

# SEQ part 3 / HCLRS

# Changelog

Changes made in this version not seen in first lecture:

21 September 2017: data memory value MUX input for call is  $PC + 10$ , not PC

21 September 2017: slide 23: add input to pre R[dstE] mux for irmovq

21 September 2017: slide 26: need MUX for 0 ALU input

21 September 2017: correct a couple instances of 'HCL2D' to 'HCLRS'

# last time

single cycle processor design strategy

*conceptual stages*

- for now: ease processor design

- consider what every instruction does for a particular stage

actual timing — clock signal

- one cycle per instruction in this design

- calculations between rising edges of clock

- rising edge of clock triggers state change (register/memory values change)

# SEQ: instruction fetch

read instruction memory at PC

split into separate wires:

**icode:ifun** — opcode

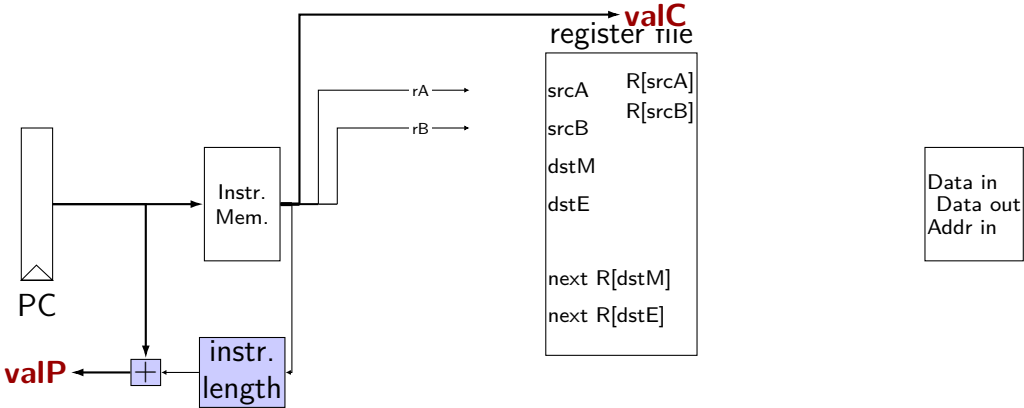
**rA, rB** — register numbers

**valC** — call target or mov displacement

compute next instruction address:

**valP** —  $PC + (\text{instr length})$

# instruction fetch

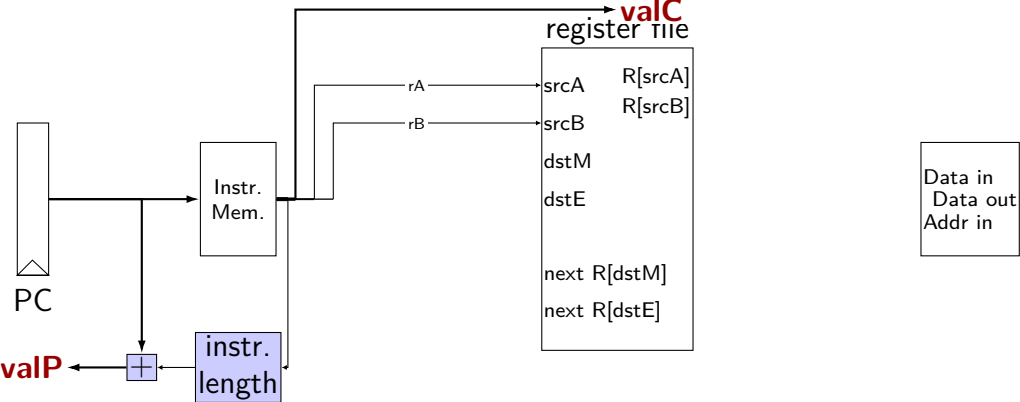


# SEQ: instruction “decode”

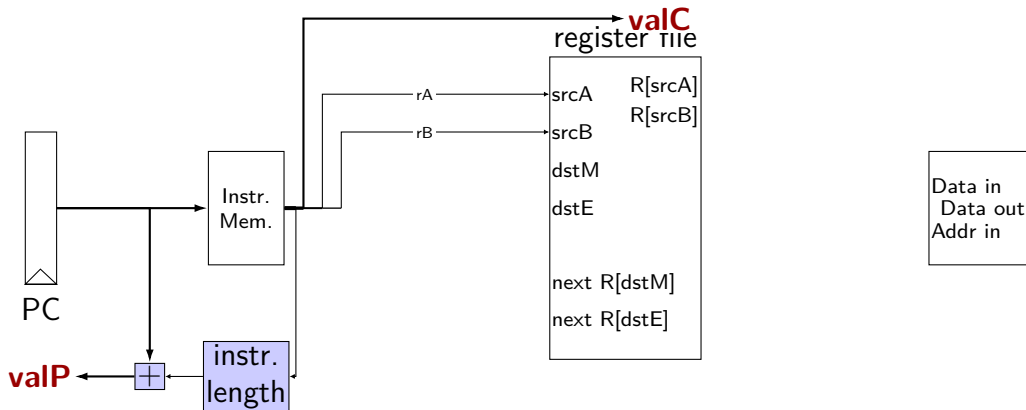
read registers

`valA`, `valB` — register values

# instruction decode (1)



# instruction decode (1)



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



# SEQ: srcA, srcB

always read rA, rB?

Problems:

- push rA

- pop

- call

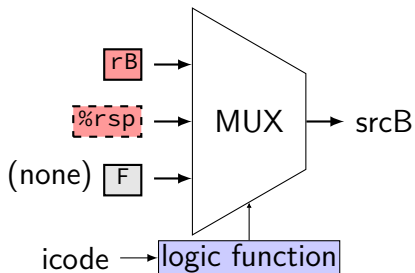
- ret

extra signals: srcA, srcB — computed input register

MUX controlled by icode

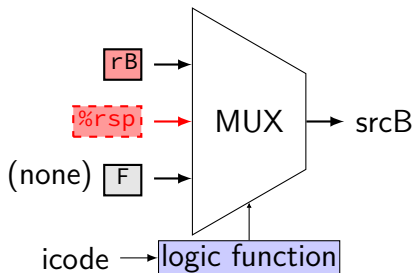
# SEQ: possible registers to read

instruction	srcA	srcB
<b>halt, nop, jCC, irmovq</b>	none	none
<b>cmovCC, rrmovq</b>	rA	none
<b>rrmovq</b>	none	rB
<b>rmmovq, OPq</b>	rA	rB
<b>call, ret</b>	none?	%rsp
<b>pushq, popq</b>	rA	%rsp



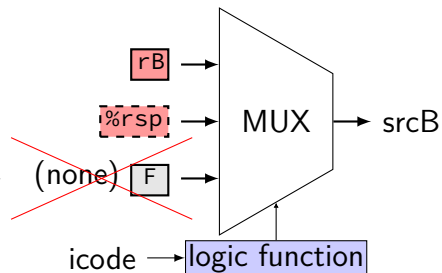
# SEQ: possible registers to read

instruction	srcA	srcB
<b>halt, nop, jCC, irmovq</b>	none	none
<b>cmovCC, rrmovq</b>	rA	none
<b>rrmovq</b>	none	rB
<b>rmmovq, OPq</b>	rA	rB
<b>call, ret</b>	none?	<b>%rsp</b>
<b>pushq, popq</b>	rA	<b>%rsp</b>

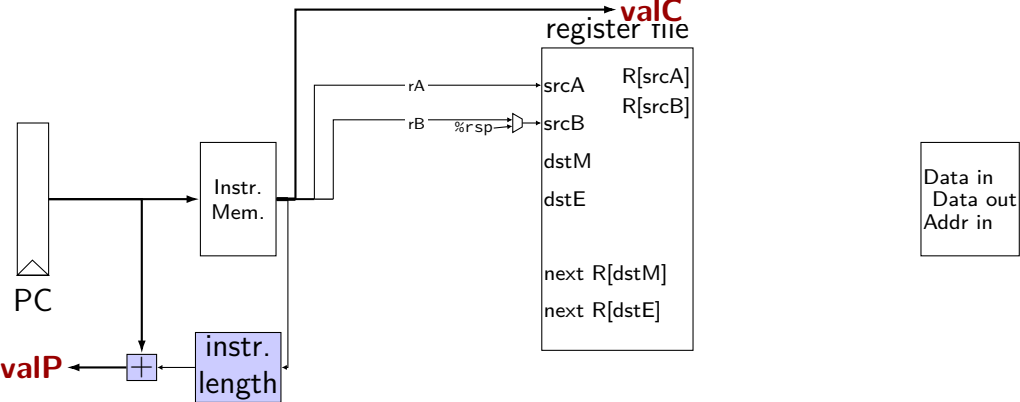


# SEQ: possible registers to read

instruction	srcA	srcB
<b>halt, nop, jCC, irmovq</b>	none	none
<b>cmovCC, rrmovq</b>	rA	none
<b>rrmovq</b>	none	rB
<b>rmmovq, OPq</b>	rA	rB
<b>call, ret</b>	none?	%rsp
<b>pushq, popq</b>	rA	%rsp



# instruction decode (2)



# SEQ: execute

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

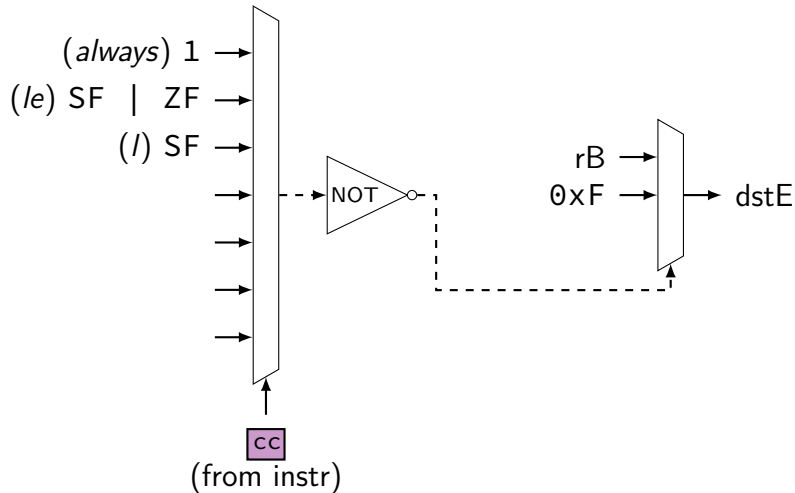
**valE** — ALU output

read prior condition codes

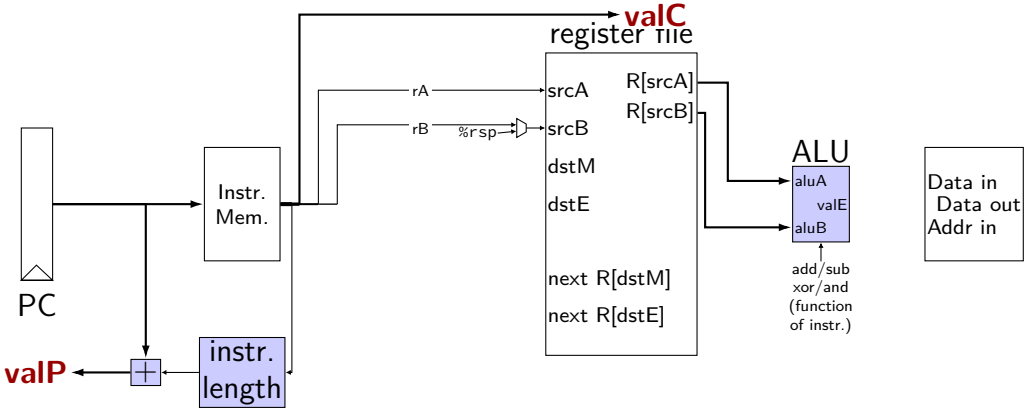
**Cnd** — condition codes based on ifun (instruction type for jCC/cmouvCC)

write new condition codes

# using condition codes: `cmov*`

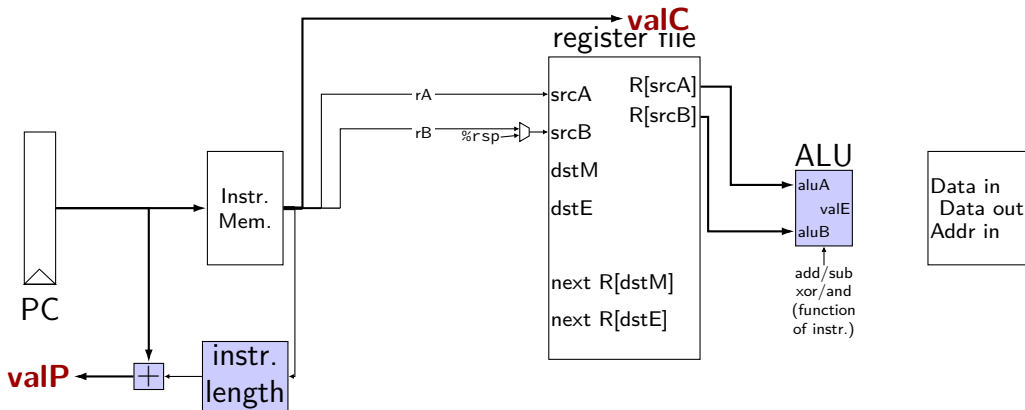


# execute (1)





# execute (1)



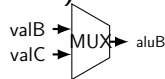
exercise: which of these instructions can this **not** work for?  
nop, addq, mrmovq, popq, call,

# SEQ: ALU operations?

ALU inputs always **valA**, **valB** (register values)?

no, inputs from instruction: (Displacement + rB)

**mrmovq**  
**rmmovq**



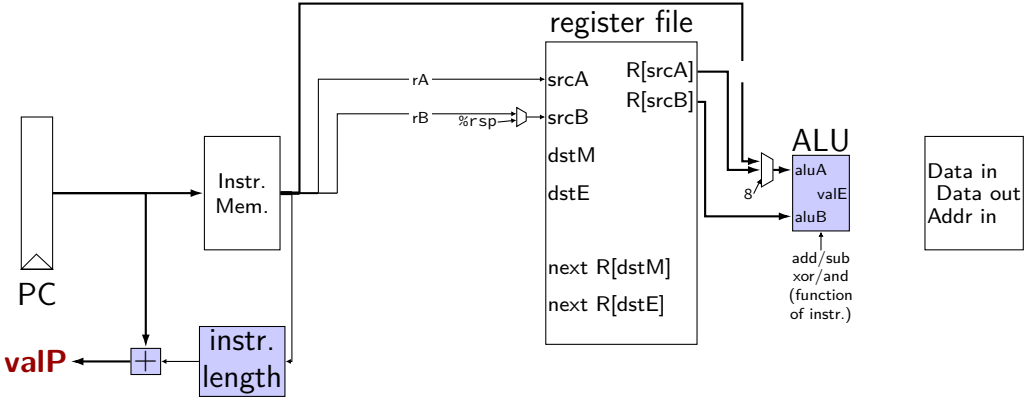
no, constants: (rsp +/- 8)

**pushq**  
**popq**  
**call**  
**ret**

extra signals: **aluA**, **aluB**

computed ALU input values

# execute (2)

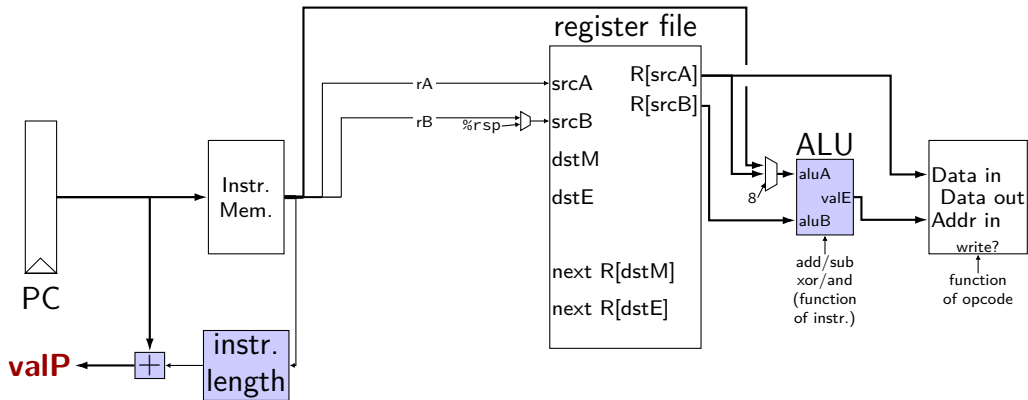


# SEQ: Memory

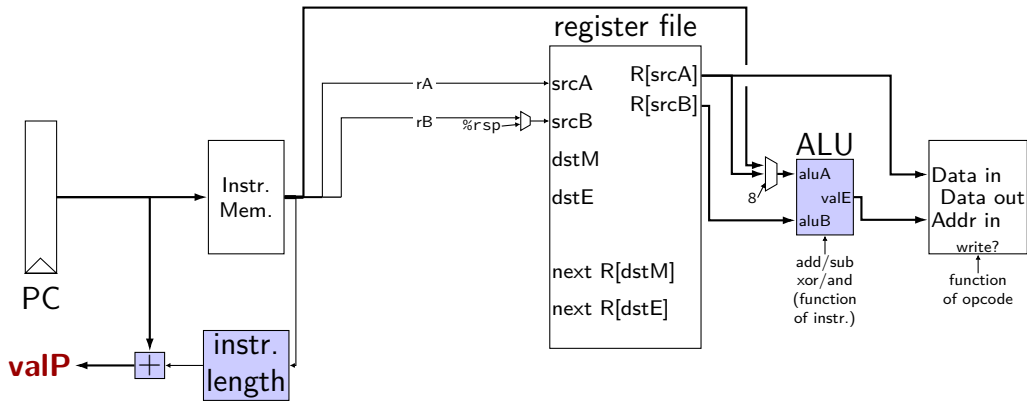
read or write data memory

**valM** — value read from memory (if any)

# memory (1)



# memory (1)



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

# SEQ: control signals for memory

read/write — read enable? write enable?

**Addr** — address

mostly ALU output

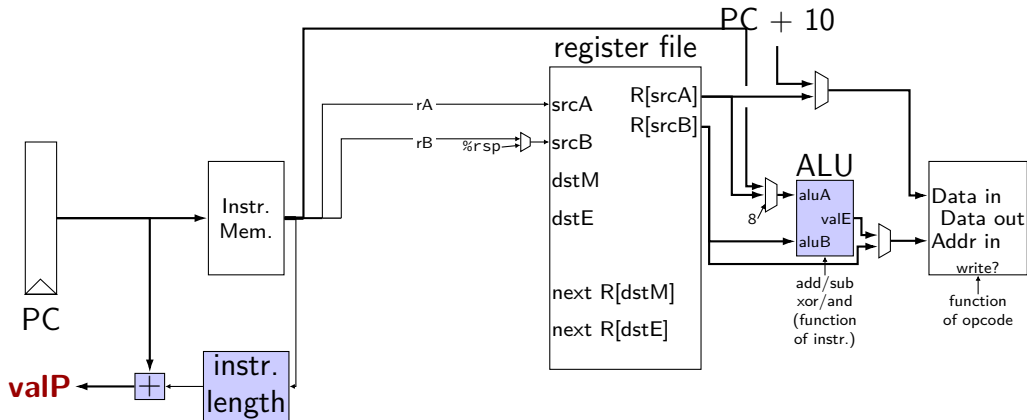
tricky cases: **popq, ret**

**Data** — value to write

mostly valB

tricky cases: **call, push**

# memory (2)

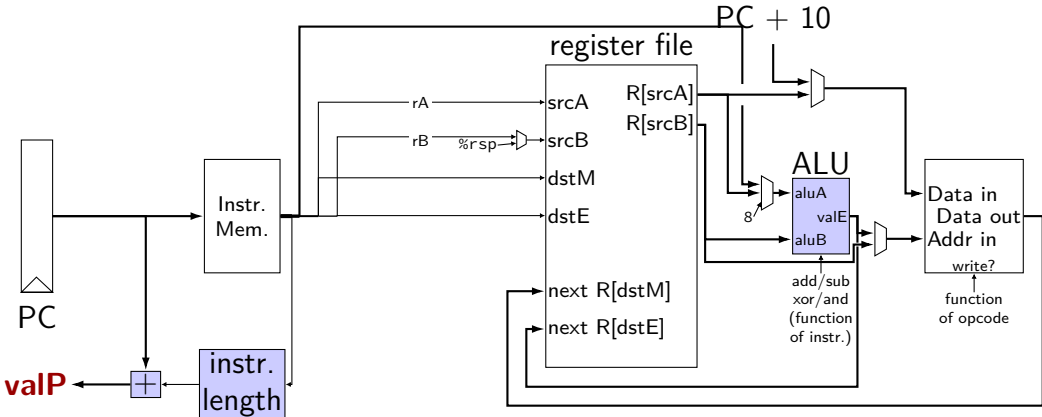




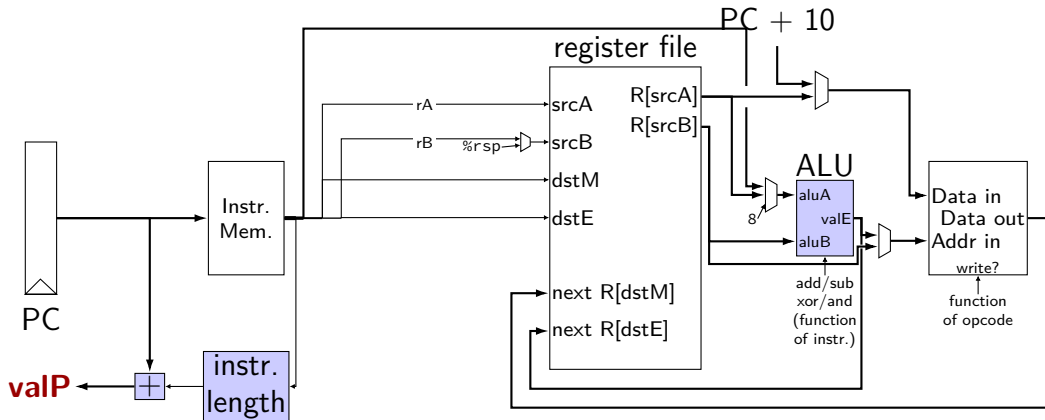
# SEQ: write back

write registers

# write back (1)



# write back (1)



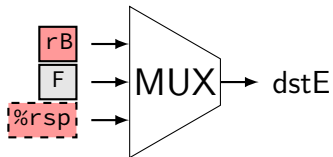
exercise: which of these instructions can this **not** work for?  
nop, pushq, mrmovq, popq, call,

# SEQ: control signals for WB

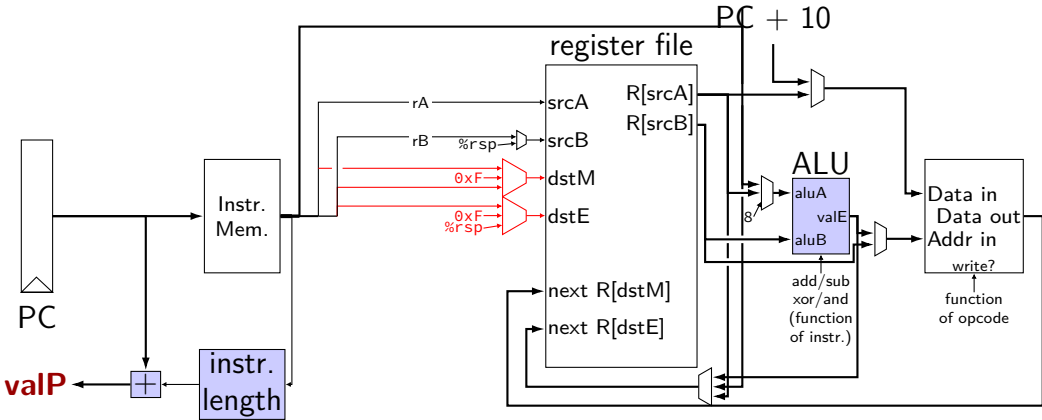
**two** write inputs — two needed by popq  
valM (memory output), valE (ALU output)

**two** register numbers  
dstM, dstE

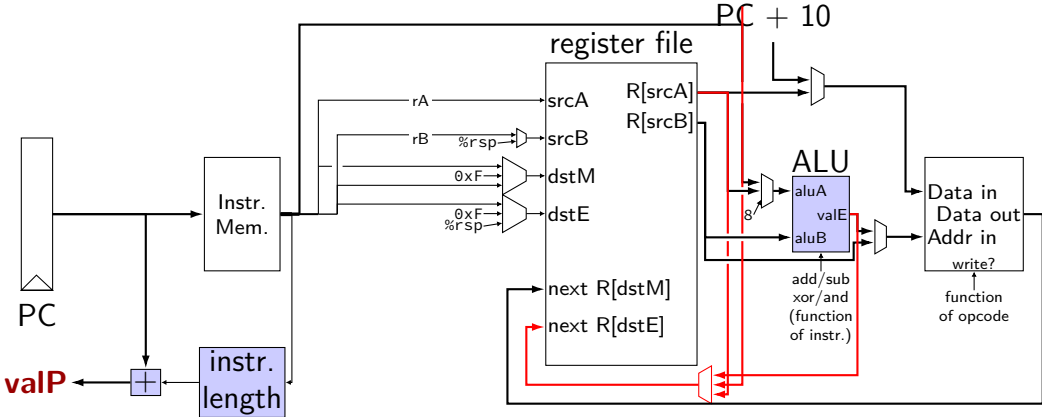
write disable — use dummy register number 0xF



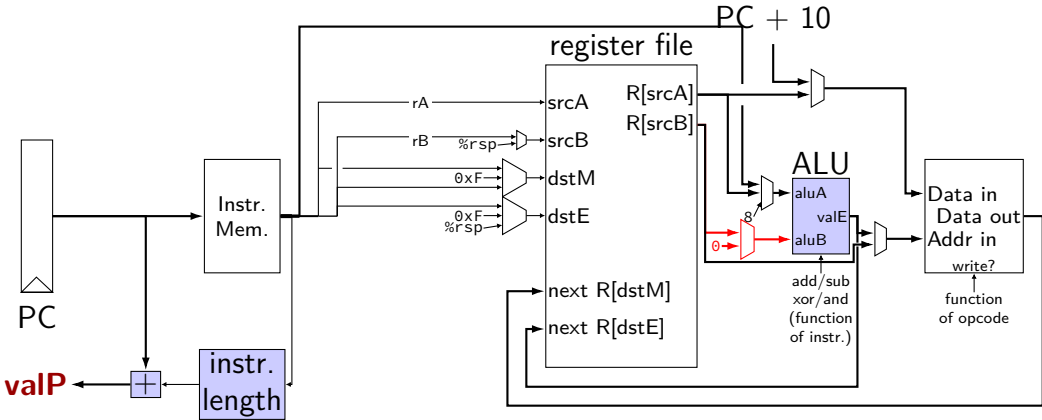
# write back (2)



# write back (3a)



# write back (3b)



# SEQ: Update PC

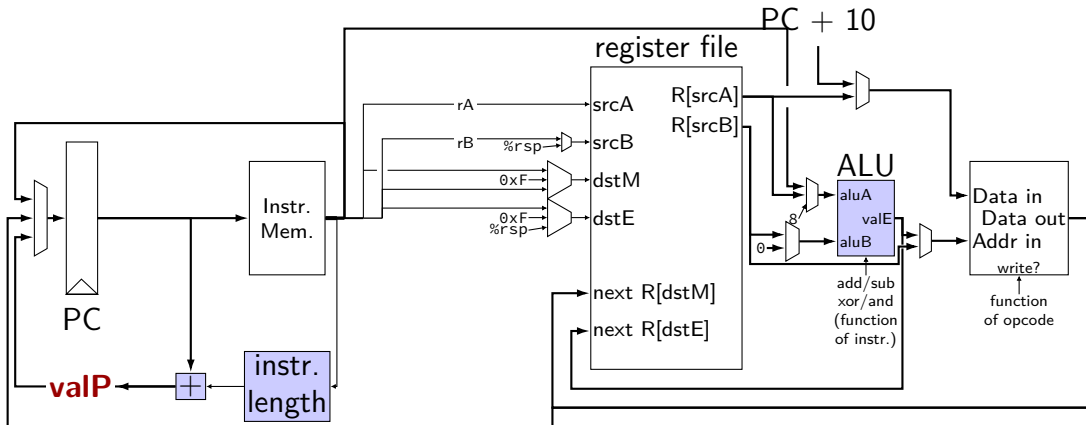
choose value for PC next cycle (input to PC register)

usually valP (following instruction)

exceptions: **call**, **jCC**, **ret**



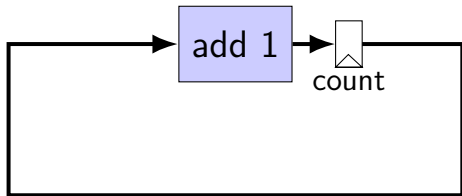
# PC update



# 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

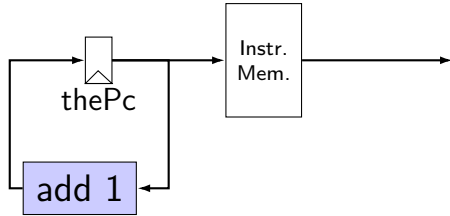
# our tool: HCLRS

built for this course

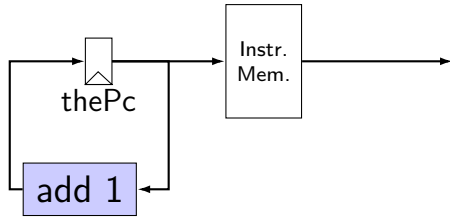
assumes you're making a processor

somewhat different from textbook's HCL

# nop CPU

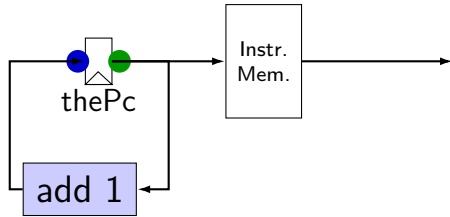


# nop CPU



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

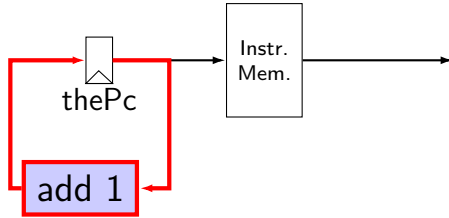
# nop CPU



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



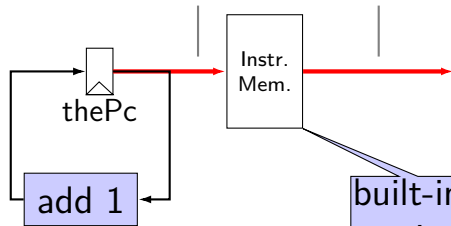
# nop CPU



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

# nop CPU

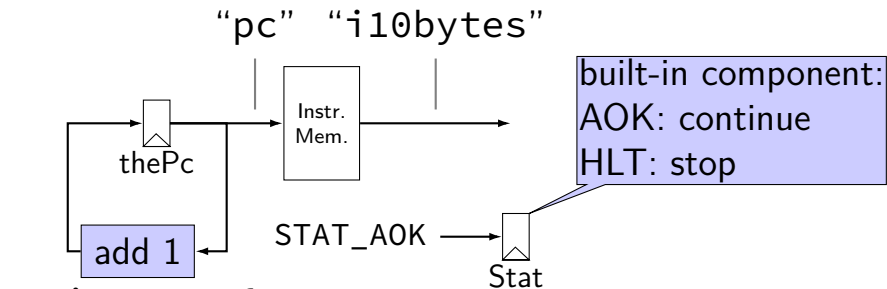
“pc” “i10bytes”



built-in component  
use is mandatory

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

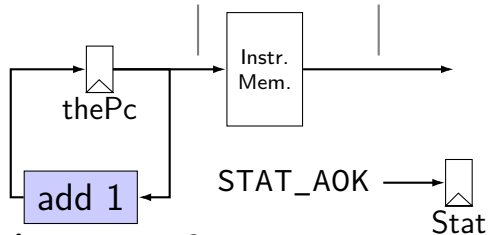
# nop CPU



```
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 traces of all assignments for debugging
-q, --quiet	only output state at the end
-t, --testing	do not output custom register banks (for autograding)
-h, --help	print this help menu
--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=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 -----+
....
```

## 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 -----+
....
```

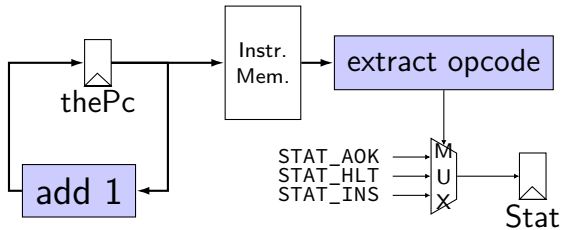
## running a simulator (2)

```
$ ./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 -----+
....
```

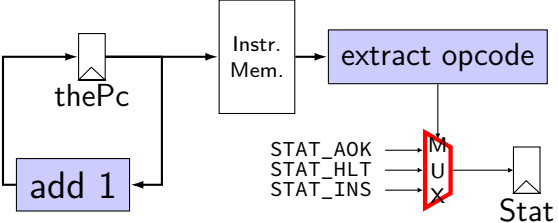
## running a simulator (2)

```
$ ./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
```

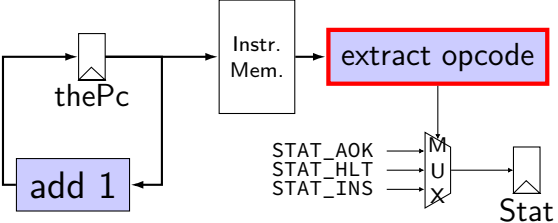
```
x = 6: result is 2
```

```
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
-----	---

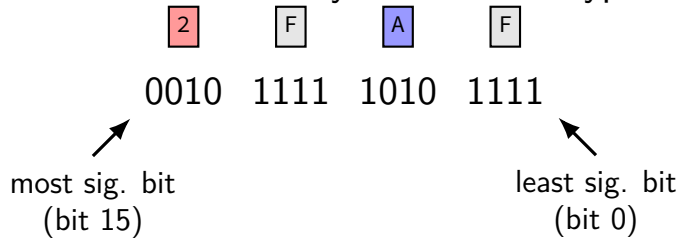
$x + 0$ : 

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

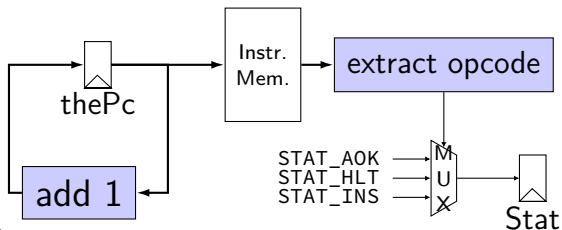


# Y86 encoding table

byte:	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq/cmovCC <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						

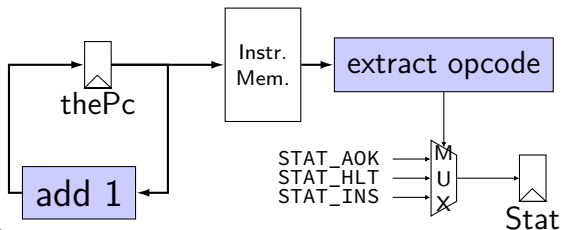
least sig. 4 bits of byte 1: bits 8–12

# nop/halt CPU



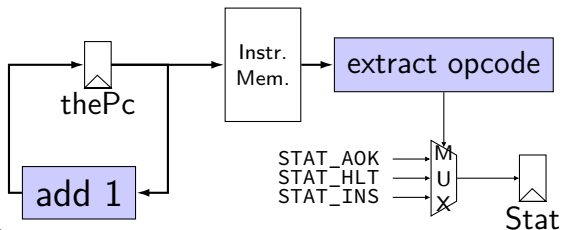
```
register pP {  
    thePc : 64 = 0;  
}  
p_thePc = P_thePc + 1;  
pc = P_thePc;  
Stat = [  
    i10bytes[4..8] == NOP : STAT_AOK;  
    i10bytes[4..8] == HALT : STAT_HLT;  
    1 : STAT_INS; (default case)  
];
```

# nop/halt CPU



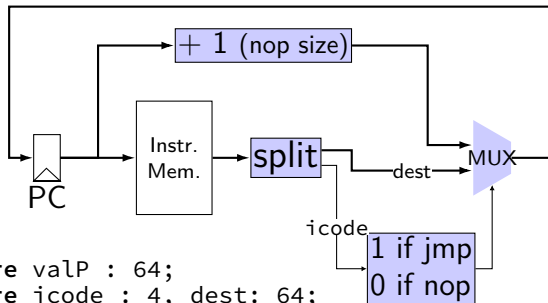
```
register pP {
    thePc : 64 = 0;
}
p_thePc = P_thePc + 1;
pc = P_thePc;
Stat = [
    i10bytes[4..8] == NOP : STAT_AOK;
    i10bytes[4..8] == HALT : STAT_HLT;
    1 : STAT_INS; (default case)
];
```

# 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)
];
```

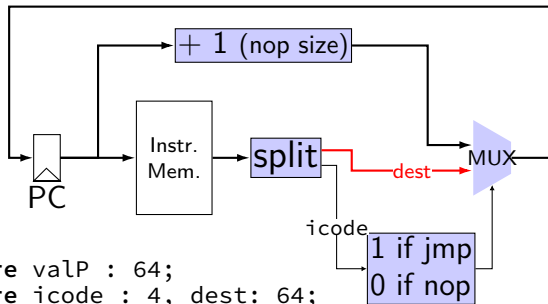
# nop/jmp CPU



```
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;
];
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;
];
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
```

# differences from book

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

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

author's environment limited adding new wires

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

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 */
```

# 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 = ...
```

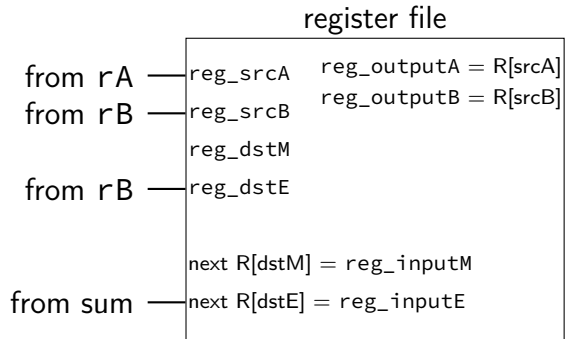
# register file picture

## register file

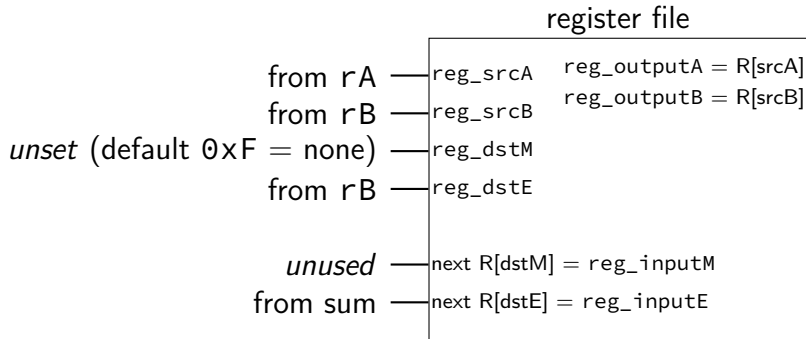
```
reg_srcA   reg_outputA = R[srcA]
reg_srcB   reg_outputB = R[srcB]
reg_dstM
reg_dstE

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

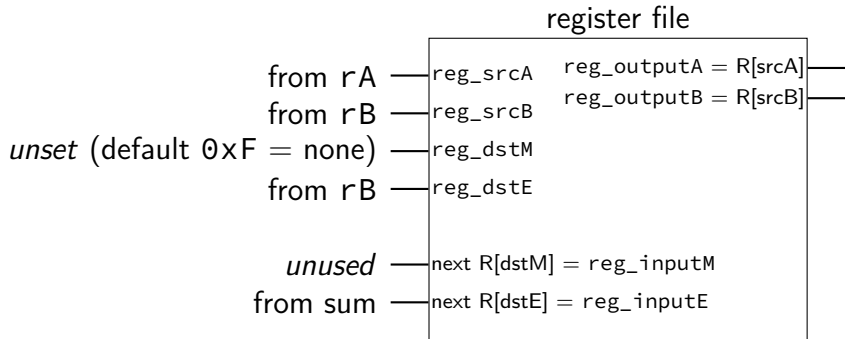
# register file picture



# 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

# 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=000000000000000000          |
| 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  |
+-----+

```

pc set to 0x0

i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)

pc = 0x0; loaded [10 : nop]

icode set to 0x1

dest set to 0x1370

Stat set to 0x1

valP set to 0x1

p\_thePc set to 0x1

```
.----- 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=000000000000000000          |
| 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  |
+-----+

```

pc set to 0x0

i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)

pc = 0x0; loaded [10 : nop]

icode set to 0x1

dest set to 0x1370

Stat set to 0x1

valP set to 0x1

p\_thePc set to 0x1

```
.----- 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=000000000000000000 |
| 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 |
+-----+
(press enter to continue)
set pc to 0x0
pc = 0x0; loaded [10 : nop]
set icode to 0x1
set valP to 0x1
set p_thePc to 0x1
set Stat to 0x1
+----- 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=00000000000000000000 |
| 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  |
+-----+

```

(press enter to continue)

set pc to 0x0

pc = 0x0; loaded [10 : nop]

set icode to 0x1

set valP to 0x1

set p\_thePc to 0x1

set Stat to 0x1

```
+----- 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
```

# HCLRS summary

declare/assign values to **wires**

**MUXes** with

```
[ test1: value1; test2: value2 ]
```

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)

## exercise: implementing ALU?

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

# on design choices

textbook choices:

memory always goes to 'M' port of register file

RSP +/- 8 uses normal ALU, not separate adders

...

do you have to do this? **no**

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

other logic: make it function correctly

# comparing to yis

```
$ ./hclrs nopjmp_cpu.hcl nopjmp.yo
```

```
...
```

```
...
```

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

```
Cycles run: 7
```

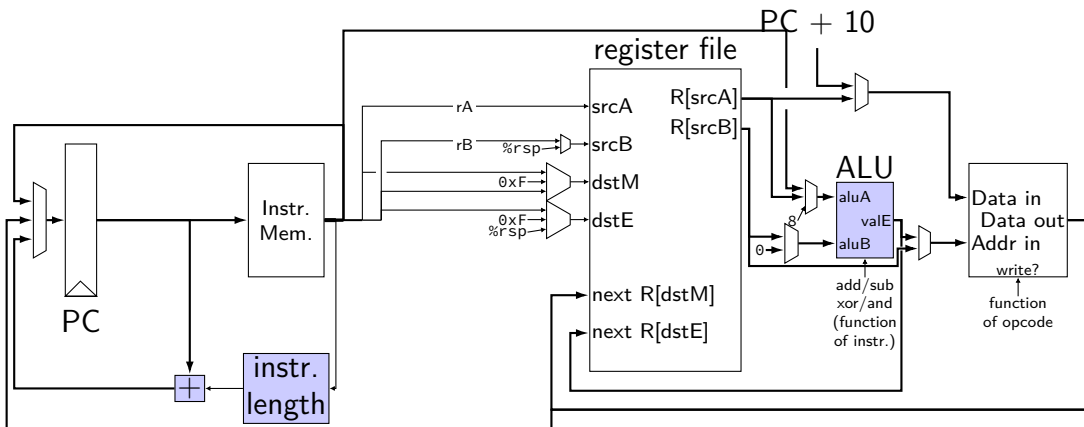
```
$ ./tools/yis nopjmp.yo
```

```
Stopped in 7 steps at PC = 0x1e. Status 'HLT', CC Z=1 S=0 O=0
```

```
Changes to registers:
```

```
Changes to memory:
```

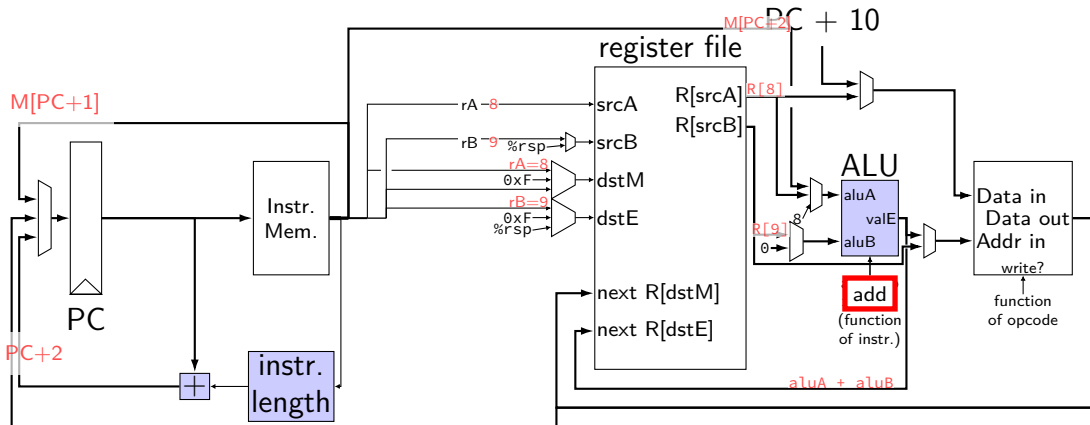
# circuit: setting MUXEs



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

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

# circuit: setting MUXEs

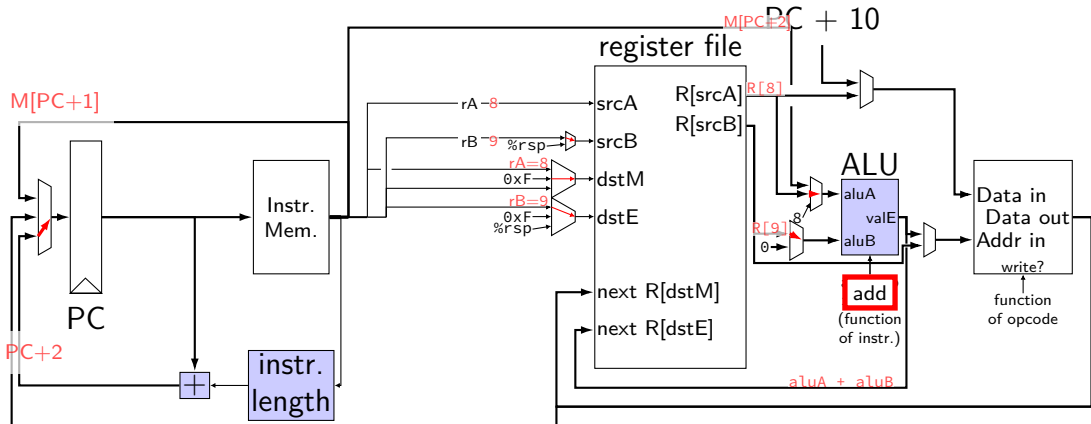


MUXEs — PC, dstM, dstE, aluA, aluB, dmemIn

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



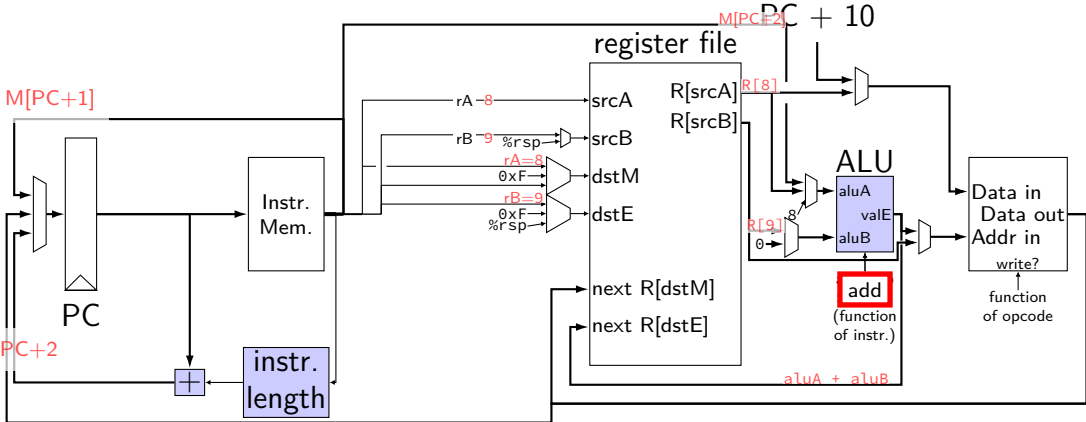
# circuit: setting MUXEs



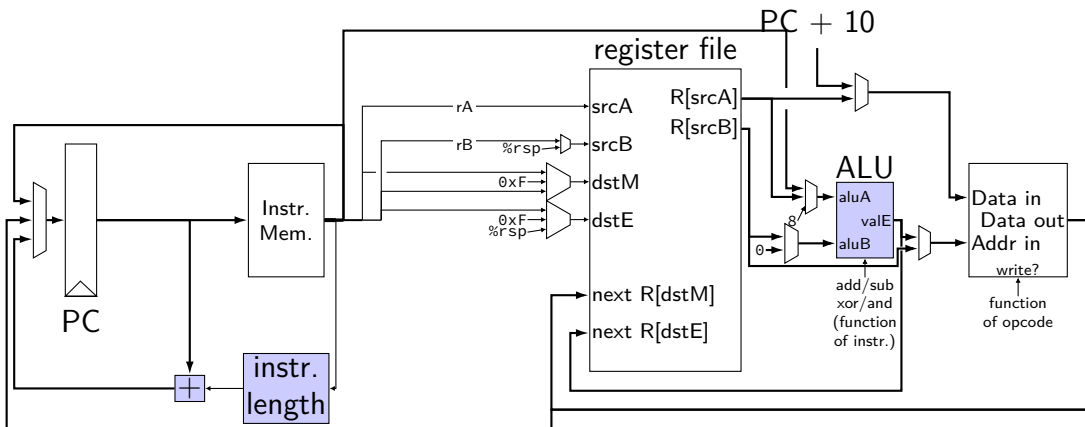
MUXes — PC, dstM, dstE, aluA, aluB, dmemIn

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

# circuit: setting MUXEs

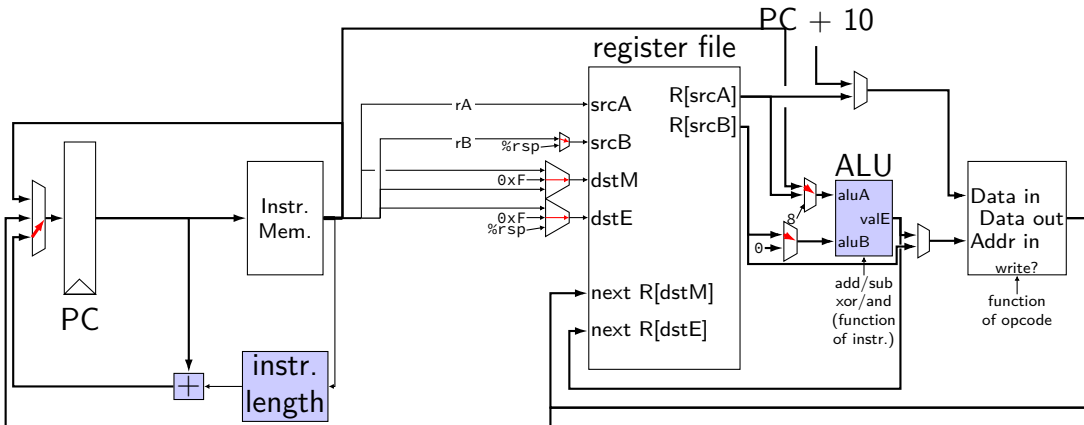


# circuit: setting MUXEs



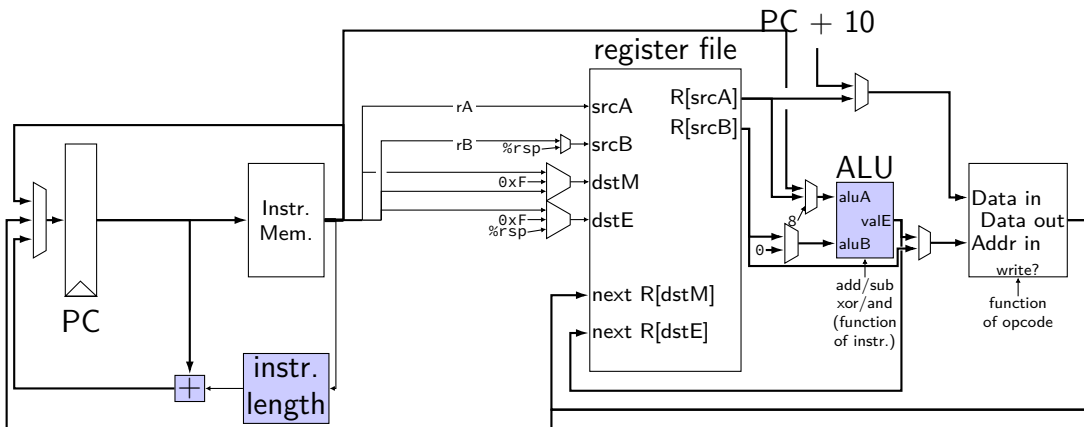
MUXEs — PC, dstM, dstE, aluA, aluB, dmemIn  
Exercise: what do they select for `rmmovq`?

# 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**?