CS 31: Intro to Systems
ISAs and Assembly

Kevin Webb
Swarthmore College
February 9, 2016
Reading Quiz
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

This week: Machine Interface

Operating system
Manage resources

Complex devices
Compute & I/O

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

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

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

Reality is more complex: there are intermediate steps!
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)

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

- **C program (p1.c)**
  - Compiled by `gcc -m32 -S`.
- **Assembly program (p1.s)**
  - Assembled by `gcc -c (or as)`.
- **Object code (p1.o)**
  - Linked by `gcc (or ld)`.
- **Executable code (a.out)**
  - Includes library code (libc.a) and other object files (p2.o, p3.o, ...).
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

int main() {
    int a = 10;
    int b = 20;
    a = a + b;
    return a;
}
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))

Executable code (a.out)

High-level language

Interface for speaking to CPU

CPU-specific format (011010...)

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 ISAs have you used? (Don’t worry if you’re not sure. Try to guess based on the types of CPUs you interact with.)

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

A. 0  
B. 1-2  
C. 3-4  
D. 5-6  
E. 7+
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”
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 from sum.c:
```
gcc -m32 -S sum.c
```

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
```

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 from sum.c:**

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

**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
```

What should these instructions do?

What is/isn’t allowed by hardware?

How complex should they be?

Example: supporting multiplication.
C statement: A = A*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
ISA Examples

- Intel IA-32 (CISC)
- ARM (RISC)
- MIPS (RISC)
- PowerPC (RISC)
- IBM Cell (RISC)
- Motorola 68k (CISC)
- Intel IA-64 (Neither)
- VAX (CISC)
- SPARC (RISC)
- Alpha (RISC)
- IBM 360 (CISC)
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”
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

- **Address**
- **Value**
  - 0xffffffff

Program:
- data
- instrs
- stack

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>General purpose registers</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>Current stack top</td>
</tr>
<tr>
<td>%ebp</td>
<td>Current stack frame</td>
</tr>
<tr>
<td>%eip</td>
<td>Instruction pointer (PC)</td>
</tr>
</tbody>
</table>

**Condition codes**

- CF
- ZF
- SF
- OF
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 (see fig 3.2)

- %ax 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>bits: 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>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</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>
</tr>
<tr>
<td>%esp</td>
<td>%sp</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>
</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

ALU

MUX

MUX

(Memory)
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)

Register File

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

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

MUX
MUX

Register File
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`

<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>
</tbody>
</table>

CPU Registers

(Memory)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0:</td>
<td></td>
</tr>
<tr>
<td>0x4:</td>
<td></td>
</tr>
<tr>
<td>0x8:</td>
<td></td>
</tr>
<tr>
<td>0xC:</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1A64</td>
<td></td>
</tr>
<tr>
<td>0x1A68</td>
<td>42</td>
</tr>
<tr>
<td>0x1A6C</td>
<td></td>
</tr>
<tr>
<td>0x1A70</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xFFFFFFFF:</td>
<td></td>
</tr>
</tbody>
</table>
Addressing Mode: Memory

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

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>
</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

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

- **movl -12(%ebp), %eax**
  - Take the address in ebp, subtract twelve 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>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Access address:
   0x1A70 – 12 => 0x1A64
Addressing Mode: Displacement

• `movl -12(%ebp), %eax`
  – Take the address in ebp, subtract three 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>11</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1A70</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Access address: 0x1A70 – 12 => 0x1A64

2. Copy value at that address to eax.

(Memory)
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

```汇编
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)

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>53</td>
</tr>
<tr>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td>0x1268</td>
<td>24</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>

A: | B: | C: | D:

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>53</td>
</tr>
<tr>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td>0x1268</td>
<td>53</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>

<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>16</td>
</tr>
<tr>
<td>0x126c</td>
<td>24</td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>

<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>16</td>
</tr>
<tr>
<td>0x1268</td>
<td>24</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<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
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;
```
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)
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>2</td>
</tr>
<tr>
<td>%edx</td>
<td>3</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1270</td>
</tr>
</tbody>
</table>
What will the machine state be after executing these instructions?

```assembly
movl %ebp, %ecx
subl $16, %ecx
movl (%ecx), %eax
orl %eax, -8(%ebp)
negl %eax
movl eax, 4(%ecx)
```

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x455C</td>
<td>7</td>
</tr>
<tr>
<td>0x4560</td>
<td>11</td>
</tr>
<tr>
<td>0x4564</td>
<td>5</td>
</tr>
<tr>
<td>0x4568</td>
<td>3</td>
</tr>
<tr>
<td>0x456C</td>
<td></td>
</tr>
</tbody>
</table>

<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>%ebp</td>
<td>0x456C</td>
</tr>
</tbody>
</table>
How would you do this in IA32?

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

<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>

C code: $z = x ^ y$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
How would you do this in IA32?

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

<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>

C code: \( z = x \ ^ \ ^ \ y \)

A:  
```assembly
movl  -8(%ebp), %eax
movl  -12(%ebp), %edx
xorl  %eax, %edx
movl  %eax, -16(%ebp)
```

B:  
```assembly
movl  -8(%ebp), %eax
movl  -12(%ebp), %edx
xorl  %edx, %eax
movl  %eax, -16(%ebp)
```

C:  
```assembly
movl  -8(%ebp), %eax
movl  -12(%ebp), %edx
xorl  %eax, %edx
movl  %eax, -8(%ebp)
```

D:  
```assembly
movl  -16(%ebp), %eax
movl  -12(%ebp), %edx
xorl  %edx, %eax
movl  %eax, -8(%ebp)
```
How would you do this in IA32?

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

\[
x = y \gg 3 \mid x \times 8
\]
(1) $z = x \ ^\ ^{\wedge} \ y$

movl -8(%ebp), %eax  # R[%eax] $\leftarrow$ x
movl -12(%ebp), %edx  # R[%edx] $\leftarrow$ y
xorl %edx, %eax  # R[%eax] $\leftarrow$ x $\ ^\ ^{\wedge}$ y
movl %eax, -16(%ebp)  # M[R[%ebp-16]] $\leftarrow$ x$^{\wedge}$y

(2) $x = y \gg 3 \ | \ x \ast 8$

movl -8(%ebp), %eax  # R[%eax] $\leftarrow$ x
imull $8, %eax  # R[%eax] $\leftarrow$ x*8
movl -12(%ebp), %edx  # R[%edx] $\leftarrow$ y
rshl $3, %edx  # R[%edx] $\leftarrow$ y $\gg$ 3
orl %eax, %edx  # R[%edx] $\leftarrow$ y$\gg$3 $|$ x*8
movl %edx, -8(%ebp)  # M[R[%ebp-8]] $\leftarrow$ result
Recall Memory Operands

- displacement(%reg)
  - e.g., addl %eax, -8(%ebp)

- IA32 allows a memory operand as the source or destination, but NOT BOTH
  - One of the operands must be a register

- This would not be allowed:
  - addl -4(%ebp), -8(%ebp)
  - If you wanted this, movl one value into a register first
Relevant XKCD

I COULD RESTRUCTURE
THE PROGRAM'S FLOW
OR USE ONE LITTLE
'GOTO' INSTEAD.

EH, SCREW GOOD PRACTICE.
HOW BAD CAN IT BE?
goto main_sub3;

*COMPILE*

...
Control Flow

• Previous examples focused on:
  – data movement (movl)
  – arithmetic (addl, subl, orl, negl, sall, etc.)

• Up next: Jumping!

(Changing which instruction we execute next.)
**Unconditional Jumping / Goto**

```c
int main() {
    int a = 10;
    int b = 20;
    goto label1;
    a = a + b;

    label1:
    return;
}
```

A label is a place you **might** jump to.

Labels ignored except for goto/jumps.

(Skipped over if encountered)

```c
int x = 20;
L1:
    int y = x + 30;
L2:
    printf("%d, %d\n", x, y);
```
Unconditional Jumping / Goto

```c
int main() {
    int a = 10;
    int b = 20;
    goto label1;
    a = a + b;

label1:
    return;
}
```

```assembly
push    %ebp
mov     %esp, %ebp
sub     $16, %esp
movl    $10, -8(%ebp)
movl    $20, -4(%ebp)
jmp     label1
movl    -4(%ebp), $eax
addl    $eax, -8(%ebp)
movl    -8(%ebp), %eax
label1:
leave
```
Unconditional Jumping

Usage besides GOTO?

```assembly
push   %ebp
mov    %esp, %ebp
sub    $16, %esp
movl   $10, -8(%ebp)
movl   $20, -4(%ebp)
jmp    label1
movl   -4(%ebp), $eax
addl   $eax, -8(%ebp)
movl   -8(%ebp), %eax
label1:
leave
```
Unconditional Jumping

- Usage besides GOTO?
  - infinite loop
  - break;
  - continue;
  - functions (handled differently)

- Often, we only want to jump when something is true / false.

- Need some way to compare values, jump based on comparison results.

```assembly
push %ebp
mov %esp, %ebp
sub $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
jmp label1
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
label1:
leave
```
Condition Codes (or Flags)

• Set in two ways:
  1. As “side effects” produced by ALU
  2. In response to explicit comparison instructions

• IA-32, condition codes tell you:
  – If the result is zero (ZF)
  – If the result’s first bit is set (negative if signed) (SF)
  – If the result overflowed (assuming unsigned) (CF)
  – If the result overflowed (assuming signed) (OF)
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:
- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

Current stack top
- Instruction pointer (PC)
- Condition codes
Instructions that set condition codes

1. Arithmetic/logic side effects (addl, subl, orl, etc.)

2. CMP and TEST:

   **cmpl b,a** like computing **a-b** without storing result
   - Sets OF if overflow, Sets CF if carry-out,
   - Sets ZF if result zero, Sets SF if results is negative

   **testl b,a** like computing **a&b** without storing result
   - Sets ZF if result zero, sets SF if a&b < 0
   - OF and CF flags are zero (there is no overflow with &)

Which flags would this `subl` set?

- Suppose `%eax` holds 5, `%ecx` holds 7

`subl $5, %eax`

If the result is zero (ZF)
If the result’s first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF
Which flags would this `cmp` set?

- Suppose `%eax` holds 5, `%ecx` holds 7

```assembly
cmp %ecx, %eax
```

- If the result is zero (ZF)
- If the result’s first bit is set (negative if signed) (SF)
- If the result overflowed (assuming unsigned) (CF)
- If the result overflowed (assuming signed) (OF)

A. ZF  
B. SF  
C. CF and ZF  
D. CF and SF  
E. CF, SF, and CF
## Conditional Jumping

- Jump based on which condition codes are set

<table>
<thead>
<tr>
<th>Jump Instructions: (fig. 3.12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>You do not need to memorize these.</td>
</tr>
</tbody>
</table>

### Jump Instructions:

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Example Scenario

```c
int userval;
scanf("%d", &userval);
if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
...
```

- Suppose user gives us a value via `scanf`
- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10
How would we use jumps/CCs for this?

int userval;
scanf("%d", &userval);

if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}

...
How would we use jumps/CCs for this?

```c
int userval;
scanf("%d", &userval);
if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
```

Assume `userval` is stored in `%eax` at this point.

(B) `cmpl $42, %eax
   jne L2
   L1:
   subl %10, %eax
   jmp DONE
```

(A) `cmpl $42, %eax
   je L2
   L1:
   addl %5, %eax
   jmp DONE
```

(C) `cmpl $42, %eax
   jne L2
   L1:
   addl %5, %eax
   jmp DONE
```

L2:
   subl %10, %eax
DONE:
...

...
Loops

• We’ll look at these in the lab!
Summary

• ISA defines what programmer can do on hardware
  – Which instructions are available
  – How to access state (registers, memory, etc.)
  – This is the architecture’s *assembly language*

• In this course, we’ll be using IA-32
  – Instructions for:
    • moving data (movl)
    • arithmetic (addl, subl, imull, orl, sall, etc.)
    • control (jmp, je, jne, etc.)
  – Condition codes for making control decisions
    • If the result is zero (ZF)
    • If the result’s first bit is set (negative if signed) (SF)
    • If the result overflowed (assuming unsigned) (CF)
    • If the result overflowed (assuming signed) (OF)