# HCLRS / single cycle CPU part 2

# last time

clock signals and rising-edge triggered logic

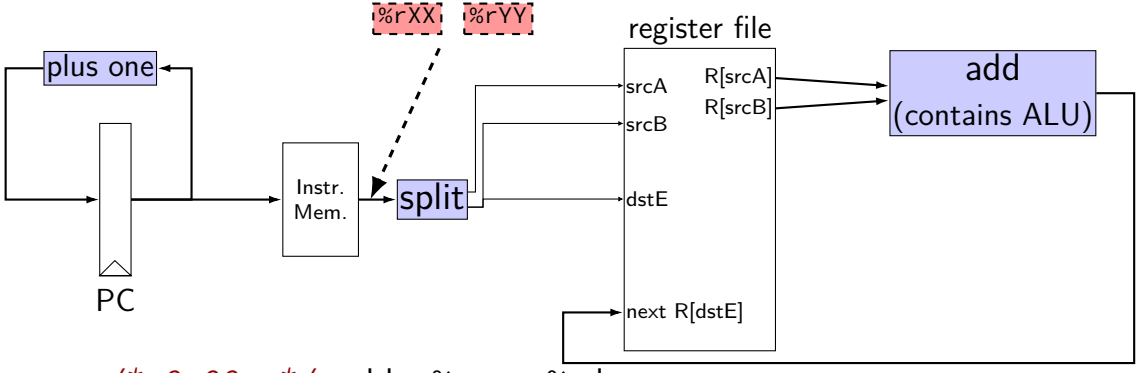
- register writes, register file writes, memory writes

decisions in hardware — MUXes

HCLRS syntax

building an jmp/nop/addq processor

# addq CPU




```
/* 0x00: */ addq %rax, %rdx
/* 0x01: */ addq %rbx, %rdx
```

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

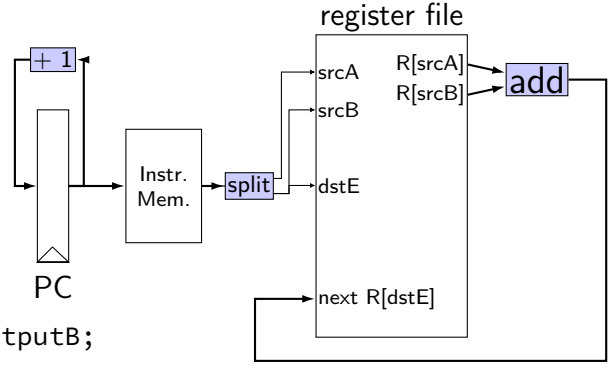
after cycle 1: PC = 0x01, rax = 10, rbx = 20, rdx = 40

after cycle 2: PC = 0x02, rax = 10, rbx = 20, rdx = 60

# demo: unpack HCLRS

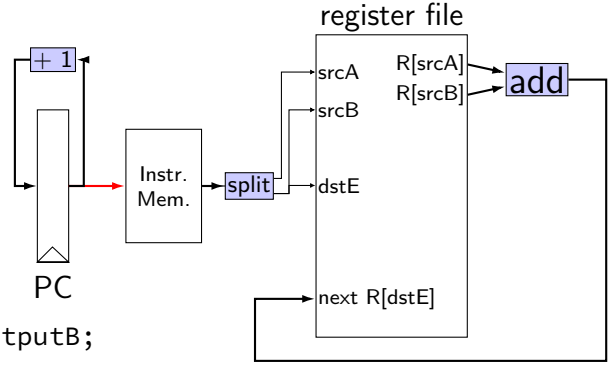
# addq in HCL

```
wire rA : 4, rB : 4, icode : 4;
register pP {
    thePC : 64 = 0;
}

pc = P_thePC;
p_thePC = P_thePC + 2;

icode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];

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

Stat = [ icode == OPQ: STAT_AOK; icode == HALT: STAT_HLT; 1: STAT_INS ];
```



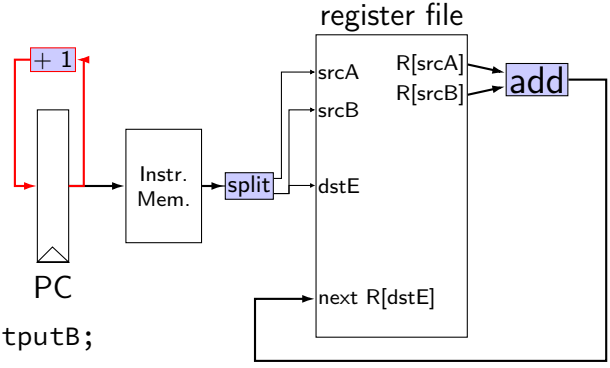
# addq in HCL

```
wire rA : 4, rB : 4, icode : 4;
register pP {
    thePC : 64 = 0;
}

pc = P_thePC;
p_thePC = P_thePC + 2;

icode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];

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

Stat = [ icode == OPQ: STAT_AOK; icode == HALT: STAT_HLT; 1: STAT_INS ];
```



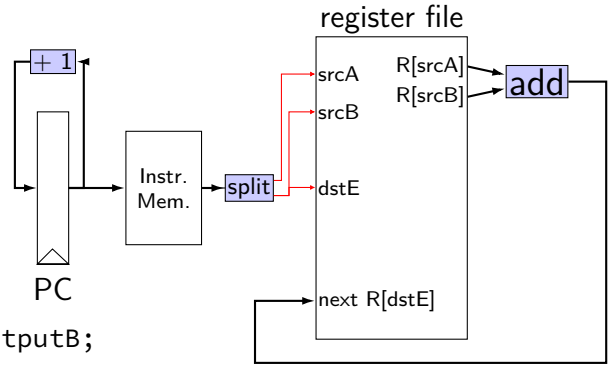
# addq in HCL

```
wire rA : 4, rB : 4, icode : 4;
register pP {
    thePC : 64 = 0;
}

pc = P_thePC;
p_thePC = P_thePC + 2;

icode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];

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

Stat = [ icode == OPQ: STAT_AOK; icode == HALT: STAT_HLT; 1: STAT_INS ];
```



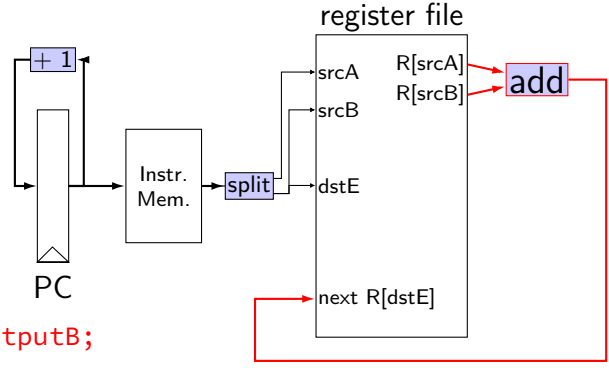
# addq in HCL

```
wire rA : 4, rB : 4, icode : 4;
register pP {
    thePC : 64 = 0;
}

pc = P_thePC;
p_thePC = P_thePC + 2;

icode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];

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

Stat = [ icode == OPQ: STAT_AOK; icode == HALT: STAT_HLT; 1: STAT_INS ];
```



# addq in HCL

```
wire rA : 4, rB : 4, icode : 4;
register pP {
    thePC : 64 = 0;
}

pc = P_thePC;
p_thePC = P_thePC + 2;

icode = i10bytes[4..8];
rA = i10bytes[12..16];
rB = i10bytes[8..12];

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

Stat = [ icode == OPQ: STAT_AOK; icode == HALT: STAT_HLT; 1: STAT_INS ];
```



# addq-test.ys → addq-test.yo

addq-test.ys:

```
addq %rax, %rbx
addq %rcx, %rdx
addq %rsi, %rdi
halt
```

---

then `make addq-test.yo` or `tools/yas addq-test.ys`

---

addq-test.yo:

```
0x000: 6003                 | addq %rax, %rbx
0x002: 6012                 | addq %rcx, %rdx
0x004: 6067                 | addq %rsi, %rdi
0x006: 00                   | halt
```

# running addq-test.yo on addq.hcl (demo)

# running with -d: debugging

# 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` — stop, 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`

# 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

# 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

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

book uses names like `valC` — not required!

- author's environment limited adding new wires

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

implement your own ALU

# differences from book

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

book uses names like `valC` — not required!

author's environment limited adding new wires

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

implement your own ALU

# differences from book

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

book uses names like `valC` — not required!

author's environment limited adding new wires

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

implement your own ALU

# 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 `o_name`

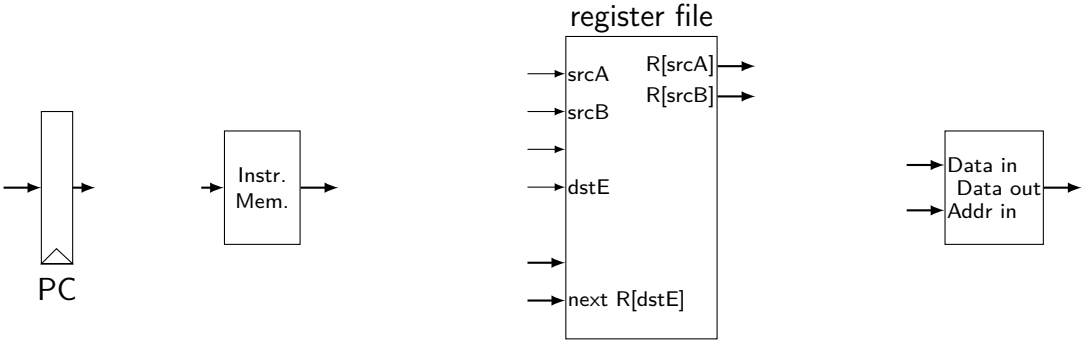
fixed functionality

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

# simple ISA 4: mov-to-register

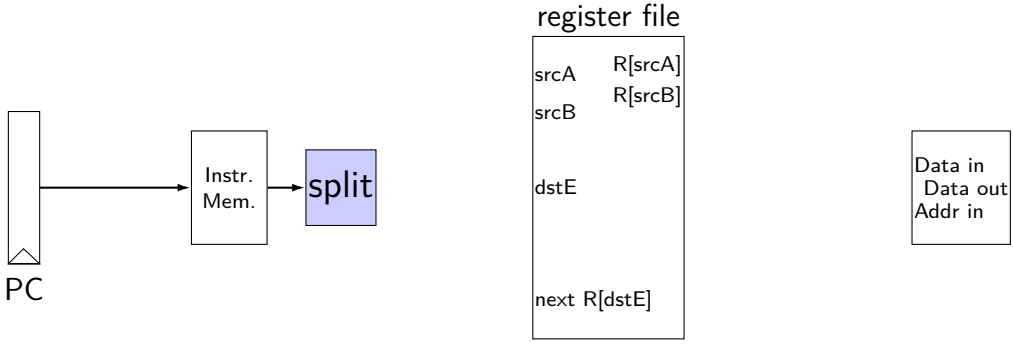
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

# mov-to-register CPU



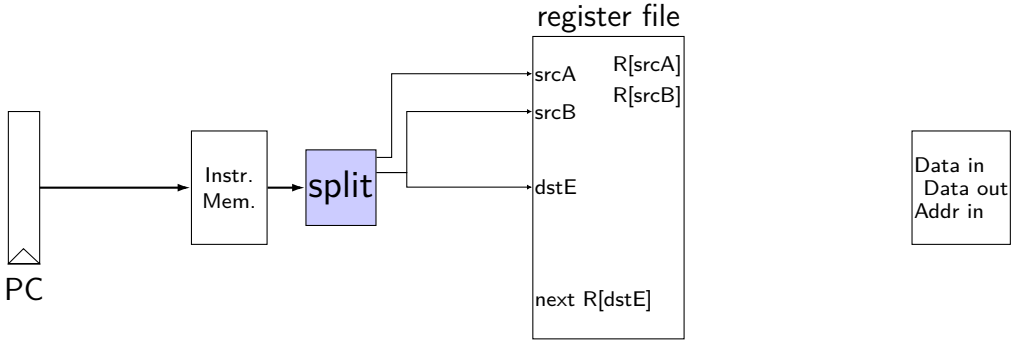

| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` *rA*, *rB* | 2 | 0 | *rA* | *rB* | |
| `irmovq` *V*, *rB* | 3 | 0 | F | *rB* | *V* |
| `mrmovq` *D*(*rB*), *rA* | 5 | 0 | *rA* | *rB* | *D* |

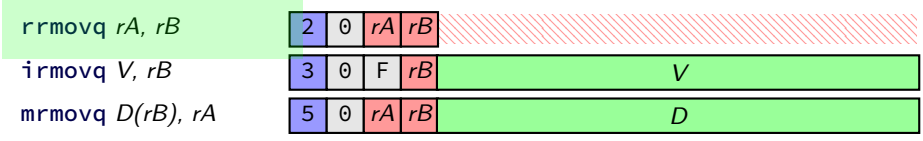
# mov-to-register CPU



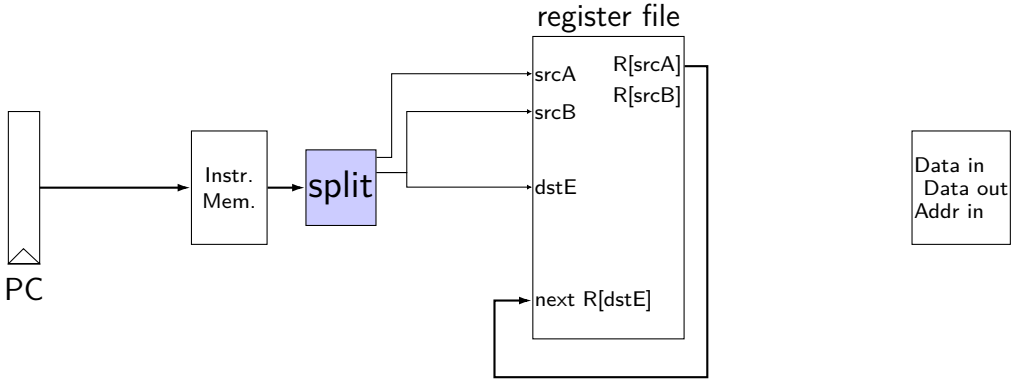

| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` *rA*, *rB* | 2 | 0 | *rA* | *rB* | |
| `irmovq` *V*, *rB* | 3 | 0 | F | *rB* | *V* |
| `mrmovq` *D*(*rB*), *rA* | 5 | 0 | *rA* | *rB* | *D* |

# mov-to-register CPU



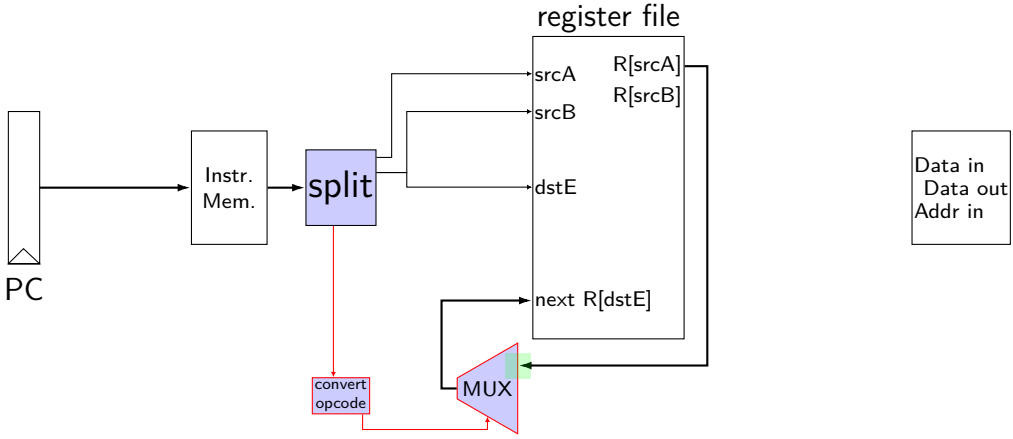



# mov-to-register CPU



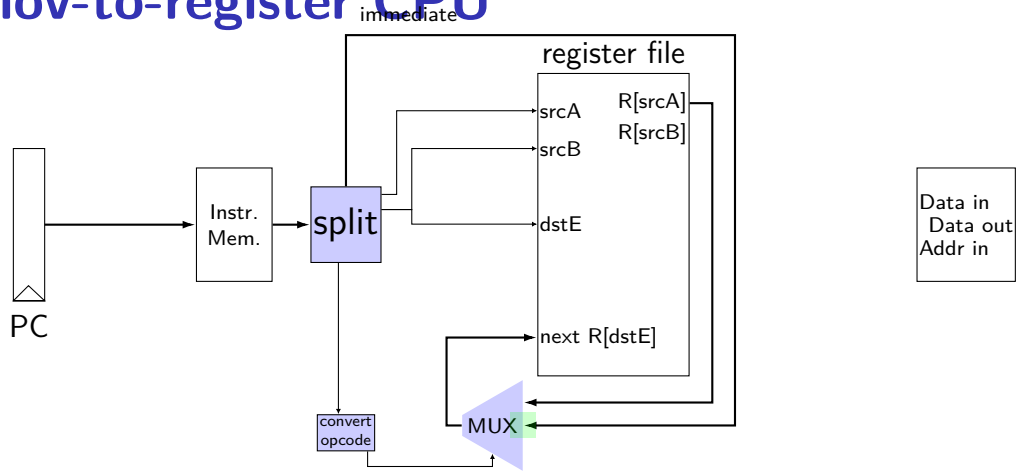

| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` *rA*, *rB* | 2 | 0 | *rA* | *rB* | |
| `irmovq` *V*, *rB* | 3 | 0 | F | *rB* | *V* |
| `mrmovq` *D*(*rB*), *rA* | 5 | 0 | *rA* | *rB* | *D* |

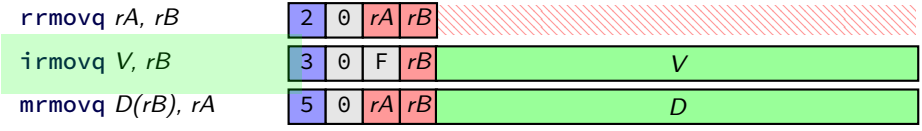
# mov-to-register CPU



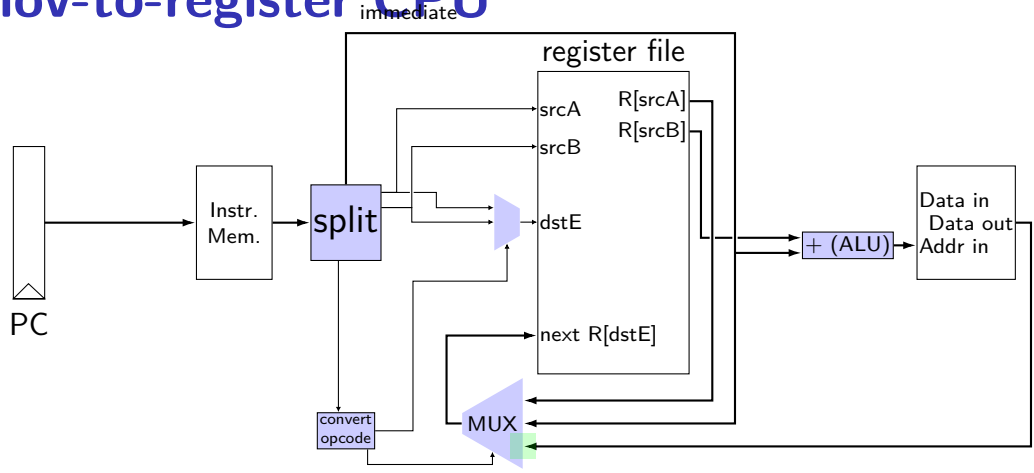

| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` *rA*, *rB* | 2 | 0 | *rA* | *rB* | |
| `irmovq` *V*, *rB* | 3 | 0 | F | *rB* | *V* |
| `mrmovq` *D*(*rB*), *rA* | 5 | 0 | *rA* | *rB* | *D* |

# mov-to-register CPU



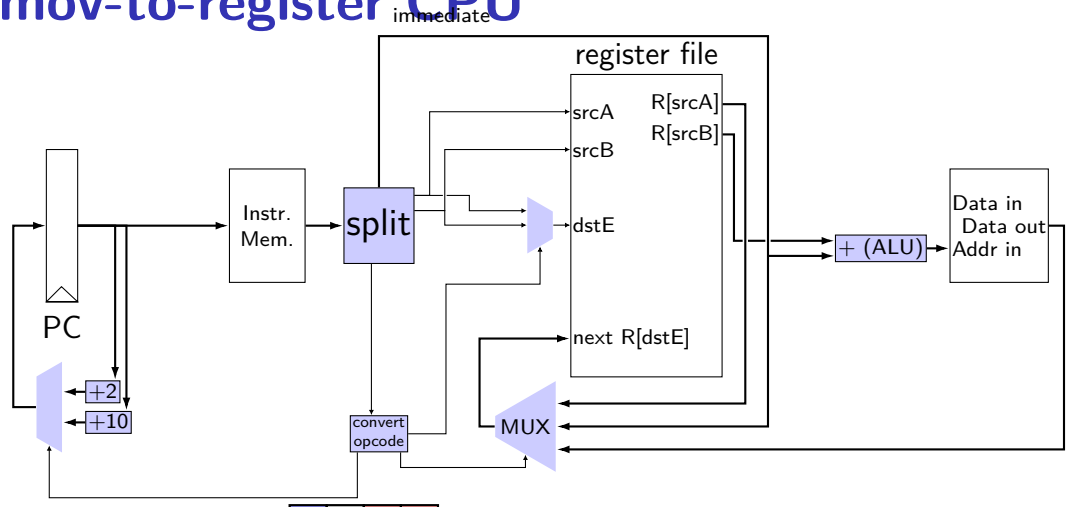



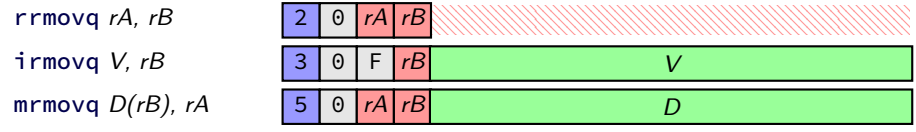
# mov-to-register CPU




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

# mov-to-register CPU

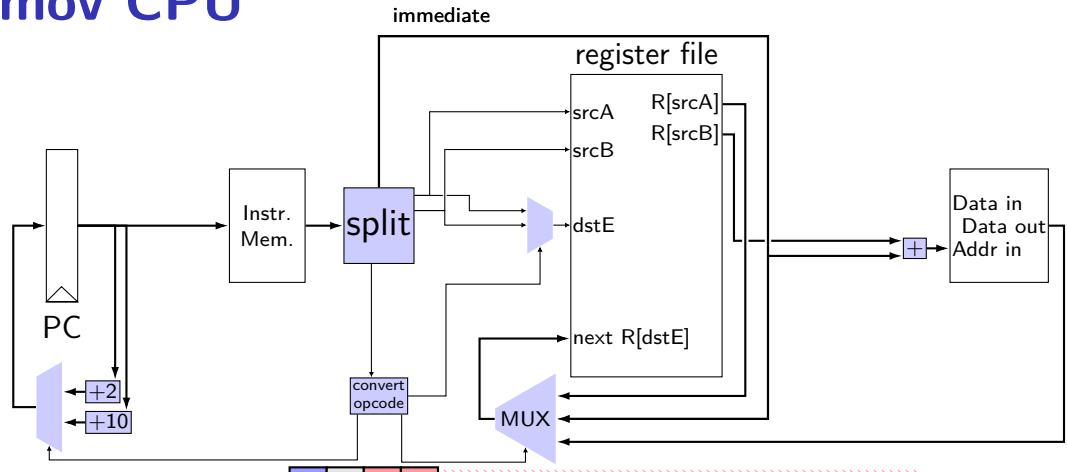


# simple ISA 4B: mov

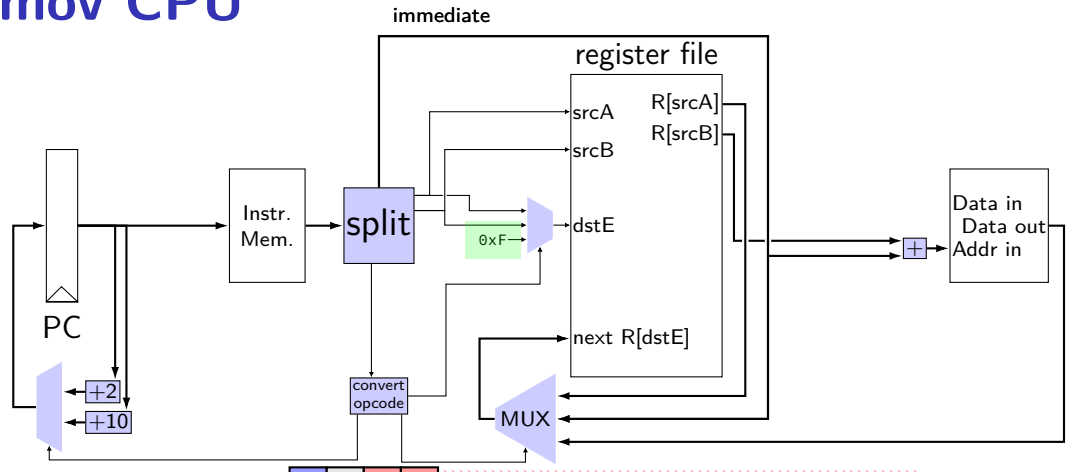
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

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

# mov CPU



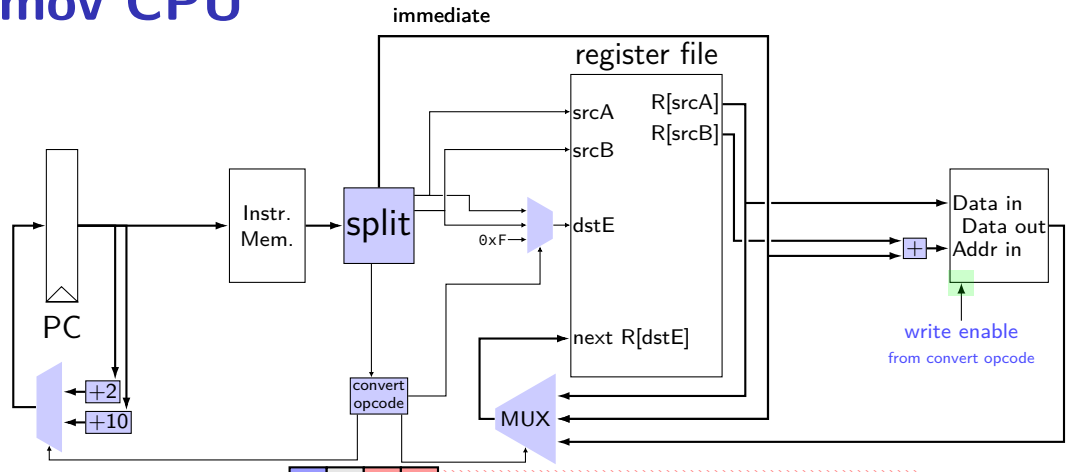

| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` *rA, rB* | 2 | 0 | *rA* | *rB* | |
| `irmovq` *V, rB* | 3 | 0 | F | *rB* | *V* |
| `mrmovq` *D(rB), rA* | 5 | 0 | *rA* | *rB* | *D* |
| `rmmovq` *rA, D(rB)* | 4 | 0 | *rA* | *rB* | *D* |

# mov CPU




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

# mov CPU




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

# data path versus control path

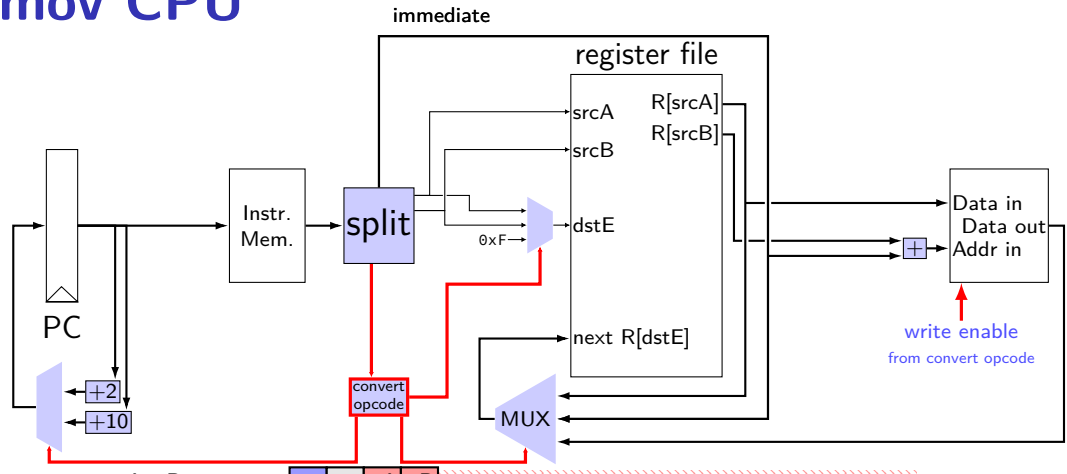
data path — signals carrying "actual data"

control path — signals that control MUXes, etc.

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

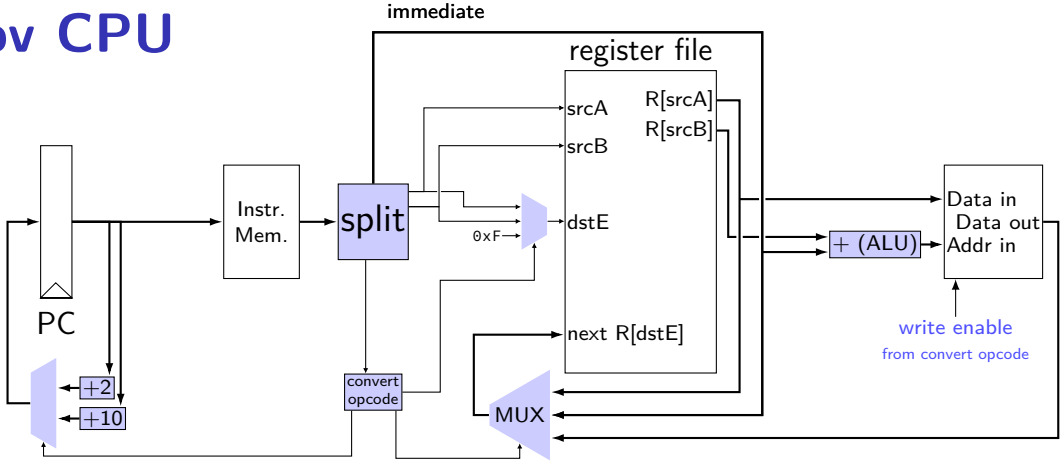
we will often omit parts of the control path in drawings, etc.

# mov CPU

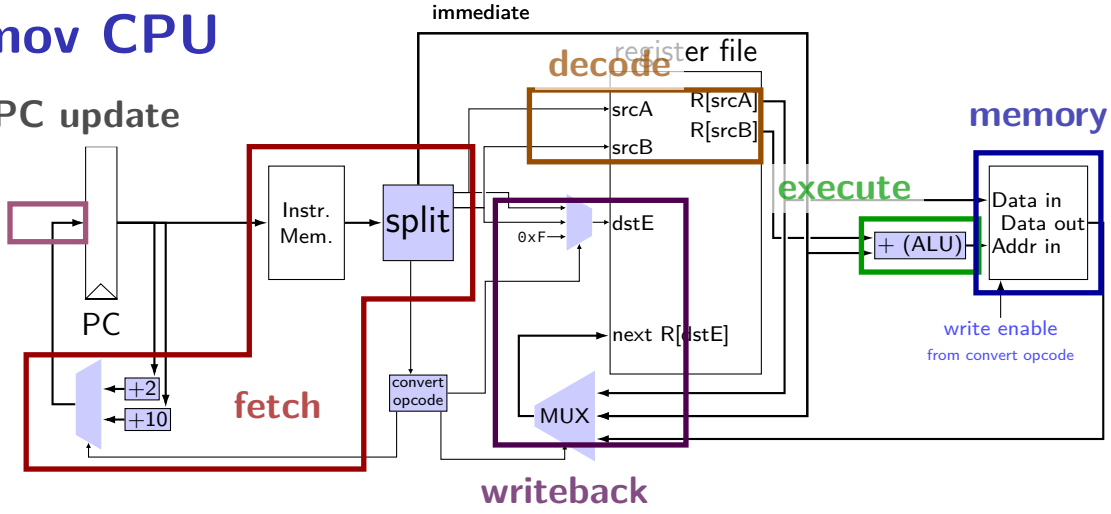



| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` *rA, rB* | 2 | 0 | *rA* | *rB* | |
| `irmovq` *V, rB* | 3 | 0 | F | *rB* | *V* |
| `mrmovq` *D(rB), rA* | 5 | 0 | *rA* | *rB* | *D* |
| `rmmovq` *rA, D(rB)* | 4 | 0 | *rA* | *rB* | *D* |

# 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

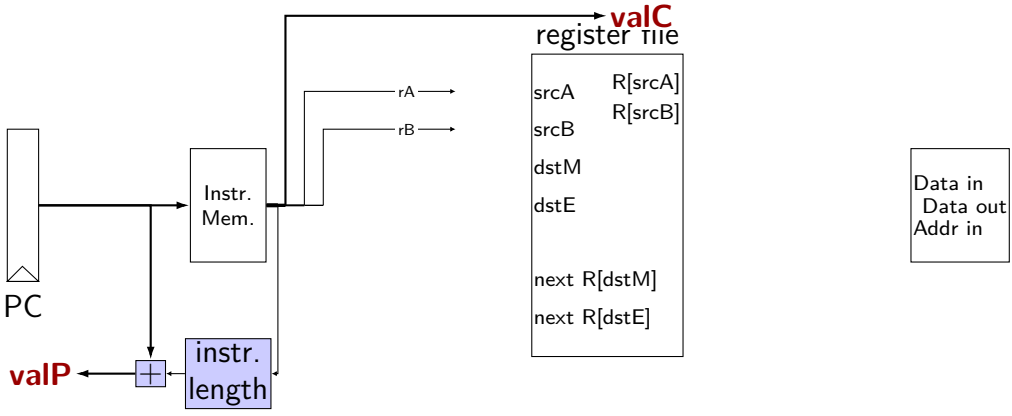
split into seperate wires:

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

compute next instruction address:

- valP — PC + (instr length)

# instruction fetch

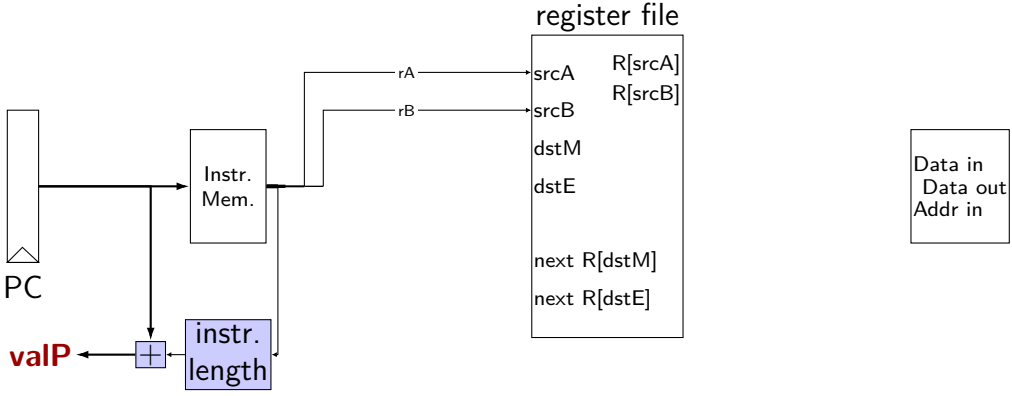


# SEQ: instruction “decode”

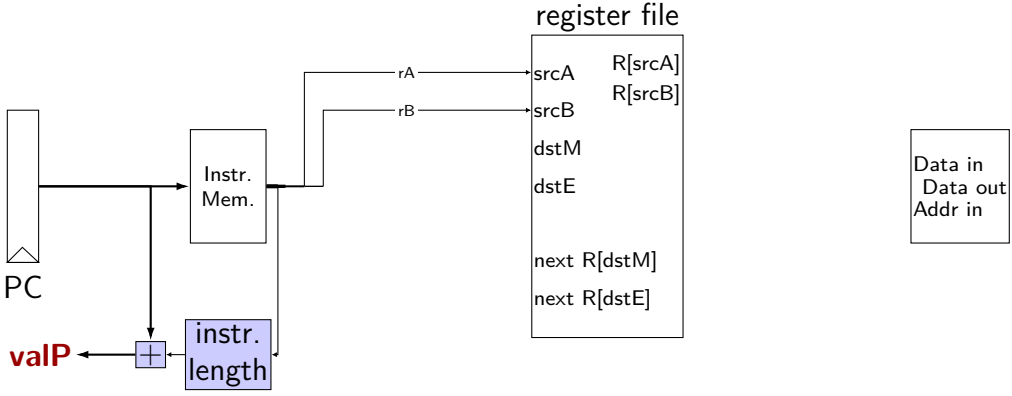
read registers

valA, valB — register values

# instruction decode (1)



# instruction decode (1)




exercise: which of these instructions can this **not** work for?
nop, addq, `mrmovq`, pushq, `jmp`,

# SEQ: srcA, srcB

always read rA, rB?

Problems: (not planned to be included our assignments)

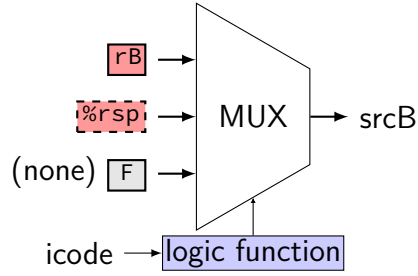
- push rA
- pop
- call
- ret

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

MUX controlled by icode

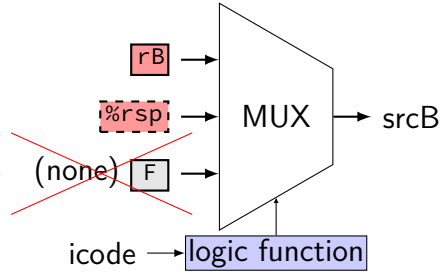
# SEQ: possible registers to read

| 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 |

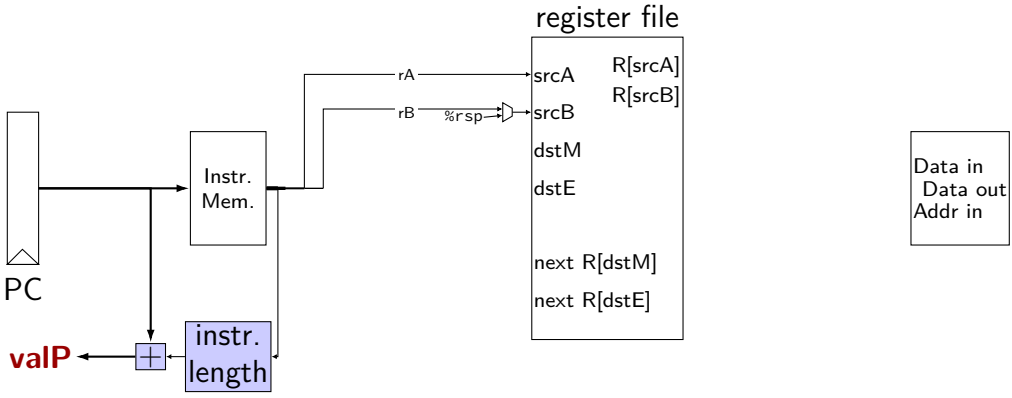


# SEQ: possible registers to read

| 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)

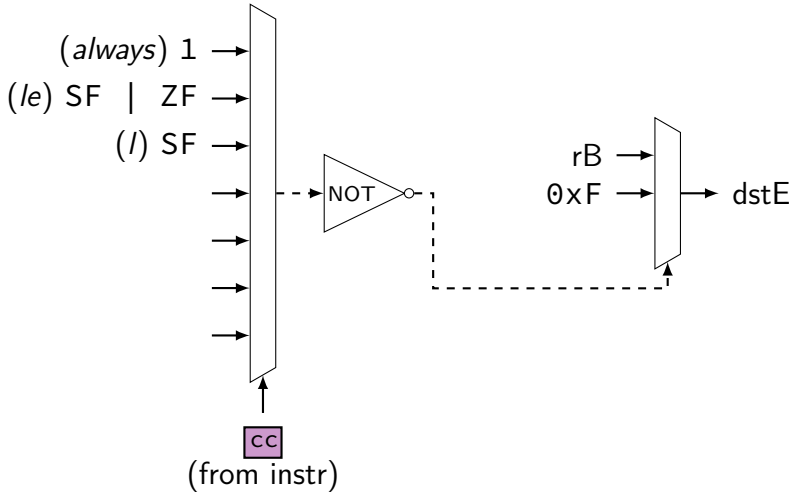
- valE — ALU output

read prior condition codes

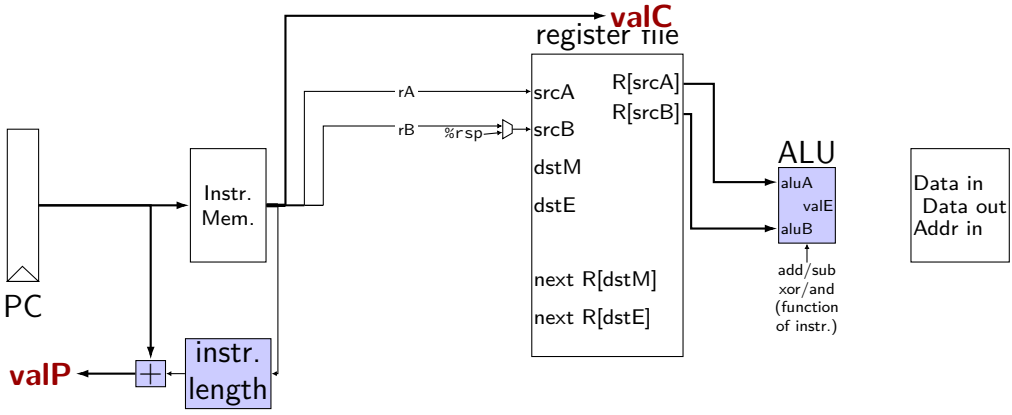
- Cnd — condition codes based on ifun (instruction type for jCC/cmovCC)

write new condition codes

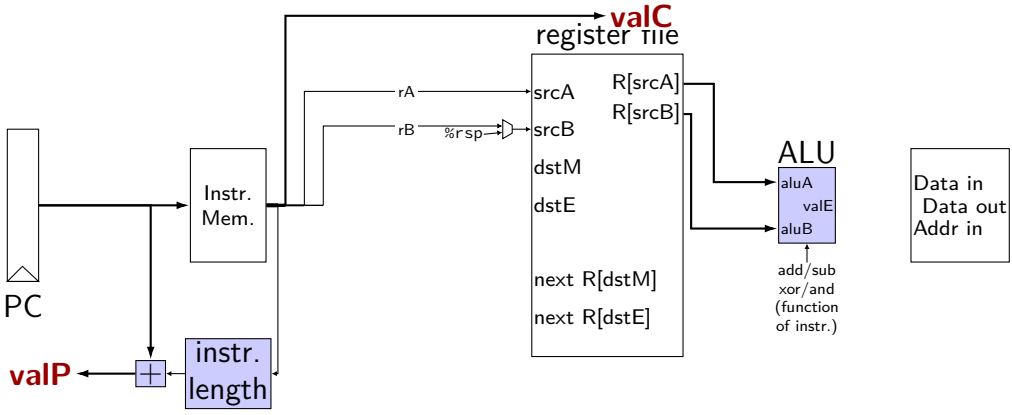
# using condition codes: cmov



# execute (1)


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

# execute (1)



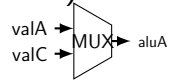

exercise: which of these instructions can this **not** work for?
`nop`, `addq`, `mrmovq`, `popq`, `call`,

# SEQ: ALU operations?

ALU inputs always valA, valB (register values)?

no, inputs from instruction: (Displacement + rB)

- `mrmovq`
- `rmmovq`



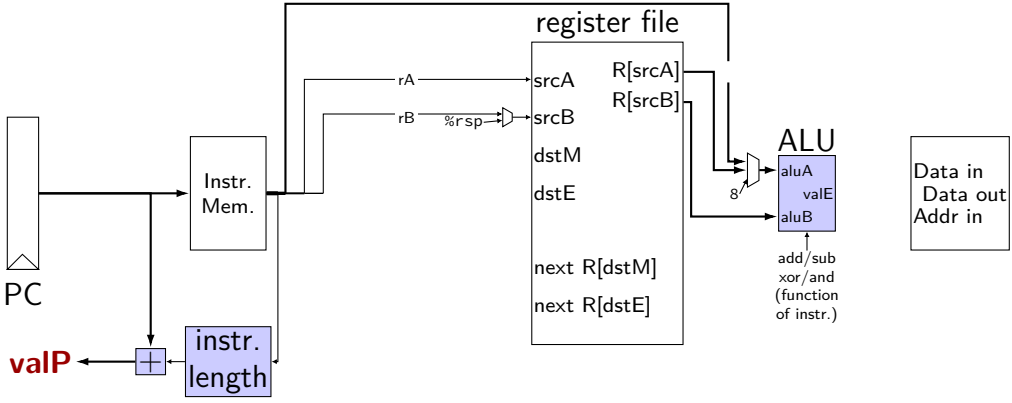

no, constants: (rsp +/- 8) (not planned to be in assignments)

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

extra signals: aluA, aluB

computed ALU input values

# execute (2)

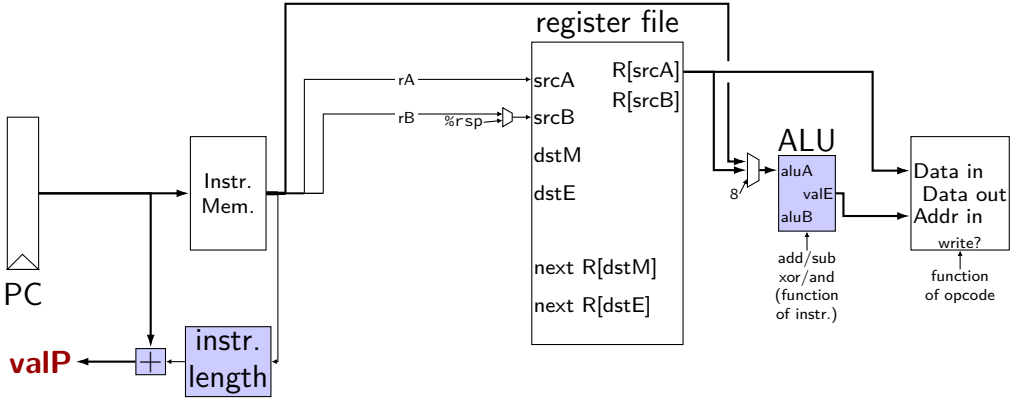

register file
rA
rB
%rsp
srcA
srcB
dstM
dstE
R[srcA]
R[srcB]
next R[dstM]
next R[dstE]
ALU
aluA
valE
aluB
8
add/sub
xor/and
(function
of instr.)
Data in
Data out
Addr in
Instr.
Mem.
PC
valP
instr.
length

# SEQ: Memory

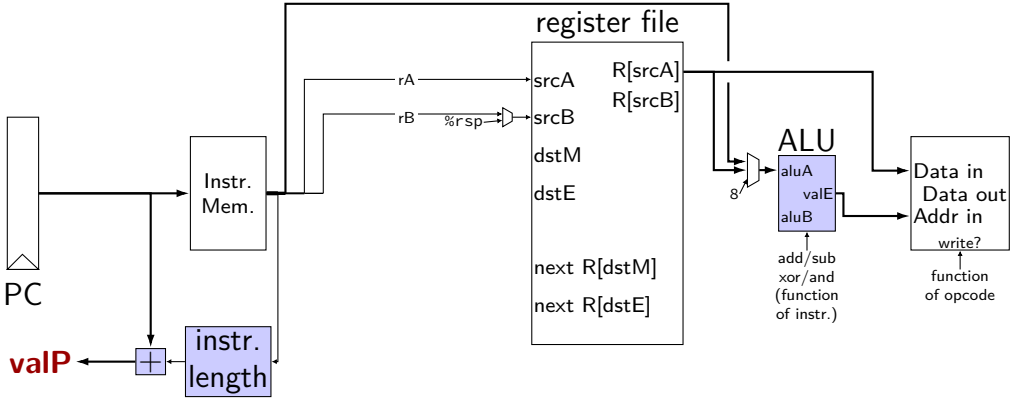
read or write data memory

valM — value read from memory (if any)

# memory (1)


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

# memory (1)




exercise: which of these instructions can this **not** work for?
`nop`, `rmmovq`, `mrmovq`, `popq`, `call`,

# SEQ: control signals for memory

read/write — read enable? write enable?

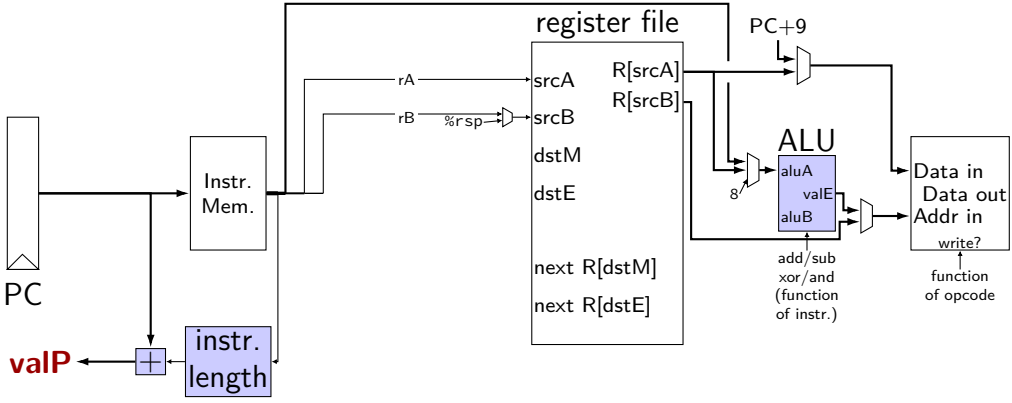
Addr — address

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

Data — value to write

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

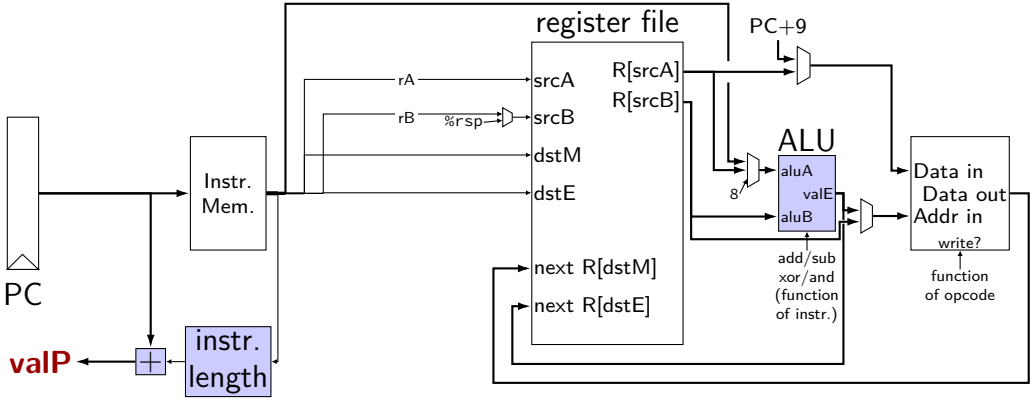
# memory (2)


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

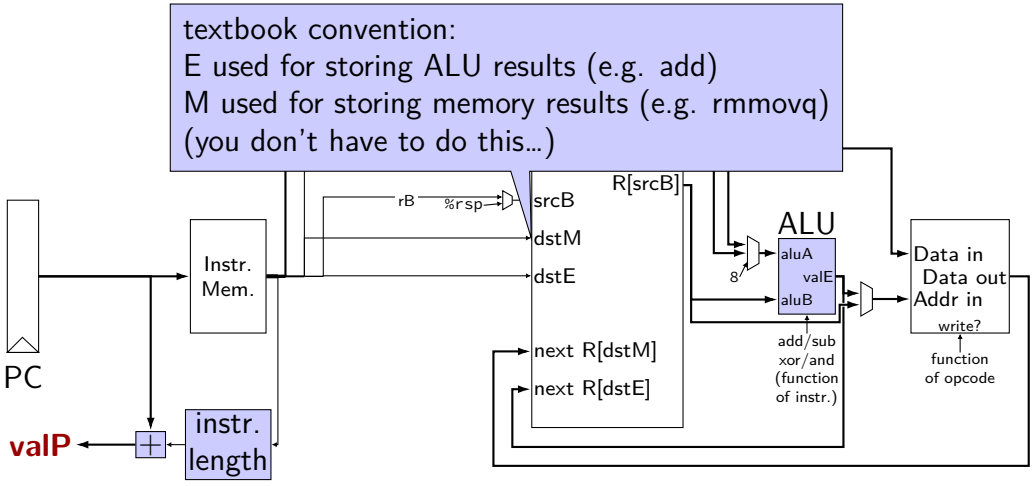
# SEQ: write back

write registers

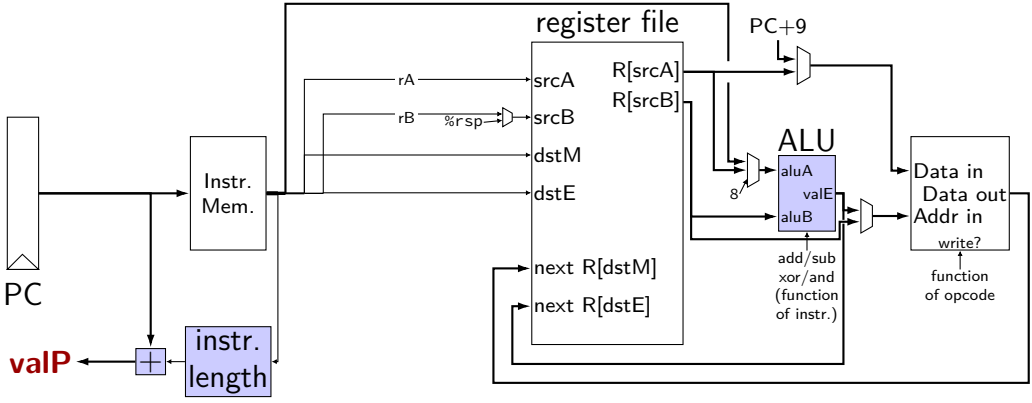
# write back (1)


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

# write back (1)


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 can this **not** work for?
`nop`, `irmovq`, `mrmovq`, `rmmovq`, `addq`

# SEQ: control signals for WB

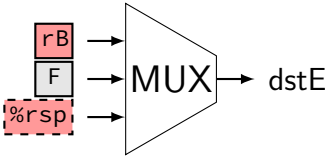
two write inputs — two needed by popq

valM (memory output), valE (ALU output)

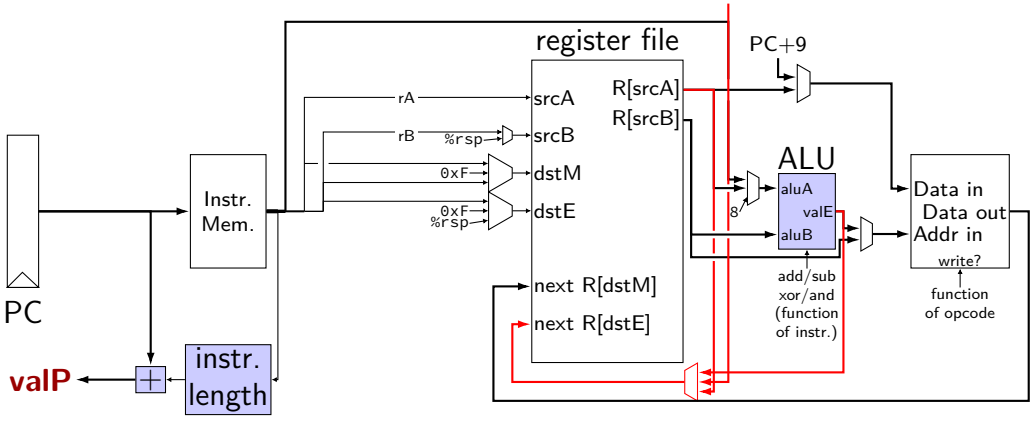
two register numbers

dstM, dstE

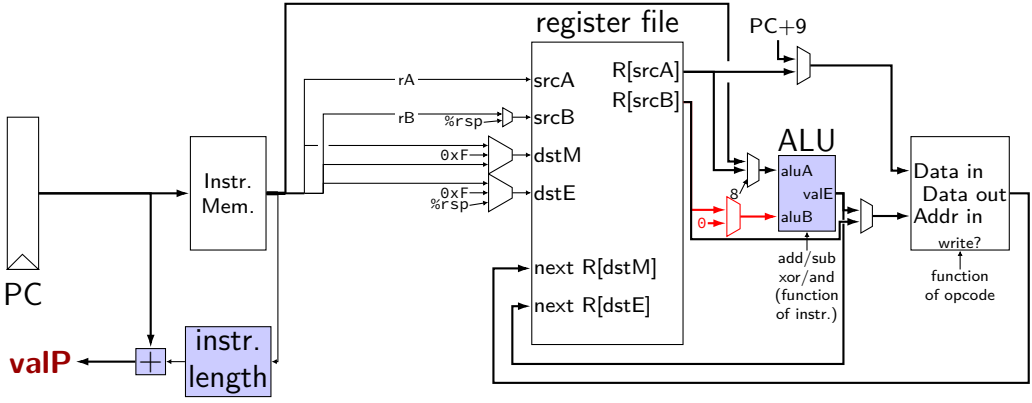
write disable — use dummy register number `0xF`



# write back (2a)



# write back (2b)

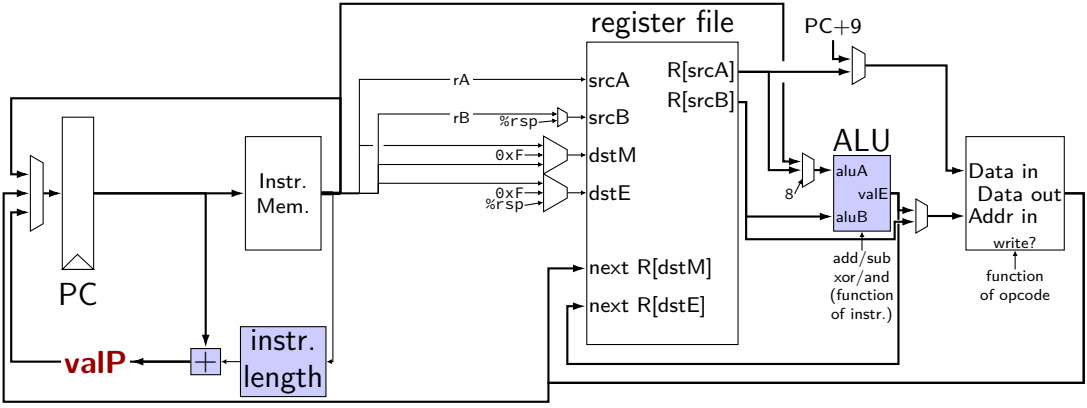


# SEQ: Update PC

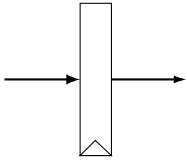
choose value for PC next cycle (input to PC register)

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

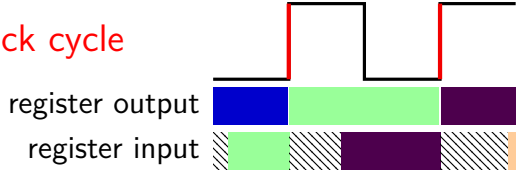
# PC update


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

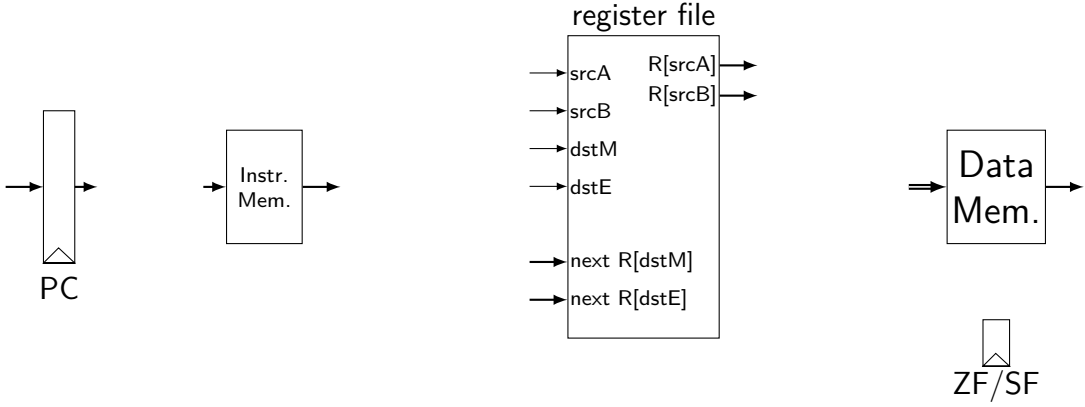
# registers



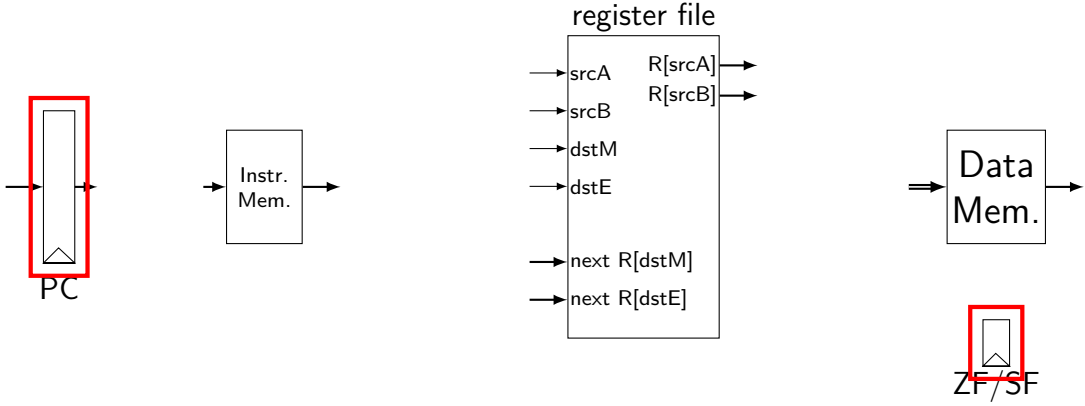

updates every clock cycle



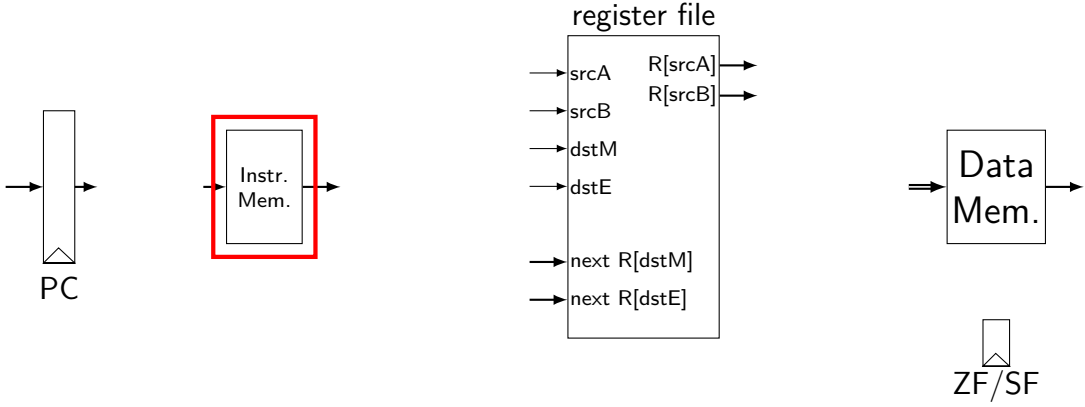
# state in Y86-64



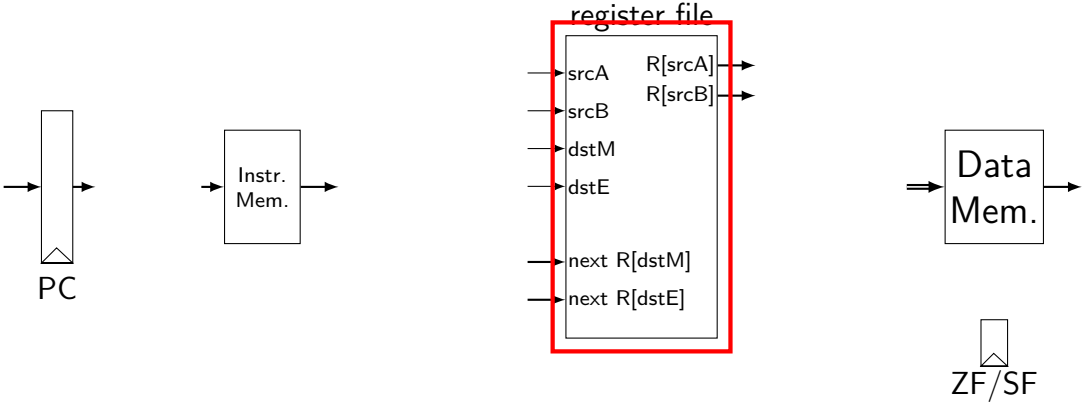
# state in Y86-64



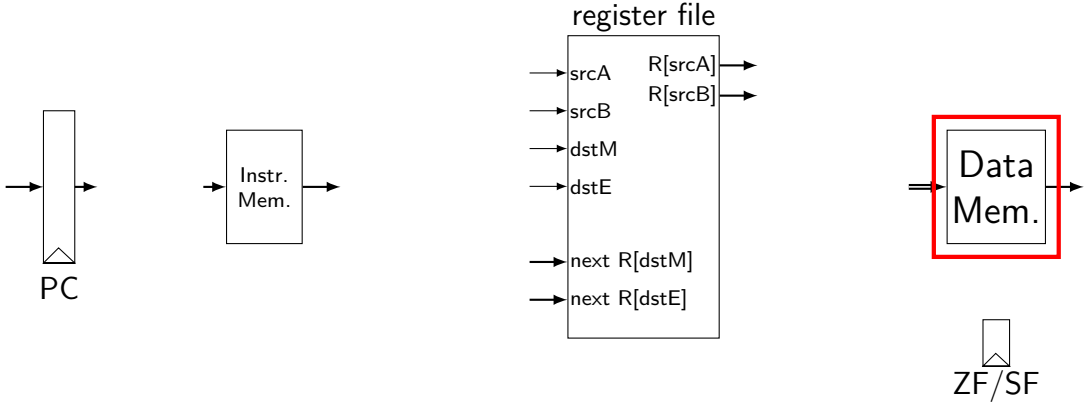
# state in Y86-64



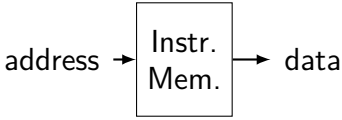
# state in Y86-64

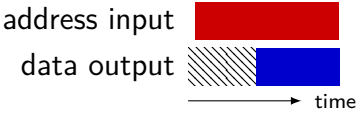


# state in Y86-64

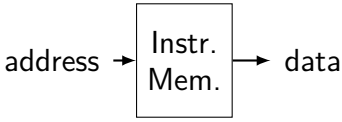


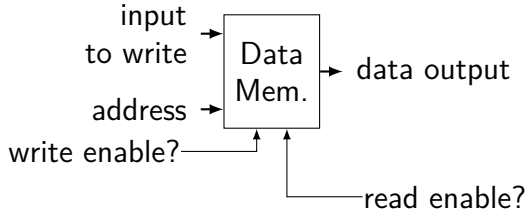
# memories

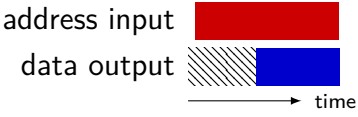


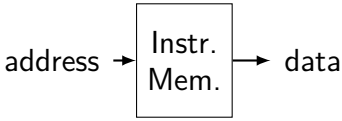
# memories

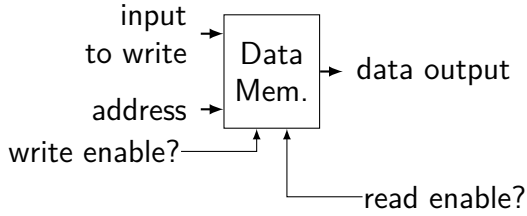

address
Instr.
Mem.
data

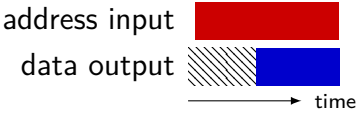

input
to write
address
write enable?
Data
Mem.
data output
read enable?

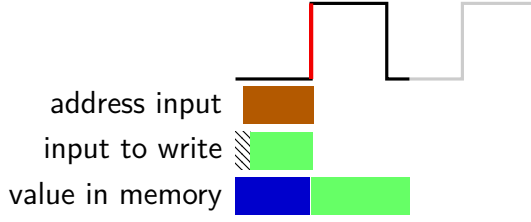

address input
data output
time

# memories

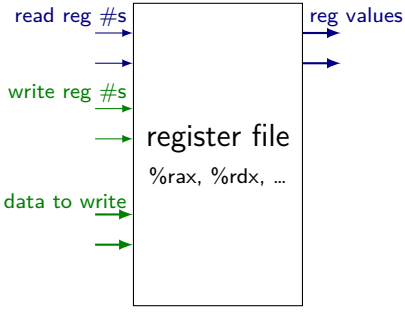

address
Instr.
Mem.
data


input
to write
address
write enable?
Data
Mem.
data output
read enable?

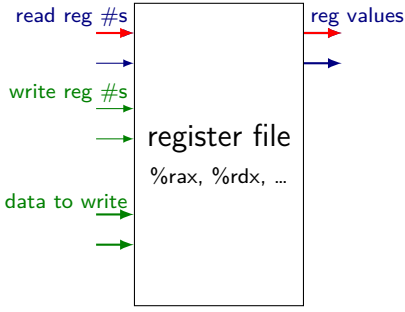

address input
data output
time

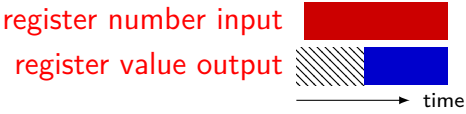

address input
input to write
value in memory

# register file

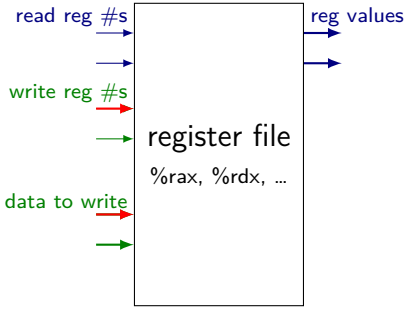


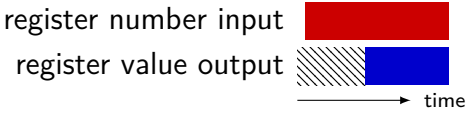
# register file

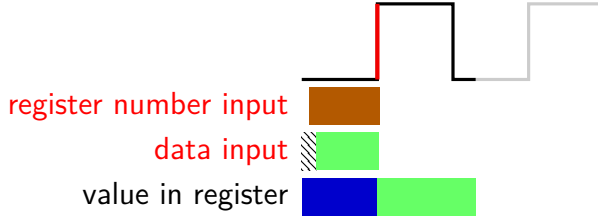


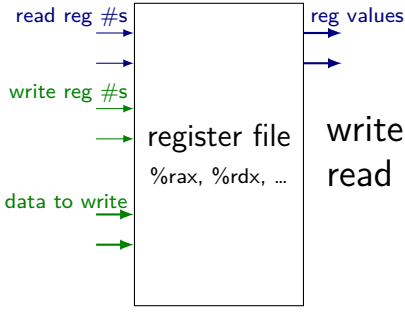
# register file



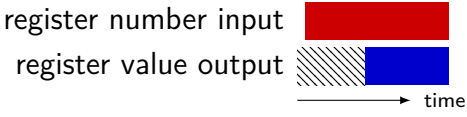



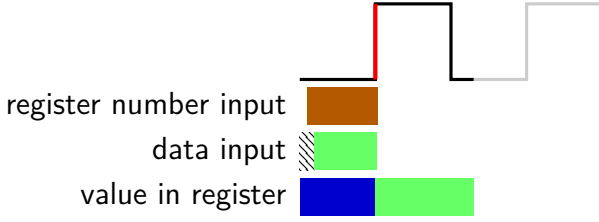



# register file

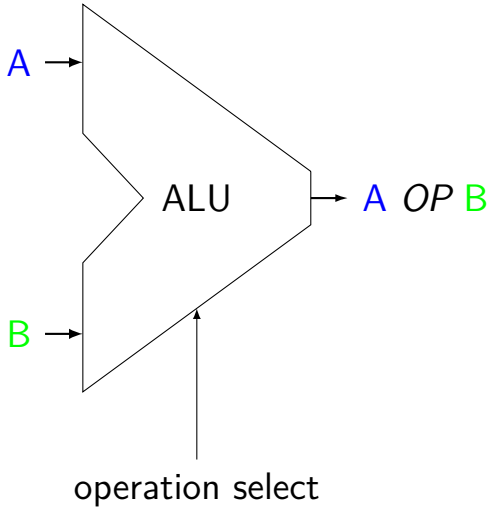



write register #15: write is ignored
read register #15: value is always 0



# ALUs




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

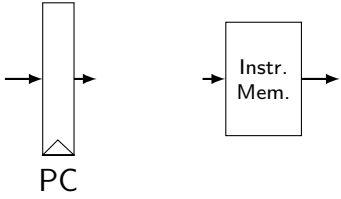
# Simple ISA 2: jmp

`jmp label`

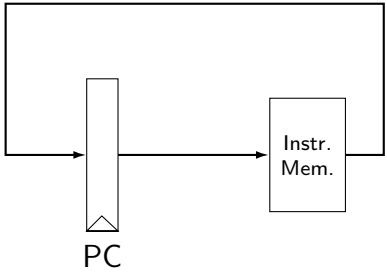
encoding: *8-byte little-endian address*

- 8 byte instructions, no opcode

# jmp CPU



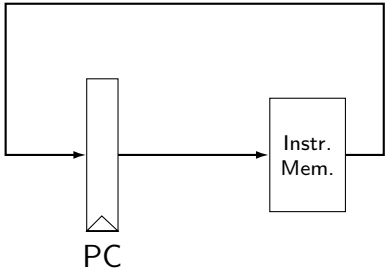
# jmp CPU



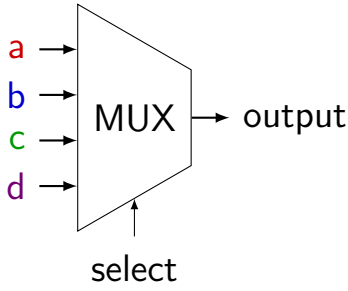
# jmp CPU

```
/* 0x00: */ jmp 0x10
/* 0x08: */ jmp 0x00
/* 0x10: */ jmp 0x08
```

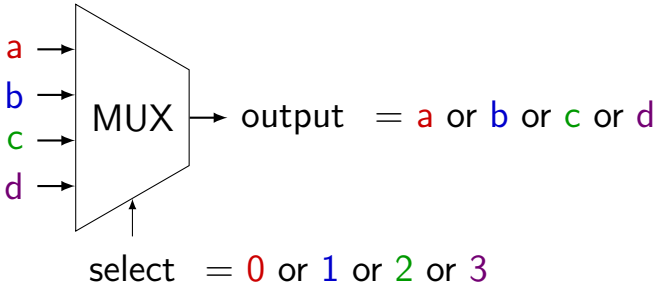
initially: PC = 0x00
after cycle 1: PC = 0x10
after cycle 2: PC = 0x08
after cycle 3: PC = 0x00



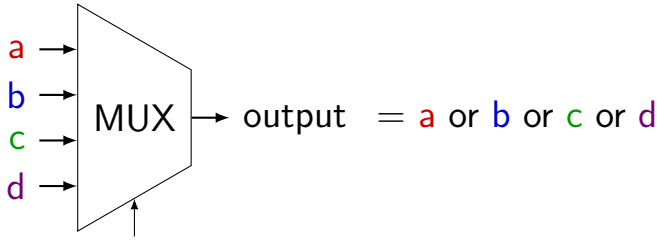
# multiplexers



# 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 |

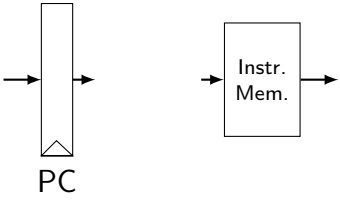
# Simple ISA 3: Jmp or No-Op

actual subset of Y86-64

`jmp LABEL` — encoded as `0x70` + address

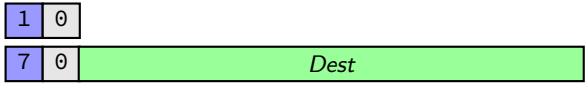
`nop` — encoded as `0x10`

# jmp+nop CPU

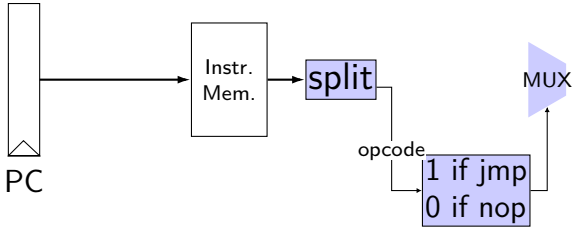


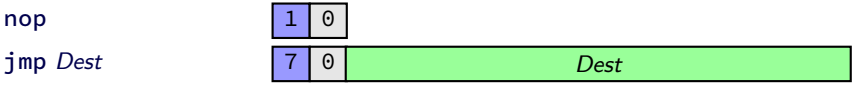

`nop`

`jmp` *Dest*

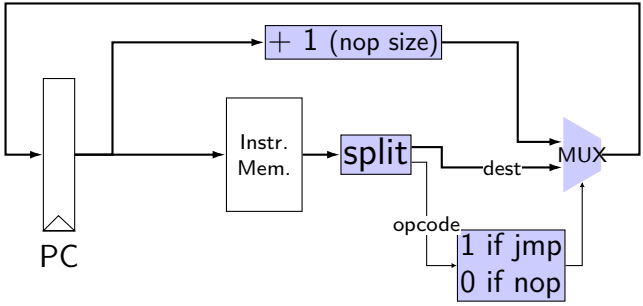


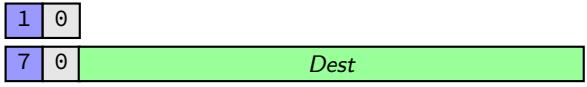
# jmp+nop CPU


Instr.
Mem.
split
MUX
PC
opcode
1 if jmp
0 if nop


nop
jmp Dest
1
0
7
0
Dest

# jmp+nop CPU

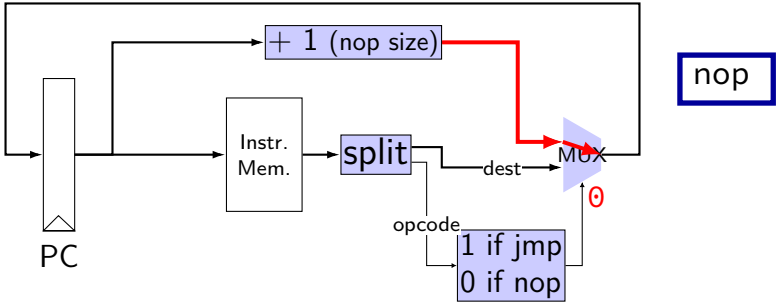


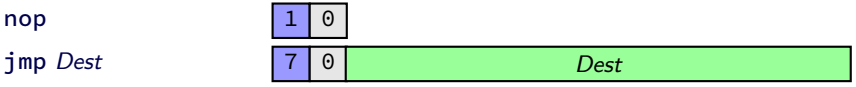




`nop`

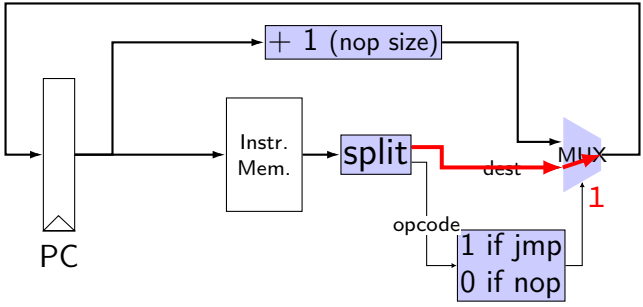
`jmp` *Dest*

# jmp+nop CPU


+ 1 (nop size)
nop
Instr.
Mem.
split
dest
MUX
0
opcode
1 if jmp
0 if nop
PC


nop
jmp Dest
1
0
7
0
Dest

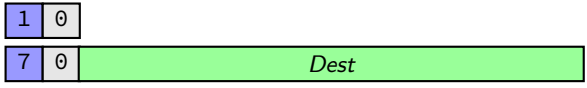
# jmp+nop CPU



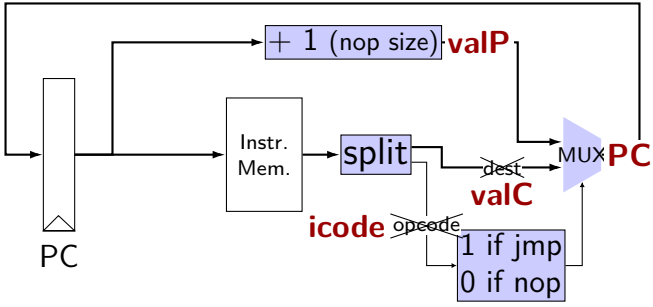

```
jmp dest
```

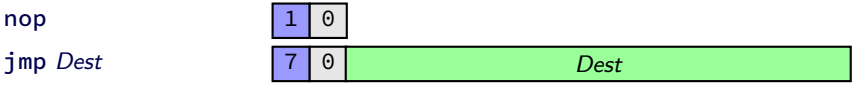
nop

jmp *Dest*

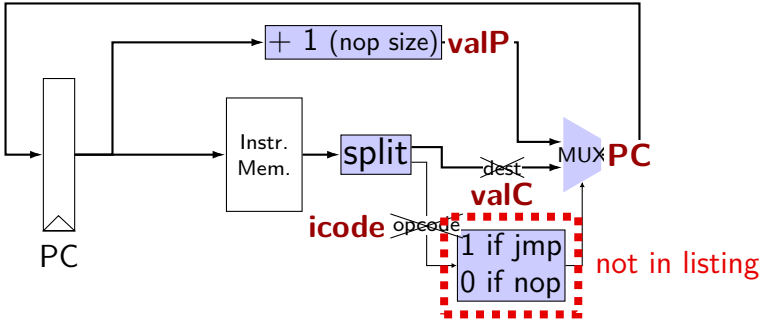


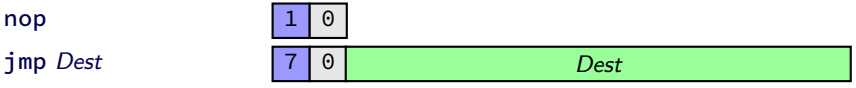
# jmp+nop CPU


+ 1 (nop size)
valP
Instr.
Mem.
split
dest
MUX
PC
valC
icode
opcode
1 if jmp
0 if nop
PC


nop
jmp Dest
1
0
7
0
Dest

# jmp+nop CPU


+ 1 (nop size)
valP
Instr.
Mem.
split
dest
MUX
PC
valC
icode
opcode
1 if jmp
0 if nop
not in listing
PC


nop
jmp Dest
1
0
7
0
Dest

# bit numbers and instructions

value from instruction memory in `i10bytes`

HCLRS numbers bits from LSB to MSB

80-bit integer, little-endian order:

first byte is least significant byte

HCLRS bit '0' is least significant bit

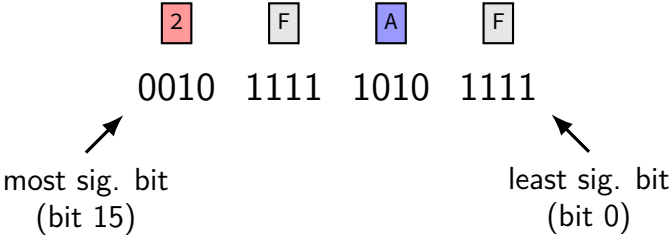
# example

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

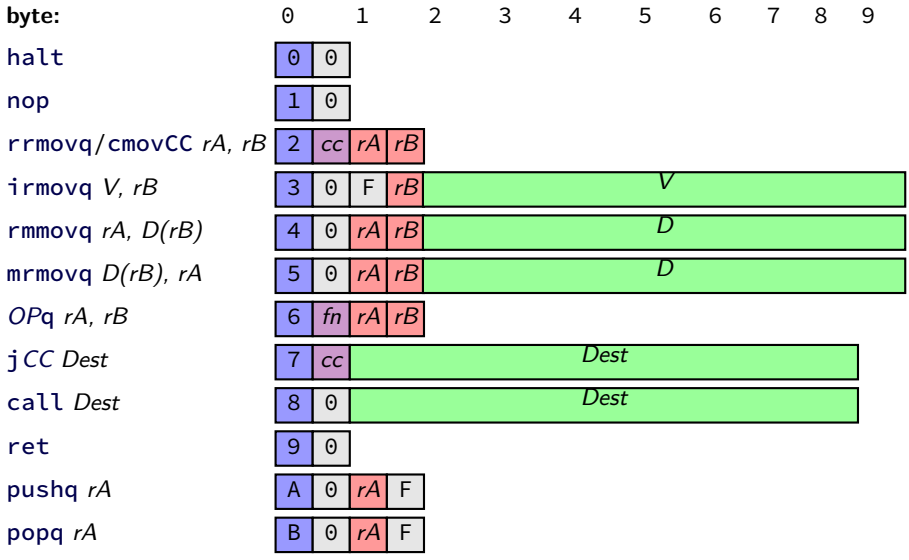
memory at $x + 0$: pushq F; at $x + 1$: rbx F

$x + 0$: A F; at $x + 1$: 2 F

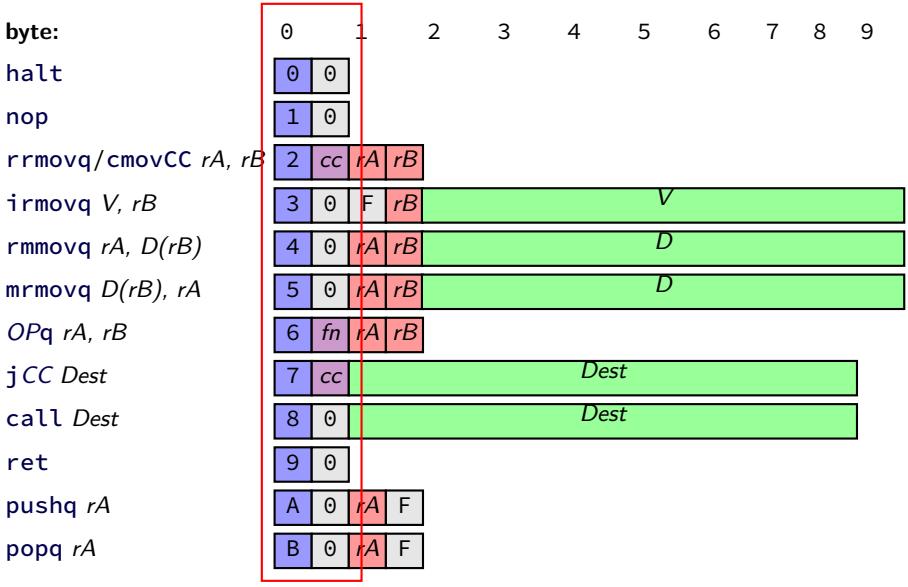
as a little-endian 2-byte number in typical English order:



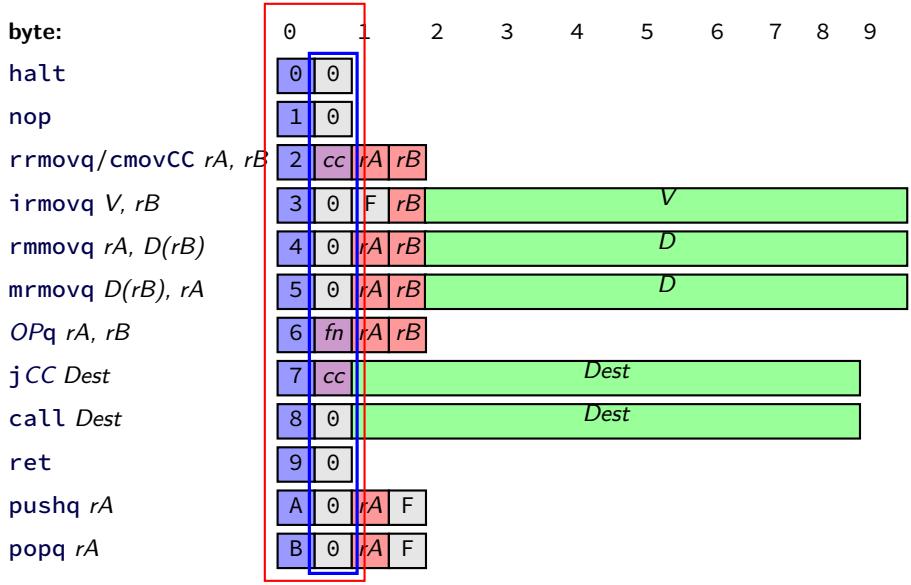
# Y86 encoding table



# Y86 encoding table

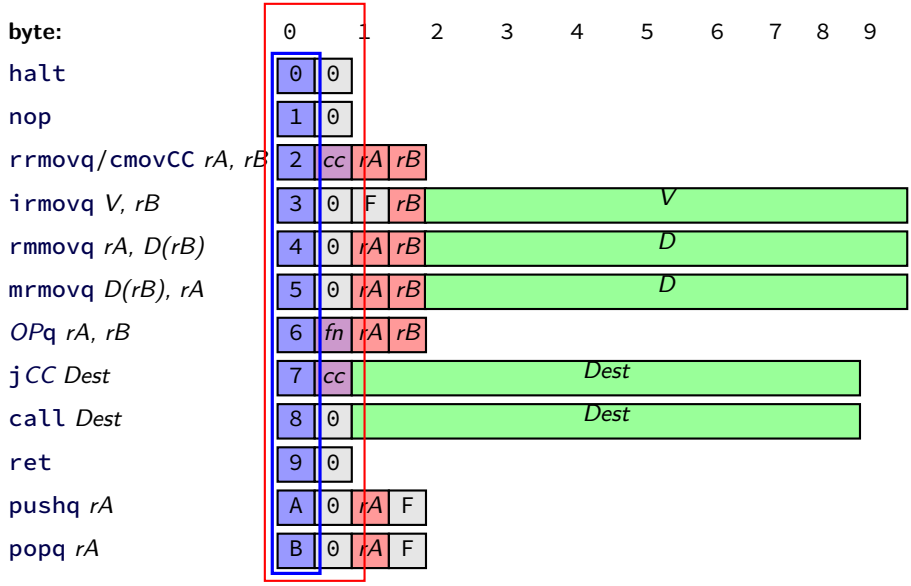


# Y86 encoding table



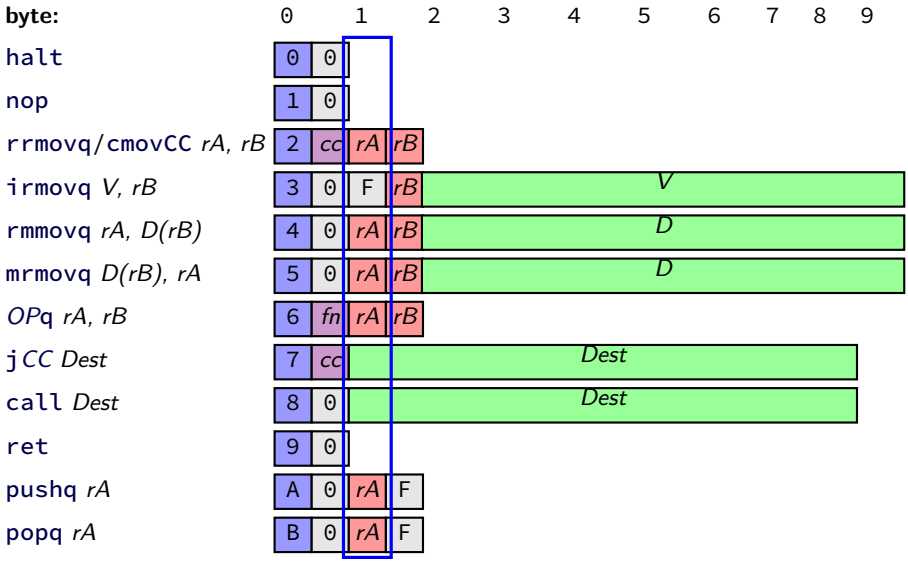

least sig. 4 bits of byte 0: bits 0–4

# Y86 encoding table



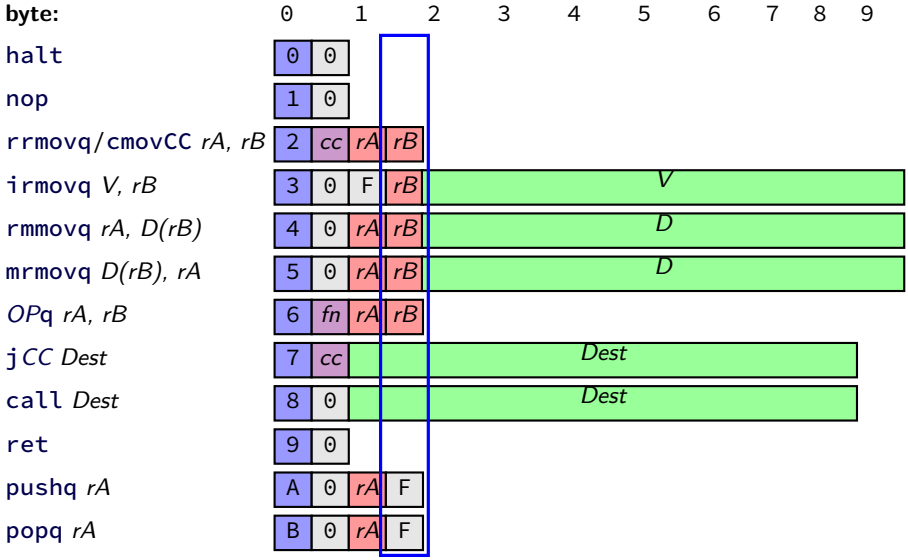

most sig. 4 bits of byte 0: bits 4–8

# 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

# 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 = ...
```

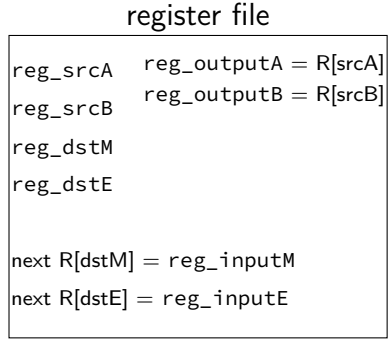
# example using register file: add CPU

```
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 = ...
```

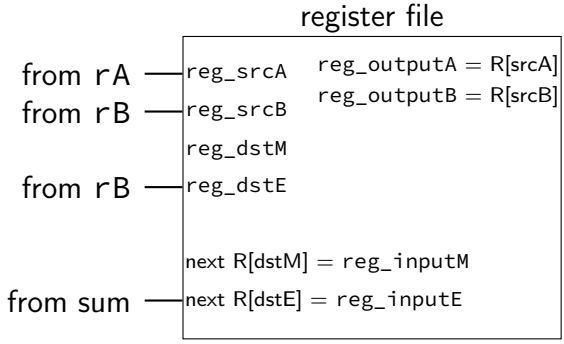
# example using register file: add CPU

```
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 = ...
```

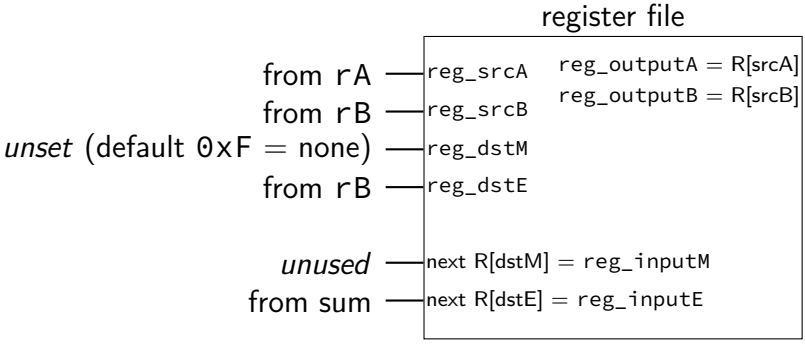
# register file picture



# register file picture



# register file picture



# register file picture