Introduction to
Machine/Assembler Language

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
COMP 40 term roadmap

- Ramp up your Programming Skills
- Big programs that teach you abstraction, pointers, locality, machine representations of data
- Intel Assembler Programming *The Bomb!*
- Building Useful Applications in your Language
- Building a Language Processor on your Emulator
- Emulating your own hardware in software
Goals for today – learn:

- How do computers *compute*?
- High level languages (C) vs. low-level (machine/ASM)
- Assembler vs. machine code: what’s the difference?
- Registers vs. main memory
- Why assembler programming is:
  - Interesting
  - Important
  - Fun!
Goals for today – learn:

- Why assembler programming is important & fun!
- Registers vs. main memory
- High level languages (C) vs. low-level (machine/ASM)
  - Data and Program
- Assembler vs. machine code: what’s the difference?
- How do computers compute?
Introducing Assembler and Machine Language
Why assembler / machine code matter

- Necessary for understanding how computers work
- Necessary for understanding
  - Compilers
  - Computer viruses
  - Anti-virus tools
  - CPU design
  - Device drivers
- Essential for debugging and performance tuning
- A wonderful art...lots of fun!
- History: how all programs were written for decades
Registers, Memory and Cache
Why Registers?

- **Fastest storage – built into CPU**
- **Compact addressing** *directly from instructions*
  - How many bits does it take to choose one of 16 registers?
  - *The typical modern machine does most of its computing in registers!*
  - To compute on data in memory…first load it into a register!

- **Comparing to L1 cache**
  - Both are fast: registers typically somewhat faster
  - Cache used *transparently* when accessing memory
  - *Registers completely integrated with CPU*
  - *Compiler directly controls placement of data in registers*
Simple view of a computer

- Registers
- Cache
- CPU
- Memory
  Programs and Data Stored as Bit Patterns
The 32 bit Intel architecture has 8 registers of 32 bits each.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</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>%esx</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>
</tbody>
</table>
IA 32 General Purpose Registers

Register names reflect traditional uses...a few still do have special roles

<table>
<thead>
<tr>
<th>31</th>
<th>0</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>%esx</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>
</tbody>
</table>
Register names reflect traditional uses...a few still do have special roles.

%esp typically used for stack pointer.
Register names reflect traditional uses...a few still do have special roles

%eax typically used to accumulate and return function results
Before there were IA32 machines, there were 16 bit machines....
...the 16 bit registers are simulated within the 32 bit ones

<table>
<thead>
<tr>
<th>侵权</th>
<th>侵权</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
And within that some instructions can update individual bytes

You want to read Bryant and O’Hallaron for the details!
What we’ve seen so far – IA 32

- 8 registers with names like `%eax`
- For IA32, each holds 32 bits
- **This is where data will be placed for:**
  - Arithmetic
  - Shifting / masking
  - Calculating pointers for addressing

- **Smaller registers from 16 & 8 bit Intel architectures implemented as part of 32 bit registers**
What we’ve seen so far – IA 32

- 8 registers with names like `%eax`
- For IA32, each holds 32 bits
- This is where data will be placed for:
  - Arithmetic
  - Shifting / masking
  - Calculating pointers for addressing
- Smaller registers from 16 bit Intel architectures implemented *as part of* 32 but registers
- But...we run on IA-64 / AMD64 / x86-64 machines!
General Purpose Registers – 64 bit

%rax
%eax
%ax
%ah
%al
General Purpose Registers – addressing all 64 bits

```
mov $123, %rax
```
General Purpose Registers – addressing 16 bits

```
%rax
%eax  %ax
%ah    $al

mov $123, %ax
```
General Purpose Registers – addressing 32 bits

mov $123,%eax
### X86-64 / AMD 64 / IA 64 General Purpose Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Width</th>
<th>Base</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-31</td>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31-0</td>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63-31</td>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31-0</td>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63-31</td>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31-0</td>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63-31</td>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31-0</td>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Additional Registers

- %r8
- %r9
- %r10
- %r11
- %r12
- %r13
- %r14
- %r15
Classes of AMD 64 registers

- **General purpose registers**
  - 16 registers, 64 bits each
  - Used to compute integer and pointer values
  - Used for integer call/return values to functions

- **XMM registers**
  - 16 Registers, 128 bits each – latest chips widen these to 256 bits (YMM)
  - Used to compute float/double values, and for parallel integer computation
  - Used to pass double/float call/return values

- **X87 Floating Point registers**
  - 8 registers, 80 bits each
  - Used to compute, pass/return long double
Classes of AMD 64 registers

- **General purpose registers**
  - 16 registers, 64 bits each
  - Used to compute integer and pointer values
  - Used for integer call/return values to functions

- **XMM registers**
  - 16 Registers, 128 bits each – latest chips widen these to 256 bits (YMM)
  - Used to compute float/double values, and for parallel integer computation
  - Used to pass double/float call/return values

- **X87 Floating Point registers**
  - 8 registers, 80 bits each
  - Used to compute, pass/return long double

We will focus mainly on integer, bitmap and pointer computation using the GPRs
History of Intel Architecture Machines
Brief history of Intel Architecture milestones

- **1978:** Intel 8086 – 16 bits 29K transistors
- **1985:** Intel 80386 – 32 bits 275K transistors
- **1995:** Intel Pentium Pro – new microarch 5.5M transistors
- **1999:** Intel Pentium III – 32 bits + SSE/SIMD 8.2M transistors
- **2003:** AMD Athlon – 64 bits ?? M transistors
  - New architecture known as x86-64 or AMD 64
  - *Backwards compatible with IA32*
Brief history of Intel Architecture milestones

- **1978**: Intel 8086 – 16 bits, 29K transistors
- **1985**: Intel 80386 – 32 bits, 275K transistors
- **1995**: Intel Pentium Pro – new microarchitecture, 5.5M transistors
- **1999**: Intel Pentium III – 32 bits + SSE/SIMD, 8.2M transistors
- **2003**: AMD Athlon – 64 bits, ? M transistors
  - New architecture known as x86-64 or AMD 64
  - *Backwards compatible with IA32*

Intel distracted working on incompatible Itanium

... AMD grabs de-facto ownership of “Intel” architecture instruction set evolution!
Brief history of Intel Architecture milestones

- **1978: Intel 8086** – 16 bits 29K transistors
- **1985: Intel 80386** – 32 bits 275K transistors
- **1995: Intel Pentium Pro** – new microarch 5.5M transistors
- **1999: Intel Pentium III** – 32 bits + SSE/SIMD 8.2M transistors
- **2003: AMD Athlon** – 64 bits ?? M transistors
  - New architecture known as x86-64 or AMD 64
  - *Backwards compatible with IA32*
- **2004: Intel Pentium 4E** – 64 bits 125M transistors
  - Intel caves and supports AMD 64 instruction set
Brief history of Intel Architecture milestones

- **1978: Intel 8086** – 16 bits 29K transistors
- **1985: Intel 80386** – 32 bits 275K transistors
- **1995: Intel Pentium Pro** – new microarch 5.5M transistors
- **1999: Intel Pentium III** – 32 bits + SSE/SIMD 8.2M transistors
- **2003: AMD Athlon** – 64 bits ?? M transistors
  - New architecture known as x86-64 or AMD 64
  - *Backwards compatible with IA32*
- **2004: Intel Pentium 4E** – 64 bits 125M transistors
  - Intel caves and supports AMD 64 instruction set
- **Today: Intel (Core i7) and AMD (e.g. Opteron) compete with same instruction set**
High Level Languages
vs.
Machine Instructions:

*How does the machine compute?*
Consider a simple function in C

```c
int times16(int i)
{
  return i * 16;
}
```

Building a machine to execute this *directly in hardware would be*:  

- **Difficult**
- **Slow**
- **Expensive**
- **Etc.**

For the first few decades of computing, nobody programmed this way at all!

All (almost all) digital computers from the earliest to modern execute very simple instructions encoded as bits.
Machine code (typical)

- **Simple instructions** – each does small unit of work
- Stored in memory
- *Bitpacked into compact binary form*
- Directly executed by transistor/hardware logic*

* We’ll show later that some machines execute user-provided machine code directly, some convert it to an even lower level machine code and execute that.
Here’s the machine code for our function

```
int times16(int i)
{
    return i * 16;
}
```

Remember:
This is what’s really in memory and what the machine executes!
Here’s the machine code for our function

```c
int times16(int i)
{
    return i * 16;
}
```

But what does it mean??

Does it really implement the times16 function?
Consider a simple function in C

```c
int times16(int i)
{
    return i * 16;
}
```

```
0:   89 f8 mov %edi, %eax
2:  c1 e0 04 shl $0x4, %eax
5:  c3 retq
```
Consider a simple function in C

```c
int times16(int i) {
    return i * 16;
}
```

Load i into result register %eax

```
0: 89 f8
2: c1 e0 04
5: c3
```

```assembly
mov %edi, %eax
shl $0x4, %eax
retq
```
Consider a simple function in C

```c
int times16(int i) {
    return i * 16;
}
```

Huh...

What language is this?

```
0: 89 f8  ← mov %edi, %eax
2: c1 e0 04 ← shl $0x4, %eax
5: c3 ← retq
```
Consider a simple function in C

```c
int times16(int i) {
    return i * 16;
}
```

Huh...

What language is this?

```
0:  89 f8  mov %edi, %eax
2:  c1 e0 04  shl $0x4, %eax
5:  c3  retq
```

ASAEMBLER LANGUAGE:

- Source code in ASCII, like C (typically written by people)
- Assembled into machine code (bits) by the assembler – produces .o file
- Remember, on our Linux systems the compiler actually produces assembler and then assembles that to make a .o
- You can see the generated assembler by using the –S option. DO IT!
Assembler vs. machine code

```c
int times16(int i)
{
    return i * 16;
}
```

Machine code

0: 89 f8
2: c1 e0 04
5: c3

Assembler language

```assembly
mov %edi,%eax
shl $0x4,%eax
retq
```

ASSEMBLER LANGUAGE:

- Source code in ASCII, like C (typically written by people)
- *Assembled* into machine code (bits) by the assembler – produces .o file
- Remember, on our Linux systems the compiler actually *produces assembler and then assembles that to make a .o*
- You can see the generated assembler by using the –S option. DO IT!
Consider a simple function in C

```c
int times16(int i)
{
    return i * 16;
}
```

Shifting left by 4 is a quick way to multiply by 16.
Consider a simple function in C

```c
int times16(int i) {
    return i * 16;
}
```

Return to caller, which will look for result in `%eax`

REMEMBER: you can see the assembler code for any C program by running gcc with the `-S` flag. Do it!!
How do we build a chip that does this calculation?
Very simplified view of computer
Very simplified view of computer

Instruction Fetcher

Instruction Decoder

Memory Interface

Registers

ALU

Cache

Memory

89 f8
c1 e0 04
c3
Instructions fetched and decoded
Instructions fetched and decoded

Arithmetic and Logic Unit executes instructions like add and shift, updating registers.
The MIPS CPU Architecture
Interpreting vs. Compiling
INTERPRETER

Software or hardware that does what the instructions say

COMPILER (translator)

Software that converts a program to another language

ASSEMBLER (translator)

Like a compiler, but the input assembler language is (mostly) 1-to-1 with machine instructions
INTERPRETER

Software or hardware that does what the instructions say

COMPILER (translator)

Software that converts a program to another language

ASSEMBLER (translator)

Like a compiler, but the input assembler language is (mostly) 1-to-1 with machine instructions
INTERPRETER

Software or hardware that does what the instructions say

COMPILER (translator)

Software that converts a program to another language

ASSEMBLER (translator)

Like a compiler, but the input assembler language is (mostly) 1-to-1 with machine instructions
What a computer is

- **A computer is a hardware interpreter** for machine language programs
- **Simple machines (RISC):**
  - Hardware directly interprets machine instructions
What a computer is

- A computer is a hardware interpreter for machine language programs

- **Simple machines (RISC):**
  - Hardware directly interprets machine instructions

- **Complex machines (CISC):**
  - The real computer handles only simple instructions
  - Hardware *converts* user’s complex machine instructions to simple instructions on the fly – then interprets
What a computer is

- A computer is a hardware interpreter for machine language programs

- Simple machines (RISC):
  - Hardware directly interprets machine instructions

- Complex machines (CISC):*
  - The real computer handles only simple instructions
  - Hardware converts user’s complex machine instructions to simple instructions on the fly – then interprets

* The true difference between RISC and CISC is a bit more subtle, but the spirit of this explanation is right
What a computer is

- A computer is a hardware interpreter for machine language programs

- Simple machines (RISC):
  - Hardware directly interprets machine instructions

- Complex machines (CISC): *
  - The real computer handles only simple instructions
  - Hardware converts user’s complex machine instructions to simple instructions on the fly – then interprets

- Software interpreters - emulators
  - You can write software programs that do the same thing as hardware computers
  - You will be writing an emulator for a simple CPU later in COMP 40!

* The true difference between RISC and CISC is a bit more subtle, but the spirit of this explanation is right
Virtual Machines

The C Language Virtual Machine

Usually compiled to assembler – occasionally interpreted

Or Python, Java, Ruby, etc.
Virtual Machines

The C Language Virtual Machine

Usually compiled to assembler – occasionally interpreted

The Intel IA64 Virtual Machine

The machine code a user gets to run
Virtual Machines

The C Language Virtual Machine

Usually compiled to assembler – occasionally interpreted

The Intel IA64 User-mode Virtual Machine

The machine code a user gets to run

Microcode

Low level code underneath a CISC machine
Virtual Machines

The C Language Virtual Machine

Usually compiled to assembler – occasionally interpreted

The Intel IA64 User-mode Virtual Machine

The machine code a user gets to run

Software emulator or hypervisor

Pretend to be a different machine or multiple machines

Hypervisors like VMWare and VM/370 create the illusion of multiple hardware machines on a single CPU