CS:APP Chapter 4
Computer Architecture
Instruction Set Architecture
指令集体系结构

任课教师：

宿红毅　　张艳　　黎有琦　　颜珂

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

为何要研究CPU设计
Why do We Study the CPU Design?

理解基本的计算机组织结构 Understand basic computer organization

- 指令集体系结构 Instruction set architecture

深度探索CPU工作机制 Deeply explore the CPU working mechanism

- 指令是如何执行的 How the instruction is executed

有助于编程 Help you programming

- 完全理解计算机的组织和工作方式有助于编写更加稳定和高效的代码 Fully understand how computer is organized and works will help you write more stable and efficient code

指令集体系结构 #1
Instruction Set Architecture #1

什么是ISA What is ISA ?

- 汇编语言抽象 Assemble Language Abstraction
  - 处理器支持的汇编语言 assembly supported by a processor
- 机器语言抽象 Machine Language Abstraction
  - 字节级表示 Byte-level representation

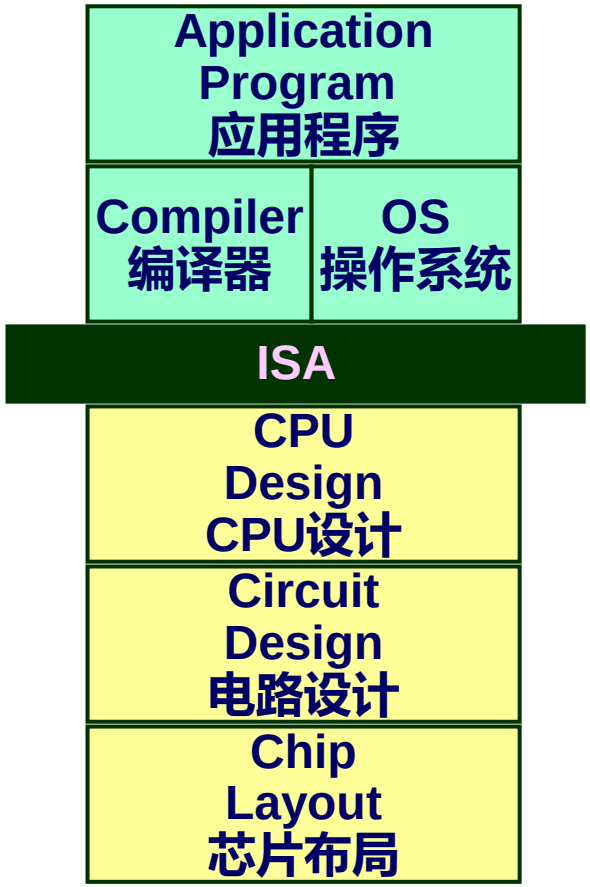
它提供什么? What does it provide?

- 真实计算机的一个抽象，隐藏了实现细节 An abstraction of the real computer, hide the details of implementation
  - 计算机指令语法 The syntax of computer instructions
  - 指令语义 The semantics of instructions
  - 执行模式 The execution model
  - 程序员可见的计算机状态 Programmer-visible computer status

指令集体系结构
Instruction Set Architecture

汇编语言视角 Assembly Language View

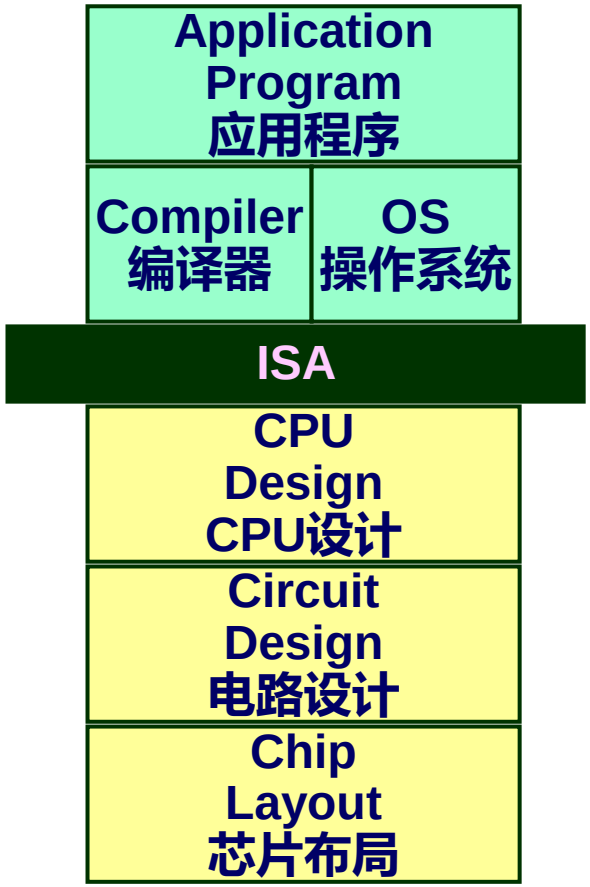
- 处理器状态 Processor state
  - 寄存器、内存。。。Registers, memory, …
- 指令 Instructions
  - `addq`, `pushq`, `ret`, …
  - 指令如何编码为字节序列 How instructions are encoded as bytes



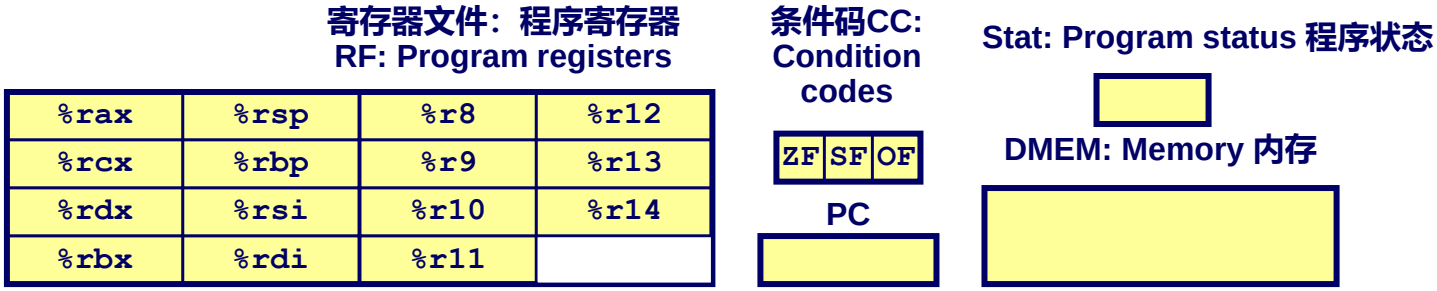
指令集体系结构
Instruction Set Architecture

抽象层次 Layer of Abstraction

- 之上：程序机器如何工作 Above: how to program machine
  - 处理器顺序执行指令 Processor executes instructions in a sequence
- 之下：需要构建什么 Below: what needs to be built
  - 使用各种技巧使其运行更快 Use variety of tricks to make it run fast
  - 例如同时执行多条指令 E.g., execute multiple instructions simultaneously



Y86-64处理器状态
Y86-64 Processor State



- 程序寄存器 Program Registers
  - 15个寄存器（省略%r15），每个64位 15 registers (omit `%r15`). Each 64 bits
- 条件码 Condition Codes
  - 算术或逻辑运算指令设置单个比特位标志 Single-bit flags set by arithmetic or logical instructions
    - » ZF: Zero零　　SF:Negative负数　　OF: Overflow溢出

Y86-64处理器状态
Y86-64 Processor State

寄存器文件：程序寄存器
RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

条件码CC: Condition codes

ZF	SF	OF

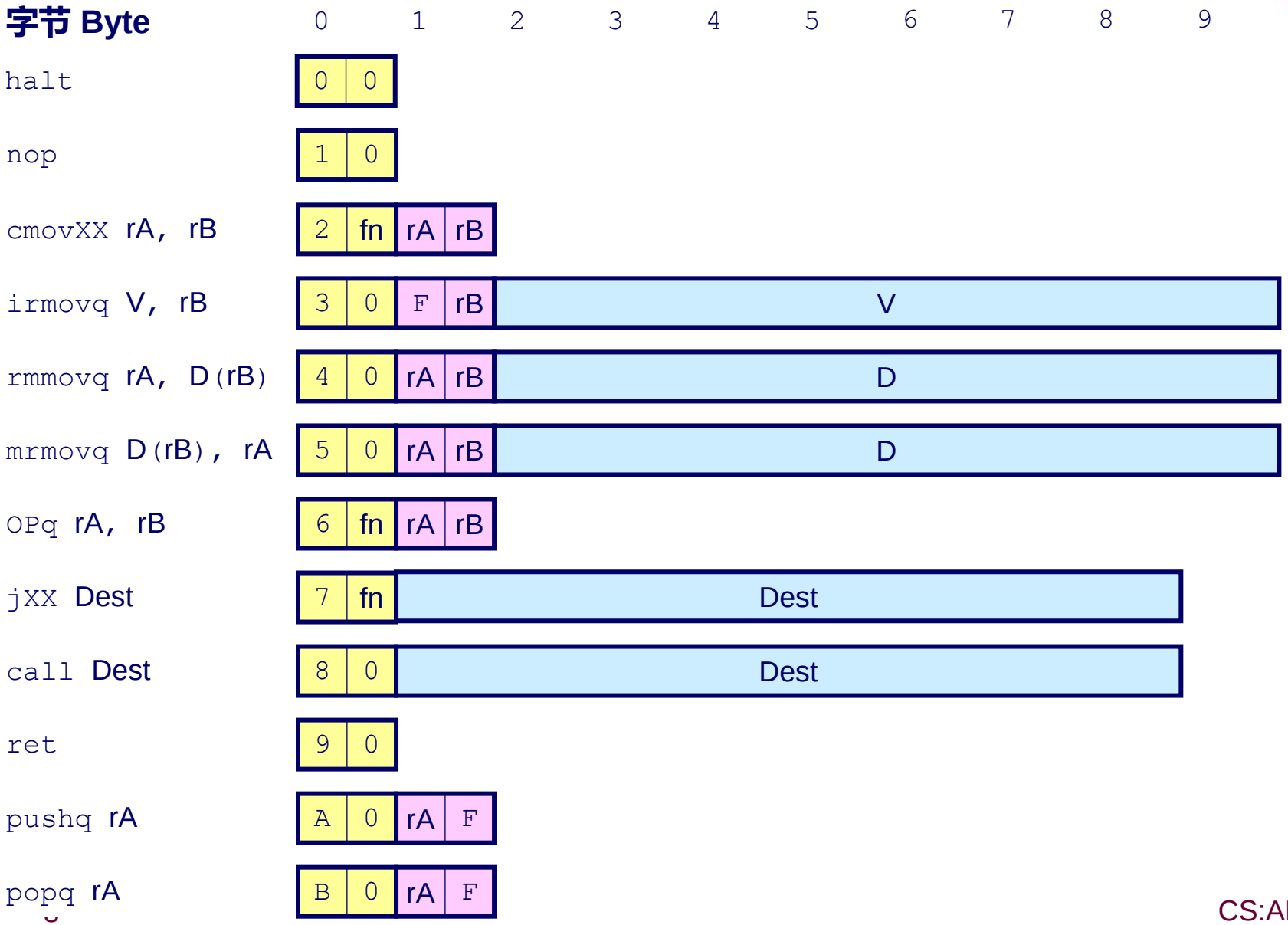
PC

Stat: Program status 程序状态

DMEM: Memory 内存

- 程序计数器 Program Counter
  - 指明下一条指令地址 Indicates address of next instruction
- 程序状态 Program Status
  - 指明正常运行还是一些错误情况 Indicates either normal operation or some error condition
- 内存 Memory
  - 字节寻址存储数组 Byte-addressable storage array
  - “字”采用小端字节顺序存储 Words stored in little-endian byte order

Y86-64 Instruction Set 指令集 #1

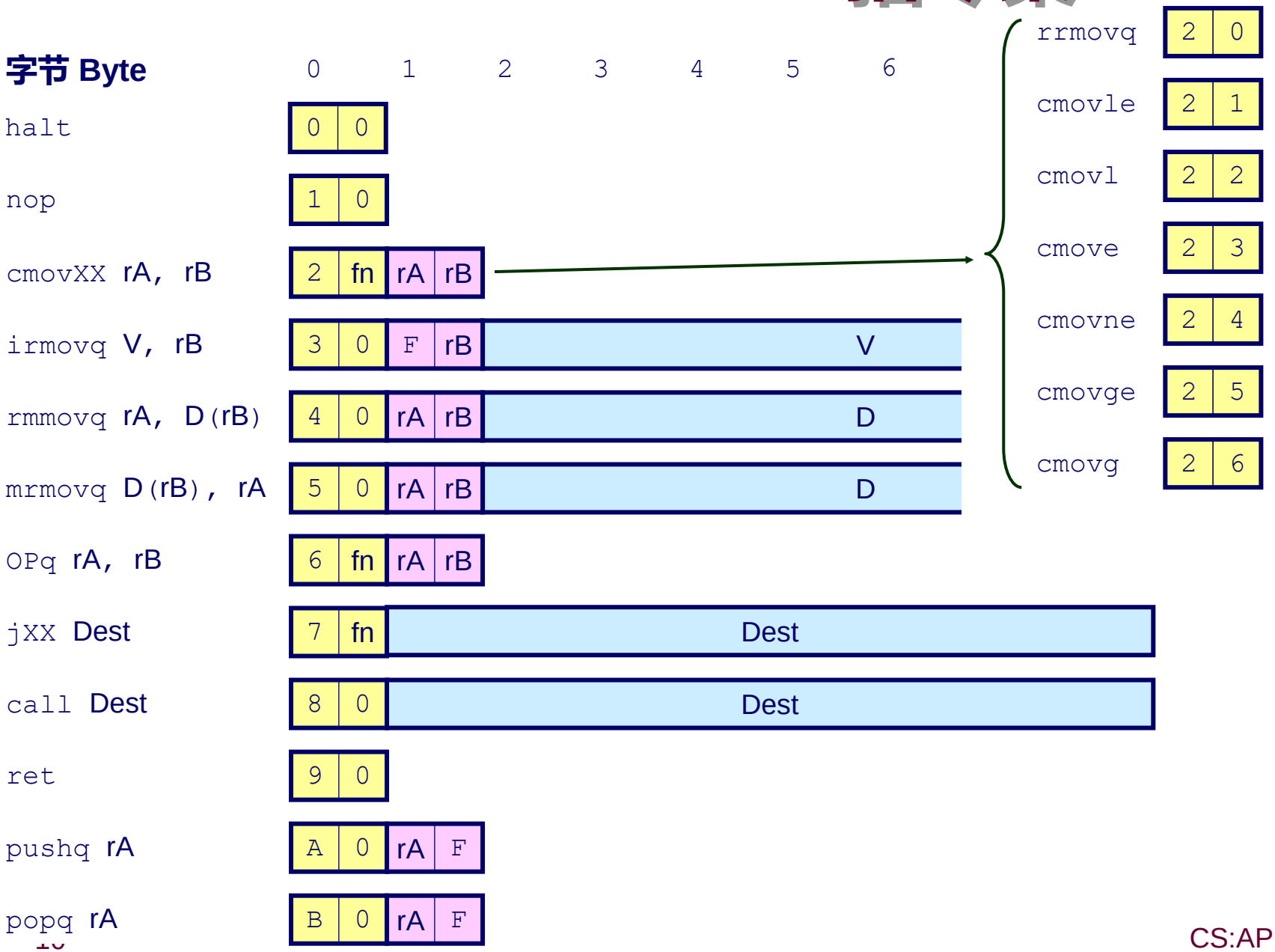


Y86-64 Instructions 指令

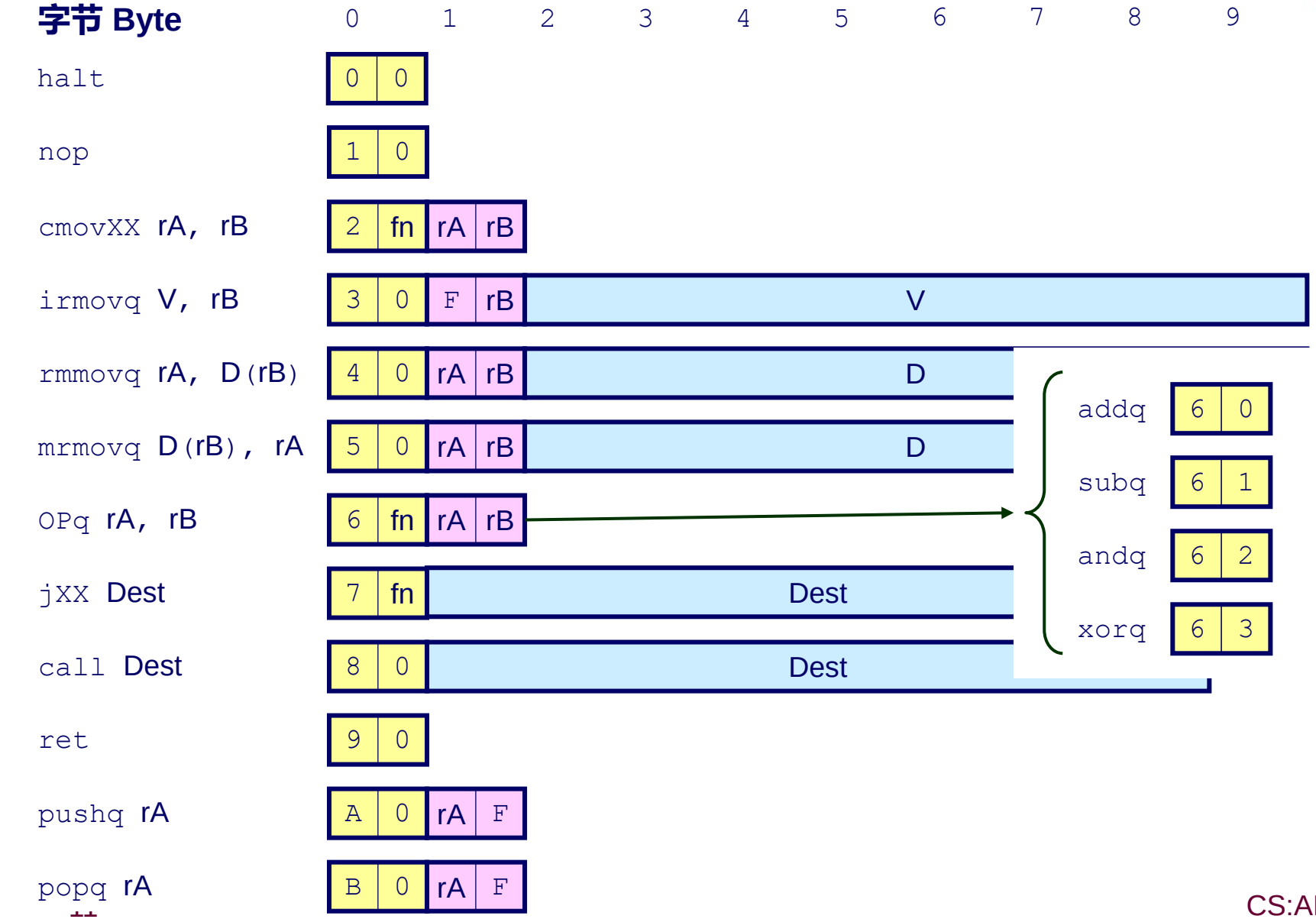
格式 Format

- 1-10字节信息从内存读出 1–10 bytes of information read from memory
  - 从第一个字节能够确定指令长度 Can determine instruction length from first byte
  - 与x86-64相比指令类型不是很多，而且编码更简单 Not as many instruction types, and simpler encoding than with x86-64
- 每次访问和修改一部分程序状态 Each accesses and modifies some part(s) of the program state

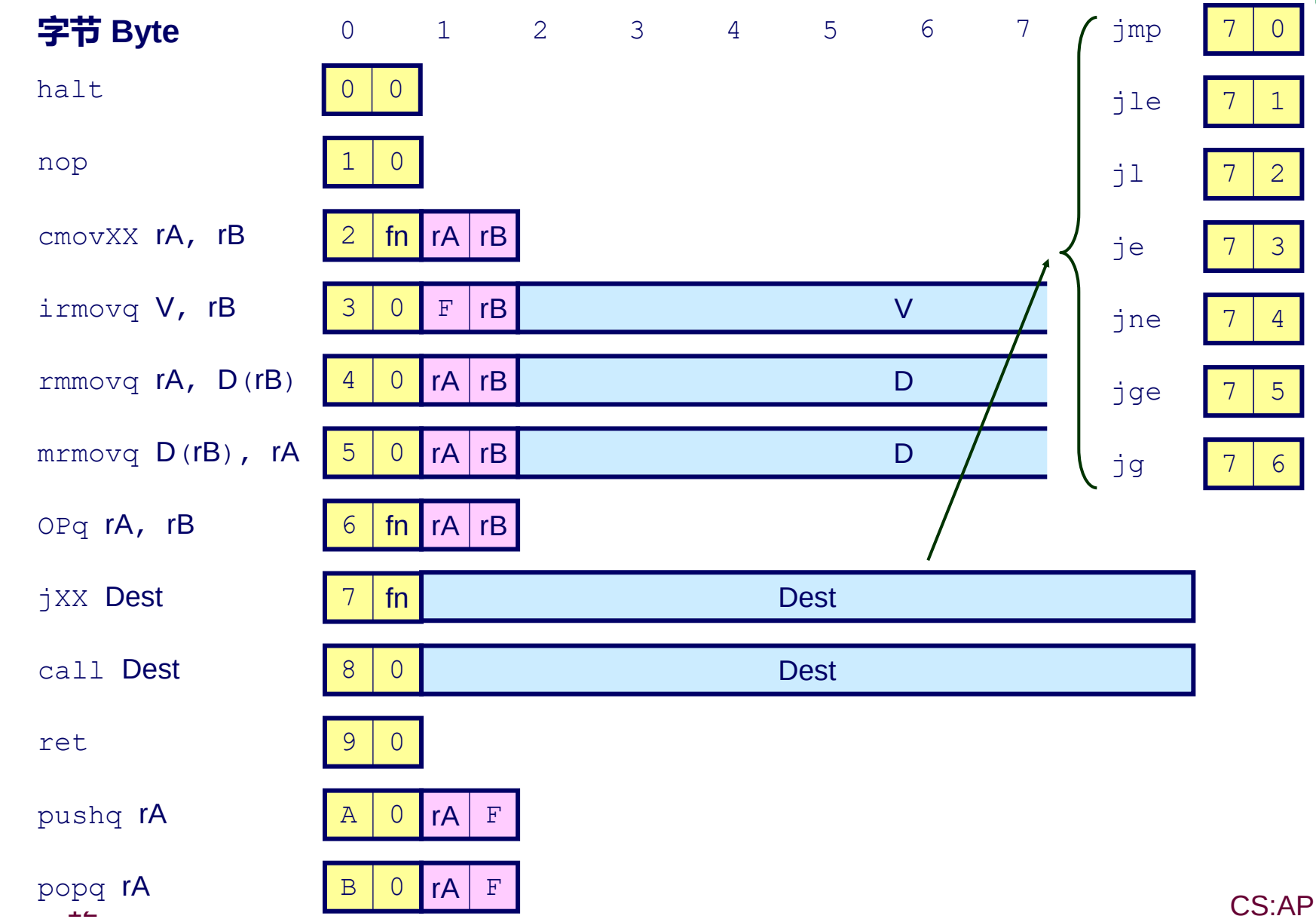
Y86-64 Instruction Set 指令集#2



Y86-64 Instruction Set 指令集#3



Y86-64 Instruction Set 指令集 #4



编码寄存器 Encoding Registers

每个寄存器有4位ID Each register has 4-bit ID

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

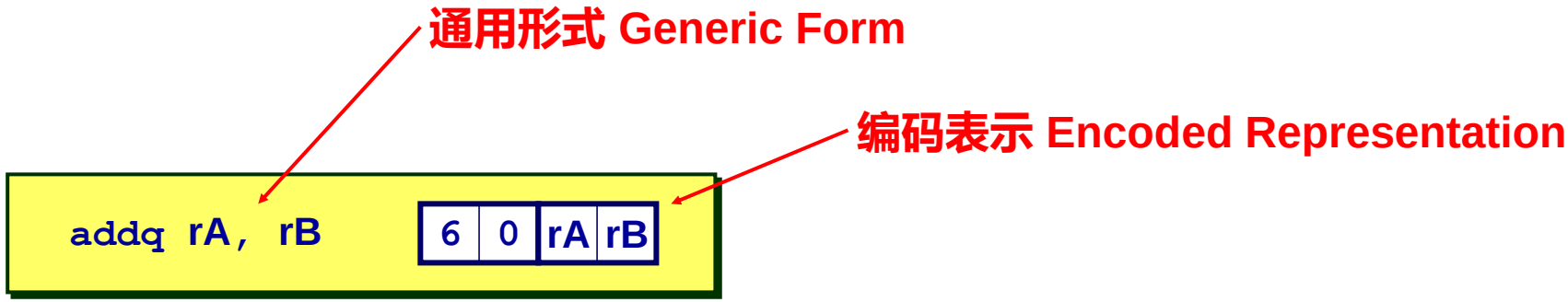
- 与x86-64编码相同 Same encoding as in x86-64

寄存器ID 15（0xF）指示“没有用到寄存器” Register ID 15 (0xF) indicates “no register”

- 在硬件设计多处将会用到 Will use this in our hardware design in multiple places

指令示例 Instruction Example

加法指令 Addition Instruction



- rB寄存器中的值加上rA寄存器中的值 Add value in register rA to that in register rB
  - 存储结果到rB寄存器 Store result in register rB
  - 注意Y86-64仅允许寄存器数据进行加法运算 Note that Y86-64 only allows addition to be applied to register data
- 根据结果设置条件码 Set condition codes based on result
- 例如 e.g., `addq %rax,%rsi` 编码 Encoding: `60 06`
- 两字节编码 Two-byte encoding
  - 第一个字节指明指令类型 First indicates instruction type
  - 第二字节给出源和目的寄存器 Second gives source and destination registers

算术和逻辑运算
Arithmetic and Logical Operations

指令代码 Instruction Code 功能码 Function Code

Add

Subtract (rA from rB)

And

Exclusive-Or

- 泛指为“OPq” Refer to generically as “OPq”
- 仅仅“功能码”编码不同 Encodings differ only by “function code”
  - 第一个指令字节低4位 Low-order 4 bits in first instruction word
- 设置条件码作为副作用 Set condition codes as side effect

传送操作 Move Operations

寄存器到寄存器 Register ➔ Register

`rrmovq` rA, rB

2	0

立即数到寄存器 Immediate ➔ Register

`irmovq` V, rB

3	0	F	rB	V

寄存器到内存 Register ➔ Memory

`rmmovq` rA, D(rB)

4	0	rA	rB	D

内存到寄存器 Memory ➔ Register

`mrmovq` D(rB), rA

5	0	rA	rB	D

- 类似x86-64传送指令 Like the x86-64 `movq` instruction
- 内存寻址格式更简单 Simpler format for memory addresses
- 给定不同名字保持区别 Give different names to keep them distinct

传送指令示例
Move Instruction Examples

x86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
编码 Encoding:	`30 82 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
编码 Encoding:	`20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
编码 Encoding:	`50 15 f4 ff ff ff ff ff ff ff`
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
编码 Encoding:	`40 64 1c 04 00 00 00 00 00 00`

条件传送指令 Conditional Move Instructions

Move Unconditionally 无条件

Move When Less or Equal 小于等于

Move When Less 小于

Move When Equal 等于

Move When Not Equal 不等

Move When Greater or Equal 大于等于

Move When Greater 大于

- 泛指为“cmovXX” Refer to generically as “`cmovXX`”
- 仅“功能码”编码不同 Encodings differ only by “function code”
- 根据条件码的值 Based on values of condition codes
- rrmovq指令的变种 Variants of `rrmovq` instruction
  - （有条件）复制值从源到目的寄存器 (Conditionally) copy value from source to destination register

跳转指令 Jump Instructions

跳转（条件）Jump (Conditionally)

- 泛指作“jXX” Refer to generically as “jxx”
- 仅“功能码”fn编码不同 Encodings differ only by “function code” fn
- 根据条件码的值 Based on values of condition codes
- 与x86-64处理器相同 Same as x86-64 counterparts
- 编码完整目的地址 Encode full destination address
  - 与x86-64中见到的PC相对寻址不同 Unlike PC-relative addressing seen in x86-64

跳转指令 Jump Instructions

Jump Unconditionally 无条件

`jmp` Dest	7	0	Dest

Jump When Less or Equal 小于等于

`jle` Dest	7	1	Dest

Jump When Less 小于

`jl` Dest	7	2	Dest

Jump When Equal 等于

`je` Dest	7	3	Dest

Jump When Not Equal 不等

`jne` Dest	7	4	Dest

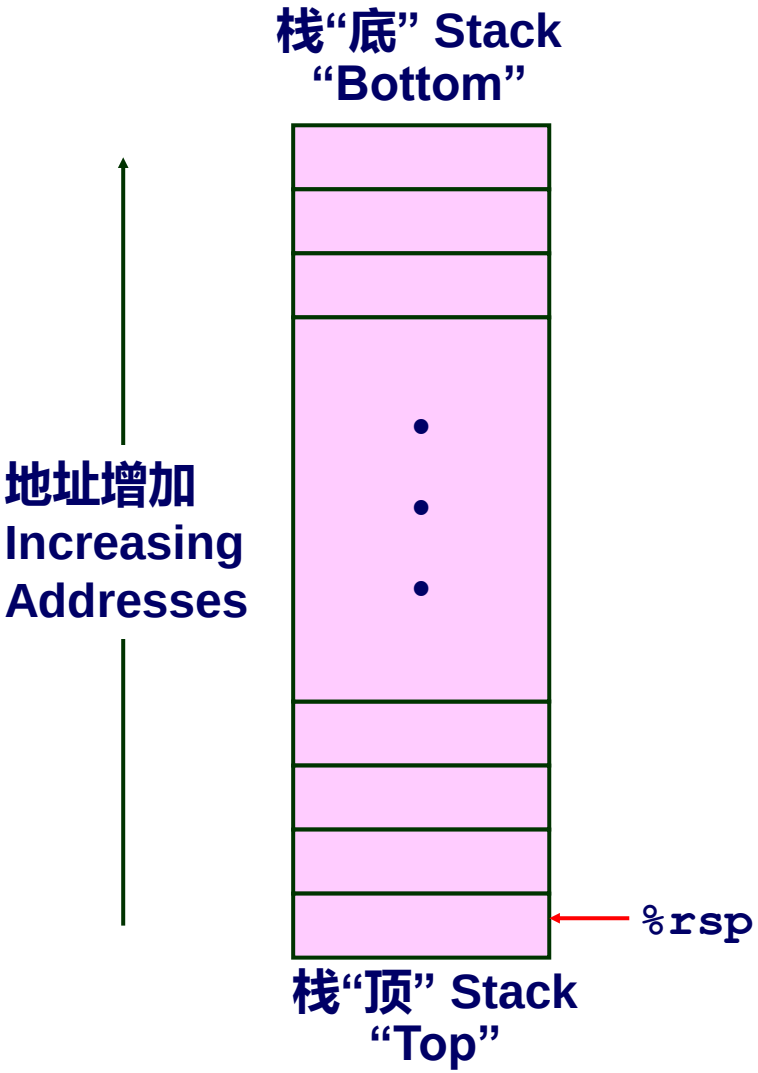
Jump When Greater or Equal 大于等于

`jge` Dest	7	5	Dest

Jump When Greater 大于

`jg` Dest	7	6	Dest

Y86-64程序栈 Y86-64 Program Stack



- 存储程序数据的内存区域 Region of memory holding program data
- 在Y86-64（和x86-64）中用于支持过程调用 Used in Y86-64 (and x86-64) for supporting procedure calls
- 栈顶由%rsp指示 Stack top indicated by `%rsp`
  - 栈顶元素的地址 Address of top stack element
- 栈向低地址方向生成 Stack grows toward lower addresses
  - 栈顶元素在栈的最低地址 Top element is at lowest address in the stack
  - 压栈时必须首先递减栈指针 When pushing, must first decrement stack pointer
  - 弹出后栈指针递增 After popping, increment stack pointer

栈操作 Stack Operations

`pushq rA`

- %rsp减8 Decrement `%rsp` by 8
- 将rA中的字存储到%rsp指向的内存单元 Store word from rA to memory at `%rsp`
- 类似 x86-64 Like x86-64

`popq rA`

- 从%rsp指向的内存读出字 Read word from memory at `%rsp`
- 存储到rA中 Save in rA
- %rsp加8 Increment `%rsp` by 8
- 类似x86-64 Like x86-64

子程序调用和返回
Subroutine Call and Return

`call Dest`

- 将下一条指令地址压入栈 Push address of next instruction onto stack
- 开始执行目的处指令 Start executing instructions at Dest
- 类似 x86-64 Like x86-64

`ret`

- 从栈弹出值 Pop value from stack
- 作为下条指令地址使用 Use as address for next instruction
- 类似x86-64 Like x86-64

杂项指令 Miscellaneous Instructions

- 不做任何事情 Don’t do anything

- 停止执行指令 Stop executing instructions
- x86-64有兼容指令，但是不能在用户模式使用 x86-64 has comparable instruction, but can’t execute it in user mode
- 我们用它来停止模拟器 We will use it to stop the simulator
- 这个编码确保程序运行到内存为零时会停机 Encoding ensures that program hitting memory initialized to zero will halt

状态码 Status Conditions

Mnemonic	Code
AOK	1

- 正常运行 Normal operation

Mnemonic	Code
HLT	2

- 遇到停机指令 Halt instruction encountered

Mnemonic	Code
ADR	3

- 遇到错误地址（指令或数据） Bad address (either instruction or data) encountered

Mnemonic	Code
INS	4

- 遇到非法指令 Invalid instruction encountered

预期的行为 Desired Behavior

- 如果AOK，则继续运行 If AOK, keep going
- 否则，停止程序执行 Otherwise, stop program execution

编写Y86-64代码 Writing Y86-64 Code

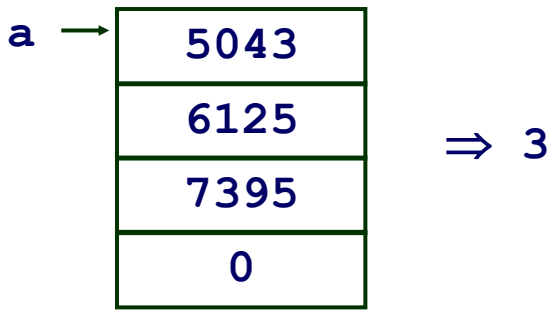
尝试尽可能使用C语言编译器 Try to Use C Compiler as Much as Possible

- 用C语言编写代码 Write code in C
- 编译成x86-64汇编 Compile for x86-64 with `gcc -Og -S`
- 移植成Y86-64汇编 Transliterate into Y86-64
- *现代编译器让这个工作变得更困难 Modern compilers make this more difficult*

代码示例 Coding Example

- 发现以空作结尾列表中元素数量 Find number of elements in null-terminated list

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



⇒ 3

Y86-64代码生成示例
Y86-64 Code Generation Example

首先尝试 First Try

- 编写典型的数组代码 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;
}
```

- 编译 Compile with `gcc -Og -S`

问题 Problem

- 在Y86-64中很难做数组索引 Hard to do array indexing on Y86-64
  - 因为没有缩放寻址方式 Since don't have scaled addressing modes

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

Y86-64代码生成示例#2
Y86-64 Code Generation Example #2

第二次尝试 Second Try

- 编写C语言代码模仿期望的Y86-64代码 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;
}
```

结果 Result

- 编译器生成和以前完全一样的代码 Compiler generates exact same code as before!
- 编译器将两个版本都转换成同样的中间格式 Compiler converts both versions into same intermediate form

Y86-64代码生成示例#3
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
```

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

Y86程序 Y86 Programs

```
int Sum(int *Start, int Count)
{
    int sum = 0;
    while (Count) {
        sum += *Start;
        Start++;
        Count--;
    }
    return sum;
}
```

Y86 Assembly

IA64 code

```
1 sum:
2     movl $0, %eax
3     jmp .L2
4  .L3:
5     addq (%rdi), %rax
6     addq $8, %rdi
7     subq $1, %rsi
8  .L2:
9     testq %rsi, %rsi
10    jne .L3
11    rep; ret
```

Y86 code

*int Sum(int *Start, int Count)*

```
1 sum:
2     irmovq $8,%r8
3     irmovq $1,%r9
4     xorq %rax,%rax
5     andq %rsi,%rsi
6     jmp test
7 loop:
8     mrmovq (%rdi),%r10
9     addq %r10,%rax
10    addq %r8,%rdi
11    subq %r9,%rsi
12 test:
13    jne loop
14    ret
```

Y86目标程序 Y86 Object Program

```
1   # Execution begins at address 0
2         .pos 0
3         irmovq    stack, %rsp       # Set up stack pointer
4         call      main              # Execute main program
5         halt                        # Terminate program
6
```

符号名 Symbolic Name

自动生成程序的初始部分 Init part of program generated automatically

汇编器伪指令 Assembler directives

- .pos 0, .pos 0x200
- .align

Y86目标程序 Y86 Object Program

```
7   # Array of 4 elements
8         .align 8
9   array:
10        .quad 0x000d000d000d
11        .quad 0x00c000c000c0
12        .quad 0x0b000b000b00
13        .quad 0xa000a000a000
14
```

数据区 Data area

- Array指示数组的起始 array denotes the start of an array
- 对齐到8字节边界 Aligned on 8-byte boundary

Y86目标程序 Y86 Object Program

```
15  main:
16      irmovq    array,%rdi
17      irmovq    $4,%rsi
18      call      sum                 # sum(array, 4)
19      ret
20
```

Y86目标程序 Y86 Object Program

```
21  # long sum(long *start, long count)
22  # start in %rdi, count in %rsi
23  sum:
24      irmovq    $8,%r8          # Constant 8
25      irmovq    $1,%r9          # Constant 1
26      xorq      %rax,%rax       # sum = 0
27      andq      %rsi,%rsi       # Set CC
28      jmp       test            # Goto test
```

Y86目标程序 Y86 Object Program

```
29  loop:
30      mrmovq (%rdi),%r10      # Get *start
31      addq %r10,%rax          # Add to sum
32      addq %r8,%rdi           # start++
33      subq %r9,%rsi           # count--. Set CC
34  test:
35      jne loop                # Stop when 0
36      ret                     # Return
37
38 # Stack starts here and grows to lower addresses
39 .pos 0x200
```

Y86目标程序 Y86 Object Program

```
38  # Stack starts here and grows to lower addresses
39      .pos 0x200
40  stack:
```

符号名 Symbolic Name

程序员必须自己写汇编代码 Programmers must write assembly codes themselves

- 包括管理内存 including manage the memory
  - 例如为数组和栈分配内存 such as allocate memory for array and stack
  - 以及避免内存覆盖 as well as avoid the memory overwritting

Y86-64样例程序结构#1
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:
```

- 程序起始地址为零 Program starts at address 0
- 必须设置栈 Must set up stack
  - 栈的位置 Where located
  - 栈指针值 Pointer values
  - 确保不会覆盖代码 Make sure don't overwrite code!
- 必须初始化数据 Must initialize data

Y86-64程序结构#2
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
```

- 程序起始地址为零 Program starts at address 0
- 必须设置栈 Must set up stack
- 必须初始化数据 Must initialize data
- 可以使用符号名 Can use symbolic names

Y86-64程序结构#3
Y86-64 Program Structure #3

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

设置对len的调用 Set up call to len

- 遵循x86-64过程规则 Follow x86-64 procedure conventions
- 数组地址作为参数传递 Push array address as argument

汇编Y86-64程序
Assembling Y86-64 Program

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

- 生成“目标代码”文件len.yo Generates “object code” file `len.yo`
  - 实际看起来像反汇编输出 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
```

模拟运行Y86-64程序
Simulating Y86-64 Program

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

- 指令集模拟器 Instruction set simulator
  - 计算每条指令对处理器状态的影响 Computes effect of each instruction on processor state
  - 打印从开始到现在的状态变化 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
```

指令集体系结构 #3
Instruction Set Architecture #3

ISA定义了处理器分类 ISA define the processor family

- 两个主要分类：Two main kind: RISC and CISC
  - RISC: SPARC, MIPS, PowerPC, ARM
  - CISC: X86 (or called IA32)

采用同样的ISA，有很多不同的处理器 Under same ISA, there are many different processors

- 来自不同的制造商 From different manufacturers
  - X86 from Intel, AMD and VIA
- 不同的型号 Different models
  - 8086, 80386, Pentium, atom, core i7

RISC vs. CISC

ISA

- 指令集体系结构 Instruction set architecture
  - 指令由特定处理器支持 Instructions supported by a particular processor
  - 其字节级编码 Their byte-level encodings

CISC

- 复杂指令集计算机 Complex instruction set computer

RISC

- 精简指令集计算机 Reduced instruction set computer

CISC

从最早的计算机开始 Involved from the earliest computers

大型主机和超级小型机 Mainframe and Minicomputers

- 早在上世纪80年代 By the early 1980s
- 其指令集增长得非常庞大 their instruction sets had grown quite large
  - 操作环形缓冲区 Manipulating circular buffers
  - 执行十进制运算 performing decimal arithmetic
  - 评估多项式 evaluating polynomials

微型计算机 Microcomputer

- 出现在上世纪70年代，有有限的指令集 Appeared in 1970s, had limited instruction sets
  - 受单芯片上晶体管数量的限制 constrained by number of transistors on a single chip
- 到上世纪80年代早期，按照这个路线增加其指令集 By the early 1980s, followed the path to increase their instruction sets

RISC



开发于上世纪80年代早期 Developed in the early 1980s

- 哲学 philosophy
  - 对于更简单的指令集格式能够生成高效代码 One are able to generate efficient code for a simpler form of instruction set
  - 复杂指令很难用编译器生成而且很少使用 Complex instructions are hard to generated with a compiler and seldom used

John Cocke (1925-2002)

- 1987 ACM Turing Award 图灵奖

David Patterson(UC Berkeley), John Hennessy (Stanford U)

- 2017 ACM Turing Award 图灵奖

RISC

IBM	Power
IBM and Motorola	PowerPC
Sun Microsystems	SPARC
Digital Equipment Corporation	Alpha
Hewlett Pack	Pa-risc
MIPS Technologies	MIPS
Acorn Computers Ltd	ARM(Acorn RISC Machine)

RISC vs. CISC

CICS	早期RISC Early RISC
指令数很多 A large number of instructions	指令较少 Many fewer instructions (<100)
有些指令执行时间较长 Some instructions with long execution times	没有执行时间很长的指令 No instruction with a long execution time
可变长编码 Variable-length encodings. (x86-64 1~15 bytes)	固定长度编码 Fixed-length encodings
多种格式寻址操作数 Multiple formats for specifying operands	简单的寻址格式 Simple addressing formats
算术和逻辑运算可以应用到内存和寄存器操作数 Arithmetic and logical operations can be applied to both memory and register operands	算术和逻辑运算只能用于寄存器操作数，装载/存储 体系结构 Arithmetic and logical operations only use register operands. *load/store Architecture*

RISC vs. CISC

CICS	早期RISC Early RISC
对机器级程序来说实现细节是不可见的 Implementation artifacts hidden from machine level programs	机器级程序来说实现细节是可见的 Implementation artifacts exposed to machine level programs
条件码 Condition codes	显式测试指令将测试结果存储在普通寄存器中以用于条件评估 explicit test instructions store the test results in normal registers for use in conditional evaluation
过程链接通过栈实现 Stack-intensive procedure linkage	过程参数传递和返回值采用寄存器实现 Registers are used for procedure arguments and return addresses

Y86-64

Y86-64指令集 The Y86-64 instruction set

- 包括了CISC和RISC二者的属性 Includes attributes of both CISC and RISC
- 可以看成采用CISC指令集（x86-64）同时应用RISC一些原则进行了简化 Can be viewed as taking a CISC instruction set (x86-64) and simplifying it by applying some of the principles of RISC

在CISC这方面 On the CISC side

- 条件码、可变长指令 condition codes, variable-length instructions
- 使用栈存储返回地址 uses the stack to store return addresses.

在RISC方面 On the RISC side,

- 装载/存储 体系结构 a load/store architecture
- 规整的指令编码 a regular instruction encoding
- 通过寄存器传递过程参数 passes procedure arguments through registers

复杂指令集 CISC Instruction Sets

- 复杂指令集计算机 Complex Instruction Set Computer
- IA32就是例子 IA32 is example

面向栈的指令集 Stack-oriented instruction set

- 使用栈传递参数，保存程序计数器 Use stack to pass arguments, save program counter
- 显式的入栈和出栈指令 Explicit push and pop instructions

运算指令能够访问内存 Arithmetic instructions can access memory

- `addq %rax, 12(%rbx,%rcx,8)`
  - 需要内存读和写 requires memory read and write
  - 复杂的地址计算方式 Complex address calculation

复杂指令集 CISC Instruction Sets

条件码 Condition codes

- 作为算术和逻辑运算指令的副作用设置条件码 Set as side effect of arithmetic and logical instructions

哲学 Philosophy

- 增加指令执行“典型的”编程任务 Add instructions to perform “typical” programming tasks

精简指令集 RISC Instruction Sets

- 精简指令集计算机 Reduced Instruction Set Computer
- IBM的内部项目，后来由Hennessy(斯坦福)和Patterson (伯克利)大规模推广 Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

指令更少，更简单 Fewer, simpler instructions

- 为了完成任务可能需要更多指令 Might take more to get given task done
- 可以执行指令用小且快速的硬件 Can execute them with small and fast hardware

面向寄存器的指令集 Register-oriented instruction set

- 有更多的寄存器（典型32个） Many more (typically 32) registers
- 用于参数传递、返回指针和临时变量存储 Use for arguments, return pointer, temporaries

精简指令集 RISC Instruction Sets

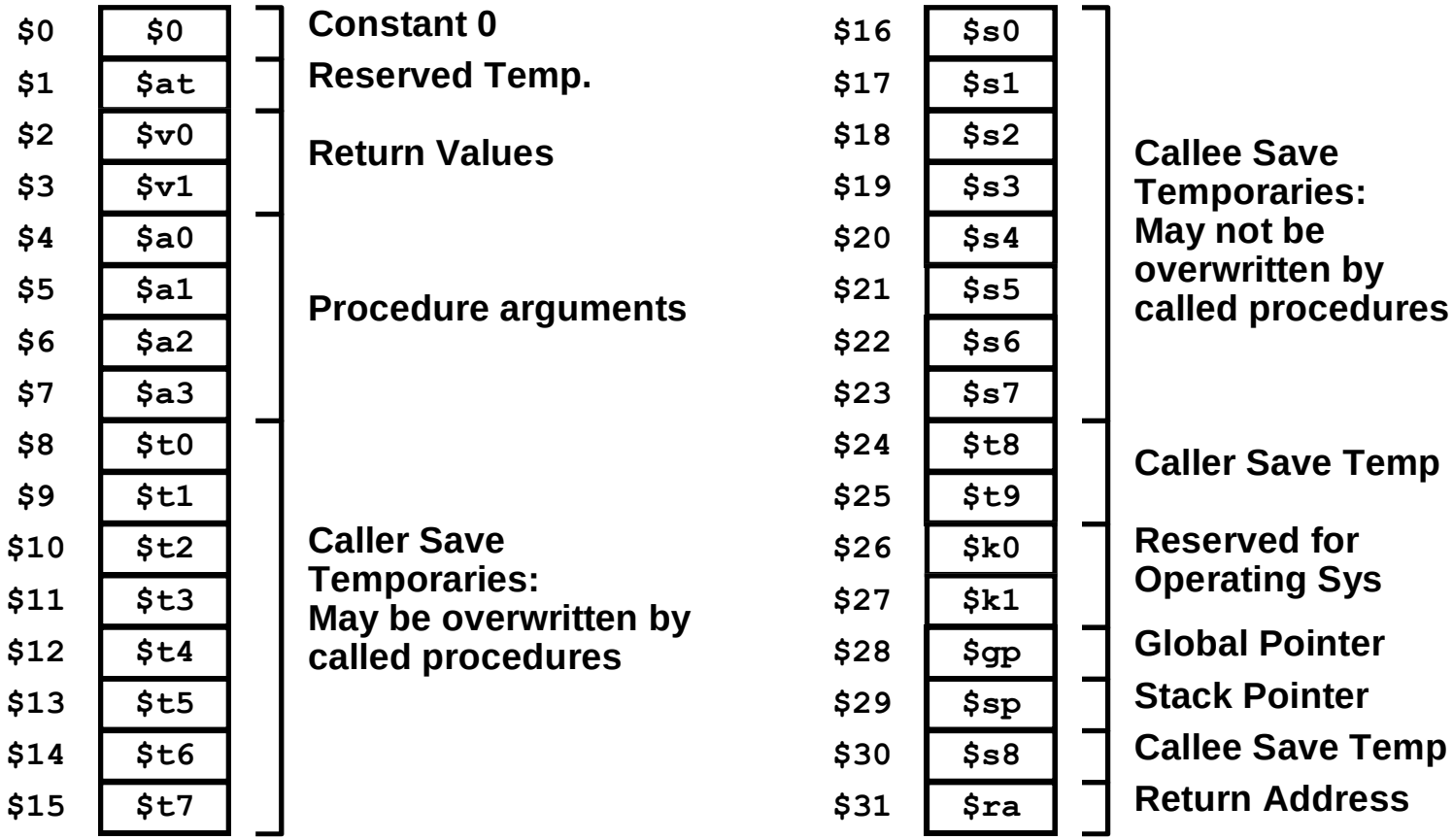
只有load和store指令能够访问内存 Only load and store instructions can access memory

- 类似于Y86-64的`mrmovq`和`rmmovq`指令 Similar to Y86-64 `mrmovq` and `rmmovq`

没有条件码 No Condition codes

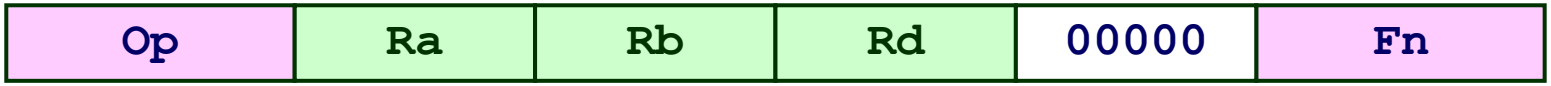
- 测试指令返回0/1在寄存器中 Test instructions return 0/1 in register

MIPS寄存器 MIPS Registers



MIPS指令示例
MIPS Instruction Examples

R-R

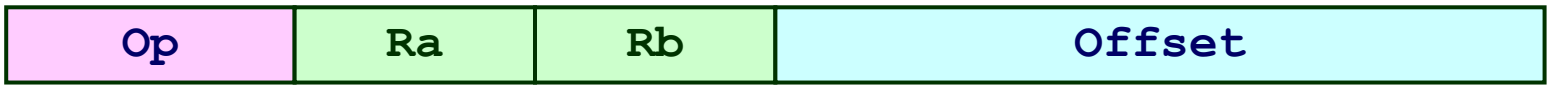



```
addu $3,$2,$1          # Register add: $3 = $2+$1
```

R-I

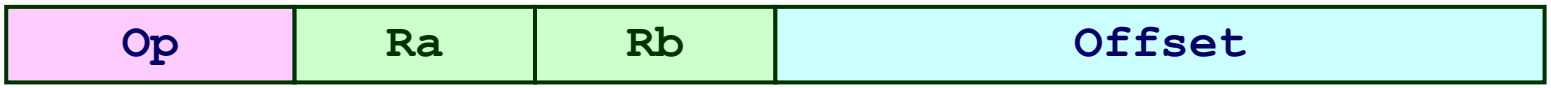

```
addu $3,$2, 3145       # Immediate add: $3 = $2+3145
sll $3,$2,2            # Shift left: $3 = $2 << 2
```

Branch




```
beq $3,$2,dest         # Branch when $3 = $2
```

Load/Store




```
lw $3,16($2)           # Load Word: $3 = M[$2+16]
sw $3,16($2)           # Store Word: M[$2+16] = $3
```

CISC vs. RISC

原始争论 Original Debate

- 争论十分激烈 Strong opinions!
- CISC支持者-编译器比较方便，很少的代码字节 CISC proponents---easy for compiler, fewer code bytes
- RISC支持者-优化编译器更佳，用简单芯片设计可以使运行更快 RISC proponents---better for optimizing compilers, can make run fast with simple chip design

CISC vs. RISC

目前状态 Current Status

- 对于台式机处理器，ISA选择并非技术问题 For desktop processors, choice of ISA not a technical issue
  - 有足够的硬件可以使任何事情运行的更快 With enough hardware, can make anything run fast
  - 代码兼容更重要 Code compatibility more important
- x86-64采纳了很多RISC功能 x86-64 adopted many RISC features
  - 更多的寄存器；使用它们作参数传递 More registers; use them for argument passing
- 对于嵌入式处理器，RISC更具优势 For embedded processors, RISC makes sense
  - 更小、更便宜、功耗更低 Smaller, cheaper, less power
  - 大部分手机使用ARM处理器 Most cell phones use ARM processor

小结 Summary

Y86-64指令集体系结构 Y86-64 Instruction Set Architecture

- 与x86-64类似的状态和指令 Similar state and instructions as x86-64
- 更简单的编码 Simpler encodings
- 某些方面介于CISC和RISC之间 Somewhere between CISC and RISC

ISA设计有多重要？ How Important is ISA Design?

- 现在比以前要低一些 Less now than before
  - 有足够的硬件，几乎可以使任何事情都变得更快 With enough hardware, can make almost anything go fast