SEQ Continued

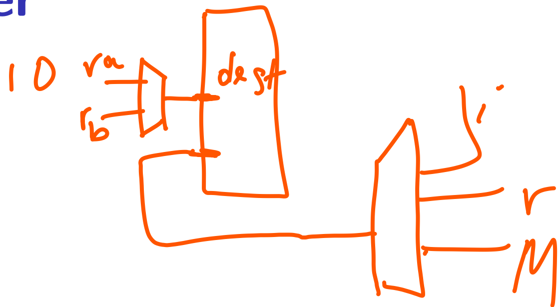
February 23, 2023

exercise: mov-to-register

```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY
```



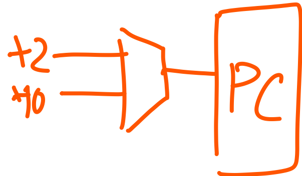
for which of these are we going to need MUXes? before...

A. register file's register number (index) inputs (`reg_srcA`, `reg_srcB`, `reg_dstE`, ...)

B. register file's value inputs (`reg_inputE/M`)

C. PC register's input

D. instruction memory's address input (pc)



mov-to-register CPU

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

Instr.
Mem.

PC

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*

mov-to-register CPU

register file

srcA

srcB

dstE

next R[dstE]

R[srcA]

R[srcB]

PC

Instr.
Mem.

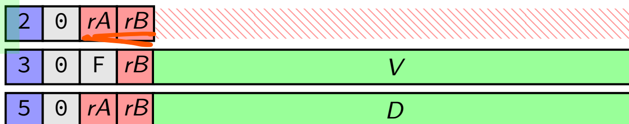
split

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



mov-to-register CPU

register file

srcA
R[srcA]
srcB
R[srcB]
dstE
next R[dstE]

PC

Instr.
Mem.

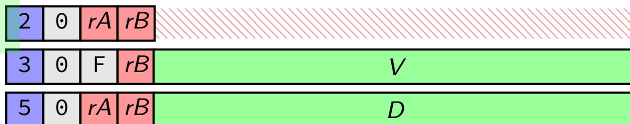
split

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



mov-to-register CPU

register file

srcA

srcB

dstE

next R[dstE]

R[srcA]

R[srcB]

PC

Instr. Mem.

split

Data in

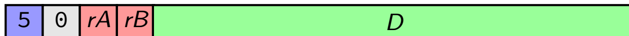
Data out

Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



mov-to-register CPU

register file

PC

Instr. Mem.

split

srcA

srcB

dstE

R[srcA]

R[srcB]

next R[dstE]

MUX

convert opcode

Data in
Data out
Addr in

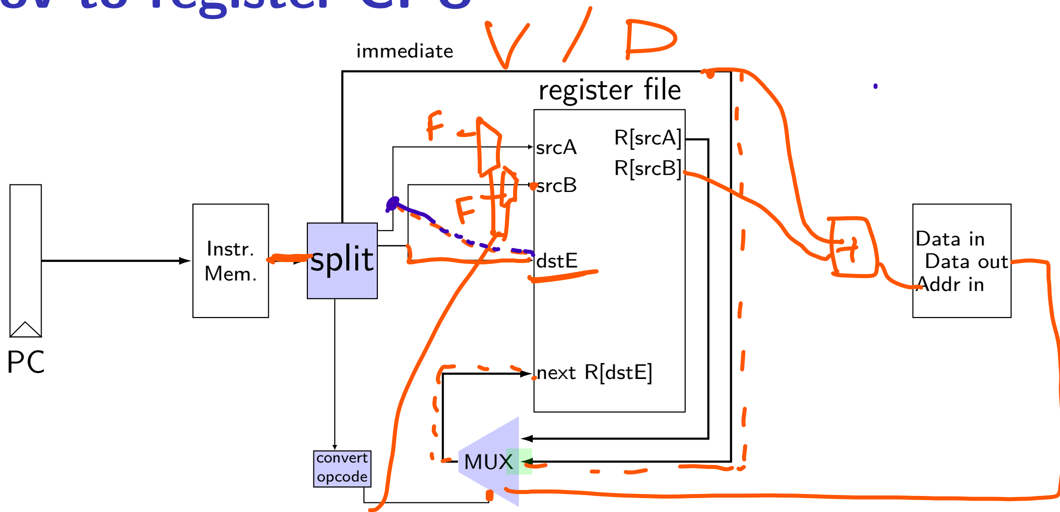
```
reg_inputE = [
   icode == RRMOVQ :
      reg_outputA;
   …
]
```

rrmovq *rA*, *rB*

irmovq *V*, *rB*

mrmovq *D*(*rB*), *rA*

mov-to-register CPU



`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU

immediate

register file

srcA

srcB

dstE

next R[dstE]

R[srcA]

R[srcB]

PC

Instr. Mem.

split

convert opcode

MUX

+ (ALU)

Data in

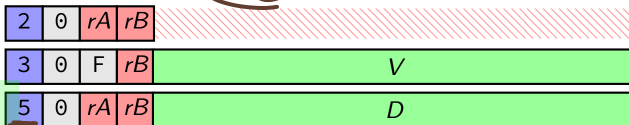
Data out

Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



mov-to-register CPU

immediate

register file

srcA

srcB

dstE

next R[dstE]

R[srcA]

R[srcB]

PC

Instr. Mem.

split

convert opcode

MUX

+ (ALU)

Data in

Data out

Addr in

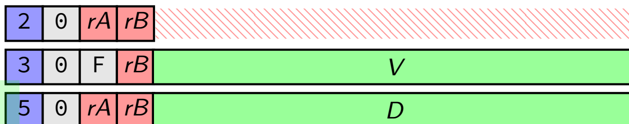
```
mem_addr = reg_outputB + …;
mem_readbit = 1;
```

rE=1

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



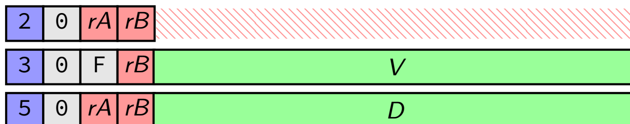
mov-to-register CPU

immediate
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
PC
Instr.
Mem.
split
+2
+10
convert
opcode
MUX
+ (ALU)
Data in
Data out
Addr in

rrmovq *rA*, *rB*

irmovq *V*, *rB*

mrmovq *D*(*rB*), *rA*



simple ISA: mov (all cases)

```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

rmmovq %rXX, 10(%rYY)
```

mov CPU

immediate

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

PC
+2
+10
Instr.
Mem.
split
convert
opcode
MUX
Data in
Data out
Addr in
+

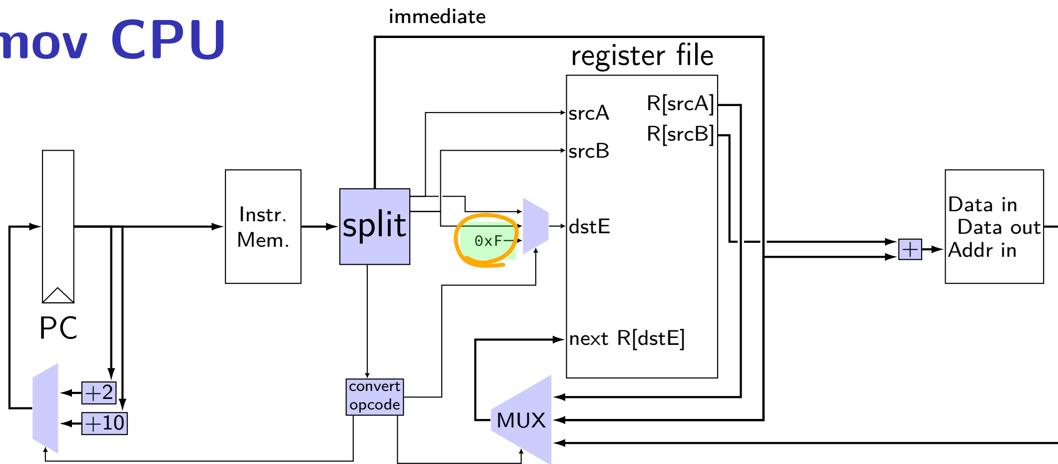
`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*

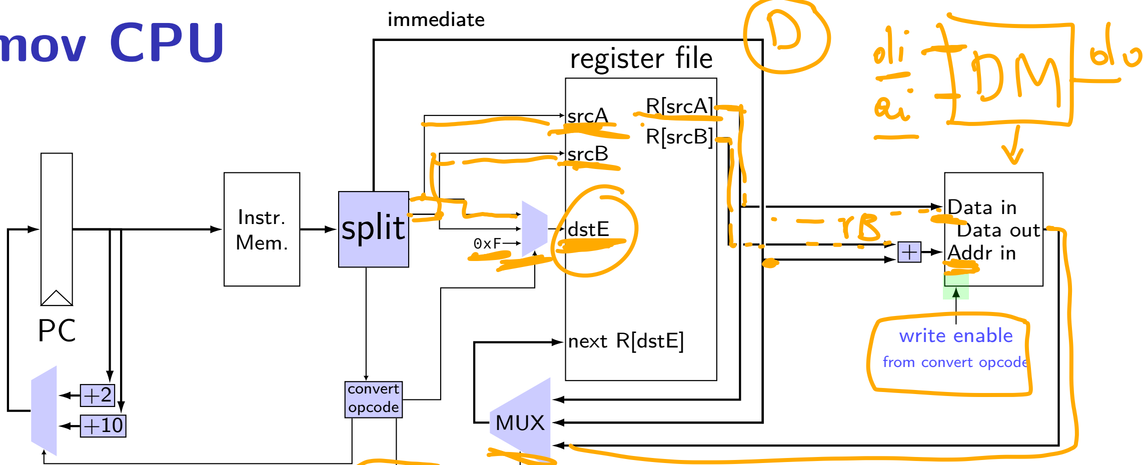
`rmmovq` *rA*, *D*(*rB*)

mov CPU



instruction					
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA*, *D*(*rB*)	4	0	*rA*	*rB*	*D*

mov CPU



`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA*, *D*(*rB*)	4	0	*rA*	*rB*	*D*

mov CPU

immediate

register file

srcA

srcB

R[srcA]

R[srcB]

dstE

next R[dstE]

PC

Instr.
Mem.

split

0xF

+2

+10

convert
opcode

MUX

Data in
Data out
Addr in

write enable
from convert opcode

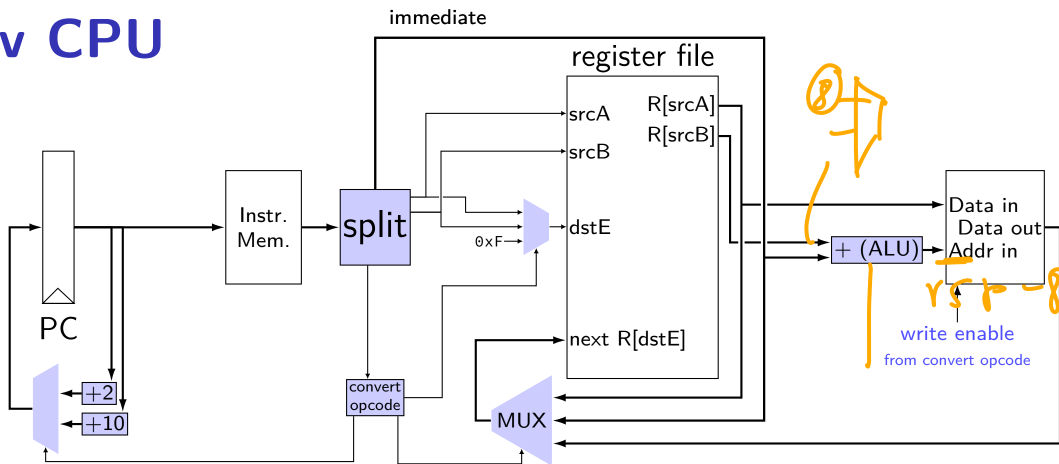
`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

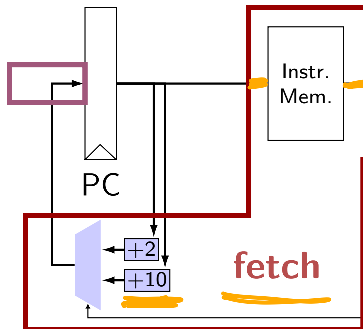
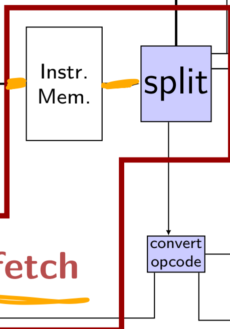
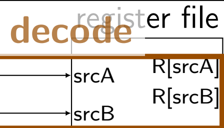
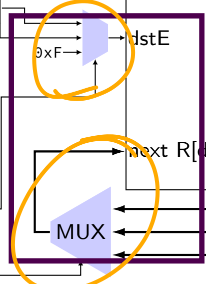
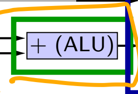
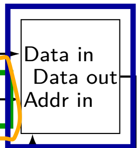
`mrmovq` *D*(*rB*), *rA*

`rmmovq` *rA*, *D*(*rB*)

mov CPU



mov CPU

Stages

conceptual division of instruction:

fetch — read instruction memory, split instruction, compute length

decode — read register file

execute — arithmetic (including of addresses)

memory — read or write data memory

write back — write to register file

PC update — compute next value of PC

stages and time

fetch / decode / execute / memory / write back / PC update

Order when these events happen `pushq %rax` instruction:

1. instruction read
2. memory changes
3. `%rsp` changes
4. PC changes

Hint: recall how registers, register files, memory works

a. 1; then 2, 3, and 4 in any order
b. 1; then 2, 3, and 4 at almost the same time
c. 1; then 2; then 3; then 4
d. 1; then 3; then 2; then 4
e. 1; then 2; then 3 and 4 at almost the same time
f. something else

SEQ: instruction fetch

read instruction memory at PC

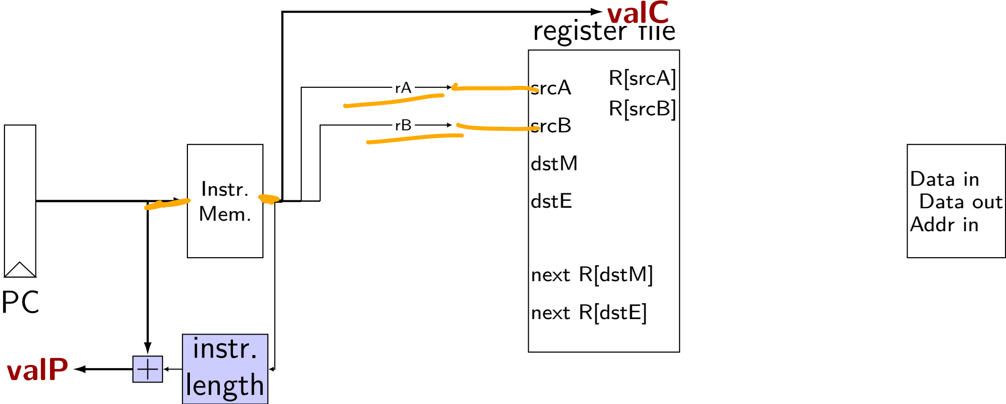
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

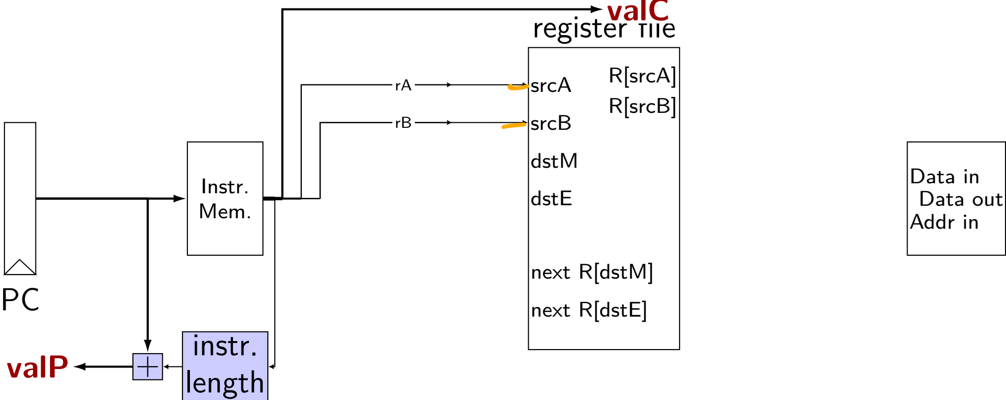


SEQ: instruction “decode”

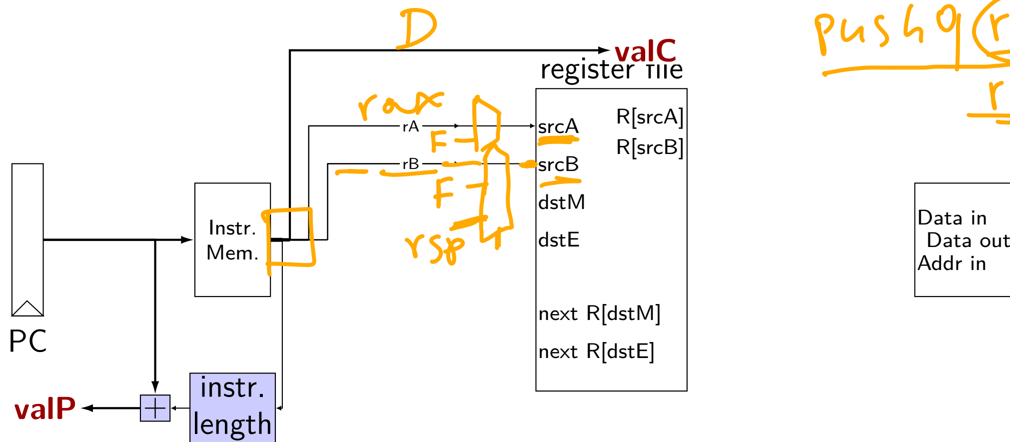
read registers

- valA, valB — register values

instruction decode (1)



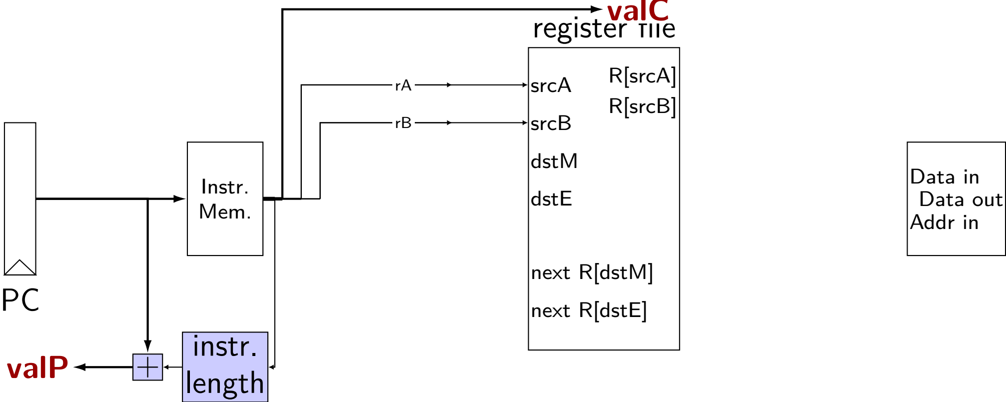
instruction decode (1)



exercise: for which instructions would there be a problem ?

`nop`, `addq`, `mrmovq`, `rmmovq`, `jmp`, `pushq`

instruction decode (1)



SEQ: srcA, srcB

always read rA, rB?

Problems:

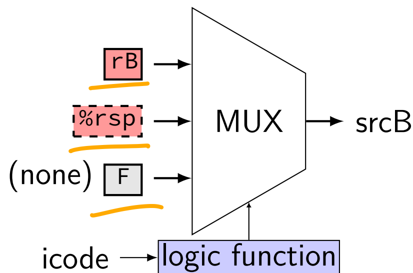
- push rA
- pop
- call
- ret

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

MUX controlled by icode

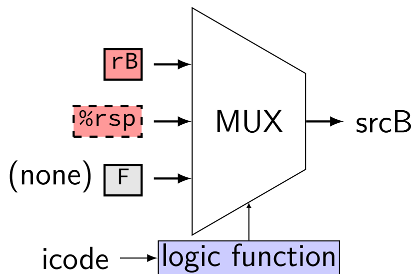
SEQ: possible registers to read

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



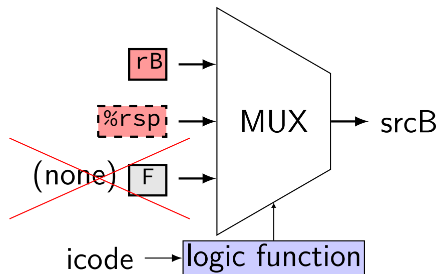
SEQ: possible registers to read

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`

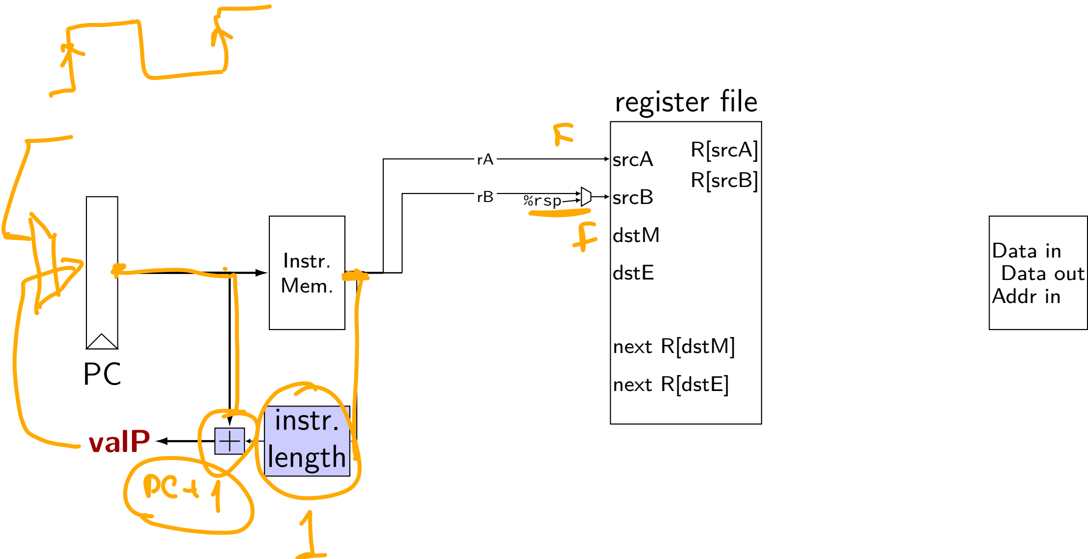


SEQ: possible registers to read

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)

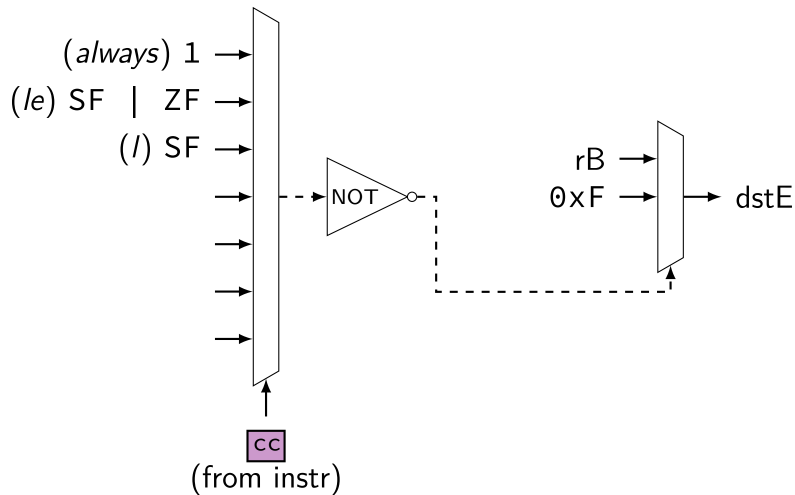
- valE — ALU output

read prior condition codes

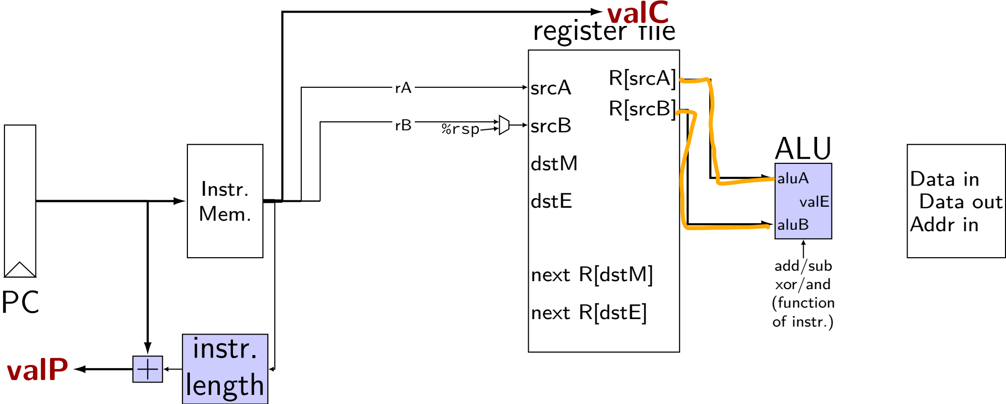
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

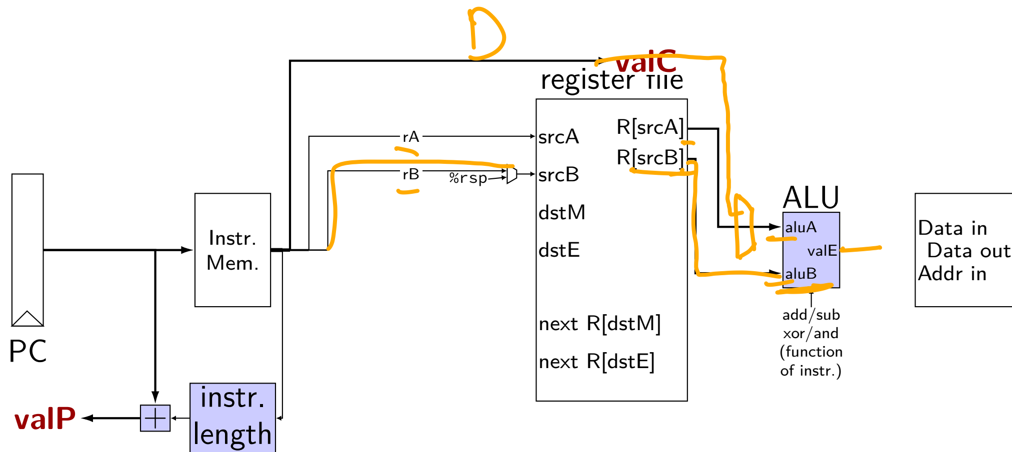
using condition codes: cmov



execute (1)



execute (1)



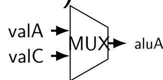
exercise: which of these instructions would there be a problem ?

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

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)



- `mrmovq`
- `rmmovq`

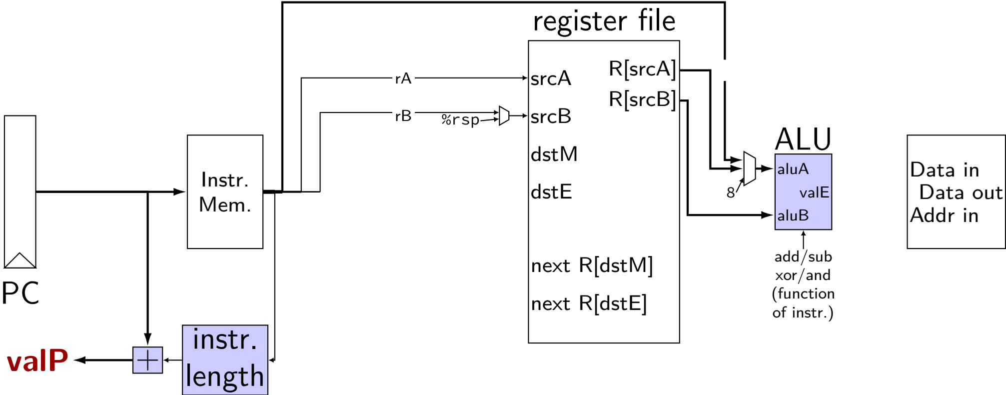
no, constants: (rsp +/- 8)

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

extra signals: aluA, aluB

computed ALU input values

execute (2)

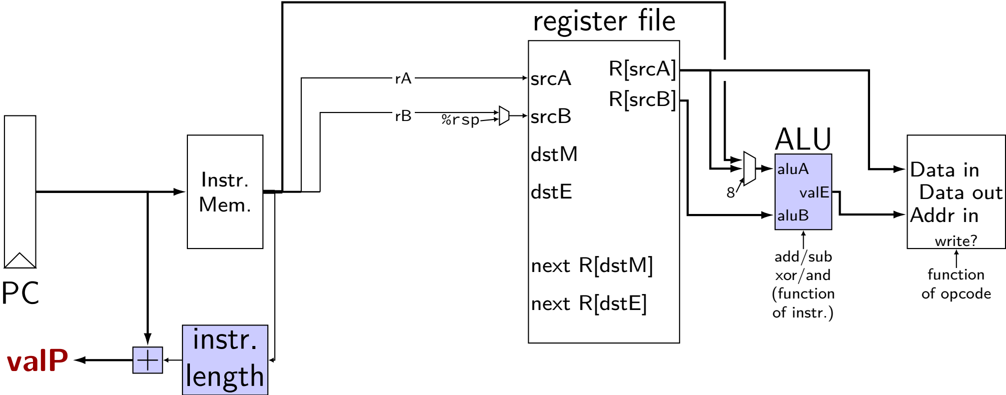


SEQ: Memory

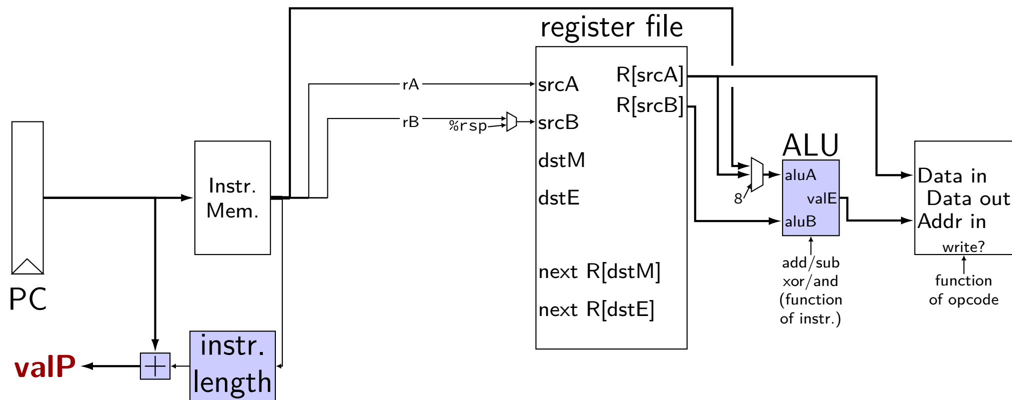
read or write data memory

- valM — value read from memory (if any)

memory (1)



memory (1)



exercise: which of these instructions would there be a problem ?

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

SEQ: control signals for memory

read/write — read enable? write enable?

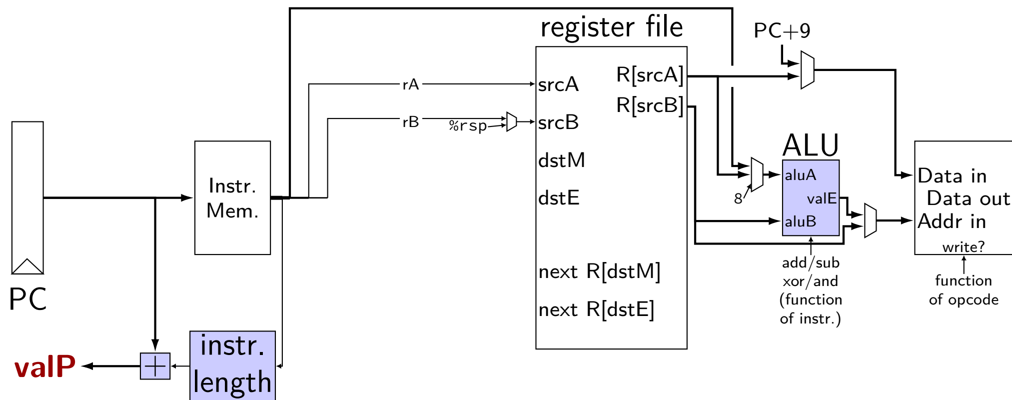
Addr — address

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

Data — value to write

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

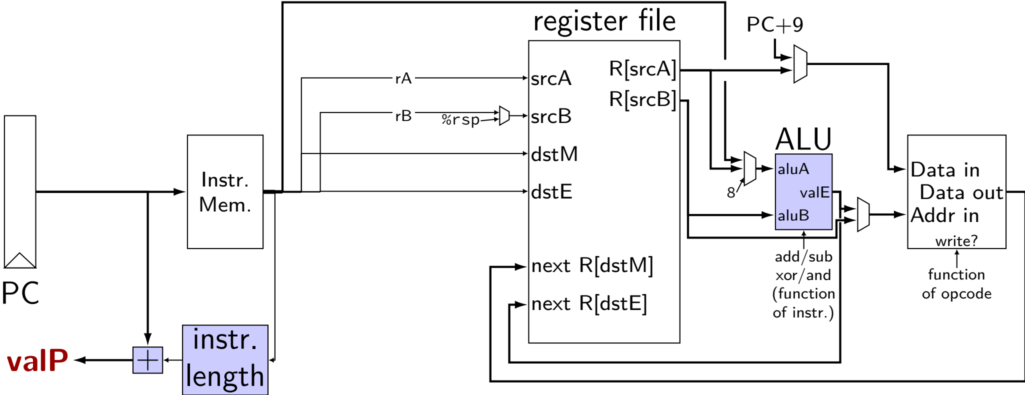
memory (2)



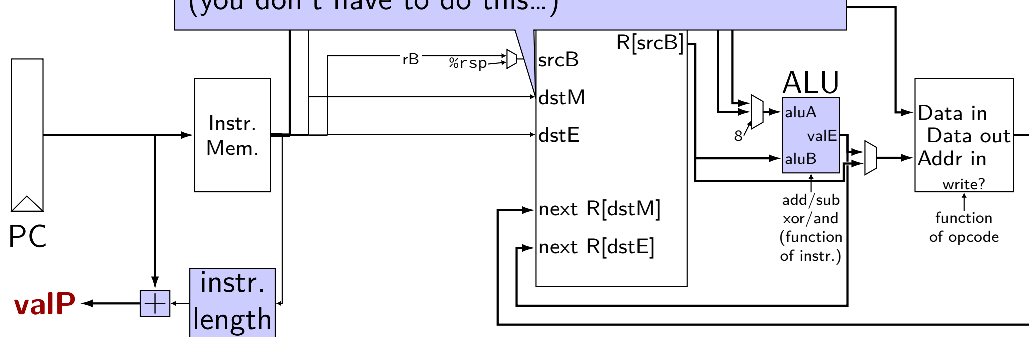
SEQ: write back

write registers

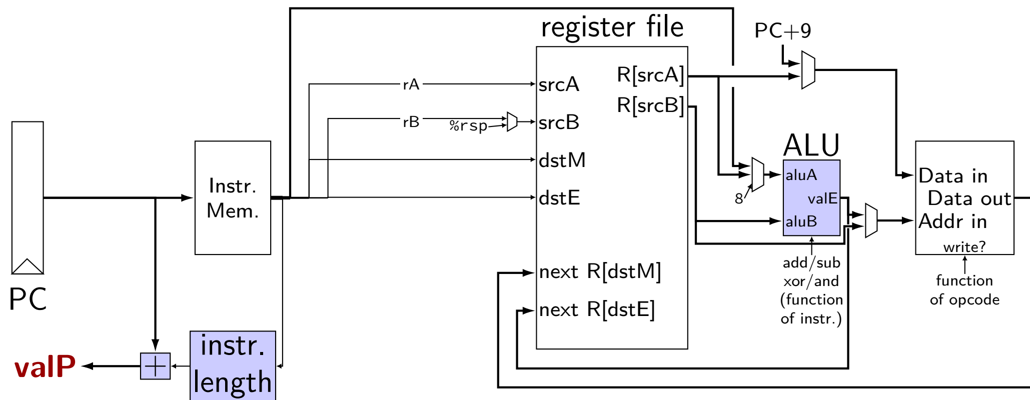
write back (1)



write back (1)



write back (1)



exercise: which of these instructions would there be a problem ?

`nop`, `irmovq`, `mrmovq`, `rmmovq`, `addq`, `popq`

SEQ: control signals for WB

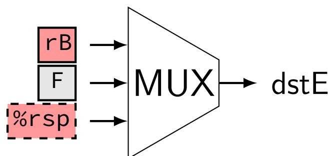
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

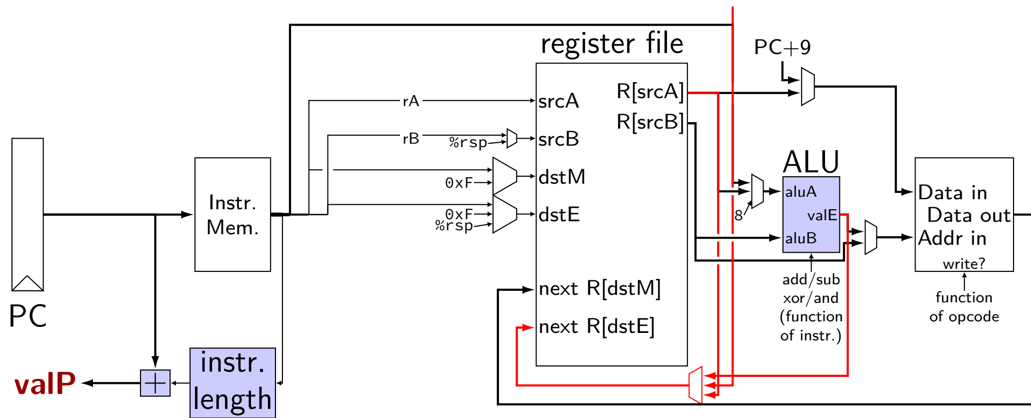
two register numbers

dstM, dstE

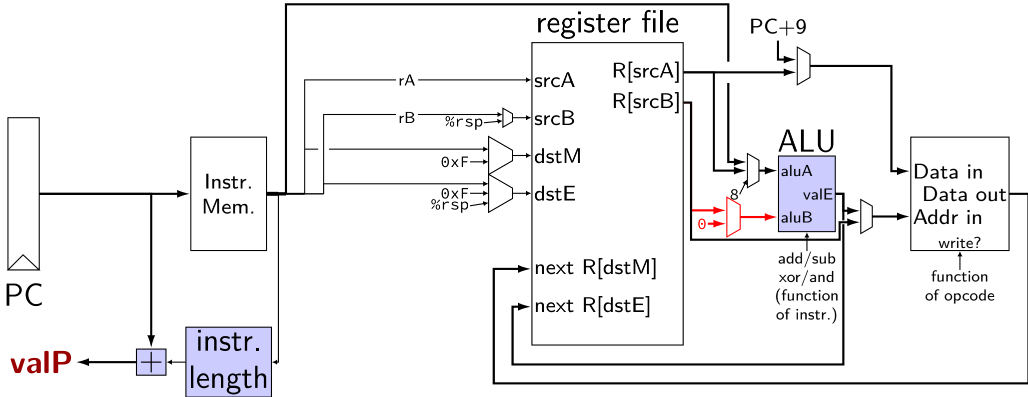
write disable — use dummy register number `0xF`



write back (2a)



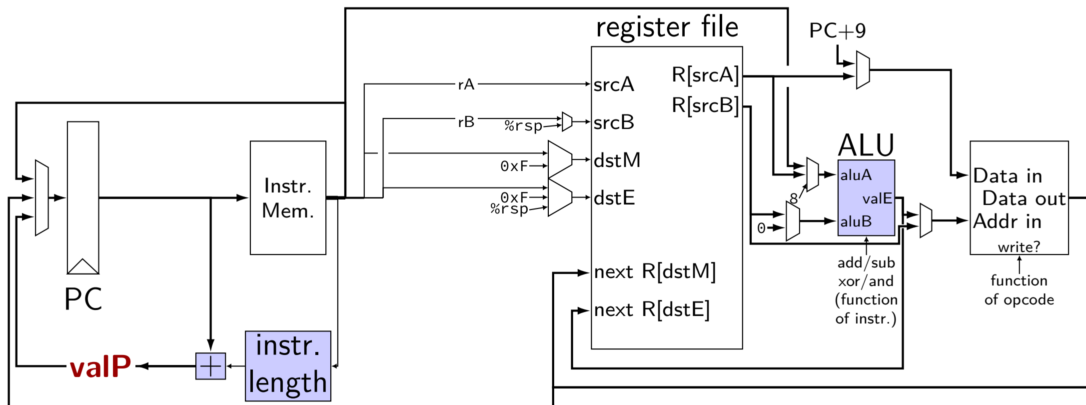
write back (2b)



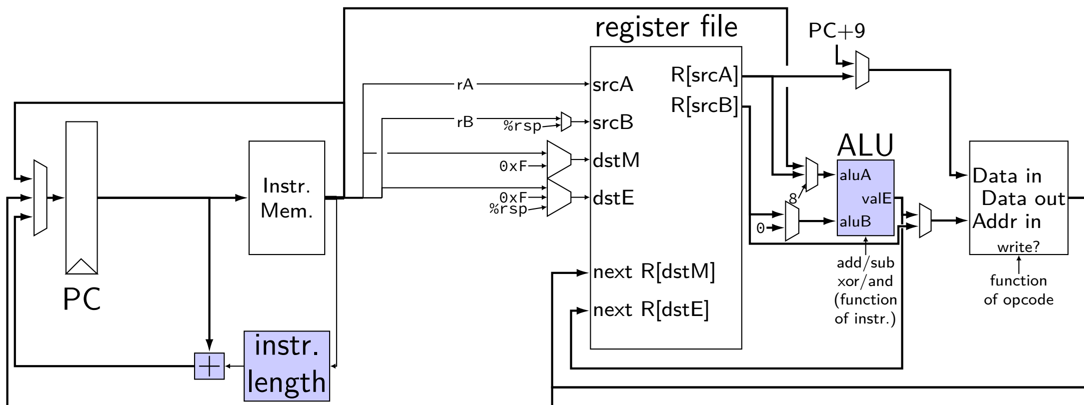
SEQ: Update PC

- choose value for PC next cycle (input to PC register)
  - usually valP (following instruction)
  - exceptions: `call`, `j`*CC*, `ret`

PC update

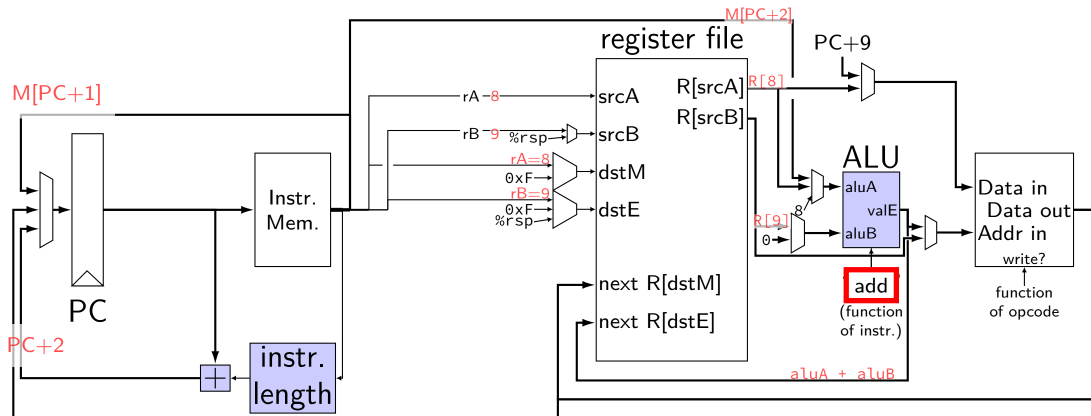


circuit: setting MUXes



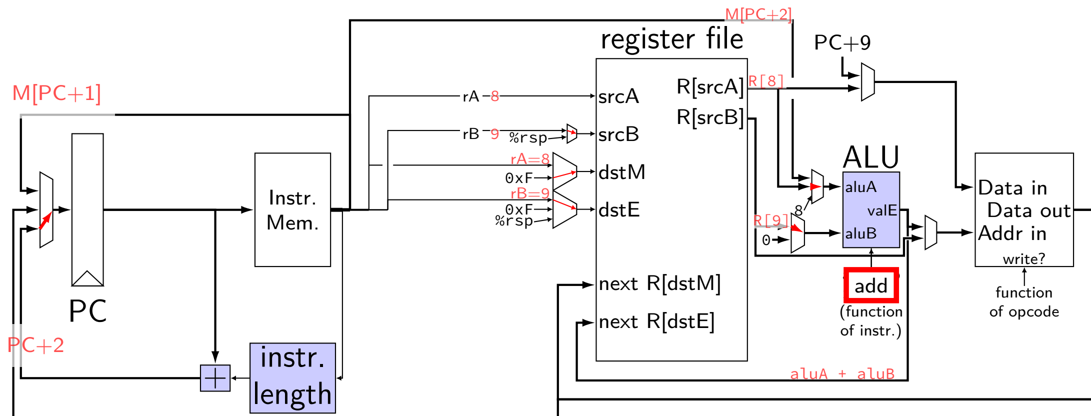
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



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

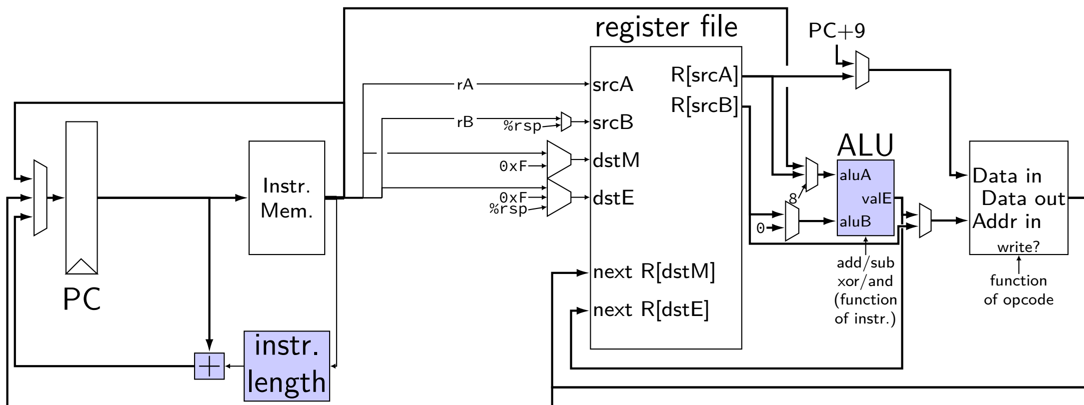
circuit: setting MUXes



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

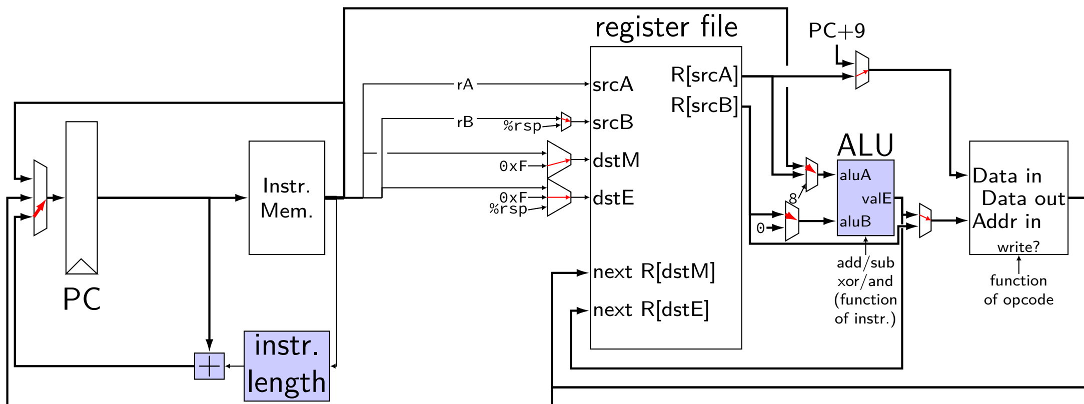
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



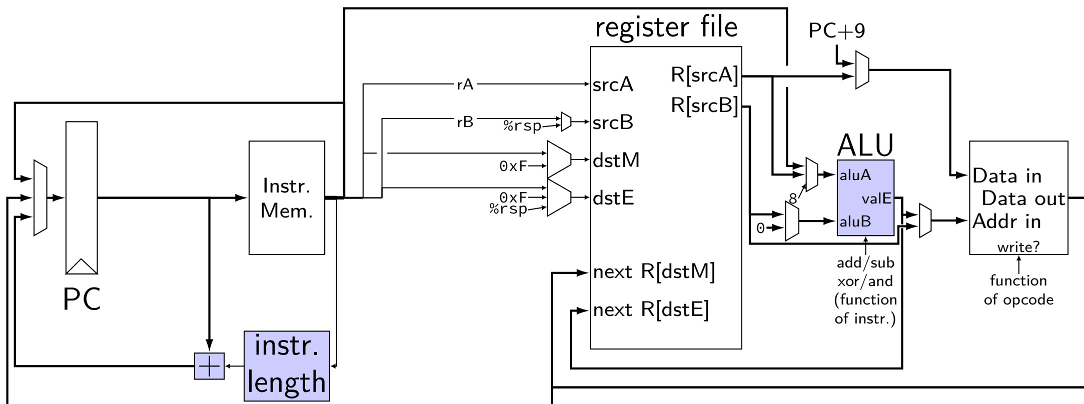
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for `rmmovq`?

circuit: setting MUXes



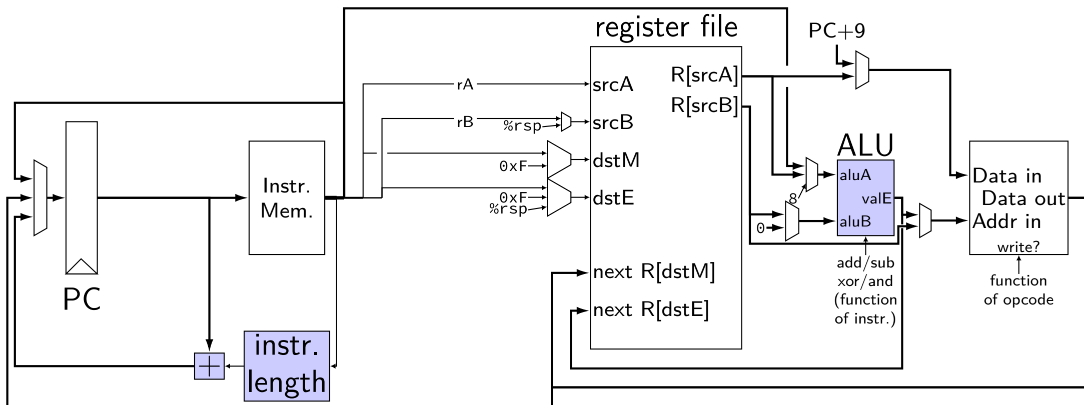
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for `rmmovq`?

circuit: setting MUXes



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

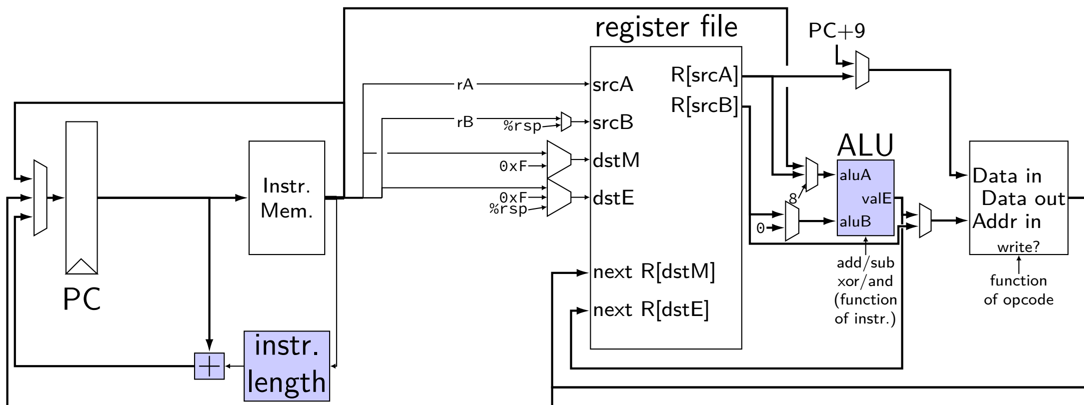
circuit: setting MUXes



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

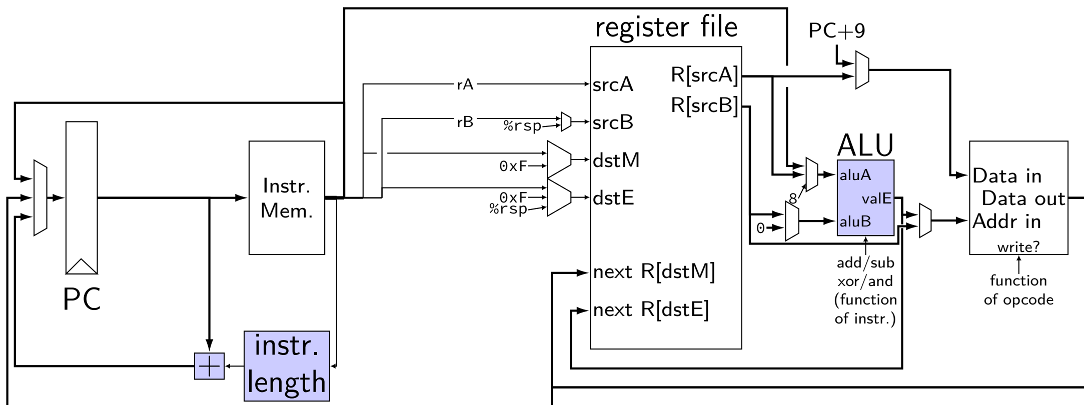
Exercise: what do they select for `mrmovq`?

circuit: setting MUXes



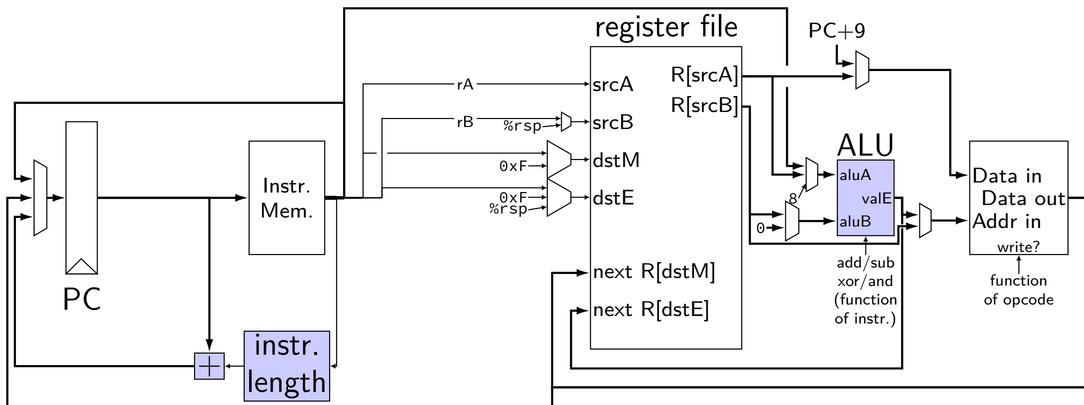
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `jle`?

circuit: setting MUXes



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

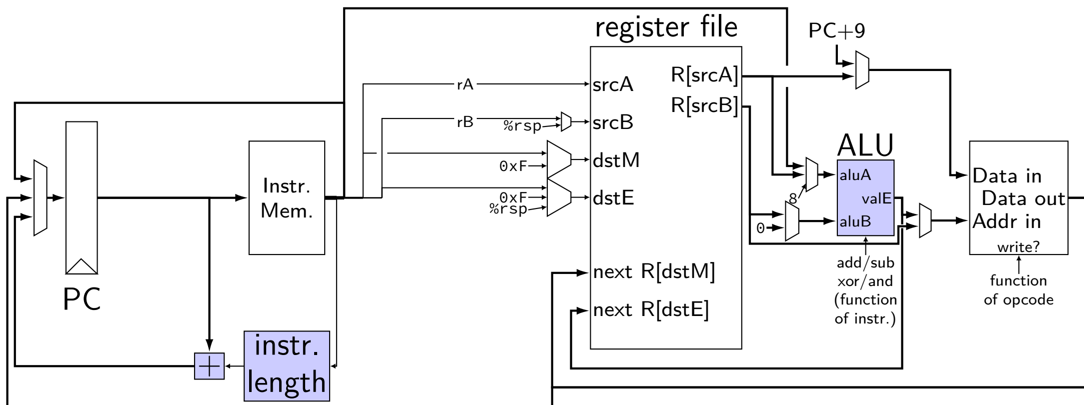
circuit: setting MUXes



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

Exercise: what do they select for `ret`?

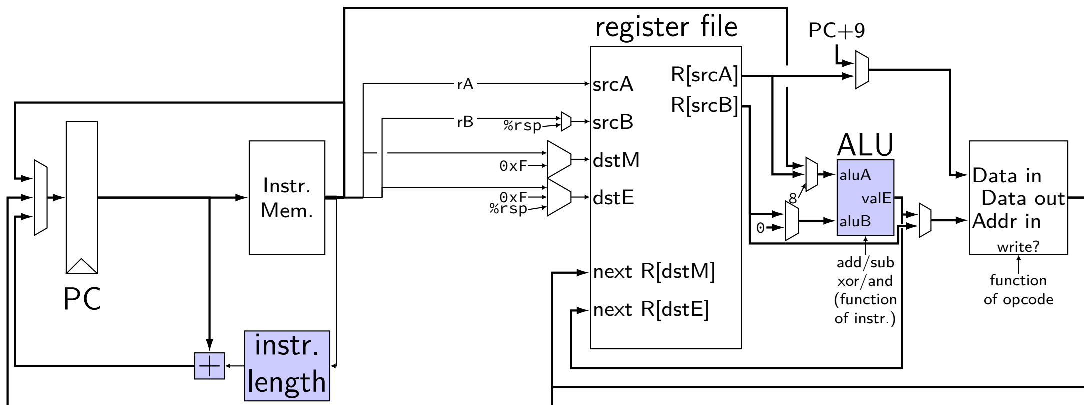
circuit: setting MUXes



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`?

backup slides

comparing to yis

```
$ ./hclrs nopjmp_cpu.hcl nopjmp.yo
...
...
+--------------------- (end of halted state) ---------------------------+
Cycles run: 7
$ ./tools/yis nopjmp.yo
Stopped in 7 steps at PC = 0x1e.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:

Changes to memory:
```

HCLRS summary

declare/assign values to wires

MUXes with

```
[ test1: value1; test2: value2; 1: default; ]
```

register banks with **`register iO`**:

next value on `i_name`; current value on `O_name`

fixed functionality

register file (15 registers; 2 read + 2 write)
memories (data + instruction)
Stat register (start/stop/error)