SEQ part 3

Samira Khan
The slides are prepared by Charles Reiss

Review

each instruction takes one cycle

read values from previous cycle

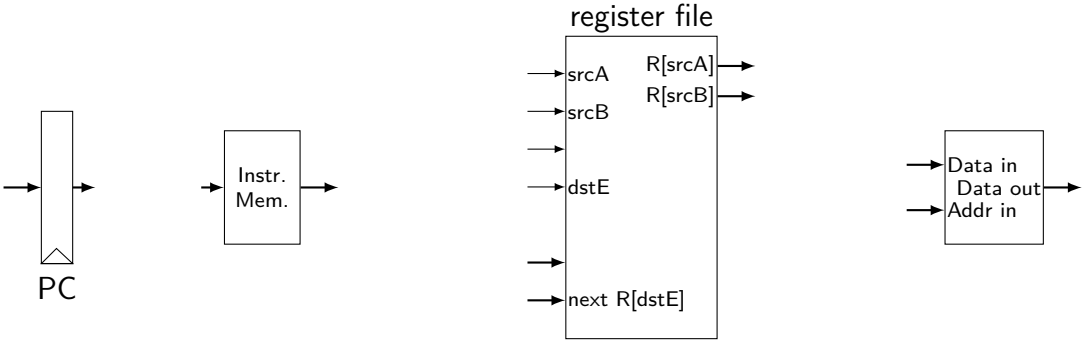
send new values to state components

control what is sent with MUXes

simple ISA 4: mov-to-register

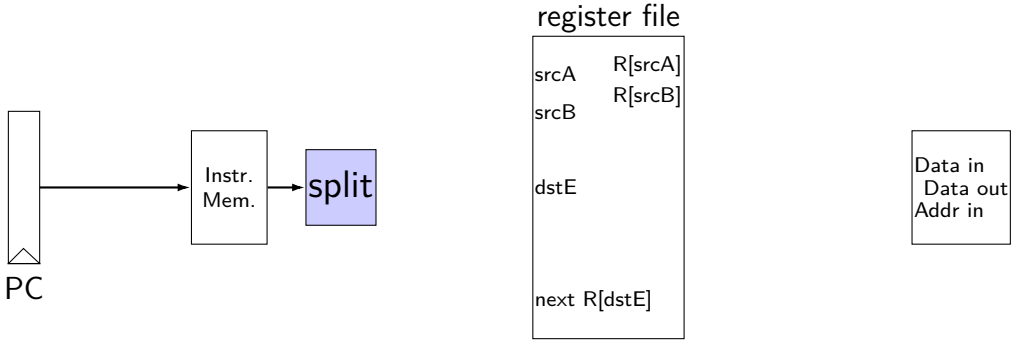
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

mov-to-register CPU



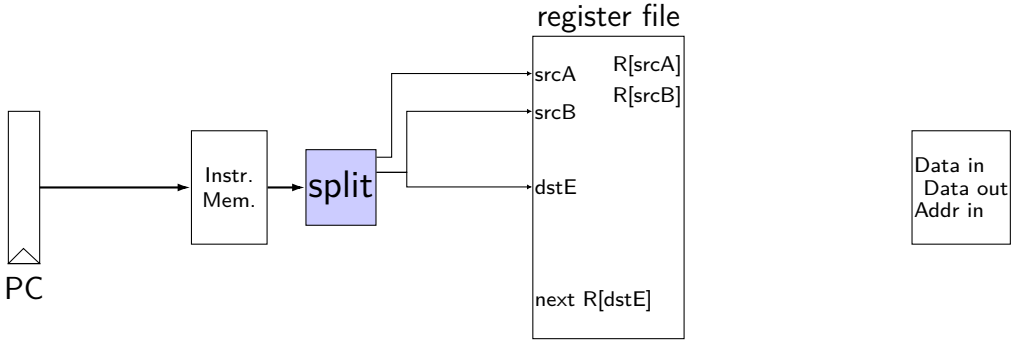
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



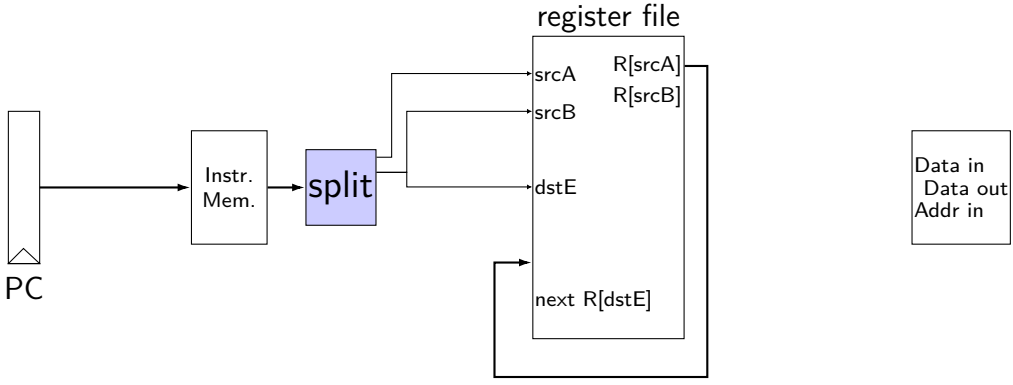
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



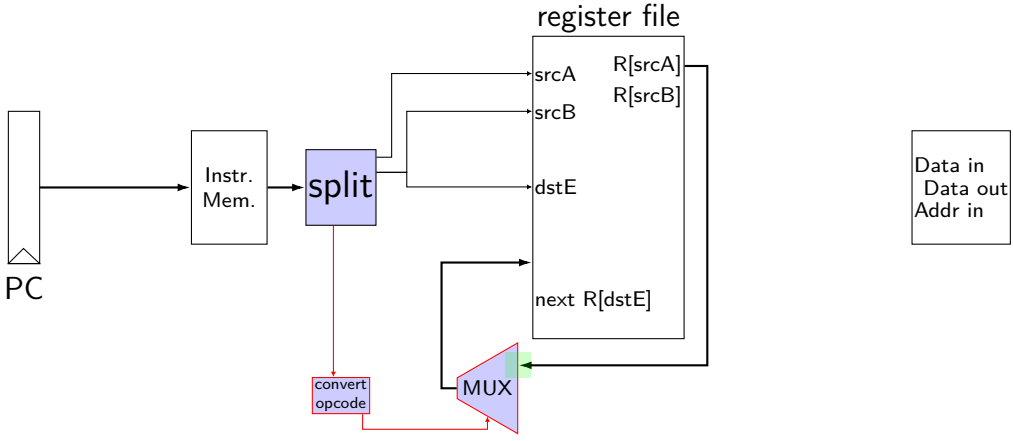
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



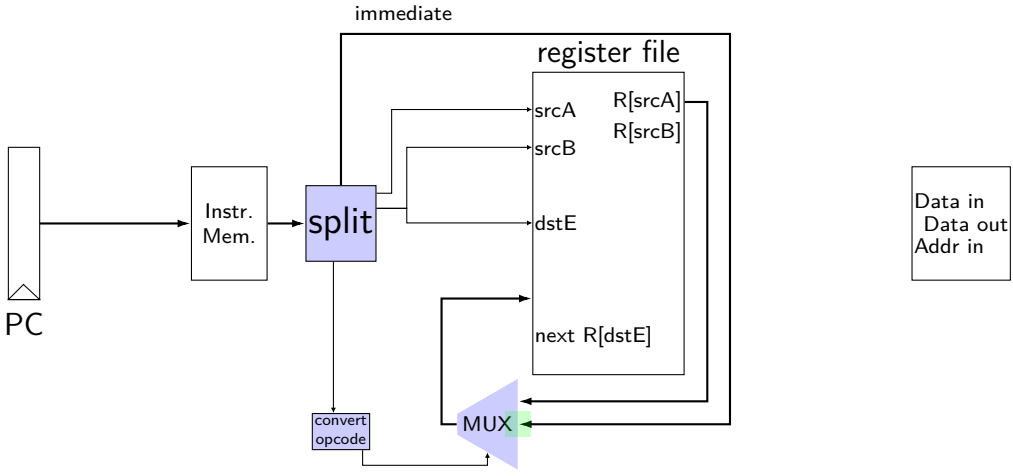
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

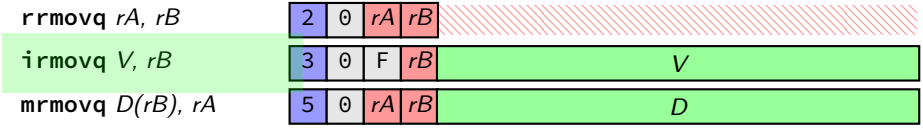
mov-to-register CPU



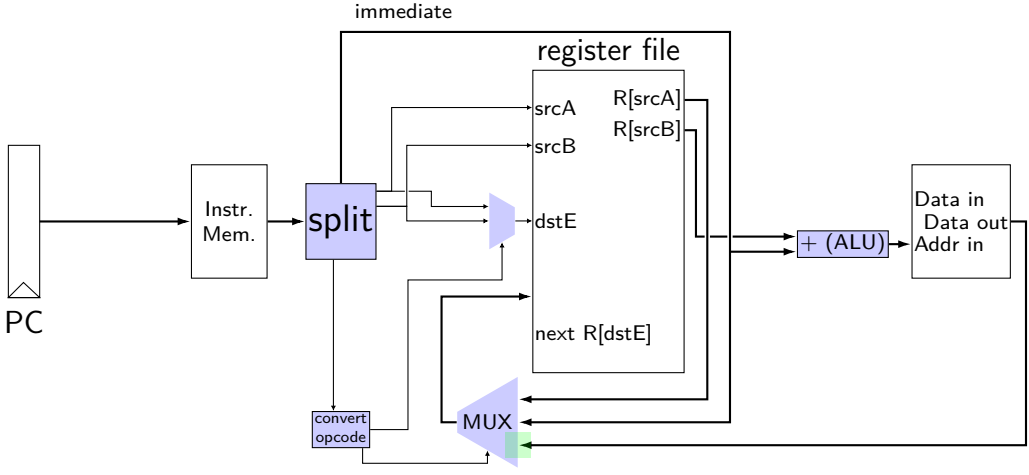
rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



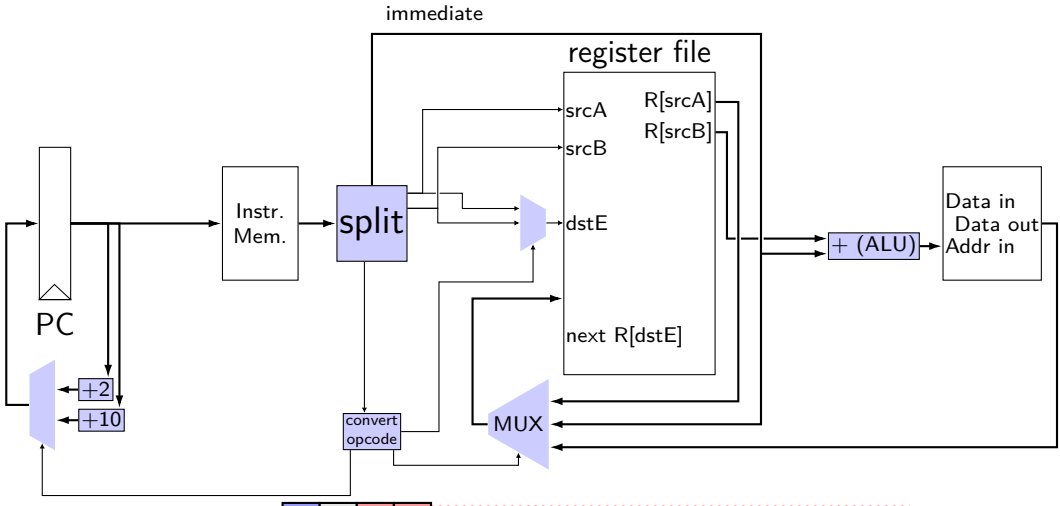


mov-to-register CPU



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

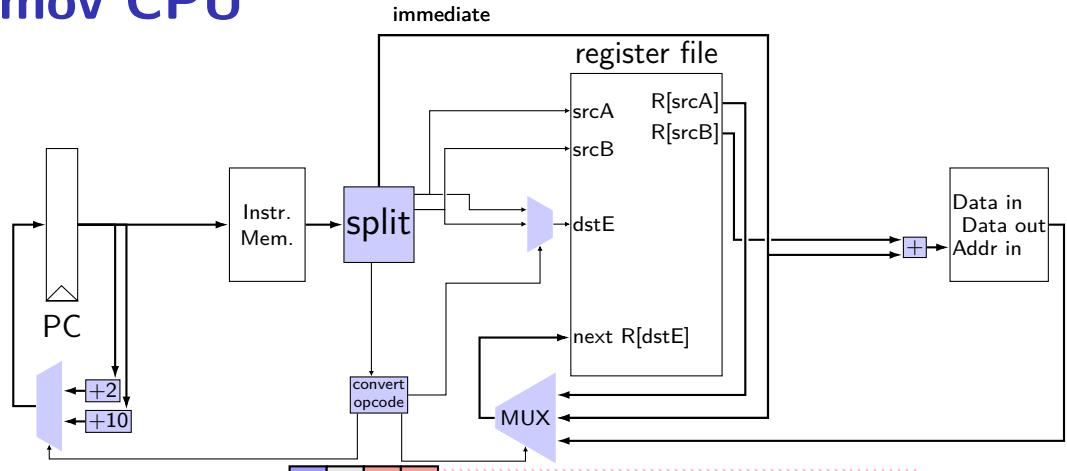


rrmovq *rA*, *rB*	2	0	*rA*	*rB*	
irmovq *V*, *rB*	3	0	F	*rB*	*V*
mrmovq *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

simple ISA 4B: mov

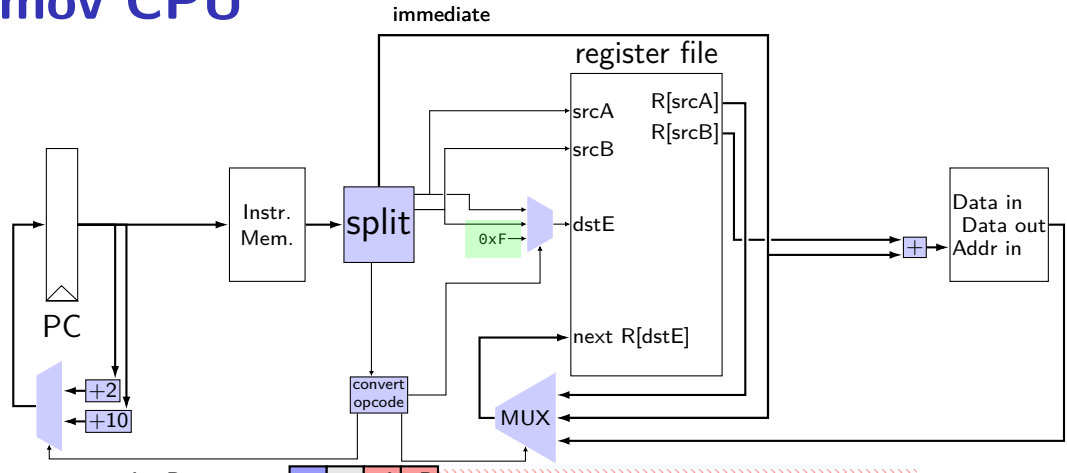
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

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

mov CPU



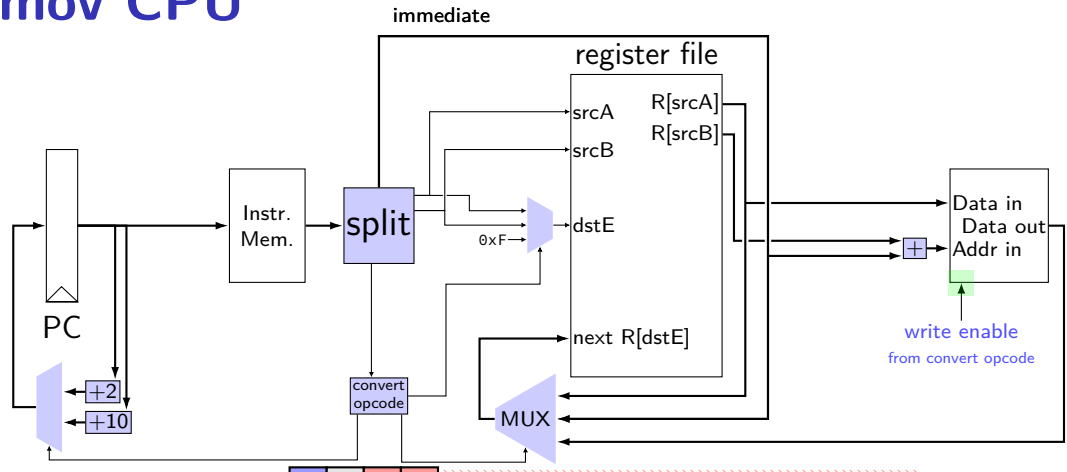
rrmovq *rA, rB*	2	0	*rA*	*rB*	
irmovq *V, rB*	3	0	F	*rB*	*V*
mrmovq *D(rB), rA*	5	0	*rA*	*rB*	*D*
rmmovq *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU



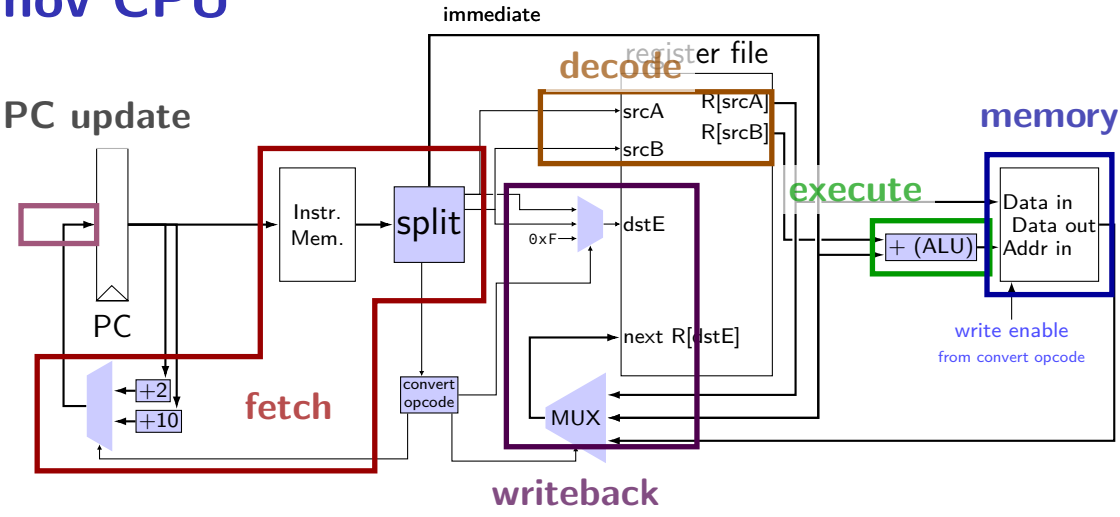
rrmovq *rA, rB*	2	0	*rA*	*rB*	
irmovq *V, rB*	3	0	F	*rB*	*V*
mrmovq *D(rB), rA*	5	0	*rA*	*rB*	*D*
rmmovq *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU

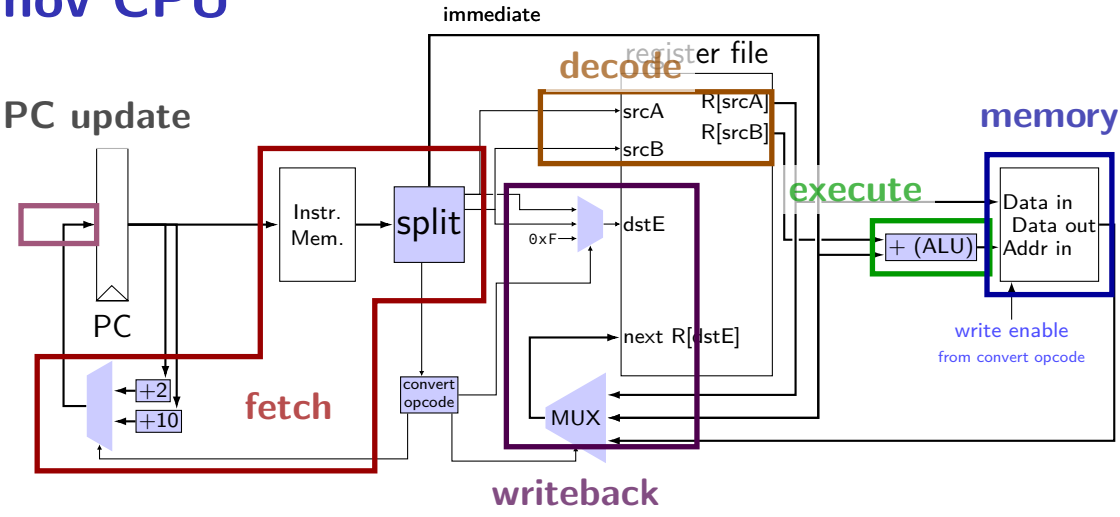


rrmovq *rA, rB*	2	0	*rA*	*rB*	
irmovq *V, rB*	3	0	F	*rB*	*V*
mrmovq *D(rB), rA*	5	0	*rA*	*rB*	*D*
rmmovq *rA, D(rB)*	4	0	*rA*	*rB*	*D*

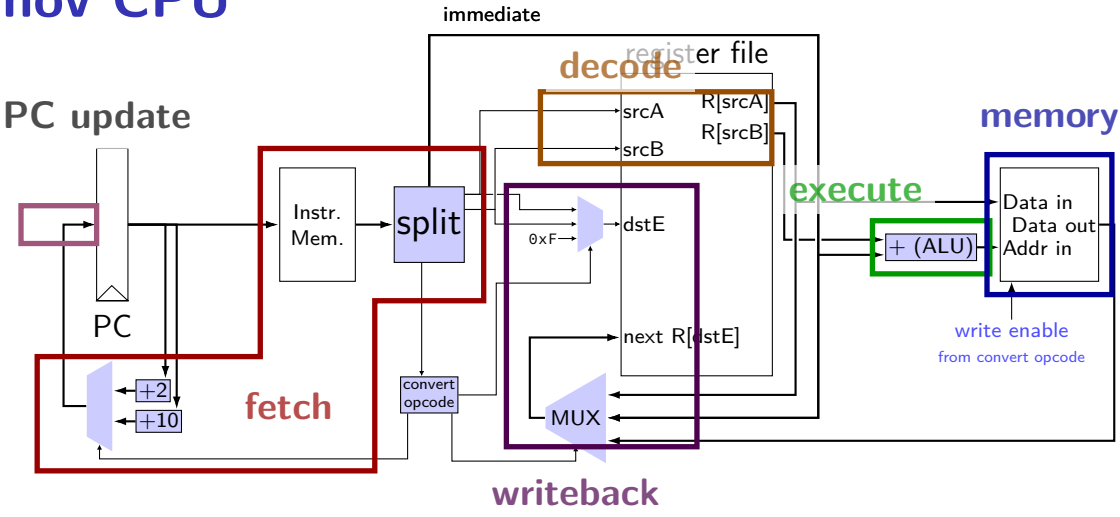
mov CPU


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

mov CPU


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

mov CPU


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

stages example: nop

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

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

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

stages example: nop

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

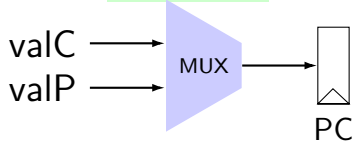
$\leftarrow$ means putting value on input wire to PC register

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

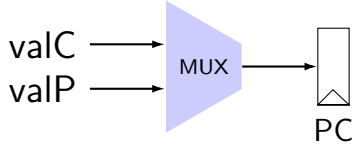
stages example: nop/jmp

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



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



stages example: rmmovq/mrmovq

stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] 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

stages example: rmmovq/mrmovq

stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]
decode	valA $\leftarrow R$[rA]	
write back		R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

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

stages example: rmmovq/mrmovq

stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] 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 $\leftarrow$	+ 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

stages example: rmmovq/mrmovq

stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] 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]
		R[rA] $\leftarrow$ valM
		C $\leftarrow$ valP

assignment means:
setting address wires to valE *and*
setting value input wires to valA *and*

stages example: rmmovq/mrmovq

stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]
decode	valA valB	
execute	valE	
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

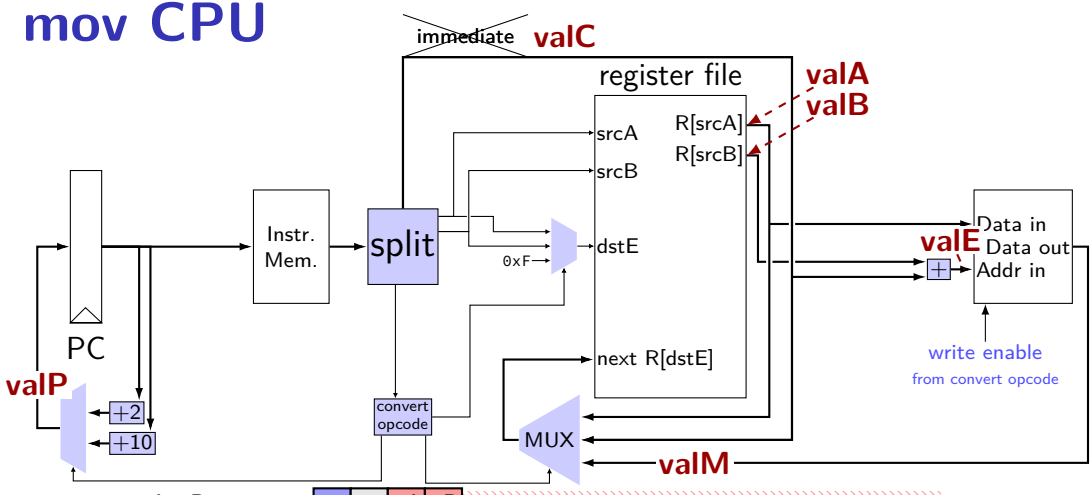
assignment means:
setting address wires to valE *and*
naming the output of the data memory

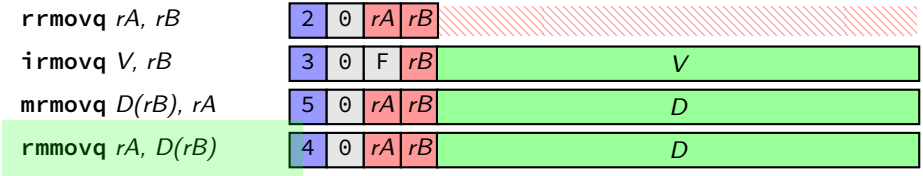
stages example: rmmovq/mrmovq

stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC + 1] valP $\leftarrow$ PC + 10 valC $\leftarrow M_8$[PC + 2]
decode	valA $\leftarrow R$[rA] valB $\leftarrow R$[rB]	valB $\leftarrow R$[rB]
execute	val	
memory	M	
write back		R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

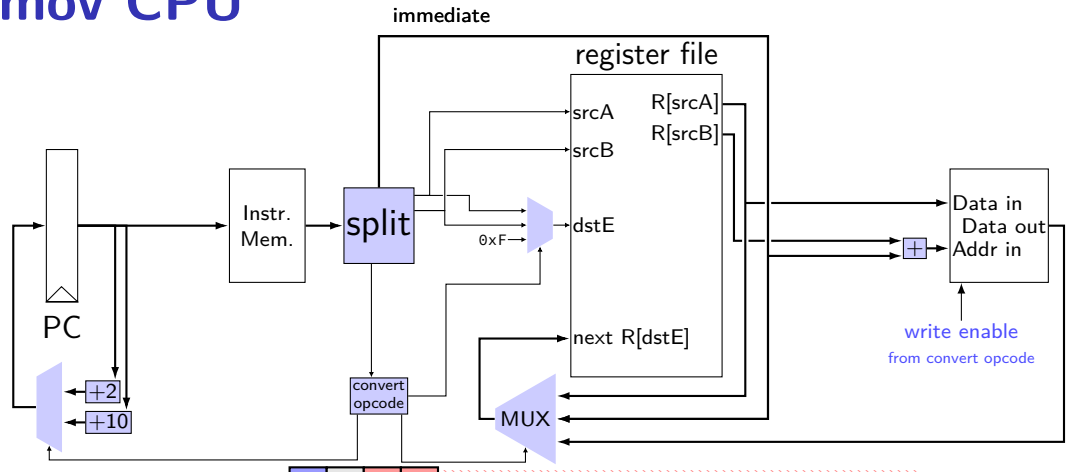
assignment means:
setting register file input wires to valM
setting register file write register number

mov CPU


immediate
valC
register file
valA
valB
srcA
srcB
R[srcA]
R[srcB]
dstE
0xF
next R[dstE]
Instr.
Mem.
split
PC
valP
+2
+10
convert
opcode
MUX
valM
valE
Data in
Data out
Addr in
write enable
from convert opcode

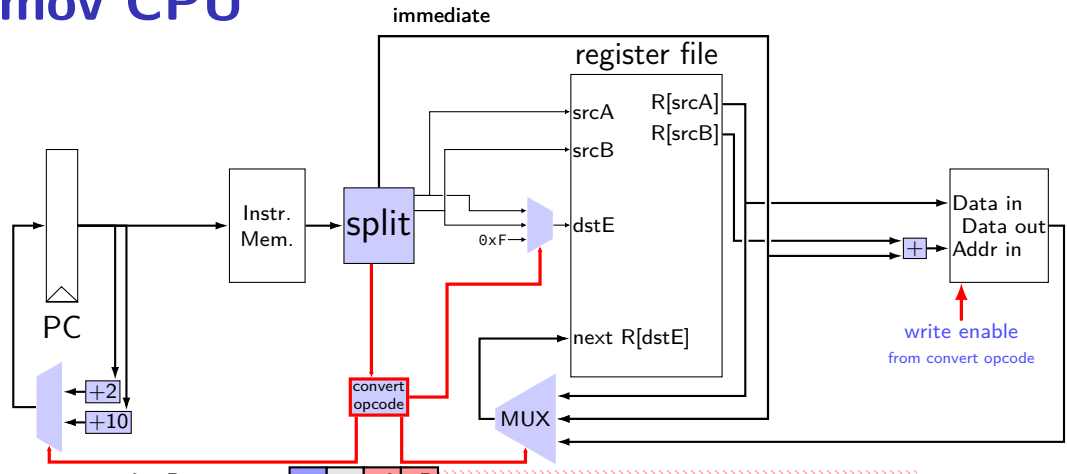

rrmovq rA, rB
irmovq V, rB
mrmovq D(rB), rA
rmmovq rA, D(rB)
2 0 rA rB
3 0 F rB V
5 0 rA rB D
4 0 rA rB D

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.

SEQ: instruction fetch

read instruction memory at PC

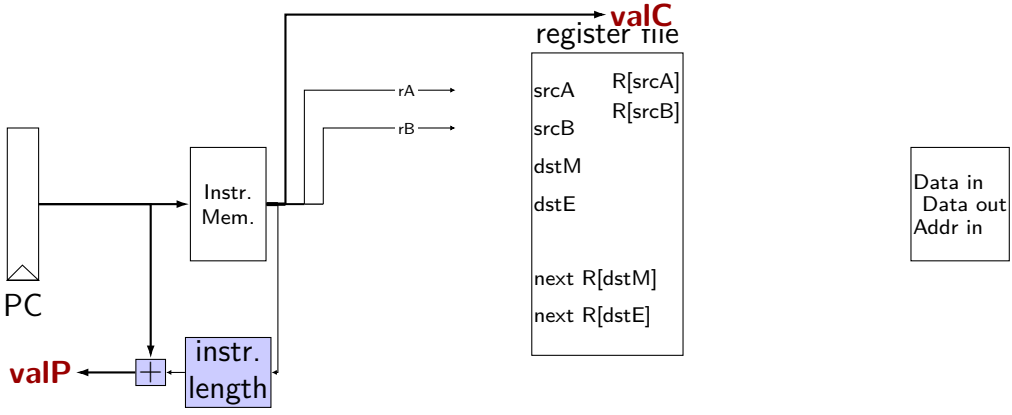
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

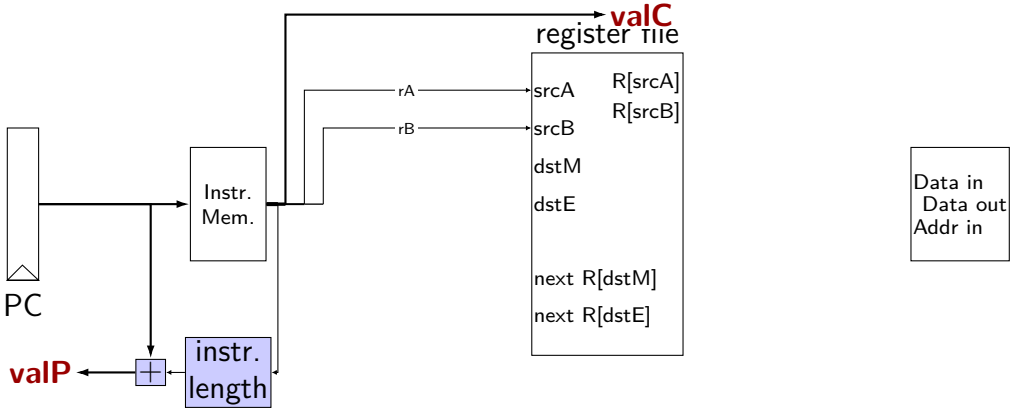


SEQ: instruction “decode”

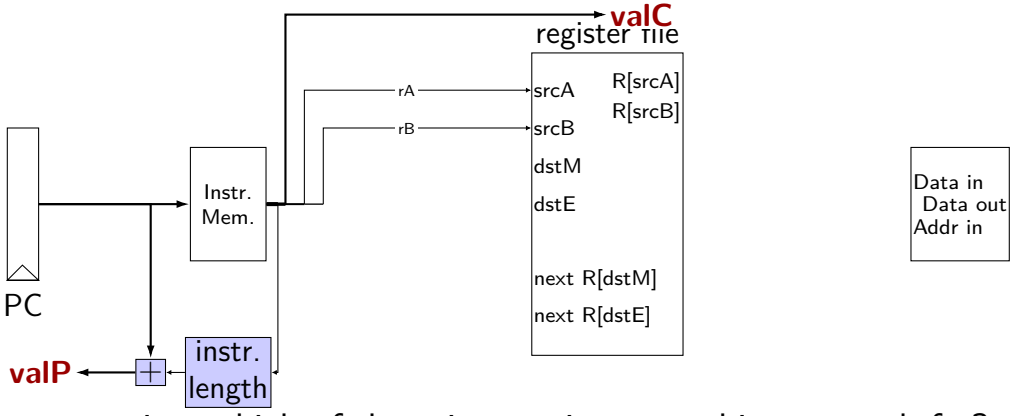
read registers

valA, valB — register values

instruction decode (1)



instruction decode (1)



exercise: which of these instructions can this **not** work for?

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

SEQ: srcA, srcB

always read rA, rB?

Problems:

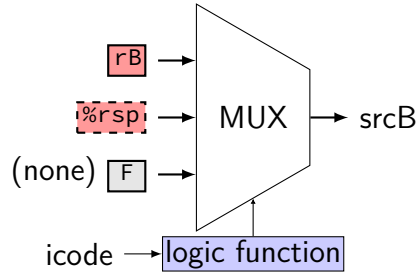
- push rA
- pop
- call
- ret

extra signals: srcA, srcB — computed input register

MUX controlled by icode

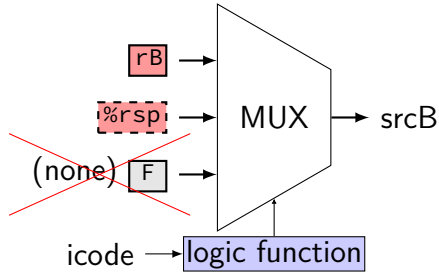
SEQ: possible registers to read

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

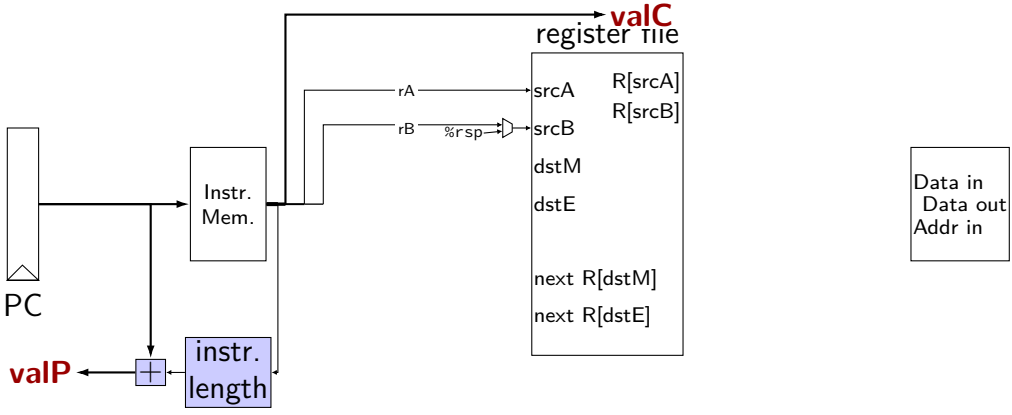


SEQ: possible registers to read

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)

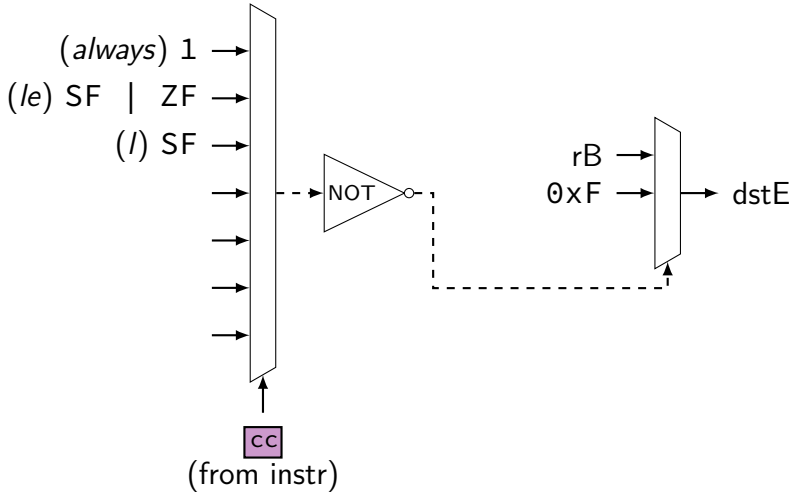
- valE — ALU output

read prior condition codes

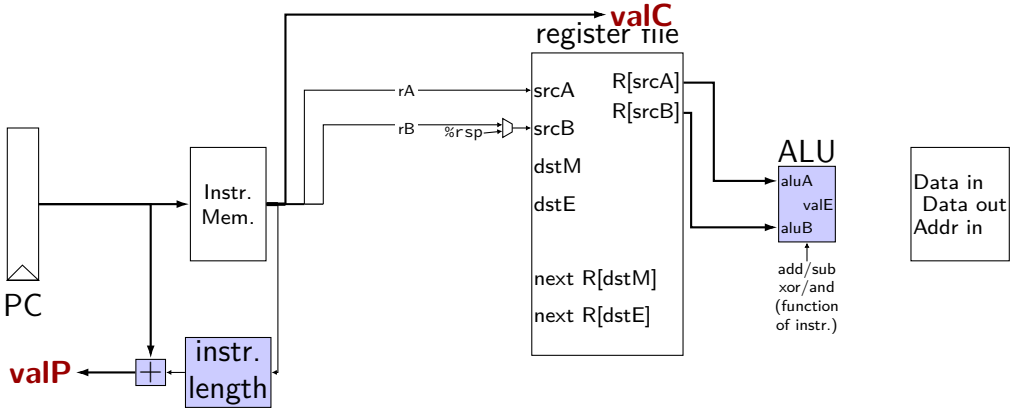
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

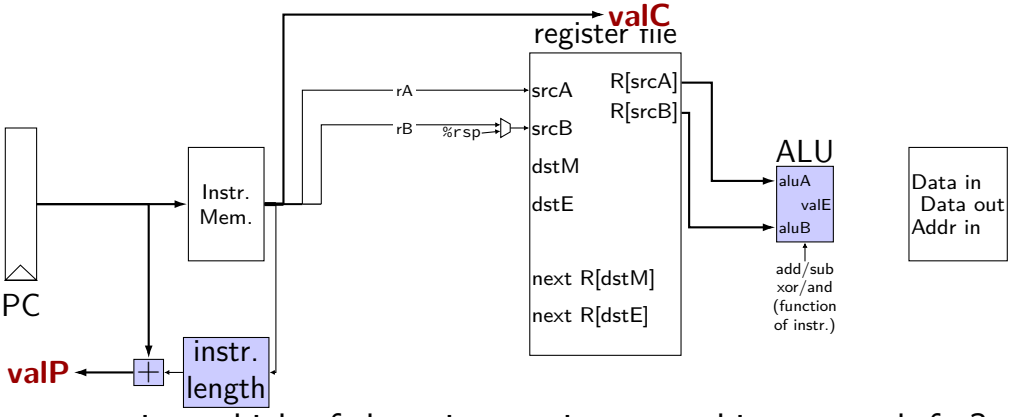
using condition codes: cmov*



execute (1)


valC
register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
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 can this **not** work for?

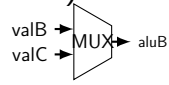
`nop`, `addq`, `mrmovq`, `popq`, `call`,

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

- **mrmovq**
- **rmmovq**



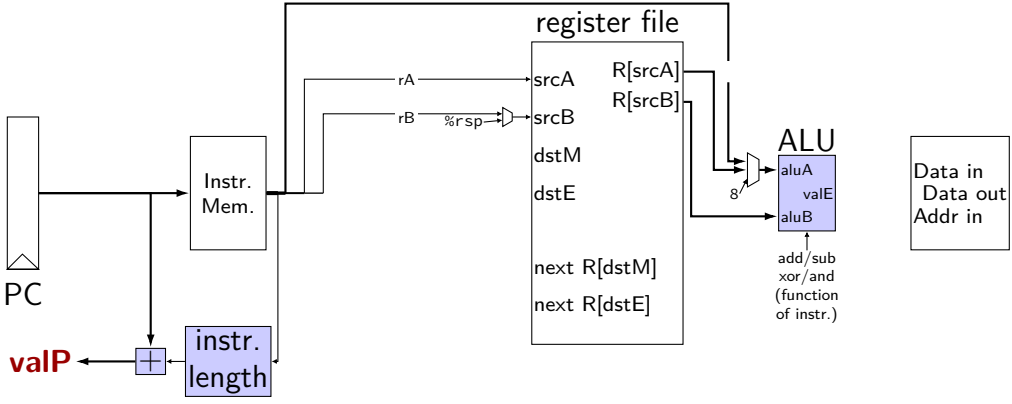
no, constants: (rsp +/- 8)

- **pushq**
- **popq**
- **call**
- **ret**

extra signals: aluA, aluB

computed ALU input values

execute (2)

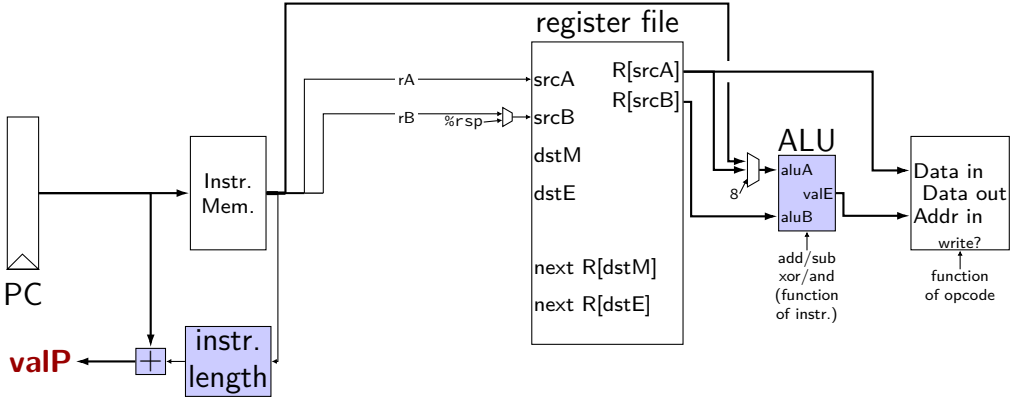

register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
Instr.
Mem.
PC
valP
instr.
length

SEQ: Memory

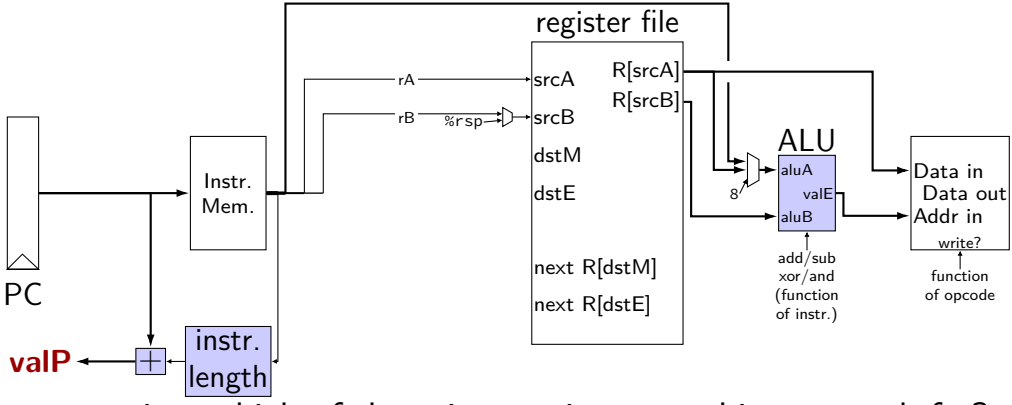
read or write data memory

valM — value read from memory (if any)

memory (1)


register file
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
Instr.
Mem.
PC
valP
instr.
length

memory (1)



exercise: which of these instructions can this **not** work for?

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

SEQ: control signals for memory

read/write — read enable? write enable?

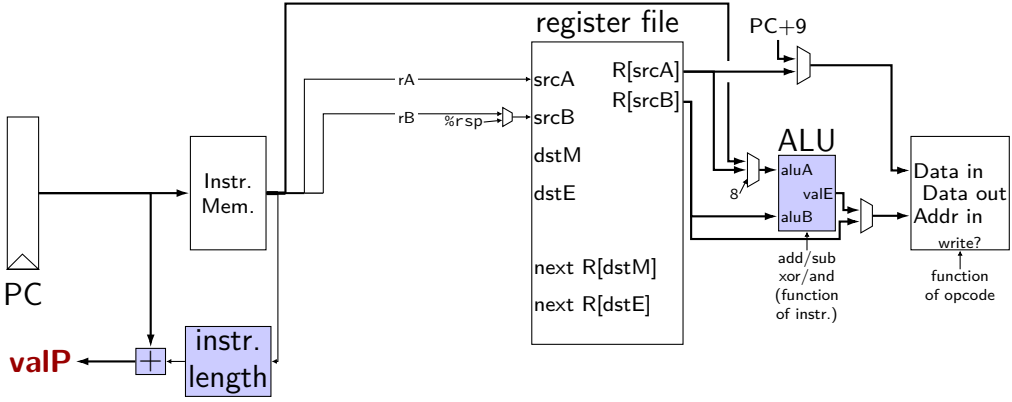
Addr — address

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

Data — value to write

- mostly valA
- tricky case: **`call`**

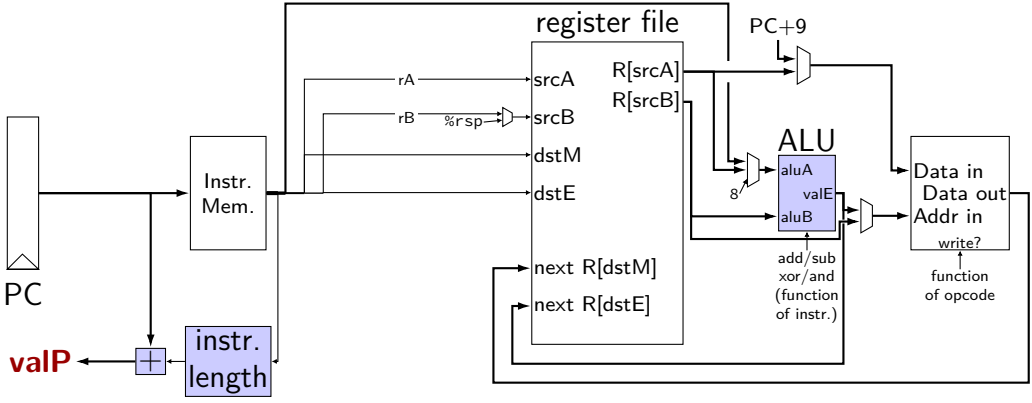
memory (2)


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
Data in
Data out
Addr in
write?
add/sub
xor/and
(function
of instr.)
function
of opcode
Instr.
Mem.
PC
valP
instr.
length

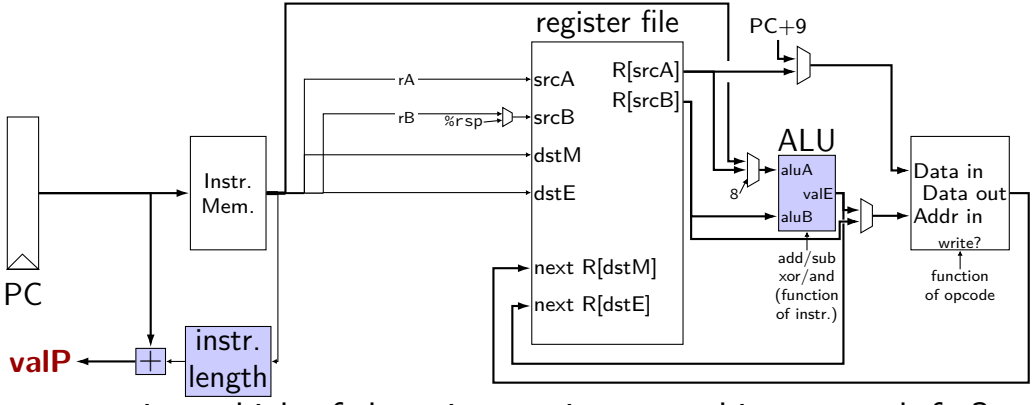
SEQ: write back

write registers

write back (1)


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



exercise: which of these instructions can this **not** work for?

`nop`, `pushq`, `mrmovq`, `popq`, `call`,

SEQ: control signals for WB

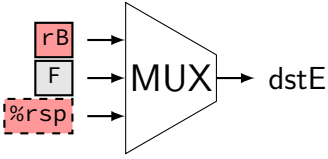
two write inputs — two needed by `popq`

valM (memory output), valE (ALU output)

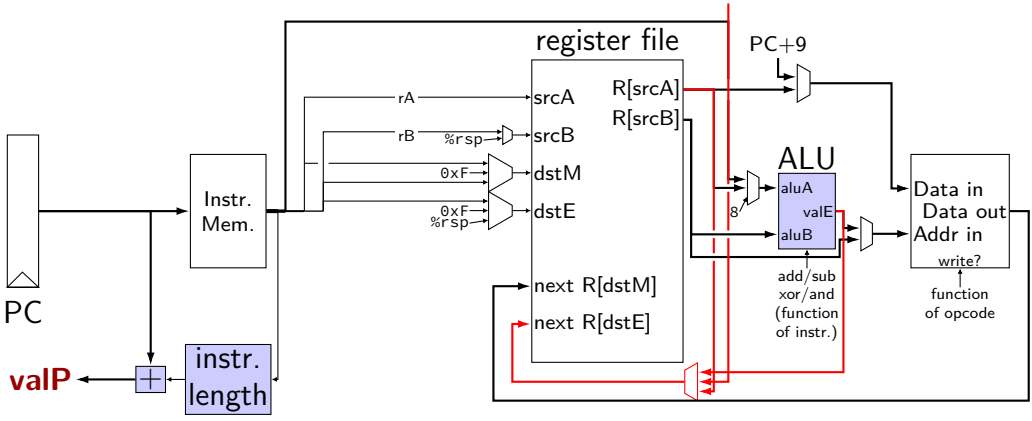
two register numbers

dstM, dstE

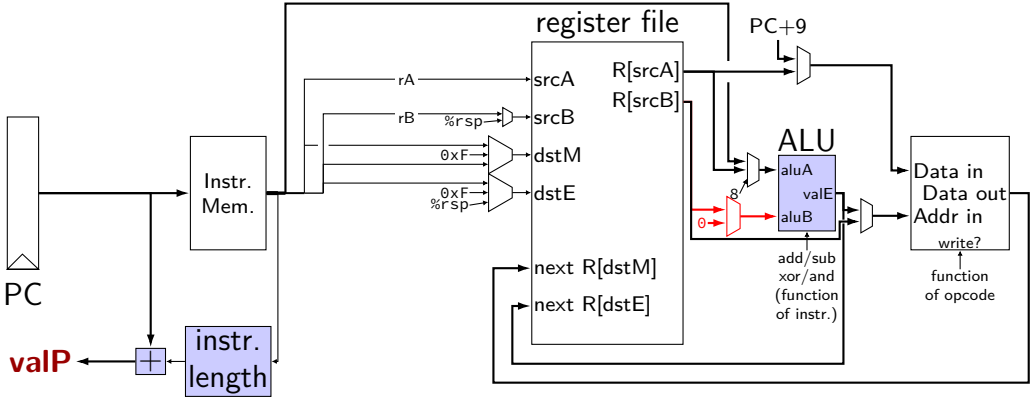
write disable — use dummy register number `0xF`



write back (2a)



write back (2b)

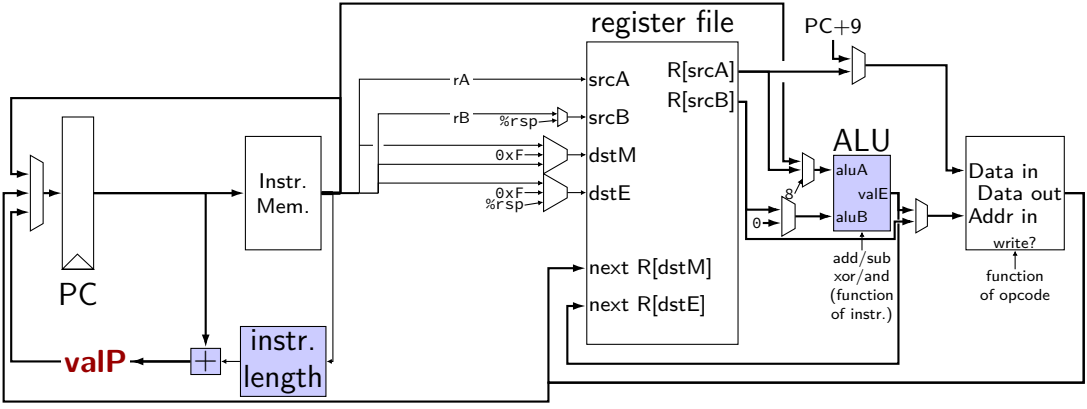


SEQ: Update PC

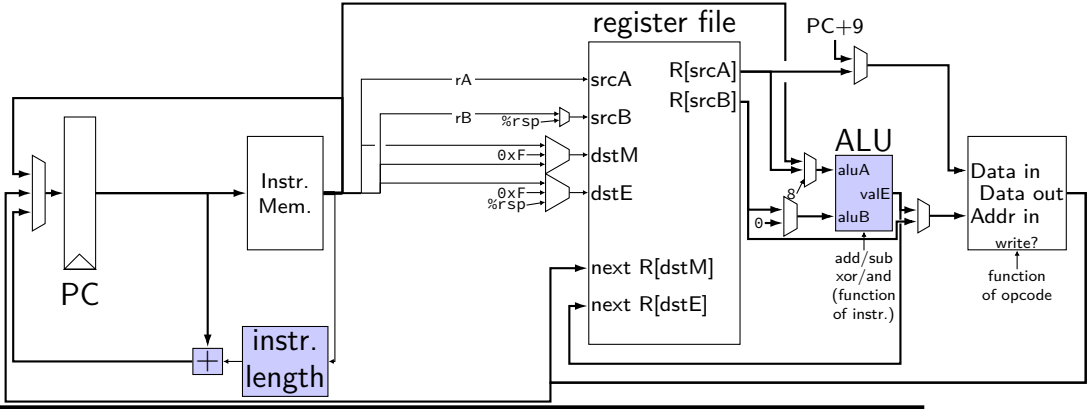
choose value for PC next cycle (input to PC register)

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

PC update


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

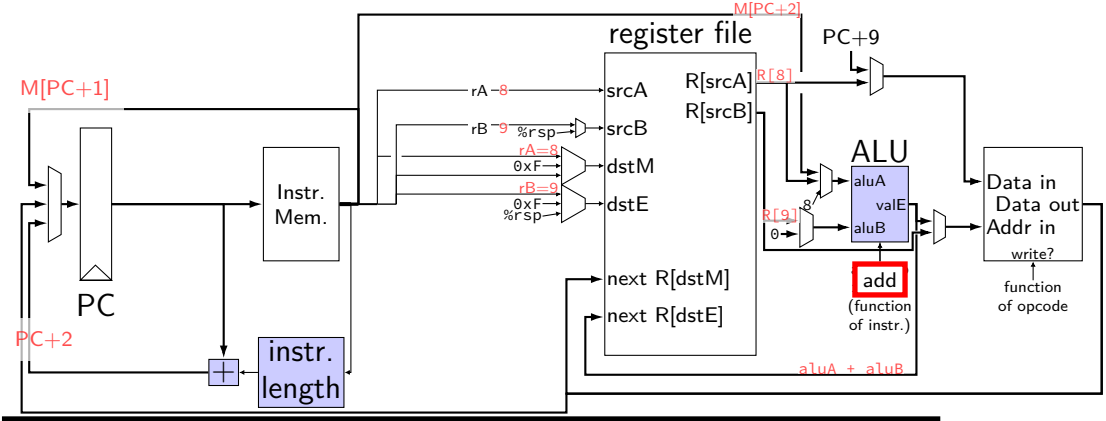
circuit: setting MUXes



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

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

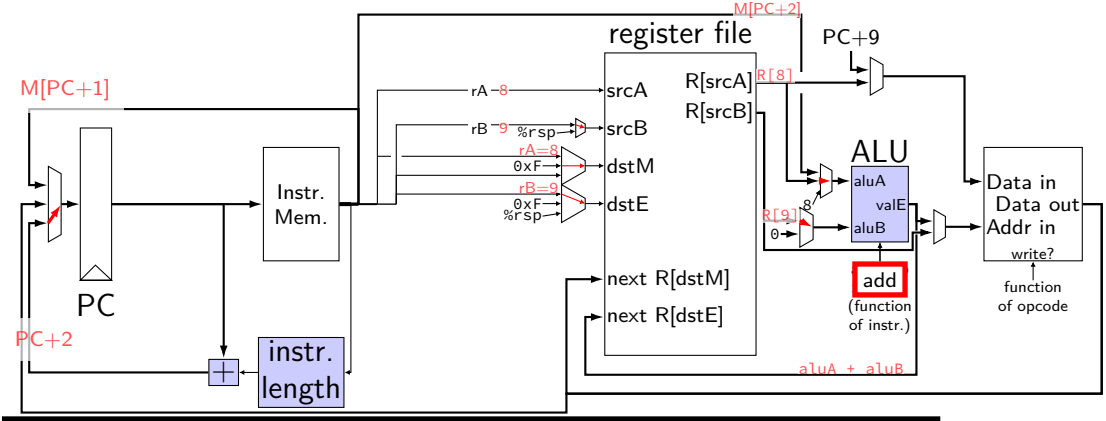
circuit: setting MUXes



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

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

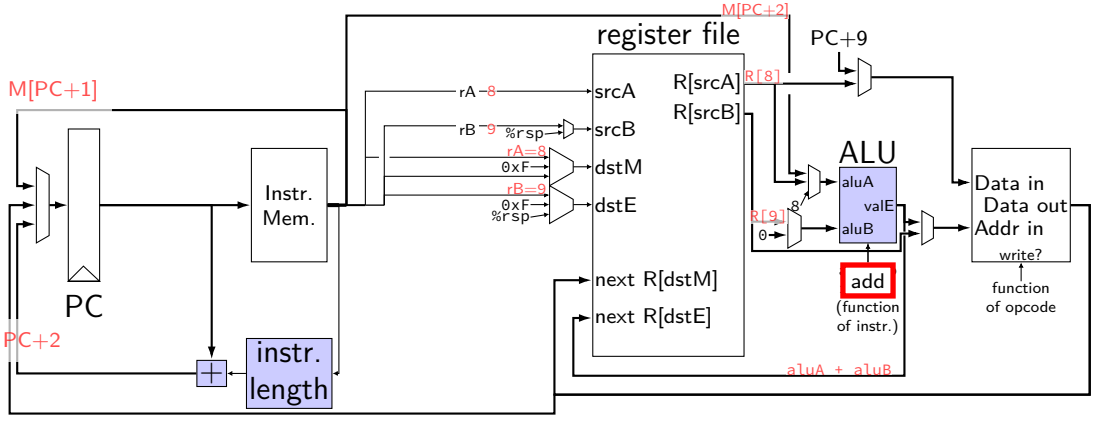
circuit: setting MUXes



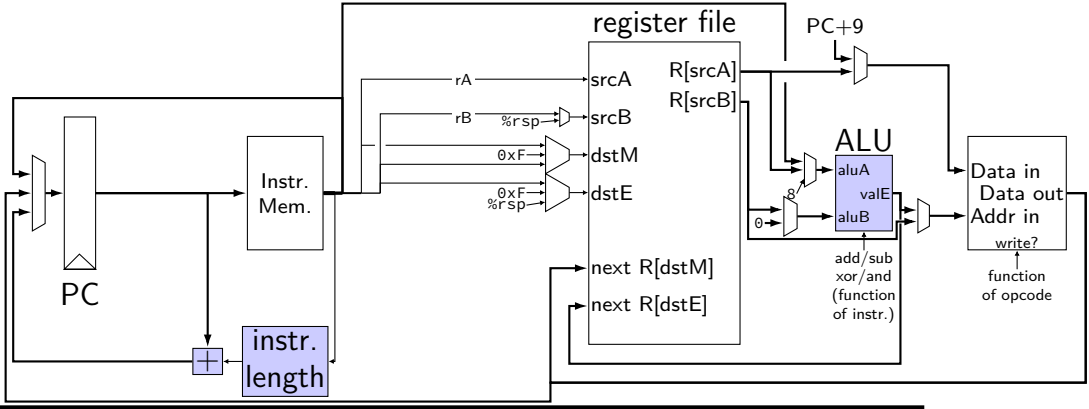
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

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

circuit: setting MUXes

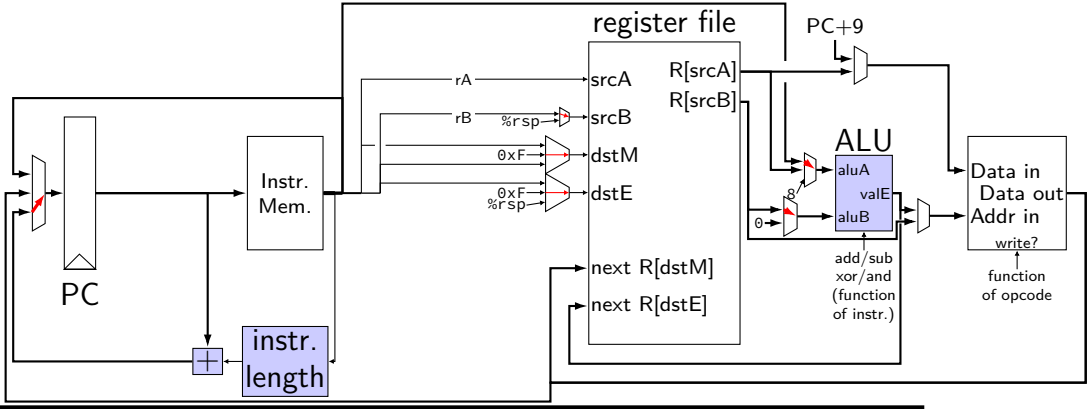


circuit: setting MUXes



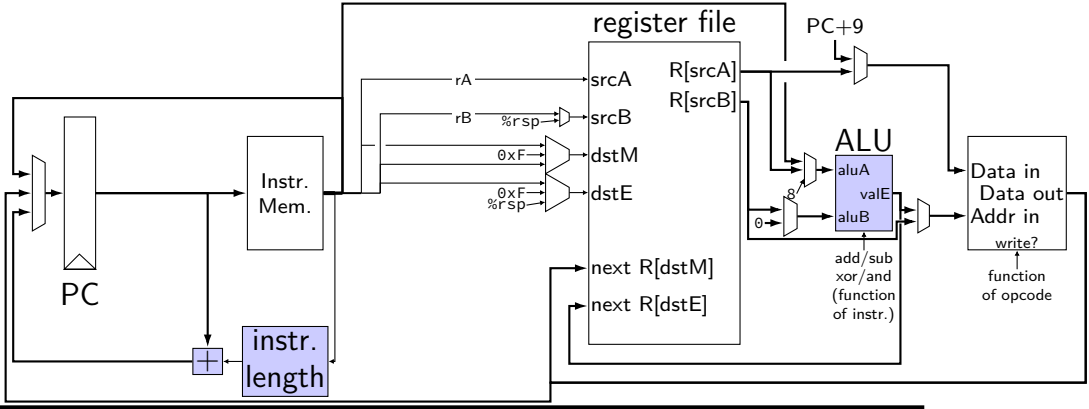
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn
Exercise: what do they select for **`rmmovq`**?

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`**?

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