X86 Assembly/GAS Syntax

General Information

Examples in this article are created using the AT&T assembly syntax used in GNU AS. The main advantage of using this syntax is its compatibility with the GCC inline assembly syntax. However, this is not the only syntax that is used to represent x86 operations. For example, NASM uses a different syntax to represent assembly mnemonics, operands and addressing modes, as do some High-Level Assemblers. The AT&T syntax is the standard on Unix-like systems but some assemblers use the Intel syntax, or can, like GAS itself, accept both.

GAS instructions generally have the form mnemonic source, destination. For instance, the following **mov** instruction:

```
movb $0x05, %al
```

will move the hexadecimal value 5 into the register al.

Operation Suffixes

GAS assembly instructions are generally suffixed with the letters "b", "s", "w", "l", "q" or "t" to determine what size operand is being manipulated.

- b = byte (8 bit)
- s = short (16 bit integer) or single (32-bit floating point)
- w = word (16 bit)
- l = long (32 bit integer or 64-bit floating point)
- q = quad (64 bit)
- t = ten bytes (80-bit floating point)

If the suffix is not specified, and there are no memory operands for the instruction, GAS infers the operand size from the size of the destination register operand (the final operand).

Prefixes

When referencing a register, the register needs to be prefixed with a "%". Constant numbers need to be prefixed with a "$".

Address operand syntax

There are up to 4 parameters of an address operand that are presented in the syntax `segment:displacement(base register, index register, scale factor)`. This is equivalent to `segment:[base register + displacement + index register * scale factor]` in Intel syntax.

The base, index and displacement components can be used in any combination, and every component can be omitted; omitted components are excluded from the calculation above[1][2].

```
movl    -8(%ebp, %edx, 4), %eax  # Full example: load *(ebp + (edx * 4) - 8) into eax
movl    -4(%ebp), %eax           # Typical example: load a stack variable into eax
```

```
movl    (%ecx), %edx            # No index: copy the target of a pointer into a register
leal    8(,%eax,4), %eax        # Arithmetic: multiply eax by 4 and add 8
leal    (%edx,%eax,2), %eax     # Arithmetic: multiply eax by 2 and add edx
```

Introduction

This section is written as a short introduction to GAS. GAS is part of the GNU Project (http://www.gnu.org/), which gives it the following nice properties:

- It is available on many operating systems.
- It interfaces nicely with the other GNU programming tools, including the GNU C compiler (gcc) and GNU linker (ld).

If you are using a computer with the Linux operating system, chances are you already have GAS installed on your system. If you are using a computer with the Windows operating system, you can install GAS and other useful programming utilities by installing Cygwin (http://www.cygwin.com/) or Mingw (http://www.mingw.org/). The remainder of this introduction assumes you have installed GAS and know how to open a command-line interface and edit files.

Generating assembly

Since assembly language corresponds directly to the operations a CPU performs, a carefully written assembly routine may be able to run much faster than the same routine written in a higher-level language, such as C. On the other hand, assembly routines typically take more effort to write than the equivalent routine in C. Thus, a typical method for quickly writing a program that performs well is to first write the program in a high-level language (which is easier to write and debug), then rewrite selected routines in assembly language (which performs better). A good first step to rewriting a C routine in assembly language is to use the C compiler to automatically generate the assembly language. Not only does this give you an assembly file that compiles correctly, but it also ensures that the assembly routine does exactly what you intended it to.[3]

We will now use the GNU C compiler to generate assembly code, for the purposes of examining the GAS assembly language syntax.

Here is the classic "Hello, world" program, written in C:

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Save that in a file called "hello.c", then type at the prompt:

```
gcc -o hello_c hello.c
```

This should compile the C file and create an executable file called "hello_c". If you get an error, make sure that the contents of "hello.c" are correct.

Now you should be able to type at the prompt:

```
./hello_c
```

and the program should print "Hello, world!" to the console.

Now that we know that "hello.c" is typed in correctly and does what we want, let's generate the equivalent 32-bit x86 assembly language. Type the following at the prompt:

```
gcc -S -m32 hello.c
```

This should create a file called "hello.s" (".s" is the file extension that the GNU system gives to assembly files). On more recent 64-bit systems, the 32-bit source tree may not be included, which will cause a "bits/predefs.h fatal error"; you may replace the "-m32" gcc directive with an "-m64" directive to generate 64-bit assembly instead. To compile the assembly file into an executable, type:

```
gcc -o hello_asm -m32 hello.s
```

(Note that gcc calls the assembler (as) and the linker (ld) for us.) Now, if you type the following at the prompt:

```
./hello_asm
```

this program should also print "Hello, world!" to the console. Not surprisingly, it does the same thing as the compiled C file.

Let's take a look at what is inside "hello.s":

```
        .file   "hello.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
LC0:
        .ascii "Hello, world!\12\0"
.globl _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
        .def    _printf;        .scl    2;      .type   32;     .endef
```

The contents of "hello.s" may vary depending on the version of the GNU tools that are installed; this version was generated with Cygwin, using gcc version 3.3.1.

The lines beginning with periods, like ".file", ".def", or ".ascii" are assembler directives -- commands that tell the assembler how to assemble the file. The lines beginning with some text followed by a colon, like "_main:", are labels, or named locations in the code. The other lines are assembly instructions.

The ".file" and ".def" directives are for debugging. We can leave them out:

```
        .text
LC0:
        .ascii "Hello, world!\12\0"
```

```
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
```

"hello.s" line-by-line

```
        .text
```

This line declares the start of a section of code. You can name sections using this directive, which gives you fine-grained control over where in the executable the resulting machine code goes, which is useful in some cases, like for programming embedded systems. Using ".text" by itself tells the assembler that the following code goes in the default section, which is sufficient for most purposes.

```
LC0:
        .ascii "Hello, world!\12\0"
```

This code declares a label, then places some raw ASCII text into the program, starting at the label's location. The "\12" specifies a line-feed character, while the "\0" specifies a null character at the end of the string; C routines mark the end of strings with null characters, and since we are going to call a C string routine, we need this character here. (NOTE! String in C is an array of datatype Char (Char[]) and does not exist in any other form, but because one would understand strings as a single entity from the majority of programming languages, it is clearer to express it this way).

```
.globl _main
```

This line tells the assembler that the label "_main" is a global label, which allows other parts of the program to see it. In this case, the linker needs to be able to see the "_main" label, since the startup code with which the program is linked calls "_main" as a subroutine.

```
_main:
```

This line declares the "_main" label, marking the place that is called from the startup code.

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
```

These lines save the value of EBP on the stack, then move the value of ESP into EBP, then subtract 8 from ESP. Note that pushl automatically decremented ESP by the appropriate length. The "l" on the end of each opcode indicates that we want to use the version of the opcode that works with "long" (32-bit) operands; usually the assembler is able to work out the correct opcode version from the operands, but just to be safe, it's a good idea to include the "l", "w", "b", or other suffix.

The percent signs designate register names, and the dollar sign designates a literal value. This sequence of instructions is typical at the start of a subroutine to save space on the stack for local variables; EBP is used as the base register to reference the local variables, and a value is subtracted from ESP to reserve space on the stack (since the Intel stack grows from higher memory locations to lower ones). In this case, eight bytes have been reserved on the stack. We shall see why this space is needed later.

```
	andl	$-16, %esp
```

This code "and"s ESP with 0xFFFFFFF0, aligning the stack with the next lowest 16-byte boundary. An examination of Mingw's source code reveals that this may be for SIMD instructions appearing in the "_main" routine, which operate only on aligned addresses. Since our routine doesn't contain SIMD instructions, this line is unnecessary.

```
	movl	$0, %eax
	movl	%eax, -4(%ebp)
	movl	-4(%ebp), %eax
```

This code moves zero into EAX, then moves EAX into the memory location EBP-4, which is in the temporary space we reserved on the stack at the beginning of the procedure. Then it moves the memory location EBP-4 back into EAX; clearly, this is not optimized code. Note that the parentheses indicate a memory location, while the number in front of the parentheses indicates an offset from that memory location.

```
	call	__alloca
	call	___main
```

These functions are part of the C library setup. Since we are calling functions in the C library, we probably need these. The exact operations they perform vary depending on the platform and the version of the GNU tools that are installed.

```
	movl	$LC0, (%esp)
	call	_printf
```

This code (finally!) prints our message. First, it moves the location of the ASCII string to the top of the stack. It seems that the C compiler has optimized a sequence of "popl %eax; pushl $LC0" into a single move to the top of the stack. Then, it calls the _printf subroutine in the C library to print the message to the console.

```
	movl	$0, %eax
```

This line stores zero, our return value, in EAX. The C calling convention is to store return values in EAX when exiting a routine.

```
	leave
```

This line, typically found at the end of subroutines, frees the space saved on the stack by copying EBP into ESP, then popping the saved value of EBP back to EBP.

```
	ret
```

This line returns control to the calling procedure by popping the saved instruction pointer from the stack.

Communicating directly with the operating system

Note that we only have to call the C library setup routines if we need to call functions in the C library, like "printf". We could avoid calling these routines if we instead communicate directly with the operating system. The disadvantage of communicating directly with the operating system is that we lose portability; our code will be locked to a specific operating system. For instructional purposes, though, let's look at how one might do this under Windows. Here is the C source code, compilable under Mingw or Cygwin:

```
#include <windows.h>

int main(void) {
    LPSTR text = "Hello, world!\n";
    DWORD charsWritten;
    HANDLE hStdout;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    WriteFile(hStdout, text, 14, &charsWritten, NULL);
    return 0;
}
```

Ideally, you'd want check the return codes of "GetStdHandle" and "WriteFile" to make sure they are working correctly, but this is sufficient for our purposes. Here is what the generated assembly looks like:

```
        .file   "hello2.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
LC0:
        .ascii "Hello, world!\12\0"
.globl _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -16(%ebp)
        movl    -16(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, -4(%ebp)
        movl    $-11, (%esp)
        call    _GetStdHandle@4
        subl    $4, %esp
        movl    %eax, -12(%ebp)
        movl    $0, 16(%esp)
        leal    -8(%ebp), %eax
        movl    %eax, 12(%esp)
        movl    $14, 8(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -12(%ebp), %eax
        movl    %eax, (%esp)
        call    _WriteFile@20
        subl    $20, %esp
        movl    $0, %eax
        leave
        ret
```

Even though we never use the C standard library, the generated code initializes it for us. Also, there is a lot of unnecessary stack manipulation. We can simplify:

```
        .text
LC0:
        .ascii "Hello, world!\12\0"
```

```
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        pushl   $-11
        call    _GetStdHandle@4
        pushl   $0
        leal    -4(%ebp), %ebx
        pushl   %ebx
        pushl   $14
        pushl   $LC0
        pushl   %eax
        call    _WriteFile@20
        movl    $0, %eax
        leave
        ret
```

Analyzing line-by-line:

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
```

We save the old EBP and reserve four bytes on the stack, since the call to WriteFile needs somewhere to store the number of characters written, which is a 4-byte value.

```
        pushl   $-11
        call    _GetStdHandle@4
```

We push the constant value STD_OUTPUT_HANDLE (-11) to the stack and call GetStdHandle. The returned handle value is in EAX.

```
        pushl   $0
        leal    -4(%ebp), %ebx
        pushl   %ebx
        pushl   $14
        pushl   $LC0
        pushl   %eax
        call    _WriteFile@20
```

We push the parameters to WriteFile and call it. Note that the Windows calling convention is to push the parameters from right-to-left. The load-effective-address ("lea") instruction adds -4 to the value of EBP, giving the location we saved on the stack for the number of characters printed, which we store in EBX and then push onto the stack. Also note that EAX still holds the return value from the GetStdHandle call, so we just push it directly.

```
        movl    $0, %eax
        leave
```

Here we set our program's return value and restore the values of EBP and ESP using the "leave" instruction.

Caveats

From The GAS manual's AT&T Syntax Bugs section (http://sourceware.org/binutils/docs/as/i386_002dBugs.html#i386_002dBugs):

The UnixWare assembler, and probably other AT&T derived ix86 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, gcc and possibly many other programs use this reversed syntax, so we're stuck with it.

For example

```
       fsub %st,%st(3)
```

results in `%st(3)` being updated to `%st - %st(3)` rather than the expected `%st(3) - %st`. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is `%st` and the destination register is `%st(i)`.

Note that even objdump -d -M intel still uses reversed opcodes, so use a different disassembler to check this. See http://bugs.debian.org/372528 for more info.

Additional GAS reading

You can read more about GAS at the GNU GAS documentation page:

https://sourceware.org/binutils/docs/as/

- X86 Disassembly/Calling Conventions

Quick reference

Instruction	Meaning
`movq %rax, %rbx`	`rbx = rax`
`movq $123, %rax`	`rax = 123`
`movq %rsi, -16(%rbp)`	`mem[rbp-16] = rsi`
`subq $10, %rbp`	`rbp = rbp -10`
`cmpl %eax %ebx`	`compare then set flags. If eax == ebx, zero flag is set.`
`jmp <location>`	`unconditional jump`
`je <location>`	`jump to <location> if equal flag is set`
`jg,jge,jl,jle,jne,...`	`>,>=,<,<=,!=,...`

Notes

1. If `segment` is not specified, as almost always, it is assumed to be `ds`, unless `base register` is `esp` or `ebp`; in this case, the address is assumed to be relative to `ss`
2. If `index register` is missing, the pointless `scale factor` must be omitted as well.
3. This assumes that the compiler has no bugs and, more importantly, *that the code you wrote correctly implements your intent*. Note also that compilers can sometimes rearrange the sequence of low-level operations in order to optimize the code; this preserves the overall semantics of your code but means the assembly instruction flow may not match up exactly with your algorithm steps.