mp2 Warmup Directions

Study the lecture notes on the tools and instruction set. Then follow along with this document. Make sure everything works for you as it is shown here and that you understand *everything*. Turn in your work on this “warmup” along with the rest of your MP2 assignment.

Here's your first snippet of assembler. It is written in i386-as using 32 bit quantities as follows:

```
        movl $8, %eax
        addl $3, %eax
        movl %eax, 0x200
```

Let's see how to get this to run on a SAPC. Since it only uses registers and a memory location, it doesn't need any "startup" module. We just have to get these instructions into memory and execute them.

1. Put the gas assembler source code in a file called tiny.s

   tiny.s:

```
      movl $8, %eax
      addl $3, %eax
      movl %eax, 0x200
      int $3
     .end
```

I've added the "int $3" to trap back to Tutor at the end. Note also that I have used the pseudo-op .end to tell the assembler that this is the end of the code to be assembled.

2. Build an executable by running the assembler i386-as and then the loader i386-ld. Normally we would put these commands in a makefile, but here you want to become familiar with the individual steps.

```
---------------
ulab(1)% i386-as -o tiny.opc tiny.s
ulab(2)% i386-ld -N -Ttext 0x1000e0 -o tiny.lnx tiny.opc
---------------
```

Here the -N flag tells ld to make a self-sufficient, simple executable, and the "-Ttext 0x1000e0" tells it to start the code area at 1000e0, so that the code itself will start 0x20 bytes after that, at 100100. (There's a 0x20-byte header at the start)

3. We can look at the contents of tiny.lnx with the help of i386-objdump, which is available under the simpler name "disas" for disassembly. To get the hex contents as well as the disassembly, use "--full":

```
---------------
ulab(3)% disas --full tiny.lnx              (on UNIX, can look at .lnx)

tiny.lnx:     file format a.out-i386-linux

Contents of section .text:
 0000 b8080000 0083c003 a3000200 00cc9090  ................
Contents of section .data:
Disassembly of section .text:
00000000 movl   $0x8,%eax
00000005 addl   $0x3,%eax
00000008 movl   %eax,0x200
0000000d int3
0000000e nop
0000000f Address 0x10 is out of bounds.
---------------
```

This shows that the machine code in hex is

```
   b8080000 0083c003 a3000200 00cc9090
```

at offset 0000 in the .text area. (.text just means code.) Actually the last 9090 is off the end of the designated code. With the help of the offsets for each instruction, we can divide up the hex contents by instruction:

```
   b808000000   movl $0x8, %eax        at offset 0
   83c003       addl $0x3,%eax         at offset 5, so movl is 5 bytes of code
   a300020000   movl %eax, 0x200       at offset 8, so addl is 3 bytes
   cc           int $3                 at offset d, so movl is 5 bytes
   90           nop                    at offset e, so int is 1 byte
   90                                  at offset f, so nop is 1 byte
```

Later, we will cover how to encode instructions in bits, but for now it is interesting to find the 0x200 address hidden in the movl %eax, 0x200 instruction, and the 08 and 03 in the first two. Surprisingly, the 08 takes up 4 bytes but the 03 only one. The instruction set is optimized to be able to add small numbers into registers very quickly. The instruction size is important to speed because each instruction must be read out of memory before it can be executed.

4. We download and run tiny.lnx, executing one instruction at a time to see how the registers change. To execute one instruction at a time, use the "t" command in Tutor, for "trace". To get started, set the EIP to 100100, pointing the CPU to address 100100 as the next instruction to execute.

```
---------------
ulab(4)% mtip -f tiny.lnx
For command help, type ~?
For help on args, rerun without args
Code starts at 0x100100
Using board # 3
(hit CR here)

Tutor> ~downloading tiny.lnx
Calling loadprog()
.Done.

Tutor> md 100100                       Look at the code: same as above
00100100    b8 08 00 00 00 83 c0 03 a3 00 02 00 00 cc 90 90 ................
Tutor> rd
EAX=0000000b EBX=00009e00   EBP=000578ac
EDX=00101b88 ECX=00101bac   ESP=003ffff0
ESI=00090800 EDI=00101d5c   EIP=0010010d
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
Tutor> md 200                          Check target area using md or mdd
```

```
00000200    0b 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Tutor> ms 200 00000000                 Clear target area (8 0's for 32-bit
write)
Tutor> md 200                          Check again--OK
00000200    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

Tutor> rs eip 100100                   Set initial EIP to start addr
Tutor> t                               Trace: execute 1 instruction
Exception 1 at EIP=00100105: Debugger interrupt
Tutor> rd                              See EIP at 100105 (i.e. offset 5), and
EAX=00000008 EBX=00009e00   EBP=000578ac      8 now in EAX
EDX=00101b88 ECX=00101bac   ESP=003ffff0
ESI=00090800 EDI=00101d5c   EIP=00100105
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
Tutor> md 200                          Check target area: nothing yet
00000200    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Tutor> t                               Execute 2nd instruction
Exception 1 at EIP=00100108: Debugger interrupt
Tutor> rd                              See b in eax, eip to offset 8
EAX=0000000b EBX=00009e00   EBP=000578ac
EDX=00101b88 ECX=00101bac   ESP=003ffff0
ESI=00090800 EDI=00101d5c   EIP=00100108
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
Tutor> md 200                          Check target area: nothing yet
00000200    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Tutor> t                               Execute 3rd instruction
Exception 1 at EIP=0010010d: Debugger interrupt
Tutor> rd                              Only EIP has changed in regs
EAX=0000000b EBX=00009e00   EBP=000578ac
EDX=00101b88 ECX=00101bac   ESP=003ffff0
ESI=00090800 EDI=00101d5c   EIP=0010010d
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
Tutor> md 200                          Check mem--yes, 0b now in 0x200
00000200    0b 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Tutor> t                               Execute int $3
Exception 3 at EIP=0010010e: Breakpoint
Tutor> ~q
Quit handler: Leaving board #3
ulab(5)% exit
exit
```

5. Try out remote gdb on tiny: See also $pcex/gdb.script.

In one window: Use system 5, 6, 7, or 8 only here!

```
-------------------------------------------------------------
ulab(1)% mtip -b 5 -f tiny.lnx  (ASK FOR A SPECIFIC BOARD NUMBER, E.G. -b 5)
For command help, type ~?
For help on args, rerun without args
Using board # 5  (NOTE THAT BOARD NUMBER #5 GETS ASSIGNED VIA MTIP)
(hit CR here)

Tutor> ~d
Code starts at 0x100100
Calling loadprog()
.Done.

Download done, setting EIP to 100100.
Tutor> gdb
```

```
Setting gdb dev to COM1, starting gdb (CTRL-C to abort).
                             <---just let it hang here
--------------------------------------------------------------
In another window:
--------------------------------------------------------------
Script started on Wed Feb 16 10:55:17 2000

ulab(1)%
ulab(1)% i386-gdb tiny.lnx
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15.1 (sparc-sun-sunos4.1.3 --target i386-linuxaout),
Copyright 1995 Free Software Foundation, Inc...(no debugging symbols found)...
(gdb) tar rem /dev/remgdb5   (SET THE LAST DIGIT HERE BASED ON BOARD ASSIGNED)
Remote debugging using /dev/remgdb5
0x100100 in tiny.opc ()
(gdb) i reg
eax            0xb      11
ecx            0x6a894  436372
edx            0x0      0
ebx            0x9e00   40448
esp            0x578a8  0x578a8
ebp            0x578ac  0x578ac
esi            0x90800  591872
edi            0x51ffc  335868
eip            0x100100 0x100100
ps             0x302    770
cs             0x10     16
ss             0x18     24
ds             0x18     24
es             0x18     24
fs             0x18     24
gs             0x18     24
(gdb) x/x 0x200
0x200:    0x00000abc            <--old contents of memory at 0x200
(gdb) set *(int *)0x200 = 0     <--how to "ms" with gdb
(gdb) x/x 0x200                 <--check results
0x200:      0x00000000
(gdb) set $eip = 0x100100       <--to run from start
(gdb) x/4i 0x100100             <--examine 4 instructions
0x100100 <tiny.opc>:    movl   $0x8,%eax
0x100105 <tiny.opc+5>:  addl   $0x3,%eax
0x100108 <tiny.opc+8>:  movl   %eax,0x200
0x10010d <tiny.opc+13>: int3
(gdb) b *0x100105              <--set breakpoint at 2nd instruction
Breakpoint 1 at 0x100105
(gdb) c                        <--continue from 0x100100
Continuing.

Breakpoint 1, 0x100105 in tiny.opc ()
(gdb) i reg
eax            0x8      8
ecx            0x6a894  436372
edx            0x0      0
ebx            0x9e00   40448
esp            0x578a8  0x578a8
```

```
ebp            0x578ac  0x578ac
esi            0x90800  591872
edi            0x51ffc  335868
eip            0x100105 0x100105
ps             0x216    534
cs             0x10     16
ss             0x18     24
ds             0x18     24
es             0x18     24
fs             0x18     24
gs             0x18     24
(gdb) b *0x100108
Breakpoint 2 at 0x100108
(gdb) c
Continuing.

Breakpoint 2, 0x100108 in tiny.opc ()
(gdb) i reg
eax            0xb      11
ecx            0x6a894  436372
edx            0x0      0
ebx            0x9e00   40448
esp            0x578a8  0x578a8
ebp            0x578ac  0x578ac
esi            0x90800  591872
edi            0x51ffc  335868
eip            0x100108 0x100108
ps             0x202    514
cs             0x10     16
ss             0x18     24
ds             0x18     24
es             0x18     24
fs             0x18     24
gs             0x18     24

(gdb) b *0x10010d
Breakpoint 3 at 0x10010d
(gdb) c
Continuing.

Breakpoint 3, 0x10010d in tiny.opc ()

(gdb) i reg
eax            0xb      11
ecx            0x6a894  436372
edx            0x0      0
ebx            0x9e00   40448
esp            0x578a8  0x578a8
ebp            0x578ac  0x578ac
esi            0x90800  591872
edi            0x51ffc  335868
eip            0x10010d 0x10010d
ps             0x302    770
cs             0x10     16
ss             0x18     24
ds             0x18     24
es             0x18     24
```

```
fs             0x18     24
gs             0x18     24
(gdb) x/x 0x200
0x200:      0x0000000b
(gdb) q
The program is running.  Quit anyway (and kill it)? (y or n) y
ulab(2)% exit
exit

script done on Wed Feb 16 11:01:37 2000
```

To everyone who may encounter this problem and ask:

Question: Why I am I getting these error messages?

```
u18(9)% cat tiny.s
# tiny.s
# mp2 Warmup

   movl $8, %eax
   addl $3, %eax
   movl %eax, 0x200
   int $3
  .end

u18(10)% i386-as -o tiny.opc tiny.s
tiny.s: Assembler messages:
tiny.s:4: Error: Rest of line ignored. First ignored character valued 0xd.
tiny.s:5: Error: invalid character (0xd) in second operand
tiny.s:6: Error: invalid character (0xd) in second operand
tiny.s:7: Error: invalid character (0xd) in second operand
tiny.s:8: Error: invalid character (0xd) in first operand
tiny.s:9: Error: Rest of line ignored. First ignored character valued 0xd.
```

Answer:

You must have used an editor such as notepad on your PC locally to create the .s file and used file transfer to put it on the ulab system. Notepad has put a carriage return character 0x0d at the end of each line in addition to the normal UNIX end of line character 0x0a.

Here is a dump of the ASCII characters that are in your source file:

```
u18(56)% od -x tiny.s
0000000 2320 7469 6e79 2e73 0d0a 2320 4761 6c69
0000020 6e61 204f 736d 6f6c 6f76 736b 6179 610d
0000040 0a23 206d 7032 2057 6172 6d75 700d 0a0a
0000060 2020 206d 6f76 6c20 2438 2c20 2565 6178
0000100 0a20 2020 6164 646c 2024 332c 2025 6561
0000120 780a 2020 206d 6f76 6c20 2565 6178 2c20
0000140 3078 3230 300a 2020 2069 6e74 2024 330a
0000160 2020 2e65 6e64 0a00 0000167
u18(57)%
```

Notice the 0d0a character sequence that occurs at the end of each line.

The assembler is not ignoring the carriage return character 0x0d at the end of each line. I was not aware of this as a problem that would occur with files transferred from a PC and i386-as, but it is easy to fix.

You can use a UNIX editor such as vi to remove the carriage return characters
OR
you can run your file through the following simple C program to clean out the carriage return characters.

```
/* clean.c: remove ASCII carriage return characters from a file
   bob wilson
   10/5/2004
   invoke this program on stdin to clean out carriage return chars
   from the input file and write the new file out on stdout

   compile the program as:

        gcc -o clean clean.c

   execute the program command as:

        clean <xxx.s >newxxx.s      where xxx is your file name
        mv newxxx.s xxx.s           overwrite the original file
*/

#include <stdio.h>

int main (void)
{
        int c;
        while ((c = getchar()) != EOF)
           if(c != 0x0d)
               putchar(c);
        return 0;
}
```

Doing either of the above should take care of your problem.

- Bob Wilson