CS 31: Intro to Systems
ISAs and Assembly

Martin Gagné
Swarthmore College
February 7, 2017
ANNOUNCEMENT

All labs will meet in SCI 252 (the robot lab) tomorrow.
Overview

• How to directly interact with hardware

• Instruction set architecture (ISA)
  • Interface between programmer and CPU
  • Established instruction format (assembly lang)

• Assembly programming (IA-32)
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O
Abstraction

Applications
Specific functionality

Operating system
Manage resources

Complex devices
Compute & I/O

This week: Machine Interface

Last week: Circuits, Hardware Implementation
Compilation Steps (.c to a.out)

- **text**
  - C program (`p1.c`)

  Usually compile to `a.out` in a single step: `gcc -m32 p1.c`

  - `-m32` tells `gcc` to compile for 32-bit Intel machines

- **Compiler**
  - `gcc -m32`

  Reality is more complex: there are intermediate steps!

- **executable binary**
  - Executable code (`a.out`)
Compilation Steps (.c to a.out)

You can see the results of intermediate compilation steps using different gcc flags.
Assembly Code

Human-readable form of CPU instructions
  • Almost a 1-to-1 mapping to Machine Code
  • Hides some details:
    • Registers have names rather than numbers
    • Instructions have names rather than variable-size codes

We’re going to use IA32 (x86) assembly
  • CS lab machines are 64 bit version of this ISA, but they can also run the 32-bit version (IA32)
  • Can compile C to IA32 assembly on our system:
    gcc -m32 -S code.c  # open code.s in vim to view
Compilation Steps (.c to a.out)

- **C program** (`p1.c`)
  - **Compiler** (`gcc -m32 -S`)
  - **Assembly program** (`p1.s`)
    - **Assembler** (`gcc -c (or as)`)
  - **Object code** (`p1.o`)
    - **Linker** (`gcc (or ld)`)
Object / Executable / Machine Code

Assembly
push %ebp
mov %esp, %ebp
sub $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), %eax
addl %eax, -8(%ebp)
movl -8(%ebp), %eax
leave

C Code
int main() {
    int a = 10;
    int b = 20;
    a = a + b;
    return a;
}
Compilation Steps (.c to a.out)

- **High-level language**
- **Interface for speaking to CPU**

**C program (p1.c)**

- Compiler (`gcc -m32 -S`)

**Assembly program (p1.s)**

- Assembler (`gcc -c (or as) `)

**Object code (p1.o)**

- Linker (`gcc (or ld) `)

**Executable code (a.out)**

- Other object files (`p2.o, p3.o, ...`)
- Library obj. code (`libc.a`)
Instruction Set Architecture (ISA)

• ISA (or simply architecture): Interface between lowest software level and the hardware.

• Defines specification of the language for controlling CPU state:
  • Provides a set of instructions
  • Makes CPU registers available
  • Allows access to main memory
  • Exports control flow (change what executes next)
Instruction Set Architecture (ISA)

• The agreed-upon interface between all software that runs on the machine and the hardware that executes it.
ISA Examples

• Intel IA-32 (80x86)
• ARM
• MIPS
• PowerPC
• IBM Cell
• Motorola 68k
• Intel IA-64 (Itanium)
• VAX
• SPARC
• Alpha
• IBM 360

How many of these have you used?
ISA Characteristics

• Above ISA: High-level language (C, Python, …)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change
  (faster, smaller, …)
  • ISA is like a CPU “family”
**Instruction Translation**

**sum.c (High-level C)**

```c
int sum(int x, int y)
{
    int res;
    res = x+y;
    return res;
}
```

**sum.s (Assembly)**

```assembly
sum:
    pushl %ebp
    movl %esp,%ebp
    subl $24, %esp
    movl -12(%ebp),%eax
    addl -8(%ebp),%eax
    movl %eax, -12(%ebp)
    leave
    ret
```

**sum.s from sum.c:**

```bash
gcc -m32 -S sum.c
```

- Instructions to set up the stack frame and get argument values
- An add instruction to compute sum
- Instructions to return from function
ISA Design Questions

**sum.c (High-level C)**

```c
int sum(int x, int y)
{
    int res;
    res = x+y;
    return res;
}
```

**sum.s (Assembly)**

```assembly
sum:
    pushl %ebp
    movl %esp,%ebp
    subl $24, %esp
    movl -12(%ebp),%eax
    addl -8(%ebp),%eax
    movl %eax, -12(%ebp)
    leave
    ret
```

**sum.s from sum.c:**

```bash
gcc -m32 -S sum.c
```

What should these instructions do?

What is/isn’t allowed by hardware?

How complex should they be?

Example: supporting multiplication.
C statement: $A = A \times B$

Simple instructions:

- LOAD A, eax
- LOAD B, ebx
- PROD eax, ebx
- STORE ebx, A

Powerful instructions:

- MULT B, A

Translation:
Load the values ‘A’ and ‘B’ from memory into registers, compute the product, store the result in memory where ‘A’ was.
Which would you use if you were designing an ISA for your CPU? (Why?)

Simple instructions:

LOAD A, eax
LOAD B, ebx
PROD eax, ebx
STORE ebx, A

Powerful instructions:

MULT B, A

A. Simple
B. Powerful
C. Something else
RISC versus CISC (Historically)

• Complex Instruction Set Computing (CISC)
  • Large, rich instruction set
  • More complicated instructions built into hardware
  • Multiple clock cycles per instruction
  • Easier for humans to reason about

• Reduced Instruction Set Computing (RISC)
  • Small, highly optimized set of instructions
  • Memory accesses are specific instructions
  • One instruction per clock cycle
  • Compiler: more work, more potential optimization
So . . . Which System “Won”?  

• Most ISAs (after mid/late 1980’s) are RISC  

• The ubiquitous Intel x86 is CISC  
  • Tablets and smartphones (ARM) taking over?  

• x86 breaks down CISC assembly into multiple, RISC-like, machine language instructions  

• Distinction between RISC and CISC is less clear  
  • Some RISC instruction sets have more instructions than some CISC sets
Intel x86 Family (IA-32)

Intel i386 (1985)
• 12 MHz - 40 MHz
• ~300,000 transistors
• Component size: 1.5 µm

Intel Core i7 4770k (2013)
• 3,500 MHz
• ~1,400,000,000 transistors
• Component size: 22 nm

Everything in this family uses the same ISA (Same instructions)!
Assembly Programmer’s View of State

CPU

32-bit Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td>next instr addr (PC)</td>
</tr>
<tr>
<td>%EFLAGS</td>
<td>cond. codes</td>
</tr>
</tbody>
</table>

Registers:

- **PC**: Program counter (%eip)
- Condition codes (%EFLAGS)
- General Purpose (%eax - %ebp)

Memory

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Program: data
instructions
stack

- Byte addressable array
- Program code and data
- Execution stack

BUS

Addresses

Data

Instructions
Processor State in Registers

- Information about currently executing program
- Temporary data (%eax - %edi)
- Location of runtime stack (%ebp, %esp)
- Location of current code control point (%eip, ...)
- Status of recent tests %EFLAGS (CF, ZF, SF, OF)

General purpose registers

Current stack top

Current stack frame

Instruction pointer (PC)

Condition codes
General purpose Registers

• Remaining Six are for instruction operands
  • Can store 4 byte data or address value (ex. 3 + 5)

<table>
<thead>
<tr>
<th>Register name</th>
<th>Register value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>3</td>
</tr>
<tr>
<td>%ecx</td>
<td>5</td>
</tr>
<tr>
<td>%edx</td>
<td>8</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td></td>
</tr>
<tr>
<td>%EFLAGS</td>
<td></td>
</tr>
</tbody>
</table>

The low-order 2 bytes and two low-order 1 bytes of some of these can be named.

%eax is the low-order 16 bits of %eax
%al is the low-order 8 bits of %eax

May see their use in ops involving shorts or chars

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Types of IA32 Instructions

• Data movement
  – Move values between registers and memory
  – Example: movl

• Load: move data from memory to register

• Store: move data from register to memory
Data Movement

Move values between memory and registers or between two registers.

**Program Counter (PC):** Memory address of next instr

**Instruction Register (IR):** Instruction contents (bits)

- Data in
- WE
- Data in
- WE
- Data in
- WE
- Data in
- WE

- 32-bit Register #0
- 32-bit Register #1
- 32-bit Register #2
- 32-bit Register #3

- ...  

- Register File

- MUX

- MUX

- ALU
Types of IA32 Instructions

• Data movement
  – Move values between registers and memory

• Arithmetic
  – Uses ALU to compute a value
  – Example: addl
Arithmetic

Use ALU to compute a value, store result in register / memory.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

Data in
WE

32-bit Register #0

32-bit Register #1

32-bit Register #2

32-bit Register #3

Register File

ALU

MUX

MUX

(Memory)
Types of IA32 Instructions

• Data movement
  – Move values between registers and memory

• Arithmetic
  – Uses ALU to compute a value

• Control
  – Change PC based on ALU condition code state
  – Example: jmp
Control

Change PC based on ALU condition code state.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

Data in, WE

32-bit Register #0

32-bit Register #1

32-bit Register #2

32-bit Register #3

... Register File

(ALU)

(Memory)
Types of IA32 Instructions

• Data movement
  • Move values between registers and memory

• Arithmetic
  • Uses ALU to compute a value

• Control
  • Change PC based on ALU condition code state

• Stack / Function call  (We’ll cover these in detail later)
  • Shortcut instructions for common operations
Addressing Modes

• Data movement and arithmetic instructions:
  • Must tell CPU where to find operands, store result

• You can refer to a register by using %:
  • %eax

• addl %ecx, %eax
  • Add the contents of registers ecx and eax, store result in register eax.
Addressing Mode: Immediate

• Refers to a constant value, starts with $ $

• movl $10, %eax
  • Put the constant value 10 in register eax.
Addressing Mode: Memory

• Accessing memory requires you to specify which address you want.
  • Put address in a register.
  • Access with () around register name.

• `movl (%ecx), %eax`
  • Use the *address in* register `ecx` to access memory, store result in register `eax`
Addressing Mode: Memory

• `movl (%ecx), %eax`
  • Use the address in register `ecx` to access memory, store result in register `eax`

CPU Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(Memory)
Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register `ecx` to access memory, store result in register `eax`

<table>
<thead>
<tr>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>%eax</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

1. Index into memory using the address in `ecx`. 
Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register `ecx` to access memory, store result in register `eax`

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>42</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Index into memory using the address in `ecx`.

2. Copy value at that address to `eax`.
Addressing Mode: Displacement

• Like memory mode, but with constant offset
  • Offset is often negative, relative to %ebp

• movl  -12(%ebp), %eax
  • Take the address in ebp, subtract 12 from it, index into memory and store the result in eax
Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
  - Take the address in `%ebp`, subtract 12 from it, index into memory and store the result in `%eax`

### CPU Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1A70</td>
</tr>
</tbody>
</table>

1. Access address:
   
   $0x1A70 - 12 = 0x1A64$
Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
- Take the address in `<ebp>`, subtract 12 from it, index into memory and store the result in `<eax>`

### CPU Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%eax</code></td>
<td>11</td>
</tr>
<tr>
<td><code>%ecx</code></td>
<td>0x1A68</td>
</tr>
<tr>
<td><code>%ebp</code></td>
<td>0x1A70</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

---

(Memory)

1. Access address:
   
   \[ 0x1A70 - 12 = 0x1A64 \]

2. Copy value at that address to `<eax>`.
Let’s try a few examples...
What will memory look like after these instructions?

- \(x\) is 2 at \(\%ebp-8\), \(y\) is 3 at \(\%ebp-12\), \(z\) is 2 at \(\%ebp-16\)

```
movl  -16(%ebp), %eax
sall  $3, %eax
imull $3, %eax
movl  -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

### Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>?</td>
</tr>
<tr>
<td>%edx</td>
<td>?</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1270</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>2</td>
</tr>
<tr>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td>0x1268</td>
<td>2</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>
What will memory look like after these instructions?

\[
x \text{ is } 2 \text{ at } \%\text{ebp}-8, \quad y \text{ is } 3 \text{ at } \%\text{ebp}-12, \quad z \text{ is } 2 \text{ at } \%\text{ebp}-16
\]

movl -16(\%ebp), %eax
sall $3, %eax
imull $3, %eax
movl -12(\%ebp), %edx
addl -8(\%ebp), %edx
addl %edx, %eax
movl %eax, -8(\%ebp)

\begin{tabular}{|c|c|}
\hline
address & value \\
\hline
0x1260 & 53 \\
0x1264 & 3 \\
0x1268 & 24 \\
0x126c & \\
0x1270 & \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
address & value \\
\hline
0x1260 & 53 \\
0x1264 & 3 \\
0x1268 & 2 \\
0x126c & \\
0x1270 & \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
address & value \\
\hline
0x1260 & 2 \\
0x1264 & 16 \\
0x1268 & 24 \\
0x126c & \\
0x1270 & \\
\hline
\end{tabular}
Solution

x is 2 at %ebp-8, y is 3 at %ebp-12, z is 2 at %ebp-16

```assembly
movl  -16(%ebp), %eax
sall   $3, %eax
imull  $3, %eax
movl  -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

Equivalent C code:

```c
x = z*24 + y + x;
```

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
<td>0x1270</td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td>0x126c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1260</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0x1268</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>
Solution

x is 2 at %ebp-8, y is 3 at %ebp-12, z is 2 at %ebp-16

movl -16(%ebp), %eax # R[%eax] ← z (2)
sall $3, %eax       # R[%eax] ← z<<3  (16)  \[\text{Z*24}\]
imull $3, %eax       # R[%eax] ← 16*3  (48)
movl -12(%ebp), %edx # R[%edx] ← y     (3)
addl -8(%ebp), %edx  # R[%edx] ← y + x (5)
addl %edx, %eax      # R[%eax] ← 48+5  (53)
movl %eax, -8(%ebp)  # M[R[ebp]+8]←5  (x=53)

Equivalent C code:

```
x = z*24 + y + x;
```

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x1270</td>
</tr>
<tr>
<td>%edx</td>
<td>0x126c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1260</td>
</tr>
<tr>
<td></td>
<td>0x1264</td>
</tr>
<tr>
<td></td>
<td>0x1268</td>
</tr>
<tr>
<td></td>
<td>0x1270</td>
</tr>
</tbody>
</table>
REMINDER

All labs will meet in SCI 252 (the robot lab) tomorrow.