Contents

1 Introduction

Every computer has a CPU which is responsible for the execution of instructions. The available instructions are defined by the CPU's architecture. This guide assumes that you are using the x86-64 architecture for an Intel CPU on a Linux operating system. Unfortunately that means if you use MacOS or Windows, this guide will only work if you use a virtual Linux machine. If your CPU is ARM-based then virtualisation will not help and you will need to get a different computer.

1.1 YASM structure

YASM is a complete rewrite of the NASM under the new BSD License, it is designed to understand multiple syntaxes although it currently only supports GAS and NASM. This guide will mostly refer to NASM syntax, thus you should do the same when researching topics discussed. Your codebase will consist of directives such as **global** which specifies

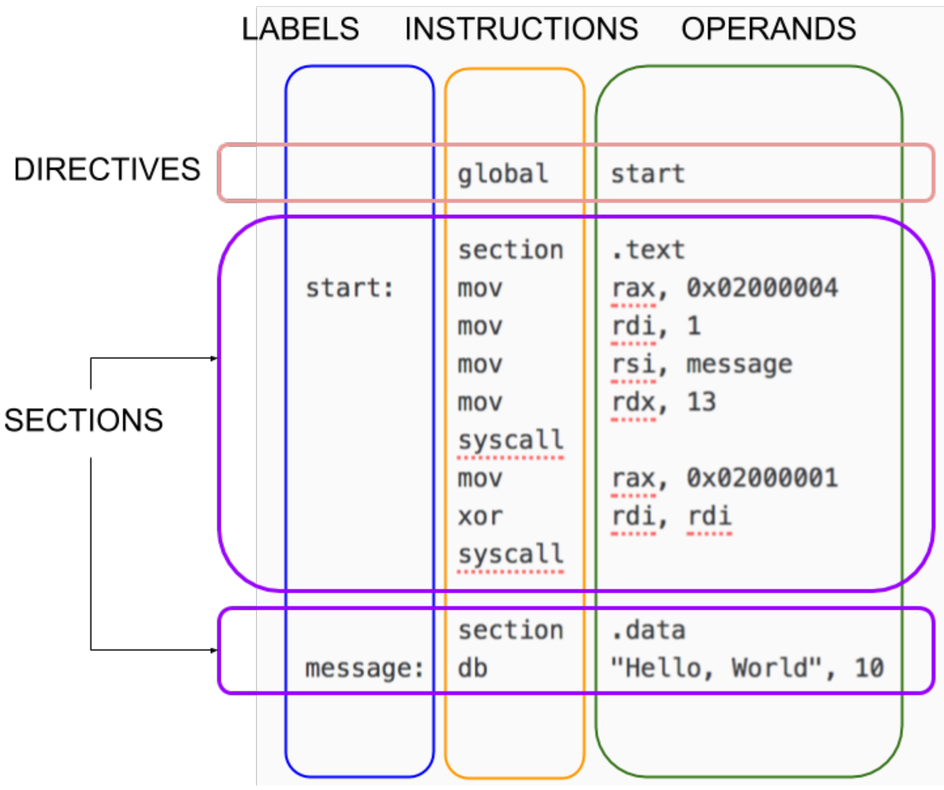

Figure 1: NASM Structure

which labels are globally accessible to external programs. **Figure 1** defines **start** as **global** because it is the entry point to this pure assembly program. In the event you want a high-level programming language like C to use your assembly function, you would set the name of the function/label as **global** instead of **start**. Next you get sections, which includes all the labels and their instructions. However, each section has a specific name and purpose. The **.data** segment is where statically initialised data is stored, while **.bss** is for dynamically allocated data. The **.text** section is where your "code" is stored. For now, let us take a look at the simplest program you could possibly write in assembly.

1.2 A Simple Program

```
  global    start
  section   .text

start:
  mov rax, 60   ; System call for exit
  mov rdi, 0    ; Exit code 0
  syscall       ; Invoke system call
```

This program will simply exit with a status code 0 upon being run. Every pure assembly program will need to build upon this structure. Let's extend it by adding some "variables" to the program.

```
_start:
  mov rax, sys_exit   ; System call for exit
  mov rdi, 0          ; Exit code 0
  syscall             ; Invoke system call

  section   .data
sys_exit db 60
```

Now our program is retrieving the byte (as defined by the pseudo-instruction (**db**)) stored at label **sys_exit** and using it in the system call. However, there is a problem now. The value for a system call never changes but sys_exit's value can be altered. Let's introduce another pseudo-instruction, **equ** to make the value immutable (constant).

```
    section   .data
  sys_exit equ 60
```

What if we want to output the value of sys_exit to the console? We would need to learn about the syscalls for I/O.

1.3 I/O System calls

The operating system provides a set of system calls for I/O. The two you will need to familiarise yourself with are
write.

```
  mov rax, 1                ; Perform write
  mov rdi, 1                ; Destination is stdout
  mov rsi, "Hello world!"   ; Data to write
  mov rdx, 12               ; Byte count to write
  syscall                   ; Perform the system call
```

and **read**

```
  mov rax, 0                ; Perform read
  mov rdi, 0                ; Destination is stdin
  mov rsi, input            ; Buffer to store input
  mov rdx, 11               ; Byte count to read
  syscall                   ; Perform the system call
```

Where **input** is a pointer to a buffer/named memory location that will hold the input of 11 bytes/characters. There are a few important things to note about taking input from the terminal:

- The input will be encoded in ASCII, thus you will need to convert it before performing arithmetic.
- When pressing enter, the newline character is automatically appended to the input.

This makes taking user input a little more complicated. I recommend making your own helper functions to accommodate these issues

1.4 Simple input helper

The first helper function will handle the additional newline character on pressing Enter.

```
input:
; Sets the registers to perform a stdin read
  mov rax, 0
  mov rdi, 0
  syscall

; Takes care of the additional null character that is added
; to the end of the input by pressing Enter in the terminal
  mov rax, 0
  mov rdi, 0
  mov rdx, 1
  mov rsi, trash
  syscall
  ret
```

Now whenever you want to take input from a user you can just use:

```
  mov rdx, size
  mov rsi, input
  call input
```

Where **trash**, **input**, and **size** are all pointers to memory locations defined in the data/bss section. In order to deal with encoding issue, you can simply convert ASCII to Integer by subtraction:

```
  mov al, '2'    ; al = 50
  sub al, '0'    ; al = 50 - 48 = 2
```

Hopefully you noticed that this solution only works with one digit values. Thus you would need to iterate over your entire input string and convert each character to an integer then add them together. However, that is out of the scope for this handbook.

2 Memory

This section deals with declaring, initialising and accessing memory that will be used in your assembly program.

```
segment .data
  var         db "hello"
  var_size    equ $-var
```

The snippet above allows you to define static data. The labels are named memory locations, they have no type, and only know that they are contiguous blocks of bytes. The label var_size is a constant that stores the size of the label before it, this works because $ represents the memory address of the assembler and **var** points to the start of the previous memory location. Therefore by subtracting them from each other, we get the length of the data stored in the previous memory location. When accessing **var** with the **mov** instruction, it is important to remember the label is like a pointer variable and needs to be dereferenced to access its value.

```
mov rax, var             ; Load the address of var into rax
mov rax, [var]           ; Load the value of var into rax
mov rax, [var + 1]       ; Offset var by 1 byte
mov rax, [var + rcx*2]   ; Offset var by 2*rcx bytes
```

Below we can see why you should not think of .data labels as variables.

```
segment .data
  array1        db "hello"
  array2        db 'h','e','l','l','o'
  array3        db 104,101,108,108,111
```

All of the above labels are *like* arrays that can be indexed. Although it might appear that array1 is a string, array2 is an array of characters, and array3 is an array of integers, that is not the case. They are all 3 blocks of bytes. Thus, the following are all equivalent:

```
mov rax, [array1 + 2] ; rax = 108 (letter 'l')
mov rax, [array2 + 2] ; rax = 108 (letter 'l')
mov rax, [array3 + 2] ; rax = 108 (letter 'l')
```

You might be surprised to see that rax would store the value 108 even though some have chars instead of integer values but that is because letters are stored in ASCII.

```
mov rax, [array1 + 6] ; rax = 101 (letter 'e')
```

The above might appear to be a mistake but everything declared in the data segment is one contiguous memory block. As long as you have memory allocated, you can access and alter it (even by mistake). array1 does not have a 6th index, thus it moved on to array2's data and found its 2nd index. This is because the syntax [array1 + 6] is the same as saying: Starting from label array1's first byte, count 6 more bytes and access that value.

index	0	1	2	3	4	5	6	7	8	9
value	h	e	l	l	o	h	e	l	l	o

Table 1: A logical interpretation of the data segment

This also means that you can update specific indexes of memory locations:

```
mov al, 'a'
mov [array1], al     ; array1 = "aello"
mov [array1 + 1], al ; array1 = "aallo"
mov [array1 + 2], al ; array1 = "aaalo"
mov [array1 + 3], al ; array1 = "aaaao"
mov [array1 + 4], al ; array1 = "aaaaa"
```

3 Functions

Writing reusable code is a fundamental concept regardless of the language you are using. However, assembly does not have "functions". A function is just a label with no specific rules attached. Thus it is advised to follow the standards to ensure your functions work as expected. Assembly has the instruction **call** which is just syntactical sugar for pushing the address of the next instruction onto the stack and then jumping to the label. Then at the end of the label, using the **ret** instruction to return to the pushed instruction.

```
_start:
  call func
  ...
func:
  ret
```

Below you will find the calling conventions for assembly functions.

For integer/pointer parameters (64-bit)									
Parameter	0	1	2	3	4	5	6	...	nth parameter
Location	rdi	rsi	rdx	rcx	r8	r9	stack + n bytes	...	stack + 0

Which is similar to but slightly different from floating point parameters which use the 128-bit registers. Take note the order in which stack parameters are pushed - It is always

For floating point parameters (128-bit)						
Parameter	0	...	7	8	...	n
Location	xmm0	...	xmm7	stack + n bytes	...	stack + 0

right to left. It is the duty of the *caller* to push and remove the parameters from the stack. Once the function is in control, the return address will be located at [**rsp**] (where rsp is the stack pointer) with the first stack parameter being located at [**rsp + 8**], etc. Another convention is that the stack pointer **rsp** must be aligned to a 16-byte boundary before making a call. However, making the call pushes an 8-byte address to the stack which breaks the alignment. Thus an offset must be subtracted from the stack pointer. It is also a convention to keep local variables on the stack at a 16-byte boundary. This creates **stack frames** which are used by GDB to trace backwards through the stack to inspect calls made.

```
push rbp
mov rbp, rsp
sub rsp, 16
...
mov rsp, rbp
pop rbp
ret
```

The above should be what every function looks like in order to conform to the calling conventions.

4 Running (and Debugging)

Now that we have covered some basic topics on how to write assembler code, let's look at how to run (and debug) your program.

```
yasm -f elf64 -g dwarf2 -l main.lst main.asm
```

The above command selects a 64-bit ELF binary with DWARF2 debugging information. It also generates a listing file called **main.lst** which can be used to debug your program. The **yasm** command generates an object file named main.o which is ready to be linked with other libraries or object files. If you are writing pure assembly with **_start** as your entry point, you need to link the program using:

```
ld -o main main.o
```

Which creates an executable file called **main** that can be run using:

```
./main
```

There are a lot of different commands to remember and they can be cumbersome to run repeatedly. Thus, since we are using Linux, I recommend taking advantage of makefiles and the bash scripting language.

4.1 Setup structure

A basic development environment for writing assembler should consist of at least a main assembly file, a makefile and a bash script.

```
environment
  - main.asm
  - makefile
  - run.sh
```

Since you will determine what goes in the main.asm, we will cover the other two files for now.

```
# makefile

  main: main.o
    ld -o main main.o
```

```
main.o: main.asm
  yasm -f elf64 -g dwarf2 -l main.lst main.asm

clean:
  rm -f main main.o main.lst

run:
  ./main
```

With the makefile inside your project directory, you can run the following commands in your terminal:

```
make clean    # remove all generated files
make          # assembles and links main
make run      # runs main
```

The above order is the recommended order because you should always clean the directory before creating new files. However, this still requires you to enter multiple commands. It also requires you to manually input values every time which becomes time consuming with many different test cases. Thus, it is recommended to use the bash script to automate the process:

```
# run.sh

  #! /bin/bash
  # redirect all the compile output to stderr and
  # stop if the compile step fails
  make clean 1>&2 > /dev/null
  make 1>&2 > /dev/null
  compiled=$?

  # if the compile step failed, exit with error message
  if [[ $compiled -ne 0 ]]; then
      echo "COMPILE FAILED"
      exit $compiled
  fi

  # execute the compiled binary and
  # redirect all output to stdout
  ./main 2>&1
```

Then you can simply run **./run.sh** in your terminal and it will perform all the steps for you. But what should you do if your program requires input from the user? You can alter the final line in **run.sh** to include the **echo** command.

```
echo "input_1
input_2
...
input_n" | ./main 2>&1
```

This will print the input to the terminal and then pipe it to the program. Each input must be on a new line because that will allow your program to interpret them as separate instances of input. For example, let's assume your program prompts the user for their name and then has another prompt for their age:

```
echo "John Smith
24" | ./main 2>&1
```

The great part about this is that it enables you to test multiple scenarios easily. You can repeat the execution instruction with different data to test your program.

```
#!/bin/bash

...code omitted for brevity...

echo "John Smith
24" | ./main 2>&1

echo "Jane Smith
19" | ./main 2>&1

echo "ajdfgjoifj48932nnjgf
fh3298tr" | ./main 2>&1
```

Where the last entry would cause unexpected results in the output of your program, possibly causing an error if not handled properly.

4.2 GDB

Welcome to your best friend for the rest of COS284. If you were able to pass the rest of your programming modules without a debugger then congratulations because you are in for a surprise. If you despised pointers in C++ then welcome to Assembler, where everything is a pointer and a number and garbage simultaneously. If you do not have gdb installed then run:

```
sudo apt-get install gdb
```

Now you can debug your program using:

```
gdb ./main
```

The first thing you want to do is add a breakpoint. This is a line of code that you want to stop at. You can add breakpoints by typing:

```
(gdb)b <line number>
```

You may add as many breakpoints as you need.

```
(gdb)r
```

This will run your program and stop at the first breakpoint you have added. Now you have a few options, you can either run the next instruction, continue running until your next breakpoint or inspect the state of your program:

```
(gdb)n  # next instruction
(gdb)c  # continue until next breakpoint
```

To examine data in your program, you can print the value of memory addresses:

```
section .data
output: db "Hello"
...
(gdb)p (char[5]) output
"Hello"

(gdb)p/x (char[5]) output
{0x59, 0x6f, 0x75, 0x20, 0x68}

(gdb) p (int) output
544567129
```

But be careful with what format you tell GDB to expect, some are more useful than others. Also take note that this will be the value of **output** at the line you have currently landed on. Registers can also be inspected, by default I recommend using the following command to see the value in every register for the entire duration of your debugging session.

```
(gdb)layout r
```

However, you might want to inspect a register's value in a specific format, then similar to above you can do:

```
mov al, '4'
...
(gdb)p $rax
52

(gdb)p/c $rax
52 '4'

(gdb)p/c $al
52 '4'
```

If you want to see the value of **output** at each breakpoint, then instead of using **print**, use **display**.

```
(gdb)display (char[5]) output
```

Finally, to exit gdb you can run:

```
(gdb)q
```

5 Additional Resources

5.1 Ascii Table

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figure 2: ASCII Table

5.2 Assembly Syntax Tables

The following tables are provided as a reference that may be used if needed, for all ASM questions in tests.

Data Items

`db`	data byte	1-byte
`dw`	data word	2-bytes
`dd`	data double word	4-bytes
`dq`	data quad word	8-bytes

Conditional Moves

instruction	effect
`cmovz`	move if ZF=1
`cmovnz`	move if ZF=0
`cmovl`	move if SF=1
`cmovle`	move if SF=1 or ZF=1
`cmovg`	move if SF=0
`cmovge`	move if SF=0 or ZF=1

Conditional Jumps

instruction	meaning	aliases	flags
`jz`	jump if zero	`je`	`ZF=1`
`jnz`	jump if not zero	`jne`	`ZF=0`
`jg`	jump if $>$ zero	`jnle ja`	`ZF=0`, `SF=0`
`jge`	jump if $\geq$ zero	`jnl`	`SF=0`
`jl`	jump if $<$ zero	`jnge js`	`SF=1`
`jle`	jump if $\leq$ zero	`jng`	`ZF=1` or `SF=1`
`jc`	jump if carry	`jb jnae`	`CF=1`
`jnc`	jump if not carry	`jae jnb`	`CF=0`

Floating Point Conditional Jumps

instruction	meaning	aliases	flags
`jb`	jump if below	`jc jnae`	`CF=1`
`jbe`	jump if below or equal	`jna`	`ZF=1` or `CF=1`
`ja`	jump if above	`jnbe`	`ZF=0` or `CF=0`
`jae`	jump if above or equal	`jnc jnb`	`CF=0`
`je`	jump if equal	`jz`	`ZF=1`
`jne`	jump if not equal	`jnz`	`ZF=0`

Useful Part of System V ABI for x86-64 Linux

Register	Usage	Preserved across function calls
`rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1^{st} return register	No
`rbx`	callee-saved register; optionally used as base pointer	Yes
`rcx`	used to pass 4^{th} integer argument to functions	No
`rdx`	used to pass 3^{rd} argument to functions; 2^{nd} return register	No
`rsp`	stack pointer	Yes
`rbp`	callee-saved register; optionally used as frame pointer	Yes
`rsi`	used to pass 2^{nd} argument to functions	No
`rdi`	used to pass 1^{st} argument to functions	No
`r8`	used to pass 5^{th} argument to functions	No
`r9`	used to pass 6^{th} argument to functions	No
`r10`	temporary register, used for passing a function's static chain pointer	No
`r11`	temporary register	No
`r12-r15`	callee-saved registers	Yes

Common C/C++ Wrapper system calls

int open(char* pathname, int flags [,int mode]);
int read(int fd, void* data, long count);
int write(int fd, void* data, long count);
long lseek(int fd, long offset, int whence);
int close(int fd);