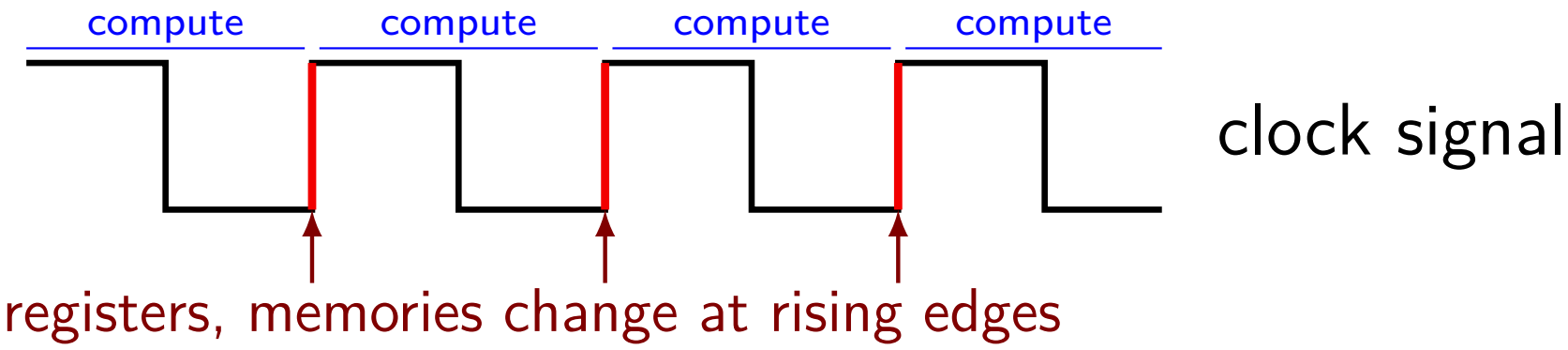
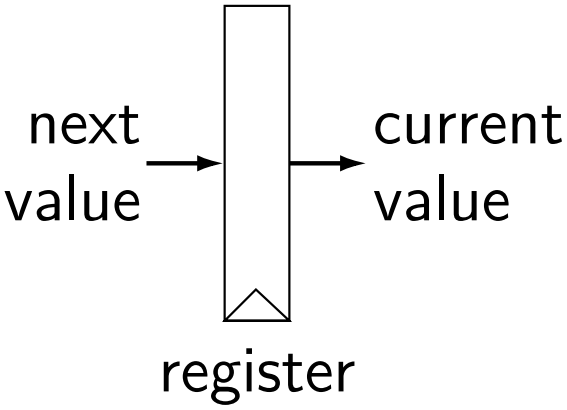
CS 3330: SEQ part 2

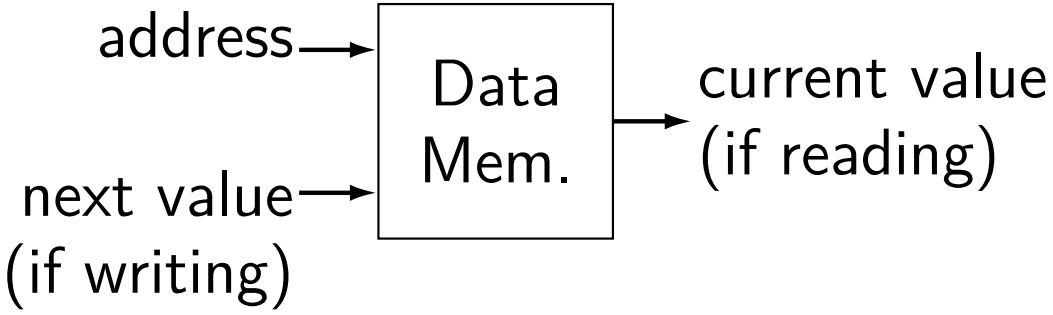
15 September 2016

Recall: Timing

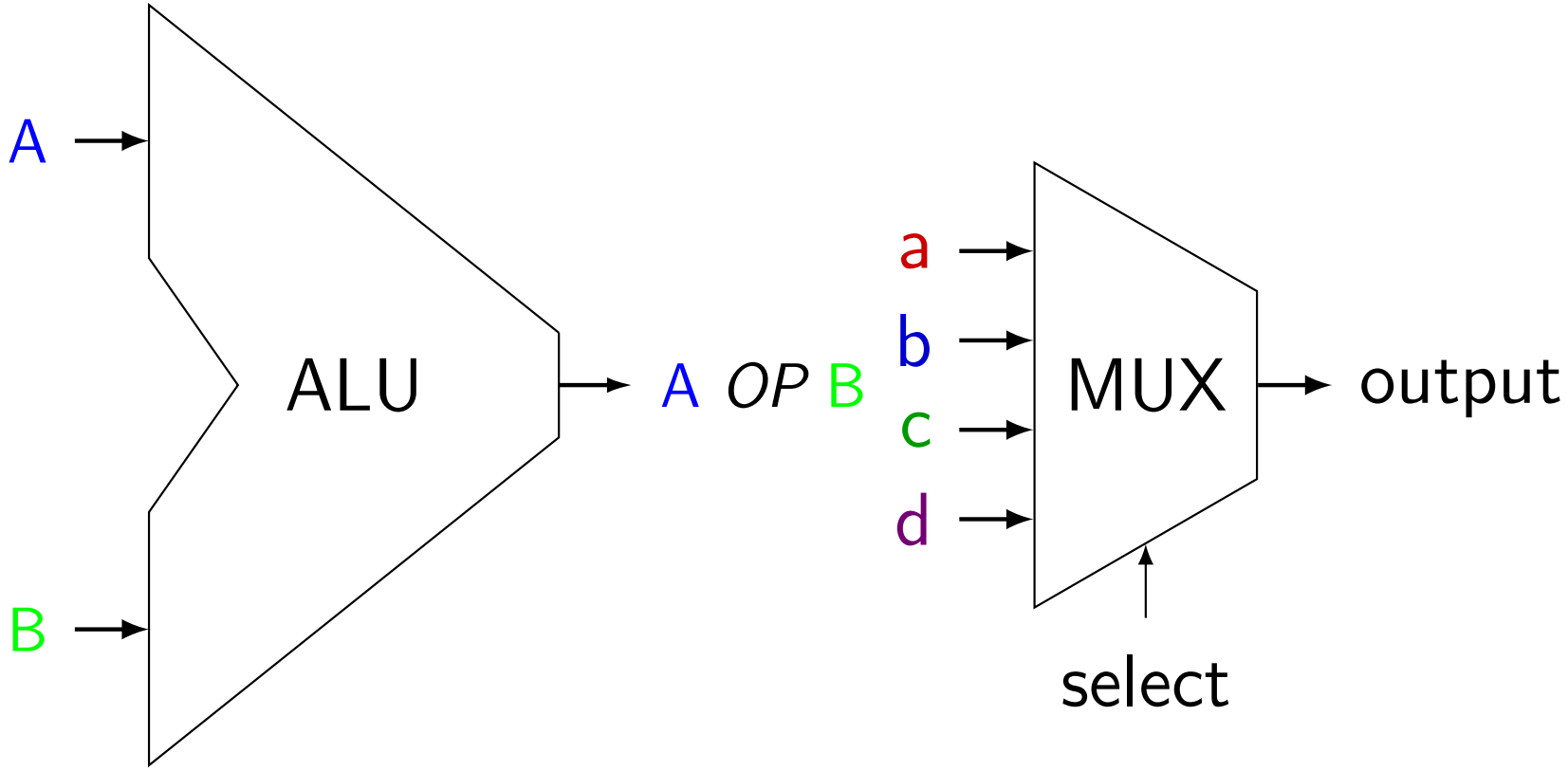
compute new values between rising edges







Recall: Computing components



MUXes — switches, select one of several inputs as output based on *select* signal.

ALU — does the primary 'math' for instructions

Recall: Stages

fetch — read instruction memory, split instruction

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

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)

SEQ: Instruction Decode

read registers

valA, valB — register values

SEQ: srcA, srcB

always read rA, rB?

Problems:

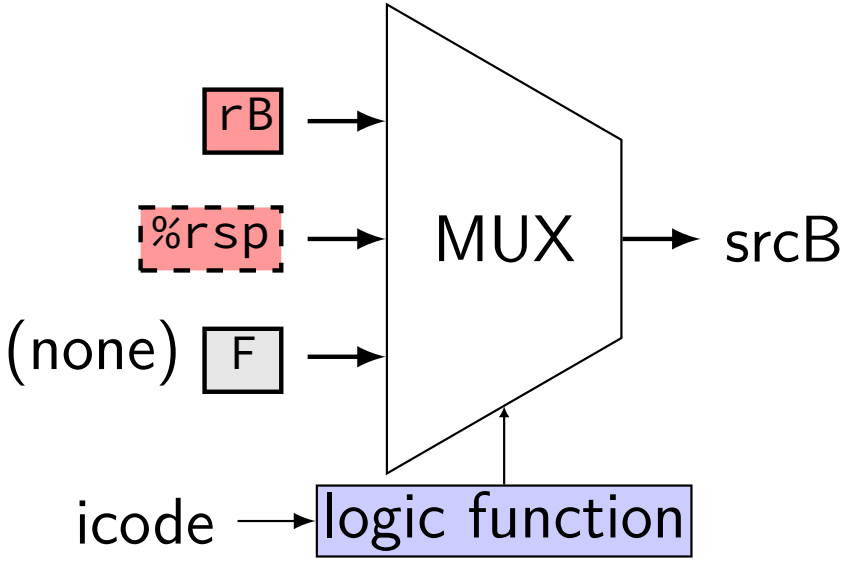
```
push rA
pop
call
ret
```

extra signals: srcA, srcB — computed input register

MUX controlled by icode

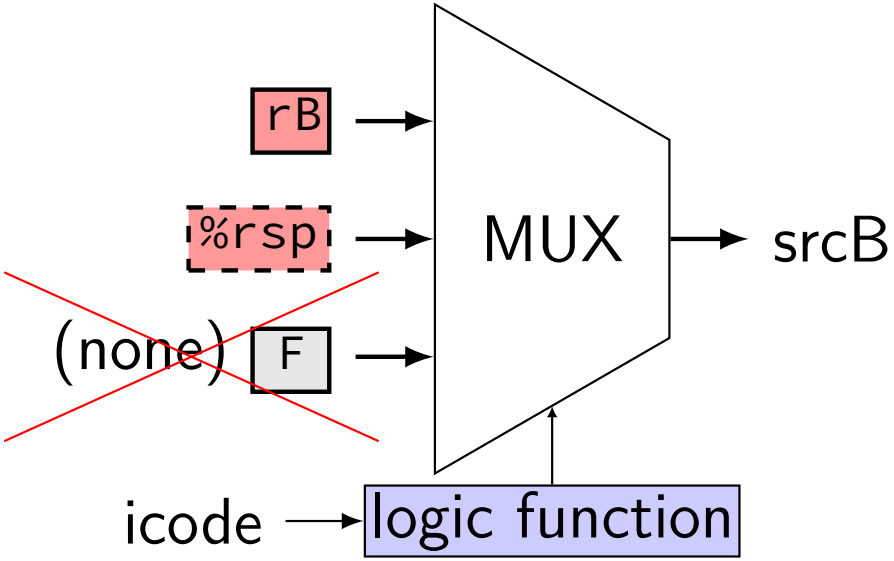
SEQ: Possible registers to read

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



SEQ: Execute

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

- valE — ALU output

read prior condition codes

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

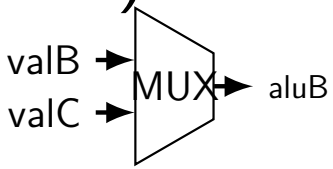
write new condition codes

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

SEQ: Memory

read or write data memory

valM — value read from memory (if any)

SEQ: Control Signals for Memory

read/write — write enable?

Addr — address

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

Data — value to write

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

SEQ: Write back

write registers

SEQ: Control Signals for WB

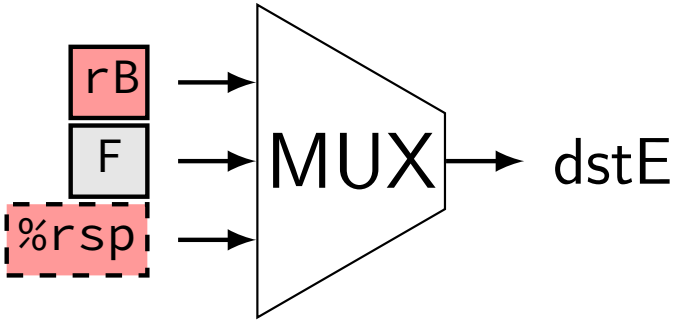
two write inputs — two needed by `popq`

valM (memory output), valE (ALU output)

two register numbers

dstM, dstE

write disable — use dummy register number `0xF`

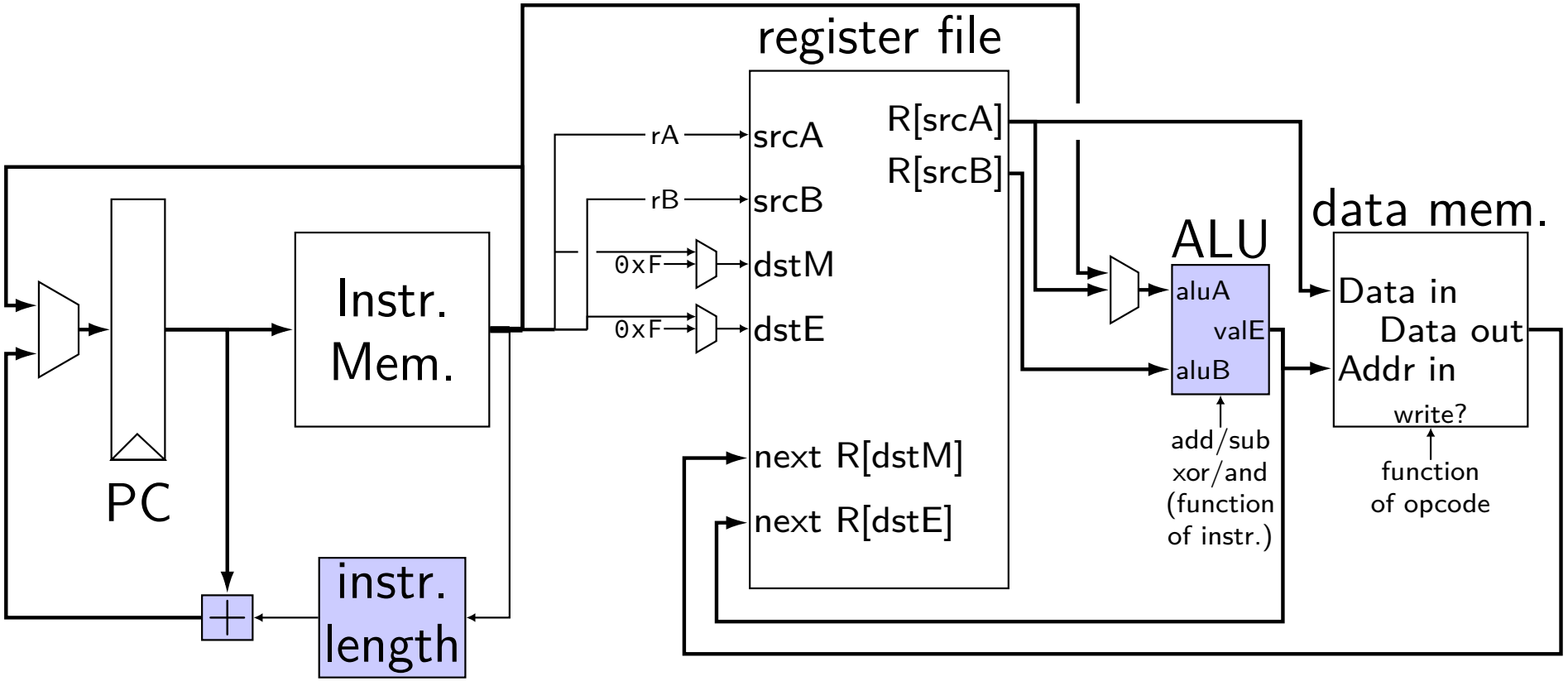


SEQ: Update PC

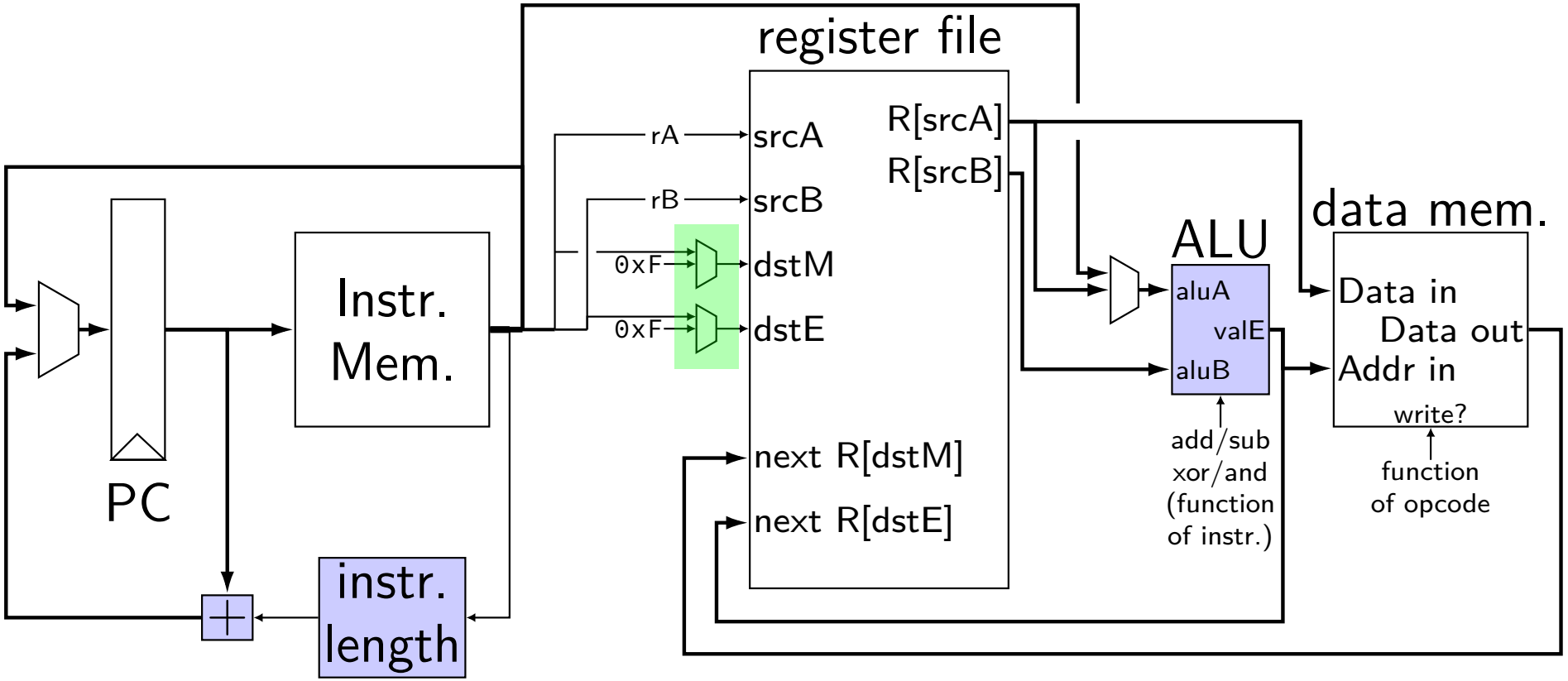
choose value for PC next cycle (input to PC register)

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

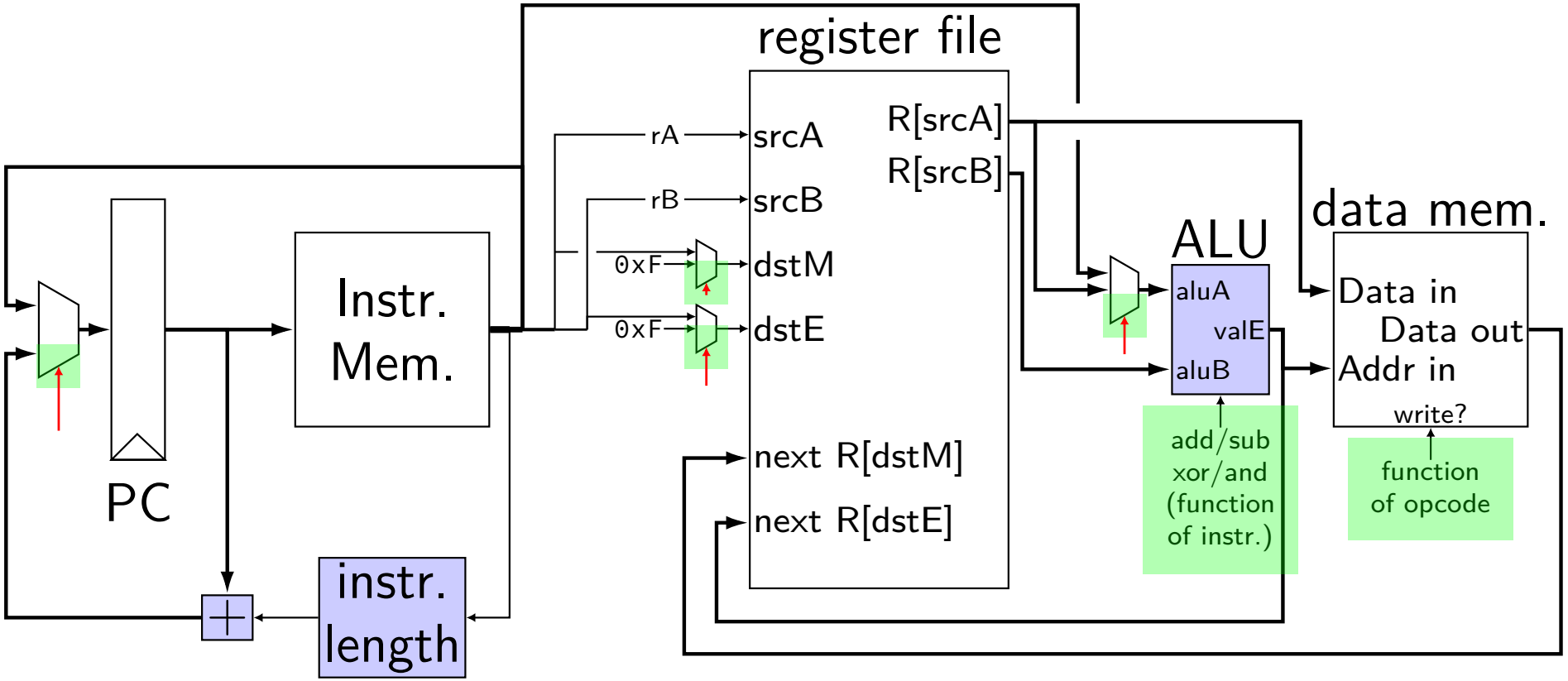
A Starting Circuit



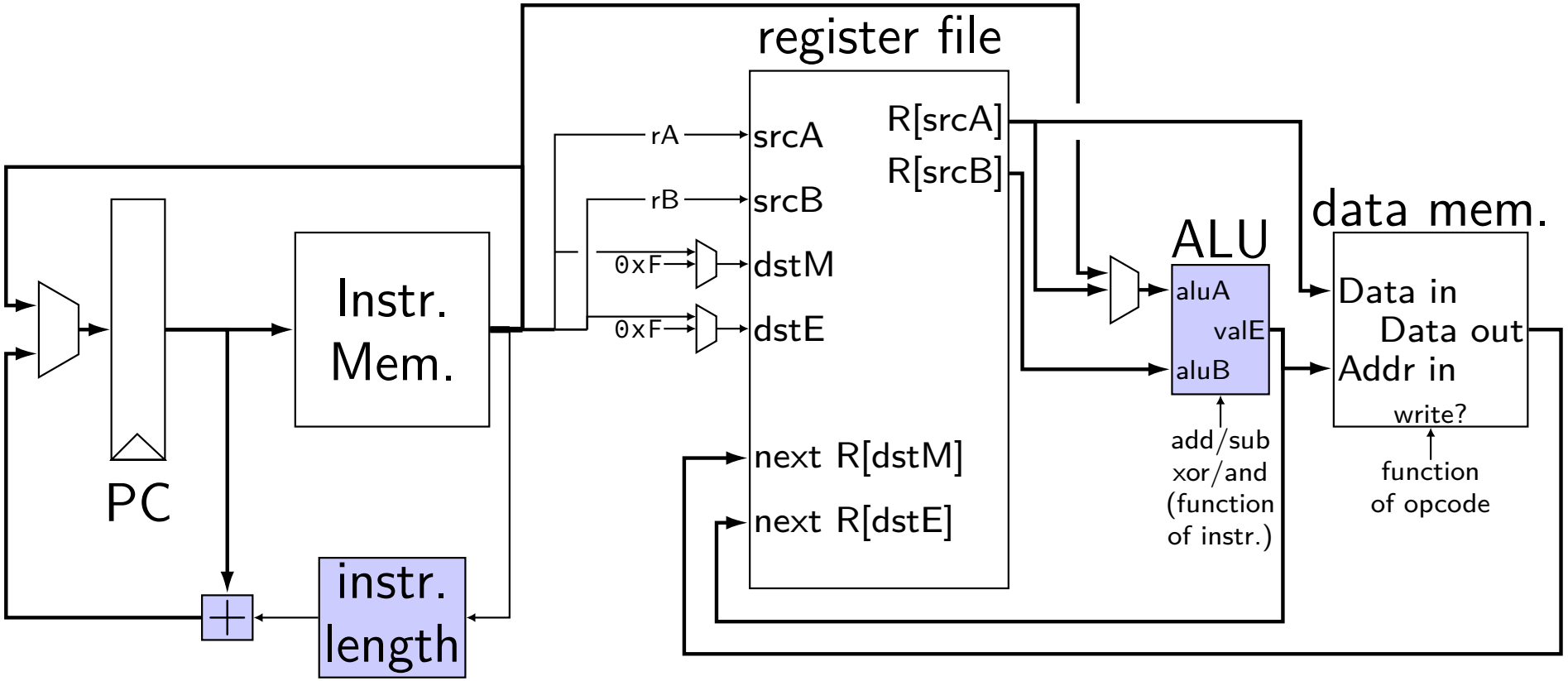
A Starting Circuit



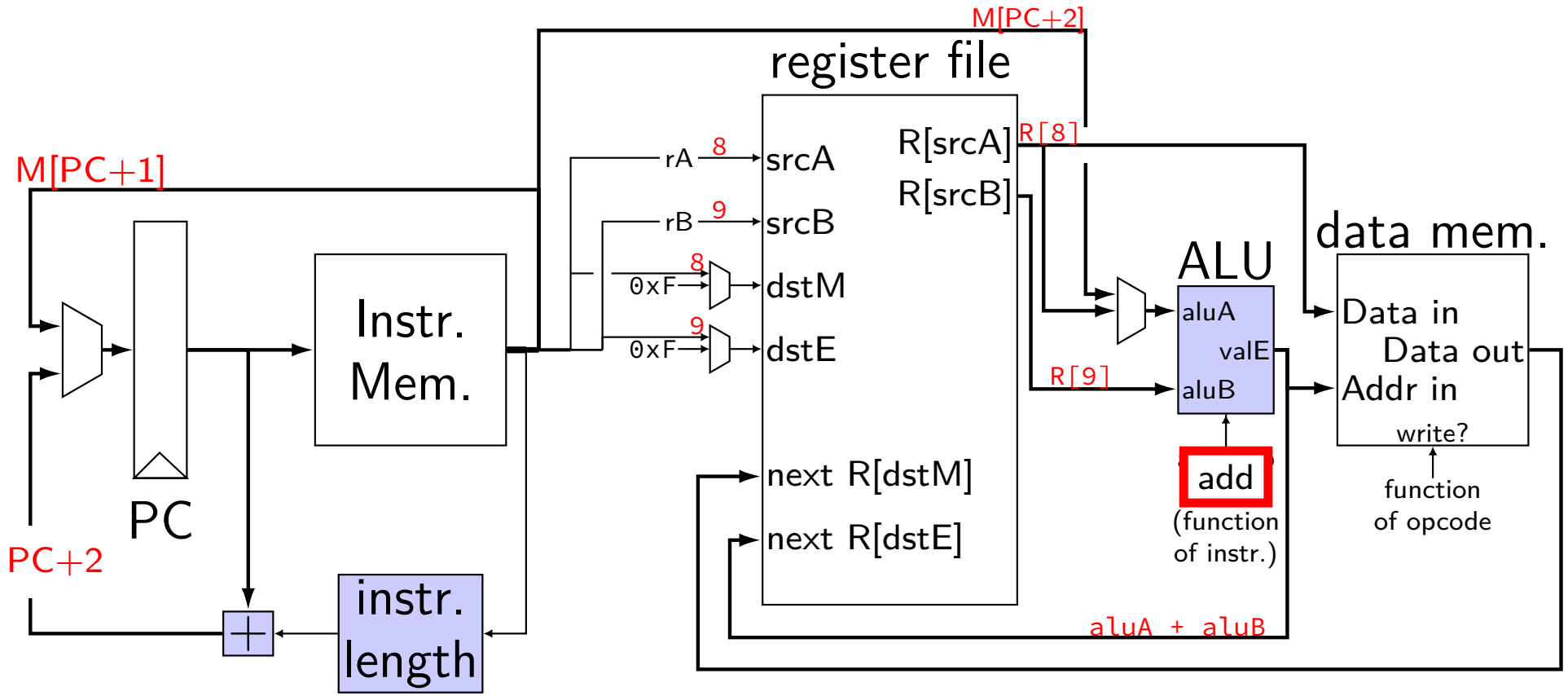
A Starting Circuit



Circuit: Setting MUXes



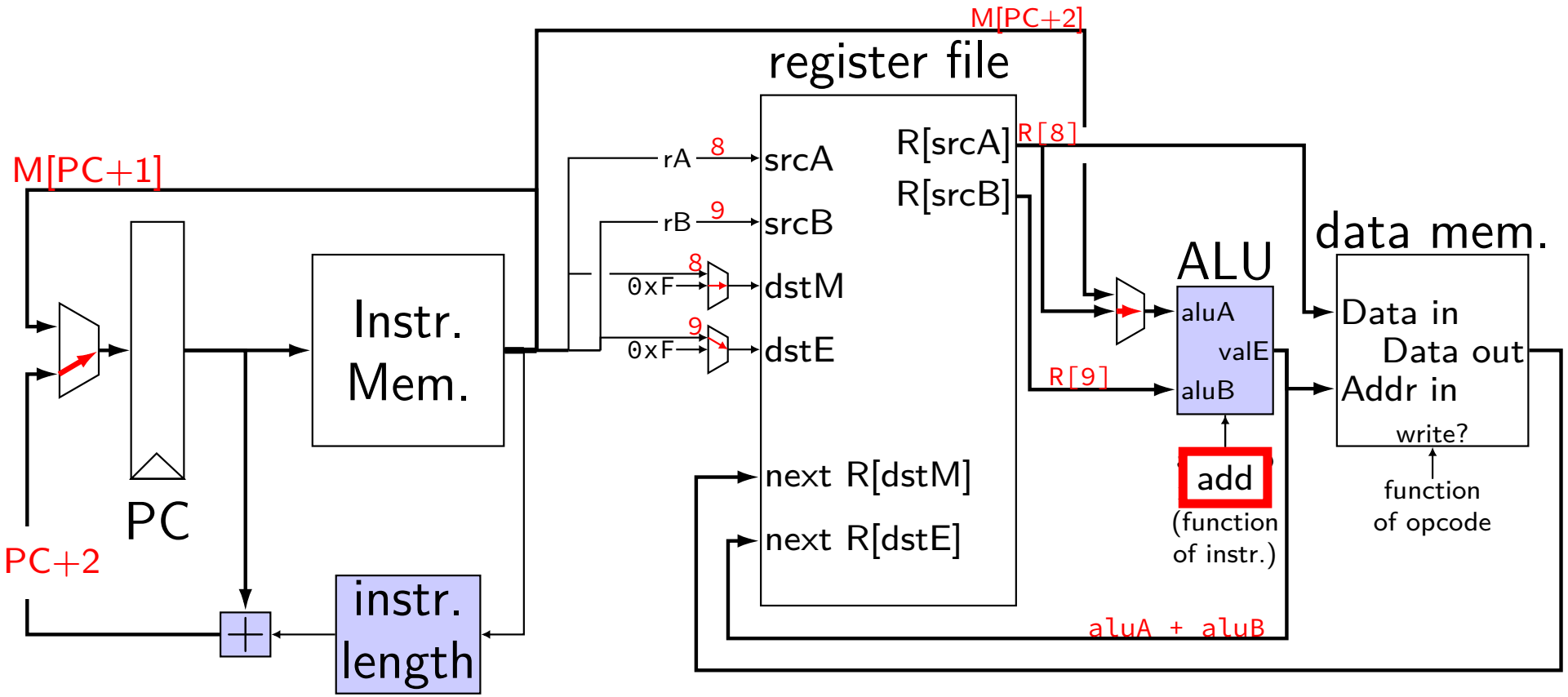
Circuit: Setting MUXes



Four MUXes — PC, dstM, dstE, aluA

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

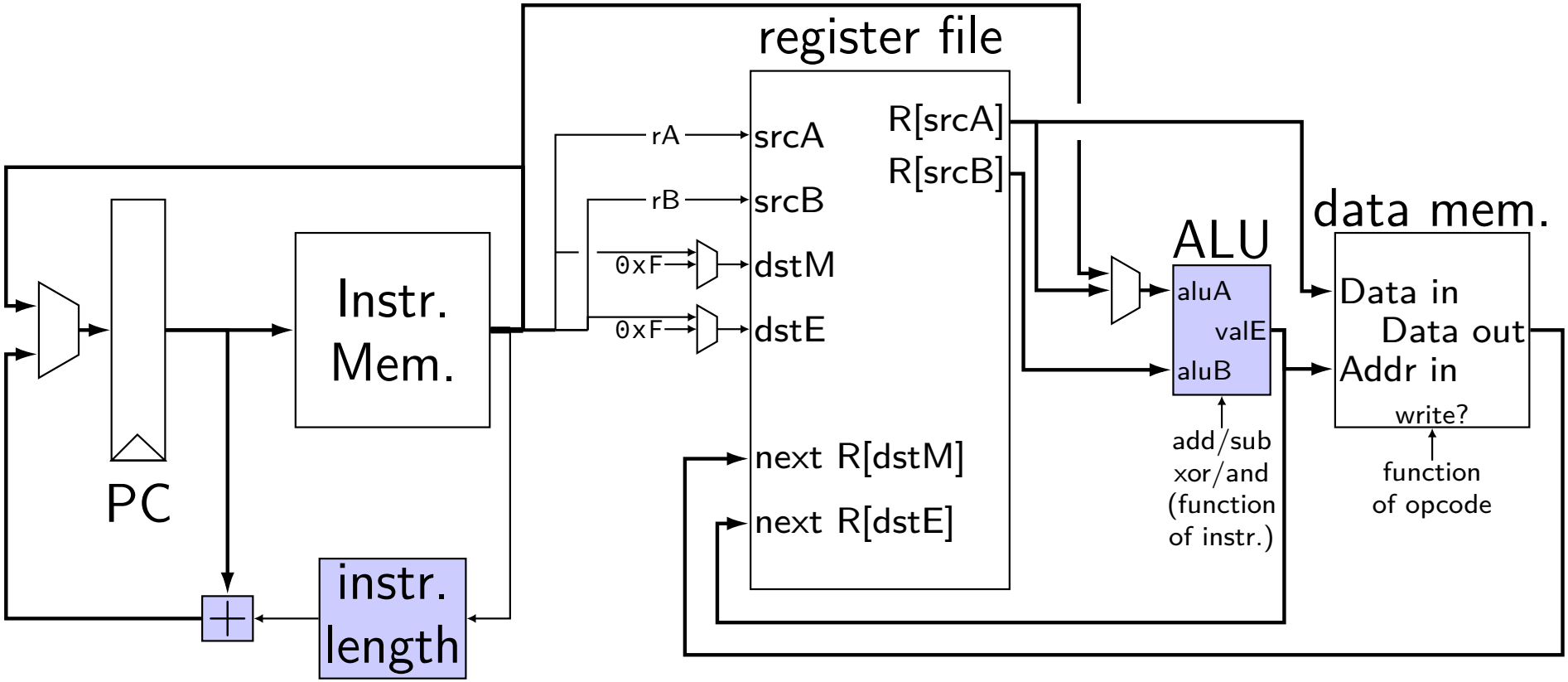
Circuit: Setting MUXes



Four MUXes — PC, dstM, dstE, aluA

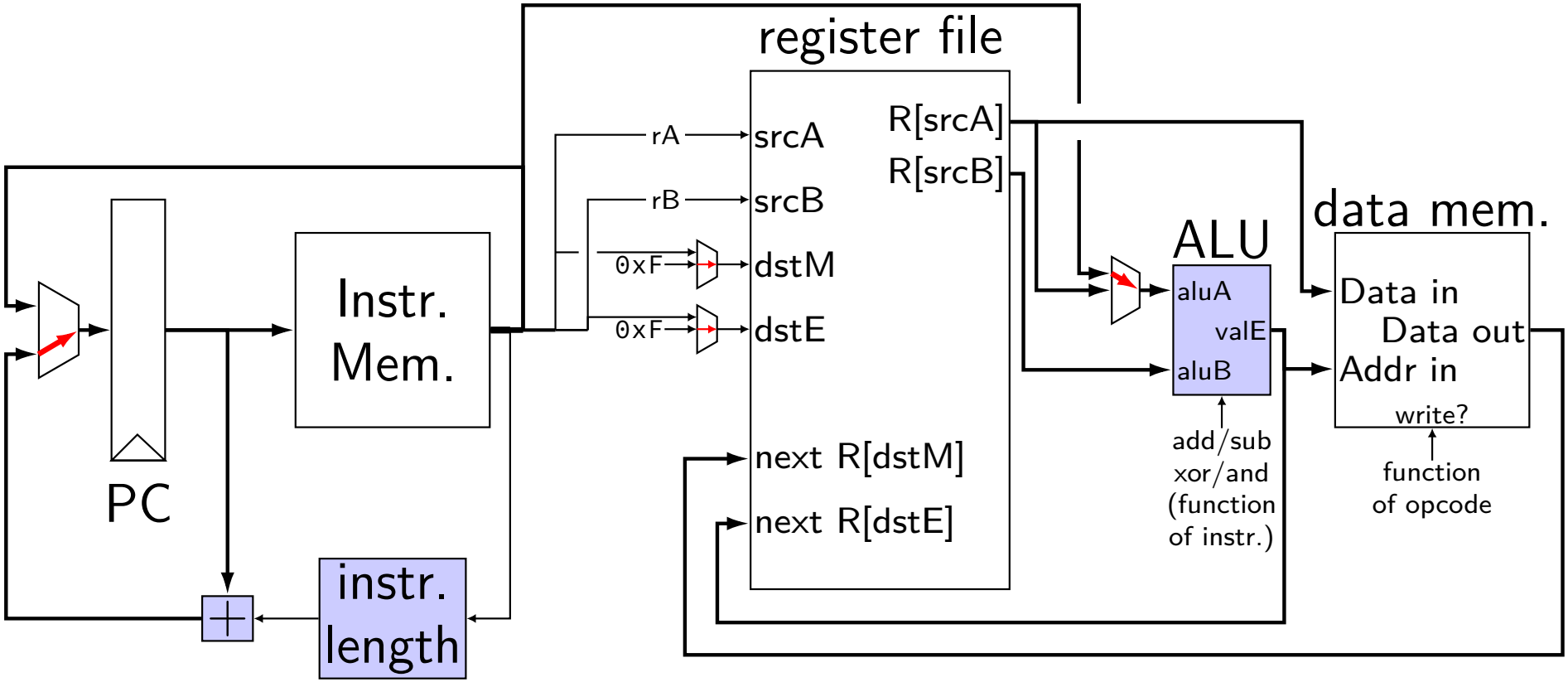
Exercise: what do they select when running `addq %r8, %r9`?

Circuit: Setting MUXes



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

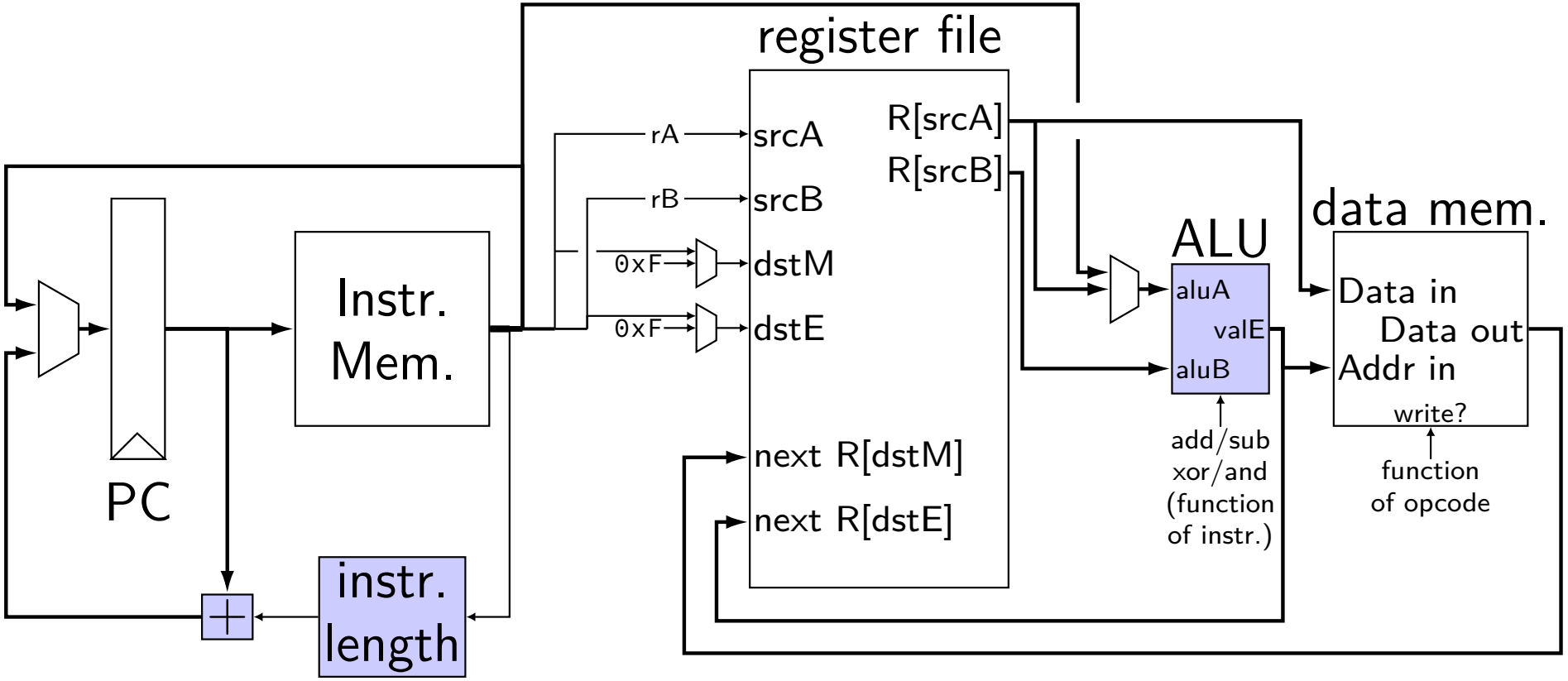
Circuit: Setting MUXes



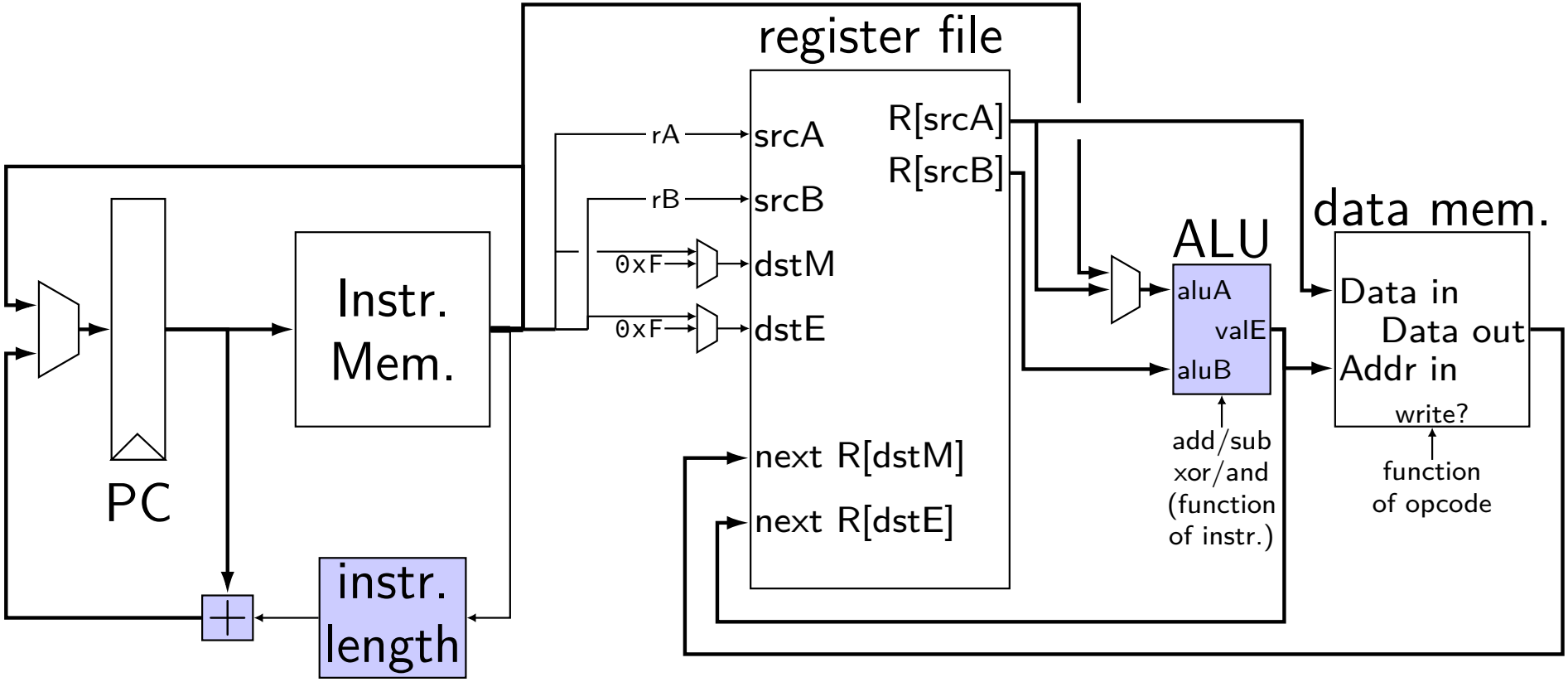
Four MUXes — PC, dstM, dstE, aluA

Exercise: what do they select for **`rmmovq`**?

Circuit: Incomplete (1)

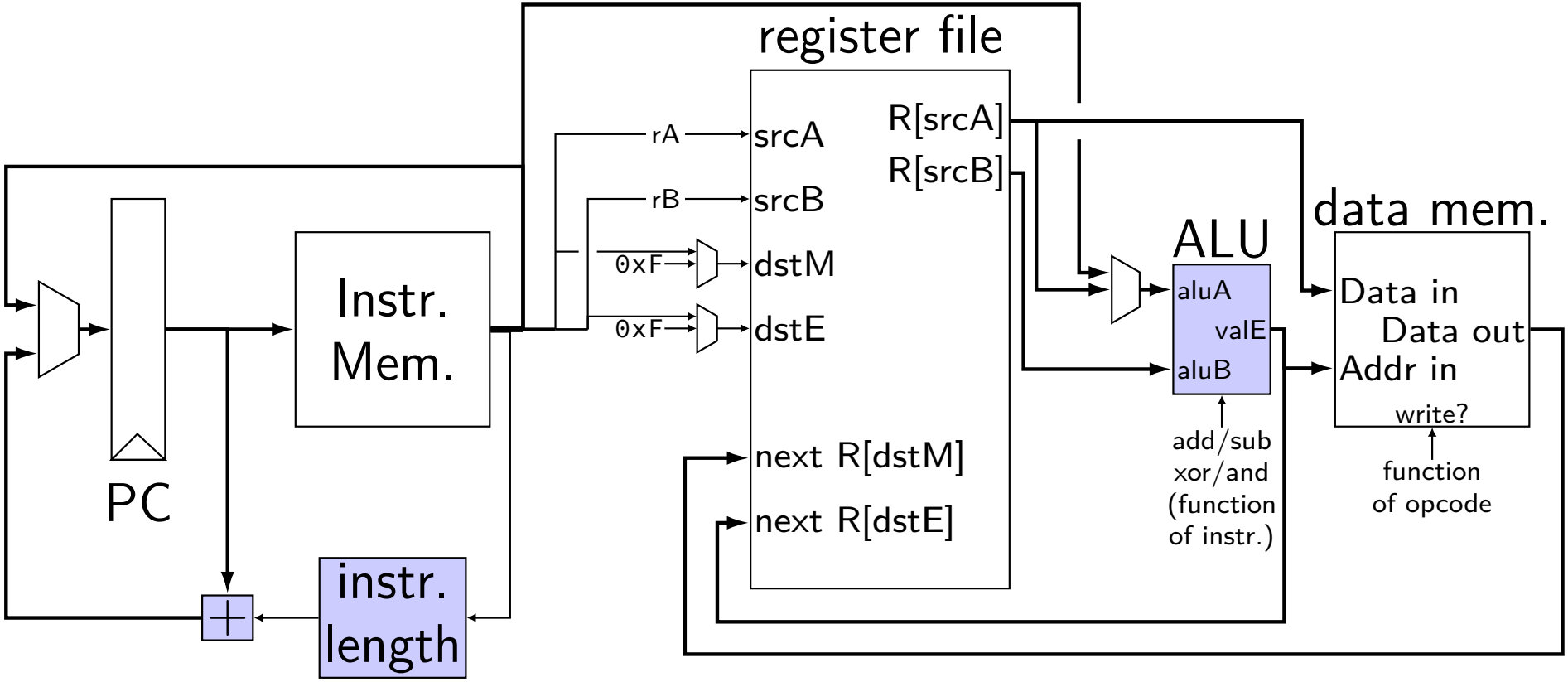


Circuit: Incomplete (1)



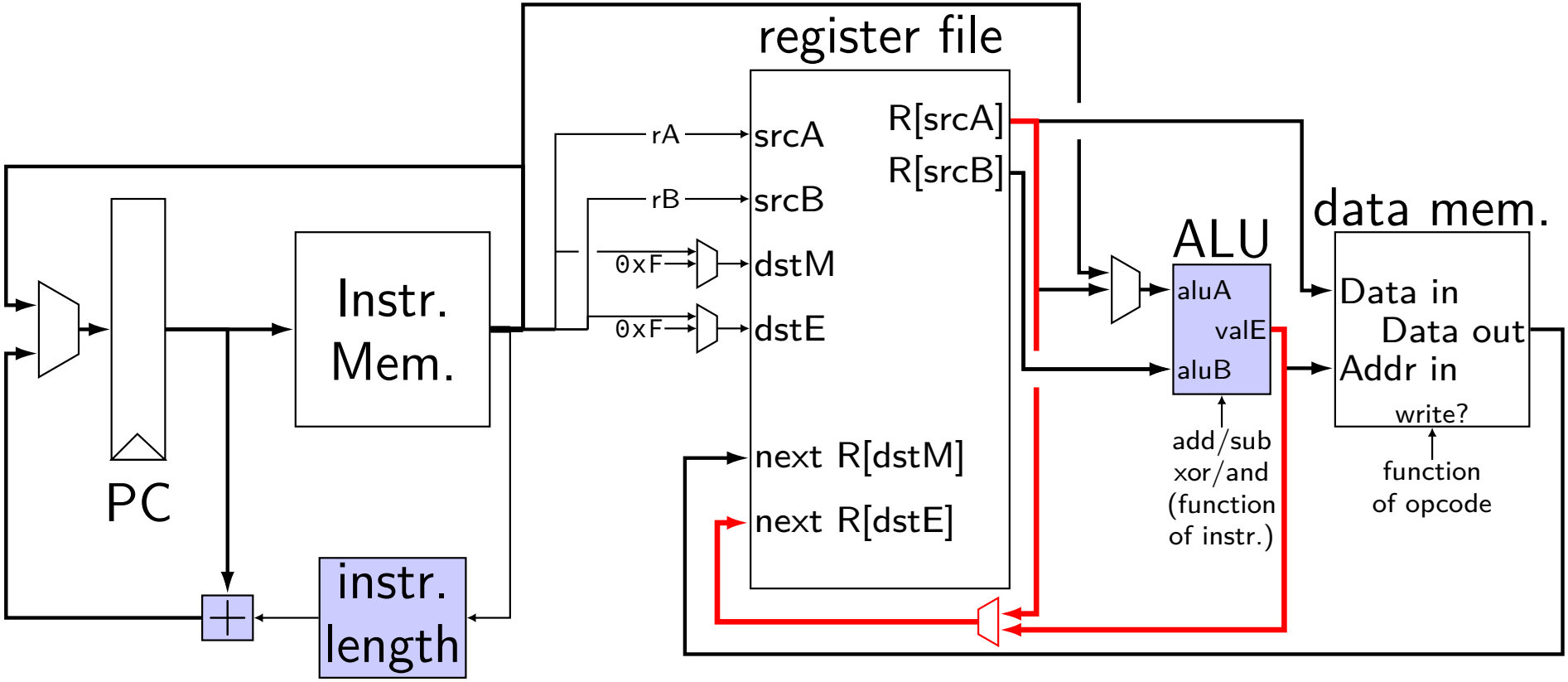
`rrmovq`, **`irmovq`**, **`mrmovq`**, **`rmmovq`**, **`jmp`**, **`call`**, **`pushq`**, **`ret`**
How many of the above instructions can the above circuit not run?
(Ignore undrawn mux selector inputs.)

Circuit: Incomplete (1)



`rrmovq` — no way to connect R[rA] to register file input
Option 1: mux on line to next R[dstE]
Option 2: mux on input to aluB

Circuit: Incomplete (1)

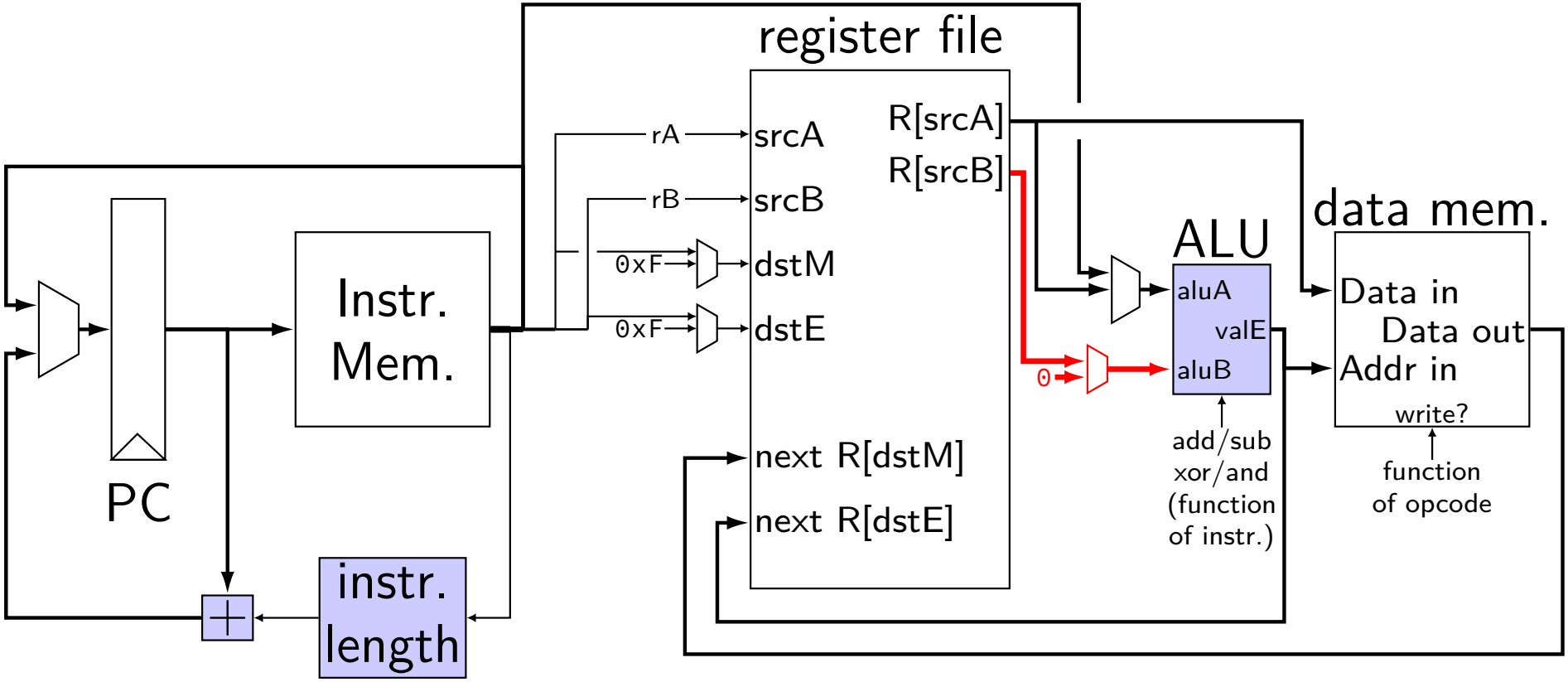


`rrmovq` — no way to connect R[rA] to register file input

Option 1: mux on line to next R[dstE]

Option 2: mux on input to aluB

Circuit: Incomplete (1)

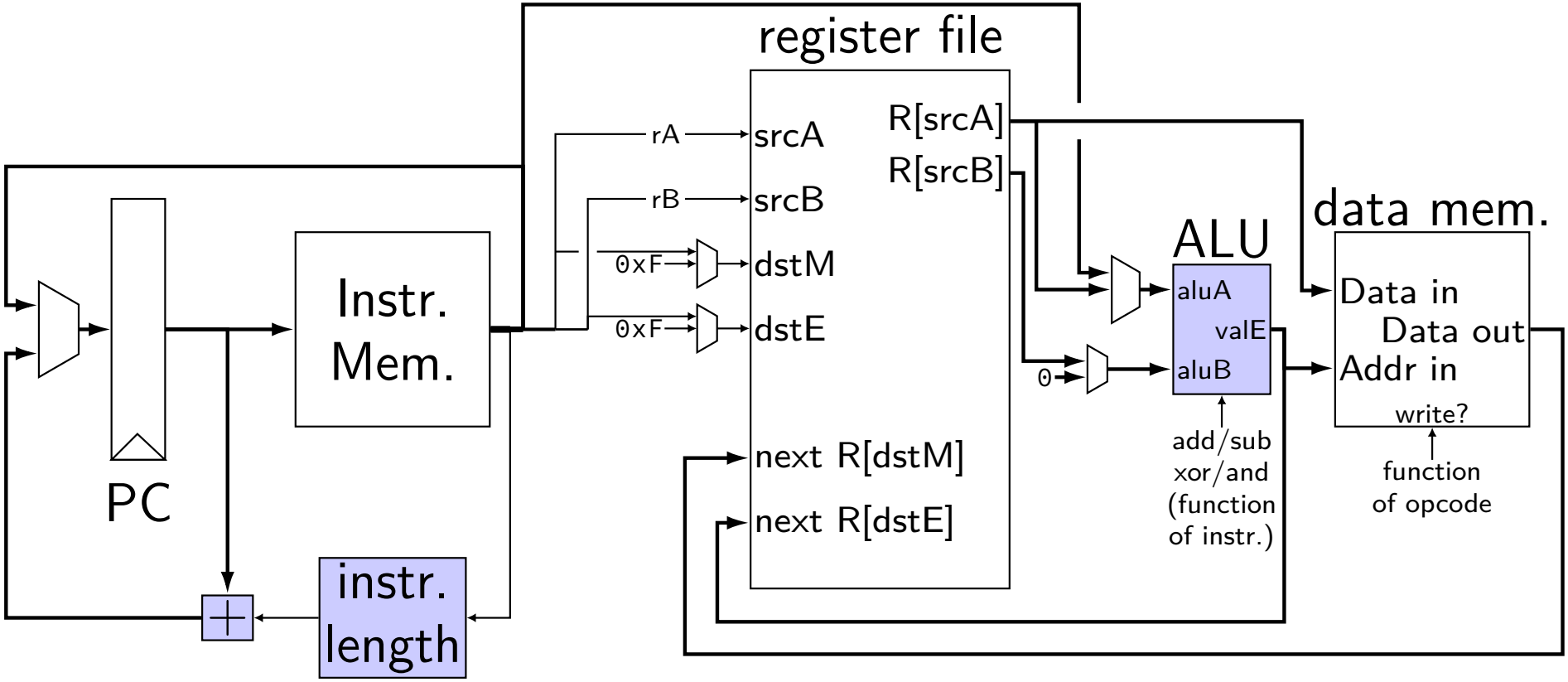


`rrmovq` — no way to connect R[rA] to register file input

Option 1: mux on line to next R[dstE]

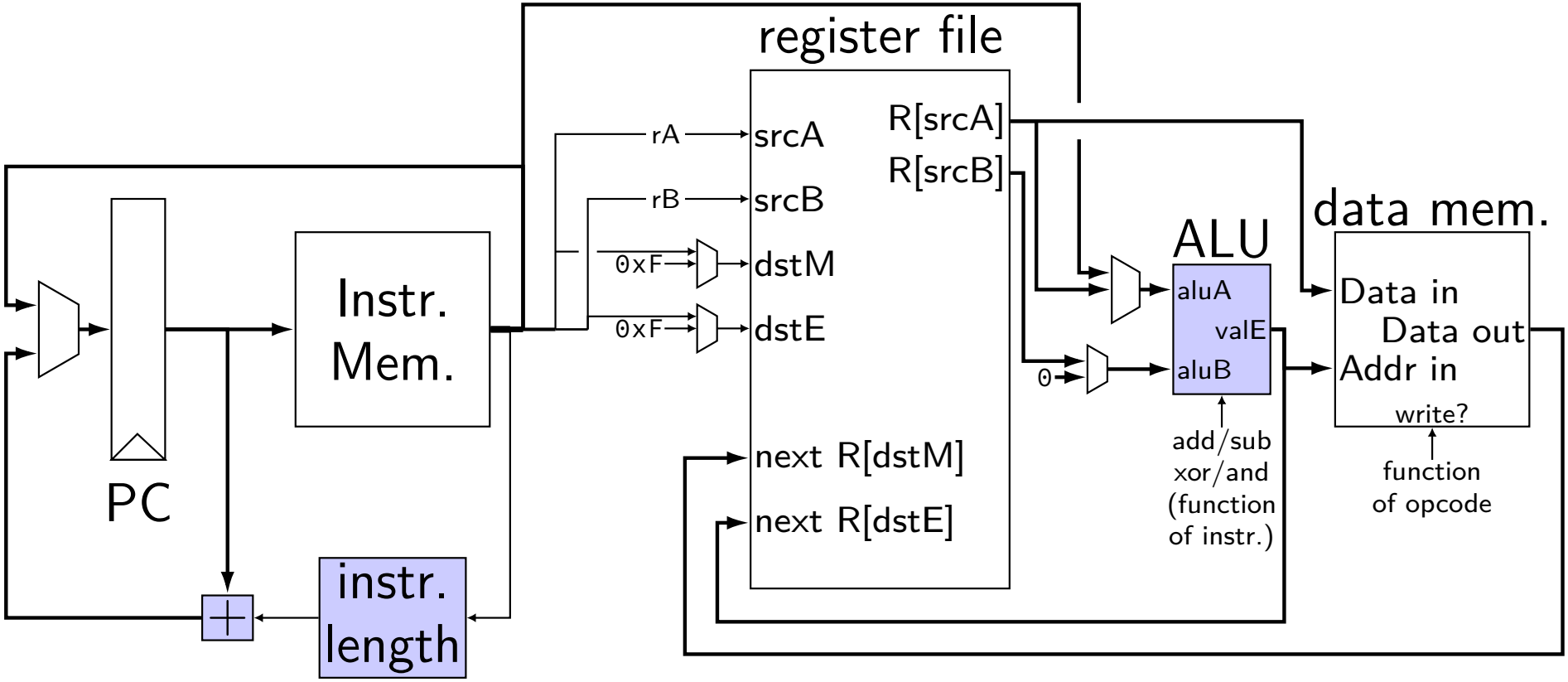
Option 2: mux on input to aluB

Circuit: Incomplete (1)



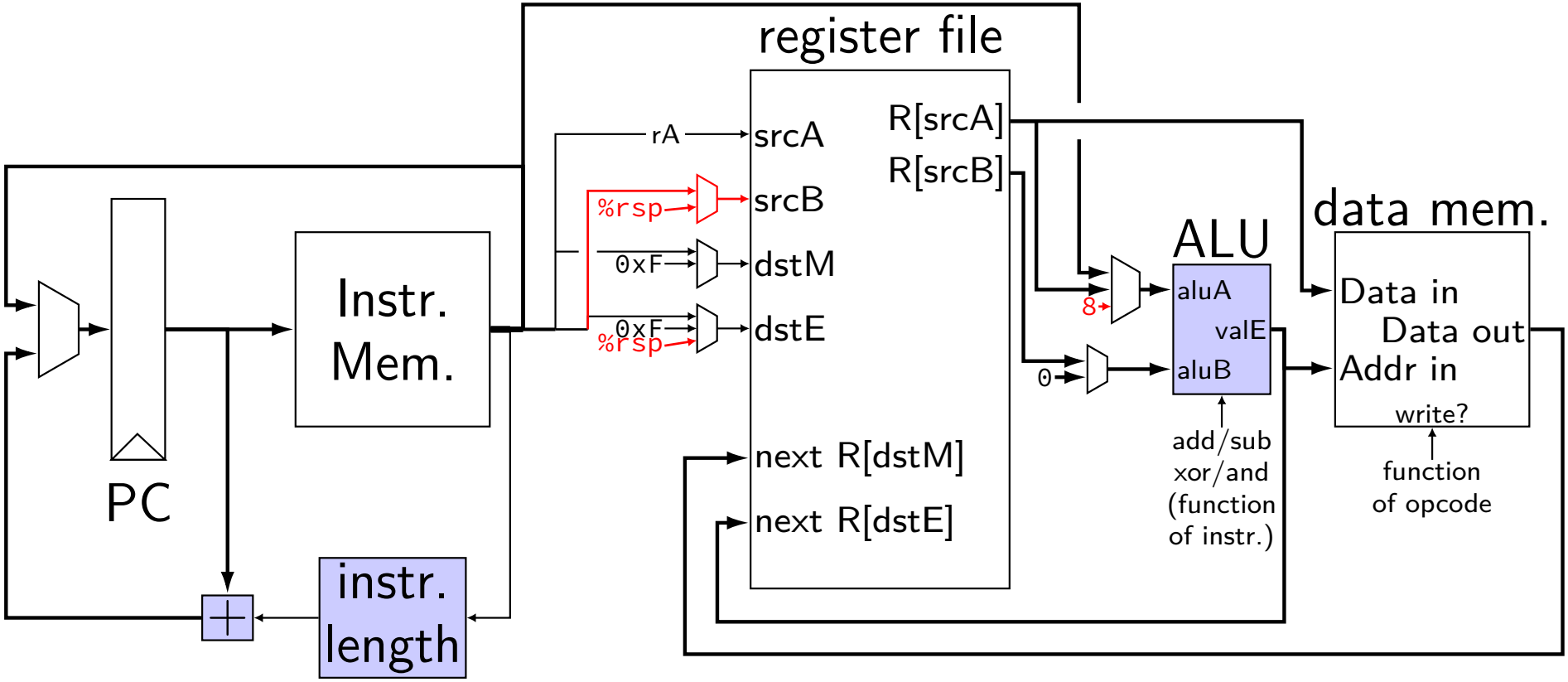
`irmovq` — originally, no way to connect V to register file input
adding mux on aluB already enough

Circuit: Incomplete (1)



pushq — no way to use rsp

Circuit: Incomplete (1)

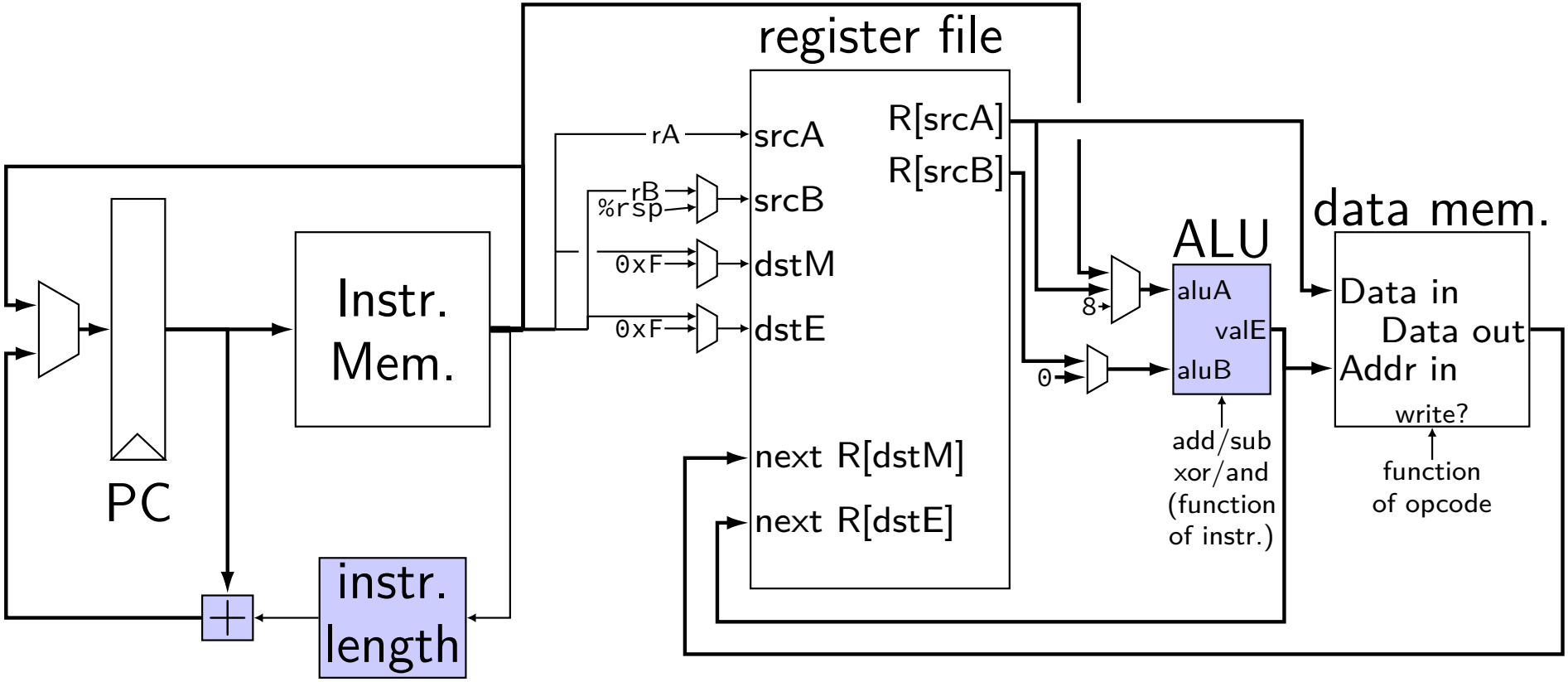


pushq — no way to use rsp

Step 1: add mux on srcB, dstE

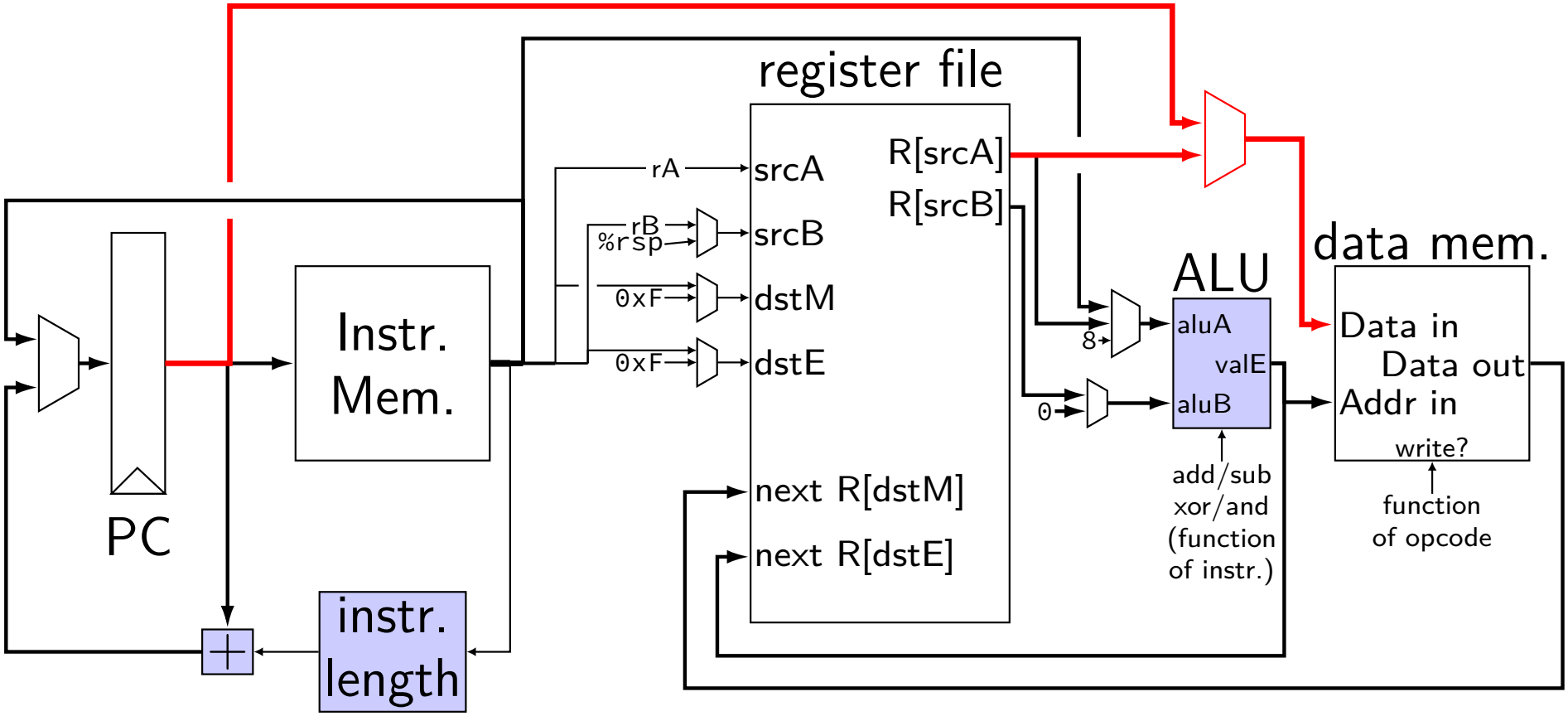
Step 2: add to aluA mux (allow constant)

Circuit: Incomplete (1)



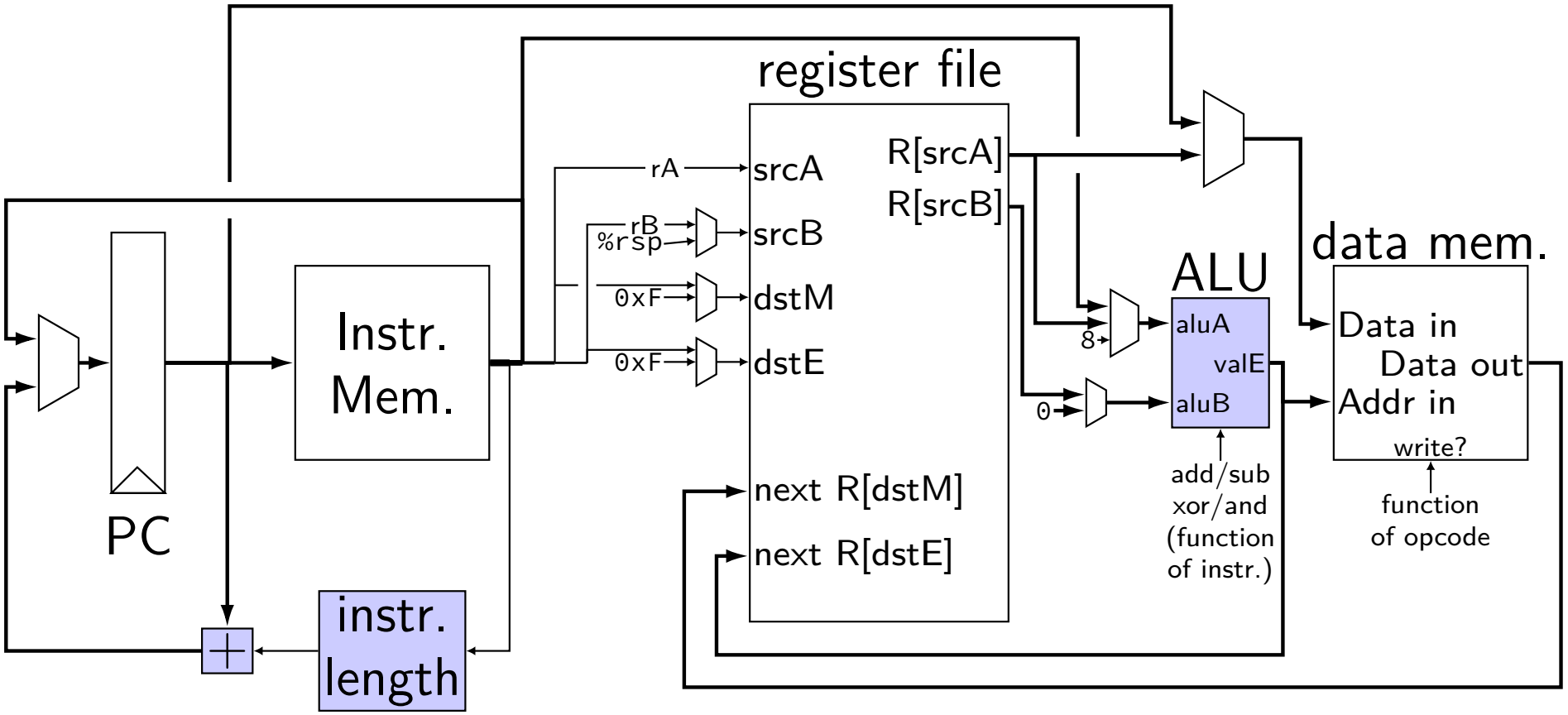
`call` —no way to connect PC to data memory

Circuit: Incomplete (1)

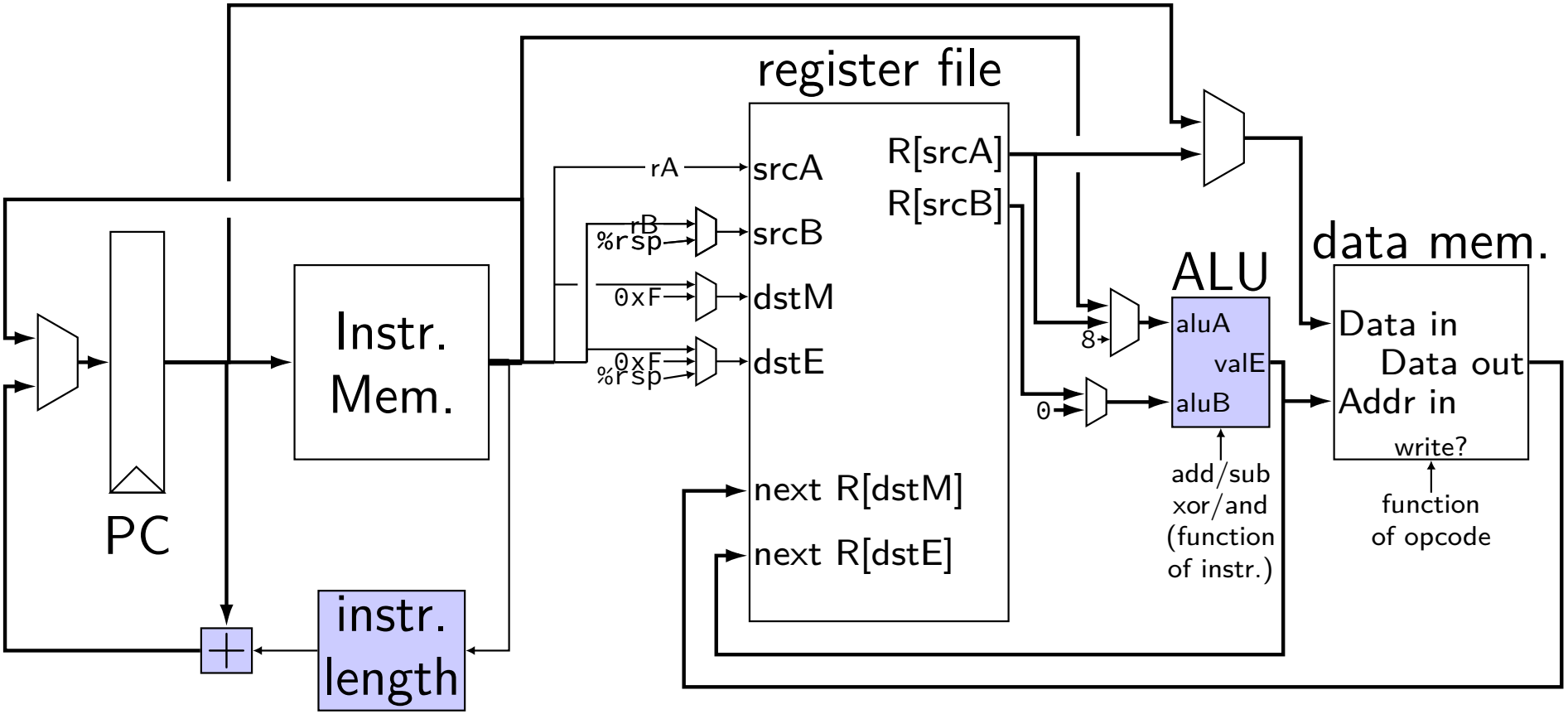


`call` —no way to connect PC to data memory
add mux on data memory data in (allow PC)

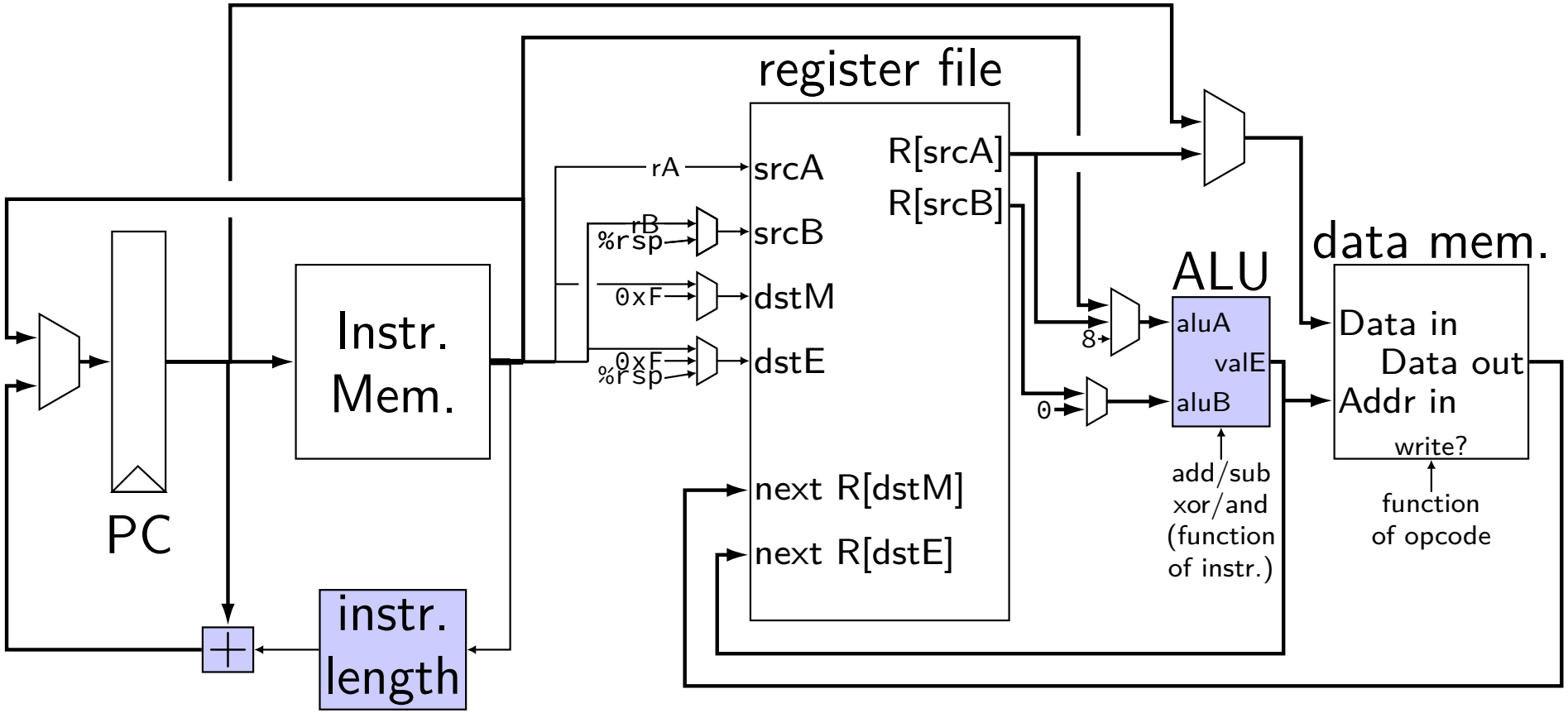
Circuit: Incomplete (1)



Circuit: More complete

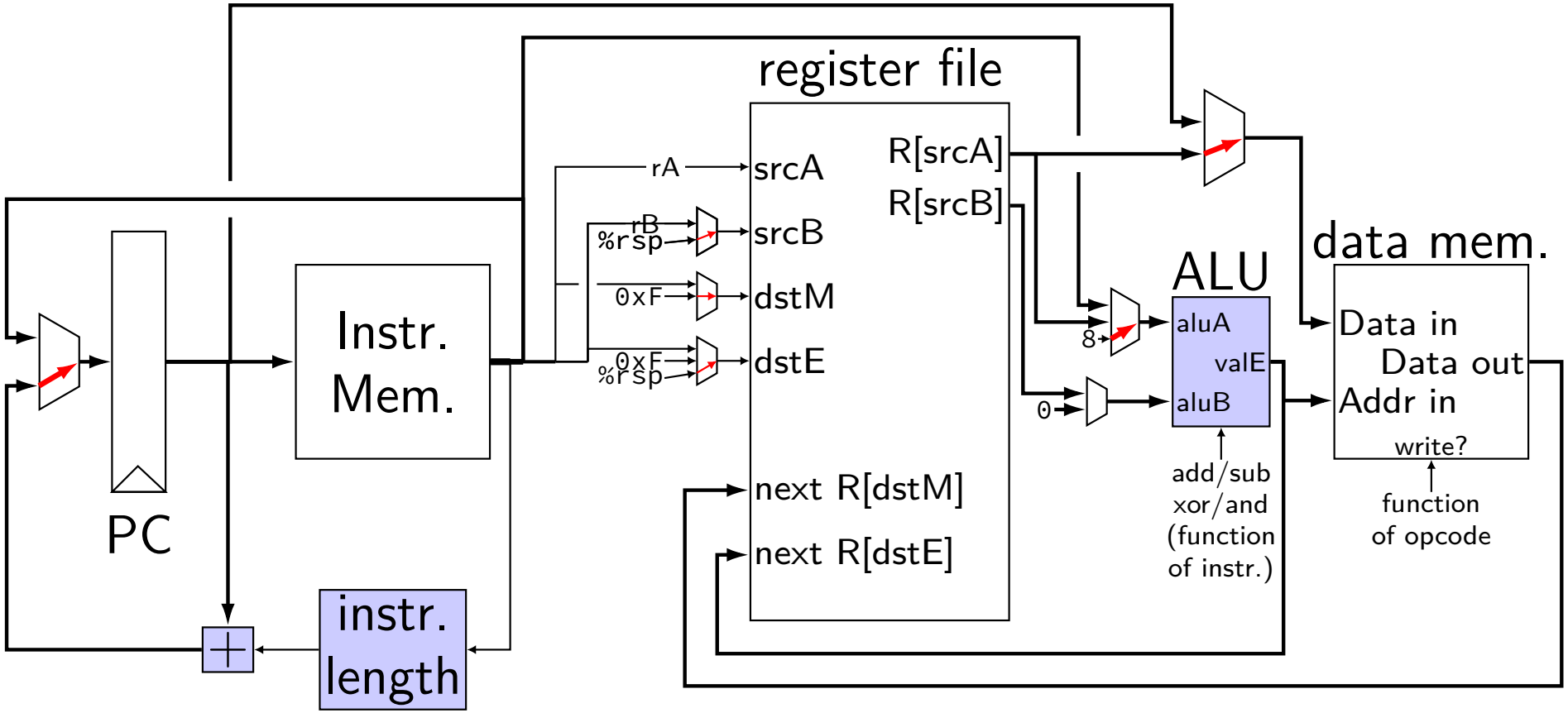


Circuit: More complete



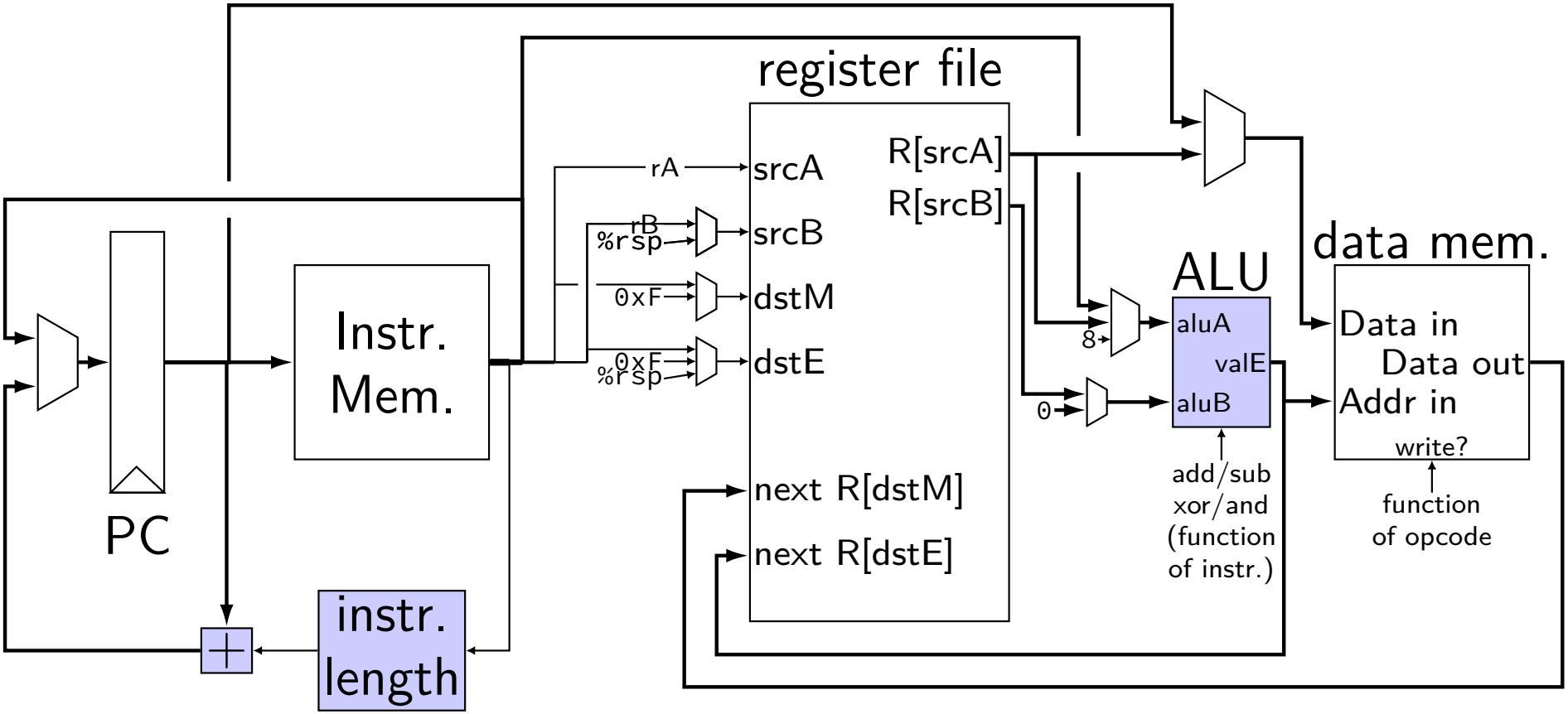
Seven MUXes — PC, srcB, dstM, dstE, aluA, aluB, data memory in
Selectors for **pushq**?

Circuit: More complete



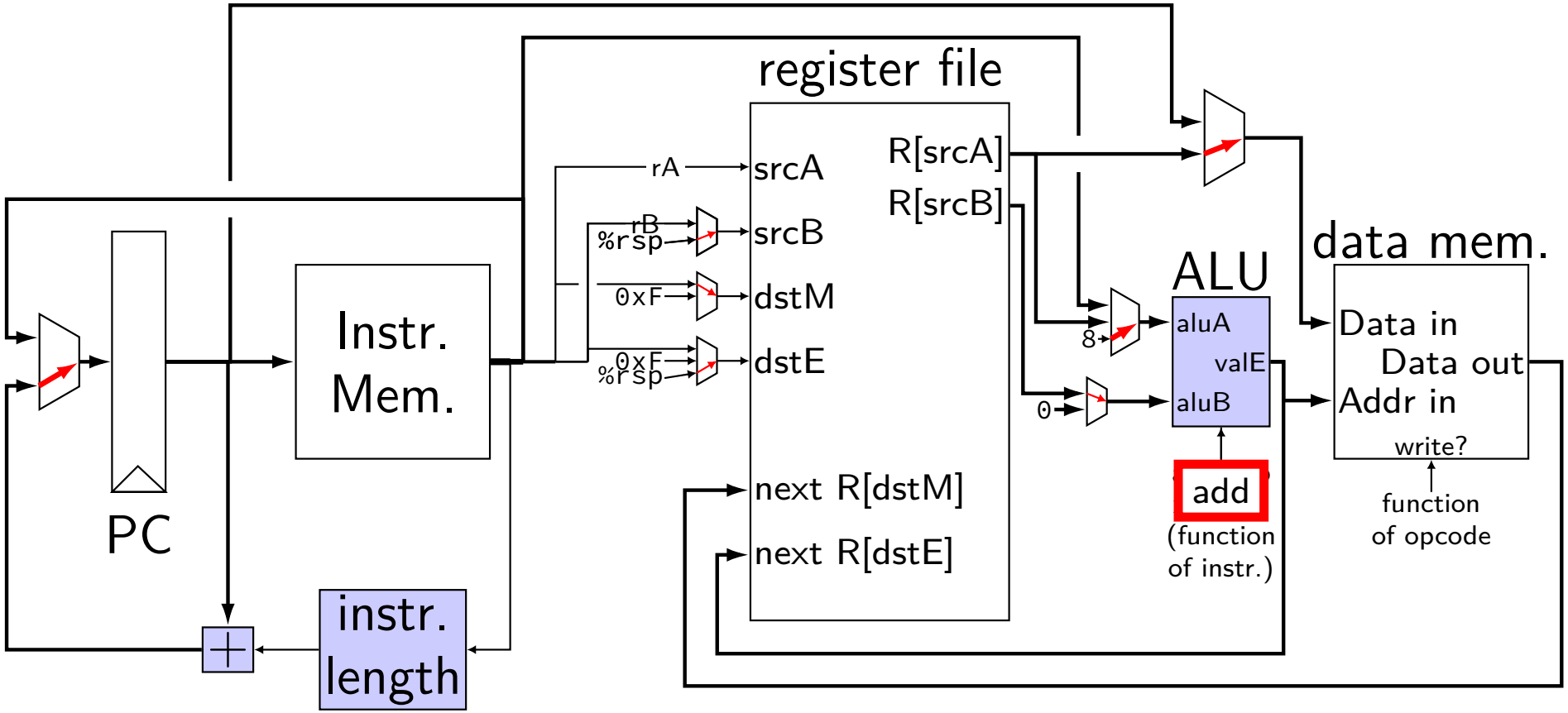
Seven MUXes — PC, srcB, dstM, dstE, aluA, aluB, data memory in
Selectors for **pushq**?

Circuit: More complete



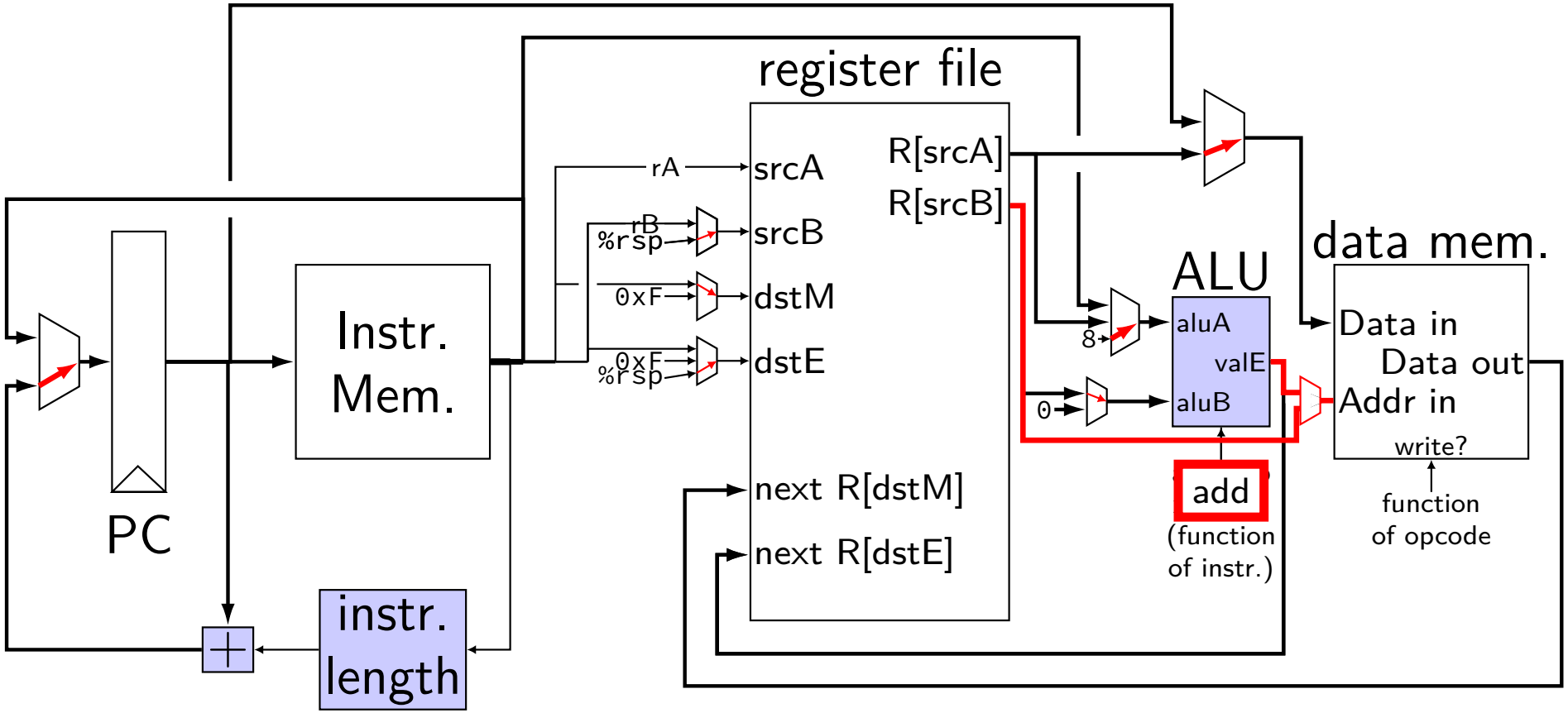
Instructions not covered: **`ret`**, **`popq`**, `cmov`*CC*, `j`*CC*
Missing condition codes for `cmov`*CC*, `j`*CC*.

Circuit: More complete



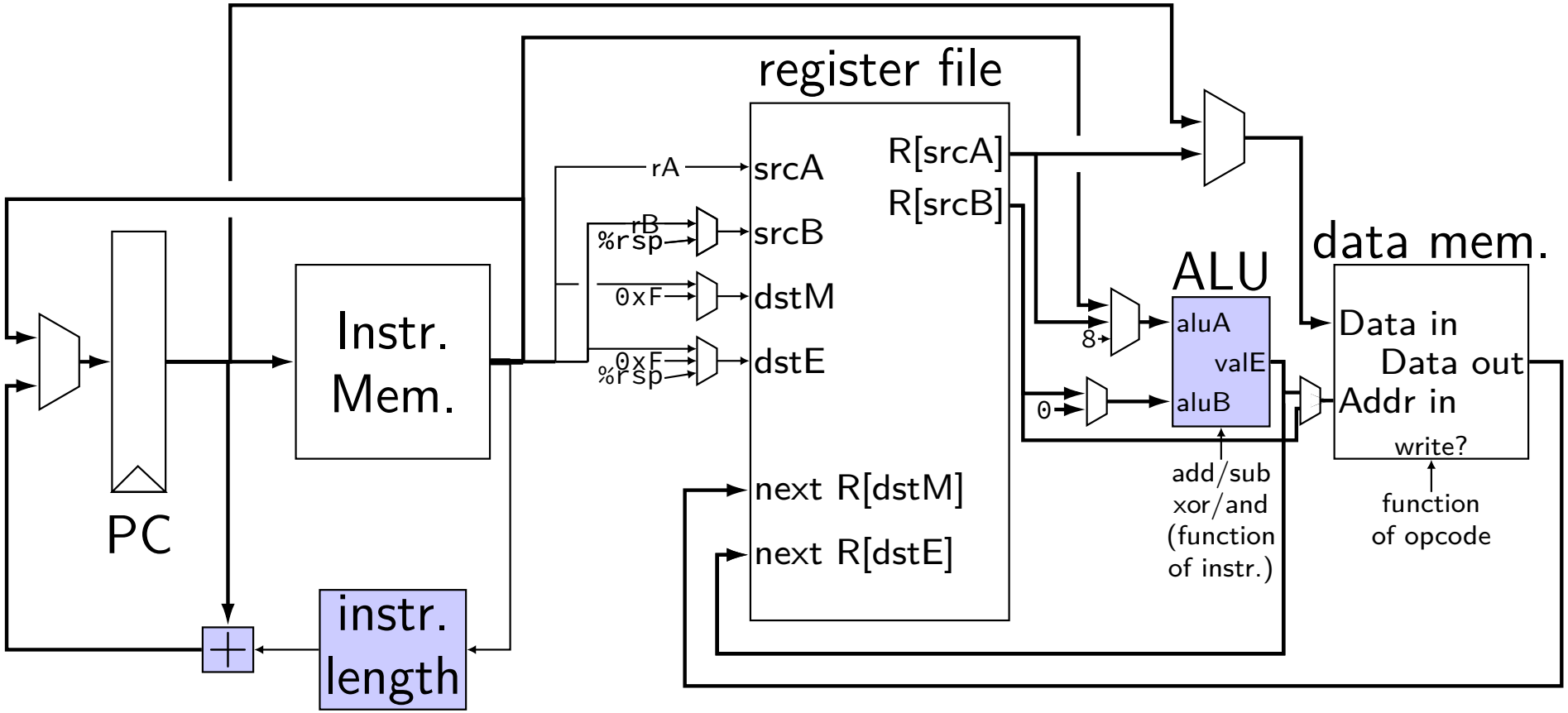
What's wrong with this implementation of **popq**?
How could it be fixed?

Circuit: More complete



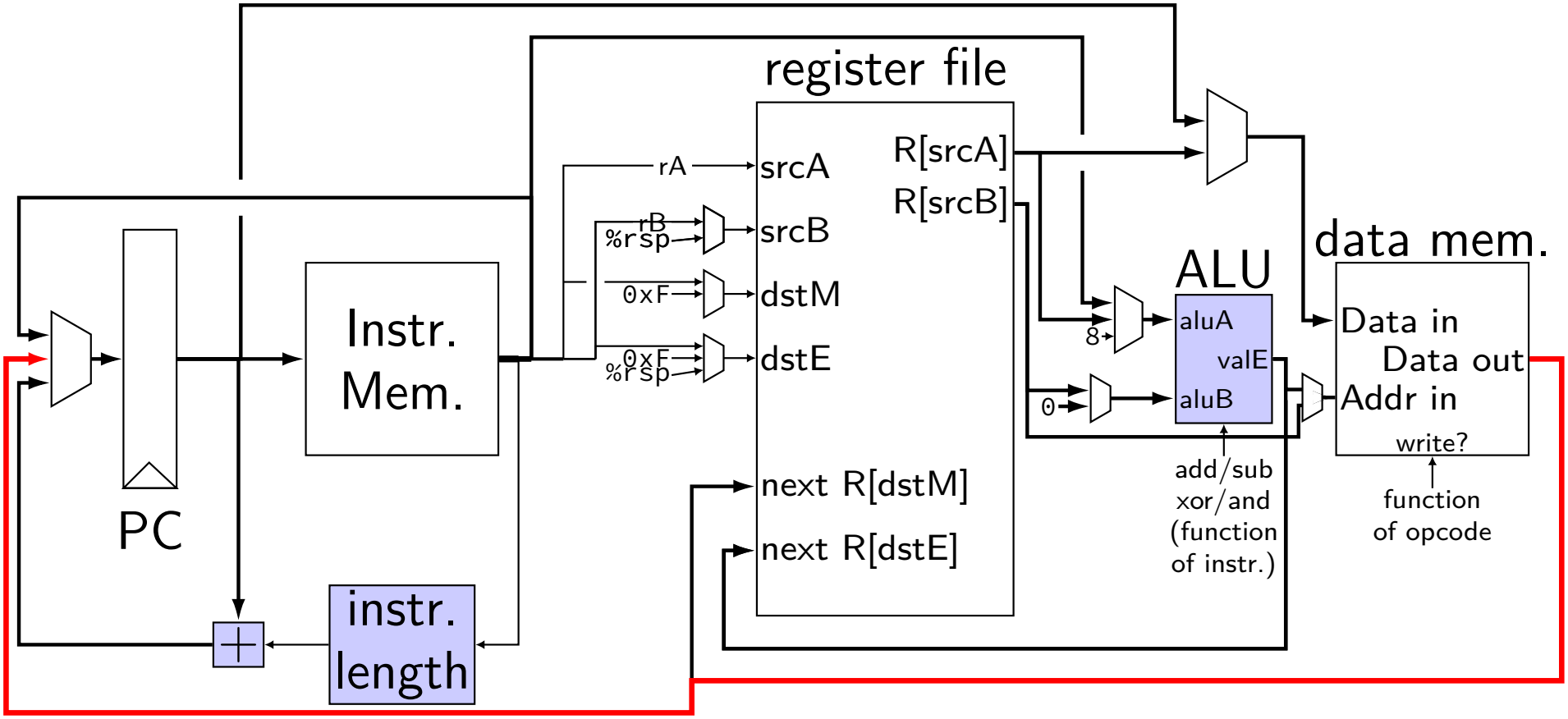
What's wrong with this implementation of **popq**?
How could it be fixed?

Circuit: More complete



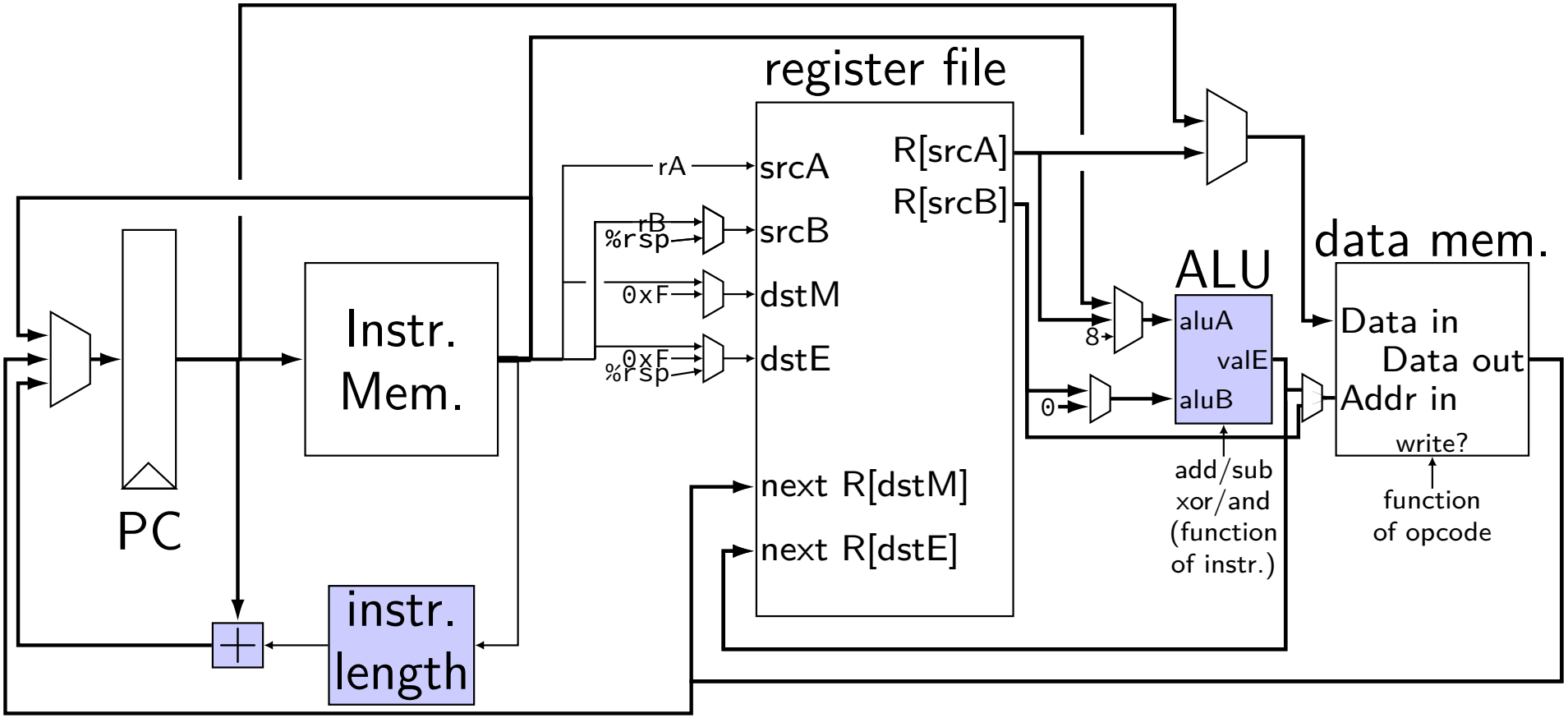
What do we need to add for **`ret`**?

Circuit: More complete

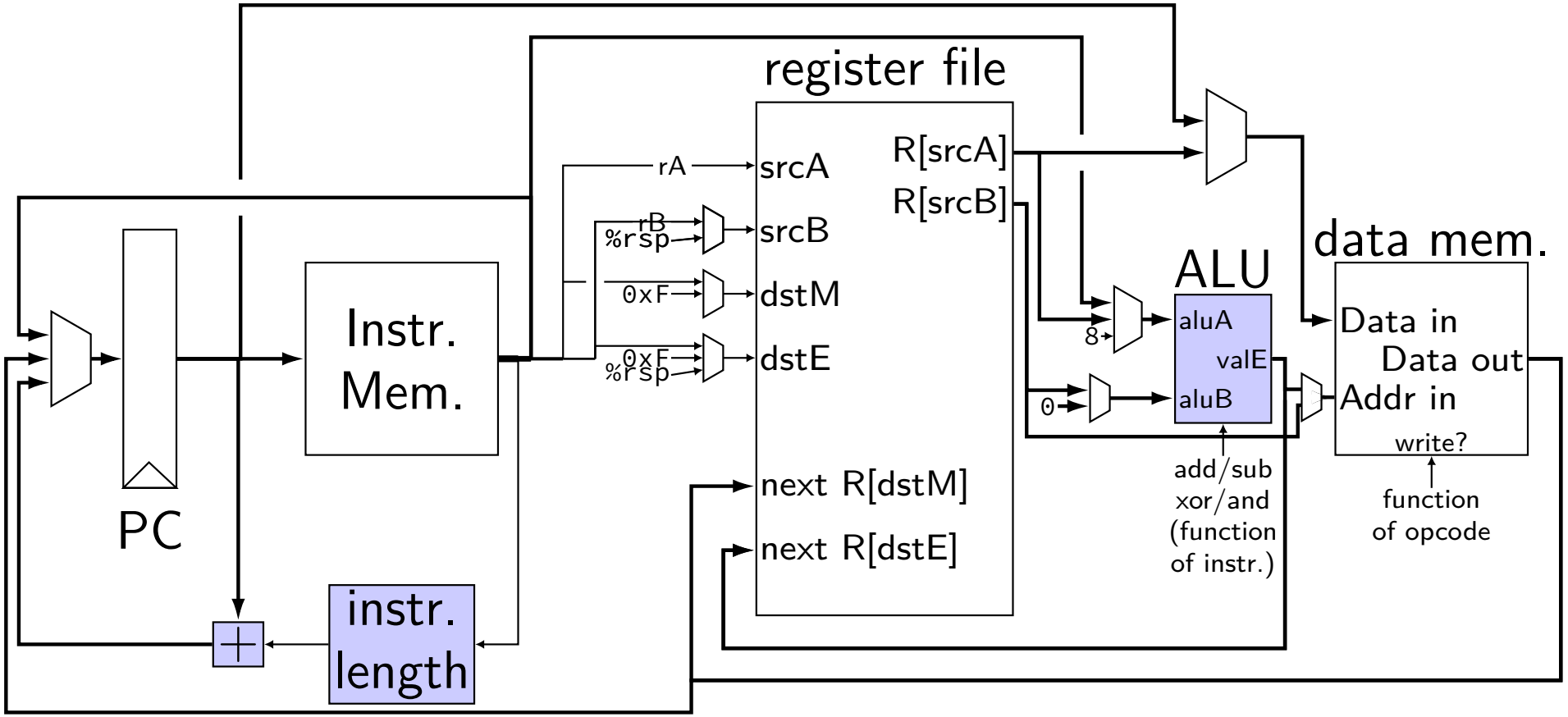


What do we need to add for **`ret`**?

Circuit: More complete



Circuit


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

Stages: pushq/popq

stage	pushq	popq
fetch	icode : ifun $\leftarrow M_1[\text{PC}]$ rA : rB $\leftarrow M_1[\text{PC}+1]$ valP $\leftarrow$ PC $+ 2$	icode : ifun $\leftarrow M_1[\text{PC}]$ rA : rB $\leftarrow M_1[\text{PC}+1]$ valP $\leftarrow$ PC $+ 2$
decode	valA $\leftarrow R[\text{rA}]$ valB $\leftarrow R[\%rsp]$	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$
execute	valE $\leftarrow$ valB $+ (-8)$	valE $\leftarrow$ valB $+ 8$
memory	$M_8[\text{valE}] \leftarrow$ valA	valM $\leftarrow M_8[$ valA $]$
write back	$R[\%rsp] \leftarrow$ valE	$R[\%rsp] \leftarrow$ valE $R[\text{rA}] \leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Stages: pushq/popq

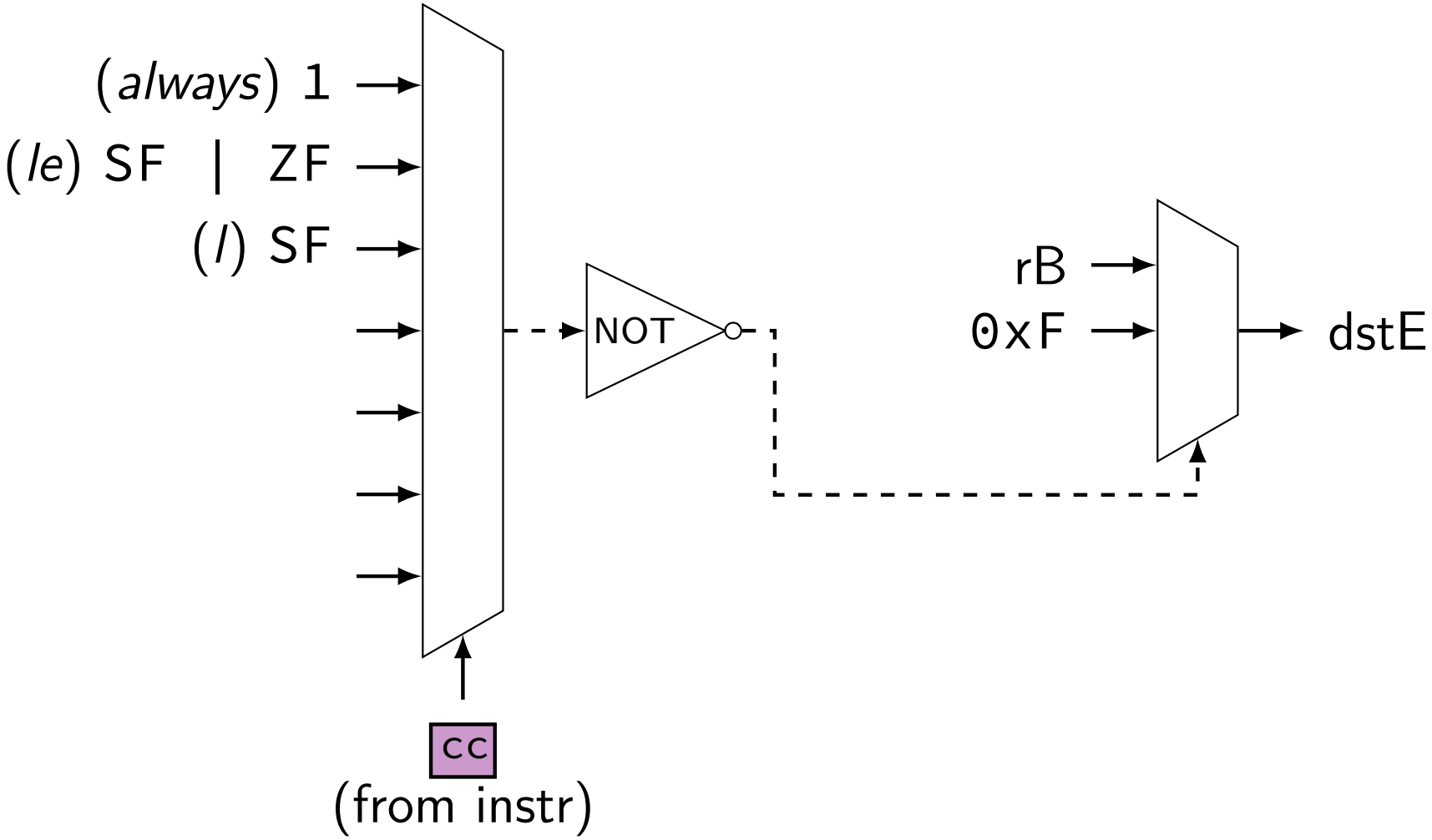
stage	pushq	popq
fetch	icode : ifun $\leftarrow M_1[\text{PC}]$ rA : rB $\leftarrow M_1[\text{PC}+1]$ valP $\leftarrow \text{PC}+2$	icode : ifun $\leftarrow M_1[\text{PC}]$ rA : rB $\leftarrow M_1[\text{PC}+1]$ valP $\leftarrow \text{PC}+2$
decode	valA $\leftarrow R[\text{rA}]$ valB $\leftarrow R[\%rsp]$	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$
execute	valE $\leftarrow \text{valB}+(-8)$	valE $\leftarrow \text{valB}+8$
memory	$M_8[\text{valE}] \leftarrow \text{valA}$	valM $\leftarrow M_8[\text{valA}]$
write back	$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Conditional movs

```
absoluteValueJumps:
      andq %rdi, %rdi
      jge same            ; if rdi >= 0, goto same
      irmovq $0, %rax     ; rax <- 0
      subq %rdi, %rax     ; rax <- rax (0) - rdi
      ret
same: rrmovq %rdi, %rax
      ret

absoluteValueCMov:
      irmovq  $0, %rax
      subq %rdi, %rax     ; rax <- -rdi
      andq %rdi, %rdi
      cmovge %rdi, %rax   ; if (rdi > 0) rax <- rd
      ret
```

Using condition codes: cmov



Systematic construction

MUX	OPq	ret	callq	pushq
next PC	PC + len	memory out	from instr	PC + len
srcB	rB	—	—	%rsp

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