Back-end support for x86 architecture and extensions to the little language

B.Tech Project submitted by

Ayush Gupta

Roll Number 190030007

Research Supervisor

Dr. Nikhil Hegde

॥ सा विद्या या विमुक्तये ॥

भारतीय प्रौद्योगिकी संस्थान धारवाड
Indian Institute of Technology Dharwad

Indian Institute of Technology Dharwad

Computer Science and Engineering

November, 2022

Abstract

In the Compiler's Lab Course, we created a compiler for a little language. That compiler generated dummy assembly instructions. Those dummy instructions were then ran on a Tiny simulator. Our goal is to generate x86 instructions from the compiler of the little language instead of using the Tiny Simulator. Thus, it becomes imperative to read and understand x86 assembly instructions. So, to generate x86 assembly instructions, I first tried to understand the Exocompilation [1]. It was hoped that Exocompilation will help in directly generating the x86 assembly instructions. However, the paper was aimed at increasing the speed of the computation by introducing the hardware optimization in the code itself using the Exocompilation library in Python.

Then I studied the assembly instructions of x86 from the Internet [2]. I tried to map the instructions generated from tiny simulator to what the actual instruction would it be in x86. This is where most of the complexities arose, since there are multiple ways to do the same thing in x86. There was also extra emphasis on the special registers in x86. In order perform any operation which required privileges only accessible by Operating System, multiple default values needed to be passed to special registers. Not only that, depending on the Operating system, these values changed as well. Finally, the most stark difference are the different sections in assembly files, which were not present tiny simulator. There are separate sections for the code and data. Lastly, it is very important to mention the number of bytes a variable is going to use. This needs to be explicitly mentioned in x86. There are other differences in how the arithmetic instructions are performed in x86, which is overly simplified in tiny simulator and the flag registers. The most important part to understand while working in x86 was that there is no concept of datatype in it. It all depends on the number of bytes provided. If inaccurate bytes are mentioned, it causes Segmentation dumped. Thus mapping of instructions are done carefully. In the end, I also needed to use standard library of C to take input of certain datatype. This just shows how closely C programming language is linked with the assembly. I used this project [3] to map the tiny simulators instructions since it provided the most optimal code.

With this BTP, I hope the reader is able to understand and write basic programming in x86 along with using C libraries in it. It will also help in introducing optimizations at hardware level. The desired goal is to generate x86 assembly instructions directly from the compiler of the little language.

Contents

List of Figures

List of Tables

Back-end support for x86 architecture and extensions to the little language

1 Introduction

Compilers is one of the core-subjects for the Computer Science and Engineering. They are extensively used for every task in Software Industry. So, as a Computer Science undergraduate, it becomes imperative to understand the workings of a compiler. Thus, it is not only important to understand the theoretical workings, but practical implementation of the compiler. The final output of a compiler are the assembly instructions. These assembly instructions are then ran on the machine(processor) and these assembly instructions are machine dependent. The pedagogy of Compiler's Project has a Tiny simulator which runs the generated assembly instructions. However, those assembly instructions of Tiny simulator are far simpler and abstract than the actual assembly instructions.

In this BTP, I have tried to understand the complexities of the actual assembly, as well as tried to generate them directly via compiler. I have focused exclusively on x86 assembly of the Intel Processors as they are one of the most widely used processors.

2 Background

In order to understand this project, you need to understand the following terms:

Assembly Language A type of low-level programming language that is intended to communicate directly with a computer's hardware. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans.

The key point to mention is that both, high level languages like C and assembly instructions are meant to be read by humans. However, they are both very different in terms of complexities. And the biggest difference is that assembly instructions are dependent on the architecture of the processor. High-level programming languages like C are independent of the machine. It is the job of the compiler of that high-level language to generate the assembly instructions as needed by that machine.

Thus, these assembly instructions need to be converted into the machine code so that it can be ran on a processor. This is done by an assembler. The '.asm' files are used for writing the x86 assembly instructions. And the assembler used is called NASM.

NASM The Netwide Assembler. A portable x86 assembler. It assembles the file and directs output to the outfile if specified. If outfile is not specified, NASM will derive a default output file name from the name of its input file, usually by appending '.o' or '.obj' (creating an object file), or by removing all extensions for a raw binary file. Failing that, the output file name will be a '.out', creating an executable file. The object file depends on the Operating System used. Thus, from the same assembly code, the object file for LINUX or MacOS can be generated.

Object File A file with '.o' or '.obj' as extension is a file generated after compiling the source code. It is the machine code but without the order of execution, meaning that the processor will not know from where to start the execution of instructions.

Executable File A file with '.out' as extension is a file generated after linking a set of object files together using a linker. The processor can directly execute these files and they also know the starting point of the execution.

Linker A computer program that takes one or more object files generated by a compiler and combines them into one, executable program. In this BTP, we have used the in-build linker of gcc (GNU Compiler Collection) as well as ld linker. Depending upon the linker, the global label, which tells the starting point of execution to the linker also changes.

3 Objective

Our goal is to generate x86 instructions from the compiler of the little language instead of using the Tiny Simulator. There should be one-to-one mapping of the instructions for the Tiny Simulator to the x86 assembly instructions.

4 Methodology

4.1 Exo Compilation Paper

To add back-end support for the x86 in the little language, We could introduce the x86 assembly code in the little language itself. So, Exo Compilation paper [1] was studied.

4.1.1 Objective of the Paper

In order to increase the speed of computation for programs requiring high computation, we need to introduce hardware level optimizations in the code. This is usually achieved in C language with assembly code sprinkled in it where different hardware having different sets of instructions. Thus, for a performance engineer who wants to improve efficiency, he/she will need to study different architecture (like x86, ARM, etc) and write optimized code. This is not only time consuming, but difficult to scale to other platforms.

4.1.2 Main Idea

So, to deal with these problems, this paper introduces a new approach, called Exo Compilation where we write optimized code on user level only, i.e. optimizations are introduced while we are writing the code. They have embedded this 'Exo' language in python which then generates the C language code and then it runs to perform on machines with higher efficiency. Whenever we need to optimize code for a new hardware, a library is written in exo instead of manual code generation in C, which is more difficult to scale.

4.1.3 Implementation

The Exo language allows programmers to write custom memories, add instructions as well as configure states in the program. Rather than directly writing code that uses hardware libraries, Exo transforms a simple program into an equivalent but more complex and high performance version targeted for a specific hardware accelerator.

4.1.4 Conclusion

As far as the results go, the Exo didn't provide as good optimization as the other libraries like MKL(for Intel processors specifically) and OpenBLAS(open source library for different architecture) but the difference wasn't that big. The simplicity and uniformity provided by Exo and further improvements in it would definitely produce better results.

However, our goal of incorporating x86 assembly instructions with the Exo language is not possible. Thus, there is need to manually map the instructions of Tiny Simulator to x86 assembly instructions.

4.2 NASM Tutorial

In order to do one-to-one mapping, we to understand the syntax of x86 assembly instructions.

4.2.1 Setup in LINUX Environment

In order to run the assembly files (with extension .asm), we need to install NASM. Run the following commands in order:

1. `wget https://www.nasm.us/pub/nasm/releasebuilds/2.14.02/nasm-2.14.02.tar.gz`
2. `tar xvfz nasm-2.14.02.tar.gz`
3. `cd nasm-2.14.02`
4. `./configure --prefix=/usr/local/nasm/2_14_02`
5. `make`
6. `make install`

The path of the prefix may need to changed. If any problem arises, refer to this blog [4]

4.2.2 Memory Segments in NASM

Data Segment It is represented by .data section and the .bss. The .data section is used to declare the memory region, where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

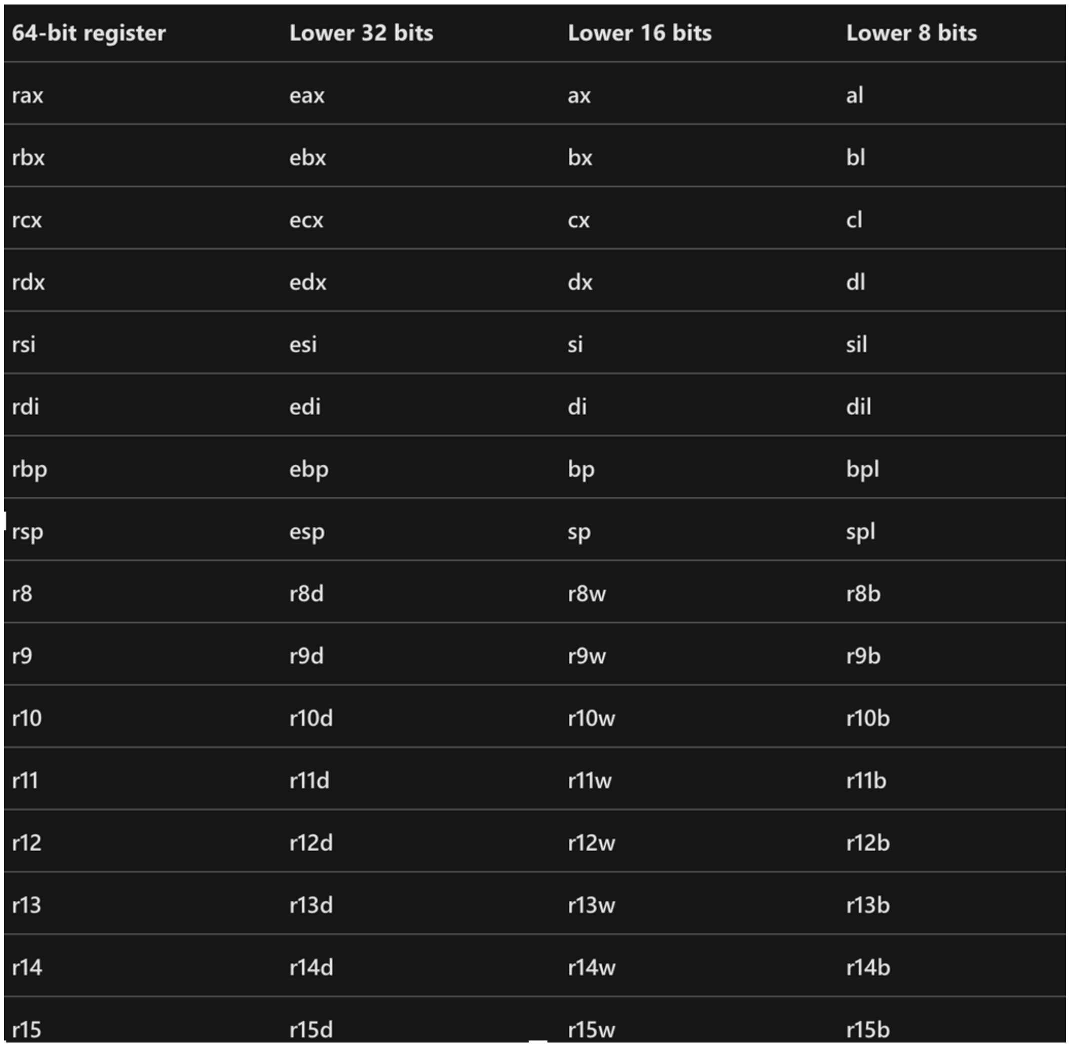
The .bss section is also a static memory section that contains buffers for data to be declared later in the program. This buffer memory is zero-filled.

Code Segment It is represented by .text section. This defines an area in memory that stores the instruction codes. This is also a fixed area.

Stack This segment contains data values passed to functions and procedures within the program.

4.2.3 Registers

Since x86 is an old architecture from the time when processors had 8-bit, there are different register names for 8-bit register. Similarly, we have 16-bit registers, 32-bit register and finally 64-bit registers. Since modern processors use 64 bits, we will use only 64 bit registers. 32 bit registers maybe required in some situations.



64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Figure 4.1: Registers

Among these, r8-r15 are the general purpose registers. rax, rbx, rcx and rdx are special registers whose values are checked during system call. rsi is the register source index and rdi is the register destination index. They are used in function calling. rbp is the base pointer, which points to the base of the current stack frame, and rsp is the stack pointer, which points to the top of the current stack frame. rbp always has a higher value than rsp because the stack starts at a high memory address and grows downwards. Apart from these, there is rip, register for the Instruction Pointer.

The following flag registers are used in Conditional Statements in NASM:

Overflow Flag (OF) It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.

Direction Flag (DF) It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.

Interrupt Flag (IF) It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.

Trap Flag (TF) It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

Sign Flag (SF) It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.

Zero Flag (ZF) It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.

Auxiliary Carry Flag (AF) It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.

Parity Flag (PF) It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.

Carry Flag (CF) It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

4.2.4 mov Instruction and Addressing Mode

Immediate: An immediate value (or simply an immediate or imm) is a piece of data that is stored as part of the instruction itself instead of being in a memory location or a register.

The syntax for mov instruction is: `mov destination, source`

where it may take one of the five following forms:

1. `mov register, register`
2. `mov register, immediate`
3. `mov memory, immediate`

4. `mov register, memory`
5. `mov memory, register`

There are three modes of addressing:

1. **Direct Memory Addressing**: When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value. This offset value is also called effective address.

   In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

   In direct memory addressing, one of the operands refers to a memory location and the other operand references a register. For example, `mov rbx, word_value`

2. **Direct Offset Addressing**: This addressing mode uses the arithmetic operators to modify an address. For example, `word_table dw 134, 345, 564, 123`

   We can access the 3rd with: `mov rcx, word_table[2]`

3. **Indirect Memory Addressing**: This addressing mode utilizes the computer's ability of Segment:Offset addressing. Generally, the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

```
my_table 10 dw 0   ; Allocates 10 words (2 bytes) each initialized to 0
mov rbx, [my_table]      ; Effective Address of my_table in rbx
mov [rbx], 110           ; my_table[0] = 110
add rbx, 2               ; rbx = rbx +2
mov [rbx], 123           ; my_table[1] = 123
```

Type Specifier	Byte Address
Byte	1
Word	2
DWord	4
QWord	8
TByte	10

Table 4.1: Type Specifier

When writing immediate directly in memory, it becomes important to specify the type specifier. For example:

`mov [rbx], 110`

it is unclear whether 110 is a byte or a word. Better way would be:

`mov [rbx], word 110`

4.2.5 Variables

Initialized Variables in NASM are mentioned in 'data' section. For example:

`choice db 'y'`

Directive	Purpose	Storage Space
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

Table 4.2: Directives

The reserve directives are used for reserving space for uninitialized data. This is written in bss section. bss stands for Block Starting Symbol. For example:

`num1 resb 2`

The 'times' directive allocates an array.

Directive	Purpose
resb	Reserve a Byte
resw	Reserve a Word
resd	Reserve a Doubleword
resq	Reserve a Quadword
rest	Reserve ten bytes

Table 4.3: Reserve Directives

4.2.6 Arithemetic and Logical Instructions

1. **inc**: increments the value

2. **dec**: decrements the value

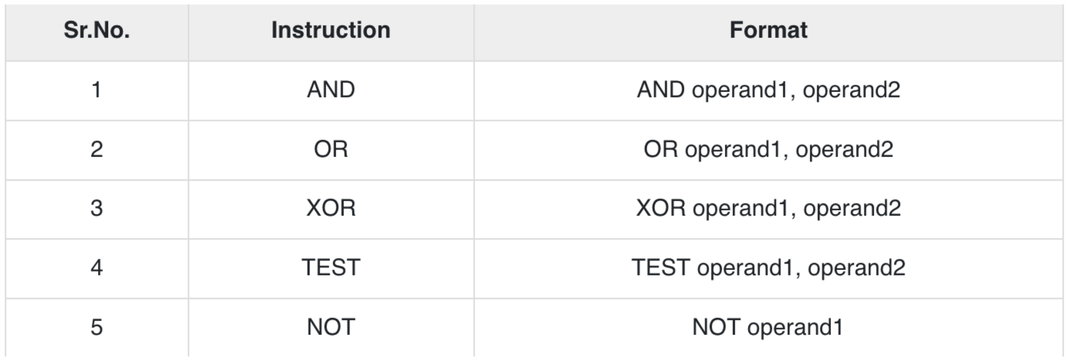
3. **add**: for performing addition. The value from the source is added to the value in the destination

4. **sub**: for performing substration. The value in the source is subtracted from the value in the destination.

5. **mul/imul**: There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag. For more details, refer the page [5]

6. **div/idiv**: The division operation generates two elements - a quotient and a remainder. In case of multiplication, overflow does not occur because double-length registers are used to keep the product. However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs. The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data. For more details, refer the page [5]

Apart from these, we have the following logical instructions:



Sr.No.	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

Figure 4.2: Logical Instructions

4.2.7 Conditions and Loops

Refer Image 4.3 for all the details conditional jumps in NASM. Loops are implemented using lables and conditional jump statements. We define a label in the text section with a simple name. If you are not using the gcc linker, then the global label is normally named '_start' (or 'start' in case of mac), else the label is 'main'. We define the entry point of execution with the global keyword.

4.2.8 Other important instructions

'int' instruction invokes the interrupt, that means, the operating system executes in privileged mode. 'push' and 'pop' are used as same as they were used in out little language. They push the registers onto the stack when there is a function call.

5 Sample Programs

Before running the programs, check these commands:

1. `nasm -felf64 filename.asm`
2. `gcc -o filename filename.o` or `ld -o filename filename.o`
3. `./filename`

First command creates the object file, second command linkes the object file to create the executable. Third command runs the executable. Please use ld command when the global label is _start and gcc when the global label is main.

5.1 Reading and Writing input strings

To write string, the rax needs to have value 4, rbx needs to have value 1, rcx needs to have the variable name and rdx needs to have the length of the message. Then invoke the OS.

To read string, the rax needs to have value 3, rbx needs to have value 2, rcx needs to have the variable name and rdx needs to have the length of the input. Then invoke the OS.

System call be done with 'int' command with 80h as a default value.

To exit a program safely, the rax needs to have value 1 and rbx needs to have value 0.

$ is used to mark the current bytes when writing data.

```
section .data                           ;Data segment
   userMsg db 'Please enter a number: ' ;Ask the user to enter a number
   lenUserMsg equ $-userMsg             ;The length of the message
   dispMsg db 'You have entered: '
   lenDispMsg equ $-dispMsg

section .bss            ;Uninitialized data
   num resb 5

section .text           ;Code Segment
   global _start

_start:                 ;User prompt
   mov rax, 4
   mov rbx, 1
   mov rcx, userMsg
   mov rdx, lenUserMsg
   int 80h

   ;Read and store the user input
   mov rax, 3
   mov rbx, 2
   mov rcx, num
   mov rdx, 5           ;5 bytes (numeric, 1 for sign) of that information
   int 80h

   ;Output the message 'The entered number is: '
   mov rax, 4
   mov rbx, 1
   mov rcx, dispMsg
   mov rdx, lenDispMsg
   int 80h

   ;Output the number entered
   mov rax, 4
```

```
    mov rbx, 1
    mov rcx, num
    mov rdx, 5
    int 80h

    ; Exit code
    mov rax, 1
    mov rbx, 0
    int 80h
```

The most important inference is that all inputs are taken in ASCII format. When the number is stored in variable num, it stored as string of ASCII characters.

5.2 Addition of Numbers given in Program

Whenever printing and reading is involved, it is done in string format. However, converting a string to an integer is complicated in NASM. So, we use printf function of C for printing a number. This shows the power of C language as it can used directly in the assembly code and is used in high performance computing.

```
global main
extern printf

section .text

main:
mov r8, 102 ;Random Value 1
mov r9, 432 ;Random Value 2

add r8, r9

mov rdi, format
mov rsi, r8
mov rax, 0

call printf

;exit code
mov rax, 1
mov rbx, 0
int 80h
```

```
format:
db "%ld", 10
```

Note that before calling the printf function, rdi register needed the label for the 'format' and rsi needed the value of the register to be printed. rax needs to have 0. Also, we need to mention the function beforehand as extern before the coding starts.

The format takes in the type specifier in NASM, then the type specifier in C and finally the maximum length of the message.

5.3 Loops in NASM

The below program will print the following output:

```
*
**
***
****
*****
******
*******
********
```

```
global    main
section    .text
main:
     mov       r10, output      ; rdx holds address of next byte to write
     mov       r8, 1            ; initial line length
     mov       r9, 0            ; number of stars written on line so far
line:
     mov       [r10], byte '*'  ; write single star
     inc       r10              ; advance pointer to next cell to write
     inc       r9               ; "count" number so far on line
     cmp       r9, r8           ; did we reach the number of stars for this line?
     jne       line             ; not yet, keep writing on this line
lineDone:
     mov       [r10], byte 10   ; write a new line char
     inc       r10              ; and move pointer to where next char goes
     inc       r8               ; next line will be one char longer
     mov       r9, 0            ; reset count of stars written on this line
     cmp       r8, maxlines     ; wait, did we already finish the last line?
```

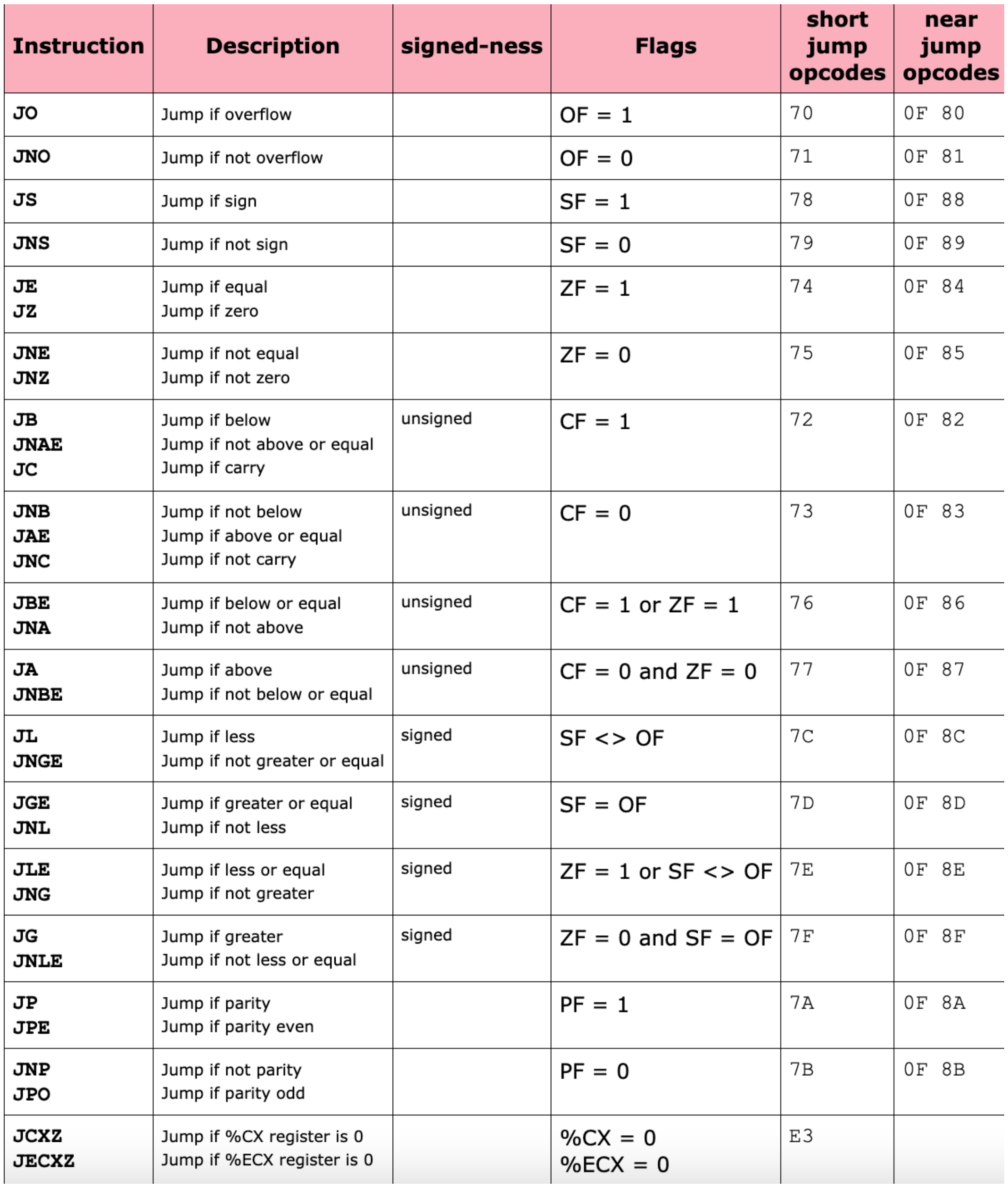
```
    jng        line              ; if not, begin writing this line
done:
    mov        rax, 4            ; system call for write
    mov        rbx, 1            ; file handle 1 is stdout
    mov        rcx, output       ; address of string to output
    mov        rdx, dataSize     ; number of bytes
    int 80h                      ; invoke operating system to do the write
    mov        rax, 1            ; system call for exit
    mov        rbx, 0            ; exit code 0
    int 80h                      ; invoke operating system to exit

section    .bss
maxlines   equ        8
dataSize   equ        44
output:    resb       dataSize
```

As you can see, the looping is same as we did in normally.

6 Challenges and Future plan of Work

The biggest challenge is converting the input number to appropriate datatype (integer or float). This will require using scanf as we used printf. After this, in order to map the instructions for the little language as in project [3]. We will first identify the global variable and write them appropriately in bss and data section. We need to declare the functions used before using extern. After this, we can map read and write instruction to corresponding multiple instructions in NASM.



Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE **JZ**	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE **JNZ**	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB **JNAE** **JC**	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB **JAE** **JNC**	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE **JNA**	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA **JNBE**	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87
JL **JNGE**	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE **JNL**	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D
JLE **JNG**	Jump if less or equal Jump if not greater	signed	ZF = 1 or SF <> OF	7E	0F 8E
JG **JNLE**	Jump if greater Jump if not less or equal	signed	ZF = 0 and SF = OF	7F	0F 8F
JP **JPE**	Jump if parity Jump if parity even		PF = 1	7A	0F 8A
JNP **JPO**	Jump if not parity Jump if parity odd		PF = 0	7B	0F 8B
JCXZ **JECXZ**	Jump if %CX register is 0 Jump if %ECX register is 0		%CX = 0 %ECX = 0	E3	

Figure 4.3: Conditional Jumps

References

[1] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, “Exocompilation for productive programming of hardware accelerators,” in *PLDI*, 2022. [Online]. Available: https://doi.org/10.1145/3519939.3523446

[2] “Nasm tutorial.” [Online]. Available: https://cs.lmu.edu/~ray/notes/nasmtutorial/

[3] M. Sameer, “Compilers lab project,” 2022. [Online]. Available: https://github.com/sameermuhd/CS316-Compilers-Lab

[4] K. Sato, “Install nasm from source.” [Online]. Available: https://noknow.info/it/os/install_nasm_from_source?lang=en

[5] “Assembly arithemetic instructions.” [Online]. Available: https://www.tutorialspoint.com/assembly_programming/assembly_arithmetic_instructions.htm