CS 6V81-05: System Security and Malicious Code Analysis
Understanding the Binary Representation

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

January 23rd, 2012
Outline

1. x86 Architecture
2. Data Representation
3. Code Representation
4. Summary
Outline

1. x86 Architecture
2. Data Representation
3. Code Representation
4. Summary
x86 Architecture
Von Neumann Architecture

- Control Unit
- Arithmetic Logic Unit
- Accumulator
- Input
- Output
- Memory

Software code is loaded in memory. Program data is also in the memory. The attacker's goal is to control the data. The data could be interpreted as code.
Von Neumann Architecture

- **Data**
  - Software code is loaded memory
  - Program data is also in the memory
Von Neumann Architecture

Data
- Software code is loaded memory
- Program data is also in the memory

Attacker’s goal
- Control the data
- The data could be interpreted as code
Architecture Approach to stop attack

Fig. 4.3 The modified Harvard architecture
Architecture Approach to stop attack

Fig. 4.3 The modified Harvard architecture

Separating Code and Data
Outline

1. x86 Architecture
2. Data Representation
3. Code Representation
4. Summary
### Binary Data In Memory and Disk

<table>
<thead>
<tr>
<th>Address</th>
<th>Data Representation</th>
<th>Hexadecimal</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>7f 45 46 02 01 01 00</td>
<td>.ELF.............</td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>02 00 3e 00 01 00 00</td>
<td>..&gt;.........$@.....</td>
<td></td>
</tr>
<tr>
<td>00000020</td>
<td>40 00 00 00 00 00 00</td>
<td>@.............@.....</td>
<td></td>
</tr>
<tr>
<td>00000030</td>
<td>00 00 00 00 40 00 38 00</td>
<td>.@...@.8...@.....</td>
<td></td>
</tr>
<tr>
<td>00000040</td>
<td>06 00 00 00 05 00 00 00</td>
<td>.............@.....</td>
<td></td>
</tr>
<tr>
<td>00000050</td>
<td>40 00 40 00 00 00 00 00</td>
<td>@.@.............@.....</td>
<td></td>
</tr>
<tr>
<td>00000060</td>
<td>f8 01 00 00 00 00 00 00</td>
<td>f8 01 00 00 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00000070</td>
<td>08 00 00 00 00 00 00 00</td>
<td>03 00 00 00 04 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00000080</td>
<td>38 02 00 00 00 00 00 00</td>
<td>38 02 40 00 00 00 00 00</td>
<td>8.......8.@.....</td>
</tr>
<tr>
<td>00000090</td>
<td>38 02 40 00 00 00 00 00</td>
<td>1c 00 00 00 00 00 00 00</td>
<td>8.@.............</td>
</tr>
</tbody>
</table>
# Binary Data In Memory and Disk

<table>
<thead>
<tr>
<th>Address</th>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>u/int_8 (char), u/int_16, u/int_32, pointers</td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>float, double(8)</td>
<td></td>
</tr>
<tr>
<td>00000020</td>
<td>C-String, Unicode-String</td>
<td></td>
</tr>
</tbody>
</table>

---

## Data Type

1. u/int_8 (char), u/int_16, u/int_32, pointers
2. float, double(8)
3. C-String, Unicode-String
Binary Representations

The diagram illustrates the binary representation of a 1-bit signal, showing voltage levels at 3.3V, 2.8V, 0.5V, and 0.0V. The signal transitions between these levels, indicating the binary states 0 and 1.
Memory Organization

- Array of Data
- Virtual Addresses (Array Index)
- OS provides address space private to particular "process"
- Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control data allocation/de-allocation
Memory Organization

Array of Data
- Virtual Addresses (Array Index)
- OS provides address space private to particular “process”
- Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control data allocation/de-allocation
Machine Words

Machine Has “Word Size”

- Most current machines use 32 bits (4 bytes) words
- Limits addresses to 4GB
- Machines support multiple data formats
Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
## Word-Oriented Memory Organization

### Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>Address</th>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td></td>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td></td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>Addr = 0016</td>
<td></td>
<td></td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>Addr = 0020</td>
<td></td>
<td></td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>
How should bytes within a multi-byte word be ordered in memory?

- **Big Endian:** Sun, PPC Mac, Internet
  - Least significant byte has highest address
- **Little Endian:** x86
  - Least significant byte has lowest address
Examples

- Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address
Examples

- Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address

0x01234567 at address 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Reading Byte-Reversed Listings

Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code
Reading Byte-Reversed Listings

**Disassembly**
- Text representation of binary machine code
- Generated by program that reads the machine code

**Assembly Example**

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>401e50:</td>
<td>ff 35 9a 71 21 00</td>
<td>pushq 0x21719a(%rip)</td>
</tr>
<tr>
<td>401e56:</td>
<td>ff 25 9c 71 21 00</td>
<td>jmpq *0x21719c(%rip)</td>
</tr>
<tr>
<td>401e5c:</td>
<td>0f 1f 40 00</td>
<td>nopl 0x0(%rax)</td>
</tr>
</tbody>
</table>
Representing Integers

1. int A = 15213;

2. int B = -15213;
Representing Integers

1. int A = 15213;

2. int B = -15213;

Question?

- Will always $x^2 > 0$?
Representing Integers

1. int A = 15213;

2. int B = -15213;

**Question?**

- Will always \( x^2 > 0 \)?
- Integer overflow
How to represent $10^{100}$

In computer science, arbitrary-precision arithmetic indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system.

Using Arrays

Several modern programming languages have built-in support for bignums, and others have libraries available for arbitrary-precision integer and floating-point math. Rather than store values as a fixed number of binary bits related to the size of the processor register, these implementations typically use variable-length arrays of digits.

How to represent $10^{100}$

Arbitrary-precision Arithmetic

In computer science, arbitrary-precision arithmetic indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system.
How to represent $10^{100}$

Arbitrary-precision Arithmetic

In computer science, arbitrary-precision arithmetic indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system.

Using Arrays

Several modern programming languages have built-in support for bignums, and others have libraries available for arbitrary-precision integer and floating-point math. Rather than store values as a fixed number of binary bits related to the size of the processor register, these implementations typically use variable-length arrays of digits.
How to represent $10^{100}$

**Arbitrary-precision Arithmetic**

In computer science, arbitrary-precision arithmetic indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system.

**Using Arrays**

Several modern programming languages have built-in support for bignums, and others have libraries available for arbitrary-precision integer and floating-point math. Rather than store values as a fixed number of binary bits related to the size of the processor register, these implementations typically use variable-length arrays of digits.

Representing Strings

**Strings in C**
- Represented by array of characters
- Each character encoded in ASCII format
- Standard 7-bit encoding of character set
- Character “0” has code 0x30
- Digit i has code 0x30+i
- String should be null-terminated
- Final character = 0

```c
char S[6] = "18243";
```
Representing of Float Point

**float: IEEE 754**

A numerical value $n$ for a float variable is:

$$n = (1 - 2s) \times (1 + m \times 2^{-23}) \times 2^{e-127},$$

where $s$ is a sign bit (zero or one), $m$ is the significand (i.e., the fraction part), and $e$ is the exponent.
Representing of Float Point

**float: IEEE 754**

A numerical value $n$ for a float variable is:

$$n = (1 - 2s) \times (1 + m \times 2^{-23}) \times 2^{e-127},$$

where $s$ is a sign bit (zero or one), $m$ is the significand (i.e., the fraction part), and $e$ is the exponent.
### How to represent $\pi$

| 3.1415926535 8979323846 2643383279 5028841971 6939937510 5820974944 5923078164 0628620899 8628034825 3421170679 8214808651 3282306647 0938446095 5058223172 |
| 5359408128 4811174502 8410270193 8521105559 6446229489 5493038196 4428810975 6659334461 2847564823 3786783165 2712019091 4564856692 3460348610 4543266482 |
| 1339360726 0249141273 7245870066 0631558817 4881520920 9628292540 9171536436 7892590360 0113305305 4882046652 1384146951 9415116094 3305727036 5759591953 |
| 0921861173 8193261179 3105118548 0744623799 6274956735 1885752724 8912279381 8301194912 9833673362 4406566430 8602139494 6395224737 1907021798 6094370277 |
| 0539217176 2931767523 8467481846 7669405132 0005681271 4526356082 7785771342 |

...
How to represent $\pi$

\[ \pi = 6 \arcsin \frac{1}{2} = 3 \sum_{n=0}^{\infty} \frac{(2n)!}{16^n (2n+1)!} \]

\[ = 6 \left( \frac{1}{2^1 \cdot 1} + \frac{1}{2^3 \cdot 3} + \frac{1 \cdot 3}{2^5 \cdot 5} + \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 7} + \cdots \right) \]

\[ = 3 + \frac{1}{8} + \frac{9}{640} + \frac{15}{7168} + \frac{35}{98304} + \frac{189}{2883584} + \frac{693}{54525952} + \frac{429}{167772160} + \cdots \]
Sign Extension

**Task**
- Given w-bit signed integer x
- Convert it to w+k-bit integer with same value

**Rules**
- Make k copies of sign bit
Sign Extension Example

```
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```
### Sign Extension Example

```c
short int x = 15213;
int    ix = (int) x;
short int y = -15213;
int    iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>
## Complement & Increment (Integer Overflow)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$\sim x$</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>$\sim x + 1$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

**Table:** $x = 15213$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>000000000 000000000</td>
</tr>
<tr>
<td>$\sim 0$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 111111111</td>
</tr>
<tr>
<td>$\sim 0 + 1$</td>
<td>0</td>
<td>00 00</td>
<td>00000000 000000000</td>
</tr>
</tbody>
</table>

**Table:** $x = 0$
Outline

1. x86 Architecture
2. Data Representation
3. Code Representation
4. Summary
## Machine Code/Assembly

### Data Representation

<table>
<thead>
<tr>
<th>Reg</th>
<th>Hex Value</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>00000000</td>
<td>55</td>
</tr>
<tr>
<td>ebx</td>
<td>00000000</td>
<td>3</td>
</tr>
<tr>
<td>ecx</td>
<td>00000000</td>
<td>8</td>
</tr>
<tr>
<td>edx</td>
<td>00000000</td>
<td>12</td>
</tr>
<tr>
<td>esi</td>
<td>00000030</td>
<td>517564</td>
</tr>
<tr>
<td>edi</td>
<td>00000030</td>
<td>65452</td>
</tr>
<tr>
<td>ebp</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>esp</td>
<td>00000046</td>
<td>65404</td>
</tr>
<tr>
<td>ip</td>
<td>00000046</td>
<td>65404</td>
</tr>
<tr>
<td>esp</td>
<td>00000050</td>
<td>31744</td>
</tr>
</tbody>
</table>

### Code Representation

```
(0) Breakpoint 1. 0x000000000007c00 in ?? ()
```

### Summary

- [x86 Architecture](#)
- [Data Representation](#)
- [Code Representation](#)
- [Summary](#)
### Instruction Decoding: One-to-one mapping?

<table>
<thead>
<tr>
<th>x86 Architecture</th>
<th>Data Representation</th>
<th>Code Representation</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Linux Kernel**:
  ```
  0x806eee0a:  e8 3d 69 00 00         call   0x806f574c
  ...
  0x806f574c:  8b ff                  mov    edi, edi
  0x806f574e:  55                     push   ebp
  0x806f574f:  8b ec                  mov    ebp, esp
  ```

- **Windows Kernel**:
  ```
  0x806f574c:  8b ff                  mov    edi, edi
  0x806f574e:  55                     push   ebp
  0x806f574f:  8b ec                  mov    ebp, esp
  ```

- **Solaris Kernel**:
  ```
  0xc04d675f:  e8 0c cf 00 00        call   0xc04e3670
  ...
  0xc04e3670:  55                    push   ebp
  0xc04e3671:  89 e5                 mov    ebp,esp
  ```

- **FreeBSD/OpenBSD Kernel**:
  ```
  0xc04d675f:  e8 0c cf 00 00        call   0xc04e3670
  ...
  0xc04e3670:  55                    push   ebp
  0xc04e3671:  89 e5                 mov    ebp,esp
  ```
Instruction Decoding: **One-to-one mapping?**

(a) Linux Kernel

0xc1087c86: e8 25 2c 00 00  
call 0xc108a8b0

0xc108a8b0: 55  
push ebp

0xc108a8b1: 89 e5  
mov ebp, esp

(b) Windows Kernel

0x806eee0a: e8 3d 69 00 00  
call 0x806f574c

0x806f574c: 8b ff  
mov edi, edi

0x806f574e: 55  
push ebp

0x806f574f: 8b ec  
mov ebp, esp

(c) Solaris Kernel

0xfe801a3c: e8 9c 01 00 00  
call 0xfe801bdd

0xfe801bdd: 55  
push ebp

0xfe801bde: 8b ec  
mov ebp, esp

(d) FreeBSD/OpenBSD Kernel

0xc04d675f: e8 0c cf 00 00  
call 0xc04e3670

0xc04e3670: 55  
push ebp

0xc04e3671: 89 e5  
mov ebp, esp
Encoding Real x86 Instructions

(a) Optional instruction prefixes

(b) General instruction format
Encoding Real x86 Instructions

(a) Optional instruction prefixes

(b) General instruction format

Prefix Bytes
Zero to four special prefix values that affect the operation of the instruction

“mod-reg-r/m” byte that specifies the addressing mode and instruction operand size.

Displacement. This is a zero, one, two, or four byte value that specifies a memory address displacement for the instruction.

Optional Scaled Indexed Byte. If the instruction uses a scaled indexed memory addressing mode.

Immediate (constant) data. This is a zero, one, two, or four byte constant value if the instruction has an immediate operand.

One or two byte instruction opcode (two bytes if the special 0Fh opcode expansion prefix is present).
Legacy prefixes (optional)

- Lock prefix (1 byte)
- Repeat prefix (1 byte)
- Segment override prefix (1 byte)
- Operand-size override prefix (1 byte)
- Address-size override prefix (1 byte)
- REX prefix (1 byte, 64-bit only)
x86 Instruction Format Reference

**Encoding**

- **Opcode** (1, 2 bytes, required)
- **ModR/M** (1 byte, if required)
- **SIB** (1 byte, if required)
- **Displacement** (1, 2 or 4 bytes, if required)
- **Immediate** (1, 2 or 4 bytes, if required)
Encoding x86 Instruction Operands, MOD-REG-R/M

Byte

<table>
<thead>
<tr>
<th>MOD</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Register indirect addressing mode or SIB with no displacement (when R/M = 100) or Displacement only addressing mode (when R/M = 101).</td>
</tr>
<tr>
<td>01</td>
<td>One-byte signed displacement follows addressing mode byte(s).</td>
</tr>
<tr>
<td>10</td>
<td>Four-byte signed displacement follows addressing mode byte(s).</td>
</tr>
<tr>
<td>11</td>
<td>Register addressing mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REG Value</th>
<th>Register if data size is eight bits</th>
<th>Register if data size is 16-bits</th>
<th>Register if data size is 32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>al</td>
<td>ax</td>
<td>eax</td>
</tr>
<tr>
<td>001</td>
<td>cl</td>
<td>cx</td>
<td>ecx</td>
</tr>
<tr>
<td>010</td>
<td>dl</td>
<td>dx</td>
<td>edx</td>
</tr>
<tr>
<td>011</td>
<td>bl</td>
<td>bx</td>
<td>ebx</td>
</tr>
<tr>
<td>100</td>
<td>ah</td>
<td>sp</td>
<td>esp</td>
</tr>
<tr>
<td>101</td>
<td>ch</td>
<td>bp</td>
<td>ebp</td>
</tr>
<tr>
<td>110</td>
<td>dh</td>
<td>si</td>
<td>esi</td>
</tr>
<tr>
<td>111</td>
<td>bh</td>
<td>di</td>
<td>edi</td>
</tr>
</tbody>
</table>
Decoding Statement Machine

Diagram of decoding process:
- Reset
- Prefix
- Opcode
- ModRM
- SIB
- Disp
- Imm
- add cl, al
- 00 C1 (if d=0), or 02 C8, if d bit is set to 1.
Encoding ADD ECX, EAX Instruction

- `add ecx, eax`
- Could also encode ADD ECX, EAX using the bytes 03 C8
Encoding ADD EDX, DISPLACEMENT Instruction

- **add edx, disp**

**000000** indicates that this is an ADD instruction.

**000011** indicates that we are adding 32 bit values together.

**00 0 0 0 0 1 1**

**MOD** 0 0 0 0 1 1

**R/M** 1 1 1 0 1

**REG**

**00**

**disp**

The combination of **MOD = 00** and **R/M = 101** indicates that this is the Displacement-only addressing mode.

One indicates that we are adding the R/M field to the REG field.

This field, along with the d bit in the opcode, indicates that the destination field is the EDX register.

**ADD edx, disp = 03, 1D, ww, xx, yy, zz**

Note: **ww, xx, yy, zz** represent the four displacement byte values with **ww** being the L.O. byte and **zz** being the H.O. byte.

32-bit displacement follows the instruction.
Encoding ADD EDI, [EBX] Instruction

- **add edi, [ebx]**

000000 indicates that this is an ADD instruction.

One indicates that we are adding 32 bit values together.

00 indicates a zero byte displacement.

011 indicates the use of the [EBX] addressing mode.

REG

This field, along with the d bit in the opcode, indicates that the destination field is the EDI register.

ADD edi, [ebx] = 03 3B
ADD EAX, [ ESI + disp8 ]

- add eax, [ esi + disp8 ]

ADD eax, [esi + disp8] = 03, 46, xx
Encoding ADD EBX, [ EBP + disp32 ] Instruction

- **add ebx, [ ebp + disp32 ]**

ADD ebx, [ebp+disp32] = 03, 9D, ww, xx, yy, zz

Note: *ww, xx, yy, zz* represent the four displacement byte values with *ww* being the L.O. byte and *zz* being the H.O. byte.
Encoding ADD EBP, [ disp32 + EAX*1 ] Instruction

- add ebp, [ disp32 + eax*1 ]
Encoding ADD ECX, [ EBX + EDI*4 ] Instruction

- add ecx, [ ebx + edi*4 ]
Encodings ADD EBP, [ disp32 + EAX*1 ] Instruction

- `add ebp, [ disp32 + eax*1 ]`

The instruction encodes as follows:

```
ADD ebp, [disp32 + eax*1] = 03, 2C, 05, ww, xx, yy, zz
```

Note: `ww, xx, yy, zz` represent the four displacement byte values with `ww` being the L.O. byte and `zz` being the H.O. byte.
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Machine Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>402688:</td>
<td>83 fa 06</td>
<td>cmp $0x6, %edx</td>
<td></td>
</tr>
<tr>
<td>40268b:</td>
<td>0f 94 c1</td>
<td>sete %cl</td>
<td></td>
</tr>
<tr>
<td>40268e:</td>
<td>84 c9</td>
<td>test %cl,%cl</td>
<td></td>
</tr>
<tr>
<td>402690:</td>
<td>b8 40 00 00 00</td>
<td>mov $0x40, %eax</td>
<td></td>
</tr>
<tr>
<td>402695:</td>
<td>75 41</td>
<td>jne 4026d8 &lt;wcstombs@plt+0x258&gt;</td>
<td></td>
</tr>
<tr>
<td>402697:</td>
<td>40 84 ff</td>
<td>test %dl,%dl</td>
<td></td>
</tr>
<tr>
<td>402699:</td>
<td>0f 84 a0 00 00 00</td>
<td>je 402740 &lt;wcstombs@plt+0x2c0&gt;</td>
<td></td>
</tr>
<tr>
<td>4026a0:</td>
<td>89 f0</td>
<td>mov %esi,%eax</td>
<td></td>
</tr>
<tr>
<td>4026a2:</td>
<td>25 00 f0 00 00</td>
<td>and $0xf000, %eax</td>
<td></td>
</tr>
<tr>
<td>4026a7:</td>
<td>3d 00 10 00 00</td>
<td>cmp $0x1000, %eax</td>
<td></td>
</tr>
<tr>
<td>4026ac:</td>
<td>0f 94 c1</td>
<td>sete %cl</td>
<td></td>
</tr>
<tr>
<td>4026af:</td>
<td>84 c9</td>
<td>test %cl,%cl</td>
<td></td>
</tr>
<tr>
<td>4026b1:</td>
<td>b8 7c 00 00 00</td>
<td>mov $0x7c, %eax</td>
<td></td>
</tr>
<tr>
<td>4026b6:</td>
<td>75 20</td>
<td>jne 4026d8 &lt;wcstombs@plt+0x258&gt;</td>
<td></td>
</tr>
<tr>
<td>4026b8:</td>
<td>40 84 ff</td>
<td>test %dl,%dl</td>
<td></td>
</tr>
<tr>
<td>4026bb:</td>
<td>0f 84 8f 00 00 00</td>
<td>je 402750 &lt;wcstombs@plt+0x2d0&gt;</td>
<td></td>
</tr>
<tr>
<td>4026c1:</td>
<td>81 e6 00 f0 00 00</td>
<td>and $0xf000, %esi</td>
<td></td>
</tr>
<tr>
<td>4026c7:</td>
<td>81 fe 00 c0 00 00</td>
<td>cmp $0xc000, %esi</td>
<td></td>
</tr>
<tr>
<td>4026cd:</td>
<td>0f 94 c0</td>
<td>sete %al</td>
<td></td>
</tr>
<tr>
<td>4026d0:</td>
<td>f7 d8</td>
<td>neg %eax</td>
<td></td>
</tr>
<tr>
<td>4026d2:</td>
<td>83 e0 3d</td>
<td>and $0x3d, %eax</td>
<td></td>
</tr>
<tr>
<td>4026d5:</td>
<td>0f 1f 00</td>
<td>nopl (%rax)</td>
<td></td>
</tr>
<tr>
<td>4026d8:</td>
<td>f3 c3</td>
<td>repz retq</td>
<td></td>
</tr>
<tr>
<td>4026da:</td>
<td>66 0f 1f 44 00 00</td>
<td>nopw 0x0(%rax,%rax,1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>403594:</td>
<td>b8 ff ff ff ff ff mov $0xffffffff, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>403599:</td>
<td>c3</td>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>
1. x86 Architecture
2. Data Representation
3. Code Representation
4. Summary
Data Representation

- Little endian in x86
- Data is decoded based on the context
  - float
  - integer, pointer
Summary

Data Representation
- Little endian in x86
- Data is decoded based on the context
  - float
  - integer, pointer

Code Representation

![Code Representation Diagram]

Instruction Prefixes
- 1-4 bytes (optional)

REX Prefix
- 1 byte (optional)

Opcode
- 1-3 bytes (mandatory)

Mod R/M
- 1, 2, or 4 bytes (if required)

SIB
- 1, 2, or 4 bytes (if required)

Displacement
- 1 byte (if required)

Immediate
- 1 byte (if required)
References

5. http://www.m5sim.org/X86_Instruction_decoding