modes, although building the combination in software will usually take more time than if the combination addressing mode existed in hardware (although there is a trade-off that slows down all operations to allow for more complexity). The x86 instructions use five different operand types: registers, constants, and three memories addressing schemes. Each form is called an addressing mode. The x86 processors support the register addressing mode, the immediate addressing mode, the indirect addressing mode, the indexed addressing mode and the direct addressing mode.

One approach to processors places an emphasis on flexibility of addressing modes. Some engineers and programmers believe that the real power of a processor lies in its addressing modes. Most addressing modes can be created by combining two or more basic addressing modes, although building the combination in software will usually take more time than if the combination addressing mode existed in hardware (although there is a trade-off that slows down all operations to allow for more complexity).

In a purely othogonal instruction set, every addressing mode would be available for every instruction. In practice, this isn't the case.

Virtual memory, memory pages, and other hardware mapping methods may be layered on top of the addressing modes.

7.1 ADDRESSING MODES WITH REGISTER OPERANDS

Register operands are the easiest to understand. Consider the following forms of the mov instruction:

```
mov ax,ax

mov ax,bx

mov ax,cx

mov ax, dx
```

In the above instructions, the first instruction is the destination register and the second operand is the source register. The first instruction does nothing. It copies the value from the ax register back into the ax register. The remaining three instructions copy the value of bx,cx, and dx into ax. Note that the original values of bx, cx and dx remain the same. The first operand (the destination register) is not limited to ax; you can move values to any of these registers. This mode of addressing is the ***register addressing mode***

7.2 ADDRESSING MODES WITH CONSTANTS

Constants are also pretty easy to deal with. Consider the following instructions:

```
mov ax,25

mov bx, 195

mov cx,2056

mov dx,1000
```

These instructions are all pretty straightforward; they load their respective registers with the specified hexadecimal constant. This mode of addressing is called the ***immediate addressing mode.***

7.3 ADDRESSING MODES WITH MEMORY STRUCTURES

There are three addressing modes which deal with accessing data in memory. There are three main addressing mode found in this category; the ***direct addressing mode,*** the ***indirect addressing mode*** and ***the index addressing mode.*** These addressing modes take the following forms:

```
mov ax, [1000]

mov ax, [bx]

mov ax, [1000+bx]
```

The first instruction uses the ***direct*** addressing mode to load ax with the 16 bit value stored in memory starting at location 1000hex.

The second instruction loads ax from the memory location specified by the contents of the bx register. This is an ***indirect*** addressing mode. Rather than using the value bx, the instruction accesses the memory location whose address appears in bx.

There are many cases where the use of indirection is faster, shorter and better. The last memory addressing mode is the ***indexed*** addressing mode. An example of this memory addressing mode is mov ax, [1000+bx]. The instruction adds the contents of bx with 1000 to produce the address of the memory value to fetch. This instruction is useful for accessing elements of arrays, records and other data structures.

7.4 ADDRESSING MODE WITH STACK MEMORY

The stack plays an important role in all microprocessors. It holds data temporarily and stores return addresses for procedures. The stack memory is a LIFO memory which describes the way data are stored and removed from the stack. Data are placed onto the stack with the PUSH instruction and removed with a POP instruction. The stack memory is maintained by two registers (the stack pointer SP or ESP and the stack segment)s. The stack pointer register always points to an area of memory located within the stack segment. The stack pointer adds to (ss*10h) to form the stack memory address in the real mode.

- PUSH ax: Copies ax into the stack
- POP cx: Removes a word from the stack and places it in cx
- PUSH dx: Copies dx into the stack
- PUSH 123: Copies 123 into the stack
- PUSH A: Copies the word content of all the registers into the stack (AX,BX,CX,DX,SI,DI,SP,BP)
- POP A: Removes data from the stack and places it in the 16 bit registers.

Example

.code

```
start:

        mov ax,23

        mov bx,44

        mov cx,13

        push ax; copies 23 into the stack

        push bx; copies 44 into the stack

        push cx; copies 13 into the stack

        pop cx; removes 13 from the stack and places it back into cx

        pop bx; removes 44 from the stack and places it back into bx

        pop ax; removes 23 from the stack and places it back into ax
```

CHAPTER EIGHT

INSTRUCTION SETS

Like any other programming language, there are going to be several instructions you use all the time, some you use occasionally, and some you will rarely, if ever, use. These are called the 80x86 instruction sets.

8.1 THE 80X86 INSTRUCTION SETS

80x86 instructions can be roughly divided into eight different classes;

1. Data movement instructions
    - mov, lea, les, push, pop, pushf, popf

1 Conversions

- cbw, cwd, xlat

2 Arithmetic instructions

- add,inc,sub,dec,cmp,neg,mul,imul,div,idiv

3 Logical shift

- and, or, xor, not, shl, shr, rcl, rcr

4 I/O instructions

- in,out

5 String instructions

- mov,stos,lods

6 Program flow instructions

- jmp, call, ret
- conditional jumps

7 Miscellaneous instructions

- clc, stc, cmc

The most commonly used of all these classes are the data movement instructions and the arithmetic instructions. The mov isnstruction is the most commonly used instruction of all the data movement instruction. The mov instruction is actually two instructions merged into the same instruction. The two forms of the mov instruction take the following forms;

mov reg, reg/memory/constant

mov reg, reg

where reg (i.e register) is any of ax,bx,cx, or dx; constant is a numeric constant (using hexadecimal notation), and memory is an operand specifying a memory location. The next section describes the possible forms the memory operand can take. The “reg/memory/constant” operand tells you that this particular operand may be a register, memory location, or a constant.

The arithmetic and logical instruction take the following forms:

add reg, reg/memory/constant

sub reg, reg/memory/constant

cmp reg, reg/memory/constant

and reg, reg/memory/constant

or reg, reg/memory/constant

not reg/memory

The following sections describe some of the instructions in these groups and how they operate. The 80x86 instruction have simple semantics.

The ***add*** instruction adds the value of the second operand to the first (register) operand, leaving the sum in the first operand.

The ***sub*** instruction subtracts the value of the second operand from the first, leaving the difference in the first operand.

The ***cmp*** instruction subtracts the value of the second operand from the first, leaving the difference in the first operand.

The **and & or** instructions compute the corresponding bitwise logical operation on the two operands and store the result in the first operand.

The ***not*** instruction invert the bit in the single memory or register operand.

8.2 CONTROL TRANSFER INSTRUCTION

The *control transfer instructions* interrupt the sequential execution of instructions in memory and transfer control to some other point in memory either unconditionally, or system organization after testing the result of thr previous cmp instruction. These instructions include the following;

ja dest--jump if above

jae dest--jump if above or equal

jb dest--jump if below

jbe dest--jump if below or equal

je dest—jump if equal

jne dest—jum if not equal

jmp dest-unconditional jump

iret return from an interrupt

The first six instructionsnin this class let you check the result of the previous cmp instruction for greater than, greater or equal, less than or equal,equality, or inequality. For example, if you compare the ax and bx registers with the cmp instruction and execute the ja instruction, the x86 CPU will jump to the specified destination location if ax was greater than bx. If ax is not greater than bx, control will fall through to the next instruction in the program. The jmp instruction unconditionally transfers control to the instruction at the destination address. The iret instruction returns control from an *interrup service routine.*The get and put instructions let you read and write integer values. Get will stop and prompt the user for a hexadecimal value and then store that value into the ax register. Put displays (in hexadecimal) the value of the ax register. The remaining instructions do not require any operands, they are **halt** and **brk.** Halt terminates program execution and brk stops the program in a state that it can be restarted.

8.3 THE STANDARD INPUT ROUTINES

While the standard library provides several input routines, there are three in particular that will be used be used most times.

- Getc (gets a character)
- Gets(gets a string)
- Getsm

Getc reads a single character from the keyboard and returns that character in a register. It does not modiffy any other registers. As usual, the carry flag returns the error status. You do not need to pass getc any values in th rgisters. Getc does not echo the input character to the display screen. You must explicitly print the character if you want it to appear on the output monitor.

The gets routine reads an entire line of text from the keyboard. It stores each successive character of the input line into a byte array whose base address in the es:di register pair. This array must have a room of 128bytes. The gets routine will read each character and place it in the array except for the carriage return character. Gets terminates the input line with a zero byte. Gets echoes each character you type to the display device, it also handles simple line editing functions such as backspace.

The getsm routine reads a string from the keyboard and returns a pointer to that string in nes:di pair register. The difference between gets an getsm is that you do not have to pass the address of an input buffer in es:di. Getsm automatically allocates storage on the heap with a call to malloc and returns a pointer to the buffer in es:di.

8.4 THE STANDARD OUTPUT ROUTINES

The basic standard output routines are; PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT and PRINTF

Putc outputs a single character to the display device. It outputs the character appearing in the al register. It does not affect any registers unless there is an error on output (the carry flag denotes error/no error).

Putcr outputs a “newline” to the standard output.

Puts (put a string) routine prints the zero terminated string at which es:di points. Puts does not automatically output a new line after printing the string.

The puth routine prints the value in the al register as exactly two hexadecimal digits including a leading zero byte if the value is in the range (0..Fh).

The puti routine puts the value in the ax as a signed 16 bit integer

The print routine is one of the most often called procedures in the library. It prints the zero terminated string that immediately follows the call to the string.

Printf uses the escape character (“\”) to print special characters in the fashion similar to, but not identical to C’s printf.

8.5 MACROS

Many assemblers support *macros*, programmer-defined symbols that stand for some sequence of text lines. This sequence of text lines may include a sequence of instructions, or a sequence of data storage pseudo-ops. Once a macro has been defined using the appropriate pseudo-op, its name may be used in place of a mnemonic. When the assembler processes such a statement, it replaces the statement with the text lines associated with that macro, then processes them just as though they had appeared in the source code file all along (including, in better assemblers, expansion of any macros appearing in the replacement text).

Since macros can have 'short' names but expand to several or indeed many lines of code, they can be used to make assembly language programs appear to be much shorter (require less lines of source code from the application programmer - as with a higher level language). They can also be used to add higher levels of structure to assembly programs, optionally introduce embedded de-bugging code via parameters and other similar features.

Many assemblers have built-in macros for system calls and other special code sequences.

Macro assemblers often allow macros to take parameters. Some assemblers include quite sophisticated macro languages, incorporating such high-level language elements as optional parameters, symbolic variables, conditionals, string manipulation, and arithmetic operations, all usable during the execution of a given macros, and allowing macros to save context or exchange information. Thus a macro might generate a large number of assembly language instructions or data definitions, based on the macro arguments. This could be used to generate record-style data structures or "unrolled" loops, for example, or could generate entire algorithms based on complex parameters. An organization using assembly language that has been heavily extended using such a macro suite can be considered to be working in a higher-level language, since such programmers are not working with a computer's lowest-level conceptual elements.