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

Human-readable language close to machine code.

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

Assembly Code
movq %rsi, (%rdx)
retq

Machine Instruction Example

C Code
*dest = t;

Assembly Code
movq %rsi, (%rdx)

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

Disassembling Object Code

Disassembled by GDB
0x00400536: 0x48 0x01 0xfe  add  %rdi,%rsi
0x00400000000400539: 0x48 0x89 0x32  mov  %rsi,(%rdx)
0x0040053c: 0xc3  retq

(a brief history of x86)

Word Size 8086 1978
First 16-bit processor. Basis for IBM PC & DOS 1MB address space

IA32 1985
First 32-bit ISA. Flat addressing, improved OS support

x86-64 2003*
Slow AMD/Intel conversion, slow adoption. Not actually x86-64 until few years later. Mainstream only after ~10 years.

Since 2016: most laptops, desktops, servers. 240 now:

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

ISA  First  Year
8086  Intel 8086  1978
First 16-bit processor. Basis for IBM PC & DOS 1MB address space

IA32  Intel 386  1985
First 32-bit ISA. Flat addressing, improved OS support

x86-64  AMD Opteron  2003*
Slow AMD/Intel conversion, slow adoption.
*Not actually x86-64 until few years later. Mainstream only after ~10 years.

Since 2016: most laptops, desktops, servers. 240 now:

ISA View

Memory

Processor

Registers

Condition Codes

Addresses

Data

Heap

Static Data (Global)
(String) Literals

Instructions

Stack

...
x86-64 registers

16 named registers
Each 64 bits (8 bytes)

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Argument 1</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument 2</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument 3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument 4</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument 5</td>
</tr>
<tr>
<td>%rsp</td>
<td>Argument 6</td>
</tr>
<tr>
<td>%rbp</td>
<td>Argument 7</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument 8</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument 9</td>
</tr>
<tr>
<td>%r10</td>
<td>Argument 10</td>
</tr>
<tr>
<td>%r11</td>
<td>Argument 11</td>
</tr>
<tr>
<td>%r12</td>
<td>Argument 12</td>
</tr>
<tr>
<td>%r13</td>
<td>Argument 13</td>
</tr>
<tr>
<td>%r14</td>
<td>Argument 14</td>
</tr>
<tr>
<td>%r15</td>
<td>Argument 15</td>
</tr>
</tbody>
</table>

1985: 32-bit extended register %eax
1978: 16-bit register %ax

Low 32 bits of %rsi
Low 16 bits of %rsi

historical artifacts

Some have special uses for particular instructions

x86-64 registers: function arguments and return value

Mnemonic:

Diana's %rdi
silk %rsi
dress %rdx
costs %rcx
$89 %r8
%9

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

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 conceptually an array[] of bytes!

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

Data movement instructions

\textbf{mov } \textit{Source, Dest}

“copy the contents of \textit{source} operand into \textit{dest} operand”

\textbf{movq} move 8-byte “quad word”
\textbf{movl} move 4-byte “long word”
\textbf{movw} move 2-byte “word”
\textbf{movb} move 1-byte “byte”

Historical terms based on the 16-bit days, not the current machine word size (64 bits)

Source, Dest operand types:

Immediate: Literal integer data
   Examples: $0x400 \quad $533

Register: One of 16 registers
   Examples: %rax \quad %rdx

Memory: consecutive bytes in memory, at address held by register
   Direct addressing: \( (\%rax) \)
   With displacement/offset: \( 8(\%rsp) \)
**Cannot do memory-memory transfer with a single instruction.**

*How would you do it?*

---

### 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Memory Addressing Modes

**Indirect**

\[(R)\] \[Reg[\text{Mem}]\]

Register \(R\) specifies memory address:
\[
\text{movq} \left( \%\text{rdcx} \right), \%\text{rax}
\]

**Displacement**

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

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

**General Form:**

\[D(R_b,R_i,S)\] \[Reg[\text{Mem} + D + S \times Reg[\text{Ri}]]\]

- \(D\): Literal “displacement” value represented in 1, 2, or 4 bytes
- \(R_b\): Base register: Any register
- \(R_i\): Index register: Any except \%rsp
- \(S\): Scale: 1, 2, 4, or 8

---

### Pointers and Memory Addressing

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

**swap:**

\[
\text{movq} \left( \%\text{rdi} \right), \%\text{rax}
\]
\[
\text{movq} \left( \%\text{rsi} \right), \%\text{rdx}
\]
\[
\text{movq} \%\text{rdx}, \left( \%\text{rdi} \right)
\]
\[
\text{movq} \%\text{rax}, \left( \%\text{rsi} \right)
\]
\[\text{retq}\]

---

### Pointers and Memory Addressing

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

**swap:**

\[
\text{movq} \left( \%\text{rdi} \right), \%\text{rax}
\]
\[
\text{movq} \left( \%\text{rsi} \right), \%\text{rdx}
\]
\[
\text{movq} \%\text{rdx}, \left( \%\text{rdi} \right)
\]
\[
\text{movq} \%\text{rax}, \left( \%\text{rsi} \right)
\]
\[\text{retq}\]
Pointers and Memory Addressing

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

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

**Memory Address**
- 0x0120
- 0x0108
- 0x0007
- 0x0003

**Address Computation Examples**

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
<th>Address Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td>0x0f00</td>
<td>0x8(%rdx)</td>
<td>0x8 + 0x0f00</td>
<td>0x0f08</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x100</td>
<td>($rdx,$rcx)</td>
<td>($rdx,$rcx)</td>
<td>0x100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>($rdx,$rcx,4)</td>
<td>($rdx,$rcx,4)</td>
<td>0x100</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]]
  - (Rb,Ri) = Mem[Reg[Rb]+Reg[Ri]+D]
  - (Rb,Ri,S) = Mem[Reg[Rb]+S*Reg[Ri]]

- Implicitly:
  - (S=1,D=0)
  - (S=1)
  - (D=0)
### Address Computation Examples

<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 * 1</td>
<td>0xf100</td>
</tr>
<tr>
<td>(,%rdx,%rcx,4)</td>
<td>0xf000 + 0x100 * 4</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>0x80 + 0x0 + 0xf000 * 2</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

### General Addressing Modes

<table>
<thead>
<tr>
<th>Register contents</th>
<th>D(Rb,Ri,S) Mem[Reg[Rb] + S*Reg[Ri] + D]</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx 0xf000</td>
<td></td>
</tr>
<tr>
<td>%rcx 0x100</td>
<td></td>
</tr>
</tbody>
</table>

### Special Cases:

- **Implicitly:**
  - (Rb,Ri) Mem[Reg[Rb] + Reg[Ri]] (S=1,D=0)
  - (Rb,Ri) Mem[Reg[Rb] + Reg[Ri] + D] (S=1)
- **D(Rb,Ri)** Mem[Reg[Rb] + Reg[Ri] + D] (S=0)

### Register contents

<table>
<thead>
<tr>
<th>General Addressing Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(Rb,Ri,S) Mem[Reg[Rb] + S*Reg[Ri] + D]</td>
</tr>
<tr>
<td>(Rb,Ri) Mem[Reg[Rb] + Reg[Ri]] (S=1,D=0)</td>
</tr>
<tr>
<td>(Rb,Ri) Mem[Reg[Rb] + Reg[Ri] + D] (S=1)</td>
</tr>
<tr>
<td>(Rb,Ri,S) Mem[Reg[Rb] + S*Reg[Ri] + D] (S=0)</td>
</tr>
</tbody>
</table>
Compute address given by this addressing mode expression and store it here.

Load effective address

Does not access memory

leaq Src, Dest

Uses: “address of” “Lovely Efficient Arithmetic”

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

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</td>
<td>0x120</td>
<td>0x400</td>
<td>leaq (%rdx,%rcx,4), %rax</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x118</td>
<td>0x0f</td>
<td>movq (%rdx,%rcx,4), %rbx</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x110</td>
<td>0x08</td>
<td>leaq (%rdx), %rdi</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x108</td>
<td>0x04</td>
<td>movq (%rdx), %rsi</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
<td>0x01</td>
<td></td>
</tr>
</tbody>
</table>

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

Load effective address

Does not access memory

leaq Src, Dest

Uses: “address of” “Lovely Efficient Arithmetic”

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

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</td>
<td>0x110</td>
<td>0x040</td>
<td>leaq (%rdx,%rcx,4), %rax</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x118</td>
<td>0x0f</td>
<td>movq (%rdx,%rcx,4), %rbx</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x110</td>
<td>0x08</td>
<td>leaq (%rdx), %rdi</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x108</td>
<td>0x04</td>
<td>movq (%rdx), %rsi</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
<td>0x01</td>
<td></td>
</tr>
</tbody>
</table>
Load effective address

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 \text{ Src, Dest}

leaq vs. movq

Memory address-space layout

<table>
<thead>
<tr>
<th>Addr</th>
<th>Perm</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>Stack</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td>2^n-1</td>
<td>W</td>
<td>Heap</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Statics</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>Literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Text</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

Call Stack

Memory region for temporary storage managed with stack discipline.

%p holds lowest stack address (address of "top" element)

Call Stack: Push, Pop

pushq \text{ Src}
1. Fetch value from Src
2. Decrement \%rsp by 8 (\text{why } 8?)
3. Store value at new address given by \%rsp

popq \text{ Dest}
1. Load value from address \%rsp
2. Write value to Dest
3. Increment \%rsp by 8
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

(Unlike the HW ISA, combines 1 operand and the destination)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>incq Dest</td>
<td>increment</td>
</tr>
<tr>
<td>location</td>
<td>decq Dest</td>
<td>decrement</td>
</tr>
<tr>
<td>location</td>
<td>negq Dest</td>
<td>negate</td>
</tr>
<tr>
<td>location</td>
<td>shrq Src, Dest</td>
<td>logical bitwise shift</td>
</tr>
<tr>
<td>location</td>
<td>shrq Src, Dest</td>
<td>arithmetic bitwise shift</td>
</tr>
<tr>
<td>location</td>
<td>xorq Src, Dest</td>
<td>bitwise exclusive or (XOR)</td>
</tr>
<tr>
<td>location</td>
<td>andq Src, Dest</td>
<td>bitwise and</td>
</tr>
<tr>
<td>location</td>
<td>orq Src, Dest</td>
<td>bitwise or</td>
</tr>
</tbody>
</table>

One-operand (unary) instructions

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>incq Dest</td>
<td>increment</td>
</tr>
<tr>
<td>location</td>
<td>decq Dest</td>
<td>decrement</td>
</tr>
<tr>
<td>location</td>
<td>negq Dest</td>
<td>negate</td>
</tr>
<tr>
<td>location</td>
<td>shrq Src, Dest</td>
<td>logical bitwise shift</td>
</tr>
</tbody>
</table>

See CSAPP 3.5.5 for: mulq, cqto, idivq, divq

lea 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;
}
```

<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></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
</tr>
</tbody>
</table>
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 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;
}
Compiler optimization example

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 0x8185, %rax
    retq
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
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`

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

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

Next lecture: