last time

HCLRS syntax

- = represents connection
- declaring wire bundles
- C expressions represent logic

instability with loops

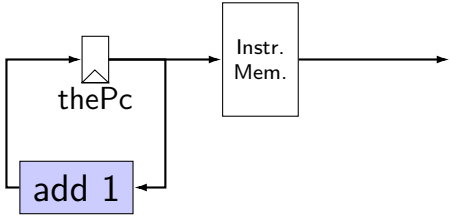
registers and the clock signal

- output stored value
- read in new value on rising edge of clock signal

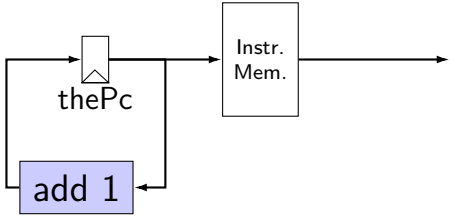
instruction memory

nop CPU

nop CPU

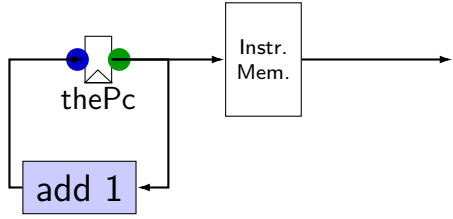


nop CPU



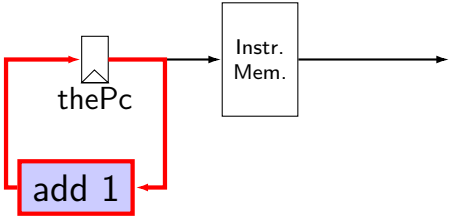

```
register pF {
    thePc : 64 = 0;
}
```

nop CPU



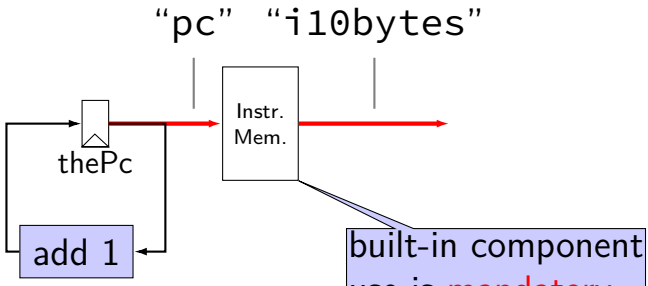

```
register pF {
    thePc : 64 = 0;
}
```

nop CPU



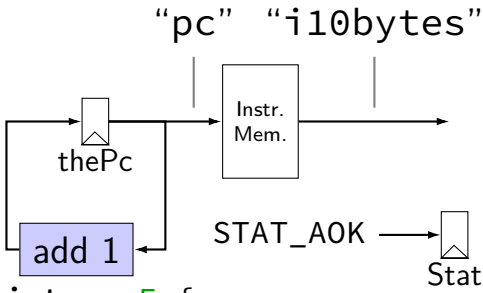

```
register pF {
    thePc : 64 = 0;
}
p_thePc = F_thePc + 1;
```

nop CPU



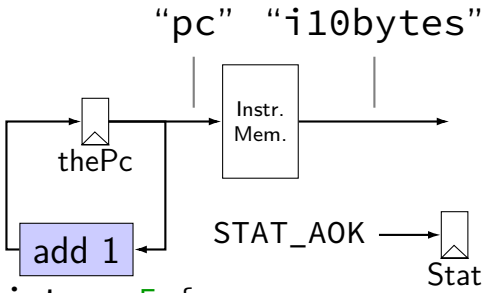

```
register pF {
   thePc : 64 = 0;
}
p_thePc = F_thePc + 1;
pc = F_thePc;
```

nop CPU




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

nop CPU




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

nop CPU: running

need a program in memory

- .yo file

`tools/yas` — convert `.ys` to `.yo`

`tools/yis` — reference interpreter for `.yo` files

- if your processor doesn't do the same thing...

can build tools by running `make`

nop CPU: creating a program

create assemby file: nops.ys:

```
nop
nop
nop
nop
nop
```

assemble using `tools/yas nops.ys` or `make nops.yo`

nop.yo

more readable/simpler than normal executables:

```
0x000: 10                                | nop
0x001: 10                                | nop
0x002: 10                                | nop
0x003: 10                                | nop
0x004: 10                                | nop
                                         |
```

loaded into data and program memory

parts left of | just comments

running a simulator (1)

```
Usage: ./hclrs [options] HCL-FILE [YO-FILE [TIMEOUT]]
Runs HCL_FILE on YO-FILE. If --check is specified, no YO-FILE may be supplied.
Default timeout is 9999 cycles.

Options:
    -c, --check          check syntax only
    -d, --debug          output wire values after each cycle and other debug
                         output
    -q, --quiet          only output state at the end
    -t, --testing        do not output custom register banks (for autograding)
    -h, --help           print this help menu
    -i, --interactive    prompt after each cycle
        --trace-assignments
                         show assignments in the order they are simulated
        --version        print version number
```

running a simulator (2)

```
$ ./hclrs nop_cpu.hcl nops.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 pF(N)  thePc=0000000000000000                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
....
+------------ timed out after  9999 cycles 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 pF(N)  thePc=000000000000270f                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
```

running a simulator (2)

```
$ ./hclrs nop_cpu.hcl nops.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 pF(N)  thePc=0000000000000000                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
....
+------------ timed out after  9999 cycles 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 pF(N)  thePc=000000000000270f                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
```

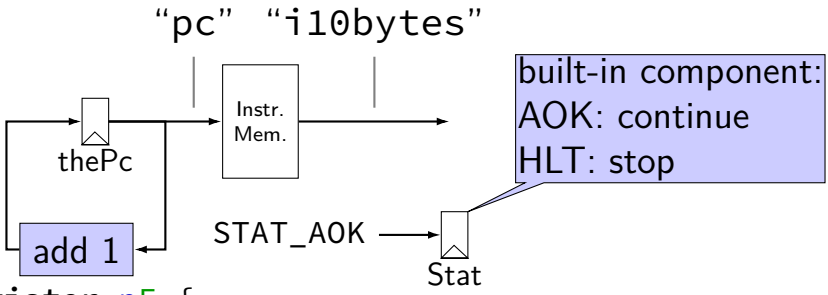
running a simulator (2)

```
$ ./hclrs nop_cpu.hcl nops.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 pF(N)  thePc=0000000000000000                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
....
+------------ timed out after  9999 cycles 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 pF(N)  thePc=000000000000270f                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
```

running a simulator (2)

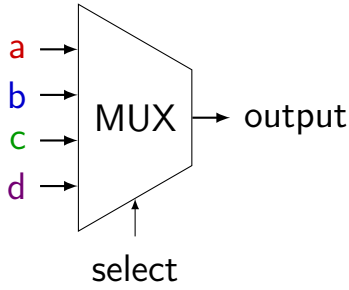
```
$ ./hclrs nop_cpu.hcl nops.yo
+------------------- between cycles    0 and    1 ----------------------+
| RAX:                0   RCX:                0   RDX:                0 |
| RBX:                0   RSP:                0   RBP:                0 |
| RSI:                0   RDI:                0   R8:                 0 |
| R9:                 0   R10:                0   R11:                0 |
| R12:                0   R13:                0   R14:                0 |
| register pF(N)  thePc=0000000000000000                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
....
+------------ timed out after  9999 cycles 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 pF(N)  thePc=000000000000270f                                |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 10 10 10  10                                        |
+-----------------------------------------------------------------------+
```

nop CPU

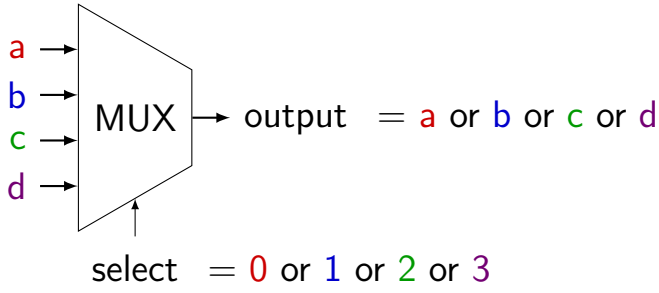



```
register pF {
   thePc : 64 = 0;
}
p_thePc = F_thePc + 1;
pc = F_thePc;
Stat = STAT_AOK;
```

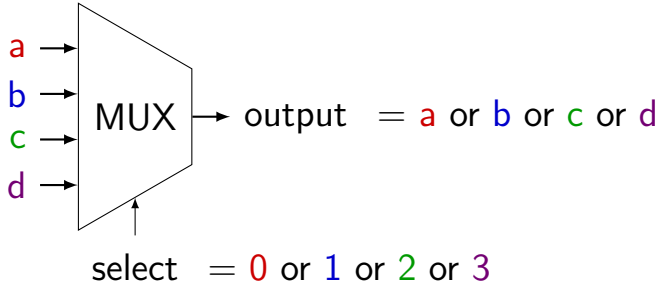
multiplexers



multiplexers



multiplexers



truth table:

select bit 1	select bit 0	output (many bits)
0	0	a
0	1	b
1	0	c
1	1	d

MUXes in HCLRS

book calls “case expression”

conditions evaluated (as if) in order

first match is output: `result = [`

```
    x == 5: 1;
    x in {0, 6}: 2;
    x > 2: 3;
    1: 4;
];
```

$x = 5$: result is 1
$x = 6$: result is 2
$x = 3$: result is 3
$x = 4$: result is 3
$x = 1$: result is 4

MUX exercise

```
foo = [
    bar > 10 : 100;
    (bar & 1) == 1 : 200;
    bar < 20 : 300;
    1 : 400;
]
```

exercise 1: if `bar` is 9, what is foo?

exercise 2: if `bar` is 10, what is foo?

exercise 3: if `bar` is 11, what is foo?

Simple ISA: nop/halt CPU

nop

encoding `10`

halt

encoding `00`

Simple ISA: nop/halt CPU

nop

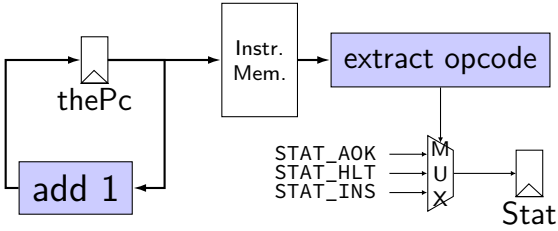
encoding `10`

halt

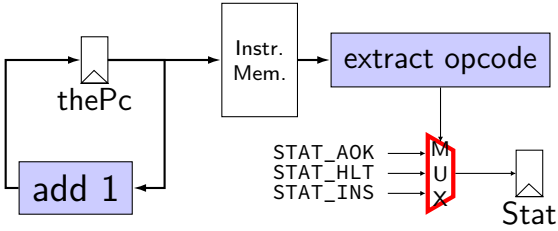
encoding `00`

our strategy: MUX to decide using opcode

nop/halt CPU

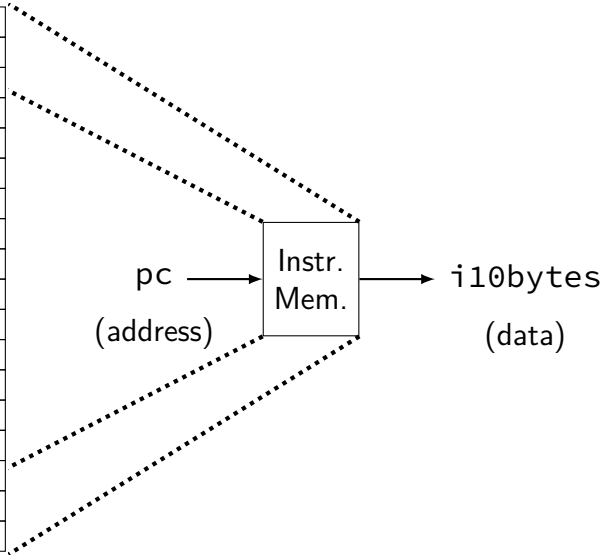

thePc
Instr.
Mem.
extract opcode
STAT_AOK
STAT_HLT
STAT_INS
M
U
X
Stat
add 1

nop/halt CPU


thePc
add 1
Instr.
Mem.
extract opcode
STAT_AOK
STAT_HLT
STAT_INS
M
U
X
Stat

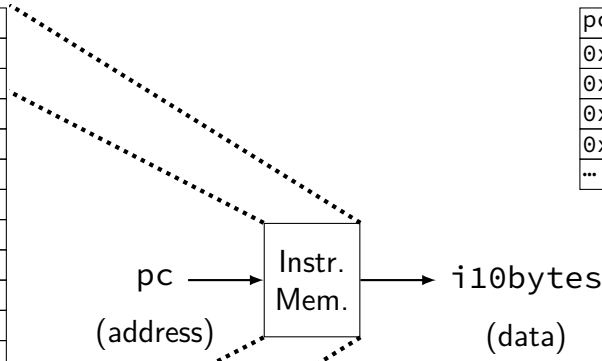
what is i10bytes?

addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



what is i10bytes?

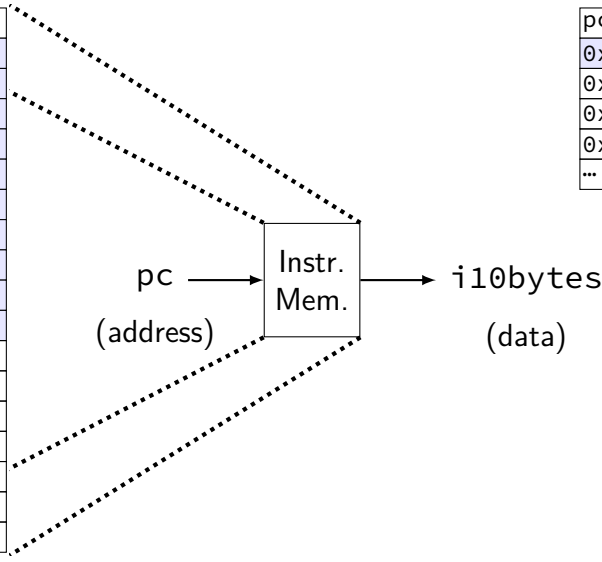
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x000000000021611260
0x001	0x010000000000216112
0x002	0x000100000000002161
0x003	0x000001000000000021
…	…

what is i10bytes?

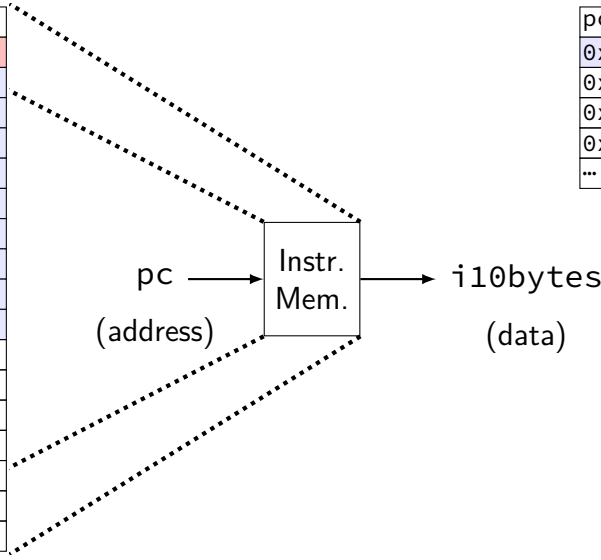
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x000000000021611260
0x001	0x010000000000216112
0x002	0x000100000000002161
0x003	0x000001000000000021
…	…

what is i10bytes?

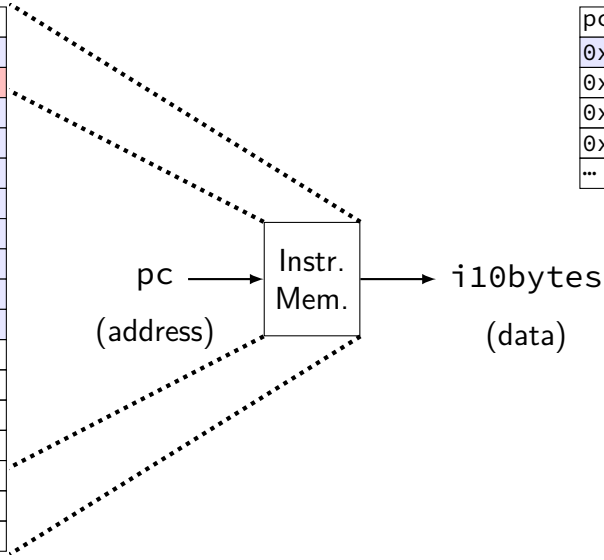
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x0000000000216112**60**
0x001	0x0100000000000216112
0x002	0x0001000000000002161
0x003	0x0000010000000000021
…	…

what is i10bytes?

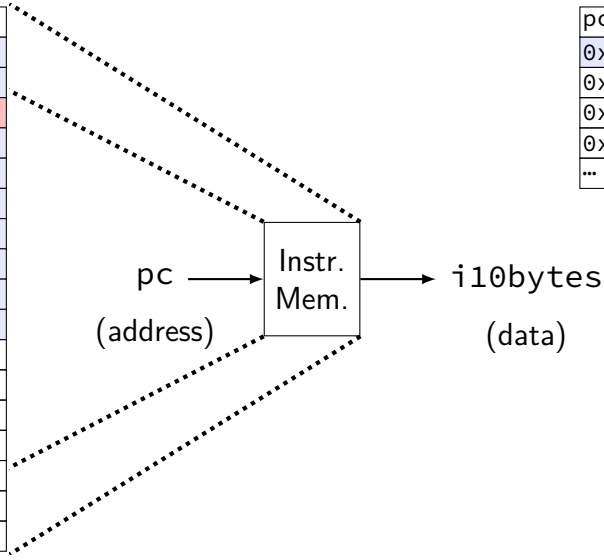
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x0000000000216112 60
0x001	0x0100000000000216112
0x002	0x0001000000000002161
0x003	0x0000010000000000021
…	…

what is i10bytes?

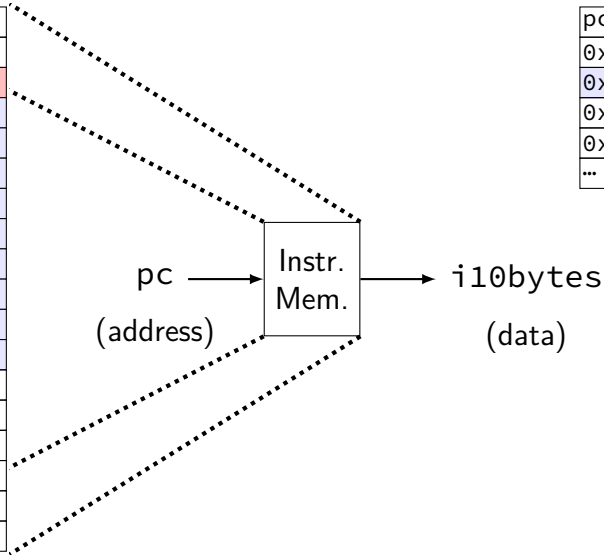
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x0000000000021611260
0x001	0x010000000000216112
0x002	0x000100000000002161
0x003	0x000001000000000021
…	…

what is i10bytes?

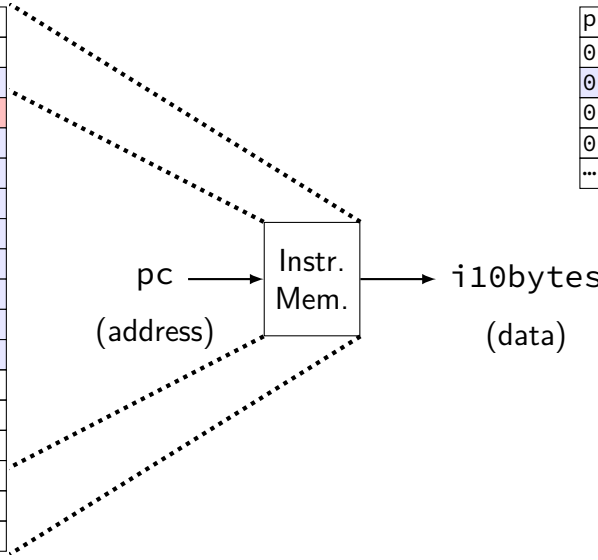
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x000000000021611260
0x001	0x010000000000216112
0x002	0x000100000000002161
0x003	0x000001000000000021
…	…

what is i10bytes?

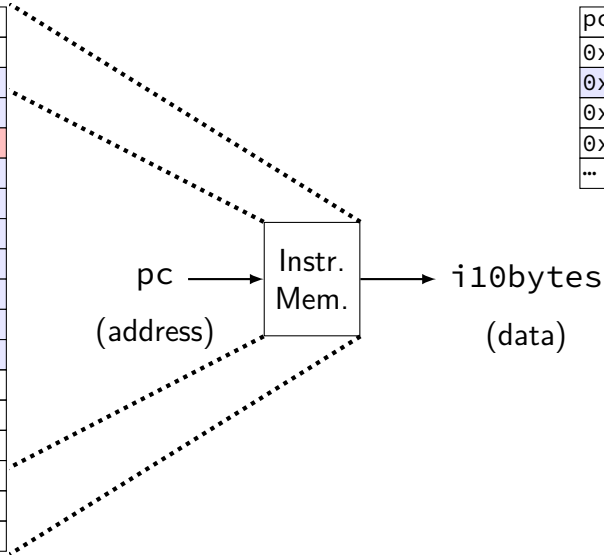
addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x000000000021611260
0x001	0x010000000000216112
0x002	0x000100000000002161
0x003	0x000001000000000021
…	…

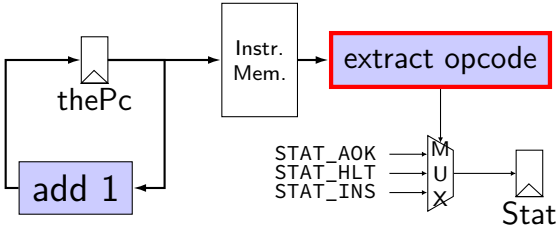
what is i10bytes?

addr.	value
0x000	0x60
0x001	0x12
0x002	0x61
0x003	0x21
0x004	0x00
0x005	0x00
0x006	0x00
0x007	0x00
0x008	0x00
0x009	0x00
0x00a	0x01
0x00b	0x00
0x00c	0x00
0x00d	0x00
0x00e	0x00
0x00f	0x00
…	…



pc	i10bytes
0x000	0x000000000021611260
0x001	0x0100000000000216112
0x002	0x0001000000000002161
0x003	0x0000010000000000021
…	…

nop/halt CPU


thePc
Instr.
Mem.
extract opcode
STAT_AOK
STAT_HLT
STAT_INS
M
U
X
Stat
add 1

subsetting bits in HCLRS

extracting bits 2 (inclusive)–9 (exclusive): `value[2..9]`

least significant bit is bit 0

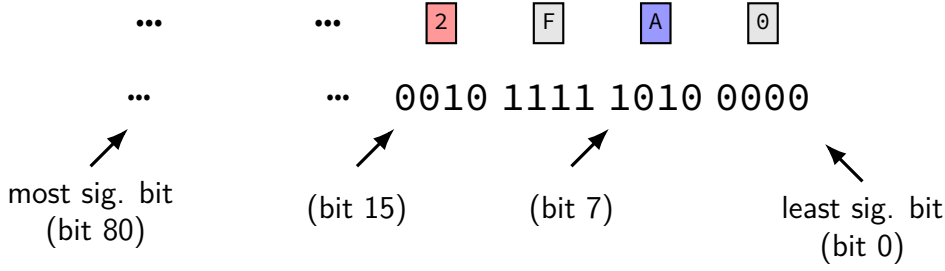
i10bytes example

`pushq %rbx` at memory address x: A 0 2 F

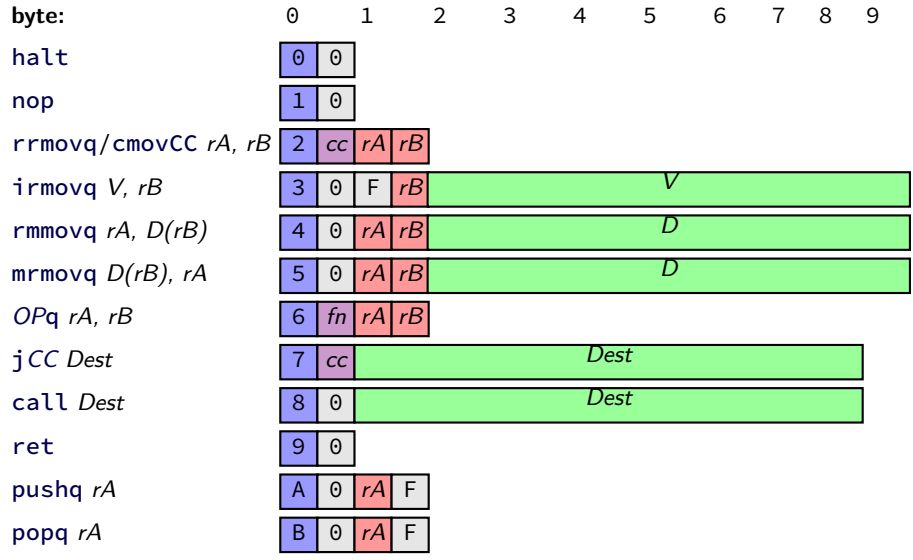
memory at $x + 0$: pushq F; at $x + 1$: rbx F

$x + 0$: A F; at $x + 1$: 2 F; at $x + 2$: (next instruction)

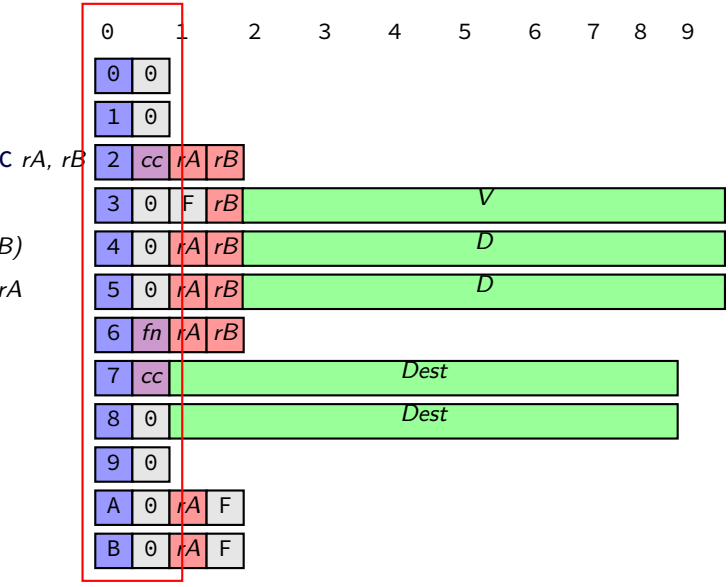
10-byte instruction memory output:



Y86 encoding table

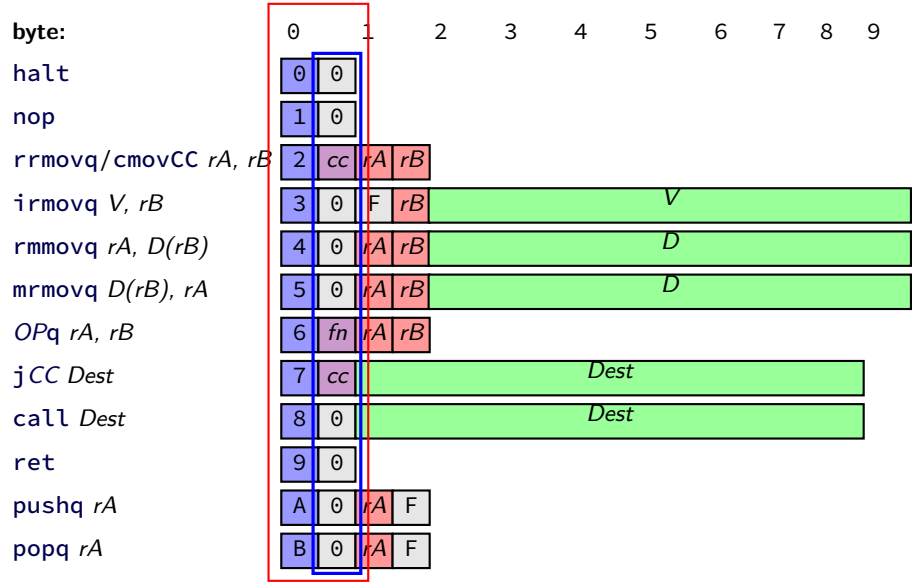


Y86 encoding table



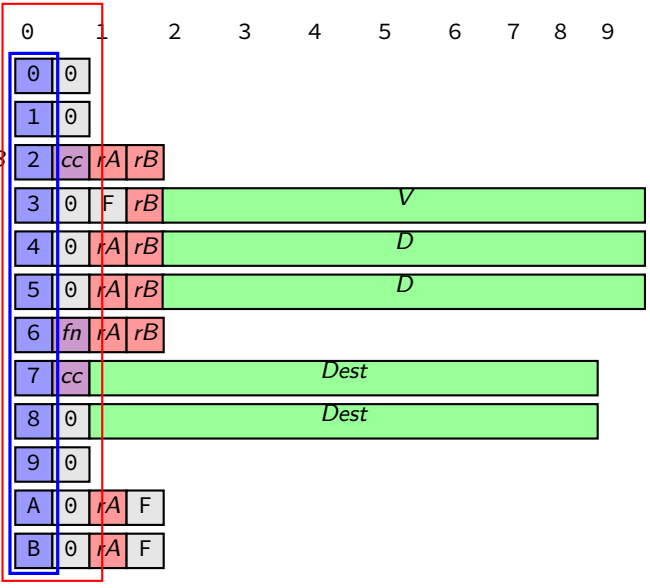
byte 0: bits 0–7

Y86 encoding table



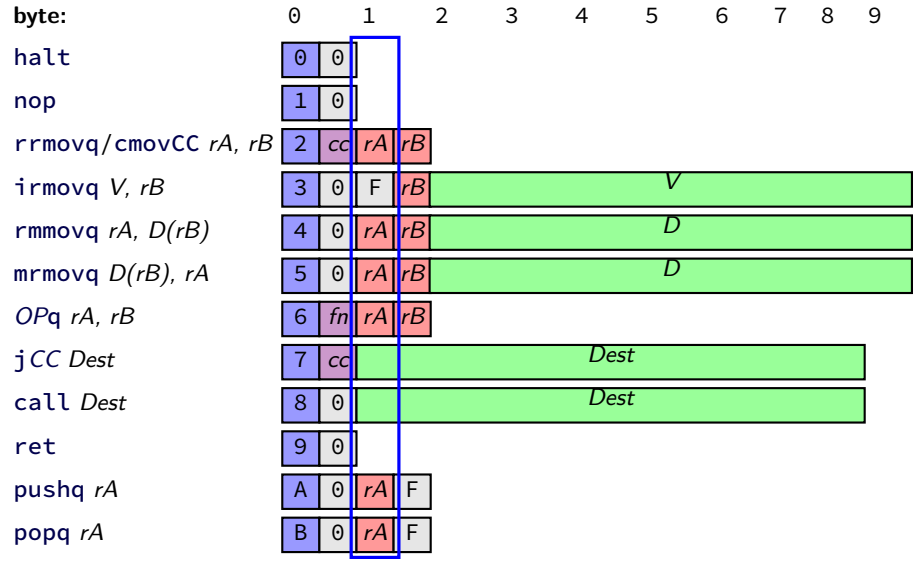
least sig. 4 bits of byte 0: bits 0–4

Y86 encoding table



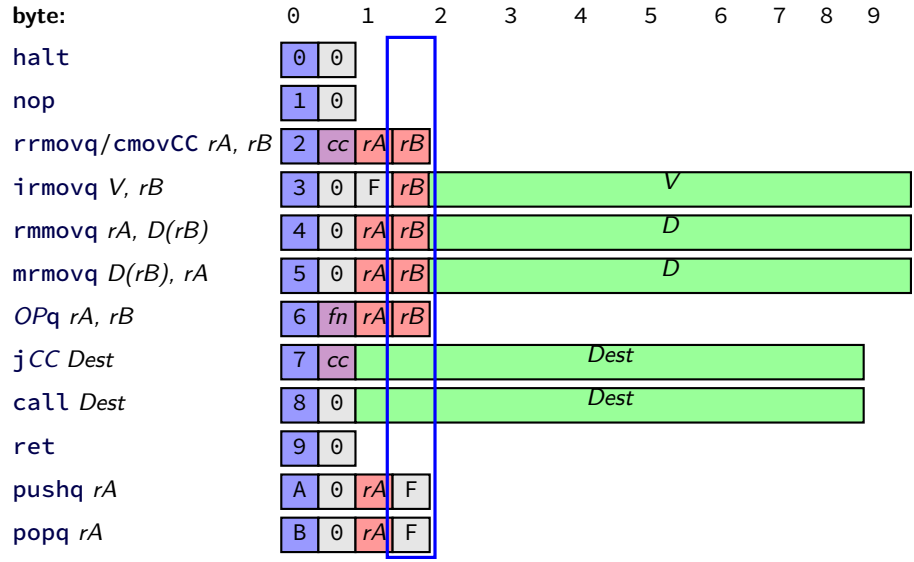
most sig. 4 bits of byte 0: bits 4–8

Y86 encoding table



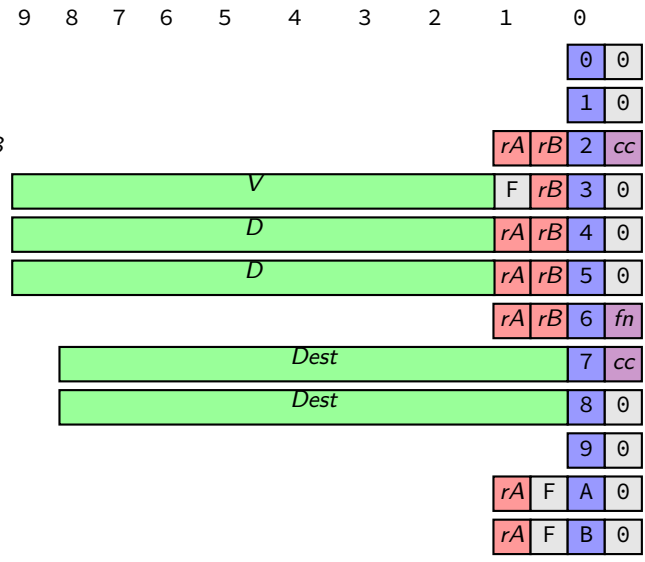
most sig. 4 bits of byte 1: bits 12–16

Y86 encoding table

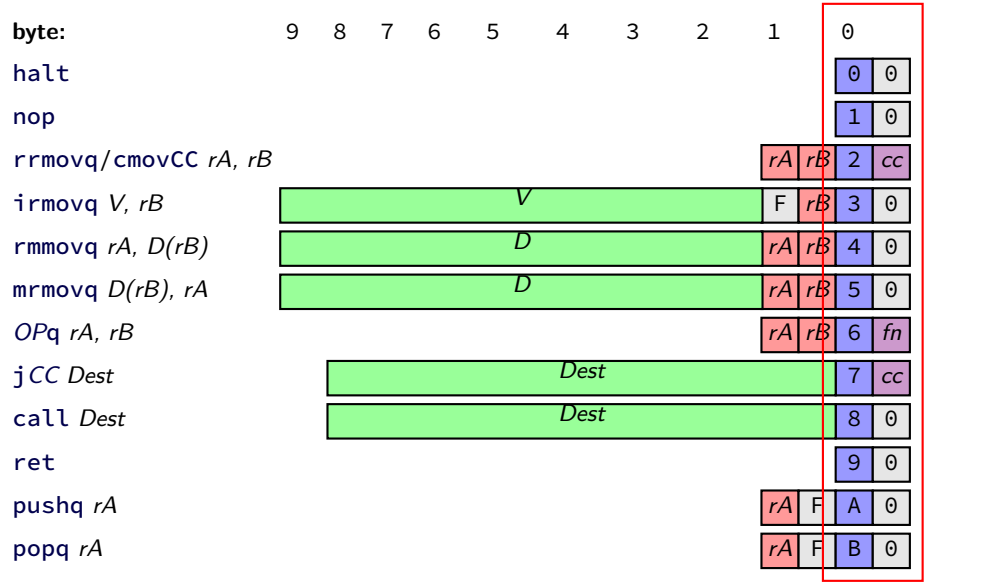


least sig. 4 bits of byte 1: bits 8–12

Y86 encoding table (written differently)

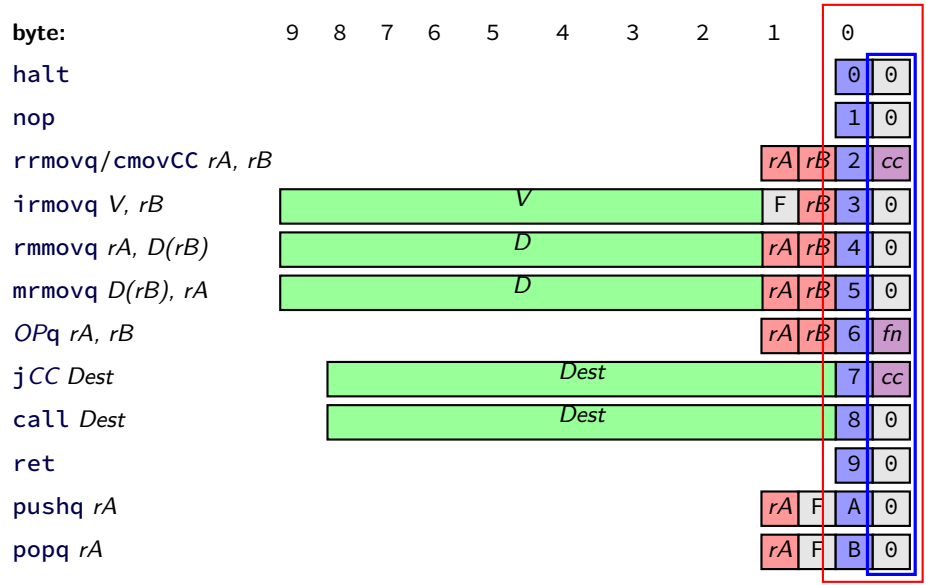


Y86 encoding table (written differently)



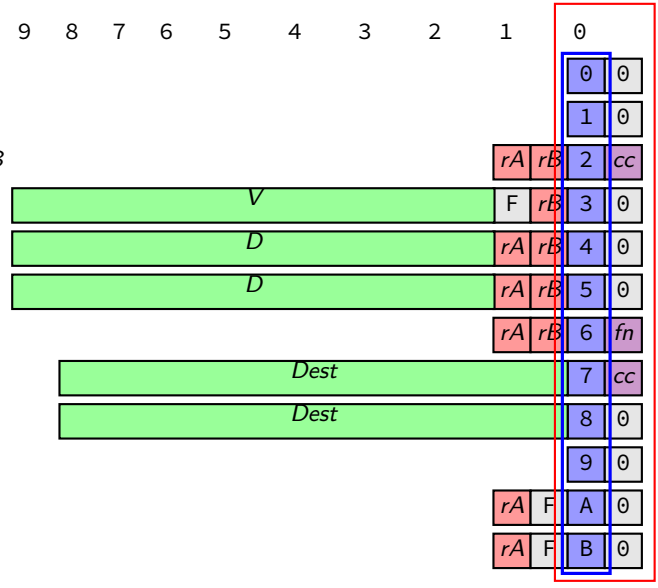
byte 0: bits 0–7

Y86 encoding table (written differently)



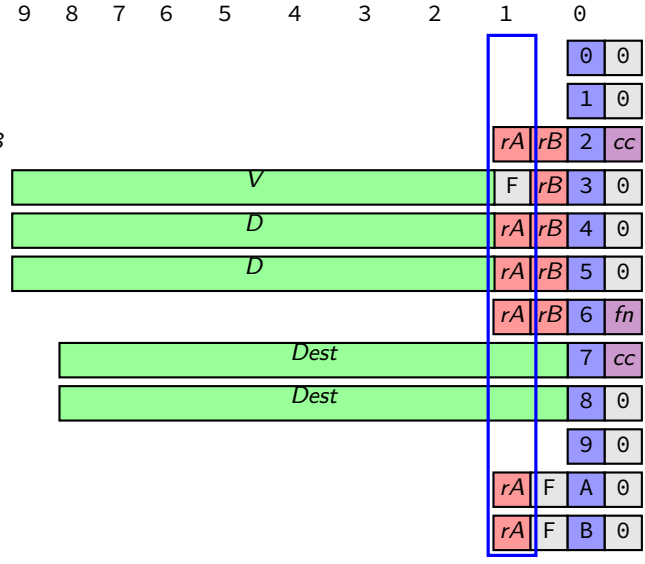
least sig. 4 bits of byte 0: bits 0–4

Y86 encoding table (written differently)



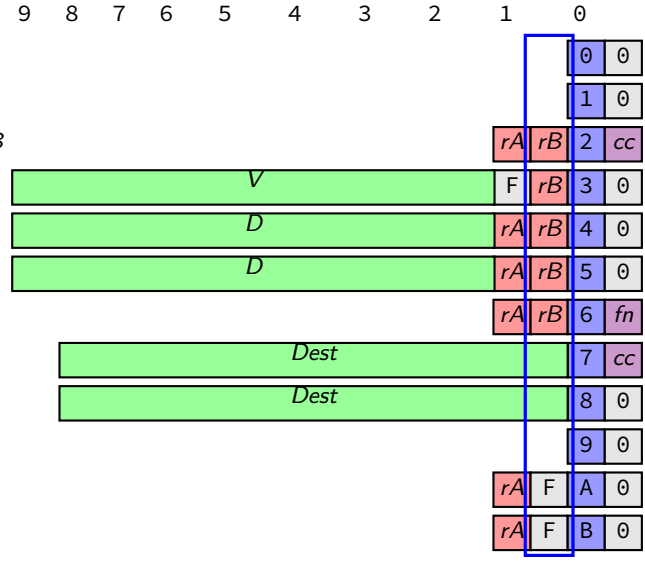
most sig. 4 bits of byte 0: bits 4–8

Y86 encoding table (written differently)



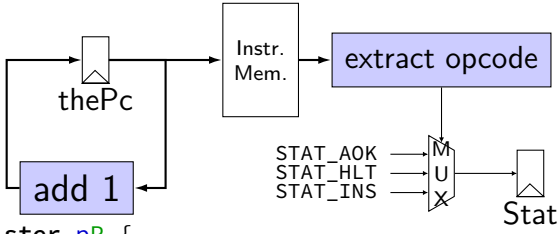
most sig. 4 bits of byte 1: bits 12–16

Y86 encoding table (written differently)



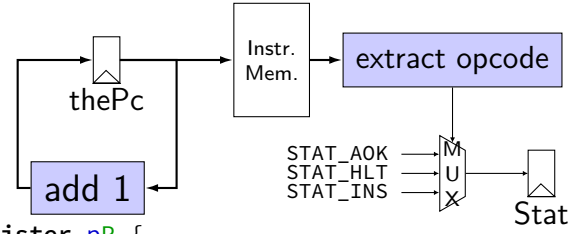
least sig. 4 bits of byte 1: bits 8–12

nop/halt CPU



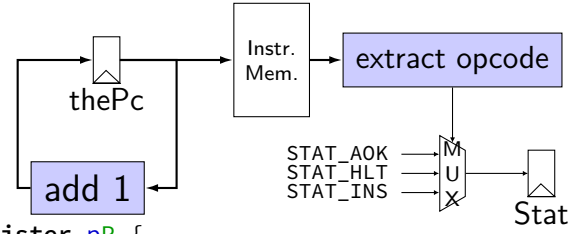

```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; // (default case)
];
```

nop/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)
];
```

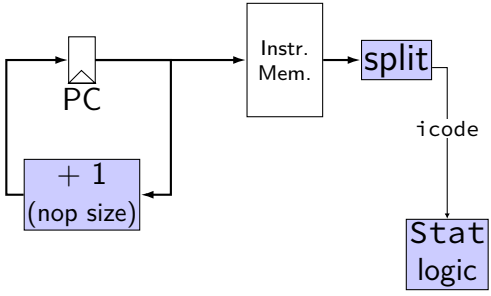
nop/halt CPU



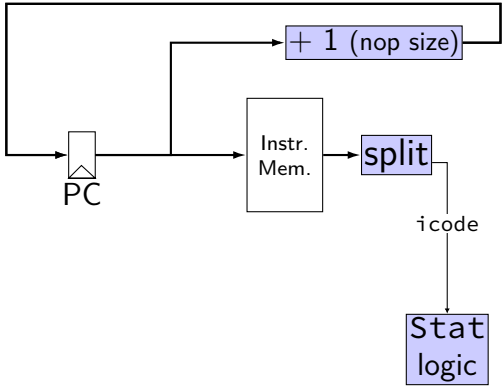

```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; // (default case)
];
```

demo

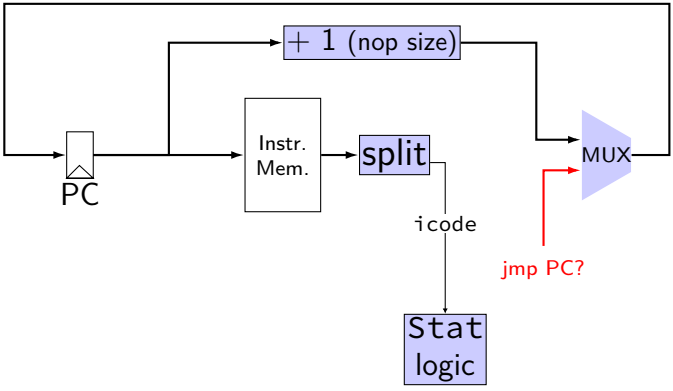
nop/halt → nop/jmp CPU



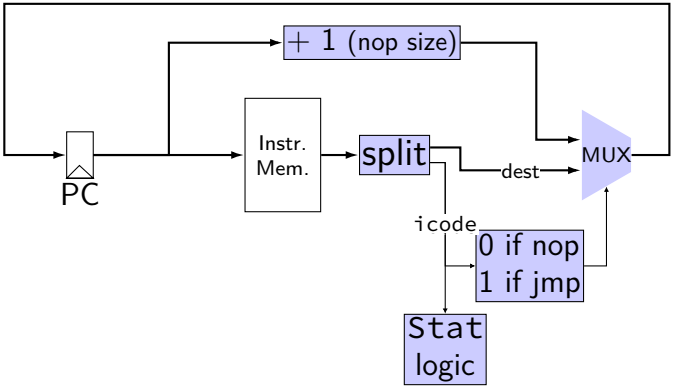
nop/halt → nop/jmp CPU



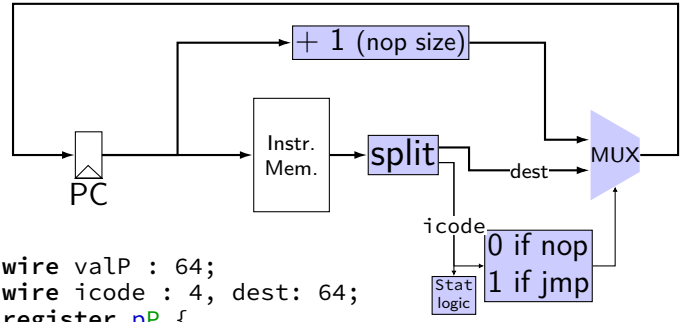
nop/halt → nop/jmp CPU



nop/halt → nop/jmp CPU



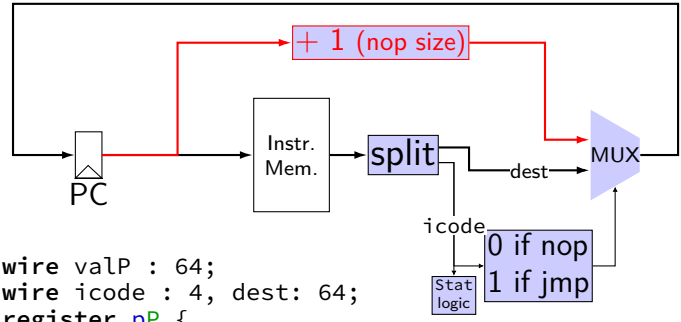
nop/jmp CPU




```
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;
];
```

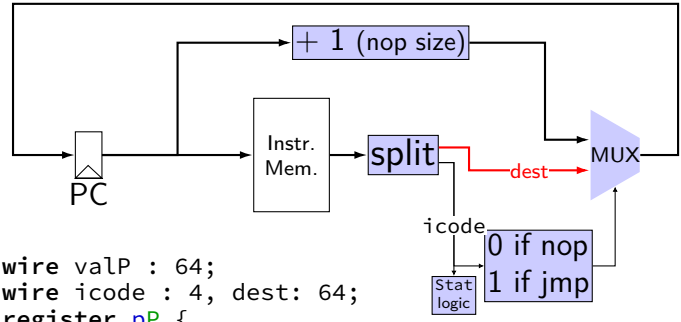
nop/jmp CPU




```
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;
];
```

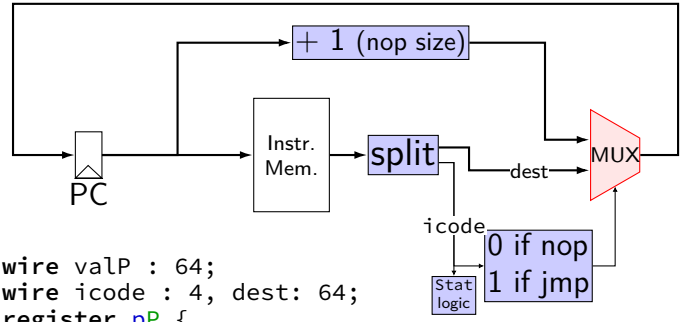
nop/jmp CPU




```
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;
];
```

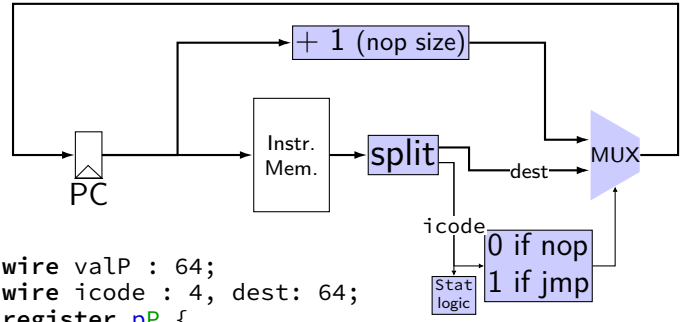
nop/jmp CPU




```
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;
];
```

demo: running nop/jmp

demo: yis

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: 7013000000000000000 |      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: 7013000000000000000  |      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
```

demo: debug and interactive mode

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 inputs to built-in components:
pc                  0x0000000000000000
Stat                              0x1

Values of outputs of built-in components:
i10bytes         0x00000000000000137010

Values of register bank signals:
P_thePc             0x0000000000000000
p_thePc             0x0000000000000001

Values of other wires:
dest                0x0000000000001370
icode                             0x1
valP                0x0000000000000001
```

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 inputs to built-in components:
pc                  0x0000000000000000
Stat                               0x1

Values of outputs of built-in components:
i10bytes         0x00000000000000137010

Values of register bank signals:
P_thePc             0x0000000000000000
p_thePc             0x0000000000000001

Values of other wires:
dest                0x0000000000001370
icode                              0x1
valP                0x0000000000000001
```

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 inputs to built-in components:
....
```

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 inputs to built-in components:
....
```

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
```

simple ISA: addq

`addq %rXX, %rYY`

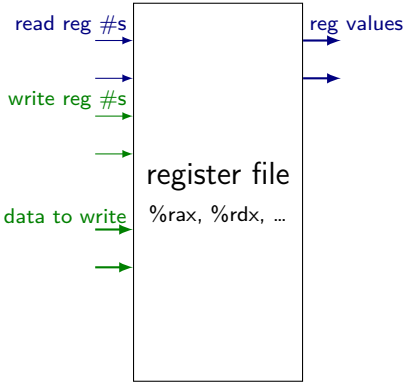
encoding: `6` `0` `%rXX` `%rYY` (two 4-bit register #s)

- 2 byte instructions, no opcode

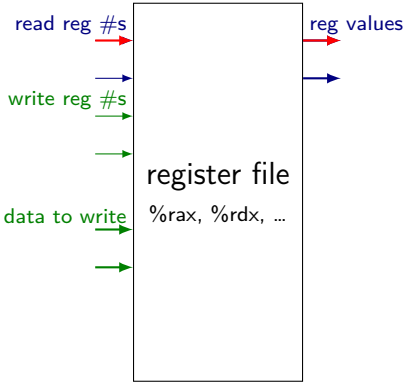
for now: no other instructions

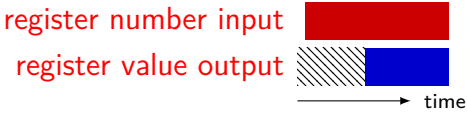
- later: adding support for nop+halt

register file

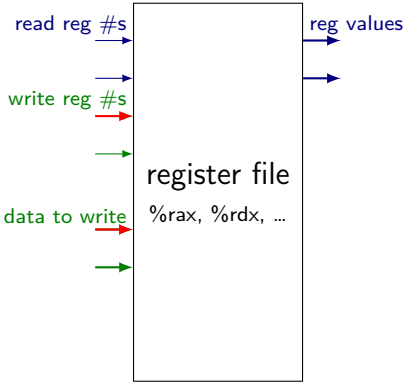


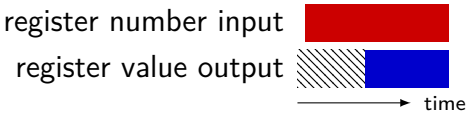
register file

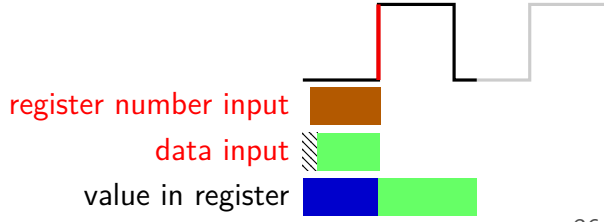




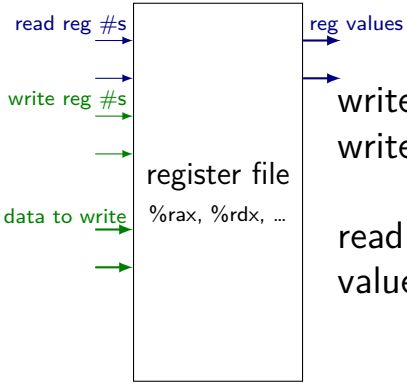
register file





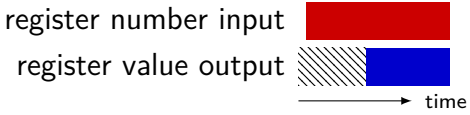


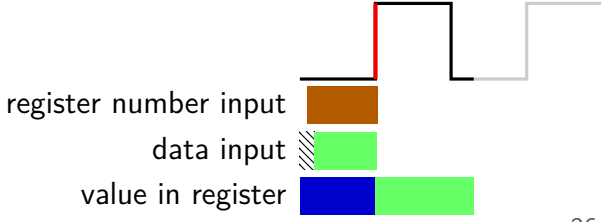
register file



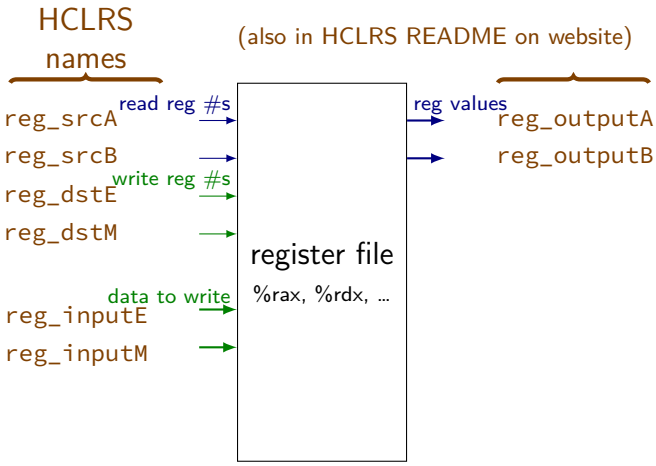
write register #15 (REG_NONE):
write is ignored

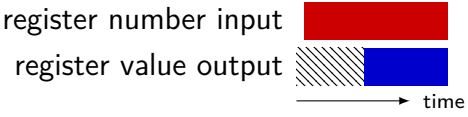
read register #15 (REG_NONE):
value is always 0

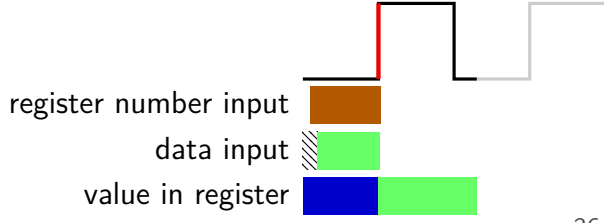




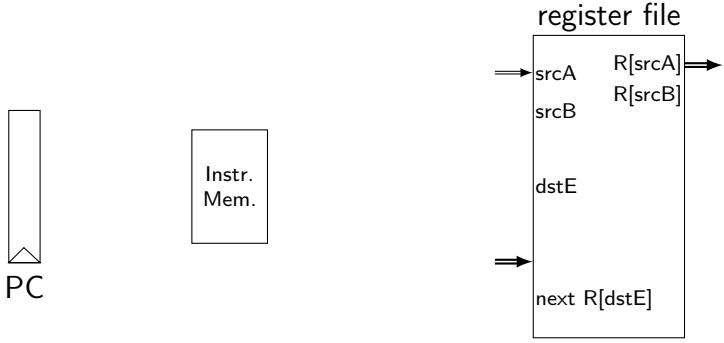
register file



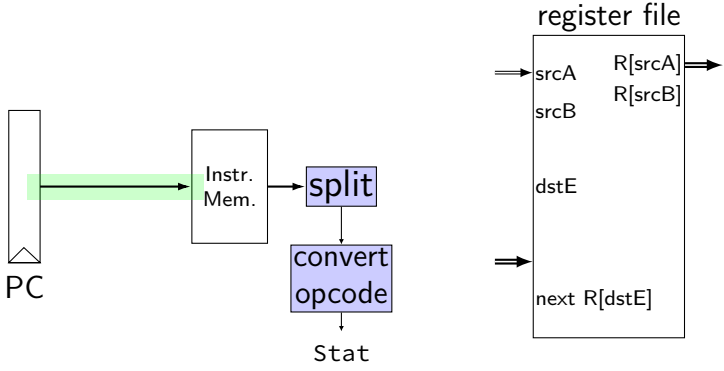




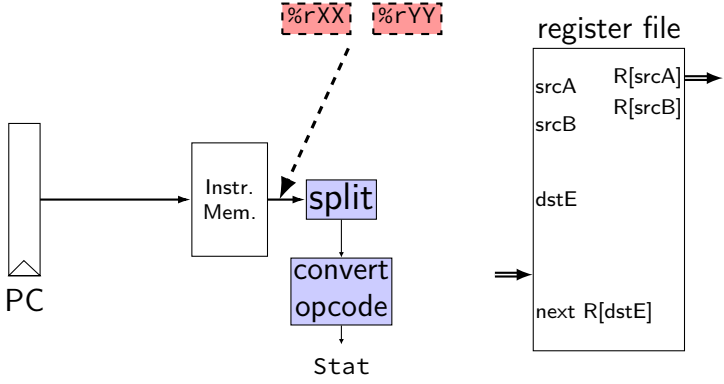
addq CPU


register file
srcA
R[srcA]
R[srcB]
srcB
dstE
Instr.
Mem.
PC
next R[dstE]

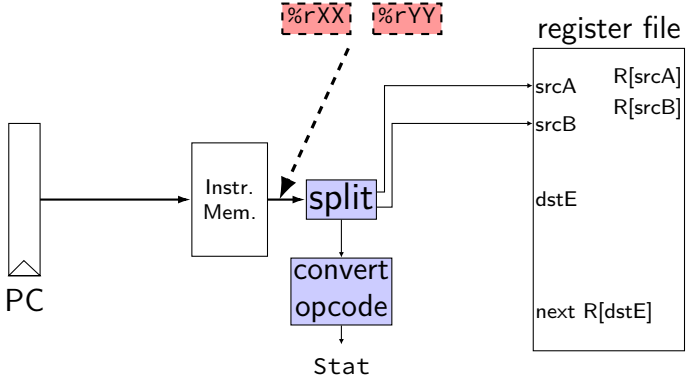
addq CPU


register file
srcA
srcB
R[srcA]
R[srcB]
dstE
next R[dstE]
PC
Instr.
Mem.
split
convert
opcode
Stat

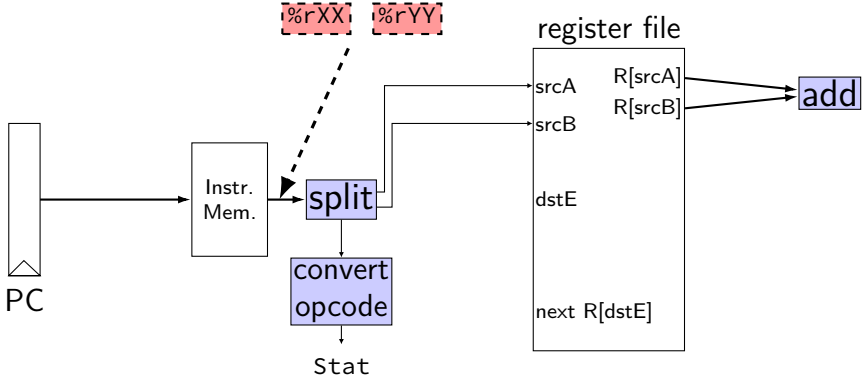
addq CPU


%rXX
%rYY
register file
srcA
R[srcA]
R[srcB]
srcB
Instr.
Mem.
split
dstE
convert
opcode
PC
next R[dstE]
Stat

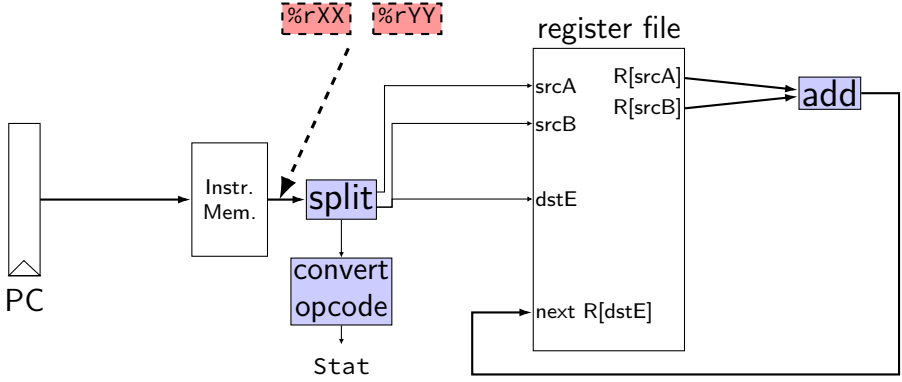
addq CPU


%rXX
%rYY
register file
srcA
R[srcA]
R[srcB]
srcB
dstE
next R[dstE]
Instr.
Mem.
split
convert
opcode
PC
Stat

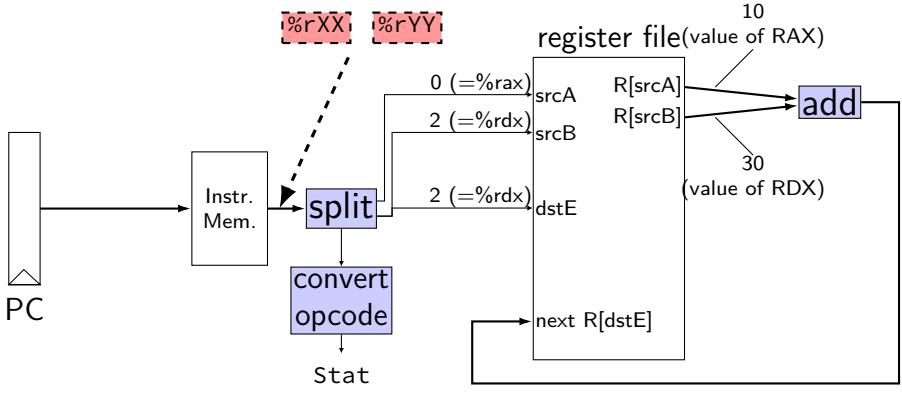
addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
add
Instr.
Mem.
split
convert
opcode
Stat
PC

addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
add
Instr.
Mem.
split
convert
opcode
Stat
PC

addq CPU

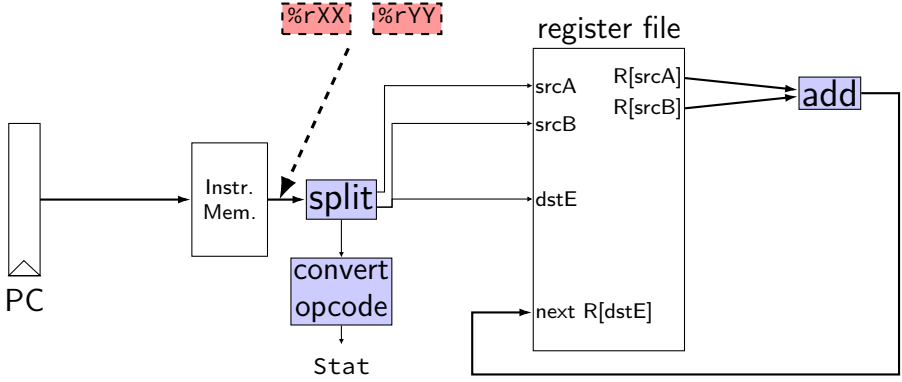



```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

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

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

addq CPU



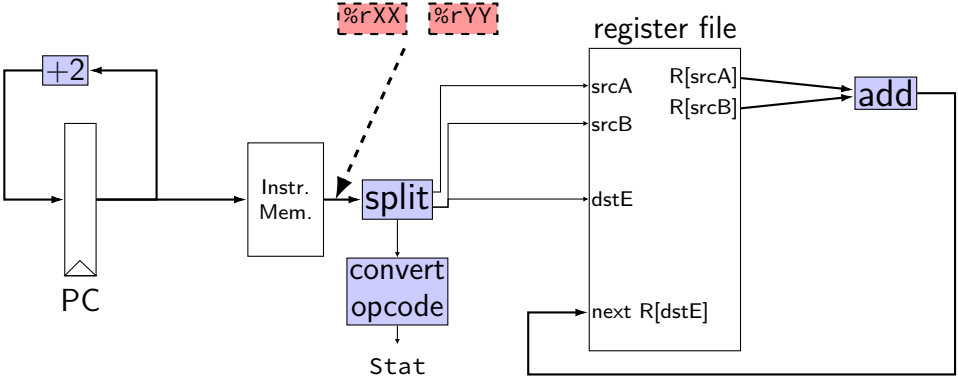

```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

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

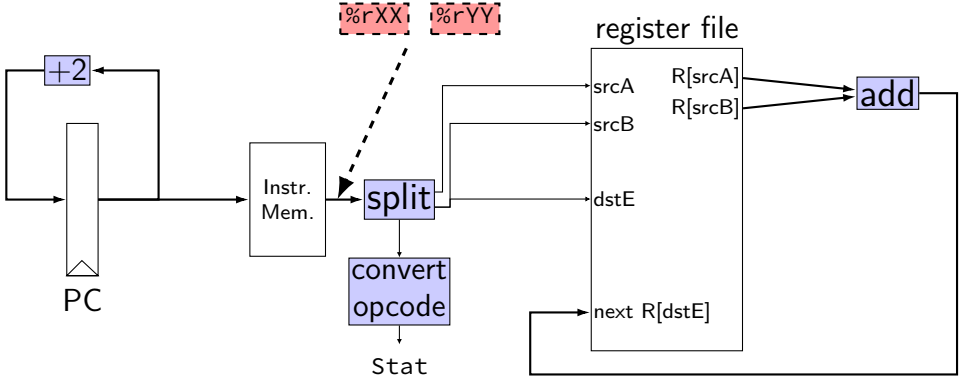
after cycle 1: PC = ????, rax = 10, rbx = 20, rdx = 40

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

addq CPU


%rXX
%rYY
register file
+2
srcA
srcB
dstE
R[srcA]
R[srcB]
add
Instr.
Mem.
split
convert
opcode
PC
next R[dstE]
Stat

addq CPU



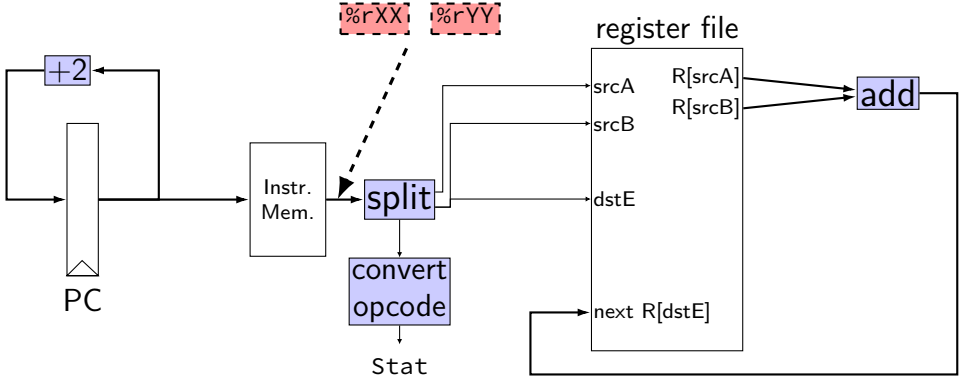

```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

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

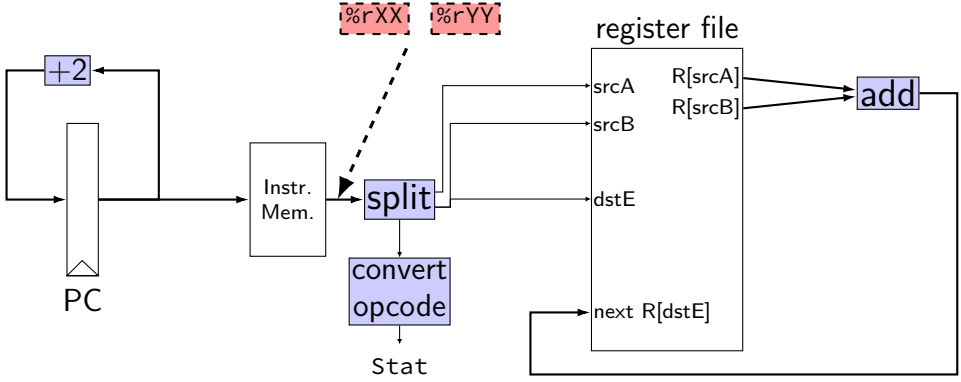
after cycle 1: PC = 0x02, rax = 10, rbx = 20, rdx = 40

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

addq CPU: HCL


%rXX
%rYY
register file
+2
srcA
R[srcA]
R[srcB]
add
srcB
Instr.
Mem.
split
dstE
convert
opcode
PC
next R[dstE]
Stat

addq CPU: HCL

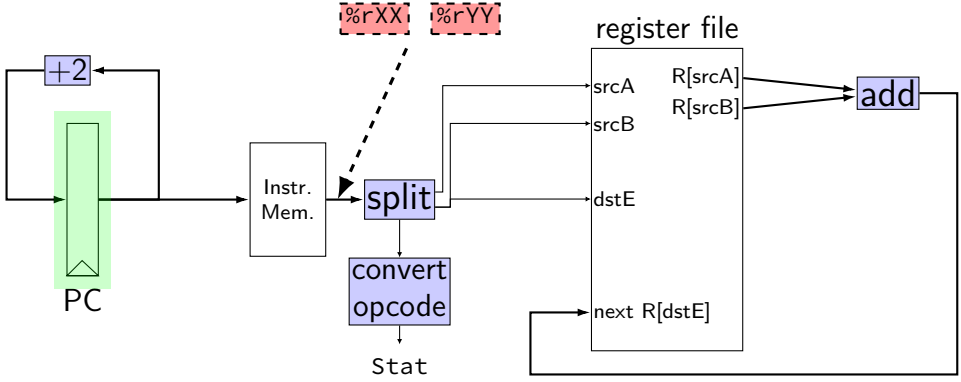



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

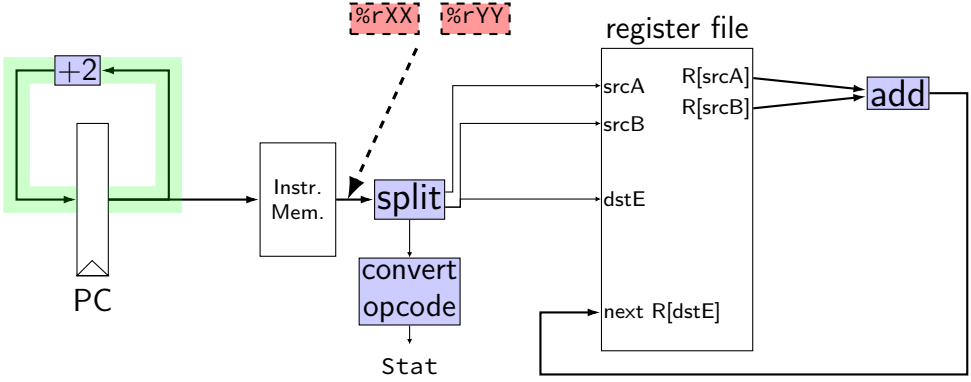



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

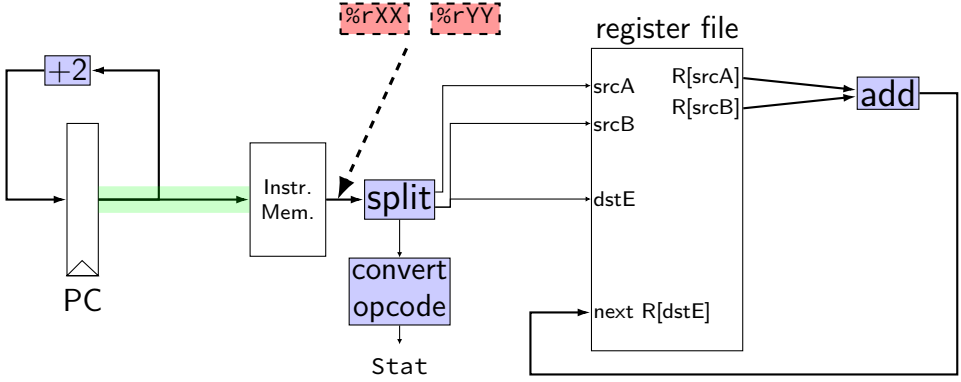



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

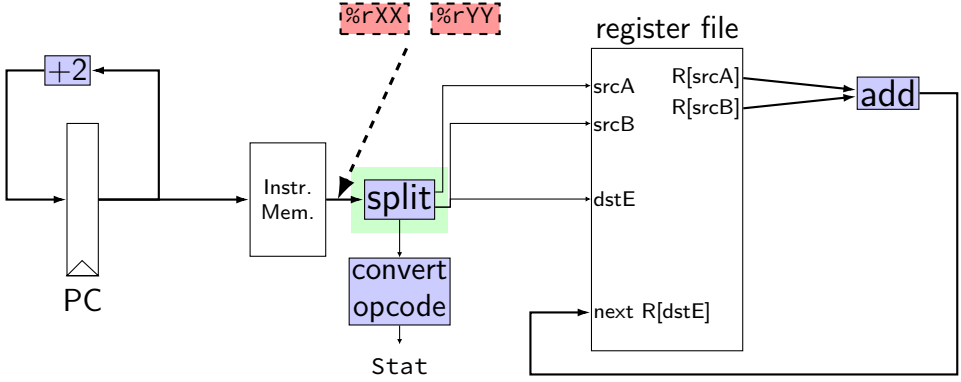



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

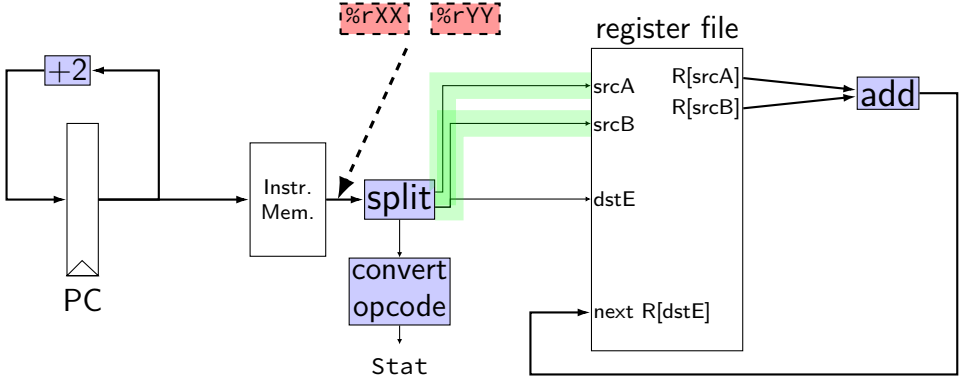



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

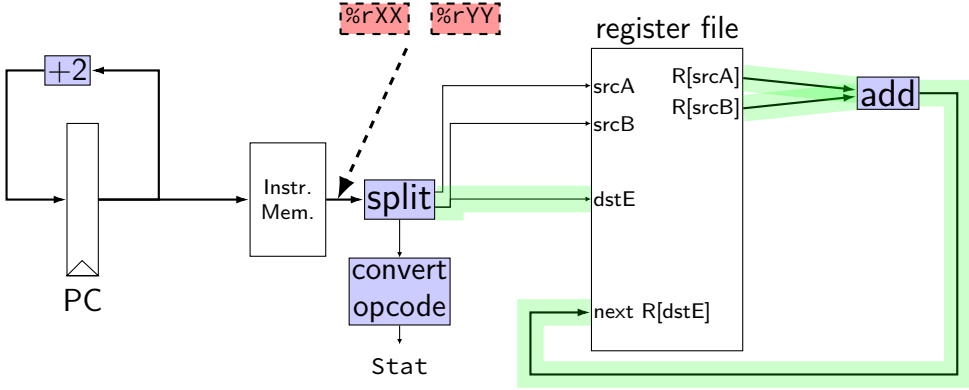



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

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

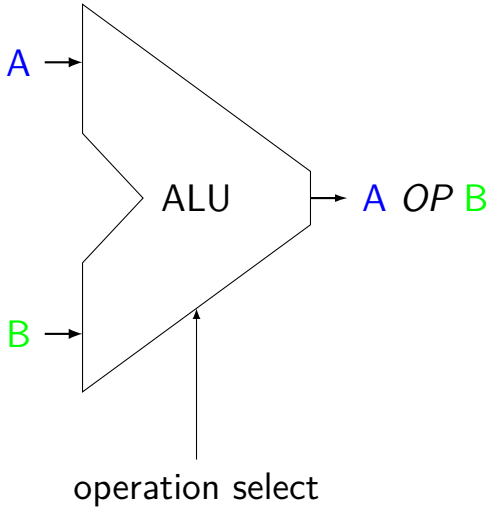
book uses names like `valC` — not required!

- author's environment limited adding new wires

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

implement your own ALU

ALUs



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

ALUs in HCLRS

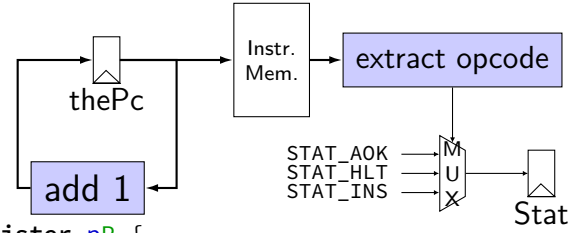
HCLRS doesn't supply an ALU

- the HCL the textbook authors use does

...but you can build one yourself

- not required — we check functionality

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)
];
```

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...)

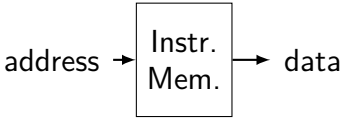
D. one of the register file 'register value to write' inputs (reg_input...)

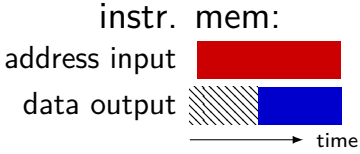
E. the instruction memory's address input (pc)

simple ISA: mov-to-register

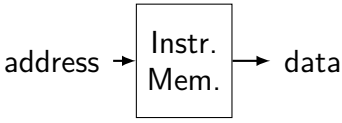
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

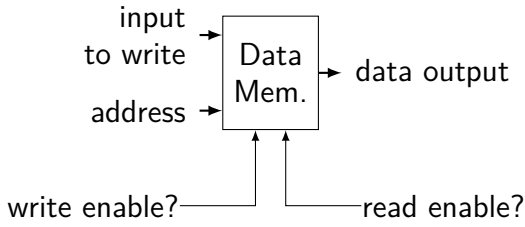
two memories





two memories

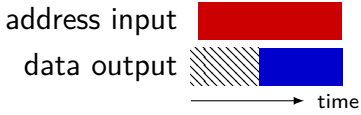




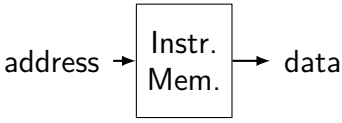
data mem. in **read** mode

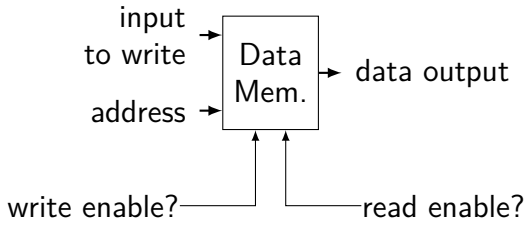
—or—

instr. mem:



two memories

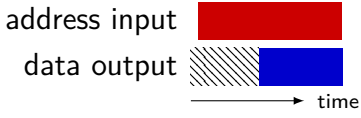




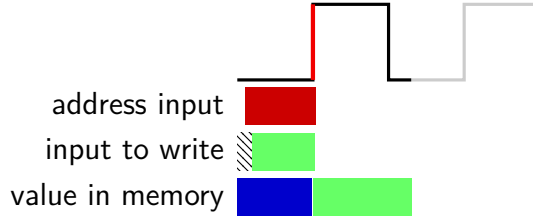
data mem. in **read** mode

—or—

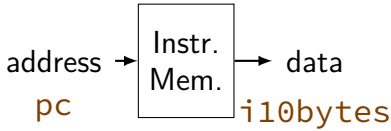
instr. mem:

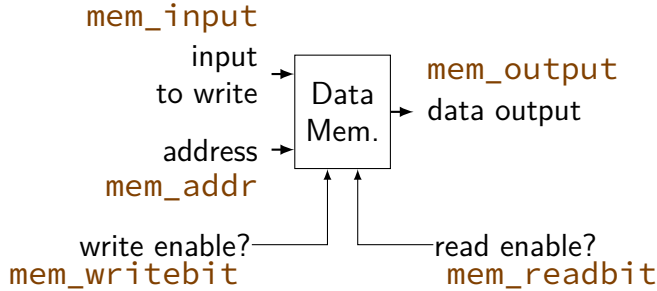


data mem.
in **write** mode:



two memories

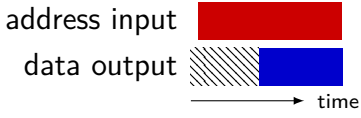




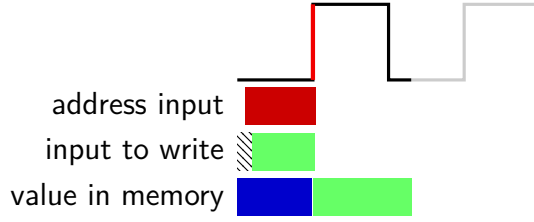
data mem. in **read** mode

—or—

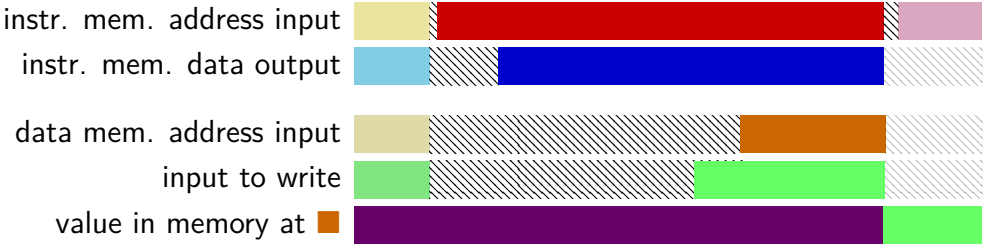
instr. mem:



data mem.
in **write** mode:



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*

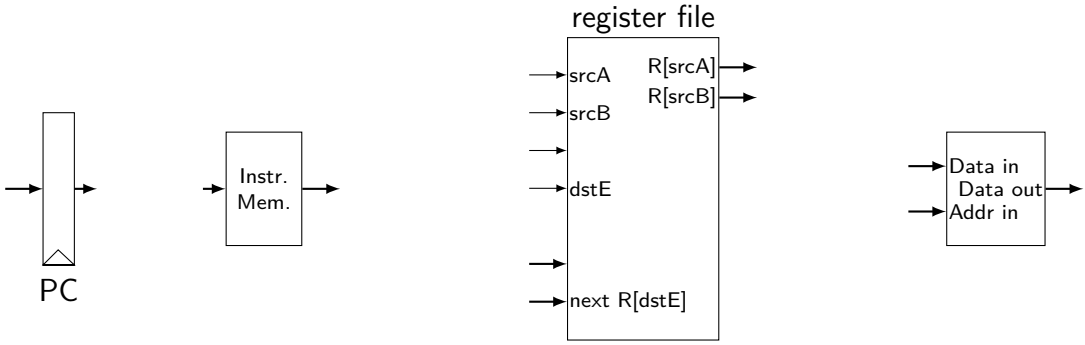
exercise: mov-to-register

```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

for which are these are we going to need MUXes? before...

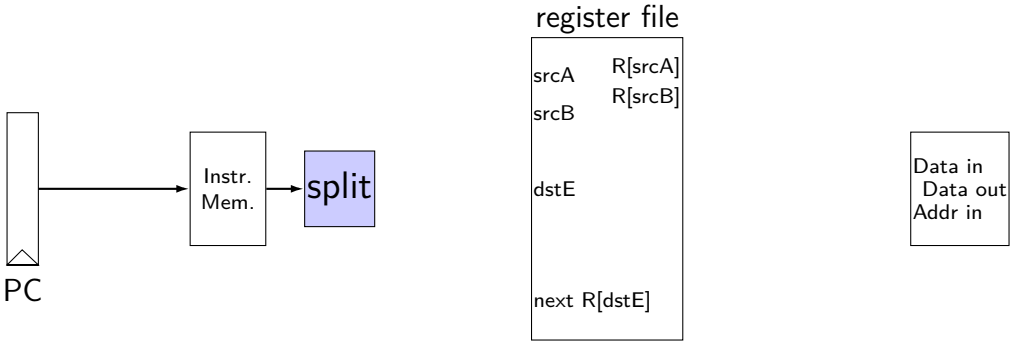
A. register file's register number (index) inputs (`reg_srcA`, `reg_srcB`, `reg_dstE`, ...)
B. register file's value inputs (`reg_inputE/M`)
C. PC register's input
D. instruction memory's address input (`pc`)

mov-to-register CPU



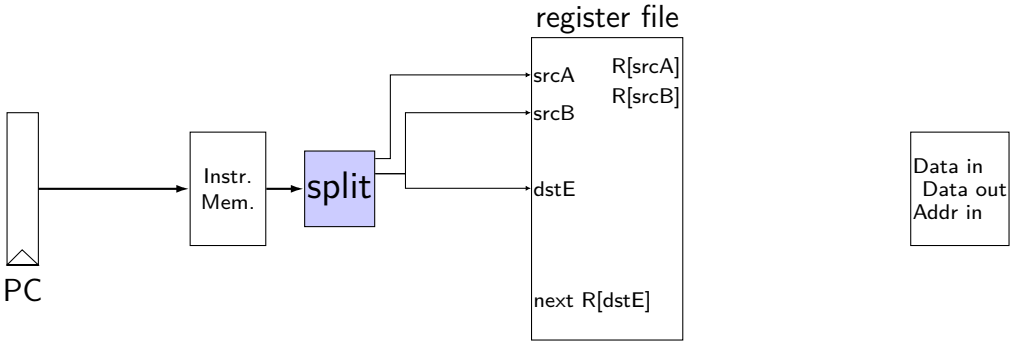
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



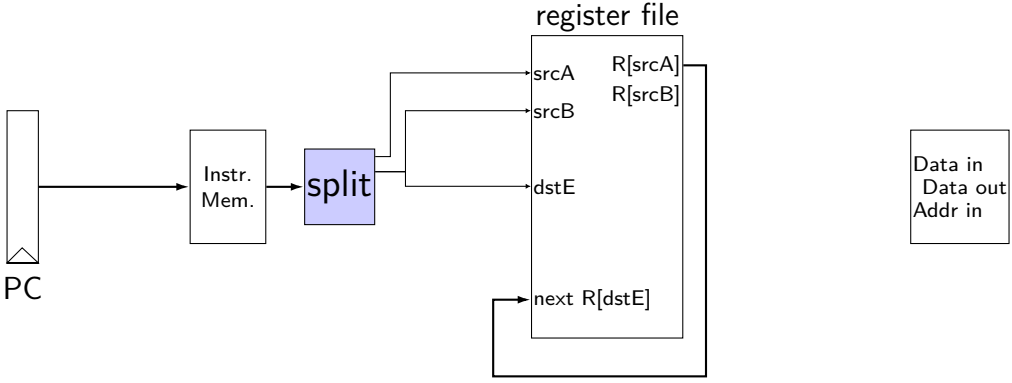
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



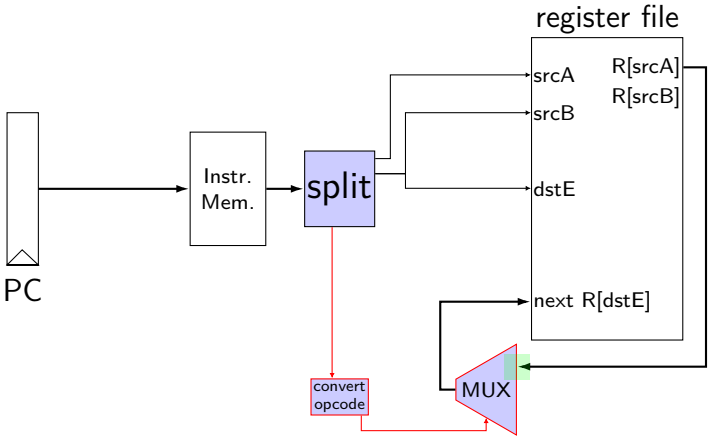
`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



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

mov-to-register CPU

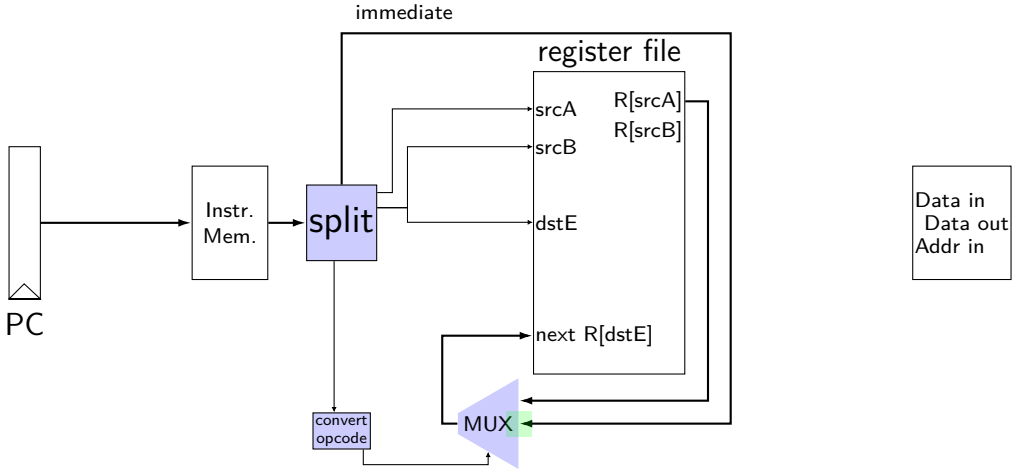


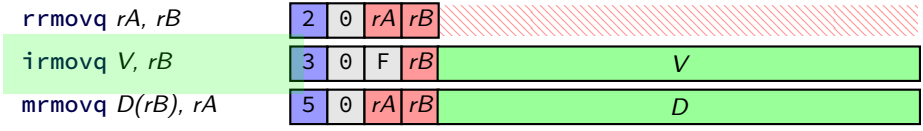
Data in
Data out
Addr in

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

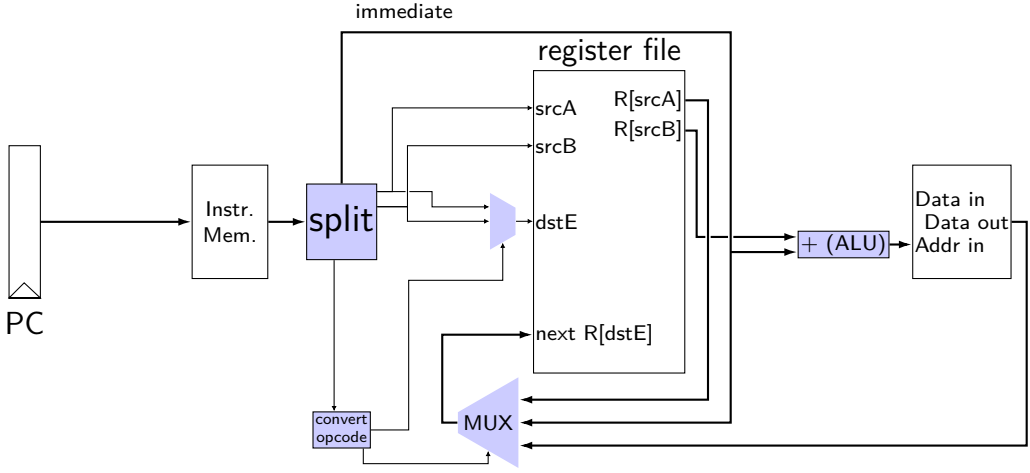
`rrmovq` *rA, rB*	2	0	*rA*	*rB*		
`irmovq` *V, rB*	3	0	F	*rB*	*V*	
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*	

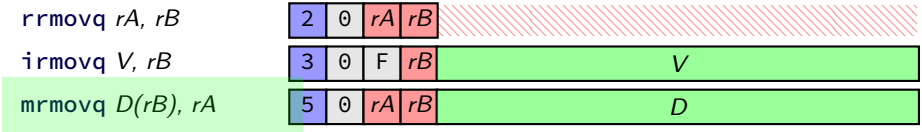
mov-to-register CPU



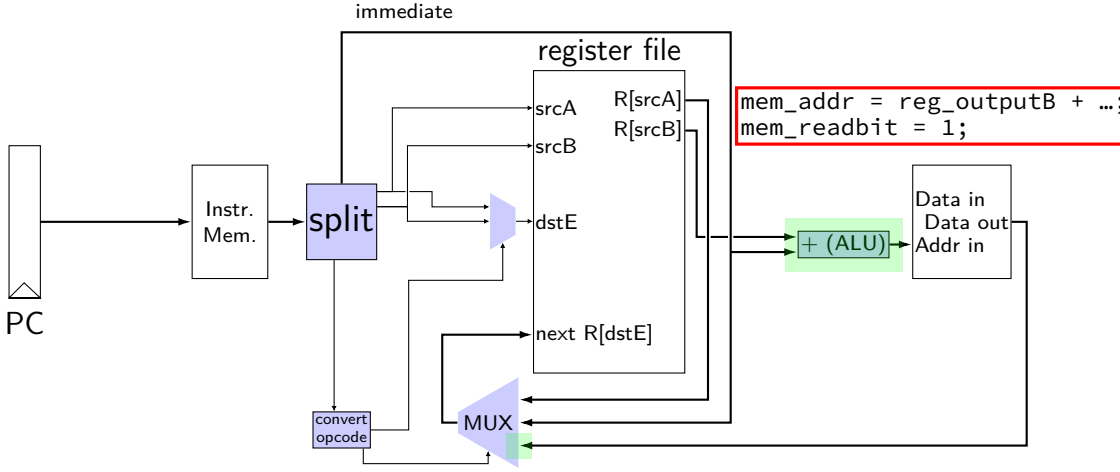


mov-to-register CPU



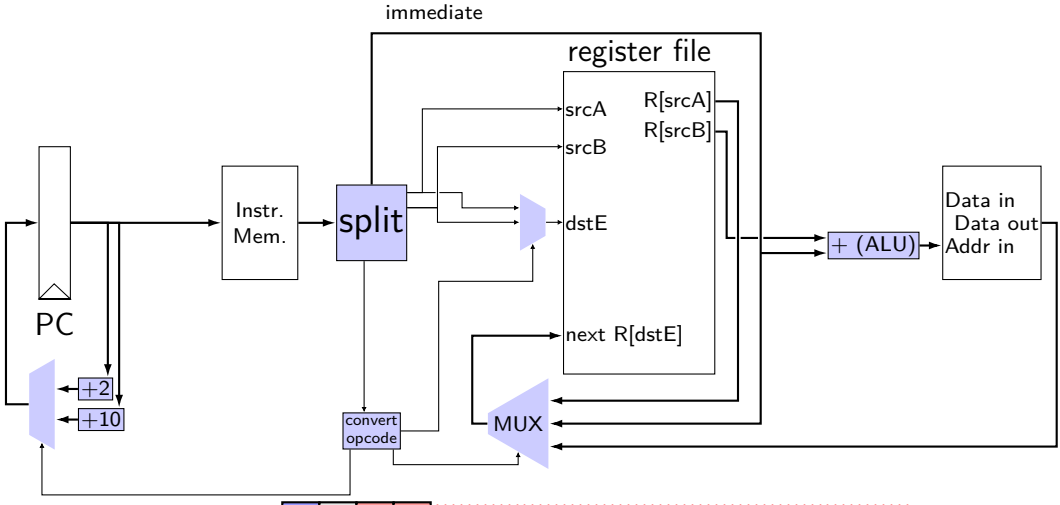


mov-to-register CPU



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

mov-to-register CPU

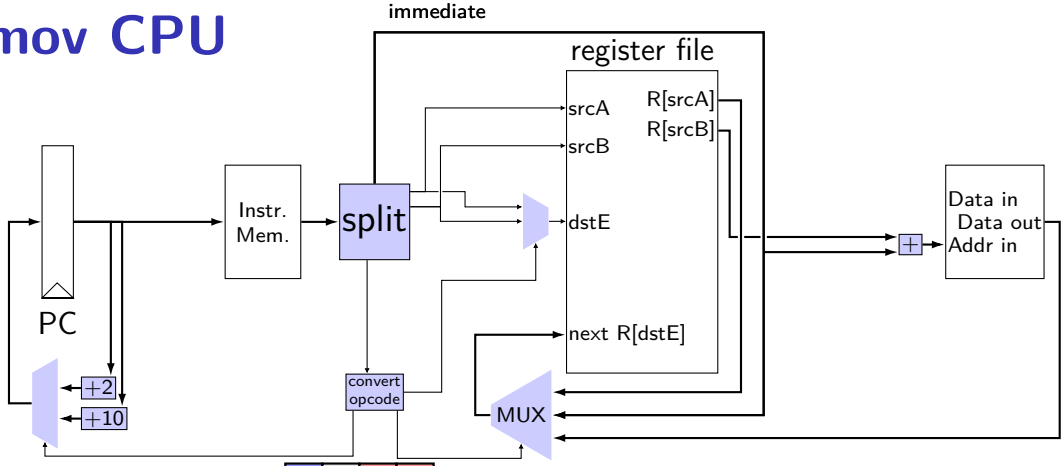


`rrmovq` *rA*, *rB*	2	0	*rA*	*rB*	
`irmovq` *V*, *rB*	3	0	F	*rB*	*V*
`mrmovq` *D*(*rB*), *rA*	5	0	*rA*	*rB*	*D*

simple ISA: mov (all cases)

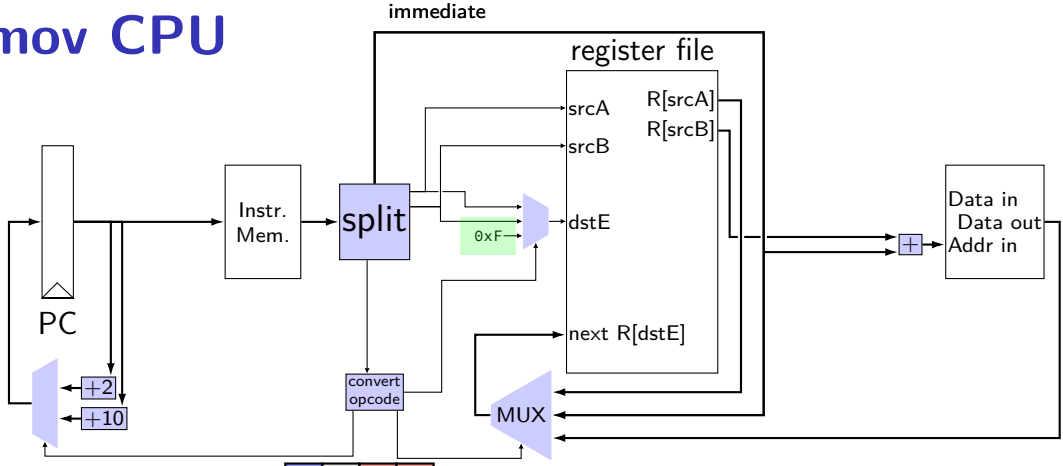
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

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

mov CPU



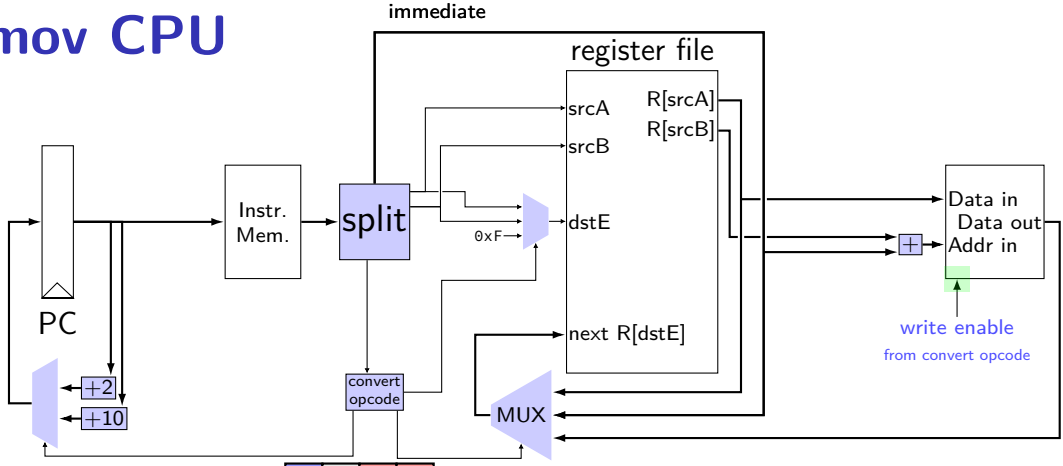
`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU



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

mov CPU



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

data path versus control path

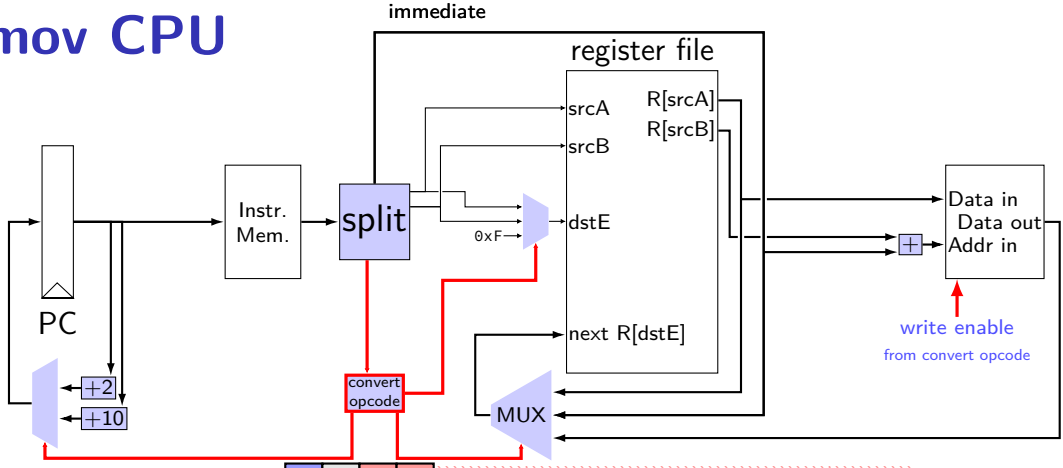
data path — signals carrying “actual data”

control path — signals that control MUXes, etc.

fuzzy line: e.g. are condition codes part of control path?

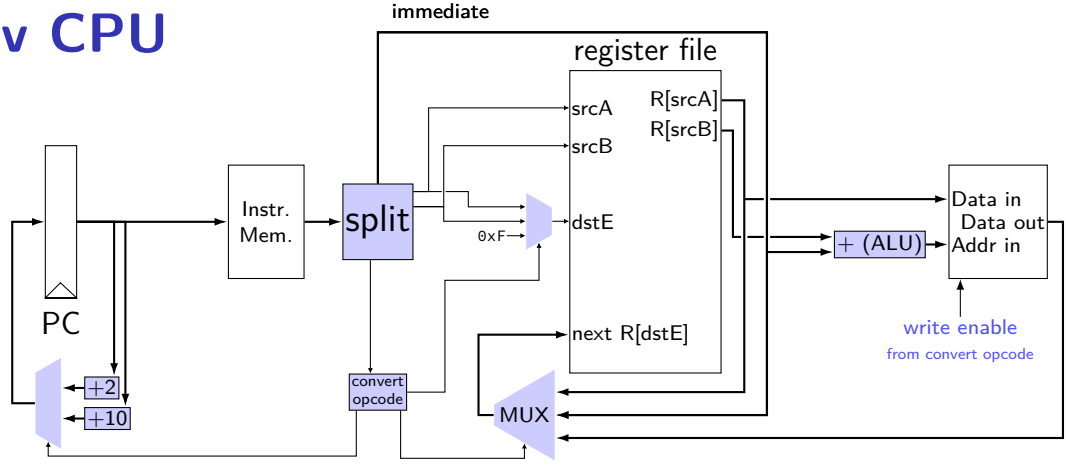
we will often omit parts of the control path in drawings, etc.

mov CPU

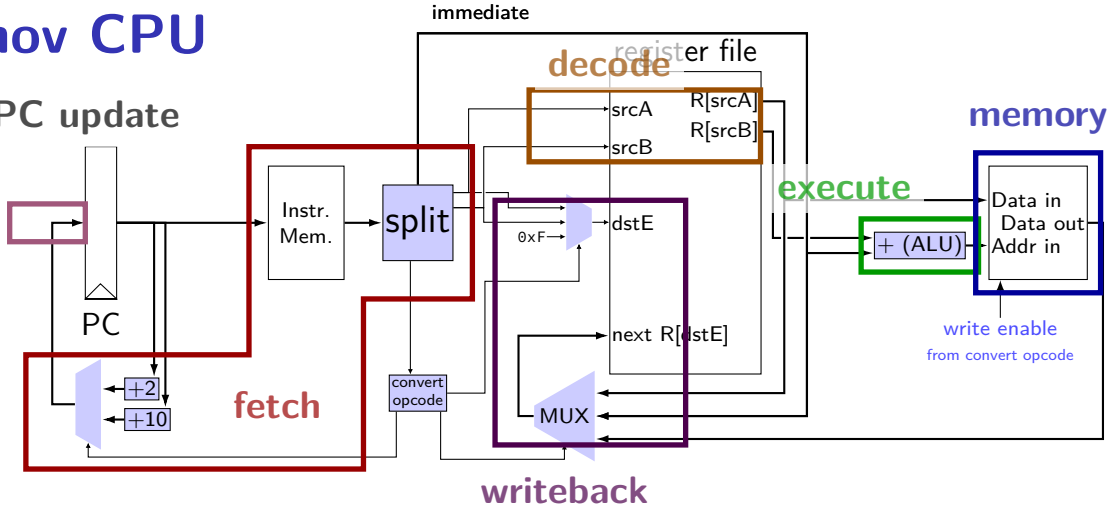


`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU


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

mov CPU


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

Stages

conceptual division of instruction:

fetch — read instruction memory, split instruction, compute length

decode — read register file

execute — arithmetic (including of addresses)

memory — read or write data memory

write back — write to register file

PC update — compute next value of PC

stages and time

fetch / decode / execute / memory / write back / PC update

Order when these events happen `pushq %rax` instruction:

1. instruction read
2. memory changes
3. `%rsp` changes
4. PC changes

Hint: recall how registers, register files, memory works

a. 1; then 2, 3, and 4 in any order
b. 1; then 2, 3, and 4 at almost the same time
c. 1; then 2; then 3; then 4
d. 1; then 3; then 2; then 4
e. 1; then 2; then 3 and 4 at almost the same time
f. something else

SEQ: instruction fetch

read instruction memory at PC

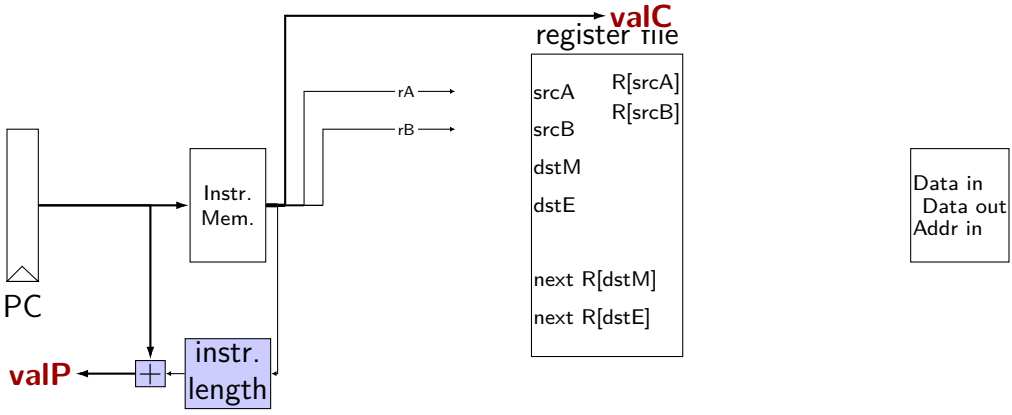
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

instruction fetch

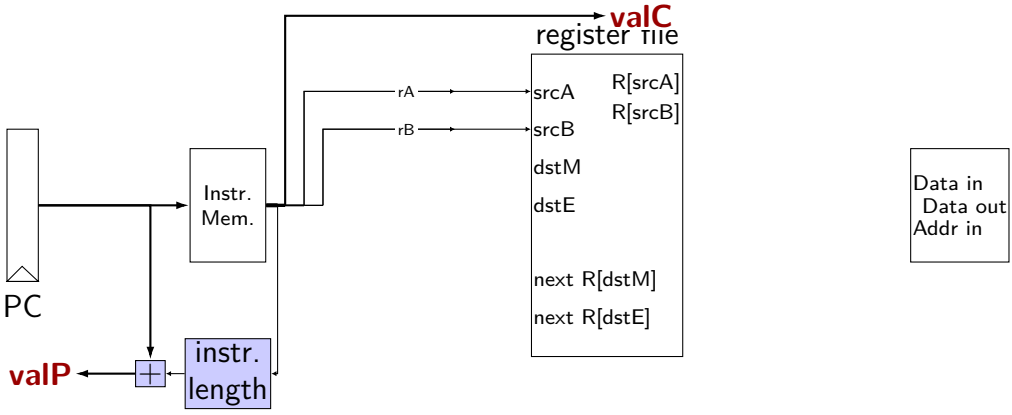


SEQ: instruction “decode”

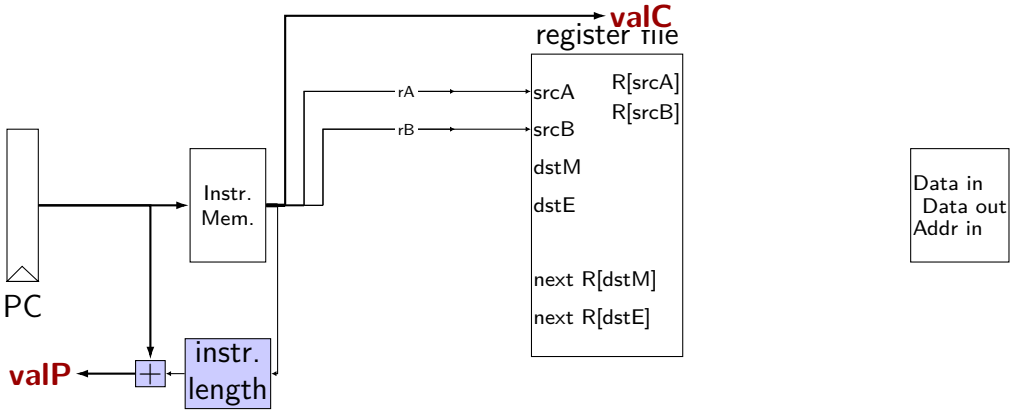
read registers

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`

SEQ: srcA, srcB

always read rA, rB?

Problems:

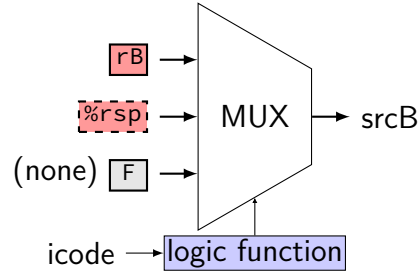
- push rA
- pop
- call
- ret

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

MUX controlled by icode

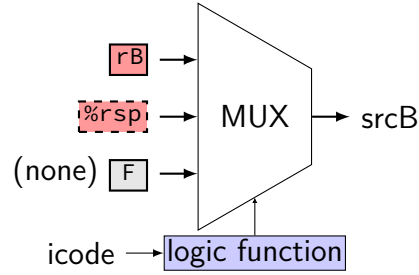
SEQ: possible registers to read

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`



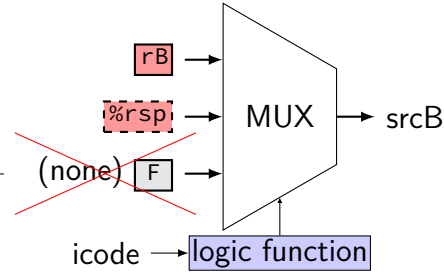
SEQ: possible registers to read

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`

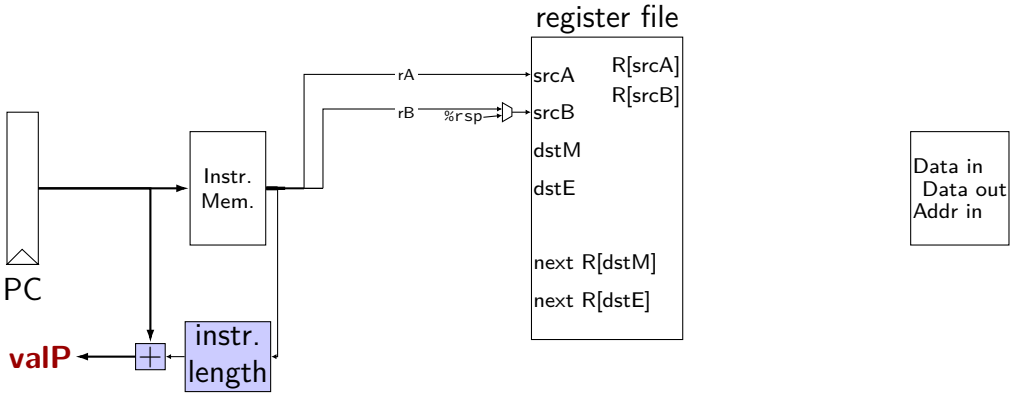


SEQ: possible registers to read

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)

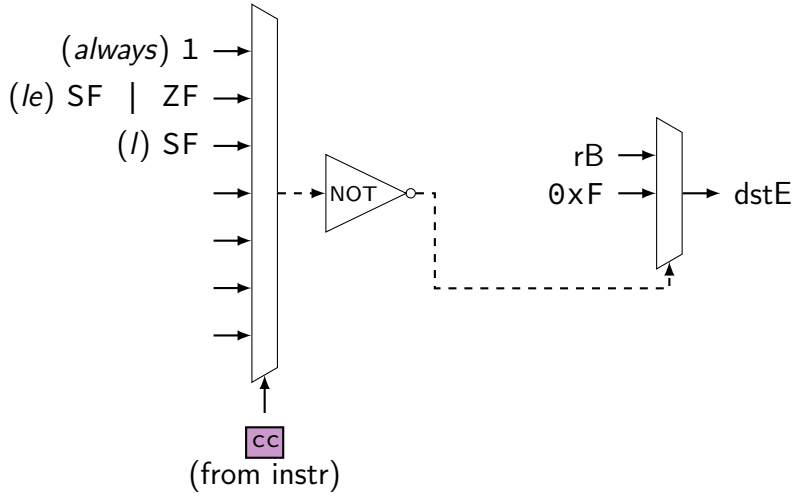
- valE — ALU output

read prior condition codes

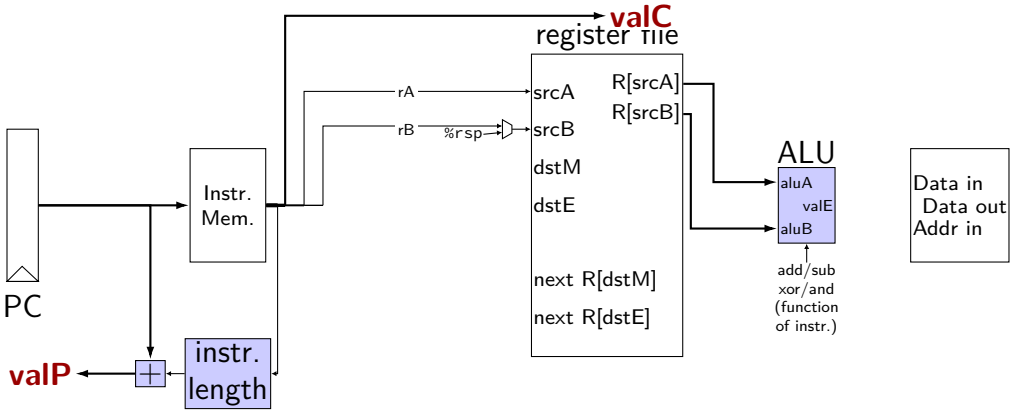
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

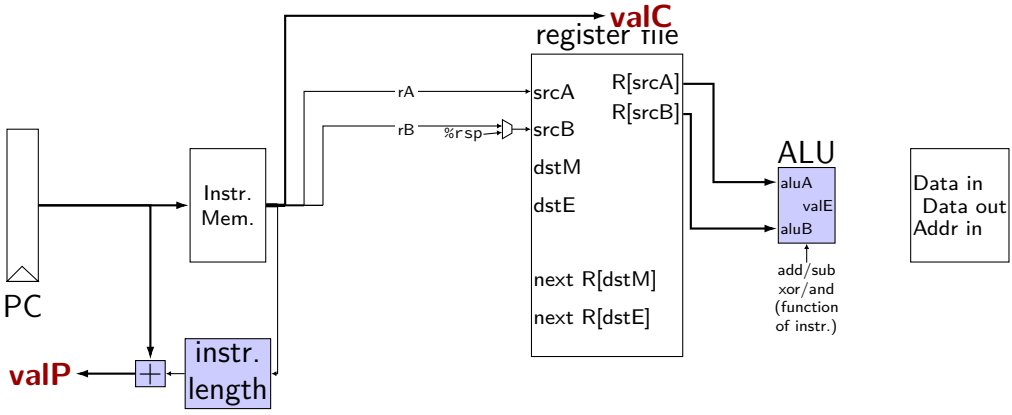
using condition codes: cmov



execute (1)


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

execute (1)



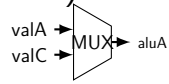
exercise: which of these instructions would there be a problem ?
`nop`, `addq`, `mrmovq`, `popq`, `call`,

SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

```
mrmovq
rmmovq
```



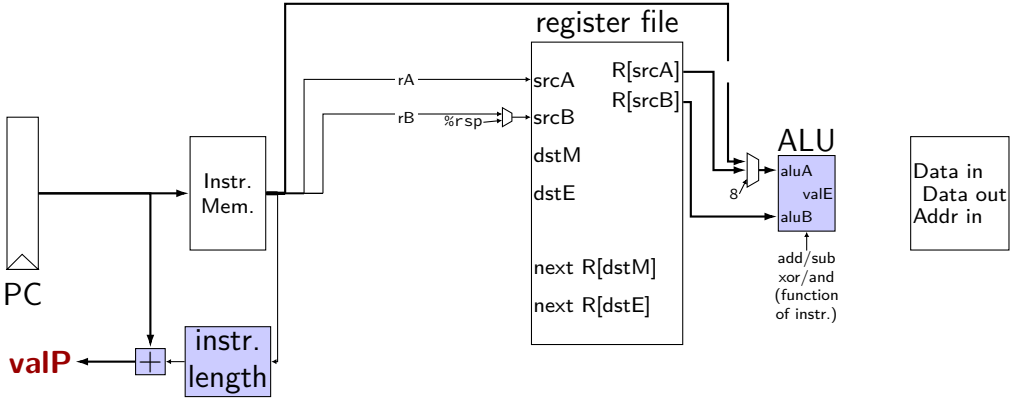
no, constants: (rsp +/- 8)

```
pushq
popq
call
ret
```

extra signals: aluA, aluB

computed ALU input values

execute (2)

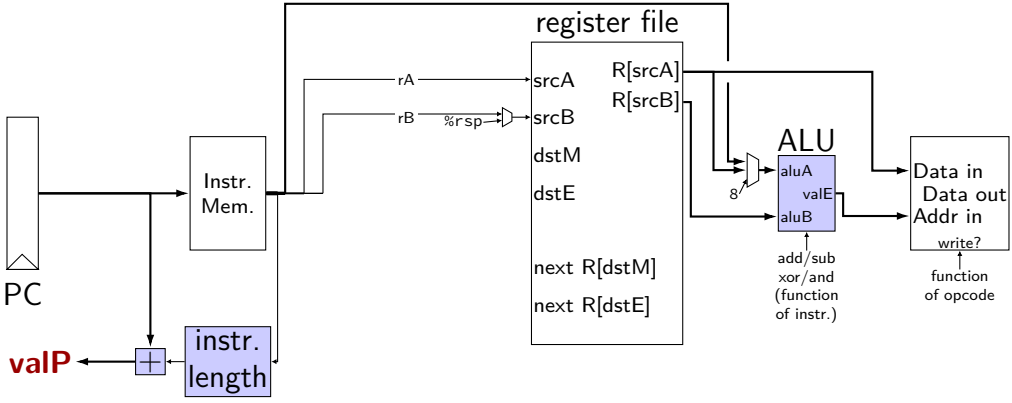

register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
Instr.
Mem.
PC
valP
instr.
length

SEQ: Memory

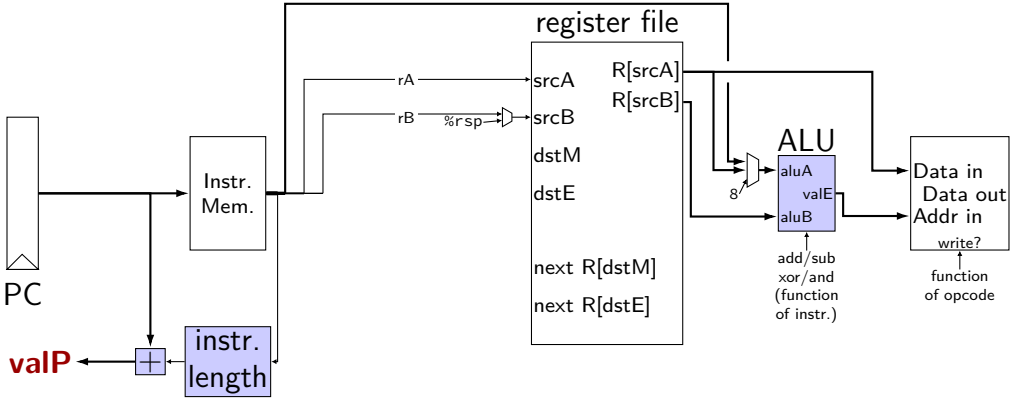
read or write data memory

valM — value read from memory (if any)

memory (1)


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

memory (1)



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

SEQ: control signals for memory

read/write — read enable? write enable?

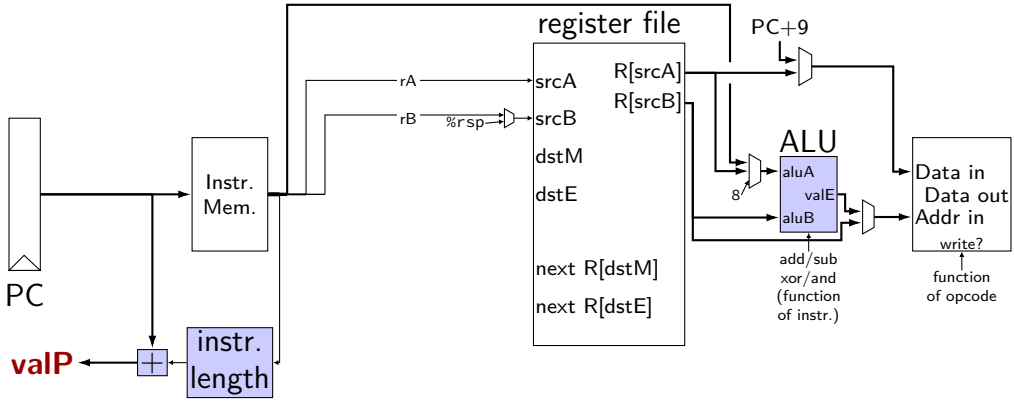
Addr — address

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

Data — value to write

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

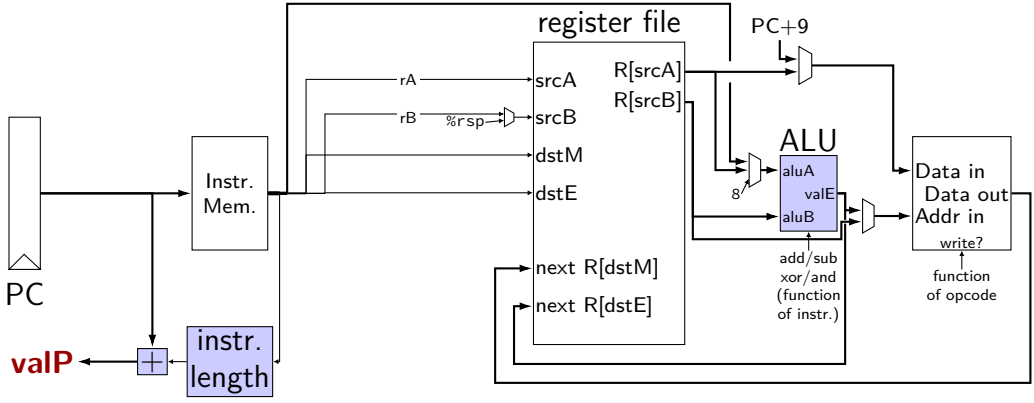
memory (2)


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

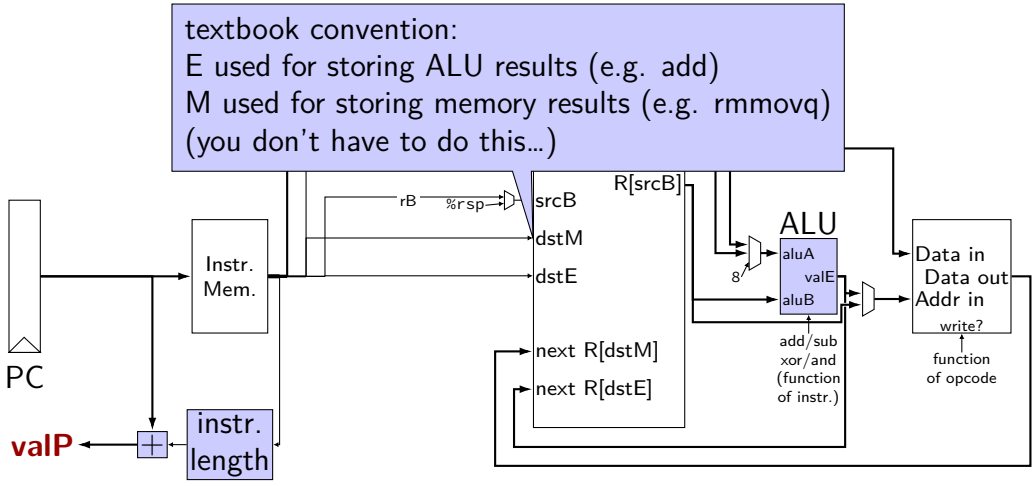
SEQ: write back

write registers

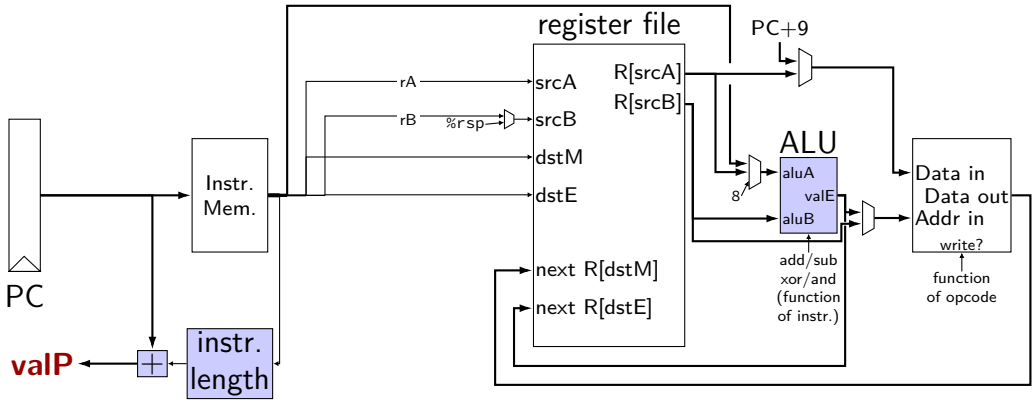
write back (1)


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

write back (1)


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

write back (1)



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

SEQ: control signals for WB

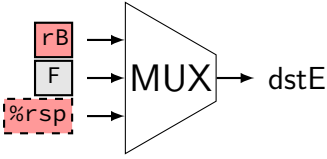
two write inputs — two needed by `popq`

valM (memory output), valE (ALU output)

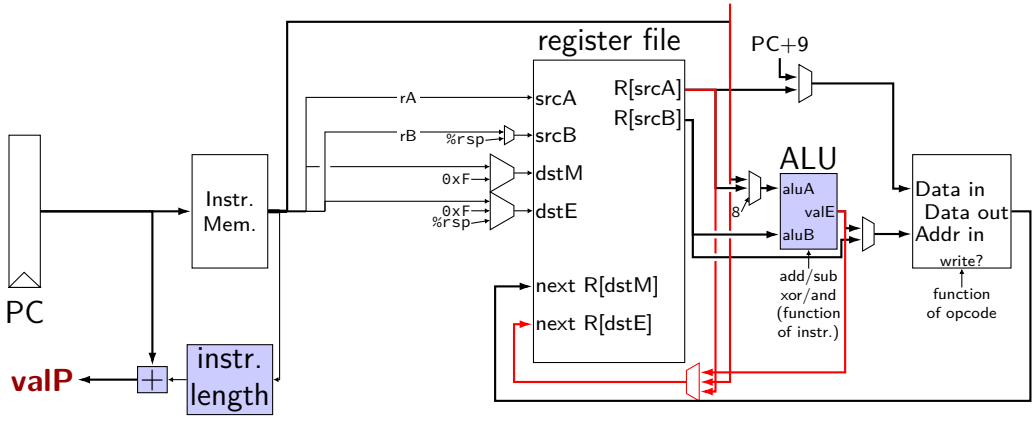
two register numbers

dstM, dstE

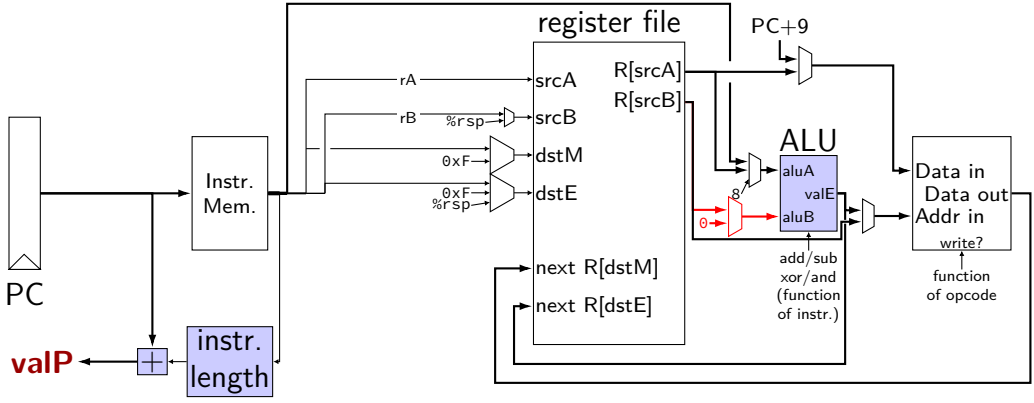
write disable — use dummy register number `0xF`



write back (2a)



write back (2b)

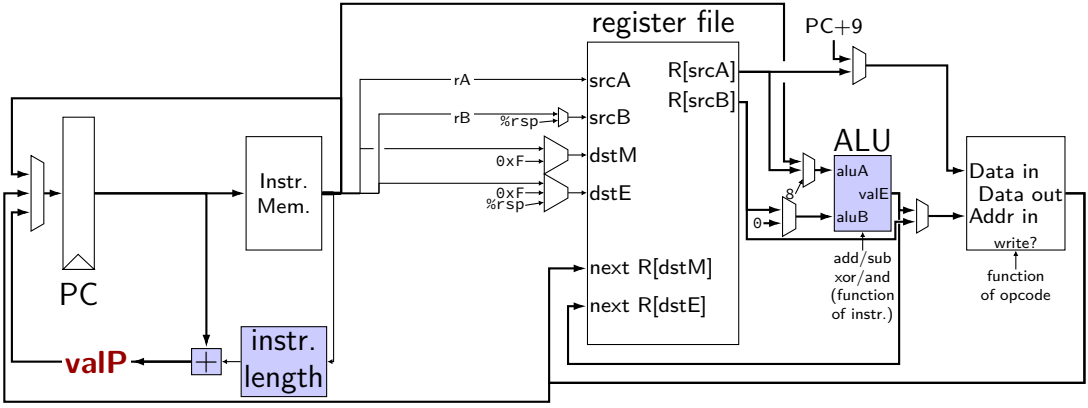


SEQ: Update PC

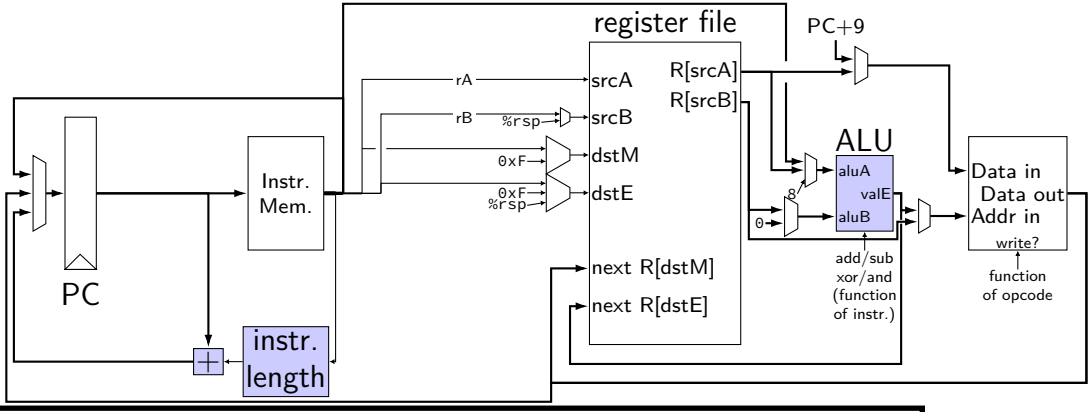
choose value for PC next cycle (input to PC register)

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

PC update


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

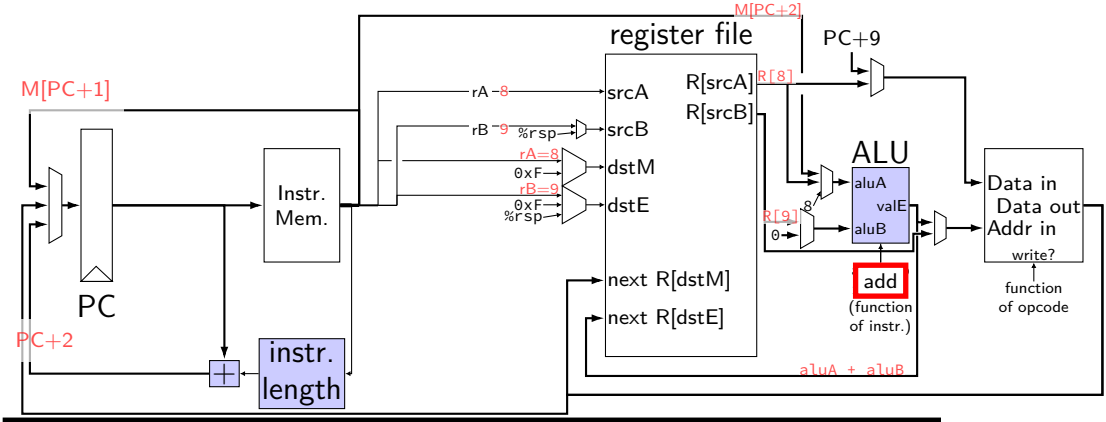
circuit: setting MUXes



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

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

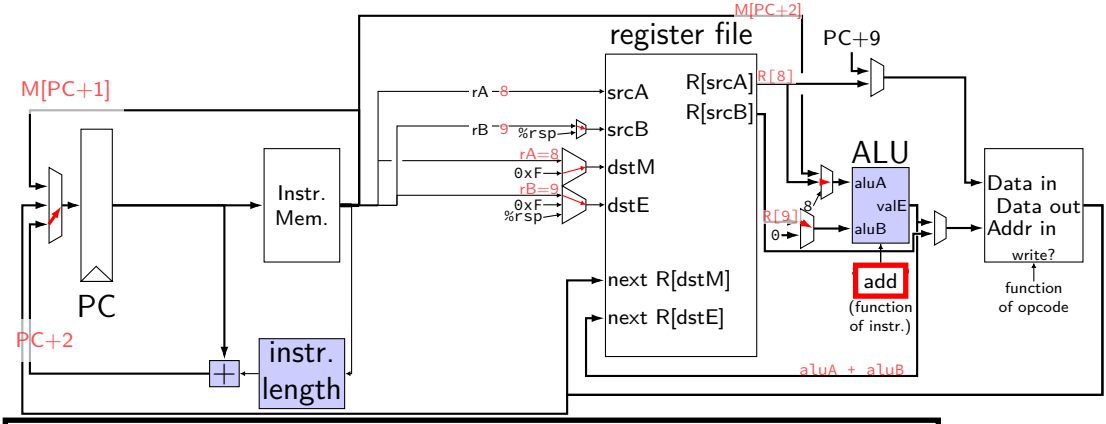
circuit: setting MUXes



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

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

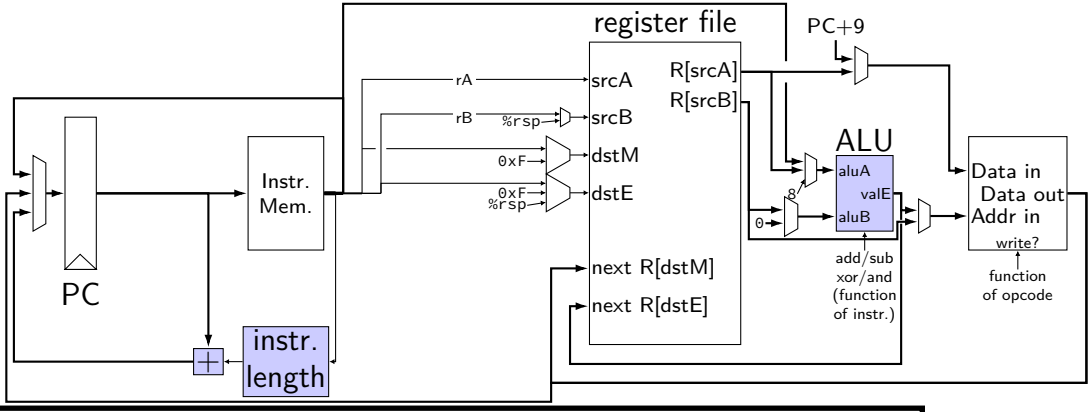
circuit: setting MUXes



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

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

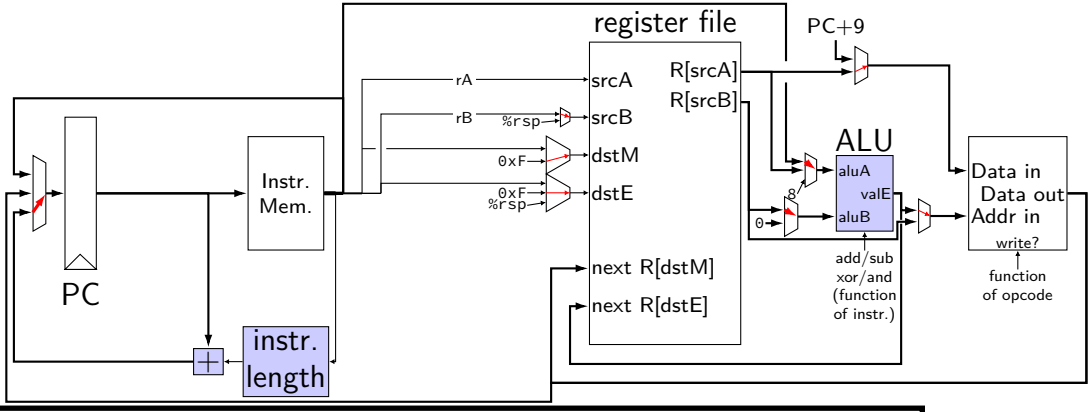
circuit: setting MUXes



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

Exercise: what do they select for `rmmovq`?

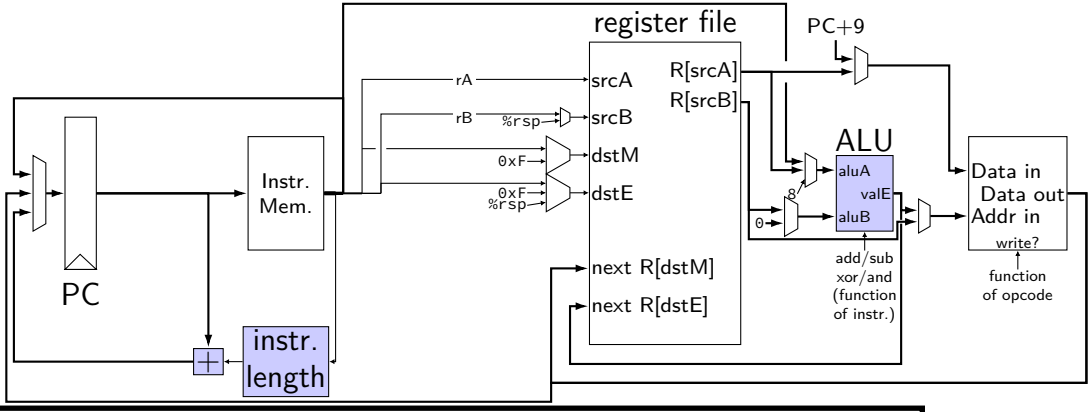
circuit: setting MUXes



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

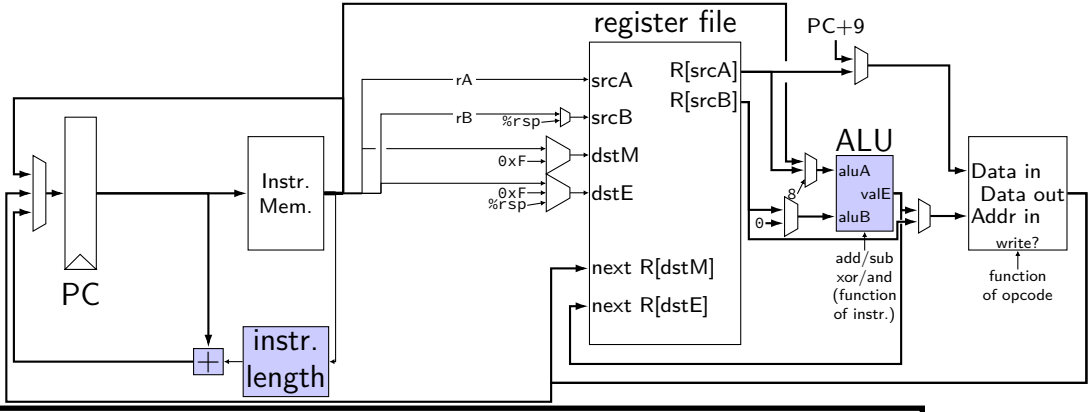
Exercise: what do they select for `rmmovq`?

circuit: setting MUXes



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

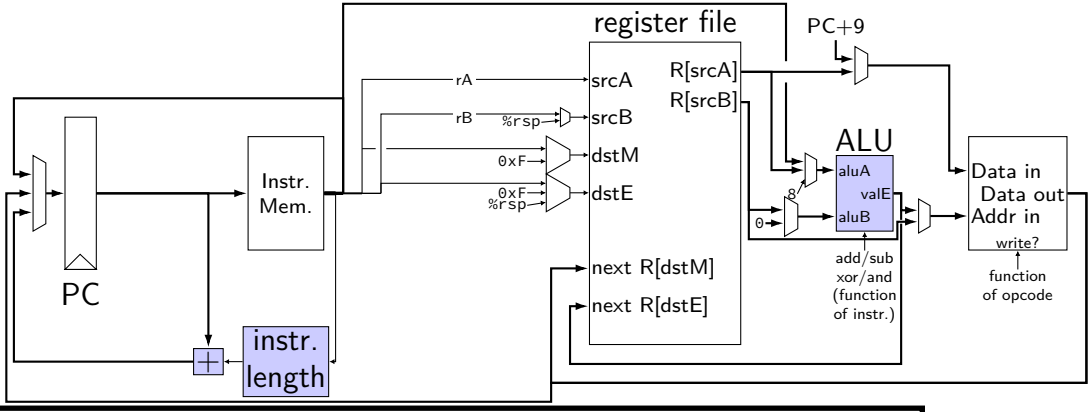
circuit: setting MUXes



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

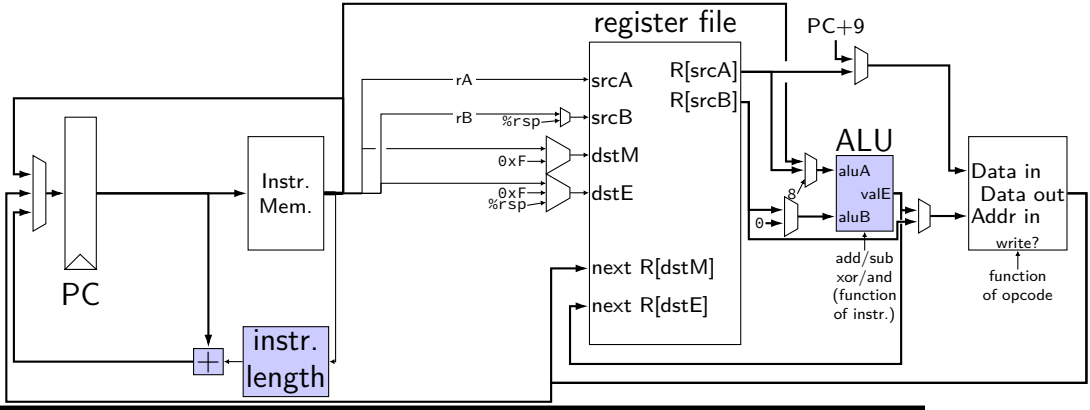
Exercise: what do they select for `mrmovq`?

circuit: setting MUXes



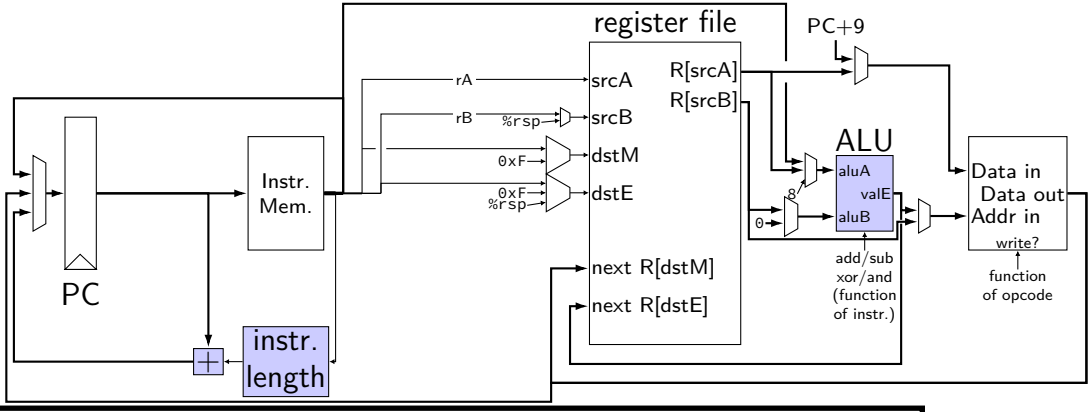
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `jle`?

circuit: setting MUXes



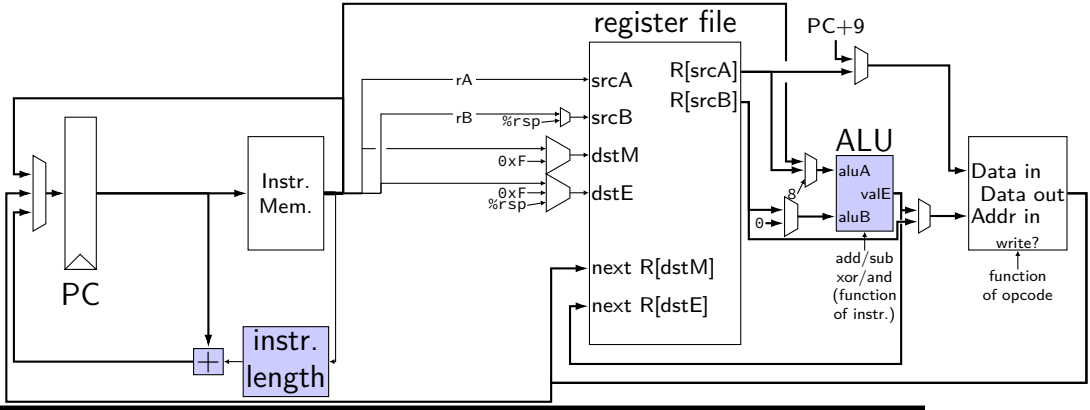
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `cmovle`?

circuit: setting MUXes



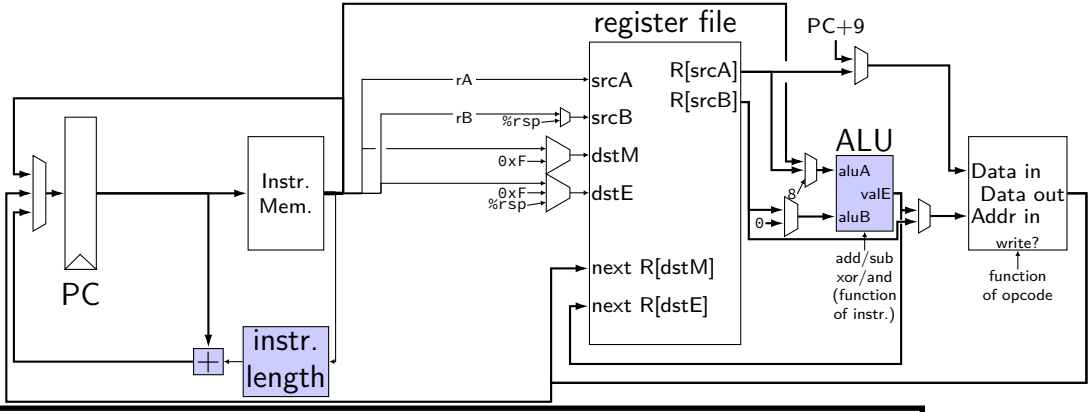
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn, dmemAddr, ...
Exercise: what do they select for `ret`?

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`?

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`** `i0`:

next value on `i_name`; current value on `0_name`

fixed functionality

register file (15 registers; 2 read + 2 write)
memories (data + instruction)
Stat register (start/stop/error)