5
x86 Assembly Language

In this chapter, we shall study the basics of the x86 family of assembly languages. They are primarily used in Intel and AMD processors, which have an overwhelmingly large market share in the desktop, laptop, and low end server markets. They are steadily making deep inroads into the middle and high end server markets as well as the smart phone market. Hence, it is essential for the reader to have a good understanding of this important class of assembly languages. At this stage we expect the reader to have a basic understanding of assembly language from Chapter 3.

5.1 Overview of the x86 Family of Assembly Languages

5.1.1 Brief History

Let us start out by noting that x86 is not one language; it is actually a family of assembly languages with a very interesting history. Intel released the 8086 microprocessor in 1978, and called it 8086. It was Intel's first 16-bit microprocessor. This microprocessor proved to be very successful in the market, and succeeded in displacing other 8-bit competitors at that time. This motivated Intel to continue this line of processors. Intel then designed the 80186 and 80286 processors in 1982. 80186 was aimed at the embedded processor market, and 80286 was aimed at desktops. Both of them were fairly successful and helped establish Intel processors firmly in the desktop market. Those days IBM was the biggest vendor of PCs (personal computers), and most IBM PCs used Intel processors. The rapid proliferation of PCs led Intel to release two more processors, 80386 and 80486, in 1985 and 1989 respectively. These were 32-bit processors. Note that as Intel moved from 8086 to 80486, it continuously added more and more instructions to the instruction set. However, it also maintained backward compatibility. This means that any program meant to run on a 8086 machine, could also run on a 80486 machine. Secondly, it also maintained a consistent assembly language format for this family of processors whose name ended with "86". Over time this family of processors came to be known as "x86".

Gradually, other companies started using the x86 instruction set. Most notably, AMD (Advanced Micro Devices) started designing and selling x86 based processors. AMD released the K5, K6, and K7 processors in the mid nineties based on the 32-bit x86 instruction set. It also introduced the x86_64 instruction set in 2003, which was a 64-bit extension to the standard 32-bit x86 Intel ISA. Many other vendors such as VIA, and Transmeta also started manufacturing x86 based processors starting from 2000.

Each vendor has historically taken the liberty to add new instructions to the base x86 instruction set. For example, Intel has proposed many extensions over the years such as Intel® MMX™, SSE1, SSE2, SSE3, and SSE4. The number of x86 instructions are more than 900 as of 2012. Similarly, AMD introduced the 3D Now!™instruction set, and VIA introduced its custom extensions. The rich history of x86 processors has led to many different extensions of the basic instruction set, and there are numerous assemblers that have their unique syntax. Almost all x86 vendors today support hundreds of instructions. Current 64-bit Intel processors support 16-bit, and 32-bit code that dates way back to the original 8086.

If we try to classify the entire family tree of x86 ISAs, we can broadly divide them as 16-bit, 32-bit, and 64-bit instruction sets. 16-bit instruction sets are rarely used nowadays. 32-bit instruction sets are extremely popular in the smart phone, embedded, and laptop/netbook markets. The 64-bit ISAs (also known as the x86-64 ISA) are mainly meant for workstation class desktop/laptops and servers. Other than minor syntactic differences the assembly languages for these instruction sets are mostly the same. Hence, learning one ISA is sufficient. In this book, we try to strike a compromise between embedded processors, laptops, desktops, smart phones, and high end servers. We thus focus on the 32-bit x86 ISA because in our opinion it falls in the middle of the usage spectrum of the x86 ISA. We shall mention the minor syntactic differences with other flavours of x86 whenever the need arises.

5.1.2 Main Features of the x86 ISA

Before delving into the details of the 32-bit x86 ISA, let us list some of its main features.

1. It is a CISC ISA. Instructions have varying lengths, and operands also do not have a fixed length.

2. There are at least 300 scalar instructions, and this number is increasing every year.

3. Almost all the instructions can have a memory operand. In fact, most instructions allow a source, and a destination memory operand.

4. Most of the x86 instructions are in the 2-address format. For example, the assembly instruction to add two registers *eax*, and *ebx*, is *add eax, ebx*. Here, we add the contents of the *eax*, and *ebx* registers, and save the results in the *eax* register.

5. x86 has many complicated addressing modes for memory operands. Along with the traditional base-offset addressing mode, it supports base-index and base-index-offset addressing modes.

6. It does not have a return address register. Function call and return instructions, save and retrieve the return address from the stack.

7. Like ARM and *SimpleRisc* , x86 has a $flags$ register that saves the outcome of the last comparison. The $flags$ register is used by conditional branch instructions.

8. Unlike *SimpleRisc* , x86 instructions do not see an unified view of instruction and data memory. The x86 memory is *segmented*. This means that instructions and data reside in different memory regions (known as *segments*). x86 machines restrict the segments that an instruction can access.

It is true that the x86 architecture is a CISC instruction set, and it has hundreds of opcodes and many addressing modes. Nevertheless, we are sure that at the end of this chapter, the reader will concur with us that the x86 instruction set is in reality a fairly simple instruction set, is easy to understand, and is very elegant. A conventional argument supporting the case of RISC ISAs is that the hardware is simpler, and more efficient. Consequently, in modern Intel/AMD processors (Pentium® 4 onwards), the x86 instructions are internally translated into RISC instructions, and the entire processor is essentially a RISC processor. We can thus get the best of both worlds.

5.2 x86 Machine Model

5.2.1 Integer Registers

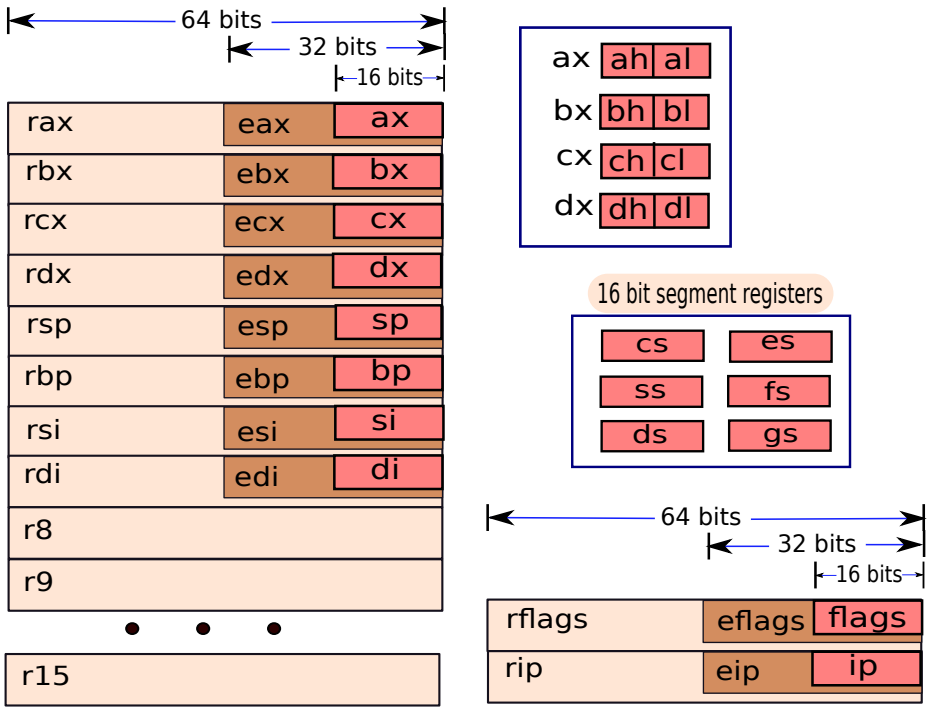

Figure 5.1: The x86 register set

Figure 5.1 shows the x86 register set. The 16 and 32-bit x86 ISAs have 8 general purpose registers. These registers have an interesting history. The original 8080 microprocessor designed forty years ago had seven 8-bit general purpose registers namely a, b, c, d, e, f and g. In the late seventies, x86 designers decided to create a 16-bit processor called 8086. They decided to keep four registers (a, b, c, and d), and suffixed them with the 'x' tag ('x' for extended). Thus, the four general purpose registers got renamed to ax, bx, cx, and dx. Additionally, the designers of the 8086 machine decided to retain some 16-bit registers namely the stack pointer (sp), and the register to save the PC (ip). The designers also introduced three extra registers in their design – bp (base pointer), si (starting index), and di (destination index). The intention of adding the bp register was to save the value of the stack pointer at the beginning of a function. Compilers are expected to set sp equal to bp at the end of the function. This operation destroys the stack frame of the callee function. The registers si, and di are used by the rep instruction that repeats a certain operation. Typically, a single rep instruction is equivalent to a simple for loop. Thus, the 8086 processor had eight 16-bit general purpose registers – ax, bx, cx, dx, sp, bp, si, and di. It was further possible to access the two bytes (lower and upper) in the registers ax – dx. For example, the lower byte in the ax register can be addressed as al, and the upper byte can be addressed as ah. 16-bit x86 instructions can use combinations of 8-bit and 16-bit operands.

The 8086 processor had two special purpose registers. The first register called ip contained the PC. The PC is typically not accessible to programmers on x86 machines (unlike the ARM ISA). The second special purpose register is the $flags$ register that saves the results of the last comparison (similar to the $flags$ register in ARM and *SimpleRisc*). The $flags$ register is used by subsequent conditional branch instructions to compute the outcome of the branch.

In the might eighties, when Intel decided to extend the 8086 design to support 32-bit registers, it decided to keep the same set of registers (8 general purpose + ip + $flags$), and similar nomenclature. However, it extended their names by adding an 'e' prefix. Thus in a 32-bit machine, register eax is the 32-bit version of ax. To maintain backward compatibility with the 8086, the lower 16 bits of eax can be addressed as ax (if we wish to use 16-bit operands). Furthermore, the two bytes in ax can be addressed as ah and al (similar to 8086). As shown in Figure 5.1, the names were changed for all the other registers also. Notably, in a 32-bit machine, the stack pointer is stored in esp, the PC is stored in eip, and the $flags$ are stored in the $eflags$ register.

There are many advantages to this strategy. The first is that 8086 code can run on a 32-bit x86 processor seamlessly. All of its registers are defined in the 32-bit ISA. This is because each 16-bit register is represented by the lower 16 bits of a 32-bit register. Hence, there are no issues with backward compatibility. Secondly, we do not need to add new registers, because we simply extend each 16-bit register with 16 additional bits. We refer to the new register with a new name (16-bit name prefixed with 'e').

Exactly the same pattern was followed while extending the x86 ISA to create the 64-bit x86-64 ISA. The first letter was replaced from 'e' to 'r' to convert a 32-bit register to a 64-bit register. For example, the register rax is the 64-bit version of eax. Its lower 32 bits can be addressed as eax. The connotation of ax, ah, and al remains the same as before. Additionally, the x86-64 ISA introduced 8 more general purpose registers namely $r8$ – $r15$. However, their subfields cannot be addressed directly. The 64-bit PC is saved in the rip register, and the flags are stored in the $rflags$ register.

The $eflags$ register

Let us now quickly discuss the structure of the $eflags$ register. Like ARM and x86, the $eflags$ register contains a set of fields, where each field or bit indicates the status of execution of the instruction that last set it. Table 5.1 lists some of the most commonly used fields in the $eflags$ register, along with their semantics.

Field	Condition	Semantics
OF	Overflow	Set on an overflow
CF	Carry flag	Set on a carry or borrow
ZF	Zero flag	Set when the result is a 0, or the comparison leads to an equality
SF	Sign flag	Sign bit of the result

Table 5.1: Fields in the $eflags$ register

5.2.2 Floating Point Registers

The floating point instructions in x86 have a dual view of the floating point register file. They can either see them as normal registers or as a set of registers organised as a stack. Let us elaborate.

To start out, x86 defines 8 floating point registers named: $st0 \ldots st7$. These are 80-bit registers, The x86 floating point format has a 64-bit mantissa, and a 15-bit exponent. It is thus more precise than double precision numbers. The registers $st0$ to $st7$ are organised as a stack. Here, $st0$ is the top of the stack, and $st7$ is the bottom of the stack as shown in Figure 5.2. Additionally, x86 has a tag register that maintains the status of each register in the stack. The tag register has 8 fields (1 field for 1 register). Each field contains 2 bits. If the value of these bits is 00, then the corresponding register contains valid data. If the value is 01, then the register contains a 0, and if it is 11, then the register is empty. 10 is reserved for special purposes. We shall refer to the stack of registers, as *the floating point stack*, or simply *the FP stack*.

The registers $st0$ to $st7$ are positions on the FP stack. $st0$ is always the top of the stack, and $st7$ is always the bottom of the stack. If we push a data item on to the FP stack, then the contents of each register get transferred to the register below it. If the stack is full (means that $st7$ contains valid data), then a stack overflow occurs. This situation needs to be avoided. Most floating point instructions operate on data values saved at the top of the stack. They pop the source operands, and push the destination operand.

5.2.3 View of Memory

Let us now describe the functionality of the segment registers (see Figure 5.1), and the view of memory. x86 instructions can have two views of memory. The first view is like ARM and *SimpleRisc* , which views memory as one large array of bytes that stores both code and data.

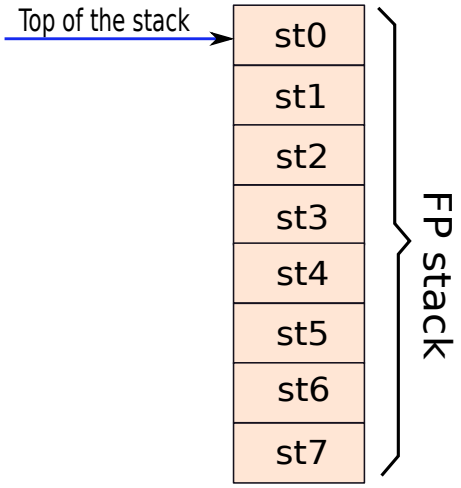

Figure 5.2: The x86 floating point register stack

This is known as the *linear memory model*. In comparison, the *segmented memory model* views memory as consisting of fixed size segments, where each segment is tailored to store one kind of data such as code, stack data, or heap data (for dynamically allocated data structures). We shall not discuss the linear model of memory because we have seen it before in Chapter 3. Let us discuss the segment registers, and the segmented memory model in this section.

Definition 40

Linear Memory Model *A* linear memory model *views the entire memory as one large array of bytes that saves both code and data.*

Segmented Memory Model *A* segmented memory model *views the memory as a sequence of multiple fixed size segments. Code, data, and the stack have their own segments.*

The Segmented Memory Model

Let us define the term *address space* as the set of all memory addresses accessible to a program. The aim of the segmented memory model is to divide the address space into separate smaller address spaces. Each address space can be specialised to store a specific type of information such as code or data.

There are two reasons for using segmentation. The first is historical. In the early days different parts of a program were physically saved at different locations. The code was saved on punch cards, and the memory data was stored in DRAM memories. Hence, it was necessary to

partition the address space among the devices that stored all the information that a program required (code, static data, dynamic data). This reason is not valid anymore. Nowadays, all the information a program requires is typically stored at the same place. However, we still need segmentation to enforce security. Hackers and viruses typically try to change the code of a program and insert their own code. Thus a normal program can exhibit malicious behaviour and can corrupt data, or transfer sensitive data to third parties. To ensure added protection, the code region is saved in a code segment. Most systems do not allow normal store instructions to modify the code segment. We can similarly partition the data segments for different classes of data. In Section 10.4.6, we will have a more thorough discussion on this topic.

Segmentation in x86

The 8086 designers had 6 segment registers that stored the most significant 16 bits of the starting location of the segment. The remaining bits were assumed to be all zeros. The *cs* register stored the upper 16 bits of the starting location of the code segment. Similarly, the *ds* register stored the upper 16 bits of the starting location for the data segment, and the *ss* register stored the corresponding set of bits for the stack segment. The *es* (extra segment), fs, and *gs* registers could be used to store information for additional user defined segments. Till date all x86 processors have preserved this model (see Figure 5.1). The contents of instructions are saved in the code segment, and the data that a program accesses is saved in the data segment. In most small programs, the stack and data segments are the same. In 8086 processors the memory address was 20 bits wide. Hence, to obtain the final address also known as the *linear address*, the 8086 processor first shifted the contents of the segment register 4 positions to the left to obtain the starting location of the segment. It then added this address with the memory address specified by the instruction. We can think of the memory address specified by an instruction as an offset in the segment, where the starting memory location of the segment is indicated by the appropriate segment register.

This strategy served the needs of the 8086 designers well. However, this strategy is not suitable for 32 and 64-bit machines. In this case, the memory addresses are 32 and 64 bits wide respectively. Thus, the segment registers need to be wider. In the interest of backward compatibility, designers did not touch the segment registers. They just changed the semantics of its contents for newer processors. Instead of saving the upper 16 bits of the starting location of a segment, the registers now contain a segment id. The *segment id* uniquely identifies a segment across all the programs running in a system. To get the starting location, 32/64-bit x86 processors, lookup a segment descriptor table with 13 bits (bits 4 to 16) of the segment id. 13 bits can specify 8192 entries, which is more than sufficient for all the programs in the system.

Modern x86 processors have two kinds of segment descriptor tables namely the local descriptor table (LDT), and the global descriptor table (GDT). The LDT is typically local to a process (running instance of a program) and contains the details of the segments for that process. The LDT is normally not used nowadays because programs do not use a lot of segments. In comparison there is only one system level GDT. The GDT can contain up to 8191 entries (the first entry is reserved). Each entry in the GDT contains the starting address of the segment, the size of the segment, and the privileges required to access the segment. Every memory access needs to go through the GDT for fetching the starting address of the segment.

This unnecessarily lengthens the critical path of a memory request, and creates contention at the GDT. To make the access to the GDT faster, modern processors have a small structure called a *segment descriptor cache* that stores a few entries of the GDT that are relevant to the currently executing process. The descriptor cache typically stores the details of all the segments that the frequently running processes use. This strategy ensures that we do not need to access the GDT on every memory access. The small and fast descriptor cache is sufficient. After accessing the descriptor cache, or the GDT, x86 processors get the starting address of the segment. They subsequently generate the memory address by adding the address specified in the instruction with the starting address of the segment. This address is then passed on to the memory system.

Definition 41

Process *It is defined as the running instance of a program. For example, if we run two copies of a program, then we create two processes.*

LDT (Local Descriptor Table) *The LDT is a per process table that saves the description of all the segments that a process uses. The LDT is indexed by a segment id, and contains the starting address of the segment, and the privileges required to access it. It is not used very frequently in modern systems.*

GDT (Global Descriptor Table) *The GDT is similar to the LDT. However, it is a system wide table that is shared by all the processes running on a machine.*

Now, that we have discussed the view of the register files, and the memory system, let us describe the addressing modes.

5.2.4 Addressing Modes

Addressing Modes for Specifying Immediates

The best thing about x86 is that there are no size restrictions on immediates. Immediates can be as large as the size of the register. For example, in a 32-bit system, the size of the immediate can be as large as 32 bits. Depending upon the assembly language, we can specify immediates in the hex format (0x...), binary format (e.g., 10101b), or in decimal. Most of the time programmers prefer the hex or decimal formats. For hexadecimal numbers most assemblers allow us to specify the number with the standard 0x prefix. Additionally, we can specify a number with the h/H suffix. For example, $21H$ is the same as $0x21$. For negative numbers, we need to simply put a '-' before the number.

Addressing Modes for Specifying Registers

All registers in x86 are addressed by their names. For example, the general purpose registers on a 32-bit machine are addressed as *eax*, *ebx* ... *edi*, according to the rules mentioned in

Section 5.2.1. We can use 16-bit register names in 32-bit mode, and we can use 16 and 32-bit register addressing in 64-bit mode. Note that we cannot do the reverse. For example, we cannot use 64-bit register names in 32-bit mode.

Addressing Modes for Memory Operands

x86 supports a variety of addressing modes for main memory. In specific, it supports the register-indirect, base-offset, base-index, and base-index-offset addressing modes as mentioned in Section 3.2.5. In addition, it also supports a new addressing mode called the base-scaled-index-offset addressing mode that scales the index by a constant factor. Let us elaborate.

$$address = \underbrace{\begin{bmatrix} cs: \\ ds: \\ ss: \\ es: \\ fs: \\ gs: \end{bmatrix} \begin{bmatrix} eax \\ ebx \\ ecx \\ edx \\ esp \\ ebp \\ esi \\ edi \end{bmatrix}}_{base} + \left[\underbrace{\begin{pmatrix} eax \\ ebx \\ ecx \\ edx \\ ebp \\ esi \\ edi \end{pmatrix}}_{index} \times \underbrace{\begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix}}_{scale} \right] + \underbrace{[displacement]}_{offset} \tag{5.1}$$

Equation 5.1 shows the generic format of a memory address in the 32-bit version of x86. The interesting aspect of x86 memory addressing is that all of these fields are optional. Hence, it is possible to have a large number of addressing modes.

Let us first consider the addressing modes that require a base register. With the base register, we can optionally specify a segment register. If we do not specify a segment register, then the hardware assumes default segments (*ds* for data, *ss* for stack, and *cs* for code). We can subsequently specify an index. The index is contained in another register (excluding *esp*). We can optionally multiply the index with a power of 2 (1, 2, 4, or 8). Lastly, we can specify a 32-bit offset known as the *displacement*. The memory address is computed using Equation 5.1.

Now, let us look at addressing modes that do not require a base register. We can just use an index register and optionally scale it by 1, 2, 4, or 8. For example, we can specify that we want to access the memory address equal to $2 \times ecx$. This approach uses the scaled-index addressing mode. We can optionally add a fixed offset (known as the displacement) to the address.

Lastly, it is possible to specify the entire 32-bit address in the displacement field, and not specify any register at all. This approach is typically used in the operating system code to directly operate on memory addresses. Regular assembly programmers need to strictly avoid such direct memory addressing because most of the time we are not aware of the exact memory addresses. For example, the starting address of the stack pointer is typically allocated at run time in modern systems, and tends to vary across runs. Secondly, this is not a portable and elegant approach. It is only meant for operating system writers.

Let us explain with examples (see Table 5.2).

Definition 42
In the x86 ISA, the fixed offset used while specifying the effective address of a memory operand, is known as the displacement.

Memory operand	Value of the address (in register transfer notation)	Addressing mode
[eax]	eax	register-indirect
[eax + ecx*2]	eax + 2 * ecx	base-scaled-index
[eax + ecx*2 - 32]	eax + 2 * ecx - 32	base-scaled-index-offset
[edx - 12]	edx - 12	base-offset
[edx*2]	edx * 2	scaled-index
[0xFFE13342]	0xFFE13342	memory-direct

Table 5.2: Example of memory operands

5.2.5 x86 Assembly Language

There are various x86 assemblers such as MASM [mas,], NASM [nas,], and the GNU assembler [gx8,]. In this book, we shall present code snippets that have been tested with the NASM assembler. The popular NASM assembler is freely available at [nas,], and is known to work on a variety of platforms including Windows® , Mac OS X, and different flavours of Linux. Note that we shall mostly avoid using NASM specific features, and we shall keep the presentation of assembly code very generic. Our assembly codes should be compatible with any assembler that supports the Intel format for x86 assembly. The only major feature of NASM that we shall use is that comments begin with a ';' character.

Let us now describe the structure of an assembly language statement in the Intel format. Its generic structure is as follows.

Structure of an Assembly Statement

```
<label>: <assembly instruction> ; <comment>
```

For an assembly instruction, the label and the comment are optional. Alternatively, we can just have a label or a comment, or a combination of both in a single line. In our code, we shall use labels starting with a '.'. However, labels can start with regular alphabets and other special characters also. For a detailed description readers can refer to the NASM documentation.

Each x86 assembly instruction has an opcode followed by a set of operands.

x86 Assembly Instruction

```
<opcode>
<opcode> <operand1>
<opcode> <operand1>, <operand2>
```

An overwhelming majority of x86 instructions are in the 0, 1 and 2-address formats. 0-address format instructions like *nop* instructions in *SimpleRisc* do not require any operands. 1-address format instructions have a single source operand. In this case the destination operand is equal to the source operand. For example, the instruction *not eax* computes the bitwise complement of *eax*, and saves the result in *eax*. In two operand instructions, the first operand is the first source operand and also the destination operand. The second operand is the second source operand. For example, *add eax, ebx*, adds the contents of *eax* and *ebx*, and subsequently saves the result in *eax*.

The source operands can be register, memory, or immediate operands. However, both the sources cannot be memory operands. Needless to say the destination operand cannot be an immediate operand. When a single operand is both the source and destination, both the rules apply.

5.3 Integer Instructions

5.3.1 Data Transfer Instructions

The *mov* Instruction

Semantics	Example	Explanation
mov (*reg*/*mem*), (*reg*/*mem*/*imm*)	mov eax, ebx	eax ← ebx

Table 5.3: Semantics of the *mov* instruction

The *mov* instruction is a very simple yet versatile instruction in the x86 ISA. It moves the contents of the second operand, into the first operand. The second operand can be a register, a memory location, or an immediate. The first operand can be a register or a memory location (Table 5.3 shows the semantics). The reader needs to note that both the operands cannot be memory locations.

We thus do not need any dedicated load/store instructions in x86. The *mov* instruction can achieve the function of loading and storing memory values because it accepts memory operands. The *mov* instruction can also transfer values between registers (similar to *SimpleRisc* and ARM). Thus, we have fused the functionality of three RISC instructions into one CISC instruction. Let us consider some examples.

Example 55
Write an x86 assembly instruction to set the value of ebx to -17.
Answer:

```
mov ebx, -17
```

Example 56
*Write an x86 assembly instruction to load ebx with the contents of (esp - eax*4 -12).*
Answer:

```
mov ebx, [esp - eax*4 -12]
```

Example 57
*Write an x86 assembly instruction to store the contents of edx in (esp - eax*4 -12).* ***Answer:***

```
mov [esp - eax*4 -12], edx
```

movsx, and *movzx* Instructions

Semantics	Example	Explanation
movsx *reg*, (*reg*/*mem*)	movsx eax, bx	eax ← sign_extend(bx), the second operand is either 8 or 16 bits
movzx *reg*, (*reg*/*mem*)	movsx eax, bx	eax ← zero_extend(bx), the second operand is either 8 or 16 bits

Table 5.4: Semantics of the *movsx*, and *movzx* instructions

The simple *mov* instruction assumes that the sizes of the operands are the same (16, or 32, or 64 bits). However, sometimes we face the need for saving a smaller register or memory operand in a larger register. For example, if we save the 16 bit register *ax* in *ebx* then we need we have two options. We can either extend the sign of the input operand, or pad it with 0s. The *movsx* instruction (see Table 5.4) copies a smaller register or memory operand to a larger register and extends its sign. For example, the following code snippet extends the sign of *bx* (from 16 to 32 bits), and saves the results in *eax*.

```
movsx eax, bx   ; eax = sign_extend(bx)
```

The *movzx* instruction is defined on the same lines. However, instead of performing a sign extension, it pads the MSB bits with 0s.

```
movzx eax, bx   ; eax = bx (unsigned)
```

Semantics	Example	Explanation
xchg (*reg*/*mem*), (*reg*/*mem*)	xchg eax, [eax + edi]	swap the contents of eax and [eax + edi] atomically

Table 5.5: Semantics of the *xchg* instruction

The Atomic Exchange (*xchg*) Instruction

The *xchg* instruction swaps the contents of the first and second operands. Here, also we cannot have two memory operands. This instruction ensures that before the operation is done, no other operation can read temporary values. For example, if we are swapping the values of *eax*, and the memory operand [*ebx*], there might be an intermediate point in the execution where the contents of *eax* are updated, but the contents of [*ebx*] are not updated. The x86 processor does not allow other threads (sub-programs that share the address space) to read the contents of [*ebx*] at this point. It makes other conflicting instructions in other execution threads wait till the *xchg* instruction completes. This property is known as *atomicity*. An instruction is *atomic* if it appears to execute instantaneously. Most of the time, atomic instructions such as *xchg* are used for implementing data structures that are shared across multiple threads. The reader should read Chapter 11 for a detailed discussion on parallel software that uses multiple threads.

Definition 43
An instruction is atomic *if it appears to execute instantaneously.*

Example 58
Write a function to swap the contents of eax, and [*esp*].

Answer:

```
xchg eax, [esp]
```

push and *pop* Instructions

The x86 architecture is explicitly aware of the stack. It has two dedicated instructions for saving and retrieving operands off the stack. The *push* instruction pushes data on the stack. In specific, the *push* instruction can push the contents of a register, memory location, or immediate on the stack. It has just one source operand. Its operation is shown in Table 5.6. Conceptually, it first saves the value of the first operand as a temporary value *temp*. Then, it decrements the

Semantics	Example	Explanation
push (*reg*/*mem*/*imm*)	push ecx	temp ← ecx; esp ← esp - 4; [esp] ← temp
pop (*reg*/*mem*)	pop ecx	temp ← [esp]; esp ← esp + 4; ecx ← temp

Table 5.6: Semantics of the *push* and *pop* instructions

stack pointer, and transfers the temporary value to the top of the stack. In a 32-bit system, we decrement the stack pointer by 4. When we are pushing a register, the processor knows its size based on the name of the register. For example, if the name of the register is ax, its size is 16 bits, and if the name of the register is eax, its size is 32 bits. However, if we are pushing a memory operand or a constant, the assembler cannot determine the size of the operand. We might be intending to push 2 bytes, 4 bytes, or 8 bytes on the stack. In this case, it is necessary to indicate the size of the operand to the assembler such that it can generate appropriate binary code. In the NASM assembler, we specify this information as follows:

```
push dword [esp]
```

The modifier *dword* (double word) represents the fact that we need to push 4 bytes on the stack. The starting address of the 4 bytes is stored in *esp*. Table 5.7 shows the list of modifiers for different sized data types.

Modifier	Size
byte	8 bits
word	16 bits
dword	32 bits
qword	64 bits

Table 5.7: Modifiers in the NASM assembler

For pushing in the value of immediate values, NASM assumes they are by default 32 bits long (if we are running NASM in 32-bit mode). We can override this setting by specifying a size modifier (word,dword,...) in the instruction.

On the same lines we can define a *pop* instruction as shown in Table 5.6. Conceptually, the pop instruction saves the top of the stack in a temporary location. It then proceeds to increment the stack pointer by 4 (in the case of 32 bits), and then it saves the temporary value in the destination. The destination can either be a register or a memory location. The *push* and *pop* instructions thus make working with the stack very easy in x86 assembly programs.

Example 59 *What is the final value of ebx?*

```
mov eax, 10
push eax
mov ebx, [esp]
```

Answer: $\boxed{10}$

Example 60
What is the final value of ebx?

```
mov ebp, esp
mov eax, 10
mov [esp], eax
push dword [esp]
mov ebx, [ebp-4]
```

Answer: *Note that ebp and esp are initially the same. After we push a value to the stack, esp gets decremented by 4. Hence, the new location of the top of the stack is equal to ebp−4. Since we push the value of eax (10) to the top of the stack using the push instruction, the value of ebx is equal to 10.*

Example 61 *What is the final value of ebx?*

```
mov eax, 17
push eax
pop dword [esp]
mov dword ebx, [esp]
```

Answer: $\boxed{17}$

5.3.2 ALU Instructions

Let us now discuss the rich set of ALU instructions that x86 processors support.

Add and Subtract Instructions

Table 5.8 shows the add and subtract operations that are typically used in x86 processors. The basic add and subtract instructions add the values of the first and second operands, and treat the first operand also as the destination operand. They set the carry and overflow fields of the $eflags$ register. The *adc* instruction adds its two source operands, and also adds the value of the carry bit. Similarly, the *sbb* instruction subtracts the second operand from the first, and then subtracts the carry bit from the result. We can use the *adc* and *sbb* instructions to add or subtract very large integers (refer to Example 62 and Example 63). In these examples,

Semantics	Example	Explanation
add (*reg*/*mem*), (*reg*/*mem*/*imm*)	add eax, ebx	eax ← eax + ebx
sub (*reg*/*mem*), (*reg*/*mem*/*imm*)	sub eax, ebx	eax ← eax - ebx
adc (*reg*/*mem*), (*reg*/*mem*/*imm*)	adc eax, ebx	eax ← eax + ebx + (carry bit)
sbb (*reg*/*mem*), (*reg*/*mem*/*imm*)	sbb eax, ebx	eax ← eax - ebx - (carry bit)

Table 5.8: Semantics of add and subtract instructions

we first operate on the lower bytes. While operating on the higher bytes we need to take the carry generated by adding or subtracting the lower bytes into account. We use the *adc* and *sbb* instructions respectively for this purpose.

Example 62
Write an x86 assembly program to add two 64-bit numbers. The first number is stored in the registers ebx, and eax, where ebx stores the higher byte, and eax stores the lower byte. The second number is stored in edx, and ecx. Save the result in ebx (higher byte), and eax(lower byte).
Answer:

```
add eax, ecx
adc ebx, edx
```

Example 63
Write an x86 assembly program to subtract two 64-bit numbers. The first number is stored in the registers ebx, and eax, where ebx stores the higher byte, and eax stores the lower byte. The second number is stored in edx, and ecx. Subtract the second number from the first number. Save the result in ebx (higher byte), and eax(lower byte).

Answer:

```
sub eax, ecx
sbb ebx, edx
```

inc, *dec*, and *neg* Instructions

Table 5.9 shows the semantics of increment (*inc*), decrement (*dec*), and negate (*neg*) instructions. The *inc* instruction, adds 1 to the source operand. In this case also the source and destination operands are the same. Similarly, the *dec* instruction subtracts 1 from the source operand, which is also the destination operand. Note that the operand can either be a register

Semantics	Example	Explanation
inc (*reg*/*mem*)	inc edx	edx ← edx + 1
dec (*reg*/*mem*)	dec edx	edx ← edx - 1
neg (*reg*/*mem*)	neg edx	edx ← -1 * edx

Table 5.9: Semantics of *inc*, *dec*, and *neg* instructions

or a memory location. The *neg* instruction computes the negative of the value stored in the first operand (register or memory). Let us consider an example (see Example 64).

Example 64
Write an x86 assembly code snippet to compute eax = -1 * (eax + 1).
Answer:

```
inc eax
neg eax
```

The Compare(*cmp*) Instruction

Semantics	Example	Explanation
cmp (*reg*/*mem*), (*reg*/*mem*/*imm*)	cmp eax, [ebx + 4]	compare the values in eax, and [ebx+4], and set the flags
cmp (*reg*/*mem*), (*reg*/*mem*/*imm*)	cmp ecx, 10	compare the contents of ecx with 10, and set the flags

Table 5.10: Semantics of the *cmp* instructions

Table 5.10 shows the *cmp* (compare) instruction. It compares two operands and sets the values of the flags. It performs the comparison by subtracting the value of the second operand from the first operand. It is conceptually a subtract instruction that does not have a destination operand.

Multiplication and Division Instructions

Table 5.11 shows the signed multiplication and division instructions in x86. They are known as *imul* and *idiv* respectively. The unsigned variants of the instructions are known as *mul* and *div*. They have exactly the same semantics as their signed counterparts. The signed instructions are more generic. Hence, we only discuss their operation in this section.

The *imul* instruction has three variants. The 1-address format variant has 1 source operand, which can either be a register or a memory address. This source operand is multiplied with the

Semantics	Example	Explanation
imul (*reg*/*mem*)	imul ecx	edx:eax ← eax * ecx
imul *reg*, (*reg*/*mem*)	imul ecx, [eax + 4]	ecx ← ecx * [eax + 4]
imul *reg*, (*reg*/*mem*), *imm*	imul ecx, [eax + 4], 5	ecx ← [eax + 4] * 5
idiv (*reg*/*mem*)	idiv ebx	Divide (edx:eax) by the contents of ebx; eax contains the quotient, and edx contains the remainder.

Table 5.11: Semantics of the *imul* and *idiv* instructions

contents of *eax*. Note that when we multiply two 32-bit numbers, we require at most 64 bits to save the result (see Section 7.2.1). Hence, to avoid overflows, the processor saves the results in the register pair (*edx*,*eax*). *edx* contains the upper 32 bits, and *eax* contains the lower 32 bits of the final product. The 2-address format version is similar to other ALU instructions that we have studied. It multiplies the first and second source operands, and saves the result in the destination register (which is the first operand). Note that in this variant of the multiply instruction, the destination is always a register, and the result is truncated to fit in the register. The *imul* instruction has another variant that requires 3 operands. Here, it multiplies the contents of the second and third operands and stores the product in the register specified by the first operand. For this variant of the *imul* instruction, the first operand needs to be a register, the second operand can be a register or memory location, and the third operand needs to be an immediate value.

The *idiv* instruction takes just 1 operand (register or memory). It divides the contents of the register pair (*edx*:*eax*) by the contents of the operand. It saves the quotient in *eax*, and the remainder in *edx*. Note that the remainder has the same sign as the dividend. A subtle point should be noted here. While using a positive dividend that fits in 32 bits, we need to explicitly set *edx* to 0, and for a negative dividend that fits in 32 bits, we need to explicitly set *edx* to -1 (for sign extension).

Let us consider a set of examples.

Example 65
Write an assembly code snippet to multiply 3 with -17, and save the result in eax.

Answer:

```
mov ebx, 3
imul eax, ebx, -17
```

Example 66
Write an assembly code snippet to compute k^3, where k is the content of ecx, and save the result in eax.

Answer:

```
mov eax, ecx
imul ecx
imul ecx
```

Example 67
Write an assembly code snippet to divide -50 by 3. Save the quotient in eax, and remainder in edx.

Answer:

```
mov edx, -1
mov eax, -50
mov ebx, 3
idiv ebx
```

At the end eax contains -16, and edx contains -2.

Logical Instructions

Semantics	Example	Explanation
and (*reg*/*mem*), (*reg*/*mem*/*imm*)	and eax, ebx	eax ← eax AND ebx
or (*reg*/*mem*), (*reg*/*mem*/*imm*)	or eax, ebx	eax ← eax OR ebx
xor (*reg*/*mem*), (*reg*/*mem*/*imm*)	xor eax, ebx	eax ← eax XOR ebx
not (*reg*/*mem*)	not eax	eax ← ∼ eax

Table 5.12: Semantics of *and*, *or*, *xor*, and *not* instructions

Table 5.12 shows the semantics of four commonly used logical operations. *and*, *or*, and *xor* instructions have exactly the same format as *add* and *sub* instructions, and most of the other 2-address format instructions. They compute the bitwise AND, OR, and exclusive OR of the first two operands respectively. The *not* instruction computes the 1's complement (flips each bit) of the source operand, which is also the destination operand (format is similar to other 1-address format instructions such as *inc*, *dec*, and *neg*).

Shift Instructions

Semantics	Example	Explanation
sar (*reg*/*mem*), *imm*	sar eax, 3	eax $\leftarrow$ eax $\ggg$ 3
shr (*reg*/*mem*), *imm*	shr eax, 3	eax $\leftarrow$ eax $\gg$ 3
sal/shl (*reg*/*mem*), *imm*	sal eax, 2	eax $\leftarrow$ eax $\ll$ 2

Table 5.13: Semantics of shift instructions

Table 5.13 shows the semantics of shift instructions. *sar* (shift arithmetic right) performs an arithmetic right shift by replicating the sign bit. *shr* (shift logical right), shifts the first operand to the right. Instead of replicating the sign bit, it fills the MSB bits with 0s. *sal* and *shl* are the same instruction. They perform a left shift. Recall that we do not have an arithmetic left shift operation. Let us consider some examples.

Example 68
What is the final value of eax?

```
mov eax, 0xdeadbeef
sal eax, 4
```

Answer: *0xeadbeef0*

Example 69 *What is the final value of eax?*

```
mov eax, 0xdeadbeef
sar eax, 4
```

Answer: *0xfdeadbee*

Example 70 *What is the final value of eax?*

```
mov eax, 0xdeadbeef
shr eax, 4
```

Answer: *0xdeadbee*

5.3.3 Branch/ Function Call Instructions

Conditional and Unconditional Branch Instructions

Semantics	Example	Explanation
jmp ⟨*label*⟩	jmp .foo	jump to .foo
j ⟨*condcode*⟩	j ⟨*condcode*⟩ .foo	jump to .foo if the ⟨*condcode*⟩ condition is satisfied

Table 5.14: Semantics of branch instructions

Condition code	Meaning
o	Overflow
no	No overflow
b	Below (unsigned less than)
nb	Not below (unsigned greater than or equal to)
e/z	Equal or zero
ne/nz	Not equal or not zero
be	Below or equal (unsigned less than or equal)
s	Sign bit is 1 (negative)
ns	Sign bit is 0 (0 or positive)
l	Less than (signed less than)
le	Less than or equal (signed)
g	Greater than (signed)
ge	Greater than or equal (signed)

Table 5.15: Condition codes in x86

Table 5.14 shows the semantics of branch instructions. *jmp* is an unconditional branch instruction that branches to a label. The assembler internally replaces the label by the PC of the label. x86 defines a series of branch instructions with the *j* prefix. These are conditional branch instructions. The *j* prefix is followed by the branch condition. The conditions are shown in Table 5.15. For example, the instruction *je* means jump if equal. If the last comparison has resulted in an equality, then the processor branches to the label; otherwise, it executes the next instruction. If the condition is not satisfied, the conditional branch is equivalent to a *nop* instruction.

Now that we have introduced branch instructions, we can implement complex algorithms using loops. Let us look at a couple of examples. We would like to advise the reader at this point that the best method to learn assembly language is by actually writing assembly programs. No amount of theoretical reading can substitute for actual practice.

Example 71

Write a program in x86 assembly to add the numbers from 1 to 10.

Answer:

x86 assembly code

```
1 mov eax, 0  ; sum = 0
2 mov ebx, 1  ; idx = 1
3 .loop:
4         add eax, ebx ; sum += idx
5         inc ebx      ; idx ++
6         cmp ebx, 10  ; compare idx and 10
7         jle .loop    ; jump if idx <= 10
```

Here, we store the running sum in eax and the index in ebx. In Line 4, we add the index to the sum. We subsequently, increment the index, and compare it with 10 in Line 6. If it is less than or equal to 10, then we continue iterating. eax contains the final value.

Example 72

Write a program in x86 assembly to test if a number stored in eax is prime. Save the result in eax. If the number is prime, set eax to 1, otherwise set it to 0. Assume that the number in eax is greater than 10.

Answer:

x86 assembly code

```
1     mov ebx, 2      ; starting index
2     mov ecx, eax    ; ecx contains the original number
3 .loop:
4     mov edx, 0      ; required for correct division
5     idiv ebx
6     cmp edx, 0      ; compare the remainder
7     je .notprime    ; number is composite
8     inc ebx
9     mov eax, ecx    ; set the value of eax again
10    cmp ebx, eax    ; compare the index and the number
11    jl .loop
12
13    ; end of the loop
14    mov eax, 1      ; number is prime
15    jmp .exit       ; exit
16
17 .notprime:
18    mov eax, 0
19 .exit:
```

In this algorithm, we keep on dividing the input (stored in eax) by a monotonically increasing index. If the remainder is equal to 0 in any iteration, then the number is composite (non prime). Otherwise, the number is prime. In specific, we perform the division in Line 5, and jump to the label .notprime *if the remainder (stored in edx) is 0. Otherwise, we increment the index in ebx, and keep iterating. Note that in each iteration, we need to set the values of eax and edx because they are overwritten by the idiv instruction.*

Example 73
Write a program in x86 assembly to find the factorial of a number stored in eax. Save your result in ecx. You can assume that the number is greater than 10.
Answer:

x86 assembly code

```
1       mov ebx, 2      ; idx = 2
2       mov ecx, 1      ; prod = 1
3
4   .loop:
5       imul ecx, ebx   ; prod *= idx
6       inc ebx         ; idx++
7       cmp ebx, eax    ; compare num (number) and idx
8       jle .loop       ; jump to .loop if idx <= num
```

In Line 2, we initialise the product to 1. Subsequently, we multiply the index with the product in Line 5. We then increment the index, and compare it with the input stored in eax. We keep on iterating till the index is less than or equal to the input.

Function Call and Return Instructions

Semantics	Example	Explanation
call ⟨*label*⟩	call .foo	Push the return address on the stack. Jump to the label .foo.
ret	ret	Return to the address saved on the top of the stack, and pop the entry

Table 5.16: Semantics of the function call and return instructions

Unlike ARM and *SimpleRisc* , x86 does not have a return address register. The *call* instruction pushes the return address on the stack, and jumps to the beginning of the function as explained in Table 5.16. Similarly, the *ret* instruction jumps to the entry at the top of the

stack. The entry at the top of the stack needs to contain the return address. The *ret* instruction subsequently pops the stack and removes the return address. Let us now consider a set of examples.

Example 74

Write a recursive function to compute the factorial of a number (≥ 1) stored in eax. Save the result in ebx.

Answer:

x86 assembly code

```
1  factorial:
2        mov ebx, 1        ; default return value
3        cmp eax, 1        ; compare num (input) with 1
4        je .return        ; return if input is equal to 1
5
6        ; recursive step
7        push eax          ; save input on the stack
8        dec eax           ; num--
9        call factorial    ; recursive call
10       pop eax           ; retrieve input
11       imul ebx, eax     ; prod = prod * num
12
13 .return:
14       ret               ; return
```

In the factorial function, we assume that the input (num) is stored in eax. We first compare the input with 1. If it is equal to 1, then we return 1 (Lines 2 to 4). However, if the input is greater than 1, then we save the input by pushing it to the stack (7). Subsequently, we decrement it and recursively call the factorial function in Line 9. The result of the recursive call is stored in ebx. To compute the result (in ebx), we multiply ebx with num (stored in eax) in Line 11.

In Example 74 we pass arguments through registers. We use the stack to only store values that are overwritten by the callee function. Let us now use the stack to pass arguments to the factorial function (see Example 75)

Example 75

Write a recursive function to compute the factorial of a number (≥ 1) stored in eax. Save the result in ebx. Use the stack to pass arguments.

Answer:

x86 assembly code

```
1
2  factorial:
```

```
3       mov eax, [esp+4] ; get the value of eax from the stack
4       mov ebx, 1       ; default return value
5       cmp eax, 1       ; compare num (input) with 1
6       je .return       ; return if input is equal to 1
7
8       ; recursive step
9       push eax       ; save eax on the stack
10      dec eax        ; num--
11      push eax       ; push the argument
12      call factorial ; recursive call
13      pop eax        ; pop the argument
14      pop eax        ; retrieve the value of eax
15      imul ebx, eax  ; prod = prod * num
16
17  .return:
18      ret            ; return
```

Here, we use the stack to pass arguments. Since the stack pointer gets automatically decremented by 4 after a function call, the argument eax is available at [esp+4] because we push it on the stack just before we call the function. To call the factorial function again, we push eax on the stack, and then pop it out after the function returns.

Let us now assume that we have a lot of arguments. In this case, we need to push and pop a lot of arguments from the stack. It is possible that we might lose track of the order of push and pop operations, and bugs might be introduced in our program. Hence, if we have a lot of arguments, it is a better idea to create space in the stack by subtracting the estimated size of the activation block from the stack pointer and moving data between the registers and stack using regular *mov* instructions. Let us now modify our factorial example to use *mov* instructions instead of push and pop instructions (see Example 76).

Example 76

Write a recursive function to compute the factorial of a number (≥ 1) stored in eax. Save the result in ebx. Use the stack to pass arguments. Avoid push and pop instructions.

Answer:

x86 assembly code

```
1   factorial:
2       mov eax, [esp+4]  ; get the value of eax from the stack
3       mov ebx, 1        ; default return value
4       cmp eax, 1        ; compare num (input) with 1
5       jz .return        ; return if input is equal to 1
6
```

```
7       ; recursive step
8       sub esp, 8          ; create space on the stack
9       mov [esp+4], eax    ; save the input eax on the stack
10      dec eax             ; num--
11      mov [esp], eax      ; push the argument
12      call factorial      ; recursive call
13      mov eax, [esp+4]    ; retrieve eax
14      imul ebx, eax       ; prod = prod * num
15      add esp, 8          ; restore the stack pointer
16
17  .return:
18      ret                 ; return
```

In this example, we have avoided push and pop instructions altogether. We instead create space on the stack by subtracting 8 bytes from esp in Line 8. We use 4 bytes to save the input (in eax) for later use. We use the rest of the 4 bytes to send the argument to the recursive function call. After the function returns, we retrieve the value of eax from the stack in Line 13. Lastly, we restore the stack pointer in Line 15.

However, this method is also not suitable for large functions in complex programming languages such as C++. In a lot of C++ functions, we dynamically allocate space on the stack. In such cases, most of the time, we do not know the size of the activation block (see Section 3.3.10) of a function in advance. Hence, deallocating an activation block becomes difficult. We need to dynamically keep track of the size of the activation block of the function. This introduces additional complexity, and additional code. It is a better idea to save the value of *esp* in a dedicated register at the beginning of a function. At the end of the function, we can transfer the saved value in the register to *esp*. This strategy effectively destroys the activation block. Most of the time, we use the *ebp* (base pointer) register to save the value of *esp* at the beginning of a function. This register is also referred to as the *frame pointer*. Now, it is possible that a called function might follow the same strategy, and overwrite the value of *ebp* set by the caller. Thus, in this case, *ebp* needs to be a callee saved register. If an invoked function overwrites the value of *ebp*, it needs to ensure that by the time it returns to the caller, the value of *ebp* is restored. By using the base pointer, we do not need to explicitly remember the size of the activation block. We dynamically allocate data structures on the stack.

Let us augment our running example with this feature (see Example 77).

Example 77
Write a recursive function to compute the factorial of a number (≥ 1) stored in eax. Save the result in ebx. Use the stack to pass arguments. Avoid push and pop instructions. Secondly, use the ebp register to store the value of the stack pointer.
Answer:

x86 assembly code

```
1  factorial:
2       mov eax, [esp+4]    ; get the value of eax from the stack
3
4       push ebp            ; save ebp
5       mov ebp, esp        ; save the stack pointer
6
7       mov ebx, 1          ; default return value
8       cmp eax, 1          ; compare num (input) with 1
9       je .return          ; return if input is equal to 1
10
11      ; recursive step
12      sub esp, 8          ; create space on the stack
13      mov [esp+4], eax    ; save input on the stack
14      dec eax             ; num--
15      mov [esp], eax      ; push the argument
16      call factorial      ; recursive call
17      mov eax, [esp+4]    ; retrieve input
18      imul ebx, eax       ; prod = prod * num
19
20 .return:
21                  mov esp, ebp    ; restore the stack pointer
22                  pop ebp         ; restore ebp
23                  ret             ; return
```

Here, we save the old value of ebp on the stack, and set its new value to the stack pointer in Lines 4 and 5, respectively. At the end of the function, we restore the values of esp and ebp in Lines 21 and 22.

Stack Management Instructions – *enter* **and** *leave*

Semantics	Example	Explanation
enter *imm*, 0	enter 32, 0	push ebp (push the value of *ebp* on the stack); mov ebp, esp (save the stack pointer in *ebp*); esp ← esp - 32
leave	leave	mov esp, ebp (restore the value of *esp*); pop ebp (restore the value of *ebp*)

Table 5.17: Semantics of the *enter* and *leave* instructions.

The four extra lines added in Example 77 are fairly generic, and are typically a part of most large functions. Programmers can add them if they are writing assembly code, or compilers can add them during automatic code generation. In either case, using the base pointer is a very convenient mechanism to manage the stack, and to destroy the activation block. Since these set of instructions are so commonly used, the designers of the x86 ISA decided to dedicate two specialised instructions for this purpose. The *enter* instruction pushes the value of *ebp* on the stack, and sets its new value to be equal to the stack pointer. Additionally, it is also possible to set the initial size of the activation block. The first argument takes the size of the activation block. If we specify 32 as the first argument, then the *enter* instruction decrements *esp* by 32. Note that during the course of execution of the function, the size of the activation block might continue to vary. The second argument for the *enter* instruction corresponds to the nesting level of the function. We shall refrain from discussing it here. Interested readers can refer to the references mentioned at the end of the chapter. We shall simply use the value of 0 for the second argument.

The *leave* instruction performs the reverse set of computations. It first restores the value of *esp*, and then the value of *ebp* (see Table 5.17). Note that the *leave* instruction is meant to be invoked just before the *ret* instruction. We can thus augment Example 77 to use the *enter* and *leave* instructions (see Example 78). Secondly, we can omit the statement that subtracted 8 from *esp* (Line 12) in Example 77 because this functionality is now built in to the *enter* instruction.

Example 78

Write a recursive function to compute the factorial of a number (≥ 1) stored in eax. Save the result in ebx. Use the stack to pass arguments. Avoid push and pop instructions. Use the enter and leave instructions to buffer the values of ebp and esp.

Answer:

x86 assembly code

```
1
2  factorial:
3       mov eax, [esp+4]  ; get the value of eax from the stack
4
5       enter 8, 0        ; save ebp and esp, decrement esp by 8
6
7       mov ebx, 1        ; default return value
8       cmp eax, 1        ; compare num (input) with 1
9       je .return        ; return if the input is equal to 1
10
11      ; recursive step
12      mov [esp+4], eax  ; save input on the stack
13      dec eax           ; num--
14      mov [esp], eax    ; push the argument
15      call factorial    ; recursive call
16      mov eax, [esp+4]  ; retrieve input
17      imul ebx, eax     ; prod = prod * num
```

```
18
19 .return:
20      leave               ; load esp and ebp
21      ret                 ; return
```

Lastly, we should mention that x86 processors have a *nop* instruction that does not do anything at all. It is mainly used for the purpose of ensuring correctness in modern processors (see Chapter 9), and for ensuring that blocks of code are aligned to 16 byte or 64 byte boundaries. We require the latter functionality for better behaviour at the level of the memory system.

5.3.4 Advanced Memory Instructions

String Instructions

Semantics	Example	Explanation
lea *reg*, *mem*	lea ebx, [esi + edi*2 + 10]	ebx ← esi + edi*2 + 10
stos(b/w/d/q)	stosd	[edi] ← eax; edi ← edi + 4 * $(-1)^{DF}$
lods(b/w/d/q)	lodsd	eax ← [esi]; esi ← esi + 4 * $(-1)^{DF}$
movs(b/w/d/q)	movsd	[edi] ← [esi] ; esi ← esi + 4 * $(-1)^{DF}$; edi ← edi + 4 * $(-1)^{DF}$
std	std	DF ← 1
cld	cld	DF ← 0
$DF \rightarrow$ Direction Flag		

Table 5.18: Semantics of advanced memory instructions

The *lea* instruction stands for *load effective address*. It has a register operand, and a memory operand. The role of the *lea* instruction is to copy the address of the memory operand (not its contents) to the register.

Let us now introduce a special set of instructions known as *string instructions*. We shall introduce the following instructions: *stos*, *lods*, and *movs*. The *stos* instruction transfers data from the *eax* register to the location specified by the *edi* register. It comes in four flavours depending upon the amount of data that we wish to transfer. It uses the 'b' suffix for 1 byte, 'w' for 2 bytes, 'd' for 4 bytes, and 'q' for 8 bytes. We show an example of the *stosd* instruction in Table 5.18. The *stosd* instruction transfers the contents of *eax* (4 bytes) to the memory address specified by *edi*. Subsequently, this instruction increments or decrements the contents of *edi* by 4 depending on the direction flag. The direction flag (DF) is a field in the *flags* register similar to zero, carry, and overflow. If the direction flag is set (equal to 1), then the *stos* instruction subtracts the size of the operand from the contents of *edi*. Conversely, if DF is equal to 0, then the *stos* instruction adds the size of the operand to *edi*.

We introduce two 0-address format instructions namely *std* and *cld* to set and reset the direction flag respectively.

The *lods* and *movs* set of instructions are defined in a similar manner. For example, the *lodsd* instruction transfers the contents of the memory location specified by *esi* to *eax*. It subsequently increments or decrements the contents of *esi* by the size of the operands based on the value of DF. The *movs* instruction combines the functionality of *lods* and *stos*. It first fetches a set of bytes from the memory address stored in *esi*. Subsequently, it writes the bytes to the memory address specified by *edi*. It increments or decrements *esi* and *edi* based on the value of the direction flag.

Trivia 2
The si register (16-bit version of esi) stands for the source index *register. Similarly, the di register stands for the* destination index *register.*

Let us now look at a set of examples.

Example 79 *What is the value of ebx?*

```
lea  edi, [esp+4]
mov  eax, 21
stosd            ; saves eax in [edi]
mov ebx, [esp+4]
```

Answer: *We save 21 (eax) in the memory address specified in edi by using the stosd instruction. This memory address is equal to (esp + 4). After executing the stosd instruction, we load the contents of this memory address into ebx. The result is equal to the value of eax seen by the stosd instruction, which is 21.*

Example 80 *What is the value of eax after executing this code snippet?*

```
lea  esi, [esp+4]
mov dword [esp+4], 19
lodsd    ; eax <-- [esi]
```

Answer: *Note the use of the modifier dword here. We need to use it because we are saving an immediate to a memory location, and we need to specify its size. The value of eax is equal to the value stored in [esp+4], which is 19.*

Example 81 *What is the value of eax after executing this code snippet?*

```
mov dword [esp+4], 192
lea  esi, [esp+4]
lea  edi, [esp+8]
movsd
mov eax, [esp+8]
```

Answer: *The movsd instruction transfer 4 bytes from the memory address specified in esi to the memory address specified in edi. Since we write 192 to the memory address specified in esi, we shall read back the same value in the last line.*

Instructions with the *rep* Prefix

The string instructions can additionally increment or decrement the values of *esi* and *edi*. We have not used this feature up till now. Let us use this feature to transfer an array of 10 integers from one location to another. This feature is very frequently used in modern processors to transfer large amounts of data between two locations.

Let us first show a conventional solution in Example 82.

Example 82 *Write a program to create a copy of a 10 element integer array. Assume that the starting address of the original array is stored in esi, and the starting address of the destination array is stored in edi.*
Answer:

```
mov ebx, 0                      ; initialise
.loop:
     mov edx, [esi+ebx*4]       ; transfer the contents
     mov [edi + ebx*4], edx
     inc ebx                    ; increment the index
     cmp ebx, 10                ; loop condition
     jne .loop
```

Example 83 *Write a program to create a copy of a 10 element integer array. Assume that the starting address of the original array is stored in esi, and the starting address of the destination array is stored in edi. Use the movsd instruction.*
Answer:

```
    cld             ; DF = 0
    mov ebx, 0      ; initialisation of the loop index
.loop:
    movsd           ; [edi] <-- [esi]
    inc ebx         ; increment the index
    cmp ebx, 10     ; loop condition
    jne .loop
```

As compared to Example 82, we reduce the number of instruction in the loop from 5 to 4.

In Example 83, we use the *movsd* instruction to replace a pair of load/store instructions with just one instruction. This reduces the number of instructions in the loop from 5 to 4. We were not able to get a bigger reduction because we still need to update the loop index, and compute the loop condition.

To make our code look even more elegant, the x86 ISA defines a *rep* prefix that can used with any string instruction. The *rep* prefix instructs the processor to execute a single string instruction n times, where n is the value stored in the *ecx* register. Every time the processor executes the string instruction, it decrements *ecx*. At the end, the value of *ecx* becomes 0. Its semantics is shown in Table 5.19.

Semantics	Example	Explanation
rep inst	rep movsd	val ← ecx; Execute the movsd instruction *val* times; ecx ← 0

Table 5.19: Semantics of *rep* instructions

Example 84 *Write a program to create a copy of a 10 element integer array. Assume that the starting address of the original array is stored in esi, and the starting address of the destination array is stored in edi. Use the rep prefix with the movsd instruction.*
Answer:

```
cld          ; DF = 0
mov ecx, 10  ; Set the count to 10
rep movsd    ; Execute movsd 10 times
```

The *rep* prefix thus allows us to fold an entire loop into just one instruction as shown in Example 84. The *rep* prefix is meant to be used with string instructions for copying large regions

of data. It makes the code for operating on strings of data very compact and elegant. The *rep* instruction has two variants namely *repe*, and *repne*. These instructions use an additional termination condition, along with the value of *ecx*. Instructions prefixed with *repe* can also terminate when the *zero* flag becomes 0, and an instruction prefixed with *repne* also terminates when the *zero* flag becomes 1.

5.4 Floating Point Instructions

x86 has a large set of floating point instructions. Let us first give a historical perspective. The early 8086 processor, and many of its successors till the Intel 486 did not have a floating point unit in the processor. They used a separate co-processor chip called the x87 that provided floating point capability. However, after the release of Intel 486, the floating point unit has been an integral part of the x86 architecture. Hence, many features of the floating point ISA are artefacts of the older era, in which a floating point instruction was essentially a message to an external processing unit.

One of the direct consequences of such a design strategy is that there are no direct communication paths between integer registers, and floating point registers. Secondly, it is not possible to load an immediate into a floating point register by specifying its value in an instruction. We can only load the value of floating point registers via memory. For example, if we wish to store a floating point constant in a floating point register, then we need to first load the constant in memory. Subsequently, we need to issue a floating point load instruction to load the constant into a floating point register. Figure 5.3 shows a conceptual organisation of the x86 ISA. The integer instructions use the integer registers, and they have their own processor state. Likewise, the floating point instructions use their set of registers, and have their own state. Both the types of instructions, however, share the memory.

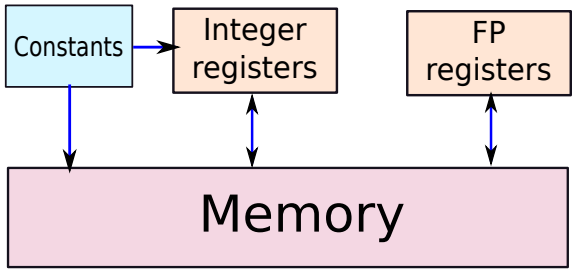

Figure 5.3: Conceptual organisation of the x86 ISA

Let us start by looking at methods to load values into the floating point registers. We shall refer to the floating point register stack as the FP stack and designate the floating point registers ($st0$ ... $st7$) as *reg* while describing the semantics of instructions. We shall also abbreviate floating point as FP for the sake of brevity.

5.4.1 Data Transfer Instructions

Load Instruction

Semantics	Example	Explanation
fld *mem*	fld dword [eax]	Pushes an FP number stored in [*eax*] to the FP stack
fld *reg*	fld st1	Pushes the contents of *st*1 to the top of the stack
fild *mem*	fild dword [eax]	Pushes an integer stored in [*eax*] to the FP stack after converting it to a 32-bit floating point number

Table 5.20: Floating point load instructions

Table 5.20 shows the semantics of the floating point load instructions. The most commonly used floating point load instruction is the fld instruction. The first variant of the fld instruction can load a 32-bit floating point value from memory, and push it to the FP stack. We can use our standard addressing modes with integer registers as described in Section 5.2.4 for specifying an address in memory. The second variant can push the contents of an existing FP register on the FP stack. We can alternatively use the $fild$ instruction that can read an integer from memory, convert it to floating point, and push it on the FP stack. Let us consider an example.

Example 85
Push the constant, 2.392, on the FP stack.
Answer: *We need to first define the constant 2.392 in the data section. In NASM, we do this as follows.*

```
section .data
        num: dd 2.392
```

We need to embed this code snippet at the beginning of the assembly file. Here, the declaration "section .data" means that we are declaring the data section. In the data section, we further declare a variable, num, that is a double word (32 bits, specified by dd), and its value is 2.392. Let us now push this value to the FP stack. We need to embed the following code snippet in the main assembly function.

```
fld dword [num]
```

The assembler treats num as a memory address. While generating code, it will replace it with its actual address. However, in an assembly program, we can seamlessly treat num as a valid memory address, and its contents can thus be represented as [num]. The fld instruction in this code snippet loads 32 bits (dword) from num to the top of the FP stack.

Exchange Instruction

Semantics	Example	Explanation
fxch *reg*	fxch st3	Exchange the contents of $st0$ and $st3$
fxch	fxch	Exchange the contents of $st0$ and $st1$

Table 5.21: Floating point exchange instructions

Table 5.21 shows the format of the floating point exchange instruction, $fxch$. It exchanges the contents of two floating point registers. The 1-address format $fxch$ instruction exchanges the contents of the register specified as the first operand and $st0$. If we do not specify any operands, then the processor exchanges $st0$ and $st1$ (the top of the stack, and the entry just below the top of the stack).

Store Instructions

Semantics	Example	Explanation
fst *mem*	fst dword [eax]	[eax] ← st0
fst *reg*	fst st4	st4 ← st0
fstp *mem*	fstp dword [eax]	[eax] ← st0; pop the FP stack
fist *mem*	fist dword [eax]	[eax] ← int(st0)
fistp *mem*	fistp dword [eax]	[eax] ← int(st0); pop the FP stack

Table 5.22: Floating point store instructions

Let us now look at the store instructions in Table 5.22. The format is similar to the floating point load instructions. We have three variants of the basic fst instruction. The first variant requires a single memory operand. It stores the contents of $st0$ in the memory location specified by the memory operand. The second variant requires a FP register operand and stores the contents of $st0$ in the FP register.

The third variant uses the 'p' suffix which is a generic suffix and is used by many other instructions also. The $fstp$ instruction initially saves the value contained in $st0$ in the memory location specified by the first operand, and then pops the stack. Since the stack size is limited, it is often necessary to pop the stack to create more space. When we are storing $st0$, we are saving a copy of its contents in main memory. Hence, it makes sense to have a variant of the fst instruction that can free the entry from the stack by popping it.

x86 has additional support for conversion of floating point values to integers. We can use the $fist$ instruction that first converts the contents of $st0$ to a signed integer by rounding it and then saves it in the location specified by the memory operand. Note that we always use a modifier (byte/word/dword/qword) for memory operands such that we can specify the number

of bytes that need to be transferred. The *fist* instruction also supports the 'p' suffix (see the semantics of the *fistp* instruction in Table 5.22).

Example 86
Transfer the contents of st0 to eax by converting the save FP number to an integer.
Answer:

```
fist dword[esp]
mov eax, [esp]
```

5.4.2 Arithmetic Instructions

Let us now consider arithmetic instructions. The floating point ISA in x86 has rich support for floating point operations, and is thus extensively used in numerical computing. Let us start with the basic floating point add instruction, and take a look at all of its variants.

Add Instruction

Semantics	Example	Explanation
fadd *mem*	fadd dword [eax]	st0 ← st0 + [eax]
fadd *reg*, *reg*	fadd st0, st1	st0 ← st0 + st1 (one of the registers has to be st0)
faddp *reg*	faddp st1	st1 ← st0 + st1; pop the FP stack
fiadd *mem*	fiadd dword [eax]	st0 ← st0 + float([eax])

Table 5.23: Floating point add instructions

The semantics of the floating point add instructions is shown in Table 5.23. The basic *fadd* instruction has two variants. The first variant uses a single memory operand. Here, we add the value of the floating point number contained in the memory location to the contents of *st*0. The result is also stored in *st*0. The second variant of the *fadd* instruction uses two floating point registers as arguments. It adds the contents of the second register to the first register.

The *fadd* instruction follows the same pattern as the floating point load and store instructions. It accepts the 'p' suffix. The *faddp* instruction typically takes 1 argument, which is a register. We show an example of the instruction *faddp st1* in Table 5.23. Here, we add the contents of *st*0 to *st*1, and save the result in *st*1. Then, we pop the stack. For working with integers, we can use the *fiadd* instruction that takes the address of an integer in memory. It adds this integer to *st*0, and saves the results in *st*0.

Subtraction, Multiplication, and Division Instructions

x86 defines subtraction, multiplication, and division instructions that have exactly the same format as the *fadd* instructions, and all of its variants as shown in Table 5.23. Let us just show

the basic form of each instruction that uses a single memory operand in Table 5.24.

Semantics	Example	Explanation
fsub *mem*	fsub dword [eax]	st0 ← st0 - [eax]
fmul *mem*	fmul dword [eax]	st0 ← st0 * [eax]
fdiv *mem*	fdiv dword [eax]	st0 ← st0 / [eax]

Table 5.24: Floating point subtract, multiply, and divide instructions

Example 87
Compute the arithmetic mean of two integers stored in eax and ebx. Save the result (in 64 bits) in esp + 4. Assume that the data section contains the integer, 2, in the memory address two.

Answer:

```
; load the inputs to the FP stack
mov [esp], eax
mov [esp+4], ebx
fild dword [esp]
fild dword[esp + 4]

; compute the arithmetic mean
fadd st0, st1
fdiv dword [two]

; save the result (converted to 64 bits) to [esp+4]
; use the qword identifier
fstp qword [esp + 4]
```

5.4.3 Instructions for Special Functions

Semantics	Example	Explanation
fabs	fabs	st0 ← |st0|
fsqrt	fsqrt	st0 ← $\sqrt{st0}$
fcos	fcos	st0 ← cos(st0)
fsin	fsin	st0 ← sin(st0)

Table 5.25: Floating point instructions for special functions

The greatness of the x86 ISA is that it supports trigonometric functions, and complex mathematical operations such as the square root, and log operations (not covered in this book). Table 5.25 shows the x86 instructions for computing the values of special functions. The $fabs$ function computes the absolute value of $st0$, the $fsqrt$ function computes the square root, the $fcos$ and $fsin$ functions compute the sine and cosine of the value stored in $st0$ respectively. All of these instructions use $st0$ as their default operand, and also write the result back to $st0$.

Example 88
Compute the geometric mean of two integers stored in eax and ebx. Save the result (in 64 bits) in esp + 4.

Answer:

```
; load the inputs to the FP stack
mov [esp], eax
mov [esp+4], ebx
fild dword [esp]
fild dword[esp + 4]

; compute the geometric mean
fmul st0, st1
fsqrt

; save the result (converted to 64 bits) to [esp+4]
; use the qword identifier
fstp qword [esp + 4]
```

5.4.4 Compare Instruction

Semantics	Example	Explanation
fcomi *reg*, *reg*	fcomi st0, st1	compare st0 and st1, and set the $eflags$ register (first register has to be st0)
fcomip *reg*, *reg*	fcomi st0, st1	compare st0 and st1, and set the $eflags$ register; pop the FP stack

Table 5.26: Floating point compare instructions

The x86 ISA has many compare instructions. In this section, we shall present only one compare instruction called $fcomi$ that compares two floating point values saved in registers, and sets the $eflags$ register. Table 5.26 shows the semantics of the $fcomi$ instruction with and

without the 'p' suffix. Once, the $eflags$ register is set, we can use regular branch instructions for implementing control flow within the program. Note that in x86 we need to use the condition codes for unsigned comparison in this case. Most of the time programmers make the mistake of using the condition codes for signed comparison such as l, le, g, or ge for testing the results of floating point comparison. This leads to wrong results. We should instead use the a (above) and b (below) condition codes.

Let us now consider an example (Example 89) that computes the value of $sin(2\theta)$, and verifies if it is equal to $2sin(\theta)cos(\theta)$. The readers should recall from their high school trigonometry class that both these expressions are actually equal, and one can be derived from the other. Example 89 experimentally verifies this fact for any given value of θ. We compute the value of $sin(2\theta)$ and $2sin(\theta)cos(\theta)$, and compare them using $fcomi$. Note that floating point arithmetic is approximate (see Section 2.4.6). Hence, the correct way to compare floating point numbers is to first subtract them, compute the absolute value of the difference, and compare the difference with a threshold. The threshold is typically a small number (10^{-5} in our case). If the difference is less than a threshold, we can infer equality.

Example 89

Compare $sin(2\theta)$ and $2sin(\theta)cos(\theta)$. Verify that they have the same value for any given value of θ. Assume that theta is stored in the data section at the label theta*, and the threshold for floating point comparison is stored at label* threshold*. Save the result in eax (1 if equal, and 0 if unequal).*

Answer:

```
; compute sin(2*theta), and save in [esp]
fld dword [theta]
fadd st0  ; st0 = theta + theta
fsin
fstp dword [esp]

; compute (2*sin(theta)*cos(theta))
fld dword [theta]
fst st1
fsin
fxch
fcos       ; st0 = cos(theta)
fmul st1   ; st0 = sin(theta) * cos (theta)
fadd st0   ; st0 = 2 * st0

; compute the modulus of the difference
fld dword [esp]  ; load (sin(2*theta))
fsub st0, st1
fabs

; compare
```

```
fld dword [threshold]
fcomi st0, st1  ; compare
ja .equal
mov eax, 0
jmp .exit

.equal:
        mov eax, 1
.exit:
```

After the end of a function, it is time to clean up the floating point registers, and stack such that another function can use them. Let us conclude this section by taking a look at the cleanup instructions.

5.4.5 Stack Cleanup Instructions

Semantics	Example	Explanation
ffree *reg*	ffree st4	Free st4
finit	finit	Reset the status of the FP unit including the FP stack and registers

Table 5.27: Floating point stack cleanup instructions

Table 5.27 shows two instructions for cleaning up the FP stack. The $ffree$ instruction sets the status of the register specified as an operand to empty. Using $ffree$ to free all the registers is a quick solution. For freeing the entire stack we need to invoke the $ffree$ instruction iteratively. For deleting the entire FP stack, a cleaner solution is to use the $finit$ instruction that does not take any arguments. It resets the FP unit, frees all the registers, and resets the stack pointer. The $finit$ instruction ensures that an unrelated function can start from a clean state.

5.5 Encoding the x86 ISA

We have taken a look at a wide variety of x86 instructions, addressing modes, and instruction formats. It is truly a CISC instruction set. However, the process of encoding is more regular. Almost all the instructions follow a standard format. In the case of ARM and *SimpleRisc* , we described the process of encoding instructions in great detail. We shall refrain from doing this here for the sake of brevity. Secondly, an opcode in x86 typically has a variety of modes, and prefixes. We do not want to digress from the main theme of this book by describing x86 in such level of detail. Let us start out by looking at the broad structure of an encoded machine instruction.

5.5.1 High Level View of x86 Instruction Encoding

Figure 5.4 shows the structure of an encoded instruction in binary.

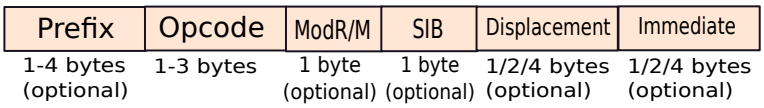

Figure 5.4: x86 binary instruction format

The first set of 1-4 bytes are used to encode the prefix of the instruction. The *rep* prefix is one such example of a prefix. There are many other kinds of prefixes that can be encoded in the first group of 1-4 bytes.

The next 1-3 bytes are used for encoding the opcode. Recall that the entire x86 ISA has hundreds of instructions. Secondly, the opcode also encodes the format of operands. For example, the *add* instruction can either have its first operand as a memory operand, or have its second operand as a memory operand. This information is also a part of the opcode.

The next two bytes are optional. The first byte is known as the ModR/M byte, which specifies the address of the source and destination registers, and the second byte is known as the SIB (scale-index-base) byte. This byte records the parameters for the base-scaled-index and base-scaled-index-offset addressing modes. A memory address might optionally have a displacement (also referred to as the offset in this book) that can be as large as 32 bits. We can thus optionally have 4 more bytes in an instruction to record the value of the displacement. Lastly, some x86 instructions accept an immediate as an operand. The immediate can also be as large as 32 bits. Hence, the last field, which is again optional, is used to specify an immediate operand.

Let us now discuss the ModR/M and SIB bytes in more detail.

ModR/M Byte

The ModR/M byte has three fields as shown in Figure 5.5.

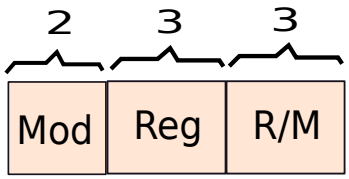

Figure 5.5: The ModR/M byte

The two MSB bits of the ModR/M byte contain the *Mod* field. The *Mod* field indicates the addressing mode of the instruction. It can take 4 values as shown in Table 5.28.

The *Mod* field indicates the addressing mode of one of the operands. It can either be a register or a memory operand. If it is a memory operand, then we have three options. We

Mod bits	Semantics
00	We use the register indirect addressing mode for one of the operands. When R/M = 100, we use the base-scaled-index addressing mode, and there is no displacement. The ids of the scale, index, and base are specified in the SIB byte. When R/M = 101, the memory address only consists of the displacement. The rest of the values of the R/M bits specify the id of the base register as shown in Table 5.29. Other than the case of R/M=101, the rest of the combinations of the R/M bits are not associated with a displacement (assumed to be 0).
01	We use a single byte signed displacement. If R/M = 100, then we get the ids of the base and index registers from the SIB byte.
10	We use a 4 byte signed displacement. If R/M = 100, then we get the ids of the base and index registers from the SIB byte.
11	Register direct addressing mode.

Table 5.28: Semantics of the Mod field

Register	Code
eax	000
ecx	001
edx	010
ebx	011
esp	100
ebp	101
esi	110
edi	111

Table 5.29: Register encoding

can either have no displacement (Mod = 00), a 8 bit displacement (Mod = 01), or a 32-bit displacement (Mod=10). If it is a register operand, then the Mod field has a value of 11.

The important point to note is that for all the memory address modes, if the R/M bits are equal to 100, then we need to use the information in the SIB byte for computing the effective memory address.

The *Reg* field encodes the second operand if it is a register. Since both the operands cannot be memory operands, we use the Mod and R/M bits for encoding one of the operands that might be a memory operand (source or destination), and use the *Reg* field for encoding the other operand, which has to be a register. The encoding for the registers is shown in Table 5.29.

For floating point instructions, the default register operand is always *st*0. Some instructions accept another FP register operand. For such instructions, we use register direct addressing (Mod = 11). We use the R/M bits for specifying the id of the additional FP register. We use 3 bits to encode the index of the register. For example, *st*0 is encoded as 000, and *st*6 is encoded as 110. For the rest of the instructions that either assume default operands, or have a single memory operand, we use the same format as defined for integer instructions.

SIB Byte

The SIB byte is used to specify the base and index registers (possibly with scaling). For example, it can be used to encode memory operands of the form [eax + ecx*4]. Recall that to use the SIB byte it is essential to set the Mod field in the ModR/M register to 100. This indicates to the processor that the SIB byte follows the ModR/M byte.

The structure of the SIB byte is shown in Figure 5.6.

The SIB byte has three fields – *scale*, *index*, and *base*. The effective memory address (before considering the displacement) is equal to $base + index \times scale$. The *base* and *index* fields point to integer registers. Both of them are 3 bits each (can encode up to 8 registers), and use the encoding shown in Table 5.29. The two MSB bits are used to specify the scale. We can have

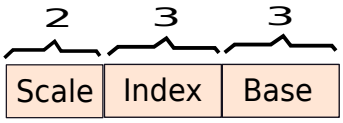

Figure 5.6: The SIB byte

four values for the scale in x86 instructions namely 1 (00), 2 (01), 4 (10), and 8 (11).

Rules for Encoding Memory Operands

Note that some rules need to be followed while encoding memory operands. The *esp* register cannot be an index, and if the value of the Mod field is 00, then *ebp* cannot be a valid *base* register. Recall that if we set the R/M bits to 101 (id of *ebp*), when the Mod field is 00, then the memory address is only a displacement. Or, in other words we can use memory direct addressing here by directly specifying its address.

If (Mod = 00), then in the SIB byte *ebp* cannot be a valid base register. If we specify the base register as *ebp* in the SIB byte, then the processor calculates the effective memory address based on the value of the scale and the index.

Example 90
Encode the instruction add ebx, [edx + ecx*2 + 32]*. Assume that the opcode for the add instruction is 0x03.*
Answer: *Let us calculate the value of the ModR/M byte. In this case, our displacement fits within 8 bits. Hence, we can set the Mod bits equal to 01 (corresponding to an 8 bit displacement). We need to use the SIB byte, because we have a scale, and an index. Thus, we set the R/M bits to 100. The destination register is ebx. Its code is 011 (according to Table 5.29). Thus, the ModR/M byte is* 01011100 *(equal to 0x5C).*

Now, let us calculate the value of the SIB byte. The scale is equal to 2. This is encoded as 01. The index is ecx (001), and the base is edx (010). Hence, the SIB byte is: 01 001 010 = 4A. The last byte is the displacement, which is equal to 0x20.

Thus, the encoding of the instruction is $\boxed{03\ 5C\ 4A\ 20}$ *in hex.*

5.6 Summary and Further Reading

5.6.1 Summary

Summary 5

1. *The x86 ISA is a family of CISC instruction sets that is primarily used by Intel and AMD processors.*
   (a) *The original x86 ISA used by 8086 processors used a 16-bit ISA.*
   (b) *Since the mid eighties, x86 processors have moved to the 32-bit ISA.*
   (c) *Finally, since 2003, most of the high end x86 architectures have moved to the 64-bit ISA.*
   (d) *The basic structures of all the ISAs is the same. There are minor differences in the syntax.*
2. *The 8 basic registers of the 16-bit x86 ISA are – ax, bx, cx, dx, sp, bp, si, and di. We use the 'e' prefix in 32-bit mode, and the 'r' prefix in 64-bit mode.*
3. *Additionally, the 16-bit x86 ISA has the ip register to save the program counter, and the flags register to save the results of the last comparison, and other fields that instructions may use.*
4. *The x86 ISA predominantly uses instructions in the 2-address format. The first operand is typically both the source, and the destination. Secondly, one of the operands can be a memory operand. It is thus possible to fetch the value of a memory location, operate on it, and write it back to memory, in the same instruction.*
5. *x86 processes see a segmented memory model. The entire memory space is partitioned into different segments. Instructions reside in the code segment by default, and data resides in the data or stack segments by default. It is in general not possible for instructions to access segments that they typically are not meant for. For example, it is in general not possible for a store instruction to change the contents of an instruction in the code segment.*
   (a) *In the 16-bit mode, the top 16 bits of the starting address of each segment are stored in a segment register.*
   (b) *The effective memory address specified by a memory instruction is added to the address contained in the segment register (after left shifting it by 4 positions) to compute the actual memory address.*
   (c) *In later ISAs (32 and 64-bit mode), the contents of segment registers are looked up in segment descriptor tables (referred to as the LDT and GDT) for obtaining the starting address of segments. To speed up memory accesses, processors typically use a small memory structure known as a segment descriptor cache that keeps the most recently used entries.*
6. *x86 integer instructions:*
   (a) *The mov instruction is one of the most versatile instructions. It can move values between two registers, or between registers and memory addresses. It can also be used to load immediates in registers or memory locations.*

(b) x86 defines a host of other arithmetic, and branch instructions.

(c) String instructions are a unique feature of the x86 ISA. They can be used to transfer large amounts of data between memory locations. To compress an entire loop of string instructions into one instruction, we typically use the rep prefix that repeats a given instruction n times, where n is the value stored in the ecx register.

7. *The x86 floating point registers can either be accessed as normal registers (st0 ... st7), or as values on a floating point stack. Most of the floating point instructions operate on st0, which is the top of the stack.*

8. *There is no direct way to load immediates into the FP registers. We need to first load them into memory, and then load them to the floating point stack. x86 has instructions for computing complex mathematical operations (such as square root), and trigonometric functions directly.*

9. *Encoding the x86 instruction set is relatively simpler, since the encoded form has a very regular structure.*

   (a) We can optionally use 1-4 bytes to encode the prefix.

   (b) The opcode's encoding requires 1-3 bytes.

   (c) We can optionally use two additional bytes known as the ModR/M and SIB bytes to encode the address of operands (both register and memory).

   (d) If the memory operand uses a displacement (offset), then we can add 1-4 bytes for encoding the displacement after the SIB byte.

   (e) Lastly, the x86 ISA accepts 32-bit immediate values. Hence, we can use the last 1-4 bytes to specify the value of an immediate operand if required.

5.6.2 Further Reading

The most accurate source of information is the x86 developer manuals released by Intel on their website [int, , INTEL, 2010].

For the sake of brevity, we have only discussed the popularly used instructions. However, there are many instructions in the x86 ISA that might prove to be useful in a specific set of scenarios, which we have not covered in this book. Intel's software developer manuals are always the best places to find this information. Secondly, we have only discussed the basic x86 ISA. The reader should definitely look at the extensions to the x86 ISA such as the MMX™, SSE, and 3d Now! (by AMD) extensions. These extensions add vector instructions, which can operate on arrays of data. These instructions are used in graphics, games, and scientific applications. The Intel AVX instruction set is the latest addition in the long line of x86 ISAs. It introduces 512 bit registers that can contain multiple integers. The interested reader should definitely take a look at this instruction set and try to write programs with it. In this book, we shall show an example using the SSE instruction set in Section 11.5.3.

The reader can additionally refer to books that describe the x86 instruction set in great detail, and have a wealth of solved examples. The following books [Cavanagh, 2013, Das, 2010, Kumar, 2003] are useful references in this regard.

Exercises

x86 Machine Model

Ex. 1 — What are the advantages of the segmented addressing mode? Why do modern x86 processors need the LDT and GDT tables?

Ex. 2 — Explain the memory addressing modes in x86.

Ex. 3 — Describe the floating point registers and the floating point stack in x86.

* **Ex. 4** — We can specify an entire 32-bit immediate in a single instruction in x86. Recall that this was not possible in ARM and *SimpleRisc* . What are the advantages and disadvantages of having this feature in the ISA?

* **Ex. 5** — We claim that using a stack based architecture makes the software very portable. It does not need to be aware of the number and semantics of registers in an ISA. Comment on this statement, and try to find other reasons for preferring a stack based machine.

** **Ex. 6** — Given an arithmetic expression containing floating point operands, how can we evaluate it using a floating point stack? What should be the order of loading and operating on operands? [HINT: A regular arithmetic operation such as – (1 + 2.5) * 3.9 – is called an infix expression. To evaluate expressions using a stack, we need to convert it into a postfix expression of the form – 1 2.5 + 3.9 *. Here, we first push 1 and 2.5 on the stack, add the result, push 3.9 on the stack, and multiply the first two entries. The reader should read more about postfix expressions in textbooks on discrete mathematics.]

Assembly Programming using Integer Instructions

Ex. 7 — Write x86 assembly code snippets to compute the following:

i) $a + b + c$

ii) $a + b - c/d$

iii) $(a + b) * 3 - c/d$

iv) $a/b - (c * d)/3$

v) $(a \ll 2) - (b \gg 3)$

Ex. 8 — Write an assembly program to convert an integer stored in memory from the little endian to the big endian format.

Ex. 9 — Compute the factorial of a positive number using an iterative algorithm.

Ex. 10 — Compute the factorial of a positive number using a recursive algorithm.

Ex. 11 — Write an assembly program to find if a number is prime.

Ex. 12 — Write an assembly program to test if a number is a perfect square.

Ex. 13 — Write an assembly program to test if a number is a perfect cube.

Ex. 14 — Given a 32-bit integer, count the number of 1 to 0 transitions in it.

Ex. 15 — Write an assembly program that checks if a 32-bit number is a palindrome. A palindrome is a number which is the same when read from both sides. For example, 1001 is a 4-bit palindrome.

Ex. 16 — Write an assembly program to examine a 32-bit value stored in *eax* and count the number of contiguous sequences of 1s. For example, the value:

$$01110001000111101100011100011111$$

contains six sequences of 1s. Write the final value in register *ebx*.

Ex. 17 — Write an assembly program to count the number of 1's in a 32-bit number.

* **Ex. 18** — Write an assembly program to find the smallest number that is a sum of two different pairs of cubes. [Note: 1729 is known as the Hardy-Ramanujan number. $1729 = 12^3 + 1^3 = 10^3 + 9^3$].

** **Ex. 19** — In some cases, we can rotate an integer to the right by n positions (less than or equal to 31) so that we obtain the same number. For example: a 8-bit number 01010101 can be right rotated by 2, 4, or 6 places to obtain the same number. Write an assembly program to *efficiently* count the number of ways we can rotate a number to the right such that the result is equal to the original number.

*** **Ex. 20** — Write an assembly language program to find the greatest common divisor of two binary numbers u and v. Assume the two inputs (positive integers) to be available in *eax* and *ebx*. Store the result in *ecx*. [HINT: The gcd of two even numbers u and v is $2*gcd(u/2, v/2)$]

Ex. 21 — Write an assembly program that uses string instructions to set the value of a range of memory addresses to 0. Reduce the code size by using the *rep* prefix.

Assembly Programming using Floating Point Instructions

Ex. 22 — How do you load and store floating point numbers?

Ex. 23 — Write an assembly program to find the roots of the equation $x^2 - x - 1 = 0$. Recall that the roots of a quadratic equation of the form $ax^2 + bx + c$ are equal to $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Ex. 24 — Verify the following trigonometric identities for random values of θ using assembly programs. Use the *rdrand* instruction that loads a random 32-bit integer into a register.

S. No.	Identity
1	$sin^2(\theta) + cos^2(\theta) = 1$
2	$sin\left(\frac{\pi}{2} - \theta\right) = cos(\theta)$
3	$cos(\theta + \phi) = cos(\theta)cos(\phi) - sin(\theta)sin(\phi)$
4	$sin(\theta) + sin(\phi) = 2sin\left(\frac{\theta+\phi}{2}\right) cos\left(\frac{\theta-\phi}{2}\right)$

Ex. 25 — Assume that we have two arrays of 10 floating point numbers, where the starting addresses of the arrays are stored in *eax* and *ebx* respectively. Find the arithmetic mean (AM), geometric mean (GM), and harmonic mean (HM) using assembly routines. Verify that $AM \geq GM \geq HM$.

* **Ex. 26** — Let us compute the value of the constant e using an assembly program. Use the following mathematical expression.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{10!}$$

** **Ex. 27** — For random values of θ show that the following identity holds:

$$sin(\theta) = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \ldots$$

x86 ISA Encoding

Ex. 28 — What are the values of the SIB and ModR/M bytes for the instruction, *mov eax, [eax + ebx*4]*?

Ex. 29 — What are the values of the SIB, ModR/M, and displacement bytes for the instruction, *mov eax, [eax + ebx*4 + 32]*?

Ex. 30 — What is the value of the $ModR/M$ byte when we need to specify a memory address that does not have any base or index registers? Assume that the value of the *reg* field is 000.

* **Ex. 31** — Assume that we have an instruction that has two operands: *eax* and [*ebp*]. How do we encode it (specify the values of the relevant bytes)?

* **Ex. 32** — What are the values of the SIB and ModR/M bytes for the instruction, *mov eax, [ebx*4]*?

Design Problems

Ex. 33 — Write an x86 assembly emulator that can read an assembly file, and execute each assembly instruction one by one.

Ex. 34 — Use the GNU compiler to generate an assembly file for a test program written in C using the command, *gcc -S -masm=intel*.