WRITING A BARE-METAL
8086 BOOTLOADER

A Practical Tutorial

By

0xLeo (github.com/0xleo)

May 3, 2019

DRAFT X.YY
Missing:

Contents

1 Writing a tiny bootloader

1.1 The power up

As soon as the magical power button is pressed on the computer, it starts working. The motherboard sends a signal to the power supply device. After receiving the signal, the power supply provides the proper amount of electricity to the computer. Once the motherboard receives the power good signal, it tries to start the CPU. The CPU resets all leftover data in its registers and sets up predefined values for each of them [1].

The 80386 CPU and later CPUs define the following predefined data in CPU registers after the computer resets:

```
IP          0xfff0
CS selector 0xf000
CS base     0xffff0000
```

1.2 Reset vector

The job of these register values is to form the next instruction on the address but, which in this case is called reset vector. Reset vector is part (one of the two) of the BIOS ROM. In turn, the reset vector contains some simple code which jumps to the main BIOS, where the CPU enters in real mode.

1.3 Real mode – entering the BIOS

In real mode (App. A.1), the registers can only address 1 MB ($2^{20} bytes$)of memory. Because the 8086 registers are 16-bit, each can address only 2^{16} bytes (64 kB). One register cannot achieve that, therefore two are used, and the method to obtain the physical address is called memory segmentation (App. A.1). Memory segmentation is used to make use of all the address space available. All memory is divided into small, fixed-size segments of 65536 bytes (64 KB). Since we cannot address memory above 64 KB with 16-bit registers, an alternate method was devised. The physical address is obtained given a segment selector register (REG) and an offset register (OFF) as:

$$PHY = 16 \cdot SEG + OFF \tag{1.1}$$

To prevent addressing locations higher than 2^{20} bytes, only the first 20 bits of the data address bus are used, called lines A0-A19. Line A20 therefore has special importance and is disabled in real mode (A.2). Now the BIOS starts, as mentioned before in real mode.

1.4 BIOS structure and flow

After initializing and checking the hardware, the BIOS needs to find a bootable device. A boot order is stored in the BIOS configuration, controlling which devices the BIOS attempts to boot from. When attempting to boot from a hard drive, the BIOS tries to find a boot sector. On hard drives partitioned with an MBR partition layout (App. A.4), the boot sector is stored in the first 446 bytes of the first sector, where each sector is 512 bytes. The final two bytes of the first sector are `0x55` and `0xaa` (signature bytes), which designates to the BIOS that this device is bootable. If these bytes are read, then the BIOS will jump to address `0x7c00`, transferring control to the bootloader.

1.5 Writing a bootloader

We will write the first part of the BIOS following the instructions from Alex Parker's tutorial [2]. The boot sector of the x86 BIOS is always located at the magic number address `0x7c00`. A typical memory layout after boot is illustrated below.

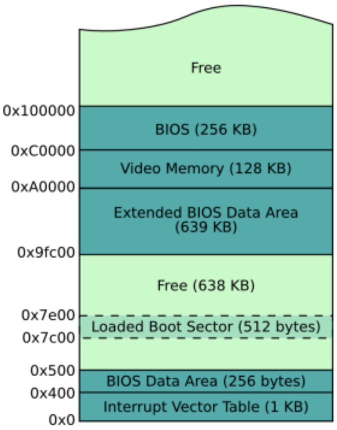

Free
0x100000
BIOS (256 KB)
0xC0000
Video Memory (128 KB)
0xA0000
Extended BIOS Data Area (639 KB)
0x9fc00
Free (638 KB)
0x7e00
Loaded Boot Sector (512 bytes)
0x7c00
0x500
BIOS Data Area (256 bytes)
0x400
Interrupt Vector Table (1 KB)
0x0

Fig. 1. Typical memory layout (real mode) after boot [3].

The tasks the bootloader code does are the following.

1. Specify start address; in Nasm, we instruct the program to start at this magic address by `org 0x7c00`.
2. Disable hardware interrupts, such as those from peripherals (nasm).
3. Parse the string and call an interrupt (`int 10h`) to print it character by character.
4. When done, pad the remaining bytes expect the last two with `512 - offsetFromBeginningOfSection - 2` zeros. Finally, define the signature bytes `0xaa` and `0x55`.

Listing 1: Hello world bootloader. (src/boot.nasm).

```
; comments by Eugene Obrezkov

org 0x7C00                          ; BIOS loads our programm at this address
bits 16                             ; We're working at 16-bit mode here

start:
    cli                         ; Disable the interrupts
    mov si, msg                 ; SI now points to our message
    mov ah, 0x0E                ; Indicate BIOS we're going to print chars
.loop    lodsb                  ; Loads SI into AL and increments SI [next char]
    or al, al                   ; Checks if the end of the string
    jz halt                     ; Jump to halt if the end
    int 0x10                    ; Otherwise, call interrupt for printing the char
    jmp .loop                   ; Next iteration of the loop

halt:    hlt                        ; CPU command to halt the execution
msg:     db "Hello, World!", 0      ; Our actual message to print

;; Magic numbers
times 510 - ($ - $$) db 0
dw 0xAA55
```

This is compiled by nasm (Nasm assembler details in the code explained in App **TODO!**) by:

```
$ nasm -f bin boot.nasm -o boot.bin
```

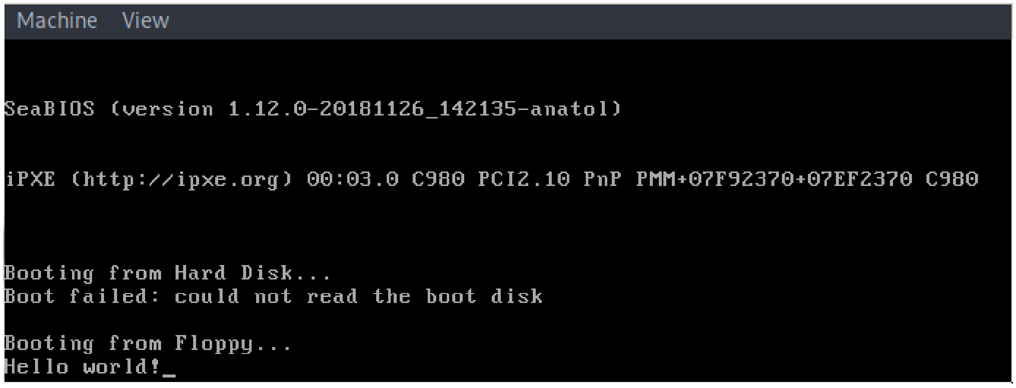
The executable code is padded with zeros and ends with the signature bytes as desired, which is seen by running

```
$ hexdump boot.bin
0000000 10be b47c ac0e c008 0474 10cd f7eb f4fa
0000010 6548 6c6c 206f 6f77 6c72 2164 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
00001f0 0000 0000 0000 0000 0000 0000 0000 aa55
```

The last step is to run this bootloader by telling qemu to boot from a floppy disk:

```
$ qemu-system-x86_64 -fda boot1.bin
```

The result is:



```
SeaBIOS (version 1.12.0-20181126_142135-anatol)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92370+07EF2370 C980



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Hello world!_
```

Fig. 2. Booting the executable with qemu.

A Appendices

A.1 The need of memory segmentation in 8086

A.1.1 Virtual to physical address

Segmentation means dividing the memory into logically different parts called segments. 8086 has a 20-bit address bus, often referred to as A0-19, hence it can access $2^{20} = 1MB$ memory, i.e. addresses from 0 to $2^{20} - 1 = FFFFF\ H$. Intel's 8086 model uses 16-bit registers and using one such register it can access up to 2^{16} memory locations. Two registers 16-bit registers are used to address the whole physical memory, called segment register (SEG) and offset register (OFF). The 1 MB of physical memory must be divided in 16 logical blocks of 64 kB each. So to get the physical address, we multiply SEG by 16 and add OFF to get the exact offset;

$$PHY = 16 * SEG + OFF = 10\ H * SEG + OFF \tag{A.1}$$

Note that in the hardware, multiplication of SEG by 16 is implemented by left shift by 4 so the previous equation can be rewritten:

$$PHY = SEG << 4 + OFF \tag{A.2}$$

The physical memory is divided into 4 segments, each which has a fixed base address (starting address) and a role;

- Code Segment (CS)
- Stack Segment (SS)
- Data Segment (DS)
- Extra Segment (ES)

Example A.1. *Suppose the Data Segment holds the Base Aaddress as 1000 H and the data you need is present in the 0020 H memory location (Offset) of the Data Segment*

Solution A.1. *Base address is multiplied by 16. Write it in binary and left shift it by 4 to find the product.*

```
SEG = 1000 H
1000H  = 0001 0000 0000b
1000H * 16 =
           = 0000 0001 0000 0000b << 4
           = 0001 0000 0000 0000b
```

Now add to it the offset `20H = 10 0000 b`

```
     0001 0000 0000 0000b
 +   0000 0000 0010 0000b
   = 0001 0000 0010 0000b
   = 10020H
```

That's the final physical address. ■

Finally, a single physical address can be represented by many different segment and offset address combinations. For instance, for the virtual address `0040:0049`, the physical address is `40H * 16 + 49H = 449 H` (when we multiply by 16 in decimal – 10 in hex, we simply add a 0 at the end, similar to multiplying with 10 in decimal). The same address can be represted by the segment:offset pair `0000:0449H`.

Segmented memory is a nice way to give 8086 programs more memory than they would normally be able to access using 16-bit registers.

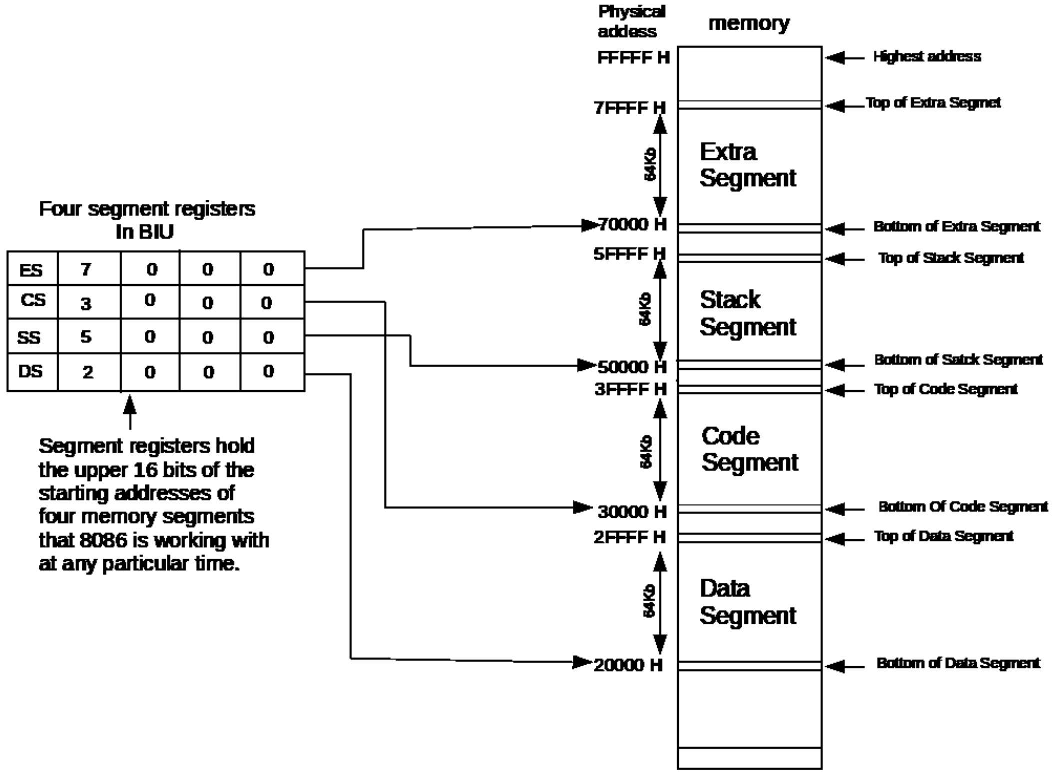
A.1.2 Segment registers

8086 has four 16-bit segment registers that point to special memory locations.

- Code segment register (CS): points to the memory location where the executable program is stored
- Data segment register (DS): points to the data segment of the memory where the data (e.g. program variables) is stored.

- Extra Segment Register (ES): also refers to a segment in the memory which is another data segment in the memory, it's up to the coder to define its usage.
- Stack Segment Register (SS): is used for addressing stack segment of the memory. It points to information such as function return addresses, arguments, local variables, etc.

As mentioned in the previous section, a segment is a logical memmory unit that contains up to 64 kB contiguous memory locations. The 8086 does not work the whole 1MB memory at any given time – only with four 64 kB segments within the whole 1MB memory. The figure below shows how the four segment registers map to 64 kB memory blocks.



ES	7	0	0	0
CS	3	0	0	0
SS	5	0	0	0
DS	2	0	0	0

Fig. 3. How segment registers map to memory locations.

A.2 Transition from real to protected mode and address line A20

When the system boots up and enters BIOS, it runs in real mode. In real mode, the system only has access to 1 MB of memory therefore needs a 20-bit data bus (lines A0-19). Predecessors of the 8086 were only using 1 MB of RAM, so a 20-bit data address would suffice to run applications. When these applications were ported to newer architectures than can access more than 1 MB of RAM, the line A20 had to be turned off for compatibility reasons.

Real mode prevents the registers from pointing to physical addresses larger than 1 MB. That means that whatever address is generated by $SEG * 16 + OFF$ is always truncated to its lower 20 bits (2^{20} = 1 MB). Real mode therefore enables lines A0-19 and keeps A20 disabled, since the highest accessible address in real mode is $(2^{16}-1)\cdot 16+(2^{16}-1) = 2^{20}-2^4+2^{16}-1 < 2^{21}$, which fits in 21 bits. In protected mode, A20, 21, ..., depending on the available memory are enabled. Other tasks are also permormed, such as disabling BIOS interrupts, enabling the Global Descriptor Table with segment descriptors suitable for code, data, and stack. More details about how protected mode starts in **TODO:** ref . Let's see an example.

For example, when `0xFFFF:0x000F=0xFFFFF`, which fits in 20 bits. If the offset is increased by l, the address is `0xFFFF:0x0010` which is `0x100000` (more than 20 bits)! In this case, 8086 will wrap around the address by removing bit 20, so `0xFFFF:0x0010` shall refer to byte `0x00000`, `0xFFFF:0x0011` shall refer to $0x00001$, and so on.

A.3 Location and role of reset vector

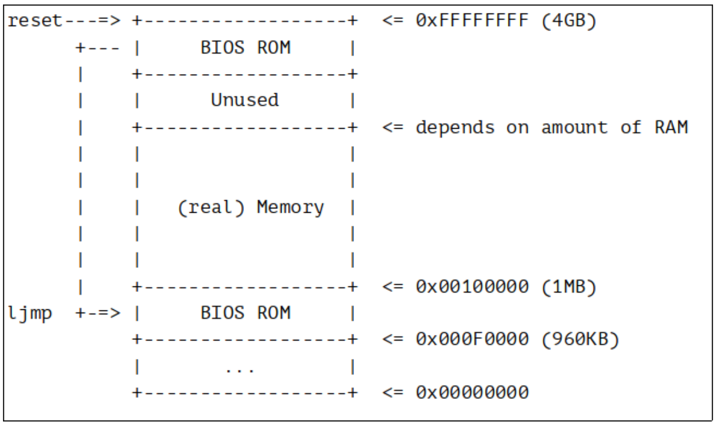
After power up, all x86 CPUs are in real mode but with a strange behaviour until a CS assignment is found! As already mentioned, for the 80386 and later CPUs, the following data are defined in the registers during after power up:

```
IP          0xfff0
CS base     0xffff0000
CS selector 0xf000
```

When fetching a new instruction (being in whatever mode: real, protected, etc) it seems that the hardware addressing logic is always using some CS cache register values to figure out what address to place on the Address Bus pins. The next instruction is:

```
Next_Instruction_Address_on_Bus= CS_segment_start_address + EIP
```

Substututing the CS and EIP values, we get that the boot instruction is `FFFF0000H + 0000FFF0H = FFFFFFF0H`. This is 16 bytes below 4 GM, so have we left real mode? This calculated address only contains far jump to the memory location mapped to the system BIOS entry point (0x0000: 0x000F0000h). The value where we jump, if `CS=XYZW H`, is (use the left shift formula) `000X YWZ0 + CSselector`, which is within the 1 MB area. For example, `CS=0xf000, IP=0xe05b`, the the next ip (instruction pointer) is `0xfe05b = 0xf000*16 + 0xe05b` (real mode). The jump is illustrated below.



```
reset---=> +------------------+  <= 0xFFFFFFFF (4GB)
      +--- |     BIOS ROM     |
      |    +------------------+
      |    |      Unused      |
      |    +------------------+  <= depends on amount of RAM
      |    |                  |
      |    |                  |
      |    |   (real) Memory  |
      |    |                  |
      |    |                  |
      |    +------------------+  <= 0x00100000 (1MB)
ljmp  +-=> |     BIOS ROM     |
           +------------------+  <= 0x000F0000 (960KB)
           |       ...        |
           +------------------+  <= 0x00000000
```

Fig. 4. The jump from the “reset vector” (top of RAM) to BIOS entry point.

To summarise;

Definition A.1. *The point in RAM pointed after power on which contains the instruction that jumps to the BIOS entry point is called* `reset vector`.

This behaviour is to assure that the power-up code will always be at the last corner of RAM. That way it will leave alone plenty of RAM for future OS disposal

A.4 The MBR partition

After jumping in the BIOS entry point, the BIOS starts. A lot of devices use MBR layout for the boot code.

Traditionally, a hard drive is formatted into 512 byte sectors (although larger sectors, such as 4 kB have recently been introduces). All the sectors on a disk platter that can be read without moving the head constitute a track. Disks usually have more than one platter. The collection of tracks on the various platters that can be read without moving the head is called a cylinder. The geometry of a hard drive is expressed in cylinders, tracks (or heads) per cylinder, and sectors/track.

> **Definition A.2.** *Cylinder 0, head 0 and sector 1 of hard disk are called master boot sector, which is also called master boot record (MBR). This record takes up 512 bytes.*

When hard disk starts, it is used to transfer system control right to the certain operating system partition which is user-specified and registered in partition table. The content of MBR is written into the certain sector by partitioning software when it partitions. MBR doesn't belong to any operating system and doesn't change with the changes of operating systems. More details about the MBR booting process in the next section.

A.4.1 How does the x86 MBR BIOS boot?

Legacy boot for x86 Linux has three steps:

1. The BIOS looks at the first block of each drive until it finds a Master Boot Record (MBR). This consists of 440 bytes of real-mode code, 6 ignored bytes, 4 instances of 16-byte primary partition records, and 2 signature bytes, 0x55 and 0xAA. That's 512 bytes. BIOS copies this block into RAM and starts executing the first byte. This all happens in x86 Real Mode.

2. Those first 440 bytes of MBR code are responsible for bootstrapping the rest of the OS. It searches the four partition table entries, finds a partition flagged as bootable, copies the first 512 bytes from that partition (the so-called Volume Boot Record or VBR) into RAM, and jumps there. That code then continues the boot process in some unspecified way – typically that VBR code (as well as the MBR code) is generated by and installed by grub, lilo, syslinux, or some similar bootloader.

3. The bootloader code identifies the kernel. It also creates a special table. The kernel eventually jumps to the its 32-bit entry point. This step is out of scope and won't be discussed further.

A.4.2 The structure of MBR

The "master boot record" has only 512 bytes and can't put too much. Its main function is to tell the computer where to find the operating system. It consists of three parts:

1. 1-446 bytes: Call the machine code of the operating system.

2. 447-510 bytes: Partition table (64 bytes).

3. 511-512 bytes: Master boot record signature bytes (`0x55` and `0xAA`).

The code (bytes 1-446) This code searches the Partition Table to find the first partition that is marked as an "active"or boot partition. Among them, the second part of the "partition table" is to divide the hard disk into several areas.

A.4.3 More about the partition table

The partition table is a 64-byte data structure located in the same sector as the Master Boot Record (cylinder 0, head 0, sector 1). The Partition Table conforms to a standard layout that is independent of the operating system. Each Partition Table entry is 16 bytes long, making a maximum of four entries available. Each entry starts at a predetermined offset from the beginning of the sector, as follows:

- Partition 1 – starts at `0x01BE (446)`

- Partition 2 – starts at `0x01CE (462)`

- Partition 3 – starts at `0x01DE (478)`

- Partition 4 – starts at `0x01EE (494)`

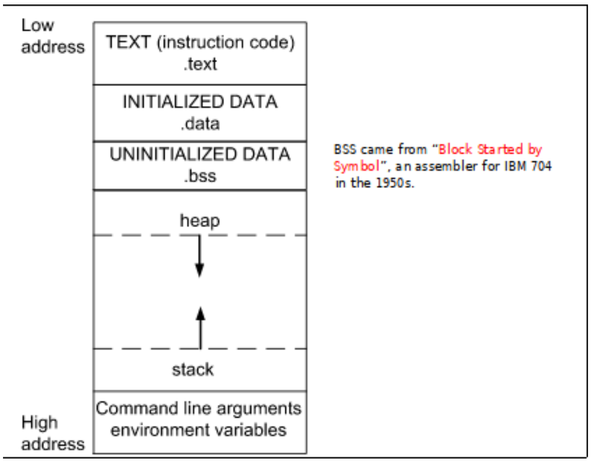
This is why MBR can only support up to four partitions. Recall that the last two bytes in the sector are its signature word and are always `0x55 0xAA`.

A.5 Nasm assembly – some tips

The bootloader code used some Nasm 8086 assembly instructions whose usage may not be too obvious. This appendix explains how to use them.

A.5.1 The structure of a Nasm program

Memory segments (sections) need to be explicitely declared in a Nasm program. At least the text section is required.



BSS came from "Block Started by Symbol", an assembler for IBM 704 in the 1950s.

Fig. 5. Memory segments diagram.

Data segment contains initialised data (variables), BSS segment contains unitialised data (reserved memory) and the text segment contains the code as well as the entry point for the linker, i.e. the label for the main program. This label is defined in the TEXT section and is `_start` if `ld` is used as linker or `main` if `gcc` is used as linker. The code below is borrowed from [4]. In this case the former linker was preferred.

Listing 2: Nasm program skeleton - invokes exit interrupt. (src/empty.nasm).

```
; Skeleton of a Nasm program
; To compile:
; nasm -f elf -g -F dwarf empty.nasm -o empty.o
; To link:
; ld -m elf_i386 empty.o -o empty

SECTION .data

SECTION .text
global  _start

_start:
    mov     ebx, 0      ; return 0 status on exit - 'No Errors'
    mov     eax, 1      ; invoke SYS_EXIT (kernel opcode 1)
    int     80h
```

More examples found here [5].

A.5.2 The $ notation

$ denotes the address of the current instruction, to be specific $ evaluates to the assembly position at the beginning of the line containing the expression.

A.5.3 Uses of $ – string length

$ can be used measures the length of a string when defining the variables in the .data section. Simply subtract from it the address of the last defined variable.

```
SECTION .data
msg     db 'Hello World!', 0Ah
msg_len equ $ - msg
```

equ is equivalent to the define macro in C, so it would just replace `msg_len` with whatever is on the right hand side and db allocates space in memory. The complete “Hello world” program that uses the $ notation is listed below.

Listing 3: Hello world in nasm using the $ for string length. (src/hello.nasm).

```
SECTION .data
msg     db      'Hello World!', 0Ah
msg_len equ     $-msg


SECTION .text
global  _start

_start:

    mov     edx, msg_len      ; number of bytes to write - one for each letter plus 0Ah (line feed character)
    mov     ecx, msg    ; move the memory address of our message string into ecx
    mov     ebx, 1      ; write to the STDOUT file
    mov     eax, 4      ; invoke SYS_WRITE (kernel opcode 4)
    int     80h
```

The latter code exists with segmentation error. To exit properly, the right interrupt needs to be used as listed here [4] but we don’t care about it in this case.

A.5.4 Uses of $ – jump to address

$ can also be used to instruct the EIP (Extended Instruction Pointer register, Extended meaning 32-bit) where to jump to relatively to the current instruction. As $ stores the EIP (address of “here”), doing `jmp $+offset` jumps a number of lines lower in the code and `jmp $-offset` jumps a number of line higher in the code. Doing `jmp $` results in an infinite loop. To sum up, it’s an alternative way of using the `goto` instruction.

However, jumping by using $ isn’t very practical as to jump to a certain location we need to know how many bytes are used for each instruction. Jumping to a label is preferred. For example in Nasm, `mov` instructions take 5 bytes and an `int` takes 2 bytes. This is demonstrated in the code below, which runs an infinite loop that prints a string. `jmp $-22` jumps one `int` (interrupt) instruction and 4 `mov` instructions up, to `mov edx, msg_len`. We jump all the way up before all register assignments as `int` overwrites them.

Listing 4: Some code that performs jump using the $ notation (src/jump.nasm).

```
SECTION .data
msg     db      'in the loop!', 0Ah
msg_len equ     $-msg


SECTION .text
global  _start

_start:
    ; number of bytes to write - one for each letter plus 0Ah (line feed character)
    mov     edx, msg_len
    mov     ecx, msg    ; move the memory address of our message string into ecx
    mov     ebx, 1      ; write to the STDOUT file
    mov     eax, 4      ; invoke SYS_WRITE (kernel opcode 4)
    int     80h         ; print interrupt - it overwrites the registers
    jmp     $-22        ; jump right below _start label - 22 = int + 4*mov
```

A.5.5 The $$ symbol

From [6]; $$ evaluates to the beginning of the current section (.text, .data, etc.) and since $ evaluates the (absolute) address of the "here", `$-$$` calculates how far we are in the current section.

A.5.6 `times` instruction

Again, from [6]; the `times` prefix causes the instruction to be assembled multiple times. For example, to fill 64 zeros in the memory:

```
zerobuf:        times 64 db 0
```

ut `times` is more versatile than that. The argument to `times` is not just a numeric constant, but a numeric expression. It can evaluate mathematical operations including the address of "here" ($). For example, to pad a string with the "." character until it's 32 bytes long, the following can be done.

```
msg     db     'in the loop!', 0Ah
        times 32-$+msg db '.'
msg_len equ    $-msg
```

Note that because 8086 processor model is `little endian`, the last defined bytes will be stored first. So the snippet above will *prepend* "." characters before the string.

The snippet below builds on `jump.asm` and prints prepends dots until the string to be printed is 32 bytes (in this case 32 characters). It continuously prints `...................in the loop!`.

Listing 5: A usage of the `times` instruction; pad a variable up to a length (src/times.nasm).

```
SECTION .data
msg     db      'in the loop!', 0Ah
        times 32-$+msg db '.'
msg_len equ      $-msg


SECTION .text
global  _start

_start:
    ; number of bytes to write - one for each letter plus 0Ah (line feed character)
    mov     edx, msg_len
    mov     ecx, msg    ; move the memory address of our message string into ecx
    mov     ebx, 1      ; write to the STDOUT file
    mov     eax, 4      ; invoke SYS_WRITE (kernel opcode 4)
    int     80h         ; print interrupt - it overwrites the registers
    jmp     $-22        ; jump right below _start label - 22 = int + 4*mov
```

Finally, `times` can be used to repeat simple instructions by making trivial unrolled loops [6], for instance:

```
times 100 movsb
```

A.5.7 Nasm compiling tips

The following commands were used for the `src/*.nasm` files to produce debuggable executable files.

```
nasm -f elf -g -F dwarf jump.nasm # compile
ld -m elf_i386 jump.o -o jump # link
gdb ./jump
./jump
```

References

[1] 0xax, *Linux insides*, http://3zanders.co.uk/2017/10/13/writing-a-bootloader/, [Online; accessed 23-April-2019], 2017.

[2] A. Parker, *Writing a Bootloader Part 1*, http://3zanders.co.uk/2017/10/13/writing-a-bootloader/, [Online; accessed 30-April-2019], 2017.

[3] N. Blunde, *Writing a Simple Operating System — from Scratch*, https://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf, [Online; accessed 30-April-2019], Dec. 2010.

[4] *Learn Assembly Language*, https://asmtutor.com/#lesson2, [Online; accessed 28-April-2019].

[5] *Sample nasm programs*, https://www.csee.umbc.edu/portal/help/nasm/sample.shtml, [Online; accessed 28-April-2019].

[6] N. D. Team, *NASM - The Netwide Assembler*, https://www.nasm.us/xdoc/2.13.01/nasmdoc.pdf/, [Online; accessed 22-April-2019], 2017.