Assembly Language
for the
386/486

Copyright © 1993 Maurice Castro

October 12, 2001

Contents

Course Overview 5

Introduction 9

1 Processor Architecture 11

1.1 Instructions . . . 11
1.2 Integers . . . 13
1.3 Registers . . . 13
1.4 Memory Organization . . . 14
1.5 Memory Hierarchy . . . 18

2 Representation and Organization 19

2.1 Representation . . . 19
2.1.1 Flow Charts . . . 19
2.1.2 Pseudo Code . . . 20
2.2 Program Structure . . . 21
2.2.1 The Top Down Approach . . . 21
2.3 Documentation . . . 22

3 Basic Operations 23

3.1 Basic Memory Access . . . 23
3.2 Operations . . . 25
3.2.1 Assignment . . . 25
3.2.2 Arithmetic . . . 25
3.2.3 Jumps . . . 26
3.2.4 Alternation . . . 26

4 Control Structures 29

4.1 Pre-Test Loops . . . 30
4.2 Post-Test Loops . . . 31
4.3 If-Then . . . 32
4.4 If-Then-Else . . . 33
4.5 If-Then-ElseIf-Else . . . 34

4.6 Switch 35

5 Subroutines - Introduction **37**

6 Addressing Techniques **39**
6.1 Addressing Modes 39
6.1.1 Direct Addressing 39
6.1.2 Indexed Addressing 39
6.1.3 Indirect Addressing 40
6.2 Pointers 42
6.2.1 ‘C’ to Assembler Examples 42

7 Subroutines - Advanced **47**
7.1 Parameter Passing 47
7.1.1 Pass by Register 47
7.1.2 Pass by Stack 48
7.1.3 Pass by Value and Pass by Reference 50
7.1.4 Returning Results 51
7.1.5 Local Variables and Stack Frames 51

8 Data Structures **57**
8.1 Vectors 57
8.2 Arrays 58
8.3 Records 60
8.4 Dope Vectors 60
8.5 Trees and Graphs 61

9 Block Structured Languages **63**

A AT&T Syntax **67**
A.1 Register Set 67
A.2 Flags 69
A.3 Assembler Syntax 70
A.3.1 General Layout 70
A.3.2 Operands 71
A.3.3 Comments 71
A.3.4 Expressions 72
A.3.5 Assembler Directives 72
A.3.6 Memory References 74

B Instruction Set **77**
B.1 Layout 77
B.1.1 Title lines 77
B.1.2 Type & Compatibility 77

B.1.3 Formats 78
B.1.4 Psuedo Instructions 79
B.1.5 Description 79
B.1.6 Flags 79
B.2 Instructions 81

Bibliography **183**

Course Overview

This course aims to provide the student with the skills required to program in assembly language. The 386 processor has been chosen because it is common and commercially relevant. Although this course will concentrate on the 386 processor, many of the skills acquired will have wider application.

The course will cover the following major topics:

- 386 architecture
  - Instructions
  - Integers
  - Register Set
  - Memory Organization
  - Memory Hierarchy
- Flow Charts and Pseudo Code
  - High and Low Level Concepts
  - Flow Chart Elements
  - Pseudo Code Elements
  - Top Down Approach
  - Documentation
- Basic Operations
  - Arithmetic
  - Jumps
  - Alternation

- Control Structures
  - Pre-Test Loops
  - Post-Test Loops
  - If-Then
  - If-Then-Else
  - If-Then-ElseIf-Else
  - Switch
- Subroutines (introduction)
  - Call and Return
- Addressing Techniques
  - Indexing
  - Indirection
  - Pointers
- Subroutines (advanced)
  - Pass by Register
  - Pass by Stack
  - Pass by Reference
  - Pass by Value
  - Returning Results
  - Stack Frames
- Data Structures
  - Vectors
  - Arrays
  - Records
  - Dope Vectors
  - Trees
- Block Structured Languages
  - Scope
  - Implementation

The course assumes a familiarity with at least one high level language. The majority of the examples using high level languages are written in the 'C' programming language. When 'C' does not have an appropriate language concept then the Pascal language will be used.

Introduction

A knowledge of assembly language programming is useful in many areas of computer science. Key among these areas are program optimization both at the user and compiler levels, code generation for compilers, and interfacing hardware.

A processor executes 'machine code'. Assembly language has a one-to-one mapping between its instructions and machine code instructions.

Although it is likely that many students will not be writing compilers or device drivers, all programmers should have an interest in the efficiency of the code they write. An understanding of the low level implementation of the code written in a high level language assists the design of programs in high level languages when speed is required.

With the improvement of compiler technology it is no longer necessary to write routines in assembly language to obtain good performance. However, it is still possible to replace critical routines in a program with carefully constructed assembly language programs to give peak performance. Typically these assembly language routines will reflect some additional knowledge about the problem that cannot be made available to the compiler.

Chapter 1

Processor Architecture

The Intel 386 and 486 processors are closely related. The 486 includes an internal cache, a few additional instructions, and a floating point unit. The differences between the processors mainly concern the systems programmer as the most significant differences relate to cache management and bus locking.

1.1 Instructions

Assembler instruction sets may be divided into categories by varying criteria. Typically the divisions are based on the type of operation, privilege levels, and the type of arguments.

Some of the types of operations are:

Flow of Control Instructions that may cause a change in the order of execution of instructions in a program. For example: Jumps, Conditional Jumps, and Subroutine Calls

Integer Instructions which manipulate integers. For example: arithmetic instructions and logical instructions on integers.

Floating Point Instructions that manipulate floating point values. For example: arithmetic instructions and logical instructions on floats.

Input Output Instructions that manipulate the IO address space.

String Operate on variable length vectors of similar items. For example: Memory Copying, and String Compares.

Divided by privilege:

Non-Privileged: Non-Privileged instructions may be executed by any process. Typically this group of instructions include arithmetic, logical, and most flow of control instructions.

Privileged: Privileged instructions must be executed by processes running at an appropriate privilege level and may include input output instructions, instructions which alter the privilege level, and instructions related to external events (eg. interrupts).

Using argument types:

Memory to Memory: Operations which take an argument from memory, transform it, and record the result in memory.

Memory to Register: Operations which take an argument from memory, transform it, and record the result in a processor register.

Register to Memory: Operations which take an argument from a processor register, transform it, and record the result in memory.

Register to Register: Operations which take an argument from a processor register, transform it, and record the result in a processor register.

and in a segmented architecture:

Single Segment: *For arithmetic or logical instructions:* Instructions which take data from one segment, and transform it. The result may either be left in a register or the result may be written into the same segment as the source.

For instructions which control the order of execution of a program: Instructions which may cause control to be transferred to code in the same segment.

Multi Segment: Instructions that either transfer data between segments, or may cause code to be executed in another segment.

The 386/486 has a segmented architecture which supports the majority of the classes of instructions described above. A key feature of the architecture of the 386/486 is that, except for string instructions, the 386/486 does **not** support **memory to memory** operations. This implies that moving data from one location to another typically involves a memory to register move followed by a register to memory move. Although this may seem to be inefficient, but it can be easily shown that there are few occasions where the optimal coding of an algorithm includes memory to memory operations. Typically the result of an operation is used in the next phase of the program, in addition to being stored in memory. By retaining the result in a register the result is readily available for subsequent operations.

1.2 Integers

The 386/486 supports 3 sizes of integers: 8 bits, 16 bits and 32 bits. The GNU As assembler defines the sizes as byte (8 bits), word or short (16 bit), and int or long (32 bit).

The 386/486 uses a **little endian** encoding of its integers. In a little endian system the low order byte is stored at the low address in memory. A big endian system stores the high order bits at the low address. (figure 1.1)

The hexadecimal number 5A4B3C2D may be represented in a computer's memory in two ways:

Little Endian:

$m+3$	$m+2$	$m+1$	m
$5A$	$4B$	$3C$	$2D$

Big Endian:

$m+3$	$m+2$	$m+1$	m
$2D$	$3C$	$4B$	$5A$

Figure 1.1: Big and Little Endian Numbers

The size of the operand of an instruction is determined by appending either a 'b' (8 bit), 'w' (16 bit), or an 'l' (32 bit) to the mnemonic.

The GNU As assembler uses the conventions of the 'C' programming language to represent numbers. Hexadecimal numbers are prefaced by **0x**, octal values by **0** and decimals begin with any digit other than zero.

1.3 Registers

The 386/486 is unusual among the current generation of microprocessors as it is not a general register processor. Specific registers on the 486 are dedicated to performing specific functions. This is unusual as it increases the difficulty in optimizing code, often requiring that information be shuffled between registers or out to memory before performing an operation.

The term 'general register' has several meanings. When this term is applied to the 386/486 it is taken to mean one of the set of registers %eax, %ebx, %ecx, and %edx. In wider usage 'general register' implies that the processor does not have registers tied to specific functions. However, the registers on a 386/486 are assigned specific functions for given operations, implying that the 386/486 is not a 'general register processor'.

The register set of the 386/486 may be accessed in 8 bit, 16 bit and 32 bit size units. The names of the major units and an example of deriving the subunits names are shown in figure 1.2.

The registers are named according to function. The general registers are %eax, %ebx, %ecx, and %edx. They are known respectively as the accumulator, base register, count register and data register. The index registers %esi and %edi are known as the source index register and the destination index register. The pointer registers %ebp and %esp are called the base pointer and the stack pointer. The segment registers %cs, %ds, %es, %ss, are known as the code segment register, the data segment register, the extra segment register, and the stack segment register. The two additional segment registers %fs and %gs are not named.

Although specific functions are assigned to the general registers and the indexes for a few functions, for other operations they may be used interchangeably.

In addition to these registers the 386/486 has a **flag register**. This register contains a set of bits which are set according to changes in the state of the processor, and arithmetic operations.

The flag register known as **eflags** and the meanings of its bits are illustrated in figure 1.3.

1.4 Memory Organization

Each address in the 386/486 consists of 2 parts: segment and offset. The segment component of the address is loaded into a segment register, and instructions either explicitly mention a segment register, or implicitly use a segment register when accessing memory. The offset component specifies the distance into the segment of the memory location that is to be referenced. Figure 1.4 shows the most general representation of segmentation.

A segment is a contiguous region in memory. Segments may be disjoint or overlap other segments. In addition a segment may be a subset or superset of other segments. A segment is defined by a base, an extent and a set of rights that users of the segment may exercise.

For simplicity the examples and exercises given in this course will be conducted in '32 bit flat mode'. This is the Intel terminology for a 386/486 processor where all the segment registers have been loaded with descriptors for the complete logical address space of the system. The programmer perceives the memory as a 4 Gigabyte contiguous array of bytes. Offsets, relative to any segment, map to the same location and value in memory. Offsets, in this mode, are equivalent to the absolute addresses.

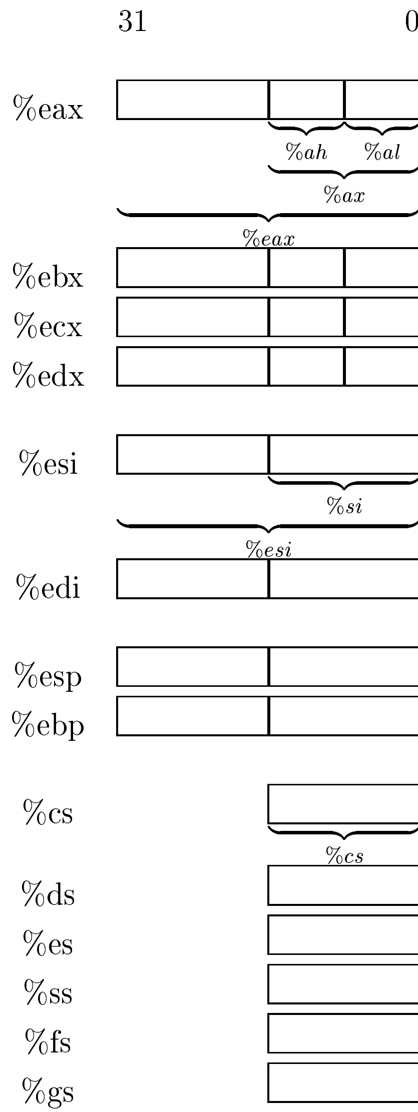


Figure 1.2: The 386/486 register set

31 0

AC Alignment Check

VM Virtual 8086 Mode

RF Resume Flag

NT Nested Task Flag

IOPL I/O Privilege Level

OF Overflow Flag

DF Direction Flag

IF Interrupt Enable Flag

TF Trap Flag

SF Sign Flag

ZF Zero Flag

AF Auxiliary Carry Flag

PF Parity Flag

CF Carry Flag

Figure 1.3: The EFLAGS register

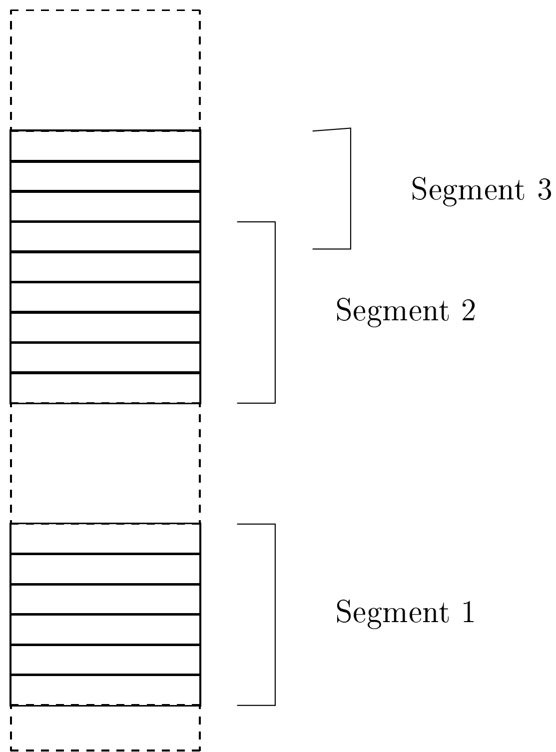


Figure 1.4: Segmentation

1.5 Memory Hierarchy

For the application programmer using an assembler there exists a two stage memory hierarchy: Registers and Main Memory. Access to registers is significantly faster than access to main memory. However, there are a strictly limited number of registers available to the programmer. By storing frequently used values in registers a program's execution time may be reduced significantly.

Chapter 2

Representation and Organization

A computer program is a specific representation of an algorithm written in a programming language. The abstraction - algorithm - may be expressed in many ways. Two will be considered in this chapter: Flow Charts and Pseudo Code. The balance of the chapter will be devoted to describing the structure of programs and documentation.

2.1 Representation

2.1.1 Flow Charts

The flow chart is a method of pictorially representing an algorithm. It represents the 'flow of execution' or the 'sequence of operations' in a codified form. The basic elements of a flow chart are shown in figure 2.1. Arrow heads are used to represent the path through the chart, however, in the absence of arrows, it is assumed that vertical lines are traversed in the downward direction.

In recent years the flow chart has come to be regarded as a poor method of representing algorithms. Some of the reasons for this are:

- A flow chart can become too complex to be easily interpreted.
- A flow chart does not clearly distinguish between the structural elements of a high level language. (do loops, while loops, for loops, and conditionals are all represented by the same construct in a flow chart)
- The majority of programmers no longer work in assembler.

For the assembly language programmer the majority of these reasons are not valid. The complexity of a flow chart can be managed by the person drawing the flow chart. Problems can be broken down into independent subsections of manageable size, flow charts can be drawn for these parts and a flow chart can be drawn to show how the subsections should be executed. The majority of assembly

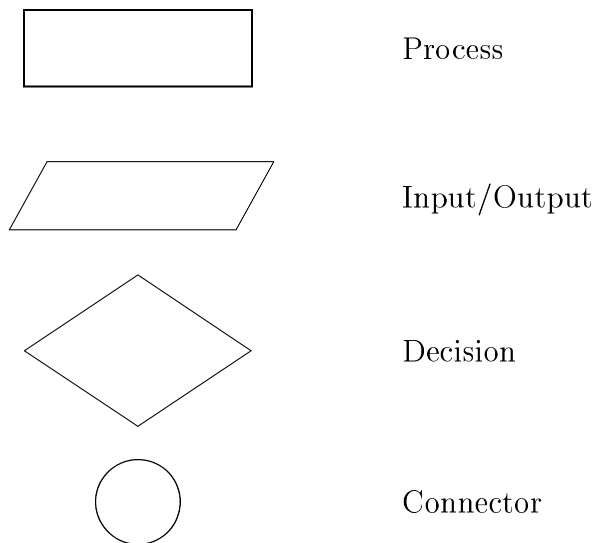


Figure 2.1: The elements of a flow chart

languages support only the structural elements that may be represented easily in a flow chart.

2.1.2 Pseudo Code

Pseudo code is a form of structured English used to represent algorithms. Keywords are used with descriptions of actions and conditions to form a representation of an algorithm. Pseudo code is more efficient for representing algorithms than English alone, as the narrative description is too verbose, and often ambiguous.

The keywords used in Pseudo Code are typically:

- **start** . . . **stop**
- **if** . . . **then** . . . **else**
- **repeat** . . . **until**
- **while** . . . **do**

Indentation is used to group operations, and comments are enclosed by '{' and '}'.

2.2 Program Structure

An algorithm in its most abstract form describes the steps in performing an operation without being concerned with the detail of a particular implementation. This is a **high level** view of a problem. A **low level** view consists of the details required for a specific implementation. Assembly language programming consists of implementing high level concepts in a low level representation. To assist in this process a **Top Down Approach** may be taken.

Assembly language programming requires strict attention to the structuring of programs. If the structure of programs is ignored then maintenance and debugging are made more complex. In addition, the readability of the program code is reduced.

2.2.1 The Top Down Approach

The top down approach consists of breaking a problem up into parts. The parts are broken up into smaller components until a sufficiently simple task is found, such that it can be implemented in a straight forward manner in the application language. This approach is also known as **Stepwise Refinement**.

The advantages of this approach are that it allows the programmer to solve manageable problems and then connect these solutions to solve a larger problem. If a fault is discovered in one of the components of the solution then only a subset of the components of the program needs to be examined and corrected. In addition if another programmer needs to modify the existing code then he/she need only understand the abstract meanings of the lower level modules, so that they can give attention to the area requiring modification without requiring full understanding of the details of the complete program.

2.3 Documentation

Programs are documented both internally and externally. Internal documentation consists of **comments** in the program code. A comment describes the purpose of a piece of code with relation to the problem, not what the code does at a low level. For example the comment "adds one to register EAX" is considerably less useful than the comment "setup to examine next array element" although they both describe the line of code:

$$addl\ \$1,\ \%eax$$

The C commenting style `/*` ... `*/` is used in GNU as.

External documentation consists of a description of how the program works in abstract terms (an algorithm), notes about any short comings or limitations of the program, and details of unusual or in-obvious features of the code.

Both internal and external documentation are required to fully document a program. In assembly language programming, good documentation practices are required as, often, the structure and meaning of a piece of code cannot be easily determined from the code itself.

Chapter 3

Basic Operations

Imperative programming languages support several fundamental classes of operations. This chapter discusses a subset of the operations available on the 386/486 divided into four classes: assignment, arithmetic, jumps and alternation. The definitions of the four classes are:

Assignment - Storing values.

Arithmetic - Operations on numbers.

Jumps - Causing the executing of an instruction other than the instruction immediately following the current instruction.

Alternation - Causing the executing of an instruction other than the instruction immediately following the current instruction based on some condition.

However, before discussing the basic operations, the concepts of values and addresses are introduced and the syntax for the GNU As assembler for simple accesses to memory is covered. The concepts of indexing and indirection will be dealt with in chapter 6.

3.1 Basic Memory Access

A colorful metaphor used for the memory of a computer is a bank of pigeonholes where letters are placed for collection by guests at a hotel. Each pigeonhole has a room number on it to uniquely identify it to the clerk at the desk, and it has space for only one message.

Using this metaphor, the address of a memory location is the room number. The address is unique in the computer. The contents of a memory location or its value, is the message contained within the pigeonhole. As only one message can fit at a time, a new message must displace any message that is already there.

The metaphor can be further extended if the clerk is allowed to write, next to the room number on the pigeonholes a number of names. This is similar to the concept of labels. Hence Mr Smith can get his message, by saying 'I am Mr Smith may I have my message please' or 'The message for room 101 please'.

An example of a short assembly language program:

```
start:
        movl d1, %eax   /* Get the data value */
        addl d2, %eax   /* Add the value of data2 to data1 */
        addl $2, %eax   /* Add 2 to the sum */
        movl %eax, 100  /* Store the result at location 100 */
        jmp exit
s1:
d1:     .long 4
d2:     .long 5
```

This short assembly language program illustrates the concepts of using and declaring labels, using addresses, and constants. The label *d1* is assigned to a long integer which initially contains the value 4. *d2* is assigned to a long integer which initially contains the value 5. The result of the arithmetic operations is placed at address 100. It is also clear that if the program is run before the contents of *d1* or *d2* are changed then the result stored at location 100 will be 11.

The declaration of a label under GNU As consists of a name followed by a colon. Several labels may refer to the one location. Thus in the above example *s1* is a synonym for *d1*.

Labels beginning with an 'L' are local labels and are not visible at link time. In addition there are ten local symbol names '0' to '9'. These are reusable within a program, and references may be made to the nearest forward ('f') or backward ('b') reference by writing *label***f** or *label***b**, respectively.

In hand coded assembler, the practice of using local symbols and local labels is strongly discouraged as it makes assembly code signigicantly more difficult to read and debug. Macros are the single exception to this rule. Local symbols simplify the writing of macros by allowing relatively context insensitive macros to be written which contain loops.

In addition to their significant role in macros, local symbols and local labels are typically heavily used in the output of compilers.

Memory is declared and initialized using the assembler directives **.byte**, **.word**, **.int**, and **.long** where bytes are 8 bits in length, words 16 bits, and integers and longs 32 bits in length. The declarations must be followed by a list of numbers, and these numbers are placed into memory to initialize the memory locations. If no numbers follow the declaration then, no space is reserved by the declaration.

The following code fragment illustrates the reservation of memory and initialization of memory locations:

```
.long 5      /* reserves 32 bits and places 5 in it */
.long 5, 7   /* reserves two longs and places 5 and 7 in them */
.byte 4, 6   /* reserves two bytes and places 4 and 6 in them */
.long        /* reserves no space */
```

Strings may be stored in memory using the **.ascii** and **.asciz** assembler directives. These directives store a series of bytes with the values of the string that follows the directive into memory. The **.asciz** form appends a byte containing a zero to the end of the string.

The following two lines of code are examples of the use of the **.ascii** and **.asciz** directives.

```
.asciz ``A string terminated by a NULL''
.ascii ``A string not terminated''
```

Immediate constants are formed by prepending a '$' to a label or a value. This construct may be used to get the address of a label, typically before passing that address to a subroutine. For example:

```
movl $10, %eax  /* Copy 10 to EAX */
movl $s1, %eax  /* Copy the address of label s1 into EAX */
```

A final note, GNU As uses the AT&T format for instructions. This format places the destination in the rightmost operand. The majority of 386/486 assemblers use the Intel format which places the destination in the leftmost operand.

3.2 Operations

3.2.1 Assignment

The fundamental assignment operation provided by the 386/486 is the **mov** instruction. This operation copies data from source to destination without altering the source or the flag registers.

The following lines of code illustrate the syntax of assignment operations under GNU As.

```
movl 10, %eax    /* Copy contents of address 10 into EAX */
movl $10, %eax   /* Put the value 10 into register EAX */
movl %ebx, %eax  /* Copy the value of EBX to EAX */
movb %edx, 10    /* Copy the low byte of EDX to location 10 */
```

3.2.2 Arithmetic

The 386/486 supports a wide range of arithmetic and bitwise logical operations on integers. The operations include: add, bitwise and, divide, integer divide, integer multiply, multiply, negate, bitwise not, bitwise or, rotate, shift, subtract and bitwise exclusive or. The format for these operations is typically: *op src*, *dst*

where the result is calculated by $dst = dst\ op\ src$. The most notable exception to this formating is the integer multiply instruction which has a three-operand form. A detailed description of the multiply instruction is found in appendix B. A set of two-operand form examples are below:

```
addl 10, %eax    /* Add the contents of address 10 to EAX */
subl $10, %eax   /* Subtract 10 from EAX */
xorl %ebx, %eax  /* EAX = EAX xor EBX */
addb %edx, 10    /* add the low byte of EDX to location 10 */
```

3.2.3 Jumps

The 386/486 supports a large number of types of jumps and subroutine calls. The five forms are defined as:

Absolute - Jump to a specified location

Relative - Jump to a location calculated by adding a signed offset to the address of the instruction following the jump instruction.

Intersegment - Jump to a location in another segment.

Indirect - Jump to a location given in either a register or a memory location.

Indirect Intersegment - Jump to a location defined by a segment offset pair given in memory location.

In this book we will be developing only single code segment programs, and the assembler will treat the jump or call mnemonic as a relative jump, or call of a sufficiently large magnitude.

```
jmp exit     /* Jump to the label exit */
call subone  /* call the function beginning at subone */
```

are examples of the syntax of relative jumps. Jumps to absolute addresses may be formed by prefixing the address with a '*'. Otherwise, the assembler will choose program counter relative addressing.

3.2.4 Alternation

The 386/486 implements alternation through the use of conditional jumps. The conditions used to determine whether to jump or not are based on combinations of bits in the EFLAGS register. It is necessary for the programmer to ensure that the appropriate bits are set in the EFLAGS register before using the conditional jump instruction to test for the condition. A typical example would be:

```
cmpl $0, %eax  /* Set flags in EFLAGS */
je zero        /* Jump to zero if EAX equals zero */
jmp nonzero    /* Not zero, jump to nonzero */
```

The compare instruction (cmp) performs the subtraction $EAX - 0$. Setting the required flags in EFLAGS but otherwise not altering any registers. Je tests the zero flag (zf) in EFLAGS, if the flag is set a jump occurs to the label zero.

The **test** and **cmp** operations set flag bits without altering either memory or general register contents. Arithmetic and bitwise logical operations alter both the flag bits and the destination of the operand of the instruction. As these operations affect the EFLAGS register conditional jumps may be used to detect the results of these operations.

The operations **mov**, **jmp** and **call** do not typically affect flag bits.

A notable feature of the architecture of the 386/486 is that all conditional jumps are implemented as relative jumps.

Chapter 4

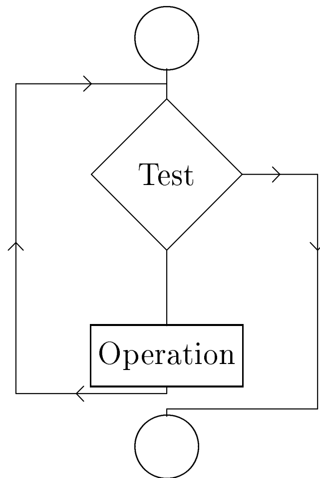
Control Structures

This chapter will illustrate typical control structures found in assembly language programs. The control structures shown in this chapter may be nested, but they should not be overlapped, when writing structured programs.

Although there are several methods of implementing the conditional structures, only one method is shown and described as an example. Provided only one method of implementing conditionals is used within a program, it is possible to construct structured programs which are easily readable.

4.1 Pre-Test Loops

Pre-test loops test that a condition is satisfied before entering the body of the loop. This class of loop is represented by the *while ... do* in pseudo code and the *while* loop in C.



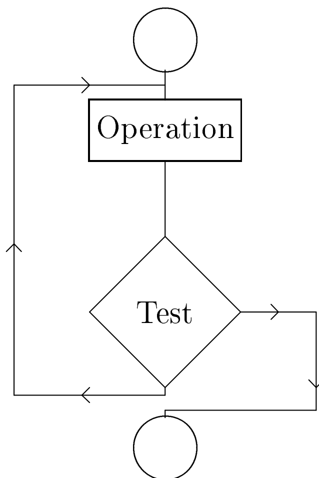
Example:

```
while z not equal 0 do
      a = a + a
      z = z - 1
```

```
ploop:
        cmpl $0, z      /* test if z is zero */
        je eloop
        movl a, %eax    /* let a equal a + a */
        addl %eax, a
        dec z           /* subtract 1 from z */
        jmp ploop
eloop:                  /* exit the loop */
```

4.2 Post-Test Loops

Post-test loops test that a condition is satisfied after executing the body of the loop. This class of loop is represented by the *repeat* ... *until* in pseudo code and the *do* ... *while* loop in C.



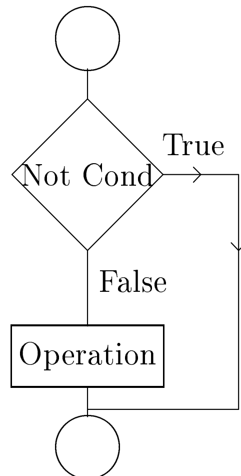
Example:

```
repeat
      a = a + a
      z = z - 1
until z equal 0
```

```
ploop:
        movl a, %eax    /* let a equal a + a */
        addl %eax, a
        dec z           /* subtract 1 from z */
        cmpl $0, z      /* test if z is zero */
        je eloop
        jmp ploop
eloop:                  /* exit the loop */
```

4.3 If-Then

The **If** . . . **Then** conditional may be expressed in assembly language by testing for the negation of the condition. If the negation is true then the consequence - the then clause - is skipped.



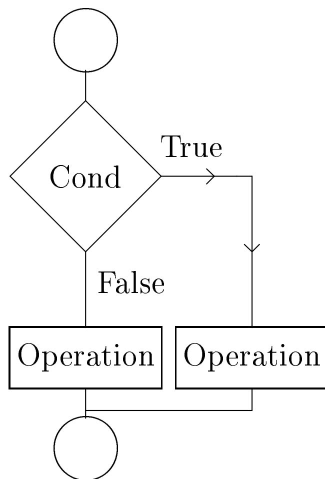
Example:

```
if z equal 0 then
      a = 1
```

```
         cmpl $0, z   /* test if z is zero */
         jne ethen
         movl $1, a   /* let a equal 1 */
ethen:                /* exit the conditional */
```

4.4 If-Then-Else

The **If** ... **Then** ... **Else** conditional is expressed in assembly language as a test for the condition: If the condition is met, then a jump to the 'true' code is made, otherwise the 'false' code is executed.



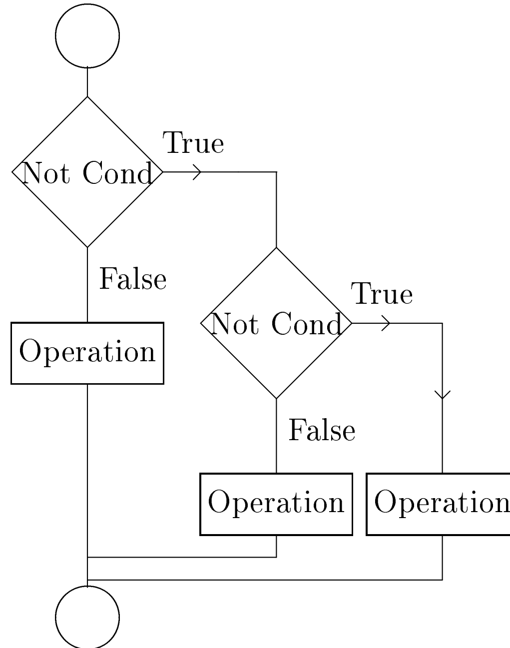
Example:

```
if z equal 0 then
      a = 1
else
      a = 2
```

```
        cmpl $0, z    /* test if z is zero */
        je then
        movl $2, a    /* let a equal 2 */
        jmp ethen
then:
        movl $1, a    /* let a equal 1 */
ethen:                /* exit the conditional */
```

4.5 If-Then-ElseIf-Else

The **If** . . . **Then** . . . **ElseIf** . . . **Else** conditional is a combination of the techniques for **If** . . . **Then** and **If** . . . **Then** . . . **Else**.



Example:

```
if z equal 0 then
      a = 1
elseif z equal 1 then
      a = 2
else
      a = 3
```

```
        cmpl $0, z   /* test if z is zero */
        jne eif1
        movl $1, a   /* let a equal 1 */
        jmp eelse
eif1:   cmpl $1, z   /* test if z is one */
        jne else
        movl $2, a   /* let a equal 2 */
        jmp eelse
else:   movl $3, a   /* let a equal 3 */
eelse:               /* exit the conditional */
```

4.6 Switch

The switch or case statement may be implemented in two ways: the first is to use the **If** ... **Then** ... **ElseIf** ... **Else** construct (see Section 4.5). The second method is to use a jump table. A vector of jump addresses is calculated for each possible input value, and the input values are used as an index into the table. This technique provides quick execution. This technique is similar to that used for dope vectors (see Section 8.4).

Chapter 5

Subroutines - Introduction

The subroutine is the primary mechanism used in structured programming to allow the division of large programs into more managable smaller parts. This chapter will introduce the concepts of a subroutine, and a 'process' or 'system' stack.

A subroutine is defined as a section of program code which may be invoked with a set of parameters, perform an action and which may return a result.

The stack data structure is comprised of a list of elements which may only be accessed from one end. There are two operations defined over a stack. The first operation is PUSH, this inserts an element at the head of the stack. The second operation POP, removes an element from the head of the stack. The 'system' or 'process' stack is provided by the operating system, and it is operated on by processor operations that use a stack. The operations **push** and **pop** are provided by the 386/486, and operate on the register %esp, also known as the stack pointer.

The system stack on the 386/486 grows *downwards* in memory. Each time an item is pushed onto the stack the stack pointer is decremented. As items are removed from the stack the stack pointer is incremented.

The 386/486 provides two operations to support subroutines. The first operation CALL (**call**) causes the address of the instruction following the call instruction to be pushed onto the system stack, and control to be passed to the address contained in the operand of the call instruction. The RETURN instruction (**ret**) pops an address of the stack and transfers control to that address.

The intrinsic mechanisms provided by the processor allow for *nested* subroutine calls. Nested subroutine calls are calls on subroutines from within a subroutine. The stack provides a history of the return addresses of the subroutine calls.

Figure 5.1 ilustrates the basic stack subroutine relationship for the following program assembled into addresses 1000_{10} to 1023_{10}.

```
1000  start:   call subone
1005           jmp exit     /* exit the program */
1010  subone:  call a       /* subroutine subone*/
1015           call b
1020           ret
1021  a:       ret          /* subroutine a */
1022  b:       ret          /* subroutine b */
```

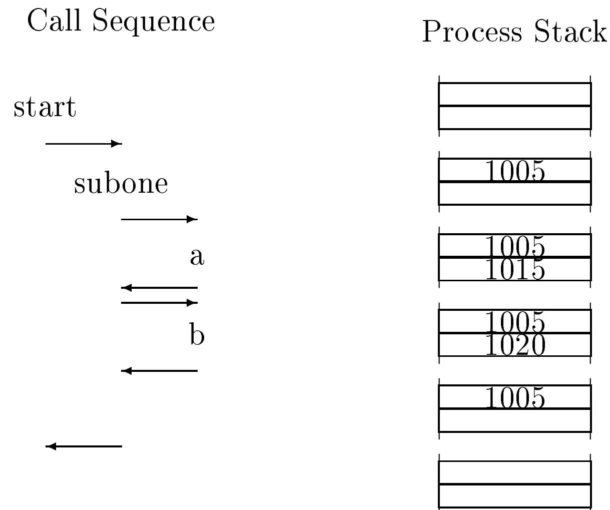


Figure 5.1: Nested Subroutines and the System Stack

Chapter 6

Addressing Techniques

Memory may be referred to by 3 distinct methods known as addressing modes: **direct**, **indexed** and **indirect**. These modes may be combined to form more complex addressing mechanisms. This chapter will define the 3 addressing modes and each mode's availability on the 386/486, and relate the concepts of non-direct addressing to pointers in high level languages.

6.1 Addressing Modes

6.1.1 Direct Addressing

Direct addressing was introduced in section 3.1. Direct addressing is the simplest mode. Essentially direct addressing returns the value found in the memory location specified in the instruction.

6.1.2 Indexed Addressing

Indexed addressing takes a start address and an offset, and returns the contents of the memory location with the address resulting from the addition.

The 386/486 supports indexed addressing using registers to represent the base address and the index. The AT&T syntax for indexed memory references is:

$$segment : disp(base, index, scale)$$

The index *index* is multiplied by the scale factor *scale* and summed with the displacement *disp* and the offset *base* to give the address of the memory location relative to the segment *segment*. (See figure 6.1). In addition to the restriction requiring base and index to be registers, scale is required to have only the values 1, 2, 4, 8 or none.

The syntax of indexed access may be explicitly written as:

$$\left\{\begin{array}{c}\%cs\\ \%ds\\ \%es\\ \%ss\\ \%fs\\ \%gs\end{array}\right\}:\left\{\begin{array}{c}No\ Displacement\\ 8-Bit\ Displacement\\ 32-Bit\ Displacement\end{array}\right\}(\left\{\begin{array}{c}\%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\},\left\{\begin{array}{c}\%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\},\left\{\begin{array}{c}1\\ 2\\ 4\\ 8\end{array}\right\})$$

The effective address of the memory location is calculated using the formula:

$$\left\{\begin{array}{c}\%cs\\ \%ds\\ \%es\\ \%ss\\ \%fs\\ \%gs\end{array}\right\}+\left\{\begin{array}{c}No\ Displacement\\ 8-Bit\ Displacement\\ 32-Bit\ Displacement\end{array}\right\}+\left\{\begin{array}{c}-\\ \%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\}+\left\{\begin{array}{c}-\\ \%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\}*\left\{\begin{array}{c}-\\ 1\\ 2\\ 4\\ 8\end{array}\right\}$$

If the segment modifier is not included then the instruction uses the default segment. If the displacement is not included then a displacement of zero is assumed.

Not all arguments of the index syntax are required to be present. Valid forms of the index syntax are:

$(base, index, scale)$ Complete form

$(base, index)$ Scale defaults to 1

$(base)$ Index defaults to 0

$(, index, scale)$ Base defaults to 0

$(, index,)$ Base defaults to zero and scale defaults to 1

6.1.3 Indirect Addressing

Indirect addressing consits of extracting the address of the destination location from the location named in the instruction. Thus a location contains the address of the location containing the required value.

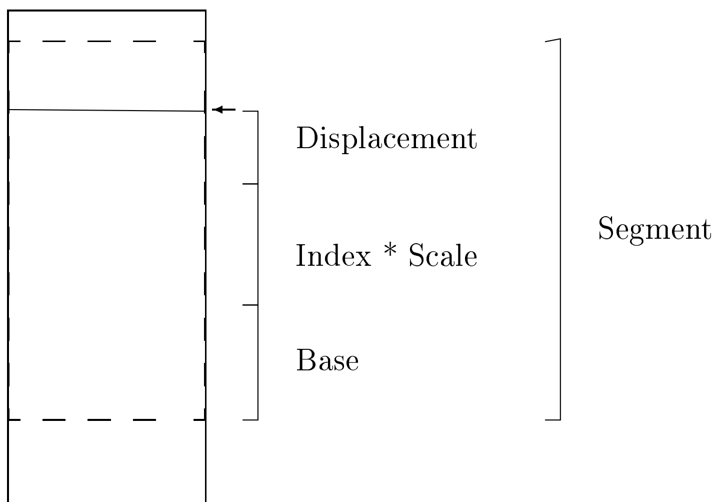
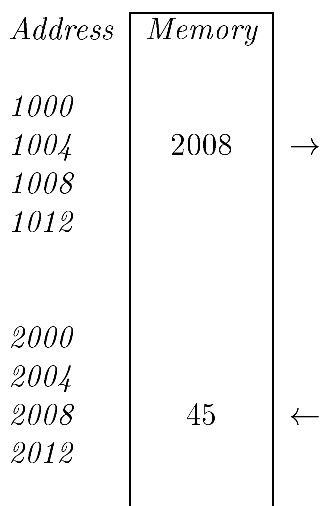


Figure 6.1: Indexed Addressing



If location *1004* is accessed indirectly the value returned will be 45 as location *1004* contains the address of location *2008* which contains the value 45.

Figure 6.2: Indirect Addressing

This is ilustrated in figure 6.2.

The 386/486 provides minimal support for indirect addressing. Specifically, it is available for *mov* and jump commands, however, only moves to and from the register %eax are supported.

Some versions of the GNU As assembler do not correctly support access via indirect addressing. Practical work in this course will not use indirect addressing.

6.2 Pointers

In a high level language, a pointer is a variable that contains an identifier that allows access to either a data item or a procedure. Pointers are typically composed of the address of an object.

The assembly language concept of non-direct access is similar. The address of an item is used to refer to the item. This concept can be extended to describe each element of an object as a data item at an offset from the base of the object.

6.2.1 'C' to Assembler Examples

Several examples of 'C' programming constructs will be presented with translations into assembly language showing how non-direct access may be used to implement the constructs of a high level language, and how indexing construct in assembly language may be used.

Arrays

C

```
int array[10];

/* ... */

array[4]++;
```

Assembler

```
array: .fill 10, 4, 0

/* ... */

movl $4, %eax
incl array(,%eax,4)
```

The array consists of 4 byte objects. Ten sets of 4 byte objects initialized to zero are created by the assembler directive **.fill**. The register %eax is loaded with index value 4 and the operation is performed after indexing into the array.

The following routine performs a similar task except on character size objects. To take account of the size change it is necessary to alter both the size of the memory operand and the scale factor.

C

```
char array[10];

/* ... */

array[4]++;
```

Assembler

```
array: .fill 10, 1, 0

/* ... */

movl $4, %eax
incb array(,%eax,1)
```

Structures

C

```
struct point
{
      int x;
      int y;
      char color;
      };
struct point first;

/* ... */

first.x = 1;
first.y = 2;
first.color = 0;
```

Assembler

```
/* point consists of 2 * 4 byte fields followed by
/* a 1 * 1 byte field */
first: .space 9, 0

/* ... */

/* get address of structure into a register */
movl $first, %eax
/* offset of x = 0 */
movl $1, 0(%eax)
/* offset of y = 4 */
movl $2, 4(%eax)
/* offset of color = 8 */
movb $0, 8(%eax)
```

Arrays of Structures

C

```
struct atom
{
      short id;
      char x;
      char y;
      };
struct atom cloud[1000];

/* ... */

cloud[4].id = 4;
cloud[4].x = 2;
cloud[4].y = 1;
```

Assembler

```
/* atom consists of 1 * 2 byte fields followed by
/* a 2 * 1 byte field */
cloud: .fill 1000, 4

/* ... */

/* get address of structure into a register */
movl $cloud, %eax
/* set up index value */
movl $4, %ebx
/* offset of id = 0 */
movb $4, (%eax,%ebx,4)
/* offset of x = 2 */
movl $1, 2(%eax,%ebx,4)
/* offset of y = 3 */
movl $2, 3(%eax,%ebx,4)
```

Chapter 7

Subroutines - Advanced

Chapter 5 introduced the basic concepts of the 'system' or 'process' stack, and the subroutine, these concepts will be expanded upon in this chapter by introducing techniques for parameter passing, local variables, and returning results.

7.1 Parameter Passing

The parameters of a subroutine are the values that are passed to a subroutine for it to operate on. There are two basic methods of passing parameters - by stack and by register - which may be combined to yield hybrid methods.

Parameters may be divided into the two classes, reference parameters and value parameters.

This section will cover the definition, implementation and characteristics of passing methods and parameter types.

7.1.1 Pass by Register

This is the simplest form of parameter passing. The information to be passed to the subroutine is loaded into registers and the subroutine called.

```
	movl $1, %eax
	movl $2, %ebx
	call trivadd

	/* ... */

trivadd:
	addl %ebx, %eax
	ret
```

The advantage of this form is that it permits the subroutine direct access to the parameters in registers. As registers are the fastest form of storage available to the processor this permits fast subroutines to be written.

Pure register passing is limited in the number and type of values that can be passed to a subroutine. This limitation is imposed by the number and size of available registers. There are additional costs in using register based passing. These result from the need to save values that were previously in registers before setting up for a call. Restoring the registers is necessary if the values are to be used after returning from the call.

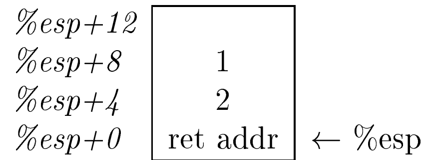
7.1.2 Pass by Stack

Passing values using the stack permits greater flexibility than passing by register. Provided there is sufficient space on the stack, any type and number of values may be transferred as parameters to a subroutine using stack based passing.

Parameters are pushed onto the stack before the subroutine is called. Indexed addressing relative to the stack pointer is used to recover the values of the parameters.

```
	pushl $1
	pushl $2
	call trivadd
	add $8, %esp

	/* ... */

trivadd:
	movl 4(%esp), %ebx
	movl 8(%esp), %eax
	addl %ebx, %eax
	ret
```

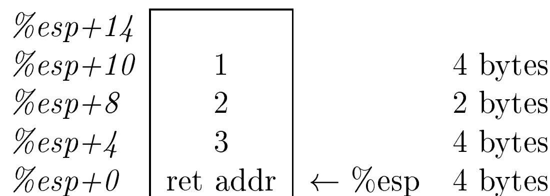
The stack can be represented diagramatically:



Parameters passed to a function may be of varying sizes. The following program fragment shows an implementation of a function which takes a long integer, followed by a word-sized integer, followed by another long integer.

```
    pushl $1
    pushw $2
    pushl $3
    call oddadd
    addl $10, %esp

    /* ... */

oddadd:
    movzwl 8(%esp), %eax
    addl 4(%esp), %eax
    addl 10(%esp), %eax
    ret
```

The stack diagram indicates the offsets and sizes of the parameters relative to the value of the stack pointer when the function is called.



In both the examples given above, the stack pointer was adjusted to point to the position it held before the parameters were pushed onto the stack. It is important to ensure that the stack pointer is pointing to a valid return address when a return is executed. Failure to do so will result in either an access violation or a jump to a location in memory where there may not be valid code.

Pass by stack has speed penalty in access to the parameters. The parameters mut be saved on the stack and later accessed by the subroutine. This time penalty aside, access by stack, provides a consistent, flexible mechanism for accessing subroutine parameters.

7.1.3 Pass by Value and Pass by Reference

In the preceeding examples all parameters passed have been passed by value That is the value of the parameter is either loaded into a register (for pass by register) or pushed onto the stack (for pass by register). Parameters may also be passed by reference, that is the address of an item may be passed to a function, and operations may be conducted on the item *insitu* in memory.

The 'C' programming language only provides passing by value. Programmers in 'C' must pass pointers to objects they wish to modify using a subroutine. Pascal provides both pass by value and pass by reference. The following is an example Pascal code fragment:

```
procedure addtwo(var result: integer; p1, p2: integer);
begin
     result := p1 + p2;
end;

{ ... }

      addtwo(res, 2, 4);
```

Translated into assembly language:

```
addtwo:
      movl 4(%esp), %eax              /* get p2 */
      addl 8(%esp), %eax              /* add p1 */
      movl 12(%esp), %edx             /* get the address of result */
      movl %eax, (%edx)               /* store the result */
      ret

      /* ... */

      pushl $result
      pushl $2
      pushl $4
      call addtwo
      add $12, %esp
```

For small data items passing by value has the advantage of providing a copy of the value to the subroutine which it may alter without destroying the value used by the calling routine. If the data item is sufficeintly large, then the convenience gained is offset by the overhead of copying the data item.

7.1.4 Returning Results

The results of a function may be returned by using either a register or by a reference to memory. Returning results by reference is equivalent to passing an additional pass by reference parameter to a function, and using that parameter for the return value.

7.1.5 Local Variables and Stack Frames

A local variable is a variable that is not visible to the caller of a subroutine but is visible to the subroutine. Local variables serve the dual purposes of reducing the amount of global storage space required for a program and providing a private storage area that a subroutine can use. Local variables are created when they are required and persist until the function exits. This ensures that the variable only consumes space when the variable is in use. Recursive routines often require a quantity of storage space in which the current state is stored. Local variables are created with each instance of a subroutine, and provide a natural location in which to store intermediate results.

Local variables are created in assembly language by reserving space on the stack after the parameters. A 'C' program fragment that generates a Fibonacci sequence as an example of a recursive program with local variables is shown below.

```
void fib(int a, int b)
{
    int c;

    printf("%d ", a);
    c = a + b;
    if (c > 50)
        return;
    fib(b, c);
    }

/* ... */

fib(1,1);
```

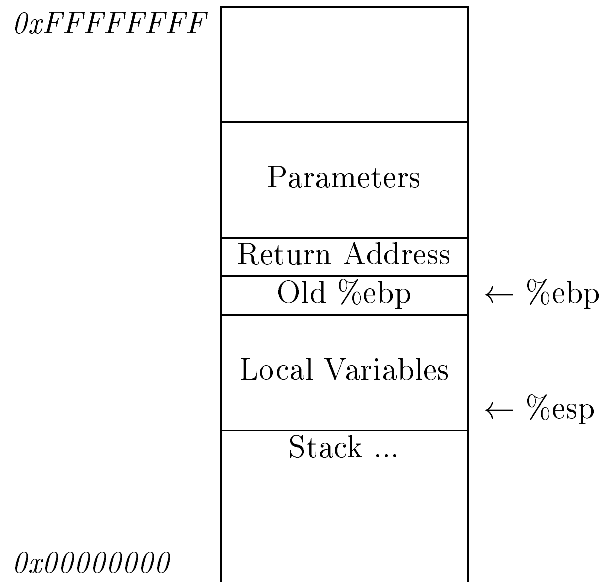
An assembly language fragment using local variables reserved on the stack directly following the parameters of the function:

```
fib:
     subl $4, %esp              /* reserve space for c */
     movl 12(%esp), %eax        /* recover the a parameter */
     call print_num             /* call fictitious print routine */
     movl 8(%esp), %ebx         /* recover the b parameter */
     movl %eax, 0(%esp)         /* store a in c */
     addl %ebx, 0(%esp)         /* add b */
     movl 0(%esp), %ecx         /* move value c into %ecx */
     cmpl $50, 0(%esp)          /* test against 50 */
     jge skip
     pushl %ebx                 /* call fib */
     pushl %ecx
     call fib
     addl $8, %esp              /* fix the stack pointer */
skip:
     addl $4, %esp              /* remove C from stack */
     ret

/* ... */

     pushl $1                   /* fib(1,1) */
     pushl $1
     call fib
     addl $8, %esp
```

A Stack frame is a data structure on the system stack, and which provides a consistent method for representing subroutines. It allows the easy creation of local variables, and permits the use of the stack instructions push and pop.

A stack frame may be created using either the **enter** instruction or by pushing the appropriate values directly onto the stack. Stack frames are destroyed by the **leave** instruction.

The simplest form of the 386/486 stack frame is:



A stack frame uses the base pointer to keep track of the division between a functions parameters and the functions local variables. The use of the base pointer also allows the deallocation of the local variable space and any stack space used by a subroutine on exiting the routine.

Local variables may be accessed using negative offsets from the base pointer and parameters are accessible using positive offsets. Use of push and pop do not affect the base pointer, so the offsets are not affected by normal activity on the stack.

The code that forms a simple stack frame with *space* bytes of local variables is:

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

This is equivalent to the command `enter $space, $0`. The `leave` instruction may be emulated by the code:

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

Leave, restores the base pointer to its previous values.

The example Fibonacci program rewritten to use a simple stack frame:

```
fib:
     pushl %ebp                /* create stack frame */
     movl %esp, %ebp
     subl $4, %esp             /* reserve space for c */
     movl 12(%ebp), %eax       /* recover the a parameter */
     call print_num            /* call ficticious print routine */
     movl 8(%ebp), %ebx        /* recover the b parameter */
     movl %eax, -4(%ebp)       /* store a in c */
     addl %ebx, -4(%ebp)       /* add b */
     movl -4(%ebp), %ecx       /* move value c into %ecx */
     cmpl $50, -4(%ebp)        /* test against 50 */
     jge skip
     pushl %ebx                /* call fib */
     pushl %ecx
     call fib
     addl $8, %esp             /* fix the stack pointer */
skip:
     movl %ebp, %esp           /* destroy stack frame */
     popl %ebp
     ret

/* ... */

     pushl $1                  /* fib(1,1) */
     pushl $1
     call fib
     addl $8, %esp
```

Using **enter** and **leave**:

```
fib:
     enter $4, $0             /* reserve space for c */
     movl 12(%ebp), %eax      /* recover the a parameter */
     call print_num           /* call fictitious print routine */
     movl 8(%ebp), %ebx       /* recover the b parameter */
     movl %eax, -4(%ebp)      /* store a in c */
     addl %ebx, -4(%ebp)      /* add b */
     movl -4(%ebp), %ecx      /* move value c into %ecx */
     cmpl $50, -4(%ebp)       /* test against 50 */
     jge skip
     pushl %ebx               /* call fib */
     pushl %ecx
     call fib
     addl $8, %esp            /* fix the stack pointer */
skip:
     leave                    /* destroy stack frame */
     ret

/* ... */

     pushl $1                 /* fib(1,1) */
     pushl $1
     call fib
     addl $8, %esp
```

Chapter 8

Data Structures

The choice of the method of representation of data in a program has a major effect on the performance of the program. Data structures determine the upper bound of the efficiency of operations on data by an algorithm. Because of the importance of the method of storage of data, this chapter will be devoted to discussing the implementation of some data structures in assembly language.

8.1 Vectors

A vector is a one dimensional array. The assembly language representation of an array consists of a set of equal size objects consecutive in memory. Elements of this set are accessed by multiplying the index of the required element by the size of the element and adding this to the base address of the array. The general representation is shown in figure 8.1.

The vector was introduced in the section on indexed addressing (6.1.2). Code was introduced into that section which used the inbuilt index granularities of 1, 2, 4, and 8 bytes. An example of a generalized indexing scheme which can be used for other element sizes follows.

$$vector[a..b]\ of\ elements$$

$\downarrow a$ $\qquad\qquad\qquad\qquad$ $b \downarrow$

$\uparrow base$

$$addrs = ((index - a) * sizeof(element)) + base$$

Figure 8.1: General Representation of a Vector

```
/* calculate offset from base */
movl $index, %ebx
subl $first, %ebx
movl $size, %eax
/* note that this multiply destroys the contents of %edx */
/* and leaves the result in %eax */
mull %ebx
/* add base to offset %eax points to beginning of item */
addl $base, %eax
/* access first word of element */
movl 0(%eax), %ecx
```

8.2 Arrays

Vectors are a restricted form of the general concept of an array. An array may have more than one dimension, hence, it may be indexed by more than one parameter.

The memory of a computer may be viewed as a one dimensional array of storage locations. Multi-dimensional arrays may be considered as an array of an array of one less dimension. By applying this view recursively until a representable vector has been reached allows an array of any number of dimensions to be constructed.

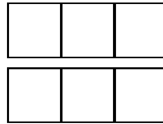
Two dimensional arrays will be used as an example of constructing multi-dimensional arrays. The concepts used in constructing and describing two dimensional arrays may be extended by induction to other multi-dimensional arrays.

There are two ways of linearizing a multidimensional array. The first is to store the first row of the array in memory followed by each subsequent row. This is known as *row-major* form. The second method stores the columns in order, and is known as *column-major* form. (See figure 8.2 for a pictorial form).

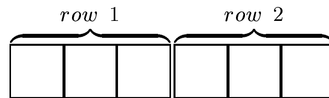
The following section of code provides access to a row-major form 2 dimensional array of arbitrary sized items represented by the Pascal like declaration:

$$arr : array[a..b, c..d] \ of \ element$$

Two Dimensional Array:



Row Major Form:



Column Major Form:

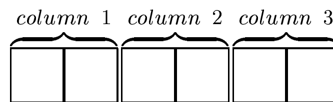


Figure 8.2: Major Forms

```
/* calculate size of a row */
movl $b, %ebx
subl $a, %ebx
movl $size, %eax
/* note that this multiply destroys the contents of %edx */
/* and leaves the result in %eax */
mull %ebx
/* work out the relative row index */
movl $rowidx, %ebx
subl $a, %ebx
/* calculate the row offset */
/* note that this multiply destroys the contents of %edx */
mull %ebx
/* store result in %ecx */
movl %eax, %ecx
/* calculate column offset */
movl $colidx, %ebx
subl $c, %ebx
movl $size, %eax
/* note that this multiply destroys the contents of %edx */
mull %ebx
/* add in stored result and base to get pointer to start of */
/* element [rowidx, colidx] */
addl %ecx, %eax
addl $arr, %eax
```

8.3 Records

A record is a synonym for structure in the context of computer languages. Structures are manipulated by adding an offset to the base address of the structure to yield the address of the element of the structure to be altered. Examples may be found in Chapter 6.

8.4 Dope Vectors

A dope vector is a one dimensional array containing the starting addresses of other objects. Multi-Dimensional arrays can be constructed using dope vectors which involves the storing the starting addresses of an array of lower dimension in the dope vector (see figure 8.3).

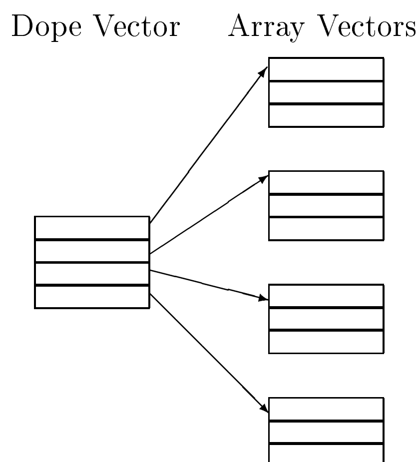


Figure 8.3: A dope vector

The sample code manipulates a four by four array of long words stored in row-major form using a dope vector implementation:

```
/* declarations for array in row major form */
dpv:  .long r0, r1, r2, r3
r0:  .fill 10, 4, 0
r1:  .fill 10, 4, 0
r2:  .fill 10, 4, 0
r3:  .fill 10, 4, 0

/* ...  */

/* retrieve address of row */
movl $rowidx, %ebx
movl dpv(,%ebx,4), %edx
/* retrieve value at column */
movl $colidx, %ebx
movl (%edx, %ebx, 4), %eax
```

8.5 Trees and Graphs

Tree and graph structures are built in assembly language in a manner similar to that used in the 'C' programming language. Essentially a node consists of a structure containing some data and a number of pointers to other nodes. By connecting the nodes together a tree or a graph can be built.

Chapter 9

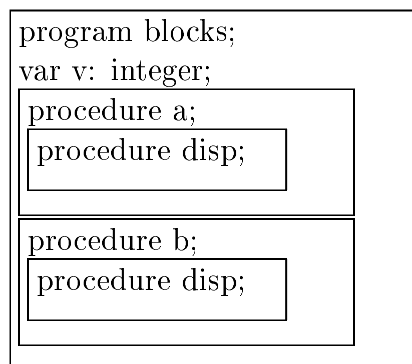
Block Structured Languages

Block structured languages allow nesting and scoping of subroutines and variables. Pascal supports these features, unlike the ‘C’ programming language. The following simple Pascal program illustrates the concept of block structuring.

```
program blocks(input, output);

procedure a;
var
   v: integer;

     procedure disp;
     begin
          writeln('a', v);
     end;
begin
     v := 1;
     disp;
     v := 2;
     disp;
end;

procedure b;
var
   v: integer;

     procedure disp;
     begin
          writeln('b', v);
     end;
begin
```

```
     v := 1;
     disp;
     v := 2;
     disp;
end;

begin
     a;
     b;
end.
```

The program's output is 'a1 a2 b1 b2'. The structure of a block structured program may be drawn:



In Pascal, the scope of a variable is the region in which it is accessible by name to a subroutine. Variables declared in blocks of which the current subroutine is a strict subset are within the scope of the current function.

The scope of a subroutine in Pascal is the region in which a function or procedure may be called by name. Procedures and functions in the current block and blocks which are one level above the current block and contained by the current block are accessible.

Block structured languages are supported in assembler by providing backward links in the stack frame to earlier stack frames. The **enter** instructions second parameter, **level**, determines the number of stack frame pointers that are inserted into the current stack frame to the previous stack frame. Figure 9.1 shows an example of the appearance of the stack with back pointers. The example shows the invocation of *disp* by procedure *a*.

By following back the chain of back pointers it is possible to access any variable in the scope of the current function.

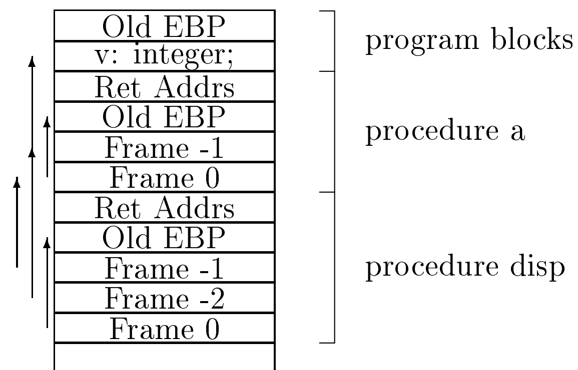


Figure 9.1: Stack Frames in a Block Structured Language

Appendix A

AT&T Syntax

A.1 Register Set

The 80386/80486 provides a set of general registers, segment registers, debug registers and control registers. The name and size is recorded for each register directly accessible using an **AT&T** type assembler. Note that all register names are preceeded by a percent sign.

General Purpose Registers:

Register Name	Width (bits)	Type
%eax	32	General Purpose Register
%ax	16	Least 16 bits of %eax
%ah	8	Greatest 8 bits of %ax
%al	8	Least 8 bits of %ax
%ebx	32	General Purpose Register
%bx	16	Least 16 bits of %ebx
%bh	8	Greatest 8 bits of %bx
%bl	8	Least 8 bits of %bx
%ecx	32	General Purpose Register
%cx	16	Least 16 bits of %ecx
%ch	8	Greatest 8 bits of %cx
%cl	8	Least 8 bits of %cx
%edx	32	General Purpose Register
%dx	16	Least 16 bits of %edx
%dh	8	Greatest 8 bits of %dx
%dl	8	Least 8 bits of %dx
%ebp	32	Base Pointer
%bp	16	Least 16 bits of %ebp
%esi	32	Source Index
%si	16	Least 16 bits of %esi
%edi	32	Destination Index
%di	16	Least 16 bits of %edi
%esp	32	Stack Pointer
%sp	16	Least 16 bits of %esp

Segment Registers:

Register Name	Width (bits)	Type
%cs	16	Code Segment
%ds	16	Data Segment
%es	16	Extra Segment
%ss	16	Stack Segment
%fs	16	Segment
%gs	16	Segment

Debug Registers:

Register Name	Width (bits)	Type
%dr0	32	Breakpoint 0 Linear Address - Debug Register
%dr1	32	Breakpoint 1 Linear Address - Debug Register
%dr2	32	Breakpoint 2 Linear Address - Debug Register
%dr3	32	Breakpoint 3 Linear Address - Debug Register
%dr6	32	Debug Control Register
%dr7	32	Debug Control Register

Test Registers:

Register Name	Width (bits)	Type
%tr3	32	Test Register
%tr4	32	Test Register
%tr5	32	Test Register
%tr6	32	Test Register
%tr7	32	Test Register

Floating Point Stack Registers:

Register Name	Type
%st	Top of NPU stack
%st(0)	Top of NPU stack
%st(1)	NPU stack register
%st(2)	NPU stack register
%st(3)	NPU stack register
%st(4)	NPU stack register
%st(5)	NPU stack register
%st(6)	NPU stack register
%st(7)	NPU stack register

A.2 Flags

The 80386 provides a flag register known as EFLAGS. This register contains bits which are set by the processor after arithmetic operations or which reflect the current state of the processor.

31 ... 0

0	0	0	0	0	0	0	0	0	0	0	0	0	AC	VM	RF	0	NF	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

The flags are:

AC Alignment Check[1].

VM Virtual 8086 Mode

RF Resume Flag

NT Nested Task Flag

IOPL IO Privelege Level (2 bits)

OF Overflow Flag

DF Direction Flag

IF Interrupt Flag

TF Trap Flag

SF Sign Flag

ZF Zero Flag

AF Auxiliary Carry Flag

PF Parity Flag

CF Carry Flag

A.3 Assembler Syntax

This section is specific to the Free Software Foundation's GNU AS assembler. Many of the other **AT&T** type assemblers use a similar set of operations, typically a subset of these.

A.3.1 General Layout

The assembler input is free form, requiring only that statements be separated by either a newline character or a semicolon. Character constants are not terminated by a newline or semicolon (';') character. A statement may be continued over more than one line by placing a backslash ('\') before the newline character.

Symbols may be made up of alphabetics, digits, '_', '$' and '.'. Symbols are case significant. The special symbol '.' refers to the current address that is being assembled to.

Strings are delimited by double-quote character ('"').

Numbers follow the conventions of C:

[1] Not available on the 80386

Decimal Any number not beginning with a zero eg. 10.

Hexadecimal A number beginning with '0x' eg. 0xa.

Octal A number beginning with zero eg. 012.

Special characters follow the conventions of C:

b Backspace

f Formfeed

n Newline

r Carriage Return

t Tab

***ooo* where *o* is an octal digit** An octal character code.

\\\\ The '\\' character.

\\" The '"' characther

Labels are a symbol followed immediately by a colon (':').

A.3.2 Operands

Immediate operands are numbers which do not represent memory locations. These are prefaced in **AT&T** type assemblers by a dollar sign ('$').

Absolute references are prefaced by an asterix ('*') to differentiate them from relative references.

The size of operands are determined explicitly by the instruction, not by reference to the size of the object refered to. Opcode suffixes are added to indicate the size of the operation.

b Byte (8-bit)

w Word (16-bit)

l Long (32-bit)

A.3.3 Comments

There are two forms of comments:

- C type: Comments may be multiline and are delimited by '/*' and '*/'.
- Line Comment type: All characters from '#', the line comment character, to the next newline are ignored.

A.3.4 Expressions

The following operators are available:

– Two's complement negation.

~ One's complement negation.

* Multiplication.

/ Division.

% Modulo.

< **or** << Left shift.

> **or** >> Right shift.

| Bitwise Or.

& Bitwise And.

^ Bitwise Xor

! Bitwise Or Not.

+ Add.

– Subtract.

A.3.5 Assembler Directives

.abort Stop assembly immediately

.align *boundary*, *pad* Adjust the location counter to the next boundary exactly divisible by $2^{boundary}$. If *pad* is present then this value of the bytes used in filling to the next boundary.

.ascii *strings* Reserves space for and stores *strings*.

.asciz *strings* Reserves space for and stores *strings* with an additional zero byte at the end of each string.

.byte *expressions* Comma separated expressions are stored into the next byte.

.comm *symbol*, *length* Declares a named common area of at least *length* bytes size.

.data *subsegment* Assembles following statements at the end of data subsegment *subsegment*. The default subsegment is 0.

.double *flonums* Comma separated floating point numbers are stored into the 64-bit floating point form.

.file *string* The *string* becomes the name of the new logical file.

.fill *repeat*, *size*, *value* Creates a block of *repeat* objects of *size* bytes containing *value*.

.float *flonums* Comma separated floating point numbers are stored into the 32-bit floating point form.

.globl *symbol* Makes *symbol* visible to the linker.

.int *expressions* Comma separated expressions are stored into the next 32 bits.

.lcomm *symbol*, *length* Declares a local common area of at least *length* bytes size. At run time the bytes of this area start off zeroed. This area is not visible to the linker.

.line *number* Assigns a logical line number to the statements following.

.long *expressions* Comma separated expressions are stored into the next 32 bits.

.octa *bignums* Comma separated big numbers are stored into the next 16 bytes.

.org *lc*, *fill* Advances the segments location counter to *lc* using *fill* as padding.

.quad *bignums* Comma separated big numbers are stored into the next 8 bytes.

.set *symbol*, *expression* Sets the value of *symbol* to *expression*.

.short *expressions* Comma separated expressions are stored into the next 16 bits.

.single *flonums* Comma separated floating point numbers are stored into the 32-bit floating point form.

.space *size*, *fill* Fills an area of *size* bytes with the value *fill*. If *fill* is omitted then the area is filled with zeros.

.text *subsegment* Assembles following statements at the end of text subsegment *subsegment*. The default subsegment is 0.

.word *expressions* Comma separated expressions are stored into the next 16 bits.

A.3.6 Memory References

Direct memory references may be made by using either a symbol, a numeric constant or an expression.

The **AT&T** syntax for indirect memory references is:

$$segment : disp(base, index, scale)$$

This may be explicitly written as:

$$\left\{\begin{array}{c}\%cs\\ \%ds\\ \%es\\ \%ss\\ \%fs\\ \%gs\end{array}\right\} : \left\{\begin{array}{c}No\ Displacement\\ 8-Bit\ Displacement\\ 32-Bit\ Displacement\end{array}\right\} (\left\{\begin{array}{c}\%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\}, \left\{\begin{array}{c}\%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\}, \left\{\begin{array}{c}1\\ 2\\ 4\\ 8\end{array}\right\})$$

The effective address of a memory location is calculated:

$$\left\{\begin{array}{c}\%cs\\ \%ds\\ \%es\\ \%ss\\ \%fs\\ \%gs\end{array}\right\} + \left\{\begin{array}{c}No\ Displacement\\ 8-Bit\ Displacement\\ 32-Bit\ Displacement\end{array}\right\} + \left\{\begin{array}{c}-\\ \%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\} + \left\{\begin{array}{c}-\\ \%eax\\ \%ebx\\ \%ecx\\ \%edx\\ \%esi\\ \%edi\\ \%ebp\\ \%esp\end{array}\right\} * \left\{\begin{array}{c}-\\ 1\\ 2\\ 4\\ 8\end{array}\right\}$$

If the segment modifier is not included then the instruction uses the default segment. If the displacement is not included then a displacement of zero is assumed. The valid forms of the index section are:

$$(base, index, scale)$$
$$(base, index)$$
$$(base)$$
$$(, index, scale)$$
$$(, index)$$

Appendix B

Instruction Set

B.1 Layout

The instructions to be used in these lab classes are provided in this chapter. The description of each instruction is divided into 6 components. The ADD instruction is presented, with commentary, as an example.

Each instruction is documented for an **AT&T** style of assembler.

B.1.1 Title lines

ADD Add

This line contains the mnemonic for the instruction on the left hand side and a description of the function of the instruction at the right.

B.1.2 Type & Compatibility

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

This set of boxes classifies the type of the instruction and the processors with which the instruction is compatible.

The type compatibility boxes are:

Flow Flow of Control - Instructions which may cause execution to change to a location other than the next instruction.

Int Integer - Instructions that operate on integer values.

Float Floating Point - Instructions which operate on floating point numbers.

Multi Multi-Segment - Instructions which operate on more than one segment.

The operating system box (OpSys) is checked if the instruction is not used by applications programs.

The processor compatability boxes are:

386 i80386 - This box is checked if this instruction is available on the 386 processor.

387 i80387 - This box is checked if this instruction is available on the 387 numeric processing unit (NPU). The floating point coprocessor for the 386.

486 i80486 - This box is checked if this instruction is available on the 486 processor.

B.1.3 Formats

Formats:
AT&T
ADD imm, mem
ADD reg, mem
ADD imm, reg
ADD mem, reg
ADD reg, reg

The **AT&T** column is used for **AT&T** type assemblers. Listed in the column is the mnemonic for the instruction and the valid types of operands for that instruction.

The operand types are:

imm An immediate value.

m14/28byte The address of a memory location extending over 14 or 28 bytes

m16int The address of a memory location that represents a 16 bit integer.

m16real The address of a memory location that represents a 16 bit real.

m2byte The address of a memory location extending over 2 bytes.

m32int The address of a memory location that represents a 32 bit integer.

m32real The address of a memory location that represents a 32 bit real.

m64int The address of a memory location that represents a 64 bit integer.

m64real The address of a memory location that represents a 64 bit real.

m80dec The address of a memory location that represents a 80 bit decimal.

m80real The address of a memory location that represents a 80 bit real.

m94/108byte The address of a memory location extending over 94 or 108 bytes.

mem The address of a memory location.

ofs A signed offset from the current memory location.

ptr A pointer. The address of a value which consists of a selector and a the address of a memory location.

reg Any register.

reg16 A 16 bit register.

reg32 A 32 bit register.

sreg A segment register.

ST The top of the NPU stack.

ST(i) The i*th* element of the NPU stack.

B.1.4 Psuedo Instructions

Pseudo:
AT&T
ADD src1, dst

The instruction is followed by pseudo operands. The pseudo operands are used in the description of the instruction that follows this section. The pseudo instruction and operands is used to group different versions of the same instruction which have the same form.

B.1.5 Description

Description
This instruction adds two integers - *src1* and *dst* - leaving the result in *dst*. The flags are set accordingly. If *src1* is an immediate byte value then it is sign extended to the size of *dst* before the addition.

This section contains a short description of the function of the instruction and any warnings relavent to its use.

B.1.6 Flags

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT
M				

This section lists the flags consulted or altered by the instruction.

The codes are:

Blank Flag is unaffected by instruction.

T Flag is tested by instruction.

M Flag is modified by instruction depending on the operands.

1 Flag is set by instruction.

0 Flag is cleared by instruction.

U The instruction's effect on the state of the flag is undefined.

B.2 Instructions

AAA ASCII Adjust after Addition

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:
AT&T
AAA
Pseudo:
AT&T
AAA
Description

This instruction is used after an ADD instruction which takes two byte size unpacked BCD numbers as its operands and leaves the byte size result in the *%al* register. The AAA instruction adjusts *%al* to contain the correct unpacked BCD result.

If CF or the lower nibble of *%al* is greater than 9 then *%al* is incremented by 6, *%ah* is incremented by 1 and CF and AF are set. Otherwise, CF and AF are cleared. In both cases the top 4 bits of *%al* are cleared.

Flags:

OF	SF	ZF	AF	PF
U	U	U	TM	U
CF	TF	IF	DF	NT
M				

AAD ASCII Adjust AX before Division

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:
AT&T
AAD
Pseudo:
AT&T
AAD
Description

This instruction is used to generate a binary number from a two byte unpacked BCD number. The result of the operation is to set *%al* to $\%al + (10 \times \%ah)$ and clear *%ah*.

Flags:

OF	SF	ZF	AF	PF
U	M	M	U	M
CF	TF	IF	DF	NT
U				

AAM ASCII Adjust AX after Multiply

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

AAM

Pseudo:

AT&T

AAM

Description

This instruction is used after a MUL instruction operating on two byte size unpacked BCD numbers that leaves the byte size result in the *%ax* register. The AAM instruction sets *%al* to *%al mod* 10 and *%ah* to *%al*/10.

Flags:

OF	SF	ZF	AF	PF
U	M	M	U	M
CF	**TF**	**IF**	**DF**	**NT**
U				

AAS ASCII Adjust AL after Subtraction

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

AAS

Pseudo:

AT&T

AAS

Description

This instruction is used after a SUB instruction which takes two byte size unpacked BCD numbers as its operands and leaves the byte size result in the *%al* register. The AAS instruction adjusts *%al* to contain the correct unpacked BCD result.

If CF then *%ah* is decremented and CF and AF are set. Otherwise, CF and AF are cleared. In both cases the top 4 bits of *%al* are cleared.

Flags:

OF	SF	ZF	AF	PF
U	U	U	TM	U
CF	**TF**	**IF**	**DF**	**NT**
M				

ADC Add With Carry

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

ADC imm, mem
ADC reg, mem
ADC imm, reg
ADC mem, reg
ADC reg, reg

Pseudo:

AT&T

ADC src1, dst

Description

This instruction adds two integers - *src1* and *dst* - and CF leaving the result in *dst*. The flags are set accordingly. If *src1* is an immediate byte value then it is sign extended to the size of *dst* before the addition.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT
TM				

ADD Add

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

ADD imm, mem
ADD reg, mem
ADD imm, reg
ADD mem, reg
ADD reg, reg

Pseudo:

AT&T

ADD src1, dst

Description

This instruction adds two integers - *src1* and *dst* - leaving the result in *dst*. The flags are set accordingly. If *src1* is an immediate byte value then it is sign extended to the size of *dst* before the addition.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT
M				

AND Logical AND

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

AND imm, mem
AND reg, mem
AND imm, reg
AND mem, reg
AND reg, reg

Pseudo:

AT&T

AND src1, dst

Description

This instruction performs a logical AND on each bit of two integers - *src1* and *dst* - leaving the result in *dst*. CF and OF are cleared and PF, SF, and ZF are set according to the result.

Flags:

OF	SF	ZF	AF	PF
0	M	M	U	M
CF	TF	IF	DF	NT
0				

ARPL Adjust RPL Field Selector

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

ARPL mem, reg
ARPL reg, mem

Pseudo:

AT&T

ARPL src1, dst

Description

This instruction is used in operating system software to prevent code requesting greater privelege than it is allowed. *dst* contains the value of a selector, *src1* is word size register. If the lower two bits of *dst* is less than *src1* then ZF is set and the lower two bits of *dst* is made equal to the lower two bits of *src1*. Otherwise, ZF is cleared.

Flags:

OF	SF	ZF	AF	PF
		M		
CF	TF	IF	DF	NT

BOUND Check Array Index Against Bounds

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BOUND mem, reg

Pseudo:

AT&T

BOUND src1, src2

Description

This instruction is used to check a signed array index is between an upper and lower bound specified in a memory block. The array index *src2* is checked against the bounds in the memory block pointed to by *src1*. The format of the memory block is 2 consecutive 16 bit signed integers. The lower bound occurs first followed by the upper bound. If *src2* is not between the lower bound and the upper bound plus the number of bytes occupied for the operand size then interrupt 5 is generated.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

BSF Bit Scan Forward

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BSF mem, reg

BSF reg, reg

Pseudo:

AT&T

BSF src1, dst

Description

This instruction scans the bits of *src1* from the LSB upwards. If all the bits are 0 then ZF is set. Otherwise, ZF is cleared and *dst* is set to the index of the least set bit.

Flags:

OF	SF	ZF	AF	PF
U	U	M	U	U
CF	TF	IF	DF	NT
U				

BSR Bit Scan Reverse

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BSR mem, reg

BSF reg, reg

Pseudo:

AT&T

BSR src1, dst

Description

This instruction scans the bits of *src1* from the MSB downwards. If all the bits are 0 then ZF is set. Otherwise, ZF is cleared and *dst* is set to the index of the greatest set bit.

Flags:

OF	SF	ZF	AF	PF
U	U	M	U	U
CF	TF	IF	DF	NT
U				

BSWAP Byte Swap

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
		×

Formats:

AT&T

BSWAP reg

Pseudo:

AT&T

BSWAP dst

Description

This instruction swaps the top and middle nibles of *dst*. The result is the conversion of a 32 bit big endian number to a little endian number or vice versa. **WARNING**: The result of this operation is undefined on a 16 bit operand.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

BT Bit Test

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BT imm, reg
BT imm, mem
BT reg, reg
BT reg, mem

Pseudo:

AT&T

BT src1, src2

Description

This instruction stores bit *src1* of *src2* in CF.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U
CF	TF	IF	DF	NT
M				

BTC Bit Test and Complement

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BTC imm, reg
BTC imm, mem
BTC reg, reg
BTC reg, mem

Pseudo:

AT&T

BTC src1, dst

Description

This instruction stores bit *src1* of *dst* in CF and then complements the bit in *dst*.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U
CF	TF	IF	DF	NT
M				

BTR Bit Test and Reset

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BTR imm, reg
BTR imm, mem
BTR reg, reg
BTR reg, mem

Pseudo:

AT&T

BTR src1, dst

Description

This instruction stores bit *src1* of *dst* in CF and then sets the bit in *dst* to 0.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U
CF	TF	IF	DF	NT
M				

BTS Bit Test and Set

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

BTS imm, reg
BTS imm, mem
BTS reg, reg
BTS reg, mem

Pseudo:

AT&T

BTS src1, dst

Description

This instruction stores bit *src1* of *dst* in CF and then sets the bit in *dst* to 1.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U
CF	TF	IF	DF	NT
M				

CALL Call Procedure or Function

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

CALL reg
CALL mem
CALL ofs
CALL ptr

Pseudo:

AT&T

CALL dst

Description

This instruction pushes the current location onto the stack and then jumps to *dst*. In the case of near destinations (reg, mem, ofs) only the IP or EIP is pushed onto the stack. Far calls (ptr) push CS before IP or EIP.

Far calls may be used to access routines at a higher protection level through call gates or a task gate.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

CBW Convert Byte to Word
CBTW Convert Byte to Word (AT&T Only)
CWDE Convert Word to Doubleword
CBTL Convert Byte to Long (AT&T Only)

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

CBW
CBTW
CWDE
CBTL

Pseudo:

AT&T

CBW
CBTW
CWDE
CBTL

Description

CBW places the sign extended form of the *%al* register into *%ax*. CWDE places the sign extended form of the *%ax* register into *%eax*. **AT&T** compilers provide to additional synonyms: CBTW and CBTL.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

CLC Clear Carry Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:
AT&T
CLC
Pseudo:
AT&T
CLC
Description
This instruction sets CF to 0.
Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT
0				

CLD Clear Direction Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:
AT&T
CLD
Pseudo:
AT&T
CLD
Description
This instruction sets DF to 0.
Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT
			0	

CLI Clear Interrupt Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

CLI

Pseudo:

AT&T

CLI

Description

If the current privelege is equal to or more priveleged than IOPL then this instruction sets IF to 0.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**
		0		

CLTS Clear Task-Switched Flag in CR0

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

CLTS

Pseudo:

AT&T

CLTS

Description

This instruction sets TS in *%cr0* to 0.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

CMC Complement Carry Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

CMC

Pseudo:

AT&T

CMC

Description

This instruction complements CF.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**
M				

CMP Compare Two Operands

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

CMP imm, mem
CMP reg, mem
CMP imm, reg
CMP mem, reg
CMP reg, reg

Pseudo:

AT&T

CMP src1, src2

Description

This instruction performs the function $src2 - src1$ setting the flags in accordance with the result. The result of the subtraction is NOT stored.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	**TF**	**IF**	**DF**	**NT**
M				

CMPS Compare String Operands
CMPSB Compare String Operands
CMPSW Compare String Operands
CMPSD Compare String Operands

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

CMPS mem, mem
CMPSB
CMPSW
CMPSD

Pseudo:

AT&T

CMPS src1, src2
CMPSB
CMPSW
CMPSD

Description

This instruction compares two elements of a string pointed to by the source and destination registers.

The no operand form of the instruction subtracts *%es:(%edi)* from *%ds:(%esi)* and sets the flags appropriately. The result of the subtraction is NOT stored. The registers *%esi and %edi* are incremented by the number of bytes of the operand size if DF is 0. If DF is 1 then the source and destination registers are decremented by the number of bytes of the operand size.

The two operand form of the instruction operates similarly. The length of the operands is determined by the size of *src2* and the segment refered to by the source index is determined by the segment prefix of *src2*. If no segment prefix is given then *%ds* is assumed.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT
M			T	

CMPXCHG Compare and Exchange

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
		×

Formats:

AT&T

CMPXCHG reg, reg

CMPXCHG reg, mem

Pseudo:

AT&T

CMPXCHG src1, dst

Description

This instruction compares the accumulator - *%al, %ax, %eax* - with *dst*. The flags are set in accordance with the result of the comparison. If ZF is set then *src1* is copied into *dst*. Otherwise, *dst* is copied into the accumulator.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M

CF	TF	IF	DF	NT
M				

CWD Convert Word to Doubleword
CWTD Convert Word to Doubleword (AT&T Only)
CDQ Convert Doubleword to Quadword
CLTD Convert Long to Quadword (AT&T Only)

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

CWD

CDQ

Pseudo:

AT&T

CWD

CWTD

CDQ

CLTD

Description

CWD places the sign extended form of the *%ax* register in the register pair *%dx:%ax*. CDQ places the sign extended form of the *%eax* register in the register pair *%edx:%eax*. **AT&T** compilers provide to additional synonyms: CWTD and CLTD.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

DAA Decimal Adjust AL after Addition

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

DAA

Pseudo:

AT&T

DAA

Description

This instruction is used after an ADD instruction which takes two byte size packed BCD numbers as its operands and leaves the byte size result in the *%al* register. The DAA instruction adjusts *%al* to contain the correct packed BCD result.

Flags:

OF	SF	ZF	AF	PF
U	M	M	TM	M
CF	TF	IF	DF	NT
TM				

DAS Decimal Adjust AL after Subtraction

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

DAS

Pseudo:

AT&T

DAS

Description

This instruction is used after a SUB instruction which takes two byte size packed BCD numbers as its operands and leaves the byte size result in the *%al* register. The DAS instruction adjusts *%al* to contain the correct packed BCD result.

Flags:

OF	SF	ZF	AF	PF
U	M	M	TM	M
CF	TF	IF	DF	NT
TM				

DEC Decrement by 1

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

DEC mem

DEC reg

Pseudo:

AT&T

DEC dst

Description

This instruction subtracts 1 from *dst*. Note that CF is not affected by this instruction.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M

CF	TF	IF	DF	NT

DIV Unsigned Divide

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

DIV mem, %al
DIV mem, %ax
DIV mem, %eax
DIV reg, %al
DIV reg, %ax
DIV reg, %eax
DIV mem
DIV reg

Pseudo:

AT&T

DIV src1, dst
DIV src1

Description

This instruction performs unsigned division on the extended register pair of *dst* by dividing by *src1* leaving the result in the extended register pair *dst*. The extended register pairs of the accumulators are: *%edx:%eax* for *%eax*; *%dx:%ax* for *%ax*; and *%ax* for *%al*.

The single operand form of this instruction divides the extended register pair of the accumulator - determined by the size of the size modifier of the opcode - by *src1*.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U

CF	TF	IF	DF	NT
U				

ENTER Make a Stack Frame

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

ENTER imm, imm

Pseudo:

AT&T

ENTER rsv, lvl

Description

This instruction creates a stack frame suitable for many high level languages. The two operands are: *rsv* the number of bytes to reserve for local variables, and *lvl* the lexical nesting level of the procedure. If *lvl* is zero then the operations performed are:

- push the current base pointer
- set the base pointer to equal the frame pointer (the value of the stack pointer after the base pointer was pushed)
- subtract *rsv* from the current stack pointer.

If *lvl* is not zero then the operations performed are:

- push the current base pointer
- push *lvl* modulo 32 minus one links to previous stack frames
- push the frame pointer (the value of the stack pointer after the base pointer was pushed)
- set the base pointer to equal the frame pointer
- subtract *rsv* from the current stack pointer.

Note: The *frame pointer* is a name for a value of the stack pointer. The *frame pointer* is NOT a register.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

F2XM1 Compute $2^X - 1$

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

F2XM1

Pseudo:

AT&T

F2XM1

Description

This instruction computes $2^{ST} - 1$ where $-1 < ST < 1$ and places the result in ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FABS Absolute Value

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FABS

Pseudo:

AT&T

FABS

Description

This instruction computes $|ST|$ and places the result in ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FADD Add
FADDP Add
FIADD Add

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FADD m32real
FADD m64real
FADD ST(i), ST
FADD ST, ST(i)
FADDP ST, ST(i)
FADD
FIADD m32int
FIADD m16int

Pseudo:

AT&T

FADD src1
FADD src1, dst
FADDP src1, dst
FADD
FIADD src1

Description

These instructions add *dst* and *src1*. The result is placed in *dst*. The no operand form and FADDP are equivalent and both pop ST off the FPU stack after completing the addition. The one operand form adds ST to *src1*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FBLD Load Binary Coded Decimal

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FBLD m80dec

Pseudo:

AT&T

FBLD src1

Description

This instruction takes a BCD number - *src1* - and converts it into an extended-real format number. The result is pushed onto the FPU stack.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

FBSTP Store Binary Coded Decimal and Pop

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FBSTP m80dec

Pseudo:

AT&T

FBSTP dst

Description

This instruction takes ST and converts it to a packed decimal integer. The result is stored at *src1* and the FPU stack is poped. If ST is not an integer then it is rounded in accordance with RC in the control word.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

FCHS Change Sign

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FCHS

Pseudo:

AT&T

FCHS

Description

This instruction computes $-ST$ and places the result in ST.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

FCLEX Clear Exceptions
FNCLEX Clear Exceptions

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FCLEX

FNCLEX

Pseudo:

AT&T

FCLEX

FNCLEX

Description

These instructions clear the exception flags, the exception status flag and the busy flag of the FPU status word. FCLEX checks for unmasked error conditions before acting. FNCLEX acts irrespective of existing error conditions.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

FCOM Compare Real
FCOMP Compare Real
FCOMPP Compare Real

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FCOM m32real
FCOM m64real
FCOM ST(i)
FCOMP m32real
FCOMP m64real
FCOMP ST(i)
FCOMP
FCOMPP

Pseudo:

AT&T

FCOM src1
FCOM
FCOMP src1
FCOMP
FCOMPP

Description

These instructions compare two real numbers: ST and *src1*. The no operand form compares ST and ST(1). The FPU flags C_0, C_2, C_3 are set in accordance with the result (as shown in the table below). FCOMP and FCOMPP pop the FPU stack on completion of the comparison.

	$C_0C_2C_3$
$ST > src1$	0 0 0
$ST < src1$	1 0 0
$ST = src1$	0 0 1
Not Comparable	1 1 1

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FCOS Cosine

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FCOS

Pseudo:

AT&T

FCOS

Description

This instruction computes $\cos(ST)$ where ST is in radians and $-2^{63} < ST < 2^{63}$. The result is placed in ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FDECSTPDecrement Stack-Top Pointer

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FDECSTP

Pseudo:

AT&T

FDECSTP

Description

This instruction decrements the FPU stack pointer. If the FPU stack pointer is pointing to ST(0) then FDECSTP changes the FPU stack pointer to point to ST(7).

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FDIV Divide
FDIVP Divide
FIDIV Divide

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FDIV m32real
FDIV m64real
FDIV ST(i), ST
FDIV ST, ST(i)
FDIVP ST, ST(i)
FDIV
FIDIV m32int
FIDIV m16int

Pseudo:

AT&T

FDIV src1
FDIV src1, dst
FDIVP src1, dst
FDIV
FIDIV src1

Description

These instructions divide *dst* by *src1*. The result is placed in *dst*. The no operand form and FDIVP are equivalent and both pop ST off the FPU stack after completing the division. The one operand form divides ST by *src1*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FDIVR Reverse Divide
FDIVPR Reverse Divide
FIDIVR Reverse Divide

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FDIVR m32real
FDIVR m64real
FDIVR ST(i), ST
FDIVR ST, ST(i)
FDIVRP ST, ST(i)
FDIVR
FIDIVR m32int
FIDIV m16int

Pseudo:

AT&T

FDIVR src1
FDIVR src1, dst
FDIVRP src1, dst
FDIVR
FIDIVR src1

Description

These instructions divide *src1* by *dst*. The result is placed in *dst*. The no operand form and FDIVRP are equivalent and both pop ST off the FPU stack after completing the division. The one operand form divides *src1* by ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FFREE Free Floating-Point Register

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FFREE ST(i)

Pseudo:

AT&T

FFREE dst

Description

This instruction sets the tag of the destination register to empty.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FICOM Compare Integer
FICOMP Compare Integer

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FICOM m16real
FICOM m32real
FICOMP m16int
FICOMP m32int

Pseudo:

AT&T

FICOM src1
FICOMP src1

Description

These instructions compare two integer numbers: ST and *src1*. The FPU flags C_0, C_2, C_3 are set in accordance with the result (as shown in the table below). FICOMP pops the FPU stack on completion of the comparison.

	$C_0C_2C_3$
$ST > src1$	0 0 0
$ST < src1$	1 0 0
$ST = src1$	0 0 1
Not Comparable	1 1 1

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FILD Load Integer

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FILD m16int
FILD m32int
FILD m64int

Pseudo:

AT&T

FILD src1

Description

This instruction converts *src1* to extended-real format and pushes it onto the FPU stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FINCSTP Increment Stack-Top Pointer

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FINCSTP

Pseudo:

AT&T

FINCSTP

Description

This instruction increments the FPU stack pointer. If the FPU stack pointer is pointing to ST(7) then FINCSTP changes the FPU stack pointer to point to ST(0).

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FINIT Initialize Floating-Point Unit
FNINIT Initialize Floating-Point Unit

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FINIT

FNINIT

Pseudo:

AT&T

FINIT

FNINIT

Description

These instructions set the FPU into the state:

- Round to nearest
- Mask all exceptions
- 64-bit precision
- All exception flags clear
- Stack register set to top of stack.
- All stack registers tagged empty.
- Instruction and data error pointers cleared.

The FINIT form checks for unmasked floating point errors before acting. The FNINIT instruction acts without checking for error conditions.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FIST Store Integer
FISTP Store Integer

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FIST m16int
FIST m32int
FISTP m16int
FISTP m32int
FISTP m64int

Pseudo:

AT&T

FIST dst
FISTP dst

Description

These instructions convert ST into a signed integer and store the result in *dst*. The FISTP form pops the FPU stack after storing the result.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FLD Load Real

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FLD m32real
FLD m64real
FLD m80real
FLD ST(i)

Pseudo:

AT&T

FLD src1

Description

This instruction pushes the real number *src1* onto the FPU stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FLD1 Load Constant $+1.0$
FLDL2T Load Constant $\log_2 10$
FLDL2E Load Constant $\log_2 e$
FLDP1 Load Constant π
FLDLG2 Load Constant $\log_{10} 2$
FLDLN2 Load Constant $\log_e 2$
FLDZ Load Constant $+0.0$

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FLDtt

Pseudo:

AT&T

FLDtt

Description

These instructions push a real number constant onto the FPU stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FLDCW Load Control Word

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FLDCW m2byte

Pseudo:

AT&T

FLDCW src1

Description

This instruction sets the FPU control word to the contents of the memory location *src1*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FLDENV Load FPU Environment

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FLDENV m14/28byte

Pseudo:

AT&T

FLDENV src1

Description

This instruction loads the FPU environment with the contents of the memory block at *src1*.

The structure of the memory block consists of an FPU control word, an FPU status word, a tag word, and the data and instruction error pointers. The size of the environment depends on the default operand size and the current processor operating mode. Typically, the data loaded by this instruction has been stored by an FSTENV or FNSTENV instruction.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FMUL Multiply
FMULP Multiply
FIMUL Multiply

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FMUL m32real
FMUL m64real
FMUL ST(i), ST
FMUL ST, ST(i)
FMULP ST, ST(i)
FMUL
FIMUL m32int
FIMUL m16int

Pseudo:

AT&T

FMUL src1
FMUL src1, dst
FMULP src1, dst
FMUL
FIMUL src1

Description

These instructions multiply *dst* and *src1*. The result is placed in *dst*. The no operand form and FMULP are equivalent and both pop ST off the FPU stack after completing the addition. The one operand form multiplies ST by *src1*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FNOP No Operation

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FNOP

Pseudo:

AT&T

FNOP

Description

This instruction performs no operation.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FPATAN Partial Arctangent

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FPATAN

Pseudo:

AT&T

FPATAN

Description

This instruction computes $\arctan(ST(1)/ST)$ and places the computed value (in radians) in ST(1). ST is then popped.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FPREM Partial Remainder

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:
AT&T
FPREM

Pseudo:
AT&T
FPREM

Description
This instruction computes the remainder from the division of ST by ST(1). The result is stored in ST. Note that the sign of ST is not altered by this operation.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FPREM1 Partial Remainder

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:
AT&T
FPREM1

Pseudo:
AT&T
FPREM1

Description
This instruction computes the remainder from the division of ST by ST(1). The result is stored in ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FPTAN Partial Tangent

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FPTAN

Pseudo:

AT&T

FPTAN

Description

This instruction computes $\tan(ST)$ where $-2^{63} < ST < 2^{63}$ and places the computed value in ST. 1.0 is then pushed onto the stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FRNDINTRound to Integer

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FRNDINT

Pseudo:

AT&T

FRNDINT

Description

This instruction rounds ST to an integer. The method of rounding is determined by RC in the FPU control word.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FRSTOR Restore FPU State

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FRSTOR m94/108byte

Pseudo:

AT&T

FRSTOR src1

Description

This instruction restores the FPU state from a memory block located at *src1*.

The structure of the memory block consists of an FPU control word, an FPU status word, a tag word, the data and instruction error pointers, and the stack registers ST to ST(7). The size of the environment depends on the default operand size and the current processor operating mode. Typically, the data restored by this instruction has been stored by an FSAVE or FNSAVE instruction.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSAVE Store FPU State
FNSAVE Store FPU State

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSAVE m94/108byte

FNSAVE m94/108byte

Pseudo:

AT&T

FSAVE dst

FNSAVE dst

Description

This instruction stores the FPU state from a memory block located at *dst*. The FPU is then initialized.

The structure of the memory block consists of an FPU control word, an FPU status word, a tag word, the data and instruction error pointers, and the stack registers ST to ST(7). The size of the environment depends on the default operand size and the current processor operating mode. Typically, the data stored by this instruction is restored by an FRSTOR instruction.

The FNSAVE form does not check for unmasked floating point errors before acting. The FSAVE form checks for errors before acting.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSCALE Scale

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSCALE

Pseudo:

AT&T

FSCALE

Description

This instruction computes $ST \times 2^{ST(1)}$ and stores the result in ST.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

FSIN Sine

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSIN

Pseudo:

AT&T

FSIN

Description

This instruction computes $\sin(ST)$ where ST is in radians and $-2^{63} < ST < 2^{63}$. The result is placed in ST.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

FSINCOS Sine and Cosine

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSINCOS

Pseudo:

AT&T

FSINCOS

Description

This instruction computes $\sin(ST)$ and $\cos(ST)$ where ST is in radians and $-2^{63} < ST < 2^{63}$. The result of the sine function is placed in ST and then the result of the cosine operation is pushed onto the FPU stack.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

FSQRT Square Root

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSQRT

Pseudo:

AT&T

FSQRT

Description

This instruction computes $\sqrt{ST}$ and places the result is ST.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

FST Store Real
FSTP Store Real

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FST m32real
FST m64real
FST ST(i)
FSTP m32real
FSTP m64real
FSTP m80real
FSTP ST(i)

Pseudo:

AT&T

FST dst
FSTP dst

Description

These instructions store ST in *dst*. FSTP pops the FPU stack after performing the store operation.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSTCW Store Control Word
FNSTCW Store Control Word

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSTCW m2byte

FNSTCW m2byte

Pseudo:

AT&T

FSTCW dst

FNSTCW dst

Description

This instruction stores the FPU control word at the memory location *dst*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSTENV Store FPU Environment
FNSTENV Store FPU Environment

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSTENV m14/28byte

FNSTENV m14/28byte

Pseudo:

AT&T

FSTENV dst

FNSTENV dst

Description

These instructions store the FPU environment in the memory block at *dst* and then masks all floating point exceptions.

The structure of the memory block consists of an FPU control word, an FPU status word, a tag word, and the data and instruction error pointers. The size of the environment depends on the default operand size and the current processor operating mode. Typically, the data stored by this instruction is restored by an FLDENV instruction.

The FNSTENV form does not check for unmasked floating point errors before acting. The FSTENV form checks for errors before acting.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSTSW Store Status Word
FNSTSW Store Status Word

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSTSW m2byte
FSTSW
FNSTSW m2byte
FNSTSW %ax

Pseudo:

AT&T

FSTSW dst
FSTSW
FNSTSW dst

Description

These instructions store the FPU status word in *dst*.

The FNSTSW form does not check for unmasked floating point errors before acting. The FSTSW form checks for errors before acting.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSUB Subtract
FSUBP Subtract
FISUB Subtract

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSUB m32real
FSUB m64real
FSUB ST(i), ST
FSUB ST, ST(i)
FSUBP ST, ST(i)
FSUB
FISUB m32int
FISUB m16int

Pseudo:

AT&T

FSUB src1
FSUB src1, dst
FSUBP src1, dst
FSUB
FISUB src1

Description

These instructions subtracts *src1* from *dst*. The result is placed in *dst*. The no operand form and FSUBP are equivalent and both pop ST off the FPU stack after completing the subtraction. The one operand form subtracts *src1* from ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FSUBR Reverse Subtract
FSUBPR Reverse Subtract
FISUBR Reverse Subtract

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FSUBR m32real
FSUBR m64real
FSUBR ST(i), ST
FSUBR ST, ST(i)
FSUBRP ST, ST(i)
FSUBR
FISUBR m32int
FISUBR m16int

Pseudo:

AT&T

FSUBR src1
FSUBR src1, dst
FSUBRP src1, dst
FSUBR
FISUBR src1

Description

These instructions subtracts *dst* from *src1*. The result is placed in *dst*. The no operand form and FSUBRP are equivalent and both pop ST off the FPU stack after completing the subtraction. The one operand form subtracts ST from *src1* and places the result in ST.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FTST Test

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FTST

Pseudo:

AT&T

FTST

Description

This instruction compares ST to 0.0. The FPU flags C_0, C_2, C_3 are set in accordance with the result (as shown in the table below).

	$C_0 C_2 C_3$
$ST > 0.0$	0 0 0
$ST < 0.0$	1 0 0
$ST = 0.0$	0 0 1
Not Comparable	1 1 1

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FUCOM Unordered Compare Real
FUCOMP Unordered Compare Real
FUCOMPP Unordered Compare Real

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FUCOM ST(i)
FUCOM
FUCOMP ST(i)
FUCOMP
FUCOMPP

Pseudo:

AT&T

FUCOM src1
FUCOM
FUCOMP src1
FUCOMP
FUCOMPP

Description

These instructions compare two real numbers: ST and *src1*. The no operand form compares ST and ST(1). The FPU flags C_0, C_2, C_3 are set in accordance with the result (as shown in the table below). FUCOMP and FUCOMPP pop the FPU stack on completion of the comparison.

	$C_0 C_2 C_3$
$ST > src1$	0 0 0
$ST < src1$	1 0 0
$ST = src1$	0 0 1
Not Comparable	1 1 1

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FWAIT Wait

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FWAIT

Pseudo:

AT&T

FWAIT

Description

This instruction checks for pending unmasked floating point exceptions before proceeding.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FXAM Examine

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FXAM

Pseudo:

AT&T

FXAM

Description

This instruction sets the FPU flags in accordance with the type of object in ST:

	$C_0 C_2 C_3$
Unsupported	0 0 0
NaN	1 0 0
Normal	0 1 0
Infinity	1 1 0
Zero	0 0 1
Empty	1 0 1
Denormal	0 1 1

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FXCH Exchange Register Contents

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FXCH ST(i)

FXCH

Pseudo:

AT&T

FXCH dst

FXCH

Description

This instruction swaps ST and *dst*. The no operand form of the instruction swaps ST and ST(1).

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FXTRACT Extract Exponent and Significand

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FXTRACT

Pseudo:

AT&T

FXTRACT

Description

This instruction computes the exponent of ST and the significand of ST. The exponent is stored in ST and the significand is pushed onto the FPU stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FYL2X Compute $y \times \log_2 x$

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FYL2X

Pseudo:

AT&T

FYL2X

Description

This instruction computes $ST(1) \times \log_2 ST$, places the result in ST(1) and pops the FPU stack. Note: ST must not be negative.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

FYL2XP1 Compute $y \times \log_2(x+1)$

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

FYL2XP1

Pseudo:

AT&T

FYL2XP1

Description

This instruction computes $ST(1) \times \log_2(ST + 1.0)$, places the result in ST(1) and pops the FPU stack. Note: ST must have the property $-(1-(\sqrt{2}/2)) \leq ST \leq \sqrt{2} - 1$.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

HLT Halt

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

HLT

Pseudo:

AT&T

HLT

Description

This instruction places the processor into a HALT state. No instructions are executed until either an enabled interrupt, an NMI or a processor reset occurs. If execution is resumed by an interrupt, then the address of the instruction after the HLT is stored on the stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

IDIV Integer Divide

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

IDIV mem
IDIV reg
IDIV mem, %ax
IDIV reg, %ax
IDIV mem, %eax
IDIV reg, %eax

Pseudo:

AT&T

IDIV src1
IDIV src1, dst

Description

This instruction performs a signed division on the extended register pair of *dst* by dividing by *src1* leaving the quotient of the result in the lower half of the extended register pair *dst* and the remainder in the upper half of the extended register pair *dst*. The extended register pairs of the accumulators are: *%edx:%eax* for *%eax*; *%dx:%ax* for *%ax*; and *%ax* for *%al*.

The single operand form of this instruction divides the extended register pair of the accumulator - determined by the size of the size modifier of the opcode - by *src1*.
Note: that the remainder has the sign as the dividend and that the magnitude of the remainder is always less than the magnitude of the divisor.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U
CF	TF	IF	DF	NT
U				

IMUL Integer Multiply

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

IMUL mem
IMUL reg
IMUL reg, reg
IMUL mem, reg
IMUL imm, reg
IMUL imm, reg, reg
IMUL imm, mem, reg

Pseudo:

AT&T

IMUL src1
IMUL src1, dst
IMUL src1, src2, dst

Description

This instruction performs a signed multiplication on two integer values.

The single operand form multiplies the lower half of the extended register pair of the accumulator by *src1* leaving the result in the extended register pair of the accumulator. The extended register pairs of the accumulators are: *%edx:%eax* for *%eax*; *%dx:%ax* for *%ax*; and *%ax* for *%al*.

The two operand form multiplies *dst* by *src1* and leaves the result in *dst*.

The three operand form multiplies *src2* by *src1* leaving the result in *dst*.

Flags:

OF	SF	ZF	AF	PF
M	U	U	U	U
CF	TF	IF	DF	NT
M				

IN Input from Port

Flow	Int	Float	Multi	IO
				×

OpSys

386	387	486
×		×

Formats:

AT&T

IN imm, %al
IN imm, %ax
IN imm, %eax
IN %dx, %al
IN %dx, %ax
IN %dx, %eax

Pseudo:

AT&T

IN src1, dst

Description

This instruction transfers a byte, word or double word from a port in the IO address space. Only the first 256 ports may be accessed using an immediate constant. The ports from 255 to 65535 must be accessed by loading *%dx* with the port number.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

INC Incrememnt by 1

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

INC mem
INC reg

Pseudo:

AT&T

INC dst

Description

This instruction adds 1 to *dst*. Note that CF is not affected by this instruction.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT

INS Input from Port to String
INSB Input from Port to String
INSW Input from Port to String
INSD Input from Port to String

Flow	Int	Float	Multi	IO
				×

OpSys

386	387	486
×		×

Formats:

AT&T

INS %dx, reg
INS %dx, mem
INSB
INSW
INSD

Pseudo:

AT&T

INS src1, dst
INSB
INSW
INSD

Description

These instructions transfer a byte, word or double word from a port in the IO address space to *%es:%edi*. Only the first 256 ports may be accessed using an immediate constant. The ports from 255 to 65535 must be accessed by loading *%dx* with the port number.

In the two operand form the size of the transfer is determined by the size of *dst*. The no operand form implicitly determines the size of the operation.

After the transfer is completed and if DF is 0 then *%edi* is incremented. Otherwise *%edi* is decremented.

This instruction is typically used with a repeat prefix.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT
			T	

INT Call Interrupt Procedure
INTO Call Interrupt Procedure
INT3 Call Interrupt Procedure

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

INT imm
INT3
INTO

Pseudo:

AT&T

INT dst
INT3
INTO

Description

These instructions generate an interrupt via software. In the case of the one operand form interrupt *dst* is generated. INT3 generates interrupt 3. INTO generates interrupt 4 if the overflow flag is set.

Flags:

OF	SF	ZF	AF	PF
T				
CF	**TF**	**IF**	**DF**	**NT**
	0			0

INVD Invalidate Cache

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
		×

Formats:

AT&T

INVD

Pseudo:

AT&T

INVD

Description

This instruction flushes the internal cache and issues a special bus cycle which indicates that external caches should be flushed. Note: Data in external write-back caches is discarded.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

INVLPG Invalidate TLB Entry

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
		×

Formats:

AT&T

INVLPG mem

Pseudo:

AT&T

INVLPG src1

Description

This instruction invalidates an entry in the translation look aside buffer (TLB) if it maps the address of *src1*.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

IRET Interrupt Return
IRETD Interrupt Return

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

IRET

IRETD

Pseudo:

AT&T

IRET

IRETD

Description

These instructions return the flow of control, at the end of an interrupt handler, to the location where the interrupt occured.

This instruction may be used to change privelege level and task.

Flags:

OF	SF	ZF	AF	PF
R	R	R	R	R
CF	TF	IF	DF	NT
R	R	R	R	T

JA **Jump if above** $(CF = 0 \cdot ZF = 0)$
JAE **Jump if above or equal** $(CF = 0)$
JB **Jump if below** $(CF = 1)$
JBE **Jump if below or equal** $(CF = 1 + ZF = 1)$
JC **Jump if carry** $(CF = 1)$
JCXZ **Jump if CX register is 0**
JECXZ **Jump if ECX register is 0**
JE **Jump if equal** $(ZF = 1)$
JZ **Jump if zero** $(ZF = 1)$
JG **Jump if greater** $(ZF = 0 \cdot SF = OF)$
JGE **Jump if greater or equal** $(SF = OF)$
JL **Jump if less** $(SF \neq OF)$
JLE **Jump if less or equal** $(ZF = 1 + SF \neq OF)$
JNA **Jump if not above** $(CF = 1 + ZF = 1)$
JNAE **Jump if not above or equal** $(CF = 1)$
JNB **Jump if not below** $(CF = 0)$
JNBE **Jump if not below or equal** $(CF = 0 \cdot ZF = 0)$
JNC **Jump if not carry** $(CF = 0)$
JNE **Jump if not equal** $(ZF = 0)$
JNG **Jump if not greater** $(ZF = 1 + SF \neq OF)$
JNGE **Jump if not greater or equal** $(SF \neq OF)$
JNL **Jump if not less** $(SF = OF)$
JNLE **Jump if not less or equal** $(ZF = 0 \cdot SF = OF)$
JNO **Jump if not overflow** $(OF = 0)$
JNP **Jump if not parity** $(PF = 0)$
JNS **Jump if not sign** $(SF = 0)$
JNZ **Jump if not zero** $(ZF = 0)$
JO **Jump if overflow** $(OF = 0)$
JP **Jump if parity** $(PF = 1)$
JPE **Jump if parity even** $(PF = 1)$
JPO **Jump if parity odd** $(PF = 0)$
JS **Jump if sign** $(SF = 1)$
JZ **Jump if zero** $(ZF = 1)$

Flow	Int	Float	Multi	IO
×				

OpSys

386	387	486
×		×

Formats:
AT&T
Jcc ofs

Pseudo:
AT&T
Jcc dst

Description

These instructions test flags and generate a relative jump to the current EIP if the condition is satisfied.

Flags:

OF	SF	ZF	AF	PF
T	T	T		T
CF	TF	IF	DF	NT
T				

JMP Jump

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

JMP reg
JMP mem
JMP ofs
JMP ptr

Pseudo:

AT&T

JMP dst

Description

This instruction generates an unconditional jump to a memory location. The memory location may be relative to the current location, or absolute.

This instruction may be used to change privelege level or task.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

LAHF Load Flags into AH Register

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

LAFH

Pseudo:

AT&T

LAFH

Description

This instruction copies the low byte of the flags word to *%ah*.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

LAR Load Access Rights Byte

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

LAR reg, reg

LAR mem, reg

Pseudo:

AT&T

LAR src1, dst

Description

This instruction loads a masked version of the access rights bits indicated by the descriptor *src1*. If the descriptor is valid, within the descriptor limits, and visible at the current privelege level then the access rights byte masked by 00FxFF00 hex (where x is undefined) is stored in *dst* and ZF cleared. Otherwise, ZF is set. If the destination register is 16 bits wide then the lower 2 bytes of the masked access rights are stored.

Flags:

OF	SF	ZF	AF	PF
		M		
CF	**TF**	**IF**	**DF**	**NT**

LEA Load Efective Address

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

LEA mem, reg

Pseudo:

AT&T

LEA src1, dst

Description

This instruction calculates the effective address of *src1* and stores it in *dst*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LEAVE High Level Procedure Exit

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

LEAVE

Pseudo:

AT&T

LEAVE

Description

This instruction returns a stack to the state equivalent to the state of the stack prior to the use of an ENTER instruction. It frees local memory, removes links to prior lexical nesting levels and restores the frame pointer. LEAVE moves *%bp* or *%ebp* to *%sp* or *%esp* and pops the old frame pointer into *%bp* or *%ebp*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LGDT Load Global Descriptor Table Register
LIDT Load Interrupt Descriptor Table Register

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

LGDT mem

LIDT mem

Pseudo:

AT&T

LGDT src1

LIDT src1

Description

These instructions loads the appropriate descriptor table register with base and limit from memory. LGDT loads the global descriptor table register. LIDT loads the interrupt descriptor table register.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LDS Load Full Pointer
LES Load Full Pointer
LFS Load Full Pointer
LGS Load Full Pointer
LSS Load Full Pointer

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

Lss mem, reg

Pseudo:

AT&T

Lss src1, dst

Description

These instructions load the register *dst* from the effective address of *src1* and then loads the appropriate segment register (ss) from the 16 bits following the value transfered to the register *dst*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LLDT Load Local Descriptor Table Register

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

LLDT mem

Pseudo:

AT&T

LLDT src1

Description

This instruction loads the local descriptor table register with a selector, *src1*, from the GDT. If selector 0 is loaded then the local descriptor table register is marked invalid.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LMSW Load Machine Status Word

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

LMSW reg

LMSW mem

Pseudo:

AT&T

LMSW src1

Description

This instruction loads the low 16 bits of CR0 with the contents of *src1*. This instruction is provided for compatibility with the 286. Note that this instruction will not switch the processor out of protected mode.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LOCK Assert LOCK# Signal Prefix

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

LOCK

Pseudo:

AT&T

LOCK

Description

This instruction causes the LOCK# signal of the 486 processor to be asserted during the instruction that follows it. The LOCK prefix may only be followed by the following:

Instruction	Operating On
BTC, BTR, BTS	*reg/mem, mem*
XCHG	*mem, reg*
XCHG	*reg, mem*
ADD, ADC, AND, OR, SBB, SUB, XOR	*reg/imm, mem*
DEC, INC, NEG, NOT	*mem*

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

LODS Load String Operand
LODSB Load String Operand
LODSW Load String Operand
LODSD Load String Operand

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

LODS mem
LODS
LODSB
LODSW
LODSD

Pseudo:

AT&T

LODS src1
LODSB
LODSW
LODSD

Description

This instruction loads the accumulator from the memory location pointed to by the source index register.
In the one operand form the size of the transfer is determined by the size of *src1*. The no operand form implicitly determines the size of the operation.

After the transfer is completed and if DF is 0 then *%edi* is incremented. Otherwise *%edi* is decremented.

This instruction is typically used with a repeat prefix.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT
			T	

LOOP Loop if ECX not equal to 0
LOOPE Loop if ECX not equal to 0 and ZF = 1
LOOPZ Loop if ECX not equal to 0 and ZF = 1
LOOPNE Loop if ECX not equal to 0 and ZF = 0
LOOPNZ Loop if ECX not equal to 0 and ZF = 0

Flow	Int	Float	Multi	IO
×				

OpSys

386	387	486
×		×

Formats:

AT&T

LOOPc ofs

Pseudo:

AT&T

LOOPc dst

Description

This instruction decrements *%ecx*. It performs a relative jump to *dst* if the condition is met.

Flags:

OF	SF	ZF	AF	PF
		T		

CF	TF	IF	DF	NT

LSL Load Segment Limit

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

LSL mem, reg

LSL reg, reg

Pseudo:

AT&T

LSL src1, dst

Description

This instruction loads *dst* with the segment limit of the descriptor indicated by *src1* provided that the descriptor is visible at the current privelege level, valid, and within descriptor table limits.

Flags:

OF	SF	ZF	AF	PF
		M		

CF	TF	IF	DF	NT

LTR Load Task Register

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

LTR reg

LTR mem

Pseudo:

AT&T

LTR src1

Description

This instruction loads the task register with the contents of *src1* and marks the loaded TSS busy. A task switch does not occur.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

MOV Move Data

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

MOV imm, mem

MOV reg, mem

MOV imm, reg

MOV mem, reg

MOV reg, reg

Pseudo:

AT&T

MOV src1, dst

Description

This instruction copies the contents of *src1* to *dst*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

MOV Move to/from Segment Registers

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

MOV reg16, sreg

MOV sreg, reg16

Pseudo:

AT&T

MOV src1, dst

Description

This instruction copies the contents of *src1* to *dst*. As the segment registers are 16 bits in size, both operands must be 16 bits wide. Note that in protected mode the segment registers are loaded with descriptors and that the base and limits of the segments are found by reference to the descriptor table. In real mode the segment register contains the base address of the segment and the limit is fixed at 64 Kbytes.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U

CF	TF	IF	DF	NT
U				

MOV Move to/from Special Registers

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

MOV reg32, %cr0/%cr2/%cr3
MOV reg32, %dr0/%dr1/%dr2/%dr3
MOV reg32, %dr6/%dr7
MOV reg32, %tr4/%tr5/%tr6/%tr7
MOV %cr0/%cr2/%cr3, reg32
MOV %dr0/%dr1/%dr2/%dr3, reg32
MOV %dr6/%dr7, reg32
MOV %tr4/%tr5/%tr6/%tr7, reg32

Pseudo:

AT&T

MOV src1, dst

Description

This instruction copies the contents of *src1* to *dst*. This instruction can modify special registers. The special registers are used in testing the processor, controlling the operating mode of the processor and debugging support for the processor.

Flags:

OF	SF	ZF	AF	PF
U	U	U	U	U
CF	TF	IF	DF	NT
U				

MOVS Move Data from String to String
MOVSB Move Data from String to String
MOVSW Move Data from String to String
MOVSD Move Data from String to String

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

MOVS mem, mem
MOVSB
MOVSW
MOVSD

Pseudo:

AT&T

MOVS src1, dst
MOVSB
MOVSW
MOVSD

Description

These instructions transfer a byte, word or double word from *%esi* to *%es:%edi*.
In the two operand form the size of the transfer is determined by the size of *dst*. A segment overide is possible for *dst*. The no operand form implicitly determines the size of the operation.

After the transfer is completed and if DF is 0 then *%edi* is incremented. Otherwise *%edi* is decremented.

This instruction is typically used with a repeat prefix.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT
			T	

MOVSww Move with Sign-Extend (AT&T Only)

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

MOVSww mem, reg

MOVSww reg, reg

Pseudo:

AT&T

MOVSww src1, dst

Description

These instructions move a value from *src1* to *dst* after sign extending the value. The MOVSww instruction determines the conversion based on the two size modifiers located at the end of the instruction. The values of *ww* may be: *bl*, *bw*, and *wl*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

MOVZww Move with Zero-Extend (AT&T Only)

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

MOVZX mem, reg

MOVZX reg, reg

Pseudo:

AT&T

MOVZww src1, dst

Description

These instructions move a value from *src1* to *dst* after zero extending the value. The MOVZww instruction determines the conversion based on the two size modifiers located at the end of the instruction. The values of *ww* may be: *bl*, *bw*, and *wl*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

MUL Unsigned Multiplication of AL or AX or EAX

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

MUL reg

MUL mem

Pseudo:

AT&T

MUL src1

Description

This instruction performs an unsigned multiplication on two integer values. It multiplies the lower half of the extended register pair of the accumulator by *src1* leaving the result in the extended register pair of the accumulator. Under an **AT&T** assembler the extended register pair is determined by the size modifier of the instruction. The extended register pairs of the accumulators are: *%edx:%eax* for *32-bit modifier*; *%dx:%ax* for *16-bit modifier*; and *%ax* for *8-bit modifier*.

Flags:

OF	SF	ZF	AF	PF
M	U	U	U	U
CF	TF	IF	DF	NT
M				

NEG Two's Complement Negation

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

NEG reg

NEG mem

Pseudo:

AT&T

NEG dst

Description

This instruction calculates the two's complement negation of the integer *dst*.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT
M				

NOP No Operation

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

NOP

Pseudo:

AT&T

NOP

Description

This instruction performs no operation.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

NOT One's Complement Negation

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

NOT reg

NOT mem

Pseudo:

AT&T

NOT dst

Description

This instruction performs a logical NOT on each bit of the integer *dst*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

OR Logical Inclusive OR

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

OR imm, mem
OR reg, mem
OR imm, reg
OR mem, reg
OR reg, reg

Pseudo:

AT&T

OR src1, dst

Description

This instruction performs a logical OR on each bit of two integers - *src1* and *dst* - leaving the result in *dst*. CF and OF are cleared and PF, SF, and ZF are set according to the result.

Flags:

OF	SF	ZF	AF	PF
0	M	M	U	M

CF	TF	IF	DF	NT
0				

OUT Output to Port

Flow	Int	Float	Multi	IO
				×

OpSys

386	387	486
×		×

Formats:

AT&T

OUT %al, imm
OUT %ax, imm
OUT %eax, imm
OUT %al, %dx
OUT %ax, %dx
OUT %eax, %dx

Pseudo:

AT&T

OUT src1, dst

Description

This instruction transfers a byte, word or double word in the accumulator to a port in the IO address space. Only the first 256 ports may be accessed using an immediate constant. The ports from 255 to 65535 must be accessed by loading *%dx* with the port number.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

OUTS Output String to Port
OUTSB Output String to Port
OUTSW Output String to Port
OUTSD Output String to Port

Flow	Int	Float	Multi	IO
				×

OpSys

386	387	486
×		×

Formats:

AT&T

OUTS reg, %dx
OUTS mem, %dx
OUTSD
OUTSW
OUTSD

Pseudo:

AT&T

OUTS src1, dst
OUTSB
OUTSW
OUTSD

Description

These instructions transfer a byte, word or double word to a port in the IO address space to *%es:%edi*. Only the first 256 ports may be accessed using an immediate constant. The ports from 255 to 65535 must be accessed by loading *%dx* with the port number.

In the two operand form the size of the transfer is determined by the size of *dst*. The no operand form implicitly determines the size of the operation.

After the transfer is completed and if DF is 0 then *%edi* is incremented. Otherwise *%edi* is decremented.

This instruction is typically used with a repeat prefix.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT
			T	

POP Pop a Word from the Stack

Flow	Int	Float	Multi	IO
	×		×	

OpSys

386	387	486
×		×

Formats:

AT&T

POP reg
POP mem
POP sreg

Pseudo:

AT&T

POP dst

Description

This instruction copies the word or doubleword pointed to by *%sp* or *%esp* in the stack segment to *dst*. It then adds 2 for a word or a byte size operation to the stack pointer, or 4 for a doubleword to the stack pointer.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

POPA Pop All General Registers
POPAD Pop All General Registers

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

POPA
POPAD

Pseudo:

AT&T

POPA
POPAD

Description

These instructions pop the following registers from the stack: *%edi*, *%esi*, *%ebp*, *%esp*, *%ebx*, *%edx*, *%ecx*, and *%eax*. Note that the value *%esp* found on the stack is disposed of, and does not alter *%esp*.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

POPF Pop Stack into Flags Register
POPFD Pop Stack into Flags Register

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

POPF

POPFD

Pseudo:

AT&T

POPF

POPFD

Description

These instructions pop a 32 bit quantity off the stack into the EFLAGS register.

Flags:

OF	SF	ZF	AF	PF
R	R	R	R	R
CF	TF	IF	DF	NT
R	R	R	R	R

PUSH Push Operand onto the Stack

Flow	Int	Float	Multi	IO
	×		×	

OpSys

386	387	486
×		×

Formats:

AT&T

PUSH reg

PUSH mem

PUSH sreg

PUSH imm

Pseudo:

AT&T

PUSH src1

Description

This instruction copies *dst* into the word or doubleword pointed to by *%sp* or *%esp* in the stack segment. It then subtracts 2 for a word or a byte size operation from the stack pointer, or 4 for a doubleword from the stack pointer.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

PUSHA Push All General Registers
PUSHAD Push All General Registers

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

PUSHA

PUSHAD

Pseudo:

AT&T

PUSHA

PUSHAD

Description

These instructions push the following onto the stack: *%eax*, *%ecx*, *%edx*, *%ebx*, the value of *%esp* before the instruction commenced, *%ebp*, *%esi*, and *%edi*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

PUSHF Push Flags Register onto the Stack
PUSHFD Push Flags Register onto the Stack

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

PUSHF

PUSHFD

Pseudo:

AT&T

PUSHF

PUSHFD

Description

These instructions push EFLAGS onto the stack.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

RCL Rotate (Carry Left)
RCR Rotate (Carry Right)
ROL Rotate (Left)
ROR Rotate (Right)

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

Rdd imm, mem
Rdd imm, reg
Rdd %cl, reg
Rdd %cl, mem

Pseudo:

AT&T

Rdd cnt, dst

Description

These instructions rotate the bits of *dst* by *cnt*. The RCx forms rotate through the carry bit, enlarging the destination *dst* by one bit. In the ROx form the bit shifted *dst* is stored in CF.

Flags:

OF	SF	ZF	AF	PF
M				

CF	TF	IF	DF	NT
TM				

REP Repeat Following String Operation
REPE Repeat While Equal Following String Operation
REPNE Repeat While Not Equal Following String Operation
REPZ Repeat While Equal Following String Operation
REPNZ Repeat While Not Equal Following String Operation

Flow	Int	Float	Multi	IO
×	×			

OpSys

386	387	486
×		×

Formats:

AT&T

REPcc

Pseudo:

AT&T

REPcc ins

Description

These instructions cause the following instruction *ins* to be repeated while the condition is satisfied.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

RET Return from Procedure or Function

Flow	Int	Float	Multi	IO
×			×	

OpSys

386	387	486
×		×

Formats:

AT&T

RET

RET imm

Pseudo:

AT&T

RET

RET cnt

Description

This instruction pops the value pointed to by the stack pointer into EIP. If *cnt* is present then it is added to the stack pointer.

This instruction may cause the privelege level to change.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

SAHF Store AH into Flags

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SAHF

Pseudo:

AT&T

SAHF

Description

This instruction copies *%ah* to the low byte of the flags word.

Flags:

OF	SF	ZF	AF	PF
	R	R	R	R
CF	TF	IF	DF	NT
R				

SAL Shift Arithmetic Left
SAR Shift Arithmetic Right
SHL Shift Left
SHR Shift Right

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

Sdd imm, mem
Sdd imm, reg
Sdd %cl, reg
Sdd %cl, mem

Pseudo:

AT&T

Sdd cnt, dst

Description

These instructions shift the bits of *dst* by *cnt*. The bit shifted out of *dst* is stored in CF. For SAL, SHL, and SHR zeros are shifted in to fill the vacated bits. For SAR the top bit is duplicated into the vacated bit.

Flags:

OF	SF	ZF	AF	PF
M	M	M	U	M
CF	TF	IF	DF	NT
M				

SBB Integer Subtraction with Borrow

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SBB imm, mem
SBB reg, mem
SBB imm, reg
SBB mem, reg
SBB reg, reg

Pseudo:

AT&T

SBB src1, dst

Description

This instruction adds CF to *src1* and then subtracts the result from *dst*. Immediate operands are sign extended before the operation is performed.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M

CF	TF	IF	DF	NT
TM				

SCAS Compare String Data
SCASB Compare String Data
SCASW Compare String Data
SCASD Compare String Data

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SCAS mem
SCASB
SCASW
SCASD

Pseudo:

AT&T

SCAS src1
SCASB
SCASW
SCASD

Description

These instructions subtract the byte, word or double word *%es:%edi* from the accumulator.
In the one operand form the size of the transfer is determined by the size of *src1*. The no operand form implicitly determines the size of the operation.

After the transfer is completed and if DF is 0 then *%edi* is incremented. Otherwise *%edi* is decremented.

This instruction is typically used with a repeat prefix.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	TF	IF	DF	NT
M			T	

SETA Set byte if above $(CF = 0 \cdot ZF = 0)$
SETAE Set byte if above or equal $(CF = 0)$
SETB Set byte if below $(CF = 1)$
SETBE Set byte if below or equal $(CF = 1 + ZF = 1)$
SETC Set byte if carry $(CF = 1)$
SETE Set byte if equal $(ZF = 1)$
SETG Set byte if greater $(ZF = 0 + SF = OF)$
SETGE Set byte if greater or equal $(SF = OF)$
SETL Set byte if less $(SF \neq OF)$
SETLE Set byte if less or equal $(ZF = 1 + SF \neq OF)$
SETNA Set byte if not above $(CF = 1)$
SETNAE Set byte if not above or equal $(CF = 1)$
SETNB Set byte if not below $(CF = 0)$
SETNBE Set byte if not below or equal $(CF = 0 \cdot ZF = 0)$
SETNC Set byte if not carry $(CF = 0)$
SETNE Set byte if not equal $(ZF = 0)$
SETNG Set byte if not greater $(ZF = 1 + SF \neq OF)$
SETNGE Set byte if not greater or equal $(SF \neq OF)$
SETNL Set byte if not less $(SF = OF)$
SETNLE Set byte if not less or equal $(ZF = 0 \cdot SF = OF)$
SETNO Set byte if not overflow $(OF = 0)$
SETNP Set byte if not parity $(PF = 0)$
SETNS Set byte if not sign $(SF = 0)$
SETNZ Set byte if not zero $(ZF = 0)$
SETO Set byte if overflow $(OF = 1)$
SETP Set byte if parity $(PF = 1)$
SETPE Set byte if parity even $(PF = 1)$
SETPO Set byte if parity odd $(PF = 0)$
SETS Set byte if sign $(SF = 1)$
SETZ Set byte if zero $(ZF = 1)$

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SETcc reg

SETcc mem

Pseudo:

AT&T

SETcc dst

Description

These instructions store 1 in the byte *dst* if the condition is met, otherwise a 0 is stored.

Flags:

OF	SF	ZF	AF	PF
T	T	T		T
CF	TF	IF	DF	NT
T				

SGDT Store Global Descriptor Table Register
SIDT Store Interrupt Descriptor Table Register

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

SGDT mem

SIDT mem

Pseudo:

AT&T

SGDT dst

SIDT dst

Description

These instructions copy the appropriate descriptor table register to the memory location *dst*.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT

SHLD Double Precision Shift Left

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SHLD imm, reg, reg
SHLD imm, reg, mem
SHLD %cl, reg, reg
SHLD %cl, reg, mem

Pseudo:

AT&T

SHLD cnt, src1, dst

Description

This instruction shifts left the concatenated registers *dst:src1* by *cnt*.

Flags:

OF	SF	ZF	AF	PF
U	M	M	U	M
CF	TF	IF	DF	NT
M				

SHRD Double Precision Shift Right

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SHRD imm, reg, reg
SHRD imm, reg, mem
SHRD %cl, reg, reg
SHRD %cl, reg, mem

Pseudo:

AT&T

SHRD cnt, src1, dst

Description

This instruction shifts right the concatenated registers *dst:src1* by *cnt*.

Flags:

OF	SF	ZF	AF	PF
U	M	M	U	M
CF	TF	IF	DF	NT
M				

SLDT Store Local Descriptor Table

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

SLDT mem

SLDT reg16

Pseudo:

AT&T

SLDT dst

Description

This instruction stores the local descriptor table register at *dst*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

SMSW Store Machine Status Word

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

SMSW mem

SMSW reg16

Pseudo:

AT&T

SMSW dst

Description

This instruction stores the low 16 bits of CR0 at *dst*. This instruction is provided for compatibility with the 286.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

STC Set Carry Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

STC

Pseudo:

AT&T

STC

Description

This instruction sets CF to 1.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**
1				

STD Set Direction Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

STD

Pseudo:

AT&T

STD

Description

This instruction sets DF to 1.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**
			1	

STI Set Interrupt Flag

Flow	Int	Float	Multi	IO

OpSys

386	387	486
×		×

Formats:

AT&T

STI

Pseudo:

AT&T

STI

Description

If the current privelege is equal to or more priveleged than IOPL then this instruction sets IF to 1.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT
		1		

STOS Store String Data
STOSB Store String Data
STOSW Store String Data
STOSD Store String Data

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

STOS mem
STOSB
STOSW
STOSD

Pseudo:

AT&T

STOS dst
STOSB
STOSW
STOSD

Description

These instructions transfers the byte, word or doubleword from the accumulator to *%es:%edi*.
In the one operand form the size of the transfer is determined by the size of *src1*. The no operand form implicitly determines the size of the operation.

After the transfer is completed and if DF is 0 then *%edi* is incremented. Otherwise *%edi* is decremented.

This instruction is typically used with a repeat prefix.

Flags:

OF	SF	ZF	AF	PF
CF	TF	IF	DF	NT
			T	

STR Store Task Register

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

STR mem

STR reg16

Pseudo:

AT&T

STR dst

Description

This instruction stores the task register at *src1*.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

SUB Integer Subtraction

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

SUB imm, mem

SUB reg, mem

SUB imm, reg

SUB mem, reg

SUB reg, reg

Pseudo:

AT&T

SUB src1, dst

Description

This instruction subtracts *src1* from *dst*. Immediate operands are sign extended before the operation is performed.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	**TF**	**IF**	**DF**	**NT**
M				

TEST Logical Compare

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

TEST imm, mem
TEST reg, mem
TEST imm, reg
TEST mem, reg
TEST reg, reg

Pseudo:

AT&T

TEST src1, src2

Description

This instruction performs the function $src2\ AND\ src1$ setting the flags in accordance with the result. The result of the Logical AND operation is NOT stored.

Flags:

OF	SF	ZF	AF	PF
0	M	M	U	M

CF	TF	IF	DF	NT
0				

VERR Verify a Segment for Reading
VERW Verify a Segment for Writing

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
×		×

Formats:

AT&T

VERR reg
VERR mem
VERW reg
VERW mem

Pseudo:

AT&T

VERR src1
VERW src1

Description

These instructions test whether a segment is accessible for a given type of operation: reading (VERR) or writing (VERW). *src1* is the descriptor of the segment to be tested. If the segment is accessible then the ZF flag is set otherwise ZF is cleared.

Flags:

OF	SF	ZF	AF	PF
		M		

CF	TF	IF	DF	NT

WAIT Wait

Flow	Int	Float	Multi	IO
		×		

OpSys

386	387	486
	×	×

Formats:

AT&T

WAIT

Pseudo:

AT&T

WAIT

Description

This instruction checks for pending unmasked floating point exceptions before proceeding.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

WBINVD Write-Back Invalidate Cache

Flow	Int	Float	Multi	IO

OpSys
×

386	387	486
		×

Formats:

AT&T

WBINVD

Pseudo:

AT&T

WBINVD

Description

This instruction flushes the internal cache and issues a special bus cycle which indicates that external caches should write back their contents to memory. A second special bus cycle is issued which indicates that external caches should be flushed.

Flags:

OF	SF	ZF	AF	PF
CF	**TF**	**IF**	**DF**	**NT**

XADD Exchange and Add

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
		×

Formats:

AT&T

XADD reg, reg

XADD reg, mem

Pseudo:

AT&T

XADD src1, dst

Description

This instruction loads *src1* from *dst* and stores the sum of the original values into *dst*.

Flags:

OF	SF	ZF	AF	PF
M	M	M	M	M
CF	**TF**	**IF**	**DF**	**NT**
M				

XCHG Exchange Register/Memory with Register

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

XCHG reg, reg

XCHG reg, mem

XCHG mem, reg

Pseudo:

AT&T

XCHG src1, dst

Description

This instruction swaps *src1* and *dst*. If either *src1* or *dst* is a memory location then the LOCK# bus signal is asserted.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

XLAT Table Lookup Translation
XLATB Table Lookup Translation

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

XLAT mem

XLATB

Pseudo:

AT&T

XLAT src1

XLATB

Description

These instructions are used to access information contained in a table located at *%ebx*. *%al* is set to the value of the byte located at *%ebx+%al*. The default segment for the table is *%ds*.

Flags:

OF	SF	ZF	AF	PF

CF	TF	IF	DF	NT

XOR Logical Exclusive OR

Flow	Int	Float	Multi	IO
	×			

OpSys

386	387	486
×		×

Formats:

AT&T

XOR imm, mem
XOR reg, mem
XOR imm, reg
XOR mem, reg
XOR reg, reg

Pseudo:

AT&T

XOR src1, dst

Description

This instruction performs a logical XOR on each bit of two integers - *src1* and *dst* - leaving the result in *dst*. CF and OF are cleared and PF, SF, and ZF are set according to the result.

Flags:

OF	SF	ZF	AF	PF
0	M	M	U	M
CF	TF	IF	DF	NT
0				

Bibliography

Intel Corporation, *i486 Microporcessor Programmer's Reference Manual*, Osborne McGraw-Hill, 1990.

Crawford, John H., Gelsinger Patrick P., *Programming the 80386*, Sybex, 1987.

Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

Hennessy, John L., Patterson, David A., *Computer Architecture: A Quanitative Approach*, Morgan Kaufmann, 1990.