x86 and Assembly

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/s20/

x86 and Assembly

Turning C into Machine Code

C Code

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

Generated x86 Assembly Code

Human-readable language close to machine code.

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

```
sum.s
```

```
sum.c
```

compiler (CS 301)

```bash
gcc -Og -S sum.c
```

assembler

Executable: sum

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

```
sum.o
```

Object Code

3-byte instruction encoding

Stored at address 0x400539

```
0x400539: 48 89 32
```

Machine Instruction Example

```
*dest = t;
movq %rsi, (%rdx)
```

C Code

Store value t where indicated by dest

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
Disassembled by \texttt{objdump -d sum}

CISC vs. RISC

\textbf{x86:} real ISA, widespread

\textbf{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

\textbf{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

We will stick to a small subset.

\texttt{HW:} toy, but based on real MIPS ISA

\textbf{x86 and Assembly}

\textbf{ISA View}

\textbf{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 Intel 8086</td>
<td>1978</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8086</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 Intel 386</td>
<td>1985</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IA32</td>
<td>First 32-bit ISA. Flat addressing, improved OS support</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>x86-64 AMD Opteron</td>
<td>2003*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x86-64</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>

2016: most laptops, desktops, servers.

240 now: 64

\texttt{x86 and Assembly}
### x86-64 registers

- `%rax`: Return Value
- `%rbx`: Argument 4
- `%rcx`: Argument 3
- `%rdx`: Argument 2
- `%rsi`: Argument 1
- `%rdi`: Argument 1 (Special Purpose: Stack Pointer)
- `%r8`: Argument 5
- `%r9`: Argument 6
- `%r10`
- `%r11`
- `%r12`
- `%r13`
- `%r14`
- `%r15`

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

- `%rax `:
- `%eax`: High and low bytes of `%ax`
- `%esi`: Low 32 bits of `%rsi`
- `%esi`: Low 16 bits of `%rsi`

Some have special uses for particular instructions.

- `%r8`: 32-bit sub-register to match

### 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)

### 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

### mov Operand Combinations

- **Source**
- **Dest**
- **Src,Dest**
- **C Analog**

- **Imm**
  - `movq $0x4,%rax` → `a = 0x4;`

- **Mem**
  - `movq $-147,(%rax)` → `*p = -147;`

- **Reg**
  - `movq %rax,%rdx` → `d = a;`
  - `movq %rax,(%rdx)` → `*q = a;`
  - `movq (%rax),%rdx` → `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:
```
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)
```
movq %rdx,8(%rsp)
```

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

Pointer and Memory Addressing

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

Registros |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x400</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

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></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>

General Addressing Modes

<table>
<thead>
<tr>
<th>D(Rb,Ri,S)</th>
<th>Mem[Reg[Rb]+S*Reg[Ri]+D]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special Cases:

<table>
<thead>
<tr>
<th>(Rb,Ri)</th>
<th>Mem[Reg[Rb]+Reg[Ri]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(Rb,Ri)</td>
<td>Mem[Reg[Rb]+Reg[Ri]+D]</td>
</tr>
<tr>
<td>(Rb,Ri,S)</td>
<td>Mem[Reg[Rb]+S*Reg[Ri]]</td>
</tr>
</tbody>
</table>

Implicitly:

<table>
<thead>
<tr>
<th>S=1,D=0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(Rb,Ri)</td>
<td>Mem[Reg[Rb]+Reg[Ri]]</td>
</tr>
<tr>
<td>(Rb,Ri)</td>
<td>Mem[Reg[Rb]+Reg[Ri]+D]</td>
</tr>
<tr>
<td>(Rb,Ri,S)</td>
<td>Mem[Reg[Rb]+S*Reg[Ri]]</td>
</tr>
</tbody>
</table>

Computes 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*i, where k = 1, 2, 4, or 8
```

```
leaq  Src, Dest
```

Assembly Code

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x0f</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x08</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaq (%rdx,%rcx,4), %rax</td>
</tr>
<tr>
<td>movq (%rdx,%rcx,4), %rbx</td>
</tr>
<tr>
<td>leaq (%rdx), %rdi</td>
</tr>
<tr>
<td>movq (%rdx), %rsi</td>
</tr>
</tbody>
</table>
### Memory 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>(2^n-1)</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run-time</td>
</tr>
<tr>
<td>Heap</td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run-time</td>
</tr>
<tr>
<td>Statics</td>
<td>RW</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>Literals</td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>Text</td>
<td>X</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

### Call Stack

- **Stack “Bottom”**
  - %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

Those bits are still there; we’re just not using them.

### Procedure Preview (more soon)

- **Call, ret, push, pop**
  - Procedure arguments passed in 6 registers:
    - %rax: Return Value
    - %rbx: Argument 1
    - %rdx: Argument 2
    - %rsi: Argument 3
    - %rdi: Argument 4
    - %r8: Argument 5
    - %r9: Argument 6

- Return value in %rax.
- 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Argument order</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq Src, Dest</td>
<td>Dest = Dest + Src</td>
<td>a.k.a. salq</td>
</tr>
<tr>
<td>subq Src, Dest</td>
<td>Dest = Dest - Src</td>
<td></td>
</tr>
<tr>
<td>imulq Src, Dest</td>
<td>Dest = Dest * Src</td>
<td></td>
</tr>
<tr>
<td>shlq Src, Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
<td></td>
</tr>
<tr>
<td>sarq Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
<td></td>
</tr>
<tr>
<td>shrq Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
<td></td>
</tr>
<tr>
<td>xorq Src, Dest</td>
<td>Dest = Dest ^ Src</td>
<td></td>
</tr>
<tr>
<td>andq Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
<td></td>
</tr>
<tr>
<td>orq Src, Dest</td>
<td>Dest = Dest</td>
<td>Logical</td>
</tr>
</tbody>
</table>

One-operand (unary) instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq Dest</td>
<td>Dest = Dest + 1</td>
<td>increment</td>
</tr>
<tr>
<td>decq Dest</td>
<td>Dest = Dest - 1</td>
<td>decrement</td>
</tr>
<tr>
<td>negq Dest</td>
<td>Dest = ~Dest</td>
<td>negate</td>
</tr>
<tr>
<td>notq Dest</td>
<td>Dest = ~Dest</td>
<td>bitwise complement</td>
</tr>
</tbody>
</table>

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

Another 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;
}
```

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

```assembly
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;
}
```

```assembly
arith:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    leaq (%rsi,%rsi,2), %rdx
    salq $4, %rdx
    leaq 4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    retq
```

```assembly
Register | Use(s)          |
---------|-----------------|
%rdi     | Argument x      |
%rsi     | Argument y      |
%rdx     | Argument z      |
%rax     |                 |
%rcx     |                 |
```