HCL 2 / SEQ continued

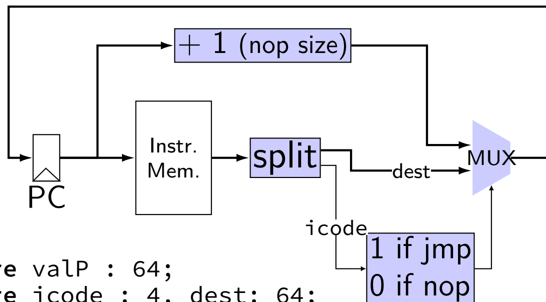
last time

building less simple CPUs

making decisions with MUXes

start HCL — describing circuits with text

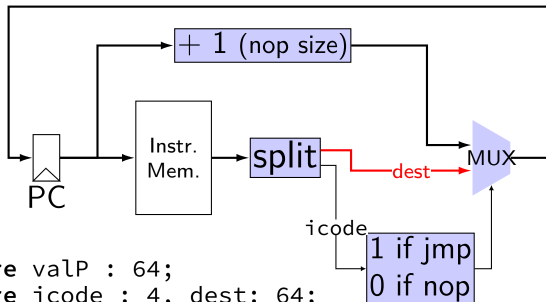
nop/jmp CPU




```
wire valP : 64;
wire icode : 4, dest: 64;
register pP {
    thePc : 64 = 0;
}
icode = i10bytes[4..8];
dest = i10bytes[8..72];
valP = [
    icode == NOP : P_thePc + 1;
    icode == JXX : dest;
    1: 0xBADBADBAD;
];
p_thePc = valP;
pc = P_thePc;
```

```
Stat = [
    (icode == NOP ||
     icode == JXX) : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
```

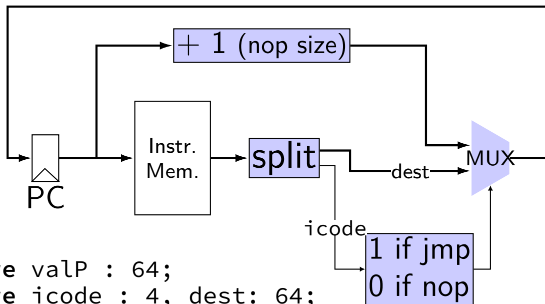
nop/jmp CPU




```
wire valP : 64;
wire icode : 4, dest: 64;
register pP {
    thePc : 64 = 0;
}
icode = i10bytes[4..8];
dest = i10bytes[8..72];
valP = [
    icode == NOP : P_thePc + 1;
    icode == JXX : dest;
    1: 0xBADBADBAD;
];
p_thePc = valP;
pc = P_thePc;
```

```
Stat = [
    (icode == NOP ||
     icode == JXX) : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
```

nop/jmp CPU




```
wire valP : 64;
wire icode : 4, dest: 64;
register pP {
    thePc : 64 = 0;
}
icode = i10bytes[4..8];
dest = i10bytes[8..72];
valP = [
    icode == NOP : P_thePc + 1;
    icode == JXX : dest;
    1: 0xBADBADBAD;
];
p_thePc = valP;
pc = P_thePc;
```

```
Stat = [
    (icode == NOP ||
     icode == JXX) : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
```

running nop/jmp/halt

`nopjmp.ys`:

```
     nop
     jmp C
B:   jmp D
C:   jmp B
D:   nop
     nop
     halt
```

…assemble with `yas`

nopjmp.yo

`nopjmp.yo`:

```
0x000: 10                    |      nop
0x001: 701300000000000000    |      jmp C
0x00a: 701c00000000000000    | B:   jmp D
0x013: 700a00000000000000    | C:   jmp B
0x01c: 10                    | D:   nop
0x01d: 10                    |      nop
0x01e: 00                    |      halt
```

nopjmp.yo

nopjmp.yo:

```
0x000: 10                     |        nop
0x001: 701300000000000000     |        jmp C
0x00a: 701c00000000000000     |  B:    jmp D
0x013: 700a00000000000000     |  C:    jmp B
0x01c: 10                     |  D:    nop
0x01d: 10                     |        nop
0x01e: 00                     |        halt
```

running nopjmp.yo

```
$ ./hclrs nopjmp_cpu.hcl nopjmp.yo
...
...
+--------------------- (end of halted state) ---------------------------+
Cycles run: 7
```

debugging mode

```
+------------------- between cycles    0 and    1 ----------------------+
| RAX:                 0   RCX:                 0   RDX:                 0 |
| RBX:                 0   RSP:                 0   RBP:                 0 |
| RSI:                 0   RDI:                 0   R8:                  0 |
| R9:                  0   R10:                 0   R11:                 0 |
| R12:                 0   R13:                 0   R14:                 0 |
| register pP(N)  thePc=0000000000000000                               |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00    |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00       |
+-----------------------------------------------------------------------+
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of wires:
Wire                           Value
dest              0x0000000000001370
i10bytes      0x00000000000000137010
icode                            0x1
pc                0x0000000000000000
P_thePc           0x0000000000000000
p_thePc           0x0000000000000001
Stat                             0x1
valP              0x0000000000000001
.------------------- between cycles    1 and    2 ----------------------+
...
```

debugging mode

```
+------------------- between cycles    0 and    1 ----------------------+
| RAX:                 0   RCX:                 0   RDX:                 0 |
| RBX:                 0   RSP:                 0   RBP:                 0 |
| RSI:                 0   RDI:                 0   R8:                  0 |
| R9:                  0   R10:                 0   R11:                 0 |
| R12:                 0   R13:                 0   R14:                 0 |
| register pP(N)  thePc=0000000000000000                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00    |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00       |
+-----------------------------------------------------------------------+
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of wires:
Wire                        Value
dest             0x0000000000001370
i10bytes     0x00000000000000137010
icode                           0x1
pc               0x0000000000000000
P_thePc          0x0000000000000000
p_thePc          0x0000000000000001
Stat                            0x1
valP             0x0000000000000001
.------------------- between cycles    1 and    2 ----------------------+
...
```

interactive + debugging mode

```
$ ./nopjmp_cpu.exe -i -d nopjmp.yo
+------------------- between cycles    0 and    1 ----------------------+
| RAX:               0   RCX:               0   RDX:               0 |
| RBX:               0   RSP:               0   RBP:               0 |
| RSI:               0   RDI:               0   R8:                0 |
| R9:                0   R10:               0   R11:               0 |
| R12:               0   R13:               0   R14:               0 |
| register pP(N)  thePc=0000000000000000                              |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f   |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00   |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00      |
+----------------------------------------------------------------------+
(press enter to continue)
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of wires:
Wire                        Value
dest           0x0000000000001370
i10bytes   0x00000000000000137010
icode                         0x1
pc             0x0000000000000000
P_thePc        0x0000000000000000
p_thePc        0x0000000000000001
Stat                          0x1
valP           0x0000000000000001
+------------------- between cycles    1 and    2 ----------------------+
```

interactive + debugging mode

```
$ ./nopjmp_cpu.exe -i -d nopjmp.yo
+------------------- between cycles    0 and    1 ----------------------+
| RAX:                0   RCX:                0   RDX:                0 |
| RBX:                0   RSP:                0   RBP:                0 |
| RSI:                0   RDI:                0   R8:                 0 |
| R9:                 0   R10:                0   R11:                0 |
| R12:                0   R13:                0   R14:                0 |
| register pP(N)  thePc=0000000000000000                               |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00    |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00       |
+-----------------------------------------------------------------------+
(press enter to continue)
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of wires:
Wire                        Value
dest           0x0000000000001370
i10bytes   0x00000000000000137010
icode                         0x1
pc             0x0000000000000000
P_thePc        0x0000000000000000
p_thePc        0x0000000000000001
Stat                          0x1
valP           0x0000000000000001
+------------------- between cycles    1 and    2 ----------------------+
```

quiet mode

```
$ ./hclrs nopjmp_cpu.hcl -q nopjmp.yo
+----------------------- halted in state: ------------------------------+
| RAX:                0   RCX:                0   RDX:                0 |
| RBX:                0   RSP:                0   RBP:                0 |
| RSI:                0   RDI:                0   R8:                 0 |
| R9:                 0   R10:                0   R11:                0 |
| R12:                0   R13:                0   R14:                0 |
| register pP(N) { thePc=0000000000000000 }                             |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00    |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00       |
+--------------------- (end of halted state) ---------------------------+
Cycles run: 7
```

differences from book

wire 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

wire 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

wire 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

things in HCLRS

register banks

wires

things for our processor:
- `Stat` register
- instruction memory
- the register file
- data memory

things in HCLRS

register banks

wires

things for our processor:

- `Stat` register
- instruction memory
- the register file
- data memory

register banks

```
register xY {
    foo : width1 = defaultValue1;
    bar : width2 = defaultValue2;
}
```

two letters: input (X) / Output (Y)

- input signals: `x_foo`, `x_bar`
- output signals: `Y_foo`, `Y_bar`

each value has width in bits

each value has initial value — *mandatory*

some other signals — `stall`, `bubble`

- later in semester

register banks

```
register xY {
    foo : width1 = defaultValue1;
    bar : width2 = defaultValue2;
}
```

two letters: input (X) / Output (Y)

input signals: `x_foo`, `x_bar`

output signals: `Y_foo`, `Y_bar`

each value has width in bits

each value has initial value — *mandatory*

some other signals — `stall`, `bubble`

later in semester

register banks

```
register xY {
    foo : width1 = defaultValue1;
    bar : width2 = defaultValue2;
}
```

two letters: input (X) / Output (Y)

- input signals: `x_foo`, `x_bar`
- output signals: `Y_foo`, `Y_bar`

each value has width in bits

each value has initial value — *mandatory*

some other signals — `stall`, `bubble`

- later in semester

things in HCLRS

register banks

wires

things for our processor:

- `Stat` register
- instruction memory
- the register file
- data memory

wires

```
wire wireName : wireWidth;
wireName = ...;
... = wireName;
... = wireName;
```

things that can accept/produce a signal

- some created implicitly – e.g. by creating register
- some builtin — supplied components (like instruction memory)

assignment — connecting wires

wires and order

```
wire icode : 4;
wire valP : 64;
register pP {
    thePc : 64 = 0;
}
p_thePc = valP;
pc = P_thePc;
Stat = [
    icode == NOP : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
valP = P_thePC + 1;
icode = i10bytes[4..8];
```

```
wire icode : 4;
wire valP : 64;
register pP {
    thePc : 64 = 0;
}
valP = P_thePC + 1;
p_thePc = valP;
pc = P_thePc;
icode = i10bytes[4..8];
Stat = [
    icode == NOP : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
```

wires and order

```
wire icode : 4;
wire valP : 64;
register pP {
    thePc : 64 = 0;
}
p_thePc = valP;
pc = P_thePc;
Stat = [
    icode == NOP : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
valP = P_thePC + 1;
icode = i10bytes[4..8];
```

```
wire icode : 4;
wire valP : 64;
register pP {
    thePc : 64 = 0;
}
valP = P_thePC + 1;
p_thePc = valP;
pc = P_thePc;
icode = i10bytes[4..8];
Stat = [
    icode == NOP : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
```

wires and order

```
wire icode : 4;
wire valP : 64;
register pP {
    thePc : 64 = 0;
}
p_thePc = valP;
pc = P_thePc;
Stat = [
    icode == NOP : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
valP = P_thePC + 1;
icode = i10bytes[4..8];
```

```
wire icode : 4;
wire valP : 64;
register pP {
    thePc : 64 = 0;
}
valP = P_thePC + 1;
p_thePc = valP;
pc = P_thePc;
icode = i10bytes[4..8];
Stat = [
    icode == NOP : STAT_AOK;
    icode == HALT : STAT_HLT;
    1 : STAT_INS;
];
```

order doesn't matter

wire is connected or not connected

wires and width

```
wire bigValueOne: 64;
wire bigValueTwo: 64;
wire smallValue: 32;
bigValueOne = smallValue; /* ERROR */
smallValue = bigValueTwo; /* ERROR */
…

wire bigValueOne: 64;
wire bigValueTwo: 64;
wire smallValue: 32;

smallValue = bigValueTwo[0..32]; /* OKAY */
```

constants and width

`10`, `0x8F3` — no width
(convert to any width)

`0b1010` — 4 bits (binary `1010` = 10)

most built-in constants `STAT_AOK`, `NOP`, etc. have widths

things in HCLRS

register banks

wires

things for our processor:

- `Stat` register
- instruction memory
- the register file
- data memory

Stat register

how do we stop the machine?

hard-wired mechanism — `Stat` register

possible values:

- `STAT_AOK` — keep going
- `STAT_HLT` — stop, normal shtdown
- `STAT_INS` — invalid instruction
- …(and more errors)

must be set

determines if simulator keeps going

things in HCLRS

register banks

wires

things for our processor:

- `Stat` register
- instruction memory
- the register file
- data memory

program memory

input wire: `pc`

output wire: `i10bytes`

- 80-bits wide (10 bytes)
- bit 0 — least significant bit of first byte
- (width of largest instruction)

program memory

input wire: `pc`

output wire: `i10bytes`

- 80-bits wide (10 bytes)
- bit 0 — least significant bit of first byte
- (width of largest instruction)

what about less than 10 byte instructions?

- just don't use the extra bits

things in HCLRS

register banks

wires

things for our processor:

- `Stat` register
- instruction memory
- the register file
- data memory

register file

four register number inputs (4-bit):
- sources: `reg_srcA`, `reg_srcB`
- destinations: `reg_dstE`, `reg_dstM`

no write or no read? register number `0xF` (`REG_NONE`)

two register value inputs (64-bit):
- `reg_inputE`, `reg_inputM`

two register output values (64-bit):
- `reg_outputA`, `reg_outputB`

example using register file: add CPU

```
wire rA : 4, rB : 4, icode : 4, ifunc: 4;
register pP {
    thePC : 64 = 0;
}
/* Fetch + PC update: */
pc = P_thePC; p_thePC = P_thePC + 2;
icode = i10bytes[4..8]; ifunc = i10bytes[0..4];
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode: */
reg_srcA = rA;
reg_srcB = rB;
/* Execute + Writeback: */
reg_inputE = reg_outputA + reg_outputB;
reg_dstE = rB;
/* Status maintainence: */
Stat = ...
```

example using register file: add CPU

```
wire rA : 4, rB : 4, icode : 4, ifunc: 4;
register pP {
    thePC : 64 = 0;
}
/* Fetch + PC update: */
pc = P_thePC; p_thePC = P_thePC + 2;
icode = i10bytes[4..8]; ifunc = i10bytes[0..4];
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode: */
reg_srcA = rA;
reg_srcB = rB;
/* Execute + Writeback: */
reg_inputE = reg_outputA + reg_outputB;
reg_dstE = rB;
/* Status maintainence: */
Stat = ...
```

example using register file: add CPU

```
wire rA : 4, rB : 4, icode : 4, ifunc: 4;
register pP {
    thePC : 64 = 0;
}
/* Fetch + PC update: */
pc = P_thePC; p_thePC = P_thePC + 2;
icode = i10bytes[4..8]; ifunc = i10bytes[0..4];
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode: */
reg_srcA = rA;
reg_srcB = rB;
/* Execute + Writeback: */
reg_inputE = reg_outputA + reg_outputB;
reg_dstE = rB;
/* Status maintainence: */
Stat = ...
```

register file picture

register file

reg_srcA

reg_srcB

reg_dstM

reg_dstE

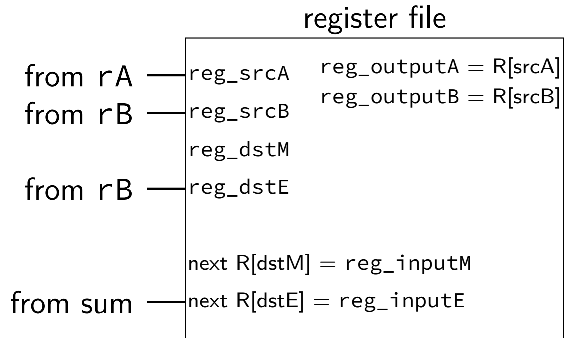
reg_outputA = R[srcA]

reg_outputB = R[srcB]

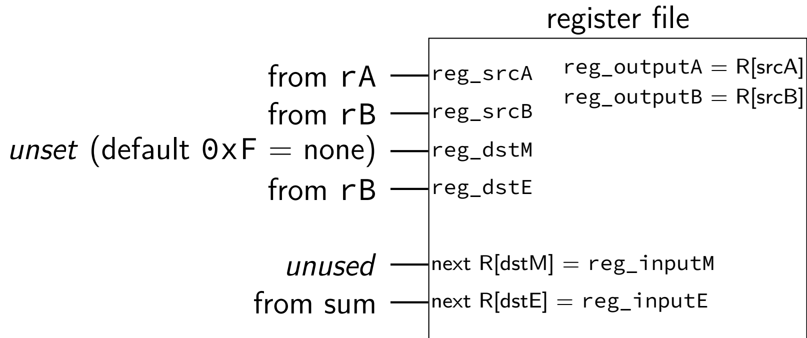
next R[dstM] = reg_inputM

next R[dstE] = reg_inputE

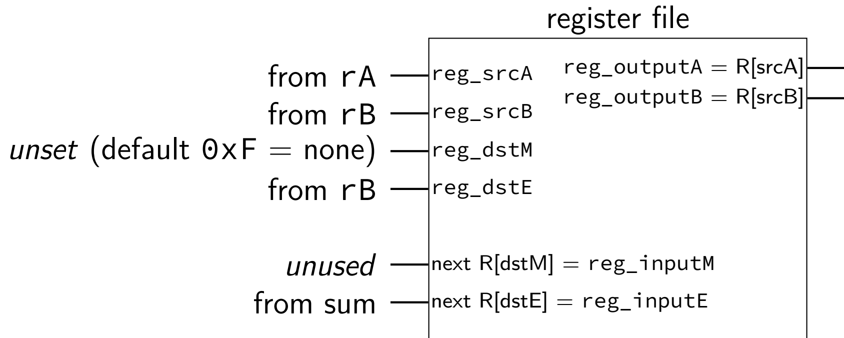
register file picture



register file picture



register file picture



things in HCLRS

register banks

wires

things for our processor:

- `Stat` register
- instruction memory
- the register file
- data memory

data memory

input address: `mem_addr`

input value: `mem_input`

output value: `mem_output`

read/write enable: `mem_readbit`, `mem_writebit`

reading from data memory

```
mem_addr = 0x12345678;
mem_readbit = 1;
mem_writebit = 0;
... = mem_output;
```

`mem_output` has value in same cycle

reading from data memory

```
mem_addr = 0x12345678;
mem_readbit = 1;
mem_writebit = 0;
... = mem_output;
```

`mem_output` has value in same cycle

reading from data memory

```
mem_addr = 0x12345678;
mem_readbit = 1;
mem_writebit = 0;
... = mem_output;
```

`mem_output` has value in same cycle

writing to data memory

```
mem_addr = 0x12345678;
mem_input = ...;
mem_readbit = 0;
mem_writebit = 1;
```

memory updated for next cycle

writing to data memory

```
mem_addr = 0x12345678;
mem_input = ...;
mem_readbit = 0;
mem_writebit = 1;
```

memory updated for next cycle

writing to data memory

```
mem_addr = 0x12345678;
mem_input = ...;
mem_readbit = 0;
mem_writebit = 1;
```

memory updated for next cycle

exercise: implementing ALU?

```
wire aluOp : 2,
     aluValueA : 64,
     aluValueB : 64,
     aluResult : 64;
const ALU_ADD = 0b00,
      ALU_SUB = 0b01,
      ALU_AND = 0b10,
      ALU_XOR = 0b11;
aluResult = [
    aluOp == ALU_ADD : aluValueA + aluValueB;
    aluOp == ALU_SUB : aluValueA - aluValueB;
    aluOp == ALU_AND : aluValueA & aluValueB;
    aluOp == ALU_XOR : aluValueA ^ aluValueB
];
```

on design choices

textbook choices:

- memory always goes to 'M' port of register file
- RSP +/- 8 uses normal ALU, not seperate adders
- ...

do you have to do this? no

you: single cycle/instruction; use supplied register/memory

other logic: make it function correctly

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)

addq with stages

```
wire rA : 4, rB : 4, icode : 4, ifunc: 4, valP : 64, valE : 64;
register pP { thePC : 64 = 0; }
/* Fetch: */
pc = P_thePC; valP = P_thePC + 2;
icode = i10bytes[4..8]; ifunc = i10bytes[0..4];
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode: */
reg_srcA = rA;
reg_srcB = rB;
/* Execute: */
valE = reg_outputA + reg_outputB;
/* Writeback: */
reg_inputE = valE; reg_dstE = rB;
/* PC update: */
p_thePC = valP;
...
```

addq with stages

```
wire rA : 4, rB : 4, icode : 4, ifunc: 4, valP : 64, valE : 64;
register pP { thePC : 64 = 0; }
/* Fetch: */
pc = P_thePC; valP = P_thePC + 2;
icode = i10bytes[4..8]; ifunc = i10bytes[0..4];
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode: */
reg_srcA = rA;
reg_srcB = rB;
/* Execute: */
valE = reg_outputA + reg_outputB;
/* Writeback: */
reg_inputE = valE; reg_dstE = rB;
/* PC update: */
p_thePC = valP;
...
```

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 $\leftarrow M_1[\mathrm{PC}]$ 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 $\leftarrow M_1[\text{PC}]$ 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

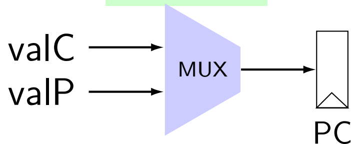
$\leftarrow$ means putting value on input wire to PC register

stages example: nop/jmp

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

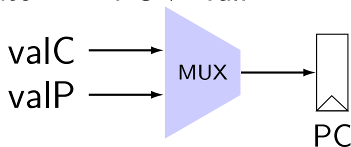
stages example: nop/jmp

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

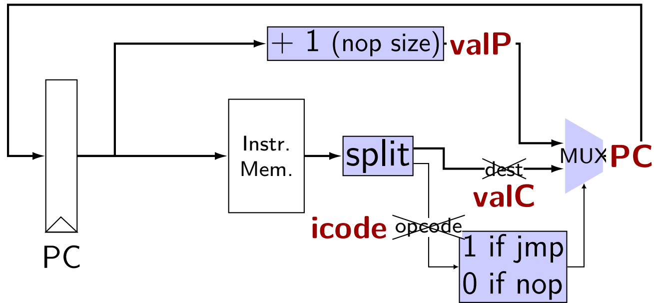


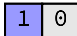
stages example: nop/jmp

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



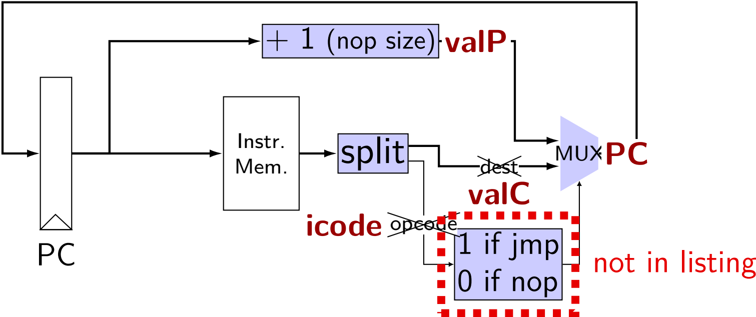
jmp+nop CPU



`nop` 

`jmp` *Dest*

jmp+nop CPU



`nop`	1	0	
`jmp` *Dest*	7	0	*Dest*

stages example: rmmovq/mrmovq

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

stages example: rmmovq/mrmovq

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

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

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 $\leftarrow$	$+$ 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 $\leftarrow M_1[\text{PC}]$ valP $\leftarrow$ PC $+ 10$ valC $\leftarrow M_8[\text{PC} + 2]$	icode : ifun $\leftarrow M_1[\text{PC}]$ valP $\leftarrow$ PC $+ 10$ valC $\leftarrow M_8[\text{PC} + 2]$
decode	valA $\leftarrow R[\text{rA}]$ valB $\leftarrow R[\text{rB}]$	valB $\leftarrow R[\text{rB}]$
execute	valE $\leftarrow$ valB $+$ valC	valE $\leftarrow$ valB $+$ valC
memory	$M_8[\text{valE}] \leftarrow$ valA	valM $\leftarrow M_8[\text{valE}]$
		$R[\text{rA}] \leftarrow$ valM
		C $\leftarrow$ valP

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

stages example: rmmovq/mrmovq

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

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

mov CPU

immediate valC

register file

srcA R[srcA]

srcB R[srcB]

dstE

next R[dstE]

valA

valB

PC

valP

+2

+10

Instr. Mem.

split

0xF

convert opcode

MUX

valM

valE

Data in

Data out

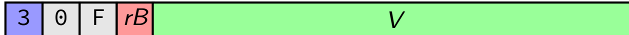
Addr in

write enable

from convert opcode

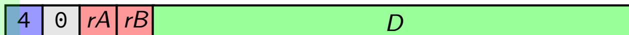
rrmovq *rA*, *rB*

irmovq *V*, *rB*



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

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



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*)

mov CPU

immediate

register file

srcA R[srcA]

srcB R[srcB]

dstE

next R[dstE]

PC

Instr. Mem.

split

0xF

convert opcode

MUX

+2

+10

Data in

Data out

Addr in

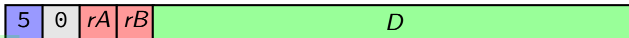
write enable

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.

SEQ: instruction fetch

read instruction memory at PC

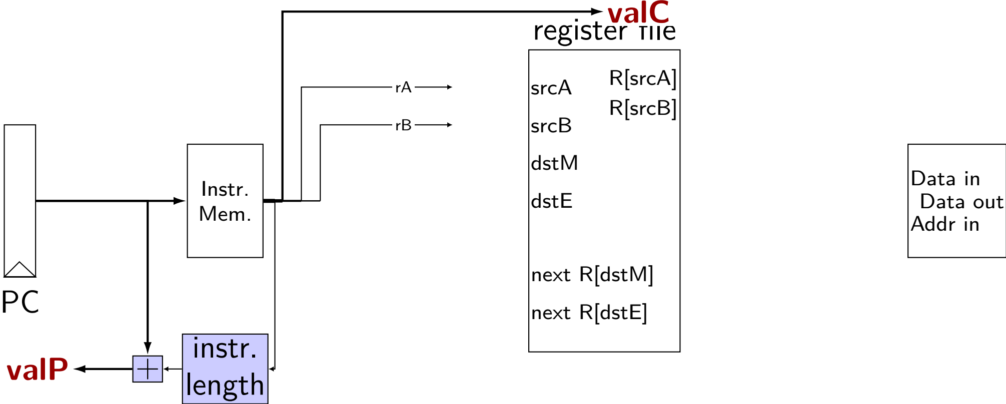
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

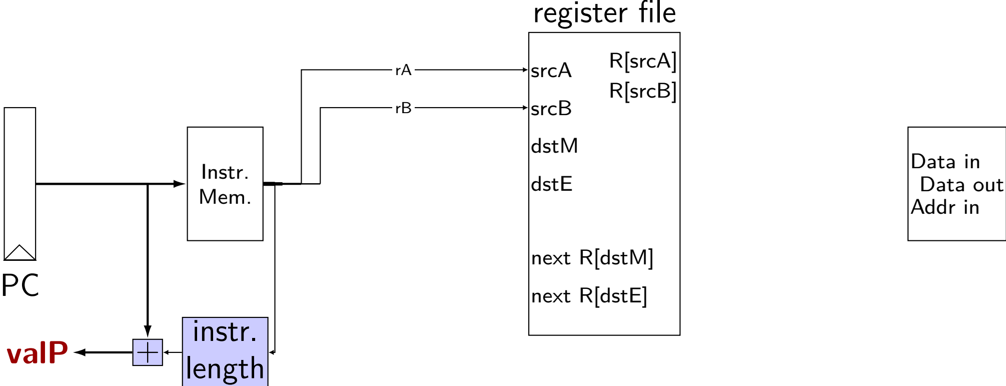


SEQ: instruction "decode"

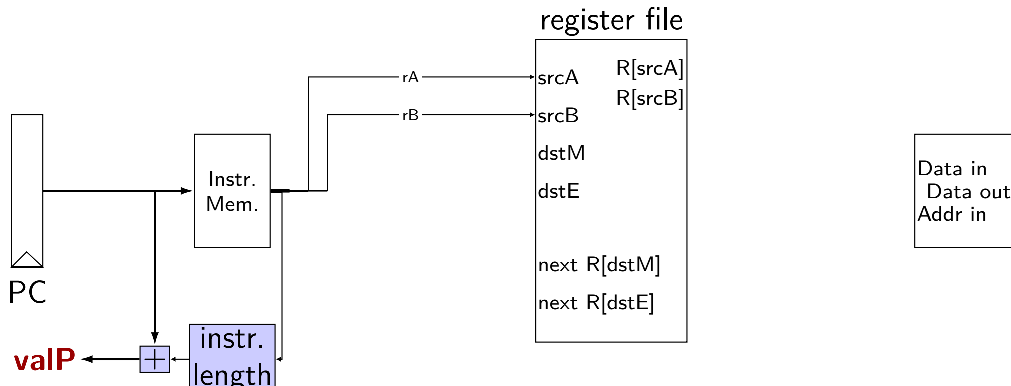
read registers

- valA, valB — register values

instruction decode (1)



instruction decode (1)



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

`nop`, `addq`, `mrmovq`, `rmmovq`, `jmp`,

SEQ: srcA, srcB

always read rA, rB?

Problems: (not planned to be included our assignments)

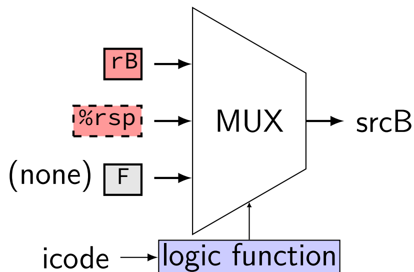
- push rA
- pop
- call
- ret

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

MUX controlled by icode

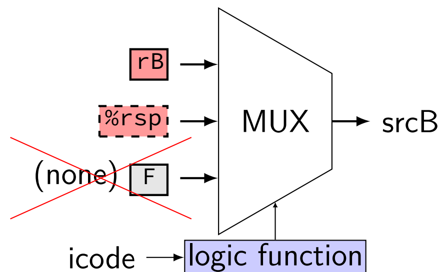
SEQ: possible registers to read

instruction	srcA	srcB
`halt`, `nop`, `j`*CC*, `irmovq`	none	none
`cmovCC`, `rrmovq`	rA	none
`mrmovq`	none	rB
`rmmovq`, *OP*`q`	rA	rB
`call`, `ret`	none?	%rsp
`pushq`, `popq`	rA	%rsp

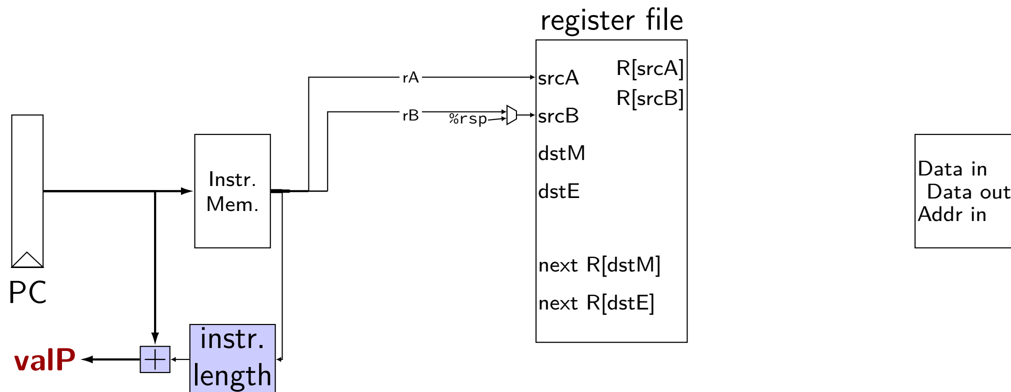


SEQ: possible registers to read

instruction	srcA	srcB
`halt`, `nop`, `j`*CC*, `irmovq`	none	none
`cmovCC`, `rrmovq`	rA	none
`mrmovq`	none	rB
`rmmovq`, *OP*`q`	rA	rB
`call`, `ret`	none?	`%rsp`
`pushq`, `popq`	rA	`%rsp`



instruction decode (2)



SEQ: execute

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

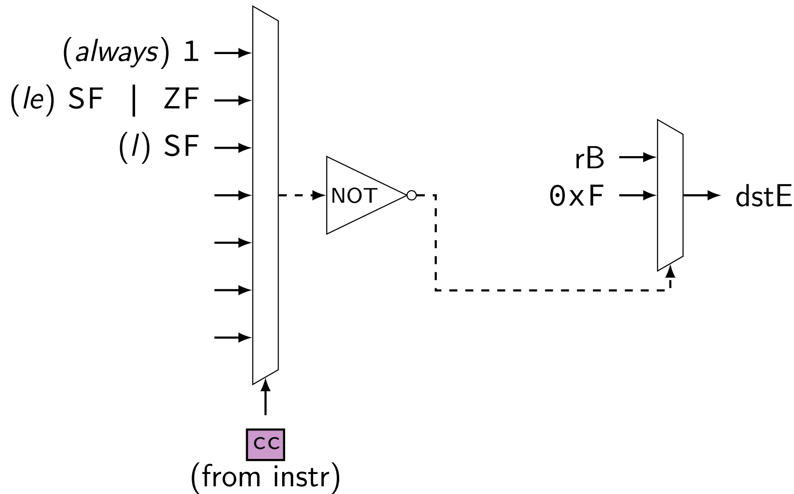
- valE — ALU output

read prior condition codes

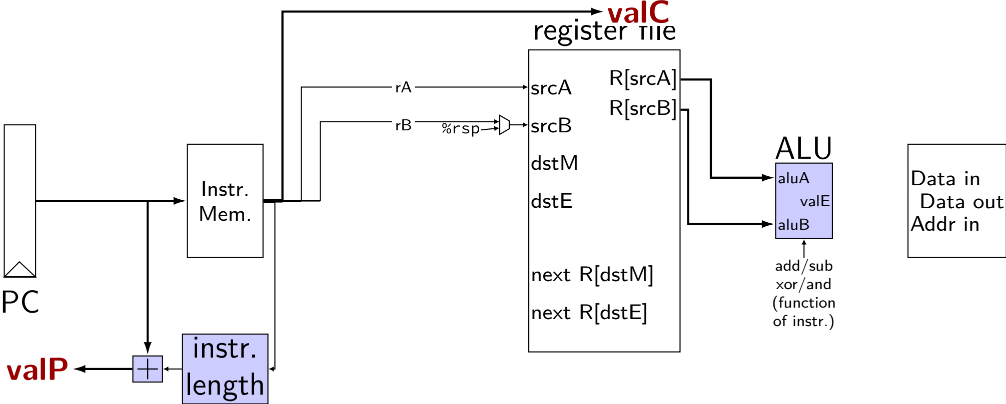
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

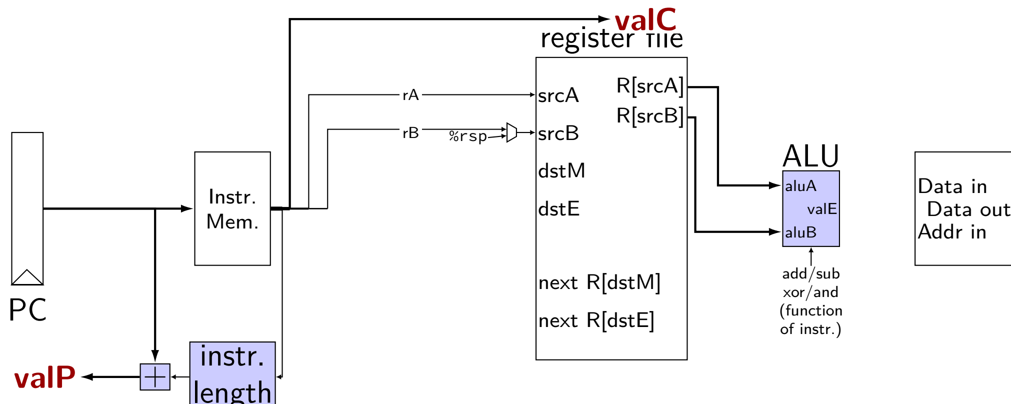
using condition codes: cmov



execute (1)



execute (1)



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

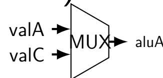
`nop`, `addq`, `mrmovq`, `popq`, `call`,

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

`mrmovq`
`rmmovq`



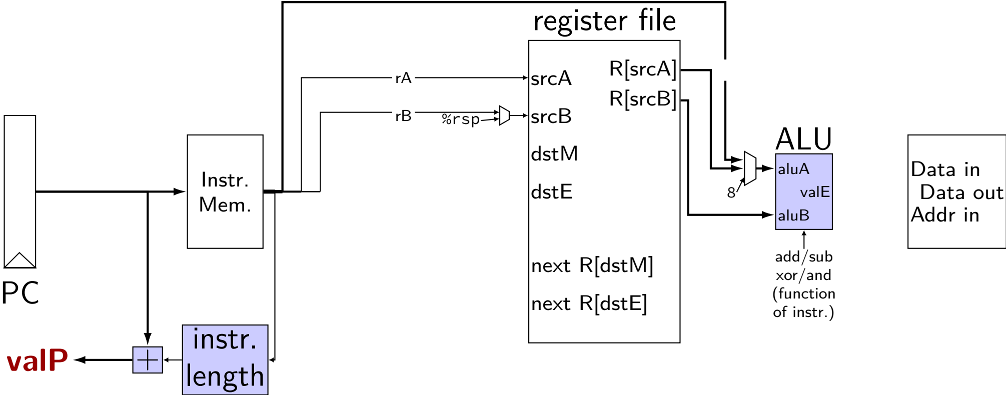
no, constants: (rsp +/- 8) (not planned to be in assignments)

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

extra signals: aluA, aluB

computed ALU input values

execute (2)

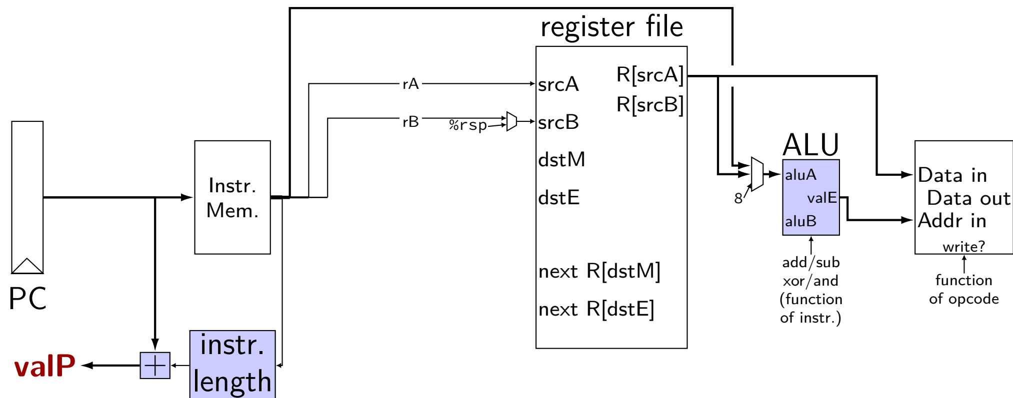


SEQ: Memory

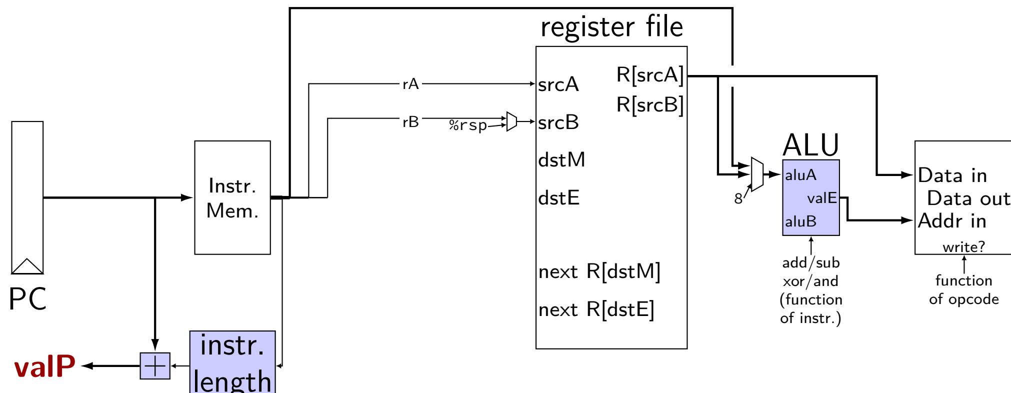
read or write data memory

- valM — value read from memory (if any)

memory (1)



memory (1)



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

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

SEQ: control signals for memory

read/write — read enable? write enable?

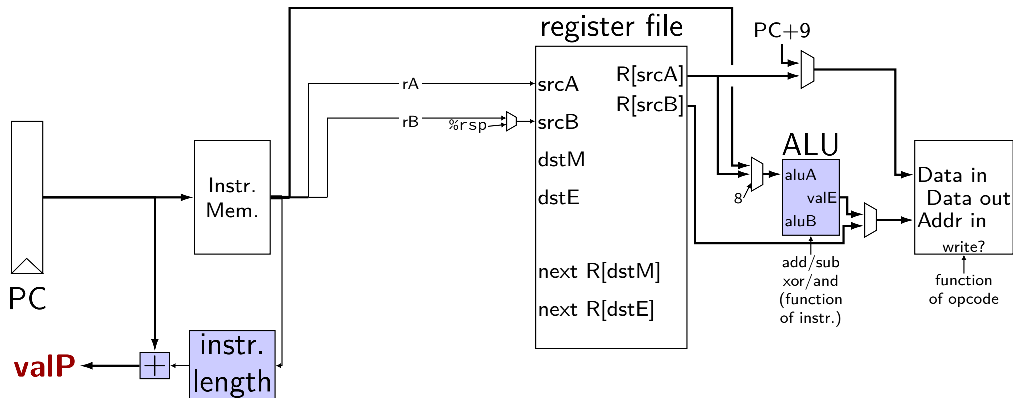
Addr — address

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

Data — value to write

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

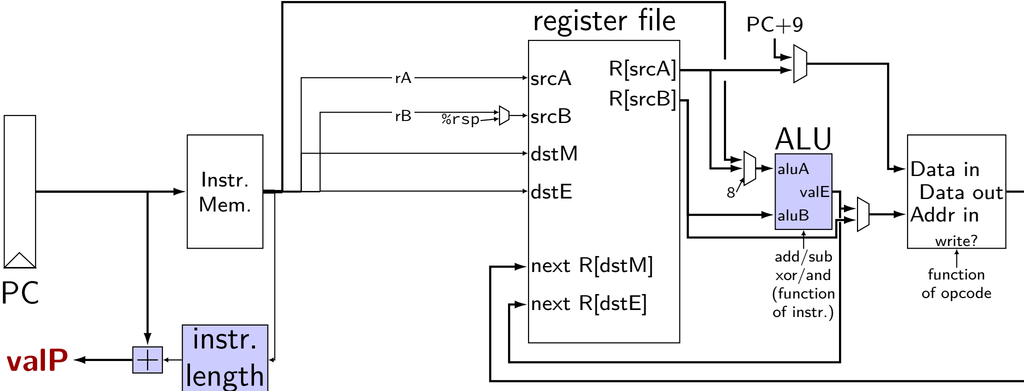
memory (2)



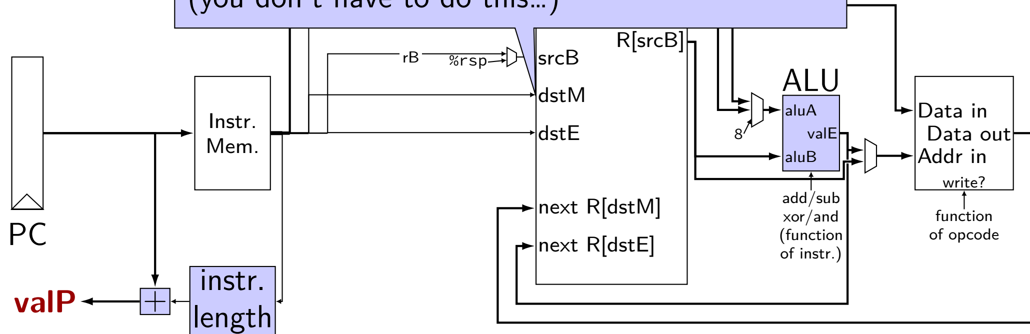
SEQ: write back

write registers

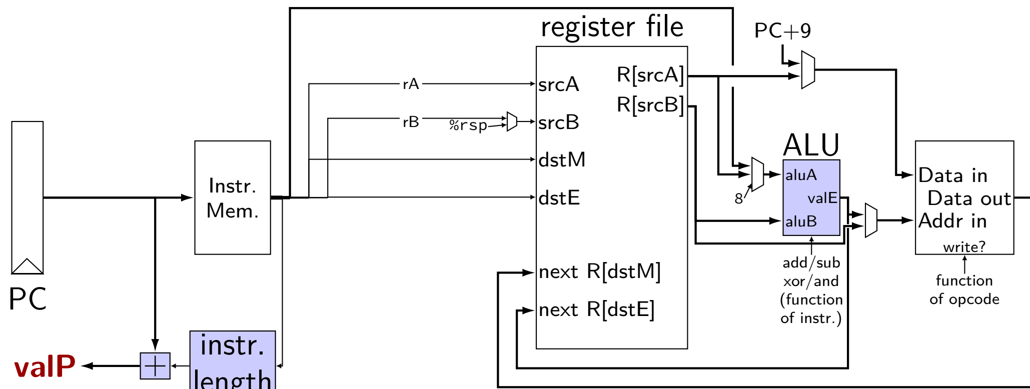
write back (1)



write back (1)



write back (1)



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

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

SEQ: control signals for WB

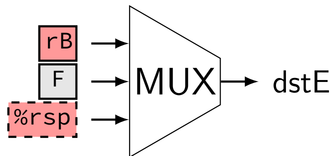
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

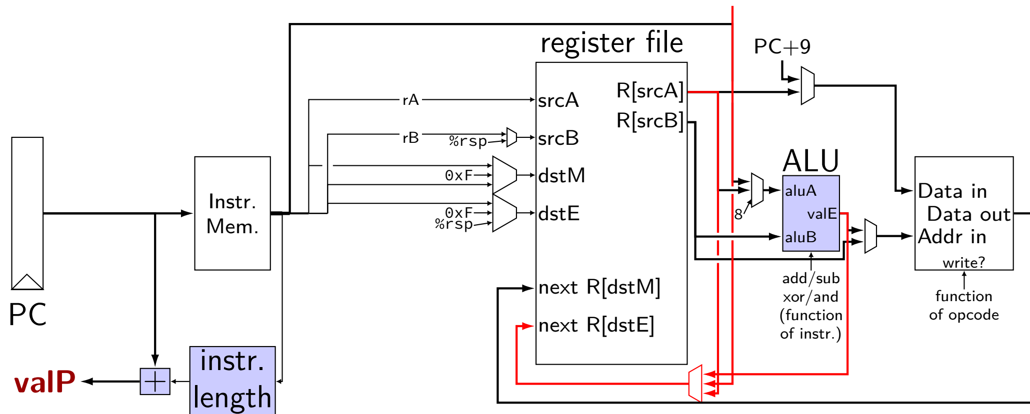
two register numbers

dstM, dstE

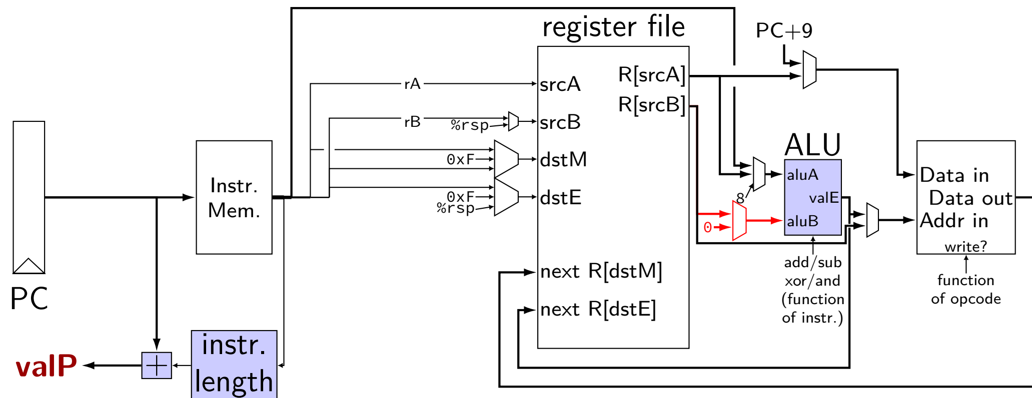
write disable — use dummy register number `0xF`



write back (2a)



write back (2b)

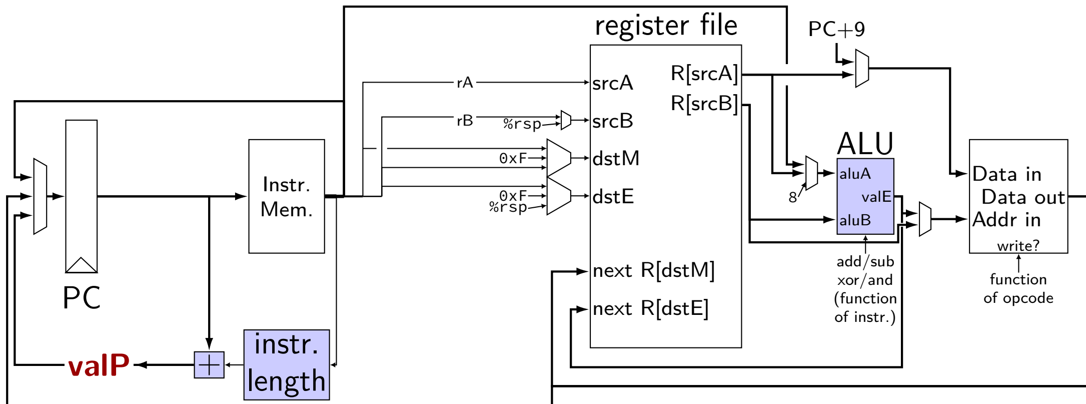


SEQ: Update PC

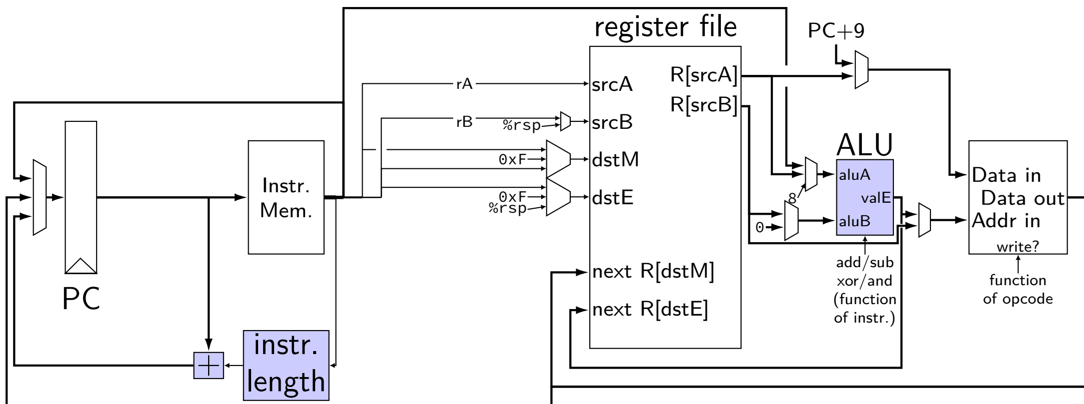
choose value for PC next cycle (input to PC register)

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

PC update

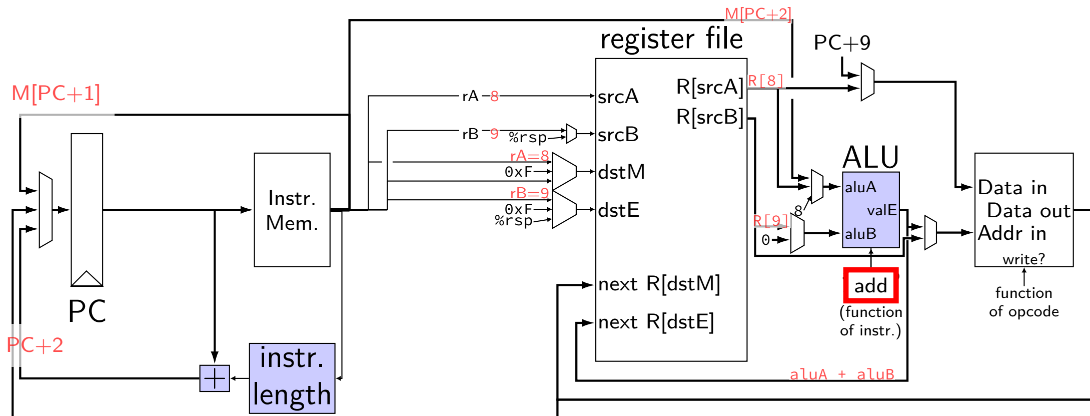


circuit: setting MUXes



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

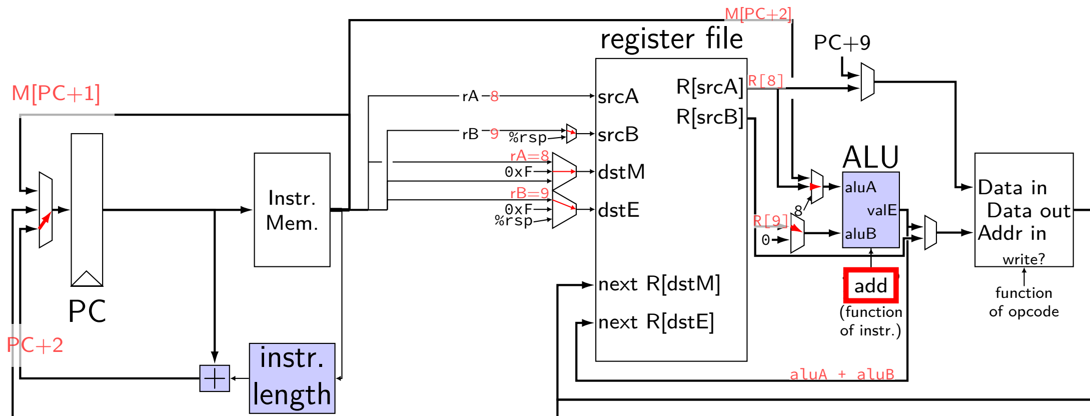
circuit: setting MUXes



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

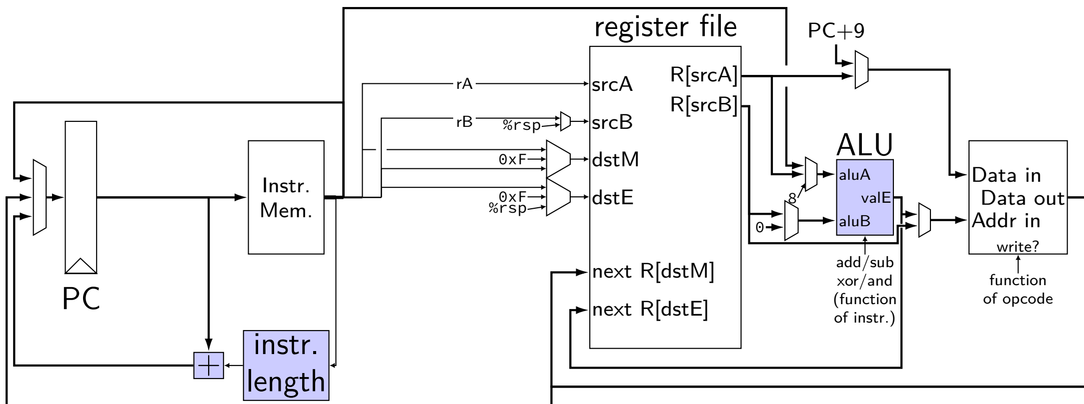
Exercise: what do they select when running `addq %r8, %r9`?

circuit: setting MUXes



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

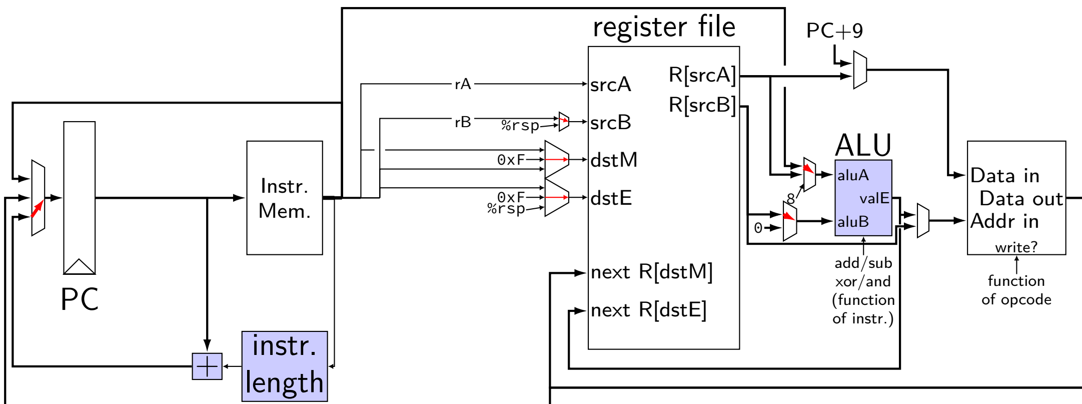
circuit: setting MUXes



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

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

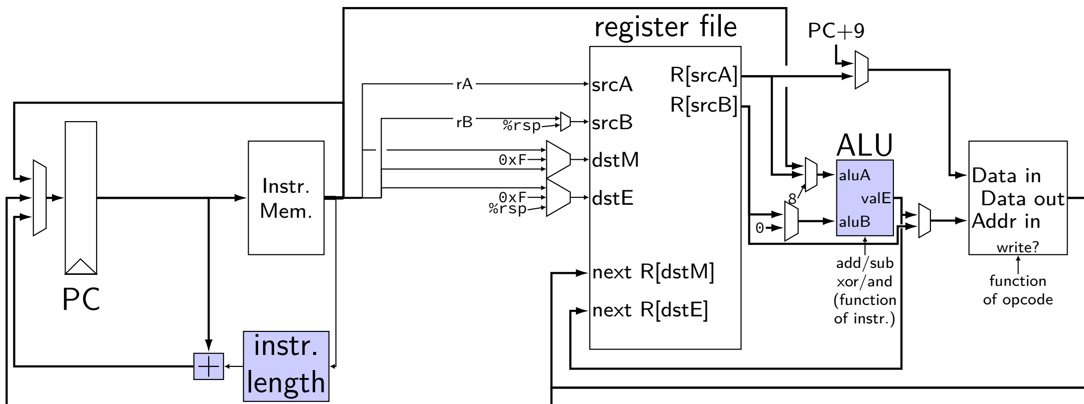
circuit: setting MUXes



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

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

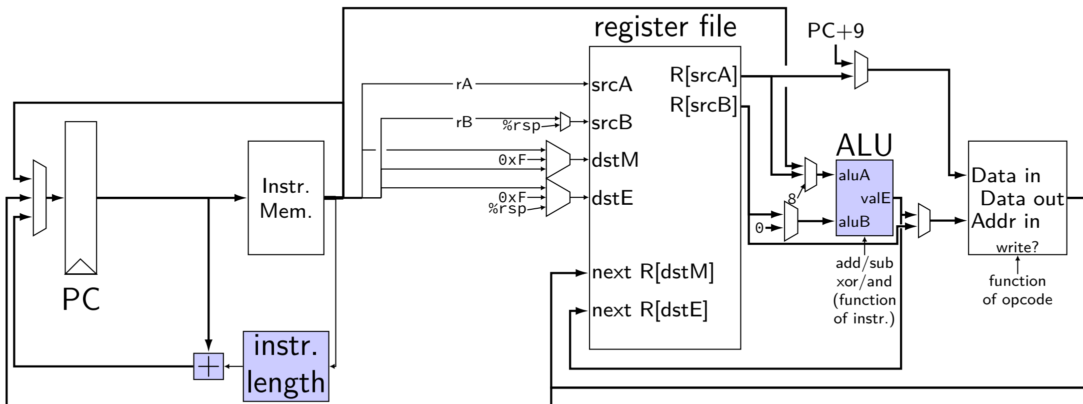
circuit: setting MUXes



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

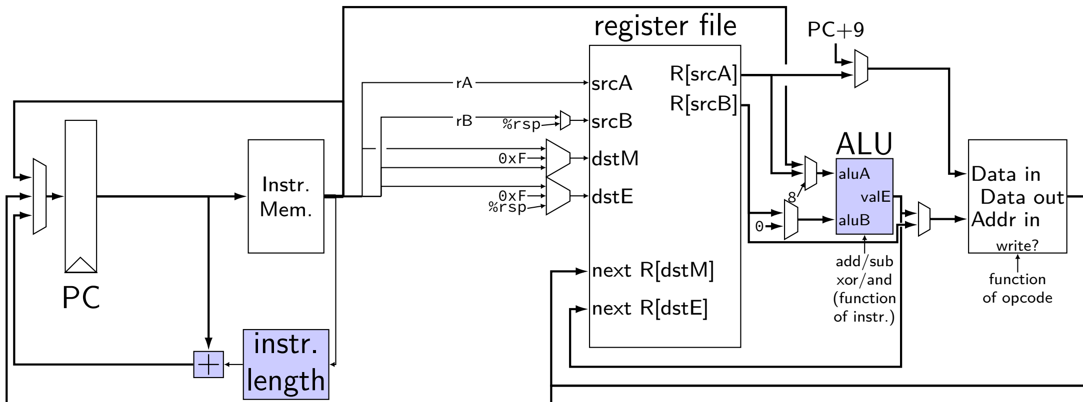
Exercise: what do they select for `irmovq`?

circuit: setting MUXes



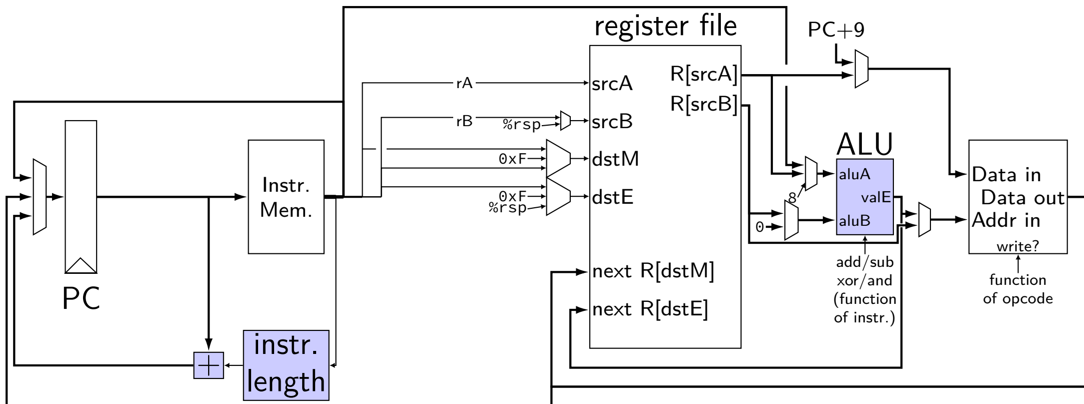
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `mrmovq`?

circuit: setting MUXes



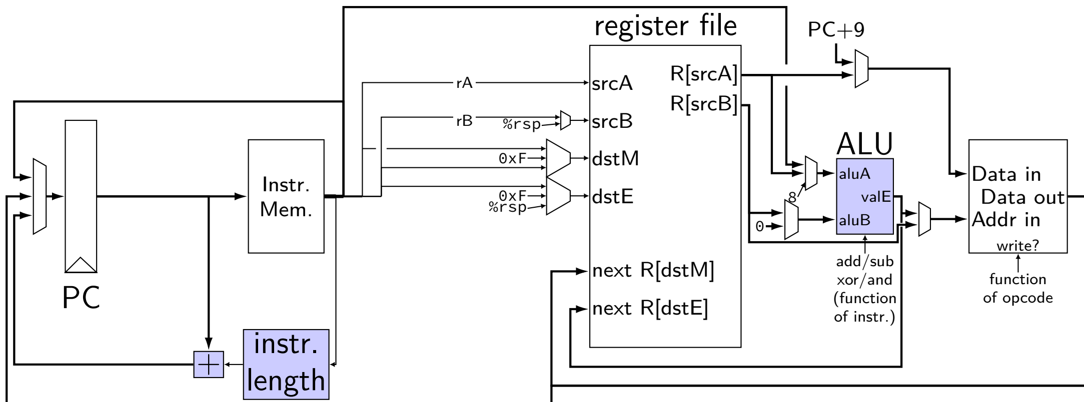
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `jle`?

circuit: setting MUXes



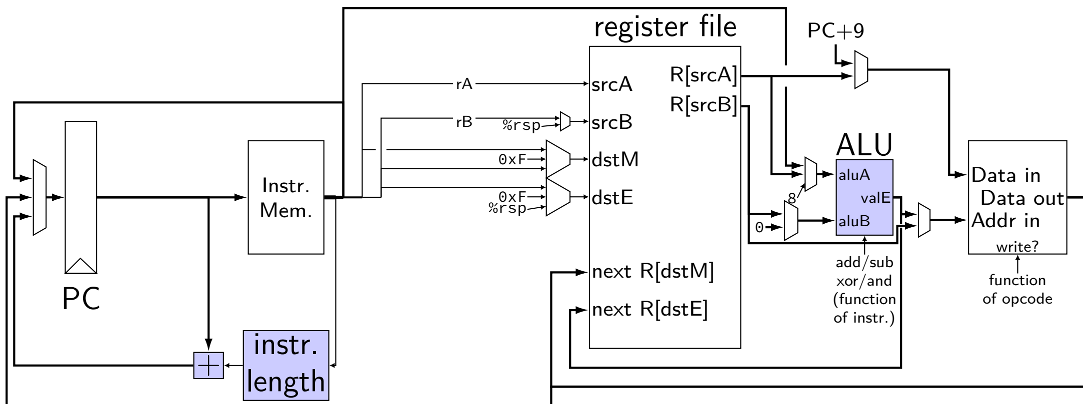
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, …
Exercise: what do they select for `cmovle`?

circuit: setting MUXes



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

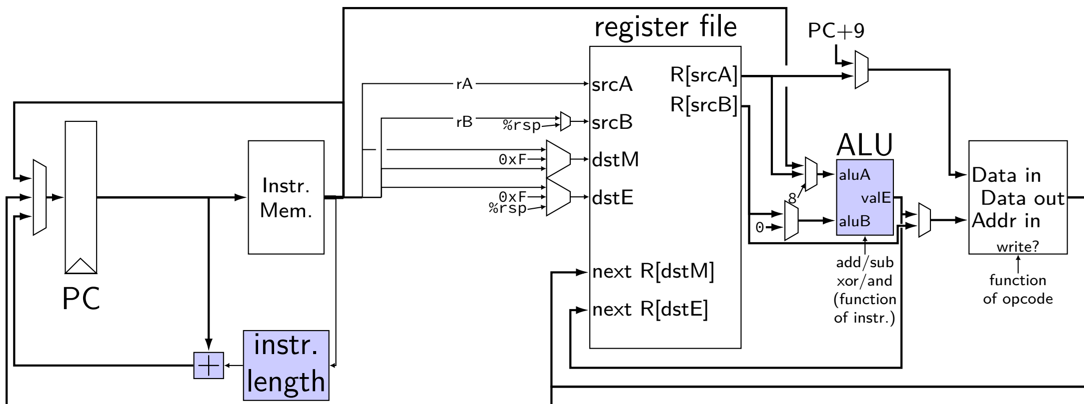
circuit: setting MUXes



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`?