

pipelining

changelog

Changes since first lecture:

4 October 2021: correct typo of “= E” for “= 0” in pipeline register declaration examples

last time

final stages

popq, ret: read from memory at current `%rsp`, compute new `%rsp`
textbook's use of `R[rA]+0` for `rrmovq`, `D+0` for `irmovq`
writeback: special case (again) for `%rsp`

pipelining idea

when finished first part of instruction A, ...
do second part of A and first part of B at same time

pipeline performance determined by slowest stage

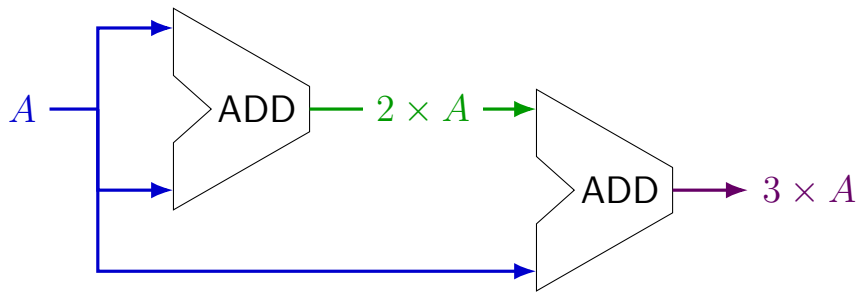
stages before: wait for slow stage to be ready
stages after: wait for slow stage to finish

latency — time from start to finish

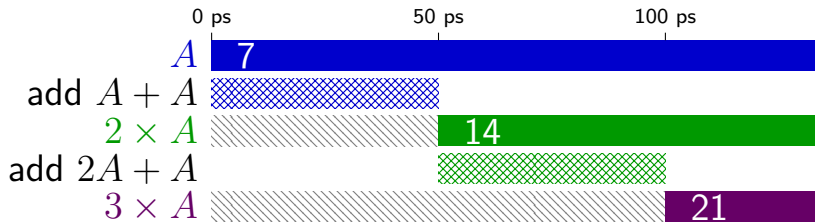
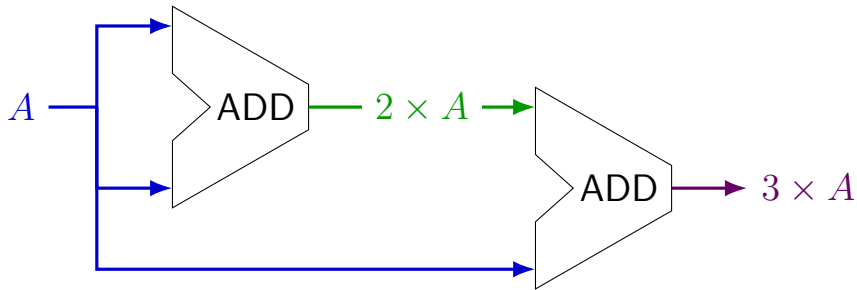
throughput — rate of completing many operations

can measure time between starting operations
or time between finishing

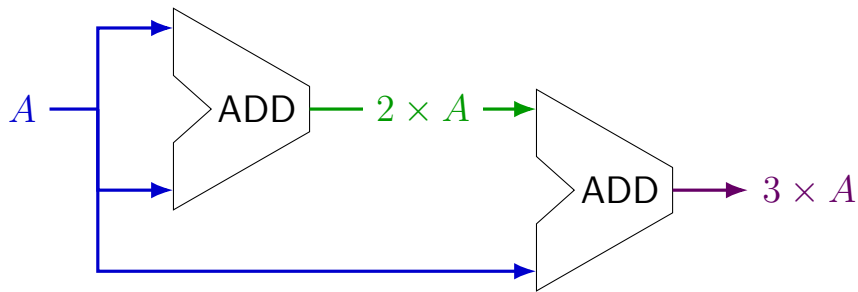
times three circuit



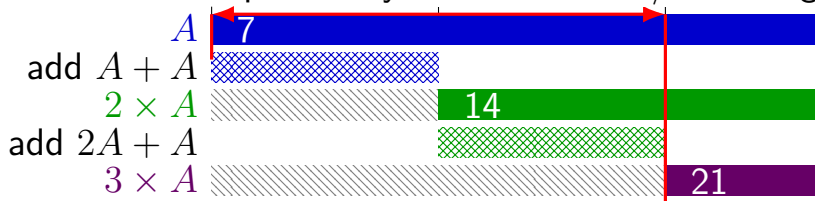
times three circuit



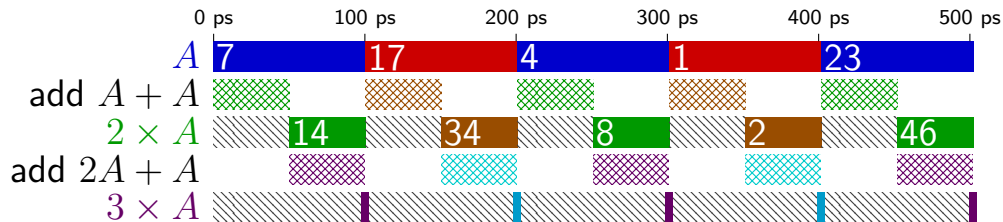
times three circuit



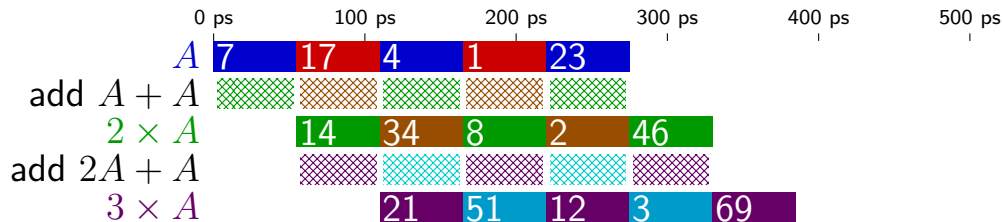
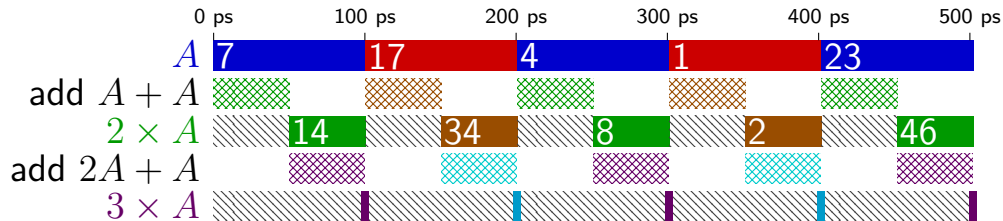
100 ps latency \Rightarrow 10 results/ns throughput



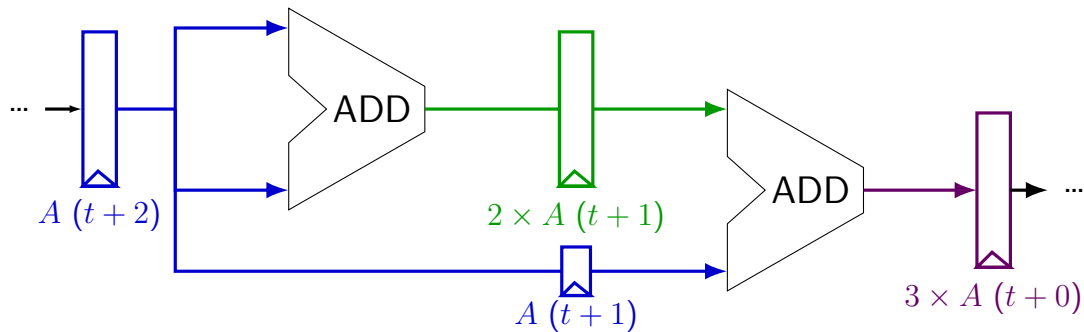
times three and repeat



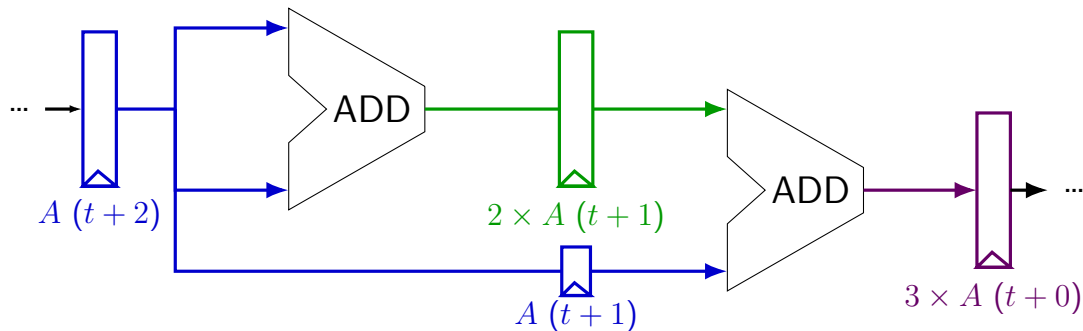
times three and repeat



pipelined times three

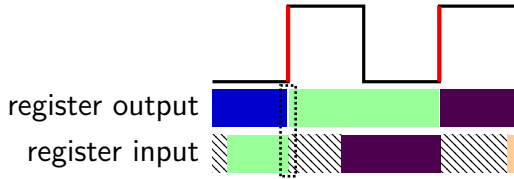
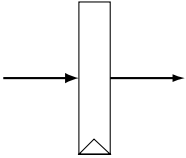


pipelined times three

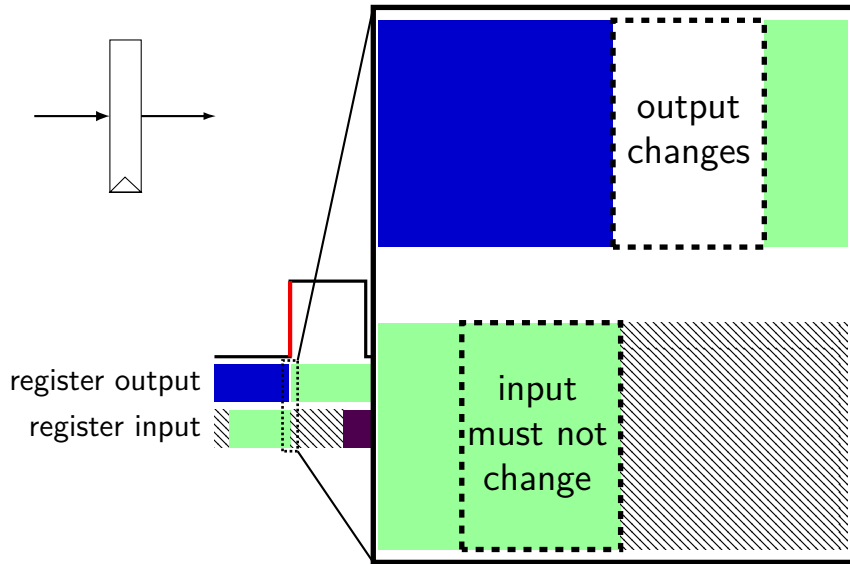


$A(t+2)$	7	17
$A(t+1)$	7	17
$2 \times A(t+1)$	14	34
$3 \times A(t+0)$		21

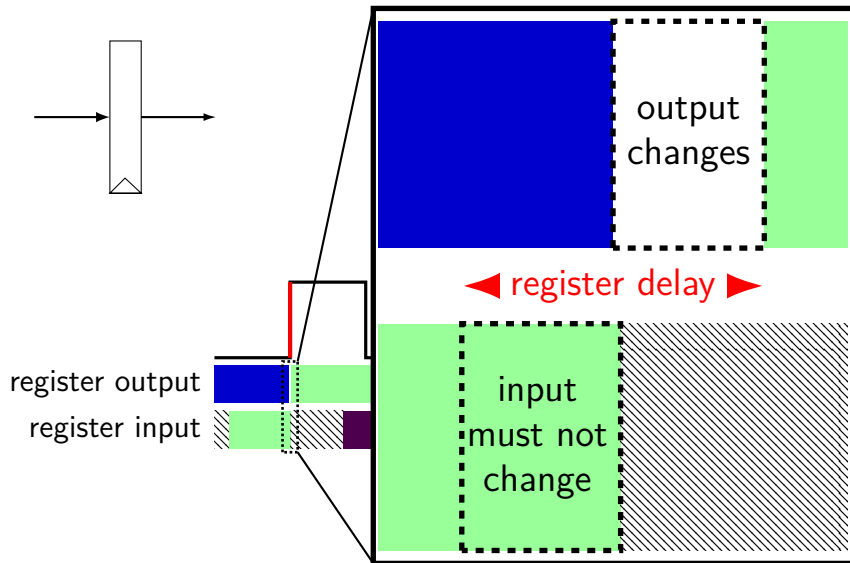
register tolerances



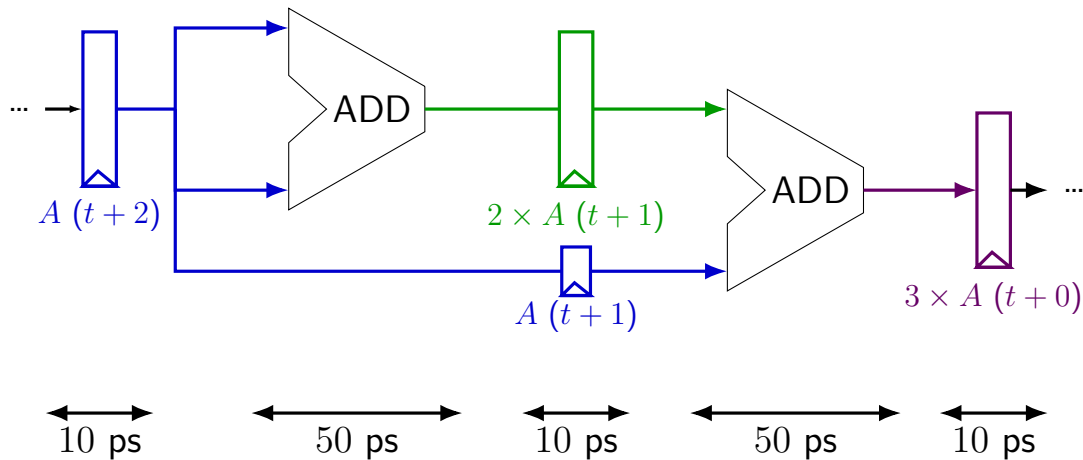
register tolerances



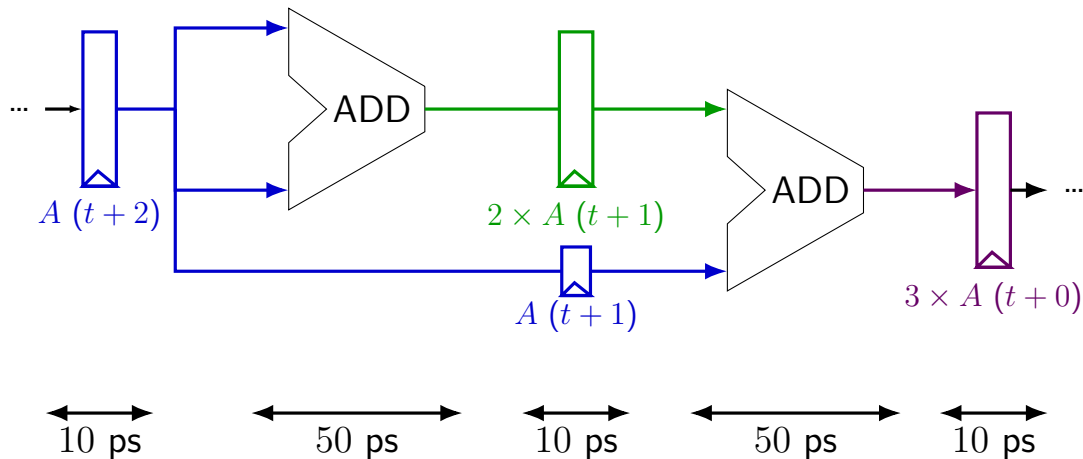
register tolerances



times three pipeline timing



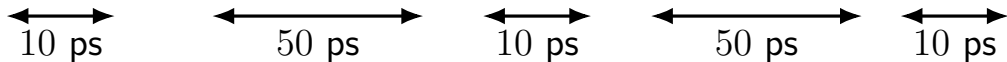
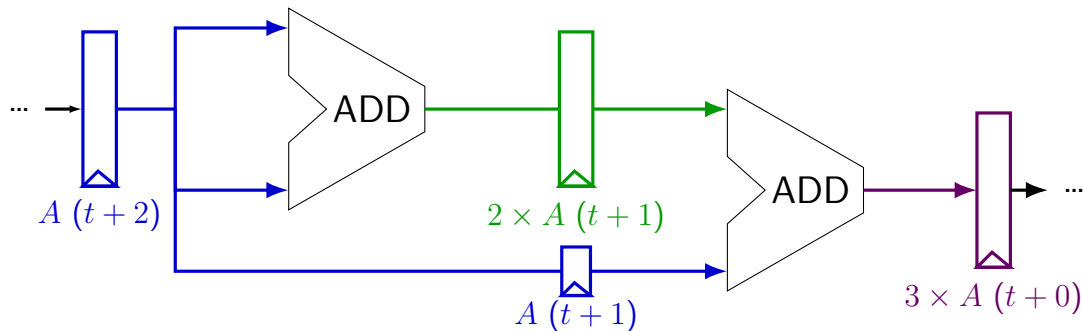
times three pipeline timing



exercise: minimum clock cycle time:

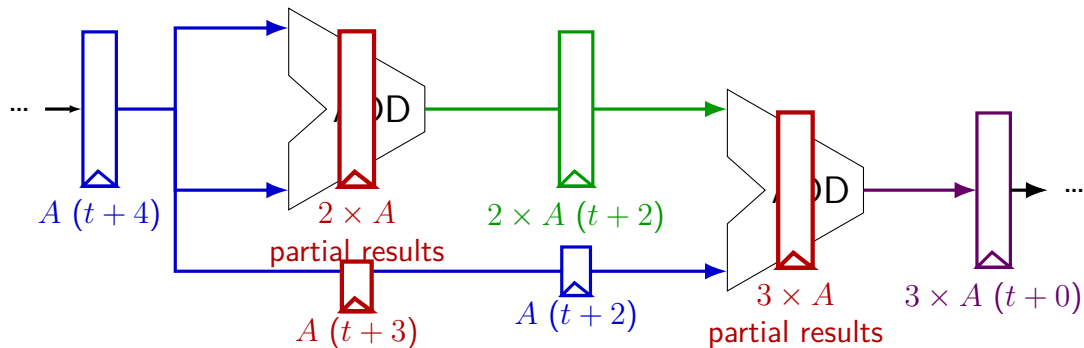
- A. 50 ps B. 60 ps B. 65 ps C. 70 ps E. 130 ps

times three pipeline timing

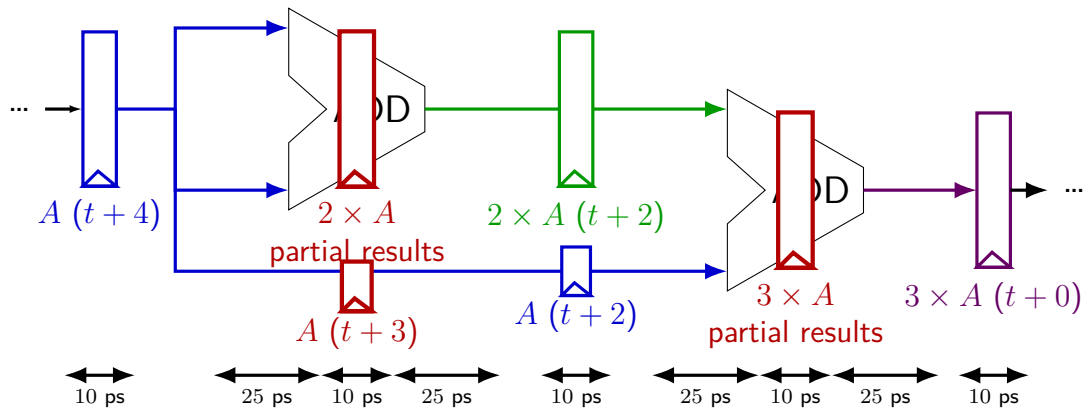


throughput: $\frac{1}{60 \text{ ps}} \approx 16 \text{ G operations/sec}$

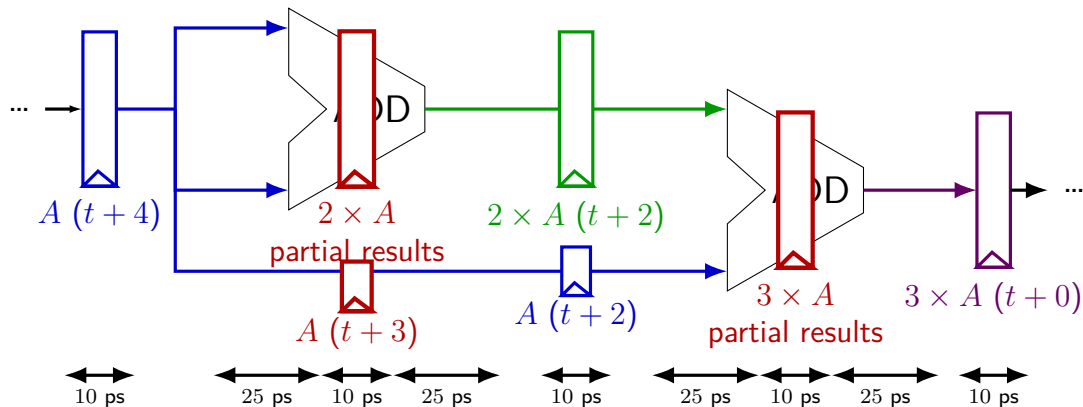
deeper pipeline



deeper pipeline



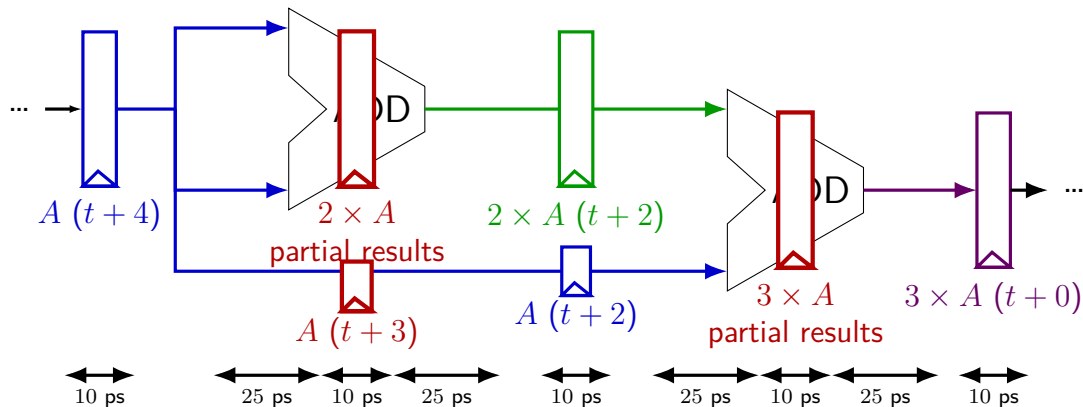
deeper pipeline



Problem: How much faster can we get?

Problem: Can we even do this?

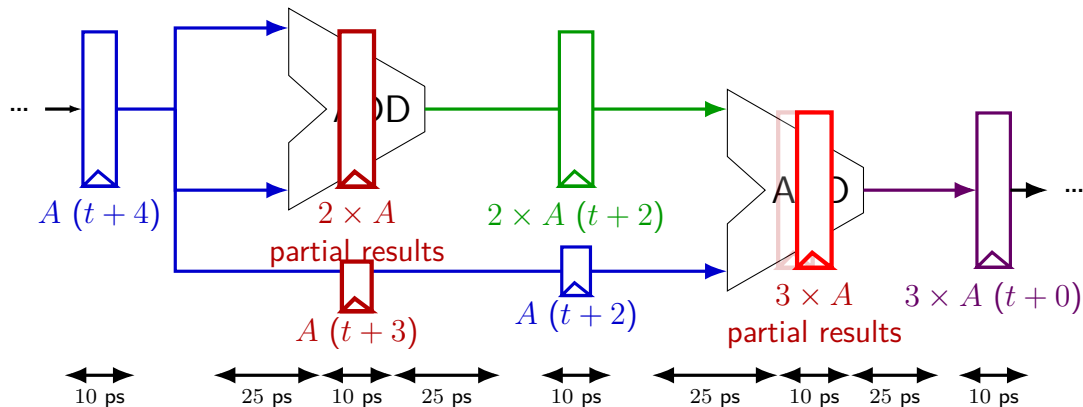
deeper pipeline



exercise: throughput now?

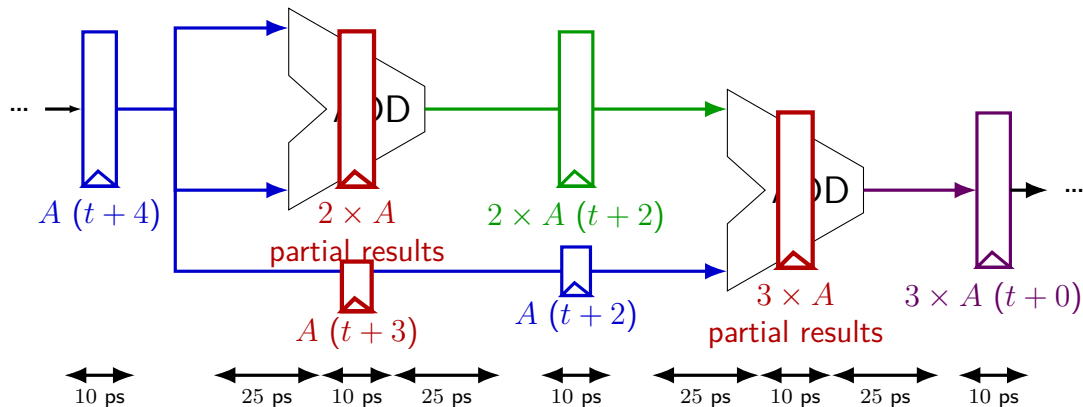
- A. $1/(25 \text{ ps})$
- B. $1/(30 \text{ ps})$
- C. $1/(35 \text{ ps})$
- D. something else

deeper pipeline



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

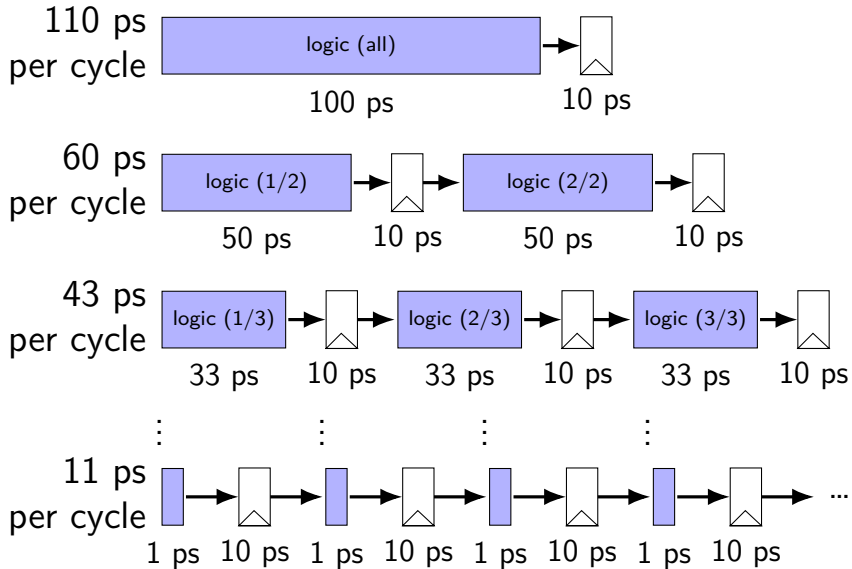
deeper pipeline



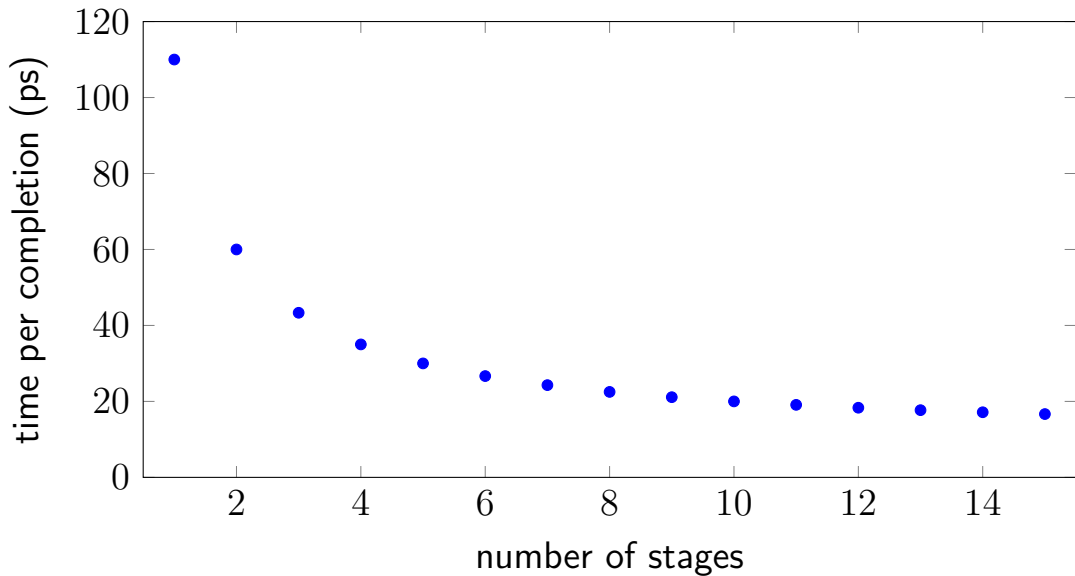
Problem: How much faster can we get?

Problem: Can we even do this?

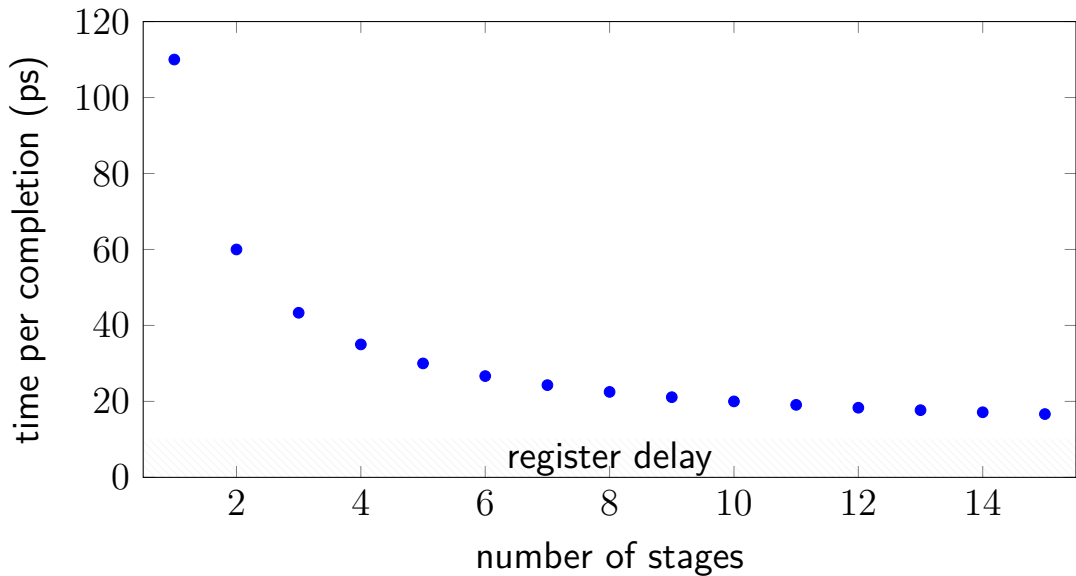
diminishing returns: register delays



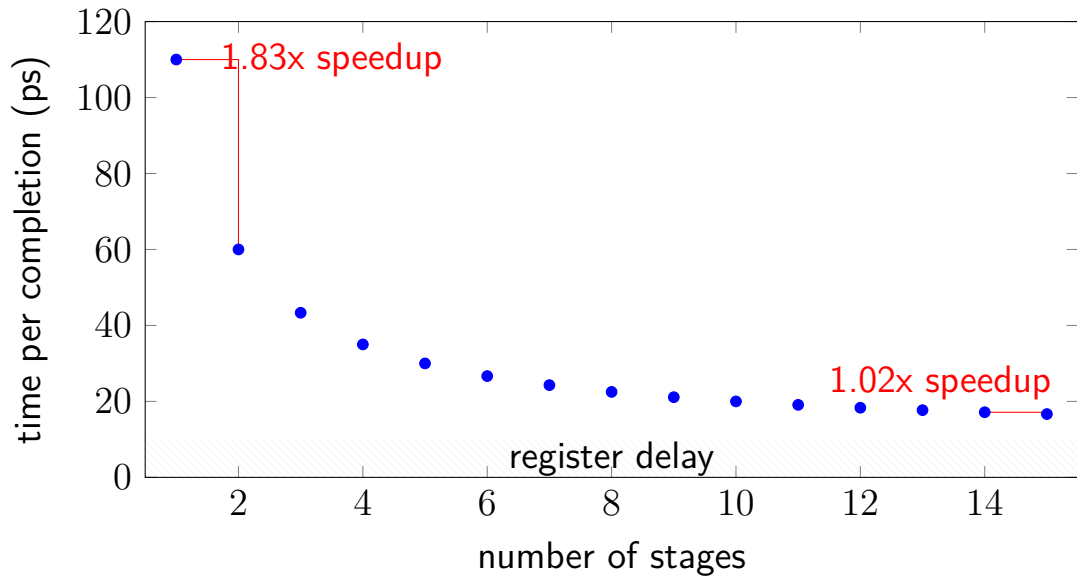
diminishing returns: register delays



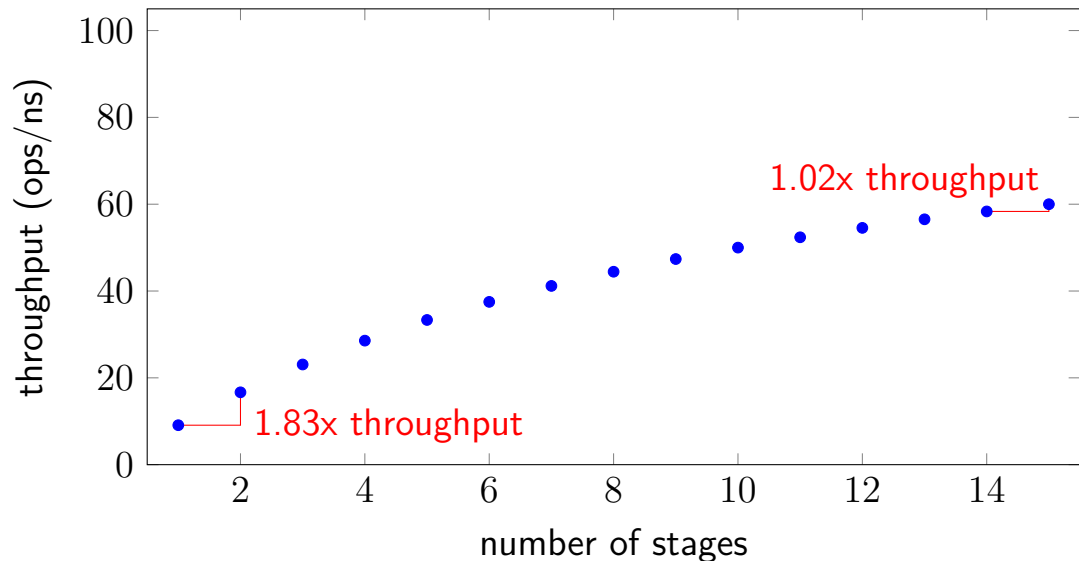
diminishing returns: register delays



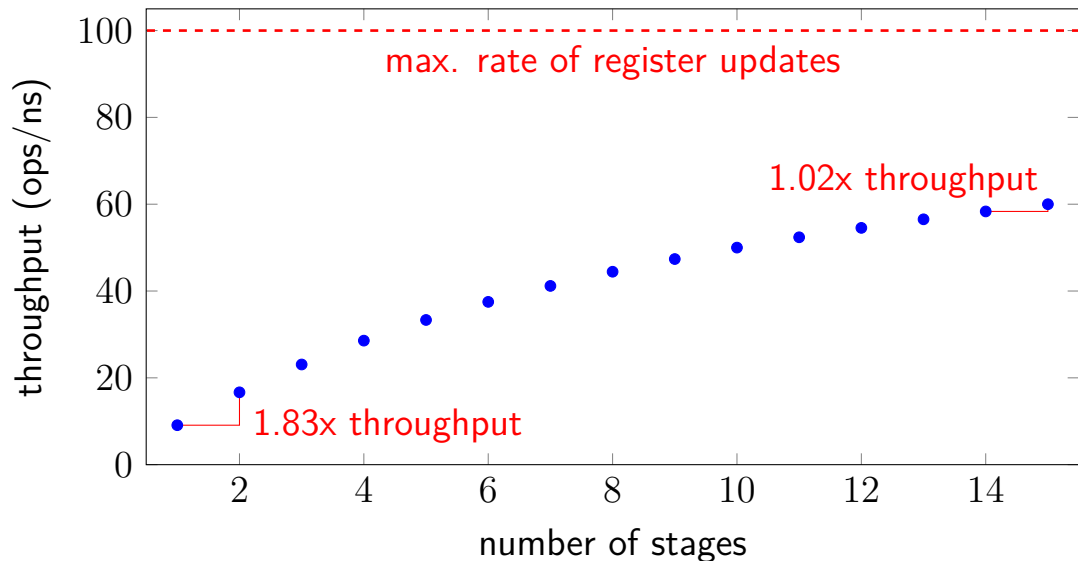
diminishing returns: register delays



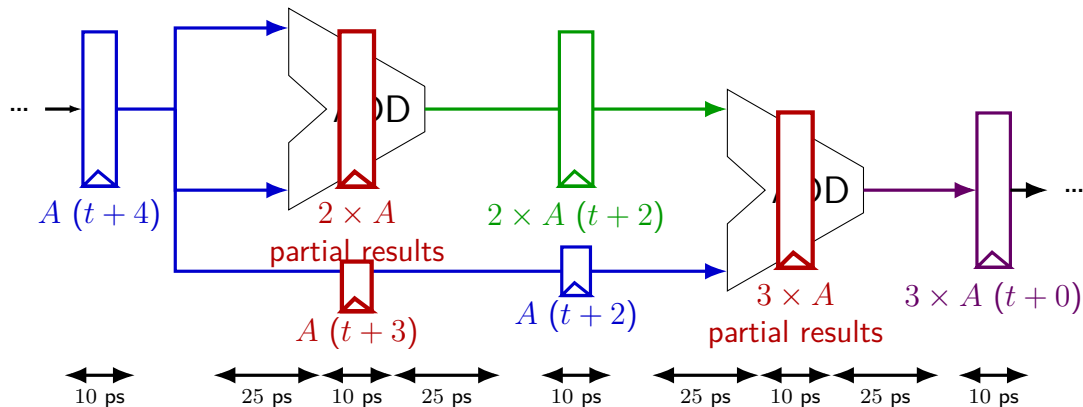
diminishing returns: register delays



diminishing returns: register delays



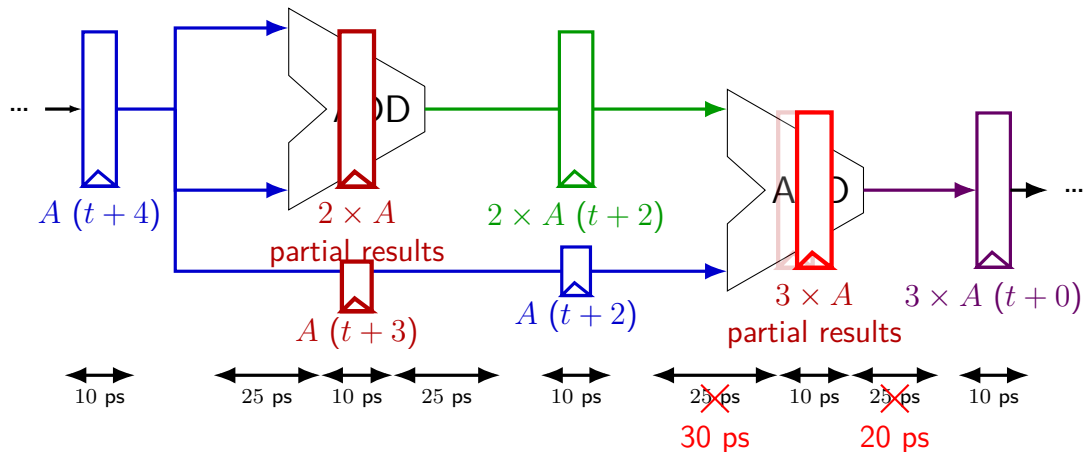
deeper pipeline



Problem: How much faster can we get?

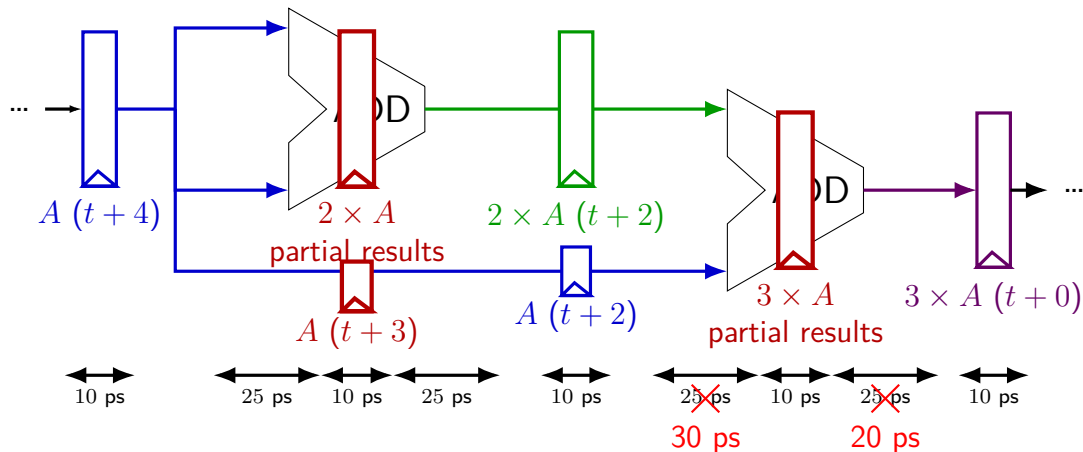
Problem: Can we even do this?

deeper pipeline



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

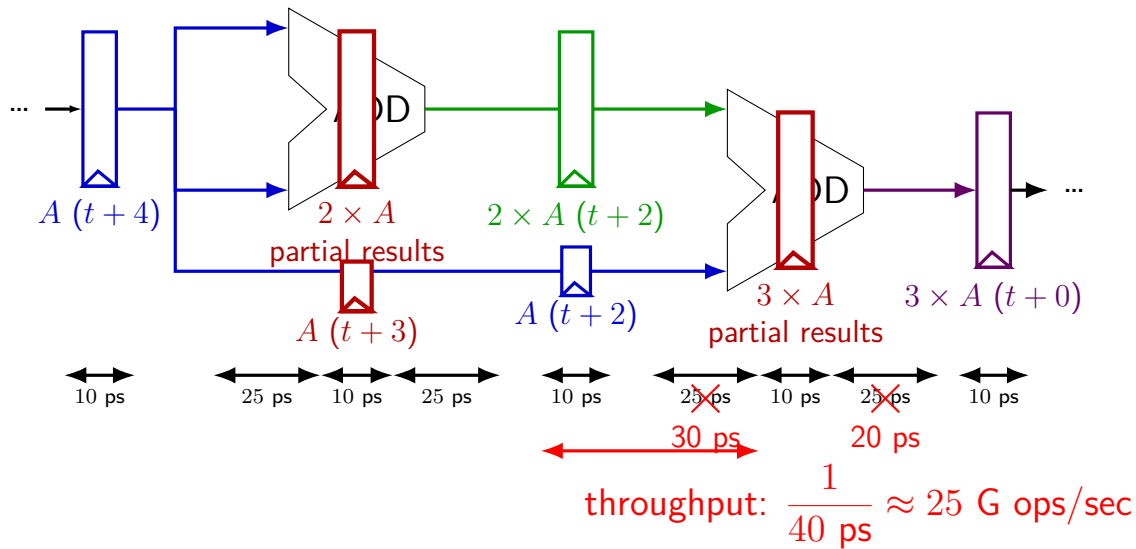
deeper pipeline



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

- A. $1/(25 \text{ ps})$
- B. $1/(30 \text{ ps})$
- C. $1/(35 \text{ ps})$
- D. $1/(40 \text{ ps})$
- E. something else

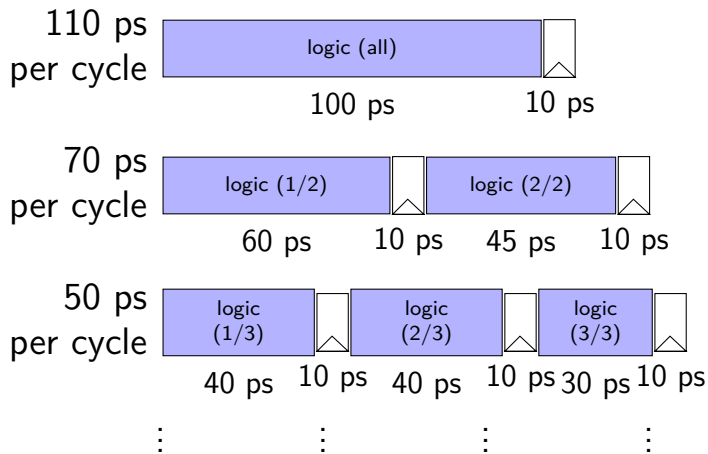
deeper pipeline



diminishing returns: uneven split

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

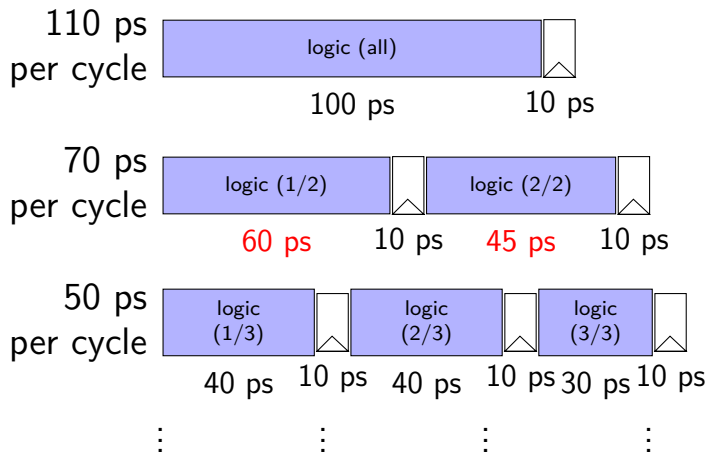
Probably not...



diminishing returns: uneven split

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

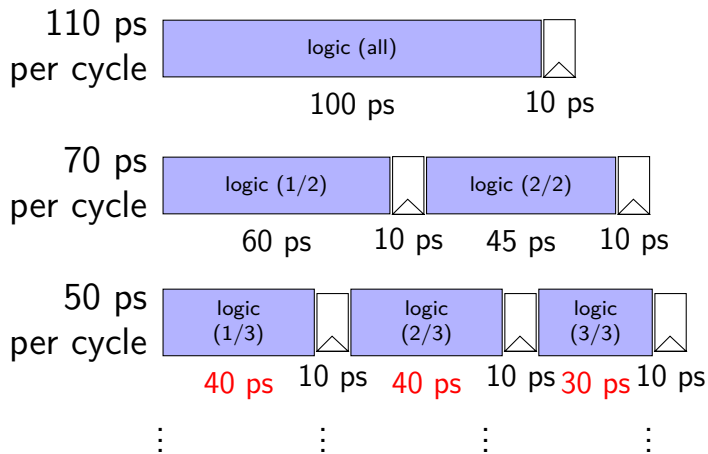
Probably not...



diminishing returns: uneven split

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

Probably not...



textbook SEQ 'stages'

conceptual order only

Fetch: read instruction memory

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

PC Update: write PC register

textbook SEQ 'stages'

conceptual order only

Fetch: read instruction memory

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/**write** data memory

Writeback: **write** register file

PC Update: **write** PC register

writes happen
at end of cycle

textbook SEQ 'stages'

conceptual order only

Fetch: read instruction memory

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

PC Update: write PC register

reads — “magic”
like combinatorial logic
as values available

textbook stages

~~conceptual order only~~ pipeline stages

Fetch/PC Update: read instruction memory;
compute next PC

Decode: read register file

Execute: arithmetic (ALU)

Memory: read/write data memory

Writeback: write register file

textbook stages

conceptual order-only pipeline stages

Fetch/PC Update: read instruction memory;
compute next PC

Decode: read register file

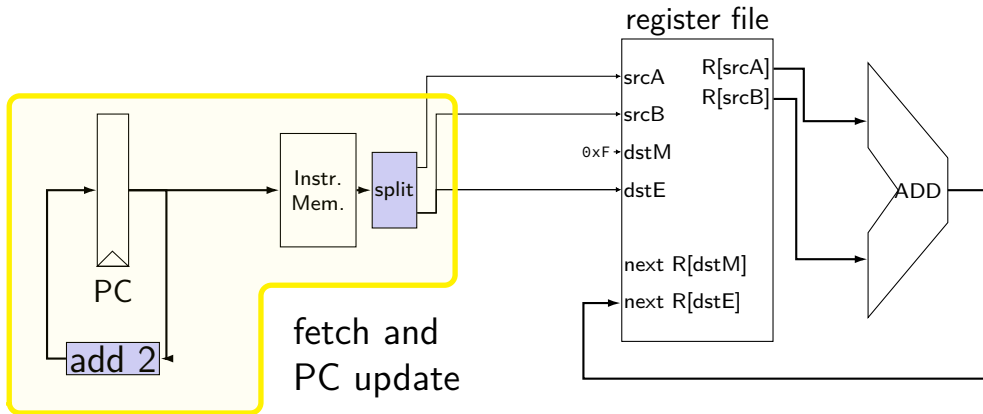
Execute: arithmetic (ALU)

Memory: read/write data memory

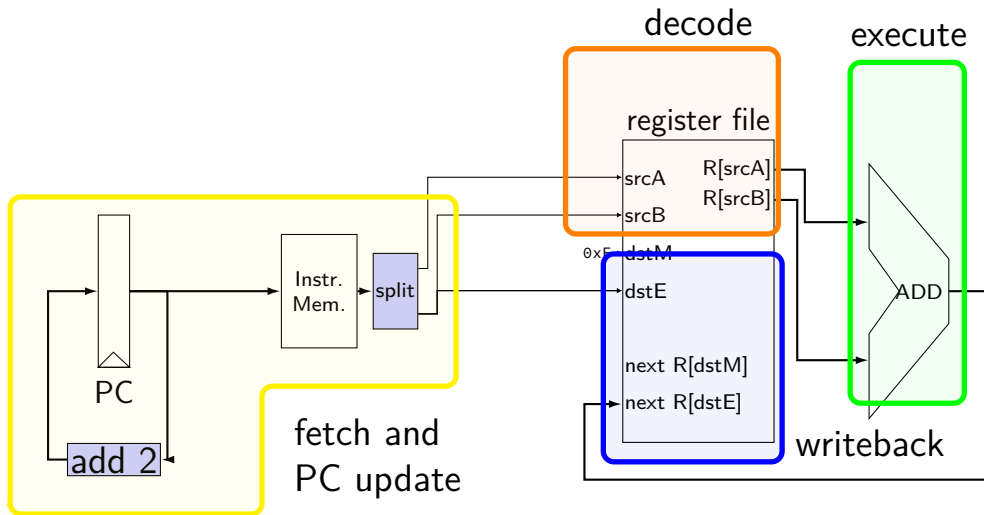
Writeback: write register file

5 stages
one instruction in each
compute next to start immediately

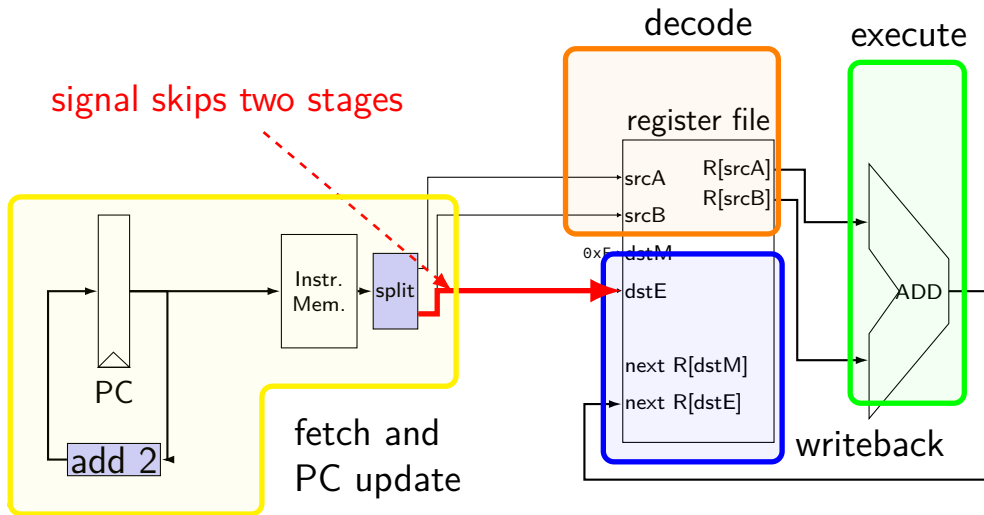
addq CPU



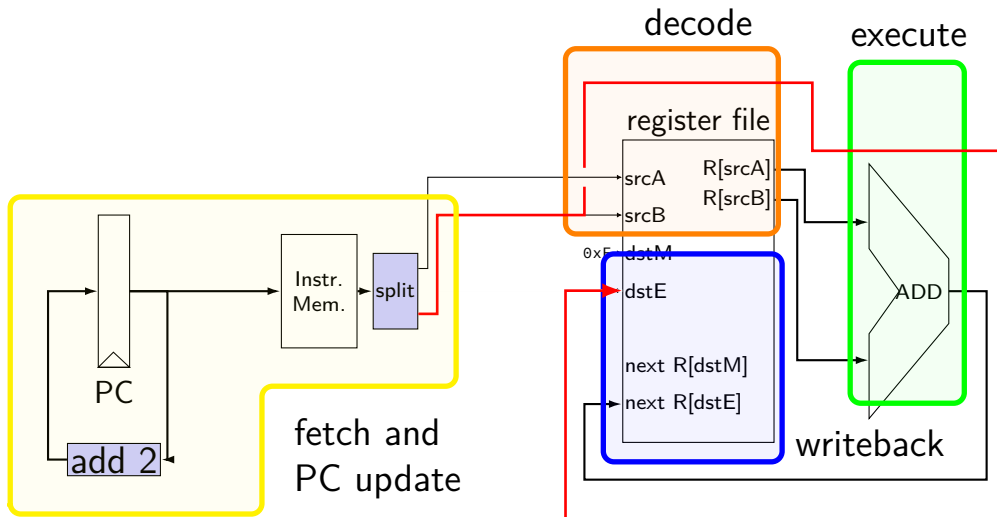
addq CPU



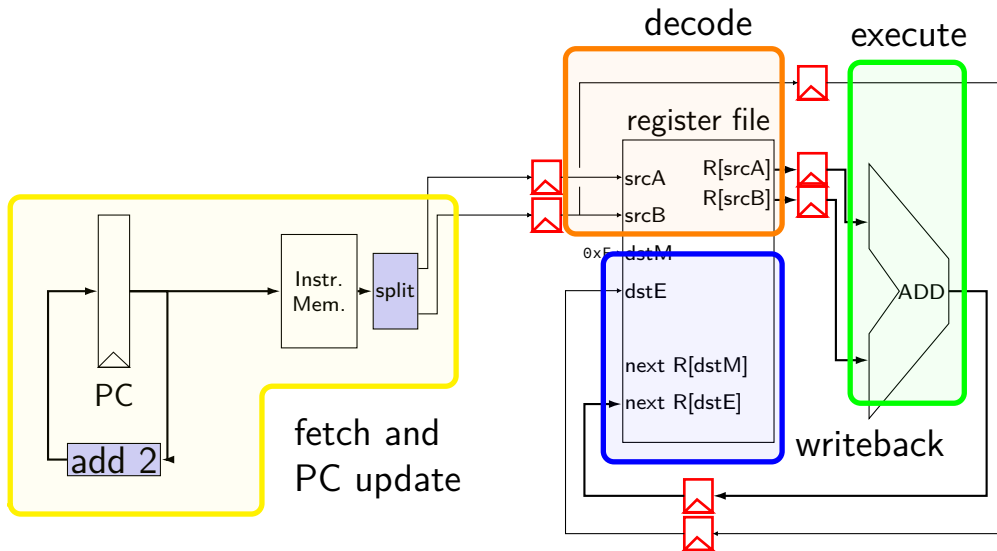
addq CPU



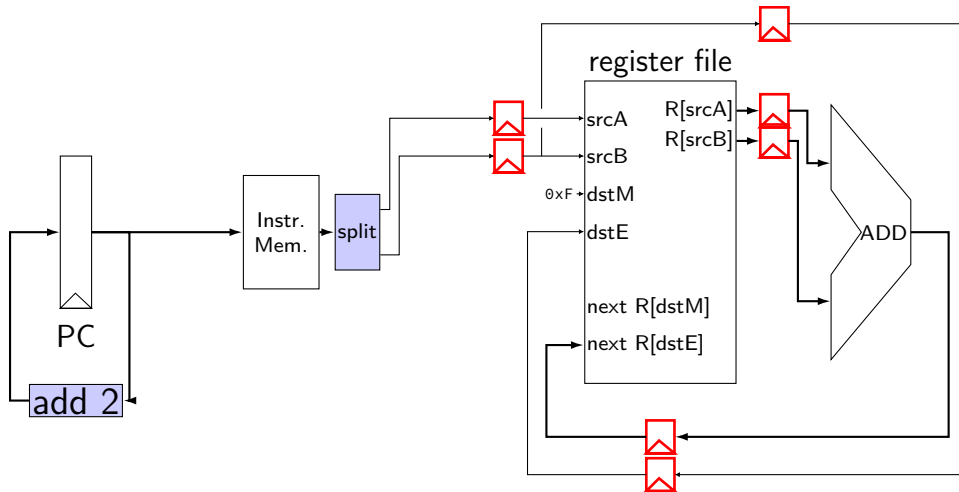
addq CPU



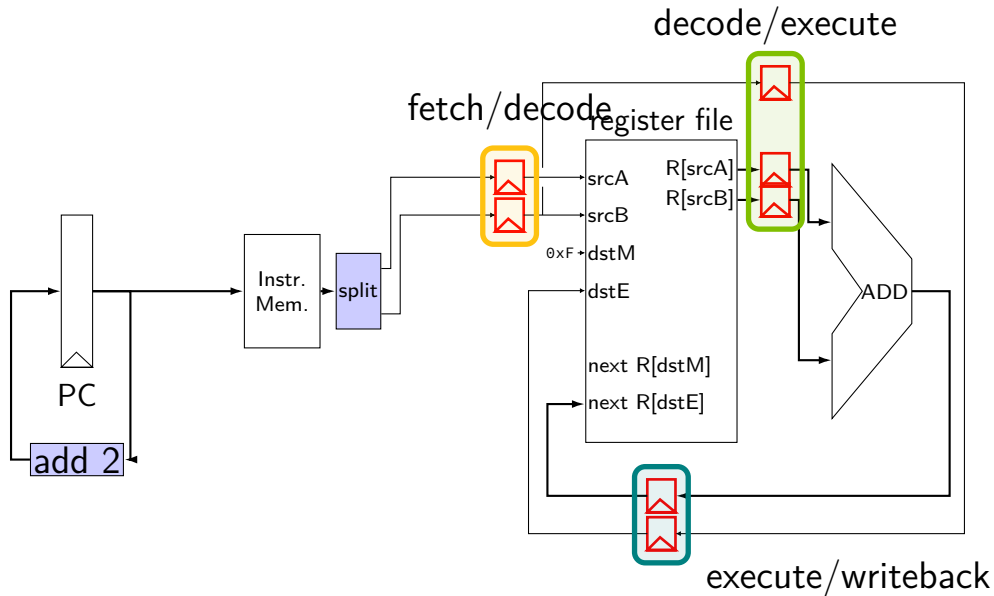
pipelined addq processor



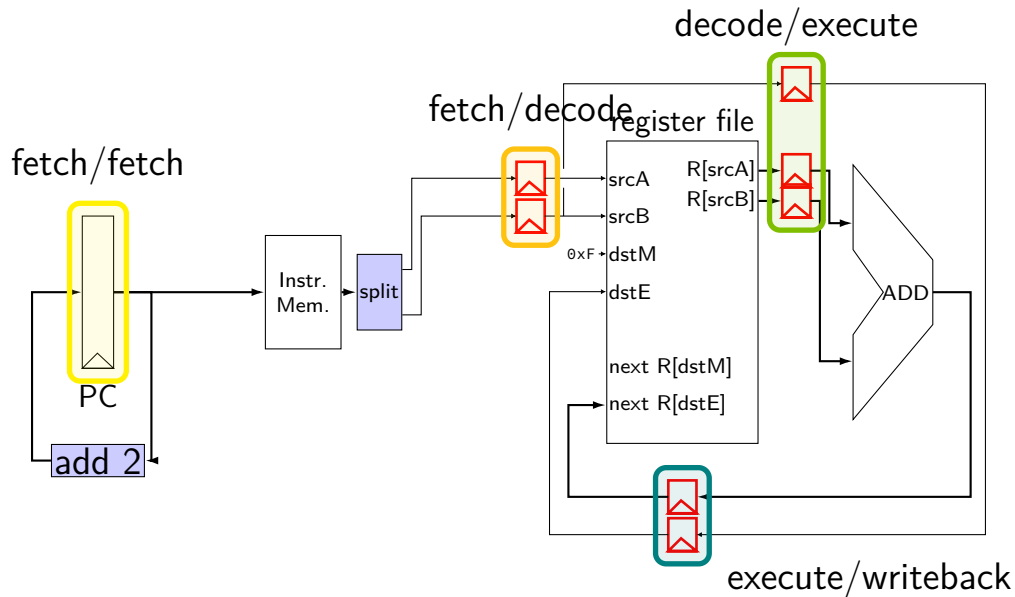
pipelined addq processor



pipelined addq processor

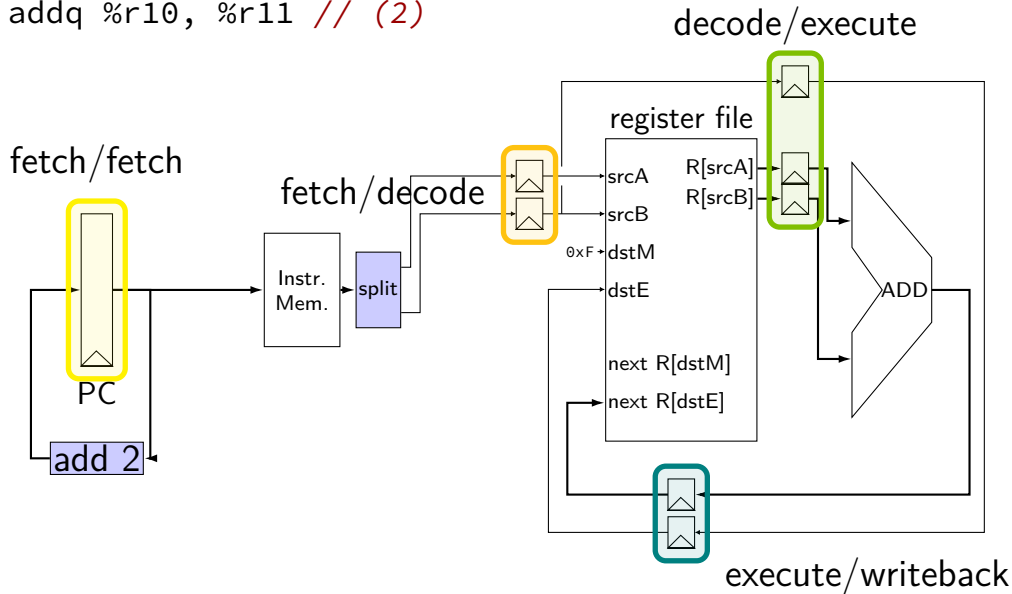


pipelined addq processor



addq execution

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



addq execution

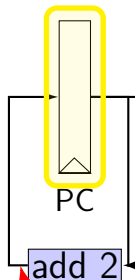
```
addq %r8, %r9 // (1)  
addq %r10, %r11 // (2)
```

fetch/fetch

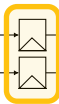
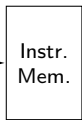
fetch/decode

decode/execute

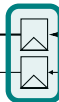
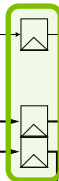
execute/writeback



addq %r8, %r9 // (1)

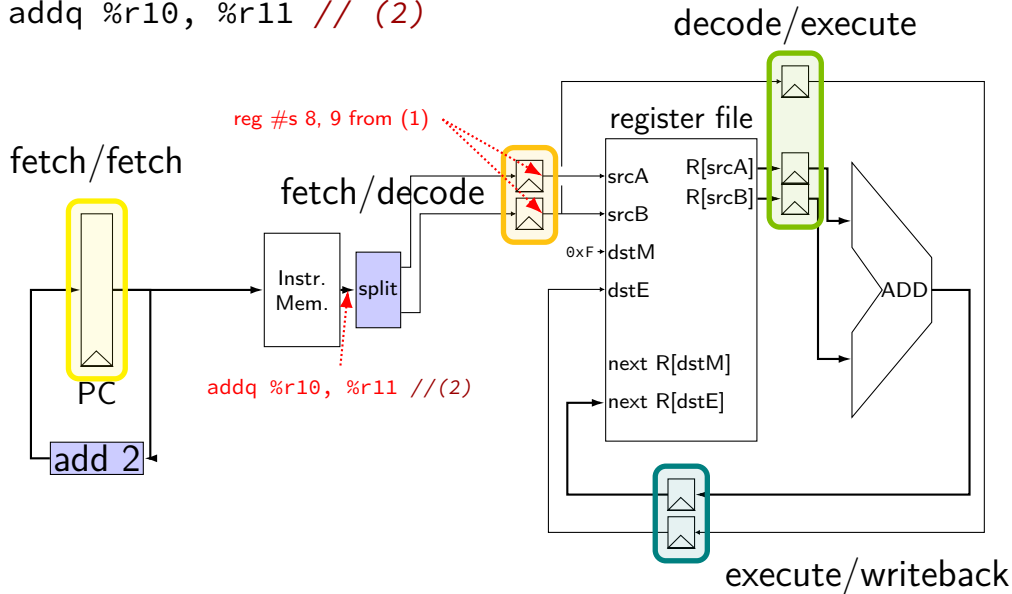


register file



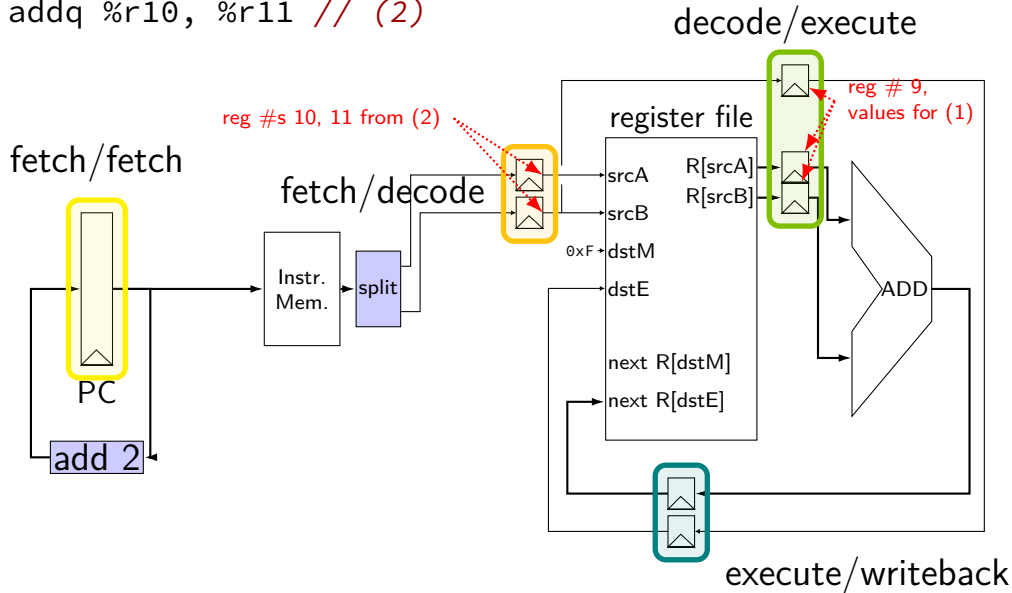
addq execution

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



addq execution

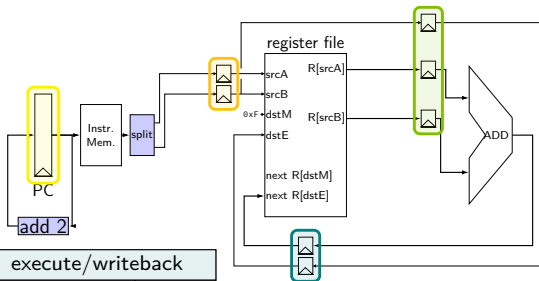
```
addq %r8, %r9 // (1)  
addq %r10, %r11 // (2)
```



addq processor timing

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

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

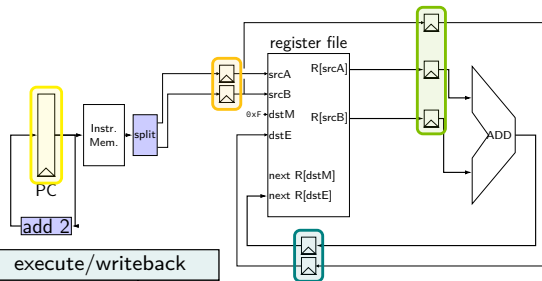


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

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

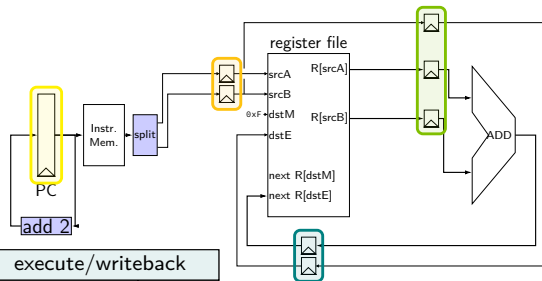
```
// initially %r8 = 800,  
//                %r9 = 900, etc.
```

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

```
addq %r10, %r11
```

```
addq %r12, %r13
```

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

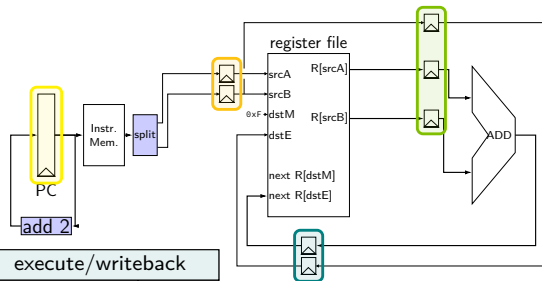


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

addq processor timing

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

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

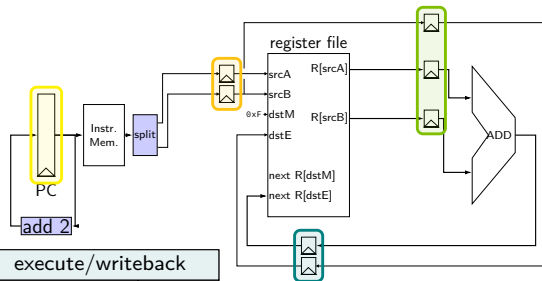


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2	0x4	10	11	800	900	9		
3	0x6	12	13	1000	1100	11	1700	9
4		9	8	1200	1300	13	2100	11
5				1700	800	8	2500	13
6							2500	8

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

critical path

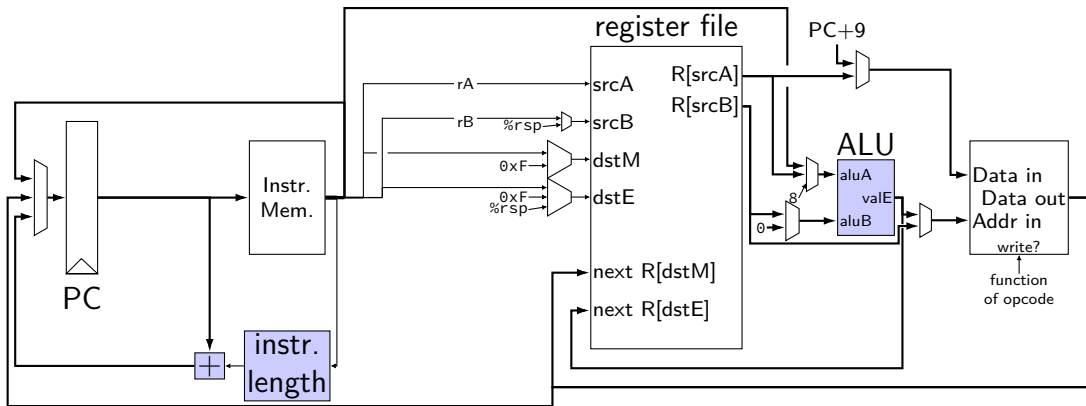
every path from state output to state input needs enough time
output — may change on rising edge of clock
input — must be stable sufficiently before rising edge of clock

critical path: slowest of all these paths — determines cycle time
times three: slowest stage ended up mattering

have to choose *one* clock cycle length
can't vary clock depending on what instruction is running

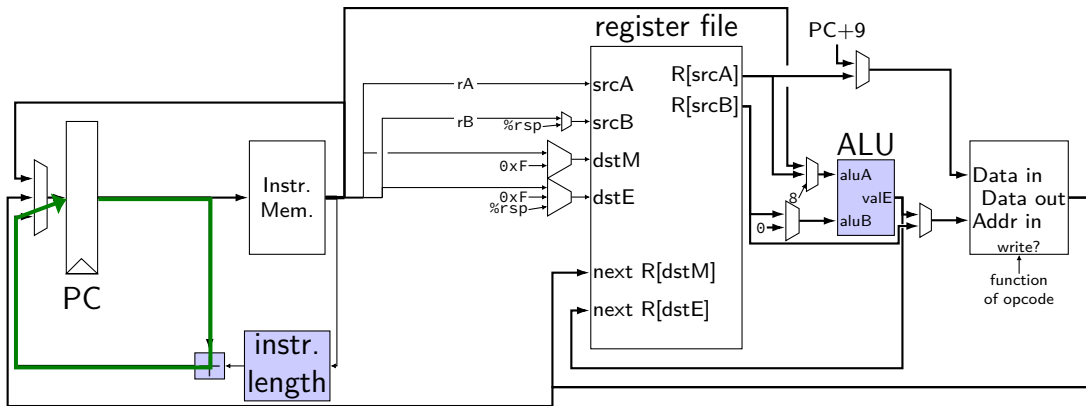
matters with or without pipelining

SEQ paths



SEQ paths

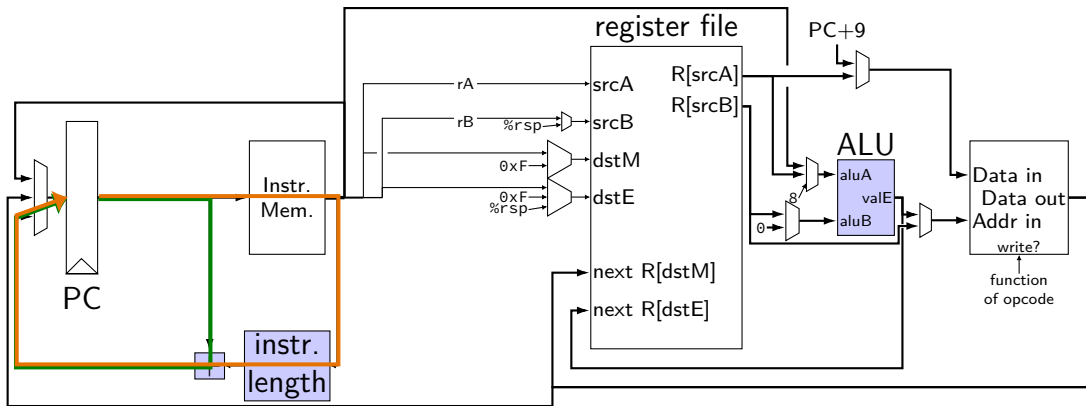
path 1: 25 picoseconds



SEQ paths

path 1: 25 picoseconds

path 2: 50 picoseconds

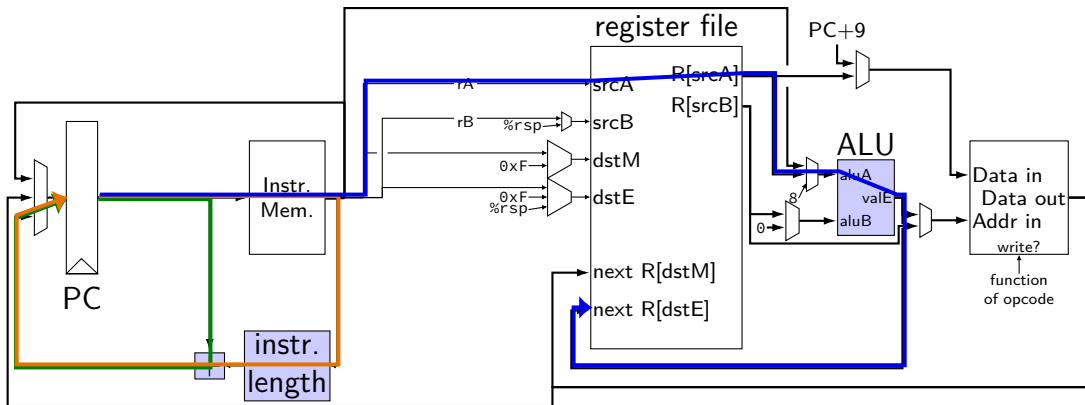


SEQ paths

path 1: 25 picoseconds

path 2: 50 picoseconds

path 3: 400 picoseconds



SEQ paths

path 1: 25 picoseconds

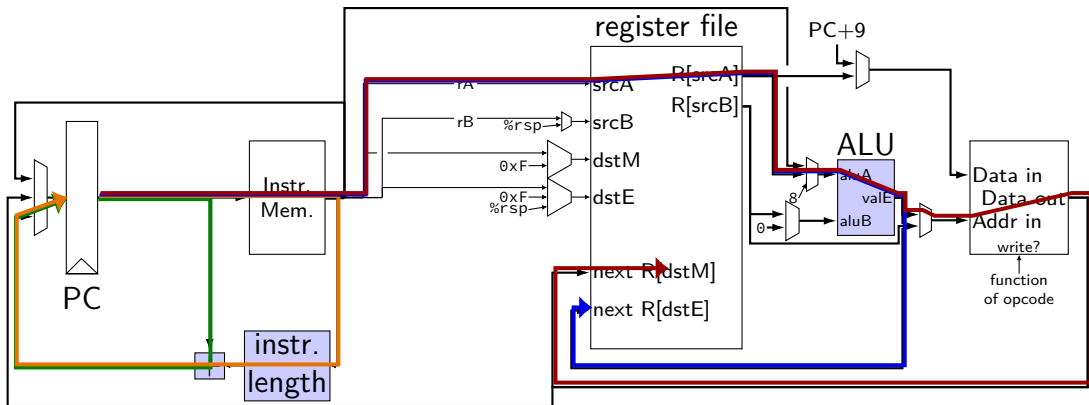
path 3: 400 picoseconds

...

path 2: 50 picoseconds

path 4: 900 picoseconds

...



SEQ paths

path 1: 25 picoseconds

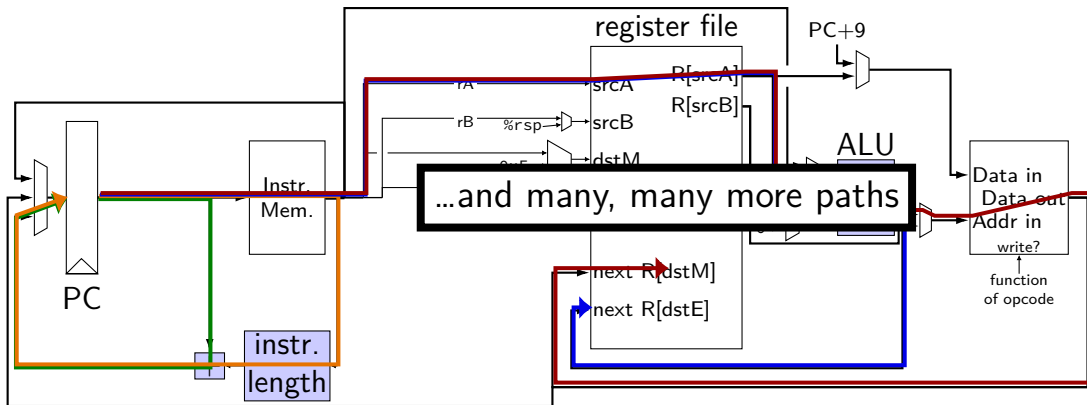
path 3: 400 picoseconds

...

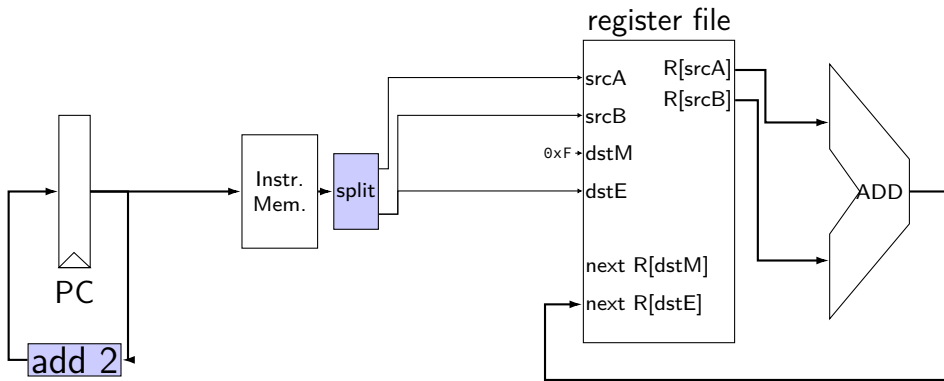
path 2: 50 picoseconds

path 4: 900 picoseconds

...

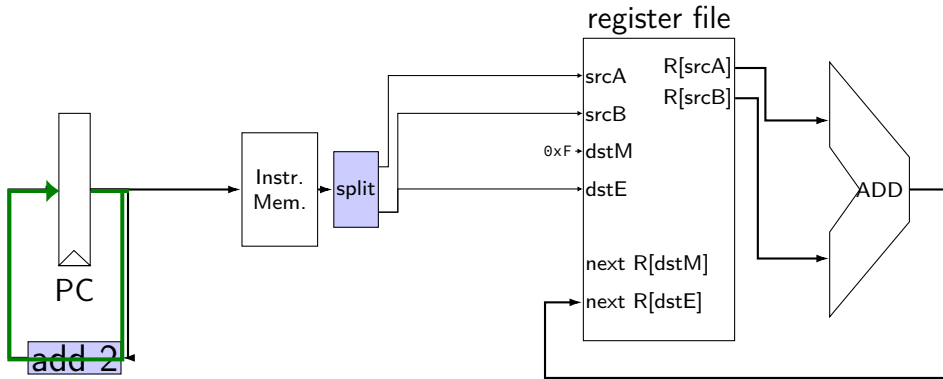


sequential addq paths



sequential addq paths

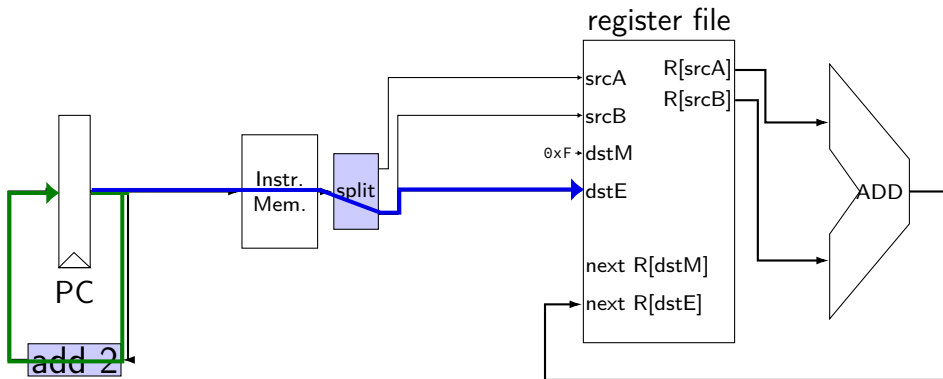
path 1: 25 picoseconds



sequential addq paths

path 1: 25 picoseconds

path 2: 375 picoseconds

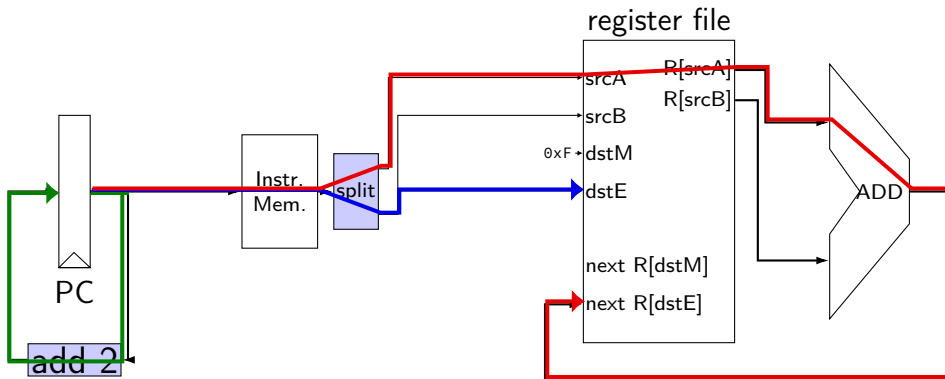


sequential addq paths

path 1: 25 picoseconds

path 2: 375 picoseconds

path 3: 500 picoseconds



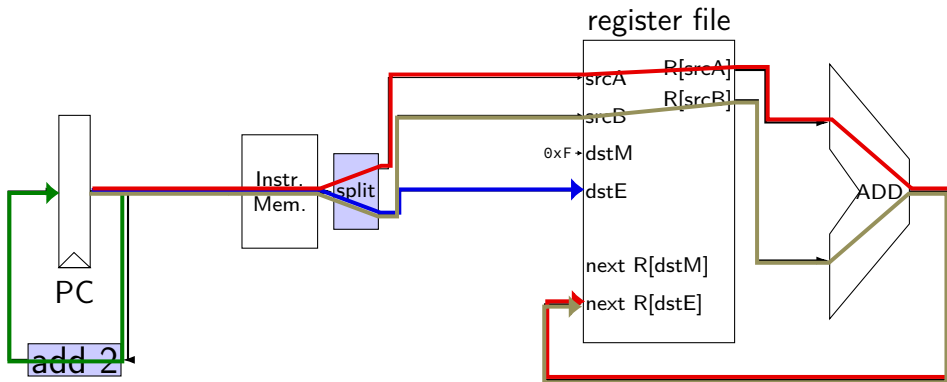
sequential addq paths

path 1: 25 picoseconds

path 2: 375 picoseconds

path 3: 500 picoseconds

path 4: 500 picoseconds



sequential addq paths

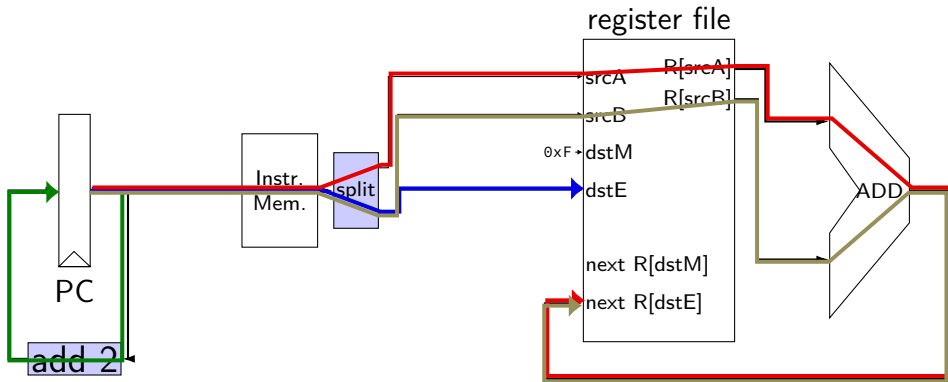
path 1: 25 picoseconds

path 2: 375 picoseconds

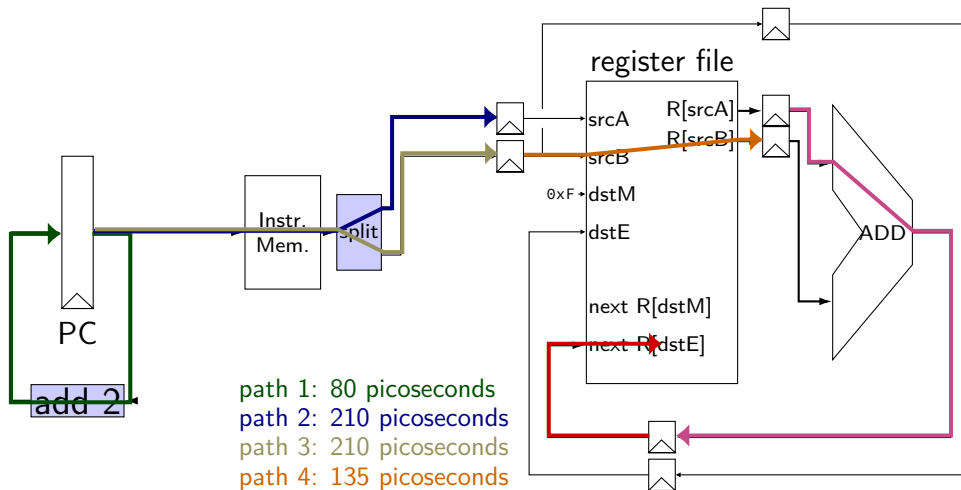
path 3: **500 picoseconds**

path 4: **500 picoseconds**

overall cycle time: **500 picoseconds** (longest path)



pipelined addq paths



path 1: 80 picoseconds

path 2: 210 picoseconds

path 3: 210 picoseconds

path 4: 135 picoseconds

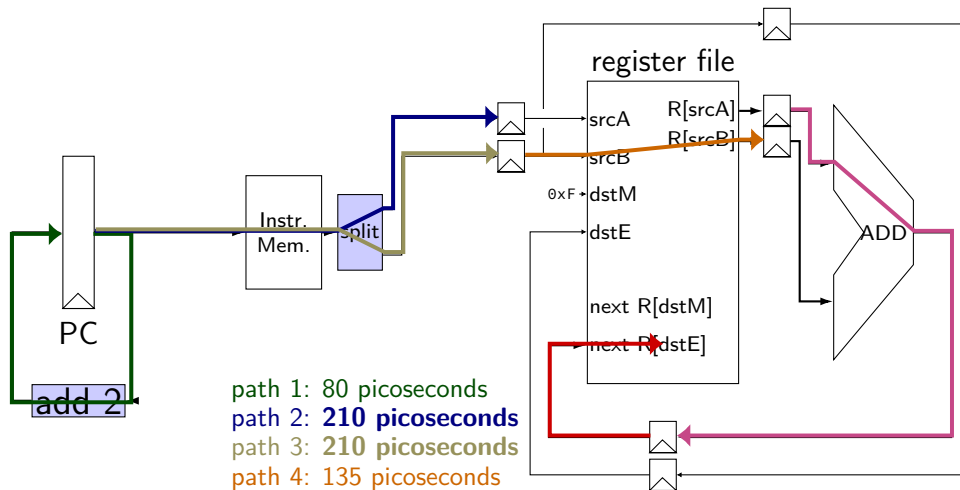
path 5: 110 picoseconds

path 6: 135 picoseconds

...

overall cycle time: 210 picoseconds

pipelined addq paths



path 1: 80 picoseconds
path 2: **210 picoseconds**
path 3: **210 picoseconds**
path 4: 135 picoseconds
path 5: 110 picoseconds
path 6: 135 picoseconds

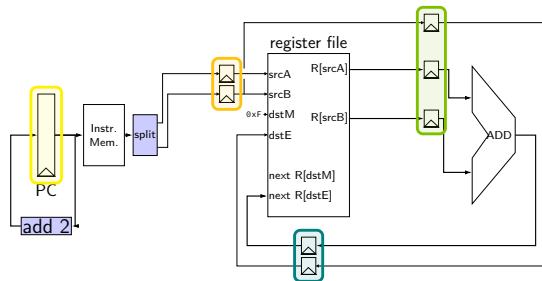
...

overall cycle time: **210 picoseconds**

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

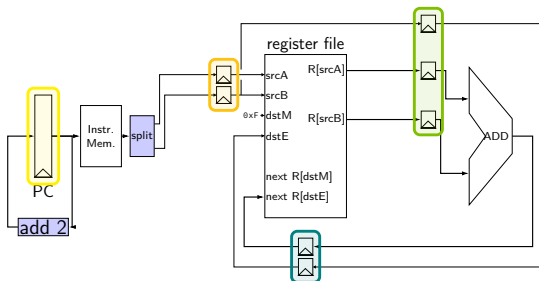
slowest path through stage + pipeline register delays

latency: 800 ps + pipeline register delays (4 cycles)

addq processor timing exercise 1

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



exercise 1: when instruction 1 stores its result in %rbx, what did instruction 3 *just complete*?

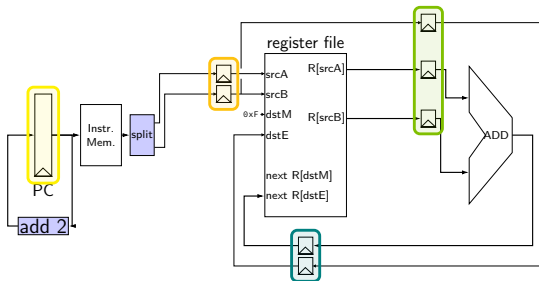
```
addq %rax, %rbx /* 1 */
addq %rcx, %rdx
addq %r8, %r9 /* 3 */
```

addq processor performance exercise

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

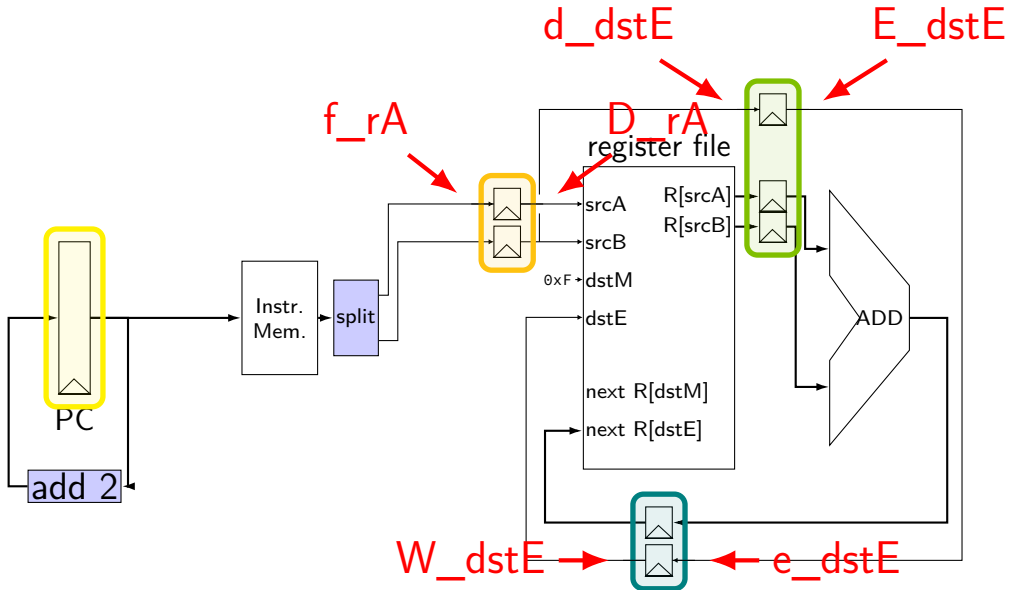
now, cycle time = 200 ps + register delays per cycle



exercise 2:

suppose we combine add and register file read together — new cycle time?

pipeline register naming convention



pipeline register naming convention

f — fetch sends values here

D — decode receives values here

d — decode sends values here

...

addq HCL

```
...
/* f: from fetch */
f_rA = i10bytes[12..16];
f_rB = i10bytes[8..12];

/* fetch to decode */
/* f_rA -> D_rA, etc. */
register fD {
    rA : 4 = REG_NONE;
    rB : 4 = REG_NONE;
}
```

```
/* D: to decode
    d: from decode */
d_dstE = D_rB;
/* use register file: */
reg_srcA = D_rA;
d_valA = reg_outputA;
...
```

```
/* decode to execute */
register dE {
    dstE : 4 = REG_NONE;
    valA : 64 = 0;
    valB : 64 = 0;
}
```

...

addq fetch/decode

unpipelined

```
/* Fetch+PC Update*/  
pc = P_pc;  
p_pc = pc + 2;  
rA = i10bytes[12..16];  
rB = i10bytes[8..12];  
/* Decode */  
reg_srcA = rA;  
reg_srcB = rB;  
reg_dstE = rB;  
valA = reg_outputA;  
valB = reg_outputB;
```

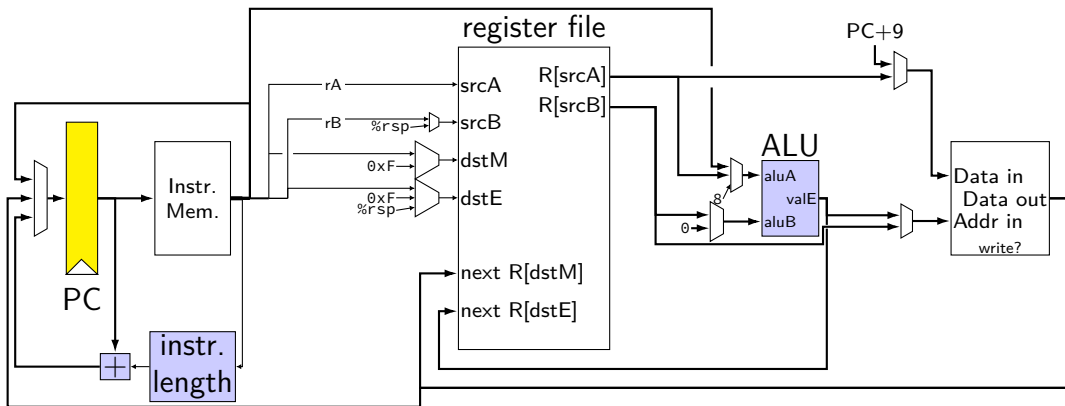
pipelined

```
/* Fetch+PC Update*/  
pc = P_pc;  
p_pc = pc + 2;  
f_rA = i10bytes[12..16];  
f_rB = i10bytes[8..12];  
/* Decode */  
reg_srcA = D_rA;  
reg_srcB = D_rB;  
d_dstE = D_rB;  
d_valA = reg_outputA;  
d_valB = reg_outputB;
```

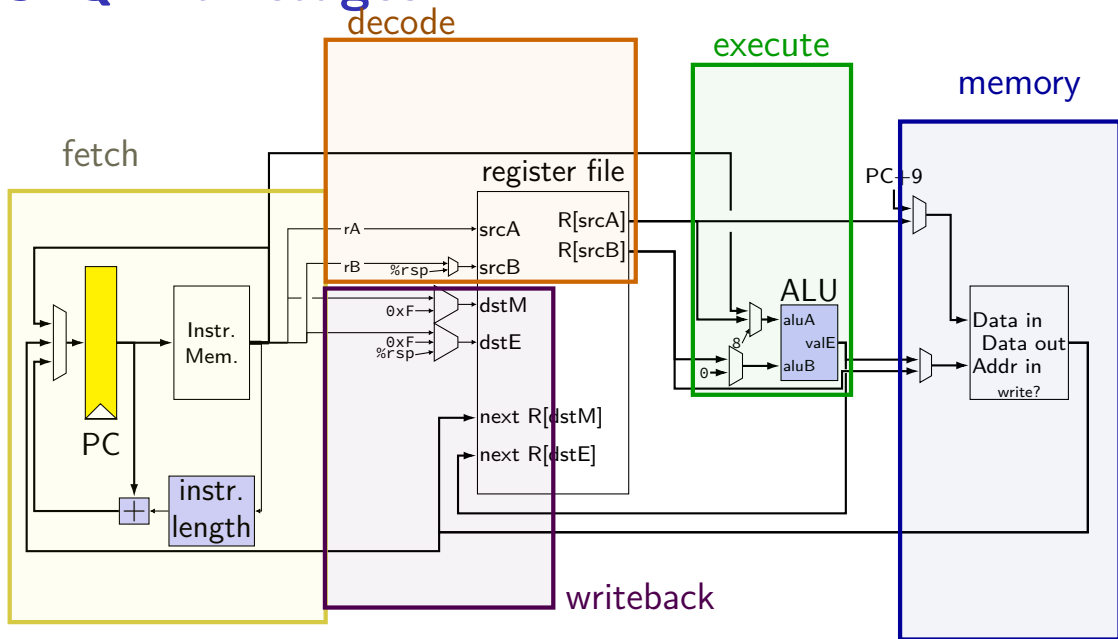
addq pipeline registers

```
register pP {  
    pc : 64 = 0;  
};  
/* Fetch+PC Update*/  
register fD {  
    rA : 4 = REG_NONE; rB : 4 = REG_NONE;  
};  
/* Decode */  
register dE {  
    valA : 64 = 0; valB : 64 = 0; dstE : 4 = REG_NONE;  
}  
/* Execute */  
register eW {  
    valE : 64 = 0; dstE : 4 = REG_NONE;  
}  
/* Writeback */
```

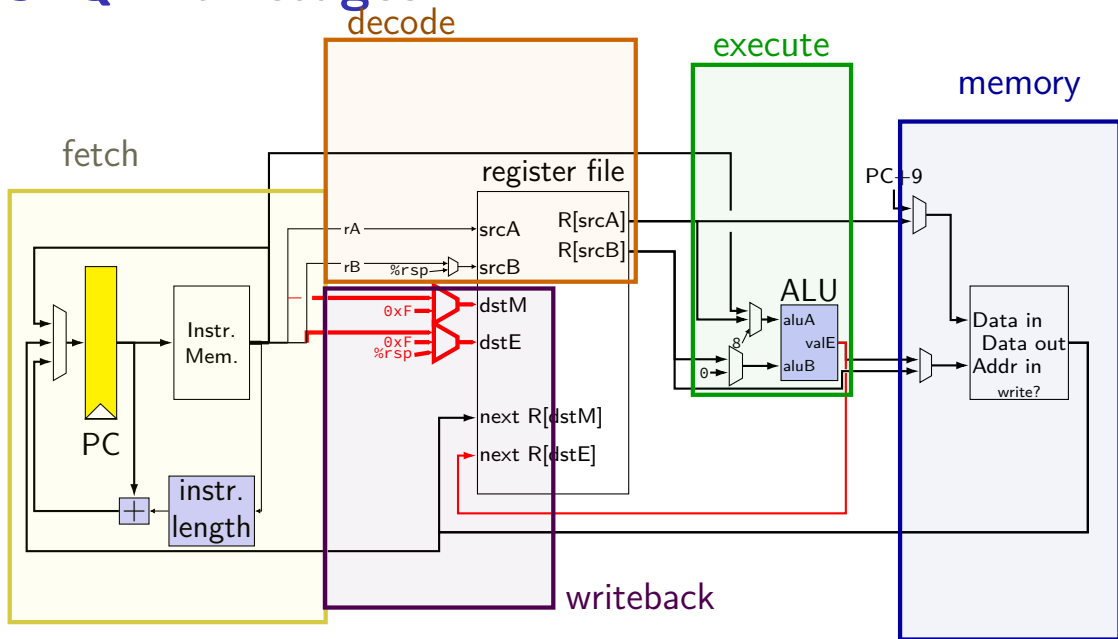

SEQ without stages



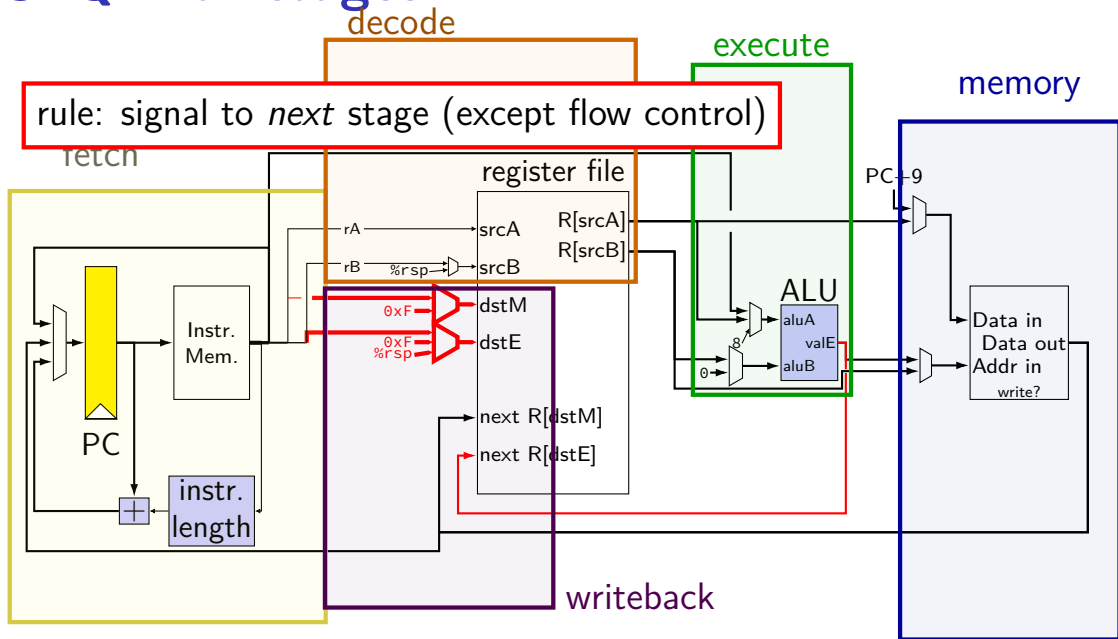
SEQ with stages



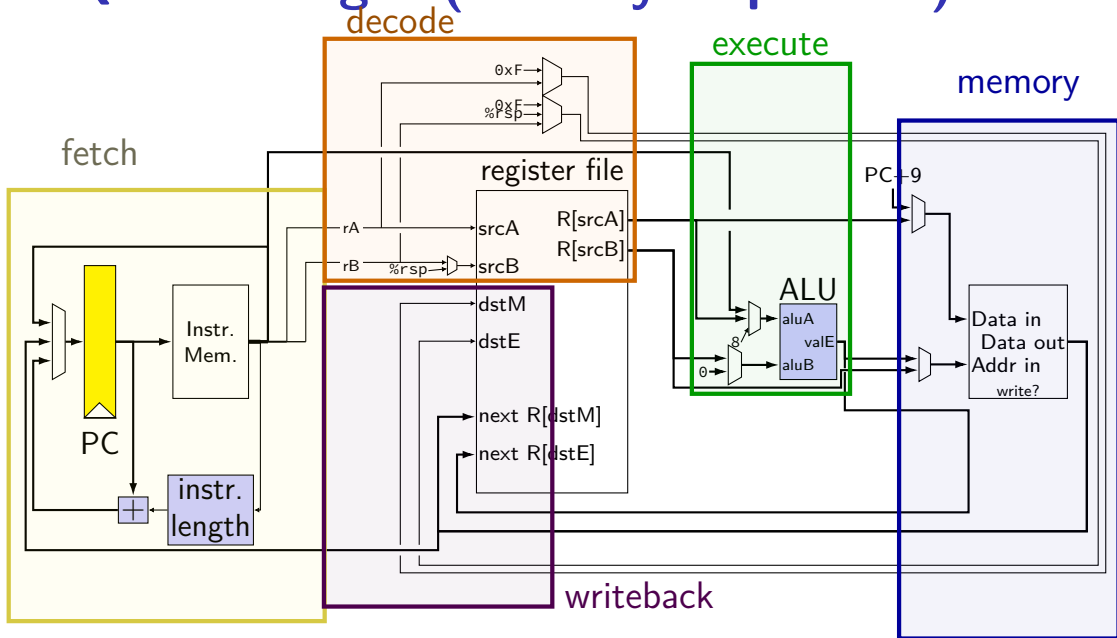
SEQ with stages



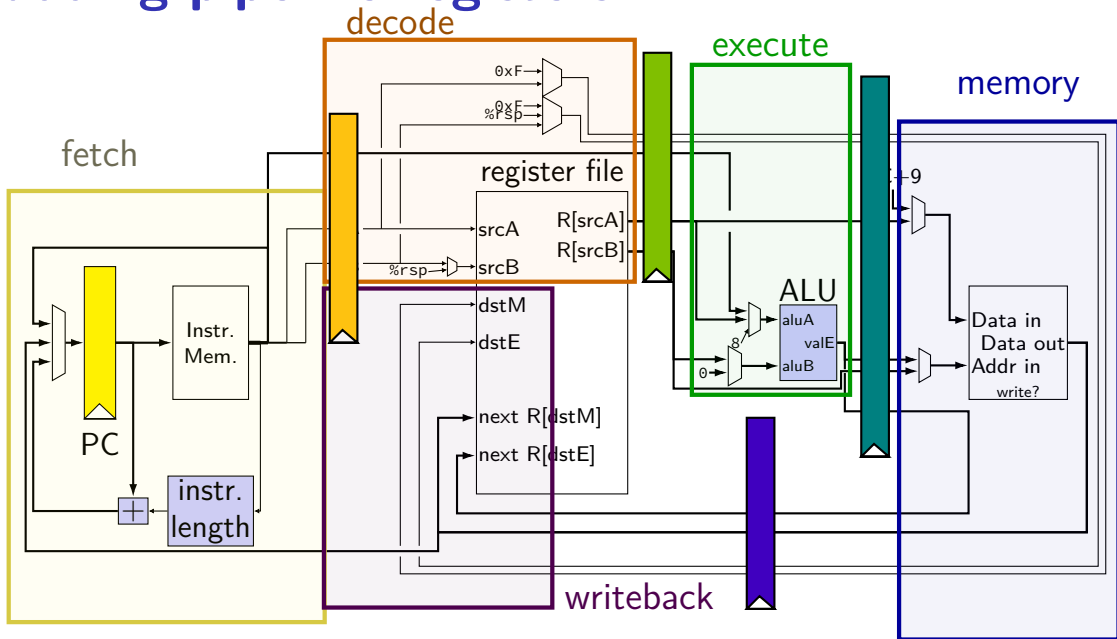
SEQ with stages



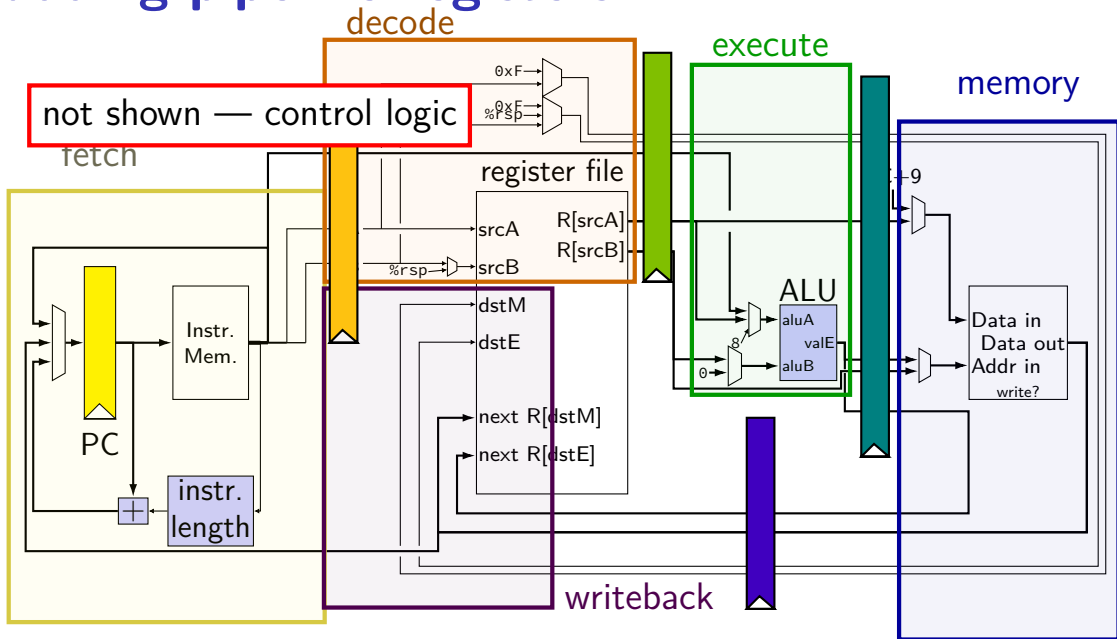
SEQ with stages (actually sequential)



adding pipeline registers



adding pipeline registers



passing values in pipeline

read **prior stage's outputs**

e.g. decode: get from fetch via pipeline registers (D_icode, ...)

send **inputs for next stage**

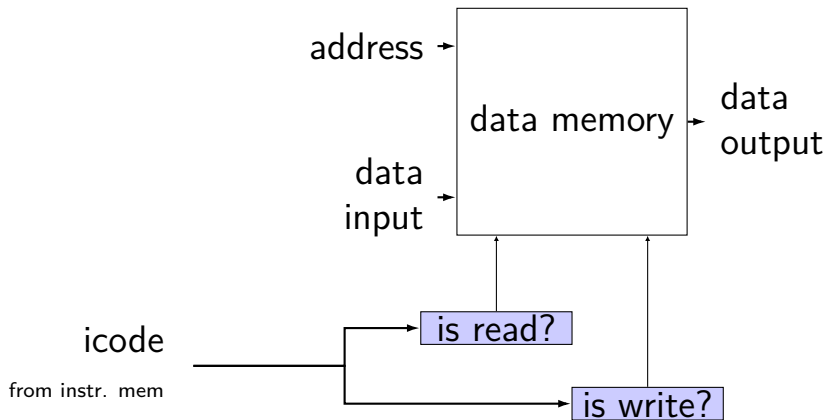
e.g. decode: send to execute via pipeline registers (d_icode, ...)

exceptions: **deliberate sharing** between instructions

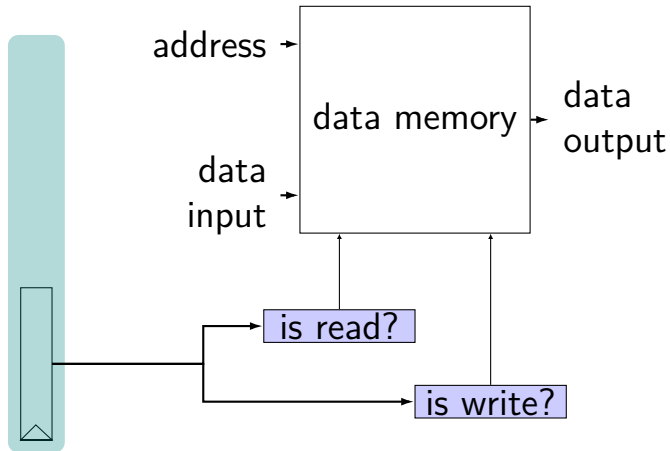
via register file/memory/etc.

via control flow instructions

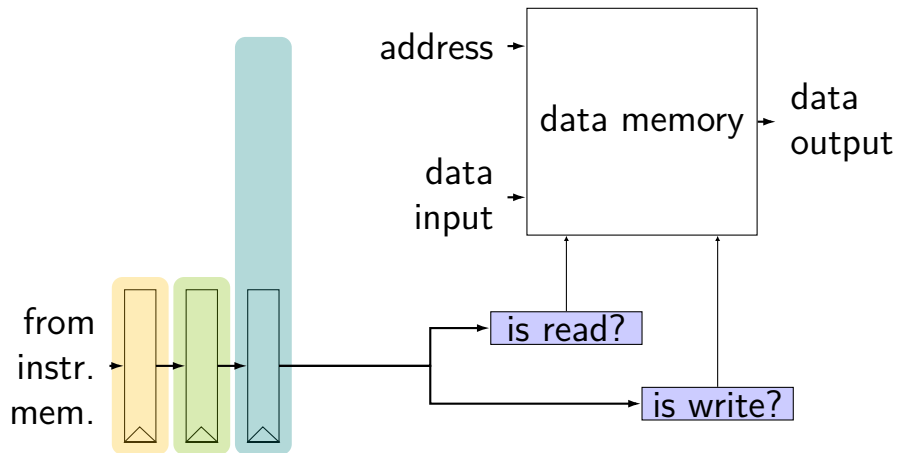
memory read/write logic



memory read/write logic



memory read/write logic



memory read/write: SEQ code

```
icode = i10bytes[4..8];  
mem_readbit = [  
    icode == MRMOVQ || ...: 1;  
    0;  
];
```

memory read/write: PIPE code

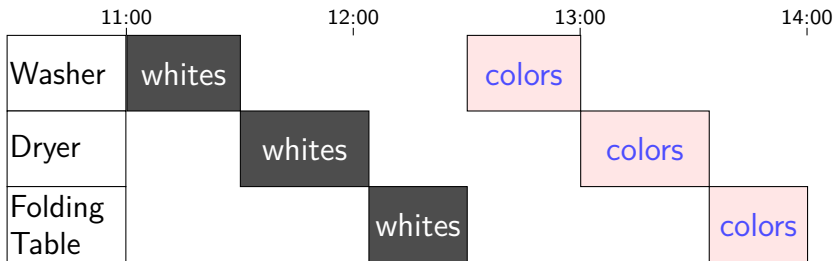
```
f_icode = i10bytes[4..8];  
register fD { /* and dE and eM and mW */  
    icode : 4 = NOP;  
}  
d_icode = D_icode;  
...  
e_icode = E_icode;  
mem_readbit = [  
    M_icode == MRMOVQ || ...: 1;  
    0;  
];
```

memory read/write: PIPE code

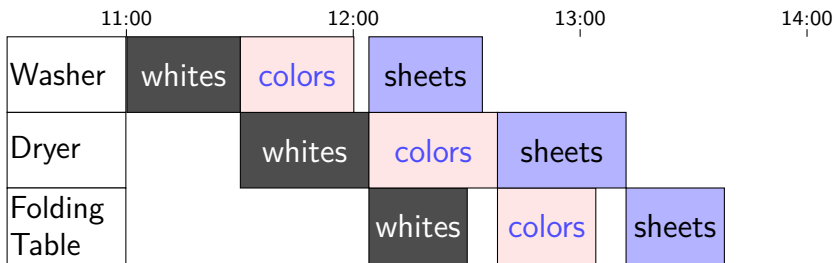
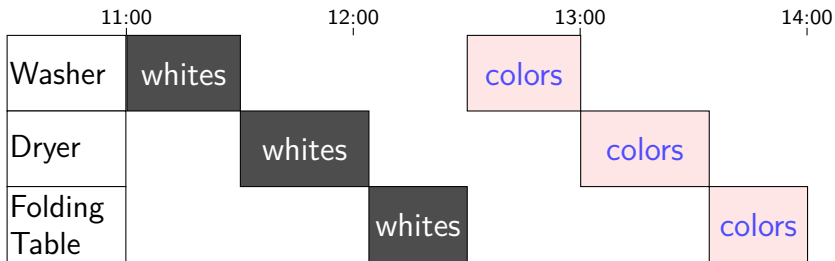
```
f_icode = i10bytes[4..8];  
register fD { /* and dE and eM and mW */  
    icode : 4 = NOP;  
}  
d_icode = D_icode;  
...  
e_icode = E_icode;  
mem_readbit = [  
    M_icode == MRMVQ || ... : 1;  
    0;  
];
```

backup slides

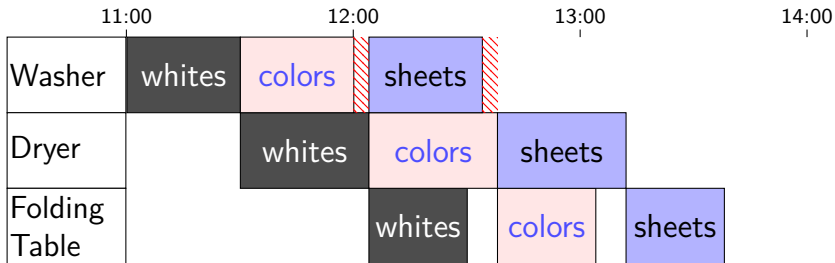
Human pipeline: laundry



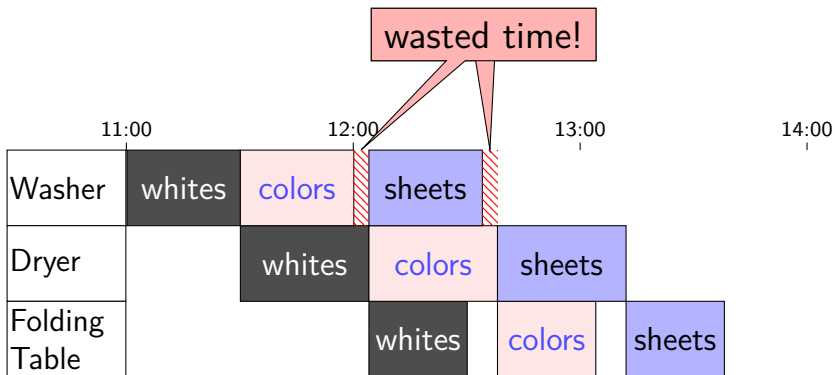
Human pipeline: laundry



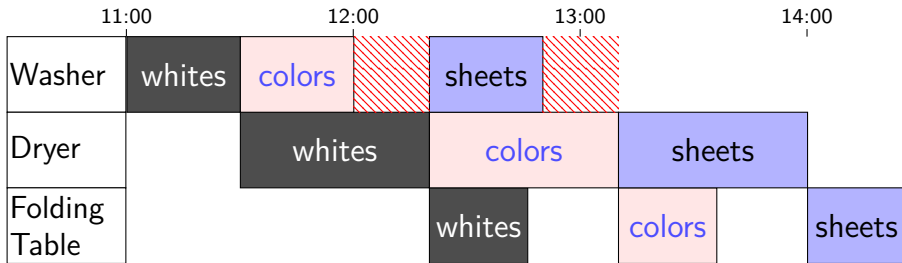
Waste (1)



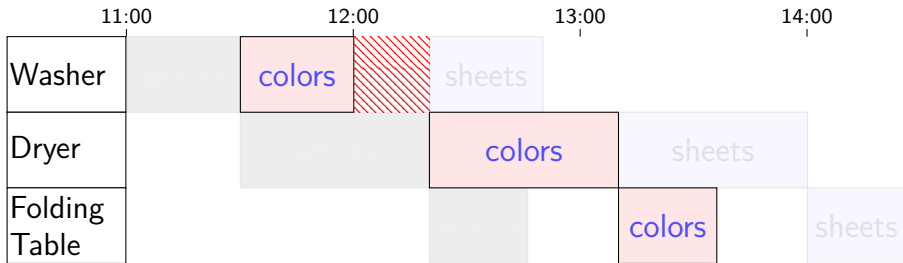
Waste (1)



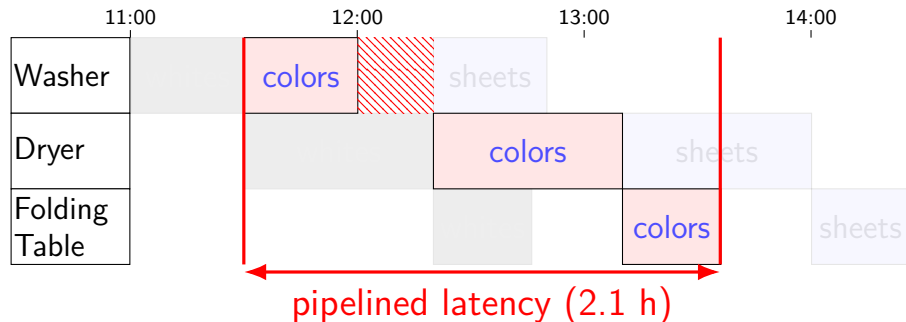
Waste (2)



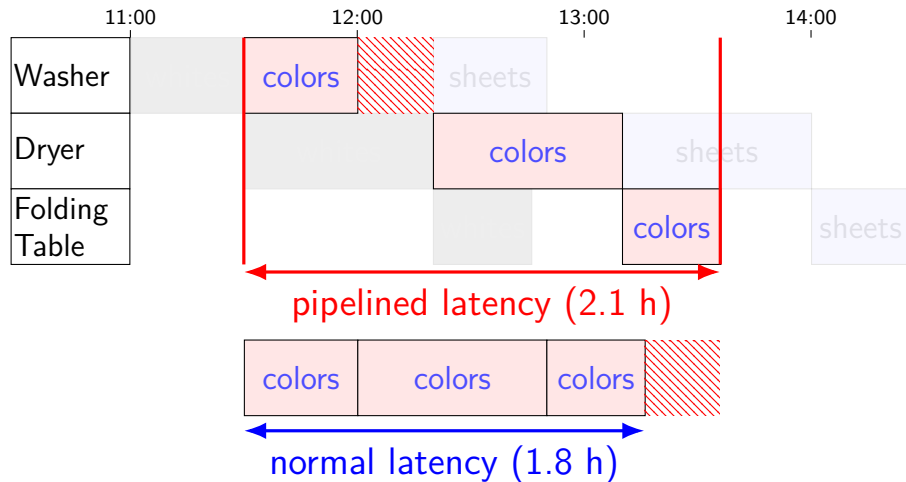
Latency — Time for One



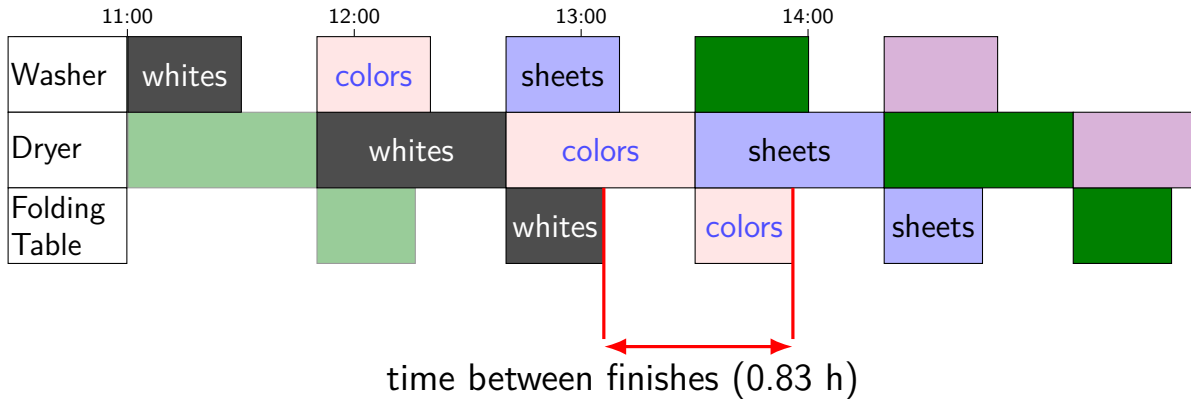
Latency — Time for One



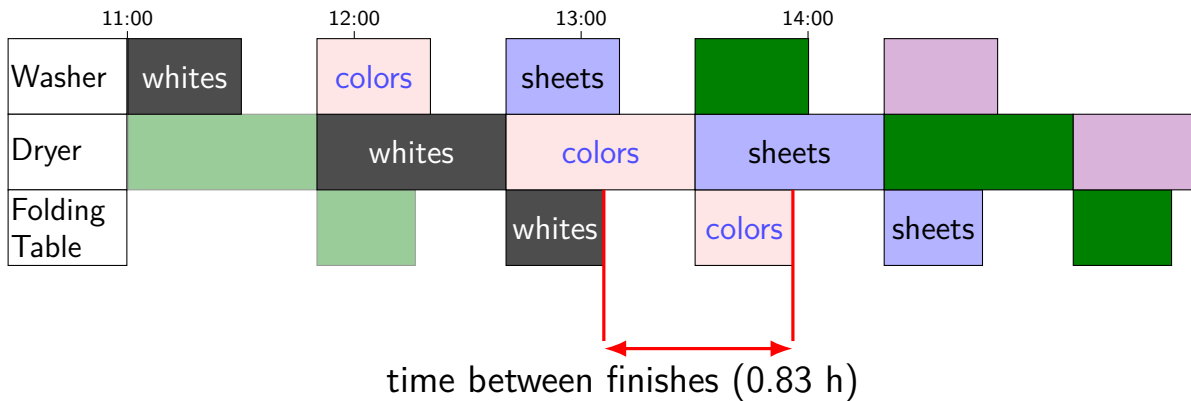
Latency — Time for One



Throughput — Rate of Many

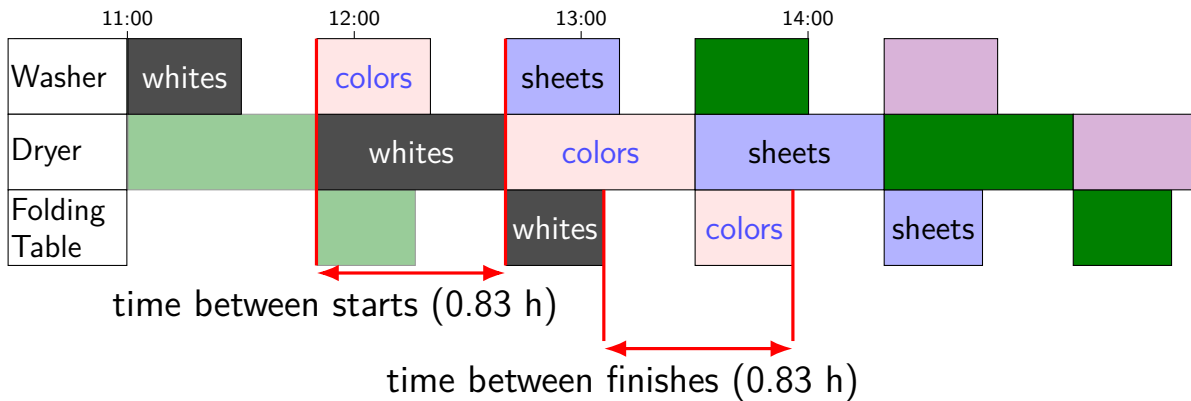


Throughput — Rate of Many



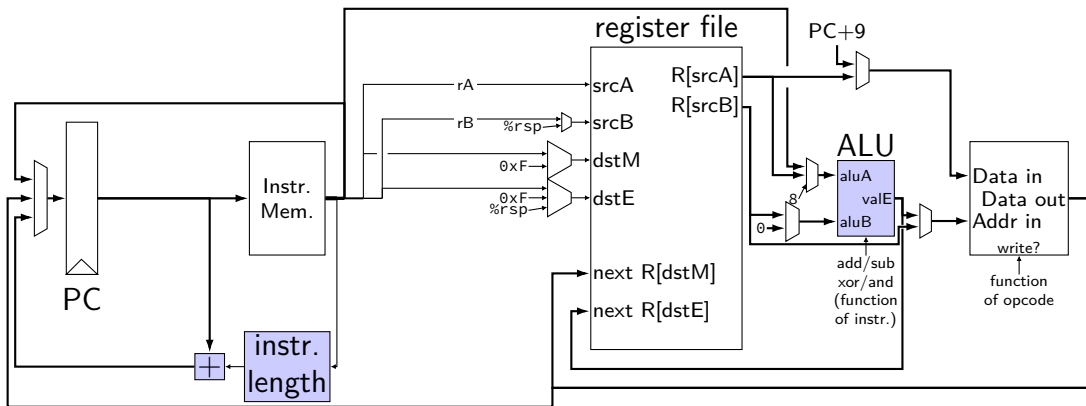
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

Throughput — Rate of Many



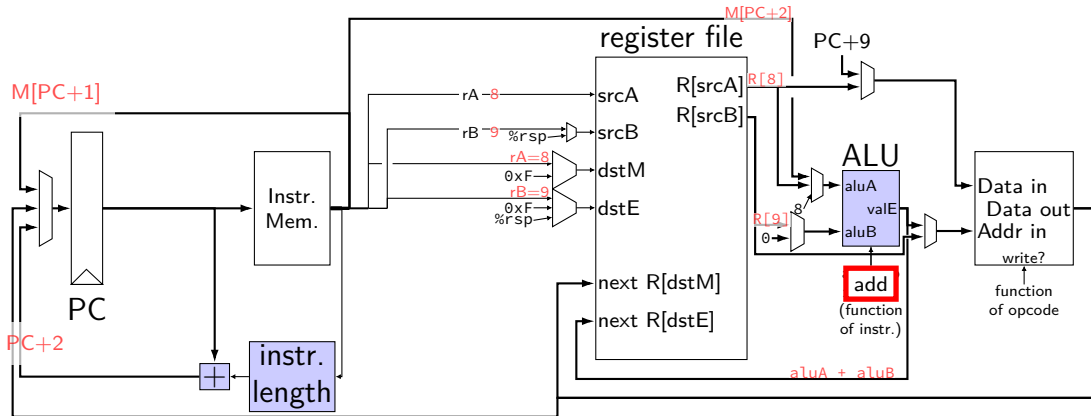
$$\frac{1 \text{ load}}{0.83\text{h}} = 1.2 \text{ loads/h}$$

circuit: setting MUXes



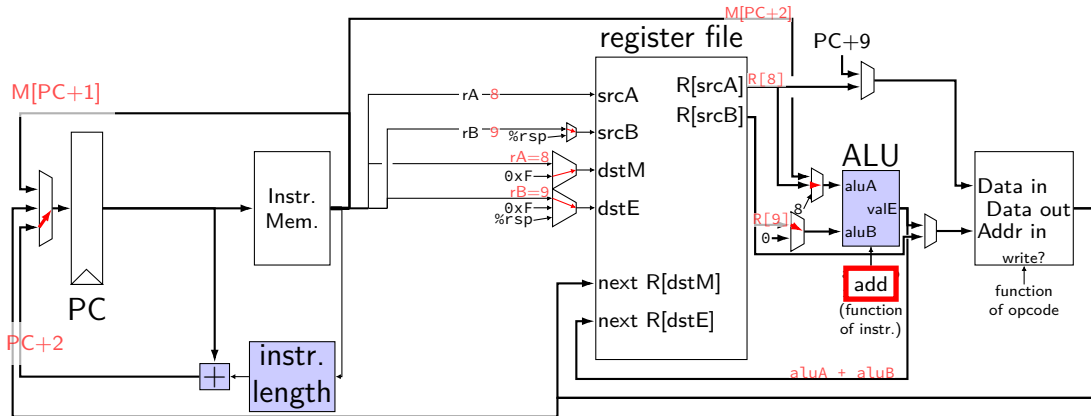
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



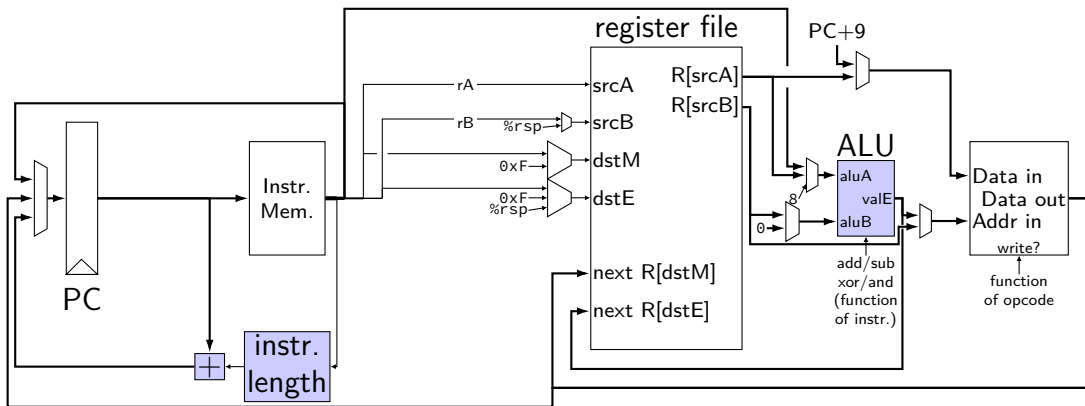
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



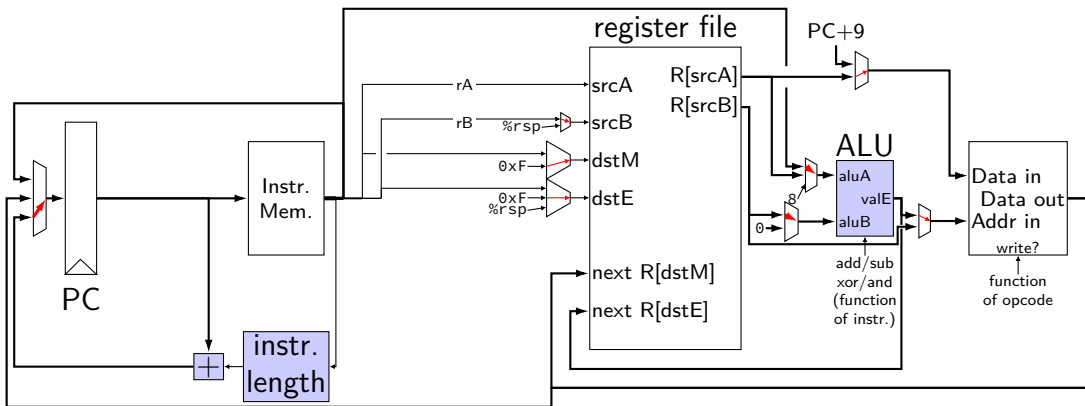
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



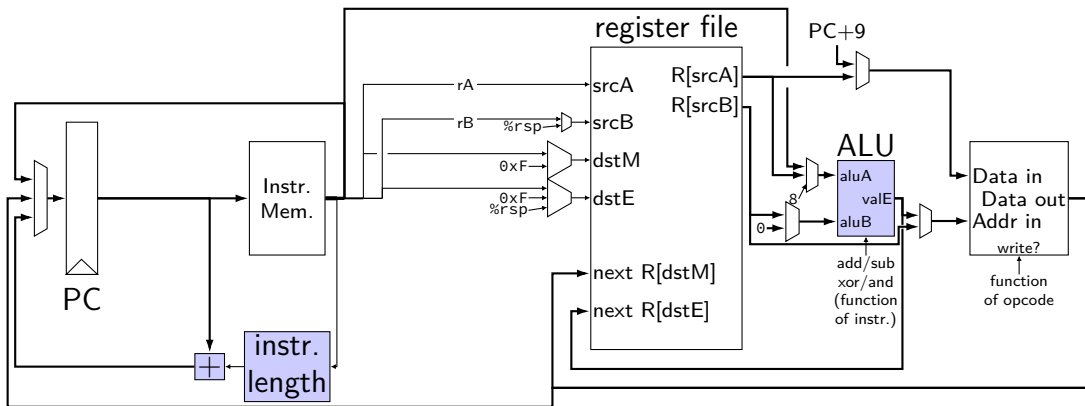
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **rmmovq**?

circuit: setting MUXes



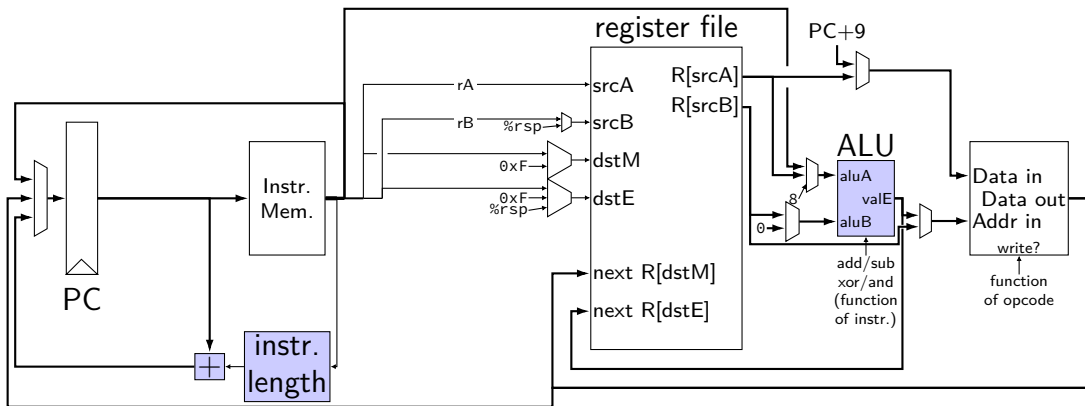
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **rmmovq**?

circuit: setting MUXes



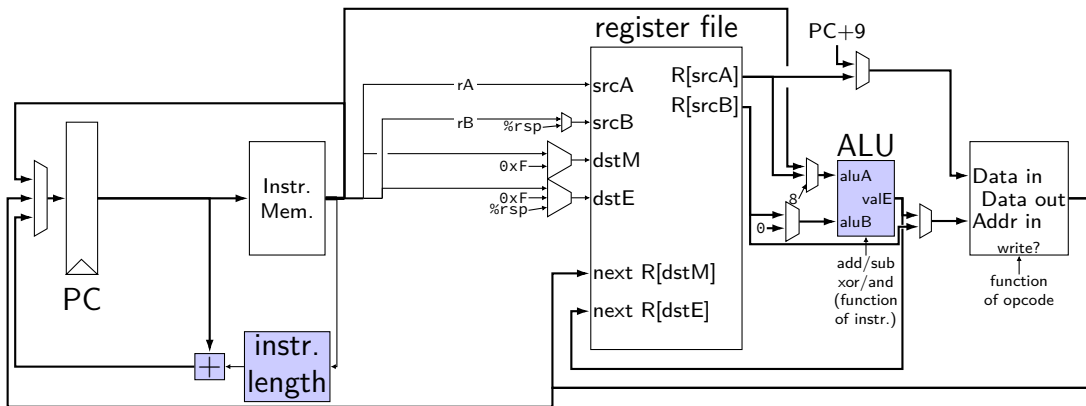
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **irmovq**?

circuit: setting MUXes



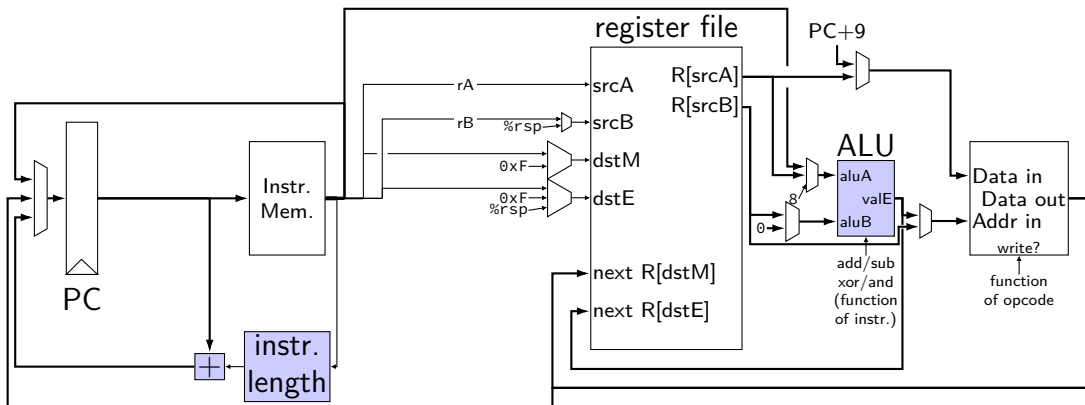
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `mrmovq`?

circuit: setting MUXes



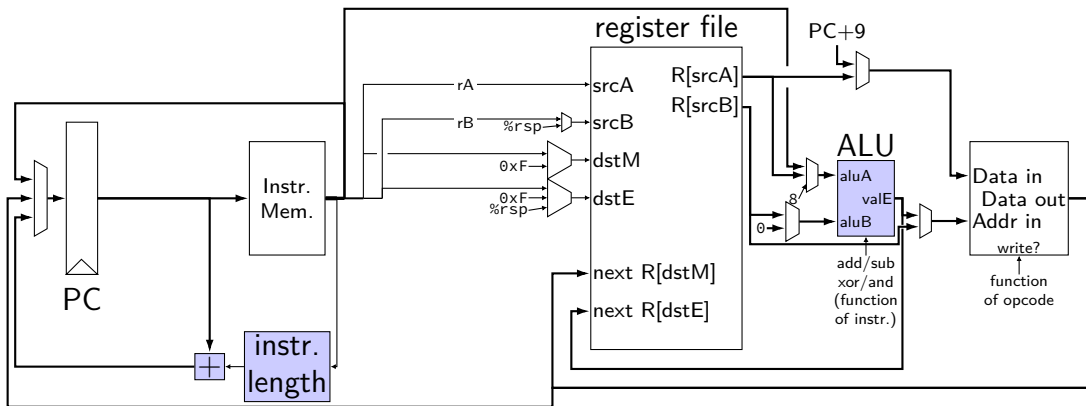
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **jle**?

circuit: setting MUXes



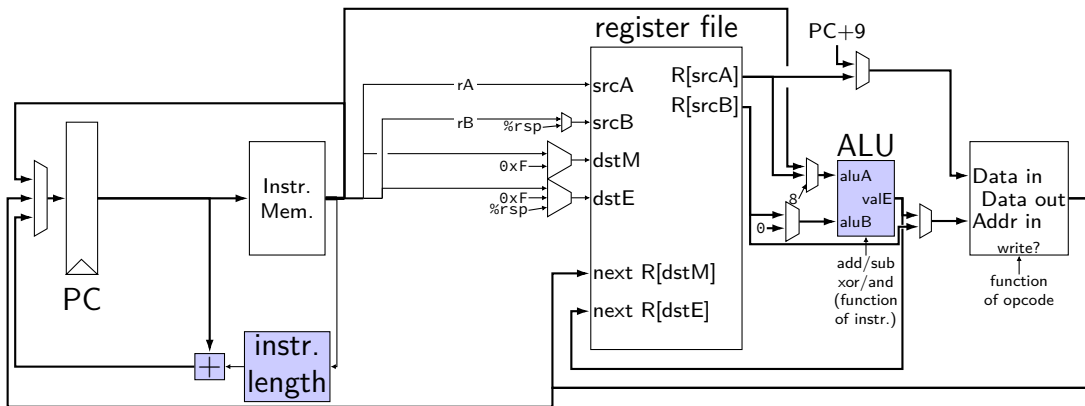
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **cmovle**?

circuit: setting MUXes



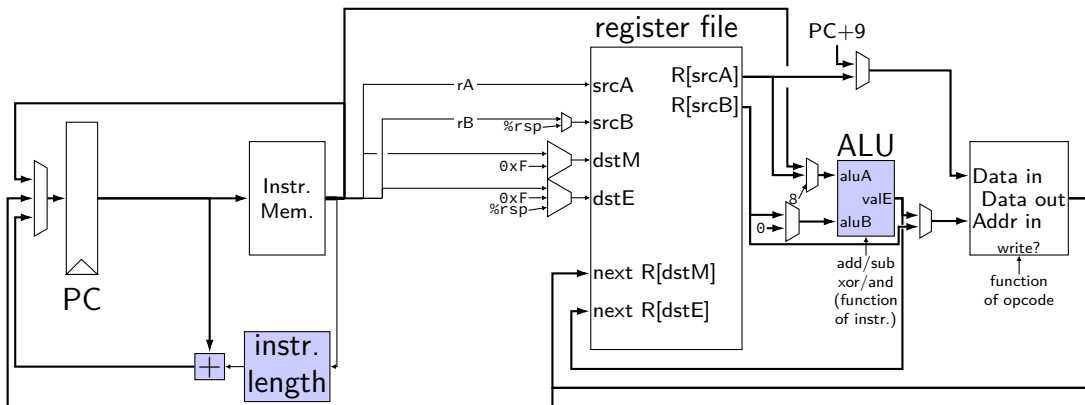
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for **ret**?

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**?