64-Bit NASM Notes

- The transition from 32- to 64-bit architectures is no joke, as anyone who has wrestled with 32/64 bit incompatibilities will attest
- We note here some key differences between 32- and 64-bit Intel assembly language programming, both in general and with NASM specifically
- It's a good idea to know, for various operating systems, how to detect the underlying version/architecture (on Unix-like platforms, `uname -a` does the trick)

Invoking 64-Bit NASM

- NASM became 64-bit capable as of version 2.0: invoke `nasm -v` to check the version you're running
- When assembling, make sure to specify a 64-bit format
  - `elf64` for most 64-bit Linux architectures
  - `macho64` for 64-bit Mac OS X
  - `win64` for 64-bit Windows
- Given the right object files, no command changes should be necessary when linking via `gcc`

Registers

- The primary new capability in 64-bit architectures is the ability to operate on a quadword's worth of data in a single instruction
- Addressable memory, both virtual and physical, becomes larger by virtue of 64-bit pointers/addresses
- Structurally, most registers are larger (64 bits wide, duh), and there are more of them (16 general-purpose registers vs. 8 in 32-bit Intel CPUs)

- Registers `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`, `esi`, and `edi` are now 64-bit: `rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsp`, `rsi`, `rdi`
- The new general-purpose registers are `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, and `r15` — these are also available in 32-bit flavors `r8d`–`r15d`
- Most of the time, operands that are smaller than 64 bits zero-extend to 64 bits
- The default operand size is 32 bits — except when pushing/popping the stack: that's 64- or 16-bit only
- When in doubt, consult Chapter 3 of the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1

Calling Conventions

- Calling conventions are platform-specific, each with official documentation — typically called the application binary interface (ABI)
- For Linux, Windows, and Mac OS X respectively, the specifications can be found at:
  - `http://www.x86-64.org/documentation/abi.pdf`
  - `http://msdn2.microsoft.com/en-gb/library/ms794533.aspx`
  - `http://developer.apple.com/Mac/library/documentation/DeveloperTools/Conceptual/LowLevelABI`

Some calling convention highlights on 64-bit Linux:

- Integer/pointer parameters are placed, in order, in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`
- Floating-point arguments go to the `xmm` registers
- Variable-argument subroutines require a value in `rax` for the number of vector registers used
- Registers `rbp`, `rbx`, and `r12` through `r15` are "caller-owned" — the called function must preserve them (either don't touch them, or save-and-restore via the stack or other mechanism)
- Integer/pointer return values are placed in `rax` or possibly `rdx`; floating point goes in `xmm0` or `xmm1`

64-Bit Examples

- The following listings include direct conversions of some of the 32-bit examples in Prof. Toal's `x86assembly` and `nasmexamples` pages to 64-bit Linux
- Note how, aside from calling conventions and selected conversion to 64-bit registers, not much has actually changed — i.e., the main concepts of good assembly language programming remain the same
- Conversion to other 64-bit Intel operating systems is left as an interesting and beneficial exercise **:)**

Of course, we start with `helloworld`…

```
        global  _start

        section .text

_start:
        ; write(1, message, 13)
        mov     eax, 4          ; system call 4 is write
        mov     ebx, 1          ; file handle 1 is stdout
        mov     ecx, message    ; address of string to output
        mov     edx, 13         ; number of bytes
        int     80h

        ; exit(0)
        mov     eax, 1          ; system call 1 is exit
        mov     ebx, 0          ; we want return code 0
        int     80h

message:
        db      "Hello, World", 10
```

No changes here, since we use interrupts instead of subroutines!

The version that uses `printf` is another story: compare this to the 32-bit version…

```
        global  main
        extern  printf

        section .text
main:   mov     rdi, message  ; rdi gets the first argument (a pointer)

        xor     rax, rax      ; printf has a variable number of arguments,
                              ; so rax needs to be set to the number of
                              ; vector registers used...zero in this case
        call    printf
        ret

message:
        db      'Hello, World', 10, 0
```

`powers.asm` needs a similar makeover since it also uses `printf` — note the use of the stack for register preservation

```
        extern  printf
        global  main

        section .data

format:
        db      '%d', 10, 0

        section .text

main:
        mov     esi, 1       ; current value
        mov     edi, 31      ; counter

L1:
        push    rsi          ; save registers
        push    rdi

        mov     rdi, format  ; address of format string

        ; second argument, the current number, is already in rsi

        xor     eax, eax     ; zero vector registers (eax is OK)
        call    printf

        pop     rdi          ; restore registers
        pop     rsi

        add     esi, esi     ; double value
        dec     edi          ; keep counting
        jne     L1

        ret
```

64-bit `fib.asm` must preserve the caller-owned `rbx` register

```
        global  main
        extern  printf

        section .text

main:
        push    rbx             ; we have to save this since we use it

        ; 32-bit operands will zero-extend to 64 bits

        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    rax             ; 32-bit stack operands are not encodable
        push    rcx             ; in 64-bit mode, so we use the "r" names

        mov     rdi, format     ; arg 1 is a pointer
        mov     rsi, rax        ; arg 2 is the current number
        xor     eax, eax        ; no vector registers in use
        call    printf

        pop     rcx
        pop     rax

        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     rbx             ; restore ebx before returning

        ret

format:
        db      '%10d', 10, 0
```

`maxofthree` in 32- and 64-bit incarnations…

```
        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
```

```
        global  maxofthree

        section .text

maxofthree:
        cmp     edi, esi  ; compare args 1 and 2
        cmovl   edi, esi  ; set edi to the larger
        cmp     edi, edx  ; compare against arg 3
        cmovl   edi, edx  ; set edi to the larger
        mov     eax, edi  ; return value in rax
        ret
```

(note how we can use the operands right away; return value remains expected in `eax`)

…both work with the same C source (why?).

```
#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;
}
```

64-bit does not change how `main` is still "just a function" — but accordingly, command line arguments need to be processed using the new ABI, as seen in 64-bit `echo.asm`

```
        global  main
        extern  printf

        section .text

main:
        mov     rcx, rdi        ; argc
        mov     rdx, rsi        ; argv

top:
        push    rcx             ; save registers that printf wastes
        push    rdx

        mov     rdi, format     ; the format string
        mov     rsi, [rdx]      ; the argument string to display
        xor     rax, rax        ; zero vector registers
        call    printf

        pop     rdx             ; restore registers printf used
        pop     rcx

        add     rdx, 8          ; point to next argument
        dec     rcx             ; count down
        jnz     top             ; if not done counting keep going

        ret

format:
        db      '%s', 10, 0
```