CSCE 212H, Spring 2008
Lab Assignment 3: Assembly Language
Assigned: Feb. 7, Due: Feb. 14, 11:59PM

February 7, 2008

1 Overview

The purpose of this assignment is to introduce you to the assembly language model of the Pentium processors and to tools for analysis of programs at that level, disassemblers and debuggers.

The portions of this assignment are:

1. Registers of the Pentium processor
2. Set up for lab3
3. Disassembling Code
    - objdump
    - Gdb GUI under CYGWIN
4. Condition Codes and Control Flow
5. Decompiling - from assembly to C
6. Logistics - submitting this lab

2 Registers of the Pentium processor

The register names for the Pentium are modification of earlier names for registers of the Intel 8080. In the 8080 there were 8 bit registers: A, B, C, D, E, H, L and 16 bit registers SP (Stack Pointer) and PC (Program Counter). For certain operations pairs (HL, DE) of registers were used for 16 bit operations.

Flags	A
B	C
D	E
H	L
SP	
PC	

Intel 8080 Registers

As the Intel processors grew up, 8 bit registers became 16 bits and the name A (accumulator) became AH (accumulator high) and AL (accumulator low). Then 16 bits were extended to 32 bit registers and an 'E' for Extended was prepended to the names.

31 16	15 8	7 0	Full Name
	AH	AL	EAX
	BH	BL	EBX
	CH	CL	ECX
	DH	DL	EDX
	SI		ESI
	DI		EDI
	BP		EBP
	IP		EIP
	SP		ESP
	Flags		EFlags

Basic Registers of Intel 80x86/Pentium

There are additional collections of registers: the segment select registers, the Multi-Media Extension (MMX) group and $\cdots$ that we will discuss later in the semester.

- EAX - sometimes still referred to as the accumulator. It can be used in 32-bit EAX, 16-bit AX, and 8-bit chunks AH and AL.
- EBX - the base register. It can hold the base address of data structures.
- ECX - the Counter register, has special role with loop counters.
- EDX - the Data register, is used in I/O operations and also in integer multiplications and divisions.
- ESI - the Source Index register.
- EDI - the Destination Index register.
- EBP - the Base Pointer register.
- EIP - the Instruction Pointer, holds the address of the next instruction to be executed.

- ESP - the Stack Pointer, holds the address of the next avaiable item on the stack.
- EFlag - the CPU status flags.

3 Setup for Lab3

Check your engineering domain email and save the attachment as "Lab3.tar". The next step is to transfer this to the correct folder so that CYGWIN can access it. The use the command

```
tar  xvf  Lab3.tar
```

to unpack the files. It should contain the files:

- sumSquares.c
- testSumSquares.c
- Makefile
- maxp7.c
- while.c
- switch.c

4 Disassembling Code

In this section we will examine tools for investigating the behavior of executable programs. In particular we will start with disassemblers. A disassembler examines the text of an executable program and generates the assembly language program.

Objdump -d

The first tool that we examine is "objdump." Executing objdump with no arguments will list the various command line options that objdump expects. In the directory Lab3 start by executing the commands:

```
gcc -g -c sumSquares.c
gcc -g -c testSumSquares.c
objdump -d sumSquares.o
```

The ouput of this last command should look something like:

```
sumSquares.o:     file format pe-i386

Disassembly of section .text:

00000000 <_sumSquares>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp,%ebp
   3: 83 ec 18              sub    $0x18,%esp
   6: c7 45 f8 00 00 00 00  movl   $0x0,0xfffffff8(%ebp)
   d: c7 45 fc 01 00 00 00  movl   $0x1,0xfffffffc(%ebp)
  14: 8b 45 fc              mov    0xfffffffc(%ebp),%eax
  17: 3b 45 08              cmp    0x8(%ebp),%eax
  1a: 7c 04                 jl     20 <_sumSquares+0x20>
  1c: eb 12                 jmp    30 <_sumSquares+0x30>
  1e: 89 f6                 mov    %esi,%esi
  20: 8b 45 fc              mov    0xfffffffc(%ebp),%eax
  23: 0f af 45 fc           imul   0xfffffffc(%ebp),%eax
  27: 01 45 f8              add    %eax,0xfffffff8(%ebp)
  2a: ff 45 fc              incl   0xfffffffc(%ebp)
  2d: eb e5                 jmp    14 <_sumSquares+0x14>
  2f: 90                    nop
  30: 8b 55 f8              mov    0xfffffff8(%ebp),%edx
  33: 89 d0                 mov    %edx,%eax
  35: eb 01                 jmp    38 <_sumSquares+0x38>
  37: 90                    nop
  38: 89 ec                 mov    %ebp,%esp
  3a: 5d                    pop    %ebp
  3b: c3                    ret
$
```

Each line shows the disassembly of a series of bytes. Note that the number of bytes in each instruction varies in this example from 1 to 7 bytes. The length of the instruction is a function of the opcode which is contained in the first couple of bytes of the instruction. The disassmebler decodes this information just as the CPU would to determine how long this instruction is.

Question 1 *In testSumSquares.o what is the assembly language statement that corresponds to the assignment "s=0x1234"?*

Question 2 *From the order of the bytes is this machine Big-Endian or Little-Endian?*

Note there is a lot of setup for the main that we will come back and discuss in a later lab. For the present we are not going to concentrate of this.

Now in sumSquares.o (gcc -g -c sumSquares.c) note that "n" would be a parameter and thus the corresponding argument value would be stored in the activation record of the function sumSquares.

Question 3 *In the address "0xfffffffc(%ebp)" what is the displacement? i.e. how many bytes from the value of the Base pointer does this address refer?*

Question 4 *In this code there are several "NOP" (no operation) instructions. When one of these is executed it effectively does nothing. Do the NOPs in this code slow it down any?*

Now compile sumSquares.c using the optimization flag (-O).

Question 5 *What major differences are there between the unoptimized compile and the optimized compile?*

1. *Where is the "sum" variable stored in each case?*
2. *Why would the optimized code be faster?*
3. *How long is each compiled function? In instructions and in bytes?*
4. *How many memory references are there in one iteration of the loop?*

Gdb GUI under CYGWIN

The Gnu Debugger (GDB) is a valuable tools for studying the execution of Unix programs. In the CYGWIN environment there is a GUI interface to gdb. First compile testSumSquares as follows:

```
gcc -g -O testSumSquares.c sumSquares.c -o test gdb test
```

There are three pulldowns across the top. The gdb starts off in the "SOURCE" mode. This may be changed to "ASSEMBLY", "MIXED" and "SRC+ASM." Try all of these to see the various views of the code. The other pulldowns are for the file you are dealing with and then for the function within that file.

Using the menu item "View" you can bring up windows that will show: the contents of registers, local variables, memory, breakpoints, etc.

Setting a Break Point

Set the file to "sumSquares.c", the function to "sumSquares" and the mode to "SRC+ASM." Highlight the statement "sum = sum + i*i;" and then click on the "RUN" icon (the running man). Bring up the registers under the View menu and you may have to rearrange windows a little to make it where you can see what you need to. The icon next to the RUN icon lets you single step, i.e., execute a single instruction.

Question 6 *What is held in register edx? in register eax?*

Question 7 *What is the address of "n"?*

5 Condition Codes

There are a collection of single bit registers called condition codes. Some of the most useful are given in the table below. They are set implicitly as a result of arithmetic operations and comparisons.

CF	Carry Flag	the most recent arithmetic operation generated a carry
ZF	Zero Flag	the most recent arithmetic operation yielded 0
SF	Sign Flag	the most recent arithmetic operation generated a negative
OF	Overflow Flag	the most recent arithmetic operation caused a two's complement overflow

The SetX instructions can then be used to save the values of flags in a byte.

```
int gt (int x, int y)
{
  return x > y;
}

translates to

movl 12(%ebp),%eax # eax = y
cmpl %eax,8(%ebp) # Compare x : y
setg %al # al = x > y
movzbl %al,%eax # Zero rest of %eax
```

Control Flow

Compile maxp7.c using the flags:-g -O -c and disassemble the code.

Now compile maxp7.c again but this time using the flags: -g -O -S. This should produce the assembly language program max.s. But there is a lot more produced also.

Question 8 *Describe the other information that is supplied in the maxp7.s file.*

Now examine the code generated for while.c.

Question 9 *What does this function do? note the decrement.*

Now examine the code generated for switch.c.

Question 10 *What happens when there is no "break"?*

6 Decompiling

A disassembler translates object code back into assembly code and thus effectively reverses the action of the assembler. We can carry this one step further and translate assembly language program back into

C, effectively reversing the action of the compiler. Such a tool is called a decompiler. We are going to decompile some assembly language programs by hand.

The URL http://www.itee.uq.edu.au/ csmweb/decompilation/ethics.html has a few comments on the ethics or decompiling.

Now try your hand at decompiling 3.31 on page 232 and 3.33 on page 233 of your text.

7 CYGWIN

CYGWIN = $\cdots$,
is a Unix emulator that runs under Windows. It was developed by Red Hat and is distributed for free. There is a directory /usr/doc that contains documentation on a number of packages but perhaps the best documentation is on the web $\cdots$
Start by signing on to your engineering domain account and then executing CYGWIN from the programs menu.
If you get the message "command not found" in response to a standard Unix command then you probably have a "path" problem. The path is a list of directories, separated by colons, in which the system will look for the command. The path can be set as in

```
PATH=''/usr/local/bin:/usr/bin:/bin:$PATH''
```

This command is usually in a "setup" file. The system should have "/etc/profile" and maybe there is a ".profile" in your home directory. First lets do a test. From the bash window (CYGWIN runs a bash shell), do the commands

```
ls -a
env     // print the environment
```

The first of these should list all the files in your "HOME" directory. In particular it will tell if you have a .profile file. The second command will print the environment and in particular show the setting of the PATH variable.

Question 11 *Do you have a ".bash_profile" file?*

Question 12 *Are all of the directories /usr/local/bin, /usr/bin and /bin on your path?*

In either case add the variable FOO="bar" by adding 'the following to your .bash_profile file.

```
FOO=''bar''
export FOO
```

The you can either "exit" the bash shell and reopen another or use the "source .profile" command.

8 Secure Web Site

The department's secure web site (https://www.cse.sc.edu) allows acess to a number of functions and access to crucial information.

Logistics

All labs will be submitted electronically using "dropbox." If there are any corrections or modifications to assignments these will be sent out via email and posted to the website.

To turn your files in you should create a directory "Lab3", place all files in this directory then use the command

```
tar xvf Lab3.tar Lab3
```

in the parent directory to create the file "Lab3.tar".

9 Logistics

Place the following files in the Lab3 directory:

- Questions.txt // this should have answers to the questions
- c331.c // the answer to problem number 31 on page 232
- c333.c // the answer to problem number 33 on page 233

Then the following sequence of commands should be executed:

```
rm      Lab3.tar      // remove the handout tar file
make    clean         // to remove some of the extraneous files
cd      ..            // change to the parent directory
tar     cvf Lab3.tar  // create the tar file
                      // then dropbox using the secure website
```

The due date is Wednesday Feb 2.