Y86 encoding / SEQ part 1

last time

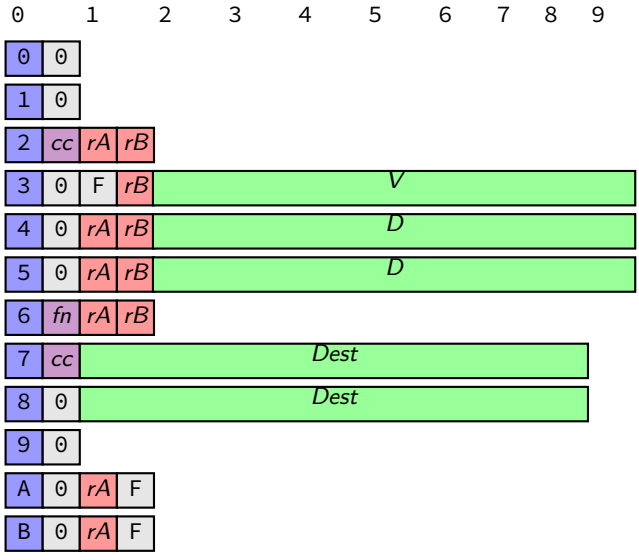
instruction set (interface) v microarchitecture (implementation)

RISC (simpler HW) v CISC (more flexible ASM)

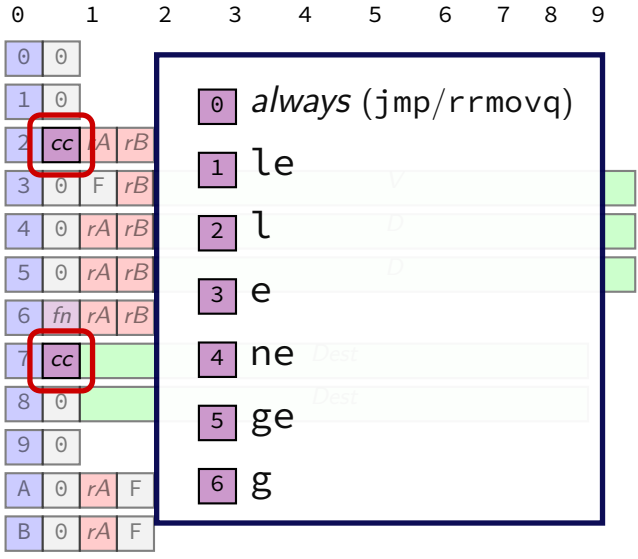
Y86-64 ISA

started Y86 encoding

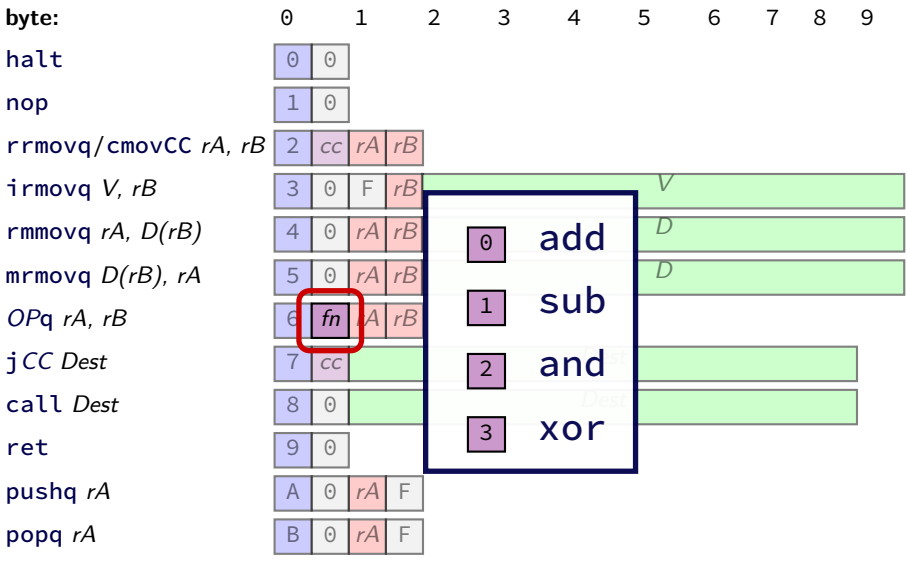
Y86-64 instruction formats



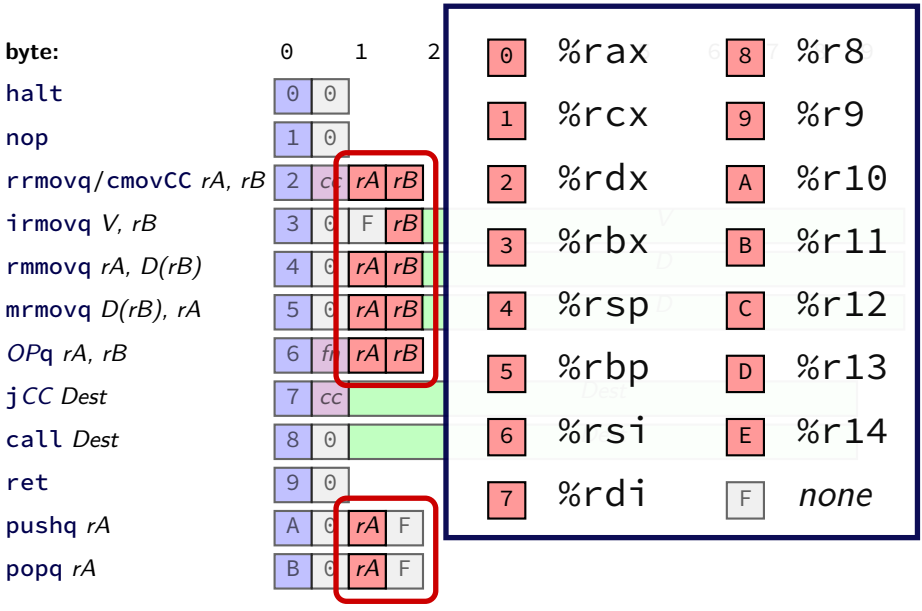
Secondary opcodes: `cmov`*cc*`/j`*cc*



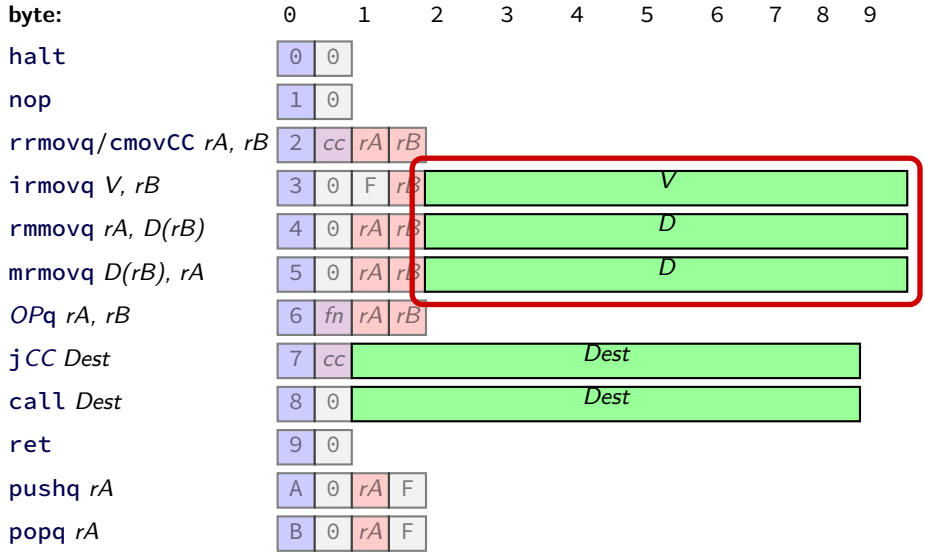
Secondary opcodes: *OP*q



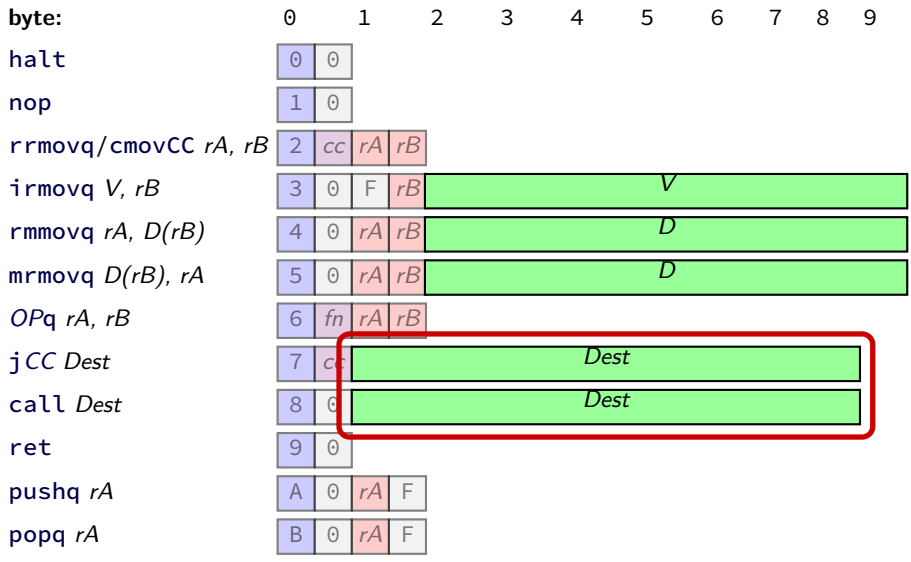
Registers: *rA, rB*



Immediates: *V, D, Dest*



Immediates: *V, D, Dest*



Y86-64 encoding (1)

```
long addOne(long x) {
    return x + 1;
}
```

x86-64:

```
    movq %rdi, %rax
    addq $1, %rax
    ret
```

Y86-64:

Y86-64 encoding (1)

```
long addOne(long x) {
    return x + 1;
}
```

x86-64:

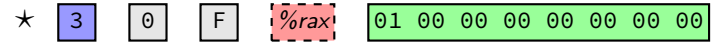
```
    movq %rdi, %rax
    addq $1, %rax
    ret
```

Y86-64:

```
    irmovq  $1,   %rax
    addq    %rdi, %rax
    ret
```

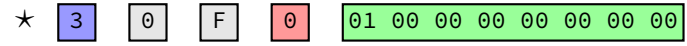
Y86-64 encoding (2)

```
addOne:
   irmovq   $1,    %rax
   addq     %rdi, %rax
   ret
```



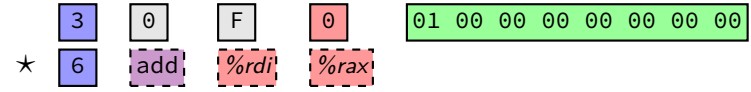
Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```



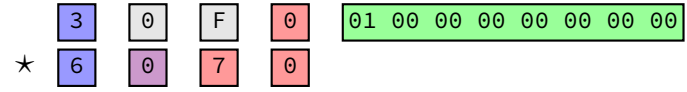
Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```



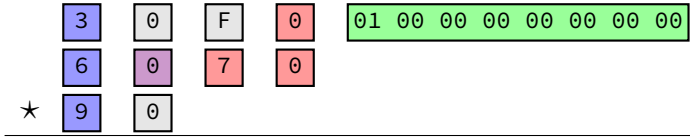
Y86-64 encoding (2)

```
addOne:
  irmovq  $1,   %rax
  addq    %rdi, %rax
  ret
```



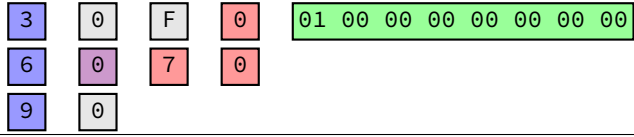
Y86-64 encoding (2)

```
addOne:
   irmovq   $1,    %rax
   addq     %rdi, %rax
   ret
```



Y86-64 encoding (2)

```
addOne:
   irmovq  $1,   %rax
   addq    %rdi, %rax
   ret
```




```
30 F0 01 00 00 00 00 00 00 00 60 70 90
```

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```

Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq     %rax, %rax
  jge doubleTillNegative
```

★ 6 add %rax %rax

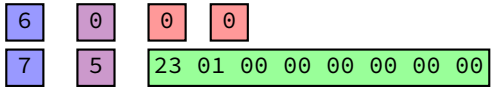
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq     %rax, %rax
  jge doubleTillNegative
```

★ 

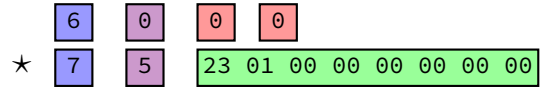
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



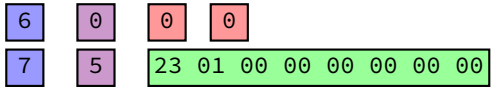
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



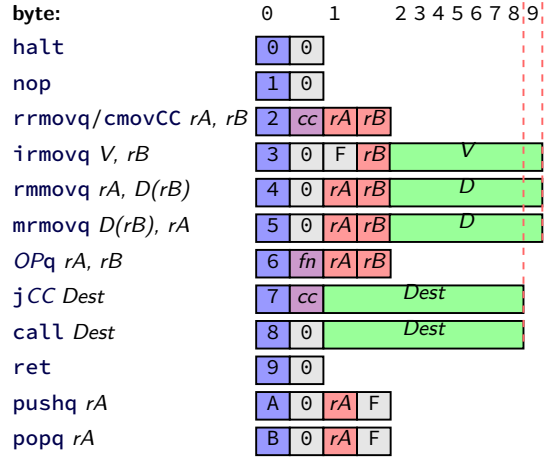
Y86-64 encoding (3)

```
doubleTillNegative:
/* suppose at address 0x123 */
  addq    %rax, %rax
  jge doubleTillNegative
```



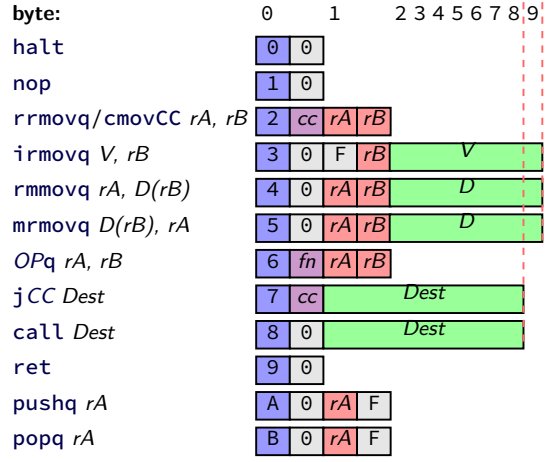
Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00 00
```



Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00 00
```

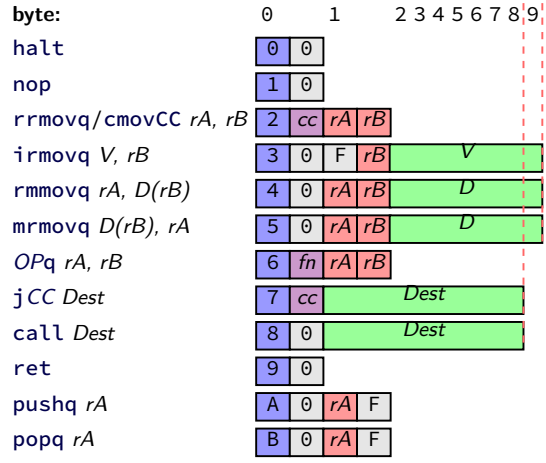


Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00 00
```

`rrmovq %rcx, %rax`

- 0 as cc: always
- 1 as reg: `%rcx`
- 0 as reg: `%rax`

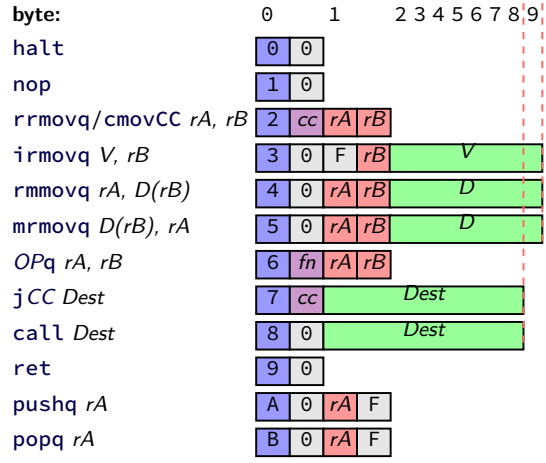


Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
```

- 0 as fn: `add`
- 1 as fn: `sub`

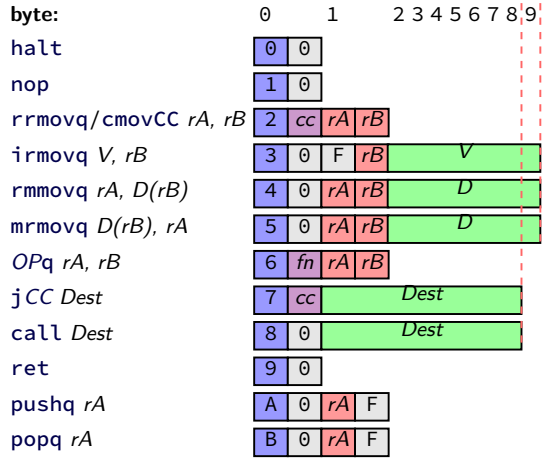


Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00
```

```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
jl     0x84
```

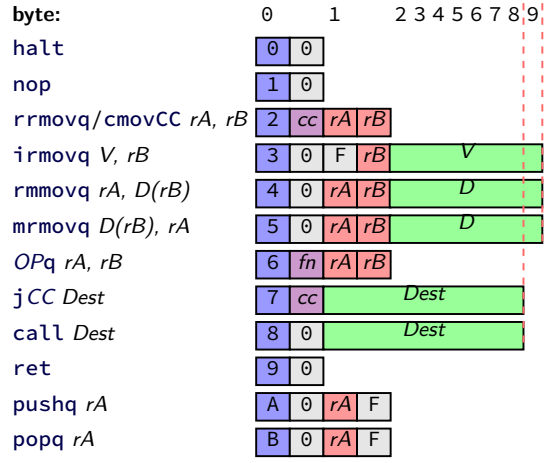
- 2 as cc: `l` (less than)
- hex 84 00… as little endian *Dest: 0x84*



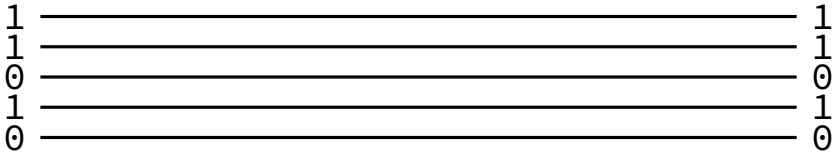
Y86-64 decoding

```
20 10 60 20 61 37 72 84 00 00 00 00 00 00 00 00
20 12 20 01 70 68 00 00 00 00 00 00 00 00
```

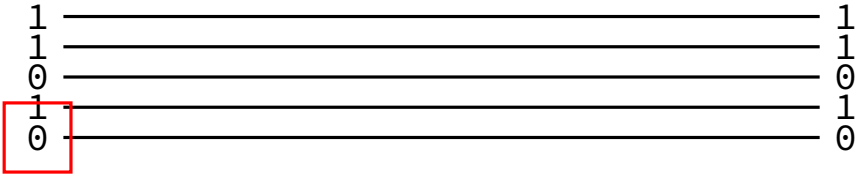
```
rrmovq %rcx, %rax
addq   %rdx, %rax
subq   %rbx, %rdi
jl     0x84
rrmovq %rcx, %rdx
rrmovq %rax, %rcx
jmp    0x68
```



circuits: wires

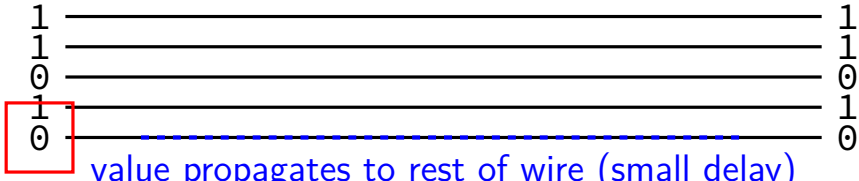


circuits: wires

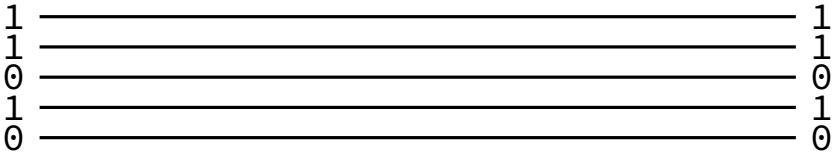


binary value — actually voltage

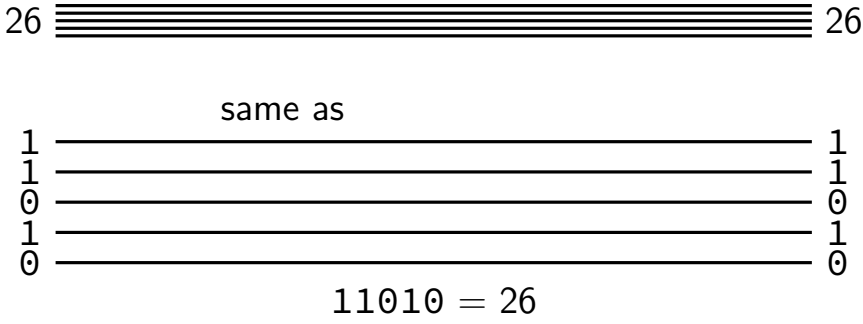
circuits: wires



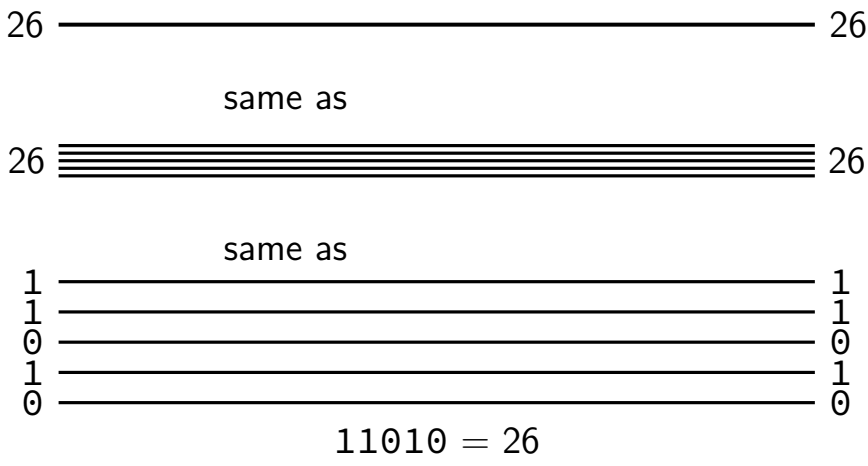
circuits: wire bundles



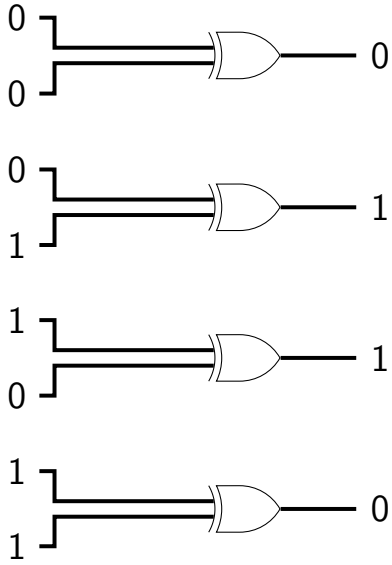
circuits: wire bundles



circuits: wire bundles



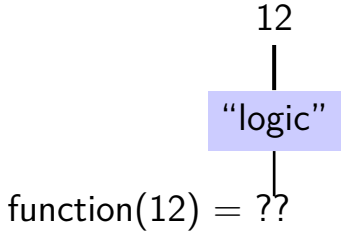
circuits: gates



circuits: logic

want to do calculations?

generalize gates:



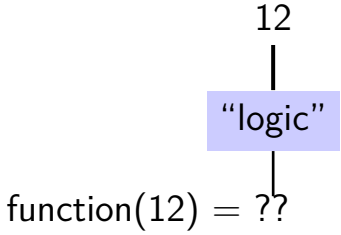
circuits: logic

want to do calculations?

generalize gates:

output wires contain result of function on input

- changes as input changes (with delay)



circuits: logic

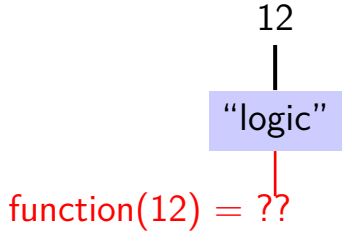
want to do calculations?

generalize gates:

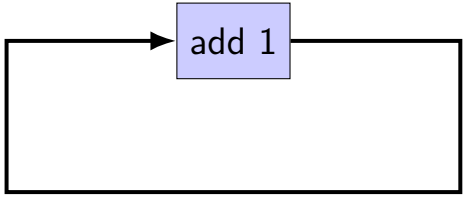
output wires contain result of function on input

- changes as input changes (with delay)

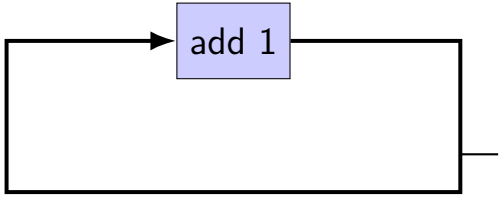
need not be same width as output



example: (broken) counter circuit

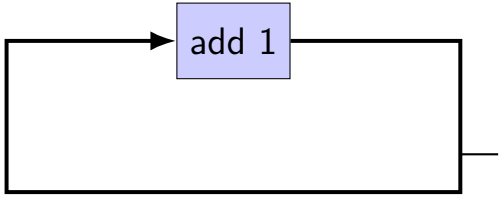


example: (broken) counter circuit



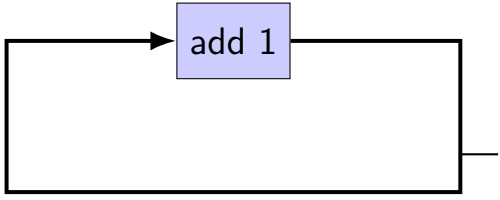
time 0: `000`

example: (broken) counter circuit

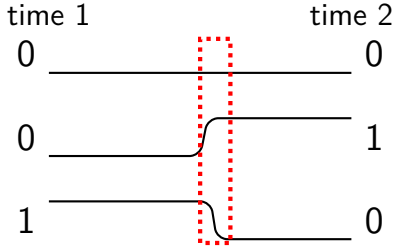


time 0: `000`
time 1: `001?`
time 2: `010?`
time 3: `011?`

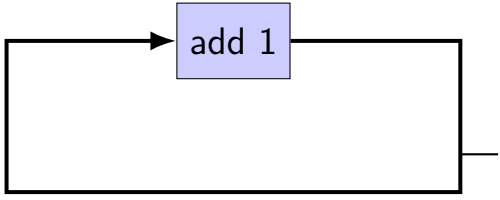
example: (broken) counter circuit



time 0: 000
time 1: 001?
time 2: 010?
time 3: 011?

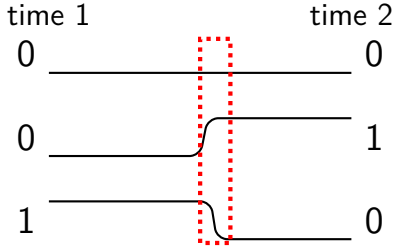


example: (broken) counter circuit



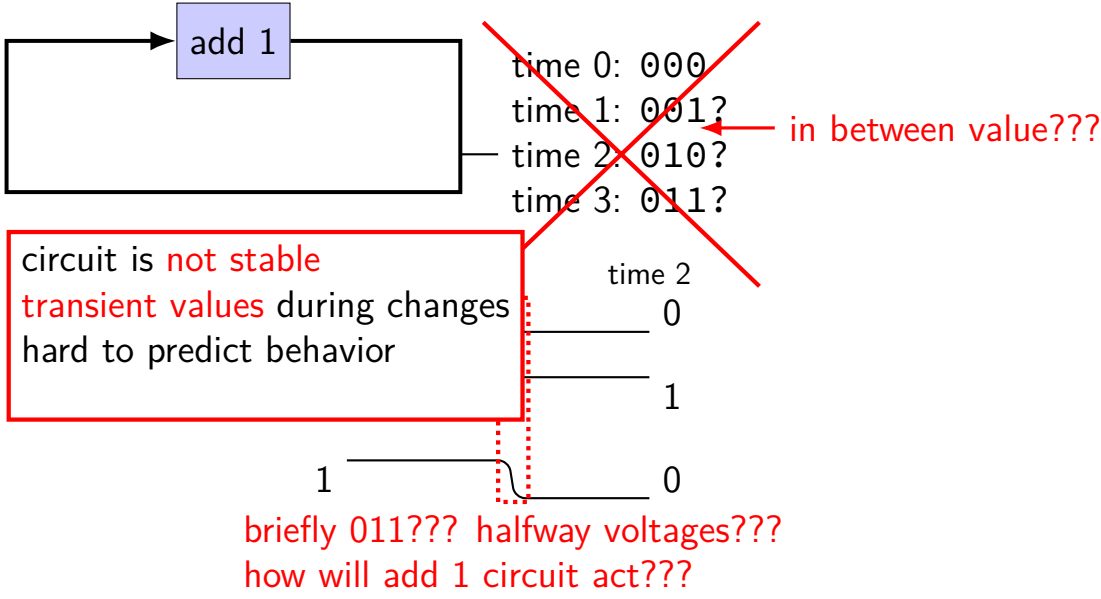
time 0: 000
time 1: 001?
time 2: 010?
time 3: 011?

in between value???



briefly 011??? halfway voltages???
how will add 1 circuit act???

example: (broken) counter circuit



circuits: state

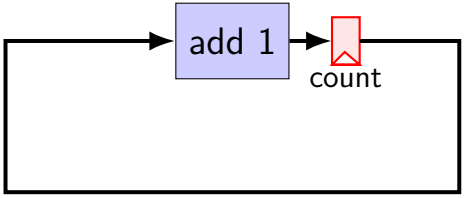
logic performs calculations all the time

never stores values!

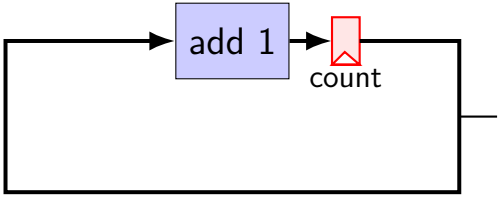
need extra elements to store values

- registers, memory

example: counter circuit (corrected)

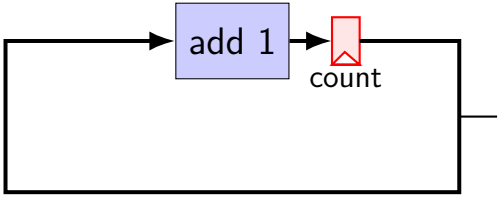


example: counter circuit (corrected)



time 0: `000`
time 1: `001`
time 2: `010`
time 3: `011`

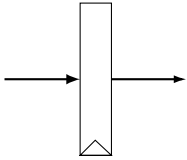
example: counter circuit (corrected)



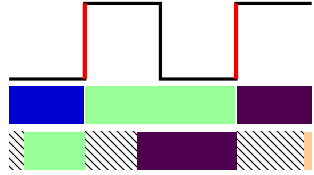
time 0: `000`
time 1: `001`
time 2: `010`
time 3: `011`

add register to store current count
updates based on "clock signal" (not shown)
avoids intermediate updates

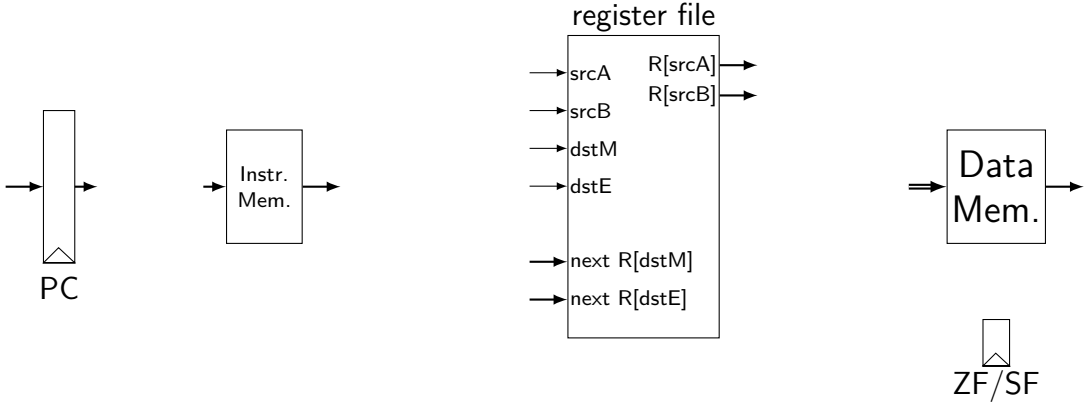
registers



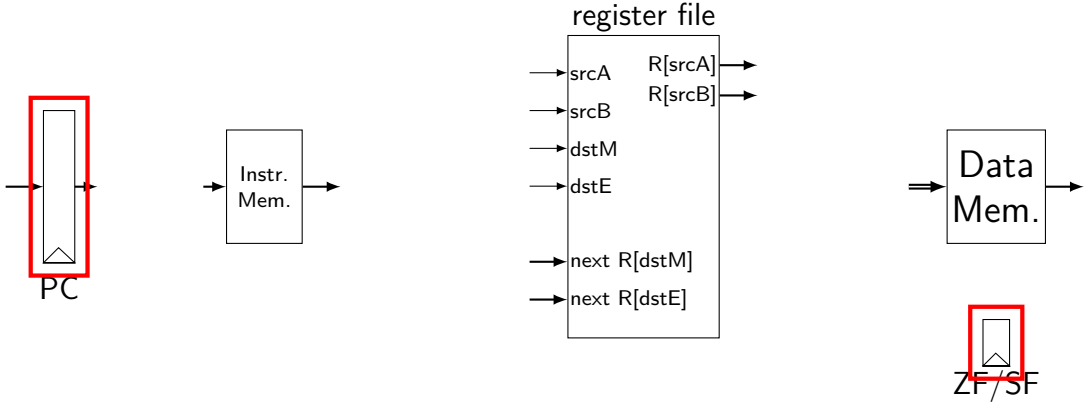
updates every clock cycle



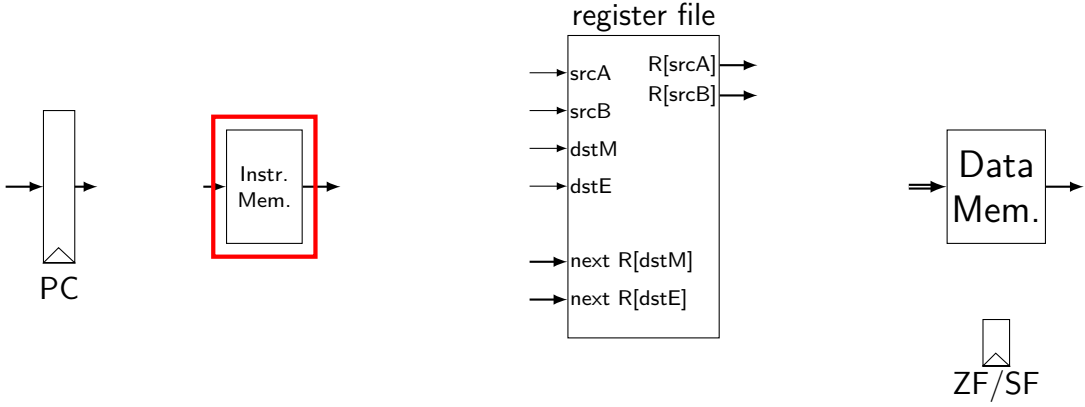
state in Y86-64



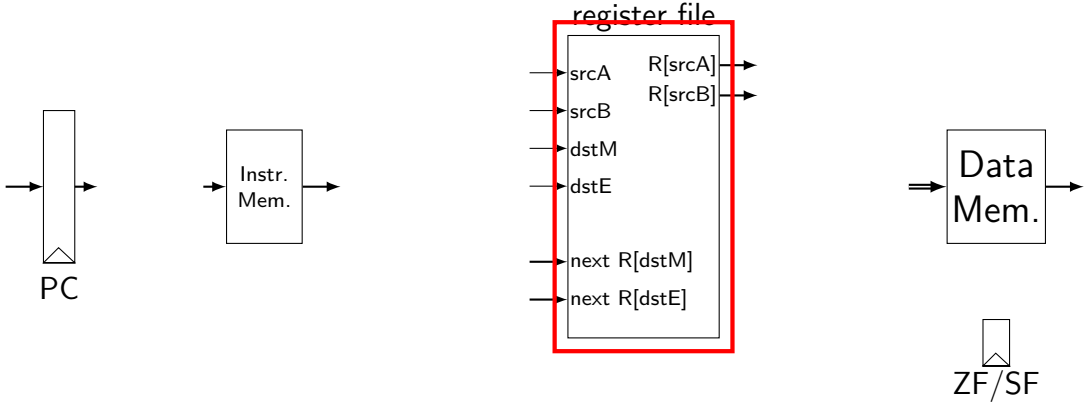
state in Y86-64



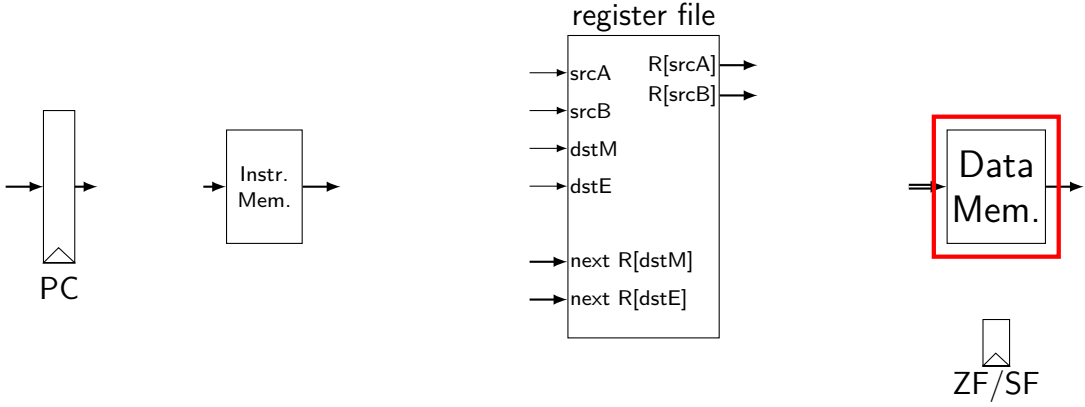
state in Y86-64



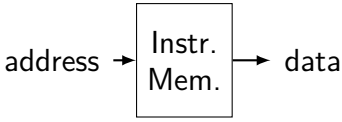
state in Y86-64

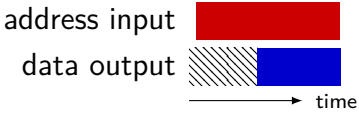


state in Y86-64

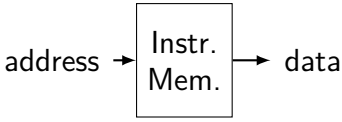


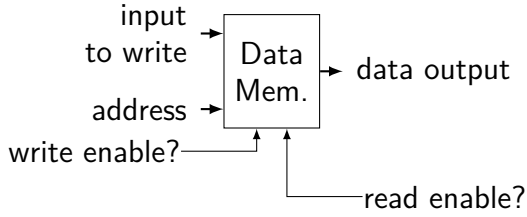
memories

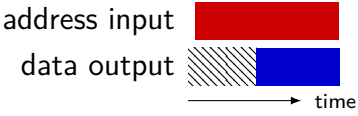




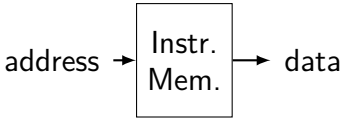
memories

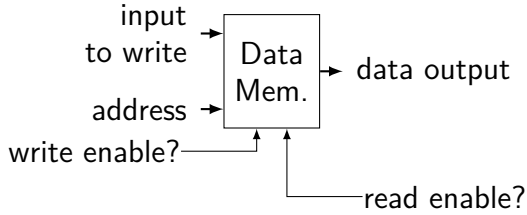

address
Instr.
Mem.
data

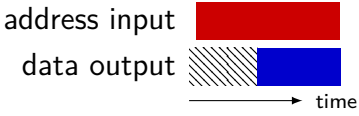

input
to write
address
write enable?
Data
Mem.
data output
read enable?

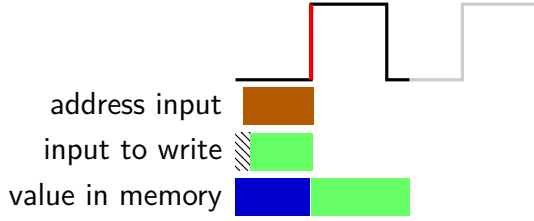

address input
data output
time

memories

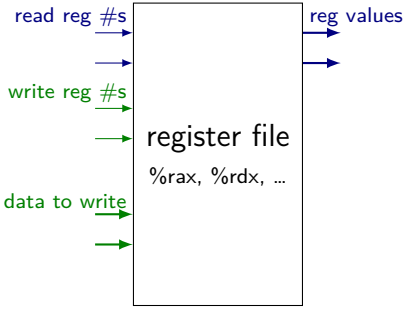

address
Instr.
Mem.
data


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

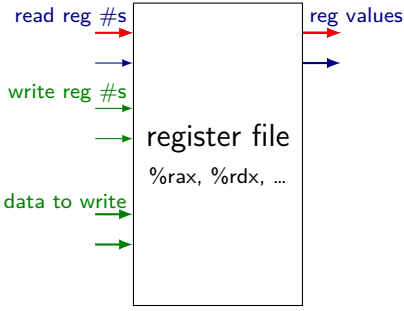

address input
data output
time

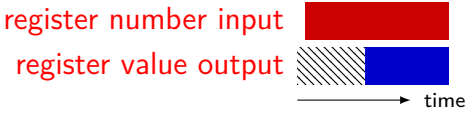

address input
input to write
value in memory

register file

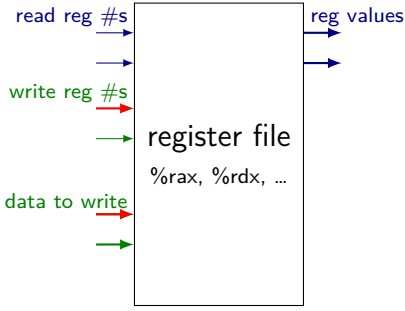


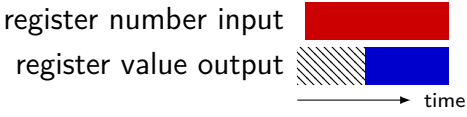
register file

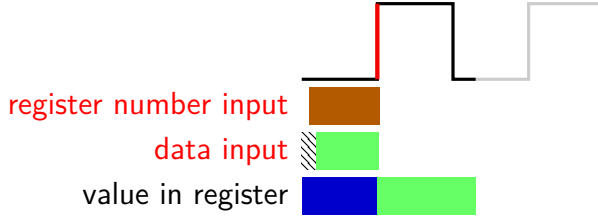




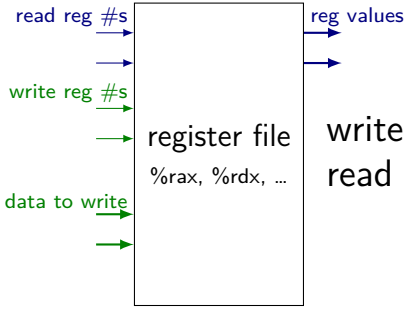
register file



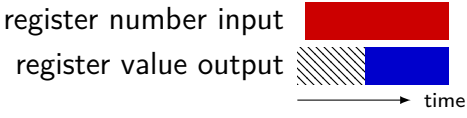


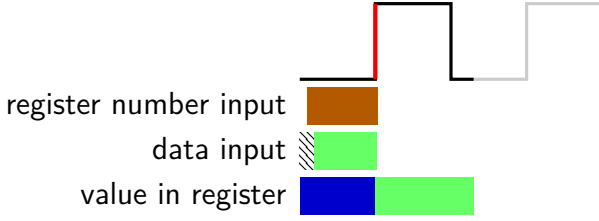


register file

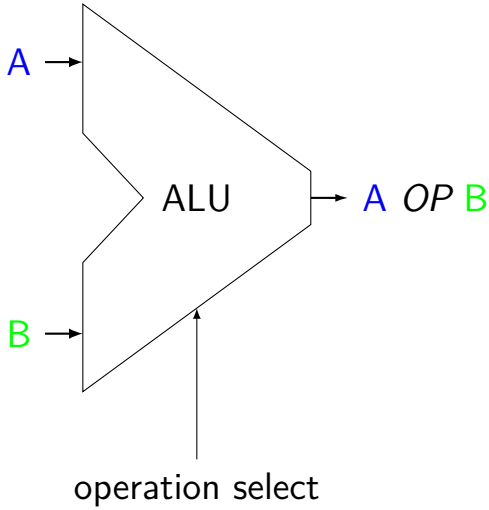


write register #15: write is ignored
read register #15: value is always 0





ALUs



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

simple ISA 1: addq

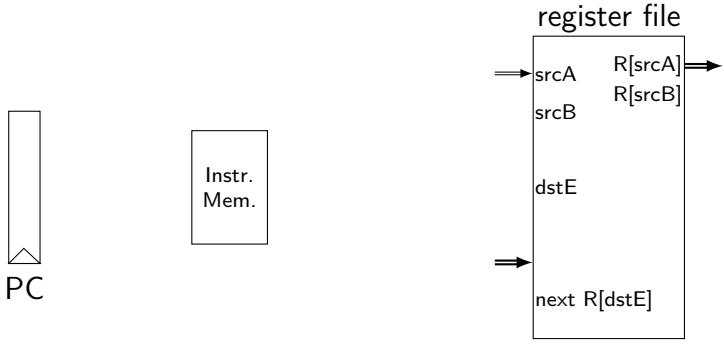
`addq %rXX, %rYY`

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

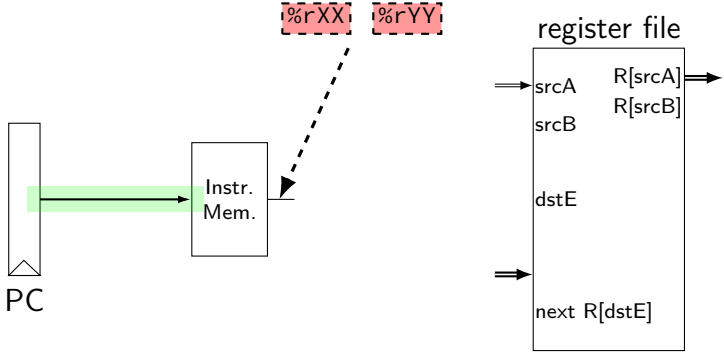
1 byte instructions, no opcode

no other instructions

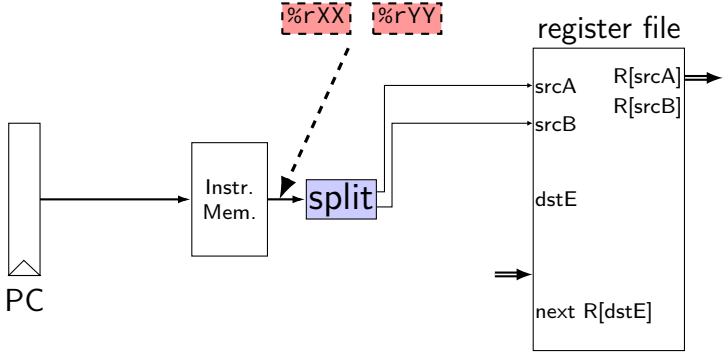
addq CPU


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

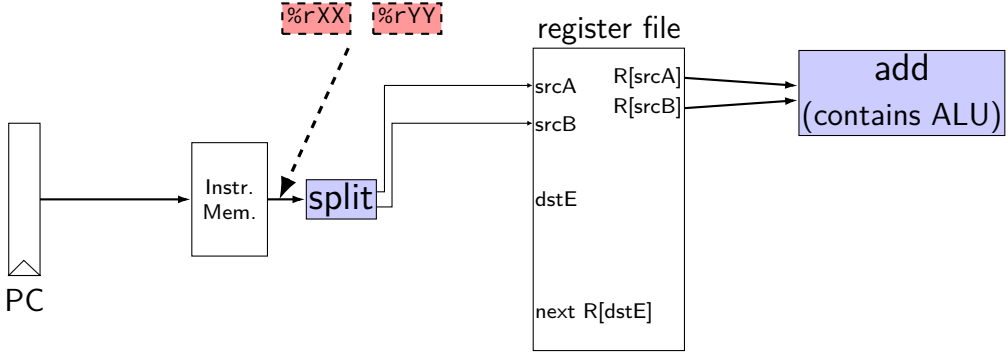
addq CPU


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

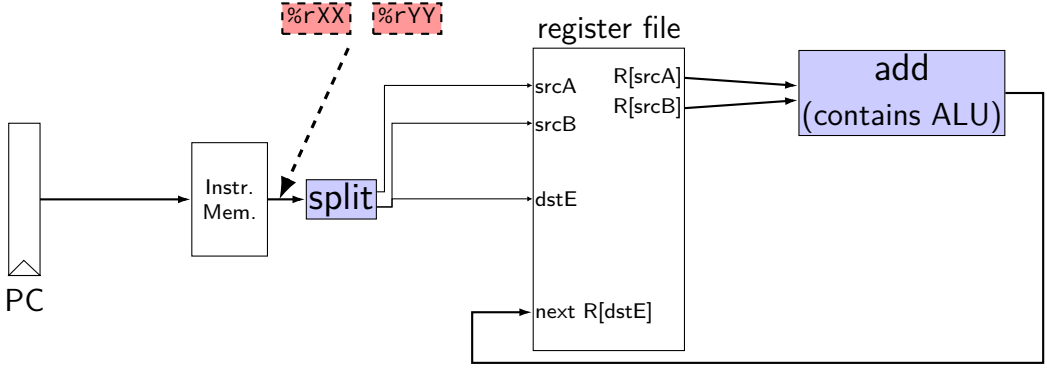
addq CPU


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

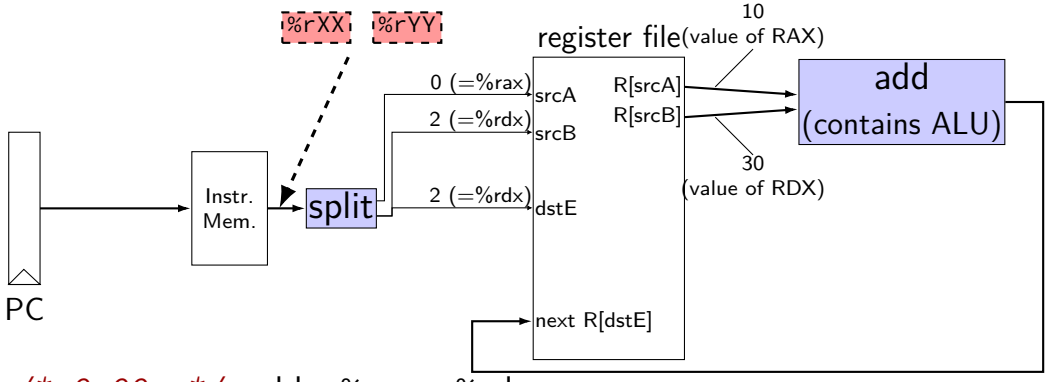
addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
add
(contains ALU)
Instr.
Mem.
split
PC

addq CPU


%rXX
%rYY
register file
srcA
srcB
dstE
next R[dstE]
R[srcA]
R[srcB]
add
(contains ALU)
Instr.
Mem.
split
PC

addq CPU

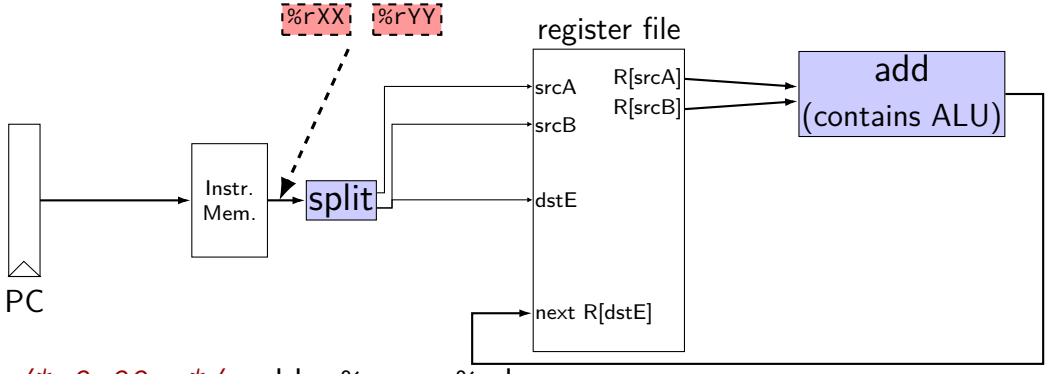



```
/* 0x00: */ addq %rax, %rdx
/* 0x01: */ addq %rbx, %rdx
```

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

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

addq CPU



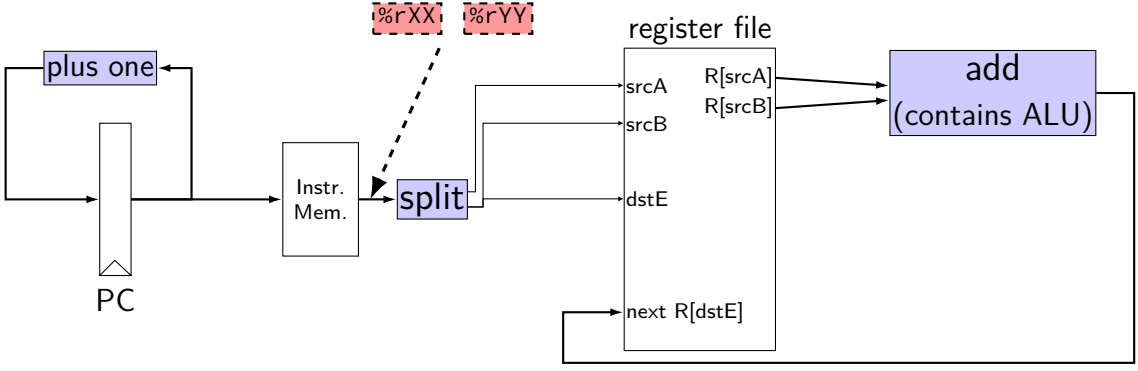

```
/* 0x00: */ addq %rax, %rdx
/* 0x01: */ addq %rbx, %rdx
```

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

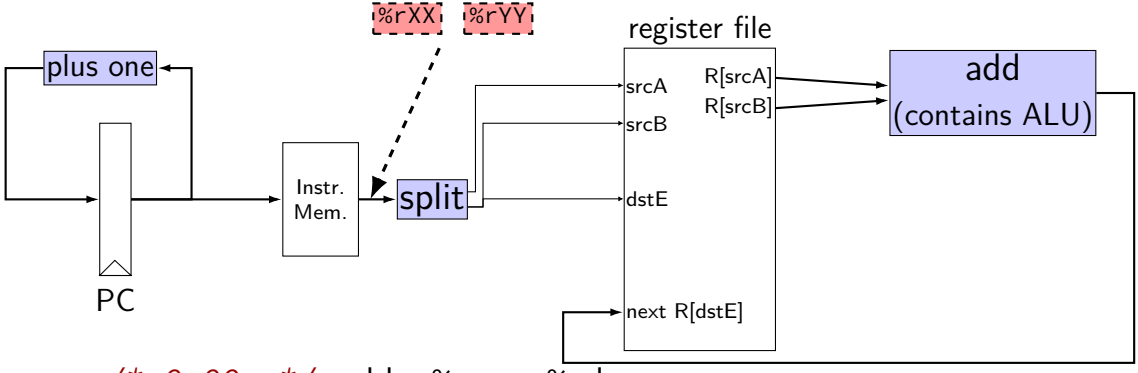
after cycle 1: PC = ????, rax = 10, rbx = 20, rdx = 40

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

addq CPU


%rXX
%rYY
register file
plus one
srcA
R[srcA]
R[srcB]
srcB
add
(contains ALU)
Instr.
Mem.
split
dstE
PC
next R[dstE]

addq CPU




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

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

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

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

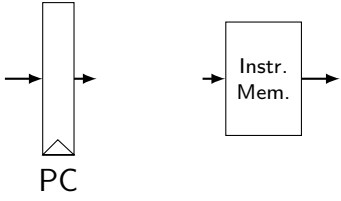
Simple ISA 2: jmp

`jmp label`

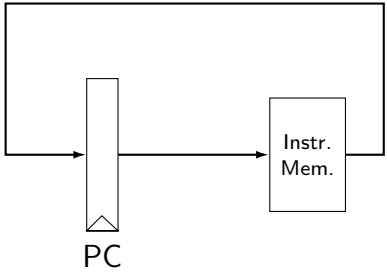
encoding: *8-byte little-endian address*

- 8 byte instructions, no opcode

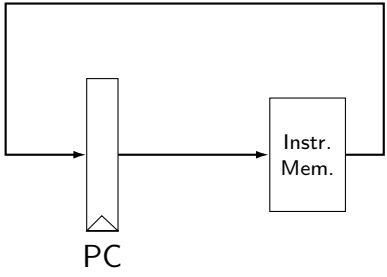
jmp CPU



jmp CPU



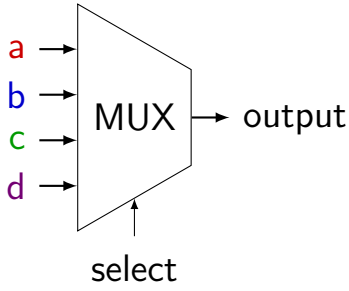
jmp CPU



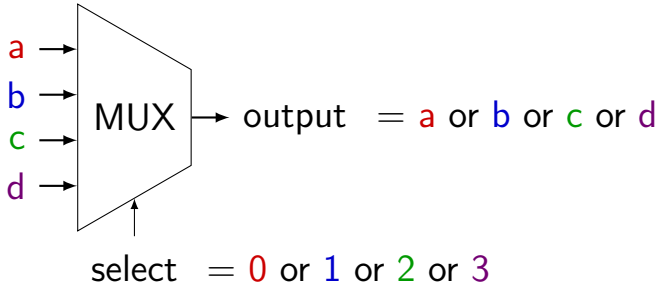

```
/* 0x00: */ jmp 0x10
/* 0x08: */ jmp 0x00
/* 0x10: */ jmp 0x08
```

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

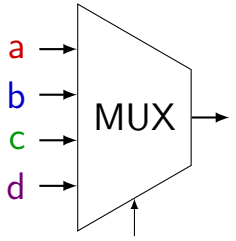
multiplexers



multiplexers



multiplexers



truth table:

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

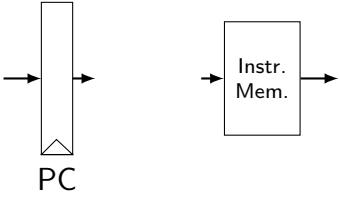
Simple ISA 3: Jmp or No-Op

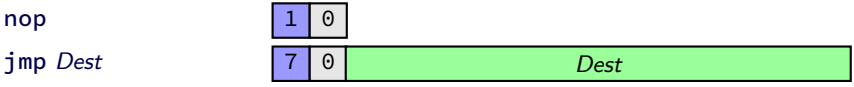
actual subset of Y86-64

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

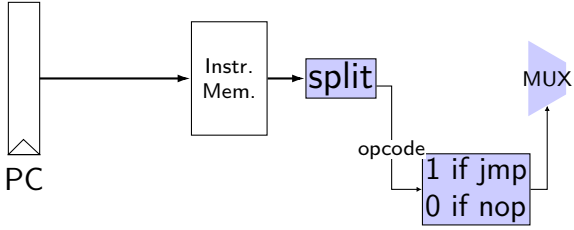
`nop` — encoded as `0x10`

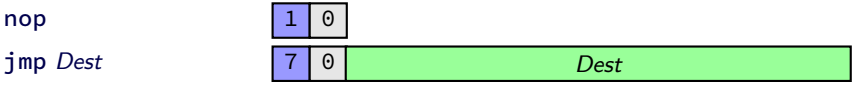
jmp+nop CPU



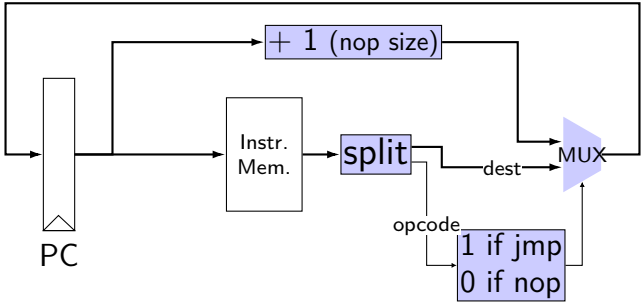


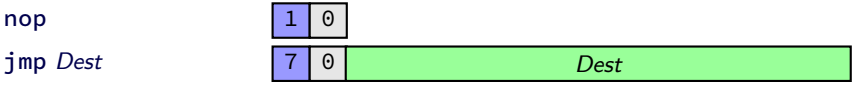
jmp+nop CPU


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

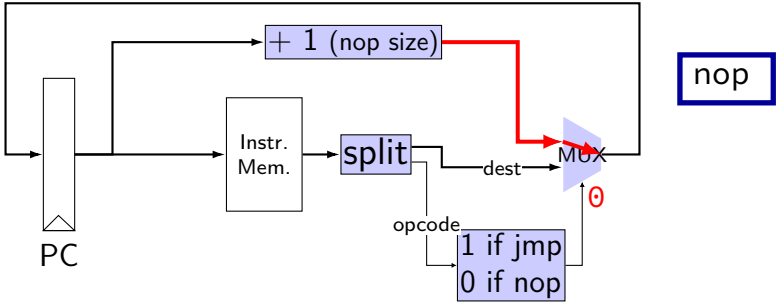

nop
jmp Dest
1
0
7
0
Dest

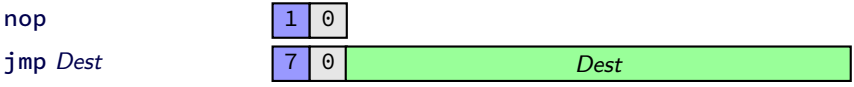
jmp+nop CPU


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

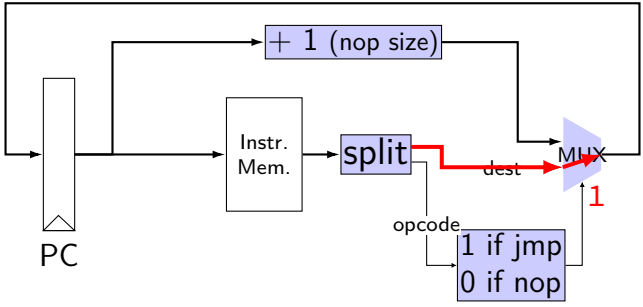

nop
jmp Dest
1
0
7
0
Dest

jmp+nop CPU


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


nop
jmp Dest
1
0
7
0
Dest

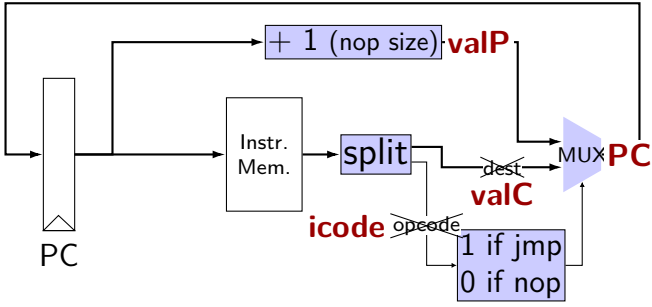
jmp+nop CPU

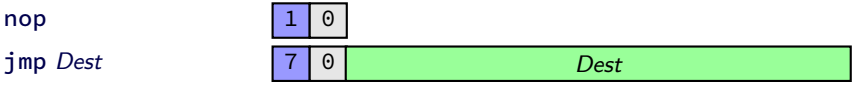



```
jmp dest
```

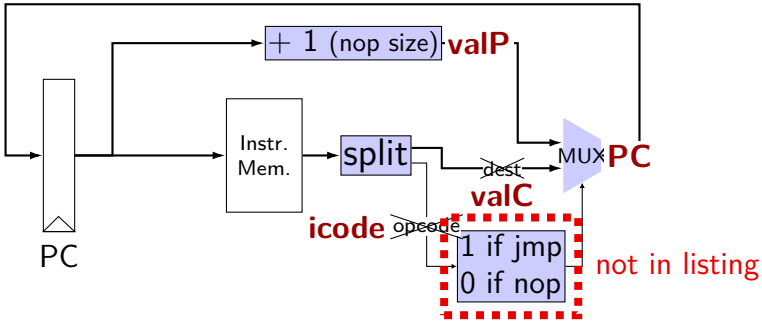
`nop`	1	0	
`jmp` *Dest*	7	0	*Dest*

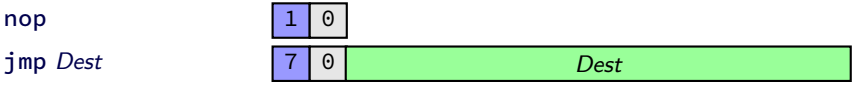
jmp+nop CPU


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


nop
jmp Dest
1
0
7
0
Dest

jmp+nop CPU


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


nop
jmp Dest
1
0
7
0
Dest

exercise: nop/add CPU

Let's say we wanted to make nop+add CPU. Where would need MUXes?

A. before one or both of the register file 'register number to read' inputs

B. before the PC register's input

C. before one of the register file 'register number to write' inputs

D. before one of the register file 'register value to write' inputs

E. before the instruction memory's address input