REF:

<http://libra.cs.virginia.edu/~aaron/08-nasm/nasmexamples.html>

**NASM Examples**

**Getting Started**

Here is a very short NASM program that displays "Hello, World" on a line then exits. Like most programs on this page, you link it with a C library:

; ----------------------------------------------------------------------------

; helloworld.asm

;

; This is a Win32 console program that writes "Hello, World" on one line and

; then exits. It needs to be linked with a C library.

; ----------------------------------------------------------------------------

global \_main

extern \_printf

section .text

\_main:

push message

call \_printf

add esp, 4

ret

message:

db 'Hello, World', 10, 0

To assemble, link and run this program under Windows:

nasm -fwin32 helloworld.asm

gcc helloworld.obj

a

Under Linux, you'll need to remove the leading underscores from function names, and execute

nasm -felf helloworld.asm

gcc helloworld.o

./a.out

## Understanding Calling Conventions

If you are writing assembly language functions that will link with C, and you're using gcc, you must obey the gcc calling conventions. These are:

* Parameters are pushed on the stack, right to left, and are removed by the caller after the call.
* After the parameters are pushed, the call instruction is made, so when the called function gets control, the return address is at [esp], the first parameter is at [esp+4], etc.
* If you want to use any of the following registers: ebx, esi, edi, ebp, ds, es, ss, you must save and restore their values. In other words, these values must not change across function calls. When you make calls, you can assume these won't change (as long as everyone plays by the rules).
* A function that returns an integer value should return it in eax, a 64-bit integer in edx:eax, and a floating point value should be returned on the fpu stack top.

This program prints the first few fibonacci numbers, illustrating how registers have to be saved and restored:

; ----------------------------------------------------------------------------

; fib.asm

;

; This is a Win32 console program that writes the first 40 Fibonacci numbers.

; It needs to be linked with a C library.

; ----------------------------------------------------------------------------

global \_main

extern \_printf

section .text

\_main:

push ebx ; we have to save this since we use it

mov ecx, 40 ; ecx will countdown from 40 to 0

xor eax, eax ; eax will hold the current number

xor ebx, ebx ; ebx will hold the next number

inc ebx ; ebx is originally 1

print:

; We need to call printf, but we are using eax, ebx, and ecx. printf

; may destroy eax and ecx so we will save these before the call and

; restore them afterwards.

push eax

push ecx

push eax

push format

call \_printf

add esp, 8

pop ecx

pop eax

mov edx, eax ; save the current number

mov eax, ebx ; next number is now current

add ebx, edx ; get the new next number

dec ecx ; count down

jnz print ; if not done counting, do some more

pop ebx ; restore ebx before returning

ret

format:

db '%10d', 0

## Mixing C and Assembly Language

This program is just a simple function that takes in three integer parameters and returns the maximum value. It shows that the parameters will be at [esp+4], [esp+8] and [esp+12], and that the value gets returned in eax.

; ----------------------------------------------------------------------------

; maxofthree.asm

;

; NASM implementation of a function that returns the maximum value of its

; three integer parameters. The function has prototype:

;

; int maxofthree(int x, int y, int z)

;

; Note that only eax, ecx, and edx were used so no registers had to be saved

; and restored.

; ----------------------------------------------------------------------------

global \_maxofthree

section .text

\_maxofthree:

mov eax, [esp+4]

mov ecx, [esp+8]

mov edx, [esp+12]

cmp eax, ecx

cmovl eax, ecx

cmp eax, edx

cmovl eax, edx

ret

Here is a C program that calls the assembly language function.

/\*

\* callmaxofthree.c

\*

\* Illustrates how to call the maxofthree function we wrote in assembly

\* language.

\*/

#include <stdio.h>

int maxofthree(int, int, int);

int main() {

printf("%d\n", maxofthree(1, -4, -7));

printf("%d\n", maxofthree(2, -6, 1));

printf("%d\n", maxofthree(2, 3, 1));

printf("%d\n", maxofthree(-2, 4, 3));

printf("%d\n", maxofthree(2, -6, 5));

printf("%d\n", maxofthree(2, 4, 6));

return 0;

}

To assemble, link and run this two-part program (on Windows):

nasm -fwin32 maxofthree.asm

gcc callmaxofthree.c maxofthree.obj

a

## Command Line Arguments

You know that in C, main is just a plain old function, and it has a couple parameters of its own:

int main(int argc, char\*\* argv)

Here is a program that uses this fact to simply echo the commandline arguments to a program, one per line:

; ----------------------------------------------------------------------------

; echo.asm

;

; NASM implementation of a program that displays its commandline arguments,

; one per line.

; ----------------------------------------------------------------------------

global \_main

extern \_printf

section .text

\_main:

mov ecx, [esp+4] ; argc

mov edx, [esp+8] ; argv

top:

push ecx ; save registers that printf wastes

push edx

push dword [edx] ; the argument string to display

push format ; the format string

call \_printf

add esp, 8 ; remove the two parameters

pop edx ; restore registers printf used

pop ecx

add edx, 4 ; point to next argument

dec ecx ; count down

jnz top ; if not done counting keep going

ret

format:

db '%s', 10, 0

Note that as far as the C Library is concerned, command line arguments are always strings. If you want to treat them as integers, call atoi. Here's a neat program to compute xy.

; ----------------------------------------------------------------------------

; power.asm

;

; Command line application to compute x^y

; Syntax: power x y

; x and y are integers

; ----------------------------------------------------------------------------

global \_main

extern \_atoi

extern \_printf

section .text

\_main:

push ebx ; save the registers that must be saved

push esi

push edi

mov eax, [esp+16] ; argc (it's not at [esp+4] now :-))

cmp eax, 3 ; must have exactly two arguments

jne error1

mov ebx, [esp+20] ; argv

push dword [ebx+4] ; argv[1]

call \_atoi

add esp, 4

mov esi, eax ; x in esi

push dword [ebx+8]

call \_atoi ; argv[2]

add esp, 4

cmp eax, 0

jl error2

mov edi, eax ; y in edi

mov eax, 1 ; start with answer = 1

check:

test edi, edi ; we're counting y downto 0

jz gotit ; done

imul eax, esi ; multiply in another x

dec edi

jmp check

gotit: ; print report on success

push eax

push answer

call \_printf

add esp, 8

jmp done

error1: ; print error message

push badArgumentCount

call \_printf

add esp, 4

jmp done

error2: ; print error message

push negativeExponent

call \_printf

add esp, 4

done: ; restore saved registers

pop edi

pop esi

pop ebx

ret

answer:

db '%d', 10, 0

badArgumentCount:

db 'Requires exactly two arguments', 10, 0

negativeExponent:

db 'The exponent may not be negative', 10, 0

## Data Sections

The text section is read-only on most operating systems, so you might find the need for a data section. On most operating systems, the data section is only for initialized data, and you have a special .bss section for uninitialized data. Here is a program that averages the command line arguments, expected to be integers, and displays the result as a floating point number. Note that there is no instruction to push an 8-byte value, so we fake it by manipulating esp.

; ----------------------------------------------------------------------------

; average.asm

;

; NASM implementation of a program that treats all its command line arguments

; as integers, as displays their average as a floating point number. This

; program uses a data section to store intermediate results, not that it has

; to, but only to illustrate how data sections are used.

; ----------------------------------------------------------------------------

global \_main

extern \_printf

extern \_atoi

section .text

\_main:

mov ecx, [esp+4] ; argc

dec ecx ; don't count program name

jz nothingToAverage

mov [count], ecx ; save number of real arguments

mov edx, [esp+8] ; argv

accumulate:

push ecx ; save values across call to atoi

push edx

push dword [edx+ecx\*4] ; argv[ecx]

call \_atoi ; now eax has the int value of arg

add esp, 4

pop edx ; restore registers after atoi call

pop ecx

add [sum], eax ; accumulate sum as we go

dec ecx

jnz accumulate ; more arguments?

average:

fild dword [sum]

fild dword [count]

fdivp st1, st0 ; sum / count

sub esp, 8 ; make room for quotient on stack

fstp qword [esp] ; "push" quotient

push format ; push format string

call \_printf

add esp, 12 ; 4 bytes format, 8 bytes number

ret

nothingToAverage:

push error

call \_printf

add esp, 4

ret

section .data

count: dd 0

sum: dd 0

format: db '%.15f', 10, 0

error: db 'There are no command line arguments to average', 10, 0

## Recursion

Perhaps surprisingly, there's nothing out of the ordinary required to implement recursive functions. You push parameters on the stack, after all! Here's an example. In C

int factorial(int n) {

return (n <= 1) ? 1 : n \* factorial(n-1);

}

In assembly language:

; ----------------------------------------------------------------------------

; factorial.asm

;

; Illustration of a recursive function.

; ----------------------------------------------------------------------------

global \_factorial

section .text

\_factorial:

mov eax, [esp+4] ; n

cmp eax, 1 ; n <= 1

jnle L1 ; if not, go do a recursive call

mov eax, 1 ; otherwise return 1

jmp L2

L1:

dec eax ; n-1

push eax ; push argument

call \_factorial ; do the call, result goes in eax

add esp, 4 ; get rid of argument

imul eax, [esp+4] ; n \* factorial(n-1)

L2:

ret

## Local Variables

After entering a function, we can reserve space for local variables by decrementing the stack pointer. For example, the C function

int example(int x, int y) {

int a, b, c;

b = 7;

return x \* b + y;

}

can be translated as follows:

\_example:

sub esp, 12 ; make room for 3 ints

mov dword [esp+4], 7 ; b = 7

mov eax, [esp+16] ; x

imul eax, [esp+4] ; x \* b

add eax, [esp+20] ; x \* b + y

ret

After "sub esp, 12" the stack looks like:

+---------+

esp | a |

+---------+

esp+4 | b |

+---------+

esp+8 | c |

+---------+

esp+12 | retaddr |

+---------+

esp+16 | x |

+---------+

esp+20 | y |

+---------+

## Stack Frames

Sometimes it is a real pain to try to keep track of the offsets of your parameters and local variables because the stack pointer keeps changing. For example, in

int example(int x, int y) {

int a, b, c;

...

f(y, a, b, b, x);

...

}

you cannot translate the function call as

push dword [esp+16]

push dword [esp+4] ; WRONG! b is really now at [esp+8]

push dword [esp+4] ; WRONG! b is really now at [esp+12]

push dword [esp] ; WRONG! a is really now at [esp+12]

push dword [esp+20] ; WRONG! y is really now at [esp+36]

call f

For this reason, many functions use the ebp register to index the "stack frame" of local variables and parameters, like this:

push ebp ; must save old ebp

mov ebp, esp ; point ebp to this frame

sub esp, \_\_\_ ; make space for locals

...

mov esp, ebp ; clean up locals

pop ebp ; restore old ebp

ret

As long as you never change ebp throughout the function, all your local variables and parameters will always be at the same offset from ebp. The stack frame for our example function is now:

+---------+

ebp-12 | a |

+---------+

ebp-8 | b |

+---------+

ebp-4 | c |

+---------+

ebp | old ebp |

+---------+

ebp+4 | retaddr |

+---------+

ebp+8 | x |

+---------+

ebp+12 | y |

+---------+