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/
Devices (transistors, etc.)
Solid-State Physics
Hardware
Software
next few weeks
Program, Application
Programming Language
Compiler/Interpreter
Operating System
Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)
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

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

Human-readable language close to machine code.

`sum.c`

`gcc -Og -S sum.c`

`sum.s`

`assembler`

Object Code

```
01010101100010011110010110
00101101000101000011000000
0011010001010001000100010
01111011000101111011100011
```

`sum.o`

Executable: `sum`

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

*C Code*

`*dest = t;`

*Assembly Code*

`movq %rsi, (%rdx)`

*Object Code*

`0x400539: 48 89 32`

*Store value t where indicated by dest*

*Move 8-byte value to memory*

\[
\text{t: Register } \%\text{rsi} \\
\text{dest: Register } \%\text{rdx} \\
\star\text{dest: Memory } \text{M}[\%\text{rdx}]
\]
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

```
0x00400536:
  0x48
  0x01
  0xfe
  0x48
  0x89
  0x32
  0xc3
```

Disassembled by GDB

```
0x000000000000400536 <+0>:  add   %rdi,%rsi
0x000000000000400539 <+3>:  mov    %rsi,(%rdx)
0x00000000000040053c <+6>:  retq
```

```
$ 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.

**HW**: toy, but based on real MIPS ISA

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

*Not actually x86-64 until few years later.*
ISA View

Processor

PC

Registers

Condition Codes

Memory

Stack

... 

Heap

Static Data (Global)

(String) Literals

Instructions

Addresses

Data

Instructions
### x86-64 registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument 4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument 3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument 2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument 1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Special Purpose: Stack Pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>Argument 5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument 6</td>
</tr>
<tr>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td></td>
</tr>
</tbody>
</table>

64-bits / 8 bytes

### Historical Artifacts

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

#### sub-registers

- `%rax` is a 64-bit register.
- `%rax` is split into `%eax` (32-bit) and `%ah` (high byte) and `%al` (low byte).
- `%rsi` is a 32-bit register.
- `%rsi` is split into `%esi` (low 32-bit) and `%si` (low 16-bit).
- `%r8` is a 32-bit sub-register to match the historical artifacts.

Some have special uses for particular instructions.
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
   \[ c = a + b; \quad z = x \ll y; \quad i = h \& g; \]

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

\texttt{mov} \quad \textit{Source, Dest}

data size \textit{is} one of \{b, w, l, q\}

\begin{itemize}
\item \texttt{movq}: move 8-byte “quad word”
\item \texttt{movl}: move 4-byte “long word”
\item \texttt{movw}: move 2-byte “word”
\item \texttt{movb}: move 1-byte “byte”
\end{itemize}

Source/Dest operand types:

\textbf{Immediate}: Literal integer data
\begin{itemize}
\item Examples: $0x400 \quad \text{and} \quad -533$
\end{itemize}

\textbf{Register}: One of 16 registers
\begin{itemize}
\item Examples: \texttt{%rax} \quad \texttt{%rdx}
\end{itemize}

\textbf{Memory}: consecutive bytes in memory, at address held by register
\begin{itemize}
\item Direct addressing: \quad \texttt{(%rax)}
\item With displacement/offset: \quad 8\texttt{(%rsp)}
\end{itemize}

Historical terms based on the 16-bit days, \textbf{not} the current machine word size (64 bits)
**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>Imm</td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
<td>a = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>d = a;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq %rax,(%rdx)</td>
<td>*q = a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
<td>d = *p;</td>
</tr>
</tbody>
</table>

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

*How would you do it?*
Memory Addressing Modes

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

Register \( R \) specifies memory address: \( \text{movq} \ (\%r\text{cx}),\%r\text{ax} \)

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

**Register** \( R \) specifies **base** memory address (e.g. base of an object)

**Displacement** \( D \) specifies literal **offset** (e.g. a field in the object)

\( \text{movq} \ \%r\text{dx},8(\%r\text{sp}) \)

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

- \( D \): Literal “displacement” value represented in 1, 2, or 4 bytes
- \( Rb \): Base register: Any register
- \( Ri \): Index register: Any except \( \%r\sp \)
- \( S \): Scale: 1, 2, 4, or 8
void swap(long* xp, long* yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
</tr>
</tbody>
</table>
## Address Computation Examples

### General Addressing Modes

<table>
<thead>
<tr>
<th>Address Expression</th>
<th>Address Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>Mem[Reg[Rb] + D]</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>Mem[Reg[Rb] + Reg[Ri]]</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>Mem[Reg[Rb] + S*Reg[Ri]]</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>Mem[Reg[Rb] + S*Reg[Ri]]</td>
</tr>
</tbody>
</table>

### Special Cases:

- (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]]

### Implicitly:

- (S=1,D=0)
- (S=1)
- (D=0)

### Register contents:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td>0xf000</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x100</td>
</tr>
</tbody>
</table>
Compute address given by this addressing mode expression and store it here.

**Load effective address**

`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</td>
<td>0x400</td>
<td>0x120</td>
<td><code>leaq (%rdx, %rcx, 4), %rax</code></td>
</tr>
<tr>
<td>%rbx</td>
<td>0xf</td>
<td>0x118</td>
<td><code>movq (%rdx, %rcx, 4), %rbx</code></td>
</tr>
<tr>
<td>%rcx</td>
<td>0x4</td>
<td>0x110</td>
<td><code>leaq (%rdx), %rdi</code></td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
<td>0x108</td>
<td><code>movq (%rdx), %rsi</code></td>
</tr>
<tr>
<td>%rdi</td>
<td>0x10</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>0x1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Memory Layout

The memory layout of a system can be divided into several regions:

- **Stack**:
  - Address: $2^{N-1}$
  - Permissions: Read/Write (RW)
  - Contents: Procedure context
  - Managed by: Compiler
  - Initialized: Run-time

- **Heap**:
  - Permissions: Read/Write (RW)
  - Contents: Dynamic data structures
  - Managed by: Programmer, malloc/free, new/GC
  - Initialized: Run-time

- **Statics**:
  - Permissions: Read/Write (RW)
  - Contents: Global variables/static data structures
  - Managed by: Compiler/Assembler/Linker
  - Initialized: Startup

- **Literals**:
  - Permissions: Read-only (R)
  - Contents: String literals
  - Managed by: Compiler/Assembler/Linker
  - Initialized: Startup

- **Text**:
  - Permissions: Executable (X)
  - Contents: Instructions
  - Managed by: Compiler/Assembler/Linker
  - Initialized: Startup

---

*x86 and Assembly* 19
Call Stack

Memory region for temporary storage managed with stack discipline.

%rsp holds lowest stack address (address of "top" element)
Call Stack: Push, Pop

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

**popq**  \textit{Dest}  
1. Load value from address \%rsp
2. Write value to \textit{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:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument 4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument 3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument 2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument 1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>Argument 5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument 6</td>
</tr>
<tr>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td></td>
</tr>
</tbody>
</table>

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.

---

**x86 and Assembly**
## Arithmetic Operations

### Two-operand instructions:

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

### One-operand (unary) instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
<th>Notes</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
**leaq for arithmetic**

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></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
</tr>
</tbody>
</table>

**arith:**
- leaq (%rdi,%rsi), %rax
- addq %rdx, %rax
- leaq (%rsi,%rsi,2), %rdx
- salq $4, %rdx
- leaq 4(%rdi,%rdx), %rcx
- imulq %rcx, %rax
- ret

*x86 and Assembly*
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;
}
```

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

logical:

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