Assembly Language Programming

Some useful instructions

- `imull`: Performs signed multiplication and stores the result in the second operand. If the second operand is left out, it is assumed to be %eax, and the full result is stored in the double-word %edx:%eax.
- `idivl`: Performs unsigned division. Divides the contents of the double-word contained in the combined %edx:%eax registers by the value in the register or memory location specified. The %eax register contains the resulting quotient, and the %edx register contains the resulting remainder.
- `cdq`: Converts the %eax word into the double-word consisting of %edx:%eax with sign extension. This is usually used before issuing an `idivl` instruction.

32-bit emulation

- Assembling:
  `as --32 program.s -o program.o`
- Linking:
  `ld -m elf_i386 program.o -o program`

Debugging

- `as --gstabs --32 program.s -o program.o`
- `ld -m elf_i386 program.o -o program`
- `gdb program`
- `Useful gdb instructions`
  - `b 1`: Put a break at line number 1.
  - `r`: Run the program.
  - `p $eax`: Print the value of %eax register.

Functions (Revision) I

```
1  # Recursive implementation of factorial function
2
3  .section .data
4
5  .section .text
6
7  .globl _start
8
9  # This is needed because we need to call factorial
10 # from other programs
11 .globl factorial
12
13
14 .type factorial, @function
15 factorial:
16   pushl %ebp            #standard function stuff - we have to
17                         #restore %ebp to its prior state before
18                         #returning, so we have to push it
19
20   movl %esp, %ebp       #This is because we don t want to modify
21                         #the stack pointer, so we use %ebp.
22
23   movl 8(%ebp), %eax    #This moves the first argument to %eax
24                         #4(%ebp) holds the return address, and
25                         #8(%ebp) holds the first parameter
26
27   cmpl $1, %eax         #If the number is 1, that is our base
```

Functions (Revision) II

```
28                      #case, and we simply return (1 is
29                      #already in %eax as the return value)
30
31  je end_factorial
32
33  decl %eax           #otherwise, decrease the value
34
35  pushl %eax          #push it for our call to factorial
36
37  call factorial      #call factorial
38
39  movl 8(%ebp), %ebx  #%eax has the return value, so we
40                      #reload our parameter into %ebx
41
42  imull %ebx, %eax    #multiply that by the result of the
43                      #last call to factorial (in %eax)
44                      #the answer is stored in %eax, which
45                      #is good since that s where return
46                      #values go.
47
48  end_factorial:
49    movl %ebp, %esp   #standard function return stuff - we
50    popl %ebp         #have to restore %ebp and %esp to where
51                      #they were before the function started
52
53    ret               #return from the function (this pops the
54                      #return address too
```

Call Assembly functions in C I

In the assembly program:

- Add .globl to the function name

In the C program:

- Call the function, similar to any other C function

```c
#include <stdio.h>

int main() {
  printf("Factorial of 7 is %d.\n", factorial(7));
  return 0;
}
```

Call Assembly functions in C II

Compile using gcc:

```
gcc -m32 factorial_c.c factorial.s -o factorial
```

We need to use -m32 to emulate 32 bit architecture. gcc calls both assember (as) and linker (ld) for us. You might need to install 32-bit version of libc. On Ubuntu, use:

```
sudo apt-get install g++-multilib libc6-dev-i386
```

Call C functions in Assembly I

```
1  # PURPOSE: This program writes the message "hello world"
2  # and exits
3  #
4
5  .section .data
6
7  helloworld:
8    # In C, strings are terminated by null character
9    .ascii "Hello World\n\0"
10
11 .section .text
12 .globl _start
13
14 _start:
15   pushl $helloworld
16   call printf
17
18   pushl $0
19   call exit
```

Call C functions in Assembly II

- Assembling:
  `as --32 hello-world.s -o hello-world.o`
  `--32` is used to emulate 32 bit architecture.
- Linking:
  ```
  ld -m elf_i386 hello-world.o -o hello-world \
  -lc -dynamic-linker /lib/ld-linux.so.2
  ```
  - -m elf_i386 : Used for 32 bit emulation
  - -lc : Link the standard C library, named libc.so
  - -dynamic-linker: This builds the executable so that before executing, the operating system will load the program /lib/ld-linux.so.2 to load in external libraries and link them with the program.

Call C functions in Assembly III

Calling functions defined in a C program.

```c
#include <stdio.h>

// Function to add two numbers
int subtract(int a, int b) {
  return a - b;
}
```

Call C functions in Assembly IV

In the assembly program:

- Push the arguments in reverse order
- Return value is in the %eax register

```
1  # Purpose : Subtract two numbers using a C function
2
3  .section .data
4  outstr:
5    .ascii "Result is %d.\n\0"
6
7  .section .text
8  .globl _start
9  _start:
10   pushl $10      # We push arguments in reverse order, 10 is the second argument
11   pushl $15      # 15 is the first argument
12   call subtract  # This function is defined in the C file
13
14   pushl %eax     # The return value is stored in eax register
15                  # This is the second argument of printf
16   pushl $outstr  # First argument of printf
17   call printf    # printf is defined in standard C library
18
19   pushl $0       # Return value of program
20   call exit      # Another standard C function
```

Call C functions in Assembly V

- We first need to create object files from both assembly and C programs.

```
as --32 subtract.s -o subtract.o
gcc -m32 -c subtract_c.c -o subtract_c.o
```

- We use linker, with the -dynamic-linker option as in previous case.

```
ld subtract.o subtract_c.o -o subtract \
-m elf_i386 -lc \
-dynamic-linker /lib/ld-linux.so.2
```