changelog

Changes since first lecture:

23 September 2021: in stages walkthrough, more consistently show valC

23 September 2021: in stages walkthrough, do not include mux on aluB input until after memory stage

last time

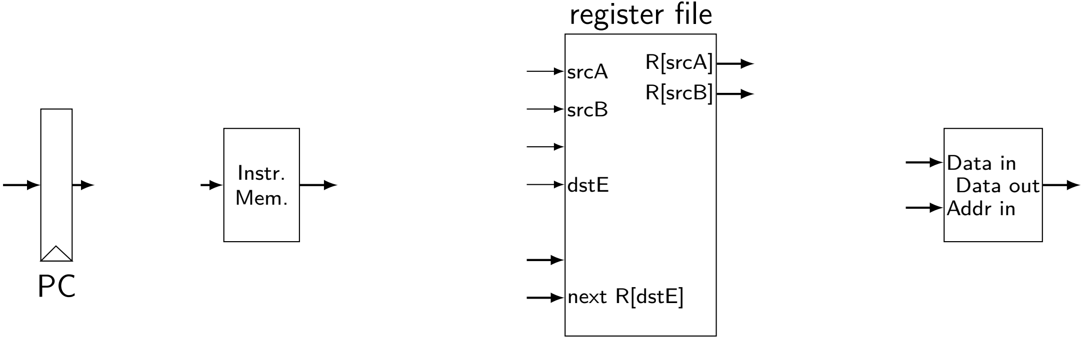
nop+jmp
- MUX to control the next PC value (input to PC register)

our register file
- read ports: set register index (srcA/B) to get value in same cycle (outputA/B)
- write ports: set register index (dstE/M) and value for next cycle (inputE/M)
- REG_NONE (15) for "don't write"

our data memory
- read/write enable signal
- reads in same cycle (from address)
- writes at end of cycle (like registers)

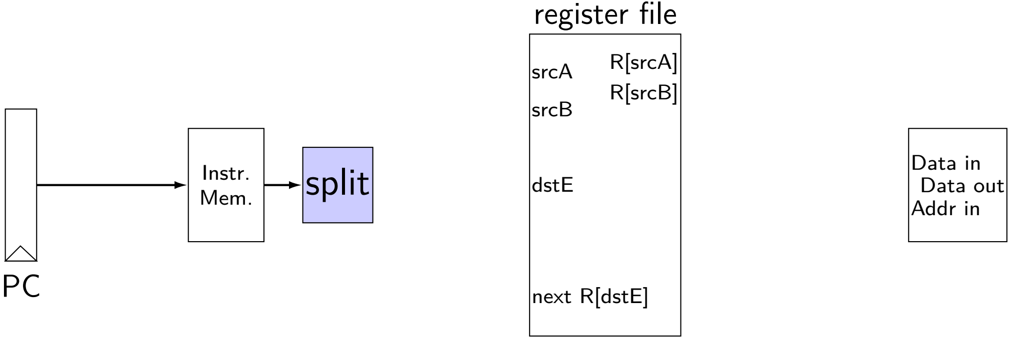
add CPU, add+nop CPU, mov-to-register CPU
- add MUXes where instructions want different things

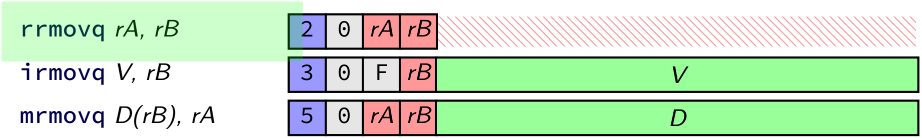
mov-to-register CPU



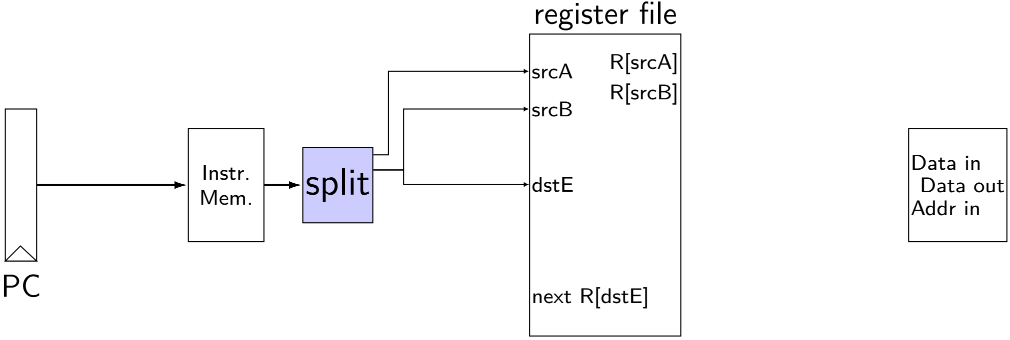
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



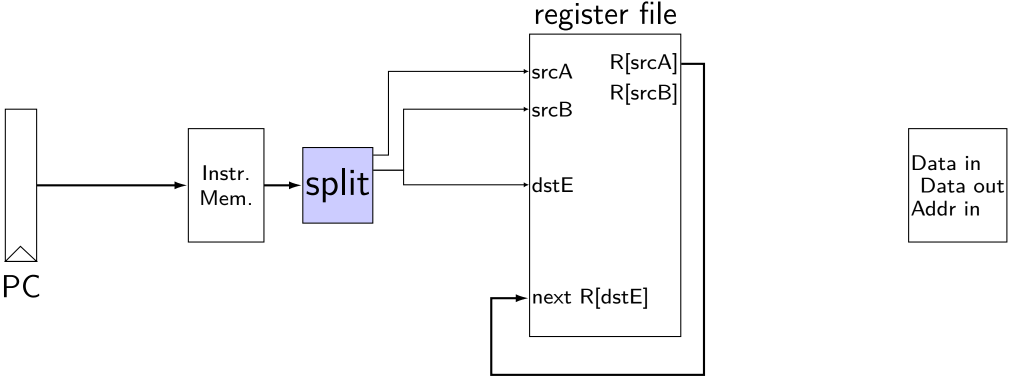


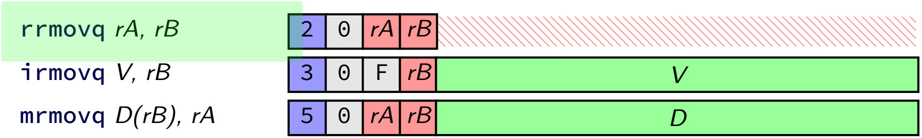
mov-to-register CPU



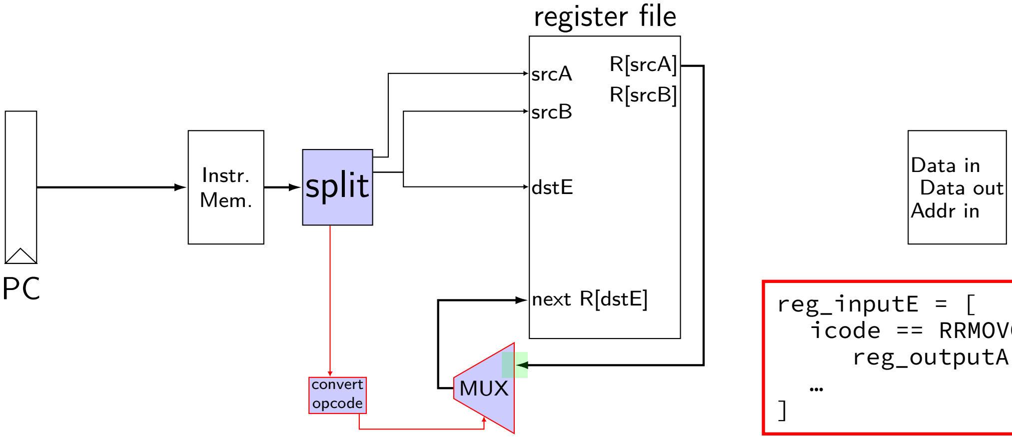
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU





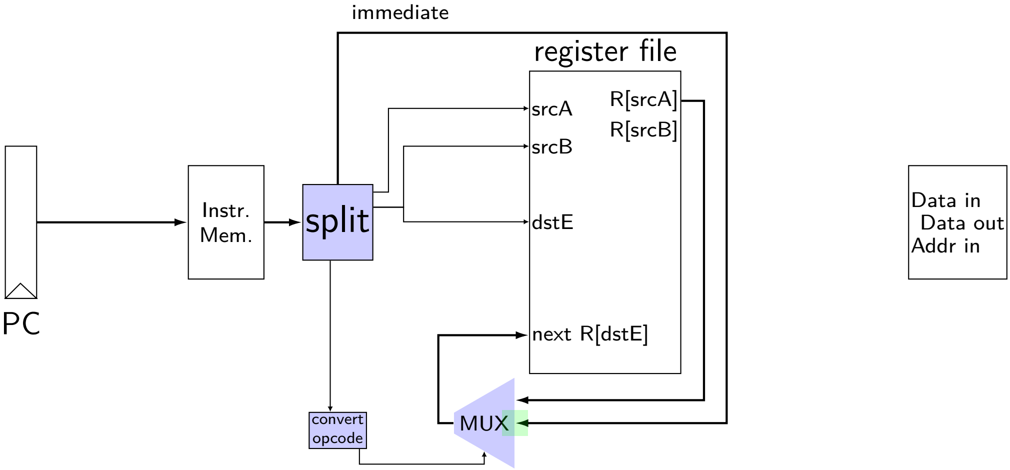
mov-to-register CPU




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

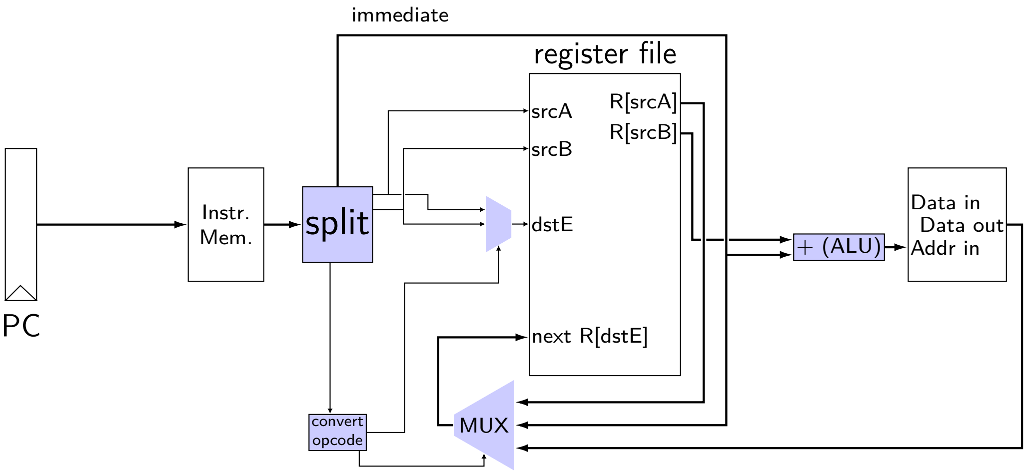
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



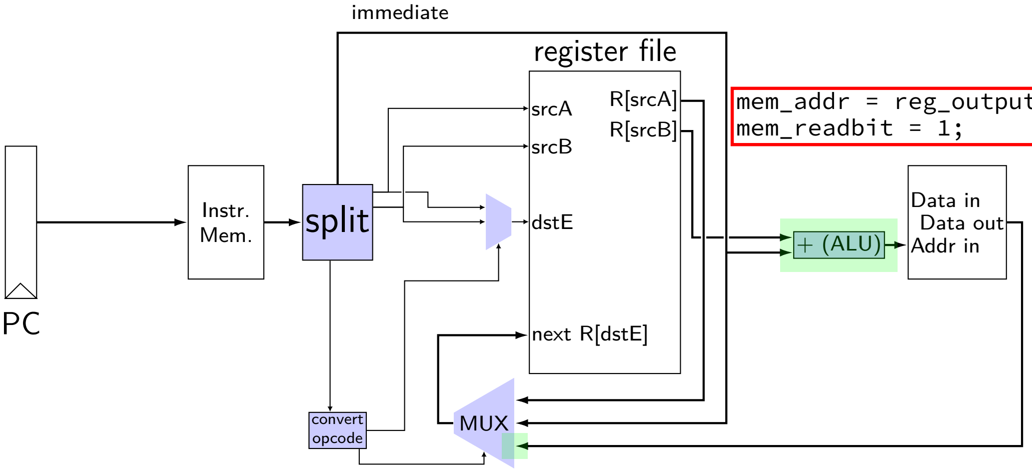
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



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

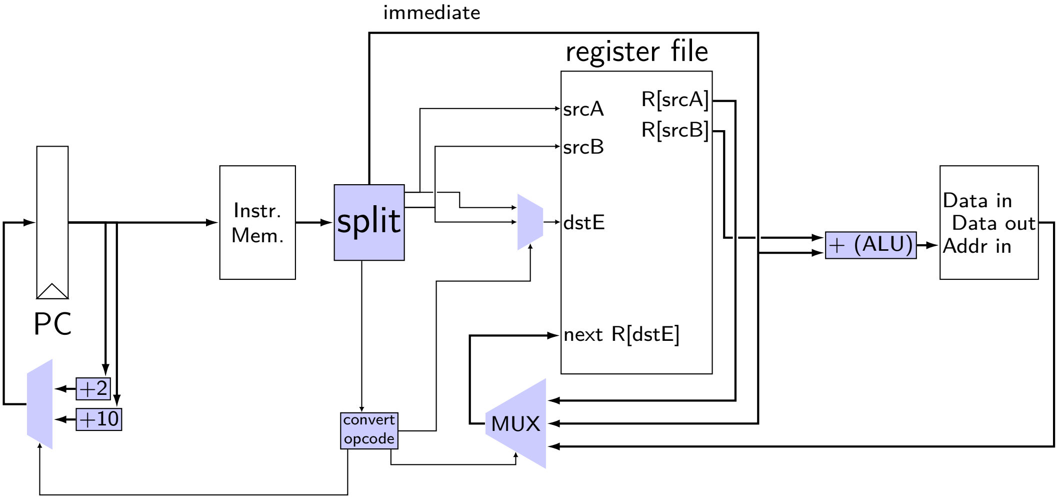
mov-to-register CPU




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

`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

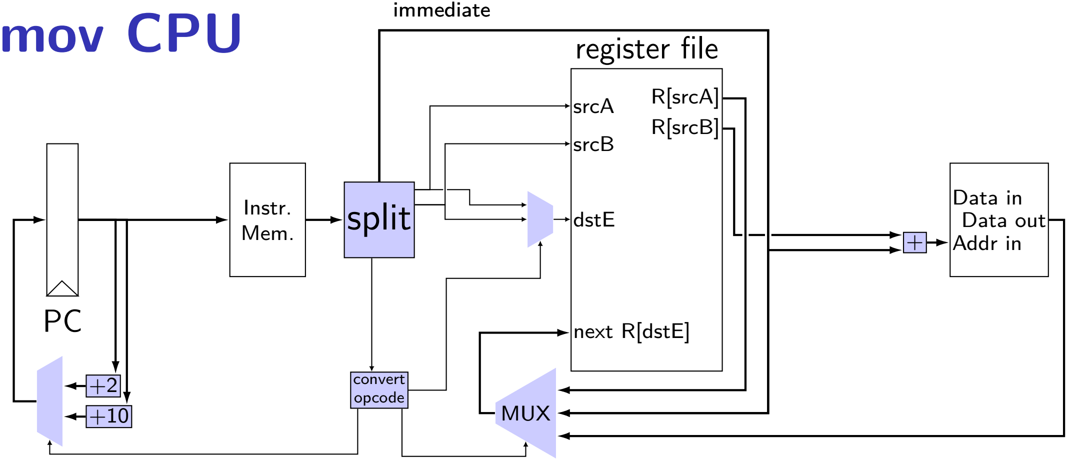


`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

simple ISA: mov (all cases)

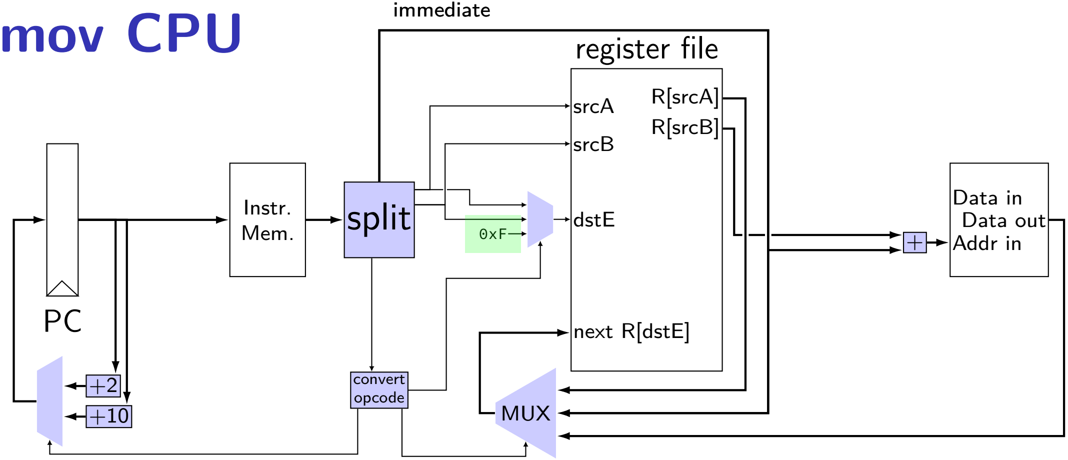
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

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

mov CPU



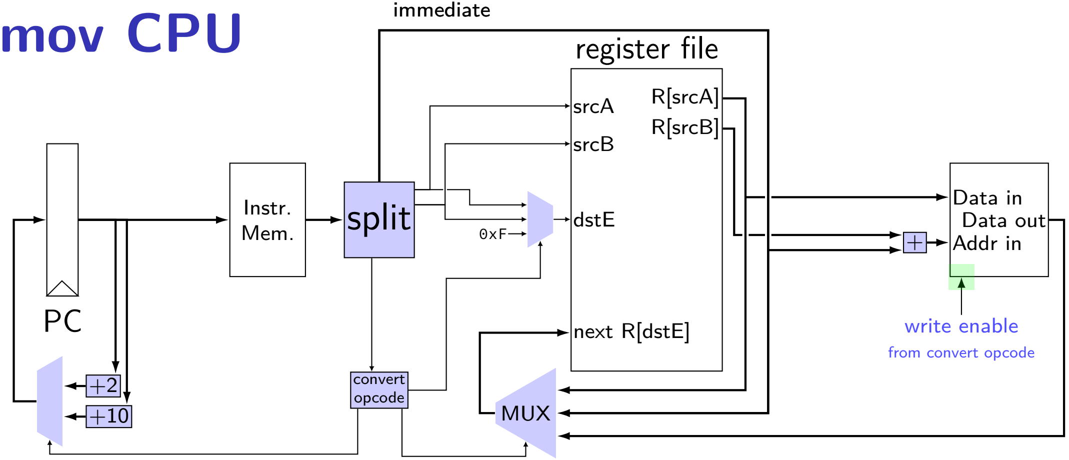
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA*, *D*(*rB*)	4	0	*rA*	*rB*	*D*

mov CPU



`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*

data path versus control path

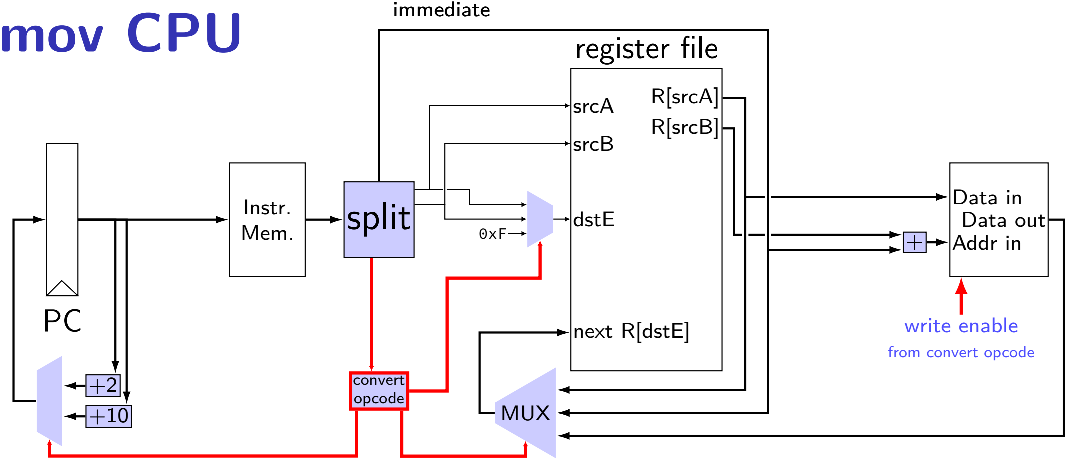
data path — signals carrying “actual data”

control path — signals that control MUXes, etc.

fuzzy line: e.g. are condition codes part of control path?

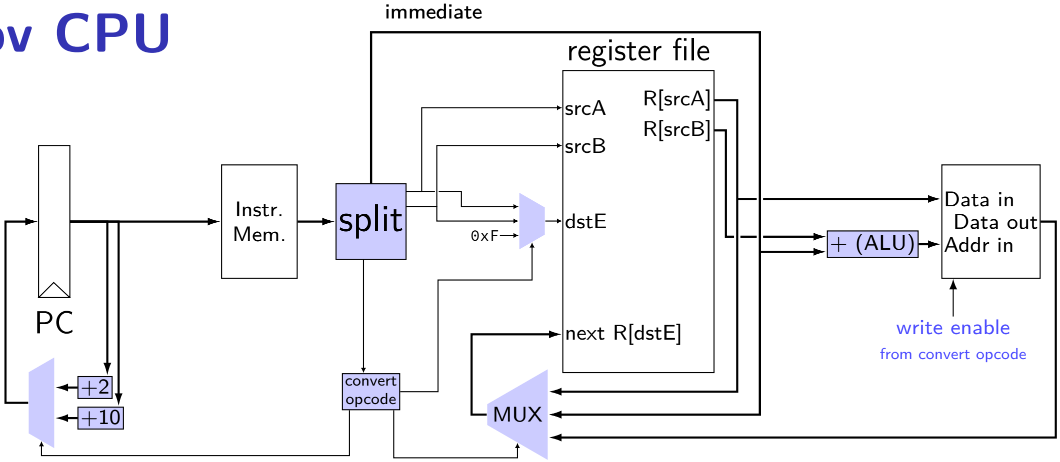
we will often omit parts of the control path in drawings, etc.

mov CPU

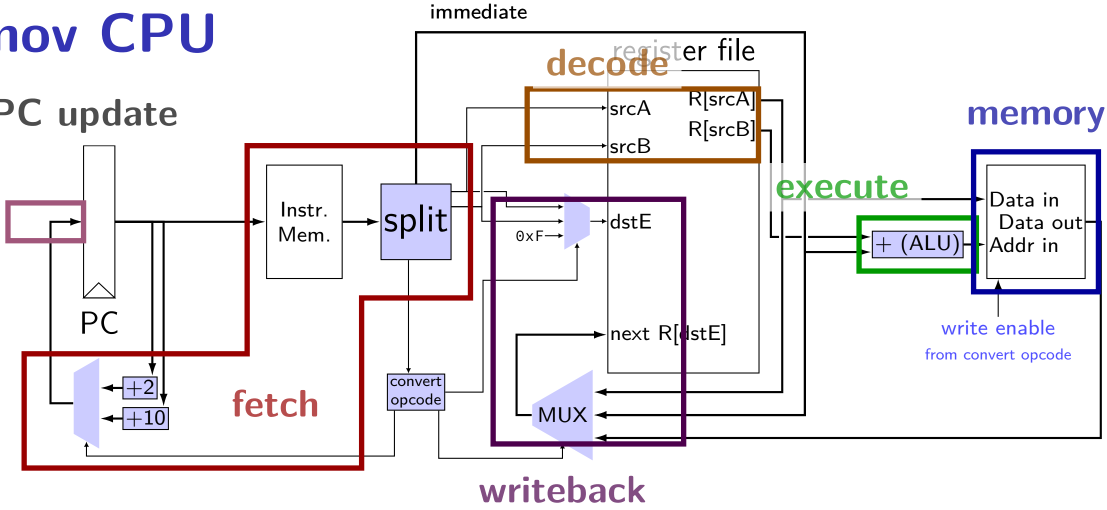


`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA*, *D*(*rB*)	4	0	*rA*	*rB*	*D*

mov CPU


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

mov CPU


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

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

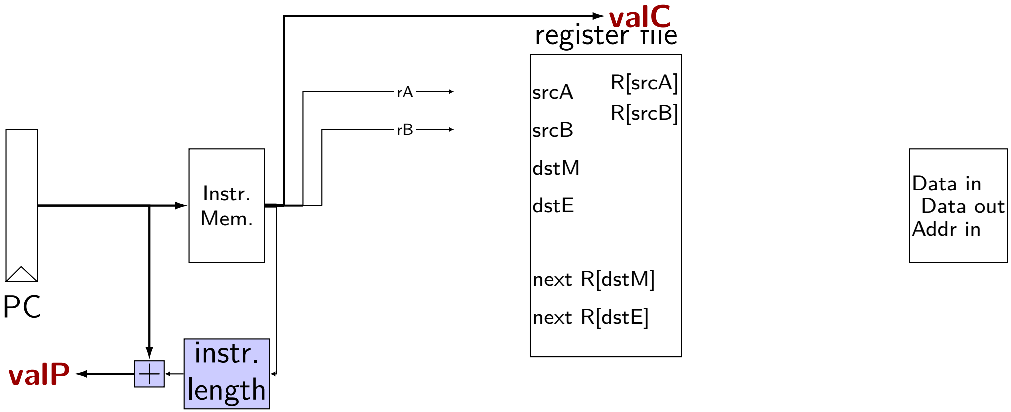
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

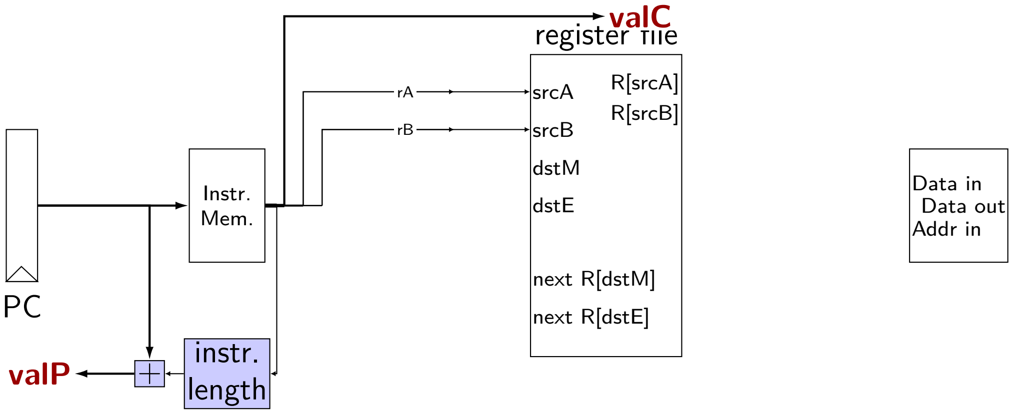


SEQ: instruction "decode"

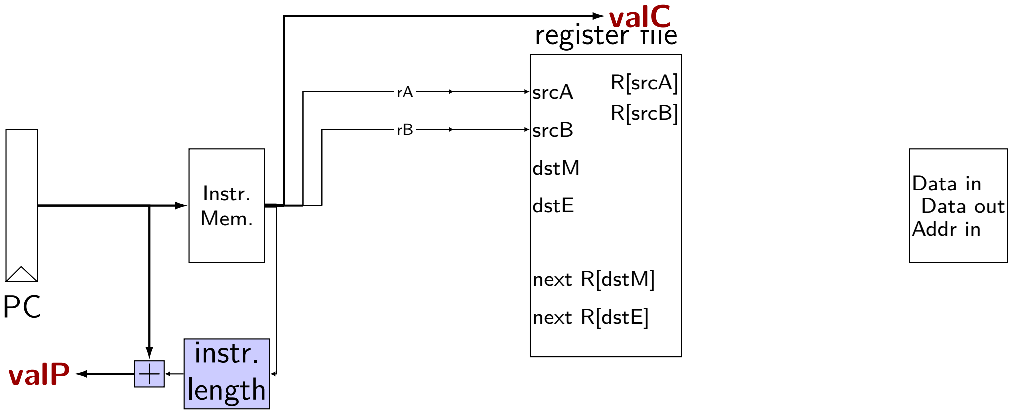
read registers

- valA, valB — register values

instruction decode (1)



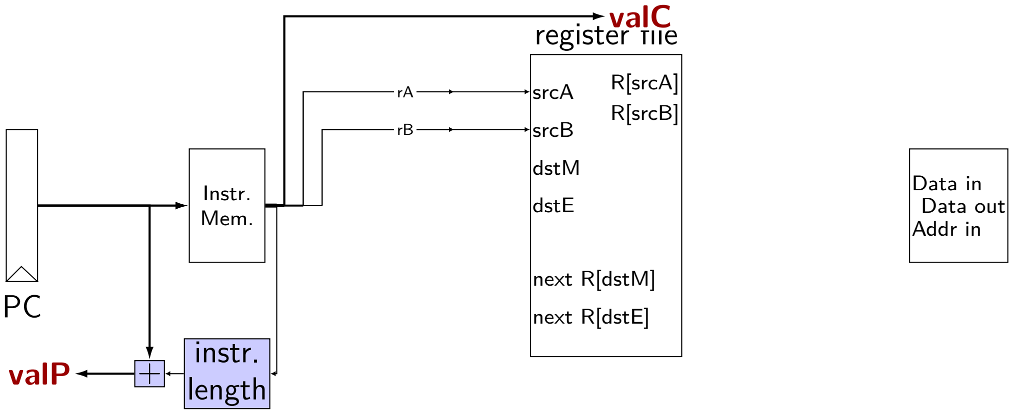
instruction decode (1)



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:

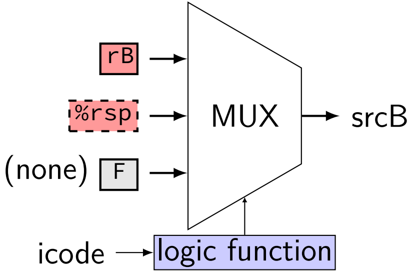
- push rA
- pop
- call
- ret

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

MUX controlled by icode

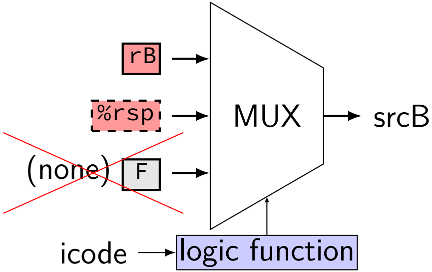
SEQ: possible registers to read

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`

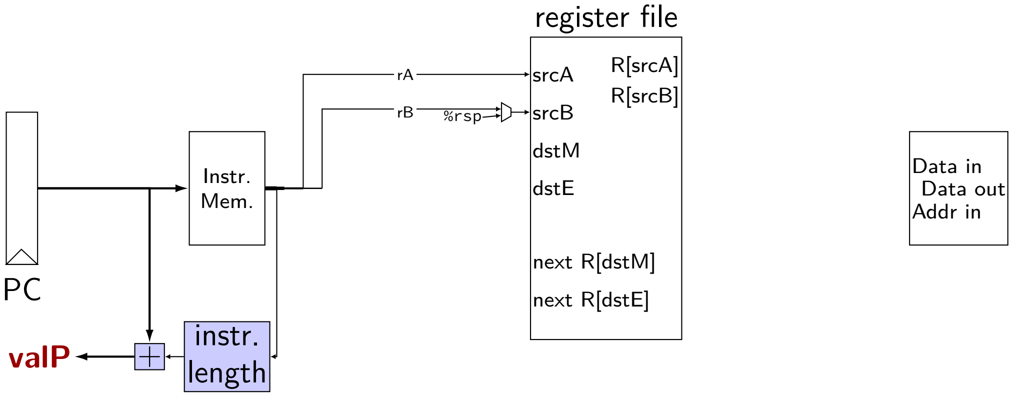


SEQ: possible registers to read

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`



instruction decode (2)



SEQ: execute

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

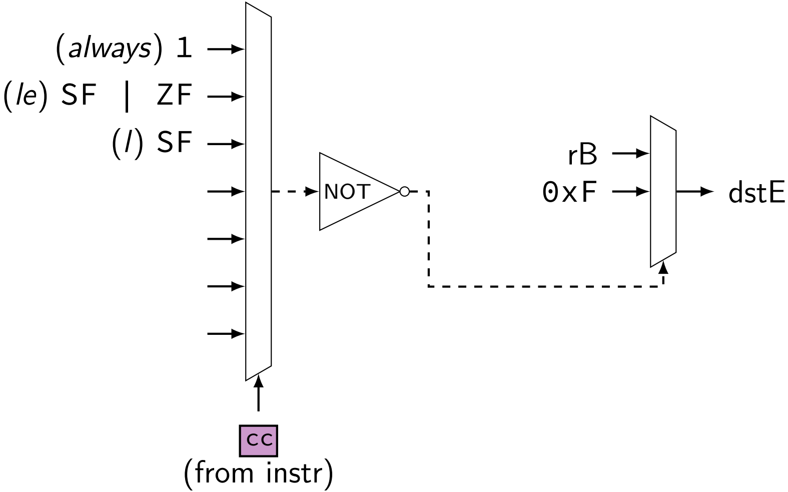
- valE — ALU output

read prior condition codes

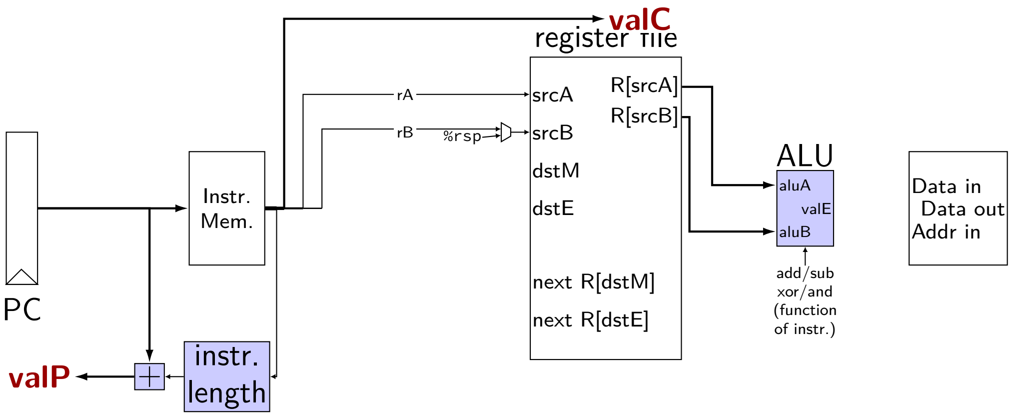
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

using condition codes: cmov

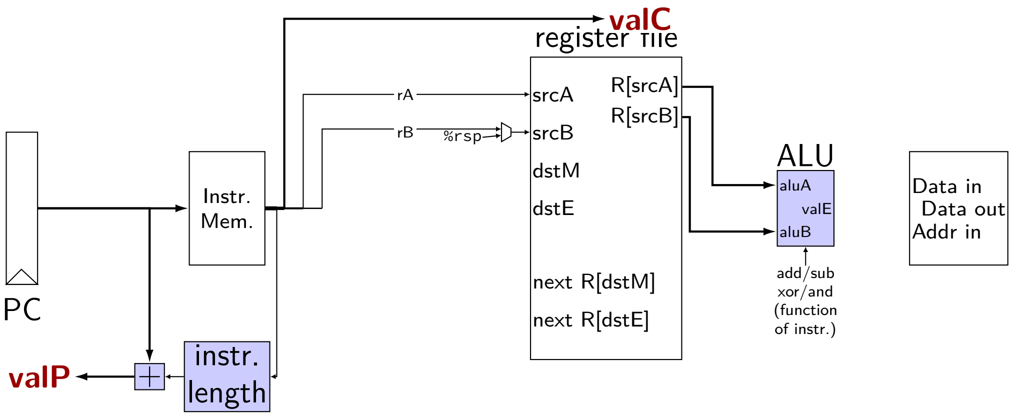


execute (1)



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
PC
Instr.
Mem.
instr.
length
valP

execute (1)



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

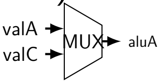
nop, addq, `mrmovq`, popq, `call`,

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

`mrmovq`
`rmmovq`



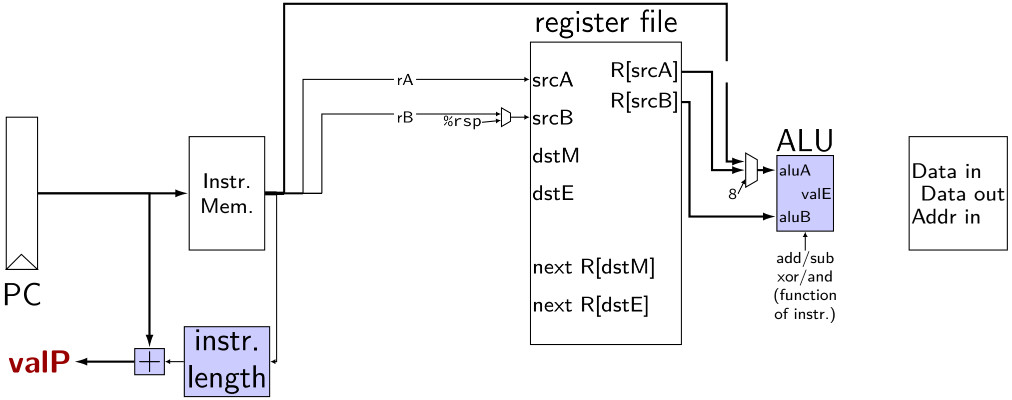
no, constants: (rsp +/- 8)

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

extra signals: aluA, aluB

computed ALU input values

execute (2)



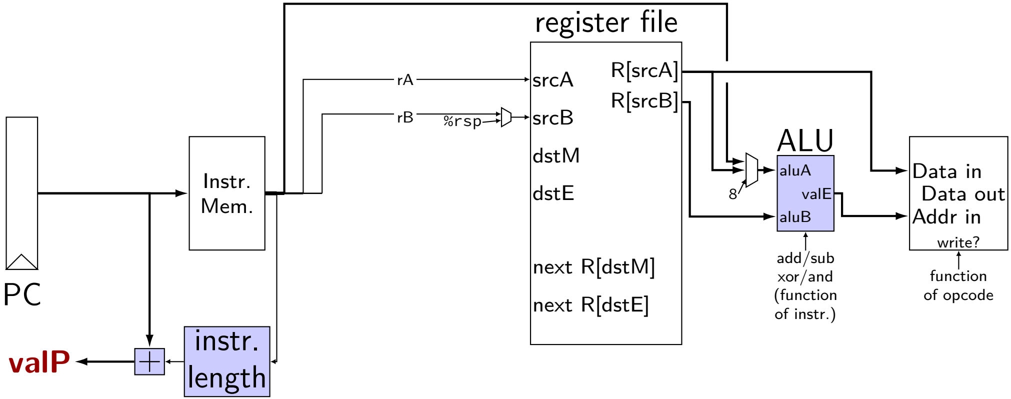
register file
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
rA
rB
%rsp
PC
Instr.
Mem.
valP
instr.
length
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in

SEQ: Memory

read or write data memory

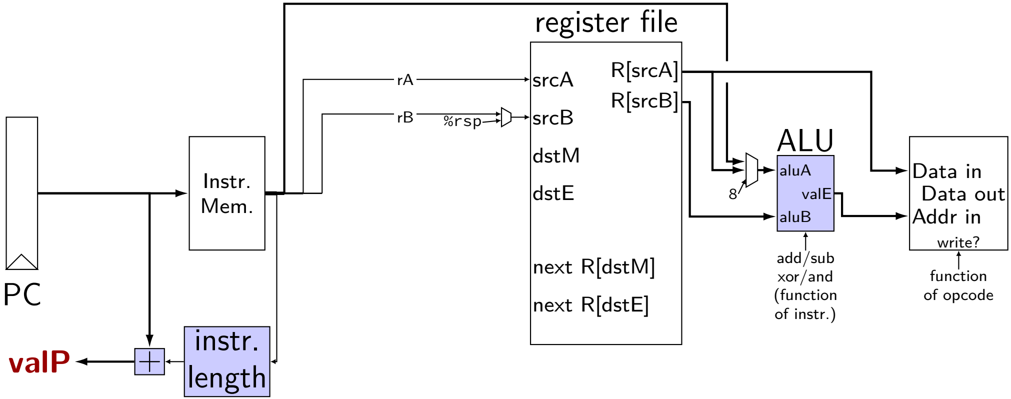
- valM — value read from memory (if any)

memory (1)



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

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?

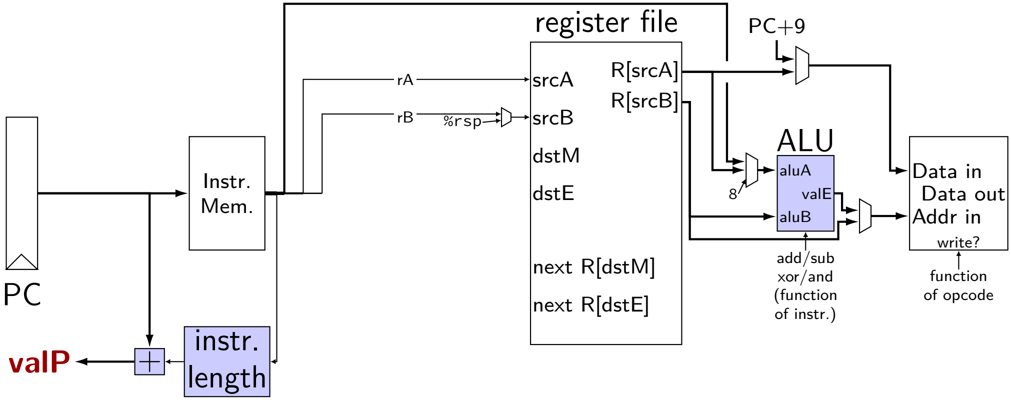
Addr — address

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

Data — value to write

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

memory (2)

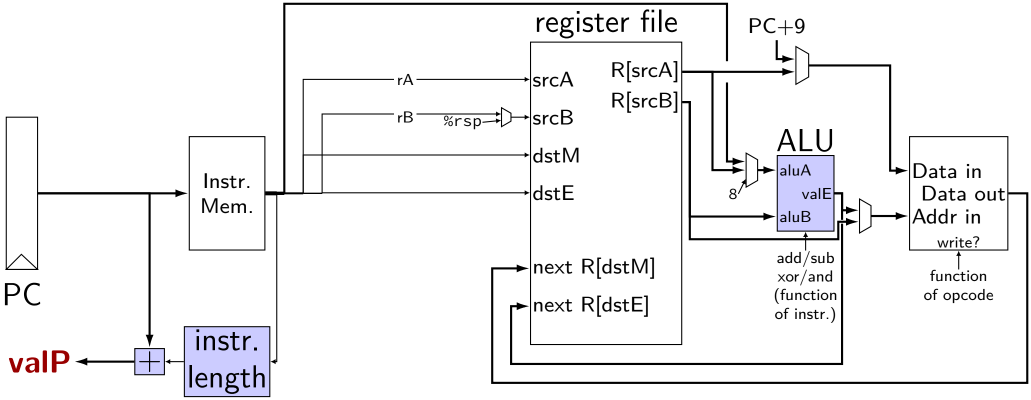


register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
next R[dstM]
next R[dstE]
R[srcA]
R[srcB]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
write?
function
of opcode
PC
Instr.
Mem.
instr.
length
valP

SEQ: write back

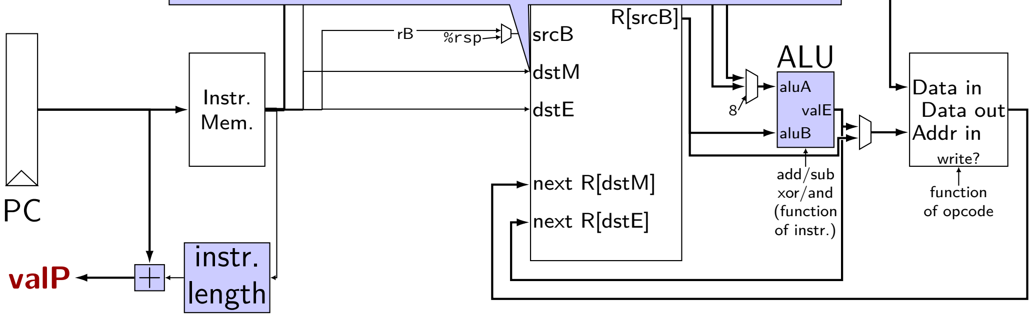
write registers

write back (1)

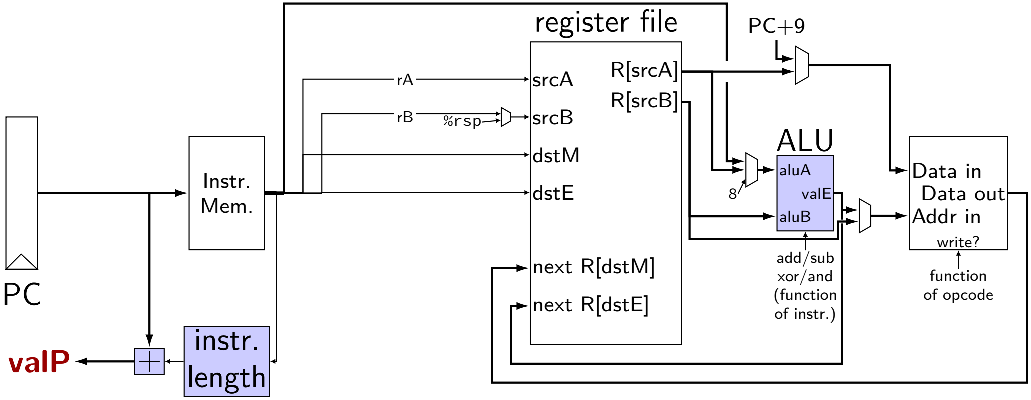

register file
PC+9
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
ALU
aluA
valE
aluB
8
Data in
Data out
Addr in
write?
function
of opcode
add/sub
xor/and
(function
of instr.)
next R[dstM]
next R[dstE]
Instr.
Mem.
PC
valP
instr.
length

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...)



write back (1)



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

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

SEQ: control signals for WB

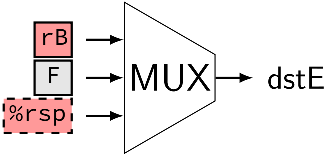
two write inputs — two needed by popq

- valM (memory output), valE (ALU output)

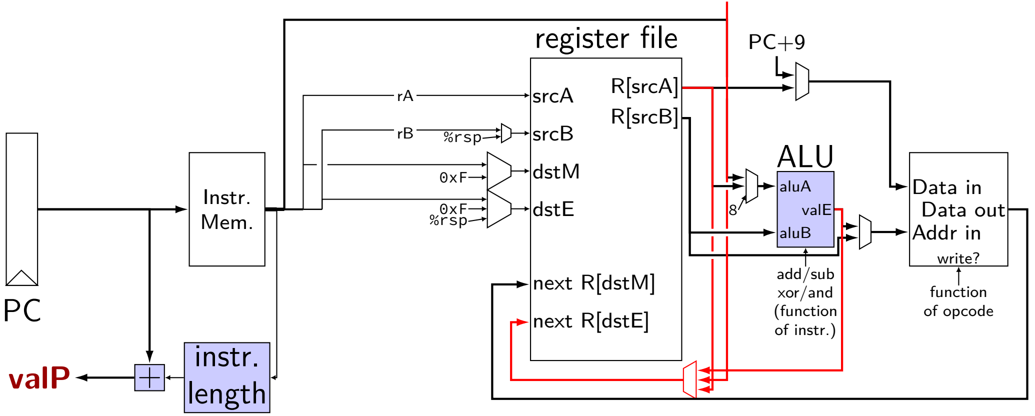
two register numbers

- dstM, dstE

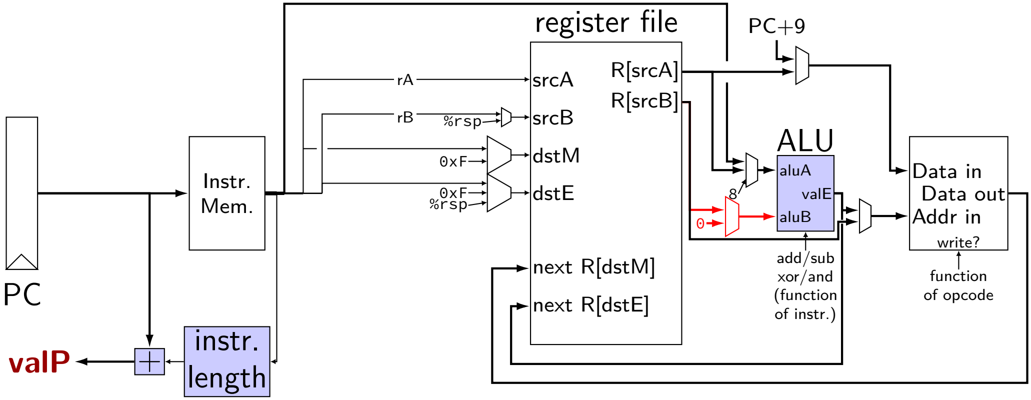
write disable — use dummy register number 0xF



write back (2a)



write back (2b)

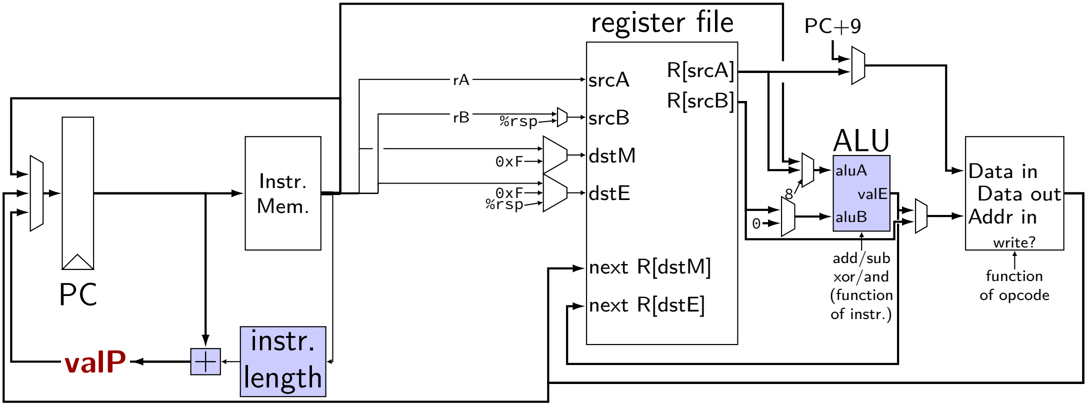


SEQ: Update PC

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

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)

backup slides