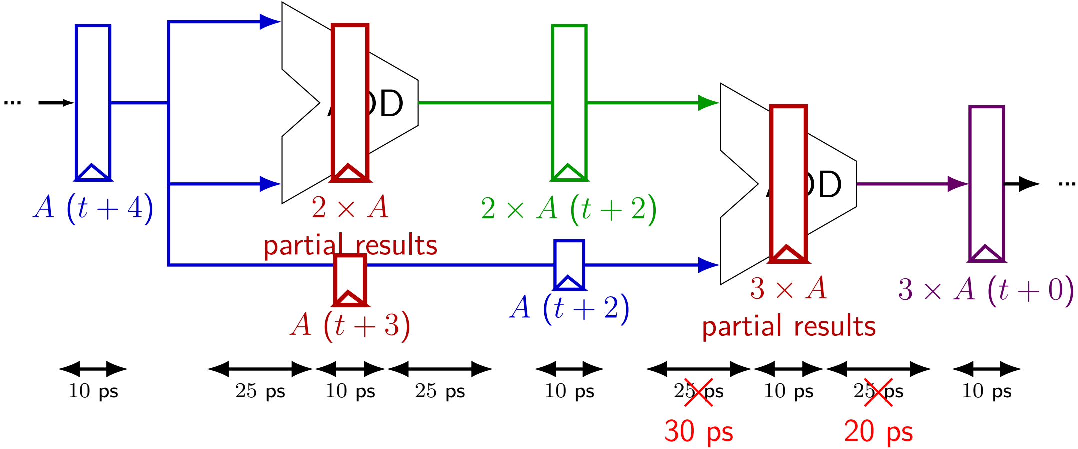
Problem: Can we even do this?

deeper pipeline



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

deeper pipeline

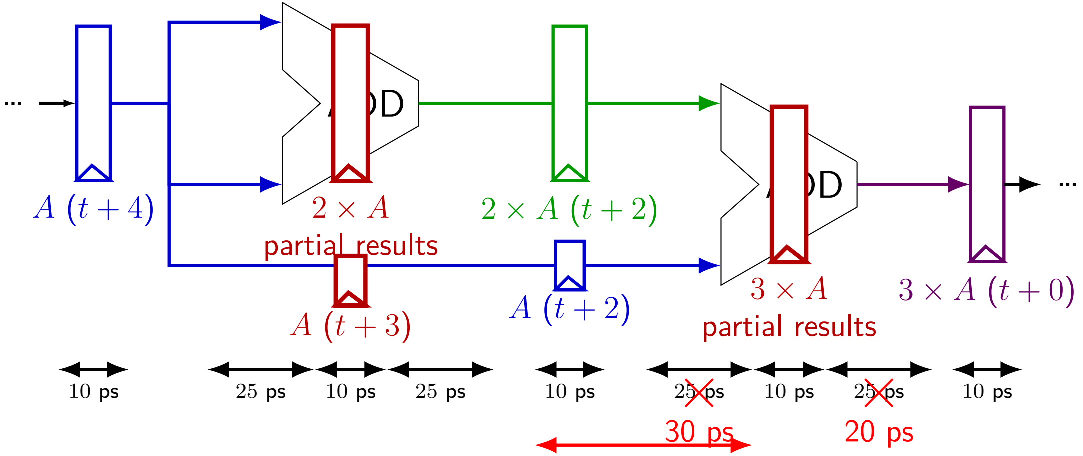


exercise: throughput now? (didn't split second add evenly)

A. 1/(25 ps) B. 1/(30 ps)

C. 1/(35 ps) D. 1/(40 ps) E. something else

deeper pipeline

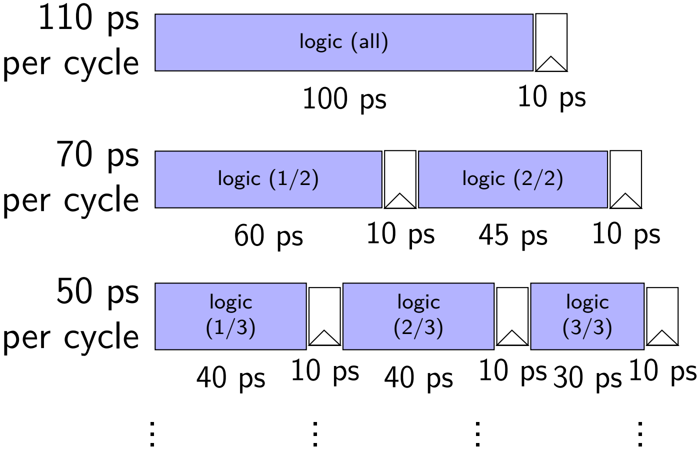


throughput: $\frac{1}{40\text{ ps}} \approx 25$ G ops/sec

diminishing returns: uneven split

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

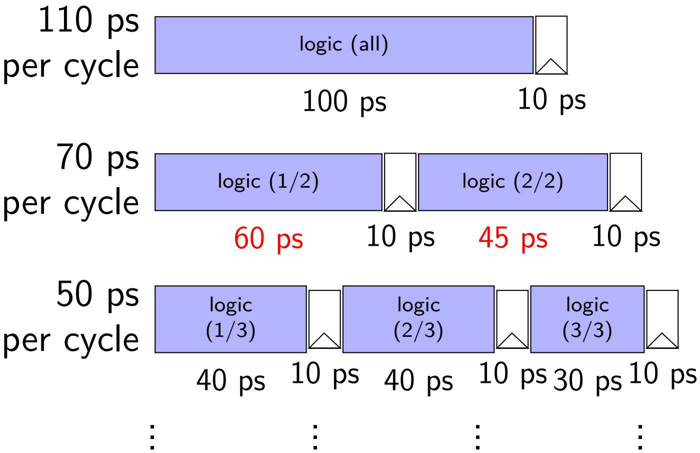
Probably not...



diminishing returns: uneven split

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

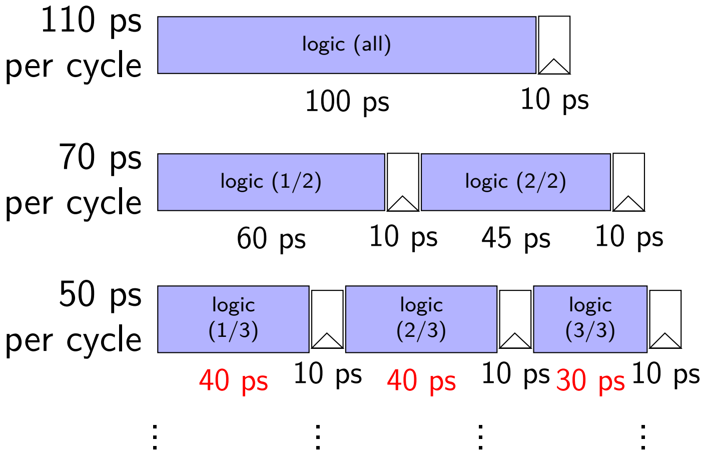
Probably not...



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

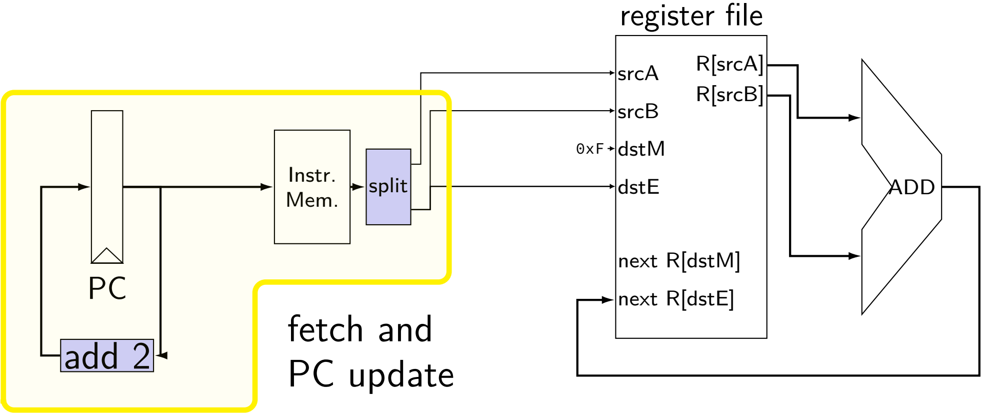
Execute: arithmetic (ALU)

Memory: read/write data memory

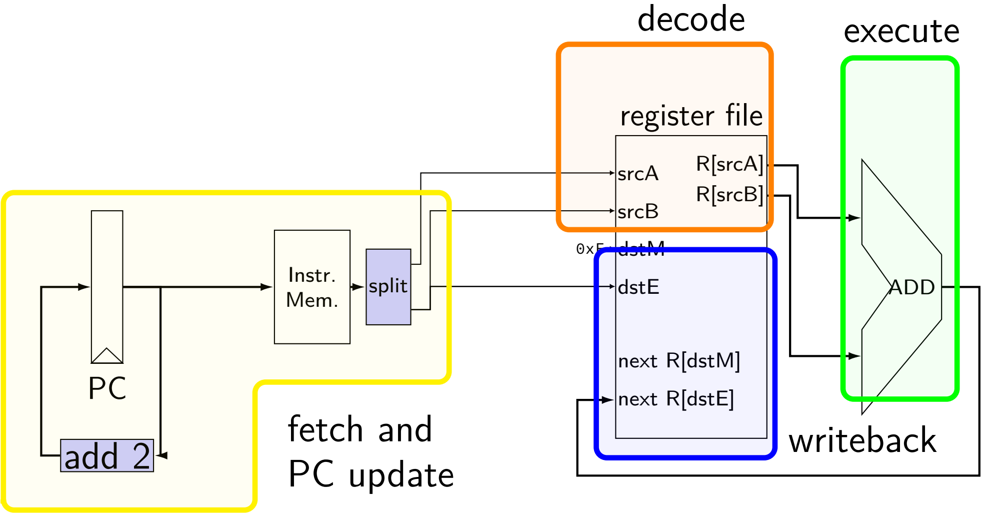
Writeback: write register file

5 stages
one instruction in each
compute next to start immediatelly

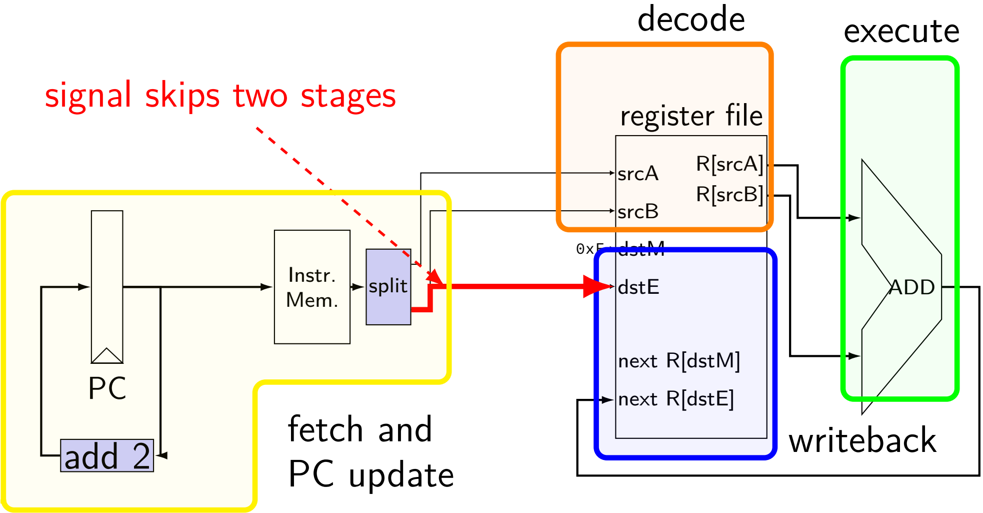
addq CPU


register file
srcA
srcB
0xF
dstM
dstE
next R[dstM]
next R[dstE]
R[srcA]
R[srcB]
ADD
PC
Instr. Mem.
split
add 2
fetch and PC update

addq CPU

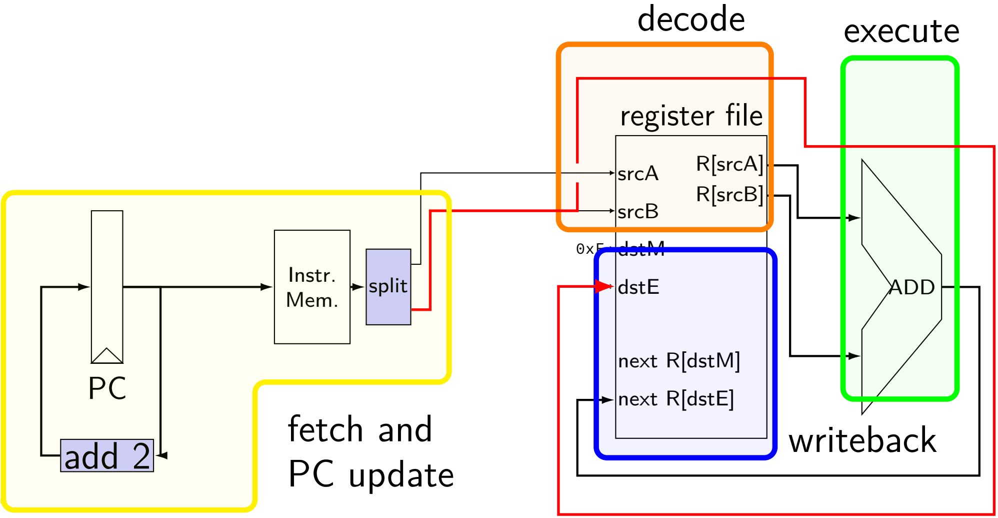

decode
execute
register file
srcA
srcB
0xF
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ADD
Instr.
Mem.
split
PC
add 2
fetch and
PC update
writeback

addq CPU

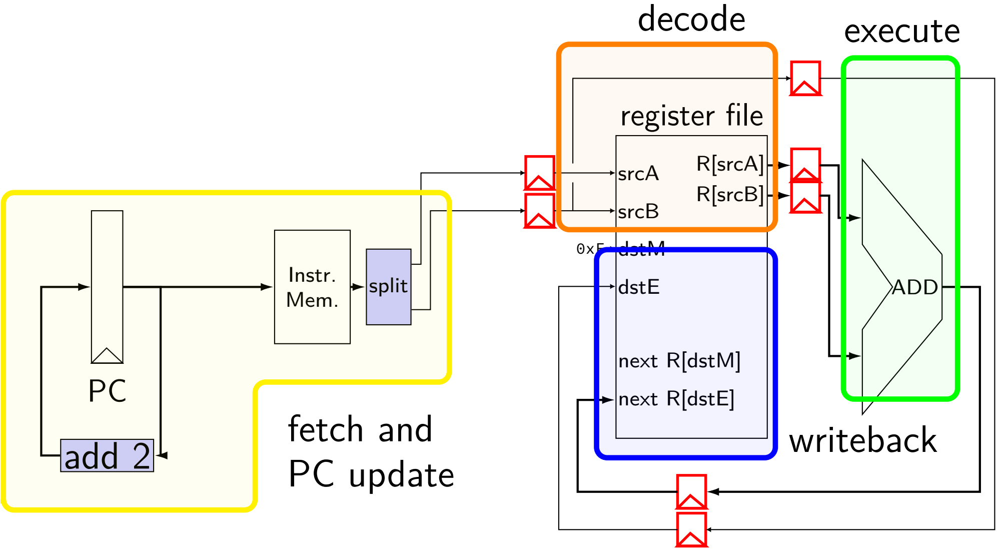


signal skips two stages

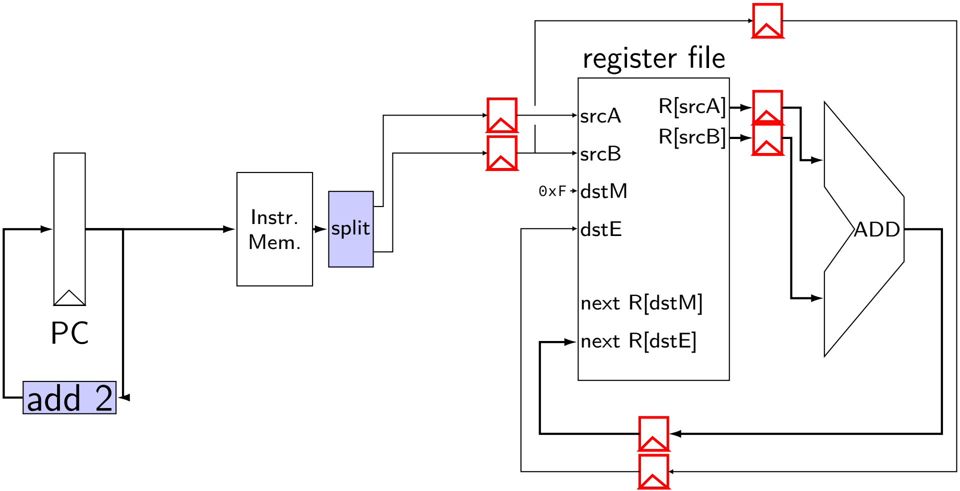
addq CPU


decode
execute
register file
srcA
srcB
R[srcA]
R[srcB]
0xF
dstM
dstE
next R[dstM]
next R[dstE]
ADD
writeback
Instr.
Mem.
split
PC
add 2
fetch and
PC update

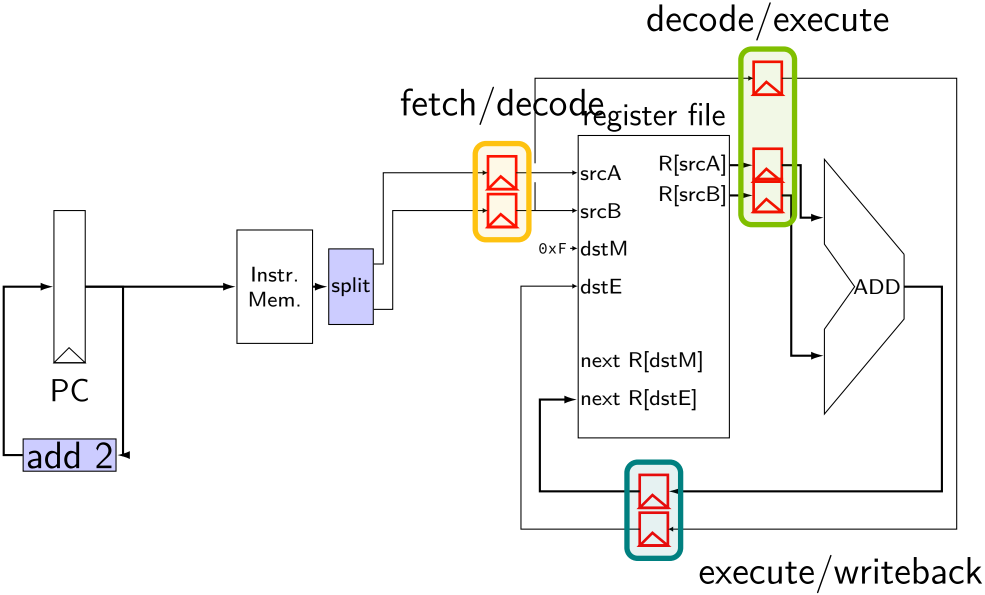
pipelined addq processor



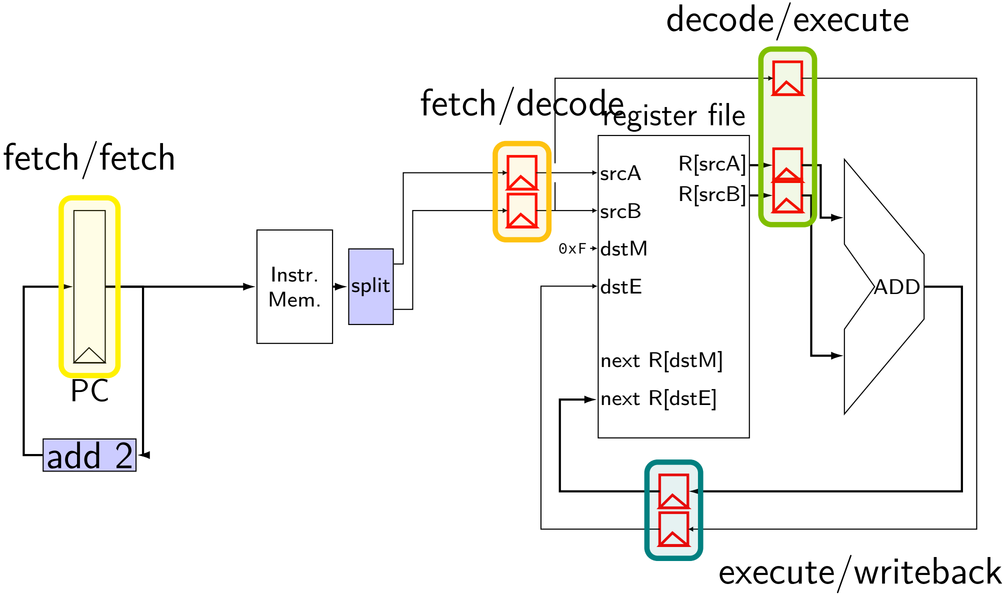
pipelined addq processor



pipelined addq processor

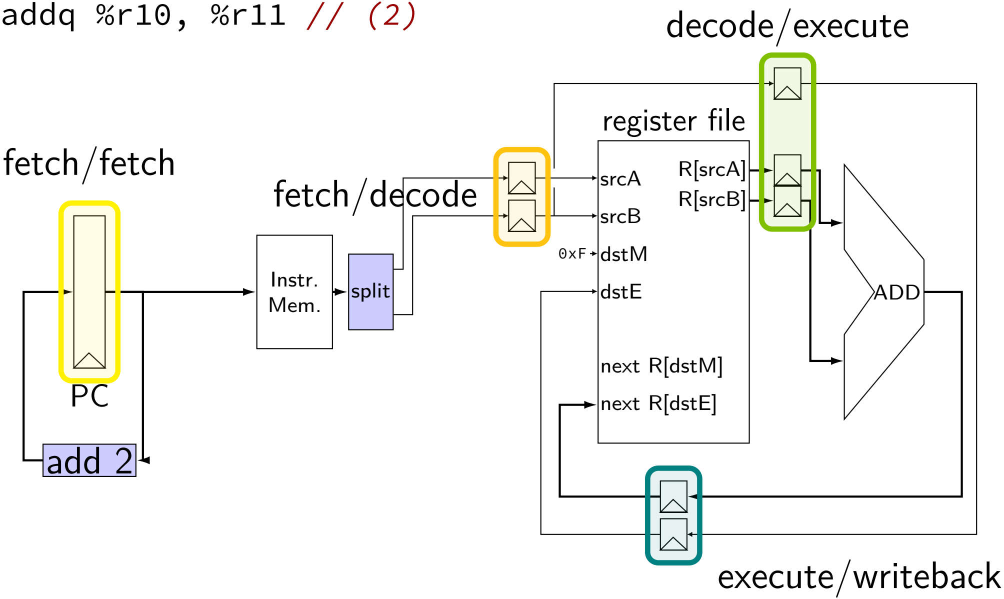


pipelined addq processor



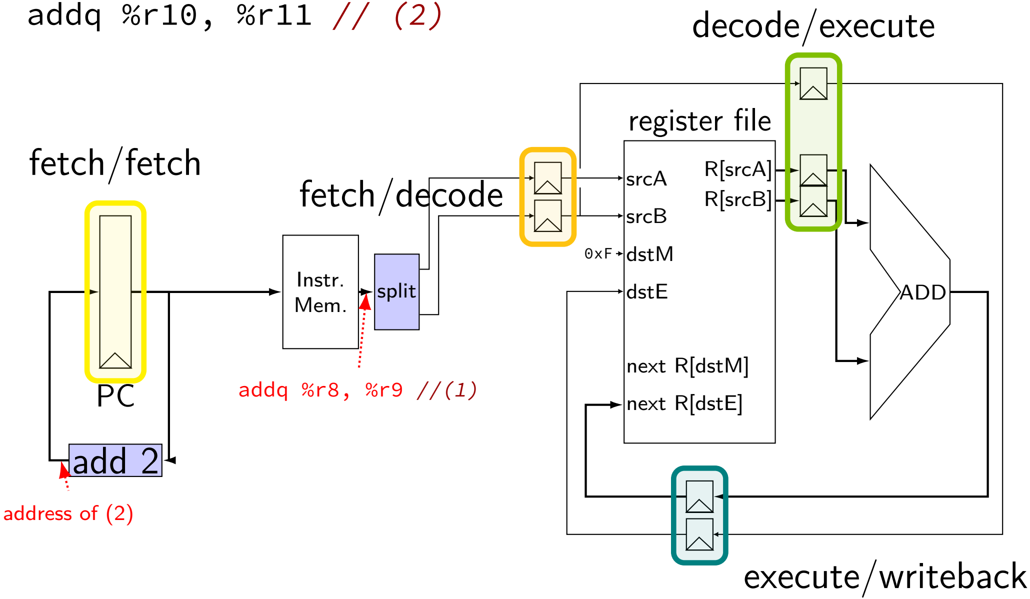
addq execution

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



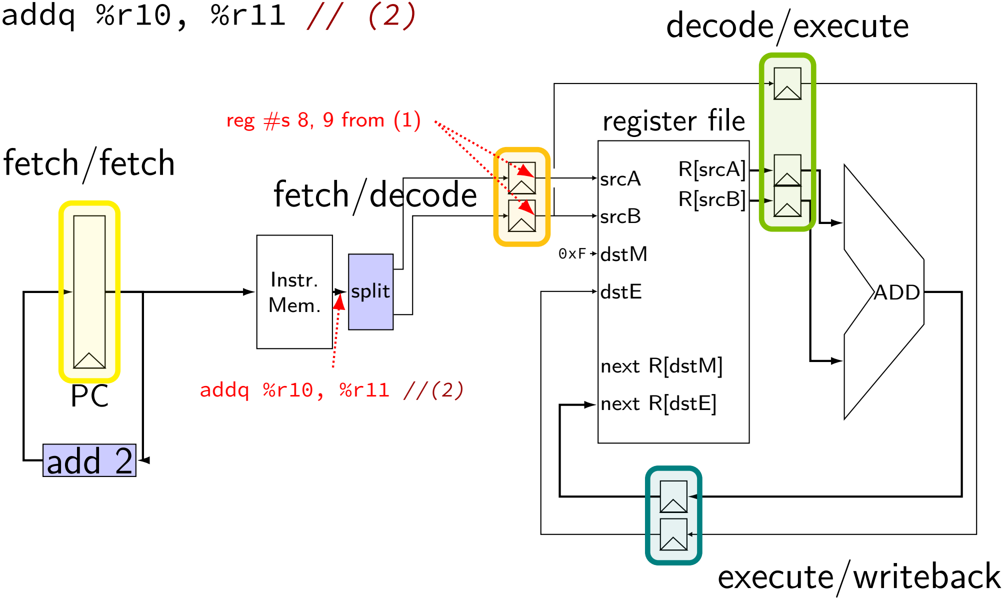
addq execution

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



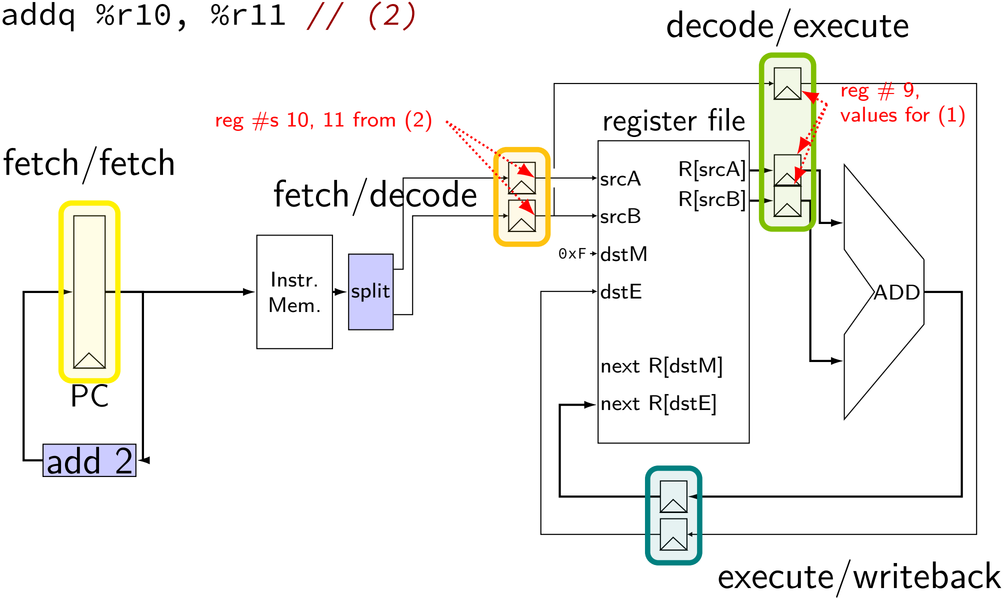
addq execution

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



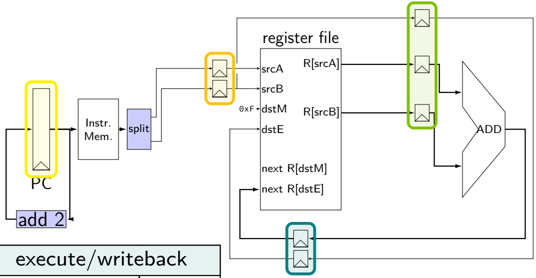
addq execution

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



addq processor timing

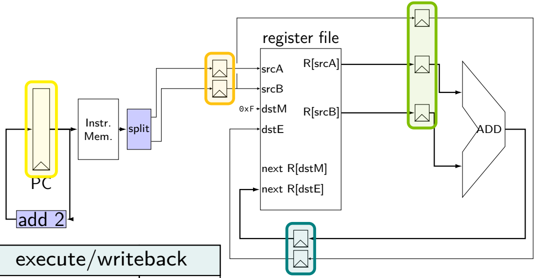
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

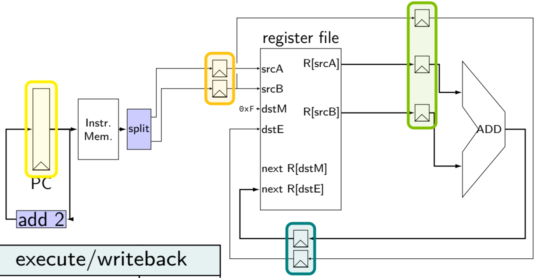
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

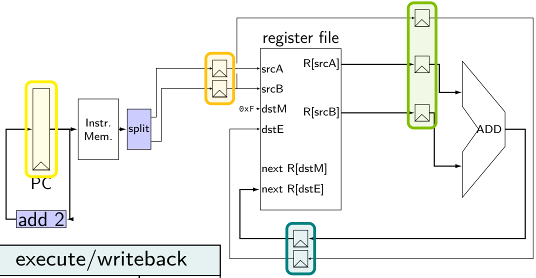
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

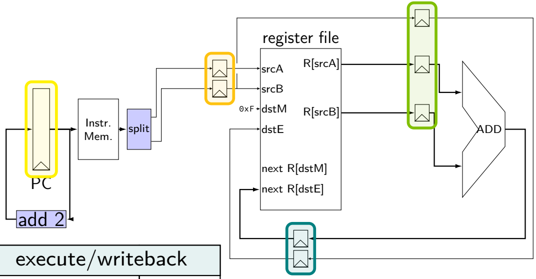
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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
```



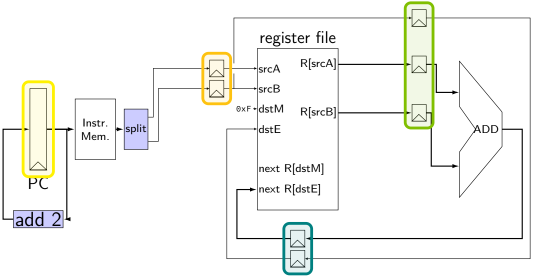
	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

backup slides

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)