Operating System Development Series

IA32 Machine Language

by Mike, 2011

Introduction

This chapter covers IA32 machine language programming. The information provided here is for information purposes only and is not needed for the development of a basic operating system or executive software. Understanding the instruction format for the IA32 (and IA64) instructions can help debugging improperly assembled instructions, v86 monitors that are required for supporting v8086 mode, emulating instructions (which is required for emulating certain FPU instructions or when developing assemblers, emulators, virtual machines, and some other types of software), and when developing certain system software like debuggers and compilers.

This chapter is also for testing a new editor being used to write the new chapters that should help improve formatting and resolve spelling errors. If this test is successful, all of the new and earlier chapters will be updated to reflect the new format. Please send any feedback if you encounter any errors.

Machine Language Overview

Machine language, also known as **machine code, native code,** and **byte code,** is the set of raw instructions and data that can be executed by a **central processing unit (CPU)**. It allows a CPU to interpret a certain set of byte sequences as an "instruction" to perform a task. These tasks are very small, such as copying small amounts of data or arithmetic. The act of building a byte sequence that represents a CPU instruction is known as **coding**. The definition of **coding** has evolved as programming languages evolved. Originally the term referred to the actual coding of the byte sequence for an instruction; today it applies to many forms of programming in second, third, and fourth generation programming languages. Computer **programs**, also known as **software,** is the collection of machine code and data that performs a complicated task, such as word processing or playing Halo®. Machine language is often interpreted by popular media as a “series of 1's and 0's”. This is an accurate description—to an extent.

Digital Logic

Digital logic is a field of electronics that utilizes **logic gates** that allows the electronics to make decisions. Some examples of logic gates include **AND gates, OR gates, NOR gates, NAND gates, NOT gates** and **XOR gates.** These gates reflect their binary operations: AND gates perform a binary AND, XOR performs a binary XOR, and so on. In order for these gates to be meaningful, a standard needed to be adopted in order to make sense of what is **true** and **false**. For example, AND gates only make sense if there are two inputs and one output. The two inputs are the two items to test for equality: if either one is false, the output is false else the output is true. The standard is to define a line with low electrical current as **false** and a line with higher electrical current as **true**. This is what connects the binary number system to digital logic electronics. In the binary number system, **0** is often denoted as **false** and **1** is often **true**. Machine language is often represented in binary because of its tight connection with the CPU instruction decoding mechanism and how its stored in RAM for instruction fetching.

Program loading

Programs are loaded into memory by the operating system, executive, or firmware. The IA32 and IA64 family of CPU's can execute programs from **Read Only Memory (ROM)** and **Random Access Memory (RAM)**. This is made possible through a common **system bus** and **physical address space (PAS)** shared by firmware and program images. Because firmware and program images can both be executed directly by the CPU cores, they utilize the same machine language byte code. Machine language is different then **microcode** (see chapter 7) that firmware might use, however, the actual firmware is still machine language.

Assembly language

Assembly language is a **second generation programming language**. It allows the programmer to write a program in a well defined language that uses **mnemonics** to aid the programmers in developing the software. For example, **MOV** is a common mnemonic common in a lot of assembly languages for different architectures. **MOV** represents an instruction that copies data from a source to a destination. It is also an example of a **data movement instruction**.

The mnemonics gave a symbolic name to the instructions and instruction forms, allowing each assembly language instruction to be translated to a single (in some cases, several possible) machine language byte sequences. The program that translates assembly language instructions into machine language is known as an **assembler**. Assemblers are sometimes incorrectly called **compilers**.

Machine Instruction overview

A **machine instruction** is a single byte sequence that performs a specific task for the CPU. A set of machine instructions has been previously defined as a **machine language**. Machine instructions are defined by a CPU manufacturer in an **instruction set** that documents all of the CPU instructions the manufacturer implemented. Instruction sets also typically include suggestive assembly language mnemonics for assembler developers to use. Instruction sets are documented in CPU specifications.

The CPU manufacture implements the machine instructions supported by a particular CPU and how the CPU interprets each instruction. This allows the CPU to “execute” machine instructions that the manufacture intended. Bugs in the CPU hardware or firmware, however, can cause the CPU to “execute” instructions that are not valid instructions. These are **undocumented instructions**. Some assemblers might define

mnemonics to undocumented instructions that are well known. Some undocumented instructions that have had benefits have later become documented as real instructions. Some instructions might be left undocumented for manufacturer testing use only, such as the IA32 **LOADALL** instruction (this bug has since been fixed.) Some instructions might also have bad effects, such as halting the system or damaging the CPU (these are known as **Halt and Catch Fire (HCF)** instructions.)

There is an instruction set for every CPU architecture. Due to the evolving nature of the software industry, certain trends in instruction sets have become common. Understanding these trends can help with understanding IA32 machine language.

CISC and RISC

Instruction sets typically fall in two categories: **Complex Instruction Set Computing (CISC)** and **Reduced Instruction Set Computing (RISC)**. Examples of RISC include PPC and ARM architectures. An example of CISC is the IA32 architecture. RISC architectures utilize a simplistic instruction set format over CISC. RISC architectures typically uses a standard encoding format for each instruction that allows each instruction to be of the same number of bytes. CISC architectures also follow a standard encoding format, but allows variable length instructions.

Operation code

An **Operating Code (OPCode)** is a single byte identifier that the CPU utilizes to determine the instruction type. For example, a MOV instruction has an opcode identifier that lets the CPU know information about the instruction, such as type (MOV). Many instruction sets use opcode to distinguish one instruction from another. Some instructions may have **multi-byte opcodes** and **extended opcodes**. This gives the instruction set more flexibility.

Addressing mode

An **Addressing Mode** defines a method for the CPU to be able to reference **addresses.** The addresses may be virtual or physical depending on the architecture. Instructions, such as **data movement instructions**, need to be able to tell the CPU how to reference addresses to obtain data. For example, many CPU's support a **direct addressing mode** that allows an instruction to tell the CPU to reference (read or write) data at a specific address. For example, in IA32 assembly language:

```
mov eax, dword [0xa0000]
```

This instruction tells the CPU to use the **direct addressing mode** to read from address 0xa0000 in the current address space. Another common addressing mode is **indirect addressing**, which allows an instruction to tell the CPU to reference data using a pointer. For example, in IA32 assembly language:

```
mov eax, [ebp]
```

This tells the CPU to read a dword from the address stored in the EBP register into the EAX register. There are many more addressing modes that

architectures may utilize.

IA32 and IA64 Instruction encoding

We are now ready to begin looking at IA32 and IA64 machine instruction encoding. In order to save space, we will use IA64 to mean IA32 and IA64 instruction sets. IA32 is a subset of IA64 and thus shares a large part of the IA32 instruction set. The IA64 instruction set implements a CISC encoding. This means that each instruction follows a specific encoding structure and is variable in length. IA32 and IA64 instructions can range from 1 byte to 12 bytes in size.

Register codes

The CPU identifies internal registers by a numerical value. Many registers share the same code; the CPU decides what register to use based on the instruction being used and the current operating mode (real, protected, or long modes). The **operand size override** prefix is also used when determining what register to use. We will cover this prefix later on.

Register codes are used in the instruction encoding to let the CPU know what registers the instruction operates on. The registers use the following codes:

REX.r = 0

Code	0	1	2	3	4	5	6	7
No REX	AL	CL	DL	BL	AH	CH	DH	BH
REX	AL	CL	DL	BL	SPL	BPL	SIL	DIL
REG16	AX	CX	DX	BX	SP	BP	SI	DI
REG32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
REG64	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
MM	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
YMM	YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
SSEG	ES	CS	SS	DS	ES	GS		
	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
	DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7

For example, in the instruction **mov bx, 0x5** we would store 3 as the register code for BX. The instruction **mov ss, ax** would require storing 2 as the register code for SS and 0 for the register code of AX. Different instructions utilize different types of registers so there will never be a conflict between needing to choose between multiple registers of the same code. For example, the instruction **mov REG16, IMM16** will always use a 16 bit general purpose register as an operand. For another example, the instruction **movups xmm, xmm/m128** will always take an XMM register only.

Long mode adds additional registers to this list. In order to support the above registers, long mode has a special flag set that allows instructions to select other registers using the same register codes. This is the **REX.r** field in the **REX prefix byte** that will be explained later on. When this bit is set, the register table will look like this:

REX.r=1

Code	0	1	2	3	4	5	6	7
No REX	R8B	R9B	R10B	R11B	R12B	R13B	R14B	R15B
REX	R8W	R9W	R10W	R11W	R12W	R13W	R14W	R15W
REG16	R8D	R9D	R10D	R11D	R12D	R13D	R14D	R15D
REG32	R8	R9	R10	R11	R12	R13	R14	R15
MM	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM8	XMM9	XMM10	XMM11	XMM12	XMM13	XMM14	XMM15
YMM	YMM8	YMM9	YMM10	YMM11	YMM12	YMM13	YMM14	YMM15
SSEG	ES	CS	SS	DS	FS	GS		
	CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
	DR8	DR9	DR10	DR11	DR12	DR13	DR14	DR15

Instruction Encoding

An IA64 instruction follows a well defined structure that originated from the 8085 CPU. Each instruction follows the following format:

For compactness, the number in parentheses is the number of bytes of the component. A number of 0 indicates that the byte is optional. For example, the **prefix bytes** can be from 0 to 4 bytes in an instruction. This means that an instruction can have 0, 1, 2, 3, or 4 prefix bytes. **The REX prefix is only valid in IA64 and long modes.** The only required field is an **operation code**. All other fields are optional and depend on if the instruction requires them. For example, the **INT (interrupt)** instruction requires an operation code and immediate byte while a **MOV** instruction might utilize all of the above fields.

To provide another example, lets take a look at the INT instruction in more detail. The INT instruction has a form:

```
INT imm8
```

where imm8 is an 8 bit immediate value and INT is the mnemonic for the operation code 0xCD. Knowing the format of the instruction encoding, we can encode a INT 5 instruction like this:

```
0xCD 0x05
```

The first byte, 0xCD, is the operation code and is in **brown**. Because prefix bytes are optional, and INT 5 does not require them, we do not need it. Mod R/M and SIB bytes are not needed either. Displacements are only used with memory a**ddressing modes** so the only other field that we need is the immediate field. The immediate field is a 0-4 byte field. We know to use it as a 1 byte field because of the instruction form INT imm8. The purpose of this example is to demonstrate that **certain fields are optional and not needed**; the fields that are needed by an instruction depends on the instruction. **The order of these fields never changes**. For example, note above how we chose to omit the fields that are not needed but kept the order of the fields: the operation code field comes before the immediate value field. In the next few sections, we will cover each of these fields in more detail.

Prefix fields

The **prefix bytes** allow an instruction to give more information to the CPU. For example, they allow the instruction to have the CPU lock the bus or to utilize a different segment register in a data movement instruction. Many of these prefixes have assembly language mnemonics. The prefix bytes are identified in 4 classes. **An instruction can use at most 1 prefix byte from each of the 4 classes**.

Class	1 prefix
0xF0	LOCK prefix
0xF2	REPNE, REPNZ prefix
0xF3	REP, REPZ, REPE prefix

Class	2 prefix
0x2E	CS Segment override
0x36	SS Segment override
0x3E	DS Segment override
0x26	ES Segment override
0x64	FS Segment override
0x65	GS Segment override

Class	3 prefix
0x66	Operand size override

Class	4 prefix
0x67	Address size override

We assume the reader knows IA32 assembly language so will omit describing these prefixes in detail. A machine instruction can only have 1 prefix byte from any of the 4 classes. Due to their being 4 classes, an instruction can have 0 to 4 prefix bytes. If an instruction attempts to use 2 or more prefix bytes from a single class, the CPU will throw an invalid instruction exception.

LOCK prefix

If the **LOCK** prefix is used on an instruction that does not support LOCK the CPU will trigger an invalid instruction exception. Some assemblers allow using LOCK on invalid instructions without giving a warning to the programmer. Due to this, we will present the list of valid instructions here.

The **LOCK** prefix can only be used on the following instructions: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR.

Operand size override

The **operand size override** allows the CPU to select between 16 bit and 32 bit operands. Assemblers typically allow the programmer to select a specific operand size indirectly using directives like **bits16** or **use32.** The IA32 and IA64 instruction sets provide two operand sizes: legacy 16 bit and a native size that is 32 bit. The native size depends on the processors current operation mode.

Operation mode	CS.d	REX.w	Native	Operand override
Real mode			16 bit	16 bit
V8086 mode			16 bit	16 bit
Protected mode	0		16 bit	32 bit
Protected mode	1		32 bit	16 bit
Long mode		0	32 bit	16 bit

For an example, lets look at the ADD AX/EAX, IMM16/IMM32 instruction. This instruction has operation code 0x05. In protected mode code, the CPU will interpret this as an ADD EAX, IMM32 instruction by default. However we can override the default behavior and copy a 16 bit immediate value by using an operand override prefix. We do this in assembly language like this:

```
add eax, 5 ; MOV EAX, IMM32
add ax, 5  ; MOV AX, IMM16
```

The first instruction will assemble to:

```
0x05 0x05 0x00 0x00 0x00
```

The second instruction will assemble to:

```
0x66 0x05 0x05 0x00
```

Notice the only differences between these two instructions are the following: (1) The first instruction uses a 32 bit immediate value and the second instruction uses a 16 bit immediate value (these are in **red**), and (2) the second instruction uses the operand override prefix (this is in **black**). This tells the CPU to use the 16 bit operand form. For completeness, the values in **brown** are the operation codes.

Address size override

The address size override prefix byte is very similar to the operand override prefix byte. Assemblers allow programmers to be able to select between address sizes by using keywords such as **byte ptr** and **dword ptr.** Due to the function being very similar to the operand override prefix, we will omit describing its purpose because it is the same but applies to address modes.

Operation mode	CS.d	REX.w	Native	Address override
Real mode			16 bit	16 bit
V8086 mode			16 bit	16 bit
Protected mode	0		16 bit	32 bit
Protected mode	1		32 bit	16 bit
Long mode		0	64 bit	32 bit
Long mode		1	64 bit	32 bit

For example, the instruction **mov eax, [0xa000]** when assembled in protected mode would not need an address size override. The assembler would treat 0xa000 as a 32 bit displacement. However, if we used **mov ax, word [0xa000]** the assembler would add an address size override prefix to the instruction to select the 16 bit address form.

REX prefix

The REX prefix enables certain 64 bit specific features. It has the following format:

```
| 0 | 1 | 0 | 0 | w | r | x | b |
```

```
+---+---+---+---+---+---+---+---+
7                               0
```

REX.w	Operand size. 0: Default, 1: 64 bit
REX.r	ModRM.reg extension
REX.x	SIB.index extension
REX.b	ModRM.rm extension

Prefix Order

The order of the prefix bytes when used in conjunction with other prefix bytes does not matter. For example, you can use **0xF3 0x2E** in the machine code to select REP and CS override. You can also use **0x2E 0xF3** to do the same thing.

Operation code field

The operation code field can be 1-3 bytes in length. All operation codes are unique; they identify the instruction to use and its operands. For example, the operation code 0 identifies the ADD REG/MEM8, REG8 instruction. The operation code 1 identifies an ADD REG/MEM16/MEM32, REG16/REG32 instruction. The IA32 and IA64 CPU manuals outline each instruction and their respective operation code.

Primary Opcode

The **primary opcode** is a single byte that is required in all instructions. It is the base of the operation code fields used to identify the instruction. The primary opcode can also take on one of the following formats depending on instruction.

```
|   |   |   |   |   |  d | s | w |
+---+---+---+---+---+----+---+---+
7                               0

|   |   |   |   |   |     tttn      |
+---+---+---+---+---+----+---+---+
7                               0

|   |   |   |   |   | register ID|
+---+---+---+---+---+----+---+---+
7                               0

|   |   |   |   |   |   |    |   mf  |
+---+---+---+---+---+----+---+---+
```

```
7                                   0
```

PO.w	Operand size
PO.s	Sign extend
PO.d	Direction
PO.tttn	Used on some FPU instructions
PO.mf	Memory format

Secondary OPCode

When the primary opcode byte is 0xf0, another byte follows called the secondary opcode byte. The secondary opcode then identifies different instructions and has the same functionality as above. These should be treated as two byte opcodes.

OPCode extension

Certain families of instructions has the same opcode but differ only by a special field called an **opcode extension**. This is a 3 bit extension that is stored in the Mod R/M.reg field. The Mod R/M byte will be explained in more detail in the next section.

Multi-byte OPCodes

The primary opcode field can be 1-3 bytes in length. Although most instructions only use 1 byte of the primary opcode field, there are some that can utilize all 3 bytes. All of these instructions also uses a secondary opcode byte (0xf0).

Mod R/M field

The Mod R/M (Register/Memory) field is used by instructions that require memory or register operands. The Mod R/M field has the following format.

```
| mod   |    reg    |    rm     |
+---+---+---+---+---+---+---+---+
7                               0
```

The Mod R/M field is slightly different depending on if the CPU is running in real, protected or long modes.

```
Real mode                                       Protected and Long modes
```

ModRM.mod	00: [Memory] 01: [Memory+DISP8] 10: [Memory+DISP16] 11: Register
ModRM.reg	Register code
ModRM.rm	If ModRM.mod = 11: register code 000: [BX+SI] 001: [BX+DI] 010: [BP+SI] 011: [BP+DI] 100: [SI] 101: [DI] 110: [BP] or [DISP16] when ModRM.mod=0 111: [BX]

ModRM.mod	00: [Memory] 01: [Memory+DISP8] 10: [Memory+DISP32] 11: Register
ModRM.reg	Register code
ModRM.rm	If ModRM.mod = 11: register code REX.b=0 REX.b=1 000: [RAX] 000: [R8] 001: [RCX] 001: [R9] 010: [RDX] 010: [R10] 011: [RBX] 011: [R11] 100: [SIB] 100: [SIB] 101: [RBP][DISP32] 101: [DISP32] 110: [RSI] 110: [R14] 111: [RDI] 111: [R15]

The ModRM.mod field is combined with Mod.rm field to determine the addressing mode. For example, the instruction **mov ax, [0xa000]** would use (in real mode) ModRM.mod = 0 (Memory) and ModRM.rm = 6 (DISP16). ModRM.reg would contain the register code for AX. If we are to use **mov ax, [bx+0xa000]** instead, ModRM would be 2 (Memory+DISP16) and ModRM.rm would be 7. Assemblers would treat 0xa000 here as a DISP16 rather then a DISP8 due to 0xa000 being a word size displacement. We can select the DISP8 form by using an **address size override** prefix.

Looking at the above tables we can deduce that there is not many registers can be used for indirect addressing. For example, **[BP]** is not a valid addressing mode, but assemblers can assemble this fine in instructions like **mov ax, [bp]**. A common trick used by these assemblers is to set ModRM.mod = 1 (Memory+DISP8) and ModRM.rm = 6 (BP). In other words, the assemblers translate this into a **[BP+DISP8]** addressing mode, setting the displacement to 0. So **mov ax, [bp]** is assembled into **mov ax, [bp+0]**.

Protected and long modes introduce another addressing mode, **[SIB]** which gives more capabilities. SIB addressing modes can be combined with ModRM.rm modes. For example, in protected mode, **mov eax, [ebx+edi*2+0xa000]** would be translated to ModRM.mod = 2 (Memory+DISP32) and ModRM.rm = 4 (SIB byte). The SIB byte tells us how to extract the **edx+edi*2** in this instruction so will be explained in the next section.

Certain instructions utilize an **extended opcode** field. Extended opcodes are identifiers that are used with the **primary opcode** when identifying instructions. This is a 3 bit field and is stored in Mod R/M.reg field for these instructions. Instructions that use an extended opcode field might still use ModRM.rm and ModRM.mod to store a register operand or memory addressing mode.

SIB field

The **Scale Index Base (SIB)** byte **follows a Mod R/M byte only if Mod R/M.rm = 4 and the CPU is in protected or long modes**. The byte

provides additional addressing modes to the IA32 and IA64 architectures. SIB addressing is combined with Mod R/M addressing in order to create a wide array of additional addressing modes.

```
| Scale |   Index   |   Base    |
+---+---+---+---+---+---+---+---+
7                               0
```

SIB.Scale	00: Factor 1 01: Factor 2 10: Factor 4 11: Factor 8
SIB.Index	Uses standard register codes If VSIB, uses VR register codes If REX.x = 1, uses 64 bit register codes If REX.x = 1 and VSIB, uses VR register codes
SIB.Base	Uses standard register codes If REX.b=1, uses 64 bit register codes

Despite the names of the register fields in the SIB byte, you can technically use any register code. For example, you can put an index register in SIB.Base.

Lets combine the SIB byte with Mod R/M again to demonstrate how they work together. Using our previous example, **mov eax, [ebx+edi*2+0xa000]**. We assume the CPU is running in protected mode for this example. EAX is the non-memory register, so that will be in Mod R/M.reg. We also have to set Mod R/M.mod = 2 to enable [Memory+DISP32] and Mod R/M.rm = 4 [SIB byte]. **ebx** is our base register, **edi** is our index register. The **register code** for EBX is 3 and the **register code** for EDI is 7. Using this, we can set SIB.index = 7 and SIB.base = 3. The scale factor, **2** goes into SIB.scale.

Putting this together, we have a Mod R/M byte of **10 000 100 binary** and an SIB byte of **10 111 011 binary**. Knowing we have a 32 bit displacement of **0xa000** and the operation code being **0x89**, we can translate our example instruction into:

```
0x89 0x84 0xbb 0x00 0xa0 0x00 0x00
```

This would be the correct translation of **mov eax, [ebx+edi*2+0xa000]**. To ease readability, the operation code is in **brown**, Mod R/M and SIB bytes are in **red**, and the displacement field is in **black**. Note that this follows the exact format of an instruction: first is the primary opcode, next is the Mod R/M byte, next is the SIB byte, and the displacement byte follows.

If the displacement byte in the above instruction looks odd, please consider that the IA32 and IA64 architectures are **little endian**.

Displacement field

The displacement field is only valid if Mod R/M.mod is mode 1 (Memory+DISP8) or mode 2 (Memory+DISP16 or Memory+DISP32). The displacement can be a byte, word or dword value and is used in conjunction with the Mod R/M and SIB bytes to add displacements to addressing modes. The displacement field always follows the Mod R/M or SIB byte.

Immediate field

The immediate field is only valid if the instruction requires it as an operand. Instructions might require an 8, 16, or 32 bit immediate value. This field must then be present as the last field in the instruction. Instructions that allow both 16 and 32 bit values depend on if an **operand override size** prefix is present to determine the size of this field.

Instruction tables

An **instruction look-up table** is utilized by certain software to help facilitate the machine language translation of instructions. The tables reflect that of a generic instruction table that provides all of the operational codes and operand types for all of the instructions. We can use a resource, such as the IA32 manuals or an online reference, to construct the tables or to help with building machine instructions. The design of these tables varies considerably; it is important to read the documentation on how to read these look-up tables.

The tables would present an instruction in a form similar to the following.

```
| 0x10 | ADC | R/MEM8 | R8 |
+------+-----+--------+----+
```

This represents an ADC instruction whose operation code is 0x10. The R/MEM8 is the first operand, and R8 is the second operand. Operands can be represented in different ways depending on the tables' design. The table may also present additional information such as effected flags the instruction sets, what form of the opcode byte the instruction might use (such as if it stores a register ID in the opcode field), supported processors, and so on. These tables can get really large in size but they all provide the same basic information typically presented in the above form.

R/MEM8 in the above means that the first operand is a “register” or “8 bit memory location”. The “R8” means the second operand is an 8 bit register. **If an instruction has a memory operand, then a Mod R/M (and possibly an SIB byte) must follow**. Also, **if an instruction takes two register operands, a Mod R/M byte must follow**. The Mod R/M byte will store the memory addressing mode information or both register codes in Mod R/M.rm and Mod R/M.reg. The CPU will know the register code is for an 8 bit register because of the opcode. **The opcode tells the CPU not only what instruction to execute, but also what operands the instruction requires**. An instruction may utilize different types of operands, because of this the same instruction can occupy multiple opcodes. For example, the above instruction form uses opcode 0x10. Other forms include but are not limited to the following.

```
| 0x11 | ADC | R/MEM16/MEM32 | R16/REG32     |
```

```
+------+-----+---------------+---------------+
| 0x12 | ADC | R8            | R/REG16/REG32 |
+------+-----+---------------+---------------+
| 0x13 | ADC | R16/REG32     | R/MEM16/MEM32 |
+------+-----+---------------+---------------+
```

If an instruction uses an operand like **REG16/REG32**, you need to deduce the operand to use based on if an **operand size override** prefix is present and the current CPU operation mode (that is, if it is running in protected mode, real mode, long mode, and so on). For example, if the instruction is an **ADC ax, word ptr [0]** and we are running this in protected mode (or, in assembly language terminology, we used **bits32** or **use32** directives), we can use opcode 0x13 for the instruction. We know this instruction takes the form **ADC REG16, MEM16/MEM32**. AX is the first operand, which is a 16 bit register (REG16). But what is [0]? In order to find out, we take into consideration that there is no address size override and that we are in protected mode. Due to there being no address size override we are to use the native size, which is 32 bit memory addressing. (Please see the section on the address size override prefix for more information.) Using this, we conclude that we select the **ADC REG16, MEM32** form. (If an address size override did exist, we would select the ADC REG16, MEM16 form).

If an instruction only has a single register operand, verify if the operand is stored in the OPCode.reg field. (Please see the Operational code section for more information.) Some instructions do this to save space, **this is what allows single byte instructions**. Some instructions might utilize two registers for operands storing them in OPCode.reg and Mod R/M.reg or Mod R/M.rm.

To complete this example, we note the following: OPCode 0x13, AX register code (0), Addressing mode is [Memory] with a displacement of 0. Due to is being in protected mode, we use the 32 bit Mod R/M form. Mod R/M.reg = 0 (selecting AX), Mod R/M.rm = 5 (DISP32) and Mod R/M.mod = 0 ([MEMORY]). This creates a Mod R/M value of **00 000 101 binary**. Due to us not using a [SIB] mode, we do not need to use an SIB byte. (For an example that does use the SIB, please see our previous example for disassembling **mov eax, [ebx+edi*2+0xa000]**). Using this information, we can build the machine code.

```
0x66 0x13 0x05 0x00 0x00 0x00 0x00
```

The OPCode is in **brown** and Mod R/M byte is in **red**. The displacement byte is a DISP32 (due to Mod R/M.rm) so must be a dword. This is identified in **black**. The 0x66 is an **operand size override prefix** which is in **blue**. We use an operand override size prefix in order to select the REG16 and MEM16 form rather then the REG32 and MEM32 form. If we omit the prefix, we will get the following instead.

```
0x13 0x05 0x00 0x00 0x00 0x00
```

In protected mode, this is an **adc eax, dword ptr [0]** instruction which was not what we wanted. Please see the operand size override prefix section for more information.

If we wanted to turn **ADC ax, word ptr [0]** into **REP ADC ax, word ptr ES:[0]** we can use an ES override prefix and REP prefix:

```
0xf3 0x26 0x66 0x13 0x05 0x00 0x00 0x00 0x00
```

The order of the prefix bytes do not matter.

Resources

The following resources are presented for supplemental reading. Please note that we do not provide support for these resources.

http://ref.x86asm.net/	IA32 and IA64 instruction table
http://www.sandpile.org/	Instruction format tables
http://wiki.osdev.org/X86-64_Instruction_Encoding	IA32 and IA64 Instruction encoding

Conclusion

This chapter provided an overview of machine language programming and the instruction encoding on IA32 and IA64 architectures. A goal of this chapter is to present the material in a new way to encourage the development of debuggers and tool-chains. This chapter can also be used as a reference with an instruction table when emulating certain instructions.

Please let me know if there are any questions or comments,

~Mike ();

OS Development Series Editor