SEQ: instruction fetch

read instruction memory at PC

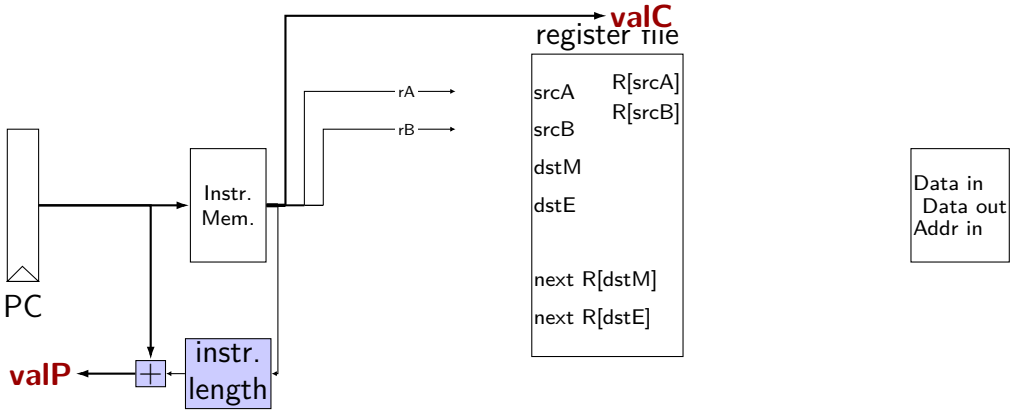
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

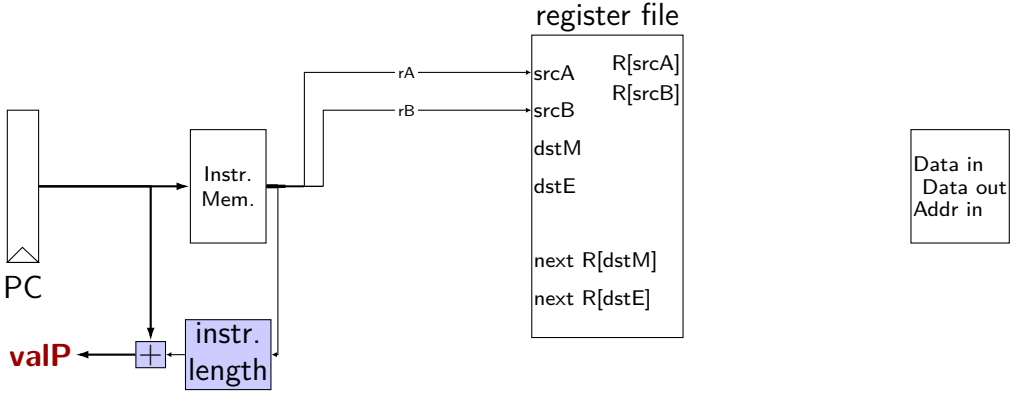


SEQ: instruction “decode”

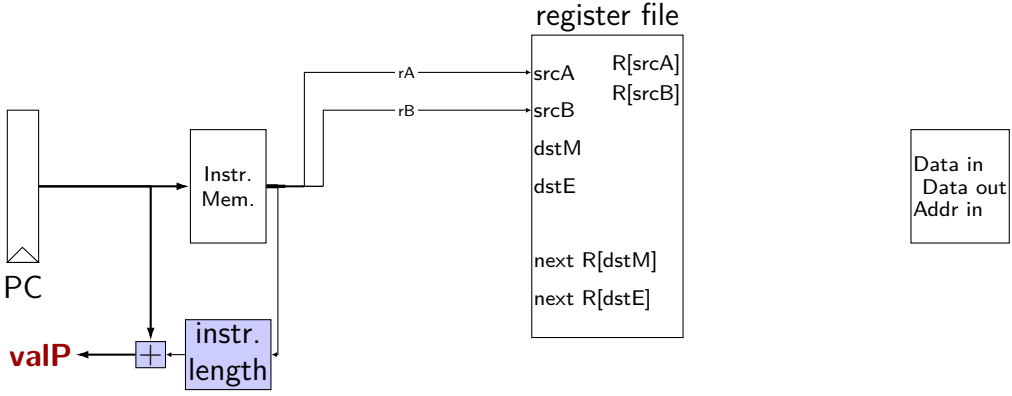
read registers

valA, valB — register values

instruction decode (1)



instruction decode (1)



exercise: which of these instructions can this **not** work for?
nop, addq, `mrmovq`, pushq, `jmp`,

SEQ: srcA, srcB

always read rA, rB?

Problems: (not planned to be included our assignments)

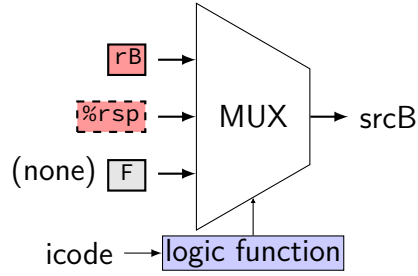
- push rA
- pop
- call
- ret

book: extra signals: srcA, srcB — computed input register

MUX controlled by icode

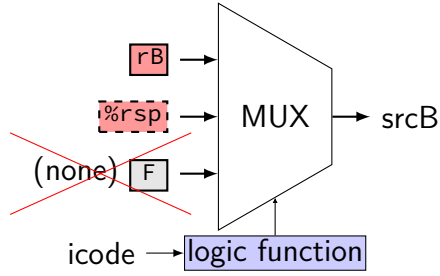
SEQ: possible registers to read

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`

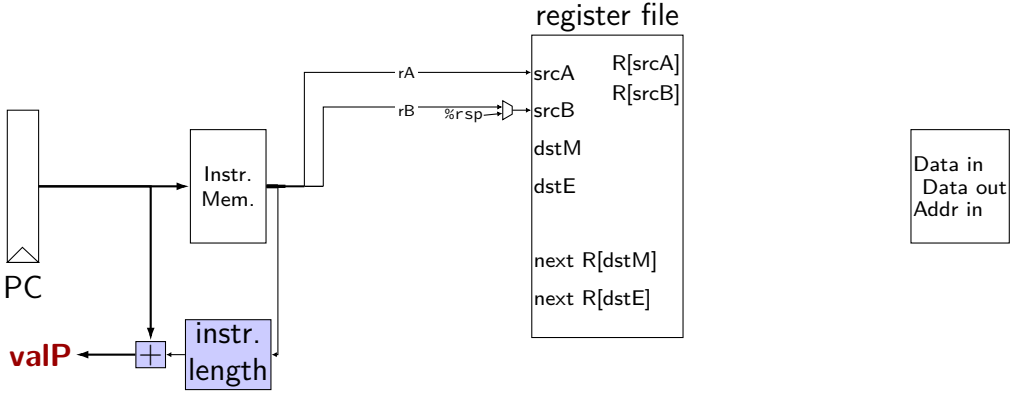


SEQ: possible registers to read

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)

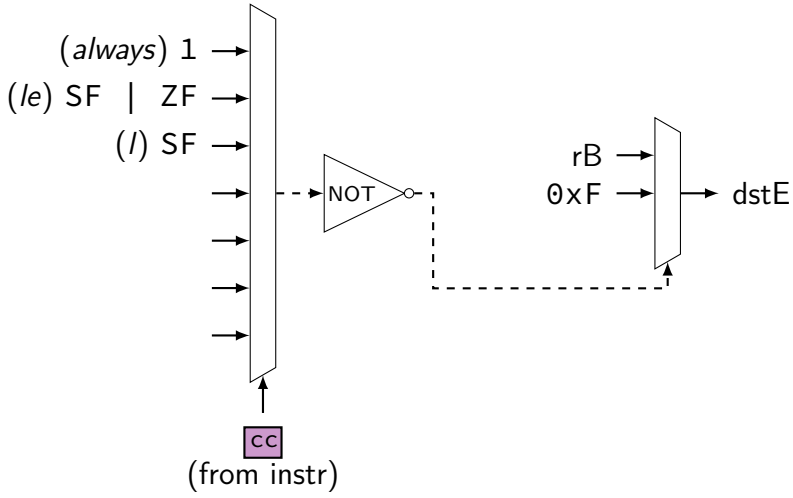
- valE — ALU output

read prior condition codes

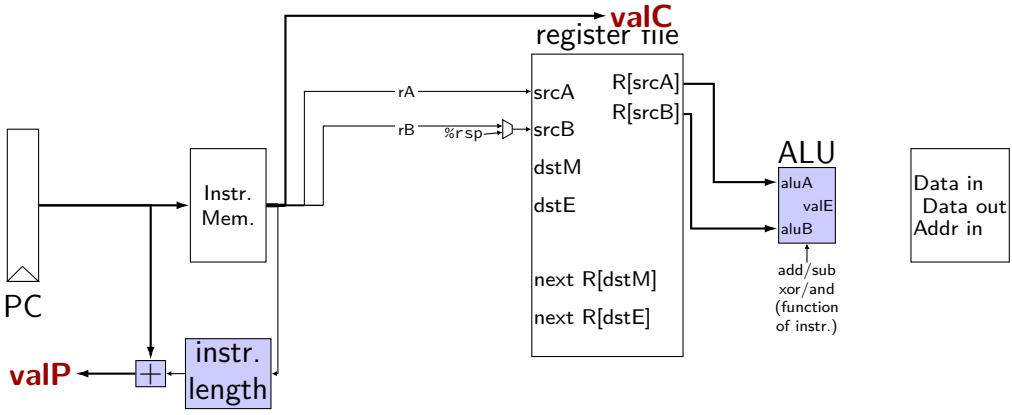
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

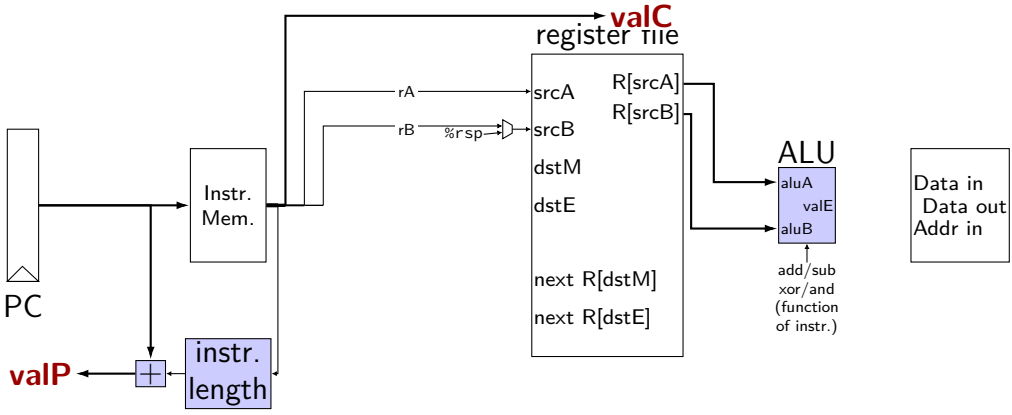
using condition codes: cmov



execute (1)


valC
register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
Instr.
Mem.
PC
valP
instr.
length

execute (1)



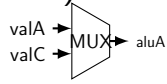
exercise: which of these instructions can this **not** work for?
`nop`, `addq`, `mrmovq`, `popq`, `call`,

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

- `mrmovq`
- `rmmovq`



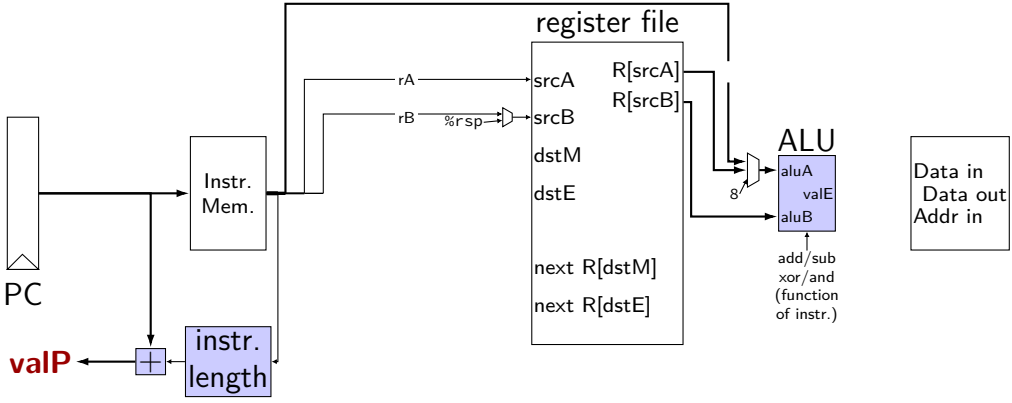
no, constants: (rsp +/- 8) (not planned to be in assignments)

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

extra signals: aluA, aluB

computed ALU input values

execute (2)

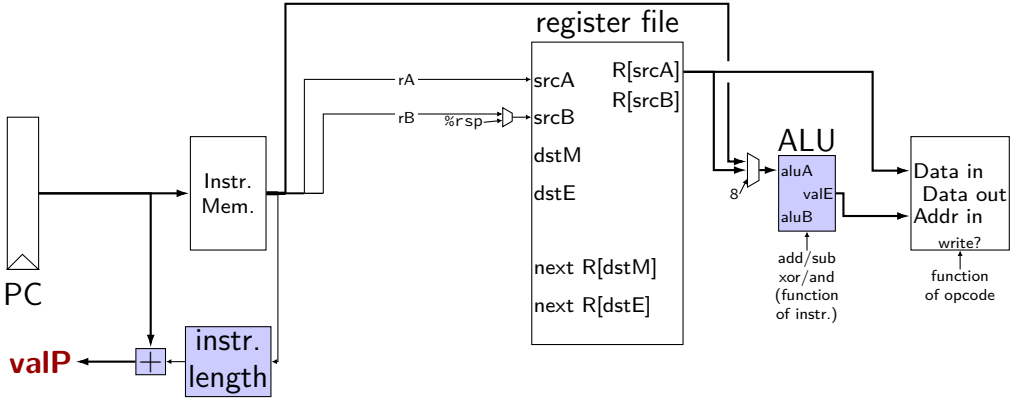

register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
Instr.
Mem.
PC
valP
instr.
length

SEQ: Memory

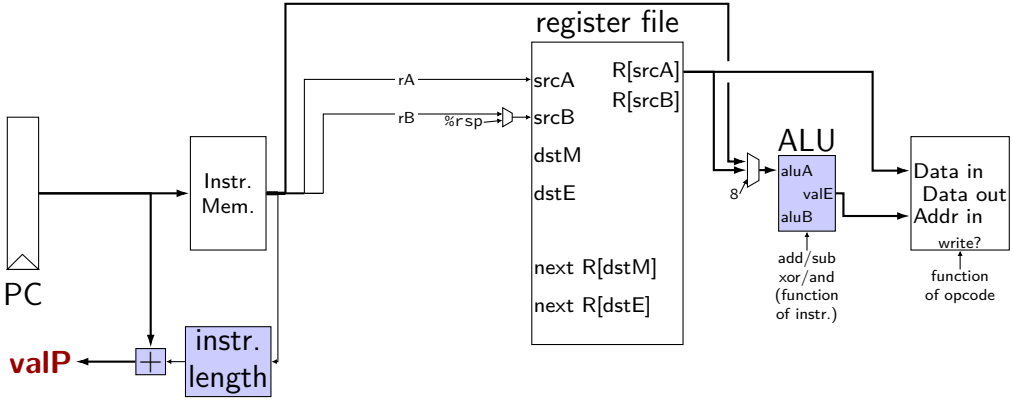
read or write data memory

valM — value read from memory (if any)

memory (1)


register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
write?
function
of opcode
Instr.
Mem.
PC
valP
instr.
length

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?

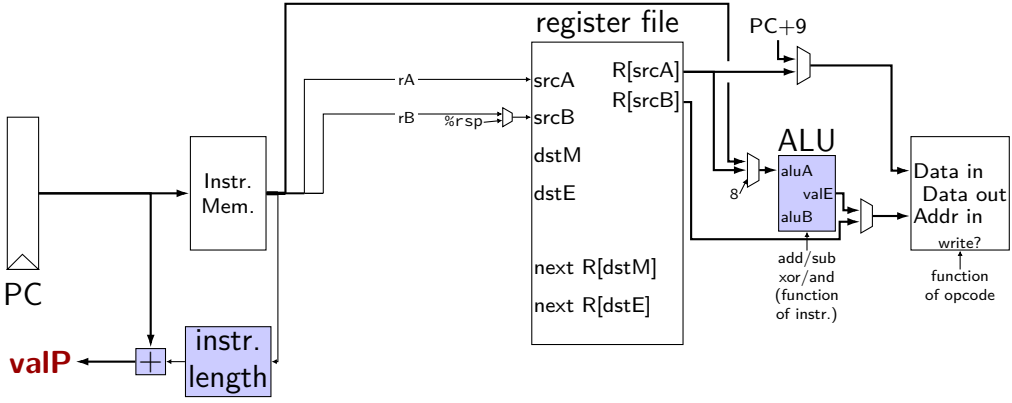
Addr — address

- mostly ALU output
- special cases (need extra MUX): `popq`, `ret`

Data — value to write

- mostly valA
- special cases (need extra MUX): `call`

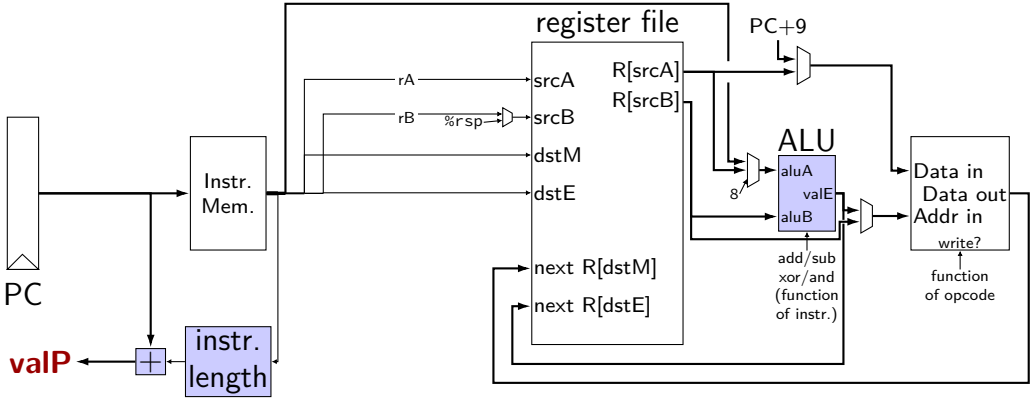
memory (2)


register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
Data in
Data out
Addr in
write?
add/sub
xor/and
(function
of instr.)
function
of opcode
Instr.
Mem.
PC
valP
instr.
length

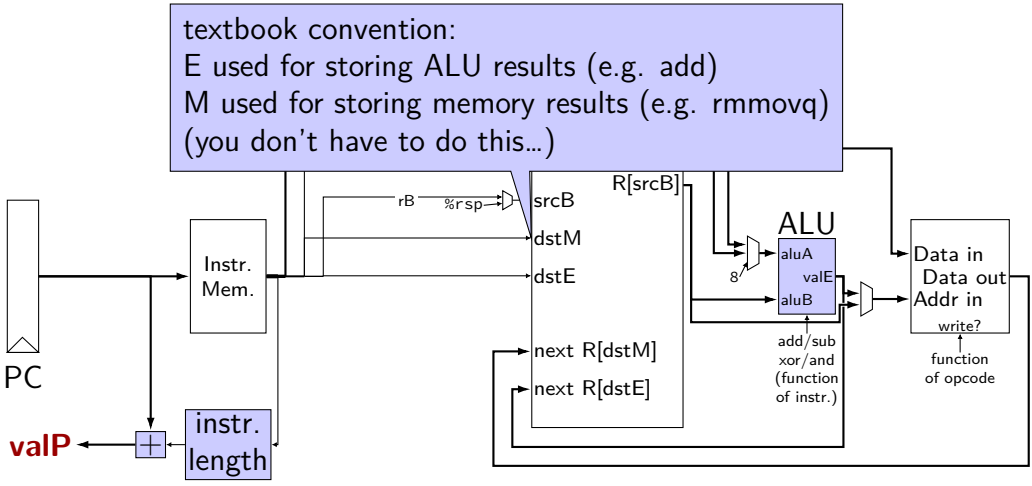
SEQ: write back

write registers

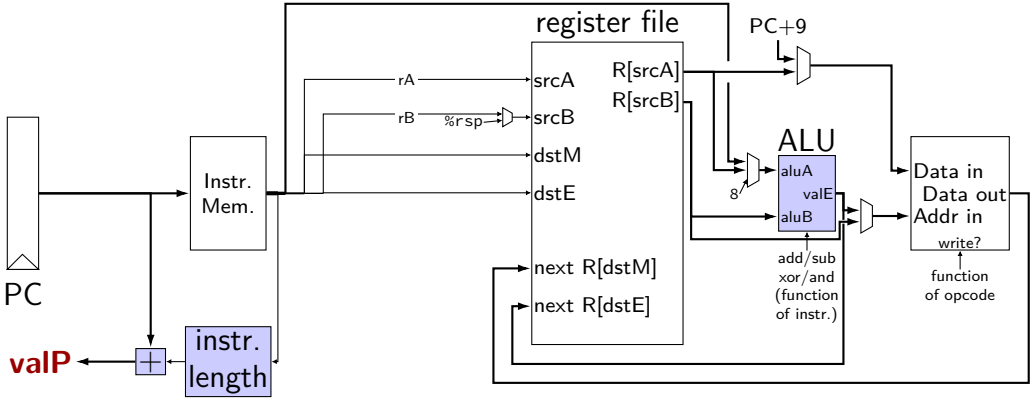
write back (1)


register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
ALU
aluA
valE
aluB
8
Data in
Data out
Addr in
write?
function
of opcode
add/sub
xor/and
(function
of instr.)
next R[dstM]
next R[dstE]
Instr.
Mem.
PC
valP
instr.
length

write back (1)


textbook convention:
E used for storing ALU results (e.g. add)
M used for storing memory results (e.g. rmmovq)
(you don't have to do this...)
PC
Instr.
Mem.
rB
%rsp
srcB
dstM
dstE
R[srcB]
ALU
aluA
valE
aluB
8
Data in
Data out
Addr in
write?
add/sub
xor/and
(function
of instr.)
function
of opcode
next R[dstM]
next R[dstE]
valP
instr.
length

write back (1)



exercise: which of these instructions can this **not** work for?
`nop`, `irmovq`, `mrmovq`, `pushq`, `addq`

SEQ: control signals for WB

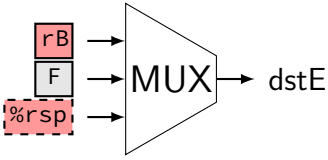
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

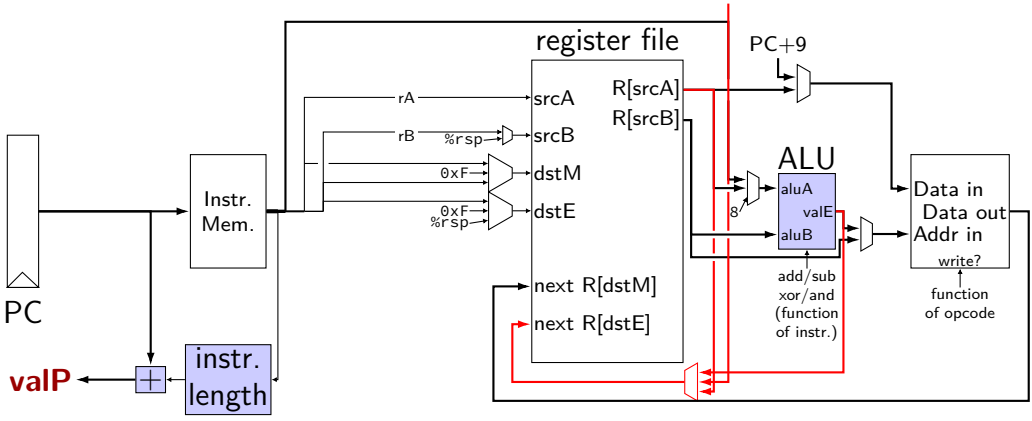
two register numbers

dstM, dstE

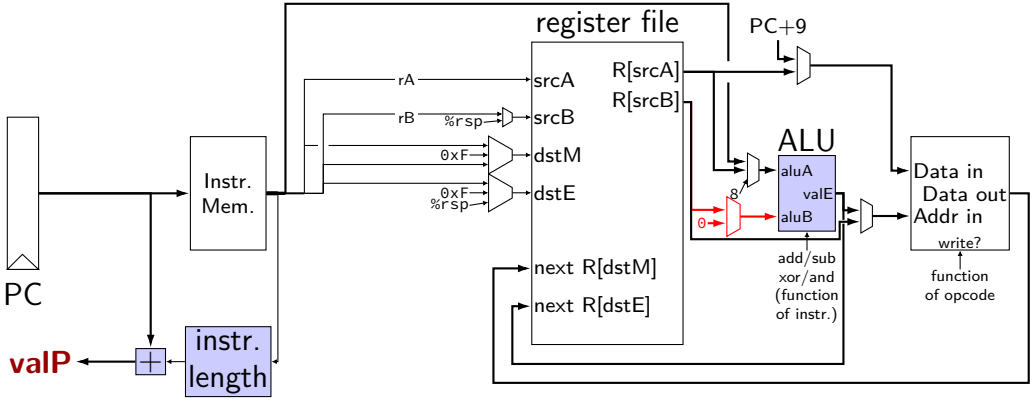
write disable — use dummy register number `0xF`



write back (2a)



write back (2b)

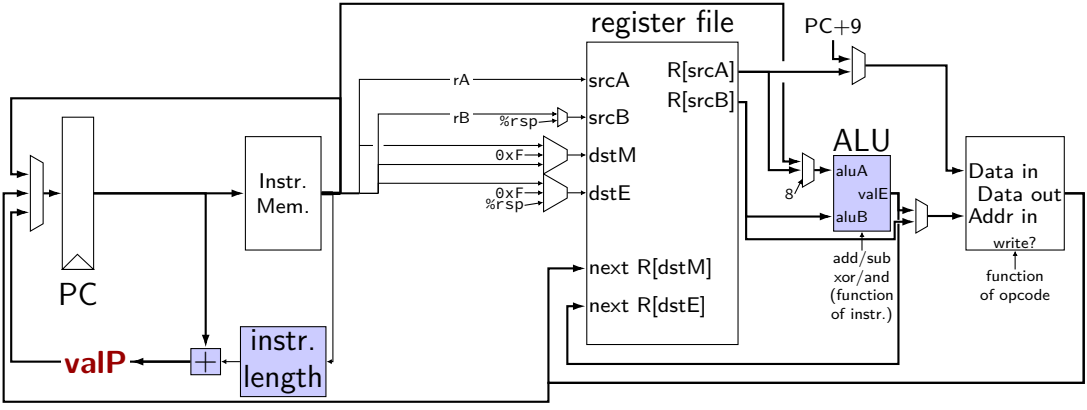


SEQ: Update PC

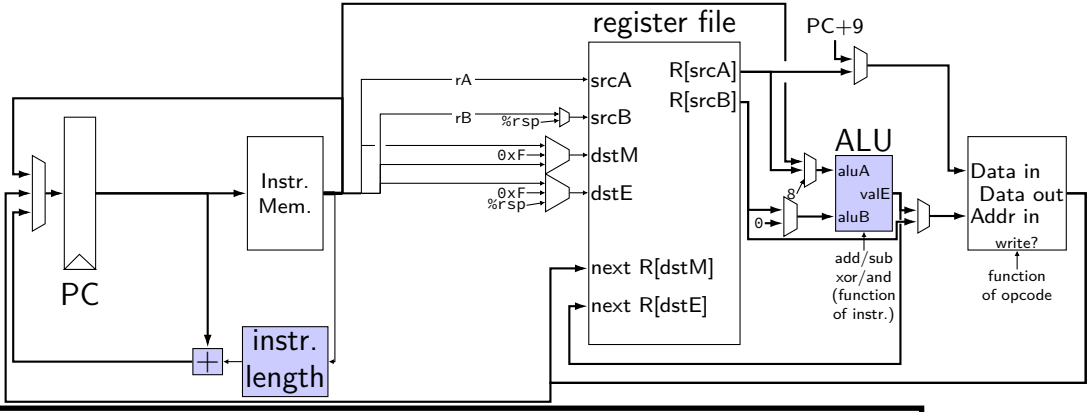
choose value for PC next cycle (input to PC register)

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

PC update


register file
PC+9
rA
rB
%rsp
0xF
0xF
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
write?
function
of opcode
Instr.
Mem.
PC
valP
instr.
length

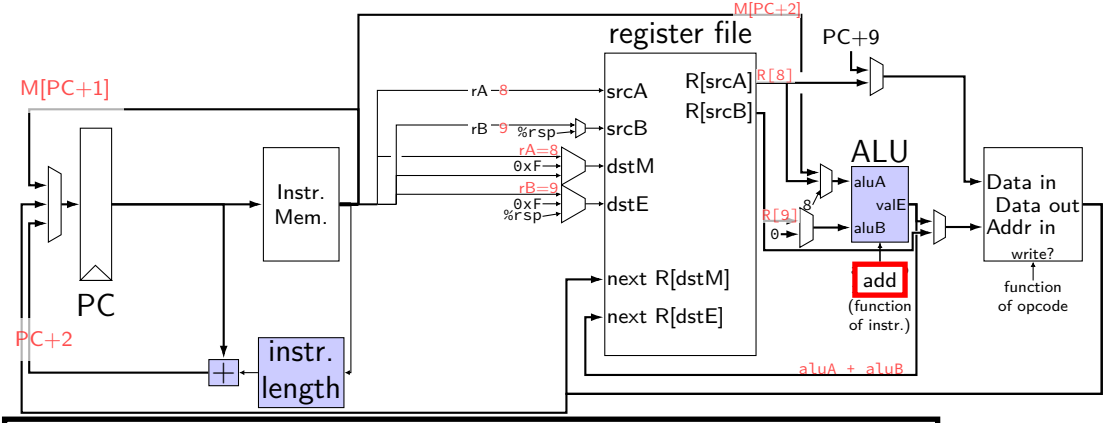
circuit: setting MUXes



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

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

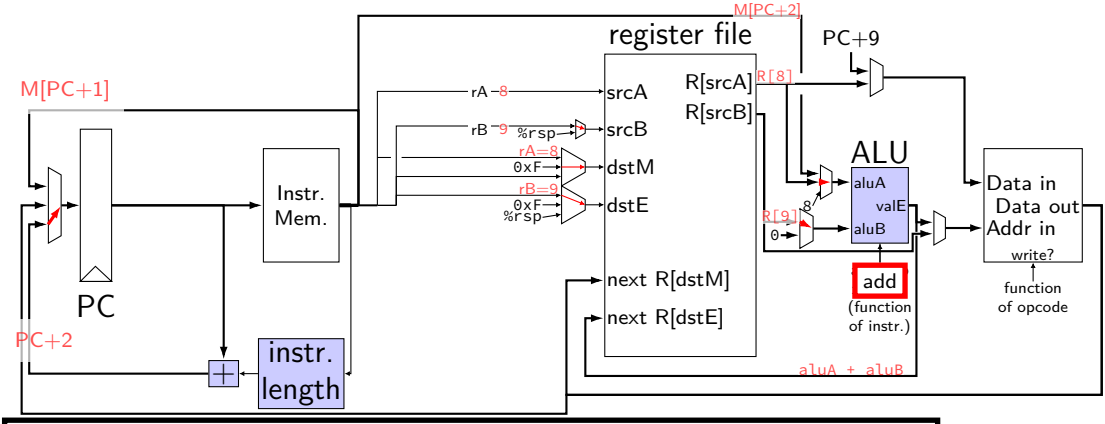
circuit: setting MUXes



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

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

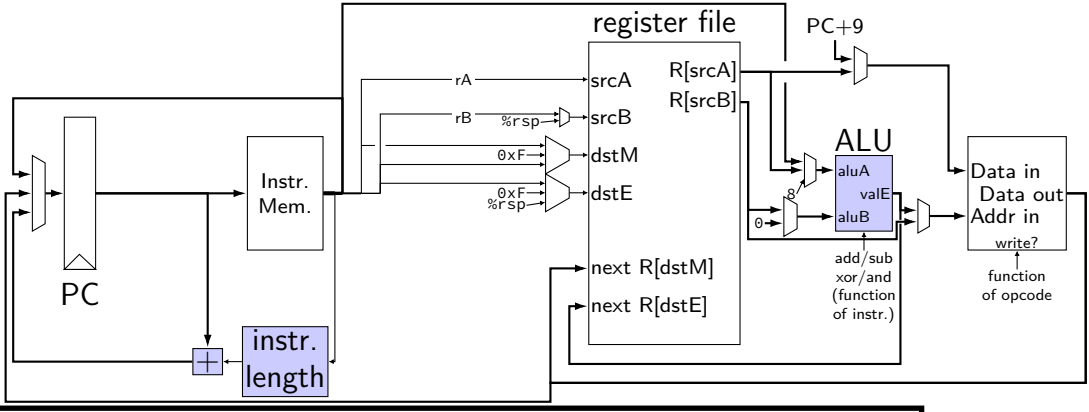
circuit: setting MUXes



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

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

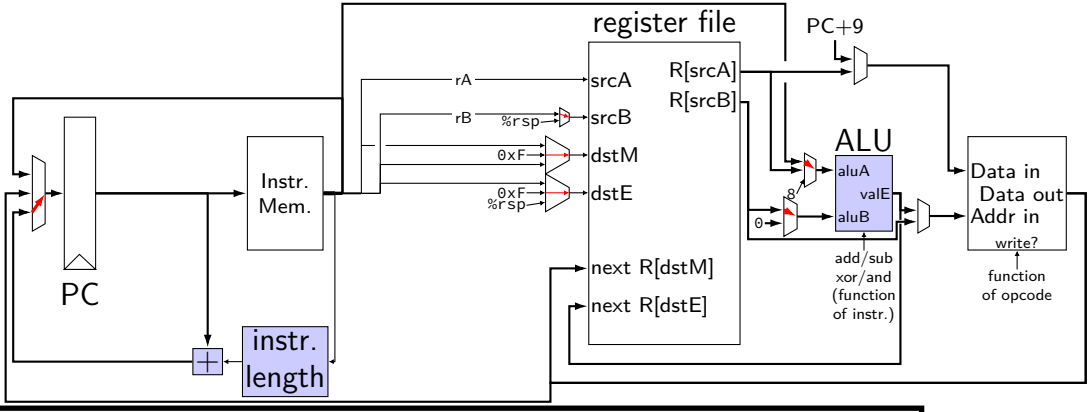
circuit: setting MUXes



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

Exercise: what do they select for `rmmovq`?

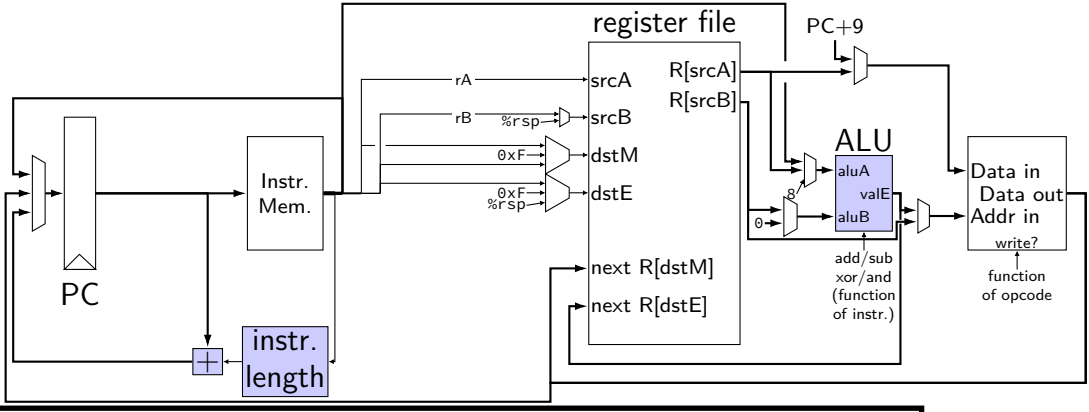
circuit: setting MUXes



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

Exercise: what do they select for `rmmovq`?

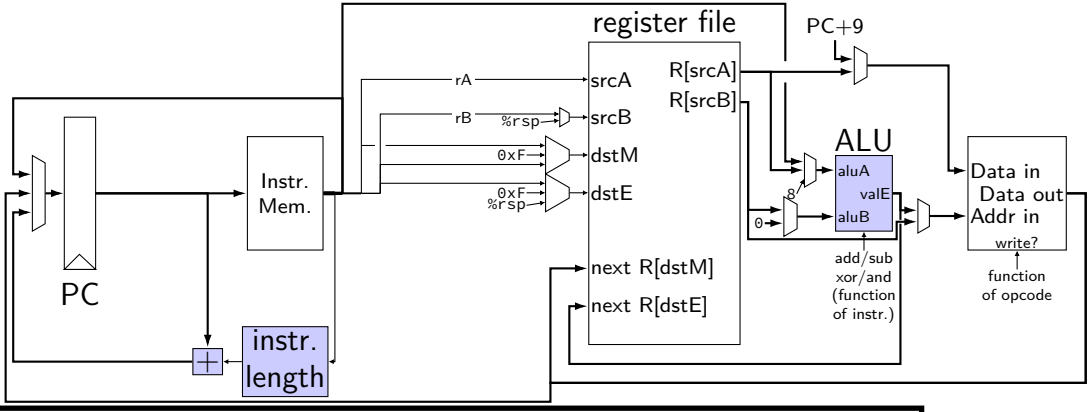
circuit: setting MUXes



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

Exercise: what do they select for `irmovq`?

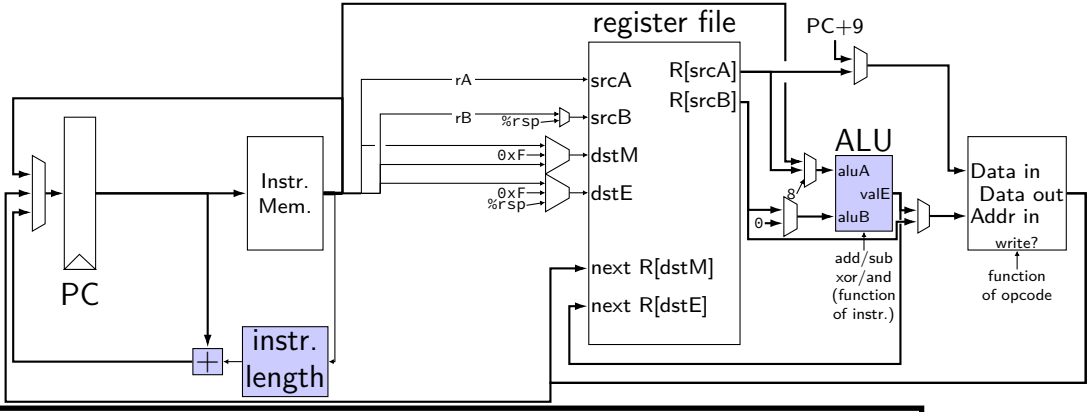
circuit: setting MUXes



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

Exercise: what do they select for `mrmovq`?

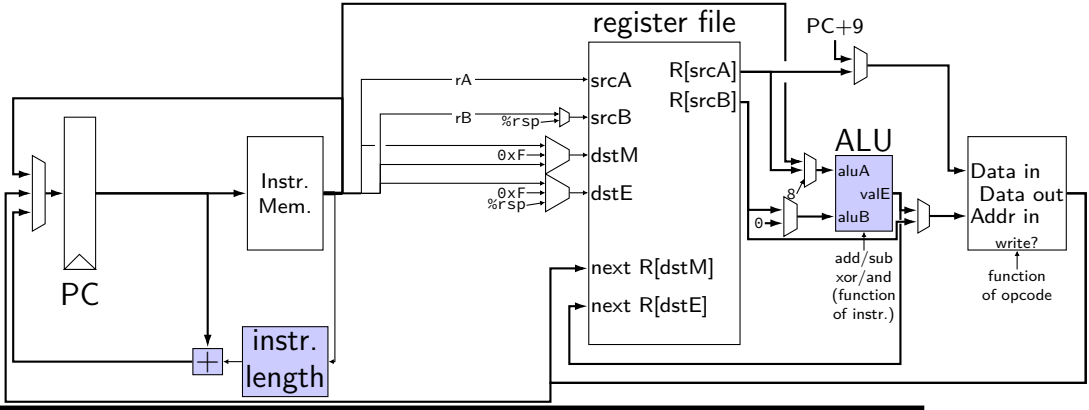
circuit: setting MUXes



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

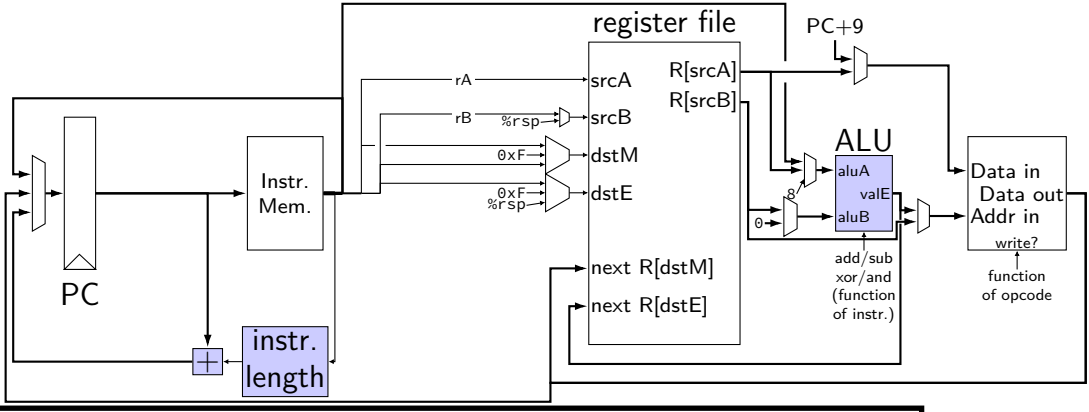
Exercise: what do they select for `jle`?

circuit: setting MUXes



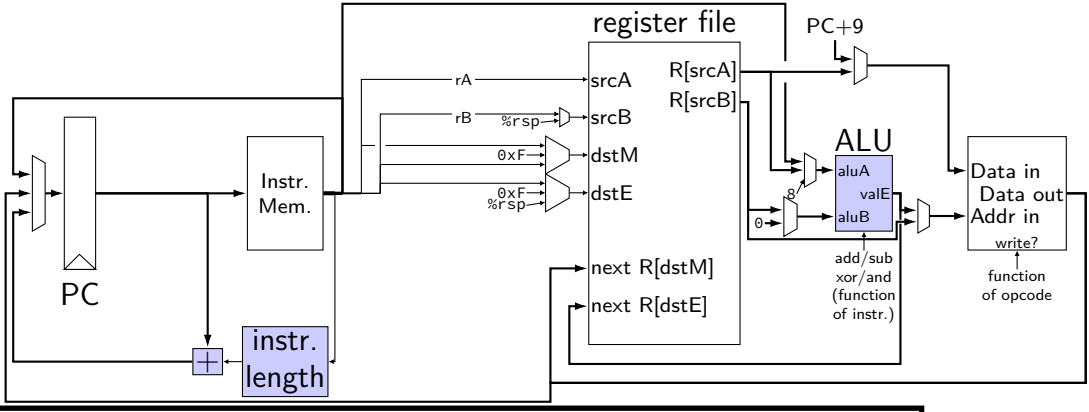
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `cmovle`?

circuit: setting MUXes



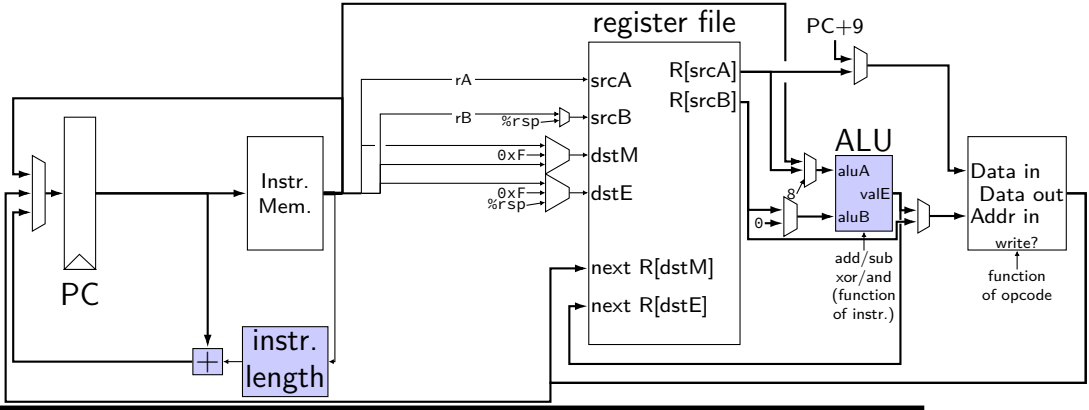
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `ret`?

circuit: setting MUXes



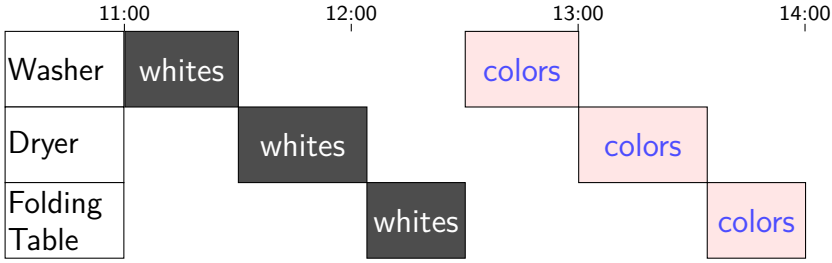
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **popq**?

circuit: setting MUXes

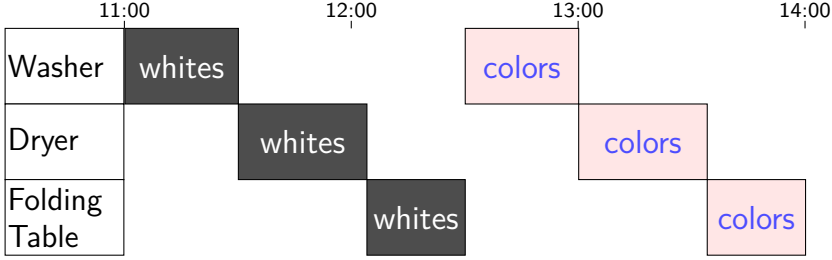


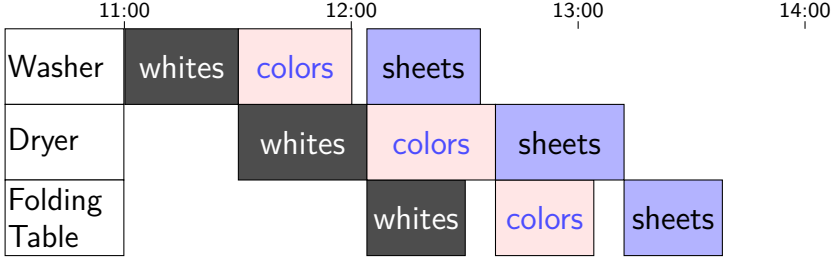
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `call`?

Human pipeline: laundry

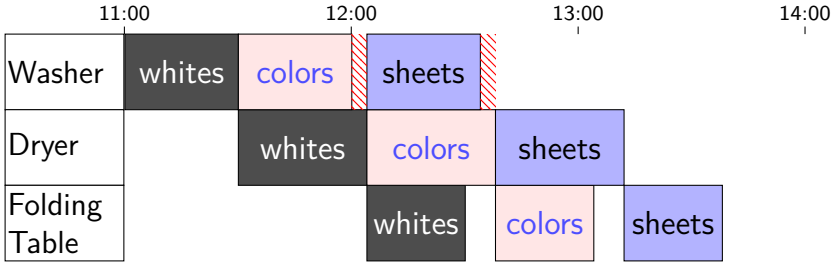


Human pipeline: laundry

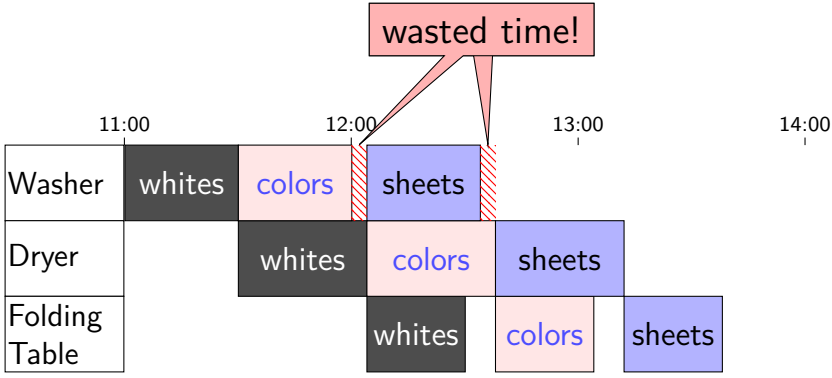




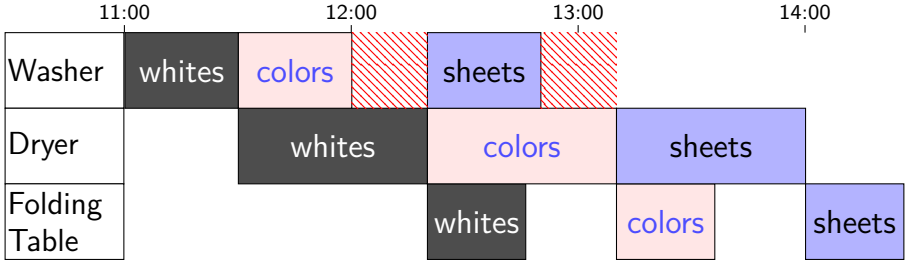
Waste (1)


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

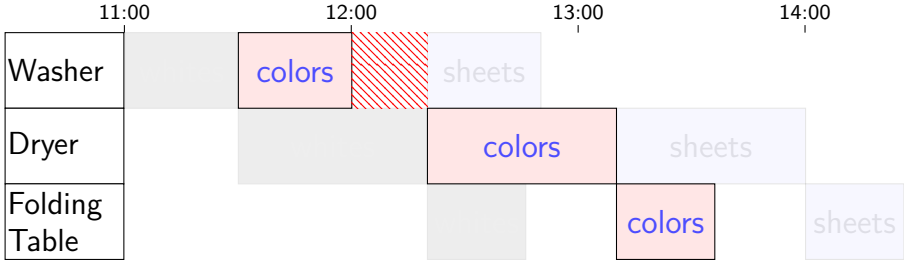
Waste (1)


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

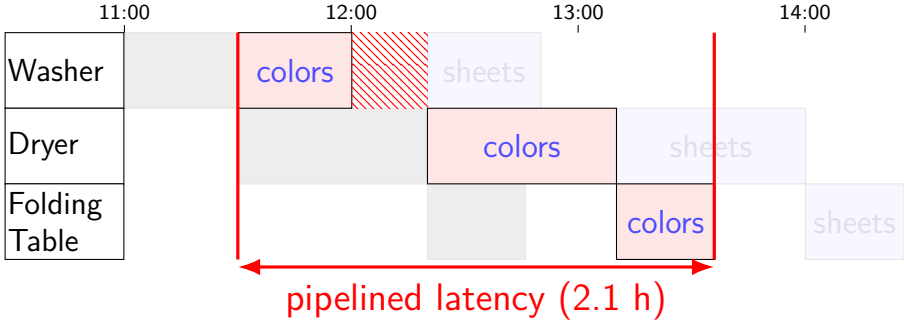
Waste (2)


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

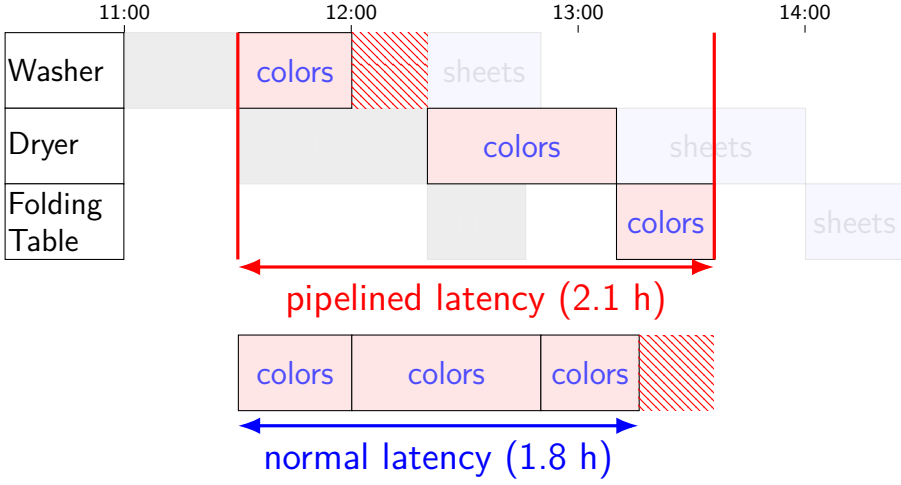
Latency — Time for One



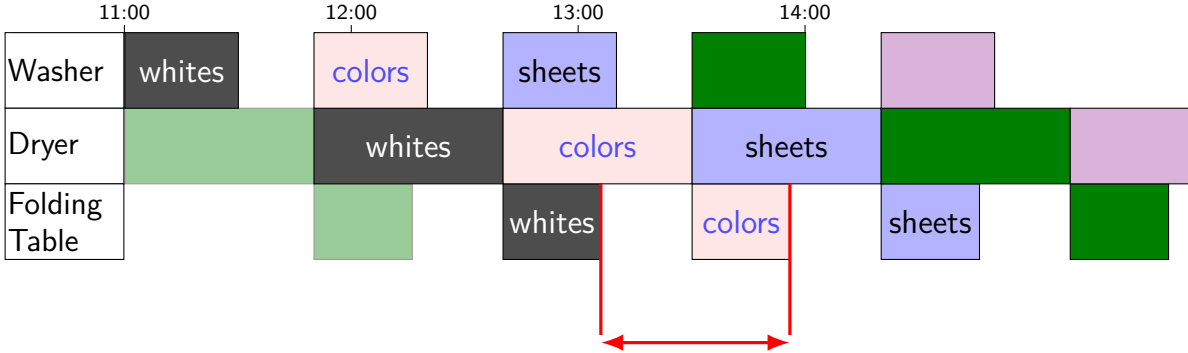
Latency — Time for One



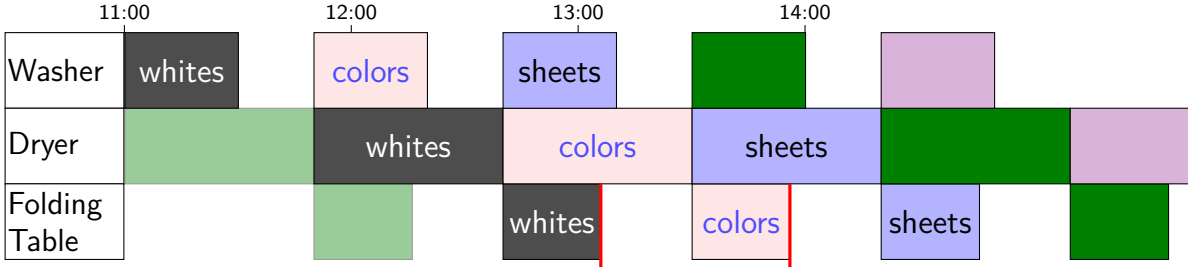
Latency — Time for One



Throughput — Rate of Many

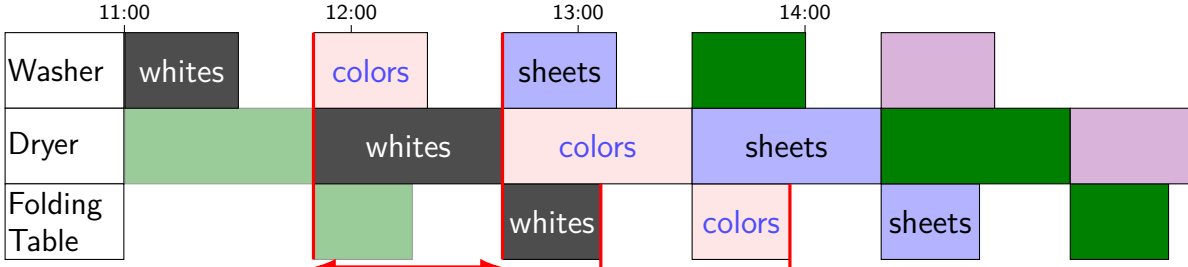


Throughput — Rate of Many



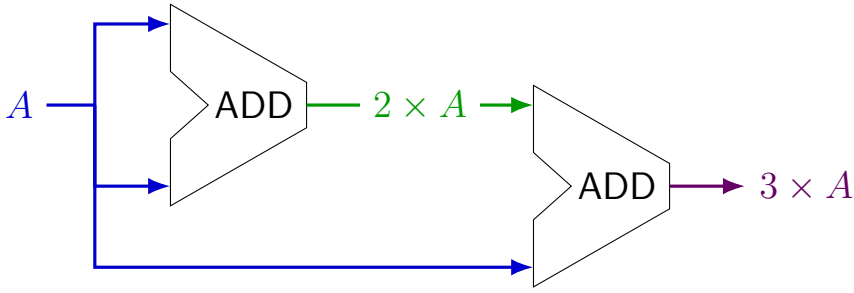
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many

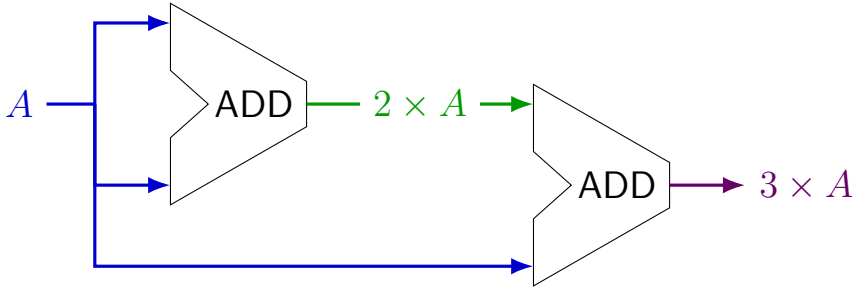


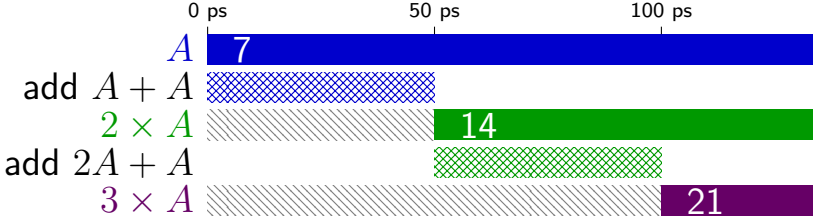
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

times three circuit

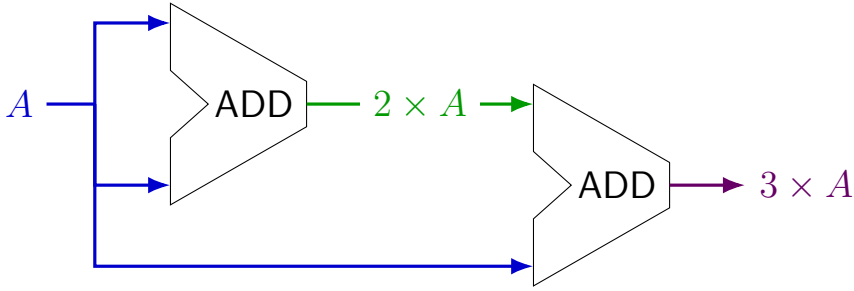


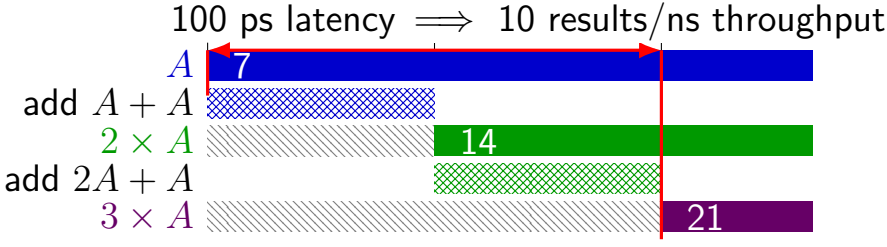
times three circuit



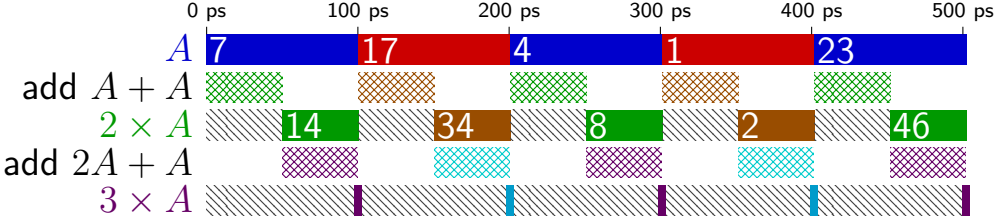


times three circuit

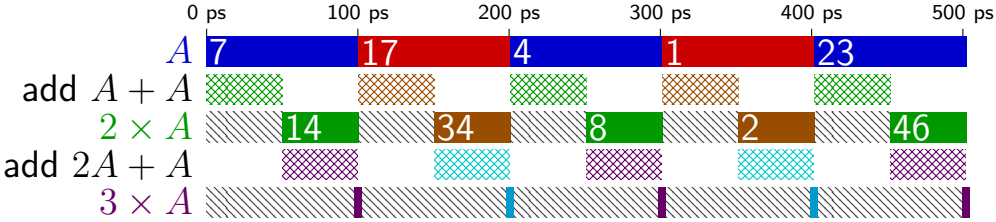


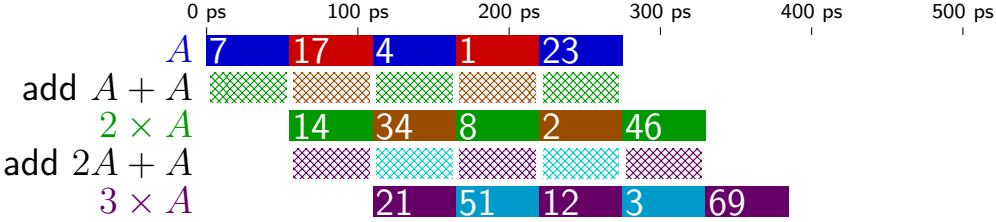


times three and repeat

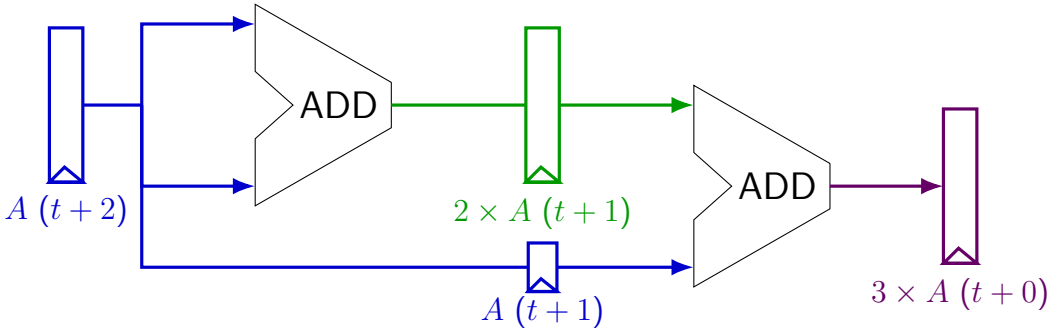


times three and repeat

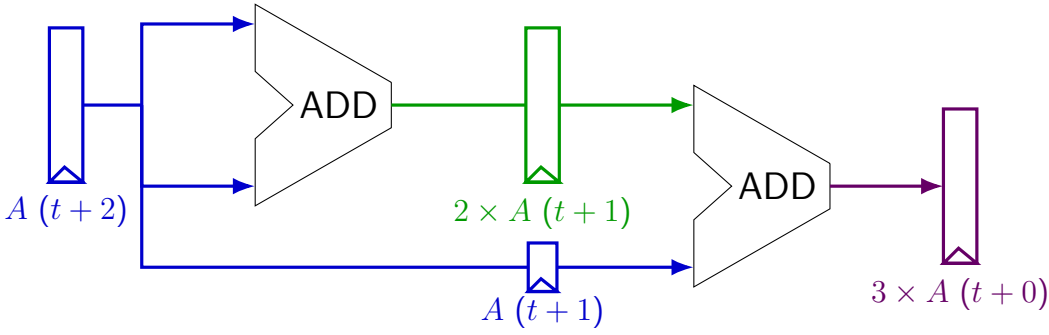


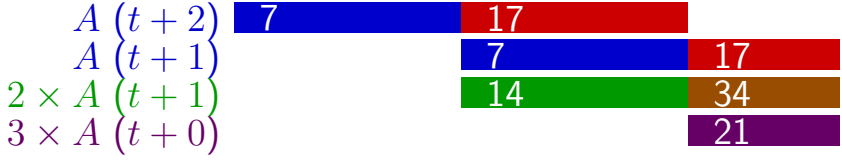


pipelined times three

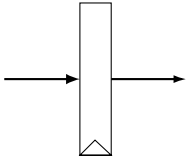


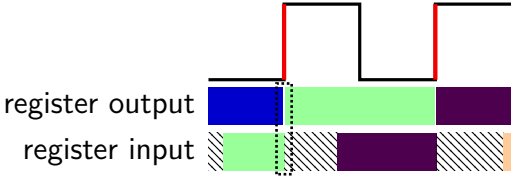
pipelined times three



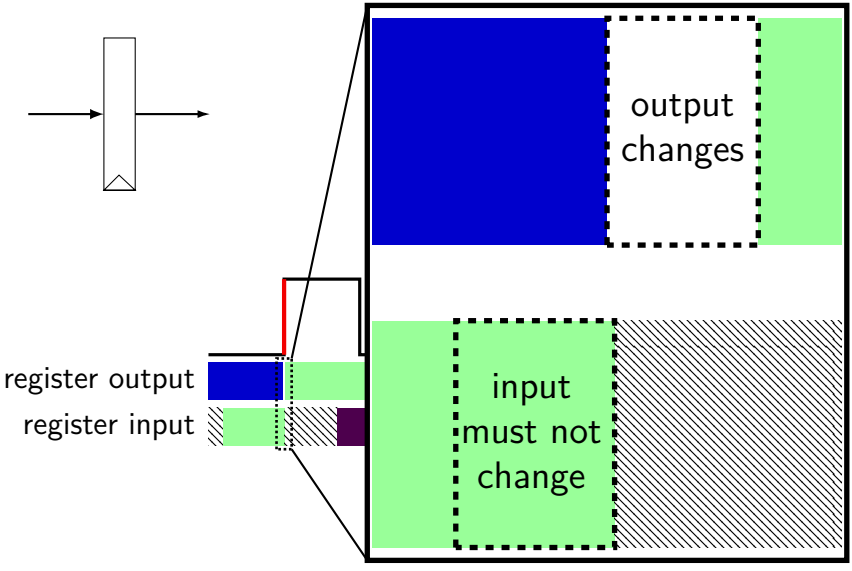


register tolerances

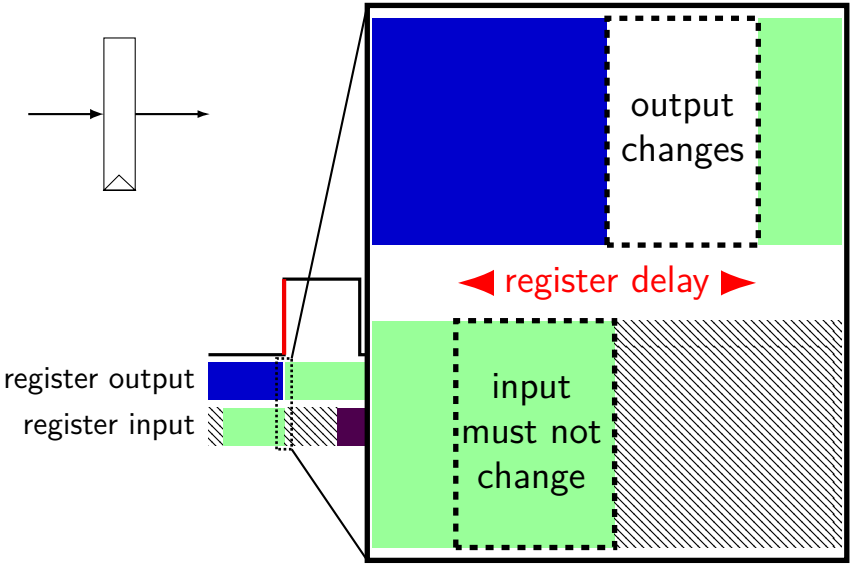




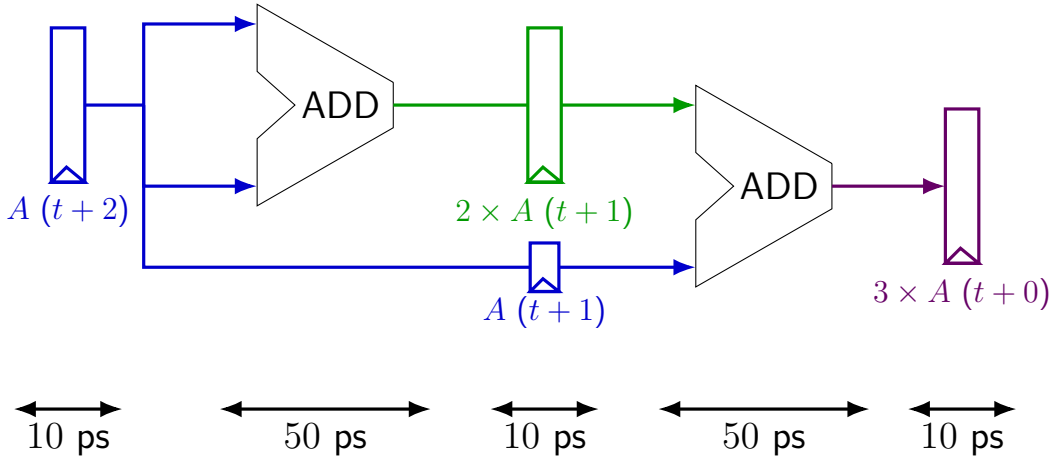
register tolerances



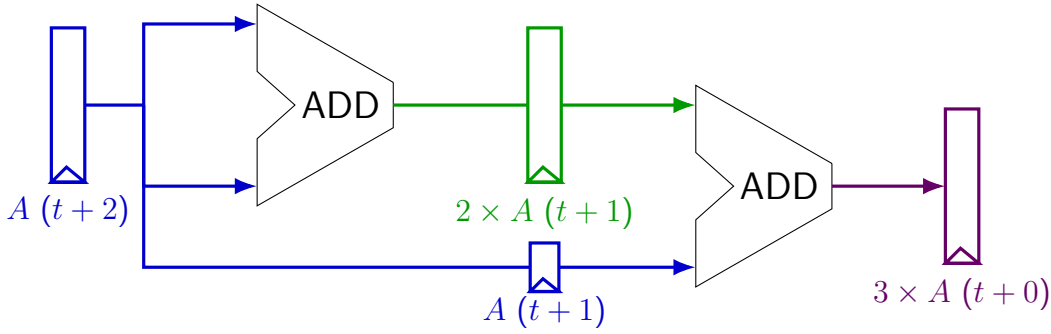
register tolerances



times three pipeline timing

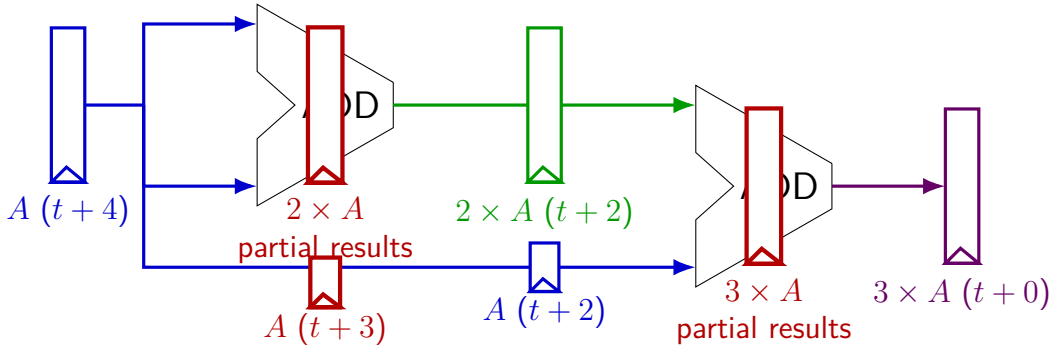


times three pipeline timing

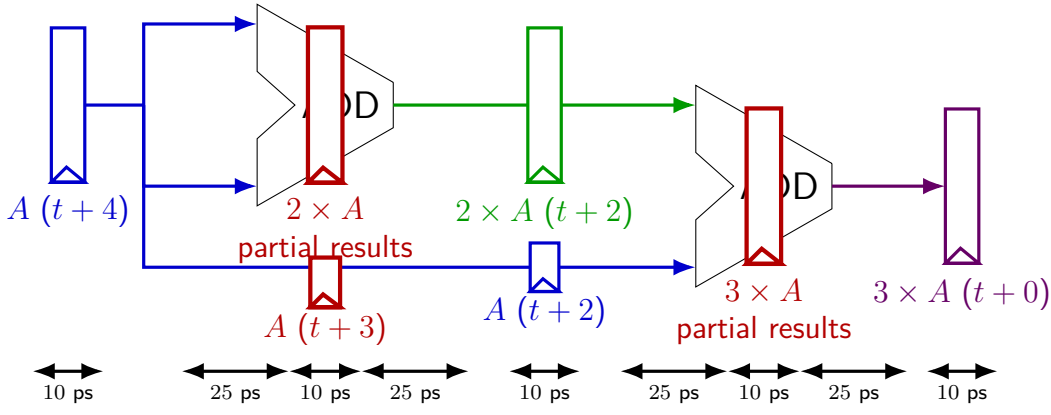


throughput: $\dfrac{1}{60\text{ ps}} \approx 16$ G operations/sec

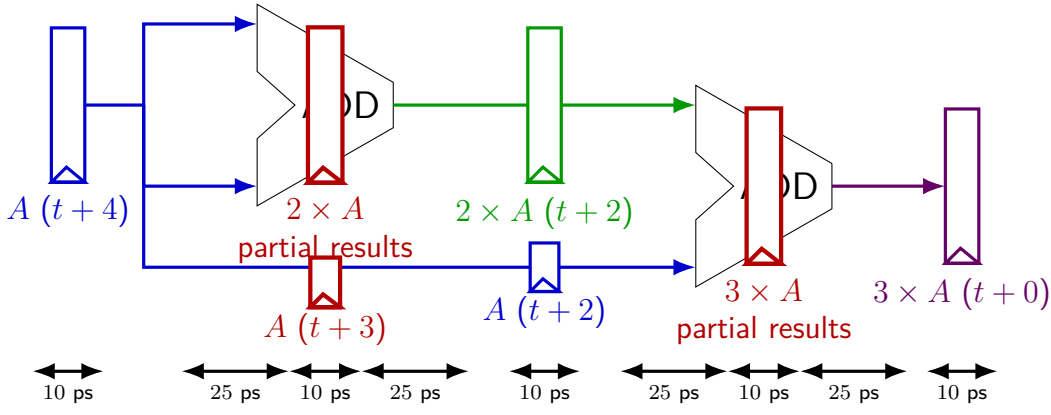
deeper pipeline



deeper pipeline

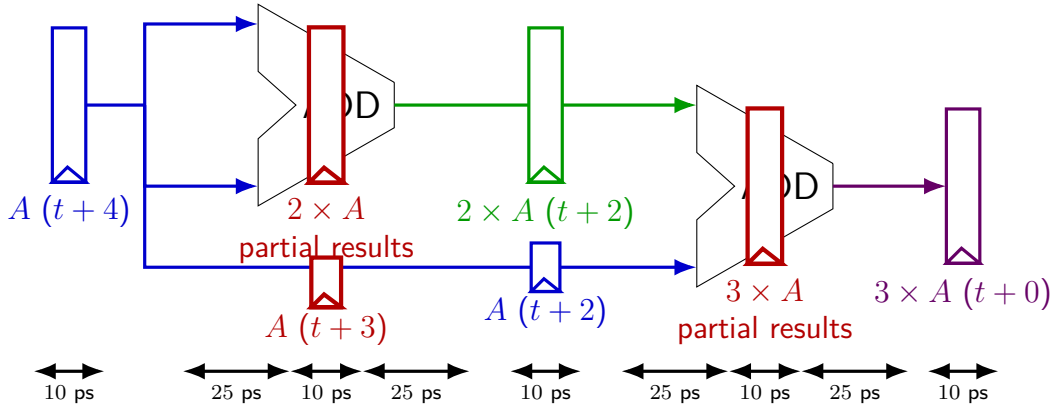


deeper pipeline



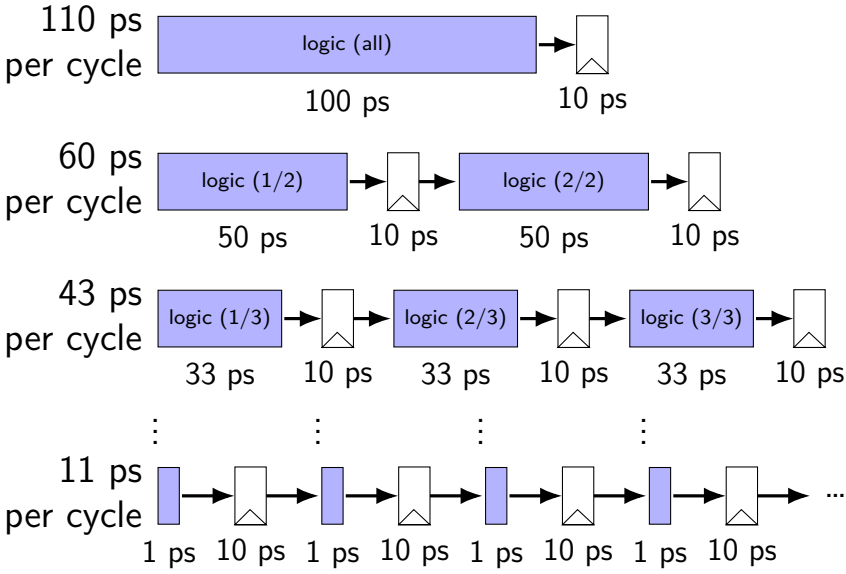
exercise: throughput now?

deeper pipeline

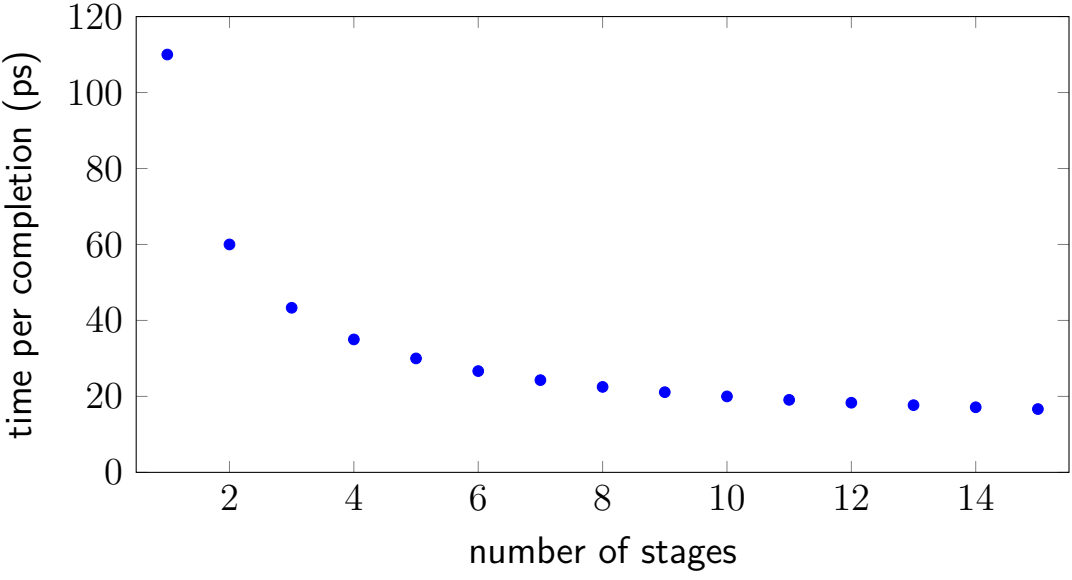


throughput: $\dfrac{1}{35\ \text{ps}} \approx 28\ \text{G ops/sec}$

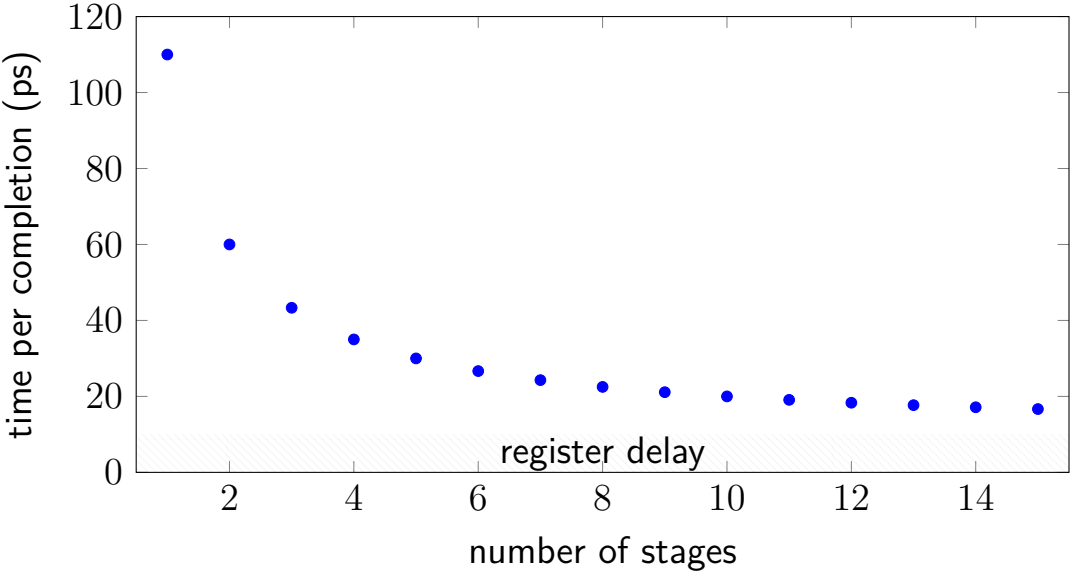
diminishing returns: register delays



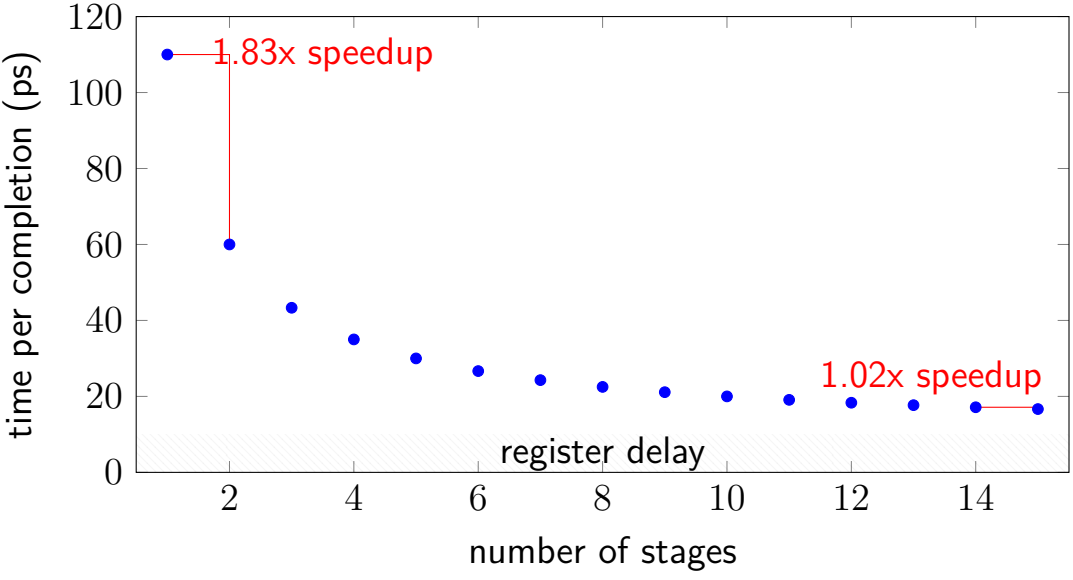
diminishing returns: register delays



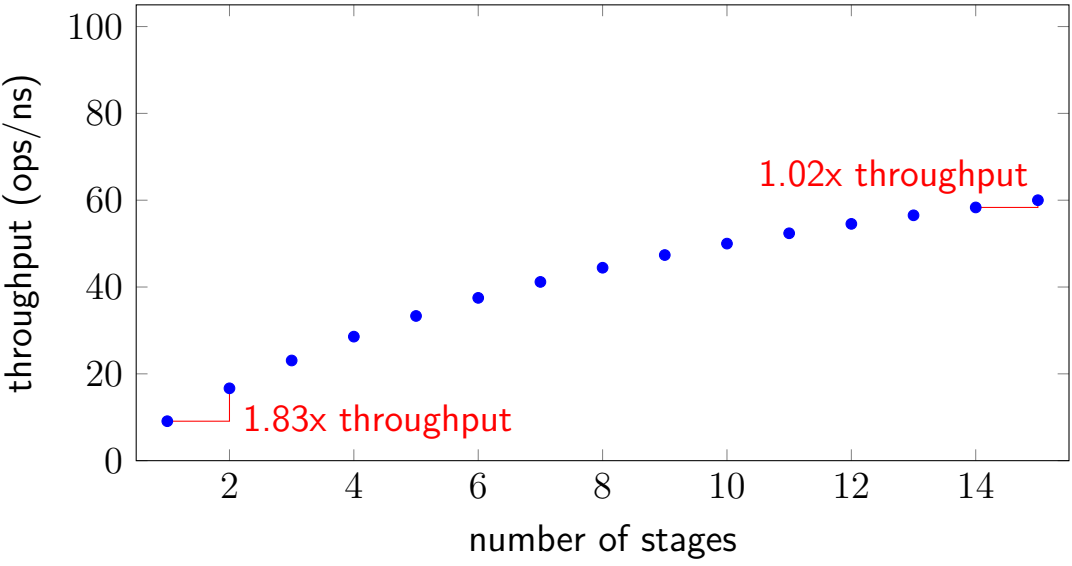
diminishing returns: register delays



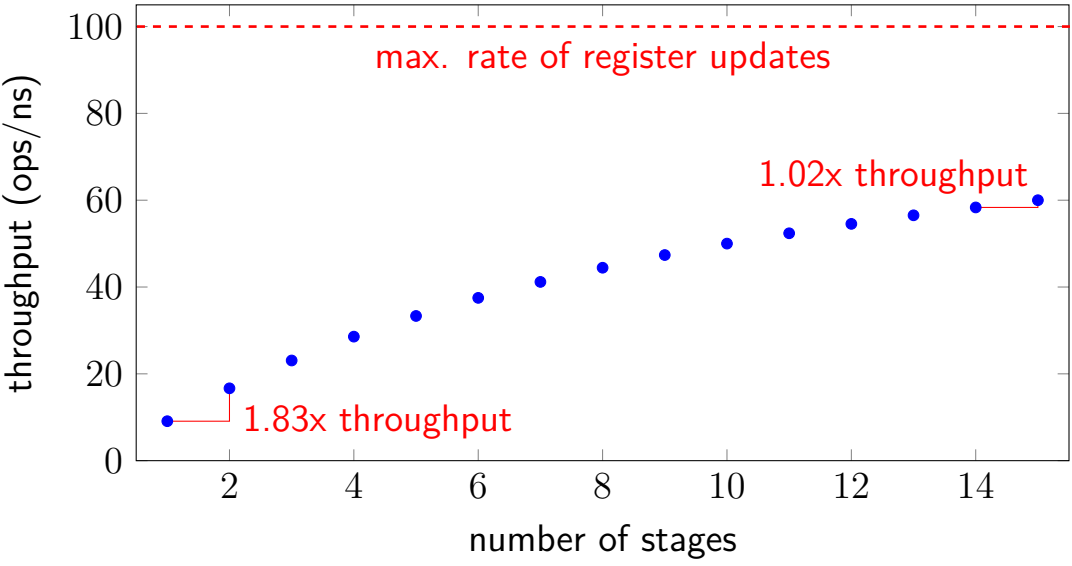
diminishing returns: register delays



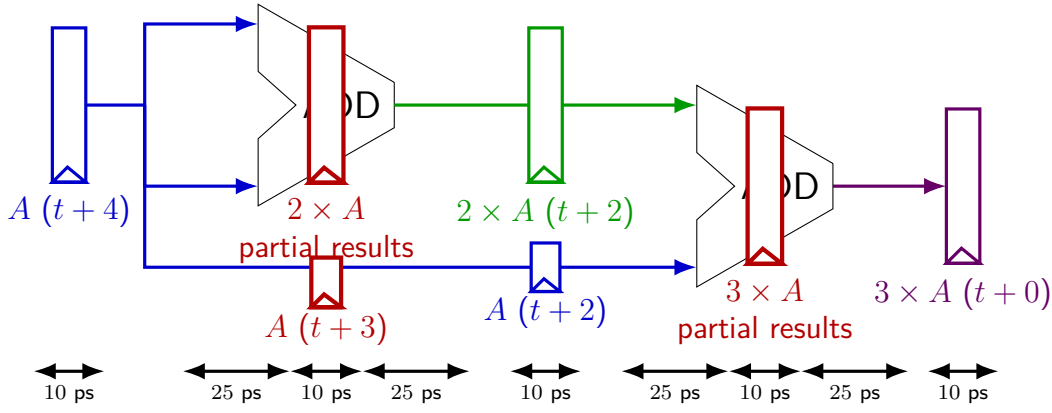
diminishing returns: register delays



diminishing returns: register delays



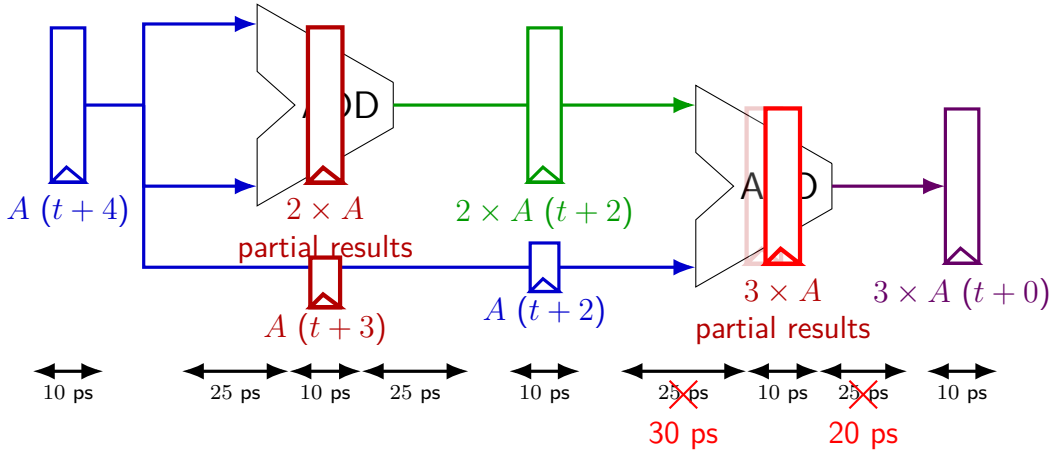
deeper pipeline



Problem: How much faster can we get?

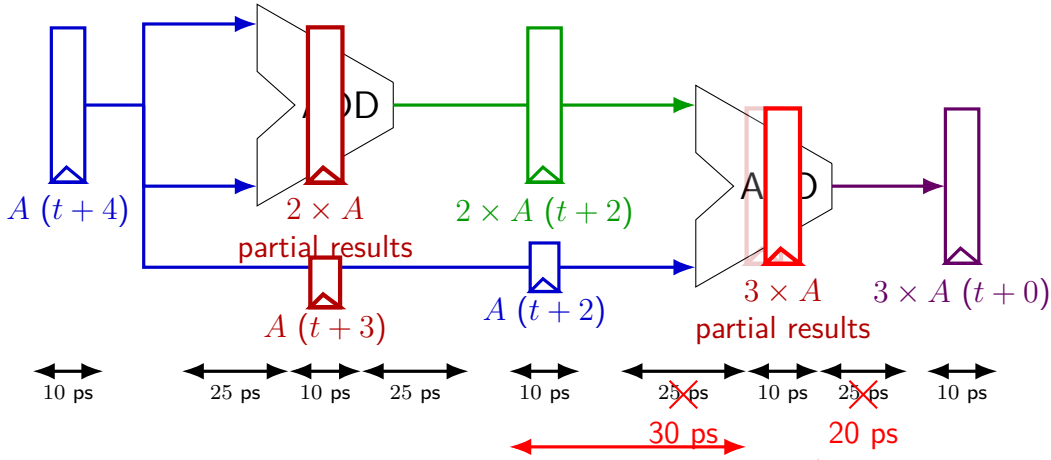
Problem: Can we even do this?

deeper pipeline



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

deeper pipeline

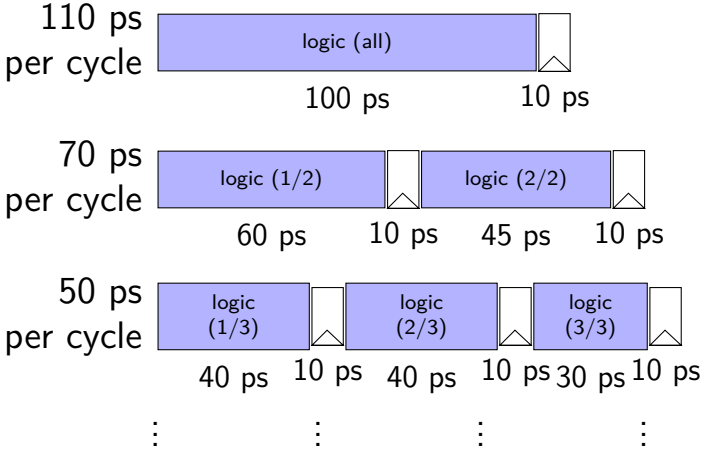


throughput: $\dfrac{1}{40\ \text{ps}} \approx 25$ G ops/sec

diminishing returns: uneven split

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

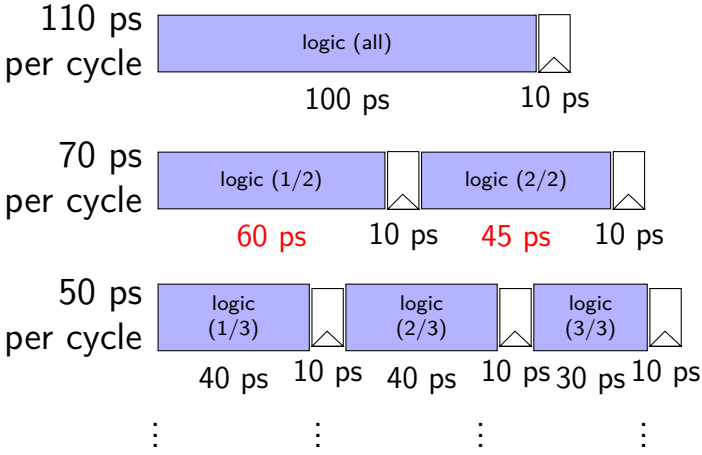
Probably not...



diminishing returns: uneven split

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

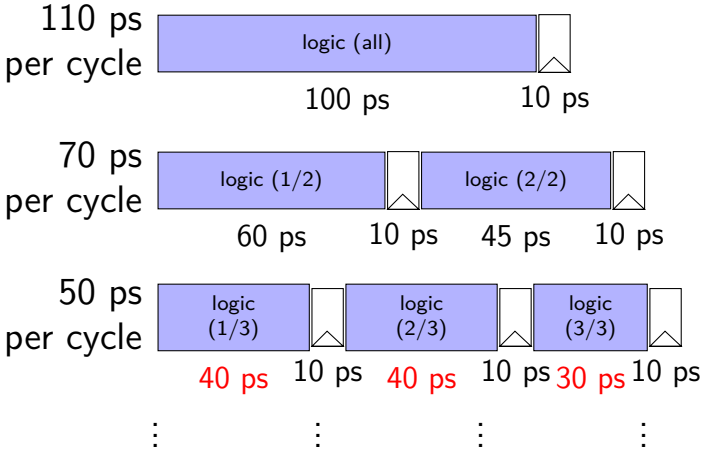
Probably not...



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

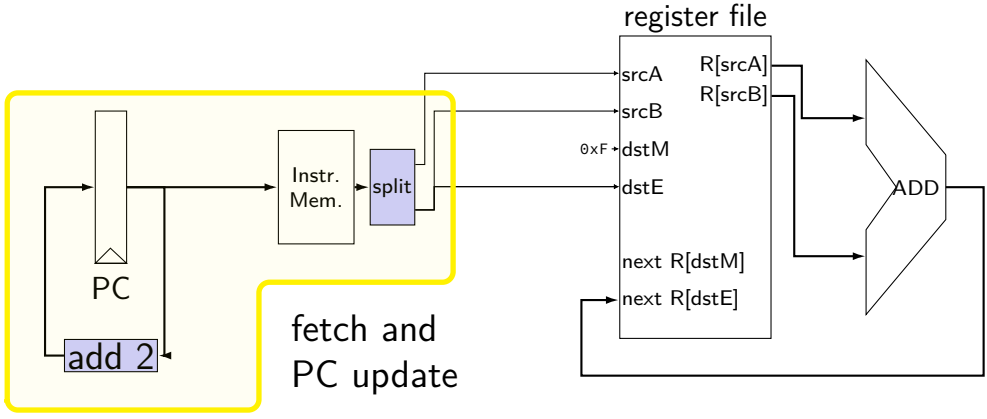
Execute: arithmetic (ALU)

Memory: read/write data memory

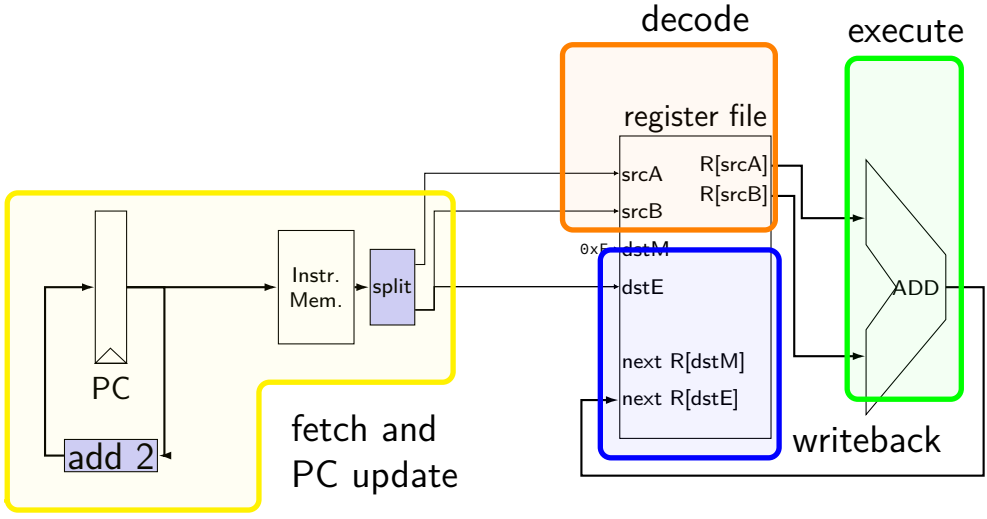
Writeback: write register file

5 stages
one instruction in each
compute next to start immediatelly

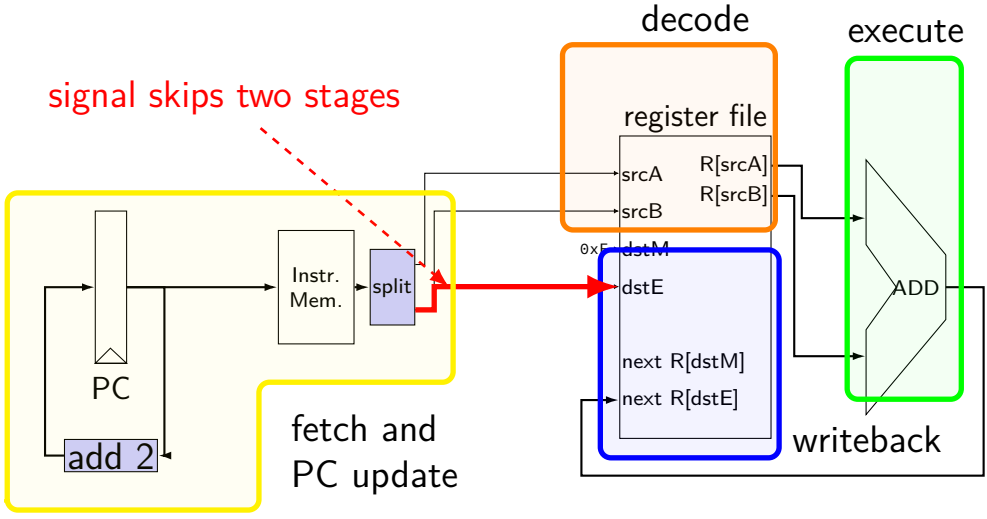
addq CPU


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

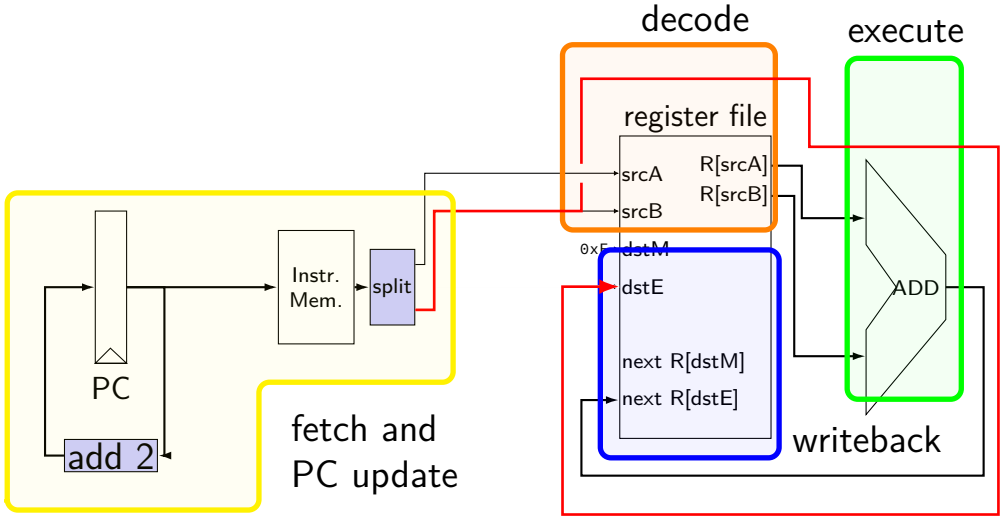
addq CPU


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

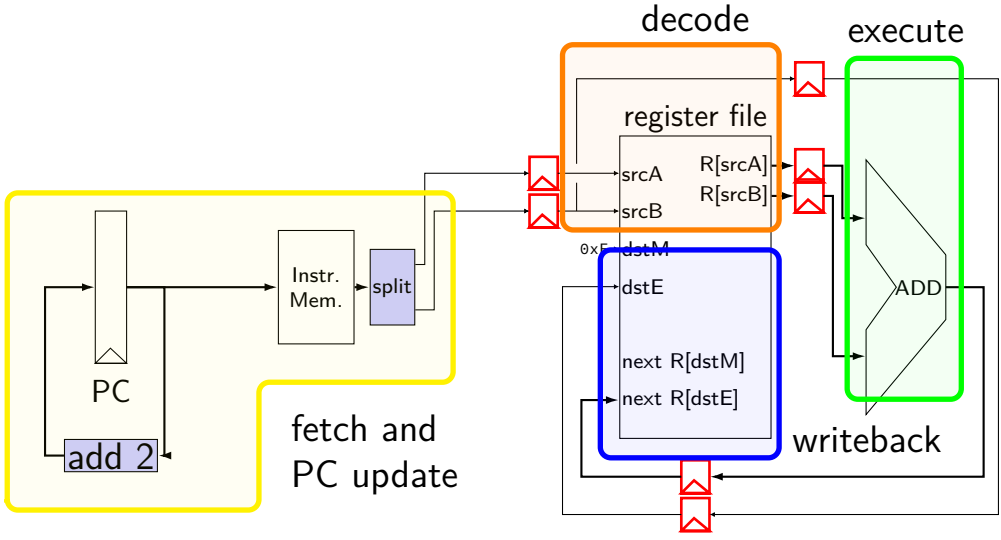
addq CPU


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

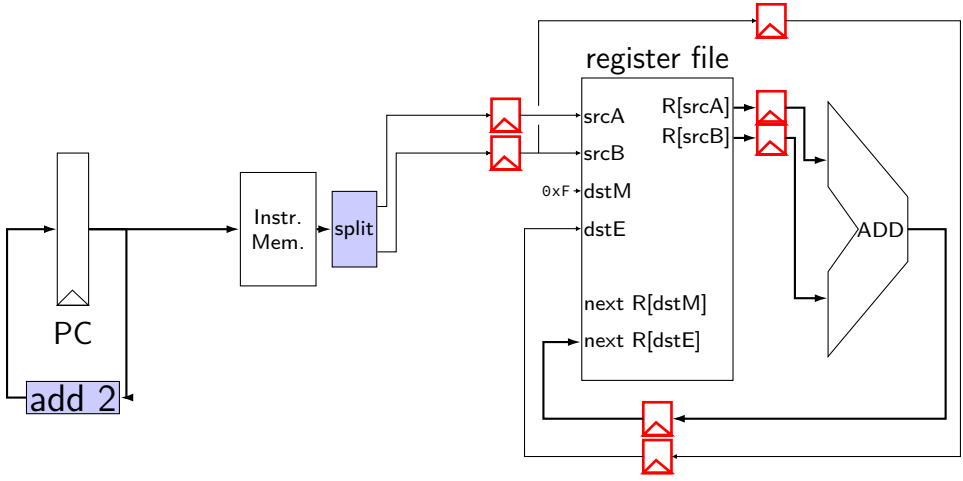
addq CPU


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

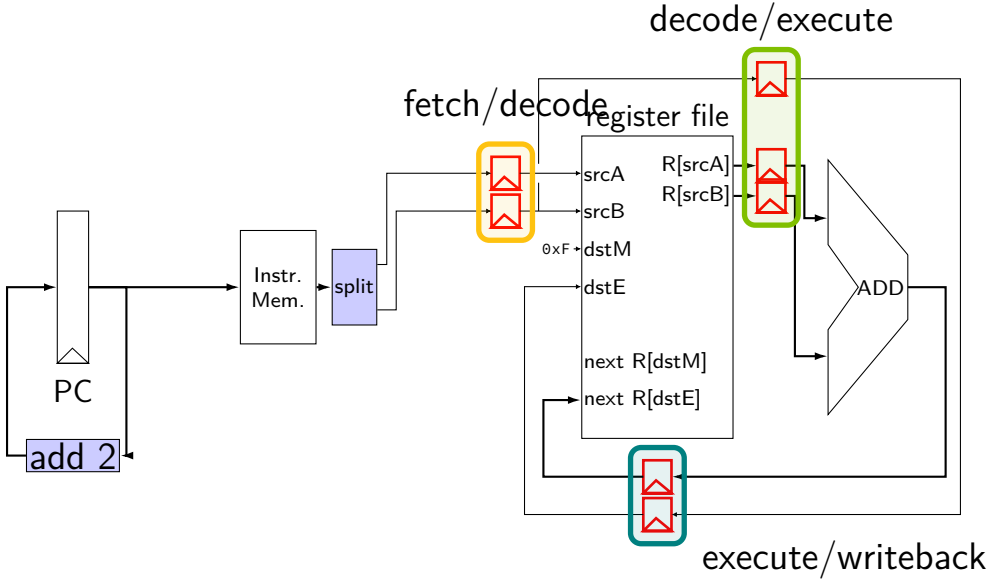
pipelined addq processor



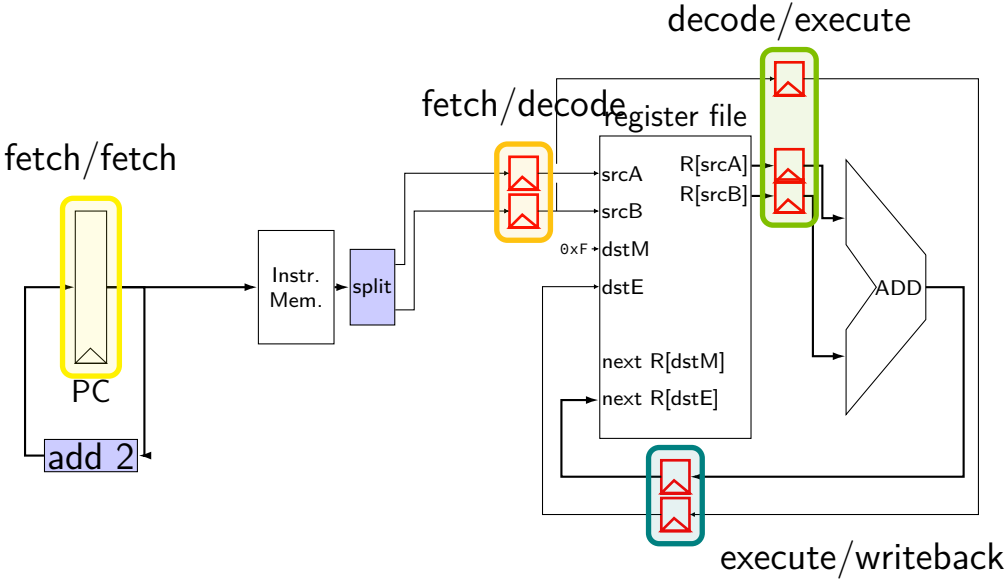
pipelined addq processor



pipelined addq processor

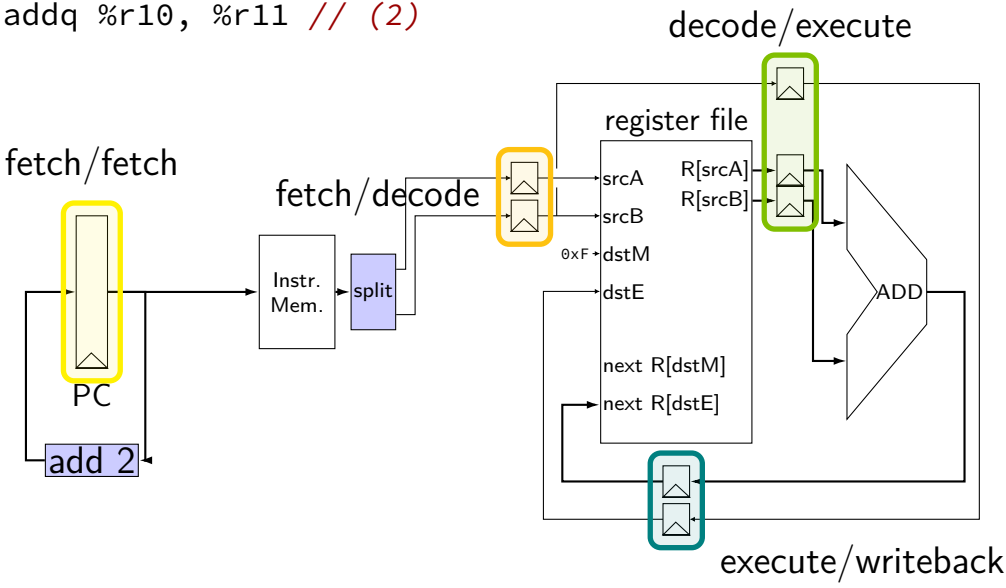


pipelined addq processor



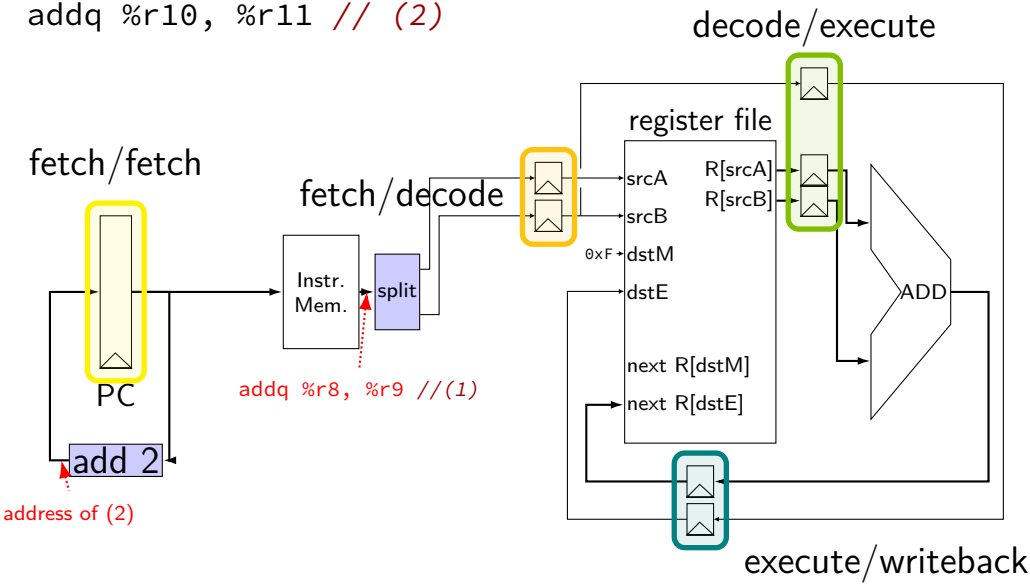
addq execution

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



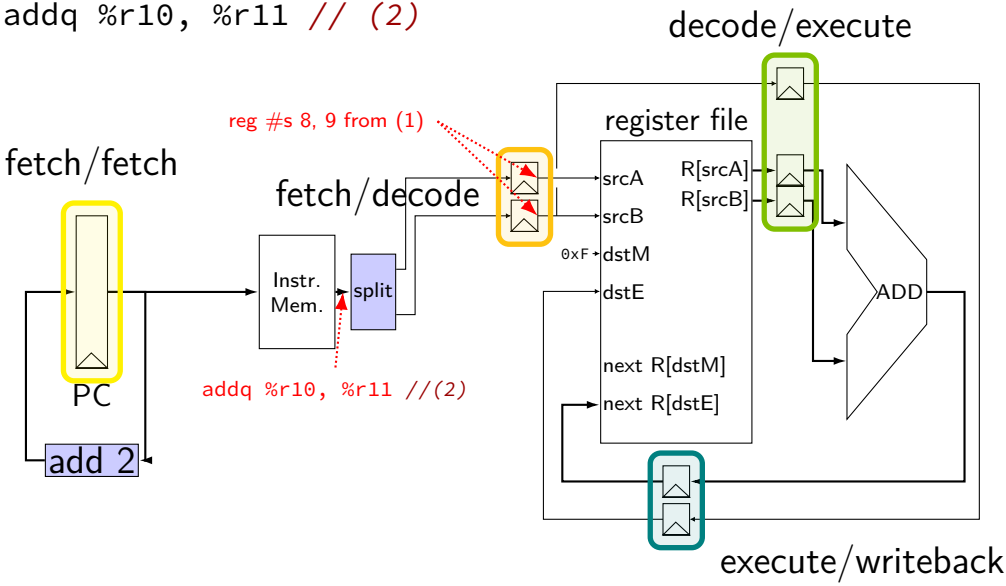
addq execution

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



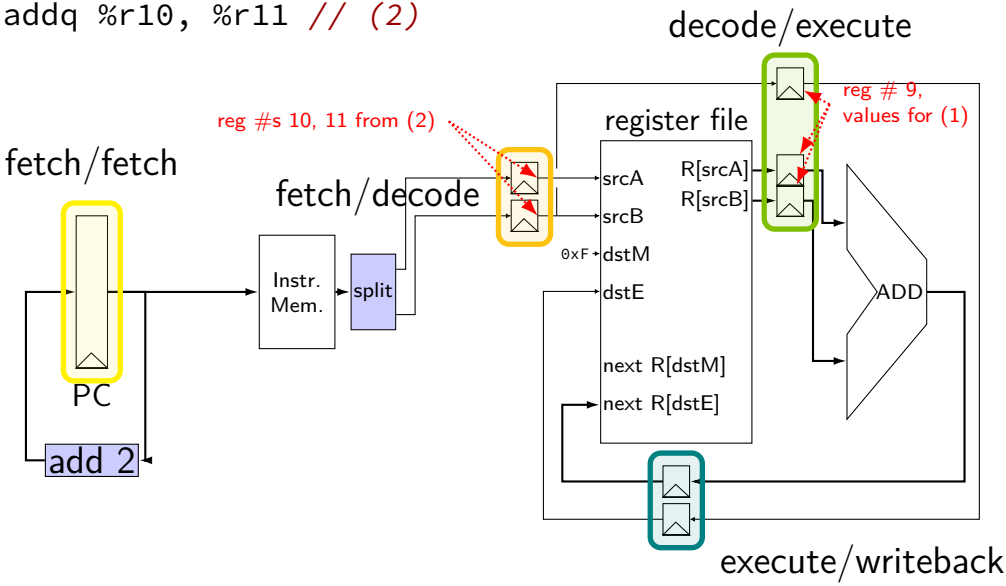
addq execution

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



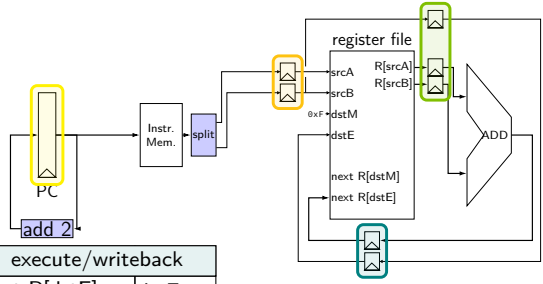
addq execution

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



addq processor timing

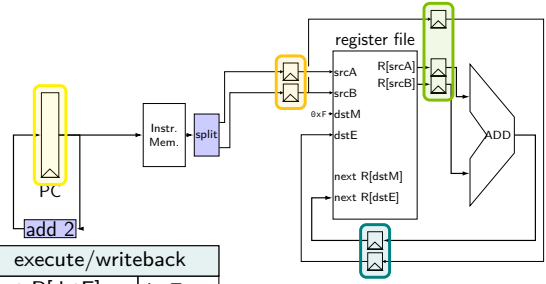
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

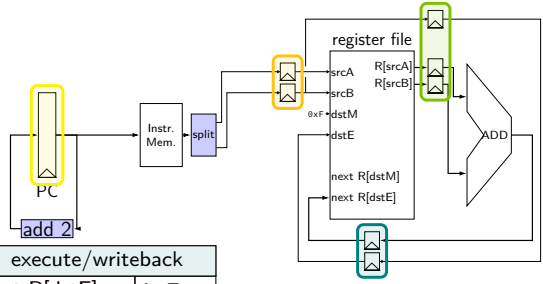
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

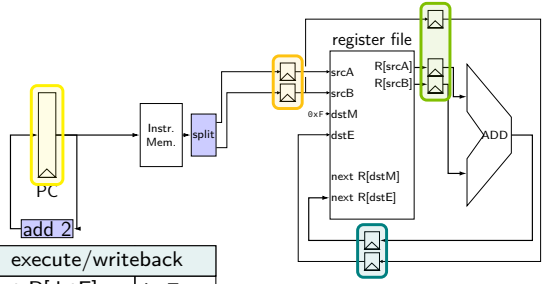
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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

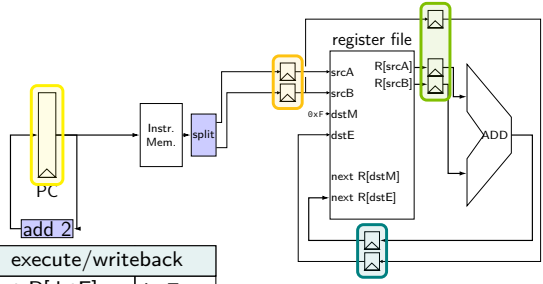
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
addq %r10, %r11
addq %r12, %r13
addq %r9, %r8
```



	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