Contents

Appendices

Chapter 1

Introduction

Assembly language programming has its own challenges. It is almost always possible to write programs in Assembly language that would run faster than the corresponding programs written in high level languages. Sometimes it is essential to write programs only in assembly language because the high level language is not powerful enough to perform certain processor specific operations.

Intel's IA32 architecture is a very common platform for programming. It is used with almost all PCs today. Several PCs run the GNU/Linux operating system. While GNU/Linux operating system can also run on processors such as Sparc, PowerPC, ARM, MIPS etc., the processors with Intel's IA32 architecture (such as Pentium class processors) are the most popular ones. Assembly language programming for GNU/Linux is done using an instruction format different from the one used for Microsoft Windows based platforms. GNU/Linux based systems use GNU tools such as `gcc` and `gas` for translating the Assembly language programs into object code. These tools use AT&T format of Assembly language while the Intel format of Assembly language is used in Windows based systems. The two formats are very different from each other (even when the instructions in two formats translate to the same machine instruction.)

This book is an attempt to teach assembly language programming on GNU/Linux operating system based IA32 processors.

1.1 Why Assembly language?

While programming in the high level languages, we need to use a compiler to translate programs to the machine language. In this process, Some instruction are are used infrequently and some are never used. This often limits the use of specialized instructions provided by the processors. Use of these specialized instructions yields higher execu-

tion performances which can not be made possible using high level language. Assembly language provides a very powerful mechanism to use all the features provided by the processor. Therefore it is possible to use specialized processor instructions and construct programs in a way to almost always achieve better performance.

Programming in Assembly language involves use of textual representation of the machine instruction that is executed by the processor. An instruction in the Assembly language program is a unit instruction for the machine. Each of these unit instructions are translated to the machine instruction that can be executed by the processor. The process of translation is carried out by a program called assembler. While the Assembly language instructions are human readable, the machine instructions are composed of several electronic signals represented by group of bits. At best, a machine instruction can be seen as a sequence of numbers often represented in binary or hexadecimal number system. The processor executes machine instructions and thereby performs the semantics associated with the instructions.

An Assembly language program, unlike other high level language programs, represents a symbolic sequence of machine instructions. Thus there are no high level constructions in an Assembly language such as `if-then-else`, loops, input-output etc. The processors normally do not have such constructs in their instruction set. While some processors may have certain kinds of instructions to simplify high level language constructions, these are not powerful enough until combined with several other instructions. It is needless to say that the Assembly language programs provide a powerful way of utilizing the processor to execute programs efficiently.

In a typical program development environment, programmers write their programs in high level languages such as C, C++, Java[1] etc. The programs written in these languages are compiled to the machine instructions (sometimes called binary programs or object codes) by a process of compilation. These programs can be executed by the processor thus implementing the behavior of the original program in a high level language. Some compilers first compile the programs to Assembly language instructions which are then assembled separately to the machine instructions. Notably among such compilers are the GNU compilers available in GNU/Linux distribution. In the most common usages, the compilers are not executed by the programmer directly. The `gcc` or `cc` tool provides an integrated front-end environment for running them.

Programs written in Assembly language are not portable across different platforms. If the requirement is to develop programs that can run on different platforms, the programming must be done in high level languages. However, it is impossible to perform many specific tasks in

[1]Java programs are compiled to an instruction set of Java virtual machine. These programs are then interpreted by the virtual machine implementation on the processor

high level languages and these can be done only using the Assembly language programming. These include tasks that are highly specific to the processor, such as, setting up various processor control registers. On the other hand, there are some tasks which can be implemented using any of the languages but are carried out most efficiently using the Assembly language. Consider for example, the operations involving specific bits of data are implemented efficiently by programming in the Assembly language. Such bit-wise operations are extremely useful in applications such as implementing fast processing for computer graphics.

Programming in Assembly language exposes details of processor architecture which helps in understanding the processor. It helps those who would ever write the compiler back-ends to generate processor specific instructions. It helps those who develop interfaces for programming environments. An example of this is an application written in multiple programming languages. In such an application program written in one language need to access functions written in a different language. This necessitate the use of cross-language interfaces written mainly in Assembly language. In this book, focus is also given to such kind of programming using Assembly language.

1.2 How to assemble programs?

On GNU/Linux operating system based distributions, one of the following two methods are used to assemble a program.

1. The front-end environment `gcc` can be used to assemble the program. Often `gcc` is mistaken for a C compiler. It is instead a front-end that would invoke the appropriate compiler and other related programs (such as preprocessor, linker etc.). By default, `gcc` uses input file extensions to find out which compiler to use. For the files ending with `.s` or `.S` extensions, the `gcc` uses an assembler to convert the Assembly language program to the machine language program.

2. The `as` program can be used to assemble the source programs. As a matter of fact, the `gcc` also executes the `as` program to assemble the Assembly language programs.

In some ways, it is customary for the book writer to give the first program that writes `hello world` on the screen. In figure 1.1 we provide an Assembly language program `hello.S` that prints `hello world`.

We will not try to understand this program right now. However this program can be typed without omitting any character such as comma, dot or space. Notice the capitalized extension in the file name `hello.S`.

```
#include <asm/unistd.h>
#include <syscall.h>
#define STDOUT 1
.data
hello:
        .ascii "hello world\n"
helloend:
.text
.globl _start
_start:
        movl    $(SYS_write),%eax // SYS_write = 4
        movl    $(STDOUT),%ebx    // fd
        movl    $hello,%ecx       // buf
        movl    $(helloend-hello),%edx   // count
        int     $0x80

        movl    $(SYS_exit),%eax
        xorl    %ebx,%ebx
        int     $0x80
        ret
```

Figure 1.1: The `hello.S` program

In order to assemble this program to a runnable `a.out` program, the following three commands can be given in that sequence.

```
/lib/cpp hello.S hello.s
as -o hello.o hello.s
ld hello.o
```

The first command is used to run a C pre-processor. Notice that in the Assembly language program, we have first three lines starting with a #. C programmers would recognize them as pre-processor directives to include files and to define constants. The C pre-processor is used to handle statements like these. Output of the C pre-processor is another Assembly language program that is named as `hello.s` (notice the extension in small case letter). The second command is used to generate an object file using the `as` assembler. This command converts the `hello.s` assembly language program to object code in file `hello.o`. Finally the third command is used to run the linker to generate executable program `a.out` from the object file.

EXERCISES:

1.1 Try executing '`as hello.S`'. Does it work? Try to figure out what hap-

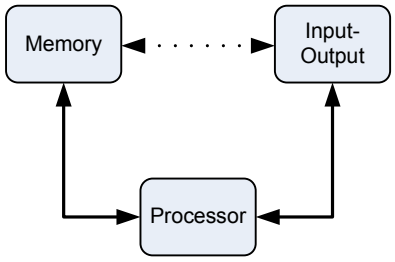


Figure 1.2: Subsystems of a digital computer

pens.

1.2 Look at `hello.s` and compare it with `hello.S`.

1.3 Look at the documentation of the `as` program. Use '`info as`' command and figure out the details regarding various command line options.

1.3 Structure of a digital computer

Broadly a digital computer has three subsystems (figure 1.2). These include processor, memory and input-output subsystem. Processor (or CPU) is the subsystem that is responsible for execution of programs. In figure 1.2, logical paths for flow of data are also shown. All digital computers provide capability for the processor to be able to read and write into memory and input-output subsystem. During the course of execution of programs, it reads instructions from the memory and then executes them. While executing an instruction, it may additionally require data to be read from the memory, or the result of computations to be stored back in the memory.

Both processor and memory are electronic devices. The interface to the external world usually involves mechanical devices as well. For example, a keyboard on a computer is a physical device thorough with a person interacts with the computer. In a similar manner many other physical devices such as screen, mouse, printer, hard disk, CDROM interface require a different way of interacting with the computer. A digital computer provides an input-output subsystem to handle such devices.

Sometimes a digital computer may also provide an optional path between memory and input-output subsystem as shown in figure 1.2. This path is normally used for direct memory access (DMA) operations and provides direct transfer of data between memory and input-output subsystems. Such operations make the system more efficient compared to the cases when the transfer is always through the processor.

This path is commonly used for bulk data transfer such as in disk read-write.

Programs are typically kept in files which are in turn stored in a storage medium. The storage medium is usually a hard disk but can be in many other forms. During the execution of a program, the program is "loaded" in memory from the storage and then control is passed to the first instruction that must be executed. The CPU, or processor always picks up its instructions from the memory subsystem and executes them. After the execution of one instruction, it then picks up another and executes them. Modern processors may execute multiple instructions at the same time but will retain the same semantics as the serial execution model. This model is also known as von-Neumann model of computing. In this model, the program instructions are stored in the memory which are executed one at a time in a sequential order.

1.4 Stored Program Model

A processor executes a program in a sequential manner. The machine instructions are stored in memory and read by the processor in the order of storage.

In order to fetch instructions from memory, the processor maintains a register called IP (instruction pointer) or PC (program counter). This register stores the address of the memory from where the instructions are read. After reading an instruction, the IP register is incremented by the size of the instruction such that it contains the address of the memory where the next instruction is stored. Upon finishing execution of the current instruction, the next instruction is fetched and this process continues as long as the processor is fed with power.

In the beginning when the processor is powered on, it starts execution from a predefined location in the memory. A predefined program (usually in a Read-Only memory) is executed in this manner. In a PC this program is also known as BIOS (basic input-output system). This program implements testing of on-board peripherals and booting process. Soon due to the booting process, an operating system gets loaded which permits the user programs to be loaded on demand and executed.

Linux operating system is a multi-user, multi-tasking operating system. There can be several programs running simultaneously on a time sharing basis. In such a model, each process is given a virtual machine (its own share of time on the CPU and its own memory space). After finishing its execution, the process returns control back to the operating system.

The instructions in a process are executed one after another. Current IA32 processors permit simultaneous execution of several instructions but semantically a sequential execution is maintained. That is,

in the parallel execution environment, no instruction will be allowed to modify a register or memory till all earlier instructions that were to modify the same register or memory have finished the execution.

There are several instructions that operate on the IP register explicitly. These instructions are called jump or call class instructions. In a jump instruction, control is passed to another location (by modifying the IP). In a call instruction also, the control is passed to another location but the current value of the IP is saved so that the control may come back to the original location upon execution of an instruction called return. Call and return type of instructions are used to implement semantics of the execution of a procedure or a function.

As instructions are executed, they modify the state of the processor and system. The state constitutes registers of the processor and memory.

In a stored program model, the order of execution of the next instruction is implicitly known. This model has been in practice since the birth of the first set of computers and is still considered a very powerful model.

Chapter 2

IA32 Processors

The processors that implement the Intel's IA32 architecture include Intel's Pentium, Pentium-Pro, Pentium-II, Pentium-III, Pentium 4, Xeon and many other processors. Similarly, several other processors from other manufacturers, such as AMD 5x86, AMD-K5, AMD-K6, AMD-Athlon, AMD-Duron, AMD-Optron, Cyrix 5x86, Cyrix M1, Cyrix M2, VIA Cyrix III, Transmeta Crusoe etc. also implement the IA32 instruction set architecture.

IA32 processors can operate in one of the following three modes of operation.

- Real mode
- Protected mode
- VM86 mode

In real mode of operation, IA32 processors emulate the behavior of a fast 16-bit 8086 processor. This is the default mode upon power-on and provides no security among applications. In the protected mode of operation, the applications can be secured from each other and privileges can be granted to perform certain kind of operations. The protected mode architecture does not provide compatibility to the 8086 architecture. Most 8086 applications can not be run in this mode. IA32 architectures provide yet another mode called VM86 to handle this condition. VM86 mode is a special 8086 compatibility mode that can be made to run under the protected mode environment. Thus many 8086 based applications can run in this mode while employing the security privileges of the protected mode. Many IA32 processors can also operate in a system management mode that is primarily intended for the operating system to perform tasks such as power saving etc. The discussions on this mode are beyond the scope of this book.

The operating system support for the virtual memory is provided in the protected mode. GNU/Linux applications, therefore, run in the

protected mode. GNU/Linux provides a flat memory architecture to the application where applications can operate on the memory using 32 bit addresses. This kind of processor architecture visible to the application makes the job of Assembly language programmer a lot simpler.

In this chapter, we will review the basic execution environment of IA32 processors as provided in GNU/Linux. This includes the register set architecture and the operand addressing modes as visible to the applications.

2.1 Basic execution environment

The IA32 processors provide a certain view of the memory and CPU registers to the tasks. While these resources may be used in several ways, the operating systems use them only in certain specific ways. For example, in GNU/Linux operating system the memory view is provided in a flat architecture that is simpler to understand.

In a flat architecture, memory is organized as a big array of addressable units. In case of IA32 processors, an addressable unit of the memory is a byte. An address provides the location or index in this array whose value is read or written into. Often CPU registers are used to store the addresses of memory locations. IA32 processors are 32-bit architectures and provide 32-bit registers which are used to store the address of memory locations. Therefore total virtual addressability provided by the GNU/Linux operating system is 4 G Bytes (2^{32} bytes). The addressable unit in IA32 processors is a byte and hence addresses identify a memory location that is one byte wide. As we shall see later, the operands larger than one byte occupy more than one memory locations but are identified by a single address.

The operating system does not permit the entire address range (*i.e.* 4GBytes) to be used in the user mode. The Linux kernel is mapped on certain address range in this area.

There are several registers in the CPU, many of which can be used only for specific purposes and are used by the operating system. The normal programs do not have sufficient privileges to modify them.

In this section we describe the basic user mode execution environment as provided by the Linux operating system.

2.1.1 CPU Registers

The IA32 processor architecture provides 8 general purpose registers that are used by the user mode tasks in Linux. From the operational view point these registers are divided in two groups. The registers (figure 2.1) in the first group (group A) are known as `eax`, `ebx`, `ecx` and `edx`. The registers in the second group (group B) are known as `esi`, `edi`, `esp` and `ebp`. All registers are 32-bit wide. Lower order 16 bits of

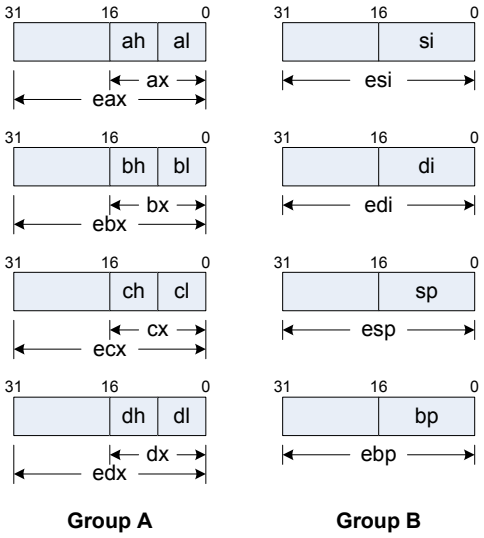
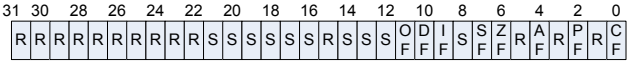


Figure 2.1: IA32 General Purpose Registers

registers in group A can also be accessed using 16 bit register names (`ax`, `bx`, `cx` and `dx`). Similarly the lower order 16 bits of the registers in group B are known as registers `si`, `di`, `sp` and `bp`. The 16-bit registers in group A can also be accessed using 8-bit register names. All such registers are available as two 8-bit registers. For example, lower 8 bits of register `ax` are known as register `al` while the higher 8 bits of register `ax` are known as register `ah`. Registers in group B, though, can not be accessed using 8-bit names.

The register architecture is better explained with an example. Consider the case when register `eax` has a value 0x11223344. The values contained in the most significant and least significant bytes of register `eax` are 0x11 and 0x44 respectively. Since the register `ax` is same as the lower 16 bits of register `eax`, the value in register `ax` is 0x3344. Similarly, the value in register `ah` is 0x33 and that in register `al` is 0x44. By modifying an 8-bit register, the corresponding 16-bit or 32-bit registers also get modified. However, such modifications have no effect on the bits other than those getting modified.

IA32 processors also have several special purpose registers. One of these, `eflags` register, is used commonly in Assembly language programs. This register stores the status of various ALU (arithmetic and logic unit) operations, control and system flags. Most of the bits of this register can be operated upon individually by instructions provided in the IA32 instruction set. Figure 2.2 shows various bits of the `eflags` register. Some flags get modified implicitly as processor executes instructions while others are modified explicitly using processor



CF Carry Flag
PF Parity Flag
AF Auxiliary Carry Flag
ZF Zero Flag
SF Sign Flag
IF Interrupt status Flag
DF Direction Flag
OF Overflow Flag
R Reserved bits. (meaning not defined)
S System flags. (meaning out of scope of this book)

Figure 2.2: IA32 `eflags` Register

instructions.

EXERCISES:

2.1 If register `edi` contains a value 0x66337744, what will be the value in register `di`?

2.2 Under what conditions, adding 1 to the value of register `dx` will not be the same as adding 1 to the value of register `edx`?

2.3 If register `ah` has a value of 0x00, what will be the value of register `ax` after register `al` is set to a value one less than the value of register `ah`.

2.1.2 Assembly language instructions

An Assembly language instruction has two parts. The first part defines the operation to be performed and is represented by the mnemonic of the instruction. For example, the Assembly language program in the figure 1.1 has several instructions. In the third instruction from the end, `xorl %ebx,%ebx`, the mnemonic `xorl` is used to define the operation. The operation defined by this instruction is to perform the bit-wise xor of two operands.

The second part of the instruction defines the operands for the instruction. In some cases, the operands might be implicit and are not given. In this instruction, there are two operands, both of which are given as register `ebx`. Thus both operands of this instruction are 32 bits in size.

The operands of an instruction are of two types – source operand and destination operand. The source operands provide input values to the operation performed by the an instruction. The result of the operation are stored in the destination operand. In an instruction source and

destination operands may be the same in which case, the value prior to the execution of the instruction provides the input and the operand changes after the execution of the instruction to store the result of execution.

As instructions are assembled by the assembler, a track of the memory addresses of the instructions is maintained. Some instructions have the memory address of another instruction as an argument. However, it is difficult for the programmer to keep track of the addresses of the instructions. For this purpose, an instruction may be labeled using names and this name may be referred to in the other instructions as the address of this instruction. For example, in figure 1.1, several labels are used – `hello`, `helloend` and `_start`. The label `_start` defines the address of the `movl` instruction. Similarly, the label `hello` defines the address of the memory location where the first byte of the string `"hello world\n"` is stored. As we shall see later, there are several non-processor-instructions used in the program. These are called pseudo-ops or assembler directives and are used by the programmer to assist assembler in the process of assembly. A pseudo-op `.ascii` is used to initialize a sequence of memory locations to the ASCII codes of characters from given string. Finally the label `helloend` is used to define the address of the memory location just after the string. Thus an expression `helloend-hello` gives the length of the string. In the program, this expression is used in a processor instruction to store length of string in register `edx`.

Finally, the last word about the program. Even though `_start` symbol is not referred to in the program, it is used as a symbol to define the address of the instruction in the program that is executed first. Thus the execution of the program always starts from location `_start` and carries on till the program makes a call to `exit` system call in Linux.

2.1.3 Operand Sizes

In the IA32 architecture, instructions can have operands of variable sizes. Typical sizes of the operands are 8 bits, 16 bits and 32 bits. The operands that are eight bits in size, are known as byte operands. Similarly the operands of sizes sixteen bits and thirty two bits are known as word and long operands.

Some instructions can also have 64-bit operands. A few specialized instructions can also operate on 1-bit wide operands. The instructions in the IA32 architecture can have 0, 1 or 2 operands.

The operands of instructions may be in the registers, in the memory or part of the instruction itself.

For the register operands, registers `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh` or `dl` can be used if the operand size is 8-bit. If the operands are of 16-bit size, registers `ax`, `bx`, `cx`, `dx`, `si`, `di`, `sp` or `bp` can be used. Similarly, if

the operands are of 32-bit size, registers `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp` or `ebp` can be used.

In IA32 processors, instructions can have multiple operands. Some instructions have no operand, while some others have one, two or three operands. However in all instructions a maximum of one operand can be specified as a memory operand. Thus, in the instructions which require only one operand, this operand may reside in the memory. Similarly, for the instructions which require two operands, one of the two operands may reside in the memory while the other operand must be a constant or a value in one of the registers.

2.1.4 Memory Model

The memory in a computer system can be thought of as an array of bytes. The index of this array is known as the address of the memory location. The value of the array element is known as the content of the memory location.

As explained earlier, instructions may have operands in the memory. In such cases, the memory addresses of the operands are specified in the instructions. There are several ways in which addresses may be specified in an instruction and these are discussed later in this chapter. IA32 processors support two kinds of addressing mechanisms. In the first kind, the addresses are 16-bit wide. In this addressing mechanism, the processor provides an addressability of 2^{20} bytes (1MB) by involving segment registers for address computation in a somewhat complicated manner. This mechanism provides a fragmented look of the memory and is used in 8086 compatibility mode (VM86) or in the real mode of execution.

In the other kind of addressing mechanism, the addresses are 32-bit wide. An operating system uses only one of the two kinds of addressing mechanisms by programming some of the control registers of the processors. The GNU/Linux operating system uses 32-bit wide addressing mechanism. In this addressing mechanism, range of the memory addressability is 2^{32} bytes as 32-bit wide addresses are used.

The operands can be one or more bytes in size. In case an operand is one byte, one memory location is used. However, when the operand is more than one byte in size, multiple memory locations are used to store such operands. For example, a 16-bit operand is stored in two consecutive memory locations. Similarly a 32-bit operand is stored in four consecutive memory locations. The memory operands in the instruction are specified using two attributes – the mechanism on how to compute the start address and size. The natural question that arises is about the storage order of the individual bytes in case of multiple byte operands. A 16-bit operand has two bytes, the most significant byte and the least significant byte. If the address of the operand in

the memory is denoted by a, then two memory locations, a and $a + 1$, are used to store the operand. The IA32 processors use little-endian storage model of the memory. In this model, the least significant byte is stored in the lower address.

For example, if the 16-bit number 0x1245 is to be stored in memory location 0x23C8, the byte 0x45 will be stored in location 0x23C8 while the byte 0x12 will be stored in location 0x23C9 (figure 2.3).

Similarly 32-bit operands are stored in the same order (i.e. least significant byte in the lowest address and the most significant byte in the highest address. An example of the 32-bit operand storage is shown in figure 2.3. In this example, 32-bit number 0x2AA3B33A is stored in memory locations 0x18D0 to 0x18D3.

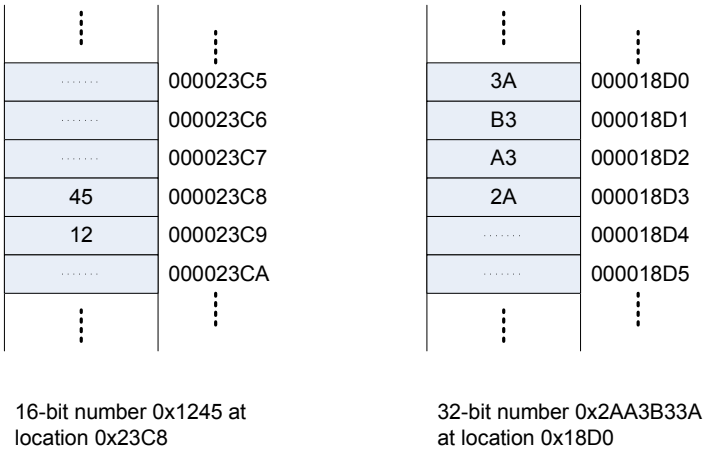


Figure 2.3: Operand Storage Model in IA32 Processors

EXERCISES:

2.4 In a IA32 based system, the bytes stored in memory locations a onward are 0xAA, 0x22, 0x24, 0x5C and 0x6E. What is the word (16-bit number) stored at location $a + 3$?

2.5 What is the value of the variable of type long (32-bit number) stored at location $a + 1$?

2.2 Operand Addressing

Instructions in an IA32 processor operate on operands and generate results. The results are stored in registers or memory locations as specified in the instructions. In some instructions, the result is stored in an implied location. Therefore the IA32 instructions need to specify

no destination (in case it is implied) or at most one destination for the result. Thus instructions in IA32 processors can have 0 or 1 destination operand. Similarly instructions can have zero, one or two source operands. In instructions that require two operands, one of these two also serves as the destination operand. In any case, operands (0, 1 or 2) of an instruction are specified in the instruction.

The mechanism of addressing operands indicates the way operands are found during the execution of an instruction. This mechanism is also known as addressing mode of the processors. The addressing modes are encoded in the instruction. During the execution of the instruction, actual values are taken using the addressing modes specified in the instruction.

There are three categories of operands in an instruction.

- The operand can be specified as a constant in the instruction itself. For example consider an instruction `add $5, %eax`. Execution of this instruction results in adding 5 to register `eax`. The constant 5 is specified in the instruction. All such operands are known as immediate constants and are denoted by a '$' sign in the instruction. Immediate constants can not be the destination of any instruction.

- Operands may be in the registers where register names are specified in the instruction. For example, in order to increment the value stored in register `eax`, instruction `inc %eax` can be used. All registers in the instructions are prefixed by a '%' sign. Registers are commonly used to keep temporary results of the program. Certain registers are used only in a specific manner in IA32 instructions as we shall see later.

- Operands may reside in the memory. The instruction has to then specify the address of the memory and, during execution, the value from the memory is read or written into. IA32 processors provide several ways to specify the address of the memory location as we shall see later.

2.2.1 Immediate Addressing

All constant operands of an instruction are specified using immediate addressing mode. In the GNU/Linux Assembly language, immediate addressing is specified using a `$` sign before the constant. Here are some example of instructions with immediate addressing.

- `sub $5, %eax`. This instruction when executed causes 5 to be subtracted from the contents of register `eax`. There are two operands in this instruction, a constant 5 and register `eax`. The constant 5 is specified using immediate addressing in the instruction.

- `pushl $10`. This instruction when executed results in pushing a 32-bit constant 10 on the stack. It may be observed that constant 10 requires 5 bits for representation (assuming signed number representation). Thus this constant may be stored in one byte, in two bytes or in four bytes. Since the size in bits for the constant can not be determined by just specifying 10, the `push` instruction is suffixed with an '`l`' to denote that 32-bit constant is to be pushed.
- `cmp $10, %eax`. This instruction causes the contents of register `eax` to be compared with a constant 10 and the various flags in the register `eflags` to be set according to the comparison. In this example, constant 10 is implied as 32-bit in size as the other operand of the instruction (register `eax`) is 32-bit in size.

2.2.2 Register Addressing

The register operands of an instruction are specified in the instruction by the names of the registers. All registers in the GNU/Linux Assembly language programs are prefixed by a '%' sign before the name of the register. Here are some examples of the instructions with register addressing.

- `add %eax, %ebx`. The instruction causes the value stored in register `eax` to be added to the value stored in register `ebx`, i.e., `ebx = ebx + eax`. Both operands in this instruction are specified using register addressing.
- `add %al, %al`. The instruction uses two 8-bit register operands both of which are given in register `al`. Since register `al` is same as bits 0 to 7 of the register `eax`, the register `eax` (and therefore register `ax`) also gets modified. This modification happens only in the lower 8 bits or register `eax` (or in the `al` part only).

Register addressing can be used to specify the names of various registers. In 32-bit operations, the registers `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp` and `ebp` can be used. In 16-bit operations, the registers `ax`, `bx`, `cx`, `dx`, `si`, `di`, `sp` and `bp` can be used. Similarly, in 8-bit operations, the registers `al`, `ah`, `bl`, `bh`, `cl`, `ch`, `dl` and `dh` can be used. Some other registers that are rarely used by the programmers are for specific purposes. These include segment registers – `cs`, `ds`, `es`, `fs`, `gs` and `ss`; system control registers – `gdtr`, `ldtr`, `cr0`, `cr1`, `cr2` etc.; various other registers such as MMX registers, floating point registers etc. We provide the details of using these registers at appropriate time later in this book.

Some instructions also use the `eflags` register or its individual bits implicitly. Most instructions also modify the flags as per the result of the operation.

2.2.3 Memory Addressing

When operand of an instruction is stored in memory, it is read from (or written to) the memory during the execution of the instruction using a memory address. Instructions specify the method to compute this memory address, also known as an effective address. Such methods are known as memory addressing modes of the processor. IA32 processors provide very powerful addressing modes to compute effective address.

Direct Addressing

The simplest method of providing an address is to specify it in the instruction. Often symbolic names are used in place of providing the numeric value of the address in the instruction. The assembler then can assign the numeric address to these symbolic names (in reality it is the linker that assigns the final addresses but that is just a small matter of detail). An example instruction that uses the memory addressing is `add 20, %eax` (Notice the missing $ in front of 20, which differentiates this from the immediate addressing mode). In this case, two 32-bit operands are added. The first operand is in memory while the second operand is in register `eax`. Final result is stored back in register `eax`. Address (20) of the first operand is specified in the instruction. In this example, the mechanism used to specify the memory operand is that of direct addressing, i.e., the address of the operand is specified in the instruction itself.

While using memory addressing, the effective address provides just the starting address of the operand in memory. The instruction also requires to know the size of the operand. In most cases, the size of the operand is known implicitly. In this example, the size is known because of the usage of 32-bit wide register `eax`. Thus in reality the memory locations 20, 21, 22 and 23 will be read to get the first operand. As IA32 processors are little-endian, the memory location at address 20 contains the least significant byte of the memory operand.

Another example of the direct addressing is `incb 200`. In this example, the value stored at memory location 200 is incremented by 1. The operand is specified using direct addressing, or by providing the address in the instruction itself. The size of the operand (byte, word or long) is not known implicitly. It is therefore specified in the instruction by suffixing `b` to `inc`. Thus only an 8-bit operand stored in location 200 is incremented by 1.

In the most commonly used scenarios, symbolic names are used to specify the addresses instead of the numbers. An example instruction of such usage is `add NUM1, %eax`, which uses direct addressing if `NUM1` is declared as a variable. the assembler while converting the programs to machine insturctions, converts these symbolic names to

the corresponding addresses in memory.

Specifying an Effective Address

In general the effective address can be provided by specifying four different components (figure 2.4). Two of these components can be registers while two other components can only be constants. The registers are used as the `base` and `index` components while computing the effective address. One of the two constants is known as the `scale` while the other constant is known as the `displacement`. The `scale` can have only four possible values, 1, 2, 4 or 8. The `displacement` can have any value and it can be specified in 8 bits or in 32 bits. Up to three of the four components can be omitted while specifying the effective address. Direct addressing is an example where `base`, `index` and `scale` components are omitted and 32-bit `displacement` is used. Figure 2.4 shows the general use of memory operand addressing.

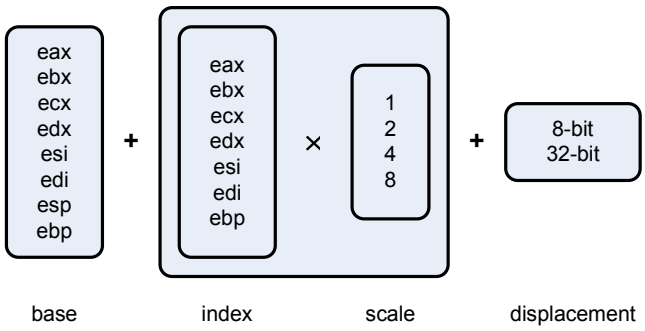


Figure 2.4: Effective Address computation in IA32 Processors

There are a few points to be mentioned. Any of the eight general purpose registers can be used as `base` component. For the purpose of the `index` component, all general purpose registers except the register `esp` can be used. The `displacement` is treated as a signed number. For example, in case the `displacement` is specified in eight bits, range of the `displacement` can be -128 to $+127$.

In GNU/Linux Assembly language, the memory operand for an instruction is specified with the following generalized syntax.

```
displacement(base,index,scale)
```

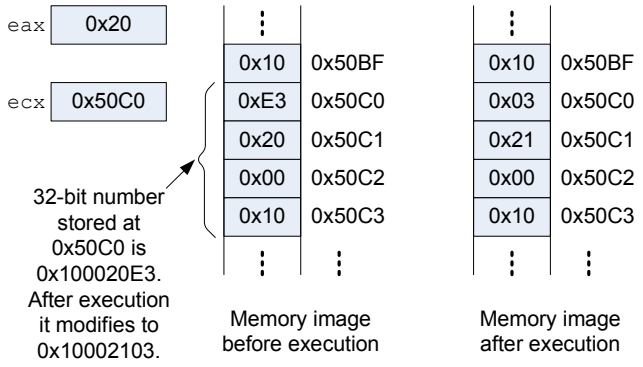
In this syntax, some components may be omitted while specifying the memory operand. The `base` and `index` components can only be specified using a register (figure 2.4). Using combinations of these components, several addressing modes are defined as described below.

Direct. In this addressing mode, `displacement` component alone is used to provide the direct address. We have already seen examples of this mode earlier.

Base or register indirect. If the memory operand is specified using just the `base` component (which can only be a register), the effective address of the memory operand is computed by reading that register. For example, in the instructions given below, the memory operand is specified using only the `base` component.

`mov (%eax), %ebx`	(32-bit memory operand whose address is in register `eax`.)
`add %eax, (%ecx)`	(32-bit memory operand whose address is in register `ecx`).

As an example, consider the case when registers `eax` and `ecx` have values 0x20 and 0x50C0. Execution of instruction `add %eax, (%ecx)` causes a 32-bit number from memory locations 0x50C0 to 0x50C3 to be read, a value 0x20 to be added to that and then stored back in memory locations 0x50C0 to 0x50C3. The net effect is shown in figure 2.5.



Effect of instruction `add %eax, (%ecx)`

Figure 2.5: Example of register indirect addressing

Using just the `base` component to access a memory operand is also called register indirect addressing mode.

Base+Displacement. If the memory operand is specified using `base` and `displacement` components, the effective address is obtained by adding `displacement` to the contents of the `base` register. This kind of addressing can be used in several ways. Let us consider an operand of

an instruction specified using `10(%ebp)`. If `displacement` component in this instruction is given as `10` while the `base` component is specified in register `ebp`. If the value in register `ebp` is 0x3020 then the effective address computed this memory operand is 0x3030. Depending upon the size of he operand, those many bytes will be read or written in the memory starting from the address 0x3030.

- In many programs generated by compiling a program written in high level language such as C, local variables are accessed using an offset in the current frame of local variables. These frames are allocated on the stack each time the function is called and deallocated when the function finishes its execution. Thus to access the local variables, offset can be represented using `displacement` component and the base address of the frame can be represented using the `base` component (which can only be a register). In codes generated by GNU C compilers, register `ebp` is used for the frame base address.

- This mode can also be used to access the field of a structure. The starting address of the structure can be kept in a `base` register and individual field of the structure can be accessed using the `displacement`.

- In order to access an individual component of a global array, say `A[i]`, this mode can be used. In this example, the starting address of the array, `A`, is a constant and can be represented using the `displacement` component. The index `i` can be stored in a register and that register may be used as the `base` component in the operand specification.

Some examples of instructions using this addressing mode are given below.

- `mov A(%esi), %al`. Move (copy) one byte from memory to register `al`. This is an example of the operand addressing case when `A` is an array of variables, each one byte in size, and the index is stored in register `esi`.

- `mov -10(%ebp), %eax`. This instruction is used to access a 32-bit local variable in a C function. Register `ebp` contains the starting address of the frame containing the local variable.

- `mov %ecx, 10(%ebx)`. The instruction can be used to change the value of a field of a record in the memory whose starting address is given in register `ebx`. The offset of the field is 10 and the length of the field is 32 bits since a 32-bit register `ecx` is being used to change the value.

Index*Scale+Displacement. If the memory operand is specified using `index`, `scale` and `displacement` components, effective address is computed by multiplying the `index` by `scale` and then adding the `displacement`. In IA32 processors the value of the `scale` can only be 1, 2, 4 or 8. As mentioned earlier, all general purpose registers except the register `esp` can be used for the `index`. This is a very powerful addressing mode and can be used to access array elements in the following way.

If an array is used where the size of each element is 1, 2, 4 or 8 bytes, then an element of the array can be accessed by this mode. The use of this mode is best explained with an example. Let's consider that an array `A` is used where each element is 16-bits, or 2 bytes wide. If the starting address of the array `A` is 0x1000 and the array has just four elements, then the size of the entire array will be 8 bytes stored in the memory locations 0x1000 to 0x1007. The element `A[0]` is then stored in memory locations 0x1000 and 0x1001. In general an element `A[i]` is stored in memory locations 0x1000+2*`i` and 0x1000+2*`i`+1. This addressing mode can be used to access element `A[i]` by giving starting address of the array as `displacement` (in this case 0x1000); register to store the value of index `i` and the size of each element (2 in this case) as the scaling factor.

Some examples of instructions using this operand addressing mode are given below.

- `add A(,%ecx,4), %edx`. The 32-bit value stored in the memory with starting address `A+4*ecx` is added to register `edx`. This addressing mode is being used here to access an element of an array `A` (array of 32-bit integers) whose index is stored in register `ecx`.

- `mov A(,%ecx), %cl`. The 8-bit number is read from the memory location `A+ecx` and put in the register `cl`. In this case, the `scale` is not specified and it is taken as 1 by the GNU assembler.

- `mov (,%eax), %eax`. The 32-bit number is read from the memory location whose address is given in the register `eax` and the value is then stored in the register `eax` itself. The `displacement` is not specified in this example and it is taken as 0 by default by the GNU assembler. Similarly the `scale` is taken as 1 by default.

  The effect of the addressing mode in this example is same as that of the register direct (or just the `base`) addressing. However, two modes are entirely different. In this case, `displacement` and `scale` are taken as 0 and 1 respectively. Thus two instructions `mov (%eax), %eax` and `mov(,%eax), %eax` have different machine bit-coding even though the effect of the execution is the same.

Base+Index*Scale. This is another very powerful addressing mode provided by the IA32 processors. In this addressing mode, three components (`base`, `index` and `scale`) are provided and the effective address is computed by the processor while executing the instruction. Two components, `base` and `index`, are provided in the registers. The third component `scale` is a constant specified in the instruction and can have only one of the four possible values, 1, 2, 4 and 8. This addressing mechanism can be used to access an array in a very powerful manner. In order to access an element of an array the `base` component can be used to provide the starting address of the array. The `scale` and `index` components can be used to provide the size of each array element and the index of the array element.

An example use of this addressing mode is given in the instruction "`incl (%ebx,%esi,4)`". This instruction is used to add one to the 32-bit operand stored in the memory. The size of the operand (32-bits) is known from the suffix '`l`' in the instruction `inc`. The address of the memory is computed by multiplying the contents of register `esi` and adding to that the contents of register `ebx`. In GNU/Linux Assembly language program, the `scale` factor may be omitted in which case, it defaults to one. Thus the following methods of specifying the memory operands are all equivalent.

`(%ebx,%esi)`, `(%ebx,%esi,)`, `(%ebx,%esi,1)`.

This addressing mode may be used in many cases. For example, elements of a local array allocated on stack can be accessed using this mode. In such scenarios, the `scale` component can be the size of each element of the array, `base` component can be the base address of the stack frame and `index` component can be the offset of the array within the frame.

Base+Index*Scale+Displacement. This mode of addressing in IA32 processors is the most powerful mode. In this mode, all four components are specified and the effective address is computed as an expression where two additions and one multiplication (or shift) are involved. This addressing mode involves two registers and two constants. The registers store part of the address information for the variable and the constants are used for `displacement` and scaling factor. This addressing mechanism can be used in a number of ways.

- This mode can be used to address an element from a two dimensional array. Let us say that we need to access an element of a two dimensional array `A` (`A[i][j]`). The size of the array is $\mathtt{m} \times \mathtt{n}$. The array is stored in a row major format in which the array elements are stored in the memory in the row-first order. Thus the order of storage in the memory is `A[0][0]`, `A[0][1]`, `A[0][2]`, $\cdots$,

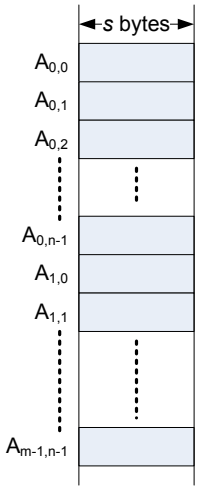


Figure 2.6: Row-major memory allocation of an array

`A[0][n-1]`, `A[1][0]`, $\cdots$, `A[m-1][n-1]`, as shown in figure 2.6. If we denote the address of element `A[0][0]` by `A`, then the address of element `A[0][1]` will be $\texttt{A}+s$ where s is the size of each element in bytes. The address of element `A[0][j]` will be $\texttt{A}+\texttt{j}*s$. Thus the address of element `A[0][n-1]` will be $\texttt{A}+(\texttt{n}-1)*s$. Each row of the array occupies $\texttt{n}*s$ bytes in memory. The address of the first element in row 1 (`A[1][0]`) will be $\texttt{A}+\texttt{n}*s$. The address of the first element in row `i` will be $\texttt{A}+\texttt{i}*\texttt{n}*s$. In a generalized form, the address of element `A[i][j]` is given by the following expression.

$$\texttt{A}+\texttt{i}*\texttt{n}*s+\texttt{j}*s$$

Total amount of memory needed for this array would be $\texttt{m}*\texttt{n}*s$ bytes.

In order to access an element of the array using this addressing mode, one register may be used to keep the expression $\texttt{i}*\texttt{n}*s$. This expression provides the offset of the starting address of row `i` and may be used as `base` component. Another register may be used to store `index` component as `j` while s may be represented by the `scale` component. The starting address of the array (`A`) can be given as `displacement` component.

- This mode is very effective if one needs to access an element of an array within a structure. Base address of the structure can be given in `base` register. The `displacement` component may be used for specifying the offset of the first element of the array with respect to the base address of the structure. The `scale` can

represent the size of each element of the array and `index` register can keep the index of the array.

- This mode can also be used to address local array elements in high level language functions. The base address of the frame can be stored in the `base` register. Offset of the first element of the array with respect to the frame can be represented using `displacement` component of the addressing mode. The size of each element, if it is 1, 2, 4 or 8, can be represented in the `scale` component and the index of the array element can be stored in a register used as `index` component of the addressing mode.

Some example usages of this addressing mode are given below.

- `add A(%ebx,%esi,4), %eax`. The 32-bit value stored in memory is added to the register `eax`. If `A` is a two dimensional array of long (32-bit numbers), then this addressing mode refers to the element `A[i][j]` where `j` is stored in register `esi` and the offset of the first element in row `i` (*i.e.* `i*n*s`) is stored in register `ebx`.
- `movl $0,str(%ebx,%esi,4)`. The 32-bit number 0 is stored in the specified memory location. If `str` is the starting address of a structure and register `ebx` contains the offset of the first element of an integer (32-bits) array within the structure, then this memory operand refers to the element of the array whose index is given in register `esi`.

In GNU/Linux Assembly language, the `scale` component need not be specified and in such a case, a default value of 1 is assumed. Thus the following memory operands are all equivalent in the Assembly language and refer to the same addressing mode.

`A(%esi,%edi,1)`, `A(%esi,%edi)`, `A(%esi,%edi,)`.

EXERCISES:

2.6 Identify the addressing mode of each of the operands in the following instructions. In case the operand is stored in the memory also state the size of the operand.

(a) `addb $3, (%esi)`
(b) `add %eax, (%ebx,%edi)`
(c) `shrl $1, 20(%eax,%ebx)`
(d) `and (,%esi,4), %eax`
(e) `add 100, %al`

2.7 The contents of some of the registers before execution of each of the following instructions are given as `eax=0xFFFFFFFF`, `ebx=0x00000010`, `esi=0x00002000`. The operands stored in the memory can all be assumed to have a value 0 in the beginning of each instruction. After the execution of each of the following instruction, give the contents of the register/memory that gets modified. In case of memory operands, also give the size and address of the operand.

(a) `movl $0,0x1000(%esi,%ebx,4)`
(b) `mov (%esi), %eax`
(c) `incb (%eax,%esi)`
(d) `add %ax, %bx`
(e) `add 0x1000(%esi,%ebx,), %eax`

Chapter 3

Basic data manipulation

In this chapter we shall look at a few IA32 processor instructions and their usage in writing programs. IA32 processor instructions can be categorized into several categories. Many of these instructions are used to manipulate data stored in registers or memory. IA32 processors provide an extremely powerful set of instructions to copy (move) data from one place to another.

3.1 Data movement instructions

Perhaps the most commonly used instruction in the IA32 processors is a `mov` instruction. The `mov` instruction can be used to perform several functions.

- Initialization of register
- Initialization of memory locations
- Copying a value from register/memory to register
- Copying a value from register to register/memory

The syntax of the `mov` instruction is as follows.

```
mov src, dest
```

During the execution of this instruction the value of the `src` operand is copied to the `dest` operand. The `src` operand may be an immediate constant in which case, it causes the initialization of the `dest` operand.

IA32 processors also have a restriction on the type of operands. No instruction in IA32 processors can have more than one memory operand. Therefore, the `mov` instruction can not be used to copy value

of a memory variable to another memory variable. In order to perform such an operation, more than one `mov` instruction will be necessary.

The following is an example of exchanging the value of two 32-bit variables stored in the memory at locations 0x100 and 0x200.

```
1 exchange:
2     mov 0x100, %eax
3     mov 0x200, %ebx
4     mov %ebx, 0x100
5     mov %eax, 0x200
```

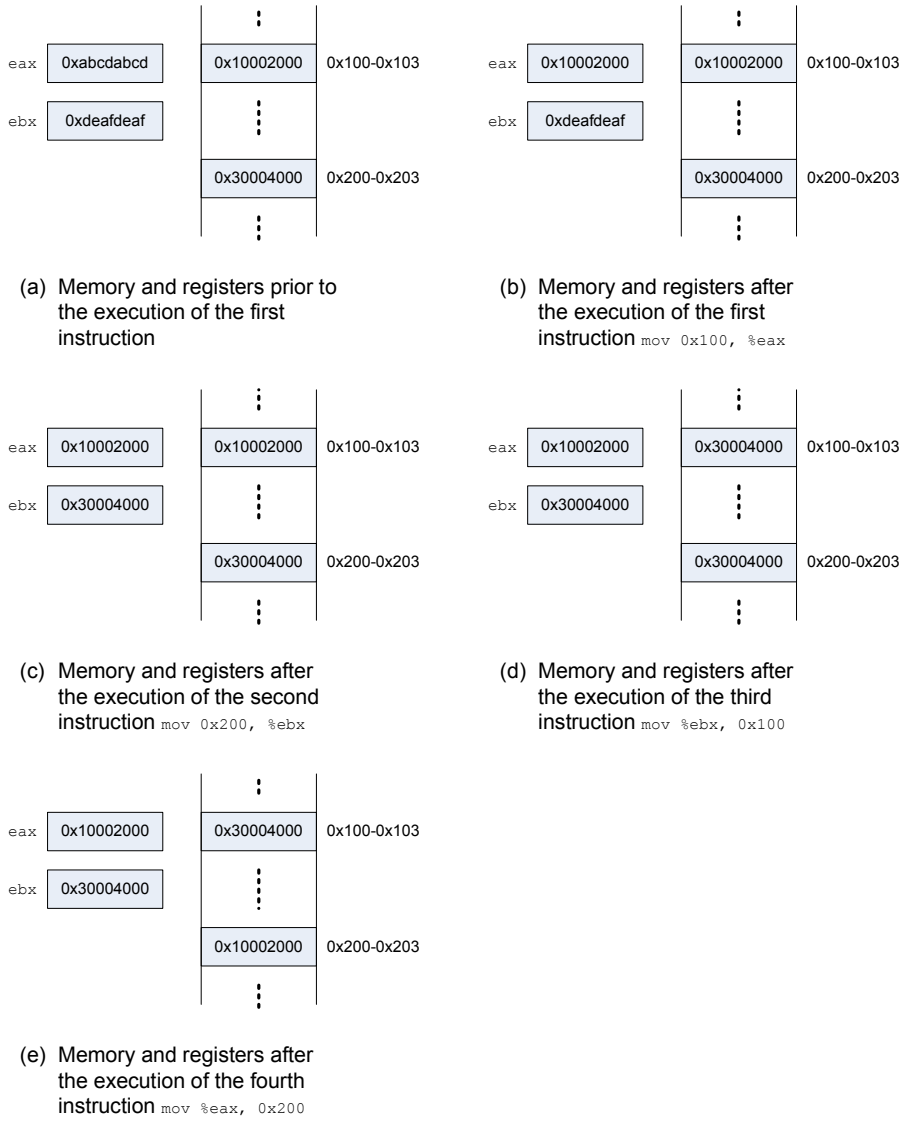
In all the instructions in this example, one operand is specified using the memory addressing while the other operand is specified using the register. The behavior of this program is shown in figure 3.1. While executing the first instruction (line 2), four bytes are read from locations 0x100, 0x101, 0x102 and 0x103 and the corresponding 32-bit number is stored in register `eax` (figure 3.1b). Similarly the second instruction (line 3) causes a 32-bit number to be read from memory locations 0x200 to 0x203 and stored in the register `ebx` (figure 3.1c). The value in register `ebx` is then stored in memory locations 0x100 to 0x103 after execution of instruction in line 4 (figure 3.1d). The exchange of the two values is completed after the fourth instruction (line 5) which updates the value stored in memory locations 0x200 to 0x203 (figure 3.1e). In this example, it can be noticed that the initial contents of registers `eax` and `ebx` (figure 3.1a) are modified at the end.

Another example of a code fragment is given below which causes an integer array `A` of 4 elements to be initialized. Each element `A[i]` is initialized to the value `i` (`i` being between 0 and 3).

```
1 init:
2     mov $0, %esi
3     mov %esi, A(,%esi,4)
4     inc %esi
5     mov %esi, A(,%esi,4)
6     inc %esi
7     mov %esi, A(,%esi,4)
8     inc %esi
9     mov %esi, A(,%esi,4)
```

In this program, register `esi` is used to store index `i` and is initialized to zero in the beginning. Later, this value is stored in the array element using base+index*scale addressing. Since each element of the array is four byte in size, the scale factor is 4. Register `esi` is used to keep the value of `i` between 0 to 3.

The program works in the following manner. The starting address of the array is represented by a symbol `A`. Prior to the execution of



(a) Memory and registers prior to the execution of the first instruction

(b) Memory and registers after the execution of the first instruction `mov 0x100, %eax`

(c) Memory and registers after the execution of the second instruction `mov 0x200, %ebx`

(d) Memory and registers after the execution of the third instruction `mov %ebx, 0x100`

(e) Memory and registers after the execution of the fourth instruction `mov %eax, 0x200`

Figure 3.1: Behavior of example program to exchange two numbers

instruction at line 3, the value of register `esi` is 0. Therefore, the effective address of memory operand in this instruction would be `A`$+0*4$, or `A`. This is also the address of the first element of the array. This location gets a value of 0 after the execution of this instruction. The value of register `esi` is modified to 1 due to the execution of instruction at line 4. Thus the effective address of the memory operand for the instruction at line 5 becomes `A` $+ 1 * 4$, or `A` $+ 4$. This is also the address of the second element of the array which is then initialized to 32-bit number 1 after the execution of instruction at line 5. In a similar manner instructions at line 7 and line 9 modify the third and the fourth elements of the array to values 2 and 3 respectively.

The first instruction uses immediate addressing for the first operand and register addressing for the second operand. Remaining `mov` instructions all use register for the first operand and memory for the second argument. The `inc` instruction requires only one operand. In this program operands in all occurrences of `inc` instructions are specified using register addressing.

IA32 processors also have an interesting instruction that exchanges the values of two operands.

```
xchg src, dest
```

However, none of the two operands in this instruction can be specified using immediate addressing. Further, like in other instructions, only one of the two operands may be a memory operand. The code to exchange two memory variables given earlier has a problem–the values stored in registers `eax` and `ebx` prior to the execution get modified after the execution of the code is completed. A code fraction that does not modify any of the registers and yet exchanges the two memory variables is given below.

```
exchange:
    xchg 0x100, %eax
    xchg 0x200, %eax
    xchg 0x100, %eax
```

The program behavior is explained with an example. Let's assume that the 32-bit values stored in register `eax`, in memory at location 0x100 and in memory at location 0x200 are 0x11223344, 0x55667788 and 0x99AABBCC respectively, as shown in figure 3.2a.

After execution of the first instruction, the values stored in register `eax` and memory at address 0x100 get exchanged. Thus the old value in register `eax` gets preserved in memory at address 0x100 onwards and the old value in memory gets copied to register `eax` (as shown in figure 3.2b). Since the IA32 processors are little-endian, the layout of

the memory is shown with the least significant byte stored in the memory location with smaller address. After the execution of the second instruction (figure 3.2c), memory values at location 0x200 and register `eax` get exchanged. Thus the original 32-bit value stored in memory location at 0x100 (*i.e.* 0x55667788) moves to the memory location at 0x200 and the register `eax` gets the value originally stored at location 0x200. The last instruction exchanges the values again (figure 3.2d) and therefore register `eax` gets its original value (*i.e.* 0x11223344) and memory location 0x100 gets the value 0x99AABBCC. The net effect of this code fragment is to exchange the values stored in memory locations 0x100 and 0x200 without modifying any register (including `eax`, which is modified during the execution but gets back its original value at the end).

IA32 processors also provide an instruction to change the endianity of a 32-bit number. IA32 processors implement little-endian storage for multi-byte data elements in the memory. In this mode, the least significant byte is stored in the lowest address, and the most significant byte is stored in the highest address of a memory variable. For example in figure 3.2a, the number stored in memory location 0x100 is 0x55667788. The least significant byte of this number (0x88) is stored at memory location 0x100. The next two bytes 0x77 and 0x66 are stored at memory locations 0x101 and 0x102 respectively. The most significant byte 0x55 is stored in the last memory location 0x103. In big-endian storage, the bytes are stored in the reverse order in memory. Thus the most significant byte is stored at the lowest address while the least significant byte is stored at the highest address. IA32 processors provide a `bswap` instruction that swaps the bytes of a 32-bit register to change the order of the storage. The syntax of the `bswap` instruction is as follows.

```
bswap r32
```

The `bswap` instruction takes only a single 32-bit operand specified using the register addressing. The operand can not be specified using immediate or memory addressing modes. The `bswap` instruction along with the `mov` instruction can be used to change the byte order of the memory variables. Such an operation is often needed when the data is provided from other processors or resources external to the processor. For example, in a computer network, the data may arrive in big-endian order while the IA32 processor may need to operate on the data in little-endian order. The following code fragment shows a mechanism to change the byte order of a memory variable whose address is provided in register `ebx`.

```
swapbytes:
    xchg (%ebx), %eax
```

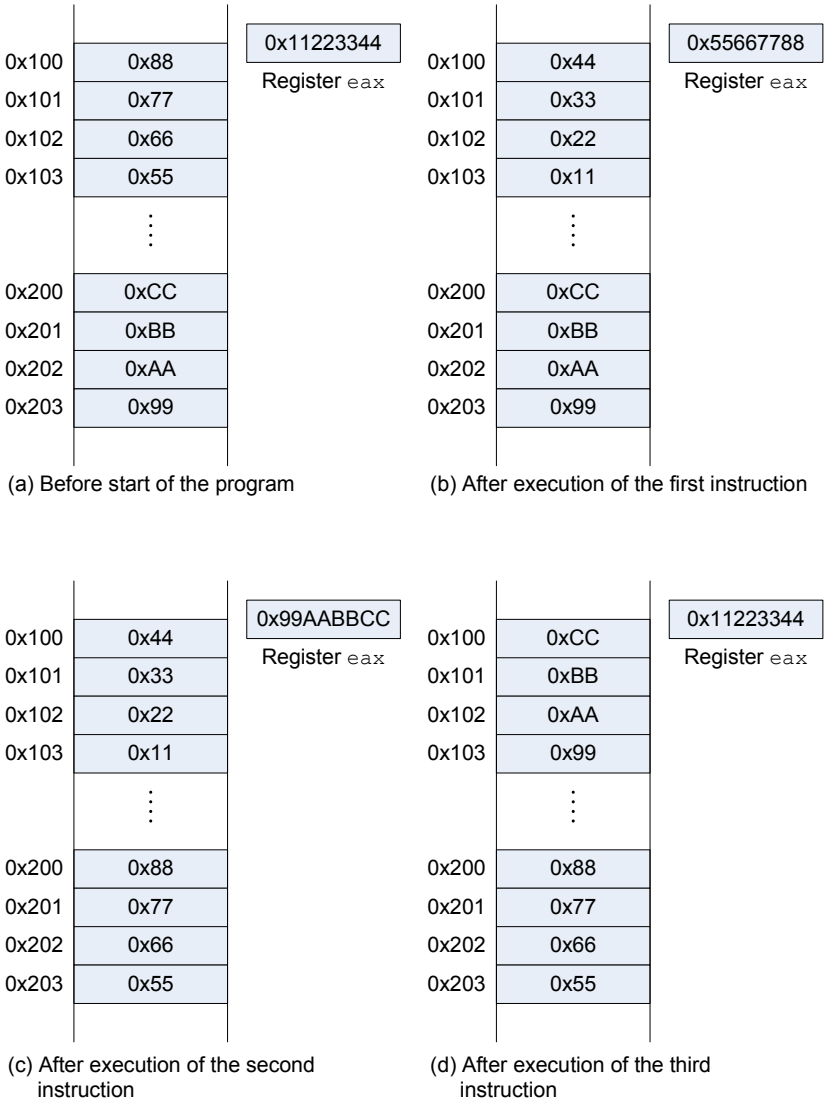


Figure 3.2: Layout of memory and register `eax`

```
bswap %eax
xchg (%ebx), %eax
```

This code works as follows. The first instruction copies the 32-bit data for changing the byte order into register `eax`. The original value of the register `eax` is preserved in that place which is then restored after the execution of the third instruction. The second instruction changes the byte order of data previously loaded in register `eax`. Thus this code is written in a way that the contents of registers `eax` and `ebx` after the execution retain the same value as it was before the execution.

EXERCISES:

3.1 None of the operands of the `xchg` instruction can be an immediate constant. Why?

3.2 Write a code fragment for changing the endian-ness of a 16-bit number stored in register `ax`. Note that the `bswap` instruction can not be used in this case as this instruction can only operate on a 32-bit register.

3.3 Write a code fragment that increments a 32-bit number stored in the memory. The memory variable is stored in the big-endian order and the order should not be changed for this variable.

3.2 Condition flags and their use

The `eflags` register in an IA32 processor stores various flags corresponding to the result of the last instruction executed. Some instructions however do not modify the flags. For example `bswap`, `xchg` and `mov` instructions do not change any flags. The `inc` instruction however does change a few flags of the `eflags` register.

3.2.1 Flags

Carry flag (CF)

The carry flag keeps the status of the final carry out while computing result for the last instruction that modifies the carry flag. An example of such an instruction is `add` instruction. While adding two numbers, the carry flag contains the carry out of the most significant bit. For example, consider an instruction `add $4, %al`. If the initial value in register `al` is 0x3F, then after execution of this instruction value in register `al` will be 0x43 and carry flag will be 0. However, if initial value of register `al` is 0xFD, the final value of register `al` will be 0x01 and carry flag will be set to 1. Carry flag therefore also represents the overflow condition assuming the operands to be unsigned numbers. For instructions that implement subtraction, the carry flag keeps the 'borrow' condition into the most significant bit.

Zero flag (ZF)

The zero flag is set to 1 if the result of the last flag-modifying instruction is all 0. Zero flag is set to 0 if the result of the last flag-modifying instruction is not 0. The value of zero flag remains unchanged upon execution of instructions that do not modify the flags. For example, if register `al` contains 0xFF initially, then `inc %al` instruction causes register `al` to get a value that is all 0 and zero flag gets set to 1. In this example, the carry flag also gets set to 1. If after subtraction of one number from another the ZF becomes 1, it reflects that two numbers were equal.

Sign flag (SF)

The sign flag stores sign of the result of last flag-modifying instruction. This flag is used for implementing arithmetic using 2's complement representation for the numbers. In 2's complement representation, the most significant bit of the number is 1 if the number is negative. Similarly, positive numbers are represented with the most significant bit set to 0. The sign flag is set to 1 if result of the last flag-modifying instruction was a negative number. The sign flag is set to 0 if the result of the last flag-modifying instruction was a non-negative number.

Parity flag (PF)

The parity flag stores the parity status of the least significant byte of the result of the last flag-modifying instruction. This flag gets set to 1 if the least significant byte of the result has even number of 1's and is set to 0 if that number is odd. For example, if register `eax` contains an initial value of 0x000000FF and 1 is added to the register, the final contents of the register will be 0x00000100. The least significant byte of the result is 0x00. None of the bits of this byte is set to 1 and therefore the number of ones in the least significant byte is even (0). The parity flag in this case will be set to 1.

Overflow flag (OF)

The overflow flag stores the overflow condition of the result of the last flag-modifying instruction. While sensing the overflow, signed arithmetic is taken into consideration. For example, in case of addition of two positive numbers, if the result turns out to be negative, overflow flag gets set to 1. Similarly, in case of addition of two negative numbers, if the result turns out to be positive, the overflow flag gets set to 1. If the result of the last flag-modifying instruction does not overflow, this flag gets set to 0.

Auxiliary carry flag (AF)

This flag contains carry out from bit 3 of the result of a flag-modifying instruction. The flag is primarily used for implementing arithmetic using binary coded decimal (BCD) number representation. The flag is of little value in most programs and is not discussed in much detail in this book.

Other flags

There are several flags in the `eflags` register which do not indicate the status of computational results. These flags are modified by certain explicit instructions provided in IA32 instruction set for this purpose. We shall see many of these instructions and flags in the coming chapters.

3.2.2 Conditions

Several instructions have a conditional execution in IA32 processors. The instructions behave in one way if the condition is true and in another way if the condition is false. The conditions are determined by the contents of the CPU flags. Many conditions combine various flags in certain ways. There are a distinct 16 number of conditions supported by the IA32 processors by combining various flags. Various conditions implemented in the IA32 processors are the following.

EQUAL (e) or ZERO (z)

This condition is true if ZF (zero flag) is set to 1. The one of the common use of this condition is to test if two numbers are equal or not. For doing so, one number may be subtracted from the other and if two numbers were equal, it would result in setting ZF to 1.

NOT EQUAL (ne) or NOT ZERO (nz)

This condition is opposite of the previous condition. The condition is true if the ZF is set to 0.

ABOVE (a) or NEITHER BELOW NOR EQUAL (nbe)

This condition is primarily used for the comparison of unsigned numbers. One number is considered above the other number if the subtraction of the second number from the first neither results in a borrow nor the result is zero. Thus this condition is true only if CF=0 and ZF=0. If any of the two flags is 1, the condition is evaluated to false.

BELOW OR EQUAL **(be) or** NOT ABOVE **(na)**

This condition is the inverse of the last condition and refers to the case when CF=1 or ZF=1. Thus if any of the two flags is 1, the condition is evaluated to true.

ABOVE OR EQUAL **(ae) or** NOT BELOW **(nb) or** NO CARRY **(nc)**

This condition is also used in connection with the comparison of the unsigned numbers and evaluates to true only if CF=0. For unsigned numbers, the 'above or equal to' condition is true when the subtraction of a number from another does not result in any carry.

BELOW **(b) or** NEITHER ABOVE NOR EQUAL **(nae) or** CARRY **(c)**

This condition is the inverse of the previous condition and refers to the case when CF=1.

GREATER **(g) or** NEITHER LESS NOR EQUAL **(nle)**

This condition is used in implementing decisions based on signed arithmetic and represents the case when the comparison of two numbers (*i.e.* subtraction of one number from the other) results in a number that is positive (when no overflow) and not zero. The condition refers to the case of ZF=0 and SF=OF.

LESS OR EQUAL **(le) or** NOT GREATER **(ng)**

This condition is the inverse of the last condition and refers to the case when ZF=1 or SF<>OF.

GREATER OR EQUAL **(ge) or** NOT LESS **(nl)**

This condition is also used for the signed number comparison and represents the case when SF=OF.

LESS **(l) or** NEITHER GREATER NOR EQUAL **(nge)**

This condition is the inverse of the last condition and refers to the case when SF<>OF.

OVERFLOW **(o)**

This condition refers to the case when OF=1. The condition is used primarily for the signed arithmetic.

NOT OVERFLOW **(no)**

This condition is the inverse of the last condition and refers to the case when OF=0.

SIGN **(s)**

This condition refers to the case when result of the last flag-modifying instruction was negative. The condition is evaluated to true if SF=1.

NO SIGN **(ns)**

This condition is the inverse of the last condition and is true if SF=0, or the result of the last flag-modifying instruction was positive.

PARITY **(p) or** PARITY EVEN **(pe)**

In IA32 processors, the parity flag, PF stores the even parity status of the result of the last flag-modifying instruction. Thus these two cases are equivalent and refer to the case when PF=1.

NO PARITY **(np) or** PARITY ODD **(po)**

This condition is the inverse of the last condition and evaluates to true when PF=0.

3.3 More data movement instructions

IA32 processors also support conditional data movement instructions. The instructions can be used for conditionally moving a 16-bit or 32-bit value from one general purpose register to another. It can also be used for conditionally moving the contents from memory to a 16-bit or 32-bit register. The instructions can not be used for 8-bit data operands, immediate data operands or for moving a value from register to memory. The syntax for the instruction is the following.

```
cmovcc source, dest
```

Here the `source` operand can be a register or a memory (16 or 32 bits in size). The `dest` operand can only be a register 16 or 32 bits in size. The condition is specified using the *cc* field in the instruction. Various variants of the `cmov` instruction are listed in table 3.1.

Use of these instructions is illustrated with an example given below. In this example, we want to set a value in register `eax` according to the value of the variable in memory location `cond`. If the variable is

Instruction	Condition	Flags
`cmove`	Equal	ZF=1
`cmovz`	Zero	ZF=1
`cmovne`	Not Equal	ZF=0
`cmovnz`	Not Zero	ZF=0
`cmova`	Above	CF=0 and ZF=0
`cmovnbe`	Neither Below nor Equal	CF=0 and ZF=0
`cmovbe`	Below or Equal	CF=1 or ZF=1
`cmovna`	Not Above	CF=1 or ZF=1
`cmovae`	Above or Equal	CF=0
`cmovnb`	Not Below	CF=0
`cmovnc`	No Carry	CF=0
`cmovb`	Below	CF=1
`cmovnae`	Neither Above nor Equal	CF=1
`cmovc`	Carry	CF=1
`cmovg`	Greater	ZF=0 and SF=OF
`cmovnle`	Neither Less nor Equal	ZF=0 and SF=OF
`cmovng`	Not Greater	ZF=1 or SF<>OF
`cmovle`	Less or Equal	ZF=1 or SF<>OF
`cmovge`	Greater or Equal	SF=OF
`cmovnl`	Not Less	SF=OF
`cmovnge`	Neither Greater nor Equal	SF<>OF
`cmovl`	Less	SF<>OF
`cmovo`	Overflow	OF=1
`cmovno`	No Overflow	OF=0
`cmovs`	Sign	SF=1
`cmovns`	No Sign	SF=0
`cmovp`	Parity	PF=1
`cmovpe`	Parity Even	PF=1
`cmovnp`	No Parity	PF=0
`cmovpo`	Parity Odd	PF=0

Table 3.1: Conditional Move Instructions in IA32 Processors

between 1 and 5 (both inclusive) then the register is set to 0. Otherwise it is set to -1 (i.e. 0xFFFFFFFF).

```
setcondition:
    mov  $0, %eax
    mov  $-1, %ebx
    cmpl $1, cond
    cmovb %ebx, %eax
    cmpl $5, cond
    cmova %ebx, %eax
```

The first two instructions initialize the registers `eax` and `ebx` to 0 and -1 respectively. The `cmp` instruction is not discussed yet. It is used for comparing (and thereby setting flags) two arguments. The third instruction compares the memory argument with a constant 0 and sets the flags accordingly. A conditional move instruction sets `eax` register to -1 (i.e. the value in register `ebx`) if the result of the comparison indicates that the variable `cond` is below 1. Similarly the last two instructions set the register `eax` to -1 if the memory variable `cond` is above 5.

This program is implemented using unsigned arithmetic. The same program can also be written with signed arithmetic as follows.

```
setcondition:
    mov  $0, %eax
    mov  $-1, %ebx
    cmpl $1, cond
    cmovl %ebx, %eax
    cmpl $5, cond
    cmovg %ebx, %eax
```

EXERCISES:

3.4 Write a code fragment to check that a signed 32-bit integer stored in memory location `num` is within a range or not. The lower and upper limits of the range are given in signed 32-bit integers stored in memory locations `lower` and `upper` respectively. The code fragment should leave a 0 in register `eax` if `num` is within the range (inclusive of the `lower` and `upper`). It should set the register `eax` to -1 in case the number `num` is outside the range.

3.5 Write a code fragment that increments the value of a register `ebx` and if there is an overflow while doing so, sets the register `eax` to -1. The register `eax` is set to 0 if there was no overflow.

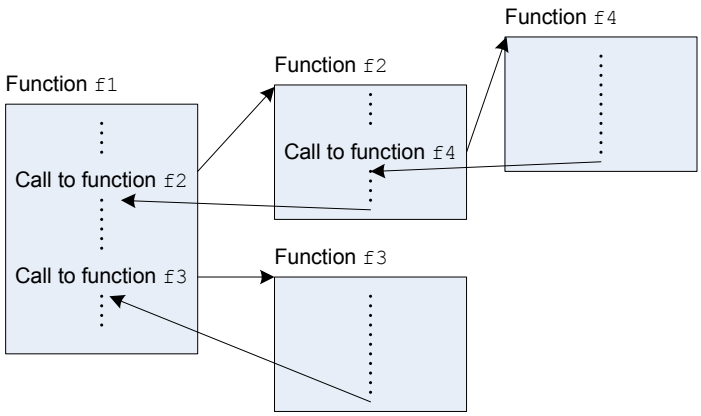


Figure 3.3: A sequence of function calls and returns

3.4 Machine stack

IA32 processors also implement a stack in the memory. The stack is a data structure in which data items are added and removed in the last-in first-out (LIFO) order. Such a data structure is extremely useful in scenarios where the functions are called by the programs. The order of calling the functions and the return back to the calling function is LIFO. Consider the function calling sequence shown in figure 3.3.

In this example, function f1 makes two calls to functions f2 and f3 respectively. Function f2 in turn makes one call to function f4. The execution starts with the first instruction of function f1. The execution continues till a call is made to function f2. At this time, the execution control is transferred to function f2. When the execution of function f2 is completed, the execution of function f1 is resumed from the next instruction. The same semantics are also maintained when function f2 makes a call to function f4. Thus while executing function f4, the execution path can be traced as "from f1" to function f2 and "from f2" to function f4. When function f4 finishes its execution, the control of the execution is given back to the calling function (i.e. function f2) so that execution of function f2 can resume from the location where it had left earlier. Similarly the return from function f2 takes control of execution to function f1. There is a natural order of passing control upon calls to the function and the reverse order of passing control upon return from the functions. A stack is an obvious choice here. Each time a function is called, the return address is pushed on the stack. The top of the stack therefore provides address within the function to which control must return when execution of the called function is over.

IA32 processors provide a mechanism to implement stack and several instructions that use this stack implicitly. The actual storage for

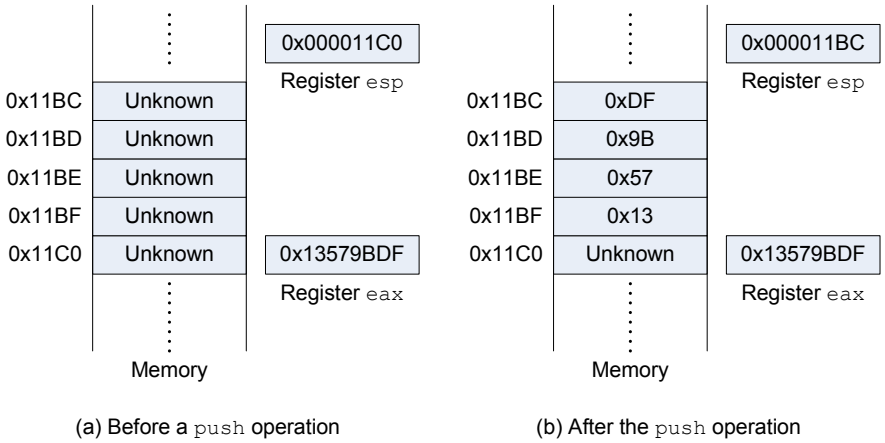


Figure 3.4: Push operation on a stack

the stack is created in the processor memory along with other items such as program instructions, data etc. The top of the stack is represented by register esp, which stores the memory address of the location where the last data item was added. Upon each push, register esp is first decremented by the number of bytes in the data item. The value of data item is then stored in the memory location whose address is available in register esp. For example, if we wish to push contents of register eax on the stack, register esp will be decremented by 4 first (as the size of register eax is four bytes) and then the value of register eax will be stored in memory. As an illustration, consider figure 3.4. The value of register esp is 0x000011C0 prior to the push operation and that of register eax is 0x13579BDF. After the push operation (figure 3.4(b)), the value of register eax remains unchanged while that of register ecx becomes 0x000011BC. The contents of memory locations before and after the push operation are also shown in figure 3.4.

The pop operation is reverse of the push operation. In the case of the pop operation, first appropriate number of bytes are read from memory addressed by register esp and then register esp is incremented by size of the data operand. For example, in order to perform a pop operation for register ecx from the memory, first four bytes will be read from the memory pointed to by register esp and then register esp will be incremented by 4. If such an operation is performed after the example of push operation in figure 3.4, the contents of register ecx will become 0x13579BDF and that of register esp will change to 0x000011C0.

The machine stack in IA32 processors is a very powerful abstraction and can be used in several ways.

- The most important use of the stack is to keep the return addresses for the function calls. We shall see such use of the stack

in the next chapter.

- Another use of the stack is to temporarily save the contents of registers and recover them later.
- Implementations of high level programming languages typically use stack to pass parameters to functions. We shall see such use of stack in the next chapter.
- Local variables of a high level language function are typically alloted on the stack. By doing so, the local variables can be alloted for recursive functions with ease.
- Execution control for recursive functions is implemented using a stack where the functions are returned in a reverse order of the call. The function that is called at the end returns first and the function invocation that is done in the beginning is concluded last.

3.5 Yet more data movement instructions

The stack related data movement instructions in IA32 processors implement push and pop operations on the stack. The syntax of these two instructions is as follows.

```
push src
pop dest
```

The `src` operand in the case of `push` instruction can be an immediate constant, a register or a memory variable. The size of the operand can only be either 16 bits or 32 bits. 8-bit registers or other operands can not be pushed on the stack. If the operand of a `push` instruction is specified using immediate addressing or memory addressing scheme, the size of the operand is not implicitly known. In order to specify the size of such operands, the `push` instruction can be suffixed by a `w` or `l` to indicate 16-bit or 32-bit operations respectively. In general, in the GNU/Linux Assembly language, any instruction can be suffixed by `b`, `w` or `l` to represent that the operands are 8-bit (byte), 16-bit (word) or 32-bit (long) in size respectively. Thus the `push` instruction can be specified as `pushw` or `pushl` whenever the size of the operand can not be determined implicitly.

If the operand is an immediate constant or is a memory variable, '`w`' and '`l`' in `pushw` and `pushl` instructions can be used to provide the size of the operand.

The `dest` operand in case of the `pop` instruction can only be a memory variable or a register. Similar to the `push` instruction, the size of the operand can only be either 16-bits or 32-bits.

Let's consider an example to understand the behavior of these instructions. The following is a small program segment.

```
1 mov   $0, %eax
2 mov   $0, %ebx
3 pushl $0x13487
4 pop   %ax
5 pop   %bx
6 push  %ebx
7 pop   %ecx
```

The instructions in lines 1 and 2 initialize registers `eax` and `ebx` both to 0. Since the operands of these instructions are registers, the size of data for 'move' operation is known implicitly. Hence, putting a suffix after the instruction is not necessary (though putting an appropriate suffix, '`l`', will not be wrong). The instruction in line 3, pushes a value on the stack. Since the size of the immediate constant 0x13487 can not be determined, a suffix '`l`' is used in the `push` instruction in line 3. Thus this instruction pushes a 32 bit number on the stack. Figure 3.5 shows the contents of memory and registers after the execution of each instruction. It is assumed that the contents of register `esp` were 0x00100124 prior to the start of execution of instructions. During the execution of the `pushl` instruction in line 3, first the register `esp` is decremented by 4 and then bytes corresponding to constant 0x13487 are stored in the memory. Instruction at line 4 copies two bytes from the stack to the register `ax` and adjusts the stack pointer. Since register `ax` represents the lower 16 bits of the register `eax`, upper 16 bits of register `eax` remain unchanged. Thus the value of the register `eax` becomes 0x00003487. The stack pointer, or register `esp`, gets modified and contains 0x00100122. After execution of the instruction at line 5, register `ebx` gets a value 0x00000001. Register `esp` returns to the old value, i.e., the value prior to the execution of this program. Instruction at line 6 again pushes the value of the register `ebx` and that value is taken out into the register `ecx` by `pop` instruction in line 7. Thus at the end of the program, contents of registers `eax`, `ebx`, `ecx` and `esp` are 0x3487, 0x1, 0x1, 0x100124 respectively.

In many Assembly language programs, instruction sequence similar to the one given in line 6 and 7, is used to copy value of a register into another. In fact in the example program, instructions at line 6 and 7 can possibly be converted to a simple `mov %ebx, %ecx` instruction. The only difference between the two scenarios is that in the first scenario, memory locations are also modified when using `push` and `pop` instructions while this is not the case when `mov` instruction is used.

EXERCISES:

3.6 The values of the registers (except register `esp`, which has some value

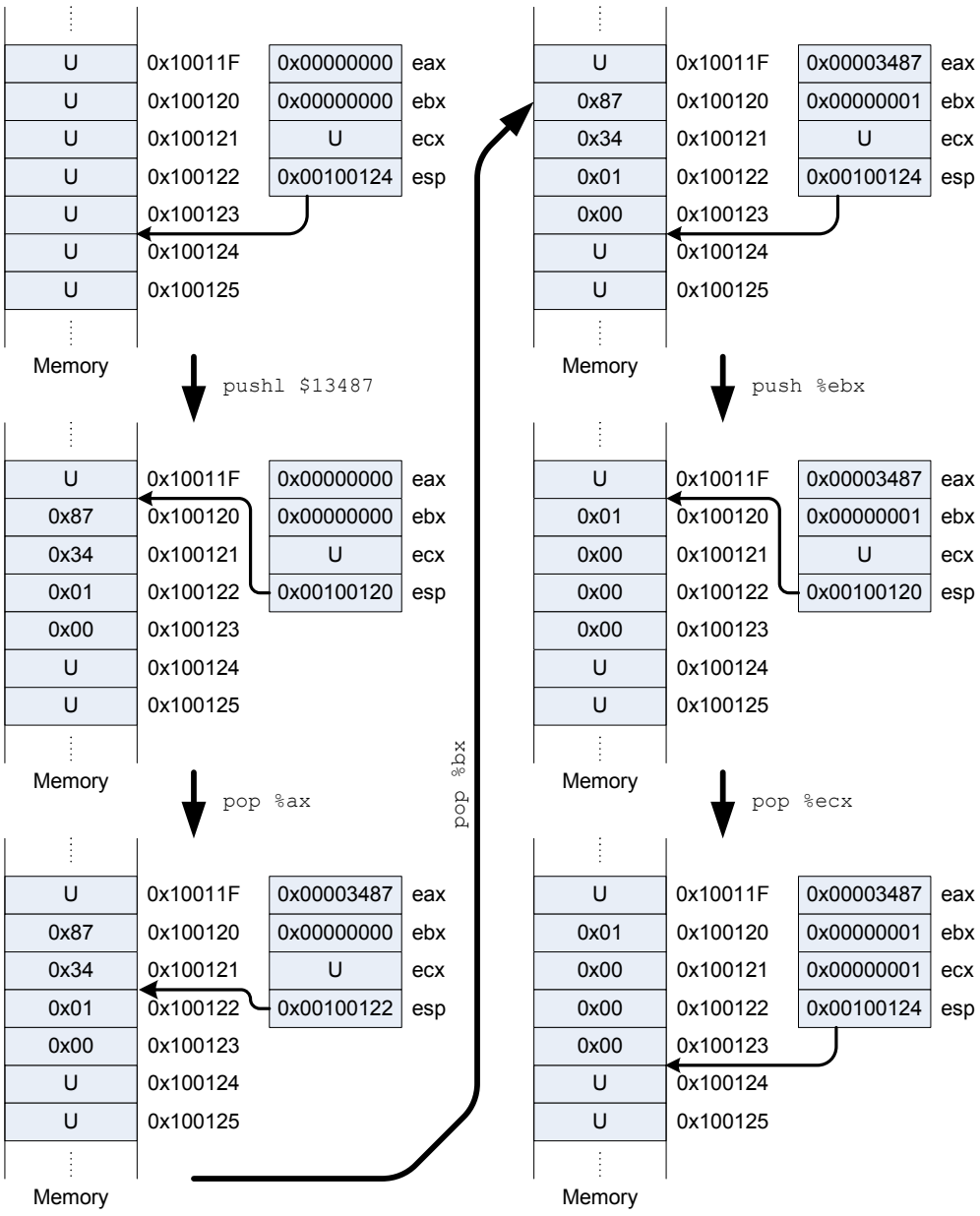


Figure 3.5: Push operation on a stack

denoted by a) are given to be all 0 prior to the execution of the following sequence of instructions.

```
1 pushw $0x487
2 pushw $0x33
3 pushw $0x57C
4 inc %esp
5 pop %eax
6 inc %esp
```

What is the value of the memory locations and registers `esp` and `eax` after the execution of each instruction?

3.7 What is the effect of the following instructions? If changes in the memory locations are to be ignored, which instruction or a sequence of instructions can be used to get the same effect?

```
push %edx
push %ecx
pop %edx
pop %ecx
```

In programs, often several registers are saved on the stack at the same time and later restored. IA32 processors provide another set of push and pop kind of instructions for that purpose. The instruction `pusha` pushes all the eight general purpose registers on the stack. The instruction can be used to push 16-bit registers or 32-bit registers by suffixing `w` and `l` to the instruction respectively. Without any suffix the instruction defaults to pushing 32-bit registers.

The `pusha` or `pushal` instruction on GNU/Linux is equivalent to pushing registers `eax`, `ecx`, `edx`, `ebx`, `esp` (the value prior to the `pusha` instruction execution), `ebp`, `esi` and `edi` in that order. Thus the value of the register `edi` is on the top of the stack after the execution of `pusha` instruction. The `pushaw` instruction behaves in a similar manner except that the 16-bit registers are pushed on the stack rather than 32-bit registers. The order of pushed registers on the stack is `ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si` and `di`. The value of the register `sp` pushed on the stack is the one prior to the start of execution of the `pushw` instruction.

EXERCISES:

3.8 In a program, we need to exchange the values of registers `eax` and `ebx`; registers `ecx` and `edx` and registers `esi` and `edi`. Write a program using the `pusha` instruction to achieve this.

3.9 How is the value of the register `esp` adjusted after a `pusha` instruction? What is the adjustment after a `pushaw` instruction?

The counterpart of the `pusha` instruction is `popa` instruction. The `popa` instruction removes 32-bit registers from the stack in the reverse order of the push. The pop of the `esp` is handled in such a way that the at the end it contains the original value as it was before the execution of the `pusha` instruction assuming that `popa` instruction was executed immediately after. The instructions `popal` and `popaw` are counterpart instructions to `pushal` and `pushaw` instructions respectively.

IA32 architectures also provide two more instructions to push flags on the stack. The instruction `pushf` pushes 32-bit `eflags` register on the stack. The instruction `pushfl` is also equivalent to the `pushf` instruction. Lower 16-bits of the `eflags` register are also known as `flags` register and are pushed on the stack upon execution of `pushfw` instruction. Syntax of these instructions are the following.

```
pushf
pushfl
pushfw
popf
popfl
popfw
```

However execution of `popf` instruction is not exactly the counterpart of the `pushf` instruction. Several flags in the `eflags` register are system flags and modification in their value (by the way of a `popf` instruction) is not permitted. The description of the exact behavior of this is not given in this book and the interested readers may refer to Intel's IA32 architecture documents for more information.

3.6 Handling unsigned and signed numbers

IA32 architectures support both unsigned and signed numbers. For the signed numbers, 2's complement representation is used. In 2's complement representation the most significant bit of an operand is 1 for negative numbers and 0 for positive numbers.

In IA32 architectures several instructions are provided that change the size of the signed numbers. For example, an 8-bit data item can be converted to 16-bit data item by an instruction `cbw` . While converting a data item to a larger size, the sign bit can be duplicated to all the extra bits. Let's consider a byte with value 0x08. The equivalent value in 16-bit is 0x0008 which is obtained by duplicating the sign bit (0) to all extra bits in 16-bit number. If the number originally was negative, the extra byte will have all bits set to 1 (i.e. value will be 0xFF). For example, the 16-bit extension of a byte 0xE2 (2's complement representation of -30) will be 0xFFE2.

The following instructions in IA32 architectures are provided to convert data sizes for the signed numbers.

```
cbtw
cwtd
cwtl
cltd
movsbw src, dest
movsbl src, dest
movswl src, dest
```

The instruction `cbtw` (also known as `cbw`) converts an 8-bit signed number stored in register `al` to a 16-bit signed number and stores it in register `ax`. Instruction `cwtd` (also known as `cwd`) converts a 16-bit signed number stored in register `ax` to a 32-bit number. The most significant 16 bits of the resultant 32-bit numbers are returned in register `dx` while the register `ax` represents the lower 16 bits (and therefore remains unchanged). Instruction `cwtl` (also known as `cwde` instruction) performs the similar operation except that it returns the 32-bit number in register `eax`. Instruction `cltd` (also known as instruction `cdq`) converts a 32-bit number to 64-bit numbers. Input 32-bit number is provided in register `eax` and resultant 64-bit number is made available in registers `edx` and `eax`. The value in register `edx` after the execution contains either 0 (if the number in `eax` was positive) or 0xFFFFFFFF (if the number in `eax` was negative).

The `movs..` instructions are more generic than any of the other instructions. These can be used to convert 8-bit data to 16-bits (`movsbw`), 8-bit data to 32-bits (`movsbl`) and 16-bit data to 32-bits (`movswl`). The `src` operand can be addressed using memory or register addressing only. The `dest` operand can only be a register whose size must match with the one specified in the instruction. An example use of the `movs..` is `movsbl %al, %edx` instruction, which convert an 8-bit signed number in register `al` to a 32-bit signed number in register `edx`.

There are a few more instructions in the IA32 architecture to convert data sizes for the unsigned numbers.

```
movzbw src, dest
movzbl src, dest
movzwl src, dest
```

The `src` can be a register operand or a memory operand. The `dest` can only be a register operand. The `movzbw` instruction converts a byte operand to a word by setting extra bits to 0. The `movzbl` instruction converts a byte operand to a long operand by setting extra 24 bits to 0. The last instruction `movzwl` converts a word operand to a long operand by setting extra 16 bits to 0.

Since in the expansion of the unsigned numbers, the extra bits are always filled with 0, there are several other ways of handling data conversion of unsigned numbers.

For example, the following two instruction sequences achieve the same effect.

```
mov $0, %eax                    movzbl %bl, %eax
mov %bl, %al
```

EXERCISES:

3.10 Write a small code fragment that initializes four registers `eax`, `ebx`, `ecx` and `edx` to a signed value stored in a byte wide memory location `init`.

3.11 Show the contents of the memory and registers, relevant to the following program segment, after the execution of each instruction.

```
mov $0x2300, %esp
mov $0x2480, %ax
movswl %ax, %ebx
pushl $-10
popl %eax
movsbl %al, %ecx
movsbl 0x22FD, %edx
```

3.12 What will be the signed number in register `eax` after execution of the following program segment.

```
mov    $-10, %al
movsbw %al, %ax
movzwl %ax, %eax
xchg   %al, %ah
bswap  %eax
```

Chapter 4

Control Transfer

We have learned in the earlier chapters that the processors execute instructions in a semantically sequential order. After an instruction is executed, the next instruction is fetched from the memory and executed. This model of execution of programs is also known as "stored program model". In most cases, the next instruction is stored in the memory immediately after the current instruction. Thus the order of execution of the instructions is same as the order of the storage in memory. However, such a model is highly restrictive. For example, in order to execute a program loop control should be transferred to beginning of the loop after execution of an iteration. All processors, therefore, support instructions which transfer control to a location other than the next one. The next instruction to be executed in such cases is the one specified in the instruction and not the one stored next in memory. IA32 processors support several instructions for control transfer. In this chapter, we shall see these instructions.

This is also an appropriate place to learn the issues with linking. Programs written in C and other programming languages can be linked to programs written in the Assembly language. Thus a program written in C can call a function written in Assembly language and access variables declared within the Assembly language program. Similarly a program written in Assembly language can call functions and access variables of a high level language program. We will learn these methods to call C functions in Assembly language programs and the methods to call Assembly language functions in C programs.

All IA32 processors provide a special purpose register, called `eip`. This register stores the address of the memory location from where the next machine instruction is read. Each Assembly language instruction is first translated into the machine instruction by an assembler. After a machine instruction is fetched from the memory, the register `eip` is incremented by the size of the instruction, in bytes, so that it contains the address of the next machine instruction. When the execution of an

instruction is completed, the next instruction is read from the memory. The control transfer instructions modify the `eip` register implicitly so that the next instruction is read from a location different than the immediately next address. Therefore, the control transfer instructions require one operand which provides address of the next instruction, also known as target instruction address.

4.1 Specifying the target instruction address

There are several ways in which the destination address may be specified in a control transfer instruction. In Assembly language programs, the addresses of the instructions are typically defined by the use of the labels attached to the instructions. These symbolic labels are used in the control transfer instructions to specify the target instruction. Symbolic labels are replaced by the addresses of the instructions when an Assembly language program is translated to the machine instruction.

In IA32 processors, the target address in a control transfer instruction can be specified in one of the following ways.

- As an immediate constant (immediate addressing)
- As content of a specified register (register addressing)
- As content of a specified memory location (memory addressing)

4.1.1 Immediate addressing

In the immediate addressing mode, address of the target instruction is specified in the instruction as a constant. In Assembly language programming, symbolic labels are used to define the addresses of instructions. These labels can be used as immediate constants in various instructions including the control transfer instructions.

Most Assembly language programs use immediate addressing to specify the target in a control transfer instruction.

In GNU/Linux Assembly language, the syntax used to specify the operand as an immediate constant is different for the control transfer instructions as compared to the one for other instructions. Unlike the data operand addressing, the immediate operand of the control transfer instructions are specified without using a '$' before the immediate operand.

For example, in IA32 architectures, a `jmp` instruction is a control transfer instruction. If a label `swapbytes` is defined, then upon execution of the instruction `jmp swapbytes`, program starts executing from the instruction whose label is defined as `swapbytes`. In this instruction, argument to the `jmp` instruction is defined using an immediate addressing. As opposed to this, consider another instruction

`mov $swapbytes, %eax`, where the first operand is addressed using immediate addressing. The execution of this `mov` instruction causes the address `swapbytes` to be moved to register `eax`. Even though in both instructions, `swapbyte` is used as an immediate constant, no '$' is prefixed to it in `jmp` instruction.

The format of machine instruction for IA32 processors encodes the target address differently when it is specified as an immediate constant in the Assembly language. The encoding mechanism in the machine instruction can specify the target address as one of the following.

- As an offset relative to the address of the jump instruction (actually offset relative to the address of the instruction immediately after the jump instruction)
- As an absolute address

In the first case, the offset is with respect to the address of the next instruction (i.e. immediately after the jump instruction). This simplifies the execution of the instruction because the value stored in the register `eip` will be the address of the next instruction and during execution the offset can be added to the `eip` register. This mode is therefore also known as `eip` relative addressing mode.

`eip` relative addressing mode

The address of the target instruction can be specified as an offset from the address of the current instruction (in reality as an offset from the address of the next instruction). Since the register `eip` contains the address of the next instruction when an instruction is executed, the address of the target instruction is obtained by adding the value of `eip` register with offset specified in the machine instruction.

When a program is executed on the computer, the operating system first loads the program into the memory from the file. In order to load the program in memory, the operating system first selects an area in memory where the program is loaded from the file. The address of memory, where the program is loaded, is not known at the time of assembling the program into machine instructions. If the assembler chooses the address of target instruction to be coded as an absolute address, the machine instruction will have to be "patched" by the operating system such that it contains proper target address during the execution. If an address is coded in the machine instruction as an `eip` relative offset, irrespective of where the program is loaded in memory, the instruction will not require any "patch" while loading. The offset remains the same irrespective of where the program is loaded. Thus even without modification the instruction behaves in the same way.

The GNU/Linux assembler typically uses the `eip` relative offset addressing for specifying the target instruction in a control transfer in-

struction. In IA32 processors, while using `eip` relative addressing, the offset can be coded in 8-bits, 16-bits or 32-bits. The GNU/Linux assembler does not use 16-bit offsets in the instructions[1]. It chooses an 8-bit offset if address of the target instruction is close enough such that the offset fits within eight bits. Otherwise, the 32-bit offset is chosen.

Some examples of the instructions that use `eip` relative addressing are given below.

- `jmp swapbytes`. This instruction uses `swapbytes` as the address of the target instruction. The address is coded using `eip` relative addressing. Thus if the address of the next instruction immediately after this instruction is a, the immediate constant put in the machine instruction will be (`swapbytes` $- a$). The offset will be coded in one byte if the value of the expression (`swapbytes` $- a$) lies within -128 to $+127$. Otherwise the offset will be coded in 32 bits.

- `jmp 0x100`. This instruction uses 0x100 as an immediate constant. Thus the address of the target instruction computed at the run time will be $a + 0\text{x}100$, where a is the address of the instruction immediately after this instruction. It may be noticed here that this format is not used to make jump to an absolute address 0x100.

- `jmp external_var`. In this instruction a symbol `external_var` is used which is possibly defined in another file or library. Such references are resolved by the linker. This instruction is coded using the `eip` relative addressing but a 32 bit offset is used as the size of the offset is not known at the time of assembling the program. A machine instruction generated by the assembler corresponding to this type of Assembly language instruction is modified later by linker to have correct offset.

Absolute address

When the address of the target instruction is directly coded in the machine instruction, it is called an absolute address. Thus upon execution of such a machine instruction, value of the `eip` register is set to the constant specified in the instruction itself. In GNU/Linux Assembly language, the use of the absolute addresses is specified by putting a '`*`' in front of the target name in the instruction. In the protected mode of operation as configured in the GNU/Linux, the memory addresses are 32-bit wide. Therefore, address in the machine instruction is coded using 32 bits. Some examples of the absolute addressing are shown in the following instructions.

[1] Actually the assembler can be made to work in 16-bit mode which is not discussed in this book. In this mode, the offsets will be 16-bit in place of 32-bit.

- `jmp *swapbytes`. Execution of this instruction causes control to be transferred to target instruction at address `swapbytes`. The effect of this instruction is similar to instruction `jmp swapbytes`. However the address of the target instruction is specified using absolute addressing.
- `jmp *0x100`. In this instruction the target address is specified as an absolute address `0x100`. Upon execution of this instruction, the control is transferred to the instruction stored at location `0x100` in the memory. Notice that the effect of the instruction `jmp 0x100` is not the same as that of the instruction `jmp *0x100`. In this first case, 0x100 is the offset from the next instruction. If the address of the next instruction is a then the address of the target instruction would be $a + 0x100$ in the first case and it will be $0x100$ in the second case.

In Assembly language programs, the address of the target instruction is usually specified with `eip` relative addressing. However, in certain cases the absolute addressing is also used. For example, if a program transfers the control to a pre-determined location which is independent of the program, absolute addressing may be used. An example of such a case would be to transfer the control to an instruction that causes the system to reboot. In GNU/Linux operating system, such kind of usage are not permitted and therefore the direct addresses are rarely used in the control transfer instructions.

4.1.2 Register addressing

The target address of a control transfer instruction may be specified using a register. Upon execution of such a control transfer instruction, the contents of the specified register are copied to the `eip` register thereby transferring the control to an address given in the register.

Use of register for specifying the target address of a control transfer instruction is an indirect mode of addressing. In GNU/Linux Assembly language, `*` is used to prefix before the operand to denote an indirect mode of addressing. For example, the correct use of an instruction that causes the control to be transferred to an address given in register `eax` would be, `jmp *%eax` rather than `jmp %eax`.

4.1.3 Memory addressing

In memory addressing, the address of target instruction is read from memory. In GNU/Linux operating system, all addresses are 32-bit wide and therefore address of the target instruction takes four memory locations for storage. A '`*`' is required to be prefixed before the memory operand as it is an indirect mode of specifying address of the target instruction.

All memory addressing modes discussed earlier for specifying the arguments of an instruction can be used here. For example, the instruction `jmp *(%eax)` specifies the use of base addressing where the address of the target instruction is stored in the memory location whose address is available in register `eax`.

Some examples of the memory addressing for specifying the address of the target instruction are given below.

- `jmp *(memvar)`. The 32-bit address is stored in the memory location `memvar`. During execution of this instruction, 32-bit value is read from the memory location `memvar` and copied to `eip` register thereby transferring the control.
- `jmp *table(%eax)`. The 32-bit address of the target instruction is stored in memory and the address of the memory location is obtained by adding displacement `table` to the contents of the register `eax`. During the execution of this instruction, first the address of memory location is computed and then the 32-bit value is read from that memory location. The value read from memory is stored in `eip` register thereby completing the control transfer.
- `jmp *table(%eax,%esi)`. The address of the memory is computed by adding displacement `table` to the contents of registers `eax` and `esi`. The 32-bit value read from the memory is stored in `eip` thereby transferring the control to the target.
- `jmp *table(%eax,%edi,4)`. The addressing mode in this instruction has all the four components of the memory addressing, i.e., displace, base, index and scale. The address of the target instruction is read from the memory location `table` + `eax` + 4 ∗ `edi`.

Memory addressing is a very powerful mechanism. Using this mechanism, for example, function addresses can be passed as arguments to another function. Such a feature is often used in high level language programs.

EXERCISES:

4.1 What is the difference between the following two instructions?
`jmp table` and `jmp *table`

4.2 Does the addressing mode in the instruction `jmp *$table` make any sense? Try it on your assembler and see. Explain the results.

4.2 Some control transfer instructions

The most commonly used control transfer instruction is `jmp`. The syntax of the `jmp` instruction is as the following.

```
jmp target
```

The `target` can be specified using any one of the addressing modes described earlier.

Upon execution of the `jmp` instruction, the register `eip` is set to the new value. Since the next instruction is read from the location whose address is stored in `eip` register, the `jmp` instruction causes control of the program to be transferred to new location.

IA32 architectures also support conditional control transfer. Upon execution of such instructions, control is transferred to new location only after the specified condition is evaluated to true. These instructions provide a very powerful mechanism for building program control. The general syntax of the instruction is as follows.

```
jcc target
```

Only the `eip`-relative addressing is permitted in the conditional jump instructions. Therefore no addressing mode based on registers or memory may be specified. In these instructions, the `target` is specified as a label of the target address. The GNU/Linux assembler automatically finds out the offset from the current instruction and encodes it as an 8-bit offset or a 32-bit offset depending upon whether the offset can fit within the eight bits or not.

The *cc* component specifies the condition under which the control transfer should take place. The conditions are evaluated at the time of execution based on the flags (as given in section 3.2). Table 4.1 gives various possibilities for conditional control transfer instructions.

Let us look at the use of these instructions by taking examples. Consider the scenario where we need to compute a function $f(x)$ as following.

$$f(x) = \begin{cases} 2x \text{ for } x > 2 \\ (x+2) \text{ for } x \leq 2 \end{cases}$$

To simplify the program, let us consider x to be an integer given in register `ebx`. The program given in figure 4.1 computes the value of function $f(x)$ and puts it in register `eax`.

The program is written in a way that needs a little explanation. First the register `eax` is set to 2. If the value of x in register `ebx` is more than 2, the register `eax` is set to x. Thus the register `eax` contains 2, if $x \leq 2$, or x if $x > 2$. By adding x to this value of register `eax`, desired value of the function is obtained.

In this program we used signed integers for computation. Therefore, value of x can be between 2^{-31} to $2^{31} - 1$. This program can be made to handle unsigned numbers. In unsigned version of the program, the value of the variable x can be between 0 and $2^{32} - 1$. Program that handles unsigned numbers is obtained by changing `jle` instruction to `jbe` in figure 4.1.

Table 4.1: Conditional Jump instructions

Instruction	Condition for jump	Flags
`je`	Equal	ZF=1
`jz`	Zero	ZF=1
`jne`	Not Equal	ZF=0
`jnz`	Not Zero	ZF=0
`ja`	Above	CF=0 and ZF=0
`jnbe`	Neither Below nor Equal	CF=0 and ZF=0
`jbe`	Below or Equal	CF=1 or ZF=1
`jna`	Not Above	CF=1 or ZF=1
`jae`	Above or Equal	CF=0
`jnb`	Not Below	CF=0
`jnc`	No Carry	CF=0
`jb`	Below	CF=1
`jnae`	Neither Above nor Equal	CF=1
`jc`	Carry	CF=1
`jg`	Greater	ZF=0 and SF=OF
`jnle`	Neither Less nor Equal	ZF=0 and SF=OF
`jng`	Not Greater	ZF=1 or SF<>OF
`jle`	Less or Equal	ZF=1 or SF<>OF
`jge`	Greater or Equal	SF=OF
`jnl`	Not Less	SF=OF
`jnge`	Neither Greater nor Equal	SF<>OF
`jl`	Less	SF<>OF
`jo`	Overflow	OF=1
`jno`	No Overflow	OF=0
`js`	Sign	SF=1
`jns`	No Sign	SF=0
`jp`	Parity	PF=1
`jpe`	Parity Even	PF=1
`jnp`	No Parity	PF=0
`jpo`	Parity Odd	PF=0

```
        /* Register ebx contains x  */
        movl    $2, %eax /* Initial value of eax is 2 */
        cmpl    $2, %ebx /* Check if x <= 2 */
        jle     L0
        movl    %ebx, %eax /* if x>2, put x into
                              register eax */
L0:     addl    %ebx, %eax /* Before execution of this
                              instruction, register eax
                              contains 2 (if x<=2) or
                              x (if x>2). After the
                              execution eax is 2+x or
                              2x. */
```

Figure 4.1: Simple use of conditional control transfer instructions

4.3 Building loops in programs

One important use of conditional control transfer instructions is to build sections in the programs that are iterated over again and again. The loops have three important parts.

- **Initialization code**. This part of the program is used to prepare for the entry into the loop. Initialization code is executed only once in the beginning just prior to entering in the loop. Thus this can be used to initialize variables like the loop iteration counter.
- **Condition testing code**. This part of the program is used to check condition prior to starting a new iteration of the loop. This part is executed each time the loop is iterated over. As soon as the condition evaluates to false, the execution of the loop is terminated by performing a conditional jump out of the loop.
- **Body**. This part of the loop is the code that is iterated over again and again.

Let's consider an example of a program that evaluates the sum of first n integers by iteration. We use two registers – `eax` to store the sum, and `ebx` to store the iteration count. Ideally register `ebx` should vary from 1 to n in steps of 1 for each iteration of the loop. We however will use register `ebx` as a down counter from n to 1 in this program.

Initial values of registers `eax` and `ebx` are set to 0 and n respectively. At the end of the program, register `eax` will contain sum of all integers from 1 to n. The conditional testing code will check if register `ebx` contains a number that is more than 0 or not. The loop terminates as soon as register `ebx` reaches a value 0. In the body of the loop, we

add contents of register ebx into the contents of register eax. We then decrement register ebx and transfer control to the condition testing code. The program is given in figure 4.2.

```
        /* Initialization code for the loop */
        movl    $0, %eax /* Sum */
        movl    $n, %ebx /* The number n */
        /* Condition testing code */
loopstart:
        cmpl    $0, %ebx /* Check it with 0 */
        jle     exitloop /* If value is <=0, exit */
        /* Loop body */
        addl    %ebx, %eax
        dec     %ebx
        jmp     loopstart
        /* At this point, register eax contains
           the sum. Register ebx will be 0 or
           less than 0. */
exitloop: . . .
```

Figure 4.2: Building loops with control transfer instructions

---

EXERCISES:

4.3 In the program in figure 4.2, the conditional control transfer is achieved by the jle instruction instead of the je instruction. What do you think is the reason for the same?

4.4 Assume that we need to implement the following C construction.

```
x = (y>0)?y:0;
```

Write a program to implement this C construct. Assume that register ebx contains the value of integer variable y prior to starting the program. The value of integer variable x is computed and put in to register eax. You must write this program in two different ways – using conditional control transfer instructions and using the conditional move instructions.

4.5 Write a program to implement the following C construct.

```
y = 0;
while (x >= 0) {
    x = x - y;
    y = y + 1;
}
```

Use registers eax and ebx for keeping variables y and x respectively. Assume that the initial value of the register ebx contains the original value of x.

Often one need to build loops that are executed a fixed number of times. A simple such loop is similar to the C code shown in figure 4.3.

```
        for (i=0; i<n; i++) {
            A[i] = i;
        }
```

Figure 4.3: A simple loop that executes a fixed number of times

IA32 processors provide several instructions to handle the loops. In these instructions registers `cx` or `ecx` can be used as loop counter. Syntax for these instructions is the following.

```
jcxz target
jecxz target
loop target
loopz target
loope target
loopnz target
loopne target
```

In all of these instructions, the target is coded using `eip`-relative addressing with only 8-bit offset. Therefore, the target instruction can not have an offset more than $+127$ or less than -128.

The `jcxz` instruction transfers the control to the `target` if register `cx` contains a value 0. Otherwise the execution continues from the next instruction onwards. The `jecxz` instruction is also similar to the `jcxz` instruction except that register `ecx` is used instead of `cx`. The C loop given in figure 4.3 can be implemented in Assembly language using the `jecxz` instruction as follows.

```
        /* Initialization code for the loop */
        movl    $n, %ecx
        /* Condition testing code */
loopstart:
        jecxz   exitloop
        /* Loop body */
        movl    %ecx, A-4(,%ecx,4)
        decl    %ecx
        jmp     loopstart
exitloop:
```

It may be noted that in this program, the loop counter `ecx` is decremented each time an iteration is over. Further the loop counter takes values from `n` to `1` in the loop body. If `A` is the starting address of the

array, address of element A[i] (i=o to n) will be A+4*i. Since the value of the loop counter varies n to 1, the address of the element is given by A+4*(ecx-1), or A-4+4*ecx. This expression is used while specifying memory operand in movl instruction.

The loop instruction first decrements ecx register and then if it contains a non-zero value, control is transferred to the target. In GNU/Linux Assembly language format it is also possible to use cx register as counter for the loop instruction. This is however out of scope of this book and is not discussed here.

The C program given in figure 4.3 can now be written in a more compact form using the loop instruction. In this modified program, the jecxz instruction is used to check for the initial value of the register ecx not being 0 before entering into the loop body. This ensures that the first time loop is not entered if register ecx has a zero value. Within the loop body, decl, jmp and jcxz instructions are all replaced by a single loop instruction.

```
        /* Initialization code */
        movl    $n, %ecx
        jecxz   exitloop
loopstart:
        /* Loop body */
        movl    %ecx, A-4(,%ecx,4)
        loop    loopstart
exitloop:
```

Two instructions loope and loopz are essentially the same. In these instructions, the jump to target is carried out if register ecx (after decrementing) is not 0 and the ZF flag is set to 1. It may also be noted that by decrementing the ecx register (in any of the loop instructions), no flags are affected. Thus ZF flag in this instruction essentially refers to the value of the flag which might have been modified in the loop body prior to the execution of loope instruction.

The other two instructions, loopne and loopnz are also the same (two different mnemonics for the same instruction). Functionally these instructions cause the control to be transferred to the target if register ecx (after decrementing) is not 0 and the ZF flag contains a value 0. As in the loop instruction, all other loop*xx* instructions first decrement the contents of register ecx.

EXERCISES:

4.6 Write a loop that compares two byte arrays, A and B and terminates as soon as a mismatch is found. The index of the mismatched byte is returned in register ecx. Assume that size of each array is n ($n > 0$), and the comparison starts from by comparing A[n-1] with B[n-1] down to comparing A[0] with B[0]. Use the loope instruction. The ZF flag at the

end of the loop will determine whether a mismatched occurred or not. If ZF=1, no mismatch occurred while if ZF=0, a mismatch occurred at index given in register ecx.

4.7 Write a loop that finds out if all elements in an integer array A contain the same value or not. Assume that size of the array is n ($n > 0$), and the value to be checked is given in register eax. The loop returns the result in ZF flag. If at end of the program, ZF is 0, all elements do not contain the same value. If ZF is 1, all elements contain the same value.

4.4 Function calls and returns

In various programming practices, functions are implemented by programmers. These functions are then called in the programs. The functions are sub-programs to which the control of execution is passed temporarily. After execution of the function is completed, control is transferred back to the calling program.

There are several reasons to why functions should be used. The most important reason is to 'reuse' the code. If a particular operation is to be done several times in the program, the most logical way to doing so is to use functions. For example, if in a program several strings are to be displayed on the screen at different times, it is better to write a function for that. Each time a string is to be displayed, the function can be called. Thus the code for displaying string is not written again and again.

Another reason for writing functions in a program is to make it readable and understandable. Often simple operations can be written out as functions and then later used in the program. While reading the program, one does not have to read all the instructions to understand how does the program work as long as the behavior of the function is known.

While calling the functions the machine stack is very helpful. Consider the figure 4.4 where function f2 is called during execution of function f1. While executing code of function f2, a call is made to another function f4. Thus the return sequence must be to first return from function f4 to function f2 and then from function f2 to function f1. The order of return to various functions is reverse of the order of calls and it follows a Last In First Out (LIFO) order. As explained in chapter 3, the stack is the most natural choice for building the function calling sequences.

In IA32 processors, functions are called by executing a call instruction. The syntax for the call instruction is as follows.

```
call target
```

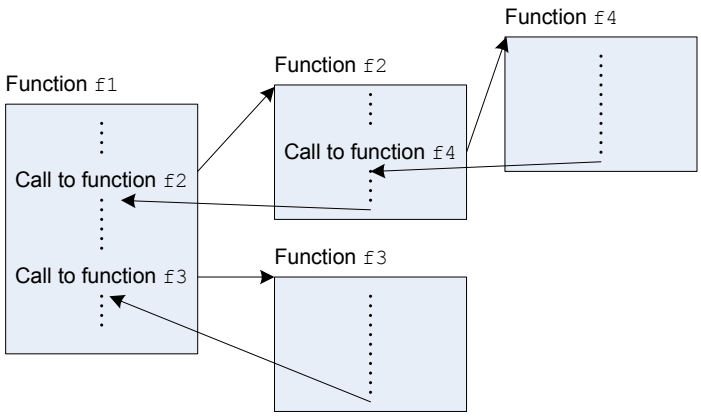


Figure 4.4: Execution of call and return instructions

The `target` can be specified using any one of the addressing modes described earlier in this chapter. Both `eip`-relative and absolute mode of addressing is supported. The `call` instruction is similar to the `jmp` instruction except that address of the instruction next to the `call` is saved on the stack so that subsequently control may be returned back to the caller.

Let us consider an example of `call` instruction. In the following program let us assume that address of function `func` is 0x1100 and the contents of register `esp` prior to executing the `call` instruction is 0x5500. Let us assume further that address of the instruction after the `call` instruction is 0x100C.

```
                .
                .
                call func
0x100C          push %eax
                .
                .
0x1100    func:
                .
                .
```

The contents of relevant memory locations and registers before and after the execution of the `call` instruction are shown in figure 4.5.

Just prior to execution of the `call` instruction, `eip` register contains address of the `call` instruction. During execution of the `call` instruction, address of the next instruction (address of the `push` instruction) is saved on the stack. Therefore stack pointer register `esp` is decremented by four and a 32-bit number 0x0000100C is stored in four bytes starting at address given by the `esp`, i.e., 0x000054FC. The

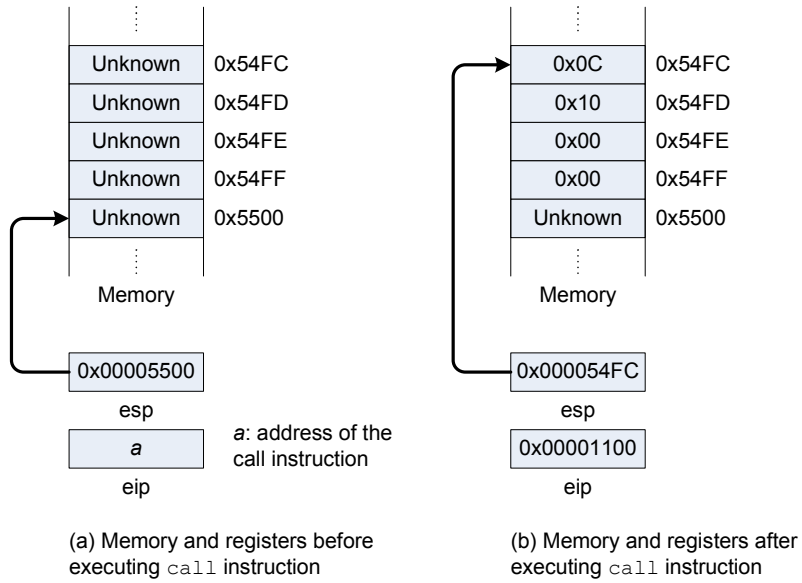


Figure 4.5: Execution of a call instruction

register `eip` is then changed to 0x00001100 thereby making a control transfer to function `func`. When the execution of the function is completed, the control is returned back to the `push` instruction using the saved value on the stack.

IA-32 architectures provide an instruction called `ret` to return control back to the caller after execution of the called function is completed. The syntax for the `ret` instruction is simple. It takes 0 or one argument as an immediate constant. The constant is used to adjust the stack pointer just immediately after the return, an operation performed in conjunction with the parameter passing. The syntax of the `ret` instructions is as shown below.

```
ret
ret n
```

The `ret` instruction pops a 32-bit offset from top of the stack into `eip` register thereby causing a control transfer. As `call` instructions push address of instruction to which control is passed upon return, `ret` instruction is used in functions to return control back to the caller. The second variation of the `ret` instruction also adds `n` to register `esp` after `eip` is popped from the stack top. We shall see the use of this variation later in this chapter.

4.5 Passing Parameters

The functions can be called with parameters passed from the caller. There are several ways in which parameters can be passed. As values of parameters are passed by the caller and used by the function, the parameter passing conventions are implemented by both of them.

4.5.1 Passing parameters through registers

In the simplest possible way, the parameters can be passed through registers. The caller can initialize registers with appropriate values and function can assume that these registers contain the initialized values. Let's consider this scheme with the following example. Here a program for a function is given that converts one 8-bit signed integer to 32-bit signed integer. The function takes its input parameter (8-bit number) in register `al` and returns the 32-bit output in register `ebx`.

```
        /* Function that converts an 8-bit signed
         * number to 32-bit signed number.
         * Input: 8-bit number in register %al
         * Output: 32-bit number in register %ebx
         * /
convert:
        movsbl   %al, %ebx
        ret
```

The function `convert` has just two instructions. The `ret` instruction is used to return control back to the caller. The function assumes that register `al` contains the parameter (8-bit signed integer).

This function can be used by the caller for converting any 8-bit integer to 32-bit integer. For example, if 8-bit number is stored in some memory variable, the following example can be used to convert it to 32-bit integer.

```
        /* Example to use the function
         * with parameter passing using
         * register */
        mov    MemVar8, %al
        call   convert
        mov    %ebx, MemVar32
```

The first instruction in this example takes 8-bit number stored in a memory location (denoted by `MemVar8` in this example) and copies it to register `al`. As function `convert` is called, the parameter is already in place. The return value of `convert` function (in register `ebx`) is used by the caller to initialize memory location `MemVar32`.

4.5.2 Passing parameters through memory

CPUs usually provide a small set of registers which are insufficient to pass large number of parameters. The idea of passing parameters through registers can be extended to passing them through memory. Some part of the memory can be used to pass parameters from caller to the called function. For example if the previously defined function `convert` is to be modified to takes parameters from the memory, we can use a memory location, say `P1` to pass the parameter. The modified function will be the following.

```
        /* Modified function to converts 8-bit
         * number to 32-bit signed number.
         * Input: 8-bit number in memory P1
         * Output: 32-bit number in memory P2
         * /
convert:
        push   %eax
        movsbl P1, %eax
        mov    %eax, P2
        pop    %eax
        ret
```

In order to call modified `convert` function, the caller should place parameter in memory location `P1`. The result after the execution of `convert` function will be available in memory location `P2`. For example, if the calling function in the previous example is to be used with this function, it will have to first copy the variable from memory location `MemVar8` to `P1`. After call to function `convert`, the calling function will have to again copy results from memory location `P2` to memory location `MemVar32`. The modified calling function is the following.

```
        /* Example to use the function
         * with parameter passing using
         * fixed area in memory */
        mov    MemVar8, %al
        mov    %al, P1
        call   convert
        mov    P2, %eax
        mov    %eax, MemVar32
```

4.5.3 Passing parameters through stack

The two parameter passing mechanisms discussed so far can not be used in the cases where functions call themselves, such as in recursion. The problem arises from the fact that two invocations of the same

functions use the same storage in memory for the parameters. Thus in order to make a call to the function second time, the parameters of the first call are overwritten. This problem can be solved by passing parameters through a stack. In this approach, for each invocation of the function, a new area is alloted on the stack and the function execution is carried out without interfering with the parameters of an earlier invocation.

The parameters are passed through the stack by pushing them on the stack in the caller function. The called function can then use the parameters available on the stack. In order to find the address of the parameters, register `esp` can be used as it contains the address of the top of the stack. Let us assume that we need to call a function with one 32-bit wide parameter. In order to make a call, the caller first pushes the value of this parameter on the stack and then calls the function. When call to the function is made, the `call` instruction pushes the return address on the stack before transferring control to the called function. Thus in the called function, top of the stack will contain the return address and the first parameter will be stored just below the return address on the stack (figure 4.6).

In IA32 processors, the top of the stack can be accessed using register indirect addressing with `(%esp)` as the operand. Similarly the address of the first parameter is four more than the stack pointer (value stored in the register `esp`). Therefore the first parameter can be accessed using `4(%esp)` as the operand (base + displacement addressing).

Upon return, the parameters pushed by the caller are to be removed from the stack. This can be done in two ways in IA32 processors.

- *Explicit removal of parameters.* The caller can remove parameters from the stack either by executing `pop` instructions and discarding the values or by just adjusting `esp` register. In our case, four bytes of parameters were pushed prior to making a call to the function. Thus the parameters can be removed by adding 4 to the value in register `esp`.
- *Implicit removal of parameters.* The called function can remove the parameters from the stack by executing the second version of the `ret` instruction. Recall that the argument of `ret` instruction can be a constant which is added to register `esp` after removing return address from the stack. Thus in our example, instruction '`ret 4`' can be executed by the called function to return control to the caller.

4.5.4 Parameter passing conventions

There are two parameter passing conventions that are used by most programming languages.

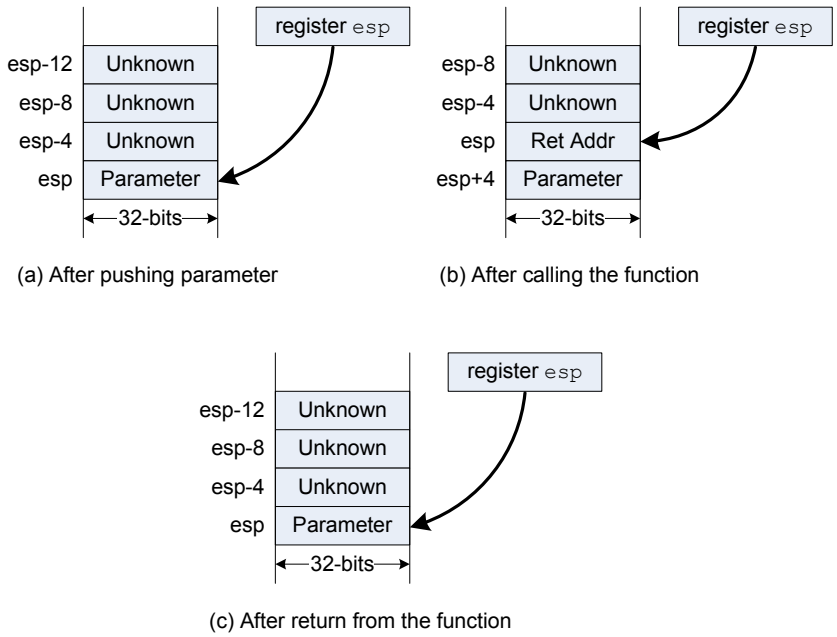


Figure 4.6: Passing parameters through stack

- Parameter passing by value
- Parameter passing by address or reference

In addition to these, there are a few other conventions that are not so commonly used. These are out of scope of this book and are not discussed.

As is clear from the examples discussed so far, the parameters are passed to the called function by copying their values to registers, memory or stack. Thus, if the called function changes the value of the parameter, the original value remains unchanged and only the copy is changed. In such a convention, parameters are said to be 'passed by value'.

Some programming languages also provide mechanisms to pass addresses of the parameters (instead of their values) to the called function. In such a case, the called function can modify the value stored at that address thereby affecting the original variable itself. Such a convention is known as parameter passing by reference. In such a case, the called function uses the addresses passed to it as parameters, to refer to the data.

There is a subtle difference between parameter passing by reference and passing reference as a parameter. Programming language C does not support parameter passing by reference. However the pro-

grammer can pass address or reference to a variable in his function and use '*' operator to refer to the value of the variable in code of the called function. This scheme however is not really parameter passing by reference. On the other hand, programming languages such as C++, PASCAL, Java etc. support parameter passing by reference.

Let's consider an example where we need to exchange two 32-bit numbers stored in memory locations `NumA` and `NumB`. We use a function called `exchangeNums` with two 32-bit parameters passed through the stack. In the called function, the first parameter is stored at location given by `4(%esp)`. Similarly the second parameter is stored at location given by `8(%esp)`. Let's first see the code of the caller as given below.

```
        :
        pushl NumB
        pushl NumA
        call  exchangeNums
        addl  $8, %esp
        :
```

The caller first pushes two parameters in reverse order. Thus in function `exchangeNums`, the top of the stack contains the return address, the value immediately below this on the stack is `NumA` that can be referred to by `4(%esp)` and the next value on the stack is `NumB` referred to by `8(%esp)`.

We now discuss an implementation of function `exchangeNums`.

```
exchangeNums:
        mov   4(%esp), %eax
        xchg  %eax, 8(%esp)
        mov   %eax, 4(%esp)
        ret
```

Let's look at the behavior of the function as shown in figure 4.7. In this figure, it is assumed that the initial values of variables `NumA` and `NumB` are 0x1016 and 0x9805 respectively. After the execution of each of the instruction in function `exchangeNums` the changes in the memory contents are shown in this figure. Just prior to the execution of `ret` instruction (figure 4.7(e)), the two parameters were exchanged. However, the variables `NumA` and `NumB` stored separately in the memory did not get changed. This is because the values of parameters were passed to the function and therefore a copy was made on the stack. Only the copies got interchanged rather than the variables themselves.

The problem is solved by passing the addresses of variables `NumA` and `NumB` to the function instead of their values. The function will also need a modification as it will have to deal with the addresses rather than the values.

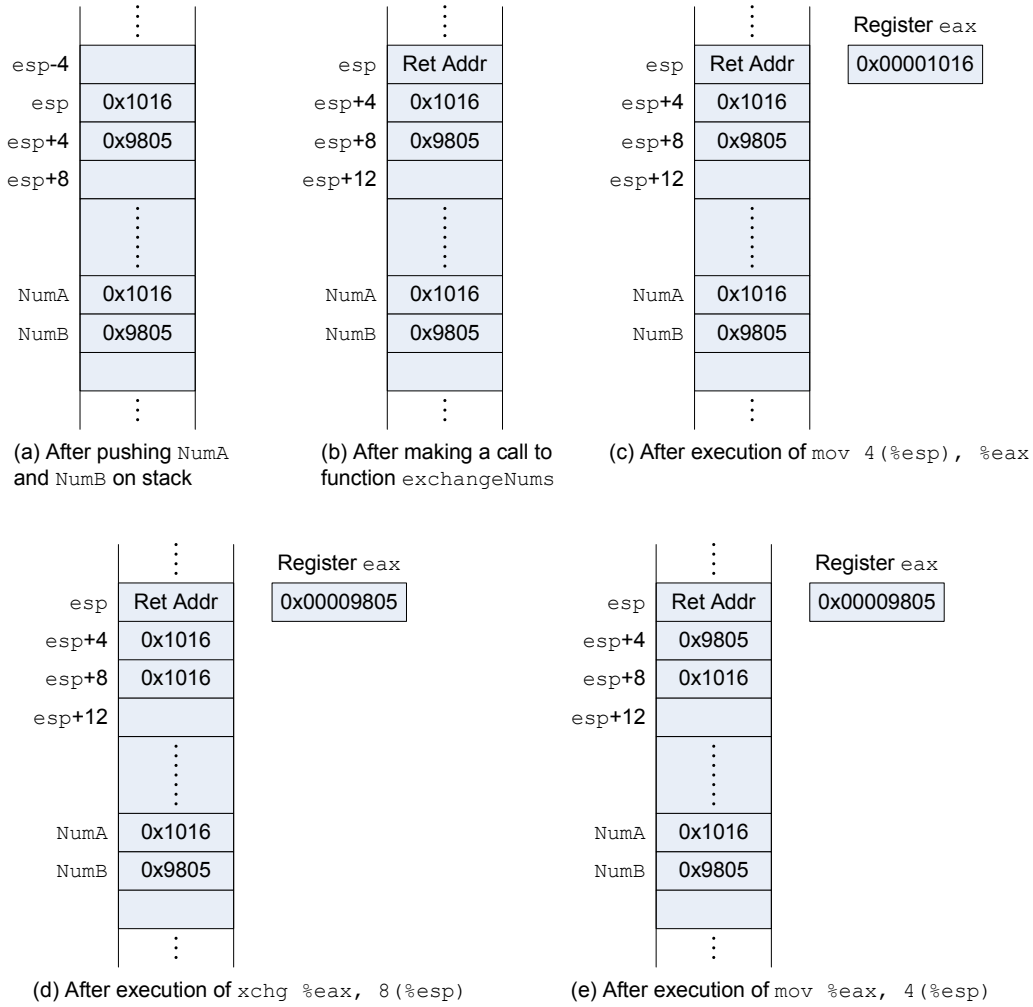


Figure 4.7: Memory contents while executing `exchangeNums`

The modified structure of the caller function is the following.

```
        :
        pushl $NumB
        pushl $NumA
        call  exchangeNums
        addl  $8, %esp
        :
```

Notice that the addresses of NumB and NumA are pushed on the stack because of the immediate addressing in pushl instruction. The correct implementation of the function exchangeNums is given below.

```
exchangeNums:
        mov   4(%esp), %eax
        mov   8(%esp), %ebx
        xchg  (%eax), %ecx
        xchg  (%ebx), %ecx
        xchg  (%eax), %ecx
        ret
```

The function first copies the addresses of the two parameters in registers eax and ebx. Using a series of xchg instructions, the values of variables NumA and NumB are changed rather than just the parameters.

EXERCISES:

4.8 Trace the modified function exchangeNums after the execution of each instruction. Show the memory contents as well as the contents of registers eax, ebx and ecx.

4.9 Another incorrect version of the function exchangeNums is given below. The function assumes that the parameters are passed using addresses. What are the problems with this function? Does it exchange the parameters?

```
exchangeNums:
        xchg  4(%esp), %eax
        xchg  8(%esp), %eax
        xchg  4(%esp), %eax
        ret
```

4.5.5 Local variables for functions

Often in high level language programs, functions need local variables whose scope is limited within the function. There are several ways to implement local variables.

- Allocating registers for local variables
- Using a fixed area in the memory for local variables
- Using stack

Local variables can be stored in the registers. We have been using them in all the functions written earlier. For example, we have been using registers `eax`, `ebx` and `ecx` to store pointers to two parameters and a temporary variable in the implementation of function `exchangeNums`. If storage requirement for local variables is only of a few bytes, registers provide an excellent place for keeping them. However if the parameters are kept in registers, during recursion and call to other functions, it has to be ensured that they do not change in an unexpected manner. The registers can be pushed on the stack and popped later to restore the value.

The other alternative of keeping local variables in memory is useful if the storage requirement is large and can not fit in the registers. However, similar to the registers, the memory variables need to be saved somehow to solve the problem of unexpected changes during recursion and call to other functions.

The last technique is the most commonly used one. In this, stack is used for storing the local variables and therefore, unintended modifications to the variables during recursion and other function calls are avoided. Each time a function is called, a new area is alloted on the stack which is not in use by any other active function. In this technique, when a function is called, the function first allocates space on the stack to keep local variables. It may be recalled that in IA32 processors, the stack pointer is decremented for each push operation. Thus at the entry of the function if a constant is subtracted from the stack pointer, an area is created on the stack that can not be used by any future push operations on the stack. This area can then be used for storing local variables as shown in figure 4.8.

The local variables can be accessed in a program using `esp`-relative addressing. For example, if one requires a space for 16 bytes of local variable storage, the first instruction in the function can subtract 16 from `esp` register. The local variables can then be stored in memory locations `(%esp)` to `15(%esp)`. The return address will be at `16(%esp)` and the parameters passed by the caller will be at `20(%esp)`, `24(%esp)`, etc. Just prior to executing the `ret` instruction, the stack pointer will need to be restored to the address of the location that contains the return address. This can be achieved by adding the same constant to `esp` register.

One may like to note here that at any point in time if the function pushes some registers on the stack, the same local variables will then be found at another offset with respect to the stack pointer. This

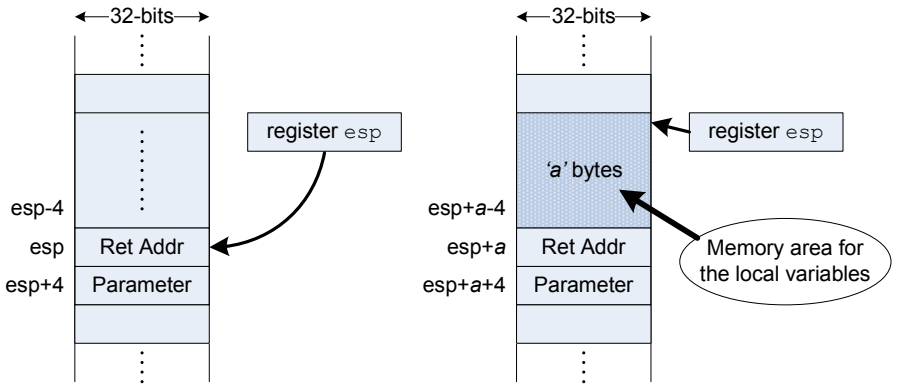


Figure 4.8: Allocation of local variables on stack

often creates a problem in program writing. The programmer will be required to keep track of the stack pointer to refer to the local variables. The offset from the stack pointer changes each time a push or pop is performed on the stack.

The solution to this is rather simple. One can use another register to note down the value of `esp` register just prior to subtracting the constant from the stack pointer. Offset from this register can be used to access the local variables and the parameters of the function. Typically register `ebp` is used for this purpose. One may notice that following items are allocated on the stack in a contiguous manner (figure 4.8(b)).

1. Values of the parameters pushed by teh caller.
2. Return address pushed because of the execution of `call` instruction.
3. Storage for local variables.

This entire region in the stack is known as the frame or activation record for the called function. A frame carries all values and variables which are local to the invocation of a function. Given an address of this frame, all local variables can be accessed within the frame. In GNU C environments, register `ebp` is used to store the address of the current frame. Therefore when a function is called, it must save the frame pointer (or register `ebp`) corresponding to the frame fo the caller and must restore it back before returning control to the caller. This almost defines the entry and exit codes for functions that must use a frame as defined by the run time system of the GNU C compiler.

Let's consider a function that takes two 32-bit parameters and requires four 32-bit local variables. The function will be executed as

usual by pushing two parameters on the stack and then making a call to the function. The implementation of the function is shown below.

```
1  FunctionExample:
2          push  %ebp
3          mov   %esp, %ebp
4          sub   $16, %esp /* 16 bytes local storage */
5          /*
6            At this point the return address is stored
7            at location 4(%ebp) and the parameters are
8            at 8(%ebp) and 12(%ebp).
9            Local variables are at -4(%ebp), -8(%ebp),
10           -12(%ebp) and -16(%ebp). Old value of %ebp
11           is stored at (%ebp).
12         */
13
14         :
15         Function Body
16         :
17
18         /*
19            Just prior to the return
20            esp and ebp registers are restored.
21         */
22         mov   %ebp, %esp
23         pop   %ebp
24         ret
```

In this code instruction at line 2 saves the old frame pointer on stack. The new frame pointer is set to within the frame of the function by `mov %esp, %ebp` instruction in line 3. Finally the space for 16 bytes of local variables is allocated on stack in line 4. Register `ebp` at this point refers to the frame made for the function. Local variables, parameter values and old frame pointer are all acccessible as memory operands.

At the time of the exit from this function (line 22-24), frame of this function is deallocated, frame pointer of the caller is restored and control is returned to the caller where parameters are removed from the stack.

EXERCISES:

4.10 Write a function `reverseInts` that takes address of four integer arguments (say a, b, c and d) as parameters and reverses them. Thus after the execution of this function, the values of d and a will be interchanged. Similarly the values of b and c will be interchanged. The program must allocate one local variable (which it may or may not use for any purpose) and must use stack to pass parameters and to store the local variable.

4.11 Modify the function in exercise 4.10 to use the register `ebp` relative addressing for the local variables and the parameters.

4.5.6 Stack management in calling functions

The calling function pushes parameters for the function on stack. After call to the function is completed, parameters are either removed by the called function or by the calling function. In case, the parameters are to be removed by the called function, the instruction `ret n` can be used where `n` is the total size of the parameters in bytes. This form of the `ret` instruction removes the return address from the stack and before transferring the control to that location, it adds `n` to the stack pointer.

As for each call, parameters are pushed on the stack, new area is given for each invocation of the function. For example, in case of recursive functions, when a function is called within itself, new set of parameters are pushed on the stack. Thus these pushed parameters do not share the same area with parameters of an earlier invocation of same function.

Let us consider implementation of recursion with an example. For this we will write a function `sum` that returns the sum of an array of integers. The function takes two arguments – the address of the array of 32-bit integers, and the number of elements of the array to be added. It returns the result in register `eax`. In order to implement this operation, recursion is used. The sum of n elements of the array can be defined as sum of $n-1$ elements of the same array added with nth element. Thus the function sum can be defined as follows.

$$\text{sum}(A,\ n) \ = \begin{cases} \text{sum}(A,\ n-1) + A[n-1] & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

An implementation of this function in C is as follows.

```
int sum(int *A, int n) {
    if (n > 0)
        return sum(A, n-1) + A[n-1];
    else
        return 0;
}
```

Implementation of this function `sum` in Assembly language is given below.

```
    sum:
       /* At the entry of the function,
        * the first parameter is the address of the
        * array. The second parameter is the number
```

```
        * of elements of the array to add.
        */
[01]   push %ebp
[02]   mov  %esp, %ebp
       /* The first parameter is at 8(%ebp).
        * The second parameter is at 12(%ebp).
        * Old value of the ebp register is at (%ebp)
        * Return address is at 4(%ebp)
        */
[03]   mov  12(%ebp), %ebx
[04]   cmp  $0, 12(%ebp)
[05]   je   basecase
[06]   dec  %ebx   /* call sum(A, n-1) */
[07]   push %ebx   /* Second parameter = n-1 */
[08]   pushl 8(%ebp) /* First parameter is same as the */
[09]   call sum      /* first parameter to this call */
[10]   add  $8, %esp /* Remove parameters */
[11]   mov  8(%ebp), %ebx /* ebx = n */
[12]   mov  12(%ebp), %esi
[13]   add  -4(%ebx, %esi, 4), %eax /* A[n-1] */
[14]   mov  %ebp, %esp
[15]   pop  %ebp
[16]   ret
    basecase:
[17]   xor  %eax, %eax /* eax = 0 */
[18]   mov  %ebp, %esp
[19]   pop  %ebp
[20]   ret
```

Let's see how this function works. Function `sum` is called for the first time by another function in the program. In order to call function `sum`, the caller pushes count of the number of elements in the array and the address of the array on stack in that order. It then executes a `cal` instruction to transfer control to function `sum`. In the body of function `sum` the parameters are checked for the base case execution. Thus in line 3 and 4, the second argument at `12(%ebp)` (number of elements to be added) is checked for 0. If the base case condition is true, a jump is made to location `basecase`. Here a value of 0 is set in register `eax` (line 17). Code in lines 18 to 20 just restores register `ebp` and `esp` and makes a return to the caller. Thus to the caller, register `eax` contains the value returned by the function `sum`.

If the second argument of the function has a value other than 1, the program continues at line 6. In here, a call to `sum` function is made to get the sum of first $n-1$ elements. It is then added to the value $A[n-1]$. In order to make another call to function `sum`, register `ebx` (it contains n) is decremented by 1 and pushed on the stack (line 6 and

line 7). The address of the array remains the same and therefore it is just copied from `8(%ebx)` to the stack (line 8). The function `sum` is now called again as all its parameters are in place. After the function returns, register `eax` contains sum of the numbers. First the stack is adjusted to remove the parameters that were pushed on stack prior to making the function call (line 10). After that $A[n-1]$ is added to the value in register `eax`. Since register `eax` already contains sum of the first $n-1$ elements, by adding $A[n-1]$ into register `eax`, register `eax` gets modified to contain sum of the first n elements. This operation is carried out by instructions in lines 11 to 13. Instructions in lines 11 and 12 update registers `ebx` and `esi` to contain the address of the first element of the array and the number n respectively. Addressing mode used in line 13 ensures that the proper value ($A[n-1]$) is added to the register `eax`. Lines 14 and 15 make a return to the caller.

Let's see the behavior of the program with an example where we add all numbers of a 3 element array A. Thus the function `sum` is called with address of the first element of the array A, and the number of elements – 3, in our example. Assume that the array elements are 8, 10 and 12 respectively for A[0], A[1] and A[2]. A flow of the control is as shown in figure 4.9. In figure 4.9(a), the memory image shows the array A stored in memory location 0x2001C onwards. Each element takes four bytes in the memory for storage. Just as the function `sum` is called, the top to the stack contains the return address. Just below the return address, the address of the array 0x2001C is stored as the first parameter. The second parameter has a value 3 that denotes the number of elements to be added and it is also stored on the stack right below the first parameter. As the function proceeds for its execution, the register `ebp` is pushed on the stack and subsequently another call is made to the function `sum` with array address, 0x2001C, and number of elements, 2, on the stack (figure 4.9(b)). Third time the function is called with number of elements being 1. Finally, the fourth time the function is called with number of elements being 0 (figure 4.9(d)). This is the base case. Therefore register `eax` is set to 0 and the control is returned to the previous call to the function `sum` as shown in figure 4.9(e). At this point n is 1 and hence $A[0]$ (or 8) is added to register `eax`. Thus register `eax` contains a value 8 (figure 4.9(f)). The function then adds second element into `eax` before returning to its caller. Therefore the register `eax` contains 18 when the control is returned to the first call to function `sum` (figure 4.9(g)). At this point, the function adds 12 to register `eax` and returns to the caller (figure 4.9(h)).

The function `sum` returns its value using a register `eax`. In fact, such a method is used in most high level languages to return value of a function.

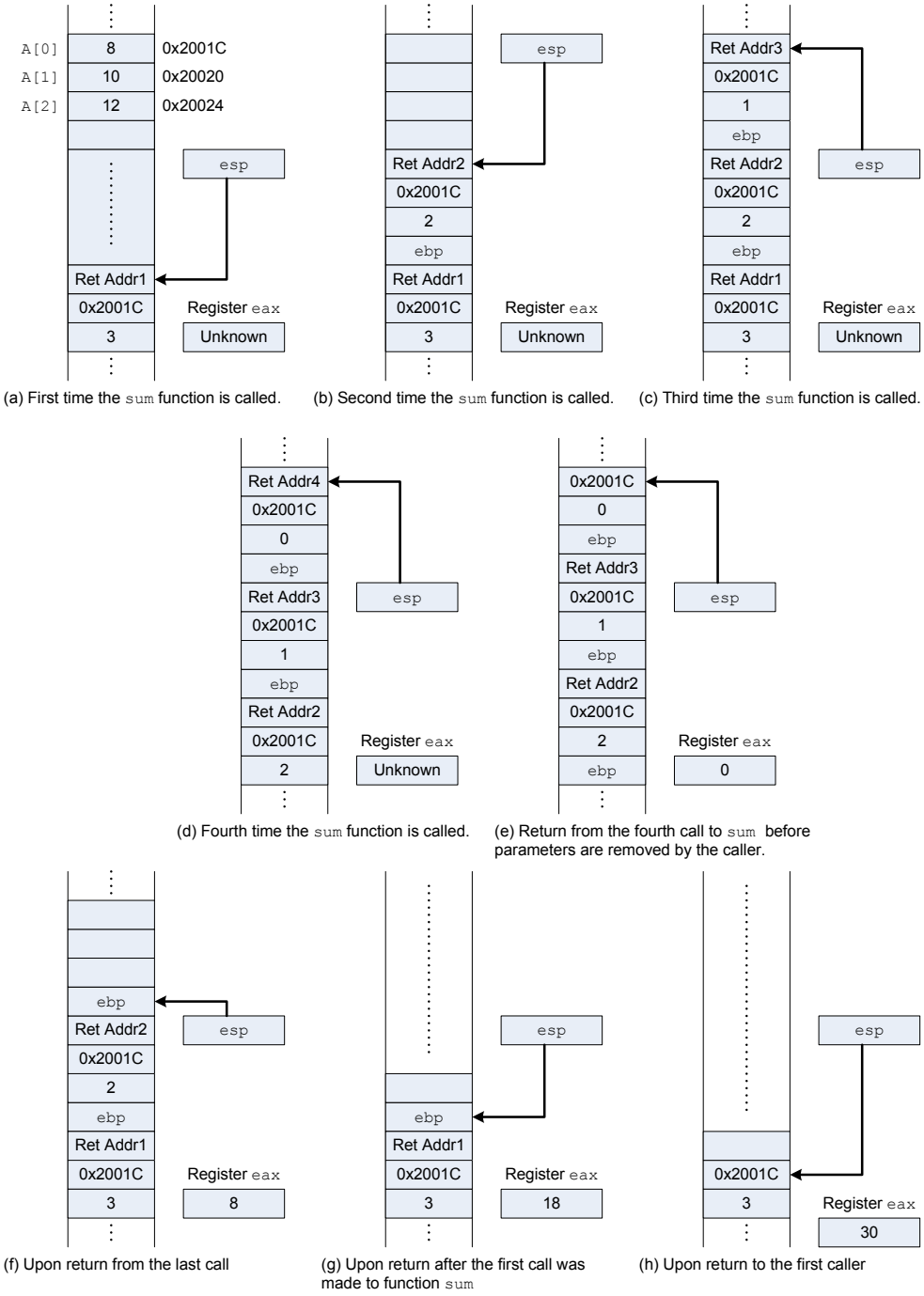


Figure 4.9: Trace of a recursive function

4.6 Interfacing with GNU C compilers

The assembly language programs can be interfaced to C functions. Programs written in C language can call assembly language functions and vice versa. For this we need to understand the basic techniques of parameter passing, parameter referencing, function prologue and epilogue code as used by C functions.

The C interface is similar to the one that is used in the function `sum` described in the previous section. The functions maintain on stack a frame that contains parameters, return address, reference to the older frame and local operands. In this mechanism parameters and return address are put on the stack by the caller (by using `push` and `call` instructions) whereas reference to older frame and space for local variables are maintained by the called function itself.

As mentioned earlier, it is convenient to use frame pointer instead of stack pointer. All C functions compiled using the GNU C compiler use `ebp` register as frame pointer. Thus the prologue code for all C functions looks like the following.

```
pushl  %ebp
movl   %esp, %ebp
subl   $n, %esp
```

Here `n` is the size of the memory used for the local variable storage. In this code, the function saves the old frame pointer on the stack, sets the new frame pointer and creates space for the local variables on the stack. One may notice that if the function does not use any memory for local variables, the last instruction need not be used.

IA32 processors provide an instruction `enter` to achieve the same functionality. The `enter` is very powerful instruction in IA32 processors. However, the C interface uses only one variation of this instruction whose syntax is shown below.

```
enter n,0
```

Constant `n` in `enter` instruction represents the size needed for local storage in bytes. The stack pointer is adjusted by this number to create a memory space that is utilized by the function for storing local variables. There are other variants of this instruction which are not discussed here. These variants are represented by non-zero values of the second argument of the instruction.

Before returning control back to the caller function, the C functions must restore the stack to the same state as it was when the function was called. Therefore, C functions must recover the space for local variables from the stack and must restore the old frame pointer. Since

the frame pointer register ebp was set to the stack pointer before creating space for local variables, recovery of space for local variables can be done by setting stack pointer back to frame pointer. Also at the same time, old frame pointer can be restored by just popping it off the stack. This functionality can be achieved by the following epilogue code executed just before returning control back to the caller.

```
        movl   %ebp, %esp
        popl   %ebp
        ret
```

IA32 processors provide an instruction leave that implements part of this functionality. The leave instruction takes no arguments. Upon execution, it first copies the value of the ebp register to the esp and then pops the old saved frame pointer into register ebp. A function can therefore replace its epilogue code by just two instructions. The syntax of the leave instruction is the following.

```
leave
```

With the use of enter and leave instructions, the entry and exit code of a function will look similar to the following.

```
Function:
   enter 16, 0
   /* 16 bytes of local variable storage
    * is allocated. Parameters are now
    * available at 4(%ebp), 8(%ebp)...
    * Old frame pointer is at (%ebp).
    * Four local variables of size 32 bits
    * are at -4(%ebp), -8(%ebp), -12(%ebp)
    * and -16(%ebp). */
    ...
    Function Body
    ...
    leave
    ret
```

EXERCISES:

4.12 If a function does not take any arguments, does it need to execute the prologue and epilogue codes? In particular does it need to save the register ebp and restore it later?

4.13 Is the use of a separate register (ebp) necessary as frame pointer? Can the stack pointer be used to access the local variables and parameters?

4.6.1 Passing parameters on stack

The parameters are passed to a function by the caller function. The most commonly used technique is to pass them through the stack. We have already seen some aspects of the parameter passing through the stack. There are several issues related to the parameter passing in general. While developing functions in high level languages, the parameters are specified in certain order by the programmer. The order of pushing parameters on the stack before calling the functions can be left to right or right to left. In left to right parameter passing mechanism, the first parameter is pushed first on the stack and the last parameter is pushed last. In right to left parameter passing mechanism, the last parameter is pushed first on the stack while the first parameter is pushed the last. The order of parameter passing is often dependent upon the language run-time support. Some languages and their implementations use one order of parameter passing while some other languages use the other. In C, the parameters are pushed in right to left order. That is the last parameter of the function call is pushed on the stack first.

This order of parameter passing has an advantage. C functions can be written with variable number of arguments. A notable such function is `printf` where the count of parameters can be any number more than 1. The first parameter often describes the number of other parameters. For example, in case of `printf` function, the first string parameter defines the number of parameters by the number of format specifiers (for example, "`%d`") in the string. While implementing functions with variable number of arguments, the location of the first parameter must be known on the stack. This is possible only if the first parameter is pushed at the end such that it is always on top of the stack.

As parameters are pushed by the caller, the removal of the parameters can be done either by the called function or by the caller. The actual technique of removal of parameters depends upon the high level language and its implementation. The GNU C compilers generate code in such a way that the parameters are removed by the caller. The parameter removal from the stack effectively amounts to adjusting the stack pointer in such a way that it points to the memory location prior to pushing the parameters. The same can be done either by executing a series of `pop` instructions or by just adding the size of all parameters (in bytes) to stack pointer `esp`. Removal of parameters by the caller instead of by the called function has several advantages as well as disadvantages. As the number of parameters passed to the called function is known to the caller function, it is easy for it to remove them from the stack. If the called function was to remove variable number of parameters, it will have to first find the number of parameters. Often it is not possible. Consider the following example of calling `printf` function.

```
printf("%d %d\n", x, y, z);
```

In this function call, the format string is used to print two integers. However in the call three integers `x`, `y` and `z` are passed. Thus in this call to `printf`, a total of four arguments are passed to the called function. This fact is known at the place where the function was called. A rough code for that is given below.

```
      :
      /* Push parameters */
      pushl z
      pushl y
      pushl x
      pushl $fmtstr
      call  printf
      /* Remove parameters */
      addl  $16, %esp
      :
fmtstr: .string "%d %d\n"
```

An interesting question arises in this example. Will the function `printf` know the number of parameters pushed on the stack? In fact, the best the `printf` function can do is to analyze the first parameter and count the format specifiers (in this case 2). Thus the `printf` can remove three parameters, including the format string. This parameter removal scheme will therefore result in inconsistency in stack. It is for this reason that the responsibility of removal of parameters from stack is left to the caller in C implementations.

However the major disadvantage of this scheme of parameter removal is that the code for parameter removal is inserted for each call to the function. In a typical C program, there can be several calls to the `printf` function. If the code for parameter removal is to be put for each call, those many extra instructions are needed. This results in an increased size of the object code. If the parameters are to be removed by the called function, it can be achieved by the `ret` instruction itself. It is only fair to say that this disadvantage is very small compared to the consistency that is achieved on the stack.

Some versions of C compilers support both kinds of parameter passing and removal. In the standard C parameter passing mechanism, the parameters are pushed from right to left order and they are removed by the caller. The other parameter passing mechanism is often called PASCAL parameter passing mechanism because it was used in PASCAL programming language. In this mechanism, the parameters are pushed in the left to right order and removed by the called function. In fact, the parameter passing scheme for PASCAL is more complex than just this. The description of this scheme is out of scope of this book.

Often C compilers support additional specifications of parameter passing mechanism using certain additional keywords. For example, Microsoft Visual C++ compiler supports a keyword `__stdcall` to indicate that the function must use a standard Windows mechanism of parameter passing.

4.6.2 Return values of functions

When a function is called by the caller, it must return a value even though some function values may not be used by the caller. The question is how to return values to the caller functions. Any of the parameter passing mechanism, as discussed in section 4.5, can be used to return values to the caller. C uses registers to return values back to the caller. In GNU C compiler for the IA32 processors, the non-floating point values are returned in register `eax` or in registers `eax` and `edx` depending upon whether size of the data type of the value is up to 32-bits or larger than 32 bits. Values for the data types less than or equal to 32-bits in size are returned in the `eax` register only. For example, the C functions returning `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, pointers or similar data types return their values in register `eax`. The data types whose size is more than 32-bits, are returned in registers `edx` and `eax`. In these cases, register `edx` contains the higher 32-bits of the return value while register `eax` contains the lower 32-bits of the return value.

The convention of return value for the floating point values is different than that of the non-floating point values. We shall see this later while discussing the floating point instructions.

4.7 An example to interface with C functions

We will see an example in which the Assembly language programs call the C functions and vice versa. We write one function `exchangeNums` in assembly which takes addresses of two 32-bit data items as parameters and exchanges their values. The function also prints the values after exchanging.

```
.globl exchangeNums
exchangeNums:
        /* In this program, the following prologue
        code can be avoided. It is included here
        so that it shows a typical interface to C */

        push  %ebp
        mov   %esp, %ebp
```

```
        /* Stack layout is as follows.
        (%ebp) contains old value of ebp
        4(%ebp) contains the return address
        8(%ebp) contains addr of the first parameter
        12(%ebp) contains addr of the second parameter */

        mov   8(%ebp), %eax
        mov   12(%ebp), %ebx
        xchg  (%eax), %ecx
        xchg  (%ebx), %ecx
        xchg  (%eax), %ecx

        /* Print the two values. Arguments
        to printf are pushed in reverse
        order */

        pushl (%ebx) /* Value of second operand */
        pushl (%eax) /* Value of first operand */
        pushl $FmtStr /* Format String for printf */
        call  printf

        /* Remove parameters of printf */

        add   $12, %esp

        /* Epilogue code */

        mov   %ebp, %esp
        pop   %ebp
        ret
FmtStr: .string "NumA: %d, NumB: %d\n"
```

The C program that calls this assembly function is as following.

```
  #include <stdio.h>
  int main (void) {
      int a, b;
      printf("Enter two numbers :");
      scanf("%d %d", &a, &b);
      /* Call the Assembly language function */
      exchangeNums(&a, &b);
      printf("Exchanged numbers are %d and %d\n", a, b);
      return 0;
  }
```

In the assembly program, the first line `.globl exchangeNums` is used for the first time. This line is used to pass some information to the linker and is not used to generate any code. In particular, this line informs the linker that symbol `exchangeNums` is a global symbol and other programs can refer to this symbol. Thus the linker automatically adjusts call to `exchangeNums` function (in C program) to refer to the assembly language function.

An additional point is that the comments written in the assembly language programs are handled differently with different versions of the `as`. Some assemblers treat text within `/*` and `*/` as comments while some others treat the `/*` as start of comment which ends at end of line. The simplest way is to use C pre-processor to handle comments. The C pre-processor (`cpp`) then removes these comments before passing the output to `as` for processing. GCC uses a file naming convention for this. The following extensions are relevant to us.

- `.S` Assembly language program that must be processed first by `cpp` before being assembled by `as`.
- `.s` Assembly language program that can be processed by `as` directly.
- `.c` C programs that are first processed by `cpp`

Our example programs can be compiled to run. In order to do so, save the Assembly language program in a file, say `exch.S`. Save the C program in another file, say `drive.c`. We will use the `cc` command (in GNU/Linux distributions, `cc` command is same as `gcc`) to compile the code. This command handles files with `.S` extension by first pre-processing and then assembling the program using `as` assembler. The program can be compiled using the following single command.

```
cc exch.S drive.c -o example
```

This command generates an executable file whose name is given as `example`.

EXERCISES:

4.14 Write an Assembly language program that can be interfaced to the C functions. The Assembly language program should take two numbers as parameters and must provide the largest of them as the return value. Using this function write a C function that reads three numbers from the input and prints them in the decreasing order. Compile the programs and test them out with various inputs.

4.15 Write an Assembly language function that takes the address of a string as parameter and checks if it contains a valid octal number or not. The return value of the function is 0 if the string is not a valid octal string. It is 1 if the string is a valid octal string. The string is terminated by a NULL character (the byte contains a value 0). The string will be valid if it contains characters from '0' to '7' only. For simplicity, you need not handle white spaces differently. Thus if the string contains any character

outside the range '0' to '7', the string will not be considered a valid one. Write a C program that reads strings from standard input and uses the Assembly language function to check if it is a valid octal string or not. The C program terminates only when it finds a valid octal string.

Chapter 5

Arithmetic and Logic instructions

IA32 architectures have a very rich instruction set for performing arithmetic and logic operations. In this chapter, we shall discuss these operations. It will be possible to execute programs discussed in this chapter by interfacing with the C library.

In IA32 architectures, the numbers are assumed to be in 2's complement representation for most instructions. However, the architecture provides support for arithmetic operations on unsigned numbers, 2's complement signed numbers and binary coded decimal numbers. We shall see the operations on binary coded decimal numbers later in this chapter.

5.1 Arithmetic operations

5.1.1 Addition and subtraction of integers

IA32 architectures provide instructions to add, subtract, multiply and divide two numbers. If the signed numbers are represented using 2's complement number representation, additions and subtractions can be performed in the same way regardless of the numbers being signed or unsigned. Two instructions that are provided in the IA32 architectures for addition and subtraction are the following.

```
add src, dest
sub src, dest
```

These instructions take two parameters. The `add` instruction adds the value provided by the `src` operand into the value of the `dest` operand. The `sub` instruction subtracts value provided by the `src`

operand from value stored in `dest` operand. The `dest` operand gets modified in both of these instructions. These instructions can be used for 8-bits, 16-bits or 32-bits immediate, register or memory operands. As the `dest` operand gets modified during the execution, it can not be an immediate operand. In addition to this, like other instructions in IA32 architectures, these instructions can have a maximum of one operand in memory. The `add` and `sub` instructions also modify flags in the `eflags` register. These flags can then be used in conjunction with other instructions to build arithmetic with higher precision.

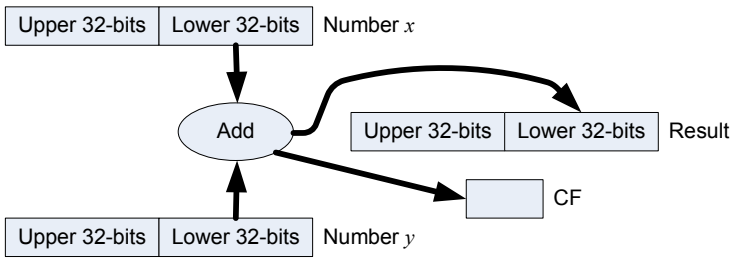
IA32 architectures provide `adc` and `sbb` instructions to support high precision arithmetic involving multiple bytes. These instructions are used in the same way as the `add` and `sub` instructions. The `adc` instruction adds the value stored in the source operand and the value of the carry flag CF into the destination operand. Similar to the `add` instruction, this instruction also sets the flags that can be used for further operations on the numbers.

The `sbb` instruction subtracts the value of the source operand and the value of the carry flag (CF) from the destination operand. While subtracting, if there is a borrow, the carry flag is set to 1. All other flags are modified as per the value of the operand. Syntax of these two instructions is as follows.

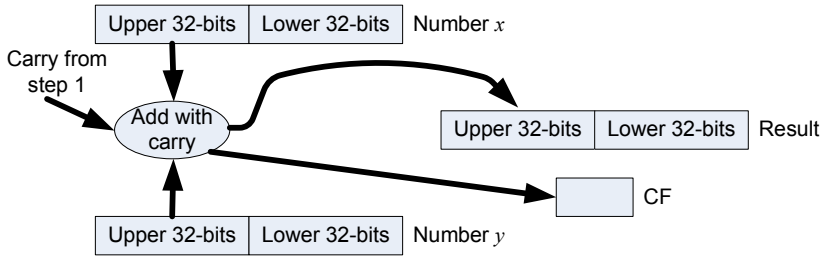
```
adc src, dest
sbb src, dest
```

An example of adding two 64-bit numbers is shown in figure 5.1. First the lower 32-bits of two numbers x and y are added using the `add` instruction. The resultant 32-bit numbers provide the lower 32-bits of the result. A carry out of this addition gets set in the CF due to the execution of the `add` instruction. Later this carry and upper 32-bits of two numbers are added together to provide the upper 32-bits of the result. For this the `adc` instruction is used which also affects the carry flag setting to indicate the carry out of the entire 64-bit addition.

To implement high precision arithmetic, the `add` and `sub` instructions can be used along with carry flag and `adc/sbb` instructions. We discuss this with an example of building arbitrary precision addition. We assume that arbitrary precision numbers are stored in memory in the least-significant-byte-first order. The function that we wish to write can be called from C environment. The function has three arguments. The first argument provides the length of the arbitrary precision numbers in bytes. The second argument is the address of the source operand and the third argument is the address of the destination operand. After execution, destination operand is modified to contain the value obtained by addition of original values of source and destination operands.



Step 1: Adding lower 32-bits of the numbers. Lower 32-bits of result and carry of addition are saved after `add` instruction.



Step 2: Adding upper 32-bits of the numbers along with previous carry. Upper 32-bits of result and carry of addition are saved after `adc` instruction.

Figure 5.1: Addition of 64-bit integers using 32-bit operations

```
.globl addprecision
addprecision:
     /* First operand is at 4(%esp) - size of numbers
      * Second operand is at 8(%esp)
      * Third operand is at 12(%esp)
      * The size of numbers is assumed to be at least
      * one byte */
     mov   4(%esp), %ecx  /* Size of numbers */
     mov   8(%esp), %ebx  /* Address of source */
     mov   12(%esp), %edx /* Addr of dest */
     clc                  /* Initially set CF=0 */
.L0:
     mov   (%ebx), %al
     adc   %al, (%edx)
     inc   %ebx         /* Increment value in ebx */
     inc   %edx
     loop  .L0
     ret
```

Some observations can be made in this program. First of all, the program does not follow norms of building stack frame. This is possible because of two reasons. First there are no local variables in this function. Second this function does not use recursion. Here parameters are copied to the registers using the stack pointer register as the base register. In general, in carefully written Assembly language programs, additional code to build and later recover the stack frame can be avoided.

The program will execute at least one iteration of the loop. Therefore the size of the numbers will be at least one byte. Initially the carry flag is set to 0 by `clc` instruction[1]. In the subsequent iterations, the `adc` instruction uses carry flag of the last execution of same instruction. It is possible as no instruction in the loop other than `adc` modify carry flag. Otherwise, it would have been necessary to save the flags by pushing and later popping them from the stack. Registers `ebx` and `edx` store the addresses in memory of specific bytes of source and destination numbers. The number of times the loop is carried out is same as the size of the numbers in bytes.

Two instructions `inc` and `dec` in IA32 architectures are used to increment or decrement their argument by one. These instructions take only one argument and it can not be specified using immediate addressing. If the argument is in memory, size of the argument (byte, word or long) can be specified by adding an appropriate suffix (`b`, `w` or

[1]Two more carry flags modifying instructions `stc` and `cmc` in IA32 processor instruction set are used to set CF=1 and to invert CF respectively

l) to the instruction. These instructions modify all flags except the CF. Since the CF does not get modified, we could use it in the program without needing to save the flags explicitly.

IA32 architecture also provides an instruction neg to negate the operand. It takes only one argument. The argument of this instruction gets modified to a value that is the negative of the original value. As the argument modifies, it can not be an immediate operand. Further if the argument is in the memory, the size must be specified using b, w or l suffix to the neg instruction. The neg instruction modifies all flags.

```
inc dest
dec dest
neg dest
```

Another very useful instruction in IA32 architectures is cmp instruction. This instruction is similar to the sub instruction except that the destination is not modified. Only the flags get modified because of the execution of this instruction. This instruction compares the two arguments and when combined with the conditional jump instructions, it can be used to implement powerful loop constructions. The syntax of the cmp instruction is as follows.

```
cmp src, dest
```

The dest argument to this instruction can not be an immediate operand and at most one argument can be specified using memory addressing.

EXERCISES:

5.1 If in program addprecision, inc instructions are replaced by the add instruction (i.e. inc %ebx is replaced by add $1, %ebx and inc %edx is replaced by add $1, %edx), what will be the impact on the program?

5.2 Introduce appropriate corrections in the program such that add instruction can be used in place of inc instruction.

5.3 Replace the loop instruction by other instructions such that the program works in the same manner.

5.4 Write an assembly language program to compute the sum of an array of integers whose size and starting address are passed as first and second parameters respectively. Write a C interface to this and test it out using C functions.

5.5 Write an assembly language program that takes an array and the number of elements in the array as arguments. Each element of the array contains an address of an integer. The assembly language function when executed should increment each integer by one. Test the function using C interface by doing input output in C.

5.6 Write an assembly language function to compute $A[0] - A[1] + A[2] - A[3] + \cdots - A[n-1]$ for a given array A of size n elements. The function takes two arguments, starting address of array and count of the number of elements (n) in the array. Function returns the computed value. Use `neg` instruction to perform this computation. Assume that the array has even number of elements.

5.1.2 Multiplication and division of integers

In 2's complement representation of integers, addition and subtraction operations can be done in the same way regardless of the numbers being signed or unsigned. IA32 architectures therefore do not provide separate instructions to perform signed and unsigned arithmetic. However, the same is not true for multiplication and division. The signed and unsigned numbers are treated differently while multiplying or dividing. Separate instructions are provided in IA32 architectures for signed and unsigned multiplication and division.

There are a total of four multiplication and division instructions in the IA32 architectures. These instructions can be used for unsigned multiplication (`mul`), signed multiplication (`imul`), unsigned division (`div`) and signed division (`idiv`). The `imul` instruction can be used in one operand, two-operand or three-operand forms while the other instructions can be used only in one operand form.

The syntax of the `imul` instruction is as follows.

```
imul src
imul src, dstreg
imul imm, src, dstreg
```

In the first form of the `imul` instruction, signed multiplication is carried out between the `src` operand and one of the implied registers. The results are returned in the implied registers. The `src` operand can be one of the registers or memory locations. This form of the instruction does not support immediate mode of addressing. If the `src` is an 8-bit operand, the implied source and destination registers are taken as `al` and `ax` registers respectively. Thus two 8-bit numbers (one provided in the `src` and the other one provided in register `al`) are multiplied using the signed multiplication algorithm. The 16-bit result is put in register `ax`. If the `src` is a 16-bit operand, it is multiplied with register `ax` and the 32-bit result is put in registers `dx` and `ax`. The `dx` register contains the higher order 16-bits of the result. Thus even if the result would fit in just 16-bits, register `dx` always gets modified in this form of the instruction. When `src` is a 32-bit register, the second operand is implied in the register `eax` and the resulting 64-bit result is provided in registers `edx` and `eax`. Register `edx` gets the higher order 32-bits of the result. The first form of the `imul` instruction implements the

expanding multiplication. In these cases, the result is always twice as wide as the input operands.

In the second form of the `imul` instruction, two operands are specified in the instruction. In this form of the instruction, the result is of the same size as that of the input operands. Two input operands must be of the same size. The `src` operand can be register, memory or an immediate constant. The second operand for the signed multiplication can only be a register specified using `dstreg` operand. The result of the multiplication is put back in the register `dstreg`. In this form of multiplication, if result of the multiplication is more than the size of register `dstreg`, the result will be incorrect. Such a condition is represented by the overflow flag (`OF`) being set to 1. The overflow flag is set to 0 otherwise.

In the third form of the `imul` instruction, one operand is specified as an immediate constant. The size of multiplication is known by the second and the third operands. The second operand can be a register or a memory location but not an immediate constant. The third operand represents the destination of the result, which can only be a register. Source and destination operands in this form of the instruction are of the same size. The overflow flag at the end of the execution carries the status of the overflow in computation. Immediate constant in this instruction is sign extended to as many bits as required by the size of the other operands.

Some second and third forms of the instructions may be equivalent. For example, `imul $10, %eax, %eax` and `imul $10, %eax` are equivalent instructions both of which mean to multiply register `eax` by 10. In fact, in this particular example, machine code for two instructions will be identical.

Some examples of the usage of `imul` instructions are the following.

- `imul %dx`. This instruction is of the first form. Upon execution of this instruction, 16-bit contents of registers `dx` and `ax` are multiplied and the resulting 32-bit number is put in registers `dx` and `ax` (register `dx` carrying the upper 16 bits of the result).

- `imul %ax, %ax`. This instruction is of the second form. Upon execution of this instruction, the value in register `ax` is multiplied by itself and the resulting 16-bit number is put back in register `ax`. This is a 16-bit multiplication operation and the result is also 16-bits.

- `imul $10, (%esp), %eax`. This instruction is of the third form. Upon execution of this instruction, a 32-bit number at the top of the stack is multiplied by 10 and the resulting 32-bit number is put in register `eax`.

The `mul` instruction also behaves in a similar manner as the first

form of the `imul` instruction. It implements unsigned multiplication. Syntax of the `mul` instruction is as follows.

```
mul src
```

The `src` operand can be a memory location or a register but not an immediate constant. If the `src` is an eight bit register or memory location, it is multiplied using unsigned multiplication algorithm to register `al` and the 16-bit result is put in register `ax`. If the `src` is a 16-bit operand, it is multiplied with register `ax` and the 32-bit result is put in registers `dx` and `ax` with the register `dx` containing the higher order 16 bits of the result. If the `src` is a 32-bit operand, register `eax` is taken as the second number and 64-bit result is put in registers `edx` and `eax` with register `edx` containing the higher order 32-bits. The overflow flag is set to 1 if results could not be fitted in the same size as of the instruction. All other flags are also modified.

The `idiv` and `div` instructions also come only in one operand form. These instructions take the divisor as operand and use the implied registers to implement signed and unsigned divisions respectively. The instructions return both, the quotient and the remainder of the division. In case, the divisor is zero or too small such that the quotient will not fit in the implied register, these instructions generate a division exception. The exceptions are like interrupts which are handled by the Linux kernel internally. In response to these, the Linux kernel prints out a message like `Floating exception (core dumped)` and generates a file called `core` containing the memory image and other information. This file can then be used later to debug the program and to identify the instruction that caused the exception. The syntax of the division instructions is as the following.

```
idiv src
div src
```

The `src` in division instructions can not be an immediate constant. The behavior of division instructions is shown in the following table.

Size of explicit operand	Location of implied operand	Result	
		quotient in	Remainder in
8 bit	`ax`	`al`	`ah`
16 bit	`dx`||`ax`	`ax`	`dx`
32 bit	`edx`||`eax`	`eax`	`edx`

If the size of the explicit operand `src` is 8 bits, the 16-bit value in register `ax` is divided by the `src` and the resulting 8-bit quotient and remainder are returned in registers `al` and `ah` respectively. If the

size of the explicit operand `src` is 16 bits, the implied 32-bit dividend value is taken from the 16-bit registers `dx` and `ax` with higher 16-bits being in register `dx`. In this case, 16-bit quotient and remainder are returned in registers `ax` and `dx` respectively. Similarly, if the size of the `src` is 32-bits, registers `edx` and `eax` are used as the dividend (higher order 32-bits are in register `edx`) and the quotient and remainder are returned in registers `eax` and `edx` respectively. When the quotient is large and can not fit in the corresponding register, overflow flag is set and the value of the result is left undefined.

The signed number division (`idiv` instruction) can be used for negative and positive numbers. In case of division operations involving negative numbers, the remainder and quotient require a little clarification. In this case, the absolute value of the remainder is always between 0 and the absolute value of the `src`. However, the sign of the remainder is same as the sign of the divisor. The following identity is always satisfied.

$$\text{Dividend} = \text{Divisor} * \text{Quotient} + \text{Remainder}.$$

Therefore the quotient is positive if divisor and dividend have same sign and negative otherwise. The following table shows the sign relationship of `idiv` instruction.

sign of			
Dividend	Divisor	Quotient	Remainder
+	+	+	+
+	−	−	−
−	+	−	+
−	−	+	−

All of these instructions modify flags. In case of the division instructions, all flags have undefined values.

EXERCISES:

5.7 Write an assembly language function that takes dividend and divisor as two arguments and performs signed division. The arguments are 32-bits in size. The function is required to return just the quotient. Test the function with C driver function that reads the numbers from stdin and prints out the quotient on the stdout. Test the assembly language function for a variety of inputs including divisor being 0.

5.8 Using 32-bit unsigned multiplication instructions and add instructions, implement a 64-bit unsigned multiplication algorithm. Develop an assembly language function that takes three addresses in the memory. The first two addresses provide the address of input arguments (each 64-bits) and the third address provides the address of memory buffer (64-bits) to store the result. In case of an overflow, print out errors. (Hint: Treat 64-bit numbers as juxtaposition of two 32-bit numbers. Let's denote the

inputs as (a, b) and (c, d) where each of a, b, c and d are 32-bits in size. You will need to use expanding multiplication to get 64-bit results for each 32-bit number pairs. Figure out why?)

5.2 Binary Coded Decimal numbers

The decimal digits are between 0 and 9. Each of these can be represented in four bits. Thus the decimal numbers can be represented either in the binary form or in the binary coded decimal (BCD) form. In the BCD from, each group of four bits represent one decimal digit. For example, a 5 digit decimal number can be represented in 20 bits. In certain cases, this kind of number representation makes handling of data easier. IA32 architectures provide a variety of instructions to handle arithmetic for the BCD numbers. Before we go in details we need to understand a little more about the BCD numbers.

The BCD representation of a decimal number is obtained by taking each of its digits and coding them in 4 bits. For example, the BCD representation of decimal number 472 is 0100 0111 0010, or 0x472. As opposed to this, the binary representation of the same number is 0001 1101 1000, or 0x1D8. Thus the BCD representation of a number can be taken as a hexadecimal string comprised of the digits of the decimal number. For example, the BCD representation of 28 is 0x28, or 0010 1000. Certain patterns do not make sense as a BCD representation. For example, 0xE5 is not a valid BCD number.

EXERCISES:

5.9 Give BCD representations for the following decimal numbers.
(i) 74 (ii) 93 (iii) 88

5.10 Give BCD representation for a number whose binary representation is 0010 1100, or 0x2C.

5.11 Which of the following numbers can not represent BCD numbers (or are invalid bit strings as BCD numbers)?
0x25, 0x4D, 0x36, 0x74, 0xE2.

5.2.1 Operations for the BCD arithmetic

In order to add two digits coded using BCD representation and keeping the result in the BCD form, certain additional operations are necessary. The sum of two binary coded digits can be a number between 0 to 18, or 0x00 to 0x12 as the input digits are between 0 and 9. We can choose to keep the result in one digit BCD representation. In this case, carry flag can be used to represent overflow condition. For example, when the sum is between 0 to 9, carry flag is set to 0 and the resultant BCD

digit is the sum itself. However, when the sum is between 10 and 18, carry flag is set to 1 and the resultant BCD digit should be between 0 and 8 obtained by subtracting 10 from the sum. With the use of regular `add` instruction, we will have the sum as 0x00 to 0x12. IA32 processors also provide an auxiliary carry (AF) flag that contains the carry out of the last four bits. Thus using the regular addition of two BCD digits, the last four bits of the sum will be 0x0 to 0xF with AF=0, or 0x0 to 0x2 with AF=1; the later being the case when the sum is between 0x10 to 0x12. We can then perform certain other operations to get the desired results. It is given in the following table.

Regular addition		Desired	
Sum (4 bits)	AF	Sum	AF
0x0 - 0x9	0	0x0 - 0x9	0
0xA - 0xF	0	0x0 - 0x5	1
0x0 - 0x2	1	0x6 - 0x8	1

The algorithm for converting the sum obtained by regular addition of two BCD digits to the sum in BCD representation can be written as follows.

```
if (four bits of sum within 0 to 9 and AF=0)
   No adjustment is necessary
if (four bits of sum within 0xA to 0xF and AF=0)
   add 6 to the sum; //AF gets set automatically
if (four bits of sum within 0 to 2 and AF=1)
   add 6 to the sum; Set AF=1;
```

IA32 architectures provide an instruction `daa` , or decimal adjust after addition that implements the algorithm as given above. The `daa` instruction performs this algorithm for a group of two digits given in eight bits of register `al` (presumably obtained by `add` instruction). We assume that two BCD numbers, each up to two digits, are added using regular `add` instruction and then the sum is adjusted so that it gets two BCD digits. If there is a carry out (of decimal addition) then the carry flag is set to 1. The `daa` instruction does not take any operand and adjusts the number in `al` register. While performing the add operation by previously executed `add` instruction, AF and CF flags would have been set appropriately. The `daa` instruction uses these flags in its operations. The algorithm as given in instruction set reference for the IA32 processors is reproduced below.

```
1 if (((al & 0x0F) > 9) or (AF = 1)) { // lower digit > 9
2   al = al + 6;
3   CF = CF | carry of last addition (add instruction);
4   AF = 1 // There is a decimal carry
5 }
```

```
6 if (((al & 0xF0) > 0x90) or (CF = 1)) { //upper digit > 9
7   al = al + 0x60;
8   CF = 1;
9 }
```

In this algorithm, it is assumed that in lines 2 and 7 when a number is added to the `al` register, AF or CF do not get modified. Thus at the end of this algorithm, AF denotes carry out of the least significant decimal digit while CF denotes carry out of the second decimal digit.

`daa` instruction can be used along with `add` and `adc` instructions to build multi-digit BCD addition. For example, the following code adds two 4-digit BCD numbers given in registers `bx` and `cx` respectively and returns the four digit result in register `dx`. The carry flag is set if there is a carry out of the summation.

```
    mov  %cl, %al
    add  %bl, %al
    daa
    mov  %al, %dl // CF is not modified
    mov  %ch, %al // by the mov instruction
    adc  %bh, %al
    daa
    mov  %al, %dh
```

EXERCISES:

5.12 Using the code given above, write an Assembly language function that adds two 32-bit numbers given as arguments on the stack. Two arguments each represent 8-digit numbers in BCD representation. The function should return the sum as output. The carry of the final summation need not be returned. Using C interface, test this assembly language function for a variety of inputs.

IA32 architectures also provide a similar instruction for decimal adjusting after subtraction. This instruction called `das` has the following algorithm as given in instruction set reference manual for the IA32 architectures.

```
if (((al & 0x0F) > 9) or (AF = 1)) {
  al = al - 6;
  CF = CF | borrow of last subtraction (sub instruction);
  AF = 1
}
if ((al > 0x9F) or (CF = 1)) {
  al = al - 0x60;
  CF = 1;
}
```

EXERCISES:

5.13 Verify the algorithm for `das` instruction with decimal subtraction.

5.14 Build multi-digit subtraction algorithm for BCD representation using the `sub`, `sbb` and `das` instructions.

IA32 processors also support unpacked BCD arithmetic. In the unpacked BCD representation, an 8-bit register contains one decimal digit rather than two. Thus the most significant four bits of the register will always be 0 while the lower order four bits represent the decimal digit between 0 and 9.

There are four instructions that can be used to handle unpacked BCD numbers.

```
aaa
aas
aam
aad
```

The instruction ASCII adjust after addition, or `aaa`, is used to adjust register `al` such that it contains a BCD digit. The instruction also assumes that next BCD digit is stored in register `ah` and it adds one to it if after adjustment register `al` results in a decimal carry. It is assumed that register `al` contains the result of a previous addition of the unpacked BCD digits. Thus the algorithm implemented by the `aaa` instruction is similar to the following.

```
if (((al & 0x0F) > 9) or (AF = 1)) {
  al = al + 6;
  ah = ah + 1;
  CF = 1;
  AF = 1
} else {
  CF = 0; AF = 0;
}
al = al & 0x0F;
```

In a similar manner, instruction ASCII adjust after subtraction, or `aas`, is used to adjust the contents of register `al` for BCD. The next higher digit is expected to be in register `ah` and it is adjusted in case of a borrow.

The `aad` instruction is used to convert a binary coded unpacked decimal digits in registers `al` and `ah` to a binary number in register `al`. After execution of this instruction, register `al` contains the value of `ah` register multiplied by 10 and added to the old value of register

al. Register ah is then set to 0 such that register ax contains same numeric value as register al.

As opposed to this, aam instruction converts an 8-bit value in register al to its unpacked BCD equivalent in registers ah and al respectively. The ah register contains the higher order digit while register al contains the least significant digit.

The aad and aam instructions can be used to first convert BCD numbers to binary and then perform division or multiplication. The results then can be converted back to unpacked BCD.

EXERCISES:

5.15 Write an Assembly language function that takes addresses of two ASCII strings of five decimal digits each as input and performs the summation of two numbers. Five digits of the result are returned in the first string.

(Hint: An ASCII character for a decimal digit stored in register al can be converted to the unpacked BCD digit by instruction '`and $0x0F, %al`'. Similarly a BCD digit stored in register al can be converted to its equivalent ASCII character by instruction '`or $0x30, %al`'.

In this exercise, the addresses are given for the most significant byte of the numbers. For the addition, one need to start with the least significant digit. Always keep the next significant digit in register ah so that BCD arithmetic instructions can adjust the next digit automatically.)

5.3 Logic, Shift and Rotate operations

5.3.1 Logic instructions

IA32 architectures support various bit-wise logic operations. These operations can be used to selectively set, reset or invert a few bits in one of the registers or memory locations. In many software systems, these instructions can be used to implement bit-wise data structures. There are following four instructions in IA32 architectures for implementing logic operations.

```
and src, dest
or src, dest
xor src, dest
not dest
```

These instructions can be used for all data types, i.e. byte, word or long. The operands can be in memory or in registers with a restriction that at most one operand of the instruction can be a memory operand. The src operand can even be an immediate constant.

The and instruction performs bit-wise logical and of two operands. Thus the zeroth bit of the destination is obtained by performing an AND

operation on the zeroth bits of `src` and `dest`. Similarly ith bit of the destination is obtained by performing an AND of ith bits of `src` and `dest` operands.

The `and` instruction can therefore be used to selectively reset the bits of the destination operand to 0. In order to do so, a mask pattern can be made corresponding to the bits that are to be set of 0. In the mask pattern bit i is set to 1 if ith bit of the destination is not to be changed. The mask bit is set to 0 if the corresponding bit of the destination is required to be reset to 0.

In a similar manner, the `or` instruction can be used to selectively set bits of the destination operand to 1. A bit set to 1 in the mask pattern can be used to set the corresponding bit in the destination to 1. A bit set to 0 in the mask pattern does not modify corresponding bit of the destination.

The `xor` instruction can be used to selectively invert the bits of the destination. A bit set to 1 in the mask results in the inversion of corresponding bit of destination by performing bit-wise xor operation.

The `not` instruction inverts all bits of its single operand. Thus a 0 becomes a 1 and a 1 becomes a 0.

We have used some of these instructions in earlier examples. In exercise 5.15, these instructions were used to convert an unpacked BCD digit to its ASCII equivalent and vice-versa. The ASCII equivalents for digits 0 to 9 have a value 0x30 to 0x39 in that order. Thus in order to convert an unpacked BCD digit, an OR operation can be performed between the unpacked BCD digit and 0x30. The same effect in this case can also be obtained by adding 0x30 to the unpacked BCD digit.

In a similar manner the reverse can be obtained by turning higher order 4 bits of ASCII code to 0 (using `and` instruction) or by subtracting 0x30 from it. In the exercise we chose to perform AND operation of ASCII coded byte with 0x0F.

5.3.2 Shift and Rotate instructions

The shift instructions are primarily for multiplication or division by a number that is of the form 2^x. There are two kinds of shift instructions. One kind of instructions handle the signed numbers and other kind of instructions handle the unsigned numbers. Shift operations for signed numbers are also known as arithmetic shifts while the shift operations for unsigned numbers are known as logical shifts. IA32 architectures provide the following instructions to perform both kind of shifts, left and right, in multiple bits.

```
sar count, dest
shr count, dest
sal count, dest
shl count, dest
```

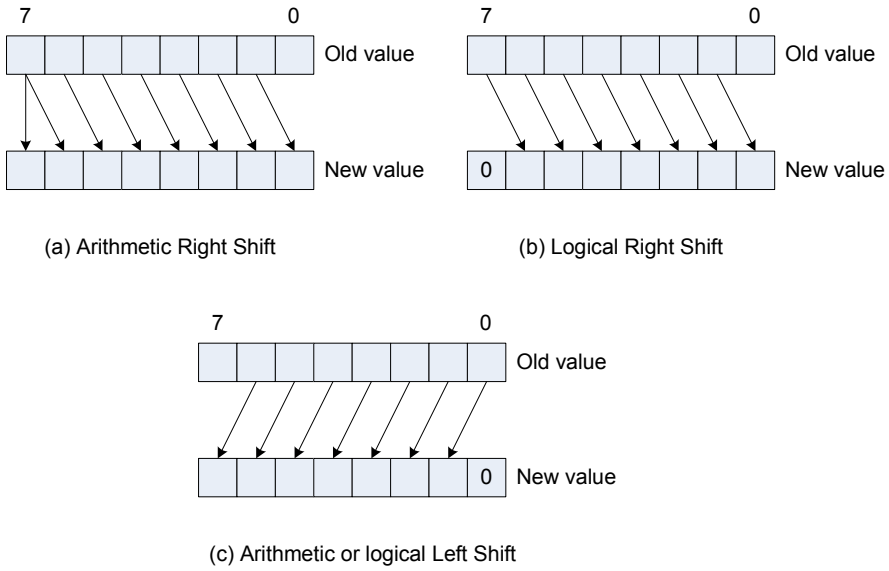


Figure 5.2: Various Shift Instruction

The `count` argument in each of these instructions can be specified by using an immediate constant or by using `cl` register. Maximum value of count should not exceed the number of bits in `dest`. A value of count more than the number of bits in the `dest` results in constant values (0 or -1 depending upon the type of shift and original value of `dest`). IA32 architectures support only up to 8-bit value for the count and therefore the assembler gives an error if the value can not fit in 8-bits.

The `sar` instruction causes the right shift assuming signed number representation. During the shift the sign bit is not changed thereby maintaining a positive number positive and a negative number negative after the shift. This instruction amounts to a division by 2^{count}. However, for a negative number, the division result saturates to -1 or all bits set to 1. Thus if a number, say -5 is shifted by 4 bits to the right, the result of the shift is -1.

Figure 5.2(a) shows the right arithmetic shift on an 8-bit operand. After the shift, the most significant bit retains its old value thereby not changing the sign.

The `shr` instruction implements the logical right shift as shown in figure 5.2(b).

The left shift for signed and unsigned numbers are identical operations. In the assembly language of IA32 architectures two separate mnemonics are provided for consistency for the same instruction. Thus `shl` and `sal` instructions both implement the same operation

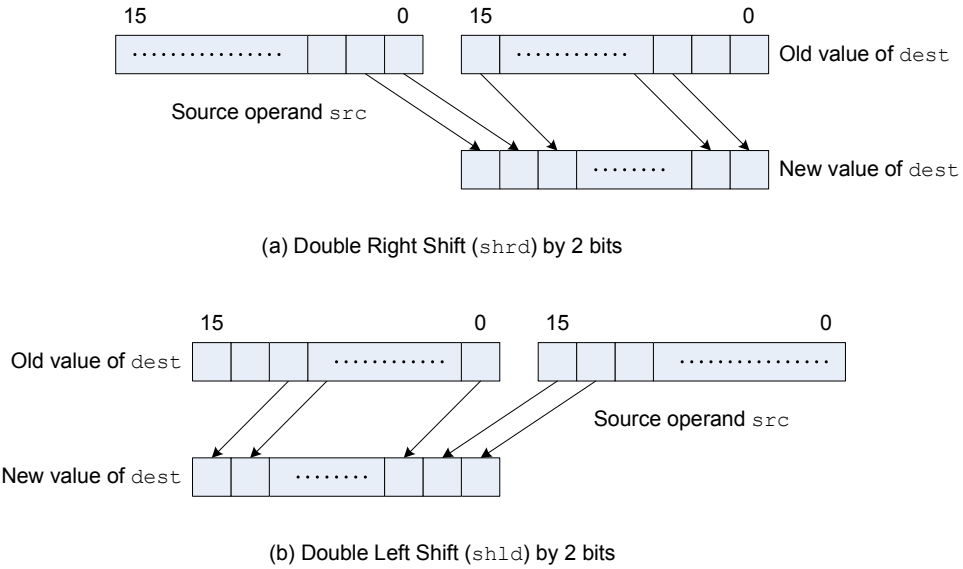


Figure 5.3: Double Shift Instructions

(figure 5.2(c)).

IA32 architectures provide two more instructions for logical shifts. These instructions operate on a number that is obtained by concatenation of two operands. Syntax of these instructions is as shown below.

```
shld count, src, dest
shrd count, src, dest
```

The count operand can be specified as an 8-bit immediate constant or as register `cl`. The `src` can be a 16-bit or 32-bit register operand while `dest` can be a register or a memory operand of the same size as of the `src` operand. While shifting the bits of `dest` operand, incoming bits are taken from `src` operand. `Src` operand however does not get modified because of this instruction. Figure 5.3 shows the operations of `shld` and `shrd` instructions.

In IA32 architecture also provides instructions to rotate bits of an operand. In rotation operations, outgoing bits are brought in place of incoming bits. As opposed to this, the incoming bits are set to either 0 or to sign bit in shift operations. Thus after n rotations, the original value is restored back in the operand of size n bits.

There are two kinds of rotations in IA32 architectures. In one kind of rotation (figure 5.4(a) and (b)), the carry flag is not involved in the rotation itself. It however gets modified. In other kind of rotations (figure 5.4(c) and (d)), the carry flag is involved in the rotation. Thus outgoing bit goes to the carry flag and incoming bit is taken from the

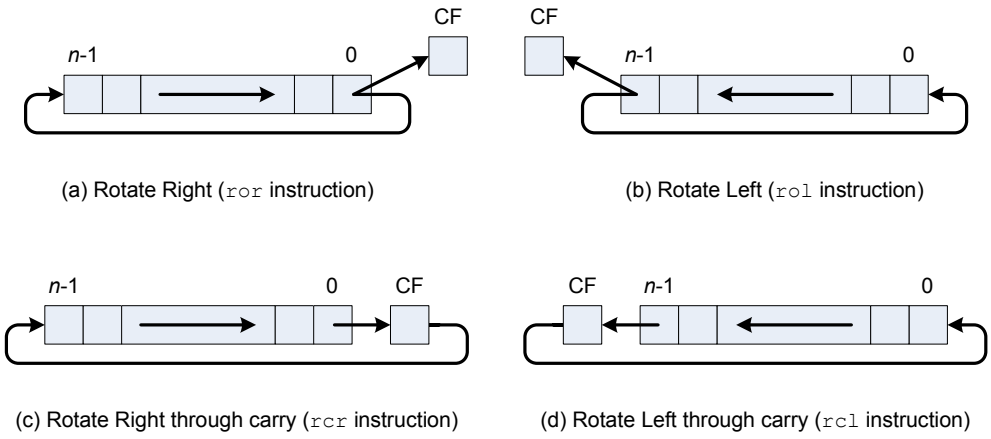


Figure 5.4: Rotate Instructions

carry flag. For multi-bit rotate operations, carry flag can be thought of as attached to the register either at MSB (for left rotation) or at LSB (for right rotation). Thus rotate through carry operation can be taken as rotation of $n + 1$ bits where n is the size of operand.

Four instructions in this category are as follows.

```
ror count, dest
rol count, dest
rcr count, dest
rcl count, dest
```

The `count` is an 8-bit number specified either as an immediate constant or as register `cl`. The operand `dest` can be any register or memory operand of size 8-bits, 16-bits or 32-bits.

Various operations of the rotate instructions are shown in figure 5.4.

EXERCISES:

5.16 In this exercise, we will build an Assembly language function that will count the number of bits set to 1 in its operand. Use rotate instruction and increment a counter if the carry flag is set. Write assembly language function with the following prototype in C and test your function through C interface.

```
int countBits(int x);
```

5.17 The earlier program of counting the number of bits has a loop that is repeated 32 number of times. We will modify this program to repeat the number of loop only as many times as the number of bits in the argument. At each iteration perform a bit-wise AND operation on x and $x - 1$. This number returns a number obtained from x such that the least

significant bit of x that is set to 1 switches to 0. This number can be set to x and the loop can be repeated till x becomes 0. Modify the program in exercise 5.16 to implement this algorithm. Test your program with C functions that perform the input and output.

5.18 In this exercise one has to write a program to compute x^y where x and y are integer parameters to the Assembly language function. We will use a shift and multiply approach for this. To simplify the program, x and y can both be taken as unsigned numbers. Assume y is an 8-bit operand where x is a 32 bit operands. The program logic is the following. Let's denote bit pattern of y as $y_7y_6 \ldots y_2y_1y_0$. Thus

$$y = y_7.2^7 + y_6.2^6 + \ldots y_2.2^2 + y_1.2^1 + y_0.2^0$$

As y_i can only be a 0 or 1, y can essentially be represented as $2^{z_1} + 2^{z_2} + \ldots$ where z_i is taken as those bit locations where y_{z_i} is 1. Now x^y can be computed as multiplication of terms $x^{2^{z_1}}$, $x^{2^{z_2}}$ etc.

For example, let's say that y is 5. Therefore y_2 and y_0 are the only bits that are 1 in the binary representation of y. Therefore $y = 2^2 + 2^0$, or z_2 and z_1 are 2 and 0 respectively. Now x^y can be evaluated as $x^{2^2}.x^{2^0}$.

In order to implement the algorithm, maintain two variables. One variable keeps the successive powers of x. Let's call this variable as `PowerX` and initialize it to x. At each step of the algorithm, `PowerX` is replaced by the square of `PowerX` to to yield x^2, x^4, x^8, ... The other variable keeps the track of successive bits of y. Let's call this variable as `Yt` which is initialized to y in the beginning. The `Result` is initialized to 1.

At each step of the algorithm (there are a total of 8 steps as y is 8 bit wide), if least significant bit of `Yt` is 1, `Result` is multiplied by `PowerX`. At the end of the step, `Yt` is shifted right by one bit location and `PowerX` is multiplied by `PowerX`.

Build a loop where this algorithm is implemented. With the appropriate initializations, the function will implement the power function for integer arguments. At any time during the multiplication overflow indicates that the result is to large to store in the alloted number of bits. Overflow may occur during squaring of `PowerX` and multiplication into `Result`. In your program, you may use registers for storing `PowerX`, `Yt` and `Result`.

Chapter 6

String and bit-oriented instructions

IA32 architectures support instructions for operating on a large number of bytes in memory called strings. The strings in context of IA32 architectures are different than the strings as used in programming languages such as C. IA32 strings do not have any specific significance for any particular byte value such as NULL in C strings.

IA32 architectures support instructions to read string–element by element, into registers and operate on the element, copy a string from one memory location to another–element by element, store string in the memory, compare two strings, search for a particular element value in string etc. By combining some instruction prefixes to repeat the execution of string instructions very powerful string processing can be added in softwares written for IA32 processors.

IA32 architectures also support a large number of instructions to operate on registers or memory at the level of bits as opposed to at the level of the size of the element such as byte, word etc. These instructions allow setting a bit to 0 or 1, inverting a bit, testing a bit, searching for a bit, looking and setting individual bits of the flags, checking flag bits and setting a byte accordingly, etc.

6.1 String in IA32 architectures

In IA32 architectures, strings are characterized by following attributes.

- **Address:** Address of a string provides the address in memory of the first or the last element of the string. Whether the address is that of the first element or the last element is decided by direction flag (described later). The address is provided in one of the two registers specially used by string instructions. After execution of

string instruction for each element, the address is automatically adjusted by the size of the element. If address originally provided was for the first location, it is incremented, otherwise it is decremented.

- **Size of each element:** As in most other IA32 instructions, string instructions also support the size of an element to be 8-bits, 16-bits or 32-bits. The GNU/Linux Assemblers support the suffixing of the instructions by 'b', 'w' or 'l' to denote byte (8-bits), word (16-bits) or long (32-bits) string operands.

- **Number of elements:** Elements in an IA32 string can have any arbitrary values. Thus unlike strings in C programming language, strings in IA32 architectures can not have any distinct terminal value. To find out the size of a string, therefore, no particular terminal value need to be used. Therefore, the size of strings in IA32 architectures is provided separately.

  In fact, the string instructions in IA32 architectures operate only on one element at a time. Thus the size of a string is not used by string instructions themselves. Typically a string instruction is used in conjunction with prefix instructions (we will know about these instructions later in this chapter) or loop instructions that uses the 'number-of-elements' information and repeat string instructions for each element.

  The string instructions take up to two string operands. The instructions that take two string operands require the size of two strings to be the same. Thus size of the strings can be specified in one register only. Therefore, register `ecx` is used in string instruction to store the count of number of elements. For each execution of a string instruction through the prefix instruction, register `ecx` is decremented.

- **Direction for address adjustment:** As said earlier, initial address of a string can be provided as address of the first element or that of the last element. Each time a string instruction is executed, the addresses are adjusted. Direction of address adjustment can be upward or downward (i.e. the address is incremented or decremented) depending upon the address being that of the first element or that of the last element.

  In IA32 architectures, a flag known as direction flag is used to represent the direction for address adjustment. If direction flag is 0, addresses are incremented and the initial address is assumed to be the address of the first element. However, if direction flag is set to 1, the addresses are decremented. In IA32 architecture, no instruction modifies the direction flag implicitly. Thus unless an explicit instruction is used to modify the direction flag or `eflags`

register, direction flag does not change. IA32 processors provide `cld` and `std` instructions to set direction flag to 0 and 1 respectively.

6.1.1 String instructions

In IA32 architectures, some string instructions operate on only one string while others operate on two strings. For example, an instruction to copy one string to another string element by element, operates on two strings. The strings are therefore characterized as source string or destination string. The string that is used to provide values of the elements is called the source string. A special source index register (register `esi`) is used to provide address of an element in source string. The string that gets modified during execution of the instruction is known as destination string. A special destination index register `edi` contains address of an element in destination string.

The string instructions supported in the IA32 architectures are the following.

```
movs
cmps
scas
lods
stos
```

None of the string instructions take any explicit operand. String addresses and direction are all implicitly known through `esi` and `edi` registers and direction flag. As none of these instructions take any operand, it is necessary to put a suffix 'b', 'w', or 'l' to denote the size of string elements.

The `movs` instruction copies a byte, word or a long from source string to destination string. After each copy, the addresses stored in the `esi` and `edi` registers are adjusted depending upon the value of the direction flag, DF, and the size of each string element.

A rough equivalent to the operations performed by `movs` instruction is following.

```
copy as many bytes as specified by the size
     from (%esi) to (%edi)
if (DF = 0)
     %edi = %edi + size
     %esi = %esi + size
else
     %edi = %edi - size
     %esi = %esi - size
```

No flags are modified by this instruction.

The `cmps` instruction compares an element of source string with an element of destination string and sets the flags according to the comparison. No string operands are however modified. The source and the destination string index registers initially point to the elements of source and destination string and are adjusted according to the size of string elements and direction flag.

A good use of the `cmps` instruction is to check if the two strings are identical or not. In the following code fragment, two strings are being compared. While this operation can be done with an instruction prefix as well, in the example code given below we do not use it.

```
    mov  $STR1, %esi // Beginning addresses of
    mov  $STR2, %edi // the two strings
    mov  $LEN, %ecx
    cld              // Set DF = 0 such that
                     // the index registers
                     // are incremented.
B0: cmpsb
    loope B0 // Decrement ecx and
             // go back if last comparison
             // had the ZF set or if the entire
             // string is compared.

// At this point if ZF = 1, two strings are
// identical. Otherwise esi and edi registers
// contain the address of the string element
// next to the mismatched one.
```

At end of the code, the zero flag indicates if there was a match or not. The flag is set to 1 when two strings are identical, and to 0 when two strings are not identical. In the later case, source and destination index registers (`esi` and `edi`) contain the addresses of elements in the string whose previous elements resulted in a mismatch.

EXERCISES:

6.1 In some cryptographic applications, a string is encrypted to generate another string. It is often desirable that the two strings must not be similar in any location. Write a program in Assembly language that checks if two strings are different in each element location or not. The program should be callable from C run-time system. Test it out with a C program that passes beginning addresses and length of the two strings to the function and prints if the two strings pass the cryptography quality test or not.

6.2 Write an Assembly language function that copies string from one location to another. Two memory areas may be overlapping. The program should

work properly by checking if destination address is between source address and (source address + length). If this condition is true, the copy should be done in reverse order (by setting direction flag to 1).

The other three instructions, `scas`, `lods` and `stos` operate with only one string operand. In case of `lods` instruction, address of the string is provided in register `esi` while in the case of `scas` and `stos` instructions, addresses of string elements are provided in register `edi`. The other operand in all these instructions is implicitly specified as register `al`, `ax` or `eax` depending upon the size of the string element being 8-bits, 16-bits or 32-bits.

The `scas` instruction compares the string element by element with value provided in register (`al`, `ax` or `eax`). In order to do so, the instruction subtracts the string element (`(%edi)`) from register `al`, `ax` or `eax` without modifying the register. The flags are then set according to the subtraction. Thus using this instruction, a particular value of the string element can be searched either from beginning or from end of the string.

EXERCISES:

6.3 Standard C library provides a function called `index`. The function takes two arguments, the beginning address of the string and the character to be searched in the string. The function returns pointer to the string location where the character is found or NULL if the character is not found. Standard C library function works with C strings. In this exercise we will develop an assembly function similar to `index` function with a difference that it would operate on strings as supported by IA32 architectures.

The function will take three arguments, the beginning address of the string, length of the string and the byte value to be searched. The return values will be similar to the ones provided by the `index` function call.

Test this function with the C interface. Also develop a function similar to this function and call it `rindex`. The `rindex` function searches for a character from end of the string and returns the first (*i.e.* last from the beginning) match or NULL if no match was found.

6.4 Write a function that counts the number of times a particular character appears in the string. The function should take three parameters from C interface. The first parameter is beginning address of the string, the second parameter is length of the string while the third parameter is character to be counted for the number of occurrences.

The instruction `lods` reads one element of string from memory and puts in register `al`, `ax` or `eax` depending upon the size of the string element. String address is specified in register `esi`. As with other instructions, register `esi` is adjusted depending upon the direction flag and the size of the string element.

Counterpart of `lods` instruction is `stos` instruction. Depending upon the size of the string element, it stores value of register `al`, `ax` or `eax` in the memory whose address is specified in register `edi`. Register `edi` also gets adjusted according to the size of the string element and the direction flag.

The string instructions implement very powerful mechanisms in IA32 architectures to perform a variety of tasks. For example, a large array can be initialized to a particular value using `lods` instruction.

6.1.2 Using string instructions in effective manner

In IA32 architecture instruction set, there are several instructions that by themselves do not perform any action but effect execution of the next instruction in certain manner. Such instructions in IA32 architectures are termed as "instruction prefix" or prefix instructions. Processors provide several instruction prefixes many of which do not really matter in the environment of the GNU/Linux operating system.

There are five prefix instructions that can be combined with string instructions to execute them in an effective manner. Prefix instructions do not require any operand and affect the execution of the next instruction.

```
rep
repe
repne
repz
repnz
```

The valid usage of these instructions are given in the table below. Certain string instructions can be used with a set of instruction prefixes. In reality, `rep`, `repz` and `repe` instructions have the same opcode. Similarly instructions `repnz` and `repne` have the same opcode.

String Inst	Can be used with				
	`rep`	`repe`	`repz`	`repne`	`repnz`
`movs`	√				
`cmps`		√	√	√	√
`scas`		√	√	√	√
`lods`	√				
`stos`	√				

In this table a '√' indicates that string instruction in that row can be used with instruction prefix in that column. If prefix instructions are used in a manner not specified, results may be unpredictable. Thus if `rep` instruction is used as a prefix to the `add` instruction, the results are undefined and depend upon the processor's implementation.

The rep prefix instruction causes execution of subsequent string instruction to be repeated ecx number of times, decrementing register ecx each time by one. Thus at end of the execution, register ecx becomes 0 and corresponding string index registers are adjusted by an amount determined by the size of string element, initial contents of register ecx and direction flag DF. Following program segment shows the use of prefix instructions to copy a string from one location to another.

```
    mov  $SRCSTR, %esi  // Assuming that string
    mov  $DESTSTR, %edi // lengths are multiple of 4
    cld                 // this program copies them as
                        // one long word each time.
    mov  $LEN, %ecx // Size in number of bytes
    shr  $2, %ecx   // Get ecx = size in long words
    rep
    movsl           // initial ecx is size in words
```

In this program, it is assumed that string length is a multiple of four bytes. In such a case, we can copy strings four bytes (or one long) at a time. Thus in a single execution of string instruction movsl, four bytes are copied from source to destination. At each iteration, index registers are incremented by four (size of the elements). Direction flag is set to 0 as index registers initially contain addresses of the beginning of the string rather than that of the end. In this program we assume that strings are non-overlapping. If they overlap, incorrect results may occur.

Prefix instructions repz and repe repeat the next string instruction till at least one of the two conditions are satisfied.

- Register ecx gets to zero after decrementing it by 1.
- The zero flag ZF becomes 0 due to execution of the string instruction (such as cmps or scas).

If register ecx is zero in the beginning, corresponding string instruction is not executed even once.

Prefix instructions repz and repe can be used with scas and cmps string instructions. These are the only two instructions that update the flags. Thus the repz instruction can be used to check if the two strings are equal or if all the element of a string are the same as the one specified in the register.

The prefix instructions repnz or repne are really the same instructions. Their behavior is similar to that of repz and repe instructions except that terminating conditions are modified as follows.

- Register ecx gets to zero after decrementing it by 1.

- The zero flag ZF becomes 1 due to execution of the string instruction.

None of the `rep` kind of prefix instructions modify any flags.

We can use these instructions in several ways. In the following example, we check if all the bytes in a particular memory block have the same value as specified in register `al` or not. In case of a mismatch, the address of the memory block is put in the `esi` register. The mismatched value is also returned in the register `al`. Thus upon return if the register `al` contains a value different than the original value, it is identified as a mismatch.

```
     // In the beginning it is assumed that
     // register esi contains starting
     // address of the string. ecx contains
     // length of the string and direction
     // flag is set appropriately. Register al
     // contains byte that is to be checked for.

     repz
     scasb
     jz nomismatch
     // If there is a mismatch, regsiter esi
     // contains address of the next location.
     // Adjusting esi and moving value in
     // register al is necessary here.

     dec %esi
     mov (%esi), %al
nomismatch:
     ret
```

EXERCISES:

6.5 Standard C library on GNU/Linux provides a function called `memset`. This function is used to set a block of memory to a predefined value. The function takes three parameters – (i) `s` as the address of the memory block, (ii) `c` as the byte to be set in the memory block (the byte is passed as an integer and only 8-bits of this integer are used to set the memory), and (iii) `n` as size of the memory block in bytes. Function always returns original value of the memory block address, *i.e.* the parameter `s`.

Implement this function in assembly language in two different ways.

- Using `stosb` instruction.
- Using a combination of `stosl`, `stosw` and `stosb` instructions.

In the second case, care must be taken to ensure that address of the memory block is a multiple of four before using `stosl` instruction. Thus use up to one `stosb` and up to one `stosw` instruction to align the address to a multiple of 4. Use `stosl` instruction with appropriate prefix instruction to fill in as many bytes as possible. At the end again up to one `stosb` and one `stosw` can be used to copy the remaining number of bytes.

Care must also be taken to ensure that all bytes of register `eax` contain same value as given in lower eight bits of parameter `c`.

6.6 Standard C library in GNU/Linux distributions provide a function called `swab` that copies a number of bytes from a source area to a destination area swapping odd and even bytes. The function takes three arguments '`from`', '`to`' and '`n`'. The `from` argument provides address of source memory block, the `to` argument provides address of destination memory block and the `n` argument provides size of the memory blocks in bytes.

If argument `n` is an odd number, the last byte is copied without any swapping. For the remaining $(n-1)$ bytes swap and copy is used where bytes at source locations $2i$ and $2i+1$ are copied to destination locations $2i+1$ and $2i$ respectively. Use `lodsw` instruction to load the values in register `ax` and then `xchg` instruction to swap bytes in `al` and `ah` registers. Use `stosw` instruction to copy `ax` register to the destination. The process can be repeated using `loop` instruction to copy all the bytes.

For simplicity, assume that the two areas given by `from` and `to` do not overlap in memory.

6.2 Bit-oriented instructions

The bit-oriented instructions in IA32 architectures test and modify individual bits in their operands. Some of these instructions also use conditional flags in various ways.

6.2.1 Bit testing and modification

The following instructions operate on individual bits and set the condition flag CF according to the bit values.

```
bt bitoffset, dest
bts bitoffset, dest
btr bitoffset, dest
btc bitoffset, dest
```

The `bitoffset` field can be an immediate constant, a 16-bit register or a 32-bit register. The `dest` can be a 16-bit register, a 32-bit register or a memory location. If a register is chosen for the `bitoffset` field, the size of the register must be same as the size of the `dest` field.

The `bt` instruction copies the specified bit from `dest` operand into the carry flag (CF). The `bts` instruction copies the specified bit into the

carry flag and at the same time sets that bit to 1. The `btr` instruction is similar to the `bts` instruction except that it set the new value of the chosen bit to 0. Similarly, the `btc` instruction complements the chosen bit after copying the old value to the carry flag.

These instructions along with `jc/jnc` instructions can be used to test values of a bit and make jump accordingly. At the same time, last three instructions may be used to change the selected bit as well.

There are the following possibilities for specifying operands of these instructions.

- The `bitoffset` can be specified as an immediate constant. In this case, the `dest` can be a register or memory operand. If the size of `dest` is 16-bit (i.e. either a 16-bit register is used or `w` is suffixed to the instruction when the operand is in memory), only the four least significant bits of the `bitoffset` are used while other bits are ignored. Similarly if the size of the `dest` is 32-bit, only the five least significant bits (i.e. offset between 0 to 31) of `bitoffset` are used.

- The `bitoffset` can be specified in a 16-bit register. In this case, the `dest` can be a 16-bit register or a 16-bit memory operand. If the `dest` operand is a register, only the four least significant bits are used to denote the bit offset. Thus a value of 0 to 15 is used for the `bitoffset`.

  If the `dest` operand is a memory operand, value of the `bitoffset` is taken as a 16-bit signed number between -2^{15} to $+2^{15}-1$ (or, -32768 to 32767). The memory address in the instruction provides the address of a buffer that stores bit data. If this address is a, byte stored at location a provides bits 0 to 7. Bytes at location $a+1$ provides bits 8 to 15 and so on. Byte stored at location $a-1$ provides bits -8 to -1. Similarly bytes stored at location $a-2$ provides bits -16 to -9. Figure 6.1 illustrates the `bitoffset` for memory operands. The bit corresponding to the bit offset is operated upon during execution of this instruction.

- The `bitoffset` can be specified in a 32-bit register. In this case, if the `dest` operand is in a register, only lower five bits of the `bitoffset` register are used and other bits are ignored. Thus an offset of 0 to 31 is used to specify the bit location of the register holding the `dest` operand.

  However, if the `dest` operand is in memory, then the `bitoffset` value is taken as a 32-bit signed number having a value between -2^{31} to $+2^{31}-1$, both inclusive. Thus using such an addressing, bits of a memory area within approximately 2^{29} bytes can be accessed.

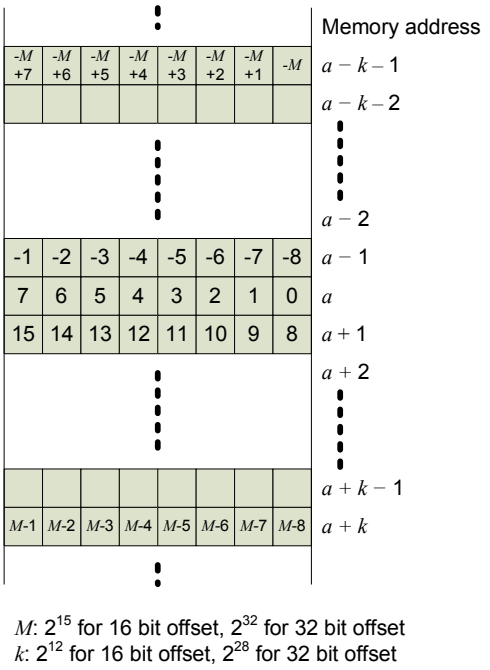


Figure 6.1: Interpretation of bit offset for memory operands

If the effective address of the `dest` operand evaluates to a, bits 0 to 7 are assumed to be at location a. Bits 8 to 15 are assumed to be at $a+1$, bits 16 to 23 at $a+2$ and so on. Similarly bits -8 to -1 are assumed to be at memory location $a-1$, bits -16 to -9 at location $a-2$ and so on (figure 6.1). The corresponding bit is tested and changed according to the instruction.

EXERCISES:

6.7 Write a program in Assembly language to test the bit offset behavior for the memory operands. In order to do so, use the `btc` instruction to complement the bit. The Assembly language function must take two parameters, the address of the buffer and the bit offset.

In a C program declare an array of size say n integers. Initialize this array to some values of your choice. From this C program, call Assembly language function with the address of the $\frac{n}{2}$th element of the array (to test the negative offset). Run C program for various values of bit offset and check if results are as expected or not.

6.8 Often in graphics application, a part of the image (for example cursor) is highlighted by changing the appearance. In this exercise, you are expected to write a C callable function that takes two arguments. The first

argument provides address of the memory location while the second provides length of the memory (in bytes). In order to highlight the image stored at this address, every alternate bit is inverted. The function returns no value. (Hint: Assume that the address provides a memory block representing 8 times the length (second argument) number of bits. Write a loop that modifies each alternate bit using negation.)

6.2.2 Searching for a set bit

In IA32 processors, two instructions can be used to return the indices of the most significant or the least significant bit that is set to 1. The following are these instructions.

```
bsf src, dest
bsr src, dest
```

Destination of these two instructions can only be a 16-bit or 32-bit register. For 16-bit destination, the source can be 16-bit register or memory. Similarly for the 32-bit destination, the source can be 32-bit register or memory.

If no bit in the `src` operand is 1 (i.e. the operand is 0), ZF flag in the `eflags` registers is set to 1 and the value of the `dest` operand is undefined.

Instruction `bsf` (bit scan forward) is used to return index of the least significant bit that is set to 1 in the `src` operand. Similarly the `bsr` instruction returns index of the most significant bit that is set to 1.

An example of using these instructions is given below. In this example, we count the number of bits by using a counter in register `eax`. The counter is first initialized to 0. We then search for a bit that is set to 1. If no such bit is found then count is returned. Otherwise count is incremented by 1 and the bit is set to 0. The process is repeated till all bits become 0.

```
countbits:
     // This function counts the number of bits
     // set to 1 in the register ebx and returns
     // the count in register eax.

     // ecx is used as a temporary register
     push %ecx
     push %ebx

     xor  %eax, %eax // eax = 0 initially
back:
     bsf  %ebx, %ecx // ecx is the bit index
```

```
     jz   done       // If ebx was 0
     // Turn the ebx bit to 0 and increment
     // the count in register eax
     btr  %ecx, %ebx // Set the bit to 0
     inc  %eax
     jmp  back
done:
     pop  %ebx // Restore the values
     pop  %ecx
     ret
```

We also use the bit-oriented instructions in another example that reverses bit pattern in a 32-bit operand in register `eax`. Thus if originally bit 0 was set to 1, it will be repositioned at bit index 31. Briefly the algorithm for our example is the following. We scan the bit pattern to find index of a bit that is set to 1. We then unset this bit in the source and set a bit in destination at location (32-i) where i is the index of the bit in the source. The program is given below.

```
reversebits:
     // eax is input and output register.

     push %ecx  // Save registers used
     push %ebx  // to hold temporary values

     xor  %ebx, %ebx // Initial value of ebx=0
back:
     bsf  %eax, %ecx // ecx is the bit index
     jz   done   // done when all bits of eax=0
     // Turn the eax bit to 0 and set the
     // 31-ecx bit of ebx
     btr  %ecx, %eax // Set the bit to 0
     // Compute 31-ecx
     sub  $31, %ecx // first compute ecx-31
     neg  %ecx      // Then negate it.
     bts  %ecx, %ebx
     jmp  back
done:
     mov  %ebx, %eax // eax is the result
     pop  %ebx // Restore the values
     pop  %ecx
     ret
```

EXERCISES:

6.9 Write an Assembly language program to find length of the longest sequence of zeros in a 32-bit register `eax`. Hint: The length of a sequence of zeros is known as one less than the difference of the bit indices of two subsequent bits that are set to 1. Thus at each iteration of the loop, value of the previous bit index can be subtracted from the value of the current bit index. As usual, the current bit will be turned off in the loop after each iteration. If the difference is more than the previously held difference, it is updated.

At the end of the loop, the difference will provide one more than the longest length. You will need to take care of the initial value of the previous index (what should you have?). Also at the last iteration, ZF will be set to 1 indicating that register `eax` has become zero. However the last sequence of zeros would not have been taken into account. Think of the solution to this problem as you write your program.

6.10 Often in digital communication systems, a technique is used to ensure that a zero occurs in a bit string after every few consecutive ones. In order to implement this, we need to find out if a 32-bit number contains a sequence of four consecutive ones or not. There are several ways to do this but we would like to implement it using bit oriented instructions of the processor.

Write a program that takes a 32-bit number as its input and returns 1 or 0 in register `eax` to indicate whether or not the input number contains a sequence of 1's with length larger than or equal to 4. (Hint: As you scan for a 1 forward, you hit upon the index that has 1 and all the bits at indices lower than that are 0. Let us call this index as i. You might add 2^i to the number to get another number and find the index of the bit that is set to 1 in this number. It should help you to find the length of the bit string that has all 1's and to the right of which it is all 0s. For example, if you add 100 to the string 001100111100, you get 001101000000. How can you use this fact to find length of the sequence of all 1s?)

6.2.3 Using condition codes

The IA32 processors handle one of the 16 conditions by combining several conditional flags as discussed earlier. They also provide instructions that return the results of these condition evaluation in one of the registers or memory. The syntax of the instructions are as follows.

```
set<u>cc</u> dest
```

The `cc` in instruction mnemonic indicates the condition. The `dest` field can be one of the 8-bit registers or an 8-bit memory location.

After the execution, if condition flags indicate that the specified condition was true, `dest` is set to 1. Otherwise `dest` is set to 0.

The following are thirty different possibilities for `set`*cc* instructions. Some of them mean the same thing as can be seen by the conditions on the flags in the third column of this table.

Instruction	Condition	Flags
sete	Equal	ZF=1
setz	Zero	ZF=1
setne	Not Equal	ZF=0
setnz	Not Zero	ZF=0
seta	Above	CF=0 and ZF=0
setnbe	Neither Below nor Equal	CF=0 and ZF=0
setbe	Below or Equal	CF=1 or ZF=1
setna	Not Above	CF=1 or ZF=1
setae	Above or Equal	CF=0
setnb	Not Below	CF=0
setnc	No Carry	CF=0
setb	Below	CF=1
setnae	Neither Above nor Equal	CF=1
setc	Carry	CF=1
setg	Greater	ZF=0 and SF=OF
setnle	Neither Less nor Equal	ZF=0 and SF=OF
setng	Not Greater	ZF=1 or SF<>OF
setle	Less or Equal	ZF=1 or SF<>OF
setge	Greater or Equal	SF=OF
setnl	Not Less	SF=OF
setnge	Neither Greater nor Equal	SF<>OF
setl	Less	SF<>OF
seto	Overflow	OF=1
setno	No Overflow	OF=0
sets	Sign	SF=1
setns	No Sign	SF=0
setp	Parity	PF=1
setpe	Parity Even	PF=1
setnp	No Parity	PF=0
setpo	Parity Odd	PF=0

6.2.4 Testing for a bit pattern

The `test` instruction in IA32 architectures can be used to set condition flags according to certain selected bits. This instruction takes two arguments and performs an AND operation of these two numbers. In the process, only the flag bits are modified and none of the input operands are changed. The instruction has the following syntax.

```
test src1, src2
```

`Src1` operand of the `test` instruction can be an immediate constant or a register. On the other hand, `src2` operand can be a register or a memory operand. Both operands must be of same size, 8-bits, 16-bits or 32-bits.

The condition flags of the processor are modified as follows.

The carry and overflow flags, CF and OF, are all set to 0. The parity, zero and sign flags are changed according to the operation (`src1` AND `src2`). Thus the sign bit is set to 1 only if both input operands have their most significant bits set to 1. The auxiliary flag (AF) bit is left undefined.

EXERCISES:

6.11 Write an Assembly language function, `scanbits`, using `test` and `bts` instructions to print indices of all the bits that are set to 1 in its input argument. Use the `printf` routine of standard C library to print the indices. The `printf` function should be called by `scanbits` function. Further, the `scanbits` function should be C callable. Test your implementation by writing a C function that takes input from the user and calls Assembly language function.

Chapter 7

Linux Kernel Interface

Operating systems such as GNU/Linux support multiple processes running simultaneously and time sharing the same CPU. At any time memory images of multiple processes exist simultaneously within the same physical memory. Operating system uses virtual memory technique to ensure that a process gets to operate on its own memory image. Similarly multiple other resources such as I/O devices etc. are used by a process in a mutually exclusive manner with other processes. In such a mechanism, a resource is used only by one process at a time and other processes wait for this process to release the resource before they can use it.

In order to implement protection of a process by other processes, multi-tasking Operating Systems such as GNU/Linux do not permit a process to do direct I/O. Instead, a process makes a call to a function within the OS to perform such operations. The operating system function then performs the I/O and other such private operations after checking for the credentials of the process. For example, a file can be read only if the process has a permission to read that file. Such functionality within the Operating System is called a mechanism of making "system calls".

The Operating System does not permit any part of the OS code to be executed by a process in the normal "user" mode. Thus calls to the Operating System functions are not made by normal `call` instructions. The processors provide a mechanism for making system call by using an instruction, often known as "trap" kind of instruction. Upon making a "trap" to the Operating System, which function within the Operating System will be executed is pre-determined by the Operating System during the time of booting the system.

A standard C programming environment provides a linkable C library that provides a procedural interface to the system calls. Through this interface, all Operating System functions can be called in a way similar to making a function call. Thus parameters to the Operat-

```
#include <asm/unistd.h>
#include <syscall.h>
#define STDOUT 1
.data
hello:
        .ascii "hello world\n"
helloend:
.text
.globl _start
_start:
        movl    $(SYS_write),%eax // SYS_write = 4
        movl    $(STDOUT),%ebx    // fd
        movl    $hello,%ecx       // buf
        movl    $(helloend-hello),%edx   // count
        int     $0x80

        movl    $(SYS_exit),%eax
        xorl    %ebx,%ebx
        int     $0x80
        ret
```

Figure 7.1: Hello World program `helloworld.S` in Assembly language

ing System call are pushed on the stack and a regular function call is made. The implementation of such function calls in C library ultimately make a system call to the Operating System.

By using Assembly language programming, a programmer can make Operating System call without going thorough the functions in the C library. This makes the image of the final executable code much smaller as the C library need not be linked with the program thereby reducing the final code size of the executable image. This is illustrated with an example in the code of figure 1.1 in Chapter 1 also reproduced here as figure 7.1. An equivalent C program for this Assembly language program is given in figure 7.2.

The C program `helloworld.c` and the Assembly language program `helloworld.S` were compiled on Linux with the following versions of the programs.

Versions of programs
`gas` : GNU assembler 2.14.90.0.7 20031029
`gcc` : gcc (GCC) 3.3.2 (Mandrake Linux 10.0 3.3.2-6mdk)
OS : 2.4.25-2mdk

The executable files obtained by compiling two files, `helloworld.S` and `helloworld.c`, have the following sizes.

```
#include <stdio.h>

int main() {
    printf("hello world\n");
    exit(0);
}
```

Figure 7.2: The `helloworld.c` program

Source file	Executable file size	
	Before `strip`	After `strip`
`helloworld.S`	1074 bytes	388 bytes
`helloworld.c`	11357 bytes	2984 bytes

It is clear from a program this small in size that C compiled program tend to be several times bigger than the programs written using Assembly language. However programming using Assembly language is not simple and requires a level of expertise that is very different than the one required for C programming. Further the programs are not portable across machines with different processors and require much more effort for the maintenance of the program.

As programs execute on the processor, certain kind of error conditions or exceptions are handled by the system. In general there are three kinds of exceptions in the system.

1. OS call or trap. Such events are synchronous to the program and will occur whenever a "trap" kind of instruction is executed in the program.

2. Fault conditions. Such events occur due to the execution of an instruction when there is an error condition relevant to that instruction. For example a divide-by-zero exception may occur when a division instruction is executed. Such events are synchronous to the program. These will occur only when the specific instruction is executed but may not occur altogether.

3. Interrupts conditions. Such events are due to the external conditions such as interrupts raised by the hardware when a key is pressed. Such events are asynchronous to the program and have no effect of which instruction is executing at the time of interrupt.

IA32 processors provide a "trap" kind instruction for handling Operating System call. The other two kinds of events are really handled by the Operating System and a user mode program may not have any control on such events.

```
int type
```

This instruction takes one argument called `type` which is an immediate constant value between 0 to 255 and represents the type of the exception condition. Each type of exception condition in IA32 architectures need to identify the interrupt type. The `type` field is identical for all such cases. The mechanism to identify the interrupt type is different for each type of the exception. In case of system call, interrupt type is given as an immediate constant to the `int` instruction. There are several exception types which are used by the IA32 processors to reflect fault conditions such as page fault, floating point exceptions, division by zero, illegal instruction etc. The processor automatically executes the handler for the associated type in case a fault condition is detected. A few more exception types are used by the system hardware to reflect hardware interrupts such as those from the hard disk controllers, floppy drives, keyboard etc. The actual behavior of this instruction is out of scope of this book. However we need to understand one instruction type that is used to make system calls in GNU/Linux on IA32 processors. In GNU/Linux, an exception of type 0x80, or 128 is used for making a system call to the operating system. Use of this instruction has been illustrated in figure 7.1 for the sample program. Thus instruction `int $0x80` is used to make a system call in GNU/Linux. Notice the presence of a `$` before 0x80 to indicate that this is an immediate constant. If this is omitted, it would mean a memory based addressing which is not a proper mode for this instruction and the assembler gives an error.

7.1 System call Identification

The GNU/Linux on an IA32 processor provides only one mechanism for entering into a system call – by using `int $0x80` instruction. Upon use of this instruction, control of the program transfers to a pre-determined location within the Operating System kernel program. This code is known as the system call handler. System call handler distinguishes between various system calls, such as `read`, `write`, `open` etc. as per the system call identification process. After the system call identification, appropriate function call is made to serve the system call.

Each system call in GNU/Linux is given a unique number for identification. This number is passed as an argument while making a system call. The system call handler looks for the validity of this number and if the system call is supported, functions to handle the system call are called.

In GNU/Linux for IA32 processors, register `eax` is used to pass the system call number. The installation of the Operating System provides a file `/usr/include/syscall.h` that defines symbolic constants for

the system call[1]. By including this file in the program (for example program in figure 7.1 includes this file), we use symbolic constants rather than the numbers for system call identification. In the program of figure 7.1, we have used to system calls. The first system call `write` is made to print the "hello world" string on `stdout`. The second system call `exit` is made to terminate the process. The system calls are made by passing constants such as `SYS_write` and `SYS_exit` in register `eax` before entering the system call thorough `int 0x80` instruction.

The GNU/Linux kernel version 2.4 provides about 250 system calls. In general, a constant `NR_syscalls` is available and is set to a constant such that all system call identification numbers are less than `NR_syscalls`. Not all identification numbers refer to a valid system call. Some of these system calls are provided for compatibility with earlier versions and better implementations are provided by alternate system calls.

7.2 Parameter Passing for system calls

In GNU/Linux, the parameters are passed to a system call through registers. All system calls in GNU/Linux take less than or equal to six parameters. The parameters are passed using registers `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp`. The first parameter is passed in register `ebx`, second in register `ecx`, third in register `edx` and so on. As mentioned earlier, in addition to the parameters of the system call, a system call identification number is passed in register `eax`.

In GNU/Linux, all system calls take only integer parameters or addresses of memory variables. Thus all parameters fit within 32 bits and only one register is used for one parameter.

Some system calls require addresses of memory variables. Such addresses are passed using registers. The system call can then update the values (if needed) into the memory directly. Some other system calls may just need large amount of data in a buffer in memory whose address is passed as parameter.

For example, `write` system call takes three parameters other than the system call identification number in register `eax`. The first parameter is an integer known as file descriptor. The second parameter is the address of buffer from where the data is written to the file identified by the file descriptor. The third parameter is an integer that provides the size of data to be written to the file. In GNU/Linux, a few descriptors are special and are available to a process automatically. The descriptor 0 is known as `stdin` or standard input. It is normally bound to the keyboard and is used by `read` system call to read data from the keyboard. The descriptor 1 is known as `stdout` or the standard output.

[1]In reality there is a hierarchical set of files that provide this definition. For our purpose we just need to include `syscall.h` in our programs.

An output to this file by means of a `write` system call normally appears on the screen. The descriptor 2 is known as `stderr` or standard error output. An output to this file also appears on the screen.

The following lines from code in figure 7.1 are used to make `write` system call.

```
1 movl    $(SYS_write),%eax // SYS_write = 4
2 movl    $(STDOUT),%ebx    // fd
3 movl    $hello,%ecx       // buf
4 movl    $(helloend-hello),%edx   // count
5 int     $0x80
```

Instruction in the first line moves the symbolic constant corresponding to the `write` system call into register `eax`. Registers `ebx`, `ecx` and `edx` are initialized to contain the parameters for the `write` system call. The file descriptor is initialized to `stdout`. Address of the buffer from where the output is to be printed is given in register `ecx`. Notice a prefix `$` before `hello` to indicate that it is immediate value (*i.e.* address) rather than the content of memory location that is being moved to `ecx` register. Finally the length is moved to `edx` register. The expression `helloend-hello` is a constant expression and is evaluated by the assembler at the time of converting this program to object code. Finally the system call is made after preparing all arguments in registers by the use of `int $0x80` instruction.

Similarly the following lines from code of figure 7.1 are used to make `exit` system call. The exit system call is used to terminate the process. Control is then returned to another process decided by the process scheduling strategy of the Operating System.

```
1 movl    $(SYS_exit),%eax
2 xorl    %ebx,%ebx
3 int     $0x80
```

Instruction in the first line puts a constant `SYS_exit` in register `eax`. This is then interpreted by the system call handler as a call to `exit` system call. The `exit` system call requires just one argument known as exit code. This is provided as a parameter in register `ebx` by setting it to zero in the second line. Bit-wise XOR operation of a value with itself always results in a bit pattern that is all 0s. Finally the system call is made in the third line of this code. Since this system call causes the process to terminate, there is no program needed after this instruction as the control will not return after the system call is made.

7.3 Return values from System Calls

The return value mechanism of System Calls are implemented in a way much similar to those in function calls. Like all functions, System Calls also return a 32-bit value as an integer. In the example code of figure 7.1, the `write` system call returns a value which is not used. The `exit` system call never returns and therefore there is no return value of `exit` system call.

System call return their values in register `eax`. There are two kinds of values in this register. All non-negative values returned by the system call represent successful execution of the system call and provide the normal value as per the functionality of the system call. All negative values on the other hand represent error conditions. In such case, the absolute of the return value provides an error code. A comprehensive list of error codes from a system call is available in file `/usr/include/asm/errno.h` which defines symbolic names to these error codes.

In addition to a direct return value from a system call, some system calls may also update the memory buffer whose address is passed as an argument. For example `read` system call reads data from a file descriptor and stores it in the buffer whose address is passed as an argument to the `read` system call.

7.4 Starting a process in GNU/Linux

In GNU/Linux, a program is executed by creating a process and then loading a program in newly created process. A process is created using `fork` system call while a program is loaded using `execve` system call. When a program is loaded, it is made to run from a memory location whose address is specified in the executable file. This address is known as the start address of the program.

By default, the start address of the program is indicated by a symbolic label `_start`. This address can be changed by an option to the `ld` command. In the program of figure 7.1 a symbol `_start` is used which defines the first instruction of the program to be executed when it is loaded in the memory.

As a contrast, the programs written in C do not specify any function by name `_start`. The programs written in C usually do not specify any start address of the program. The C library provides a startup code that provides `_start` address. This startup code makes a call to user function `main`. In this way, the `main` becomes the first user function to be executed.

Programs written in Assembly language can define any address as the start address of the program. In order to do so, a symbol must be declared global using `.globl` keyword and then by using the `-e` option

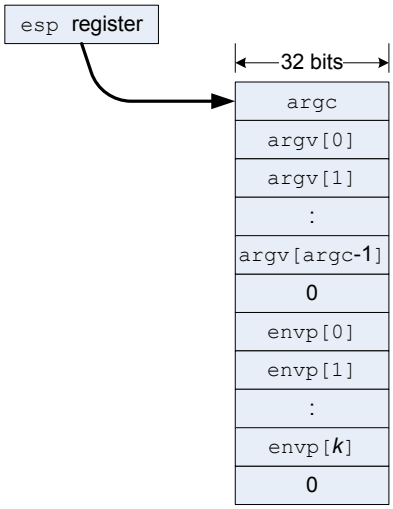


Figure 7.3: Stack layout at start of a program

of the linker `ld` (see Appendix E) that symbol may be defined as the start address.

When a program is loaded in the memory upon execution of `execve` system call, the Operating System performs several operations. It initializes the stack and various CPU registers before starting the execution of the program from its start address. The actual mechanism of this is out of scope of this book and is taught usually in various courses on Operating System.

7.5 Command Line Arguments

Command line arguments to a program are passed through the system call `execve` which causes a new program to be loaded. After the new program is loaded in the memory, operating system pushes parameters on the initialized stack before passing control to the starting address.

Just prior to the first instruction getting executed, the stack layout has the following items as shown in figure 7.3.

1. Number of arguments (`argc`) including the name of the program on top of the stack (32-bit number).

2. Address of starting memory location where the null terminated string is stored that refers to the name of the program. (`argv[0]`).

3. Addresses of the strings that store the first and subsequent arguments to the program if present (`argv[1]` .. `argv[argc`-1`]`).

4. A 32-bit number 0 on the stack as a delimiter between command line arguments and environment.
5. Addresses of environment strings (`envp[0]`..`envp[`k`]`).
6. A 32-bit number 0 to mark the end of the environment strings.

The minimum number of parameters pushed on the stack will be four for the case when no command line argument and environment is passed to the program. In such a case, the `argc` will be 1 to include just the name of the program. The second parameter on stack will be the address of string containing the name of the program. The third and fourth parameters on the stack will be two 32-bit numbers containing all zeros. The last two parameters indicate the end of the command line parameters and the end of the environment strings respectively.

7.6 Prominent System Calls

In this section we discuss some prominent system calls. A more detailed list of system calls can be found in file `syscall.h` available as a standard C include file.

7.6.1 File related system calls

System Call: `open`

Inputs:

`eax`: SYS_open
`ebx`: address of filename string
`ecx`: flags
`edx`: mode

Return Values:

Normal Case: file descriptor

Error Values: `-EEXIST`, `-EISDIR`, `-EACCES`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-ENXIO`, `-ENODEV`, `-EROFS`, `-ETXTBSY`, `-EFAULT`, `-ELOOP`, `-ENOSPC`, `-ENOMEM`, `-EMFILE`, `-ENFILE`

`Open` system call is used to open a file and return its file descriptor used later by other system calls. The address in `ebx` points to a null terminated string giving file name including the path. Various bits in flags argument provide the operations for which the file is open (for example, read-only, write-only, read-write etc.). In case, a new file is created, the mode argument provides the protection mechanism for the file.

System Call: `creat`

Inputs:

`eax`: SYS_creat

`ebx`: address of filename string

`ecx`: mode

Return Values:

Normal Case: file descriptor

Error Values: `-EEXIST`, `-EISDIR`, `-EACCES`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-ENXIO`, `-ENODEV`, `-EROFS`, `-ETXTBSY`, `-EFAULT`, `-ELOOP`, `-ENOSPC`, `-ENOMEM`, `-EMFILE`, `-ENFILE`

`Creat` system call is used to create a new file and return its file descriptor. The file is left open for writing.

System Call: `close`

Inputs:

`eax`: SYS_close

`ebx`: file descriptor

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EINTR`, `-EIO`

`Close` system call is used to close a file that was opened using `open` or `creat` system calls.

System Call: `read`

Inputs:

`eax`: SYS_read

`ebx`: file descriptor

`ecx`: buffer address

`edx`: length

Return Values:

Normal Case: Number of bytes read

Error Values: `-EINTR`, `-EAGAIN`, `-EIO`, `-EISDIR`, `-EBADF`, `-EINVAL`, `-EFAULT`

`Read` system call is used to read data from a file previously opened using `open` system call. The file descriptor returned by the `open` system call is used to define the file from where data is read into the buffer. The argument in `edx` register specifies the maximum number of bytes to read.

System Call: `write`

Inputs:

`eax`: SYS_write

`ebx`: file descriptor

`ecx`: buffer address

`edx`: length

Return Values:

Normal Case: Number of bytes written

Error Values: `-EINTR`, `-EAGAIN`, `-EIO`, `-EBADF`, `-EINVAL`, `-EFAULT`, `-EFBIG`, `-EPIPE`, `-ENOSPC`

`Write` system call is used to write data from specified buffer to a file. File descriptor (previously obtainted by `open` or `creat` system call) in register `ebx` defines the file, length and address of the buffer are in `edx` and `ecx` respectively.

System Call: `lseek`

Inputs:

`eax`: SYS_lseek

`ebx`: file descriptor

`ecx`: offset

`edx`: offset qualifier

Return Values:

Normal Case: offset from the start of the file

Error Values: -EBADF, -ESPIPE, -EINVAL, -EOVERFLOW

The lseek system call is used to move the current file pointer to any other offset as specified by the offset parameter in register ecx. Register edx contains a parameter that further qualifies whether the offset is specified from the beginning of the file (SEEK_SET, or 0) or from the current location of the file (SEEK_CUR, or 1) or from the end of the file (SEEK_END, or 2).

System Call: llseek

Inputs:

eax: SYS_llseek

ebx: file descriptor

ecx: High order 32 bits of offset

edx: Lower 32 bits of offset

esi: address of 64 bit number to provide result

edi: offset qualifier

Return Values:

Normal Case: 0

Error Values: -EBADF, -EINVAL, -EFAULT

The newer operating system versions support large files (file whose size is larger than 2GB). While 2GB is very large and sufficient for most applications, with the increasing disk sizes, it is possible to have files larger than 2GB. With the support of large file, a file size could be as large as 8 Exa-bytes ($\approx 8 \times 10^{18}$ bytes). The file offset, size etc. for the large file system are 64-bit numbers.

Functionally, llseek system call is similar to lseek except that the offsets are 64-bits. Thus the return value is also provided in a 64-bit memory location whose address is passed in register esi.

System Call: pread64

Inputs:

eax: SYS_pread64

ebx: file descriptor

ecx: buffer address

edx: count

esi: offset

Return Values:

Normal Case: Number of bytes read

Error Values: -EINTR, -EAGAIN, -EIO, -EISDIR, -EBADF, -EINVAL, -EFAULT

Pread64 system call is used to read data of a file from any specified offset. The file must be seekable. Corresponding libc function is known as pread (see pread(2)).

System Call: pwrite64

Inputs:

eax: SYS_pwrite64

ebx: file descriptor

ecx: buffer address

edx: count

esi: offset

Return Values:

Normal Case: Number of bytes written

Error Values: -EINTR, -EAGAIN, -EIO, -EBADF, -EINVAL, -EFAULT, -EFBIG, -EPIPE, -ENOSPC

Pwrite64 system call is used to write data to a file at specified offset. Corresponding libc function is known as pwrite (see pwrite(2)).

System Call: dup

Inputs:

eax: SYS_dup

ebx: file descriptor

Return Values:

Normal Case: New file descriptor

Error Values: `-EBADF`, `-EMFILE`, `-EINTR`, `-EBUSY`

`Dup` system call duplicates a file descriptor to a new one. The file can then be referred to by both file descriptors.

System Call: `dup2`

Inputs:

`eax:` SYS_dup2

`ebx:` file descriptor

`ecx:` new file descriptor

Return Values:

Normal Case: New file descriptor

Error Values: `-EBADF`, `-EMFILE`, `-EINTR`, `-EBUSY`

`Dup2` system call duplicates a file descriptor to the specified new file descriptor. If the specified new file descriptor is already open, it is closed first before duplicating.

System Call: `fsync`

Inputs:

`eax:` SYS_fsync

`ebx:` file descriptor

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EROFS`, `-EINVAL`, `-EIO`

The operating system maintains a cache in the memory where the modified data for a file is stored. This cache is periodically flushed to the disk. However there may be few instances when the data is not synchronized between the disk cache and the disk. `Fsync` system call synchronizes the cached file data and meta data on the disk storage.

System Call: `fdatasync`

Inputs:

eax: SYS_fdatasync

ebx: file descriptor

Return Values:

Normal Case: 0

Error Values: -EBADF, -EROFS, -EINVAL, -EIO

Fsync system call synchronizes the cached file data only (and not meta data) on the disk storage.

System Call: sync

Inputs:

eax: SYS_sync

Return Values:

Normal Case: 0

Error Values: No Errors. System call is always successful.

Sync system call synchronizes data from cache to the disk for all files.

System Call: link

Inputs:

eax: SYS_link

ebx: address of old pathname string

ecx: address of new pathname string

Return Values:

Normal Case: 0

Error Values: -EXDEV, -EPERM, -EFAULT, -EACCES, -ENAMETOOLONG, -ENOENT, -ENOTDIR, -ENOMEM, -EROFS, -EEXIST, -EMLINK, -ELOOP

The files in GNU/Linux operating system are identified in two ways – file pathname and file inode number. While the first one is human readable and is easy to organize, the second one is efficient for performing operations on the files. A directory provides mapping between file names and inode numbers. An association between file name and file inode is called a link. A file may be known by multiple names all pointing to the same inode.

The `link` system call is used to create a new link to an existing file. Register `ebx` provides the address of null terminated string that provides one of the names of the file. Register `ecx` provides the address of a non-existing file path name string. The second name is created with the same inode numbers as that of the first path name. Thus the file will have one more link (sometimes also known as a hard link) given by the second path name.

System Call: `symlink`

Inputs:

`eax:` SYS_symlink

`ebx:` address of old pathname string

`ecx:` address of new pathname string

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EFAULT`, `-EACCES`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-ENOMEM`, `-EROFS`, `-EEXIST`, `-ELOOP`, `-ENOSPC`, `-EIO`

In GNU/Linux, there are few special files which are known as symbolic link files. Symbolic link files will have their own inode but the file data will contain the name of the file to which this link points to.

The `symlink` system call creates a symbolic link file with pathname given in register `ecx`. The symbolic link file will contain the data as the pathname given in register `ebx`.

System Call: `unlink`

Inputs:

`eax:` SYS_unlink

`ebx:` address of pathname string

Return Values:

Normal Case: 0

Error Values: -EACCES, -EPERM, -EISDIR, -EFAULT, -ENAMETOOLONG, -ENOENT, -ENOTDIR, -ENOMEM, -EROFS, -ELOOP, -EIO

Unlink system call removes a link given by the pathname string. When a file has no link referring to it, the file is deleted.

System Call: readlink

Inputs:

eax: SYS_readlink

ebx: address of pathname string

ecx: buffer address

edx: buffer length

Return Values:

Normal Case: Number of bytes read

Error Values: -ENOTDIR, -EINVAL, -ENAMETOOLONG, -ENOENT, -EACCES, -ELOOP, -EINVAL, -EIO, -EFAULT, -ENOMEM

The contents of a symbolic link file can only be read using readlink system call. The normal read system call reads the contents of the file that the symbolic link points to rather than the symbolic link itself.

System Call: rename

Inputs:

eax: SYS_rename

ebx: address of old pathname string

ecx: address of new pathname string

Return Values:

Normal Case: 0

Error Values: -EISDIR, -EXDEV, -ENOTEMPTY, -EEXIST, -EBUSY, -EINVAL, -EMLINK, -ENOTDIR, -EFAULT, -EACCES, -EPERM, -ENAMETOOLONG, -ENOENT, -ENOMEM, -EROFS, -ELOOP, -ENOSPC

The rename system call renames a file link (name) to another. The name may be moved from one directory to another by specifying appropriate path.

System Call: newstat

Inputs:

eax: SYS_newstat

ebx: address of pathname string

ecx: address of stat buffer

Return Values:

Normal Case: 0

Error Values: -ENOENT, -ENOTDIR, -ELOOP, -EFAULT, -EACCES, -ENOMEM, -ENAMETOOLONG

The newstat system call provides the file control information for the file whose pathname is provided in a null terminated string. The call takes an address of a buffer whose format is available in a system include file /usr/include/asm/stat.h as a C definition of struct stat.

Earlier versions of GNU/Linux supported relatively smaller file systems. With the availability of large disks, larger file systems became a reality. The newstat system call provides data in the buffer that supports larger file systems. For compatibility reasons current GNU/Linux systems also provide the older stat system call which take the buffer address as that of a C definition of struct old_kernel_stat. The C library wrapper (lstat(2)) however has changed so as to refer to the newstat system call.

System Call: stat

Inputs:

eax: SYS_stat

ebx: address of pathname string

ecx: address of old stat buffer

Return Values:

Normal Case: 0

Error Values: `-ENOENT`, `-ENOTDIR`, `-ELOOP`, `-EFAULT`, `-EACCES`, `-ENOMEM`, `-ENAMETOOLONG`

The `stat` system call is provided only for the compatibility with the older versions of the operating system.

System Call: `newlstat`

Inputs:

`eax`: SYS_newlstat

`ebx`: address of pathname string

`ecx`: address of stat buffer

Return Values:

Normal Case: 0

Error Values: `-ENOENT`, `-ENOTDIR`, `-ELOOP`, `-EFAULT`, `-EACCES`, `-ENOMEM`, `-ENAMETOOLONG`

The `newlstat` system call is similar to the `newstat` system call except for the way the symbolic links are treated. In case of a symbolic link, `newstat` system call provides control information for the symbolic link file itself while `newlstat` system call provides control information for the file that the symbolic link points to.

The corresponding old compatibility version of the system call is `lstat` which takes buffer address of buffer containing data as specified in C datatype `struct old_kernel_stat`.

System Call: `newfstat`

Inputs:

`eax`: SYS_newfstat

`ebx`: file descriptor

`ecx`: address of stat buffer

Return Values:

Normal Case: 0

Error Values: -EBADF, -EFAULT, -EACCES, -ENOMEM

The newfstat system call is similar to the newstat system call except that it takes a file descriptor for an open file rather than the file path name string.

The corresponding old compatibility version of this system call is fstat.

System Call: stat64

Inputs:

eax: SYS_stat64

ebx: address of pathname string

ecx: address of stat64 buffer

Return Values:

Normal Case: 0

Error Values: -ENOENT, -ENOTDIR, -ELOOP, -EFAULT, -EACCES, -ENOMEM, -ENAMETOOLONG

The stat64 call is similar to the newstat system call except that supports large files and therefore provides data in buffer of type struct stat64 defined in system include file /usr/include/asm/stat.h.

System Call: lstat64

Inputs:

eax: SYS_lstat64

ebx: address of pathname string

ecx: address of stat64 buffer

Return Values:

Normal Case: 0

Error Values: -ENOENT, -ENOTDIR, -ELOOP, -EFAULT, -EACCES, -ENOMEM, -ENAMETOOLONG

The lstat64 system call is similar to the stat64 system call except that in case of symbolic links it provides information for the file that the symbolic link points to. The stat64 system call provides information of the symbolic link file itself.

System Call: `fstat64`

Inputs:

`eax`: SYS_fstat64

`ebx`: file descriptor

`ecx`: address of stat64 buffer

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EFAULT`, `-EACCES`, `-ENOMEM`

The `fstat64` system call is similar to the `stat64` system call except that it takes the file descriptor of an open file.

System Call: `truncate`

Inputs:

`eax`: SYS_truncate

`ebx`: address of the pathname string

`ecx`: length

Return Values:

Normal Case: 0

Error Values: `-EACCES`, `-EFAULT`, `-EFBIG`, `-EINTR`, `-EINVAL`, `-EIO`, `-EISDIR`, `-ELOOP`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-EROFS`, `-ETXTBSY`

The `truncate` system call is used to change the size of a file. If the new file size (in register `ecx`) is smaller than the old file size, the file is truncated. If the new file size is larger, file is enlarged by padding with zeros.

System Call: `ftruncate`

Inputs:

`eax`: SYS_ftruncate

`ebx`: file descriptor

`ecx`: length

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EINVAL`, `-EFBIG`, `-EINTR`, `-EIO`, `-EISDIR`, `-EROFS`, `-ETXTBSY`

The `ftruncate` system call is similar to the `truncate` system call except that is takes an open file descriptor.

System Call: `truncate64`

Inputs:

`eax`: SYS_truncate64

`ebx`: address of the pathname string

`ecx`: Lower 32 bits of length

`edx`: Higher 32-bits of length

Return Values:

Normal Case: 0

Error Values: `-EACCES`, `-EFAULT`, `-EFBIG`, `-EINTR`, `-EINVAL`, `-EIO`, `-EISDIR`, `-ELOOP`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-EROFS`, `-ETXTBSY`

The `truncate64` system call is for supporting large files whose size is given in 64-bits. Two registers `ecx` and `edx` are used to provide the new size.

System Call: `ftruncate64`

Inputs:

`eax`: SYS_ftruncate64

`ebx`: file descriptor

`ecx`: Lower 32 bits of length

`edx`: Higher 32-bits of length

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EINVAL`, `-EFBIG`, `-EINTR`, `-EIO`, `-EISDIR`, `-EROFS`, `-ETXTBSY`

The `ftruncate64` system call is similar to the `truncate64` system call except that is takes an open file descriptor.

System Call: `flock`

Inputs:

`eax`: SYS_flock

`ebx`: file descriptor

`ecx`: lock operation

Return Values:

Normal Case: 0

Error Values: `-EWOULDBLOCK`, `-EBADF`, `-EINTR`, `-EINVAL`, `-ENOLCK`

The `flock` system call is used to obtain an advisory lock on an open file whose descriptor is given in register `ebx`. Various lock operations are defined as `LOCK_SH`(1) for getting a shared lock, `LOCK_EX`(2) for getting an exclusive lock and `LOCK_UN`(8) for unlocking. The locks are advisory in nature, that is, all processes that would share the file must make a system call to `flock`. If an operation such as `read` or `write` is performed by a process without getting a lock, the operation is carried out without any regard to the lock.

This system call works only for the local files. The files that are provided by NFS are not locked using this system call. Instead `fcntl` or `fcntl64` system call are can be used in such scenarios.

System Call: `chown`

Inputs:

`eax`: SYS_chown

`ebx`: address of pathname string

`ecx`: user ID

`edx`: group ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EROFS`, `-EFAULT`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOMEM`, `-ENOTDIR`, `-EACCES`, `-ELOOP`

The `chown` system call changes the owner and group of the specified file. The user ID and the group ID can be specified as -1 for keeping them unchanged.

System Call: `lchown`

Inputs:

`eax`: SYS_lchown

`ebx`: address of pathname string

`ecx`: user ID

`edx`: group ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EROFS`, `-EFAULT`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOMEM`, `-ENOTDIR`, `-EACCES`, `-ELOOP`

The `lchown` system call is similar to the `chown` system call except that in the case of the file being a symbolic link, operation is performed on the symbolic link file rather than the file to which the symbolic link points to. In case of `chown` system call, the operations are performed on the file to which the symbolic link points to.

System Call: `fchown`

Inputs:

`eax`: SYS_fchown

`ebx`: file descriptor

`ecx`: user ID

`edx`: group ID

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-ENOENT`, `-EPERM`, `-EROFS`, `-EIO`

The `fchown` system call is similar to the `chown` system call except that it takes the file descriptor of an open file rather than the path name of the file.

System Call: `chmod`

Inputs:

eax: SYS_chmod

ebx: address of pathname string

ecx: mode

Return Values:

Normal Case: 0

Error Values: -EPERM, -EROFS, -EFAULT, -ENAMETOOLONG, -ENOENT, -ENOMEM, -ENOTDIR, -EACCES, -ELOOP, -EIO

The chmod system call changes the protection mode for the file.

System Call: fchmod

Inputs:

eax: SYS_fchmod

ebx: file descriptor

ecx: mode

Return Values:

Normal Case: 0

Error Values: -EBADF, -EPERM, -EROFS, -EIO

The fchmod system call changes the protection mode for an open file whose file descriptor is specified in register ebx.

System Call: rmdir

Inputs:

eax: SYS_rmdir

ebx: address of pathname string

Return Values:

Normal Case: 0

Error Values: -EPERM, -EFAULT, -EACCES, -ENAMETOOLONG, -ENOENT, -ENOTDIR, -ENOTEMPTY, -EINVAL, -EBUSY, -ENOMEM, -EROFS, -ELOOP

The rmdir system call is used to delete an empty directory.

System Call: `mkdir`

Inputs:

`eax`: SYS_mkdir

`ebx`: address of pathname string

`ecx`: mode

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EEXIST`, `-EFAULT`, `-EACCES`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-ENOMEM`, `-EROFS`, `-ELOOP`, `-ENOSPC`

The `mkdir` system call is used to make an empty directory with the permissions specified by mode argument in register `ecx`.

System Call: `chdir`

Inputs:

`eax`: SYS_chdir

`ebx`: address of pathname string

Return Values:

Normal Case: 0

Error Values: `-EFAULT`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOMEM`, `-ENOTDIR`, `-EACCES`, `-ELOOP`, `-EIO`

Each process in GNU/Linux has a working directory. This working directory defines the paths for the files which are relative to the current working directory. The working directory can be changed using `chdir` system call.

System Call: `fchdir`

Inputs:

`eax`: SYS_fchdir

`ebx`: file descriptor

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EACCES`

The `fchdir` system call is similar to the `chdir` system call except that it takes an open file descriptor rather than a file path name.

System Call: `chroot`

Inputs:

`eax`: SYS_chroot

`ebx`: address of the pathname string

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EFAULT`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOMEM`, `-ENOTDIR`, `-EACCES`, `-ELOOP`, `-EIO`

Each process also maintains the root directory for locating the files whose name start with a leading '/'. By default the root directory is the root of the file system and can be changed using `chroot` system call.

System Call: `readdir`

Inputs:

`eax`: SYS_readdir

`ebx`: file descriptor

`ecx`: address of directory entry buffer

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EFAULT`, `-EINVAL`, `-ENOENT`, `-ENOTDIR`

The `readir` system call is an old system call and is provided in GNU/Linux for compatibility reasons only. The address in register `ecx` points to a buffer in which the directory entry is returned. The format of the directory entry is available as a C structure definition of `struct dirent` in `/usr/include/linux/dirent.h` file.

System Call: `getdents`

Inputs:

`eax`: SYS_getdents

`ebx`: file descriptor

`ecx`: address of directory entry buffer

`edx`: count

Return Values:

Normal Case: number of bytes read

Error Values: `-EBADF`, `-EFAULT`, `-EINVAL`, `-ENOENT`, `-ENOTDIR`

The `getdents` system call is the modern version of the reading directories. Given the buffer address, as many directory entries are filled in the buffer as possible. The call returns when there are not enough directory entries to be copied to the buffer or when the next directory entry would cause the buffer to overflow. Actual number of bytes filled in the buffer is given as the return value.

GNU libc provides a wrapper function called `readdir` which calls `getdents` system call.

System Call: `getdents64`

Inputs:

`eax`: SYS_getdents64

`ebx`: file descriptor

`ecx`: address of dirent64 buffer

`edx`: count

Return Values:

Normal Case: number of bytes read

Error Values: `-EBADF`, `-EFAULT`, `-EINVAL`, `-ENOENT`, `-ENOTDIR`

The `getdents64` system call is primarily for the large file support. The system call is similar to the `getdents` system call except that the buffer is filled with data whose definition is provided as C datatype `struct dirent64`.

System Call: `mknod`

Inputs:

`eax`: SYS_mknod

`ebx`: address of the pathname string

`ecx`: mode

`edx`: dev

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EINVAL`, `-EEXIST`, `-EFAULT`, `-EACCES`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOTDIR`, `-ENOMEM`, `-EROFS`, `-ELOOP`, `-ENOSPC`

The `mknod` system call is used to create special files such as device files, named pipes etc. Arguments in registers `ecx` and `edx` provide information such as access modes, major minor numbers for devices and other informations related to the type of the file being created.

7.6.2 File system related system calls

System Call: `mount`

Inputs:

`eax`: SYS_mount

`ebx`: address of the device name string

`ecx`: address of the directory name string

`edx`: address of the file system name string

`esi`: mount flags

`edi`: address of file system specific data

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-ENODEV`, `-ENOTBLK`, `-EBUSY`, `-EINVAL`, `-ENOTDIR`, `-EFAULT`, `-ENOMEM`, `-ENAMETOOLONG`, `-ENOENT`, `-ELOOP`, `-EACCES`, `-ENXIO`, `-EMFILE`

The mount system call is used to mount a file system of a specified type (such as `ext2`, `ext3` etc.) to a specified directory of the system wide file structure. Mount flags and data specific to the file system are also specified. More details can be found in manual page `mount`(2).

System Call: `umount`

Inputs:

`eax`: SYS_umount

`ebx`: address of the path name

`ecx`: unmount flags

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EBUSY`, `-EINVAL`, `-ENOTDIR`, `-EFAULT`, `-ENOMEM`, `-ENAMETOOLONG`, `-ENOENT`, `-ELOOP`, `-EACCES`

The `umount` system call is used to unmount previously mounted file system at directory whose name is provided in register `ebx`. Unmount flags specify whether to force unmount a file system when busy (`MNT_FORCE`) and whether to do a lazy umount (`MNT_DETACH`). A combination of these may also be provided. The GNU libc wrapper provides a `umount2` function call for this system call.

System Call: `oldumount`

Inputs:

`eax`: SYS_oldumount

`ebx`: address of the path name

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EBUSY`, `-EINVAL`, `-ENOTDIR`, `-EFAULT`, `-ENOMEM`, `-ENAMETOOLONG`, `-ENOENT`, `-ELOOP`, `-EACCES`

The `oldumount` system call is primarily for compatibility reasons and is similar to `umount` system call except that it does not take any umount flags. The GNU libc wrapper provides a `umount` function call corresponding to this system call.

System Call: `statfs`

Inputs:

`eax`: SYS_statfs

`ebx`: address of pathname string

`ecx`: address of statfs buffer

Return Values:

Normal Case: 0

Error Values: -EACCES, -ELOOP, -ENAMETOOLONG, -ENOENT, -ENOTDIR, -EFAULT, -EINTR, -EIO, -ENOMEM, -ENOSYS, -EOVERFLOW

The statfs system call provides the information about the file system (rather than the file) in which the specified file is stored. It fills in the information in a specific format as C defintion of struct statfs defined in /usr/include/bits/statfs.h.

System Call: fstatfs

Inputs:

eax: SYS_fstatfs

ebx: file descriptor

ecx: address of statfs buffer

Return Values:

Normal Case: 0

Error Values: -EBADF, -EFAULT, -EINTR, -EIO, -ENOMEM, -ENOSYS, -EOVERFLOW

The fstatfs system call is functionally the same as the statfs system call except that it takes an open file descriptor.

System Call: statfs64

Inputs:

eax: SYS_statfs64

ebx: address of pathname string

ecx: address of statfs64 buffer

Return Values:

Normal Case: 0

Error Values: -EACCES, -ELOOP, -ENAMETOOLONG, -ENOENT, -ENOTDIR, -EFAULT, -EINTR, -EIO, -ENOMEM, -ENOSYS, -EOVERFLOW

The statfs64 system call is similar to the statfs system call except that it supports large files and therefore the data structure used is struct statfs64.

System Call: `fstatfs64`

Inputs:

`eax:` SYS_fstatfs64

`ebx:` file descriptor

`ecx:` address of statfs64 buffer

Return Values:

Normal Case: 0

Error Values: `-EBADF`, `-EFAULT`, `-EINTR`, `-EIO`, `-ENOMEM`, `-ENOSYS`, `-EOVERFLOW`

The `fstatfs64` system call is functionally the same as the `statfs64` system call except that it takes an open file descriptor.

7.6.3 Process Information

A process in GNU/Linux has three kinds of user ids – real user ID, effective user ID and saved user ID. In addition it has three kinds of group IDs – real group ID, effective group ID and saved group ID. Processes are identified by a unique process ID. In addition, precesses have a parent child relationship. For job control reasons, the processes may be grouped together and given a unique process group ID. There are various system calls in GNU/Linux that handle the process and process related information.

System Call: `getuid`

Inputs:

`eax:` SYS_getuid

Return Values:

Normal Case: Real user ID of the process

Error Values: No Errors. System call is always successful.

System Call: `geteuid`

Inputs:

`eax:` SYS_geteuid

Return Values:

Normal Case: Effective user ID of the process

Error Values: No Errors. System call is always successful.

System Call: `getgid`

Inputs:

`eax`: SYS_getgid

Return Values:

Normal Case: Group ID of the process

Error Values: No Errors. System call is always successful.

System Call: `getegid`

Inputs:

`eax`: SYS_getegid

Return Values:

Normal Case: Effective group ID of the process

Error Values: No Errors. System call is always successful.

System Call: `getresuid`

Inputs:

`eax`: SYS_getresuid

`ebx`: Address of an integer to get real user ID

`ecx`: Address of an integer to get effective user ID

`edx`: Address of an integer to get saved user ID

Return Values:

Normal Case: 0

Error Values: `-EFAULT`

The `getresuid` system call returns all three kinds of user IDs in memory locations whose addresses are passed to the system call as arguments.

System Call: `getresgid`

Inputs:

`eax`: SYS_getresgid

`ebx`: Address of an integer to get real group ID

`ecx`: Address of an integer to get effective group ID

`edx`: Address of an integer to get saved group ID

Return Values:

Normal Case: 0

Error Values: `-EFAULT`

The `getresgid` system call returns all of three kinds of group IDs of the calling process.

System Call: `getpid`

Inputs:

`eax`: SYS_getpid

Return Values:

Normal Case: Process ID of the calling process

Error Values: No Errors. System call is always successful.

System Call: `getppid`

Inputs:

`eax`: SYS_getppid

Return Values:

Normal Case: Process ID of parent of the calling process

Error Values: No Errors. System call is always successful.

System Call: `getpgid`

Inputs:

`eax`: SYS_getpgid

`ebx`: process ID

Return Values:

Normal Case: Process group ID

Error Values: `-ESRCH`

The `getpgid` system call returns the process group ID for the process whose process ID is passed in register `ebx`. If the value in register `ebx` is passed as 0, the process group ID of the calling process is returned.

System Call: `getpgrp`

Inputs:

`eax`: SYS_getpgrp

Return Values:

Normal Case: Process Group ID

Error Values: No Errors. System call is always successful.

The `getpgrp` system call returns the process group ID of the calling process.

System Call: `getsid`

Inputs:

`eax`: SYS_getsid

`ebx`: Process ID

Return Values:

Normal Case: Session ID

Error Values: `-EPERM`, `-ESRCH`

The `getsid` system call returns the session ID for the process whose process ID is provided in register `ebx`. If a 0 is passed in register `ebx`, the session ID of the calling process is returned.

System Call: `setuid`

Inputs:

`eax`: SYS_setuid

`ebx`: new user ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EAGAIN`

A non-root process can set its effective user ID to new user ID if its saved used ID or the effective user ID is same as the new user ID. Thus this system call is typically used by an suid application to initially provide the privileges based on the owner of the executable file and later drop them to the privileges based on the user ID of the process.

If a root process calls this system call then the real, saved and effective user IDs are all set to the new user ID. Thus an suid program with root ownership can use the root privileges initially and after the use of root privileges is over, it can drop the root privileges permanently.

System Call: `setreuid`

Inputs:

`eax`: SYS_setreuid

`ebx`: real user ID

`ecx`: effective user ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`

The `setreuid` system call sets real and effective user IDs of calling process. If any of the IDs provided is -1, that particular ID remains unchanged. While a root owned process can set any user ID or effective user ID, there are certain restrictions on non-root owned processes. They can provide the value in register `ebx` as real user ID of the process or the effective user ID of the process. All other values result in an error condition. In a similar manner the value provided in register `ecx` can only be the real user ID, the effective user ID or the saved user ID.

System Call: `setresuid`

Inputs:

`eax`: SYS_setresuid

`ebx`: real user ID

`ecx`: effective user ID

`edx`: saved user ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EAGAIN`

The `setresuid` system call sets the real user ID, the effective user ID, and the saved user ID of the calling process. Non-root user processes (*i.e.* processes with each of real, effective and saved user ID nonzero) may change the real, effective and saved user ID each to one of the current user ID, effective user ID or saved user ID.

System Call: `setgid`

Inputs:

`eax`: SYS_setgid

`ebx`: new group ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`

The `setgid` system call is used to set the effective group ID to the specified group ID. A root process can also set the real and saved group IDs to the specified group ID.

System Call: `setregid`

Inputs:

`eax`: SYS_setregid

`ebx`: real group ID

`ecx`: effective group ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`

The `setregid` system call sets real and effective group IDs of calling process. If any of the IDs provided is -1, that particular ID remains unchanged. While a root owned process can provide any value in registers `ebx` or `ecx`, certain security related restrictions are imposed on non-root owned processes. They can provide the value in register `ebx` as real group ID of the process or the effective group ID of the process. Similarly the value provided in register `ecx` can only be one of the real group ID, the effective group ID or the saved group ID.

System Call: `setresgid`

Inputs:

`eax`: SYS_setresgid

`ebx`: real group ID

`ecx`: effective group ID

`edx`: saved group ID

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EAGAIN`

The `setresgid` system call sets the real group ID, the effective group ID, and the saved group ID of the calling process. Non-root user processes (*i.e.* processes with each of real, effective and saved user ID nonzero) may change the real, effective and saved group ID each to one of the current group ID, effective group ID or saved group ID.

System Call: `setpgid`

Inputs:

`eax`: SYS_setpgid

`ebx`: Process ID

`ecx`: Process group ID

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EACCES`, `-EPERM`, `-ESRCH`

The `setpgid` system call is used to set the process group (ID in register `ecx`) of the process with ID in register `ebx`. If `ebx` is 0, the process ID of the calling process is used. If `ecx` is 0, the process ID of the process corresponding to register `ebx` is used. Thus passing both paramters as 0 sets the current process as the leader of a newly created process group with group ID same as the process ID. If a process is moved from one process group to another using this system call, both processes must be in the same session.

System Call: `setsid`

Inputs:

`eax`: SYS_setsid

Return Values:

Normal Case: session ID

Error Values: `-EPERM`

The `setsid` system call is used to create a new session ID for the calling process. All descendants of the calling process will belong to the same session until one of those processes sets a new session by making a system call `setsid`.

System Call: `times`

Inputs:

`eax`: SYS_times

`ebx`: Address of the `struct tms` buffer

Return Values:

Normal Case: Number of clock ticks since system reboot

Error Values: `-EFAULT`

The `times` system call returns the user time and system time of the calling process and accumulated user and system times of all children processes. All times are in unit of clock ticks.

System Call: `getcwd`

Inputs:

`eax:` SYS_getcwd

`ebx:` Buffer address

`ecx:` Buffer size

Return Values:

Normal Case: Length of the buffer filled

Error Values: `-EACCES`, `-EFAULT`, `-EINVAL`, `-ENOENT`, `-ERANGE`

The `getcwd` system call fills the buffer with an absolute path of the current working directory for the calling process subject to a maximum size passed as an argument in register `ecx`.

The system call returns the number of bytes filled in the buffer. The corresponding libc wrapper for this system call returns a buffer pointer in case of success and NULL in case of failure.

System Call: `getgroups`

Inputs:

`eax:` SYS_getgroups

`ebx:` Size of integer array

`ecx:` Address of integer array to receive list of groups

Return Values:

Normal Case: Number of groups

Error Values: `-EFAULT`, `-EINVAL`

Each process may be in one of many supplementary groups. The `getgroups` system call provides a list of those group IDs upto a maximum size as specified in register `ebx`. Return value provides the actual number of entries in the array as filled by the system call.

System Call: `setgroups`

Inputs:

`eax`: SYS_setgroups

`ebx`: Number of entries in the integer array

`ecx`: Address of integer array providing list of groups

Return Values:

Normal Case: 0

Error Values: `-EFAULT`, `-EPERM`, `-EINVAL`

The `setgroups` system call sets a list of supplementary groups for the calling process. Only the root user can make this system call.

7.6.4 Process Management

System Call: `fork`

Inputs:

`eax`: SYS_fork

Return Values:

Normal Case: Child process ID in the parent process and 0 in the child process

Error Values: `-EAGAIN`, `-ENOMEM`

The `fork` system call creates a new process. The calling process also becomes the parent of the newly created process. The child process so created has its own address space which is initialized to the value from the parent process. Various files opened in the parent process are shared by the child as open files. The child process has its own process ID and has its parent process ID as the process ID of the calling process.

Newly created process belongs to the same session and process group as the parent.

This system call has two returns, one each in the parent and child processes. To the parent process the return value is the process ID of the child while to the child process the return value is 0. In case of an error, no child process is created.

System Call: `vfork`

Inputs:

`eax:` SYS_vfork

Return Values:

Normal Case: Child process ID in the parent process and 0 in the child process

Error Values: `-EAGAIN`, `-ENOMEM`

The `vfork` system call is similar to the `fork` system call except that the newly created process is not completely initialized. In Linux, the most common use of process creation is to execute another program. In such cases, the `fork` system call carries extra burden of initializing the address space of the child process which would be re-initialized at the time of loading the new program. Under such cases, `vfork` system call provides an efficient way of creating process which must be used only to load another program.

System Call: `clone`

Inputs:

`eax:` SYS_clone

`ebx:` clone flags

`ecx:` new stack pointer

`edx:` address where parent thread ID is returned

`esi:` address where child thread ID is returned

Return Values:

Normal Case: Child process ID in the parent process and 0 in the child process

Error Values: `-EAGAIN`, `-ENOMEM`, `-EINVAL`, `-EPERM`

The `clone` system call is a versatile mechanism to create a new process. The creation and characteristics of the newly created process can be controlled by a combination of several flags which provide the mechanisms for the following.

`CLONE_PARENT`**:** The newly created process has the same parent as that of the calling process. If not set, parent of the newly created process will be the calling process.

`CLONE_FS`: The newly created process shares the file system specific entities such as root of the file system, current working directory and umask. If not set, these entities are duplicated from the calling process.

`CLONE_FILES`: The newly created process shared the file descriptors as with the calling process. If not set, the file descriptors are duplicated into the newly created process.

`CLONE_SIGHAND`: Share the signal handlers between two processes. The signal handlers are copied from the calling process if this flag is not specified.

`CLONE_VM`: Share all virtual memory between the two processes. If not set, the new process has its own address space that is initialized from the parent.

The address of the memory where the stack for the new process should be is given in register `ecx`. The new process may have same stack address (and even share stack if `CLONE_VM` is specified) by setting register `ecx` to 0.

If two processes share the same stack, results are going to be undefined and may cause weird errors.

System Call: `execve`

Inputs:

`eax`: SYS_execve

`ebx`: address of pathname string

`ecx`: address of an array of multiple command line arguments

`edx`: address of an array of multiple environment strings

Return Values:

Normal Case: Does not return

Error Values: `-EACCES`, `-EPERM`, `-E2BIG`, `-ENOEXEC`, `-EFAULT`, `-ENAMETOOLONG`, `-ENOENT`, `-ENOMEM`, `-ENOTDIR`, `-ELOOP`, `-ETXTBSY`, `-EIO`, `-ENFILE`, `-EMFILE`, `-EINVAL`, `-EISDIR`, `-ELIBBAD`

The `execve` system call initializes the memory map of the calling process by new program loaded from an executable file given in register `ebx`. Register `ecx` contains an address of an array of string addresses. Each of these string addresses are pushed on the process stack so

they are available as argument to the starting point in the program. Command line arguments are ended by a null address to delimit the command line arguments from environment strings.

System Call: `exit`

Inputs:

`eax`: SYS_exit

`ebx`: exit_code

Return Values:

Normal Case: Does not return

Error Values: No Errors. System call is always successful.

The `exit` system call is used to terminate the calling process. The exit_code in register `ebx` is then available to the parent process when it calls a `wait` kind of system call.

System Call: `waitpid`

Inputs:

`eax`: SYS_waitpid

`ebx`: ID

`ecx`: memory address to store status

`edx`: options

Return Values:

Normal Case: Process ID of child

Error Values: `-ECHILD`, `-EINVAL`, `-EINTR`

The `waitpid` system call is used to wait for a named child process to terminate. If the specified child process had already terminated, the system call returns immediately with the exit status of child process. The `waitpid` system call may also terminate due to a signal delivered to the calling process in which case, the status indicates appropriate values. The ID argument in register `ebx` is coded in the following ways.

< -1: ID indicates negative of a process group ID. Any child process whose process group ID is equal to the absolute value of the specified ID would cause this system call to terminate.

-1: Wait for any child process to terminate.

0: Wait for any child process to terminate whose process group is the same as of the calling process.

> 0: Wait for the termination of child process with process ID as specified ID.

System Call: `pause`

Inputs:

`eax`: SYS_pause

Return Values:

Normal Case: Does not return normally

Error Values: `-EINTR`

The `pause` system call makes the calling process wait for a signal to be delivered. It therefore does not return until a signal is delivered and in that case it returns an error value $-$`EINTR`.

System Call: `kill`

Inputs:

`eax`: SYS_kill

`ebx`: ID

`ecx`: signal

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-ESRCH`, `-EPERM`

The `kill` system call is used to dispatch a signal (signal type in register `ecx`) to the named process or processes. ID argument in register `ebx` is interpreted as follows.

> 0: The signal is delivered to a process with process ID as value in `ebx`.

0: The signal is delivered to every process in the process group of the calling process.

-1: The signal is sent to every process that the calling process has permission to send signals to. The `init` process with process ID 1 is also not sent any signal in GNU/Linux.

< -1: The signal is sent to every process in the process group ($-$ID).

System Call: `signal`

Inputs:

`eax`: SYS_signal

`ebx`: signal type

`ecx`: address of signal handler function

Return Values:

Normal Case: Previous signal handler

Error Values: `-EINVAL`, `-ERESTARTNOINTR`

The `signal` system call sets up a signal handler for the specified signal. The address of the signal handler in register `ecx` can be a constant `SIG_IGN` or `SIG_DFL`, or an address of a function. When a signal is delivered to a process, it is ignored if the corresponding signal handler is `SIG_IGN`, a default action is taken if the corresponding signal handler is `SIG_DFL` or the specified function is executed.

Some signals such as `SIGKILL` and `SIGSTOP` can not be caught or ignored. Only the default action for these signals can be carried out.

System Call: `sigaction`

Inputs:

`eax`: SYS_sigaction

`ebx`: signal type

`ecx`: Address of new signal action `struct sigaction`

`edx`: Address where the old signal action is returned in `struct sigaction`

Return Values:

Normal Case: 0

Error Values: `-EFAULT`, `-EINVAL`, `-ERESTARTNOINTR`

The `sigaction` system call is more flexible than the `signal` system call for specifying the signal handlers. The addresses in registers `ecx` and `edx` point to memory of type `struct sigaction`. Values pointed by register `ecx` are the new signal action specifications while the old values are returned in memory area pointed to by register `edx`.

Signals `SIGKILL` and `SIGSTOP` can not be caught or ignored.

System Call: `alarm`

Inputs:

`eax`: SYS_alarm

`ebx`: seconds

Return Values:

Normal Case: Number of seconds remaining from previous alarm or 0

Error Values: No Errors. System call is always successful.

The `alarm` system call causes a `SIGALARM` signal to be delivered to the calling process after the specified number of seconds. If there was an alarm set previously, it is canceled and the time remaining is returned. Otherwise the return value is 0.

System Call: `nanosleep`

Inputs:

`eax`: SYS_nanosleep

`ebx`: memory address of requested wait time data

`ecx`: memory address of remaining wait time data

Return Values:

Normal Case: 0

Error Values: `-EINTR`, `-EINVAL`, `-EFAULT`

The `nanosleep` system call delays the execution of the calling process by the specified time. The system call may be interrupted due to delivery of a signal. In that case, it fills in the remaining time in the address provided in register `ecx` provided it is not NULL, and returns $-$`EINTR`. Registers `ebx` and `ecx` both have addresses of time specification as given by C data type `struct timespec`. This data structure can contain the time in units of nanoseconds.

System Call: `umask`

Inputs:

`eax`: SYS_umask

`ebx`: mask

Return Values:

Normal Case: old mask

Error Values: No Errors. System call is always successful.

This system call sets a new umask for the calling process. The umask is used by the `open` and `creat` system calls to modulate the mode for the new files.

System Call: `nice`

Inputs:

`eax`: SYS_nice

`ebx`: increment

Return Values:

Normal Case: 0

Error Values: `-EPERM`

The `nice` system call can change the priority (nice value) of the calling process. A normal process can only provide a positive increment (to lower the priority). Only the superuser may provide a negative increment to raise the priority.

System Call: `setpriority`

Inputs:

`eax`: SYS_setpriority

`ebx`: Constant to indicate kind of processes

`ecx`: ID

`edx`: nice value

Return Values:

Normal Case: 0

Error Values: `-ESRCH`, `-EINVAL`, `-EPERM`, `-EACCES`

The `setpriority` system call sets the default priority of various kinds of processes. Value in regsiter `ebx` can be one of the following.

`PRIO_PROCESS` to indicate that the priority is to be set for the process whose process ID is given in register `ecx` or for the calling process when `ecx` is 0.

PRIO_PGRP to indicate that the priority is to be set for processes in the process group whose ID is given in register ecx or for processes in the process group of the calling process when ecx is 0.

PRIO_USER to indicate that the priority is to be set for all processes belonging to a user whose user ID is given in ecx or for the user of the calling process when ecx is 0.

System Call: getpriority

Inputs:

eax: SYS_getpriority

ebx: Constant to indicate kind of processes

ecx: ID

Return Values:

Normal Case: 20+nice value

Error Values: -ESRCH, -EINVAL, -EPERM, -EACCES

The getpriority system call returns the priority of the names processes. The two arguments in registers ebx and ecx have same meaning as in setpriority system call.

Nice value of a process can be between −20 to 19. In order to avoid negative values for the normal case, getpriority system call returns 20 added to nice value. Thus the return value is always positive or zero for the normal case.

7.6.5 Inter-process interaction

System Call: pipe

Inputs:

eax: SYS_pipe

ebx: address of an array of two integers

Return Values:

Normal Case: 0

Error Values: -EMFILE, -ENFILE, -EFAULT

The `pipe` system call is used to create a pipe with a pair of open file descriptors for reading from and writing into the pipe. The value provided in the first location of the array is read file descriptor while the value in the second location is write file descriptor.

This mechanism is commonly used to create communication channel between two related (parent and child) processes. For example, a shell uses this mechanism to create a pipeline of commands where output of one command is redirected to the input of another.

System Call: `ipc`

Inputs:

`eax:` SYS_ipc

`ebx:` IPC call type

`ecx:` first argument

`edx:` second argument

`esi:` third argument

`edi:` address of buffer as fourth argument

`ebp:` fifth argument

Return Values:

Normal Case: non-negative value as per IPC call type

Error Values: `-EINVAL`, `-EFAULT`, `-ENOSYS`, `-E2BIG`, `-ENOMEM`, `-EIDRM`, `-EACCES`, `-EINTR`, `-ERANGE`, `-EPERM`. `EAGAIN`, `-ENOMSG`, `-EEXIST`, `-ENOENT`, `-ENOSPC`

GNU/Linux operating system provides IPC support for semaphore, messages and shared memory constructs. A single system call `ipc` is used for all such IPC mechanisms. Parameter in `ebx` distinguishes between various IPC operations and rest of the arguments are based on the IPC operation chosen.

GNU libc wrapper provides the following function calls all of which use `ipc` system call.

`semget` – Create a semaphore
`semop` – Semaphre operations
`semctl` – Semaphore control
`msgget` – Get a message box
`msgsnd` – Message send
`msgrcv` – Receive message
`msgctl` – Message control
`shmget` – Get a shared memory segment
`shmat` – Attach shared memory segment in address space
`shmdt` – Detach shared memory segment
`shmctl` – Shared memory segment control

System Call: `socketcall`

Inputs:

`eax:` SYS_socketcall

`ebx:` call type

`ecx:` address of argument array

Return Values:

Normal Case: non-negative return value as per the call type

Error Values: `-EAGAIN`, `-EWOULDBLOCK`, `-EBADF`, `-ENOTSOCK`, `-EOPNOTSUPP`, `-EINTR`, `-ECONNABORTED`, `-EINVAL`, `-EMFILE`, `-ENFILE`, `-EFAULT`, `-ENOBUFS`, `-ENOMEM`, `-EPROTO`, `-EPERM`, `-ENOSR`, `-ESOCKTNOSUPPORT`, `-EPROTONOSUPPORT`, `-ETIMEDOUT`, `-ERESTARTSYS`, `-EACCES`, `-EROFS`, `-ENAMETOOLONG`, `-ENOTDIR`, `-ELOOP`, `-EISCONN`, `-ECONNREFUSED`, `-ETIMEDOUT`, `-ENETUNREACH`, `-EADDRINUSE`, `-EINPROGRESS`, `-EALREADY`, `-EAFNOSUPPORT`, `-EACCES`, `-ENOTSOCK`, `-ENOTCONN`, `-EFAULT`, `-ENOPROTOOPT`, `-ECONNRESET`, `-EDESTADDRREQ`, `-EMSGSIZE`, `-EPIPE`

The `socketcall` system call is a single system call for the following GNU libc functions.

`accept`– accept a connection on a socket
`bind`– bind a name to a socket
`connect`– initiate a connection on a socket
`getpeername`– get name of connected peer socket
`getsockname`– get socket name
`getsockopt`– get options on socket
`listen`– listen for connections on a socket
`recv`– receive a message from connected socket
`recvfrom`– receive a message from any kind of socket
`send`– send message to a connected socket
`sendto`– send message to any kind of socket
`setsockopt`– set options on sockets
`shutdown`– partly or fully close a full-duplex connection
`socket`– create a socket (communication endpoint)
`socketpair`– create a pair of connected sockets

7.6.6 Input Output related system calls

In GNU/Linux, input and output is performed typically using operating system supported mechanisms. These mechanisms include read and write using `read` and `write` system calls. The I/O devices are then considered as a stream of bytes. Most I/O devices however are not seekable and therefore system calls such as `lseek` may not work.

In order to perform an I/O, a device file must be opened using `open` system call. A file descriptor obtained by use of the `open` system call can be used for almost all file related commands. However there are certain operations that do not fit into this kind of mechanisms. These include operations such as configuring a device to do operations in certain ways. The GNU/Linux operating system provides a system call `ioctl` for such configurations of the device and device drivers. However the interpretation of its arguments are device specific.

In addition to performing input and output using read and write system calls, GNU/Linux on IA32 processors also provides a direct input output mechanisms. This mechanism is available for devices which are mapped on the I/O address space of the processor (as will be discussed in chapter 8). Under this mechanism, the Operating System kernel may provide selective access permissions on certain I/O port locations. In addition to this, the Operating System kernel may increase the I/O privileges of a process to be able to perform I/O on any port. Both of these operations are permitted only to the root user. However root user may pass such privileges to any other user by setting the user and group IDs of the process to some thing other than the root.

Use of such mechanisms are discussed in chapter 8.

System Call: `ioctl`

Inputs:

`eax`: SYS_ioctl

`ebx`: file descriptor

`ecx`: command

`edx`: argument specific to the command (usually an address)

Return Values:

Normal Case: 0 or a positive value dependent upon the command

Error Values: `-EBADF`, `-EFAULT`, `-ENOTTY`, `-EINVAL`

The `ioctl` system call is used to manipulate the device parameters or configure the device in certain ways using device special files. In particular, many operating characteristics of character special files such as terminals may be controlled with ioctl requests.

Parameters in registers `ecx` and `edx` are device driver specific. The return value of the `ioctl` system call is also device driver specific.

System Call: `ioperm`

Inputs:

`eax`: SYS_ioperm

`ebx`: Start I/O port address

`ecx`: number of I/O ports

`edx`: turn on value

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EPERM`, `-EIO`

The `ioperm` system call is used to grant permissions to do the input output directly by a process bypassing the device drivers. In a single call to this system call a range of I/O addresses may be turned on or off based on the parameter in register `edx`. In IA32 processors, processes may be selectively granted I/O permission for I/O addresses between 0 and 0x3FF. For I/O addresses outside this range the only option is to permit input output on all I/O addresses.

System Call: `iopl`

Inputs:

`eax:` SYS_iopl

`ebx:` IOPL level

Return Values:

Normal Case: 0

Error Values: `-EINVAL, -EPERM`

The `iopl` system call is used to raise the input output privilege level of a process. The level 0 is to withdraw all input output privileges (as for the normal processes) while level 3 is used to provide blanket permission to perform input output. The levels 1 and 2 are not used in GNU/Linux operating system.

7.6.7 Memory Management

System Call: `brk`

Inputs:

`eax:` SYS_brk

`ebx:` New data address

Return Values:

Normal Case: 0

Error Values: `-ENOMEM`

The `brk` system call is used to change the data segment allocation of the process. When a program is loaded in the memory using `execve` system call, the system allocates a data segment based on the values in the executable file. This may be increased using `brk` system call.

The `brk` system call is almost like memory allocation but it must be distinguished from `malloc` function call of the libc which allocates memory only within its data allocation.

System Call: `mmap2`

Inputs:

`eax`: SYS_mmap2

`ebx`: Memory address

`ecx`: length

`edx`: protection

`esi`: mapping mode flags

`edi`: file descriptor

`ebp`: offset within the file in units of page size

Return Values:

Normal Case: address where the file is mapped

Error Values: `-EBADF`, `-ENODEV`, `-EPERM`, `-EINVAL`, `-ENOMEM`, `-EAGAIN`, `-EACCES`, `-ETXTBSY`

The `mmap2` system call maps a file into the user address space. Offset from where the file is mapped is specified in units of page sizes and is usually zero (to specify the beginning of the file). Register `ebx` contains an address in the VM address space of the process. Both memory address and length must be page aligned and if they are not, the file is mapped at closest address that is page aligned. Memory address, length, file offset etc. are taken as hint to the system call. The real address where the file is mapped is returned as the return value of the system call.

Protection argument in register `edx` and mapping mode flags in registr `esi` specify the way the file is mapped and the access controls available (read, write, execute etc.). The protection argument must agree with the file opening mode and whether the file is execuable or not.

System Call: `old_mmap`

Inputs:

`eax`: SYS_old_mmap

`ebx`: address of memory map argument structure

Return Values:

Normal Case: address where the file is mapped

Error Values: `-EBADF`, `-ENODEV`, `-EPERM`, `-EINVAL`, `-ENOMEM`, `-EAGAIN`, `-EACCES`, `-ETXTBSY`, `-EFAULT`

The old_mmap system call is similar to the mmap2 system call except that it takes its arguments in memory and the address of the memory is passed as argument in register ebx. In addition, the file offset for this call is specified in units of bytes and must be page aligned. The arguments are passed in memory with layout as in C data type struct mmap_arg_struct.

GNU libc wrapper provides a function mmap as the front end of this system call. The function takes all the six arguments separately rather than in a memory structure.

System Call: munmap

Inputs:

eax: SYS_munmap

ebx: address

ecx: length

Return Values:

Normal Case: 0

Error Values: -EINVAL, -ENOMEM, -EAGAIN

The munmap system call is used to unmap a virtual address block. Both address and length must be page size aligned. A part of the previously mapped area may also be unmapped using this system call.

System Call: mremap

Inputs:

eax: SYS_mremap

ebx: old address

ecx: old length

edx: new length

esi: flags

edi: new address

Return Values:

Normal Case: mapped address

Error Values: -EINVAL, -EFAULT, -EAGAIN, -ENOMEM

The mremap system call is used to map an existing memory segment to new segment. The most common use of this system call is to grow or shrink an existing memory segment. The GNU libc wrapper does not take the new address argument and can only be used to grow or shrink the memory segment.

System Call: mlock

Inputs:

eax: SYS_mlock

ebx: address

ecx: length

Return Values:

Normal Case: 0

Error Values: -ENOMEM, -EPERM, -EINVAL

The mlock system call is used by a root owned process to lock certain number of pages in the memory. These pages will not be swapped out till they have been unlocked.

System Call: munlock

Inputs:

eax: SYS_munlock

ebx: address

ecx: length

Return Values:

Normal Case: 0

Error Values: -ENOMEM, -EINVAL

The munlock system call is used to re-enable the paging activity for the specified address range.

System Call: mlockall

Inputs:

`eax`: SYS_mlockall

`ebx`: flags

Return Values:

Normal Case: 0

Error Values: `-ENOMEM`, `-EPERM`, `-EINVAL`

The `mlockall` system call is used to lock all pages of the calling process. The process must be owned by the root. Argument in register `ebx` indicates the way locking will be affected.

System Call: `munlockall`

Inputs:

`eax`: SYS_munlockall

Return Values:

Normal Case: 0

Error Values: No Errors. System call is always successful.

The `munlockall` system call unlocks all pages in the address space of the calling process and enables paging.

7.6.8 Other system calls

System Call: `time`

Inputs:

`eax`: SYS_time

`ebx`: address of an integer

Return Values:

Normal Case: time in seconds since EPOCH

Error Values: `-EFAULT`

The `time` system call returns the number of seconds elapsed since 00:00:00, January 1, 1970, also known as EPOCH. If the address in register `ebx` is non-NULL, the returned value is also stored in that location.

System Call: `stime`

Inputs:

`eax`: SYS_stime

`ebx`: address of an integer

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EFAULT`

The `stime` system call sets the system time. The address in register `ebx` points to an integer whose value is the number of seconds elapsed since EPOCH.

System Call: `gettimeofday`

Inputs:

`eax`: SYS_gettimeofday

`ebx`: address of `struct timeval` variable

`ecx`: address of `struct timezone` variable

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EFAULT`

The `gettimeofday` system call returns the time in units of microseconds since EPOCH and the timezone information previously set using `settimeofday` system call.

System Call: `settimeofday`

Inputs:

`eax`: SYS_settimeofday

`ebx`: address of `struct timeval` variable

`ecx`: address of `struct timezone` variable

Return Values:

Normal Case: 0

Error Values: `-EPERM`, `-EINVAL`, `-EFAULT`

The `settimeofday` system call takes the time in units of microseconds since EPOCH and sets the system time. The timezone information is not used in any meaningful manner in GNU/Linux but must be a valid address or NULL. The timezone information is saved and returned when `gettimeofday` system call is used.

System Call: `sethostname`

Inputs:

`eax`: SYS_sethostname

`ebx`: address of name string

`ecx`: length

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EPERM`, `-EFAULT`

The `sethostname` system call is used by a root owned process to set the name of the host to given string. Length of the string buffer is also provided.

System Call: `gethostname`

Inputs:

`eax`: SYS_gethostname

`ebx`: address of name string

`ecx`: length

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EFAULT`

The `gethostname` system call is used to get the name of the host. Length of the string buffer is also provided.

System Call: `setdomainname`

Inputs:

eax: SYS_setdomainname

ebx: address of name string

ecx: length

Return Values:

Normal Case: 0

Error Values: -EINVAL, -EPERM, -EFAULT

The setdomainname system call sets the domain name for the machine and can be used only by a root owned process.

System Call: newuname

Inputs:

eax: SYS_newuname

ebx: address in memory for struct new_utsname variable

Return Values:

Normal Case: 0

Error Values: -EFAULT

The newuname system call provides system specific information such as operating system name, hostname, operating system release, operating system version, domainname etc. The system call takes an address of the following data structure.

```
struct new_utsname {
    char sysname[65];
    char nodename[65];
    char release[65];
    char version[65];
    char machine[65];
    char domainname[65];
};
```

All string are returned with null termination.

The GNU libc wrapper provides a uname function call as a front end to the newuname system call. Argument to GNU libc wrapper is an address of struct utsname which is similar to the struct new_utsname but provides points rather than the array itself. There is no system call for getdomainname and GNU libc wrapper provides this function by calling newuname system call.

System Call: `reboot`

Inputs:

`eax`: SYS_reboot

`ebx`: magic1

`ecx`: magic2

`edx`: command

`esi`: address to arguments

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EPERM`, `-EFAULT`

The `reboot` system call may be used by root owned process to reboot machine in a variety of ways as defined by `command`. It can also be used to enable or disable the `Ctrl-ALT-Delete` key sequence (CAD key sequence).

The value in `ebx` register must be `LINUX_REBOOT_MAGIC1`, which is a constant 0xfee1dead. The magic2 argument in `ecx` register can be one of the following constants.

`LINUX_REBOOT_MAGIC2` (*i.e.* 0x28121969)
`LINUX_REBOOT_MAGIC2A` (*i.e.* 0x5121996)
`LINUX_REBOOT_MAGIC2B` (*i.e.* 0x16041998)
`LINUX_REBOOT_MAGIC2C` (*i.e.* 0x20112000)

Value in register `edx` provides a command to `reboot` system call and can be one of the following.

`LINUX_REBOOT_CMD_RESTART` (0x1234567): Restart system.

`LINUX_REBOOT_CMD_HALT` (0xcdef0123): Halt system.

`LINUX_REBOOT_CMD_POWER_OFF` (0x4321fedc): Switch off power if possible.

`LINUX_REBOOT_CMD_RESTART2` (0xa1b2c3d4): Restart system with command in `esi`.

`LINUX_REBOOT_CMD_CAD_ON` (0x89abcdef): Enable CAD key sequence for reboot.

`LINUX_REBOOT_CMD_CAD_OFF` (0): Disable CAD key sequence.

7.7 Example of using system calls

In this section, we shall work out a program, wholly in Assembly language, that prints all its command line arguments and environment variables.

The program is first preprocessed with `cpp` program. We need this preprocessing to be able to use constants such as `SYS_write` for making a `write` system call.

We also define a constant called `STDOUT` to be used as a file descriptor for `write` system call. The GNU/Linux operating system uses a file descriptor of 1 as standard output file descriptor and therefore `STDOUT` is defined as constant 1.

On line 3, the program uses a assembler pseudo op `.data` to describe that subsequent program lines in the Assembly program define data. More about the assembler pseudo ops is described in chapter 11.

In our program, we print all command line arguments and environment variables one on a line. Therefore at the end of printing one command line argument or environment string, we need to print a new line character. This new line character '`\n`' is put into the data section of the program using `.ascii` assembler pseudo op in line 5.

Finally the program starts from line 7. The `.globl` pseudo op makes the label `_start` a global label so that linker can know about it. The linker uses this global label as the start address of the program.

At the entry of the program, stack will have several parameters pushed by the `execve` system call. Top of the stack will contain the number of command line arguments (`argc`). In our program we will not be using this parameter. Subsequent locations on the stack will contain the address of each command line argument starting at locations `4(%esp)` onwards. Following the address of the last command line argument, a 32-bit number 0 is put on the stack by the `execve` system call. Following this the addresses of environment strings are put on the stack. Finally a 32-bit number 0 is put on the stack to mark the end of the environment strings.

Program uses `esi` register as an index on the stack in such a way that `(%esp, %esi)` refers to the address of the string that is printed. The address is copied to register `ecx` in line 29.

If the value in register `ecx` is 0, it marks the separation between command line arguments and environment strings and the program control is transfered to `envp_print` to start printing the environment strings.

Program also uses an Assembly function function `str_len` to find the length of the string whose address is given in register `ecx`. The value is returned in register `edx` so as to make it compatible to the `write` system call which expects the length of the string in register `edx`.

After printing a string, the program also prints new line string. The new line string is a single byte string whose content is character '\n'.

```
#include <syscall.h>
#define STDOUT 1
.data
newline:
        .ascii "\n"
.text
.globl _start
_start:
//
// The programs are started with arguments and
// environment strings put on the stack when the
// kernel loads a program using execve system
// call. At the entry of the program the stack
// layout is
// (esp): argc  -- Number of arguments
// 4(esp): argv[0] -- Address of program name string
// 8(esp): argv[1] -- Address of next arg string
// ...
// NULL         -- Command line args ended.
//              -- Environment strings next on stack
// Environment strings
// NULL         -- End of all strings.
//
// We need not use argc. The offset of argv[0] is 4.
        mov     $4, %esi // Offset for arg in esi
nextarg:
        // Load the address of the argument string
        // in register ecx
        mov     (%esp, %esi), %ecx
        add     $4, %esi // Offset for next arg
        cmp     $0, %ecx // Check if end of args
        je      envp_print
        // Command line argument is to be printed.
        // We need the length of the string.
        // Get length in %edx
        call    str_len
        // At this point, ecx is the address of
        // commnd arg string, edx = length to print
        movl    $(SYS_write),%eax
        movl    $(STDOUT),%ebx
        // Make a write system call
        int     $0x80
        // Print new line character.
```

```
44          // Need to reload eax since it changes
45          // to the return value of the syscall
46          movl    $(SYS_write),%eax
47          movl    $newline, %ecx // Buffer address
48          movl    $1, %edx       // Length to print
49          int     $0x80
50          // Print the next command line argument
51          jmp     nextarg
52          // All command line arguments are printed.
53          // Print the environment string.
54          // esi is the offset on stack for the first
55          // environment string.
56  envp_print:
57          // Environment string address in ecx
58          mov     (%esp, %esi), %ecx
59          add     $4, %esi
60          cmp     $0, %ecx // Check if all done
61          je      done
62          // Get length of string in edx
63          call    str_len
64          // Print the environment string
65          movl    $(SYS_write),%eax
66          movl    $(STDOUT),%ebx
67          int     $0x80
68          // Print new line
69          movl    $(SYS_write),%eax
70          movl    $newline, %ecx
71          movl    $1, %edx
72          int     $0x80
73          // Jump to print next environment string
74          jmp     envp_print
75          // All done. Make an exit system call.
76  done:
77          movl    $(SYS_exit),%eax
78          xorl    %ebx,%ebx
79          int     $0x80
80  // Assembly language function to find the length
81  // of a string.
82  // Input: ecx is address of the string
83  // Output: edx is the length of the string
84  // Registers modified: al, edi
85  str_len:
86
87  // We use string instructions to look for a null
88  // Character and find the length by start address
89  // of the string (ecx) from the address of the
```

```
90  // null character.
91
92          mov     %ecx, %edi
93          cld
94          xor     %al, %al
95          push    %ecx    // Save ecx as repne
96          mov     $5000, %ecx // Max length of string
97          repne   scasb   // modifies it.
98          pop     %ecx    // Restore ecx
99          mov     %edi, %edx
100         sub     %ecx, %edx
101         ret
```

EXERCISES:

7.1 Write an Assembly language function that prints the value of a signed integer given in register `eax`. For doing this, you will need to define an array of characters in which the digits of the integer converted to their ASCII equivalents will be put it. Later this string will be printed with `write` system call.

7.2 Write an Assembly language program that takes one command line argument as the name of a file and checks if that file can be opened for reading or not. If it can be read, the program prints "file xyz is readable." Otherwise it prints a message that the file is unreadable. You will need to use an `open` system call with `O_RDONLY` flag. In this case, the `mode` argument in register `edx` is ignored.

Chapter 8

Input-Output in Linux

There are several ways of performing an input-output operation in GNU/Linux. The operating system kernel provides standard device drivers for devices such as keyboard and display (together known as tty), virtual terminals, serial port drivers, USB drivers, parallel port drivers, Ethernet controller drivers, disk controller drivers etc. Such drivers are identified by the operating system kernel by major and minor numbers. Using `open`, `read`, `write` and `ioctl` system calls, input from the device and output to the device can be achieved. Device drivers typically provide a high level abstraction of physical devices. For example, there is no direct way of operating on the disk controllers using device drivers. The Operating System typically provides an abstraction of storage as collection of files and directories.

Another method to perform an input-output operation is by using the processor instructions. This provides an ultimate control over the devices and the input-output operation can be achieved even without following the semantics provided by the operating system.

In this chapter, we shall explore these two methods of input and output operations with the primary focus of using direct input-output operation with the use of processor instructions.

8.1 Device Drivers in Linux

Device drivers in GNU/Linux are part of the Operating System Kernel. Various functionalities of the device drivers are invoked implicitly when a process makes a system call such as `open`, `read`, `write` or `ioctl`. Internally to the kernel, the device drivers are identified by three attributes, type of the device (a character device or a block device) and two numbers called major and minor numbers.

The `open` system call takes a special device file name as an argument. The file attributes for this device file name provide the values of

major number, minor number and type of the device. As an example, the following is a device file entry in GNU/Linux. This entry can be seen by issuing a command `ls -l /dev/misc/rtc`.

```
crw-------  1 root video 10, 135 Jan 1 1970 /dev/misc/rtc
```

This device is represented by a file `/dev/misc/rtc` which is owned by `root` user and `video` group. It is a character device as indicated by a 'c' as the leading character in permission mode field. The device can be read or written by the owner (`root` in this case) only. Major and minor numbers of this device are 10 and 135 respectively.

When a system call such as `read` and `write` is executed by the process, the device specific read and write operations are carried out by the kernel device drivers.

Some device drivers are implicitly used by the kernel. For example, by doing a read or write of a file, the kernel may cause the disk device driver to be implicitly invoked. Similarly while operating on a network socket, the Ethernet device driver may implicitly be invoked by the kernel.

An abstraction like this provides a unified approach for handling devices as well as files. As a programmer, one is spared of implementing aspects such as low level synchronization details, handling of interrupts raised by the devices and such other fine details of an input-output operation.

However using such an abstraction also makes the input-output operations slow. Each time a process makes a system call (`read` and `write`) to operate on device, execution mode is switched from user level to the kernel level. Kernel then makes an internal call to the generic read or write functions. The generic read and write functions identify associated attributes for the file and call device specific read and write functions. In addition to these, certain file attributes may be checked each time a read or write is performed.

Some of the efficiency issues are handled by buffering and providing large units of data transfers between the device driver and user process. In this way the overheads due to the system call get divided over a number of bytes. For example, while the process may request one byte to be read from a file, the drivers in the OS would read one block of data in the buffer making the subsequent read and write operations to just return data from the buffer.

Certain operations such as those for configuring the device, changing parameters in the device driver or performing small amount of input and output such as setting a particular device register or reading a special device register are handled using `ioctl` calls.

An abstraction like this even allows the operating system to provide virtual devices. Virtual devices do not exist physically and provide functionality that are implemented purely in software by device drivers.

For example, GNU/Linux provides a random number generator as a device. This is purely a software solution and requires no special device hardware. The random number generator is implemented by a character device with major and minor numbers as 1 and 8 respectively and is provided by a device file `/dev/random`. A read from this device provides a pseudo random number byte. In a similar manner, devices such as `/dev/null`, `/dev/pty/*`, `/dev/zero` and `/dev/mem` are all virtual devices that provide certain functionality without any physical device.

8.2 Input-Output Addressing

The physical devices are connected to the CPU on the bus. The CPU can transfer data to a device thereby initiating the hardware device to perform certain operations. The CPU can also read device data thereby implementing an input from the device.

In order to perform such input and output operations, the CPU needs to address the device similar to the way it addresses the memory. Thus each input-output device has a set of addresses to which it is mapped in the address space of the processor.

IA32 processors provide two separate address spaces, one intended for the input-output devices and another for memory. A physical I/O device may be mapped to any address space. There are separate sets of instructions to operate in these address spaces. All IA32 instructions seen till now operate in the memory address space. When an operand of an instruction is in memory, the processor reads or writes into the memory address space. Instructions that operate on the I/O address space are discussed in this chapter. There are certain differences in these two address spaces. Unlike memory address space, data in the I/O address space is never cached in processor caches. An I/O operation takes place immediately after an instruction is executed to read or write in I/O address space. This kind of behavior is essential for the I/O devices as explained with an example.

Many times a single I/O address space is used to transfer a sequence of bytes to the device. For example, let's consider an Ethernet controller. The data packet that is transmitted over the communication network is given by the processor to the Ethernet controller. This is typically achieved by outputting the data packet to a single I/O address in Ethernet controller one byte at a time. If this location is cached in the CPU caches, all such writes will proceed to a single location in the cache, each time overwriting the previously written byte. Thus the cache location will have the last byte of the packet that was written to the cache location. The cache is later flushed to the controller's buffer. Thus the controller will see only the last byte of the packet. This behavior is certainly not correct and therefore requires some special considerations.

If the address location of the Ethernet controller is not cached, such problems do not arise. In this case, all write operations will result in a byte being written to the Ethernet controller. Such behavior can be achieved in a number of ways. Almost all processors (memory management units to be precise) provides instructions to make certain memory pages as non-cacheable. If the Ethernet Controller is mapped to memory address space, use of such a mechanism is essential to disable the caching of Ethernet Controller locations. However if the Controller is mapped to I/O address space, no special care need to be taken with regard to the caching. Locations in the I/O address spaces are never cached in the processor caches.

8.2.1 I/O mapped on Memory address space

When an input-output device register is mapped to usual memory address space of the processor, the phenomenon is known as memory mapped I/O. In this case, the input-output operation takes place using the memory data transfer bus cycles. As explained earlier, it might be necessary to indicate that such memory addresses are not cached in the CPU.

The input-output operation takes place when one of the operand of an instruction is in memory and the effective address evaluates to the physical address at which the device is mapped to. This adds another complexity. The addresses specified in the programs are virtual addresses which are translated to a physical address by a memory management unit (MMU) within the processor. GNU/Linux controls the virtual to physical address mapping by setting various tables needed by the MMU. To a real memory, such mappings do not matter as long as the Operating System keeps them consistent for reading and writing. For example, if an address specified in the program as 0x50000 gets mapped to 0x20000, all read and write to memory location 0x50000 are carried out to a physical address 0x20000. If this location were to be a real memory, it would not matter to the program where and how this mapping is carried out. However if this location is mapped to an input or output register of a device, the address translation now starts mattering. The device is mapped to a physical address of 0x50000 and therefore will not sense any read or write operations to physical address 0x20000. It is therefore necessary for I/O mapped on memory address space to implement the following two features.

1. The part of the address space mapped to the device should not be cached with in the CPU.
2. No address translation should be carried out for addresses in this address space. This however raises a concern of security and needs a mechanism of avoidance of simultaneous access by multiple processes.

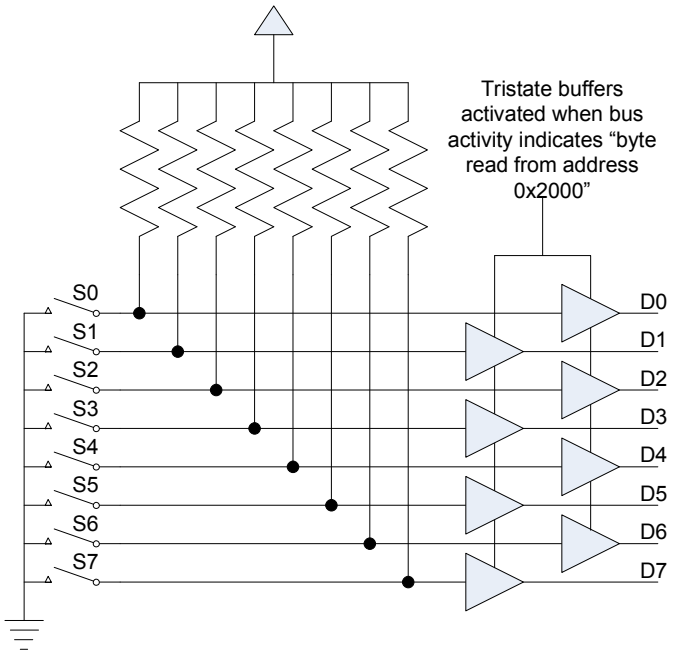


Figure 8.1: Push button switches on Bus

The input and output based on the device drivers spares the programmer worrying about such gory details. Device drivers in the kernel implement mechanisms by which physical addresses of input-output devices are protected and no process can get access to them.

Let's consider this mechanism with an example. Let's say that we have a device such as a set of eight push button switches mapped to a physical memory address 0x2000. An indicative block diagram of such a setup is shown in figure 8.1.

In this setup, there are eight switches whose ON and OFF status can be read by the CPU. The switch statuses are mapped to memory location 0x2000. When processor performs a memory read from this location, the address and other control on the bus indicate a memory read from location 0x2000. This bus condition is detected and the corresponding tristate buffers are activated to provide the status of the switches on the bus.

Such a setup is known as memory mapped input-output. In order to have this system work correctly the following must be ensured.

- Location 0x2000 is not cached in the processor data cache. If this aspect is not ensured, each time the processor reads data from location 0x2000, value is supplied from the cache without making any bus activity. This therefore provides a stale value of the switch

status.

- Access is made to physical location 0x2000 rather than the program address 0x2000 which is often a virtual address.

Such aspects are usually forgotten by programmers resulting in incorrect program executions. Operating systems provide device drivers which perform such operations correctly and ensure that input-output devices are protected from other processes running in a multi-process system such as in GNU/Linux.

8.2.2 I/O mapped on I/O address space

As mentioned earlier, IA32 processors provide a separate input-output address space of 2^{16} bytes. This address space uses just 16-bit addresses for accessing a location within the device. Memory management unit within IA32 processors ensure that these addresses are never cached in the processor caches and they do not pass through any address translation scheme. Thus the address provided in the program is the physical address of the device registers. Addresses in I/O address space are also known as "ports". Thus an IA32 architecture provides a maximum of 2^{16} byte wide ports, or 2^{15} 16-bit wide ports, or 2^{14} 32-bit wide ports.

IA32 processors provide separate set of instructions to read and write from I/O address space. When these instructions are used, the data is read or written to I/O address space rather than the memory address space. The following are the instructions provided in the IA32 instruction sets.

```
inb port
inw port
inl port
in port, dest
```

The `inb`, `inw` and `inl` instructions are used to read a byte, word or a long from an I/O location to registers `al`, `ax` and `eax` respectively. The `port` address can be specified as an immediate constant or as `dx` register. The `dest` operand is always a register `al` for byte read, `ax` for word read and `eax` for long read. IA32 instruction sets impose one more restriction on these instructions. the port address when specified as an immediate constant can only be a number between 0 to 255. For input-output addressing outside this range, 16-bit addressing is a must for which only the `dx` register can be used. Thus when port address is specified as an immediate constant, the range of the address is limited to between 0 to 255 only.

The following are the output instructions to operate in the I/O address space.

```
outb port
outw port
outl port
out src, port
```

The `outb`, `outw` and `outl` instructions are used to write a value to the specified port from a byte value in register `al`, word value in register `ax` and long value in register `eax` respectively. The `port` address can be specified as an immediate constant between 0 to 255 or any address between 0 to $2^{16} - 1$ in register `dx`. In the last form of `out` instructions, two arguments `src` and `port` address can be specified. The `src` can only be `al`, `ax` or `eax` register while the `port` can be an immediate constant between 0 to 255 or a 16-bit address in register `dx`.

IA32 instruction set also supports string based input-output operations using the following instructions.

```
insb
insw
insl
outsb
outsw
outsl
```

The `ins` and `outs` instructions are string oriented instructions that perform input and output from I/O address space to and from memory. The I/O address is implicitly specified in register `dx`. For the input instructions, destination memory address is specified in register `edi`. In a similar manner, source memory address is implied in register `esi` in case of output instructions. After an I/O operation is performed, the memory address registers (`edi` for input and `esi` for output instructions) are adjusted by the size of the data transfer. The adjustment is in the form of increment or decrement based on the direction flag in a way similar to that in other string instructions described in chapter 6.

As an example of using I/O address space, consider the setup of figure 8.1. In this figure, the control of tristate buffers can be changed to identify the I/O bus cycle. Thus the buffers would be activated when bus activity indicates an I/O read operation on I/O address, say 0x200 as shown in figure 8.2. This setup can then be used to read switches using `in` instruction for port address being 0x200. The following instruction sequence can be used to read the status of the switches in this example.

```
mov $0x200, %dx
inb %dx
```

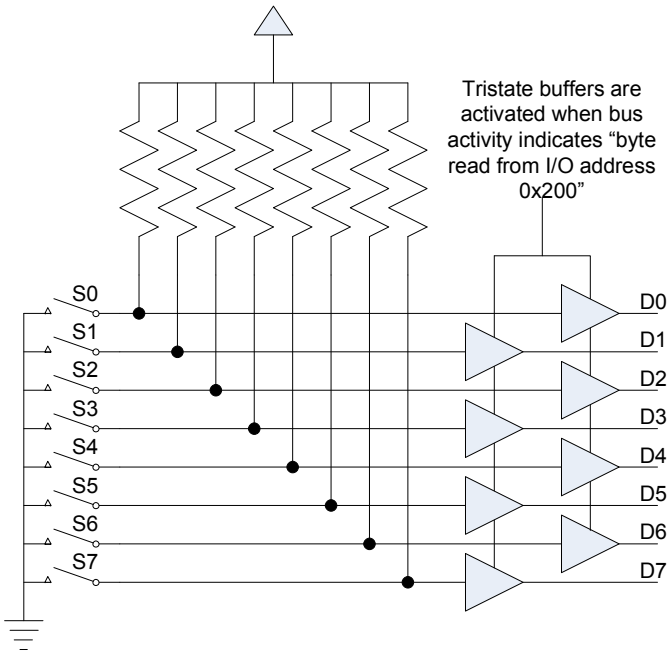


Figure 8.2: Switches mapped to the I/O address space

8.2.3 I/O with direct memory access (DMA)

Often there are devices which do a large amount of data transfer. Such devices include disks, tape drives, CD ROM and similar storage devices. For example, a typical unit of data transfer from a disk to the processor is one sector of the disk (usually 512 bytes). If this data is read using a program that reads one byte at a time, it results in an inefficient system of data transfer. For such devices, it is usually better to set up a transfer such that the device can write in the processor memory directly. Such operations are called DMA based I/O transfer. Many devices in GNU/Linux operating system use this mechanism. Use of DMA, setting up the DMA operation and other controls using the DMA are carried out by the device drivers only and require extensive setup. These are therefore not discussed here.

8.3 Input-Output protection

The addresses in the I/O address space are protected against simultaneous accesses by other processes running on the system. Such protections are typically implemented by the Operating System. It is not possible for a process to perform input and output operations un-

less the Operating System has enabled the access to those ports. IA32 architectures provide two methods of such protection schemes.

- IA32 architectures provide four levels of I/O privileges between 0 to 3. GNU/Linux operating system normally uses just two of them, 0 and 3. Normal processes are provided with no I/O privileges (I/O privilege level = 0) in the user mode of execution and with all I/O privileges (I/O privilege level = 3) in the kernel mode of execution. When a normal process executes an I/O instruction, exceptions are raised because it does not have enough I/O privileges. In the kernel mode of execution, the I/O privilege level of the process is raised to 3. A process can then execute I/O instructions.
- In addition to maintaining I/O privilege level, IA32 architectures also use a bitmap mask for individual ports. This bitmap data structure can only be modified at high privilege level (i.e. in kernel mode) and provides whether I/O is permitted to the specified address in the I/O address map, or not. When permission is granted for an I/O using this bitmap, I/O privilege level is not considered. IA32 processors maintain this bitmap only for first 1024 I/O addresses (port address between 0 to 0x3FF).

Thus an operating system can provide I/O permission in two different methods. In the first method, I/O permissions are granted by raising the I/O privilege level of the process. In this case, the process can perform I/O to any location. In the second method, bitmap mask is used to selectively grant the I/O permissions to specified address. In this mechanism access to first 1024 I/O addresses (0 to 0x3FF) can be provided selectively by using a bit mask maintained for each process. When the bit corresponding to an I/O address is set, the access to that I/O location is denied to the process. Otherwise, the I/O access is permitted.

The GNU/Linux, management of I/O privilege level and setting of I/O permission bit map can be performed using following system calls.

System Call: `ioperm`

Inputs:

`eax`: SYS_ioperm

`ebx`: start I/O address range

`ecx`: number of ports

`edx`: turn-on value

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EPERM`, `-ENOMEM`

The `ioperm` system call is used to manipulate the permissions for I/O addresses starting from the I/O address given in register `ebx`. Number of I/O port addresses whose permission is manipulated is given in register `ecx`. The permissions are changed depending upon the turn-on value specified in register `edx`. If register `edx` is 0, the input-output permissions are withdrawn (the actual bit map is set to 1), and if it is any value other than 0, the input-output permissions are given (the bit map is set to 0). Use of `ioperm` system call requires root privileges.

IA32 architectures support bitmap based permissions only for the first 1024 I/O addresses (0 to 0x3FF). Thus the values in register `ebx` must be less than or equal to 0x3FF. Further the expression `ebx + ecx − 1` should also be less than or equal to 0x3FF. There is no method by which IA32 processors can provide selective permissions of access to I/O address beyond 0x3FF. However, access to such I/O addresses can be given by raising the privilege level of the process for it to be able to perform input-output operation to any address. This is achieved by `iopl` system call.

When a new process is created using `fork`, `clone` or a similar system call, it starts with no I/O permissions. Therefore I/O permissions granted using `ioperm` system call are not inherited upon process creation. However when a process performs `execve` system call, the I/O permissions as set prior to making a call to `execve` are retained. This system call therefore can be used by a root owned process to enable access to certain ports before changing the user id of the process thereby giving port access permissions to non-privileged tasks.

System Call: `iopl`

Inputs:

`eax:` SYS_iopl

`ebx:` level

Return Values:

Normal Case: 0

Error Values: `-EINVAL`, `-EPERM`

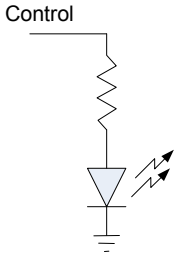


Figure 8.3: Controlling an LED

The `iopl` system call changes the I/O privilege level of the current process. The value of the `level` in register `ebx` can be only a number between 0 to 3. Any process can lower its I/O privilege level but it must have root privilege to raise its I/O privilege level.

In addition to granting unrestricted access to I/O address space, a high I/O privilege level also allows the process to disable interrupts which interferes with the operations performed by the operating system and probably crash the system.

I/O privilege level is inherited by child processes when they are created using `fork` call. These are also passed on to the new program executed using `execve` system call.

The normal value of I/O privilege level of a process is 0, thereby granting no permission for direct input-output operations.

8.4 A case study

In this case study, we shall use the parallel port on a PC to interface a set of eight LEDs and control display of these LEDs using a program in Assembly language. Thus it will be possible for the software running on the PC to switch one or more LEDs to ON or OFF state. A single LED can be controlled using a single bit control as shown in figure 8.3. When the `Control` signal is high, the LED will be turned on. The LED will be off when the control signal is low.

8.4.1 Parallel Port on PCs

In order to understand this case study, it is necessary to describe the parallel port hardware on the PC in brief. Parallel port on a PC is identified by a 25 pin D type female connector, also known as IEEE 1284 type A connector. It carries several signals as shown in figure 8.4. Data is carried over D0 to D7 signals. Many signals are dedicated to the handshake for data transfer (Strobe#, Ack# and Busy#). Several signals are

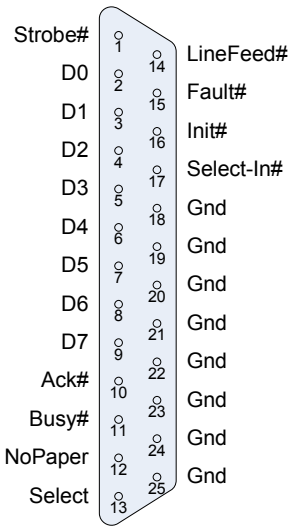


Figure 8.4: Parallel Port on a PC

used by the printers to indicate conditions that seek attention from the user (such as NoPaper) and are not used in our case study.

The parallel port transfers eight bits of data (on lines D0 to D7) at a time. There are several variants of the parallel port which evolved over time. All variants of parallel ports support the basic data transfer of 8 bits under program control. Modern computers support standardized parallel port known as IEEE 1284 standard. This standard defines various modes of operations including “centronics” compatibility mode, EPP (Enhanced Parallel Port) mode and ECP (Extended Capabilities Port) mode.

Data handshake (using signals Strobe#, Ack# and Busy#) is implemented using software control in standard “centronics” compatibility mode, using hardware in EPP mode and using DMA based transfer and control in ECP mode. ECP mode of communication achieves the fastest transfer mode. The protocol for data handshake is shown in figure 8.5. Data is expected to be valid at rising edge of Strobe# signal. Device on the printer port is expected to provide Busy# and Ack# signals to complete the protocol. Most implementations on the PCs ignore Ack# signal.

A standard printer port LPT1 on a PC is normally mapped to base I/O address as 0x378. In standard Centronics compatibility mode, it uses three I/O addresses (0x378, 0x379 and 0x37A).

Port 0x378 is used for the data port. When a value is written to this port, it appears on D0 to D7 lines of the printer port. Port 0x379 is a read-only status port. When read, it provides the status of various

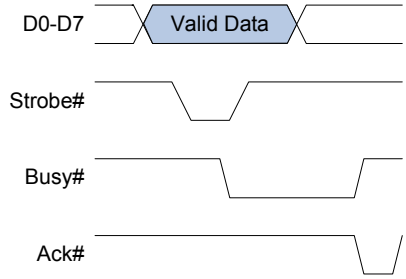


Figure 8.5: Handshake on PC Parallel Port

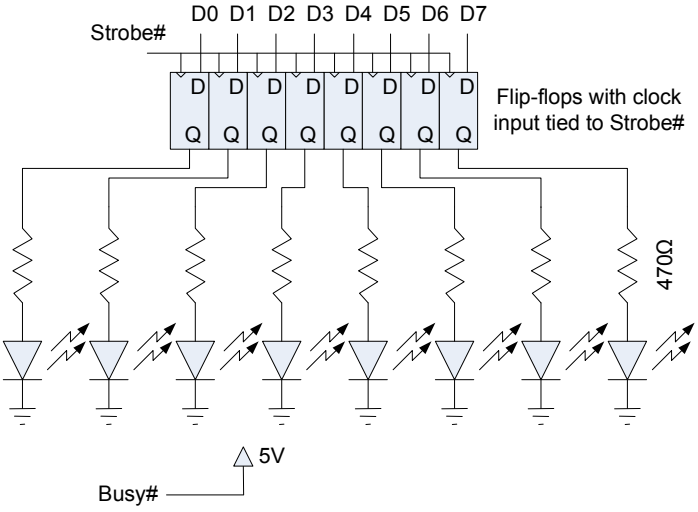


Figure 8.6: Interfacing LEDs on PC Parallel Port

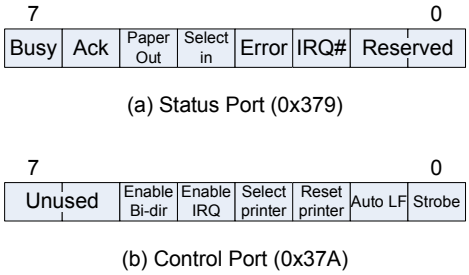


Figure 8.7: I/O port definitions for LPT1 on PCs

control signals on printer port. Port 0x37A is a write-only control port and is used to provide output control signals (such as Strobe#) on the printer port. Both status and control ports are shown in figure 8.7.

8.4.2 Interfacing LEDs to Parallel Port

In our case study we shall use simple software controlled handshake. We will use data lines to provide signal values from the PC to the row of LEDs and Strobe# signal to indicate the change in data. All signals will be produced with software control in standard "centronics" compatibility mode.

We use a set of eight flip-flops that latch data from parallel printer port on the rising edge of Strobe# signal. The output of flip-flops will then be used to control eight LEDs as shown in figure 8.6. In order to complete the protocol, Busy# signal is connected to 1 implying that our device on the printer port is always ready to accept data.

8.4.3 Software interface for Parallel Port I/O

Software interface to drive the LEDs on the parallel port is fairly easy. The software function must output a byte indicating the LEDs to port address 0x378 and then should provide a pulse on Strobe# line by manipulating the port address 0x37A. We will write an Assembly language function that can be called from functions written in C. It takes one argument that is interpreted as ON and OFF state of the LEDs. C prototype of this function is the following. The function always returns a 0 to indicate success.

```
int outLEDs(unsigned char LEDstates);
```

An Assembly language implementation to do such an operation is given below.

```
.globl outLEDs
outLEDs:
```

```
  // Argument at 4(%esp)
  // Just one byte is to be looked at
  // 4(%esp)
  push %dx
  // Parameter is now at 6(%esp)
  mov $0x378, %dx // LPT1 base address
  mov 6(%esp), %al
  outb %dx
  // Set Strobe# signal to 0.
  // Strobe# signal is set to 0 by
  // setting Strobe to 1 in control
  // port.
  mov $1, %al
  add $2, %dx // Control port
  outb %dx
  nop         // Small delay
  nop
  nop
  nop
  mov $0, %al // Strobe#=1
  outb %dx
  pop %dx
  mov $0, %eax // Return value
  ret
```

As explained earlier, this program will not be able to run unless the permissions are granted to perform I/O at ports 0x378 and 0x37A. We shall use the IOPERM system call to grant permissions as given in the following C callable function `SetPerm`.

```
// C Callable function to setup
// Permissions to perform I/O
// at addresses 0x378:0x37A
// C Prototype is
//   int SetPerm(void);
// Return Values
//   0: Success
//   Any other Value: Error
// This function can only be
// called by the root user.

#include <asm/unistd.h>
#include <syscall.h>
.globl SetPerm
SetPerm:
  push %ebx
```

```
  push %ecx
  push %edx
  mov $0x378, %ebx
  mov $3, %ecx
  mov $1, %edx
  mov $(SYS_ioperm), %eax
  int $0x80
  pop %edx
  pop %ecx
  pop %ebx
  ret
```

Finally a complete assembly language program that turns on every alternate LED in sequence is given below.

```
#include <asm/unistd.h>
#include <syscall.h>
#define LEDValSet 0x55
.data
// These values will be used in nanosecond
// sleep system call. Values are for 50ms.
time_spec:
   .long 0        // Seconds
   .long 50000000 // Nanoseconds
.text
.globl _start
_start:
   call SetPerm
   pushl $LEDValSet
loop_forever:
   xorb $0xFF, (%esp) // Invert bits
   call outLEDs
   // Parameter remains on stack
   // For subsequent iteration
   // of the loop. At each iteration
   // bits of the parameter are
   // inverted.

   // Delay of 50ms before inverting
   // the LEDs (i.e. next iteration)
   mov $SYS_nanosleep, %eax
   mov $time_spec, %ebx
   mov $0, %ecx
   int $0x80
   // At each iterations the on LEDs
   // are turned off and off LEDS are
```

```
// turned on. Program is in an
// infinite loop. Ways to kill
// this process are to press ^C
// or by kill command.
jmp loop_forever
```

Chapter 9

Handling Real Number Arithmetic

Digital computers are extremely powerful in handling data that can be stored in finite amount of storage. For real numbers, however, the finite storage model is a serious limitation. Between any two distinct real numbers there are infinite real numbers. It is due to this property of real numbers that they can not be stored using a finite number of bits. Therefore real numbers are represented in a different manner on digital computers. In this chapter, we shall learn methods to store real numbers on a digital computer. We then also understand the data types and instructions supported by IA32 architectures to store real numbers.

The IA32 architectures handle operations on real numbers in a different manner than what we have seen so far for integers. There are two kinds of instruction sets in IA32 architectures for handling real numbers. Instruction set of IA32 architectures finds its roots in 8086 CPU from Intel. This CPU did not provide any support for handling arithmetic using real numbers. Instead Intel provided a numeric co-processor called 8087 which would work alongside the 8086 CPU and operate for instructions to handle real numbers. Later several 8087 compatible numeric co-processors were introduced by Intel to operate with different kinds of CPUs. For example, 80287 numeric co-processor was used with 80286 CPUs from Intel. Similarly, 80387 numeric co-processors were used with 80386 CPUs. Around this time, the numeric co-processor and the regular processor could be integrated in a single package. For compatibility with the original 8087 co-processor, later CPUs from Intel supported x87 instruction set for real number computations. These therefore got incorporated in IA32 instruction sets.

Subsequently, instruction sets in Intel processors were enhanced

to support other kinds of instructions such as MMX, SSE and SSE2. These instruction sets also overlap with the real number support provided by x87 floating point unit. In this chapter we discuss the x87 related architecture and instruction set. Before we learn about the specifics of IA32 architecture, it is also worthwhile to understand the representation mechanisms for real numbers.

9.1 Representation of real numbers

Consider a decimal number 2.35. Each of the digits in this number have a positional value. The positional value of digit 2 is 2×10^0, while the positional values of digits 3 and 5 are 3×10^{-1} and 5×10^{-2} respectively. In a similar way, consider a binary number 01.110 where the binary point (analogous to the decimal point) separates integer and fractional parts. The positional values of bits 0, 1, 1, 1 and 0 are 0×2^1, 1×2^0, 1×2^{-1}, 1×2^{-2} and 0×2^{-3} respectively. This number evaluates to a value 1.75 in decimal.

Some numbers that can be represented in finite number of digits in decimal number system can not be represented in finite number of bits in binary number system. In such numbers, the fractional part leads to a non-terminating recurring sequence of bits. For example, 1.3 in decimal will lead to $1.0100110011001\ldots$, or $1.0\overline{1001}$, in binary. In such situations, representation on computers will consider only a finite number of significant bits and ignore the rest of them. For example, in this case if we consider only 8 bits after the binary point, the number in binary representation will be 1.01001100 which is 1.296875 in decimal. This representation therefore introduces an error of 0.003125. More the number of bits to consider, better is the accuracy in representation. For example, if we consider 12 bits after binary point, the binary number will be 1.010011001100 which corresponds to 1.2998046875, a number that is much closer to the original 1.3 with an error of 0.0001953125.

EXERCISES:

9.1 Represent the following decimal numbers in binary.
(a) 1.675 (b) 5.03125 (c) 3.10825 (d) 5.9

9.2 Find decimal representation for the following binary numbers.
(a) 1101.01101 (b) 101.1101001 (c) $110.1\overline{11011}$

9.3 Assume that following recurring sequences in binary representations are stored on a computer using 10 bits after the binary point. Find the error introduced due to representation on the computer.
(a) $11.01\overline{11011}$ (b) $101.1011\overline{0011}$

9.2 Fixed point representation

As we had seen earlier, the binary number system is the most natural number system for representing a data item on a computer. A real number can be written as a sequence of bits with an embedded binary point. In general there is no need to represent the binary point if location of the binary point is known and is fixed a-priori. For example, if we assume that binary point will always be before the fourth bit from the right, the binary point may be dropped in the representation. Let us see this with an example. Consider a real number 1.75 whose binary representation is 1.11. In order to make the binary point location before the fourth bit from the right, we add two zeros to make it 1.1100. Now that the binary point is implied, it can be dropped from the representation to make it 11100. A representation where the binary point is implied at some location (in this example before the fourth bit from the right), is called a fixed point representation. As in integer representation, the total number of bits are also fixed. This is in addition to the location of the binary point. Thus there can be fixed point numbers represented in a byte, word and long data widths. Each of these will assume the presence of a binary point at some fixed location. In our example, if we assume a byte wide integer for representing number, we will need to add three leading zeros to make it 00011100. Thus in our example, the four most significant bits of the byte are used to represent the integer part of the real number while the lower order four bits are used to represent the fractional part after the binary point. Similarly if the binary point is implied before the fourth bit from the right and the number is represented using word or long data types, 12 bits in word data type and 28 bits in long data type are used for the integer part and four bits are used for fractional part.

The smallest number that can be represented using fixed point representation is the one with integer part being 0 and fractional part being all zeros except one in the least significant bit. For example, in our scenario with four bits to represent the fractional part, the smallest number is $00\ldots0001$, or $(1/16)$, or 0.0625. The smallest number that can be represented in a given representation system is also known as precision of the representation scheme. In our example, 0.0625 is the precision of fixed point number system where the binary point is implied before the fourth bit from the right.

Let us consider another example of representing 6.2 as a number. The binary representation of 6.2 is $110.\overline{0011}$. We shall use fixed point number system where the binary point is implied before the sixth bit from the right. Thus the fraction gets truncated to six bits and the number results in 110.001100 (or, approximated to 6.1875 in decimal). This number can not be represented using byte data type without losing the integer part. In word data type, the fixed point representation for this number is 0000000110001100. In this representation, The last six

bits are used to hold fractional part while the remaining ten bits are used to hold integer part.

It may be seen by treating this bit string as an integer and converting this bit string to decimal that the same bit pattern is also an integer representation of 396 in decimal. This number 396 is obtained by multiplying 6.2 by 64 (or 2^6) and dropping the fractional part from the result. In general, a fixed point representation of a real number can be obtained by multiplying it by 2^k, where k is the number of bits to represent the fraction part, and dropping the fractional part from the result. The integer so obtained provides representation of the real number using fixed point representation scheme.

We will denote a fixed point number using $\mathcal{F}_{n,k}$, where n is bit-width of the fixed point number and k is bit-width used to represent fractional part of the real number. In this scheme, $n-k$ bits are used to represent integer part of the real number. Therefore, fixed point representation $\mathcal{F}_{n,k}$ of a real number r can be obtained by multiplying r by 2^k, taking its integer part and then representing that in n bits.

Let us consider $\mathcal{F}_{8,4}$ representation of 3.8 as an example. This number when multiplied by 2^4, or 16, results in 60.8. Representation of its integer part (60) in binary is 00111100. Therefore $\mathcal{F}_{8,4}$ representation of 3.8 is 00111100.

The binary number representation of 3.8 is a nonterminating bit sequence $11.\overline{1100}$. By truncating the fractional part to only four bits, we incur an inaccuracy in the representation. In general this inaccuracy is always smaller than the precision of the number system. The $\mathcal{F}_{8,4}$ representation of 3.8 is 00111100 which is an accurate representation of 3.75. Therefore the representation of 3.8 has a representational error of 0.05.

Negative real numbers are represented using 2's complement for the corresponding integer representation. As an example, we shall work out $\mathcal{F}_{16,3}$ representation of -1.75. In order to represent this number, we shall multiply it is 2^3 (or, 8) to obtain -14. Representation of -14 is 16 bits is $1111\,1111\,1111\,0010$, or 0xFFF2. This number can also be obtained by taking $\mathcal{F}_{16,3}$ representation of real number 1.75 and then negating it using 2's complement arithmetic. On IA32 processors, the most commonly used data types are 8-bit, 16-bit or 32-bit wide. Thus an obvious choice for n is 8, 16 or 32.

9.2.1 Addition and subtraction using fixed point representation

Fixed point number representation $\mathcal{F}_{n,k}$ is essentially an integer representation of the real number r scaled by 2^k. Therefore,

$$\mathcal{F}_{n,k}(r) = \lfloor r * 2^k \rfloor$$

Given two real number r_1 and r_2, their fixed point representations can be written as the following.

$$\begin{aligned}\mathcal{F}_{n,k}(r_1) &= \lfloor r_1 * 2^k \rfloor \\ \mathcal{F}_{n,k}(r_2) &= \lfloor r_2 * 2^k \rfloor\end{aligned}$$

The summation of two number r_1 and r_2 can be denoted by r and its fixed point representation should be $\lfloor (r_1 + r_2) * 2^k \rfloor$. However on a digital computer the sum is obtained by adding $\mathcal{F}_{n,k}(r_1)$ and $\mathcal{F}_{n,k}(r_2)$ using integer addition. This is only an approximation of the actual summation and sometimes results in an error, known as truncation error. The truncation errors arise due to the fact that $\lfloor (r_1 + r_2) * 2^k \rfloor$ is not always the same as $\lfloor r_1 * 2^k \rfloor + \lfloor r_2 * 2^k \rfloor$. In general the truncation errors may or may not arise in summation of the fixed point numbers. The errors do not arise when the following is satisfied.

$$\lfloor (r_1 + r_2) * 2^k \rfloor = \lfloor r_1 * 2^k \rfloor + \lfloor r_2 * 2^k \rfloor$$

As an example, let us consider two real numbers 1.6 and 1.2. The sum of these two real numbers is 2.8. If we consider $\mathcal{F}_{8,4}$ representation of these real numbers, we get the following.

$$\begin{aligned}\mathcal{F}_{8,4}(1.6) &= 00011001 \\ \mathcal{F}_{8,4}(1.2) &= 00010011 \\ \mathcal{F}_{8,4}(2.8) &= 00101100\end{aligned}$$

Integer addition of $\mathcal{F}_{8,4}(1.6)$ and $\mathcal{F}_{8,4}(1.2)$ is $00011001 + 00010011$, or 00101100, which is indeed the $\mathcal{F}_{8,4}$ representation of 2.8. This example addition, therefore does not lead to any truncation error.

We consider another example of adding two real numbers 1.8 and 1.9. The addition of these two real numbers is 3.7. The $\mathcal{F}_{8,4}$ representations of these numbers are the following.

$$\begin{aligned}\mathcal{F}_{8,4}(1.8) &= 00011100 \\ \mathcal{F}_{8,4}(1.9) &= 00011110 \\ \mathcal{F}_{8,4}(3.7) &= 00111011\end{aligned}$$

By adding $\mathcal{F}_{8,4}(1.8)$ and $\mathcal{F}_{8,4}(1.9)$ using integer arithmetic, we get 00111010. This bit pattern is the $\mathcal{F}_{8,4}$ representation of 3.625 leading

to a truncation error of 0.075. As given above, the $\mathcal{F}_{8,4}(3.7)$ is 00111011, which is equivalent to 3.6875 leading to a representation inaccuracy of 0.0125. In this particular example, the addition using integer arithmetic leads to a larger truncation error. The truncation errors can creep in any arithmetic operation including subtraction, multiplication and division and are not limited to only the addition.

9.2.2 Multiplication and division of fixed point numbers

Let us consider two real numbers r_1 and r_2, whose fixed point representations are $\mathcal{F}_{n,k}(r_1)$ and $\mathcal{F}_{n,k}(r_2)$ respectively. Therefore,

$$\begin{aligned} \mathcal{F}_{n,k}(r_1) &= \lfloor r_1 * 2^k \rfloor \approx r_1 * 2^k \\ \mathcal{F}_{n,k}(r_2) &= \lfloor r_2 * 2^k \rfloor \approx r_2 * 2^k \end{aligned}$$

Multiplication of $\mathcal{F}_{n,k}(r_1)$ and $\mathcal{F}_{n,k}(r_2)$ yields $\lfloor r_1 * 2^k \rfloor * \lfloor r_2 * 2^k \rfloor$ which is an approximation of $r_1 * r_2 * 2^{2k}$. Shifting this result of multiplication by k bits to the right results in approximately $r_1 * r_2 * 2^k$, which is fixed point representation of $r_1 * r_2$.

Therefore, multiplication of two fixed point numbers is computed by integer multiplication followed by a division by 2^k (or right shift by k bits).

Similarly in case of a division of two fixed point numbers, the numerator is first multiplied by 2^k (or left shifted by k bits) and then an integer division is performed by the denominator.

Both multiplication and divisions can have high truncation errors. This is illustrated with an example. Let us consider two real numbers 5.4 and 8.3. Multiplication of these two number is 44.82. The fixed point representations of these three numbers using $\mathcal{F}_{16,8}$ scheme are as follows.

$$\begin{aligned} \mathcal{F}_{16,8}(5.4) &= 0000\,0101\,0110\,0110 \ \ (0\text{x}0566) \\ \mathcal{F}_{16,8}(8.3) &= 0000\,1000\,0100\,1100 \ \ (0\text{x}084\text{C}) \\ \mathcal{F}_{16,8}(44.82) &= 0010\,1100\,1101\,0001 \ \ (0\text{x}2\text{CD}1) \end{aligned}$$

Using integer multiplication of 0x0566 and 0x084C we obtain a number 0x2CCA48. To complete the fixed point multiplication, this number is right shifted by 8 bits giving a value 0x2CCA. This number is fixed point representation of 44.7890625 using $\mathcal{F}_{16,8}$ scheme. Thus the integer multiplication followed by a right shift results in an error of 0.0309375. This error includes two components, truncation error as well as representation error.

Often a careful design of algorithms can provide values that have smaller truncation errors. Such algorithm designs are out of scope of this book. However we illustrate this with an example. In general, if the range of the numbers are known a-priori, they can be represented in fixed point representation with finer precision. Arithmetic involving numbers with finer precision results in smaller truncation and representational errors. For example, if we choose $\mathcal{F}_{20,12}$ representation scheme for representing 5.4 and 8.3, we get 0x05666 and 0x084CC respectively. Multiplication and subsequent shift by 12 bits results in 0x2CD17 (or, 44.818115234375), a number that is much closer to the correct value (44.82).

Fixed point number representation is an interesting technique used in digital computers. In this method, there is no need to use special instructions for arithmetic. Integer arithmetic instructions can be used for computations involving numbers in fixed point representation. Addition and subtraction of fixed point numbers are the same as integer addition and subtraction. Multiplication and division of fixed point numbers is implemented using multiplication and division operations on integers in addition to shift operations. This aspect of fixed point number representation results in a fast arithmetic computation in algorithms. Integer arithmetic for real numbers is commonly used in applications such as digital signal processing with real-time constraints. While fixed point arithmetic can be implemented using integer instructions, IA32 processors also provide numeric co-processor for handling computations involving real numbers represented as floating point numbers.

EXERCISES:

9.4 In the following arithmetic, use $\mathcal{F}_{12,6}$ fixed point representation scheme and find truncation error.

(a) $12.6 + 11.3$ (b) $-9.8 - 13.2$ (c) 7.2×4.1 (d) $15.9/3.0$

9.5 Write four routines, `fixAdd`, `fixSub`, `fixMul` and `fixDiv` in Assembly language for performing addition, subtraction, multiplication and division of real numbers respectively. Assume that the numbers are represented using $\mathcal{F}_{32,16}$ fixed point representation. For each of the routines, two input numbers are given in registers `eax` and `ebx` upon entry. Result of the operation is to be given in register `eax`.

9.6 Write four more routines that can be called using C interface. These routines must take the arguments on stack and must call the corresponding routines written in exercise 9.5. Test these routines by calling them in a C program and passing integer part of two real numbers after multiplying by 2^{16}. Check the results if they are correct.

9.3 Floating Point Representation

Fixed point representation provides limited range and precision for representing real numbers. In fixed point representation $\mathcal{F}_{n,k}$, the precision is constant and is given by 2^{-k}. However in real problems, the requirement of precision is dependent on the magnitude of the number itself. As an example, consider that we need a variable to store the mass of various objects. If this variable stores the mass of a planet, precision of 1g is far too fine. However if the same variable is used to store the mass of an electron, precision as fine as 1×10^{-30}g may also not be sufficient.

In normal writing, usually scientific representation scheme is used to handle such kind of numbers. In this representation, a real number is written using three parts, a sign (positive or negative), a significant part and an exponent. The following are some examples of such a number representation.

Mass of a small ship :	2.5×10^{8} g
Mass of a person :	7.0×10^{4} g (or, 70Kg)
Mass of a carbon atom :	1.992×10^{-23} g
Mass of an electron :	9.02×10^{-28} g

In scientific representation, the precision is relevant to only the significant part. Thus a precision of 4th decimal place will mean that the significant part is correct up to the fourth decimal place irrespective of the exponent. Thus in our examples with fourth decimal place of precision, the mass of an atom can be represented with an accuracy of 0.0001×10^{-23}, or, 10^{-27}g. Similarly the masses of a person and a ship can be represented with an accuracy of the order of 1g and 10Kg respectively. A number in scientific representation is called normalized number if the integer portion of the significant part is between 1 and 9 (both inclusive). For example, 0.0001×10^{-23} is not a normalized number while 1.992×10^{-23} is normalized. 0.0001×10^{-23} can be written in normal form as 1.0×10^{-26}.

Analogous to scientific representation, a scheme is used in digital computers where the precision of the number is dependent only on the significant part. This representation scheme is called floating point number representation scheme. For a computer, the number of bits used for the exponents, significant part etc. need to be defined in a precise manner. Further, it is desirable that same number representation be used by all programs so that the programs work seamlessly. Most implementation of programming languages and most architectures support an inter-operable standard known as IEEE754 floating point number representation scheme. In this system the exponents are represented in powers of two. Similarly the significant part is represented as a binary number (similar to a fixed point number represen-

tation). The following table illustrates some of the concepts involved in this number system.

Scheme	**Representation**
Decimal Number	140.625
Decimal Scientific Representation	1.40625×10^2
Binary Number	10001100.1010
Binary Scientific Representation	1.00011001010×2^7
Decimal Number	-0.125
Decimal Scientific Representation	-1.25×10^{-1}
Binary Number	-0.001
Binary Scientific Representation	-1.0×2^{-3}

If a real number is to be represented in binary scientific representation scheme, there are four components each of which must be represented. These components are signs and magnitudes of each of significant part and exponent (power of two).

Table 9.1: IEEE754 Single Precision Number Format

	Sign	Mantissa	Exponent
Width (in bits)	1	23	8
Positive Normalized number	0	f	$1 \leq e \leq 254$
	Value=$1.f \times 2^{e-127}$		
Negative Normalized number	1	f	$1 \leq e \leq 254$
	Value=$-1.f \times 2^{e-127}$		
Max Positive Normalized number	$1.111...11 \times 2^{127}$ $\approx 2^{128}$, or, $\approx 3.4 \times 10^{38}$		
Min Positive Normalized number	$1.000...01 \times 2^{-126}$ $\approx 2^{-126}$, or, $\approx 1.17 \times 10^{-38}$		
$+0.0$	0	00...00	0
-0.0	1	00...00	0
$+\infty$	0	00...00	255 (all 1s)
$-\infty$	1	00...00	255 (all 1s)

A number in decimal scientific representation is called normalized if there is exactly one non-zero digit (1 to 9) before the decimal point. Analogous to this, in a binary normalized representation there is exactly one non-zero bit before the binary point. In binary number system this bit can only be a 1. Therefore, in normalized form of binary representation, the most significant bit will always be 1 (except the case when 0.0 is to be represented). Since the most significant bit is always 1, it can be omitted altogether from the representation. Using a normalized number scheme, however 0.0 can not be represented. This number is treated in a special manner in the IEEE754 representation.

Table 9.2: IEEE754 Double Precision Number Format

	Sign	Mantissa	Exponent
Width (in bits)	1	52	11
Positive Normalized number	0	f	$1 \leq e \leq 2046$
	Value=$1.f \times 2^{e-1023}$		
Negative Normalized number	1	f	$1 \leq e \leq 2046$
	Value=$-1.f \times 2^{e-1023}$		
Max Positive Normalized number	$1.111...11 \times 2^{1023}$ $\approx 2^{1024}$, or, $\approx 1.8 \times 10^{308}$		
Min Positive Normalized number	$1.000...01 \times 2^{-1022}$ $\approx 2^{-1022}$, or, $\approx 2.2 \times 10^{-308}$		
$+0.0$	0	00..00	0
-0.0	1	00..00	0
$+\infty$	0	00..00	2047 (all 1s)
$-\infty$	1	00..00	2047 (all 1s)

IEEE754 representation scheme supports numbers in normalized and non-normalized forms. We will first consider the normalized form.

9.3.1 Normalized number representation

In the normalized number representation, the integer part (prior to the binary point) is always 1. Therefore 0.0 can not be represented using normalized representation scheme. IEEE754 representation scheme has a special representation for 0.0. While in computations there is no significance of a sign before 0.0, IEEE754 provides room to represent $+0.0$ differently from -0.0.

Exponent of the numbers in normalized form are added a constant value k to ensure that it is always positive. This kind of representation is known as 'excess-k' representation. For example, in excess-127 representation, 127 is added to all exponents. Therefore in excess-127 representation -3 as an exponent will be represented as -3+127, or a positive integer 124. Similarly an exponent of 5 will be represented as a positive integer 132. The exponent in excess-127 representation will be positive only when it is not less than -126.

The representation of a real number in IEEE754 representation includes sign of the number, exponent in excess-k form and mantissa (significant part in normalized form with the most significant bit removed). However few kinds of numbers can not be stored in this way. For such numbers, specialized representations are used. A notable example is that of the representation of 0.0, which is represented by all 0s for mantissa and exponent. In addition, IEEE754 representation scheme also supports the representation of $\pm\infty$.

Table 9.3: Examples of IEEE754 Floating Point Number

Scheme	Representation		
Decimal Number	-24.75		
Decimal Scientific	-2.475×10^1		
Binary Number	-11000.11		
Binary Scientific	-1.100011×2^4		
Excess-127 exponent	$-1.100011E10000011$		
Excess-1023 exponent	$-1.100011E10000000011$		
IEEE754 Single Precision	Sign 1	Exp 10000011	Mantissa 1000110...00
IEEE754 number (in hex)	0xC1C6 0000		
IEEE754 Double Precision	Sign 1	Exp 10000000011	Mantissa 10001100...0000
IEEE754 number (in hex)	0xC038 C000 0000 0000		
Decimal Number	0.0625		
Decimal Scientific	6.25×10^{-2}		
Binary Number	0.0001		
Binary Scientific	1.0×2^{-4}		
Excess-127 exponent	$1.0E01111011$		
Excess-1023 exponent	$1.0E01111111011$		
IEEE754 Single Precision	Sign 0	Exp 01111011	Mantissa 00...00
IEEE754 number (in hex)	0x3D80 0000		
IEEE754 Double Precision	Sign 0	Exp 01111111011	Mantissa 00...0000
IEEE754 number (in hex)	0x3FB0 0000 0000 0000		

IEEE754 representation scheme provides two different forms of representation – single precision and double precision. Tables 9.1 and 9.2 provide details about single and double precision floating point formats of IEEE754 representation.

Let's consider a few examples of floating point number representations as given in table 9.3. In these examples, two decimal real numbers, -24.75 and 0.0625 are represented in single and double precision floating point number representation schemes. Number -24.75 is -1.100011×2^4 in binary scientific representation. Hence the exponent is 4 while the sign of the number is 1 (for negative). As discussed earlier, leading '1.' in the significand is not represented in the floating point representations. Hence the single and double precision representations are 0xC1C6 0000 and 0xC038 C000 0000 0000 respectively. In a similar manner, 0.0625 is 1.0×2^{-4} in binary scientific representation. It translates to 0x3D80 0000 and 0x3FB0 0000 0000 0000 in single and

double precision floating point representations.

EXERCISES:

9.7 Represent the following real numbers in 32-bit single point precision number representation.
(i) 125.609375 (ii) -87093.0 (iii) 1.414

9.8 What do the following bit patterns represent when treated as 32-bit single precision floating point numbers.
(i) 0x40170000 (ii) 0x43020000 (iii) 0xC1700000

9.9 Convert the single precision floating point numbers in exercise 9.8 to double precision floating point numbers.

9.3.2 Denormalized number representation

Often during the evaluation of a large expression, intermediate results are very small. These may become so insignificant that they are treated as zero. This phenomenon often gives incorrect results of computation in a large expression. IEEE754 representation provides a mechanism to extend the range for representing such small numbers. In this mechanism, the numbers are represented in denormalized form. The objective of such a provision in the representation scheme is primarily to provide larger range to keep intermediate results. Therefore, the numbers that can be represented using the normalized form can not be represented using the denormalized form and vice versa. This provides a uniqueness in representing numbers.

From the tables 9.1 and 9.2, it is clear that certain kind of bit patterns are not used in normalized number representation. These bit patterns are used for representing denormalized numbers and few special kind of numbers known as "Not A Number", or NaN. When the exponent in excess-k representation is 0 and the mantissa is not zero, the number is in the denormalized form. Denormalized number can be positive or negative depending upon the sign bit of the number. Positive denormalized numbers provide values between the minimum positive normalized number and 0.0. Similarly, negative denormalized numbers provide values between 0.0 and the maximum negative normalized number.

Value of a denormalized number is interpreted as $0.f \times 2^{-k+1}$, where f is the bit pattern in the mantissa bits and k is the excess-k constant of the number representation scheme. In single precision floating point numbers, k is 127 and therefore the value of the number would be $0.f \times 2^{-126}$ while in double precision floating point numbers, k is 1023 and therefore the value of the number would be $0.f \times 2^{-1022}$.

A summary of denormalized numbers in IEEE754 representation scheme is given in table 9.4.

Table 9.4: Denormalized numbers in IEEE754 floating point Number Format

	Sign	Mantissa	Exponent
Positive denormalized number (Single Precision)	0	f	0
	Value=$0.f\times 2^{-126}$		
Negative denormalized number (Single Precision)	0	f	0
	Value=$-0.f\times 2^{-126}$		
Positive denormalized number (double Precision)	1	f	0
	Value=$0.f\times 2^{-1022}$		
Negative denormalized number (double Precision)	1	f	0
	Value=$-0.f\times 2^{-1022}$		
Max. Positive denormalized number (Single Precision)	$0.111...111\times 2^{-126}$ $\approx 2^{-126}$, or, $\approx 1.17\times 10^{-38}$		
Min. Positive denormalized number (Single Precision)	$0.000...001\times 2^{-126}$ $=2^{-149}$, or, $\approx 1.4\times 10^{-45}$		
Min. Negative denormalized number (Single Precision)	$-0.111...111\times 2^{-126}$ $\approx -2^{-126}$, or, $\approx -1.17\times 10^{-38}$		
Max. Negative denormalized number (Single Precision)	$-0.000...001\times 2^{-126}$ $= -2^{-149}$, or, $\approx -1.4\times 10^{-45}$		
Max. Positive denormalized number (Double Precision)	$0.111...111\times 2^{-1022}$ $\approx 2^{-1022}$, or, $\approx 2.2\times 10^{-308}$		
Min. Positive denormalized number (Double Precision)	$0.000...001\times 2^{-1022}$ $=2^{-1074}$, or, $\approx 4.94\times 10^{-324}$		
Min. Negative denormalized number (Double Precision)	$-0.111...111\times 2^{-1022}$ $\approx -2^{-1022}$, or, $\approx -2.2\times 10^{-308}$		
Max. Negative denormalized number (Double Precision)	$-0.000...001\times 2^{-1022}$ $= -2^{-1074}$, or, $\approx -4.94\times 10^{-324}$		

As an example of denormalized form of the IEEE single precision floating point numbers, let us consider a real number 0.0625×2^{-128}. Since this number is smaller than the minimum that can be represented using normalized form of the numbers (i.e. $\approx 2^{-126}$), it can only be represented using denormalized form. The number 0.0625×2^{-128} can be rewritten as $(0.0625/4) \times 2^{-126}$ or, 0.015625×2^{-126}. The significant part in binary number system can be written as 0.000001. The number representation is shown in the following table.

sign	exponent	mantissa
0	0000 0000	0000 0100 0000 0000 0000 000
Hex: 0x0002 0000		

9.3.3 Not-A-Number number representation

Certain bit patterns in IEEE754 representation are named as NaN, or Not-A-Number. These numbers are represented by exponent field of the number being all 1s and the mantissa field being any value other than 0. As mentioned earlier, numbers with the exponent being all 1s and mantissa being all 0s represent $\pm\infty$. Thus representation of NaN is non-conflicting with any other valid numbers including $\pm\infty$. NaNs are used to handle errors in computations where the result may be undefined. For example, a division of 0 by 0 should lead to a NaN. Under IEEE754 representation, there is no distinction between the positive and negative values of the NaN and these are all ignored.

IA32 architectures further divide the NaNs into two categories, Signaling NaN (or SNaN) and Quiet Nan (or QNaN). In a nutshell, QNaNs are used as arguments to a computation when the computation should not raise any exception conditions and must result in a meaningful value (such as QNaN). When SNaNs are used as arguments to computations, exceptions of kind "Invalid Floating Point Value" are raised during the computations.

EXERCISES:

9.10 Given the following real numbers find out whether the number can be represented using IEEE754 single precision floating point number representation scheme or not. If it can be represented, find out whether the number will be normalized or denormalized.

a. -1.625×2^{-126} b. 0.00625×2^{-146} c. 0.625×2^{-146}
d. 1625.0×2^{-126} e. 0.01250×2^{-129} f. 0.5125×2^{-149}

9.11 For the following hexadecimal strings representing numbers in IEEE754 single precision floating point representation scheme, find out if the numbers are in denormalized form, normalized form or special numbers (such as ± 0, $\pm\infty$ or NaN). Also find the values of the real numbers represented by these.

a. 0xC1C4 0000 b. 0x42C0 0000 c. 0x8060 0000
d. 0xFF80 0000 e. 0x8000 0000 f. 0x7FF0 0000

9.4 Floating point numbers in IA32 architectures

IA32 architectures support IEEE754 number representation scheme of floating point numbers. There are three kinds of floating point numbers in IA32 architectures as shown in figure 9.1.

1. Single Precision Floating Point numbers
2. Double Precision Floating Point numbers, and

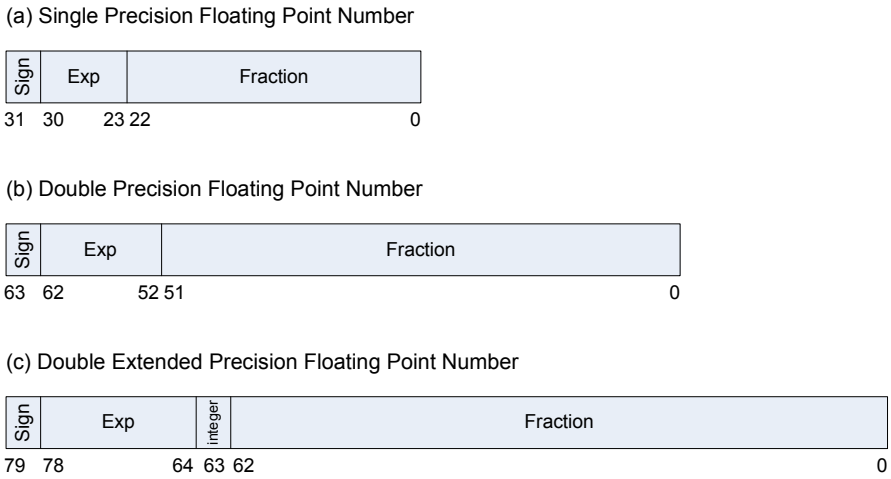


Figure 9.1: Floating Point Data Types in IA32 Processors

3. Double Extended-Precision Floating Point numbers.

9.4.1 Single Precision Floating Point Numbers

The single precision floating point numbers resemble the IEEE754 single precision floating point numbers. These are 32-bit wide numbers and have three parts – a sign, a mantissa and an exponent. Sign is one-bit wide and represents the sign of the number (0: Positive, 1: Negative). Mantissa is 23-bit wide and represents fractional part of the floating point number in binary. A leading one bit integer is not stored and is assumed as 1 in the normalized numbers and 0 in the denormalized numbers as in IEEE754 representation. Exponents are represented in excess-127 representation. The range of real numbers that can be represented using single precision positive floating point numbers is approximately from 2^{-126} to 2^{128} (in normalized form), or, approximately from 1.18×10^{-38} to 3.40×10^{38}, and $\approx 2^{-126}$ to 2^{-149} in denormalized form (1.17×10^{-38} to 1.4×10^{-45}). A similar range is represented by the negative floating point numbers. In addition to these numbers certain special values ± 0 and $\pm\infty$ can also be represented as in IEEE754 single precision numbers.

9.4.2 Double Precision Floating Point Numbers

Double Precision Floating Point Numbers of IA32 architectures are similar to the IEEE754 double precision floating point numbers. These are 64-bit (8-byte) wide numbers with sign, mantissa and exponent

in excess-1023 representation. The approximate range of real numbers represented by positive double precision normalized floating point numbers is from 2^{-1022} to 2^{1024}, or from 2.23×10^{-308} to 1.79×10^{308}. The range of the numbers in denormalized form is from 2^{-1022} to 2^{-1074} (or approximately from 2.2×10^{-308} to 4.94×10^{-324}.)

9.4.3 Double Extended-Precision floating point Numbers

Double extended-precision floating point numbers are 80-bit wide. This number format is used to perform computations internally in IA32 processors when instructions provided by x87 floating point unit are used. Numbers in double extended-precision format can be stored in the memory or can be loaded from the memory using x87 FPU instructions of IA32 architectures. All numbers in single precision or in double precision format are converted to double extended-precision format while loading them into a floating point register from the memory. Similarly the numbers in registers (in double extended-precision format) are first converted to single precision or double precision format before storing them in the memory.

Double extended-precision format numbers use 15-bit field to store exponents in excess-16383 format. Unlike in single and double precision representations, one bit integer part of the significand is also stored in double extended-precision representation as shown in figure 9.1(c). One bit is used to store this integer and 63 bits are used to store the fraction part. Thus a total of 64 bits are used to store the significand. One bit is used to store the sign of the number.

Double extended-precision format numbers can store numbers in a huge range from 2^{-16382} to 2^{16384} in normalized form (when the integer part of the significand is 1). The range of the numbers in denormalized form is 2^{-16382} to 2^{-16444}. This range translates to decimal range of 3.37×104932–1.18×10^{4932} in normalized form. In addition to these numbers, $\pm\infty$ and ± 0 are also representable as in IEEE754 representations.

9.5 Architecture of floating point processor

All IA32 processors provide support for the x87 instructions that operate on real numbers. These instructions operate in their own environment within the IA32 architecture. This environment includes 8 general purpose data registers, each 80-bit wide and capable of storing a real number in double extended-precision format. Historically these registers were known as x87 FPU data registers. Subsequent to the introduction of MMX technology (discussed in chapter 10), these registers were also used as MMX registers and therefore the use of these registers conflict with the usage of MMX instructions. In general, MMX

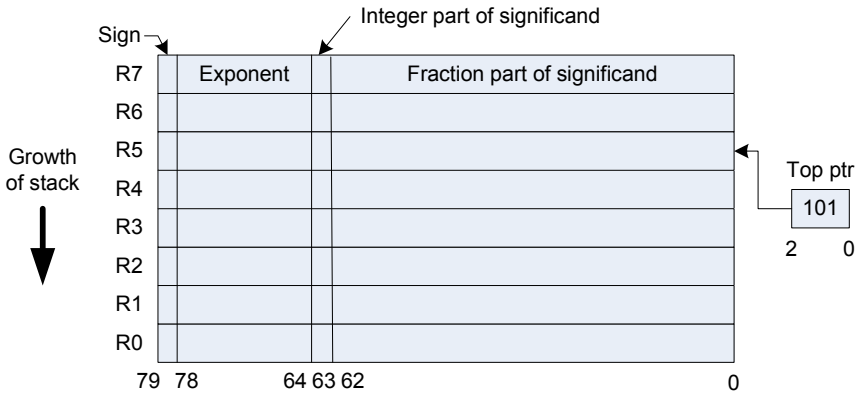


Figure 9.2: x87 Data Registers in IA32 Processors

instructions and x87 FPU instructions are executed in a mutually disjoint manner.

In x87 operating environment (figure 9.2), 8 general purpose registers are organized as a stack of registers. There is a 3-bit stack top indicator as shown in figure 9.2. The x87 operating environment also has a 16-bit status word register. The stack top indicator is essentially a 3-bit field in status word register and contains the index to the register assumed to be at the top of the stack. Most instructions in x87 environment operate on general purpose registers using register stack-relative addressing.

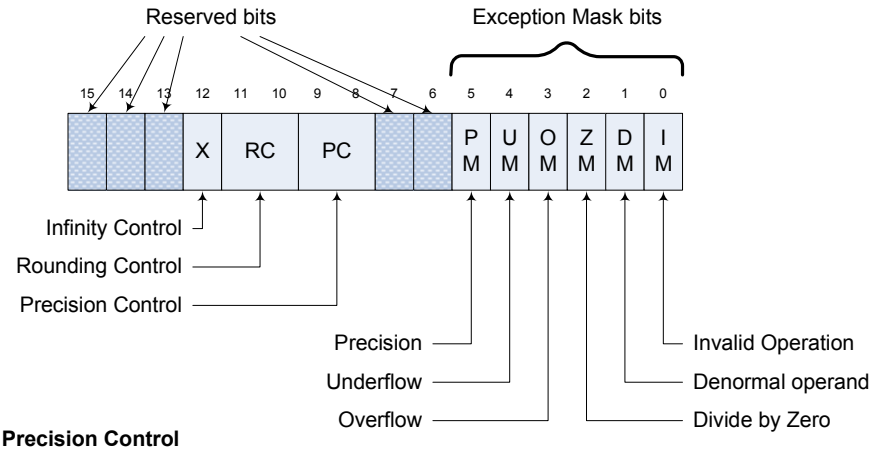
The values stored in general purpose x87 data registers are always in double extended-precision format. When an integer, BCD (binary-coded decimal), single precision or double precision memory operand is loaded into one of these registers, the operand is implicitly converted to double extended-precision format. Similarly values in the general purpose x87 data registers are converted to appropriate format (single precision, double precision or integer formats) before they are stored in memory.

In addition to the general purpose data registers, x87 operating environment also includes three 16-bit registers which are often used in controlling the computations. These registers are the following.

1. x87 Control Register

2. x87 Status Register

3. x87 Tag Register



Precision Control

00: Single Precision (24 bits)
01: Unused. Reserved.
10: Double Precision (53 bits)
11: Double Extended-Precision (64 bits). (Default)

Rounding Control

00: Round to nearest integer
01: Round down (towards $-\infty$)
10: Round up (towards $+\infty$)
11: Round towards zero (truncate)

Exception Mask

0: Unmask corresponding exception (cause exception to occur)
1: Mask corresponding exception (cause exception to not occur)

Figure 9.3: x87 Control Register

9.5.1 x87 Control Register

The control register in x87 FPU is used to control the way x87 instructions handle the precision and rounding of operands. During the computations, various floating point error conditions such as divide-by-zero may occur depending upon the operands of the instructions. The x87 FPU can be programmed to raise a floating point exception in case of such errors. These exceptions are handled in a special manner in the program. Various bits in the Control Register are used to individually mask the error conditions which may raise a floating point exception to the CPU. Format of the Control Register for x87 FPU is shown in figure 9.3.

Functionality of bits in the Control Register are classified in two groups, bits for exception masking and bits for computation control. The x87 FPU provides six different kinds of floating point exceptions. Occurrence of these exceptions can be masked by setting corresponding mask bit in the control register to 1.

Computation control bits are used to control precision and round-

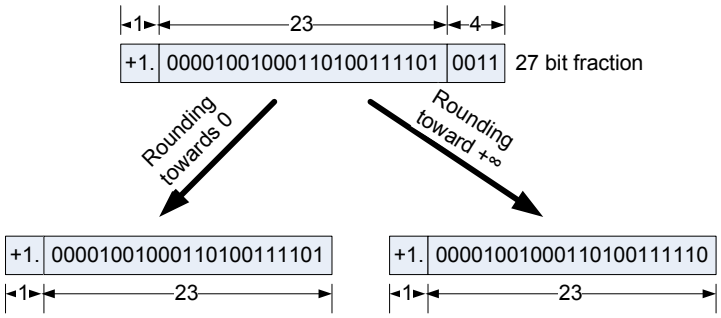
ing. In addition there is a bit for infinity control provided for compatibility with the historic x87 FPUs. Under GNU/Linux based computations, this bit should always be 0.

The precision control bits of control register are used to specify the precision (single precision, double precision or double extended-precision) of floating point computations. Usually, the best results are obtained using double extended-precision and are the most common choice for computation. During evaluation of a floating point expression, values of variables are loaded to the registers from memory. During such a load operation, these values are automatically converted to double extended-precision numbers. If intermediate computations are done with double extended-precision control the final results will have the least possible rounding off errors. It is for this reason it is possibly the best to leave precision control bits to 11 – to use double extended-precision.

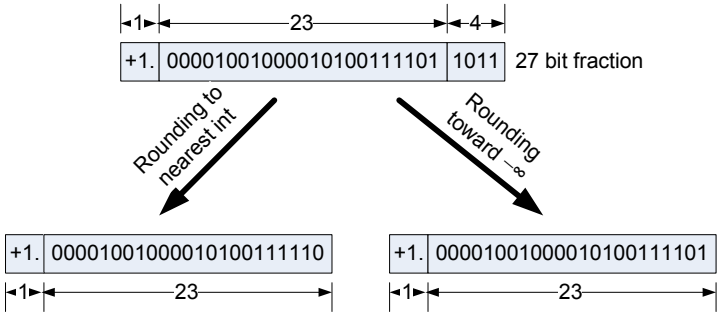
Rounding control bits of the control register are used to define the way rounding is performed during the computations. Rounding is needed when a number in the registers (in double extended-precision format) is to be converted to a lower precision format such as single or double precision floating point number. IEEE754 provides four different kinds of rounding controls, all of which are supported by the IA32 architectures.

Rounding controls are illustrated with an example (figure 9.4) where a number is converted to lower precision. Let us consider a positive number whose significant part is $1.000\,0100\,1000\,1101\,0011\,1101\,0011$ as shown in the figure 9.4(a). There are 27 significant bits after the binary point. In a single precision number only 23 bits can be stored after the binary point and therefore this number can not be represented in single precision format without losing the accuracy. In order to convert this number to single precision format, the last four bits 0011 have to be removed after rounding off. If the truncation mode or rounding-toward-zero method is used, these bits are just dropped (or converted to zero). Thus the rounded off significant part in single precision format would be $1.000\,0100\,1000\,1101\,0011\,1101$. However, if the rounding-toward-plus-infinity control is used, 0011 would be removed but a 1 will be added to the least significant bit of the remaining bits because the removed bit pattern is other than a zero. The rounded off number would therefore be $1.000\,0100\,1000\,1101\,0011\,1110$. The other two controls, namely rounding-toward-minus-infinity and rounding-toward-nearest will be identical to rounding-toward-zero in this example.

In a similar way, let's consider another 28-bit real number with 27 bit fraction as shown in figure 9.4(b). In this example, a real number in binary representation $1.000\,0100\,1000\,0101\,0011\,1101\,1011$ is rounded off to the nearest integer and toward $-\infty$. In this example, rounding-toward-zero will be identical to rounding-toward-minus-infinity and rounding-toward-plus-infinity will be identical to rounding-



(a) Rounding Controls (towards 0 and +∞)



(a) Rounding Controls (towards nearest and −∞)

Figure 9.4: Rounding Controls in IA32 architectures

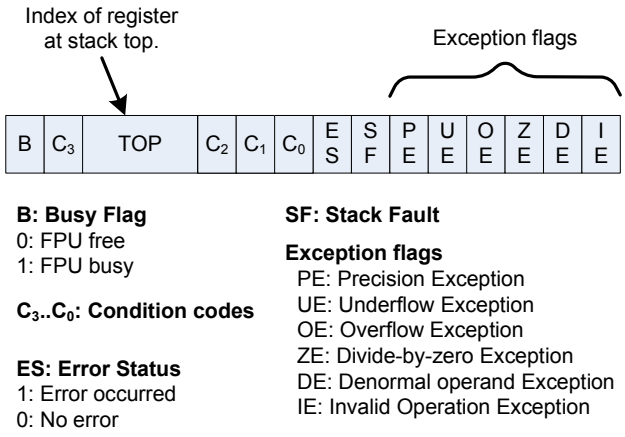


Figure 9.5: x87 Status Register

toward-nearest.

9.5.2 x87 Status Register

Status register in the x87 FPU indicates various exception flags, condition codes, stack top and such details. During the execution of x87 instructions, these status bits are set by the x87 FPU to indicate its state.

Most bits in the status register are set by the FPU but remain sticky. When a particular bit gets set to 1 to reflect an error condition and next instruction is executed which does not result in the same error, the corresponding status bit is not cleared. Thus upon execution of the subsequent instructions, status bits do not get cleared unless an explicit instruction is executed to clear them. Therefore the status bits reflect the cumulative errors since the last clearing of the bits and exhibit a property of being sticky.

As mentioned earlier, the data registers of x87 FPU are organized as stack of registers. Most instructions operate on registers using register stack addressing and produce result on top of the register stack. The register stack is implicitly managed by the x87 FPU during execution of various instructions. Index of the register at top of the register stack can be found in TOP field of the status register.

Four condition code flags C_0 to C_3 are generated during the execution of instructions. These indicate the results of floating point comparison and arithmetic operations. Status register can be saved in memory or taken into ax register. Thereafter regular integer compare and branch instructions may be used to operate according to the condition codes. As mentioned earlier, the condition code flag bits are sticky

in nature and reflect the cumulative status since the clearing.

Exception flags in the status register indicate various floating point error conditions that may occur during the execution of x87 instructions. These flags indicate cumulative errors since the last clearing of the status register. The exception flags are set independent of corresponding mask bits set in the control register. ES bit in status register indicates a summary condition of error. If any of the unmasked floating point exceptions occur during the computation, ES bit is set in the status register. For example, if an instruction causes divide-by-zero floating point exception, the ZE flag is status register is set to 1. However if the ZM bit in the mask register is 1, ES bit is not set due to this condition.

Stack fault (SF) bit in status register indicates errors due to stack overflow or underflow conditions while operating on x87 data registers. For example, when there is only one data item on the stack and an addition operation is performed which takes two data items from the stack, a stack fault occurs and SF bit is set to 1. This fault is due to the underflow of the stack. Similarly, when all the eight registers are in use for the stack and an attempt is made to load an item, stack overflow occurs. During stack fault conditions, C_1 bit indicates specific error condition (1: overflow, 0: underflow). The SF bit is also a sticky bit similar to other exception flags and indicates cumulative error since the last clearing of status register.

FPU busy bit (B) is provided in IA32 architectures for historic compatibility only and most programs do not use this bit. The x87 FPU instructions can operate in parallel with general purpose instructions. In order to ensure that no inconsistencies occurred while operating in parallel, old x87 FPU implementations used this flag which could be checked by the softwares. Modern implementations of IA32 architectures handle operations between x87 FPU and general purpose architecture in a better way and do not require the busy bit to be monitored by the programs.

9.5.3 x87 Tag Register

Each data register of x87 FPU has an associated 2-bit tag contained in a tag register (figure 9.6). Code in the tag register indicates the type of the value of the corresponding data register. If the data register contains a valid normalized value, corresponding tag code is 00. Other values of the tag indicate whether the data register has a zero value (whether positive or negative), or whether the data register has a special value (NaN, $\pm\infty$ or a number in denormalized form), or whether the data register has no value. Tag values are used to raise stack faults. In case an instruction uses a value from an empty register, stack underflow fault is raised. Similarly when an instruction load a value on the stack

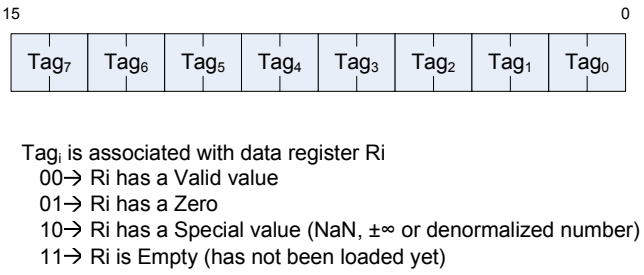


Figure 9.6: x87 Tag Register

top and find the corresponding register non-empty, stack overflow fault is generated.

The values in the tag registers are used to decode the values of data registers without performing expensive and complex data decoding.

9.5.4 Floating Point Exceptions

IA32 architectures detect six floating point exceptions during the floating point computations.

1. Invalid Operation Exception (IE)
2. Divide-by-zero Exception (ZE)
3. Denormalized Operand Exception (DE)
4. Numeric Overflow Exception (OE)
5. Numeric Underflow Exception (UE)
6. Precision (inexact result) Exception (PE)

The first three exceptions are raised depending upon the values of input operands. These are therefore handled before any computation is carried out. The last three exceptions are raised after the computation is performed on the basis of the value of the result.

The processing of exceptions depends upon the value of the corresponding mask bit in the control register. When condition of an exception is detected and that particular exception is masked, only the corresponding flag bit is set in the status register. The computation is however carried out in a reasonable manner with results being NaN or ∞ or a denormalized number. When the exception is unmasked, ES bit in the x87 status register is set and a software exception handler is invoked. In GNU/Linux operating system, the exceptions are usually masked.

Invalid operation exception is raised when operands of the instruction contain invalid values such as a NaN. IA32 architectures provide two kinds of NaNs, called SNaN and QNaN. SNaNs are called Signalling NaNs while QNaNs are called Quiet NaNs. In particular, the IE exception is raised in response to one of the operands being SNaN.

Behavior of the processor in case of IE exception depends on the mask bit in the control register. If IE exception is masked, the processor performs the operation and produces a QNaN as result of the operation. If the IE exception is unmasked, the exception causes a software handler to be invoked and the operands are not changed. In either case, IE flag in the status register is set. Various kind of operations such as multiplication of 0 by $\pm\infty$, division of $\pm\infty$ by $\pm\infty$ etc., raise IE exception.

Denormalized operand exception is raised when one of the operands of an instruction is in denormalized form. Under GNU/Linux this exception is masked. In this case, the processor performs computations using the denormalized number and produce results which may be in denormalized, normalized or any other special form. In case the exception is not masked computation is not carried out and a software handler is invoked.

Divide-by-zero exception is reported by the instructions that perform division when the divisor contains a zero and the dividend is a finite number other than 0. A zero divided by zero generates IE exception rather than a ZE exception. Similarly a NaN divided by 0 gives IE exception rather than a ZE exception. When ZE exception is masked, the result is produced as $+\infty$ or $infty$.

Numeric overflow exception (OE) is caused by an instruction when the post-rounding result would not fit into the destination operand. For example, when a large double-extended precision floating point number is (number $> 2^{128}$) is to be stored in a single precision floating point format, a numeric overflow is set to have occurred. If the exception is masked, the results are set to values depending upon the rounding control. For example, if the rounding control is toward $+\infty$, the result would be either $+\infty$ or the smallest negative finite number. The smallest negative finite number is negative of the largest positive finite number and is indicated by sign bit being 1, exponent being $11\ldots10$ and mantissa being all 1s.

When the rounding control is toward-nearest, the result is $\pm\infty$ depending upon the sign of the true result. When the rounding control is toward $-\infty$, the result is either the largest positive finite number or $-\infty$ depending upon the sign of the true result. Finally, when the rounding control is toward-zero, the result is either the largest positive finite number or the smallest negative finite number depending upon the sign of the true result.

Numeric Underflow exception (UE) is reported when the offending instruction generates a result whose magnitude is less than the small-

est possible normalized number for that representation. When this exception is masked, the result is converted to denormalized form and possible an inexact value is stored.

Precision exception (PE) is generated when a number can not be represented in the format. For example when a division operation is performed involving two floating point numbers 1.0 and 12.0, the results is a recurring infinite sequence of bits. This number can not be stored in any format without loss of precision. PE exception is a frequently occurring exception the most reasonable handling of this exception is to store the inexact result as per the rounding control. This kind of handling is done by the processor when PE exception is masked in the x87 Control Register.

9.5.5 x87 register addressing

x87 data registers are organized as a stack of registers. The top of the stack is indicated by the TOP field in x87 status register. These data registers are addressed using a notation `st(`i`)` where i is a constant between 0 to 7 and indicates a register 'i' locations below the register stack top. In GNU/Linux assembly language, all registers are prefixed by a `%`. Therefore registers `st(`i`)` are also prefixed by a `%` in the assembly language.

The register stack uses registers in a circular buffer. Thus if the TOP field in the x87 status register has a value 5, then `st(0)` refers to x87 data register R5. Register `st(1)` refers to register R4. As the stack of registers is organized in a circular buffer, register `st(6)` indicates data register R7. Two such examples are shown in table 9.5 where corresponding data registers are listed for all stack registers when TOP field contains 5 and 2 respectively.

Table 9.5: Register Stack addressing in x87 FPU

Stack register	Data Register when TOP=5	Data Register when TOP = 2
`st(0)`	R5	R2
`st(1)`	R4	R1
`st(2)`	R3	R0
`st(3)`	R2	R7
`st(4)`	R1	R6
`st(5)`	R0	R5
`st(6)`	R7	R4
`st(7)`	R6	R3

In stack based computations, stack top is used very frequently. In x87 instructions, the stack top can also be indicated as `st` which is a

short hand representation for `st(0)`.

Many instructions in x87 FPU use implied addressing where the stack top is implied as an operand of the instruction.

9.6 Floating point instructions

In context of floating point instructions, certain addressing modes (refer section 2.2) are not available. In particular none of the x87 FPU instructions use immediate addressing mode for the operands. The register addressing mode as described in section 2.2 is also not used. These instructions use double extended-precision x87 general-purpose data registers as a stack of registers. Memory addressing modes as given in section 2.2 are also used by x87 instructions whenever an operand is in memory.

x87 floating point instructions are grouped in the following categories.

1. Basic arithmetic instructions
2. Constant loading instructions
3. Trigonometric, logarithmic and exponentiation instructions
4. Data comparison instructions
5. Data transfer instructions
6. FPU control instructions

9.6.1 Basic arithmetic instructions

Basic arithmetic instructions are used to perform operations such as addition, subtraction, multiplication and division involving numbers in floating point formats. All of such operations require two source and one destination operands. In these instructions, one source operand is always the stack top `st`. The other source operand can be either in memory or in one of the registers on register stack. Destination operand is always a register, either on top of the register stack or any other register depending upon the instruction format. We start with the addition of two floating point numbers.

Addition instructions

There are three kinds of addition instructions in x87 FPU.

```
fadd memVar
fadd src, dest
faddp
faddp dest
fiadd memVar
```

In the first form of `fadd` instruction, a memory variable is used. This memory variable is assumed to contain a floating point number, either in single precision or in double precision floating point format. In order to provide this size, the instruction `fadd` can be suffixed by an '`s`' or an '`l`' representing single precision and double precision floating point numbers respectively. Some versions of the GNU `as`, assume the memory operand to contain single precision floating point number if no suffix is added to the instruction. Memory variable may be provided in one of the memory addressing modes as discussed in section 2.2.

In the second form of `fadd` instruction, two register operands are used. One of these two register operands must be the stack top – either `st` or `st(0)`. For example, instruction `fadd %st, %st(2)` adds two floating point numbers, first one stored on the top of the stack and the second one two locations underneath; and puts the result in `st(2)`. Similarly, instruction `fadd %st(5), %st` adds two floating point numbers available in registers `st(5)` and `st(0)` and stores the result in register `st(0)`.

The `faddp` instruction adds two numbers and pops the top of the stack. In the first form, All source and destination operands are implied. It adds two numbers on the top of the stack (i.e. `st(0)` and `st(1)`), adjusts the stack top by decrementing it by 1 and puts the result in new `st(0)`. Since the stack top pointer is reduced by 1, old `st(1)` becomes the new `st(0)`. Therefore, this instruction equivalently removes two stack variables, adds them and puts the result on the stack.

In the second form of `faddp` instruction, one register is specified which is one of the sources and the destination of the addition. The other source is on the stack top. After the addition is performed, stack top is decremented by 1. Thus the result would be available in `st(i-1)` if the source was in register `st(i)`.

Finally the last instruction `fiadd` is used to add an integer to the floating point number on the stack top. The integer operand for `fiadd` is taken from memory using one of the addressing modes discussed in section 2.2. The integer operand can be 32-bit (long) or 16-bit (word). A suffix of '`l`' and '`s`' need to be added to the instruction `fiadd` to indicate the size of the integer variable as 32-bit and 16-bit respectively.

There are certain special cases related to the floating point additions. When the sum of the two operands is 0, it is stored as $+0$ except the case when rounding-toward-minus-infinity rounding control is used. In this case, the result is -0. When $\pm\infty$ are involved in the

addition, the result is either $\pm\infty$ or a NaN. When $+\infty$ is added to $-\infty$, NaN is the output and IE exception is raised. When $+\infty$ is added to any valid number other than $-\infty$, the result is $+\infty$. Similarly $-\infty$ is the output when a valid number other than $+\infty$ is added to $-\infty$. If any of the operand of the addition is a NaN, the result is also a NaN.

After the addition, condition flags C_1 is set to 0 if there was a stack underflow. In this case, the SF bit in the status register is also set. All other condition flags are left undefined after an addition. Addition operations may also raise IE, SF, DE, UE, OE and PE exceptions depending upon the input operands and the result of the addition.

Some examples of the floating point addition instructions are given below.

`faddl abc`	Add a double precision floating point number stored in 64-bit memory location `abc` to the stack top.
`fadds x`	Add a single precision floating point number in a 32-bit memory location `x` to the stack top.
`fadd %st, %st(3)`	Add a floating pointer number in register `st` to register `st(3)`.
`fadd %st(6), %st`	Add floating point number in `st(6)` to `st`
`faddp %st(3)`	`st(3) = st(3) + st`. Stack pointer is decremented and therefore the result will be known as `st(2)`.
`faddp`	`st(1) = st(1) + st`. After stack pointer adjustment, the result is on the stack top and is known as `st`.
`fiadds m16`	Add a 16-bit integer (note 's' suffix) stored at memory location `m16` to `st`.
`fiaddl m32`	Add a 32-bit integer stored in memory location `m32` to a floating point number in `st`. (note 'l' suffix to `fiadd`)

Subtraction instructions

There are various different kinds of subtraction instructions in x87 FPU.

```
fsub memVar
fsub src, dest
fsubp
fsubp dest
fisub memVar
fsubr memVar
fsubr src, dest
fsubrp
fsubrp dest
fisubr memVar
```

Addressing modes and basic functionality of the subtraction instructions are similar as in the addition instructions. Subtraction operation is available in two forms, normal subtraction (`fsub`, `fsubp`, `fisub` instructions) and reverse subtraction (`fsubr`, `fsubrp`, `fisubr` instructions). While the operations performed by normal subtraction instructions are `dest = dest − src`, the operations performed by the reverse subtraction instructions are `dest = src − dest`. In the first form of `fsub` instruction, the `dest` is `st(0)`. In the second form of `fsub` instruction, either the `src` or the `dest` must be `st(0)`. In the first form of `fsubp` instruction, `dest` is top of the register stack `st(0)` while the other implied source is `st(1)`. After the execution of the instruction, stack depth is reduced by 1 and the result is left on top of the register stack. In the second form of the `fsubp` instruction, the `src` is implied as top of the register stack. This instruction subtracts the `src` from `dest`, leaving the result in `dest` and then removes one item from top of the register stack. If the original `dest` operand was `st(3)`, it would be known as `st(2)` after the execution and will contain the updated value. Instruction `fisub` subtracts an integer, 16-bit or 32-bit, from top of the register stack `st(0)`. 16-bit integer is indicated by a 's' suffix in `fisubs` instruction. Similarly, 32-bit integer is indicated by a 'l' suffix in `fisubl` instruction.

`fsubr` instruction in the first form must be suffixed by 's' or 'l' to indicate a single precision or double precision floating point number in memory. In this form of `fsubr` instruction, top of the register stack is subtracted from the source and the result is put back on the top of the register stack. In the second from of the `fsubr` instruction, either the `src` or the `dest` operand must be top of the register stack `st(0)`. In this form the `dest` operand is subtracted from the `src` operand and the result is put in `dest`. The first form of the `fsubrp` instruction subtracts `st(1)` from `st(0)` and puts the result in `st(1)`. After the subtraction, one item is removed from the register stack and hence the result will be known as `st(0)`. The second form of the `fsubrp` instruction subtracts its `dest` from top of the register stack and puts the result back in `dest`. After subtraction, one item is removed from the register stack decreasing the stack depth by 1. Instruction `fisubr` must be suffixed

by 's' or 'l' to indicate 16-bit or 32-bit integer in memory. Top of the register stack is decremented from the corresponding integer and the result is replaced on the top of the register stack.

Some examples of floating point subtraction instructions are given below.

`fsubs a`	Subtract a single precision floating point (32-bit) number stored at memory location `a` from `st(0)`. Result goes back in `st(0)`.
`fsubl a`	Subtract a double precision floating point (64-bit) number stored at memory location `a` from `st(0)`. Result is stored in `st(0)`.
`fsub st(3), st(0)`	`st(0)` = `st(0)` – `st(3)`
`fsub st(0), st(4)`	`st(4)` = `st(4)` – `st(0)`
`fsubp`	`st(1)` = `st(1)` – `st(0)`. After the operation, one item is removed from the register stack and hence `st(1)` becomes `st(0)`.
`fsubp st(3)`	`st(3)` = `st(3)` – `st(0)`. After the operation, one item at top is removed from the register stack and hence `st(3)` becomes `st(2)`.
`fisubs b`	`st(0)` = `st(0)` – 16-bit integer from memory location `b`.
`fisubl m32`	`st(0)` = `st(0)` – 32-bit integer from memory location `m32`.
`fsubrl a`	`st(0)` = Double precision floating point number stored at memory location `a` – `st(0)`.
`fsubr st(3), st(0)`	`st(0)` = `st(3)` – `st(0)`.
`fsubrp`	`st(1)` = `st(0)` – `st(1)`. After the operation, one item is removed from the register stack and hence `st(1)` becomes `st(0)`.
`fsubrp st(2)`	`st(2)` = `st(0)` – `st(2)`. After the operation, one item at top is removed from the register stack and hence `st(2)` becomes `st(1)`.
`fisubrl a1`	`st(0)` = 32-bit integer from memory location `a1` – `st(0)`.

Multiplication and division instructions

x87 FPU provides multiplication instructions in formats similar to the addition instructions. Division instructions are available in two forms – normal division and reverse division – in a way similar to the subtraction instructions. Following are the multiplication and division instructions.

```
fmul memVar
fmul src, dest
fmulp
fmulp dest
fimul memVar
fdiv memVar
fdiv src, dest
fdivp
fdivp dest
fidiv memVar
fdivr memVar
fdivr src, dest
fdivrp
fdivrp dest
fidivr memVar
```

In the instructions where both `src` and `dest` operands are specified, one of the operand must be `st(0)`.

Some examples of the multiplication and division instructions are the following.

`fmuls sp`	`st(0) = st(0)` $*$ Single precision floating point number stored at memory location `sp`.
`fmul st(3), st(0)`	`st(0) = st(0)` $*$ `st(3)`.
`fmulp`	`st(1) = st(1)` $*$ `st(0)`. After the operation, one item is removed from the register stack and hence `st(1)` becomes `st(0)`.
`fmulp st(6)`	`st(6) = st(6)` $*$ `st(0)`. After the operation, one item at top is removed from the register stack and hence `st(6)` becomes `st(5)`.
`fimuls m16`	`st(0) = st(0)` $*$ 16-bit integer from memory location `m16`.
`fdivl dp`	`st(0) = st(0)` / Double precision floating point number stored at memory location `dp`.
`fdiv st(0), st(3)`	`st(3) = st(3)` / `st(0)`.
`fdivp`	`st(1) = st(1)` / `st(0)`. After the operation, one item is removed from the register stack and hence `st(1)` becomes `st(0)`.
`fidivl m32`	`st(0) = st(0)` / 32-bit integer from memory location `m32`.
`fdivrl a`	`st(0) =` Double precision floating point number stored at memory location `a` / `st(0)`.
`fdivr st(3), st(0)`	`st(0) = st(3)` / `st(0)`.
`fdivrp st(3)`	`st(3) = st(0)` / `st(3)`. After the operation, one item at top is removed from the register stack and hence `st(3)` becomes `st(2)`.
`fidivrs abc`	`st(0) =` 16-bit integer from memory location `abc` / `st(0)`.

All addition, subtraction, multiplication and division instructions generate the floating point exceptions based on their input operands and result. IE exception is raised when the operands are invalid for the operation. For example, when the two operands in multiplication operation are ± 0 and $\pm\infty$, an IE exception is raised. Similarly when ± 0 is divided by ± 0 or when $\pm\infty$ is divided by $\pm\infty$, IE exception is raised. IE exception is also raised when one of the operands is SNaN.

ZE exception is raised only by the division instructions when the divisor is ± 0 and the dividend is a number other than ± 0, $\pm\infty$, or a NaN.

DE exception is raised when the result is in denormalized form. UE exception is raised when the result is too small for the destination format. OE exception is raised when the result is too large for the destination format. PE exception is raised when the result can not be

represented without loss of accuracy. In this case, condition flag C_1 indicates the rounding direction as well.

Like in other instructions, these instructions may also raise a stack fault exception (SF bit in the status register). Stack overflow and underflow is indicated by condition flag C_1 in case of a stack fault exception.

Remainder Computation

x87 FPU instruction set includes two instructions for computation of remainder of division operation when one floating point number is divided by another. These instructions are the following.

```
fprem
fprem1
```

These instructions compute the remainder when `st(0)` is divided by `st(1)` and return the remainder in `st(0)`. The value of dividend stored previously in register `st(0)` is overwritten. These instructions essentially compute remainder using the expression $(\texttt{st(0)} - p \times \texttt{st(1)})$, where p is an integer obtained by the division of `st(0)` by `st(1)`. The two instructions differ in the way they compute integer p. In `fprem` instruction, integer p is computed by truncating the division result to an integer. In `fprem1` instruction, this integer is computed by rounding the division result to the nearest integer.

In reality, these two instructions to compute remainder do not return the real remainder. These instructions implement what is known as partial remainder. When the exponents of the two numbers in `st(0)` and `st(1)` differ by more than 64, the instructions perform a reduction of dividend in `st(0)` in such a way that it could be used as dividend in successive execution of the same instruction. At each step, the instructions will definitely reduce the value stored in `st(0)`. If the remainder is computed completely (the reduction is complete), condition flag C_2 is set to 0. Otherwise this flag is set to 1.

The following is a code example of computing the actual remainder of two numbers stored in registers `st(0)` and `st(1)`.

```
.globl Remainder
Remainder:
    fprem1     // Partial remainder in IEEE754 style
    fstsw %ax  // Save flags in register ax
    sahf       // Store flags in EFLAGS register
    // At this point C2 flag of x87 status register
    // gets in PF bit of EFLAGS register.
    jp    Remainder // Execute again if reduction incomplete
    ret
```

Certain instructions in this code fragment are not discussed yet. In particular, the instruction `fstsw` is used to store the status word of x87 FPU in CPU register `ax`. Upper eight bits of register `ax` (register `ah`) are then stored in lower 8 bits of eflags register (figure 2.2) using `sahf` instruction. Thus condition flags C_0, C_2 and C_3 get transferred to CF, PF and ZF in `eflags` register respectively. Later `jp` instruction is used to transfer control conditionally to compute partial remainder again. The other way to check for the conditional flags is to use `test` instruction to check for the value in register `ax`.

The standard GNU/C library provides `fmod`, `fmodf` and `fmodl` function calls whose behavior is implemented by `fprem` instruction. It also provides another function call `drem` whose behavior is similar to the one implemented by `fprem1` instruction.

There are some properties of the results of these instructions after the reduction is completed. While using the `fprem` instruction, the sign of the results is always the same as that of the dividend in `st(0)`. The magnitude of the result is always less than the magnitude of the divisor. On the other hand while using `fprem1` instruction, magnitude of the remainder is always less then or equal to the half of magnitude of the divisor. The result is always between $\pm d/2$, where d is the divisor.

The following table gives some examples of values returned by two different versions of the remainder computing instructions for the same divisor and dividend.

Dividend	Divisor	Remainder by	
		`fprem`	`fprem1`
3.00	0.80	0.60	−0.20
−3.00	0.80	−0.60	0.20
−3.00	−0.80	−0.60	0.20
3.00	−0.80	0.60	−0.20

EXERCISES:

9.12 Write functions in Assembly language which are equivalent to the C library functions `fmod`, `fmodf`, `fmodl` and `drem`. You must use the `frem` and `frem1` instructions. Assume that the input arguments are available on the register stack at locations `st(0)` and `st(1)`.

Square root Computation

Square root of a number on the top of the register stack `st(0)` is computed using a single instruction `fsqrt` in x87 FPU instruction set.

```
fsqrt
```

This instruction returns the result in register `st(0)` overwriting the value stored previously.

In case the original number is negative other than -0, an IE exception is raised. When the input is either a ± 0, or a $+\infty$ or a NaN, the square root value is the same as the input.

Miscellaneous Computations

The following instructions operate on `st(0)` and return their values in the same register overwriting the value stored previously.

```
fabs
fchs
frndint
```

The `fabs` instruction is used to compute absolute value of register `st(0)`. It always returns a positive floating point number and may raise only a stack fault.

The `fchs` instruction is used to change the sign of the input argument on top of the register stack, `st(0)`.

The `frndint` instruction rounds the source value on the register stack top to the nearest integer as per the rounding control bits in the x87 control word. The result is returned in floating point format in `st(0)`.

In addition to the above mentioned instructions, the following two instructions are also available which take implicit arguments.

```
fxtract
fscale
```

The `fxtract` instruction extracts the exponent and significand of its input operand on top of the register stack `st(0)`. It removes one operand from the stack and puts two arguments on the stack. After the execution of this instruction, the significand is left on top of the register stack in `st` while the exponent is left at one location below in `st(1)`. This instruction therefore effectively increases the depth of the register stack by 1 and may raise a stack overflow exception (stack fault with C_1 = 1). The instruction also needs to remove one item from the stack and may cause a stack underflow as well.

`fscale` instruction takes two operands on the stack (i.e. `st(0)` and `st(1)`) and replaces the top of the stack by the result of computation $\texttt{st(0)} \times 2^{\texttt{st(1)}}$. Thus this instruction effectively performs an inverse computation of `fxtract` instruction. However while `fxtract` instruction increases the stack depth by 1, `fscale` instruction does not change the stack depth.

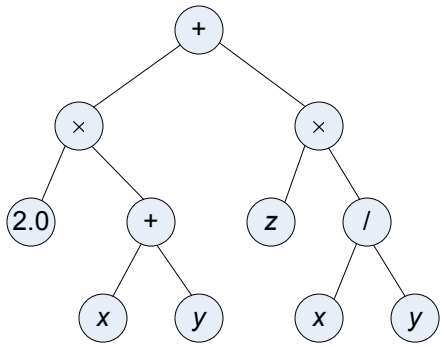


Figure 9.7: Expression tree for $(2.0(x+y)+z.(x/y))$.

9.6.2 Evaluation of floating point expression

The best method to evaluate an arbitrary floating point expression is to write it using stack computations. The x87 FPU is organized as a stack machine and is therefore very effective in evaluating expressions using stack based computations. For that purpose, an expression may be converted to postfix form and then evaluated using stack. For example, let's consider an expression $(2.0(x+y)+z.(x/y))$. The expression tree for this expression is shown in figure 9.7. The corresponding postfix representation of the expression is $(2.0\ x\ y\ +\ \times\ z\ x\ y\ /\ \times\ +)$.

On a stack computer this expression can be evaluated by a simple algorithm. The postfix expression is scanned from left to right. When a number or a variable is encountered, it is pushed on the stack. When an operator is encountered, enough arguments are popped from the stack, operation is performed and the result is pushed on the stack. These steps are illustrated in table 9.6 when it is applied on the postfix representation of expression of figure 9.7.

Table 9.6: Evaluation of $(2.0(x+y)+z.(x/y))$ using stack. Postfix representation of the expression is $(2.0\ x\ y\ +\ \times\ z\ x\ y\ /\ \times\ +)$.

Scanned item	Operation performed
2.0	Push 2.0 on stack. Stack = (2.0)
x	Push x on the stack. Stack = $(x, 2.0)$
y	Push y on the stack. Stack = $(y, x, 2.0)$
$+$	Remove two values from the stack, add them and put result back on stack. Stack = $(x+y, 2.0)$
$\times$	Remove two values from the stack, multiply them and put result back on stack. Stack = $(2.0(x+y))$
z	Push z on the stack. Stack = $(z, 2.0(x+y))$

x	Push x on the stack. Stack = $(x, z, 2.0(x+y))$
y	Push y on the stack. Stack = $(y, x, z, 2.0(x+y))$
$/$	Remove two values from the stack, divide and put result back on stack. Stack = $(x/y, z, 2.0(x+y))$
$\times$	Remove two values from the stack, multiply them and put result back on stack. Stack = $(z.(x/y), 2.0(x+y))$
$+$	Remove two values from the stack, add them and put result back on stack. Stack = $(2.0(x+y)+z.(x/y))$

At the end of the steps of the algorithm, top of the register stack contains the result.

The same algorithm can easily be implemented using x87 floating point instructions as shown in the following program. Some of the instructions used in this program are not yet explained and will be covered later in this chapter.

```
// Assume 2.0, x, y and z are single precision floating
// point numbers stored in memory locations two, x, y and
// z respectively. Result is left on stack top (%st(0))
expr_eval:
    finit       // Initlialize FPU
    flds   two  // Load 2.0 on stack top
    flds   x
    flds   y    // Stack = y, x, 2.0
    faddp       // Add and pop
    fmulp       // Multiply and pop
    flds   z    // Stack = z, 2.0(x+y)
    flds   x    // Stack = x, z, 2.0(x+y)
    flds   y    // Stack = y, x, z, 2.0(x+y)
    fdivrp      // Divide reverse and pop
    fmulp       // Stack = z.(x/y), 2.0(x+y)
    faddp       // Stack top = result
    ret
```

EXERCISES:

9.13 Write Assembly language program fragments to compute the following expressions involving real numbers.
(i) $x + y * z + 2z$ (ii) $x^2 - yz + y$ (iii) $2.0 + (x/(y-z))$

9.6.3 Constant loading instructions

In the x87 FPU instruction set, there are certain instructions that can load commonly used constants on the stack top. These instructions are the following.

```
fld1
fldz
fldpi
fldl2e
fldln2
fldl2t
fldlg2
```

These instructions are used to load constants on top of the register stack. Various constants are +1.0 (`fld1`), +0.0 (`fldz`), π (`fldpi`), $\log_2 e$ (`fldl2e`), $\log_e 2$ (`fldln2`), $\log_2 10$ (`fldl2t`), $\log_{10} 2$ (`fldlg2`). Since the depth of the stack may increase by 1, these instructions may cause a stack overflow fault.

9.6.4 Trigonometric, log and exponentiation instructions

Trigonometric instructions

In x87 FPU instruction set, following instructions are available that compute trigonometric functions for arguments stored on the stack top.

```
fsin
fcos
fsincos
fptan
fpatan
```

The `fsin` and `fcos` instructions compute sine and cosine of an angle in radians provided in `st(0)`. After the computation, register `st(0)` is replaced by the result of the function. If the value of the input angle is $\pm\infty$, an IE exception is raised. Otherwise the value of the function computed is between ± 1. A NaN is returned if the input angle was a NaN.

Unlike `fsin` and `fcos` instructions, `fsincos` instruction computes and leaves two values on the register stack. It removes the angle from the top of the register stack, computes sine and cosine and pushes them on the register stack. Upon completion of this instruction, stack depth is one more than the original depth. The top of the stack contains cosine of the angle while a location below (the one that originally had the angle) contains sine of the angle.

`fptan` instruction removes the angle provided on top of the register stack and computes its tangent. It then pushes the result (overwriting the angle) and a constant 1.0 (increasing the stack depth by 1) on the stack. Pushing of 1.0 is for historic compatibility and simplifies the

computations of functions such as cotangent by issuing `fdivr` instruction immediately after the `fptan` instruction.

Trigonometric instructions `fsin`, `fcos`, `fsincos` and `fptan` accept a range of input angle to be between -2^{63} to 2^{63}. If the input angle is outside this range, conditional flag C_2 is set and the stack top is not modified. Programs usually can take the remainder of the angle divided by 2π before computing the value of the function.

An example of program that computes $\sin(\theta)$ where θ in radians is made available in `st(0)` is given below. The program is simple to understand. First constant π is loaded twice on the stack and then added. The stack top would then contain 2π and the location below this would have the angle θ in radians. By swapping the top two locations, we are ready to compute the remainder. The instruction to evaluate remainder `fprem1` only computes partial remainder. Hence a method as described earlier is used to evaluate the remainder. Having computed the remainder, sine is computed and then 2π is removed from the stack.

```
// Compute sine of an arbitrary angle.
// Angle is available in st(0)
sin:
    // Prepare to compute remainder of x/(2.pi)
    fldpi      // Load pi twice on stack and add
    fldpi
    faddp      // Stack contains 2.pi and x
    fxch       // Swap st(1) and st(0)
sin_rem:
    fprem1     // Partial remainder (IEEE754 std.)
    fstsw %ax  // take condition flag to eflags
    sahf
    jp sin_rem // Execute again till reduction complete.
    fsin       // Stack = (sin(x), 2.pi)
    // Remove 2.pi from the stack
    fstp %st(1) // Copy sin(x) to st(1) and pop
    ret
```

`fpatan` instruction computes the arctangent. It removes two values stored on the stack (`st(0)` and `st(1)`), computes $\arctan($`st(1)`$/$`st(0)`$)$ and pushes the result back on the register stack modifying the original value of `st(1)`. Stack depth is reduced by 1 after the execution of this instruction. The resulting angle on top of the register stack, `st(0)`, is in radians.

Logarithmic instructions

There are following two instructions in x87 FPU instruction set to compute logarithm.

```
fyl2x
fyl2xp1
```

Both of these instructions take two implied arguments x and y on the register stack in registers `st(0)` and `st(1)` respectively. After the successful execution of this instruction, both operands are removed and the result is pushed on the register stack. Thus at the end of the execution, register stack depth is reduced by 1 and `st(0)` contains the result. The value in original register `st(1)` is therefore overwritten by the result.

`fyl2x` instruction computes $y.\log_2 x$ with y and x being in registers `st(1)` and `st(0)` respectively. The instruction may indicate a stack overflow or underflow fault in the status register.

`fyl2xp1` instruction computes the expression $y.\log_2(x+1)$ with y and x being in registers `st(1)` and `st(0)` respectively. In order to use this instruction, the value of x must lie within the range $\pm(1-1/\sqrt{2})$. The results are undefined for x being outside this range.

These two instructions along with other instructions to load constants can be used to compute natural logarithm or logarithm on base 10. Let's consider this with the following relation between the logarithmic functions.

$$\log_{10} x = \log_2 x \times \log_{10} 2$$

Therefore in order to compute $\log_{10} x$, registers `st(1)` and `st(0)` may be loaded with $\log_{10} 2$ and x respectively and instruction `fyl2x` may be executed. The following code is an implementation of this.

```
// Compute log(x) on base 10.
// Number x in single precision format is given on
// the stack at location 4(%esp).
// At the end result is given in %st(0)
log10:
    fldlg2  // load constant log(2) base 10.
    flds 4(%esp) // Load single precison x
    fyl2x   // Result on the stack top
    ret
```

Instruction for exponentiation

There is one instruction in x87 FPU instruction set to compute $2^x - 1$ where x is provided in register `st(0)`.

```
f2xm1
```

After the execution of `f2xm1` instruction, the result is left on the stack top `st(0)` overwriting the value of x stored prior to the execution. The acceptable range of values for x is between -1.0 and $+1.0$. The result is undefined for a value of x outside this range.

This instruction along with other instructions such as, `fscale`, `fxtract`, `fyl2x` and other integer instructions can be used to compute any arbitrary exponentiation.

9.6.5 Data comparison instructions

Floating point number can be compared in several different ways with other floating point numbers or integers. The result of the comparison is stored in either `eflags` register or in floating point conditional flags C_3 to C_0. There are three broad categories of comparison instructions.

1. Normal Comparison instructions that modify condition flags in x87 status register.
2. Unordered comparison instructions that modify flags in x87 status register.
3. Normal and Unordered comparison instructions that modify flags in `eflags` register.

The instructions in the first category are the following.

```
fcom MemVar
fcomp MemVar
fcom src
fcomp src
fcom
fcomp
fcompp
ficom MemVar
ficomp MemVar
```

All of these instructions set condition flags C_3, C_2 and C_0 according to the comparison.

The `fcom` and `fcomp` instructions compare their source argument with a number stored in `st(0)`. The source operand can be one of the following.

- A single precision floating point number stored in memory. In this case, a suffix 's' is required in the instruction mnemonic (*i.e.* the instructions are written as `fcoms` or `fcomps`).
- A double precision floating point number stored in memory. In this case, a suffix 'l' is added to the instruction mnemonic and the instructions are written as `fcoml` or `fcompl`.

- A register in the register stack. For example, `fcom %st(3)` instruction compares stack top (`st(0)`) with register `st(3)`.

- Implied as `st(1)`. In this case the instruction does not require any further argument.

The `fcomp` instruction is similar to the `fcom` instruction except that it removes one operand from the register stack as well. While comparing $+0$ and -0 are treated equally. Thus a comparison of $+0$ with -0 sets flags as if two numbers are equal.

`fcompp` instruction takes no arguments and compares `st(0)` with `st(1)`. It removes both operands from the register stack. Thus the stack depth gets reduced by 2 after the execution of this instruction.

The `ficom` and `ficomp` instructions compare a floating point number in `st(0)` with a 16-bit or a 32-bit integer stored in memory. The instruction mnemonic is suffixed by '`s`' and '`l`' depending upon whether the `MemVar` argument provides a 16-bit value or a 32-bit value. The `ficomp` instruction removes one floating point number from the register stack. The `ficom` instruction does not remove any operands from the register stack. Thus after the execution of `ficomp` instruction, depth of register stack is reduced by 1 while it does not change after the execution of `ficom` instruction.

Various flags are set according to the following logic.

`st(0)` = NaN or the source = NaN: C_3, C_2, C_0 = 111.

`st(0)` $>$ source: C_3, C_2, C_0 = 000.

`st(0)` $=$ source: C_3, C_2, C_0 = 100.

`st(0)` $<$ source: C_3, C_2, C_0 = 001.

These condition flags can be taken into `ax` register using `fstsw` instruction that is described later. Register `ah` can then be stored in `eflags` register using `sahf` instruction. This way C_0 flag gets copied to CF, C_2 gets copied to PF and C_3 gets copied to ZF flag. Thus the ZF flag gets set when two operands are equal while the CF gets set when `st(0)` is smaller than the source.

There are two kinds of comparison operations in x87 FPU – Unordered comparison and normal comparison. The unordered relationship is true when at least one operand of the comparison instruction is a NaN. By definition, NaN is not a number and therefore can not be less than, equal or greater than any other number. The normal comparison instructions raise an IE floating point exception for this case. Under GNU/Linux all interrupts are masked and therefore, only the IE flag is set in the status register. Unordered comparison instructions do not raise IE exception when any operand of the instruction is a NaN. In case of unordered relation being true, condition flags C_3, C_2 and C_0 are all set to 1.

The following are unordered comparison instruction.

```
fucom src
fucomp src
fucom
fucomp
fucompp
```

In terms of the functionalities under GNU/Linux, there is very little difference between these instructions and the corresponding normal comparison instructions.

In addition to these instructions, there is one instruction that compares the value on the top of the register stack with 0.0 and sets condition flags C_3, C_2 and C_0 according to the comparison.

```
ftst
```

Thus after the execution of `ftst` instruction, flags are set as follows.

C_3, C_2, C_0 = 111 if `st(0)` is NaN.

C_3, C_2, C_0 = 000 if `st(0)` $>$ 0.0.

C_3, C_2, C_0 = 100 if `st(0)` $=$ 0.0.

C_3, C_2, C_0 = 001 if `st(0)` $<$ 0.0.

The third broad category of comparison instructions are thos that set the flags in `eflags` registers directly. The comparison instruction discussed till now set the condition flags in x87 status register. In order to use conditional jump instructions, these flags should be first taken to the a regular CPU register and tested. In the modern IA-32 processors, there are versions of the comparison instructions that can set the CPU flags in `eflags` register directly. These instructions avoid instructions to move data from x87 status register to one of the CPU registers and then copying it to the `eflags` register.

```
fcomi src, %st
fucomi src, %st
fcomip src, %st
fucomip src, %st
```

The `fcomi` and `fucomi` instructions compare a register on register stack with `%st(0)`. One operand of these instruction is always `st(0)`. Correspoding `fcomip` and `fucomip` instructions compare and remove one value from the register stack. The instructions modify ZF, PF and CF flags in the `eflags` register as follows.

`st(0)` $>$ src: ZF, PF, CF = 000.

`st(0)` $=$ src: ZF, PF, CF = 100.

`st(0)` $<$ src: ZF, PF, CF = 001.

`st(0)` $=$ NaN, or `src` $=$ NaN: ZF, PF, CF = 111.

Classification of a number

A floating point number on top of the register stack may contain several kinds of values such as NaN, Normalized number, denomalized number, zero, infinity etc. In x87 FPU there is one instruction that can be used to quickly classify the number type so that the software may take an appropriate action. For example, a printing software may print `nan` for a NaN or $\pm$`inf` for $\pm\infty$ by looking at the classification results.

```
fxam
```

The `fxam` instruction examines the value stored on top of the register stack (`st(0)`). It does not raise any floating point exception including that of the stack fault. When the stack is empty appropriate value is set in the condition flags as per the following table.

Class	C_3	C_2	C_1	C_0
Positive NaN	0	0	0	1
Negative NaN	0	0	1	1
Positive Normal number	0	1	0	0
Negative Normal number	0	1	1	0
$+\infty$	0	1	0	1
$-\infty$	0	1	1	1
$+0$	1	0	0	0
-0	1	0	1	0
Empty register stack	1	0	U	0
Positive Denormal number	1	1	0	0
Negative Denormal number	1	1	1	0
Unsupported format	0	0	U	0

U: Undefined value for C_1 flag.

9.6.6 Data transfer instructions

In x87 FPU, data in x87 data registers is always in double extended-precision format. All instructions that operate on registers perform computations using double extended-precision format. Even instructions that take one operand in memory internally convert their memory operand to double extended-precision format before performing the computations.

Data transfer instructions load data from memory to a register, store a register to memory or move data between x87 data registers. In the first two cases, data is converted to, or from, double extended-precision format to the format of memory operand. While loading, data transfer instructions can handle floating point numbers in single precision, double precision or in double extended-precision; integers in 16-bit, 32-bit or 64-bit formats; and 18-digit binary coded decimal numbers

in 80-bit format. While storing data from double extended-precision number format can be converted to one of these formats depending upon the instruction.

The following instructions are used for floating point data transfer.

```
fld MemVar
fld src
fst MemVar
fst src
fstp MemVar
fstp src
```

`fld` instruction is used to load data from memory or one of the registers to the register stack top. The memory variable can be 32-bit single precision number, 64-bit double precision number or 80-bit double extended-precision number (also known as temporary real) for which the suffices 's', 'l' and 't' are used to the `fld` instruction mnemonic. This instruction can also be used to load a variable already stored in one of the stack register to the stack top. The `fld` instruction increases the register stack depth by 1.

`fst` instruction is used to store data from top of the register stack to memory or to another register. If the argument of this instruction is in the memory, a suffix of 's' or 'l' must be used corresponding to single precision or double precision numbers respectively. The instruction can not be used to store number in double extended-precision format.

`fstp` instruction is similar to the `fst` instruction. In addition to the work done by an `fst` instruction, this instruction removes value on top of the stack, reducing stack depth by 1. In addition, `fstp` instruction can be used to store double extended-precision number with a 't' suffix to the instruction.

The following are some examples of these instructions.

`flds sp_f`	Load a single precision floating point number from memory location `sp_f` to top of the register stack.
`fldl dp_f`	Load double precision floating point number from `dp_f` to top of the register stack.
`fldt edp_f`	Load double-extended precision floating point number from memory to top of the register stack.
`fld %st(3)`	Load value in `st(3)` to top of the register stack. Stack is adjusted and therefore, the old value in `st(3)` becomes `st(4)`.
`fsts sp_f`	Store `st(0)` in memory after conversion to single precision floating point format.

`fstl dp_f`	Store `st(0)` in memory as double precision floating point number.
`fst %st(3)`	Copy `st(0)` to `st(3)`.
`fstps sp_f`	Store and remove `st(0)` to memory as single precision floating point number. Depth of register stack reduces by 1.
`fstpl dp_f`	Store and remove `st(1)` to memory as double precision floating point number. Depth of register stack reduces by 1.
`fstpt edp_f`	Store and remove stack top to memory as temporary real or double extended-precision floating point format. Depth of register stack reduces by 1.
`fstp %st(1)`	Copy `st(0)` to `st(1)` and remove. Old `st(1)` becomes `st(0)`. Thus this operation is equivalent to removing `st(1)`.
`fstp %st(4)`	Copy `st(0)` to `st(4)`. Remove `st(0)`. `st(4)` gets known as `st(3)` after the execution.

In addition to the floating point load and store instructions, regular integers can also be loaded in x87 FPU register stack or saved from the register stack to memory. While transferring data from memory to register stack, integers are converted to double extended-precision format. Similarly they are converted from double extended-precision format to integer format while saving in memory. The supported integer format are 16-bit, 32-bit or 64-bit in size. Following instructions provide the mechanism to transfer integer data to and from x87 FPU register stack top.

```
fild MemVar
fist MemVar
fistp MemVar
```

Instructions need to be suffixed by 's', 'l' or 'q' to represent 16-bit, 32-bit or 64-bit integer. While `fild` and `fistp` instructions can have any size of the memory operand, `fist` instruction does not support 64-bit operands.

`fild` instruction causes the register stack depth to be increased by 1. `fistp` instruction stores and removes the register stack top causing the register stack depth to be reduced by 1. `fist` instruction does not modify the register stack depth.

`fist` and `fistp` instructions convert the real number in double extended-precision format to integer format and uses rounding control information for that purpose. `fld` instruction converts the integer format to double extended-precision format.

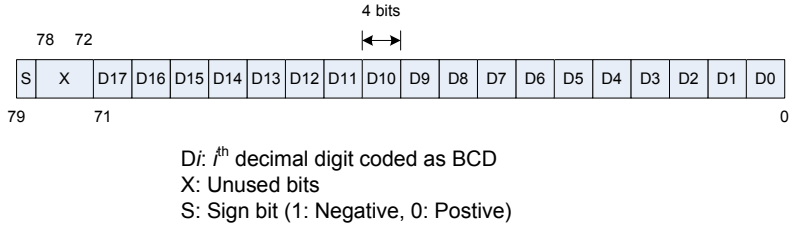


Figure 9.8: BCD numbers in x87 FPU

Integers are also supported in BCD format as shown in figure 9.8. There are two instructions to load and store integers in BCD format.

```
fbld MemVar
fbstp MemVar
```

The first instruction reads an 80-bit (10-byte) BCD value, which is assumed to be in the format as shown in figure 9.8, converts that value to double extended-precision floating point number and pushes it on the register stack. Thus depth of the register stack increases by 1. The second instruction converts the double extended-precision floating point number in `st(0)` to a BCD number in the format given in figure 9.8 and stores it in the given memory location. If the number in stack top is not an exact integer, it is rounded off as per the rounding control in x87 control register. A number such as $\pm\infty$, NaN or one that is too large to fit in 18-digit BCD number, is converted to a special value called BCD indefinite. In BCD indefinite, two most significant bytes have a value 0xFF while values of other 8 bytes are undefined.

Exchanging values in registers

Value of stack top register can be exchanged with value of any other register on register stack using `fxch` instruction as given below.

```
fxch src
fxch
```

One operand for the exchange is always the top of the register stack. The other operand can only be a register such as `st(i)` as in the first form, or can be implied as `st(1)` in the second form.

`fxch` instruction is often useful in temporarily exchanging the values of two registers, computing a function that need implied argument on the stack top (such as square root) and then exchanging it again to put it back to the original register.

Conditional copying of data between registers

The x87 FPU instruction set supports conditional copying of data from top of the register stack to another a register based on the value of condition flags in `eflags` registers. These instructions can be used to replace a conditional jump based program constructions to set values in registers. As discussed earlier x87 FPU data comparison instructions are used to compare floating point numbers. Many of these instructions set condition flags in x87 FPU status register. These condition codes can then be transfered to the `eflags` register by using appropriate instructions as discussed earlier. Some other comparison instructions set flags directly in the `eflags` register.

Conditional data copying instructions use flags only in the `eflags` register and hence it would be essential to copy floating point conditional flags to the `eflgas` registers. However data movement based on the integer instructions of IA32 processors which set flags in `eflags` registers can be done directly. The following are the instructions that move data from one register to the stack top based on the conditions in `eflags` register.

```
fcmovcc src, %st
```

The following conditions can be specified.

Instruction	Condition for move	Flags
`fcmove`	Equal	ZF=1
`fcmovne`	Not Equal	ZF=0
`fcmovnbe`	Neither Below nor Equal	CF=0 and ZF=0
`fcmovbe`	Below or Equal	CF=1 or ZF=1
`fcmovnb`	Not Below	CF=0
`fcmovb`	Below	CF=1
`fcmovu`	Unordered	PF=1
`fcmovnu`	Not unordered	PF=0

As an example consider instruction `fcmovu %st(5), %st`. This instruction will result in copying the value stored in register `%st(5)` to the stack top without changing the depth of the register stack. The copy will occur only when the result of a previous comparison instruction modified PF flag (in `eflags` register) to 1. This insturction can be used to check if a NaN was the result of any previous operation and in that case, set a particular value and avoid NaN in future computations.

Let us consider the following logic to be implemented in an Assembly language program.

x = y / z;
if (x == **NaN**) x = +**inf**;

The corresponding code in Assembly language is given below.

```
example:
    flds z       // Load z and y on stack
    flds y
    fdivp        // Stack = (y/z)
    fldz         // Compute infinity as 1/0
    fld1
    fdiv %st(1), %st // Stack = (inf, 0.0, y/z)
    fxch         // Stack = (0.0, inf, y/z)
    // Compare 0.0 and y/z
    fcomi %st(2) // Set flags in eflags
    fxch %st(2)  // Stack = (y/z, inf, 0.0)
    // If unordered, copy infinity to st(0)
    fcmovu %st(1), %st
    fstp %st(1)  // Remove two items from
    fstp %st(1)  // register stack
    // Stack = (x)
    ret
```

9.6.7 FPU control instructions

The FPU control instructions are used to operate on the status, control and tag registers. Many of these instructions come in two different variations. One variation with '`fn`' prefix is used to perform the operation without checking for any pending error conditions from previous operations. Thus floating point exceptions generated during the execution of previous instructions might not have been handled. The other variation with '`f`' prefix only is used to preform the operation after checking for the error conditions from any previous floating point operation. These two variations only differ in the way floating point exceptions are handled. Under GNU/Linux, floating point exceptions are masked and therefore both instructions behave in a similar manner. However instructions with '`fn`' prefix are smaller instructions and should be preferred over instructions with '`f`' prefix only.

The state of x87 FPU is described by the values of the status, control and tag registers. It can be initialized using one of the following two instructions.

```
finit
fninit
```

These instructions initialize the FPU state to the following.

The FPU control register is initialized to 0x37F, which masks all floating point exceptions, set rounding control to round-to-nearest and precision control to double extended-precision.

The status word is set to 0 meaning thereby that no exceptions are set, all condition flags are set to 0 and the register stack top is initialized to 0.

The tag word is set to 0xFFFF which makes all data registers marked as empty.

This is also the condition after a reset to the processor. As MMX operating environment interfere with x87 FPU registers, `fninit` instruction must be executed when MMX computing environment is changed to x87 FPU? computing environment. Many program examples in this chapter include the use of `fninit` instruction.

All exception flags can be cleared using the following instructions.

```
fclex
fnclex
```

This instruction essentially sets all exception flags in the status register to 0. These flags include PE, UE, OE, ZE, DE, IE, ES, SF and B bits in the status register as shown in figure 9.5.

The register stack pointer in x87 status register can be manipulated using the following two instructions.

```
fincstp
fdecstp
```

The `fincstp` instruction increments the stack pointer while the `fdecstp` instruction decrements the stack pointer. No other state is modified by these instructions.

Individual data registers of x87 FPU can be marked free using the following instruction.

```
ffree src
```

The register indicated by the operand of the `ffree` instruction is marked free. Two bits of corresponding tag in the tag register are set to 11 after the execution of this instruction.

Information in the x87 FPU status and control registers can be stored in memory and register or loaded from memory with the following instructions.

```
fstcw MemVar
fnstcw MemVar
fldcw MemVar
fstsw MemVar
fstsw %ax
fnstsw MemVar
fnstsw %ax
```

Control word can be stored in a two-byte wide memory location using `fstcw` or `fnstcw` instructions. Similarly a value stored in two-byte wide memory location can be loaded in the x87 Control register upon execution of `fldcw` instruction. Status word can be stored in memory or in `ax` register using `fstsw` or `fnstsw` instructions. As discussed earlier, two instructions `fstsw` and `fnstsw` work identically in GNU/Linux system. Similarly instructions `fstcw` and `fnstsw` instructions work in an identical manner.

In addition to the instructions described above, there are these following instructions that save or restore entire state of the x87 FPU in memory. Details of these instructions are out of scope of this book.

```
fstenv MemVar
fnstenv MemVar
fldenv MemVar
fsave MemVar
fnsave MemVar
frstor MemVar
fxsave MemVar
fxrstor MemVar
```

In addition to the above mentioned instructions there are the following instructions supported by the x87 FPU.

```
fwait
wait
fnop
```

`fwait` or `wait` instruction is used to wait for x87 FPU to finish its execution and is not typically used in programs. `fnop` instruction is a "No Operation" instruction and it does not do any processing on any of the FPU registers.

9.7 Parameter passing

The parameters between various functions can be passed in a number of ways two of which are the most common.

1. Through FPU register stack.
2. Through stack in memory

The most effective manner in which parameters are passed between Assembly language functions is through register stack. However the size of the register stack is limited and therefore large number of parameters can not be passed through stack. Further deep recursions of

function calls where parameters are passed through stack is also not possible due to limited size of the register stack.

Most x87 FPU instructions take their arguments through the register stack, it is natural to load parameters on register stack using `fld` instruction and then call the functions.

As mentioned in chapter 4, code generated by C compiler passes parameters using stack in memory. Single precision floating point numbers are pushed as 32-bit numbers on the stack while double precision floating point numbers are pushed as 64-bit numbers on the stack. As mentioned in chapter 4, the stack at the entry of a function has certain layout. Here the return address is at top of the stack in memory. Frame pointer with `ebp` register can be used to access local variables as discussed in earlier chapters.

In GNU C runtime environment, the floating point return values of a function are returned on the register stack in `st(0)`.

Here is an example of a C-callable assembly language function that computes the remainder of two single precision floating point numbers passed as arguments. While computing the remainder, the program uses rounding control as rounding-to-nearest.

```
// C Callable function to compute remainder.
// C Prototype is
// float Rem_float(float a, float b);
.globl Rem_float
Rem_float:
    fninit     // Initialize the x87 FPU
    flds 8(%esp) // Load b on register stack
    flds 4(%esp) // Load a on register stack
Rem1:
    fprem1      // Partial remainder in IEEE754 style
    fstsw %ax  // Save flags in register ax
    test $0x0400,%ax  // Look for C2 flag
    jnz    Rem1 // Execute again if reduction incomplete
    fstp %st(1) // Adjust the stack to remove one item.
    ret
```

The prototype of the Assembly language function `Rem_float` in C can be written as the following.

float Rem_float(**float** a, **float** b);

A similar C-callable assembly language function that computes the remainder of division when two double precision floating point numbers are passed as arguments is also given below.

```
.globl Rem_double
Rem_double:
```

```
    fninit     // Initialize the x87 FPU
    fldl 12(%esp) // First argument (double precision)
    fldl 4(%esp) // Second argument (double precision)
Rem1:
    fprem1
    fstsw %ax
    test $0x0400,%ax
    jnz    Rem1
    fstp %st(1) // Result on top of the register stack.
    ret
```

The prototype of this function in C would be the following.

double Rem_double(**double** a, **double** b);

The codes for `Rem_float` and `Rem_double` functions are similar to each other. In fact, they are identical except for the loading of parameters on register stack. This is possible in most such functions in IA32 architectures because x87 FPU operates on data registers assuming them to be in double extended-precision format. The only difference comes when data is loaded from memory to when data is stored in memory.

EXERCISES:

9.14 Write a C callable assembly language function to compute larger of the two roots of a quadratic equation $ax^2+bx+c=0$ where a, b and c are single precision floating point arguments passed by the C function. The return value of the function is also a single precision floating point number.

9.15 Write an Assembly function to compute 10^y ($|y| \leq 1$). You may use the following identity to compute this.

$$x^y = 2^{y \log_2 x}$$

Use `f2xm1` instruction to compute. (Hint: You may recall that `f2xm1` instruction computes $2^x - 1$ where x is less than 1. Since $\log_2 10 \approx 3.32$, $y \log_2 10$ can have a maximum value of about 3.32. Therefore compute x^y as $\left(2^{\frac{1}{4} y \log_2 x}\right)^4$. a^4 can be computed by squaring a twice.)

Test your code by calling your function from C for various values of y.

9.16 Write a bisection method to find the root of the following equation.

$$f(x) = x^3 - \sqrt{x} + x^2 = 0$$

We know a trivial root of this equation at $x = 0$. The value of $f(x)$ is less than 0.0 for $x = 0.25$ and more than 0.0 for $x = 1$. Hence there must be at least one root between 1.0 and 0.25. The bisection method will start with the search range being 0.25 to 1.0 and will keep dividing the range into half such that the function $f(x)$ has different signs at two extremes

of the range. The root will be found when $\left|\frac{\Delta x}{x}\right| < \epsilon$. Here Δx is the size of the range and x is the left most extreme of the range. Take $\epsilon = 0.001$).

Chapter 10

SIMD instruction sets

The historical growth of IA32 architectures has led to several enhancement of the instruction set. The Intel introduced Multimedia Extension (MMX) instruction set with Pentium processors in 1993. This instruction set was designed to have Single-Instruction Multiple-Data (SIMD) instructions. Most instructions in this instruction set operate simultaneously on multiple data values. MMX instruction set was primarily introduced to provide fast execution mechanisms for applications such as image processing, multimedia applications including voice, data communication etc. Later the SIMD instruction sets were enhanced to include more instructions and registers within the IA32 architectures. Starting with Pentium-III, Intel introduced Streaming SIMD Extension (SSE) instruction set in 1999. While MMX instructions operate on integer data, the SSE instructions provide similar capabilities for floating point data. SSE instruction set is targeted at applications that operate on large arrays of floating point numbers such as 3-D graphics, video encoding and decoding etc. A second enhancement to the SSE instruction set, called SSE2 instruction set, was introduced in 2000 with Pentium 4 processors. SSE2 enhancements to the instruction set provided 128-bit wide registers and operations on multiple integers and floating point numbers that could be packed in 128-bit registers. The intended applications for the SSE2 instruction set are the same as for the SSE instruction set but they should run faster with SSE2 instructions. AMD on the other hand provided 3DNow!^{TM} extensions to instruction set for handling 3-D graphics and similar applications.

In this chapter we shall see MMX, SSE and SSE2 instruction sets and architectural enhancements to support these instructions. Most modern processors from Intel support all the three instruction sets (and may be more). Intel also provides a capability in the modern processors to inquire whether or not a specific instruction set is supported by the processor. For this an instruction called `cpuid` is provided in the instruction sets of various processors. Description and application of

this has not been included in this book.

10.1 SIMD environment

SIMD instruction sets (*i.e.* MMX, SSE and SSE2) provide few additional registers called MMX registers and XMM registers. MMX registers include eight 64-bit registers named `mm0` to `mm7`. These registers are aliased to x87 FPU data registers `R0` to `R7`. During the execution of x87 FPU instructions, these registers are referred to as register-stack using `st(0)` to `st(7)` notations. However, in SIMD instructions, these registers can only be addressed directly. Data registers in x87 FPU environment are 80-bit wide and are capable of storing double extended-precision floating point numbers. Out of these 80-bits, only 64-bits (just the part of the register used to store the significand) are used by instructions the SIMD instruction sets.

Use of the same registers for MMX and X87 FPU makes it tricky to program simultaneously using both instruction sets. It is recommended to use `fninit` or `finit` instruction to initialize the x87 FPU environment at the time of switching from SIMD to x87 environment. There are a few more methods to initialize the state between MMX and x87 instruction sets and are discussed later in this chapter.

In addition to the MMX registers, SIMD instruction sets also provide eight 128-bit wide registers called XMM registers. These registers are named as `xmm0` to `xmm7`. XMM registers can store four single precision floating point numbers, or two double precision floating point numbers, or multiple integers in byte, word (16-bit), long (32-bit), quad word (64-bit) or in a single 128-bit integer formats. XMM registers are independent registers and are not shared with the x87 FPU data registers unlike MMX registers.

The registers and supported data types in SIMD environment are shown in figure 10.1. As shown in this figure, the MMX registers are 64-bit wide (figure 10.1(a)) and support 64-bit quad word, two packed 32-bit long words, four packed 16-bit words and 8 packed bytes (figure 10.1(c)). The XMM registers are 128-bit wide (figure 10.1(b)) and support integer and floating point data types. Supported integer data types (figure 10.1(d)) include 128-bit double quad word, two packed 64-bit quad words, four packed 32-bit long words, eight packed 16-bit words and sixteen packed bytes. Supported floating point data types (figure 10.1(e)) include two packed double precision or four packed single precision floating point numbers.

SIMD instructions essentially perform the same operation on multiple data items as per the data types shown in figure 10.1. For example, an add operation that operates on packed 16-bit words will take two 4-packed word integers, add them as shown in figure 10.2 and provide the result as 4-packed word integers.

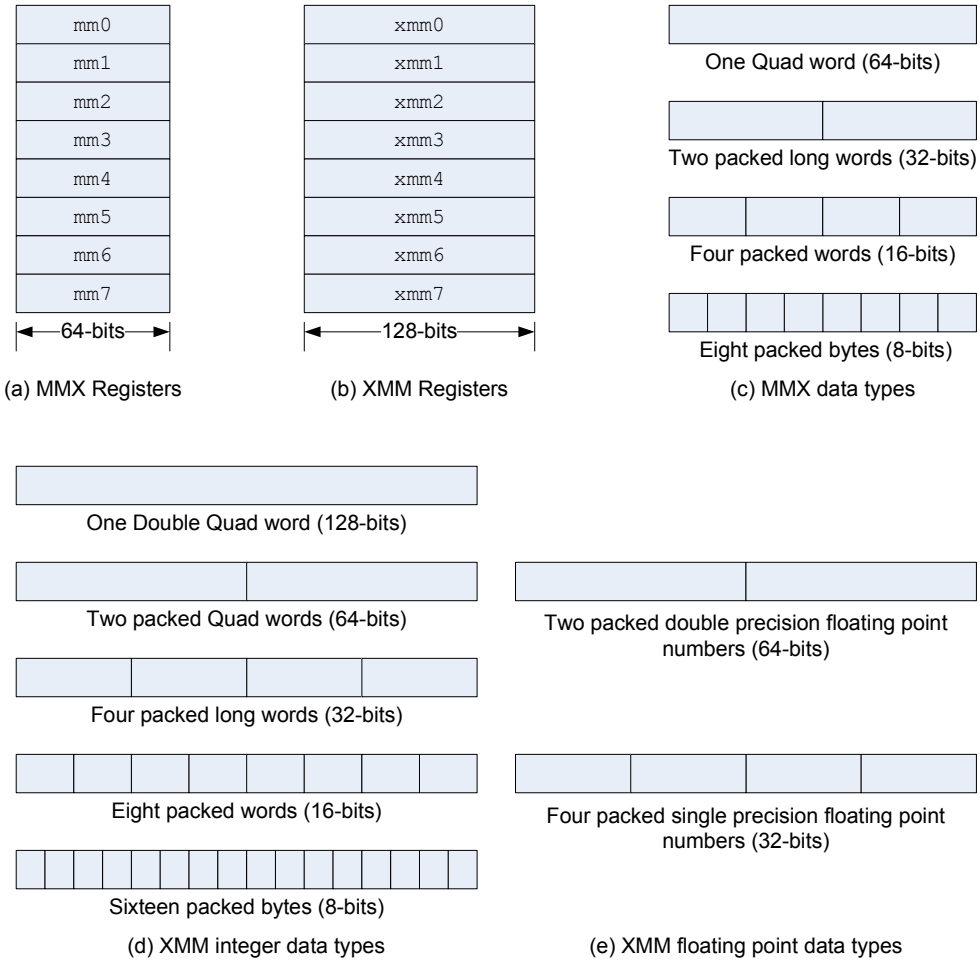


Figure 10.1: MMX registers and supported data types

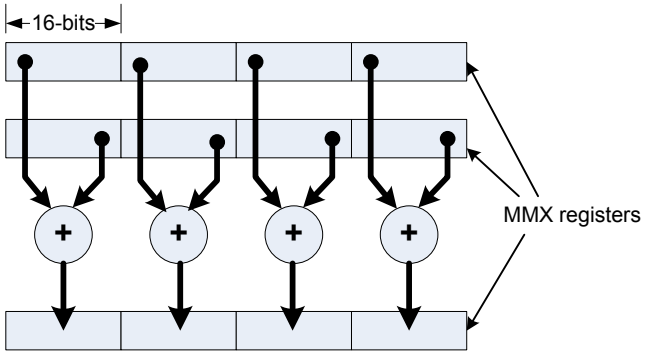


Figure 10.2: SIMD add operation on 4-packed words

In addition to the data registers `mm0` to `mm7` and `xmm0` to `xmm7`, SIMD instructions also provide a single register for 32-bit floating point status and control operations with a functionality similar to that of x87 FPU control and status registers. This register is called MXCSR register and is independent of the x87 status and control registers. Layout of this register is shown in figure 10.3. MXCSR register is not affected by the execution of `finit` or `fninit` instructions. However the behavior of various flags and control bits are similar to that of the flags and controls in x87 FPU registers.

Contents of the MXCSR register can be modified through `ldmxcsr` and `fxrstor` instructions. These can be saved using `stmxcsr` and `fxsave` instruction. The function of the most bits of the MXCSR register are similar to that of the corresponding bits in x87 status and control registers. There was two new control bits that need some explanation. The floating point computations can be made to execute faster if a special case of handling denormalized numbers is done away with. Recall that the denormalized numbers in IEEE754 representations are used to store extremely low values to minimize truncation errors. In most graphics and multimedia applications, such cases will not normally arise. Even when these cases arise, the small numbers may almost always be treated as zeros. In the MXCSR register, `FZ` bit is used to control flush-to-zero behavior. In GNU/Linux, if this bit is set to 1 the result returned by most SIMD instructions is 0 when the output of the instruction should be a very small denormalized floating point number. The other control bit `DAZ` is used to control treating denormals-as-zeros behavior. When this bit is set, floating point input operands to a SIMD instruction are treated as zeros if they are in denormalized form.

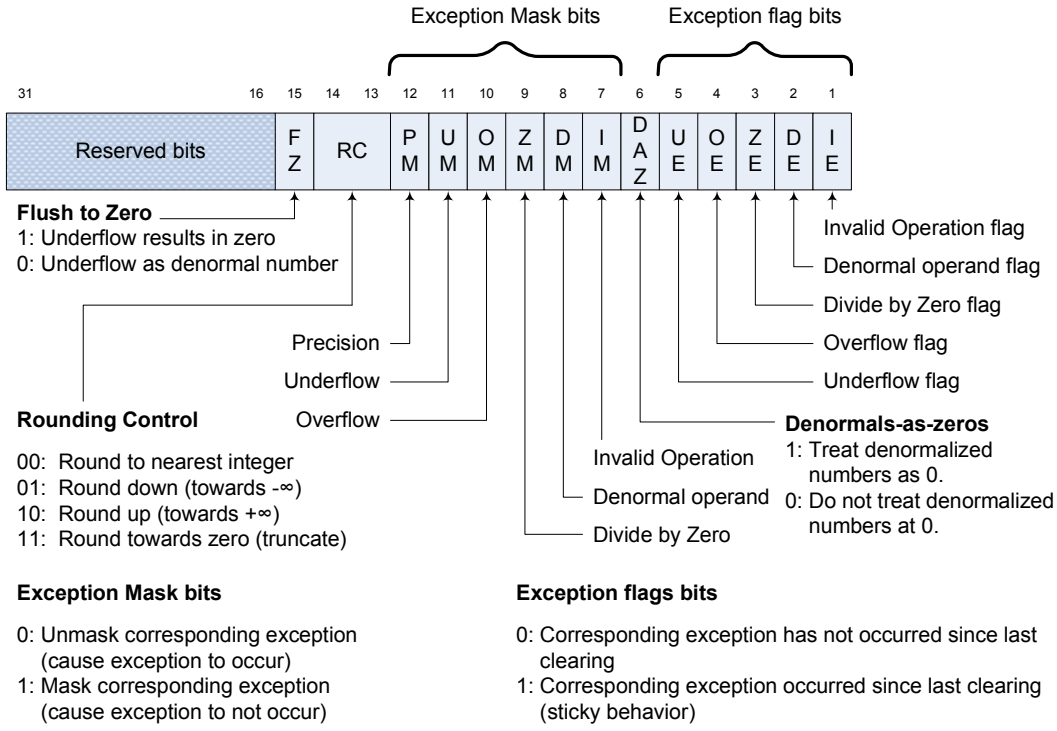


Figure 10.3: MXCSR register

10.2 SIMD instructions

SIMD instructions can broadly be categorized as follows.

1. Arithmetic, logic, shift and rotate instructions
2. Data transfer and conversion instructions
3. Comparison instructions
4. Floating point instructions

10.2.1 Arithmetic, logic, shift and rotate instructions

SIMD instruction sets operate on multiple data. However, various flags in the architecture such as CF, ZF etc. can only represent conditions on one result. Hence in SIMD instruction sets, these flags bits are not used. Instead SIMD instruction sets provide various kinds of arithmetic operations. These operations handle the conditions of result overflow or underflow in a way that the result is still meaningful. The following kinds of operations are available in IA32 architectures.

1. Normal operation (with wrap-around)
2. Operations with signed saturation
3. Operations with unsigned saturation

The operations are first illustrated with an example. Let's assume that we need to add two byte-wide integers, 0xA3 and 0xC2. By adding these two numbers we get 0x165, a number that does not fit in a single byte. The normal addition process would ignore the most significant bit and provide a byte-wide value 0x65. This scheme is also known as wrap-around scheme because whenever the result is larger than 0xFF, the extra bit is dropped. Thus the result is treated as module 256. If we assume that these byte values are used to represent signed numbers, then the values for these two bytes are -93 and -62 in decimal. Summation of these two numbers will be -155 in decimal, which is below the minimum possible using signed number representation. Hence this number can not be represented in 8 bits. Similarly, if we assume that these two byte values are used to represent unsigned numbers, then the values represented by these bytes are 163 and 194 in decimal. The true addition of these numbers now results in 357, which again can not be represented in 8-bits unsigned number format. The general purpose instructions provide overflow flag (to represent the signed number overflow) and carry flag (to represent the unsigned number overflow) to indicate that the result can not be represented in desired number of bits. However in SIMD instructions flag bits can not be

Table 10.1: Examples of SIMD byte operations with saturations

A	B	Op	Result with		
			wrap-around	Signed saturation	Unsigned saturation
0xA3	0xC2	A+B	0x65	0x80	0xFF
0x57	0xB2	A+B	0x09	0x09	0xFF
0x78	0x40	A+B	0xB8	0x7F	0xB8
0x40	0x72	A−B	0xCE	0xCE	0x00
0x44	0x23	A−B	0x21	0x21	0x21

used as there are multiple results generated by a single instruction. The normal wrap-around operation results in 0x65 (and no flags will be modified). Signed saturation and unsigned saturation operations provide result as 0x80 (*i.e.* −128) and 0xFF (*i.e.* 255) which are the minimum negative and maximum positive values in the corresponding representations.

In general the saturation modes of operations ensure that the results are not outside the range of minimum and maximum possible values. For example, while using signed saturation when the true result of an operation is smaller than the minimum negative number, the result is set to the minimum negative number. Similarly the result is set to the maximum positive number when the true result is more than the maximum positive number. While using unsigned saturation, the results are set to 0 if the true result is negative and set to maximum positive value if the true result is larger than that.

Some examples of SIMD arithmetic operations using byte wide data structures and various modes of saturation are given in table 10.1.

SIMD integer arithmetic instructions

IA32 architectures provide various SIMD instructions for arithmetic operations on integers. Some of these operations were introduced with MMX instruction set while some others were introduced with SSE2 instruction set. In particular, operations that use MMX registers were introduced with MMX instruction set. Most of these operations were enhanced to use XMM registers that became available with SSE2 instruction set.

Instruction	Operation	Inst sets
Addition instructions		
`paddb src, dest`	Add packed bytes	MMX, SSE2
`paddw src, dest`	Add packed words	MMX, SSE2

Instruction	Operation	Inst sets
`paddd src, dest`	Add packed longs (or doublewords)	MMX, SSE2
`paddq src, dest`	Add packed quad words	SSE2
`paddsb src, dest`	Add packed bytes with signed saturation	MMX, SSE2
`paddsw src, dest`	Add packed words with signed saturation	MMX, SSE2
`paddusb src, dest`	Add packed bytes with unsigned saturation	MMX, SSE2
`paddusw src, dest`	Add packed words with unsigned saturation	MMX, SSE2
Subtraction instructions		
`psubb src, dest`	Subtract packed bytes	MMX, SSE2
`psubw src, dest`	Subtract packed words	MMX, SSE2
`psubd src, dest`	Subtract packed longs (or doublewords)	MMX, SSE2
`psubq src, dest`	Subtract packed quad words	SSE2
`psubsb src, dest`	Subtract packed bytes with signed saturation	MMX, SSE2
`psubsw src, dest`	Subtract packed words with signed saturation	MMX, SSE2
`psubusb src, dest`	Subtract packed bytes with unsigned saturation	MMX, SSE2
`psubusw src, dest`	Subtract packed words with unsigned saturation	MMX, SSE2
Multiplication instructions		
`pmulhw src, dest`	Multiply packed signed words and store high result	MMX, SSE2
`pmullw src, dest`	Multiply packed signed words and store low result	MMX, SSE2
`pmulhuw src, dest`	Multiply packed unsigned words and store high result	SSE, SSE2
`pmuludq src, dest`	Multiply packed unsigned longs	SSE2

All of these instructions can operate on 64-bit packed data stored in MMX register or on 128-bit packed data stored in XMM registers. The `src` operand of the instruction can also be a memory operand. During execution, 64-bit data (8 bytes) is read in case the other operand is a 64-bit MMX register and 128-bit data (16-bytes) is read in case the other operand is a 128-bit XMM register. The `dest` operand of the instruction can only be an MMX register or XMM register. Both operands of the instructions must be of the same size. For example, it

is not possible to mix and add data in an XMM register with data in an MMX register. In case of an addition, multiple packed items in the `src` operand are added to the respective packed items in `dest` register. In case of a subtraction operation, values read from the `src` operand are subtracted from the values in `dest` operand.

Multiplication operation requires a bit of explanation. Multiplying two integers of size n bits produces a result of size $2n$ bits. Thus after the multiplication, destination may contain the high order n bits as in `pmulhw` and `pmulhuw` instructions or low order n bits as in `pmullw` instruction. The `pmuludq` instruction takes 2 long words if an MMX register is used or 4 long words if an XMM register is used. However it multiplies only every alternate numbers of its operand and produces 2 quadwords in case of an MMX register or 4 quadwords in case of an XMM register. Figure 10.4 shows the behavior of the SIMD multiplication instructions.

Some examples of these instructions are given below.

`paddb abc, %mm3`: Read 8 bytes from memory location `abc` (lower order byte first) and add it (with wrap-around) to the contents of register `mm3` using addition of 8 packed bytes.

`paddsw %xmm1, %xmm2`: `xmm2 = xmm2 + xmm1` using addition with signed saturation of 8 packed words (16-bits).

`psubq %xmm2, %xmm5`: `xmm5 = xmm5` $-$ `xmm2` using subtraction of two packed quad words (64-bits) with wrap-around.

`pmulhw 8(%ebx,%ebp), %xmm4`: Read 16 bytes from the specified memory address and multiply with `xmm4` using signed integer multiplication. While executing this instruction, it is assumed that the operends contain 8 packed words (16-bits). Higher order 16-bits of each of the multiplication results are stored back in `xmm4`.

`pmuludq %mm3, %mm8`: `mm8 = mm8` $*$ `mm3`. Lower order 32-bits of `mm8` and `mm3` are multiplied. Resulting 64-bit number is stored back in `mm8`.

A few more arithmetic instructions provided by the SIMD instruction sets are the following.

Instruction	Operation	Instruction sets
`pmaddwd src, dest`	Multiply and add packed words (16-bits)	MMX, SSE2
`pavgb src, dest`	Compute average of packed bytes	SSE, SSE2
`pavgw src, dest`	Compute average of packed words	SSE, SSE2

Instruction	Operation	Instruction sets
`pmaxub src, dest`	Maximum of packed unsigned bytes	SSE, SSE2
`pmaxsw src, dest`	Maximum of packed signed words (16-bits)	SSE, SSE2
`pminub src, dest`	Minimum of packed unsigned bytes	SSE, SSE2
`pminsw src, dest`	Minimum of packed signed words (16-bits)	SSE, SSE2
`psadbw src, dest`	Compute $\sum_i \lvert \texttt{src}_i - \texttt{dest}_i \rvert$ using packed unsigned bytes. 16-bit result gets in `dest`.	SSE, SSE2

Each of these instructions can take 64-bit MMX arguments in MMX or SSE instruction sets, or 128-bit XMM arguments in SSE2 instruction set. Similar to the other SIMD instructions, `src` operand of the instruction can be in memory or in MMX/XMM register. The `dest` operand of the instruction can only be in MMX/XMM register.

The `pmaddwd` instruction in MMX version takes two 64-bit operands, each containing four packed 16-bit signed numbers. If these numbers in `src` operand are named as x_0, x_1, x_2, x_3 and those in `dest` operand are named as y_0, y_1, y_2, y_3 then the instruction computes two 32-bit signed numbers as $x_3y_3 + x_2y_2$ and $x_1y_1 + x_0y_0$ and stores in the `dest` operand as two packed signed long (32-bit) integers as shown in figure 10.5(a).

In SSE2 version the `pmaddwd` instruction uses XMM registers. If the input numbers are $x_0 \ldots x_7$ and $y_0 \ldots y_7$ in `src` and `dest` operands, then after the execution of the instruction, `dest` operand contains four packed signed long integers with values being $x_7y_7 + x_6y_6$, $x_5y_5 + x_4y_4$, $x_3y_3 + x_2y_2$ and $x_1y_1 + x_0y_0$ as shown in figure 10.5(b).

Instructions `pavgb` and `pavgw` compute averages of packed bytes and words respectively. The average computation in these instructions is defined as $(a + b + 1)/2$. In SSE versions, `pavgb` and `pavgw` use MMX registers and compute the averages of 8 packed bytes and 4 packed words respectively. In SSE2 versions, these instructions use XMM registers and compute the averages of 16 packed bytes and 8 packed words respectively. The result is stored in the `dest` operand.

The `pmaxub`, `pmaxsw`, `pminub` and `pminsw` instructions return the maximum and minimum of their packed arguments. The `pmaxub` and `pminub` interpret their arguments as packed unsigned bytes and return the maximum and minimum packed unsigned bytes while `pmaxsw` and

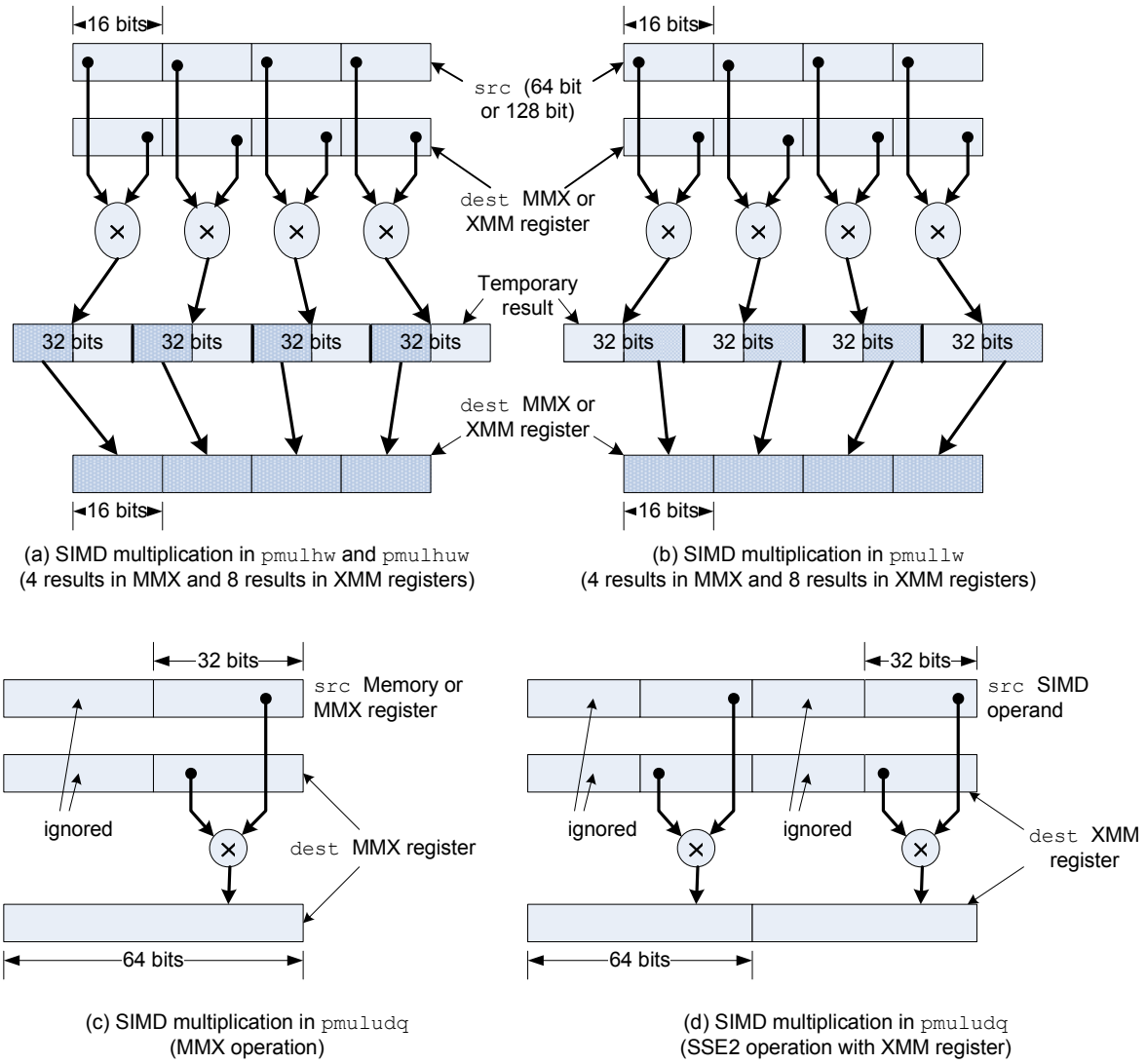


Figure 10.4: SIMD Multiplication in IA32 architectures

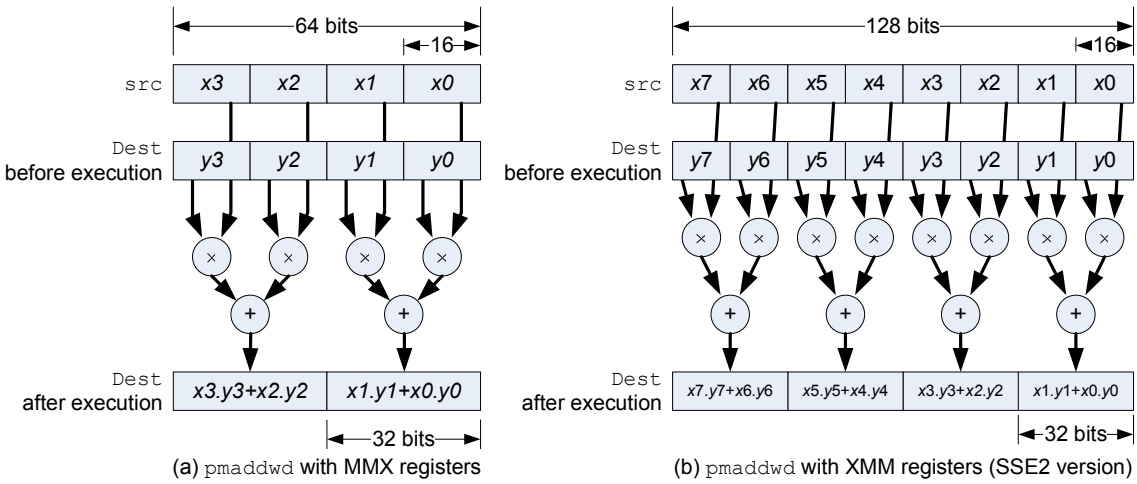


Figure 10.5: `pmaddwd` SIMD instruction

`pminsw` instructions interpret their arguments as packed signed words and return their results as packed signed words.

Finally, `psadbw` instruction takes two arguments as packed unsigned bytes; computes the absolute values of the difference between corresponding bytes of the two arguments and adds these computed values. Result is returned as 16-bit unsigned word in the `dest` register. For instruction `fsadbw %mm2, %mm3`, this operation is shown in figure 10.6. Initial values of `mm2` and `mm3` are 0x34579A1C2406543D and 0x2337B4343F66E28C respectively. After the execution of the `fsadbw` instruction, register `mm3` modifies to contain 0x01BB as shown in the figure. This number is the sum of all absolute differences between respective bytes in the packed data structure.

SIMD logic instructions

The following are the SIMD instructions to perform logic operations.

Instruction	Operation	Instruction sets
Logic operations on MMX/XMM registers		
`pand src, dest`	Bitwise logical AND (`dest` = `dest` AND `src`)	MMX, SSE2
`pandn src, dest`	Bitwise logical AND NOT. (`dest` = NOT(`dest`) AND `src`)	MMX, SSE2
`por src, dest`	Bitwise logical OR (`dest` = `dest` OR `src`)	MMX, SSE2

Instruction	Operation	Instruction sets
`pxor src, dest`	Bitwise logical XOR (`dest` = `dest` XOR `src`)	MMX, SSE2
Logic operations on packed single precision floating point numbers		
`andps src, dest`	Bitwise logical AND (`dest` = `dest` AND `src`)	SSE
`andnps src, dest`	Bitwise logical AND NOT. (`dest` = NOT(`dest`) AND `src`)	SSE
`orps src, dest`	Bitwise logical OR (`dest` = `dest` OR `src`)	SSE
`xorps src, dest`	Bitwise logical XOR (`dest` = `dest` XOR `src`)	SSE
Logic operations on packed double precision floating point numbers		
`andpd src, dest`	Bitwise logical AND (`dest` = `dest` AND `src`)	SSE2
`andnpd src, dest`	Bitwise logical AND NOT. (`dest` = NOT(`dest`) AND `src`)	SSE2
`orpd src, dest`	Bitwise logical OR (`dest` = `dest` OR `src`)	SSE2
`xorpd src, dest`	Bitwise logical XOR (`dest` = `dest` XOR `src`)	SSE2

The logic instructions perform bit-wise operations between `src` and `dest` operands and provide the results in `dest` operand. The following instructions exhibit identical behavior even though they have different machine instruction.

`pand` with MMX operands	`andps`
`pand` with XMM operands	`andpd`
`pandn` with MMX operands	`andnps`
`pandn` with XMM operands	`andnpd`
`por` with MMX operands	`orps`
`por` with XMM operands	`orpd`
`pxor` with MMX operands	`xorps`
`pxor` with XMM operands	`xorpd`

SIMD shift and rotate instructions

The SIMD instruction sets support the following shift and rotate instructions.

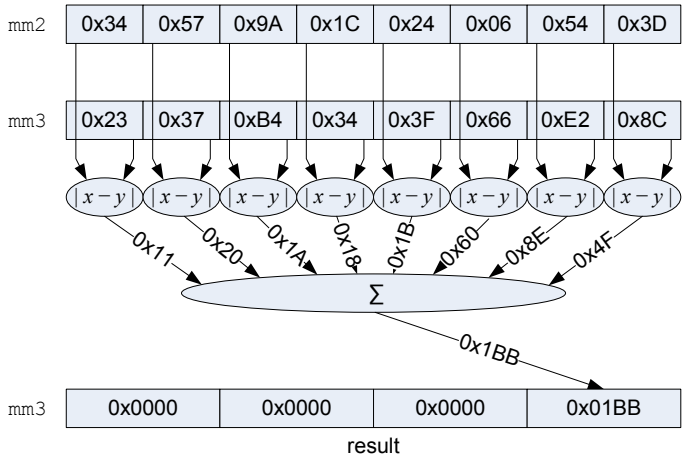


Figure 10.6: Execution of instruction `fsadsw %mm2, %mm3`

Instruction	Operation	Instruction sets
`psllw count, dest`	Shift logical left packed words (16-bits)	MMX, SSE2
`pslld count, dest`	Shift logical left packed longs (32-bits)	MMX, SSE2
`psllq count, dest`	Shift logical left packed quadwords (64-bits)	MMX, SSE2
`pslldq count, dest`	Shift logical left double-quadwords (128-bits)	SSE2
`psrlw count, dest`	Shift logical right packed words (16-bits)	MMX, SSE2
`psrld count, dest`	Shift logical right packed longs (32-bits)	MMX, SSE2
`psrlq count, dest`	Shift logical right packed quadwords (64-bits)	MMX, SSE2
`psrldq count, dest`	Shift logical right double-quadwords (128-bits)	SSE2
`psraw count, dest`	Shift arithmetic right packed words (16-bits)	MMX, SSE2
`psrad count, dest`	Shift arithmetic right packed longs (32-bits)	MMX, SSE2

The `count` operand of these instruction provide the shift count in number of bits for word, long and quad word shifts, and shift count in number of bytes for double-quadwords (`pslldq` and `psrldq` instructions). Shift count can be an immediate constant prefixed by a '$' sign, or an MMX/XMM register or a memory location (64-bit/128-bit wide). In `pslldq` and `psrldq` instructions, however, the shift count can only be an immediate constant.

In logical left shift, incoming bits are set to 0. Thus if the shift count is $\geq$ 16 for `psllw`, $\geq$32 for `pslld`, $\geq$64 for `psllq` or $\geq$64 for `pslldq` instructions, the result is all 0s. Similarly in the logical right shift operations, incoming bits are also set to 0. In the arithmetic right shift, incoming bits are set to the original sign bit. Logical and arithmetic shift operations are shown pictorially in figure 5.2.

The following are some examples of SIMD shift instructions.

```
pslld $9, %mm3          Shift left each of the 32-bit packed
```

`psllq %xmm1, %xmm2`	Shift left 64-bit packed quadwords in `xmm2` by a 128-bit shift count in `xmm1`.
`psrlw $6, %xmm5`	Shift logical right each of the packed 16-bit words in `xmm5` register by 6 bits.
`psrad 8(%ebx), %xmm4`	Shift right arithmetic 32-bit packed long word integers in `xmm4` by a count stored in the specified 128-bit memory location.

SIMD instruction sets provide a very powerful mechanism to perform repeated operations. As an example consider the following C code for which an Assembly language program using SIMD instruction set is also given. In this code example, all elements in an array of 16 short integers (each 16-bits) are negated.

```
void NegateArray(short a[16]) {
    int i;
    for(i=0; i<16; i++) {
        a[i] = (a[i] == 0x8000)? 0x7FFF : (-a[i]);
    }
}
```

It can be easily realized that the operation being performed within the loop body is negation of an array element with saturation. In 16-bit number representation there is only one case of a negative number (-2^{15}) for which the corresponding positive number can not be represented. In this case, the number is set to the maximum positive value ($2^{15} - 1$).

This code fragment can be written using SIMD instruction sets as the following.

```
// Address of the first element of the array is passed
// on the stack at 4(%esp).
.globl NegateArray
NegateArray:
   movl   4(%esp), %ebx // Address of the array in ebx
   movdqu (%ebx), %xmm0 // Load 8 words(a[0..7]) in xmm0
   psllw  $16, %xmm1    // set xmm1 = 0
   psubsw %xmm0, %xmm1  // word subtract with saturation
   movdqu %xmm1, (%ebx) // Store a[0..7]
   movdqu 16(%ebx), %xmm0 // Load a[8..15]
   psllw  $16, %xmm1
   psubsw %xmm0, %xmm1  // Negate with signed saturation
```

```
movdqu %xmm1, 16(%ebx)
ret
```

The basic algorithm is as following. Using the XMM registers, 8 short variables (16-bits) can be loaded from the memory. `movdqu` instruction is used to load and store these from memory to XMM register or XMM register to the memory. This instruction is explained later in section 10.2.2. We have used `xmm0` register for this purpose. In addition `xmm1` register is used to store all 0s. This is achieved by shifting packed words in `xmm1` by 16 bits to ensure that each word becomes 0. Finally `psubsw` instruction is used to subtract given array from 0 with signed saturation. The same sequence of operations is performed twice, first for a[0] to a[7] and later a[8] to a[15].

10.2.2 Data transfer and conversion instructions

Data transfer instructions

SIMD instruction sets have a variety of instructions to move data from memory to XMM/MMX registers, between XMM/MMX registers and from XMM/MMX registers to memory. Some of these instructions also move data from general purpose registers to XMM/MMX registers or from XMM/MMX registers to general purpose registers.

```
movd src, dest  —  MMX, SSE2
movq src, dest  —  MMX, SSE2
```

The `movd` instruction is used to move a single 32-bit integer from `src` to `dest`. One argument of the instruction must be an MMX/XMM register while the other argument must be a general purpose 32-bit register or a 32-bit memory location. When the MMX/XMM register is the destination of the instruction, lower 32-bits are set to the value from the source operand while the remaining bits are set to 0. When the MMX/XMM register is the source operand, only lower 32-bits are moved to the destination and other bits remain unchanged.

For example, instruction '`movd %xmm3, %eax`' copies 32-bit data in lower 32-bits of `xmm3` register to `eax`. Instruction `movd %ecx, %mm7` copies 32-bit number in `ecx` register to lower 32-bits of `mm7` register and sets the upper 32-bits of the `mm7` register to 0. Instruction `movd mem, %mm4` reads 32-bits of data from memory location `mem` and copies that to the lower 32-bit of `mm4` register while setting the upper 32-bits to 0.

The `movq` instruction works almost in a similar manner as `movd` instruction except that it moves a 64-bit integer. One argument of this instruction must be in an MMX/XMM register while the other argument must be an MMX/XMM register or a 64-bit memory. As this instruction

moves 64-bit data items, it can not be used to move data between general purpose registers and SIMD registers. In addition, this instruction can not be used to copy data between an MMX register and an XMM register. For example, '`movq %xmm2, %mm3`' is an invalid instruction. As another example, instruction '`movq %xmm2, %xmm4`' copies lower 64-bits of `xmm2` register to lower 64-bits of `xmm4` register. Upper 64-bits of `xmm4` register are set to 0. These instructions are useful to move data between XMM registers and memory. For example, `movq %xmm2, mem` instruction writes 64-bit value stored in lower half of `xmm2` register to memory location `mem`, and `movq mem %xmm3` instruction reads a 64-bit value from memory lcoation `mem` to lower 64-bits of `xmm3` register and sets the upper 64-bits of the `xmm3` register to 0.

```
movss src, dest  —  SSE
movsd src, dest  —  SSE
```

The `movss` and `movsd` (move scalar) instructions are used to move a single data item from one XMM register to another, from memory to XMM register or from XMM register to memory. The data is 32-bit single precision floating point number in case of `movss` instruction while it is 64-bit double precision floating point number in case of `movsd` instruction. When the data item is loaded from the memory, the upper 96 bits or the upper 64 bits are set to 0 for `movss` and `movsd` instructions respectively. When data is copied from one XMM register to another, or from XMM register to memory only 32-bits or 64-bits, as the case may be, are modified in the destination operand. All other bits remain unchanged. The following are some examples of these instructions.

`movss %xmm2, %xmm3`	`xmm3`$[31:0] \Leftarrow$ `xmm2`$[31:0]$. All other bits remain unchanged in `xmm3`.
`movsd %xmm1, mem`	Store 64-bit data from `xmm1`$[63:0]$ to memory location `mem`.
`movss mem, %xmm4`	32-bit data is read from memory location `mem` and stored in `xmm4` (*i.e.* `xmm4`$[31:0] \Leftarrow$ `mem`). Other bits are set to 0 (*i.e.* `xmm4`$[127:32] \Leftarrow 0$).

```
movhpd src, dest  —  SSE2
movlpd src, dest  —  SSE2
movhps src, dest  —  SSE
movlps src, dest  —  SSE
```

The `movhpd` instruction copies a floating point number (double precision) from upper half of an XMM register to memory or from memory to the upper half of an XMM register. The lower half remains unchanged.

The movlpd instruction works in a similar manner except that it uses the lower half of the XMM register to move to memory or from memory. The upper half remains unchanged.

For example movhpd %xmm3, mem instruction stores xmm3$[127:64]$ to 64-bit memory location mem. Instruction movlpd abc, %xmm2 reads 64-bit double precision floating point number from memory location abc and stores it xmm2$[63:0]$ while keeping all other bits of the xmm2 registers unchanged.

The movhps and movlps instructions are similar to movhpd and movlpd instructions except that these instructions move 2-packed single precision floating point numbers from upper half or lower half of an XMM register to memory or vice versa. For example, the instruction 'movlps %xmm4, mem' stores xmm4$[63:0]$, taking them to be 2-packed single precision floating point numbers, to 64-bit memory location mem while keeping the remaining bits unchanged. Similarly execution of instruction 'movhps mem, %xmm7' leads to reading 2-packed single precision floating point numbers from memory and storing them in the upper half of the xmm7 register while keeping xmm7$[63:0]$ unchanged.

The following is an example code to load four single-precision floating point numbers stored at memory location 8(%ebx) onwards to XMM register xmm5.

```
// Load 4 single-precision floating point numbers from
// memory to XMM register
// The address of the first set of two numbers is 8(%ebx)
// while that of the next set of two numbers is 16(%ebx)
    movlps 8(%ebx), %xmm5  // Load numbers in lower half
    movhps 16(%ebx), %xmm5 // Load in upper half
```

```
movaps src, dest  —  SSE
movapd src, dest  —  SSE2
movups src, dest  —  SSE
movupd src, dest  —  SSE2
```

The movaps and movups instructions move four single precision floating point numbers from memory to an XMM register or from an XMM register to memory or from an XMM register to another XMM register. These numbers are taken as 4-packed single precision numbers in the XMM register. In the example given above, instruction 'movups 8(%ebx), %xmm5' can be used in place of two instructions.

The movapd and movupd instructions are similar to movaps and movups instructions except that the arguments of the instructions contain two double precision floating point numbers in memory and these numbers are treated as 2-packed double precision numbers in XMM registers.

The mova.. instructions access memory only at aligned addresses. Therefore the memory address must be a multiple of 16 (aligned at 16-byte boundaries). On the other hand movu.. instructions can access data from any address. The mova.. instructions are faster to execute compared to corresponding movu.. instructions and should be used only if the address is guaranteed to be aligned.

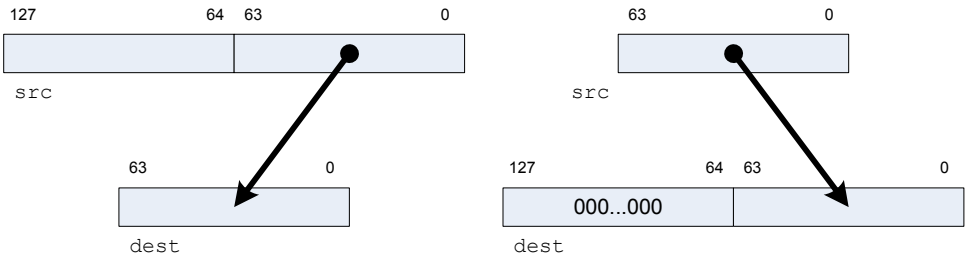
```
movdqa src, dest   —  SSE2
movdqu src, dest   —  SSE2
movq2dq src, dest  —  SSE2
movdq2q src, dest  —  SSE2
```

The movdqa and movdqu instructions are used to copy contents of an XMM register specified as src operand to another XMM register or to a double quadword (128-bits) memory location specified as dest operand. These instruction can also be used to load 128-bits of information from memory to the XMM register specified as dest. While loading or storing data in memory, the memory address must be aligned to 16-byte (128-bit) address in movdqa instruction or may be unaligned in movdqu instruction. movdqa instruction is faster and must be used if the addresses are known to be aligned. movdqu instruction is slower but is more general than movdqa instruction and will not cause any bus error exception in Linux.

The movq2dq and movdq2q instructions are the only instructions to move data between XMM and MMX registers. The src and dest operands of movq2dq instruction can only be an MMX register and an XMM register respectively. This instruction copies 64-bit contents of the MMX register to lower 64-bits of XMM register and sets the upper 64-bits of the XMM register as 0. The movdq2q instruction is used to copy the contents from lower 64 bits of an XMM register specified as src operand to the MMX register specified as dest operand. Figure 10.7 shows the behavior of these two instructions.

```
movhlps src, dest  —  SSE
movlhps src, dest  —  SSE
```

The movhlps and movlhps instructions work with the XMM registers only. They move 2-packed single precision floating point numbers from upper half of the src XMM register to lower half of dest XMM register (movhlps instruction) or vice versa (movlhps instruction). All other bits of the dest XMM register remain unmodified. For example, the instruction 'movhlps %xmm3, %xmm5' copies contents of xmm3[127:64] to xmm5[63:0] while keeping xmm5[127:64] unchanged. These instructions can be used to swap the contents of XMM registers in various ways. The following is an example code snippet that can be used to swap the contents of xmm3[127:64] and xmm3[63:0] using xmm4 as a temporary register.



(a) XMM to MMX register movement (`movdq2q` instruction)

(b) MMX to XMM register movement (`movq2dq` instruction)

Figure 10.7: Movement of data between XMM and MMX registers

```
// Swap upper and lower halves of xmm3.
movhlps %xmm3, %xmm4 // Save upper half of xmm3 in xmm4
movlhps %xmm3, %xmm3 // Copy lower half to upper
movlhps %xmm4, %xmm3 // Copy saved part to lower half
```

The SIMD instruction sets provide three instructions to extract the sign of their arguments and provide the values in one of the general purpose registers.

```
movmskps src, dest  —  SSE
movmskpd src, dest  —  SSE2
pmovmskb src, dest  —  SSE2
```

The `movmskps` and `movmskpd` instructions take an XMM register as `src` operand and one 32-bit general purpose register as `dest` operand. The `movmskps` instruction copies the sign bits of four single precision floating point numbers in the XMM registers into the lower four bits of the 32-bit register. Other 28 bits of the register are set to 0. In IEEE754 floating point representation, the most significant bit of the single precision floating point number indicates the sign. Hence the instruction effectively copies bits 31, 63, 95 and 127 of specified XMM register to bits 0, 1, 2 and 3 of the 32-bit register in that order. All other bits of the 32-bit register are set to 0.

The `movmskpd` instruction copies the sign bits of 2-packed double precision floating point numbers into the lower two bits of the 32-bit register. Remaining 30 bits of the register are set to 0. Thus the instruction effectively copies bits 63 and 127 of XMM register into bits 0 and 1 of the 32-bit register while turning bits 2 to 31 to 0.

The `pmovmskb` instruction takes `src` operand as MMX or XMM register. Assuming these register to have packed signed bytes, the sign bits of 8 bytes in MMX or 16 bytes in XMM registers are copied to the

dest register. Remaining bits of the general purpose 32-bit register specified as dest operand are set to 0. For example, the instruction pmovmskb %mm4, %eax copies bits 7, 15, 23, 31, 39, 47, 55 and 63 of mm4 register to bits 0 to 7 of the eax register while setting bits 8 to 31 of the eax register to 0. In a similar way 'pmovmskb %xmm7, %eax' would copy 16 sign bits of xmm7 register to lower half of register eax (or to register ax). Upper half of register eax becomes all zeros.

Data conversion instructions

The SIMD instruction sets provide a variety of data conversion instructions. The following are the data conversion instructions that convert data between various integer formats.

Conversion to smaller width		
packsswb src, dest	—	MMX, SSE2
packssdw src, dest	—	MMX, SSE2
packuswb src, dest	—	MMX, SSE2
Interleaved unpacking		
punpckhbw src, dest	—	MMX, SSE2
punpckhwd src, dest	—	MMX, SSE2
punpckhdq src, dest	—	MMX, SSE2
punpckhqdq src, dest	—	SSE2
punpcklbw src, dest	—	MMX, SSE2
punpcklwd src, dest	—	MMX, SSE2
punpckldq src, dest	—	MMX, SSE2
punpcklqdq src, dest	—	SSE2

The pack... instructions convert packed data from larger bit width to smaller bit-width. The src operand of these instructions can be an MMX register, XMM register or a memory location while the dest operand must be only an MMX/XMM register. Size of the memory operand is determined by size of the dest operand. If the dest operand is an XMM register, the memory operand is 128-bit wide. If the dest operand is an MMX register, the memory operand is 64-bit wide. These instructions take input values in src and dest operands and provide results in dest operand.

packsswb instruction moves signed packed words (16-bits) to signed packed bytes with signed saturation. Thus if the value of the packed word is larger than 0x7F (+127), the corresponding byte value will be set to 0x7F. Similarly if the value of the packed word is smaller than 0x80 (−128), the corresponding value of the byte will be set to 0x80. packssdw instruction is similar to the packsswb. It converts signed packed double words (long, 32-bits) to packed words (16-bits) with signed saturation. If the value in packed long integer is larger than 0x7FFF (or $2^{15} - 1$) then the corresponding packed word is set

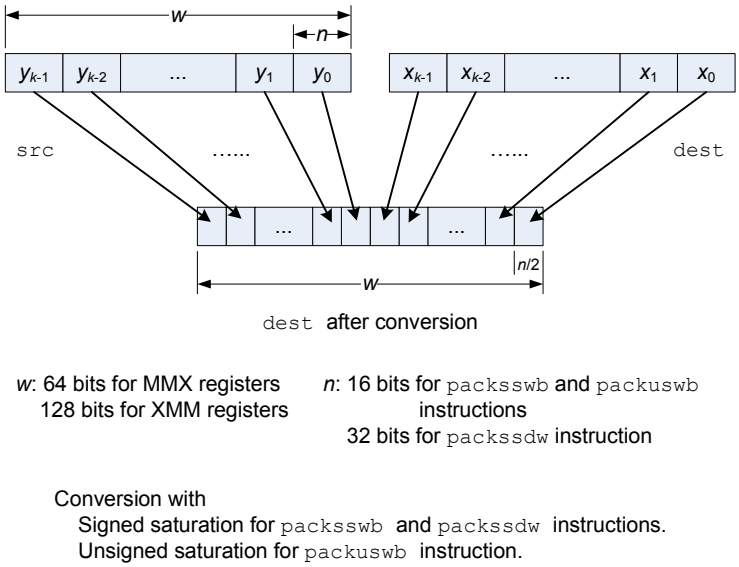


Figure 10.8: Behavior of `packsswb`, `packssdw` and `packuswb` instructions

to 0x7FFF. On the other hand if the value in the packed long word is smaller than 0x8000 (or -2^{15}), the corresponding packed word is set to 0x8000. `packuswb` instruction converts unsigned packed word (16-bits) to unsigned packed bytes with unsigned saturation. In this mode, if the packed word is larger than 0xFF (or, 255), the corresponding packed byte is set to 0xFF. The behavior of these three instructions is shown in figure 10.8.

`punpck...` instructions combine values in two operands and return the interleaved results in the `dest` operand. The `punpckhbw`, `punpckhwd`, `punpckhdq` and `punpckhqdq` instructions combine packed items in the upper half of their argument. On the other hand instructions `punpcklbw`, `punpcklwd`, `punpckldq` and `punpcklqdq` combine packed items in lower half of their arguments.

The `punpckhbw`, `punpckhwd`, `punpckhdq`, `punpcklbw`, `punpcklwd` and `punpckldq` instructions can operate in 64-bit or 128-bit modes. In 64-bit mode, the `src` operand can be either an MMX register or an 8-byte memory location. In this mode, `dest` operand can only be an MMX register. In 128-bit mode, the `src` operand can be either an XMM register or a 16-byte memory location. The `dest` operand can only be an XMM register.

The `punpckhqdq` and `punpcklqdq` instructions can operate only in 128-bit modes where the `src` operand can be an XMM register or a 16-byte memory location while the `dest` operand can only be an XMM

register.

Behavior of `punpck...` instructions is shown in figure 10.9. The `punpckh...` instructions use packed data in the upper half of their `src` and `dest` operands and modify the `dest` argument to provide alternate packed item from the `src` and `dest`. `punpckl...` instructions use the packed data in the lower half of their `src` and `dest` operands. Both instructions pack alternate data items in the result from `src` and `dest` arguments respectively as shown in figure 10.9.

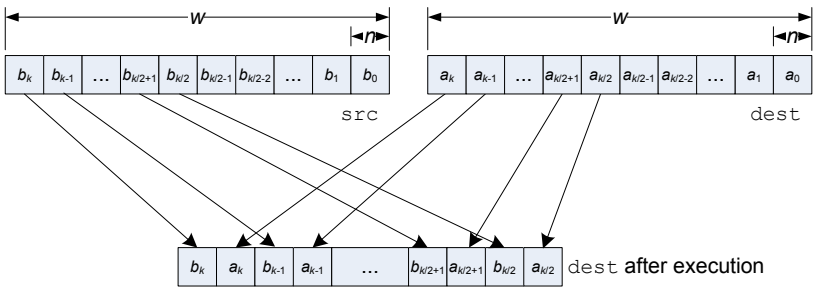
Let's consider the following C code that converts an array of `short` to an array of `char` ensuring a reasonable way of handling overflow.

```
void ShortToChar(short s[24], char c[24]) {
    int i;
    for(i=0; i<24; i++) {
        if (s[i] < −128) c[i] = −128;
        else if (s[i] > 127) c[i] = 127;
        else c[i] = s[i];
    }
}
```

This program can be written using instructions from SIMD instruction set to execute much faster than the program written using general purpose instruction set. It may be recalled that C programs pass parameters through stack. Array parameters are passed by address of the first element. Therefore in our example, location `4(%esp)` contains the address of array `s` while location `8(%esp)` contains the address of array `c` at the entry of the function.

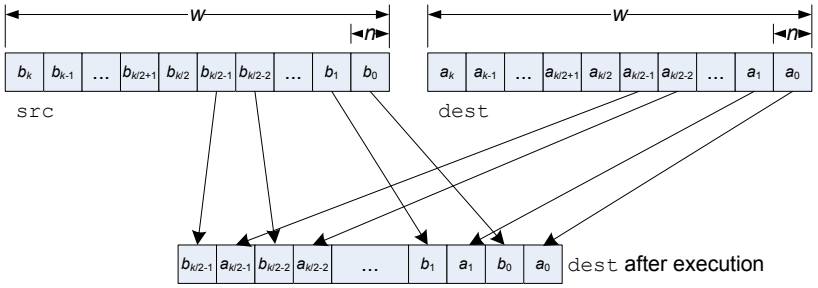
```
1  .globl ShortToChar
2  ShortToChar:
3     movl  4(%esp), %eax
4     movl  8(%esp), %ebx
5     movdqu (%eax), %xmm0 //Read 16 bytes (s[0:7])
6     // Convert 16-bit numbers to 8-bit numbers with
7     // Signed saturation
8     packsswb 16(%eax), %xmm0
9     movdqu %xmm0, (%ebx) //Store c[0:15]
10    // Repeat the same operation now with MMX
11    // instructions for remaining 8 numbers
12    movq 32(%eax), %mm0 //Read 8 bytes (s[16:23])
13    packsswb 40(%eax), %mm0
14    movq %mm0, 16(%ebx) //Store c[16:23]
15    ret
```

First two instructions in this program bring addresses of the arrays `s` and `c` into registers `eax` and `ebx` respectively. Later `movdqu` instruction at line 5 is used to load 16 bytes from memory location whose



(a) Interleaving with `punpckh`... instructions.

w: 64 bits for MMX registers, 128 bits for XMM registers.
n: 8, 16, 32, 64 bits for `punpckhbw`, `punpckhwd`, `punpckhdq`, `punpckhqdq` respectively.
k: 8 or 16 for `punpckhbw` (8:MMX, 16:XMM),
4 or 8 for `punpckhwd` (4:MMX, 8:XMM),
2 or 4 for `punpckhdq` (2:MMX, 4:XMM), and
2 for `punpckhqdq` (only XMM).



(b) Interleaving with `punpckl`... instructions.

w: 64 bits for MMX registers, 128 bits for XMM registers.
n: 8, 16, 32, 64 bits for `punpcklbw`, `punpcklwd`, `punpckldq`, `punpcklqdq` respectively.
k: 8 or 16 for `punpcklbw` (8:MMX, 16:XMM),
4 or 8 for `punpcklwd` (4:MMX, 8:XMM),
2 or 4 for `punpckldq` (2:MMX, 4:XMM), and
2 for `punpcklqdq` (only XMM).

Figure 10.9: Behavior of `punpck...` SIMD instructions

address is given in eax. Since each short variable occupies two bytes in the memory, this instruction brings eight short values (s[0:7]) into register xmm0. These values are packed with (s[8:15]) using packsswb instruction. The src operand of this instruction is specified as the address of s[8] (*i.e.* 16(%eax)). After packing 16-bit data (s[0:7] and s[8:15]) into bytes in register xmm0, these 16 bytes are written to array c using movdqu instruction in line 9. Thus 16 short values get converted into 16 byte wide values. Later instructions perform the similar operations but this time using MMX register mm0 to convert remaining 8 short integer values to 8 byte wide values. The starting addresses of short values are 32(%eax) from where four short values (s[16:19]) are read into mm0 register in line 12. These are then packed as bytes along with the remaining four short values (s[20:23]) with start address as 40(%eax). The result in mm0 register is stored in c[16:23].

Programs often require similar conversions between various large arrays. The SIMD data conversion instructions work efficiently on such programs and speed up execution of the programs.

SIMD instruction sets also provide instructions to convert floating point numbers to integers, integers to floating point numbers and floating point numbers in one format to another. These instructions are given below.

Conversion between packed integers and floats		
`cvtpi2ps src, dest`	—	SSE
`cvtps2pi src, dest`	—	SSE
`cvttps2pi src, dest`	—	SSE
`cvtpi2pd src, dest`	—	SSE2
`cvtpd2pi src, dest`	—	SSE2
`cvttpd2pi src, dest`	—	SSE2
`cvtdq2ps src, dest`	—	SSE2
`cvtps2dq src, dest`	—	SSE2
`cvttps2dq src, dest`	—	SSE2
`cvtdq2pd src, dest`	—	SSE2
`cvtpd2dq src, dest`	—	SSE2
`cvttpd2dq src, dest`	—	SSE2
Conversion between packed floats		
`cvtpd2ps src, dest`	—	SSE2
`cvtps2pd src, dest`	—	SSE2
Conversion between scalar integers and floats		
`cvtsi2ss src, dest`	—	SSE
`cvtss2si src, dest`	—	SSE
`cvttss2si src, dest`	—	SSE
`cvtsi2sd src, dest`	—	SSE2
`cvtsd2si src, dest`	—	SSE2
`cvttsd2si src, dest`	—	SSE2
Conversion between scalar floats		
`cvtss2sd src, dest`	—	SSE2
`cvtsd2ss src, dest`	—	SSE2

The `cvtpi2ps`, `cvtps2pi` and `cvttps2pi` instructions convert between packed single precision floating point numbers in an XMM register and 2-packed 32-bit (long) integers in an MMX register. The `cvtpi2ps` instruction takes its `src` operand as an MMX register or an 8-byte memory location and converts two long integers (each 32-bit wide) to two single precision floating point numbers and stores the result in an XMM register specified as `dest` operand. While storing the results, only the lower two floating point numbers in XMM are updated. The upper two floating point numbers previously stored in the XMM register are not modified.

The `cvtps2pi` and `cvttps2pi` instructions convert two single precision floating point numbers stored in lower half of the specified XMM register, or in an 8-byte memory location, to 2-packed 32-bit (long) integers specified as `dest` argument of the instruction. The `dest` argument of these instructions can only be an XMM register. When the conversion is inexact, rounding of data is performed. In `cvtps2pi` instruction, rounding is performed as per the rounding control bits in the `mxcsr` register. In `cvttps2pi` instruction, rounding is performed

by truncation or rounding-toward-zero. When the result of the conversion is larger than what can be stored in 32-bit long word, the integer is set to 0x80000000.

The `cvtpi2pd`, `cvtpd2pi` and `cvttpd2pi` instructions operate in a similar manner except that they use double precision floating point numbers instead of single precision floating point numbers. Thus in `cvtpi2pd` instructions, the destination is an XMM register which holds 2-packed double precision floating point numbers. In `cvtpd2pi` and `cvttpd2pi` instructions, the source is an XMM register or a 16-byte memory location that provides 2-packed double precision floating point numbers. These two instructions also perform rounding in a manner similar to the rounding performed by `cvtps2pi` and `cvttps2pi` instructions. `cvtpd2pi` instruction uses the rounding control information in `mxcsr` register while `cvttpd2pi` instruction truncates the result.

The `cvtdq2ps`, `cvtps2dq` and `cvttps2dq` instructions convert data between 4-packed long words (32-bits) and 4-packed single precision floating point numbers. The `src` operands of these instructions are either an XMM register or a 16-byte (128-bit) memory location. The `dest` operands can only be XMM registers. Rounding behaviors of `cvtps2dq` and `cvttps2dq` instructions are as that of `cvtps2pi` and `cvttps2pi` instructions. `cvtps2dq` instruction uses the rounding control information in `mxcsr` register while `cvttps2dq` instruction truncates the result.

The `cvtdq2pd`, `cvtpd2dq` and `cvttpd2dq` instructions convert data between two long words (32-bits) and 2-packed double precision floating point numbers. These instructions show a similar behavior as that of `cvtpi2pd`, `cvtpd2pi` and `cvttpd2pi` instructions except that the integers are in XMM registers. Thus `cvtdq2pd` instruction takes its `src` operand in lower half of an XMM register or in an 8-byte (64-bit) memory location. Two 32-bit integers in the `src` operand are converted to 2-packed double precision floating point numbers and stored in the XMM register specified as `dest` operand. The `cvtpd2dq` instruction takes 2-packed double precision floating point numbers in an XMM register or in a 16-byte memory location specified as `src` operand and converts that to two long word (32-bit) integers. The integers are then resturned in lower half of an XMM register specified as `dest` operand while the upper half of the XMM register is set to all zeros. Rounding mechanism and handling of overflows in these instructions is similar to that in the `cvtps2pi` and `cvttps2pi` instructions. `cvtpd2dq` instruction uses the rounding control information in `mxcsr` register while `cvttpd2dq` instruction truncates the result.

The `cvtps2pd` and `cvtpd2ps` instructions convert data between single precision floating point numbers and double precision floating point numbers. The `cvtps2pd` instruction takes its `src` operand as an XMM register or an 8-byte (64-bit) memory location. The two single precision

floating point numbers in the lower half of the XMM register or in the memory are then converted to two double precision floating point numbers and stored in the XMM register specified as `dest` operand of the instruction. The `cvtpd2ps` instruction converts two double precision floating point numbers in `src` operand, specified as an XMM register or in a 16-byte memory location, to two single precision floating point numbers. The results are then stored in the lower half of the XMM register specified as `dest` operand. The upper half of the XMM register is set to all zeros. We have already seen that 32-bits pattern of all zeros represents +0.0 in IEEE754 single precision floating point number format. Thus the result becomes 4-packed single precision floating point numbers with upper two numbers being both +0.0 while converting a double precision number to single precision number, rounding is performed as per the rounding control bits in `mxcsr` register.

The `cvtsi2ss`, `cvtss2si` and `cvttss2si` instructions convert data between a scalar single precision floating point number in an XMM register and a 32-bit integer. The `cvtsi2ss` instruction takes its `src` operand as a 32-bit integer in a general purpose 32-bit register or in a 4-byte (32-bit) memory location and converts that to a single precision floating point number. The result is then stored in the lower 32-bits of an XMM register specified as `dest` operand. Other 96 bits of the XMM register remain unchanged. The `cvtss2si` and `cvttss2si` instructions take their `src` operand as XMM register or a 32-byte memory location. The least significant 32-bits of the XMM register or the data read from the memory location is handled as a single precision floating point number and converted to a single 32-bit number. The result is then stored in a general purpose 32-bit register specified as `dest` operand of the instructions. While converting floating point numbers, rounding control is performed if the conversion is not inexact. The `cvtss2si` instruction performs the rounding according to the rounding control bits in `mxcsr` register. The `cvttss2si` instruction performs the truncation (rounding-toward-zero) for inexact conversions. When the converted integer is too large to fit in 32-bit numbers, a constant 0x80000000 is returned.

The `cvtsi2sd`, `cvtsd2si` and `cvttsd2si` instructions work in the similar manner as `cvtsi2ss`, `cvtss2si` and `cvttss2si` instructions except that these instructions convert a single 32-bit integer to, and from, a single double-precision floating point number. The `cvtsi2sd` instruction takes a 32-bit integer as `src` operand in a general purpose 32-bit register or in a 32-bit memory location. The integer is then converted to a double precision floating point number and stored in the least significant 64-bits of the XMM register specified as `dest` operand. The `cvtsd2si` and `cvttsd2si` instructions convert a double precision floating point number in the lower half of an XMM register or in a 64-bit memory location specified as `src` operand to a 32-bit integer. The result is then returned in a general purpose register specified as `dest`

operand of the instructions. While converting, rounding is performed and overflow is handled for inexact conversions in a way similar to the corresponding `cvtss2si` and `cvttss2si` instructions.

`cvtss2sd` and `cvtsd2ss` instructions convert scalar floating point numbers between single precision and double precision formats. The `cvtss2sd` instruction takes one single-precision floating point number in the least significant 32-bits of an XMM register or in a 32-bit wide memory location specified as `src` operand and converts that to a single double precision floating point number. The result is stored in lower 64 bits of an XMM register specified as `dest` operand of the instruction. The `cvtsd2ss` instruction takes XMM register or an 8-byte (64-bit) wide memory location as `src` operand and converts one double-precision floating point number in the least significant 64-bits of the XMM register (or in the memory location) to one single-precision floating point number. The result is returned in the least significant 32-bits of the XMM register specified as `dest` operand. All other bits of the `dest` operands of these two instructions remain unmodified.

EXERCISES:

10.1 Write an Assembly language program with instructions from SIMD instruction sets to implement the behavior of the following C function. Assume that each floating point number is smaller than the maximum value of the integer.

```
void FloatDivide(float f[16]) {
    int i, fi;
    for (i=0; i<16; i++) {
        fi = (int)f[i];
        fi = fi >> 4;
        f[i] = (float)fi;
    }
}
```

10.2 Using the SIMD instructions discussed till now, write an Assembly language program to compute the absolute values as per the following C program.

```
void FloatAbs(float f[4]) {
    int i;
    for (i=0; i<4; i++) {
        f[i] = (f[i]<0)? −f[i]: f[i];
    }
}
```

Hint: The floating point numbers in single precision format can be converted to absolute value by turning the sign bit to 0. Prepare in XMM register 4-packed long words where each long word has the most signifi-

cant bit set to 0 and all other bits are 1 (*i.e.* long word is 0x7FFFFFFF). Use this to turn the sign bit of the floating point number to 0.

SIMD instructions for packing and unpacking real numbers

There are a following instructions in the SIMD instruction sets to interleave the floating point numbers in their operands and provide the result.

```
unpckhps src, dest  —  SSE
unpcklps src, dest  —  SSE
unpckhpd src, dest  —  SSE2
unpcklpd src, dest  —  SSE2
```

The `src` operand of all these instructions can be one of the XMM register or a 16-byte (128-bit) memory location. The `dest` operand of these instructions can only be an XMM register.

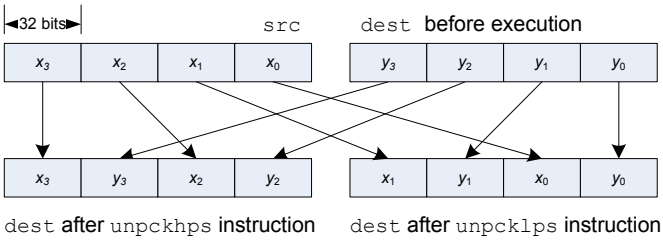
The `unpckhps` instruction (figure 10.10(a)) interleaves two single precision floating point numbers, stored in the upper half of the `src` operand, with two single precision floating point numbers, stored in upper half of the `dest` operand, and returns the result as 4-packed single precision floating point numbers in `dest`. The `unpcklps` instruction interleaves in the similar way except the input numbers are taken from the lower half of the operands as shown in figure 10.10(a).

The `unpckhpd` instruction takes the double precision floating point numbers stored in the upper half of its arguments and pack them as two numbers to return result in `dest` operand. The `unpcklpd` instruction does the similar operation with the double precision floating point numbers stored in the lower half of the arguments. The behavior of these instructions is illustrated in figure 10.10(b).

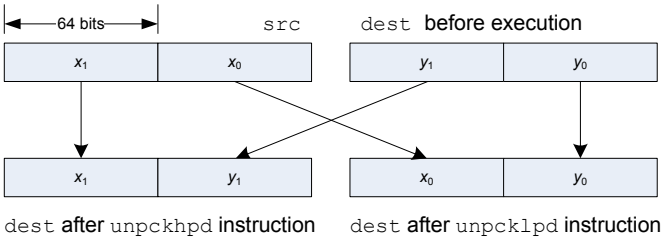
10.2.3 SIMD instructions for Data Shuffling

In addition to the data conversion instructions see till now, there are a few instructions that can be used to shuffle packed data in SIMD registers in various ways. Using these kind of instructions it is possible to rearrange packed data in any order. For example, if a register contain four data items as x_0, x_1, x_2 and x_3, these can be shuffled to reverse the order to get x_3, x_2, x_1 and x_0. The shuffle kind of instructions can also be used to copy a single data item to several locations in the register. For example, it is possible to pack x_1 four times and put result in another register.

The following are the data shuffle instructions in the SIMD instruction sets.



(a) Interleaving with `unpckhps` and `unpcklps` instructions.



(b) Interleaving with `unpckhpd` and `unpcklpd` instructions.

Figure 10.10: The `unpckhps`, `unpcklps`, `unpckhpd` and `unpcklpd` SIMD instructions

```
pshufw cntl, src, dest   —  SSE
pshufd cntl, src, dest   —  SSE2
pshuflw cntl, src, dest  —  SSE2
pshufhw cntl, src, dest  —  SSE2
shufps cntl, src, dest   —  SSE
shufpd cntl, src, dest   —  SSE2
pextrw cntl, src, dest   —  SSE
pinsrw cntl, src, dest   —  SSE
```

Data shuffling instructions take three operands. The `cntl` operand is used to control the shuffling and can only be an immediate constant. The `src` operand provides the data to shuffle. The result of the instruction is returned in `dest` operand. In some instructions, initial values in the `dest` operand also provide data used in shuffle operations. The `pshufw` instruction takes its `src` and `dest` operands as MMX (64-bit) operands. The `src` operand can be an MMX register or a 64-bit memory location while the `dest` operand can only be an MMX register.

Let's represent the 4-packed words (16-bits) in `src` operand as x_3, x_2, x_1, x_0 and in `dest` operands as y_3, y_2, y_1, y_0. The `pshufw` instruction can be used to put any x_i in to any y_j as per the `cntl` operand. The index i of each of x_i can be represented in 2-bits. Two bits in `cntl`[1:0] provide an index i such that x_i is copied to y_0. In a similar manner bits in `cntl`[3:2], `cntl`[5:4] and `cntl`[7:6] provide the indices of x that would be copied to y_1, y_2 and y_3 respectively. This behavior of the instruction is shown in figure 10.11.

For example, if the `cntl` operand is 0x1B (or, 00_01_10_11 in binary), the packed words are reversed as x_0, x_1, x_2 and x_3 in the destination. Similarly when the `cntl` operand is 0x00 (or, 00_00_00_00 in binary), the destination operand gets x_0, x_0, x_0 and x_0 after the execution.

The `pshufd` instruction operates with XMM registers and works in a manner similar to that of the `pshufw` instruction. It shuffles 4-packed long words (32-bits) in the `src` operand which can be an XMM register or a 16-byte memory location. The output is returned in `dest` operand which can only be an XMM register. The behavior of this instruction is shown in figure 10.11.

The `src` operand of `pshuflw` and `pshufhw` can be an XMM register or a 16-byte (128-bit) memory location. The `dest` operand can only be an XMM register. These instructions shuffle four out of eight 16-bit words in the `src` operand. The `pshuflw` instruction (figure 10.12(a)) shuffles four words (16-bits) stored in the lower half of `src` operand. The upper half of the `src` operand is copied to the `dest` operand. Let's represent the 8-packed words in the `src` operand as x_7, x_6, x_5, x_4, x_3, x_2, x_1 and x_0. The 8-packed words in `dest` operand are denoted as y_7, y_6, y_5, y_4, y_3, y_2, y_1 and y_0. If 2-bit indices in the `cntl` are represented as i_3, i_2, i_1, i_0 as shown in figure 10.12, the `pshuflw` instruction defines the destination to contain x_7, x_6, x_5, x_4, x_{i_3}, x_{i_2}, x_{i_1} and x_{i_0}.

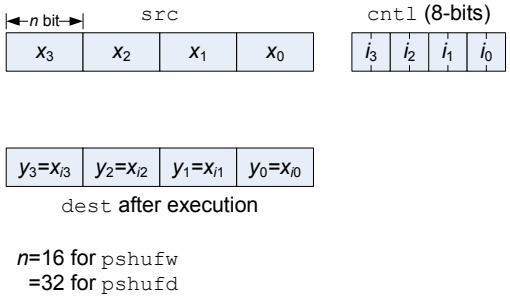


Figure 10.11: The pshufw and pshufd SIMD instructions

The pshufhw instructions shuffles data in the upper half while keeping the lower half of the destination unchanged. It therefore sets the output to x_{4+i_3}, x_{4+i_2}, x_{4+i_1}, x_{4+i_0}, x_3, x_2, x_1 and x_0. This operation is shown in figure 10.12(b).

Let's consider an example of reversing the contents of an array of 64 integers. In this example an array 'a' of 64 integers is given as an input and after the execution a[0] get the original value of a[63], a[1] gets the original value of a[62] etc. The following C code is used to implement this logic.

```
void ReverseArray(int a[64]) {
    int i, t;
    for(i=0; i<32; i++) {
        t=a[i];
        a[i]=a[63-i];
        a[63-i]=t;
    }
}
```

Using the powerful SIMD instructions, we shall implement this algorithm to execute much faster than the code written using traditional instructions. We shall use the XMM registers which can store four 32-bit values. There will be eight iterations of the loop in each of which four integers from the front of the array will be swapped with four integers at the end of the array. The following is the function in Assembly language.

```
1 .globl ReverseArray
2 ReverseArray:
3    // At the start of the code, 4(%esp)=starting
4    // address of array a.
5    mov 4(%esp), %eax
6    mov $0, %ebx  // ebx= front index (0..4)
```

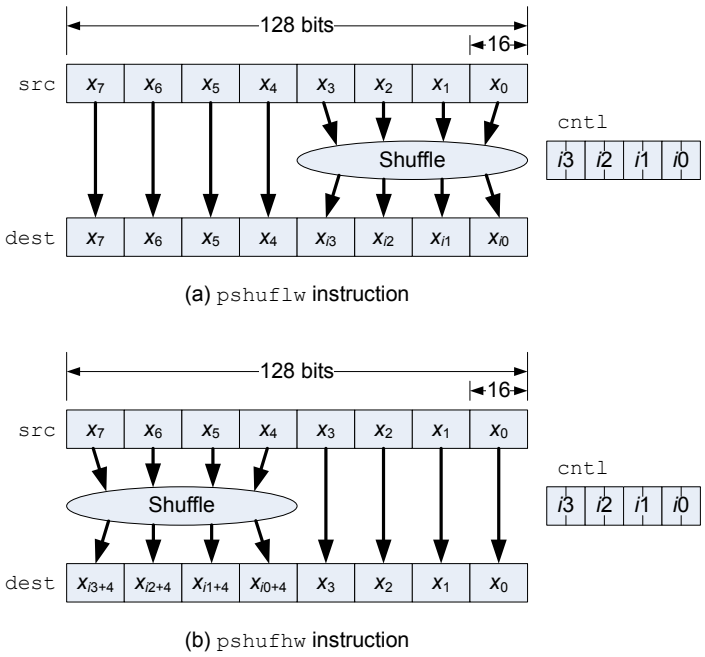


Figure 10.12: The `pshuflw` and `pshufhw` SIMD instructions

```
7     mov $60, %edx // edx = tail index (60..63)
8  R0:
9     movdqu (%eax,%ebx,4), %xmm0 // Load xmm0 from front
10    movdqu (%eax,%edx,4), %xmm1 // xmm1 from back.
11    pshufd $0x1B, %xmm0, %xmm0 // Reverse longs in xmm0
12    pshufd $0x1B, %xmm1, %xmm1 // and xmm1
13    movdqu %xmm1, (%eax,%ebx,4) // Store in front
14    movdqu %xmm0, (%eax,%edx,4) // and back
15    add $4, %ebx // move front ptr forward
16    sub $4, %edx // move back ptr backward
17    cmp %edx, %ebx // Loop as long as the front index
18    jl R0  /* is smaller than the tail index. */
19    ret
```

The Assembly language program works as follows. We use registers `ebx` and `edx` to store the front and tail indices of the input array. The initial values of these registers are set to 0 and 60 (lines 6 and 7) to represent integers 0...3 and 60...63. The `movdqu` instructions in lines 9 and 10 load data from memory to `xmm0` and `xmm1` registers. These integers are reversed in the registers using shuffle instructions. For doing this the `cntl` parameter of the shuffle instructions is 00_01_10_11 in

binary or 0x1B in hexadecimal. Thus y_3 gets a new value as x_0, y_2 gets x_1, y_1 gets x_2 and y_0 gets x_3. Having reversed both registers, `xmm0` and `xmm1`, the values loaded from the front end are stored at the tail side while the values loaded from the tail end are stored at the front side using `movdqu` instructions in lines 13 and 14. The front and tail index registers are adjusted for the next iteration and as long as the front index is less than the tail index, the new iteration is carried out.

EXERCISES:

10.3 Write an Assembly language function using SIMD instruction set to perform the following algorithm in C.

```
typedef struct {
    int min, max;
} course_data;
void initCourses(course_data courses[128]) {
    int i;
    for(i=0; i<128; i++) {
        courses[i].min = 1000;
        courses[i].max = 0;
    }
}
```

Hint: The C structures are alloted in a complicated manner but in this case, the array `courses` can be treated as an array of 256 integers with alternating `min` and `max` elements. Scalars 1000 and 0 may be loaded using `pushl` instructions followed by movement of scalars in the XMM registers and a `popl` instruction to balance the stack. Additionally try this program second time with `pinsrw` instruction as well (introduced on page 292 and discussed later).

10.4 Implement the following C code behavior in Assembly language function using instructions from SIMD instruction sets.

```
void initFlArray(float f[12]) {
    int i;
    for(i=0; i<12; i++) {
        f[i] = 1.0;
    }
}
```

Hint: Use x87 instructions to load 1.0 and save that in memory. Load it as a scalar in an XMM register treating it as 32-bit integer. Copy it as 4-packed data item using shuffle instructions on all four locations of the XMM register and then store it in memory three times to initialize all floating point numbers.

The `shufps` and `shufpd` instructions shuffle data as per the shuffle control specified as `cntl` operand. The input data is provided in `src`

and dest operands while the result is stored only in dest operand. The dest operand can only be an XMM register while the src operand can be a 16-byte memory location or an XMM register. **** and one 64-bit number in shufpd instruction) ***** In shufps instruction, two 32-bit numbers in the lower half of the dest register are derived from the initial value of the dest register. Two 32-bit numbers in the upper half of the result are derived from the src operand. The Shuffle control cntl controls which data items from the dest are copied to the lower half and which data items from the src are copied to the upper half of the dest.

Let the input numbers be denoted as s_3, s_2, s_1 and s_0 in the src operand of the shufps instruction and as d_3, d_2, d_1 and d_0 in the the initial value of the dest operand (figure 10.13(a)). The cntl operand is an 8-bit number that is broken into four numbers of 2-bit each and represented by i_3, i_2, i_1 and i_0 respectively. The result of the operation of the shufps instruction is represented by the following.

$$\begin{aligned} d_0 &= d_{i_0} \\ d_1 &= d_{i_1} \\ d_2 &= s_{i_2} \\ d_3 &= s_{i_3} \end{aligned}$$

In shufpd instruction, one 64-bit number in the lower half of the dest register is derived from the initial value of the dest register. The other 64-bit number in the upper half of dest register is derived from the src operand. Two bits in the shuffle control cntl controls which value to appear in the final value in dest register. If 2-packed 64-bit input numbers in src operand of shufpd instruction are denoted by s_1 and s_2 and those by d_1 and d_2 in dest operand then the final value of d_1 is chosen from s_1 and s_0 while the final value of d_0 is chosen from d_1 and d_0. The choice is determined by bit 0 and bit 1 of 8-bit cntl operand denoted as i_0 and i_1 respectively. All other bits of the cntl operand are ignored. The result of the shufpd instruction is represented by the following.

$$\begin{aligned} d_0 &= d_{i_0} \\ d_1 &= s_{i_1} \end{aligned}$$

Behavior of the shufps and shufpd instructions is shown in figure 10.13.

The pextrw instruction is used to extract a specified word (16-bit) from MMX or XMM register and store the zero extended result in a

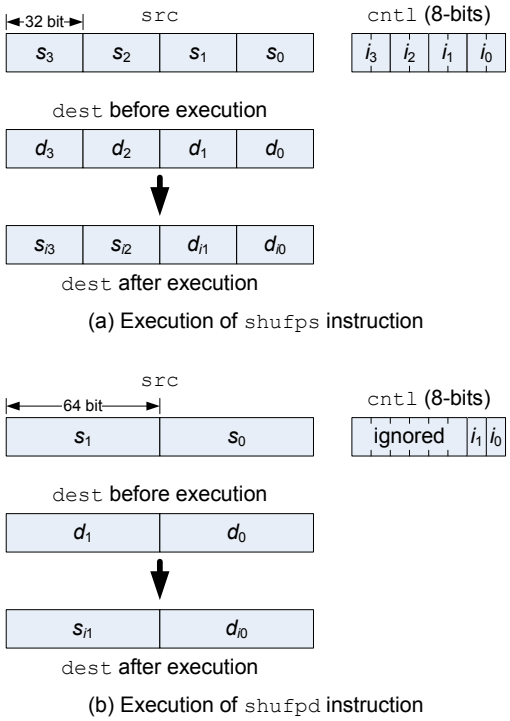


Figure 10.13: The shufps and shufpd SIMD instructions

general purpose 32-bit register (*i.e.* with 0s in the upper half and the extracted 16-bit word in the lower half of the 32-bit register). The `dest` operand for this instruction can only be a 32-bit general purpose register while the `src` operand can be an MMX or XMM register. In case, an MMX register is chosen, only the lower two bits of `cntl` operand are used and the remaining bits are ignored. Thus these bits provide the index of the word between 0 to 3 in the MMX register. In case of an XMM register, the lower three bits of the `cntl` operand are used which provide an index between 0 to 7 for the 16-bit word stored in the XMM register.

The `pinsrw` instruction modifies the specified word, whose index is given in `cntl` operand, of chosen MMX or XMM register specified as `dest` operand. The `src` operand can be a 32-bit general purpose register or a 2-byte (16-bit) memory location. In case, a 32-bit general purpose register is used, only the least significant 16-bits are used.

10.2.4 Comparison instructions

The SIMD instruction sets provide several comparison instructions for data in SIMD registers or in memory. The following instruction are provided for the integer comparison.

`pcmpeqb src, dest`	—	MMX, SSE2
`pcmpeqw src, dest`	—	MMX, SSE2
`pcmpeqd src, dest`	—	MMX, SSE2
`pcmpgtb src, dest`	—	MMX, SSE2
`pcmpgtw src, dest`	—	MMX, SSE2
`pcmpgtd src, dest`	—	MMX, SSE2

The `pcm...` instructions compare their `src` and `dest` operands and provide result of the comparison in the `dest` operand. In the MMX versions, the `src` can be a 64-bit memory location or an MMX register while the `dest` can only be an MMX register. In the SSE2 version, the `src` can be a 128-bit memory location or an XMM register while the `dest` can only be an XMM register.

The `pcmpeqb`, `pcmpeqw` and `pcmpeqd` instructions compare packed bytes, packed words (16-bits) or packed long words (double word or 32-bits) respectively. The output is provided as packed byte, word (16-bits) or long word (32-bits) in `pcmeqb`, `pcmeqw` and `pcmeqd` instructions respectively. The corresponding data items of the packed data are compared for equality. If two data items `src`[i] and `dest`[i] are equal, the i^{th} data item in the result is set to all 1s. Otherwise it is set to 0.

The `pcmpgtb`, `pcmpgtw` and `pcmpgtd` instructions compare data in bytes, words and long word formats respectively and set the result to all 1s or all 0s depending upon the outcome of comparison. If the `dest`[i]

is more than the src[i] using signed comparison, the dest is replaced by all 1s. Otherwise it is replaced by all 0s.

Figure 10.14 shows the outcome of instructions pcmpgtw %mm2, %mm3 and pcmpeqw %mm2, %mm3 for a given set of values in the registers.

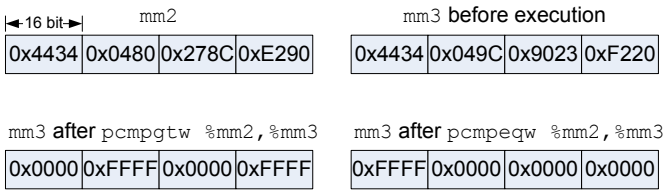


Figure 10.14: The pcmpgtw and pcmpeqw instructions

SIMD instruction sets also provide instructions to compare floating point numbers in a variety of ways. There are two category of instructions for comparison of floating point numbers. One kind of instructions modify their dest argument to provide the result of the comparison in a way similar to the integer comparison instructions. The other kind of instructions set the flags as per the result of comparison and are used for comparing the scalars.

The following are the instructions to compare floating point numbers.

```
cmpccps src, dest  —  SSE
cmpccpd src, dest  —  SSE2
cmpccss src, dest  —  SSE
cmpccsd src, dest  —  SSE2
comiss src, dest   —  SSE
comisd src, dest   —  SSE2
ucomiss src, dest  —  SSE
ucomisd src, dest  —  SSE2
```

The floating point comparison provided by SSE and SSE2 instruction sets can perform the following kinds of comparisons.

cc	Comparison type
eq	True if dest is equal to src. For equality, $+0$ is considered equal to -0. If any one operand is NaN, result is false.
lt	True if dest is less than src. If any one operand is NaN, result is false.
le	True if dest $\leq$ src. If any one operand is NaN, result is false.
unord	True if either dest or src is unordered and can not be compared. Therefore if one of the operand is a NaN, the result is true.

cc	Comparison type
neq	True if dest is not equal to src. If one operand is a NaN, result is true.
nlt	True if dest is $\geq$ src. If one operand is a NaN, result is true.
nle	True if dest $>$ src. If any one operand is NaN, result is true.
ord	True if dest and src are ordered (comparable). Therefore only when one of the operand is a NaN, the result is false.

The src operand of cmp... instructions can be an XMM register or a 16-byte (128-bit) memory location. The dest operand can only be an XMM register. The cmp... instructions compare their floating point operands for the specified condition and when the result is true, corresponding item in dest XMM register is replaced by a bit pattern that is all 1s. If the result is false, corresponding item in dest XMM register is replaced by all 0s.

The cmp*cc*ps instructions compare 4-packed single precision floating point numbers in src with 4-packed floating point numbers in dest and set each of the four 32-bit numbers in the dest as 0xFFFFFFFF or 0x0 depending upon the outcome of the comparison. For example, cmpeqps %xmm0, %xmm1 instruction will modify xmm1 register to represent the result of comparison of values in xmm0 and xmm1 registers. If the four numbers stored in these registers are denoted by x_0, x_1, x_2 and x_3 in xmm0 and y_0, y_1, y_2 and y_3 in xmm1 then the new value of y_i will be 0 if $x_i \neq y_i$ while it will be 0xFFFFFFFF if $x_i = y_i$.

The cmp*cc*pd instruction compares 2-packed double precision floating point numbers in src and dest operands and set the corresponding 64-bit value in the dest operand as all 0s if the comparison result is false or all 1s if the comparison result is true.

The cmp*cc*ss and cmp*cc*sd instructions compare only one floating point numbers of the src and dest operands in the least significant part of the operand. Outcome of the comparison is given in the same part of the XMM register specified as dest. For example, instruction cmpltss %xmm0, %xmm1 compares the single precision floating point numbers in xmm0[0:31] and xmm1[0:31] and if xmm1[0:31] is smaller than xmm0[0:31], xmm1[0:31] is set to all 1s. Otherwise xmm1[0:31] is set to all 0s. In a similar manner, instruction cmpnltsd %xmm3, %xmm2 compares the double precision floating point numbers stored in the lower half of registers xmm2 and xmm3. The lower half of xmm2 register is modified to reflect the result of the comparison. If xmm2[63:0] is more than or equal to xmm3[63:0], xmm2[63:0] gets a value of 64 bits all set to 1. Otherwise they are all set to 0.

All other bits of the dest register remain unmodified in cmp*cc*ss and cmp*cc*sd instructions. The src operand of these instructions can

be a 2-byte or 4-byte memory operand (for single precision and double precision floating point numbers respectively) or an XMM register. The `dest` operand can only be an XMM register.

The `comiss` and `comisd` instructions compare their scalar arguments in `src` and `dest` operands and set the flags in `eflags` register without modifying any of the operands. The `src` operand can be a 32-bit memory location for `comiss` and 64-bit memory location for `comisd` instructions. It can also be an XMM register in which case only the lower 32-bit or 64-bit value in the register is used for comparison. The `dest` operand can only be in an XMM register. The following table shows the outcome of the comparison.

Comparison	flags		
	ZF	PF	CF
Unordered	1	1	1
`dest>src`	0	0	0
`dest<src`	0	0	1
`dest=src`	1	0	0

The `ucomiss` and `ucomisd` instructions are similar to `comiss` and `comisd` instructions and have only a different floating point exception behavior. Under GNU/Linux, these instructions therefore work in an identical manner as that of `comiss` and `comisd` instructions.

EXERCISES:

10.5 Implement the following C algorithm using instructions from SIMD instruction sets.

```
void LimitArrayDiff (short a[80], short b[80]) {
    int i;
    for(i=0; i<80; i++) {
        a[i] = (a[i] > b[i]) ? (a[i]−b[i]): 0;
    }
}
```

Hint: Load data in XMM registers (and save `a` in another XMM register as well), subtract and keep the results in an XMM register. Compare `a` and `b` to set the bit mask to 0s when the result of the comparison is false. Perform a logical AND operation on the comparison result with the subtraction result to achieve the desired result. In each iteration of the loop, eight short integers will be operated upon.

Such kind of code fragments occur very frequently in digital signal processing applications.

10.6 Implement the following C algorithm using instructions from SIMD instruction sets.

```
void HalfStrength(short s[80]) {
```

```
    int i;
    for(i=0; i<80; i++) {
        a[i] = (a[i] >> 1)
        if (a[i] == -1) a[i] = 0;
    }
}
```

This algorithm implements reduction in signal strength (possibly after an Analog to Digital converter) by half. By shifting a number using arithmetic shift, a negative number will never become more than -1. Thus this code also checks for the number being -1 and set the value to 0 if true.

10.2.5 Floating point instructions

The SSE and SSE2 instruction sets provide instructions to operate on single precision and double precision floating point numbers as packed numbers or as scalars. The floating point numbers are supported only in the XMM registers as 4-packed single precision floating point numbers in SSE and as 2-packed double precision floating point numbers in SSE2 instruction sets.

The following are the instructions to operate on floating point numbers.

Instruction	Operation	Instn
Instructions to operate on packed data items		
`addps src, dest`	`dest` = `dest` + `src` with packed single precision numbers	SSE
`addpd src, dest`	`dest` = `dest` + `src` with packed double precision numbers	SSE2
`subps src, dest`	`dest` = `dest` − `src` with packed single precision numbers	SSE
`subpd src, dest`	`dest` = `dest` − `src` with packed double precision numbers	SSE2
`mulps src, dest`	`dest` = `dest` ∗ `src` with packed single precision numbers	SSE
`mulpd src, dest`	`dest` = `dest` ∗ `src` with packed double precision numbers	SSE2
`divps src, dest`	`dest` = `dest` / `src` with packed single precision numbers	SSE
`divpd src, dest`	`dest` = `dest` / `src` with packed double precision numbers	SSE2
`sqrtps src, dest`	`dest` = $\sqrt{\texttt{src}}$ with packed single precision numbers	SSE

Instruction	Operation	Instn
`sqrtpd src, dest`	`dest` = $\sqrt{\texttt{src}}$ with packed double precision numbers	SSE2
`maxps src, dest`	`dest` = max(`dest`, `src`) with packed single precision numbers	SSE
`maxpd src, dest`	`dest` = max(`dest`, `src`) with packed double precision numbers	SSE2
`minps src, dest`	`dest` = min(`dest`, `src`) with packed single precision numbers	SSE
`minpd src, dest`	`dest` = min(`dest`, `src`) with packed double precision numbers	SSE2
`rcpps src, dest`	`dest` = 1.0/`src` with packed single precision numbers	SSE
`rsqrtps src, dest`	`dest` = 1.0/$\sqrt{\texttt{src}}$ with packed single precision numbers	SSE
Instructions to operate on scalar floating point data items		
`addss src, dest`	`dest`[31:0] = `dest`[31:0]+`src`[31:0]	SSE
`addsd src, dest`	`dest`[63:0] = `dest`[63:0]+`src`[63:0]	SSE2
`subss src, dest`	`dest`[31:0] = `dest`[31:0]−`src`[31:0]	SSE
`subsd src, dest`	`dest`[63:0] = `dest`[63:0]−`src`[63:0]	SSE2
`mulss src, dest`	`dest`[31:0] = `dest`[31:0]∗`src`[31:0]	SSE
`mulsd src, dest`	`dest`[63:0] = `dest`[63:0]∗`src`[63:0]	SSE2
`divss src, dest`	`dest`[31:0] = `dest`[31:0]/`src`[31:0]	SSE
`divsd src, dest`	`dest`[63:0] = `dest`[63:0]/`src`[63:0]	SSE2
`sqrtss src, dest`	`dest`[31:0] = $\sqrt{\texttt{src}}$[31:0]	SSE
`sqrtsd src, dest`	`dest`[63:0] = $\sqrt{\texttt{src}}$[63:0]	SSE2
`maxss src, dest`	`dest`[31:0] = max(`dest`[31:0], `src`[31:0])	SSE
`maxsd src, dest`	`dest`[63:0] = max(`dest`[63:0], `src`[63:0])	SSE2
`minss src, dest`	`dest`[31:0] = min(`dest`[31:0], `src`[31:0])	SSE
`minsd src, dest`	`dest`[63:0] = min(`dest`[63:0], `src`[63:0])	SSE2
`rcpss src, dest`	`dest`[31:0] = 1.0/`src`[31:0]	SSE
`rsqrtss src, dest`	`dest`[31:0] = 1.0/$\sqrt{\texttt{src}}[31:0]$	SSE

As mentioned earlier, XMM registers support 4-packed single precision or 2-packed double precision floating point numbers. Thus an instruction that operates on packed floating point data structures perform its operation with two or four data items in the XMM registers. For example, instruction `addps` operates with 4-packed single precision floating point numbers.

The `src` operand of all floating point instructions that operate with packed data can be either an XMM register or a 16-byte (128-bit) memory location that contains four single precision or two double precision floating point numbers. The `dest` operand is always an XMM register.

The `src` operand of instructions that operate with scalar data can be an XMM register or a four byte (for a single precision floating point number) or an eight byte (for a double precision floating point number) memory location. If the `src` operand is in an XMM register, only the lower 32-bit for single precision numbers and only the lower 64-bits for double precision numbers are used. The `dest` operand can only be an XMM register. In case of single precision floating point operations, only the lowest 32-bits of the `dest` XMM register are modified and the remaining 96 bits remain unmodified. In case of double precision floating point operations, the lowest 64-bits are modified and the upper 64-bits remain unmodified.

The floating point operations performed by these instructions are addition, subtraction, multiplication and division of two floating point numbers, finding minimum and maximum of the two floating point numbers, finding square root of a floating point number, finding reciprocal of a number and finding reciprocal of square root of a number.

Some examples to show the effectiveness of these instructions are given in section 10.3.

10.2.6 SIMD control instructions

There are a few control instructions that need to be understood for operations between SIMD environment, x87 FPU environment and general purpose environment. As mentioned earlier, x87 FPU registers and MMX registers are aliased to each other. In the MMX instruction set, registers are accessed directly while in x87 FPU instruction set registers are accessed using register stack. In order to move between XMM environment and x87 FPU environment, care must be taken to ensure that programs work correctly.

MMX instruction set provides an instruction called `emms` to clear the MMX state and prepare for a default x87 FPU state. The following is the syntax of this instruction.

`emms` — MMX

This instruction marks all x87 FPU registers to be empty by putting 0xFFFF in the x87 Tag register.

In addition to using this instruction, the code may also use `finit` or `fninit` instructions in the x87 FPU instruction set to initialize the state of the x87 FPU.

In addition, all floating point operations in the SSE and SSE2 instruction sets modify the floating point flags in the `mxcsr` register. Con-

tent of this register can be set or read using the following two instructions.

```
ldmxcsr MemVar  —  SSE
stmxcsr MemVar  —  SSE
```

The `ldmxcsr` instruction reads a four byte memory location specified by `MemVar` and stores the contents in `mxcsr` register. The `stmxcsr` instruction saves contents of the `mxcsr` register in a four byte memory location.

10.3 Use of SIMD instruction sets

We have seen several examples of codes to use instructions in the SIMD instruction sets. We shall see some more examples in this section.

We start with the following computation.

$$a_i \leftarrow \left\lfloor \sqrt{|a_i|} \right\rfloor \quad \forall\, i = 0, 32$$

In this computation, a is an array of 32 integers. All integers in this array are replaced by the integer square root of their absolute value.

As usual, we shall use the XMM environment for this computation. We shall load four integers at a time in an XMM register. The negative integers will be converted to positive value by taking 2's complement. The 2's complement is evaluated by first inverting all bits and then adding 1. In order to invert all bits, we shall perform bitwise XOR will a mask that is all 1s. This mask can be obtained by using arithmetic shift instruction.

The following is the Assembly language code for this.

```
1  .globl IntSquareRoot
2  IntSquareRoot:
3      // Address of the array is at 4(%esp) when called.
4      mov  4(%esp), %esi
5      mov  $8, %ecx // 8 Iterations in all
6  IntSq0:
7      movdqu (%esi), %xmm0 // Load four integers
8      movdqu %xmm0, %xmm1  // Save in xmm1 as well
9      psrad  $31, %xmm0    // xmm0 will contain the mask
10     pxor   %xmm0, %xmm1  // Bitwise XOR in xmm1
11     // By logical right shifting bit mask in xmm0
12     // We can have a 1 in place of those integers
13     // which are negative.
14     psrld  $31, %xmm0    // xmm0 longs = 1 or 0.
15     paddd  %xmm0, %xmm1  // xmm1=absolute values of int
```

```
16      cvtdq2ps %xmm1, %xmm1 // xmm1=float numbers
17      sqrtps %xmm1, %xmm1 // xmm1 = sqrt(abs(int))
18      cvtps2dq %xmm1, %xmm1 // Convert to integer
19      movdqu %xmm1, (%esi)
20      add  $16, %esi // esi = adrress of next set of ints
21      loop IntSq0
22      ret
```

The C prototype for this function would be the following.

void IntSquareRoot(**int** a[32]);

The function `IntSquareRoot` takes the address of the array and copies it in register `esi` in line 4. At each iteration, it computes the square root of four integers and hence a total of 8 iterations are carried using `loop` instruction. The loop counter is loaded with 8 in line 5. At each iteration, four integers are loaded in registers `xmm0` and `xmm1` in lines 7 and 8. Packed integers in register `xmm1` are right shifted using arithmetic shifts by 31 locations in line 9. This ensures that just the sign bit is repeated in all bits of the integers. Thus the integer becomes 0 if originally it was positive or all 1s if originally it was negative. Negative numbers are negated to get the absolute values of the integers. In order to do so, all bits of the negative numbers are inverted and one is added. We use the mask in `xmm1` to achieve this. Bits of the negative numbers are inverted while making no changes to the positive numbers by use of `pxor` instruction in line 10. The bit mask in `xmm1` is right shifted by 31 locations (this time with unsigned shifts) to get a value of 1 or 0 in 32-bit integers. The integers become 1 if they were negative originally and 0 if they were positive originally. These integers are added to the inverted integers in line 15. This then converts all integers to their absolute values.

The integers are then converted to floating point numbers (line 16); square root is taken (line 17) and then converted back to integers (line 18). Converted integers are stored back in the memory. At each iteration, 32 is added to register `esi` which then points to the next set of 4 integers.

Let's consider another example where we need to find maximum of several floating point numbers. We shall implement the computation of the following expression.

$$\max_{i=0}^{32} s_i$$

In this expression, each s_i is a single precision floating point number. We shall implement this computation using a function in Assembly language whose C prototype is the following.

float getMaxFloat(**float** s[32]);

In the Assembly language program, we implement finding of the maximum number using the following algorithm. The 32 numbers in array s are organized as 8 sets of numbers, each set containing four single precision floating point numbers. Let us call these sets as x_0 to x_7. A set x_i comprises of floating point numbers s_{4i}, s_{4i+1}, s_{4i+2} and s_{4i+3}. In an XMM register four single precision floating point numbers in a set can be loaded. We load the first set x_0 in register xmm0. We compute the maximum of numbers in sets x_0 and x_1 and replace xmm0 with four numbers where each number is the maximum of the corresponding numbers in sets x_0 and x_1. We repeat this for all other sets, x_2 to x_7. At the end, register xmm0 shall contain four single precision numbers each of which is a maximum of the corresponding numbers in all eight sets. If we call these numbers as a_0, a_1, a_2 and a_3, then a_0 is the maximum of $s_{4i} \forall i = 0, 7$. Similarly a_1 is the maximum of $s_{4i+1} \forall i = 0, 7$, a_2 is the maximum of $s_{4i+2} foralli = 0, 7$ and a_3 is the maximum of $s_{4i+3} \forall i = 0, 7$.

We now need to find the maximum of a_0 to a_3 stored as 4-packed single precision numbers in xmm0 and return the result. For C interface, the floating point results are returned on top of the x87 register stack. We shall therefore move from the SIMD environment to x87 environment before returning the result.

Following is the Assembly language code for function getMaxFloat.

```
1  .globl getMaxFloat
2  getMaxFloat:
3      // Starting Address of the array s[] is at 4(%esp).
4      mov  4(%esp), %esi // Array address in %esi
5      movups  (%esi), %xmm0 // Load first set
6      addl $16, %esi // esi = address of the next set
7      mov  $7, %ecx // 7 sets to compare from.
8  getMaxSet:
9      maxps (%esi), %xmm0
10     addl $16, %esi // esi = adderss of the next set
11     loop getMaxSet
12 // At this point xmm0 = set wise maximum
13 // The numbers in xmm0 are (a3, a2, a1, a0)
14 // We copy in xmm1 as (a1, a0, a3, a2)
15 // Find the maximum again
16     pshufd $0x4E, %xmm0, %xmm1
17     maxps %xmm1, %xmm0
18 // At this point a0 is max of all even indexed numbers
19 // of array s. a1 is the max of all odd indexed numbers.
20     pshufd $0xB1, %xmm0, %xmm1
21     maxps %xmm1, %xmm0
22 // At this point a0 is the max of all the numbers
23 // We need to move it to x87 FPU register top
```

```
24 // First create a local space on the stack to save it
25     subl $4, %esp
26     movss %xmm0, (%esp)
27     fninit  // Initialize x87 state
28     flds (%esp) // Load on stack top
29     add  $4, %esp // Free up the space for local storage
30     ret
```

The code proceeds as follows. First x_0 set of numbers is loaded in `xmm0` in line 5. Register `xmm0` is then compared with numbers in sets x_1 to x_7 each time replacing it with the maximum using instructions in lines 8 to 11. At this point we have a_0 to a_3 in register `xmm0` arranged as a_3, a_2, a_1 and a_0. Register `xmm0` is shuffled in line 16 to get a_1, a_0, a_3 and a_2. Results are put in register `xmm1`. A maximum is computed (line 17) between `xmm1` and `xmm0` and the result is taken in register `xmm0`. At this time, register `xmm0` contains the maximum of a_0 and a_2 in bit locations 0 to 31 and maximum of a_1 and a_3 in bit locations 32 to 63. These are again shuffled, and maximum computed to get the maximum of all floats in bits 0 to 31 of register `xmm0` (lines 20 and 21).

As the C run time environment expects the results to be on top of x87 register stack, we need to carry it there. In order to do so, 4 byte local space is created in line 25 and one single precision number is stored in this local space in line 26. Finally this number is loaded on x87 register stack in line 28. Local space is recovered and the control is returned back to the called with result on top of the x87 FPU register stack.

EXERCISES:

10.7 Implement the following functionality of C code in Assembly language using SIMD instructions.

```
void add_array_float(float p[64], float q[64]) {
    int i;

    for(i=0; i<64; i++) {
        p[i] += q[i];
    }
}
```

10.8 Implement the following functionality of C code in Assembly language using SIMD instructions.

```
void Op_array_float(float p[64], float q[64]) {
    int i;

    for(i=0; i<64; i++) {
        p[i] = (p[i] + 2) * (q[i] * 3);
```

```
    }
}
```

10.9 Implement the following functionality of C code in Assembly language using SIMD instructions.

```
void Scale_array_float(float p[64], int r) {
    int i;

    for(i=0; i<64; i++) {
        p[i] = r * fabs(p[i])
    }
}
```

10.10 Implement the following functionality of C code in Assembly language using SIMD instructions.

```
void Scale_Reversed_array_float(float p[64], float q[64], int r) {
    int i;

    for(i=0; i<64; i++) {
        p[i] = q[i] * p[63−i];
    }
    for(i=0; i<64; i++) {
        p[i] = p[i] * r;
    }
}
```

10.11 Write an Assembly language function to compute the average of 64 elements stored in an array of double precision floating point numbers. The prototype of the function in C would be the following.

```
double Average(double d[64]);
```

The C environment expects the return value of the function on top of the x87 register stack. Therefore the program must switch from SIMD to x87 FPU environment before returning the result.

10.12 Given the marks of students in several assignments, the task of this exercise is to compute the average of the class for each assignment. Assume that there are eight assignments and 64 students in the class. The marks are available in the following C data type.

```
typedef struct {
    float assignment[8];
} Record;
```

C prototype of the function to be implemented in Assembly language is the following.

void ComputeAverage(Record student[64], Record *avg);

The return value of all eight averages is provided in memory location `avg` (address of the location is passed as the second argument to this function.)

Hint: The memory allocation of array `student` will be first all eight marks for `student`[0], then all eight marks for `student`[1] and so on. Using SSE instructions, two XMM registers can be used to load all the eight marks of first student. Both of these registers can then be added with the marks of remaining 63 students. Later a division by 64 would provide the desired result.

10.13 In this exercise we shall implement a simple digital signal processing application of low pass filtering. Let's assume that a system provides continuous 16-bit integer samples as the digital form of the waveform. Let this sequence be called x_n, x_{n-1}, ..., x_j, ..., x_2, x_1, x_0. The filtered output sequence is denoted by y_i for i being between n and 0. The filtered output is defined as the following.

$$\begin{aligned} y_i &= x_i - x_{i-1} \quad i > 0 \\ y_0 &= x_0 \end{aligned}$$

Write an Assembly language function that implements this filtering. Assume the following C function prototype for the Assembly language function that you have to implement.

void LowPassFilter(**short** in[64], **short** out[64]);

Hint: Load eight x_i samples into an XMM register. Use a shuffle operation or a shift operation to align x_i and x_{i-1} for all i. Use subtraction and save. This way, in each iteration, seven value of y_i will be computed. Increment the pointers by 7 locations to repeat this operation. Care must be taken at the last iteration to save elements without overflowing the memory bounds.

Chapter 11

Assembler Directives and Macros

GNU assembler or `gas` that is used on GNU/Linux as a standard assembler `as` is a very powerful and flexible assembler. The `gas` assembler is available for several processors and several output formats of the binary object file. On most GNU/Linux installations for IA32 processors, the assembler generates an object file in ELF format.

The assembler may be configured in several ways to generate object code. These methods include the following.

1. Command line options for `gas` as discussed in Appendix D.
2. Instructions embedded in the Assembly language program known as Assembler directives or pseudo-ops.

In this chapter, we look at the assembler directives. The assembler directives are divided into several groups based on the functionality.

1. Link control directives for organization of data and code in different sections and units each with its own attributes such as "read-only", "initialized" etc. These assembler directives are used to produce information in the generated object file so as to instruct the linker to link such sections in a certain way. The operating system would then honor such attributes of the sections while loading them in the memory and use protection mechanisms to ensure attributes such as "read-only".
2. Directives for specifying attributes for labels so as to help the process of linking.
3. Code and data generation directives. These directives instruct the assembler on the way the code is generated.

4. Directives for conditional assembly and directives for inclusion of files.

5. Directives for defining macros and using them in the program. Use of macros helps a programmer defines a sequence of Assembly instructions as a single unit and later using this to expand the instructions.

6. Directives for controlling the process of assembly and generation of list file. The list file exhibits the way the code is generated and is helpful to the programmers in certain ways.

All assembler directives names begin with a period ('.') as shown in various programs in this book such as in figure 1.1. The rest of the name is usually in small case letters.

11.1 Link control directives

The linker supports data and code organization in chunks of memory called sections. Each of these sections are usually loaded independent of each other by the operating system. The usual sections supported by the linker are the following.

Data Sections: Data sections are sections used for defining data with initial values.

Text Sections: Text sections are used to define the sections of memory that contains programs. These sections are therefore similar to data sections with initial values.

The `bss` section: The `bss` section is used to define uninitialized data. These sections are not initialized and therefore the executable file does not include any initial image for these sections. While loading a program, the operating system sets all bytes in the `bss` section to zeros.

Each of the sections may have additional attributes such as read-write, read-only etc. For example, a usual text section would be marked shared and read-only. The C programs usually generate two kinds of data sections call `rodata` and `data` section. The `rodata` section is used to keep constants such as constant strings, and is marked read-only. The `data` section is usually marked read-write.

The `gas` assembles its programs usally into two sections – text and data. Within a section, data may be grouped separately by using a notion of subsections. All subsections within a section are grouped together as part of a single section when an object files is generated.

Therefore in an object file the subsections lose their independent identity with the entire section being treated as a single unit.

As an example, one may want to group all constants in a text section separate from the program bytes. The programmer may use two subsections (say 1 and 0) for storing constants and programs. Later all subsections will be combined in numeric order and put into the final section. Use of subsections is optional and when not specified, a default subsection 0 is used.

Size of each subsection is made a multiple of four bytes by adding one, two or three bytes containing a zero.

While assembling a program, the `gas` maintains a location counter for each subsection and uses this to emit code or data bytes. Each time a byte is produced, the location counter is incremented by 1. When a label definition is encountered, it is set to the value of the location counter.

`.data` *subsection*

The `.data` assembler directive causes `gas` to assemble the subsequent statements in specified data subsection. If subsection is not specified, it defaults to zeroth subsection.

`.text` *subsection*

The `.text` assembler directive causes `gas` to assemble the subsequent statements in specified text subsection. If subsection is omitted, it defaults to zeroth text subsection.

`.section` *name, "flags", @type*

The `.section` assembler directive defines a section by the given name, if it is not defined already. In addition to defining a new section, it causes the `gas` to assemble the subsequent statements in the named section. While defining a section, flags and type arguments can specify attributes of the section. When the `.section` directive is encountered the first time, a new section is defined with the given values of *flags* and *type* arguments. In the subsequent usage, *flags* and *type* arguments are not required. In general, *flags* and `type` arguments are optional and when not specified the first time, they have a default value.

The *flags* argument is a string within double quotes and may contain any combination of the following characters.

flag	Meaning
`a`	section is allocatable
`w`	section is writable
`x`	section is executable

The `type` argument may contain one of the following values.

`@progbits`: Section should be initialized with the data in the object file.

`@nobits`: Section is not initialized with the data in the object file.

If no flags and types are specified, the default values depend upon the section name. For an unrecognized section name, the default values are: Not alloted in memory, Not writable, Not executable (*i.e.* no flags), and that the section contains data. Such sections are normally used for inserting debugging and other control information in the executable file.

For example, consider the following assembly code fragment.

```
.section .text
item1:  .int 1234
.section .data
item2:  .int 5678
.section .bss
item3:  .int 90
.data 0
item2a: .float 12.625
.data 1
item2b: .ascii "hello world"
```

When assembled, the `gas` will define five local symbols (the symbol definition for local symbols is not stored in the generated object file) as `item1`, `item2` etc. The symbol `item1` is defined in section `.text` and it represents the name of a 32-bit memory location that is initialized to an integer 1234. The `.text` section typically contains code and read-only data. Thus the programs may not change the value of memory location `item1`.

The second symbol `item2` is defined in section `.data` which is a read-write initialized data section. The symbol `item2` represents the name of a 32-bit memory location that would be initialized to an integer 5678 in the beginning. During the execution of the program, instructions may change the value of memory location `item2`.

The symbol `item3` is defined in section `.bss` which is a read-write uninitialized data section. When the program is loaded in the memory, locations corresponding to the `.bss` section are set to 0. An initial value of 90 is ignored as this section can not have any initial values.

The symbols `item2a` and `item2b` are defined in section `.data`. Symbol `item2a` is in subsection 0 of `.data` section and therefore comes right after the definition of `item2`. Symbol `item2b` is in subsection 1 of the `.data` section and will come after all subsection 0 definitions are assembled. Both of these symbold represent read-writable memory areas that are initialized in the beginning.

11.2 Directives for label attribute specifications

`.equ` *symbol, expression*

`.set` *symbol, expression*

The `.equ` and `.set` assembler directives have the same functionality and may be used interchangeably. The same effect may also be achieved using '*symbol=expression*' assembly language statement.

These directives define a symbol (for subsequent use in the program) to contain a value of the expression. The expression is evaluated at the time of assembly and therefore should not use any run-time variables or registers.

A symbol may be defined several times and once defined, its definition will be effective till a new definition of the same symbol is encountered in the program.

Definitions of global symbols (declared using `.globl` directive) are emitted in the generated object file. In case a global symbol is defined multiple times, the last value defined in the source is used in the generated object file. The definitions of such symbols are then available to the linker which links symbols across multiple object files.

`.equiv` *symbol, expression*

The `.equiv` directive is similar to `.equ` or `.set` assembler directive except that it may be used to define a symbol only once. The assembler will produce an error message if symbol is already defined.

`.comm` *symbol, length, alignment*

The `.comm` assembler directive declares a common symbol named *symbol* possibly in the `.bss` section. A common symbol is allocated only once in the entire program even if it is declared in many source files. If the symbol is actually allocated in any modules by one of directives, the linker `ld` allocates length bytes in uninitialized space. If multiple modules define the same symbol as common with different lengths, the linker choses the maximum of all lengths.

The `.comm` directive may take an optional third argument *alignment*. This parameter specifies the desired alignment of the symbol in units of bytes. More on alignment is explained later while describing the `.align` directive.

`.lcomm` *symbol, length*

The `.lcomm` directive defines a local common `symbol` of `length` bytes. The addresses are allocated in the `.bss` section and are initialized to 0 when the OS loader loads the program. The `symbol` may also be declared global using `.globl` directive and in that case, the

definition will be put into the generated object file. A symbol defined without `.globl` assembler directive is normally not visible to `ld`.

`.global` *symbol*

`.globl` *symbol*

The `.global` or `.globl` assembler directives make the `symbol` visible to `ld` by declaring it to be global. Definition of a symbol defined global using this directive is inserted in the generated object file and is visible to other modules. Thus a program in other files may change the values of a variable that is not declared in that file but is declared `.globl` in other files.

We have used this assembler directive in most of our programs. The symbol `_start` must be declared global so that the linker may know about it and may set this as the start of the program. Any Assembly language function must be declared global if it is to be called from programs in other files including those in C.

Programming languages such as C declare all functions (except the ones that are declared `static`) as global and such functions may be called by an Assembly language program. In such cases, the linker links the definition of such symbols with the usage in the object files generated by the assembler.

11.3 Directives to control code and data generation

Changing the location counter

`.org` *new-location-counter, fill*

The `.org` assembler directive causes the location counter of the current section to *new-location-counter*. The `new-location-counter` can be specified as an absolute expression (such as 0x1500) or an expression that evaluates to a value within the current section (such as `_start+100`). The `new-location-counter` is relative to the start of current section and not to the start of the current subsection.

The `new-location-counter` can only refer to symbols or labels that have been declared prior to the use of `.org` directive. Symbols or labels declared later are treated as undefined.

the `.org` directive may only advance the location counter, or leave it unchanged. It can not be used to move the location counter backwards. As the location counter is advanced, the intervening bytes are filled with value given as *fill*. The *fill* may be an expression derived from values known as the time of Assembly. The *fill* argument is optional and defaults to 0 when not specified.

Code generation control

`.code16`

The `gas` supports writing code to run in real mode or in 16-bit protected mode. When `.code16` directive is in effect, 16-bit code is generated for the instructions. We have not discussed this mode of programming in this book.

`.code32`

This directive causes `gas` to generate 32-bit code for the instructions and is the default code generation mode for `gas`. The `.code32` directive is rarely used as it is the default in effect. It may be needed to switch back from 16-bit code generation mode to 32-bit code generation mode when the program includes both types of codes.

`.arch` *cputype*

The `.arch` directive specifies the IA32 CPU type to the assembler. Many CPUs do not support all instructions and instruction sets. For example, the i486 CPU does not support MMX, SSE or SSE2 instructions. When an instruction is encountered during the assembly that is not supported for the CPU type defined by `.arch` directive, `gas` issues a warning. The choices for cputype are specific to the version of the GNU binutils that includes the `gas`. The following CPU types are understood by the `gas` in `binutils` version 2.16.

`i8086`: Intel 8086/8088 CPU

`i186`: Intel 80186 CPU

`i286`: Intel 80286 CPU

`i386`: Intel 80386 CPU

`i486`: Intel 80486 CPU

`i586`: Intel Pentium CPU

`i686`: Intel Pentium pro CPU

`pentium`: Intel Pentium CPU

`pentiumpro`: Intel Pentium pro CPU

`pentiumii`: Intel Pentium II (Pentium Pro + MMX instruction set)

`pentiumiii`: Intel Pentium III (Pentium II + MMX2 + SSE instruction sets)

`pentium4`: Intel Pentium IV (Pentium III + SSE2)

`prescott`: Intel Prescott CPU (Pentium IV + SSE3)

`k6`: AMD K6 processor. (Pentium + MMX + K6 extensions)

`k6_2`: AMD K6 with 3DNow!™

`athlon`: AMD K6 with Athlon, MMX2, 3DNow!™ and 3DNow!™ extensions

`sledgehammer`: AMD Athlon with Sledgehammer extensions, SSE and SSE2

`.mmx`: MMX instruction set

`.sse`: MMX, MMX2 and SSE instruction sets

`.sse2`: MMX, MMX2, SSE and SSE2 instruction sets

`.3dnow`: MMX and 3DNow!™ instruction sets

`.3dnowa`: MMX, MMX2, 3DNow!™ and 3DNow!™ extensions

`.padlock`: VIA PadLock instruction set for cryptography application

Repetition of statements

`.rept` count

The `.rept` directive is used to repeat a number of Assembly language statements between `.rept` and `.endr` directives. The statements are repeated *count* number of times.

For example, the following two codes are equivalent. The first code repeats lines between `.rept` and `.endr` four times.

```
.rept   4           .ascii  "-"
.ascii  "-"         .ascii  "-"
.endr               .ascii  "-"
                    .ascii  "-"
```

`.irp` symbol, values

The `.irp` directive is used to assemble a sequence of statements between `.irp` and `.endr` directives a number of times assigning different *values* to *symbol*. Each time the sequence of statements are assembled, the *symbol* is set to a value from *values* argument. The *symbol* is referred within the sequence of statements by using a '\' before the *symbol*.

As an example, in the following code the sequence of Assembly statements on the left is expanded to the Assembly statements on the right.

```
.irp    ind,1,2,3,4         a1: .int 1
a\ind: .int    \ind         a2: .int 2
.endr                       a3: .int 3
                            a4: .int 4
```

`.irpc` *symbol,valueset*

The `.irpc` directive provides a similar functionality as of the `.irp` directive except that the values are set of characters. Each time the statements are expanded, the *symbol* is set to a character from the *valueset*.

For example, the following two codes are equivalent.

```
.irp    ind,1,2,3,4          .irpc   ind,1234
a\ind: .int    \ind          a\ind: .int \ind
.endr                        .endr
```

Defining data

`.byte` *expressions*

The `.byte` directive takes zero or more expressions, separated by commas. Each expression is evaluated to one byte value and stored in the current section, each time incrementing the location counter by 1. An expression that evaluates to a negative number is emitted in 2's complement representation of corresponding absolute value.

`.hword` *expressions*

`.short` *expressions*

`.word` *expressions*

The `.hword`, `.short` and `.word` directives have same functionality in `gas` for IA32 processors. These directives expect zero or more expressions, and emit a 16 bit number for each. While evaluating these expressions, the lower order byte is stored first and higher order byte is stored next. Each time an expression is emitted in the current section, the location counter is incremented by 2. An expression that evaluates to a negative number is emitted in 2's complement representation of corresponding absolute value.

`.int` *expressions*

`.long` *expressions*

The `.int` and `.long` directives have same functionality in `gas`. Both of these directives expect zero or more expressions, and emit a 32-bit (4 byte) number for each expression. The lowest order byte is stored first and the highest order byte is stored the last. Each time an expression is emitted in a section, the location counter is incremented by 4.

`.quad` *expressions*

The `.quad` directive expects zero or more expressions and emits a 64-bit (8 byte) integer for each expression. The location counter is incremented by 8 each time an expression is emitted. The least order

byte is emitted first while the highest order byte is emitted the last to be consistent with the storage order of IA32 processors.

`.octa` *expressions*

The `.octa` directive is functionally similar to the `.quad` directive except that each expression is evaluated to a 128-bit (16 byte) integer and the location counter is incremented by 16 each time an expression is emitted in the current location.

`.float` *expressions*

`.single` *expressions*

`.ffloat` *expressions*

The `.float`, `.single` and `.ffloat` directives implement the same functionality. These directives take zero or more real number expressions and emit them in 32-bit (4 byte) IEEE754 single precision floating point number format. Each time an expression is emitted in the current section, location counter is incremented by 4. The least order byte is stored at the lowest address while the highest order byte is stored at the highest address.

`.double` *expressions*

`.dfloat` *expressions*

The `.double` and `.dfloat` directives implement the same functionality. These directives convert each real number expression to IEEE754 double precision floating point number format and emit them in the current section, each time incrementing the location counter by 8.

`.tfloat` *expressions*

The `.tfloat` directive converts its arguments to double extended-precision floating point number format (80-bits) and emits them in the current section, incrementing the location counter by 10 each time an expression is emitted.

`.ascii` *"string"...*

The `.ascii` directive takes zero or more string literals (enclosed within double quotes) separated by commas. It assembles each string into consecutive addresses in the current section. The strings are not treated as strings of C programs. In particular no '`\0`' character is appended at the end of the string. The strings can have the following special sequences to represent certain characters within the string.

`\b`	: Backspace character
`\f`	: Form Feed character
`\n`	: New Line character
`\r`	: Carriage Return character
`\t`	: Horizontal Tab character
`\`$d_1d_2d_3$	: Character with given octal code For example, `\012` is New Line character
`\\`	: Character '\'
`\"`	: Double quote character

.asciz *"string"...*

.string *"string"...*

The `.asciz` and `.srting` directives are functionally similar and are almost like `.ascii` directive. These directives assemble their string arguments with a NULL character (ASCII code = 0) appended at the end. Thus each string is followed by a byte containing a zero.

Alignment within sections

.align *alignment, fill-value, max-skip*

.balign *alignment, fill-value, max-skip*

The `.align` and `.balign` directives advance the location counter in the current subsection enough such that it becomes a multiple of *alignment*. As the location counter is advanced, the intervening bytes are filled with the *fill-value*. The *fill-value* is an optional argument and defaults to 0 in data sections and `nop` instruction in text sections. The *max-skip* is also an optional argument. If present, it defines the maximum number of bytes that should be skipped by this alignment directive. If achieving the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all.

Alignment is a commonly used mechanism in IA32 processors to improve efficiency of bus accesses. For example, if a 32-bit integer is aligned to four byte boundaries in memory, entire integer can be accessed in a single memory cycle. An unaligned integer would require two memory cycles.

The programs are typically fetched and stored in instruction cache. If a function is aligned to the cache block boundary (typically 32 bytes), it will lead to an efficient use of cache. For example let's consider a function whose code fits in 30 bytes and therefore may be accommodated in a single cache block. If the function is not aligned to a cache block boundary, it may occupy two cache blocks, both of which will remain partially filled. Since the cache blocks are filled entirely, it would result in reading 64 bytes from memory instead of 30 bytes.

As an example, consider the following code fragment.

```
1        .data
2        .org 13, 0
3        .align 8, 0
4    x:  .quad 2000, 0x4433
```

The `.data` directive in the first line choses the data section (and subsection 0). The location counter is set to 13 filling up the first 13 bytes to 0 by the `.org` directive in line 2. `.align` directive in line 3 sets the location counter to 16 (to make it up a multiple of 8) filling up the next three bytes to 0. Finally a symbol `x` is defined to represent location counter being 16. In the next 16 bytes, from location 16 to 31, two 64-bit integers are assembled having the values 2000 and 0x4433. At the end, the location counter of subsection 0 in `.data` section gets set to 32.

`.balignw` *alignment, fill-value, max-skip*

`.balignl` *alignment, fill-value, max-skip*

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a 16-bit (2 byte) word value. The `.balignl` directive treats the fill pattern as a 32-bit (4 byte) longword value. During the fill operation, the least order byte is stored first and the highest order byte is stored the last. If size of the space created by advancing the location counter is not a multiple of two (in `.balignw`) or four (in `.balignl`), the padding pattern for these bytes is undefined.

As an example, consider the following code fragment.

```
.data
.org 12, 0
.balignw 8, 0x1100
```

The first two lines of this code choose subsection 0 in `.data` section and set the location counter to 12 filling first 12 bytes to 0. The last line would align the location counter to a multiple of 8. It therefore sets the location counter to 16. Intervening four bytes are set to 0x00, 0x11, 0x00 and ox11. Thus this code fragment sets the `.data` section to contain first 12 bytes to contain 0 and the next 4 bytes to contain two words each having a value 0x1100.

A relevant part of the `gas` generated assembly program listing for these lines of code is shown below.

```
1                            .data
2 0000 00000000              .org 12, 0
2      00000000
2      00000000
3 000c 00110011              .balignw 8, 0x1100
```

.p2align *lg2alignment, fill-value, max-skip*

.p2alignw *lg2alignment, fill-value, max-skip*

.p2alignl *lg2alignment, fill-value, max-skip*

The `.p2align`, `p2alignw` and `p2alignl` assembler directives are similar to the corresponding `.balign..` directives. These directives are called "power-2" alignment directives and take the first argument *lg2alignment* as the number of low-order zero bits the location counter must have after advancement.

For example the statement `.p2align 3` advances location counter until it is a multiple of 8 (2^3). This will ensure that the last three bits of the location counter are set to 0.

The *max-skip* arguments of these directives have the same meaning as of the other alignment directives. These arguments set the limit on the number of bytes that can be skipped with these directives. If the location counter would need an advancing of more than the *max-skip*, alignment is not carried out.

.skip *count, fill*

.space *count, fill*

The `.skip` and `.space` directives advance the location counter in the current subsection by *count*. The space so created is filled with bytes give by *fill* argument. If the *fill* argument is omitted, it is assumed to be zero.

11.4 Directives for Conditional assembly

There are two ways the program fragments may be assembled conditionally. He have used the C preprocessor constructs in assembly program written in this book. The C preprocessor supports definition of symbolic constants (such as `STDOUT` in program of figure 1.1), conditional evaluation using constructs such as `#if`, `#ifdef`, `#ifndef` and inclusion of files using `#include` directives.

In addition to these, the `gas` also supports conditional assembly using assembler directives many of which are similar to the C preprocessor statements but operate within the `gas` environment. These conditional expressions can use the symbols defined in the program using `.set`, `.equ` and `.equiv` directives.

There are two forms of conditional structures in `gas`. The first form encloses a code block within `.if` and `.endif` directives as follows.

`.if` *cond-expr*
    code block that will be assembled if *cond-expr* evaluates to true.
`.endif`

The second form defines two code blocks. Only one of these is included depending upon the outcome of the evaluation of conditional expression.

The conditional constructs of `gas` include the following directives.

`.if` *cond-expr*
code block that will be assembled if *cond-expr* evaluates to true.
`.else`
code block that will be assembled if *cond-expr* evaluates to false.
`.endif`

`.if` *cond-expression*

The `.if` directive marks the beginning of a code block which is assembled if the conditional expression *cond-expression* evaluates to a non-zero value. The end of the conditional section of code is marked by `.endif` or `.else`.

`.ifdef` *symbol*

A variant of the `.if` directive which assembles the code block if the specified *symbol* has been defined.

`.ifndef` *symbol*

`.ifnotdef` *symbol*

Variants of the `.if` directive which assemble the code block only when the specified *symbol* is not defined.

`.else`

The `.else` directive marks the end of code block corresponding to `.if` directive. It also marks the beginning of a code block to be assembled if the condition for the preceding `.if` was false.

`.endif`

The `.endif` directive marks the end of the conditional blocks of code.

`.include` *"file"*

The `.include` directive provides a way to include a *file* at specified points in the source program. The code from file is assembled as if it followed the point where `.include` directive was used. After the end of the included file is reached, assembly of the original file continues.

`.err`

The `.err` directive is used normally along with the conditional assembly. When the `gas` encounters an `.err` directive it prints an error message and does not generate an object file.

The common use of this directive is to use it with conditional assembly. For example, it can be used to flag an error when a needed symbol is undefined as shown in the following code fragment.

```
.data
intarr:
.ifndef ARRSIZE
  .err
.else
  .rept ARRSIZE
    .int 0
  .endr
.endif
```

This code fragment defines an array of integers starting from memory location `intarr`. The number of items in the array is given by a symbol `ARRSIZE`. If this symbol is undefined in the program, an error message is printed and no object code is generated. Otherwise, an integer array of `ARRSIZE` is generated from `.rept` assembler directive in the else block of the conditional statement.

11.5 Directives for assembly listing

The `gas` can be configured through a command line argument to generate the listing of assembly language programs. The list files provide programmers an insight into the process of assembly. The programmers can monitor the code that is generated including aspects such as those code block which are dropped due to conditional assembly. The format of the list file can be controlled in several ways using the assembler directives.

.eject

Force a page break at this point in the assembly listing.

.list

.nolist

These two directives control whether or not the assembly listing is to be generated for the code following the assembler directive.

These two directives maintain an internal listing control counter. This counter is normally set to 0 in the beginning. The `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled and can be enabled by a command line switch as described in Appendix D. When the listings are enabled by the command line switch, the initial value of the listing counter is set to one.

.psize *lines, columns*

The `.psize` directive may be used to define the format of the paper on which the list file may be printed. This provides the number of

lines, and optionally the number of columns, to use for each page in the generated listings.

The default values of *lines* and *columns* are 60 and 200 respectively.

.title *"title-heading"*

Each page in the listing for an assembly language program starts with a line that provides the name of the source program file. Subsequent to this, an optional title and an optional subtitle may be printed on each page on second and third lines respectively.

The `.title` directive defines the second line of each page in the assembly listing.

The `.title` affects subsequent pages, as well as the current page.

.sbttl *"subtitle-heading"*

The `.sbttl` directive defines the subtitle that is printed on the third line of each page in the assembly listings.

The `.sbttl` affects subsequent pages, as well as the current page.

A common way to organize the title and subtitle would be to give title heading the name of the project and the subtitle heading the name of the function.

11.6 Directives for macros

The macros are ways to implement a program with repeated program structures in a concise manner. The use of macros require less of typing and debugging when sequences of program statements have a repeated structure.

Macros are very much similar to an in-line function in a high level language. When `gas` encounters a macro definition, it remembers it. When it encounters an Assembly language statement that uses a previously defined macro, the macro definition is substituted during the assembly process.

The assembler directive `.macro` is used to initiate the definition of a macro while the `.endm` is used to end the macro definition.

For example, the following macro can be used to exchange 64-bit values in register `eax:ebx` and `ecx:edx`.

```
.macro exch64
    push %esi      /* Save esi */
    mov %eax, %esi /* Use esi as temp reg */
    mov %ecx, %eax /* for exchanging eax */
    mov %esi, %eax /* and ecx */
    mov %ebx, %esi /* Exchange ebx and edx */
    mov %edx, %ebx
    mov %esi, %ebx
```

```
    pop %esi
.endm
```

Later `exch64` macro can be used like an instruction in the program to exchange the values of two 64-bit numbers stored in `eax:ebx` and `ecx:edx`.

Macros can have parameters to expand in a different way each time they are used. A general way to define a macro is the following.

.macro *macro-name macro-args*

This defines a macro by name *macro-name*. There can be 0 or more arguments to a macro definition. These arguments may also be defined to have a default value and may be used in the body of the macro.

For example, the following definition specifies a macro `intarray` that defines an array of integers initialized with a sequence of numbers. The size and name of the array are argument to the macro with a default value of the size as 8.

```
.macro intarray name, size=8
\name:
    .set val, 0
    .rept \size
        .long val
        .set val, val+1
    .endr
.endm
```

As shown in the body of the macro, parameters are referred by a leading '\'. The following is an example of using macros. The second line defines an array of four integers called `Arr1` while the third line defines an array of eight integers. The code fragment also defines a byte memory location that contains the size of the first array in bytes.

```
    .data
    intarray Arr1, 4
    /* Size of Arr1 array */
    sz1: .byte . - Arr1
    .align 4, 0
    intarray Arr2
    /* Size of Arr2 array */
    sz2: .byte . - Arr2
```

Corresponding list file contents resulting from assembling this program are the following.

```
  1                .macro intarray name, size=8
  2                \name:
  3                    .set val, 0
  4                    .rept \size
  5                        .long val
  6                        .set val, val+1
  7                    .endr
  8                .endm
  9
 10                    .data
 11 0000 00000000     intarray Arr1, 4
 11      01000000 02000000
 11      03000000
 12                    /* Size of Arr1 array */
 13 0010 10           sz1: .byte . - Arr1
 14 0011 000000       .align 4, 0
 15 0014 00000000     intarray Arr2
 15      01000000 02000000
 15      03000000 04000000
 15      05000000 06000000
 15      07000000
 16                    /* Size of Arr2 array */
 17 0034 20           sz2: .byte . - Arr2
```

Notice a use of '.' in defining the values of sz1 and sz2. In gas, this special symbol denotes the value of current location counter in the current subsection. Thus the expression '.-Arr1' gives the difference between the current location counter and symbol Arr1. In our example, this expression evaluates to the size (in bytes) of Arr1.

The gas allows a recursive use of the macros. The following example shows the strengths of recursive definition combined with conditional assembly.

```
.macro  sum from=0, to=5
  .long   \from
  .if     \to-\from
     sum     "(\from+1)",\to
  .endif
.endm
```

With this definition, sum 0,3 expands to the following.

```
  .long   0
  .long   1
```

```
  .long   2
  .long   3
```

The default value of the macro parameters `from` and `to` are 0 and 5 respectively. Thus the use of this macro as '`sum`' results in the following expansion.

```
  .long   0
  .long   1
  .long   2
  .long   3
  .long   4
  .long   5
```

The `gas` maintains a counter of how many macro expansions have been carried out during the assembly process. This is available within a macro definition as a pseudo-variable called `\@`. This can be used to define labels or symbols in the macro definition, which when expanded, would not conflict with other expansions for the same macro.

For example, consider the following macro definition.

```
.macro scallreturn
      int $0x80
      cmp $0, %eax
      jnl noerror\@
      neg %eax
      mov %eax, errno
      mov $-1, %eax
   noerror\@:
.endm
```

This definition roughly implements the GNU libc behavior of handling return values from system calls. It makes a call to the system call handler and then compares the return value. If there had been an error, the error code is stored in an external variable called `errno` and then the register `eax` is set to -1.

Since the code needs to make a jump to a forward location, two different invocations of this macro will result in a conflict of the jump label. Use of '`\@`' solves this problem. Each time this macro is expanded, this pseudo variable will have a unique value thus resulting in non-conflicting labels.

`.endm`

The `.endm` directive marks the end of a macro definition.

`.exitm`

The `.exitm` directive can be used to exit the macro expansion when this statement is reached. This can be used along with conditional assembly mechanism to expand macros and exit out when the expansion is completed.

`.purgem` *macro-name*

The `.purgem` directive is used to delete definition of the given *macro-name*. When a macro is defined, it can be used only till a `.purgem` is encountered by the assembler. If the macro name is used again after the use of `.purgem`, it results in undefined macro error.

Appendix A

Number Representation System

A digital computer uses binary system. In this system there can only be two states represented by a single binary variable. For representation of numbers, these states are used to denote a digit of binary number system, also called a bit. A bit can have only two values 0 and 1.

In order to represent numbers, multiple such bits are used.

A.1 Representation of an Integer

Most programming languages support two kinds of integers, signed integers and unsigned integers. The unsigned integers can represent only non-negative numbers including zero. The signed numbers can represent positive numbers, negative numbers and zero.

A.1.1 Unsigned number representation

Let's assume that a bit pattern for an unsigned number is stored in n bits. The most common choices for n are 8, 16, 32 and 64 in IA32 processor architectures. Such integers are also known as unsigned byte (8-bit), word (16-bit), long (32-bit) and quadword (64-bit). If the bit pattern is denoted by

$$b_{n-1}b_{n-2}\ldots b_2b_1b_0$$

then the value of the number is given by

$$N = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Representation of a decimal number can be found by successive division of this number by 2. Each time the number is divided by 2, remainder defines a bit b_i.

As an example, let's consider decimal number 242 to be represented in binary. Division by 2 yields a remainder of 0 (b_0) and quotient of 121. Another division by 2 gives a remainder of 1 (b_1) and quotient of 60. By continuing the division in this way, we get the values of b_2, b_3, b_4, b_5, b_6, b_7 as 0, 0, 1, 1, 1, 1 respectively. The binary representation of number 242 then can be written as 11110010. Often it is easier to read the number in hexadecimal representation which groups four bits at a time. The hexadecimal representation of this number would be 0xF2.

If this number is to be represented in a byte, the contents of the byte will be 0xF2. If the same is to be represented in a word, the contents of the word will be 0x00F2. Similarly, the contents of a long word would be 0x000000F2.

Minimum and maximum value of an unsigned number using n bits of representation is given by 0 (when all n bits are zeros) and $2^n - 1$ (when all n bits are 1) respectively.

The values of unsigned numbers therefore range from 0 to 255 (or, $2^8 - 1$) for an unsigned byte, from 0 to 65535 (or, $2^{16} - 1$) for an unsigned word, from 0 to 4294967295 $(or, 2^{32} - 1)$ for an unsigned long, and from 0 to $2^{64} - 1$ for an unsigned quadword.

A.1.2 Signed number representation

Let's assume that the bit pattern for a signed number is represented using n bits. If the number is positive, then the most significant bit of the number is kept as 0 while the remaining $n-1$ bits provide the value. This scheme is similar to that of using unsigned number representation except that the most significant bit is always kept 0. In this scheme therefore we effectively have only $n - 1$ bits to represent the positive number leading to the minimum and maximum values of the number being 0 (when all bits are 0s) and $2^{n-1} - 1$ (when $n - 1$ bits are 1 and the most significant bit is 0.)

There are a variety of techniques used for representing negative numbers. The simplest one is to use sign and magnitude separately. The most significant bit of the n bit representation is 0 positive numbers and 1 for negative numbers. In the remaining $n - 1$ bits, the magnitude of the number is stored. For example, the representation of +5 and -5 using n as 8 would be the following.

$$\begin{aligned} +5 &= 00000101 \\ -5 &= 10000101 \end{aligned}$$

This scheme however is not friendly to the computer hardware for arithmetic computations. The modern computers use 2's complement number representation scheme. In this scheme, the negative numbers

are stored with the most significant bit as 1. The bit pattern for the negative numbers is obtained by taking 2's complement of their absolute value.

2's complement of the number can be obtained in a number of ways.

1. Adding a constant 2^n to the negative numbers (so that they become positive). Since 2^n is analogous to 0 using n bits, this mechanism effectively adds a 0.

2. Inverting all bits of the representation of absolute value of the negative number and then adding 1 to it.

Let's consider an example of representing negative numbers. Consider a negative number -10 that we wish to represent in eight bits.

By adding 2^8 (or 256) to -10, we get 246. Representation of 246 in binary with 8 bits would be 11110110. The most significant bit of this bit string is 1 indicating a negative number.

Alternately, the representation of absolute value (10) is 00001010. By inverting all bits we get 11110101. An addition of 1 to this yields 11110110 which is the representation of -10 using 2's complement number representation scheme.

The minimum value of the negative number range is -2^{n-1} (when all bits are zero except the most significant bit which is 1 to indicate the number as negative). The maximum value of the negative number range is -1 corresponding to the bit pattern having all bits as 1. Thus the entire range of signed numbers that can be represented using 2's complement number representation is from -2^{n-1} to $2^{n-1} - 1$.

It may be noted that given a bit pattern, it is not possible to say whether the number is represented using signed number system or unsigned number system. For example, consider an 8-bit pattern as 10001100. This number can be interpreted as unsigned numbers to give a value 140. The same bit pattern when interpreted as signed number represents a number -116.

As a matter of fact, any bit pattern by itself does not carry any information on the representation scheme. For example, a bit pattern may be interpreted as an instruction, as signed or unsigned integer, as real number or as any thing else. The interpretation of the bit pattern is done in the programs based on the kind of representation.

A.1.3 BCD numbers

BCD numbers are used in IA32 processors for various historic reasons. In this system of number representation scheme, a decimal number is represented using decimal digits rather than converting it to binary.

Each decimal digit can be represented in just four bits as follows.

0 = 0000	5 = 0101
1 = 0001	6 = 0110
2 = 0010	7 = 0111
3 = 0011	8 = 1000
4 = 0100	9 = 1001

A number of k decimal digits needs $4k$ bits for representation, or alternately, a bit string of n bits can represent numbers with $n/4$ decimal digits. Consider a number 23 to be represented using BCD representation in 16 bits. The number representation is given below.

0000000000100011 (or hexadecimal 0x0023).

This scheme by itself does not provide any natural mechanism for representing negative numbers. On IA32 processors, the BCD numbers in general purpose instruction set are supported by only six instructions – `daa`, `das`, `aaa`, `aad`, `aam` and `aas`. All of these instructions take two decimal digit numbers stored in a byte. The range of the numbers supported is from 00 to 99 only.

BCD numbers are also supported by x87 FPU instruction set. In this scheme, the numbers are stored using 80-bit (10-byte) data structure. In this data structure, the lower 9 bytes are used to store 18 decimal digit number while the higher order byte is 0x00 for positive numbers, 0x80 for negative numbers and 0xFF for indefinite numbers.

Figure 10.8 shows the BCD number representation scheme in x87 FPU instruction set.

A.2 Representation of characters

The characters are primarily used for input and output. For example, an integer can be displayed on the screen as a sequence of digits. Each of these digits is printed using characters. The printable characters include digits, alphabet both in small case and upper case of English and possibly in other scripts as well, special characters such as one of "`!#@%$^*&)(_`" etc. The printable characters can be enumerated and each character is given a unique code. These codes are passed around between devices and computer for doing an input output operation. It is essential that all devices use the same coding mechanism, failing which it would be hard for a program to work in a reasonable manner. For example, when a user presses 'A' on the keyboard, the computer receives the code for this letter. When the computer sends it to screen for display, the screen device should understand the same code as letter 'A' and display it.

There are various character coding standards that have evolved over time and are used in modern computers and programming languages.

A.2.1 ASCII code

American Standard Code for Information Interchange (ASCII) code is a seven bit coding standard that defines 128 characters including several printable characters and some control characters. The ASCII code being in the range 0 to 127, fit well in almost all supported data structures such as byte, word, long etc. of modern machines. The ASCII character code chart is given in Appendix G. Character codes between 0 to 31 and character code 127 provide 33 control characters. Remaining character codes between 32 to 126 provide 95 printable characters. Definition of some of the control characters is the following.

NUL: The NUL character is coded as 0. In GNU libc functions and C programming language this character is used to mark the end of a character string.

BEL: The BEL character is used to give a beeper sound. This character therefore does not change any thing on the display.

BS: The BS or backspace character is normally used to delete a character on the left of the cursor. This character is used primarily for interactive input-output.

HT: The horizontal tab or just the tab character is used to move the cursor forward on the same line to the nearest column that is aligned to a multiple of 8. This character is used typically to align the text in tabular form when shown on the displays.

LF: The LF or linefeed character is used to indicate the start of the new line. The usual meaning of this character is to set the cursor on the next row but the same column. However the standard way it is implemented in GNU/Linux and many programming languages is to set the cursor to the next row and first column.

FF: The FF or form feed character is used to end the page and move to the beginning of the next page on a printer device.

CR: The CR or carriage return character is used to position the cursor at the first column of the same line.

The ASCII character code nicely fits in a single byte. However using a single byte, it is possible to enumerate 256 characters between 0 to 255. Often there are many more printable characters needed in practical displays. These include characters such as box drawing characters, characters in international code sets etc. Accordingly many character sets evolved that use the range of characters between 128 to 255 in a variety of manners. With the invent of Internet some of these codes were standardized as ISO standards.

A.2.2 ISO8859 code

The ISO8859 is a set of standards defined by ISO/IEC to represent international character codes using 8 bit character coding schemes. These codes are compatible to the ASCII standard and therefore define characters with codes only between 128 to 255. The codes between 0 to 127 are the same as the ASCII character codes.

ISO 8859 is divided into 16 parts. Many of these are not in use now and have been abandoned. The following table shows some of these coding standards.

ISO 8859-1	Covers most Western European languages such as Danish, Dutch, English, Finnish, French, German, Spanish etc.
ISO 8859-2	Covers central and eastern European languages that use a Roman alphabet such as Polish, Croatian, Czech, Slovak, Slovenian and Hungarian.
ISO 8859-5	Covers languages such as Bulgarian, Macedonian, Russian, Serbian, and Ukrainian.
ISO 8859-6	Covers the most common Arabic language characters.
ISO 8859-8	Covers the modern Hebrew alphabet as used in Israel.
ISO 8859-11	Covers most glyphs needed for the Thai language.
ISO 8859-15	A revision of 8859-1 that removes some little-used symbols replacing them with the characters such as euro symbol.

A.2.3 Unicode and ISO/IEC 10646 standard code

ISO10646 and Unicode standards define a term called "Universal Character Set" or just the UCS. In this character coding standard, all characters in the scripts of the world can be represented.

In the Universal Character Set each character is assigned a code as well as an official name. The codes are commonly preceded by "U+" followed by a hexadecimal number such as `U+0041` for the character "Latin capital letter A". The UCS characters between `U+0000` and `U+00FF` are identical to those defined in the ISO 8859-1. Some UCS characters are shown in figure A.1.

The UCS characters may be stored in one of several character encoding forms for the Universal Character Set. Two commonly used schemes are UCS-2 and UCS-4. The UCS-2 uses a single code between 0 and 65535 for each character. A character code uses exactly two bytes for representation.

The UCS-4 uses a single code value between 0 to 0x7FFFFFFF for each character fitting each character code exactly in four bytes for rep-

	UCS Code	Name of the Character
R	U+0052	LATIN CAPITAL LETTER R
ಠ	U+0C95	KANNADA LETTER KA
Ù	U+00D9	LATIN CAPITAL LETTER U WITH GRAVE
≈	U+2248	ALMOST EQUAL TO
न	U+0928	DEVANAGARI LETTER NA
Ж	U+0416	CYRILLIC CAPITAL LETTER ZHE
ם	U+05DE	HEBREW LETTER MEM
ى	U+0649	ARABIC LETTER ALEF MAKSURA
ਈ	U+0A08	GURUMUKHI LETTER II
த	U+0BA4	TAMIL LETTER TA
જ	U+0A9C	GUJARATI LETTER JA

Figure A.1: UCS Character Examples

resentation. The character codes are however defined only between 0 to 0x10FFFF.

Both UCS-2 and UCS-4 codes are non-compatible with the ASCII coding standard which uses just 7 bits (or one byte in most implementations) for the code. For such compatibilities, the UCS also defines several methods for encoding a string of characters as a sequence of bytes, such as UTF-8 and UTF-16. The UTF-8 coding standard has the following features.

1. UCS characters between `U+0000` and `U+007F` are encoded with just one byte with a value between 0x00 to 0x7F. Thus a simple ASCII string becomes the same as UFT-8 string.
2. UCS characters larger than `U+007F` are encoded as sequence of several bytes. All of these bytes are non-conflicting with the bytes for ASCII. Each of these bytes have the most significant bit set.
3. UTF-8 coding standard can encode all possible UCS codes between 0 to 0x7FFFFFFF.

The Unicode Standard additionally specifies algorithms for rendering presentation forms of scripts, handling of bi-directional texts, algorithms for sorting and string comparison etc.

A.3 Floating Point numbers

Floating point numbers are typically represented using IEEE754 standard method of representing real numbers. There are three kinds of numbers based on precision and range. These numbers are called single-precision floating point numbers, double-precision floating point numbers and double extended-precision floating point numbers.

The real number representation schemes have been covered in details in chapter 9.

Appendix B

IA32 Processor Instruction Set

In this appendix, instructions from IA32 processor instruction set are given.

B.1 Representation used

In the description of IA32 processor instruction set, the following terms are used.

r8	One of %al, %ah, %bl, %bh, %cl, %ch, %dl or %dh
r16	One of %ax, %bx, %cx, %dx, %si, %di, %bp or %sp
r32	One of %eax, %ebx, %ecx, %edx, %esi, %edi, %ebp or %esp
r	r8 or r16 or r32
im8	8 bit immediate constant
im16	16 bit immediate constant
im32	32 bit immediate constant
imm	8-bit or 32-bit immediate constant
m	Memory operand. Size dependent upon the instruction.
m8	8 bit wide memory
m16	16 bit wide memory
m32	32 bit wide memory
m48	48 bit wide memory
m64	64 bit wide memory
m80	80 bit wide memory
m128	128 bit wide memory
rm8	r8 or m8
rm16	r16 or m16
rm32	r32 or m32

`rmi8`	`r8` or `m8` or `im8`
`rmi16`	`r16` or `m16` or `im16`
`rmi32`	`r32` or `m32` or `im32`
`target`	Label in the program
`sr`	one of `%ds`, `%es`, `%fs`, `%gs` or `%ss`
`%st`	x87 FPU register stack top
`%st(i)`	x87 FPU register on register stack
`mm`	One of MMX registers `%mm0` to `%mm7`
`m32mm`	An MMX register or 32 bit wide memory
`m64mm`	An MMX register or 64 bit wide memory
`xmm`	One of XMM registers `%xmm0` to `xmm7`
`m32xmm`	An XMM register or 32 bit wide memory
`m64xmm`	An XMM register or 64 bit wide memory
`m128xmm`	An XMM register or 128 bit wide memory

In all instructions given below, the destination of the instruction is the last argument.

B.2 General Purpose Instructions

aaa — ASCII ADJUST `%al` AFTER ADDITION (Page 99)

Variants	*Flags affected*
`aaa`	AF and CF represent carry. OF, SF, ZF, PF undefined.

aad — ASCII ADJUST `%ax` BEFORE DIVISION (Page 99)

Variants	*Flags affected*
`aad`	SF, ZF, PF as per the result. OF, AF, CF undefined.

aam — ASCII ADJUST `%ax` AFTER MULTIPLICATION (Page 99)

Variants	*Flags affected*
`aam`	SF, ZF, PF as per the result in `%al`. OF, AF, CF undefined.

aas — ASCII ADJUST %al AFTER SUBTRACTION (Page 99)

Variants	*Flags affected*
`aas`	AF and CF represent borrow. OF, SF, ZF, PF undefined.

adc — ADD WITH CARRY (Page 88)

Variants	*Flags affected*

`adcb im8, rm8`
`adcw im16, rm16`
`adcl im32, rm32`
`adc r8, rm8`
`adc r16, rm16`
`adc r32, rm32`
`adc rm8, r8`
`adc rm16, r16`
`adc rm32, r32`
	SF, CF, OF, SF, ZF, PF according to the result.

Operation: `dest = src + dest + CF`.

add — INTEGER ADDITION (Page 87)

Variants	*Flags affected*

`addb im8, rm8`
`addw im16, rm16`
`addl im32, rm32`
`add r8, rm8`
`add r16, rm16`
`add r32, rm32`
`add rm8, r8`
`add rm16, r16`
`add rm32, r32`
	SF, CF, OF, SF, ZF, PF according to the result.

Operation: `dest = src + dest`.

and — BITWISE LOGICAL AND OPERATION (Page 100)

Variants	*Flags affected*

`andb im8, rm8`
`andw im16, rm16`
`andl im32, rm32`
`and r8, rm8`
`and r16, rm16`
`and r32, rm32`
`and rm8, r8`
`and rm16, r16`
`and rm32, r32`
	OF, CF = 0, AF Undefined. SF, ZF and PF according to the result.

Operation: `dest = src` $\wedge$ `dest`.

bound	CHECK REGISTER FOR BOUNDS	
Variants		*Flags affected*
`bound m32, r16` `bound m64, r32`		None.

Operation: Check if a register is out of bound. The memory variable provides lower and upper bounds. Exception is generated if out of bound condition is true.

bsf	SCAN FORWARD FOR BIT 1	(Page 118)
Variants		*Flags affected*
`bsf rm16, r16` `bsf rm32, r32`		ZF=1, if source was 0. 0 otherwise. PF, SF, OF, CF, AF undefined.

bsr	SCAN REVERSE FOR BIT 1	(Page 118)
Variants		*Flags affected*
`bsr rm16, r16` `bsr rm32, r32`		ZF=1, if source was 0. 0 otherwise. PF, SF, OF, CF, AF undefined.

bswap	REVERSE (SWAP) BYTE ORDER	(Page 31)
Variants		*Flags affected*
`bswap r32`		None

bt	BIT TEST	(Page 115)
Variants		*Flags affected*

`bt r16, rm16`
`bt r32, rm32`
`btw imm, rm16`
`btl imm, rm32`
		CF=addressed bit. AF, SF, ZF, OF, PF undefined.

btc	BIT TEST AND INVERT(COMPLEMENT)	(Page 115)
Variants		*Flags affected*

`btc r16, rm16`
`btc r32, rm32`
`btcw imm, rm16`
`btcl imm, rm32`
		CF=initial value of addressed bit. AF, SF, ZF, OF, PF undefined.

btr — BIT TEST AND RESET TO 0 (Page 115)

Variants	*Flags affected*

`btr r16, rm16`
`btr r32, rm32`
`btrw imm, rm16`
`btrl imm, rm32`
	CF=initial value of addressed bit. AF, SF, ZF, OF, PF undefined.

bts — BIT TEST AND SET TO 0 (Page 115)

Variants	*Flags affected*

`bts r16, rm16`
`bts r32, rm32`
`btsw imm, rm16`
`btsl imm, rm32`
	CF=initial value of addressed bit. AF, SF, ZF, OF, PF undefined.

call — CALL FUNCTION (Page 61)

Variants	*Flags affected*

`call target`
`callw *rm16`
`calll *rm32`
	None

There are a few other variants of the `call` instruction which are not covered here. These are used primarily for system programming and real-mode of execution.

cbw/cbtw — CONVERT BYTE IN `%al` TO WORD IN `%ax` (Page 46)

Variants	*Flags affected*
`cbw` `cbtw`	None

cbtw — *See* `cbw` (Page 47)

cdq/cltd — CONVERT LONG WORD IN `%eax` TO QUAD WORD IN `%edx:%eax`

Variants	*Flags affected*
`cdq` `cltd`	None

clc — CLEAR CF TO 0 (Page 90)

Variants	*Flags affected*
`clc`	CF=0. SF, PF, ZF, OF, AF are not modified.

cld — CLEAR DF TO 0 (Page 109)

Variants	*Flags affected*
`cld`	None

cli — CLEAR INTERRUPT FLAG TO 0

Variants	*Flags affected*
`cli`	None

cltd — *See* `cdq` (Page 47)

cmc — COMPLEMENT (INVERT) CF (Page 90)

Variants	*Flags affected*
`cmc`	CF as per the operation. SF, PF, ZF, OF and AF are not modified.

cmova/cmovnbe — CONDITIONAL MOVE IF 'ABOVE'/'NOT-BELOW-OR-EQUAL' (Page 38)

Variants	*Flags affected*

`cmova rm16, r16`
`cmova rm32, r32`
`cmovnbe rm16, r16`
`cmovnbe rm32, r32`
	None

cmovae/cmovnb/cmovnc — CONDITIONAL MOVE IF 'ABOVE-OR-EQUAL'/'NOT-BELOW'/'NO-CARRY' (Page 38)

Variants	*Flags affected*

`cmovae rm16, r16`
`cmovae rm32, r32`
`cmovnb rm16, r16`
`cmovnb rm32, r32`
`cmovnc rm16, r16`
`cmovnc rm32, r32`
	None

cmovb/cmovc/ cmovnae

CONDITIONAL MOVE IF 'BELOW'/'CARRY'/'NOT-ABOVE-OR-EQUAL' (Page 38)

Variants

```
cmovb rm16, r16
cmovb rm32, r32
cmovc rm16, r16
cmovc rm32, r32
cmovnae rm16, r16
cmovnae rm32, r32
```

Flags affected

None

cmovbe/cmovna

CONDITIONAL MOVE IF 'BELOW-OR-EQUAL'/'NOT-ABOVE' (Page 38)

Variants

```
cmovbe rm16, r16
cmovbe rm32, r32
cmovna rm16, r16
cmovna rm32, r32
```

Flags affected

None

cmovc

See `cmovb` (Page 38)

cmove/cmovz

CONDITIONAL MOVE IF 'EQUAL'/'ZERO' (Page 38)

Variants

```
cmove rm16, r16
cmove rm32, r32
cmovz rm16, r16
cmovz rm32, r32
```

Flags affected

None

cmovg/cmovnle

CONDITIONAL MOVE IF 'GREATER'/'NOT-LESS-OR-EQUAL' (Page 38)

Variants

```
cmovg rm16, r16
cmovg rm32, r32
cmovnle rm16, r16
cmovnle rm32, r32
```

Flags affected

None

cmovge/cmovnl

CONDITIONAL MOVE IF 'GREATER-OR-EQUAL'/'NOT-LESS' (Page 38)

Variants

```
cmovge rm16, r16
cmovge rm32, r32
cmovnl rm16, r16
cmovnl rm32, r32
```

Flags affected

None

cmovl/cmovnge

CONDITIONAL MOVE IF 'LESS'/'NOT-GREATER-OR-EQUAL' (Page 38)

Variants

```
cmovl rm16, r16
cmovl rm32, r32
cmovnge rm16, r16
cmovnge rm32, r32
```

Flags affected

None

cmovle/cmovng

CONDITIONAL MOVE IF 'LESS-OR-EQUAL'/'NOT-GREATER' (Page 38)

Variants

```
cmovle rm16, r16
cmovle rm32, r32
cmovng rm16, r16
cmovng rm32, r32
```

Flags affected

None

cmovna

See `cmovbe` (Page 38)

cmovnae

See `cmovb` (Page 38)

cmovnb

See `cmovae` (Page 38)

cmovnbe

See `cmova` (Page 38)

cmovnc

See `cmovae` (Page 38)

cmovne/cmovnz

CONDITIONAL MOVE IF 'NOT-EQUAL'/'NOT-ZERO' (Page 38)

Variants

```
cmovne rm16, r16
cmovne rm32, r32
cmovnz rm16, r16
cmovnz rm32, r32
```

Flags affected

None

cmovng	*See* `cmovle`	(Page 38)

cmovnge	*See* `cmovl`	(Page 38)

cmovnl	*See* `cmovge`	(Page 38)

cmovnle	*See* `cmovg`	(Page 38)

cmovno	CONDITIONAL MOVE IF 'NO-OVERFLOW'	(Page 38)

Variants	*Flags affected*
`cmovno rm16, r16` `cmovno rm32, r32`	None

cmovnp/cmovpo	CONDITIONAL MOVE IF 'NO-PARITY'/'PARITY-ODD'	(Page 38)

Variants	*Flags affected*

`cmovnp rm16, r16`
`cmovnp rm32, r32`
`cmovpo rm16, r16`
`cmovpo rm32, r32`
	None

cmovns	CONDITIONAL MOVE IF 'NO-SIGN' (NON-NEGATIVE)	(Page 38)

Variants	*Flags affected*
`cmovns rm16, r16` `cmovns rm32, r32`	None

cmovo	CONDITIONAL MOVE IF 'OVERFLOW'	(Page 38)

Variants	*Flags affected*
`cmovo rm16, r16` `cmovo rm32, r32`	None

cmovp/cmovpe	CONDITIONAL MOVE IF 'PARITY'/'PARITY-EVEN'	(Page 38)

Variants	*Flags affected*

`cmovp rm16, r16`
`cmovp rm32, r32`
`cmovpe rm16, r16`
`cmovpe rm32, r32`
	None

cmovpe — *See* `cmovp` (Page 38)

cmovpo — *See* `cmovnp` (Page 38)

cmovs — CONDITIONAL MOVE IF 'SIGN' (NEGATIVE) (Page 38)

Variants	*Flags affected*
`cmovs rm16, r16` `cmovs rm32, r32`	None

cmp — COMPARE SOURCE AND DESTINATION (Page 91)

Variants	*Flags affected*

`cmpb im8, rm8`
`cmpw im16, rm16`
`cmpl im32, rm32`
`cmp r8, rm8`
`cmp r16, rm16`
`cmp r32, rm32`
`cmp rm8, r8`
`cmp rm16, r16`
`cmp rm32, r32`
	Flags are set according to `dest−src`.

cmpsb — COMPARE STRING BYTE

Variants	*Flags affected*
`cmpsb`	Flags are set according to the result.

cmpsl — COMPARE STRING WORD (16-BIT)

Variants	*Flags affected*
`cmpsl`	Flags are set according to the result.

cmpsw — COMPARE STRING LONG WORD (32-BIT)

Variants	*Flags affected*
`cmpsw`	Flags are set according to the result.

cmpxchg	COMPARE AND EXCHANGE	
Variants		*Flags affected*

`cmpxchg r8, rm8`
`cmpxchg r16, rm16`
`cmpxchg r32, rm32`
		ZF modified according to comparison. SF, OF, CF, PF, AF remain unmodified.

This instruction compares the destination with `%al`, `%ax` or `%eax`. If two operands are equal, destination is set to the value in source. Otherwise `%al`, `%ax` or `%eax` is set to the value in `dest`.

cmpxchg8b	COMPARE AND EXCHANGE 64 BIT NUMBERS	
Variants		*Flags affected*
`cmpxchg8b m64`		ZF modified according to comparison. SF, OF, CF, PF, AF remain unmodified.

This instruction compares `%edx:%eax` with memory operand and sets ZF accordingly. If two are equal it loads 64-bit value of `%ecx:%ebx` into memory. Otherwise, it loads 64-bit value in memory to registers `%edx:%eax`.

cpuid	LOAD REGISTERS WITH PROCESSOR IDENTIFICATION INFORMATION	(Page 261)
Variants		*Flags affected*
`cpuid`		None

cwd/cwtd	CONVERT WORD IN `%ax` TO LONG WORD IN `%dx:%ax`	(Page 47)
Variants		*Flags affected*
`cwd` `cwtd`		None

cwde/cwtl	CONVERT WORD IN `%ax` TO LONG WORD IN `%eax`	(Page 47)
Variants		*Flags affected*
`cwde` `cwtl`		None

cwtd	*See* `cwd`	(Page 47)

cwtl	*See* `cwde`	(Page 47)

daa — BCD ADJUSTMENT OF `%al` AFTER AN ADDITION (Page 97)

Variants	*Flags affected*
`daa`	CF, AF=decimal carry, OF undefined, PF, ZF, SF: As per the result in `%al`

das — BCD ADJUSTMENT OF `%al` AFTER A SUBTRACTION (Page 98)

Variants	*Flags affected*
`das`	CF, AF=decimal carry, OF undefined, PF, ZF, SF: As per the result in `%al`

dec — DECREMENT (Page 91)

Variants	*Flags affected*

`decb rm8`
`decw rm16`
`decl rm32`
	CF unchanged, OF, ZF, AF, SF, PF change according to the result.

Operation: `dest=dest-1`.

div — UNSIGNED DIVISION (Page 94)

Variants	*Flags affected*

`divb rm8`
`divw rm16`
`divl rm32`
	All flags change and are undefined.

Instruction argument is treated as divisor. Dividend is implied in registers as given on page 94.

enter — CREATE STACK FRAME FOR AN ENTRY TO A FUNCTION (Page 78)

Variants	*Flags affected*
`enter im16, im8`	None

The most common used value of the second immediate constant is 0.

idiv — INTEGER (SIGNED) DIVISION (Page 94)

Variants	*Flags affected*

`idivb rm8`
`idivw rm16`
`idivl rm32`
	All flags change and are undefined.

imul — INTEGER (SIGNED) MULTIPLICATION (Page 92)

Variants	*Flags affected*
`imul rm8`	SF, ZF, ZF, PF: Undefined.
`imul rm16`	CF, OF: indicate signed
`imul rm32`	overflow.
`imul rmi16, r16`	
`imul rmi32, r32`	
`imul im8, r16`	
`imul im16, r32`	
`imul im8, rm16, r16`	
`imul im8, rm32, r32`	
`imul im16, rm16, r16`	
`imul im32, rm32, r32`	

In one operand form, the destination and source registers are implied (as described on page 92). In two operand form, destination operand is multiplied with the source operand and result goes back to the destination operand. In three operand form, instruction specifies two sources that are multiplied and the result is stored in the destination given as the last operand.

in — READ FROM AN INPUT PORT (Page 194)

Variants	*Flags affected*
`inb im8`	None
`inw im8`	
`inl im8`	
`inb %dx`	
`inw %dx`	
`inl %dx`	
`in im8, %al`	
`in im8, %ax`	
`in im8, %eax`	
`in %dx, %al`	
`in %dx, %ax`	
`in %dx, %eax`	

inc — INCREMENT (Page 91)

Variants	*Flags affected*
`incb rm8`	CF unchanged, OF, ZF, AF,
`incw rm16`	SF, PF change according to
`incl rm32`	the result.

Operation: `dest=dest+1`.

insb	INPUT ONE BYTE FROM A PORT (ADDRESS IN %dx) INTO STRING	(Page 195)
Variants	*Flags affected*	
`insb`	None	

insl	INPUT ONE 32-BIT WORD FROM A PORT (ADDRESS IN %dx) INTO STRING	(Page 195)
Variants	*Flags affected*	
`insl`	None	

insw	INPUT ONE 16-BIT LONG WORD FROM A PORT (ADDRESS IN %dx) INTO STRING	(Page 195)
Variants	*Flags affected*	
`insw`	None	

int	TRAP OR SOFTWARE INTERRUPT	(Page 126)
Variants	*Flags affected*	
`int im8`	Undefined. Depend on the OS configurations.	

into	INTERRUPT IF OF IS SET
Variants	*Flags affected*
`into`	Undefined. Depend on the OS configurations.

iret	RETURN FROM INTERRUPT SERVICE ROUTINE
Variants	*Flags affected*
`iret`	Flags are popped from stack

ja/jnbe	JUMP IF 'ABOVE'/'NO-CARRY'/'NOT-BELOW-OR-EQUAL'	(Page 56)
Variants	*Flags affected*	
`ja target` `jnbe target`	None	

jae/jnb/jnc	JUMP IF 'ABOVE-OR-EQUAL'/'NOT-BELOW'	(Page 56)
Variants	*Flags affected*	

`jae target`
`jnb target`
`jnc target`
	None	

jb/jc/jnae	JUMP IF 'BELOW'/'CARRY'/'NOT-ABOVE-OR-EQUAL'	(Page 56)
Variants	*Flags affected*	

`jb target`
`jc target`
`jnae target`
	None	

jbe/jna	JUMP IF 'BELOW-OR-EQUAL'/'NOT-ABOVE'	(Page 56)
Variants	*Flags affected*	
`jbe target` `jna target`	None	

jc	*See* `jb`	(Page 56)

jcxz	JUMP IF REGISTER `%cx=0`	(Page 59)
Variants	*Flags affected*	
`jcxz target`	None	

je/jz	JUMP IF 'EQUAL'/'ZERO'	(Page 56)
Variants	*Flags affected*	
`je target` `jz target`	None	

jecxz	JUMP IF REGISTER `%ecx=0`	(Page 59)
Variants	*Flags affected*	
`jecxz target`	None	

jg/jnle	JUMP IF 'GREATER'/'NOT-LESS-OR-EQUAL'	(Page 56)
Variants	*Flags affected*	
`jg target` `jnle target`	None	

jge/jnl	JUMP IF 'GREATER-OR-EQUAL'/'NOT-LESS'	(Page 56)
Variants	*Flags affected*	
`jge target` `jnl target`	None	

jl/jnge	JUMP IF 'LESS'/'NOT-GREATER-OR-EQUAL'	(Page 56)
Variants	*Flags affected*	
`jl target` `jnge target`	None	

jle/jng	JUMP IF 'LESS-OR-EQUAL'/'NOT-GREATER'	(Page 56)
Variants	*Flags affected*	
`jle target` `jng target`	None	

jmp	JUMP	(Page 55)
Variants	*Flags affected*	

`jmp target`
`jmpw *rm16`
`jmpl *rm32`
	None	

jna	*See* `jbe`	(Page 56)

jnae	*See* `jb`	(Page 56)

jnb	*See* `jae`	(Page 56)

jnbe	*See* `ja`	(Page 56)

jnc	*See* `jae`	(Page 56)

jne/jnz	JUMP IF 'NOT-EQUAL'/'NOT-ZERO'	(Page 56)
Variants	*Flags affected*	
`jne target` `jnz target`	None	

jng	*See* `jle`	(Page 56)

jnge	*See* `jl`	(Page 56)
jnl	*See* `jge`	(Page 56)
jnle	*See* `jg`	(Page 56)
jno	JUMP IF 'NOT-OVERFLOW'	(Page 56)

Variants	*Flags affected*
`jno target`	None

jnp/jpo	JUMP IF 'PARITY-ODD'/'NO-PARITY'	(Page 56)

Variants	*Flags affected*
`jnp target` `jpo target`	None

jns	JUMP IF 'SF=0' (NON-NEGATIVE)	(Page 56)

Variants	*Flags affected*
`jns target`	None

jnz	*See* `jne`	(Page 56)
jo	JUMP IF 'OF=1'	(Page 56)

Variants	*Flags affected*
`jo target`	

jp/jpe	JUMP IF 'PARITY-EVEN'/'PARITY'	(Page 56)

Variants	*Flags affected*
`jp target` `jpe target`	None

jpe	*See* `jp`	(Page 56)
jpo	*See* `jnp`	(Page 56)
js	JUMP IF 'SF=1' (NEGATIVE)	(Page 56)

Variants	*Flags affected*
`js target`	None

jz	*See* `je`	(Page 56)

lahf — LOAD 8 FLAG BITS INTO %ah REGISTER

Variants	*Flags affected*
`lahf`	None

lds — LOAD REGISTER %ds AND DESTINATION

Variants	*Flags affected*
`lds m32, r16` `lds m48, r32`	None

lea — LOAD EFFECTIVE ADDRESS

Variants	*Flags affected*
`lea m, r16` `lea m, r32`	None

This instruction computes effective address for the memory operand and puts that into a register. Most common use of this instruction is to compute a complicated expression involving up to two addition (one constant and two registers), and one constant multiplication.

leave — DEALLOCATE STACK FRAME OF A FUNCTION (Page 79)

Variants	*Flags affected*
`leave`	None

les — LOAD REGISTER %es AND DESTINATION

Variants	*Flags affected*
`les m32, r16` `les m48, r32`	None

lfs — LOAD REGISTER %fs AND DESTINATION

Variants	*Flags affected*
`lfs m32, r16` `lfs m48, r32`	None

lgs — LOAD REGISTER %gs AND DESTINATION

Variants	*Flags affected*
`lgs m32, r16` `lgs m48, r32`	None

lodsb	LOAD BYTE IN %al FROM STRING	(Page 109)
Variants	*Flags affected*	
lodsb	None	

lodsl	LOAD LONG WORD IN %eax FROM STRING	(Page 109)
Variants	*Flags affected*	
lodsl	None	

lodsw	LOAD WORD IN %ax FROM STRING	(Page 109)
Variants	*Flags affected*	
lodsw	None	

loop	DECREMENT COUNTER AND JUMP IF COUNTER NOT ZERO	(Page 59)
Variants	*Flags affected*	
loop target	None	

loope/loopz	DECREMENT COUNTER AND JUMP IF COUNTER NOT ZERO AND ZF=1	(Page 59)
Variants	*Flags affected*	
loope target loopz targer	None	

loopne/loopnz	DECREMENT COUNTER AND JUMP IF COUNTER NOT ZERO AND ZF = 0	(Page 59)
Variants	*Flags affected*	
loopne target loopnz target	None	

loopnz	*See* loopne	(Page 59)

loopz	*See* loope	(Page 59)

lss	LOAD REGISTER %ss AND DESTINATION	
Variants	*Flags affected*	
lss m32, r16 lss m48, r32	None	

mov	COPY (MOVE) DATA FROM SOURCE TO DESTINATION	(Page 27)

Variants	*Flags affected*

`mov rmi8, r8`
`mov rmi16, r16`
`mov rmi32, r32`
`mov r8, m8`
`mov r16, m16`
`mov r32, m32`
`movb im8, m8`
`movw im16, m16`
`movl im32, m32`
`mov rm16, sr`
`mov sr, rm16`
	None

movsb	MOVE STRING BYTE FROM SOURCE TO DESTINATION	(Page 109)

Variants	*Flags affected*
`movsb`	None

movsbl	MOVE FROM BYTE TO 32-BIT LONG WORD WITH SIGN EXTENSION	(Page 47)

Variants	*Flags affected*
`movsbl rm8, r32`	None

movsbw	MOVE FROM BYTE TO 16-BIT WORD WITH SIGN EXTENSION	(Page 47)

Variants	*Flags affected*
`movsbw rm8, r16`	None

movsl	MOVE STRING LONG WORD (32-BIT) FROM SOURCE TO DESTINATION	(Page 109)

Variants	*Flags affected*
`movsl`	None

movsw	MOVE STRING WORD (16-BIT) FROM SOURCE TO DESTINATION	(Page 109)

Variants	*Flags affected*
`movsw`	None

movswl	MOVE FROM 16-BIT WORD TO 32-BIT LONG WORD WITH SIGN EXTENSION	(Page 47)
Variants	*Flags affected*	
`movswl rm16, r32`	None	

movzbl	MOVE FROM BYTE TO 32-BIT LONG WORD WITH ZERO EXTENSION	(Page 47)
Variants	*Flags affected*	
`movzbl rm8, r32`	None	

movzbw	MOVE FROM BYTE TO 16-BIT WORD WITH ZERO EXTENSION	(Page 47)
Variants	*Flags affected*	
`movzbw rm8, r16`	None	

movzwl	MOVE FROM 16-BIT WORD TO 32-BIT LONG WORD WITH ZERO EXTENSION	(Page 47)
Variants	*Flags affected*	
`movzwl rm16, r32`	None	

mul	UNSIGNED MULTIPLY	(Page 94)
Variants	*Flags affected*	

`mulb rm8`
`mulw rm16`
`mull rm32`
	OF, CF=1 if upper half of result != 0, 1 otherwise. ZF, SF, AF, PF Undefined.	

neg	NEGATE	(Page 91)
Variants	*Flags affected*	

`neg r`
`negb m8`
`negw m16`
`negl m32`
	CF=0 if source was 0, 1 otherwise. OF, ZF, SF, AF, PF as per the result.	

nop	NO OPERATION
Variants	*Flags affected*
`nop`	None

This instruction can be coded in 1 byte to 9 bytes in machine language. It is primarily used for alignment purposes in code sections.

---

not — PERFORM BITWISE LOGICAL NOT OPERATION (Page 100)

Variants	*Flags affected*

`not r`
`notb m8`
`notw m16`
`notl m32`
	None

---

or — PERFORM BITWISE LOGICAL OR OPERATION (Page 100)

Variants	*Flags affected*

`orb im8, rm8`
`orw im16, rm16`
`orl im32, rm32`
`or r8, rm8`
`or r16, rm16`
`or r32, rm32`
`or rm8, r8`
`or rm16, r16`
`or rm32, r32`
	OF and CF = 0, AF is undefined. ZF, PF and SF set according to the result.

---

out — OUTPUT A BYTE, WORD OR LONG WORD TO A PORT (Page 195)

Variants	*Flags affected*

`outb im8`
`outw im8`
`outl im8`
`outb %dx`
`outw %dx`
`outl %dx`
`out %al, im8`
`out %ax, im8`
`out %eax, im8`
`out %al, %dx`
`out %ax, %dx`
`out %eax, %dx`
	None

outsb	OUTPUT STRING BYTE TO PORT (ADDRESS IN %dx)	(Page 195)
Variants	*Flags affected*	
`outsb`	None	

outsl	OUTPUT STRING LONG WORD TO PORT (ADDRESS IN %dx)	(Page 195)
Variants	*Flags affected*	
`outsl`	None	

outsw	OUTPUT STRING WORD TO PORT (ADDRESS IN %dx)	(Page 195)
Variants	*Flags affected*	
`outsw`	None	

pop	POP A VALUE FROM STACK	(Page 42)
Variants	*Flags affected*	

`pop r16`
`pop r32`
`popw m16`
`popl m32`
`pop sr`
	None	

popal/popaw	POP ALL 16-BIT OR 32-BIT GENERAL-PURPOSE REGISTERS FROM STACK	(Page 46)
Variants	*Flags affected*	
`popal` `popaw`	None	

popaw	*See* `popal`	(Page 46)

popf	POP 16-BIT OR 32-BIT VALUE FROM STACK TO %eflags	(Page 46)
Variants	*Flags affected*	
`popfw` `popfl`	Values as taken from stack.	

push	PUSH A VALUE ONTO STACK	(Page 42)

Variants

```
pushw im16
pushl im32
push r16
push r32
pushw m16
pushl m32
push sr
```

Flags affected

None

pushal/pushaw	PUSH ALL 16-BIT OR 32-BIT GENERAL-PURPOSE REGISTERS ONTO STACK	(Page 45)

Variants

```
pushal
pushaw
```

Flags affected

None

pushaw	*See* `pushal`	(Page 45)

pushf	PUSH 16-BIT OR 32-BIT VALUE FROM `%eflags` REGISTER ONTO STACK	(Page 46)

Variants

```
pushfw
pushfl
```

Flags affected

None

rcl	ROTATE LEFT THROUGH CARRY	(Page 104)

Variants

```
rcl im8, r
rcl %cl, r
rclb im8, m8
rclw im8, m16
rcll im8, m32
rclb %cl, m8
rclw %cl, m16
rcll %cl, m32
```

Flags affected

CF: Value shifted, OF: Undefined, SF, ZF, PF, AF: unchanged.

rcr	ROTATE RIGHT THROUGH CARRY	(Page 104)

Variants	*Flags affected*

`rcr im8, r`
`rcr %cl, r`
`rcrb im8, m8`
`rcrw im8, m16`
`rcrl im8, m32`
`rcrb %cl, m8`
`rcrw %cl, m16`
`rcrl %cl, m32`
	CF: Value shifted, OF: Undefined, SF, ZF, PF, AF: unchanged.

rep	REPEAT NEXT STRING INSTRUCTION WHILE %ecx NOT ZERO	(Page 112)

Variants	*Flags affected*
`rep`	None

repe/repz	REPEAT NEXT STRING INSTRUCTION WHILE ZF=1 AND %ecx NOT ZERO	(Page 112)

Variants	*Flags affected*
`repe` `repz`	None

repne/repnz	REPEAT NEXT STRING INSTRUCTION WHILE ZF=0 AND %ecx NOT ZERO	(Page 112)

Variants	*Flags affected*
`repne` `repnz`	None

repnz	*See* `repne`	(Page 112)

repz	*See* `repe`	(Page 112)

ret	RETURN FROM A FUNCTION	(Page 63)

Variants	*Flags affected*
`ret` `ret im16`	None

rol — ROTATE LEFT (Page 104)

Variants

```
rol im8, r
rol %cl, r
rolb im8, m8
rolw im8, m16
roll im8, m32
rolb %cl, m8
rolw %cl, m16
roll %cl, m32
```

Flags affected

CF: Value shifted, OF: Undefined, SF, ZF, PF, AF: unchanged.

ror — ROTATE RIGHT (Page 104)

Variants

```
ror im8, r
ror %cl, r
rorb im8, m8
rorw im8, m16
rorl im8, m32
rorb %cl, m8
rorw %cl, m16
rorl %cl, m32
```

Flags affected

CF: Value shifted, OF: Undefined, SF, ZF, PF, AF: unchanged.

sahf — STORE ZF, CF, AF, PF AND SF FROM %ah REGISTER

Variants

```
sahf
```

Flags affected

As per the value in %ah

sal/shl — SHIFT LEFT ARITHMETIC/LOGICAL (Page 101)

Variants	*Flags affected*

`sal im8, r`
`sal %cl, r`
`salb im8, m8`
`salw im8, m16`
`sall im8, m32`
`salb %cl, m8`
`salw %cl, m16`
`sall %cl, m32`
`shl im8, r`
`shl %cl, r`
`shlb im8, m8`
`shlw im8, m16`
`shll im8, m32`
`shlb %cl, m8`
`shlw %cl, m16`
`shll %cl, m32`
	CF, OF, AF: Undefined. ZF, PF,SF: According to the result

sar — SHIFT ARITHMETIC RIGHT (Page 101)

Variants	*Flags affected*

`sar im8, r`
`sar %cl, r`
`sarb im8, m8`
`sarw im8, m16`
`sarl im8, m32`
`sarb %cl, m8`
`sarw %cl, m16`
`sarl %cl, m32`
	CF, OF, AF: Undefined. ZF, PF,SF: According to the result

sbb — SUBTRACT WITH BORROW (Page 88)

Variants	*Flags affected*

`sbbb im8, rm8`
`sbbw im16, rm16`
`sbbl im32, rm32`
`sbb r8, rm8`
`sbb r16, rm16`
`sbb r32, rm32`
`sbb rm8, r8`
`sbb rm16, r16`
`sbb rm32, r32`
	SF, CF, OF, SF, ZF, PF according to the result.

Operation: `dest=dest−(src+CF)`.

scasb	COMPARE STRING BYTE WITH %al	(Page 109)
Variants	*Flags affected*	
scasb	SF, CF, OF, SF, ZF, PF according to the outcome of comparison.	

scasl	COMPARE STRING LONG WORD WITH %eax	(Page 109)
Variants	*Flags affected*	
scasl	SF, CF, OF, SF, ZF, PF according to the outcome of comparison.	

scasw	COMPARE STRING WORD WITH %ax	(Page 109)
Variants	*Flags affected*	
scasw	SF, CF, OF, SF, ZF, PF according to the outcome of comparison.	

seta/setnbe	SET BYTE IF 'ABOVE'/'NOT-BELOW-OR-EQUAL'	(Page 121)
Variants	*Flags affected*	
seta rm8 setnbe rm8	None	

setae/setnb/setnc	SET BYTE 'IF-ABOVE-OR-EQUAL'/'NOT-BELOW'/'NO-CARRY'	(Page 121)
Variants	*Flags affected*	
setae rm8 setnb rm8 setnc rm8	None	

setb/setnae/setc	SET BYTE IF 'BELOW'/'NOT-ABOVE-OR-EQUAL'/'CF=1'	(Page 121)
Variants	*Flags affected*	
setb rm8 setnae rm8 setc rm8	None	

setbe/setna

SET BYTE IF 'BELOW-OR-EQUAL'/'NOT-ABOVE' (Page 121)

Variants	*Flags affected*
`setbe rm8`	None
`setna rm8`	

setc

See `setb` (Page 121)

sete/setz

SET BYTE IF 'EQUAL'/'ZERO' (Page 121)

Variants	*Flags affected*
`sete rm8`	None
`setz rm8`	

setg/setnle

SET BYTE IF 'GREATER'/'NOT-LESS-OR-EQUAL' (Page 121)

Variants	*Flags affected*
`setg rm8`	None
`setnle rm8`	

setge/setnl

SET BYTE IF 'GREATER-OR-EQUAL'/'NOT-LESS' (Page 121)

Variants	*Flags affected*
`setge rm8`	None
`setnl rm8`	

setl/setnge

SET BYTE IF 'LESS'/'NOT-GREATER-OR-EQUAL' (Page 121)

Variants	*Flags affected*
`setl rm8`	None
`setnge rm8`	

setle/setng

SET BYTE IF 'LESS-OR-EQUAL'/'NOT-GREATER' (Page 121)

Variants	*Flags affected*
`setle rm8`	None
`setng rm8`	

setna

See `setbe` (Page 121)

setnae

See `setb` (Page 121)

setnb — *See* `setae` (Page 121)

setnbe — *See* `seta` (Page 121)

setnc — *See* `setae` (Page 121)

setne/setnz — SET BYTE IF 'NOT-EQUAL'/'NOT-ZERO' (Page 121)

Variants	*Flags affected*
`setne rm8` `setnz rm8`	None

setng — *See* `setle` (Page 121)

setnge — *See* `setl` (Page 121)

setnl — *See* `setge` (Page 121)

setnle — *See* `setg` (Page 121)

setno — SET BYTE IF 'NO-OVERFLOW' (Page 121)

Variants	*Flags affected*
`setno rm8`	

setnp/setpo — SET BYTE IF 'PARITY-ODD'/'NO-PARITY' (Page 121)

Variants	*Flags affected*
`setnp rm8` `setpo rm8`	None

setns — SET BYTE IF 'NOT-SIGN' (NON-NEGATIVE) (Page 121)

Variants	*Flags affected*
`setns rm8`	None

setnz — *See* `setne` (Page 121)

seto — SET BYTE IF 'OF=1' (Page 121)

Variants	*Flags affected*
`seto rm8`	

setp/setpe — SET BYTE IF 'PARITY-EVEN'/'PF=1' (Page 121)

Variants	*Flags affected*
`setp rm8` `setpe rm8`	None

setpe — *See* `setp` (Page 121)

setpo — *See* `setnp` (Page 121)

sets — SET BYTE IF 'SF=1' (NEGATIVE) (Page 121)

Variants	*Flags affected*
`sets rm8`	None

setz — *See* `sete` (Page 121)

shl — *See* `sal` (Page 101)

shld — SHIFT LEFT DOUBLE (Page 103)

Variants	*Flags affected*

`shld im8, r16, rm16`
`shld %cl, r16, rm16`
`shld im8, r32, rm32`
`shld %cl, r32, rm32`
	CF=last bit shifted out. SF, PF, ZF according to the result. AF, OF undefined.

shr — SHIFT LOGICAL RIGHT (Page 101)

Variants	*Flags affected*

`shr im8, r`
`shr %cl, r`
`shrb im8, m8`
`shrw im8, m16`
`shrl im8, m32`
`shrb %cl, m8`
`shrw %cl, m16`
`shrl %cl, m32`
	CF, OF, AF: Undefined. ZF, PF,SF: According to the result

shrd — SHIFT RIGHT DOUBLE (Page 103)

Variants	*Flags affected*

`shrd im8, r16, rm16`
`shrd %cl, r16, rm16`
`shrd im8, r32, rm32`
`shrd %cl, r32, rm32`
	CF=last bit shifted out. SF, PF, ZF according to the result. AF, OF undefined.

stc	SET CF=1	(Page 90)
Variants	*Flags affected*	
`stc`	CY=1. All others are unaffected.	

std	SET DIRECTION FLAG=1	(Page 109)
Variants	*Flags affected*	
`std`	DF=1. All others are unaffected.	

sti	SET INTERRUPT FLAG=1	
Variants	*Flags affected*	
`sti`	IF=1. All others are unaffected.	

stosb	STORE BYTE INTO THE STRING	(Page 109)
Variants	*Flags affected*	
`stosb`	None	

stosl	STORE LONG WORD INTO THE STRING	(Page 109)
Variants	*Flags affected*	
`stosl`	None	

stosw	STORE WORD INTO THE STRING	(Page 109)
Variants	*Flags affected*	
`stosw`	None	

sub	SUBTRACT	(Page 87)
Variants	*Flags affected*	

`subb im8, rm8`
`subw im16, rm16`
`subl im32, rm32`
`sub r8, rm8`
`sub r16, rm16`
`sub r32, rm32`
`sub rm8, r8`
`sub rm16, r16`
`sub rm32, r32`
	SF, CF, OF, SF, ZF, PF according to the result.	

Operation: `dest=dest−src`.

test	LOGICAL COMPARE	(Page 121)

Variants	*Flags affected*

`test imm, r`
`testb im8, m8`
`testw im16 m16`
`testl im32 m32`
`test r8, m8`
`test r16, m16`
`test r32, m32`
	Flags set as per the logical AND operation of the two operands.

xadd	EXCHANGE SOURCE WITH DESTINATION AND ADD TWO OPERANDS

Variants	*Flags affected*

`xadd r8, rm8`
`xadd r16, rm16`
`xadd r32, rm32`
	Flags are set according to the result of addition.

Operation: `dest=src+dest`. `src`=old value of `dest`.

xchg	EXCHANGE VALUES OF TWO OPERANDS	(Page 30)

Variants	*Flags affected*

`xchg r8, rm8`
`xchg r16, rm16`
`xchg r32, rm32`
`xchg rm8, r8`
`xchg rm16, r16`
`xchg rm32, r32`
	None

xlat/xlatb	TABLE LOOKUP TRANSLATION

Variants	*Flags affected*
`xlat` `xlatb`	None

Operation: Translates a byte in `%al` from a table lookup. Table address is in `%ebx`.

xor	PERFORM BITWISE LOGICAL EXCLUSIVE OR BETWEEN TWO OPERANDS	(Page 100)

Variants	*Flags affected*

`xorb im8, rm8`
`xorw im16, rm16`
`xorl im32, rm32`
`xor r8, rm8`
`xor r16, rm16`
`xor r32, rm32`
`xor rm8, r8`
`xor rm16, r16`
`xor rm32, r32`
	OF, CF are cleared. ZF, PF, SF are set according to the result. AF is undefined.

B.3 x87 FPU instructions

All x87 FPU instructions modify C_0, C_1, C_2 and $C + 3$ condition flags in x87 FPU status register.

f2xm1	REPLACE `%st` BY $2^{\%\texttt{st}} - 1$	(Page 246)

Variants	*Flags affected*
`f2xm1`	None

fabs	REPLACE `%st` BY ITS ABSOLUTE VALUE	(Page 241)

Variants	*Flags affected*
`fabs`	None

fadd	ADD FLOATING-POINT VALUES	(Page 233)

Variants	*Flags affected*

`fadds m32`
`faddl m64`
`fadd %st(i), %st`
`fadd %st, %st(i)`
	None

Arguments in memory are single or double precision floating point numbers.

faddp	ADD FLOATING-POINT AND POP REGISTER STACK	(Page 233)

Variants	*Flags affected*
`faddp` `faddp %st(i)`	None

Result of addition is left on top of the register stack.

fbld	LOAD BCD NUMBER	(Page 253)
Variants	*Flags affected*	
`fbld m80`	None	

Stack depth is incremented by 1 due to the load. Newly loaded value is on the stack top.

fbstp	STORE BCD FROM `%st` IN MEMORY AND POP REGISTER STACK	(Page 253)
Variants	*Flags affected*	
`fbstp m80`	None	

fchs	CHANGE SIGN OF `%st`	(Page 241)
Variants	*Flags affected*	
`fchs`	None	

fclex	CLEAR FLOATING-POINT EXCEPTION FLAGS AFTER CHECKING FOR ERROR CONDITIONS	(Page 256)
Variants	*Flags affected*	
`fclex`	None	

fcmovbe	CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'BELOW-OR-EQUAL'	(Page 254)
Variants	*Flags affected*	
`fcmovbe %st(i), %st`	None	

fcmovb	CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'BELOW'	(Page 254)
Variants	*Flags affected*	
`fcmovb %st(i), %st`	None	

fcmove	CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'EQUAL'	(Page 254)
Variants	*Flags affected*	
`fcmove %st(i), %st`	None	

fcmovnbe

CONDITIONALLY COPY (MOVE) FLOATING-POINT VALUE IF 'NOT-BELOW-OR-EQUAL' (Page 254)

Variants	*Flags affected*
`fcmovnbe %st(i), %st`	None

fcmovnb

CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'NOT-BELOW' (Page 254)

Variants	*Flags affected*
`fcmovnb %st(i), %st`	None

fcmovne

CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'NOT-EQUAL' (Page 254)

Variants	*Flags affected*
`fcmovne %st(i), %st`	None

fcmovnu

CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'NOT-UNORDERED' (Page 254)

Variants	*Flags affected*
`fcmovnu %st(i), %st`	None

fcmovu

CONDITIONALLY COPY (MOVE) FLOATING POINT VALUE IF 'UNORDERED' (Page 254)

Variants	*Flags affected*
`fcmovu %st(i), %st`	None

fcom

COMPARE FLOATING POINT VALUES AND SET CONDITION FLAGS IN X87 FPU STATUS REGISTER (Page 247)

Variants	*Flags affected*

`fcoms m32`
`fcoml m64`
`fcom %st(i)`
`fcom`
	None

fcomi	COMPARE FLOATING-POINT AND SET FLAGS IN `%eflags`	(Page 249)

Variants	*Flags affected*
`fcomi %st(i), %st`	ZF, PF, CF as per the comparison. Others remain unaffected.

fcomip	COMPARE FLOATING-POINT, SET FLAGS IN `%eflags` AND POP	(Page 249)

Variants	*Flags affected*
`fcomip`	ZF, PF, CF as per the comparison. Others remain unaffected.

fcomp	COMPARE FLOATING POINT VALUES AND POP REGISTER STACK.	(Page 247)

Variants	*Flags affected*

`fcomps m32`
`fcompl m64`
`fcomp %st(i)`
`fcomp`
	None

Condition flags in x87 FPU status register are changed to reflect the result of comparison.

fcompp	COMPARE FLOATING POINT VALUES IN `%st(1)` AND `%st`; AND POP TWICE	(Page 247)

Variants	*Flags affected*
`fcompp`	None

Condition flags in x87 FPU status register are changed to reflect the result of comparison.

fcos	REPLACE `%st` BY COSINE OF `%st`	(Page 244)

Variants	*Flags affected*
`fcos`	None

fdecstp	DECREMENT FPU REGISTER STACK POINTER	(Page 256)

Variants	*Flags affected*
`fdecstp`	None

fdiv	DIVIDE FLOATING POINT VALUES	(Page 237)
Variants	*Flags affected*	

`fdivs m32`
`fdivl m64`
`fdiv %st(i), %st`
`fdiv %st, %st(i)`
	None	

fdivp	DIVIDE FLOATING POINT VALUES AND POP REGISTER STACK	(Page 237)
Variants	*Flags affected*	
`fdivp` `fdivp %st(i)`	None	

fdivr	REVERSE DIVIDE FLOATING POINT VALUES	(Page 237)
Variants	*Flags affected*	

`fdivrs m32`
`fdivrl m64`
`fdivr %st(i), %st`
`fdivr %st, %st(i)`
	None	

fdivrp	REVERSE DIVIDE FLOATING POINT VALUES AND POP REGISTER STACK	(Page 237)
Variants	*Flags affected*	
`fdivrp` `fdivrp %st(i)`	None	

ffree	MARK FLOATING POINT REGISTER FREE	(Page 256)
Variants	*Flags affected*	
`ffree %st(i)`	None	

fiadd	ADD INTEGER IN MEMORY TO `%st`	(Page 233)
Variants	*Flags affected*	
`fiadds m16` `fiaddl m32`	None	

ficom	COMPARE INTEGER AND `%st`	(Page 247)
Variants	*Flags affected*	
`ficoms m16` `ficoml m32`	None	

Comparison results are made available in x87 status register.

ficomp	COMPARE INTEGER WITH %st AND POP REGISTER STACK	(Page 247)
Variants	*Flags affected*	
`ficomps m16` `ficompl m32`	None	

Comparison results are made available in x87 status register.

fidiv	DIVIDE %st BY INTEGER IN MEMORY	(Page 237)
Variants	*Flags affected*	
`fidivs m16` `fidivl m32`	None	

fidivr	DIVIDE INTEGER IN MEMORY BY %st AND STORE RESULT IN %st	(Page 237)
Variants	*Flags affected*	
`fidivrs m16` `fidivrl m32`	None	

fild	LOAD INTEGER IN MEMORY TO REGISTER STACK	(Page 252)
Variants	*Flags affected*	

`filds m16`
`fildl m32`
`fildq m64`
	None	

Register stack depth increases by 1 after execution of this instruction.

fimul	MULTIPLY %st BY INTEGER IN MEMORY	(Page 237)
Variants	*Flags affected*	
`fimuls m16` `fimull m32`	None	

fincstp	INCREMENT FPU REGISTER STACK POINTER	(Page 256)
Variants	*Flags affected*	
`fincstp`	None	

finit — INITIALIZE FPU AFTER CHECKING ERROR CONDITIONS (Page 255)

Variants	*Flags affected*
`finit`	None

fist — STORE `%st` AS INTEGER IN MEMORY AFTER CONVERSION (Page 252)

Variants	*Flags affected*
`fists m16` `fistl m32`	None

fistp — STORE `%st` AS INTEGER IN MEMORY AFTER CONVERSION AND POP REGISTER STACK (Page 252)

Variants	*Flags affected*

`fistps m16`
`fistpl m32`
`fistpq m64`
	None

fisubr — SUBTRACT `%st` FROM INTEGER IN MEMORY WITH RESULT IN `%st` (Page 235)

Variants	*Flags affected*
`fisubrs m16` `fisubrl m32`	None

fisub — SUBTRACT INTEGER IN MEMORY FROM `%st` (Page 235)

Variants	*Flags affected*
`fisubs m16` `fisubl m32`	None

fld — LOAD SINGLE, DOUBLE AND EXTENDED DOUBLE PRECISION FLOATING POINT NUMBERS FROM MEMORY (Page 251)

Variants	*Flags affected*

`flds m32`
`fldl m64`
`fldt m80`
	None

Depth of register stack increases by 1 after this insruction.

fld1 — LOAD CONSTANT +1.0 ON REGISTER STACK (Page 244)

Variants	*Flags affected*
`fld1`	None

Depth of register stack increases by 1 after this instruction.

fldcw — LOAD FPU CONTROL WORD FROM MEMORY (Page 256)

Variants	*Flags affected*
`fldcw m16`	None

fldenv — LOAD FPU ENVIRONMENT FROM MEMORY (Page 257)

Variants	*Flags affected*
`fldenv m`	None

fldl2e — LOAD $\log_2 e$ ON REGISTER STACK (Page 244)

Variants	*Flags affected*
`fldl2e`	None

Depth of register stack increases by 1 after this instruction.

fldl2t — LOAD $\log_2 10$ ON REGISTER STACK (Page 244)

Variants	*Flags affected*
`fldl2t`	None

Depth of register stack increases by 1 after this instruction.

fldlg2 — LOAD $\log_{10} 2$ ON REGISTER STACK (Page 244)

Variants	*Flags affected*
`fldlg2`	None

Depth of register stack increases by 1 after this instruction.

fldln2 — LOAD $\log_e 2$ ON REGISTER STACK (Page 244)

Variants	*Flags affected*
`fldln2`	None

Depth of register stack increases by 1 after this instruction.

fldpi — LOAD π ON REGISTER STACK (Page 244)

Variants	*Flags affected*
`fldpi`	None

Depth of register stack increases by 1 after this instruction.

fldz	LOAD +0.0 ON REGISTER STACK	(Page 244)

Variants	*Flags affected*
`fldz`	None

Depth of register stack increases by 1 after this instruction.

fmul	MULTIPLY FLOATING POINT VALUES	(Page 237)

Variants	*Flags affected*

`fmuls m32`
`fmull m64`
`fmul %st(i), %st`
`fmul %st, %st(i)`
	None

fmulp	MULTIPLY FLOATING POINT VALUES AND POP REGISTER STACK	(Page 237)

Variants	*Flags affected*
`fmulp` `fmulp %st(i)`	None

Result remains on top of the register stack after execution.

fnclex	CLEAR FLOATING-POINT EXCEPTION FLAGS WITHOUT CHECKING FOR ERROR CONDITIONS	(Page 256)

Variants	*Flags affected*
`fnclex`	None

fninit	INITIALIZE FPU WITHOUT CHECKING ERROR CONDITIONS	(Page 255)

Variants	*Flags affected*
`fninit`	None

fnop	NO OPERATION INSTRUCTION IN X87 FPU	(Page 257)

Variants	*Flags affected*
`fnop`	None

fnsave	SAVE X87 FPU STATE WITHOUT CHECKING ERROR CONDITIONS	(Page 257)

Variants	*Flags affected*
`fnsave m`	None

fnstcw	STORE X87 CONTROL REGISTER WITHOUT CHECKING ERROR CONDITIONS	(Page 256)
Variants	*Flags affected*	
`fnstcw m16`	None	

fnstenv	STORE X87 FPU ENVIRONMENT WITHOUT CHECKING ERROR CONDITIONS	(Page 257)
Variants	*Flags affected*	
`fnstenv m`	None	

fnstsw	STORE X87 STATUS REGISTER WITHOUT CHECKING ERROR CONDITIONS	(Page 256)
Variants	*Flags affected*	
`fnstsw m16` `fnstsw %ax`	None	

fpatan	COMPUTE ARCTANGENT OF `%st(1)/%st`, POP REGISTER STACK AND REPLACE `%st` BY THE RESULT	(Page 244)
Variants	*Flags affected*	
`fpatan`	None	

fprem	REPLACE `%st` BY PARTIAL REMAINDER OF `%st/%st(1)`	(Page 239)
Variants	*Flags affected*	
`fprem`	None	

fprem1	REPLACE `%st` BY IEEE PARTIAL REMAINDER OF `%st/%st(1)`	(Page 239)
Variants	*Flags affected*	
`fprem1`	None	

fptan	COMPUTE PARTIAL TANGENT OF `%st` AND PUSH RESULT AND 1.0 ON REGISTER STACK	(Page 244)
Variants	*Flags affected*	
`fptan`	None	

This instruction increases the stack depth by 1. After execution result is in %st(1) and 1.0 is in %st.

frndint	ROUND %st TO INTEGER AS PER ROUNDING CONTROL IN X87 CONTROL REGISTER	(Page 241)
Variants	*Flags affected*	
frndint	None	

frstor	RESTORE FPU STATE	(Page 257)
Variants	*Flags affected*	
frstor m	None	

This instruction is counterpart to fnsave/fsave instruction.

fsave	SAVE X87 FPU STATE AFTER CHECKING ERROR CONDITIONS	(Page 257)
Variants	*Flags affected*	
fsave m	None	

fscale	SCALE %st BY %st(1)	(Page 241)
Variants	*Flags affected*	
fscale	None	

This instruction reverses the effects of fxtract instruction.

fsin	REPLACE %st BY ITS SINE	(Page 244)
Variants	*Flags affected*	
fsin	None	

fsincos	COMPUTE SINE AND COSINE OF %st	(Page 244)
Variants	*Flags affected*	
fsincos	None	

This instruction increase the stack depth by 1. After execution %st contains computed cosine and %st(1) contains computed sine.

fsqrt	REPLACE %st BY ITS SQUARE ROOT	(Page 240)
Variants	*Flags affected*	
fsqrt	None	

fstcw	STORE X87 CONTROL WORD AFTER CHECKING ERROR CONDITIONS	(Page 256)

Variants	*Flags affected*
`fstcw m16`	None

fstenv	STORE X87 FPU ENVIRONMENT AFTER CHECKING ERROR CONDITIONS	(Page 257)

Variants	*Flags affected*
`fstenv m`	None

fst	STORE FLOATING POINT VALUE IN %st TO DESTINATION	(Page 251)

Variants	*Flags affected*

`fsts m32`
`fstl m64`
`fst %st(i)`
	None

fstp	STORE FLOATING POINT VALUE IN %st TO DESTINATION AND POP REGISTER STACK	(Page 251)

Variants	*Flags affected*

`fstps m32`
`fstpl m64`
`fstpt m80`
`fstp %st(i)`
	None

fstsw	STORE X87 STATUS WORD AFTER CHECKING ERROR CONDITIONS	(Page 256)

Variants	*Flags affected*
`fstsw m16` `fstsw %ax`	None

fsub	SUBTRACT FLOATING POINT VALUES	(Page 235)

Variants	*Flags affected*

`fsubs m32`
`fsubl m64`
`fsub %st(i), %st`
`fsub %st, %st(i)`
	None

fsubp — SUBTRACT FLOATING POINT VALUES AND POP REGISTER STACK (Page 235)

Variants	*Flags affected*
`fsubp`	None
`fsubp %st(i)`	

fsubr — REVERSE SUBTRACT FLOATING POINT VALUES (Page 235)

Variants	*Flags affected*
`fsubrs m32`	None
`fsubrl m64`	
`fsubr %st(i), %st`	
`fsubr %st, %st(i)`	

fsubrp — REVERSE SUBTRACT FLOATING POINT VALUES AND POP REGISTER STACK (Page 235)

Variants	*Flags affected*
`fsubrp`	None
`fsubrp %st(i)`	

ftst — COMPARE `%st` AGAINST 0.0 (Page 249)

Variants	*Flags affected*
`ftst`	None

Result of comparison is available in x87 status register.

fucom — COMPARE UNORDERED FLOATING POINT VALUES (Page 249)

Variants	*Flags affected*
`fucom`	None
`fucom %st(i)`	

Result of comparison is available in x87 status register.

fucomi — COMPARE UNORDERED FLOATING POINT VALUES (Page 249)

Variants	*Flags affected*
`fucomi %st(i) %st`	ZF, PF, CF according to the comparison. All others unaffected.

Result of comparison is available in ZF, PF and CF.

fucomip	COMPARE UNORDERED FLOATING POINT VALUES AND POP REGISTER STACK	(Page 249)

Variants	*Flags affected*
`fucomip %st(i), %st`	ZF, PF, CF according to the comparison. All others unaffected.

Result of comparison is available in ZF, PF and CF.

fucomp	COMPARE UNORDERED FLOATING POINT VALUES AND POP REGISTER STACK	(Page 249)

Variants	*Flags affected*
`fucomp`	None
`fucomp %st(i)`	

Result of comparison is available in x87 status register.

fucompp	COMPARE UNORDERED FLOATING POINT VALUES AND POP REGISTER STACK TWICE	(Page 249)

Variants	*Flags affected*
`fucompp`	None

Result of comparison is available in x87 status register.

fxam	EXAMINE FLOATING POINT VALUE IN `%st`	(Page 250)

Variants	*Flags affected*
`fxam`	None

Condition flags in x87 status register are set to indicate the type of value in `%st`.

fxch	EXCHANGE FLOATING POINT VALUES IN REGISTERS	(Page 253)

Variants	*Flags affected*
`fxch`	None
`fxch %st(i)`	

fxrstor	RESTORE X87 AND SIMD STATE	(Page 257)

Variants	*Flags affected*
`fxrstor m`	None

Entire state is 512 bytes in size.

fxsave	SAVE X87 AND SIMD STATE	(Page 257)
Variants	*Flags affected*	
`fxsave m`	None	

Entire state is 512 bytes in size.

fxtract	EXTRACT EXPONENT AND SIGNIFICAND	(Page 241)
Variants	*Flags affected*	
`fxtract`	None	

fyl2x	COMPUTE $\%\texttt{st}(1) * \log_2(\%\texttt{st})$	(Page 246)
Variants	*Flags affected*	
`fyl2x`	None	

fyl2xp1	COMPUTE $\%\texttt{st}(1) * \log_2(\%\texttt{st} + 1)$	(Page 246)
Variants	*Flags affected*	
`fyl2xp1`	None	

fwait/wait	WAIT FOR FPU TO COMPLETE	(Page 257)
Variants	*Flags affected*	
`fwait` `wait`	None	

B.4 SIMD integer handling instructions

In the instructions described in this section, all instructions that take XMM arguments are extensions of the basic MMX technology that was introduced with SSE2 extensions in IA32 architectures. For convenience they are listed together.

emms	EMPTY MMX STATE AND MAKE X87 FPU REGISTERS FREE	(Page 304)
Variants	*Flags affected*	
`emms`	None	

maskmovdqu	STORE SELECTED BYTES FROM AN XMM REGISTER INTO MEMORY	
Variants	*Flags affected*	
`maskmovdqu xmm, xmm`	None	

Memory address is implied in `%edi`. The second argument provides the mask for the bytes of the first argument that will be stored in memory.

maskmovq	STORE SELECTED BYTES FROM AN MMX REGISTER INTO MEMORY	
Variants	*Flags affected*	
`maskmovq mm, mm`	None	

movd	MOVE 32-BIT LONG WORD	(Page 276)
Variants	*Flags affected*	

`movd rm32, mm`
`movd mm, rm32`
`movd rm32, xmm`
`movd xmm, rm32`
	None	

movdq2q	COPY 64-BIT QUAD WORD INTEGER FROM LOWER HALF OF XMM REGISTER TO MMX REGISTER	(Page 279)
Variants	*Flags affected*	
`movdq2q xmm, mm`	None	

movdqa	MOVE 128-BIT DOUBLE QUADWORD BETWEEN 16-BYTE ALIGNED MEMORY AND XMM REGISTER	(Page 279)
Variants	*Flags affected*	
`movdqa xmm, m128xmm` `movdqa m128xmm, xmm`	None	

movdqu	MOVE 128-BIT DOUBLE QUADWORD BETWEEN UNALIGNED MEMORY AND XMM REGISTER	(Page 279)
Variants	*Flags affected*	
`movdqu xmm, m128xmm` `movdqu m128xmm, xmm`	None	

movnti	NON-TEMPORAL STORE OF A 32-BIT LONG WORD FROM GENERAL-PURPOSE REGISTER INTO MEMORY	
Variants	*Flags affected*	
`movnti r32, m32`	None	

movntq

NON-TEMPORAL STORE OF A 64-BIT QUAD WORD FROM MMX REGISTER INTO MEMORY

Variants	*Flags affected*
`movntq mm, m64`	None

movq

MOVE 64-BIT QUAD WORD (Page 276)

Variants	*Flags affected*

`movq m64mm, mm`
`movq mm, m64mm`
`movq m64xmm, xmm`
`movq xmm, m64xmm`
	None

movq2dq

MOVE 64-BIT QUAD WORD FROM MMX REGISTER TO LOWER HALF OF XMM (Page 279)

Variants	*Flags affected*
`movq2dq mm, xmm`	None

packssdw

PACK MULTIPLE 32-BIT LONG WORDS INTO 16-BIT WORDS WITH SIGNED SATURATION (Page 281)

Variants	*Flags affected*
`packssdw m64mm, mm` `packssdw m128xmm, xmm`	None

packsswb

PACK MULTIPLE 16-BIT WORDS INTO BYTES WITH SIGNED SATURATION (Page 281)

Variants	*Flags affected*
`packsswb m64mm, mm` `packsswb m128xmm, xmm`	None

packuswb

PACK MULTIPLE 16-BIT WORDS INTO BYTES WITH UNSIGNED SATURATION (Page 281)

Variants	*Flags affected*
`packuswb m64mm, mm` `packuswb m128xmm, xmm`	None

paddb	ADD PACKED BYTES FROM TWO SOURCES	(Page 267)

Variants	*Flags affected*
`paddb m64mm, mm`	None
`paddb m128xmm, xmm`	

paddd	ADD PACKED 32-BIT LONG WORDS FROM TWO SOURCES	(Page 268)

Variants	*Flags affected*
`paddd m64mm, mm`	None
`paddd m128xmm, xmm`	

paddq	ADD PACKED 64-BIT QUAD WORDS FROM TWO SOURCES	(Page 268)

Variants	*Flags affected*
`paddq m64mm, mm`	None
`paddq m128xmm, xmm`	

paddsb	ADD PACKED SIGNED BYTES WITH SIGNED SATURATION	(Page 268)

Variants	*Flags affected*
`paddsb m64mm, mm`	None
`paddsb m128xmm, xmm`	

paddsw	ADD PACKED 16-BIT SIGNED WORDS WITH SIGNED SATURATION	(Page 268)

Variants	*Flags affected*
`paddsw m64mm, mm`	None
`paddsw m128xmm, xmm`	

paddusb	ADD PACKED UNSIGNED BYTES WITH UNSIGNED SATURATION	(Page 268)

Variants	*Flags affected*
`paddusb m64mm, mm`	None
`paddusb m128xmm, xmm`	

paddusw	ADD PACKED 16-BIT UNSIGNED WORDS WITH UNSIGNED SATURATION	(Page 268)

Variants	*Flags affected*
`paddusw m64mm, mm`	None
`paddusw m128xmm, xmm`	

paddw	ADD PACKED 16-BIT WORDS FROM TWO SOURCES	(Page 267)
Variants	*Flags affected*	
`paddw m64mm, mm` `paddw m128xmm, xmm`	None	

pand	PERFORM BITWISE LOGICAL AND OPERATION	(Page 272)
Variants	*Flags affected*	
`pand m64mm, mm` `pand m128xmm, xmm`	None	

pandn	PERFORM BITWISE LOGICAL AND OPERATION BETWEEN SOURCE AND BITWISE NOT OF DESTINATION	(Page 272)
Variants	*Flags affected*	
`pandn m64mm, mm` `pandn m128xmm, xmm`	None	

pavgb	COMPUTE AVERAGE OF PACKED UNSIGNED BYTES	(Page 269)
Variants	*Flags affected*	
`pavgb m64mm, mm` `pavgb m128xmm, xmm`	None	

pavgw	COMPUTE AVERAGE OF PACKED 16-BIT UNSIGNED WORDS	(Page 269)
Variants	*Flags affected*	
`pavgw m64mm, mm` `pavgw m128xmm, xmm`	None	

pcmpeqb	COMPARE PACKED BYTES FOR EQUALITY	(Page 298)
Variants	*Flags affected*	
`pcmpeqb m64mm, mm` `pcmpeqb m128xmm, xmm`	None	

pcmpeqd	COMPARE PACKED 32-BIT LONG WORDS FOR EQUALITY	(Page 298)
Variants	*Flags affected*	
`pcmpeqd m64mm, mm` `pcmpeqd m128xmm, xmm`	None	

pcmpeqw — COMPARE PACKED 16-BIT WORDS FOR EQUALITY (Page 298)

Variants	*Flags affected*
`pcmpeqw m64mm, mm` `pcmpeqw m128xmm, xmm`	None

pcmpgtb — COMPARE PACKED SIGNED BYTES FOR 'GREATER-THAN' RELATION (Page 298)

Variants	*Flags affected*
`pcmpgtb m64mm, mm` `pcmpgtb m128xmm, xmm`	None

pcmpgtd — COMPARE PACKED 32-BIT SIGNED LONG WORDS FOR 'GREATER-THAN' RELATION (Page 298)

Variants	*Flags affected*
`pcmpgtd m64mm, mm` `pcmpgtd m128xmm, xmm`	None

pcmpgtw — COMPARE PACKED 16-BIT SIGNED WORDS FOR 'GREATER-THAN' RELATION (Page 298)

Variants	*Flags affected*
`pcmpgtw m64mm, mm` `pcmpgtw m128xmm, xmm`	None

pextrw — EXTRACT SPECIFIED 16-BIT WORD FROM MMX/XMM TO GENERAL PURPOSE REGISTER (Page 292)

Variants	*Flags affected*
`pextrw im8, mm, r32` `pextrw im8, xmm, r32`	None

pinsrw — INSERT WORD FROM MEMORY OR LOWER HALF OF GENERAL PURPOSE REGISTER INTO MMX/XMM REGISTER (Page 292)

Variants	*Flags affected*

`pinsrw im8, r32, mm`
`pinsrw im8, m16, mm`
`pinsrw im8, r32, xmm`
`pinsrw im8, m16, xmm`
	None

pmaddwd — PERFORM SIGNED MULTIPLICATION AND ADDITION ON PACKED 16-BIT WORDS (Page 269)

Variants	*Flags affected*
`pmaddwd m64mm, mm` `pmaddwd m128xmm, xmm`	None

pmaxsw — COMPUTE MAXIMUM OF SIGNED PACKED 16-BIT WORDS (Page 270)

Variants	*Flags affected*
`pmaxsw m64mm, mm` `pmaxsw m128xmm, xmm`	None

pmaxub — COMPUTE MAXIMUM OF UNSIGNED PACKED BYTES (Page 270)

Variants	*Flags affected*
`pmaxub m64mm, mm` `pmaxub m128xmm, xmm`	None

pminsw — COMPUTE MINIMUM OF SIGNED PACKED 16-BIT WORDS (Page 270)

Variants	*Flags affected*
`pminsw m64mm, mm` `pminsw m128xmm, xmm`	None

pminub — COMPUTE MINIMUM OF UNSIGNED PACKED BYTES (Page 270)

Variants	*Flags affected*
`pminub m64mm, mm` `pminub m128xmm, xmm`	None

pmovmskb — MOVE SIGN MASKS OF PACKED BYTES TO GENERAL PURPOSE REGISTER (Page 280)

Variants	*Flags affected*
`pmovmskb mm, r32` `pmovmskb xmm, r32`	None

pmulhuw	PERFORM UNSIGNED MULTIPLICATION ON PACKED 16-BIT WORDS AND STORE HIGH RESULT	(Page 268)

Variants	*Flags affected*
`pmulhuw m64mm, mm` `pmulhuw m128xmm, xmm`	None

pmulhw	PERFORM SIGNED MULTIPLICATION ON PACKED 16-BIT WORDS AND STORE HIGH RESULT	(Page 268)

Variants	*Flags affected*
`pmulhw m64mm, mm` `pmulhw m128xmm, xmm`	None

pmullw	PERFORM SIGNED MULTIPLICATION ON PACKED 16-BIT WORDS AND STORE LOW RESULT	(Page 268)

Variants	*Flags affected*
`pmullw m64mm, mm` `pmullw m128xmm, xmm`	None

pmuludq	PERFORM UNSIGNED MULTIPLICATION ON PACKED 32-BIT WORDS AND STORE 64-BIT QUAD WORD RESULT	(Page 268)

Variants	*Flags affected*
`pmuludq m64mm, mm` `pmuludq m128xmm, xmm`	None

por	PERFORM BITWISE LOGICAL OR OPERATION	(Page 272)

Variants	*Flags affected*
`por m64mm, mm` `por m128xmm, xmm`	None

psadbw	COMPUTE 16-BIT SUM OF ABSOLUTE DIFFERENCES OF PACKED BYTES	(Page 270)

Variants	*Flags affected*
`psadbw m64mm, mm` `psadbw m128xmm, xmm`	None

pshufd	SHUFFLE PACKED 32-BIT LONG WORDS	(Page 292)
Variants	*Flags affected*	
`pshufd im8, m128xmm, xmm`	None	

pshufhw	SHUFFLE PACKED 16-BIT HIGH WORDS	(Page 292)
Variants	*Flags affected*	
`pshufhw im8, m128xmm, xmm`	None	

pshuflw	SHUFFLE PACKED 16-BIT LOW WORDS	(Page 292)
Variants	*Flags affected*	
`pshuflw im8, m128xmm, xmm`	None	

pshufw	SHUFFLE PACKED 16-BIT WORDS IN MMX REGISTER	(Page 292)
Variants	*Flags affected*	
`pshufw im8, m64mm, mm`	None	

pslld	PERFORM LOGICAL LEFT SHIFT ON PACKED 32-BIT LONG WORDS	(Page 274)
Variants	*Flags affected*	

`pslld im8, mm`
`pslld m64mm, mm`
`pslld im8, xmm`
`pslld m128xmm, xmm`
	None	

pslldq	PERFORM LOGICAL LEFT SHIFT (SHIFT COUNT IN BYTES) ON 128-BIT DOUBLE QUADWORD	(Page 274)
Variants	*Flags affected*	
`pslldq im8, xmm`	None	

psllq	PERFORM LOGICAL LEFT SHIFT ON PACKED 64-BIT QUAD WORDS	(Page 274)
Variants	*Flags affected*	

`psllq im8, mm`
`psllq m64mm, mm`
`psllq im8, xmm`
`psllq m128xmm, xmm`
	None	

psllw — PERFORM LOGICAL LEFT SHIFT ON PACKED 16-BIT WORDS (Page 274)

Variants
```
psllw im8, mm
psllw m64mm, mm
psllw im8, xmm
psllw m128xmm, xmm
```
Flags affected
None

psrad — PERFORM RIGHT ARITHMETIC SHIFT ON PACKED 32-BIT LONG WORDS (Page 274)

Variants
```
psrad im8, mm
psrad m64mm, mm
psrad im8, xmm
psrad m128xmm, xmm
```
Flags affected
None

psraw — PERFORM RIGHT ARITHMETIC SHIFT ON PACKED 16-BIT WORDS (Page 274)

Variants
```
psraw im8, mm
psraw m64mm, mm
psraw im8, xmm
psraw m128xmm, xmm
```
Flags affected
None

psrld — PERFORM RIGHT LOGICAL SHIFT ON PACKED 32-BIT LONG WORDS (Page 274)

Variants
```
psrld im8, mm
psrld m64mm, mm
psrld im8, xmm
psrld m128xmm, xmm
```
Flags affected
None

psrldq — PERFORM LOGICAL RIGHT SHIFT (SHIFT COUNT IN BYTES) ON 128-BIT DOUBLE QUADWORD (Page 274)

Variants
```
psrldq im8, xmm
```
Flags affected
None

psrlq	PERFORM RIGHT LOGICAL SHIFT ON PACKED 64-BIT QUAD WORDS	(Page 274)

Variants	*Flags affected*

`psrlq im8, mm`
`psrlq m64mm, mm`
`psrlq im8, xmm`
`psrlq m128xmm, xmm`
	None

psrlw	PERFORM RIGHT LOGICAL SHIFT ON PACKED 16-BIT WORDS	(Page 274)

Variants	*Flags affected*

`psrlw im8, mm`
`psrlw m64mm, mm`
`psrlw im8, xmm`
`psrlw m128xmm, xmm`
	None

psubb	PERFORM SUBTRACTION ON PACKED BYTES FROM TWO SOURCES	(Page 268)

Variants	*Flags affected*
`psubb m64mm, mm` `psubb m128xmm, xmm`	None

psubd	PERFORM SUBTRACTION ON PACKED 32-BIT LONG WORDS FROM TWO SOURCES	(Page 268)

Variants	*Flags affected*
`psubd m64mm, mm` `psubd m128xmm, xmm`	None

psubq	PERFORM SUBTRACTION ON PACKED 32-BIT LONG WORDS FROM TWO SOURCES	(Page 268)

Variants	*Flags affected*
`psubq m64mm, mm` `psubq m128xmm, xmm`	None

psubsb	PERFORM SUBTRACTION ON PACKED SIGNED BYTES WITH SIGNED SATURATION	(Page 268)

Variants	*Flags affected*
`psubsb m64mm, mm` `psubsb m128xmm, xmm`	None

psubsw — PERFORM SUBTRACTION ON PACKED 16-BIT SIGNED WORDS WITH SIGNED SATURATION (Page 268)

Variants	*Flags affected*
`psubsw m64mm, mm` `psubsw m128xmm, xmm`	None

psubusb — PERFORM SUBTRACTION ON PACKED UNSIGNED BYTES WITH UNSIGNED SATURATION (Page 268)

Variants	*Flags affected*
`psubusb m64mm, mm` `psubusb m128xmm, xmm`	None

psubusw — PERFORM SUBTRACTION ON PACKED 16-BIT UNSIGNED WORDS WITH UNSIGNED SATURATION (Page 268)

Variants	*Flags affected*
`psubusw m64mm, mm` `psubusw m128xmm, xmm`	None

psubw — PERFORM SUBTRACTION ON PACKED 16-BIT WORDS FROM TWO SOURCES (Page 268)

Variants	*Flags affected*
`psubw m64mm, mm` `psubw m128xmm, xmm`	None

punpckhbw — UNPACK AND INTERLEAVE HIGH-ORDER BYTES (Page 281)

Variants	*Flags affected*
`punpckhbw m64mm, mm` `punpckhbw m128xmm, xmm`	None

punpckhdq — UNPACK AND INTERLEAVE HIGH-ORDER 32-BIT LONG WORDS (Page 281)

Variants	*Flags affected*
`punpckhdq m64mm, mm` `punpckhdq m128xmm, xmm`	None

punpckhqdq — UNPACK AND INTERLEAVE HIGH-ORDER 64-BIT QUAD WORDS (Page 281)

Variants	*Flags affected*
`punpckhqdq m128xmm, xmm`	None

punpckhwd — UNPACK AND INTERLEAVE HIGH-ORDER 16-BIT WORDS (Page 281)

Variants	*Flags affected*
`punpckhwd m64mm, mm`	None
`punpckhwd m128xmm, xmm`	

punpcklbw — UNPACK AND INTERLEAVE LOW-ORDER BYTES (Page 281)

Variants	*Flags affected*
`punpcklbw m64mm, mm`	None
`punpcklbw m128xmm, xmm`	

punpckldq — UNPACK AND INTERLEAVE LOW-ORDER 32-BIT LONG WORDS (Page 281)

Variants	*Flags affected*
`punpckldq m64mm, mm`	None
`punpckldq m128xmm, xmm`	

punpcklqdq — UNPACK AND INTERLEAVE LOW-ORDER 64-BIT QUAD WORDS (Page 281)

Variants	*Flags affected*
`punpcklqdq m128xmm, xmm`	None

punpcklwd — UNPACK AND INTERLEAVE LOW-ORDER 16-BIT WORDS (Page 281)

Variants	*Flags affected*
`punpcklwd m64mm, mm`	None
`punpcklwd m128xmm, xmm`	

pxor — PERFORM BITWISE EXCLUSIVE OR OPERATION (Page 273)

Variants	*Flags affected*
`pxor m64mm, mm`	None
`pxor m128xmm, xmm`	

B.5 SIMD instructions for efficiency

clflush	FLUSH AND INVALIDATE A CACHE LINE THAT CONTAINS m8
Variants	*Flags affected*
`clflush m8`	None

lfence	MEMORY LOAD FENCE
Variants	*Flags affected*
`lfence`	None

No further load will take place till all pending memory load operations are completed.

mfence	MEMORY FENCE
Variants	*Flags affected*
`mfence`	None

No further memory operation (load or store) will take place till all pending memory operations are completed.

pause	PROVIDE A HINT TO IMPROVE THE PERFORMANCE OF SPIN-WAIT LOOPS
Variants	*Flags affected*
`pause`	None

prefetch	PREFETCH DATA INTO CACHE
Variants	*Flags affected*

`prefetcht0 m8`
`prefetcht1 m8`
`prefetcht2 m8`
`prefetchnta m8`
	None

These instructions initiate bringing data close to the processor so that subsequence memory read operations possibly result in a cache hit.

sfence	STORE FENCE
Variants	*Flags affected*
`sfence`	None

No further store will take place till all pending memory store operations are competed.

B.6 SIMD floating point instructions

addpd	ADD PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 302)
Variants	*Flags affected*	
`addpd m128xmm, xmm`	None	

addps	ADD PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 302)
Variants	*Flags affected*	
`addps m128xmm, xmm`	None	

addsd	ADD SCALAR DOUBLE PRECISION FLOATING POINT NUMBER	(Page 303)
Variants	*Flags affected*	
`addsd m64xmm, xmm`	None	

addss	ADD SCALAR SINGLE PRECISION FLOATING POINT NUMBER	(Page 303)
Variants	*Flags affected*	
`addss m32xmm, xmm`	None	

andnpd	PERFORM BITWISE LOGICAL AND OPERATION BETWEEN SOURCE AND NOT OF DESTINATION CONTAINING PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 273)
Variants	*Flags affected*	
`andnpd m128xmm, xmm`	None	

andnps	PERFORM BITWISE LOGICAL AND OPERATION BETWEEN SOURCE AND NOT OF DESTINATION CONTAINING PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 273)
Variants	*Flags affected*	
`andnps m128xmm, xmm`	None	

andpd	PERFORM BITWISE LOGICAL AND OPERATION OF PACKED DOUBLE PRECISION FLOATING POINT VALUES	(Page 273)
Variants	*Flags affected*	
`andpd m128xmm, xmm`	None	

andps	PERFORM BITWISE LOGICAL AND OPERATION OF PACKED SINGLE PRECISION FLOATING POINT VALUES	(Page 273)
Variants	*Flags affected*	
`andps m128xmm, xmm`	None	

cmpeqpd	COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpeqpd m128xmm, xmm`	None	

cmpeqps	COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpeqps m128xmm, xmm`	None	

cmpeqsd	COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpeqsd m64xmm, xmm`	None	

cmpeqss	COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpeqss m32xmm, xmm`	None	

cmplepd	COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'LESS-OR-EQUAL'	(Page 299)
Variants	*Flags affected*	
`cmplepd m128xmm, xmm`	None	

cmpleps — COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'LESS-OR-EQUAL' (Page 299)

Variants	*Flags affected*
`cmpleps m128xmm, xmm`	None

cmplesd — COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'LESS-OR-EQUAL' (Page 299)

Variants	*Flags affected*
`cmplesd m64xmm, xmm`	None

cmpless — COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'LESS-OR-EQUAL' (Page 299)

Variants	*Flags affected*
`cmpless m32xmm, xmm`	None

cmpltpd — COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'LESS-THAN' (Page 299)

Variants	*Flags affected*
`cmpltpd m128xmm, xmm`	None

cmpltps — COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'LESS-THAN' (Page 299)

Variants	*Flags affected*
`cmpltps m128xmm, xmm`	None

cmpltsd — COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'LESS-THAN' (Page 299)

Variants	*Flags affected*
`cmpltsd m64xmm, xmm`	None

cmpltss	COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'LESS-THAN'	(Page 299)
Variants	*Flags affected*	
`cmpltss m32xmm, xmm`	None	

cmpneqpd	COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpneqpd m128xmm, xmm`	None	

cmpneqps	COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'NOT-EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpneqps m128xmm, xmm`	None	

cmpneqsd	COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpneqsd m64xmm, xmm`	None	

cmpneqss	COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpneqss m32xmm, xmm`	None	

cmpnlepd	COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-LESS-THAN-OR-EQUAL-TO'	(Page 299)
Variants	*Flags affected*	
`cmpnlepd m128xmm, xmm`	None	

cmpnleps — COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'NOT-LESS-THAN-OR-EQUAL-TO' (Page 299)

Variants	*Flags affected*
`cmpnleps m128xmm, xmm`	None

cmpnlesd — COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-LESS-THAN-OR-EQUAL-TO' (Page 299)

Variants	*Flags affected*
`cmpnlesd m64xmm, xmm`	None

cmpnless — COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-LESS-THAN-OR-EQUAL-TO' (Page 299)

Variants	*Flags affected*
`cmpnless m32xmm, xmm`	None

cmpnltpd — COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-LESS-THAN' (Page 299)

Variants	*Flags affected*
`cmpnltpd m128xmm, xmm`	None

cmpnltps — COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'NOT-LESS-THAN' (Page 299)

Variants	*Flags affected*
`cmpnltps m128xmm, xmm`	None

cmpnltsd — COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-LESS-THAN' (Page 299)

Variants	*Flags affected*
`cmpnltsd m64xmm, xmm`	None

cmpnltss	COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'NOT-LESS-THAN'	(Page 299)

Variants	*Flags affected*
`cmpnltss m32xmm, xmm`	None

cmpordpd	COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'ORDERED'	(Page 299)

Variants	*Flags affected*
`cmpordpd m128xmm, xmm`	None

cmpordps	COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'ORDERED'	(Page 299)

Variants	*Flags affected*
`cmpordps m128xmm, xmm`	None

cmpordsd	COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'ORDERED'	(Page 299)

Variants	*Flags affected*
`cmpordsd m64xmm, xmm`	None

cmpordss	COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'ORDERED'	(Page 299)

Variants	*Flags affected*
`cmpordss m32xmm, xmm`	None

cmpunordpd	COMPARE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'UNORDERED'	(Page 299)

Variants	*Flags affected*
`cmpunordpd m128xmm, xmm`	None

cmpunordps	COMPARE PACKED SINGLE PRECISION FLOATING POINT NUMBER FOR RELATION 'UNORDERED'	(Page 299)

Variants	*Flags affected*
`cmpunordps m128xmm, xmm`	None

cmpunordsd	COMPARE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'UNORDERED'	(Page 299)

Variants	*Flags affected*
`cmpunordsd m64xmm, xmm`	None

cmpunordss	COMPARE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS FOR RELATION 'UNORDERED'	(Page 299)

Variants	*Flags affected*
`cmpunordss m32xmm, xmm`	None

comisd	PERFORM ORDERED COMPARISON OF SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS AND SET FLAGS IN `eflags` REGISTER	(Page 299)

Variants	*Flags affected*
`comisd m128xmm, xmm`	ZF, PF, CF according to comparison. All others are unaffected.

comiss	PERFORM ORDERED COMPARISON OF SCALAR SINGLE PRECISION FLOATING POINT NUMBERS AND SET FLAGS IN `eflags` REGISTER	(Page 299)

Variants	*Flags affected*
`comiss m128xmm, xmm`	ZF, PF, CF according to comparison. All others are unaffected.

cvtdq2pd	CONVERT PACKED SIGNED 32-BIT LONG WORD INTEGERS TO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 286)
Variants	*Flags affected*	
`cvtdq2pd m128xmm, xmm`	None	

cvtdq2ps	CONVERT PACKED SIGNED 32-BIT LONG WORD INTEGERS TO PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 286)
Variants	*Flags affected*	
`cvtdq2ps m128xmm, xmm`	None	

cvtpd2dq	CONVERT PACKED DOUBLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)
Variants	*Flags affected*	
`cvtpd2dq m128xmm, xmm`	None	

cvtpd2pi	CONVERT PACKED DOUBLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS.	(Page 286)
Variants	*Flags affected*	
`cvtpd2pi m128xmm, mm`	None	

cvtpd2ps	CONVERT PACKED DOUBLE PRECISION FLOATING POINT NUMBERS TO PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 286)
Variants	*Flags affected*	
`cvtpd2ps m128xmm, xmm`	None	

cvtpi2pd	CONVERT PACKED SIGNED 32-BIT LONG WORD INTEGERS TO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 286)
Variants	*Flags affected*	
`cvtpi2pd m64mm, xmm`	None	

cvtpi2ps	CONVERT PACKED SIGNED 32-BIT LONG WORD INTEGERS TO PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 286)

Variants	*Flags affected*
`cvtpi2ps m64mm, xmm`	None

cvtps2dq	CONVERT PACKED SINGLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)

Variants	*Flags affected*
`cvtps2dq m128xmm, xmm`	None

cvtps2pd	CONVERT PACKED SINGLE PRECISION FLOATING POINT NUMBERS TO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 286)

Variants	*Flags affected*
`cvtps2pd m128xmm, xmm`	None

cvtps2pi	CONVERT PACKED SINGLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)

Variants	*Flags affected*
`cvtps2pi m128xmm, mm`	None

cvtsd2si	CONVERT SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS TO A SIGNED 32-BIT LONG WORD INTEGER	(Page 286)

Variants	*Flags affected*
`cvtsd2si m64xmm, r32`	None

cvtsd2ss	CONVERT SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS TO SCALAR SINGLE PRECISION FLOATING POINT NUMBERS	(Page 286)

Variants	*Flags affected*
`cvtsd2ss m128xmm, xmm`	None

cvtsi2sd	CONVERT SIGNED 32-BIT LONG WORD INTEGER TO SCALAR DOUBLE PRECISION FLOATING-POINT VALUE	(Page 286)
Variants	*Flags affected*	
`cvtsi2sd rm32, xmm`	None	

cvtsi2ss	CONVERT SIGNED 32-BIT LONG WORD INTEGER TO SCALAR SINGLE PRECISION FLOATING-POINT VALUE	(Page 286)
Variants	*Flags affected*	
`cvtsi2ss rm32, xmm`	None	

cvtss2sd	CONVERT SCALAR SINGLE PRECISION FLOATING POINT NUMBERS TO SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 286)
Variants	*Flags affected*	
`cvtss2sd m32xmm, xmm`	None	

cvtss2si	CONVERT A SCALAR SINGLE PRECISION FLOATING-POINT VALUE TO A SIGNED 32-BIT LONG WORD INTEGER	(Page 286)
Variants	*Flags affected*	
`cvtss2si m32xmm, r32`	None	

cvttpd2dq	CONVERT WITH TRUNCATION PACKED DOUBLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)
Variants	*Flags affected*	
`cvttpd2dq m128xmm, xmm`	None	

cvttpd2pi	CONVERT WITH TRUNCATION PACKED DOUBLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)
Variants	*Flags affected*	
`cvttpd2pi m128xmm, mm`	None	

cvttps2dq	CONVERT WITH TRUNCATION PACKED SINGLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)

Variants	*Flags affected*
`cvttps2dq m128xmm, xmm`	None

cvttps2pi	CONVERT WITH TRUNCATION PACKED SINGLE PRECISION FLOATING POINT NUMBERS TO PACKED SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)

Variants	*Flags affected*
`cvttps2pi m64xmm, xmm`	None

cvttsd2si	CONVERT WITH TRUNCATION SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS TO SCALAR SIGNED 32-BIT LONG WORD INTEGERS	(Page 286)

Variants	*Flags affected*
`cvttsd2si m64xmm, r32`	None

cvttss2si	CONVERT WITH TRUNCATION A SCALAR SINGLE PRECISION FLOATING-POINT VALUE TO A SCALAR SIGNED 32-BIT LONG WORD INTEGER	(Page 286)

Variants	*Flags affected*
`cvttss2si m32xmm, r32`	None

divpd	DIVIDE PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 302)

Variants	*Flags affected*
`divpd m128xmm, xmm`	None

divps	DIVIDE PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 302)

Variants	*Flags affected*
`divps m128xmm, xmm`	None

divsd	DIVIDE SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`divsd m64xmm, xmm`	None	

divss	DIVIDE SCALAR SINGLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`divss m32xmm, xmm`	None	

ldmxcsr	LOAD MXCSR REGISTER FROM MEMORY	(Page 305)
Variants	*Flags affected*	
`ldmxcsr m32`	None	

maxpd	COMPUTE MAXIMUM IN PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`maxpd m128xmm, xmm`	None	

maxps	COMPUTE MAXIMUM IN PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`maxps m128xmm, xmm`	None	

maxsd	COMPUTE MAXIMUM IN SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`maxsd m64xmm, xmm`	None	

maxss	COMPUTE MAXIMUM IN SCALAR SINGLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`maxss m32xmm, xmm`	None	

minpd — COMPUTE MINIMUM IN PACKED DOUBLE PRECISION FLOATING POINT NUMBERS (Page 303)

Variants	*Flags affected*
`minpd m128xmm, xmm`	None

minps — COMPUTE MINIMUM IN PACKED SINGLE PRECISION FLOATING POINT NUMBERS (Page 303)

Variants	*Flags affected*
`minps m128xmm, xmm`	None

minsd — COMPUTE MINIMUM IN SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS (Page 303)

Variants	*Flags affected*
`minsd m64xmm, xmm`	None

minss — COMPUTE MINIMUM IN SCALAR SINGLE PRECISION FLOATING POINT NUMBERS (Page 303)

Variants	*Flags affected*
`minss m32xmm, xmm`	None

movapd — MOVE TWO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS BETWEEN XMM REGISTERS OR BETWEEN XMM REGISTER AND ALIGNED MEMORY (Page 278)

Variants	*Flags affected*
`movapd m128xmm, xmm` `movapd xmm, m128xmm`	None

movaps — MOVE FOUR PACKED SINGLE PRECISION FLOATING POINT NUMBERS BETWEEN XMM REGISTERS OR BETWEEN XMM REGISTER AND ALIGNED MEMORY (Page 278)

Variants	*Flags affected*
`movaps m128xmm, xmm` `movaps xmm, m128xmm`	None

movhlps	MOVE TWO PACKED SINGLE PRECISION FLOATING POINT NUMBERS FROM UPPER HALF OF SOURCE TO THE LOWER HALF OF DESTINATION	(Page 279)

Variants	*Flags affected*
`movhlps xmm, xmm`	None

movhpd	MOVE DOUBLE PRECISION FLOATING POINT NUMBER BETWEEN UPPER HALF OF XMM REGISTER AND MEMORY	(Page 277)

Variants	*Flags affected*
`movhpd m64, xmm` `movhpd xmm, m64`	None

movhps	MOVE TWO SINGLE PRECISION FLOATING POINT NUMBERS BETWEEN UPPER HALF OF XMM REGISTER AND MEMORY	(Page 277)

Variants	*Flags affected*
`movhps m64, xmm` `movhps xmm, m64`	None

movlhps	MOVE TWO PACKED SINGLE PRECISION FLOATING POINT NUMBERS FROM LOWER HALF OF SOURCE TO THE UPPER HALF OF DESTINATION	(Page 279)

Variants	*Flags affected*
`movlhps xmm, xmm`	None

movlpd	MOVE DOUBLE PRECISION FLOATING POINT NUMBER BETWEEN LOWER HALF OF XMM REGISTER AND MEMORY	(Page 277)

Variants	*Flags affected*
`movlpd m64, xmm` `movlpd xmm, m64`	None

movlps	MOVE TWO PACKED SINGLE PRECISION FLOATING POINT NUMBERS FROM LOWER HALF OF SOURCE TO THE LOWER HALF OF DESTINATION	(Page 277)
Variants	*Flags affected*	
`movlps m64, xmm` `movlps xmm, m64`	None	

movmskpd	EXTRACT SIGN BITS FROM TWO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS IN XMM REGISTER TO GENERAL PURPOSE REGISTER	(Page 280)
Variants	*Flags affected*	
`movmskpd xmm, r32`	None	

movmskps	EXTRACT SIGN BITS FROM FOUR PACKED SINGLE PRECISION FLOATING POINT NUMBERS IN XMM REGISTER TO GENERAL PURPOSE REGISTER	(Page 280)
Variants	*Flags affected*	
`movmskps xmm, r32`	None	

movntdq	NON-TEMPORAL STORE OF 128-BIT DOUBLE QUAD WORD FROM AN XMM REGISTER INTO MEMORY	
Variants	*Flags affected*	
`movntdq xmm, m128`	None	

movntpd	NON-TEMPORAL STORE OF TWO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS FROM AN XMM REGISTER INTO MEMORY	
Variants	*Flags affected*	
`movntpd xmm, m128`	None	

movntps — NON-TEMPORAL STORE OF FOUR PACKED SINGLE PRECISION FLOATING POINT NUMBERS FROM AN XMM REGISTER INTO MEMORY

Variants	*Flags affected*
`movntps xmm, m128`	None

movsd — MOVE SCALAR DOUBLE PRECISION FLOATING POINT NUMBER BETWEEN XMM REGISTERS OR BETWEEN AN XMM REGISTER AND MEMORY (Page 277)

Variants	*Flags affected*
`movsd m64xmm, xmm` `movsd xmm, m64xmm`	None

movss — MOVE SCALAR SINGLE PRECISION FLOATING POINT NUMBER BETWEEN XMM REGISTERS OR BETWEEN AN XMM REGISTER AND MEMORY (Page 277)

Variants	*Flags affected*
`movss m32xmm, xmm` `movss xmm, m32xmm`	None

movupd — MOVE TWO PACKED DOUBLE PRECISION FLOATING POINT NUMBERS BETWEEN XMM REGISTERS OR BETWEEN XMM REGISTER AND UNALIGNED MEMORY (Page 278)

Variants	*Flags affected*
`movupd m128xmm, xmm` `movupd xmm, m128xmm`	None

movups — MOVE FOUR PACKED SINGLE PRECISION FLOATING POINT NUMBERS BETWEEN XMM REGISTERS OR BETWEEN XMM REGISTER AND UNALIGNED MEMORY (Page 278)

Variants	*Flags affected*
`movups m128xmm, xmm` `movups xmm, m128xmm`	None

mulpd	MULTIPLY PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 302)
Variants	*Flags affected*	
`mulpd m128xmm, xmm`	None	

mulps	MULTIPLY PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 302)
Variants	*Flags affected*	
`mulps m128xmm, xmm`	None	

mulsd	MULTIPLY SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`mulsd m64xmm, xmm`	None	

mulss	MULTIPLY SCALAR SINGLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`mulss m32xmm, xmm`	None	

orpd	PERFORM BITWISE LOGICAL OR OPERATION OF PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 273)
Variants	*Flags affected*	
`orpd m128xmm, xmm`	None	

orps	PERFORM BITWISE LOGICAL OR OPERATION OF PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 273)
Variants	*Flags affected*	
`orps m128xmm, xmm`	None	

rcpps	COMPUTE RECIPROCALS ($\frac{1}{x}$)OF PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`rcpps m128xmm, xmm`	None	

rcpss	COMPUTE RECIPROCAL ($\frac{1}{x}$) OF SCALAR SINGLE PRECISION FLOATING POINT NUMBER	(Page 303)
Variants	*Flags affected*	
`rcpss m32xmm, xmm`	None	

rsqrtps	COMPUTE RECIPROCALS OF SQUARE ROOTS ($\frac{1}{\sqrt{x}}$) OF PACKED SINGLE PRECISION FLOATING POINT VALUES	(Page 303)
Variants	*Flags affected*	
`rsqrtps m128xmm, xmm`	None	

rsqrtss	COMPUTE RECIPROCAL OF SQUARE ROOT ($\frac{1}{\sqrt{x}}$) OF SCALAR SINGLE PRECISION FLOATING POINT NUMBER	(Page 303)
Variants	*Flags affected*	
`rsqrtss m32xmm, xmm`	None	

shufpd	SHUFFLES VALUES IN PACKED DOUBLE PRECISION FLOATING-POINT OPERANDS	(Page 292)
Variants	*Flags affected*	
`shufpd im8, m128xmm, xmm`	None	

shufps	SHUFFLES VALUES IN PACKED SINGLE PRECISION FLOATING-POINT OPERANDS	(Page 292)
Variants	*Flags affected*	
`shufps im8, m128xmm, xmm`	None	

sqrtpd	COMPUTE PACKED SQUARE ROOTS ($\sqrt{x}$) OF PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 303)
Variants	*Flags affected*	
`sqrtpd m128xmm, xmm`	None	

sqrtps — COMPUTE SQUARE ROOTS ($\sqrt{x}$) OF PACKED SINGLE PRECISION FLOATING POINT NUMBERS (Page 302)

Variants	*Flags affected*
`sqrtps m128xmm, xmm`	None

sqrtsd — COMPUTE SCALAR SQUARE ROOT ($\sqrt{x}$) OF SCALAR DOUBLE PRECISION FLOATING POINT NUMBER (Page 303)

Variants	*Flags affected*
`sqrtsd m64xmm, xmm`	None

sqrtss — COMPUTE SQUARE ROOT ($\sqrt{x}$) OF SCALAR SINGLE PRECISION FLOATING POINT NUMBER (Page 303)

Variants	*Flags affected*
`sqrtss m32xmm, xmm`	None

stmxcsr — SAVE MXCSR REGISTER IN MEMORY (Page 305)

Variants	*Flags affected*
`stmxcsr m32`	None

subpd — SUBTRACT PACKED DOUBLE PRECISION FLOATING POINT NUMBERS (Page 302)

Variants	*Flags affected*
`subpd m128xmm, xmm`	None

subps — SUBTRACT PACKED SINGLE PRECISION FLOATING POINT NUMBERS (Page 302)

Variants	*Flags affected*
`subps m128xmm, xmm`	None

subsd — SUBTRACT SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS (Page 303)

Variants	*Flags affected*
`subsd m648xmm, xmm`	None

subss	SUBTRACT SCALAR SINGLE PRECISION FLOATING POINT NUMBERS	(Page 303)

Variants	*Flags affected*
`subss m32xmm, xmm`	None

ucomisd	PERFORM UNORDERED COMPARISON OF SCALAR DOUBLE PRECISION FLOATING POINT NUMBERS AND SET FLAGS IN `eflags` REGISTER	(Page 299)

Variants	*Flags affected*
`ucomisd m64xmm, xmm`	ZF, PF, CF according to comparison. All others are unaffected.

ucomiss	PERFORM UNORDERED COMPARISON OF SCALAR SINGLE PRECISION FLOATING POINT NUMBERS AND SET FLAGS IN `eflags` REGISTER	(Page 299)

Variants	*Flags affected*
`ucomiss m32xmm, xmm`	ZF, PF, CF according to comparison. All others are unaffected.

unpckhpd	UNPACKS AND INTERLEAVES DOUBLE PRECISION FLOATING POINT NUMBERS IN THE UPPER HALVES OF OPERANDS	(Page 290)

Variants	*Flags affected*
`unpckhpd m128xmm, xmm`	None

unpckhps	UNPACKS AND INTERLEAVES SINGLE PRECISION FLOATING POINT NUMBERS IN THE UPPER HALVES OF OPERANDS	(Page 290)

Variants	*Flags affected*
`unpckhps m128xmm, xmm`	None

unpcklpd	UNPACKS AND INTERLEAVES DOUBLE PRECISION FLOATING POINT NUMBERS IN THE LOWER HALVES OF OPERANDS	(Page 290)
Variants	*Flags affected*	
`unpcklpd m128xmm, xmm`	None	

unpcklps	UNPACKS AND INTERLEAVES SINGLE PRECISION FLOATING POINT NUMBERS IN THE LOWER HALVES OF OPERANDS	(Page 290)
Variants	*Flags affected*	
`unpcklps m128xmm, xmm`	None	

xorpd	PERFORM BITWISE LOGICAL XOR OPERATION OF PACKED DOUBLE PRECISION FLOATING POINT NUMBERS	(Page 273)
Variants	*Flags affected*	
`xorpd m128xmm, xmm`	None	

xorps	PERFORM BITWISE LOGICAL XOR OPERATION OF PACKED SINGLE PRECISION FLOATING POINT NUMBERS	(Page 273)
Variants	*Flags affected*	
`xorps m128xmm, xmm`	None	

B.7 System instructions

arpl	ADJUST REQUESTED PRIVILEGE LEVEL (RPL) OF A SEGMENT SELECTOR	
Variants	*Flags affected*	
`arpl r16, rm16`	ZF=1 if the RPL was changed. 0 Otherwise. All others are not affected.	

clts	Clear 'task-switched' flag in control register `%cr0`
Variants	*Flags affected*
`clts`	None

hlt	Halt processor
Variants	*Flags affected*
`hlt`	None

invd	Invalidate cache without any write back of modified data
Variants	*Flags affected*
`invd`	None

invlpg	Invalidate TLB Entry corresponding to the page containing address `m`
Variants	*Flags affected*
`invlpg m`	None

lar	Load access rights from the segment selector
Variants	*Flags affected*
`lar rm16, r16` `lar rm32, r32`	None

lgdt	Load global descriptor table register (`%gdtr`) from memory
Variants	*Flags affected*
`lgdt m48`	None

lidt	Load interrupt descriptor table register (`%idtr`) from memory
Variants	*Flags affected*
`lidt m48`	None

lldt	LOAD LOCAL DESCRIPTOR TABLE REGISTER (`%ldtr`) FROM SEGMENT SELECTOR
Variants	*Flags affected*
`lldt rm16`	None

lmsw	LOAD MACHINE STATUS WORD IN `%cr0` FROM MEMORY
Variants	*Flags affected*
`lmsw rm16`	None

lock	LOCK BUS CYCLE FOR THE NEXT INSTRUCTION
Variants	*Flags affected*
`lock`	None

This instruction causes the next instruction to be executed atomically.

lsl	READ SEGMENT LIMIT FIELD OF THE SPECIFIED SELECTOR
Variants	*Flags affected*
`lsl rm16, r16` `lsl rm32, r32`	ZF=1 if segment limit is loaded successfully. 0 Otherwise. No other flags are affected.

ltr	LOAD TASK REGISTER (TR) FROM REGISTER/MEMORY
Variants	*Flags affected*
`ltr rm16`	None

mov	LOAD AND STORE SPECIAL REGISTERS
Variants	*Flags affected*
`mov splReg, r32` `mov r32, splReg`	None

Special registers include debug registers `%dr0` to `%dr7` and control registers `%cr0`, `%cr2`, `%cr3`, `%cr4`, `%cr8`.

rdmsr	READ MODEL SPECIFIC REGISTER (MSR) IN %edx:%eax
Variants	*Flags affected*
`rdmsr`	None

rdpmc	READ PERFORMANCE MONITORING COUNTERS (PMC) IN %edx:%eax
Variants	*Flags affected*
`rdpmc`	None

rdtsc	READ TIME STAMP COUNTER (TSC) IN %edx:%eax
Variants	*Flags affected*
`rdtsc`	None

rsm	RESUME PROGRAM AFTER SYSTEM MANAGEMENT MODE (SMM)
Variants	*Flags affected*
`rsm`	None

sgdt	STORE GLOBAL DESCRIPTOR TABLE REGISTER (GDTR) IN MEMORY
Variants	*Flags affected*
`sgdt m48`	None

sidt	STORE INTERRUPT DESCRIPTOR TABLE REGISTER (IDTR) IN MEMORY
Variants	*Flags affected*
`sidt m48`	None

sldt	STORE LOCAL DESCRIPTOR TABLE REGISTER (LDTR) IN MEMORY OR REGISTER
Variants	*Flags affected*
`sldt rm16`	None

smsw	STORE MACHINE STATUS WORD FIELD OF %cr0
Variants	*Flags affected*
`smsw rm16` `smsw r32`	None

str	STORE TASK REGISTER (TR)
Variants	*Flags affected*
`str rm16`	None

sysenter	FAST SYSTEM CALL ENTRY
Variants	*Flags affected*
`sysenter`	None

sysexit	FAST EXIT FROM A SYSTEM CALL
Variants	*Flags affected*
`sysexit`	None

verr	VERIFY SEGMENT IF IT CAN BE READ
Variants	*Flags affected*
`verr rm16`	ZF=1 if segment is readable. 0 Otherwise. Other flags are not affected.

verw	VERIFY SEGMENT IF IT CAN BE WRITTEN
Variants	*Flags affected*
`verw rm16`	ZF=1 if segment is readable. 0 Otherwise. Other flags are not affected.

wbinvd	INVALIDATE CACHE WITH WRITE BACK OF DIRTY CACHE LINES
Variants	*Flags affected*
`wbinvd`	None

wrmsr	WRITE VALUE IN `%edx:eax` TO SELECTED MODEL SPECIFIC REGISTER (MSR)
Variants	*Flags affected*
`wrmsr`	None

Appendix C

Suggested Programming Projects

1. Design a simple hardware that interfaces with parallel port on the PC and provides a sets of eight micro switches and eight LEDs. The status of micro switches are read using the input from the PC parallel port while the display of LEDs is controlled by using the output to the PC parallel port.

2. Design another simple hardware that connects one 7-segment LED display to the PC parallel port. Write a function that can print a hexadecimal digit on the 7-segment display.

3. Update the design of the previous exercise to interface four seven-segment LED displays to the parallel port of a PC. In this design, the hardware must select one 7-segment display at a time out of four and light up LEDs for that one. Next time it must select another 7-segment LED display and light up LEDs in that. This process must be iterated for all the four displays. If the displays are refreshed about 10 to 15 times in a second, the LED displays will not fluctuate.

   Much of the control is to be done from the software on the PCs. The PCs must provide first the selection and then data for LEDs. On the PC parallel port, 8-bit of output can be performed. If the output has most significant bit as 0, the remaining 7 bits are to be used for the LED data. On the other hand, if the most significant bit is 1, the 4 bits out of remaining 7 bits should be used as the LED selection.

4. Compute x^y for any general value of x and non-negative y. You may use hints in exercise 9.14 to compute x^z for z being between ± 1.0. Now y can be separated in significand and exponent using

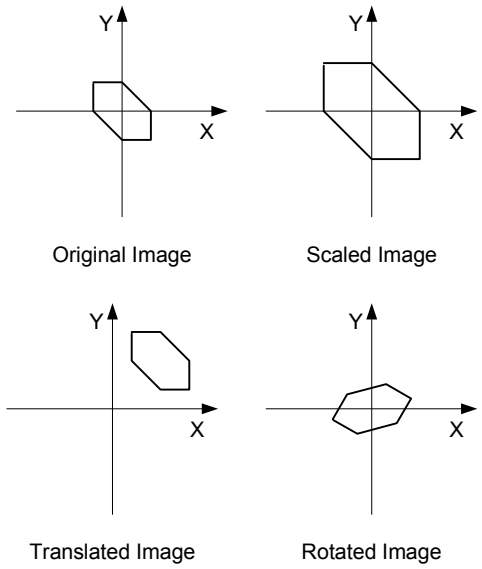


Figure C.1: Graphics operations

`fxtract` instruction. Therefore $y = s.2^e$ where s and e are significand and exponent of y respectively. Now x^y can be written as $(x^s)^{2^e}$. A number can be raised 2^p by squaring it p times.

5. Applications in graphics often use scaling, translation and rotation operations as shown in figure C.1.

   These operations are carried out using affine transformation matrix. A pixel at location (x, y) gets transformed to (x', y') where the following mathematical relation defines the transformation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s\cos\theta & s\sin\theta & t_x \\ -s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

   In the affine transformation matrix, s is the scaling factor, θ is the angle of rotation and (t_x, t_y) is the translation.

   In this exercise, you have to implement the affine transformation function in Assembly language with the following C prototype.

```
void AffineTransform(float scale, float rotate,
                     int translate[2], int pixel[2]);
```

   The arguments of the function have the following meaning. The `scale` argument represents the scaling factor s in the affine transformation matrix. The `rotate` argument is rotation angle θ in

radian. The translation system is represented by `translate` argument which provide t_x and t_y. Input pixel coordinates (x, y) are given in `pixel` argument. After execution of the function, the new location of the pixel will be return in the `pixel` argument.

Implement this program in two different ways. In the first implementation use instructions from x87 instruction set. In the second implementation, use the instructions from the SIMD instruction sets to enhance the speed of the operation.

Measure the performance improvement of the second program over the first program by calling these functions from C several times and then measuring the cumulative time using `time` command.

Appendix D

GNU Assembler

GNU Assembler supports instruction sets of multiple processors. It generates the assembly output in a variety of object file formats, the most commonly of them being the ELF. Among the processor instruction sets that are supported by the GNU Assembler, there is a commonality of the uses of assembler directives (see chapter 11) and syntax. Some of the instruction sets supported by the GNU Assembler include IA32, IA64, HP PA-RISC, MIPS, PowerPC, ARM, Sparc and 680x0.

There are some generic command switches for the Assembler while there are a few that depend upon the instruction set and the processor. In a similar way there are certain assembler directives that are processor dependent. In this book we include only the IA32 specific directives (described in chapter 11) and command switches.

D.1 Command line

The GNU assembler can be executed in a number of ways. In most systems it is available as `gas`. On the command prompt a command `gas` invokes in the GNU Assembler in such systems. In many other systems it is available as `as`. In some systems any of the two commands can be used to invoke the GNU Assembler.

Often the GNU Assembler is also invoked using the GNU compiler driver `gcc`. The GNU compiler driver `gcc` uses file extensions to identify the type of the source and then invokes appropriate tool to convert the program. It typically uses `.S` and `.s` extensions which are relevant to the Assembly language programs. The files with extension `.S` are assumed to have Assembly language programs with C preprocessor directives such as `#include` or `#define`. These files are first processed by the C preprocessor and then by the GNU Assembler. The files with extension `.s` are assumed to have Assembly language programs without C preprocessor directives and are processed by the GNU Assembler

directly.

GNU Assembler can be configured in a variety of ways using command line options. In this appendix, only those options are described which are meaningful for the IA32 Assembly language.

The following is the syntax of the command line.

```
as [ -a[cdhlns][=file] ] [ --defsym sym=val ]
 [ --gstabs ] [ --help ] [ -I dir ] [ -J ] [ -L ]
 [ --keep-locals ] [ -o objfile ] [ -R ] [ --statistics ]
 [ -v ] [ -version ] [ --version ] [ -W ] [ -Z ]
 [ -- | files ... ]
```

All arguments within `[` and `]` are optional. The following is the interpretation of the options.

`-a`: Produce assembly listings. Following finer controls are also available.

- `-ac`: omit false conditionals paths in the listing
- `-ad`: omit debugging directives in the listing
- `-ah`: include high-level source
- `-al`: include assembled bytes in the listing
- `-am`: include expansions of macros
- `-as`: include symbol information.

Optionally '`=file`' can be used to specify the file name for the generated listing. Be default the listing is produced on standard output.

Several options may be combined. For example, `-alhc` specifies `-al`, `-ah` and `-ac` together. When only `-a` is specified, it defaults to `-ahls`. When `-a` is not specified, listing is not generated.

`--defsym sym=value`: Define the symbol `sym` to `value` before assembling the input file. This option is same as using `.define` or `.set` within the source programs. The `value` must be an integer constant which can be specified in decimal, in hexadecimal (such as 0x2FF) or in octal by prefixing a zero (0) as in 0377.

`--gstabs`: Include debugging information in the generated object file in stabs format. This information can be used for the symbolic debugging of the code.

`--help`: Print a summary of the command line options and exit.

`-I dir`: Add specified directory to the search path for `.include` directives.

`-J`**:** Don't warn about signed overflow in directives such as `.int`, `.long` etc. Default setting is to give a warning whenever there is an overflow and value of the given expressions in such directives won't fit in the specified size.

`-L`**:** Include information of local symbols in the symbol table of the generated object file.

`--keep-locals`**:** Same as `-L`.

`-o objfile`**:** Produce the object code in the specified file. By default the name of the object file is `a.out`. It may be noted that this default file is not an executable file which is in contrast to what GNU linker (and `gcc`) produce.

`-R`**:** Merge data section into text section.

`--statistics`**:** Print statistics such as total size (in bytes) and total time (in seconds) used by `gas`.

`--strip-local-absolute`**:** Do not include local absolute symbols in the symbol table of generated object file.

`-v`**:** Print the version information of `gas`.

`-version`**:** Same as `-v`.

`--version`**:** Print the version information of `gas` and then exit without assembling any program.

`-W`**:** Do not print warning messages.

`-Z`**:** Generate an object file in spite of errors. By default the object file is not generated in case of errors.

`-- | files ...`**:** The names of the files to assemble. '`--`' stands for standard input. The assembler can take multiple files as input and assemble each one of them.

Appendix E

GNU Linker

GNU linker is typically available as `ld` on a GNU/Linux system. A linker is used to generate an executable file from one or more object files and possibly one or more archives (also known as libraries of codes). It is used to combine the specified object and archive files in certain ways. When an object file is generated by an assembler or by a compiler, it usually defines certain symbols such as names of the functions, names of memory locations that represent data etc. and other such definitions. It also uses some undefined names such as functions that are implemented in other object files or in a library. The compilers and assemblers usually compile functions and data in several logical organizations known as sections. Some of these sections are standard sections such as `.text`, `.data` and `.bss`. Usually compilers generate object code assuming the start point of these sections to be at offset 0. Thus it is possible to have several object files, each having certain number of bytes in a section defined in multiple files with all such definitions assuming the start address of the section to be at location 0. This is clearly a conflict because not all objects can actually be placed at location 0. To help the process of combining several object files, compilers generate enough information in the object files that include the following.

1. Symbols defined in the object file. The definition includes (at least logically) the name of the symbol, section and the offset within the section.

2. Symbols used in the object file. The definition includes the locations where a symbol is used and the name of the symbol. In addition it includes other informations such as the relocation type which is used by the linker while relocating the sections of the object files.

3. Symbols which it must get from the outside. These symbols in

an object file are those symbols which are not defined within the object file and must be defined in other object files. An executable file can be generated only when all such symbols have been defined in one of the object file or in one of the objects in the archive (or library). These symbols are also known as external symbols.

As the linker combines various object files and objects from archive files, it relocates the sections within files so as they do not conflict. This process is called relocation. Therefore during the combining process of object files, the linker relocates data and programs (usually independent of each other) in the files. As it relocates the code and data, the symbols acquire new definitions and therefore all references to such symbols (as given in the relocation table) are tidied up by the linker.

Linking is usually the last step in converting a source program to an executable program. GNU compiler driver `gcc` has intelligence built in the process where it can run the linker `ld` to generate the final executable file.

In this appendix, we describe GNU linker to understand its command line behavior. Usually the linkers can also accept the linker script files using which the linking process can be controlled in a fine manner. For example by using such scripts, it is possible to generate executable code that can be stored in a ROM in the system binding certain variables to absolute locations. Definition of such scripts is outside the scope of this book.

E.1 Command line interface

GNU linker can be invoked on the command line by executing `ld`. Several options can be specified to the linker to configure it in a number of ways. Only some command line options relevant to this book are specified in this appendix. The linker is however much more complex and supports many other modes. Interested readers are encouraged to read on-line documentations of linker which can be invoked by `info ld` on the command line or can be found on numerous web sites including those maintained by the GNU.

The general format of the command line is the following.

```
ld [-options] object-files ...  -llib-specs ...
```

Some of the options are described here.

E.1.1 General Options

`--help`: Display help information on using command-line options. The `ld` displays this information on the standard output and then exits.

`-v`: Display the version information of `ld`.

`--verbose`: Display verbose information during the linking process. The information includes version numbers, files status (whether they can be opened or not) and verbose script outputs as they are processed.

`--version`: Same as `-v` option.

E.1.2 Output Control Options

`-Bdynamic`: The linker can link object files with two kinds of libraries. Use of this option instructs the `ld` to link against dynamic libraries. This option is also the default option on GNU/Linux operating system. This option may be used multiple times on the command line. Each time it is used, it affects all those libraries which are specified by `-l` option given after `-Bdynamic`.

`-Bshareable`: Create a shared library. The output file generated by the linker becomes a library file rather than the executable file. Generated library is a shared library.

`-Bstatic`: This option indicates to the linker that it must link against the shared static libraries. It may also be used multiple times on the command line. Each time it is used, it affects all those libraries which are specified by `-l` option after `-Bstatic`.

`-Bsymbolic`: When creating a shared library (see `-Bshareable`), bind references to global symbols to the definition within the shared library, if any. Normally, it is possible for a program linked against a shared library to override the definition within the shared library.

`-e` *`entry`*: As the linker produces an executable file, it needs to know the starting address of the program. This is also known as the entry point of the program. By default a symbol by name `_start` is used as the entry point in the program (see example program 1.1). This default value can be changed by using `-e` option which configures the `ld` to use *`entry`* as the explicit starting address of the program.

In assembly programs, this symbol must be declared as `.globl` so that the symbol table information for this symbol is stored in the object file generated by the assembler. Otherwise the linker will not be able to find the definition of this symbol and will give an error.

In C programs, the starting point symbol `_start` is defined in C run time library in an object (typically `crt0.o`). This C startup

code is used to typically set up the run time environment and then to execute `main` function.

`-E`: GNU linker `ld` can create an executable file that is linked dynamically. For this purpose it uses a dynamic symbol table that contains the set of symbols visible from dynamic objects during run time. The `-E` option is used to indicate that the `ld` should add all symbols to the dynamic symbol table.

By default, the dynamic symbol table contains only those symbols which are referenced by some dynamic object mentioned in the link. The default behavior is normally good for all programs.

Some programs use explicit mechanisms to load and link library functions using `dlopen` function in the C runtime library. Since the linker does not know the name of the library that is linked using `dlopen`, it has no mechanism to find out the list of symbols that may be used by such libraries. If this library does not use any symbols defined in the program, the default behavior of the linker will be alright. However if the library may need a symbol definition from the program, this option may be used to include all symbols of the program in the dynamic symbol table.

`-i`: The GNU linker `ld` can be used to perform an incremental linking. In this case, multiple object files may be combined into a single object file with relocation and resolution of external undefined references. There may still be a few such undefined references left in the resulting object file. The resulting object file may be relocated further when it is combined with other object files to generate the executable file.

This option is same as `-r` option.

`-n`: While combining the sections, the linker sets the resulting `.text` section to be read-only if this option is specified. In certain kinds of binary file formats, the output type may be indicated by certain identifications known as magic numbers. If the output file format supports such mechanism, the output is marked as `NMAGIC`.

`-N`: When this option is given to the `ld`, it marks the resulting `.text` and `.data` sections readable as well as writable. If output file format also supports the magic number based identification, the output is marked as `OMAGIC`.

`-o` *`output-file-name`*: The name of the output file generated by the linker can be be specified using `-o` option. By default the name of the output file is `a.out` but can be changed with this option.

`-r`: This option is same as `-i` option for incremental linking. It is used to indicate to the linker that it must generate relocatable output

(that is to generate an object file that can in turn serve as input to subsequent runs of `ld`).

`-s`: Strip off all symbol information as the output file is created.

`-S`: Strip off all debugger symbol information while retaining information for other symbols as the output file is created.

`-shared`: This option is same as `-Bshareable`.

`-static`: This option is same as `-Bstatic`.

`-Tbss` *org*: Use *org* as the starting address for the `.bss` section of the output file. *org* must be a single hexadecimal number with or without leading '`0x`'.

`-Tdata` *org*: Use *org* as the starting address for the `.data` section of the output file. *org* must be a single hexadecimal number with or without leading '`0x`'.

`-Ttext` *org*: Use *org* as the starting address for the `.text` section of the output file. *org* must be a single hexadecimal number with or without leading '`0x`'.

`-x`: Delete information about all local symbols. Local symbols are those symbols that are not declared `.globl` in an Assembly language program.

`-X`: Delete all temporary local symbols. The symbols are temporary local symbols if their names start with '`.L`'.

E.1.3 Output Format Options

`--defsym symbol=expression`: Generate a new definition of a global `symbol` in the output file. The value for this symbol is an absolute address given by `expression`. This option may be used many times to define multiple symbols in the command line. Only a limited form of arithmetic is understood by the linker in the use of the expression. It may comprise of hexadecimal constants, the name of existing symbols, '+' and '−' to add or subtract hexadecimal constants or symbols.

More elaborate expressions may also be specified using linker scripts description is which is out of scope of this book.

`--oformat output-format`: `ld` may be configured to support more than one kind of object file. In such a case, `--oformat` option is used to specify the specific binary file format for the output object file. This option is usually not used even when `ld` is configured to

support alternative object formats. When this option is not specified, the `ld` produces a default output format which is usually acceptable in most situations.

`-u symbol`: This option is used to force `symbol` to be marked as an undefined symbol while generating the output file. This option along with the incremental linking may be used in certain ways. For example there may be more than one implementation of a single symbol. A few files may use one definition while other files may use the other definition. Using the incremental linking process, an object file may be created with files that use one definition. Later this symbol may be marked as undefined and during the subsequent linking it will force the second definition of the symbol be linked.

E.1.4 Library Search Options

`-larchive`: The GNU linker `ld` uses an elaborate mechanism to locate library files. Using `-l` option a name of the archive is specified. This name is added to the list of files that are being linked together.

`ld` uses a search mechanism to locate the named archive. For this purpose it maintains a list of directories, also known as path-list. For each archive specified using `-l` option, the linker searches its path-list for the library file that represents the named archive. If the linker supports the shared libraries (default behavior but may be changed with `-static`) it looks for the library file with an extension `.so`. If the shared library is not available or if the static libraries are linked explicitly, the library file with an extension of '`.a`' is looked for.

The file names of the library that are searched for a given named archive *name* are `lib`*name*`.so`, *name*`.so`, `lib`*name*`.a` or *name*`.so`.

An archive is a collection of several object files with an additional directory of symbols to speed up the search for a name. Such object files included in the archive are also called the archive members.

The `-l` option may be used multiple times to specify multiple archives.

`-Ldir`: Add the specified directory `dir` to the path-list used for searching library archives. The linker also maintains a default path-list which is usually sufficient for most applications. This option may be used multiple times for including multiple directories in the path-list. When a directory is specified using `-L` option, it

is added in the order in which these are specified on the command line. The default path-list is then added to the end of this path-list. The archives are therefore searched in the directories specified on the command line (in the order they are specified on the command line) followed by the default search path.

For example consider the following options specified on the command line.

```
-lc -L/usr/lib -L/usr/local/lib
```

These options cause the library file `libc.so` to be searched in the directories `/usr/lib` and then in `/usr/local/lib` and then in the default search path. The search will stop as soon as the file is found.

E.1.5 Options for analyzing link process

`--cref`**:** Output a cross reference table. If a linker map file is being generated (see `-M` and `-Map` options below), the cross reference table is printed to the map file. Otherwise, it is printed on the standard output. The cross reference table provides the following information for each symbol.

1. The file in which the symbol was defined.
2. List of files in which the symbol was used.

The output includes a list of file names for each symbol. In this list, the first file listed is the location of the definition. The remaining files contain references to the symbol.

`-M`**:** Linker may be configured by this option to provide the mapping information of the symbols. The mapping information is printed on the standard output and includes the following.

- The memory locations where the object files and symbols are mapped.
- The memory locations to define how the common symbols (symbols defined using `.comm` directive in the assembler) are allocated.
- All archive members (objects in the library files) included in the link along with the name of the symbol which caused the archive member to be linked in.

`-Map mapfile`**:** Print a linker mapping information of the symbols in file specified as `mapfile`. Also see `-M` option above.

`--stats`: Compute and display statistics about the operations of the linker. The statistics includes items such as execution time and memory usage.

`-t`: Print the names of the input files as `ld` processes them.

`-y symbol`: Print the name of each linked file in which symbol appears. This option may be given any number of times each time possibly for a different symbol. This option is useful when the linker reports an undefined symbol during the linking process but it is not clear where this reference is coming from.

E.1.6 Options for Link process control

`--no-warn-mismatch`: Normally `ld` gives an error if mismatched input files are linked together. The mismatch may occur for several reasons such as when two object files have been compiled for different processors. This option forces `ld` to link such mismatched files together without giving any errors. This option should only be used with care as the resulting file may be incorrect to execute.

`-T scriptfile`: The linker may be configured to use a file that contains the linker commands. This option is used to configure linker to read link commands from linker script file `scriptfile`. The description of the linker script files, syntax and commands that may be used in the linker script files is out of the scope of this book.

`--warn-once`: Only warn once for each undefined symbol, rather than once per module which refers to it.

`--warn-section-align`: Warn if the address of an output section is changed because of alignment.

Appendix F

GNU Debugger

A debugger is used to trace a program during its runtime to identify possible errors in the program. These errors can then be corrected in the source programs.

During the trace of the programs, debuggers permit a user to look at values of program variables. If some values are found to be wrongly computed, they can even be corrected and the execution trace may be resumed to find further errors in the program.

Debuggers run in one of the following two modes.

1. Interactive mode. In this mode, the control is with the debugger. The user program does not run and is in stopped state. Various debugger commands can be given in this mode. These include commands to inspect variables of the program, modify their values, look at the source code of the program and to see the state of the machine. Debugger switches into program execution mode when a command is given to resume the program execution. Initially, the debugger starts in interactive mode.

2. Program execution mode. In this mode, the control is with the user program. The user program continues to execute till it hits a breakpoint condition and stops. The debugger then gets the control and enters interactive mode.

The debugger provides several commands which operate on program lines, variables, function names etc. These are usually referred to by their names rather than some address in the memory. Debuggers therefore need to know the mapping between such symbols and the corresponding addresses in memory. This information is usually provided by the compilers and assemblers and is carried into the final executable file by the linker. The debuggers then use this information in the executable file to simplify the process of symbolic debugging.

Often the user tries to find out the memory contents using programming language expressions. For example, in order to find the value of a memory whose address is available in a pointer type variable in the program, it will be easier to specify expressions such as '`*p`' to find the values. These kind of expressions are different for various programming languages. Debuggers therefore provide support for various programming languages.

The GNU debugger is used to debug programs written in Assembly language and other high level languages such as C, C++, Fortran etc. In this appendix, certain commands and methods are described to debug an Assembly language program using GNU debugger.

F.1 Compiling for a debug session

As mentioned earlier, the executable file must be compiled separately for a debug session. In this process, the symbol information must be carried into the executable file. In addition, line information of the source files must also be carried in the executable file.

There are two different ways to compile an Assembly language program for debugging.

1. The easiest way to compile a program with debug information is with GNU compiler front end `gcc`. For this to work, the filename extension of the file must be '`.S`' or '`.s`'. Files with extensions of a upper case '`.S`' are preprocessed by the C pre-processor `cpp` before running though the GNU assembler. Files with extensions of a lower case '`.s`' are directly processed by the GNU assembler `gas`.

   Command line for `gcc` must include `-g` option. The following is an example to compile an Assembly language program `prog.S`.

   ```
   gcc -g prog.S
   ```

   This command generates an executable file `a.out` which can be run under debugger control.

2. Another method of compiling Assembly language programs is to run them through `as` and then link through `ld`. The following is the sequence of commands to generate an executable code that can be debugged.

   ```
   as --gstabs -o prog.o prog.s
   ld -o a.out prog.o
   ```

   '`--gstabs`' option on the command line of `as` specifies that the assembler must generate debug information in stabs format which is used by the GNU debugger `gdb`. "`-o`" options on two command

lines provide the names of the output files for that particular process. For example, the object file generated by the `as` is given a name `prog.o` while linker generated executable file is given a name `a.out`.

Multiple files may each be assembled in a similar manner. All object files along with the debug information generated in this process may be linked together using `ld` command. The linker then provides debug information of each source file in the final executable.

F.2 Starting a debug session

The `gdb` is primarily a command line tool. It is invoked using `gdb` on the command line. The user executable program that is being debugged can be given as a command line argument to the `gdb`.

The following is an example to start the `gdb` session for a program which had been previously compiled/assembled and linked using `gcc` or `as` and `ld`. In this example, it is assumed that the executable file name is `helloworld` generated from the code in figure 1.1.

```
gdb helloworld
```

In response to this command, the `gdb` program is initiated and it enters the interactive mode. Following would be a typical screen shot of running `gdb`.

```
GNU gdb 6.0-2mdk (Mandrake Linux)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb)
```

At the prompt `(gdb)`, the debugger waits for the user to type in commands to execute. At this point the program can be executed using `run` command. The following is an example of running this progam.

```
(gdb) run
Starting program: /home/moona/assembly/Programs/Chap1/fig1.1/helloworld
hello world

Program exited normally.
```

A general syntax for the `run` command under `gdb` includes possibilities for I/O redirection by < and > symbols, giving command line arguments for the program right after `run` command and other such shell specific behaviors.

When a `run` command is given, the debugger moves to program execution mode. When the program is terminated, the control is again returned back to the debugger. A more meaningful debug session will involve setting up a breakpoint before executing the code.

F.3 Breakpoints

Breakpoints are those locations in the program where the user would like the program code to stop and return control to the debugger. During the execution of a program whenever a breakpoint is hit, the program stops its execution and the debugger gets into interactive mode. At this point the user can look at the variables of the program, modify their values and continue execution.

F.3.1 Specifying a breakpoint

A breakpoint is given by `break` command on the `(gdb)` prompt. The following are the possibilities for setting a breakpoint.

1. `break` *symbol*. This sets a breakpoint at a symbolic label in the program. The symbol can be name of a function or any other label defined in the program. The following is an example of setting a breakpoint right in the beginning of the program.

```
(gdb) break _start
Breakpoint 1 at 0x8048074: file hello.S, line 11.
(gdb)
```

   This shows that breakpoint is set at program address 0x8048074. This address corresponds to the symbol `_start` in the program and corresponding location in the source is line 11 in file `hello.S`.

2. `break *`*address*. This sets a breakpoint at the given address as shown in the following example. Address argument can be specified in hexadecimal number format by prefixing it with '0x', in decimal or in octal format by prefixing it with a '0'.

```
(gdb) break *0x8048079
Breakpoint 2 at 0x8048079: file hello.S, line 12.
(gdb)
```

3. `break` *line_num*. This sets a breakpoint at the specified line of the currently selected source program. In the beginning till a program is run, currently selected source is not defined and therefore this command may not work.

4. `break` *file:line*. This sets a breakpoint at the specified location of the specified source file. The following shows this method to set breakpoints in the source programs.

```
(gdb) break hello.S:13
Breakpoint 3 at 0x804807e: file hello.S, line 13.
(gdb)
```

Breakpoint command can also be abbreviated as 'b', 'br' or 'bre'.

F.3.2 Removing a breakpoint

Each breakpoint is given a unique number. Breakpoints can be removed using `delete` command as shown below.

```
(gdb) delete 1
(gdb)
```

`delete` command can also be abbreviated as `del`.

F.3.3 Listing current breakpoints

Currently selected breakpoints can be seen using `info break` command as shown below.

```
(gdb) info break
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x08048079 hello.S:12
3   breakpoint     keep y   0x0804807e hello.S:13
(gdb)
```

F.4 Program Execution Control

There are two different ways of controlling the program execution.

1. Tracing or stepping through code.
2. Running program till a breakpoint is hit or the program is executed completely.

In the first mode, the execution is carried out one line or one instruction at a time. After an instruction is executed the control is transferred to `gdb` and it enters in interactive mode. In the second mode, control is transferred to the user program. The control is transferred back to the debugger only when a breakpoint is hit during the execution of the program or when the program is completed.

F.4.1 Stepping through code

There are two different ways of stepping through the programs depending upon the behavior of handling function calls in the program. The first method is known as "step-into" code. The second mode is known

as "step-over" mode. Two modes of stepping through programs are almost identical except the case when a function call is encountered. In the "step-over" mode, the program executes the entire function call and stops at the next line in the program. In the "step-through" mode, the program steps into the function and stops at the first line of the function definition. Stepping through code may then be continued within the function.

In the `gdb`, there are two commands known as `step` and `next`. The `step` command implements "step-into" mode of debugging while `next` command implements "step-over" mode of debugging.

As the execution is carried out, `gdb` shows the next program line that will be executed when control is transferred to the user program. The following are some examples of using `step` and `next` commands of `gdb`.

```
Breakpoint 3, nextarg () at prog.S:29
29              mov     (%esp, %esi), %ecx
(gdb) step
nextarg () at prog.S:30
30              add     $4, %esi // Offset for next arg
(gdb) step
nextarg () at prog.S:31
31              cmp     $0, %ecx // Check if end of args
(gdb) step
nextarg () at prog.S:32
32              je      envp_print
(gdb) step
nextarg () at prog.S:36
36              call    str_len
(gdb) step
str_len () at prog.S:92
92              mov     %ecx, %edi
(gdb) continue
Breakpoint 3, nextarg () at prog.S:29
29              mov     (%esp, %esi), %ecx
(gdb) next
nextarg () at prog.S:30
30              add     $4, %esi // Offset for next arg
(gdb) next
nextarg () at prog.S:31
31              cmp     $0, %ecx // Check if end of args
(gdb) next
nextarg () at prog.S:32
32              je      envp_print
(gdb) next
nextarg () at prog.S:36
36              call    str_len
(gdb) next
nextarg () at prog.S:37
37              movl    $(SYS_write),%eax
```

In this example, `step` and `next` commands are used to show the stepping through code when '`call str_len`' instruction is executed. When `step` command is used, the control is broken at line 92 of the program where function `str_len` is defined. On the other hand, when `next` command is used, the control is broken at line 37 of the program right after line 36.

Usually, the lines in an Assembly language program correspond to one machine instruction. These are certain cases when it may not be true. GNU debugger provides two more commands to step through machine instructions. These commands are `stepi` and `nexti` and are used to "step-into" or "step-over" the machine instructions. The effect of this is shown in the following example where a line containing '`repne scasb`' instruction is shown. The `step` command takes beyond the execution of this line while `stepi` command breaks the control each time `repne` and `scasb` instructions are executed.

```
(gdb) step
str_len () at prog.S:96
96              mov     $5000, %ecx // Max length of string
(gdb) step
str_len () at prog.S:97
97              repne   scasb   // modifies it.
(gdb) step
str_len () at prog.S:98
98              pop     %ecx    // Restore ecx
     .
     .
     .
(gdb) stepi
str_len () at prog.S:97
97              repne   scasb   // modifies it.
(gdb) stepi
str_len () at prog.S:97
97              repne   scasb   // modifies it.
(gdb) stepi
str_len () at prog.S:97
97              repne   scasb   // modifies it.
     .
     .
     .
```

F.4.2 Continue till breakpoint

The second mode of program control is achieved when the user program is run till it hits a breakpoint or till it completes. There are two commands which are commonly used for such kind of debugging in `gdb`. The `run` command is used to run the program from the beginning. It is typically used the first time when a program is loaded. It may be used subsequently when one wants to start the debug session all over again. We have already seen an example of the `run` command of `gdb`. In general command line arguments to the program may be given after the `run` command as shown in the following example.

```
(gdb) break nextarg
Breakpoint 1 at 0x8048079: file prog.S, line 29.
(gdb) run sample arg
Starting program: /home/moona/assembly/Programs/Chap7/printargs/a.out sample arg

Breakpoint 1, nextarg () at prog.S:29
29              mov     (%esp, %esi), %ecx
(gdb)
```

When a breakpoint is hit, the `gdb` shows an identification of the breakpoint, nearest label, line number and line of the source program.

Subsequently a `continue` command may be used to resume execution of user program from the current state till it hits a breakpoint again.

F.5 Displaying data

When a program is in execution, values of the variables may be found in memory or CPU registers. `gdb` provides a variety of commands to see the values of those variables.

F.5.1 Memory contents

Contents of a memory location can be seen using `x` (stands for eXamine) command of `gdb`. This command can be used to examine memory contents in any of the several formats, independent of the data types of associated variable. Generalized syntax of the `x` command is `x/cfs addr`. Here `c` is the count of the number of memory items that should be shown, and `f` and `s` represent format and size of each memory item. All three of these are optional and have a default value when not specified. When count is not specified, it defaults to 1. The following are the possible values of `f` and `s`.

Table F.1: Format for displaying memory

Format character	Meaning
x	Hexadecimal (default)
d	Signed decimal
u	Unsigned decimal
o	Octal
t	Binary
a	Address
c	Character
f	Single precision float
s	Null terminated string
i	Diassembled machine instructions

Address of the memory location can be given as symbolic name, a hexadecimal constant or an expression involving symbolic name and constants. Following are some examples of the use of this command.

```
(gdb) x /5i nextarg
0x8048079 <nextarg>:    mov    (%esp,%esi,1),%ecx
0x804807c <nextarg+3>:  add    $0x4,%esi
0x804807f <nextarg+6>:  cmp    $0x0,%ecx
```

Table F.2: Specification of size of memory items

Format character	Meaning
b	Byte
h	Half word (16-bit)
w	long Word (32-bit) (default)
g	Giant word (64-bit)

```
0x8048082 <nextarg+9>:  je     0x80480a8 <envp_print>
0x8048084 <nextarg+11>: call   0x80480e0 <str_len>
(gdb)  x /5i nextarg+3
0x804807c <nextarg+3>:  add    $0x4,%esi
0x804807f <nextarg+6>:  cmp    $0x0,%ecx
0x8048082 <nextarg+9>:  je     0x80480a8 <envp_print>
0x8048084 <nextarg+11>: call   0x80480e0 <str_len>
0x8048089 <nextarg+16>: mov    $0x4,%eax
(gdb)  x /8xw &esp
0xbfffe190:     0x00000001      0xbffff913      0x00000000      0xbffff950
0xbfffe1a0:     0xbffff95b      0xbffff969      0xbffff983      0xbffff9fa
(gdb)  x /8c 0xbffff95b
0xbffff95b:     76 'L'  79 'O'  71 'G'  78 'N'  65 'A'  77 'M'  69 'E'  61 '='
(gdb)
```

F.5.2 Formatted display of variables

The gdb provides another command called print that displays the contents of memory locations in a formatted manner. It takes an expression and optionally a format character for the display. The expression evaluates to a memory location. Default format is as per the data type of the expression. Format characters can be as shown in table F.1 except that s and i can not be given as arguments.

The print command is normally used for printing variables of high level languages.

F.5.3 Display of register contents

Various registers of the processor can be viewed using info command. In addition they can also be viewed using print command when the registers are passed as expression. In the later case, a '$' need to be prefixed to the name of the register. For example print $eax will print the contents of register eax. When these registers are passed as argument to the x command, they provide the address of the memory location as shown in the example earlier.

Following are four relevant commands to view the contents of registers.

1. `info reg`. This command displays the contents of all general purpose registers.

2. `info float`. This command displays the contents of all x87 FPU registers.

3. `info vector`. This command displays the contents of all SIMD registers.

4. `info all`. This command displays the contents of all registers including general purpose, x87 FPU and SIMD registers.

Following is a screen shot of using commands to view registers.

```
(gdb)  p $ebx
$4 = 0
(gdb)  p $edx
$5 = 0
(gdb)  info reg
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbfffe190       0xbfffe190
ebp            0x0      0x0
    .
    .
    .
(gdb) info float
  R7: Empty   0x00000000000000000000
  R6: Empty   0x00000000000000000000
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
=>R0: Empty   0x00000000000000000000

Status Word:         0x0000
                       TOP: 0
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0xffff
Instruction Pointer: 0x00:0x00000000
Operand Pointer:     0x00:0x00000000
Opcode:              0x0000
(gdb)  print $mm1
$7 = {uint64 = 0, v2_int32 = {0, 0}, v4_int16 = {0, 0, 0, 0},
  v8_int8 = "\000\000\000\000\000\000\000\000"}
(gdb)  info vector
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0},
  v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0},
  uint128 = 0x00000000000000000000000000000000}
xmm1           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0},
  v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0},
  uint128 = 0x00000000000000000000000000000000}
    .
    .
    .
(gdb)
```

The MMX and XMM registers in IA32 processors can store values in various formats. Commands to show contents of these registers display various views. For example, MMX registers are shown as 64 bit unsigned integer (`uint64`), two packed 32-bit integers (`v2_int32`), four packed 16-bit integers (`v4_int16`) and eight packed 8-bit integers (`v8_int8`). In a similar manner XMM registers are shown as packed single precision floating point numbers, packed double precision floating point numbers, packed bytes, packed 16-bit integers, packed 32-bit integers, packed 64-bit integers and a single 128 bit integer.

F.6 Modifying data

F.6.1 Modifying memory, variables

The `print` command of `gdb` is more generic than what has been described. It can be additionally used to change the values of program variables. The `gdb` also provides another command called `set var` which is also used to set the values of variables in the programs.

For example, `print t=40` sets a value 40 in variable `t` of the program and then prints it (effectively it prints 40). If there is no need to print the value, `set var` command may be used. For example `set var x=40` sets the value of a program variable `x` to 40. In a similar manner `set var *(unsigned short *)0xbfffe190 = 50` can be used to set value of memory locations 0xbffffe190 and 0xbffffe191 to 50 and 00 respectively. In this command, constant 0xbfffe190 is first type-casted to a pointer to unsigned short data type. Later a '*' operator is used to store value 50 at that memory location. As the address is type-casted to unsigned short pointer, value 50 is treated as an unsigned short (16-bits) and stored at that memory location.

F.6.2 Modifying registers

Registers are available as variables within the `gdb` with a prefix of '$' to the register names. These can be substituted for in any expression that may take a symbolic name. Therefore the `print` and `set` commands may be used to modify the contents of any CPU register. For example, in order to change contents of register `eax`, command "`print $eax=20`" may be used. This command will set the register `eax` to 20 and print 20 on the screen.

The following are a few examples of using commands to change CPU registers.

```
(gdb)  set var $eax = 20
(gdb)  info reg
eax            0x14     20
ecx            0xbffff905       -1073743611
```

```
    .
    .
    .
(gdb)  print $xmm1
$11 = {v4_float = {0, 0, 0, 0}, v2_double = {0, 0},
  v16_int8 = '\0' <repeats 15 times>, v8_int16 = {0, 0, 0, 0, 0, 0, 0, 0},
  v4_int32 = {0, 0, 0, 0}, v2_int64 = {0, 0},
  uint128 = 0x00000000000000000000000000000000}
(gdb)  set $xmm1.v16_int82=23
(gdb)  print $xmm1
$12 = {v4_float = {2.11221641e-39, 0, 0, 0}, v2_double = {
    7.4471898181459447e-318, 0},
  v16_int8 = "\000\000\027", '\0' <repeats 12 times>, v8_int16 = {0, 23, 0, 0,
    0, 0, 0, 0}, v4_int32 = {1507328, 0, 0, 0}, v2_int64 = {1507328, 0},
  uint128 = 0x00000000000000000000000000170000}
(gdb)  set $xmm1.v4_int32={45, 90, -32, 1220}
(gdb)  print $xmm1
$13 = {v4_float = {6.30584309e-44, 1.26116862e-43, -nan(0x7fffe0),
    1.70958413e-42}, v2_double = {1.9097962120910746e-312,
    2.5909568607527874e-311},
  v16_int8 = "-\000\000\000Z\000\000\000\004\000", v8_int16 = {45, 0, 90,
    0, -32, -1, 1220, 0}, v4_int32 = {45, 90, -32, 1220}, v2_int64 = {
    386547056685, 5244155068384}, uint128 = 0x000004c4ffffffe00000005a0000002d}
(gdb)
```

F.7 Other useful commands

F.7.1 Quitting a debug session

A debug session can be terminated by executing `quit` command on `(gdb)` prompt.

F.7.2 Disassembly

The `gdb` provides a useful command called `disassem` to show part of the program in disassembled machine instruction format. It takes an address of the memory location as an argument. The output format is similar to the one shown by "`x /i`" command.

F.7.3 Listing source programs

The lines of the source program around the current line can be seen using `list` or `li` command. Subsequent use of this command shows further lines in the source program.

Appendix G

ASCII Character Set

Code		Character	Code		Character
hex	Dec		hex	Dec	
00	000	NUL \0	20	032	Space
01	001	^A	21	033	!
02	002	^B	22	034	"
03	003	^C	23	035	#
04	004	^D	24	036	$
05	005	^E	25	037	%
06	006	^F	26	038	&
07	007	Bell	27	039	'
08	008	Backspace \b	28	040	(
09	009	Tab \t	29	041	)
0A	010	Line feed \n	2A	042	*
0B	011	^K	2B	043	+
0C	012	Form feed \f	2C	044	,
0D	013	CR \r	2D	045	-
0E	014	^N	2E	046	.
0F	015	^O	2F	047	/
10	016	^P	30	048	0
11	017	^Q	31	049	1
12	018	^R	32	050	2
13	019	^S	33	051	3
14	020	^T	34	052	4
15	021	^U	35	053	5
16	022	^V	36	054	6
17	023	^W	37	055	7
18	024	^X	38	056	8
19	025	^Y	39	057	9
1A	026	^Z	3A	058	:
1B	027	Escape	3B	059	;
1C	028	^\	3C	060	<
1D	029	^]	3D	061	=
1E	030	^^	3E	062	>
1F	031	^_	3F	063	?

Code		Character	Code		Character
hex	Dec		hex	Dec	
40	064	@	60	096	`
41	065	A	61	097	a
42	066	B	62	098	b
43	067	C	63	099	c
44	068	D	64	100	d
45	069	E	65	101	e
46	070	F	66	102	f
47	071	G	67	103	g
48	072	H	68	104	h
49	073	I	69	105	i
4A	074	J	6A	106	j
4B	075	K	6B	107	k
4C	076	L	6C	108	l
4D	077	M	6D	109	m
4E	078	N	6E	110	n
4F	079	O	6F	111	o
50	080	P	70	112	p
51	081	Q	71	113	q
52	082	R	72	114	r
53	083	S	73	115	s
54	084	T	74	116	t
55	085	U	75	117	u
56	086	V	76	118	v
57	087	W	77	119	w
58	088	X	78	120	x
59	089	Y	79	121	y
5A	090	Z	7A	122	z
5B	091	[	7B	123	{
5C	092	\	7C	124	|
5D	093	]	7D	125	}
5E	094	^	7E	126	~
5F	095	_	7F	127	DEL

Appendix H

Solutions to selected exercises

EX 1.1: On executing `as hello.S` we get several errors. The comments in the program are not ignored by '`as`' and are reported as invalid syntax. When the program is run through C pre-processor, it removes these comments during the generation of `hello.s`. Constants such as `SYS_WRITE` are also expanded by the C pre-processor. They remain undefined when `hello.S` program is assembled without C pre-processor.

EX 1.2: `hello.S` and `hello.s` are different. Lines in `hello.s` are derived from `hello.S` after the C pre-processor has processed the pre-processor directives such as `#include <asm/unistd.h>` and removed various comments such as `//SYS_write = 4`. The constants are replaced by their values. For example, `$(SYS_write)` is changed to `$(4)`. File `hello.s` can be used directly to generate an object file but `hello.S` must be processed by the C pre-processor.

EX 2.1: 0x7744.

EX 2.2: If the value of register `dx` is 0xFFFF then adding 1 will make it 0x0000 without changing any of the upper 16-bits of register `edx`. However if 1 is added to the register `edx` in which the `dx` is 0xFFFF then the upper bits will also change.

EX 2.3: 0x00FF.

EX 2.4: The word at the location $a + 3$ is 0x6E5C.

EX 2.5: The long refers to 32-bit number. So the value of long stored at location $a + 1$ will be the 32-bit number stored in memory location $a + 1$ to $a + 4$ which is 0x6E5C2422.

EX 2.6:

(a) `$3` : Immediate. `(%esi)` : Register Indirect. Size is 1 byte (since 'b' suffix is used in `addb`).

(b) `eax` : Register. `(%ebx,%edi)` : Base + Index. Size is 4 bytes (since 32-bit `eax` register is used).

(c) `$1` : Immediate. `20(%eax,%ebx)` : Base + Index + Displacement. Size is 4 bytes (since '`l`' is specified in `shrl`).

(d) `(,%esi,4)` : Index $*$ Scale. `eax` : Register. Size is 4 bytes.

(e) `100` : Direct. `al` : Register. Size is 8-bits.

EX 2.7:

(a) `$0` will be stored at memory location 0x00003040. The size of the memory operand is 32-bits as 'l' is specified.

(b) 32-bit value stored at the address 0x00002000 (*i.e.* 0x00000000) will be read and copied to register `eax`. Therefore, `eax` = 0x00000000. The size of the memory operand is 32-bits.

(c) Operand at the location `esi + eax` will be incremented by one. So 0x01 will be stored at location 0x00001FFF. The size is 1 byte ('b' is used in `incb` instruction).

(d) `bx` will contain 0x000F. As `bx` is modified, `ebx` will contain 0x0000000F.

(e) `eax` remains the same. The 32-bit memory operand is stored at location 0x00003010 and its value is 0x00000000.

EX 3.1: None of the operands of the `xchg` instruction can be an immediate constant because the immediate constants can not be destination of any instruction. Since `xchg` instruction modifies both operands, it is not possible to have immediate constants as operands of this instruction.

EX 3.2: `xchg %al, %ah`

EX 3.3: Let the 32-bit number be stored at location `x`.

```
xchg    x, %eax
bswap   %eax
inc     %eax
bswap   %eax
xchg    x, %eax
```

EX 3.4:

```
mov   $0, %eax
mov   $-1, %ebx
mov   num, %ecx
cmp   lower, %ecx
cmovl %ebx, %eax
cmp   upper, %ecx
cmovg %ebx, %eax
```

EX 3.5:

```
mov   $0, %eax
mov   $-1, %ecx
inc   %ebx
      cmovo %ecx, %eax
```

EX 3.6:

(1) `pushw $0x487` produces the following memory and register values.

U	$a-3$
0x87	$a-2$
0x04	$a-1$
U	a

eax	0x00000000
esp	$a-2$

(2) `pushw $0x33` produces the following values.

U	$a-5$		
0x33	$a-4$		
0x00	$a-3$		
0x87	$a-2$	**eax**	0x00000000
0x04	$a-1$	**esp**	$a-4$
U	a		

(3) `pushw $0x57C` produces the following values.

U	$a-7$		
0x7C	$a-6$		
0x05	$a-5$		
0x33	$a-4$		
0x00	$a-3$		
0x87	$a-2$	**eax**	0x00000000
0x04	$a-1$	**esp**	$a-6$
U	a		

(4) `inc %esp` produces the following values.

U	$a-7$		
0x7C	$a-6$		
0x05	$a-5$		
0x33	$a-4$		
0x00	$a-3$		
0x87	$a-2$	**eax**	0x00000000
0x04	$a-1$	**esp**	$a-5$
U	a		

(5) `pop %eax` produces the following values.

U	$a-7$		
0x7C	$a-6$		
0x05	$a-5$		
0x33	$a-4$		
0x00	$a-3$		
0x87	$a-2$	**eax**	0x87003305
0x04	$a-1$	**esp**	$a-1$
U	a		

(6) `inc %esp` produces the following values.

U	$a-7$		
0x7C	$a-6$		
0x05	$a-5$		
0x33	$a-4$		
0x00	$a-3$		
0x87	$a-2$	**eax**	0x87003305
0x04	$a-1$	**esp**	a
U	a		

EX 3.7: Here the first two instructions push the values of registers `edx` and `ecx` in that order. Therefore, the top of the stack contains the value of `ecx` and the value of `edx` is right below that. First pop will copy top 4 bytes from stack to register `edx`, and the second pop will copy next 4 bytes to register `ecx`. Thus the value of registers are interchanged.

To implement the same effect we can use '`xchg %ecx, %edx`' instruction.

EX 3.8:

```
pusha
pop   %esi
pop   %edi
pop   %eax   ; remove esp from stack to temporary
pop   %ebp
pop   %eax
pop   %ecx
pop   %edx
pop   %ebx
```

EX 3.9: If the initial value of `esp` register is a then after the execution of `pusha` instruction it becomes $a - 8 * 4$, or $a - 32$.

`pushaw` works in the same way for 16-bit registers only. So after `pushaw` instruction the register `sp` contains $a - 8 * 2$, or $a - 16$.

EX 3.10:

`movsbl init, %eax`	or	`movsbl init, %eax`
`movsbl init, %ebx`		`mov %eax, %ebx`
`movsbl init, %ecx`		`mov %eax, %ecx`
`movsbl init, %edx`		`mov %eax, %edx`

EX 3.11: After `mov $0x2300, %esp` instruction, `esp` modifies to 0x00002300.
After `mov $0x2480, %ax` instruction, `ax` modifies to 0x2480.
After `movswl %ax, %ebx` instruction, `ebx` modifies to 0x00002480.
After `pushl $-10` instruction, `esp` modifies to 0x000022FC. The memory contents at locations 0x22FC–0x22FF become 0xFFFFFFF6.
After `popl %eax` instruction, `eax` modifies to 0xFFFFFFF6. `esp` modifies to 0x2300.
After `movsbl %al, %ecx` instruction, `ecx` modifies to 0xFFFFFFF6.
After `movsbl 0x22FD, %edx` instruction, `edx` modifies to 0xFFFFFFFF.

EX 3.12:

Instruction	Effect
`mov $-10, %al`	`al` = 0xF6 (2's complement representation of -10)
`movsbw %al, %ax`	`ax` = 0xFFF6
`movzwl %ax, %eax`	`eax` = 0x0000FFF6
`xchg %al, %ah`	`eax` = 0x0000F6FF
`bswap %eax`	`eax` = 0xFFF60000

EX 4.1: `jmp table` is an `eip` relative addressing mode whereas `jmp *table` is an absolute addressing mode. Assembler will generate the right code in a way that the effect is the same in both cases.

EX 4.2: `*$table` does not make any sense as the addressing mode. On assembling it we get the following error message.

```
Error: immediate operand illegal with absolute jump.
```

EX 4.3: `jle` instruction is used to build an extra safeguard where it is ensured that the initial value in register `ebx` is always a positive value. The loop is carried out only n times when n is positive.

EX 4.4: Program using conditional control transfer instruction.

```
movl   $0, %eax /* set eax = 0 as a default value */
cmpl   $0, %ebx /* if y <= 0, jump out */
```

```
        jle    L0
        movl   %ebx, %eax /* Set eax = y */
L0:
  ...
```

Program using conditional move instruction.

```
        movl   $0, %eax
        cmpl   $0, %ebx
        cmovg  %ebx, %eax /* if y>0, eax = y */
        ...
```

EX 4.5:

```
        movl   $0, %eax  /* y = 0 */
loopstart:
        cmpl   $0, %ebx
        jl     exitloop  /* jump out if x < 0 */
        sub    %eax, %ebx /* x = x - y */
        incl   %eax       /* y = y + 1 */
        jmp    loopstart
exitloop:
  ...
```

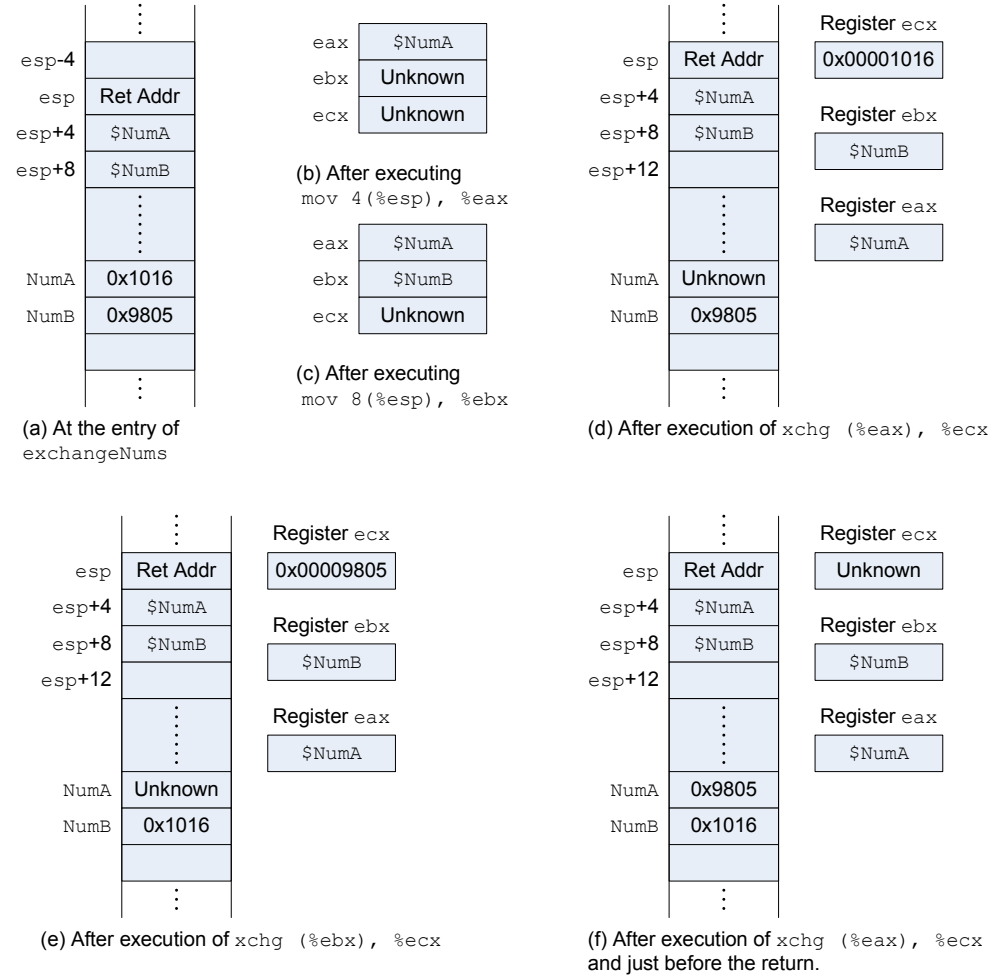
EX 4.6:

```
     movl   $n, $ecx
loopstart:
     movl   (A-4)(, %ecx, 4), %eax
     cmpl   %eax, (B-4)(, %ecx, 4)
     loope  loopstart
     /* At this point,
        ZF = 1 if no mismatch
           = 0 if mismatch.
     ecx is the index in array in case
     of mismatch */
```

EX 4.7:

```
     movl   $n, %ecx
loopstart:
     cmpl   %eax, A-4(, %ecx, 4)
     loope  loopstart
```

EX 4.8:



EX 4.9: This function only exchanges the addresses of the parameters on the stack but the parameters remain the same.

EX 4.10:

```
.globl reverseInts
reverseInts:
     sub   $4, %esp /* Space for local variable */
     /* At this point the stack layout is
      * (%esp):   Local variable
      * 4(%esp):  Return address
      * 8(%esp):  Address of a
      * 12(%esp): Address of b
      * 16(%esp): Address of c
      * 20(%esp): Address of d */
     movl 8(%esp), %edx  /* Addr of a */
     movl 20(%esp), %ecx /* Addr of d */
```

```
     xchg (%edx), %eax
     xchg (%ecx), %eax
     xchg (%edx), %eax    /* a and d exchanged */
     movl 12(%esp), %edx  /* Addr of b */
     movl 16(%esp), %ecx  /* Addr of c */
     xchg (%edx), %eax
     xchg (%ecx), %eax
     xchg (%edx), %eax    /* b and c exchanged */
     add $4, %esp /* Recover space for local var */
     ret
```

In this program, the local variable space is just created but the local variable itself is not used. The program modifies registers `edx` and `ecx` which could be saved in the local variables if needed.

EX 4.11:

```
.globl reverseInts
reverseInts:
     push  %ebp
     movl  %esp, %ebp
     sub   $4, %esp
     /* At this point the frame layout is
      * -4(%ebp): Local variable
      * (%ebp):   Old frame pointer
      * 4(%ebp):  Return address
      * 8(%ebp):  Address of a
      * 12(%ebp): Address of b
      * 16(%ebp): Address of c
      * 20(%ebp): Address of d */
     push %edx  /* Save registers that are */
     push %ecx  /* modified in the function */
     movl 8(%ebp), %edx  /* Addr of a */
     movl 20(%ebp), %ecx /* Addr of d */
     xchg (%edx), %eax
     xchg (%ecx), %eax
     xchg (%edx), %eax    /* a and d exchanged */
     movl 12(%ebp), %edx  /* Addr of b */
     movl 16(%ebp), %ecx  /* Addr of c */
     xchg (%edx), %eax
     xchg (%ecx), %eax
     xchg (%edx), %eax    /* b and c exchanged */
     pop  %ecx            /* Restore registers */
     pop  %edx
     mov  %ebp, %esp /* Recover space for local var */
     pop  %ebp       /* Restore old frame pointer */
     ret
```

EX 4.12: In general a function can always use `esp` based addressing to access its local variables and parameters. It therefore need not use `ebp` at all. In

particular if a function does not use any local variables in memory and does not take any arguments it will never need to access the frame and hence `ebp` need not be used. In this case, it need not save and restore register `ebp`.

EX 4.13: No the use of the frame pointer is not necessary. The stack pointer can be used to access the local variables and parameters.

EX 4.14:

Assembly program is the following.

```
.globl greater
greater:
     movl  4(%esp), %eax
     cmpl  8(%esp), %eax
     jg    g_exit
     movl  8(%esp), %eax
g_exit:
     ret
```

C code to use this assembly program is the following.

```
int greater(int, int);
int main(void)
   int a, b, c, g;
   printf("Enter three numbers: ");
   scanf("%d %d %d", &a, &b, &c);
   g = greater(a, b);
   if (b == g)
      b = a;
      a = g;

   g = greater(a, c);
   if (c == g)
      c = a;
      a = g;

   g = greater(b, c);
   if (c == g)
      c = b;
      b = g;

   printf("Decreasing order: %d %d %d\n", a, b, c);
   return 0;
```

EX 4.15: The Assembly language function is the following.

```
.globl isoctal
isoctal:
   mov 4(%esp), %ecx /* Start address of string */
   mov $1, %eax  /* Default return value: Success */
```

```
loopst:
   movb (%ecx), %bl
   cmp $0, %bl   /* End of string? */
   je  commonret
   inc %ecx      /* Ready for the next character */
   cmp $'0', %bl /* Check against ASCII 0 */
   jb  error     /* Not Octal */
   cmp $'7', %bl
   ja  error
   jmp loopst
error:
   mov $0, %eax
commonret:
   ret
```

The C program that uses this function is the following.

```
int isoctal(char *s);
int main(void)
   int i;
   char p[100];
   do
      printf("Enter string: ");
      scanf(" %100s", p);
    while (isoctal(p) == 0);
   printf("Got a valid octal string\n");
   return 0;
```

EX 5.1: The `inc` does not modify the flags whereas the `add` instruction does. Replacing `inc` by `add` will result in an incorrect execution as the carry flag of `add` instruction will interfere in the operation due to `adc`.

EX 5.2: The correct program is the following.

```
.globl addprecision
addprecision:
   mov   4(%esp), %ecx
   mov   8(%esp), %ex
   mov   12(%esp), %edx
   clc
.L0:
   mov   (%ebx), %al
   adc   %al, (%edx)
   pushf
   add   $1, %ebx
   add   $1, %edx
   popf
   loop  .L0
   ret
```

EX 5.3: The program without the `loop` instruction is given below. The `jecxz` instruction in this program does not modify flags. Unlike other two programs, this program will work even if the size of the numbers is given as 0.

```
.globl addprecision
addprecision:
   mov   4(%esp), %ecx
   mov   8(%esp), %ex
   mov   12(%esp), %edx
   clc
.L0:
   jecxz  exitloop
   adc    %al, (%edx)
   inc    %ebx
   inc    %edx
   decl   %ecx
   jmp    .L0
exitloop:
   ret
```

EX 5.4: The Assembly language function `sum` is the following.

```
.globl sum
sum:
   movl   4(%esp), %ecx /* Size of the array */
   movl   8(%esp), %ebx /* Address of the array */
   movl   $0, %eax      /* Initial sum is 0 */
Loopstart:
   addl   -4(%ebx,%ecx,4), %eax
   loop   Loopstart
   ret
```

EX 5.5: The Assembly language function `increment` is the following.

```
.globl increment
increment:
     movl   4(%esp), %ecx /* Size of the array */
     movl   8(%esp), %ebx /* Address of the array */
Loopstart:
     mov    -4(%ebx, %ecx, 4), %eax
     incl   (%eax)
     loop   Loopstart
     ret
```

EX 5.6: The Assembly language function `fold` is the following.

```
.globl fold
fold:
     movl  4(%esp), %ecx /* Size of the array */
     movl  8(%esp), %ebx /* Address of the array */
     movl  $0, %eax      /* Initial value of sum */
```

```
Loopstart:
     add   -4(%ebx,%ecx,4), %eax
     neg   %eax
     loop  Loopstart
     neg   %eax          /* Needed as n is even */
     ret
```

EX 5.7:

```
.globl Division
Division:
     movl  4(%esp), %eax /* Dividend */
     mov   $0, %edx /* edx must be sign extension */
     cmp   $0, %eax /* Is eax negative? */
     jge   .L0      /* If positive */
     dec   %edx
.L0: idivl 8(%esp)  /* Divide edx:eax by divisor */
     ret
```

EX 5.8:

```
/* 64-bit multiplication
 * Inputs: Address of first number at 4(%esp)
 * Address of the second number at 8(%esp)
 * Address of the third number (result) at 12(%esp)
 * Algorithm: Treat first number as (a b).
 * Second as (c d), (a, b, c, d: all 32 bits).
 * The result can be written down as
 * (ResultHI ResultLO) (ResultHI and ResultLO are
 * 32 bits each). ResultLO = lower 32 bits of (b*d)
 * ResultHi = Sum of lower 32 bits of (a*c),
 * lower 32 bits of (c * b) and upper 32 bits
 * of (b*d).
 */
.globl mult64
mult64:
   mov  4(%esp), %esi // Addr of First num
   mov  8(%esp), %edi // Addr of Second num
   mov  12(%esp), %ecx // Addr of Third num
   mov  (%esi), %ebx  // Lower 32bit of first num
   mov  (%edi), %eax  // Lower 32bit of sec num
   mul  %ebx  // Compute b*d.
   mov  %eax, (%ecx)  //ResultLo
   mov  %edx, 4(%ecx) //ResultHi=upper32bit of b.d
   mov  4(%edi), %eax // Upper 32bit of sec num
   mul  %ebx  // Compute c*b.
   add  %eax, 4(%ecx) //add lo32bit of c.b
   mov  4(%esi), %eax //Upper32 of first num
   mov  (%edi), %ebx  //Lower32 of second num
```

```
mul  %ebx  // Compute a*d
add  %eax, 4(%ecx) //add lo32bit of a*d
ret
```

EX 5.9: (i) 0x74 or 0111 0100. (ii) 0x93 or 1001 0011. (iii) 0x88 or 1000 1000.
EX 5.10: 0x2C is 44 in decimal. The BCD representation is 0100 0100, or 0x44.
EX 5.11: 0x4D and 0xE2 are invalid BCD number.
EX 5.12:

```
.globl add_BCD8
add_BCD8:
     push  %ebp
 mov   %esp, %ebp
     sub   $4, %esp  /* Space for temp variable */
 // First parameter at 8(%ebp)
 // Second parameter at 12(%ebp)
 // Temp variable at -4(%ebp)
     mov   $0, %esi
     clc
     pushf           /* Needed to balance stack */
Loopstart:
     movb  8(%ebp, %esi), %al /* Two digits of number1 */
     popf
     adc   12(%ebp, %esi), %al /* Add 2 digits of num2 */
     daa             /* Convert it to BCD */
     pushf
     movb  %al, -4(%ebp, %esi) /* Save in temp location */
     inc   %esi      /* Prepare for next set of 2 digits */
     cmp   $4, %esi
     jnz   Loopstart
     popf            /* Balance stack */
     mov   -4(%ebp), %eax /* Return value */
     leave
     ret
```

EX 5.14: Example of four digit code is given below.

```
/* Assume src1 is in bx
 * src2 is in cx
 * Result src1-src2 is put in dx */
mov %cl, %al
sub %bl, %al
das
mov %al, %dl
mov %ch, %al
sub %bh, %al
das
mov %al, %dh
...
```

EX 5.15:

```
.global AsciiAdd
AsciiAdd:
     movl  $5, %ecx     /* Number of digits */
     movl  4(%esp), %esi /* Addresses of first */
     movl  8(%esp), %edi /* and second string */
     movb  -1(%esi,%ecx), %al /* Digit of first str */
 and  $0x0F, %al
L0:  movb  -1(%edi,%ecx), %bl /* Second string digit */
     movb  -2(%esi,%ecx), %ah
     and  $0x0F, %ah
     and  $0x0F, %bl
     add  %bl, %al
     aaa
     or  $0x30, %al
     movb  %al, -1(%esi,%ecx)
     movb  %ah, %al
     loop  L0
     ret
```

EX 5.16: Assembly language function `countBits`:

```
.globl countBits
countBits:
     movl  4(%esp), %ebx
     movl  $32, %ecx
     movl  $0, %eax
 L0: ror  $1, %ebx
     jnc  L1
     inc  %eax
 L1: loop L0
     ret
```

C test program:

```
#include <stdio.h>
int countBits(int);
int main()
   int b;
   printf("Enter a number: ");
   scanf("%d", &b);
   printf("No of 1s in %d = %d\n", b, countBits(b));
   return 0;
```

EX 5.17: The C test program will be same as the one given in solution for exercise 5.16. Assembly code of the modified function is given below.

```
.globl countBits
countBits:
     movl  4(%esp), %ebx  /* x */
     xor   %eax, %eax /* Set eax = 0 */
L0:  cmp   $0, %ebx
     jz    L1
     mov   %ebx, %ecx /* Compute x-1 */
     dec   %ecx
     and   %ecx, %ebx /* x = x & (x -1) */
     inc   %eax       /* Number of bits set to 1 */
     jmp   L0
L1:  ret
```

EX 5.18:

```
.globl Power
Power:
     /* 4(%esp) is x. 8(%esp) is y */
     /* Function returns 0xFFFFFFFF in overflow cases */
     mov $1, %eax /* Initial value of power */
     mov 4(%esp), %ebx /* PowerX */
     xor %ecx, %ecx
     mov 8(%esp), %cl /* ecx = Yt */
 jecxz done  /* if ecx was 0 to start with */
L1:
     /* Test if least significant bit of ecx is 1 */
     mov $1, %edx
     and %ecx, %edx
     jz  L2
     /* LSB was 1 and hence power = power * PowerX */
     mul %ebx /* edx:eax = ebx * eax */
     cmp $0, %edx
     jne overflow
L2:
     shr $1, %ecx /* Shift Yt by 1 location */
     jecxz done
     /* PowerX = PowerX * PowerX */
     xchg %ebx, %eax
     mul %eax /* edx:eax = sqr(PowerX) */
     cmp $0, %edx
     jne overflow
     xchg %ebx, %eax /* ebx = New PowerX */
 jmp L1
overflow:
     mov $0xFFFFFFFF, %eax
done:
     ret
```

EX 6.1: Assembly language function is the following.

```
.globl crypt_test
crypt_test:
     movl  12(%esp), %ecx /* Length */
     movl  4(%esp), %esi /* Addr of string 1 */
     movl  8(%esp), %edi /* Addr of string 2 */
     movl  $1, %eax /* Success value as default */
     cld
Loopstart:
     cmpsb /* Compare byte by byte */
     je  L1 /* Test fails if two bytes are same */
     loop  Loopstart
     ret
L1:  movl  $0, %eax
     ret
```

C Program to call Assembly language function is the following.

```
#include <stdio.h>
#include <string.h>
int crypt_test(char *, char *, int);
int main()
   int i;
   char a[11],b[11];
   printf("Type in the first string (10 characters): ");
   scanf(" %10s", a);
   printf("Type in the second string (10 characters): ");
   scanf(" %10s", b);
   i=crypt_test(a, b, strlen(a));
   if(i==1)
      printf("Crypt Quality Test passed\n");
   else
      printf("Crypt Quality Test failed\n");
   return 0;
```

EX 6.2: The Assembly language function is the following.

```
      /* C Callable function. The C prototype is
       * copy (src, dest, size).
       *   src: Address of the source string
       *   dest: Address of the destination string
       *   size: Size in bytes */
.globl copy
copy:
     movl  4(%esp), %esi
     movl  8(%esp), %edi
     movl 12(%esp), %ecx
     /* Check for the overlapping strings */
     /* Is edx < src + size? */
```

```
     movl  %esi, %edx
     add  %ecx, %edx
     cmp  %edx, %edi
     jl   less
     cld
     jmp copy_str
less: /* Do a reverse copy. esi and edi should be
       * the addresses of the last element */
     add  %ecx, %esi
     add  %ecx, %edi
     dec  %esi
     dec  %edi
     std
copy_str:
     movsb
     loop copy_str
     ret
```

EX 6.3: Codes for `index` and `rindex` are the following.

```
.globl index
index:
     movl  4(%esp), %edi
     movl  8(%esp), %ecx
     movb  12(%esp), %al
     cld
L1:
     scasb
     loopne L1
     jne   L2
     movl  %edi, %eax
     dec   %eax
     ret
L2:
     mov   $0, %eax
     ret
.globl rindex
rindex:
     movl  4(%esp), %edi
     movl  8(%esp), %ecx
     movb  12(%esp), %al
     add   %ecx, %edi
     dec   %edi /* Address of the last element */
     std
L3:
     scasb
     loopne  L3
     jne   L4
     movl  %edi, %eax
     inc   %eax /* compensate decrmnt by scasb */
```

```
     ret
L4:
     mov   $0, %eax
     ret
```

EX 6.4: Assembly code is the following.

```
.globl count
count:
     movl  4(%esp), %edi
     movl  8(%esp), %ecx
     movb 12(%esp), %al
     movl  $0, %edx
     cld
 L1: scasb
     jnz  L2
     inc  %edx /* edx is the count */
 L2: loop L1
     movl  %edx, %eax
     ret
```

EX 6.5: The `memset` is the normal program. More efficient program with the use of `stosl`, `stosw` and `stosb` instructions named as `memset_fast`.

```
.globl memset
memset:
     movl  4(%esp), %edi
     movb  8(%esp), %al
     movl  12(%esp), %ecx
     cld
     push  %edi
     rep   stosb
     pop   %eax /* Address of string in eax */
     ret
.globl memset_fast
memset_fast:
     movl  4(%esp), %edi
     movb  8(%esp), %al
     movl  12(%esp), %ecx
     jecxz done
 /* Copy the byte in al to ah and upper half of eax */
 movb  %al, %ah
 push  %ax  /* Push ax twice and pop into eax */
 push  %ax
 pop   %eax
     cld
     test  $1, %edi /* Check if address is an odd address */
     jz    L0       /* If even address */
     stosb
```

```
     decl  %ecx /* Decrement count */
L0:  /* Check if address is a multiple of 4 or not */
     test $2, %edi
     jz  L1
     stosw
     subl $2, %ecx /* Decrement count by 2 */
L1:
     movl %ecx, 12(%esp) /* Save the count */
     /* edi is always a multiple of 4 */
     shr  $2, %ecx     /* Count in units of long */
     rep  stosl
     /* Copy the remaining bytes (up to 3) */
     mov  12(%esp), %ecx
     and  $3, %ecx /* Remaining bytes */
     cmp  $2, %ecx
     jl   L2
     stosw
     subl $2, %ecx
L2:
     jecxz done
     stosb
done:
     movl  4(%esp), %eax
     ret
```

EX 6.6:

```
.globl swab
swab:
     movl  4(%esp), %esi
     movl  8(%esp), %edi
     movl  12(%esp), %ecx
     cld
     shr   $1, %ecx
     cmp   $0, %ecx  /* If n was 0 or 1, skip */
     jl    L2        /* the loop */
L1:
     lodsw
     xchg  %al, %ah
     stosw
     loop  L1
L2:
     mov   12(%esp), %ecx /* Check if n was odd */
     and   $1, %ecx
     jz    L3             /* Done. n is even */
     lodsb
     stosb
L3:  ret
```

EX 6.8:

```
.globl Highlight
Highlight:
    /* C Callable function. The C prototype is
     * void Highlight(char *s, unsigned int len);
     */
    movl  4(%esp), %ebx /* Address */
    movl  8(%esp), %ecx /* Count */
    shl   $3, %ecx      /* Find number of bits */
dec   %ecx
L0: btc   %ecx, (%ebx)
    sub   $2, %ecx      /* ecx decremented twice. */
    jns   L0            /* if ecx is positive. */
    ret
```

EX 6.9:

```
.globl LongestSequence0
LongestSequence0:
    /* Not a C-callable function as the argument is
     * passed through the register eax. Return
     * value of the function is given in eax */
    movl  $-1, %edx /* bit index of prev bit set to 1 */
    movl  $0, %ebx  /* Length of sequence */
L1: bsf   %eax, %ecx /* Index of LSB that is 1 */
    jnz   L2         /* If there isn't any 1 */
mov   $32, %ecx  /* Index=32 to take care of last */
jmp   L3         /* sequence to 0. */
L2:
    btr   %ecx, %eax /* Set that bit to 0 */
L3:
    push  %ecx       /* Save current index */
    sub   $1, %ecx   /* Length of seq = current index -
                        previous index - 1 */
    sub   %edx, %ecx /* ecx = length of sequence */
    pop   %edx       /* Make current ind as prev ind */
js    done       /* -ve after last seq is done */
    cmp   %ebx, %ecx
    cmova %ecx, %ebx /* If ecx > ebx, ebx = ecx */
    jmp   L1
done:
    movl  %ebx, %eax
    ret
```

EX 6.10:

```
.globl IsLongSequence1
IsLongSequence1:
    /* C Callable function with the prototype as
     * int IsLongSequence1(unsigned int); */
```

```
    movl  4(%esp), %ebx
L0:
    xor   %eax, %eax /* Set eax = 0. => No seq found */
    bsf   %ebx, %ecx /* Index of LSB that is 1 */
    jz    exit       /* If no 1 is found */
    movl  %ecx, %edx /* ecx is i (i < 32) */
    movl  $1, %eax   /* compute 2**i */
    shl   %cl, %eax  /* i is guaranteed < 32 */
    add   %eax, %ebx /* Add and find index of 1 again */
    bsf   %ebx, %ecx
    jnz    L1
    /* if No 1 found after add. This scenarios is same
       as 1 found at location 32 */
    mov   $32, %ecx
    jmp   L2
L1:
    /* Clear bit at ecx location */
    btr   %ecx, %ebx
L2: /* Compute length of sequence */
    sub   %edx, %ecx
    cmp   $4, %ecx
    jge   done
    jmp   L0
done:
    movl  $1, %eax
exit:
    ret
```

EX 6.11:

```
.data
fmtStr:
.string "The position of bit set to 1 is = %d\n"
.text
.globl scanbits
scanbits:
     /* The algorithm is the following.
      * For i=31 down to 0
      *    Set eax = 2**i.
      *    test this mask with the argument to find
      *      if ith bit is set in the argument.
      *    If argument has the bit set. Print */
     pushl %ebp
     movl  %esp, %ebp
     movl  8(%ebp), %ebx /* ebx is the argument */
     movl  $32, %ecx
L1:
     mov   $0, %eax   /* eax = 2**i */
     dec   %ecx
     bts   %ecx, %eax
```

```
     /* test eax with ebx */
     test  %eax, %ebx
     jz    no_print
     push  %ecx
     pushl $fmtStr
     call  printf
     pop   %ecx
     pop   %ecx  /* Restore ecx */
no_print:
     jecxz done
     jmp   L1
done:
     leave
     ret
```

EX 7.1:

```
#include <syscall.h>
#define STDOUT 1
     /* This function is not C callable. In addition, it
      * does not use any C library function */
.lcomm buffer, 10 /* Buffer is an array of 10 bytes */
.globl PrintInt
PrintInt:
     /* Number is in eax. Max 10 digits number *
      * Algorithm is the following.
      * Keep dividing eax by 10 till it becomes 0.
      * at each step, the remainder (between 0 to 9) is
      * converted to an ASCII character and put in buffer
      * backwards.
      * When number becomes zero, print the buffer.
      *
      * In this program, a _start is also added to make it
      * runnable. The program must be compiled after
      * preprocessing it with cpp */
      mov   $(buffer+9), %edi
L1:
      xor   %edx, %edx /* Set %edx = 0 */
      mov   $10, %ecx
      div   %ecx /* quotient in eax. Remainder in %edx */
      add   $0x30, %edx
      mov   %dl, (%edi)
      dec   %edi
      cmp   $0, %eax
      jz    print
      jmp   L1
print:
      /* edi points to one location below the print string */
      inc   %edi
      /* Length of buffer to print: (buffer+10) -edi */
```

```
      mov   $(buffer+10), %edx
      sub   %edi, %edx
      mov   $SYS_write, %eax
      mov   $STDOUT, %ebx
      mov   %edi, %ecx
      int   $0x80
      ret
.data
NUM:  .int 233
NEWLINE: .string "\n"
.text
.globl _start
_start:
      mov   NUM, %eax
      call  PrintInt
  /* Print newline */
  mov   $SYS_write, %eax
  mov   $STDOUT, %ebx
  mov   $NEWLINE, %ecx
  mov   $1, %edx
  int   $0x80
      /* Exit */
      mov   $SYS_exit, %eax
      mov   $0, %ebx
      int   $0x80
```

EX 7.2:

```
#include <syscall.h>
#define O_RDONLY 0
#define STDOUT 1
#define STDERR 2
.data
NoArg:
      .ascii "No arguments are given.\n"
endNoArg:
File:
      .ascii "File "
endFile:
Readable:
      .ascii " is readable.\n"
endReadable:
NonReadable:
      .ascii " is not readable.\n"
endNonReadable:
.text
str_len:
      /* Given address in ecx, it returns the length of
       * null terminated string in edx. The function is
       * not declared a .globl as it is used locally only */
```

```
      mov   $0, %edx
L1:
      cmpb  $0, (%ecx, %edx)
      jz    done
      inc   %edx
      jmp   L1
done: ret

.globl _start
_start:
      /* At this point the stack layout is
       * (%esp): argc
       * 4(%esp): Address of the program name string
       * 8(%esp): command line argument
       * 12(%esp): 0
       * ... */
      mov   8(%esp), %ebx
      cmp   $0, %ebx /* Check if argument is given */
      jnz   check_readable
      mov   $SYS_write, %eax
      mov   $STDERR, %ebx
      mov   $NoArg, %ecx
      mov   $(endNoArg - NoArg), %edx
      int   $0x80
      jmp   exit
check_readable:
      /* First print "file xyz" */
      mov   $SYS_write, %eax
      mov   $STDOUT, %ebx
      mov   $File, %ecx
      mov   $(endFile - File), %edx
      int   $0x80
      mov   $SYS_write, %eax
      mov   $STDOUT, %ebx
      mov   8(%esp), %ecx
      call  str_len
      int   $0x80
      /* Now open file in RD mode */
      mov   $SYS_open, %eax
      mov   8(%esp), %ebx
      mov   $(O_RDONLY), %ecx
      int   $0x80
      /* Check the return value */
      mov   $Readable, %ecx
      mov   $(endReadable-Readable), %edx
      cmp   $0, %eax
      jnl   print
      mov   $NonReadable, %ecx
      mov   $(endNonReadable-NonReadable), %edx
print:
```

```
      mov   $STDOUT, %ebx
      mov   $SYS_write, %eax
      int   $0x80
exit:
      mov   $SYS_exit, %eax
      mov   $0, %ebx
      int   $0x80
```

EX 9.1: (a) $1.1010\overline{1100}$ (b) 101.00001 (c) $11.000110111011\ldots$ (d) $101.111\overline{0011}$
EX 9.2: (a) 13.40625 (b) 5.8203125 (c) $6.93548387096\ldots$
EX 9.3: (a) $\approx 9.4506 \times 10^{-4}$ (b) 0.00078125
EX 9.4:

(a) $\mathcal{F}_{12,6}(12.6)$ = 0011 0010 0110. $\mathcal{F}_{12,6}(11.3)$ = 0010 1101 0011. Integer summation yields 0101 1111 1001. $\mathcal{F}_{12,6}(23.9)$ is also the same. Hence no truncation error.

(b) $\mathcal{F}_{12,6}(-9.8)$ = 1101 1000 1101. $\mathcal{F}_{12,6}(13.2)$ = 0011 0100 1100. Subtraction using integer arithmetic yields 1010 0100 0001. $\mathcal{F}_{12,6}(-23)$ = 1010 0100 0000. Hence there is a truncation error of 0.015625.

(c) $\mathcal{F}_{12,6}(7.2)$ = 0001 1100 1100. $\mathcal{F}_{12,6}(4.1)$ = 0001 0000 0110. Multiplication using integer arithmetic followed by a right shift of 6 bits yields 0111 0101 1011. $\mathcal{F}_{12,6}(29.52)$ = 0111 0110 0001. Hence there is a truncation error of 0.09375.

(d) $\mathcal{F}_{12,6}(15.9)$ = 0011 1111 1001. $\mathcal{F}_{12,6}(3.0)$ = 0000 1100 0000. A left shift of dividend (by 6 bits) and then integer division by the divisor yields 0001 0101 0011. $\mathcal{F}_{12,6}(5.3)$ = 0001 0101 0011. Hence there is no truncation error.

EX 9.5:

```
.globl fixAdd
fixAdd:
   add   %ebx, %eax
   ret

.globl fixSub
fixSub:
   sub   %ebx, %eax
   ret

.globl fixMul
fixMul:
   push  %edx
   imul  %ebx  /* Multiple eax by ebx */
   /* Result is in edx:eax */
   shrd  $16, %edx, %eax /* Shift edx:eax by 16bits */
   pop   %edx
   ret

.globl fixDiv
```

```
fixDiv:
   push  %edx
   cltd        /* Convert number in eax to 64-bit */
   shld  $16, %eax, %edx /* shift 16bits of eax into edx */
   shl   $16, %eax /* Shift eax by 16 bits */
   /* edx:eax is original number in eax shifted by 16 bits */
   idiv  %ebx  /* divide by the seccond number */
   pop   %edx
   ret
```

EX 9.6:

```
.extern fixAdd
.extern fixSub
.extern fixMul
.extern fixDiv
.globl CfixAdd
CfixAdd:
   mov   4(%esp), %eax
   mov   8(%esp), %ebx
   jmp   fixAdd /* Return of fixAdd will return to C */

.globl CfixSub
CfixSub:
   mov   4(%esp), %eax
   mov   8(%esp), %ebx
   jmp   fixSub

.globl CfixMul
CfixMul:
   mov   4(%esp), %eax
   mov   8(%esp), %ebx
   jmp   fixMul

.globl CfixDiv
CfixDiv:
   mov   4(%esp), %eax
   mov   8(%esp), %ebx
   jmp   fixDiv
```

Following C program can be used to test these routines.

```
extern int CfixAdd(int, int);
extern int CfixSub(int, int);
extern int CfixMul(int, int);
extern int CfixDiv(int, int);
#define TwoPwr16  65536.0
int main(void)
   float f1, f2;
```

```
int result;
printf("Test of Fixed Point Functions\n");
printf("Enter two real numbers :");
scanf("%f %f", &f1, &f2);
result = CfixAdd((int)(f1*TwoPwr16), (int)(f2*TwoPwr16));
printf("Addition: %f\n", (double)result/TwoPwr16);
result = CfixSub((int)(f1*TwoPwr16), (int)(f2*TwoPwr16));
printf("Subtraction: %f\n", (double)result/TwoPwr16);
result = CfixMul((int)(f1*TwoPwr16), (int)(f2*TwoPwr16));
printf("Multiplication: %f\n", (double)result/TwoPwr16);
result = CfixDiv((int)(f1*TwoPwr16), (int)(f2*TwoPwr16));
printf("Division: %f\n", (double)result/TwoPwr16);
return 0;
```

EX 9.7: (i) 0x42FB3800 (ii) 0xC7AA1A80 (iii) 0x3FB4FDF4
EX 9.8: (i) 2.359375 (ii) 130.00 (iii) -15.00
EX 9.9: (i) 0x4002E00000000000 (ii) 0x4060400000000000
(iii) 0xC02E000000000000
EX 9.10:

(a) Can be represented as normalized number.
(b) Can be represented as denormalized number.
(c) Can be represented as denormalized number.
(d) Can be represented as normalized number.
(e) Can be represented as denormalized number.
(f) Can be represented as denormalized number.

EX 9.11:

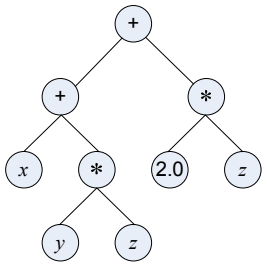
(a) Normalized number: -24.5
(b) Normalized number: 96.0
(c) Denormalized Number: -0.75×2^{-126}
(d) Special Number: $-\infty$
(e) Special Number: -0.0
(g) Special Number: NaN

EX 9.12:

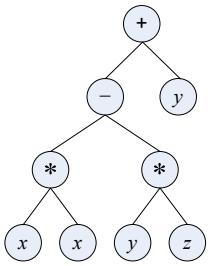
```
/* These functions are not C Callable. The functions
 * are otherwise equivalent to a C library functions
 * by the same name. If the arguments are on the
 * register stack, fmod, fmodf and fmodl functions
 * are all the same. Register arguments have data
 * only in the double extended-precision format */
.globl fmod
fmod:
.globl fmodf
fmodf:
.globl fmodl
fmodl:
    /* To compute the remainder of a/b where a is in
     * st(0) and b is in st(1). Abs(Remainder) is
     * between 0.0 and Abs(b). Sign of remainder
     * same as that of a. */
    fprem       // Partial remainder
    fstsw %ax   // Save FPU flags in register ax
```

```
    sahf        // C2 flag: Whether reduction complete?
    jp    fmodl // Execute again if reduction incomplete
    fstp %st(1) // Adjust the stack to remove one item.
    ret         // While retaining the stack top value.
.globl drem
drem:
    /* To compute the BSD4.3 remainder of a/b. a is in
     * st(0) and b is in st(1). Remainder is always
     * between +b/2 and -b/2 */
    fprem1      // Partial remainder
    fstsw %ax   // Take FPU flags to eflags
    sahf        // Reduction flag is in PF
    jp    drem  // Execute again if reduction incomplete
    fstp %st(1) // Adjust the stack while keeping
    ret         // result on stack top.
```

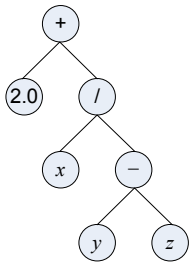
EX 9.13: The following are the expression trees for the expressions.



(i) Expression: $x + y.z + 2z$
Postfix form: $x\ y\ z\ *\ +\ 2\ z\ *\ +$



(ii) Expression: $x^2 - y.z + y$
Postfix form: $x\ x\ *\ y\ z\ *\ -\ y\ +$



(iii) Expression: $2 + (x\ /\ (y - z))$
Postfix form: $2\ x\ y\ z\ -\ /\ +$

Codes for the three expression evaluations are the following.

```
/* Exercise (i). Postfix: x y z * + 2 z * + */
.data
    Two: .float 2.0
.text
.globl eval1
eval1:
    fninit    // Initialize the coprocessor
    flds   x  // Load x on register stack
    flds   y
    flds   z
    fmulp     // Multiply and pop
    faddp     // Add and pop. Stack=(x+y.z)
    flds   Two
    flds   z
    fmulp     // Multiply and pop. Stack = (2z, x+y.z)
    faddp     // Result on stack top
    ret
/* Exercise (ii). Postfix: x x * y z * - y + */
```

```
.globl eval2
eval2:
    fninit
    flds   x
    fmul   %st(0) // Multiple x by x. Result on stack top.
    flds   y
    flds   z
    fmulp         // Stack = (y.z, x^2)
    fsubrp        // Stack = (x^2 - y.z)
    flds   y
    faddp         // Stack = (x^2 - y.z + y)
    ret
/* Exercise (iii). Postfix: 2 x y z - / + */
.globl eval3
eval3:
    fninit
    flds   Two
    flds   x
    flds   y
    flds   z
    fsubrp        // Stack = (y-z, x, 2.0)
    fdivrp        // Stack = (x/(y-z), 2.0)
    faddp
    ret
```

EX 9.14: The larger root is $\frac{-b+\sqrt{b^2-4ac}}{2a}$. Code for evaluating this root is the following.

```
// C Callable function. Prototype is
// float root1(float a, float b, float c);
.globl root1
root1:
    fninit           // Reset the x87 FPU
    // a is in 4(%esp). b in 8(%esp) and c in 12(%esp).
    flds    8(%esp)
    fmul    %st, %st // Compute b^2
    pushl   $4       // Load 4 on FPU register stack
    fildl   (%esp)
    addl    $4, %esp
    flds    4(%esp)  // Load a
    flds    12(%esp) // and c
    fmulp
    fmulp            // 4.a.c
    fsubrp           // Stack = b^2 - 4.a.c
    fsqrt
    flds    8(%esp)  // Load b
    fldz             // Load 0.0
    fsubp            // Stack = (-b, b^2 - 4.a.c.)
    faddp
```

```
    pushl   $2       // Load 2 on FPU stack
    fildl   (%esp)
    addl    $4, %esp
    flds    4(%esp)  // Load a
    fmulp            // Stack = (2a, -b+sqrt(b^2-4ac))
    fdivrp           // Result in st(0)
    ret
```

EX 9.15:

```
/* C Callable function with prototype
 * float TenPowerY(float y); */
.globl TenPowerY
TenPowerY:
    // Load y on stack
    flds    4(%esp)
    // Load 10
pushl   $10
fildl   (%esp)
addl    $4, %esp
    fyl2x          // Stack = y.lg2(10)
    pushl $4
    fildl (%esp)   // Load 4.0 on FPU stack
    addl  $4, %esp
    fdivrp         // Stack = (y.lg2(10))/4
    f2xm1          // Stack = 2^((y.lg2(10))/4) -1
    fld1           // Load 1.0
    faddp          // Stack = 2^((y.lg2(10))/4)
    fmul %st, %st  // Compute Square
    fmul %st, %st  // Square again
    ret            // Result on stack top as 10^y
```

EX 9.16:

```
evalfx:
   // Computes f(x). At entry, x is given on st(0) and st(1)
   // Result is returned in st(0). This function is not
   // C Callable.
   fsqrt           // Stack = (sqrt(x), x)
   fld    %st(1)   // Duplicate x again
   fmul   %st, %st // Stack=(x^2, sqrt(x), x)
   fld    %st(2)   // Stack = (x, x^2, sqrt(x), x)
   fmul   %st(1), %st // Stack = (x^3, x^2, sqrt(x), x)
   faddp           // Stack = (x^3+x^2, sqrt(x), x)
   fsubp           // Stack = (x^3+x^2 - sqrt(x), x)
   ret
/*
 * bisect is C callable function.
 * C Prototype is float bisect(void);
```

```
 */
.data
  float25: .float 0.25
  epsilon: .float 0.001
.text
.globl bisect
bisect:
   // Lower and Upper are 0.25 and 1.0 in the beginning
   fninit
   flds    epsilon    // Load epsilon
   flds    float25    // Load 0.25
   fld1               // and 1 on register stack
   // Stack at (x2, x1, e)
b1:
   fld     %st        // Duplicate %st
   // Compute (x2-x1)/x1. x2 is %st, x1 is %st(2)
   fsub    %st(2),%st // Stack=((x2-x1), x2, x1, e)
   fdiv    %st(2),%st // Stack=((x2-x1)/x1, x2, x1, e)
   fabs               // Absolute value of (delta X)/X
   fcomip  %st(3),%st // Stack=(x2, x1, e)
   jc      done
   // Compute 0.5*(x1+x2)
   fld     %st(0)
   fadd    %st(2), %st(0) // Stack=((x2+x1), x2, x1, e)
   pushl   $2
   fildl   (%esp)
   add     $4, %esp
   fdivrp  %st(1)     // Stack=((x2+x1)/2, x2, x1, e)
   fld     %st        // Duplicate st(0)
   call    evalfx
   // evalfx can consume three more location on stack.
   // Before entry, Stack = (mid, mid, x2, x1, e)
   // After return, Stack = (f(mid), mid, x2, x1, e)
   // Compare fx with 0.0
   fldz
   fcomip %st(1), %st // stack=(f(mid), mid, x2, x1, e)
   fxch
   fstp   %st(1)      // Stack=(mid, x2, x1, e)
   // If CF = 1 if f(mid) > 0, 0 otherwise
   jc     b2
   fstp   %st(2)      // Stack=(x2, mid, e)
   jmp    b1
b2:
   fstp   %st(1)      // Stack=(mid, x2, e)
   jmp    b1
done:
   // At this point Stack=(x2, x1, e)
   // Remove x2 and e from stack and return x1 on st(0)
   fxch
   fstp   %st(1)
```

```
   fstp   %st(1)
   ret
```

EX 10.1:

```
/* C Callable funcation with the following
 * prototype
 * void FloatDivide(float f[16]);
 * Each element of the array f is replaced
 * by a float containing floor(f/16).
 */
.globl FloatDivide
FloatDivide:
   mov  4(%esp), %ebx /* Get Addr in ebx */
   xor  %esi, %esi
L1:
   /* Convert four floats to 4 packed ints */
   cvttps2dq (%ebx, %esi), %xmm0
   psrld $4, %xmm0 /* RgtShft 4 locations */
   cvtdq2ps %xmm0, %xmm1 /* Back to float */
   movups %xmm1, (%ebx, %esi)
   add $16, %esi
   cmp $64, %esi
   jne L1
   ret
```

EX 10.2:

```
/* C Callable function FloatAbs.
 * C Prototype void FloatAbs(float f[4]);
 */
.globl FloatAbs
FloatAbs:
   mov  4(%esp), %ebx // Address in ebx
   mov  $0x7FFFFFFF, %eax
   movd %eax, %xmm1
   punpcklqdq %xmm1, %xmm1
   movdqu %xmm1, %xmm2
   psllq $32, %xmm1
   por %xmm2, %xmm1
   pand (%ebx), %xmm1
   movups %xmm1, (%ebx)
   ret
```

EX 10.3:

```
/* C Callable function.
 * void initCourses(course_data courses[128]);
 */
```

```
.globl initCourses
initCourses:
   pushl $1000 // Svae 1000,0 on stack
   pushl $0
   movq (%esp), %xmm0 // Load 64 bits in xmm0
   addl $8, %esp // Do away the effect of push
   // xmm0[31:0] = 0, xmm0[63:32] = 1000
   pshufd $0x44, %xmm0, %xmm0
   // xmm0: 1000, 0, 1000, 0
   mov $64, %ecx // 64 times in the loop
   mov $0, %esi  // Offset within the array
   mov 4(%esp), %ebx  // Addres of array
I0:
   movdqu %xmm0, (%ebx, %esi)  // Init 2 elements
   add $16, %esi // offset of next 2 elements
   loop I0
   ret
```

EX 10.4:

```
/* C Callable function.
 * void initFlArray(float f[12]);
 */
.globl initFlArray
initFlArray:
   fld1         // Load 1.0 on x87 reg stack
   sub $4, %esp
   fstps (%esp) // Save in local variable
   movd (%esp), %xmm0 // Bring it in xmm0
   add $4, %esp // Recover space on stack
   // xmm0[31:0] = 1.0
   pshufd $0x0, %xmm0, %xmm0
   // xmm0: 1.0, 1.0, 1.0, 1.0
   mov 4(%esp), %ebx // Address of array in %ebx
   movdqu %xmm0, (%ebx)  // Init f[0]..f[3]
   movdqu %xmm0, 16(%ebx)  // Init f[4]..f[7]
   movdqu %xmm0, 32(%ebx)  // Init f[8]..f[11]
   ret
```

EX 10.5:

```
.globl LimitArrayDiff
LimitArrayDiff:
   mov 4(%esp), %esi // Address of a
   mov 8(%esp), %edi // Address of b
   xor %ebx, %ebx // Offset within arrays
L1:
   movdqu (%esi, %ebx), %xmm0 //8 nums from a
   movdqu (%edi, %ebx), %xmm1
```

```
   movdqu %xmm0, %xmm2
   psubw %xmm1, %xmm0 // Compute a[]-b[]
   pcmpgtw %xmm1, %xmm2 // 1s if a[i]>b[i]
   pand  %xmm2, %xmm0
   movdqu %xmm0, (%esi, %ebx)
   add $16, %ebx
   cmp $160, %ebx
   jne L1
   ret
```

EX 10.6:

```
.globl HalfStrength
HalfStrength:
   mov 4(%esp), %esi // Address of s
   xor %ebx, %ebx // Offset within s
   pcmpeqw %xmm2, %xmm2 // Set Xmm2 = all 1s
L1:
   movdqu %xmm2, %xmm0 // Set xmm0 = all 1s
   movdqu (%esi, %ebx), %xmm1 //8 nums from a
   psraw $1, %xmm1 // a[] = a[] >> 1
   pcmpeqw %xmm1, %xmm0 // xmm0=1s if a[]==-1
   pxor %xmm2, %xmm0 // Invert bits of xmm0
   pand %xmm0, %xmm1 // a[i]=0 if a[i] was -1
   movdqu %xmm1, (%esi, %ebx)
   add $16, %ebx
   cmp $160, %ebx
   jne L1
   ret
```

EX 10.7:

```
.globl add_array_float
add_array_float:
   mov 4(%esp), %esi // Address in p
   mov 8(%esp), %edi // Address in q
   xor %ebx, %ebx // Offset within p and q
A1:
   movups (%esi, %ebx), %xmm0 // p[]
   movups (%edi, %ebx), %xmm1 // q[]
   addps %xmm1, %xmm0
   movups %xmm0, (%esi, %ebx)
   add $16, %ebx // Offset for next iteration
   cmp $256, %ebx
   jnz A1
   ret
```

EX 10.8:

```
.globl Op_array_float
Op_array_float:
   mov 4(%esp), %esi // Address in p
   mov 8(%esp), %edi // Address in q
   xor %ebx, %ebx // Offset within p and q
   mov $0x40000000, %eax // Float 2.0 in eax
   movd %eax, %xmm2 // Load 2.0 in xmm2
   pshufd $0, %xmm2, %xmm2 // 2.0 as 4 packed
   mov $0x40400000, %eax // Float 3.0 in eax
   movd %eax, %xmm3
   pshufd $0, %xmm3, %xmm3 // 3.0, 4 packed
A1:
   movups (%esi, %ebx), %xmm0 // p[]
   movups (%edi, %ebx), %xmm1 // q[]
   addps %xmm2, %xmm0 // p[] + 2.0
   mulps %xmm3, %xmm1 // q[] * 3.0
   mulps %xmm1, %xmm0 // (p[]+2)*(q[]*3)
   movups %xmm0, (%esi, %ebx)
   add $16, %ebx // Offset for next iteration
   cmp $256, %ebx
   jnz A1
   ret
```

EX 10.9:

```
.globl Scale_array_float
Scale_array_float:
   mov 4(%esp), %esi // Address in p
   xor %ebx, %ebx // Offset within p
   fildl 8(%esp) // Load r as floating in %st
   sub $4, %esp  // Create space on stack
   fstps (%esp)   // Save r as float in local
   movd (%esp), %xmm1 // Bring r in xmm1
   add $4, %esp  // Recover space on stack
   pshufd $0, %xmm1, %xmm1 // r as 4 packed floats
   mov  $0x7FFFFFFF, %eax // Bit mask for computing
   movd %eax, %xmm2       // abs value of float
   pshufd $0, %xmm2, %xmm2 // Copy as 4-packed
S1:
   movups (%esi, %ebx), %xmm0 // p[]
   // Compute fabs(p[]
   pand %xmm2, %xmm0
   mulps %xmm1, %xmm0 // Compute r*abs(p[])
   movups %xmm0, (%esi, %ebx) // Save
   add $16, %ebx // Offset for next iteration
   cmp $256, %ebx
   jnz S1
   ret
```

EX 10.10:

```
.globl Scale_Reversed_array_float
Scale_Reversed_array_float:
   mov 4(%esp), %esi // Address of p
   mov 8(%esp), %edi // Address of q
   add $(4*60), %edi // Address of q[60]
   xor %ebx, %ebx // Offset within p and q
   fildl 12(%esp) // Load r as floating in %st
   sub $4, %esp  // Create space on stack
   fstps (%esp)   // Save r as float in local
   movd (%esp), %xmm0 // Bring r in xmm0
   add $4, %esp  // Recover space on stack
   pshufd $0, %xmm0, %xmm0 // r as 4 packed floats
S1:
   movups (%esi, %ebx), %xmm1 // p[]
   movups (%edi), %xmm2 // q[63-i..]
   pshufd $0x1b, %xmm2, %xmm2 // Reverse q
   mulps %xmm2, %xmm1 // p[i] = p[i]*q[63-i]
   mulps %xmm0, %xmm1 // p[i] = r*p[i]
   movups %xmm1, (%esi, %ebx) // Save
   add $16, %ebx // Offset for next iteration
   sub $16, %edi // Address of next four q[]s
   cmp $256, %ebx
   jnz S1
   ret
```

EX 10.11:

```
.globl Average
Average:
   mov 4(%esp), %esi // Address of d
   pxor %xmm0, %xmm0 // Set xmm0 = 0
   xor %ebx, %ebx  // Set ebx = 0
A1:
   addpd (%esi, %ebx), %xmm0 // Add 2 doubles
   add $16, %ebx
   cmp $512, %ebx // Done for all 64 doubles?
   jnz A1
   // At this point
   // xmm0[63:0] contain sum of all d[2*i]
   // xmm0[127:64] contain sum of d[2*i+1]
   // for i = 0 to 31.
   movupd %xmm0, %xmm1 // Copy Xmm0 to Xmm1
   shufpd $1, %xmm1, %xmm1 // Exchange two
                     // doubles in xmm1
   addpd %xmm1, %xmm0 // Sum in xmm1 [63:0]
   sub $8, %esp // Create space for a double
   movq %xmm0, (%esp)
   fldl (%esp) // Load sum on x87 stack
   movl $64, (%esp) // Divide by 64
   fildl (%esp) // Load 64 on x87 stack
```

```
   add $8, %esp
   fdivrp // Divide and pop stack
   ret
```

EX 10.12:

```
.globl ComputeAverage
ComputeAverage:
   mov 4(%esp), %esi // Address of records
   // We shall use xmm0 an xmm1 to store
   // 8 sums of 8 floats each in all record
   pxor %xmm0, %xmm0 // Set xmm0 = 0
   movupd %xmm0, %xmm1 // Set xmm1 = 0
   xor %ebx, %ebx  // Set ebx = 0
   mov $0x42800000, %ecx // 64.0 as SP float
   movd %ecx, %xmm3 // Prepare xmm3 = all 64.0
   pshufd $0, %xmm3, %xmm3 // copy 4 times
   mov $64, %ecx
C1:
   // Add 8 floats of a record in 8 floats
   // of xmm0 and xmm1
   addps (%esi, %ebx), %xmm0 // Add 4 floats
   addps 16(%esi, %ebx), %xmm1 // Add 4 more
   add $32, %ebx
   loop C1
   divps %xmm3, %xmm0 // Divide sum by 64
   divps %xmm3, %xmm1
   mov 8(%esp), %ebx // Address of result
   movupd %xmm0, (%ebx)
   movupd %xmm1, 16(%ebx)
   ret
```

EX 10.13:

```
.globl LowPassFilter
LowPassFilter:
   mov 4(%esp), %esi // Address of input arr
   mov 8(%esp), %edi // Address of output arr
   mov (%esi), %ax // Read x0
   mov %ax, (%edi) // y0 = x0.
   // We shall use xmm0 to store 8 shorts
   // as x[i+7]..x[i].
   // We also use xmm1 to store 8 shorts
   // as x[i+6]..x[i-1]
   xor %ebx, %ebx
   mov $8, %ecx // 8 loops of 7 items each.
L1:
   movdqu  (%esi, %ebx), %xmm0
   movupd %xmm0, %xmm1 // xmm1 = x[i+7]..x[i]
```

```
// Shift xmm0 right by 2 bytes so that it
// gets 0, x[i+7]..x[i+1]
psrldq $2, %xmm0
psubw %xmm1, %xmm0 // xmm0=8 words.
// Seven of these are valid as y[i+7]..y[i+1]
// Most significant one is -x[i+7] and will be
// discarded.
movdqu %xmm0, 2(%edi, %ebx) // Store Y
// In the memory, 8 shorts are written. Only 7
// of them are good. The most significant one
// gets overwritten in the next iteration.
// To ensure that the array is not out of bound
// Last iteration is handled differently outside
// the loop.
add $14, %ebx // i = i+7
loop L1
// Do the remaining 7 shorts
movdqu (%esi, %ebx), %xmm0 // Read x[63:56]
movupd %xmm0, %xmm1
psrldq $2, %xmm0 // xmm0=0, x63..x57
psubw %xmm1, %xmm0
// At this point xmm0 contains -x63, y63..y57.
// only 14 bytes are to be written
// First write 4 bytes
movd %xmm0, %eax
mov %eax, 2(%edi, %ebx)
psrldq $4, %xmm0
// Next 4 bytes
movd %xmm0, %eax
mov %eax, 6(%edi, %ebx)
psrldq $4, %xmm0
// Next 4 bytes
movd %xmm0, %eax
mov %eax, 10(%edi, %ebx)
psrldq $4, %xmm0
// Remaining two bytes
movd %xmm0, %eax
mov %ax, 14(%edi, %ebx)
ret
```

Appendix I

References

1. GNU documentation for GNU binutils that include GNU Assemblers (`gas`) and GNU linker (`ld`). The documentation and code can be found at the following web site.

   `http://www.gnu.org/software/binutils/`

2. GNU Compiler Collection (or GCC) includes front ends for C, C++, Assembly and many other languages. The documentation, code and libraries for `gcc` are available at the following web site.

   `http://gcc.gnu.org/`

3. Assembly language programming can be done in several ways. Even the syntax of the instructions vary from machine to machine and assembler to assembler. A huge amount of information is available on Internet on programming using the Assembly language. In particular, the following web site is an extremely good source of information related to programming using Assembly language in GNU/Linux.

   `http://www.linuxassembly.org/`

4. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture. This document can be found at the following web site.

   `http://www.intel.com/design/Pentium4/manuals/253665.htm`

   This document describes the architecture and programming environment of all IA32 architectures from Intel.

5. IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M. This document can be found at the following web site.

   `http://www.intel.com/design/Pentium4/manuals/253666.htm`

6. IA-32 Intel Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z. This document can be found at the following web site.

   http://www.intel.com/design/Pentium4/manuals/253667.htm

   Last two documents describe instructions of the IA32 processors from intel and corresponding bit patterns for the machine instructions.

7. IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1. This document can be found at the following web site.

   http://www.intel.com/design/Pentium4/manuals/253668.htm

8. IA-32 Intel Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 2. This document can be found at the following web site.

   http://www.intel.com/design/Pentium4/manuals/253669.htm

   Last two documents describe the environment of IA32 architectures for memory management, memory protection, task management, interrupt and exception handling and system management. Most of this information is useful for the developers of an Operating System and has been omitted from this book.

9. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. This document is available at the following URL.

   http://www.amd.com/us-en/assets/content_type/

   white_papers_and_tech_docs/24592.pdf

   This document describes the basic execution environment of IA32 processor available from AMD.

10. AMD64 Architecture Programmer's Manual Volume 2: System Programming. This document is available at the following URL.

    http://www.amd.com/us-en/assets/content_type/

    white_papers_and_tech_docs/24593.pdf

    This document describes the systems programming environment of IA32 processors from AMD. It includes information for the developers of Operating systems and other such system programs.

11. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. This document is available at the following URL.

    http://www.amd.com/us-en/assets/content_type/

    white_papers_and_tech_docs/24594.pdf

12. AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions. This document is available at the following URL.

    `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26568.pdf`

13. AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions. This document is available at the following URL.

    `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26569.pdf`

    Last three documents describe the instructions in AMD instruction sets. Most instructions are covered in this book. Some instructions from instruction sets such as SSE3, 3DNow!$^{\text{TM}}$ and extensions to 3DNow!$^{\text{TM}}$ are not covered in this book. These documents are excellent sources of such informations.

14. On-line manual pages for the system calls. For example to see a manual page of `fork` system call, execute a command '`man 2 fork`' on a GNU/Linux machine. On on-line pages actually provide details about the corresponding libc functions but are helpful in understanding the behavior of the system call. In particular, the error return is handled differently. The system calls return negative values for errors while the libc functions return -1 for error and set a global variable called `errno` to contain positive error code.

15. Manual page for `errno` for description of the error values. To see the man page for `errno` issue a command such as '`man errno`' or '`man 3 errno`'.

16. Web resources are available for several uncovered instruction sets such as SSE3, 3DNow!$^{\text{TM}}$ and extensions to 3DNow!$^{\text{TM}}$ instruction set. The best way to start looking for this information is to start a search on Google.

17. The IA32 assembly language discussed in this book is in a format known as AT&T format. Intel defined another format for IA32 assembly language and is known as Intel syntax. Most assemblers on Windows based machines use Intel syntax for assembly language programming. Some assemblers which accept Assembly language programs in Intel format and generate code to run on Windows machine are MASM from Microsoft and TASM from Boreland. Versions of these can be downloaded from various web sites.

On GNU/Linux there are assemblers that accept Intel syntax and generate ELF binaries. One such assembler is NASM that can be downloaded from

`http://nasm.sourceforge.net/`

In addition, the GNU assembler `gas` also supports Intel format which has not been discussed in this book. The Intel format is far more complicated than the AT&T format and requires a larger amount of understanding. The interested readers can find great amount of material describing Intel format on Internet.

18. Among many tools for programming using assembly language, there is one that brings the Assembly language to a higher level of abstraction. The High Level Assembly (HLA) language has been used for teaching at various places and is developed at University of California, Riverside. Various softwares, documentation and code examples can be downloaded from the following web site.

    `http://webster.cs.ucr.edu/`

19. Details about Unicode and ISO/IEC codes for character representation are available at Unicode web site. As of writing this book, Unicode standard 4.0 is the latest standard. The code charts, details about coding standards and references for rendering algorithms can be accessed as Unicode web site as the following.

    `http://www.unicode.org/`

    ISO/IEC standards for ISO8859 and ISO10646 can be bought through ISO web site as the following.

    `http://www.iso.org/`

Index