

SEQ part 2

1

Changelog

Changes made in this version not seen in first lecture:

- 19 September 2017: slide 18: send R[srcB] to ALU instead of R[srcA]
- 19 September 2017: slide 27: set register file write register number, not write enable

1

last time

arithmetic right shift — copy the sign bit

register, register file, memories

read continuously (possibly based on address/register number)

write on **rising edge of clock signal**

muxes to make decisions

e.g. PC + 1 (nop) or address read from memory (jmp)?

for now: fit all work of instruction within clock cycle

5

a correction

last time I said arithmetic right shift is sra on x86

on some other architectures (e.g. MIPS), but not on x86

the correct mnemonic is sar

note: if we ever use shift instructions on a test, we will tell you what they are

(don't bother memorizing their mnemonics)

6

arithmetic right shift

logical right shift — add zeroes 1011000 → 0101100

arithmetic right shift — copy sign bit 1011000 → 1101100

$(x \gg 1) \approx x / 2$ even for signed

$(x \gg 2) \approx x / 4$ even for signed

$(x \gg 3) \approx x / 8$ even for signed

$((-56) \gg 3) == -7$ (arithmetic: 11001000 → 11111001)

$((\text{unsigned char})200) \gg 3 == 25$ (logical:
11001000 → 00011001)

7

anonymous feedback (1)

“How on earth do you expect us to answer a similar question on the quiz as the final example during class, to which you gave a half-explained answer? ...” (paraphrased:) and your explanation didn’t give a yes/no answer to yes/no questions

8

the question last week

nop+add — where do we need a MUX (or similar logic)?

register to read — no; read and ignore result on nop
but ‘cleaner’ to specify no register explicitly??

PC input — yes; need to handle instruction length

register #/value to write — yes (either one); don’t change register
on nop

register number for “none” (15) or new value = old value

instruction memory address — no; always equal to PC output

9

the question this week

jmp+addq+mrmovq

read reg. # — no; read and ignore for jmp

PC input — yes; variable instruction length, jumps

write reg. # — need MUX on this or register value input

write enable for memory — no; hard-wire to false

address input to instruction memory — no; equal to PC output

ALU inputs — yes; mrmovq needs to add constant from instruction

10

anonymous feedback (2)

(paraphrased) the question on timing on the post-quiz was unfair since it was in next week's textbook material

intention: how addq CPU + register file + registers worked

(but, yes, easier if you had read ahead)

11

anonymous feedback (3)

(paraphrased) competition scoreboard for bomb HW was intimidating/demoralizing

competitions make *some* of our students learn more

I like it better than extra credit because it's more optional

But better ways of making it feel optional?

really should be no obligation to do more than assignment says

12

anonymous feedback (4)

(paraphrased) you should drop any question most students get wrong on the quizzes

I think this policy would create some perverse incentives

13

anonymous feedback (4)

(paraphrased) please check your slides more carefully

14

lists HW

due tomorrow

hopefully you've started — don't fight segfaults at the last minute

15

bit puzzles lab/HW

bit fiddling problems, e.g. isLessThanOrEqualTo

all in C

restrictions on operators you can use

lab — practice problems, complete what you can

N.B.: you can work together

uneven difficulty; don't expect anyone to do all of them

homework — **different** problems, complete all

16

simple ISA 4: mov-to-register

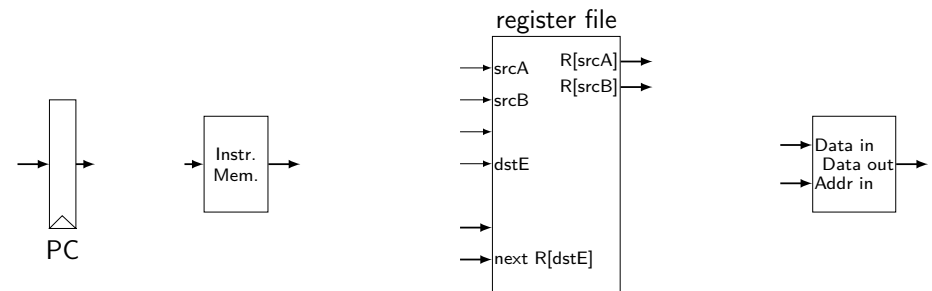
```
irmovq $constant, %rYY
```

```
rrmovq %rXX, %rYY
```

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

17

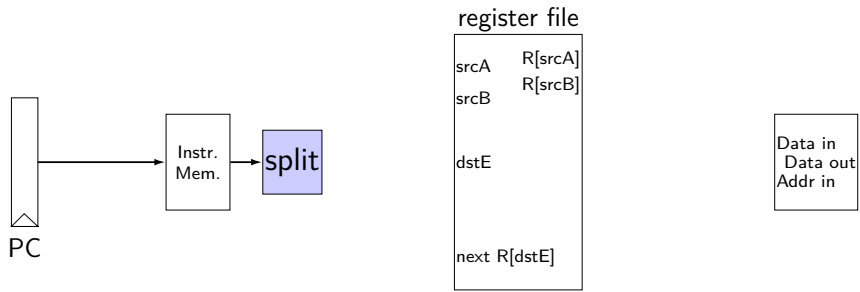
mov-to-register CPU



<code>rrmovq rA, rB</code>	2	0	rA	rB	
<code>irmovq V, rB</code>	3	0	F	rB	V
<code>mrmovq D(rB), rA</code>	5	0	rA	rB	D

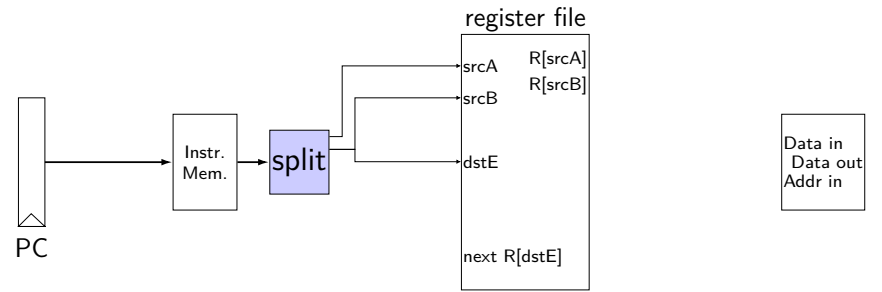
18

mov-to-register CPU



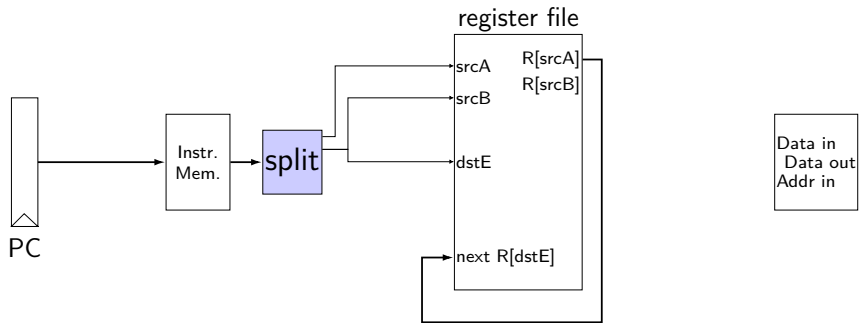
<i>rrmovq rA, rB</i>	2	0	rA	rB	
<i>irmovq V, rB</i>	3	0	F	rB	V
<i>mrmovq D(rB), rA</i>	5	0	rA	rB	D

mov-to-register CPU



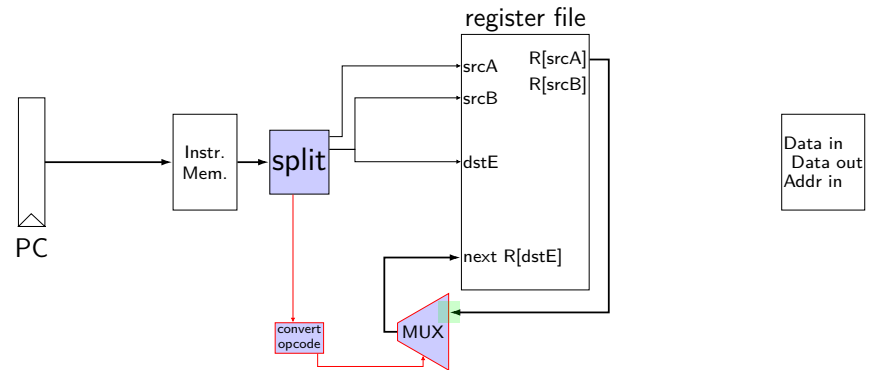
<i>rrmovq rA, rB</i>	2	0	rA	rB	
<i>irmovq V, rB</i>	3	0	F	rB	V
<i>mrmovq D(rB), rA</i>	5	0	rA	rB	D

mov-to-register CPU



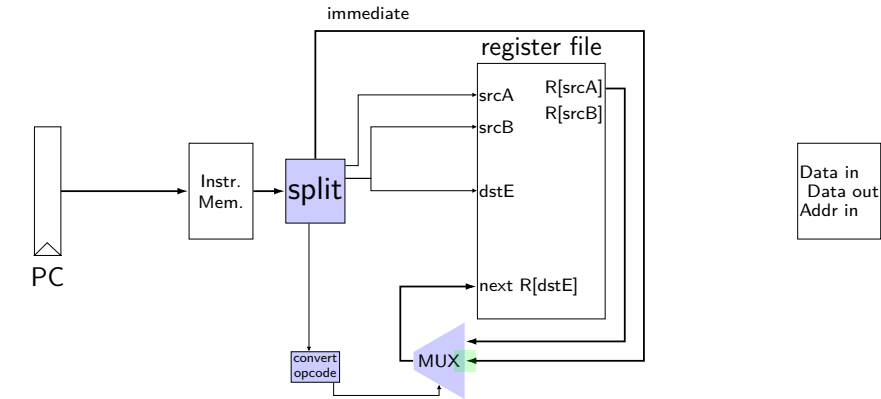
<i>rrmovq rA, rB</i>	2	0	rA	rB	
<i>irmovq V, rB</i>	3	0	F	rB	V
<i>mrmovq D(rB), rA</i>	5	0	rA	rB	D

mov-to-register CPU



<i>rrmovq rA, rB</i>	2	0	rA	rB	
<i>irmovq V, rB</i>	3	0	F	rB	V
<i>mrmovq D(rB), rA</i>	5	0	rA	rB	D

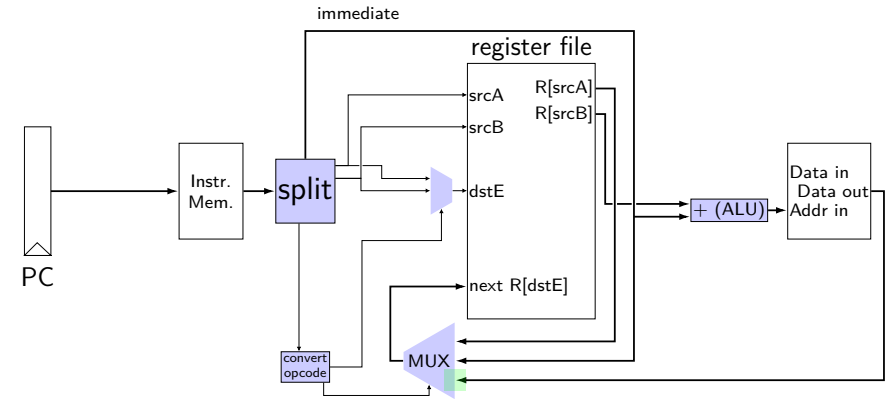
mov-to-register CPU



<code>rrmovq rA, rB</code>	2	0	rA	rB	
<code>irmovq V, rB</code>	3	0	F	rB	V
<code>mrmovq D(rB), rA</code>	5	0	rA	rB	D

18

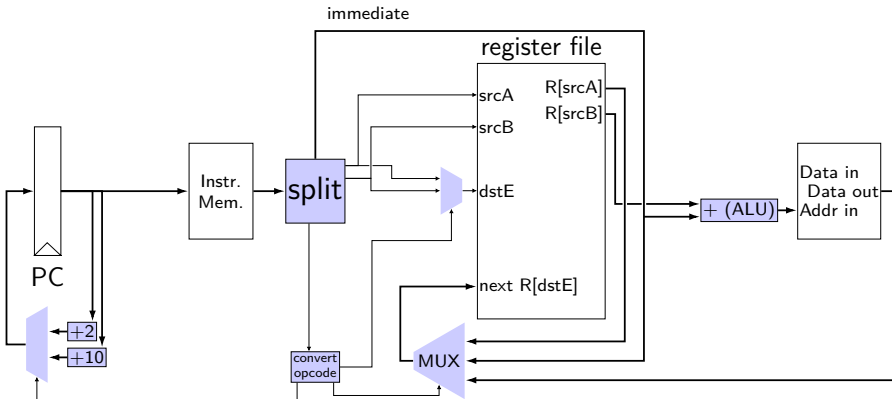
mov-to-register CPU



<code>rrmovq rA, rB</code>	2	0	rA	rB	
<code>irmovq V, rB</code>	3	0	F	rB	V
<code>mrmovq D(rB), rA</code>	5	0	rA	rB	D

18

mov-to-register CPU



<code>rrmovq rA, rB</code>	2	0	rA	rB	
<code>irmovq V, rB</code>	3	0	F	rB	V
<code>mrmovq D(rB), rA</code>	5	0	rA	rB	D

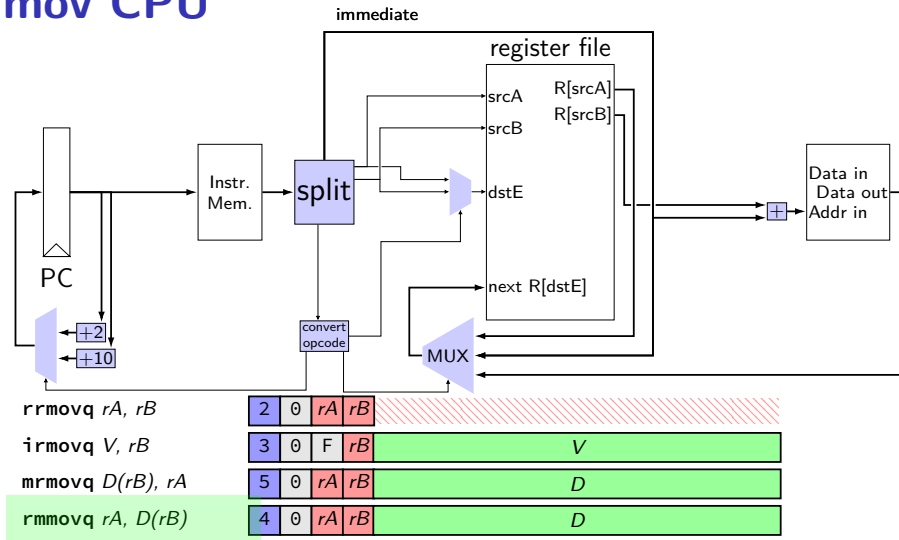
18

simple ISA 4B: mov

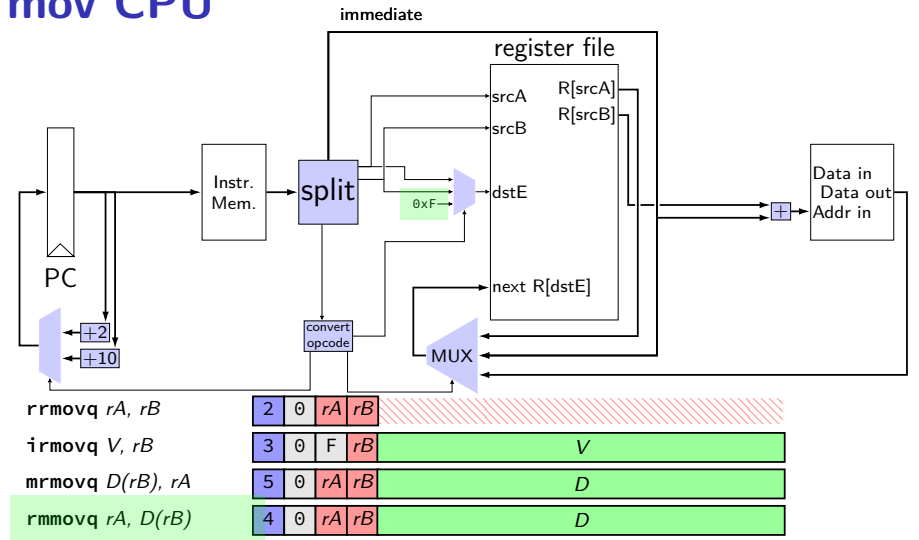
`irmovq $constant, %rYY`
`rrmovq %rXX, %rYY`
`mrmovq 10(%rXX), %rYY`
`rmmovq %rXX, 10(%rYY)`

19

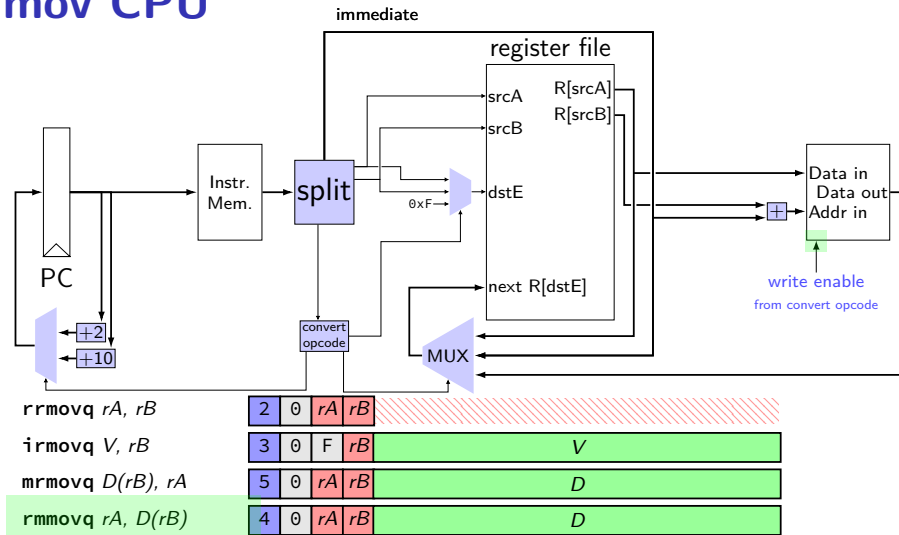
mov CPU



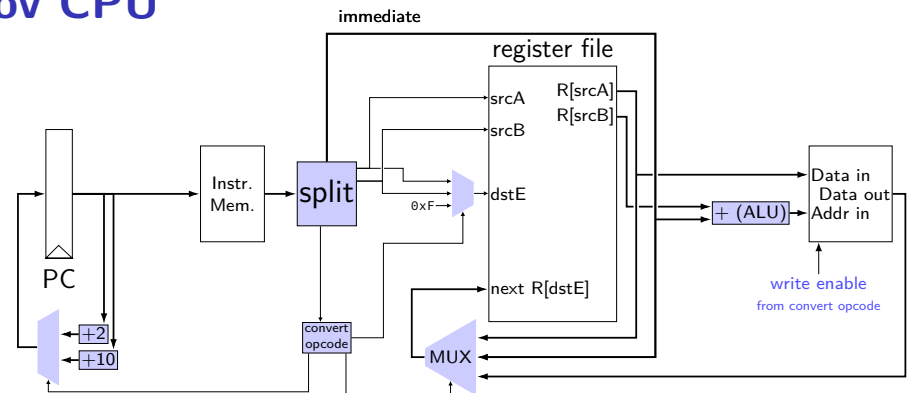
mov CPU



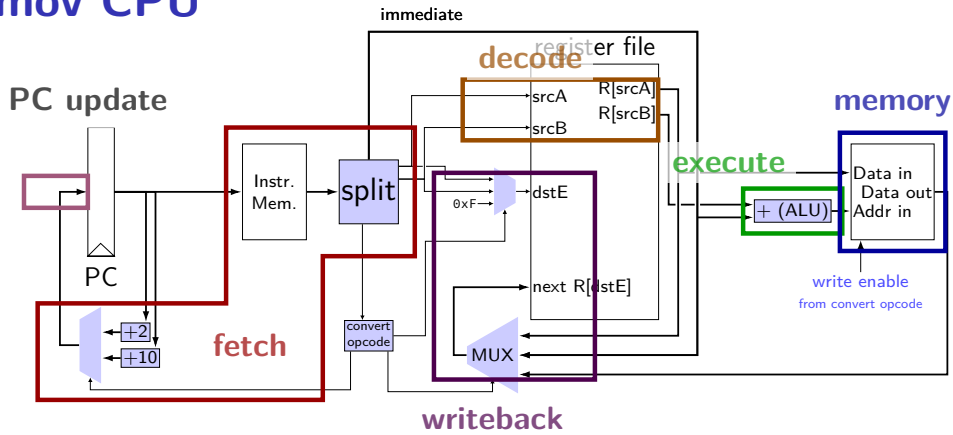
mov CPU



mov CPU



mov CPU



21

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

22

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

23

stages example: nop

stage	nop
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 1$
decode	
memory	
write back	
PC update	PC $\leftarrow valP$

24

stages example: nop

stage	nop
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 1$
decode	
memory	
write back	
PC update	PC \leftarrow valP

part of output wires
from instruction memory

24

stages example: nop

stage	nop
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 1$
decode	
memory	
write back	
PC update	PC \leftarrow valP

name of a wire
 \leftarrow means putting a value on a wire

24

stages example: nop

stage	nop
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 1$
decode	
memory	
write back	
PC update	PC \leftarrow valP

\leftarrow means putting value on
input wire to PC register

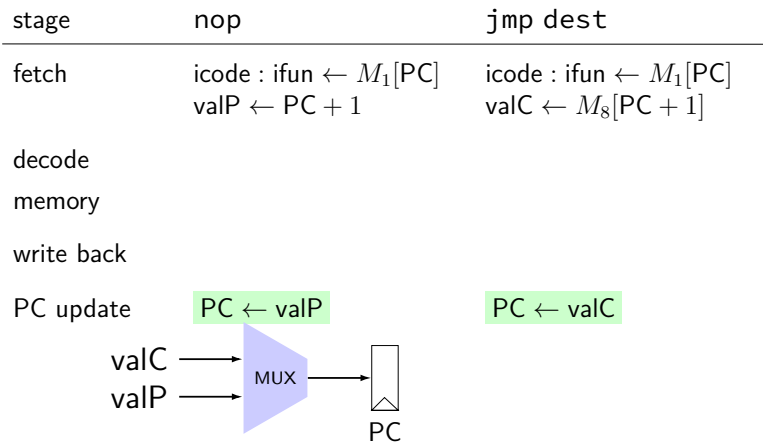
24

stages example: nop/jmp

stage	nop	jmp dest
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 1$	icode : ifun $\leftarrow M_1[PC]$ valC $\leftarrow M_8[PC + 1]$
decode		
memory		
write back		
PC update	PC \leftarrow valP	PC \leftarrow valC

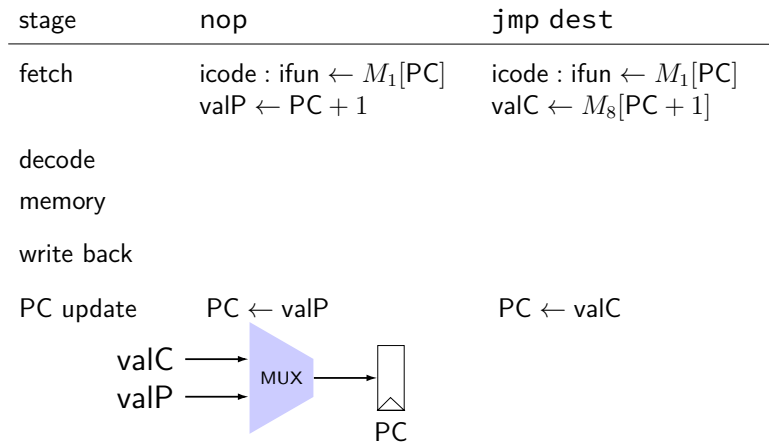
25

stages example: nop/jmp



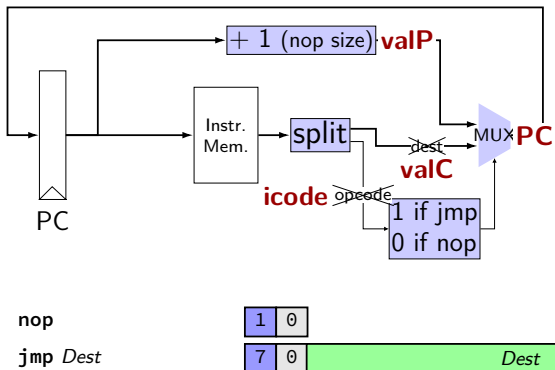
25

stages example: nop/jmp



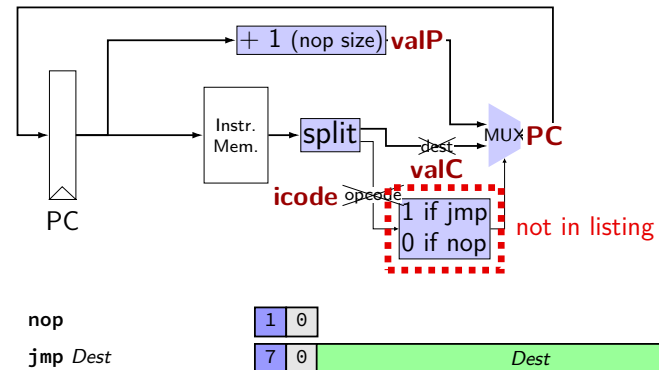
25

jmp+nop CPU



26

jmp+nop CPU



26

stages example: rmmovq/mrmovq

stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$
decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valB $\leftarrow R[rB]$
execute	valE $\leftarrow valB + valC$	valE $\leftarrow valB + valC$
memory	$M_8[valE] \leftarrow valA$	valM $\leftarrow M_8[valE]$
write back		$R[rA] \leftarrow valM$
PC update	PC $\leftarrow valP$	PC $\leftarrow valP$

27

stages example: rmmovq/mrmovq

stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$
decode	valA $\leftarrow R[rA]$	valB $\leftarrow R[rB]$
execute	valE $\leftarrow valB + valC$	valE $\leftarrow valB + valC$
memory	$M_8[valE] \leftarrow valA$	valM $\leftarrow M_8[valE]$
write back		$R[rA] \leftarrow valM$
PC update	PC $\leftarrow valP$	PC $\leftarrow valP$

assignment means:
 setting **register number** input register file *and*
 naming output wires of register file

27

stages example: rmmovq/mrmovq

stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$
decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valB $\leftarrow R[rB]$
execute	valE $\leftarrow valB + valC$	valE $\leftarrow valB + valC$
memory	$M_8[valE] \leftarrow valA$	valM $\leftarrow M_8[valE]$
write back		$R[rA] \leftarrow valM$
PC update	PC $\leftarrow valP$	PC $\leftarrow valP$

reading $R[rA]$ not needed
 but would be harmless

27

stages example: rmmovq/mrmovq

stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$
decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valB $\leftarrow R[rB]$
execute	valE $\leftarrow valB + valC$	valE $\leftarrow valB + valC$
memory	$M_8[valE] \leftarrow valA$	valM $\leftarrow M_8[valE]$
write back		$R[rA] \leftarrow valM$
PC update	PC $\leftarrow valP$	PC $\leftarrow valP$

assignment means:
 setting **address** wires to valE *and*
 setting **value input** wires to valA *and*
 setting memory **write enable** to 1

27

stages example: rmmovq/mrmovq

stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$
decode	valA $\leftarrow M_8[valE]$	valM $\leftarrow M_8[valE]$
execute		
memory		
write back		$R[rA] \leftarrow valM$
PC update	$PC \leftarrow valP$	$PC \leftarrow valP$

assignment means:
setting **address** wires to valE and
naming the output of the data memory

27

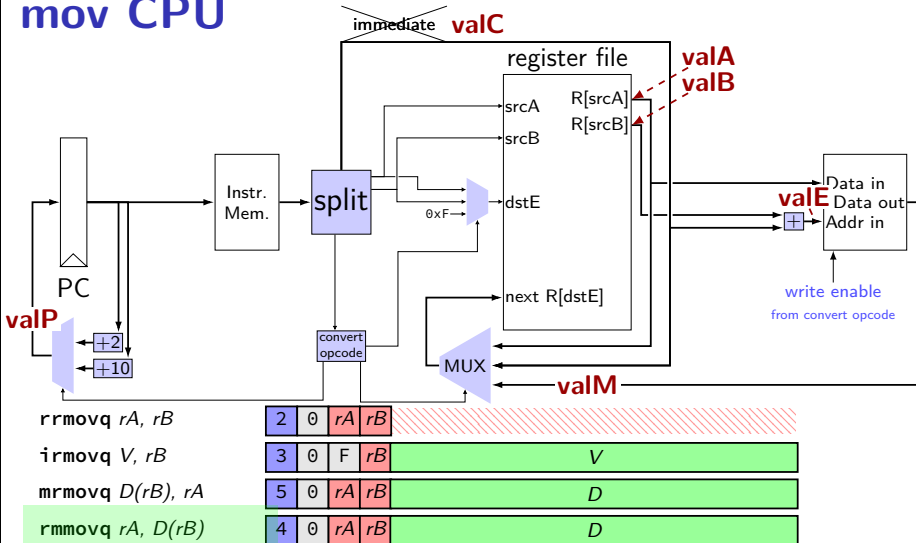
stages example: rmmovq/mrmovq

stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
fetch	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$	icode : ifun $\leftarrow M_1[PC]$ valP $\leftarrow PC + 10$ valC $\leftarrow M_8[PC + 2]$
decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	valB $\leftarrow R[rB]$
execute	valM $\leftarrow M_8[valE]$	
memory		
write back		$R[rA] \leftarrow valM$
PC update	$PC \leftarrow valP$	$PC \leftarrow valP$

assignment means:
setting register file input wires to valM
setting register file **write register number**

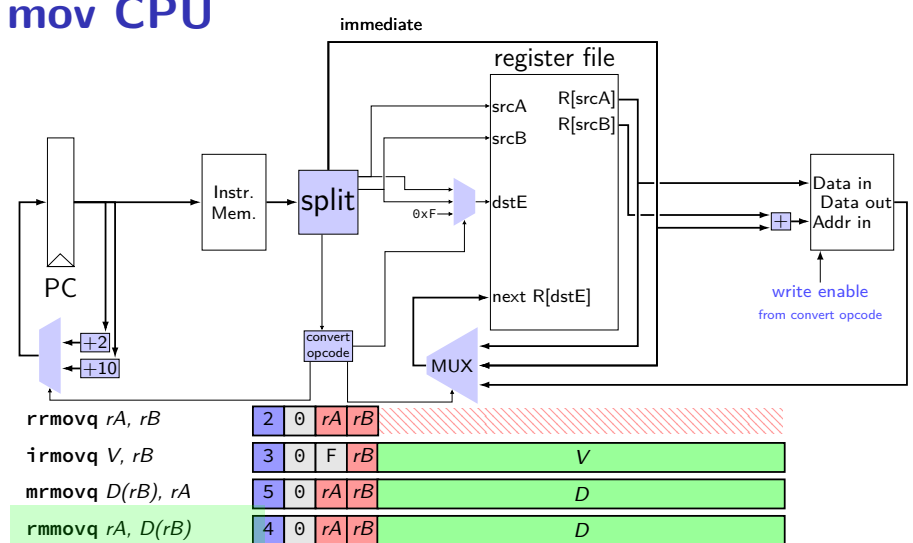
27

mov CPU



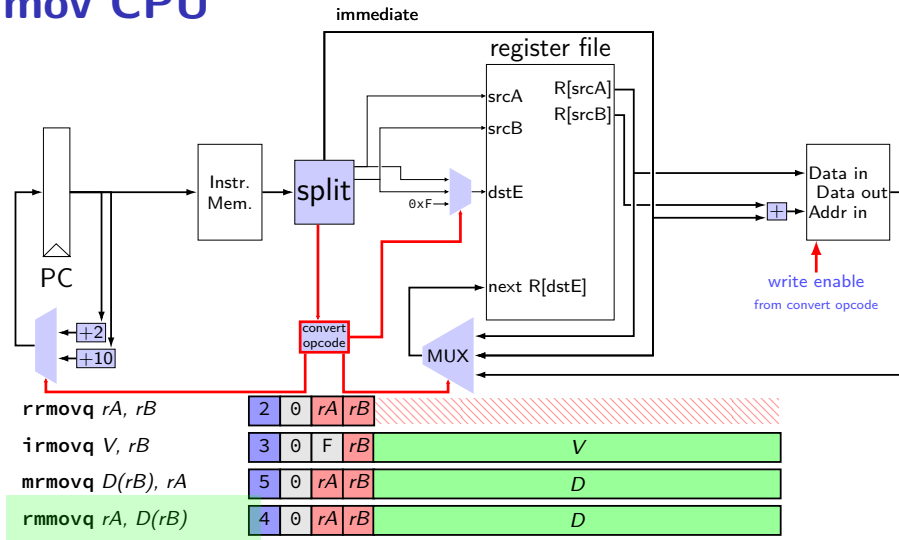
28

mov CPU



28

mov CPU



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.

SEQ: instruction fetch

read instruction memory at PC

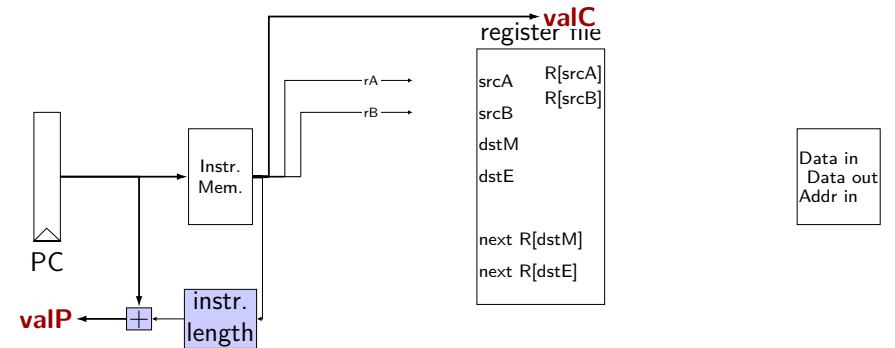
split into separate wires:

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

compute next instruction address:

- `valP` — $PC + (\text{instr length})$

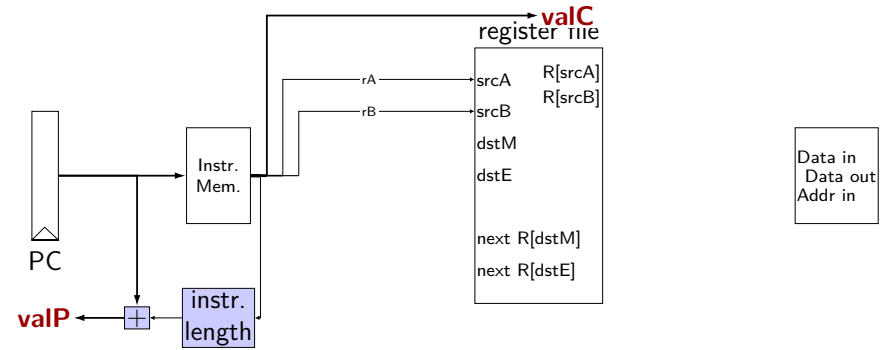
instruction fetch



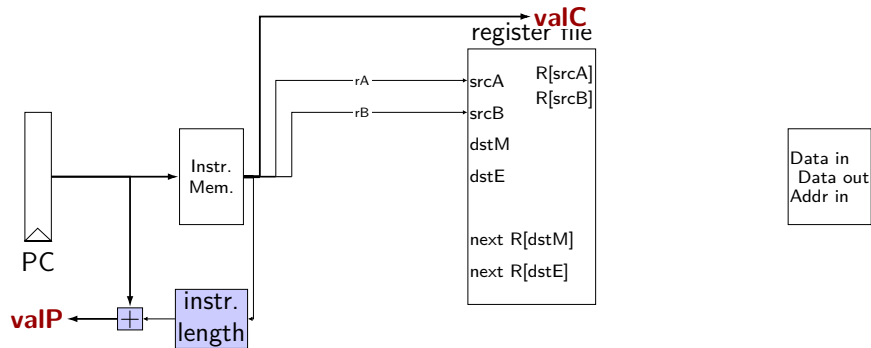
SEQ: instruction “decode”

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, popq, call,

SEQ: srcA, srcB

always read rA, rB?

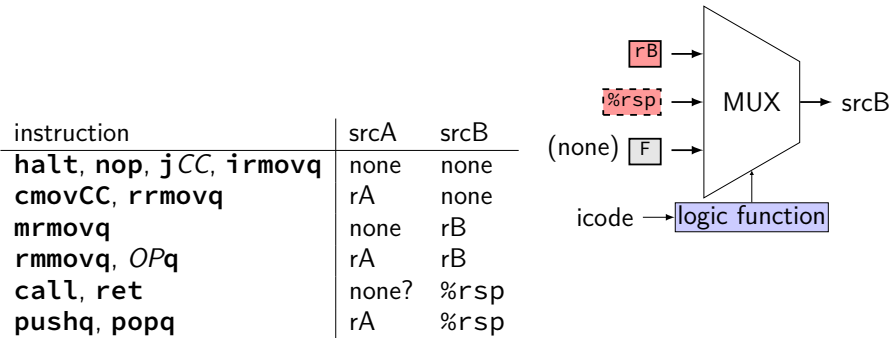
Problems:

- push rA
- pop
- call
- ret

extra signals: srcA, srcB — computed input register

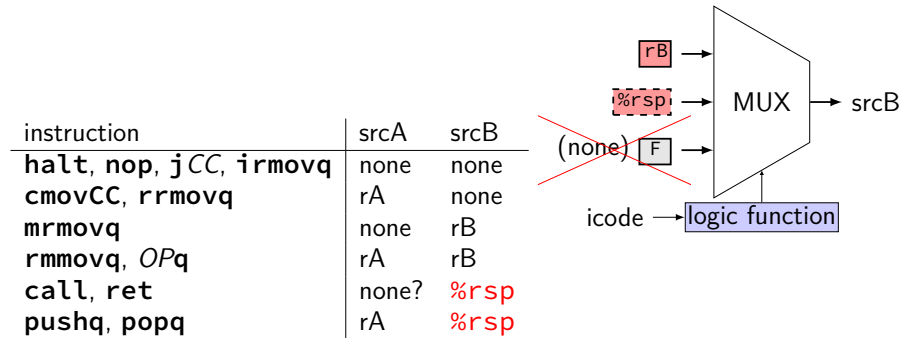
MUX controlled by icode

SEQ: possible registers to read



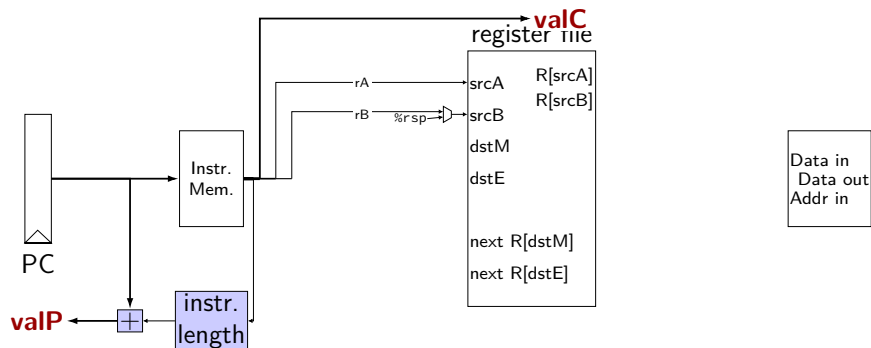
35

SEQ: possible registers to read



35

instruction decode (2)



36

SEQ: execute

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

valE — ALU output

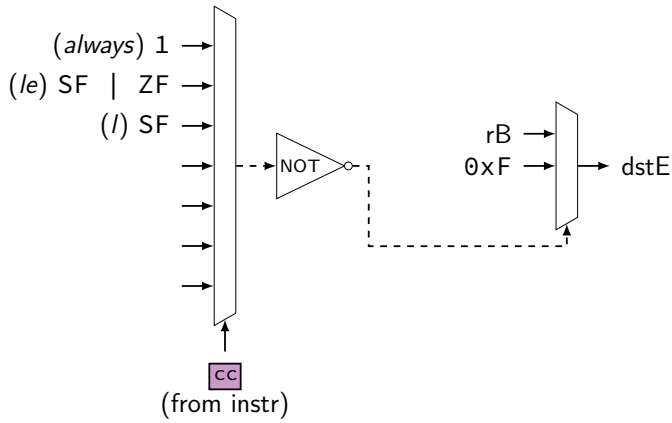
read prior condition codes

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

write new condition codes

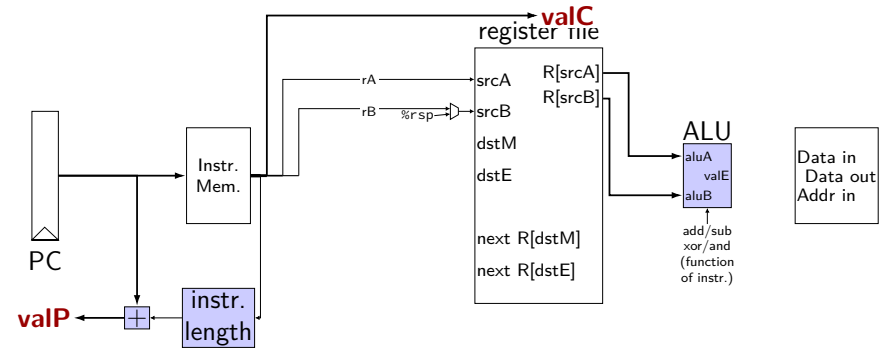
37

using condition codes: cmov*



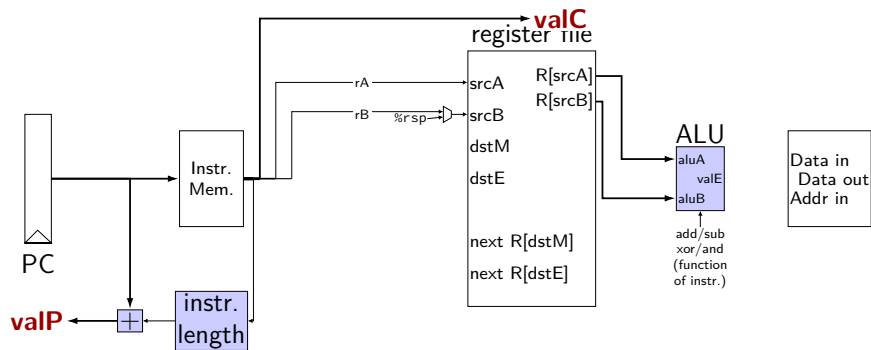
38

execute (1)



39

execute (1)



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

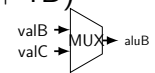
39

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

mrmovq
rmmovq



no, constants: (rsp +/- 8)

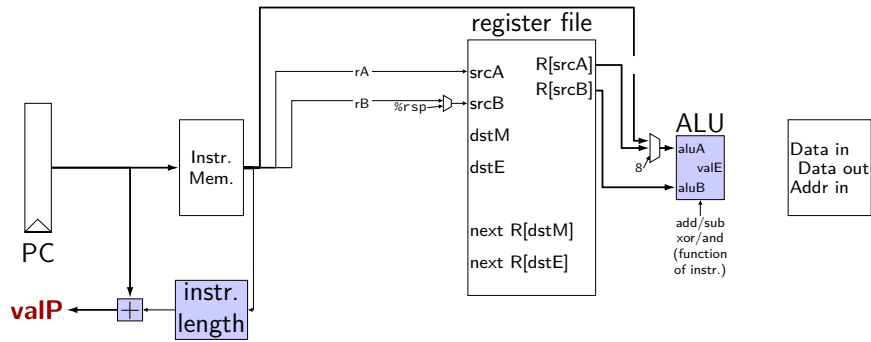
pushq
popq
call
ret

extra signals: **aluA**, **aluB**

computed ALU input values

40

execute (2)



41

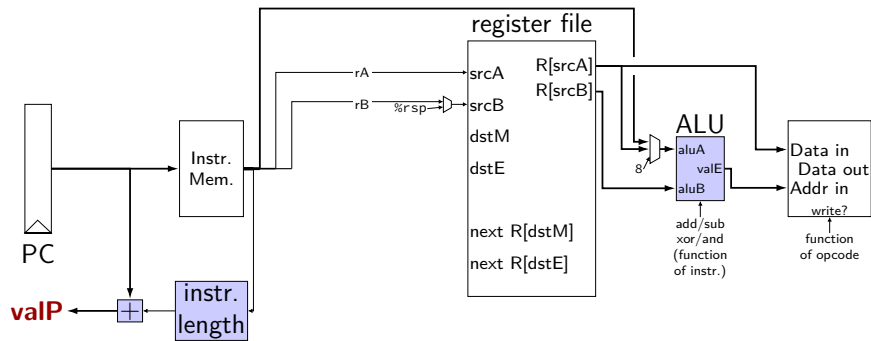
SEQ: Memory

read or write data memory

$valM$ — value read from memory (if any)

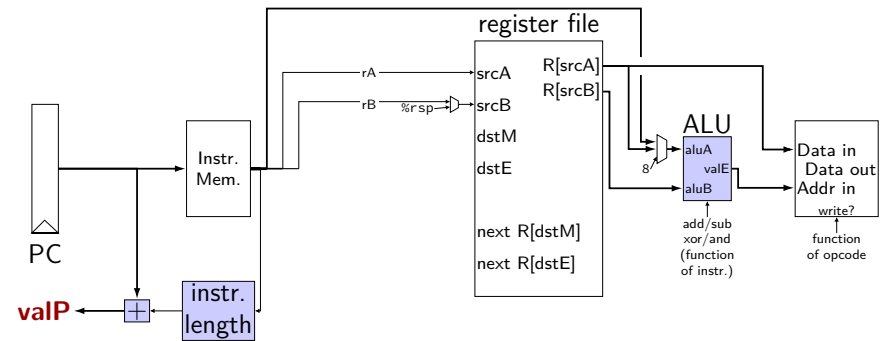
42

memory (1)



43

memory (1)



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

43

SEQ: control signals for memory

read/write — read enable? write enable?

Addr — address

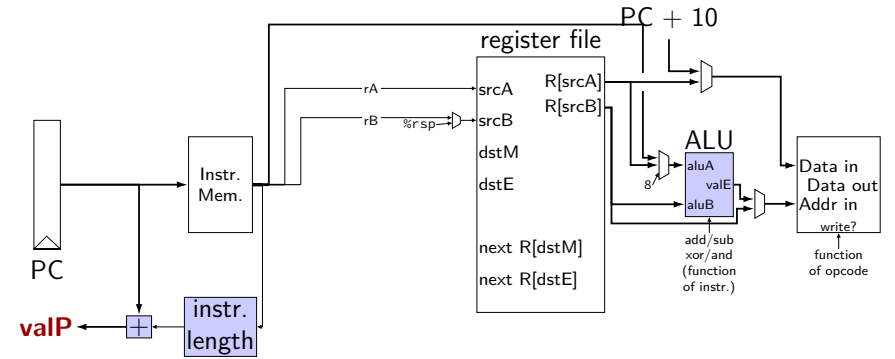
mostly ALU output
tricky cases: **popq, ret**

Data — value to write

mostly valB
tricky cases: **call, push**

44

memory (2)



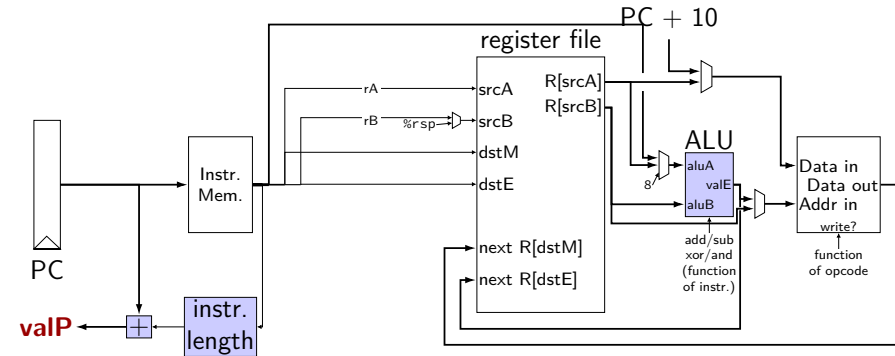
45

SEQ: write back

write registers

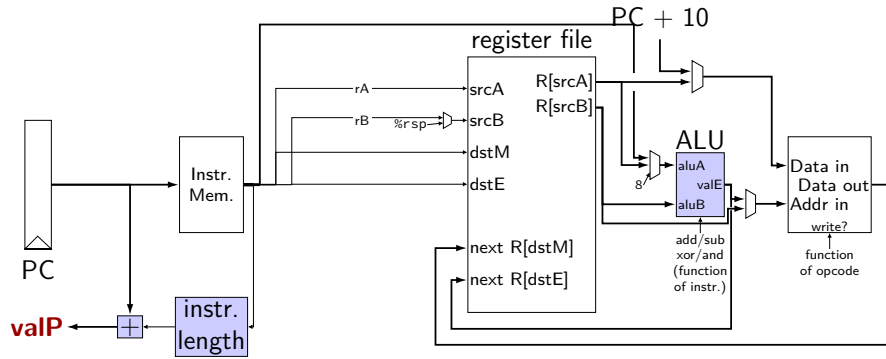
46

write back (1)



47

write back (1)



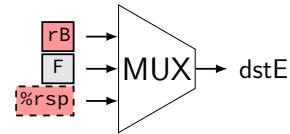
exercise: which of these instructions can this **not** work for?
 nop, pushq, mrmovq, popq, call,

SEQ: control signals for WB

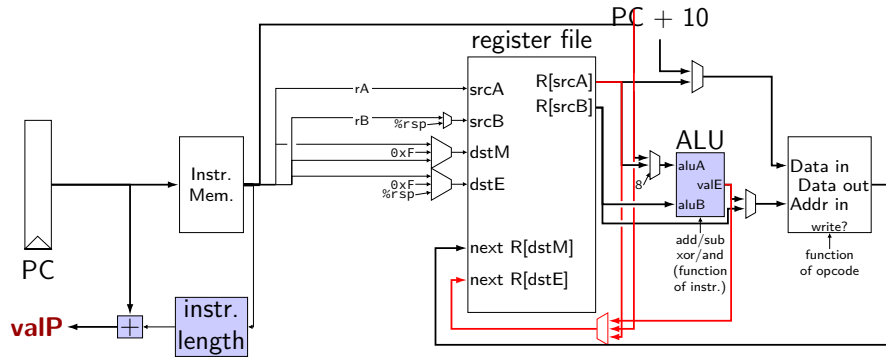
two write inputs — two needed by popq
 valM (memory output), valE (ALU output)

two register numbers
 dstM, dstE

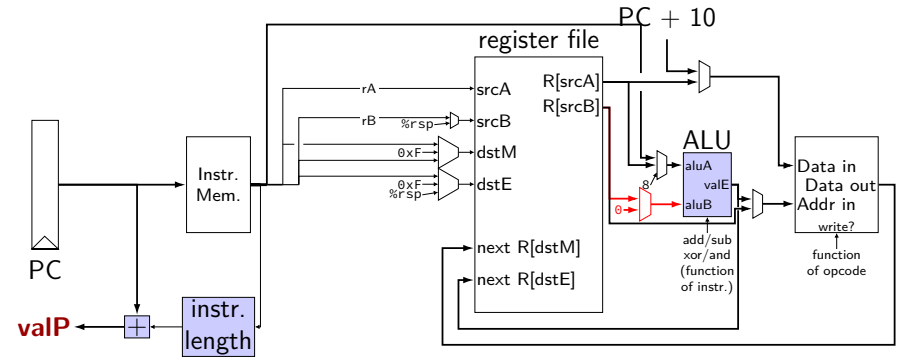
write disable — use dummy register number 0xF



write back (2a)



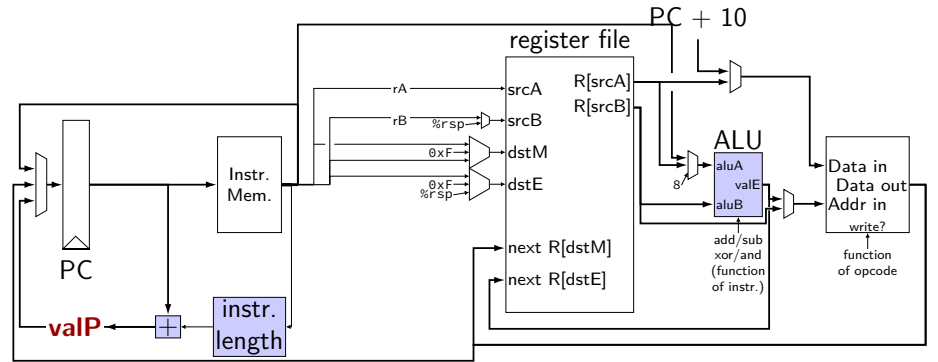
write back (2b)



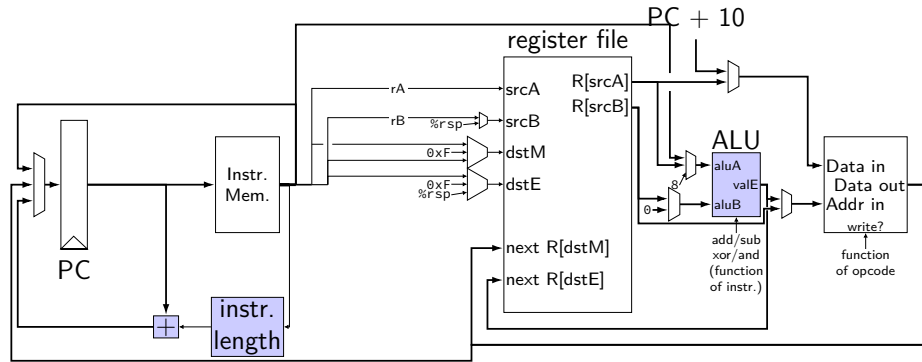
SEQ: Update PC

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

PC update

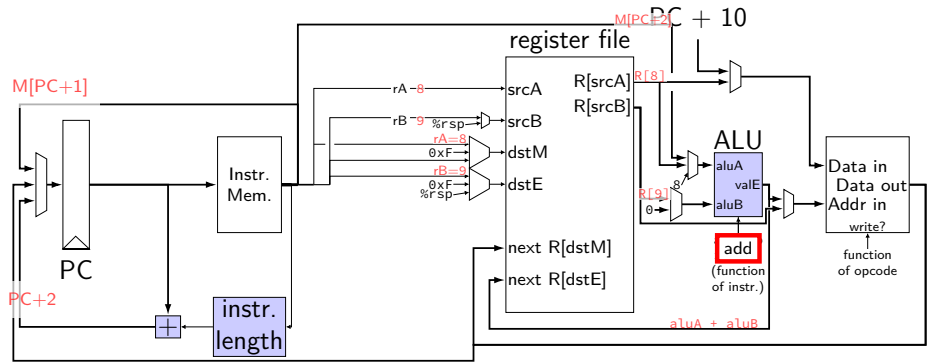


circuit: setting MUXes



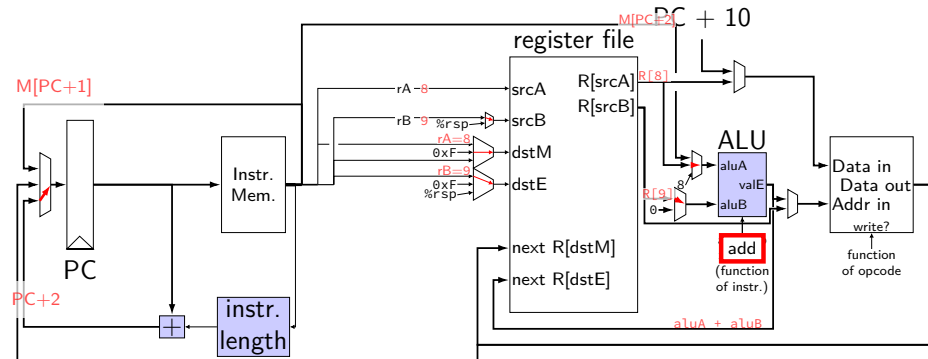
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn
 Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



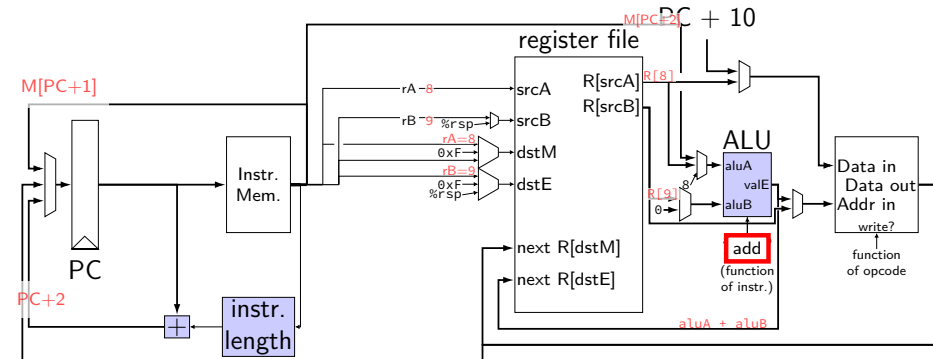
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn
 Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes

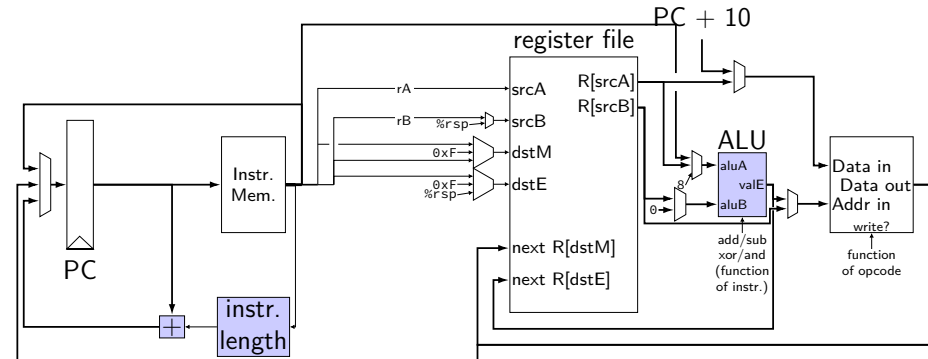


MUXes — PC, dstM, dstE, aluA, aluB, dmemIn
 Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes

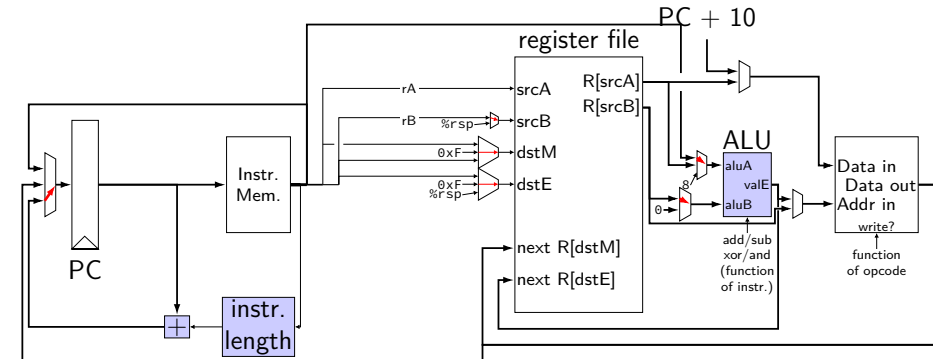


circuit: setting MUXes



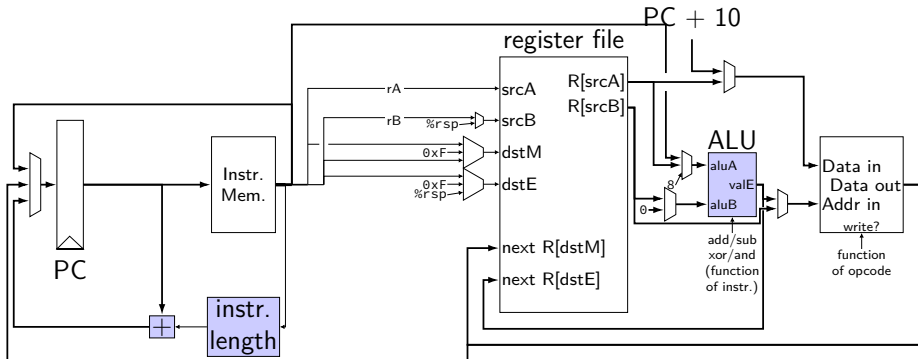
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn
 Exercise: what do they select for `rmmovq`?

circuit: setting MUXes



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

circuit: setting MUXes



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

53

Summary

- each instruction takes one cycle
- divided into stages for **design convenience**
- read values **from previous cycle**
- send **new values** to state components
- control what is sent with **MUXes**

54