Ananda Gunawardena

Lecture 21
C and Assembly

High level languages are designed to allow programmers to write programs in a way that is closer to the logical thinking of the program flow. However, programming in assembly language requires programmers to understand the instruction set architecture of the machine in addition to the logical program flow. Writing a program in machine language or assembly language is like programming a microprocessor kit. It requires the understanding of low level details of how a machine may execute a set of instructions and the fetch-execute cycle among other things. Today most programmers don't deal directly with assembly language, unless the task requires direct interfacing with hardware. For example, a programmer may consider using an assembly language to write a device driver or optimize part of a game program using assembly code.

To understand the assembly code, Let us consider the simple code below.

```
#this is in the file first.s
.global main
main:
      movl $20, %eax
      movl $10, %ebx
      ret
```

The first line of the program is a comment. The .global assembler directive makes the symbol main visible to the linker. This line makes the program linked up with the C start up library. If we try to remove this line, then we get the following message

% gcc first.s
/usr/lib/crt1.o: In function `_start':
: undefined reference to `main'
collect2: ld returned 1 exit status

The commands like movl $20, %eax means that the bit pattern of 20 is moved to register eax.

Since the inception of high level programming languages like cobol and fortran, there are only a handful of instances where a programmer need to write an assembly program. Modern compilers make it easier for us to develop, test and debug programs. Compilers convert high level programs written in a language like C into assembly code. Using the GNU C compiler –S option, we can generate the assembly code for a source code. For example, consider the program **simple.c**

#include <stdio.h>

```
int main(){
 int x=10,y=15;
 return 0;
}
```

The purpose of the above program is to define and initialize two variables x and y. When the program is compiled with **–S** option

- **gcc – S first.c**

it produces the assembly code file **simple.s** as shown below.

```
        .file   "first.c"
        .text
.globl main
        .type   main, @function

main:
        pushl   %ebp            -- save a copy of ebp in stack
        movl    %esp, %ebp       -- move esp to ebp
        subl    $8, %esp         -- reserve space on stack for two values
        andl    $-16, %esp       -- alignment operation
        movl    $0, %eax         -- move 0 to register eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    $10, -4(%ebp)   -- move 10 to saved space
        movl    $15, -8(%ebp)   -- move 15 to saved space
        movl    $0, %eax        -- prepare to return 0 from main
        leave
        ret         .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.4.6"
```

The intent here is to give some level of understanding of how assembly code works. There are traces of the initialization of x and y in the code as well as many uses of esp (stack pointer) and ebp (base pointer) references. The l at the end of each instruction indicates that we are using opcode that works with 32-bit operands. The registers are indicated by % in front and -4(%ebp) for example, indicates a reference to ebp-4 location. For example,

```
movl    $10, -4(%ebp)
indicates moving the value 10 to ebp-4.
```

Let us look at another example of a C program converted into assembly code.

```
#include <stdio.h>
int main(){
 printf("hello world\n");
 return 0;
}
```

// Assembly code

```
.LC0:
        .string     "hello world\n"
```

```
        .text
.globl main
        .type main, @function
main:
       pushl %ebp
       movl  %esp, %ebp
       subl  $8, %esp
       andl  $-16, %esp
       movl  $0, %eax
       addl  $15, %eax
       addl  $15, %eax
       shrl  $4, %eax
       sall  $4, %eax
       subl  %eax, %esp
       movl  $.LC0, (%esp)
       call  printf
       movl  $0, %eax
       leave
       ret
```

In this course, you are not required to understand a real assembly language or program in assembly code, but understand the concepts behind assembly programs like handling registers, stacks, branch instructions, fetch-execute cycle at a very high level. In later courses like systems programming (15-213), you will be exposed to real assembly language. In this short introduction we only expect to understand how to interpret simple assembly programs, how to assemble programs into object code and how to link them up with a C program.

Looping with Assembly

Unlike high level languages, assembly language does not have any direct loop constructs. Instead a programmer makes use of command like **je** (jump equal) to test for loop exit condition. Here is a program to find the factorial of 4 and we will assemble and run this program

```
# factorial of 4
# in file factorial.s

      .LC0:
       .string "%d \n"
       .text
.global main
main:
       movl $4, %eax
       movl $1,  %ebx
L1:    cmpl $0, %eax
       je L2
       imul %eax, %ebx
       decl %eax
       jmp L1
L2:    movl    %ebx, 4(%esp)
      movl    $.LC0, (%esp)
       call    printf
      movl    $0, %eax
```

```
leave
ret
```

The program correctly prints out the factorial of 4.
Caution: This program does not contain all assembly directives needed. May segfault.

Subroutines

A subroutine is a piece of code that is designed to perform a subtask in the program. Subroutines can have local variables, take arguments, and pass the results back to the calling program. Consider the following subroutine foo that return the value 4 to main.

```
# in file subroutine.c and subroutine.s
.globl foo
      .type foo, @function
foo:
      pushl %ebp
      movl  %esp, %ebp
      movl  $4, %eax
      popl  %ebp
      ret
.LC0:
      .string     "The value is %d \n"
      .text
.globl main
      .type main, @function
main:
      pushl %ebp
      movl  %esp, %ebp
      movl  $0, %eax
      subl  %eax, %esp
      call  foo
      movl  %eax, -4(%ebp)
      movl  -4(%ebp), %eax
      movl  %eax, 4(%esp)
      movl  $.LC0, (%esp)
      call  printf
      movl  $0, %eax
      leave
      ret
```

A subroutine is called with the instruction “call foo”. This instruction transfers the control of the program to foo subroutine. The two special registers ebp (base pointer) and esp (stack pointer) handles call and return mechanisms of subroutine calls. The values are returned to the calling program via register eax.

Stack

A stack is defined as a data structure that allows two operations, push and pop. Both operations are handled from the top of the stack. Imagine a stack of books, where you can add a book to the top (push) and remove a book from the top (pop). Stacks are useful data structures for saving the status of a program before branching out. For example, when a subroutine is called from main, the status of the environment is pushed into the stack. Upon return from the subroutine, the status variables are pop from the stack to restore the calling program status. The key operations of a stack are the push and pop. For example,

popl %eax

places the top element of the stack on the register eax and change the stack pointer esp. The stack pointer (esp) points to the top of the stack. To "decrement" the stack pointer esp we can simply add 4 as follows. Stacks may grow upward or downward. The instruction

addl $4, %esp

causes the stack pointer to point to the next 4 bytes of memory. Note that if the stack grows downward, then a push operation subtracts 4 from esp and pop operation adds 4 to esp.

Local Variables

C programs generally define local variables whose scope is the module where they are defined. The registers are used to manipulate the variables, but local variables are generally stored in the stack. Some variables, perhaps declared as register variables may hold space in a register, but most variables do not get register space but instead allocated space in the stack. Consider the following program that defines a local variable of value 10 and push that into the stack. Upon exit from foo all local variables are removed from the stack. In the above code, call foo causes the address of the ***instruction after call foo*** is to be saved in the stack.

```
# in file local.s
.global main
main: movl $3, %eax
        call foo                # save the address of the next instruction  on stack
        subl $4, %eax
        ret
foo:
        pushl %ebp              # save the address of base pointer
        movl %esp, %ebp         # move the base pointer to where stack pointer
        movl $10, %ebx          # push the local variable value to a register
        pushl %ebx              # push the value of register to stack
        movl %ebp, %esp         #  restore the esp upon return
        popl  %ebp              #  restore the base pointer
        ret                     # pop the stack to see where to return
```

Mixing C and Assembly

C programs and assembly programs can be mixed to form a complete program. For example, consider the following C program

```
// file in main.c
#include <stdio.h>
int main(){
   int i = foo(5);
   printf("The value is %d \n", i);
return 0; }
```

Now consider the function foo written in assembly

```
# file in foo.s
.global foo
foo:
        movl 4(%esp), %eax    # (esp+4) contains the value 5
        imull %eax, %eax      # multiply the register eax by itself
        ret                   # return values are given back thru eax
```

Both programs can be compiled, linked and run using

- **gcc main.c foo.s**
- **./a.out**

Global Variables

A global variable in a C program is a variable declared outside of any function (including main). For example, the following program defines a global variable x as follows.

```
// in file global.c
int x = 10;
int main( ) {
    int y  = x;
}
```

The assembly output produced by C compiler includes a global identification of the variable x.

```
.globl x
      .data
      .align 4
      .type x, @object
      .size x, 4
x:
      .long 10
      .text
.globl main
      .type main, @function
```

We have provided only a part of the output. The full assembly code can be generated by compiling the file global.c

> gcc –S global.c

Conclusion

This brief introductory lecture was intended to give you some very basic facts about assembly language, the stack, linking assembly and C programs. It is difficult to find good references for this subject. However, we recommend a few.

References:

[1] From C to Assembly – Linux Gazette, Issue 94, September 2003 - Hiran Ramakutty

[2] **Guide to Assembly Language Programming in Linux (Paperback)**
by Sivarama P. Dandamudi, Springer-Verlag, 2005

[3] *Computer Systems: A Programmer's Perspective* by Randal E. Bryant and David R. O'Hallaron
Prentice Hall, 2003. [15-213 textbook]