NASM Examples

NASM is a pretty awesome assembler. Let's learn some NASM programming by example. These notes barely scratch the surface of what you can do, so after you've gone through this page, you'll need to hit the official NASM documentation.

Preliminaries

Please note: all of these examples, except for those in the last few sections, will **only run on a modern 64-bit Linux installation**.

Make sure both nasm and gcc are installed.

Getting Started

Our first program will use Linux system call 1 to write a message and Linux system call 60 to exit.

hello.asm

```
; ----------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ----------------------------------------------------------------------------------------

        global  _start

        section .text
_start:
        ; write(1, message, 13)
        mov     rax, 1                  ; system call 1 is write
        mov     rdi, 1                  ; file handle 1 is stdout
        mov     rsi, message            ; address of string to output
        mov     rdx, 13                 ; number of bytes
        syscall                         ; invoke operating system to do the write

        ; exit(0)
        mov     eax, 60                 ; system call 60 is exit
        xor     rdi, rdi                ; exit code 0
        syscall                         ; invoke operating system to exit
message:
        db      "Hello, World", 10      ; note the newline at the end
```

```
$ nasm -felf64 hello.asm && ld hello.o && ./a.out
Hello, World
```

Using a C Library

Remember how in C execution seems to start with "main"? That's because the C library actually has the `_start` label inside itself! The code at `_start` does some initialization, then it calls `main`, then it does some clean up, then it issues system call

60. So you just have to implement `main`. We can do that in assembly:

hola.asm

```
; ----------------------------------------------------------------------------------------
; Writes "Hola, mundo" to the console using a C library. Runs on Linux or any other system
; that does not use underscores for symbols in its C library. To assemble and run:
;
;     nasm -felf64 hola.asm && gcc hola.o && ./a.out
; ----------------------------------------------------------------------------------------

        global  main
        extern  puts

        section .text
main:                                   ; This is called by the C library startup code
        mov     rdi, message            ; First integer (or pointer) argument in rdi
        call    puts                    ; puts(message)
        ret                             ; Return from main back into C library wrapper
message:
        db      "Hola, mundo", 0        ; Note strings must be terminated with 0 in C
```

```
$ nasm -felf64 hola.asm && gcc hola.o && ./a.out
Hola, mundo
```

Understanding Calling Conventions

When writing code for 64-bit Linux that integrates with a C library, you must follow the calling conventions, explained fully in the AMD64 ABI Reference. You can also get this information from Wikipedia. The most important points are:

- From left to right, pass as many parameters as will fit in registers. The order in which registers are allocated, are:
  - For integers and pointers, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`.
  - For floating-point (float, double), `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7`.
- Additional parameters are pushed on the stack, right to left, and are to be *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 `[rsp]`, the first memory parameter is at `[rsp+8]`, etc.
- **The stack pointer `rsp` must be aligned to a 16-byte boundary before making a call**. Fine, but the process of making a call pushes the return address (8 bytes) on the stack, so when a function gets control, `rsp` is not aligned. You have to make that extra space yourself, by pushing something or subtracting 8 from `rsp`.
- The only registers that the called function is required to preserve (the calle-save registers) are: `rbp`, `rbx`, `r12`, `r13`, `r14`, `r15`. All others are free to be changed by the called function.
- The callee is also supposed to save the control bits of the XMCSR and the x87 control word, but x87 instructions are rare in 64-bit code so you probably don't have to worry about this.
- Integers are returned in `rax` or `rdx:rax`, and floating point values are returned in `xmm0` or `xmm1:xmm0`.

Here is a program that illustrates how registers have to be saved and restored:

fib.asm

```
; -----------------------------------------------------------------------------
; A 64-bit Linux application that writes the first 90 Fibonacci numbers. To
; assemble and run:
;
;     nasm -felf64 fib.asm && gcc fib.o && ./a.out
; -----------------------------------------------------------------------------

        global  main
        extern  printf
```

```
        section .text
main:
        push    rbx                     ; we have to save this since we use it

        mov     ecx, 90                 ; ecx will countdown to 0
        xor     rax, rax                ; rax will hold the current number
        xor     rbx, rbx                ; rbx will hold the next number
        inc     rbx                     ; rbx is originally 1
print:
        ; We need to call printf, but we are using rax, rbx, and rcx.  printf
        ; may destroy rax and rcx so we will save these before the call and
        ; restore them afterwards.

        push    rax                     ; caller-save register
        push    rcx                     ; caller-save register

        mov     rdi, format             ; set 1st parameter (format)
        mov     rsi, rax                ; set 2nd parameter (current_number)
        xor     rax, rax                ; because printf is varargs

        ; Stack is already aligned because we pushed three 8 byte registers
        call    printf                  ; printf(format, current_number)

        pop     rcx                     ; restore caller-save register
        pop     rax                     ; restore caller-save register

        mov     rdx, rax                ; save the current number
        mov     rax, rbx                ; next number is now current
        add     rbx, rdx                ; get the new next number
        dec     ecx                     ; count down
        jnz     print                   ; if not done counting, do some more

        pop     rbx                     ; restore rbx before returning
        ret
format:
        db  "%20ld", 10, 0
```

```
$ nasm -felf64 fib.asm && gcc fib.o && ./a.out
                   0
                   1
                   1
                   2
                   .
                   .
                   .
  679891637638612258
 1100087778366101931
 1779979416004714189
```

Mixing C and Assembly Language

This program is just a simple function that takes in three integer parameters and returns the maximum value.

maxofthree.asm

```
; -----------------------------------------------------------------------------
; A 64-bit function that returns the maximum value of its three 64-bit integer
; arguments.  The function has signature:
;
;   int64_t maxofthree(int64_t x, int64_t y, int64_t z)
;
; Note that the parameters have already been passed in rdi, rsi, and rdx.  We
; just have to return the value in rax.
; -----------------------------------------------------------------------------
```

```
        global  maxofthree
        section .text
maxofthree:
        mov     rax, rdi                ; result (rax) initially holds x
        cmp     rax, rsi                ; is x less than y?
        cmovl   rax, rsi                ; if so, set result to y
        cmp     rax, rdx                ; is max(x,y) less than z?
        cmovl   rax, rdx                ; if so, set result to z
        ret                             ; the max will be in rax
```

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

callmaxofthree.c

```c
/*
 * A small program that illustrates how to call the maxofthree function we wrote in
 * assembly language.
 */

#include <stdio.h>
#include <inttypes.h>

int64_t maxofthree(int64_t, int64_t, int64_t);

int main() {
    printf("%ld\n", maxofthree(1, -4, -7));
    printf("%ld\n", maxofthree(2, -6, 1));
    printf("%ld\n", maxofthree(2, 3, 1));
    printf("%ld\n", maxofthree(-2, 4, 3));
    printf("%ld\n", maxofthree(2, -6, 5));
    printf("%ld\n", maxofthree(2, 4, 6));
    return 0;
}
```

```
$ nasm -felf64 maxofthree.asm && gcc callmaxofthree.c maxofthree.o && ./a.out
1
2
3
4
5
6
```

Command Line Arguments

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

```c
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

```
; -----------------------------------------------------------------------------
; A 64-bit program that displays its command line arguments, one per line.
;
; On entry, rdi will contain argc and rsi will contain argv.
; -----------------------------------------------------------------------------

        global  main
        extern  puts
        section .text
main:
        push    rdi                     ; save registers that puts uses
        push    rsi
        sub     rsp, 8                  ; must align stack before call
```

```
        mov     rdi, [rsi]              ; the argument string to display
        call    puts                    ; print it

        add     rsp, 8                  ; restore %rsp to pre-aligned value
        pop     rsi                     ; restore registers puts used
        pop     rdi

        add     rsi, 8                  ; point to next argument
        dec     rdi                     ; count down
        jnz     main                    ; if not done counting keep going

        ret
```

```
$ nasm -felf64 echo.asm && gcc echo.o && ./a.out dog 22 -zzz "hi there"
./a.out
dog
22
-zzz
hi there
```

A Longer Example

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 x^y.

power.asm

```
; ----------------------------------------------------------------------------
; A 64-bit command line application to compute x^y.
;
; Syntax: power x y
; x and y are (32-bit) integers
; ----------------------------------------------------------------------------

        global  main
        extern  printf
        extern  puts
        extern  atoi

        section .text
main:
        push    r12                     ; save callee-save registers
        push    r13
        push    r14
        ; By pushing 3 registers our stack is already aligned for calls

        cmp     rdi, 3                  ; must have exactly two arguments
        jne     error1

        mov     r12, rsi                ; argv

; We will use ecx to count down form the exponent to zero, esi to hold the
; value of the base, and eax to hold the running product.

        mov     rdi, [r12+16]           ; argv[2]
        call    atoi                    ; y in eax
        cmp     eax, 0                  ; disallow negative exponents
        jl      error2
        mov     r13d, eax               ; y in r13d

        mov     rdi, [r12+8]            ; argv
        call    atoi                    ; x in eax
        mov     r14d, eax               ; x in r14d
```

```
        mov     eax, 1                  ; start with answer = 1
check:
        test    r13d, r13d              ; we're counting y downto 0
        jz      gotit                   ; done
        imul    eax, r14d               ; multiply in another x
        dec     r13d
        jmp     check
gotit:                                  ; print report on success
        mov     rdi, answer
        movsxd  rsi, eax
        xor     rax, rax
        call    printf
        jmp     done
error1:                                 ; print error message
        mov     edi, badArgumentCount
        call    puts
        jmp     done
error2:                                 ; print error message
        mov     edi, negativeExponent
        call    puts
done:                                   ; restore saved registers
        pop     r14
        pop     r13
        pop     r12
        ret

answer:
        db      "%d", 10, 0
badArgumentCount:
        db      "Requires exactly two arguments", 10, 0
negativeExponent:
        db      "The exponent may not be negative", 10, 0
```

```
$ nasm -felf64 power.asm && gcc -o power power.o
$ ./power 2 19
524288
$ ./power 3 -8
The exponent may not be negative
$ ./power 1 500
1
$ ./power 1
Requires exactly two arguments
```

Floating Point Instructions

Floating-point arguments go int the xmm registers. Here is a simple function for summing the values in a double array:

sum.asm

```
; ----------------------------------------------------------------------------
; A 64-bit function that returns the sum of the elements in a floating-point
; array. The function has prototype:
;
;   double sum(double[] array, uint64_t length)
; ----------------------------------------------------------------------------

        global  sum
        section .text
sum:
        xorsd   xmm0, xmm0              ; initialize the sum to 0
        cmp     rsi, 0                  ; special case for length = 0
        je      done
next:
        addsd   xmm0, [rdi]             ; add in the current array element
        add     rdi, 8                  ; move to next array element
        dec     rsi                     ; count down
```

```
        jnz     next                    ; if not done counting, continue
done:
        ret                             ; return value already in xmm0
```

A C program that calls it:

callsum.c

```
/*
 * Illustrates how to call the sum function we wrote in assembly language.
 */

#include <stdio.h>
#include <inttypes.h>

double sum(double[], uint64_t);

int main() {
    double test[] = {
        40.5, 26.7, 21.9, 1.5, -40.5, -23.4
    };
    printf("%20.7f\n", sum(test, 6));
    printf("%20.7f\n", sum(test, 2));
    printf("%20.7f\n", sum(test, 0));
    printf("%20.7f\n", sum(test, 3));
    return 0;
}
```

```
$ nasm -felf64 sum.asm && gcc sum.o callsum.c && ./a.out
          26.7000000
          67.2000000
           0.0000000
          89.1000000
```

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.

average.asm

```
; -----------------------------------------------------------------------------
; 64-bit program that treats all its command line arguments as integers and
; 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   atoi
        extern   printf
        default  rel

        section  .text
main:
        dec      rdi                    ; argc-1, since we don't count program name
        jz       nothingToAverage
        mov      [count], rdi           ; save number of real arguments
accumulate:
        push     rdi                    ; save register across call to atoi
        push     rsi
        mov      rdi, [rsi+rdi*8]       ; argv[rdi]
        call     atoi                   ; now rax has the int value of arg
```

```
        pop     rsi                     ; restore registers after atoi call
        pop     rdi
        add     [sum], rax              ; accumulate sum as we go
        dec     rdi                     ; count down
        jnz     accumulate              ; more arguments?
average:
        cvtsi2sd xmm0, [sum]
        cvtsi2sd xmm1, [count]
        divsd   xmm0, xmm1              ; xmm0 is sum/count
        mov     rdi, format             ; 1st arg to printf
        mov     rax, 1                  ; printf is varargs, there is 1 non-int argument

        sub     rsp, 8                  ; align stack pointer
        call    printf                  ; printf(format, sum/count)
        add     rsp, 8                  ; restore stack pointer

        ret

nothingToAverage:
        mov     rdi, error
        xor     rax, rax
        call    printf
        ret

        section  .data
count:  dq      0
sum:    dq      0
format: db      "%g", 10, 0
error:  db      "There are no command line arguments to average", 10, 0
```

```
$ nasm -felf64 average.asm && gcc average.o && ./a.out 19 8 21 -33
3.75
$ nasm -felf64 average.asm && gcc average.o && ./a.out
There are no command line arguments to average
```

Recursion

Perhaps surprisingly, there's nothing out of the ordinary required to implement recursive functions. You just have to be careful to save registers, as usual.

factorial.asm

```
; ----------------------------------------------------------------------------
; An implementation of the recursive function:
;
;   uint64_t factorial(uint64_t n) {
;       return (n <= 1) ? 1 : n * factorial(n-1);
;   }
; ----------------------------------------------------------------------------

        global  factorial

        section .text
factorial:
        cmp     rdi, 1                  ; n <= 1?
        jnbe    L1                      ; if not, go do a recursive call
        mov     rax, 1                  ; otherwise return 1
        ret
L1:
        push    rdi                     ; save n on stack (also aligns %rsp!)
        dec     rdi                     ; n-1
        call    factorial               ; factorial(n-1), result goes in %rax
        pop     rdi                     ; restore n
        imul    rax, rdi                ; n * factorial(n-1), stored in %rax
        ret
```

An example caller:

callfactorial.c

```
/*
 * An application that illustrates calling the factorial function defined elsewhere.
 */

#include <stdio.h>
#include <inttypes.h>

uint64_t factorial(uint64_t n);

int main() {
    for (uint64_t i = 0; i < 20; i++) {
        printf("factorial(%2lu) = %lu\n", i, factorial(i));
    }
    return 0;
}
```

```
$ nasm -felf64 factorial.asm && gcc -std=c99 factorial.o callfactorial.c && ./a.out
factorial( 0) = 1
factorial( 1) = 1
factorial( 2) = 2
factorial( 3) = 6
factorial( 4) = 24
factorial( 5) = 120
factorial( 6) = 720
factorial( 7) = 5040
factorial( 8) = 40320
factorial( 9) = 362880
factorial(10) = 3628800
factorial(11) = 39916800
factorial(12) = 479001600
factorial(13) = 6227020800
factorial(14) = 87178291200
factorial(15) = 1307674368000
factorial(16) = 20922789888000
factorial(17) = 355687428096000
factorial(18) = 6402373705728000
factorial(19) = 121645100408832000
```

SIMD Parallelism

The XMM registers can do arithmetic on floating point values one opeation at a time or multiple operations at a time. The operations have the form:

```
operation  xmmregister_or_memorylocation, xmmregister
```

For floating point addition, the instructions are:

```
addpd — do 2 double-precision additions
addps — do just one double-precision addition, using the low 64-bits of the registe
addsd — do 4 single-precision additions
addss — do just one single-precision addition, using the low 32-bits of the registe
```

TODO - show a function that processes an array of floats, 4 at a time.

Saturated Arithmetic

The XMM registers can also do arithmetic on integers. The instructions have the form:

```
operation  xmmregister_or_memorylocation, xmmregister
```

For integer addition, the instructions are:

```
paddb — do 16 byte additions
paddw — do 8 word additions
paddd — do 4 dword additions
paddq — do 2 qword additions
paddsb — do 16 byte additions with signed saturation (80..7F)
paddsw — do 8 word additions with unsigned saturation (8000..7FFF)
paddusb — do 16 byte additions with unsigned saturation (00..FF)
paddusw — do 8 word additions with unsigned saturation (00..FFFF)
```

TODO - SHOW AN EXAMPLE

Graphics

TODO

Local Variables and Stack Frames

First, please read Eli Bendersky's article That overview is more complete than my brief notes.

When a function is called the caller will first put the parameters in the correct registers then issue the `call` instruction. Additional parameters beyond those covered by the registers will be pushed on the stack prior to the call. The call instruction puts the return address on the top of stack. So if you have the function

```
int64_t example(int64_t x, int64_t y) {
    int64_t a, b, c;
    b = 7;
    return x * b + y;
}
```

Then on entry to the function, x will be in edi, y will be in esi, and the return address will be on the top of the stack. Where can we put the local variables? An easy choice is on the stack itself, though if you have enough regsters, use those.

If you are running on a machine that respect the standard ABI, you can leave rsp where it is and access the "extra parameters" and the local variables directly from rsp for example:

```
          +----------+
rsp-24    |    a     |
          +----------+
rsp-16    |    b     |
          +----------+
rsp-8     |    c     |
          +----------+
rsp       | retaddr  |
          +----------+
rsp+8     | caller's |
          | stack    |
          | frame    |
          | ...      |
          +----------+
```

So our function looks like this:

```
        global  example
        section .text
example:
        mov     qword [rsp-16], 7
        mov     rax, rdi
        imul    rax, [rsp+8]
        add     rax, rsi
        ret
```

If our function were to make another call, you would have to adjust rsp to get out of the way at that time.

On Windows you can't use this scheme because if an interrupt were to occur, everything above the stack pointer gets plastered. This doesn't happen on most other operating systems because there is a "red zone" of 128 bytes past the stack pointer which is safe from these things. In this case, you can make room on the stack immediately:

```
example:
        sub rsp, 24
```

so our stack looks like this:

```
               +----------+
        rsp    |    a     |
               +----------+
        rsp+8  |    b     |
               +----------+
        rsp+16 |    c     |
               +----------+
        rsp+24 | retaddr  |
               +----------+
        rsp+32 | caller's |
               | stack    |
               | frame    |
               | ...      |
               +----------+
```

Here's the function now. Note that we have to remember to replace the stack pointer before returning!

```
        global  example
        section .text
example:
        sub     rsp, 24
        mov     qword [rsp+8], 7
        mov     rax, rdi
        imul    rax, [rsp+8]
        add     rax, rsi
        add     rsp, 24
        ret
```

Using NASM on OS X

TODO

Using NASM on Windows

TODO