Instruction Set Architecture(ISA)

- Computer Architecture = ISA + microarchitecture (Ch. 4)
- ISA = **Register organization**

  + **Addressing mode**

  + **Instruction set**

  + Instruction format (Ch. 4)

  + ...
- Intel x86 Architecture (third gen: IA-32, eighth gen: x86-64)
- CISC vs RISC

CISC:

Instruction can reference different operand types

– Immediate, register, memory

• Arithmetic operations can read/write memory

• Memory reference can involve complex computation

– Rb + S*Ri + D

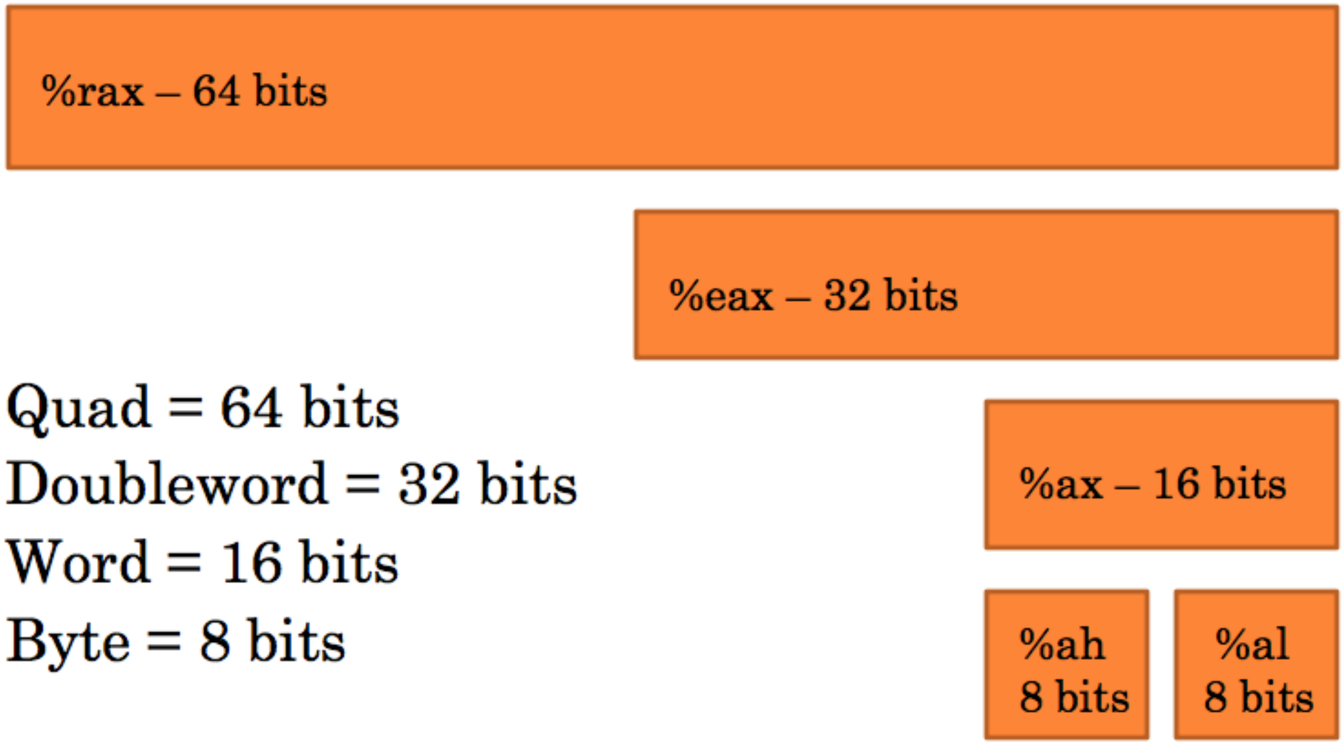
– Useful for arithmetic expressions, too

• Instructions can have varying lengths

– IA32 instructions can range from 1 to 15 bytes

Registers (what and where)

- Some place in CPU that store bits.
- Not on the stack or in main memory. (where is stack?)
- (Fig 3.1, Fig 3.2)



- Quad = 64 bits
- Doubleword = 32 bits
- Word = 16 bits
- Byte = 8 bits

These are all parts of the same register

Register Organization

- GP registers: %rax, %rbx, %rcx, %rdx, %r8~%r15
  - %rax: Accumulator, return value
  - %rbx: Base index (for use with arrays)
  - %rcx: Counter (for use with loops and strings), integer argument 0
  - %rdx: Data, integer argument 1
  - %r8: integer argument 2, %r9: integer argument 3
- Index registers:
  - %rsi: *Source index* for string operations.
  - %rdi: *Destination index* for string operations.
- Pointer registers:
  - %rsp: Stack pointer for top address of the stack.
  - %rbp: Stack base pointer for holding the address of the current stack frame.
- Instruction pointer register:
  - %rip: Instruction pointer. Holds the program counter, the current instruction address.
- Segment registers (location of Code, Stack, Data, etc)
- Flag registers, FP registers, SSE registers, etc

Compiling Into Assembly

```
int sum(int x, int y)
{
int t = x+y;
return t;
}
```

```
_sum:
Push %ebp
mov %esp,%ebp
mov 12(%ebp),%eax
add 8(%ebp),%eax
mov %ebp,%esp
pop %ebp
ret
```

Machine Instruction Example

Int t= x+y

C Code

– Add two signed integers

•**Assembly**

add 8(%ebp),%eax

– Add two 4−byte integers

• “Long” words in GCC parlance

•Same instruction whether signed or unsigned

– Operands:

x: Register **%eax**

y: Memory M[**%ebp+8]**

t: Register **%eax**

– Return function value in **%eax**

•Object Code

0x401046: 03 45 08

– 3−byte instruction

– Stored at address **0x401046**

Moving Data

movl Source,Dest:

– Move 4-byte (“long”) word

– Lots of these in typical code

• **Operand Types**

– Immediate: Constant integer data

• **Like C constant, but prefixed with ‘$’**

• E.g., $0x400, $-533

• Encoded with 1, 2, or 4 bytes

– Register: One of 8 integer registers

• But %esp and %ebp reserved for special use

• Others have special uses for particular instructions

– **Memory: 4 consecutive bytes of memory**

• Various “address modes”

Data Accessing (operand specifier)

- (Fig 3.3)
- Immediate: $Imm -> Imm
- Register: E -> R[E]
- Memory: Imm(Eb, Ei, s) = M[Imm + R[Eb] + R[Ei] * s]
  - s must be 1, 2, 4, 8, why?

- (Fig 3.4)
- Data movement instructions
  - (extension qualifiers: s,z; size qualifiers: b, l, w)
  - mov: NO mem-to-mem (2 controllers? requires DMA!)
  - push: push stack
  - pop: pop stack
  - lea: load effective address, computes not only addresses

Data Accessing (cont'd)

- Most of the time parenthesis means dereference
  - (%eax): Contents of memory at address stored, %eax
  - 4(%ebx, %ecx, 8): Contents of memory stored at address, %ebx + 8*%ecx + 4
- Sometimes parenthesis are used just for addressing
  - leal (%ebx, %ecx, 8), destination: Take only the values %ebx + 8*%ecx
    - Does not dereference, uses the calculated value directly
  - jmpq *0x402660(,%rax,8)
    - The * does the dereference

Some Arithmetic Operations

Format Computation

- Two Operand Instructions

addl Src,Dest Dest = Dest + Src

subl Src,Dest Dest = Dest - Src

imull Src,Dest Dest = Dest * Src (difference between Signed and Unsigned: high-half of result)

sall Src,Dest Dest = Dest << Src Also called shll

sarl Src,Dest Dest = Dest >> Src Arithmetic

shrl Src,Dest Dest = Dest >> Src Logical

xorl Src,Dest Dest = Dest ^ Src

andl Src,Dest Dest = Dest & Src

orl Src,Dest Dest = Dest | Src

One Operand Instructions

incl Dest Dest = Dest + 1

decl Dest Dest = Dest - 1

negl Dest Dest = - Dest

notl Dest Dest = ~ Dest

Condition Codes

Single Bit Registers

– CF Carry Flag SF Sign Flag

– ZF Zero Flag OF Overflow Flag

- Implicitly Set By Arithmetic Operations

– addl Src,Dest

– C analog: t = a + b

– CF set if carry out from most significant bit

- Used to detect unsigned overflow

– ZF set if t == 0

– SF set if t < 0

– OF set if two's complement overflow

- (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

- Not Set by leal instruction

Setting Condition Codes

Explicit Setting by Compare Instruction

- cmpl Src2,Src1
- cmpl b,a like computing a−b without setting destination
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if a == b
- SF set if (a−b) < 0
- OF set if two's complement overflow
  - (a>0 && b<0 && (a−b)<0) || (a<0 && b>0 && (a−b)>0)

Implementing Loops

- IA32
  - All loops translated into form based on “do‑while”
- x86‑64
  - Also make use of “jump to middle”
- Why the difference
  - IA32 compiler developed for machine where all operations costly
  - x86‑64 compiler developed for machine where unconditional branches incur (almost) no overhead

Switch Statement Example

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

- Multiple case labels
  - Here: 5, 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

What range of values takes default?

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	Return value

Note that **`w`** not initialized here

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

Indirect jump

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # Use default
    jmp     *.L4(,%rdi,8)     # goto *JTab[x]
```

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at `.L4`

- **Jumping**
  - **Direct:** `jmp .L8`
  - Jump target is denoted by label `.L8`

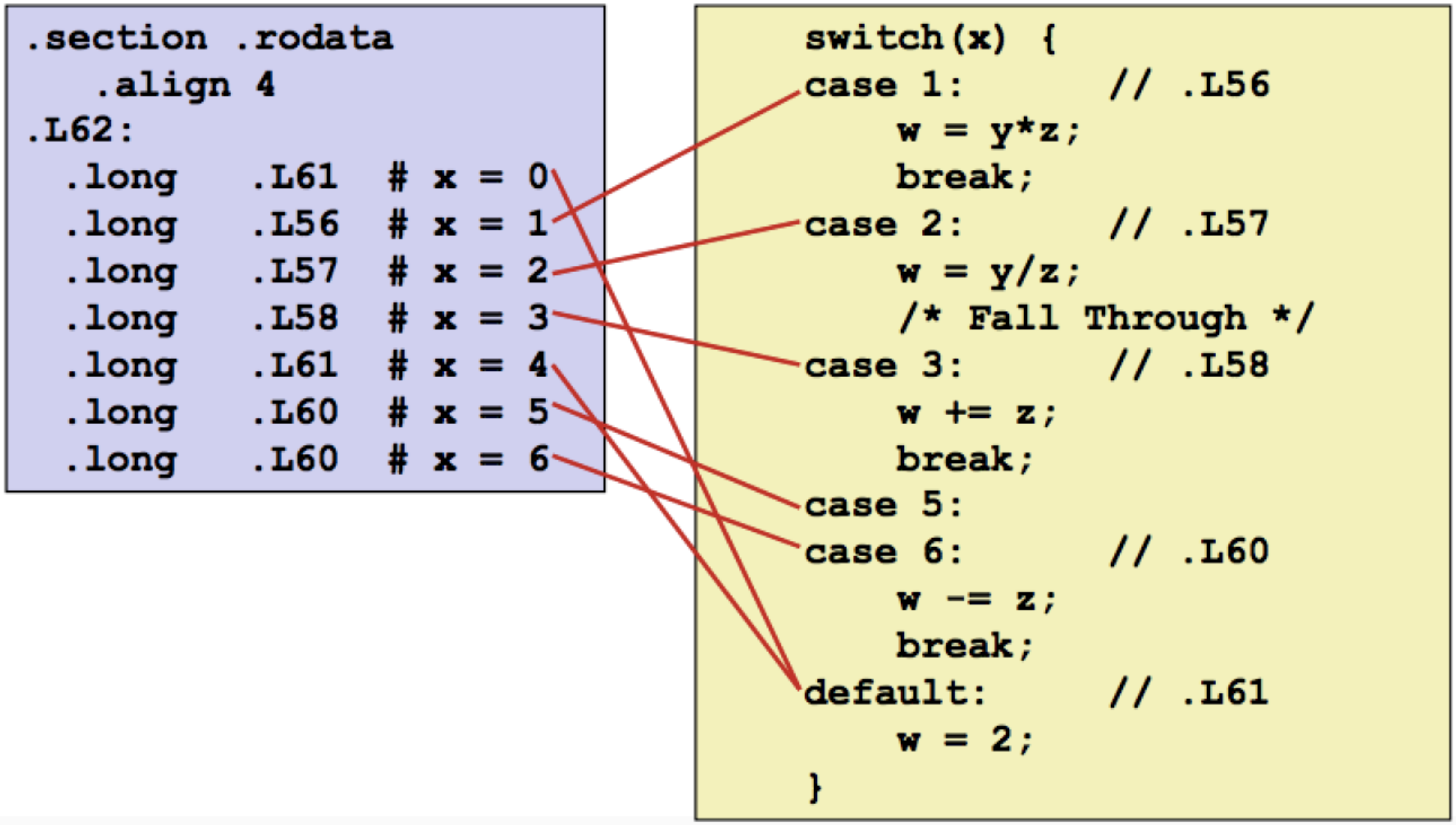
  - **Indirect:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \le x \le 6$

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Jump Table

Jump table



```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
case 5:
case 6:       // .L60
    w -= z;
    break;
default:      // .L61
    w = 2;
}
```