Assembly Language and Architecture

Second Student Edition

Maurice Castro

Assembly Language and Architecture:
Second Student Edition

Printing History

1995:	Assembly Language and Architecture: Student Edition
1996:	Assembly Language and Architecture: Second Student Edition

ISBN: 0 646 28775 3

Contents

Overview

This textbook is to support a course which 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.

Knowledge of computer architecture is also extremely valuable in optimising programs which run on that architecture. Thus computer architecture knowledge has practical value for programmers as well as being an important area of the discipline of Computer Science.

Chapter 1

Processor Architecture

This book focuses on the Intel 80386 computer architecture as an example of a popular modern computer system.

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

Computer 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 Number Representation

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**. The definition of the bits of the **eflags** register 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.

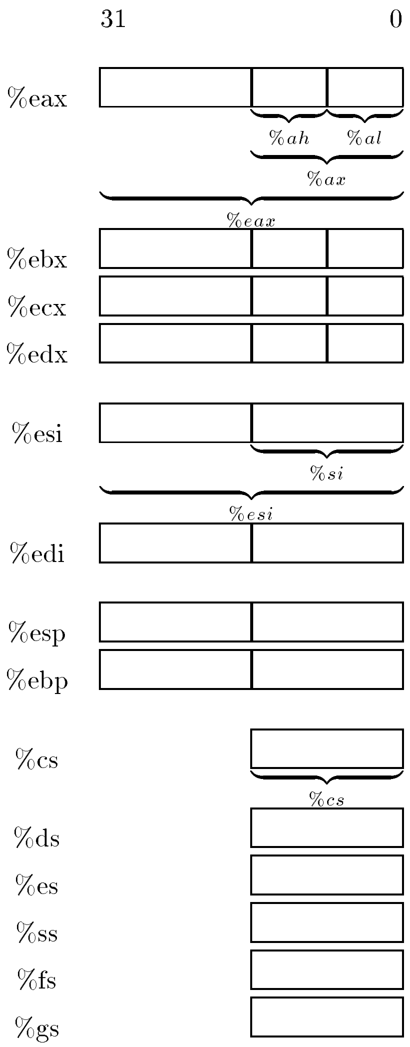


Figure 1.2: The 386/486 register set

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

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

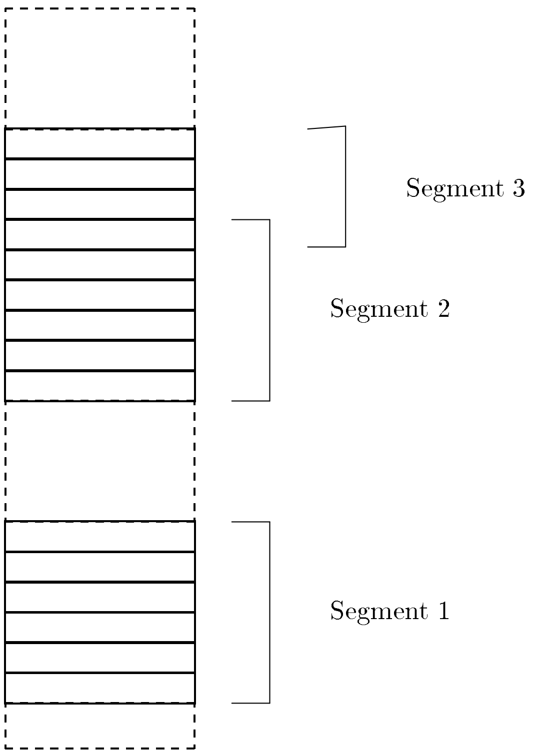


Figure 1.4: Segmentation

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.

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:

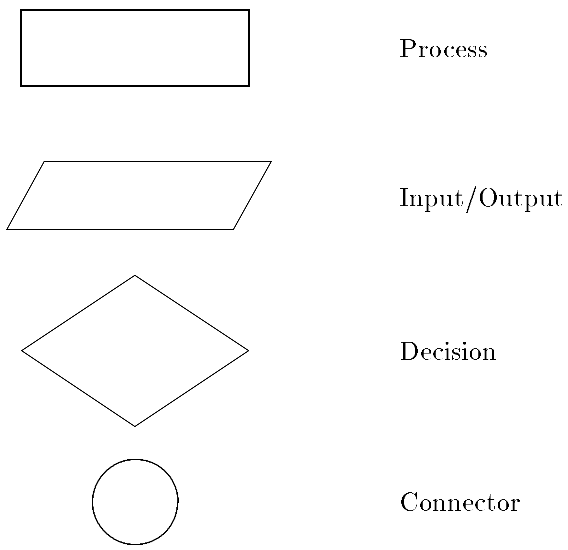


Figure 2.1: The elements of a flow chart

- 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 assembly language.

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 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 7.

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 value of data1 */
        addl d2, %eax    /* Add value of data2 to data1 */
        addl $2, %eax    /* Add 2 to the sum */
        movl %eax, 100   /* Store 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      /* reserve 32 bits and put 5 in it */
.long 5, 7   /* reserve 2 longs and put 5 and 7 in them */
.byte 4, 6   /* reserve 2 bytes and put 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 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 content of address 10 to 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 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 contents of address 10 to EAX */
subl $10, %eax    /* Subtract 10 from EAX */
xorl %ebx, %eax   /* EAX = EAX xor EBX */
addb %edx, 10     /* add 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 = 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

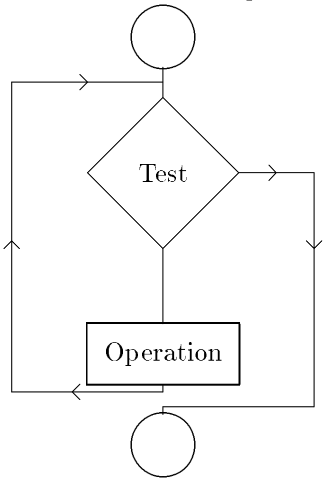
Control Structures

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.



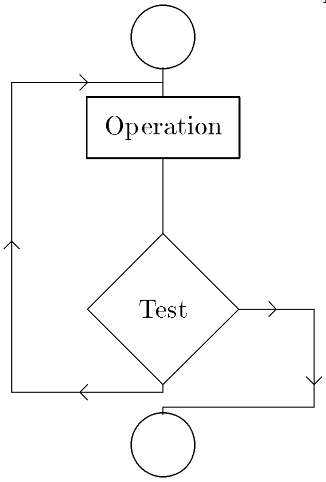
Example:

```
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.



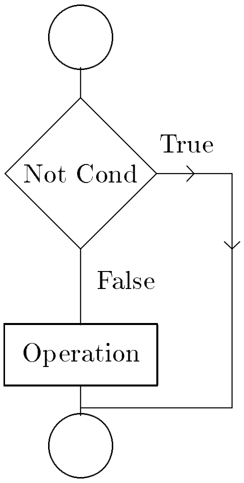
Example:

```
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.



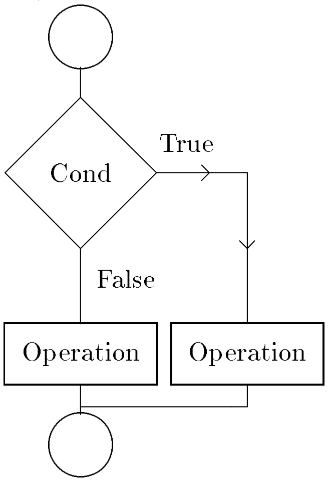
Example:

```
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.



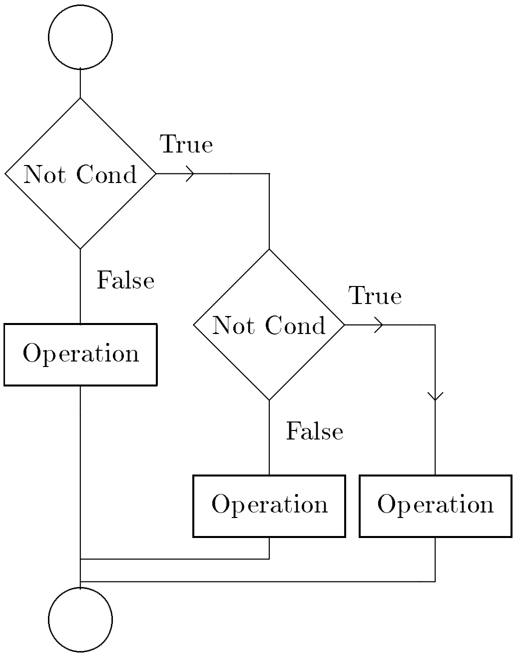
Example:

```
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 9.4).

Chapter 5

Subroutines - Introduction

The subroutine is the primary mechanism used in structured programming to allow the division of large programs into more manageable 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 */
```

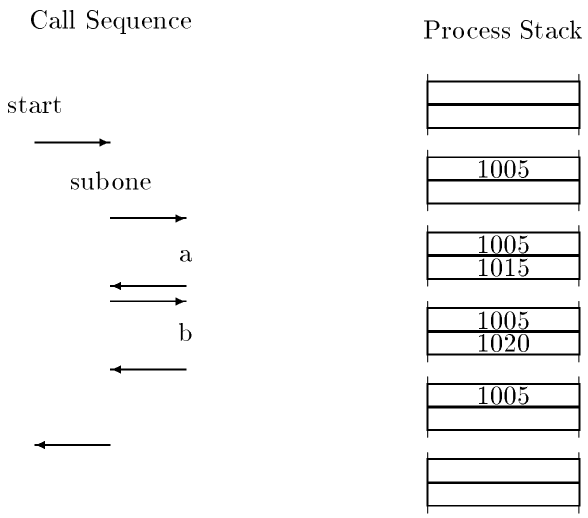


Figure 5.1: Nested Subroutines and the System Stack

Chapter 6

Macros

One of the many uses of the subroutine construct (introduced in chapter 5) is to allow a piece of code to be invoked more than once. Macros perform a similar role in that they allow the programmer to include a piece of code many times within a single program without the need to explicitly rewrite the code. This facility is used to enhance the readability and maintainability of code.

This chapter will introduce the macro concept and discuss the **m4** macro package. A macro is defined as follows:

> A macro is a string in which particular sites in the string are marked so that other strings can be inserted at those sites

A macro processor takes an input file and performs substitutions on the input text to produce output text. This process is known as macro expansion. All the actions of the macro processor occur at the source code level.

Both subroutines and macros provide mechanisms for constructing modular code. However, there are significant differences between the two approaches. Because macros are expanded before the code is assembled they have the advantage that they do not incur the overheads of a call and a return. The cost of a call is the time taken to write the address of the return location onto the stack. For a return the cost is the time taken to retrieve the return location from the stack.

Using macros can result in a 'code explosion'. As each use of a macro generates the code contained within the macro the size of the assembled program increases proportionately. The use of large numbers of macros can result in a great increase in a program's size.

Subroutines have the additional advantage of supporting recursion. As macros do not make use of the stack they cannot be used to implement recursive routines.

6.1 Using M4 with GNU As

This section will provide a minimal introduction to some functions of the **m4** macro processor and how those functions are used with GNU As.

Three **m4** instructions will be used:

- changequote
- include
- define
- undefine

Before **m4** macro processor can be used to process an assembly language file it is necessary to change the macro processor's quote characters from ' ' ' and ' ' ' to characters which are not used by the assembler. This is accomplished through the *changequote* instruction. A suitable choice of alternate characters would be:

```
changequote([,])
```

which changes the macro processor's opening quotation character to '[' and the closing quotation character to ']'.

The *include* instruction is used to include the contents of another file at the current location. This is typically used to import definitions which are common to a number of separately assembled modules. The instruction:

```
include(seg.h)
```

would include the file 'seg.h' at the current location.

The *define* instruction is used to create new macros. The format of the instruction is

define(*name*, *replacement*)

Each instance of *name* is replaced by *replacement* in the text processed by the macro processor. The macro name is only recognized when it is surrounded by non-alphanumerics. The code

```
changequote([,])
define(N, [1])
addl N, d1
addl NN, %eax
```

produces

```
addl 1, d1
addl NN, %eax
```

after being run through the **m4** macro processor.

The use of arguments in macros allows different invocations of the macro to have different results. Within the *replacement* section of the macro each occurrence of **$n** is replaced by the nth argument when the macro is used. Only the first nine arguments are accessible and any arguments which are not supplied are replaced with an empty string. The arguments of a macro are drawn from a comma separated list contained in parenthesis immediately following the macro name. **NOTE:** no space is permitted between the macro name and the opening parenthesis. The code

```
changequote([,])
define(pushb, [movb $1, %eax
movb %eax, (%esp)
decl %esp])
pushb(d1)
pushb(d2)
```

produces

```
movb d1, %eax
movb %eax, (%esp)
decl %esp
movb d2, %eax
movb %eax, (%esp)
decl %esp
```

after being run through the **m4** macro processor.

The undefine instruction is used to prevent further substitutions using a macro. The following code fragment is an example of the use of undefine:

```
undefine([pushb])
```

Note that quote characters are used around the macro name to ensure that the macro is not expanded by the macro processor.

6.2 Additional Reading

Further information relating to the **m4** macro processor can be found in Kernighan, Brian W., Ritchie, Dennis M., *The M4 Macro Processor*, Bell Laboratories.

Chapter 7

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.

7.1 Addressing Modes

7.1.1 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.

7.1.2 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 7.1). 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:

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

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

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

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. If the scale is not included then a scale of 1 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

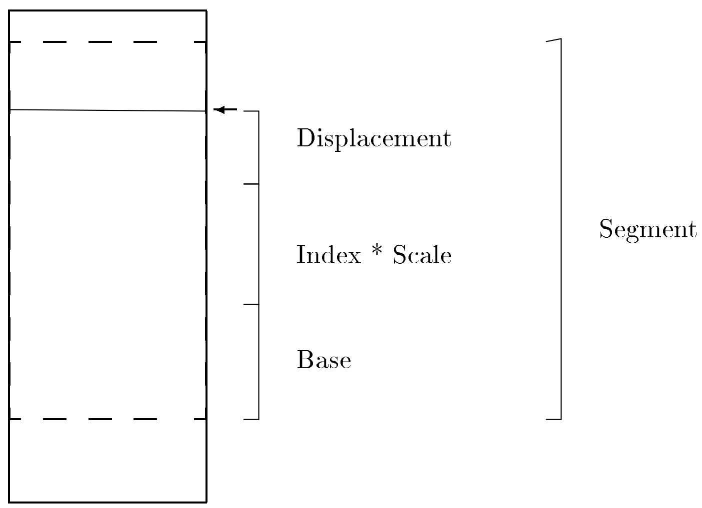


Figure 7.1: Indexed Addressing

7.1.3 Indirect Addressing

Indirect addressing consists 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.

This is illustrated in figure 7.2.

Address	*Memory*	
1000		
1004	2008	→
1008		
1012		
2000		
2004		
2008	45	←
2012		

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 7.2: Indirect Addressing

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.

7.2 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.

7.2.1 '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 8

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.

8.1 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.

8.1.1 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.

8.1.2 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:

%esp+12		
%esp+8	1	
%esp+4	2	
%esp+0	ret addr	$\leftarrow$ %esp

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.

%esp+14			
%esp+10	1		4 bytes
%esp+8	2		2 bytes
%esp+4	3		4 bytes
%esp+0	ret addr	← %esp	4 bytes

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.

8.1.3 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 addrs 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.

8.1.4 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.

8.1.5 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:

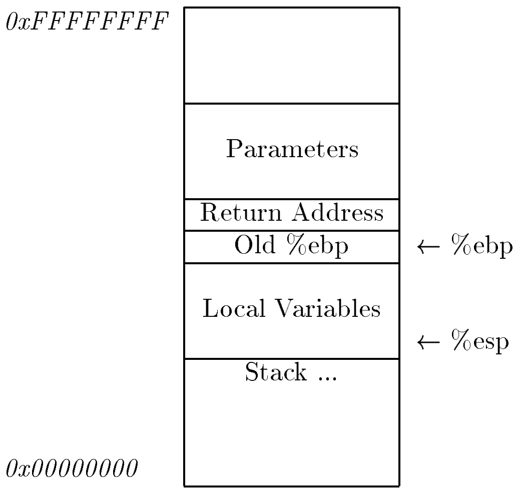
```
fib:
     subl $4, %esp            /* reserve space for c */
     movl 12(%esp), %eax      /* recover the a parameter */
     call print_num           /* call fake print routine */
     movl 8(%esp), %ebx       /* recover the b parameter */
     movl %eax, 0(%esp)       /* store a in c */
     addl %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 fake 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 fake 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 9

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.

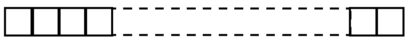
9.1 Vectors

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 9.1.

The vector was introduced in the section on indexed addressing (7.1.2). 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.

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

$\downarrow a$ $b \downarrow$



$\uparrow base$

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

Figure 9.1: 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
```

9.2 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

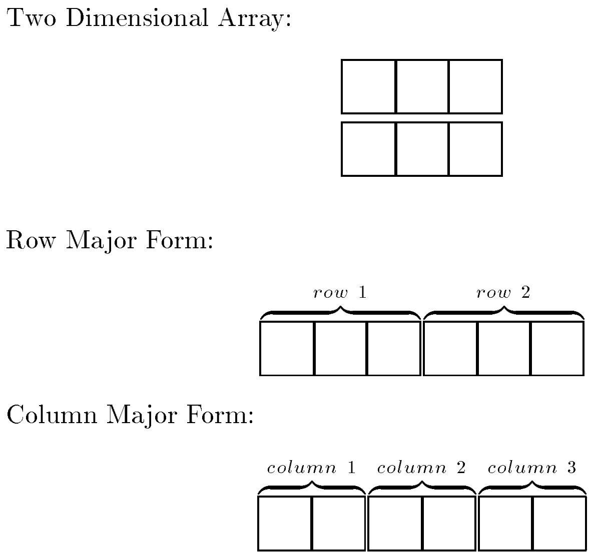


Figure 9.2: Major Forms

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 9.2 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:

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

```
/* 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
```

9.3 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 7.

9.4 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 9.3).

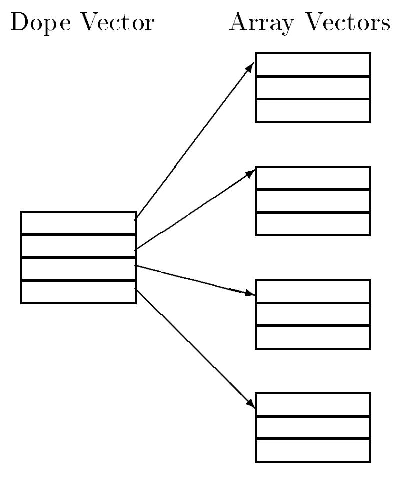


Figure 9.3: A dope vector

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
```

9.5 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.

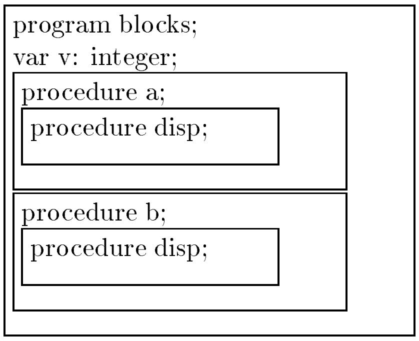
Chapter 10

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 assembly language by providing backward links in the stack frame to earlier stack frames. The **enter** instruction's second parameter, **level**, determines the number of stack frame pointers that are inserted into the current stack frame to the previous stack frame. Figure 10.1 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.

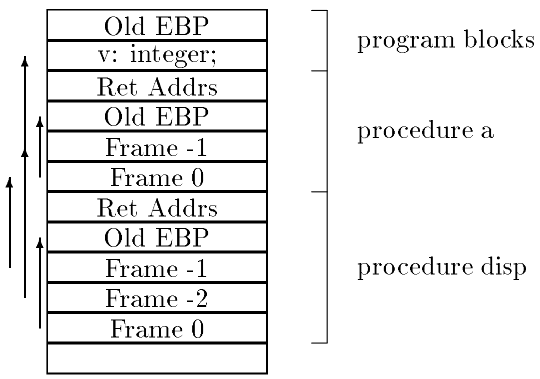


Figure 10.1: Stack Frames in a Block Structured Language

Chapter 11

Floating Point

In previous chapters we have covered the representation and operations required for the manipulation of signed and unsigned integers. Although a large class of mathematical problems can be solved directly using integers there is a larger class of problems that are best performed using fractional representation. In this chapter two new representations will be introduced: fixed point and floating point.

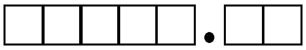
11.1 Decimal Representation

Before developing the binary scheme used in computers to represent numbers the parallel concepts in the familiar decimal notation will be covered. A decimal fixed point numbering scheme is frequently used for the handling of money. For example a monetary system may be based on the cent and a dollar is composed of one hundred cents. Thus we perform all our operations in cents and scale the result to report the result in dollars.

$$100\ cents\ =\ 1\ dollar$$

$$15\ cents\ =\ 0.15\ dollars$$

The fixed point for our dollar representation is between the second and third columns from the right.



A fixed point scheme performs well when the problem domain is based on an indivisible quantity. In the case of our example system it is not normally useful to speak about fractions of a cent.

Scientific notation is closely related to floating point arithmetic. A number written in scientific notation consists of a number multiplied by some power of 10. For example:

$$15 * 10^6 = 15000000$$

$$15 * 10^{-2} = .15$$

Scientific notation allows the representation of a wide range of values compactly.

11.2 The Decimal / Binary Point

In decimal notation the fractional component of the number is written beyond the decimal point. The columns beyond the decimal point represent fractions of powers of ten decreasing in size. For example:

$$21.23456_{10}$$

may be written as

$$2 * 10 + 1 * 1 + 2 * \frac{1}{10} + 3 * \frac{1}{100} + 4 * \frac{1}{1000} + 5 * \frac{1}{10000} + 6 * \frac{1}{100000}$$

or

$$2 * 10^1 + 1 * 10^0 + 2 * 10^{-1} + 3 * 10^{-2} + 4 * 10^{-3} + 5 * 10^{-4} + 6 * 10^{-5}$$

A similar representation is available for binary numbers. In the case of a binary number, values beyond the binary point represent decreasing fractions of powers of two. For example:

$$10.0110101_2$$

may be written as

$$1*2+0*1+0*\frac{1}{2}+1*\frac{1}{4}+1*\frac{1}{8}+0*\frac{1}{16}+1*\frac{1}{32}+0*\frac{1}{64}+1*\frac{1}{128}$$

or

$$1*2^1+0*2^0+0*2^{-1}+1*2^{-2}+1*2^{-3}+0*2^{-4}+1*2^{-5}+0*2^{-6}+1*2^{-7}$$

11.3 Normal Numbers

A number stored in normal form has a value in the position before the point other than zero, but no values in any more significant positions. The following are normal decimal numbers:

$$1.3_{10}$$

$$1.05*10^2{}_{10}$$

$$1.4*10^{-2}{}_{10}$$

All normalized binary numbers start with a 1 followed by the binary point.

$$1.11001_2$$

Normalizing numbers ensures that the number is both uniquely represented and represented in the most accurate form possible within the representation.

11.4 Floating Point Binary Numbers

A floating point binary number consists of several parts:

s	exponent field	significand field

where

Sign Bit A bit used to indicate the sign of the number. If the sign bit is clear the number is positive otherwise the number is negative. (S)

Exponent Field A number. The number may be biased. The value of a biased number is given by $value = biased\ number - bias$. (E)

Significand Field / Mantissa Field An unsigned normalized number. (F)

In general the value of a floating point number may be calculated using the formula:

$$-1^S * F * 2^E$$

The following examples have an 7 bit significand a 4 bit exponent with a bias of 8 and a single sign bit. The format of the number is the sign bit followed by the exponent field followed by the significand.

S_2	E_2	F_2	S	E_{10}	F_{10}	$value_{10}$
0	1000	1000000	+	0	1.0	1
1	1000	1000000	-	0	1.0	-1
0	1000	1100000	+	0	1.5	1.5
1	0111	1110000	-	-1	1.75	-0.875
0	1001	1110000	+	1	1.75	3.5

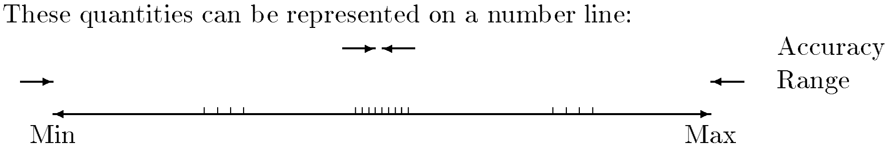
11.5 Range and Precision

Range and precision are two parameters which describe a number representation.

- The range of a representation is defined as the greatest and least value that can be represented. Ranges are typically symmetric about zero.

- The precision of a representation is defined by the size of the steps between adjacent values.

These quantities can be represented on a number line:



Typically the range of a representation is specified in terms of the maximum and minimum values represented by the exponent. In addition, the precision is often quoted as the number of bits required to state the number represented by the significand.

If a number is stored in a normalized form it is possible to exploit the property that it has a leading 1 by dropping the leading leading 1 and simply assuming that it is present. This effectively yields an extra bit of precision.

11.6 Properties of Non-Integer Numbers

The fixed point and floating point representations of numbers allow the representation of non-integer values. Each representation has differing properties which affect its usefulness for a given task.

A fixed point representation:

- Has a reduced range of values in comparison to an integer of equivalent size. This is due to the fixed point value being effectively a scaled integer value.

- All combinations of bits represent a value of a fixed point number. Hence there are no wasted bit combinations.

- The accuracy of a value is independent of its magnitude.

A floating point representation:

- Typically has an increased range of values in comparison to an integer of equivalent size.
- Not all bit combinations represent different numbers. For example there are a large number of bit combinations which can be used to represent zero.
- The accuracy of a value is dependent on the magnitude of the number. This is due to the uneven spread of values on the number line caused by steps in the exponent value doubling the separation between adjacent values.

In addition to these features, a floating point representation can include additional values known as **NaN**s or *Not a Number* and positive and negative infinity. These values are not ordinary numbers and are used as results when an exception would have to be raised if the NaNs were not available.

11.7 Overflow and Underflow

There are two types of errors which are particularly significant in floating point calculations: Overflow and Underflow.

Overflow The result of a calculation cannot be represented because the calculated exponent is too large to be accommodated in the exponent field.

Underflow The result of a calculation cannot be represented because the calculated exponent is too small to be accommodated in the exponent field.

11.8 Algorithms For Basic Operations

Addition and multiplication are basic operations required over numbers. Fixed point addition is performed in the same way integer addition is carried out. Fixed point multiplication is the same as integer multiplication, with exception that the result is rescaled after the multiplication.

The algorithms for addition and multiplication over floating point numbers are more complicated. Figure 11.1 and 11.2[1] illustrate algorithms for performing these operations.

11.9 IEEE 754

The IEEE 754 Standard for binary floating point arithmetic describes a set of formats and operations for floating point numbers. It should be noted that there are both required and optional features specified in the standard and that most of the current computers comply at least partially with the standard.

A short summary of the major features of the standard is provided here.

	Single	Single Extd‡	Double	Double Extd‡
Precision†	24	≥ 32	53	≥ 64
E_{max}	127	≥ 1023	1023	≥ 16383
E_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent Bias	127		1023	

†Includes assumed bit for normalized numbers.

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

‡Extended.

The form of the floating point number is:

s	exponent field	significand field

A convention is required to store the value zero. This is due to the assumed leading bit for normalized numbers. The convention for representing zero is for the exponent field and the significand to be set to zero. In addition to this there are a number of classes of special

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

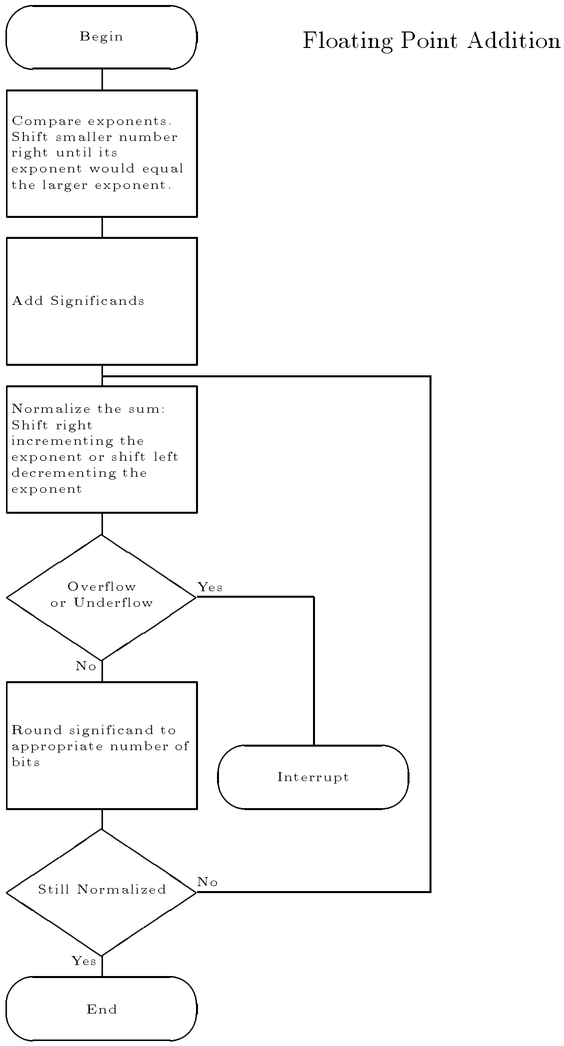


Figure 11.1: Floating Point Addition

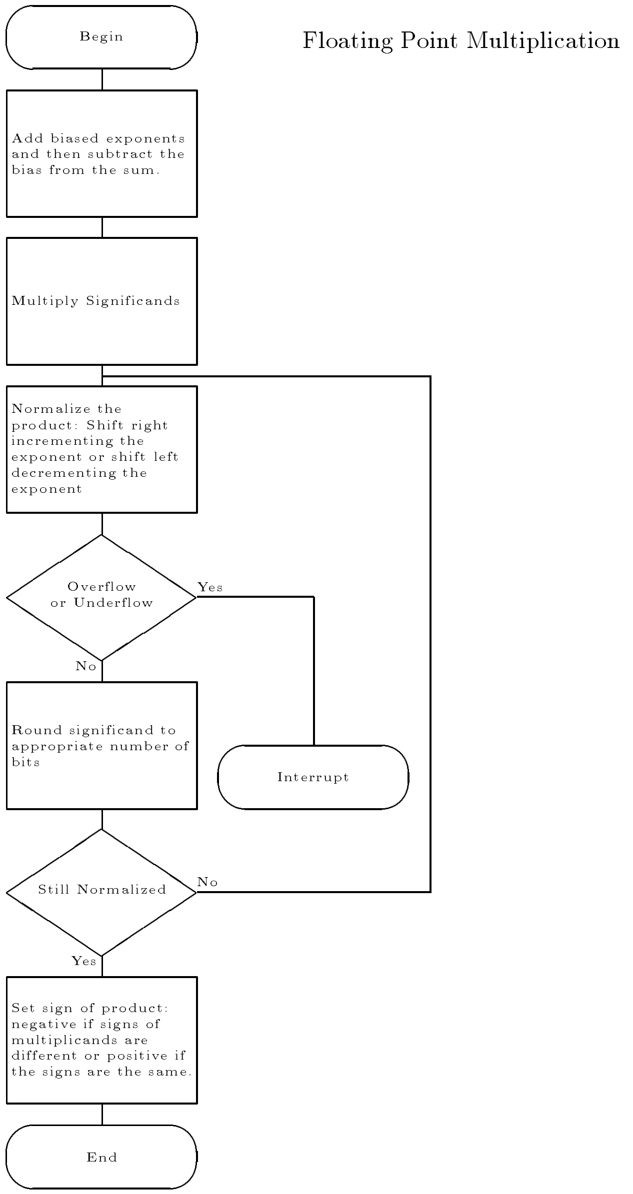


Figure 11.2: Floating Point Multiplication

values that are represented by setting the exponent field to zero or to all ones. Included in these values are NaNs, positive and negative infinity and denormal numbers. A denormal number (sometimes known as a subnormal number) is represented as a number with a zero exponent field and a non-zero significand. These numbers are used to represent numbers which are smaller than the smallest representable normalized number.

11.10 Additional Reading

Further information relating to floating point arithmetic can be found in **section 4.8** of Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

Chapter 12

The Processor

12.1 Data Paths

Instead of studying the detail of the 80386 processor's data paths, the simpler data paths of the 8086 will be covered. This simpler processor allows the general concepts of a microprocessor architecture to be discussed without the need to cover the detail of the more complex processor. This material will parallel the material applied to a simplified MIPS processor found in **chapter 5** of Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

12.1.1 Overview

The diagram in figure 12.1 illustrates the key data paths of the 8086 processor. Each component of the processor will be discussed and the interactions of the components will be identified.

Components

ALU The arithmetic logic unit (ALU) of the processor performs arithmetic and logical functions. The ALU has two bus inputs which are

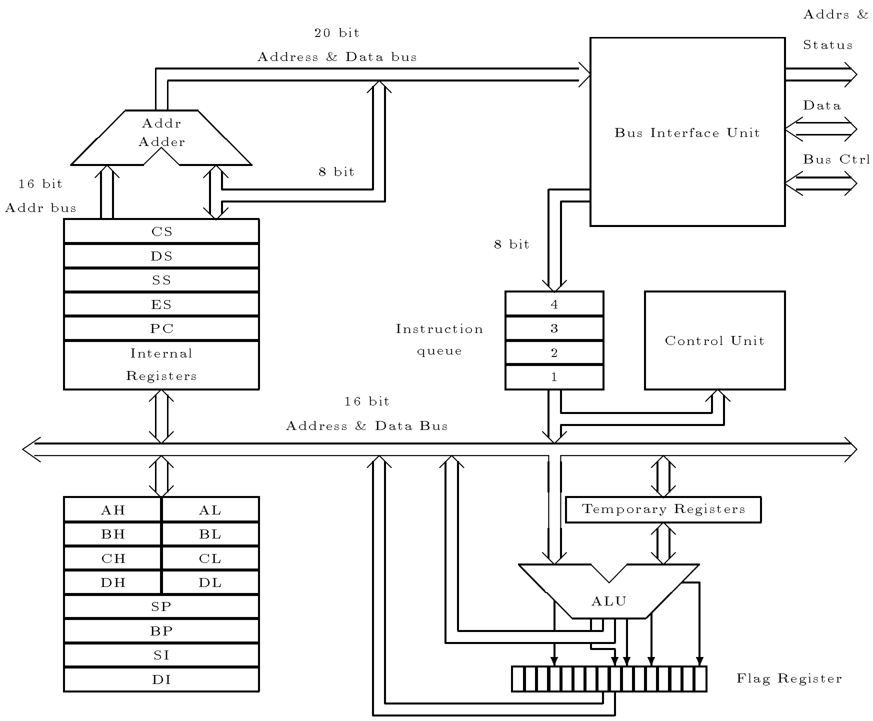


Figure 12.1: Block Diagram of the 8086 processor

combined to produce an output. The function provided by the ALU is determined by its control inputs. In addition to performing either an arithmetic or logical function the ALU sets bits in the status register to indicate features of the result.

Register File The 8086 block diagram shows two logically distinct register files. A register file is a collection of registers. The first of these is a register file containing the general registers, stack pointer, base pointer, and the index registers. The second register file contains the segment registers and the program counter.

Control Unit The control unit decodes the instruction stream and co-ordinates the activities of the processor.

Buses A bus is a collection of conductors. Note that a conventional bus may only be 'driven' by one device at a time, although many devices may observe the state of the bus. A device is said to 'drive the bus' if it is a source of current for the bus. To observe the value of the bus it is necessary to sink or consume some current from the bus.

Other Elements

Instruction Queue The instruction queue buffers four bytes of instructions.

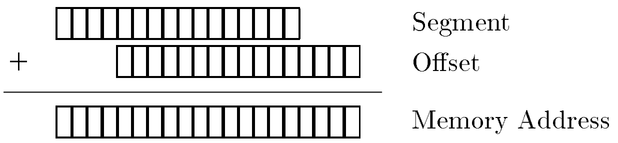
Temporary Registers These registers are used to hold data items to be fed to the ALU.

Address Adder This is used to form the 20 bit address used by the 8086 to access memory.

Bus Interface Unit The bus interface unit fetches data and code from memory. The bus interface unit also monitors the interrupt and other control lines.

12.1.2 Addressing

The 8086 supports a 20 bit address space by combining a 16 bit address with a 16 bit segment register. The 16 bit segment register is left shifted by 4 bits and is added to the 16 bit address to yield a 20 bit address.



This operation is required each time an address is to be output to the bus.

12.1.3 Fetch-Execute Cycle

The basic operation of the 8086 is dominated by a single sequence of operations. This sequence is known as the fetch and execute cycle. The following descriptions give the flavor of the operations and how they might be implemented. Note that this is not a precise description of the operation of the 8086 chip. The invariant part of this cycle over all instructions is:

1. Output PC to Address Adder

   Output CS to Address Adder

2. Output 20 bit address to system bus

   Increment PC

3. Store returned bytes in Instruction Queue

4. Decode first byte of instruction

The remaining steps of the cycle vary with the instruction. If the instruction is a multi-byte instruction then further bytes are read from the instruction queue.

Two example the sequences will be illustrated: incrementing a register and adding a register and an integer will be covered. These examples exercise most of the data paths through the processor.

Increment AX:

5. Output enable AX onto the 16 bit Address & Data Bus

   Input enable the ALU

   Select increment funcion

6. Input enable AX onto the 16 bit Address & Data Bus

Add AX and location 1004 leaving the result in AX:

5. Output enable AX onto the 16 bit Address & Data Bus

   Input enable a Temporary Register

6. Output 1004 to Address Adder

   Output DS to Address Adder

7. Output 20 bit address to system bus

8. Store returned bytes in Instruction Queue

9. Output enable Instruction Queue

   Input enable ALU

   Select add funcion

10. Input enable AX onto the 16 bit Address & Data Bus

12.2 Control

There are two classes of design for the circuits that control the operation of a processor or the control unit. The historically earlier 'hard-wired control units' were replaced by 'microprogrammed control units' during the 1970s and 1980s. With the advent of the RISC philosophy and the demand for higher performance computers hard-wired control units and hybrid control units have regained popularity.

12.2.1 Microprogrammed Control Units

In a microprogrammed control unit an instruction is represented as a series of microinstructions. Each microinstruction consists of several fields. These fields control the assertion of control signals and the selection of the next microinstruction. A hardware register called the μPC is used to point to the next microinstruction to be executed.

	Control Field	Sequence Field
000	1000000	001 000 000
001	0100000	010 000 000
010	0100000	000 010 000

Figure 12.2: An imaginary microinstruction format

The simplest form of a microprogrammed control unit (figure 12.2) would operate as follows: The first microinstruction would assert a control line to cause the PC to be output. Subsequent microinstructions would load the contents of the appropriate memory location to be loaded into the microinstruction decode register. This value would be decoded and cause the microprogram counter to be set. Upon completion of the sequence a microprogram instruction would cause the μPC to be set to the begining of the fetch sequence.

Decisions are made in the microprogram by selecting which microinstruction to go to next. This is done either by following on to the next instruction or by skipping to another instruction based on an input line to the control unit.

Vertical & Horizontal Microcode

Microprogrammed control units may be broken up into 2 classes:

Horizontal control units These resemble the simple control units in that the microinstruction is wide and contains a bit for each control line

Vertical control units Are narrower than horizontal control units as the instruction is divided into fields which are either decoded or translated into the control signals.

12.2.2 Hard-wired Control Units

Hard-wired control units perform a similar function to microprogrammed control units. Both types of control units output control signals in a sequence to cause an action. The difference between the two forms of control unit is found in the implementation. A hard-wired control unit uses logic gates to generate the control signals.

12.3 Interrupts, Exceptions and Traps

This section is concerned with the interface between external events and exceptional internal events and the flow of control within a processor. The presence of interrupts both eases the task of the programmer by allowing the programmer to ignore exceptional or external events within the program and complicates the programmer's task by requiring exceptional events to be handled in a transparent way.

12.3.1 Key Definitions

Interrupt Interrupts are a form of forced procedure call which are caused by an event external to the program. Interrupts are *asynchronous* to the program. Interrupts are typically caused by a peripheral asserting a wire connected via some circuitry to an interrupt pin on the processor.

Trap A trap is a form of forced procedure call which is caused by an exceptional event within a program that has been detected by the processor hardware. Traps are *synchronous* to the program. An example of an event that might cause a trap would be arithmetic overflow.

Transparency Transparent code stores the necessary process state and register values to ensure that when it returns the interrupted code is unable to determine that an interruption has occurred

Critical Section A critical section is a section of code which must be executed atomically.

12.3.2 Implementation

The typical implementation for a microprocessor based interrupt mechanism is for the processor to check for an external interrupt condition at the end of or the beginning of an sequence that implements an instruction.

In a microprogrammed control unit this would correspond to a branch to the microcode to implement interrupts at the end of the sequence of microinstructions that implements an instruction if an interrupt signal is present.

This mechanism does not require the processor to restart an instruction part way through.

Interrupts are implemented as a forced procedure call. This means that after the interrupt condition is noted at least the following actions are required: a return address is stored and the interrupt routine called.

There are three major implementation forms:

- Mechanisms which use an *interrupt vector*
- Mechanisms which use *vectored interrupts*
- Mechanisms which use a status register

These mechanisms differ in the mechanism in which they convey to the *Interrupt Handler* the cause of the interrupt.

Interrupt Vectors

The destination addresses for each type of interrupt are stored in a table. When an interrupt occurs the processor makes a forced procedure call to the location indicated in the interrupt vector.

Vectored Interrupts

In a vectored interrupt system the location called by the processor is determined by the cause of the interrupt. Typically this is implemented as a set of addresses a fixed distance apart which the processor calls when an interrupt occurs.

Status Register

In this case a processor jumps to a single address when any interrupt occurs. It is the task of the interrupt handler to consult the status register to determine the cause of the interrupt. This register is called the *cause* register in the MIPS architecture.

Two Examples - 80386 & R4000

80386 The 80386 uses an interrupt vector with 256 entries. When an interrupt occurs:

- push EFLAGS onto the stack
- push Instruction pointer onto the stack
- clear interrupt flag
- the processor jumps to the location indicated by the interrupt type

R4000 The R4000 uses a single entry point for interrupts. The exact memory location of the interrupt entry point depends on the operating mode of the processor.

When an interrupt occurs:

- the EPC (exception program counter) is loaded with the current program counter value
- the bit in the Cause register corresponding to the interrupt value is set
- the processor jumps to the interrupt handler

12.3.3 Masks & Priorities

As an interrupt can effectively occur at any time within a program's execution they are essentially unpredictable. At times it is inconvenient to have to handle an interrupt. An interrupt mask is used to prevent an interrupt having an effect.

An interrupt mask is a set of bits which - if set - allow the interrupt to be noticed by the processor.

Under what circumstances is it necessary to not notice an interrupt immediately?

- While executing a time critical routine
- During the initial phase of handling another interrupt
- When handling a more important activity

The first two of these problems are solved by setting the mask to prevent interruption while performing critical tasks. The second is solved by introducing priorities.

In a priority based interrupt scheme each interrupt is allocated a priority.

- If the currently executing interrupt handler has a lower priority than a new interrupt then the currently executing interrupt handler is pre-empted and the new one commenced.
- If the currently executing interrupt handler has a higher priority than a new interrupt then the new interrupt is recorded and dealt with when all higher priority interrupt handlers have completed.

12.3.4 Non-maskable Interrupts

Processors frequently have an interrupt that cannot be masked out or ignored. This is typically used to indicate a failure in a critical component of the computer. A classic example would be a power failure detection. If the power fails the system should attempt to perform necessary house keeping before function is fully lost.

Only critical functions should use the NMI as there is no mechanism to ensure the correct return from interrupt after an NMI.

12.3.5 Real time systems

A real time system needs to respond in a predictable way to any set of input conditions. As noted in the section on masks a problem arises when an interrupt pre-empts an interrupt in a stack based system. This is caused by the requirement for saving state on the stack. Some real time systems solve this problem by having an area of storage assigned to each interrupt routine and storing the state in that region when an interrupt occurs. This ensures that there is always a place for the state to be stored that is rapidly accessible. In extremely critical applications hardware assistance is provided for saving the state.

12.3.6 Interrupt Service Routines

An Interrupt Service Routine (ISR) performs the following operations

1. Save all registers
2. Save any status information relating to the cause of the interrupt
3. Enable higher priority interrupts
4. Perform required service
5. Clear the cause of the current interrupt
6. Return from interrupt

The first two tasks fall in a *critical section*. By ensuring that this information is correctly saved it is possible to be interrupted and returned to *transparently*.

The mechanism described is sufficiently general and conservative to be used on both normal and real time systems. In a normal system the data in 1 and 2 would be stored upon the stack. In a real time system the information would be stored in a location local to the ISR.

A sample 80386 ISR

This routine handles the keyboard interrupt handler of a IBM-PC running in protected mode. This is a complete assembly language routine for performing the task. It should be noted that this routine deals with a number of issues specific to the IBM-PC architecture:

- The need to alter the segment registers for protected mode operation.
- The interaction with the keyboard controller.
- The need to reset the interrupt controller.

```
int33:
      /* interrupts are already off */
      /* store state */
      pushal
      push %ds
      push %es
      /* switch to privileged mode */
      mov $privds, %ax
      mov %ax, %ds
      mov %ax, %es
      /* store cause of interrupt information */
      /* scan the key */
      inb $0x60, %al    /* get char */
      pushl %eax        /* push key data onto stack */
      inb $0x61, %al    /* get control register */
      movw %ax, %cx
      orb 0x80, %al     /* strobe bit high */
      outb %al, $0x61
      movw %cx, %ax     /* strobe bit low */
      outb %al, $0x61
      /* re-enable interrupts now critical work done */
      sti
      /* call ‘C’ routine to handle remainder of processing */
```

```
call _keyhandler
addl $4, %esp     /* remove parameter */
/* reset interrupt controller chip */
mov $0x20, %eax   /* eoi master */
outb %al, $0x20
/* restore state and complete interrupt */
pop %es
pop %ds
popal
iret
```

12.4 Additional Reading

Further information relating to exceptions can be found in **chapter 5** of Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

Chapter 13

Input/Output

The I/O systems of a computer are critical to the performance of a computer. Relatively few jobs are completely CPU bound thus nearly all jobs depend on the performance of the I/O system to account for some fraction of their total performance. In this chapter we will look at the general characteristics of input output devices. For detailed tabulations of the characteristics of individual devices the reader is referred to the contents of the additional reading section.

13.1 Types of Devices

There are two major classes of devices:

- Block Mode
- Stream Mode

Block Mode Transfers are made in blocks. A block is defined as a regular structure with some maximum size and some minimum size. Typically block mode devices employ blocks of exactly the same size. If error detection is employed then processing of the information within a block cannot commence until the full block is available. Examples of block mode devices: Disk drives & Network Controllers.

Stream Mode This is a generalization of character mode. Information is transferred in typically byte sized quantities and each item is processable immediately on reception. Examples of stream mode devices: Terminals & Serial devices.

Properties of Devices

Block mode devices place a lower load on the system resources on a per byte basis. This is because the system needs only to notice and handle a transfer on the completion of a block transfer. In contrast, a stream mode device requires the system to intervene with the arrival of each new item.

Block mode devices tend to have better error correction than stream mode devices as stronger error detection and correction techniques can be applied.

Software and hardware can convert the behaviour of a stream mode input to partly resemble block mode input or vice versa. Although this conversion is possible the properties of the underlying device cannot be completely masked. In many cases attempting to a hide the mode of access results in undesirable behaviour.

13.2 Interrupts & Polling

There are two major mechanisms available to the programmer for interacting with external devices:

- Interrupts
- Polling

13.2.1 Interrupts

Interrupts an external event causes an interrupt the ISR deals with the device.

Each time an interrupt occurs the processor performs the sequence of operations described in section 12.3. This sequence of events is known as the overhead of the interrupt as it occurs each time an interrupt occurs and does not directly contribute to the handling of the event.

13.2.2 Polling

Polling The processor regularly checks each device and notes the state of the device. If the device requires servicing the required operations are carried out

For polling to work it is necessary to ensure that the *polling loop* can be completed sufficiently quickly that the fastest device will not have made more than one request in the time it takes to go round the loop handling all possible requests. Failure to do this leaves the system vulnerable to data overruns.

13.2.3 Comparison

Polling has a lower overhead than interrupts as the polling loop does not need to perform the actions of an interrupt handler or the actions forced by an interrupt. This indicates that a polled system can get greater through put than an interrupt based system. Polling, however, requires that system be dedicated to carrying out the polling loop and that actions take known maximum times. This makes a polled system inflexible. The requirement that the system be designed for the worst case implies that in ordinary usage it is likely that there will be wasted system resources.

13.3 DMA & Co-processors

A system architect may choose to reduce the load associated with a device by off laying part of the work onto a device external to the processor but capable of accessing part or all of the system's memory.

13.3.1 DMA

A DMA or Direct Memory Access device is essentially a simple processor attached to the memory bus of the system. DMA devices may be used to accomplish several tasks:

- Memory to Memory - a DMA device can copy a block of memory from one point in the system memory to another with minimal processor intervention.

- Device to Memory - A DMA device can be used to interact with a device to copy the results returned by the device into the systems memory. Both block mode and stream mode devices may be interfaced this way.

As the DMA device shares the bus with the processor the action of the DMA device affects the ability of the processor to interact with memory. The DMA device and the processor arbitrate for the memory bus. The effect of this is that neither processor nor DMA device can have unrestricted use of the bus, but due to the burst nature of bus traffic the delay introduced by the sharing of the bus is not proportional to the usage of the bus.

The design and parameters of DMA devices vary from unit to unit, however, there are common features to the designs.

The common operations to DMA controllers are the need to setup the device to transfer and to inform the system that the transfer is complete. The information required to setup the transfer is the source of the data, the destination address and the size of the transfer. At the completion of a DMA operation the processor is notified by an interrupt raised by the DMA controller.

13.3.2 Co-processors

DMA devices are a special restricted form of co-processor. High performance I/O devices may have significant autonomy. In this case the device carries out a sequence of operations based on a sequence of high level instructions passed from the main processor. Examples of this type of device are graphics co-processors which accept lists of operations to carry out and intelligent serial interfaces which poll a large number of serial ports and make available the ports information as blocks of characters.

13.4 Architectural Consequences

The choice of the mechanism for handling I/O devices depends on a number of factors:

- Mode of device
- Data rate
- Tolerable latency
- Available hardware

Where large amounts of data are to be transferred or a high data rate is required either a block mode device or a DMA controller should be provided. This reduces the load on the processor by ensuring that the processor deals with large lumps of information. The alternative of having to keep up with a large number of interrupts or a tight polling loop would have a detrimental effect on the performance of the machine.

Where the volume of data is small or the data rate is low then the processor can in general be used to handle the type of operation using interrupts or polling started by a timer interrupt (A regular timer tick interrupt occurs and the system polls the required devices).

If fast response to asynchronous inputs is required - low latency - then interrupts should be used.

Frequently the type of access mechanism is determined by the existing hardware and the improvement of performance of using additional hardware is traded for the reduced cost of using existing - less optimal - hardware.

13.5 Additional Reading

Further information relating to input output devices can be found in **chapter 8** of Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

Chapter 14

Memory

A major constraint on system performance is the rate at which data can be accessed by the processor. The majority of the data used by the processor is stored in the memory of the computer, hence the significance of memory performance to overall system performance.

This chapter will discuss the principles and implementation of the memory hierarchy.

14.1 Principles

Temporal Locality The principle of temporal locality states that if an item has been accessed recently it is likely to be accessed in the near future.

Spatial Locality The principle of spatial locality states that if an item has been accessed it is likely that items close to it will be accessed in the near future.

14.2 Caching

Before applying the principles of temporal and spatial locality it is necessary to observe that in general faster memory is more expensive than slower memory. Using the principle of locality and organizing that recently accessed items and adjacent items are moved to fast memory it is possible to generate a memory system which has costs approaching that of the slower memory but with performance approaching the faster memory.

This is known as caching information.

14.2.1 Operation

Memory is accessed by naming an address and the memory unit or processor transferring the contents of the location on the bus. A cache unit sits between the processor. There are two distinct operations that the cache deals with: processor reading memory and processor writing to memory.

If the processor is reading from memory: If the cache knows the contents of the memory location then the cache returns the contents of its copy of the memory location's contents to the processor. If the cache does not know the contents of the memory location then the cache requests the memory to return the content of that location. When the result arrives the cache stores the value in the cache and passes it through to the processor.

If the processor is writing to memory: The cache stores a copy of the location's contents and subsequently passes the value through to memory (not necessarily immediately)

There are two major variants of caches available which are distinguished by their behaviour on writes:

Write Through The data is written into the cache and into the next stage of the memory hierarchy simultaneously.

Write Back The data is written into the cache only. Only when a cache flush is issued is the data written to the next stage of the memory hierarchy.

14.2.2 Implementation

There are several implementations of caches one mechanism will be covered here as an illustration. Figure 14.1 illustrates a simple scheme. In the scheme presented part of the address is used as an index into the cache's table. If the tag and the upper component of the address match and the cache entry is valid then a cache hit is said to have occurred and the value is returned to the processor or updated in the cache depending on whether the operation is a read or a write.

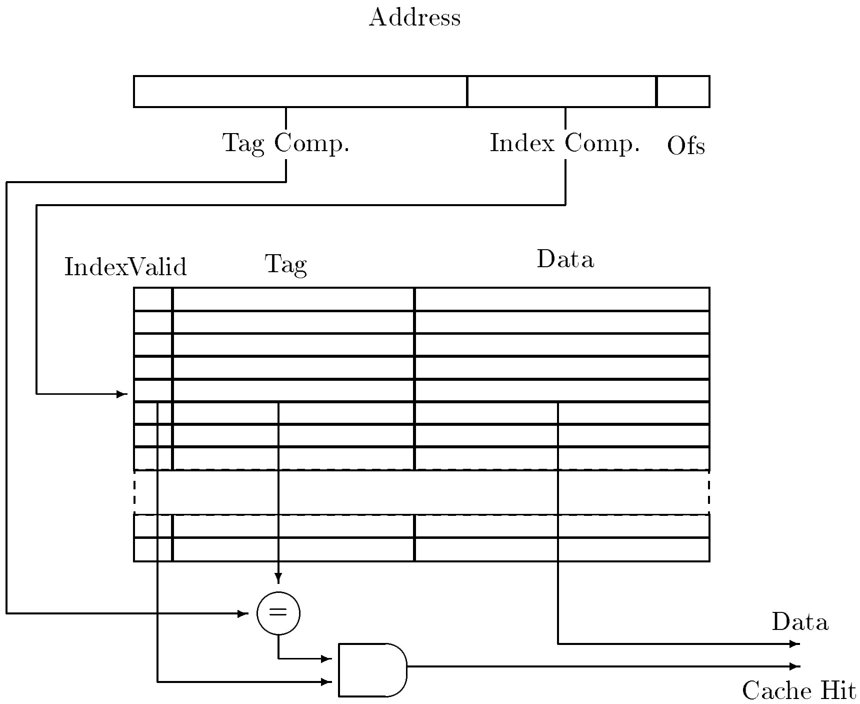


Figure 14.1: Simple caching scheme

14.3 Virtual Memory

Modern processors have an address space which is far larger than their physical memory. Virtual memory is a mechanism which allows the address space to be used and for the computer to have individual programs which are greater than the total size of the computer memory.

There are two types of virtual memory implementations:

Paging Under paging the memory is divided into equal sized objects know as pages which are paged into and out of the physical memory to disk.

Segmentation Under segmentation objects called segments of a size selected by the programmer are swapped into and out of the memory of the computer to disk.

Both mechanisms employ an additional layer of translation to convert an uttered address into a physical address.

A program larger than the physical memory is made possible as when a program utters either the name of a segment not in memory or the address of a page not in memory a page or segment can be copied to disk and the required page or segment brought in.

14.3.1 Paging

In a paged system a page table is used to translate each address uttered by a program to a physical address. Essentially part of the logical address is used to index into the page table to recover the physical address at which the page was loaded. The least significant part of the logical address is added to the address where the page was loaded to yield the sought after location.

The description above describes the page table as a single table. In practice it is usual to have multiple levels of page tables. In the most popular form - two level paging - there is a top level page table which points to the physical locations of the second level page tables. The address is divided into three parts the most significant component indexes the top level page table. The middle component indexes the second level page tables. The least significant component is added to the address recovered from the second level page table. The advantage of this scheme is that it allows the page table itself to be paged. This has two advantages unused components of the page table do not need to exist reducing the size of the page table and allowing processes to co-exist in memory, by swapping top level page tables the address space can be switched quickly between processes.

14.3.2 Segmentation

In a segmented system all addresses consist of a segment and an offset from the segment. When a loaded segment is accessed the offset is added to the location where the segment was loaded to access the required location.

14.3.3 Fragmentation

Fragmentation occurs when there is unused or unusable space in the memory of the system. Both paging and segmentation suffer form fragmentation. Paging suffers from internal fragmentation - the lost space is contained in unused parts of a loaded element. Segmentation suffers from external fragmentation - it is not necessarily possible to use all the memory of the machine as the uneven sized segments cannot be divided to ensure that all the space is filled.

External fragmentation is one of the causes for segmentation's unpopularity as a memory management technique. Currently Intel is the only manufacturer bringing out new microprocessors that support segmentation. The management of external fragmentation adds complexity to an operating system and hence the move to support paging.

14.3.4 Memory Protection

Both paging and segmentation allow access rights to be attached to the page or segment. Typically these access rights specify that the data contained may be accessed by:

- read
- write
- execute
- privileged code only

14.3.5 Making it efficient

Due to the principal of locality it can be seen that by holding recently used items in memory and removing items which have not been accessed recently from memory it is possible to create a system which has speeds approaching the faster memory in the system but at the cost closer to that of the slower recording medium.

14.4 The Memory Hierarchy

The fastest memory devices present in the system reside in the processor directly coupled to the internal data bus of the processor - the registers. Next in speed is on chip memory. Many modern processors have small primary caches on the processor chip (in the order of 8K bytes size, $< 10nS$). Off chip caches are next in speed and considerably larger (in the order of 100K bytes to 4M bytes size, $15nS$). Main memory (in the order of 10M bytes, $< 80nS$) follows. Finally hard disks or secondary storage (in the order of 1G bytes) is the slowest.

14.5 Additional Reading

Further information relating to memory can be found in **chapter 7** of Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

Chapter 15

Advanced Topics

This chapter covers:

- Pipelined processors
- Superscalar processors
- The RISC / CISC controversy

The first two topics introduce general techniques for improving the performance of microprocessors beyond the limits of a simple "von Neumann" architecture. Both techniques introduce parallel operations into a processor.

The final topic explores the concepts behind RISC processors and contrasts this with CISC processors. This topic is of particular relevance to students as the section attempts to identify a number of fallacies perpetuated by both the RISC and CISC manufacturers in advertising and the information they supply to users.

15.1 Pipelining Processors

Pipelining is a technique for overlapping the execution of instructions. In a pipelined machine the next instruction starts executing before the previous instruction has completed executing. In the chapter on the internals of a processor it was observed that each instruction was made up of several components and frequently those components were common to all instructions. Pipelining exploits this commonality by making each common component a step in the pipeline. Thus in a single step of the pipeline when a component of an instruction completes executing it is passed on to the next phase of the pipeline and the component of the next instruction is admitted for processing.

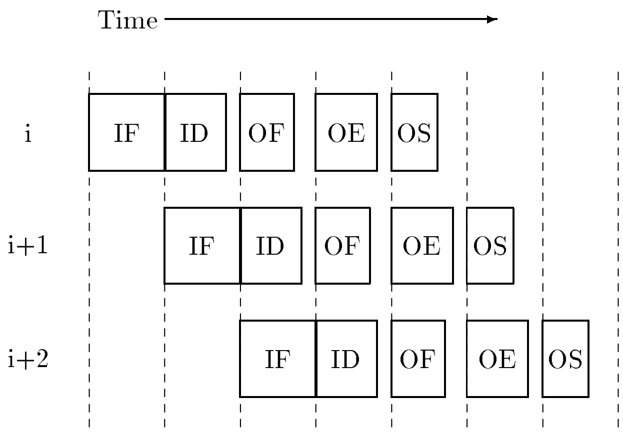
The technique of pipelining improves processors in the same way Henry Ford revolutionised car manufacture. Ford altered the face of car assembly by introducing the production line. Prior to Henry Ford each car was hand crafted by a group of skilled craftsmen. In a production line a group of workers are responsible for a single stage of the total process and then pass the product on to the next stage of the production line. Similarly, in a pipelined microprocessor, each component handles its part of the execution and then hands the job onto the next stage of the pipeline.

The simile of pipelining and a production can be extended to cover the critical design elements of a pipeline. The rate of production is limited by the slowest step in production. If a given step takes longer than the other steps in the process then time is wasted in the other steps. The aim of making all steps of the pipeline take the same time is hence a critical goal of a microprocessor designer.

The effect of pipelining is to improve the throughput of the processor by increasing the number of instructions processed per unit time. Pipelining does **not** decrease the time an individual instruction takes.

15.1.1 Pipeline Implementation

Pipelines may vary in length and in the tasks assigned to each step in the pipeline. Figure 15.1 describes an example of the division of work in a pipeline.



Steps in the pipeline:

IF	Instruction Fetch
ID	Instruction Decode
OF	Operand Fetch
OE	Operand Execution
OS	Operand Store

Figure 15.1: An example of the division of pipeline stages and the overlapping of execution

15.1.2 Data Hazards

Data dependency in a pipeline arises when the input of an instruction depends on the output of another instruction, both of which are executing in the pipeline. A number of mechanisms have been suggested to deal with this problem:

- Stall the pipeline. This involves stopping the progression of instructions entering a pipeline at the stage of the pipeline a dependency has been detected at and only resuming the progress of earlier stages after the dependent value has been set. (Figure 15.2)
- Ignore the problem and have the compiler reorder the instructions, where possible, and insert no-ops where necessary to prevent data dependencies in the pipeline.

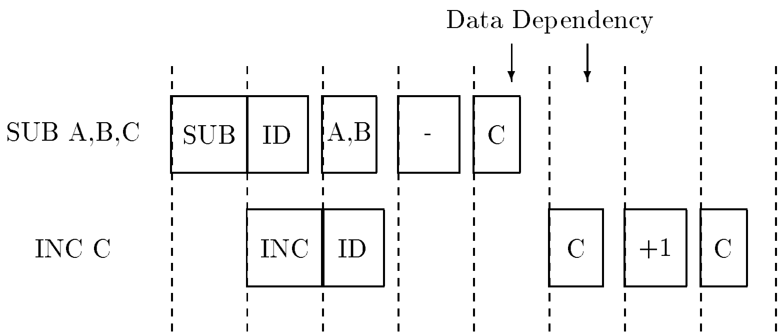


Figure 15.2: A pipeline stall caused by a data dependency

To detect data dependencies adds complexity to the processor as data interlocks need to be added to two steps of the pipeline. The data interlock consists of a test to check if data to be output is used as input data. Data interlocks need to be applied to both registers and external addresses to make stalling the pipeline an effective mechanism for ensuring the expected operation of code.

Data forwarding can be used to reduce the effect of a pipeline stall. In a system using data forwarding instead of waiting for the save of the dependent data to be competed the data is passed directly to the dependent input and the pipeline restarted. This technique reduces the time the pipeline is stalled at the cost of increasing the complexity of the processor.

15.1.3 Branch Hazards

A branch hazard occurs when a jump occurs. There are two varieties of branch hazards: conditional and unconditional. In the case of an unconditional jump input to the pipeline is stalled until the correct address for the jump is determined. In the case of a conditional jump two solutions are possible:

- continue executing the following instruction but be prepared to throw away any of its consequences if the jump occurs
- stall the pipeline until the destination address is known

The former mechanism increases processor performance at the expense of increasing the complexity of the processor.

A further mechanism for dealing with branch hazards - *delayed branch* - is well suited to RISC processors (section 15.3.2). The delayed branch mechanism redefines the branch instruction to take place one instruction after the branch instruction. This means that the pipeline will never have to stall on a branch. This technique requires a compiler to re-order instructions to ensure that there is a suitable instruction or no-op after each branch.

The use of compiler techniques to avoid pipeline stalls caused by branch hazards or data hazards is advantageous regardless of the hardware support provided for coping with the hazard when it arises Avoiding pipeline stalls maintains the throughput of the processor.

15.2 Superscalar Processors

A superscalar processor contains multiple functional units contained within the processor. The functional units may be of the same or different types. In a superscalar architecture more than one instruction can be *issued* at a time provide the instructions are independent. The superscalar mechanism differs from the pipeline mechanism in that it is necessary that the operation of a dependent instruction not be commenced. The steps of instruction execution of a two issue superscalar processor is illustrated in figure 15.3.

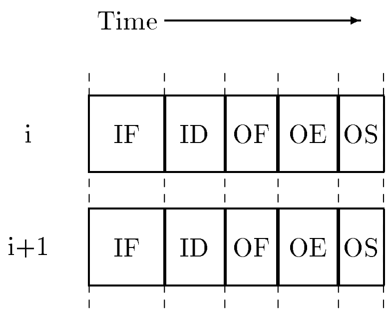


Figure 15.3: Instruction execution in a two issue superscalar processor

The simplest mechanism for ensuring that instructions issued are not dependent is to have each functional unit handle a different type of instruction. A classic example of this type of operation is to have a floating point and an integer unit. Instructions can be issued simultaneously to the two units as there are no interdependencies between the units. This mechanism also simplifies the control of the processor as the order of instruction types can be specified and the functional units loaded in order. The absence of a suitable instruction indicates that a given functional unit should execute a no-op.

When similar functional units are used or it is possible for dependent instructions to issued at the same time it is necessary to include mechanisms similar to the interlocks found in pipelined systems to cause one or more functional units to execute a no-op to prevent the dependent instructions being executed at the same time.

To gain maximum benefit from superscalar processors it is necessary to supply independent instructions to each of the functional units and avoid idle time in any unit. This requires that the compiler organize the instructions into an order which allows parallel operations to take place.

15.3 The RISC / CISC Controversy

15.3.1 CISC

The characteristics of a Complex Instruction Set Computer (CISC) are

- a rich instruction set
- many addressing modes

CISC machines were motivated by microcode. Because of the ease of writing microcode to implement an instruction it was perceived that adding instructions to microcode resulted in an improvement to an instruction set at relatively low cost. Microcode is stored in small high speed memories and hence it was perceived that microcode executed faster than assembly code. This led instruction set designers to add more complex instructions that performed common operations desired by programmers. Finally there was a perception that compilers were complex and by giving the compiler a wider choice of operations and addressing modes it would simplify the design of the compiler.

15.3.2 RISC

The characteristics of a Reduced Instruction Set Computer (RISC) are:

- Simple instructions
- Uniform instruction length
- Few instruction formats
- Orthogonal instruction set
- Few addressing modes
- Load-Store Architecture

- Few data types
- Many registers

RISC processors were driven by the invention of caches, an improvement in compiler technology, and the desire to reduce the complexity of the processor architectures to simplify the task of introducing pipelining. RISC was formed on the observation that the majority of the work of a processor was done by a minority of the instructions, hence an overall speedup could be had if the frequently executed instructions were improved. This led to the introduction of hybrid microprogram and hardwired controllers. This alteration increased processor complexity. The next step in development was motivated by an improvement in compilers and the introduction of caches. With caches made with the same technology as the control store it is no longer true that microcode executed more quickly than simple instructions. The improved compilers were in fact hampered by more complex instructions and multiple addressing modes as they attempted to choose the best mode and instruction from a wide range of possibilities.

The RISC processor aimed to provide a simple set of instructions that worked quickly. In addition the processor was designed to be simple so that the more flexible hardwired control units could be designed at reasonable cost.

The load-store architecture simplifies the design of the processor by limiting memory access to only the load and store instructions. All other instructions have registers as both source and destination.

The regular instruction length simplifies control and decoding logic.

15.3.3 Comparison

In practice the designers of both RISC and CISC processors are driven by the same goals - maximum throughput. This has caused convergence in design. RISC processors commonly include a microcode based floating point unit because it yields better performance than attempting to perform the same operations in machine code. CISC machines have hardwired control for some of their instructions to achieve single cycle execution on some instructions. The mechanisms of pipelining and superscaling are applicable to both architectures, although more difficult to apply to a CISC processor. In essence the distinction between the RISC and CISC processor of today is becoming more blurred.

Measures of performance are complicated by the RISC / CISC design split. The simple measures of MIPS (millions of instructions per second) and processor speed obviously do not serve as adequate measures of performance as the instructions executed can perform vastly different amounts of work. It is common to cite these figures in advertising. If these figures are to be compared it is essential that the comparison only be used between processors of similar types.

15.4 Additional Reading

Further information relating to pipelining processors **chapter 6** and superscalar processors **section 6.11** can be found in Hennessy, John L., Patterson, David A., *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 1994.

RISC issues and pipelining are discussed in Patterson, David A., Reduced Instruction Set Computers, Communications of the ACM, 28(1), January 1985.

Chapter 16

The R2000

16.1 General

The R2000 microprocessor produced by MIPS Computer Systems in 1987 was based on work conducted at Stanford University. A significant feature of the MIPS product was the absence of hardware interlocks to prevent data and branch hazards hence its reliance on compilers aware of the pipeline architecture of the machine. MIPS is an acronym for Microprocessor without Interlocking Pipe Stages. This design choice simplified the processor design and increased pipeline throughput. Another interesting feature of this processor is that the processor can be configured as either a big endian or a little endian processor. In addition it is possible to select the endian mode from software.

16.2 Gross Features

- 32 bit processor
- Address range 2^{31}
- 32 General registers

- Load-Store Architecture

- RISC

16.3 Register Set

- 32 general registers (32 bit)

  - *R0 - R31*

- 2 multiply-divide registers (32 bit)

  - *HI & LO*
  - Result of 32 bit Multiplication
  - Quotient and Remainder of Integer division

16.4 Data Types

The R2000 supports 6 integer data types: signed and unsigned integers of 8, 16 and 32 bit size.

The R2010 floating point co-processor supports IEEE-754 floating point numbers of 32 and 64 bit size.

16.5 Instruction Formats

The R2000 supports 3 instruction formats:

R-Type instruction:

6 bits	*5 bits*	*5 bits*	*5 bits*	*5 bits*	*6 bits*
op	rs	rt	rd	shamt	funct

I-Type instruction:

6 bits	*5 bits*	*5 bits*	*16 bits*
op	rs	rt	address

J-Type instruction:

6 bits	*26 bits*
op	address

op	operation
rs	source register
rt	second register
rd	destination register
shamt	shift amount
funct	variant of operation
address	address

16.6 Addressing Modes

The R2000 supports 4 addressing modes:

Register Addressing The memory location is given by the value of a register

Base Addressing The address of the memory location is calculated by adding the contents of a register to the *address* in the instruction

Immediate Addressing The address of the memory location is the *address* in the instruction.

PC-Relative Addressing The address is the sum of the PC and the *address* in the instruction.

The jump instruction uses the 'J-type' format and exploits the requirement that instructions be aligned on 32 bit boundaries. The address of the destination is calculated:

> The 26 bit *address* field is shifted left 2 bits and combined with the top 4 bits of the PC to provide a 32 bit address.

A consequence of this arrangement is that a linker must avoid attempting to make a jump over a 256M byte boundary. If a jump over a 256M byte boundary is required then the destination address must be loaded into a register and a jump register instruction issued.

16.7 Instruction Set

The R2000 supports 74 instructions.

The subroutine mechanism is typical of many RISC processors, in that the R2000 does not have explicit stack instructions. The mechanism used by the R2000 to implement subroutine calls is to use a **jump-and-link** instruction which jumps to a specified location and stores the return address in a specified register. Returning from a subroutine is performed by issuing a **jump-to-register** instruction. It is the responsibility of each subroutine to ensure that the return address is saved. In addition the caller must save the registers that it wishes preserved over a call.

16.8 Virtual Memory

The R2000 does not support page tables to provide automatic translation of virtual to physical addresses. Instead the R2000 has a 64 entry Translation Lookaside Buffer (TLB). Each virtual address is divided into two components: a 20 bit virtual page number and a 12 bit offset. As each virtual address is uttered the processor attempts to match the virtual page number and PID value with an entry in TLB table if there is a match and there is no rights violation then the base in the TLB entry is added to the offset and a physical address returned, otherwise a trap occurs. This mechanism simplifies the virtual address translation hardware but requires greater intervention than a page table based system.

Virtual Address:

20 bits	*12 bits*
Virtual Page	Offset

TLB entry:

20 bits	*6 bits*	*6 bits*	*20 bits*	1	1	1	1	*12 bits*
Virtual Page	PID		Page Frame	N	D	V	G	Offset

N	Not cached
D	Dirty
V	Valid
G	Global (Suppress PID check)

The PID value in the TLB entry is checked against the PID field in the Entry HI register. If the Global bit is set then the PID check is avoided.

16.9 Summary

In addition to the RISC characteristics exhibited by the R2000 processor the designers have opted for the simplest implementation at the cost of forcing complexity onto the compiler or programmer. The result is a processor that has a high instruction throughput, a minimal instruction set and few features aimed at making the compiler's or the programmer's task easier.

Chapter 17

The 80386

17.1 General

The 80386 microprocessor produced by the Intel Corporation in 1985 is a member of the 80x86 family of microprocessors which has served as the processor for the popular IBM-PC personal computer range. One of the driving forces in the design of the 80386 has been to maintain backward compatibility with software written fro earlier members of the 80x86 family. This requirement caused the designers to provide multiple modes of operation and forced a highly redundant instruction set onto the designers.

The 80386 is a little endian processor.

17.2 Gross Features

- 32 bit processor
- Address range 2^{32}
- 8 General registers
- CISC

17.3 Register Set

- 8 general registers (32 bit)

- 6 segment selector registers (16) bit

The 80386 dedicates some of the general registers to specific functions for some instructions.

17.4 Data Types

The 80386 supports the following integer data types: signed and unsigned integers of 8, 16 and 32 bit size, and signed 64 bit integers.

In addition bit fields, pointers, and strings are supported.

The 80387 floating point co-processor supports IEEE-754 floating point numbers.

In all 23 data types are present in the 80386.

17.5 Instruction Formats

The 80386 supports a single variable length instruction format. This is equivalent to the number of formats that could be formed by taking all the legal combinations of the variable length format.

Instruction Prefix	Address Size Prefix	Operand Size Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
bytes			

Opcode	MOD R/M	SIB	Disp	Imm
1 or 2	0 or 1	0 or 1	0 1 2 or 4	0 1 2 or 4
bytes				

17.6 Addressing Modes

The 80386 supports 11 addressing modes. The addressing of the 80386 modes have been discussed earlier in this book.

17.7 Instruction Set

The 80386 supports 216 instructions. The 80387 supports 80 floating point operations.

17.8 Virtual Memory

The 80386 supports:

- Paging
- Segmentation
- Paged Segmentation

Segmentation is supported using a descriptor table the segment descriptors index into the descriptor table. Each descriptor table entry contains the following information:

- Base Address
- Limit
- Presence Bit - Causes a fault if clear and an attempt has been made to load segment descriptor
- Privilege Level - Specifies minimum privilege required to access segment
- Type - Access rights or Special system types
- Segment Descriptor bit - used to distinguish system segments from user or kernel segments

- Accessed Bit - set if segment has been loaded
- Granularity - if 0 then the limit is in bytes otherwise the limit specifies the number of 32 bit long words that are accessible.

When a process wishes to use a segment it attempts to load a segment register with the segment selector for the segment required.

Segment Selector:

13 *bits*	1	2
Index	T	RPL

Index	Index into Global or local descriptor table
T	Table: 1 for local descriptor table, 0 for global descriptor table
RPL	Requested Privilege Level

The operating system is responsible for dealing with traps caused by invalid and not present descriptors.

The paging scheme used in the 80386 is based on a two level page table structure. Similar to that described earlier.

When performing paged segmentation the segment is decoded to generate the virtual address which is translated by examining the page tables.

17.9 Summary

The 80386 is processor with a diverse and rich instruction set. The paged segmentation scheme employed by the 80386 is rare in current microprocessors and few operating systems take advantage of the presence of segment registers.

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

[1] Not available on the 80386

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:

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:

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

The effective address of a memory location is calculated:

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

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

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				

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

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

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				

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				

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

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				

INC Increment 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

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

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

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

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				

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				

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				

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

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				

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				

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 *src*2 *AND* *src*1 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				

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				

Appendix C

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.

Kernighan, Brian W., Ritchie, Dennis M., *The M4 Macro Processor*, Bell Laboratories.

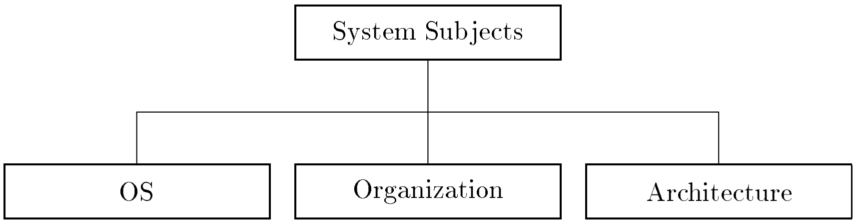
Patterson, David A., *Reduced Instruction Set Computers*, Communications of the ACM, 28(1), January 1985.

Appendix D

OHP slides

Course Outline

What is Computer Architecture?



- Operating Systems - Concerned with the software layer which interacts directly with hardware.
- Computer Organization - Concerned with the details of the subsystems composing a computer system
- Computer Architecture - Concerned with the complete computer system and the interconnections of the components

+ +

The Course in Outline

- Part 1 - 386 Assembly Language Programming
  - 386 Specifics
  - General Programming Techniques
  - Assembly Language Operations
  - Low Level Implementation
- Part 2 - General Architecture
  - Processor Fundamentals
  - IO
  - Memory
  - Architectural Issues
  - R2000 vs 80386

+ +

386 Assembly Language Programming

References: This text. Although there are other books on 80386 programming, students are advised that at present we know of no other text that describes programming the 80386 using a notation consistent with the assembler used in this course.

Processor Architecture

- Instructions
- Integers
- Registers
- Memory

+ +

Instructions

The 386 processor implements a variable length instruction set. The length of an instruction may vary with addressing mode in addition to instruction type.

Types of Operations

- Flow of Control
- Integer
- Floating Point
- Input Output
- String

Privilege

- Non-Privileged
- Privileged

Segments - On Segmented Memory Architecture Only

- Single Segment
- Multi Segment

+ +

Argument Type

- Memory to Memory
- Memory to Register
- Register to Memory
- Register to Register

The 386 does not support Memory to Memory moves[a].
Although moving information from one memory location to another is a fairly common operation the absence of this class of operations does not cause a major performance loss. Why?

- Locality of access
- Memory to Register is faster than Memory to Memory

[a] String instructions are an exception

Integers

In mathematics: a number with no fractional component

In computing an arbitrary sized integer may be considered as a contiguous array of bits. The array has the following features:

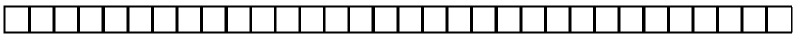
MSB Most Significant Bit - The bit of the array with the greatest absolute weight

LSB Least Significant Bit - The bit of the array with the least absolute weight

These features are not simply related to the ordering of a bit in the array.

Conventionally an integer may be drawn as:

MSB LSB



$Max\ Index$ 0

This conventional representation can be different from the underlying physical representation.

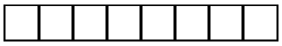
The sources of difference are:

- Bit Indexing
- Endian-ness

Bit Indexing

Most modern computers have the 8-bit byte or a multiple of the 8-bit byte as the fundamental element of storage. Systems may either be labelled

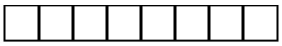
MSB LSB



8 0

or

MSB LSB



0 8

This labelling of bits is only significant in interfacing to hardware.

Big Endian & Little Endian Machines

Little Endian The least significant byte is stored in the byte with the smallest address

Big Endian The most significant byte is stored in the byte with the smallest address

The 386 is **Little Endian**.

Application programmers can detect the effects of a particular endianness of a machine if they work with language structures equivalent to C unions or use pointers to access components of an integer.

+ +

Little Endian

m + 3	MSB
	3
	2
m	LSB

The little endian method has the architectural advantage:

- Given the address of an unsigned number it may be used with any of the processor's integer sizes and interpreted directly.

Big Endian

m + 3	LSB
	2
	3
m	MSB

+ +

Integers can be represented in several ways: BCD, unsigned fixed length, and signed fixed length. Typically when referring to integers in this course we will be referring to an unsigned fixed length value. When referring to a signed integer we will be referring to a two's complement fixed length value.

BCD: Binary Coded Decimal

A binary coded decimal is a variable length encoding of an integer value The 386 supports 2 forms: BCD and packed BCD. A BCD number is encoded in the low order 4 bits of a byte. The encoding is given in the following table.

Binary Value	Decimal Represented
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

A packed BCD value is encoded as pairs of BCD digits within a byte using the encoding in the table above. The more significant BCD digit is at the MSB end.

+ +

Fixed Length Integers

The 386 supports 3 sizes of fixed length integers:

Byte 8-bit value

Word 16-bit value

Long 32-bit value

These data types may be interpreted as either signed or unsigned integers. Two's complement representation is used for signed interpretation. The assembler used in the practical classes uses the suffixes 'b', 'w' and 'l' to indicate the size of the operand.

Pointers

The 386 supports 2 classes of pointers: near pointers and far pointers. Near pointers are essentially a 32-bit offset from the start of a segment. Far pointers are 48 bits in length and consist of a 16-bit segment selector and a 32-bit offset from the start of that segment.
For the practical classes we will only be using near pointers.

+ +

Registers

A register is a storage location distinct from the main memory of a computer. Typically it is distinguished by having a different addressing mechanism to main memory. Registers are usually few in number, typically less than 32, and typically part of the processor. Registers, because of their close association with the processor have a small access time in comparison to memory.
The 386

- is not a general register processor as some registers are assigned special purposes.
- has a relatively small number of general registers: 4 general + 2 almost general

Registers by name and function:

%eax	Accumulator	General
%ebx	Base	General
%ecx	Count	General
%edx	Data	General
%esi	Source Index	Index
%edi	Destination Index	Index
%esp	Stack Pointer	Pointer
%ebp	Base Pointer	Pointer

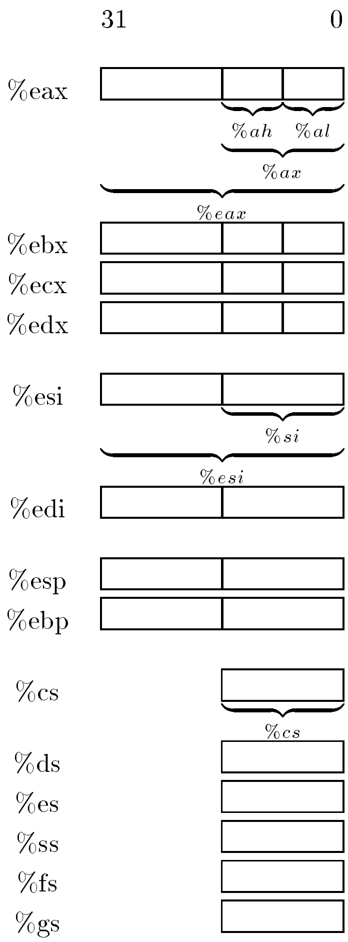
+ +

%cs	Code Segment	Segment
%ds	Data Segment	Segment
%es	Extra Segment	Segment
%ss	Stack Segment	Segment
%fs		Segment
%gs		Segment
EFLAGS	Flag Register	Flags

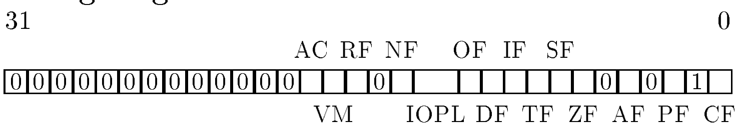
The names of the general registers are mainly of interest only in a historical context. There are still, however, a few functions which use these registers in a non-general way. Most notable of these are: *mul*, *imul*, *div* and *idiv*.

The general registers may be treated as either 8, 16 or 32 bit registers. The index registers may be treated as either 16 or 32 bit registers. The segment registers are 16 bits wide.

Note that during the practical classes attempting to change the segment registers will probably cause you program to be terminated. Segment registers may only contain segment descriptors in protected mode. Attempting to load a segment descriptor without the correct privilege results in a privilege violation.


31
0
%eax
%ah
%al
%ax
%eax
%ebx
%ecx
%edx
%esi
%si
%esi
%edi
%esp
%ebp
%cs
%cs
%ds
%es
%ss
%fs
%gs

Flag Register



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

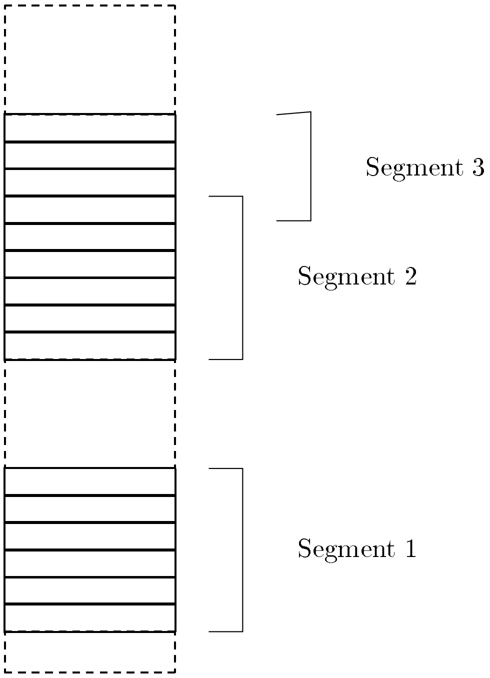
PF Parity Flag

CF Carry Flag

+ +

Segmentation

The 386 implements a segmented memory organization. This means that every address consists of 2 parts: a segment and an offset. A segment is a contiguous area of memory which has a starting address and an extent. Valid references within a segment must have offsets less than the extent of the segment. Segments may be overlapped or be disjoint.
When segments are overlapped a location may be addressed by two distinct names. The address is said to have two aliases.



+ +

In the practical classes segments will effectively be ignored as the 386 implicitly uses particular segment registers with different addressing operations and all the required segment registers have been loaded with descriptors which cover the complete 4Gb address space of the processor and alias all addresses to their offset value.
This is known in Intel parlance as '32 bit flat mode'.

Memory Hierarchy

At this stage an assembly language programmer's view of the memory hierarchy will be introduced. This view consists of 3 elements: registers, memory, and backing store. Later in the course the computer architect's view will be introduced.

Registers	Fast ($< 10nS$)	Very Small ($\sim 64bytes$)
Memory	Medium ($< 100nS$)	Medium ($\sim 16Mbytes$)
Secondary	Slow ($< 15mS$)	Large ($\sim 1Gbyte$)

The figures are conservative and based on typical low end workstation components.
System performance is improved by maximizing the use of the faster elements of the memory hierarchy. The principle of 'Temporal Locality' is central to this type of optimization.

+ +

Assembly & Linking

Assembly language has a one-to-one relationship between assembly instructions and machine instructions. An assembler takes human readable mnemonics and outputs a sequence of binary machine code instructions. For example:

Assembly Language	Memory Dump (Hex)
movl $4, %ecx	cc 04 00 00 00
movl $6, %eax	b8 06 00 00 00
mul %ecx	f7 e1
movl $0, %edx	ba 00 00 00 00

The format of a 486 assembly instruction is:

Instruction Prefix	Address Size Prefix	Operand Size Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
bytes			

Opcode	MOD R/M	SIB	Disp	Imm
1 or 2	0 or 1	0 or 1	0 1 2 or 4	0 1 2 or 4
bytes				

+ +

The general process of assembly consists of 2 logical phases:
Pass 1: For each line of code

- Identifying the instruction
- Identifying the addressing mode
- Identifying segments (on a segmented architecture only)
- Identifying the size of the operands
- Resolving the operands
  - Symbols - check symbol table - if symbol defined get address otherwise get dummy value of correct size and fill in the address when determined and insert symbol as an undefined symbol
  - Immediates - get value from source code
- Generate instruction in the processor instruction format

Pass 2:

- Fix references which contain dummy values

+ +

The traditional representation of a 2 pass assembler is ordered in a different way but contains the same processes as the above. The *GNU as* assembler is a one pass assembler which means that it operates in a way similar to the above but instead of having a second pass it uses 'back patching' when it discovers the address of a symbol that has been used before it has been defined.

Linking

It is desirable to be able to divide a program into multiple source files to

- Isolate unrelated code
- Re-compile or re-assemble only source files which have been changed
- Work with human manageable size segments of code

To support multiple source files it is necessary to have a program which 'glues' the separate files together. That program is the **linker**. It is also necessary to provide a new class of symbol - the *external reference*. The external reference is a symbol which is resolved by the linker and the address of the symbol is substituted into code using the symbol.

+ +

Components of a *GNU as* program

```
.globl main
.globl write_int_r

.data
four: .int 4

.text
main:
      movl four, %eax
      movl $6, %ecx
      mull %ecx
      call write_int_r
      ret
```

- Assembler directives - Instructions to the assembler
  - Segment directives
  - Reserving space for variables
- External references - 2 types
  - Making a symbol declared in this file public
  - Allowing access to a public symbol
- Labels
- Immediate values
- Variables

+ +

Basic Operations

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.

Pigeon hole metaphor

+ +

A short program

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

- Comments
- Labels, Addresses, Constants
- Reserving Space

Memory Reservation Directives

.byte 8 bit integer

.word 16 bit integer

.int 32 bit integer

.long 32 bit integer

.ascii String of bytes

.asciz String of bytes terminated by a null

+ +

Note that space directives do not reserve space unless there is an initialization value present. Each element of the initialization value list reserves space of the size of the directive.

```
.long 5       /* reserves 32 bits & places 5 in it */
.long 5, 7    /* reserves 2 longs & places 5 & 7 */
.byte 4, 6    /* reserves 2 bytes & places 4 & 6 */
.long         /* reserves no space */
```

Strings

`f: .ascii "hi"`

f	'h'
	'i'

`f: .asciz "hi"`

f	'h'
	'i'
	0

Immediates

Immediate values are formed by prefixing a '$' to a label or a number. In the case of a label the immediate value is the address represented by the label. For numbers, the value is the constant represented by the number.

+ +

AT&T Format

$$a \ op \ b \rightarrow c$$
$$opcode \ src_n, \ldots \ dest$$

The AT&T instruction format places the result in the rightmost operand. The 80386 instruction set mainly employs 2 operand instructions, hence the rightmost operand typically doubles as both a source and a destination.

$$a - b \rightarrow a$$
$$subl \ b, \ a$$

+ +

Operations

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.

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

Arithmetic

A large number of arithmetic operations are supported at the instruction level. These instructions can be grouped as arithmetic and bitwise logical.

add Add source to destination leaving result in destination. Note that only one argument may be a memory location.
cf. sub

and Bitwise-and the source and destination leaving result in destination. Note that only one argument may be a memory location.
cf. or, xor

+ +

mul Perform an unsigned multiply on the %eax register (or a subset) and the instruction argument leaving the result in the register pair %edx:%eax (or a subset).

imul Perform a signed multiplication.

- As **mul** but performs an signed multiplication
- Two operand form multiplies source by destination leaving the result in destination.
- Three operand form multiplies source1 by source2 and leaves result in destination.

div Perform an unsigned division.

- Single operand form. Divide the register pair %edx:%eax (or subset) by instruction argument. The result is returned in %eax and the remainder in %edx (or subsets).
- Two operand form. As above but explicitly identifies registers.

idiv As **div** but performs signed division.

not One's complement negation of operand

neg Two's complement negation of operand

+ +

Jumps

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.

Alternation
All decisions in a 386 assembly language program are based on the state of the EFLAGS register. The EFLAGS register is altered by the operations:

- *arithmetic*
- *bitwise logical*
- test
- cmp

The majority of other operations do not affect the EFLAGS register. Conditional jumps are taken or not depending on the state of the bits in EFLAGS.

+ +

Conditional Jumps

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 $(ZF = 1)$

Setting the condition flags

Test Performs a bitwise-and without returning a result.

Cmp Performs a subtraction without returning a result.

Both mechanisms set the condition flags in the EFLAGS register.

+ +

The format of an assembly language program

There is a long historical tradition connected with the layout of an assembly language program. The layout described maximizes the readability of an assembly language program.

1. Programs are laid out in 3 regions
2. Labels, assembler segment directives, and global declarations go in the leftmost region
3. Code and space reserving directives go in the center region
4. Comments go in the rightmost region if they relate to an instruction otherwise comments may start in the other 2 regions.

\+ +

For example:

```
/* Program to square a number */
.globl main
.globl write_int_r
.globl read_int_r

.data
n:   .int 0

.text
main:
     call read_int_r   /* read in integer */
     movl %eax, n
     mull n            /* square integer */
     call write_int_r  /* write out result */
     ret
```

+ +

Representation & Organization

In Brief

- Algorithms - are recipes for performing operations
- During this course we will represent algorithms using
  - Flow Charts
  - Pseudo Code
  - Narrative Descriptions
  - Mathematical Formula
  - Programming Languages
- The advantages of using a representation other than the program code itself.

Algorithms

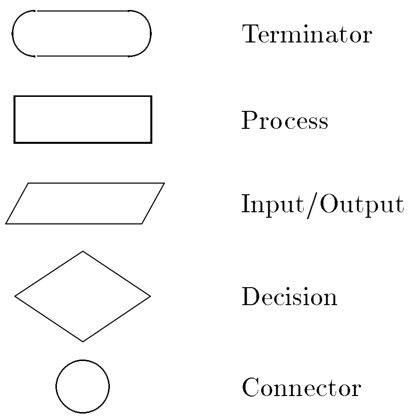
- An algorithm is a set of instructions for performing some task.
- The detail of the expression of an algorithm may be specified by the author of the algorithm.
- The generality of an algorithm may be specified by the author.

- Algorithms may be partitioned into parts
- Algorithms may be expressed in many ways

Representation
Human languages provide us with a way of expressing instructions. However, as a mechanism for expressing algorithms, the narrative description, is poor. The problem with languages similar to English are:

- Lack of precision
- Overloaded meanings
- Verbosity
- Subject to individual interpretation

A good method of representing an algorithm must avoid these problems and also be clear and readable.
Flowcharts

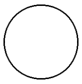


Terminator

This symbol is used to indicate the beginning and end of a flowchart. The word start is placed in the symbol to indicate the start of the routine, the word stop is used to indicate the end of the routine.

Process

A description of one or more actions are placed inside the box. An action may be defined by another flowchart.

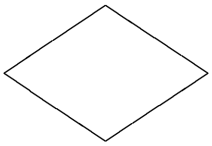


Connector

This symbol is used to connect lines on separate pages of a flow chart. A number or name is placed in the circle to identify the connection.

Input/Output

Input or Output to a variable is represented by this symbol. The word 'read' or 'write' is followed by the variable to be read in or output.



Decision

The condition written in this box determines which of the paths are to be followed out of the box. The paths leading out of the box are labelled with a possible result of the condition.

+ +

Lines are used to connect the elements of the flow chart. It is assumed that control flows down the page unless an arrow is used to alter the direction of flow.
The author of a flow chart is responsible for the readability of the flow chart. It is often necessary to break a complex process down into components and draw individual charts for those components.
Advantages and Disadvantages
The major advantage of the flowchart is that the operations supplied by the flowchart symbols are similar to the operations available to the assembly language programmer.
The flow chart has become unpopular for the following reasons:

- 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 assembly language.

+ +

Pseudo Code

Pseudo code or **Structured English** provides precision and removes the ambiguity present in a narrative description. This achieved by using defined keywords which structure the description.
The keywords in pseudo code are:

- **start** ... **stop**
  - *Start* and *Stop* delimit the algorithm.
- **if** ... **then** ... **else**
  - The *then* clause of this structure is executed if the condition following *if* is satisfied. Otherwise the *else* clause is executed.
- **repeat** ... **until**
  - The body of the loop is executed until the condition following *until* is satisfied.
- **while** ... **do**
  - The body of the loop is executed while the condition following *while* is satisfied.

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

+ +

Example - Vertical Parity on A Data Stream

Narrative Description

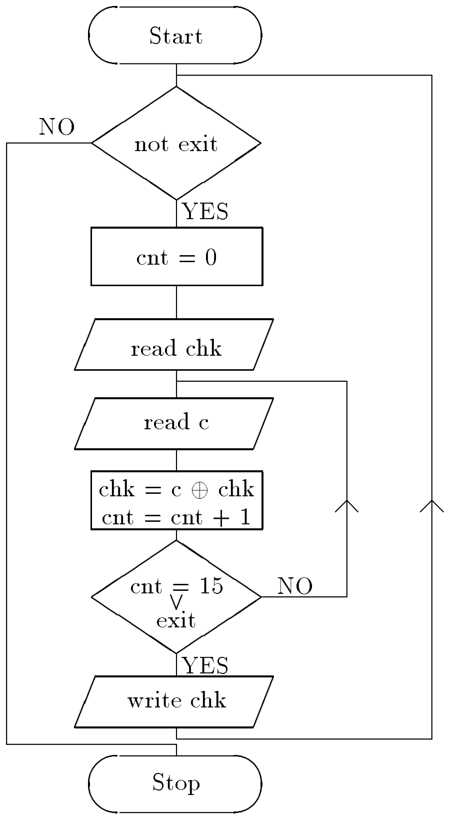
A vertical parity is to be added to a data stream. A block size of 15 is required. Produce the vertical parity by using the exclusive-or function to xor each element of the incoming stream. Exit from the process if the exit variable is set.

Pseudo Code

```
while (not exit) do
   cnt = 0
   read chk
   repeat
      read c
      chk = c xor chk
      cnt = cnt + 1
   until ((cnt = 15) or (exit = true))
   write chk
```

+ +

Flow Chart


Start
not exit
NO
YES
cnt = 0
read chk
read c
chk = c ⊕ chk
cnt = cnt + 1
cnt = 15
∨
exit
NO
YES
write chk
Stop

+ +

Assembly Code

```
.globl verchk
.globl read_char
.globl write_char
.globl exit

.data
cnt:    .int 0
chk:    .byte 0

.text
verchk:
l1:     cmpl $1, exit
        je x1

        movl $0, cnt
        call read_char
        movb %eax, chk

l2:     call read_char

        xorb %eax, chk
        incl cnt

        cmpl $15, cnt
        je xl1
        cmpl $1, exit
        je xl1

xl1:    movb chk, %eax
        call write_char
        jmp l1

x1:     ret
```

+ +

Algorithms & Implementations

It is a common practice, in computer science, to write algorithms without considering the implementation architecture. Typically algorithm writers assume:

- Infinite precision
- Infinite accuracy
- Unlimited word size
- Infinite memory

A consequence of this is that implementing algorithms requires great care to ensure that situations which were unanticipated by the algorithm writer are avoided, handled or noted. Failure to take into account the architecture on which an algorithm is implemented can result in incorrect answers.

Common Problems

- Large positive numbers can become negative under addition using a signed representation
- Large negative numbers can become positive under subtraction when using a signed representation
- Large positive numbers can become small positive numbers under addition when using an unsigned representation

+ +

- Under two's complement notation there is one more negative number than positive number

Solutions

- Limit problem space
- Defensive programming practices

Program Structure

High Level view Describes the steps of solving a problem without reference to the details of a particular implementation.

Low Level view Consists of the details required for a particular implementation.

Both views are required to implement programs effectively. In addition there is a spectrum of views between the two extremes. The process of **stepwise refinement** can be used to provide both a high and low level representation within a single program.

Top Down Approach

The Top Down Approach is implemented using stepwise refinement. Essentially a problem is defined in a abstract terms. The abstract concepts are implemented using simpler elements until a sufficiently simple element is found and implemented at the base level.

Advantages

- Reduces problems to components of a manageable size
- Isolates components of the solution allowing for easier maintenance
- Makes the program more accessible to other programmers

+ +

Mechanisms for Implementing Stepwise Refinement
The major mechanism available to the assembly language programmer for the implementation of stepwise refinement is the subroutine. A subroutine can be used to contain the details of a step in the solution. The subroutine call can be used to represent the abstract concept.
An example of this is in 'Practical Sheet 3' where the factorial function is used as part of the combination function. In this case the factorial function is the basic unit of operation that has been implemented. The combinatorial function is implemented in terms of this low level function.

Internal Documentation

The C language comment delimiters are used to mark comments under *GNU as*.
In a well commented program, comments:

- Explain the action of a piece of code in the problem domain.
- Comments do **not** explain the function of an assembly language instruction. (This should be obvious from the instruction itself and a definition of the instruction set)

+ +

- Clarify any unusual coding
- Describe the function of a subroutine

Comment a program as it is written. Adding comments at the end invariably loses some pertinent information relating to the reasons for a particular coding.
Comment liberally, but do not state the obvious. A blow by blow description of a mundane, easily understood task is a waste of time. A detailed description of the higher level abstract operation of a function or exploitation of some existing precondition are worthwhile.
Good & Bad Comments

```
addl $1, %eax       /* Add one to EAX */
addl $1, %eax       /* Increment array ptr */

movl $0, %edx       /* Zero EDX */
movl $0, %edx       /* Clear top word EDX:EAX */

call write_int_r    /* Write out value in EAX */
call write_int_r    /* Display result */

je exit             /* if equal jump to exit */
je exit             /* if equal exit subroutine */
je exit             /* if x = y exit subroutine */
```

+ +

Introduction to Subroutines

Before describing the operation and mechanism of a subroutine, the concept of the system stack will be covered.

The System Stack

The 386 supports stack operations using the Stack Pointer register (%esp) and the Base Pointer register (%ebp). The operation of the system stack will be discussed here with reference to the stack pointer register (%esp), the base pointer register and its usage will be introduced in a later section.

The abstract datastructure is defined:

Stack An ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the *top* of stack.[a]

Two operations are defined over a stack: **push** and **pop**.

Push add a new item to the collection

Pop remove an item from the collection

[a] Tenenbaum, A M., Augenstein, M J., Data Structures Using Pascal, Prentice-Hall, 1991, pg 67

+ +

Implementation of the System Stack on the 386

The 386 stack contains objects which are 16 and 32 bits in size. The stack has a granularity of 16 or 32 bits. This granularity is characterized by the fact that the 386 provides two forms of both push and pop.

16 bit pushw & popw

32 bit pushl & popl

The function of each of these instructions can be emulated with a series of instructions. Writing the equivalent operation will be covered after the addressing modes of the 386 are discussed.

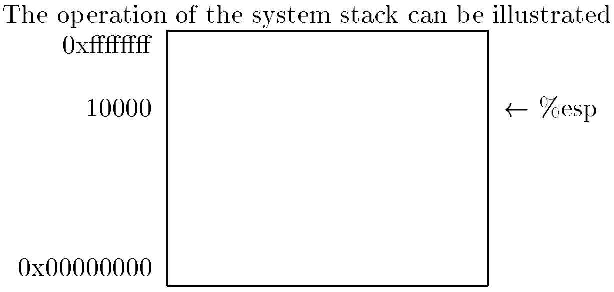
Push

```
IF pushw
THEN
    ESP = ESP - 2
    copy content of source to word pointed to by ESP
ELSE
    ESP = ESP - 4
    copy content of source to long pointed to by ESP
```

Pop

```
IF popw
THEN
    copy content of word pointed to by ESP to dest
    ESP = ESP + 2
ELSE
    copy content of long pointed to by ESP to dest
    ESP = ESP + 4
```

The operation of the system stack can be illustrated as follows:



pushl $4

0xffffffff		
10000		
ffff	0	
fffe	0	
fffd	0	
fffc	4	← %esp
0x00000000		

pushw $2

0xffffffff		
10000		
ffff	00	
fffe	00	
fffd	00	
fffc	04	
fffb	00	
fffa	02	← %esp
0x00000000		

+ +

popw %eax

0xffffffff		
10000		
ffff	00	
fffe	00	
fffd	00	
fffc	04	← %esp
fffb	00	
fffa	02	
0x00000000		

AX ← 2

popl %eax

0xffffffff		
10000		← %esp
ffff	00	
fffe	00	
fffd	00	
fffc	04	
fffb	00	
fffa	02	
0x00000000		

EAX ← 4

popl %eax

Address	Contents	
0xffffffff		
10000		
ffff	00	
fffe	00	← %esp
fffd	00	
fffc	04	
fffb	00	
fffa	02	
0x00000000		

EAX ← 0x040002

popw %eax

Address	Contents	
0xffffffff		
10000		← %esp
ffff	00	
fffe	00	
fffd	00	
fffc	04	
fffb	00	
fffa	02	
0x00000000		

AX ← 0

+ +

Subroutines and the System Stack

The address of a subroutine is a 32 bit integer. The stack is used to store these addresses allowing subroutines to be nested.

Call Call pushes the address of the next instruction onto the stack and jumps to the argument address

Ret Pops the stack and jumps to the address recovered

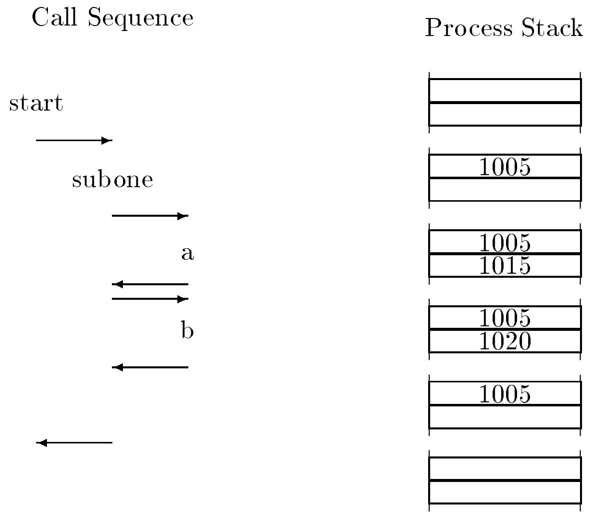
This scheme allows the easy nesting of subroutines and lets the processor do most of the work in managing subroutine calls. However, this scheme does require the programmer to ensure:

- All pushes are matched to pops within a subroutine
- That access to the stack data area must not over-write addresses placed on the stack.
- That the stack pointer must not be corrupted by the program

Stacks & Subroutines

A simple nested subroutine

```
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 */
```



+ +

Simple Parameter Passing
There are many ways of passing parameters to subroutines. Two simple methods of passing parameters will be introduced. Later in the course additional methods will be covered.

- Pass by register
- Pass by memory

Pass by Register
When using this form of parameter passing the parameters are placed in registers before the subroutine is called. The subroutine uses the parameters found in the registers and performs its function.

+ Speed: No memory move overhead
+ Simple: No name clashes
- Number: Limited number of parameters can be passed
- Size: Limited amount of information can be passed

Example

```
mov $1, %eax
mov $2, %eax
call test
```

+ +

Pass by Memory
When using this form of parameter passing the parameters are placed in memory locations before the subroutine is called. The subroutine uses the parameters found in the memory locations and performs its function.

+ Number: Any number of parameters can be passed
+ Size: Any amount of information can be passed
- Speed: Requires a memory move before a parameter can be accessed
- Complex: It is necessary to ensure that name clashes do not occur

Example

```
mov $1, parm1
mov $2, parm2
call test
```

+ +

Preserving Registers

Although subroutines are useful in breaking up a program into smaller, more manageable sized segments there it is necessary to ensure that no subroutine called by your routine or subroutine called by a subroutine you call disturbs a register containing information used by your routine. For example:

```
mov $4, %ebx
mov $3, %eax
call write_int_r
addl %ebx, %eax
call write_int_r
```

Would produce a strange result if the *write_int_r* subroutine had a **side-effect** that altered the EBX register.
Because it is possible to call routines for which the source code is unavailable it is necessary to have techniques that allow you to save and restore the values of registers across subroutine calls. In addition these techniques can be used to write well behaved subroutines that have no side-effects.
Two techniques will be introduced for preserving register contents:

- Save to static memory location
- Save to stack

+ +

The two techniques can be used to:

- Avoid problems caused by subroutine side-effects
- Prevent subroutine side-effects

To achieve the former required registers are saved before calling the subroutine and restored after the completion of the subroutine. To achieve the latter the registers which a subroutine may alter are saved by the called subroutine and restored before the subroutine returns.

Save to static memory location

In this method the value of the register is stored in a memory location. This mechanism does not support recursion. An example of this method would be:

```
movl %ebx, sebx
movl %eax, seax
call test
movl sebx, %ebx
movl seax, %eax
```

Save to stack

A more flexible method is to push the contents of registers onto the stack. This method has the advantage that it can be used in recursive subroutines.

+ +

```
pushl %ebx
pushl %eax
call test
popl %eax
popl %ebx
```

Subroutines with no side-effects

In order to write a subroutine with no visible side-effects it is necessary to introduce a method for saving the flag register. **Pushf** pushes the contents of EFLAGS onto the stack, using this in conjunction with **pusha**[a] it is possible to store the majority of the user alterable state of the 386 operating in 386 flat mode.

Example: Debugging Subroutines

A first cut at a debugging routine might be:

```
eaxout:
       pushl %eax
       call write_int_r
       movl nl , %eax
       call write_char_r
       popl %eax
       ret
.data
nl:    .ascii "\n"
```

[a] Pushes the general registers onto the stack

+ +

It eases the task of debugging code markedly if the code used to assist in the debugging has minimal side-effects. The following subroutine illustrates the method used to achieve this aim and provides fairly comprehensive output:

```
regout:
       pusha
       pushf
       call write_int_r
       movl sp , %eax
       call write_char_r
       movl %ebx, %eax
       call write_int_r
       movl sp , %eax
       call write_char_r
       movl %ecx, %eax
       call write_int_r
       movl sp , %eax
       call write_char_r
       movl %edx, %eax
       call write_int_r
       movl nl , %eax
       call write_char_r
       popf
       popa
       ret
.data
nl:    .ascii "\nl"
sp:    .ascii " "
```

+ +

Recursion - A Minimalist Example

A recursive version of the factorial function can be written without the need to save values. Only a limited class of recursive problems may be implemented in this way. In the section on advanced subroutines a general method will be discussed for the implementation of recursive routines.

```
.data
n:      .long 0
pi:     .long 0
.text
fact:
     movl $1, pi     /* initialize */
     movl %eax, n
     call fact1
     ret
fact1:
     cmpl $0, n      /* termination condition */
     je efact        /* n = 0 */
     movl pi, %eax   /* recurrence relation */
     mull n          /* pi = n (n+1) */
     movl %eax, pi
     decl n          /* n-- */
     call fact1      /* recurse */
     ret
efact:
     movl pi, %eax   /* put results into %eax */
     ret
```

+ +

Student Exercise

Rewrite the recursive factorial program to use registers instead of memory locations to store intermediate results.

Sample Answer

```
fact:
        movl $1, %ebx    /* initialize pi = %ebx */
        call fact1
        ret
fact1:
        cmpl $0, %eax    /* termination condition */
        je efact         /* n = 0 */
        movl %eax, %ecx
        movl %ebx, %eax
        mull %ecx        /* pi = n (n+1) */
        movl %eax, %ebx
        movl %ecx, %eax
        decl %eax        /* n-- */
        call fact1       /* recurse */
        ret
efact:  movl %ebx, %eax  /* put results into %eax */
        ret
```

The purpose of the exercise:

- To demonstrate the convenience of named memory locations
- To illustrate register optimization

+ +

Overheads & Macros

Although subroutines are highly useful, there is an associated cost with each subroutine call. That cost is the time taken to write the address of the return location onto the stack on a call and the time taken to retrieve the return location from the stack on return.

When writing time critical applications the cost of these overheads can be significant. To address this problem and still retain the properties of code modularity the **macro** was introduced.

A macro is a piece of code which is textually substituted into the source code of a program before the program is assembled. This allows subroutine like textual organization of the code but does not incur the call and return overheads. Macros have an associated cost known as 'code explosion', that is that each instance of the macro generates the code of the macro, enlarging the assembled program's size.

The *GNU as* assembler does not have an inbuilt macro language. Macros are only mentioned in this course for completeness.

The following example uses the m4 macro preprocessor to illustrate how macros might be used.

+ +

```
.globl main
.globl write_int_r

changequote([,])
define(dmp, [mov $1, %eax
        call prtout])

.data
nl: .asciz "\n"
sp: .asciz " "

.text
main:
        movl $'a, %ebx
        dmp(%ebx)
        ret
```

After being run through the m4 preprocessor:

```
.globl main
.globl write_int_r

.data
nl: .asciz "\n"
sp: .asciz " "

.text
main:
        movl $'a, %ebx
        mov %ebx, %eax
        call prtout
        ret
```

+ +

Control Structures

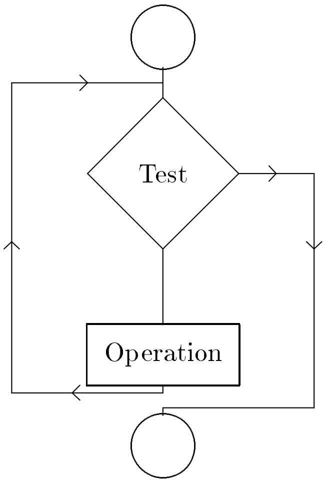
Why standard control structures?

- Make code less confusing
- Make code more modular
- Make code easier to modify

This section will cover the standard structures and methods of combining them to form programs. It should be noted that there are alternate implementations of these structures.

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

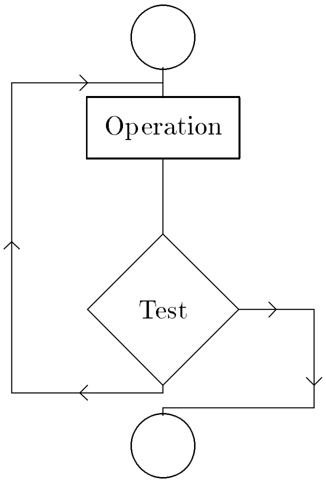
```
while z not equal 0 do
      a = a + a
      z = z - 1
```

```
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 */
```

+ +

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.



+ +

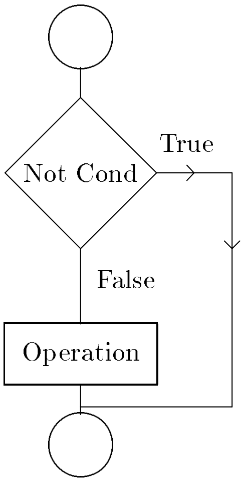
Example

```
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 */
```

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

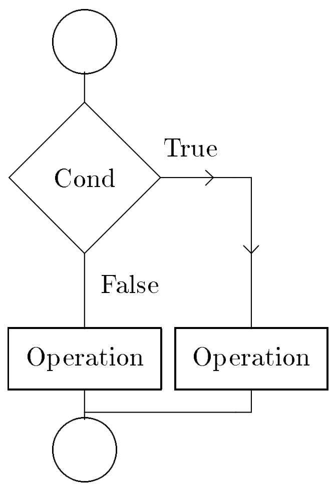
```
if z equal 0 then
      a = 1
```

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

+ +

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

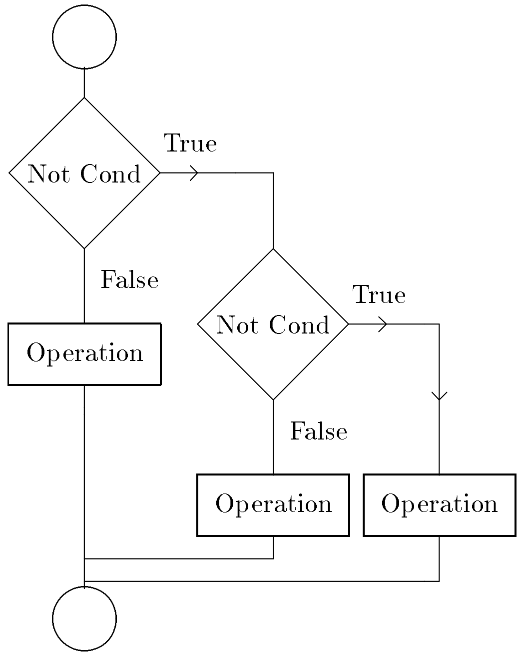
```
if z equal 0 then
      a = 1
else
      a = 2
```

```
        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 */
```

+ +

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 */
```

+ +

Switch

The switch or case statement may be implemented in two ways: the first is to use the **If** ... **Then** ... **ElseIf** ... **Else** construct. 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

+ +

Addressing Modes

So far we have used two addressing modes: immediate and direct. Now we will examine 2 further addressing modes. The syntax, semantics and use of all the available addressing modes on the 80386 will be discussed. The 80386 supports the following addressing modes:

Immediate The value returned is the value of the argument

Direct The value returned is the value contained by the location specified by the argument

Indirect The value returned is the value of the location specified by the contents of a register or memory location[a]

Indexed The value returned is the value of the location specified by the sum of at least a base value and an index value multiplied by an item size.

[a] Indirection using memory locations is limited on the 386/486

+ +

Indexed Addressing on the 80386

In immediate mode addressing the value is explicitly encoded into the instruction. In direct addressing the address of the location which contains the value is explicitly mentioned.

Indexed mode addressing calculates the *effective address* of the location referred to. The *effective address* is the actual address of the referred location. In a general system the effective address can be based on the contents of memory locations, the contents of registers and constants. In the 80386 the effective address is based on the contents of registers and constants only.

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

$$displacement + base + (index * scale)$$

The effective address is relative to a segment. The default segments are:

Ref Type	Seg	Default
Instruction	CS	Automatic with Instruction Fetch
Stack	SS	Push, Pop, & any memory reference with ESP or EBP as a base register
Local	DS	All data references **except** strings and when relative to the stack
String	ES	Destination of strings

+ +

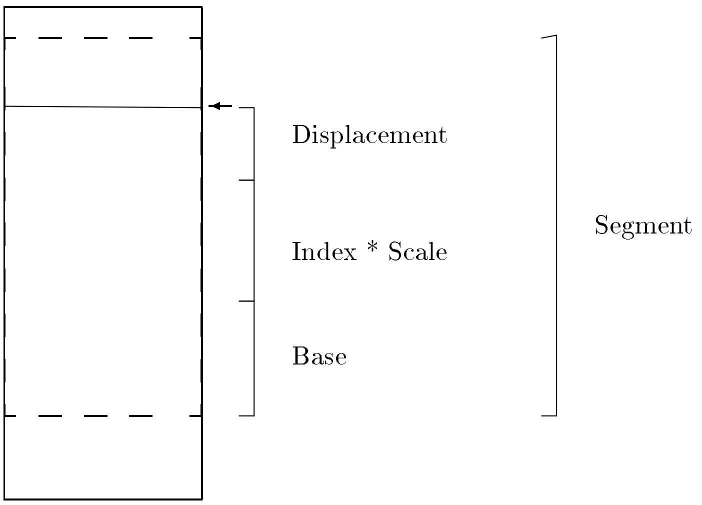
The default segment can be overridden by using a segment override. In 32-bit flat mode the segmentation has no effect on user programs as all the segments are mapped to the same region of memory.
The 80386 instruction set allows the parameters of the effective address calculation to be replaced only as indicated:

$$\left\{ \begin{array}{c} No\ Disp \\ 8 - Bit\ Disp \\ 32 - Bit\ Disp \end{array} \right\} + \left\{ \begin{array}{c} - \\ \%eax \\ \%ebx \\ \%ecx \\ \%edx \\ \%esi \\ \%edi \\ \%ebp \\ \%esp \end{array} \right\} + \left\{ \begin{array}{c} - \\ \%eax \\ \%ebx \\ \%ecx \\ \%edx \\ \%esi \\ \%edi \\ \%ebp \end{array} \right\} * \left\{ \begin{array}{c} - \\ 1 \\ 2 \\ 4 \\ 8 \end{array} \right\}$$

The following combinations of the effective address calculation are permitted:

- Displacement
- Base
- Base + Displacement
- (Index * Scale) + Displacement
- Base + Index + Displacement
- Base + (Index * Scale) + Displacement

+ +



Indirect Addressing

Indirect addressing consists 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.
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.

$movl\ 1004(,1),\%eax$

Address	*Memory*	
1000		
1004	2008	$\rightarrow$
1008		
1012		
2000		
2004		
2008	45	$\leftarrow$
2012		

GNU as Syntax

Immediate Addressing
Immediate values are prefaced by '$'.
Examples

```
movl $4, %eax
movl $fred, %eax
```

Direct Addressing
Direct addresses are not prefaced.
Examples

```
movl 0x4, %eax
movl john, %eax
```

Indexed Addressing
Addresses of this form are expressed with the syntax:

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

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

Examples

```
movl %eax, foo(%ebx)
movl %eax, (%ebx,%ecx,2)
```

Indirect Addressing
Warning: Not supported by the version of *GNU as* used in practical work. Do not use.
Is a syntactic exception under *GNU as*. It is expressed as

$$location(,1)$$

Note that only moves and jumps are supported by indirect addressing and that the moves must be to or from the %eax register.
Examples

```
movl %eax, foo(,1)
movl foo(,1), %eax
```

Indexed Addressing - Detailed Examples

Data Declarations

```
.data
a:      .long 1,2,3
b:      .long 0
c:      .byte 1,2,3
d:      .byte 0
sp:     .ascii " "
```

Picture of Memory

sp	0x20
d	0x00
	0x03
	0x02
c	0x01
	0x00
	0x00
	0x00
b	0x00
	0x00
	0x00
	0x00
	0x03
	0x00
	0x00
	0x00
	0x02
	0x00
	0x00
	0x00
a	0x01

+ +

Displacements - From Base

Program

```
.globl main
.globl write_int_r
.globl dispnum

.data
a:      .long 1,2,3
b:      .long 0
c:      .byte 1,2,3
d:      .byte 0
sp:     .ascii " "

.text
main:
        movl $a, %ebx
        movl (%ebx), %eax
        call dispnum
        movl 4(%ebx), %eax
        call dispnum
        movl $b, %ebx
        movl -4(%ebx), %eax
        call dispnum
        ret
```

Results

```
1 2 3
```

+ +

Indexing - From Displacement

Program

```
.globl main
.globl write_int_r
.globl dispnum

.data
a:      .long 1,2,3
b:      .long 0
c:      .byte 1,2,3
d:      .byte 0
sp:     .ascii " "

.text
main:
        movl $0, %ebx
        movl a(,%ebx,4), %eax
        call dispnum
        movl $1, %ebx
        movl a(,%ebx,4), %eax
        call dispnum
        movl $-1, %ebx
        movl b(,%ebx,4), %eax
        call dispnum
        ret
```

Results

```
1 2 3
```

+ +

Indexing - Size

Program

```
.globl main
.globl write_int_r
.globl dispnum
.data
a:      .long 1,2,3
b:      .long 0
c:      .byte 1,2,3
d:      .byte 0
sp:     .ascii " "
.text
main:
        movl $a, %ebx
        movl $0, %ecx
        movl (%ebx,%ecx,4), %eax
        call dispnum
        movl $c, %ebx
        movl $1, %ecx
        movl $0, %eax
        movb (%ebx,%ecx,1), %eax
        call dispnum
        movl $0, %ecx
        movl $0, %eax
        movl (%ebx,%ecx), %eax
        call dispnum
        ret
```

Results

```
1 2 197121
```

+ +

Revision

Instant Revision of Indexing

The format of an indexed memory access is:

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

where the valid values are:

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

The *effective address* - relative to the segment - of the operation may be calculated using:

$$displacement + base + (index * scale)$$

Instant Revision of Assembler Directives

.fill *repeat*, *size*, *value* Creates a block of memory *repeat* * *size* containing *value* represented in *size* bytes.

.space *repeat*, *size* Creates a block of memory *repeat* * *size* containing zeros

+ +

Examples of converting 'C' to Assembly Language

The following fragments of 'C' code will be translated to assembly language. This will illustrate the use of indirect addressing and the implementation of high level language constructs in assembly language.

Arrays

C

```
int array[10];

/* ... */

array[4]++;
```

Assembler

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

/* ... */

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

Note:

- Objects are 4 bytes in size
- Scale Factor is 4
- Transfer size is 4 (long)

Picture of Memory

	0x00	
	0x00	
	0x00	
	0x00	array[9]
	0x00	
	0x00	
	0x00	
	0x00	array[4]
	0x00	
	0x00	
	0x00	
	0x00	array[3]
	0x00	
	0x00	
	0x00	
	0x00	array[2]
	0x00	
	0x00	
	0x00	
	0x00	array[1]
	0x00	
	0x00	
	0x00	
array	0x00	array[0]

+ +

C

```
char array[10];

/* ... */

array[4]++;
```

Assembler

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

/* ... */

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

Note:

- Objects are 1 byte in size
- Scale Factor is 1
- Transfer size is 1 (byte)

+ +

Picture of Memory

	0x00	array[9]
	0x00	array[4]
	0x00	array[3]
	0x00	array[2]
	0x00	array[1]
array	0x00	array[0]

+

+ +

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)
```

+ +

Picture of Memory

	0x00	first.color
	0x00	
	0x00	
	0x00	
	0x00	first.y
	0x00	
	0x00	
	0x00	
first	0x00	first.x

+ +

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 */
movw $4, (%eax,%ebx,4)
/* offset of x = 2 */
movb $1, 2(%eax,%ebx,4)
/* offset of y = 3 */
movb $2, 3(%eax,%ebx,4)
```

+ +

Picture of Memory

	0x00	cloud[999].y
	0x00	cloud[999].x
	0x00	
	0x00	cloud[999].id
	0x00	cloud[4].y
	0x00	cloud[4].x
	0x00	
	0x00	cloud[4].id
	0x00	cloud[3].y
	0x00	cloud[3].x
	0x00	
	0x00	cloud[3].id
	0x00	cloud[2].y
	0x00	cloud[2].x
	0x00	
	0x00	cloud[2].id
	0x00	cloud[1].y
	0x00	cloud[1].x
	0x00	
	0x00	cloud[1].id
	0x00	cloud[0].y
	0x00	cloud[0].x
	0x00	
cloud	0x00	cloud[0].id

+ +

Pointers
C

```
int x;
int *y;

x = 4;
y = &x;
printf("%d %d %d", x, y, *y);
```

Assembler

```
.globl write_int_r
.globl write_char_r

x:   .int 0
y:   .long 0
sp:  .ascii " "

     movl $4, x
     movl $x, y
     movl x, %eax
     call write_int_r
     movl sp, %eax
     call write_char_r
     movl y, %eax
     call write_int_r
     movl sp, %eax
     call write_char_r
     movl y, %ebx
     movl (%ebx), %eax
     call write_int_r
```

+ +

Results

```
4 548 4
```

In this case the dereference operation is implemented as an indexed access to the base address.

+ +

Subroutines - Advanced

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:

- pass by stack
- pass by register

These mechanisms can be combined to yield a hybrid
There are two types parameters:

- reference parameters
- value parameters

We will be covering:

- definition
- implementation
- characteristics

of parameter passing methods and parameter types.

+ +

Pass by Register

This is the simplest form of parameter passing and should be familiar as it has been used in prac work to date. 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 advantages of this form are

- it permits the subroutine direct access to the parameters in registers

The disadvantages of this form are

- Limited number of parameters can be passed
- Limited amount of information can be passed
- Requires additional work to generate a local copy of the parameter value

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 diagrammatically:

%esp+12		
%esp+8	1	
%esp+4	2	
%esp+0	ret addr	← %esp

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.

%esp+14			
%esp+10	1		4 bytes
%esp+8	2		2 bytes
%esp+4	3		4 bytes
%esp+0	ret addr	← %esp	4 bytes

It is essential that the stack pointer be reset to the position it held before parameters were pushed either by the called subroutine or by code following the call on the subroutine. Failure to do this before a return uses the return value stored on the stack 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 must 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.

+ +

Pass by Value and Pass by Reference

Pass by value should be familiar as both the examples in practical work and the 'C' programming language use this construct.

Pass by value Passes a complete copy of the object named as a parameter to the function

Pass by reference Passes the address of the object named as a parameter to the function

Pass by reference is seen in the Pascal programming language and is represented by **var** parameters. 'C' does not support pass by reference. 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 sufficiently large, then the convenience gained is offset by the overhead of copying the data item.

+ +

It is useful to note that the assembly language version of the program closely resembles the 'C' implementation of the program.

```
void addtwo(int *result, p1, p2)
{
    *result = p1 + p2;
    }

/* ... */

    addtwo(&res, 2, 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.

+ +

Local Variables

A local variable is a variable that is not visible to the caller of a subroutine but is visible to the subroutine. Local variables:

- reduce the amount of global storage space required for a program
- provide 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.
Note: Creating space on the stack is NOT equivalent to malloc's in C. Space creation using the stack is equivalent to local variables in C. A malloc is equivalent to reserving space on the heap.
Local variables are created in assembly language by reserving space on the stack after the parameters.

+ +

A program fragment which demonstrates the creation of a local variable on a stack is:

```
/* fn takes two word size parameters and needs */
/* eight bytes of local storage space */
fn:
    subl $8, %esp          /* reserve space */
    movl 12(%esp), %eax    /* recover p1 */
    movl 16(%esp), %ebx    /* recover p2 */

    /* ... */

    addl $8, %esp          /* reset stack */
    ret
```

The stack for this subroutine:

Address	Stack	
0xffffffff		
	p2 (msb)	
	—	
	—	
	p2 (lsb)	
	p1 (msb)	
	—	
	—	
	p1 (lsb)	
	ret addr (msb)	
	—	
	—	
	ret addr (lsb)	
	reserved	
	reserved	
	reserved	
	reserved	
	reserved	
	reserved	
	reserved	
	reserved	← %esp
0x00000000		

+ +

Assembler Bug

There is a bug in the version of *GNU as* being used for the practical class. This bug manifests itself as follows:

```
Input Code              Generated Code

cmpb $0, %eax           cmpl $0, %eax    <- error
cmpw $0, %eax           cmpw $0, %eax
cmpl $0, %eax           cmpl $0, %eax

cmpb $0, %ebx           cmpl $0, %ebx    <- error
cmpw $0, %ebx           cmpw $0, %ebx
cmpl $0, %ebx           cmpl $0, %ebx
```

The correct code can be generated by rewriting the instruction as:

```
Input Code              Generated Code

cmpb $0, %al            cmpb $0, %al

cmpb $0, %bl            cmpb $0, %bl
```

+ +

Stack Frames

Previously the concept of a local variable was introduced. The mechanism of referencing local variables and function parameters off the stack pointer is quite unwieldy. Each time more space is reserved on the stack or a push or a pop is executed within a subroutine the offsets for the parameters and local variables need to be recalculated. This is tedious and error prone.

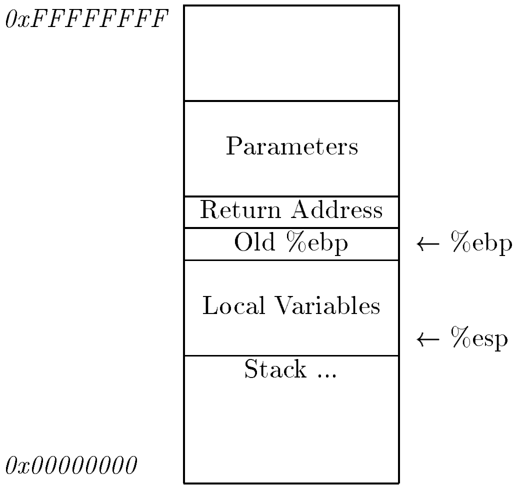
The stack frame is a mechanism which can be used to avoid the deficiencies described. On the 386/486 a stack frame is constructed by using the base pointer in conjunction with the stack pointer.

To create a simple stack frame:

1. Push %ebp
2. Move %esp to %ebp
3. Reserve space on the stack

This results in a structure which has %ebp pointing to a location just above the local variables and a fixed known distance below the parameters. Thus both the parameters and the local variables may be referenced by offsets from %ebp safely. Note also that %esp is free to be used by push and pop.

A simple stack frame may be represented:



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.

+ +

An example The following 'C' program fragment generates a Fibonacci sequence as an example of a recursive program with local variables:

```
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 fict. 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
```

+ +

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 fict. 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 fict. 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
```

+ +

Student Exercise

Write a function called ackerman which is compatible with the following 'C' function.

```
int ackerman(m, n, p)
int m;
int n;
int p;
{
     int res;
     int t;
     if (m == 0)
     {
           res = n + p;
           }
     else if (n == 0)
     {
           if (m == 1)
                res = 0;
           else
                res = 1;
           }
     else
     {
           t = ackerman(m, n-1, p);
           res = ackerman(m-1, t, p);
           }
     return(res);
     }
```

p		+16
n		+12
m		+8
ret addr		+4
old %ebp	← %ebp	
res		-4
t		-8

+ +

```
/* function ackerman takes 3 parameters m n & p */
/* performs ackerman's function */
ackerman:
   pushl %ebp              /* setup stack frame */
   movl %esp, %ebp
   subl $8, %esp           /* reserve local vars */
   movl 8(%ebp), %eax      /* if m = 0 */
   cmpl $0, %eax
   je r1
   movl 12(%ebp), %eax     /* else if n = 0 */
   cmpl $0, %eax
   je r2
                           /* else */
                           /* t=ackerman(m,n-1,p) */
   movl 16(%ebp), %eax     /* p */
   pushl %eax
   movl 12(%ebp), %eax     /* n-1 */
   decl %eax
   pushl %eax
   movl 8(%ebp), %eax      /* m */
   pushl %eax
   call ackerman
   movl %eax, -8(%ebp)     /* store in t */
   addl $12, %esp          /* reset stack ptr */
                           /* res=ackerman(m-1,t,p) */
```

```
   movl 16(%ebp), %eax    /* p */
   pushl %eax
   movl -8(%ebp), %eax    /* t */
   pushl %eax
   movl 8(%ebp), %eax     /* m-1 */
   decl %eax
   pushl %eax
   call ackerman
   movl %eax, -4(%ebp)    /* store in res */
   addl $12, %esp         /* reset stack ptr */
   jmp conc
r1:
                          /* n + p */
   movl 12(%ebp), %eax    /* n */
   addl 16(%ebp), %eax    /* p */
   movl %eax, -4(%ebp)    /* store in res */
   jmp conc
r2:
   movl 8(%ebp), %eax     /* if m = 1 */
   cmpl $1, %eax
   je r3
   movl $1, -4(%ebp)      /* res = 1 */
   jmp conc
r3:
   movl $0, -4(%ebp)      /* res = 0 */
   jmp conc
conc:
   movl -4(%ebp), %eax
   leave
   ret
```

+ +

Data Structures

Both the choice of algorithm and the choice of data structures influence the efficiency of programs. Because of the significant role of data structures in all programming this lecture will be devoted to the details of data structures.
This lecture will cover:

- Implementation of data structures at the assembly code level
- Low level mechanisms for accessing data stored in data structures
- The description of basic data structures

The data structures to be discussed:

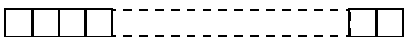
- Vectors
- Arrays
- Records / Structures
- Dope Vectors
- Trees and Graphs

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.
General Representation of a Vector:

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

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



$\uparrow base$

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

The vector was introduced in the section on indexed addressing. 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.

+ +

```
/* 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
```

+ +

Arrays

Vectors are one dimensional arrays. Multi-dimensional arrays will be introduced here.

As the memory of a computer may be viewed as a one dimensional array of storage locations, it is necessary to develop a mechanism that allows the representation of a multi-dimensional array in a one dimensional array.

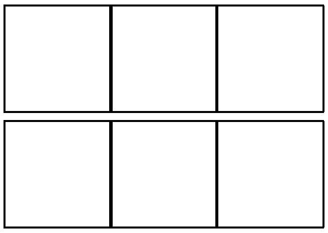
A general mechanism for representing multi-dimensional arrays:

> A multi dimensional array may be considered an array of arrays of one less dimension. A mechanism for implementing a one dimensional array is known. Hence an array of arbitrary dimension may be represented by induction.

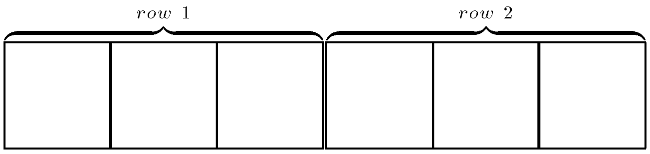
Because of the frequent use of two dimensional arrays, a two dimensional array will be used as an example of multidimensional arrays.

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.

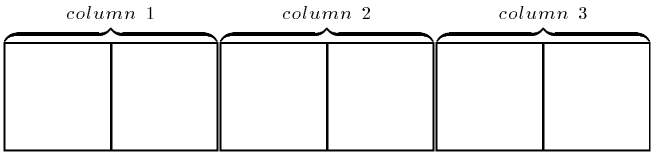
Two Dimensional Array:



Row Major Form:



Column Major 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:

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

+ +

```
/* 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
```

+ +

Records / Structures

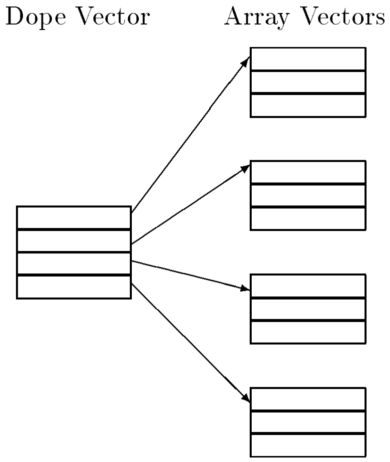
A record is a synonym for structure in the context of computer languages. Structures are manipulated by adding an offset to the base address of the structure to yield the address of the element of the structure to be altered.

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

/* ... */

first.color = 0;

/* 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 color = 8 */
addl $8, %eax
movb $0, (%eax)
```

+ +

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.



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 4, 4, 0
r1:  .fill 4, 4, 0
r2:  .fill 4, 4, 0
r3:  .fill 4, 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
```

+ +

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. As with 'C' it is possible to construct a graph with an arbitrary number of connections by using conventions and careful manipulation of pointers.

The following is an example of a routine to manipulate a graph with an arbitrary number of connections. The routine performs a depth first search on a graph.

+ +

```
.data
root:   .long 0, 0, n1, n2, n3, n4, 0
n1:     .long 1, 0, n5, n1, 0
n2:     .long 2, 0, n3, 0
n3:     .long 3, 0, root, 0
n4:     .long 4, 0, 0
n5:     .long 5, 0, root, 0
.text
dfs:                   /* depth first search */
   enter $4, $0        /* create space for cur ptr */
   movl 8(%ebp), %ebx  /* recover param */
   cmpl $0, 4(%ebx)    /* check for mark */
   jne done
   movl %ebx, -4(%ebp) /* store current pointer */
   addl $8, -4(%ebp)   /* set to first value */
   incl 4(%ebx)        /* mark */
   movl (%ebx), %eax   /* output the node name */
   call printnode
l1:
   movl -4(%ebp), %ebx /* test for end of node list */
   movl (%ebx), %eax
   cmpl $0, %eax
   je done
   pushl %eax
   call dfs
   addl $4, %esp       /* reset stack */
   addl $4, -4(%ebp)   /* increment ptr node list */
   jmp l1
done:
   leave
   ret
```

+ +

Floating Point

In the practical classes and lectures we have used integers exclusively to solve numeric problems. Although a large number of problems can be solved in the integer domain, there are problems which are best solved in the domain of reals. To methods will be introduced to allow the representation of real numbers:

- Fixed point
- Floating point

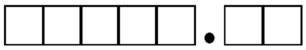
Before describing the formats used in computer systems - formats based on binary representation - some similar formats in decimal notation will be covered.

Decimal Representation

Fixed Point: In our day to day existence we frequently use a fixed point representation for money. The Australian currency is based on the cent - a fraction of a cent is not a particularly useful concept in personal financial transactions. However, the cent is an unwieldy value as it is too small. To make the system easier to use we quote most prices in dollars or multiples of 100 cents. This is an example of a scaled number scheme.

$$100\ cents = 1\ dollar$$
$$15\ cents = 0.15\ dollars$$

For convenience we carry out operations on dollar size values and place the fixed point for our dollar representation inbetween the second and third columns from the right.



A fixed point scheme performs well when the problem domain is based on an indivisible quantity.
Scientific Notation / Floating Point: Scientific notation is closely related to floating point arithmetic. A number written in scientific notation consists of a number multiplied by some power of 10. For example:

$$15 * 10^{6} \;=\; 15000000$$

$$15 * 10^{-2} \;=\; .15$$

Scientific notation allows the representation of a wide range of values compactly.
Limitations: Neither decimal fixed point representation nor scientific notation can specify all numbers. There are some numbers such as π and e (transcendental numbers) and $\sqrt{2}$ (irrational numbers) which cannot be represented precisely in decimal notation without an infinite number of digits. In addition if there are limitations placed on the number of decimal places, the number of significant figures or the size of the powers that scientific notation can have, then both schemes are limited in range and accuracy.

+ +

The Decimal / Binary Point

In decimal notation the fractional component of the number is written beyond the decimal point. The columns beyond the decimal point represent fractions of powers of ten decreasing in size. For example:

$$21.23456_{10}$$

may be written as

$$2 * 10 + 1 * 1 + 2 * \frac{1}{10} + 3 * \frac{1}{100} + 4 * \frac{1}{1000} + 5 * \frac{1}{10000} + 6 * \frac{1}{100000}$$

or

$$2 * 10^1 + 1 * 10^0 + 2 * 10^{-1} + 3 * 10^{-2} + 4 * 10^{-3} + 5 * 10^{-4} + 6 * 10^{-5}$$

A similar representation is available for binary numbers. In the case of a binary number, values beyond the binary point represent decreasing fractions of powers of two.

+ +

For example:

$$10.01101_2$$

may be written as

$$1 * 2 + 0 * 1 + 0 * \frac{1}{2} + 1 * \frac{1}{4} + 1 * \frac{1}{8} + 0 * \frac{1}{16} + 1 * \frac{1}{32}$$

or

$$1 * 2^1 + 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} + 0 * 2^{-4} + 1 * 2^{-5}$$

+ +

Normal Numbers
A number stored in normal form has a value in the position before the point other than zero, but no values in any more significant positions. The following are normal decimal numbers:

$$1.3_{10}$$
$$2.05 * 10^2{}_{10}$$
$$1.4 * 10^{-2}{}_{10}$$

All normalized binary numbers start with a 1 followed by the binary point.

$$1.11001_2$$

Normalizing numbers ensures that the number is both uniquely represented and represented in the most accurate form possible within the representation.

+ +

Floating Point Binary Numbers

A floating point binary number consists of several parts:

s	exponent field	significand field

where

Sign Bit A bit used to indicate the sign of the number. If the sign bit is clear the number is positive otherwise the number is negative. (S)

Exponent Field A number. The number may be biased. The value of a biased number is given by $value = biased\ number - bias$. (E)

Significand Field / Mantissa Field An unsigned normalized number. (F)

In general the value of a floating point number may be calculated using the formula:

$$-1^S * F * 2^E$$

+ +

The following examples have an 7 bit significand a 4 bit exponent with a bias of 8 and a single sign bit. The format of the number is the sign bit followed by the exponent field followed by the significand.

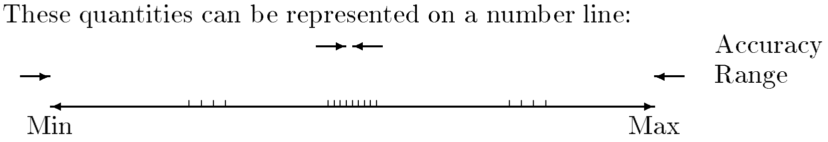
S_2	E_2	F_2	S	E_{10}	F_{10}	$value_{10}$
0	1000	1000000	+	0	1.0	1
1	1000	1000000	-	0	1.0	-1
0	1000	1100000	+	0	1.5	1.5
1	0111	1110000	-	-1	1.75	-0.875
0	1001	1110000	+	1	1.75	3.5

Range & Precision

Range and precision are two parameters which describe a number representation.

- The range of a representation is defined as the greatest and least value that can be represented. Ranges are typically symmetric about zero.
- The precision of a representation is defined by the size of the steps between adjacent values.

These quantities can be represented on a number line:



Typically the range of a representation is specified in terms of the maximum and minimum values represented by the exponent. In addition, the precision is often quoted as the number of bits required to state the number represented by the significand.
If a number is stored in a normalized form it is possible to exploit the property that it has a leading 1 by dropping the leading leading 1 and simply assuming that it is present. This effectively yields an extra bit of precision.

Student Exercises

Using binary floating point numbers with the structure:

1	8	4
S	Mantissa	Exponent

The Exponent is biased by 8.
Write the following as binary floating point numbers:

$$20$$

$$5$$

$$3.125$$

$$12$$

$$1.25$$

$$0.1875$$

$$0.0625$$

$$6$$

$$-1$$

$$-0.125$$

$$0$$

$$-40$$

Answers

Using binary floating point numbers with the structure:

1	8	4
S	Mantissa	Exponent

The Exponent is biased by 4.
Write the following as binary floating point numbers:

$$20 \rightarrow 10100_2 \rightarrow 1.0100_2 * 2^4{}_{10} \rightarrow 0\ 10100000\ 1100$$

$$5 \rightarrow 0\ 10100000\ 1010$$

$$3.125 \rightarrow 0\ 11001000\ 1001$$

$$12 \rightarrow 0\ 11000000\ 1011$$

$$1.25 \rightarrow 0\ 10100000\ 1000$$

$$0.1875 \rightarrow 0\ 11000000\ 0101$$

$$0.0625 \rightarrow 0\ 10000000\ 0100$$

$$6 \rightarrow 0\ 110000000\ 1010$$

$$-1 \rightarrow 1\ 100000000\ 1000$$

$$-0.125 \rightarrow 1\ 10000000\ 0101$$

$$0 \rightarrow 0\ 000000000\ 0000$$

$$-40 \rightarrow 1\ 10100000\ 1101$$

+ +

Properties of Non-Integer Numbers

Fixed Point:

- Has a reduced range of values in comparison to an integer of equivalent size. This is due to the fixed point value being effectively a scaled integer value.
- All combinations of bits represent a value of a fixed point number. Hence there are no wasted bit combinations.
- The accuracy of a value is independent of its magnitude.

Floating Point:

- Typically has an increased range of values in comparison to an integer of equivalent size.
- Not all bit combinations represent different numbers. For example there are a large number of bit combinations which can be used to represent zero.
- The accuracy of a value is dependent on the magnitude of the number. This is due to the uneven spread of values on the number line caused by steps in the exponent value doubling the separation between adjacent values.

+ +

A floating point representation can include additional values known as **NaN**s or *Not a Number* and positive and negative infinity. These values are not ordinary numbers and are used as results when an exception would have to be raised if the NaNs were not available.

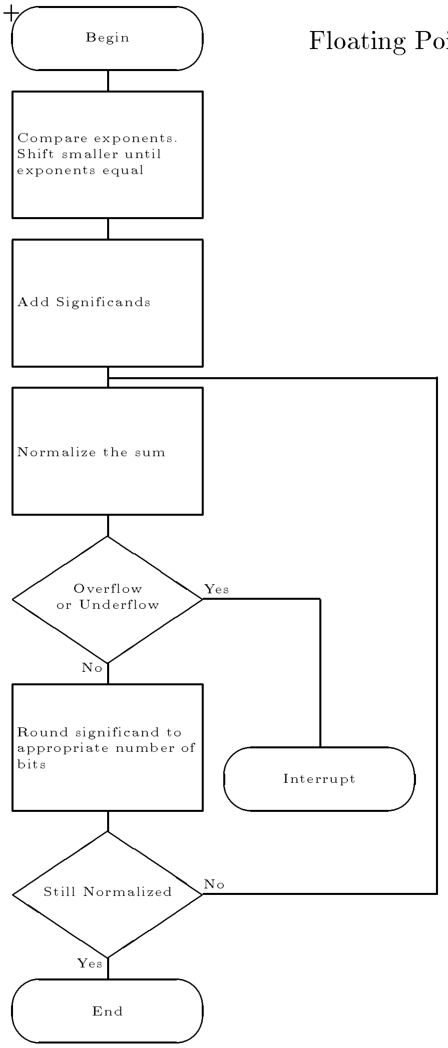
Overflow and Underflow

There are two types of errors which are particularly significant in floating point calculations: Overflow and Underflow.

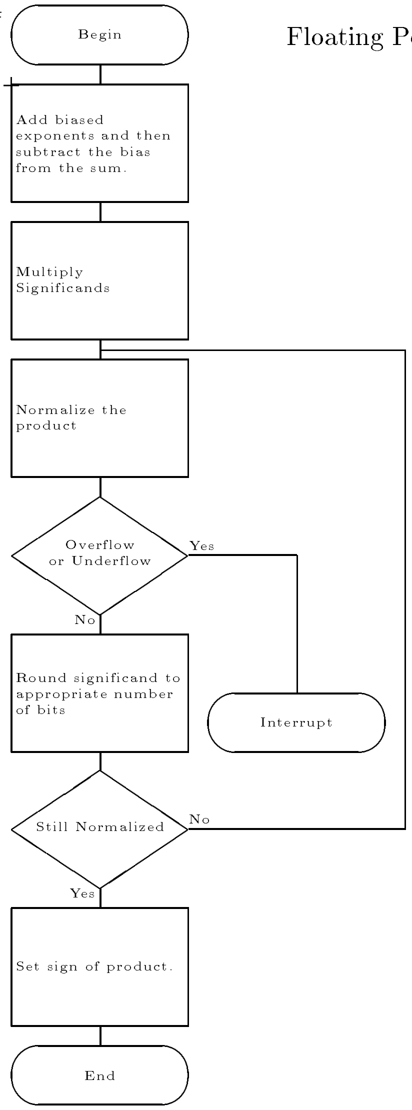
Overflow The result of a calculation cannot be represented because the calculated exponent is too large to be accommodated in the exponent field.

Underflow The result of a calculation cannot be represented because the calculated exponent is too small to be accommodated in the exponent field.

Floating Point Addition



Floating Point Multiplication



+ +

Properties of Operations

Is Floating Point Addition Associative?

No!

Proof: The following is true if addition is associative

$$(a + b) + c = a + (b + c)$$

let the numbers be of the form

$$a = -1^{a_s} * 2^{a_e} * a_f$$

if the lsb of b_f and c_f are set, $b_e = c_e$ and $a_e = b_e + 1$ then
Left hand side: $a+b$ the lsb will not be set and $(a+b)+c$ lsb of mantissa not set
Right hand side: $b + c$ the lsb will not be set but there will be a carry into the next place. Thus one bit of the mantissa from $a+(b+c)$ will be set due to the grouped sum hence the result will be different to the rhs.

Is $-x$ the same as $0 - x$?

Not necessarily.
Because there are two representations of 0 in a sign magnitude system (ie. $+0$ and -0) the following can be expressed:

$$0 - 0 = +0 \; defined \; IEEE \; 754$$

$$-(0) = -0 \; defined \; IEEE \; 754$$

Thus $-x$ is not the same as $0 - x$ when $x = 0$ under IEEE 754.

+ +

Floating Point Numbers
Architectural Choices

A designer of a floating point unit has many degrees of freedom. The choice of the following factors influences performance and capability:
Radix of the Exponent: In all previous examples we have had a base 2 radix for the exponent. This can be generalized to allow for different radix sizes giving a new form for the equation to determine the value of the floating point value:

$$-1^S * F * B^E$$

The consequences of changing B from 2:

- Potential to handle wider range of numbers for same exponent size
- Loss of implied 1 for first digit

Width of Exponent: The wider the exponent field the larger the range of representable floating point numbers
Width of Significand: The wider the significand field the greater the accuracy of representable floating point numbers

+ +

Total Number Width: Although increased accuracy and range is desirable in a number, increasing the total width of the number increases the loading and operation times.

Choice of Rounding Schemes

Truncation All computations become systematically smaller.

Round to Nearest Requires additional precision in the floating point unit, hence slower.

Ordering of Components The component ordering SEF has the advantage that it allows simple magnitude comparisons for most floating point numbers. Other arrangements have been used.

+ +

IEEE 754

The IEEE 754 Standard for binary floating point arithmetic describes a set of formats and operations for floating point numbers. It should be noted that there are both required and optional features specified in the standard and that most of the current computers comply at least partially with the standard.

A short summary of the major features of the standard is provided here.

	Single	Single Extended
Precision†	24	≥ 32
E_{max}	127	≥ 1023
E_{min}	-126	≤ -1022
Exponent Bias	127	

	Double	Double Extended
Precision†	53	≥ 64
E_{max}	1023	≥ 16383
E_{min}	-1022	≤ -16382
Exponent Bias	1023	

†Includes assumed bit for normalized numbers.

The form of the floating point number is:

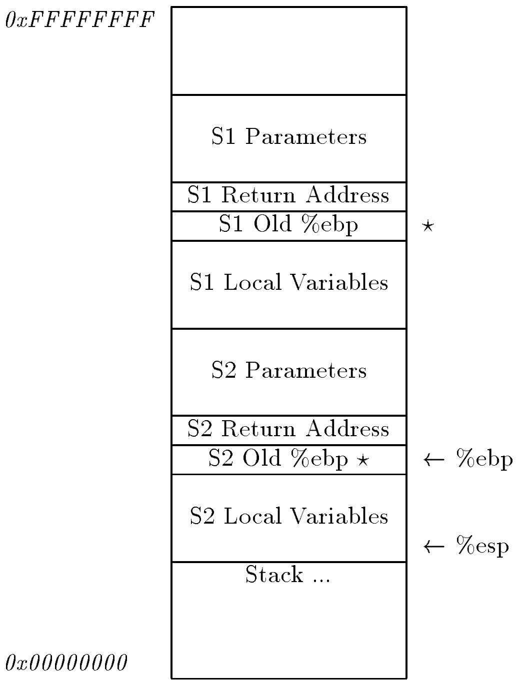
s	exponent field	significand field

A convention is required to store the value zero. This is due to the assumed leading bit for normalized numbers. The convention for representing zero is for the exponent field and the significand to be set to zero. In addition to this there are a number of classes of special values that are represented by setting the exponent field to zero or to all ones. Included in these values are NaNs, positive and negative infinity and denormal numbers. A denormal number (sometimes known as a subnormal number) is represented as a number with a zero exponent field and a non-zero significand. These numbers are used to represent numbers which are smaller than the smallest representable normalized number.

Block Structured Languages

Review of Stack Frames

Before discussing block structured languages we will review the stack frame as used with 'flat' languages. This was covered in the lectures on advanced subroutines.



+ +

Stack frames are regular structures that are constructed on the stack by subroutines. The process of generating one of these structures consists of:

- The calling routine pushing parameters for the called routine onto the stack.
- The calling routine calling the subroutine. (This results in the return address being pushed onto the stack)
- The subroutine performs one of:
    - enter $*size*, $0 (This results in %ebp being pushed onto the stack, %ebp being set equal to %esp and then subtracting size from %esp.

      or
    - pushl %ebp
    - movl %esp, %ebp
    - subl $*size*, %esp

Adopting the stack frame convention for all subroutines makes it easier to write subroutines. The stack frame convention ensures that parameters are accessible by positive offsets from %ebp and local variables are accessible by negative offsets from %ebp. The convention ensures that the offsets required to access parameters and local variables are consistent with respect to %ebp throughout the subroutine.

+ +

Stack frames are destroyed by one of:

- leave
- ret

  or

- movl %ebp, %esp
- popl %ebp
- ret

+ +

Flat & Block Structured Languages

Flat Languages: A flat language

- allows no two functions to have the same name
- all functions are visible to other functions
- has two classes of variables - variables visible to all functions and variables visible only to the function they are named in.

C is a flat language

Block Structured Languages: A block structured language

- may have functions local to functions hence different functions may have the same name.
- has functions which are only visible within their block
- has two classes of variables - variables visible to all functions and variables visible to any function within the block.

Pascal is a block structured language

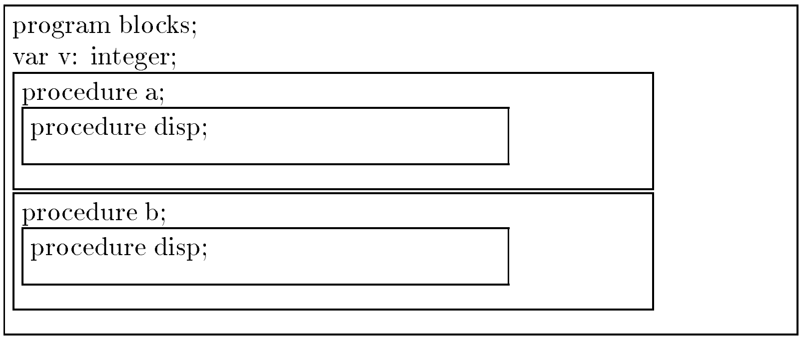
+ +

```
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.
```

Output:

```
a1  a2  b1  b2
```

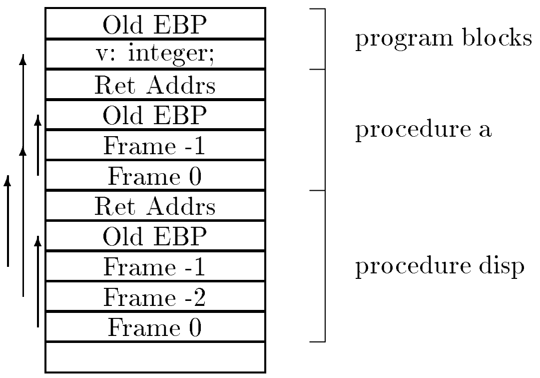
The block structure of the program may be drawn:



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.

Implementing Block Structuring in Assembly Language

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. The example, below, shows the invocation of disp by procedure a.



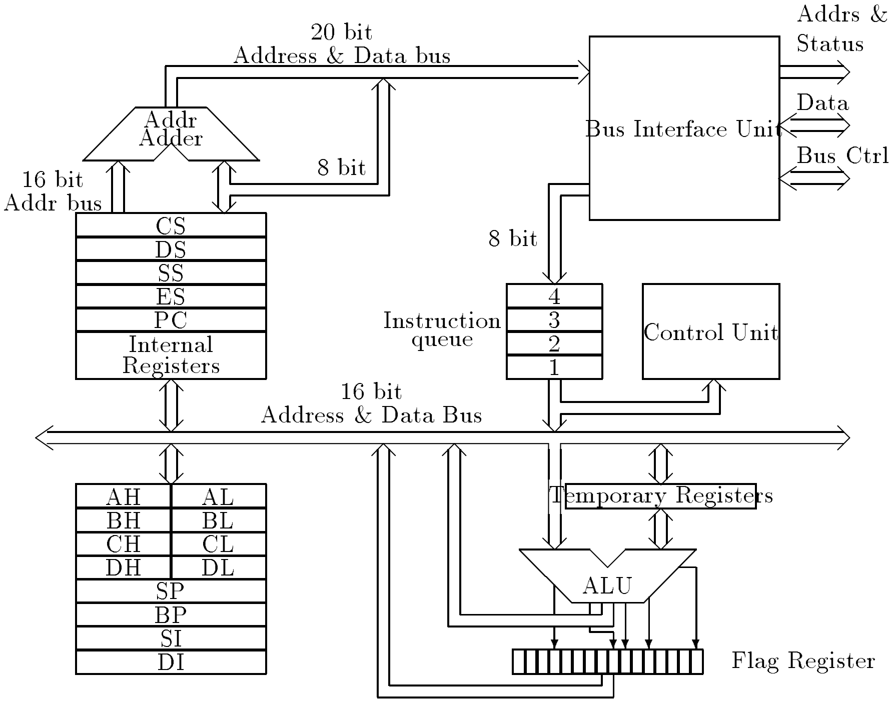
By following back the chain of back pointers it is possible to access any variable in the scope of the current function.

+ +

The Processor

Data Paths

Instead of studying the detail of the 80386 processor's data paths, the simpler data paths of the 8086 will be covered. This simpler processor allows the general concepts of a microprocessor architecture to be discussed without the need to cover the detail of the more complex processor.



+ +

Components

ALU

The arithmetic logic unit (ALU) of the processor performs arithmetic and logical functions. The ALU has two bus inputs which are combined to produce an output. The function provided by the ALU is determined by its control inputs. In addition to performing either an arithmetic or logical function the ALU sets bits in the status register to indicate features of the result.

Register File

The 8086 block diagram shows two logically distinct register files. A register file is a collection of registers. The first of these is a register file containing the general registers, stack pointer, base pointer, and the index registers. The second register file contains the segment registers and the program counter.

Control Unit

The control unit decodes the instruction stream and co-ordinates the activities of the processor.

Buses

A bus is a collection of conductors. Note that a conventional bus may only be 'driven' by one device at a time, although many devices may observe the state of the bus. A device is said to 'drive the bus' if it is a source of current for the bus. To observe the value of the bus it is necessary to sink or consume some current from the bus.

+

Other Elements

Instruction Queue The instruction queue buffers four bytes of instructions.

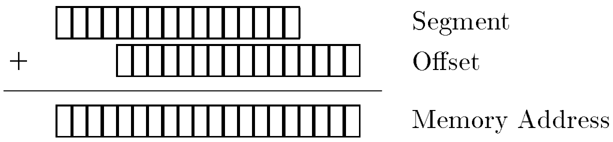
Temporary Registers These registers are used to hold data items to be fed to the ALU.

Address Adder This is used to form the 20 bit address used by the 8086 to access memory.

Bus Interface Unit The bus interface unit fetches data and code from memory. The bus interface unit also monitors the interrupt and other control lines.

Addressing

The 8086 supports a 20 bit address space by combining a 16 bit address with a 16 bit segment register. The 16 bit segment register is left shifted by 4 bits and is added to the 16 bit address to yield a 20 bit address.



This operation is required each time an address is to be output to the bus.

+ +

Fetch-Execute Cycle

The basic operation of the 8086 is dominated by a single sequence of operations. This sequence is known as the fetch and execute cycle. The following descriptions give the flavor of the operations and how they might be implemented. Note that this is not a precise description of the operation of the 8086 chip. The invariant part of this cycle over all instructions is:

1. Output PC to Address Adder

   Output CS to Address Adder

2. Output 20 bit address to system bus

   Increment PC

3. Store returned bytes in Instruction Queue

4. Decode first byte of instruction

The remaining steps of the cycle vary with the instruction. If the instruction is a multi-byte instruction then further bytes are read from the instruction queue.

Two example the sequences will be illustrated: incrementing a register and adding a register and an integer will be covered. These examples exercise most of the data paths through the processor.

Increment AX:

5. Output enable AX onto the 16 bit Address & Data Bus

   Input enable the ALU

   Select increment function

6. Input enable AX onto the 16 bit Address & Data Bus

+ +

Add AX and location 1004 leaving the result in AX:

5. Output enable AX onto the 16 bit Address & Data Bus

   Input enable a Temporary Register
6. Output 1004 to Address Adder

   Output DS to Address Adder
7. Output 20 bit address to system bus
8. Store returned bytes in Instruction Queue
9. Output enable Instruction Queue

   Input enable ALU

   Select add function
10. Input enable AX onto the 16 bit Address & Data Bus

+ +

Instant Revision of Digital Logic

Combinatorial Logic

In combinatorial logic the result is determined only by the input values. The basic operators of combinatorial logic are:

NOT logical not

AND logical and

OR logical or

These operations can be implemented directly in hardware gates.

Data Storage Elements

A general logic circuit may contain data storage elements - the most simple of the elements are called flip-flops. These data storage elements store an input value and output that value on demand.

Sequential Circuits

A sequential circuit consists of combinatorial logic being used to process the inputs and outputs of data storage elements. There is a special class of sequential circuits known as *synchronous circuits*. A synchronous circuit has all data storage elements output data at the same time. We will only refer to synchronous circuits in this course.

+ +

Control

In the previous lecture the general structure of the processor was discussed and the sequence of events required to implement two example operations were discussed. This lecture will cover the mechanisms for implementing those sequences.
Regardless of the implementation, the control of the processor, performs the following:

- Decode the instruction
- Generate the sequence of control signals that implement the instruction.

Decoding Instructions
Typically instructions are composed of fields of bits which control part of the operation of an instruction. Early in the course the instruction format for the 386/486 was discussed.

Instruction Prefix	Address Size Prefix	Operand Size Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
	bytes		

Opcode	MOD R/M	SIB	Disp	Imm
1 or 2	0 or 1	0 or 1	0 1 2 or 4	0 1 2 or 4
		bytes		

The most notable feature of the format is that it permitted variable length instructions. To accommodate this the encoding of the components was selected to indicate if there was another part of the instruction following.

This can be illustrated by looking at the opcodes that are used by the add instruction (taken with modification from i486 Microprocessor Programmers Manual, Intel, 1990):

04 *ib*	ADD *imm8*, AL
05 *iw*	ADD *imm16*, AX
05 *id*	ADD *imm32*, EAX
80 */0 ib*	ADD *imm8, r/m8*
81 */0 iw*	ADD *imm16, r/m16*
81 */0 id*	ADD *imm32, r/m32*
00 */r*	ADD *r8, r/m8*
01 */r*	ADD *r16, r/m16*
01 */r*	ADD *r32, r/m32*
02 */r*	ADD *r/m8, r8*
03 */r*	ADD *r/m16, r16*
03 */r*	ADD *r/m32, r32*

The fields of an instruction are more clearly seen in a fixed length instruction set. The following example, the R2000 encoding, is taken from Henessy & Patterson.

+ +

R-Type instruction:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

I-Type instruction:

6 bits	5 bits	5 bits	16 bits
op	rs	rt	address

op	operation
rs	source register
rt	second register
rd	destination register
shamt	shift amount
funct	variant of operation
address	address

With this simpler instruction format it can be easily seen that the bits of the instruction are grouped and simple combinations of the bits are used to determine the function, the source and destination of the operation. As with the 486 instruction set the bits of the opcode (op field in the case of the R2000) determine the interpretation of the rest of the bits of the field. Specifically, the value of the OP field determines if an I-type or an R-type instruction format is used.

+ +

Sequencing Control Signals

There are 2 distinct classes of control units:

- Microprogrammed Control Units
- Hard-Wired Control Units

Microprogrammed Control

A microprogrammed control unit behaves similarly to the processor itself. Essentially a microprogrammed control unit has a store of microinstructions which it fetches and executes to control the processor.

In a microprogrammed control unit

- an instruction is represented as a series of microinstructions
- each microinstruction consists of several fields there are 2 classes of fields
  - fields which correspond to control fields
  - fields which determine which microinstruction to execute next
- a hardware register called the μPC is used to point to the next microinstruction to be executed

+ +

An imaginary microinstruction format:

	Control Field	Sequence Field
000	1000000	001 000 000
001	0100000	010 000 000
010	0100000	000 010 000

A simple microprogrammed control unit could generate the following sequence:

- Assert line to output PC
- Load returned result into Instruction Decode Register
- Jump to location in microcode indicated by decoded instruction

...

- Set μPC to zero

Decisions are made in the microcode by controlling which instruction is next to be executed. There are a number of mechanisms available to implement this:

- Skip next instruction if condition met
- Set μPC to value if condition met
- Set μPC to one of a set of values depending on condition

Microprogrammed control units may be broken up into 2 classes:

Horizontal control units These resemble the simple control units in that the microinstruction is wide and contains a bit for each control line

Vertical control units Are narrower than horizontal control units as the instruction is divided into fields which are either decoded or translated into the control signals.

A real world example: 8088

- 504 microinstructions
- 21 bits wide

5 *bits*	5 *bits*	3 *bits*	4 *bits*	3 *bits*	1 *bits*
src	dest	type	ALU	reg	cc

- Length of sequence: Average 5 or 6, Maximum 16

+ +

Hard-wired control units
A hard-wired control unit is essentially a synchronous circuit which produces the control sequence which fetches instructions and then generates the sequence which corresponds to the fetched instruction.

A comparison of the two mechanisms
Microprogrammed control units

- require relatively simple logic to construct the controller
- majority of the work is concentrated in producing correct microcode for the instruction set
- microcode is relatively easy to generate and check - it is regular and often repeated
- microcode can be represented symbolically making it easier for humans to understand

Hard-wired control units

- allow a more flexible and hence optimizable approach to design
- it is easier to construct parallel operations in a hard-wired control unit
- are hard to verify
- are complex
- are not necessarily regular
- difficult to represent symbolically

At present hard-wired control units are the preferred for:

- high-performance machines such as super computers - the need for maximum performance
- simple RISC CPUs - a sufficiently simple design to allow the faster implementation

+ +

Microprogrammed control units are used in:

- the last generation of CPUs - 8086, 80286, 68000 - simplifies design
- very complex instruction set machines - VAX - the cost of attempting to implement a vast number of instructions in hard-wired control cannot be justified in a medium performance computer

It is now common to adopt a hybrid design approach which has complex instructions implemented using microcode and simple instructions implemented in hard-wired control. The convergence of architectures will be discussed in a future lecture.

+ +

Interrupts

In this lecture we will be covering:

- Interrupts & Traps
- Implementation of interrupts
- Masks & Priorities
- Non-maskable interrupts
- Interrupt Service Routines (ISRs)

Key Definitions

Interrupt Interrupts are a form of forced procedure call which are caused by an event external to the program. Interrupts are *asynchronous* to the program. Interrupts are typically caused by a peripheral asserting a wire connected via some circuitry to an interrupt pin on the processor.

Trap A trap is a form of forced procedure call which is caused by an exceptional event within a program that has been detected by the processor hardware. Traps are *synchronous* to the program. An example of an event that might cause a trap would be arithmetic overflow.

+ +

Implementation

The typical implementation for a microprocessor based interrupt mechanism is for the processor to check for an external interrupt condition at the end of or the beginning of an sequence that implements an instruction.
In a microprogrammed control unit this would correspond to a branch to the microcode to implement interrupts at the end of the sequence of microinstructions that implements an instruction if an interrupt signal is present.
This mechanism does not require the processor to restart an instruction part way through.
Interrupts are implemented as a forced procedure call. This means that after the interrupt condition is noted at least the following actions are required: a return address is stored and the interrupt routine called.
There are two major implementation forms:

- Mechanisms which use an *interrupt vector*
- Mechanisms which use *vectored interrupts*
- Mechanisms which use a status register

These mechanisms differ in the mechanism in which they convey to the *Interrupt Handler* the cause of the interrupt.

Interrupt Vectors
The destination addresses for each type of interrupt are stored in a table. When an interrupt occurs the processor makes a forced procedure call to the location indicated in the interrupt vector.

Vectored Interrupts
In a vectored interrupt system the location called by the processor is determined by the cause of the interrupt. Typically this is implemented as a set of addresses a fixed distance apart which the processor calls when an interrupt occurs.

Status Register
In this case a processor jumps to a single address when any interrupt occurs. It is the task of the interrupt handler to consult the status register to determine the cause of the interrupt. This register is called the *cause* register in the MIPS architecture.

+ +

Two Examples - 80386 & R4000

80386

The 80386 uses an interrupt vector with 256 entries. When an interrupt occurs:

- push EFLAGS onto the stack
- push Instruction pointer onto the stack
- clear interrupt flag
- the processor jumps to the location indicated by the interrupt type

R4000

The R4000 uses a single entry point which is jumped to when an exception occurs (the exact memory location depends on the operating mode of the processor).
When an interrupt occurs:

- the EPC (exception program counter) is loaded with the current program counter value
- the bit in the Cause register corresponding to the interrupt value is set
- the processor jumps to the interrupt handler

+ +

Masks & Priorities

As an interrupt can effectively occur at any time within a program's execution they are essentially unpredictable. At times it is inconvenient to have to handle an interrupt. An interrupt mask is used to prevent an interrupt having an effect.
An interrupt mask is a set of bits which - if set - allow the interrupt to be noticed by the processor.
Under what circumstances is it necessary to not notice an interrupt immediately?

- While executing a time critical routine
- During the initial phase of handling another interrupt
- When handling a more important activity

The first two of these problems are solved by setting the mask to prevent interruption while performing critical tasks. The second is solved by introducing priorities.
In a priority based interrupt scheme each interrupt is allocated a priority.

- If the currently executing interrupt handler has a lower priority than a new interrupt then the currently executing interrupt handler is pre-empted and the new one commenced.

+ +

- If the currently executing interrupt handler has a higher priority than a new interrupt then the new interrupt is recorded and dealt with when all higher priority interrupt handlers have completed.

Non-maskable Interrupts
Processors frequently have an interrupt that cannot be masked out or ignored. This is typically used to indicate a failure in a critical component of the computer. A classic example would be a power failure detection. If the power fails the system should attempt to perform necessary house keeping before function is fully lost.
Only critical functions should use the NMI as there is no mechanism to ensure the correct return from interrupt after an NMI.
Real time systems
A real time system needs to respond in a predictable way to any set of input conditions. As noted in the section on masks a problem arises when an interrupt pre-empts an interrupt in a stack based system. This is caused by the requirement for saving state on the stack. Some real time systems solve this problem by having an area of storage assigned to each interrupt routine and storing the state in that region when an interrupt occurs. This ensures that there is always a place for the state to be stored that is rapidly accessible. In extremely critical applications hardware assistance is provided for saving the state.

+ +

Interrupt Service Routines

An Interrupt Service Routine (ISR) performs the following operations

1. Save all registers
2. Save any status information relating to the cause of the interrupt
3. Enable higher priority interrupts
4. Perform required service
5. Clear the cause of the current interrupt
6. Return from interrupt

The first two tasks fall in a *critical section*. By ensuring that this information is correctly saved it is possible to be interrupted and returned to *transparently*.

The mechanism described is sufficiently general and conservative to be used on both normal and real time systems. In a normal system the data in 1 and 2 would be stored upon the stack. In a real time system the information would be stored in a location local to the ISR.

Transparency Transparent code stores the necessary process state and register values to ensure that when it returns the interrupted code is unable to determine that an interruption has occurred

Critical Section A critical section is a section of code which must be executed atomically.

+ +

Input / Output

Why I/O is important:

- Few jobs are completely CPU bound thus nearly all jobs depend on I/O performance to limit at least part of their operation.
- The speed of I/O is orders of magnitude slower than the speed of the processor. Minor mismanagement of I/O has a greater impact than minor mismanagement of the CPU.

Types of Devices

There are two major classes of devices:

Block Mode Transfers are made in blocks. A block is defined as a regular structure with some maximum size and some minimum size. Typically block mode devices employ blocks of exactly the same size. If error detection is employed then processing of the information within a block cannot commence until the full block is available. Examples of block mode devices: Disk drives & Network Controllers.

Stream Mode This is a generalization of character mode. Information is transferred in typically byte sized quantities and each item is processable immediately on reception. Examples of stream mode devices: Terminals & Serial devices.

+ +

Device Properties

Block mode devices place a lower load on the system resources on a per byte basis. This is because the system needs only to notice and handle a transfer on the completion of a block transfer. In contrast, a stream mode device requires the system to intervene with the arrival of each new item.
Block mode devices tend to have better error correction than stream mode devices as stronger error detection and correction techniques can be applied.
Software and hardware can convert the behavior of a stream mode input to partly resemble block mode input or vice versa. Although this conversion is possible the properties of the underlying device cannot be completely masked. In many cases attempting to a hide the mode of access results in undesirable behavior.

Interrupts & Polling

There are two major mechanisms available to the programmer for interacting with external devices:

- Interrupts
- Polling

+ +

Interrupts an external event causes an interrupt the ISR deals with the device.

Each time an interrupt occurs the processor performs the sequence of operations described in previous lecture. This sequence of events is known as the overhead of the interrupt as it occurs each time an interrupt occurs and does not directly contribute to the handling of the event.

Polling The processor regularly checks each device and notes the state of the device. If the device requires servicing the required operations are carried out

For polling to work it is necessary to ensure that the *polling loop* can be completed sufficiently quickly that the fastest device will not have made more than one request in the time it takes to go round the loop handling all possible requests. Failure to do this leaves the system vulnerable to data overruns.

+ +

Comparison

- Polling has lower overhead
- Polling is capable of greater throughput
- Polling requires greater use of the system
- Polling is inflexible
- Polling must be designed for the worst case
- Interrupts are more flexible but must handle unusual cases such as a repeated interrupt.
- Interrupts place a lower load on a system when infrequent.

+

+ +

DMA & Co-processors

These mechanisms allow part of the work associated with dealing with a device to be done by an entity other than the processor.

DMA

A DMA or Direct Memory Access device is essentially a simple processor attached to the memory bus of the system. DMA devices may be used to accomplish several tasks:

- Memory to Memory - a DMA device can copy a block of memory from one point in the system memory to another with minimal processor intervention.
- Device to Memory - A DMA device can be used to interact with a device to copy the results returned by the device into the systems memory. Both block mode and stream mode devices may be interfaced this way.

As the DMA device shares the bus with the processor the action of the DMA device affects the ability of the processor to interact with memory. The DMA device and the processor arbitrate for the memory bus. The effect of this is that neither processor nor DMA device can have unrestricted use of the bus, but due to the burst nature of bus traffic the delay introduced by the sharing of the bus is not proportional to the usage of the bus.

+ +

The design and parameters of DMA devices vary from unit to unit, however, there are common features to the designs.
The common operations to DMA controllers are the need to setup the device to transfer and to inform the system that the transfer is complete. The information required to setup the transfer is the source of the data, the destination address and the size of the transfer. At the completion of a DMA operation the processor is notified by an interrupt raised by the DMA controller.

Co-processors

DMA devices are a special restricted form of co-processor. High performance I/O devices may have significant autonomy. In this case the device carries out a sequence of operations based on a sequence of high level instructions passed from the main processor. Examples of this type of device are graphics co-processors which accept lists of operations to carry out and intelligent serial interfaces which poll a large number of serial ports and make available the ports information as blocks of characters.

+ +

Architectural Consequences

The choice of the mechanism for handling I/O devices depends on a number of factors:

- Mode of device
- Data rate
- Tolerable latency
- Available hardware

Where large amounts of data are to be transferred or a high data rate is required either a block mode device or a DMA controller should be provided. This reduces the load on the processor by ensuring that the processor deals with large lumps of information. The alternative of having to keep up with a large number of interrupts or a tight polling loop would have a detrimental effect on the performance of the machine.

Where the volume of data is small or the data rate is low then the processor can in general be used to handle the type of operation using interrupts or polling started by a timer interrupt (A regular timer tick interrupt occurs and the system polls the required devices).

If fast response to asynchronous inputs is required - low latency - then interrupts should be used.

Frequently the type of access mechanism is determined by the existing hardware and the improvement of performance of using additional hardware is traded for the reduced cost of using existing - less optimal - hardware.

+ +

Memory

The importance of Memory:

- The majority of the frequently used data in the computer is stored in memory.

Thus the data rate from memory typically limits the speed of CPU bound processes. The data rate of memory also limits the performance of I/O devices as - in general - I/O devices transfer data to and from memory.
Principles

Temporal Locality The principle of temporal locality states that if an item has been accessed recently it is likely to be accessed in the near future.

Spatial Locality The principle of spatial locality states that if an item has been accessed it is likely that items close to it will be accessed in the near future.

Motivation for Caching
Approximate RAM costs:

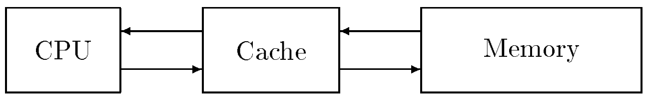
Type	Speed	Cost	Size	$/M byte
SRAM	25nS	$29	256K bit	928
DRAM	80nS	$160	1M byte	160

In general: Fast memory is more expensive than slow memory.

A processor operating at 33MHz has a maximum demand on memory of 1 item every ~30nS. Thus if the complete memory was required to keep up with the processor the system would be made up of very expensive RAM.
If it were possible to store more frequently used items in faster memory and less frequently used items in slower more expensive memory then performance could be improved so that it approached the fast memory speed and cost reduced to approach the slow memory cost.

Cache Operation

The cache sits between the CPU and the main memory and contains a quantity of fast RAM which is used to store recently accessed information.



The cache deals with two types of access:

- Processor reads
- Processor writes

Both accesses are prefixed by the processor sending an address to the cache.

+ +

Processor Reads:

- If the cache contains a copy of the location: The cache returns a copy of the information to the processor
- If the cache does **not** contain a copy of the location: The cache forwards the address to the memory. On completion of the memory request, the cache stores the returned result and passes the result to the processor.

Processor Writes:

- The cache stores a copy of the location's contents and subsequently forwards a copy to the memory.

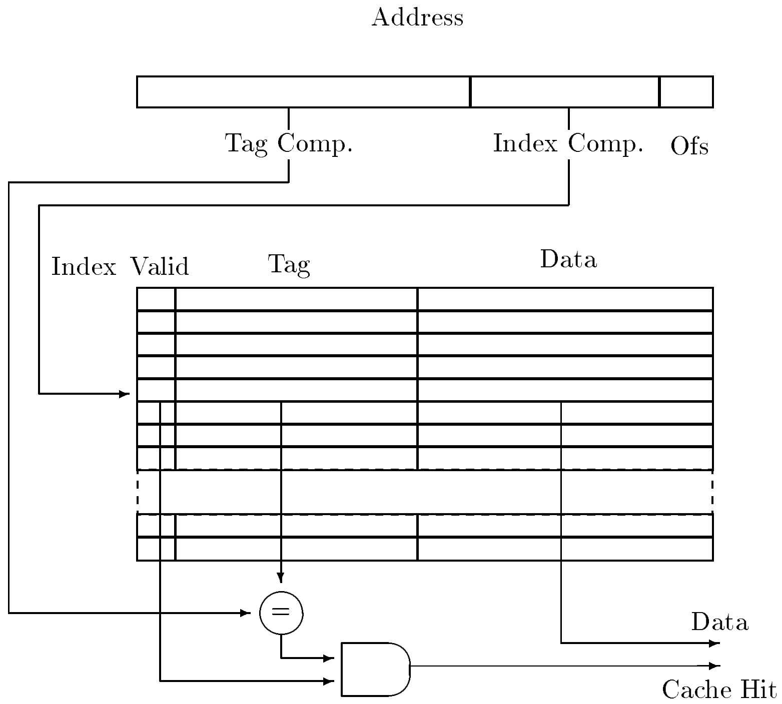
There are two major variants of caches available which are distinguished by their behavior on writes:

Write Through The data is written into the cache and into the next stage of the memory hierarchy simultaneously.

Write Back The data is written into the cache only. Only when a cache flush is issued is the data written to the next stage of the memory hierarchy.

+ +

Cache Implementation



Part of the address is used as an index into the cache's table. If the tag and the upper component of the address match and the cache entry is valid then a cache hit is said to have occurred and the value is returned to the processor or updated in the cache depending on whether the operation is a read or a write.

+ +

Exploiting Locality
The principle of locality implies that a cache yields profit if it holds recently used information and information close to recently used information. This is exploited by the expedient of replacing items that hash to the same location in the table each time a new item is encountered and extending the cached data beyond the accessed word to a set of words adjacent to the fetched word.
Even simple caching schemes yield high cache hit rates more complex multiway associative schemes yield more marginal improvements for a given amount of cache memory.

+

+ +

Virtual Memory

- Modern processors have a large address range
- Modern processors have a small amount of memory compared to address range
- Programmer's frequently want to express programs that will not fit into main memory completely at one time.

Virtual memory is a mechanism which allows programmers to use a large portion of the processor's address space and load the required components of a program on demand.
There are 2 mechanisms which can be used for virtual memory:

Paging Under paging the memory is divided into equal sized objects know as pages which are paged into and out of the physical memory to disk.

Segmentation Under segmentation objects called segments of a size selected by the programmer are swapped into and out of the memory of the computer to disk.

Both mechanisms employ an additional layer of translation to convert an uttered address into a physical address.

+ +

A program larger than the physical memory is made possible as when a program utters either the name of a segment not in memory or the address of a page not in memory a page or segment can be copied to disk and the required page or segment brought in.

Paging

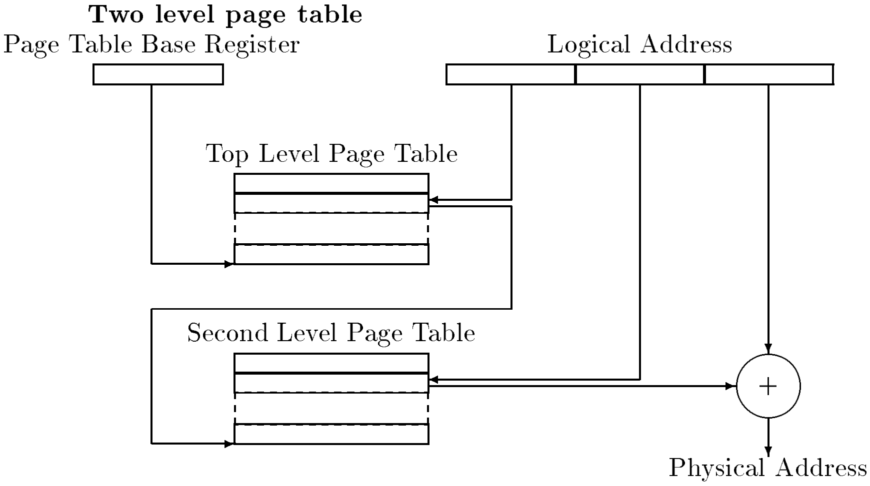
In a paged system a page table is used to translate logical addresses into physical addresses. The translation consists of:

- The more significant part of the logical address is used as an index into the page table
- The physical address of the base of the page is extracted from the page table
- The less significant of the logical address is added to the base of the page to give the physical address in memory of the reference

In practice multi level page tables are used to conserve the amount of memory consumed by the page tables. A multi level page table requires only the active page tables to be present in memory.

+ +

Two level page table



All address translation for paging is carried out transparently by the hardware.

Segmentation

As previously mentioned in these lectures a segment is defined as

> A section of memory denoted by a base address, an extent and a set of rights.

Hence all accesses in a segmented system are expressed as offsets within a segment.

\+ +

In a segmented system the segments are loaded into memory to allow process's to access information contained in the segment when a process names a segment it wishes to load.

Fragmentation

Fragmentation occurs when there is unused or unusable space in the memory of the system.

Internal Fragmentation lost space is contained in unused parts of a loaded element.

External Fragmentation space is lost because uneven sized objects cannot be loaded in a way to ensure that all the space is filled.

Both paging and segmentation suffer form fragmentation. Paging suffers from internal fragmentation. Segmentation suffers from external fragmentation.

External fragmentation is one of the causes for segmentation's unpopularity as a memory management technique. Currently Intel is the only manufacturer bringing out new microprocessors that support segmentation. The management of external fragmentation adds complexity to an operating system and forces part of the task of space management onto the programmer.

+ +

Memory Protection

Both paging and segmentation allow access rights to be attached to the page or segment. Typically these access rights specify that the data contained may be accessed by:

- read
- write
- execute
- privileged code only

The Memory Hierarchy

- Registers - reside in the processor.
- On chip memory - small primary cache on processor chip (in the order of 8K bytes size, $< 10nS$).
- Off chip caches (in the order of 100K bytes to 4M bytes size, $15nS$).
- Main memory (in the order of 10M bytes, $< 80nS$)
- hard disks or secondary storage (in the order of 1G bytes)

+ +

Advanced Topics

The topics:

- Pipelined processors
- Superscalar processors
- The RISC / CISC controversy

Pipelining and Superscaling are techniques for increasing the throughput of a microprocessor by introducing parallelism into the processor.
The RISC / CISC topic covers the definition, differences and motivations behind the two major processor types.

Pipelining

- A technique for overlapping instruction execution
- Breaks the execution of an instruction up into phases
- A step in the pipeline processes that phase of the instruction and allows the instruction to proceed to the next step in the pipeline.
- This allows as many instructions as there are pipeline steps to be in execution at one time.

+ +

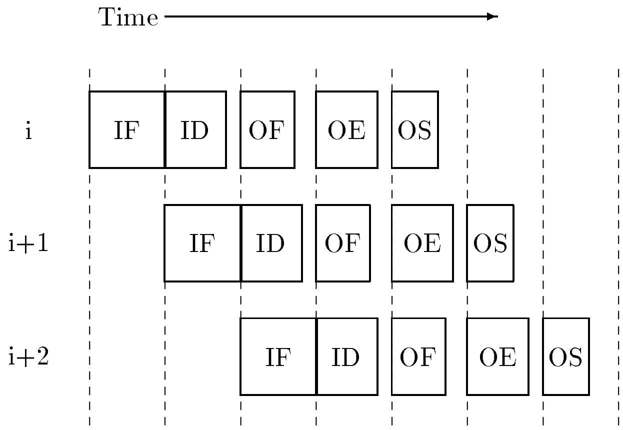
Pipelining is similar to a production line. In a production line instead of devoting a single worker to completing the full task of creating a product from start to finish, a group of specialised workers work on a single part of the process. Only after the product has passed through all steps of the process is it complete.
Balancing a pipeline:

- The speed of a pipeline is restricted by the throughput of the slowest element of the pipeline
- A balanced pipeline has all the stages take the same time

The effect of pipelining is to improve the throughput of the processor by increasing the number of instructions processed per unit time. Pipelining does **not** decrease the time an individual instruction takes.

Pipeline Implementation

Pipelines may vary in length and in the tasks assigned to each step in the pipeline. An example of the division of pipeline stages and the overlapping of execution.



Steps in the pipeline:

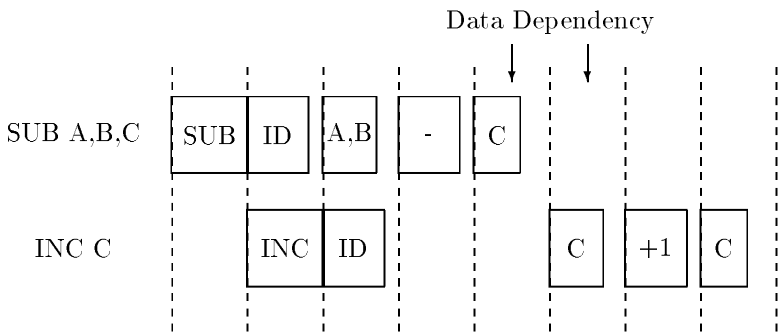
IF	Instruction Fetch
ID	Instruction Decode
OF	Operand Fetch
OE	Operand Execution
OS	Operand Store

+ +

Data Hazards

Data dependency in a pipeline arises when the input of an instruction depends on the output of another instruction, both of which are executing in the pipeline. A number of mechanisms have been suggested to deal with this problem:

- Stall the pipeline. This involves stopping the progression of instructions entering a pipeline at the stage of the pipeline a dependency has been detected at and only resuming the progress of earlier stages after the dependent value has been set.
- Ignore the problem and have the compiler reorder the instructions, where possible, and insert no-ops where necessary to prevent data dependencies in the pipeline.



+ +

Data Interlocks:

- Detect dependencies
- Add complexity to the processor
- Test if output is used as input in a later stage of pipeline
- Need to be applied to both registers and addresses to ensure correct operation
- Slow the logic of the processor down

Data Forwarding:

- Reduces the effect of a pipeline stall
- Passes operation results back to earlier stages instead of waiting for save phase to complete.
- Reduces stall time & adds to processor complexity

Branch Hazards
A branch hazard occurs when a jump occurs. There are two varieties of branch hazards: conditional and unconditional. In the case of an unconditional jump input to the pipeline is stalled until the correct address for the jump is determined.

In the case of a conditional jump two solutions are possible for handling a branch hazard:

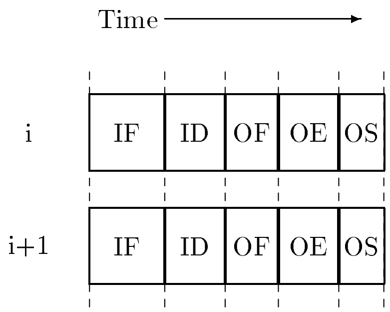
- continue executing the following instruction but be prepared to throw away any of its consequences if the jump occurs
- stall the pipeline until the destination address is known

The former mechanism increases processor performance at the expense of increasing the complexity of the processor.
An additional mechanism is available for both conditional and unconditional jumps: The **delayed branch**. This mechanism redefines the branch instruction to take place one instruction after the branch instruction. This means that the pipeline will never have to stall on a branch. This technique requires a compiler to re-order instructions to ensure that there is a suitable instruction or no-op after each branch.
The use of compiler techniques to avoid pipeline stalls caused by branch hazards or data hazards is advantageous regardless of the hardware support provided for coping with the hazard when it arises Avoiding pipeline stalls maintains the throughput of the processor.

Superscalar Processors

A superscalar processor contains multiple functional units. The functional units may be of the same or different types. In a superscalar architecture more than one instruction can be *issued* at a time provide the instructions are independent. The superscalar mechanism differs from the pipeline mechanism in that it is necessary that the operation of a dependent instruction not be commenced.

Instruction execution in a two issue superscalar processor:



The absence of a suitable instruction indicates that a given functional unit should execute a no-op.

Functional units of different types:

- Are the simplest mechanism for ensuring that instructions issued are not dependent
- Separate floating point and integer units are a classic example.
- simplify the control of the processor as the order of instruction types can be specified and the functional units loaded in order.

Similar functional units:

- Can have dependent instructions
- Require mechanisms similar to the interlocks found in pipelined systems to cause one or more functional units to execute a no-op to prevent the dependent instructions being executed at the same time.

To gain maximum benefit from superscalar processors it is necessary to supply independent instructions to each of the functional units and avoid idle time in any unit. This requires that the compiler organize the instructions into an order which allows parallel operations to take place.

+ +

RISC / CISC Controversy

CISC

The characteristics of a Complex Instruction Set Computer (CISC) are

- a rich instruction set
- many addressing modes

CISC machines were motivated by microcode.

- The ease of writing microcode to implement an instruction resulted in the perception that adding instructions to microcode yielded an improvement to an instruction set at relatively low cost.
- Microcode is stored in small high speed memories and hence it was perceived that microcode executed faster than assembly code.
- Instruction set designers added more complex instructions to performed common operations desired by programmers.
- Finally there was a perception that compilers were complex and by giving the compiler a wider choice of operations and addressing modes it would simplify the design of the compiler.

+ +

RISC
The characteristics of a Reduced Instruction Set Computer (RISC) are:

- Simple instructions
- Uniform instruction length
- Few instruction formats
- Orthogonal instruction set
- Few addressing modes
- Load-Store Architecture
- Few data types
- Many registers

RISC processors were driven by the invention of caches, an improvement in compiler technology, and the desire to reduce the complexity of the processor architectures to simplify the task of introducing pipelining.

+ +

RISC:

- Formed on the observation that the majority of the work of a processor was done by a minority of the instructions.
- An overall speedup could be had if the frequently executed instructions were improved.

The steps to RISC:

- The introduction of hybrid microprogram and hardwired controllers increased processor complexity.
- an improvement in compilers and the introduction of caches.
  - caches made with the same technology as the control store allowed instructions to execute at the speed of microcode
  - Improved compilers were hampered by more complex instructions and multiple addressing modes. The compiler was forced to choose the best mode and instruction from a wide range of possibilities.

The RISC processor aimed to provide a simple set of instructions that worked quickly. In addition the processor was designed to be simple so that the more flexible hardwired control units could be designed at reasonable cost.

The load-store architecture simplifies the design of the processor by limiting memory access to only the load and store instructions. All other instructions have registers as both source and destination.
The regular instruction length simplifies control and decoding logic.
Comparison

- RISC & CISC designers driven by same goal of maximum useful work.
- RISC processors commonly include a microcode based floating point unit because it yields better performance than attempting to perform the same operations in machine code.
- CISC machines have hardwired control for some of their instructions to achieve single cycle execution on some instructions.
- The mechanisms of pipelining and superscaling are applicable to both architectures, although more difficult to apply to a CISC processor.
- the distinction between the RISC and CISC processor of today is becoming more blurred.

+ +

Measures of performance:

- simple measures of MIPS (millions of instructions per second) and processor speed obviously do not serve as adequate measures of performance
- instructions executed on different processors can perform vastly different amounts of work
- measurements in terms of MIPS should only be used between processors of similar types.

It is strongly recommended that students examine the article:

Patterson, David A., Reduced Instruction Set Computers, Communications of the ACM, 28(1), January 1985.

+ +

The R2000

General History

- MIPS Computer Systems 1987
- MIPS is an acronym for Microprocessor without Interlocking Pipe Stages
- Based on work at Stanford University
- Hardware interlocks to prevent data and branch hazards are absent. Relies on compilers aware of the pipeline architecture of the machine to generate correct results. Resulted in faster operating speed by simplifying design.
- processor can be configured as either a big endian or a little endian processor in addition to allowing software selection.

Gross Features

- 32 bit processor
- Address range 2^{31}
- 32 General registers
- Load-Store Architecture
- RISC

+ +

Register Set

- 32 general registers (32 bit)
  - *R0 - R31*
- 2 multiply-divide registers (32 bit)
  - *HI & LO*
  - Result of 32 bit Multiplication
  - Quotient and Remainder of Integer division

Data Types
The R2000 supports 6 integer data types: signed and unsigned integers of 8, 16 and 32 bit size.
The R2010 floating point co-processor supports IEEE-754 floating point numbers of 32 and 64 bit size.

Instruction Formats

The R2000 supports 3 instruction formats:

R-Type instruction:

6 *bits*	5 *bits*	5 *bits*	5 *bits*	5 *bits*	6 *bits*
op	rs	rt	rd	shamt	funct

I-Type instruction:

6 *bits*	5 *bits*	5 *bits*	16 *bits*
op	rs	rt	address

J-Type instruction:

6 *bits*	26 *bits*
op	address

op	operation
rs	source register
rt	second register
rd	destination register
shamt	shift amount
funct	variant of operation
address	address

+ +

Addressing Modes

The R2000 supports 4 addressing modes:

Register Addressing The memory location is given by the value of a register

Base Addressing The address of the memory location is calculated by adding the contents of a register to the *address* in the instruction

Immediate Addressing The address of the memory location is the *address* in the instruction.

PC-Relative Addressing The address is the sum of the PC and the *address* in the instruction.

The jump instruction uses the 'J-type' format and exploits the requirement that instructions be aligned on 32 bit boundaries. The address of the destination is calculated:

> The 26 bit *address* field is shifted left 2 bits and combined with the top 4 bits of the PC to provide a 32 bit address.

A consequence of this arrangement is that a linker must avoid attempting to make a jump over a 256M byte boundary. If a jump over a 256M byte boundary is required then the destination address must be loaded into a register and a jump register instruction issued.

+ +

Instruction Set

The R2000 supports 74 instructions.
The subroutine mechanism is typical of many RISC processors:

- the R2000 does not have explicit stack instructions
- subroutine calls are implemented by using a **jump-and-link** instruction - jump to a specified location and store the return address in a register.
- returning from a subroutine is performed by issuing a **jump-to-register** instruction.

It is the responsibility of each subroutine to ensure that the return address is saved. In addition the caller must save the registers that it wishes preserved over a call.

Virtual Memory

The R2000

- does not support page tables to provide automatic translation of virtual to physical addresses.
- a 64 entry Translation Lookaside Buffer (TLB) is used for address translation.
- Each virtual address is divided into a 20 bit virtual page number and a 12 bit offset.

Address translation:

- the processor attempts to match the virtual page number and PID value with an entry in TLB table
- if there is a match and there is no rights violation then the base in the TLB entry is added to the offset and a physical address returned
- otherwise a trap occurs.

This mechanism simplifies the virtual address translation hardware but requires greater intervention than a page table based system.
Virtual Address:

20 *bits*	12 *bits*
Virtual Page	Offset

TLB entry:

20 *bits*	6 *bits*	6 *bits*	20 *bits*	1	1	1	1	12 *bits*
Virtual Page	PID		Page Frame	N	D	V	G	Offset

N	Not cached
D	Dirty
V	Valid
G	Global (Suppress PID check)

The PID value in the TLB entry is checked against the PID field in the Entry HI register. If the Global bit is set then the PID check is avoided.

Summary

In addition to the RISC characteristics exhibited by the R2000 processor the designers have opted for the simplest implementation at the cost of forcing complexity onto the compiler or programmer. The result is a processor that has a high instruction throughput, a minimal instruction set and few features aimed at making the compiler's or the programmer's task easier.

+ +

The 80386

- Produced by the Intel Corporation in 1985
- Member of the 80x86 family of microprocessors
- First of the paged architecture processors to serve as the processor for the popular IBM-PC personal computer range.
- A major constraint on the design of the 80386 has been to maintain backward compatibility with software written for earlier members of the 80x86 family. This requirement forced the designers to provide multiple modes of operation and a highly redundant instruction set. For backward compatibility:
  - Virtual 86 mode
  - Real Mode
  - 16 bit protected mode (286 compatibility)

  In addition the architecture added:
  - 32 bit protected mode
- Little endian

+ +

Gross Features

- 32 bit processor
- Address range 2^{32}
- 8 General registers
- CISC

Register Set

- 8 general registers (32 bit)
- 6 segment selector registers (16) bit

The 80386 dedicates some of the general registers to specific functions for some instructions.
Data Types
The 80386 supports the following integer data types: signed and unsigned integers of 8, 16 and 32 bit size, and signed 64 bit integers.
In addition bit fields, pointers, and strings are supported.
The 80387 floating point co-processor supports IEEE-754 floating point numbers.
In all 23 data types are present in the 80386.

+ +

Instruction Formats

The 80386 supports a single variable length instruction format. This is equivalent to the number of formats that could be formed by taking all the legal combinations of the variable length format.

Instruction Prefix	Address Size Prefix	Operand Size Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
bytes			

Opcode	MOD R/M	SIB	Disp	Imm
1 or 2	0 or 1	0 or 1	0 1 2 or 4	0 1 2 or 4
bytes				

Addressing Modes

The 80386 supports 11 addressing modes. The addressing of the 80386 modes have been discussed in the course.

Instruction Set

The 80386 supports 216 instructions. The 80387 supports 80 floating point operations.

The instruction set contained in the notes contains 86 instructions.

+ +

Virtual Memory
The 80386 supports:

- Paging
- Segmentation
- Paged Segmentation

Segmentation is supported using a descriptor table the segment descriptors index into the descriptor table. Each descriptor table entry contains the following information:

- Base Address
- Limit
- Presence Bit - Causes a fault if clear and an attempt has been made to load segment descriptor
- Privilege Level - Specifies minimum privilege required to access segment
- Type - Access rights or Special system types
- Segment Descriptor bit - used to distinguish system segments from user or kernel segments

+ +

- Accessed Bit - set if segment has been loaded
- Granularity - if 0 then the limit is in bytes otherwise the limit specifies the number of 32 bit long words that are accessible.

When a process wishes to use a segment it attempts to load a segment register with the segment selector for the segment required.
Segment Selector:

13 *bits*	1	2
Index	T	RPL

Index	Index into Global or local descriptor table
T	Table: 1 for local descriptor table, 0 for global descriptor table
RPL	Requested Privilege Level

The operating system is responsible for dealing with traps caused by invalid and not present descriptors.
The paging scheme used in the 80386 is based on a two level page table structure. Similar to that described in earlier lectures.
When performing paged segmentation the segment is decoded to generate the virtual address which is translated by examining the page tables.

+ +

Summary
The 80386 is processor with a diverse and rich instruction set. The paged segmentation scheme employed by the 80386 is rare in current microprocessors and few operating systems take advantage of the presence of segment registers.

+