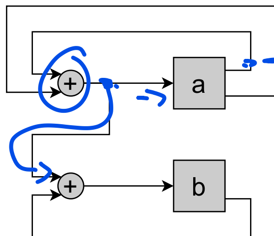
# SEQ Continued

February 21, 2023

# quiz Q1

```
register xY {
     a : 64 = 2;
     b : 64 = ???;
}

x_a = Y_a + Y_a;
x_b = Y_b + x_a;
```




| clk | x_a | x_b | Y_a | Y_b |
|---|---|---|---|---|
| 0 | 4 | 4+w | 2 | w |
| 1 | 8 | 8+4+w | 4 | 4+w |
| 2 | | | 8 | 8 + 4+w = 1024<br>w = 1024-12=1012 |

# quiz Q2

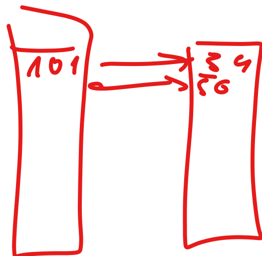
PC set to 0x101

i10bytes[0..80] = 0x33221100DEBC9A785634

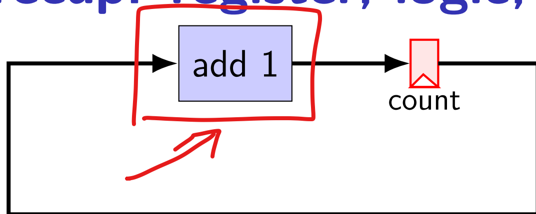
i10bytes[0..8] = 0x34

i10bytes[8..12] = 0x6

**i10bytes[12..20] = 0x85**



# recap: register, logic, counter circuit

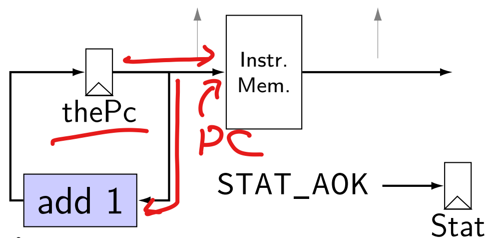



```
register xY {
        count : 3 = 0b000 ;
}
 x_count = Y_count + 0b001;
```

| time | Y_count | x_count |
|---|---|---|
| start | 000 | 001 |
| start + 1 rising edge | 001 | 010 |
| start + 2 rising edges | 010 | 011 |
| start + 3 rising edges | 011 | 100 |
| … | … | … |

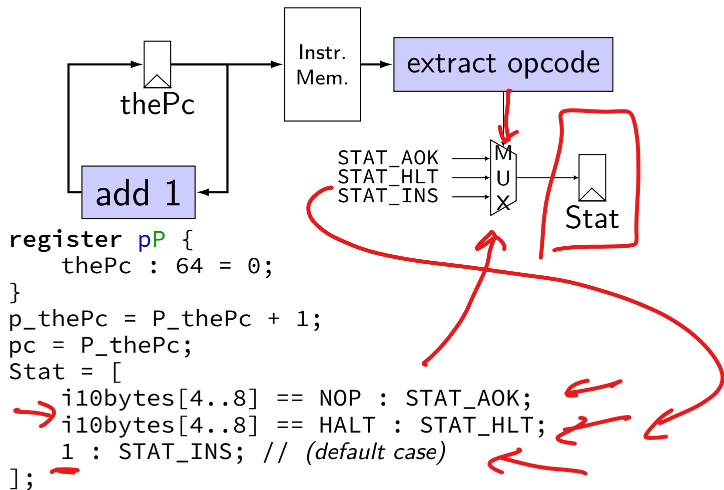
*program counter, simplest CPU*

# recap: Instr. Mem., pc, nop CPU, Stat




```
register pF {
    thePc : 64 = 0;
}
p_thePc = F_thePc + 1;
pc = F_thePc;
Stat = STAT_AOK;
```

# recap: MUX, opcode, nop/halt CPU




```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; // (default case)
];
```

*nop/jmp CPU, register file, addq CPU, build a processor*

# simple ISA: addq

`addq %rXX, %rYY`

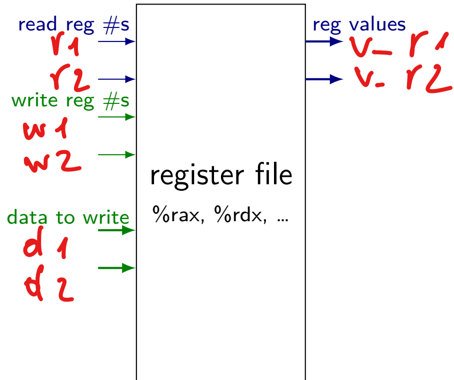
encoding: `6` `0` `%rXX` `%rYY` (two 4-bit register #s)

- 2 byte instructions, no opcode

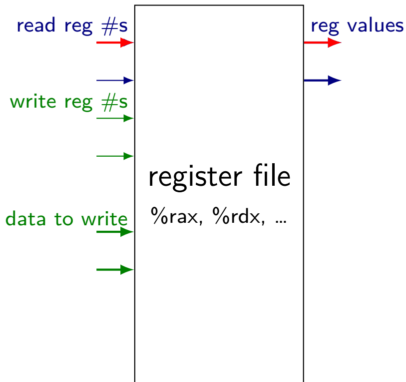
for now: no other instructions

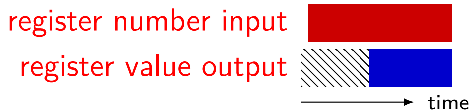
- later: adding support for nop+halt

# register file

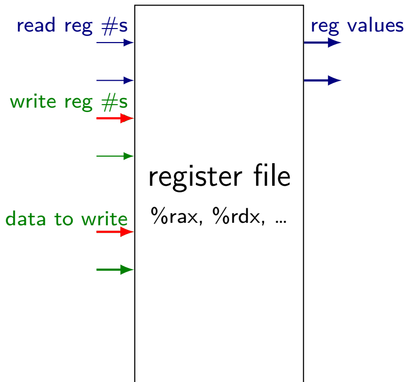


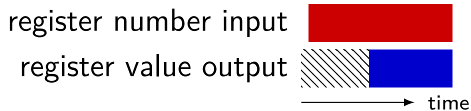
# register file

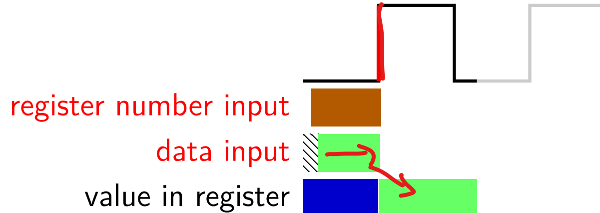


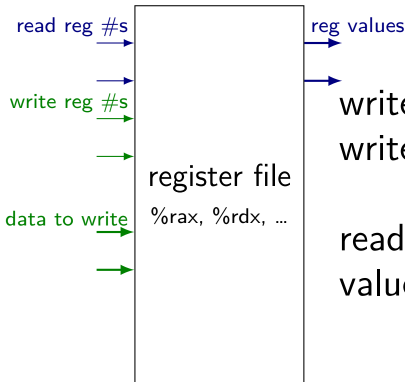
# register file



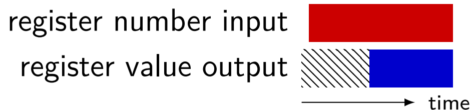



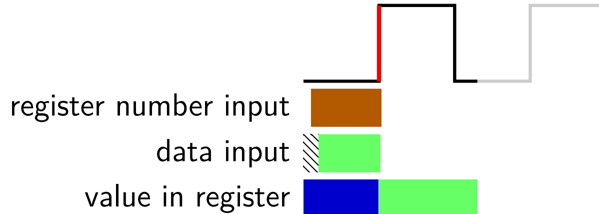
# register file



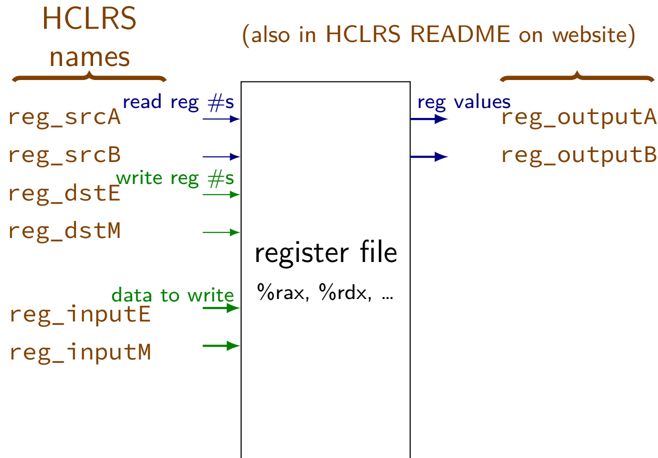

write register #15 (REG_NONE):
write is ignored

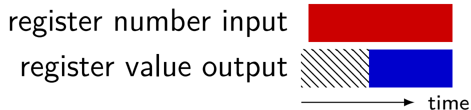
read register #15 (REG_NONE):
value is always 0

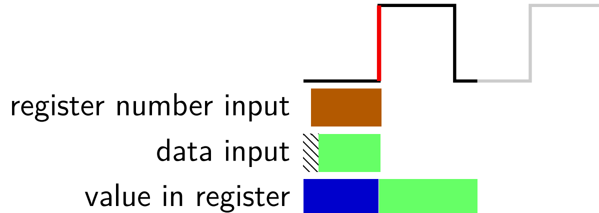


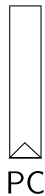
# register file

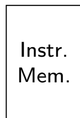


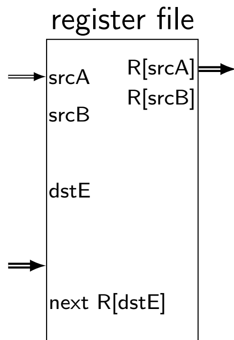



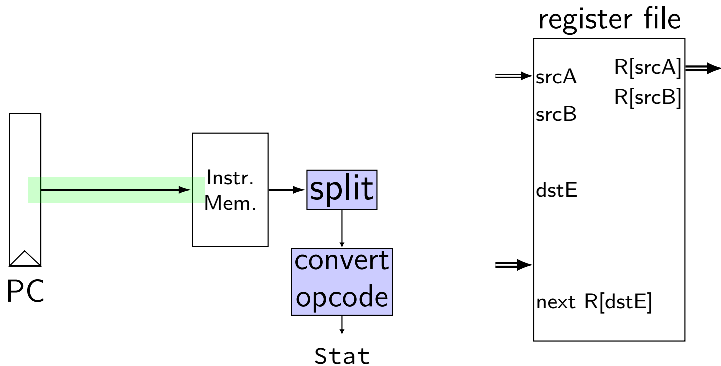
# addq CPU



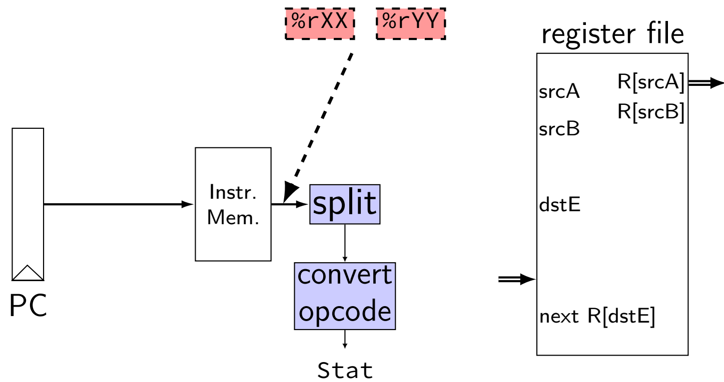



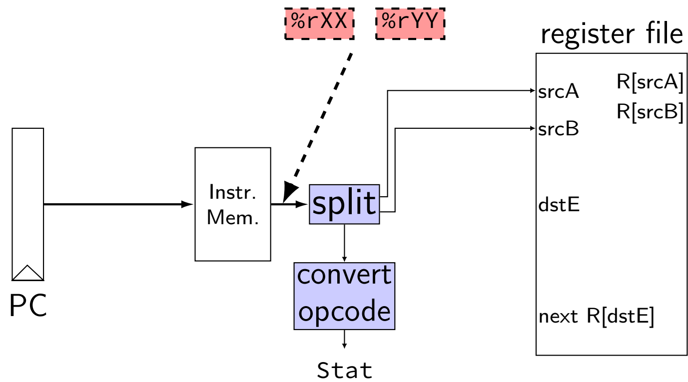
# addq CPU



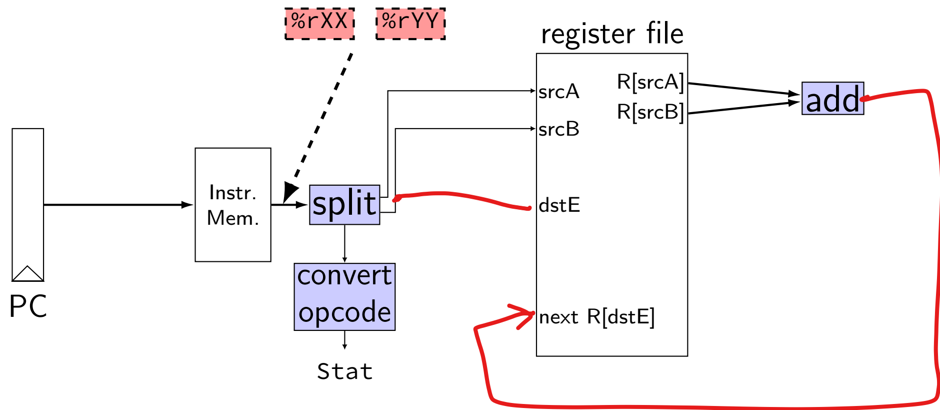
# addq CPU



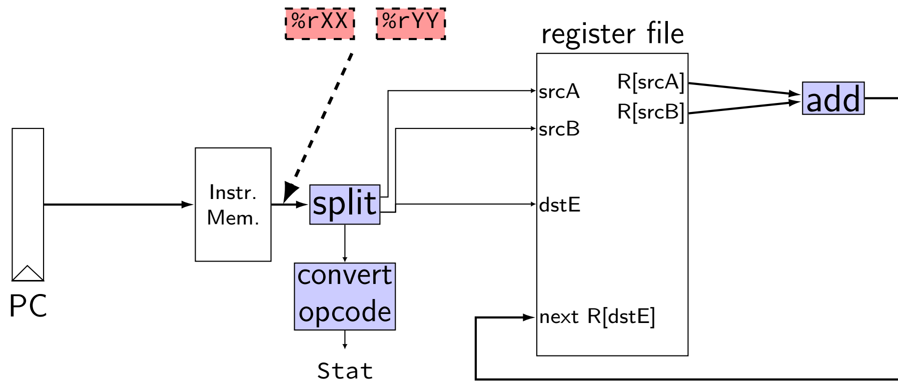
# addq CPU



# addq CPU



# addq CPU



# addq CPU

%rXX %rYY

register file

(value of RAX) 10

0 (=%rax) srcA R[srcA]

2 (=%rdx) srcB R[srcB]

add

30 (value of RDX)

2 (=%rdx) dstE

next R[dstE]

PC

Instr. Mem.

split

convert opcode

Stat

40

```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

initially: PC = 0x00, rax = 10, rbx = 20, rdx = 30

after cycle 1: PC = ????, rax = 10, rbx = 20, rdx = 40

# addq CPU

%rXX %rYY

register file

srcA R[srcA]

srcB R[srcB]

dstE

next R[dstE]

add

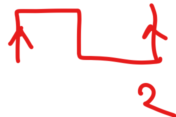
PC

Instr. Mem.

split

convert opcode

Stat



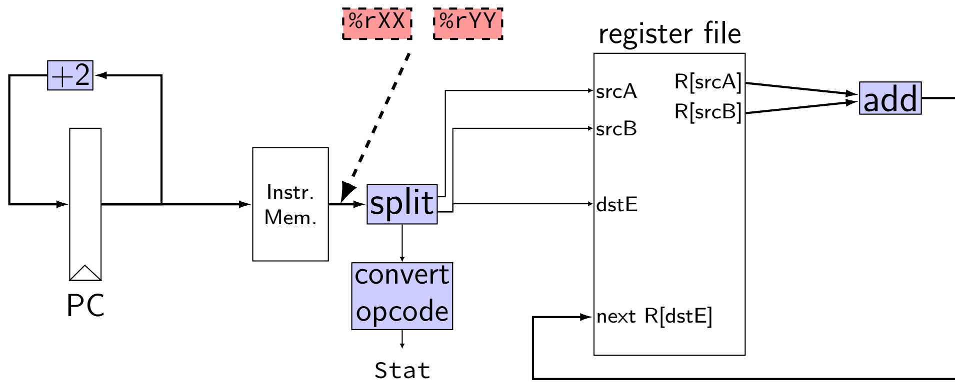
```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

initially: PC = 0x00, rax = 10, rbx = 20, rdx = 30

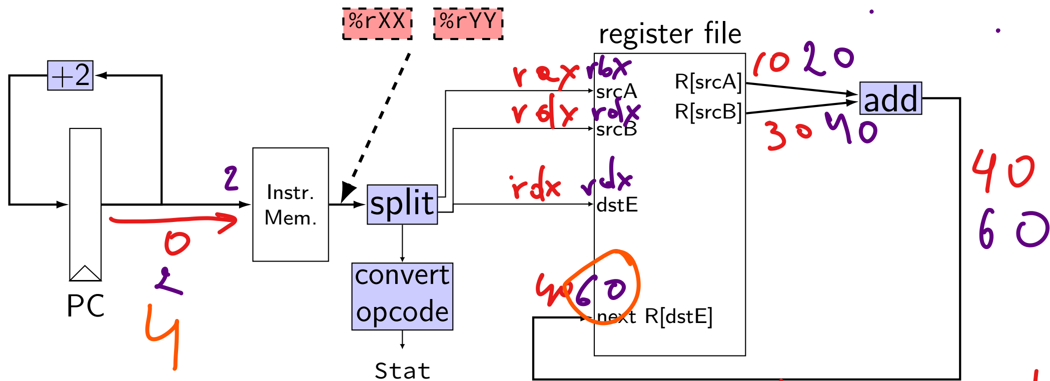
after cycle 1: PC = ????, rax = 10, rbx = 20, rdx = 40

after cycle 2: PC = ????, rax = ??, rbx = ??, rdx = ??

# addq CPU



# addq CPU



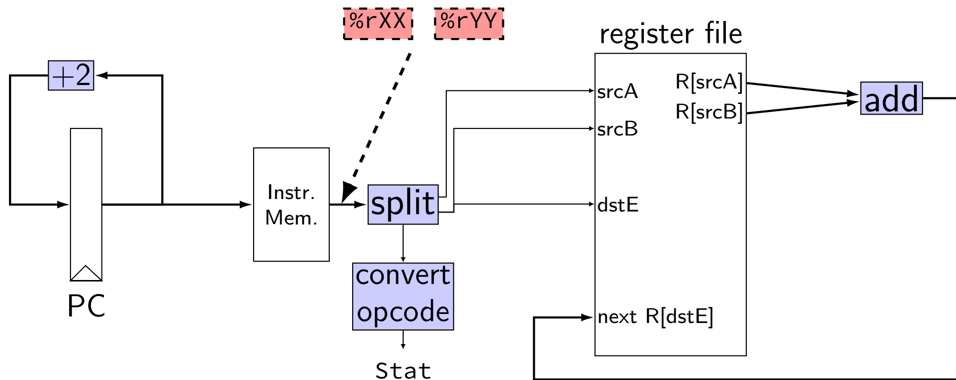

```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

initially: PC = 0x00, rax = 10, rbx = 20, rdx = 30

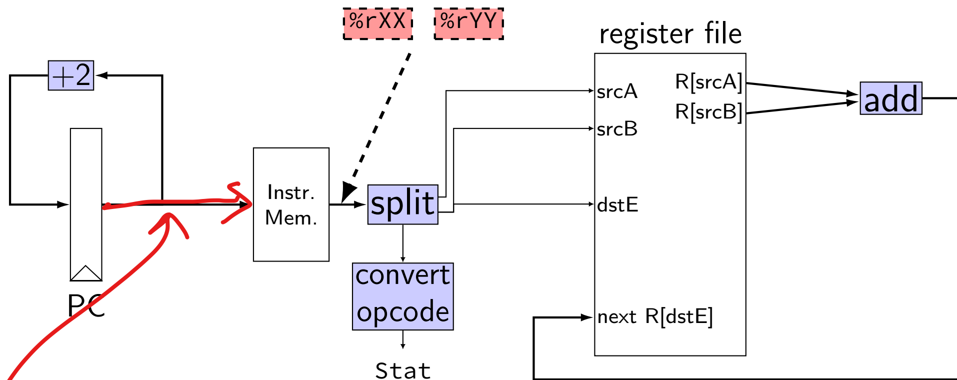
after cycle 1: PC = 0x02, rax = 10, rbx = 20, rdx = 40

after cycle 2: PC = 0x04, rax = 10, rbx = 20, rdx = 60

# addq CPU: HCL



# addq CPU: HCL

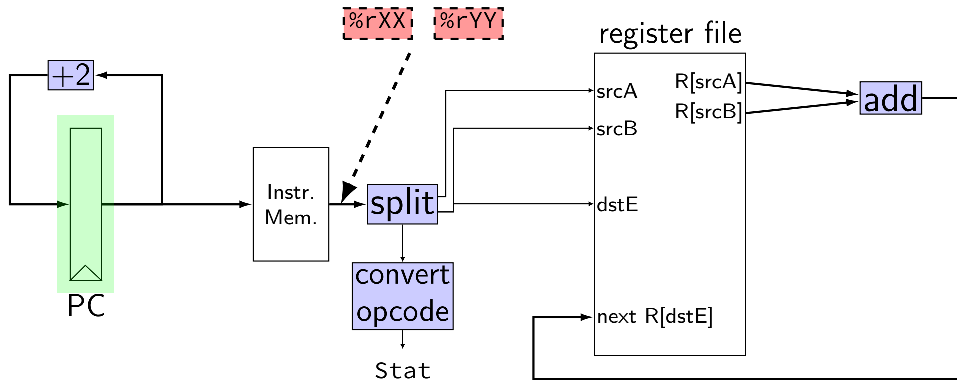



```
register pP {
      pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
      reg_outputA +
      reg_outputB;
```

# addq CPU: HCL

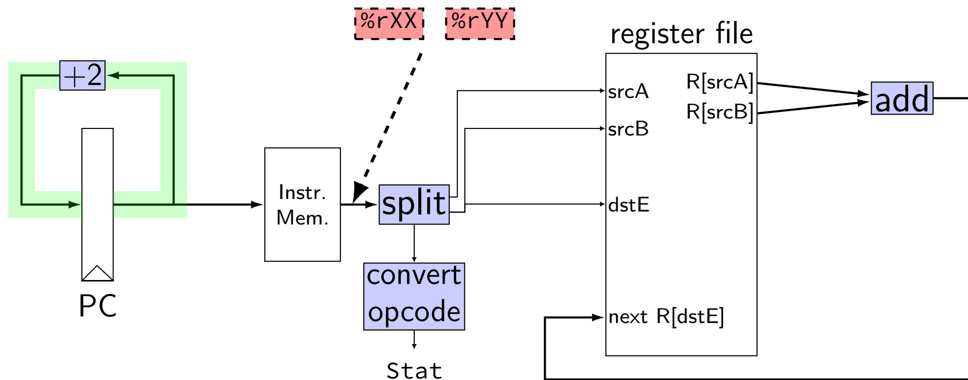



```
register pP {
    pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
      reg_outputA +
      reg_outputB;
```

# addq CPU: HCL

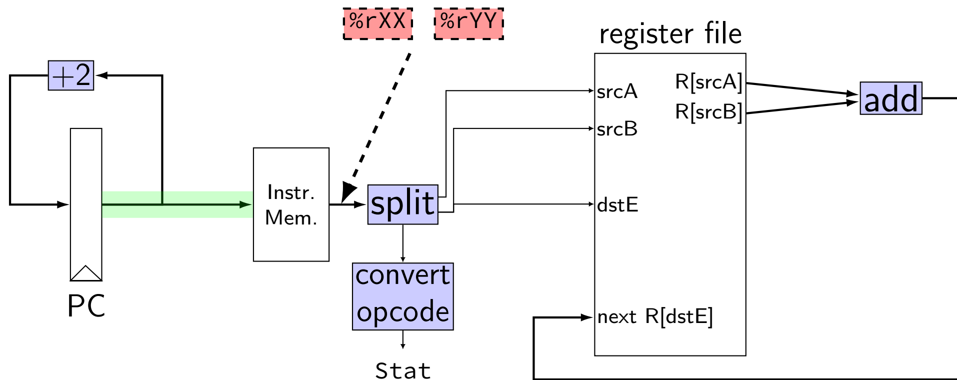



```
register pP {
    pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
      reg_outputA +
      reg_outputB;
```

# addq CPU: HCL

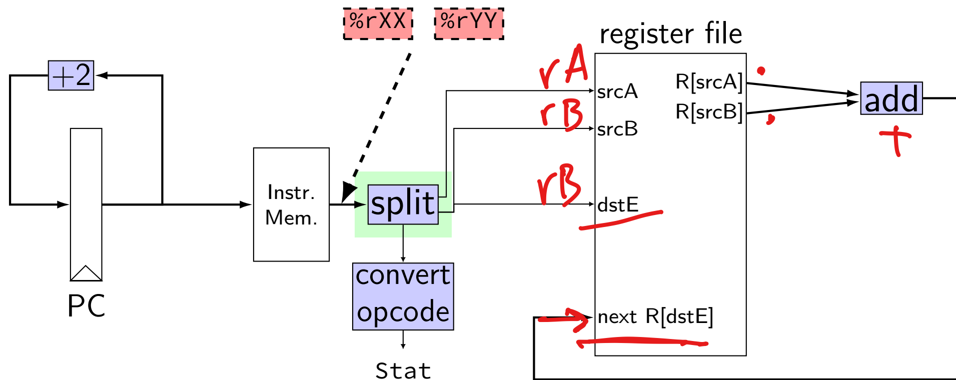



```
register pP {
    pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
      reg_outputA +
      reg_outputB;
```

# addq CPU: HCL

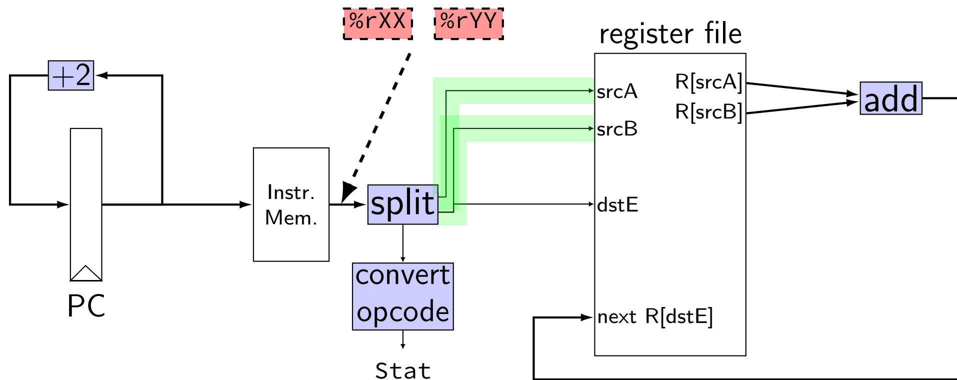



```
register pP {
      pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
        reg_outputA +
        reg_outputB;
```

# addq CPU: HCL

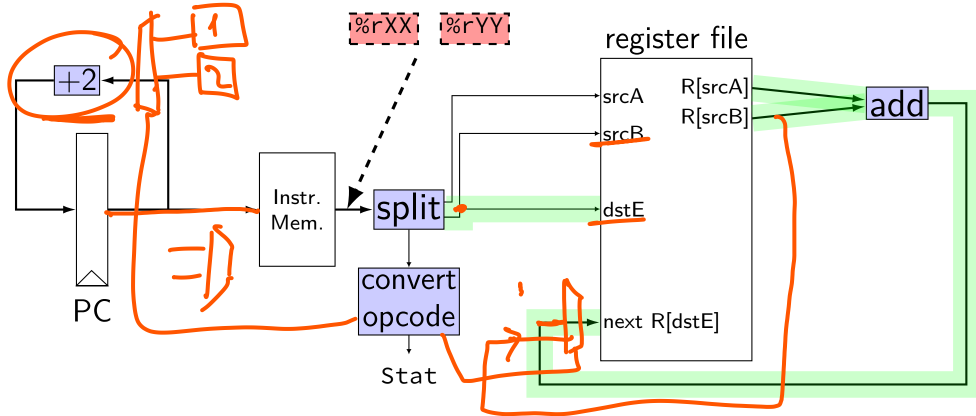



```
register pP {
    pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
      reg_outputA +
      reg_outputB;
```

# addq CPU: HCL




```
register pP {
    pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
wire opcode : 4;
wire rA : 4, rB : 4;
opcode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];
```

```
reg_srcA = rA;
reg_srcB = rB;
reg_dstE = rB;
reg_inputE =
      reg_outputA +
      reg_outputB;
```

# differences from book

**w**ire not **b**ool or **i**nt

book uses names like `valC` — not required!

author's environment limited adding new wires

MUXes must have default (`1 : something`) case

implement your own ALU

# differences from book

**w**ire not **b**ool or **i**nt

book uses names like `valC` — not required!

author's environment limited adding new wires

MUXes must have default (`1 : something`) case

implement your own ALU

# differences from book

**w**ire not **b**ool or **i**nt

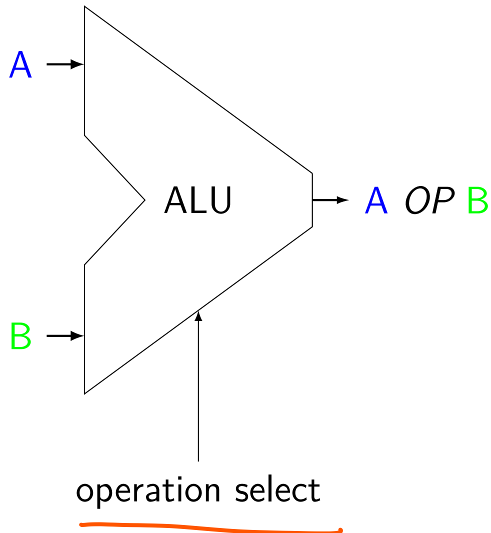
book uses names like `valC` — not required!

author's environment limited adding new wires

MUXes must have default (`1 : something`) case

implement your own ALU

# ALUs (Arithmetic Logic Unit)




A *OP* B

Operations needed:
add — **addq**, addresses
sub — **subq**
xor — **xorq**
and — **andq**
more?

# ALUs not for PC increment

our processor will have one ALU

not used for PC increment (computing next instruction address)

- need to do other computation in same cycle
- don't need a general circuit for it

i) ALU 1 operation
1 clk

# ALUs in HCLRS

HCLRS doesn't supply an ALU

- the HCL the textbook authors use does

...but you can build one yourself

- not required — we check functionality

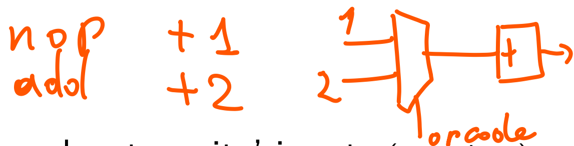
# exercise: nop/add CPU

Let's say we wanted to make a add+nop CPU. Where would we need MUXes? Before...

(modify add CPU to also support the nop instruction)

A. one or both of the register file 'register number to read' inputs (reg_src...)

B. the PC register's input (p_pc)




C. one of the register file 'register number to write' inputs (reg_dst...)

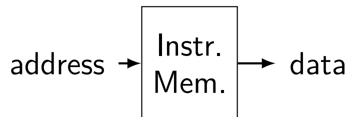
D. one of the register file 'register value to write' inputs (reg_input...)

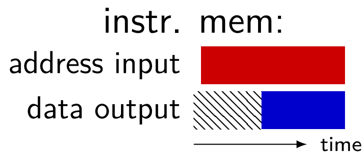
E. the instruction memory's address input (pc)

# simple ISA: mov-to-register

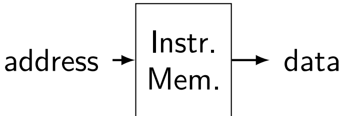
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

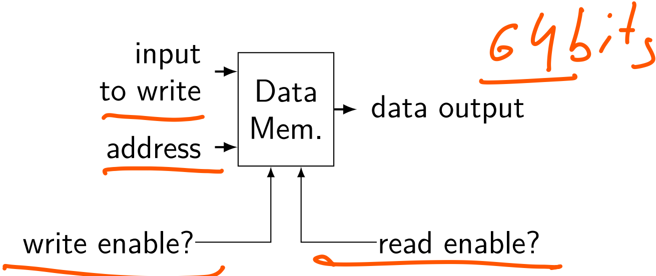
# two memories



# two memories

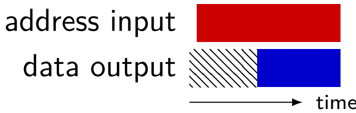


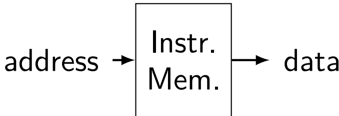

data mem. in **read** mode

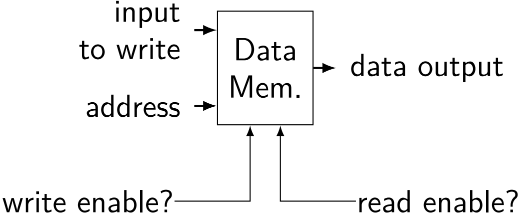
—or—

instr. mem:



# two memories

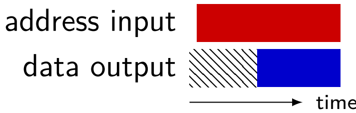


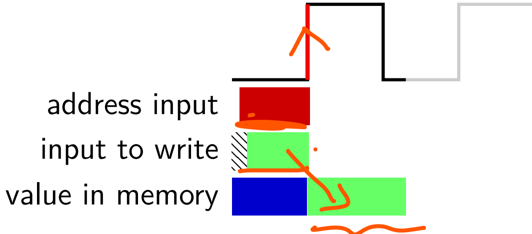

data mem. in **read** mode

—or—

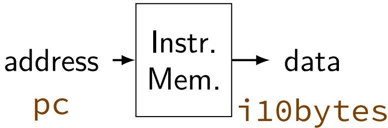
instr. mem:

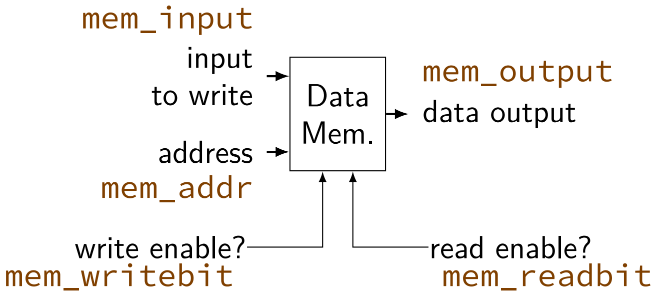


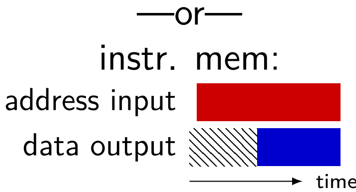

data mem.
in **write** mode:

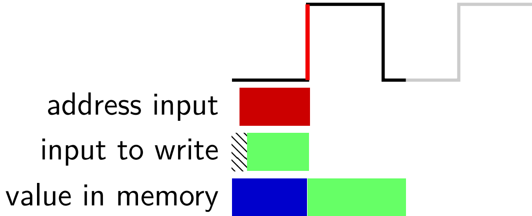


# two memories

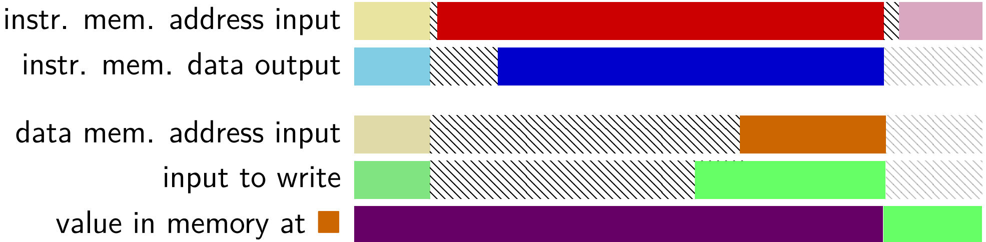


# two memories? (read + write)



# really two memories??

in Y86-64 (and many real CPUs):
writing to address $X$ in data memory:
changes address $X$ in instruction memory

# really two memories??

in Y86-64 (and many real CPUs):
writing to address $X$ in data memory:
changes address $X$ in instruction memory

so really just one memory??

we'll explain when we talk about *caches*

# exercise: mov-to-register

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

2

for which of these are we going to need MUXes? before…

A. register file's register number (index) inputs (`reg_srcA`, `reg_srcB`, `reg_dstE`, …)
B. register file's value inputs (`reg_inputE/M`)
C. PC register's input +2 +10
D. instruction memory's address input (pc)

# mov-to-register CPU

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

Instr.
Mem.

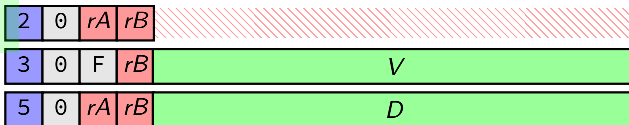
PC

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



# mov-to-register CPU

register file

srcA R[srcA]
srcB R[srcB]
dstE
next R[dstE]

PC

Instr. Mem.

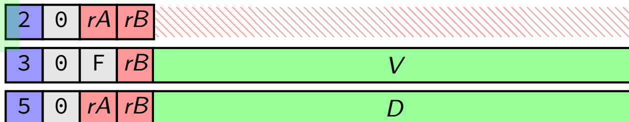
split

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



# mov-to-register CPU

register file

srcA R[srcA]
srcB R[srcB]
dstE
next R[dstE]

PC

Instr.
Mem.

split

Data in
Data out
Addr in

| | | | | | |
|---|---|---|---|---|---|
| `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

register file

srcA
srcB
dstE
R[srcA]
R[srcB]
next R[dstE]

PC

Instr.
Mem.

split

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*

# mov-to-register CPU

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

PC

Instr.
Mem.

split

convert
opcode

MUX

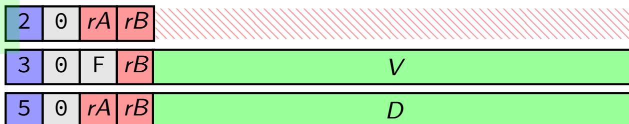
Data in
Data out
Addr in

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

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*



# mov-to-register CPU

immediate

register file

srcA
R[srcA]
R[srcB]
srcB
dstE
next R[dstE]

PC

Instr.
Mem.

split

convert
opcode

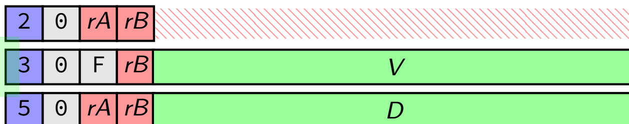
MUX

Data in
Data out
Addr in

**rrmovq** *rA*, *rB*

**irmovq** *V*, *rB*

**mrmovq** *D*(*rB*), *rA*



# mov-to-register CPU

immediate

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

PC

Instr.
Mem.

split

convert
opcode

MUX

+ (ALU)

Data in
Data out
Addr in

`rrmovq` *rA, rB*

`irmovq` *V, rB*

`mrmovq` *D(rB), rA*

# mov-to-register CPU

immediate

register file

srcA R[srcA]
srcB R[srcB]
dstE
next R[dstE]

PC

Instr. Mem.

split

convert opcode

MUX

+ (ALU)

Data in
Data out
Addr in

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

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D(rB)*, *rA*

# mov-to-register CPU

immediate

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

PC

Instr.
Mem.

split

convert
opcode

MUX

+2
+10

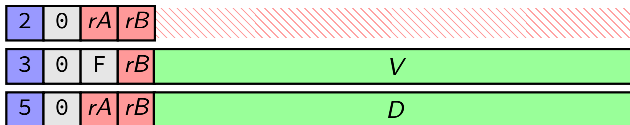
+ (ALU)

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

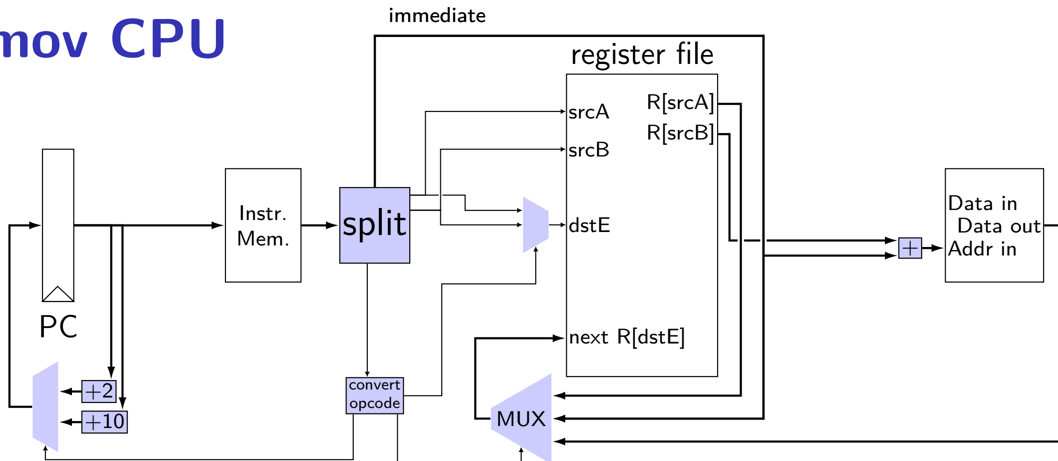
`mrmovq` *D*(*rB*), *rA*



# simple ISA: mov (all cases)

```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

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

# mov CPU




| Instruction | Encoding | | | | |
|---|---|---|---|---|---|
| `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

MUX

+2
+10

Data in
Data out
Addr in

`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

`mrmovq` *D*(*rB*), *rA*

`rmmovq` *rA*, *D*(*rB*)

# 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
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*)

# 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

immediate

register file

srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]

PC
Instr. Mem.
split
0xF
+2
+10
convert opcode
MUX

Data in
Data out
Addr in
+

write enable
from convert opcode

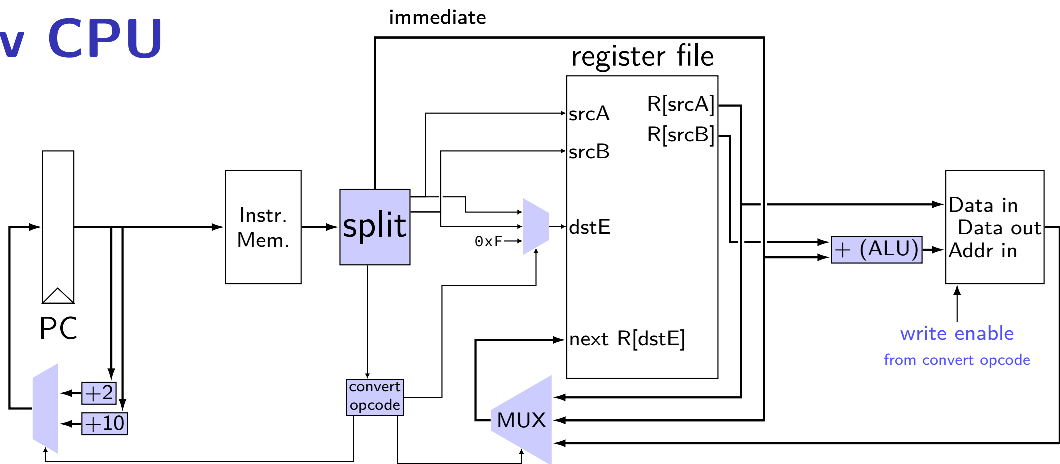
`rrmovq` *rA*, *rB*

`irmovq` *V*, *rB*

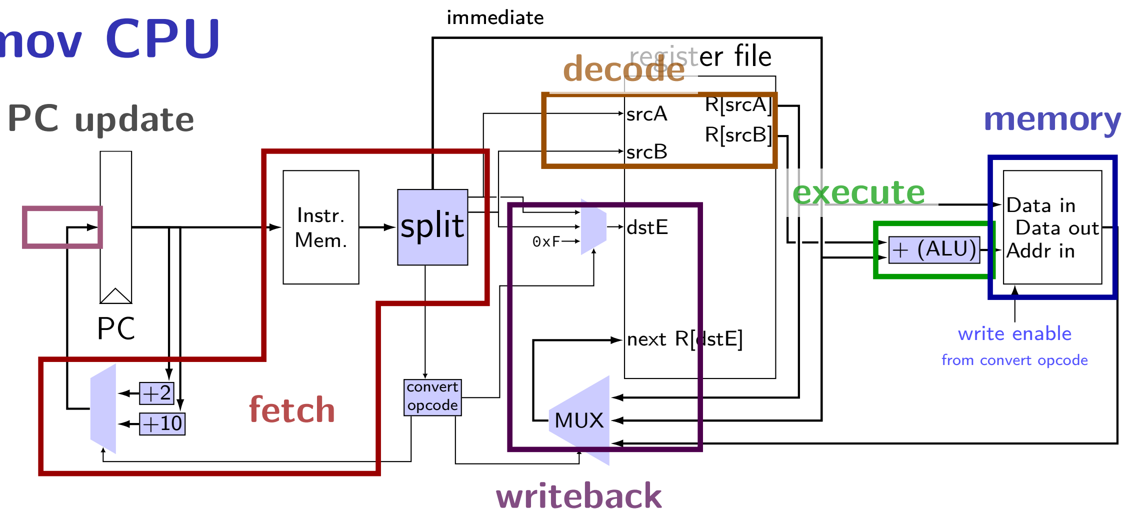
`mrmovq` *D*(*rB*), *rA*

`rmmovq` *rA*, *D*(*rB*)

# mov CPU



# mov CPU



# 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

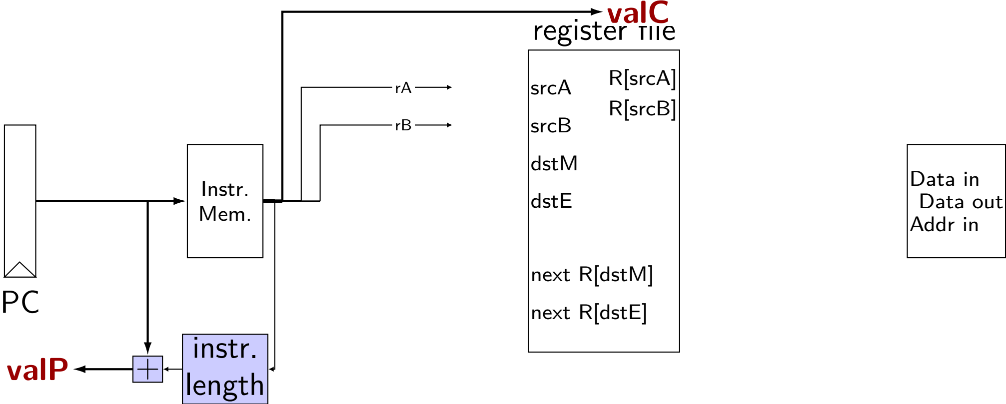
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

# instruction fetch

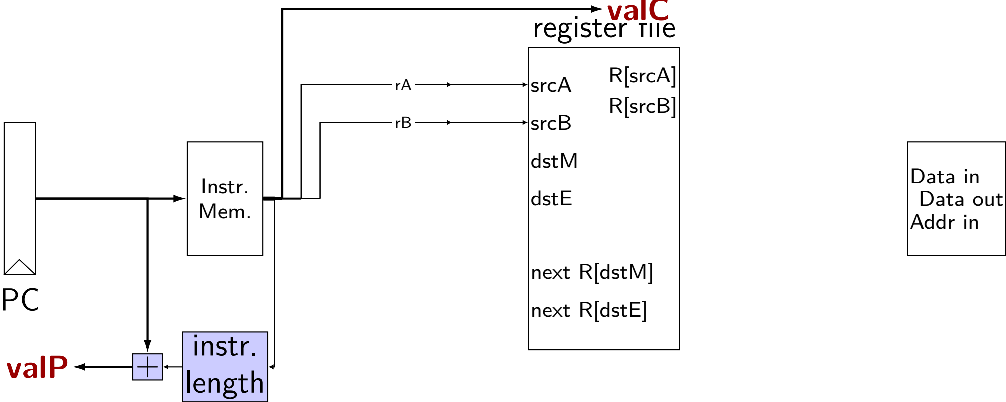


# SEQ: instruction “decode”

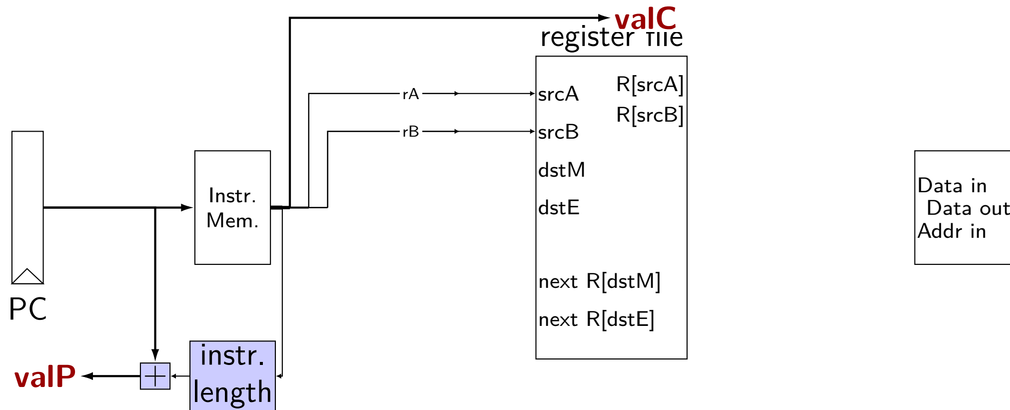
read registers

- valA, valB — register values

# instruction decode (1)



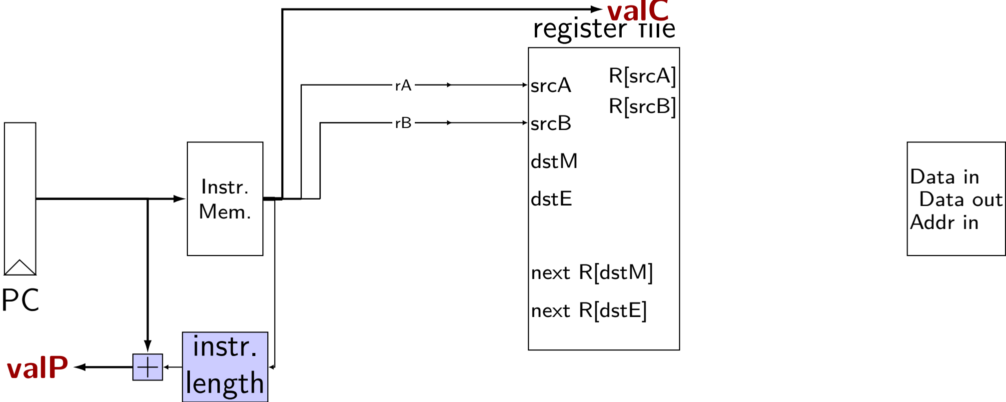
# instruction decode (1)




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:

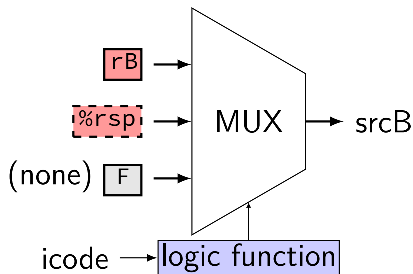
- push rA
- pop
- call
- ret

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

MUX controlled by icode

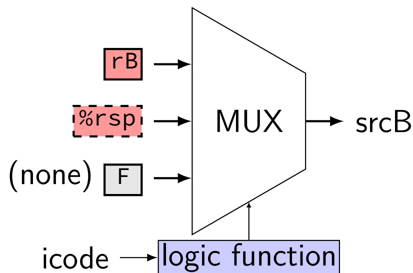
# SEQ: possible registers to read

| 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` |



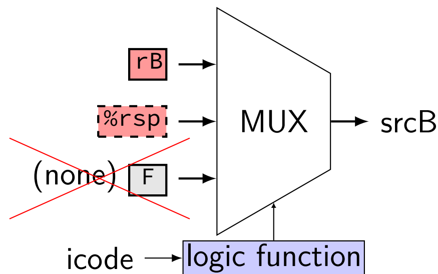
# SEQ: possible registers to read

| 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` |

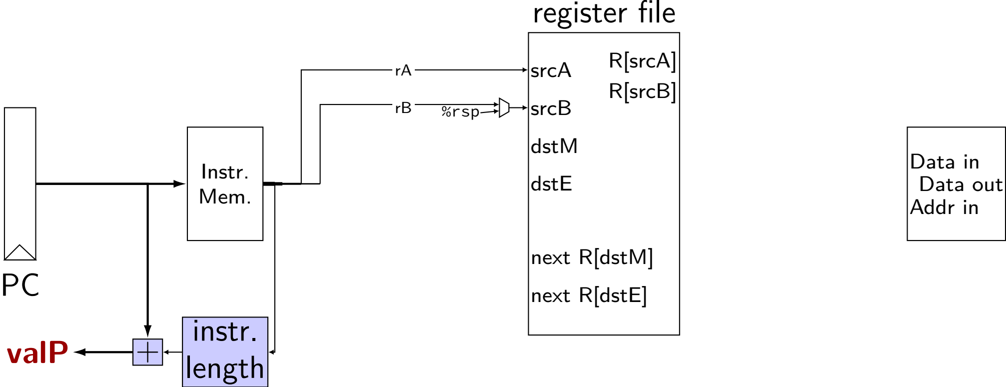


# SEQ: possible registers to read

| 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` |



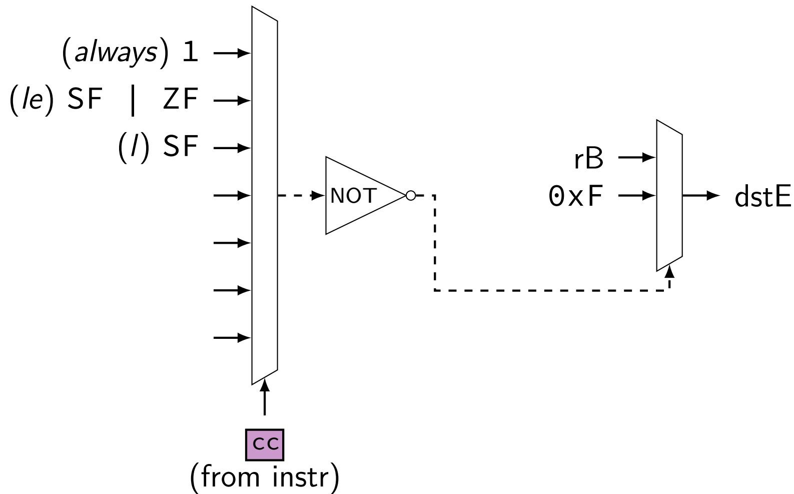
# instruction decode (2)



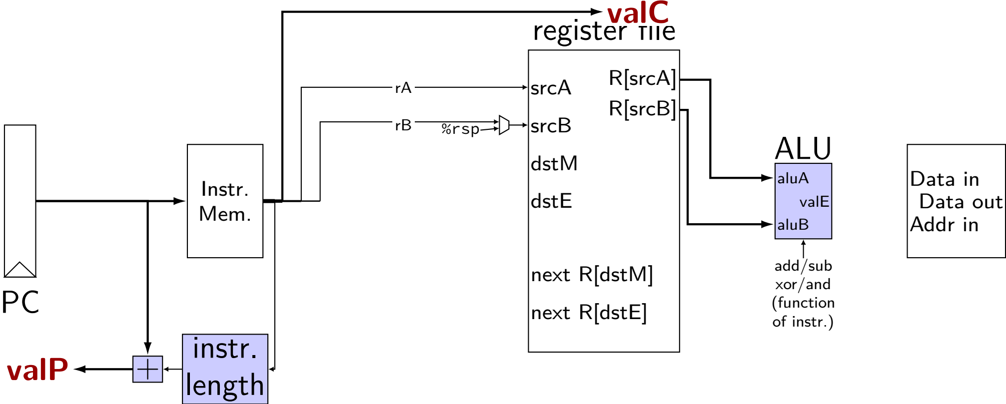
# SEQ: execute

- 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

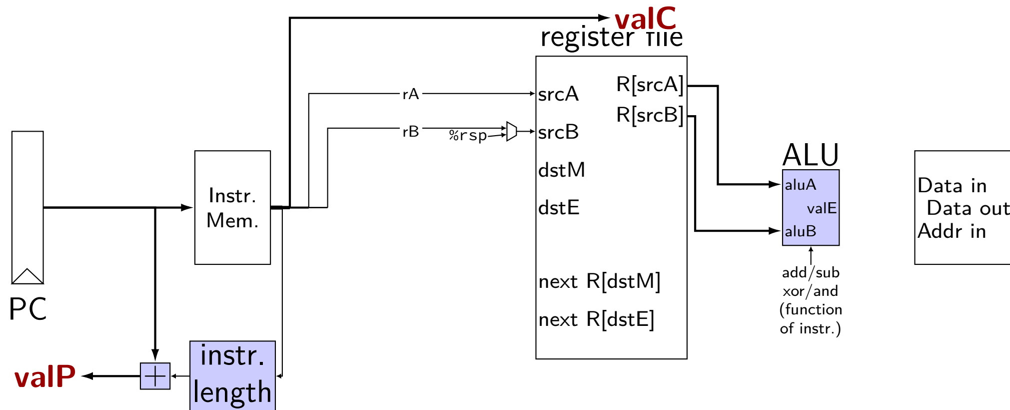
# using condition codes: cmov



# execute (1)



# execute (1)



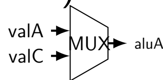

exercise: which of these instructions would there be a problem ?
`nop`, `addq`, `mrmovq`, `popq`, `call`,

# SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

`mrmovq`
`rmmovq`



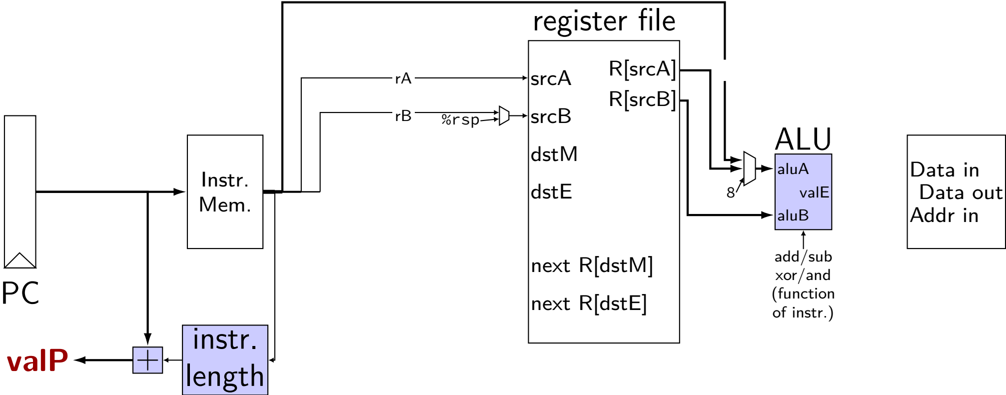

no, constants: (rsp +/- 8)

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

extra signals: aluA, aluB

computed ALU input values

# execute (2)

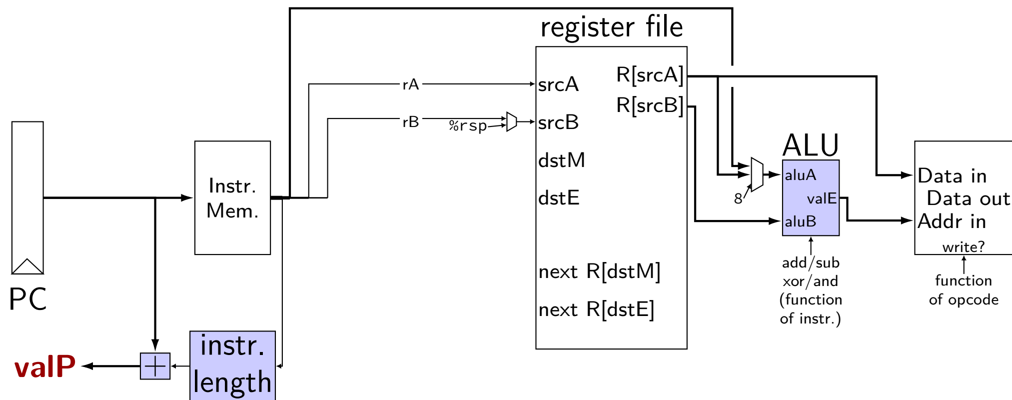


# SEQ: Memory

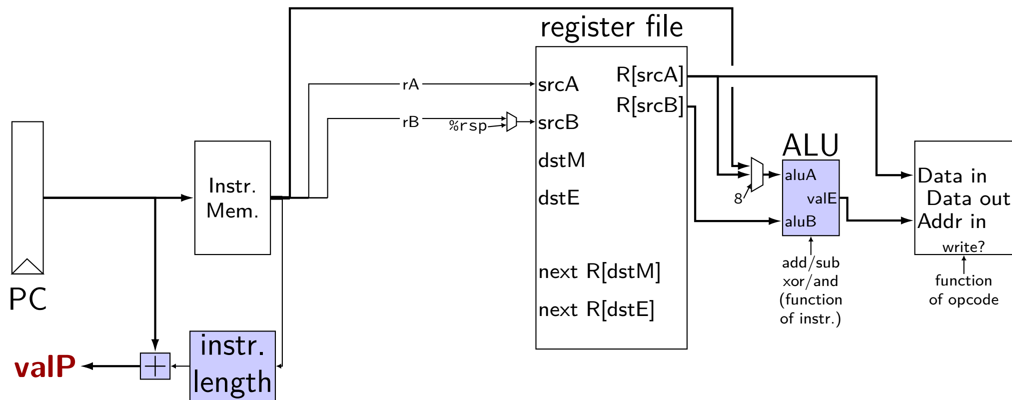
read or write data memory

- valM — value read from memory (if any)

# memory (1)



# 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?

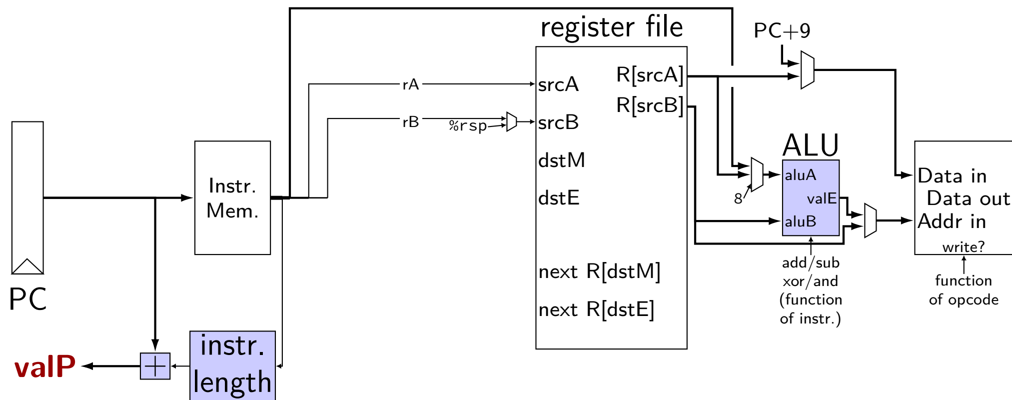
Addr — address

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

Data — value to write

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

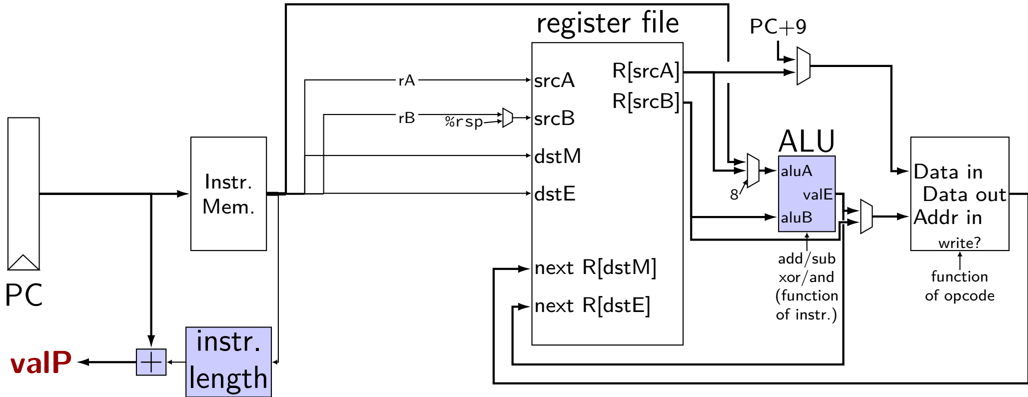
# memory (2)



# SEQ: write back

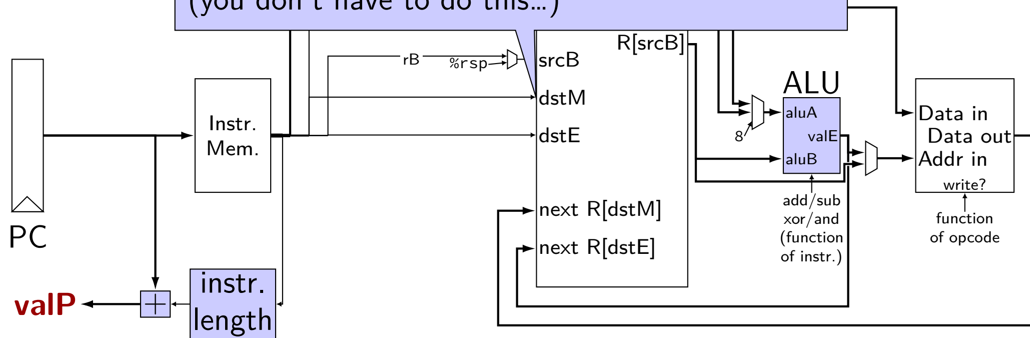
write registers

# write back (1)

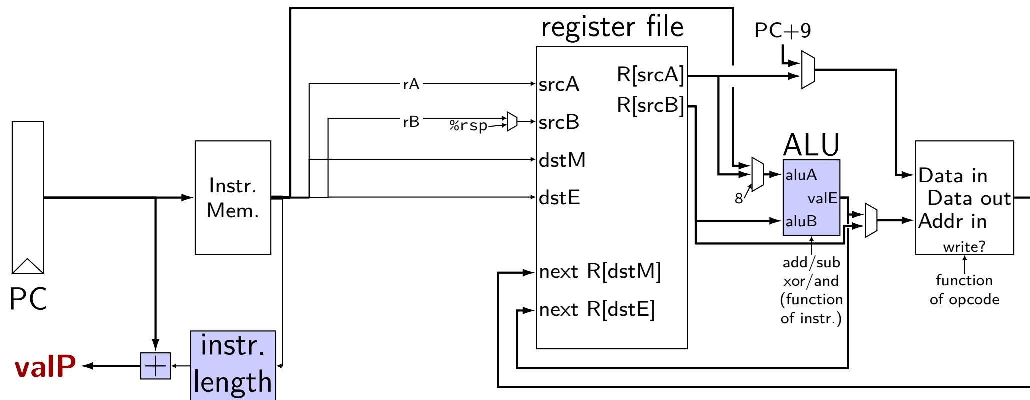


# write back (1)

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

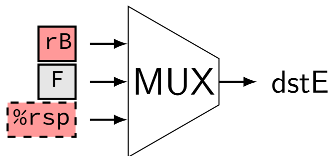
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

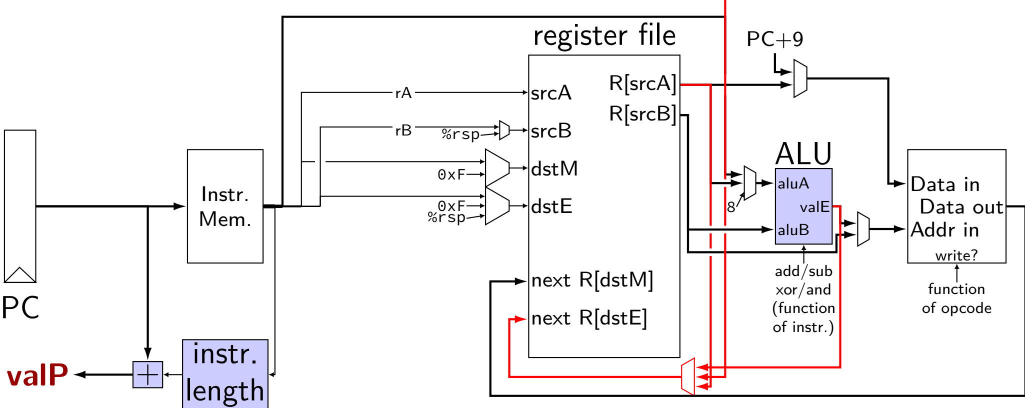
two register numbers

dstM, dstE

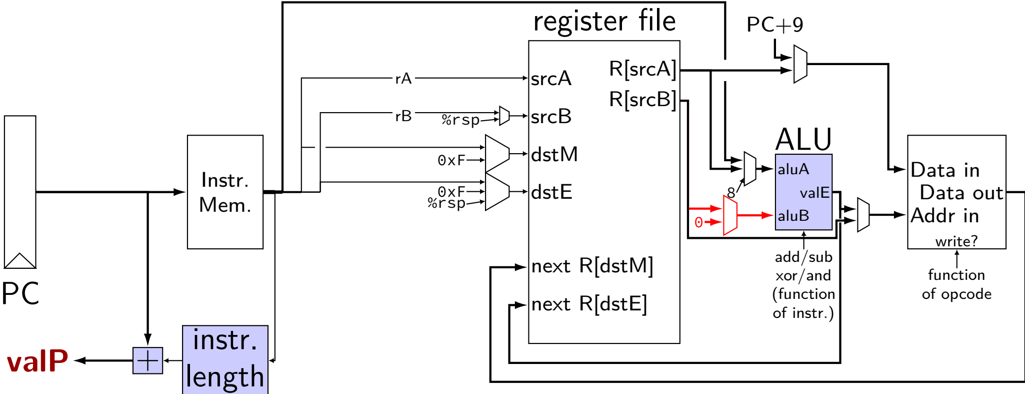
write disable — use dummy register number 0xF



# write back (2a)



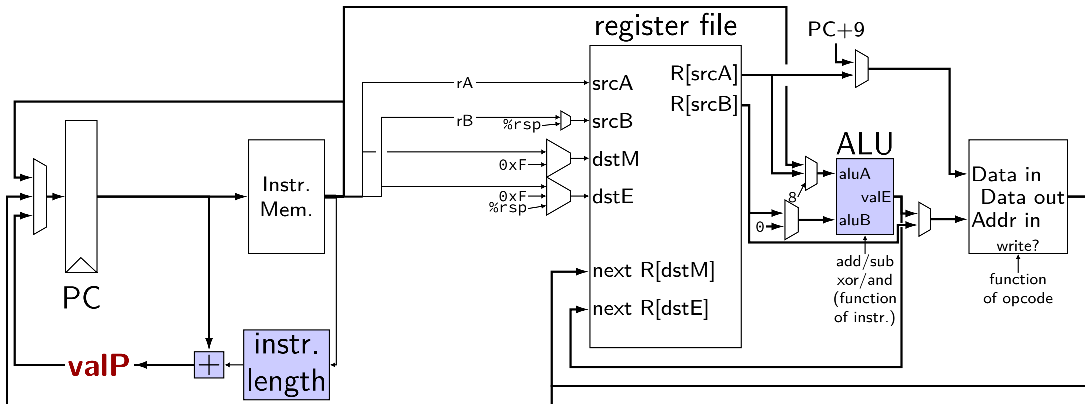
# write back (2b)



# SEQ: Update PC

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

# PC update



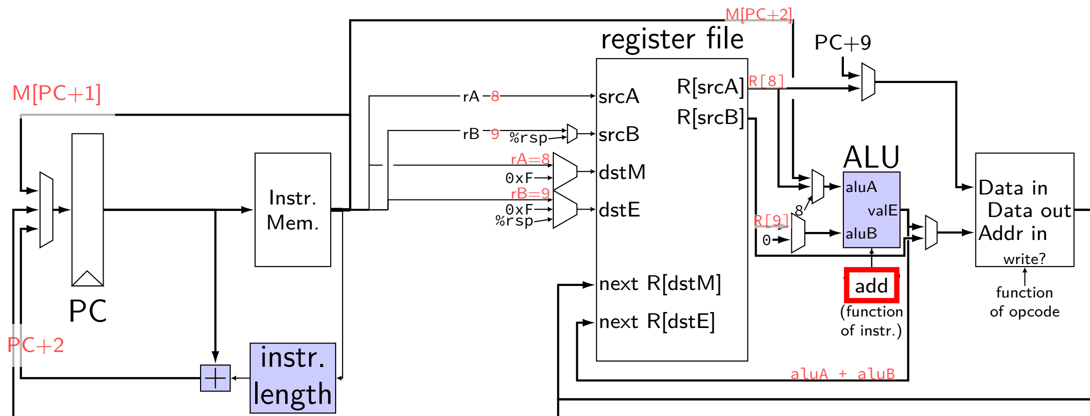
# circuit: setting MUXes

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

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

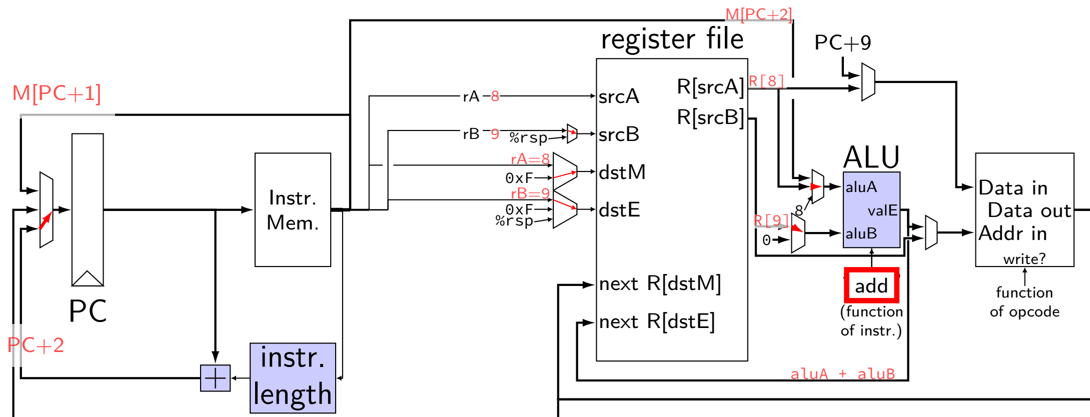
Exercise: what do they select when running `addq %r8, %r9`?

# circuit: setting MUXes




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

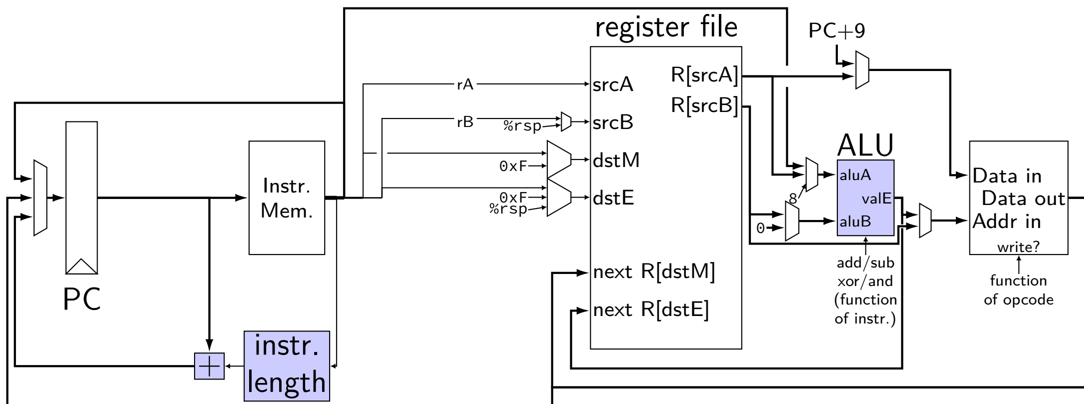
# circuit: setting MUXes




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

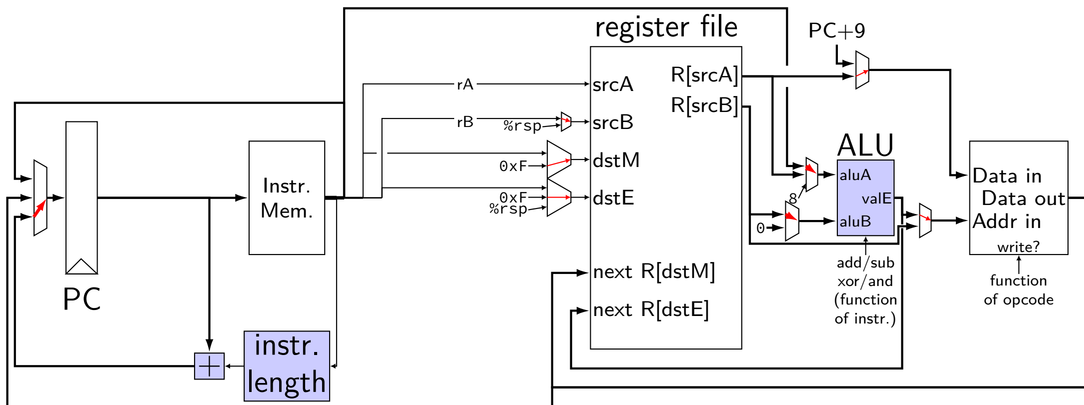
Exercise: what do they select when running `addq %r8, %r9`?

# circuit: setting MUXes




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

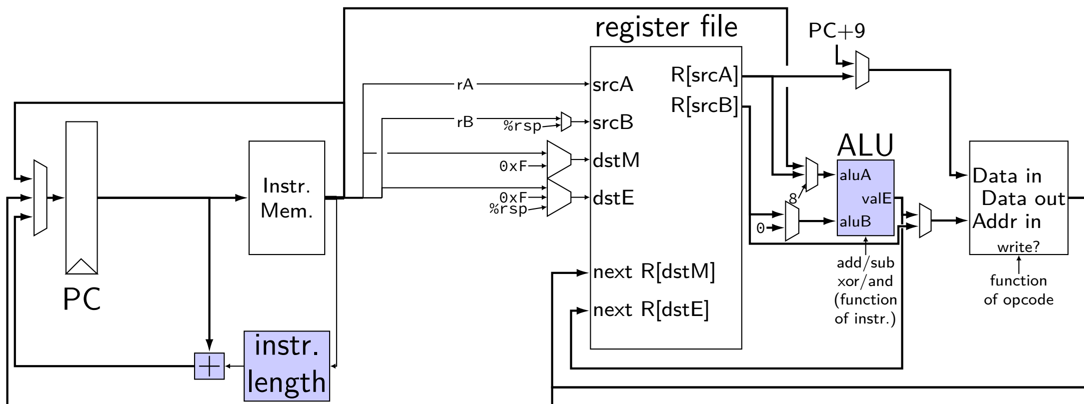
# circuit: setting MUXes




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

Exercise: what do they select for `rmmovq`?

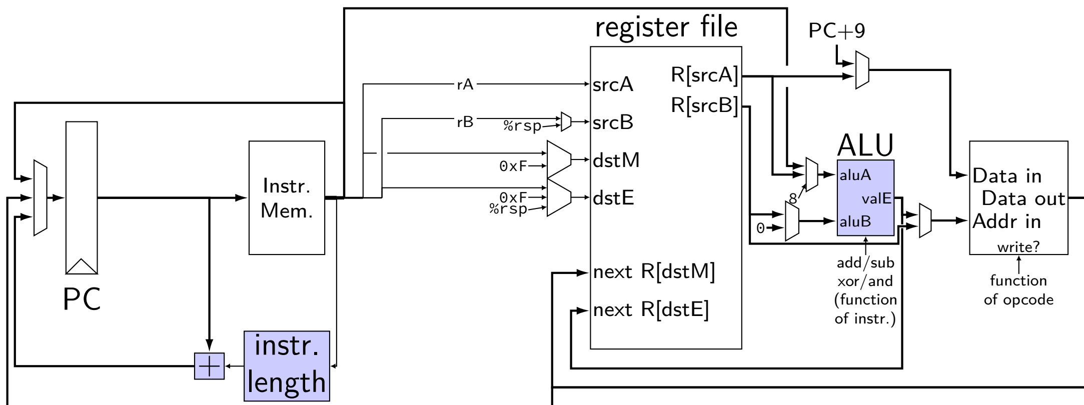
# circuit: setting MUXes




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

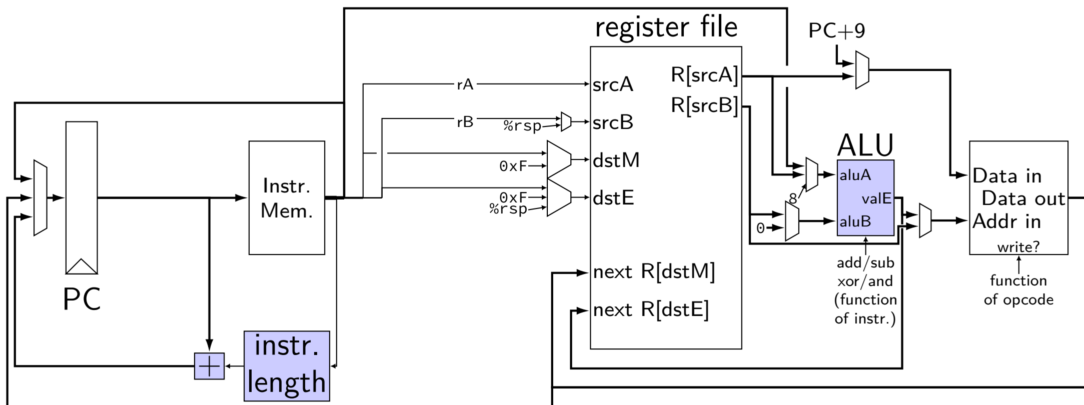
Exercise: what do they select for `irmovq`?

# circuit: setting MUXes




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

# circuit: setting MUXes



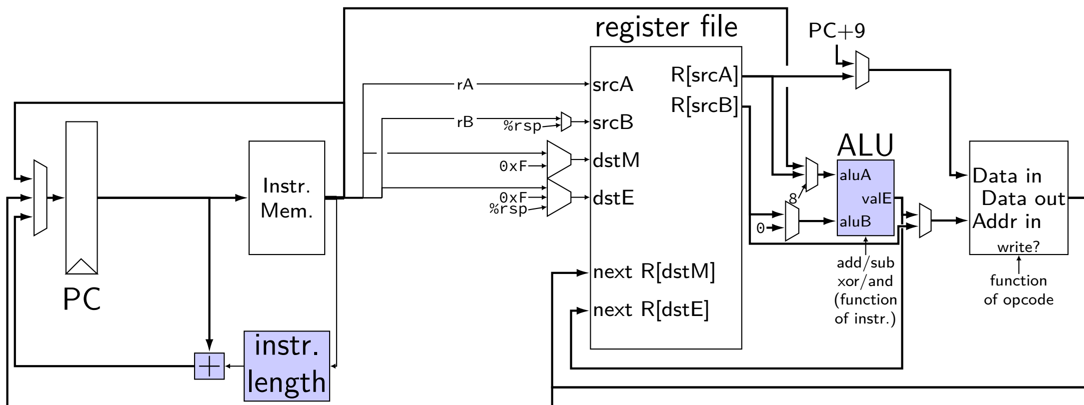

MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `jle`?

# circuit: setting MUXes



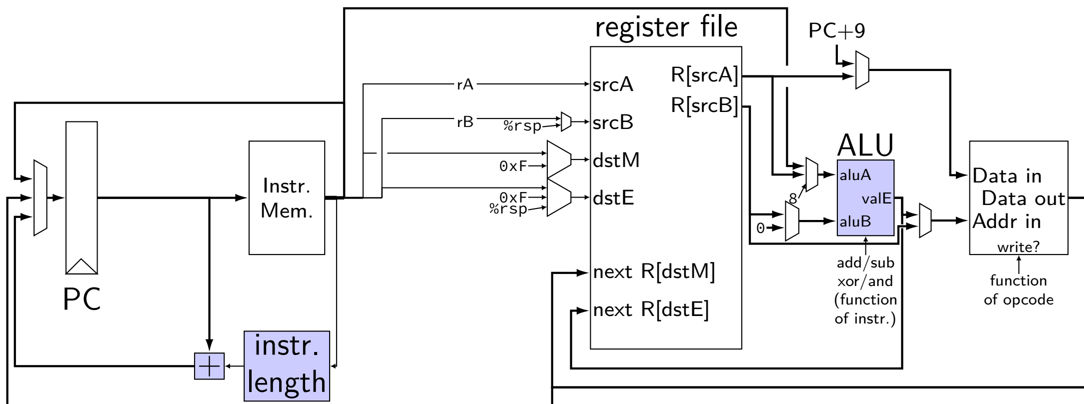
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `cmovle`?

# circuit: setting MUXes



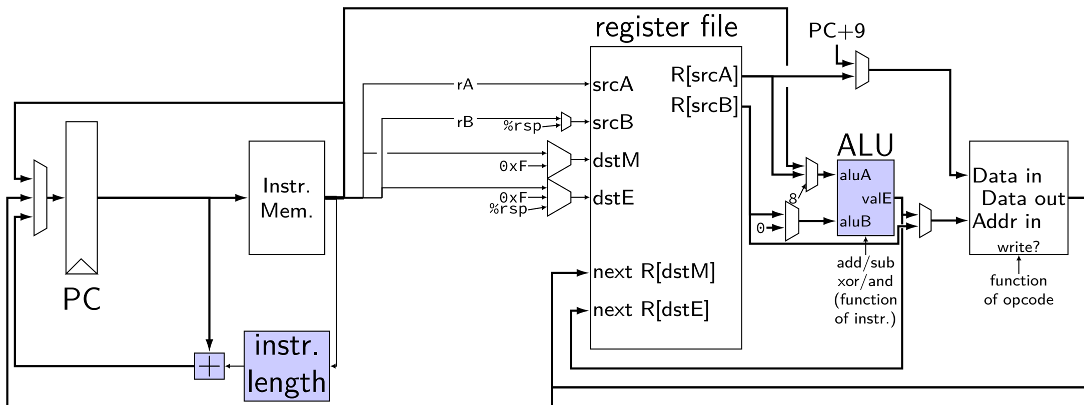

MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `ret`?

# circuit: setting MUXes




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

# circuit: setting MUXes




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

# 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)