Pipelining

last time

HCLRS components
- data memory
- register file

more choices
- textbook adds 0 for irmovq, you don't have to
- textbook uses 'M' port for memory results, you don't hvae to

building CPUs approach 1: working stage-by-stage
- systematically: what values for what instructions
- figure out what MUXes you need
- example: RSP/rA/rB for decode (register read)

building CPUs approach 2: figuring out MUXes
- textbook has *a solution* for what MUXes to use
- what lets us read/write the correct values?

SEQ: instruction fetch

read instruction memory at PC

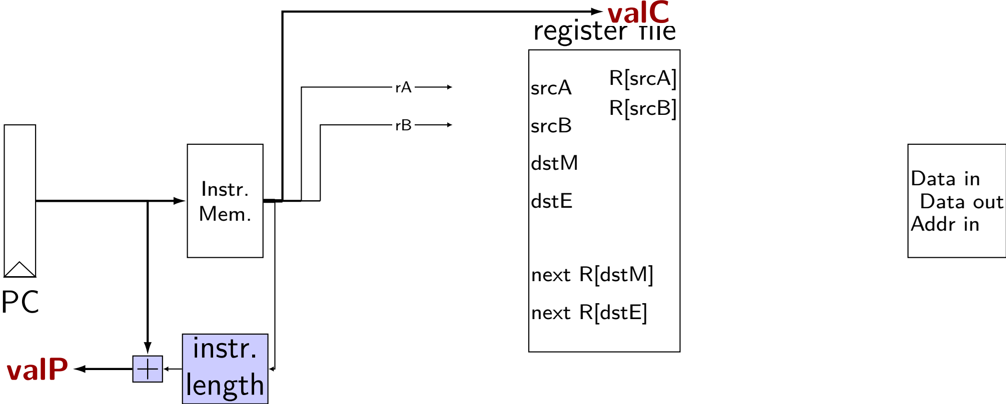
split into seperate wires:

- icode:ifun — opcode
- rA, rB — register numbers
- valC — call target or mov displacement

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

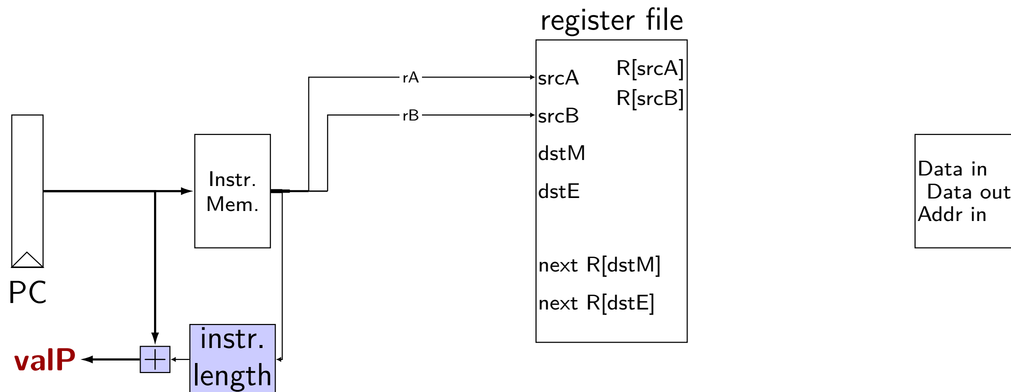


SEQ: instruction "decode"

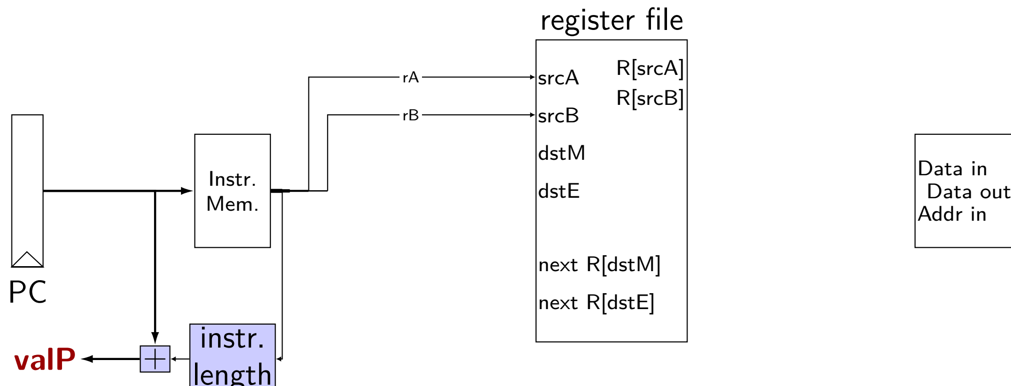
read registers

- valA, valB — register values

instruction decode (1)



instruction decode (1)



exercise: which of these instructions can this **not** work for?

`nop`, `addq`, `mrmovq`, `popq`, `call`,

SEQ: srcA, srcB

always read rA, rB?

Problems:

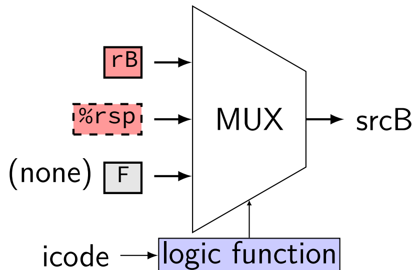
- push rA
- pop
- call
- ret

extra signals: srcA, srcB — computed input register

MUX controlled by icode

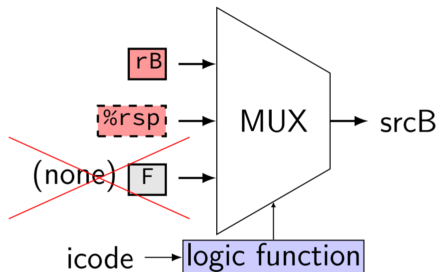
SEQ: possible registers to read

instruction	srcA	srcB
`halt`, `nop`, `j`*CC*, `irmovq`	none	none
`cmovCC`, `rrmovq`	rA	none
`mrmovq`	none	rB
`rmmovq`, *OP*`q`	rA	rB
`call`, `ret`	none?	`%rsp`
`pushq`, `popq`	rA	`%rsp`

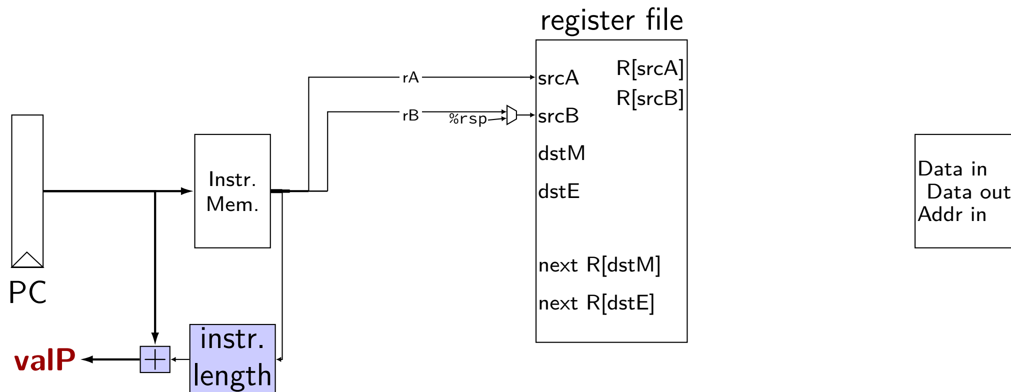


SEQ: possible registers to read

instruction	srcA	srcB
`halt`, `nop`, `j`*CC*, `irmovq`	none	none
`cmovCC`, `rrmovq`	rA	none
`mrmovq`	none	rB
`rmmovq`, *OP*`q`	rA	rB
`call`, `ret`	none?	%rsp
`pushq`, `popq`	rA	%rsp



instruction decode (2)



SEQ: execute

perform ALU operation (add, sub, xor, and)

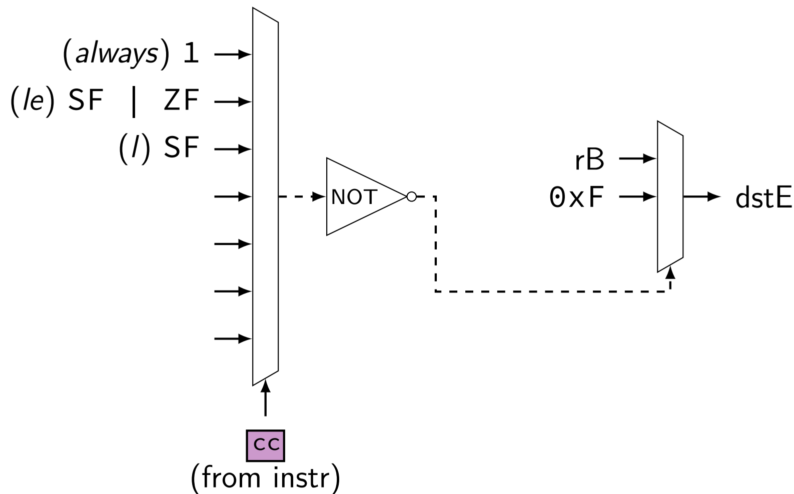
valE — ALU output

read prior condition codes

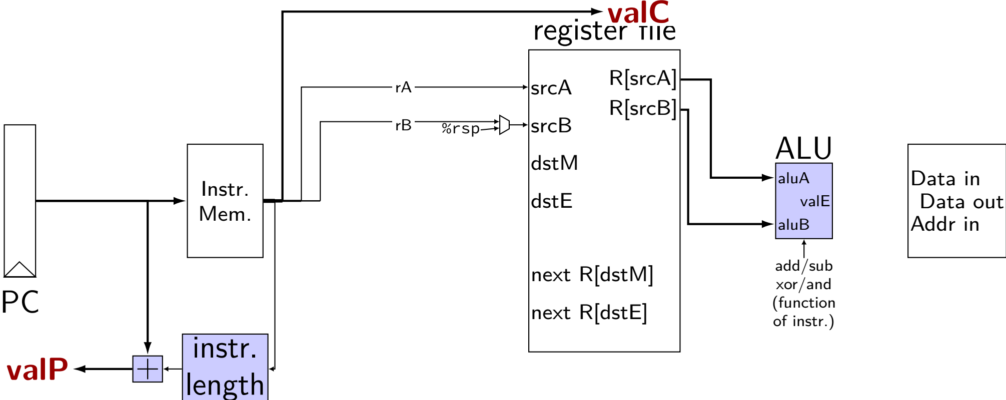
Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

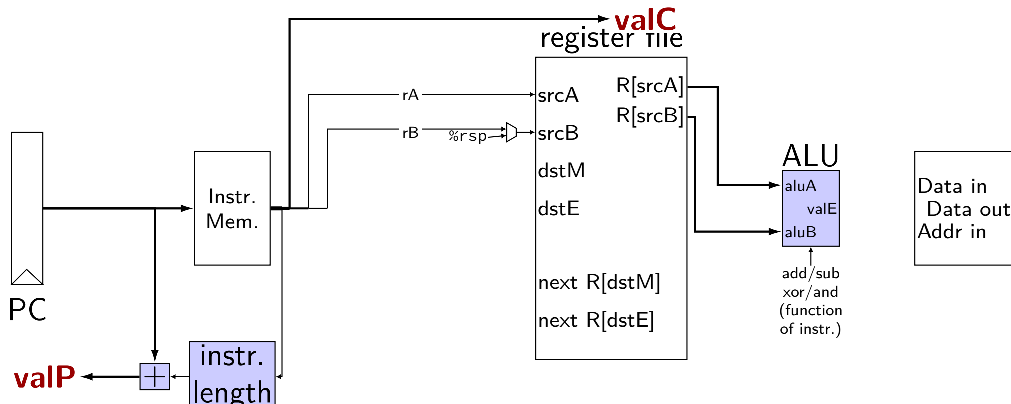
using condition codes: cmov



execute (1)



execute (1)



exercise: which of these instructions can this **not** work for?

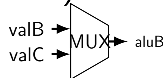
nop, addq, `mrmovq`, popq, `call`,

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

`mrmovq`
`rmmovq`



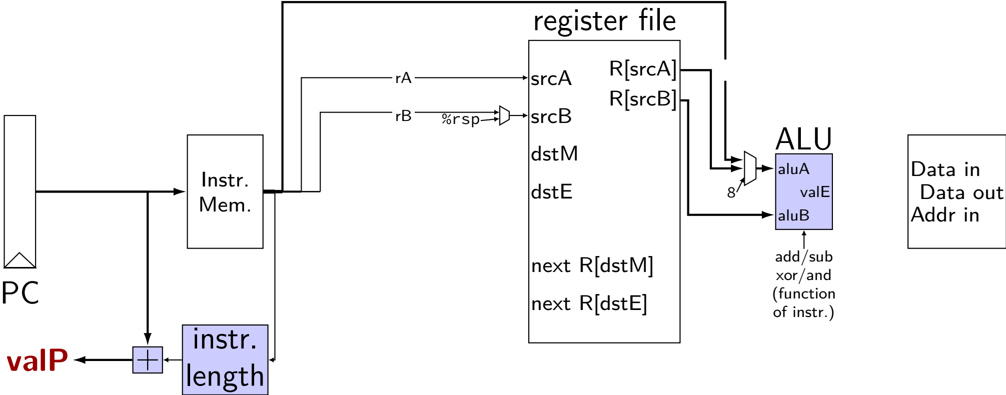
no, constants: (rsp +/- 8)

`pushq`
`popq`
`call`
`ret`

extra signals: aluA, aluB

computed ALU input values

execute (2)

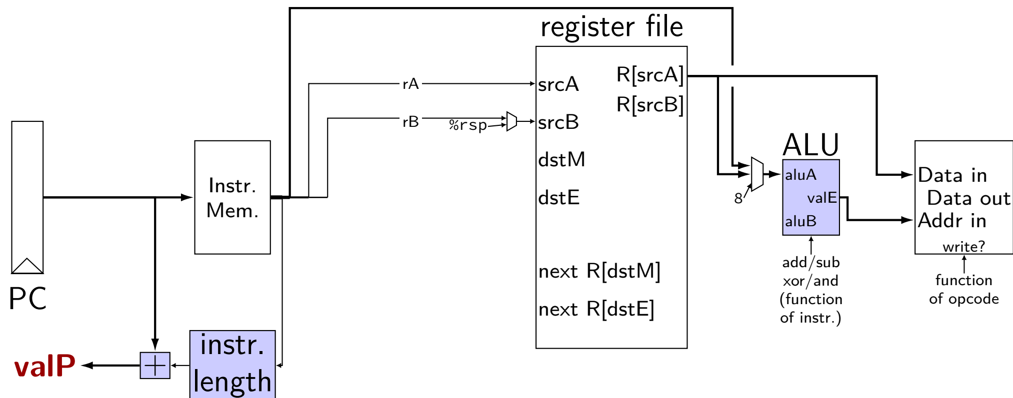


SEQ: Memory

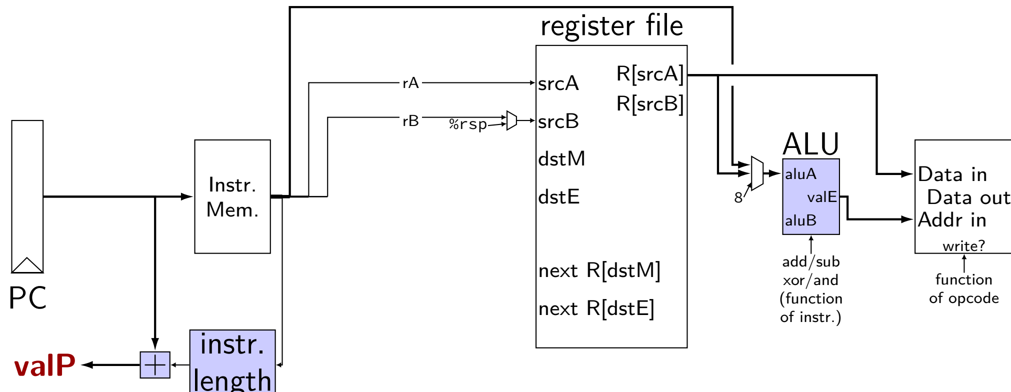
read or write data memory

- valM — value read from memory (if any)

memory (1)



memory (1)



exercise: which of these instructions can this **not** work for?

`nop`, `rmmovq`, `mrmovq`, `popq`, `call`,

SEQ: control signals for memory

read/write — read enable? write enable?

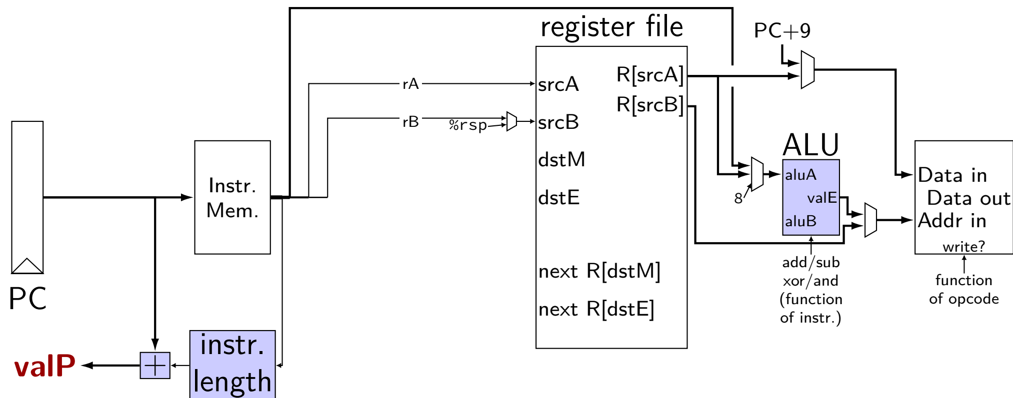
Addr — address

- mostly ALU output
- tricky cases: `popq`, `ret`

Data — value to write

- mostly valB
- tricky cases: `call`, `push`

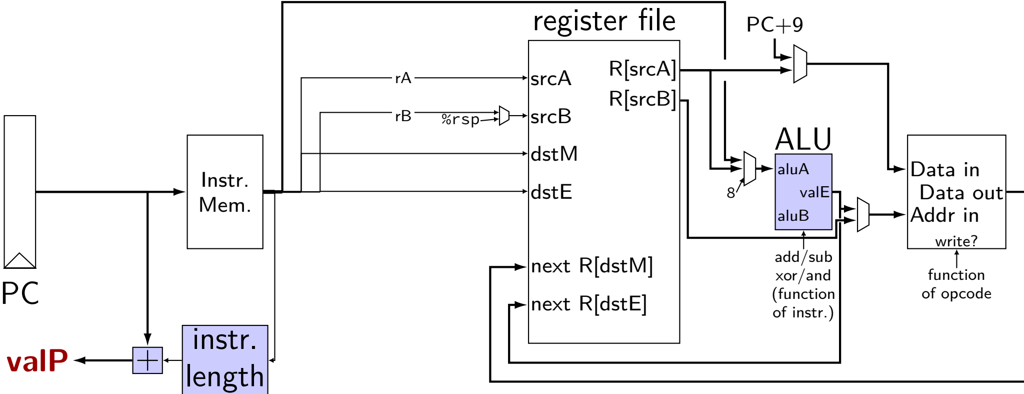
memory (2)



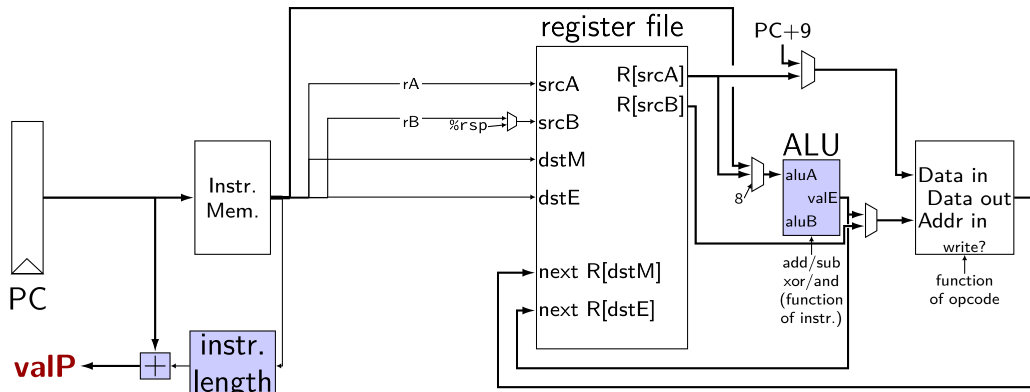
SEQ: write back

write registers

write back (1)



write back (1)



exercise: which of these instructions can this **not** work for?

`nop`, `pushq`, `mrmovq`, `popq`, `call`,

SEQ: control signals for WB

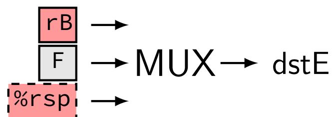
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

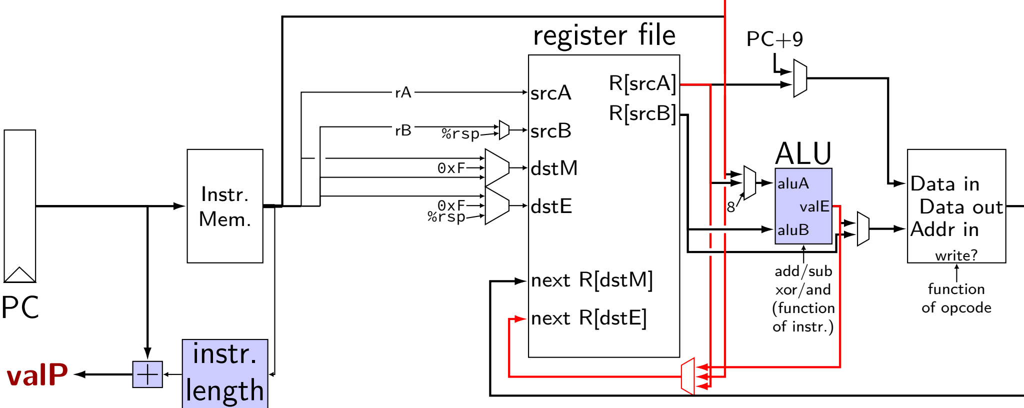
two register numbers

dstM, dstE

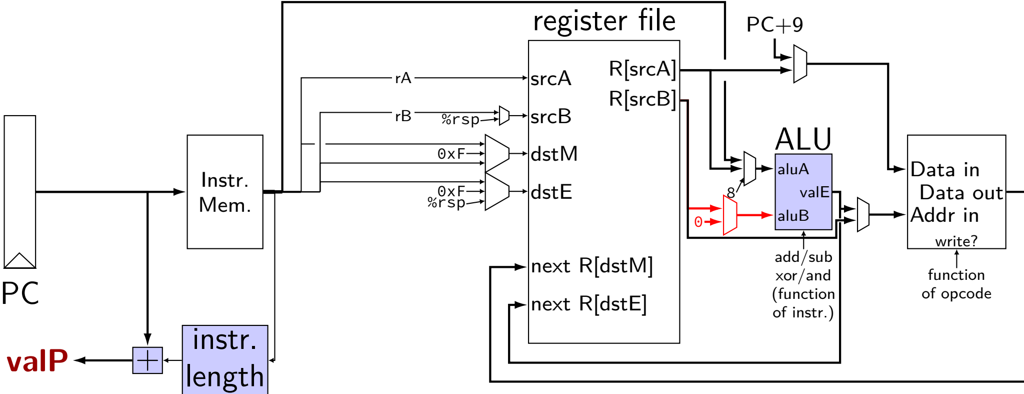
write disable — use dummy register number `0xF`



write back (2a)



write back (2b)

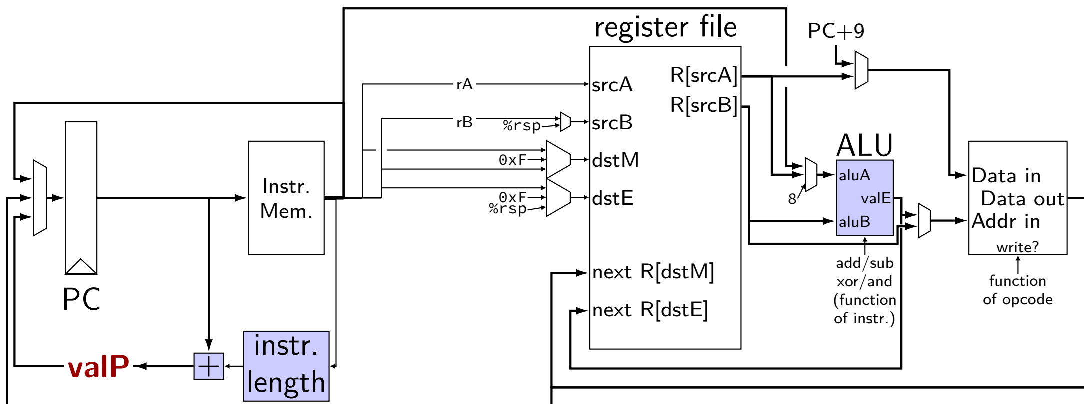


SEQ: Update PC

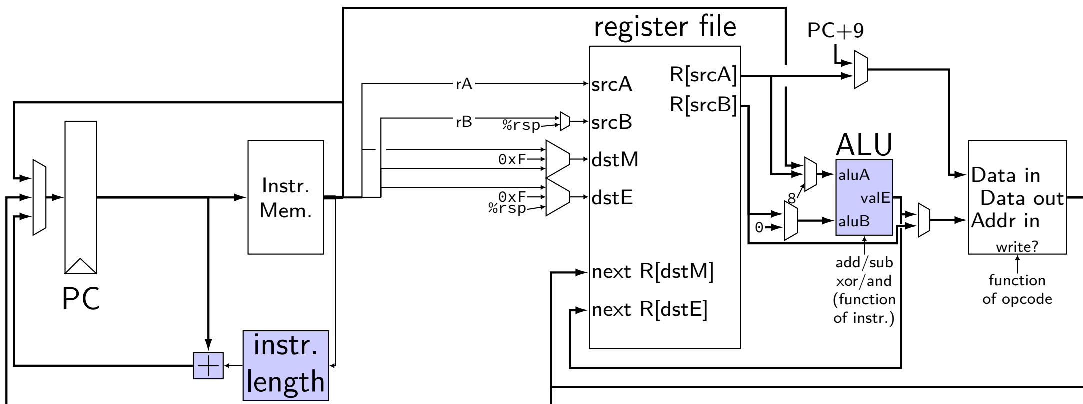
choose value for PC next cycle (input to PC register)

- usually valP (following instruction)
- exceptions: `call`, `j`*CC*, `ret`

PC update



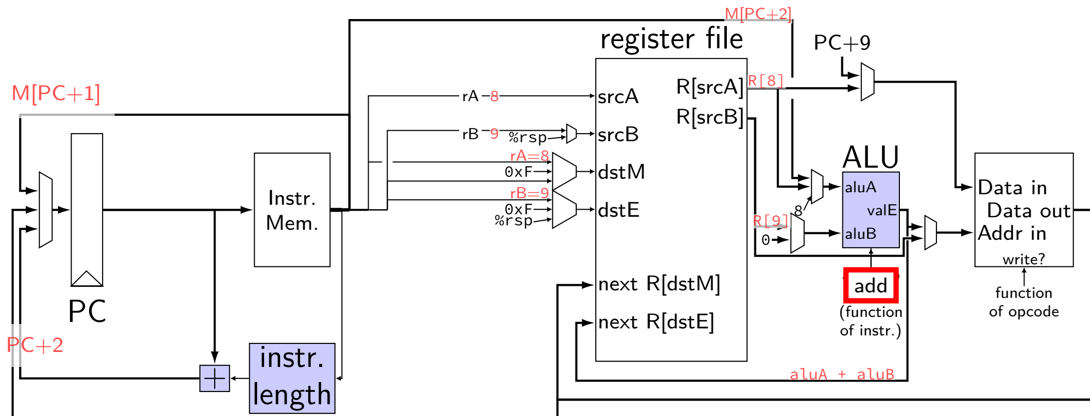
circuit: setting MUXes



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

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

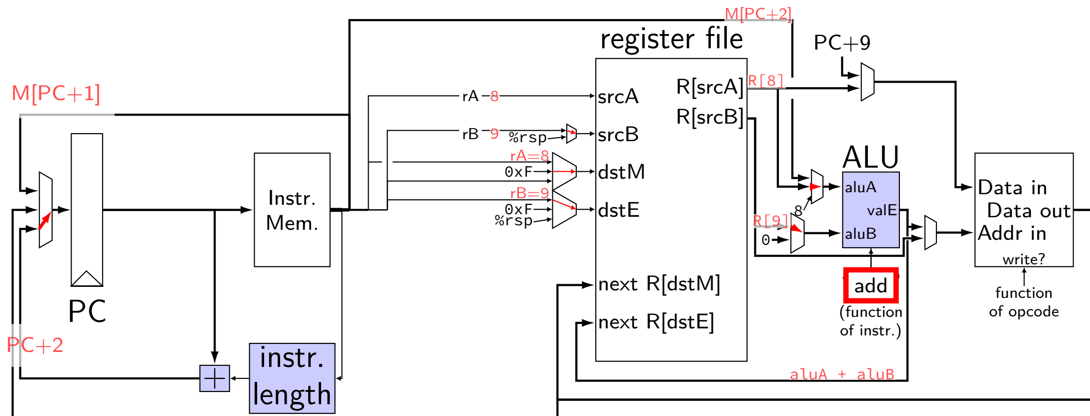
circuit: setting MUXes



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

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

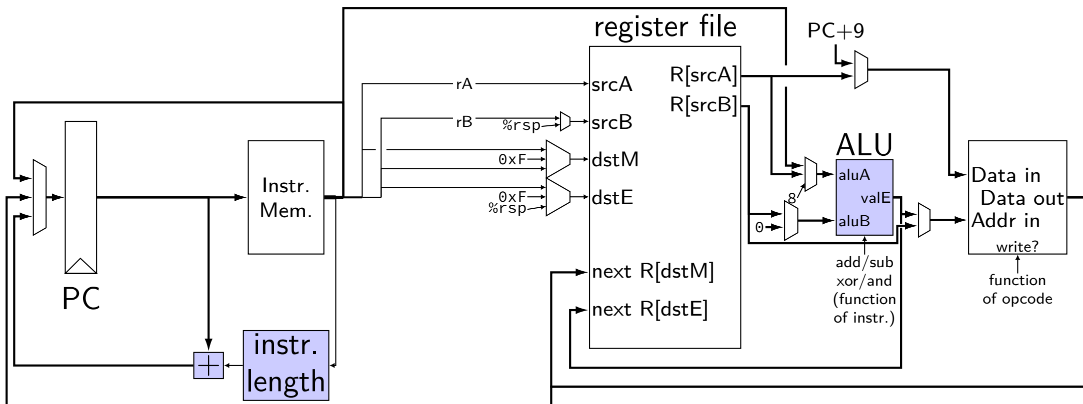
circuit: setting MUXes



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

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

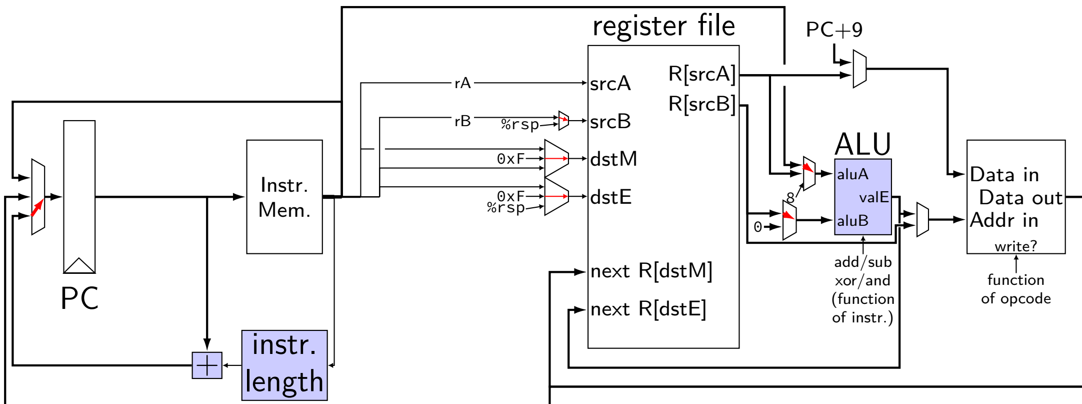
circuit: setting MUXes



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

Exercise: what do they select for `rmmovq`?

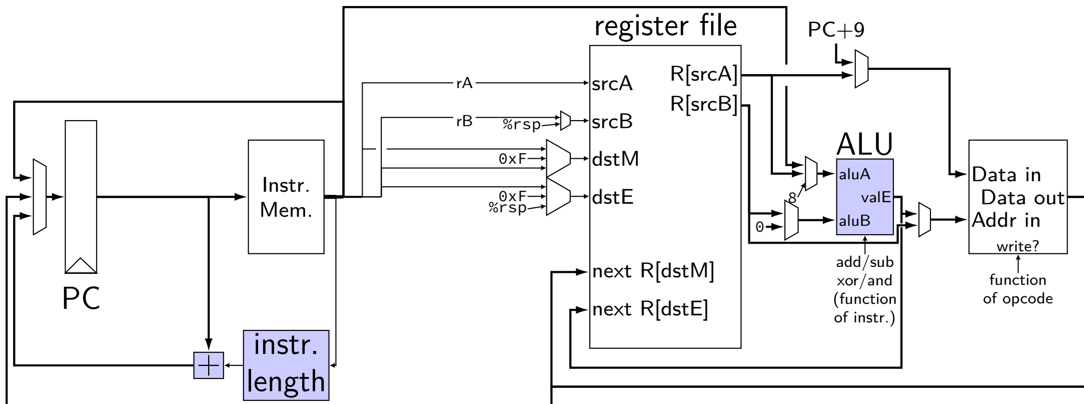
circuit: setting MUXes



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

Exercise: what do they select for `rmmovq`?

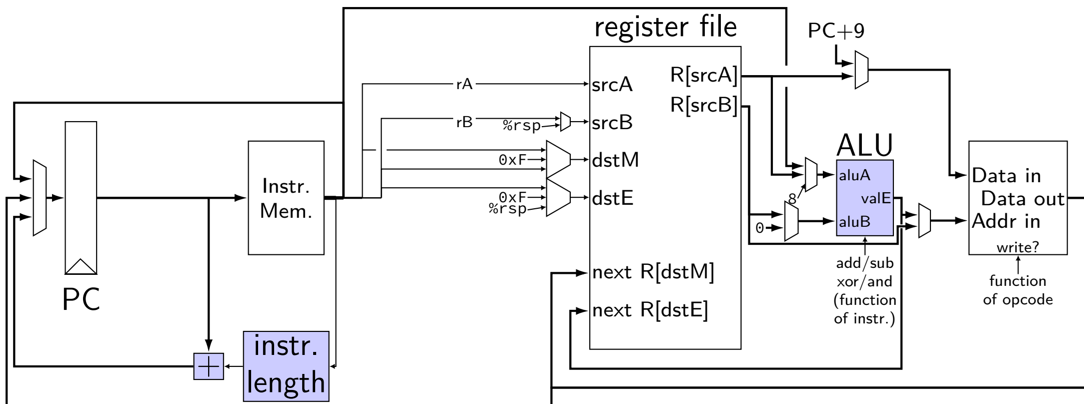
circuit: setting MUXes



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

Exercise: what do they select for `call`?

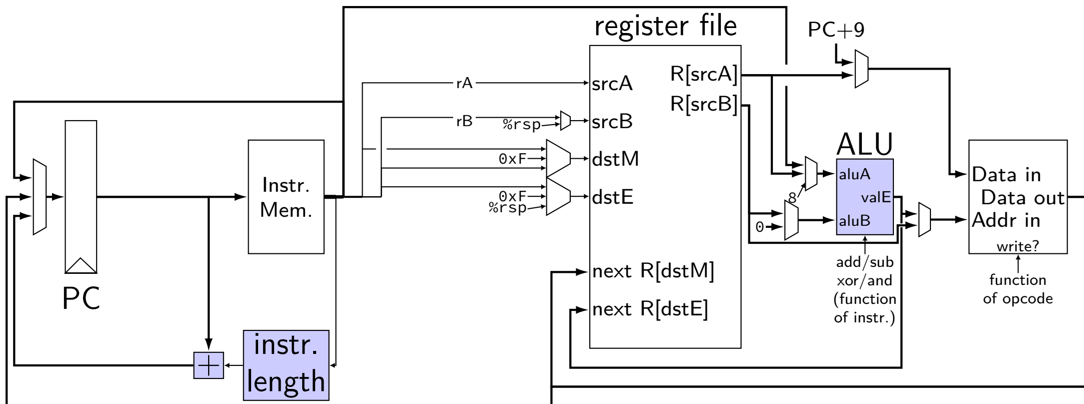
circuit: setting MUXes



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

Exercise: what do they select for `ret`?

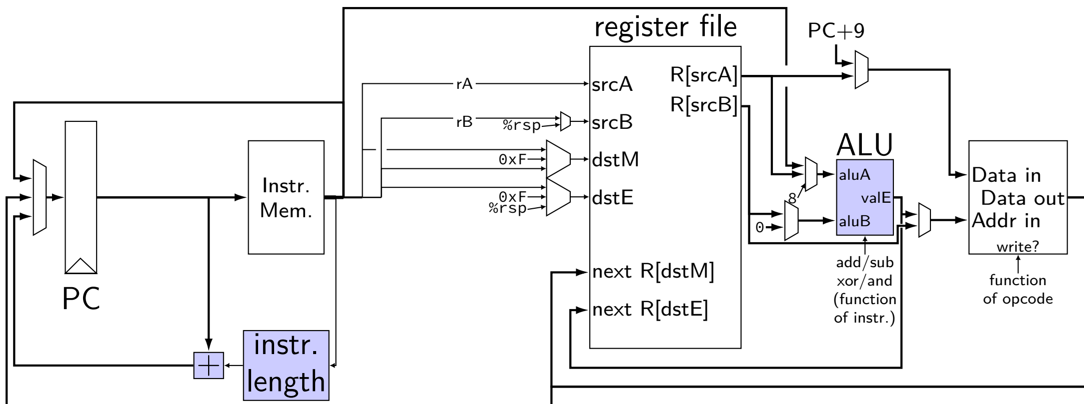
circuit: setting MUXes



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

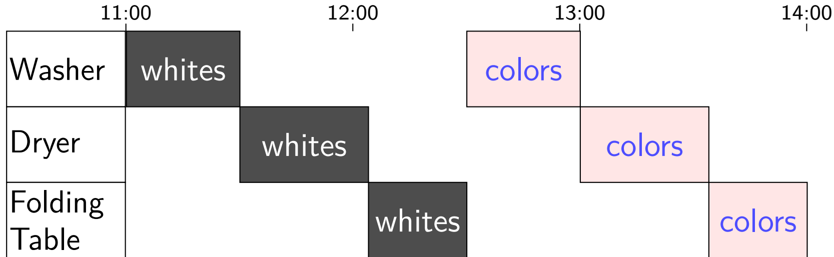
Exercise: what do they select for `irmovq`?

circuit: setting MUXes

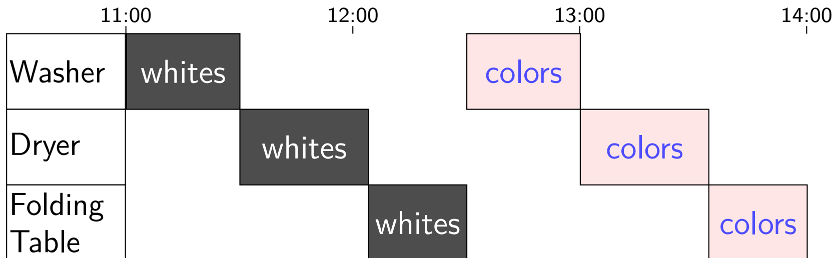


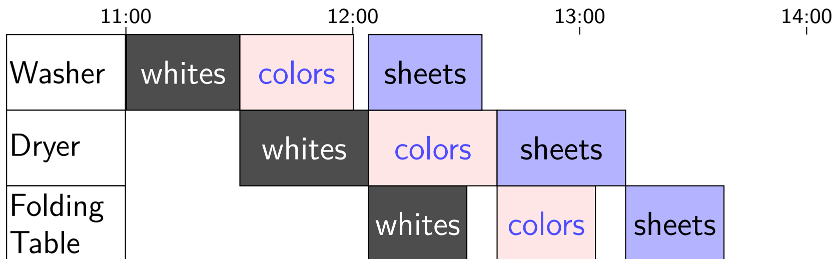
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for **popq**?

Human pipeline: laundry

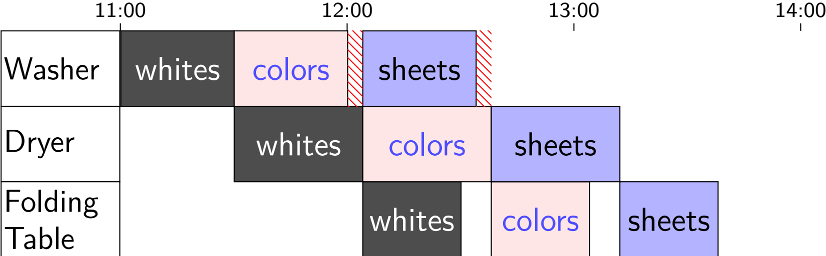


Human pipeline: laundry

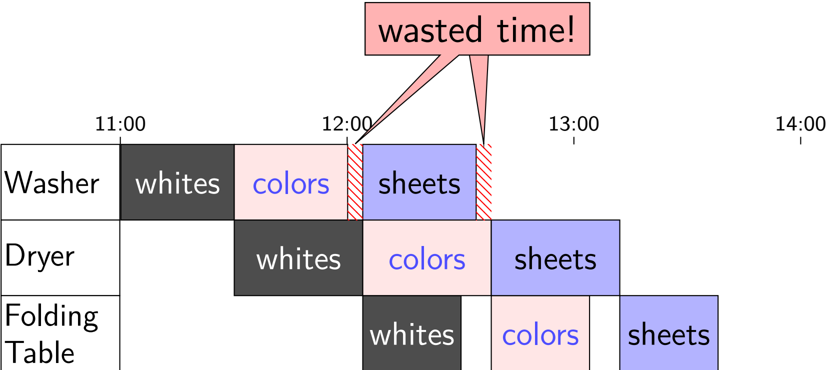




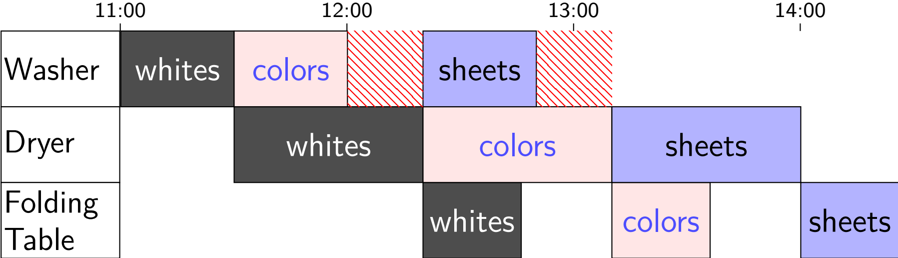
Waste (1)



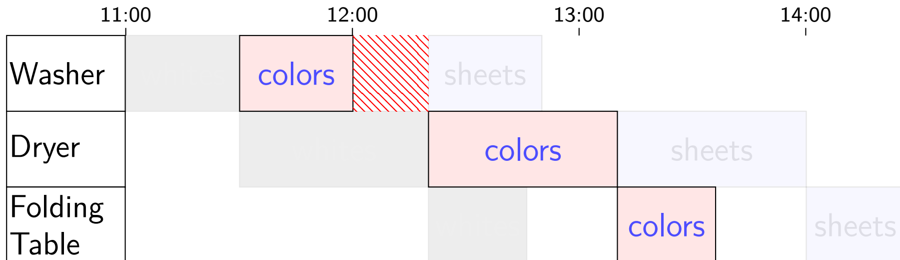
Waste (1)



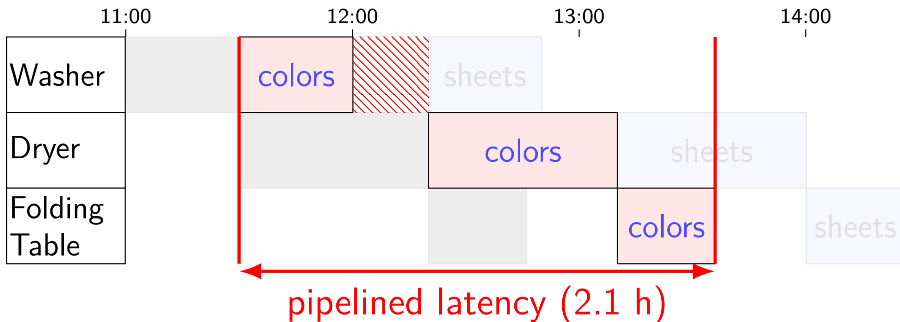
Waste (2)



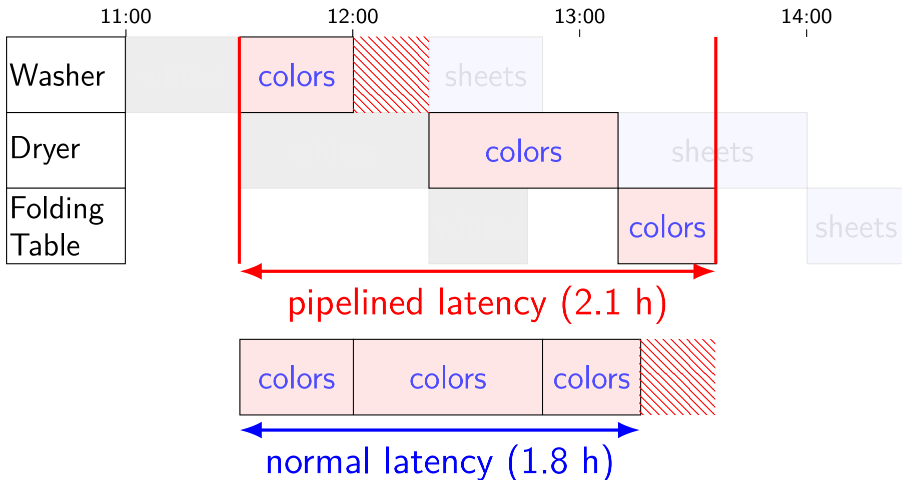
Latency — Time for One



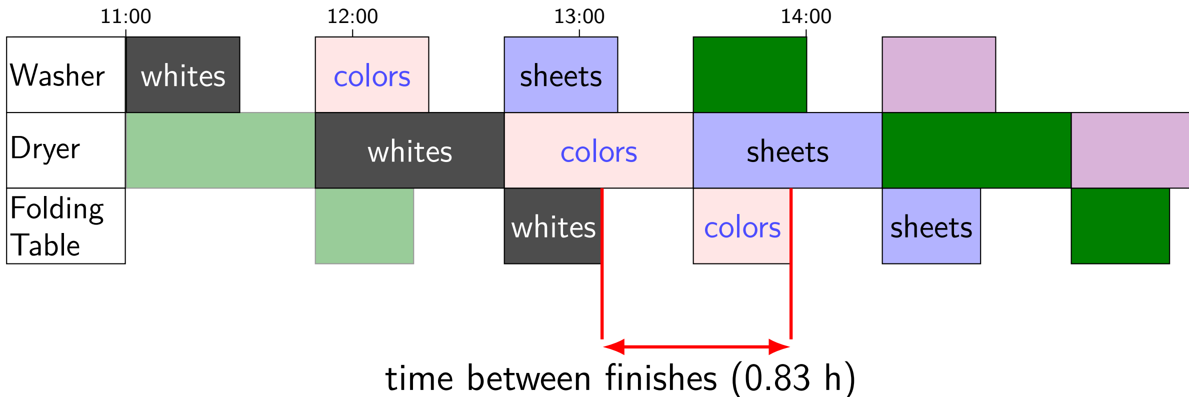
Latency — Time for One



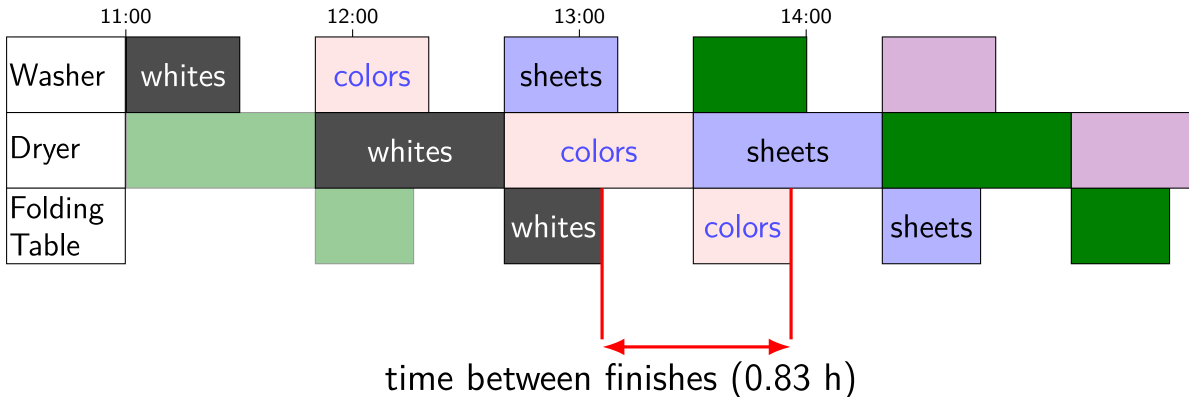
Latency — Time for One



Throughput — Rate of Many

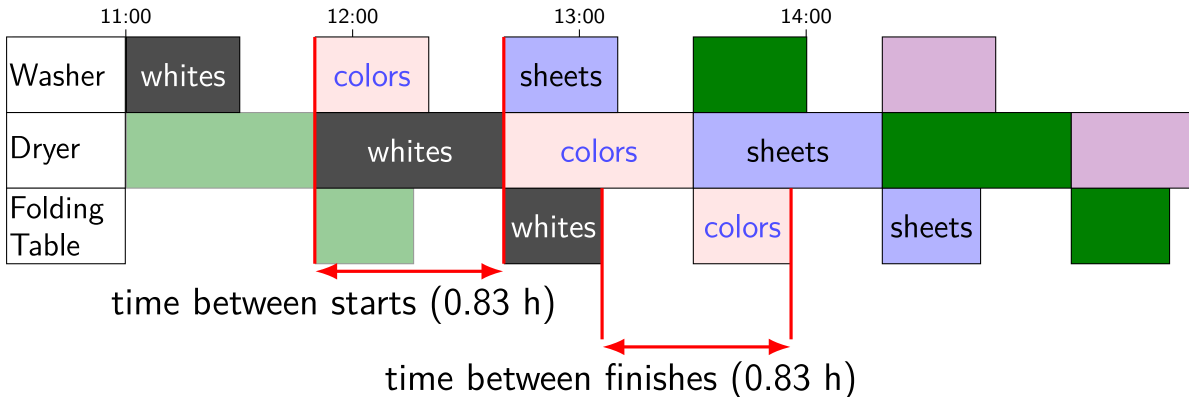


Throughput — Rate of Many



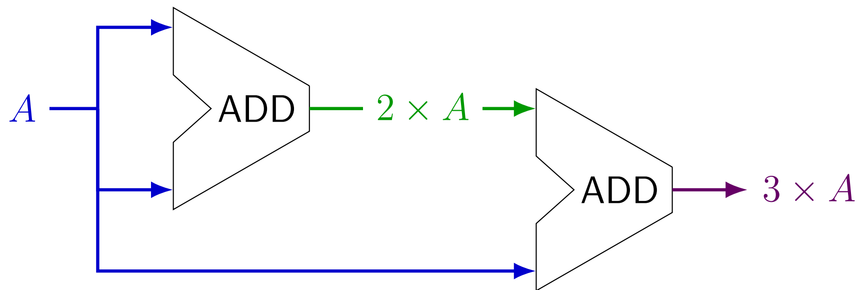
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many

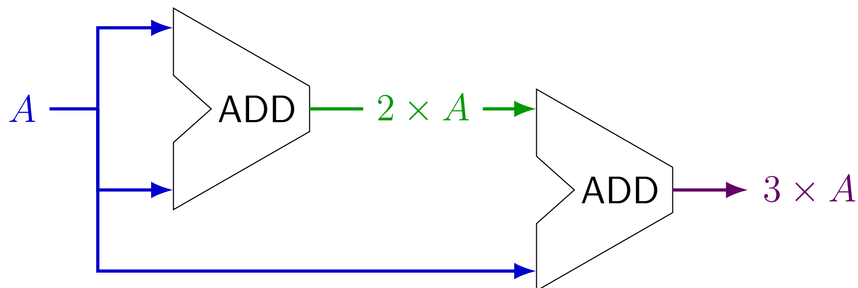


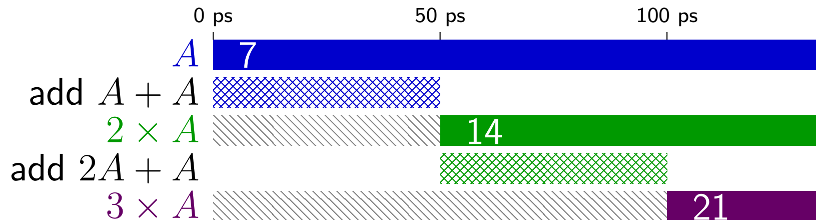
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

times three circuit

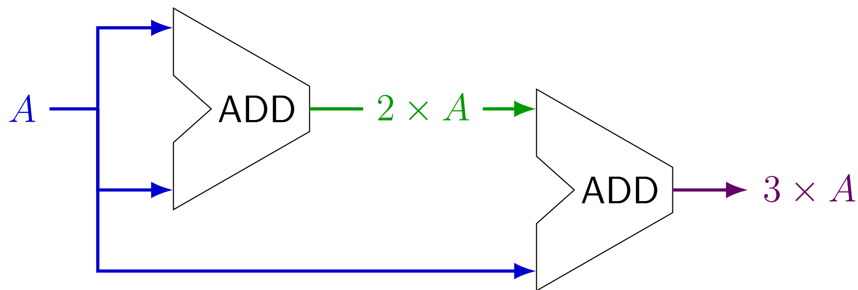


times three circuit

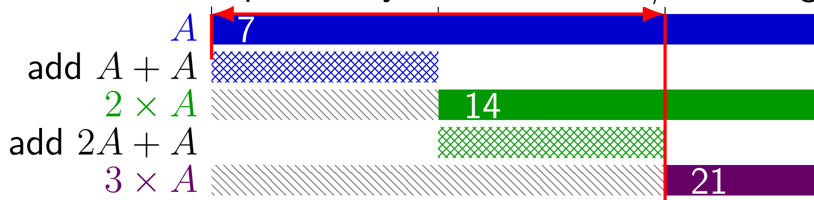




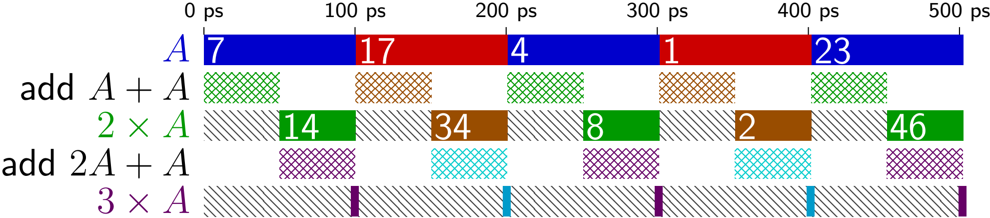
times three circuit



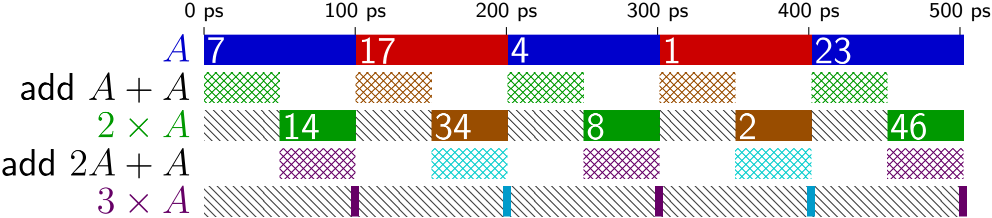
100 ps latency $\implies$ 10 results/ns throughput

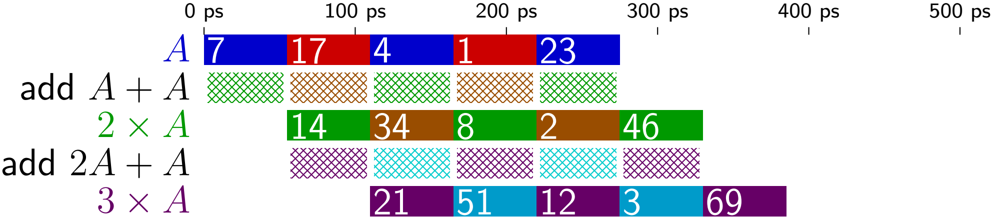


times three and repeat

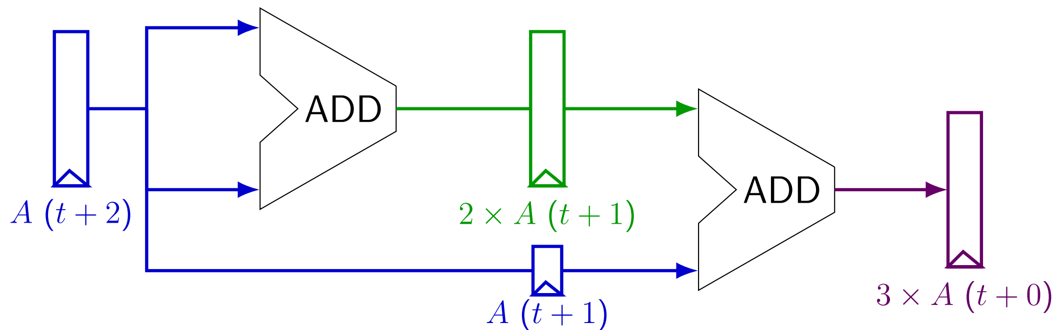


times three and repeat

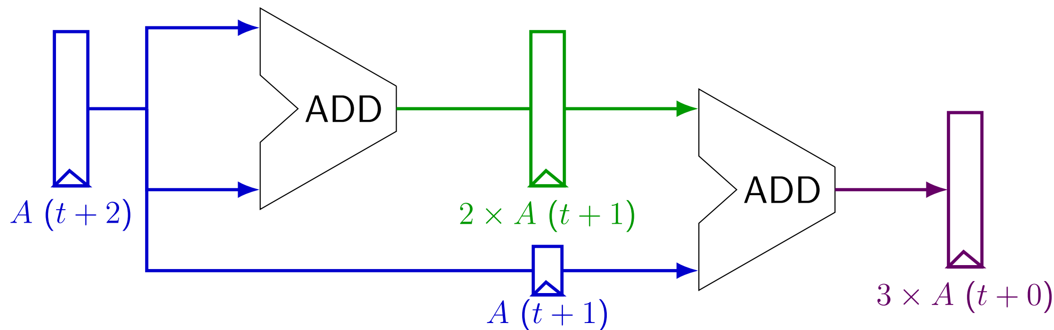




pipelined times three

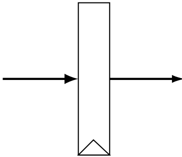


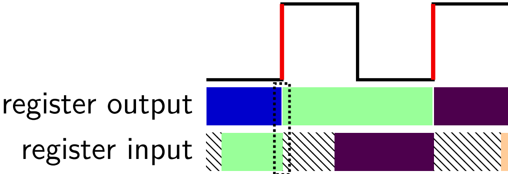
pipelined times three



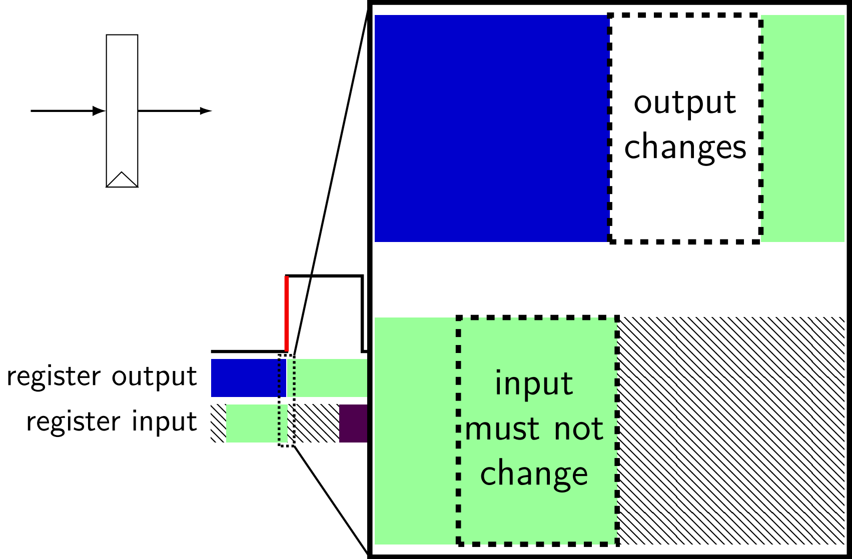
$A(t+2)$	7	17	
$A(t+1)$		7	17
$2 \times A(t+1)$		14	34
$3 \times A(t+0)$			21

register tolerances

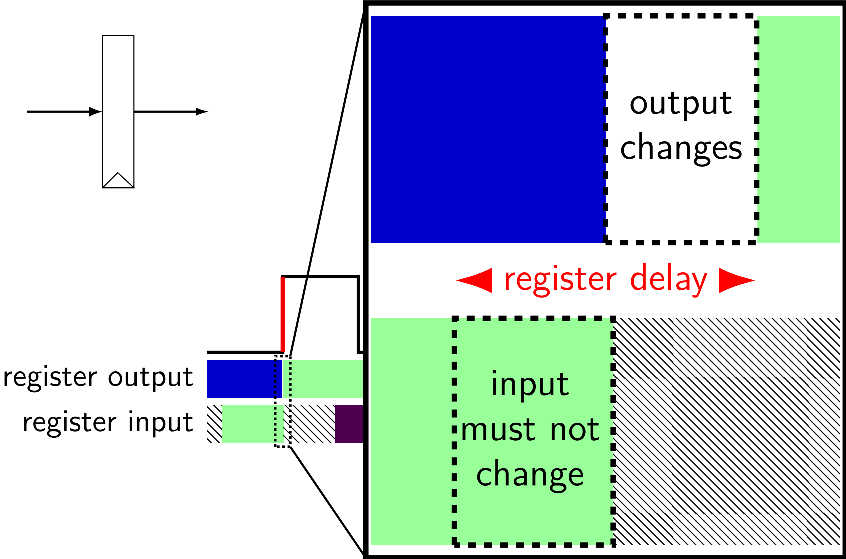




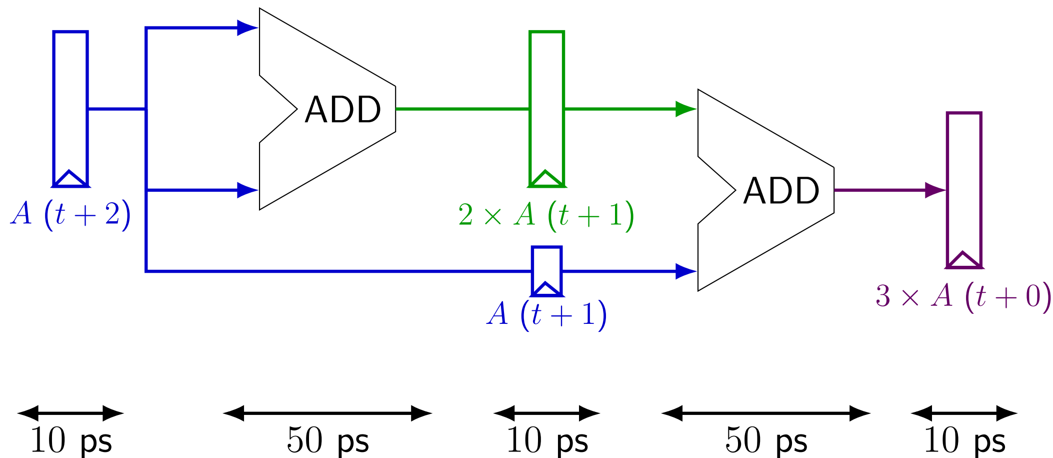
register tolerances



register tolerances



times three pipeline timing



times three pipeline timing

ADD

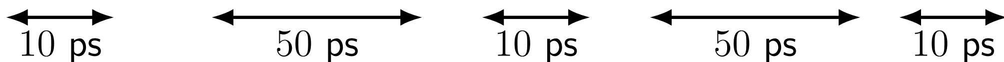
ADD

$A\ (t+2)$

$2 \times A\ (t+1)$

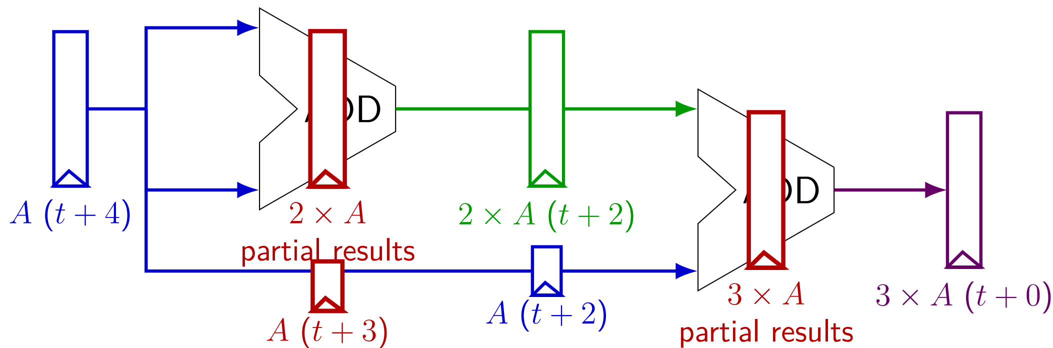
$A\ (t+1)$

$3 \times A\ (t+0)$

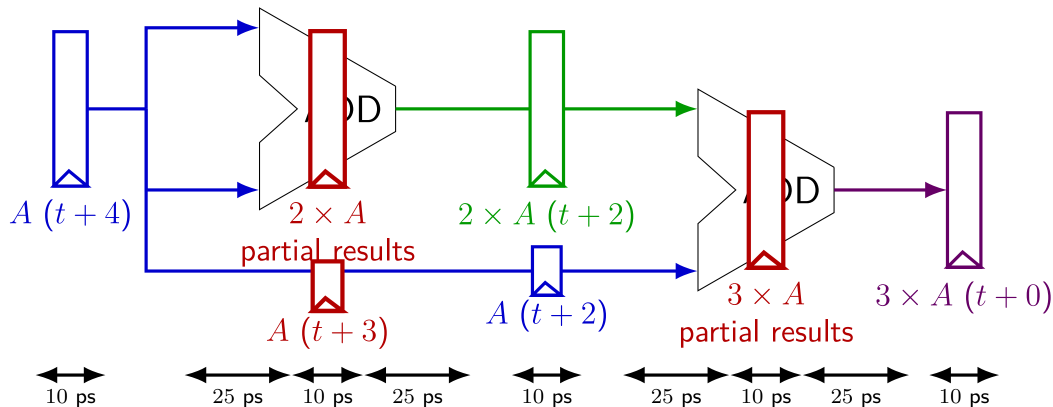


throughput: $\frac{1}{60\ \text{ps}} \approx 16$ G operations/sec

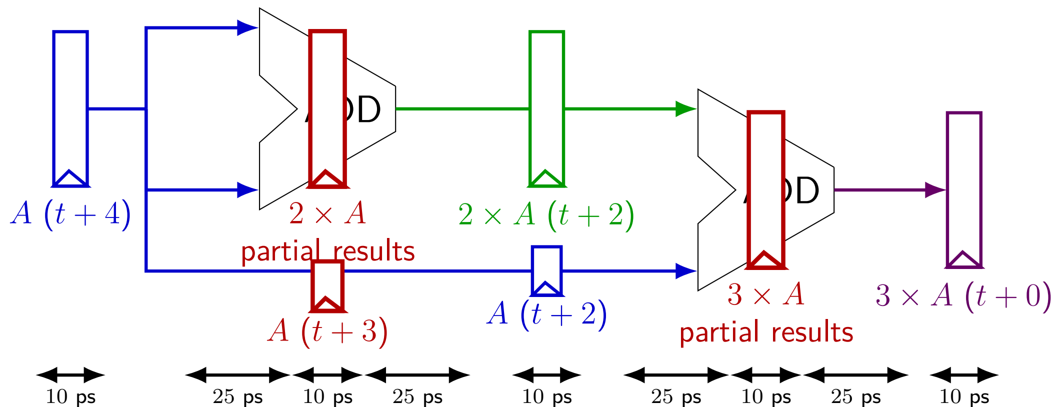
deeper pipeline



deeper pipeline

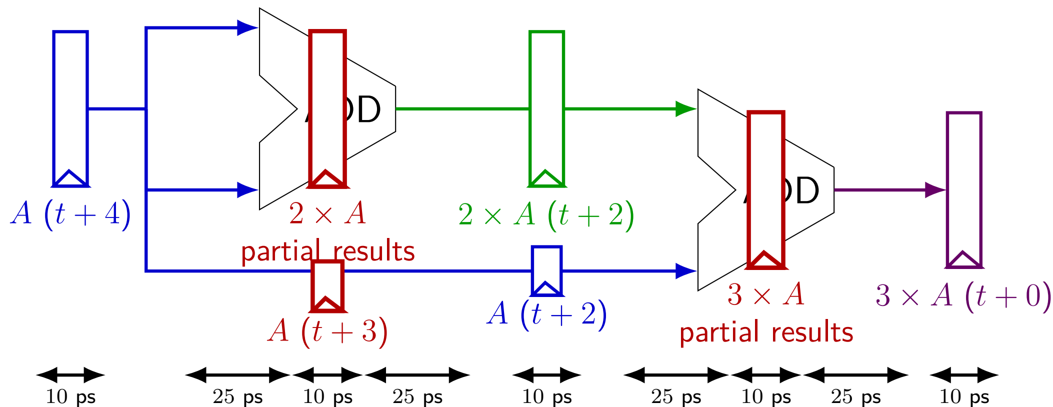


deeper pipeline

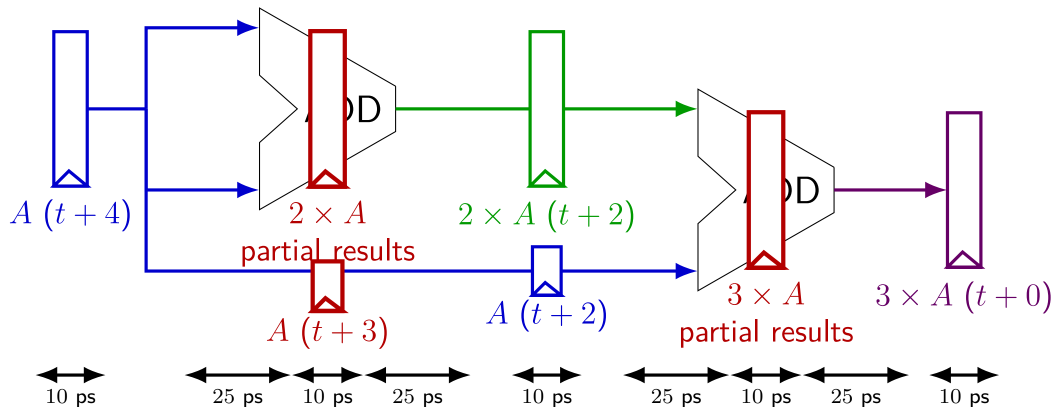


exercise: throughput now?

deeper pipeline



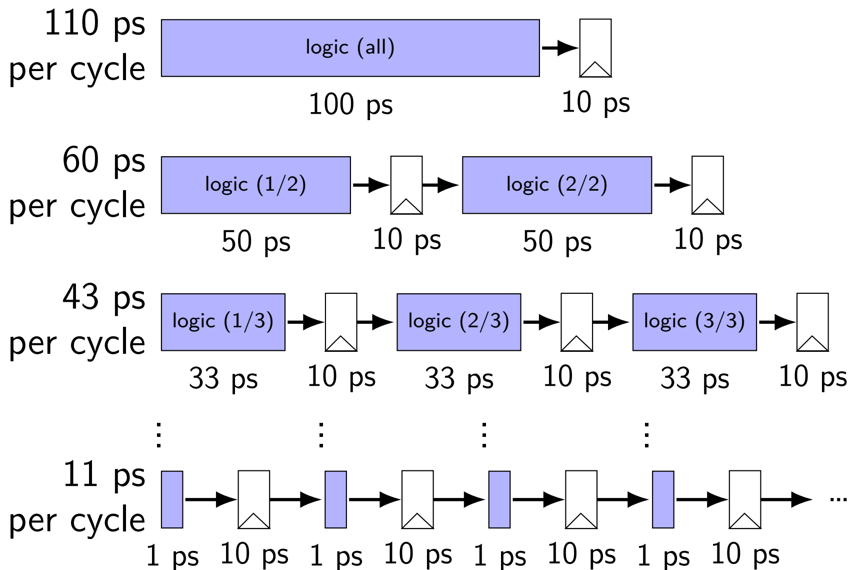
deeper pipeline



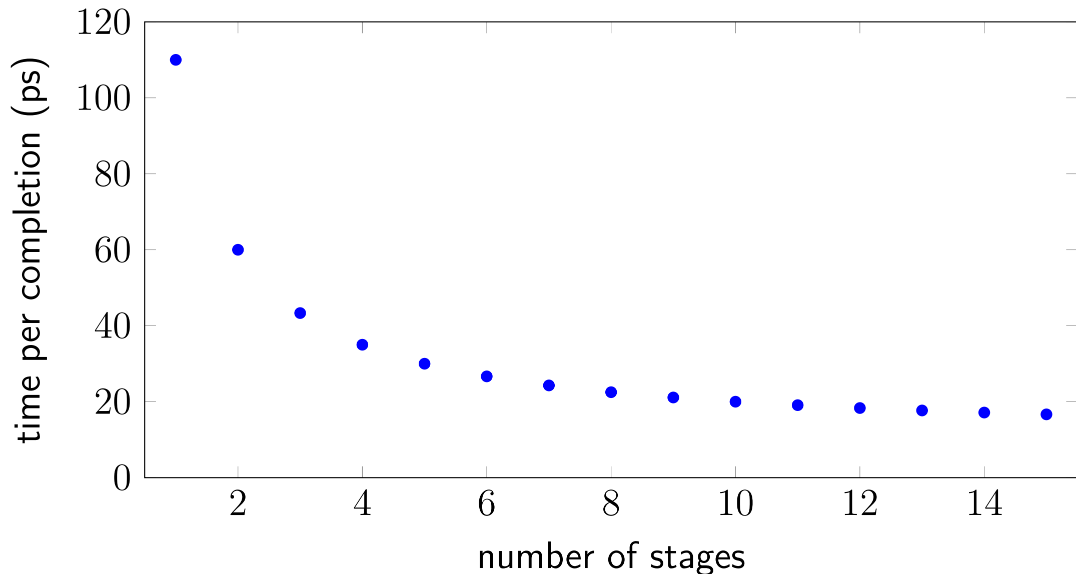
Problem: How much faster can we get?

Problem: Can we even do this?

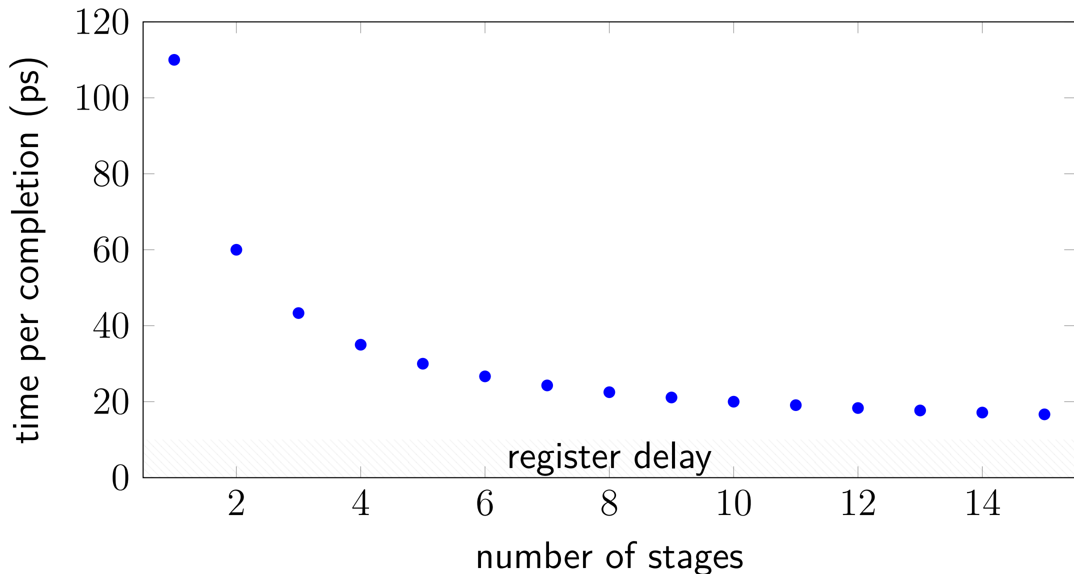
diminishing returns: register delays



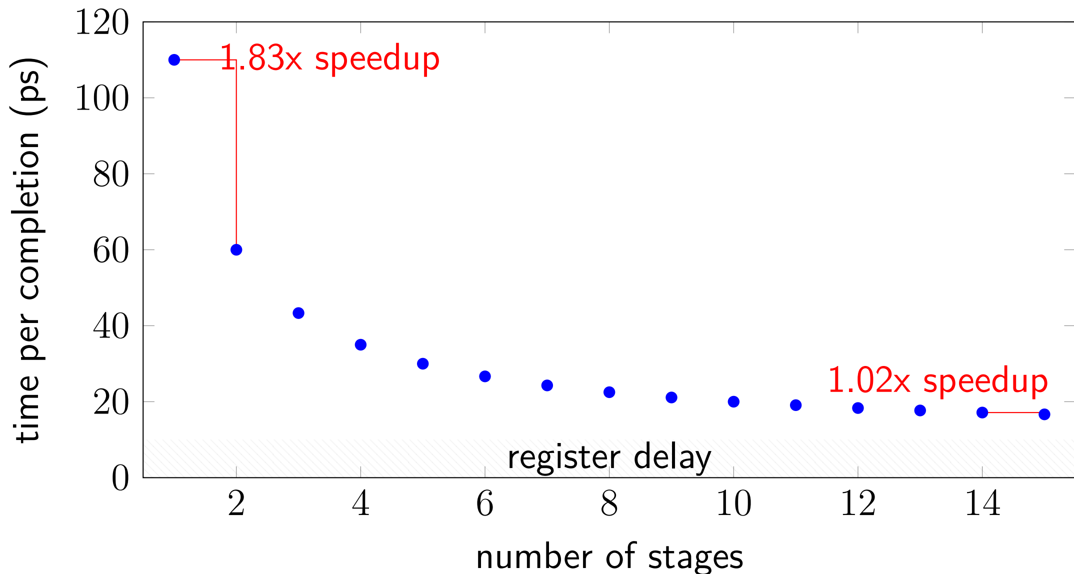
diminishing returns: register delays



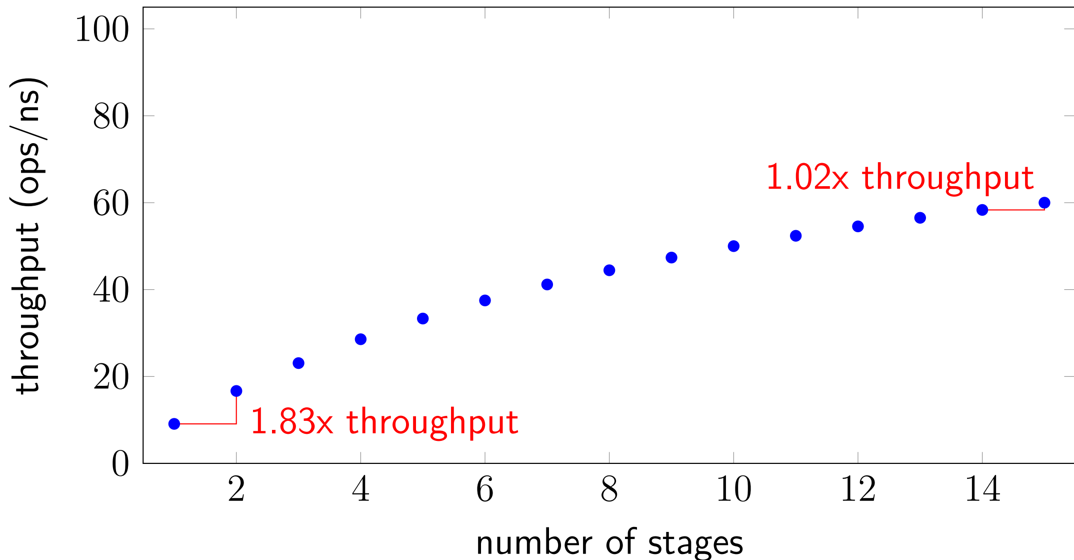
diminishing returns: register delays



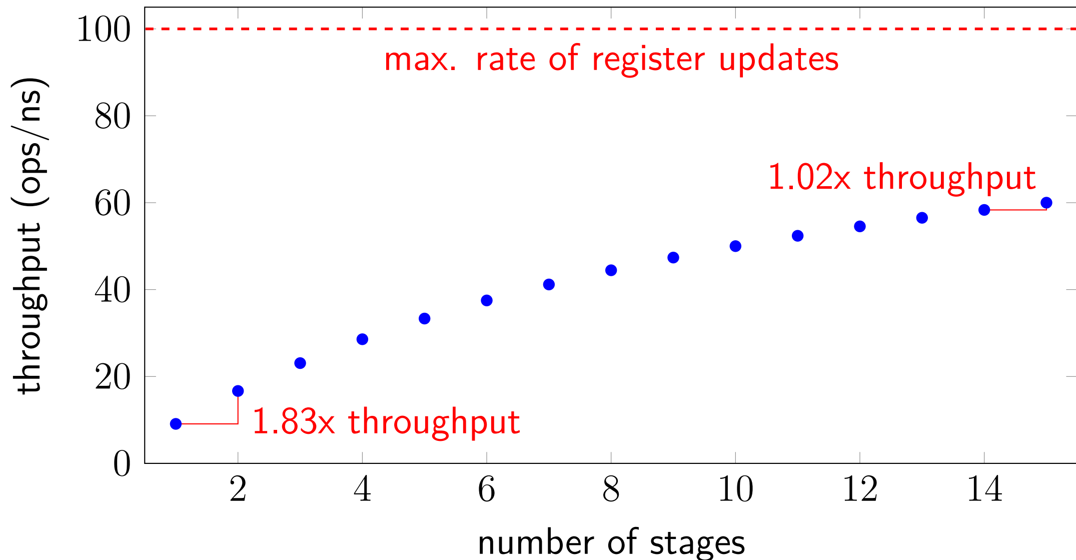
diminishing returns: register delays



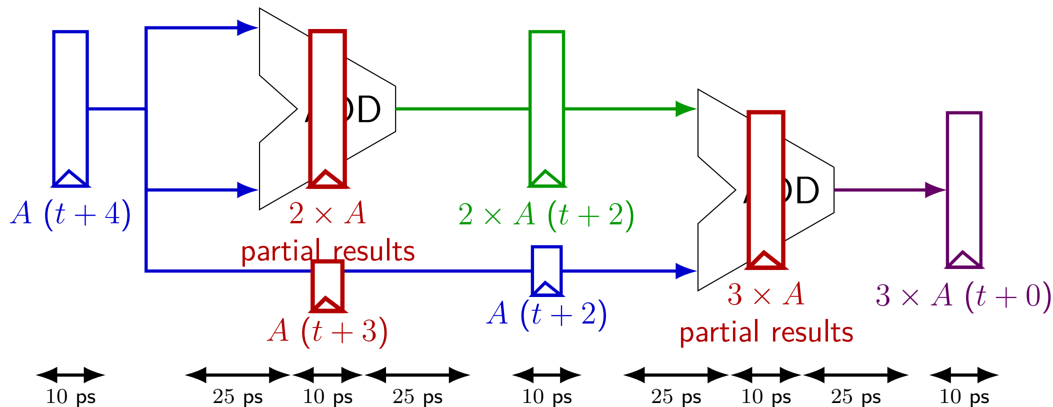
diminishing returns: register delays



diminishing returns: register delays



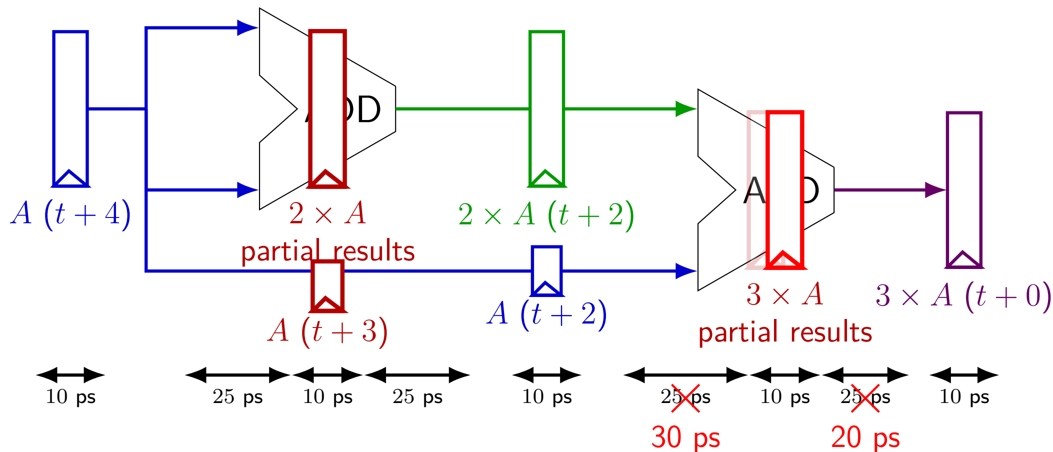
deeper pipeline



Problem: How much faster can we get?

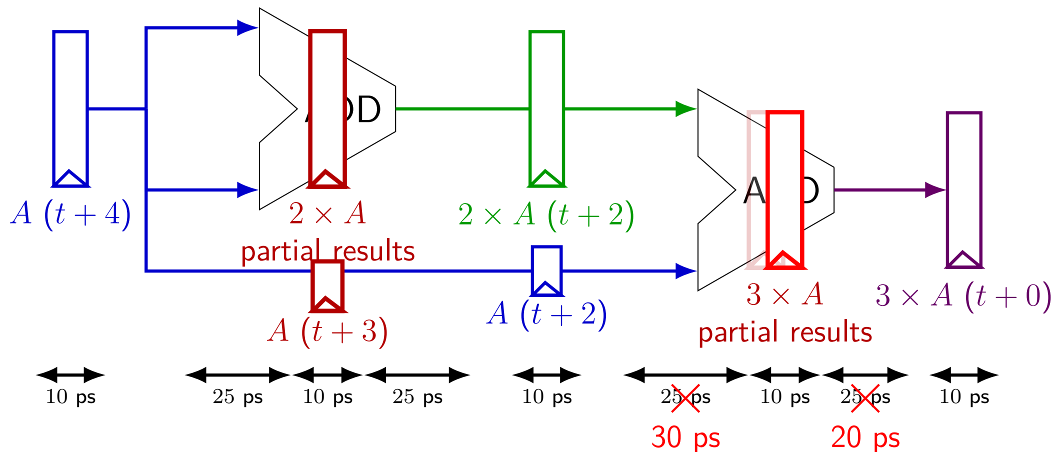
Problem: Can we even do this?

deeper pipeline



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

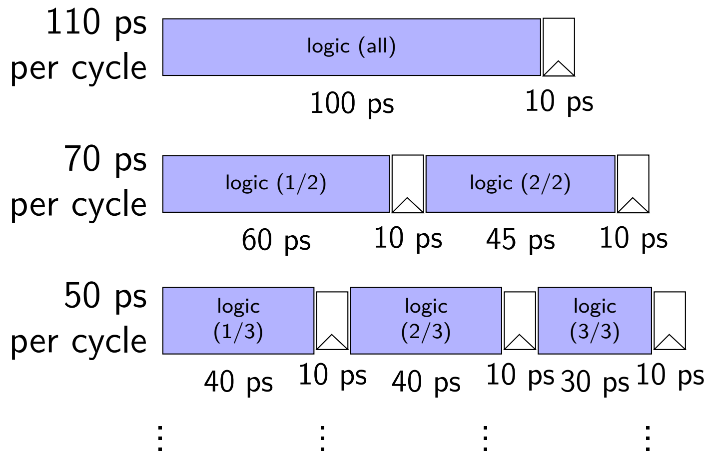
deeper pipeline



diminishing returns: uneven split

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

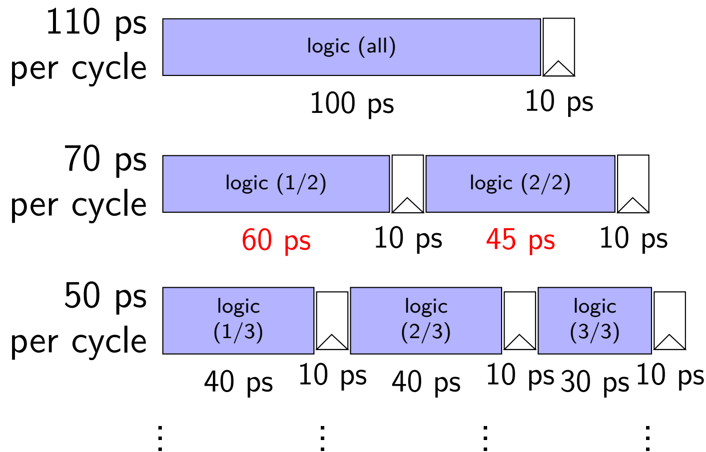
Probably not...



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...