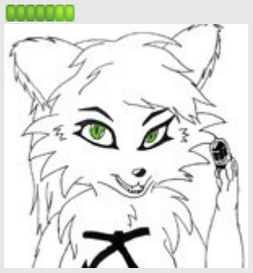
04-05-2007 #1

**Black Phoenix**

That furred thing

WIREMOD WM DEVELOPER



Join Date Feb 2007
Location Kyiv, Ukraine
Posts 3,551

## Assembly Programming Tutorial

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

Hi! Welcome to ZAssembly lesson. This assembly lesson is for Zyelios CPU, version 2.20. Yes, it is compatible with your CPU from recent SVN. I assume that you have some grasp on some programming language like Pascal, C or C++. I don't want to go over the basic concepts of programming all over again.

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.

If you are impatient, just read preliminary chapter, chapter 1 and 2. Then skim chapter 3 through 7.

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. Howeve 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 wit 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 pointer 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 t 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 th 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 we 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 registe and next is any register, usually EBX, EDX, ESI or EDI. The register pair usually writte like this: ES:#EDI with a colon between them, and # symbol in front of register, whic 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 memor 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 mus 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 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. Upor 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 i 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, try to clarify things here. I assume that you are able convert numbers from decimal tc 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:

```
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 *garrysmod/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"

**Making Labels**

In assembly, to make labels is simple, just put any name and stick it with a colon( : ). Label usually serves as a tag of where you'd like to jump and so on. You have to pick unique names for each label, otherwise the assembler will fail. My personal way to giv labels unique names - let each subroutine have prefix for labels, which (for example) points to line number that was when procedure was created. Neat? Maybe.

**Note**

In ZCPU, if program execution reaches zero opcode, or end of program (no code), you program will stop the CPU. Pay attention to this. You might have to re-trigger CLK input of your CPU. Thats why you may also want to put "jmp start" in the end to loop the execution of program (so CPU operates in "gate mode")

Assembly language command in ZCPU platform is usually formatted as follows (just like in x86):

Code:

```
mnemonic target,source;
```

Mnemonic is just the jargon for assembly commands. Why is it called that way? It is because that the commands in assembly (somewhat) resemble English words. Then followed by the target, and then comma, and then the source. Label cant precede the command, unless they are separated with ";". For example:

Code:

```
mov eax,1234;
```

Is to move the value 1234 into the register EAX. Simple right?

If there is only one parameter (like in "int 10"), usually it denotes the source or the destination depending on the command. Like in jmp start, it means jump to the destination start. As in int 2, you are calling interrput number 2.

**Don't forget that several mnemonics in same line should be seperated with ";" symbol! Same goes to labels.**

If you did everything right, you can upload this program in CPU. Look in console if any errors have occured. No errors? Look at size, our program takes only 9 bytes!

# Chapter 2: Variables in Assembly

**Welcome**

Hi! Welcome to the second chapter of this series. This time, I'm going to talk about th variables in Assembly. This is quite significant since it behaves differently than that of the high programming languages like C/C++, Pascal, or Java. I hope after following this lesson, you'll be able to use the assembly variable context fully. Let's begin!

Before starting, I just remind you that assembly language is NOT case-sensitive. It means that "this", "This", and "tHiS" are considered as the same. I just forgot whethe I have mentioned this in the last chapter or not.

Moreover, I remind you that comments in zyelios assembly begins with a double dash (//). Everything after a double dash until the end of the line is ignored. So, it's like a double-slash in C++ or a quote in BASIC.

**Variables Declaration**

If you recall our first program from the last chapter, it looked like this:

Code:

```
jmp start;

  //Your data allocation here

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

Notice where you'd put your variable declarations. It's right after the jmp start statement. Well, you can actually place your declarations anywhere inside the program. However, for now, let's just place them there. Placing it after the start label can be disastrous if you do not handle it carefully.

OK, now, there are 2 ways you can declare a variable - using DB macro, and using alloc macro. Both allocate single varible (alloc can also allocate array of values, but more on that later). We will use the alloc for now, because its easier to understand an to use.

How can we declare variables then? The declaration syntax is as follows:

Code:

```
alloc var_name,value;
```

That's very simple. You need to change var_name with your variable name. You will need to change value into default value for variable, or you can ignore that, and use this syntax:

Code:

```
alloc var_name;
```

In that case variable will be defaulted to 0.

For example:

Code:

```
jmp start;

  //Your data allocation here

  alloc money,1234;
  alloc time,0;
  alloc counter;
  alloc temp;
```

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

Nicey!

**Moving Around Values**

Still remember what registers are? Registers can be treated as variables that reside in CPU chips, right? We have a handful of those registers: EAX, EBX, ECX, EDX, ESI, EDI and so on. If you need some review, scroll up.

"If we have some registers built-in, why on earth do we need another variables to declare then?" Hmm, there are a couple of reasons. The first is that of course we neec to have more variables than just about twelve. The second is that some registers can' be used for storing values at all, for example CS and IP, as they are critical to running the program. That limits our freedom to about 6 or 7 registers, which is not adequate for most of our needs.

However, if you need to do some calculations or commands involving the variables, in most occasion, you'll have to load the variable values to the registers. Loading those values up to the registers and storing the value from registers to variables can be don through the mov command.

The syntax of the mov command is mov a,b which means assign b to a (i.e. a := B ). So, in our first program, we observe the command mov eax,0. That means eax = 0. We give the register EAX value 0. Note that we CAN'T do mov 0,eax because that makes no sense :-)

Similarly, you can load the variables to a register or store them back. You can even transfer values between registers. Let's look at the example below:

Code:

```
jmp start;

  alloc myvar;

start:
  mov #myvar,10;
  mov ecx,#myvar;
  mov eax,ecx;
```

Ah. that's pretty straight forward. The first statement (i.e. mov #myvar,10) is to load myvar variable with constant value - ten. The second statement (mov ecx,#myvar) is to transfer that value from myvar to CX. The third statement (mov eax,ecx) is to copy the value from ECX register to EAX register. Note the sharp sign when you deal with variables. These sharp signs are good to distinguish the variable from its address so that in later on when we deal with pointers, we are not confused.

**Sharp in MOVs**

Variables in assembly are treated differently than that of any high level programming language (Pascal, C/C++, Java, etc). The assembler actually treat variables as a label that has an address in the memory (RAM in most cases) associated to it. Moreover, th assembly language is later assembled (or compiled) into a machine codes. Informations concerning variable names and their respective types are LOST. Also, assembler does not check the types of variables. It simply doesn't care.

This fact confuses a lot of people learning assembler, especially if they have no low level view at all. However, this is the benefit of assembler. You can tweak it around, use it or abuse it. ;-)

That's the way the variables work.

**[Not Exactly] String Variables**

You can define strings variables in assembly. It is as follows:

Code:

```
message:
   db "Hello World!",0;
```

String variables are required to be stored as db variables. The string is then surrounded by quotes. If you begin a string with a double quote, you'll have to close it with another double quote. This is neat.

Why do we have to end our string with a zero? Because zero is a useful value to stop our string write operations, so they wont read extra memory we dont want for them to :-)

How strange! Yes, we've got to live with these peculiarities. So, before doing your string, make sure how your string will go. You may want to add "size" value in front of string, or you may want to have zero in the end, its up to you.

How are the string variables are stores then? It is similar to the normal variables except now, each characters of the string is converted to its corresponding ASCII code. Uh oh, need to memorize them. ;-) You can download any ASCII table program somewhere on the net. There are plenty of resources.

So, suppose we have that message variable above stored at address 100h. The memory contents is as follows:

Code:

```
Address     Value     ASCII Code (hex)
100           H       48h
101           e       65h
102           l       6ch
103           l       6ch
104           o       6fh
105           <space> 32h
106           W       57h
107           o       6fh
108           r       72h
109           l       6ch
110           d       64h
111           !       21h
112           #0      00h
```

Hmm... The string is stored contiguously.

**Using alloc**
Alloc can be used in 4 ways:

alloc; - will allocate 1 value and set it to zero
alloc <value>; - will allocate <value> values
alloc <name>; - will allocate 1 value with specific name
alloc <name>,<value>; - will allocate variable with specific name and <value>.
alloc <name>,<value>,<value2>; - will allocate array with name and set its members to <value2>

**Closing**

I know this new variable concepts will overwhelm you for a moment, but fret not. You'll get used to it. One suggestion is to draw its memory table. This will clarify most situations. If you are in doubt, you're always welcome to review through the preliminary chapter, chapter 1 and 2.

So far, I haven't even explained on a program that says "Hello World!" All I did is to explain you the crucial concepts on assembly language just to enable you to write some basic assembly program. With these key concepts in mind, you'll hopefully program in assembly more gracefully.

OK, I think that's all for now. See you next time.

# Chapter 3: Arithmetic Instructions

**Welcome**

Hi! Welcome to the third chapter of this series. Now, we'll try to learn a bit more abou doing math in assembly. I hope that you really grasped the idea discussed in the previous chapter. The arithmetic operations discussed here is done in reals.

**Addition And Subtraction**

Additions and subtractions are straightforward. It takes the following formats:

Code:

```
add x,y; //--> means:  x = x + y
sub x,y; //--> means:  x = x - y
```

Additions and subtractions can be done on any registers except the segment registers Just like the mov command, you can have one of them as memory locations. I encourage you to experiment on how to use this command. What is allowable and what is not. The following is legal in assembly (assuming the variables are already defined):

Code:

```
add eax,5;  // --> means:  eax = eax + 5
add ebx,ecx;// --> means:  ebx = ebx + ecx
add #n,eax; // --> means:  #n  = #n  + eax
add ecx,#n; // --> means:  ecx = ecx + #n
sub edi,edi;// --> means:  edi = edi - edi  (in other words: edi = 0)
sub eax,esi;// --> means:  ...
sub eax,4;  // --> means:  ...
```

As I pointed out in the previous chapter, for those of you that has 80286 processor or faster may actually add or subtract variables with constants. For example:

Code:

```
add #i,10;
```

**Multiplication, Division, and Remainder**
Multiplication, division, and reminder are same as ADD and SUB:

Code:

```
mul eax,5;  // --> means:  eax = eax * 5
mul ebx,ecx;// --> means:  ebx = ebx * ecx
div #n,eax; // --> means:  #n  = #n  / eax
div ecx,#n; // --> means:  ecx = ecx / #n
mod edi,eax;// --> means:  edi = edi mod eax
mod eax,esi;// --> means:  ...
mod eax,4;  // --> means:  ...
```

**Increment and Decrement**

Often times, we'd like to incrementing something by 1 or decrement thing by 1. You can use add x, 1 or sub x, 1 if you'd like to, but ZASM assembly has a special instruction for them. Instead of add x, 1 we use inc x. These are equivalent. Likewise

in subtraction, you can use dec x for subtraction.

**A Nice Tip**

The arithmetic operations can have special properties. For example: add x, x is actually equal to multiplying x by 2. Similarly, sub x, x is actually setting x to 0. These arithmetic is faster than doing mul or doing mov x, 0. Ha! Even more, its code size is smaller. No wonder why the assembly wizards often fond of this subtraction.

**Closing**

OK, I think that's all for now. See you next time.

# Chapter 6: Branching

**Using Unconditional and Conditional Jumps**

**Welcome**

Hi! Welcome to the sixth chapter of this series. Now, I'd like to explain about how can we do jumps, conditionally or unconditionally. Conditional jumps always consider some condition. If the condition is satisfied, then the jump is taken, otherwise it is not. The conditions are usually reflected in the processor flags. This may be confusing at first. So, I'll take a different approach so that you, that are more familiar with higher level language, can quickly absorb the idea. This conditional jumps is basically like an if construct by large.

On the other hand, unconditional jumps do not regard any conditions. So, it is more like gotos in a sense. In assembly, we can do a more interesting stuffs for sure. But I don't want to explain the bag of clever tricks here at first. I'm covering the basic ones and will reveal more deeper materials later, probably in lesson 2 or so.

**Making Labels**

Labels are essential to jump instructions. It marks the destination. Of course you need to set where to jump, don't you? ;-) Making labels in assembly are easy. Perhaps I already explained this in chapter 1, but let me reclarify this once more. Labels can be made like this:

Code:

```
example:
```

So, we can pick out any names and stick a colon after it (":"). Voila! That's it. You must make sure that all label names throughout your programs are unique, no duplicates.

**Unconditional Jumps**

Syntax of jump instructions are the same. It usually takes form of jump_instr dest_label and of course we have lots of different jump instructions in ZCPU processor For unconditional jump, the instruction is jmp. For example:

Code:

```
jmp   somewhere
              ..
              ..
 somewhere: // it will continue here
```

As mentined earlier, unconditional jumps takes no regard on conditions. So, whenever the processor arrives at the instruction jmp somewhere, it will directly skip all the

instructions below it up to until the instruction marked by the label somewhere.

**Conditional Jumps**

Conditional jumps are just verisimilar. However, before the jump instruction, we (usually) have to put a comparison or testing instruction. The comparison instruction I'm going to introduce here is cmp. As I already mention, the conditional jump is analoguous to an if construct. So, I'm going to explain it this way. Look at the followin example:

**Assembly:**

Code:

```
Assembly
    cmp   eax, ebx
    jg    isgreater
       :
       :   // block 1  (else part)
       :
    jmp   after
isgreater:
       :
       :   // block 2  (then part)
       :
after:
       :   // after if
```

**C/C++/Java Equivalent:**

Code:

```
if (eax > ebx)
    {
       // block 2
    } else
    {
       // block 1
    }
    // after if
```

**Pascal Equivalent:**

Code:

```
if eax > ebx then
    begin
       { block 2 }
    end else
    begin
       { block 1 }
    end
    { after if }
```

This should be clear enough. You can observe that the cmp instruction above is used t compare eax and ebx. The jump instruction is the one that actually does the check. So, the jump instruction used above is jg which means "jump if greater". Other commonly used jump instructions are as follows:

Code:

```
Instruction     Meaning
-----------------------------------------------
jg              Jump if greater
jge             Jump if greater or equal
jl              Jump if less
jle             Jump if less or equal
je              Jump if equal
jne             Jump if not equal
jc              Jump if carry flag is set
```

**Example 1**

Let's try to wrap up with an example. Say, we'd like to add 1+2+3+...+10. The program in C/C++ or Pascal is simple. Let's examine the assembly counterpart:

Code:

```
start:
   mov  ecx, 10       // The counter is in ECX
   mov  eax, 0        // I use EAX as a sum holder

myloop:
   add  eax, ecx      // eax = eax + ecx
   dec  ecx           // decrement the counter
   cmp  ecx, 0        // is ecx zero?
   jne  myloop        // if not, then loop. If yes, then quit.

quit:
   // here eax will hold the value of 1+2+...+10
```

Hopefully the examples above helps you to understand. Of course that example is not optimized and just plain vanilla.

**Example 2: Factorial**

The world of programming tutorials seems incomplete without doing this factorial example. I assume you know what factorial means. Suppose we'd like to calculate 8 factorial. Let's examine the example below:

Code:

```
start:
   mov  ecx, 8         // Again, ECX becomes the counter
   mov  eax, 1         // We will store factorial in EAX
                       // We initially set it to 1
myloop:
   mul  eax,ecx        // EAX = EAX * ECX
   dec  ecx            // decrement counter
   cmp  ecx, 0
   jne  myloop

quit:
   //here EAX will hold the value of the factorial
```

**Closing**

OK, I think that's all for now. Higher assembly level constructs like if, switch (or case..of in Pascal), while, and for are built using these commands. It involves a go to like construct and seems to be a taboo for structured programming approach. Well, that's all we've got. Anyway, jumping around is not that bad. ;-) It's probably better for you to construct higher level programming structure in terms of instructions you've learned so far. However, assembly has a loop instruction too. I'll discuss this on the next chapter. See you next time.

# Chapter 7: Loop Instructions

**Welcome**

Hi! Welcome to the seventh chapter of this series. Wow! You're kinda patient guy! Usually assembly learners gave up at the first three chapters. They cannot see any results printed on screen. This involves a lot of instructions. The style I'm using so far is to cover a few instructions step by step, refering to the higher-level programming language styles. I admit that it is hard to imagine every execution just in mind. Those impatient ones may want to go for a debugger like Turbo Debugger and stuffs to directly review the effect of each instruction executed. But hey, you're here! Alright, enough rant. Straight to the stuff.

Now, I'm going to cover loop instructions in ZCPU assembly. Actually, there is only one major loop instruction: loop. The others are just variants. This instruction can take any register via special opcodes, but the default one ("loop") uses ECX register. So, today I will only explain about plain vanilla loop. If you are interested, others are loopa for eax, loopb for ebx, loopd for edx.

**Loop Construct**

Syntactically, loop instruction resembles jump instruction: loop somelabel. Typically, the label is somewhere up there. Look at the following example:

Code:

```
:
   mov ecx, 100  // set the counter
mylabel:
      :
      :  // repeated instructions here
      :
   loop mylabel
```

As you observe, this structure is just like do..while construct in C/C++/Java, or like inverted repeat...until in Pascal. But here is the difference: when the processor takes loop instruction, it will first decrease the register ECX by one. After that, ECX is tested whether it is zero or not. If it is not zero, then jump to mylabel. So, it's kinda countdown counter. No, you can't modify this behavior. If you are dissatisfied with this you can always use conditional branches instead. But trust me, this construct is a bit faster than the usual branches. Therefore, I encourage you to transform your loop to conform with this norm. (Pardon the pun) :-)

Well, there is no restrictions on where the label is located, actually. You can place the label way down if you'd like to. This may not make sense because in that way, nothing gets repeated. Unless, some assembly wizards get some tricks to perform their arcane "sorcery".

Code:

```
Example
```

Let's take 1+2+...+10 example from the last chapter. We'll transform it to using loop instruction. The next snippets will show you how. Woow.... that's pretty short, right?

Code:

```
start:
   mov  ecx, 10       // The counter is in ECX
   mov  eax, 0        // I use EAX as a sum holder

myloop:
   add  eax, ecx      // eax = eax + ecx
   loop myloop

quit:
   // here ax will hold the value of 1+2+...+10
```

**Closing**

OK, I think that's all for now. See you next time. Get some popcorn, and prepare for more chapters

# Chapter 8: Stacks

**Welcome**

Hi! Welcome to the ninth chapter of this series. Today, I'd like to share about stacks and how we can use them in our program. Actually, this chapter acts as an overture fo the next one about subroutines. I think you should know this first before moving on. Soon, you will understand too.

I expect you to already understand the concept of general stack data structure, which is by nature last in first out (LIFO). I'm not going to reexplain it once more. I don't tal about linked list now, but the stack we discussed here is implemented in a kind of array. However, the concept is still the same.

**Why Stack?**

There are several reasons why we need stacks:

- To save register values if we ran out of registers.
- To pass parameters to subroutines
- To make space for local variables in subroutines
- To preserve original register values if we change them in a subroutine

Realizing this importance, we should learn stacks.

Stack operations mainly done by two instructions either push or pop. The instruction push will push values into the stack, while pop will pop it out. The syntax is like this:

Code:

```
push  x    // push x into stack
   pop   x    // pop  x out of stack
```

Of course, the nature of stack still apply: last in first out (LIFO). Remember the stack of plates.

**A Little Bit About Memory Layout**

Understanding the instructions is not enough. The more important thing is know how to use it. Before you know how to use it, you must understand how the memory is lie out in our assembly program.

The memory address of our computer has 2 elements: a segment and an offset. This pair will define exactly where in the memory, which I had covered here.

Recall that we have segment registers in our processor. Since every thing fits into one segment, there is no need to set segment registers. By default, the assembly will set all segment registers pointing into that single segment. That's why I said "Ignore setting DS" or something like that once.

You should know that register CS by default points to the segment where the code resides. DS will point to the data segment. ES usually pointed to data segment too. SS will point to stack segment. Since CS, DS, ES, and SS point to the same segment, it means code, data, and stack resides in the same region. How can we manage this? If you think this is a mess, this architecture applies to the modern computer nowadays whereas the concept was invented somewhere in the 1940s. If you've heard the name "Von Neumann architecture", this is it. Code, data, and stack is put in the same memory.

OK, now a bit deeper. The stack is not only pointed by SS register. But also ESP register. So, the pair SS:ESP points the top of the stack. Initially, ESP is set to the ver bottom of the segment, at address 0FFFEh (not 0FFFFh, that's the bottom end of the segment). Each time we push something into the stack, this SP register will be decremented up by 1. If we push something, SP will be incremented down by 1. Whereas, our code and our data starts at offset 0000h. So, the layout looks something like this:

Code:

```
Address | Meaning
    0   | Data & Code
    :   |      :
   EIP  | Next instruction
    :   |      :
    :   |      :
```

```
   ⋮   │      ⋮
   ⋮   │      ⋮
 65535 │ Top of stack
```

Notice that we have no boundary on stack. I draw a smear there since the boundary is not strictly defined. So, if you push too many things into the stack, the chance is it wil overwrite your code. If your code is overwritten, your processor may be running garbage. But don't worry, this arrangement will provide sufficient room for normal use of stacks.

Note that I only mention EIP there because CS:EIP pair will define the next instruction to be run. How about data? Well, the segment usually has to be DS (or sometimes ES and the offset part can point to anywhere in the segment.

To cope with this situation, recall our first assembly program:

Code:

```
DATA;
  // your data and subroutine here
CODE;
  // your code
```

When you exercise and try to build an assembly program, how many bytes is usually the result after compilation? It's usually less than 200 bytes, right? So, it's far from th end of the segment. Thus, you don't have to worry about running out of stack space. There is a "huge" empty region in-between.

**Application**

It's better for you to run this program in a debugger to see the effects.

Code:

```
start:
   mov  eax, 100  // initially ESP = 65535, EAX is set to 100
   push eax       // now ESP = 65534
   mov  eax, 200  // EAX = 200
   pop  eax       // ESP back to 65535, EAX is back to 100
```

See the effect? When EAX is stored in the stack, we're free to modify EAX. If we pop the old value back, voila! The old value is restored. This can be handy in storing (very temporary value. You don't need to declare variables and thus reducing the code size by some bytes.

In fact, subroutine call will use a lot of stack stuffs. So, you'd better understand the underlying concept before going on.

**Other Uses**

Can we push a constant? Yeah, we can. So, doing push 1 will store 1 on stack. No nee to specify # and stuff.

**Closing**

OK, I think that's all for now. See you next time. Popcornn!

# Chapter 9: Making Subroutines

**Welcome**

Hi! Welcome to the tenth chapter of this series. I hope that you really grasped the stack concept discussed in the last chapter. Now, I'm going to explain important concept in assembly language: Subroutine. ZASM itself does not have special function definition syntax, so you gotta "write" it yourself. Don't worry, I'll teach you how!

**Subroutine Syntax**

OK, so, how is the syntax? Let's suppose we'd like to declare a procedure called addstuff that take a 2 numbers as parameters. There are two possible syntaxes: one using C-style calling (using stack to pass parameters), and second is more classic - using registers to pass parameters. We will discuss both.

**C-Style calling with stack:**

Code:

```
addstuff:
  push ebx; //Save ebx, because we use it. We don't save EAX, because we retu

  mov eax,esp;
  add eax,4;     //2 (our pushed eax and ebx) + 1 (return address) + 2 (the f
  mov eax,ss:#eax; //Read from stack

  mov ebx,esp;
  add ebx,3; //2 (our pushed eax and ebx) + 1 (return address) + 1 (the secon
  mov ebx,ss:#ebx; //Read from stack

  add eax,ebx; //Result will be returned in EAX

  pop ebx;
  ret;
```

May not seem so simple, but it really is! You just should pay attention to stack - how much values you pushed and popped. Usually your parameters will be on addresses [ESP+<number of pushes inside function>+<number of parameters>+1],[ESP+<number of pushes inside function>+<number of parameters>-1+1],[ESP+<number of pushes inside function>+<number of parameters>-2+1] and so on (I.e. decreasing by 1 when parameter number is increasing by 1).

Note also that you have to mention the ret instruction to make sure that your subroutine properly returns to its caller. If you omit it, the processor will continue executing whatever instructions beneath it. So be careful! Beginning assembly programmer tends to forget this because of the features in high level languages.

Also note SS segment modifier - sometimes it might be different from 0, so to preserve code workability in all envrioments we add it.

Then how can we invoke it?

Code:

```
  push 123;       //First param
  push 456;       //Secod param
  call addstuff;
  add  esp,2;     //Remember those two values we pushed? They did not go anywh
```

Note the last instruction. The ESP register is added by 2. Why? We must restore the stack register too. You can do a pop instead, but the pop needs a victim register, whic may not be available. That's why we choose the add esp, 2 instead. If you're confused about this, you might wanna to pass parameters using registers, it's easier.

Note on return value: our value will return it's result in EAX register, so you better to have it free before calling it. Returning value is optional of course.

**ASM-Style calling using registers:**

Code:

```
addstuff: //Assuming when calling EAX will be set to 1st value, EBX to 2nd, a
  add eax,ebx; //Result will be returned in EAX
  ret;
```

Sweet, so simple! Invoking it is also pretty simple:

Code:

```
  mov eax,123;
  mov ebx,456;
  call addstuff;
  //EAX is set to PREVIOUS EAX + EBX, now profit!
```

Actually it's far more simple than it seems. Too much simple. Remember, that we only have 8 GPR registers. Minus one that is used for stack pointer. Minus one or two that might be used as temp storage (usually it happens so). We get only maximum of 6! parameters to pass. That's not much.

**Note:** in all function alls you gotta worry about the registers - when calling a function and returning from it registers are not saved on stack automaticaly, so you gotta do it yourself. Just push registers you will use at start, and pop them in the end.

You can use stack to allocate local variables - for example you could do this:

Code:

```
  mov ebp,esp;
  sub esp,123; //Allocate 123 bytes for your storage
  //Now assuming we don't modify ESP anywhere (no push or pop below this line
  //(for example, here we copy from #eax to local storage, and from local sto
  //(eax and ebx are already set, esi and edi are already pushed)
  mov esi,eax;
  mov edi,esp; //Save to stack!
  mcopy 123; //copy 123 bytes

  mov esi,esp; //Load from stack
  mov edi,ebx;
  mcopy 123; //copy 123 bytes

  mov esp,ebp; //restore stack pointer, free up local variable space
```

**A Word of Caution**

When you modify certain registers in a subroutine, it is likely you interfering the main program. Why? Suppose that the main program relies on EAX register a lot. Then it calls the subroutine. The subroutine modifies EAX, do something and returns. The EAX register has changed. This may be not expected by the caller. What would happen? Chaos.

I think it is a good habit to document a subroutine. At least give a comment above it. One example of this comment for the subroutine above is like this:

```
//-----------------------------------
// addup
// Adding 1 through n
// input:
// EAX - N
// returns:
// EAX = 1+2+...+n
//-----------------------------------
```

**Routine Placement**

OK, you now understand how subroutine works. Now, where we should place this? Recall our very first program:

Code:

```
jmp start;

  //Area 1

start:
```

```
  mov eax,0;
  add eax,10;
  add  esp,2;

  jmp end;

  //Area 2

end:
```

You can place your subroutines either in area 1 or area 2. Pascal programmers will probably love area 1 better where as C/C++ programmers will prefer area 2. Example Let's put our addstuff example to the main program. Putting it into area 1 will look like this:

Code:

```
jmp start;

addstuff:
  push ebx; //Save ebx, because we use it. We don't save EAX, because we retu

  mov eax,esp;
  add eax,4;     //2 (our pushed eax and ebx) + 1 (return address) + 2 (the f
  mov eax,ss:#eax; //Read from stack

  mov ebx,esp;
  add ebx,3; //2 (our pushed eax and ebx) + 1 (return address) + 1 (the secon
  mov ebx,ss:#ebx; //Read from stack

  add eax,ebx; //Result will be returned in EAX

  pop ebx;
  ret;

start:
  push 123;       //First param
  push 456;       //Secod param
  call addstuff;
  jmp end;

  //Data or stuff here

end:
```

Or alternatively in the second area like this:

Code:

```
jmp start;

  //Data and stuff here

start:
  push 123;       //First param
  push 456;       //Secod param
  call addstuff;
  jmp end;

addstuff:
  push ebx; //Save ebx, because we use it. We don't save EAX, because we retu

  mov eax,esp;
  add eax,4;     //2 (our pushed eax and ebx) + 1 (return address) + 2 (the f
  mov eax,ss:#eax; //Read from stack

  mov ebx,esp;
  add ebx,3; //2 (our pushed eax and ebx) + 1 (return address) + 1 (the secon
  mov ebx,ss:#ebx; //Read from stack

  add eax,ebx; //Result will be returned in EAX

  pop ebx;
  ret;

end:
```

Or you can put the subroutine in both areas. Whichever you prefer. Don't forget to giv comments too!

**Closing**

OK, I think that's all for now. By this time, you should be able to program a lot of assembly with ease. Why don't you try to code your favourite subroutines just for exercise?

*Other and new chapters will be posted as I write/rework them*
*I ported this from x86 tutorial, so if you find any misleads/errors, report them to me please. Thanks.*



*Last edited by Black Phoenix; 03-20-2008 at 01:49 PM.*