why you shouldn’t use assembly

• ... and for that matter, why for a lot of things you shouldn’t be using C either ...

• Corbató's Law:
  "The number of lines of code a programmer can write in a fixed period of time is the same independent of the language used."

why we’re doing assembly here

• learn how the machine works
• possibly faster code. one scenario:
  – write in C
  – profiler
  – tweak the assembly for parts the profiler shows
• what if no compiler?
• processor features not easily accessed by higher-level language

chapter 3 part 1

why we shouldn’t use assembly

• compilers generate pretty fast, efficient code
• tedious, easy to screw up
• not portable
the point

• we’re not trying to prepare you for a job doing assembly programming
• it’s about learning how computers work

how

• look at the assembly generated by GCC
• write some of our own assembly from scratch

warning: GCC and Cygwin

• no difference up until now
• better use GCC on Linux with this stuff
some terminology

- x86 name for the chips
- IA32 the name of the instruction set
  - IA = Intel Architecture

- note difference between:
  - architecture: what you need to know to program assembly -- instruction set, registers
  - microarchitecture: implementation
  - e.g., IA32 on non-Intel chips (e.g. AMD)

GAS

- assembler used by GCC
- differences with NASM
  - AT&T syntax
- be careful when reading the Intel manuals
  - operand order!

instruction set

- set of of operations that a processor supports
- examples
  - load x bytes from this address into register y
  - add what’s in register i to what’s in register j
- primitive stuff
- usually takes lots of these primitive ops to do something really useful

Other instruction sets

- IA32, Intel64 (x86-64 or x64), IA64
- SPARC
- ARM
- PowerPC
- MIPS
- Alpha
- JVM

- we’re using: IA32
  - not easiest, but popular
  - focusing on GCC output on Linux
Some History: Why should we care?

• Important things to take away from the history lesson in the chapter:
  – Moore’s law
  – Evolution of register names
  – Backward compatibility:
    • Goal: run progs compiled for earlier versions of chip
    • But: old baggage support features that new OS, compilers rarely use

<table>
<thead>
<tr>
<th>name</th>
<th>date</th>
<th>transistors</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>16-bit processor. DOS. 1MB address space. DOS allows 640K</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
<td>Windows</td>
</tr>
<tr>
<td>80386</td>
<td>1985</td>
<td>275K</td>
<td>32-bit registers. Flat addressing. Can run a Unix OS</td>
</tr>
<tr>
<td>80486</td>
<td>1989</td>
<td>1.9M</td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
<td></td>
</tr>
<tr>
<td>/MMX</td>
<td>1997</td>
<td>4.5M</td>
<td>instructions helpful for multimedia processing</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
<td>conditional move instructions</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
<td></td>
</tr>
<tr>
<td>Pentium IV</td>
<td>2001</td>
<td>42M</td>
<td></td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
<td></td>
</tr>
<tr>
<td>i7</td>
<td>2008</td>
<td>731M</td>
<td></td>
</tr>
</tbody>
</table>

Looking GCC assembly output

• gcc –S

• another possibly helpful switch:
  – -fverbose-asm

some tools

• compiler – GCC
• assembler – as aka gas
• linker – ld
• debugger – gdb
• disassembler - objdump
• profiler - gprof
famous quote

If automobiles had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside. —Robert X. Cringely

<table>
<thead>
<tr>
<th>name</th>
<th>date</th>
<th>transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2001</td>
<td>10M</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td>Itanium 2 Dual-Core</td>
<td>2006</td>
<td>1.7B</td>
</tr>
</tbody>
</table>

In parallel. IA64 chips.

From [Intel](http://download.intel.com/pressroom/kits/core7/images/Nehalem_Die_callout.jpg).

From [wikipedia](http://en.wikipedia.org/wiki/Moore%27s_law).
8086 Register

- Example general purpose register
- 8 general purpose

386 registers

aside: goals then and now

- big goal then:
  - cram as much processing power on a chip possible
- big goal now:
  - cram as much processing power on a chip possible
  - don’t use so much power
  - some environments: keep the chip small
  - think cell phone

How bad is your electric bill?

<table>
<thead>
<tr>
<th>Company</th>
<th>Servers</th>
<th>Electricity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>eBay</td>
<td>16K</td>
<td>$0.6 \times 10^5$ MWh</td>
<td>$\approx 3.7M$</td>
</tr>
<tr>
<td>Akamai</td>
<td>40K</td>
<td>$1.7 \times 10^5$ MWh</td>
<td>$\approx 10M$</td>
</tr>
<tr>
<td>Rackspace</td>
<td>50K</td>
<td>$2 \times 10^5$ MWh</td>
<td>$\approx 12M$</td>
</tr>
<tr>
<td>Microsoft</td>
<td>&gt;200K</td>
<td>&gt;$6 \times 10^5$ MWh</td>
<td>&gt;36M</td>
</tr>
<tr>
<td>Google</td>
<td>&gt;500K</td>
<td>&gt;$6.3 \times 10^5$ MWh</td>
<td>&gt;38M</td>
</tr>
<tr>
<td>USA (2006)</td>
<td>10.9M</td>
<td>610 $\times 10^5$ MWh</td>
<td>$4.5B$</td>
</tr>
<tr>
<td>MIT campus</td>
<td>2.7 $\times 10^5$ MWh</td>
<td>$62M$</td>
<td></td>
</tr>
</tbody>
</table>

from, Cutting the Electric Bill for Internet-Scale Systems, Qureshi et al, CCR 2009.
Registers:
- General purpose
- PC (EIP IA32, RIP x86-64)
- Condition codes

64-bit general purpose registers

32-bit "Register File"

CPU

Memory
but wait, there’s more in x86-64

<table>
<thead>
<tr>
<th>register</th>
<th>common use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>accumulator, return</td>
</tr>
<tr>
<td>EBX</td>
<td>pointer to items in data segment</td>
</tr>
<tr>
<td>ECX</td>
<td>loop control</td>
</tr>
<tr>
<td>EDX</td>
<td>I/O pointer</td>
</tr>
<tr>
<td>ESI</td>
<td>src ptr for string ops</td>
</tr>
<tr>
<td>EDI</td>
<td>dst ptr for string ops</td>
</tr>
<tr>
<td>ESP</td>
<td>stack pointer</td>
</tr>
<tr>
<td>EBP</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

data types in Intel-speak

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1 byte</td>
</tr>
<tr>
<td>word</td>
<td>2 bytes</td>
</tr>
<tr>
<td>doubleword</td>
<td>4 bytes</td>
</tr>
<tr>
<td>quadword</td>
<td>8 bytes</td>
</tr>
<tr>
<td>double quadword</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

• so a word isn’t a word?

“general purpose” registers

<table>
<thead>
<tr>
<th>register</th>
<th>common use</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>EAX</td>
</tr>
<tr>
<td>RBX</td>
<td>EBX</td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
</tr>
<tr>
<td>RSI</td>
<td>ESI</td>
</tr>
<tr>
<td>RDI</td>
<td>EDI</td>
</tr>
<tr>
<td>RBP</td>
<td>EBP</td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
</tr>
</tbody>
</table>

64-bit Register File

<table>
<thead>
<tr>
<th>RAX</th>
<th>EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBX</td>
<td>EBX</td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
</tr>
<tr>
<td>RSI</td>
<td>ESI</td>
</tr>
<tr>
<td>RDI</td>
<td>EDI</td>
</tr>
<tr>
<td>RBP</td>
<td>EBP</td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
</tr>
</tbody>
</table>
simplest function ever

```
1 int sum()
2 {
3   int x=30;
4   int y=57;
5   int z=39;
6   return x+y+z;
7 }
```

gcc output of sum function

```
1 .text
2 .globl _sum
3 _sum:
4     pushl %ebp #
5     movl %esp, %ebp #
6     subl $24, %esp #
7     movl $30, -20(%ebp) #, x
8     movl $57, -16(%ebp) #, y
9     movl $39, -12(%ebp) #, z
10    movl -16(%ebp), %eax # y, y
11    addl -20(%ebp), %eax # y, x, D.1509
12    addl -12(%ebp), %eax # z, D.1508
13    leave
14    ret
```

let’s try a full program

```
1 /* file summain.c */
2 int main(void)
3 {
4   int x=30;
5   int y=57;
6   int z=39;
7   return x+y+z;
8 }
```
most of the same on ARM

its assembly output

what to do with it

• assembling:
  – as -o summain.o summain.s

• linking:
  – ld -o sum main.o

same code on SPARC
What’s going on

- Assembler -- .s to .o
  - binary encoding of each instruction
  - connections missing for code in different files
- Linker – references between files
  - static linker – copies library code into binary
  - dynamic linker – linkage when program is run
another look at disassembled binary

```
sum: file format elf32-i386
Disassembly of section .text:
08048054 <_start>:
  bb 0e 00 00 00 mov $0xe,%ebx
  83 c3 1d add $0x1d,%ebx
  83 c3 0a add $0xa,%ebx
  b8 01 00 00 00 mov $0x1,%eax
  cd 80 int $0x80
```

- left – addresses of the instructions
- center – opcode + operands
- some instructions longer than others:
  - more frequently used instructions – shorter opcodes
  - some operations take more operands

what can we disassemble?

- any executable
- disassembler interprets bytes as assembly src
- no source code required

disassembly

- objdump –d filename
- produces assembly from binary file
- works for .o file or full executable

disassembly example

```
/* file AnotherSimpleSum.s */
.section .text
.globl _start
_start:
  /* pushl %ebp */
  movl %esp, %ebp
  /* movl %esp, %ebp */
  movl $14, %ebx
  addl $29, %ebx
  addl $10, %ebx
  movl $1, %eax
  int $0x80
```

sum: file format elf32-i386
disassembled binary

```
08048054 <_start>:
  bb 0e 00 00 00 mov $0xe,%ebx
  83 c3 1d add $0x1d,%ebx
  83 c3 0a add $0xa,%ebx
  b8 01 00 00 00 mov $0x1,%eax
  cd 80 int $0x80
```

very exciting program

```c
#include <unistd.h>

int main(int argc, char **argv) {
    _exit(17);
}
```

* can see the exit value in the shell by doing:
  – echo $?

assembly equivalent

```c
#include <unistd.h>

int main(int argc, char **argv) {
    _exit(17);
}
```

/* file exitonly.s

only slightly modified from Programming from the Ground Up. */

.section .data

.section .text

.globl _start

_start:
    movl $1, %eax /* this is the linux kernel command */
    /* number (system call) for exiting */
    /* a program */
    movl $0, %ebx /* this is the status number we will */
    /* return to the operating system. */
    /* Change this around and it will */
    /* return different things to */
    /* echo $? */
    int $0x80 /* this wakes up the kernel to run */
    /* the exit command */

aside: can do this in GDB too

- disas – disassembles current function
- disas sum – disassemble the sum function
- disas addr – disassemble function at addr
- disas ad₁ ad₂ – disas between addr ad₁, ad₂

very exciting program

```c
#include <unistd.h>

int main(int argc, char **argv) {
    _exit(17);
}
```
what are the syscalls in my OS?

- in Linux,
  - man syscalls
    - seems to work on the Fedora boxes in the lab
      but not my Ubuntu machine
  - unistd.h lists them:

```c
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod ...
```

Another syscall example.

```c
#include <stdio.h>

void print_question_printf();

int main(void)
{
    print_question_printf();
    return 0;
}

void print_question_printf()
{
    char *str = "Who lives in a pineapple under the sea?\n";
    printf("%s", str);
}
```

syscalls

- how we get services from the OS
- important idea
- more next semester

what happens during a syscall?

1. interrupt pin goes high during current instruction
2. control to interrupt handler
3. interrupt handler runs
4. handler returns to next instruction
C Library and system calls

- What’s the relationship between the functions in the C Library and system calls exported by an OS?

- Tool: strace
  – *strace program*
  runs *program* and prints the system calls executed

---

**same thing using write system call**

```c
#include <unistd.h>
#include <string.h>

void print_question_write();

int main(void) {
    print_question_write();
    return 0;
}

void print_question_write() {
    char *str = "Who lives in a pineapple under the sea?\n";
    write(STDOUT_FILENO, str, 40);
}
```

---

**strace on the write version**

```c
execve("./print_question_write", ["./print_question_write"], [/* 39 vars */]) = 0
brk(0) = 0xcc0000
...
write(1, "Who lives in a pineapple under the sea?\n") = 40
...
```

---

**Almost the same**

```c
.equ STDOUT, 1
.equ MSG_LEN, 40
.equ WRITE_SYSCALL, 4

.section .data
str:
.ascii "Who lives in a pineapple under the sea?\n"

.section .text
.globl _start
_start:
/* write */
movl $WRITE_SYSCALL, %eax
movl $STDOUT, %ebx
movl $str, %ecx
movl $MSG_LEN, %edx
int $0x80
/* exit */
movl $1, %eax
movl $0, %ebx
int $0x80
```

---
binary compatibility

- Linux runs on my Intel desktop
- Windows runs on my Intel desktop
- Why can’t I take my Windows binaries and run them on Linux and vice versa?

for now: old program as template

```c
/* file exitonly.s */
only slightly modified from Programming from the Ground Up.

.globl _start
_start:
    movl $1, %eax /* this is the linux kernel command */
    movl $0, %ebx /* this is the status number we will */
    int $0x80 /* this wakes up the kernel to run */
```

strace on the printf version

```c
execve("./print_question_printf", ["./print_question_printf", /* 39 vars */], brk(0) = 0x1d81000)

write(1, "Who lives in a pineapple under t"..., 40)
```

So what does strace tell us?

What does `strace` tell us about the relationship between the C library function calls and syscalls provided by Linux (or any other OS)?
another simple program

```
.section .data
.section .text
.globl _start
_start:
    movl $20, %ebx /* %ebx=20 */
    addl $30, %ebx /* %ebx+=30 */
    movl $1, %eax /* exit syscall number */
    int $0x80 /* this wakes up the kernel to run */
    /* the exit command */
```