SEQ part 2

last time

registers

- output a stored value
- changed the stored value on rising edge of clock signal
- common clock signal to whole circuit/processor

MUXes

- gate-like circuit for making decision
- extremely abstract HCLRS representation

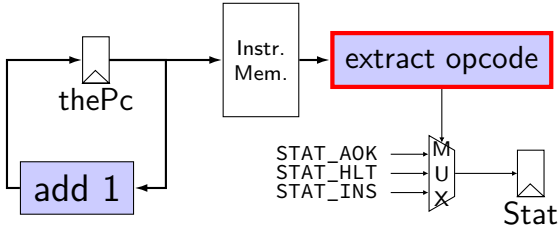
instruction memory

- input address: get value at address after delay
- read 10 bytes — size of largest instruction

nop/halt CPU

- MUX based on opcode to decide whether to continue or not
- extract bits 4-8 (most significant bits of 1st byte) of i10bytes signal

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

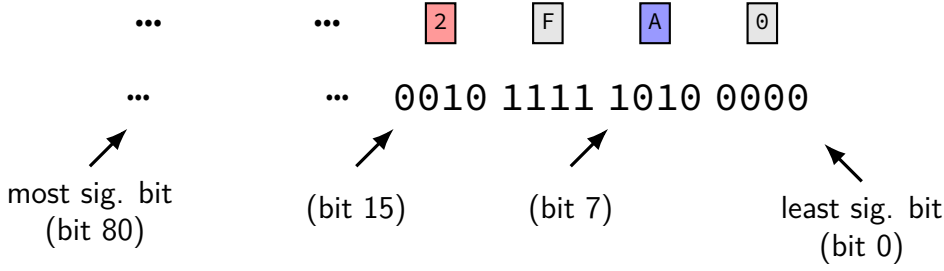
i10bytes example

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

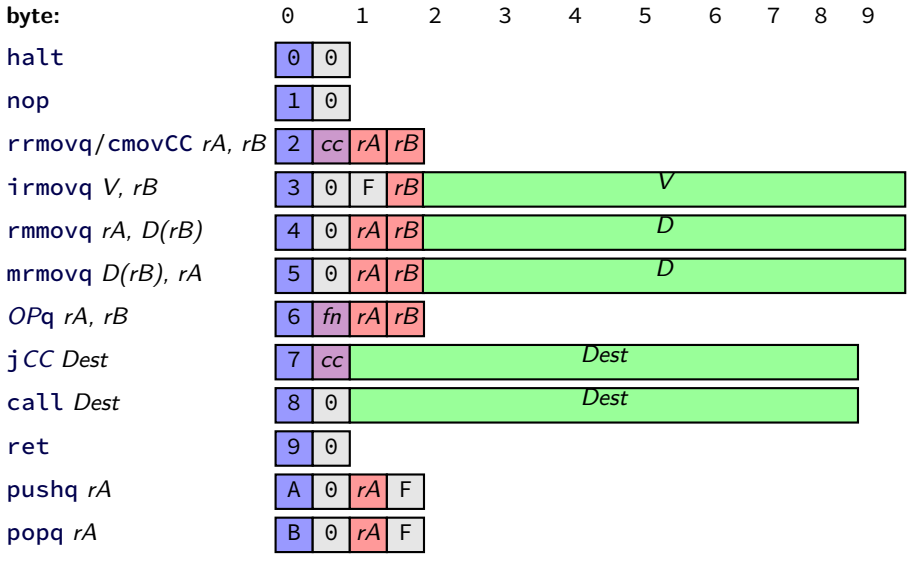
memory at $x + 0$: pushq F; at $x + 1$: rbx F

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

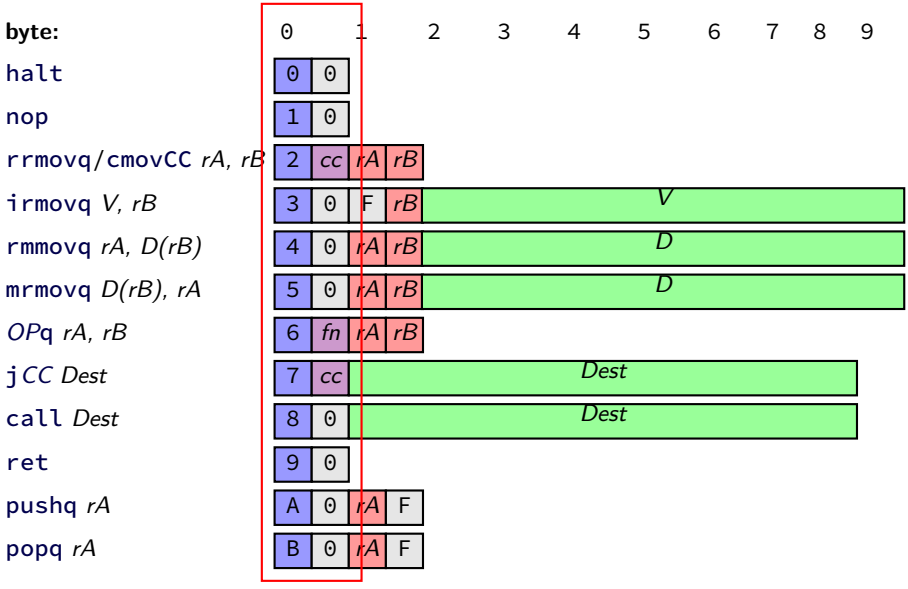
10-byte instruction memory output:



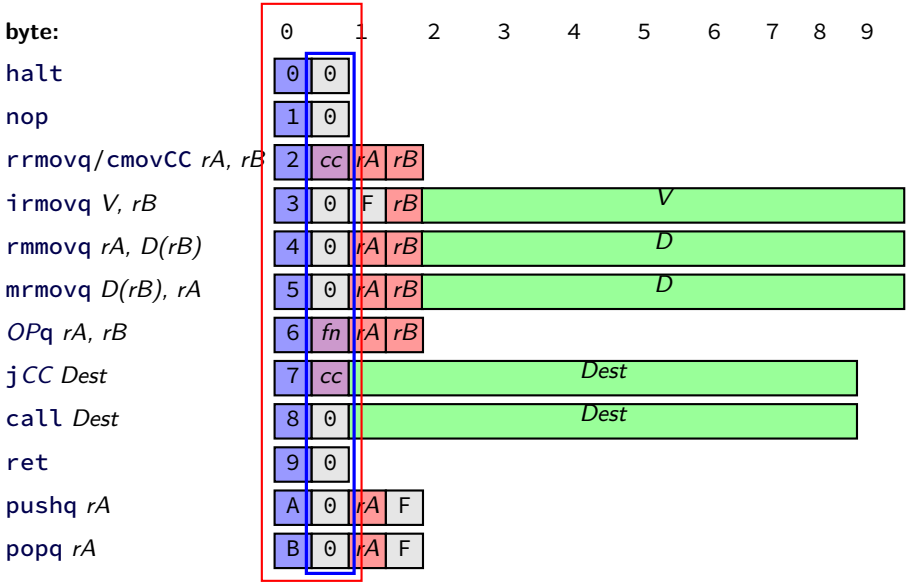
Y86 encoding table



Y86 encoding table

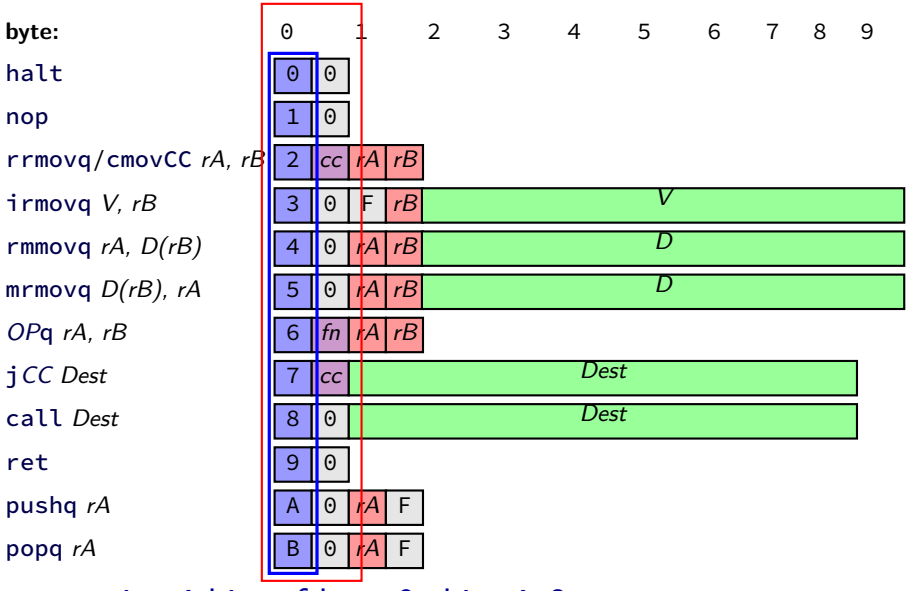


Y86 encoding table



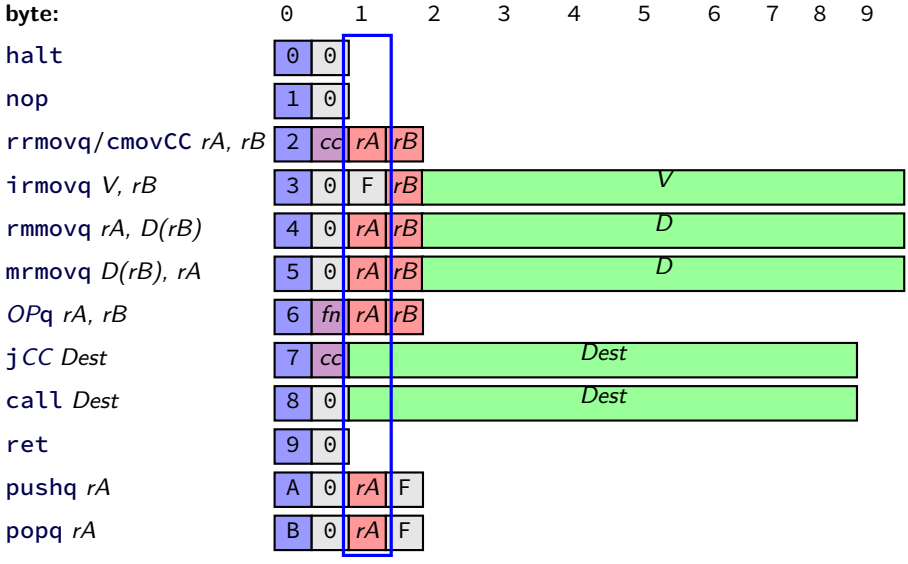
least sig. 4 bits of byte 0: bits 0–4

Y86 encoding table



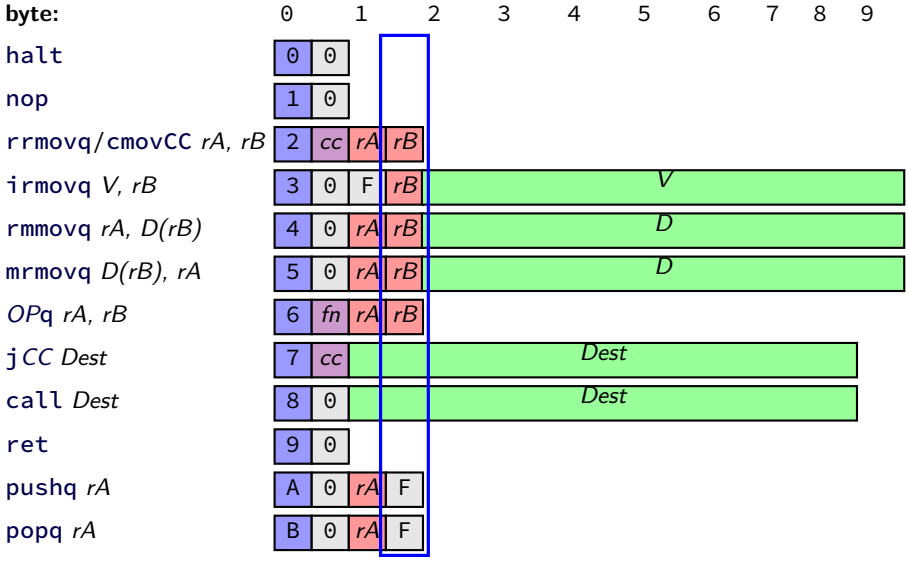
most sig. 4 bits of byte 0: bits 4–8

Y86 encoding table



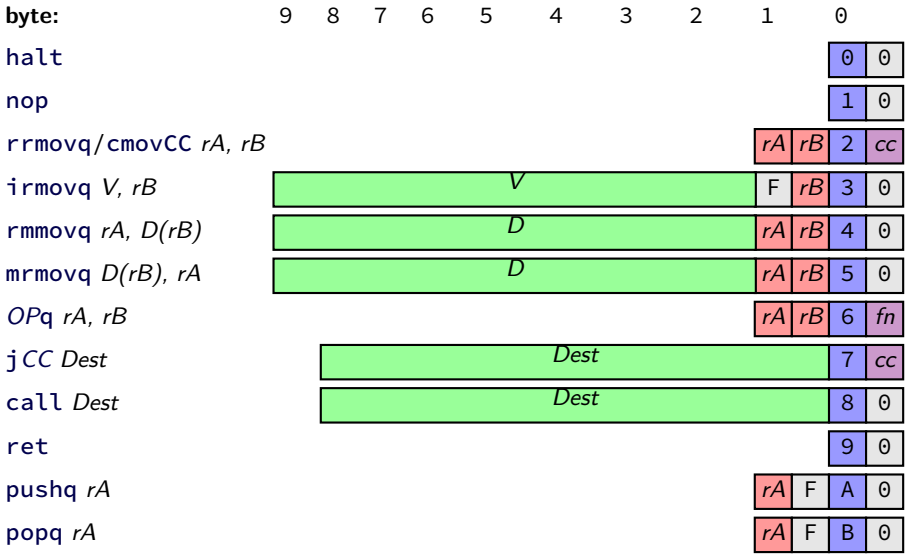
most sig. 4 bits of byte 1: bits 12–16

Y86 encoding table

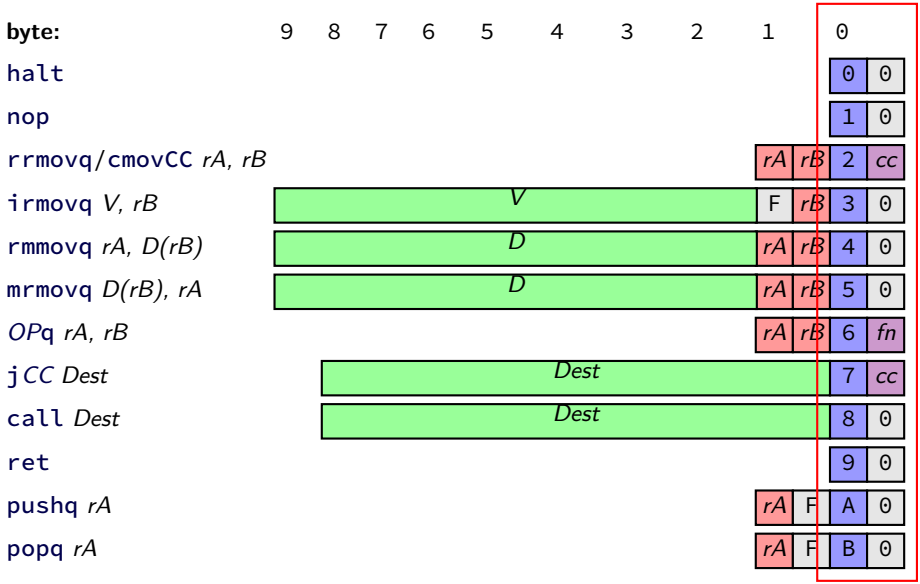


least sig. 4 bits of byte 1: bits 8–12

Y86 encoding table (written differently)

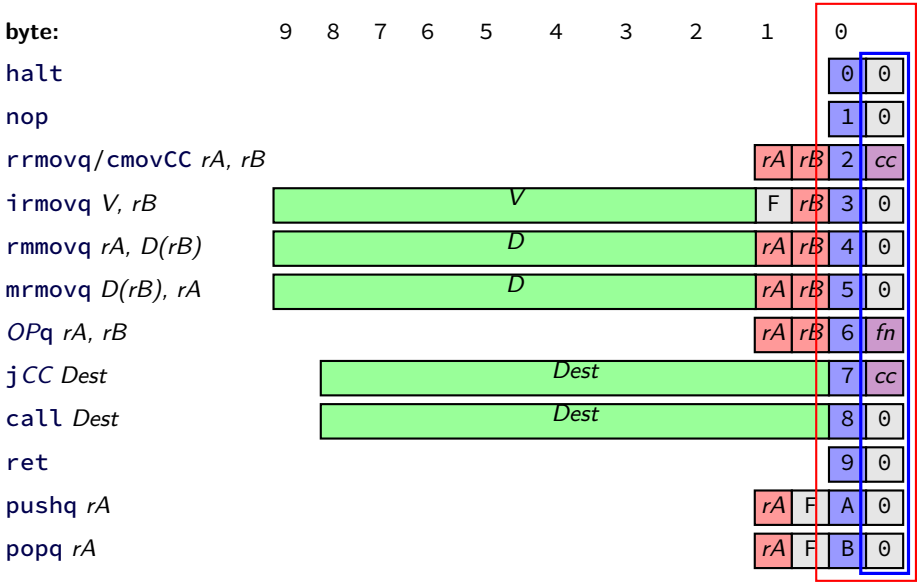


Y86 encoding table (written differently)



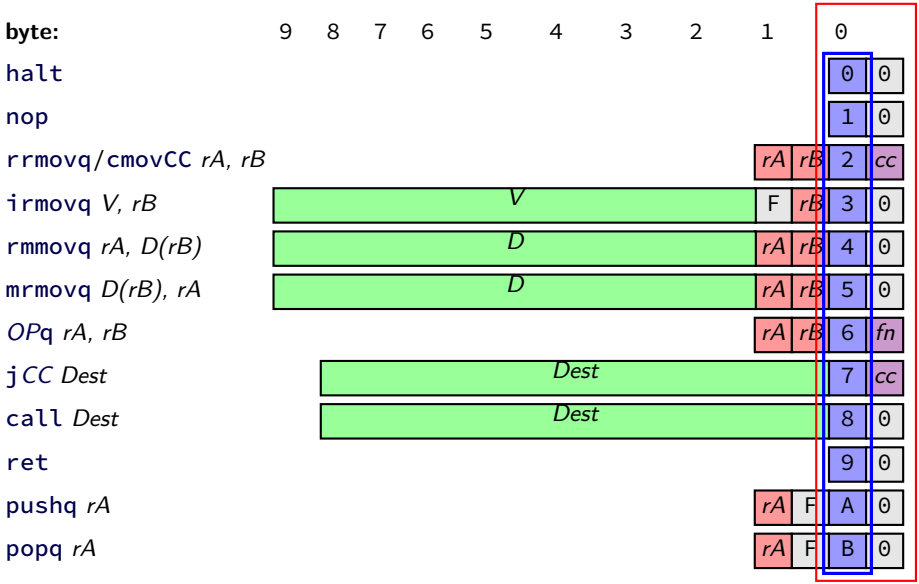
byte 0: bits 0–7

Y86 encoding table (written differently)



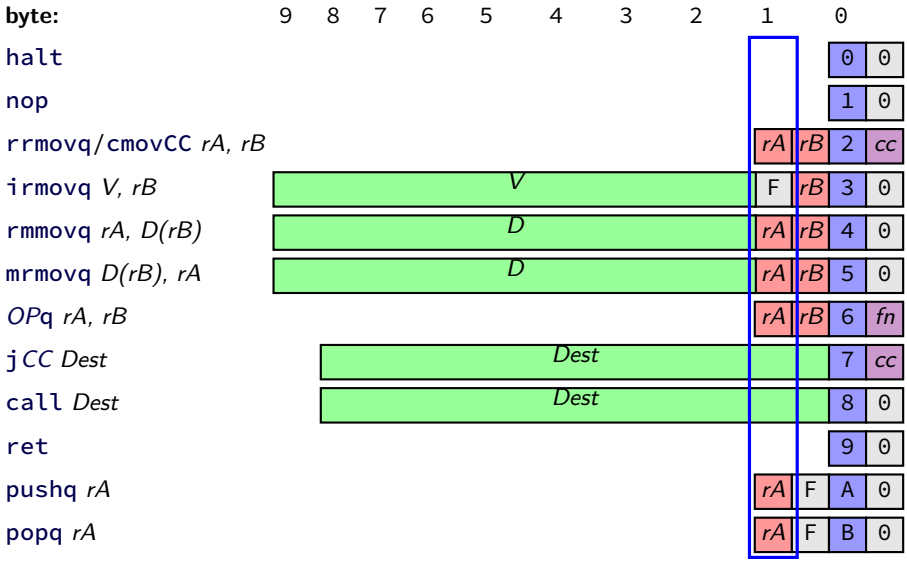
least sig. 4 bits of byte 0: bits 0–4

Y86 encoding table (written differently)



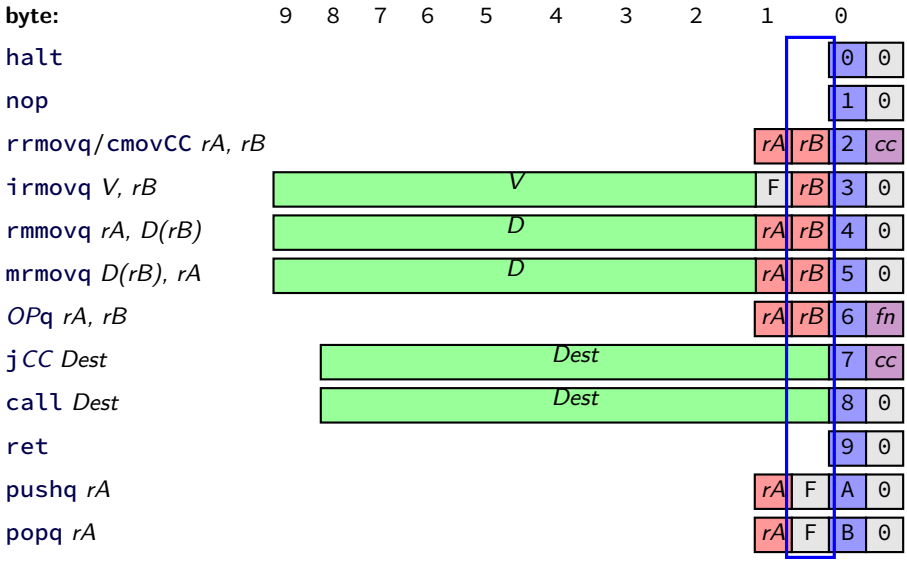
most sig. 4 bits of byte 0: bits 4–8

Y86 encoding table (written differently)



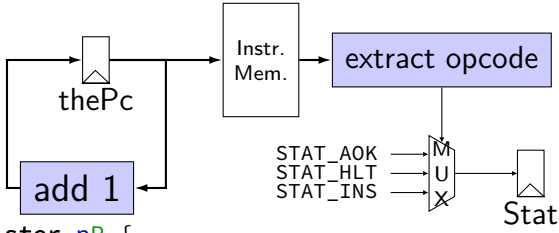
most sig. 4 bits of byte 1: bits 12–16

Y86 encoding table (written differently)



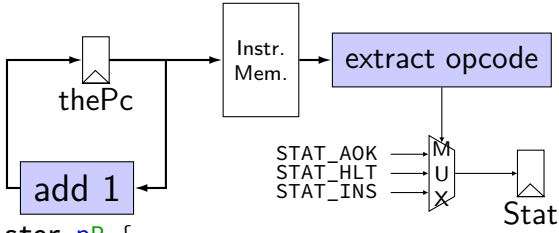
least sig. 4 bits of byte 1: bits 8–12

nop/halt CPU



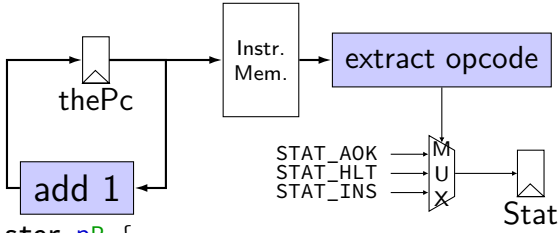

```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; // (default case)
];
```

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)
];
```

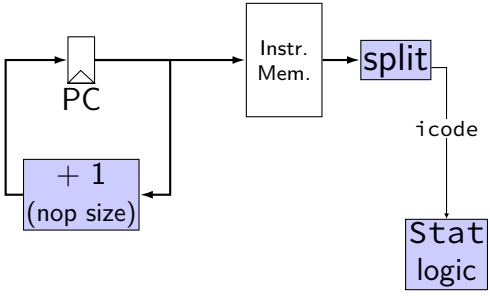
nop/halt CPU



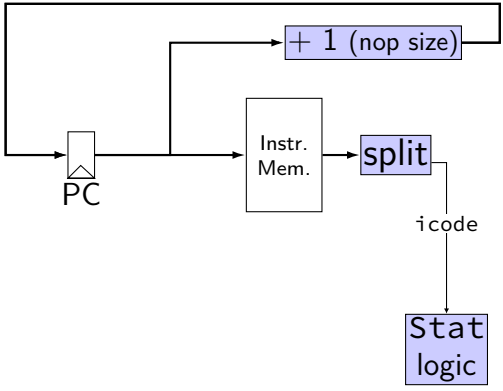

```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; // (default case)
];
```

demo

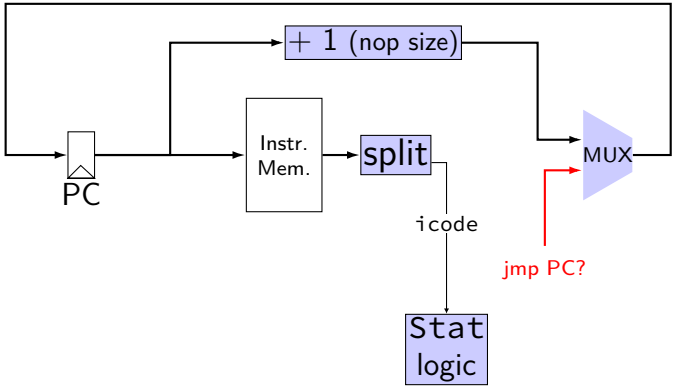
nop/halt → nop/jmp CPU



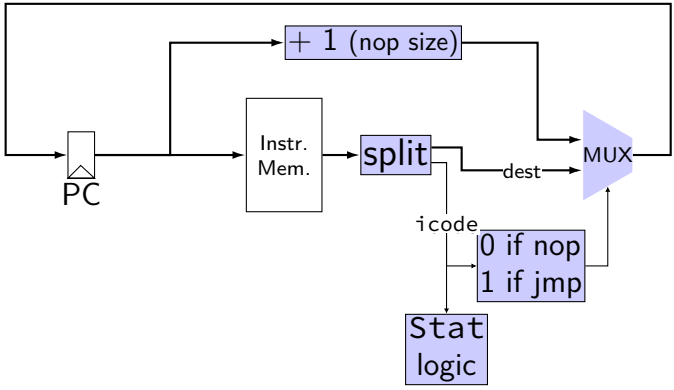
nop/halt → nop/jmp CPU



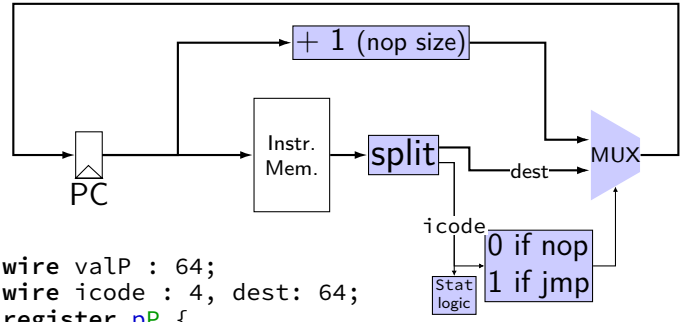
nop/halt → nop/jmp CPU



nop/halt → nop/jmp CPU



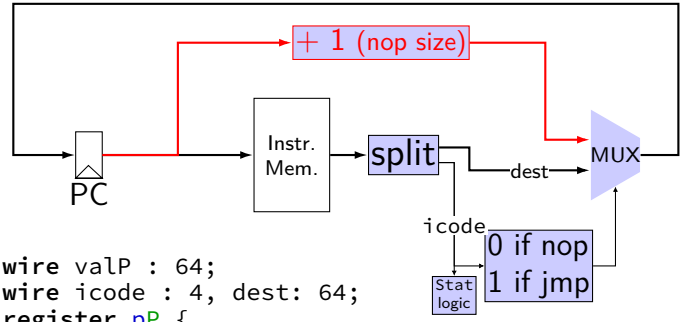
nop/jmp CPU




```
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;
];
```

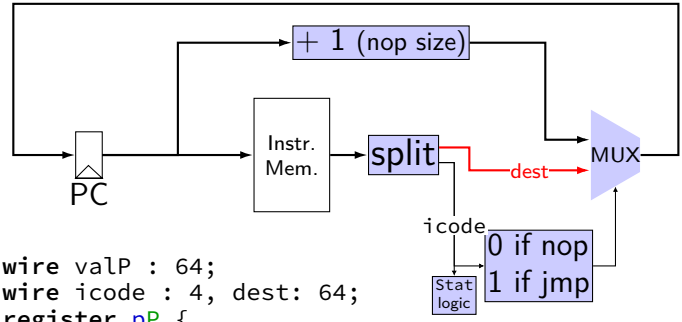
nop/jmp CPU




```
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;
];
```

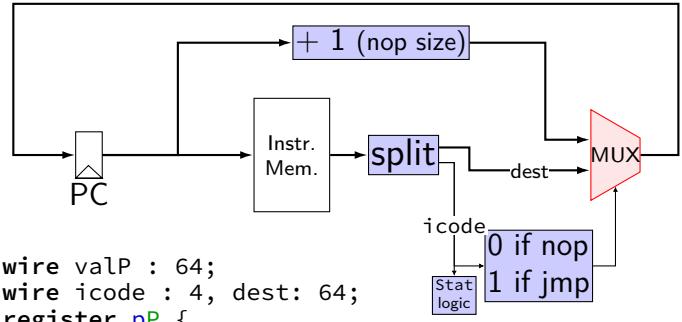
nop/jmp CPU




```
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;
];
```

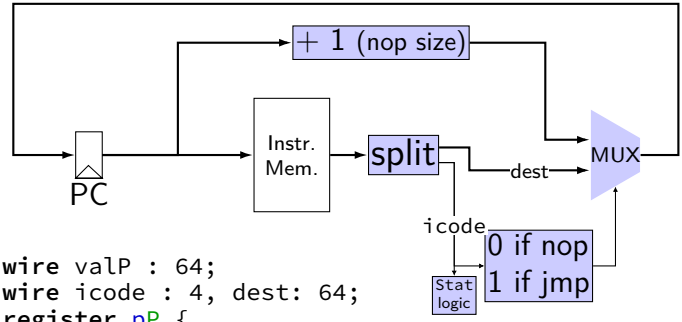
nop/jmp CPU




```
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: debug and interactive mode

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: 701c000000000000000 | B:   jmp D
0x013: 700a000000000000000 | 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
```

simple ISA: addq

`addq %rXX, %rYY`

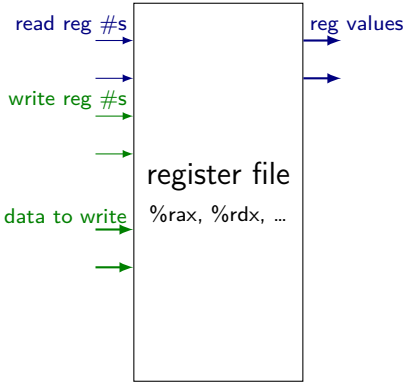
encoding: `6` `0` `%rXX` `%rYY` (two 4-bit register #s)

- 2 byte instructions, no opcode

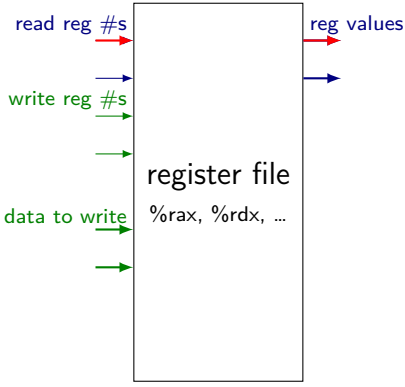
for now: no other instructions

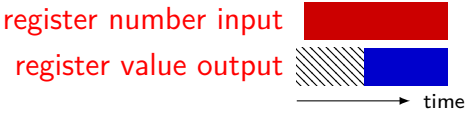
- later: adding support for nop+halt

register file

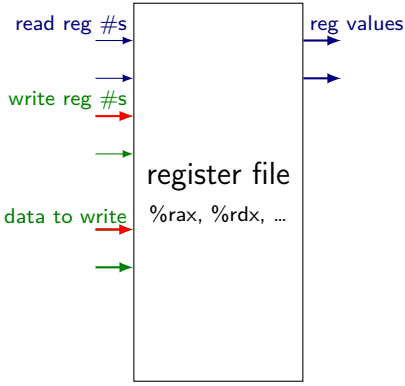


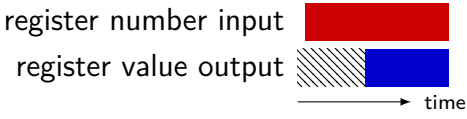
register file

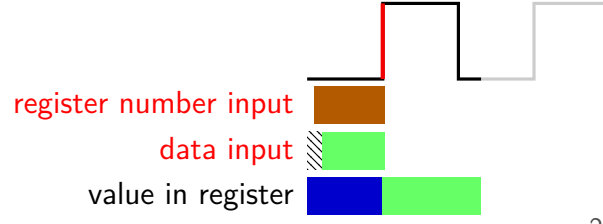




register file



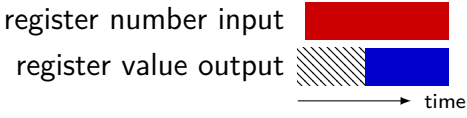


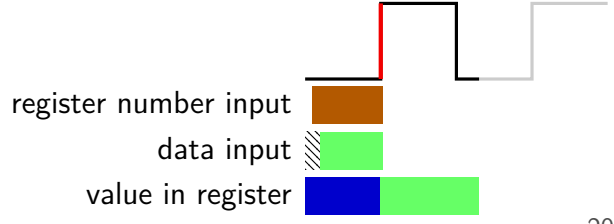


register file

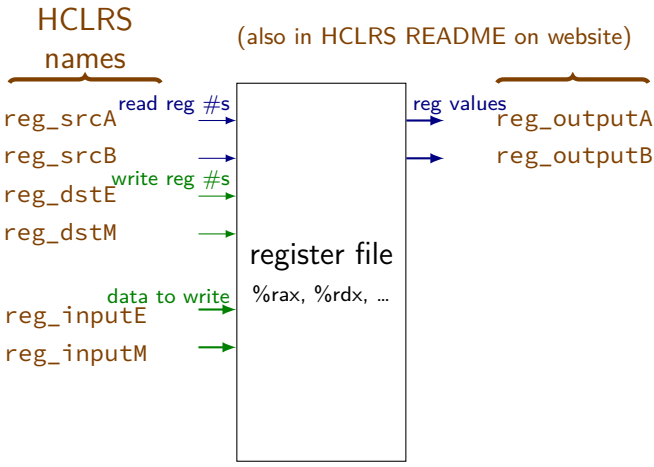
write register #15 (REG_NONE):
write is ignored

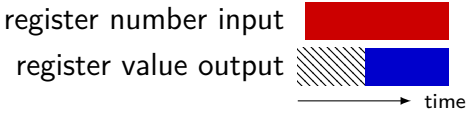
read register #15 (REG_NONE):
value is always 0

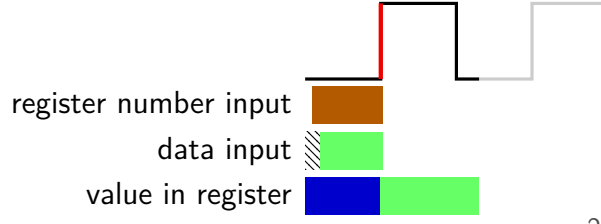




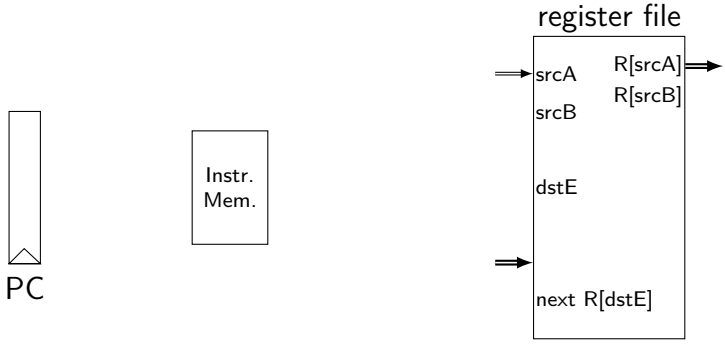
register file



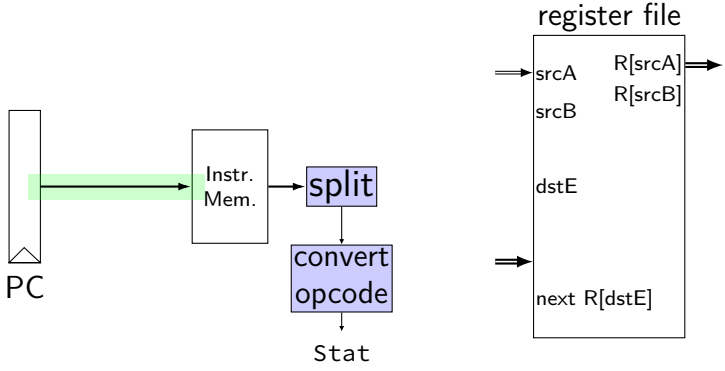




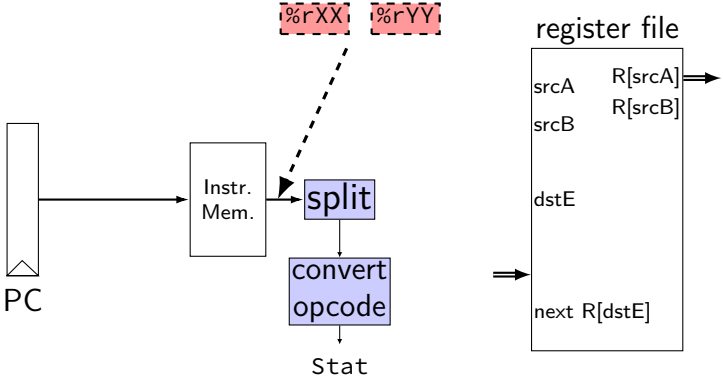
addq CPU


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

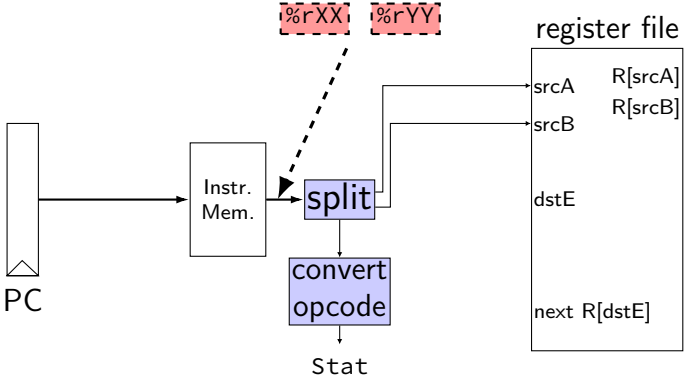
addq CPU


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

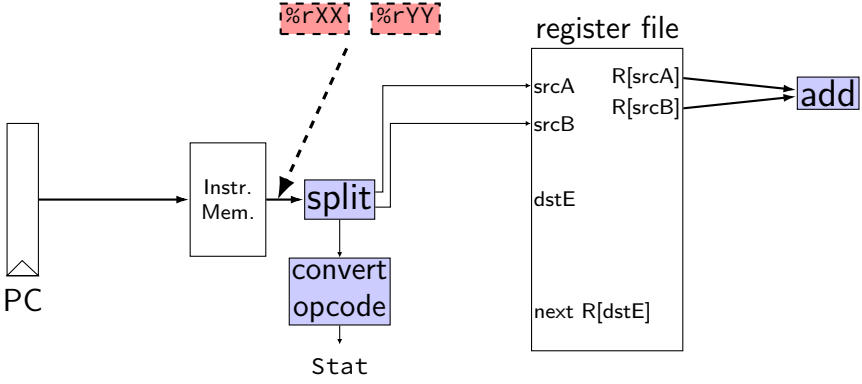
addq CPU


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

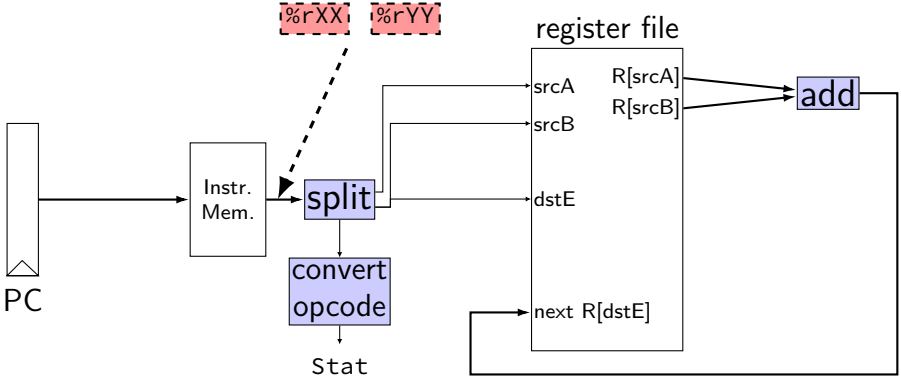
addq CPU


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

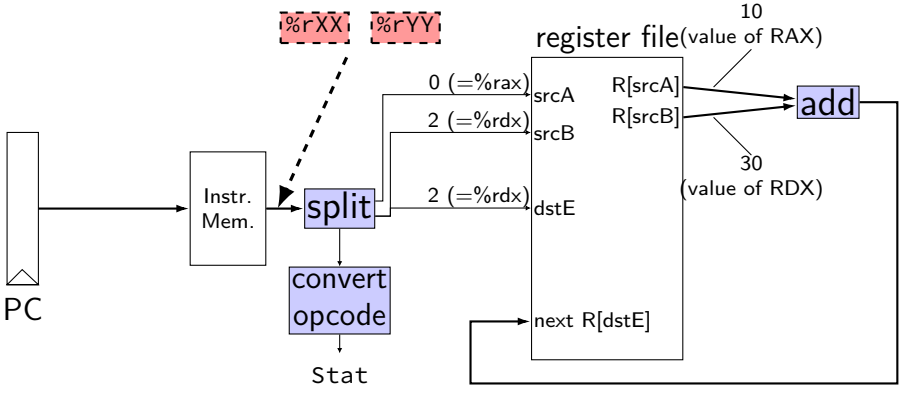
addq CPU


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

addq CPU


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

addq CPU

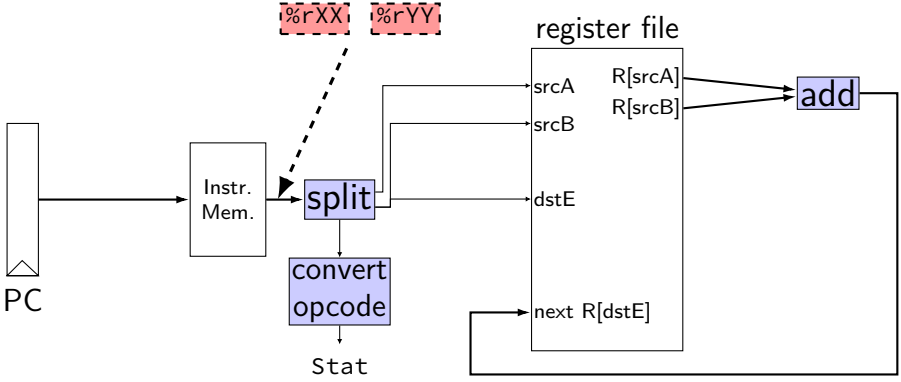



```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

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

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

addq CPU



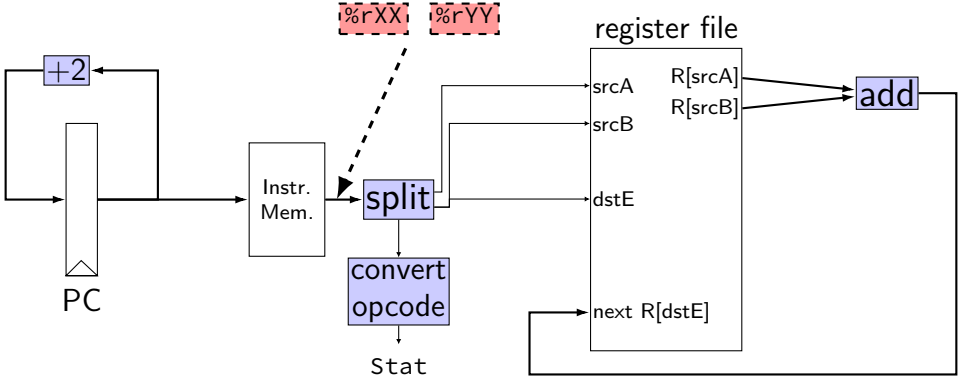

```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

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

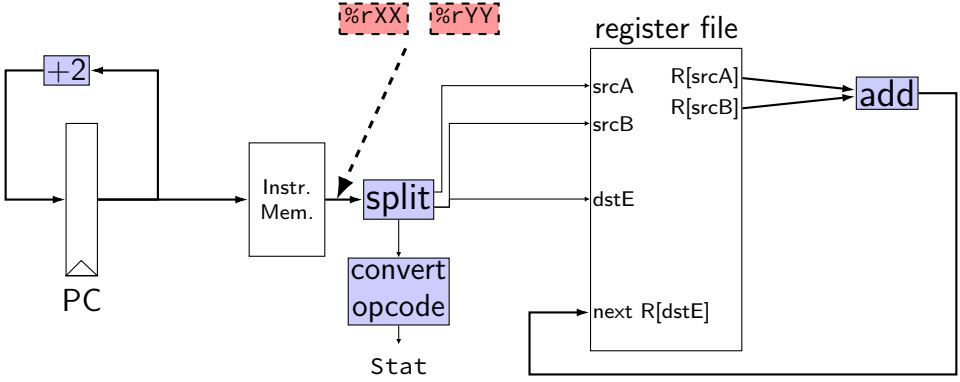
after cycle 1: PC = ????, rax = 10, rbx = 20, rdx = 40

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

addq CPU


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

addq CPU



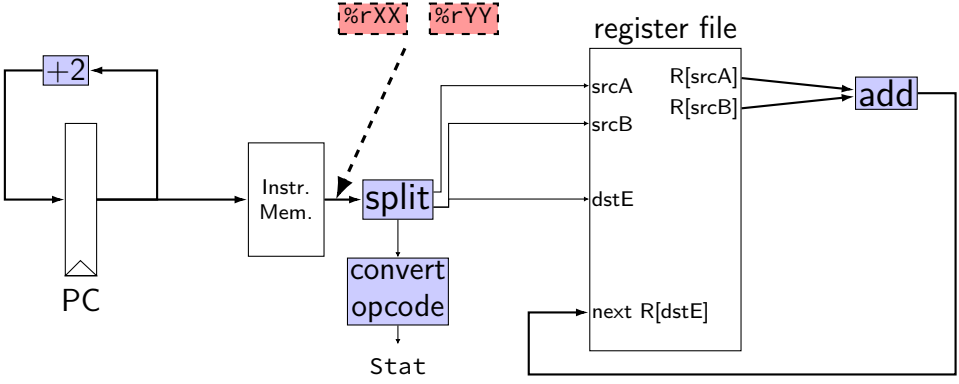

```
/* 0x00: */ addq %rax, %rdx
/* 0x02: */ addq %rbx, %rdx
```

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

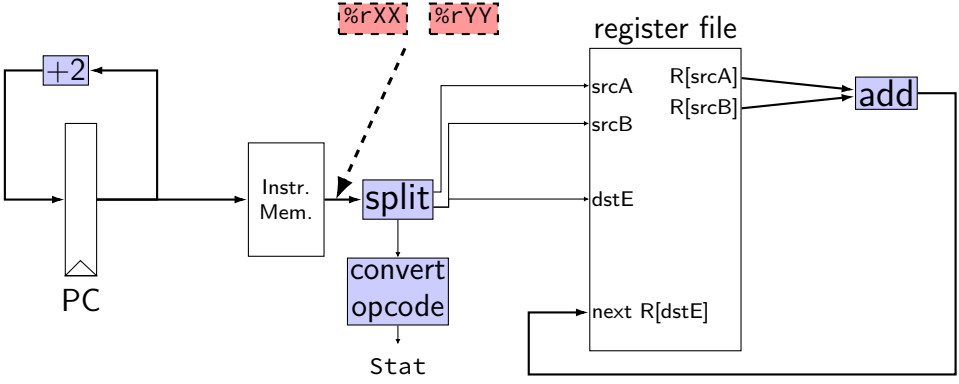
after cycle 1: PC = 0x02, rax = 10, rbx = 20, rdx = 40

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

addq CPU: HCL


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

addq CPU: HCL

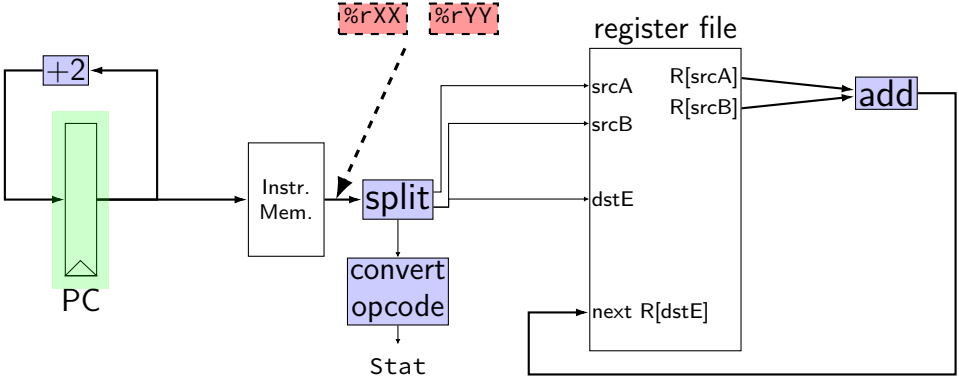



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

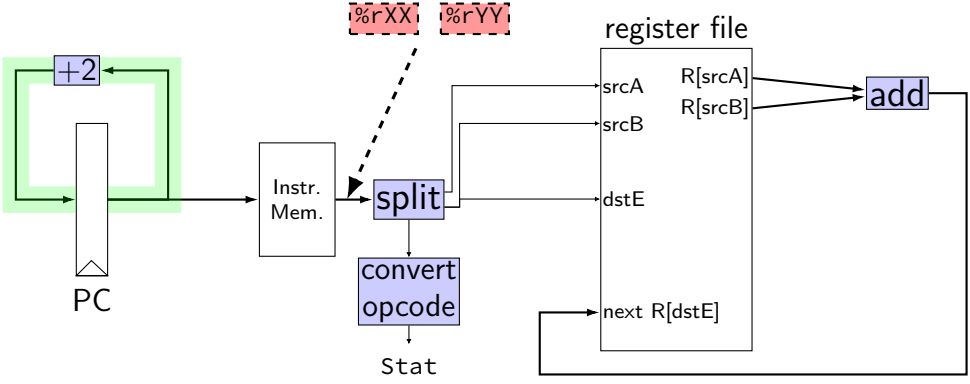



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

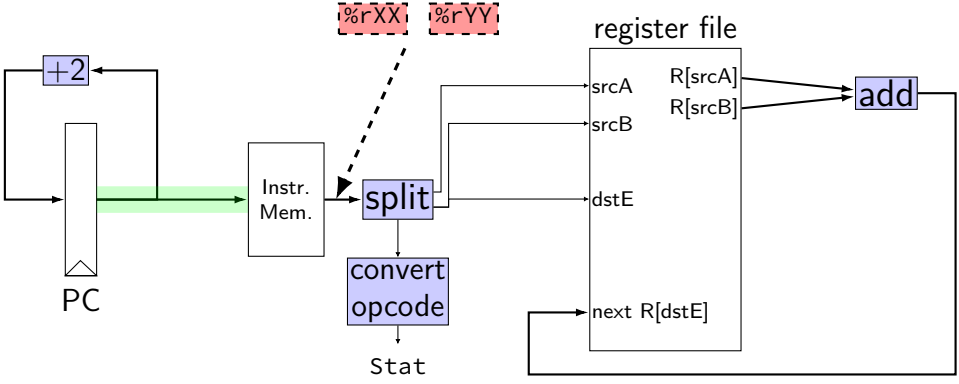



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

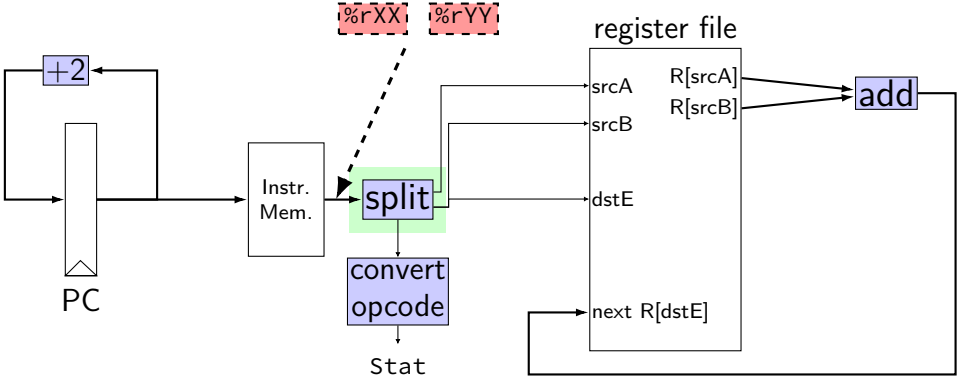



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

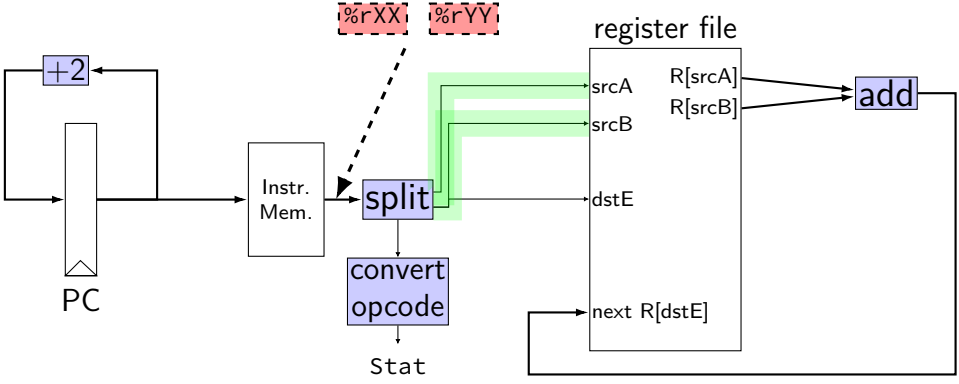



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

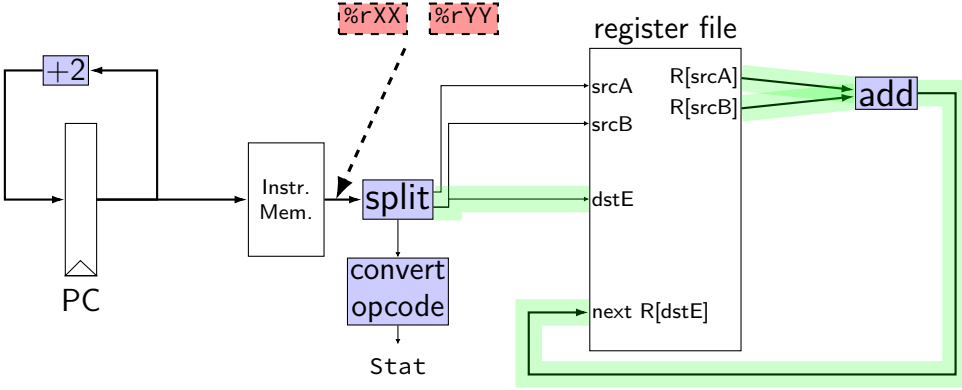



```
register pP {
     pc : 64 = 0;
}
p_pc = P_pc + 2;
pc = P_pc;
```

```
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

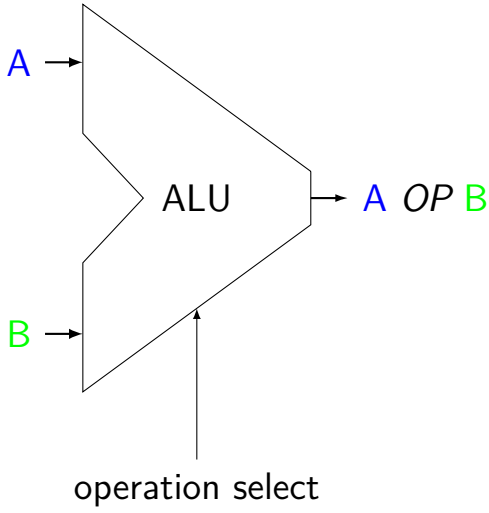
book uses names like `valC` — not required!

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

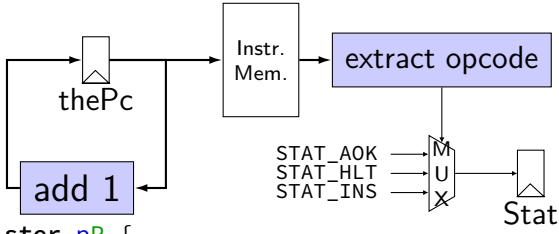
HCLRS doesn't supply an ALU

- 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...)

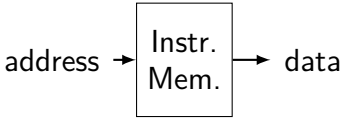
D. one of the register file 'register value to write' inputs (reg_input...)

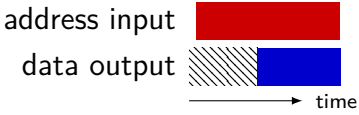
E. the instruction memory's address input (pc)

simple ISA: mov-to-register

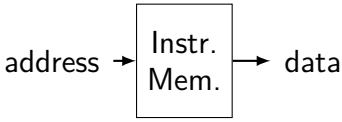
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

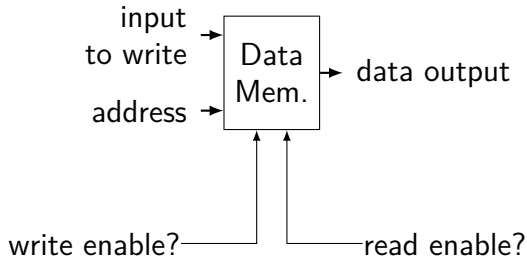
two memories

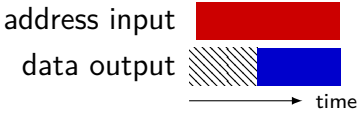




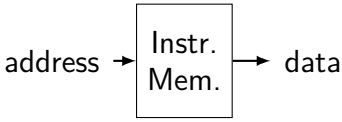
two memories

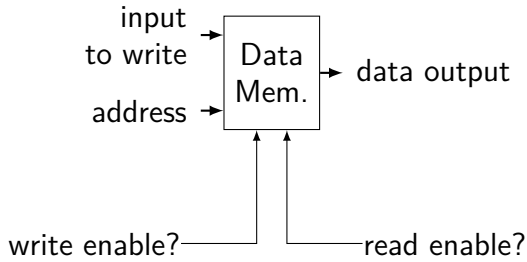


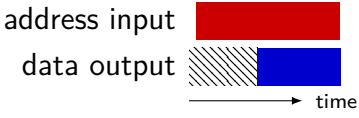


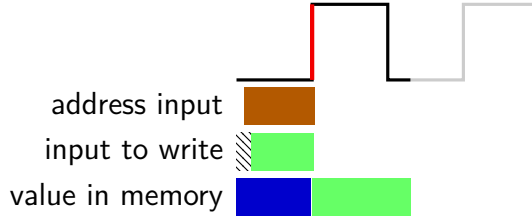


two memories

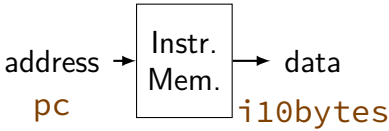


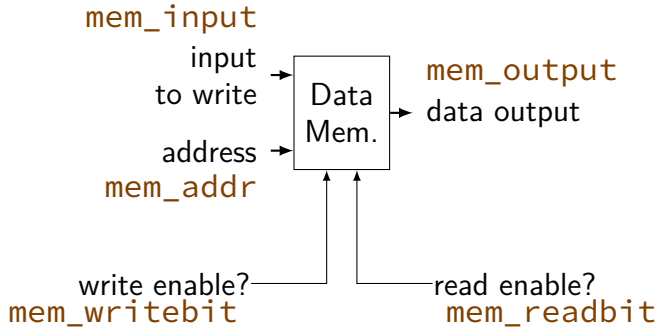


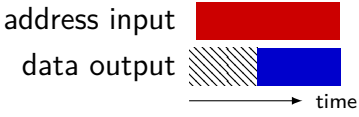


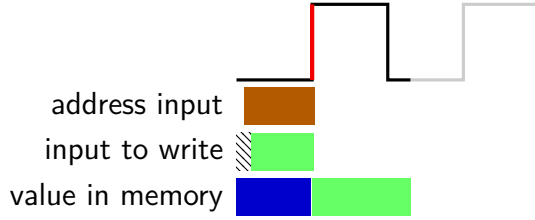


two memories









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*

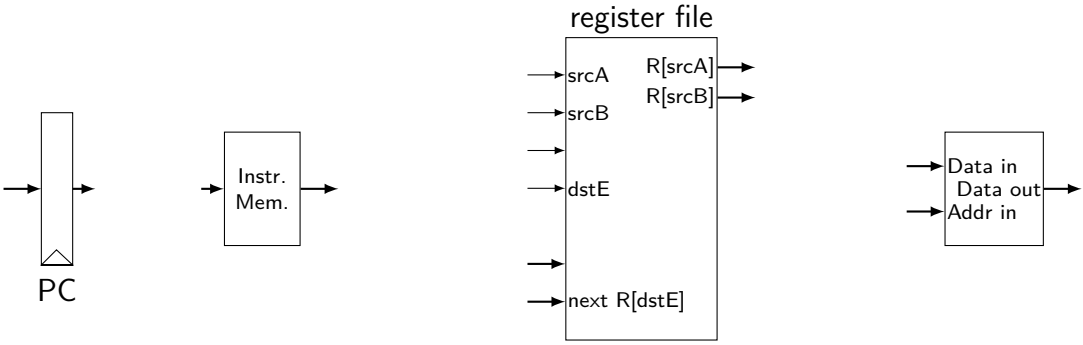
exercise: mov-to-register

```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

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

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

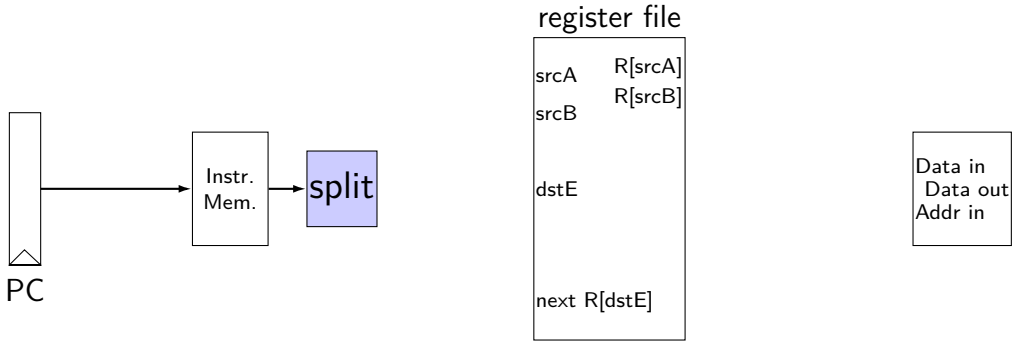
A. register file's register number (index) inputs (`reg_srcA`, `reg_srcB`, `reg_dstE`, ...)
B. register file's value inputs (`reg_inputE/M`)
C. PC register's input
D. instruction memory's address input (`pc`)

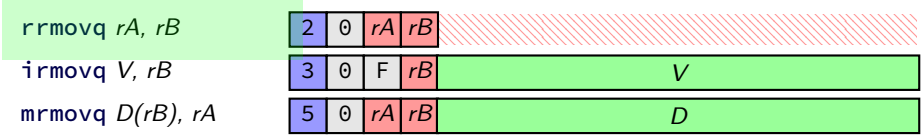
mov-to-register CPU



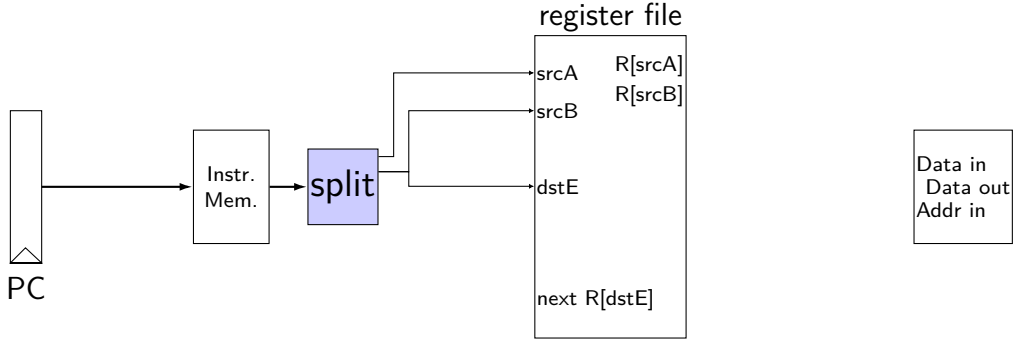
`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*

mov-to-register CPU



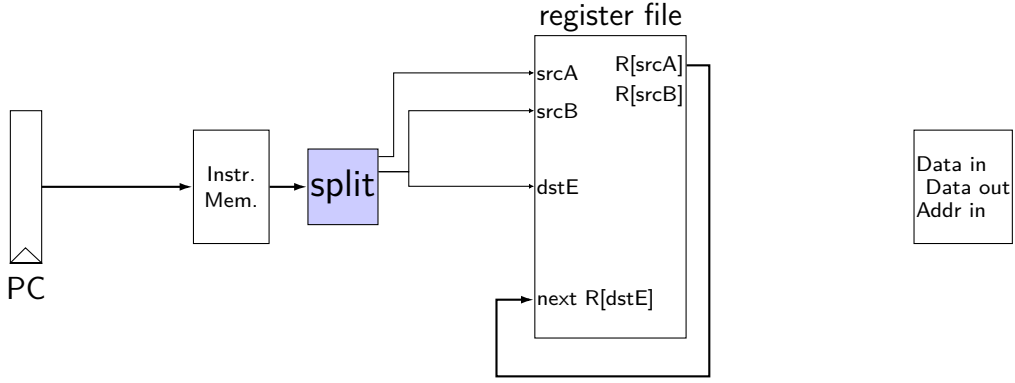


mov-to-register CPU



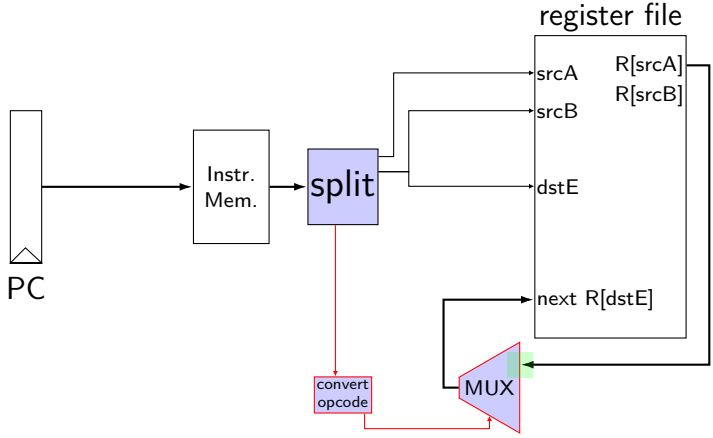
`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*

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

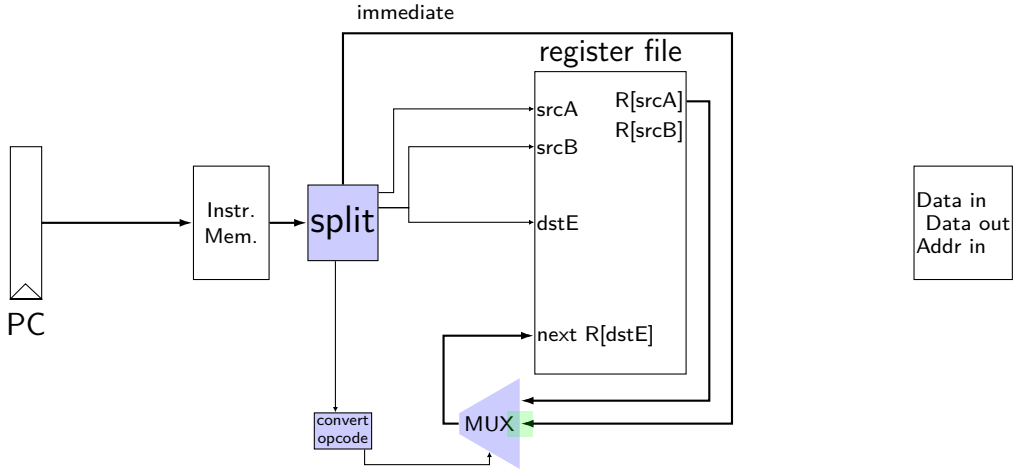


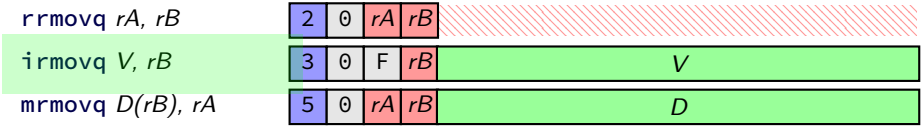
Data in
Data out
Addr in

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

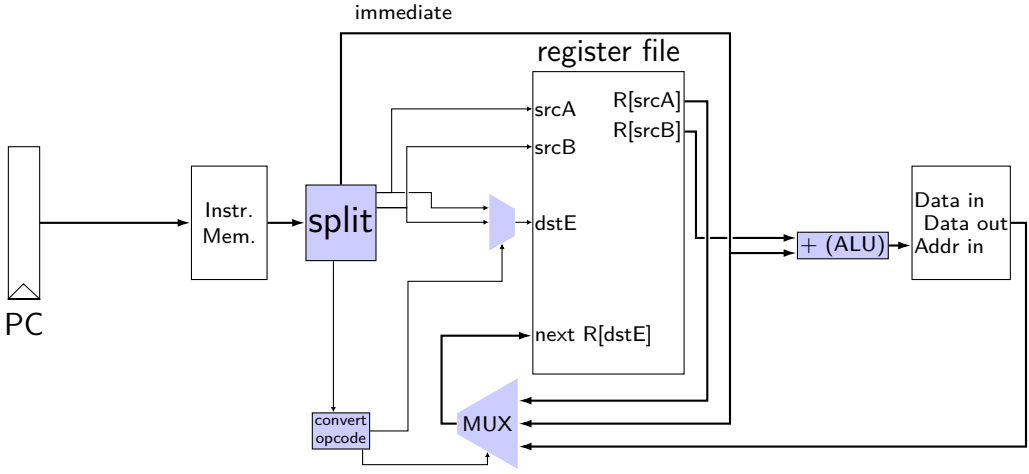
`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*

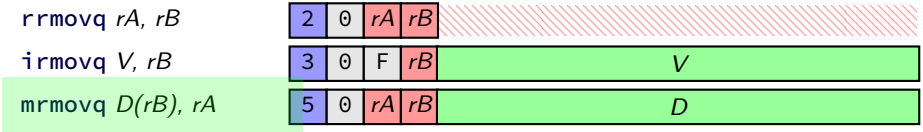
mov-to-register CPU



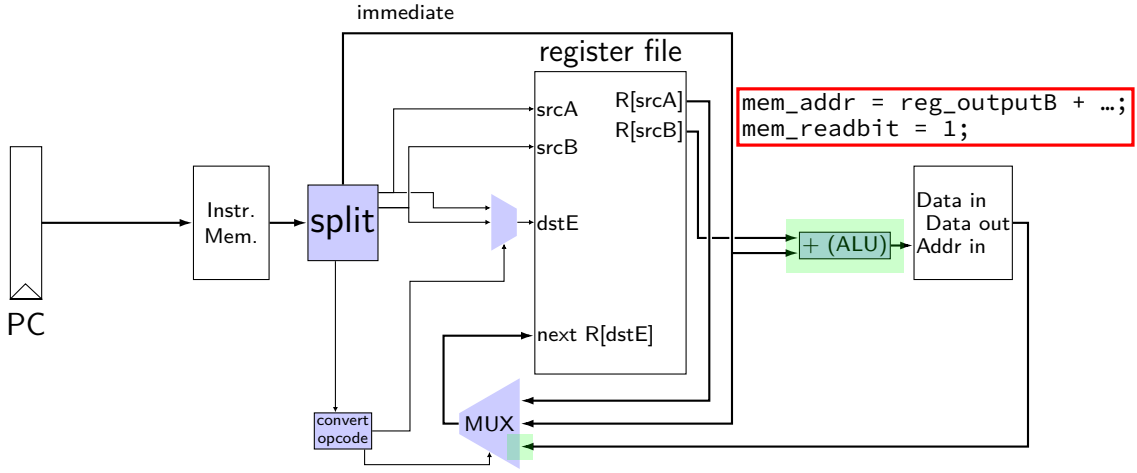


mov-to-register CPU



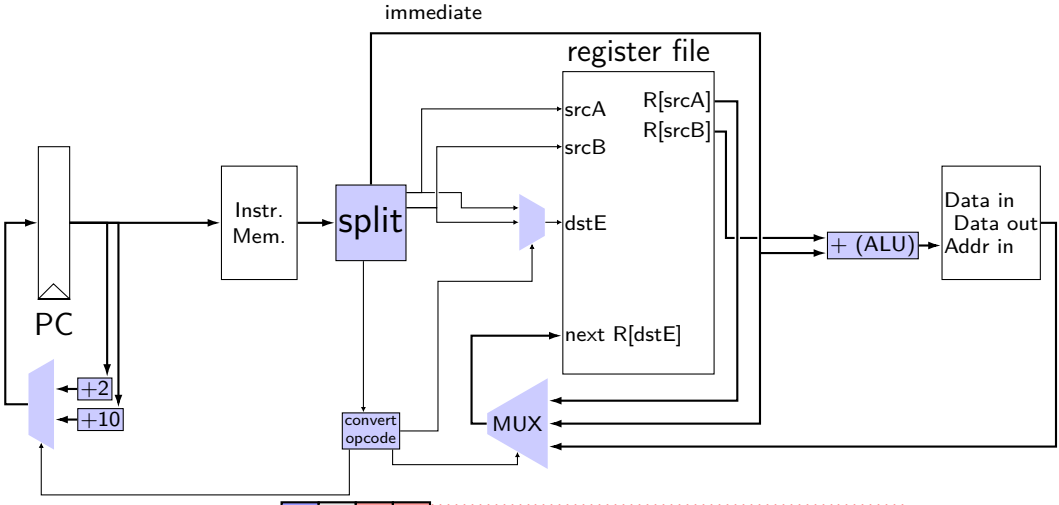


mov-to-register CPU



`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

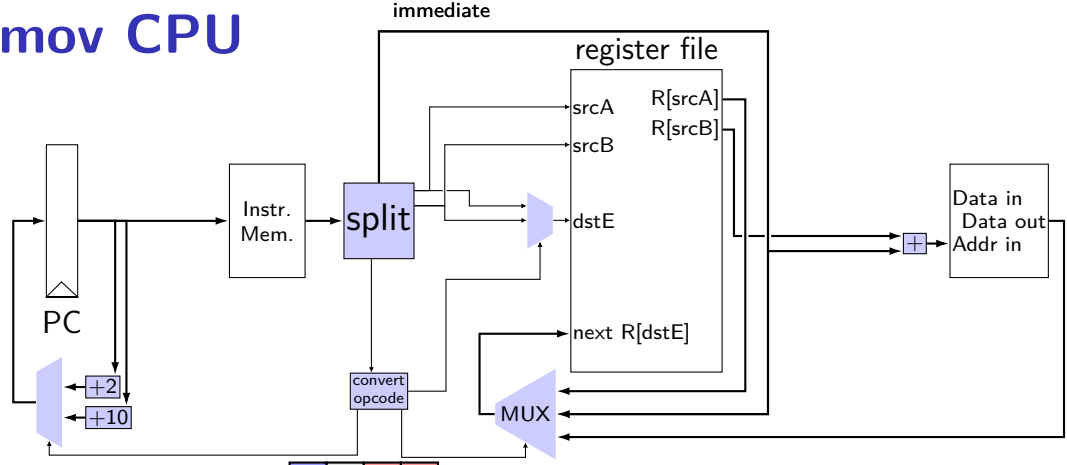


`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*

simple ISA: mov (all cases)

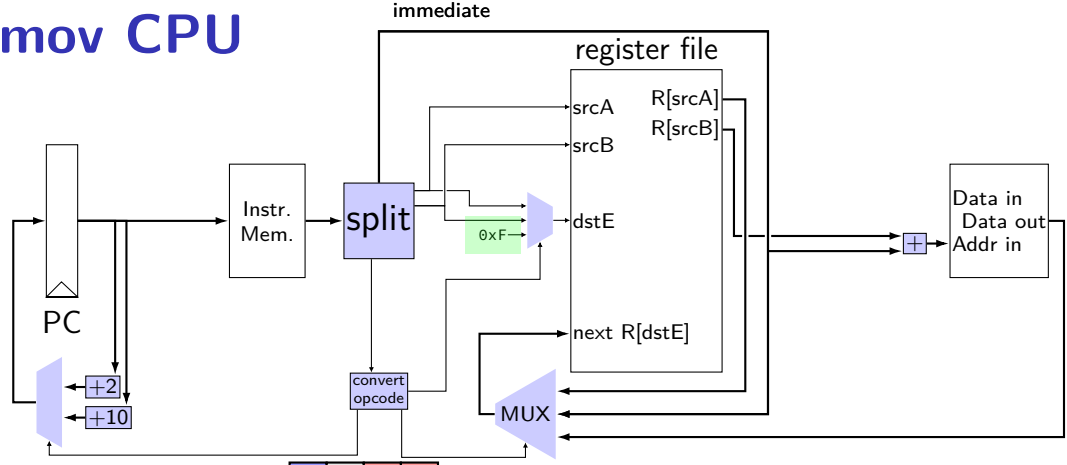
```
irmovq $constant, %rYY

rrmovq %rXX, %rYY

mrmovq 10(%rXX), %rYY

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

mov CPU

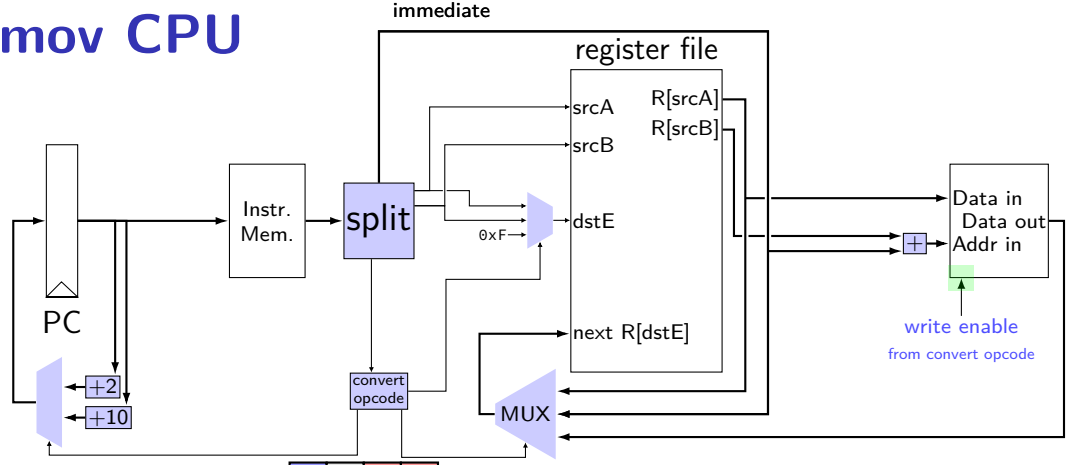


`rrmovq` *rA, rB*	2	0	*rA*	*rB*	
`irmovq` *V, rB*	3	0	F	*rB*	*V*
`mrmovq` *D(rB), rA*	5	0	*rA*	*rB*	*D*
`rmmovq` *rA, D(rB)*	4	0	*rA*	*rB*	*D*

mov CPU


immediate
register file
srcA
srcB
dstE
R[srcA]
R[srcB]
next R[dstE]
Instr.
Mem.
split
0xF
PC
+2
+10
convert
opcode
MUX
Data in
Data out
Addr in
rrmovq rA, rB
irmovq V, rB
mrmovq D(rB), rA
rmmovq rA, D(rB)
2 0 rA rB
3 0 F rB V
5 0 rA rB D
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*

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)