Intel assembly language using gcc

QOTD

"Assembly language programming is difficult. Make no mistake about that. It is not for wimps and weaklings."
- Tanenbaum's 6th, page 519

These notes are a supplement to our text. Some of this information is specific to our class. The syntax of our assembly code is different from that in the text.

The sections are:

1. Intro: using gcc, more help
2. Hello world
3. Some of the basics
4. The stack
5. Functions: name/declaration, entry and exit, parameters, return value, local variables, calling a function
6. Global variables
7. Names
8. Input/output

Read on!

1. Intro

The syntactic style supported by gcc is generally known as the "AT&T style." It was developed in conjunction with Unix. The text uses the MASM (Microsoft Assembly) assembly code style.

Using gcc

We'll use gcc as our assembler. It's pretty simple. For gcc, assembly language files use the suffix ".s". With your assembly file as a parameter, gcc assembles and links your code. It creates an executable file call `a.exe`.

```
> gcc test.s
> a
… your program runs…
```

You can also specify the executable file name using the -o option to gcc.

```
> gcc test.s -o test.exe
> test
… your program runs…
```

If you have multiple files to assemble, just add them to your gcc command line.

More help
Other related lecture notes are:

- Some Intel assembly instructions - my list of important assembly instructions
- Debugging assembly programs - my common bugs and debugging hints

On our k: drive, I also have a collection of tiny assembly code programs. These small programs demonstrate things like local variables, global variables, calling functions, scanf/printf, if statements, loops, etc. I have a file `template.s` there that you can copy and use as a starting point for your assembly language coding.

2. Hello world
Here's hello world in C:

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

In assembly code this is:

```
      .text               # begin text segment
FMT:  .ascii "Hello world\12\0"     # printf format string
      .align 4

      .globl _main        # declare _main
_main:
      pushl %ebp          # enter _main()
      movl %esp, %ebp

      pushl $FMT          # call printf( fmt)
      call _printf
      addl $4,%esp

      movl $0,%eax        # set return value, eax=0
      movl %ebp, %esp     # exit _main()
      popl %ebp
      ret
```

3. Some of the basics
As in C/C++, the first function called is _main. The underscore (_main) is a convention adopted to avoid name conflicts and is used on many "system" function names.

Comments start with a pound sign (#). C++ style comments (// and /* ... */) are also supported, but not after every statement type. So, I would just stick with #.

Assembly programs contain assembly instructions and assembler directives called pseudo-instructions. Pseudo-instructions generally start with a dot (.) and have many purposes. They guide the assembler in constructing correct machine code for your assembly code.

Instructions have the general format:

```
 label:          opcode          operand-list
```

Each of these fields is optional (blank lines are OK). Anything on a line after the pound sign (#) is a comment and is ignored by the assembler. Here are some examples:

```
FMT1:    .ascii "Enter num: %d\0"     # scanf string
         addl $4,%esp                 # restore stack
```

Programs have the following segments in memory with syntax in parens:

- Text - the main section containing your code and constant values (`.text`)
- Data - contains initialized global variables (`.data`)
- Bss - contains un-initialized global variables (`.bss`)

The registers most commonly used are: `%eax, %ebx, %ecx, %edx, %ebp, %esp`.

4. The stack

A section of program memory is reserved for the stack. The stack is used for function parameters and local variables.

Two registers are designated to hold pointers (addresses) into different locations in the stack. These must be properly maintained at all times:

- `%esp` - stack pointer, or the top of the stack
- `%ebp` - frame pointer for local variables and function parameters within a function

Stack notes:

- The stack starts at a high memory address. As items are added to the stack, it grows downward. Therefore, adding to the stack causes the address in `%esp` to decrease. A pop of the stack, reduces the number of items on the stack, and increments the address in `%esp`.
- The `pushl src` instruction places the value in `src` on the stack and then decrements the stack pointer `(%esp)` by 4 bytes.
- The `popl dest` instruction copies the top value on the stack to `dest` and then increments the stack pointer `(%esp)` by 4 bytes.
- The stack pointer `(%esp)` or frame pointer (`%ebp`) can be moved an arbitrary number of bytes using the `addl` instruction.

Usage of the stack is explained in greater detail in the following sections.

5. Functions

As in C, your assembly program is a collection of functions. So, it's important to be able to create and call functions. That's what this section describes.

To explain some things, I'll use this super simple function called `plus10()`:

```
      .globl plus10
plus10:
      pushl %ebp        # enter plus10()
      movl %esp, %ebp

      movl 8(%ebp),%edx # %edx = first parameter
      addl $10,%edx           # %edx = %edx + 10
      movl %edx,%eax    # %eax = %edx, set return value

      movl %ebp, %esp   # exit plus10()
      pop %ebp
      ret
```

♣ Function name/declaration

The declaration of a function requires two lines at the start: a `.globl` pseudo-instruction and a label for the function's name. In our example these are:

```
      .globl plus10
plus10:
```

♡ Function entry and exit

Upon entering a function, the previous function's frame pointer (`%ebp`) must be saved on the stack, and a new frame pointer for this function must be established. This is done with 2 lines of code:

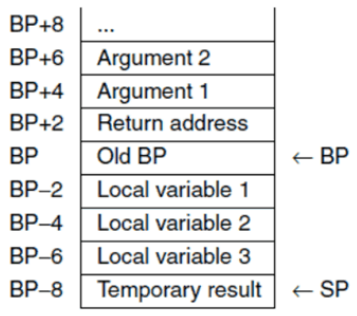
```
pushl %ebp        # enter plus10()
movl %esp, %ebp
```

Before leaving a function with the `ret` statement, you must restore the old frame pointer by:

```
movl %ebp, %esp   # exit plus10()
pop %ebp
ret
```

I place this entry and exit code in each function.

Figure C-6 from our text sheds some light on how this works.



We need to change one thing in this figure. Our addresses are 4 bytes, not 2. So, all the 2 byte quantities shown above should be 4.

Remember that the stack works down from high memory. SP, the stack pointer, (register `%esp`) holds the address of the top of the stack at all times. BP, the frame pointer, (register `%ebp`), is used to access the parameters and local variables of a function.

- **Function parameters** (or "arguments" in the figure) are found N bytes above the frame pointer. For 4 byte quantities, a function's parameters are located at `8(%ebp)`. It's second at `12%(ebp)`. Then `16%(ebp)` and so on… Notice how, by adding 8 bytes to the frame pointer, we jump over the old frame pointer and return address of the calling function.
- **Local variables** are found below the frame pointer. For 4 byte quantities, the first local variable is at -4(%ebp), then -8(%ebp) and so on...

For novices, it's a good idea to comment in your function this relationship between the frame pointer and parameters and local variables. For example:

```
# 8(%ebp) is first param N
# 12 (%ebp) is second paramer list_id
#
# -4(%ebp) is tmp
# -8(%ebp) is loop index i
```

Lastly, in figure C-6, look at "Return address" and "Old BP" on the stack. "Return address" is placed there by the instruction `call`. It is the address of the calling function. The instruction `ret` uses this to return control to the calling function. The "Old BP" was placed there by our function entry code.

♠ Function parameters

Function parameters must be pushed onto the stack before calling the function. Therefore, using our function entry and exit code, the first parameter always resides at `8(%ebp)`.

I used the parameter to `plus10()` in this statement:

```
movl $8(%ebp),%edx            # %edx = N, first parameter
```

♢ Function return value

By convention, the function return value is placed in the `%eax` register. Like this:

```
movl %edx,%eax    # %eax = %edx, set return value
```

♡ Local variables

Space for local variables is created on the stack by decrementing the stack pointer (`%esp`) using the `subl` instruction. Here's an example:

```
# create a local var on stack, set it to 10
subl $4,%esp      # sub 4 from esp to create local var
                  # -4(%ebp) is tally

movl $10,-4(%ebp) # tally = 10
```

In general, you subtract the number of bytes you need for the number of local variables in your function. For two local variables, we would subtract 8 bytes from the stack pointer: `subl $8,%esp`.

♣ Calling a function

Calling a function is a three-step process:

- Place parameters to the function on the stack using `pushl`
- Call the function using the `call` instruction
- Adjust the stack pointer `(%esp)` using `addl` to effectively remove the parameters from the stack

Here's a code snippet to call our sample function `plus10`:

```
pushl %edx        # push parameter
call plus10       # call plus10()
addl $4,%esp      # restore stack
```

The last step in the snippet above adds 4 to the stack pointer. This is because our parameter to `plus10()` was 4 bytes.

Here's a code fragment calling a function with two parameters: `foo( 7, 14)`. Notice that restoring the stack after `call` adds 8 bytes because it has two parameters.

```
pushl $14        # push arg 14
pushl $7         # push arg 7
call foo
addl $8, %esp    # remove args from stack
```

6. Global variables

Global variables are (usually) placed in the data section. The name of the global variable is specified using a label. The size of the variable is indicated by the pseudo-opcode specified:

- .long - an 4 byte integer
- .word - a 2 byte integer
- .byte - a 1 byte quantity, char or integer
- .space - allocates a number of bytes, such as storage for an array
- .ascii - a string, terminated by 0

Here's a code snippet:

```
.data
x:     .long 17    # long x = 17
y:     .word 3     # short y = 3
ch:    .byte 'W'   # char ch = 'W'
z:     .byte 1     # unsigned char z = 1
array: .space 20   # 20 bytes for array, for example 5 longs
FMT1: .ascii "This is a test.\12\0"       # printf format
```

Look at that string FMT above. The "\12" is an ASCII line feed. The "\0" must appear at the end of each string to signify the end of the string in memory.

These are global variables with an initial value. Un-initialized global variables should be defined in the `bss` section.

7. Names

The syntax of names in gcc assembly usually take some getting used to.

The $ indicates a literal value. The integer 16 is `$16`. The get the value of a global variables, it's: `$var_name`. There are two common cases for labels. If you are using label in a jump instruction, then you want the address and not the value, like this: `jmp LOOP_TOP`. If you use a label for a string constant, then you want the value, like this: `pushl $FMT1`.

Registers start with %. For example: `%esp`. A number and parens add an offset to the value in a register: `8(%ebp)`. This example reads: the address in `%ebp` plus 8 bytes. Negative numbers work too: `-16(%ebp)`.

8. Input/Output

Keeping it simple, we will input data using `scanf` and output data using `printf`. If this is your first exposure to `printf` and `scanf`, then huzzah! These are ubiquitous notions used across many languages.

Each function includes a format string with I/O data and directives in it. These directives usually start with %. Each function also takes a variable number of parameters, as specified in the format string.

Here's one tiny example. The output will be "Hello world and Bill".

```
printf( "Hello, world and %s", "Bill");
```

There are many sources of information on printf and scanf:

- The "Hello World" example in the beginning shows a call of `printf()`.
- My k: drive suite contains many examples of each.
- Here are C++ references: en.cppreference.com/w/cpp/io/c/fprintf, en.cppreference.com/w/cpp/io/c/fscanf
- Java's `String.format()` method uses a printf-style format string. The `PrintStream` class has a `printf()` method that may be used with System.out, like this:

```
System.out.printf( "Hello world, %s", name);
```

That's it. That's enough.
Thanks, Bill