CS165 – Computer Security

Understanding low-level program execution
Oct 1st, 2015
“A computer lets you make more mistakes faster than any invention in human history - with the possible exceptions of handguns and tequila”
What will **executing** this program do?

```c
#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}
```

42.c
To answer the question “Is this program safe/secure/vulnerable?”

We need to know “What will executing this program do?”
To answer the question

“Is this program safe/secure/vulnerable?”

We need to know

“What will executing this program do?”

Understanding the compiler and machine semantics are key.
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}

void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}
void answer(char *name, int x) {
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}

void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}

Alice

0011010
1101010
1000101

Compilation

Alice, the answer is 4393
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 4300 + 93;
    answer(argv[1], x);
}

Alice, the answer is 4393

The *compiler* and *machine* determines the semantics
“Compiled Code”

Source Language → Compilation → Target Language

Input → Output

Terminology: Target language = machine-readable code = binary instructions
“Interpreted Code”

Source Language

Input

Interpretation

Output
Source Language → Compilation → Target Language
Source Language
4393.c in C

Compilation

Target Language
4393 in x86
Source Language
4393.c in C

Compilation

Target Language
4393 in x86

Pre-processor (cpp)
4393.c

Compiler (cc1)

Assembler (as)

Linker (ld)

4393
$\texttt{cpp}$

```
#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
...
```

#include expansion
#define substitution
$ gcc -S

#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
...

Creates Assembly
gcc –S 4393.c outputs 4393.s

_answer:
Leh_func_begin1:
  pushq %rbp
Ltmp0:
  movq %rsp, %rbp
Ltmp1:
  subq $16, %rsp
Ltmp2:
  movl %esi, %eax
  movq %rdi, -8(%rbp)
  movl %eax, -12(%rbp)
  movq -8(%rbp), %rax
  ....
$ as <options>

Creates object code

```
//_answer:
Leh_func_begin1:
    pushq %rbp
Ltmp0:
    movq %rsp, %rbp
Ltmp1:
    subq $16, %rsp
Ltmp2:
    movl %esi, %eax
    movq %rdi, -8(%rbp)
    movl %eax, -12(%rbp)
    movq -8(%rbp), %rax
....
```

4393.s
Pre-processor (cpp) \rightarrow Compiler (cc1) \rightarrow Assembler (as) \rightarrow Linker (ld)

Source \rightarrow Source \rightarrow Assembly \rightarrow Binary \rightarrow Binary

$ ld <options> $

```
0101100101010101101010101
1010101010101011111111100
00110101011010101001011
01011101010100101100001010
10111101
```

4393.o

Links with other files and libraries to produce an exe

Source

Assembly

Binary

Binary
Disassembling

• Today: using objdump (part of binutils)
  – objdump –D <exe>

• If you compile with “-g”, you will see more information
  – objdump –D –S
The program **binary** (aka executable)

Final executable consists of several **segments**

- Text for code written
- Read-only data for constants such as “hello world” and globals
- ...

**Binary**

- Code Segment (`.text`)
- Data Segment (`.data`)
- ...

...
The program *binary* (aka executable)

Final executable consists of several *segments*

- Text for code written
- Read-only data for constants such as “hello world” and globals
- ...

$ readelf –S <file>
Machine Instruction Example

- **C Code**
  - Add two signed integers

- **Assembly**
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - \( x \): Register \( %eax \)
    - \( y \): Memory \( M[%ebp+8] \)
    - \( t \): Register \( %eax \)
      - Return function value in \( %eax \)

- **Object Code**
  - 3-byte instruction
  - Stored at address \( 0x80483ca \)

```
int t = x+y;

addl 8(%ebp),%eax

Similar to expression:
\[ x += y \]

More precisely:

```c
int eax;
int *ebp;
eax += *(ebp[2])
```

```
0x80483ca: 03 45 08
```
Agenda

• Compilation Workflow

• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
Basic Execution

File system

Binary
- Code
- Data
- ...

Process Memory

Processor

Process

Memory

Basic Execution

File system

Binary
- Code
- Data
- ...

Process Memory

Processor

Process

Memory
Basic Execution

- Binary
- File system

- Processor
- Process Memory
  - Code
  - Data
  - Stack
  - Heap
Basic Execution

Binary
File system

Code
Data
...
Stack
Heap
Process Memory

Fetch, decode, execute

Processor
Basic Execution

Binary

File system

Process Memory

Processor

Fetch, decode, execute

read and write

Code

Data

Stack

Heap
x86 Processor

- EIP
- EFLAGS
- EAX
- EDX
- ECX
- EBX
- ESP
- EBP
- ESI
- EDI
x86 Processor

- EAX
- EDX
- ECX
- EBX
- ESP
- EBP
- EDI
- ESI
- EIP
- EFLAGS

Address of next instruction
x86 Processor

- EAX
- EDX
- ECX
- EBX
- ESP
- EBP
- EDI
- ESI
- EIP
- EFLAGS

Address of next instruction

Condition codes
x86 Processor

- EAX
- EDX
- ECX
- EBX
- ESP
- EBP
- EDI
- ESI
- EIP
- EFLAGS

- Address of next instruction
- Condition codes
- General Purpose
Registers have up to 4 addressing modes

1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register
# EAX, EDX, ECX, and EBX

- **EAX**
  - AH
  - AL

- **EDX**
  - DH
  - DL

- **ECX**
  - CH
  - CL

- **EBX**
  - BH
  - BL

- **AX**
  - AH
  - AL

- **DX**
  - DH
  - DL

- **CX**
  - CH
  - CL

- **BX**
  - BH
  - BL

- 32 bit registers *(three letters)*
- Lower bits (bits 0-7) *(two letters with L suffix)*
- Mid-bits (bits 8-15) *(two letters with H suffix)*
- Lower 16 bits (bits 0-15) *(2 letters with X suffix)*
### ESP, EBP, ESI, and EDI

<table>
<thead>
<tr>
<th>EAX</th>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>DH</td>
<td>DL</td>
</tr>
<tr>
<td>ECX</td>
<td>CH</td>
<td>CL</td>
</tr>
<tr>
<td>EBX</td>
<td>BH</td>
<td>BL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP</td>
<td>BP</td>
</tr>
<tr>
<td>ESI</td>
<td>SI</td>
</tr>
<tr>
<td>EDI</td>
<td>DI</td>
</tr>
</tbody>
</table>

| Bit 32 | 16  | 15  | 0 |

- Lower 16 bits (bits 0-15) (2 letters)
x86 Implementation

- **EIP** is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and cond. JMP
x86 Instruction Set

• Instructions classes:
  – Data Movement: MOV, PUSH, POP, ...
  – Arithmetic: TEST, SHL, ADD, ...
  – I/O: IN, OUT, ...
  – Control: JMP, JZ, JNZ, CALL, RET
  – String: REP, MOVSB, ...
  – System: IRET, INT, ...

• Volume 2A: Instruction Set Reference, A-M
  Volume 2B: Instruction Set Reference, N-Z
  – Intel syntax: OP DST, SRC
  – AT&T (gcc/gas) syntax: OP SRC, DST
### Basic Ops and AT&T vs Intel Syntax

<table>
<thead>
<tr>
<th>Meaning</th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx = eax</td>
<td>movl %eax, %ebx</td>
<td>mov ebx, eax</td>
</tr>
<tr>
<td>eax = eax + ebx</td>
<td>addl %ebx, %eax</td>
<td>add eax, ebx</td>
</tr>
<tr>
<td>ecx = ecx &lt;&lt; 2</td>
<td>shl $2, %ecx</td>
<td>shl ecx, 2</td>
</tr>
</tbody>
</table>

- AT&T is **at odds** with assignment order. It is the default for objdump, and traditionally used for UNIX.

- Intel order **mirrors** assignment. Windows traditionally uses Intel, as is available via the objdump ‘-M intel’ command line option.
Agenda

• Compilation Workflow
• x86 Execution Model
  – Basic Execution
  – Memory Operation
  – Control Flow
  – Memory Organization
x86: Byte Addressable
x86: **Byte Addressable**

... 

Address 3 holds 1 byte
Address 2 holds 1 byte
Address 1 holds 1 byte
Address 0 holds 1 byte
x86: **Byte Addressable**

*It’s convention:* lower address at the bottom

- Address 3 holds 1 byte
- Address 2 holds 1 byte
- Address 1 holds 1 byte
- Address 0 holds 1 byte
x86: **Byte Addressable**

I can fetch bytes at any address

Address 3 holds 1 byte
Address 2 holds 1 byte
Address 1 holds 1 byte
Address 0 holds 1 byte
x86: Byte Addressable

Address 0 holds 1 byte
Address 1 holds 1 byte
Address 2 holds 1 byte
Address 3 holds 1 byte

I can fetch bytes at any address

Memory is just like using an array!
x86: Byte Addressable

I can fetch bytes at any address

Address 3 holds 1 byte
Address 2 holds 1 byte
Address 1 holds 1 byte
Address 0 holds 1 byte

Memory is just like using an array!

Alternative: **Word addressable**

**Example:** For 32-bit word size, it’s valid to fetch 4 bytes from Mem[0], but not Mem[6] since 6 is not a multiple of 4.
 Addresses are indicated by operands that have a bracket “[]” or paren “()”, for Intel vs. AT&T, resp.
Addressing

Addresses are indicated by operands that have a bracket “[ ]” or paren “()”, for Intel vs. AT&T, resp.

What does `mov dl, [al]` do?

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0x0</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>
x86: Addressing bytes

Addresses are indicated by operands that have a bracket “[]” or paren “()”, for Intel vs. AT&T, resp.

What does `mov dl, [al]` do?

Moves 0xccc into dl

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0x0</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>
Addresses are indicated by operands that have a bracket “[]” or paren “()”, for Intel vs. AT&T, resp.

What does \texttt{mov edx, [eax]} do?

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>
x86: Addressing bytes

Addresses are indicated by operands that have a bracket “[ ]” or paren “()”, for Intel vs. AT&T, resp.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

What does `mov edx, [eax]` do?

Which 4 bytes get moved, and which is the LSB in edx?
Endianess

- **Endianess**: Order of individually addressable units
- **Little Endian**: Least significant byte first

so address \( a \) goes in littlest byte (e.g., AL), \( a+1 \) in the next (e.g., AH), etc.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Addr |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0xaa</td>
</tr>
<tr>
<td>0xbb</td>
</tr>
<tr>
<td>0xcc</td>
</tr>
<tr>
<td>0xdd</td>
</tr>
<tr>
<td>0xee</td>
</tr>
<tr>
<td>0xff</td>
</tr>
</tbody>
</table>

54
mov edx, [eax]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \( a \) goes in the least significant byte (the **littlest** bit) \( a+1 \) goes into the next byte, and so on.
mov edx, [eax]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

address $a$ goes in the least significant byte (the littlest bit) $a+1$ goes into the next byte, and so on.
mov edx, [eax]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Endianess: Ordering of individually addressable units

Little Endian: Least significant byte first

... so ...

address \(a\) goes in the least significant byte (the littlest bit) \(a+1\) goes into the next byte, and so on.
**mov edx, [eax]**

**Register** | **Value**
---|---
eax | 0x3
edx | 0xcc
ebx | 0x5

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \(a\) goes in the least significant byte (the littlest bit) \(a+1\) goes into the next byte, and so on.
mov edx, [eax]

Endianess: Ordering of individually addressable units
Little Endian: Least significant byte first
... so ...
address $a$ goes in the least significant byte (the littlest bit) $a+1$ goes into the next byte, and so on.
mov edx, [eax]

EDX = 0xffffeeddcc!

**Register | Value**
eax | 0x3
dedx | 0xcc
ebx | 0x5

**Endianess:** Ordering of individually addressable units

**Little Endian:** Least significant byte first

... so ...

address $a$ goes in the least significant byte (the **littlest** bit) $a+1$ goes into the next byte, and so on.
mov [eax], ebx

Register | Value
---|---
eax | 0x3
edx | 0xcc
ebx | 0x5

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \( a \) goes in the least significant byte (the **littlest** bit) \( a+1 \) goes into the next byte, and so on.
mov [eax], ebx

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \(a\) goes in the least significant byte (the **littlest** bit) \(a+1\) goes into the next byte, and so on.
mov [eax], ebx

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address $a$ goes in the least significant byte (the **littlest** bit) $a+1$ goes into the next byte, and so on.
mov [eax], ebx

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Endianess**: Ordering of individually addressable units

**Little Endian**: Least significant byte first

... so ...

address \(a\) goes in the least significant byte (the *littlest* bit) \(a+1\) goes into the next byte, and so on.
mov [eax], ebx

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Endianess: Ordering of individually addressable units
Little Endian: Least significant byte first
... so ...
address \( a \) goes in the least significant byte (the \texttt{littlest} bit) \( a+1 \) goes into the next byte, and so on.
There are other ways to address memory than just [register].

These are called *Addressing Modes*.

An *Addressing Mode* specifies how to calculate the effective memory address of an operand by using information from registers and constants contained with the instruction or elsewhere.
Motivation: Addressing Buffers

```c
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```
Motivation: Addressing Buffers

typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);

What is x? what memory cell does it ref?
Motivation: Addressing Buffers

typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);
y = *( (uint32_t *) (ptr+8));
Motivation: Addressing Buffers

typedef uint32_t addr_t;
uint32_t w, x, y, z;
uint32_t buf[3] = {1,2,3};
addr_t ptr = (addr_t) buf;

w = buf[2];
x = *(buf + 2);
y = *( (uint32_t *) (ptr+8));

**Equivalent**
(addr_t)(ptr + 8) = (uint32_t *) buf+2
Motivation: Addressing Buffers

Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)

Say at imm +r₁
Motivation: Addressing Buffers

Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)

Say at imm +r_1

Constant scaling factor s, typically 1, 2, 4, or 8
Type `buf[s];`
`buf[index] = *(<buf addr>+sizeof(Type)*index)`

Say at `imm + r_1`

Constant `scaling` factor `s`, typically 1, 2, 4, or 8

Say in Register `r_2`
Motivation: Addressing Buffers

Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)

- Say at \text{imm} + r_1
- Constant \textit{scaling} factor \textit{s}, typically 1, 2, 4, or 8
- Say in Register \textit{r}_2

\text{imm} + r_1 + s \times r_2

AT&T: \text{imm} (r_1, r_2, s)
Intel: r_1 + r_2 \times s + \text{imm}
## AT&T Addressing Modes for Common Codes

<table>
<thead>
<tr>
<th>Form</th>
<th>Meaning on memory M</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm (r)</td>
<td>M[r + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂)</td>
<td>M[r₁ + r₂ + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂, s)</td>
<td>M[r₁ + r₂*s + imm]</td>
</tr>
<tr>
<td>imm</td>
<td>M[imm]</td>
</tr>
</tbody>
</table>
Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%edx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx, %ecx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%edx, %ecx, 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80 (, %edx, 2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>
Referencing Memory

Loading a **value** from memory: mov

\[ \text{<eax>} = *\text{buf}; \]

\[
\begin{align*}
\text{mov} & \ -0x38(\%ebp),\%eax \ (A) \\
\text{mov} & \ \text{eax}, \ [\text{ebp}-0x38] \ (I)
\end{align*}
\]

Loading an **address**: lea

\[ \text{<eax>} = \text{buf}; \]

\[
\begin{align*}
\text{lea} & \ -0x38(\%ebp),\%eax \ (A) \\
\text{lea} & \ \text{eax}, \ [\text{ebp}-0x38] \ (I)
\end{align*}
\]
Suppose I want to access address 0xdeadbeef directly.

Loads the address: `lea eax, 0xdeadbeef (I)`

Deref the address: `mov eax, 0xdeadbeef (I)`

Note missing $. This distinguishes the address from the value.
Understanding Swap

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

### Registers and Values

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

### Instructions

- `movl 8(%ebp), %edx`  # edx = xp
- `movl 12(%ebp), %ecx`  # ecx = yp
- `movl (%edx), %ebx`  # ebx = *xp (t0)
- `movl (%ecx), %eax`  # eax = *yp (t1)
- `movl %eax, (%edx)`  # *xp = t1
- `movl %ebx, (%ecx)`  # *yp = t0
Understanding Swap

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

%ebp | 0x104

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
# Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x124</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>-4</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

```
movl 8(%ebp), %edx     # edx = xp
movl 12(%ebp), %ecx    # ecx = yp
movl (%edx), %ebx      # ebx = *xp (t0)
movl (%ecx), %eax      # eax = *yp (t1)
movl %eax, (%edx)      # *xp = t1
movl %ebx, (%ecx)      # *yp = t0
```
# Understanding Swap

- **%eax**
- **%edx** 0x124
- **%ecx** 0x120
- **%ebx**
- **%esi**
- **%edi**
- **%esp**
- **%ebp** 0x104

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12 0x120 0x110</td>
</tr>
<tr>
<td>xp</td>
<td>8 0x124 0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0 0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 0x124</td>
</tr>
<tr>
<td>456 0x120</td>
</tr>
<tr>
<td>114 0x11c</td>
</tr>
<tr>
<td>118 0x118</td>
</tr>
<tr>
<td>104 0x104</td>
</tr>
<tr>
<td>100 0x100</td>
</tr>
<tr>
<td>108 0x108</td>
</tr>
<tr>
<td>10c 0x10c</td>
</tr>
<tr>
<td>110 0x110</td>
</tr>
<tr>
<td>114 0x114</td>
</tr>
</tbody>
</table>

```assembler
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0
```
Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ecx</td>
<td>123</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x104</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
</tbody>
</table>

movl  8(%ebp), %edx  # edx = xp
movl  12(%ebp), %ecx # ecx = yp
movl  (%edx), %ebx   # ebx = *xp (t0)
movl  (%ecx), %eax   # eax = *yp (t1)
movl  %eax, (%edx)   # *xp = t1
movl  %ebx, (%ecx)   # *yp = t0

Offset:
- yp  12  0x120  0x110
- xp  8   0x124  0x10c

Rtn adr:
- %ebp  0  0x104
- -4    0x100

Address:
- 123  0x124
- 456  0x120
-      0x11c
-      0x118
-      0x114
-      0x108
-      0x104
-      0x100
Understanding Swap

%eax  456
%edx  0x124
%ecx  0x120
%ebx  123
%esi  
%edi  
%esp  
%ebp  0x104

movl  8(%ebp), %edx  # edx = xp
movl  12(%ebp), %ecx  # ecx = yp
movl  (%edx), %ebx  # ebx = *xp (t0)
movl  (%ecx), %eax  # eax = *yp (t1)
movl  %eax, (%edx)  # *xp = t1
movl  %ebx, (%ecx)  # *yp = t0

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

123
456
0x120
0x11c
0x118
0x114

Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>12</th>
<th>0x120</th>
<th>0x110</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
<td>4</td>
<td>0x124</td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
<td>-4</td>
<td>0x108</td>
<td></td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>4</td>
<td>0x104</td>
</tr>
<tr>
<td>8</td>
<td>0x10c</td>
</tr>
<tr>
<td>12</td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0