X86 ASSEMBLY

NASM

Sudo apt install nasm

Nasm is an *assembler*, which is similar to a compiler in that it translates assembly in to machine code.

An assembler is a program that converts assembly (low level code), into machine code. A compiler is a program that converts high level code (C, Java, C++, …) in to low level code (assembly, machine code…)

Global \_start ; defines entry point

Section .text ; section

\_start: ; label called \_start (also the entry point)

Mov eax, 1 ; move integer 1 in to eax register

Mov ebx, 2 ; move integer 2 in to ebx register

Int 0x80 ; perform interrupt

Int 0x80 – 80 is an interrupt handler for a system call. The system call which it makes is determined by the eax register value. In this example, the value 1 was moved in to the eax register, 1 is the value for the exit call function. Ebx is used as the exit status, in this example 2 is passed to ebx. Usually, you would pass 0 to ebx to show that that program ran with no errors.

To assemble in to machine code you first need to make the object file, i.e. the assemble (1). And then you need to link these object files to create an executable (2).

1. Nasm -f elf32 main.asm -o main.o
2. Ld -m elf -i386 main.o -o main

Run the program using ./main. After running the program you can view the exit status by doing: “echo $?” in the terminal.

OPERANDS

|  |  |  |
| --- | --- | --- |
| **Operand** | **Desc** | **Example** |
| Mov | Move | Mov eax, ebx |
| Int | Interrupt | Int 0x80 |
| Sub | Subtract | Sub ebx, 2 |
| Add | Add | Add ebx, ebx |
| Mul | Multiply – always applied to eax | Mul ebx (eax \*= ebx) |
| Div | Divide – always applied to eax | Div ebx (eax /= ebx) |

HELLO WORLD

Global \_start

Section .data ; data section to create constants

Msg db “Hello World!”, 0x0a ; create variable msg(db =define byte),0x0a = \n

Len equ $ - msg ; create variable len equal to length of message

Section .text ; section for actual code

Mov eax, 4 ; 4 is the system write system call

Mov ebx, 1 ; 1 is stdout file descriptor for system write call

Mov ecx, msg ; bytes to write

Mov edx, len ; number of bytes to write

Int 0x80 ; perform system write call

Mov eax, 1 ; system exit call

Mov ebx, 0 ; exit call status

Int 0x80 ; perform exit

An assembly program can be divided in to 3 sections:

* .data – declaring constants
* .bss – declaring variables
* .text – the actual code

Although these can be called anything, these section names follow the current best practices.

INSTRUCTION POINTER

The instruction pointer is the internal pointer in the processor labelled EIP which holds the location of the machine code that the processor is currently executing. This allows the processor to jump around the code by altering this pointer.

The EIP instruction pointer is changed by using the *jmp* operator.

Global \_start

Section .text

\_start:

Mov ebx, 42

Mov eax, 1

Jmp skip

Mov ebx, 13

Skip:

Int 0x80

; after compilation the exit status will be 42, as the code after where jmp was specified was skipped, and the instruction pointer never returned back to this position.

CONDITIONALS

|  |  |
| --- | --- |
| **Operand** | **Description** |
| Je A, B | Jump if equal |
| Jne A, B | Jump if not equal |
| Jg A, B | Jump if greater |
| Jge A, B | Jump if greater or equal |
| Jl A, B | Jump if lesser |
| Jls A, B | Jump if less or equal |

\_start:

Mov ebx, 42

Mov eax, 1

Mov ecx, 99

Cmp ecx, 100 ; compare ecx to the integer 100

Jl skip ; jump if ecx is less than 100 (which it is)

Mov ebx, 13

Skip:

Int 0x80

LOOPING

Global \_start

Section .text

\_start:

Main ebx, 1 ; starting value

Mov ecx, 4 ; number of iterations

Label:

Add ebx, ebx

Dec ecx ; decrease the number of iterations by 1 each loop

Cmp ecx, 0 ; compare the iterator (ecx in this eg), to 1

Jg label ; jump back to label start if ecx greater than 0

Mov eax, 1 ; prepare system exit call

Int 0x80 ; exit

MEMORY ACCESS

Global \_start

Section .data

Addr db “yellow” ; db = define byte (1 byte), dw = 2 bytes

Section .text

\_start:

Mov [addr], byte ‘H’ ; replace index 0 of addr with H

Mov [addr + 5], byte ‘!’ ; replace index 5 of addr with !

Mov eax, 4 ; prepare system write call

Mov ebx, 1

Mov ecx, addr

Mov edx, 6

Int 0x80 ; perform system write call

Mov eax, 1 ; prepare system exit

Mov ebx, 0

Int 0x80 ; perform exit call

The stack

The stack is a LIFO data structure which you can push on top off and pop from. It is just an array with a pointer pointing to the top of the stack.

The *ESP* register is the register/pointer that points to the top of the stack

|  |  |  |
| --- | --- | --- |
| **Code** | **Memory address** | **Stack value** |
| Push 1234 | [00] | 0 |
| Push 8765 | [04] | 0 |
| Push 246 | [08] | 0 |
| Push 357 | [12] | 0 |
|  | [16] | 357 (current ESP register val) |
|  | [20] | 246 |
|  | [24] | 8765 |
|  | [28] | 1234 |
|  | [32] | 0 |

Notice that when pushing, the memory addresses decrement by 4. When popping, the memory address increment. The memory addresses go up by 4 as we are pushing 4 byte integers (32 bits).

You can also work with the ESP register directly, for example (assuming push 357 in table above wasn’t executed):

Sub esp, 4 ; point the ESP register to the current value minus 4 bytes

Mov [esp], dword 357 ; write 357 at this new address

Pop ebx ; pop from the stack and store value in ebx register

When popping from the stack, the value is not removed from the stack until another value is pushed on to it. In the example above, the value 357 will now be the value of 357 and the ESP register will be pointing at the value below it (246 in this example). If we push a new value, then 357 will be replaced with this.

*Program which allocates a string on the stack and writes it to stdout:*

Sub esp, 4 ; create space on stack

Mov [esp], byte ‘H’

Mov [esp + 1], byte ‘e’

Mov [esp + 2], bytes ‘y’

Mov [esp + 3], byte ‘!’

Mov eax, 4 ; prepare stdout sys call

Mov ebx, 1

Mov ecx, esp

Mov edx, 4

Int 0x80

Mov eax, 1

Mov ebx,

Int 0x80

; output: Hey!

FUNCTIONS

Functions can be invoked using the *call* operation. This pushes the *EIP* (instruction pointer), to the stack, and performs a jump to the function.

\_start:

Call func

Mov eax, 1

Int 0x80

Func:

Mov ebx, 42

Pop eax ; the return location which *call* pushed on the stack

Jmp eax ; now we can jump back to where we *called*

This can also be written like (best practice):

Func:

Mov ebx, 42

ret ; return

What if you want to use the stack in this function and still want to return from it?

You could be really careful and pop anything off the stack which you pushed on to it. This has to be done before calling *ret* because you want the *ret* instruction to return the address which the call operation pushed on to the top of the stack.

There is a common technique for preserving the stack using a register known as the *base pointer* (EBP).

Func:

Push ebp ; see \*\*

Mov ebp, esp ; store top of the stack at ebp

Sub esp, 2 ; allocate 2 bytes to the stack

Mov [esp], byte ‘H’

Mov [esp + 1], byte ‘i’

Mov eax, 4

Mov ebx, 1

Mov ecx, esp

Mov edx, 2

Int 0x80

Mov esp, ebp ; restore esp with the ebp value at top of this func

Pop ebp ; see \*\*

ret

\*\* what if the function calls another function within it?

A common technique is to push the old value of EBP on to the stack, and pop it before you return. (see underline instructions above). This will allow for nested function calls.

Passing parameters

Arguments are passed in via the stack

\_start:

Push 21 ; the argument to pass to times2

Call times2 ; call function

Mov ebx, eax ; return value (42), used as exit status

Mov eax, 1 ; system exit call

Int 0x80

Times2:

Push ebp

Mov ebp, esp

Mov eax, [ebp + 8] ; this is where arg 1 lives, arg2 = +12, etc…

Add eax, eax ; function logic – times arg by 2

Mov esp, ebp ; restore the stack pointer

Pop ebp ; pop off the old base pointer

Ret

*Example of stack pointer*

[00] 0

[04] 0

[08] 0

[12] 0

[16] 0

[20] old base pointer ; by this line: mov ebp, esp

[24] times2 return address ; by this line: call times2

[28] 21 ; by this line: push 21

[32] 0

Run the program, then echo the exit status via: echo $? ; Output: 42

TEST

PART 1

1. Write a program which exits successfully with the value 5 as the exit status
2. Assemble and display the exit status of the program above
3. What are the 2 different parts to assemble and assembly program
4. Give example of:
   1. Mov
   2. Add
   3. Sub
   4. Mul
   5. Div
5. Write a hello world program

PART 2

1. What 3 sections can an assembly program be broken down in to
2. Create a basic loop program which doubles a number over 3 iterations
3. Give an example of jumping using:
   1. Jump if equal to
   2. Jump if greater than
   3. Jump if lesser than and equal to
4. What is the ESP register?
5. What data structure is *the stack*
6. Give example of pushing/popping from the stack

PART 3

1. Push/pop on/off the stack by modifying ESP directly
2. Create a function which modifies the ebx value to use as the exit status
3. Create a function which modifies the stack within the function, and modifies the ebx value, before returning to the point where the function was called and exiting the program
4. Pass an argument to a function, double it within the function, return it from the function and use this value as exit status of the program