Understanding Code

by Kwazy Webbit

Introduction

There is no single layer of abstraction to executable code. It has many, the lowest of which is binary.
It is important to understand that binary can represent anything. Executable code and data are, at the lowest level, the exact same thing: a collection of 1’s and 0’s. You can try to run data as code, but most likely that will just cause a crash. Trying to use executable code as, for example, picture data will also be either invalid or, at best, just random. This is because there is a structure to both of them which allows them to become more than just binary. To be useful, you need something that understands this structure and will interpret it in the proper way.
As a more concrete (non-binary) example, one could have four numbers:

112, 43, 149, 184

They could mean pretty much anything. If I were to tell you it was a line for example, one could imagine it being a line in 2 dimensions, starting at coordinates (112, 43), and ending in (149, 184). However, if I were to tell you it was a square, you could think of it as a square with those coordinates as top-left and bottom-right points.
It can be anything, it all depends on your interpretation. The problem is, how do we make sense of it all? How does the computer know what to do with what? How can WE know what it really does? In this essay, I will not go into understanding data, since data structures are too diverse (think alone of the image formats you’ve seen). Each file format has a different structure. Programs use the file extension as a hint for the structure to expect.
Instead, I will focus on executable code, specifically that for the x86 processor. I will start at binary, and eventually end up in C.

* I chose the C programming language because it is very close to regular mathematical notation, and simple expressions are fairly easy to read even for non-programmers. If you do have trouble understanding C, there is a plethora of information to be found on the web.
Binary to Hexadecimal

As mentioned before, the lowest level of information (in a computing environment) is binary. Code, as the computer sees it, is an endless row of 1’s and 0’s. It is nearly impossible for humans to follow what is happening by seeing it. If you are interested in how the circuits in your CPU work, I suggest getting some electronics books. I do not know enough about it to explain in detail how it works (though I have seen it work on much simpler processors). For purposes of explanation, binary is a clumsy format, as the amount of binary digits is too large for us to easily oversee. That is why we never normally edit anything in binary, but instead go to a direct translation, known as hexadecimal format. It is just a numbering format. Just as a numerical representation of binary has two digits: 0 and 1, and decimal has 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), hexadecimal has 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. You may wonder why this format is chosen over the decimal system which we are all used to working with, and is thus much more intuitive for us humans. The answer is simple. It is because underneath it all the numbers are still binary, being translated. Using 4 bits at a time you can make exactly 16 different values, from 0 to 15. In hexadecimal, that is from 0 to F. This makes it a very practical system to write down 4 bits with one character. I’ve included a quick lookup table, if you aren’t familiar with it already.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>F</td>
</tr>
<tr>
<td>10000</td>
<td>16</td>
<td>10</td>
</tr>
</tbody>
</table>

As you can see, at binary 10000, the hexadecimal value (10) still makes sense, since the first digit (1) can still represent 0001, while the second digit (0) represents the 0000, resulting in 00010000, which matches the binary. The decimal however is now 16, which is not as obvious to convert anymore.
Hexadecimal to Assembly Code

Using hexadecimal notation we have a shorthand for writing down binary code, giving us more overview. It still makes no sense to a human though, because essentially it is still just a lot of numbers. Following is a sample of code in hexadecimal format:

83EC20535657FF158C40400033DBA39881400053

As said, this is just a shorthand for the binary digits it represents. That means it doesn’t give any explanation to what it does, but it is a lot shorter than the binary representation: The hexadecimal representation is 40 characters, while the binary would be 160 (Since every hexadecimal digit represents 4 bits)

The code above is not one big instruction*. It is, in reality, several small ones. On some processors, every instruction has a certain size (e.g. 2 bytes) so you can easily chop code up into parts of that size to get the different instructions (assuming you have a valid starting position). The x86 processor is a little more complex (it is a CISC† architecture after all), and has different sized instructions. You might now wonder how we can ever split up different instructions this way. The idea is, you take the first byte, look at its value, and that byte will tell you how to proceed. Several things could happen:

- It could be a single byte instruction: e.g. 90h‡ is the ‘NOP’ instruction (No OPeration) and is only 1 byte in size.
- The instruction is not yet complete: e.g. Instructions starting with 0Fh need more bytes to fully define their function.
- The instruction is defined by a single byte, but it needs parameters: e.g. 8Bh moves one register into another. The byte following 8Bh will describe where it moves from, and where it moves to.
- The instruction is not yet complete and it needs parameters.

Because we will need to know what an instruction is in order to split them up anyway, we will combine the process of splitting up the different instructions with translating them to a human-readable equivalent. This ‘human-readable equivalent’ is known as ‘assembly language’, often abbreviated to ASM. The process of translating a program from raw code to ASM, is known as ‘disassembling’ (lit. ‘taking apart’). It takes some skill to read ASM. However, since every instruction in ASM performs a fairly trivial task (even on a

---

* The term ‘instruction’ here refers to the actual code bytes, while the term ‘operation’ refers to the task that instruction performs.
† Complex Instruction Set Computer, that means it has a lot of different instructions which do a lot of different detailed things. Its counterpart is the RISC (Reduced Instruction Set Computer) where only a few instructions exist, doing simple tasks. This allows lower complexity, and single instructions are generally executed faster. However, since less is done in a single instruction, it has been long debated which solution is better.
‡ Hexadecimal values are generally indicated by appending a ‘h’. In C notation, they are represented by preceding the value with ‘0x’, for example ‘0x90’ means 90h.
CISC processor), they are easy to understand by themselves. It takes some experience however, to keep an overview of what non-trivial action is being done, more on this later. First, we are going to look at the separate instructions.

Since there is no clear system to see what operation a hexadecimal code performs (it is basically a matter of looking it up in a reference, and writing it down), it is a rather tedious job. However, as it is important to understand how this works, I will demonstrate using the example above.

Let’s take another look at the hexadecimal code:

83EC20535657FF158C40400033DBA39881400053

We will assume the first byte is a valid starting point (and not halfway through an instruction, because this would ruin our disassembly process*) and go from there. We take the first byte, which is 83h, and we’ll take a manual to look it up. I used the table in Appendix A1† to look it up. This says it requires another byte to describe the full operation, and that this byte should be in the form of a ‘mod R/M’ byte. To see what the full operation is to use the information from this byte and look under “group #1” in Appendix A2. In this case, the byte is ECh. A mod R/M byte consist of 3 bitfields:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meaning</td>
<td>mod</td>
<td>reg</td>
<td>R/M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To separate these bitfields, we have to go back to binary, which gives:

EC = 1110 1100 = 11 101 100

Using Appendix A2 we see that for the bitfields matching xx101xxx, the operation is SUB. The other two bitfields describe the first operand of the SUB operation. Looking at Appendix B, we find that 11 means that it uses a register directly, and 100 means that that register is ESP. Using the original description of Appendix A1, we have one more operand to fill, the ‘Ib’ (Input byte). Quite simply, the next byte is to be used, which is the 3rd byte (20h).

Putting all this together, we find the first ASM instruction:

83EC20 SUB ESP, 20

Which lets us continue to decode the next instruction (starting with 53h).

Let’s do one more. Looking up 53h (in Appendix A1) shows it is a single byte instruction with no parameters:

PUSH rBX (= PUSH EBX)

---

* Some programs purposely use this ‘starting point’-problem to confuse disassemblers, to prevent outsiders looking at how their program works. This technique is known as ‘obfuscation’.
† The tables in Appendix A are taken from http://www.sandpile.org
So now we have translated the first 4 bytes into their ASM equivalents:

```
83EC20 SUB ESP, 20
53 PUSH EBX
```

As you probably realized by now, disassembling takes quite long to do manually. Luckily, there are plenty of ready made tools (aptly called ‘disassemblers’) to perform this process for us (e.g. HIEW).

Using HIEW, the hexadecimal example is translated to ASM as follows:

```
83EC20                       sub         esp,020
53                           push        ebx
56                           push        esi
57                           push        edi
FF158C404000                 call        d,[0040408C]
33DB                         xor         ebx,ebx
A398814000                   mov         [00408198],eax
53                           push        ebx
```

Some programs are a bit more clever though, trying to understand the flow of a program. For example, it could look at the addresses used and see which ones point to a string, or analyze the flow of the program (following jumps). These more advanced disassemblers include IDA and WDasm. Using IDA, the most advanced disassembler available at the time of writing, the result is:

```
sub     esp, 20h
push    ebx
push    esi
push    edi
call    ds:GetProcessHeap
xor     ebx, ebx
mov     hHeap, eax
push    ebx             ; lpModuleName
```

As you can see, IDA has done some more analysis. Here, it has figured out where the call is going, and it understands that the return value from that Windows function (GetProcessHeap) is a handle to a heap, so it has appropriately renamed the variable to hHeap. In this example there was little IDA could do, but usually it gives quite a lot more information than a less specialized program such as HIEW.

These advanced features save us a lot of work looking into everything manually, and gives a good starting point for further analysis of the program. In ASM, we can see what the code is doing one little step at a time, but to make this useful we need a bigger picture. A higher level of abstraction minimizes explanations of HOW things are happening, thus leaving the focus on WHAT is happening. A language like C gives us that overview.
Assembly code to C

Now that we have the ASM code, it is understandable for humans what the program is doing. However, since every ASM instruction only performs a trivial task, it is hard to see what non-trivial function a program is performing. Let’s see an ASM listing as created by HIEW:

```
push ebp
mov ebp,esp
sub esp,048 ;"H"
push ebx
push esi
push edi
mov d,[ebp][-08],000000000 ;"
jmp .00041230B  

mov eax,[ebp][-08]
add eax,001 ;"☺
mov [ebp][-08],eax
mov eax,[ebp][08]
push eax
call lstrlenA ;KERNEL32.dll
```

As you can see, ASM uses a lot of simple instructions to work together and ultimately perform a useful task. We’ll start at the first instruction and work down, trying to keep an overview of what is going on, using a ‘Pseudo-C’ notation, and eventually translating to proper C code.

```
Asm to C:

push ebp
mov ebp,esp
sub esp,048 ;"H"
push ebx
push esi
push edi
mov d,[ebp][-08],000000000 ;"
```

As you can see, ASM uses a lot of simple instructions to work together and ultimately perform a useful task. We’ll start at the first instruction and work down, trying to keep an overview of what is going on, using a ‘Pseudo-C’ notation, and eventually translating to proper C code.

---

* Called ‘Pseudo-C’ because, even though it follows the general structure of C in terms of operators, it is a literal translation of the ASM code, and thus still uses registers directly.
Here are the first few lines:

```
.004122F0: 55                           push        ebp
.004122F1: 8BEC                         mov         ebp,esp
.004122F3: 83EC48                       sub         esp,048 ;"H"
.004122F6: 53                           push        ebx
.004122F7: 56                           push        esi
.004122F8: 57                           push        edi
```

The first two operations create what is known as a ‘stackframe’. This is essentially a ‘local’ stack inside the function, where extra room can be reserved for local variables. This is done simply by lowering the stack pointer a bit further, for as many bytes as are necessary for the local variables.

One of the main advantages of a stackframe is that the EBP register can be used as a fixed point to reference variables (above EBP are the parameters, and below it are the local variables).

Note that the stackpointers (ESP and EBP) have to be restored before leaving the function, to avoid stack corruption.

```
.004122F0: 55                           push        ebp
.004122F1: 8BEC                         mov         ebp,esp
.004122F3: 83EC48                       sub         esp,048 ;"H"
```

This code creates such a stack frame, and creates 48h bytes of space for local variables.

Windows requires that a few registers (besides ESP and EBP) are preserved during a callback function, namely EBX, ESI, and EDI. These are stored safely on the (local) stack, ready to be restored right before leaving the function. This allows for free use of these registers while inside the function.

It seems most practical to look at the last few instructions, since we now already know several tasks that need to be performed there. And looking, we indeed find exactly what we expected:

```
.00412348: 5F                           pop         edi
.00412349: 5E                           pop         esi
.0041234A: 5B                           pop         ebx
.0041234B: 8BE5                         mov         esp,ebp
.0041234D: 5D                           pop         ebp
.0041234E: C3                           retn
```

First, the 3 registers are restored from our (local) stack. Then the stack is restored to its state when the function was called, and the function returns. Note that we could not have restored the stack before the 3 registers, because the registers were stored on our local stack. Converting all this to C is very easy. We know now that it is probably a function, because of it’s stackframe, storing/restoring registers, as well as its retn (return from function) at the end:

```c
void SomeFunction()
{
    //...code...
}
```
I’ve assumed for now that this is a void function, because there is no change in EAX before the return. This does not mean EAX is never changed. But for now, we will assume the value in EAX is ignored.

Now we will proceed further into the body of this function:

```assembly
C745F800000000    mov    d,[ebp][-08],000000000 ;"EBP
00412300: EB09 jmps .00041230B  -----↓  (1)
00412302: 8B45F8 mov    eax,[ebp][-08]
00412305: 83C001 add    eax,001 ;"O*
00412308: 8945F8 mov    [ebp][-08],eax
0041230B: 8B4508 mov    eax,[ebp][08]
0041230E: 50 push    eax
0041230F: FF1584A34300 call    lstrlenA ;KERNEL32.dll
00412315: 3945F8 cmp    [ebp][-08],eax
00412318: 7D2E jge     .000412348  -----↓  (2)
```

We notice a value being referenced a lot:

d,[ebp][-08] == dword ptr[ebp-08] (in another notation)

Since it is below our EBP it is on our local stack, so the function is storing a local variable there. We know that it is DWORD size (because it’s being read using a dword ptr), and that it’s probably a signed value (because it’s being compared to the result of lstrlenA, which is a signed int). On the win32 platform, the standard signed dword size value in C is the (signed) int. Let’s rename it to int_local1 for easier reading (I also removed the hexadecimal representation of the code, and the less helpful comments):

```assembly
mov    int_local1, 000000000
jmps   .00041230B  -----↓  (1)
mov    eax, int_local1
add    eax,001
mov    int_local1, eax
mov    eax,[ebp][08]
push   eax
call   lstrlenA ;KERNEL32.dll
cmp    [ebp][-08],eax
jge     .000412348  -----↓  (2)
```

Be careful here. Do not confuse [ebp][08] with [ebp][-08]. Even though they look alike, they are different addresses. The variable at [ebp][08] is *always* (assuming a normal stackframe) the first parameter passed to our function. We will thus (for the time being) rename that value to dw_param1. Now that we have identified a local variable, and cleaned things up a little, we will make a start at converting to Pseudo-C:

```
int_local1 = 0;
goto label_41230B;
eax = int_local1;
eax = eax + 1;
int_local1 = eax;
label_41230B:
eax = dw_param1;
eax = lstrlenA(eax); //lstrlenA returns its result in eax
```
Rather strange looking code, but it’s a start. Let’s be a little bit less literal about it, and use our brain. Looking at the 3 lines:

```c
eax = int_local1;
eax = eax + 1;
int_local1 = eax;
```

We see that it is really a very simple instruction, that could be simplified into a mere:

```c
int_local1++;  
```

The only difference between these two representations though, is that EAX no longer has the same value after the new representation. We should take care in doing so, because the value in EAX might be used afterwards.

In this case, the next line is:

```c
eax = dw_param1;
```

Which means we can freely replace the instruction, since EAX gets overwritten before being read anyway. The next part:

```c
eax = dw_param1;
eax = lstrlenA(eax);  // lstrlenA returns its result in eax
if( int_local1 >= eax)
    goto label_412348;
```

could also be made a lot easier to view, because you can combine a lot of instructions in C. We can do it as follows:

```c
if( int_local1 >= lstrlenA(dw_param1) )
    goto label_412348;
```

Again, we should now look at if EAX is used afterwards, so that we don’t miss another location where this value was being used. On the very next line, however, EAX is overwritten, so we are free to make this change. Because we know lstrlenA is expecting a pointer to a string, we will rename the parameter now to pString to represent this, giving us a total code of:

```c
int_local1 = 0;
goto label_41230B;
int_local1++;  
label_41230B:
    if( int_local1 >= lstrlenA(pString))
        goto label_412348;
```

Looking for further references to this section, we find the line

```iasm
.00412346: EBBA                         jmps       .000412302  -----↑ (3)
```
jumping to the line with `int_local1++;`, which makes it all appear to be big loop. If you’re familiar with C coding, you might already have figured out what structure we are looking at here. It seems to have the functionality of a ‘for’-loop. Let’s try to make a for-loop that mimics this behavior (while also renaming `int_local1` to `i`). Rewriting the C code, we end up with:

```c
for(i = 0; i < lstrlenA(pString); i++)
{
    //...rest of code...
}
```

It’s slowly beginning to make sense. We now know it is a function that goes through a for-loop, ranging from 0 to the length of the string it gets as its (first) parameter. Now we need to know what it does inside the loop.

The code was:

```assembly
 .0041231A: mov         eax, pString
 .0041231D: add         eax, i
 .00412320: mov         cl, [eax]
 .00412322: mov         [ebp][-01], cl
 .00412325: movzx       eax, b, [ebp][-01]
 .00412329: cmp         eax, 061 ; "a"
 .0041232C: jl         .000412346  -----↓ (1)
 .0041232E: movzx       eax, b, [ebp][-01]
 .00412332: cmp         eax, 07A ; "z"
 .00412335: jg         .000412346  -----↓ (2)
 .00412337: movzx       eax, b, [ebp][-01]
 .0041233B: sub         eax, 020 ; " 
 .0041233E: mov         ecx, pString
 .00412341: add         ecx, i
 .00412344: mov         [ecx], al
```

Here, we find another local variable in use. It appears to be of the `unsigned char` type, because it is byte size (referenced by using a byte ptr), and is used as unsigned (by the `movzx` instructions). In Pseudo-C, we now have:

```pseudo-c
 eax = pString;
eax = eax + i;
cl = *(eax);
ch_local2 = cl;

eax = (DWORD) ch_local2;
if(eax < 0x61) // "a"
    goto label_412346;

eax = (DWORD) ch_local2;
if(eax > 0x7A) // "z"
    goto label_412346;

eax = (DWORD) ch_local2;
eax = eax - 0x20;
ecx = pString;
ecx = ecx + i;
*(ecx) = al;
```
Let’s make this code a bit more clever, and thus shorter, renaming the character to \( c \) for shortness, as well as assuming it’s using the char as a character and not as a byte sized number:

\[
c = pString[i];
\]

\[
if((c < 'a') || (c > 'z'))
    goto label_412346;
\]

\[
pString[i] = c-0x20;
\]

We notice that the address 412346h is simply the end of the loop, so we can either replace the ‘goto label_412346’ with a ‘continue;’, or we can invert the conditional jumps. I chose the latter, because it seemed like a more natural way to describe the condition, as you will see later. Inverting the condition might require some explanation:

When the program goes to the end of the loop if \((c<'a') \mid (c>'z')\), then it DOESN’T go to the end of the loop, if \((c>='a') \& \& (c<='z')\), which allows an if construction as follows:

\[
c = pString[i];
\]

\[
if((c >= 'a') \& \& (c <= 'z'))
    pString[i] = ch_local2-0x20;
//...end of loop
\]

This makes it look MUCH clearer. We can now begin to understand what this code is doing. Let’s put all of the code we have together.

```c
void SomeFunction(char* pString)
{
    int i;        //Local variables have to be declared
    unsigned char c;    //at the start of the function.

    for(i = 0; i < lstrlenA(pString); i++)
    {
        c = pString[i];

        if((c >= 'a') \& \& (c <= 'z'))
            pString[i] = c-0x20;
    }
}
```

A great deal shorter than the ASM code we started from. Now that we’ve converted all this back to C, we should be able to figure out the task it performs.

It takes every character in the string it gets, and if that character is between ‘a’ and ‘z’ (so, if it is a lowercase alphabetic character), it subtracts 20h. This is exactly the difference between the uppercase and lowercase characters. So what this function does is ‘convert a string to uppercase’, and should be renamed as ToUppercase.

In this manner all code can slowly be converted, though some structures are harder to identify than others.
Conclusion

A normal engineering process goes from the source code (C) to a binary format (.exe), while what I have described in this document goes entirely the other way. That is the reason this process is called Reverse Engineering. We have seen this is not impossible to do. With tools however, the task can be simplified a lot. The main tool one would use for this kind of thing is IDA. It is both flexible and powerful, and even for the translation back to (pseudo) C code there are plugins under development*. To create proper C code from an ASM or Pseudo-C listing is a task not to be underestimated. It is quite hard to recognize high level structures at first. A good exercise is to write your own program in MSVC++, and debug it with the disassembly view on. This gives your C code along with the ASM code it represents, which will give you a good understanding of how the two relate to one another.

As with most things, practice makes perfect.

Kwazy Webbit

Webbithole: http://its.mine.nu/
RETeam: http://www.reteam.org/

Special thanks go to DEATH, for proofreading this essay and being a perfectionist :-)

* IDA (http://www.datarescue.com/) combined with Lantern (http://www.xopesystems.com/lantern/) forms a tool that automates almost the entire process I described in this document. The only thing it doesn’t do is create actual C code, since that requires a lot of understanding and recognition of structures. It does create the Pseudo-C code I have used throughout the ‘Assembly Code to C’ chapter. See their site for details.
Appendix A1

Key to decoding single-byte instructions (*group references Appendix A2)

### IA-32 architecture

#### one byte opcodes

<table>
<thead>
<tr>
<th>xch</th>
<th>x0h</th>
<th>x1h</th>
<th>x2h</th>
<th>x3h</th>
<th>x4h</th>
<th>x5h</th>
<th>x6h</th>
<th>x7h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xh</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>PUSH</td>
<td>POP</td>
</tr>
<tr>
<td></td>
<td>Eb,Gb</td>
<td>Ev,Gv</td>
<td>Gb,Eb</td>
<td>Gv,Ev</td>
<td>AL,1b</td>
<td>rAX,1z</td>
<td>ES</td>
<td>POP</td>
</tr>
<tr>
<td>1xh</td>
<td>ADC</td>
<td>ADC</td>
<td>ADC</td>
<td>ADC</td>
<td>ADC</td>
<td>ADC</td>
<td>PUSH</td>
<td>POP</td>
</tr>
<tr>
<td></td>
<td>Eb,Gb</td>
<td>Ev,Gv</td>
<td>Gb,Eb</td>
<td>Gv,Ev</td>
<td>AL,1b</td>
<td>rAX,1z</td>
<td>SS</td>
<td>SS</td>
</tr>
<tr>
<td>2xh</td>
<td>AND</td>
<td>AND</td>
<td>AND</td>
<td>AND</td>
<td>AND</td>
<td>AND</td>
<td>ES:</td>
<td>DAA</td>
</tr>
<tr>
<td></td>
<td>Eb,Gb</td>
<td>Ev,Gv</td>
<td>Gb,Eb</td>
<td>Gv,Ev</td>
<td>AL,1b</td>
<td>rAX,1z</td>
<td>SS:</td>
<td>AAA</td>
</tr>
<tr>
<td>3xh</td>
<td>XOR</td>
<td>XOR</td>
<td>XOR</td>
<td>XOR</td>
<td>XOR</td>
<td>XOR</td>
<td>INC</td>
<td>INC</td>
</tr>
<tr>
<td></td>
<td>Eb,Gb</td>
<td>Ev,Gv</td>
<td>Gb,Eb</td>
<td>Gv,Ev</td>
<td>AL,1b</td>
<td>rAX,1z</td>
<td>INC</td>
<td>INC</td>
</tr>
<tr>
<td>4xh</td>
<td>INC</td>
<td>INC</td>
<td>INC</td>
<td>INC</td>
<td>INC</td>
<td>INC</td>
<td>INC</td>
<td>INC</td>
</tr>
<tr>
<td></td>
<td>eAX</td>
<td>eCX</td>
<td>eDX</td>
<td>eBX</td>
<td>eSP</td>
<td>eBP</td>
<td>eSI</td>
<td>eDI</td>
</tr>
<tr>
<td>5xh</td>
<td>PUSH</td>
<td>PUSH</td>
<td>PUSH</td>
<td>PUSH</td>
<td>PUSH</td>
<td>PUSH</td>
<td>PUSH</td>
<td>PUS</td>
</tr>
<tr>
<td></td>
<td>rAX</td>
<td>rCX</td>
<td>rDX</td>
<td>rBX</td>
<td>rSP</td>
<td>rBP</td>
<td>rSI</td>
<td>rDI</td>
</tr>
<tr>
<td>6xh</td>
<td>PUSHA</td>
<td>PUSHA</td>
<td>POPA</td>
<td>POPA</td>
<td>BOUND</td>
<td>ARPL</td>
<td>FS:</td>
<td>GS:</td>
</tr>
<tr>
<td></td>
<td>POPAD</td>
<td>POPAD</td>
<td>POPAD</td>
<td>POPAD</td>
<td>Ev,Gv, Ma</td>
<td>Ev, Gw</td>
<td>(80386+)</td>
<td>(80386+)</td>
</tr>
<tr>
<td>7xh</td>
<td>JMP</td>
<td>JNO</td>
<td>JB</td>
<td>JNB</td>
<td>JZ</td>
<td>JNZ</td>
<td>JBE</td>
<td>JNBE</td>
</tr>
<tr>
<td>8xh</td>
<td>group #1</td>
<td>group #1</td>
<td>group #1</td>
<td>group #1</td>
<td>TEST</td>
<td>TEST</td>
<td>XCHG</td>
<td>XCHG</td>
</tr>
<tr>
<td></td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
<td>Eb,Gb</td>
<td>Gb,Eb</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
</tr>
<tr>
<td>9xh</td>
<td>NOP</td>
<td>PAUSE (F3h)</td>
<td>XCHG</td>
<td>XCHG</td>
<td>XCHG</td>
<td>XCHG</td>
<td>XCHG</td>
<td>XCHG</td>
</tr>
<tr>
<td></td>
<td>ESCA</td>
<td>rCX, rAX</td>
<td>rCX, rAX</td>
<td>rDX, rAX</td>
<td>rDX, rAX</td>
<td>rBX, rAX</td>
<td>rSF, rAX</td>
<td>rBP, rAX</td>
</tr>
<tr>
<td>Axh</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td></td>
<td>AL, Ob</td>
<td>AL, Ob</td>
<td>AL, Ob</td>
<td>AL, Ob</td>
<td>AL, Ob</td>
<td>AL, Ob</td>
<td>AL, Ob</td>
<td>AL, Ob</td>
</tr>
<tr>
<td>8xh</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td></td>
<td>AL, Ib</td>
<td>CL, Ib</td>
<td>DL, Ib</td>
<td>BL, Ib</td>
<td>AH, Ib</td>
<td>CH, Ib</td>
<td>DH, Ib</td>
<td>BH, Ib</td>
</tr>
<tr>
<td>Cxh</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
</tr>
<tr>
<td></td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
<td>Eb, Ib</td>
<td>Ev, Ib</td>
</tr>
<tr>
<td>0xh</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
<td>group #2</td>
</tr>
<tr>
<td></td>
<td>Eb, L</td>
<td>Ev, L</td>
<td>Eb, L</td>
<td>Ev, L</td>
<td>Eb, L</td>
<td>Ev, L</td>
<td>Eb, L</td>
<td>Ev, L</td>
</tr>
<tr>
<td>Dxh</td>
<td>LOOPNE</td>
<td>LOOPNZ</td>
<td>LOOPZ</td>
<td>LOOPNE</td>
<td>LOOPNZ</td>
<td>LOOPZ</td>
<td>LOOPNE</td>
<td>LOOPNZ</td>
</tr>
<tr>
<td></td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
</tr>
<tr>
<td>Exh</td>
<td>LOCK</td>
<td>LOCK</td>
<td>LOCK</td>
<td>LOCK</td>
<td>LOCK</td>
<td>LOCK</td>
<td>LOCK</td>
<td>LOCK</td>
</tr>
</tbody>
</table>

*group references Appendix A2*
<table>
<thead>
<tr>
<th>xh</th>
<th>x8h</th>
<th>x9h</th>
<th>xAh</th>
<th>xBh</th>
<th>xCh</th>
<th>xDh</th>
<th>xEh</th>
<th>xFh</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xh</td>
<td>OR Eb,Gb</td>
<td>OR Ev,Gv</td>
<td>OR GB,EB</td>
<td>OR Gv,Ev</td>
<td>OR AL,1b</td>
<td>OR rAX,IZ</td>
<td>PUSH CS</td>
<td>two byte opcodes (80286+)</td>
</tr>
<tr>
<td>1xh</td>
<td>SBB Eb,Gb</td>
<td>SBB Ev,Gv</td>
<td>SBB GB,EB</td>
<td>SBB Gv,Ev</td>
<td>SBB AL,1b</td>
<td>SBB rAX,IZ</td>
<td>PUSH DS</td>
<td>POP DS</td>
</tr>
<tr>
<td>2xh</td>
<td>SUB Eb,Gb</td>
<td>SUB Ev,Gv</td>
<td>SUB GB,EB</td>
<td>SUB Gv,Ev</td>
<td>SUB AL,1b</td>
<td>SUB rAX,IZ</td>
<td>CS: Hint Not Taken for Jcc (P4+)</td>
<td>DAS</td>
</tr>
<tr>
<td>3xh</td>
<td>CMP Eb,Gb</td>
<td>CMP Ev,Gv</td>
<td>CMP GB,EB</td>
<td>CMP Gv,Ev</td>
<td>CMP AL,1b</td>
<td>CMP rAX,IZ</td>
<td>DS: Hint Taken for Jcc (P4+)</td>
<td>AAS</td>
</tr>
<tr>
<td>4xh</td>
<td>DEC eAX</td>
<td>DEC eCX</td>
<td>DEC eDX</td>
<td>DEC eBX</td>
<td>DEC eSP</td>
<td>DEC eBP</td>
<td>DEC eSI</td>
<td>DEC eDI</td>
</tr>
<tr>
<td>5xh</td>
<td>POP rAX</td>
<td>POP rCX</td>
<td>POP rDX</td>
<td>POP rBX</td>
<td>POP rSP</td>
<td>POP rBP</td>
<td>POP rSI</td>
<td>POP rDI</td>
</tr>
<tr>
<td>6xh</td>
<td>PUSH Iz (80186+)</td>
<td>IMUL Gv,Ev,Iz (80186+)</td>
<td>PUSH Ib (80186+)</td>
<td>IMUL Gv,Ev,1b (80186+)</td>
<td>INS Yb,DX (80186+)</td>
<td>INS Yz,DX (80186+)</td>
<td>OUTS DX,Xb (80186+)</td>
<td>OUTS DX,Xz (80186+)</td>
</tr>
<tr>
<td>7xh</td>
<td>JS Jb</td>
<td>JNS Jb</td>
<td>JP Jb</td>
<td>JNP Jb</td>
<td>JL Jb</td>
<td>JNL Jb</td>
<td>JLE Jb</td>
<td>JNLE Jb</td>
</tr>
<tr>
<td>8xh</td>
<td>MOV Eb,Gb</td>
<td>MOV Ev,Gv</td>
<td>MOV GB,EB</td>
<td>MOV Gv,Ev</td>
<td>MOV Mw,Sw</td>
<td>MOV rAX,1b</td>
<td>LEA Gv,M</td>
<td>MOV Sw,Mw MOV Sw,Rv group #10</td>
</tr>
<tr>
<td>9xh</td>
<td>CBW (8088)</td>
<td>CBW/CWDE (80386+)</td>
<td>CALL Ap</td>
<td>WAIT FWAIT</td>
<td>PUSHF Fv</td>
<td>POPF Fv</td>
<td>SAHF</td>
<td>LAHF</td>
</tr>
<tr>
<td>Axh</td>
<td>TEST AL,1b</td>
<td>TEST rAX,Iz</td>
<td>STOS Yb,AL</td>
<td>STOS Yv,rAX</td>
<td>LODS AL,Xb</td>
<td>LODS rAX,Xv</td>
<td>SCAS Yb,AL</td>
<td>SCAS Yv,rAX</td>
</tr>
<tr>
<td>Bxh</td>
<td>MOV rAX,Iv</td>
<td>MOV rCX,Iv</td>
<td>MOV rDX,Iv</td>
<td>MOV rBX,Iv</td>
<td>MOV rSP,Iv</td>
<td>MOV rBP,Iv</td>
<td>MOV rSI,Iv</td>
<td>MOV rDI,Iv</td>
</tr>
<tr>
<td>Cxh</td>
<td>ENTER lw,1b (80186+)</td>
<td>LEAVE (80186+)</td>
<td>RET far lw</td>
<td>RET far INT3</td>
<td>INT 1b</td>
<td>INTO</td>
<td>IRET</td>
<td></td>
</tr>
<tr>
<td>Dxh</td>
<td>ESC 1</td>
<td>ESC 1</td>
<td>ESC 1</td>
<td>ESC 1</td>
<td>ESC 1</td>
<td>ESC 1</td>
<td>ESC 1</td>
<td>ESC 1</td>
</tr>
<tr>
<td>Exh</td>
<td>CALL Jz</td>
<td>JMP Jz</td>
<td>JMP Ap</td>
<td>JMP JB</td>
<td>IN AL,DX</td>
<td>IN eAX,DX</td>
<td>OUT DX,AL</td>
<td>OUT DX,eAX</td>
</tr>
<tr>
<td>Fxh</td>
<td>CLC STC</td>
<td>CLI STI</td>
<td>CLD STD group #4 INC/DEC</td>
<td>group #5 INC/DEC etc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

note: The opcodes marked with * are aliases to other opcodes.
## Appendix A2

### IA-32 architecture

### Opcode groups

<table>
<thead>
<tr>
<th>mod R/M</th>
<th>xx000xxx</th>
<th>xx001xxx</th>
<th>xx010xxx</th>
<th>xx011xxx</th>
<th>xx100xxx</th>
<th>xx101xxx</th>
<th>xx110xxx</th>
<th>xx111xxx</th>
</tr>
</thead>
</table>

#### Group #1
(80..83h)
- ADD
- OR
- ADC
- SBB
- AND
- SUB
- XOR
- CMP

#### Group #2
(C0..C1h) (D0..D3h)
- ROL
- RCR
- RCL
- RCR
- SHL
- SHR
- SAL*
- SAR

#### Group #3
(F6..F7h)
- TEST
- TEST*
- Ib/Iz
- NOT
- NEG
- MUL
- AL/rAX
- IMUL
- AL/rAX
- DIV
- AL/rAX
- IDIV
- AL/rAX

#### Group #4
(FEn)
- INC Eb
- DEC Eb

#### Group #5
(FFh)
- INC Ev
- DEC Ev
- CALL Ev
- CALL Mp
- JMF Ev
- JMF Mp
- PUSH Ev

#### Group #6
(0Fh,00h)
- SLDT Mw
- SLDT Gv
- STR Mw
- STR Gv
- LLDT Mw
- LLDT Gv
- LTR Mw
- LTR Gv
- VERR Mw
- VERR Gv
- VERN Mw
- VERN Gv
- JMPF
- Ev
- (IA-64)

#### Group #7
(0Fh,01h)
- SGDT Ms
- SIDT Ms
- MONITOR
- (C8h)
- MMAIT
- (C9h)
- (see
- CPUID)

#### Group #8
(0Fh,BAh)

#### Group #9
(0Fh,C7h)
- CMPXCHG
- Mq
- (see
- CPUID)

#### Group #10
(8Fh)
- POP Ev

#### Group #11
(0Fh,89h)
- UD2

#### Group #12
(C6h) (C7h)
- MOV

#### Group #13
(0Fh,71h)
- PSRLW
- PRq, Ib
- (MMX)
- (66h)
- PSRLW
- VRo, Ib
- (SSe2)
- (66h)
- PSRLD

#### Group #14
(0Fh,72h)
- PSRLD
- PRq, Ib
- (MMX)
- (66h)
- PSRLD
- FRq, Ib
- (MMX)
- (66h)
- PSRLD
- PRq, Ib
- (MMX)
- (66h)
- PSRLD
<table>
<thead>
<tr>
<th>Group 15</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VRo, Ib (SSE2)</td>
<td>VRo, Ib (SSE2)</td>
<td>VRo, Ib (SSE2)</td>
<td>VRo, Ib (SSE2)</td>
<td></td>
</tr>
<tr>
<td>(66h) PSRLQ PRq, Ib (MMX)</td>
<td>(66h) PSRLDQ VRo, Ib (SSE2)</td>
<td>(66h) PSLLQ PRq, Ib (MMX)</td>
<td>(66h) PSLLDQ VRo, Ib (SSE2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group 16</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FXSAVE M512 (see CPUID)</td>
<td>FXRSTOR M512 (see CPUID)</td>
<td>SIMDCSR Md (SSE)</td>
<td>STMXCSR Md (SSE)</td>
<td>LFENCE (SSE2-MEM)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group 17</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFETCH-NTA M (SSE-MEM)</td>
<td>PREFETCH-T0 M (SSE-MEM)</td>
<td>PREFETCH-T2 M (SSE-MEM)</td>
<td>HINT_NOP M (P6+)</td>
<td>HINT_NOP M (P6+)</td>
</tr>
</tbody>
</table>

Note: The opcodes marked with * are aliases to other opcodes.
Appendix A3

IA-32 architecture
32bit mod R/M byte

<table>
<thead>
<tr>
<th>reg=</th>
<th>AL</th>
<th>CL</th>
<th>DL</th>
<th>BL</th>
<th>AH</th>
<th>DH</th>
<th>BH</th>
</tr>
</thead>
<tbody>
<tr>
<td>r8(/z)</td>
<td>AX</td>
<td>CX</td>
<td>DX</td>
<td>BX</td>
<td>SP</td>
<td>BP</td>
<td>SI</td>
</tr>
<tr>
<td>r16(/z)</td>
<td>EAX</td>
<td>ECX</td>
<td>EDX</td>
<td>EBX</td>
<td>ESP</td>
<td>EBP</td>
<td>ESI</td>
</tr>
<tr>
<td>r32(/z)</td>
<td>MM0</td>
<td>MM1</td>
<td>MM2</td>
<td>MM3</td>
<td>MM4</td>
<td>MM5</td>
<td>MM6</td>
</tr>
<tr>
<td>xmm(/z)</td>
<td>XMM0</td>
<td>XMM1</td>
<td>XMM2</td>
<td>XMM3</td>
<td>XMM4</td>
<td>XMM5</td>
<td>XMM6</td>
</tr>
<tr>
<td>sreg</td>
<td>ES</td>
<td>CS</td>
<td>SS</td>
<td>DS</td>
<td>FS</td>
<td>GS</td>
<td>res.</td>
</tr>
<tr>
<td>eee</td>
<td>CR0</td>
<td>CR1</td>
<td>CR2</td>
<td>CR3</td>
<td>CR4</td>
<td>CR5</td>
<td>CR6</td>
</tr>
<tr>
<td>/digit (opcode)</td>
<td>DR0</td>
<td>DR1</td>
<td>DR2</td>
<td>DR3</td>
<td>DR4</td>
<td>DR5</td>
<td>DR6</td>
</tr>
<tr>
<td>reg=</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>effective address</th>
<th>mod</th>
<th>R/M</th>
<th>value of mod R/M byte (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>00</td>
<td>08</td>
</tr>
<tr>
<td>[ECX]</td>
<td>00</td>
<td>01</td>
<td>09</td>
</tr>
<tr>
<td>[EDX]</td>
<td>00</td>
<td>02</td>
<td>0A</td>
</tr>
<tr>
<td>[EBX]</td>
<td>00</td>
<td>03</td>
<td>0B</td>
</tr>
<tr>
<td>[esi]</td>
<td>100</td>
<td>04</td>
<td>0C</td>
</tr>
<tr>
<td>[edi]</td>
<td>100</td>
<td>05</td>
<td>0D</td>
</tr>
<tr>
<td>[EAX]+sbyte</td>
<td>00</td>
<td>00</td>
<td>08</td>
</tr>
<tr>
<td>[ECX]+sbyte</td>
<td>00</td>
<td>01</td>
<td>09</td>
</tr>
<tr>
<td>[EDX]+sbyte</td>
<td>00</td>
<td>02</td>
<td>0A</td>
</tr>
<tr>
<td>[EBX]+sbyte</td>
<td>00</td>
<td>03</td>
<td>0B</td>
</tr>
<tr>
<td>[esi]+sbyte</td>
<td>100</td>
<td>04</td>
<td>0C</td>
</tr>
<tr>
<td>[edi]+sbyte</td>
<td>100</td>
<td>05</td>
<td>0D</td>
</tr>
<tr>
<td>[EAX]+sdword</td>
<td>00</td>
<td>00</td>
<td>08</td>
</tr>
<tr>
<td>[ECX]+sdword</td>
<td>00</td>
<td>01</td>
<td>09</td>
</tr>
<tr>
<td>[EDX]+sdword</td>
<td>00</td>
<td>02</td>
<td>0A</td>
</tr>
<tr>
<td>[EBX]+sdword</td>
<td>00</td>
<td>03</td>
<td>0B</td>
</tr>
<tr>
<td>[esi]+sdword</td>
<td>100</td>
<td>04</td>
<td>0C</td>
</tr>
<tr>
<td>[edi]+sdword</td>
<td>100</td>
<td>05</td>
<td>0D</td>
</tr>
<tr>
<td>AL/AX/EAX/XMM0/XMM0</td>
<td>11</td>
<td>00</td>
<td>08</td>
</tr>
<tr>
<td>CL/CX/ECX/MM1/MM01</td>
<td>00</td>
<td>01</td>
<td>09</td>
</tr>
<tr>
<td>DL/EX/EDX/MM2/MM02</td>
<td>00</td>
<td>02</td>
<td>0A</td>
</tr>
<tr>
<td>BL/BX/EBX/MM3/MM03</td>
<td>00</td>
<td>03</td>
<td>0B</td>
</tr>
<tr>
<td>AH/SP/ESP/MM4/MM04</td>
<td>00</td>
<td>04</td>
<td>0C</td>
</tr>
<tr>
<td>CH/BP/EBP/MM5/MM05</td>
<td>00</td>
<td>05</td>
<td>0D</td>
</tr>
<tr>
<td>DH/SI/ESI/MM6/MM06</td>
<td>00</td>
<td>06</td>
<td>0E</td>
</tr>
<tr>
<td>BH/DI/EDI/MM7/MM07</td>
<td>00</td>
<td>07</td>
<td>0F</td>
</tr>
</tbody>
</table>