If this is your first visit, be sure to check out the **FAQ** by clicking the link above. You may have to **register** before you can post: click the register link above to proceed. To start viewing messages, select the forum that you want to visit from the selection below.

If you'd like to receive bonuses in the Official Wiremod Gmod Server, Link your Steam account to you forum account **here!**

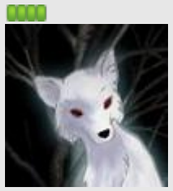
Thread: The "Foxy" CPU.

09-28-2008 #1

-=Fox=-

Wire Sofaking

WIREMOD MEMBER



Join Date: Feb 2007
Location: Somewhere in my Mind...
Posts: 1,864
Blog Entries: 7

The "Foxy" CPU.

Introduction

I've seen a few tutorials on how the CPU works and what to do with it. Black Pheonix's Documentations and ASM tutorial are enough for the tech and code savvy, but may fall short for "laymen" such as myself.

I had a heck of a hard time learning the CPU, but I found out it's not terribly difficult once you understand How and Why something works.

So from here, I'll be giving out tid-bits of information and explanations for what I currently understand about the CPU, how I thought about it and how I got it to make sense to me!

What is zASM or more specifically, what is Assembly in General?

An interesting description follows:
"a group of machine parts that fit together to form a self-contained unit"

This is a mechanical description, but as we soon find out this can also be applied to zASM Programming!

Just like Spark Plugs, Pistons, and a Crank Shaft make up an Engine, sections of code work together to produce an effect, just as an engine performs work with fuel, programs perform work with data.

Now I know a lot of coders and ASM people would have me go through the "History of Assembly and why stuff is called what it is." but for the purposes of just learning the material, I'm going to spare the reader the Historic stuff.

Section I
(syntax overview, basic mathematics & simple I/O (input/output))

One of the most useful of zASM (and NASM) commands is "mov", a mnemoic (an equivalent name) for "Move", it illustrates the syntax of a lot of other instructions as well and how ASM works with some data.

Now, before you can "mov" stuff around we have to have a means for moving stuff around! Let's say we have a

couple of buckets, lets call one bucket "eax" and another bucket "ebx".

These "Buckets" eax and ebx (among others which we will get to) are called Registers, they hold data much the same as a bucket holds water.

So this begs the question, "How do I do that?!"

Syntax:

Here is a example of moving a value around, consider the following instruction:

mov eax,1
[Instruction (Destination),(Source)]

The first instruction is "mov", it tells the CPU that were going to move something. The second part is "eax" (also known as a paremeter), this is the Register we are going to move something into (ie. pour the water into). The 1 is what we are going to put into the register.

So what does it do?
"Move into eax, 1" Kind of like Yoda speak.
It takes a 1 and sticks it into the "bucket" called eax.

Remember Syntax is like this:
[Instruction (Destination),(Source)]

The exception is for single parameter Instructions like jmp (Jump) and other instructions too!

Special Notes for "mov"
When using Mov from source to destination the source is NOT changed. For example

mov eax,ebx

The contents of ebx are NOT changed, but the contents of eax are replaced by what was in ebx. So you can still use the contents of ebx later on.

Of course this isn't doing a whole lot, or really doing anything useful at this point, we need to add something to this to make it do something useful.

Enter the **Simple Math** Mnemonics!

add (Add)
sub (Subtract)
mul (Multiply)
div (Divide)

Let's say we want to make a program that will calculate the percentage of a given value. For this Program we need to gather a formula such as.

Value/Whole = Percent

So for this program we need to have a dividing part, some means to hold the values and a means to display the results!

First things first we need to understand the we need to tell the CPU how to do everything... EVERYTHING. There is very little the CPU can do on its own (unlike the E-gate).

So we need a means to pull the data into registers, perform the calculation, then shove it to an output. You'll need a CPU and a Data port to use the following code.

->> More information on the Data Port can be found in Section IX. <<-

```
mov ebx,300
in eax,1
div eax,ebx
```

```
out 1,eax
```

Let's go though this line by line.

mov ebx,300
This takes the value of 300 and puts it into the register called ebx. This is what we will be the "Whole" part of the formula.

in eax,1
The "in" instruction tells the CPU to grab the number that is given (wired to) to the data ports, port1. Remember; **Instruction Destination,Source**. This is the "Value" part of the formula.

div eax,ebx
The "div" instruction is divide, and were taking ebx (which is currently 300) and dividing it with eax (whatever is wired into port1). This is the dividing function.

out 1,eax
The "out" instruction tells the CPU to put the value into the output port1 on the data port. It will put the contents of eax there (the result of what div did)

You can use any of the basic math instructions this way.

If you want to try something a little more complex you can make a program that will take more than one value as a parameter and then produce an output.

Examples:

```
in eax,1
in ebx,2
add eax,ebx
out 3,eax
```

This will take input from port1 and port2 of the Data Port and add them together, then output the result to output port3.

Input/Output
(I/O notes, in & out detailed)

The "in" instruction tells the CPU that it is going to get information from a port. That instruction has 2 parameters that it needs.

1) What port to use.
2) Where to put the information!

```
in eax,0
```

Takes information from the Dataport (wired to the I/O Bus of the CPU) and puts it into the register called "eax", from there you can do calculations and anything else you want to do with it.

The "out" instruction does the exact opposite

```
out 0,eax
```

Takes whatever is in the eax register an puts it on port 0 of the dataports output.

Important Notes

When using math instructions that take more than one parameter (like add,sub,mul,div) the destination register is where the result is stored!

```
sub eax,ebx
and
mul eax,10
```

Both the results are stored in eax, so when you output (out) the results don't forget to:

out 1,eax
[Instruction **Destination**,Source]
"Output to port 1, the contents of eax"

As always, experiment!

Port
(Another Method of using Ports)

The "port" instruction is another way to use ports, it is a simplified version of using "in/out" and codewise is smaller (therefore more efficient). The rest of the tutorial uses in/out because at the time of writing port did not work.

Of course, now it does. So I'll teach you how to use this one as well!

Moving Data in from port0 (port number 0) into eax;

mov eax,port0

Moving Data out from port0 from eax;

mov port0,eax
You may be wondering, "Wait.. how does that work?". If you remember above we talked about the syntax;

mnemonic **destination**,***source***
[Instruction **TO**,**FROM**]

When we put port0 in the area for destination, that's where the data is going to go **TO**. Likewise if it is in the source area thats where data is going to be pulled **FROM**.

->> More information on the Data Port (uses In/Out and Ports) can be found in Section IX. <<-

Section II
(Basic looping, Jumps and Labels)

Now we get to the fun part!

Remember the Engine analogy I used above? What does an engine do when you spark it and give it fuel? It shoves pistons down and turns the crankshaft. It makes a loop, as long as it has fuel it will keep going and going, over and over repeating the same action.

With that repetition it can be used to do lots of things, like move your car, a train or even generate electricity. Programs are a lot like that, as long as the CPU has data to work with it will keep going if you tell it to.

The trick is, to get it to do something useful. Like an engine, you have to give it a frame, some other stuff (transmission, axles etc.) and wheels to make a car.

Let's take a look at a program we had for an example above.

```
in eax,1
in ebx,2
add eax,ebx
out 3,eax
```

This program is designed to do one thing, take two inputs, add them and output the results. But what will happen if you just run the program like this?

It will only run once! D'oh! So what we need to do is give it some spark, a means to repeat the action over and over, so that when you change the inputs to add different numbers together, it can do that, instead of adding the numbers that are there, only once.

First things first, in order to do a loop, the CPU needs to know WHERE to loop to! Remember you have to tell the CPU how to do just about everything.

Introducing, Labels!
(Labels and Comments)

For the remainder of the parts in this tutorial, we're going to be using comments, a LOT. So here's how they are used!

mov eax,ebx //This is a comment! Comments start with "//" it tells the CPU "Don't
//process me, I'm just here for information for the humans."
//You can devote entire lines to Comments if you want to..
//Just be careful as some text editors may wrap your text and the CPU
//may see a word as a command and throw you an error!
// <-- so make sure that those are there!

Labels
(Labels)

Labels are a way to give the CPU "bookmarks" or "reminders". Take our example program. Let's add a label, we'll call it "Main" cause it will be used for the main part of the program. But before we go anywhere else we need to add comments to our code so we know and others know how the program works.

Note: Try not to make labels that look like instructions, such as In, Out, Cmp and stuff like that. I break that rule all the time for simplicity, but it is best to avoid doing that as it can cause confusion!

Code:
```
Main: //This is the Label, remember Label: <-- with a colon, it can
            //be anything, even Kittens: Poop: and L33T:
in eax,1 //This takes the Data port1 and shoves that value into eax
in ebx,2 //Takes Data port2 and shove that to ebx
add eax,ebx //Adds eax & ebx
out 3,eax //puts the value of eax into the output port3 on the Data Port
```

There we go, now we run our program... wait... were missing something...

Jmp
(Jumps!)

Code:
```
Main:
in eax,1
in ebx,2
add eax,ebx
out 3,eax
jmp Main // There we go! (you don't need a colon here, just the name of the label)
```

The jmp (Jump) instruction takes one parameter to tell the CPU, WHERE to jump to!

In the highlighted example above "jmp Main" tells the CPU, that after it executes all the rest of the instructions above, that it must jump back to the label called "Main:".

Now our program has legs, it can now loop! So as you change the input of ports 1 and 2, the output will change on port3 of the Data Port!

Section III

Compares and Conditional Jumps
(compare and other jump commands)

The CPU is quick at doing math, but it can also do "non-mathy-stuffs", it can also compare values and make a special jump called a "Conditional Jump".

Now, why would we want to use Conditional Jumps? Well, let's find an example where you can use one!

As always, make a plan for what kind of program you want to make, write down formulas you need and what the program is designed to do. For this example were going make a program that reacts to button presses!

The general behavior of the program is to give an output (in this case 1) when the button is pressed! It will need to loop so that if we let go of the button or the CPU encounters a zero (0), it will keep running the program to wait for another button press.

To do this we need a new instruction:

cmp (compare)

The cmp command makes comparisons and sets a flag in the CPU when a compare is true or false. (I'm not sure what flag it sets though... but you can still use it without knowing it, Need maor info!)

To use cmp you need to give it a pair of items to compare to and then an action to follow it up.

cmp eax,10

This will compare the contents of eax with the value 10. But since the cmp is useless on its own, we need a Conditional Jump to make it work.

cmp eax,10
je Main

This example contains a Condition Jump! In this case "je" which means simply "**J**ump if **E**qual". There is also a companion jump to go with that one

cmp eax,10
jne Main

Jne means "**J**ump if **N**ot **E**qual", we will use one of the jumps in the next example.

Code:
```
Main: //<-- don't forget a label!
in eax,0      //Port0 also works for Data Ports, this takes in values at port0
cmp eax,10 //Compare the contents of eax (port0) with the value 10
je Equal     //Jump to Equal if condition is met to the label called "Equal"
jmp Main   //In the even that the compare yeilds a "False" result, do this
             //instead. In this case jmp to Main
Equal:     //The Equal Label!
out 0,1    //puts a 1 on the output port0 of the data port
jmp Main  //Jump back to Main after doing the above instruction
```

As you can tell this program has multiple jumps! Remember after making a jump to another label (Equal in this case) you need to jump back to Main so that the program loops, if you don't jump back to Main the program will make the jump when the conditions are met, but it won't jump back!

For an example this program will only run once:

Code:
```
Main:
in eax,0
cmp eax,100
je Same  //Will jump to the label "Same" when eax = 100
```

```
jmp Main //This is like "otherwise", if the cmp fails, it will simply goto the
           //NEXT instruction and continue processing more instructions
Same:
out 1,eax
//CPU goes "Looks like I'm done here." and quits.
```

As you can see, once it jumps to "Same:" it doesn't have another jump to make, so the program simply stops. So in order to keep the program looping you need to add another jump, in this case, back to Main. ^_^

Lets try a more complex example that uses both "je" and "jne".

Code:

```
Main:
in eax,0 //In to eax from port0
cmp eax,9001 //compare to eax 9001... ITS OVER NINE THOUSAAAAND! :D
je OvrNineThou
jne Fail //You'll notice we don't use jmp after this, can you think of why?

OvrNineThou:
out 0,eax //when eax is 9001 it will jump to here where it is outputted!
jmp Main //Don't for get to Jump BACK!

Fail:
out 0,0 //Outputs a zero to port0 in the event of Fail
jmp Main // <-- yup!
```

There are several types of Conditional Jumps and they are in the documentation including "Jump if Less Than or Equal" and "Jump if Greater Than", but for now we won't cover those specifically.

Section IV

At this point, you already know what you need to in order to make basic programs that will do work for you. You can stop, right now, and go play with the CPU.

If you want to know some more advanced items, then by all means, continue!

Macros
(Data; and Code;)

Macros (as we work with them here for now) are things that the CPU's compiler deals with. The compiler turns the code you write into "Machine Code", basically a bunch of numbers nobody but BP can understand (:lol.

They are **Optional, not required for programs to work.**

I'll introduce you to Two Macros that are important if your going to do more advanced tasks, such as Allocations, Definitions, the DB macro and working with memory.

Lets start with these Two macros:

Data;

Code;

Note that these items have **Semi-Colons** they are not labels, the CPU doesn't work with them directly, but the underlying code that is generated by these macros are!

You don't have to worry about how they work, they simply make organization easier and they look pretty. In order to use them effectively you need something for them to work with.

Lets move on to...

Variables and Allocations
(alloc and define)

alloc (Allocate, means to "set aside"/"make room for") and define are two VERY important instructions that allow you to use Variables and to have room to put extra variables in case you run out of Registers!

Let's work with **define** first. There are a few more things to define, but for now we'll cover the basics. Let's also use the Macros we talked about above.

To use define simply tell it what to define.

define Cats,9001
[Instruction Name,Value]

Code:

```
Data; //This section contains anything you need to store, I use it mainly
        //for variables and allocated values, you can even put code up here!
define Val,100 //This creates a variable named "Val" he equals 100.

Code; //Code usually starts here, so put your instructions AFTER this macro

Main:          //This is the label, Required if you use jumps!
in eax,0       //Let's say we put in a 10 here, eax will equal that
cmp eax,Val //The compare will fail! Since 10 doesn't equal 100
je Win         //Since the compare failed it moves on past this one
jmp Main     //this jump is not conditional, so it jumps back to Main.

Win:          //BUT! if we get a 100 (the value of Val)
out 0,9001 //Then it outputs 9001 to the output port0 on the data port
jmp Main    //Jump back to Main when finished!
```

Define is the equivalent to a constant, it gives you a solid value that cannot be changed. This is good for use with things like memory offsets and locations of highspeed device memory. (a bit later on)

The instruction alloc is used to store values (or can contain an existing changeable value). It is allocated internal memory. For this exercise we're just going to allocate single values.

The best way to think about Memory (in the case of the CPU) is to think of them as "Bins", "cubby holes", or even prison "cells" that you can stuff things into. These "cells" can hold only a single number, a very large or a very small number, but only one number at a time.

To use alloc (singally, for this example)

alloc Stuf
[instruction parameter]

You can also use alloc to assign an initial value that is modifiable (unlike define).

alloc Stuf,9001
[Instruction parameter,value]

So how do we use this? Let's say we've run out of Registers to hold and manipulate values (egad!), that our program requires a TON of variables.

To spare you the agony I'll pretend that I can only use eax, and that's it. Let's do a subtracting program this time. I'm going to work with two variables, with one Register.

Code:

```
Data;
alloc temp // this is the allocation, it makes room for Temp and defaults to 0
Code;

Main:
in eax,1            // grab a number from port1
mov #temp,eax //store the value in allocated value called temp
in eax,0           // grab number from port0
sub eax,#temp //Subtract Temp from eax
out 0,eax        //send the result to output port0!
jmp Main         //loopy loopy!
```

Pretty neat eh?
"Woah... hey, what the heck is this -> # ????"

That my friend is called a hash, what it does is it tells the CPU that we are referring to the **Memory Location** of the allocated value. It tells the CPU, "Hey... this is pointing to where this value is in memory.".

This is because alloc allocates Memory "Cells" to use, so in order for the CPU to know what "Cell" to manipulate, we have to point to it. For example, if we **alloc temp** again and do this...

mov temp,eax

What will happen is that eax will equal the value of the allocated memory cell (the Cell number), instead of what that memory cell **contains**! It won't store the value, so watch out!

The correct way to do that is to:

mov #temp,eax

This will shove the contents of eax into the memory cell of temp.

Section V
Memory Works
(Reading From & Writing To Memory)

Memory manipulation and use is one of the more geeky things you can do with the CPU. In order to work with Highspeed devices (like the Console) you need to know how to work with memory.

In the previous section, I introduced you to the Hash (#), it tells the CPU that whatever follows it, refers to a memory location. The instruction alloc uses memory, hence why we have to use #.

Writing
(Shoving stuff into memory)

For the Example here we're going to use a program, that takes in values from a ranger and stores them. We won't do anything like flow control or data rates, or even reading (yet), so you don't have to worry about that for now.

Code:
```
Data;

define off,5000 //here I picked the 5000th memory cell (CPU has 64,000+)
                //This is away from both the code and stack so we don't                    //Get Errors.
Code;
Setup: //You can run code outside of a loop!
 add eax,off //off refers to the constant 5000, eax now = 5000

Write:
 in ebx,0 //Connect a distance ranger to port0
 mov #eax,ebx //The number 5000 is now used as a memory location
 inc #eax //New instruction! inc (increment) adds one to its parameter (#eax)
jmp Write
```

There, as you might have noticed, all this program does is shove stuff into memory from a ranger. The code is pretty simple so we can go thorugh it line by line.

in ebx,0
Takes whatever is on the data ports port0 and puts it in the register ebx.

mov #eax,ebx

Mov will take what is in ebx (port0) and put it into the MEMORY location (using #) of what eax equals (5000), so it shoves wherever is in ebx to the memory cell which is 5000.

inc #eax
This is a new instruction were working with, in this case inc or **Inc**rement. In this case were taking a memory location (refered to by a register, eax) and incrementing it by 1. That's it!

jmp Write
As you might have noticed we jumped back to Write instead of Setup! We wanted to setup eax so that it was ready for the next part of the program where we use it (and change it!) later on.

If we used **mov eax,off** at the start of the loops, then the **inc** instruction would only work once, rather than continiously. By jumping to Write, we skip that instruction so that when we loop, the inc instruction will keep incrementing the memory location.

So that as the data comes in from the ranger, it will always have the next available empty cell to store information. Remember Cells can only hold ONE number at a time!

Reading
(Pulling stuff out and viewing it!)

Reading from memory is almost identical to putting stuff into memory, for such devices as the Keyboard or the Wired Socket Radio (among others) in order to read values from them, you need to access the memory (which is why Highspeed devices are so fast!).

In fact, let's take the code from above and tweak it so that it does the exact opposite!

(for the assumption that there is actually data to read)

Code:
```
Data;

define off,5000

Code;
Setup:
 add eax,off

Read: //use labels that are descriptive!
 out ebx,0 //Connect a screen Output port0
 mov ebx,#eax //this one is inverted switching ebx and #eax
 inc #eax
jmp Read
```

Now you might look at that and go, "Well that was easy...". That's because it is! The tricky part is making buttons that will switch between both reading and writing.

So, that's exactly what were going to do!

But just before we do that were going to go over "inc" a little bit more.

Increment & Decrement
(inc and dec instructions and equivalent functions)

The inc instruction is arguably one of the most useful instructions out there. I've been using the E-Gate for a while, and about 70% of the time I usually had an incrementing function in the code.

Whether your writing or reading from cells, an incrementing function is very important. It's like wheels on a car, it allows your program to move through data, cell by cell (or skipping cells even!). Or if you've got a math function that must be repeated over and over. There are many uses for inc, even as a sort of "switch" to use as a variable in other parts of your program.

The inc instruction is one that may cause a little confusion, since it can work on registers and memory locations, like mov. But has completely different results for what you use it on. For example:

mov eax,10 //eax is now 10
inc eax

eax will equal 11, it's the same as; add eax,1

mov eax,10
inc #eax

Use this bit of code will make eax point to memory cell #10 and increment the value in **memory** instead of the register itself! So keep an eye on that they are both valid uses.

You can also use inc as a counter, to keep track of how many times some bit of code is executed, or even make a delay or timer.

For examples:

Looperz:
inc eax //if registers aren't defined, they start at 0
jmp Looperz

This will keep incrementing eax forever and ever (or until the CPU gets a number it can't handle.)

Code:

```
Main:
 out 0,0 //Make sure port0 is zero!
 inc eax //Increment eax by 1
 cmp eax,1000 //compare eax to 1000
je Boom //jump if equal to "Boom"
jmp Main //if compare fails, jump back to Main and try again.
  //Remember, eax will keep its value throughout the program
  //until you reset it or put something else in eax.

Boom:
 out 0,1 //put a one on the output port0 of the data port
 mov eax,0 //reset eax to 0 so it can do it all over again!
jmp Main //Jump Back to Main
```

This one will increment eax until it hits 1000, then output a 1 to the dataports port0, reset and do it again.

Of course you can use an incrementing **function** (a bit of code that does something similar) without using inc directly. As if I can confuse you more  .

For example if you need a program to count by 2's or even 10's you can do that too. The inc instruction only increments by one (afaik so far) and that's a limitation that can get in the way in some cases.

Main:
add eax,10
jmp Main

This will add 10 to eax every time the program loops, so it will increment eax by 10 each time, although it doesn't use inc, it has the same effect. As you'll find out, there are (like the E-Gate) multiple solutions to any given problem.

Decrement

As with a lot of instructions inc has a companion instruction "dec" or **Dec**rement, which is the exact opposite of inc!

mov eax,100

Main:
dec eax

jmp Main

As you can guess this program will Decrement eax by one each time the program loops... of course since the CPU can handle negative numbers in cells... it'll just keep going :lol:

So you'll need to compare and reset it, with something like this;

Code:

```
 mov eax,100 //put 100 into eax, keep out of loop so this isn't repeated.

Main:
 dec eax   //Decrement eax by 1
 cmp eax,0 //compare to 0
je Reset //go to reset if zero
jmp Main //loop if not zero

Reset:
 mov eax,0 //reset eax to 0
jmp Main //Return to Main
```

Just like that!

And now we return you to our regularly scheduled programming!

Code Example #1
(Concept Overview)

As always, make a plan for the program, even if it's small. Write down what you need, collect formulas/information and the steps it may take to do the job. Remember, you have to tell the CPU how to do everything.

Were going to make a program that will, with a pair of buttons "record" data then "play" it back on a screen. It will do this as fast as the CPU can handle so we get Lots of "Hi Resolution" data!

1. We will need a part of the program to watch the buttons for changes.
2. Records Data to memory (Input on the Data Port0)
3. Reads Data from memory (Output on the Data Port1)
(using different ports to avoid confusion)
4. Returns to watch for the buttons so that it doesn't keep recording when we don't want it to.

This is the most complex code in this tutorial up to this point, so be SURE you have read and have a good grasp of the concepts. Preferably having tried and done the above examples yourself.

Code:

```
Btn1: //Check button #1
 in eax,0 //In from Port0
 mov ecx,3000 //set/reset ecx
 cmp eax,1 //is button 1 on?
je Write //if so jump to Write Label
jne Btn2 //goto check button #2

Btn2: //Check button #2
 in eax,1 //In from Port1
 cmp eax,1 //is button 2 on?
je Read //if Yes jump to Read Label
jne Btn1 //if no buttons are not pressed, go back to checking buttons.

Write: //Write to Memory
 in eax,0   //double check the button
 cmp eax,1 //make sure the Write button is on
jne Btn1  //If the button is not on, don't record, go check buttons.
 in ebx,3 //Connect data to record to port3
 mov #ecx,ebx //move data from port to Memory cell
 inc ecx //move to next memory cell
jmp Write //loop to keep writing to memory!

Read: //Read from Memory
 in eax,1 //Check this Button
 cmp eax,1 //Make sure it's on
jne Btn1 //If not, go check the buttons
```

```
 out 0,#ecx //using ecx as a memory location
 inc ecx //move to next cell to read
jmp Read //loop to keep reading from memory!
```

Whew!!!

That might be a little overwhelming, but that's what it takes, so when I say you have to tell the CPU how to do everything... I mean it...

As you can see the program has 4 obvious parts to it, like an engine all the parts have to work well in order for the engine to run. Your first useful program might not be as straight forward as the one above, it does take making several programs in order to really get the hang of it.

But practice makes perfect! So don't give up!

Section VI
(The dreaded Stack of DOOM...)

For me, the first time I heard "The Stack" it sounded intimidating, like this monster thing that would be really hard to work with, but as it turns out... it's really really simple.

The Stack has been compared to many many times to a stack of plates. More accurately, if you ever been to a buffet restaurant, where you can grab a plate and each as much as you like, pay attention next time to how the plates are held in place.

Usually it's a hole in a counter just big enough for the plates, on the bottom is a spring and a small platform. As plates are stacked on top of it, it sinks down making room for more plates so they don't stack too high and fall over. As you pull a plate off the top, the platform rises slightly so that the next plate takes its place.

The stack is like that, it follows whats called "LIFO" or **L**ast **I**n **F**irst **O**ut. So like a stack of plates (or a stack of anything), in order to get to the bottom of the stack, you need to take off what's on top. Or if your stacking items, the stack grows from the bottom up, in the reverse order you put them there.

The Stack is a special type of storage system, when combined with Calls (later) it can be used to pass parameters in some uses of the Stack and Call, we can do what's called a "C-Style Calling Convention". But we'll get to that later!

For now, a code example doesn't work well here (yet) so let's go over what the stack is made of and what uses it. The Stack is a section of memory that is easy to access, and can be used to store values.

It does have some oddities to it though.

The Stack really only has one rule that must be followed. You must pop off the same amount of values you push on to it. Programs that use the stack must maintain a "balance" pop'ing off all the values that were push'ed on BEFORE you make a loop. Otherwise we can get odd results, wrong results or even the dreaded *STACK OVERFLOW*. Eeek!

So how does it work?!?!

The Stack
(ESP Register and The Stack)

Let's go over the Stack Pointer, the register ESP is used as the stack pointer, its value is the memory location of the next available memory cell. So if you push 3 items onto the stack and fill 3 memory cells ESP points the the 4th memory cell, the one that isn't occupied.

Lets say we have a stack that can hold 3 numbers, normally the stack is clear, so it doesn't contain anything.

[Cell10] - 0 <-(ESP)
[Cell09] - 0
[Cell08] - 0
[Cell07] - 0

Now let's **PUSH** stuff into the stack, say... the numbers 1,2 and 3, in that order.

[Cell10] - 1
[Cell09] - 2
[Cell08] - 3
[Cell07] - 0 <-(ESP)

The number 1 was pushed onto the stack first so it's on the "bottom", followed by 2 then 3.

That action is equivalent to this zASM code;

mov eax,1
push eax
mov eax,2
push eax
mov eax,3
push eax

Now let's say we want to take something off of the stack, we have to **POP** it off. So if have our stack like above and we pop a number off here is what we get.

[Cell10] - 1
[Cell09] - 2
[Cell08] - 3 <-(ESP)
[Cell07] - 0

Just like that!

That in code would be (for popping one value);

pop eax //eax equals 3 the value that was last pushed on!

If we pop another value from the same stack with:

pop eax //using the same register again will OVERWRITE it!

We will have:

[Cell10] - 1
[Cell09] - 2 <-(ESP)
[Cell08] - 3
[Cell07] - 0

So then eax will equal 2 while esp will equal the number of the cell it's pointing to (in this example Cell #9).

Unfortunately we can't get back to Cell 8 where it contains the number 3, by trying to push something again. It will simply overwrite that cell. Pushing a number like 4 again should leave us with this:

[Cell10] - 1
[Cell09] - 4
[Cell08] - 3 <-(ESP)
[Cell07] - 0

Because doing pop eax from there would be:

[Cell10] - 1
[Cell09] - 4 <-(ESP)
[Cell08] - 3
[Cell07] - 0

So eax would equal 4 instead of 3!

push - Write to ESP, then move to next cell.
pop - Move to previous cell, then Read from ESP

So at this point, you may be DYING to know, how the heck do we get to that 3 (assuming it's still there)!

Since we've already popped it before hand, the only way to get back to it would be to modify the value of esp... however... this would be beyond the scope of this section of the tutorial. Messing with ESP and SS can (if not done correctly) result in broken programs and utter failure as the CPU gives you errors.

So let's take a look at some code examples on how to use the Stack so that we don't have problems with it.

This bit of code is actually from my Port Reporter, it takes an incoming value, rounds, and does modulus to it and shoves the results to allocated values.

Basically takes a 3-digit number and splits it up for displaying as a string to the console screen.

It uses the stack effectively.
It also contains math instructions that I haven't mentioned, but are in the documentation. For now, I won't go over them other than whats in the comments.

Code:

```
fint #eax //fint floors values, in this case a number stored in #eax
push #eax //Saves eax the stack
div #eax,100  // eax/100
fint #eax     //Floor #eax, stores in #eax <- that's IMPORTANT TO KNOW!
mov #hun,#eax //Put the results in #hun
pop #eax      //Pop eax from the stack, gives or origional number back!
mod #eax,100  //Apply different math (modulus) to #eax
push #eax     //Save the results!
div #eax,10   // eax/10
fint #eax     //Floor eax to give a whole number
mov #ten,#eax //Store the results in the #ten for Tens
pop #eax      //Restore number saved earlier to apply different math to it
mod #eax,10   // eax%10 this time (modulus)
mov #one,#eax //Save to #one for ones!
```

As you'll notice it has an even number of push and pop instructions. This is important! If you don't do this, you have the potential to have a stack overflow/underflow.

Also you'll notice that the entire procedure is done in memory as I didn't have enough registers to use individually for the whole program so I just stuck with eax and supported it with memory and the stack.

Section VII
(Calls & Conditional Calls)

Calls are just like they sound, they're calls! But to what!? Why other code bits of course! Calls are like jumps, the difference is, unlike a jump, they return to the place in code where you made the call from! Instead of going through the code and having to be looped again.

To use a call simply type the instruction and the label of the function you want to call.

call Label

This is like dialing the telephone, it tells the CPU to go to "Label" and start executing code from there. Unfortunately like a call, the CPU doesn't know how to hang up the phone automatically, so you have to tell the CPU to "hang up" with this one instruction.

ret

This is called **Ret**urn, it will "end the phone call" and tell the CPU to finish where it left off!

Every call made must be ended so the CPU can continue what it was doing before, so after ever call, must be a ret.

So how do we make calls? Like this:

Main:
in eax,0 //in from port0
in ebx,1 //in from port1
call Add //call label Add
out 3,eax //CPU returns here after the call is finished!
jmp Main //jumps back to main

Add:
add eax,ebx
ret

You may be thinking to yourself that calls are too similar to jmps, why not just keep using jumps? That's an excellent question! One reason would be modularity.

Calls allow you to "modularize" your program, like Legos, you can put pieces together to form a whole. Let's take for example a very simple calculator program, one with only add and subtract.

We'll be using regular calls and a "Conditional Call" called "ce" or **C**all if **E**qual, like jmps calls can use results from a compare (cmp).

Code:

```
Data;
alloc Param1 //Port1
alloc Param2 //Port2

Code;
Main:
 mov #Param1,0 //clear both alloc'd values
 mov #Param2,0
 in #Param1,1 //save parameter 1
 in #Param2,2 //save parameter 2
 in eax,0 //we'll use this as the port (0) to tell what function to do!
 cmp eax,0
je Main //jmp to main if port 0 is 0, so we can change params while running
jne Cmpr //when eax is non-zero we'll goto Cmpr

Cmpr: //an abreviation of "compare"
 cmp eax,1
ce Add //call Add if we get a 1 on eax
 cmp eax,2
ce Sub //call Sub if we get a 2 on eax
jmp Main

Add:
 add #Param1,#Param2 //Remember math functions save result in 1st parameter of the instruction.
call Out
ret

Sub:
 sub #Param1,#Param2
call Out
ret
```

Now this program is pretty useful... but to be really useful as a general calculator it needs the basic 4 functions... so far we only have 2! So we need to "ADDZ MAOR FUNKSHUNS!" So let's "plug in" **mul** (Multiply) and **div** (Divide)!

Since the program is modularized we only need to add a pair of code bits that will do our bidding and then just tweak the code slightly.

Code:

```
Data;
alloc Param1 //Port1
alloc Param2 //Port2

Code;
Main: //We don't have to change any of this!
 mov #Param1,0 //clear both alloc'd values
 mov #Param2,0
 in #Param1,1
 in #Param2,2
```

```
 in eax,0
 cmp eax,0
je Main
jne Cmpr

Cmpr: //We'll need to adjust this section
 cmp eax,1
ce Add
 cmp eax,2
ce Sub
 cmp eax,3
ce Mul  //add a call for the Mul function
 cmp eax,4
ce Div  //add a call for the Div function, easy!
jmp Main

Add: //no change here!
 add #Param1,#Param2 //Remember math functions save result in 1st parameter of the instruction.
call out
ret
```

Not Bad! And you can keep adding functions to this code, by adjusting the Cmpr section and then simply plugging in the new code! Don't you just love modularity?

Calls and The Stack
(How they can work together)

The Stack and Calls can also work together, the Stack can pass information from one call to another, for example if you need more than one math function in your code and you need to send the results to the other function you can do that with the stack.

Using calls and the stack together you can actually free up the registers you need in order to reuse them in another part of the code. So if you run out of registers, you can push them (saves their values) before making a call. Thereby freeing up the registers so you can use them up again.

But... be sure you POP them back before you end the call! Remember all pushes to the stack MUST have corresponding pops within calls!

Using a calling convention it is possible to completely isolate segments of code! Isolating the registers the call uses so the whatever they were used for earlier won't interfere with the called code.

A Calling Convention is a method of doing all that. There are a variety of calling conventions that you can use, some conventions adjust the ESP value directly, some work with ESP and EBP and some with both ESP/EBP and/or offsets. It can be quite confusing.

*****Important!*****
To add insult to injury, when making a call, the stack is being used to save the memory location of where it left off. So **the return address for the call will be on TOP of ths stack!**

That introduces another problem. Pushing values, making the call, then poping the first value for use simply won't work... **because you'll pop the location of where the CPU needs to return to** and if you don't put that back on the stack... the CPU freaks out because it doesn't know where to go back to!

So for now, we'll start with the simplest convention I know. It'll become clearer as we go through the code.
Let's try something simple. A little math and moving values around.

Code:

```
Caller: //this is a method of doing multiple calls, kind of a "caller registry".
 call Input
 call Maths
 call Save
 call Output
jmp Caller

Input: //Let's grab some inputs, using ports instead of "in"!
 mov eax,port0
```

```
 mov ebx,port1
ret //remember the rets!

Maths:
 add eax,ebx
ret

Save: //This is TRICKY! using the Stack to save within calls
 pop ecx //Save the return address somewhere!!
 push eax //Save eax, it has the result of the maths
 push ecx //Put the return address BACK on the TOP stack... LIFO LIFO LIFO!
ret //BEFORE YOU DO A RET!

Output: //Same procedure to pull values
 pop ecx //SAVE the return address (anywhere works)
 pop eax //pull the data you need
 mov port2,eax //output data (do whatever you want to data)
 push ecx //When your done put the return address back on top of the stack!
ret //If you do it wrong it won't return to "Caller" but FAIL!
```

As you can see, the calling convention is very very simple, but when done right, it should work! Just remember the procedure!

- Make the call
- Save Return address (first item on the stack after a call)
- Do stuffs
- Put the return address back
- Do ret

Another use for the Stack is freeing up registers, so let's get another example where we have used up "all" of the registers and we need to free them up in order to do some stuff.

Let's pretend that there are only 4 registers. In this program, we'll take inputs, report the states of the ports AND output an average.

Code:

```
Code;
 alloc Savstk \\allocate a place to save stack return addresses
 alloc Avg \\save the resulting math function here

Data;
Input:
 mov eax,port0
 mov ebx,port1
 mov ecx,port2
 mov edx,port3
call Maths
jmp Input

Maths: //"all" the registers are "in use" so we need to save them!
 pop #Savstk //Save the return address FIRST! This is a call!
 push eax
 push ebx
 push ecx //save the values at the registers...
 push edx //we'll use this data later.
 add eax,ebx
 add ecx,edx //The registers dont' hold origional data...
 add eax,ecx //good thing we saved them!
 div eax,4 //averaging, Add values, divide by number of values.
 mov #Avg,eax //save the result!
jmp Output

Output: //Remember, we still have a ret to do!
 pop edx //Remember the order that you pushed values onto the stack...
 pop ecx //when you pop them, they'll be in reverse order...
 pop ebx //so pay attention to that!
 pop eax //Now we have restored the register states!
 mov port0,eax //Output the port states
```

There we go, a pair of examples on how to do stack stuffs, using a very simple calling convention. As long as your return address is on top of the stack before doing a ret, everything should be fine!

The rest of the tutorial can be found here!

http://www.wiremod.com/forum/cpu-tut...-part-2-a.html

*Last edited by -=Fox=-; 12-27-2008 at 05:46 PM. **Reason:** My speling sucks. So I are ficksing it.*

Share

http://tiny.cc/OMFGWTFBBQ

Best People On Wiremod!

Black Phoenix, Azrael, **Jat Goodwin**, Magos Mechanicus, ITSBTH, Fizyk, g33v3s,**tuusita**, InfectiousFight, ief015

Pointless things that are pointless, are pointlessly pointless, therefore pointlessness is pointless.
So pointlessly pointing out the pointlessness of this pointless signature is utterly pointless.
My IQ is 123

Reply With Quote

09-28-2008 #2

ebs

Wire Amateur

WIREMOD MEMBER

Join Date: Apr 2007
Location: Denmark
Posts: 61

Re: The "Foxy" CPU.

Very nice tut. I like how you explain every thing, by thing.
But "in" and "out" can also be set to 0.. Like.

Code:

```
mov eax,100;
out 0,eax;
```

or

Code:

```
in eax,0;
```

Share

Reply With Quote

09-28-2008 #3

GUN

Wire Sofaking

WIREMOD MEMBER

Join Date: Jan 2008
Location: California
Posts: 669

Re: The "Foxy" CPU.

wow that made it somehow simpler to understand, i like how you went line by line and a full description of it all, very nice and now i want more. i want to be able to use all advanced wire components, i have egate 1 and 2 down so far and with more of these tuts maby cpu soon

Share

Maybe we're all gonna die, but we're gonna die in *really cool ways*.

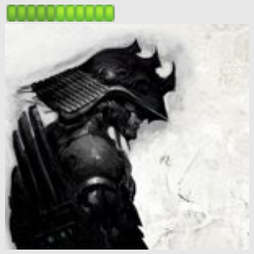
Reply With Quote

09-29-2008 #4

Jat Goodwin

Official Bastard of Wire

WIREMOD MEMBER



Join Date: Aug 2008
Location: Colorado Springs
Posts: 2,768

Re: The "Foxy" CPU.

That was amazing, you have a gift for teaching. I have never learned CPUs or attempted to learn it, but I understood that perfectly and only had to read through it once.

Share

Official Bastard of Wire

I Require More Minions! Join us on the IRC !
List of Reasons to idle on the IRC: Wire QDB

Reply With Quote

09-29-2008 #5

Black Phoenix

That furred thing

WIREMOD WM DEVELOPER

Re: The "Foxy" CPU.

By the way, the only actual exceptions from the (dest)(source) are xchg (dest,dest), mcopy/mxchg (source,source), callf/jmpf (dest,dest), lidtr (source), spp/cpp/ and some others (dest,dest).

JMP uses first operand/parameter as jump destanation

Great tutorial, I'll probably sticky it along with CCFreak2K's tutorial. Or maybe subforum for CPU tutorials and references? Because there's plenty of them, and we dont want to clutter up the stickies

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

Share

I'm a wire-crazy person with a tail.

Take a **daily** journey into my brain

D2K5

Reply With Quote

09-29-2008 #6

Bean

Wirererer

Join Date: Feb 2008
Posts: 118

Re: The "Foxy" CPU.

also, Percent is out of 100... so you'd need to
mul eax,100;
before outputting right?

Share

My color is Blue

I value knowledge, logic, and deceit. I love to pursue wisdom but also to manipulate and deceive. At my best, I am brilliant and progressive. At my worst, I am treacherous and cold. My symbol is a water droplet. My enemies are green and red.

Take the Magic: The Gathering 'What Color Are You?' Quiz.

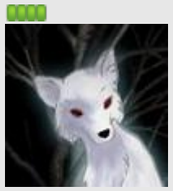
Reply With Quote

09-29-2008 #7

-=Fox=-

Wire Sofaking

WIREMOD MEMBER



Join Date: Feb 2007
Location: Somewhere in my Mind...
Posts: 1,864
Blog Entries: 7

Re: The "Foxy" CPU.

Awesome comments :lol:

I actually teach people how to use computers, relatively complex devices made simple. It's all about relating to the material and how it's "phrased" the way I teach (or try to teach) is a naturalistic approach where one idea flows into another.

BP, Thanks! I knew there was other instructions that used that, I just want to start with the simple stuff that I understand, so that I can completely solidify what I know, so this helps me as well!

Bean, your right! If we had 100 parts and we have 50 of them, then yes it would be 50%, but if I have 300 parts, and I take 150 of them, I also have 50% ^_^

ebs, your also right in and out can be used for numbers 0-(however many ports you have wired up) but for 1 data port you can use 0-7. The above is an example.

Share

http://tiny.cc/OMFGWTFBBQ

Best People On Wiremod!

Black Phoenix, Azrael, **Jat Goodwin**, Magos Mechanicus, ITSBTH, Fizyk, g33v3s,**tuusita**, InfectiousFight, ief015

Pointless things that are pointless, are pointlessly pointless, therefore pointlessness is pointless.
So pointlessly pointing out the pointlessness of this pointless signature is utterly pointless.
My IQ is 123

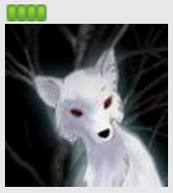
Reply With Quote

09-29-2008 #8

-=Fox=-

Wire Sofaking

WIREMOD MEMBER



Join Date: Feb 2007
Location: Somewhere in my Mind...
Posts: 1,864
Blog Entries: 7

Re: The "Foxy" CPU.

Updated with spelling corrections, Moar content and... Codez Boxzes!

^_^

Share

http://tiny.cc/OMFGWTFBBQ

Best People On Wiremod!

Black Phoenix, Azrael, **Jat Goodwin**, Magos Mechanicus, ITSBTH, Fizyk, g33v3s,**tuusita**, InfectiousFight, ief015

Pointless things that are pointless, are pointlessly pointless, therefore pointlessness is pointless.
So pointlessly pointing out the pointlessness of this pointless signature is utterly pointless.
My IQ is 123

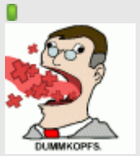
Reply With Quote

09-30-2008 #9

Free Fall

Wirererer

WIREMOD MEMBER



Join Date: Dec 2007
Location: Got digitalized and now lives in his PC's RAM
Posts: 349

Re: The "Foxy" CPU.

Wow, these tutorials are very easy to understand (I already understood BP's tutorials, but that just a side node). Pretty good work there. Keep it up!

What we need now is..... a easy section about Interrupts!

BP: I agree with your idea, to make a subforum for tutorials. Then you got it a bit more ordered, and you don't have to sticky pages of tutorials ^_^

Share

Needz moar Lua

Reply With Quote

09-30-2008 #10

Bean

Wirererer

Join Date: Feb 2008
Posts: 118

Re: The "Foxy" CPU.

I second the interupts

Share

My color is Blue

I value knowledge, logic, and deceit. I love to pursue wisdom but also to manipulate and deceive. At my best, I am brilliant and progressive. At my worst, I am treacherous and cold. My symbol is a water droplet. My enemies are green and red.

Take the Magic: The Gathering 'What Color Are You?' Quiz.

Reply With Quote

Page 1 of 13 1 2 3 11 ... Last

« Previous Thread | Next Thread »

Similar Threads

The "Foxy" CPU - Part 2 By -=Fox=- in forum CPU Tutorials	Replies: 40 Last Post: 06-27-2010, 03:00 PM
Want to understand "interval" and "timer" in Expression 2 By anthraxyhe in forum Installation and Malfunctions Support	Replies: 25 Last Post: 12-08-2009, 09:36 AM
Advanced Duplicator doesn't save Radio's "Values" setting By TomyLobo in forum Bug Reports Archive	Replies: 0 Last Post: 02-14-2009, 06:13 AM
Xtensity's "Spaceman Turret V6" Tutorial, "Head Shots FTW"	Replies: 24

By Xtensity in forum Gate Nostalgia (Old School Wiring) Discussion & Help
Last Post: 09-20-2008,

Need the pack that has the "Generator" and "Razor" models...
By Mr. Brightside in forum Installation and Malfunctions Support
Replies: 2
Last Post: 06-06-2007,

Bookmarks

Digg
del.icio.us
StumbleUpon
Google
Facebook

Posting Permissions

You may not post new threads
You may not post replies
You may not post attachments
You may not edit your posts

BB code is On
Smilies are On
[IMG] code is On
[VIDEO] code is On
HTML code is Off
Trackbacks are On

Pingbacks are On
Refbacks are On

Forum Rules