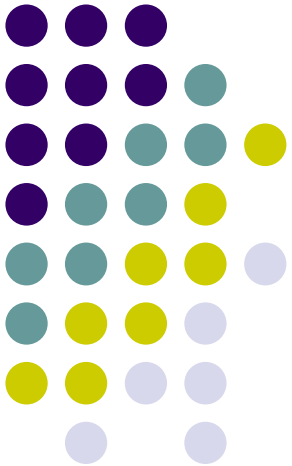
System Programming

02. A simple computer and Y86-64 ISA (ch 4.1)



2019. Fall

Instructor: Joonho Kwon

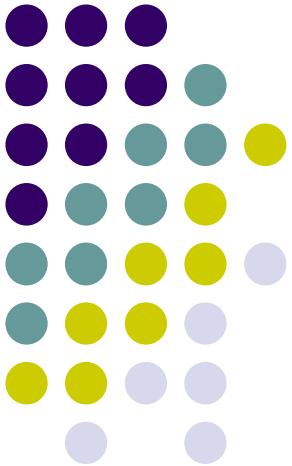
jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

A simple computer



These slides are based from http://cs.lmu.edu/~ray/notes/simplecomputer/

Let us try to understand

C0000003
00000001
000F4240
00000001
30000064
40000001
10000001
50000002
E0000003
C0000009

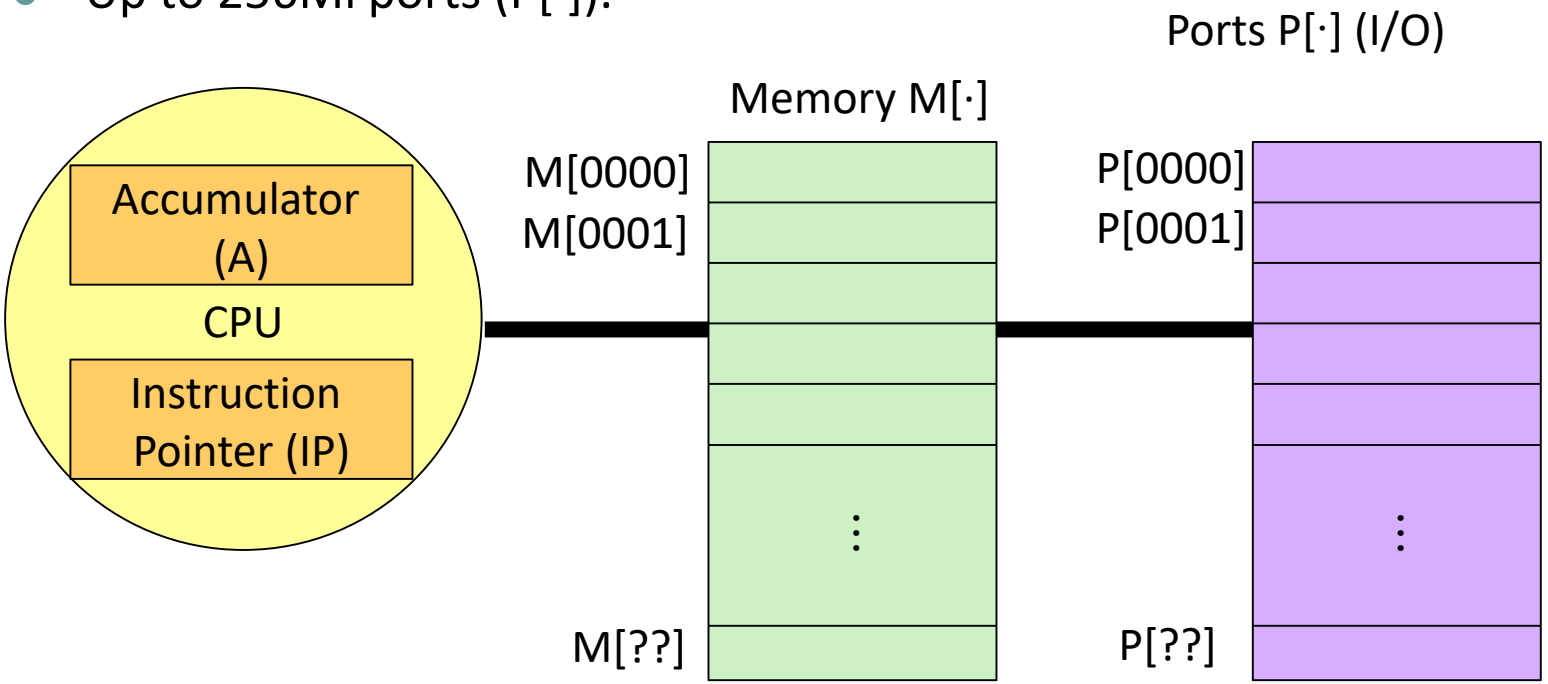
What does it mean?

How can we interpret?

What information we need?

Really, a simple computer

- Basic Components
  - A 32-bit register, called the accumulator (A).
  - A 32-bit register, called the instruction pointer (IP).
  - 256Mi words of memory, each 32-bits wide (M[·]).
    - 256 MiB = 256 * 2^{20} = 2^{8} * 2^{20} = 2^{28}
  - Up to 256Mi ports (P[·]).



Operation

- At each “step”
  - the computer reads the memory word whose address is in IP and then increments IP
  - Then it carries out the instruction that that word represents.

  - This is how most computers, uh, cores, work. It is called the **fetch-execute cycle.**

Again, the code. What we know?

```
C0000003
00000001
000F4240
00000001
30000064
40000001
10000001
50000002
E0000003
C0000009
```

- 28 bits will refer to a memory address or port number
  - Which 28 bits?
- How about remaining 4 bits ?

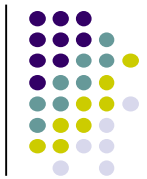
- Assumption
  - The first 4 bits as the **opcode**.
  - The remaining 28 bits will refer to a memory address or port number.

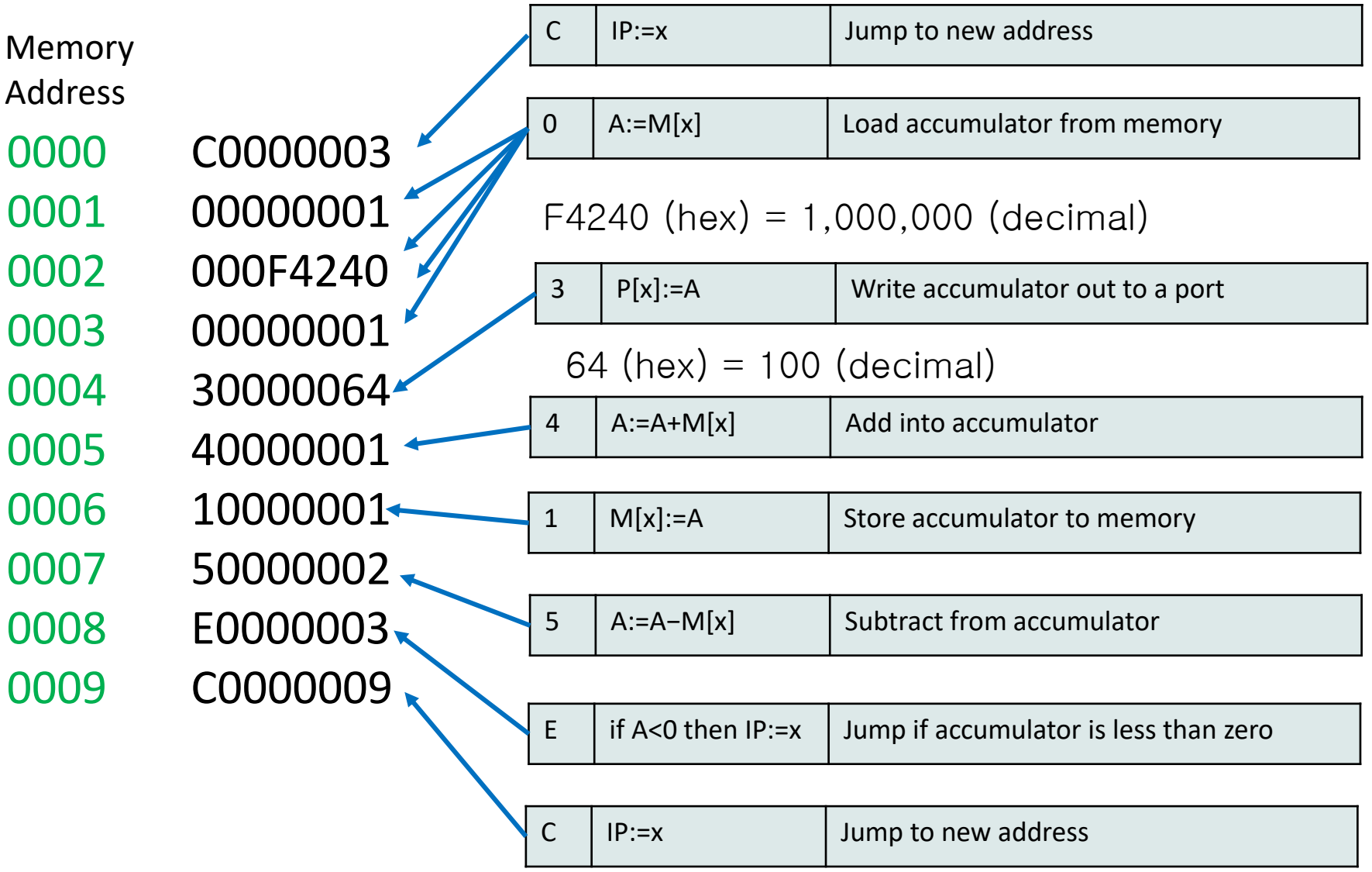
The Instructions

- We formally specify the operation of each instruction as follows.
  - All arithmetic is signed, modular, 32-bit integer arithmetic.

op	Action	Remarks
0	A:=M[x]	Load accumulator from memory
1	M[x]:=A	Store accumulator to memory
2	A:=P[x]	Read from a port into the accumulator
3	P[x]:=A	Write accumulator out to a port
4	A:=A+M[x]	Add into accumulator
5	A:=A−M[x]	Subtract from accumulator
6	A:=A×M[x]	Multiply into accumulator
7	A:=A÷M[x]	Divide accumulator
8	A:=A mod M[x]	Modulo
9	A:=A∧M[x]	Bitwise AND
A	A:=A∨M[x]	Bitwise OR
B	A:=A⊕M[x]	Bitwise XOR
C	IP:=x	Jump to new address
D	if A=0 then IP:=x	Jump if accumulator is zero
E	if A<0 then IP:=x	Jump if accumulator is less than zero
F	if A>0 then IP:=x	Jump if accumulator is greater than zero

Meaning




Memory Address
0000 C0000003
0001 00000001
0002 000F4240
0003 00000001
0004 30000064
0005 40000001
0006 10000001
0007 50000002
0008 E0000003
0009 C0000009
C IP:=x Jump to new address
0 A:=M[x] Load accumulator from memory
F4240 (hex) = 1,000,000 (decimal)
3 P[x]:=A Write accumulator out to a port
64 (hex) = 100 (decimal)
4 A:=A+M[x] Add into accumulator
1 M[x]:=A Store accumulator to memory
5 A:=A−M[x] Subtract from accumulator
E if A<0 then IP:=x Jump if accumulator is less than zero
C IP:=x Jump to new address

An Example Program

Memory Address	
0000	C0000003
0001	00000001
0002	000F4240
0003	00000001
0004	30000064
0005	40000001
0006	10000001
0007	50000002
0008	E0000003
0009	C0000009

- when loaded into memory at address 0
  - outputs powers of two, starting with 1, and going just past 1,000,000, to port 100 (64 hex):

Machine code

- Machine code
  - Hard to read
  - Just listing the contents of memory that the processor executes

```
0000  C0000003
0001  00000001
0002  000F4240
0003  00000001
0004  30000064
0005  40000001
0006  10000001
0007  50000002
0008  E0000003
0009  C0000009
```

Assembly code (1/2)

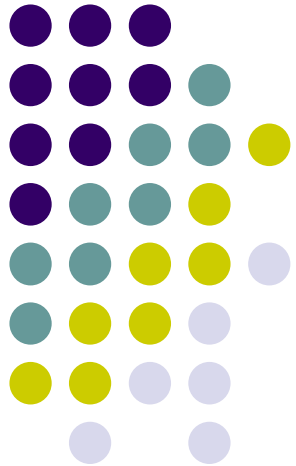
- Let's use mnemonics for each instruction
  - the mnemonics will be
    - LOAD, STORE, IN, OUT, ADD, SUB, MUL, DIV, MOD, AND, OR, XOR, JUMP, JZ, JLZ, JGZ

```
0000  C0000003          JUMP    start       ; begin by jumping over the data area
0001  00000001  pow:    1                   ; store the current power value here
0002  000F4240  limit:  1000000             ; we'll be computing powers up to this amount
0003  00000001  start:  LOAD    pow         ; bring the value into accumulator to use
0004  30000064          OUT     100         ; output the current power
0005  40000001          ADD     pow         ; adding to itself makes the next power!
0006  10000001          STORE   pow         ; store it (for next time)
0007  50000002          SUB     limit        ; compare with limit, subtracting helps
0008  E0000003          JLZ     start       ; if not yet past limit, keep going
0009  C0000009  end:    JUMP    end         ; this "stops" the program!
```

Assembly code (2/2)

- In general, each line of an assembly language program contains:
  - An optional label
    - so you don't have to memorize physical addresses
  - Either
    - A data value
    - An instruction and its operands. An operand can be a direct value or a label.
      - Labels are just convenient shorthands for values anyway.
  - Comments, beginning with the ; character.

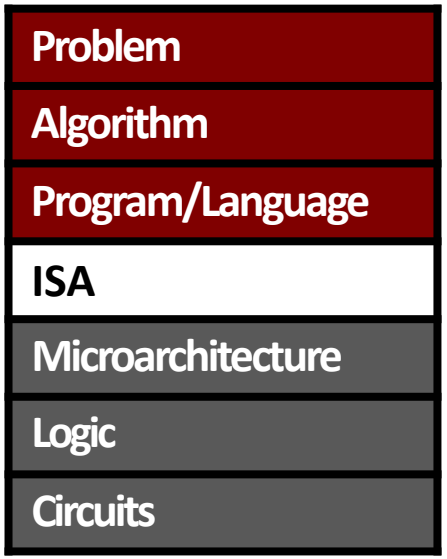
Ch04.1. Y86-64 ISA



These slides are based on the text.

Levels of Transformation

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - ISA, uarch, circuits
  - "Architecture" = ISA + microarchitecture



ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implements “acceleration”
  - Add instruction vs. Adder implementation
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Bit serial, ripple carry, carry lookahead adders
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, …
- Uarch usually changes faster than ISA
  - Few ISAs (x86, SPARC, MIPS, Alpha) but many uarchs

ISA

- Instructions
  - Opcodes, Addressing Modes
  - Instruction Types and Format
  - Registers, Condition Codes

- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O
- Task Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

Example ISAs

- x86 — dominant in desktops, servers
- ARM — dominant in mobile devices
- POWER — Wii U, IBM supercomputers and some servers
- MIPS — common in consumer wifi access points
- SPARC — some Oracle servers, Fujitsu supercomputers
- z/Architecture — IBM mainframes
- Z80 — TI calculators
- SHARC — some digital signal processors
- Itanium — some HP servers
- RISC V — some embedded
- …

ISA Tradeoffs

- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?

Instruction length (1/2)

- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)

Instruction length (2/2)

- Variable length: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
  - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently

Addressing modes

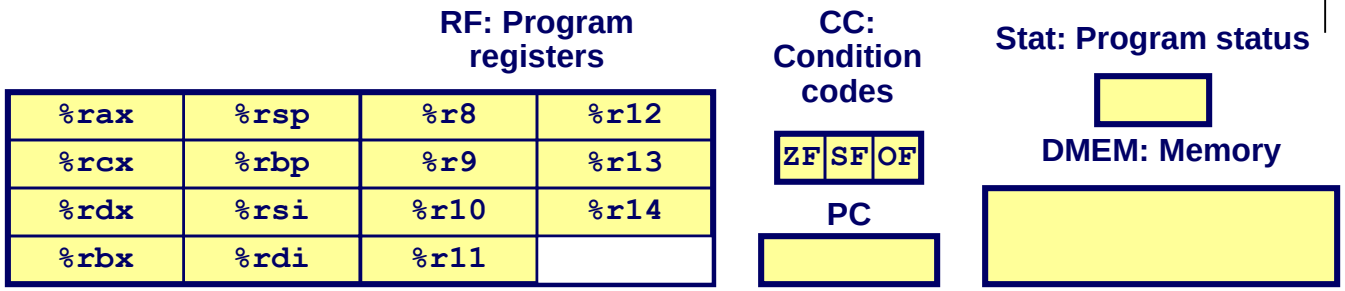
- Addressing mode
  - specifies how to obtain an operand of an instruction
    - Register
    - Immediate
    - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, …)
- Example
  - x86-64: 10(%r11,%r12,4)
  - ARM: %r11 << 3 (shift register value by constant)
  - VAX: ((%r11)) (register value is pointer to pointer)

Condition codes

- Codes
  - cmpq %r11, %r12
  - je somewhere

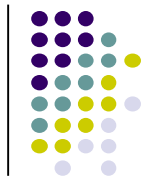
- could do:
  - /* _Branch if _EQual */
  - beq %r11, %r12, somewhere

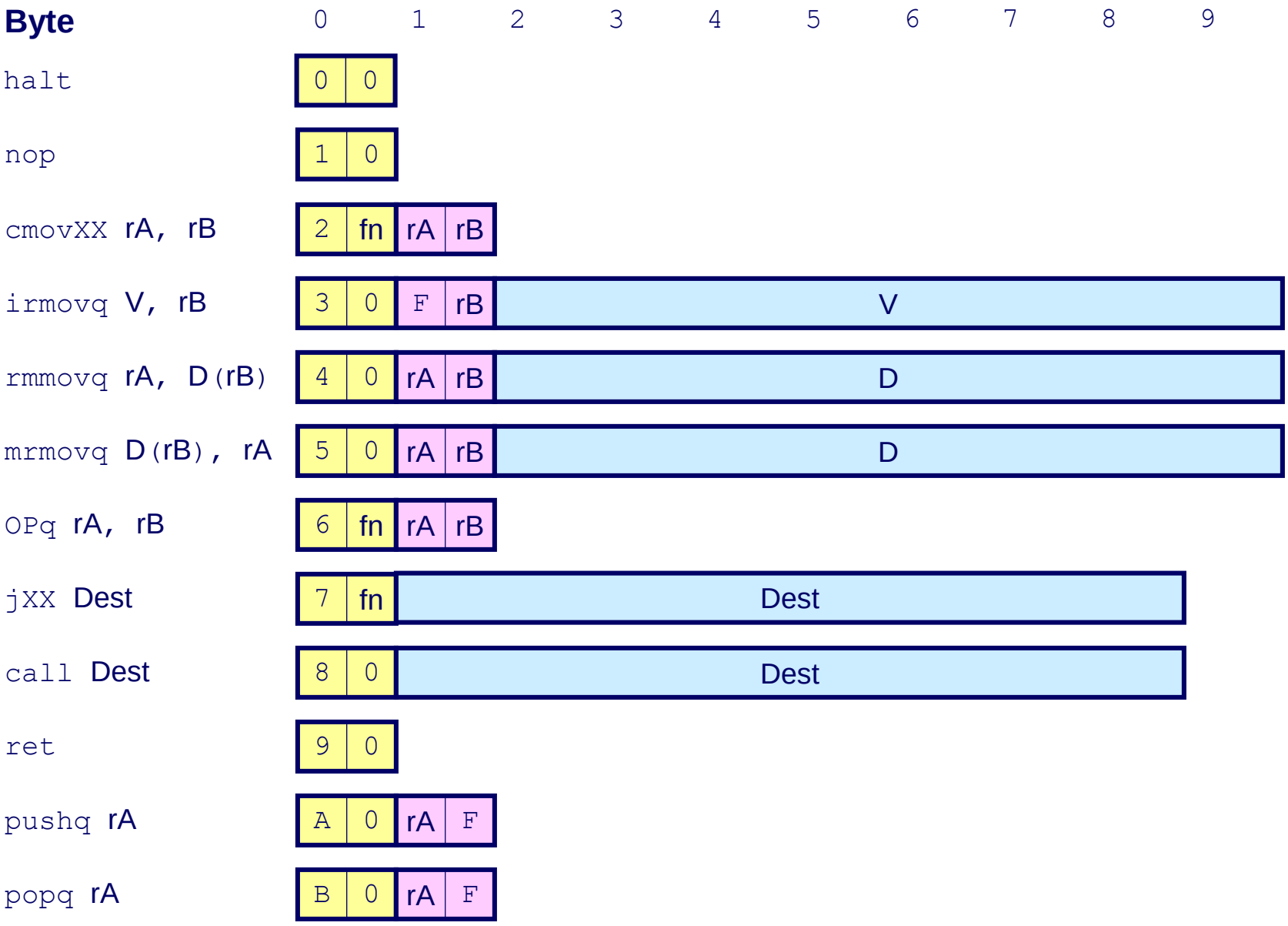
Y86-64 Processor State



- Program Registers
  - 15 registers (omit %r15). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
  - ZF: Zero SF:Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86-64 Instruction Set #1

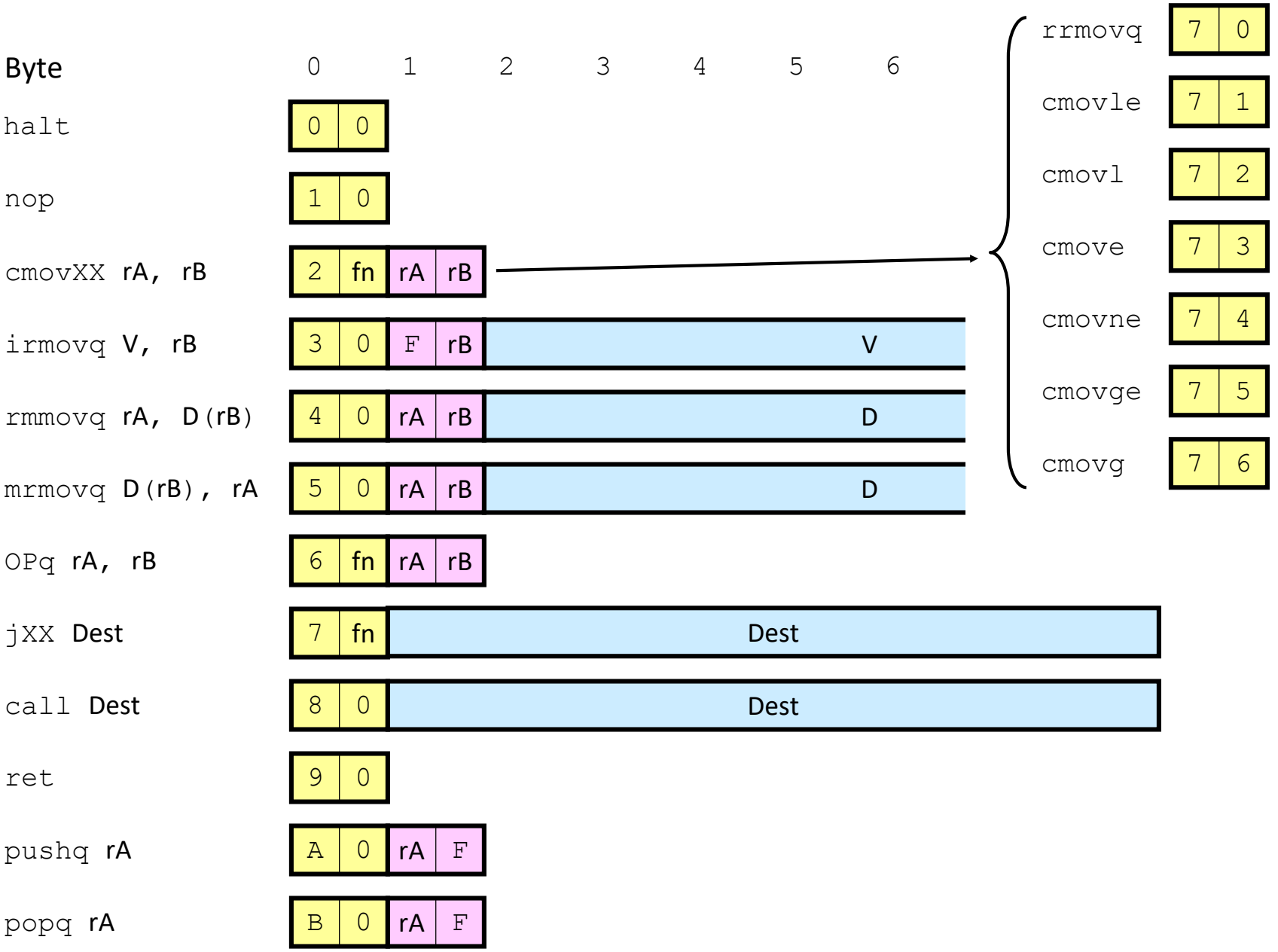




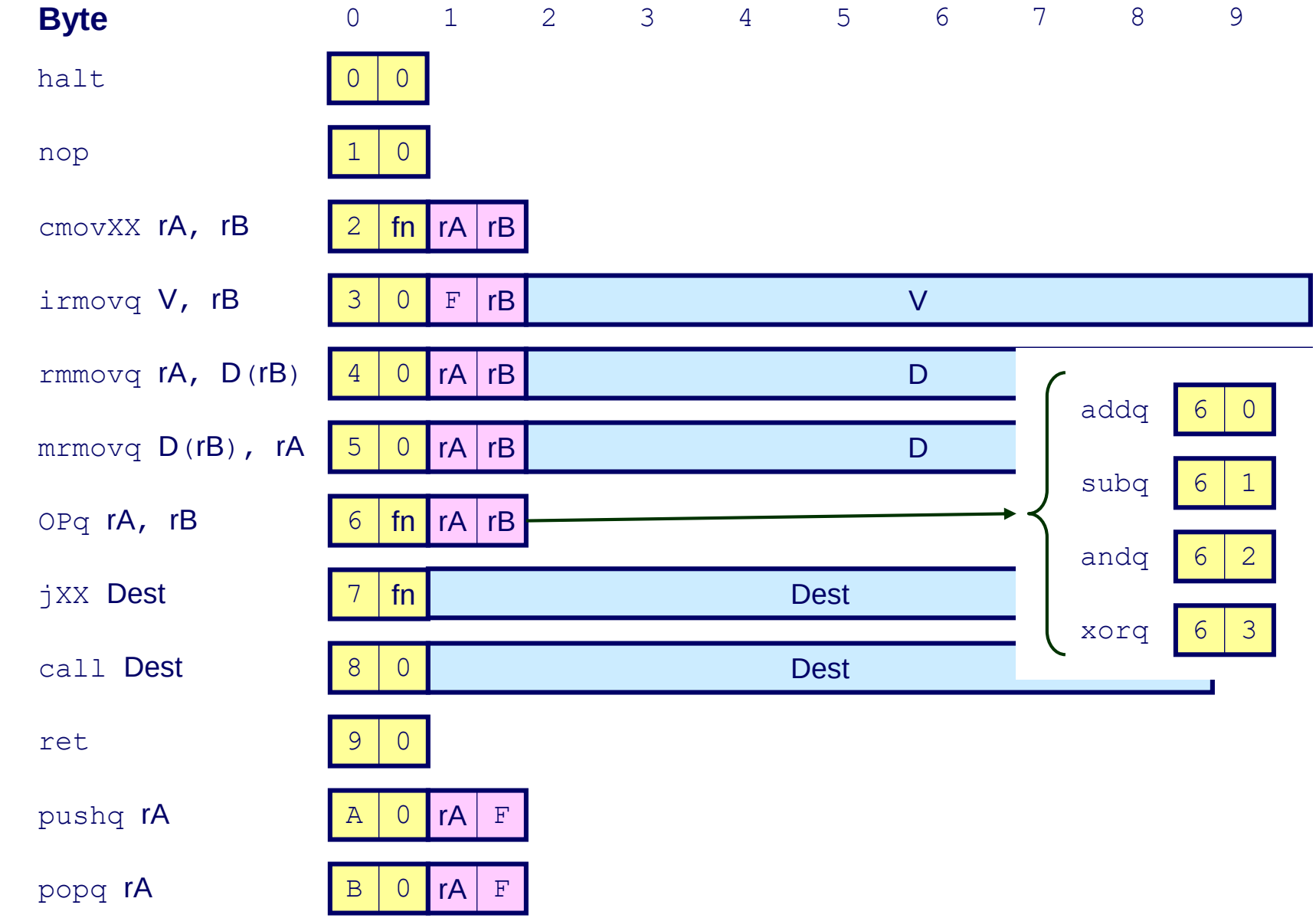
Y86-64 Instructions

- Format
  - 1–10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with x86-64
  - Each accesses and modifies some part(s) of the program state

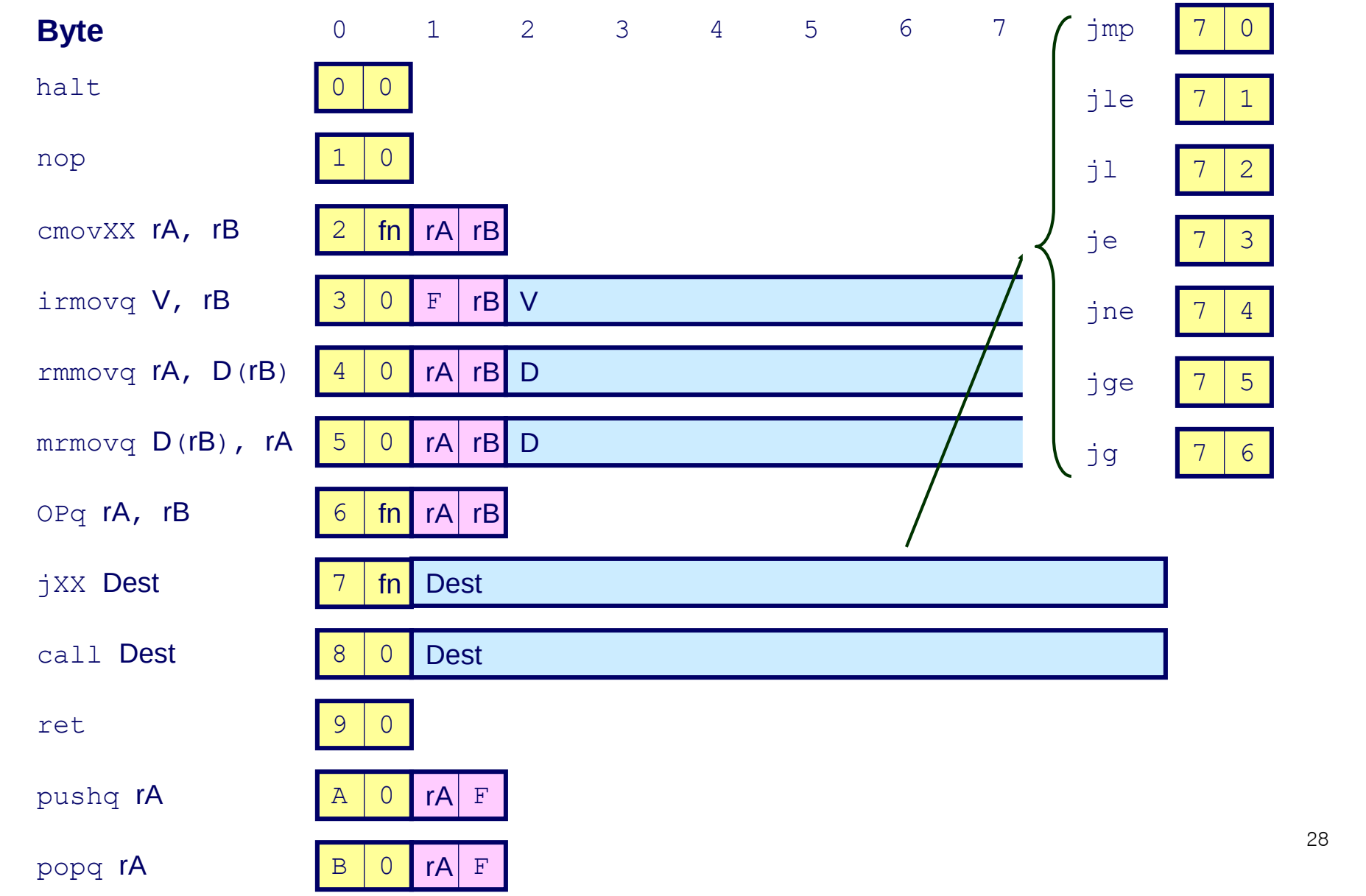
Y86-64 Instruction Set #2



Y86-64 Instruction Set #3



Y86-64 Instruction Set #4



Encoding Registers

- Each register has 4-bit ID
  - Same encoding as in x86-64

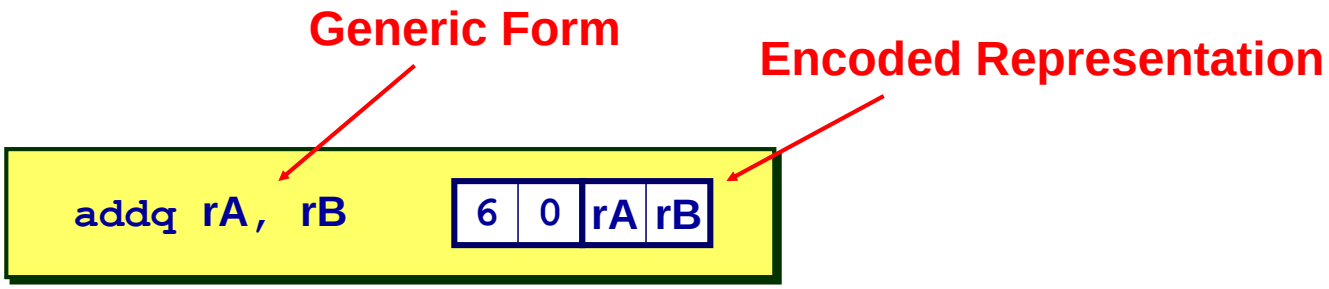
%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

- Register ID 15 (0xF) indicates “no register”
  - Will use this in our hardware design in multiple places

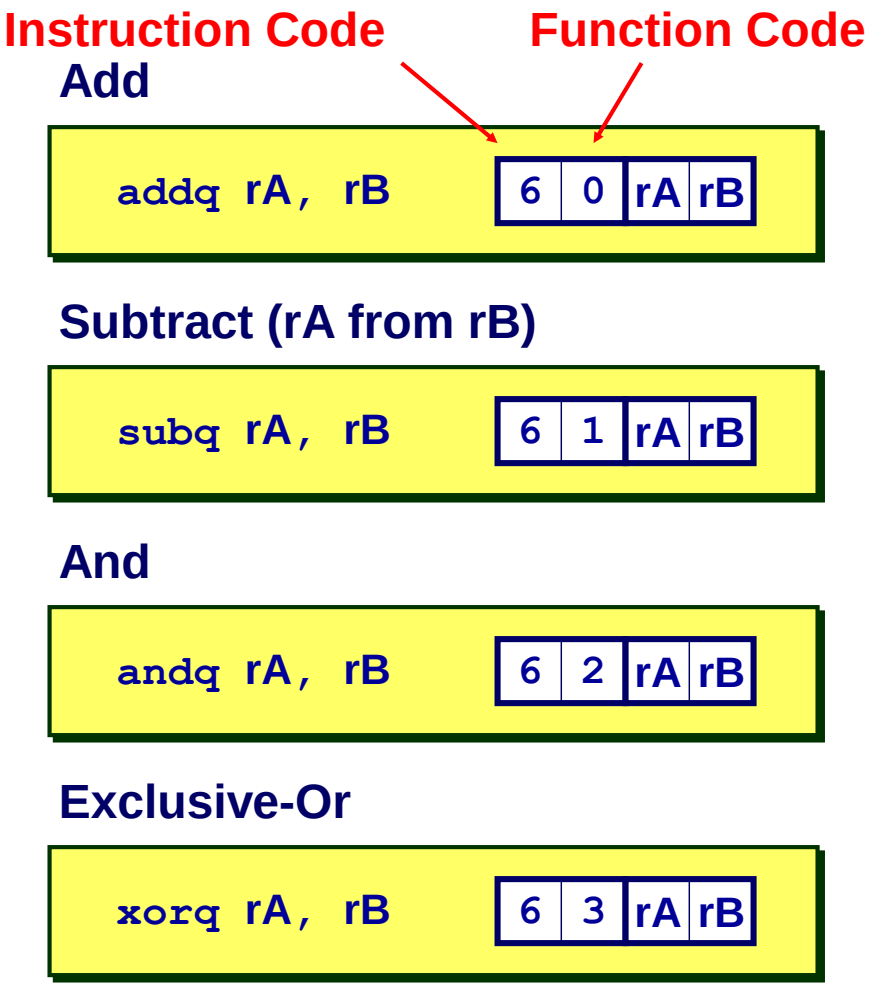
Instruction Example

- Addition Instruction
  - Add value in register rA to that in register rB
    - Store result in register rB
    - Note that Y86-64 only allows addition to be applied to register data
  - Set condition codes based on result



  - e.g., addq %rax,%rsi Encoding: 60 06
  - Two-byte encoding
    - First indicates instruction type
    - Second gives source and destination registers

Arithmetic and Logical Operations



- Refer to generically as "OPq"
- Encodings differ only by "function code"
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

Move Operations

Register ➔ Register

`rrmovq` rA, rB | 2 | 0 |

Immediate ➔ Register

`irmovq` V, rB | 3 | 0 | F | rB | V |

Register ➔ Memory

`rmmovq` rA, D(rB) | 4 | 0 | rA | rB | D |

Memory ➔ Register

`mrmovq` D(rB), rA | 5 | 0 | rA | rB | D |

- Like the x86-64 movq instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding:	`30 F2 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding:	`20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding:	`50 15 f4 ff ff ff ff ff ff ff`
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
Encoding:	`40 64 1c 04 00 00 00 00 00 00`

Conditional Move Instructions

Move Unconditionally

`rrmovq` rA, rB

2	0	rA	rB

Move When Less or Equal

`cmovle` rA, rB

2	1	rA	rB

Move When Less

`cmovl` rA, rB

2	2	rA	rB

Move When Equal

`cmove` rA, rB

2	3	rA	rB

Move When Not Equal

`cmovne` rA, rB

2	4	rA	rB

Move When Greater or Equal

`cmovge` rA, rB

2	5	rA	rB

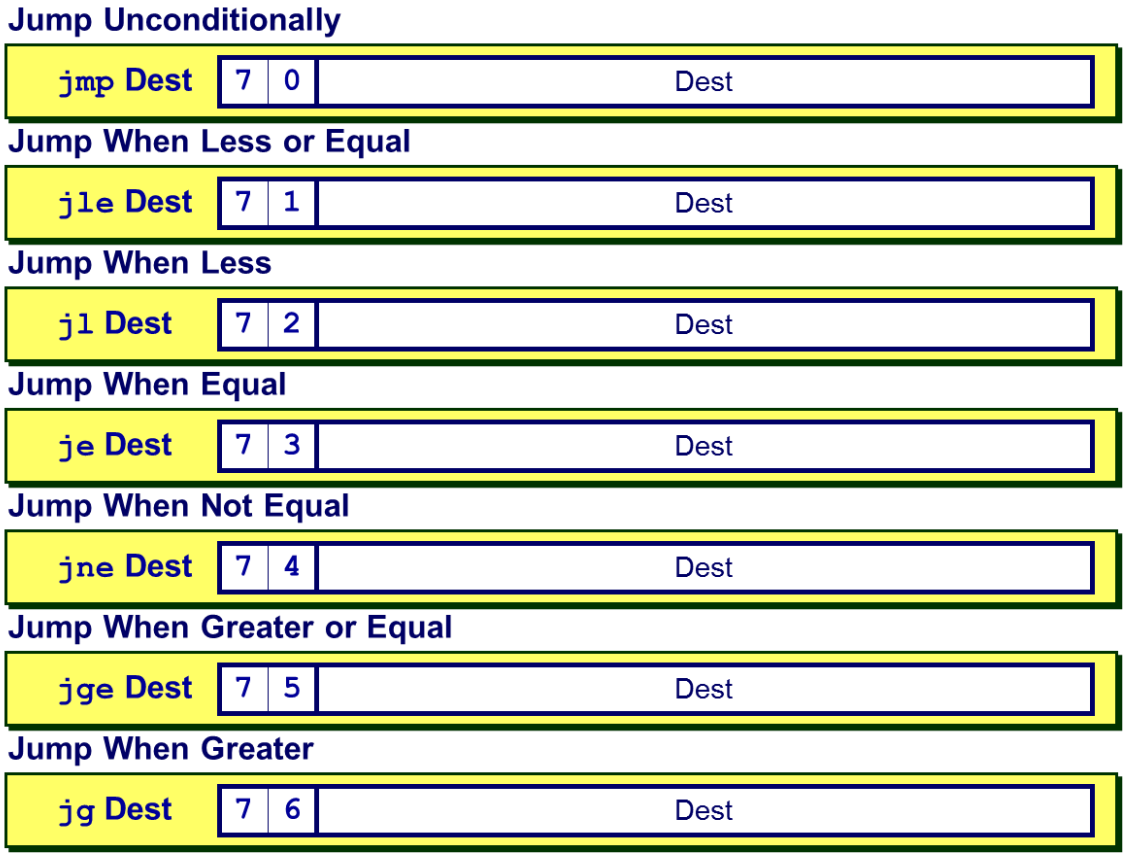
Move When Greater

`cmovg` rA, rB

2	6	rA	rB

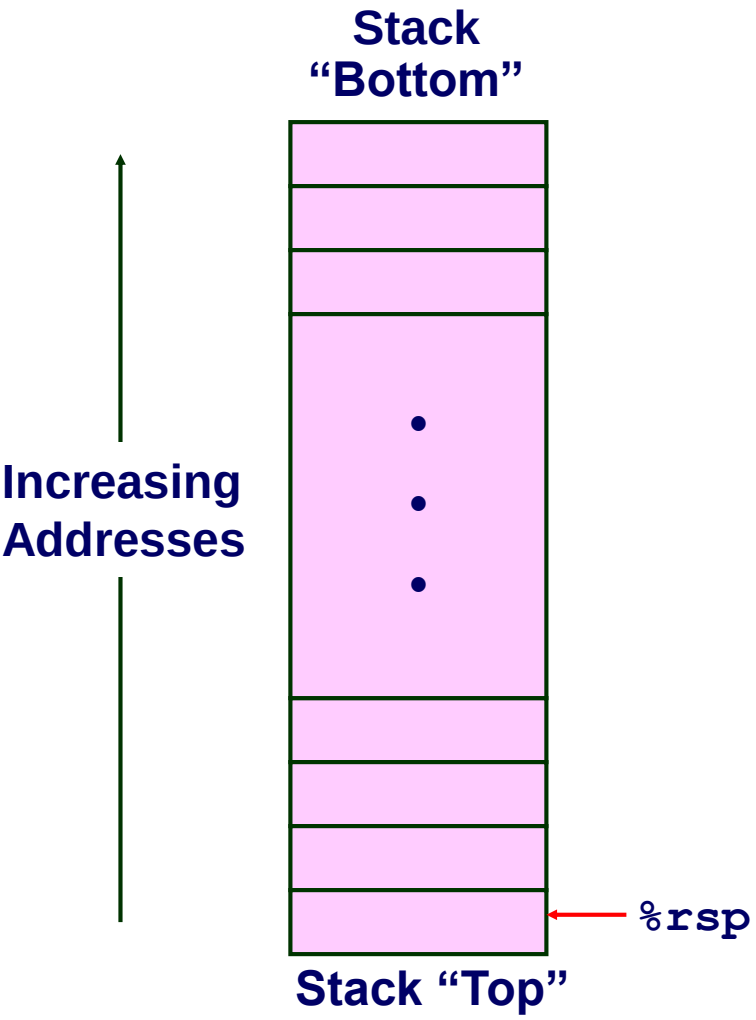
- Refer to generically as "cmovXX"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of rrmovq instruction
  - (Conditionally) copy value from source to destination register

Jump Instructions



- Refer to generically as "jXX"
- Encodings differ only by "function code"
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

Y86 Program Stack



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by %rsp
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

Stack Operations

- Decrement %rsp by 8
- Store word from rA to memory at %rsp
- Like x86-64

- DecreRead word from memory at %rsp
- Save in rA
- Increment %rsp by 8
- Like x86-64

Subroutine Call and Return

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

- Pop value from stack
- Use as address for next instruction
- Like x86-64

Miscellaneous Instructions

- Don’t do anything

- Stop executing instructions
- x86-64 has comparable instruction, but can’t execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code
AOK	1

- Normal operation

Mnemonic	Code
HLT	2

- Halt instruction encountered

Mnemonic	Code
ADR	3

- Bad address (either instruction or data) encountered

Mnemonic	Code
INS	4

- Invalid instruction encountered

- Desired Behavior
  - If AOK, keep going
  - Otherwise, stop program execution

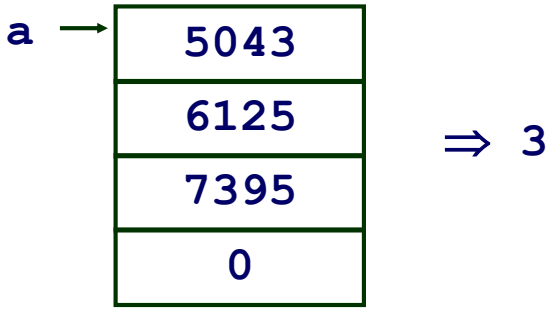
Writing Y86-64 Code

- Try to Use C Compiler as Much as Possible
  - Write code in C
  - Compile for x86-64 with gcc –Og –S
  - Transliterate into Y86-64
  - Modern compilers make this more difficult

Coding Example

- Find number of elements in null-terminated list

```
int len1(int a[]);
```



$\Rightarrow$ 3

Y86-64 Code Generation Example

- First Try
  - Write typical array code

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
  long len;
  for (len = 0; a[len]; len++)
       ;
  return len;
}
```

  - Compile with `gcc -Og -S`

- Problem
  - Hard to do array indexing on Y86-64
  - Since don't have scaled addressing modes

```
L3:
   addq $1,%rax
   cmpq  $0, (%rdi,%rax,8)
   jne L3
```

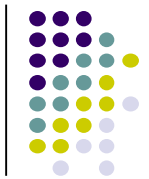
Y86-64 Code Generation Example #2

- Second Try
  - Write C code that mimics expected Y86-64 code

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```

- Result
  - Compiler generates exact same code as before!
  - Compiler converts both versions into same intermediate form

Y86-64 Code Generation Example #3



```
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax         # len = 0
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    je Done                 # If zero, goto Done
Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi          # a++
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    jne Loop                # If !0, goto Loop
Done:
    ret
```

Register	Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

Y86-64 Sample Program Structure #1

```
init:                       # Initialization
    . . .
    call Main
    halt

    .align 8                # Program data
array:
    . . .

Main:                       # Main function
    . . .
    call len    . . .

len:                        # Length function
    . . .

    .pos 0x100              # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

Y86-64 Program Structure #2

```
init:
      # Set up stack pointer
      irmovq Stack, %rsp
      # Execute main program
      call Main
      # Terminate
      halt

# Array of 4 elements + terminating 0
      .align 8
Array:
      .quad 0x000d000d000d000d
      .quad 0x00c000c000c000c0
      .quad 0x0b000b000b000b00
      .quad 0xa000a000a000a000
      .quad 0
```

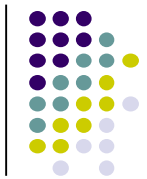
- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names

Y86-64 Program Structure #3

```
Main:
      irmovq array,%rdi
      # call len(array)
      call len
      ret
```

- Set up call to len
  - Follow x86-64 procedure conventions
  - Push array address as argument

Assembling Y86-64 Program

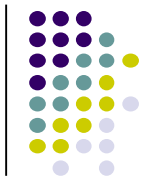


```
unix> yas len.ys
```

- Generates “object code” file len.yo
- Actually looks like disassembler output

```
0x054:                          | len:
0x054: 30f80100000000000000     |   irmovq $1, %r8          # Constant 1
0x05e: 30f90800000000000000     |   irmovq $8, %r9          # Constant 8
0x068: 30f00000000000000000     |   irmovq $0, %rax         # len = 0
0x072: 50270000000000000000     |   mrmovq (%rdi), %rdx     # val = *a
0x07c: 6222                     |   andq %rdx, %rdx         # Test val
0x07e: 73a000000000000000       |   je Done                 # If zero, goto Done
0x087:                          | Loop:
0x087: 6080                     |   addq %r8, %rax          # len++
0x089: 6097                     |   addq %r9, %rdi          # a++
0x08b: 50270000000000000000     |   mrmovq (%rdi), %rdx     # val = *a
0x095: 6222                     |   andq %rdx, %rdx         # Test val
0x097: 748700000000000000       |   jne Loop                # If !0, goto Loop
0x0a0:                          | Done:
0x0a0: 90                       |   ret
```

Simulating Y86-64 Program



```
unix> yis len.yo
```

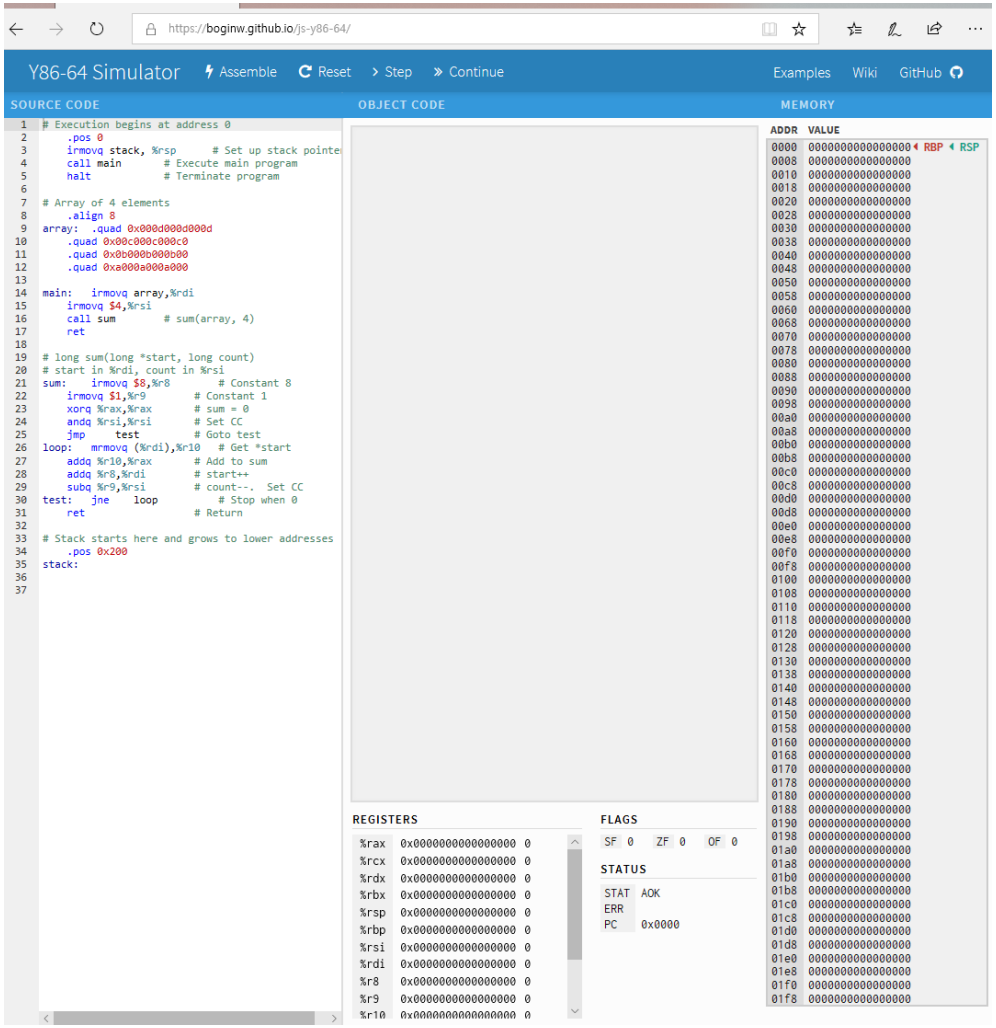
- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000004
%rsp:   0x0000000000000000      0x0000000000000100
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000001
%r9:    0x0000000000000000      0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000      0x0000000000000053
0x00f8: 0x0000000000000000      0x0000000000000013
```

Y86-64 Simulator

- https://boginw.github.io/js-y86-64/



Example: SUM function

```
long sum(long *start, long count)
{
   long sum = 0;
   while (count) {
      sum += *start;
      start ++;
      count--;
   }
   return sum;
}
```

Y86 code (1/2)

```
long sum(long *start, long count)
{
  long sum = 0;
  while (count) {
    sum += *start;
    start ++;
    count--;
  }
  return sum;
}
```

```
                                  | # Execution begins at address 0
0x0000:                           | .pos 0
0x0000: 30f40002000000000000      | irmovq stack, %rsp # Set up stack pointer
0x000a: 8038000000000000          | call main # Execute main program
0x0013: 00                        | halt # Terminate program
                                  |
                                  | # Array of 4 elements
0x0014:                           | .align 8
0x0018: 0d000d000d                | array: .quad 0x000d000d000d
0x0020: c000c000c0                | .quad 0x00c000c000c0
0x0028: 000b000b000b              | .quad 0x0b000b000b00
0x0030: 00a000a000a0              | .quad 0xa000a000a000
|
0x0038: 30f7180000000000000       | main: irmovq array,%rdi
0x0042: 30f6040000000000000       | irmovq $4,%rsi
0x004c: 8056000000000000          | call sum # sum(array, 4)
0x0055: 90                        | ret
                                  |
```

Y86 code (2/2)

```
long sum(long *start, long count)
{
  long sum = 0;
  while (count) {
    sum += *start;
    start ++;
    count--;
  }
  return sum;
}
```

```
                                    | # long sum(long *start, long count)
                                    | # start in %rdi, count in %rsi
0x0056: 30f80800000000000000        | sum: irmovq $8,%r8 # Constant 8
0x0060: 30f90100000000000000        | irmovq $1,%r9 # Constant 1
0x006a: 6300                        | xorq %rax,%rax # sum = 0
0x006c: 6266                        | andq %rsi,%rsi # Set CC
0x006e: 7087000000000000            | jmp test # Goto test
0x0077: 50a70000000000000000        | loop: mrmovq (%rdi),%r10 # Get *start
0x0081: 60a0                        | addq %r10,%rax # Add to sum
0x0083: 6087                        | addq %r8,%rdi # start++
0x0085: 6196                        | subq %r9,%rsi # count--. Set CC
0x0087: 7477000000000000            | test: jne loop # Stop when 0
0x0090: 90                          | ret # Return
                                    |
                                    | # Stack starts here, grows to lower addresses
0x0091:                             | .pos 0x200
0x0200:                             | stack:
```

Summary

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC

Q&A