# SEQ part 3

## last time

HCLRS built-in register file

- inputs for register indices to read+write
- reads as indices received
- writes occur at rising edge of clock (but setup earlier)
- REG_NONE = 0xF to “not write”

HCLRS built-in data memory

- accesses same data as instruction memory
- read like instruction memory
- writes occur at rising edge of clock (but setup earlier)
- read+write enable signals

processor stages

- conceptual division of processor’s work
- actual ordering: components compute as received + write at rising edge

stages: fetch / decode / execute / memory / writeback / PC update

## 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

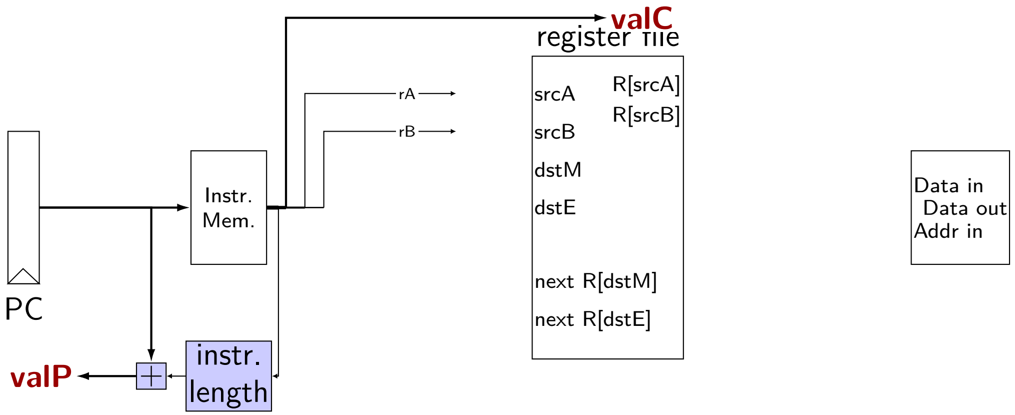
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

# instruction fetch

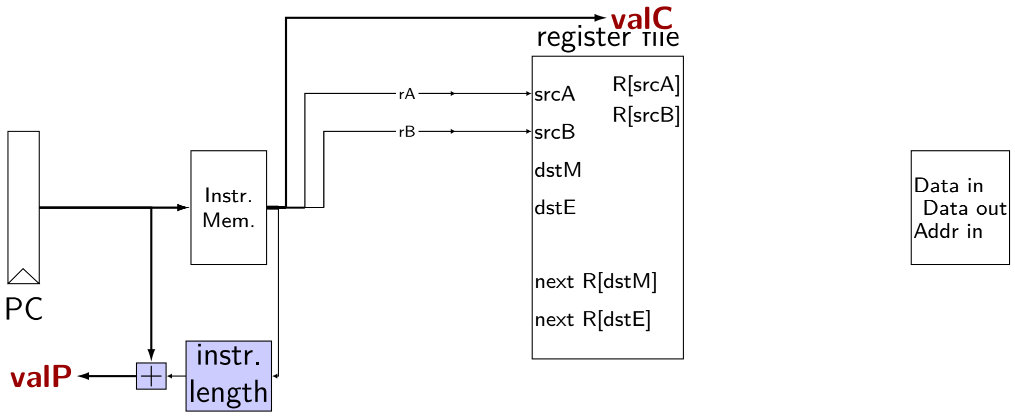


## SEQ: instruction “decode”

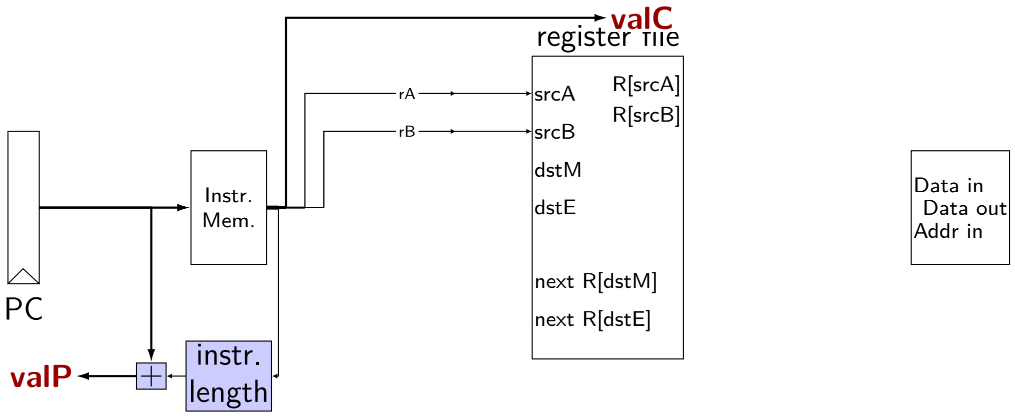
read registers

- valA, valB — register values

# instruction decode (1)



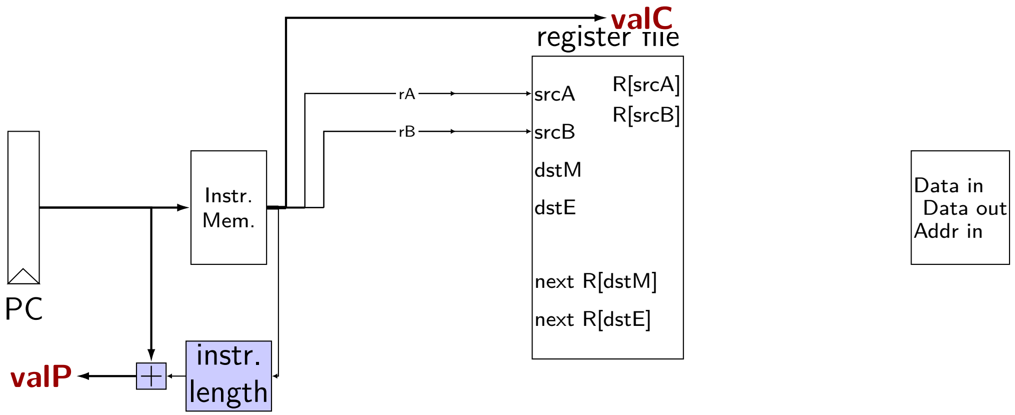
# instruction decode (1)



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:

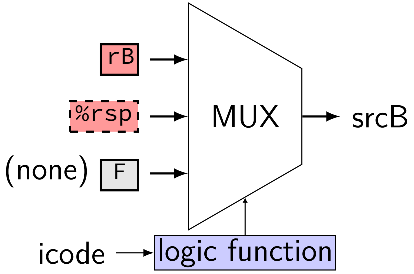
- push rA
- pop
- call
- ret

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

MUX controlled by icode

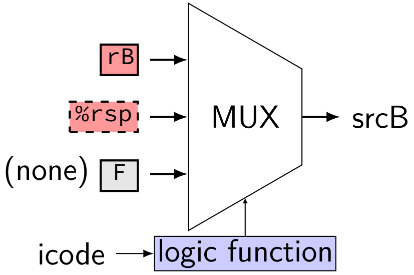
# SEQ: possible registers to read

| 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 |



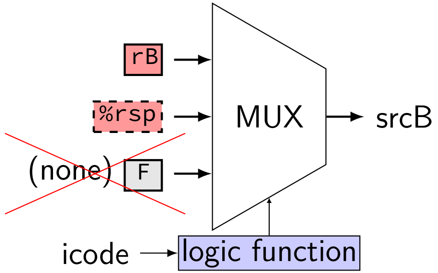
## SEQ: possible registers to read

| 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` |

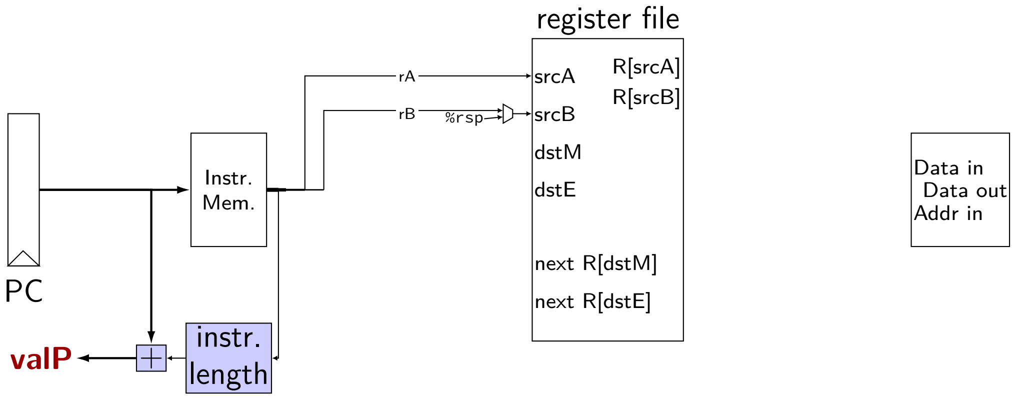


## SEQ: possible registers to read

| 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)

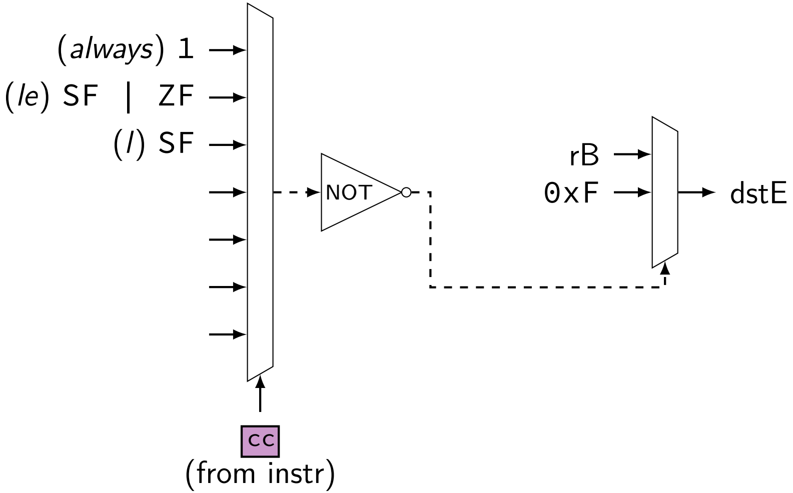
- valE — ALU output

read prior condition codes

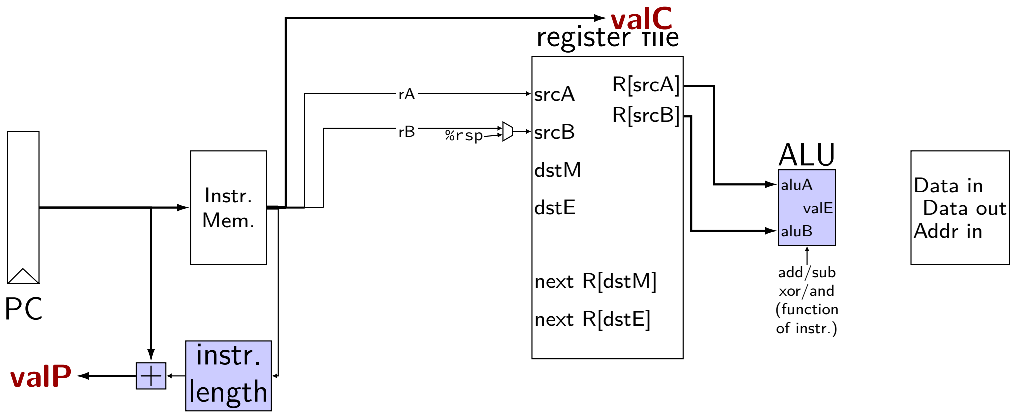
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

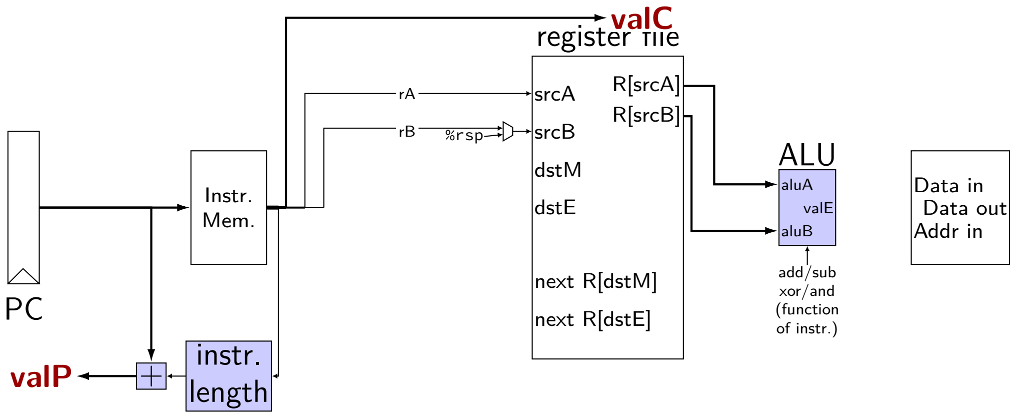
# using condition codes: cmov



# execute (1)


valC
register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
next R[dstM]
next R[dstE]
R[srcA]
R[srcB]
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 would there be a problem ?

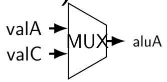
`nop`, `addq`, `mrmovq`, `popq`, `call`,

# SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

- `mrmovq`
- `rmmovq`



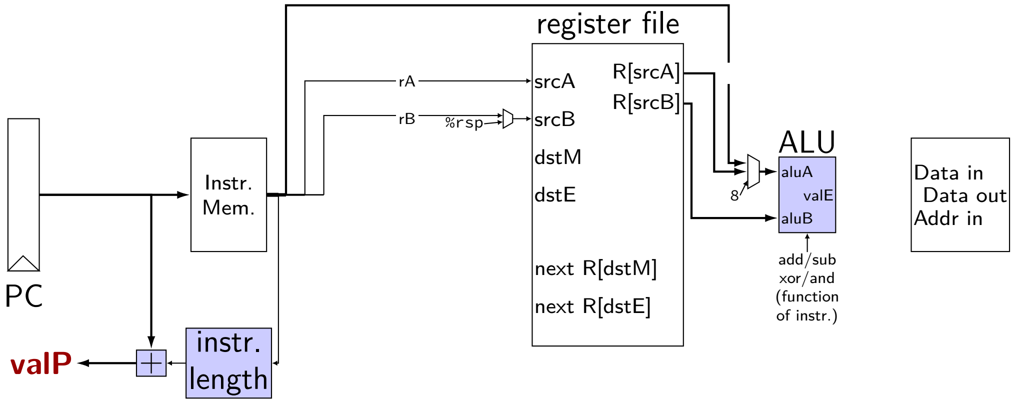
no, constants: (rsp +/- 8)

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

extra signals: aluA, aluB

- computed ALU input values

## execute (2)

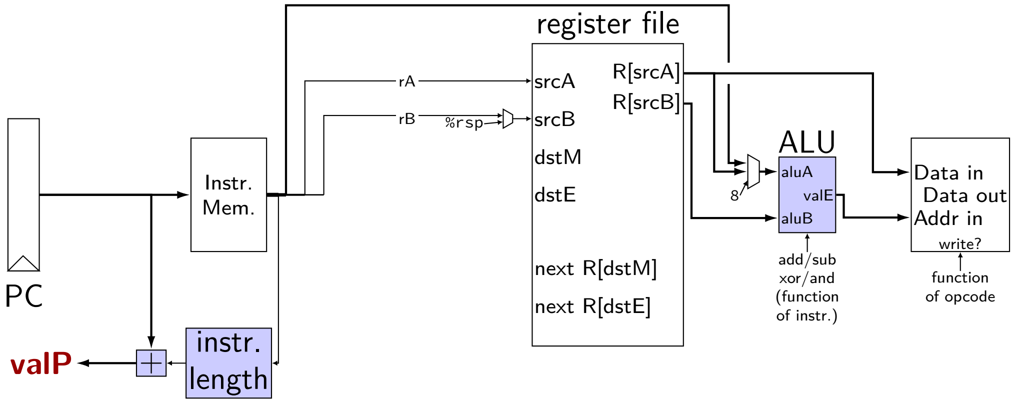

register file
srcA
srcB
dstM
dstE
next R[dstM]
next R[dstE]
R[srcA]
R[srcB]
rA
rB
%rsp
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
PC
Instr.
Mem.
instr.
length
valP

## SEQ: Memory

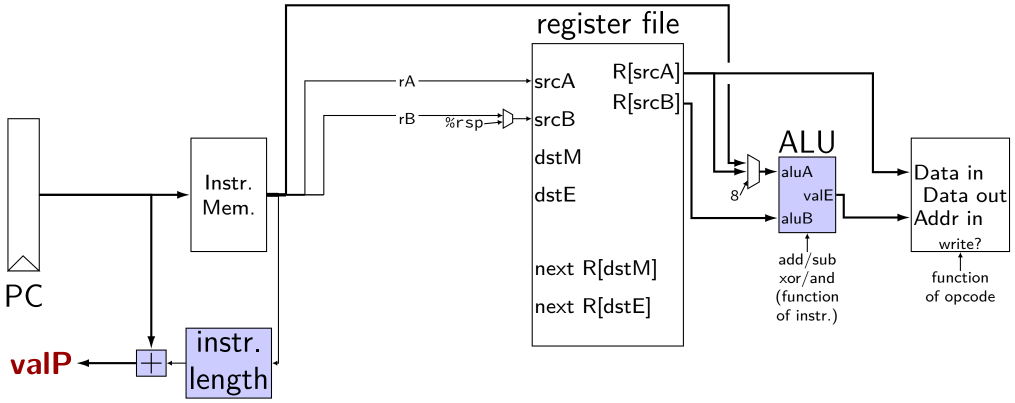
read or write data memory

- valM — value read from memory (if any)

# memory (1)


register file
PC
Instr.
Mem.
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
valP
instr.
length

## 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?

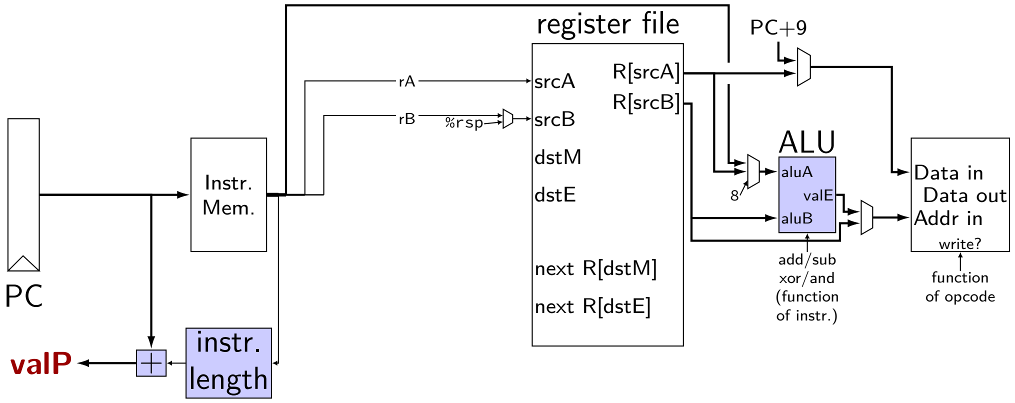
Addr — address

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

Data — value to write

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

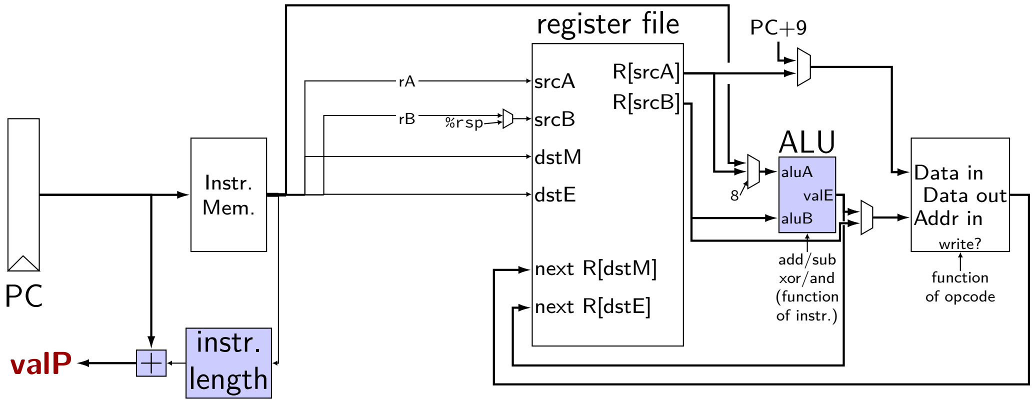
# memory (2)


register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
next R[dstM]
next R[dstE]
R[srcA]
R[srcB]
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

## SEQ: write back

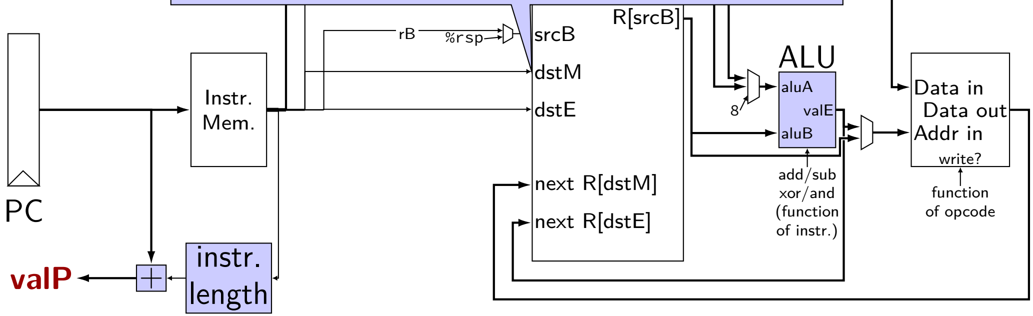
write registers

# write back (1)

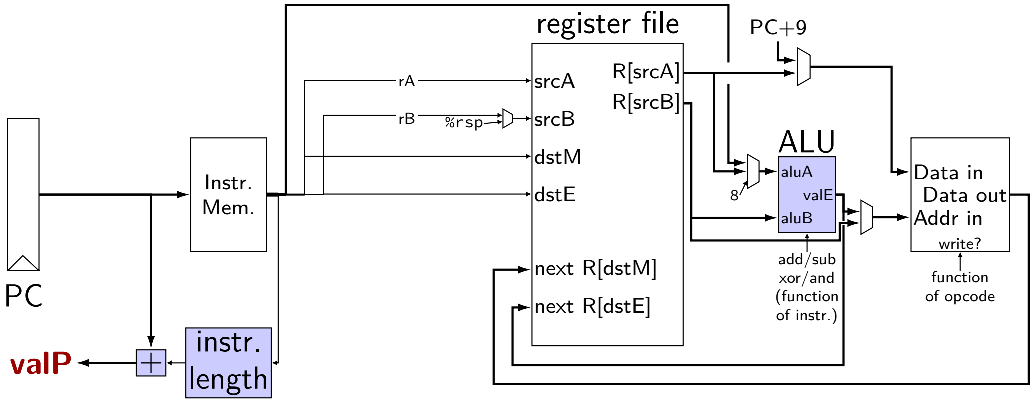

register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
write?
function
of opcode
PC
Instr.
Mem.
valP
instr.
length

# 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...)



# write back (1)



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

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

## SEQ: control signals for WB

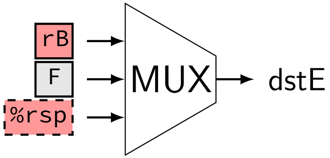
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

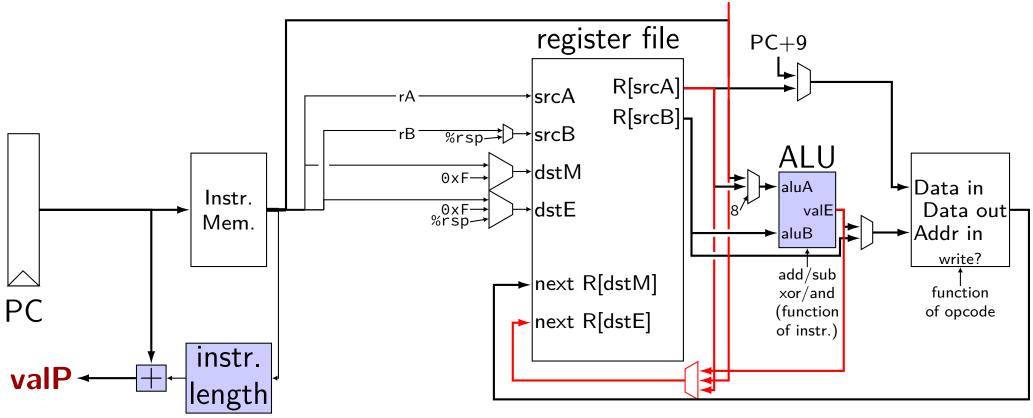
two register numbers

dstM, dstE

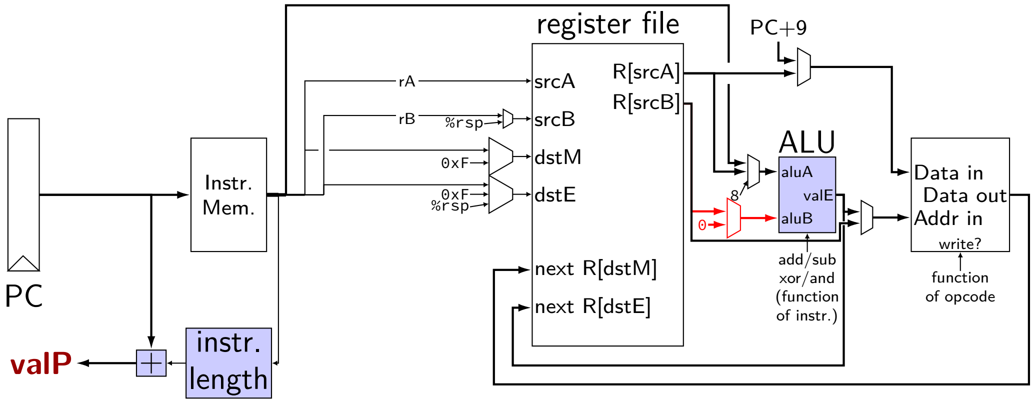
write disable — use dummy register number `0xF`



# write back (2a)



# write back (2b)

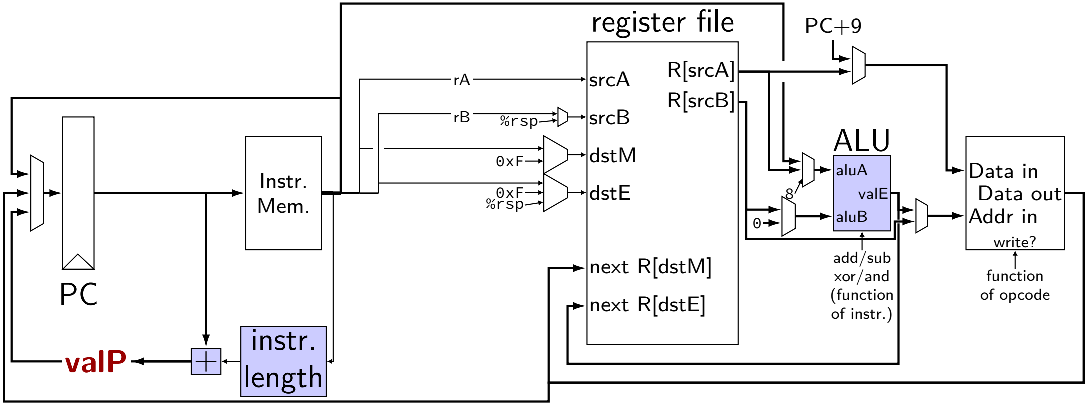


## SEQ: Update PC

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

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

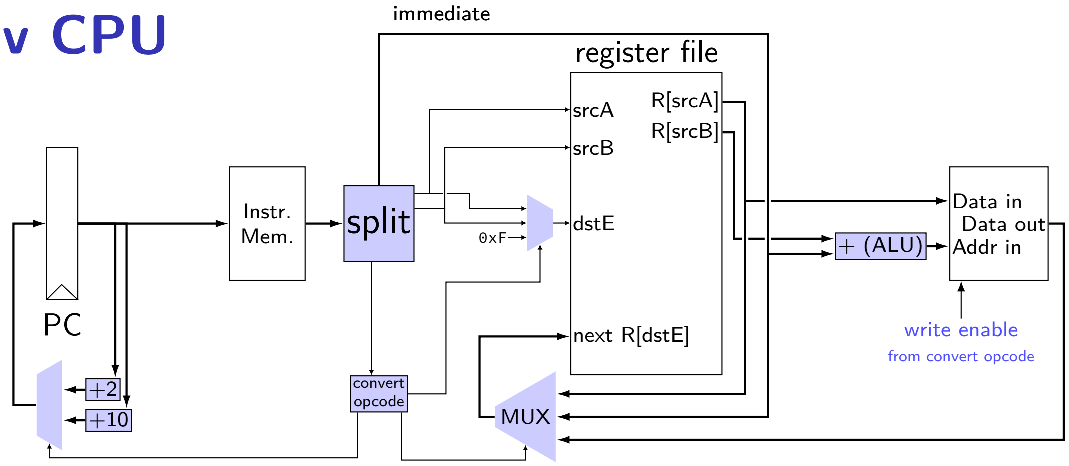
# PC update



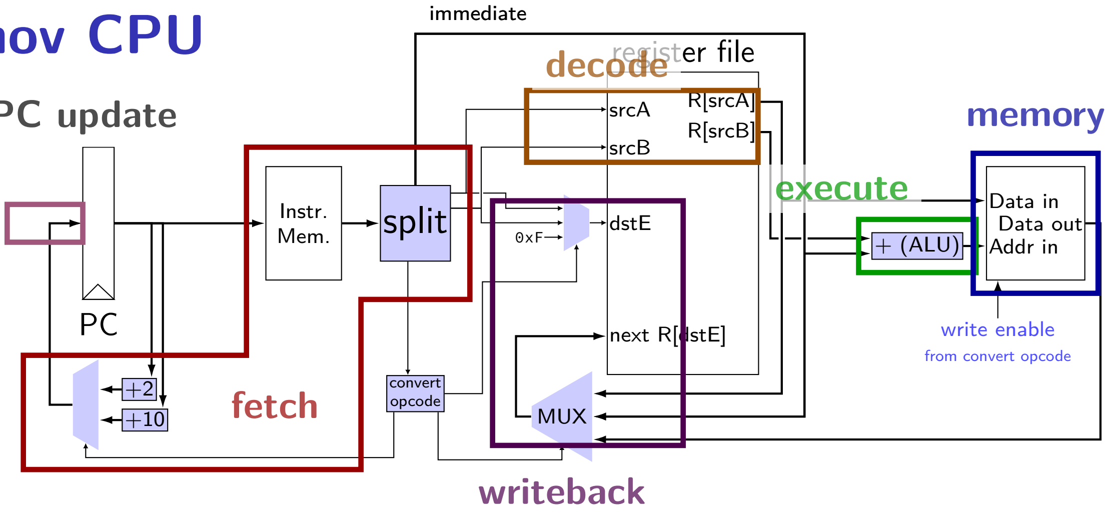
register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
0xF
0xF
%rsp
ALU
aluA
valE
aluB
8
0
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
write?
function
of opcode
Instr.
Mem.
PC
next R[dstM]
next R[dstE]
valP
+
instr.
length

## backup slides

# mov CPU


immediate
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
PC
Instr.
Mem.
split
0xF
convert opcode
MUX
+2
+10
+ (ALU)
Data in
Data out
Addr in
write enable
from convert opcode

# mov CPU


immediate
register file
decode
PC update
memory
srcA
srcB
R[srcA]
R[srcB]
Instr.
Mem.
split
execute
Data in
Data out
Addr in
+ (ALU)
0xF
dstE
PC
write enable
from convert opcode
next R[dstE]
+2
+10
convert
opcode
fetch
MUX
writeback