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 Machine Code

C Code

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

sum.c

**compiler (CS 301)**

gcc -Og -S sum.c

Generated x86 Assembly Code

Human-readable language close to machine code.

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

sum.s

**assembler**

Executable: sum

Resolve references between object files, libraries, (re)locate data

**linker**

Machine Instruction Example

C Code

*dest = t;

Assembly Code

```
movq %rsi, (%rdx)
```

Object Code

3-byte instruction encoding

Stored at address 0x400539

0x400539:  48 89 32
Disassembling Object Code

Disassembled by objdump -d sum

Disassembler

Object

Disassembled by GDB

0x00400536: 0x48 01 fe add %rdi,%rsi
0x00400539: 0x48 89 32 mov %rsi,(%rdx)
0x0040053c: c3 retq

a brief history of x86

<table>
<thead>
<tr>
<th>ISA</th>
<th>First</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>Intel 8086</td>
<td>1978</td>
</tr>
<tr>
<td>32</td>
<td>Intel 386</td>
<td>1985</td>
</tr>
<tr>
<td>64</td>
<td>AMD Opteron</td>
<td>2003</td>
</tr>
</tbody>
</table>

ISA View

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.

ISA View

Word Size

16

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

32

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

64

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

240 now:

2016: most laptops, desktops, servers.

x86 Basics 5

x86 Basics 6

x86 Basics 7

x86 Basics 8
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. Arithmetical/logical 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

---

Data movement instructions

**mov** _Source, Dest_

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: 8(\%rsp)

---

### movq Operand Combinations

\[
\begin{array}{ccc}
\text{Source} & \text{Dest} & \text{Src, Dest} \\%reg & \text{Reg} & \text{movq} $0x4, \%rax & \text{movq} \%rax, \%rdx & \text{movq} (%rax), \%rdx, \%d & \text{movq} \%rax, \%rdx, \%q \\text{Mem} & \text{Reg} & \text{movq} $-147, (%rax) & \text{movq} \%rax, (%rdx) & \text{movq} (%rax), \%rdx, \%p \\end{array}
\]

**C Analog**

- \( a = 0x4 \)
- \( *p = -147 \)
- \( d = a \)
- \( *q = a \)
- \( d = *p \)

---

**Cannot do memory-memory transfer with a single instruction.**

**How would you do it?**
Memory Addressing Modes

Indirect (R) Mem[Reg[R]]
Register R specifies memory address: \texttt{movq (%rcx), %rax}

Displacement D(R) Mem[Reg[R]+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)
\texttt{movq %rdx, 8(%rsp)}

General Form: D(Rb,Ri,S) Mem[Reg[\text{Reg[Rb]} + S\ast\text{Reg[Ri]} + D]]
D: Literal “displacement” value represented in 1, 2, or 4 bytes
Rb: Base register: \textit{Any register}
Ri: Index register: \textit{Any except %rsp}
S: Scale: 1, 2, 4, or 8

Pointers and Memory Addressing

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

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

Pointers and Memory Addressing
void swap(long* xp, long* yp){
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

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

### General Addressing Modes

<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></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ D(Rb,Ri,S) = \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri] + D] \]

**Special Cases:**
- (Rb,Ri)  \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \)  \( (S=1, D=0) \)
- D(Rb,Ri)  \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \)  \( (S=1) \)
- (Rb,Ri,S)  \( \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \)  \( (D=0) \)

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

**Load effective address**

\[ \text{leaq} \ Src, \ Dest \]

**Does not access memory**

**Uses:** "address of" "Lovely Efficient Arithmetic"

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

**Register**
- \( \%\text{rax} \)
- \( \%\text{rbx} \)
- \( \%\text{rcx} \)
- \( \%\text{rdx} \)
- \( \%\text{rdi} \)
- \( \%\text{rsi} \)

**Memory address-space layout**

- **Addr:** \( 2^{N-1} \)
- **Perm:** RW
- **Contents:**
  - Procedure context: Compiler
  - Dynamic data structures: Programmer, malloc/free, new/GC
  - Global variables/static data structures: Compiler/Assembler/Linker
  - String literals: Compiler/Assembler/Linker
  - Instructions: Compiler/Assembler/Linker

**Run time**

**Initialized**

**Call Stack**

Memory region for temporary storage managed with stack discipline.

\( \%\text{rsp} \) holds lowest stack address (address of "top" element)
Call Stack: **Push, Pop**

### pushq *Src*
1. Fetch value from *Src*
2. Decrement `%rsp` by 8 *(why 8?)*
3. Store value at new address given by `%rsp`

### popq *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
     - `%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;`

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

---

**Procedure Preview** *(more soon)*

**call, ret, push, pop**

Procedure arguments passed in 6 registers:

| `%rax` | Return Value |
| `%rbx` | Argument 1   |
| `%rcx` | Argument 2   |
| `%rdx` | Argument 3   |
| `%rsi` | Argument 4   |
| `%rdi` | Argument 5   |
| `%rbp` | Stack pointer |

Allocate/push new stack frame for each procedure call.

Some local variables, saved register values, extra arguments.

Deallocate/pop frame before return.

---

**Arithmetic Operations**

**Two-operand instructions:**
- **Format**: `addq Src, Dest`, `subq Src, Dest`, `imulq Src, Dest`, `salq Src, Dest`, `xorq Src, Dest`, `andq Src, Dest`

- **Computation**:
  - `Dest = Dest + Src`
  - `Dest = Dest - Src`
  - `Dest = Dest * Src`
  - `Dest = Dest << Src`
  - `Dest = Dest >> Src`
  - `Dest = Dest & Src`

**One-operand (unary) instructions**
- `incq Dest`, `decq Dest`, `neq Dest`, `notq Dest`

- **Computation**:
  - `Dest = Dest + 1`
  - `Dest = Dest - 1`
  - `Dest = ~Dest`

*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;
}
```

### 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;
}
```

```c
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
   
   ```c
   %reg ← Mem[address]
   ```

   **Store** register data into memory
   
   ```c
   Mem[address] ← %reg
   ```

2. **Arithmetic/logic on register or memory data**

   ```c
   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