Creating your own Abstract Processor

© Aodrulez.

Introduction

In this paper, I'll try to explain to you how to create your own Abstract Processor. It can be really simple or absolutely complex & it'll depend entirely on your imagination. First of all, let me explain what is an Abstract Processor. Its nothing but a purely theoretical processor architecture that one can develop by programming at software level. If that sounded too complex, think of it like creating your own processor by writing code in lets say c/c++ or PERL with registers, stack-size etc that you define. Don't worry much if you didn't get the concept yet. Am sure you will grasp it on the way.

Less theory, more code.

Alrighty! I hate theory as much as you do. So lets get started with a practical example that'll help you understand this better. Lets say, that you want to create your own Abstract Processor. The first thing that you'll need to decide might be its name. :)
Yea.. thats right! Its your own processor so think of a cool name for it. Once you've nailed that down, the next thing would be to decide what goes into it & its specifications. So here, as an example lets design an Abstract Processor named '**Aod**'. Now that we've decided on a name, lets decide the specifications.

Specification

Processor name	**Aod**
Registers	IP,SP,X,FLAG
Stack	256 bytes.

Thats one really simple design. Let me explain each & every part of it. Every processor needs to have registers. Some of these might be special & for internal use only while others might be general purpose ones that we can let the programmers use to write software for our architecture. That apart, a processor needs to have some amount of memory, to which it can read & write data. (RAM?). Thats the most basic necessity of every design. You can add more to this if you want & if you are really adventurous, remove some of these.
The beauty of writing an abstract processor is that you dont have to worry about the internal electronics stuff like how will the Data-Bus or the Address-Bus work, how the I/O pins will work etc. You can create any design that you fancy.

Explanation of each part

IP = Instruction Pointer.

Every processor needs to keep a track of the code it has to execute. Thus every processor needs to have this register compulsorily. Here, this register will point to the next instruction that the processor has to execute.

SP = Stack Pointer.

Just like the instruction pointer, we need to have a stack pointer that'll point to some location on the stack. This is where we read/write data.

X = General Purpose Register.

Well, this register is provided to the programmer to store data temporarily. Its like variables inside a processor that can store any value. Although this is not really compulsory, its good to have these.

FLAG = Internal register.

This one is a special register. Its used internally to store certain data like if there was a carry when a subtraction operation was carried out.. etc. We'll dig deeper into this later.

STACK = Read/Write Memory.

Well, this is like your computers' RAM. Here, we can store data temporarily. The processor can read as well as write data to any location on the stack. We've decided to have a stack size of 256 bytes.

Yea.. thats all! Thats your processor's specification. Ain't that easy? But..we still have some more thinking to do. :) We hav'nt yet decided the Instruction-Set for our processor, have we? In the next section, we'll decide upon a simple Instruction-Set for our '**Aod**' processor.

Instruction-Set

The fundamental function of a processor is to execute code, right? For that, it needs to have a set of instructions to follow using which we can program it to execute some task for us. This unique set of instructions for a particular processor architecture is popularly called as its Instruction-Set. So, lets design our own instruction set.

OpCode	Instruction	Description
0x01	input	Takes one char as input & saves it on the STACK where SP is pointing to.
0x02	output	Prints the character from the stack where SP is pointing to.
0x03	mov sp,x	SP=X
0x04	mov x,data	X=byte following the OpCode.
0x05	cmp x,data	FLAG=X-byte following the OpCode.
0x06	je data	If FLAG==0, IP=IP+byte following OpCode.
0x07	inc x	X=X+1
0X08	inc sp	SP=SP+1
0x09	mov [sp],x	Write the value in X to the location on STACK specified by SP
0x0A	halt	End execution

This is just a small set that I could think of right now, enough to give you a working demo of the idea. A good processor will need some more instructions or maybe I should say, a clever Instruction-Set. This is where your creativity will help you. Now, lemme explain the concept of OpCode. It stands for 'Operation Code'. On second thoughts, lemme give a full-blown explanation of the internal working of a processor.

Some interesting stuff

See, the processor needs code so that it can start executing them. 'Code' is nothing but a sequence of 1s & 0s & so is data. To help you visualize better, think of a straight line of 1s & 0s & lets assume that this is the code that has to be executed. Ah yes, one more thing.. '**Units of Data**'. I know it'll sound stupid as most of you already know this, but what exactly are the units of data? Data, like any other quantity needs to be measured. We measure **Mass** in terms of grams & Kilograms. Similarly, even Data has its own units. They are :

1. Bits == Binary Digits. (Either '1' or '0' since its binary)
2. Nibble == 4 Bits.
3. Byte == 8 Bits or 2 Nibbles.
4. Kilobyte == 1024 Bytes.

And so on..

So lets say, I have 1 byte worth of data. How many bits are there in all?
1 Byte == 8 Bits.

Cool, now lets think of some basic Mathematics. When I say Binary, its to the 'Base 2'. This is one of the Numbering systems. Similarly there are others like Decimal (which we usually use. Its Base 10.), Hexadecimal (Base 16) & so on. I find it easier to remember & understand this 'Base' term by thinking of it this way.

Binary == Base 2 == Maximum number of unique digits == 2. (1 & 0)
Decimal == Base 10 == Maximum number of unique digits == 10. (0 to 9)
Hexadecimal == Base 16 == Max. unique digits == 16. (0 to 9 & A to F)

Lets think of Decimal system.
Q. What is the Biggest 2 digit number that I can have?
A. 99. Because in decimal, the biggest digit is '9' & when we are talking about 2 digits, it has to be 99.

Q. In Binary, what is the Biggest 8 digit number?
A. Yea.. you guessed it right. It is **11111111**.

With this in mind, lets try some conversion. Kick up your calculator & convert the above Binary number **11111111** to Decimal. It turns out to be **255** in Decimal. (Note : This is Conversion. If you find the calculator too complex, try some online conversion tool.) I hope you still remember that **1 Byte == 8 Bits.**

So tell me, what is the biggest **Byte** of data you can have?
Its **255 in Decimal** or **11111111 in Binary** or, if you convert it to Hex, it'll be **0xFF in Hexadecimal.**

People often get confused between **Data** & **Numbering Systems.** I'll give you a simple example. Lets say, I want to measure a length of **5mm** of a piece of cloth. Its exactly the same as saying **0.5cm** of cloth or **0.005 meters** of cloth. The length is still the same. Am just expressing it in different units of Length.

In the same way, Data can be expressed in multiple **Units** as well as **Numbering Systems**. What am trying to explain is this:

```
1 Byte of Data. == up to 11111111 in Binary
                == up to 255 in Decimal
                == up to 0xFF in Hex.
```

The quantity of data remains the same. We are simply expressing it in multiple numbering systems. Phew! I hope you are not confused now. :)

Lets again come back to the initial sequence of 1s & 0s the processor has to execute. I hope you understand now that here, we are talking in terms of Binary. So, let me put it in a different way, the processor has a continuous stream of Binary data that its supposed to execute. It'll be tedious for us Humans to understand & work with Binary data, right? So, what we do is.. we take the initial 8 Bits (1 Byte) & convert it into the corresponding Hexadecimal Number. We term this number as the OpCode. Its a special sequence of 1s & 0s that instructs the processor to carry out some task. So, instead of working with the Binary Numbers, we usually convert these bytes to Hexadecimal numbers (its still the same.) & assign an instruction to it.

Now, again lookup the **Instruction-Set** table we've designed. Am sure, the opcodes section makes more sense now. The next section will cover programming of the processor.

Aod Processor implemented in PERL

```
--------------------------------------- snip here ---------------------------------------
#!/usr/bin/perl
use strict;
use warnings;
my $IP=0;
my $SP=0;
my $FLAG=0;
my $X=0;
my @STACK=(0);
my @ROM=(0x04,0x41,0x09,0x02,0xA);
my $opcode=0;
while($IP>=0)
{
        $opcode=$ROM[$IP];
        if($opcode == 0x01){$STACK[$SP]=ord(getc());$IP++;}
        if($opcode == 0x02){printf("%c",$STACK[$SP]);$IP++;}
        if($opcode == 0x03){$SP=$X;$IP++;}
        if($opcode == 0x04){$X=$ROM[++$IP];$IP++;}
        if($opcode == 0x05){$FLAG=$X-$ROM[++$IP];$IP++;}
        if($opcode == 0x06){if($FLAG==0){$IP=$IP+$ROM[++$IP];}else{$IP=$IP+2;}}
        if($opcode == 0x07){$X++;$IP++;}
        if($opcode == 0x08){$SP++;$IP++;}
        if($opcode == 0x09){$STACK[$SP]=$X;$IP++;}
        if($opcode == 0x0A){print"\nHalt.\n";exit(1);}
}
--------------------------------------- snip here ---------------------------------------
```

Yea! Its as simple as that! The above code is a fully functional **Aod** Processor along with a dummy ROM (code to execute). Lets analyze the above @ROM array.

```
@ROM[0] == 0x04
@ROM[1] == 0x41
@ROM[2] == 0x09
@ROM[3] == 0x02
@ROM[4] == 0x0A
```

Lets try to understand what it does. :)

Disassembly (Refer the Instruction-Set Table)

@ROM[0] == 0x04	(mov x,data)
@ROM[1] == 0x41	(since the previous command was **mov x,data ,** this byte is the data provided. So skipping this byte.)
@ROM[2] == 0x09	(mov [sp],x)
@ROM[3] == 0x02	(output)
@ROM[4] == 0x0A	(halt)

Lets see exactly what should happen here by manually analyzing every single instruction.

Manual Walk-through.

IP=0, SP=0, X=0, FLAG=0, STACK[SP]=0
0x04 mov x,data (data == the following byte & hence data=0x41)

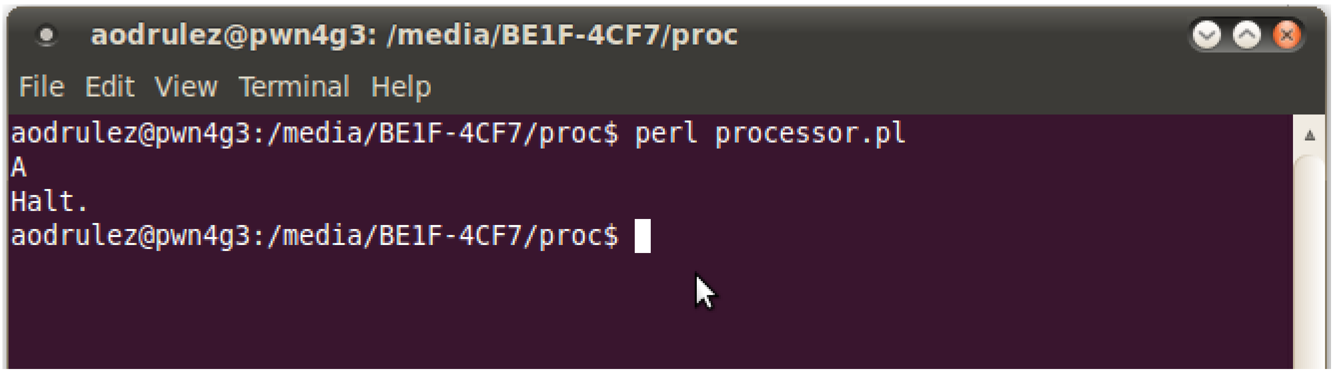
IP=2, SP=0, X=0x41, FLAG=0, STACK[SP]=0
0x09 mov [sp],x

IP=3, SP=0, X=0x41, FLAG=0, STACK[SP]=0x41
0x02 output (Should print the character at STACK[SP] which is 0x41= 'A')

IP=4, SP=0, X=0x41, FLAG=0, STACK[SP]=0x41
0x0A halt (End execution.)

Expected output:
A
Halt.

This is how your abstract processor or any processor for that matter, works at the lowest possible level. A screenshot of the above PERL implementation with the output from our dummy code :


aodrulez@pwn4g3: /media/BE1F-4CF7/proc
File Edit View Terminal Help

```
aodrulez@pwn4g3:/media/BE1F-4CF7/proc$ perl processor.pl
A
Halt.
aodrulez@pwn4g3:/media/BE1F-4CF7/proc$ 
```

More Sample Code to try out. :)

1.
```
@ROM=(0x04,0x41,0x09,0x02,0x07,0x09,0x02,0x07,0x09,0x02,0x07,
      0x09,0x02,0x07,0x09,0x02,0x0A);
```

2.
```
@ROM=(0x04,69,0x09,0x02,0x04,110,0x09,0x02,
      0x04,116,0x09,0x02,0x04,101,0x09,0x02,
      0x04,114,0x09,0x02,0x04,32,0x09,0x02,
      0x04,97,0x09,0x02,0x04,32,0x09,0x02,
      0x04,67,0x09,0x02,0x04,104,0x09,0x02,
      0x04,97,0x09,0x02,0x04,114,0x09,0x02,
      0x04,32,0x09,0x02,0x04,58,0x09,0x02,
      0x04,32,0x09,0x02,0x04,50,0x03, 0x01,
      0x04,2,0x03,0x04,89,0x09,0x02,0x04,111,
      0x09,0x02,0x04,117,0x09,0x02,0x04,32,
      0x09,0x02,0x04,84,0x09,0x02,0x04,121,
      0x09,0x02,0x04,112,0x09,0x02,0x04,101,
      0x09,0x02,0x04,100,0x09,0x02,0x04,32,
      0x09,0x02,0x04,58,0x09,0x02,0x04,32,0x09,
      0x02,0x04,50,0x03,0x02,0x0A);
```

Conclusion

This research paper was born out of a personal project for an International Malware Conference's online challenge that I was designing. Initially, it was just a thought & took hours of hard work to understand every aspect of implementing such a theory at a very practical level. But once I got it working, I got completely hooked onto it & within a span of weeks learnt more about computers & their underlying theory than I did in all these years, combined. This is like learning Programming, Reverse-Engineering & 'how to write a compiler, debugger' all rolled into one. Not to forget that understanding this clearly will mean that one can Reverse-Engineer code belonging to any Platform or Architecture. This is a Universal approach to the Art of Reversing.

(**Note** : This is a highly simplified example to help you understand the core concepts.)

Practical Implementations

- 'Virtual-Machine Theory' thats used in many copy-protection Mechanisms.
- As an Anti-Debugging technique.
- As a portable Architecture.

Sky is the limit if you are creative.

Proof Of Concept

Am working on an Abstract Design that I've named as **Aod8**. It'll be up on Github pretty soon, so stay tuned. Its part of **MalCon-2011** '**Capture the Mal Challenge**' & I'll be uploading all the tools, code, easter-eggs etc which includes a full-blown 3-pass Assembler, a usable debugger & the source-code of the CTM challenge.

References

- Online Javascript based Emulators for various Architectures.
- Esoteric Languages like Brainfuck

Greetz Fly out to :

- TheBlueGenius.
- Amforked()
- ne0 & TCC.
- Orchidseven.com
- Malcon.org
- Shantanu Gawde for Beta-Testing the CTM challenge. :)
- The legendary +ORC

Point your Flames/Suggestions/Greetz to :

- Email : atul.alex@orchidseven.com
- Email : f3arm3d3ar@gmail.com
- Blog : http://aodrulez.blogspot.com
- Twitter : http://twitter.com/Aodrulez

Have a Wonderful Day!