x86 Basics

Translation tools: C -> assembly <-> machine code

x86 registers, data movement instructions, memory addressing, arithmetic instructions

CSAPP book is highly useful and well-aligned with class for the remainder of the course.

https://cs.wellesley.edu/~cs240/

Turning C into Actual Machine Code

void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}

Generated x86 Assembly Code

sum:
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    retq

Real-world security & performance implications

Machine Instruction Example

C Code
Store value t where indicated by dest

*dest = t;

Assembly Code
Move 8-byte value to memory
t: Register %rsi
dest: Register %rdx
*dest: Memory M[rdx]

Object Code
3-byte instruction encoding
Stored at address 0x400539

Program, Application
Compiler/Interpreter
Instruction Set Architecture
Operating System
Microarchitecture
Digital Logic
Devices (transistors, etc.)
Solid-State Physics
Disassembling Object Code

Disassembled by `objdump -d sum`

```
0000000000400536 <sumstore>:
 400536:  48 01 fe  add  %rdi,%rsi
 400539:  48 89 32  mov  %rsi,(%rdx)
 40053c:  c3      retq
```

Disassembler

Object Disassembled by GDB

```
Object
0x0000000000400536:
 0x048  0x00800000000000000400536 <+0>: add %rdi,%rsi
 0x010  0x00000000000000000400539 <+3>: mov %rsi,(%rdx)
 0x0e0  0x0000000000000000040053c <+6>: retq

Disassembler
$ gdb sum
(gdb) disassemble sumstore
(disassemble function)
(gdb) x/7b sum
(examine the 13 bytes starting at sum)
```

CISC vs. RISC

x86: real ISA, widespread

CISC: maximalism
Complex Instruction Set Computer
Many instructions, specialized.
Variable-size encoding, complex/slow decode.
Gradual accumulation over time.
Original goal:
• humans program in assembly
• or simple compilers generate assembly by template
• hardware supports many patterns as single instructions
• fewer instructions per SLOC
Usually fewer registers.
We will stick to a small subset.

RISC: minimalism
Reduced Instruction Set Computer
Few instructions, general.
Regular encoding, simple/fast decode.
1980s+ reaction to bloated ISAs.
Original goal:
• humans use high-level languages
• smart compilers generate highly optimized assembly
• hardware supports fast basic instructions
• more instructions per SLOC
Usually many registers.

a brief history of x86

<table>
<thead>
<tr>
<th>Word Size</th>
<th>ISA</th>
<th>First</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8086</td>
<td>Intel 8086</td>
<td>1978</td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 16-bit processor. Basis for IBM PC &amp; DOS 1MB address space</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>IA32</td>
<td>Intel 386</td>
<td>1985</td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 32-bit ISA. Flat addressing, improved OS support</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>x86-64</td>
<td>AMD Opteron</td>
<td>2003*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slow AMD/Intel conversion, slow adoption. *Not actually x86-64 until few years later. Mainstream only after ~10 years.</td>
<td></td>
</tr>
</tbody>
</table>

Since 2016: most laptops, desktops, servers.

240 now: 64

ISA View
x86-64 registers

- 16 named registers
- Each 64 bits (8 bytes)

x86-64 registers: function arguments and return value

Mnemonic:

Diana's silk dress costs $89

Arguments 7 and above are passed via stack, not in registers.

Data movement instructions

1. Data movement between memory and register
   Load data from memory into register
   \%reg ← Mem[address]
   Store register data into memory
   Mem[address] ← \%reg

2. Arithmetic/logic on register or memory data
   \( c = a + b; \quad z = x \ll y; \quad i = h \& g; \)

3. Comparisons and Control flow to choose next instruction
   Unconditional jumps to/from procedures
   Conditional branches

Mnemonic:

mov Source, Dest
"copy the contents of source operand into dest operand"

data size is one of \( b, w, l, q \)
movq: move 8-byte "quad word"
movl: move 4-byte "long word"
movw: move 2-byte "word"
movb: move 1-byte "byte"

Source/Dest operand types:

Immediate: Literal integer data
Examples: 0x400, 533
Register: One of 16 registers
Examples: \%rax, \%rdx
Memory: Consecutive bytes in memory, at address held by register
Direct addressing: \( (\%rax) \)
With displacement/offset: \( B(\%rsp) \)

Historical terms based on the 16-bit days, not the current machine word size (64 bits)
Cannot do memory-memory transfer with a single instruction. How would you do it?

\textbf{mov\ Operand Combinations}

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Imm}</td>
<td>\text{Reg}</td>
<td>\text{memq} $0x4,%rax</td>
<td>a = 0x4;</td>
</tr>
<tr>
<td>\text{Mem}</td>
<td>\text{Reg}</td>
<td>\text{memq} $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>\text{Reg}</td>
<td>\text{Reg}</td>
<td>\text{memq} %rax,%rdx</td>
<td>d = a;</td>
</tr>
<tr>
<td>\text{Mem}</td>
<td>\text{Reg}</td>
<td>\text{memq} (memory address)</td>
<td>d = *p;</td>
</tr>
</tbody>
</table>

Pointers and Memory Addressing

void swap(long* xp, long* yp){
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

\textbf{Memory Addressing Modes}

\textbf{Indirect (R)} \quad \text{Mem}[\text{Reg}[R]]

Register \text{R} specifies memory address: \text{movq} (%rcx), %rax

\textbf{Displacement D(R)} \quad \text{Mem}[\text{Reg}[R]+D]

Register \text{R} specifies base memory address (e.g. base of an object)
Displacement \text{D} specifies literal \textit{offset} (e.g. a field in the object)
\text{movq} %rdx, 8(%rsi)

\textbf{General Form: D(Rb,Ri,S)} \quad \text{Mem}[\text{Reg}[\text{Rb}]+S*\text{Reg}[\text{Ri}]+D]

\text{D:} \quad \text{Literal “displacement” value represented in 1, 2, or 4 bytes}
\text{Rb:} \quad \text{Base register: Any register}
\text{Ri:} \quad \text{Index register: Any except \%rsp}
\text{S:} \quad \text{Scale: 1, 2, 4, or 8}

Pointers and Memory Addressing
Pointers and Memory Addressing

void swap(long* xp, long* yp){
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

*rdi = xp
*rsi = yp
%rax = t0
%rdx = t1

Registers

Memory

Address Computation Examples

void swap(long* xp, long* yp){
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

movq (%rdi),%rax
movq (%rsi),%rdx
movq %rdx,(%rdi)
movq %rax,(%rsi)
retq

%rdi
%rsi
%rax
%rdx

Address Computation

<table>
<thead>
<tr>
<th>Register contents</th>
<th>Address Computation</th>
<th>Address Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx 0x0f00</td>
<td>0x8(%rdx) + 0x0f00</td>
<td>0x8(,%rdx)</td>
</tr>
<tr>
<td>%rcx 0x100</td>
<td>0x8(%rdx) + 0x100</td>
<td>(%rdx,%rcx)</td>
</tr>
<tr>
<td></td>
<td>0x8(%rdx,4)</td>
<td>0x80(,%rdx,2)</td>
</tr>
</tbody>
</table>

General Addressing Modes

D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]

Special Cases:

- Implicitly: (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]
- D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]
- (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]
### Address Computation Examples

#### General Addressing Modes

- **Register**: \( \%rdx \), \( \%rcx \)
- **Implicitly**: \( (Rb, Ri) \)
- **Special Cases**:
  - \( \%rdx \): \( 0xf000 + 0x100 \times 1 \)
  - \( \%rdx, \%rcx, 4 \): \( 0xf000 + 0x100 \times 4 \)
  - \( 0x80(\%rdx, 2) \): \( 0x80 + 0x0 + 0xf000 \times 2 \)

#### Address Expression

<table>
<thead>
<tr>
<th>Address Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0x8(%rdx) )</td>
<td>( 0x8 + 0xf000 )</td>
<td>( 0xf008 )</td>
</tr>
<tr>
<td>( (%rdx, %rcx) )</td>
<td>( 0xf000 + 0x100 \times 1 )</td>
<td>( 0x100 )</td>
</tr>
<tr>
<td>( (%rdx, %rcx, 4) )</td>
<td>( 0xf000 + 0x100 \times 4 )</td>
<td>( 0xf400 )</td>
</tr>
<tr>
<td>( 0x80(%rdx, 2) )</td>
<td>( 0x80 + 0x0 + 0xf000 \times 2 )</td>
<td>( 0x1e080 )</td>
</tr>
</tbody>
</table>

#### Example

**Register contents**
- \( \%rdx \): 0xf000
- \( \%rcx \): 0x100

**Address**
- \( 0xf008 \)
- \( 0x100 \)
- \( 0xf400 \)
- \( 0x1e080 \)

### Compute address given by this addressing mode expression and store it here.

**leaq** *Src, *Dest

**Does not access memory**

**Uses**:
- “address of” “Lovely Efficient Arithmetic”
- \( p = \&x[i] \); \( x + k \times i \), where \( k = 1, 2, 4, \) or 8

**leaq vs. movq**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( %rax ) 0x110</td>
<td>0x400</td>
<td>0x120</td>
<td>leaq (%rdx, %rcx, 4), %rax movq (%rdx, %rcx, 4), %rbx</td>
</tr>
<tr>
<td>( %rbx ) 0xf</td>
<td>0x118</td>
<td>0x118</td>
<td>leaq (%rdx, %rcx, 4), %rbx movq (%rdx, %rcx, 4), %rbx</td>
</tr>
<tr>
<td>( %rcx ) 0x4</td>
<td>0x8</td>
<td>0x8</td>
<td>leaq (%rdx, %rdi), %rbx movq (%rdx, %rdi), %rsi</td>
</tr>
<tr>
<td>( %rdx ) 0x100</td>
<td>0x100</td>
<td>0x100</td>
<td>leaq (%rdx, %rdi), %rsi</td>
</tr>
<tr>
<td>( %rdi ) 0x1</td>
<td>0x1</td>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>( %rsi ) 0x100</td>
<td>0x100</td>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

### Compute address given by this addressing mode expression and store it here.

**leaq** *Src, *Dest

**Does not access memory**

**Uses**:
- “address of” “Lovely Efficient Arithmetic”
- \( p = \&x[i] \); \( x + k \times i \), where \( k = 1, 2, 4, \) or 8

**leaq vs. movq**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( %rax ) 0x110</td>
<td>0x400</td>
<td>0x120</td>
<td>leaq (%rdx, %rcx, 4), %rax movq (%rdx, %rcx, 4), %rbx</td>
</tr>
<tr>
<td>( %rbx ) 0xf</td>
<td>0x118</td>
<td>0x118</td>
<td>leaq (%rdx, %rcx, 4), %rbx movq (%rdx, %rcx, 4), %rbx</td>
</tr>
<tr>
<td>( %rcx ) 0x4</td>
<td>0x8</td>
<td>0x8</td>
<td>leaq (%rdx, %rdi), %rbx movq (%rdx, %rdi), %rsi</td>
</tr>
<tr>
<td>( %rdx ) 0x100</td>
<td>0x100</td>
<td>0x100</td>
<td>leaq (%rdx, %rdi), %rsi</td>
</tr>
<tr>
<td>( %rdi ) 0x1</td>
<td>0x1</td>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>( %rsi ) 0x100</td>
<td>0x100</td>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>
leaq Src, Dest

DOES NOT ACCESS MEMORY

Uses: “address of” “Lovely Efficient Arithmetic”

\[ p = \&x[i]; \quad x + k*i, \text{ where } k = 1, 2, 4, \text{ or } 8 \]

### leaq vs. movq

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax 0x110</td>
<td>0x120</td>
<td>leaq (%rdx,%rcx,4), %rax</td>
<td></td>
</tr>
<tr>
<td>%rbx 0x8</td>
<td>0x118</td>
<td>movq (%rdx,%rcx,4), %rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx 0x4</td>
<td>0x10</td>
<td>leaq (%rdx), %rdi</td>
<td></td>
</tr>
<tr>
<td>%rdx 0x100</td>
<td>0x108</td>
<td>movq (%rdx), %rsi</td>
<td></td>
</tr>
<tr>
<td>%rdi 0x100</td>
<td>0x100</td>
<td>leaq (%rdx,%rcx,4), %rax</td>
<td></td>
</tr>
<tr>
<td>%rsi 0x1</td>
<td>0x100</td>
<td>movq (%rdx,%rcx,4), %rbx</td>
<td></td>
</tr>
</tbody>
</table>

---

## Memory address-space layout

### Addr

2^n-1

### Perm

RW

### Contents

Procedure context

Dynamic data structures

Global variables/static data structures

String literals

Instructions

### Managed by

Compiler

Programmer, malloc/free, new/GC

Compiler/Assembler/Linker

Compiler/Assembler/Linker

Compiler/Assembler/Linker

### Initialized

Run time

Run time

Startup

Startup

Startup

---

## Call Stack

Memory region for temporary storage managed with stack discipline.

%rsp holds lowest stack address (address of “top” element)

Stack "Bottom"

Stack Pointer: %rsp

higher addresses

stack grows toward lower addresses

Stack "Top"
**x86: Three Basic Kinds of Instructions**

1. **Data movement between memory and register**
   - **Load** data from memory into register
     \[
     \%\text{reg} \leftarrow \text{Mem}[\text{address}]
     \]
   - **Store** register data into memory
     \[
     \text{Mem}[\text{address}] \leftarrow \%\text{reg}
     \]

2. **Arithmetic/logic on register or memory data**
   \[
   c = a + b; \quad z = x << y; \quad i = h \& g;
   \]

3. **Comparisons and Control flow to choose next instruction**
   - Unconditional jumps to/from procedures
   - Conditional branches

---

**Arithmetic Operations**

<table>
<thead>
<tr>
<th>Two-operand instructions:</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subq</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>mulq</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest \times Src</code></td>
</tr>
<tr>
<td><code>divq</code> <code>Src, Dest</code></td>
<td><code>Dest = \text{integer division of} \ Dest \div Src</code></td>
</tr>
<tr>
<td><code>andq</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest \&amp; Src</code></td>
</tr>
<tr>
<td><code>orq</code> <code>Src, Dest</code></td>
<td>`Dest = Dest</td>
</tr>
<tr>
<td><code>xorq</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest \oplus Src</code></td>
</tr>
<tr>
<td><code>incq</code> <code>Dest</code></td>
<td><code>Dest = Dest + 1</code></td>
</tr>
<tr>
<td><code>decq</code> <code>Dest</code></td>
<td><code>Dest = Dest - 1</code></td>
</tr>
<tr>
<td><code>negq</code> <code>Dest</code></td>
<td><code>Dest = -Dest</code></td>
</tr>
<tr>
<td><code>notq</code> <code>Dest</code></td>
<td><code>Dest = \overline{Dest}</code></td>
</tr>
</tbody>
</table>

See CSAPP 3.5.5 for: `mulq, cqto, idivq, divq`
leaq for arithmetic

```c
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>

### Example

```c
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>

### Example

```c
long arith(long x, long y, long z){
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```
long logical(long x, long y){
    long t1 = x^y;
    long t2 = t1 >> 17;
    long mask = (1<<13) - 7;
    long rval = t2 & mask;
    return rval;
}

logical:
    movq %rdi,%rax
    xorq %rsi,%rax
    sarq $17,%rax
    andq $8185,%rax
    retq
Compiler optimization example

```c
long logical(long x, long y){
    long t1 = x^y;
    long t2 = t1 >> 17;
    long mask = (1<<13) - 7;
    long rval = t2 & mask;
    return rval;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, t1, t2, rval</td>
</tr>
</tbody>
</table>

logical:
- movq %rdi,%rax
- xorq %rsi,%rax
- sarq $17,%rax
- andq $8185,%rax
- retq

x86: Three Basic Kinds of Instructions

1. Data movement between memory and register
   - Load data from memory into register
     %reg ← Mem[address]
   - Store register data into memory
     Mem[address] ← %reg
   - Memory is an array[] of bytes!

2. Arithmetic/logic on register or memory data
   - c = a + b;    z = x << y;    i = h & g;

   Next Time:

3. Comparisons and Control flow to choose next instruction
   - Unconditional jumps to/from procedures
   - Conditional branches