The misadventures of x86

Martin Valencia
University of Arizona
Tucson, AZ 85721
mvalencia1350@email.arizona.edu

Reza Munoz-Asayesh
University of Arizona
Tucson, AZ 85721
rasayesh@email.arizona.edu

April 21, 2020

Abstract

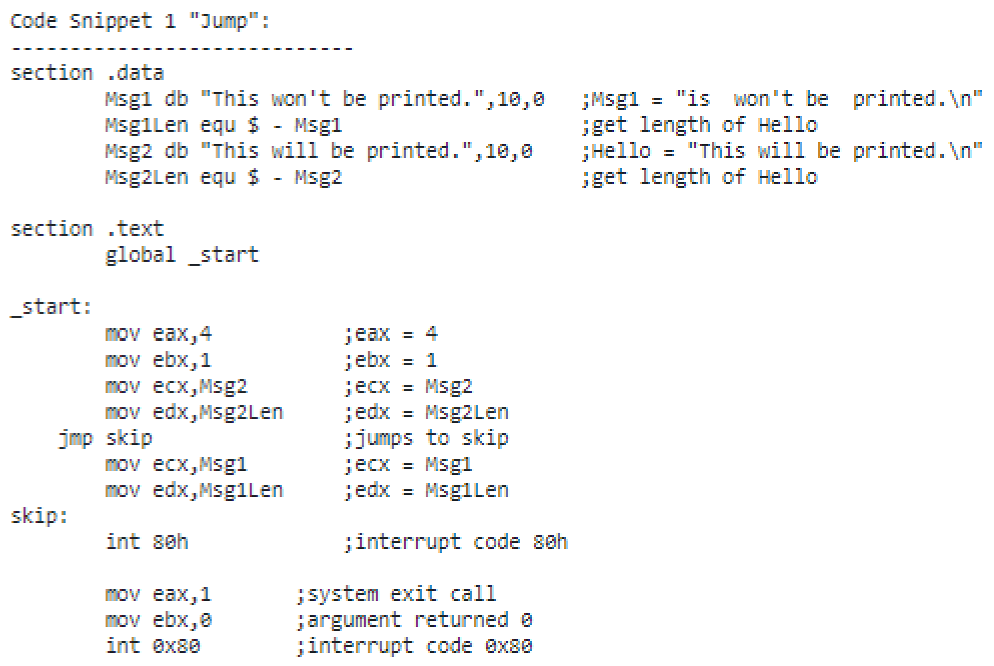
1 Introduction

2 History

In 1978, Intel released a 16-bit microprocessor chip called the Intel 8086. The 8086 project began in May 1976 as successor to the 8080 8-bit microprocessor and temporary solution to the iAPX 432 commercial failure. A new architecture was defined by Stephen P. Morse with the assistance of Bruce Ravenel. This architecture was created to support full 16-bit processing and maintain backwards compatibility with the 8080 and 8085 architectures. Making it possible to automatically convert source code with little to no user editing. In addition, new instructions were included such as microcoded multiply and divide, full support for signed integers, self-repeating operations, and base plus offset addressing. This architecture was named "x86" after Intel's successor microprocessor chips ending with the number 86. Currently x86 is still used for low-level programming and a majority of mobile devices utilize the x86 architecture.[4]

3 Control Structures

When it comes to control structures x86 lets the user implement "jumps" this simple control structure is extremely volatile and allows the user to do both loops and conditional if/else statements. In x86 there are two types of jumps conditional and unconditional. Unconditional jumps consist of just a jump statement with a label address that it will always jump to by default. While conditional jumps will jump to the given address based off of a given condition that will return true or false, if false then the jump will not occur but if the conditional evaluates to true then the jump will occur. In order for the conditional statement to evaluate and run however, a comparison must be done that will return the required true/false flag. This comparison is in the form of CMP which is an Instruction that compares two values, it can take another register as arguments or it can take Integers to compare against each other. When this CMP instruction is computed, the evaluation is placed into the register EFLAGS.
X86's jump instruction is expressed as JMP, such that jump takes in a "label" or address where it will jump to. An example of an unconditional Jump that goes to a given address will be shown here:

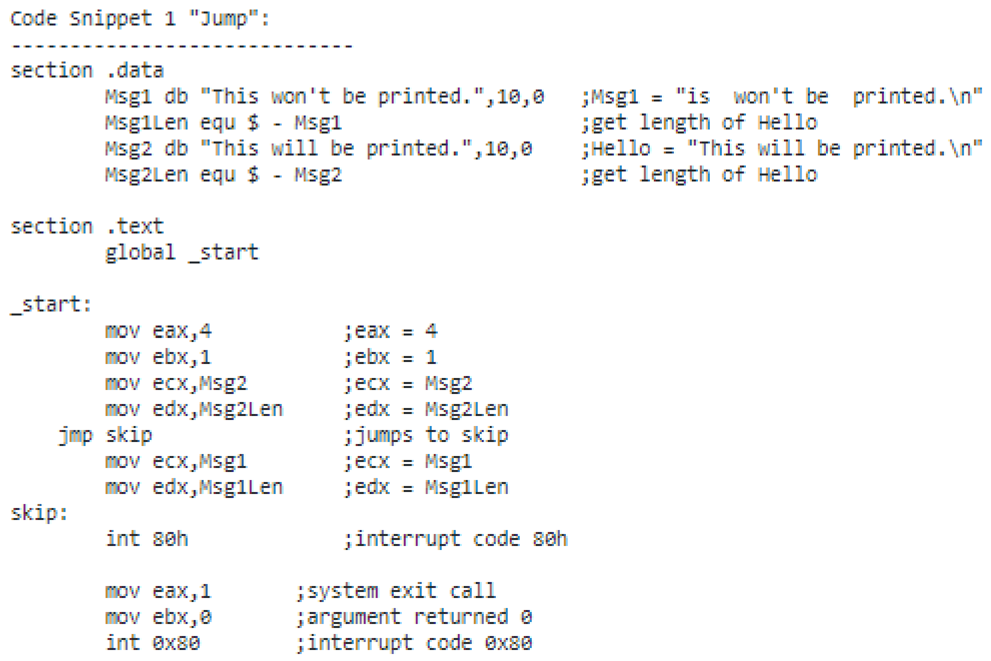


```
Code Snippet 1 "Jump":
-----------------------------
section .data
        Msg1 db "This won't be printed.",10,0   ;Msg1 = "is  won't be  printed.\n"
        Msg1Len equ $ - Msg1                    ;get length of Hello
        Msg2 db "This will be printed.",10,0    ;Hello = "This will be printed.\n"
        Msg2Len equ $ - Msg2                    ;get length of Hello

section .text
        global _start

_start:
        mov eax,4           ;eax = 4
        mov ebx,1           ;ebx = 1
        mov ecx,Msg2        ;ecx = Msg2
        mov edx,Msg2Len     ;edx = Msg2Len
    jmp skip                ;jumps to skip
        mov ecx,Msg1        ;ecx = Msg1
        mov edx,Msg1Len     ;edx = Msg1Len
skip:
        int 80h             ;interrupt code 80h

        mov eax,1       ;system exit call
        mov ebx,0       ;argument returned 0
        int 0x80        ;interrupt code 0x80
```

As for conditional Jumps, in x86 there are a multitude of different conditional comparisons that can be used these will be presented below with a small snippet of code to demonstrate the comparison instruction CMP and the jump instruction that is conditional. Let it be noted that the comparison in CMP compares left argument to the right argument. This list is not exhaustive [2] (Control Flow Instructions):

1. je – Jump if equal
Ex.
cmp eax, 1 ; compare eax to 1
je equalTo1 ; jump to address “equalTo1” if eax is equal to 1
2. jge – Jump if greater than or equal
Ex.
cmp eax, 10 ; compare eax to 10
jge lessThan10 ; jump to “lessThan10” if eax is less than 10
3. jG – Jump if greater than
Ex.
cmp eax, 10 ; compare eax to 10
jG lessThan10 ; jump to “lessThan10” if eax is less than 10
4. jns – Jump if not signed
Ex.
mov ecx, 10 ; ecx = -10
jns skip ; if ecx is not a signed number jump to skip
5. js – Jump if signed
Ex.
mov ecx, -10 ; ecx = -10
js skip ; if ecx is a signed number jump to skip
6. jl – Jump if less than
Ex.
cmp eax, 10 ; compare eax to 10
jl lessthan10 ; jump to “lessthan10 if eax is less than 10
7. jle – Jump if less than or equal
Ex.
cmp eax, 10 ; compare eax to 10
jle lessThan10 ; jump if eax is less than 10
8. jne – Jump if not equal
Ex.
cmp ecx, 100 ; ecx = 10
jne skip ; if ecx != 100 jump to skip

In the scenario that any of these conditional statements is false, the program will simply fall through the code and not execute the jump, thus executing the instruction directly below the jump/cmp instructions. These different instructions that show how to compare two items and execute a specific as a result of the comparison can be related to if/else if/else statements that are in languages like c and java etc. These comparisons will go to the jump if the "if statement" is true and will skip the jump if the comparison is false. In x86 the jump instruction is limited to which address space can be reached, the maximum jump distance from the current address is a 32-bit size, and this applies to both 64- and 32-bit versions of x86 programs.
Rather than have separate control structures for if conditional statements and for loops / while loops, in x86 you can create for and while loops using the conditional jump statements and addresses. An example is shown here in this small program that computes the factorial of a number that is stored in the register ebx.



```
Code Snippet 1 "Jump":
-----------------------------
section .data
        Msg1 db "This won't be printed.",10,0   ;Msg1 = "is  won't be  printed.\n"
        Msg1Len equ $ - Msg1                    ;get length of Hello
        Msg2 db "This will be printed.",10,0    ;Hello = "This will be printed.\n"
        Msg2Len equ $ - Msg2                    ;get length of Hello

section .text
        global _start

_start:
        mov eax,4           ;eax = 4
        mov ebx,1           ;ebx = 1
        mov ecx,Msg2        ;ecx = Msg2
        mov edx,Msg2Len     ;edx = Msg2Len
    jmp skip                ;jumps to skip
        mov ecx,Msg1        ;ecx = Msg1
        mov edx,Msg1Len     ;edx = Msg1Len
skip:
        int 80h             ;interrupt code 80h

        mov eax,1       ;system exit call
        mov ebx,0       ;argument returned 0
        int 0x80        ;interrupt code 0x80
```

This program essentially creates a while loop that will continually break down a number subtracting one from it every loop and multiplying the new difference, with the previous product until the factorial of the originally stored number in ebx is found. This loop will continually repeat itself while ebx is greater than 1, notice the snippet "jg label" which will check to see if ebx is greater than one and will jump to the established address space that is labelled as "label:". Once ebx is 1 then the loop will end, and the jump will not execute thus ending the program and outputting the factorial. This demonstrates how you can make both a while loop to end on a given condition or how you can use a register that stores a specific number to run the loop a specific number of times like a for loop in java or c.

4 Data

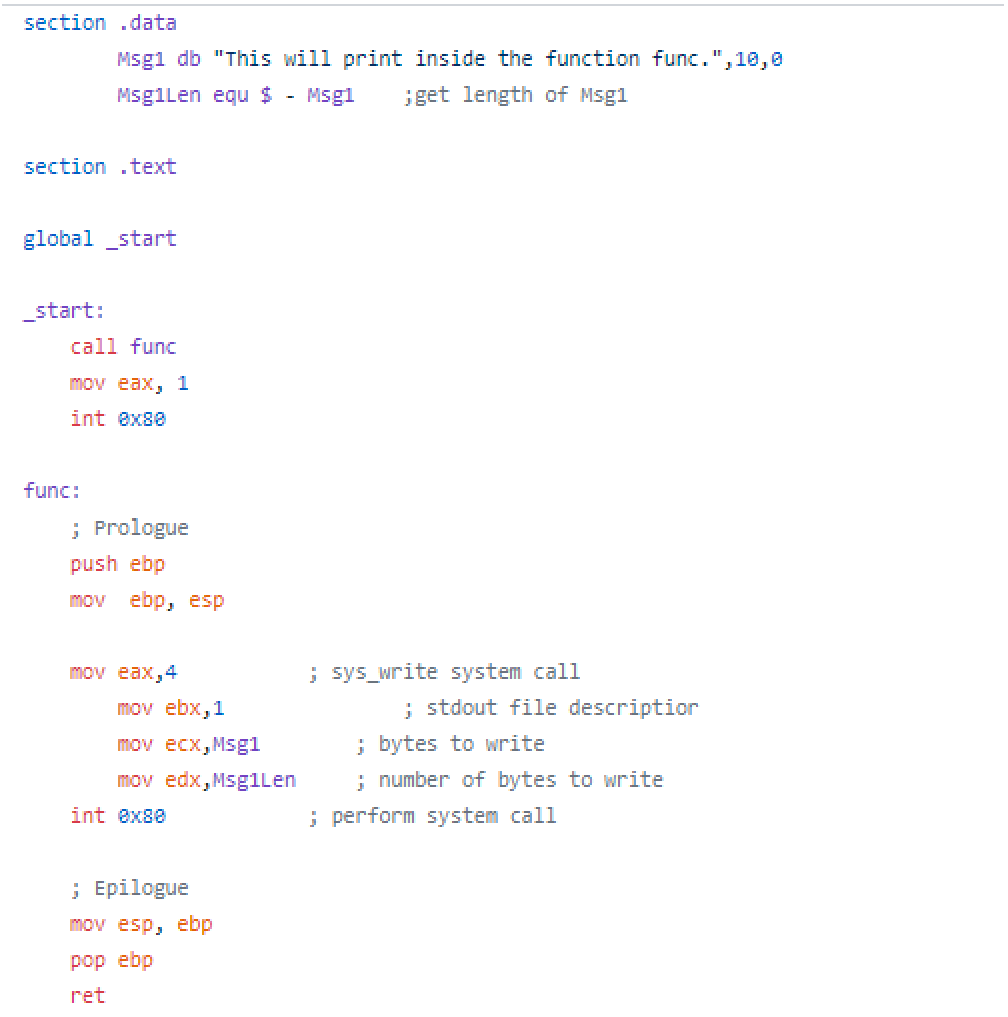
In x86 there are no data types as defined by higher level languages. Instead, the primary characteristic of data is it's size. Since we're focusing on x86 32 bit architecture, the supported data sizes are byte, word, doubleword, quadword, and ten bytes.

Directive	Purpose	Storage
DB	Define Byte	Allocates 1 byte
DW	Define Word	Allocates 2 bytes
DD	Define Doubleword	Allocates 4 bytes
DQ	Define Quadword	Allocates 8 bytes
DT	Define Ten Bytes	Allocates 10 bytes

Nasm provides the ability to denote data with directives. These are commands that are included in the assembler syntax and are not related to the x86 processor instruction set. What a directive does is allocate the amount of storage need for a defined variable.

5 Subprograms

Some major sub-programs of x86 that are useful are the struc and the function, each of these sub programs have their own unique strengths and uses. Starting off with the function in C, functions allow you to organize your code into sections that do specific tasks. These sections called "functions" are established by creating a label where the function starts and ends with a return, that will reset the stack pointer to the place the stack was at after the function was called. In this code here we make a function that accomplishes the task of printing the sentence "This will print inside the function func. " is started when the code uses the instruction "call" and the given value for call is "func" which is the functions name. So, the use of "call" would go "call functionName".



```
section .data
        Msg1 db "This will print inside the function func.",10,0
        Msg1Len equ $ - Msg1    ;get length of Msg1

section .text

global _start

_start:
    call func
    mov eax, 1
    int 0x80

func:
    ; Prologue
    push ebp
    mov  ebp, esp

    mov eax,4           ; sys_write system call
        mov ebx,1               ; stdout file descriptor
        mov ecx,Msg1        ; bytes to write
        mov edx,Msg1Len     ; number of bytes to write
    int 0x80            ; perform system call

    ; Epilogue
    mov esp, ebp
    pop ebp
    ret
```

This function takes no parameters and so does not need to have anything passed in. However, if needed you could pass in variables into the function like so:

1.7.6 Function calls

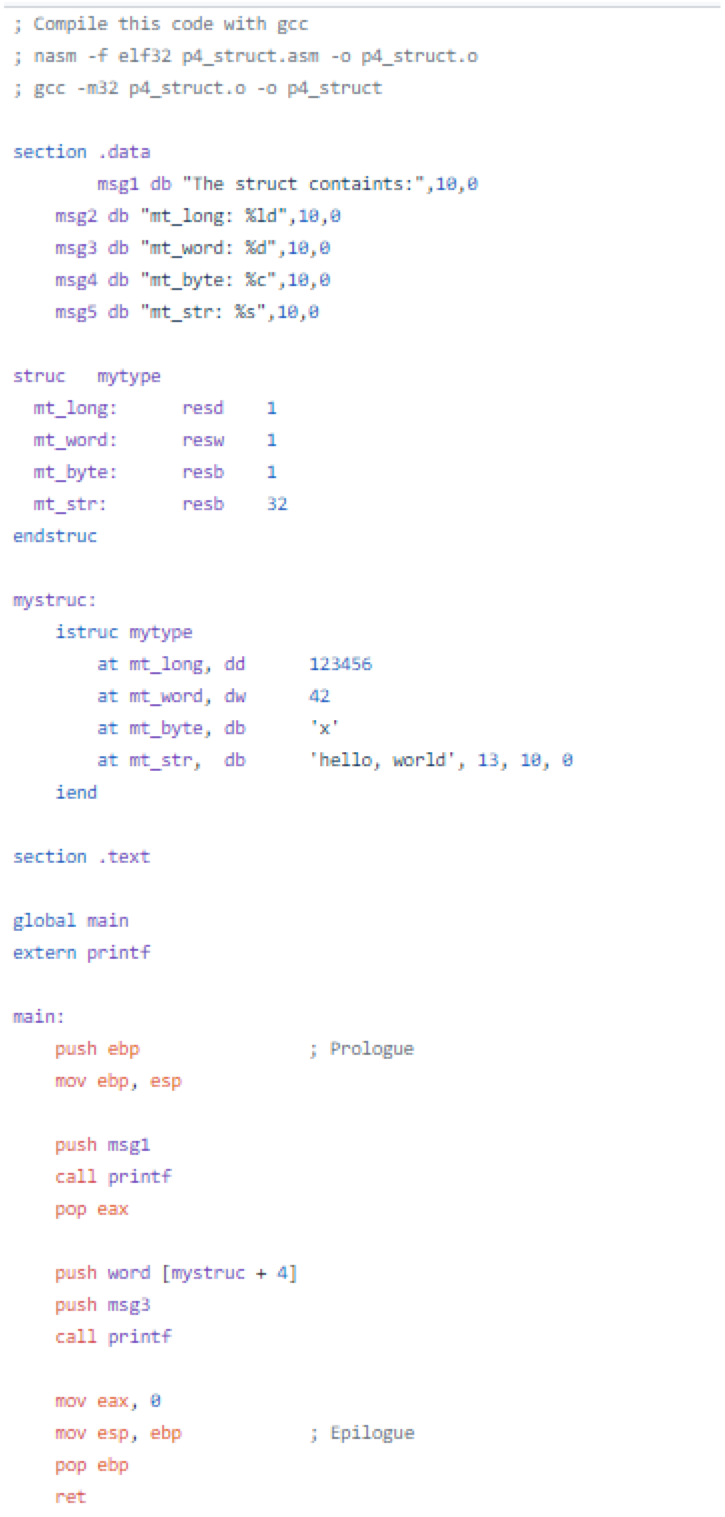
1.7.6.1 C

```
function (int x, char y );
```

1.7.6.2 x86 assembly

```
mov eax, y
push eax
mov eax, x
push eax
call function
```

[1] Credit: sensepost (A Crash Course in x86 Assembly for Reverse Engineers).
Which is the equivalent of:
Function(int x, int y) or Function(int x, char y) etc... depending on what you pass in, The type is determined by what is in the register that is passed into the function although x86's registers could contain different types.
X86 also has a sub-program as mentioned earlier called a strucs, that can be used similar to how C has structs.
This program in which we make a struct in, contains a struct called "mystruc" and this struct has its own "type" that is established by the user as "mytype". Each struct can hold information that the user establishes for instance this struct in the code contains some values including "123456" , "42", 'x', and "hello, world 13, 10 , 0" to include the newline character at the end.



```
; Compile this code with gcc
; nasm -f elf32 p4_struct.asm -o p4_struct.o
; gcc -m32 p4_struct.o -o p4_struct

section .data
        msg1 db "The struct containts:",10,0
    msg2 db "mt_long: %ld",10,0
    msg3 db "mt_word: %d",10,0
    msg4 db "mt_byte: %c",10,0
    msg5 db "mt_str: %s",10,0

struc   mytype
  mt_long:      resd    1
  mt_word:      resw    1
  mt_byte:      resb    1
  mt_str:       resb    32
endstruc

mystruc:
    istruc mytype
        at mt_long, dd      123456
        at mt_word, dw      42
        at mt_byte, db      'x'
        at mt_str,  db      'hello, world', 13, 10, 0
    iend

section .text

global main
extern printf

main:
    push ebp                ; Prologue
    mov ebp, esp

    push msg1
    call printf
    pop eax

    push word [mystruc + 4]
    push msg3
    call printf

    mov eax, 0
    mov esp, ebp            ; Epilogue
    pop ebp
    ret
```

Strucs are accessed like the stack is, using an offset from the address of the strucs will give you a specific item in the strucs for instance when calling mystruc. Unlike C you can not use the dot notation nor the -> notation to access a certain element of a struct you must use the constant name to get the correct offset position, this is why there are two “struc” looking pieces of code, one toward the top contains the same name of the types and their sizes so that you can call the type and get its offset when accessing a specific variable and example is provided here “NASM, since it has no intrinsic structure support, does not support any form of period notation to refer to the elements of a structure once you have one (except the above local-label notation), so code such as mov ax,[mystruc.mt$_{w}ord]isnotvalid.mt_{w}ordisaconstantjustlikeanyotherconstant, sothecorrectsyntaxismovax, [mystruc+ mt_{w}ord]ormovax, [mystruc + mytype.word]$.j[2]

6 Summary

References

[1] A Crash Course in x86 Assembly for Reverse Engineers. `https://sensepost.com/blogstatic/2014/01/SensePost_crash_course_in_x86_assembly-.pdf`. Accessed: 2020-04-20.

[2] The Netwide Assembler: NASM. `https://www.nasm.us/xdoc/2.11.02/html/nasmdoc4.html#section-4.12.10`. Accessed: 2020-04-20.

[3] x86 Assembly Guide. `https://www.cs.virginia.edu/~evans/cs216/guides/x86.html`. Accessed: 2020-04-13.

[4] Leo J Scanlon. *8086/8088/80286 assembly language*. New York, N.Y. : Brady Books : Distributed by Prentice, 1988.