

## SEQ part 2

# Changelog

Changes made in this version not seen in first lecture:

20 September: mov CPU examples: fix some errors with wrong register number and MUX signal being shown in red

# last time

## Y86-64 state components

registers: PC, SF/ZF

register file, data/instruction memory

always read within clock cycle (between rising edges)

always write at rising edge of clock

## MUXes — switch statement for circuit

select input is  $i$ : connect data input  $i$  to output

## building simple single-cycle processor

always one instruction per cycle

cycle starts: rising edge — new PC

all reads and computation happen within cycle

all writes happen at following rising edge

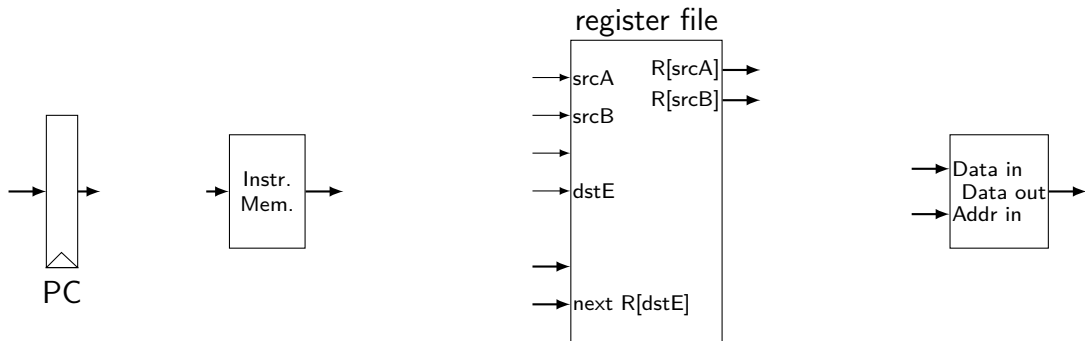
## simple ISA 4: mov-to-register

```
irmovq $constant, %rYY
```

```
rrmovq %rXX, %rYY
```

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

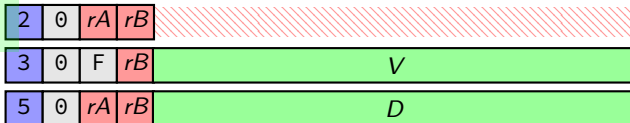
# mov-to-register CPU



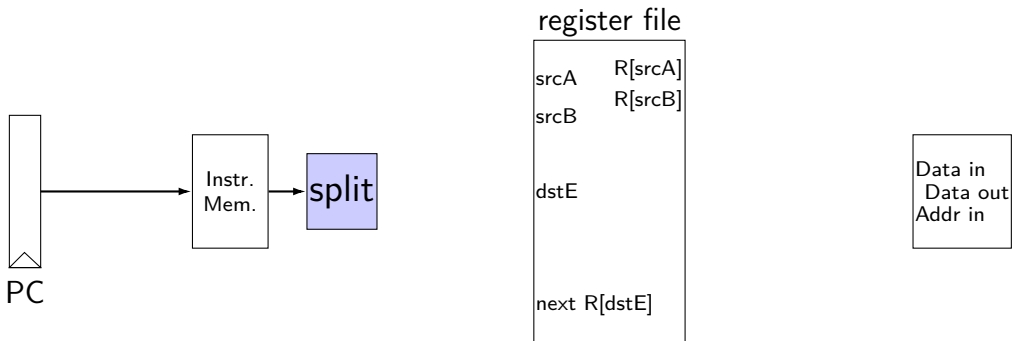
`rrmovq rA, rB`

`irmovq V, rB`

`mrmovq D(rB), rA`



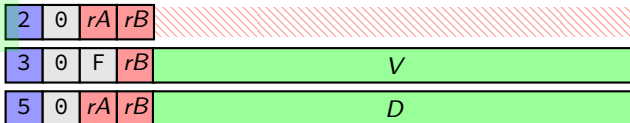
# mov-to-register CPU



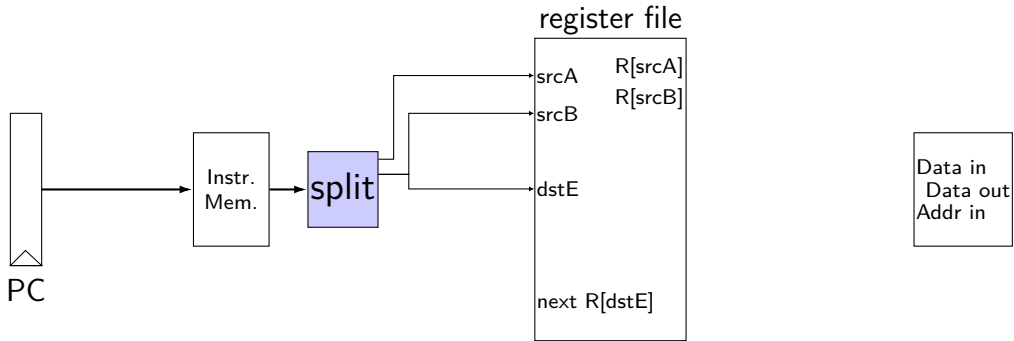
`rrmovq rA, rB`

`irmovq V, rB`

`mrmovq D(rB), rA`



# mov-to-register CPU



`rrmovq rA, rB`



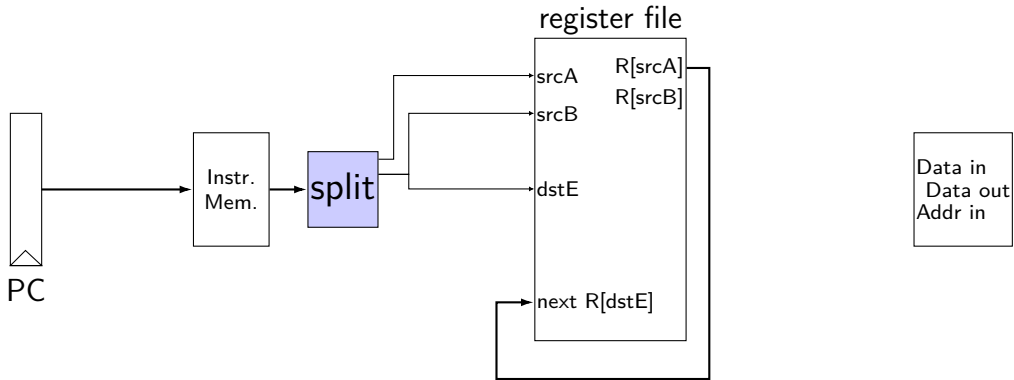
`irmovq V, rB`



`mrmovq D(rB), rA`



# mov-to-register CPU



`rrmovq rA, rB`



`irmovq V, rB`

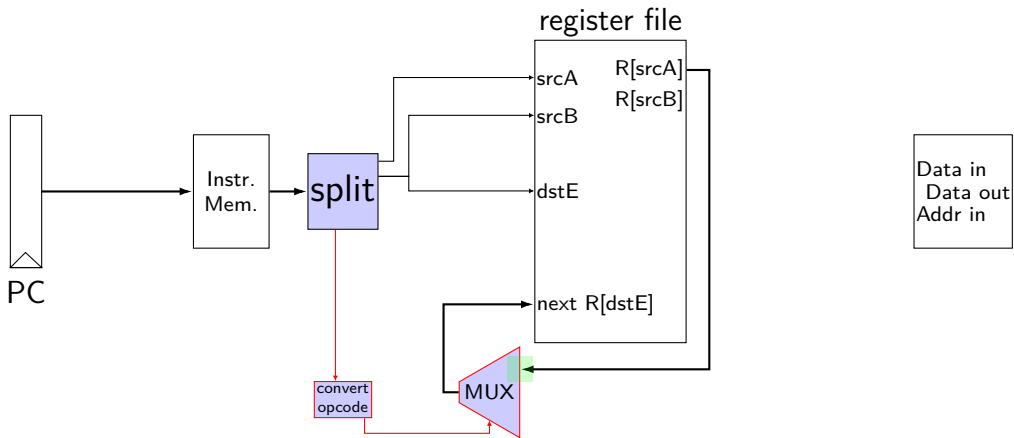


`mrmovq D(rB), rA`





# mov-to-register CPU



`rrmovq rA, rB`



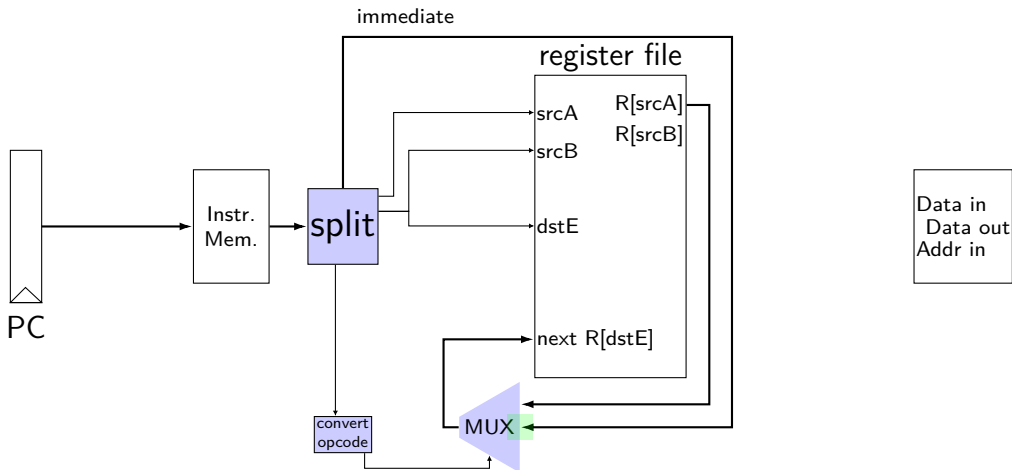
`irmovq V, rB`



`mrmovq D(rB), rA`



# mov-to-register CPU



`rrmovq rA, rB`



`irmovq V, rB`

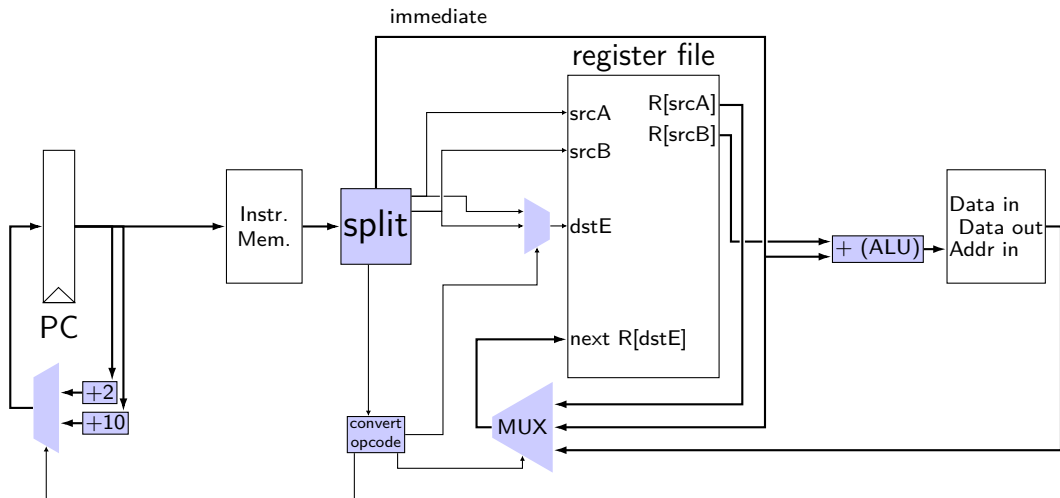


`mrmovq D(rB), rA`





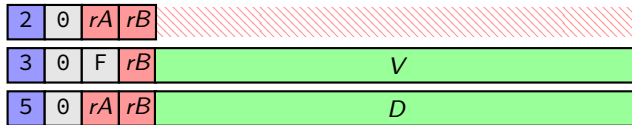
# mov-to-register CPU



`rrmovq rA, rB`

`irmovq V, rB`

`mrmovq D(rB), rA`



## simple ISA 4B: mov

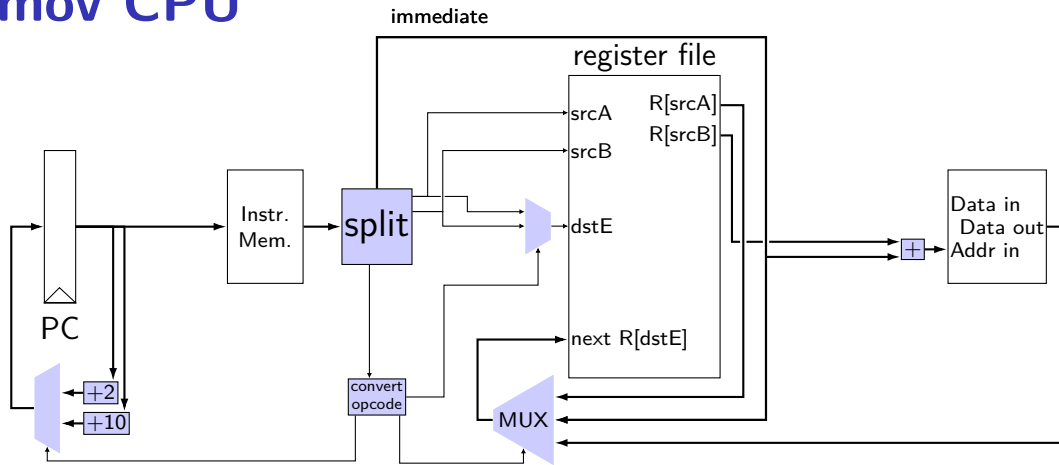
`irmovq $constant, %rYY`

`rrmovq %rXX, %rYY`

`mrmovq 10(%rXX), %rYY`

`rmmovq %rXX, 10(%rYY)`

# mov CPU



`rrmovq rA, rB`



`irmovq V, rB`



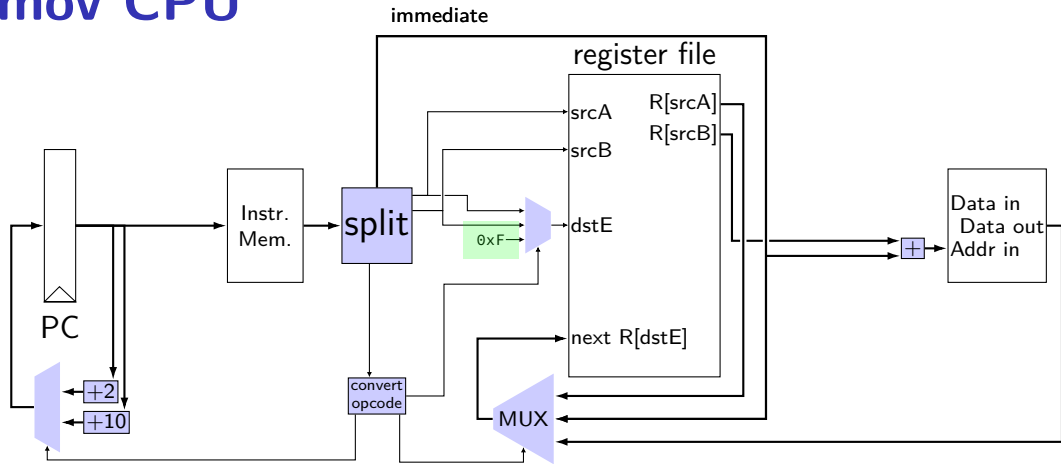
`mrmovq D(rB), rA`



`rmmovq rA, D(rB)`



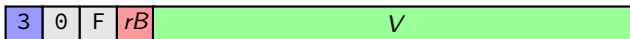
# mov CPU



`rrmovq rA, rB`



`irmovq V, rB`



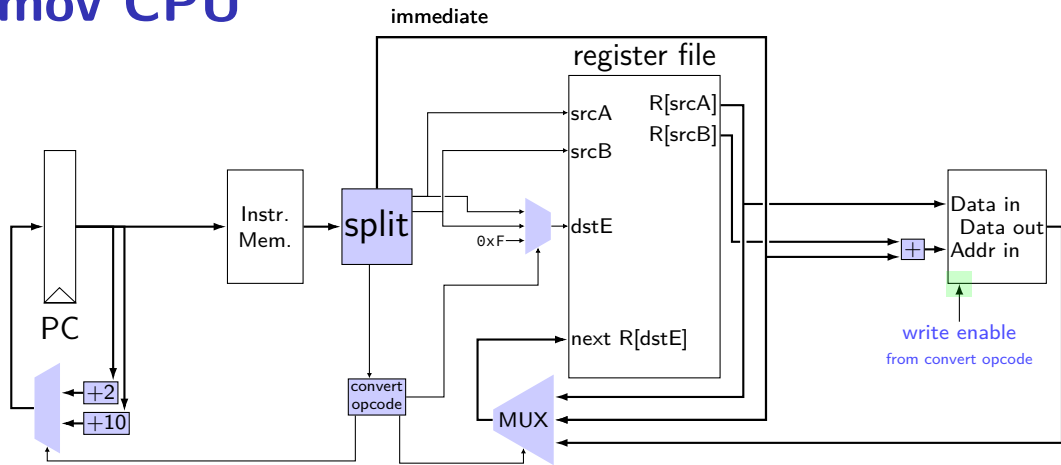
`mrmovq D(rB), rA`



`rmmovq rA, D(rB)`



# mov CPU



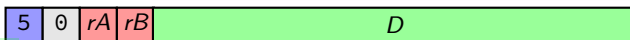
`rrmovq rA, rB`



`irmovq V, rB`



`mrmovq D(rB), rA`

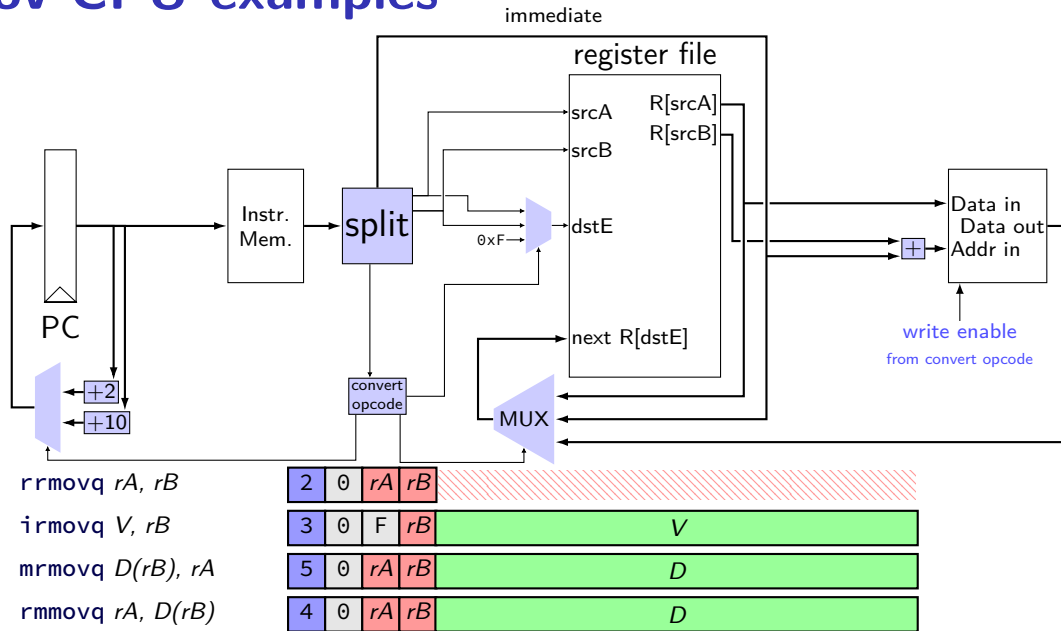


`rmmovq rA, D(rB)`



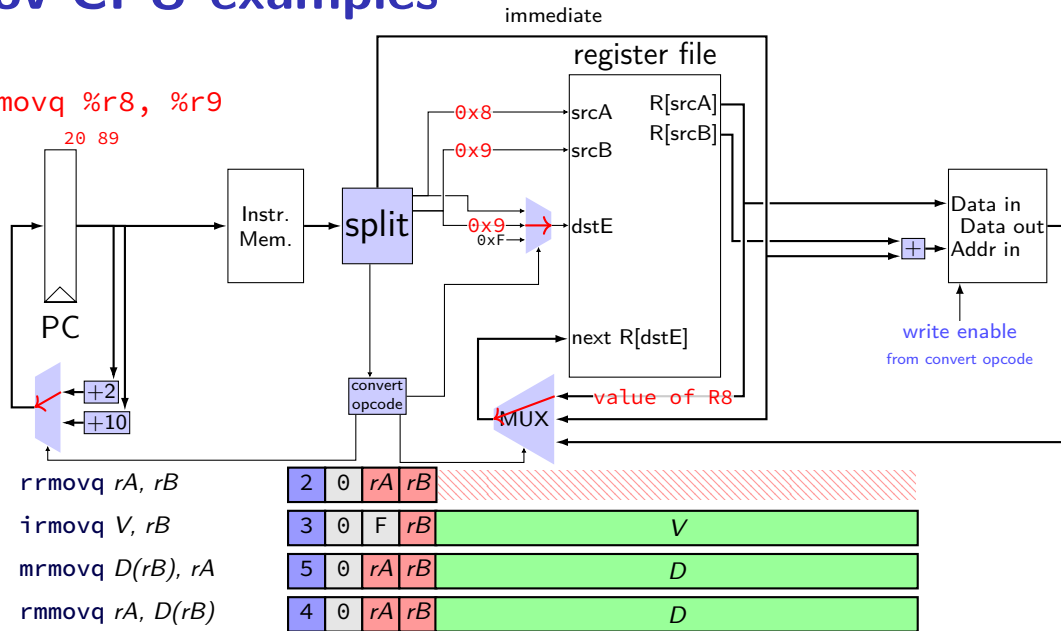


# mov CPU examples



# mov CPU examples

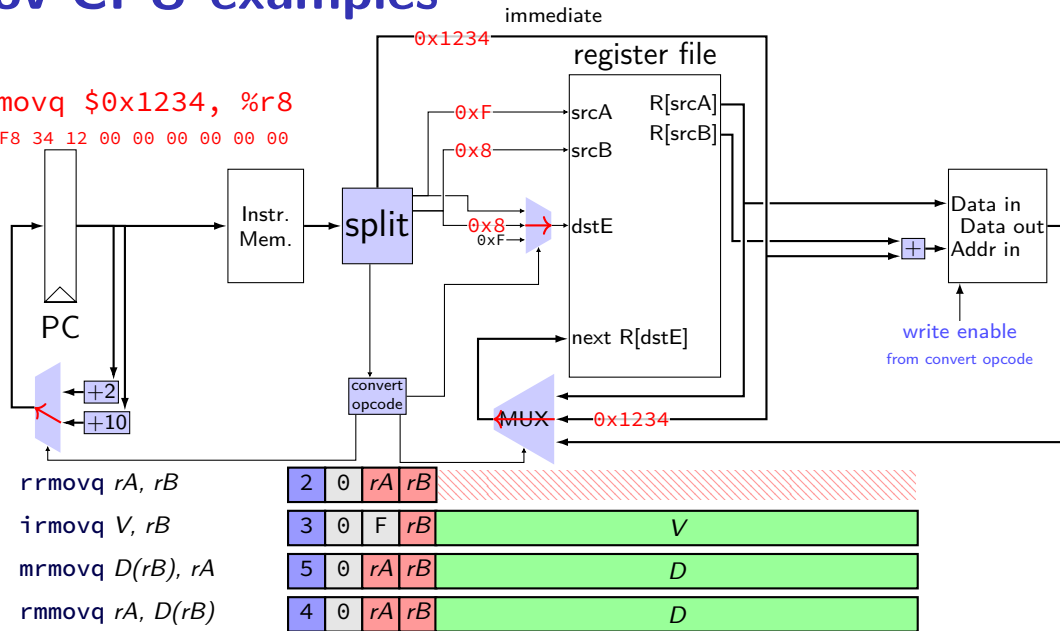
`rrmovq %r8, %r9`



# mov CPU examples

`irmovq $0x1234, %r8`

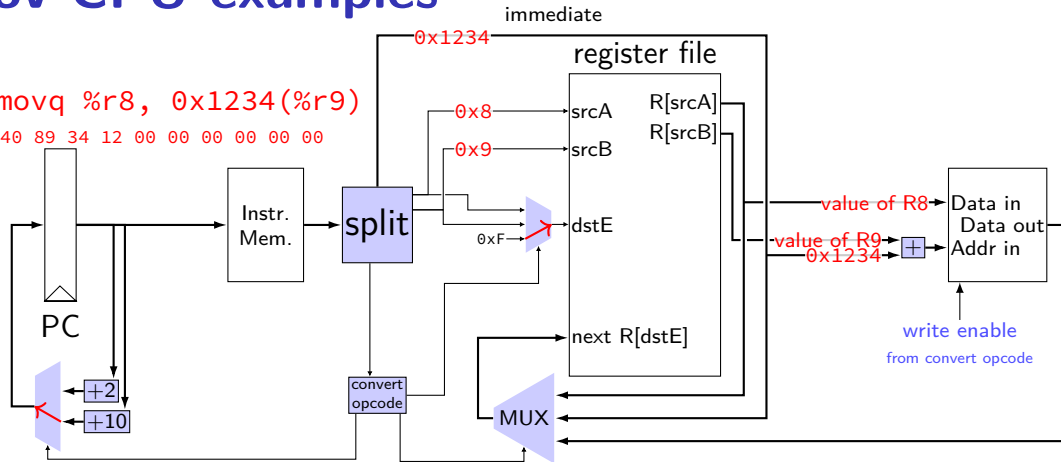
30 F8 34 12 00 00 00 00 00 00



# mov CPU examples

`rmmovq %r8, 0x1234(%r9)`

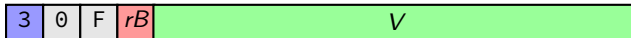
40 89 34 12 00 00 00 00 00 00



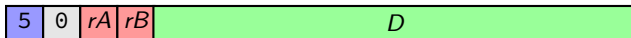
`rrmovq rA, rB`



`irmovq V, rB`



`mrmovq D(rB), rA`



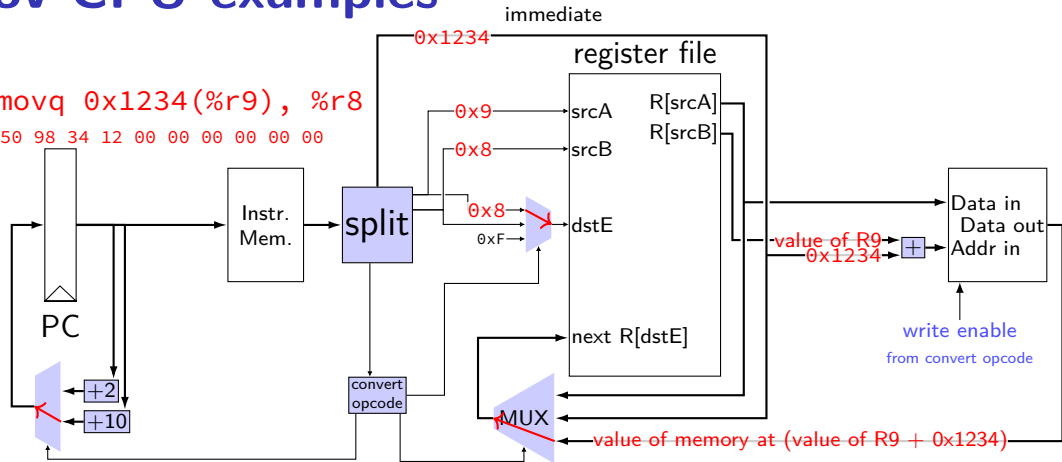
`rmmovq rA, D(rB)`



# mov CPU examples

`mrmovq 0x1234(%r9), %r8`

50 98 34 12 00 00 00 00 00 00



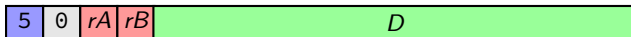
`rrmovq rA, rB`



`irmovq V, rB`



`mrmovq D(rB), rA`



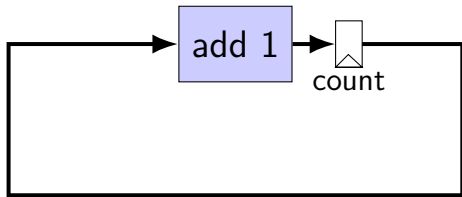
`rmmovq rA, D(rB)`



# describing hardware

how do we describe hardware?

pictures?



# circuits with pictures?

yes, something you can do

such commercial tools exist, but...

not commonly used for processors

# hardware description language

programming language for hardware

(typically) text-based representation of circuit

often abstracts away details like:

- how to build arithmetic operations from gates

- how to build registers from transistors

- how to build memories from transistors

- how to build MUXes from gates

- ...

those details also not a topic in this course



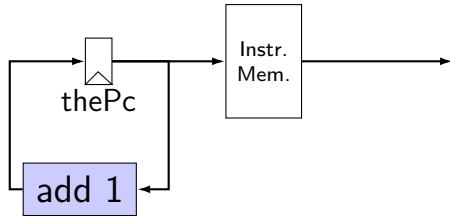
# our tool: HCLRS

built for this course

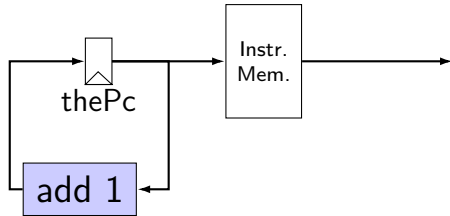
assumes you're making a processor

somewhat different from textbook's HCL

# nop CPU

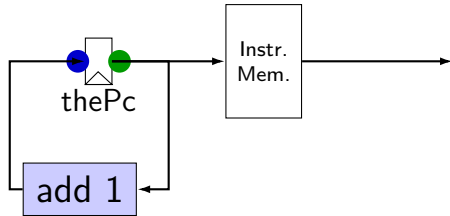


# nop CPU



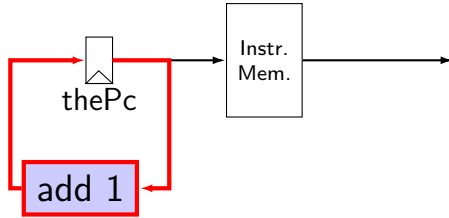
```
register pF {  
    thePc : 64 = 0;  
}
```

# nop CPU



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

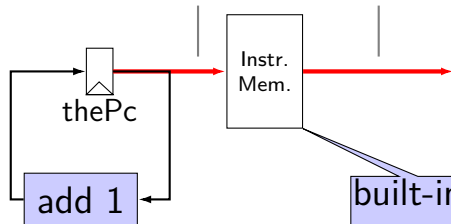
# nop CPU



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

# nop CPU

“pc” “i10bytes”

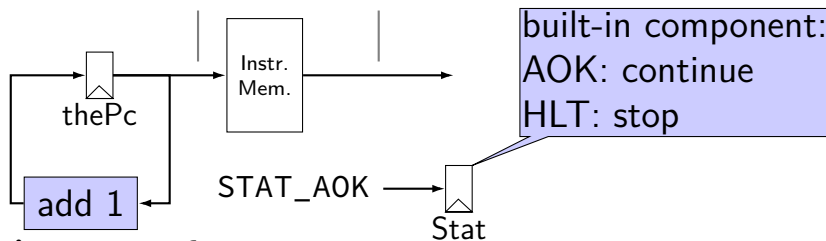


built-in component  
use is **mandatory**

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

# nop CPU

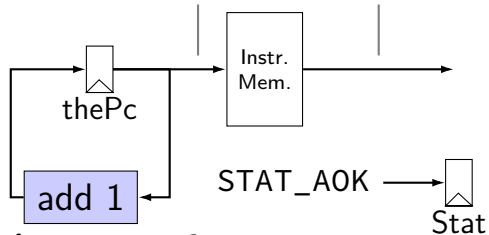
“pc” “i10bytes”



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

# nop CPU

“pc” “i10bytes”



```
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 assembly file: nops.ya:

```
nop  
nop  
nop  
nop  
nop
```

assemble using `tools/yas nops.ya` 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 |
| 0x00000000_:    10 10 10 10 10 |
+-----+
```

```
pc = 0x0; loaded [10 : nop]
```

```
+----- between cycles      1 and      2 -----+
....
```

## 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=000000000000000000 |
| used memory:     _0 _1 _2 _3 _4 _5 _6 _7   _8 _9 _a _b _c _d _e _f |
| 0x00000000_:    10 10 10 10 10           |
+-----+

```

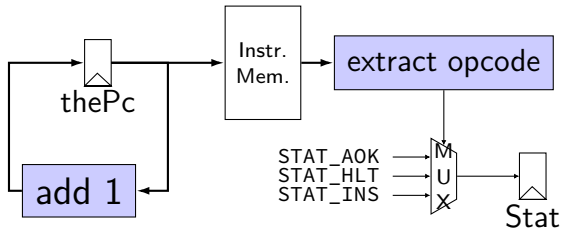
```
pc = 0x0; loaded [10 : nop]
```

```
+----- between cycles      1 and      2 -----+
....
```

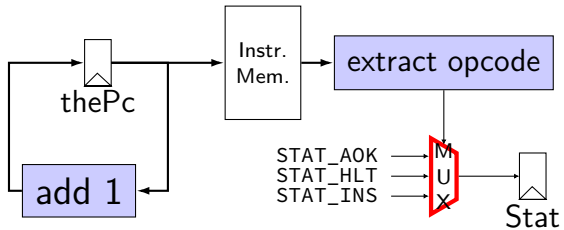




# nop/halt CPU



# nop/halt CPU



# 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
```

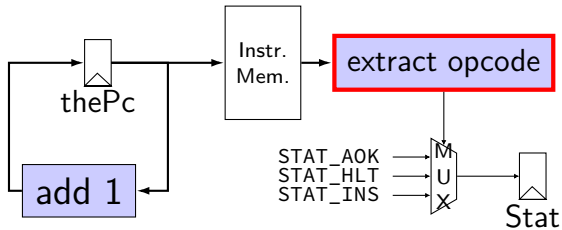
```
  x = 6: result is 2
```

```
  x = 3: result is 3
```

```
  x = 4: result is 3
```

```
  x = 1: result is 4
```

# nop/halt CPU



# subsetting bits in HCLRS

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

least significant bit is bit 0

# bit numbers and instructions

value from instruction memory in  $i10$  bytes

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
-----	---

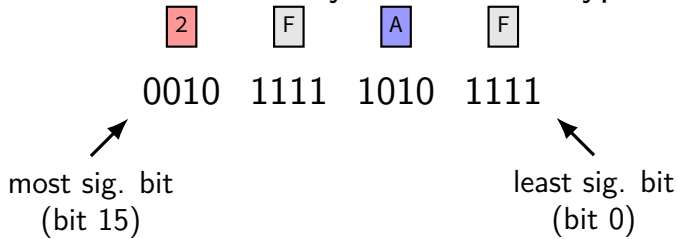
$x + 0$ : 

A	F
---	---

; at  $x + 1$ : 

2	F
---	---

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



# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC rA, rB	2	cc	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						



# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC rA, rB	2	cc	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

byte 0: bits 0-7

# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC rA, rB	2	cc	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

least sig. 4 bits of byte 0: bits 0–4

# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC rA, rB	2	cc	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

most sig. 4 bits of byte 0: bits 4–8

# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC <i>rA</i> , <i>rB</i>	2	cc	<i>rA</i>	<i>rB</i>						
irmovq <i>V</i> , <i>rB</i>	3	0	F	<i>rB</i>	<i>V</i>					
rmmovq <i>rA</i> , <i>D(rB)</i>	4	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
mrmmovq <i>D(rB)</i> , <i>rA</i>	5	0	<i>rA</i>	<i>rB</i>	<i>D</i>					
OPq <i>rA</i> , <i>rB</i>	6	fn	<i>rA</i>	<i>rB</i>						
jCC <i>Dest</i>	7	cc	<i>Dest</i>							
call <i>Dest</i>	8	0	<i>Dest</i>							
ret	9	0								
pushq <i>rA</i>	A	0	<i>rA</i>	F						
popq <i>rA</i>	B	0	<i>rA</i>	F						

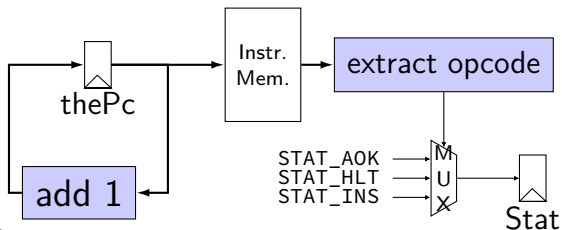
most sig. 4 bits of byte 1: bits 12–16

# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC rA, rB	2	cc	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jCC Dest	7	cc			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

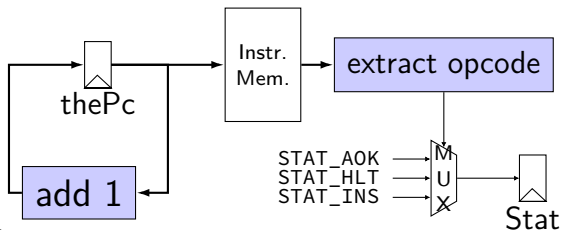
least sig. 4 bits of byte 1: bits 8–12

# nop/halt CPU



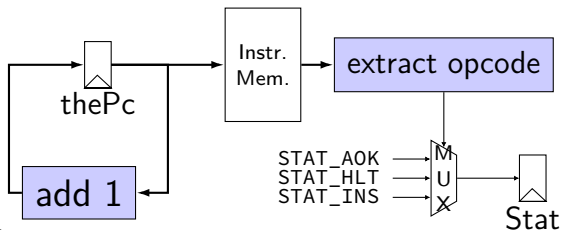
```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; // (default case)
];
```

# 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)
];
```

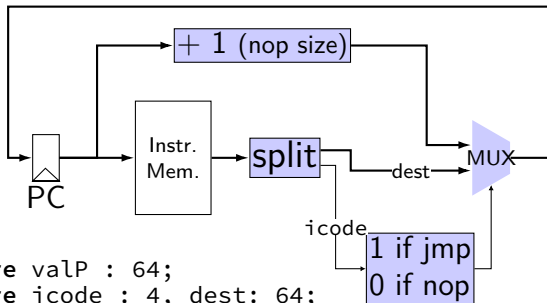
# nop/halt CPU



```
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)
];
```



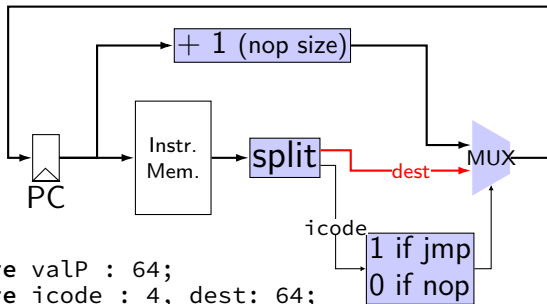
# nop/jmp CPU



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

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

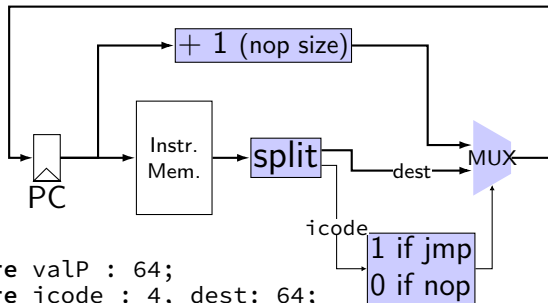
# nop/jmp CPU



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

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

# nop/jmp CPU



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

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

# running nop/jmp/halt

`nopjmp.ys:`

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

...assemble with `yas`

# nopjmp.yo

nopjmp.yo:

0x000:	10		nop
0x001:	70130000000000000000		jmp C
0x00a:	701c0000000000000000	B:	jmp D
0x013:	700a0000000000000000	C:	jmp B
0x01c:	10	D:	nop
0x01d:	10		nop
0x01e:	00		halt

# nopjmp.yo

nopjmp.yo:

0x000:	10		nop
0x001:	70130000000000000000		jmp C
0x00a:	701c0000000000000000	B:	jmp D
0x013:	700a0000000000000000	C:	jmp B
0x01c:	10	D:	nop
0x01d:	10		nop
0x01e:	00		halt

# running nopjump.yo

```
$ ./hclrs nopjump_cpu.hcl nopjump.yo
```

```
...
```

```
...
```

```
+----- (end of halted state) -----+
```

```
Cycles run: 7
```

# debugging mode

```
+----- between cycles      0 and      1 -----+
| RAX:           0   RCX:           0   RDX:           0   |
| RBX:           0   RSP:           0   RBP:           0   |
| RSI:           0   RDI:           0   R8:           0   |
| R9:            0   R10:          0   R11:          0   |
| R12:          0   R13:          0   R14:          0   |
| register pP(N)  thePc=00000000000000000000000000000000 |
| used memory:   _0 _1 _2 _3 _4 _5 _6 _7  _8 _9 _a _b  _c _d _e _f |
| 0x00000000_:  10 70 13 00  00 00 00 00  00 00 70 1c  00 00 00 00 |
| 0x00000001_:  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	0x0000000000000001370
i10bytes	0x000000000000000137010
icode	0x1
pc	0x00000000000000000000
P_thePc	0x00000000000000000000
p_thePc	0x00000000000000000001
Stat	0x1
valP	0x00000000000000000001

```
.----- 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 |
| 0x00000000_:      10 70 13 00  00 00 00 00  00 00 70 1c  00 00 00 00 |
| 0x00000001_:      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	0x00000000000001370
i10bytes	0x0000000000000137010
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 |
| 0x00000000_:    10 70 13 00  00 00 00 00  00 00 70 1c  00 00 00 00 |
| 0x00000001_:    00 00 00 70  0a 00 00 00  00 00 00 00  10 10 00  |
+----- (end of halted state) -----+
```

```
Cycles run: 7
```

# differences from book

**wire** not **bool** or **int**

book uses names like `val C` — 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 **bool** or **int**

book uses names like `val C` — 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 **bool** or **int**

book uses names like `val C` — not required!

author's environment limited adding new wires

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

implement your own ALU

# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory



# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory

# register banks

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

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

input signals: x\_foo, x\_bar

output signals: Y\_foo, Y\_bar

each value has width in bits

each value has initial value — *mandatory*

some other signals — stall, bubble

later in semester

# register banks

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

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

input signals: x\_foo, x\_bar

output signals: Y\_foo, Y\_bar

each value has **width in bits**

each value has initial value — *mandatory*

some other signals — stall, bubble

later in semester

# register banks

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

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

input signals: x\_foo, x\_bar

output signals: Y\_foo, Y\_bar

each value has width in bits

each value has **initial value** — *mandatory*

some other signals — stall, bubble

later in semester

# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory

# wires

```
wire wireName : wireWidth;
```

```
wireName = ...;
```

```
... = wireName;
```

```
... = wireName;
```

things that can accept/produce a signal

- some created implicitly – e.g. by creating register

- some builtin — supplied components (like instruction memory)

assignment — connecting wires

# wires and order

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

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

# wires and order

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

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



# wires and order

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

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

order doesn't matter  
wire is connected or not connected

# wires and width

```
wire bigValueOne: 64;  
wire bigValueTwo: 64;  
wire smallValue: 32;  
bigValueOne = smallValue; /* ERROR */  
smallValue = bigValueTwo; /* ERROR */  
...  
wire bigValueOne: 64;  
wire bigValueTwo: 64;  
wire smallValue: 32;  
  
smallValue = bigValueTwo[0..32]; /* OKAY */
```

# constants and width

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

0b1010 — 4 bits (binary 1010 = 10)

most built-in constants STAT\_AOK, NOP, etc. have widths

# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory

# Stat register

how do we stop the machine?

hard-wired mechanism — Stat register

possible values:

STAT\_AOK — keep going

STAT\_HLT — stop, normal shutdown

STAT\_INS — invalid instruction

...(and more errors)

must be set

determines if **simulator** keeps going

# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory

# program memory

input wire: pc

output wire: i10bytes

80-bits wide (10 bytes)

bit 0 — least significant bit of first byte  
(width of largest instruction)

# program memory

input wire: pc

output wire: i10bytes

80-bits wide (10 bytes)

bit 0 — least significant bit of first byte  
(width of largest instruction)

what about less than 10 byte instructions?

just don't use the extra bits



# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory

# register file

four **register number** inputs (4-bit):

sources: reg\_srcA, reg\_srcB

destinations: reg\_dstM reg\_dstE

no write or no read? register number 0xF (REG\_NONE)

two **register value** inputs (64-bit):

reg\_inputE, reg\_inputM

two **register output** values (64-bit):

reg\_outputA, reg\_outputB

## example using register file: add CPU

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

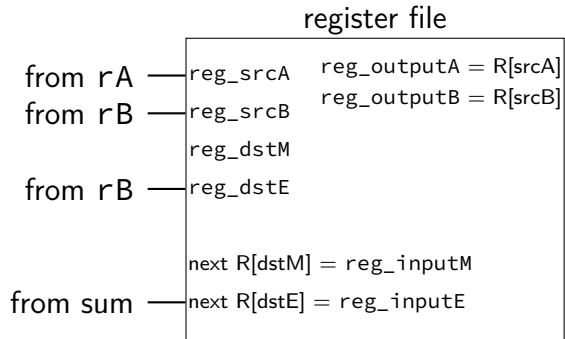
# register file picture

## register file

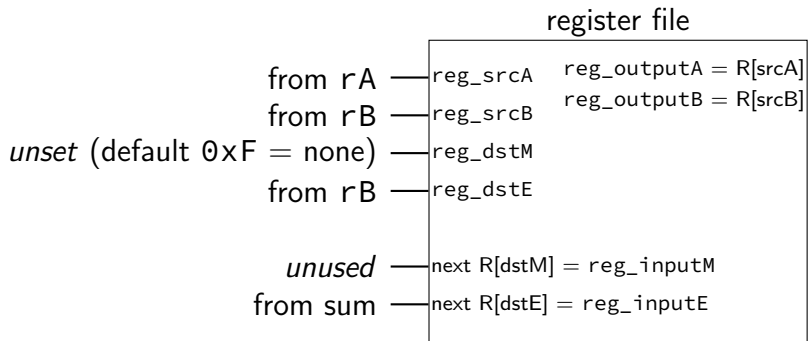
```
reg_srcA   reg_outputA = R[srcA]
reg_srcB   reg_outputB = R[srcB]
reg_dstM
reg_dstE

next R[dstM] = reg_inputM
next R[dstE] = reg_inputE
```

# register file picture

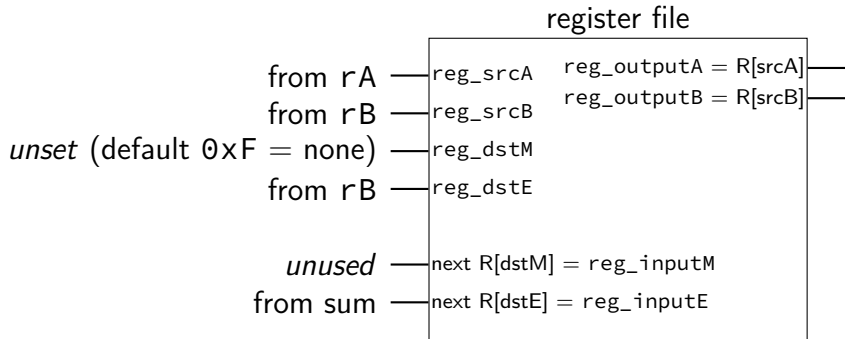


# register file picture





# register file picture



# things in HCLRS

register banks

wires

things for our processor:

- Stat register

- instruction memory

- the register file

- data memory

# data memory

input address: mem\_addr

input value: mem\_input

output value: mem\_output

read/write enable: mem\_readbit, mem\_writebit

# reading from data memory

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

mem\_output has value **in same cycle**

# reading from data memory

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

mem\_output has value **in same cycle**

# reading from data memory

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

mem\_output has value **in same cycle**

## writing to data memory

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

memory updated for next cycle

# writing to data memory

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

memory updated for next cycle



# writing to data memory

```
mem_addr = 0x12345678;
```

```
mem_input = ...;
```

```
mem_readbit = 0;
```

```
mem_writebit = 1;
```

memory updated for next cycle

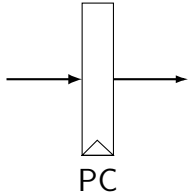




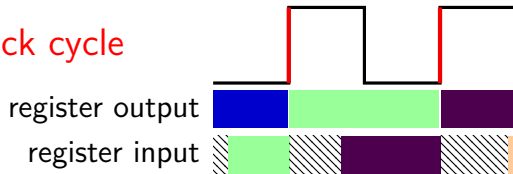


# backup slides

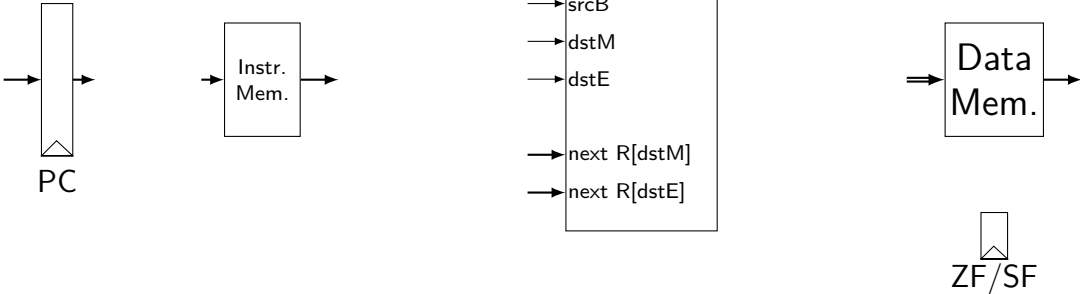
# registers



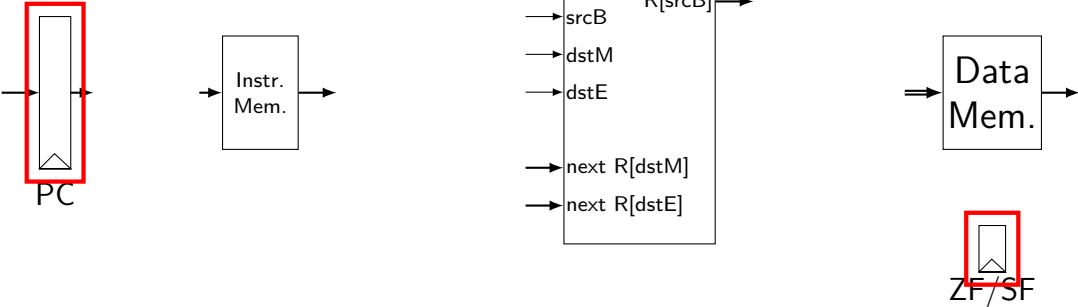
updates every **clock cycle**



# state in Y86-64

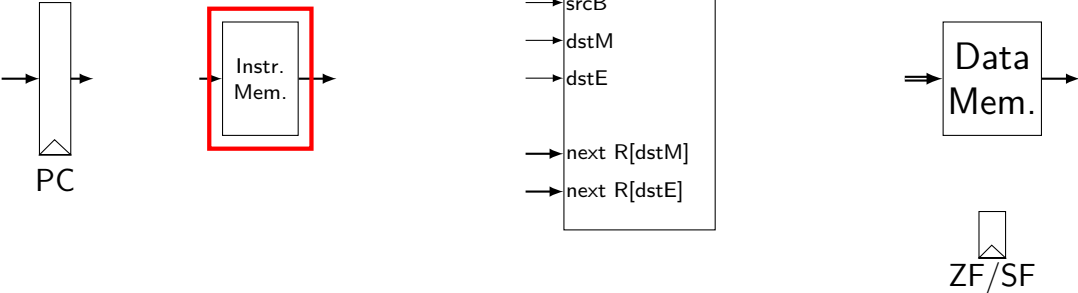


# state in Y86-64

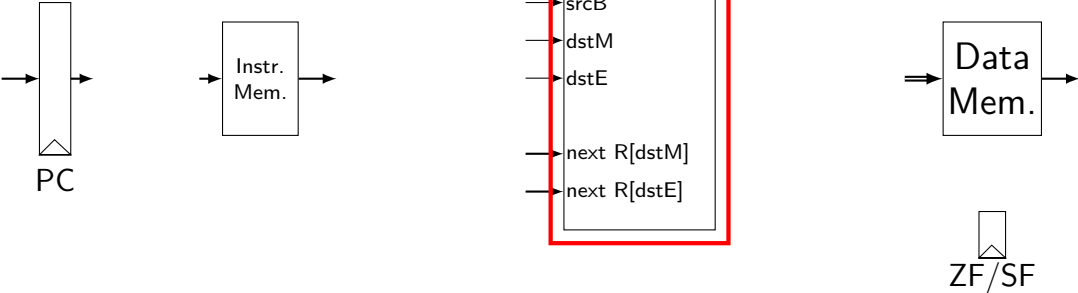




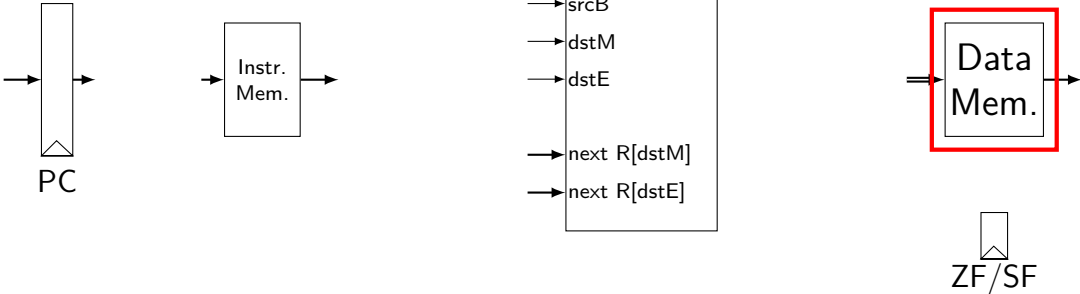
# state in Y86-64



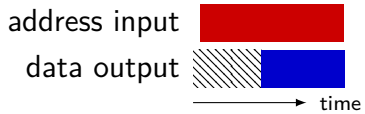
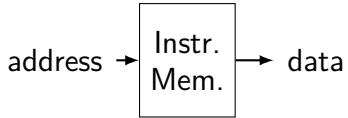
# state in Y86-64



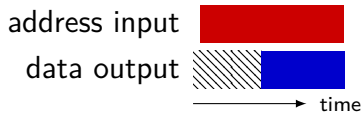
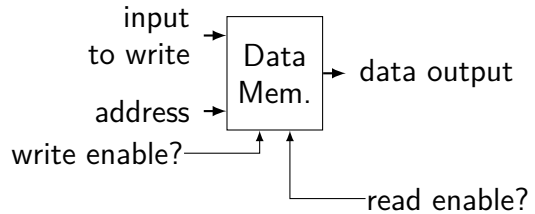
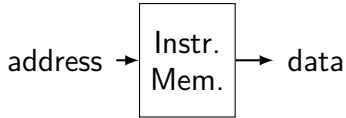
# state in Y86-64



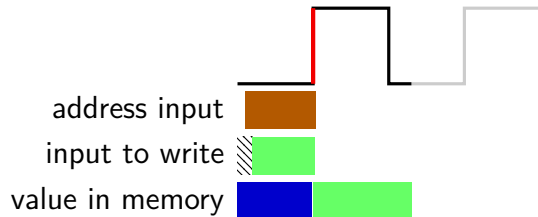
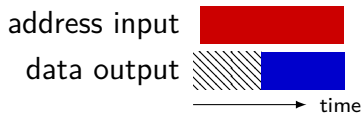
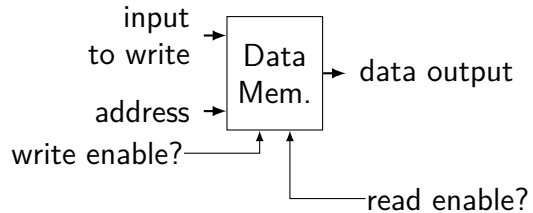
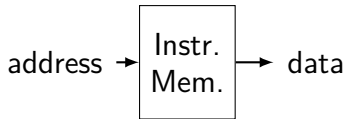
# memories



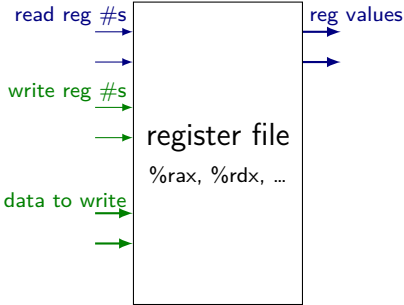
# memories



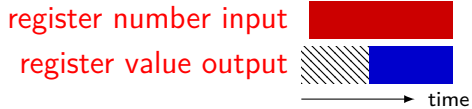
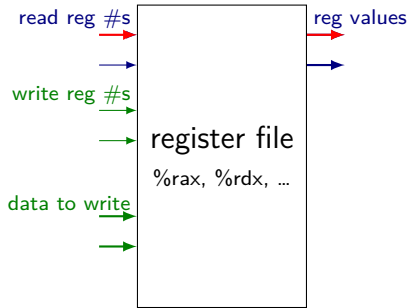
# memories



# register file

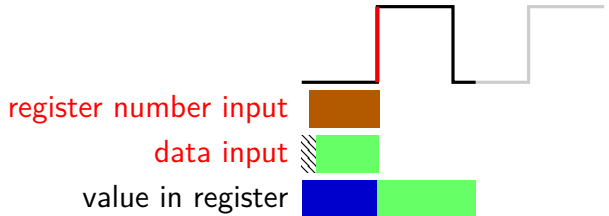
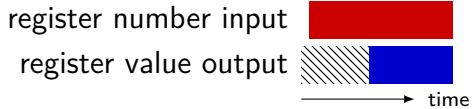
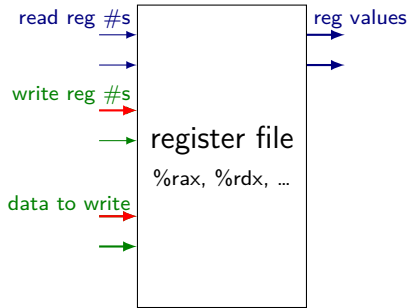


# register file

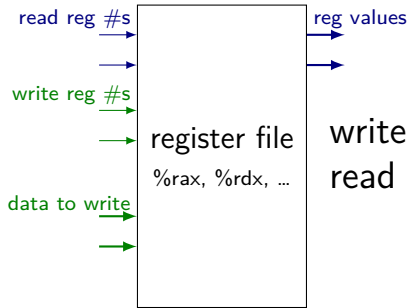




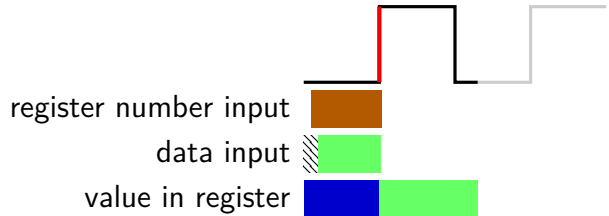
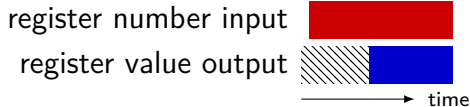
# register file



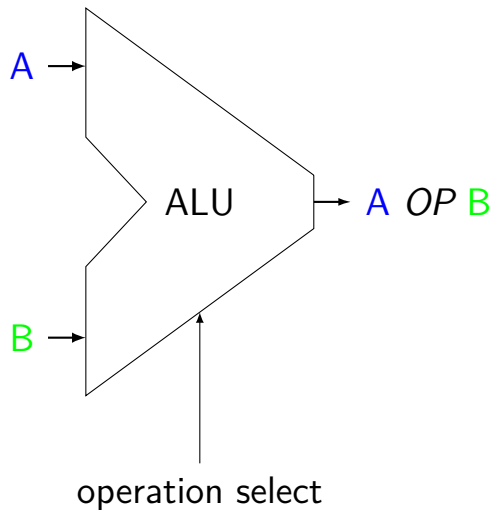
# register file



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

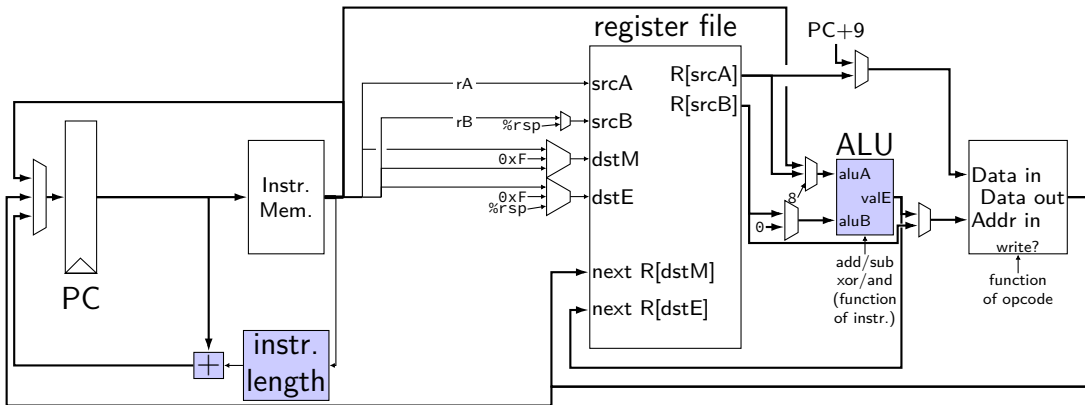


# ALUs



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

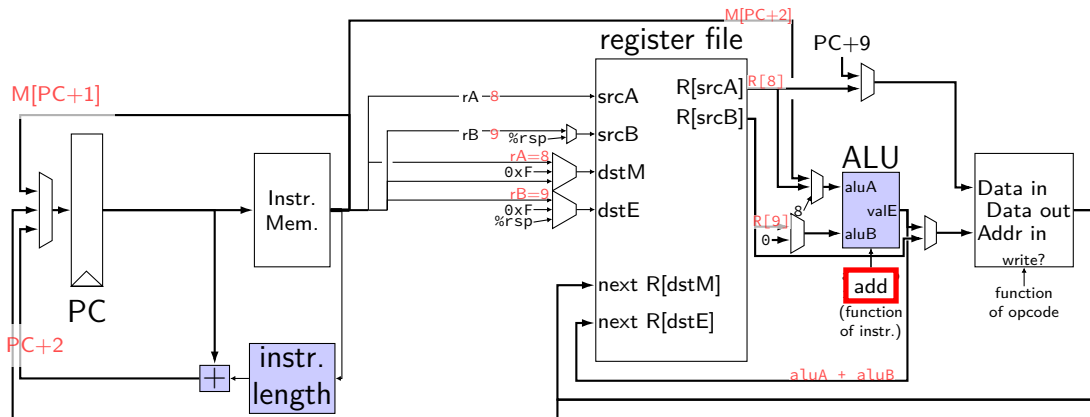
# circuit: setting MUXes



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

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

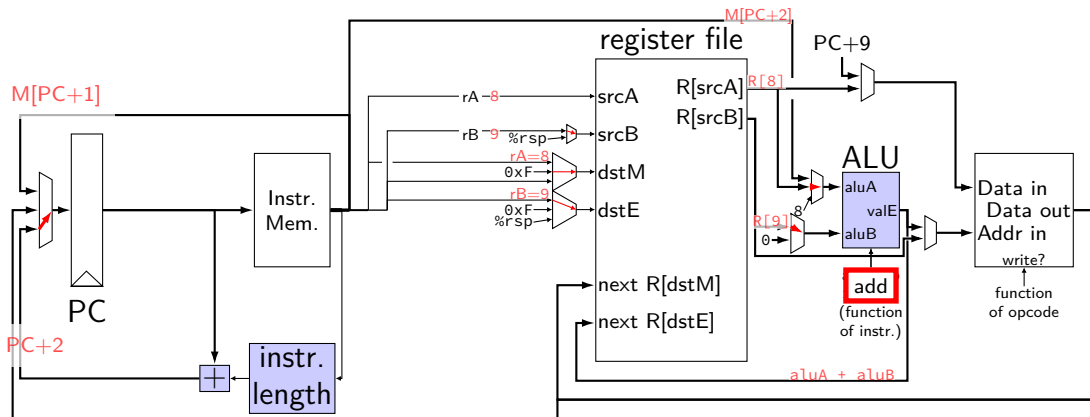
# circuit: setting MUXEs



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

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

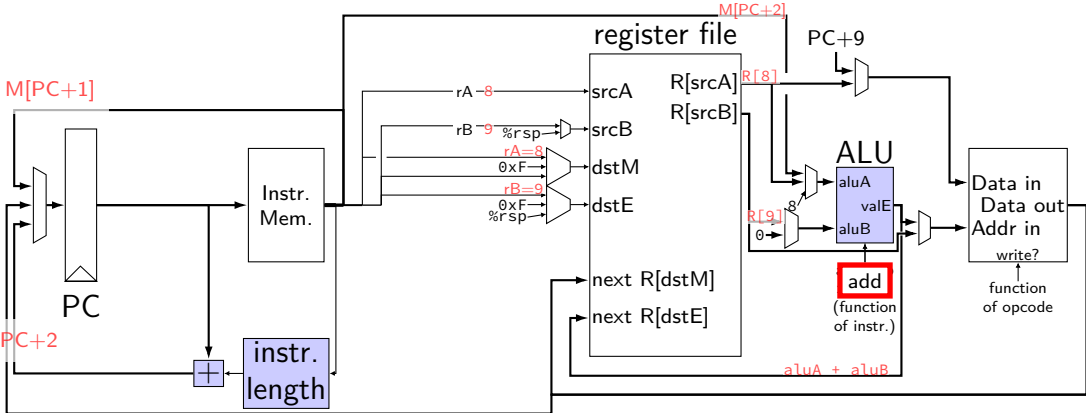
# circuit: setting MUXEs



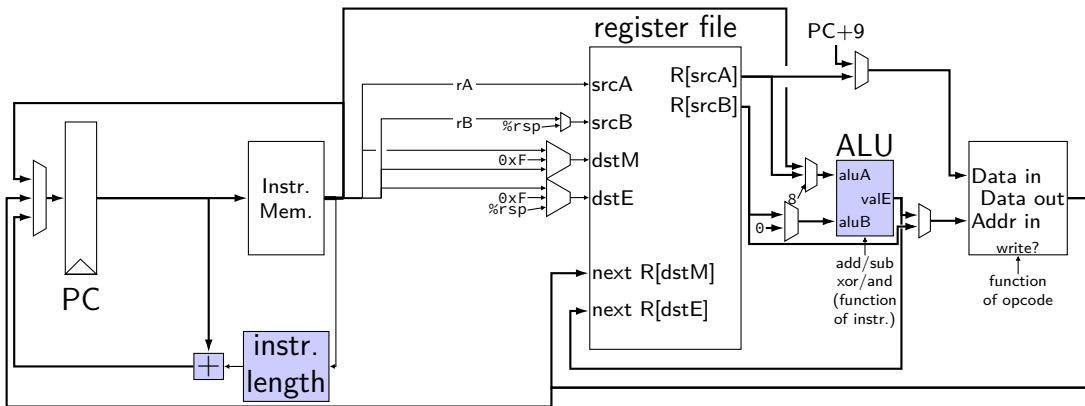
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

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

# circuit: setting MUXEs



# circuit: setting MUXes

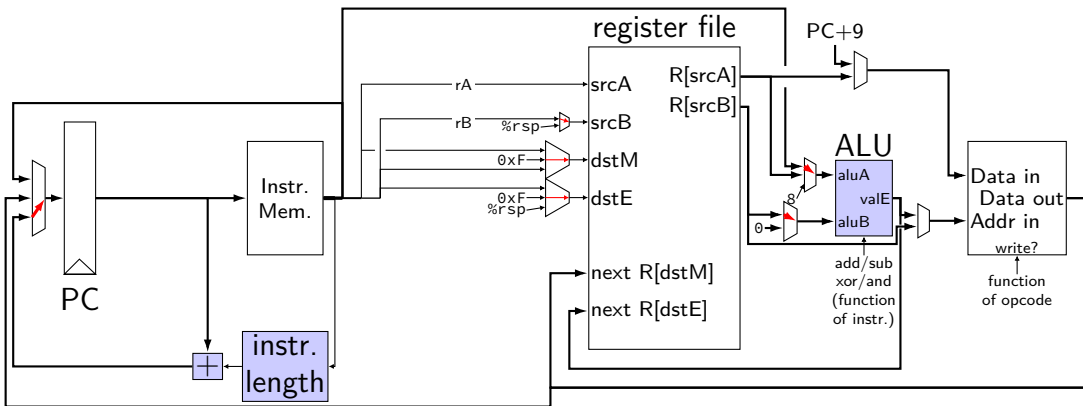


MUXes — PC,  $dstM$ ,  $dstE$ ,  $aluA$ ,  $aluB$ ,  $dmemIn$

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



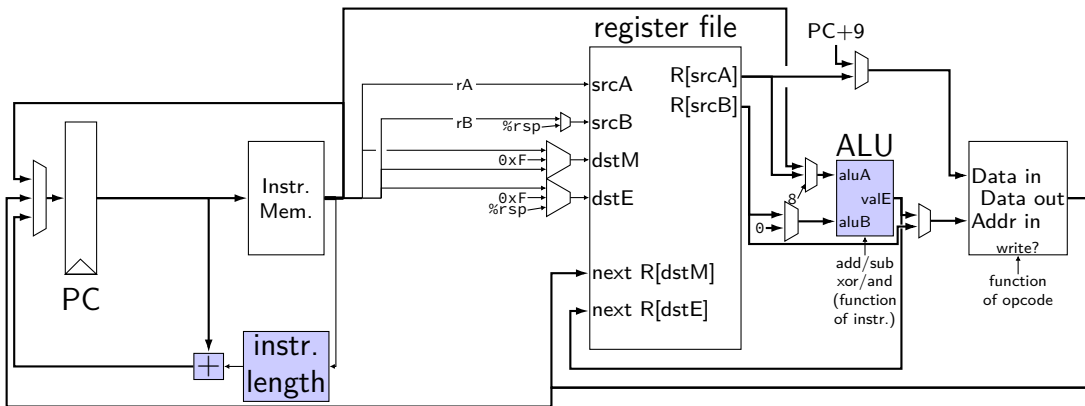
# circuit: setting MUXes



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

Exercise: what do they select for `rmmovq`?

# circuit: setting MUXes



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

Exercise: what do they select for **call**?