Intel Instruction Tutorial

695.744
T. McGuire
Johns Hopkins University

Introduction

The Intel Instruction Set Architecture (ISA) can be very complicated and cumbersome. Throughout the class, we will see new instructions that may show more of the Intel intricacies. This paper will serve as a tutorial/refresher for the Intel assembly language that will be required in 695.744.

1 Terminology

`Bit` (or **`Binary Digit`**) - A **`bit`** is used to signify the existence or absence of data. As this is binary, a **`bit`** can be represented as a **`0`** or a **`1`**.

`Nibble` (or **`Nybble`**) - A **`nibble`** is 4 consecutive bits. The **`nibble`** comes in handy when quickly converting from *hexadecimal* (base 16) to *binary* (base 2). For example, the byte **`0xC3`** can be broken up into 2 **`nibbles`**: **`0xC`** and **`0x3`**. In binary, these would be **`0b1100`** and **`0b0011`**, respectively.

`Byte` (**`BYTE`**) - A **`byte`** is the smallest, addressable size in the Intel Architecture. It consists of 8 consecutive bits. When discussing **`byte`** ordering, *endianness* matters when there are 2 or more bytes. That is, a single byte has only one representation.

`Word` (**`WORD`**) - A **`word`** consists of 2 consecutive bytes. For example, **`0x4142`** is considered a word.

`Double Word` (**`DWORD`**) - A **`double word`**, or **`DWORD`**, consists of 4 consecutive bytes. Due to the Intel Architecture being *little endian*, when the value **`0x41424344`** is stored in memory it is stored backwards (**`0x44 0x43 0x42 0x41`**). See Figure 1.

`Quad Word` (**`QWORD`**) - A **`quad word`**, or **`QWORD`**, consists of 8 consecutive bytes.

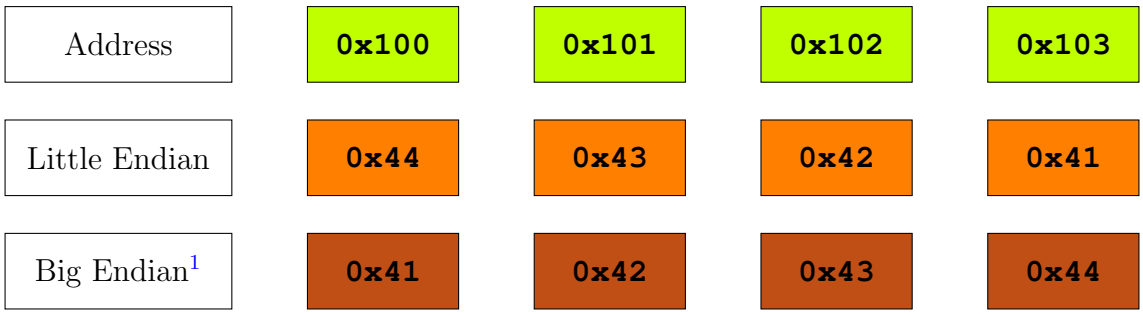


Figure 1:
Little Endian vs. Big Endian storing the **`DWORD`** value **`0x41424344`** in memory

[1]Big Endian is also referred to as Network Byte Order as it is the endianness used when sending network data.

Opcode - A 1-, 2- or 3-byte ***required*** value that represents the machine code value for an instruction. The **ModR/M** byte can optionally extend an opcode by utilizing the 3-bit **REG** field as an extension.

ModR/M- An optional 1-byte value immediately following an opcode that identifies which registers and/or memory addresses are operands. This is only necessary for opcodes with encoding requiring the **ModR/M** byte. The first 2 bits represent the addressing mode, the next 3 bits represent the **REG** field (i.e. the operand in the **r32** position -or- the opcode extension) and the last 3 bits represent the **R/M** field (i.e. the register or memory operand in the **r/m32** position)

SIB - An optional 1-byte value immediately following a **ModR/M** byte that identifies the scale, index register, and base register for those instructions requiring it. This is only necessary for instructions that have a scale (and some instructions that use the **esp** register). Similar to the **ModR/M** byte, the first 2 bits represent the **S**cale, the next 3 bits represent the **I**ndex register (i.e. the register that is multiplied by either 1, 2, 4 or 8) and the last 3 bits represent the **B**ase register (i.e. the register ***without*** a scaling factor).

See the section on **SIB** for how to interpret and encode this byte.

Label - An optional identifier which is followed by a colon.

Mnemonic - A reserved name for a class of instruction opcodes, which have the same function. In other words, it is the human form of an instruction that represents an opcode. For example, the mnemonic add is the human readable form of the opcode **0x03** amongst others.

```
label1: mnemonic operand1, operand2, operand3
```

Immediate - An 8-bit, 16-bit, 32-bit or 64-bit value that represents a literal number. For example, in the following instruction, **0x11223344** is an immediate value, but **0xaabbccdd** is not (it is a displacement).

```
mov dword [ ecx + 0xaabbccdd ], 0x11223344
```

Displacement - An 8-bit, 16-bit, or 32-bit value that represents either a memory location or an offset from a memory location. In the following examples, **0xaabbccdd** is a displacement, but **0x11223344** is not (it is an immediate).

```
mov dword [ ecx + 0xaabbccdd ], 0x11223344
mov dword [ 0xaabbccdd ], 0x11223344
```

AT&T Syntax - A representation of assembly language in which the first operand is the source operand and the second operand is the destination operand. It is often easily recognizable as this representation uses the **%** character to precede registers. This is often the default for **objdump**.

```
movl %ecx, %edx
```

The above example reads the contents of the **ecx** register and stores this value into the **edx** register.

Intel Syntax - A representation of assembly language in which the first operand is the destination operand and the second operand is the source operand. This is typically the default for IDA Pro and **ndisasm**. This is the syntax we will be using.

```
mov edx, ecx
```

The above example reads the contents of the **ecx** register and stores this value into the **edx** register. It is the same instruction described in the AT&T section above.

Stack - The stack is a Last In First Out (LIFO) data structure. In the context of Intel, the **esp** register points to the current stack location. The stack pointer can constantly change due to calls to functions or using instructions such as **push** and **pop** to alter the stack contents. See the section "Stack Usage" below for more information.

Two's Complement - This is the representation by which integers are stored in the Intel Architecture. Non-negative numbers are in the range **0x00000000** through **0x7FFFFFFF**. Negative numbers are in the range **0x80000000** - **0xFFFFFFFF**. For example, the number **5** is represented as **0x00000005**. In binary, this is

0b0000 0000 0000 0000 0000 0000 0000 0101

To determine the value of **-5**, we take **5** (in binary) and flip all the bits [this step forms the ones' complement]. Doing so yields

0b1111 1111 1111 1111 1111 1111 1111 1010

Next we add one to get the two's complement.

0b1111 1111 1111 1111 1111 1111 1111 1011

Converting back to hex, **0xFFFFFFFB**, which is -5 decimal.

2 Intel Registers (32-bit)

Registers are located on the CPU. Accessing registers is much faster than accessing memory. Thus, the more often a process can use the registers as an access/storage mechanism, generally the faster a function will execute. When discussing registers, there is no concept of endianess in the register. Thus, if we say that a register contains the value **0x41424344**, it is always the value **0x41424344** no matter which architecture we are discussing.

eip - Extended Instruction Pointer. The **eip** register is a special, reserved register on the Intel processor that contains a pointer to the currently executing instruction. As an instruction is executed, the CPU updates the **eip** register accordingly so it can fetch/decode/execute the next instruction(s). The 32-bit architecture does not allow direct access of **eip**, **so instructions such as this are not valid**.

```
mov eip, 0x41414141
mov eax, eip
push eip
```

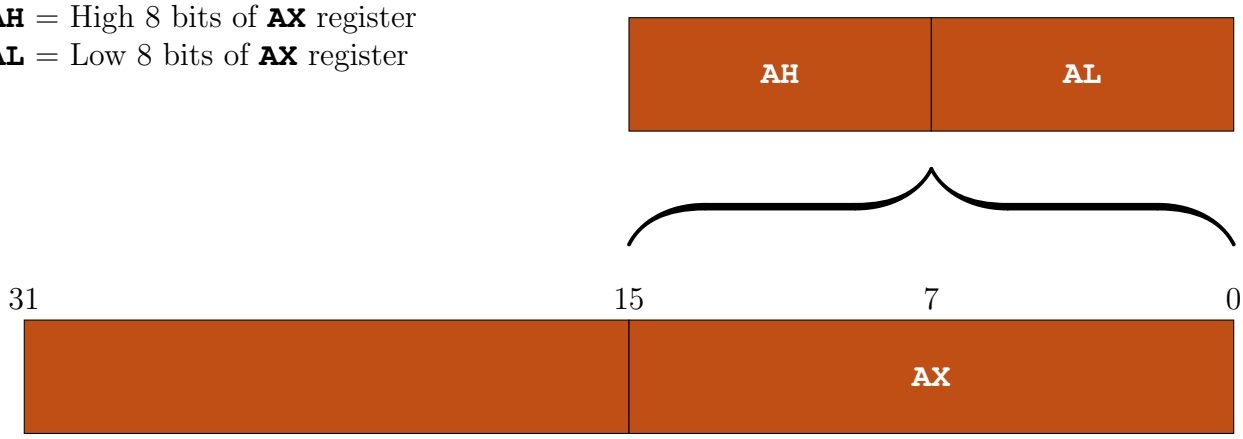
Numeric	Human	Purpose	Save by Convention[2]
000	EAX	EAX is used as a return value. It is also often used as an **a**ccumulation register.	NO
001	ECX	ECX is used as a **c**ounter register. Some instructions (e.g. **rep movsb**) require ECX to be set prior to invocation.	NO
010	EDX	EDX is an extension to the accumulator. For example, **imul eax**, will update both EAX and EDX register. Otherwise, EDX can be used as a general-purpose register.	NO
011	EBX	EBX had a special purpose in 16-bit mode, but in 32-bit mode it can be used as a general-purpose register.	SAVE (generally)
100	ESP	This is the **s**tack register. **push**/**pop** instructions implicitly use ESP as the register that will be used to store/remove values to/from the stack.	SAVE
101	EBP	This is the **b**ase **p**ointer register. Often times it is used to build a **stack frame**.	SAVE
110	ESI	This is the **s**ource **i**ndex register. In certain instructions (e.g. **movsb**), the ESI register should point to the source buffer.	SAVE
111	EDI	This is the **d**estination **i**ndex register. In certain instructions (e.g. **movsb**), the EDI register should point to the destination buffer.	SAVE

Figure 2: General Purpose Registers

[2]By convention, the registers labeled SAVE should be saved by your assembly code and restored to the original value before returning control to the calling routine. Note that, when compiling code the compiler does this automatically. When writing assembly without the aid of a compiler, the author must perform the save and restore steps.

The low 16-bits of every General Purpose Register (GPR) can be accessed by removing the leading "**e**" from the register name. Thus, **ax**, **cx**, **dx**, **bx**, **sp**, **bp**, **si**, **di** are all valid register names. However, only **eax**, **ecx**, **edx** and **ebx** can reference the high/low 8 bits. Thus, **al**/**ah**, **cl**/**ch**, **dl**/**dh**, **bl**/**bh** are all valid references, but **esp**, **ebp**, **esi** and **edi** do not have such equivalents.

EAX = Extended Accumulation Register (32 bits)
AX = Accumulation Register (16 bits)
AH = High 8 bits of **AX** register
AL = Low 8 bits of **AX** register



Let's assume **EAX = 0x41424344**. This implies that

AX = 0x4344
AH = 0x43
AL = 0x44

Other Registers [See section "VI: Processor Memory Modes and Protection Levels (Rings)" for more information]

CS - Code Segment Register. Anytime code is accessed, it is accessed using the special segment register **CS**.

DS - Data Segment Register. Generally, anytime data not on the stack is accessed, it is done via the special segment register **DS**. Some instructions use **ES** as a default segment.

SS - Stack Segment Register. Anytime data on the stack is accessed, it is done via the special segment register **SS**.

ES - Extra Segment Register. Some instructions use **ES** as a default segment. An example is the movsb instruction where **edi** is referenced from the **ES** segment.

FS - Extra Segment Register. The **FS** register is used heavily by the Microsoft compiler and by the GNU compiler for Thread Local Storage (TLS). It stores local thread information in this register for quick, efficient access.

GS - Extra Segment Register. When used, the **GS** register is used for a similar purpose as the **FS** register.

EFLAGS - System, Status and Control flags are all found in the **EFLAGS** register. Note the Carry Flag (**CF**) and Zero Flag (**ZF**) are stored here. Certain instructions have conditional behavior based on these flags. For example the **jz** instruction will jump to the specified location only if the **ZF** is set. Otherwise, the jump will be skipped.

CR0 - **CR4** - Control Registers 0 through 4. **CR0** controls processor features including write-protect which, when enabled, prevents the CPU from writing to read-only memory. **CR3** is used to store the Page Map Table database. **CR4** controls various CPU operations including whether or not Physical Address Extensions (PAE) are enabled.

DR0 - **DR7** - Debug Registers 0 through 7. **DR0** through **DR3** can be used to set a breakpoint. Thus, there are only 4 hardware breakpoints that can be set. The bits in **DR7** define the behavior of these breakpoints. **DR6** is used as the debug status register helping the debug handler identify which condition was hit. When the Debug Extensions are enabled (the **CR4** register determines this), **DR4**/**DR5** are mapped to **DR6**/**DR7**.

3 Intel Instruction Format

The Intel instructions have a variable length format. The general format for the instructions can be seen in Figure 3. Two of the most important concepts in the instruction format are the **opcode** and the **MODR/M** byte. The opcode can consist of 1-3 bytes and can be looked up in the Intel Instruction Manual. The **MODR/M** byte deserves some extra attention. The entire instruction can be a maximum size of 15 bytes. Figure 3 shows the exact encoding of the instruction. That is, if both a displacement and an immediate exist, the order is displacement followed by immediate.

Figure 3: General Intel Instruction Format

Instruction prefixes can be in any order. They are typically divided into 4 groups. An instruction should contain up to 1 prefix from each group. That is, an instruction would typically not have 2 prefixes from Group 1.

GROUP	Description	Notes
1	Lock and Repeat prefixes	LOCK = 0xF0 REPNE/REPNZ = 0xF2 REP/REPE/REPZ = 0xF3 [3]
2	Segment Override Branch hints (no longer wideley used)	0x2E - CS segment override 0x36 - SS segment override 0x3E - DS segment override 0x26 - ES segment override 0x64 - FS segment override 0x65 - GS segment override 0x2E - Branch not taken (**jcc**) 0x3E - Branch taken (**jcc**)
3	Operand-size override prefix	0x66 - Overrides the size of an operand (e.g. switch between 16- and 32-bit operand sizes)
4	Address-size override prefix	0x67 - Overrides the size of an address (e.g. switch between 16- and 32-bit address sizes)

[3] The REP prefixes are only intended for string and input/output instructions.

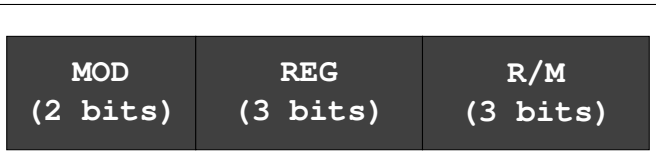


Figure 4: The MODR/M Byte

Note: The "r/m" below corresponds to the "R/M" field in the MODR/M figure above.

MOD	Assembly Syntax	Explanation and Samples
00	**[r/m]**	The **r/m32** operand's memory address is located in the **r/m** register. **mov [edi], eax** **mov edi, [eax]** **mov [edi], 0x11223344** **push [edi]**
00	**[disp32]**	SPECIAL CASE: If the MOD is 00 **and** the R/M value is 101, this is a special case. This indicates the **r/m32** location is a **memory location that is a displacement32 only**. **mov [0x11223344], eax** **mov eax, [0x11223344]** **mov [0x11223344], 0x55667788** **push [0x11223344]**
01	**[r/m + byte]**	The **r/m32** operand's memory address is located in the **r/m** register + a 1-byte displacement. **mov [eax + 4], esi** **mov esi, [eax + 4]** **push [edx + 4]**
10	**[r/m + dword]**	The **r/m32** operand's memory address is located in the **r/m** register + a 4-byte displacement. **mov [eax + 0x11223344], esi** **mov esi, [eax + 0x11223344]** **push [edx + 0x11223344]**
11	**r/m**	The **r/m32** operand is a direct register access. **mov eax, edi**

Figure 5: The mod bits of the MODR/M byte (first 2 bits)

A common question that is asked: **Which instruction encodings require the MODRM byte?**

Most instruction encodings require the MODR/M byte.

M	1st operand memory access (**imul [eax]**)
MI	1st operand memory access, 2nd operand immediate (**add [eax], 1**)
MR	1st operand memory, 2nd operand register (**add [eax], ecx**)
RM	1st operand register, 2nd operand memory access (**add ecx, [eax]**)
RMI	1st operand register, 2nd operand memory, 3rd operand immediate (**imul eax, [ecx], 8**)

Note: **M** can be a register or memory, while **R** is explicitly a direct register access.

The following encodings **do not** require a MODR/M byte.

NP No operands
```
nop
0x90
```

O Add the register number to the opcode
```
push ecx
0x50 + 1
0x51
```

I Operand is immediate value
```
push 4
0x68 0x04 0x00 0x00 0x00
```

OI Add the register number to the opcode and the first operand is an immediate value
```
mov ecx, 4
0xb8 + 1 0x04 0x00 0x00 0x00
0xb9 0x04 0x00 0x00 0x00
```

D What follows is a relative displacement
```
jz rel8
```
0x74 cb (8-bit relative displacement follows the opcode **0x74**)

3.1 Sample MODR/M Encodings

Let's break down the **mov** and **add** instructions for four different formats: one with both operands being direct register access, one with one of the operands used as an indirect memory access from a register, one of the operands used as an indirect memory access with a displacement and a register, and a final instruction with an immediate as its second operand.

3.1.1 Both operands being direct register access

mov esi, ebx

This can be encoded as either **mov r32, r/m32** or **mov r/m32, r32**.

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.
2. We look for the syntax that matches ours (i.e. both operands are 32-bit registers). We see that there are two opcodes that match (**0x89** and **0x8b**). The **0x89** opcode has the syntax **r/m32, r32** whereas the **0x8b** has the syntax **r32, r/m32**. When referencing memory (which this first example does not), the **0x89** opcode can be used to write to memory while the **0x8b** opcode can be used to read from memory. Since both operands are direct register access, we can choose either. Let's use **0x8B** as our example.
3. Looking at the Op/En we see that the opcode **0x8B** has an Op/En of **RM** (**RM** indicates the second operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.
4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **ebx** (in our case because we chose **0x8b**, the syntax is **mov r32, r/m32**, thus our **MOD** bits are determined by the **r/m** portion) is a direct register access, we set the top 2 bits to be **0b11**. The next 3 bits indicate the destination register (in this case **esi**). Looking at the table from class, **esi** is **0b110**. Finally **ebx** is **0b011**. Thus, the MODR/M byte is **11110011** or **0xF3**.
5. The final instruction is **0x8B 0xF3**.

3.1.2 One operand (the r/m32 one) is indirect memory access

mov [esi], ebx
This is encoded as **mov [r/m32], r32**.

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.
2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is direct register). We see that the **0x89** opcode has the syntax **r/m32, r32** whereas the **0x8b** has the syntax **r32, r/m32**. The **0x8B** syntax will not work since the first operand is a direct register access (**r/m32** is the Intel syntax for register or memory access while **r32** is direct register access). Thus, we must choose the **0x89** opcode.
3. Looking at the Op/En we see that the opcode **0x89** has an Op/En of **MR** (**MR** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.
4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **[ESI]** is an indirect memory access with no offset, we set the top 2 bits to be **0b00** (See Figure 5 for clarification). The next 3 bits indicate the source register (in this case **ebx**). Looking at the table from class, **ebx** is **0b011**. Finally **esi** is **0b110**. Thus, the MODR/M byte is **00011110** or **0x1E**.
5. The final instruction is **0x8B 0xF3**.

3.1.3 add with immediate operand

add edi, 0x11223344
This is encoded as **add r/m32, imm32**.

1. We look up the **add** instruction in the manual and see that it is on page Vol 2A 3-31.
2. We look for the syntax that matches ours (i.e. the left operand is a register access while the right operand is an immediate value). We see that the **0x81** opcode has the syntax **r/m32, imm32**. It is the only available option to us, so we must choose the **0x81** opcode.
3. Looking at the Op/En we see that the opcode **0x81** has an Op/En of **MI** (**MI** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2A 3-31, we see the MODR/M byte is required.
4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **edi** is a direct register access, we set the top 2 bits to be **0b11** (See Figure 5 for clarification). The next 3 bits have a special meaning in this case. Looking back at the opcode we chose, it has a syntax we have not seen. What does the **/0** mean? Previously we had seen **/r**. The **/r** indicates the operand in the **r32** spot will be encoded into the REG bits. The same is true for the **/digit** (in this case **/0**). This value extends the opcode. In these cases, the choice is made for us. We simply have to convert this digit to its 3-bit representation, in this case **0b000** and place it in the REG field. That leaves one field remaining. This encoding uses

the first operand as **r/m32**, thus it goes in the R/M field of the MODR/M byte. Thus, the MODR/M byte is **11000111** or **0xC7**. The last item is to encode the **immediate** value as little-endian.

5. The final instruction is **0x81 0xC7 0x44 0x33 0x22 0x11**.

3.1.4 r/m32 operand is [disp32]

mov [0x11223344], ebx
This is encoded as **mov [disp32], r32**.

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.
2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is direct register). We see that the **0x89** opcode has the syntax **r/m32, r32** whereas the **0x8b** has the syntax **r32, r/m32**. The **0x8B** syntax will not work since the first operand is a direct register access (**r/m32** is the Intel syntax for register or memory access while **r32** is direct register access). Thus, we must choose the **0x89** opcode.
3. Looking at the Op/En we see that the opcode **0x89** has an Op/En of **MR** (**MR** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.
4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **[disp32]** is an indirect memory access only, we set the top 2 bits to be **0b00** and we set R/M to **0b101** (See Figure 5 for clarification). The next 3 bits indicate the source register (in this case **ebx**). Looking at the table from class, **ebx** is **0b011**. Thus, the MODR/M byte is **00011101** or **0x1D**.
5. Since there is a **disp32**, we must encode this next. This is simply **0x44 0x33 0x22 0x11**.
6. The final instruction is **0x89 0x1D 0x44 0x33 0x22 0x11**.

3.1.5 r/m32 operand indirect memory access with disp8

mov [esi + 0x04], ebx
This is encoded as mov [r/m32 + disp8], r32.

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.
2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is direct register). We see that the **0x89** opcode has the syntax **r/m32, r32** whereas the **0x8b** has the syntax **r32, r/m32**. The **0x8B** syntax will not work since the first operand is a direct register access (**r/m32** is the Intel syntax for register or memory access while **r32** is direct register access). Thus, we must choose the **0x89** opcode.
3. Looking at the Op/En we see that the opcode **0x89** has an Op/En of **MR** (**MR** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.
4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **[ESI + 0x4]** is an indirect memory access with an 8-bit offset, we set the top 2 bits to be **0b01** (See Figure 5 for clarification). The next 3 bits indicate the source register (in this case **ebx**). Looking at the table from class, **ebx** is **0b011**. Finally, **esi** is **0b110**. Thus, the MODR/M byte is **01011110** or **0x5E**.
5. Since there is a **disp8**, we must encode this next. This is simply **0x04**.
6. The final instruction is **0x89 0x5E 0x04**.

3.1.6 `add` with displacement and immediate operand

`add [ edi + 0xaabbccdd ], 0x11223344`
This is encodeded as **`add r/m32, imm32`**.

1. We look up the **`add`** instruction in the manual and see that it is on page Vol 2A 3-31.

2. We look for the syntax that matches ours (i.e. the left operand is a register access while the right operand is an immediate value). We see that the **`0x81`** opcode has the syntax **`r/m32, imm32`**. It is the only available option to us, so we must choose the **`0x81`** opcode.

3. Looking at the Op/En we see that the opcode **`0x81`** has an Op/En of **`MI`** (**`MI`** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.

4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **`edi`** is an indirect memory access with a 32-bit displacement (**`0xaabbccdd`**), we set the top 2 bits to be **`0b10`** (See Figure 5 for clarification). The next 3 bits have a special meaning in this case. Looking back at the opcode we chose, it has a syntax we have not seen. What does the **`/0`** mean? Previously we saw **`/r`**. The **`/r`** indicates the **`r32`** operand will be encoded into the REG bits. The same is true for the **`/digit`** (in this case **`/0`**). This value **extends** the opcode. In these cases, the choice is made for us. We simply have to convert this digit to its 3-bit representation, in this case **`0b000`** and place it in the REG field. That leaves one field remaining. This encoding uses the first operand as **`r/m32`**, thus it goes in the R/M field of the MODR/M byte. Thus, the MODR/M byte is **`10000111`** or **`0x87`**. We next have to encode the **displacement** value as little-endian. And the last item is to encode the **immediate** value as little-endian.

5. The final instruction is **`0x81 0x87 0xDD 0xCC 0xBB 0xAA 0x44 0x33 0x22 0x11`**.

4 SIB Byte Encoding

Let's focus on the **SIB** byte. The **SIB** byte is used when scaling needs to occur or often when we are adding two registers together to calculate an address. There are other circumstances when the **SIB** is required, for example, when accessing **[ESP]** directly. We will discuss these situations in more detail later, but below is a tutorial on how to interpret the **SIB** byte.

Figure 6: The SIB Byte

SCALE	Assembly Syntax	Explanation
00	**[indexreg*1 + basereg + displacement]**	The index register is multiplied by 1. The base register followed by a displacement.
00	**[indexreg*1 + displacement]**	SPECIAL CASE: If the BASE is **0b101** and the addressing mode (MOD bits) is **0b00**, there is no base register.
01	**[indexreg*2 + basereg + displacement]**	The index register is multiplied by 2. The base register followed by a displacement.
01	**[indexreg*2 + displacement]**	SPECIAL CASE: If the BASE is **0b101** and the addressing mode (MOD bits) is **0b00**, there is no base register.
10	**[indexreg*4 + basereg + displacement]**	The index register is multiplied by 4. The base register followed by a displacement.
10	**[indexreg*4 + displacement]**	SPECIAL CASE: If the BASE is **0b101** and the addressing mode (MOD bits) is **0b00**, there is no base register.
11	**[indexreg*8 + basereg + displacement]**	The index register is multiplied by 8. The base register followed by a displacement.
11	**[indexreg*8 + displacement]**	SPECIAL CASE: If the BASE is **0b101** and the addressing mode (MOD bits) is **0b00**, there is no base register.

Figure 7: The scale bits of the SIB byte (first 2 bits)

Another common question that is asked: **Which instruction encodings require the SIB byte?**

A SIB byte must follow a MODR/M byte. Thus, if the instruction does not have a MODR/M byte it will not have a SIB byte. However, only certain encodings require the SIB byte. We know exactly which encodings require the SIB byte. Looking at Table 2-2, we see that the **`[--][--]`** nomenclature represent those encodings that require a SIB byte.

The scale refers to a register being multiplied by **`1`**, **`2`**, **`4`**, or **`8.`** These are the only possible values that a register can be multiplied (scaled) by. The reason is simple. The scale field is only 2 bits, which has 4 representations. $2^{0b00} = 1$, $2^{0b01} = 2$, $2^{0b10} = 4$, $2^{0b11} = 8$. (**Note**: the exponents are written in binary).

If both operands are direct register accesses (i.e. Addressing Mode **`0b11`**), we do not use the SIB byte. By looking at Table 2-2, we see that there are no scenarios in Addressing Mode **`0b11`** that have the **`[--][--]`** representation, thus no SIB byte is required.

First a quick tutorial on how to read Table 2-3; the SIB byte. Our first thing to note is that the **`esp`** register cannot be *scaled*. This can be seen by the **`none`** located in the **`esp`** position in each scaling mode. Thus, the following instructions are illegal:

```
mov [ esp*4 ], ecx
mov eax, [ esp*8 ]
```

You might be asking, well then how do you encode the following? We will see this later.

```
mov [ esp ], ecx
mov ecx, [ esp ]
```

4.1 Sample SIB Byte Encodings

Let's look at a few examples to see how the SIB byte is used in conjunction with the MODR/M byte to encode instructions.

4.1.1 Both operands being direct register access

`mov esi, ebx`

1. No SIB is required as there is no scaling in effect. Encode as you normally would. Note that the following are illegal instructions:

```
mov esi*2, ebx
mov esi, ebx*8
```

4.1.2 Scaling by 8 with base register

mov [esi*8 + edi], ebx

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.

2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is direct register). We see that the **0x89** opcode has the syntax **r/m32, r32** whereas the **0x8b** has the syntax **r32, r/m32**. The **0x8B** syntax will not work since the first operand is a direct register access (**r/m32** is the Intel syntax for register or memory access while **r32** is direct register access). Thus, we must choose the **0x89** opcode.

3. Looking at the Op/En we see that the opcode **0x89** has an Op/En of **MR** (**MR** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.

4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **[ESI*8]** is an indirect memory access with no offset, we set the top 2 bits to be **0b00**. The next 3 bits (the REG field) indicate the source register (in this case **ebx**). Looking at the table from class, **ebx** is **0b011**. Since **esi** is scaled (in this case, multiplied by **8),** we know we need a SIB byte. Looking at Table 2-2 with Addressing Mode **0b00**, we see that to indicate a SIB byte, we must set the R/M field to the value of **esp** (**0b100**). Don't worry; we will use the values of **esi** and **edi** in the SIB byte.

5. Up to this point, our encoding is: **0x89 0x1C**. We now need to encode the SIB byte. To do so, we look at Table 2-3 on page Vol 2A 2-7. We know the scale (SS) is **8 or** mode **0b11**. The next 3 bits represent the scaled index register; in this case **esi** (**0b110**). Finally, we see the base register (the one not scaled) is **edi** (**0b111**). Thus, our SIB byte (in binary) is **11110111** or **0xF7**.

6. The final encoding is **0x89 0x1C 0xF7**.

4.1.3 Scaling by 4 with base register, displacement and immediate

mov [esi*4 + edi + 0xaabbccdd], 0x11223344

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.

2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is an immediate). We see that the **0xC7 /0** opcode has the syntax **r/m32, imm32**. Thus, we must choose the **0xC7** opcode.

3. Looking at the Op/En we see that the opcode **0xC7** has an Op/En of **MI** (**MI** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.

4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **[ESI*4 + EDI + 0xaabbccdd]** is an indirect memory access with a 32-bit displacement, we set the top 2 bits to be **0b10**. The next **3 bits** (the REG field) indicate the source register or opcode extension. In this case, the opcode is **0xC7 /0**, so we set the REG field to **0b000**. Since **esi** is scaled (in this case, multiplied by **4**), we know we need a SIB byte. Looking at Table 2-2 with Addressing Mode **0b10**, we see that to indicate a SIB byte, we must set the R/M field to the value of **esp** (**0b100**). Don't worry; we will use the values of **esi** and **edi** in the SIB byte.

5. Up to this point, our encoding is: **0xC7 0x84**. We now need to encode the SIB byte. To do so, we look at Table 2-3 on page Vol 2A 2-7. We know the scale (**SS**) is **4** or mode **0b10**. The next 3 bits represent the scaled index register; in this case **esi** (**0b110**). Next, we see the base register (the one not scaled) is **edi** (**0b111**). Thus, our SIB byte (in binary) is **10110111** or **0xB7**. We still have to encode the displacement followed by the immediate (yes, in that order). Don't forget this is little-endian!

6. The final encoding is **0xC7 0x84 0xB7 0xdd 0xcc 0xbb 0xaa 0x44 0x33 0x22 0x11**. This is an 11-byte instruction!

4.1.4 Scaling by 2 with base register, no displacement

`lea ecx, [ ebx + ebp*2 ]`

1. We look up the **`lea`** instruction in the manual and see that it is on page Vol 2A 3-580. This is known as the "load effective address" instruction. Unlike other instructions, when the brackets (**`[ ]`**) are used, we do not dereference the memory location, but rather simply perform the math operations within the brackets. Thus, the instruction above will multiply the **`ebp`** register by **`2,`** add it to the **`ebx`** register and store the result in **`ecx`**. This instruction is used to calculate memory addresses as well as perform some simple optimized math functions.

2. We look for the syntax that matches ours (i.e. the left operand is a register while the right operand is memory location). We see that the **`0x8D /r`** opcode has the syntax **`r32, m`**. Thus, we must choose the **`0x8D`** opcode.

3. Looking at the Op/En we see that the opcode **`0x8D`** has an Op/En of RM (RM indicates the first operand should go in the REG location). Looking at the Instruction Operand Encoding table on page Vol 2A 3-580, we see the MODR/M byte is required.

4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **`[EBX + EBP*2]`** is an indirect memory access with no offset, we set the top 2 bits to be **`0b00`**. The next 3 bits (the REG field) indicate the destination register (in this case **`ecx`**). Looking at the table from class, **`ecx`** is **`0b001`**. Since **`ebp`** is scaled (in this case, multiplied by **`2`**), we know we need a SIB byte. Looking at Table 2-2 with Addressing Mode **`0b00`**, we see that to indicate a SIB byte, we must set the R/M field to the value of **`esp`** (**`0b100`**). Don't worry, we will use the value of **`ebp`** and **`ebx`** in the SIB byte.

5. Up to this point, our encoding is: **`0x8D 0x0C`**. We now need to encode the SIB byte. To do so, we look at Table 2-3 on page Vol 2A 2-7. We know the scale (**`SS`**) is **`2 or`** mode **`0b01`**. The next 3 bits represent the scaled index register; in this case **`ebp`** (**`0b101`**). Finally, the base register is **`ebx`** (**`0b011`**).

6. The final encoding is **`0x8D 0x0C 0x6B`**. There is neither a displacement nor an immediate value.

4.1.5 Scaling without a base register

`mov [ esi*4 ], 0x11223344`

Note: We can rewrite this as: **`mov [ esi*4 + 0x00000000 ], 0x11223344`**. We will see this variation come in handy later.

1. We look up the **`mov`** instruction in the manual and see that it is on page Vol 2B 4-35.

2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is an immediate). We see that the **`0xC7 /0`** opcode has the syntax **`r/m32, imm32`**. Thus, we must choose the **`0xC7`** opcode.

3. Looking at the Op/En we see that the opcode **`0xC7`** has an Op/En of **`MI`** (**`MI`** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.

4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **`[ESI*4]`** is an indirect memory access without a displacement, we set the top 2 bits to be **`0b00`**. The next 3 bits (the REG field) indicate the source register or opcode extension. In this case, the opcode is **`0xC7 /0`**, so we set the REG field to **`0b000`**. Since **`esi`** is scaled (in this case, multiplied by **`4`**), we know we need a SIB byte. Looking at Table 2-2 with Addressing Mode **`0b00`**, we see that to indicate a SIB byte, we must set the R/M field to the value of **`esp`** (**`0b100`**). Don't worry; we will use the value of **`esi`** in the SIB byte.

5. Up to this point, our encoding is: **`0xC7 0x04`**. We now need to encode the SIB byte. To do so, we look at Table 2-3 on page Vol 2A 2-7. We know the scale (**`SS`**) is **`4 or`** mode **`0b10`**. The next 3 bits represent the scaled index register; in this case **`esi`** (**`0b110`**). Now, we see there is no base register. To handle this situation, we look at the top row of the Table 2-3 and notice the **`ebp`** register as a base register indicates something special. Looking at the note at the bottom of the table, we see addressing mode **`0b00`** without a base register must have a 32-bit displacement. Remember our note from above? We can use a 32-bit **`0x00000000`** as the displacement. Thus, our SIB byte (in binary) is **`10110101`** or **`0xB5`**. We still have to encode the displacement followed by the immediate (yes, in that order). Don't forget this is little-endian!

6. The final encoding is
   `0xC7 0x04 0xB5 0x00 0x00 0x00 0x00 0x44 0x33 0x22 0x11`. This is another 11-byte instruction!

4.1.6 Encoding simple esp

mov [esp], ecx

Note: We can think of **[esp]** with a base register of **esp** and no scaled register.

1. We look up the **mov** instruction in the manual and see that it is on page Vol 2B 4-35.

2. We look for the syntax that matches ours (i.e. the left operand is a memory access while the right operand is direct register). We see that the **0x89** opcode has the syntax **r/m32, r32** whereas the **0x8b** has the syntax **r32, r/m32**. The **0x8B** syntax will not work since the first operand is a direct register access (**r/m32** is the Intel syntax for register or memory access while **r32** is direct register access). Thus, we must choose the **0x89** opcode.

3. Looking at the Op/En we see that the opcode **0x89** has an Op/En of **MR** (**MR** indicates the first operand should go in the R/M location). Looking at the Instruction Operand Encoding table on page Vol 2B 4-35, we see the MODR/M byte is required.

4. Using our diagram from class, we know that the top 2 bits of the MODR/M byte are used to determine if this is a memory offset or direct register access. Since **[ESP]** is an indirect memory access with no offset, we set the top 2 bits to be **0b00**. The next 3 bits (the REG field) indicate the source register (in this case **ecx**). Looking at the table from class, **ecx** is **0b001**. Next, we try to encode **esp** in the R/M field. Looking at Table 2-2, we see that we cannot simply encode **esp** because the **esp** register value is used to indicate a SIB byte. We will still encode **esp**, into the R/M location because we will be using the SIB byte. Looking back to our note, if we think of the first operand as being the base register (**esp**) and no scaled register, we can use Table 2-3 to encode the SIB byte. Since **esp** is the base register the last 3 bits should be encoded as **0b100**. Since there is no scaling, we look at the first column and see that none indicates there is no scaling and is encoded as **esp** (**0b100**). This is also why **esp** cannot be the scaled register! Finally, we need to encode the scale (**SS**). Look at the table again, we see that it doesn't matter which scale we choose, because in each case **esp** translates to no scale. Thus, we can pick any of the **SS** modes. Let's choose **0b11**. Thus, our final SIB byte (in binary) is **11100100** or **0xE4**.

5. The final encoding is **0x89 0x0C 0xE4**. But the last byte can also be **0x24**, **0x64**, or **0xA4**. As discussed, some instructions have multiple encodings!

4.1.7 Branching Instructions (`rel32`)

```
offset_01010000:
01010000:  C3              retn
01010001:  90              nop
01010002:  90              nop
01010003:  90              nop
01010004:  90              nop
01010005:  90              nop
01010006:  90              nop
01010007:  90              nop
01010008:  E9 F3 FF FF FF  jmp offset_01010000
```

Note: The same reasoning below is true for the **`call rel32`** instruction as well.

We will focus on the **`jmp`** at offset **`0x01010008`**. To calculate the location we perform the following steps.

1. Recognize the opcode (**`0xE9`**) is a **`jmp rel32`**.
2. Recognize the instruction (**`jmp`**) is located at address **`0x01010008`**.
3. Obtain the **`rel32`** displacement by reading in the next 4 bytes and treat them as a 32-bit value: **`0xFFFFFFF3`**.
4. Add the current instruction address (**`0x01010008`**) to the size of the instruction (1 byte opcode + 4 byte displacement = 5 bytes) and to the 32-bit displacement (**`0xFFFFFFF3`**) **`0x01010008`** + **`0x00000005`** + **`0xFFFFFFF3`** = **`0x101010000`**.
5. We recognize that this new value (**`0x101010000`**) is actually a 33-bit value. We have to take the low 32-bits only and ignore the overflow. Thus the final branch target is: **`0x01010000`**.

4.1.8 Branching Instructions (rel8)

```
offset_01010000:
01010000:  C3              retn
01010001:  90              nop
01010002:  90              nop
01010003:  90              nop
01010004:  90              nop
01010005:  90              nop
01010006:  90              nop
01010007:  90              nop
01010008:  EB F6           jmp offset_01010000
```

Again, we will focus on the **jmp** at offset **0x01010008**. To calculate the location we perform the following steps.

1. Recognize the opcode (**0xEB**) is a **jmp rel8**.
2. Recognize the instruction (**jmp**) is located at address **0x01010008**.
3. Obtain the **rel8** displacement by reading in the next byte (**0xF6**) and treat it as a sign extended 32-bit value: **0xFFFFFFF6**. How this is done is language dependent. In Java, everything is a signed value, so you can use it as is. In C, you need to make sure this value is read as a signed value (ensure you don't have the `unsigned` qualifier by it).

   Example:

   ```
   /* Assumes currentInstruction is 0x01010008 */
   char offset = currentInstruction[ 1 ];

   /* This line sign extends 0xF6 to 0xFFFFFFF6 */
   int disp32 = (int) offset;
   ```

4. Add the current instruction address (**0x01010008**) to the size of the instruction (2) and finally to the 32-bit displacement (**0xFFFFFFF6**)
   0x01010008 + **0x00000002** + **0xFFFFFFF6** = **0x101010000**.
5. We recognize that this new value (**0x101010000**) is actually a 33-bit value. We have to take the low 32-bits only and ignore the overflow. Thus the final branch target is: **0x01010000**.

5 Stack Usage

As mentioned, the stack is a LIFO data structure. When data are pushed onto the stack, (for example, the **push** instruction) the stack pointer is **decremented** by 4 bytes (in the 32-bit case). The value being pushed is stored at the memory location to which the **esp** register points. To remove an item from the stack, the **pop** instruction can be used. This instruction will take the value located at the memory address pointed to by **esp** and place it in the desired register (e.g. **pop ecx** would place the value in **ecx**) and then the **esp** register is **incremented** by 4 bytes. The stack has many uses, two of which are passing arguments to functions as well as saving the state of a register so it can be restored at a later time.

```
BITS 32
;
; Note that push_ecx and push_ecx2 are equivalent operations
push_ecx:
    push    ecx

push_ecx2:
    sub     esp, 4
    mov     dword [ esp ], ecx

;
; Note that pop_ecx and pop_ecx2 are equivalent operations
pop_ecx:
    pop     ecx

pop_ecx2:
    mov     ecx, dword [ esp ]
    add     esp, 4
```

Figure 8: Equivalent examples of pushing/popping items to/from the stack.

The following figures (Figure 9 - Figure 16) explain the stack layout when calling a *stdcall* function. This example also shows how the arguments are passed via the stack as well as how the return address is saved and restored using the stack.

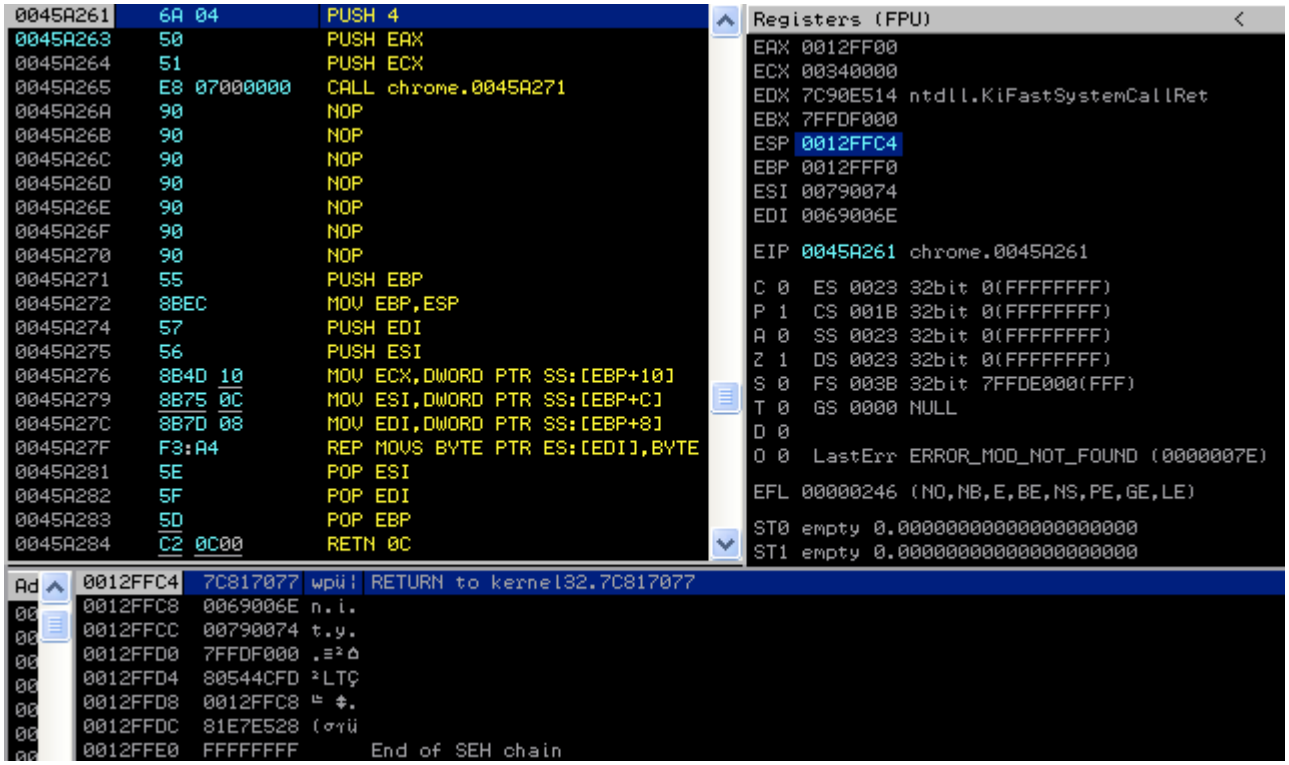


Figure 9: Beginning of call to stdcall function

In Figure 9 (and the subsequent figures in this section), the top left of the screen shows the disassembly of a function, the top right of the screen shows the current register state and the bottom of the screen shows the current state of the stack. Notice **eip** points to **0x0045A261** (**push 4** in the disassembly view) while **esp** is pointing to **0x0012FFC4**, which is the top value in the stack view.

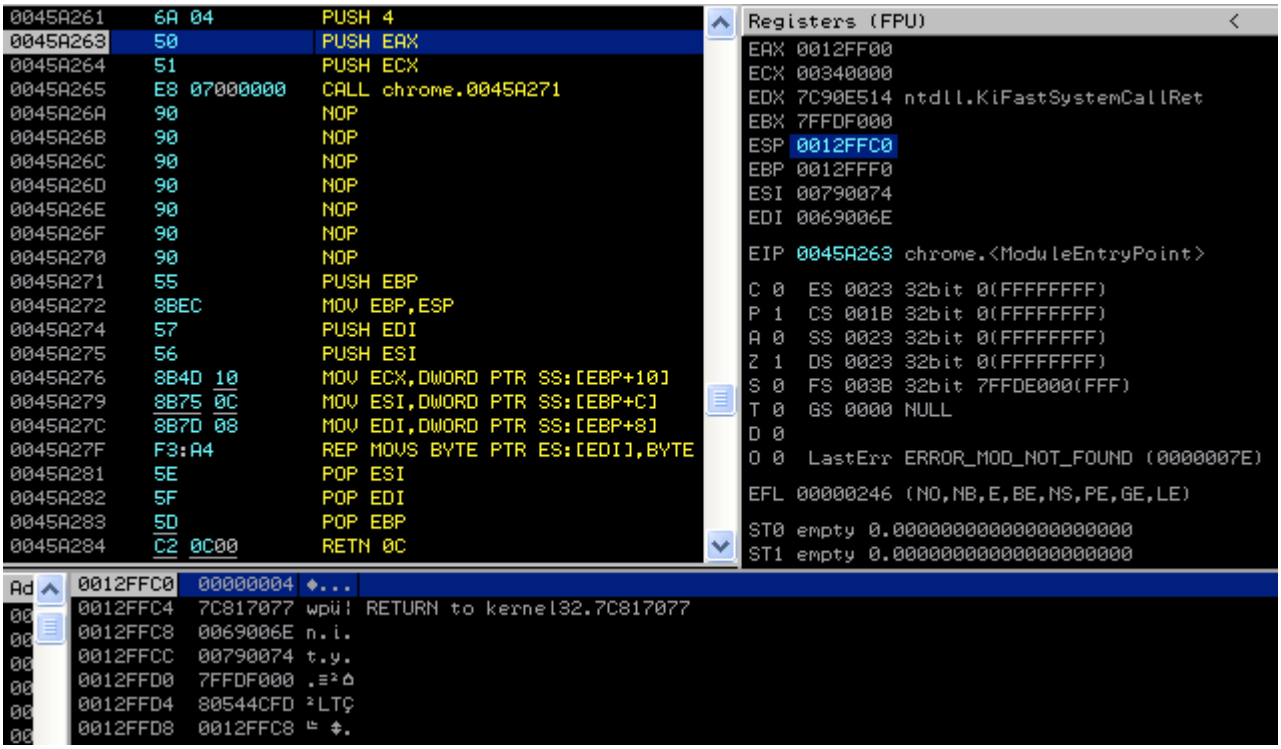


Figure 10: After **`push 4`** instruction

In Figure 10, the instruction **`push 4`** has been executed. Notice that **`eip`** has **incremented** by the size of the instruction (2 bytes in this case) and now points to the **`push eax`** instruction. In addition, **`esp`** was **decremented** by 4 bytes to account for the value **`0x00000004`** being pushed to the stack. **`esp`** now points to the location **`0x0012FFC0`** and the value at that location is **`0x00000004`**.

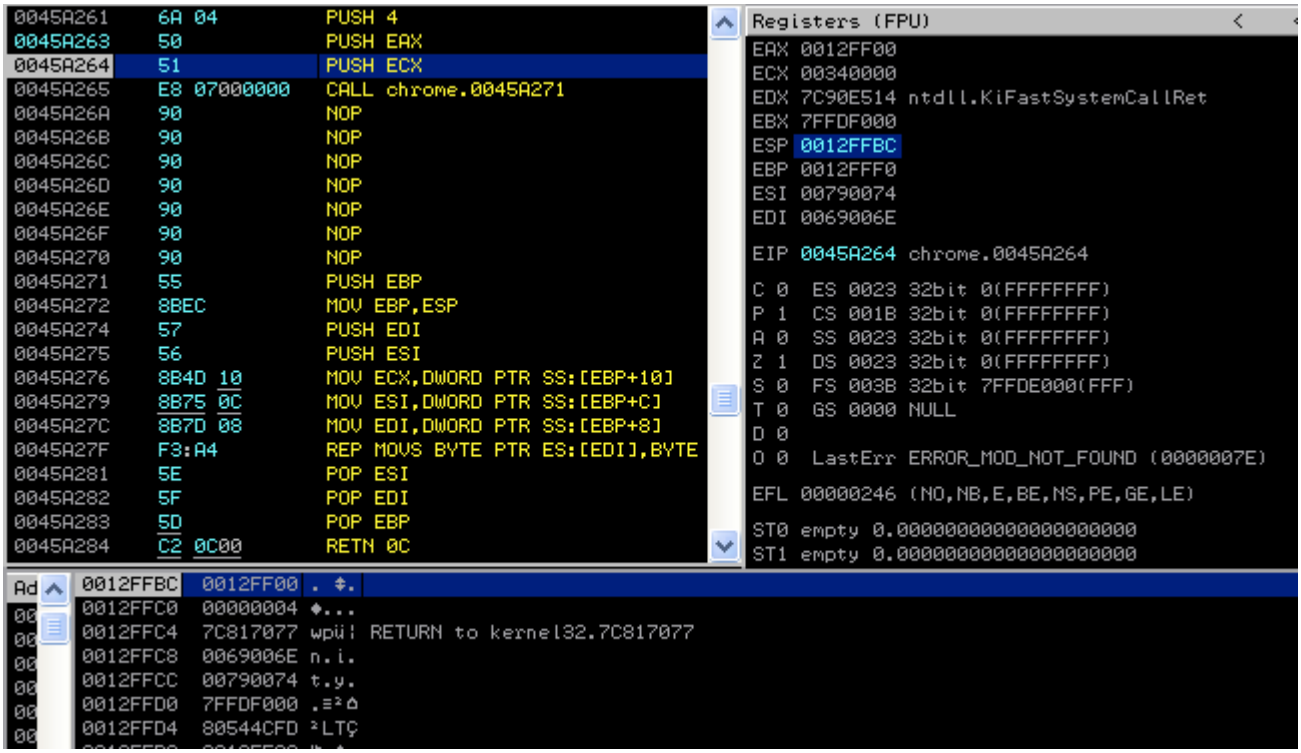


Figure 11: After **`push eax`** instruction

In Figure 11, the instruction **`push eax`** has been executed. Notice that **`eip`** has **incremented** by the size of the instruction (1 byte in this case) and now points to the **`push ecx`** instruction. In addition, **`esp`** was **decremented** by 4 bytes to account for the value of **`eax`** (**`0x0012FF00`**) being pushed to the stack. **`esp`** now points to the location **`0x0012FFBC`** and the value at that location is **`0x0012FF00`**. The value **`0x00000004`** is still on the stack and can be referenced by **`[ esp + 4 ]`**.

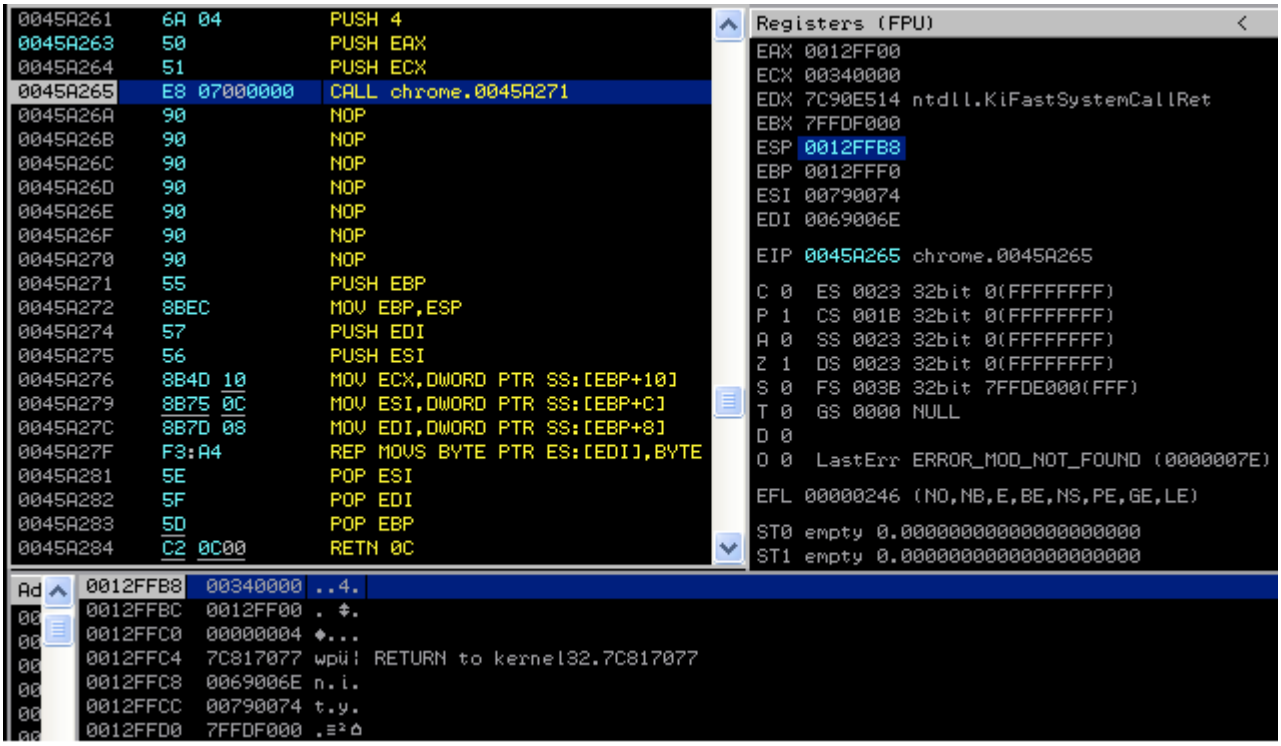


Figure 12: After **`push ecx`** instruction

In Figure 12, the instruction **`push ecx`** has been executed. Notice that **`eip`** has **incremented** by the size of the instruction (1 byte in this case) and now points to the **`call 0045A271`** instruction. In addition, **`esp`** was **decremented** by 4 bytes to account for the value of **`ecx`** (**`0x00340000`**) being pushed to the stack. **`esp`** now points to the location **`0x0012FFB8`** and the value at that location is **`0x00340000`**. The value **`0x00000004`** is still on the stack and can be referenced by **`[ esp + 8 ]`**. At this point, a call will be made. The **`call`** instruction pushes the return address onto the stack as part of its execution.

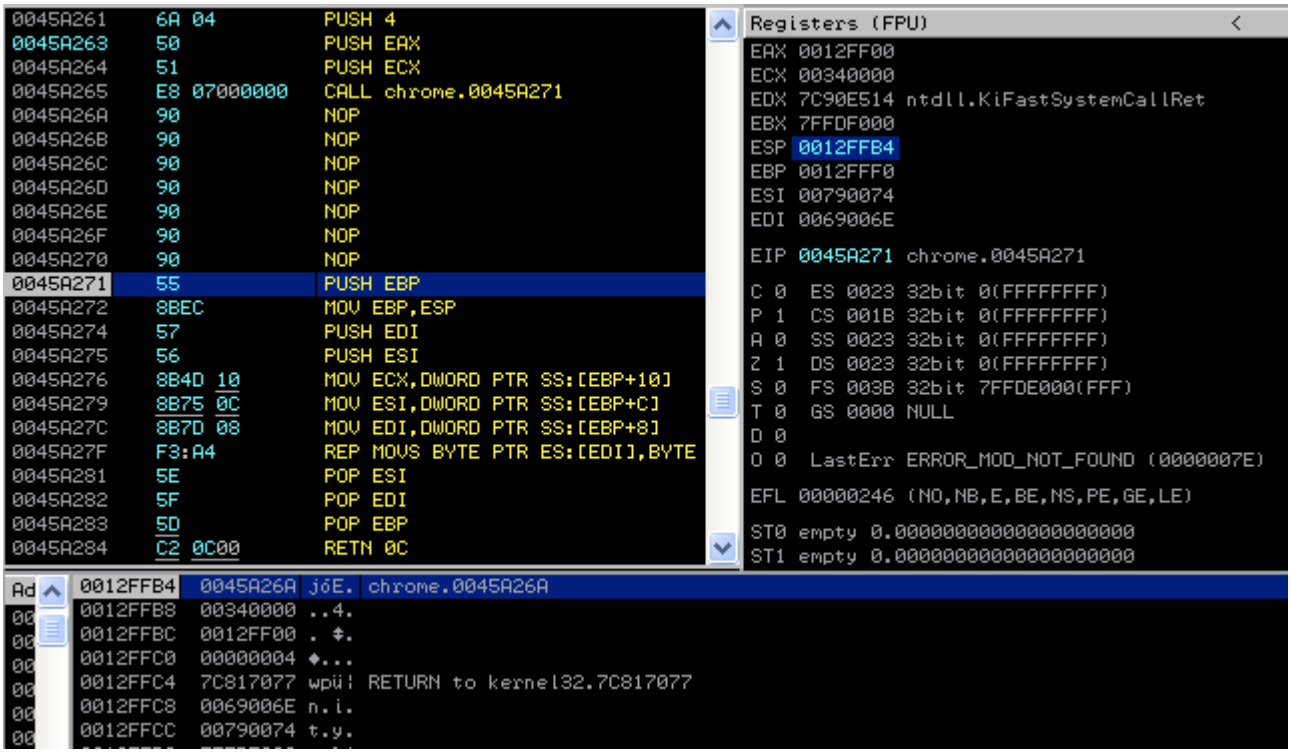


Figure 13: After the **`call`** instruction

In Figure 13, the instruction **`call 0045A271`** has been executed. Notice that **`eip`** has been **reset** to **`0x0045A271`**. The **`call`** instruction explicitly sets the **`eip`** register to the address of the function that is being called. It does **not** increment **`eip`** by the size of the instruction only, like the previous instructions had. In addition, **`esp`** was **decremented** by 4 bytes to account for the **return address** being pushed to the stack. **`esp`** now points to the location **`0x0012FFB4`** and the value at that location is **`0x0045A26A`**. The value **`0x0045A26A`** is the address of the instruction that **immediately** follows the call instruction. This is a very important concept to understand. The call instruction explicitly sets the **`eip`** register with the address of the function **and** pushes the address of the next instruction onto the stack. This is required, as the called function must have a way to return to the calling function.

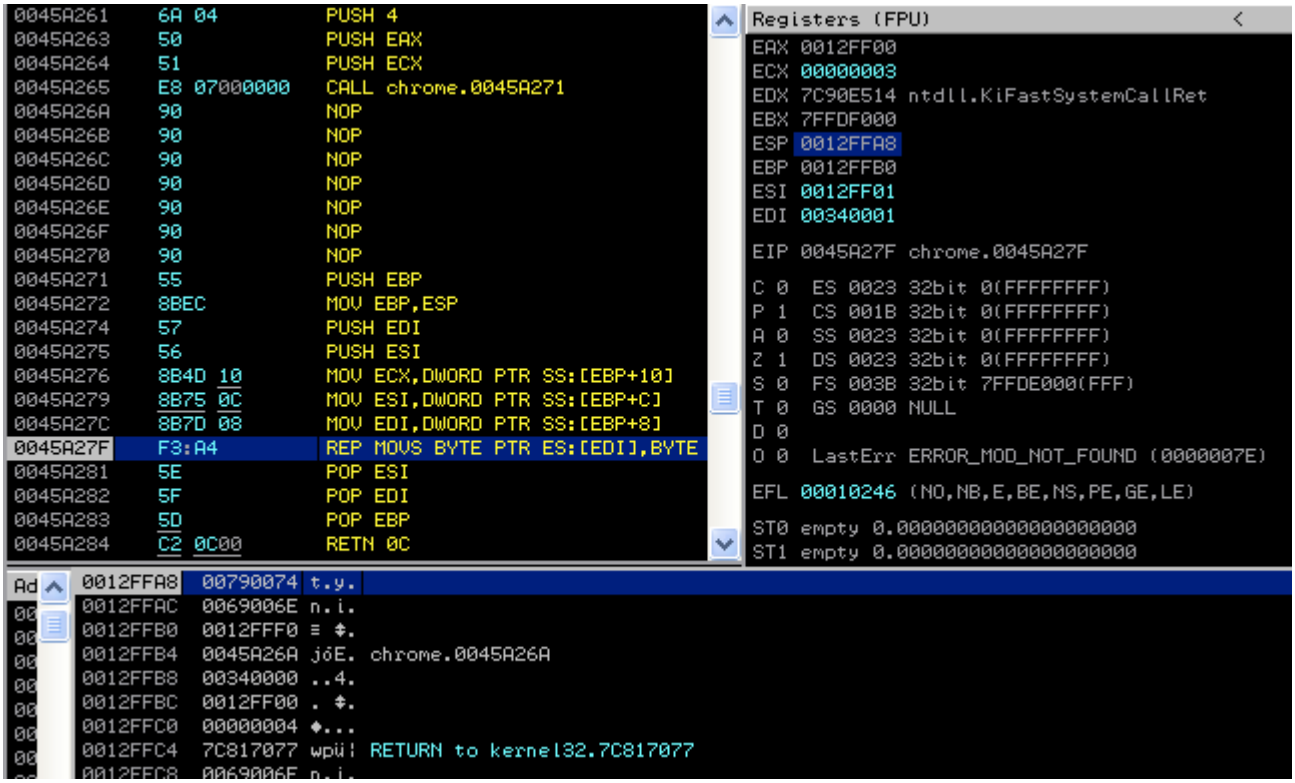


Figure 14: The **`rep movsb`** instruction

In Figure 14, the instructions from **`push ebp`** through **`mov edi, dword ptr [ebp + 8]`** have been executed. The current instruction is the **`rep movsb`** instruction. This will execute the **`movsb`** instruction the number of times indicated by the **`ecx`** register. **`movsb`** takes the value in **`esi`** and copies **`ecx`** bytes into the pointer in **`edi`**. Essentially, the function being called is a **`memcpy(dst, src, numbytes)`**. Notice the 3 references off **`ebp`** (**`0x0045A276`** through **`0x0045A27C`**). The **`ebp`** register can be used as a **base pointer register**. The instruction sequence **`push ebp`** / **`mov ebp, esp`** is generally a prologue to a function. This saves **`ebp`** to the stack (**`ebp`** is a register that must be preserved across function calls by convention). **`ebp`** is then replaced with the current stack pointer **`esp`**. This allows the **`ebp`** register to access function parameters at fixed offsets. Recall that **`esp`** can change during the function call making accessing parameters difficult to do.

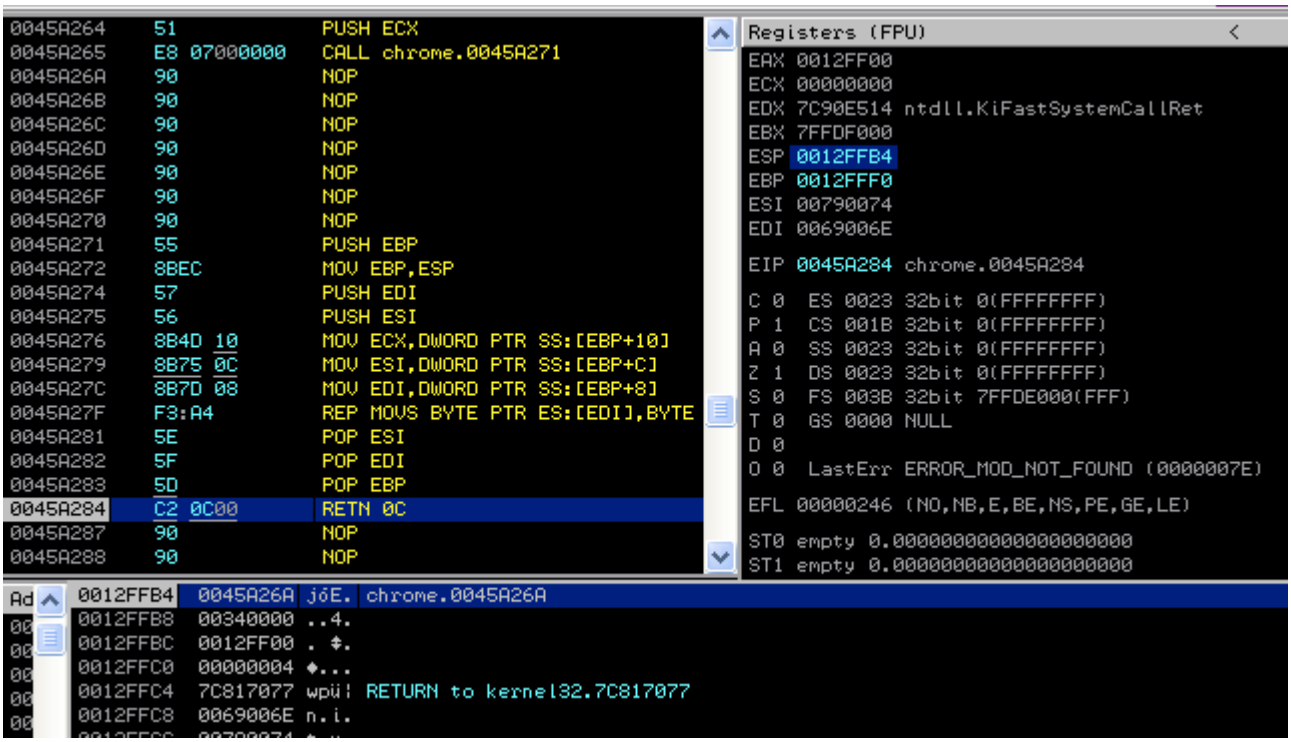


Figure 15: Stack alignment before the **`retn`** instruction

In Figure 15, the stack has been restored to the value it was upon first entering this function. Note: Compare the values in the registers in Figure 15 to those in Figure 13. Which have changed and why? The next instruction to be executed is the **`retn 0x0c`**. Upon completion of this instruction, **`eip`** will be set to **`0x0045A26A`** (the value at the top of the stack is essentially **`pop`**'d into **`eip`**; this is the return address). Next, **`0x0C`** bytes will be added to **`esp`** to ensure the **`esp`** register points to the value prior to the passing of the arguments and the return address (see Figure 9).

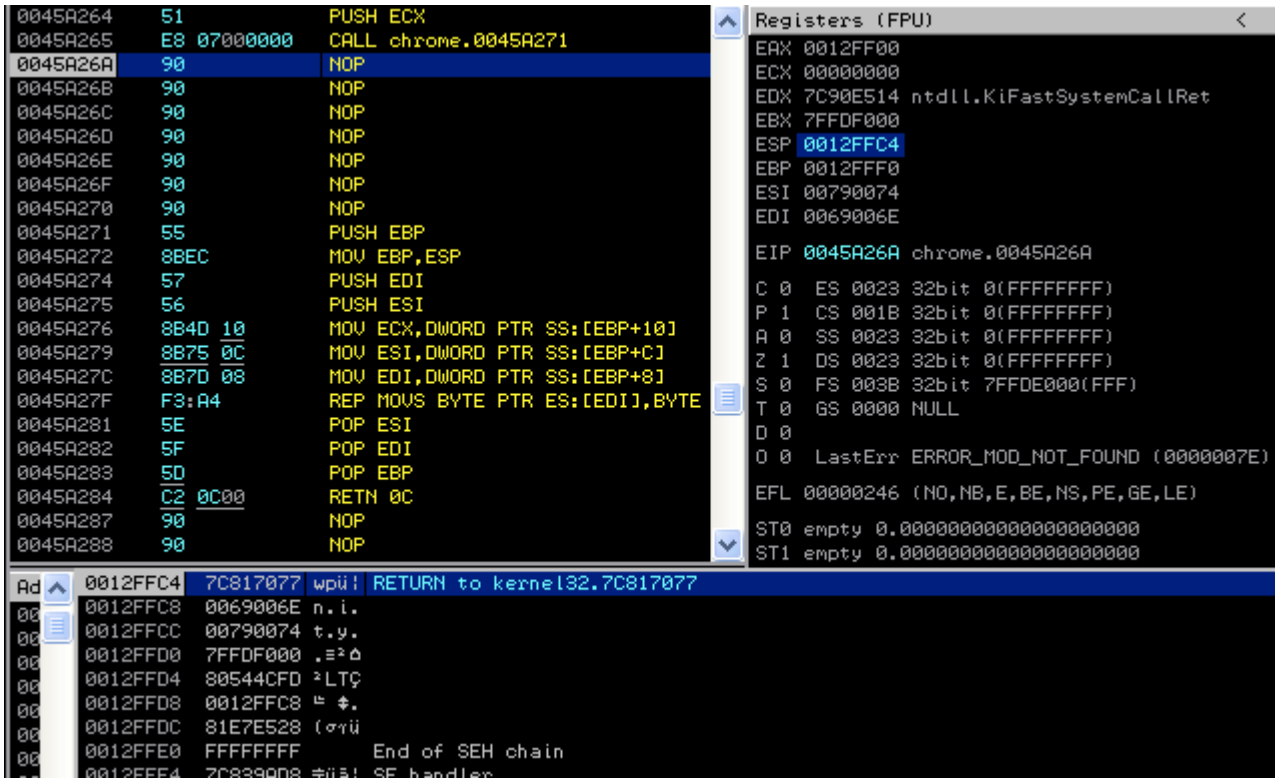


Figure 16: Successful return from **`stdcall`** function

Finally, in Figure 16, the control has been returned to the calling function. Notice how the stack has been returned to the same value it was in Figure 9. Be sure to understand how the stack went from looking like it did in Figure 15 to that in Figure 16, just by the **`retn 0xc`** instruction.

6 Calling Conventions

```
BITS 32

stdcall_asm:
    push    ebp             ; save ebp
    mov     ebp, esp        ; set ebp to the current esp
    push    edi             ; save edi
    mov     edi, dword [ ebp + 08h ]    ; get first arg
    xor     edi, dword [ ebp + 0ch ]    ; xor second arg
    mov     eax, edi        ; save result as return value in eax
    pop     edi             ; restore edi
    pop     ebp             ; restore ebp
    retn    8               ; return 8 (2 * 4) since there are 2 arguments

_start:
    push    41424344h
    push    61626364h
    call    stdcall_asm
    test    eax, eax
```

Figure 17: `stdcall` function being called

`stdcall` - With the *stdcall* calling convention, the called function is responsible for cleaning up the stack.

```
BITS 32

cdecl_asm:
    push    ebp             ; save ebp
    mov     ebp, esp        ; set ebp to the current esp
    push    edi             ; save edi
    mov     edi, dword [ ebp + 08h ]    ; get first arg
    xor     edi, dword [ ebp + 0ch ]    ; xor second arg
    mov     eax, edi        ; save result as return value in eax
    pop     edi             ; restore edi
    pop     ebp             ; restore ebp
    retn                    ; just return since this is cdecl the stack
                            ; is NOT cleaned up

_start:
    push    41424344h
    push    61626364h
    call    cdecl_asm
    add     esp, 8          ; clean up the arguments
    test    eax, eax
```

Figure 18: `cdecl` function being called

`cdecl` - With *cdecl*, the calling function is responsible for cleaning up the stack. This is the calling convention required if a function takes a variable number of arguments. In this case, the compiler knows how many arguments are passed at compile time, thus it can force the **calling** function to properly clean up the stack without modifying the **called** function.

```
BITS 32

fastcall_asm:
    xor     ecx, edx
    mov     eax, ecx
    retn                ; just return since this is fastcall the stack
                        ; is NOT cleaned up

_start:
    mov     ecx, 41424344h
    mov     edx, 61626364h
    call    fastcall_asm
    test    eax, eax
```

Figure 19: `fastcall` function being called

`fastcall` - With *fastcall*, the first two parameters are passed in registers (arg1 = **ecx**, arg2 = **edx**) and the rest are passed on the stack. *fastcall* is used when speed is of the utmost importance. Fewer memory accesses make the function perform faster.

```
BITS 32

thiscall_asm:
    mov     eax, dword [ ecx ]      ; get the virtual func table
    call    dword [ eax + 04h ]     ; call the 2nd function
    retn    8           ; return 8 (2 * 4) since there are 2 arguments

_start:
    push    41424344h
    push    61626364h
    mov     ecx, class_ptr
    call    thiscall_asm
    test    eax, eax
```

Figure 20: `thiscall` function being called

`thiscall` - The *thiscall* convention is used when making C++ method calls. When declaring a method in a C++ class, there is always an implicit argument that is passed: the *class pointer*. Depending on the compiler, this can be passed via the **ecx** register (as is the case in Windows) or passed as the first argument on the stack (as is the case with the GNU compiler). Depending on the compiler, *thiscall* may follow the *stdcall* (Windows case) or *cdecl* (GNU case) cleanup policy.

7 Processor Memory Modes and Protection Levels (Rings)

The Intel Processor can operate in either Real Mode (all addressing is segmented) or Protected Mode (where Virtual Memory and Paging can be used). Aside from bootstrap code, which typically runs in Real Mode, most code in modern operating systems runs in Protected Mode with Virtual Addressing and Paging enabled.

In addition, the Intel Processors have 4 protection levels (often called rings). Typically only 2 of these are used by operating systems (including Windows and Linux). The ring levels are Ring 0, Ring 1, Ring 2 and Ring 3. Ring 0 is the least restrictive mode, referred to as supervisory mode (or kernel mode) and ring 3 is typically called user mode. The trusted operating system code runs in Ring 0 and has direct access to all system resources. User applications run in Ring 3 and have limited access to system resources, often only through trusted system calls. Certain instructions are only accessible when the processor is running in supervisory (Ring 0) mode. For example, the clear interrupt instruction (**`cli`**) **cannot** be called from Ring 3.

The current level is often called the Current Privilege Level (CPL). There is another level called the I/O (Input/Output) Privilege Level (IOPL). The IOPL determines which rings have access to I/O ports. If the IOPL (2-bit field in **`EFLAGS`**) is set to 0, then only Ring 0 can utilize the **`IN`**/**`OUT`** instructions. If it is set to 3, then Ring 3 and below can access the instructions.

Don't worry, we will be discussing these concepts in greater detailer later in the semester!

7.1 `EFLAGS` Register

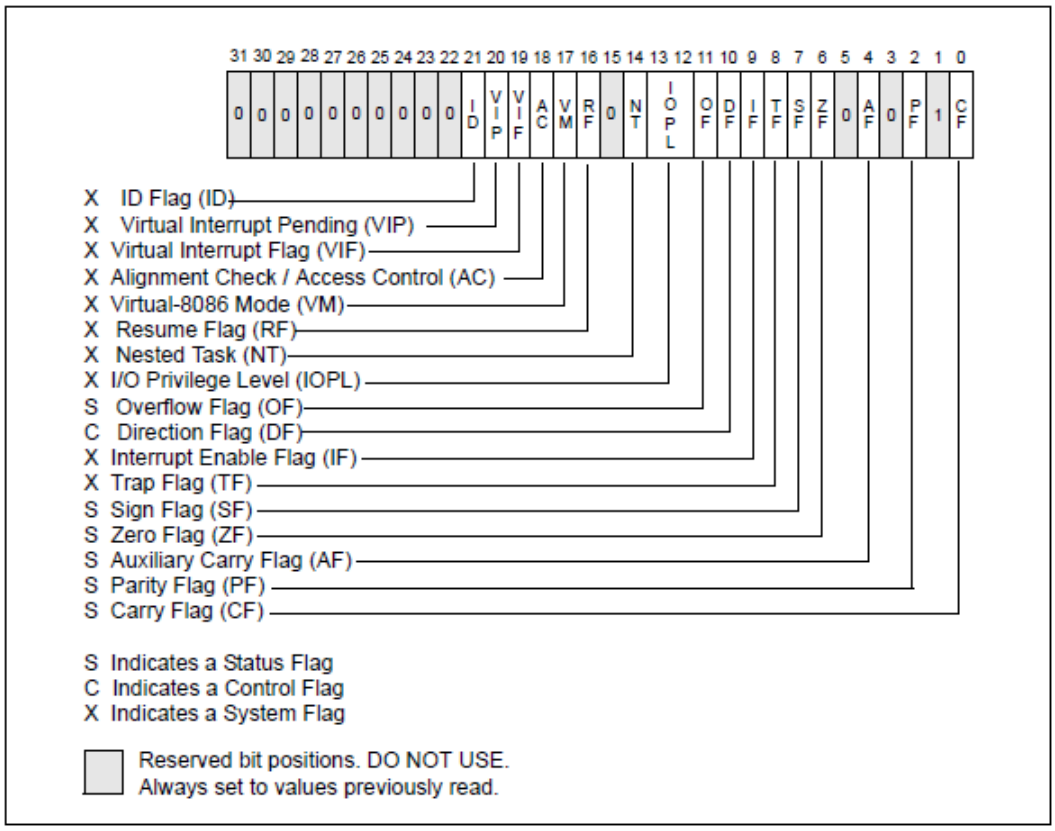


Figure 21: `EFLAGS` Register

EFLAGS is another register, but with a specific purpose. [As an aside, the 'e' in the register names stands for extended when the instruction set was extended from 16-bits to 32-bits.] The **EFLAGS** register is manipulated by certain instructions. The Intel Instruction Manual shows you which instructions modify which flags. We will discuss a few of these instructions. In general, the **EFLAGS** register is not easily accessible from higher level languages; often times you need to use assembly to access and manipulate it.

CF - This 1-bit value is the carry flag. When set, it indicates that a carry took place. For example, the **add** instruction will set the **CF** (along with the **OF**) if an overflow occurs. An overflow means the result was too large to fit into the register.

PF - A 1-bit value indicate the parity of the operation. If an operation results in an even parity (even number of 1s), **PF** is set to **1.** If the result is odd parity (odd number of 1s) it is set to **0.** For example, **xor eax, eax** will set the **PF** to **1 because** the parity of **0x00000000** is even.

ZF - A 1-bit value to indicate the result being **0.** If the result of an operation is **0,** the **ZF** is set. For example, **sub esi, esi** sets the **ZF** to **1 as** the result is **0.** This is one of the more important flags as it is used to test for **NULL** values. We will examine this later.

SF - A 1-bit value that is set if the sign of the result is set. It is a very quick way to determine if the result was positive or negative. While **sub esi, esi** sets the **SF** to **0** (the result is non-negative), a follow on **dec esi** will set **SF** to **1** as **0xFFFFFFFF** interpreted as an integer is **-1**.
IF - A 1-bit value that can only be set/unset by privileged instructions **STI**/**CLI** respectively when running in protected mode (real mode can always set/unset it).
DF - A 1-bit value that dictates which direction string operations will operate; from left-to-right or from right-to-left. If the **DF** is cleared (**CLD ; set to 0**) operations will be performed from left-to-right. If it is set to **1** (**STD**), string operations will be performed from right-to-left.
OF - A 1-bit value that signifies if an overflow has occurred, indicating the two's complement operation is too large to fit in the register.
IOPL - A 2-bit value that dictates which privilege modes can execute **IN**/**OUT** instructions.

7.2 Control Registers

On the Intel Processor (x86), there are 8 control registers, however, only 4 are really used: **CR0**, **CR2**, **CR3**, and **CR4**.

CR0 - CR0 contains flags that modify the basic operations of the processor. Some important bit values are in the table below. Note that bit 0 is the far right bit and bit 31 is the far left bit.

Bit	Description
0	**PE** (Protection Enabled) If set to **1**, the processor operates in protected mode, else it operates in real mode.
16	**WP** (Write Protect) If set to **1**, the processor cannot write to read-only memory when operating in supervisory mode (Ring 0).
31	**PG** (Paging Enabled) If set to **1**, enable paging and make use of the **CR3** register, otherwise disable paging.

CR2 - Contains the Page Fault Linear Address when a Page Fault occurs.

CR3 - This is an important register when discussing Virtual Memory. Every process has its own "view" of memory. On the Intel processor, this is implemented by utilizing the **CR3** register. Each process has its own **CR3** value unique to the process. Linear addresses can be translated to physical addresses using **CR3**. The upper 20 bits of this register are called the Page Directory Base Register (PDBR). This is the physical address of the page directory.

CR4 - This is used in protected mode to enable extended features. The below table are some more interesting bit fields from **CR4**. Note that bit 0 is the far right bit and bit 31 is the far left bit. Bits 20 and 21 are designed to make it more difficult to exploit software vulnerabilities in the kernel.

Bit	Description
5	**PAE** (Physical Address Extensions) If set to **1**, the processor uses 36-bit physical addressing instead of the normal 32-bit physical addressing.
13	**VMXE** (Virtual Machine Extensions) If set to **1**, enables the use of virtual machine extensions.
20	**SMEP** (Supervisory Mode Execution Protection) If set to **1**, attempting to execute an address from a higher numbered ring level will generate a fault. That is, if the processor is running in Ring 0 and a user mode (Ring 3) address is attempting to execute, the processor will fault.
21	**SMAP** (Supervisory Mode Access Protection) If set to **1**, attempting to access an address from a higher numbered ring level will generate a fault. That is, if the processor is running in Ring 0 and a user mode address (Ring 3) is attempting to be read/written, the processor will fault.

7.3 Segment Registers

```
ES 0023 32bit 0(FFFFFFFF)
CS 001B 32bit 0(FFFFFFFF)
SS 0023 32bit 0(FFFFFFFF)
DS 0023 32bit 0(FFFFFFFF)
FS 003B 32bit 7FFDF000(FF)
GS 0000 NULL
```

On the Intel Processor (x86), there are 6 segment registers: **CS**, **DS**, **SS**, **ES**, **FS**, **GS**. Most operations have a default segment. You can override most default segments by prepending the appropriate segment register to the instruction.

The segment registers get translated to a base address and a length. In the above example, **ES**, **CS**, **SS**, **DS** all start at address **0** and have a length of **0xFFFFFFFF**. Thus, each segment represents the same memory - that is, there really isn't that much of a difference in valid memory addresses amongst these segments. For example, try these two different instructions (assuming **edx** is a valid memory location) and see what the results are:

```
mov eax, dword [ edx ]
mov ebx, dword [ cs:edx ]
```

CS - Code Segment register is used as the segment for any accesses relating to execution. Thus, when a **call** is executed, the segment that is used is **CS**.

DS - Data Segment register is used as the segment for any accesses relating to data accesses. For example, when accessing registers **eax**, **ecx**, **edx**, **ebx**, **esi**, and **edi** with a **mov** instruction, the **DS** register is used. When accessing **esp** or **ebp**, the **SS** register is used.

ES - Used in instructions (by default) such as the **movsb** instruction. The **edi** register is determined from the **ES** segment. In this particular instruction, the **ES** segment cannot be overridden.

SS - Stack Segment register is used when the **esp** register is being accessed.

FS - Notice the **FS** register is different than the other segment registers. It has a base address of **0x7FFDF000** and a size of **0xFFF**. This indicates that if we were to access **[fs:0]**, it would grab the first byte at the virtual memory address **0x7FFDF000**. If we accessed **[fs:0x30]**, we would get the byte at location **0x7FFDF030**. Microsoft Windows (and other operating systems) tend to use the **FS** and **GS** registers as "thread local storage" selectors. In the case of 32-bit Windows, the **FS** register in user mode represents the NT Thread Environment Block (TEB), while in kernel mode the **FS** register represents the Kernel Thread Block (KTHREAD).