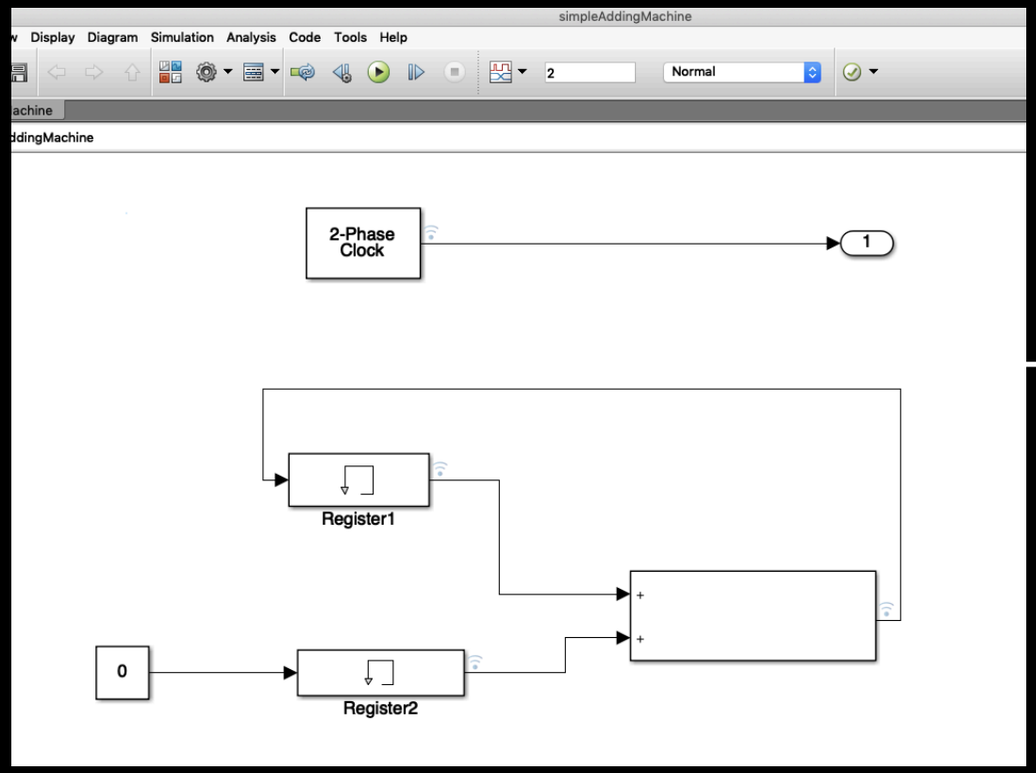
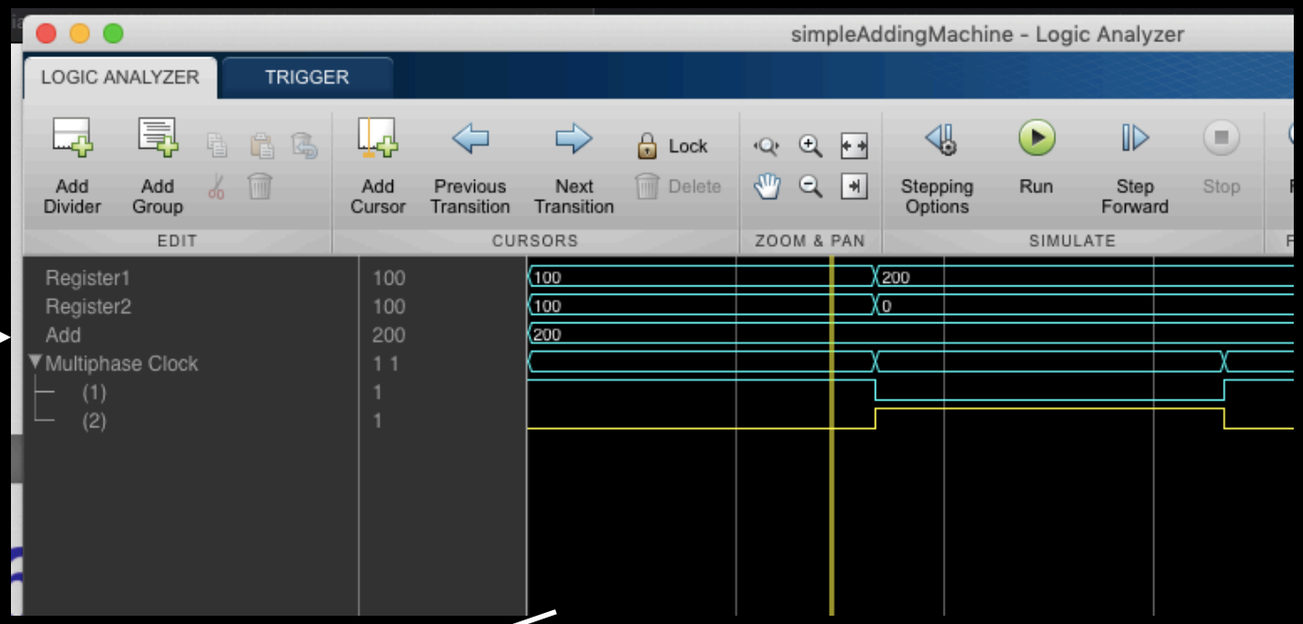
DANIEL GRAHAM

Y86 ARCHITECTURE

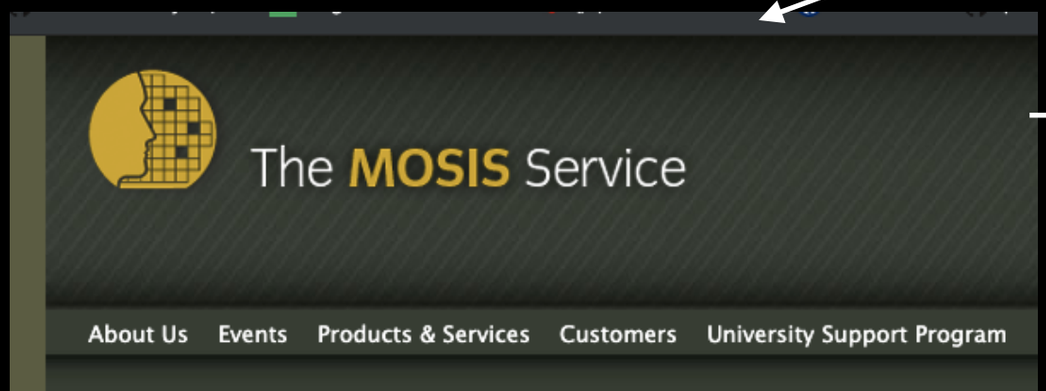
SINGLE CYCLE PROCESSOR

VHDL TO CHIP (OVERVIEW)


simpleAddingMachine
Display Diagram Simulation Analysis Code Tools Help
2
Normal
2-Phase Clock
1
Register1
0
Register2

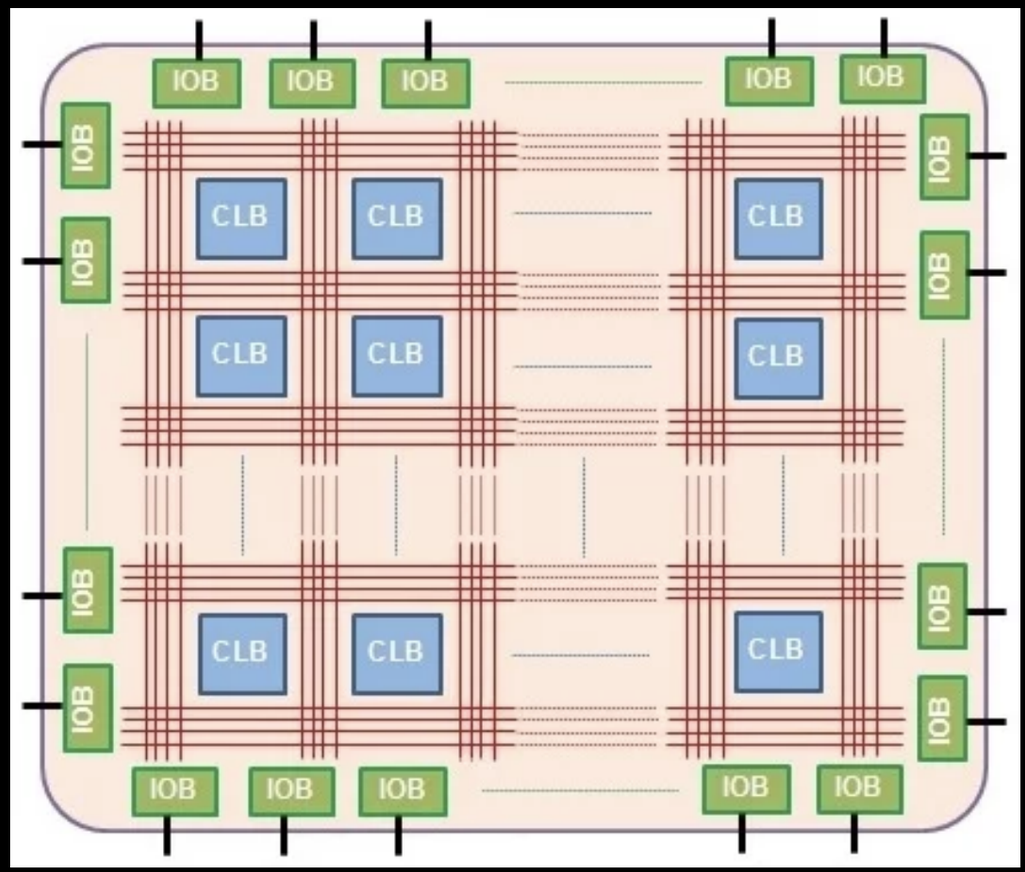

simpleAddingMachine - Logic Analyzer
LOGIC ANALYZER
TRIGGER
Add Divider
Add Group
Add Cursor
Previous Transition
Next Transition
Lock
Delete
Stepping Options
Run
Step Forward
Stop
EDIT
CURSORS
ZOOM & PAN
SIMULATE
Register1 100
Register2 100
Add 200
Multiphase Clock 1 1
(1) 1
(2) 1
100
200
100
0
200

New Variable
Analyze Code
simpleAddingMachine *
lysis
Code
Tools
Help
HDL Code
Data Objects
HDL Workflow Advisor...
Options...
Generate HDL
Generate Test Bench
Remove HDL Coder Configuration from Model


The MOSIS Service
About Us
Events
Products & Services
Customers
University Support Program


IBM
PowerPC 603™
©IBM 1993
PPC603evFB200r
19647004MM


ALTERA
Stratix IV
EP4SGX230KF40C3ES
H AAAA80849A
KOREA

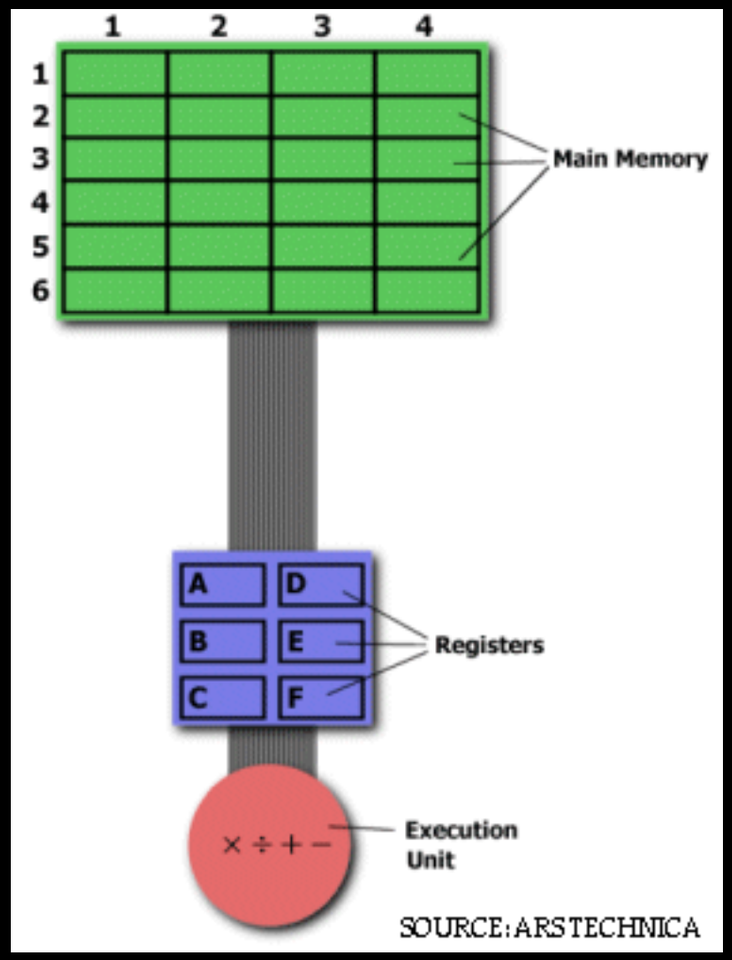

IOB
CLB

WHAT INSTRUCTIONS WILL THE MACHINE SUPPORT

WHY MAKE THINGS COMPLEX?

- The CISC Approach
  - The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible.
- Less Lines faster code, RIGHT?

LET'S LOOK AT AN EXAMPLE



- CISC APPROACH

location in memory

```
MULT 2:3, 5:2
```

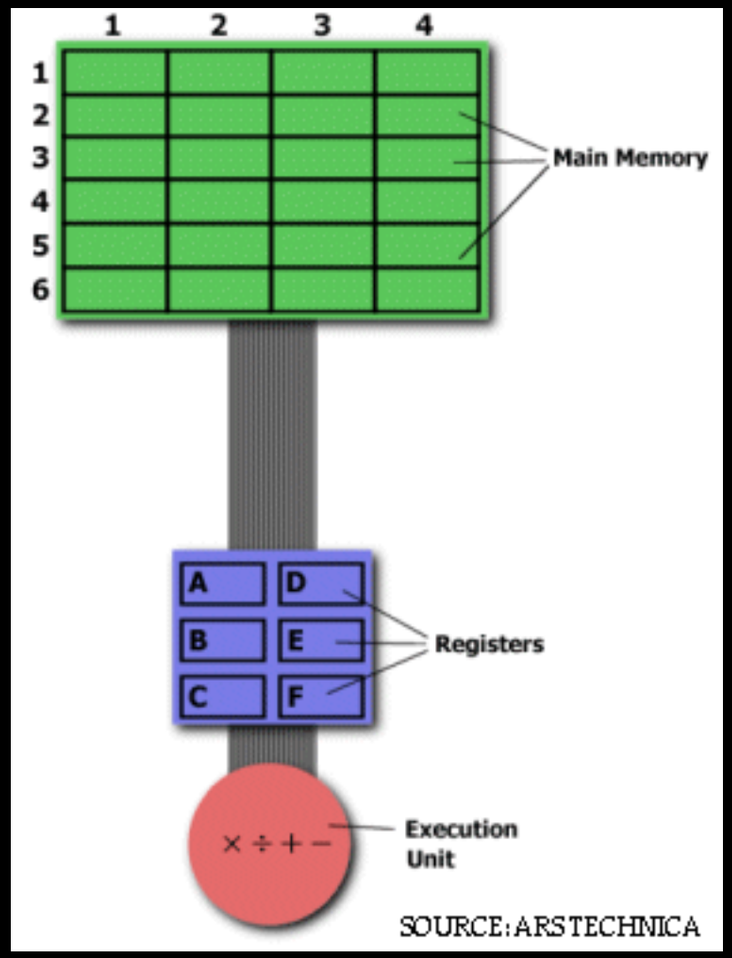
Complex instruction operates directly on memory

Easy for the complier to translate from high level statement

Requires more hardware
Requires more clock cycles to Complete 4

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

LET'S LOOK AT AN EXAMPLE



- RISC APPROACH

```
LOAD  2:3, A
LOAD  5:2, B
PROD  B, A
STORE A, 2:3
```

Use instructions that can be executed in one clock cycle

More difficult for the compiler to generate
a = a*b

Simpler hardware less transistor
More space for registers
Also shorter clock cycles.

Takes 4 cycles

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

IT'S ALL ABOUT TRADEOFFS

- RISC APPROACH

```
LOAD  2:3, A
LOAD  5:2, B
PROD  B, A
STORE A, 2:3
```

- CISC APPROACH

```
MULT 2:3, 5:2
```

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \text{ x } \frac{\text{cycles}}{\text{instruction}} \text{ x } \frac{\text{instructions}}{\text{program}}$$

Today, the Intel x86 is arguable the only chip which retains CISC architecture.


VAX 11/780

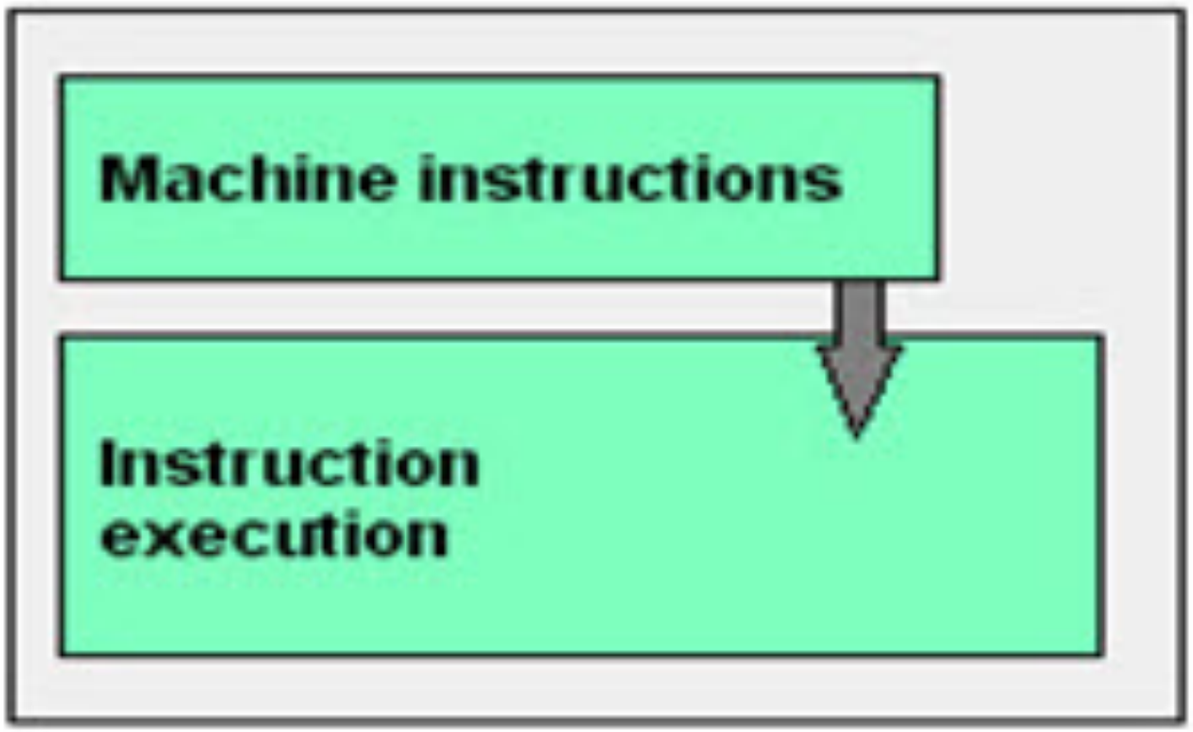
some VAX instructions

MATCHC *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

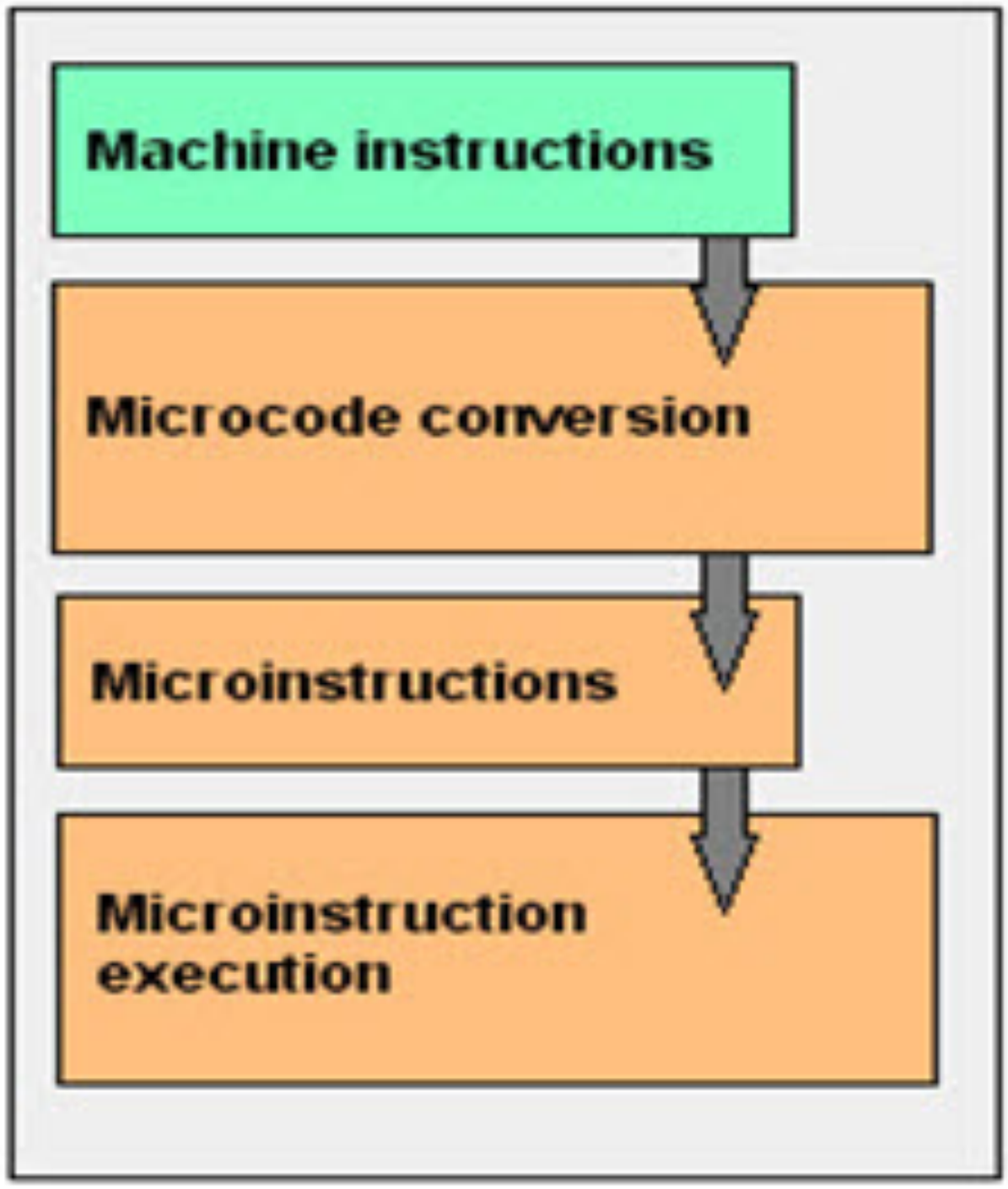
POLY *x*, *coefficientsLen*, *coefficientsPtr*
Evaluate the polynomial whose coefficients are pointed to by *coefficientPtr* at the value *x*.

EDITPC *sourceLen*, *sourcePtr*, *patternLen*, *patternPtr*
Edit the string pointed to by *sourcePtr* using the pattern string specified by *patternPtr*.

RISC


Machine instructions
Instruction execution

CISC


Machine instructions
Microcode conversion
Microinstructions
Microinstruction execution

microcode

MATCHC *haystackPtr*, *haystackLen*, *needlePtr*, *needleLen*
Find the position of the string in needle within haystack.

loop in hardware???

typically: lookup sequence of microinstructions (“microcode”)

secret simpler instruction set

is CISC the winner?

well, can't get rid of x86 features

backwards compatibility matters

more application-specific instructions

but...compilers tend to use more RISC-like subset of instructions

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*CC*	`popq`
`andq`	`cmov`*CC*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

Y86-64 instruction set

based on x86

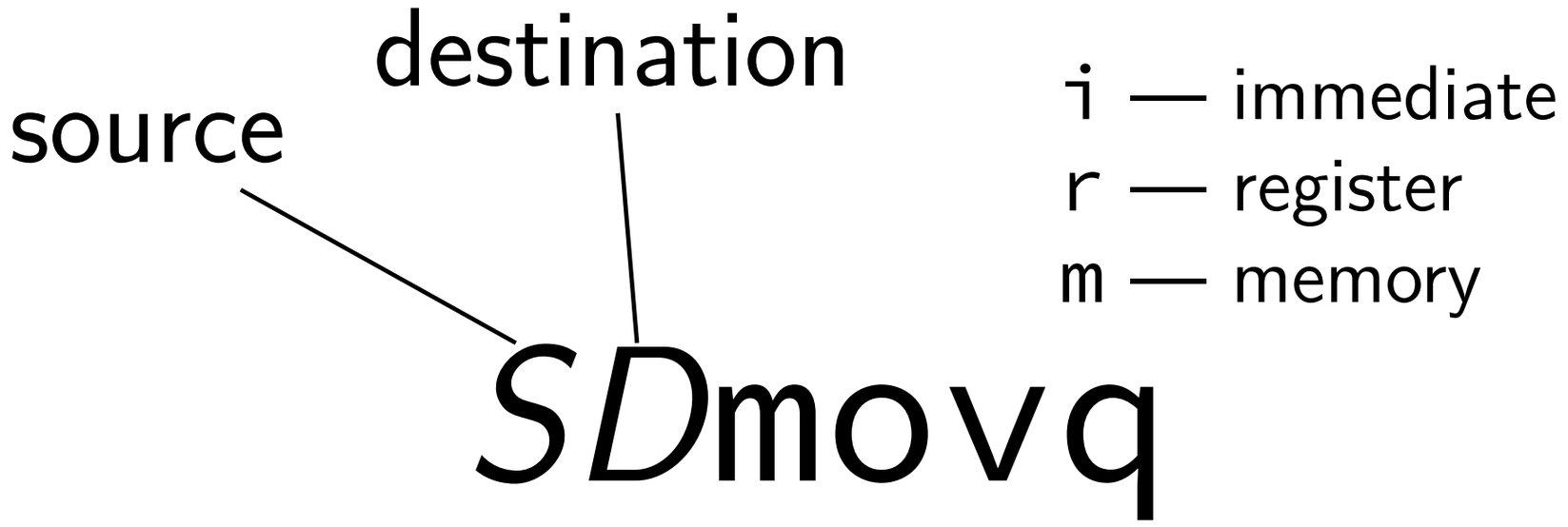
omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*`CC*`	`popq`
`andq`	`cmov`*`CC*`	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

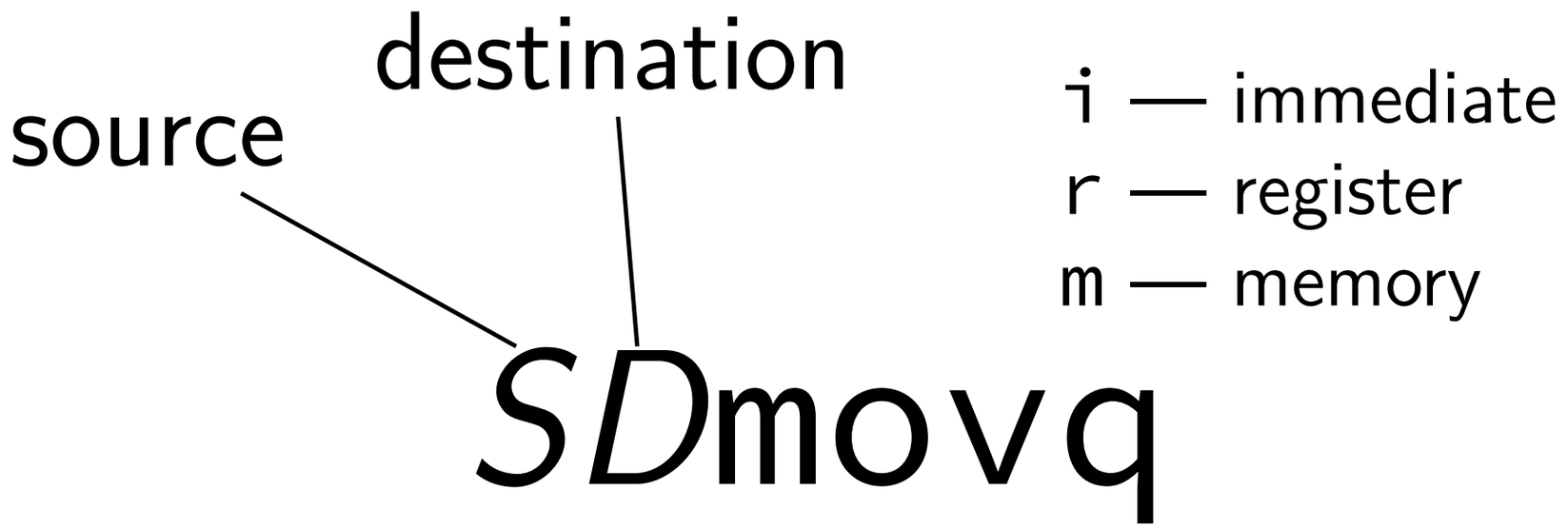
much, much simpler encoding

Y86-64: movq



i — immediate
r — register
m — memory

Y86-64: movq

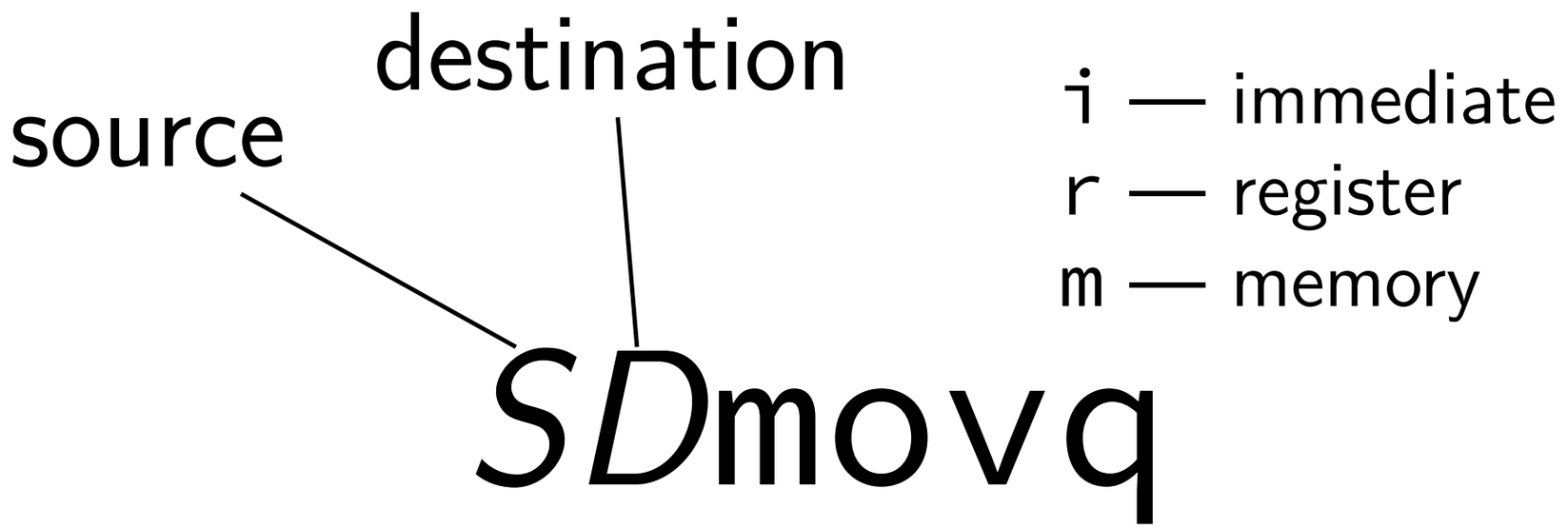


irmovq ~~immovq~~ ~~iimovq~~

rrmovq rmmovq ~~rimovq~~

mrmovq ~~mmmovq~~ ~~mimovq~~

Y86-64: movq



i — immediate
r — register
m — memory

irmovq

~~immovq~~

rrmovq

rmmovq

mrmovq

~~mmmovq~~

Y86-64 instruction set

based on x86

omits most of the 1000+ instructions

leaves

`addq`	`jmp`	`pushq`
`subq`	`j`*`CC`*	`popq`
`andq`	`cmov`*`CC`*	`movq` (renamed)
`xorq`	`call`	`hlt` (renamed)
`nop`	`ret`	

much, much simpler encoding

cmovCC

conditional move

exist on x86-64 (but you probably didn't see them)

Y86-64: register-to-register only

instead of:

```
    jle skip_move
    rrmovq %rax, %rbx
skip_move:
    // ...
```

can do:

```
    cmovg %rax, %rbx
```

halt

(x86-64 instruction called `hlt`)

Y86-64 instruction `halt`

stops the processor

real processors: reserved for OS

Y86-64: specifying addresses

Valid: `rmmovq %r11, 10(%r12)`

Y86-64: specifying addresses

Valid: `rmmovq %r11, 10(%r12)`

Invalid: ~~`rmmovq %r11, 10(%r12,%r13)`~~

Invalid: ~~`rmmovq %r11, 10(,%r12,4)`~~

Invalid: ~~`rmmovq %r11, 10(%r12,%r13,4)`~~

Y86-64: accessing memory (1)

r12 $\leftarrow$ memory[10 + r11] + r12

Invalid: ~~`addq 10(%r11), %r12`~~

Instead:

```
mrmovq 10(%r11), %r11
/* overwrites %r11 */

addq %r11, %r12
```

Y86-64: accessing memory (2)

r12 ← memory[10 + 8 * r11] + r12

Invalid: ~~`addq 10(,%r11,8), %r12`~~

Y86-64: accessing memory (2)

r12 ← memory[10 + 8 * r11] + r12

Invalid: ~~`addq 10(,%r11,8), %r12`~~

Instead:

```
/* replace %r11 with 8*%r11 */
addq %r11, %r11
addq %r11, %r11
addq %r11, %r11

mrmovq 10(%r11), %r11
addq %r11, %r12
```

Y86-64 constants (1)

```
irmovq $100, %r11
```

only instruction with non-address constant operand

Y86-64 constants (2)

r12 $\leftarrow$ r12 + 1

Invalid: ~~`addq $1, %r12`~~

Y86-64 constants (2)

r12 $\leftarrow$ r12 + 1

Invalid: ~~`addq $1, %r12`~~

Instead, need an extra register:

```
irmovq $1, %r11
addq %r11, %r12
```

Y86-64: operand uniqueness

only one kind of value for each operand

instruction name tells you the kind

(why `movq` was 'split' into four names)

Y86-64: condition codes

ZF — value was zero?

SF — sign bit was set? i.e. value was negative?

this course: no OF, CF (to simplify assignments)

set by `addq`, `subq`, `andq`, `xorq`

not set by anything else

Y86-64: using condition codes

`subq SECOND, FIRST` (value = FIRST - SECOND)

`j__` or `cmov__`	condition code bit test	value test
`le`	`SF` $= 1$ or `ZF` $= 1$	value ≤ 0
`l`	`SF` $= 1$	value < 0
`e`	`ZF` $= 1$	value $= 0$
`ne`	`ZF` $= 0$	value $\neq 0$
`ge`	`SF` $= 0$	value ≥ 0
`g`	`SF` $= 0$ and `ZF` $= 0$	value > 0

missing `OF` (overflow flag); `CF` (carry flag)

Y86-64: conditionals (1)

~~`cmp`~~, ~~`test`~~

Y86-64: conditionals (1)

~~`cmp`~~, ~~`test`~~

instead: use side effect of normal arithmetic

Y86-64: conditionals (1)

~~`cmp`~~, ~~`test`~~

instead: use side effect of normal arithmetic

instead of

```
cmpq %r11, %r12
jle somewhere
```

maybe:

```
subq %r11, %r12
jle
```

(but changes `%r12`)

push/pop

```
pushq %rbx
```

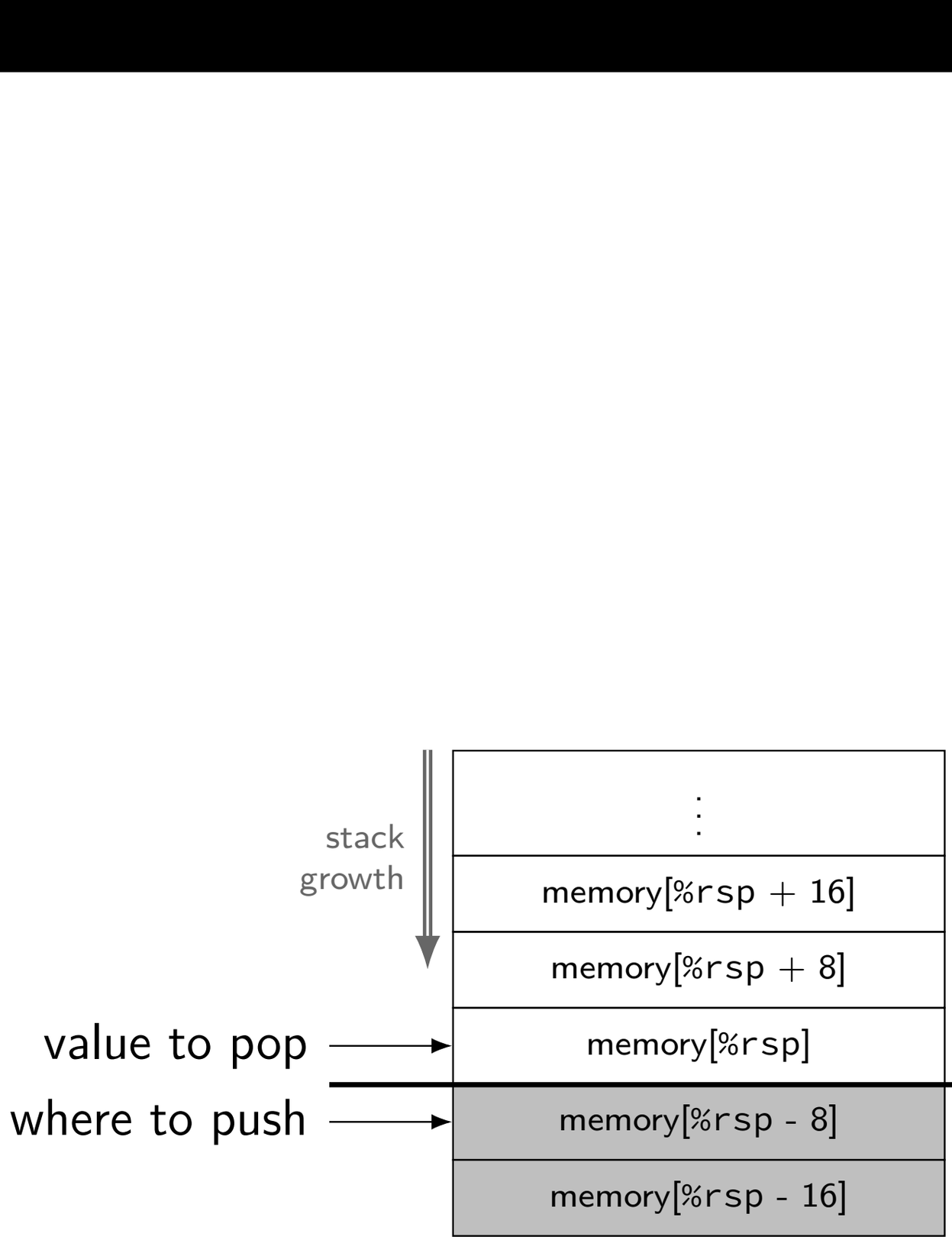
%rsp ← %rsp − 8

memory[%rsp] ← %rbx

```
popq %rbx
```

%rbx ← memory[%rsp]

%rsp ← %rsp + 8



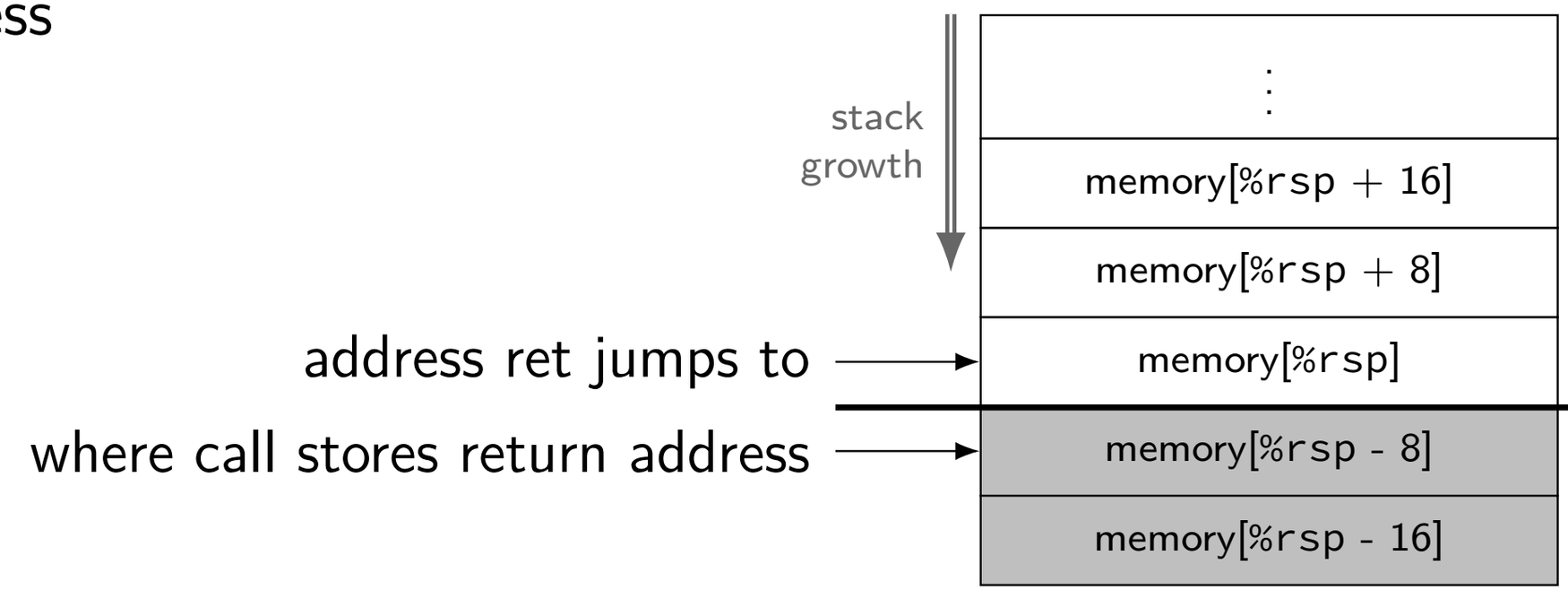
call/ret

`call` `LABEL`

push PC (next instruction address) on stack
jmp to LABEL address

`ret`

pop address from stack
jmp to that address



Y86-64 state

`%r`*XX* — 15 registers

- ~~`%r15`~~ missing
- smaller parts of registers missing

ZF (zero), SF (sign), ~~OF (overflow)~~

- book has OF, we'll not use it
- ~~CF~~ (carry) missing

`Stat` — processor status — halted?

PC — program counter (AKA instruction pointer)

main memory

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;
}
```

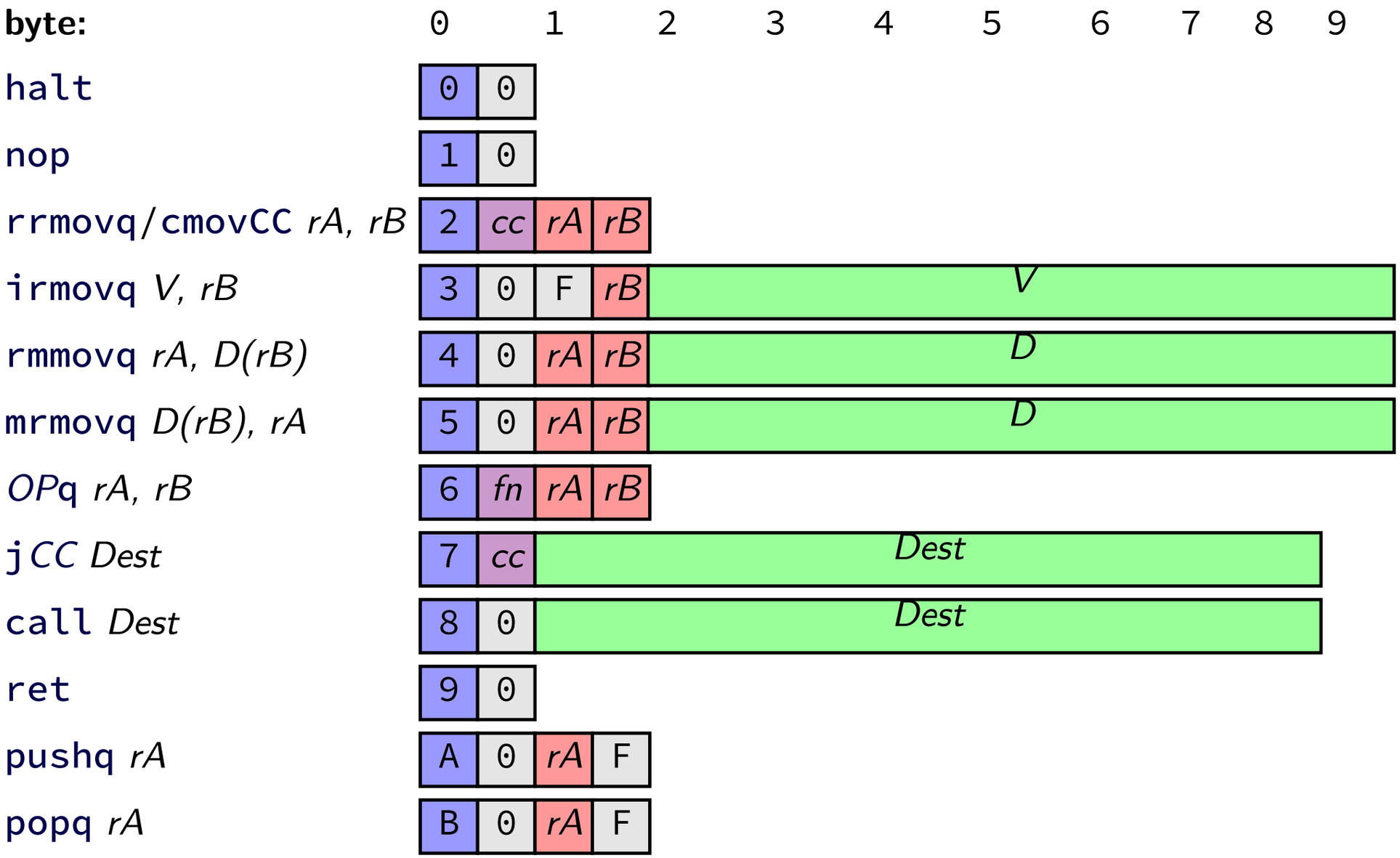
x86-64:

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

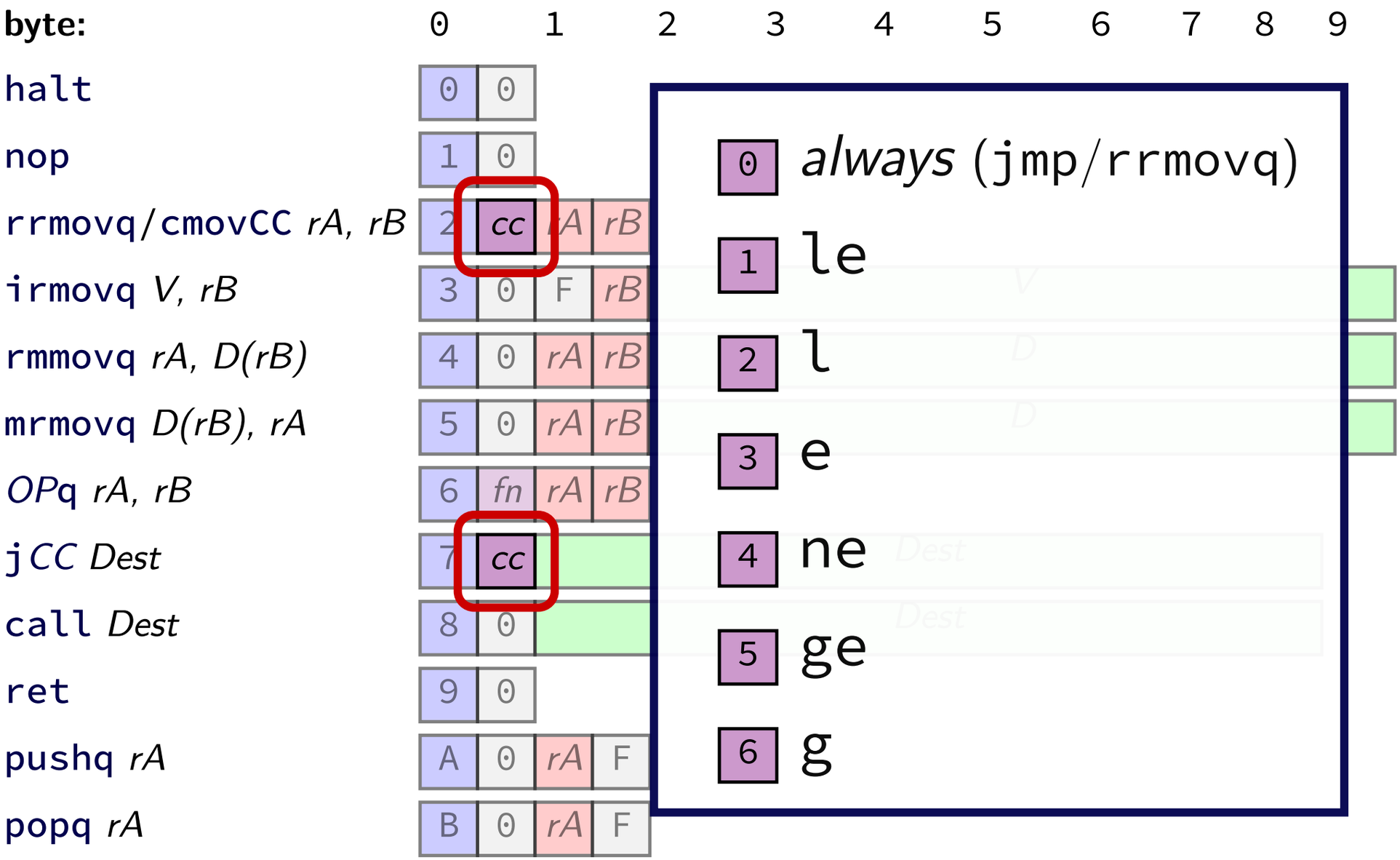
Y86-64:

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

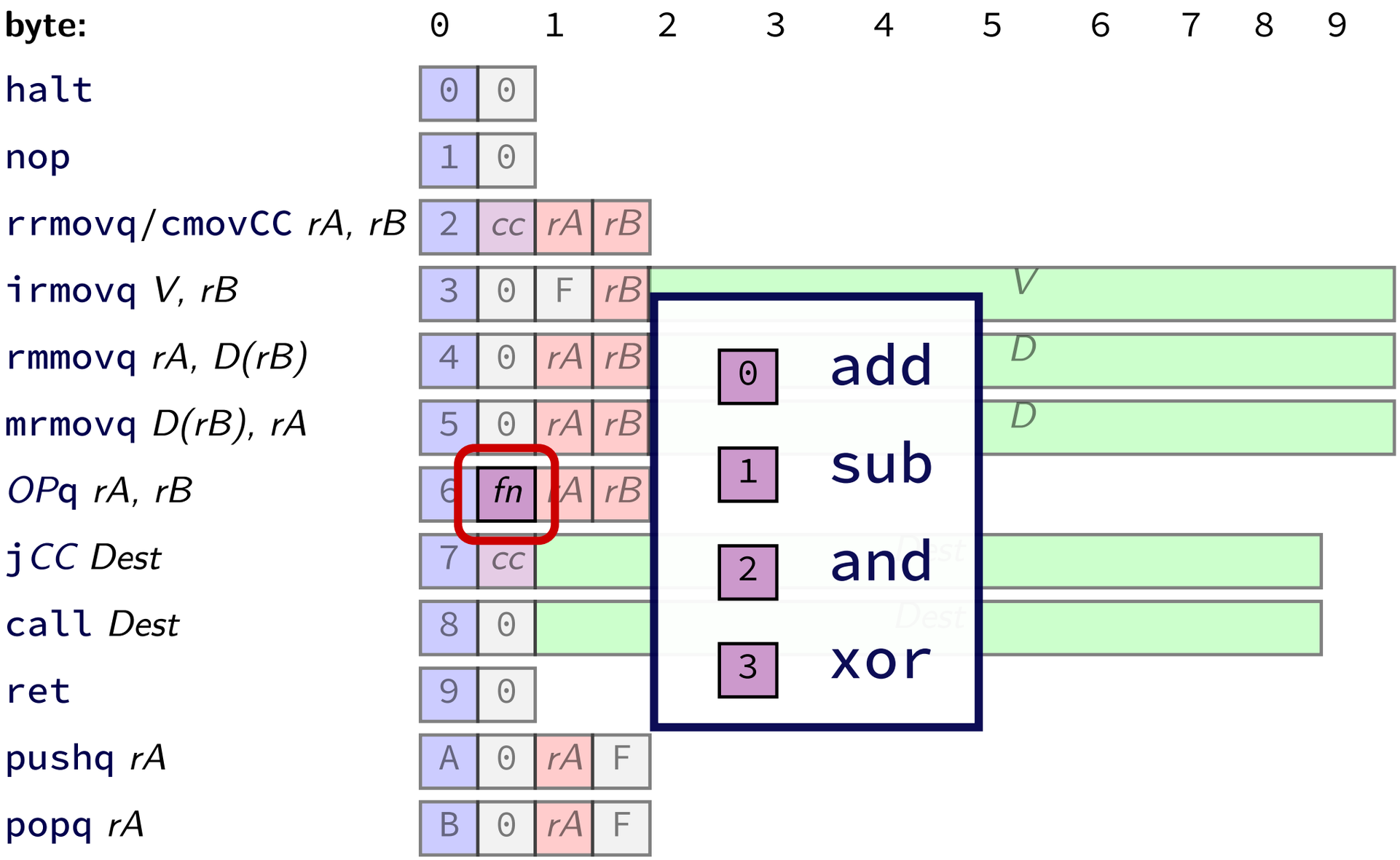
Y86-64 instruction formats



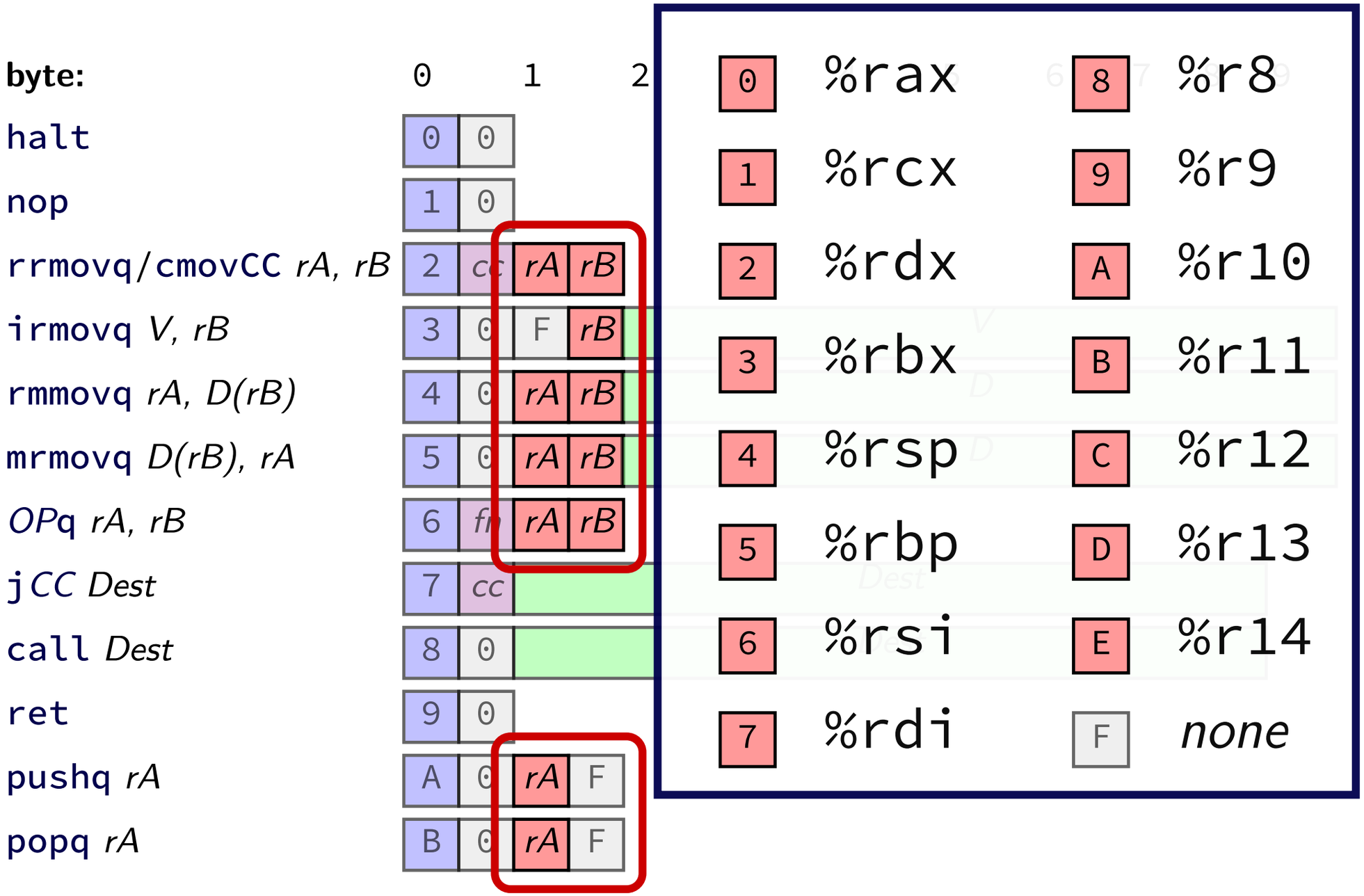
Secondary opcodes: `cmovcc`/`jcc`



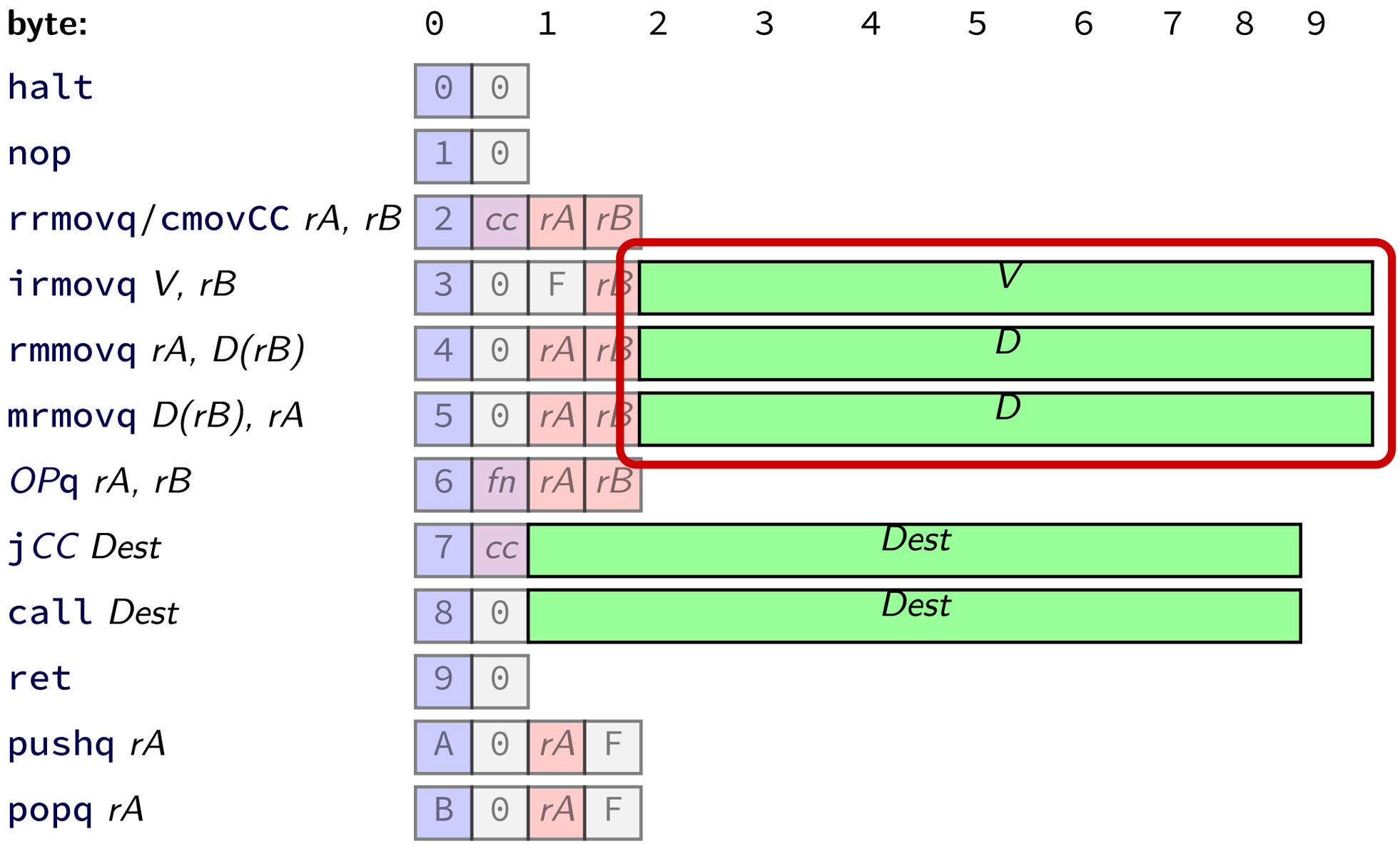
Secondary opcodes: *OP*q



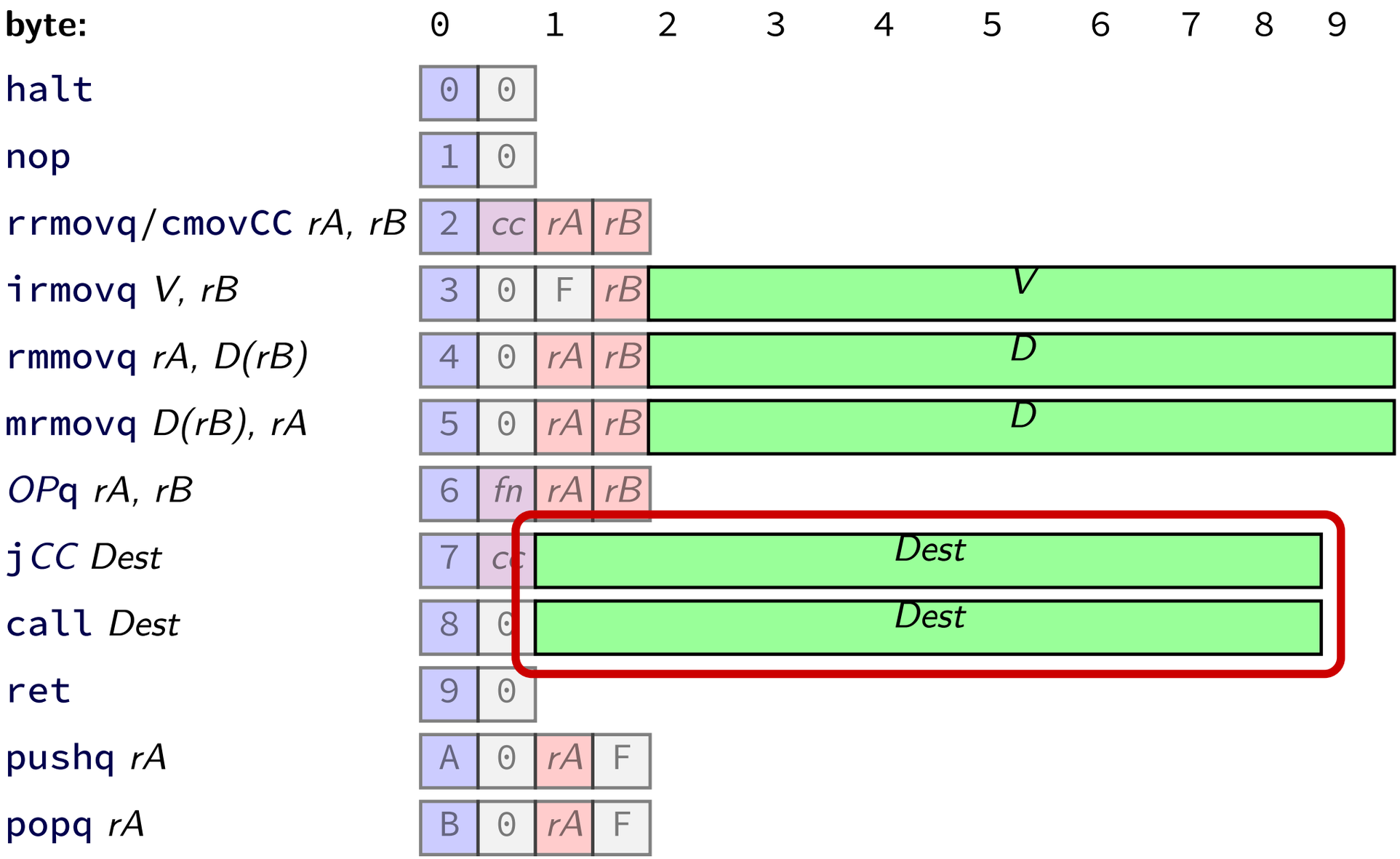
Registers: *rA*, *rB*



Immediates: *V, D, Dest*



Immediates: *V, D, Dest*



Y86-64 encoding (1)

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

x86-64:

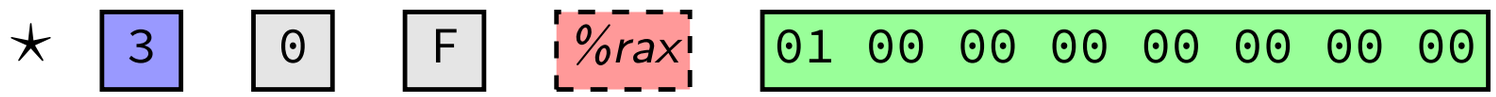
```
    movq %rdi, %rax
    addq $1, %rax
    ret
```

Y86-64:

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

Y86-64 encoding (2)

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



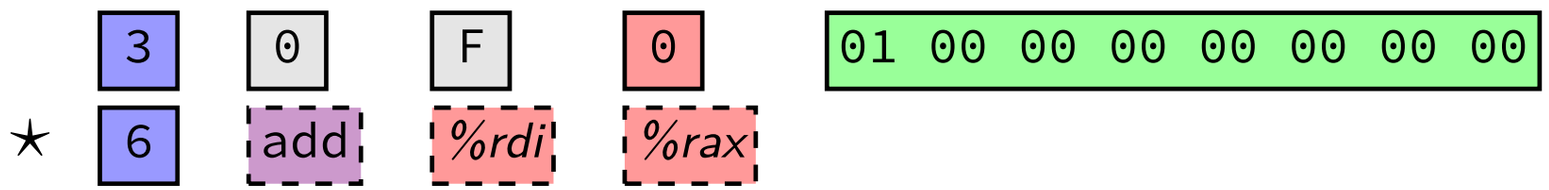
Y86-64 encoding (2)

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

★ | 3 | 0 | F | 0 | 01 00 00 00 00 00 00 00

Y86-64 encoding (2)

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



Y86-64 encoding (2)

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

	3	0	F	0	01 00 00 00 00 00 00 00
★	6	0	7	0	

Y86-64 encoding (2)

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

	3	0	F	0	01 00 00 00 00 00 00 00
	6	0	7	0	
⋆	9	0			

Y86-64 encoding (2)

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

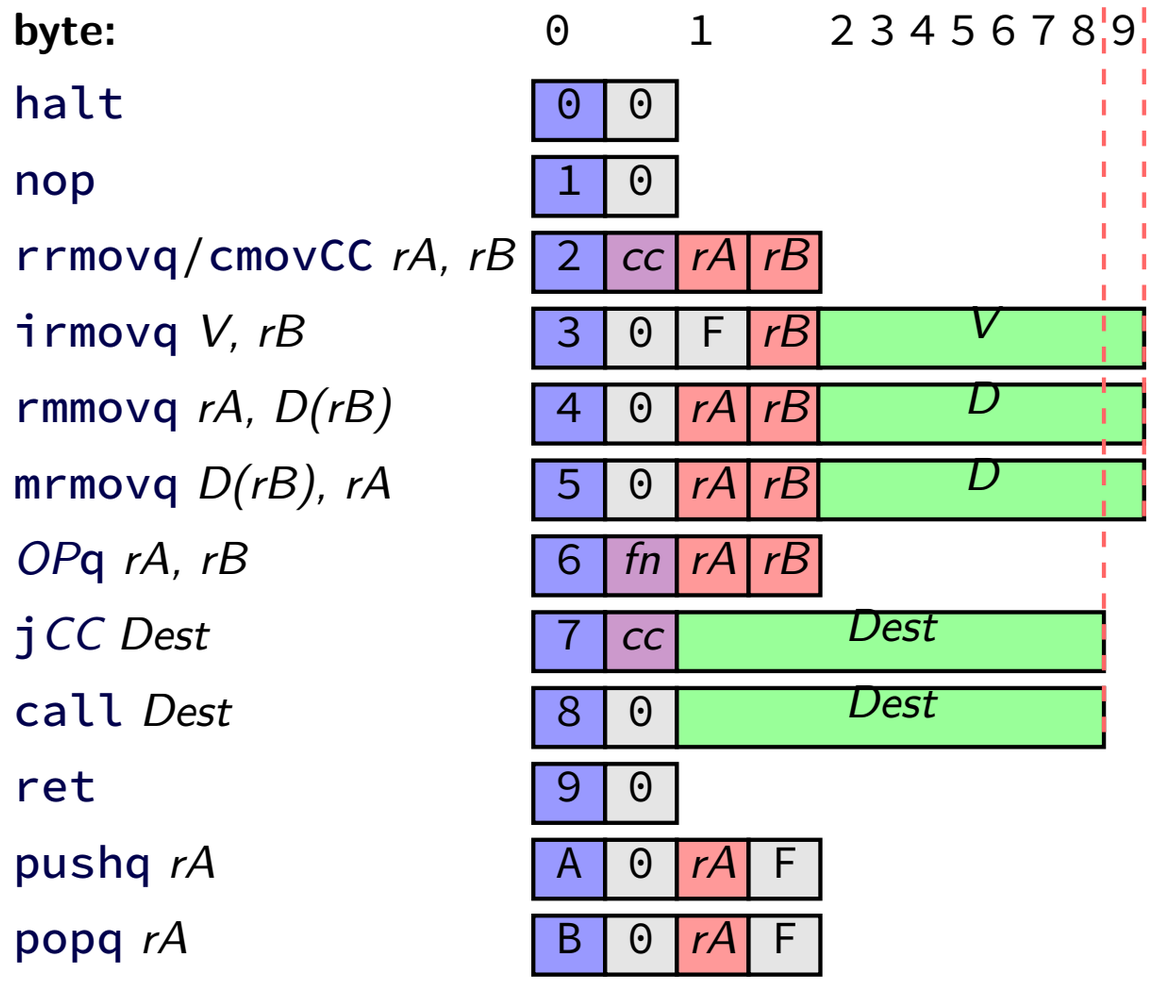
3	0	F	0	01 00 00 00 00 00 00 00
6	0	7	0	
9	0			

---

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

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
20 12 20 01 70 68 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
```