

Forum:[Hack Forums](#)

Thread: [Assembly Programming Tutorial - 1](#)

Deadbyte - February 11, 2015, 3:16 pm

Assembly Programming Tutorial - 1

I assume that you have some grasp on some programming language like Pascal, C or C++.

Note that you might be frustrated for not getting outputs printed on the screen. But please be understand that it would involve a lot of basic instructions. So, my style here is to churn out several chapters before doing output on the screen. I suggest you to have a debugger like Turbo Debugger to directly watch the effect of each instruction. This can be great for learning.

OK, here is the contents of the lesson:

Preliminary lesson -- Low Level Basic Concepts

Talks about registers, flags, memory, stacks, and interrupts. Don't worry about that too much. You might be confused with so many concepts. However, as you follow the lesson, I think every concepts should be clear enough.

Chapter 1 -- Basic program structure

Begin your journey in assembly by observing the simplest program structure.

Chapter 2 -- Variables in Assembly

Discover the unique concept of variables in assembly language. The notion is far different than that of the normal high level programming language. I also explain how mov instruction works.

Chapter 3 -- Arithmetic Instructions

How can we perform some arithmetic in assembly? Check this to find out.

Chapter 4-5 -- Under construction (Bitwise logic here, but it's unavailable in current CPU version)

Chapter 6 -- Branching

Branch is essential for all programs. Let's try some assembly branching instructions to improve the logic of our programs.

Chapter 7 -- Loop

The loop instruction in assembly can be useful to resemble higher level programming language construct.

Chapter 8 -- Stacks

Using push and pop and knowing how the stack behaves. Some details about tiny memory mode is explained here.

Chapter 9 -- Making Subroutines

Using subroutines to mimic structured programming approach.

Chapter 10 -- A Bit of Theory

Addressing modes and memory modes explained

Low Level Programming Basic Concepts

Introduction

Hi! Welcome to low-level world! This time I would like to explain the basic concepts of low level.

Registers

What is registers exactly? You can consider it as variables inside the CPU chip. Yeah! That depicts registers so close. There are several registers exist in PC:

EAX, EBX, ECX, EDX, CS, DS, ES, SS, ESP, EBP, ESI, EDI, CMPR, and IP/EIP

They are all 32-bit floating point. You can treat it as if they are real variables. However, each register has its own use.

EAX, EBX, ECX, and EDX are general purpose registers. They can be assigned to any value you want. Of course you need to adjust it into your need. EAX is usually called accumulator register, or just accumulator. Most of arithmetical operations are done with EAX. Sometimes other general purpose registers can also be involved in arithmetical operation, such as EDX. The register EBX is usually called base register. The common use is to do array operations. EBX is usually worked with other registers, most notably ESP to point to stacks. The register ECX is commonly called counter register. This register is used for counter purposes. That's why our PC can do looping. EDX register is the data register. It is usually for reserving data value.

The registers CS, DS, ES, and SS are called segment registers. You may not fiddle with these registers. You can only use them in the correct ways only. CS is called code segment register. It points to the segment of the running program. We may NOT modify CS directly. Oh yes, what is "segment" anyway? It's discussed later. DS is called data segment register. It points to the segment of the data used by the running program. You can point this to anywhere you want as long as it contains the desired data. ES is called extra segment register. It is usually used with EDI and doing pointers things. The couple DS:ESI and ES:EDI are commonly used to do memory operations. SS is called stack segment register. It points to stack segment.

The register ESI and EDI are called index registers. These registers are usually used to process arrays or strings. ESI is called source index and EDI is destination index. As the name follows, ESI is always pointed to the source array and EDI is always pointed to the destination. This is usually used to move a block of data, such as strings and arrays. These register is commonly coupled with DS and ES.

The register EBP, ESP, and IP are called pointer registers. EBP is base pointer, ESP is stack pointer, and IP is instruction pointer. Usually EBP is used for preserving space to use local variables. ESP is used to point the current stack. Although ESP can be modified easily, you must be cautious. It's because doing the wrong thing with this register could cause your program in ruin. IP denotes the current pointer of the running program. It is always coupled with CS and it is NOT modifiable. So, the couple of CS:IP is a pointer pointing to the current instruction of running program. You can NOT access CS nor IP directly.

The CMPR register is used to store the result of previous compare. It is used by CMP and conditional instructions. You can NOT access it directly.

Memory

Of course the programs code (code) is placed in the memory. The memory is actually numbered as their address. That is from 0, 1, 2, and so on. To address data in the memory, CPU uses registers. Originally, CPU only has 64KB memory on-chip. However, you can expand the address space using hi-speed link mechanism, which extends memory to any size. The technique called segmentation was invented. That means the memory is divided virtually into several areas called segments.

Upon the arrival of segmentation, the segment registers are also exist to corporate the idea. They are used to simplify access to memory regions, by "snapping" memory to "virtual memory" blocks, which can be addressed from address 0, and so on. The real address is formed as sum of segment register value and memory address.

It means that if we say the segment number 100, then we can access the memory from address 100. Segment number 200 allows us to access memory from address 200, and so on. Why did they do that? It is for the sake of the operating system memory management stuff. Therefore, you can "bind" some memory regions, for example data region, or extra device memory region.

The memory access must be done in a pair of register. The first is the segment register and next is any register, usually EBX, EDX, ESI or EDI. The register pair usually written like this: ES:#EDI with a colon between them, and # symbol in front of register, which means that we reference the memory pointed by EDI. The pair is called the segmentffset pair. So, ES:#DI means that the segment part is addressed by ES, and the offset part is addressed by EDI. Of course there is interleaving between segments.

If the ES contains 0, and EDI is 5, means that we access the memory 5. If ES:EDI = 10:5 then it actually access the actual address 15 ($10 + 5 = 15$). Remember the interleaving I've mentioned above. So, 0:15 and 10:5 is actually the same address. How could the processor do that? The register pair segmentffset contains the logical address. The actual address or the absolute address need to be calculated from the logical address. We just need to add the segment value to address! Simple, eh?

Stacks

Now, let's talk about specific memory lay out. You load the program code into memory. A specific amount of memory is reserved in order to make the program runs as expected. There is one thing: there must be a room for the code itself, then there must be a room for data, and the last thing is there must be a room for stack.

What is stack exactly? You can say that it is a temporary area to store temporary things. It is mainly used to pass the parameter value to procedures or functions. Sometimes, it also act as temporary space to allocate for local variables. Therefore, the role of the stack is very important.

How the stack works? It works exactly as the stack in linked list! The last item pushed into stack is going to be popped first. LIFO concept works here. At this moment, you don't have to know how stack work in depth. The main thing is that you know that stack here uses LIFO concept, but it is NOT a linked list.

How to adjust stacks? Reserve as much memory as needed for stack. If you use many parameters in your procedure or functions, you need to reserve bigger stack. Usually 2 KB or 4 KB is enough for many programs. However, if you use a lot of local variables, you need to reserve more.

That's all about the stacks, let's advance.

Interrupts

What is interrupts exactly? It is like its name: interrupts. It interrupts processes. Upon a request of an interrupt, the processor usually stores only the CS:IP and state of the running program, then it goes to the interrupt routine. After processing the interrupt, the processor restores all states stored and resume the program. There are three kind of interrupts: hardware (other than CPU) interrupts, software interrupts, and CPU-generated interrupts.

Hardware interrupts occurs if one of the hardware inside your computer needs immediate processing. Delaying the process could cause unpredictable, or even, catastrophic effects. Keyboard interrupt is one of the example. If you pressing a key in your keyboard, you generate an interrupt. Keyboard chips notify the processor that they have a character to send. Can you imagine if

the processor ignores the request and go on with his own business? Your key is never processed!

Software interrupts occurs if the running program requests the program to be interrupted and do something else. It is usually like waiting the user input from keyboard, or may be request the graphic driver to initialize itself to graphic screen.

CPU-generated interrupts occurs if the processor knows that is something wrong with the running code. It is usually directed for crash protection. If your program contains instructions that processor doesn't know, the processor interrupts your program. It also happens if you divide a number with 0. The result does not exist anyway.

Interrupts have a lot of uses. They may have routines that ease programming life. Changing into graphic screen, waiting for a key, accessing files, disks and so on are done through interrupts.

Notes

Finally, you get all the basic concepts. There are still a lot more to learn in low-level programming. Now, go to the implementation. Pascal, C / C++, or even assembler.

Chapter 1: Basic Program Structure

Welcome

Welcome to your first assembly lesson. I suggest you to read the preliminary lesson before proceeding. If you have read it, I understand that everything is not clear. Now, I try to clarify things here. I assume that you are able convert numbers from decimal to binary or to hexadecimal and vice versa. You would probably need a ZCPU documentation, so you can look up opcodes and references.

Why Assembly?

Some of you may complain about assembly: It's difficult, error prone, hard to debug, takes a lot of time to develop, etc. Yes, that's true. However:

1. Assembly is fast. A LOT faster than any compiler of any language could ever produce.
2. Assembly is a lot closer to machine level than any language because the commands of assembly language is mapped 1-1 to machine instructions.
3. Assembly code is A LOT smaller than any compiler of any language could ever produce.
4. In Assembly, you have the RAW power of your cpu. You can tweak it any way you want.
5. In Assembly, we can do a lot of things that we can't do in any higher level language, such as playing with processor features, etc.
6. Assembly provides a way to replace large amount of gates with a single CPU chip, when even expression gate can't help!

Basic Structure

This structure is quite straight forward to implement. However, there are some restrictions apply: Code and data would be contained within 64KB (internal memory)
CPU is not operation in protected mode, so you can't handle errors via interrupts. You can define your own interrupts though.

External devices would be linked to beyond-64KB memory

Well, don't bother about the 64KB restriction. It is a whole lot of code and data -- for now. Trust me! Probably after you code some assembly language, you'd understand why Bill Gates said that 1 MB

is THE unreachable limits years ago.

OK, let's look into our first program (ZASM 1.0):

Code:

PHP Code

```
<?php  
jmp start;
```

```
//Your data allocation here
```

start:

```
mov eax,0;  
add eax,10;
```

It's pretty much similar. Let's save it as "myprog.txt" in garrysmo/d/data/CPUChip/ folder (this is where your programs go).

You can also copy it into ZASM directly, if you wish doing so.

Here is line-by-line explanation on the program:

jmp start - jumps over data region (where you can place your variables)

start: - entry point of program. Code execution starts here (after the jump)

mov eax,0 - store zero in EAX register

add eax,10 - add ten to EAX register. By the end EAX will hold value of "10"

Part 2 is coming soon!

This tutorial is based on Roby's programming tutorial © by Roby Joehanes 1997, 2000, 2001

Bithigh - February 12, 2015, 6:44 am

Good job on this.

You should use

Code xor eax, eax

to set eax to 0. Its a lot faster than using mov eax, 0. Also by xor eax, eax you don't have a NULL byte in shellcode if you're writing payload for exploit.

Deadbyte - February 12, 2015, 9:01 am

(Feb 12, 2015 07:44 AM)Bithigh Wrote: [\[CITE\]](#) Good job on this.

You should use

Code xor eax, eax

to set eax to 0. Its a lot faster than using mov eax, 0. Also by xor eax, eax you don't have a NULL byte in shellcode if you're writing payload for exploit.

Thank you for the notice I will modify.