Linux assemblers: A comparison of GAS and NASM

A side-by-side look at GNU Assembler (GAS) and Netwide Assembler (NASM)

Ram Narayan

October 17, 2007

This article explains some of the more important syntactic and semantic differences between two of the most popular assemblers for Linux®, GNU Assembler (GAS) and Netwide Assembler (NASM), including differences in basic syntax, variables and memory access, macro handling, functions and external routines, stack handling, and techniques for easily repeating blocks of code.

Introduction

Unlike other languages, **assembly programming** involves understanding the processor architecture of the machine that is being programmed. Assembly programs are not at all portable and are often cumbersome to maintain and understand, and can often contain a large number of lines of code. But with these limitations comes the advantage of speed and size of the runtime binary that executes on that machine.

Though much information is already available on assembly level programming on Linux, this article aims to more specifically show the differences between syntaxes in a way that will help you more easily convert from one flavor of assembly to the another. The article evolved from my own quest to improve at this conversion.

This article uses a series of program examples. Each program illustrates some feature and is followed by a discussion and comparison of the syntaxes. Although it's not possible to cover every difference that exists between NASM and GAS, I do try to cover the main points and provide a foundation for further investigation. And for those already familiar with both NASM and GAS, you might still find something useful here, such as macros.

This article assumes you have at least a basic understanding of assembly terminology and have programmed with an assembler using Intel® syntax, perhaps using NASM on Linux or Windows. This article does not teach how to type code into an editor or how to assemble and link (but see the sidebar for a quick refresher). You should be familiar with the Linux operating system (any

Linux distribution will do; I used Red Hat and Slackware) and basic GNU tools such as gcc and ld, and you should be programming on an x86 machine.

Now I'll describe what this article does and does not cover.

Building the examples

Assembling:
GAS:
```
as –o program.o program.s
```

NASM:
```
nasm –f elf –o program.o program.asm
```

Linking (common to both kinds of assembler):
```
ld –o program program.o
```

Linking when an external C library is to be used:
```
ld ––dynamic-linker /lib/ld-linux.so.2 –lc –o program program.o
```

This article covers:

- Basic syntactical differences between NASM and GAS
- Common assembly level constructs such as variables, loops, labels, and macros
- A bit about calling external C routines and using functions
- Assembly mnemonic differences and usage
- Memory addressing methods

This article does not cover:

- The processor instruction set
- Various forms of macros and other constructs particular to an assembler
- Assembler directives peculiar to either NASM or GAS
- Features that are not commonly used or are found only in one assembler but not in the other

For more information, refer to the official assembler manuals (see Related topics for links), as those are the most complete sources of information.

Basic structure

Listing 1 shows a very simple program that simply exits with an exit code of 2. This little program describes the basic structure of an assembly program for both GAS and NASM.

Listing 1. A program that exits with an exit code of 2

Line	NASM	GAS

```
; Text segment begins
section .text

   global _start

; Program entry point
   _start:

; Put the code number for system call
      mov   eax, 1

; Return value
      mov   ebx, 2

; Call the OS
      int   80h
```

```
# Text segment begins
.section .text

   .globl _start

# Program entry point
   _start:

# Put the code number for system call
      movl  $1, %eax

/* Return value */
      movl  $2, %ebx

# Call the OS
      int   $0x80
```

Now for a bit of explanation.

One of the biggest differences between NASM and GAS is the syntax. GAS uses the AT&T syntax, a relatively archaic syntax that is specific to GAS and some older assemblers, whereas NASM uses the Intel syntax, supported by a majority of assemblers such as TASM and MASM. (Modern versions of GAS do support a directive called `.intel_syntax`, which allows the use of Intel syntax with GAS.)

The following are some of the major differences summarized from the GAS manual:

- AT&T and Intel syntax use the opposite order for source and destination operands. For example:
  - Intel: `mov eax, 4`
  - AT&T: `movl $4, %eax`
- In AT&T syntax, immediate operands are preceded by `$`; in Intel syntax, immediate operands are not. For example:
  - Intel: `push 4`
  - AT&T: `pushl $4`
- In AT&T syntax, register operands are preceded by `%`; in Intel syntax, they are not.
- In AT&T syntax, the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of `b`, `w`, and `l` specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (not the opcodes themselves) with `byte ptr`, `word ptr`, and `dword ptr`. Thus:
  - Intel: `mov al, byte ptr foo`
  - AT&T: `movb foo, %al`
- Immediate form long jumps and calls are `lcall/ljmp $section, $offset` in AT&T syntax; the Intel syntax is `call/jmp far section:offset`. The far return instruction is `lret $stack-adjust` in AT&T syntax, whereas Intel uses `ret far stack-adjust`.

In both the assemblers, the names of registers remain the same, but the syntax for using them is different as is the syntax for addressing modes. In addition, assembler directives in GAS begin with a ".", but not in NASM.

The `.text` section is where the processor begins code execution. The `global` (also `.globl` or `.global` in GAS) keyword is used to make a symbol visible to the linker and available to other linking object modules. On the NASM side of Listing 1, `global _start` marks the symbol `_start` as a visible identifier so the linker knows where to jump into the program and begin execution. As with NASM, GAS looks for this `_start` label as the default entry point of a program. A label always ends with a colon in both GAS and NASM.

Interrupts are a way to inform the OS that its services are required. The `int` instruction in line 16 does this job in our program. Both GAS and NASM use the same mnemonic for interrupts. GAS uses the `0x` prefix to specify a hex number, whereas NASM uses the `h` suffix. Because immediate operands are prefixed with `$` in GAS, 80 hex is `$0x80`.

`int $0x80` (or `80h` in NASM) is used to invoke Linux and request a service. The service code is present in the EAX register. A value of 1 (for the Linux exit system call) is stored in EAX to request that the program exit. Register EBX contains the exit code (2, in our case), a number that is returned to the OS. (You can track this number by typing `echo $?` at the command prompt.)

Finally, a word about comments. GAS supports both C style (`/* */`), C++ style (`//`), and shell style (`#`) comments. NASM supports single-line comments that begin with the ";" character.

Variables and accessing memory

This section begins with an example program that finds the largest of three numbers.

Listing 2. A program that finds the maximum of three numbers

NASM

```
; Data section begins
section .data

   var1 dd 40

   var2 dd 20

   var3 dd 30


section .text

   global _start

   _start:

; Move the contents of variables
      mov   ecx, [var1]
      cmp   ecx, [var2]
      jg    check_third_var
      mov   ecx, [var2]

   check_third_var:
      cmp   ecx, [var3]
      jg    _exit
      mov   ecx, [var3]

   _exit:
      mov   eax, 1
      mov   ebx, ecx
      int   80h
```

GAS

```
// Data section begins
.section .data

   var1:
      .int 40
   var2:
      .int 20
   var3:
      .int 30

.section .text

   .globl _start

   _start:

# move the contents of variables
      movl  (var1), %ecx
      cmpl  (var2), %ecx
      jg    check_third_var
      movl  (var2), %ecx

   check_third_var:
      cmpl  (var3), %ecx
      jg    _exit
      movl  (var3), %ecx

   _exit:
      movl  $1, %eax
      movl  %ecx, %ebx
      int   $0x80
```

You can see several differences above in the declaration of memory variables. NASM uses the `dd`, `dw`, and `db` directives to declare 32-, 16-, and 8-bit numbers, respectively, whereas GAS uses the `.long`, `.int`, and `.byte` for the same purpose. GAS has other directives too, such as `.ascii`, `.asciz`, and `.string`. In GAS, you declare variables just like other labels (using a colon), but in NASM you simply type a variable name (without the colon) before the memory allocation directive (`dd`, `dw`, etc.), followed by the value of the variable.

Line 18 in Listing 2 illustrates the memory indirect addressing mode. NASM uses square brackets to dereference the value at the address pointed to by a memory location: `[var1]`. GAS uses a circular brace to dereference the same value: `(var1)`. The use of other addressing modes is covered later in this article.

Using macros

Listing 3 illustrates the concepts of this section; it accepts the user's name as input and returns a greeting.

Listing 3. A program to read a string and display a greeting to the user

NASM

```
section .data

   prompt_str  db   'Enter your name: '

; $ is the location counter
   STR_SIZE  equ  $ - prompt_str

   greet_str  db  'Hello '


   GSTR_SIZE  equ  $ - greet_str


section .bss

; Reserve 32 bytes of memory
   buff  resb  32

; A macro with two parameters
; Implements the write system call
   %macro write 2
      mov   eax, 4
      mov   ebx, 1
      mov   ecx, %1
      mov   edx, %2
      int   80h
   %endmacro


; Implements the read system call
   %macro read 2
      mov   eax, 3
      mov   ebx, 0
      mov   ecx, %1
      mov   edx, %2
      int   80h
   %endmacro


section .text

   global _start

   _start:
      write prompt_str, STR_SIZE
      read  buff, 32

```

GAS

```
.section .data

   prompt_str:
      .ascii "Enter Your Name: "
   pstr_end:
      .set STR_SIZE, pstr_end - prompt_str

   greet_str:
      .ascii "Hello "

   gstr_end:
      .set GSTR_SIZE, gstr_end - greet_str

.section .bss

// Reserve 32 bytes of memory
   .lcomm  buff, 32

// A macro with two parameters
//  implements the write system call
   .macro write str, str_size
      movl  $4, %eax
      movl  $1, %ebx
      movl  \str, %ecx
      movl  \str_size, %edx
      int   $0x80
   .endm


// Implements the read system call
   .macro read buff, buff_size
      movl  $3, %eax
      movl  $0, %ebx
      movl  \buff, %ecx
      movl  \buff_size, %edx
      int   $0x80
   .endm


.section .text

   .globl _start

   _start:
      write $prompt_str, $STR_SIZE
      read  $buff, $32
```

049	`; Read returns the length in eax`	
050	`    push  eax`	`// Read returns the length in eax`
051		`    pushl %eax`
052	`; Print the hello text`	
053	`    write greet_str, GSTR_SIZE`	`// Print the hello text`
054		`    write $greet_str, $GSTR_SIZE`
055	`    pop   edx`	
056		`    popl  %edx`
057	`; edx  = length returned by read`	
058	`    write buff, edx`	`// edx = length returned by read`
059		`  write $buff, %edx`
060	`  _exit:`	
061	`    mov   eax, 1`	`  _exit:`
062	`    mov   ebx, 0`	`    movl  $1, %eax`
	`    int   80h`	`    movl  $0, %ebx`
		`    int   $0x80`

The heading for this section promises a discussion of macros, and both NASM and GAS certainly support them. But before we get into macros, a few other features are worth comparing.

Listing 3 illustrates the concept of uninitialized memory, defined using the `.bss` section directive (line 14). BSS stands for "block storage segment" (originally, "block started by symbol"), and the memory reserved in the BSS section is initialized to zero during the start of the program. Objects in the BSS section have only a name and a size, and no value. Variables declared in the BSS section don't actually take space, unlike in the data segment.

NASM uses the `resb`, `resw`, and `resd` keywords to allocated byte, word, and dword space in the BSS section. GAS, on the other hand, uses the `.lcomm` keyword to allocate byte-level space. Notice the way the variable name is declared in both versions of the program. In NASM the variable name precedes the `resb` (or `resw` or `resd`) keyword, followed by the amount of space to be reserved, whereas in GAS the variable name follows the `.lcomm` keyword, which is then followed by a comma and then the amount of space to be reserved. This shows the difference:

NASM: `varname resb size`

GAS: `.lcomm varname, size`

Listing 2 also introduces the concept of a location counter (line 6). NASM provides a special variable (the `$` and `$$` variables) to manipulate the location counter. In GAS, there is no method to manipulate the location counter and you have to use labels to calculate the next storage location (data, instruction, etc.).

For example, to calculate the length of a string, you would use the following idiom in NASM:

```
prompt_str db 'Enter your name: '
STR_SIZE equ $ - prompt_str    ; $ is the location counter
```

The `$` gives the current value of the location counter, and subtracting the value of the label (all variable names are labels) from this location counter gives the number of bytes present between the declaration of the label and the current location. The `equ` directive is used to set the value of the variable STR_SIZE to the expression following it. A similar idiom in GAS looks like this:

```
prompt_str:
```

```
    .ascii "Enter Your Name: "
pstr_end:
    .set STR_SIZE, pstr_end - prompt_str
```

The end label (`pstr_end`) gives the next location address, and subtracting the starting label address gives the size. Also note the use of `.set` to initialize the value of the variable STR_SIZE to the expression following the comma. A corresponding `.equ` can also be used. There is no alternative to GAS's `set` directive in NASM.

As I mentioned, Listing 3 uses macros (line 21). Different macro techniques exist in NASM and GAS, including single-line macros and macro overloading, but I only deal with the basic type here. A common use of macros in assembly is clarity. Instead of typing the same piece of code again and again, you can create reusable macros that both avoid this repetition and enhance the look and readability of the code by reducing clutter.

NASM users might be familiar with declaring macros using the `%beginmacro` directive and ending them with an `%endmacro` directive. A `%beginmacro` directive is followed by the macro name. After the macro name comes a count, the number of macro arguments the macro is supposed to have. In NASM, macro arguments are numbered sequentially starting with 1. That is, the first argument to a macro is %1, the second is %2, the third is %3, and so on. For example:

```
%beginmacro macroname 2
    mov eax, %1
    mov ebx, %2
%endmacro
```

This creates a macro with two arguments, the first being `%1` and the second being `%2`. Thus, a call to the above macro would look something like this:

```
macroname 5, 6
```

Macros can also be created without arguments, in which case they don't specify any number.

Now let's take a look at how GAS uses macros. GAS provides the `.macro` and `.endm` directives to create macros. A `.macro` directive is followed by a macro name, which may or may not have arguments. In GAS, macro arguments are given by name. For example:

```
.macro macroname arg1, arg2
    movl \arg1, %eax
    movl \arg2, %ebx
.endm
```

A backslash precedes the name of each argument of the macro when the name is actually used inside a macro. If this is not done, the linker would treat the names as labels rather then as arguments and will report an error.

Functions, external routines, and the stack

The example program for this section implements a selection sort on an array of integers.

Listing 4. Implementation of selection sort on an integer array

NASM

```
section .data

   array db
      89, 10, 67, 1, 4, 27, 12, 34,
         86, 3

   ARRAY_SIZE equ $ - array


   array_fmt db "  %d", 0


   usort_str db "unsorted array:", 0


   sort_str db "sorted array:", 0


   newline db 10, 0



section .text
   extern puts

   global _start

   _start:

      push  usort_str
      call  puts
      add   esp, 4

      push  ARRAY_SIZE
      push  array
      push  array_fmt
      call  print_array10
      add   esp, 12

      push  ARRAY_SIZE
      push  array
      call  sort_routine20

; Adjust the stack pointer
      add   esp, 8

      push  sort_str
      call  puts
      add   esp, 4

      push  ARRAY_SIZE
      push  array
      push  array_fmt
      call  print_array10
      add   esp, 12
      jmp   _exit

      extern printf

   print_array10:
      push  ebp
      mov   ebp, esp
      sub   esp, 4
      mov   edx, [ebp + 8]
      mov   ebx, [ebp + 12]
      mov   ecx, [ebp + 16]
```

GAS

```
.section .data

   array:
      .byte  89, 10, 67, 1, 4, 27,
 12,
             34, 86, 3

   array_end:
      .equ ARRAY_SIZE, array_end -
 array

   array_fmt:
      .asciz "  %d"

   usort_str:
      .asciz "unsorted array:"

   sort_str:
      .asciz "sorted array:"

   newline:
      .asciz "\n"


.section .text


   .globl _start

   _start:

      pushl $usort_str
      call  puts
      addl  $4, %esp

      pushl $ARRAY_SIZE
      pushl $array
      pushl $array_fmt
      call  print_array10
      addl  $12, %esp

      pushl $ARRAY_SIZE
      pushl $array
      call  sort_routine20

# Adjust the stack pointer
      addl  $8, %esp

      pushl $sort_str
      call  puts
      addl  $4, %esp

      pushl $ARRAY_SIZE
      pushl $array
      pushl $array_fmt
      call  print_array10
      addl  $12, %esp
      jmp   _exit



   print_array10:
      pushl %ebp
      movl  %esp, %ebp
      subl  $4, %esp
      movl  8(%ebp), %edx
      movl  12(%ebp), %ebx
```

```
    mov   esi, 0

  push_loop:
    mov   [ebp - 4], ecx
    mov   edx, [ebp + 8]
    xor   eax, eax
    mov   al, byte [ebx + esi]
    push  eax
    push  edx

    call  printf
    add   esp, 8
    mov   ecx, [ebp - 4]
    inc   esi
    loop  push_loop

    push  newline
    call  printf
    add   esp, 4
    mov   esp, ebp
    pop   ebp
    ret

  sort_routine20:
    push  ebp
    mov   ebp, esp

; Allocate a word of space in stack
    sub   esp, 4

; Get the address of the array
    mov   ebx, [ebp + 8]

; Store array size
    mov   ecx, [ebp + 12]
    dec   ecx

; Prepare for outer loop here
    xor   esi, esi

  outer_loop:
; This stores the min index
    mov   [ebp - 4], esi
    mov   edi, esi
    inc   edi

  inner_loop:
    cmp   edi, ARRAY_SIZE
    jge   swap_vars
    xor   al, al
    mov   edx, [ebp - 4]
    mov   al, byte [ebx + edx]
    cmp   byte [ebx + edi], al
    jge   check_next
    mov   [ebp - 4], edi

  check_next:
    inc   edi
    jmp   inner_loop

  swap_vars:
    mov   edi, [ebp - 4]
    mov   dl, byte [ebx + edi]
    mov   al, byte [ebx + esi]
    mov   byte [ebx + esi], dl
    mov   byte [ebx + edi], al

    inc   esi
    loop  outer_loop

    mov   esp, ebp
    pop   ebp
    ret

  _exit:
    mov   eax, 1
    mov   ebx, 0
```

```
    movl  16(%ebp), %ecx

    movl  $0, %esi

  push_loop:
    movl  %ecx, -4(%ebp)
    movl  8(%ebp), %edx
    xorl  %eax, %eax
    movb  (%ebx, %esi, 1), %al
    pushl %eax
    pushl %edx

    call  printf
    addl  $8, %esp
    movl  -4(%ebp), %ecx
    incl  %esi
    loop  push_loop

    pushl $newline
    call  printf
    addl  $4, %esp
    movl  %ebp, %esp
    popl  %ebp
    ret

  sort_routine20:
    pushl %ebp
    movl  %esp, %ebp

# Allocate a word of space in stack
    subl  $4, %esp

# Get the address of the array
    movl  8(%ebp), %ebx

# Store array size
    movl  12(%ebp), %ecx
    decl  %ecx

# Prepare for outer loop here
    xorl  %esi, %esi

  outer_loop:
# This stores the min index
    movl  %esi, -4(%ebp)
    movl  %esi, %edi
    incl  %edi

  inner_loop:
    cmpl  $ARRAY_SIZE, %edi
    jge   swap_vars
    xorb  %al, %al
    movl  -4(%ebp), %edx
    movb  (%ebx, %edx, 1), %al
    cmpb  %al, (%ebx, %edi, 1)
    jge   check_next
    movl  %edi, -4(%ebp)

  check_next:
    incl  %edi
    jmp   inner_loop

  swap_vars:
    movl  -4(%ebp), %edi
    movb  (%ebx, %edi, 1), %dl
    movb  (%ebx, %esi, 1), %al
    movb  %dl, (%ebx, %esi, 1)
    movb  %al, (%ebx,  %edi, 1)

    incl  %esi
    loop  outer_loop

    movl  %ebp, %esp
    popl  %ebp
    ret

  _exit:
```

145	`int   80h`	`movl  $1, %eax`
		`movl  $0, %ebx`
		`int   $0x80`

Listing 4 might look overwhelming at first, but in fact it's very simple. The listing introduces the concept of functions, various memory addressing schemes, the stack and the use of a library function. The program sorts an array of 10 numbers and uses the external C library functions `puts` and `printf` to print out the entire contents of the unsorted and sorted array. For modularity and to introduce the concept of functions, the sort routine itself is implemented as a separate procedure along with the array print routine. Let's deal with them one by one.

After the data declarations, the program execution begins with a call to `puts` (line 31). The `puts` function displays a string on the console. Its only argument is the address of the string to be displayed, which is passed on to it by pushing the address of the string in the stack (line 30).

In NASM, any label that is not part of our program and needs to be resolved during link time must be predefined, which is the function of the `extern` keyword (line 24). GAS doesn't have such requirements. After this, the address of the string `usort_str` is pushed onto the stack (line 30). In NASM, a memory variable such as `usort_str` represents the address of the memory location itself, and thus a call such as `push usort_str` actually pushes the address on top of the stack. In GAS, on the other hand, the variable `usort_str` must be prefixed with `$`, so that it is treated as an immediate address. If it's not prefixed with `$`, the actual bytes represented by the memory variable are pushed onto the stack instead of the address.

Since pushing a variable essentially moves the stack pointer by a dword, the stack pointer is adjusted by adding 4 (the size of a dword) to it (line 32).

Three arguments are now pushed onto the stack, and the `print_array10` function is called (line 37). Functions are declared the same way in both NASM and GAS. They are nothing but labels, which are invoked using the `call` instruction.

After a function call, ESP represents the top of the stack. A value of `esp + 4` represents the return address, and a value of `esp + 8` represents the first argument to the function. All subsequent arguments are accessed by adding the size of a dword variable to the stack pointer (that is, `esp + 12`, `esp + 16`, and so on).

Once inside a function, a local stack frame is created by copying `esp` to `ebp` (line 62). You can also allocate space for local variables as is done in the program (line 63). You do this by subtracting the number of bytes required from `esp`. A value of `esp – 4` represents a space of 4 bytes allocated for a local variable, and this can continue as long as there is enough space in the stack to accommodate your local variables.

Listing 4 illustrates the base indirect addressing mode (line 64), so called because you start with a base address and add an offset to it to arrive at a final address. On the NASM side of the listing, `[ebp + 8]` is one such example, as is `[ebp – 4]` (line 71). In GAS, the addressing is a bit more terse: `4(%ebp)` and `-4(%ebp)`, respectively.

In the `print_array10` routine, you can see another kind of addressing mode being used after the `push_loop` label (line 74). The line is represented in NASM and GAS, respectively, like so:

NASM: `mov al, byte [ebx + esi]`

GAS: `movb (%ebx, %esi, 1), %al`

This addressing mode is the base indexed addressing mode. Here, there are three entities: one is the base address, the second is the index register, and the third is the multiplier. Because it's not possible to determine the number of bytes to be accessed from a memory location, a method is needed to find out the amount of memory addressed. NASM uses the byte operator to tell the assembler that a byte of data is to be moved. In GAS the same problem is solved by using a multiplier as well as using the `b`, `w`, or `l` suffix in the mnemonic (for example, `movb`). The syntax of GAS can seem somewhat complex when first encountered.

The general form of base indexed addressing in GAS is as follows:

```
%segment:ADDRESS (, index, multiplier)
```

or

```
%segment:(offset, index, multiplier)
```

or

```
%segment:ADDRESS(base, index, multiplier)
```

The final address is calculated using this formula:

```
ADDRESS or offset + base + index * multiplier.
```

Thus, to access a byte, a multiplier of 1 is used, for a word, 2, and for a dword, 4. Of course, NASM uses a simpler syntax. Thus, the above in NASM would be represented like so:

```
Segment:[ADDRESS or offset + index * multiplier]
```

A prefix of `byte`, `word`, or `dword` is used before this memory address to access 1, 2, or 4 bytes of memory, respectively.

Leftovers

Listing 5. A program that reads command line arguments, stores them in memory, and prints them

NASM

```
section .data

; Command table to store at most
;  10 command line arguments
   cmd_tbl:
      %rep 10
         dd 0
```

GAS

```
.section .data

// Command table to store at most
//  10 command line arguments
   cmd_tbl:
      .rept 10
         .long 0
```

008	%endrep	.endr
009		
010	section .text	.section .text
011		
012	global _start	.globl _start
013		
014	_start:	_start:
015	; Set up the stack frame	// Set up the stack frame
016	mov ebp, esp	movl %esp, %ebp
017	; Top of stack contains the	// Top of stack contains the
018	; number of command line arguments.	// number of command line arguments.
019	; The default value is 1	// The default value is 1
020	mov ecx, [ebp]	movl (%ebp), %ecx
021		
022	; Exit if arguments are more than 10	// Exit if arguments are more than 10
023	cmp ecx, 10	cmpl $10, %ecx
024	jg _exit	jg _exit
025		
026	mov esi, 1	movl $1, %esi
027	mov edi, 0	movl $0, %edi
028		
029	; Store the command line arguments	// Store the command line arguments
030	; in the command table	// in the command table
031	store_loop:	store_loop:
032	mov eax, [ebp + esi * 4]	movl (%ebp, %esi, 4), %eax
033	mov [cmd_tbl + edi * 4], eax	movl %eax, cmd_tbl(, %edi, 4)
034	inc esi	incl %esi
035	inc edi	incl %edi
036	loop store_loop	loop store_loop
037		
038	mov ecx, edi	movl %edi, %ecx
039	mov esi, 0	movl $0, %esi
040		
041	extern puts	
042		
043	print_loop:	print_loop:
044	; Make some local space	// Make some local space
045	sub esp, 4	subl $4, %esp
046	; puts function corrupts ecx	// puts functions corrupts ecx
047	mov [ebp - 4], ecx	movl %ecx, -4(%ebp)
048	mov eax, [cmd_tbl + esi * 4]	movl cmd_tbl(, %esi, 4), %eax
049	push eax	pushl %eax
050	call puts	call puts
051	add esp, 4	addl $4, %esp
052	mov ecx, [ebp - 4]	movl -4(%ebp), %ecx
053	inc esi	incl %esi
054	loop print_loop	loop print_loop
055		
056	jmp _exit	jmp _exit
057		
058	_exit:	_exit:
059	mov eax, 1	movl $1, %eax
060	mov ebx, 0	movl $0, %ebx
061	int 80h	int $0x80

Listing 5 shows a construct that repeats instructions in assembly. Naturally enough, it's called the repeat construct. In GAS, the repeat construct is started using the `.rept` directive (line 6). This directive has to be closed using an `.endr` directive (line 8). `.rept` is followed by a count in GAS that specifies the number of times the expression enclosed inside the `.rept/.endr` construct is to be repeated. Any instruction placed inside this construct is equivalent to writing that instruction `count` number of times, each on a separate line.

For example, for a count of 3:

```
.rept 3
    movl $2, %eax
.endr
```

This is equivalent to:

```
movl $2, %eax
movl $2, %eax
movl $2, %eax
```

In NASM, a similar construct is used at the preprocessor level. It begins with the `%rep` directive and ends with `%endrep`. The `%rep` directive is followed by an expression (unlike in GAS where the `.rept` directive is followed by a count):

```
%rep <expression>
   nop
%endrep
```

There is also an alternative in NASM, the `times` directive. Similar to `%rep`, it works at the assembler level, and it, too, is followed by an expression. For example, the above `%rep` construct is equivalent to this:

```
times <expression> nop
```

And this:

```
%rep 3
   mov eax, 2
%endrep
```

is equivalent to this:

```
times 3 mov eax, 2
```

and both are equivalent to this:

```
mov eax, 2
mov eax, 2
mov eax, 2
```

In Listing 5, the `.rept` (or `%rep`) directive is used to create a memory data area for 10 double words. The command line arguments are then accessed one by one from the stack and stored in the memory area until the command table gets full.

As for command line arguments, they are accessed similarly with both assemblers. ESP or the top of the stack stores the number of command line arguments supplied to a program, which is 1 by default (for no command line arguments). `esp + 4` stores the first command line argument, which is always the name of the program that was invoked from the command line. `esp + 8`, `esp + 12`, and so on store subsequent command line arguments.

Also watch the way the memory command table is being accessed on both sides in Listing 5. Here, memory indirect addressing mode (line 33) is used to access the command table along with an offset in ESI (and EDI) and a multiplier. Thus, `[cmd_tbl + esi * 4]` in NASM is equal to `cmd_tbl(, %esi, 4)` in GAS.

Conclusion

Even though the differences between these two assemblers are substantial, it's not that difficult to convert from one form to another. You might find that the AT&T syntax seems at first difficult to understand, but once mastered, it's as simple as the Intel syntax.

Related topics

- Consult the NASM and GAS manuals for complete introductions to these two assemblers:
  - GAS: GNU Assembler
  - NASM: Netwide Assembler
- Read an explanation of selection sort on Wikipedia.