

CSC236

**Computer Organization
and
Assembly Language Programming**

Class Notes

Mr. Dana Lasher

Version: 0713

**TAVENNER PUBLISHING CO.
137 Inlet Pointe Drive
Anderson, South Carolina 29625**

By Dana Lasher 2013, All rights reserved.

No part of this publication may be produced, stored in a retrieval system, or transmitted in any form by any means, electronic, mechanical, photocopying, recording or otherwise without the permission of the authors.

ISBN: 978-1-937435-51-6

Chapter Contents

1. **Introduction to Computer Architecture**
2. **Number Systems**
3. **Basic Computer System Components**
4. **Intel 8086 Architecture**
5. **PC Assembler**
6. **Assembler Opcodes and Directives**
7. **Additional Thoughts on Assembler Opcodes and Directives**
8. **File Input / Output**
9. **Multiply and Divide**
10. **Indirect Addressing**
11. **Subroutines**
12. **Linking High Level Languages with Assembler Code**
13. **Machine Code**
14. **Testing Programs**
15. **Boolean Logic**
- 16A. **Microcode and String Instructions**
- 16B. **Microcode Machine Example**
- 16C. **IEEE Floating-Point**
- 16D. **How To Use The Floating-Point Co-Processor**
17. **Interrupts and Input / Output**
18. **Design Trade-offs**
19. **Performance Issues in Processor Design**
20. **Post 8086 Architecture**
21. **Common Programming Bugs**
22. **Code Complexity**
23. **Programming for Efficiency**
24. **How To Design A Program**
25. **8086 Register Usage Summary**
26. **ASCII Codes**
27. **ARM Architecture**

Introduction To Computer Architecture And Assembly Language

In this introduction we will try to answer four questions:

1. What is computer architecture?
2. What is assembler language?
3. How are they related?
4. What is the purpose of this class?

Computer Architecture

The traditional definition of computer architecture is:

Computer architecture is the specification of the logical relationships among the parts of a computer system.

The architecture defines the attributes as seen by the programmer. It describes the functional behavior of the machine such as what instructions it will execute (add, subtract, multiply), but it does not specify the physical design or the performance characteristics of the machine. It describes what is does but not the physical characteristics of the components used to build it.

One way to study the architecture of a computer and thus understand the functions it performs is to study the computer's machine instruction set, which is also called the *machine language* of the computer. Machine language is the boundary between software and hardware.

The two main ideas that we have tried to provide about architecture are:

1. For any system, you can study its logical and physical characteristics.
 - Logical = Function This describes what the system will do.
 - Physical = How the system is built. This describes its speed, cost, size and weight.
2. Historically, Computer Scientists view a computer as a *black box* system. This is a system that we do not open up to look inside it. Instead, we study the architecture of a computer by studying its machine instruction set. This provides us with view of the functions that the computer can perform. This is a different view from Computer Engineers who do go inside the computer and do work with the physical characteristics of the computer.

There is a very significant specific application of this concept to computers. Prior to 1964, there were many computer vendors selling incompatible computers based upon different architectures and machine instruction sets. Some of these vendors were: GE, RCA, IBM, Honeywell, DEC, Univac.

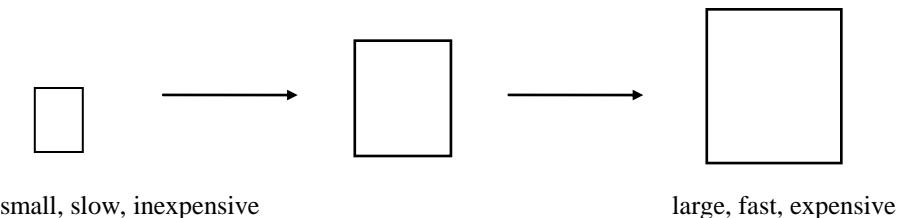
This multiple incompatible computers created a number of problems.

- Users could not share software packages.
- Users could not use another computer as backup.
- Users could not take advantage of price/performance improvements made to the computers of other vendors.

Even within a single vendor, such as IBM, there were often multiple incompatible architectures. IBM had the 1400 series for small commercial users and the 7070 series for large commercial users and the 7090 series for scientific users. This created a special problem for IBM. When users needed to upgrade their computer they would often evaluate other vendors since there is no easy growth path within the IBM family of computers.

IBM wanted to solve that growth problem and provide an easy way to migrate to a more powerful computer.

In 1964, IBM announced a single computer architecture called *System 360*. Within S/360 there would be many computer models, built in different countries, and built out of different components with different performance characteristics. However, they would all have the same machine instruction set. This means that they would run the same software, thus providing a growth path. The original models announced in 1964 were 30, 40, 50, 60, 70.



To prevent the customer from downsizing, IBM supported *Upward Compatibility*. This meant that you could always migrate to a larger and faster computer, but you could not easily migrate to a smaller and slower computer.

This was achieved by adding new functions or features as you moved up the model line. These new functions included more main memory, faster and larger disk drives, special floating-point hardware, etc. Once a customer used the new functions they could not easily move backwards.

The concept of upward compatibility is very important. It provides a growth path for customers and preserves their investment in software. This concept is used by many computer manufacturers today to provide a family of compatible computer models. This family can range from slow and inexpensive to fast and expensive. The family can also span many generations of computer systems. However, they all execute the same machine instruction set. This means that the software written for one model will run another model. As customers need more speed they can upgrade their computer without having to rewrite their software. IBM did this with their S/360 family of computers and Intel has done this with their many generations of computers ranging from the 8086 through the Pentium.

Assembly Language

Assembly language is a symbolic programming language whose statements match 1:1 with a specific machine instruction set.

The machine code for a computer is defined in terms of the binary 1s and 0s needed to construct an instruction such as *add*. It is very hard to program in machine code. Assembly language allows you stay at the machine language level but still be *symbolic*. That means in assembly language we can use the word *add* instead of having to write 10110101. However, basic assembler does not have any high level constructs such as *for*, *while* or *if-then-else*. It also has an absolute minimum of Input and Output support.

Often, in studying the architecture of a computer, we choose to study the computer's assembler language. Traditionally, operating systems and Input/Output device drivers are written in assembler language to achieve maximum performance. The key to assembler is that it does not hide the real hardware characteristics of a machine from your view, like C++ or Java might do. Assembler language is not portable across different machine architectures. You would not want to write applications such as payroll programs in assembler, but it will help you learn the architecture of a given computer.

Why is it valuable to study computer architecture and assembler language? One reason is that you may wish to be a computer designer or analyst. These careers are based upon knowing how computers are built and how one can use the capabilities of a computer to maximize performance. However, even if you are an application programmer, the knowledge obtained in studying architecture can help you in designing better and more efficient software.

Today, many modern computers are viewed as a multiple layered system.

- **The real hardware.** This consists of transistors configured as logic circuits.
- **The microprogram level.** Microinstructions control the flow of data through the machine, and control the activation of the real hardware (e.g. tell the ADDER circuit to add together two numbers). A series of microinstructions, called a microprogram, are used to implement each of the instructions in the machine language instruction set. Microprogramming allows the designer to easily implement a specific set of machine instructions and is thus a key concept in building a compatible family of computers.
- **The machine instruction set.** The most basic instructions that the computer can understand and execute.
- **Symbolic assembler language.** A symbolic programming language whose instructions have a one to one correspondence with a specific machine language.
- **The operating system.** The operating system has the function of managing and allocating the resources of the computer (CPU time, I/O devices, data files) to users.
- **High Level Languages.** This can be C++ or Java or application packages such as data base managers.

How does all this relate to this class?

The low cost of microprocessors allows computers to be used in any environment regardless of price: scales, watches, vending machines, ovens, and greeting cards. The problem is that as hardware costs have been dropping, software costs have been rising and the demand for more programmers has been increasing. To meet this need we can dip deeper into the well of humanity and pull up more people and make them programmers. However, they may not have the logical thought processes necessary to be a competent programmer. Instead they may end up being mediocre programmers and creating bad programs.

An alternate approach is to make existing programmers more productive with better hardware and software development tools. This is the job of the computer analyst or computer scientist.

The goal of this class is to strip away the comfort of High Level Languages such as C++ and Java and introduce you to the low level architectural world, with the hope that you may be the designer of a future generation of computing systems. We will do this by studying the architecture of a contemporary computer and by programming at the assembly language level.

Number Systems

Contents

Number Systems	2
A - Unsigned Positional Number Systems	3
Addition	4
Conversion	5
Subtraction	6
Hexadecimal	8
Finite Precision Arithmetic and Overflows	10
Precision versus Accuracy	12
B - Signed Positional Numbers	1
Signed Magnitude	3
One's Complement	4
Two's Complement	5
How can I do hex two's complement problems	8
A summary of why two's complement is significant	11
C - Number Systems Review	1
Unsigned Positional Number System Review	1
Signed Positional Number Systems Review	2
Two's Complement Number Systems Review	3
Hints on converting signed numbers to decimal	4
D - Additional Thoughts On Number Systems	1
Why does flipping the bits + 1 create the two's complement	2
Why does end around carry work for One's Complement	3
Why does the carry-in / carry-out rule detect a two's complement overflow	4
An alternate way to calculate two's complement	8
Can you just look at negative number and tell its decimal value	9
How can an overflow be detected in signed magnitude addition	10
How can an overflow be detected in ones complement addition	11
How can an overflow be detected when adding a two's complement signed number and an unsigned number	12
How can you do math, such as add and subtract, in the various systems	15
Additional insight	17
E - Fractions	1
Conversion of fractions among decimal, binary and hex	2

Number Systems

Number systems are the foundation upon which Computer Science is built.

To be successful in the following endeavors:

- understanding computer architecture
- designing computer systems
- assembly language programming
- high level language programming

one must have a solid understanding of number systems.

The basic number system topics include adding, subtracting and converting unsigned and signed integers using bases 2, 10 and 16.

Advanced number system topics include multiplication, division and fractions in those same bases.

The discussion of number systems is broken into these parts.

- A. Unsigned number systems
- B. Signed number systems
- C. A summary of unsigned and signed number systems
- D. A collection of problems, proofs and additional thoughts on number systems
- E. Fractions

A - Unsigned Positional Number Systems

The number systems we are interested in are Positional Number Systems. All PNS have a base and set of digits that run from the 0 to the base-1.

For example, our decimal number system has these characteristics.

PNS Name	Base	Digits
Decimal	10	0...9

When we write a number such as 179_{10} this is just shorthand for the way that the number really exists in its true positional format which is:

$$1 \times 10^2 + 7 \times 10^1 + 9 \times 10^0 = 100 + 70 + 9 = 179$$

That is, there is a set of positions represented by the base raised to successively higher powers (starting with zero) and a number is generated by multiplying the value of that position by a digit.

We use a subscript to indicate the specific base being used. So we know that 1011_2 is a binary number and not a decimal number.

Not all number systems are positional. Roman Numerals is not a positional number system. Roman Numerals use letters to represent numbers: I=1 II=2 III=3 IV=4 V=5 X=10 L=50 C=100. Below we 1+1 and 2+1 and 3+1. clearly a special rule is needed for the last calculation.

$$\begin{array}{r} & \text{I} & & \text{II} & & \text{III} \\ & + & \text{I} & + & \text{I} & + & \text{I} \\ & \text{II} & & \text{III} & & \text{IV} & \\ \hline & & & & & & \end{array}$$

The primary reason that we are interested in PNS is that it is easy to develop rules to do arithmetic in PNS.

The PNS used by computers is binary. Its base is 2 and its digits are 0 and 1. Computers use binary because each binary digit (bit) can represent the state of an electronic switch: 0=off, and 1=on. When we write a binary number such as 1011_2 then this is a shorthand for the way that number really exists which is:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

To determine the decimal value of that binary number we can just perform the arithmetic in decimal and we get $8 + 0 + 2 + 1 = 11_{10}$

Addition

The key to PNS is that we can perform addition directly on the digits. We do this right to left (least significant position to most significant position) one position at a time following this rule:

```
If the sum of the digits is less than the base
then the result is represented by a single digit,
Else (the sum is equal or greater than the base)
then the result is reduced by the base and represented by a single digit
and a carry into the next higher position.
```

This rule may seem obvious, but it will play a major role when we study signed numbers and how computer architects decide how to build modern computers.

Lets apply the rule to two examples of addition in decimal (3+4) and (7+5).

$$\begin{array}{r}
 & & & \text{carry} \\
 & & 1 & \\
 & 7 & & \\
 + 5 & & & \\
 \hline
 12 & & & \\
 \end{array}$$

7+5=12 units.
 12 units is greater than the base of 10
 so it is reduced by the base with the
 result of 2 and a carry into the next
 position.

sum is → 7
 less than
 the base

Since there are only two digits in binary, we can see all combinations of adding two binary digits.

$$\begin{array}{cccc}
 & & & \text{carry} \\
 & & 1 & \\
 0 & 0 & 1 & 1 \\
 + 0 & + 1 & + 0 & + 1 \\
 \hline
 0 & 1 & 1 & 10
 \end{array}$$

The rule can also be used on multiple digit numbers. We just apply the rule to all the positions, going from the least significant to the most significant.

This example adds the binary value 1011 (equal to decimal 11) to itself.

$$\begin{array}{r}
 1\ 1 \quad <\!\!-\!\!> \text{ carry} \\
 1\ 0\ 1\ 1_2 \\
 + 1\ 0\ 1\ 1_2 \\
 \hline
 1\ 0\ 1\ 1\ 0_2
 \end{array}$$

To verify the result is correct we convert it to decimal by performing the arithmetic of multiplying each digit by its positional value:
 $16 + 0 + 4 + 2 + 0 = 22_{10}$

Conversion

To convert from binary to decimal just perform the arithmetic in decimal.

This is called *expansion of powers*,

$$\begin{array}{ccccccccc} \text{so } 10110_2 & = & 1 & 0 & 1 & 1 & 0 & \text{--- digits} \\ & & 16 & 8 & 4 & 2 & 1 & \text{--- positional values} \\ & & = 16 + 0 + 4 + 2 + 0 & = 22_{10} \end{array}$$

To convert from decimal to binary perform *reduction of powers*.

$$22_{10} = \underline{\quad} \times 2^5 + \underline{\quad} \times 2^4 + \underline{\quad} \times 2^3 + \underline{\quad} \times 2^2 + \underline{\quad} \times 2^1 + \underline{\quad} \times 2^0$$

We just need to fill in the blank spaces with digits.

$2^5 = 32$ which is greater than 22 so the left most binary digit is 0
 $2^4 = 16$ so that digit is 1 and we have $22-16 = 6$ left to convert
 $2^3 = 8$ which is greater than 6 so that digit is 0
 $2^2 = 4$ so that digit is 1 and we have $6-4 = 2$ left to convert
 $2^1 = 2$ so that digit is 1 and we have $2-2 = 0$ left to convert
 $2^0 = 1$ which is greater than 0 so that digit is 0.

The converted number is:

$$22_{10} = 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 010110_2$$

Subtraction

Real subtraction is not actually performed by modern computers. The operation $C=A-B$ is performed by adding complements. So $C=A-B$ will be done as $C=A+(-B)$. However, we consider unsigned subtraction for completeness.

To perform real subtraction we work from right to left. If we have to subtract a larger digit from a smaller digit then we re-distribute the data to allow the subtraction process to continue.

First, the decimal example $85 - 7$.

We start by rewriting the numbers in their positional format.

$$\begin{array}{rcl} 85 & = & 8 \times 10^1 + 5 \times 10^0 \\ - 07 & = & \underline{0 \times 10^1 + 7 \times 10^0} \end{array}$$

The first operation $5-7$ cannot be performed. So we redistribute the data from next most significant position. One of 10^1 values becomes 10×10^0 . Then we perform the subtraction of $(10+5) - 7 = 8$.

$$\begin{array}{rcl} 85 & = & 8 \times 10^1 + 5 \times 10^0 = \cancel{8 \times 10^1} + 5 \times 10^0 \\ - 07 & = & \underline{0 \times 10^1 + 7 \times 10^0} = \underline{0 \times 10^1 + 7 \times 10^0} \\ \hline 78 & & 7 \times 10^1 + 8 \times 10^0 \end{array}$$

Now the binary example $100 - 001$.

We start by rewriting the numbers in their positional format.

$$\begin{array}{rcl} 100 & = & 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ - 001 & = & \underline{0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0} \end{array}$$

The first operation $0-1$ cannot be performed. So we redistribute the data from next most significant positions. Then we perform the subtraction.

$$\begin{array}{rcl} 100 & = & 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = \begin{matrix} 0 \times 2^2 \\ \cancel{1 \times 2^2} \\ \hline 0 \times 2^2 \end{matrix} + \begin{matrix} 1 \times 2^1 \\ \cancel{1 \times 2^1} \\ \hline 0 \times 2^1 \end{matrix} + \begin{matrix} 1 \times 2^0 \\ \cancel{0 \times 2^0} \\ \hline 1 \times 2^0 \end{matrix} \\ - 001 & = & \underline{0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0} = \underline{0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0} \\ \hline 011 & & 0 \quad 1 \quad 1 \end{array}$$

Performing unsigned subtraction the way just described is cumbersome when working with large numbers. We now present a technique for speeding up the unsigned subtraction process.

The problem of redistribution occurs whenever we have to subtract 0 - 1.

We can use the technique of rewriting a 1 bit followed a string of 0 bits as a 0 bit followed by a string of 1 bits plus an additional 1.

For example:

$$10000_2 = 01111_2$$

Note that both sides have the same value.

- $10000_2 = 16_{10}$
- $01111_2 + 1_2 = 15_{10} + 1_{10} = 16_{10} = 10000_2$

However the right side makes it easier to perform subtraction.

Subtracting 1 from
the left side
requires a lot of
redistribution.

$$\begin{array}{r} \downarrow \\ 10000_2 \\ - \underline{00001}_2 \end{array}$$

Subtracting 1 from
the right side does
not require any
redistribbtion.

$$\begin{array}{r} \downarrow \\ 01111_2 \\ \underline{00001}_2 \\ \hline 0111_2 \end{array}$$

We perform this process any time we need to subtract 0 - 1. In this example, bold italics with strike through shows which **10** pair becomes **01** + 1.

$$\begin{array}{r} & \mathbf{1} & 1 & 1 \\ & \mathbf{1} & 11 & 11 & \mathbf{1} 11 \\ & \mathbf{01} & \mathbf{001} & 001 & \mathbf{01}001 \\ 22_{10} = & 10110_2 & 10\mathbf{10} & 10\mathbf{110} & 10110 & \mathbf{10}110 \\ - \underline{11}_{10} = & - \underline{01011}_2 & = \underline{01011} & = \underline{01011} & = \underline{01011} & = \underline{01011} \\ \hline 11_{10} & & 1 & 11 & 011 & 01011 \end{array}$$

Hexadecimal

Many years ago the telephone industry determined that the average person could remember 7 digits and this became the basis for 7 digit telephone numbers.

This means that large binary numbers such as 101101011110101011000011 will be hard to remember and work with. So we use shorthand for binary. We group 4 binary digits together: 1011 0101 1110 1010 1100 0011. Each group of 4 binary digits can have the binary values 0000 to 1111 which correspond to the decimal values 0 to 15.

The digits of a PNS run from 0 to the base-1. So if we have digits that run from (0) to (15) then the base will be 16. This is the hexadecimal system. However 15 is not a good digit! How could we tell the signle digit 15 from the two digit number 15. Instead we need a single character to represent each digit and so the letters A to F are used for the hexadecimal digits 10 to 15. This table shows the corresponding decimal, hex and binary values.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Conversion between binary and hex just uses this look-up table.

$$1011\ 0101\ 1110\ 1010\ 1100\ 0011_2 = \text{B5EAC3}_{16}$$

Conversion from hex to decimal uses the standard expansion of powers.

$$1A7_{16} = 1 \times 16^2 + 10 \times 16^1 + 7 \times 16^0 = 423_{10}$$

Conversion from decimal to hex uses the standard reduction of powers.

$$423_{10} = \underline{\quad} \times 16^2 + \underline{\quad} \times 16^1 + \underline{\quad} \times 16^0$$

$16^2 = 256$ There is one 256 in 423 so the digit is 1 and $423 - 256 = 167$ left.
 $16^1 = 16$ There are ten 16s in 167 so the digit is A and $167 - 160 = 7$ left.
 $16^0 = 1$ There are seven 1s in 7 so the digit is 7.

$$423_{10} = 1 \times 16^2 + A \times 16^1 + 7 \times 16^0 = 1A7_{16}$$

Adding and subtracting hex uses the standard rule for PNS.

Below is the math for each column for adding the hex numbers 1A7 + B89. I show the hex digits being added, their decimal equivalents, the final hex result.

Carry ---->	11	hex digits	decimal addition	final hex result
	1A7	1. 7+9	7+9 = 16	0 plus a carry
+ B89		2. 1+A+8	1+10+8 = 19	3 plus a carry
		3. 1+1+B	1+1+11 = 13	D
	D30			

We can check our results by using expansion of powers.

- $1A7_{16} = 423_{10}$
- $B89_{16} = 11 \times 16^2 + 8 \times 16^1 + 9 \times 16^0 = 2953_{10}$
- $D30_{16} = 13 \times 16^2 + 3 \times 16^1 + 0 \times 16^0 = 3376_{10} = 423_{10} + 2953_{10}$

Below is the 3 step math process for subtracting D30 - B89.

2 16	16	16
D 3 0	C 2 16	C 2 16
- B 8 9	- D 3 0	- D 3 0
-----	-----	-----
7	A 7	1 A 7

Step 1 Step 2 Step 3
redistributes redistributes calculates
data to data to C-B
calculate 0-9 calculate 2-8

Remember that *all electronic circuits are really binary circuits*. Hex is only shorthand used by humans to represent large binary values.

Finite Precision Arithmetic and Overflows

If you are working with a pencil and paper and generate a result that needs additional digit positions, then you can just create those new positions. This is called *infinite precision arithmetic*.

When you build a real machine, such as a calculator or computer, you must fix the number of digits that can be used (e.g. the size of the calculator's display). This is called *finite precision arithmetic*. If you now generate a result that needs more digits, you are in trouble! This is called an *overflow*.

*An overflow is the generation of a result
that does not fit in the data field being used*

The implication of an overflow is that the result calculated is wrong and should not be used in further operations.

For unsigned numbers, an overflow occurs when you:

- add two numbers and the result is too large to fit in the assigned data field
- subtract a larger number from a smaller number attempting to generate a negative number

The way computers handle unsigned overflows is very different from the way calculators handle overflows.

If you have a calculator with an 8 digit capacity and you generate a 9 digit answer then the calculator stops and displays the word *ERROR* in the display. You usually need to clear the calculator and start again.

Overflows in computers are handled as follows.

Addition: In a computer, if you generate an unsigned overflow on an addition, the system completes the execution of the instruction. The result generated is the correct low order part of the desired result. There will also be a flag, called the *carry flag*, set to indicate that an overflow occurred and the result is wrong.

If we add the 4-bit (1 hex digit) unsigned binary numbers ($1111_2 + 0010_2$) we expect to get 10001_2 . Since this is a 5 bit result, and does not fit in a 4-bit binary (1 hex digit) field, instead we get 0001_2 and an overflow indication.

Note the correct result should be 10001_2 and the actual result is the correct low order four bits.

<u>Binary</u>	<u>Hex</u>	<u>Decimal</u>	We show 3 bases: binary, hex and decimal
111	1 ←		Carries
1111_2	F_{16}	15_{10}	The carry out of the most significant digit, shown as 1, signifies an overflow.
$+ 0010_2$	2_{16}	2_{10}	
-----	---	---	
0001_2	1_{16}	1_{10}	This rule is true for all bases if the numbers are unsigned.

The reason the hardware works this way is to allow the programmer to handle the overflow condition and possibly write software to support large variable length data sizes such as 100 digit numbers.

Subtraction: In a computer, if you generate an unsigned overflow on a subtraction, the system completes the execution of the instruction. The result is generated as if the system could perform a borrow (redistribute data) into the most significant digit. This result will be the correct low order part of the desired result. The same carry flag will be set to indicate that a borrow was assumed to occur into the most significant digit and thus an overflow occurred and the result is wrong.

For example, if we subtract $4_{10} - 7_{10}$ the correct result of -3_{10} is not a valid unsigned number.

In binary this would look like $0100_2 - 0111_2$.
The result calculated is 1101_2 and an overflow indication.

Simulated borrow into the most significant digit

$$\begin{array}{r}
 0100_2 = 4_{10} \quad \text{Operation done as this} \\
 - 0111_2 = -7_{10} \\
 \hline
 1101_2 = 13_{10}
 \end{array}
 \qquad
 \begin{array}{r}
 10100_2 \\
 - 0111_2 \\
 \hline
 1101_2
 \end{array}$$

The result looks like 13 instead of -3, which of course is wrong!
It is wrong because -3 does not exist in the world of unsigned numbers.

If we did this in hex, it looks like:

$$\begin{array}{r}
 4_{16} = 14_{10} \quad \text{Operation done as} \\
 - 7_{16} = -7_{10} \quad \leftarrow \quad \begin{array}{l} \text{In decimal, the top value becomes } 16 + 4 = 20 \\ \text{The bottom value is 7} \\ 20 - 7 = 13 \text{ in decimal or D in hex.} \end{array} \\
 \hline
 D_{16} = 13_{10}
 \end{array}$$

A borrow into the most significant digit signifies an unsigned overflow when subtracting. This rule is true for any base.

Note that **if** the top number were 5 digits and the most significant digit was 1 then the correct result would be 1101_2 . So the actual result shown above is the correct low order four bits.

$$\begin{array}{r}
 10100_2 = 20_{10} \\
 - 00111_2 = -7_{10} \\
 \hline
 01101_2 = 13_{10}
 \end{array}$$

Again, the reason the hardware works this way is to allow the programmer to handle the overflow condition and possibly write software to support large variable length data sizes.

Precision versus Accuracy

Stop for a moment and think about the two words ***precision*** and ***accuracy***.
Do you know what they mean? Do you know how they differ?

A computer performs this calculation: $1.00000000 + 2.00000000 = 4.78438721$

- Is the result 4.78438721 precise?
- Is the result 4.78438721 accurate?

Precision and accuracy are quite different.

Precision is a measure of the number of digits used to perform the calculation.

Accuracy is the measure of how close your result is to the correct value.

Given this calculation: $1.00000000 + 2.00000000 = 4.78438721$

- The result 4.78438721 is very precise. It has lots of digits.
- The result 4.78438721 is not accurate. The result is wrong.

B - Signed Positional Numbers

Introduction and Goal

Modern computers do not perform subtraction the way we did it for unsigned positional number systems.

Instead, they use complement arithmetic and perform $A - B$ as $A + (-B)$.

This process allows the CPU to use the same adder circuit for both addition and subtraction. A separate subtraction circuit is not needed, thus saving hardware.

However, in order for this to work, we must ***select a representation for signed numbers that uses the exact same rule for addition that is used by unsigned PNS.***

When we find such a number system then we can then have one single machine instruction that works on both unsigned and signed numbers. We can have an instruction such as `ADD X,Y` that correctly adds the variables X and Y regardless of whether X and Y are unsigned or signed values.

The Bad News

There are many ways to represent signed numbers and not all of them have the desired characteristic of using the same rule for addition that is used by unsigned PNS.

For example, the everyday decimal number system we use, called signed magnitude, does not have this feature.

We create a signed magnitude number by placing a + or - sign in front of an unsigned number. For example, 10 becomes +10 or -10. If we try to add signed magnitude numbers using the rule for adding unsigned numbers, we quickly encounter major problems.

unsigned	signed magnitude	
10	+10	
+ 05	+ -05	
--	---	We don't know how to handle signs and the
15	?15	<- magnitude is wrong.

We can expand the rule for adding digits in unsigned PNS to handle signed magnitude numbers. These are the results we are looking for.

+10	+10	-10	-10
+05	-05	+05	-05
--	--	--	--
+15	+05	-05	-15

This leads to the rule for addition of signed magnitude numbers:

```
If the signs are the same  
    then add the digits using rule for unsigned PNS  
    and the result has the sign of the input numbers  
Else if the magnitudes are different  
    then subtract the digits of the smaller number  
    from the digits of the larger number  
    and the result has the sign of the larger input number  
Else if the magnitudes are the same  
    then the result is zero and the sign is either + or -
```

This is an extension of the basic rule for addition in unsigned PNS. We could use it, but that would mean different rules for unsigned and signed numbers and separate computer instructions for unsigned and signed numbers. The rule is also pretty complicated and harder to convert into hardware logic.

Instead, lets search for a signed number system that can use the same rule for addition as is used for unsigned numbers.

A Tale of Three Signed Number Systems

We will look at three different ways to represent signed binary numbers. These three different number systems are named:

- signed magnitude
- one's complement
- two's complement

For each of these number systems we look at three characteristics:

1. What is rule for adding bits?
2. How many ways can you represent zero?
3. Is there the same quantity of positive and negative values?

We will look at these three number systems in binary. We will use 4 bit binary numbers. The left most bit will be the sign bit.

- 0 sign bit indicates a positive number
- 1 sign bit indicates a negative number

We will see that all 3 number systems use the same representation for their positive values and these positive values look like unsigned values.

It will be negative values that differ in these 3 number systems.

Signed Magnitude

In signed magnitude, you negate a value by inverting the sign bit.

This process works both ways. Invert the sign of a positive number and it becomes negative. Invert the sign of a negative number and it becomes positive. Here are some signed magnitude values.

bin	dec	dec	bin

0000	+0	-0	1000
0001	+1	-1	1001
0010	+2	-2	1010
0011	+3	-3	1011
0100	+4	-4	1100

First let us see if we can add bits using the rule for unsigned PNS.

If you add +1 and -1 using the unsigned rule you get:

$$\begin{array}{r} 0001 \\ + 1001 \\ \hline 1010 = -2 \end{array}$$

This is clearly not the right answer.

How does signed magnitude match our three characteristics?

1. It needs an extended addition rule to work properly. For use with binary, the *sign is the left most bit and the digits are all the remaining bits*.

```
If the signs are the same
    then add the digits using rule for unsigned PNS
    and the result has the sign of the input numbers
Else if the magnitudes are different
    then subtract the digits of the smaller number
    from the digits of the larger number
    and the result has the sign of the larger input number
Else if the magnitudes are the same
    then the result is zero and the sign is either + or -
```

2. It has two values for zero (+0 and -0).
3. It has the same quantity of positive and negative values.

Primarily because it requires an extension to the rule for addition, signed magnitude is not used by computer manufacturers.

Section 2D shows how to detect an overflow when adding signed magnitude numbers.

One's Complement

In one's complement, you negate a value by inverting all the bits.

This process works both ways. Invert all the bits of a positive number and it becomes negative. Invert all the bits of a negative number and it becomes positive. Here are some one's complement values.

bin	dec	dec	bin
0000	+0	-0	1111
0001	+1	-1	1110
0010	+2	-2	1101
0011	+3	-3	1100
0100	+4	-4	1011

If you add +1 and -1 using the unsigned rule you get:

$$\begin{array}{r} 0001 \\ + 1110 \\ \hline 1111 = -0 \end{array}$$

The above example works correctly, but now try -1 + -1

$$\begin{array}{r} 1110 \\ + 1110 \\ \hline 1100 = -3 \text{ This is wrong.} \end{array}$$

You cannot add one's complement numbers using the standard rule for unsigned PNS. However, you can extend the unsigned PNS rule to work for one's complement.

The extended rule for one's complement addition is called *end around carry*. Its derivation is shown in section D of these notes.

Add the bits using the rule for unsigned PNS. If there is a carry out of the most significant bit then add it into the least significant bit.

Lets apply it to adding -1 + -1

$$\begin{array}{r} 111 & \leftarrow \text{--- carry} \\ 1110 & \leftarrow \text{--- -1} \\ + 1110 & \leftarrow \text{--- -1} \\ \hline & \\ 1100 & \\ 1 & \\ \hline & \\ 1101 = -2 & \text{which is correct.} \end{array}$$

How does one's complement match our three characteristics.

1. It needs an extended addition rule to add. This extension is *end around carry*.
2. It has two values for zero (+0 and -0).
3. It has the same quantity of positive and negative values.

Computers have been built using one's complement: DEC PDP-1 in 1960, the first mini computer and the Univac 1100, in 1962, a popular mainframe.

Section 2D shows how to detect an overflow when adding one's complement numbers.

Two's Complement

The definition of two's complement is: $N + TC(N) = 2^d$

N = Any number.

TC(N) = The complement of N which is also known as -N.

d = The number of binary digits (bits) being used.

The complement is calculated by rewriting the equation and only saving the result to d digits and ignoring carries out of the sign position.

$$TC(N) = 2^d - N = 10000_2 - N \text{ if } d = 4$$

So, for example, the two's complement of 0001_2 is $10000_2 - 0001_2 = 1111_2$

If you complement positive zero you get positive zero. The two's complement of 0000_2 is $10000_2 - 0000_2 = 0000_2$ (remember we only save the low 4 bits). There is no bit pattern that represents negative zero. It does not exist.

Here are some more two's complement values.

bin	dec	dec	bin	
0000	+0	-0		There is no negative zero
0001	+1	-1	1111	←
0010	+2	-2	1110	
0011	+3	-3	1101	
0100	+4	-4	1100	

If you add +1 and -1 using the rule for unsigned PNS then you get the correct answer of +0.

$$\begin{array}{r} 0001 \\ + 1111 \\ \hline 0000 = +0 \end{array}$$

How does two's complement match our three characteristics.

1. It uses the same rule for addition as unsigned numbers.
2. It has one value for zero.
3. It has the same quantity of positive and negative values but they are not symmetrical around zero. There is always one more negative non-zero value than positive non-zero value.

Two's complement is the number system selected for signed numbers by modern computer manufacturers.

There is a shortcut for calculating two's complement that does not require subtraction. The shortcut is to flip the bits and add 1.

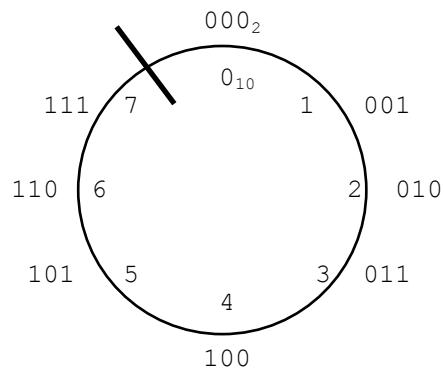
The shortcut is: $TC(N) = \text{flip_bits} + 1$

Most people use this method. Of course it works both ways. Complement a positive number and you get its negative value. Complement a negative number and you get its positive value.

Sometimes viewing a number system pictorially (a number wheel) helps you understand what is happening.

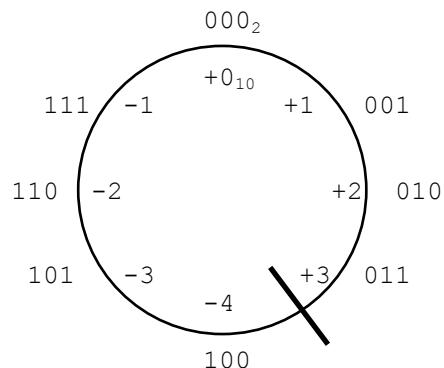
To simplify the picture we only use 3 binary digits instead of 4 binary digits.

The binary values are shown on the outside of the wheel and the decimal values are shown on the inside of the wheel.



Unsigned number wheel with 3 digits

When 001₂ is added to 111₂ the result is 000₂ and a carry. The carry indicates an unsigned overflow occurred and that the result 000₂ is wrong.



Signed number wheel with 3 digits

When 001₂ is added to 111₂ the result is 000₂ and a carry. The result of 000₂ is correct and the carry does **NOT** indicate a signed overflow occurred.

An overflow occurs when 001₂ is added to 011₂ yielding 100₂. This is (+1₁₀) + (+3₁₀) yielding (-4₁₀) which is wrong.

Note there is a -4 but not a +4. With 3 binary digits the range of decimal values is -1 to -4 and +0 to +3.
There is one more non-zero negative number than non-zero positive number.

A binary two's complement overflow can be detected in two ways.

1. If you add two numbers with like signs and the result has a different sign then an overflow occurred.

You cannot have an overflow when adding unlike signs since the magnitude of the result cannot be bigger than the magnitude of the inputs, which were valid.

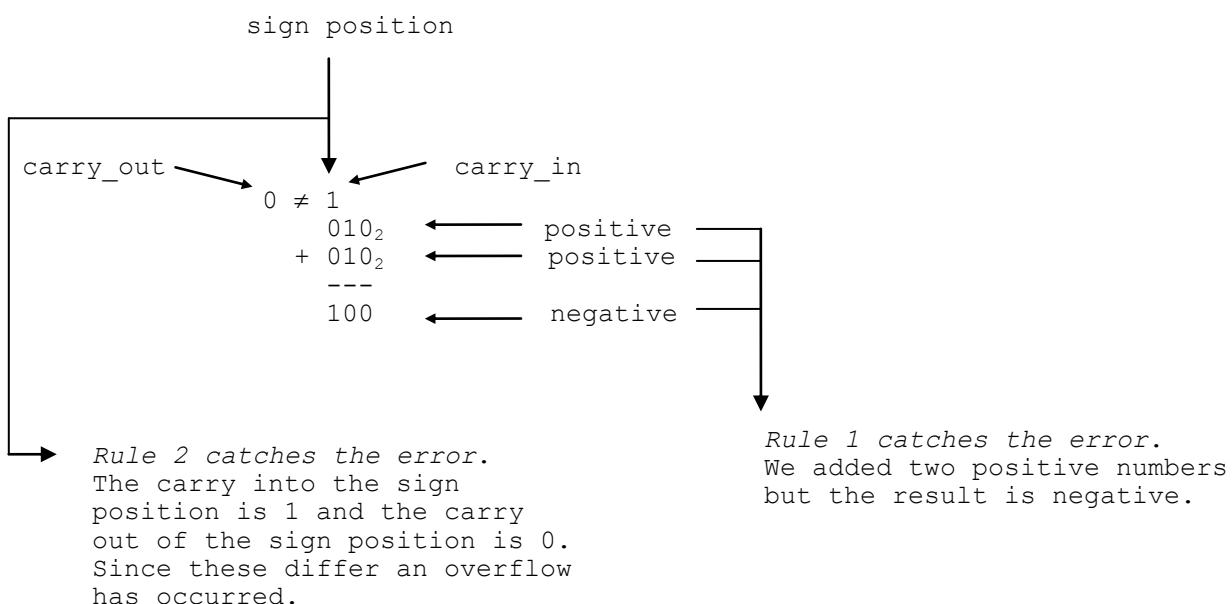
2. If the Carry-Out of the sign position is not equal to the Carry-In to the sign position then an overflow occurred.

Both rules always work and both rules always give the same indication of whether a signed overflow occurred.

Lets look at an example of adding 3 bit two's complement numbers.

We add decimal +2 to itself. We would like to get +4 as the result, but +4 is outside the range of valid answers for 3 bit two's complement numbers. That range, from the previous page, is -4 to -1 and +0 to +3.

Decimal +2 has the bit pattern 010.



How can I do hex two's complement problems

Remember, there is not any hex hardware. Hardware is binary. Although numbers in the computer are binary, humans often write those binary numbers in hex format.

You can do arithmetic in hex or you can convert it to binary and do the work. Add these two's complement hex numbers and indicate if a signed overflow occurred.

$$\begin{array}{r} 0111 \\ 50_{16} \quad \longrightarrow \quad 0101\ 0000_2 \\ + 70_{16} \quad \longrightarrow \quad 0111\ 0000_2 \\ \hline \text{C0}_{16} \quad \longleftarrow \quad 1100\ 0000_2 \end{array}$$

This example converts to binary and then does the addition.

Did a signed overflow occur? Yes. Both rules show an overflow occurred.

- The Carry_in to the sign position was 1 and the Carry_out of the sign position was 0 so a signed overflow occurred.
- Both input values are positive but the result is negative, so a signed overflow occurred ($50_{16} = 80_{10}$ and $70_{16} = 112_{10}$ so the correct result should be 192_{10} which does not fit in a signed byte).

As you develop experience, you can do the addition without converting.

$$\begin{array}{r} 50_{16} \\ + 70_{16} \\ \hline \text{C0}_{16} \end{array}$$

The result is straightforward: $50 + 70 = \text{C0}$.

But how can you tell if a signed overflow occurred?

We will use the rule that states an overflow occurred when you add two numbers with the same sign and the result has a different sign.

You can tell if a hex two's complement number such as 50, 70 or C0 is positive or negative. Convert just the left most digit to binary, so $5_{16} = 0101_2$. Then look at the left most bit. If it is 0 then the number is positive, if it is 1 then the number is negative. So 50 is positive, 70 is positive, C0 is negative.

An even faster way is to recognize that

- hex digits 0-7 all have 0 as the left most bit so they are all positive
- hex digits 8-F all have 1 as the left most bit so they are all negative

This means that $50 + 70 = \text{C0}$ and we have the sum of two positive numbers being negative so a signed overflow occurred.

$$\begin{array}{r} 50_{16} \quad \longleftarrow \quad \text{positive} \\ + 70_{16} \quad \longleftarrow \quad + \text{positive} \\ \hline \text{C0}_{16} \quad \longleftarrow \quad \text{negative result means there was a signed overflow} \end{array}$$

Watch out for the following trap!

If you use the carry-in/carry-out rule, it **looks** like the carry-in equals the carry-out and that means there should not be an overflow.

$$\begin{array}{r} \text{carry-out --> } 0 \ 0 <-- \text{ carry-in} \\ \quad \quad \quad 50_{16} \\ + \quad \quad 70_{16} \\ \hline \quad \quad \quad C0_{16} \end{array}$$

What happened?

The problem is that the carry-in/carry-out rule is a binary rule. That rule looks at the carry-in and the carry-out of the binary sign position. In the example above, the carries are between hex digits and not the binary sign position.

If you convert to binary and do the math then the rule does work.

You can do hex two's complement subtraction the same ways.

Either convert to binary or work directly in hex.

First lets convert to binary and calculate A-B as A+(-B).

Subtract these two's complement hex numbers C0 - D0 and specify whether a signed overflow occurred.

$$\begin{array}{r} & \text{00001 } 111 \\ \text{C0}_{16} & \longrightarrow 1100 \ 0000_2 & \longrightarrow 1100 \ 0000 \\ - \text{D0}_{16} & \longrightarrow 1101 \ 0000_2 & \longrightarrow 0010 \ 1111 \\ & & & \downarrow \\ & & & 1 \\ \hline & \text{F0}_{16} & \longrightarrow 1111 \ 0000_2 & \longrightarrow 1111 \ 0000 \end{array}$$

flip the bits
and add 1

Did a signed overflow occur? No. Both rules show that to be the case.

- The Carry_in to the sign position was 0 and the Carry_out of the sign position was 0 so a signed overflow did not occur.
- When adding a positive and negative number, there never is an overflow.

We can verify our result by converting the numbers to decimal.

$\text{C0}_{16} = -64_{10}$ and $\text{D0}_{16} = -48_{10}$ so the correct answer be -16_{10} which is F0_{16} .

Now lets do the calculation directly in hex.

$$\begin{array}{l} \text{Step 1} \\ 0-0=0 \\ \downarrow \\ \text{C0}_{16} \quad \text{C0}_{16} \quad \text{C } 0_{16} \\ - \text{D0}_{16} \quad - \text{D0}_{16} \quad - \text{D } 0_{16} \\ \hline 0 \quad \quad \quad \text{F } 0 \end{array}$$

Step 2
We cannot do C-D so we assume there is a 1 in the next significant position and redistribute data. $\text{C}_{16}=12_{10}$ and $\text{D}_{16}=13_{10}$ so we then calculate $(16_{10}+12_{10}) - 13_{10} = 15_{10} = \text{F}_{16}$

Did an overflow occur? C0 is negative and D0 is negative. The calculation of negative - negative is equal to negative + (- negative) or negative + positive. There is never an overflow when adding unlike signs.

From this logic we can even create a new rule.

There is never an overflow when subtracting like signs

If you perform (negative - positive) then a positive result indicates an overflow

If you perform (positive - negative) then a negative result indicates an overflow

A summary of why two's complement is significant

- Two's complement was chosen as the way computers store signed numbers because then computers can use the same electronic circuits, and machine instructions, for adding unsigned positional numbers and signed two's complement numbers.
- The hardware **cannot** tell if a bit pattern represents an unsigned positional number or a signed two's complement number. It is the responsibility of the **programmer** to keep track whether a variable is unsigned or signed.

For example, the binary number 11111111 can be either decimal 255 as an unsigned value or -1 decimal as a signed number. The hardware cannot tell and does not know; the programmer must know.

- The way that one detects unsigned and signed overflows differs. For example if you add 11111111 + 00000001 the expected results differ.

	Actual Binary	Expected Unsigned	Expected Signed
Carries →	1 = 1111 111		
	1111 1111	255	-1
	0000 0001	1	+1
	-----	---	-
	0000 0000	256	+0

The **expected unsigned** result is 256. The actual result was 0. The carry out of the most significant position signifies that the value of 0 is not correct, an overflow occurred.

The **expected signed** result is +0. The actual result was +0. The fact that the Carry-In to the sign position equals the Carry-Out of the sign position signifies that +0 is the correct answer.

- Subtraction can always be performed in either of two ways.

- Pure subtraction: A - B
- Adding the complement: A + B + 1 (B means invert the bits of B)

For example: 00000001 - 11111111 can be done either way.

- Pure subtraction

$$\begin{array}{r}
 & 1 \\
 & 1111 111 \\
 \underline{+} & 0000 0001 \\
 - & 1111 1111 \\
 \hline
 & 0000 0010
 \end{array}$$

A - B Answer

You always get the same numerical result

- Adding of the two's complement

$$\begin{array}{r}
 0000 0001 \quad A \\
 0000 0000 \quad + B \text{ inverted} \\
 + \quad 1 \quad + 1 \\
 \hline
 0000 0010 \quad \text{Answer}
 \end{array}$$

You always get the **SAME NUMERICAL RESULT**.

Most people would rather do the adding of the complement instead of the pure subtraction. Since you get the same result either way and the hardware does not know if the numbers are signed or unsigned you can even subtract unsigned numbers using this method.

However, the indication of an overflow is different depending on how the programmer is interpreting these values.

- If these are unsigned values then we have 1 minus 255 and the borrow into the most significant bit signifies the answer of 2 is wrong.
- If these are signed values then we have (+1) minus (-1) and the Carry-In to the sign equal to the Carry-Out of the sign (both 0) signifies the answer of +2 is correct.

This last example should answer the burning question:

How do you detect an overflow on a binary two's complement subtraction?

The answer is that you perform the subtraction by adding the complement and then look at the Carry-In and the Carry-Out of the sign position. If they are equal then there was not an overflow.

Warning ... adding the complement is done in a single step.

Okay: $A - B = A + \underline{B} + 1$

Not okay: $A - B \neq A + (\underline{B} + 1)$

The use of the one's complement and a low order one instead of the two's complement of the second operand is necessary for the proper recognition of overflow when subtracting the most negative number.

C - Number Systems Review

Unsigned Positional Number System Review

- Positional number systems are the foundation of mathematics.
- Each positional number system has a name, a base, and a set of digits that range from 0 through the base-1.
- Positional number systems allow you to perform operations directly on the digits.
- In binary, numbers range from 0 to $2^d - 1$, where d = the number of binary digits (bits) being used.
- The rule for adding digits is:

```
If the sum of the digits is less than the base  
    then the result is represented by a single digit  
Else  
    the sum is reduced by the base and represented by a single  
    digit and a carry.
```

- An overflow is the generation of a result that does not fit in the number of digits available. It means the answer you calculated is wrong.

An unsigned overflow is detected by a carry out of the most significant digit on addition, or a borrow into the most significant digit on subtraction.

The rule above is true for any base.

Signed Positional Number Systems Review

There are multiple ways to represent signed numbers. We looked at three:

1. Signed Magnitude
2. One's Complement
3. Two's Complement

They all have these characteristics:

- The left most bit is the sign bit where 0 indicates a positive number and 1 indicates a negative number.
- The positive numbers and unsigned numbers all look identical.
- It is the representation of *negative values* that differ.

The three characteristics of interest for each signed number system are:

- Can you perform arithmetic operations directly on the bits just as if they were unsigned numbers?
- How many representations of zero are there in the number system?
- How many non-zero positive and non-zero negative digits are there in the number system?

Signed Magnitude number system:

- To negate a number you invert the sign bit.
- Can you do arithmetic operations directly on the data?
No, you need a more complex algorithm.
- There are two representations of zero.
- There are the same number of non-zero positive and negative digits.
- Example: 0010 = +2 1010 = -2

One's Complement number system:

- To negate a number you invert all the bits.
- Can you do arithmetic operations directly on the data?
No, you need a more complex algorithm
- There are two representations of zero.
- There are the same number of non-zero positive and negative digits.
- Example: 0010 = +2 1101 = -2

Two's Complement number system:

- To negate a number you invert all the bits and add 1.
- Can you do arithmetic operations directly on the data? Yes!
- There is one representation of zero.
- There are the same number of positive and negative digits, however, they are not symmetrical around zero.
- Example: 0010 = +2 1110 = -2

Two's Complement Number Systems Review

- Two's complement extends unsigned positional number systems to handle negative numbers for most modern computers.
- It is chosen because it uses the same rule for adding digits as unsigned positional numbers. The sign bit is treated just like any other digit.
- Half the bit patterns are positive (those with the sign bit of 0) and half are negative (those with the sign bit of 1). Because there is only a positive zero, there will be one more non-zero negative digit than non-zero positive digit; this will always be the most negative number possible.
- The definition of two's complement is: $N + TC(N) = 2^d$
- The two's complement of a number can be calculated in different ways.

$$TC(N) = 2^d - N$$

$TC(N)$ = flip the bits and add 1

- Complementing a positive number yields a negative number (except zero).
- Complementing a negative number yields a positive number (except the most negative number).
- An overflow on addition (incorrect result) can be detected in two ways:

If you add two numbers with like signs but the result has the opposite sign then an overflow occurred.

If the carry out of the sign bit differs from the carry into the sign bit then an overflow occurred.

- Complementing the most negative number will result in an overflow.

Hints on converting signed numbers to decimal

We have discussed three systems used to represent signed binary numbers:

- Signed Magnitude
- One's Complement
- Two's Complement

The general rule to convert *any* signed binary numbers to decimal is:

1. If the number is positive, the left most bit = 0, then just use expansion of powers to determine the decimal magnitude of the binary number. The sign of that magnitude is +.
2. If the number is negative, the left most bit = 1, then apply the rules of the signed number system being used to convert the binary number to positive. Complementing the negative number makes it positive. Now use rule #1 above to determine the magnitude of the number being converted. The sign of that magnitude is -.

Examples.

System	Sample	Pos/Neg Number	Conversion Rule	Result
Signed Magnitude	01010000	Positive	1. Expand powers. 2. Add a + sign.	80_{10} $+80_{10}$
Signed Magnitude	11010000	Negative	1. Invert sign. 2. Expand powers. 3. Add a - sign.	01010000_2 80_{10} -80_{10}

System	Sample	Pos/Neg Number	Conversion Rule	Result
One's complement	01010000	Positive	1. Expand powers. 2. Add a + sign.	80_{10} $+80_{10}$
One's complement	11010000	Negative	1. Invert all bits. 2. Expand powers. 3. Add a - sign.	00101111_2 47_{10} -47_{10}

System	Sample	Pos/Neg Number	Conversion Rule	Result
Two's complement	01010000	Positive	1. Expand powers. 2. Add a + sign.	80_{10} $+80_{10}$
Two's complement	11010000	Negative	1. Flip bits + 1. 2. Expand powers. 3. Add a - sign.	00110000_2 48_{10} -48_{10}

D - Additional Thoughts On Number Systems

One of the greatest assets you can bring to an employer is the ability to take the knowledge you have and apply it toward the solution of *new problems*.

The skill of solving new problems comes through practice.

Toward that end, we continually try to develop problems and questions that you have not seen before, but which can be solved by applying both the basic information you have learned and logic.

This section provides some of these new problems for you to solve.

Although we provide solutions, you should try to derive the solutions yourself.

Why does flipping the bits + 1 create the two's complement

Notation: N A binary number in the two's complement number system.
 $TC(N)$ The two's complement of N that is equal to $-N$.
 d The number of binary digits being used.

Given:

The definition of two's complement is:

$$N + TC(N) = 2^d$$

You calculate the two's complement by re-arranging terms: $TC(N) = 2^d - N$

We use the notation \underline{N} to represent the result of flipping the bits of N .

1. Any binary number composed of d digits will be a string of 1s and 0s.
2. If you flip all the bits of that number to calculate \underline{N} then every digit that was a 1 will now be a 0 and every digit that was a 0 will be a 1.
For example: $N=10001111 \quad \underline{N}=01110000$
3. If you add $N + \underline{N}$ you will always get a string of 1 bits that is d digits long.
That is $\underline{N} + N = \text{a string of } d \text{ bits}$

Using the value above:

$$\begin{array}{r} 10001111 \\ + 01110000 \\ \hline 11111111 \end{array}$$

4. 2^d is defined as a 1 bit followed by d 0 bits.

For example: $d=2 \quad 2^2 = 100 \quad \leftarrow 2 \text{ zeroes}$
 $d=3 \quad 2^3 = 1000 \quad \leftarrow 3 \text{ zeroes}$
 $d=4 \quad 2^4 = 10000 \quad \leftarrow 4 \text{ zeroes}$

5. Subtracting 1 from 2^d will create a string of d 1 bits.
That is $2^d - 1 = \text{a string of } d \text{ bits}$.

For example: $d=2 \quad 2^2 - 1 = 11 \quad \leftarrow 2 \text{ 1 bits}$
 $d=3 \quad 2^3 - 1 = 111 \quad \leftarrow 3 \text{ 1 bits}$
 $d=4 \quad 2^4 - 1 = 1111 \quad \leftarrow 4 \text{ 1 bits}$

6. Combining the results from #3 and #5 we get: $\underline{N} + N = 2^d - 1$
7. Rearranging terms we get: $\underline{N} + 1 = 2^d - N$
8. Therefore, flipping the bits + 1 yields the two's complement of a number.

Why does end around carry work for One's Complement

It is possible to perform addition on One's Complement data by making a simple modification to the rule for addition of unsigned positional numbers.

Any carry out of the most significant digit is added into the least significant digit (LSD). This is called end around carry.

For example, using 4 bits:

$$\begin{array}{r} +5 \\ -3 \\ +2 \end{array} \quad \begin{array}{l} \text{--->} \\ \text{--->} \end{array} \quad \begin{array}{r} 1 \ 1 \\ | \\ 0101 \\ 1100 \\ \hline 0001 \\ \text{--->} \\ 1 \\ \hline 0010 \end{array} \quad = +2$$

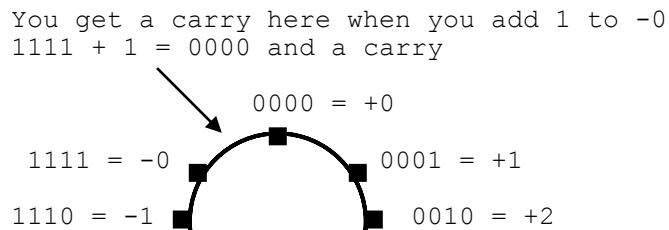
end around carry is added to LSD

Why does this work?

- With any reasonable number system, if you count sequentially by adding the number 1, you would expect to see the following progression:
... -3, -2, -1, 0, +1, +2, +3 ...
- However, with One's Complement you get this progression:
... -3, -2, -1, -0, +0, +1, +2, +3 ...

That is, when add 1 to -0 you do not get the expected result of +1. Instead you get +0. This is like saying that if you have no money and someone gives you a dollar then you still have no money. It is not logical.

- To fix this illogical fact, we want to skip over the second appearance of zero when counting by 1. Also note that when you go from -0 to +0 by adding 1 you get a carry out of the most significant bit.



- The extension for one's complement just uses the carry obtained when adding 1 to -0 to **skip over +0** and go to +1. Now the number system works logically.
- Can you ever get two end around carries in a calculation? The answer is no. Here is a simple explanation using 4 bits. To get a second end around carry, the result of the original addition would have to generate a carry (the first end around carry) and a numerical value of 1111. Then the first end around carry when added to 1111 would generate the second end around carry.

However, a result of a carry and 1111 means that the original addition generated the value 11111. This cannot happen. The largest result will be generated when you add the largest 4-bit values $1111 + 1111 = 11110$. So it is not possible to generate a carry and a result of 1111.

Why does the carry-in / carry-out rule detect a two's complement overflow

Rule: A two's complement overflow occurred if the carry out of the sign bit is not equal to the carry into the sign bit.

In reviewing textbooks, I have seen a number of textbooks that have one of these problems.

- They incorrectly explain how to detect a two's complement overflow.
- They correctly provide one of the ways to detect a two's complement overflow but do not explain why it works.
- They have no reference at all to how the overflow flag is set.

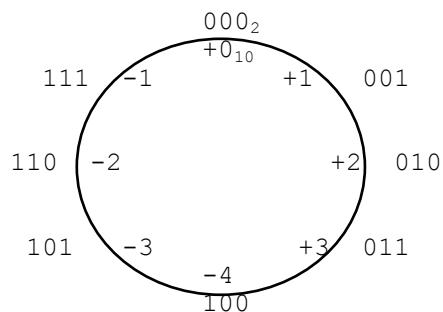
Here are some quotes from textbooks.

- *Correct but without explanation.* The hardware detects a signed overflow by comparing the carry into the sign bit with the carry out of the sign bit. If they are different then an overflow has occurred and OF is set to 1. If they are the same then no overflow occurred and OF is set to 0.
- *Correct but without enough detail.* If an arithmetic operation produces a result that exceeds the capacity of the register then the OF flag is set.
- *Incorrect.* When adding integers of the same sign, a carry into the sign position will result in the OF flag being set.
- *Incorrect.* The OF flag indicates a carry into and out of the sign bit following a signed arithmetic operation.

It is not surprising that new assembler students will be confused, if the people writing the texts do not clearly present the information.

Therefore, we present the derivation of the rule that describes how to detect a two's complement overflow.

First, the logic of the situation. We will use three bit numbers for examples.



1. Can two positive numbers be added and have the result *wrong* and *positive*?

That is, can we ever wrap around the wheel and have the sum of two positive numbers be a smaller positive number than either of the input numbers? Lets look at the worst case, where we add the largest positive number to itself. This would be decimal $(+3) + (+3)$ or binary $(011) + (011)$. The result is binary (110) . That does *not* wrap around the wheel. Instead it generates a result that is in the negative range of numbers. We conclude that if we add two positive numbers and the result exceeds the largest valid positive number then that result will be *wrong* and will appear to be *negative*.

2. Can two negative numbers be added and have the result be *wrong* and *negative*?

That is, can we ever wrap around the wheel and have the sum of two negative numbers be a less negative number than either of the input numbers? Lets look at the worst case, where we add the most negative number to itself. This would be decimal $(-4) + (-4)$ or binary $(100) + (100)$. The result is binary (000) . That does *not* wrap around the wheel. Instead it generates a result that is in the positive range of numbers. We conclude that if we add two negative numbers and the result exceeds the most negative number then that result will be *wrong* and will appear to be a *positive* number.

3. Can a positive and a negative number be added and get a result that is wrong.

Lets add all four boundary conditions.

most positive + most negative	most positive + least negative	least positive + most negative	least positive + least negative
$(+3) + (-4) = -1$	$(+3) + (-1) = +2$	$(+0) + (-4) = -4$	$(+0) + (-1) = -1$
result is okay	result is okay	result is okay	result is okay

All the operations with the boundary values are correct. Any non-boundary values will be less demanding and thus will also be correct. Therefore, if we add a positive number and a negative number then an overflow cannot occur.

This yields our basic logical rule for detecting overflows.

If you add two numbers with like signs and the sign of the result is not the same as the original signs then a signed overflow occurred. If you add two numbers with unlike signs the result will always be correct.

Now we expand our analysis to show that the rule that uses the carry into the sign position (CI) and the carry out of the sign position (CO) will detect signed overflows consistent with our logical rule previously derived.

If the CO = CI then a signed overflow did not occur.

If the CO ≠ CI then a signed overflow did occur.

Case 1. If we add two positive numbers then we have the following situation.

$$\begin{array}{r} \text{sign position} \\ \downarrow \\ 0 \ x \ x \ x \dots \\ + 0 \ x \ x \ x \dots \\ \hline \end{array}$$

The carry out of the sign position will always be zero. Why is that true?

1A. If the carry into the sign position is zero then we have the following.

$$\begin{array}{r} \text{CO=0 } 0=\text{CI} \\ 0 \ x \ x \ x \dots \\ + 0 \ x \ x \ x \dots \\ \hline 0 \end{array}$$

1B. If the carry into the sign position is one then we have the following.

$$\begin{array}{r} \text{CO=0 } 1=\text{CI} \\ 0 \ x \ x \ x \dots \\ + 0 \ x \ x \ x \dots \\ \hline 1 \end{array}$$

- In 1A two positive numbers were added and got a correct positive result. How do we know the result is correct? Remember, the logical rule previously derived specifies that if you add numbers with like signs and sign of the result has that same sign then the result is correct. Note that in this situation the CO = CI.
- In 1B two positive numbers were added and got an incorrect negative result. In this situation the CO ≠ CI.

Case 2. If we add two negative numbers then we have the following situation.

$$\begin{array}{r} \text{sign position} \\ \downarrow \\ 1 \ x \ x \ x \dots \\ + 1 \ x \ x \ x \dots \\ \hline \end{array}$$

The carry out of the sign position will always be one. Why is that true?

2A. If the carry into the sign position is zero then we have the following.

$$\begin{array}{r} \text{CO=1 } 0=\text{CI} \\ 1 \ x \ x \ x \dots \\ + 1 \ x \ x \ x \dots \\ \hline 0 \end{array}$$

2B. If the carry into the sign position is one then we have the following.

$$\begin{array}{r} \text{CO=1 } 1=\text{CI} \\ \quad 1 \ x \ x \ x \dots \\ + \underline{1 \ x \ x \ x \dots} \\ \hline 1 \end{array}$$

- In 2A two negative numbers were added and we got an incorrect positive result. In this situation the CO \neq CI.
- In 2 two negative numbers were added and got a correct negative result. In this situation the CO = CI.

Case 3. If we add a positive and a negative number then we have the following.

$$\begin{array}{r} \text{sign position} \\ \downarrow \\ \begin{array}{r} 0 \ x \ x \ x \dots \\ + \underline{1 \ x \ x \ x \dots} \end{array} \end{array}$$

The carry out of the sign position will always be the same as the carry into the sign position. Why is that true?

3A. If the carry into the sign position is zero then we have the following.

$$\begin{array}{r} \text{CO=0 } 0=\text{CI} \\ \quad 0 \ x \ x \ x \dots \\ + \underline{1 \ x \ x \ x \dots} \\ \hline 1 \end{array}$$

3B. If the carry into the sign position is one then we have the following.

$$\begin{array}{r} \text{CO=1 } 1=\text{CI} \\ \quad 0 \ x \ x \ x \dots \\ + \underline{1 \ x \ x \ x \dots} \\ \hline 0 \end{array}$$

In case 3A and 3B the CO = CI and we know from our logical rule that the results in case 3A and 3B are correct.

We have shown that the rule using the carry into the sign position and the carry out of the sign position generates the same result as our logical rule.

We derived and know the logical rule is correct. Thus this rule is also correct.

If the CO = CI then a signed overflow did not occur.

If the CO \neq CI then a signed overflow did occur.

An alternate way to calculate two's complement

A new algorithm for calculating the two's complement is being proposed.

Start at the left and invert all the bits until you reach the last 1 bit

For example: Binary original number = $\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\ \text{invert} \qquad \qquad \text{leave} \end{array}$ = +10 (dec)

↓ ↓

Binary two's complement = $\underline{\underline{1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0}}$ = -10 (dec)

Answer these questions.

- A. Why this is the same as *flipping the bits + 1* ?
- B. There is one value for which the algorithm does not work. What is that value?
- C. Modify the algorithm so that it will work for all inputs.

- A. Why is this the same as *flipping the bits + 1*

Any number to be complemented by *flipping the bits and adding 1* can be broken into two parts.

- Part-A is all bits to the left of the right most 1 bit
- Part-B is the right most 1 bit and any trailing 0 bits.

Using the example above we have:

$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\ \text{Part-A} \qquad \qquad \text{Part-B} \end{array}$

Part-B , by its definition, will always consist of a 1 bit followed by a string of 0 bits. Anytime you invert a 1 bit followed by a string of 0 bits and add 1 you end up with the original value and there will not be a carry out of the left most 1 bit. For example: $1000 = 0111 + 1 = 1000$

This means that when we complement by *flipping the bits + 1* initially Part-A and Part-B will be both be inverted.

$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\ \text{Part-A} \qquad \qquad \text{Part-B} \end{array} \rightarrow \begin{array}{r} 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \text{Part-A} \qquad \qquad \text{Part-B} \end{array} \xrightarrow{\qquad 0 \ 1 \qquad} \begin{array}{r} 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \\ \text{Part-A} \qquad \qquad \text{Part-B} \end{array}$

However, when we add the 1, Part-B will return to its original value while Part-A will remain inverted.

Well, this is the same as what the new algorithm does. It inverts all bits up to the right most 1 bit, and leaves the right most 1 bit and any trailing 0 bits alone. So the new algorithm is the same as *flipping the bits + 1*.

- B. The algorithm, as stated, will not work if all the bits are zero bits.

- C. For any non zero number, start at the left and invert all the bits until you reach the last 1 bit

Can you just look at negative number and tell its decimal value

Many people can just look at a positive two's complement value and quickly determine its value. For example 0101 is +5.

It is not so easy to just look at a negative two's complement number and determine its value. For example 1101 equals what decimal value?

We present a technique that will help.

1. Take the left most bit and assign to it, its negative positional value.
2. Treat all the other bits as positive and add their sum to the negative value.

So 1101 is calculated as follows.

$$\begin{array}{l} \overbrace{\quad\quad\quad}^{\rightarrow} = +5 \\ \overbrace{\quad\quad\quad}^{\rightarrow} = -2^3 = -8 \end{array} \quad \left. \begin{array}{l} -8 + 5 = -3 \\ \text{(flip the bits and add 1 to verify)} \end{array} \right\}$$

Why does this work?

Look at all the possible negative values using 4 bits.

binary	decimal	hi bit	low bits	hi bit neg. value	low bits sum	how far from most negative value of -8 are we ...
1111	-1	1	111	-8	7	7
1110	-2	1	110	-8	6	6
1101	-3	1	101	-8	5	5
1100	-4	1	100	-8	4	4
1011	-5	1	011	-8	3	3
1010	-6	1	010	-8	2	2
1001	-7	1	001	-8	1	1
1000	-8	1	000	-8	0	0

You can see that the sum of the low bits is exactly how far away we are from the maximum negative value, so by adding that sum to the maximum negative value we get the decimal equivalent.

Example with 8 bits.

What is the decimal value of 10101010 ?

Answer:

$$\begin{array}{r} -128 + 32 + 8 + 2 = -128 + 42 = -86 \\ \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \end{array}$$

Verify: Take 10101010 and flip bits + 1 = 01010101 + 1 = 01010110 = +86
So if the negative of this number is +86 then this number must be -86.

How can an overflow be detected in signed magnitude addition

This is the rule for adding in signed magnitude.

Case	Operation
1	If the signs are the same then add the digits using rule for unsigned PNS and the result has the sign of the input numbers
2	Else if the magnitudes are different then subtract the digits of the smaller number from the digits of the larger number and the result has the sign of the larger input number
3	Else if the magnitudes are the same then the result is zero and the sign is either + or -

In cases 2 and three, an overflow cannot occur because the magnitude of the result can never be larger than the inputs.

In case 1 there can be an overflow. The digit bits that are used in the addition are all the bits except the sign bit. If there is a carry out of these digit bits into the sign position then the result does not fit.

This leads to the following rule:

A carry out of the most significant digit bit signifies an overflow.

In the example below we add +7 and +3. The digit bits are shown in italic.

The carry out of the most significant digit,
into the sign position signifies an overflow.


$$\begin{array}{r} \mathbf{1}11 \\ 0111 \quad +7 \\ + \underline{0011} \quad +3 \\ \hline 010 \end{array} \quad \text{--> wrong magnitude for this addition}$$

How can an overflow be detected in ones complement addition

This is the rule for adding in one's complement.

Add the bits using the rule for unsigned PNS. If there is a carry out of the most significant bit then add it into the least significant bit.

We can use our basic logical rule for detecting an overflow.

You cannot have an overflow when adding unlike signs since the magnitude of the result cannot be bigger than the magnitude of the inputs.

If you add two numbers with like signs and the result has a different sign then an overflow occurred.

Example 1. There is not an end around carry

$$\begin{array}{r} +7 \\ +3 \\ \hline \end{array} \quad \left. \begin{array}{l} 0111 \\ 0011 \\ \hline 1010 \end{array} \right\} \quad \begin{array}{l} \text{positive input} \\ \text{positive input} \\ \text{negative result means an overflow} \end{array}$$

Example 2. There is an end around carry

$$\begin{array}{r} -4 \\ -4 \\ \hline \end{array} \quad \left. \begin{array}{l} 1011 \\ 1011 \\ +1011 \\ \hline 0110 \\ \rightarrow 1 \\ \hline 0111 \end{array} \right\} \quad \begin{array}{l} \text{negative input} \\ \text{negative input} \\ \text{positive result means an overflow} \end{array}$$

How can an overflow be detected when adding a two's complement signed number and an unsigned number.

- When adding unsigned binary numbers, an overflow is detected when there is a carry out of the most significant bit.
- When adding two's complement signed numbers, an overflow is detected when the carry out of the sign bit is not equal to the carry into the sign bit.

Develop a rule for detecting an overflow when you add a signed number to an unsigned number with the result being an unsigned number.

For this exercise, we assume the unsigned number is a byte (0...255), the signed number is a byte (-128...+127), and the result is an unsigned byte (0...255).

The technique will only use as test variables these bits:

- The carry flag.
- The Most Significant Bits (MSB) of the inputs.
- The Most Significant Bit (MSB) of the result.

There are *many* rules that can be developed. We start with the simplest, most intuitive, least mathematical rule.

Break the problem into 4 cases based upon the value of the MSB of the two inputs.

MSB Unsigned	MSB Signed	Unsigned byte range	Signed byte range	Unsigned Result
0	0	000 ... 127	+0 ... +127	000 ... 254 no overflow possible
1	1	128 ... 255	-1 ... -128	000 ... 254 no overflow possible
1	0	128 ... 255	+0 ... +127	overflow if result > 255
0	1	000 ... 127	-1 ... -128	overflow if result < 0

The first two cases show that if the MSB of the two inputs are the same then the result will always be in the valid range of an unsigned byte and thus an overflow is not possible.

In the third case, the MSB of the signed byte is zero. This means it looks like an unsigned value. We can, therefore, treat both numbers as unsigned values and use our standard rule for detecting overflows for unsigned numbers. A carry out of the MSB is an overflow.

In the fourth case, an overflow will occur if the magnitude of the negative number is greater than the magnitude of the unsigned number. In this case, we would be trying to generate a negative result (e.g. 17 + (-20) = -3). This can be detected by the MSB of the result becoming a 1.

These four cases lead to one of many possible rules:

```
If msb of signed equals msb of unsigned then an overflow never occurs
Else if msb signed is 0 and msb of the unsigned is 1
    then a carry out of the msb of the result indicates an overflow occurred
Else if msb of the signed is 1 and msb of the unsigned is 0
    then if msb of result is 1 an overflow occurred.
```

For those who want more math, we present the following, which is based on the previous table and uses these bits.

MSBU	left bit of unsigned input
MSBR	left bit of unsigned result
CI	carry into MSB of calculation

MSBS	left bit of signed input
-	don't care about this bit
CO	carry out of MSB of calculation

Case 1. MSB of unsigned = 0 MSB of signed = 0

Unsigned	Signed	MSBU	MSBS	CI	CO	MSBR	Notes
000...127	+0...+127	0	0	-	0	-	Always ok
0xxxxxxx	U						
+ 0xxxxxxx	S						

Case 2. MSB of unsigned = 1 MSB of signed = 1

Unsigned	Signed	MSBU	MSBS	CI	CO	MSBR	Notes
128...255	-1...-128	1	1	-	1	-	Always ok
1xxxxxxx	U						
+ 1xxxxxxx	S						

Case 3. MSB of unsigned = 1 MSB of signed = 0

Unsigned	Signed	MSBU	MSBS	CI	CO	MSBR	Notes
128...255	+0...+127	1	0	0	0	1	Ok: result < 256
128...255	+0...+127	1	0	1	1	0	Error: result > 255
1xxxxxxx	U	The error occurs when the expected result is > 255.					
+ 0xxxxxxx	S	When this occurs, the carry out is 1 and the MSB of the result is set to 0. It appears that by adding a positive value to an unsigned number, the result got smaller which is not possible.					

Case 4. MSB of unsigned = 0 MSB of signed = 1

Unsigned	Signed	MSBU	MSBS	CI	CO	MSBR	Notes
000...127	-1...-128	0	1	0	0	1	Error: result < 0
000...127	-1...-128	0	1	1	1	0	Ok: result > 0
0xxxxxxx	U	The error occurs when the expected result is < 0.					
+ 0xxxxxxx	S	When this occurs, the carry out is 0 and the MSB of the result is set to 1. It appears that by adding a negative value to an unsigned number, the result got bigger which is not possible.					

These cases can be converted into many different rules. Here are three.

1. If the MSB of the inputs are different and the MSB of the unsigned byte changes then an overflow occurred.
2. If the CO is not equal to the MSB of signed byte then overflow occurred.
3. If the signed number is positive then a carry out means overflow,
Else if the signed number is negative then no carry out means an overflow.

If those were not enough, we present one last method for detecting an overflow when adding a signed number to an unsigned number.

This is a table with all possible combinations of the left most bit of the inputs and results.

MSB Unsigned	MSB Signed	MSB Result		
0	0	0	no overflow	We added 0...127 and 0...127 and the result was still 0...127
0	0	1	no overflow	We added 0...127 and 0...127 and the result was 128...255
0	1	0	no overflow	We added 0..127 and a negative number and the result was still 0...127
0	1	1	overflow	We added 0...127 and a negative number and the result was negative which cannot be represented as an unsigned value
1	0	0	overflow	We added 128...255 and 0...127 and the result exceeded 255 which can not be represented as an unsigned byte
1	0	1	no overflow	We added 128...255 and 0...127 and the result was still 128...255
1	1	0	no overflow	We added 128...255 and a negative number and the result was 0...127
1	1	1	no overflow	We added 128...255 and a negative number and the result was still 128...255

You can use any Boolean function to detect an overflow. This is one.

If (MSBU != MSBS) and (MSBR != MSBU) then an overflow occurred.

How can you do math, such as add and subtract, in the various systems

You always have two choices.

1. Do the math using the rules of the specific system being used.
2. Convert to decimal, do the math, convert back to the system being used.

We present some examples.

Given the following 4 bit binary signed magnitude number: 1001
What is the 4 bit signed magnitude result of doubling that value.

- signed magnitude 1001 = -1
- doubling -1 yields -2
- $-2 = 1010$ as signed magnitude

This example converts to decimal, does the operation and converts result to the requested system.

Given the following 4 bit binary one's complement number: 1011
What is the one's complement result of dividing that value by 2.

- one's complement 1011 = -4
- $-4/2 = -2$
- $-2 = 1101$ as one's complement

This example converts to decimal, does the operation and converts result to the requested system.

Given these 4 bit binary signed magnitude numbers: 1011 and 0011
What is the 8 bit signed magnitude result of multiplying those two numbers.

- signed magnitude numbers
 $1011 * 0011 = -3 * +3 = -9$
- $-9 = 10001001$ as signed magnitude

This example converts to decimal, does the operation and converts result to the requested system.

Given these 4 bit binary one's complement numbers.
What is the 4 bit one's complement quotient resulting from the division.

$$11101011 / 0100 = ?$$

- $11101011 / 0100 = -20 / +4 = -5$
- $-5 = 1010$ as one's complement

This example converts to decimal, does the operation and converts result to the requested system.

Given these 4 bit binary signed one's complement numbers: 1010 and 0110
 What is the one's complement sum of the two numbers.

In this example we do the math in one's complement, using end around carry.

$$\begin{array}{r}
 1010 \\
 + 0110 \\
 \hline
 \end{array}
 \quad \left. \begin{array}{c} 111 \\ 1010 \\ 0110 \\ \hline 0000 \\ \rightarrow 1 \\ 0001 \end{array} \right\}
 \begin{array}{l}
 \text{--- Negative input} \\
 \text{--- Positive input} \\
 \text{--- Result must be correct}
 \end{array}$$

We have rules for adding numbers in signed magnitude, ones complement and two's complement. When doing subtraction we can always convert the subtraction to an addition of a complement. That means $A - B$ can always be done as $A + (-B)$

Given these 4 bit binary signed magnitude numbers: 0100 and 1001.
 What is the binary signed magnitude difference of the two numbers.

First we perform the operation by converting to decimal.

$$\begin{array}{r}
 0100 \\
 - 1001 \\
 \hline
 \end{array}
 \quad \left. \begin{array}{c} +4 \\ -1 \end{array} \right\}
 \quad \left. \begin{array}{c} +4 \\ +1 \\ +5 \end{array} \right\}
 = 0101 \text{ as signed magnitude}$$

Alternatively we work directly in signed magnitude, doing the operation $A - B$ as $A + (-B)$.

$$\begin{array}{r}
 0100 \\
 - 1001 \\
 \hline
 \end{array}
 \quad \left. \begin{array}{c} 0100 \\ + 0001 \\ \hline 0101 \end{array} \right\}
 \quad \left. \begin{array}{c} A \\ + -B \end{array} \right\}
 \quad \left. \begin{array}{l} \text{No carry out of the most significant digit} \\ \text{into the sign means the result} \\ \text{is correct.} \end{array} \right\}$$

Given these 4 bit binary signed one's complement numbers: 0100 and 1110.
 What is the binary one's complement difference of the two numbers

$$\begin{array}{r}
 0100 \\
 - 1110 \\
 \hline
 \end{array}
 \quad \left. \begin{array}{c} +4 \\ -1 \\ +5 \end{array} \right\}
 \quad \left. \begin{array}{c} +4 \\ +1 \\ +5 \end{array} \right\}
 \quad \left. \begin{array}{c} 0100 \\ + 0001 \\ \hline 0101 \end{array} \right\}
 \quad \left. \begin{array}{c} A \\ + -B \end{array} \right\}
 \quad \begin{array}{l} \text{Added two positive numbers} \\ \text{and the result is positive} \\ \text{and thus it is correct.} \end{array}$$

Additional insight

Given this two's complement hex number: ABCD
Is the number *ODD* or *EVEN*? Is the number positive or negative.

EVEN numbers are multiples of two. *ODD* numbers are not. It is not necessary to convert the whole number to decimal to see if it is a multiple of two. Only convert the low hex digit to binary ($D = 1101$). The low order binary digit has the positional value 2^0 , which is 1. All other positional values are multiples of two. So, if the low order bit is 1 then the number is *ODD*. Since in ABCD the low order bit is 1 then ABCD is *ODD*.

The sign bit is the left most bit. To determine the sign of ABCD, we only need to convert the left most hex digit to binary ($A = 1010$). The left most bit is 1 so ABCD is negative.

The two's complement number $ABCD_{16} = -21555_{10}$ which is negative and odd.

Given a 10 bit binary two's complement data field, what range of decimal numbers can be represented.

- $2^{10} = 1024$
- half the number of digits will be negative and half positive
- there is only a positive zero so there will be one more non-zero negative value than positive value
- the 512 negative values are $-512 \dots -1$
- the 512 positive values are $+0 \dots +511$
- the range of values is then $-512 \dots +511$

Given these 20 digit hex unsigned numbers, will there be an unsigned overflow if they are added together.

$$\begin{array}{r} A5E793F0A22B8C5D1F6D \\ + \underline{4BAF667DA0C5B3AA99F1} \end{array}$$

It is not always necessary to actually add the numbers to determine if an unsigned overflow will occur. An unsigned overflow occurs if there is a carry out of the left most bit. When adding two digits in any column, the only value for a carry is 0 or 1. This means that the worst case for the left most bit column will be $A+4+1$ where the 1 is a carry into that position. Well, $A+4+1=F$ and there is no carry out, so there will not be an overflow.

If the numbers were two's complement signed numbers, the top number is negative and the bottom number is positive. Thus no overflow can occur.

E - Fractions

We start with a review of decimal fractions. Fractions are just an extension of positional number systems to include negative exponents. So the number 25.73 in its true positional format is

$$2 \times 10^{+1} + 5 \times 10^0 + 7 \times 10^{-1} + 3 \times 10^{-2} = 2 \times 10 + 5 \times 1 + 7/10 + 3/100 = 25.73$$

In order to perform arithmetic, it is necessary to align the decimal points.

So $10.8 + 7.32$ must be written with the decimal points correctly lined up.

$$\begin{array}{r} 10.8 \quad \text{---> fill the empty position with 0 --->} \quad 10.80 \\ + 7.32 \\ \hline \end{array} \quad \begin{array}{r} + 7.32 \\ \hline \end{array} \quad \begin{array}{r} 18.12 \\ \hline \end{array}$$

Note that the decimal point is not physical but instead it is just a logical concept. It is the location where all digits to the left of the decimal point are integers and all digits to the right of the decimal point are fractional values.

You would get exactly the same numerical digits if you omit the decimal point and remember that the last two digits are fractions.

$$\begin{array}{r} 1080 \\ + 732 \\ \hline \end{array} \quad \begin{array}{r} 1812 \\ \hline \end{array}$$

This is really important, because many CPUs only have integer arithmetic hardware. For example, in the 8086, only fixed-point integer bytes and words can be processed. There were not enough electronic circuits on that chip to natively support floating point numbers.

So there are three basic ways to support fractions.

1. Add floating-point hardware in the form of a co-processor like the 8087.
2. Add floating-point hardware on the chip when there are enough circuits. This occurred when the 80486 was introduced.
3. Use integer arithmetic hardware and just assume that a hex point exists and all digits to the left are integers and all the digits to the right are fractions. On the 8086 one location to place this assumed hex point is between the high byte and low byte of a word. This will most easily allow one to perform addition, subtraction, multiplication and division. Unfortunately, only using one byte to store a fraction will severely limit the number of fractional digits that can be stored.

A one byte hex fraction is $?x16^{-1} + ?x16^{-2}$ where we fill in the digits. The smallest value would be $.01_{16} = 0x16^{-1} + 1x16^{-2} = .00390625_{10}$ which means that we can only handle 2 to 3 decimal digits.

Conversion of fractions among decimal, binary and hex

Conversion from **other bases to decimal** uses expansion of powers.
All the math is done in base 10 since we are converting to decimal.

Binary to decimal

$$.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$.101_2 = 1 \times .500 + 0 \times .250 + 1 \times .125$$

$$.101_2 = .500 + .000 + .125$$

$$.101_2 = .625_{10}$$

If you are working with a fixed number of binary fractional digits, then you can speed up the process. With three binary fractional digits the low order position is 2^{-3} which is 1/8 so $.101_2$ can be looked at as 5/8 which is .625

Hex to decimal

$$.A2_{16} = 10 \times 16^{-1} + 2 \times 16^{-2}$$

$$.A2_{16} = 10 \times .0625 + 2 \times .00390625$$

$$.A2_{16} = .625 + .0078125$$

$$.A2_{16} = .6328125_{10}$$

In this example we are working with 2 hex digits. The low order position is 16^{-2} which is 1/256 so $.A2$ can be calculated as 162/256 which is .6328125

Another way to look at it is:

$$.A2_{16} = \frac{10}{16} + \frac{2}{256} = \frac{160}{256} + \frac{2}{256} = \frac{162}{256} = .6328125_{10}$$

Conversion from **decimal to other bases** uses reduction of powers.
First we will do it the basic way. We find the missing binary digits.

$$.625_{10} = \underline{\quad} \times 2^{-1} + \underline{\quad} \times 2^{-2} + \underline{\quad} \times 2^{-3} + \dots$$

Step 1
Since .625 is more than .5000
the first digit is 1 and we
have .125 left to convert.

$2^{-1} = .5000$
$2^{-2} = .2500$
$2^{-3} = .1250$
$2^{-4} = .0625$

$$\begin{array}{r} .625_{10} = 1 \times 2^{-1} + \underline{\quad} \times 2^{-2} + \underline{\quad} \times 2^{-3} + \dots \\ \underline{.500} \\ .125 \end{array}$$

Step 2
Since .125 is less than .2500
the second digit is 0 and we
have .125 left to convert.

$$\begin{array}{r} .625_{10} = 1 \times 2^{-1} + 0 \times 2^{-2} + \underline{\quad} \times 2^{-3} + \dots \\ \underline{.500} \\ .125 \end{array}$$

Step 3
Since .125 is equal to .1250
the third digit is 1 and we
have .000 left to convert.

$$\begin{array}{r} .625_{10} = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + \dots \\ \underline{.500} \\ \underline{.125} \\ \underline{.125} \\ 000 \end{array}$$

The final result is: $.625_{10} = .101_2$

Speeding up the conversion of **decimal to other bases**. Follow this logic.

We know that $.625_{10} = .101_2$ from the previous page.

If we multiply both sides by 2 then the equation must still be true.
We do each multiplication in the correct base.

$$\begin{aligned}.625_{10} \times 2_{10} &= 1.250_{10} \\ .101_2 \times 2_2 &\text{ just moves the binary point and the result is } 1.01_2\end{aligned}$$

So it must be true that: $1.250_{10} = 1.01_2$

The integer parts must be same: 1 = 1 which you can see.
The fractions must be same: $.250_{10} = .01_2$ which you can show from the previous page

We can subtract the integer part from both sides yielding $.250_{10} = .01_2$

We can then repeat the process.

How does this help? *Each time we repeat the process, the binary digit directly to the right of the binary point is revealed to us. It is revealed as the integer part of the multiplication.*

Example: Convert $.625_{10}$ to binary.

To do this multiply by 2. The integer part of the result is next binary digit. Remember the digit so you can build the binary number. Remove the integer part of the result. Repeat until the result is zero or until you have enough binary digits.

$$\begin{aligned}.625 \times 2 &= 1.250 \\ .250 \times 2 &= 0.500 \\ .500 \times 2 &= 1.000\end{aligned}$$



These are the string of binary digits: $.625_{10} = .101_2$

Example: Convert $.3$ to binary.

$$\begin{aligned}.3 \times 2 &= 0.6 \\ .6 \times 2 &= 1.2 \\ .2 \times 2 &= 0.4 \\ .4 \times 2 &= 0.8 \\ .8 \times 2 &= 1.6 \\ .6 \times 2 &= 1.2\end{aligned}$$

At this point we will repeat the sequence of decimal values 6, 2, 4, 8 and they yield the repeating binary sequence 1001

$$.3_{10} = .0 \underline{1001} \underline{1001} \underline{1001} \dots_2$$

I used a calculator to convert $.0 \underline{1001} \underline{1001} \underline{1001}_2$ to decimal and got $.29993$ so it is converging on $.3$ as its value.

Example: Convert $.6328125_{10}$ to hex.

$$\begin{array}{r} .6328125 \times 16 = 10.125 \\ .125 \times 16 = 2.0 \end{array}$$

$$.6328125_{10} = A2_{16}$$

This is an example of using fractions in a computer.

Let us assume that we have a fractional decimal number such as 3.14159 , which you may recognize as PI. We wish to convert PI to a hex fractional number so we can do arithmetic operations using PI on a computer such as the 8086.

We will use one byte for the integer and one byte for the fraction.

Using a one-byte hex fraction requires us to truncate decimal PI to 3.14.

We convert 3.14_{10} to hex in two steps.

- First convert the integer.
- Second convert the fraction.

Each part is stored as a byte, which is 2 hex digits.

1. The integer 3 is easy to convert. It becomes 03_{16} .

2. To convert the decimal fraction we use the process previously shown.

$$\begin{array}{r} .14 \times 16 = 2.24 \\ .24 \times 16 = 3.84 \\ .84 \times 16 = 13.44 \end{array} \quad \left. \right\} \quad \text{This yields } .23D_{16}$$

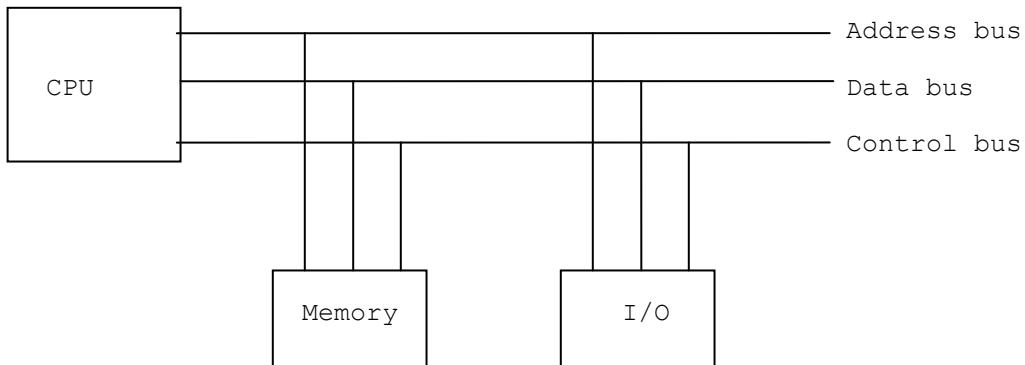
We stop, not because we have a perfect conversion, but because we can only store two hex digits in the byte we have allocated for the fraction.

We have converted three digits to allow us to decide whether to truncate or round up the result. Each hex digit can be 0-F and we round up if the last digit is 8-F. Since D is more than 8 we round the result to $.24_{16}$.

Combining the integer and fraction, $3.14_{10} = 03.24_{16}$ and if we store that in a hex word with the assumed hex point between the high and low byte we have $\text{PI}=0324_{16}$.

Basic Hardware Components Of A Computer System

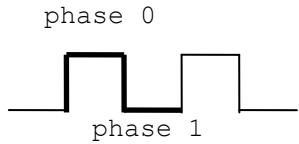
This section covers the basic hardware components of a general purpose computer: Memory, Central Processing Unit (CPU), Input/Output (I/O) devices and the communications lines (wires) that connect these components together. These lines are called buses. There are three different types of bus: address bus, data bus, and control bus. These components fit together as shown in the following diagram.



- **CPU:** The control part of the computer. Its function is to execute programs stored in the memory by fetching instructions, verifying the validity of the instructions, locating the required data, and then executing the instruction.
- **Memory:** The electronic storage media used to hold programs and data.
- **I/O devices:** The devices are used to get information into and out of memory. Examples are: keyboards, displays, printers and disks.
- **Address bus:** The address bus is a group of lines that are used to identify a location in memory that the CPU wishes to access (read or write).
- **Data bus:** The data bus is a group of lines used to carry data to or from the CPU.
- **Control bus:** The control bus is a group of lines that carry control information from the CPU out to memory or I/O devices and status information back to the CPU from memory or I/O devices. There are a multitude of different functions performed by the different control lines. Three important control lines are:
 - *address valid line* This line is set to invalid until the CPU needs the memory to perform a read or write operation. In the invalid state, memory stays idle. When the CPU has work for the memory then it sets this line to the valid state.
 - *read/write (r/w) line* When the CPU wishes to read a memory location, it places the binary address of that location on the address bus. It sets the R/W line to 'read'. It then sets the address valid line to the valid state. The memory then knows that the information from the specified memory address is to be placed onto the data bus where that data will be read by the CPU.
 - *clock* The clock line carries synchronizing information to all the components in the system so that they can determine when to perform their various functions.

The Clock And Bus Cycles

As the computer executes instructions, data is continually being moved between the CPU and memory. The timing of all these operations are controlled by the clock. The clock signal looks like a never ending square wave. Each cycle of the square wave consists of a *phase 0* and a *phase 1*. A complete cycle is called a bus cycle or machine cycle. Operations such as instruction execution and reading or writing data take place in one or more of these bus cycles.



As an example of the clocks use, lets look at reading a byte from memory.

In phase 0

- CPU puts a memory address on the address bus
- CPU sets the r/w line to read
- CPU sets the address valid line to valid
- memory becomes active when the address valid line is set to valid and it puts the requested data on the data bus

When the clock changes to phase 1

- the CPU takes the requested information off the data bus
- the CPU sets the address valid line to invalid

In many microprocessors, an instruction requires multiple clock cycles. For example a processor with a 600 MegaHertz clock (600 million cycles per second) whose average instruction requires four clock cycles, would execute about 150 million instructions per second. Not all instructions necessarily require the same number of clock cycles. For example, an *add* may use four, but a *divide* may use seventy.

Memory

The computer memory is used to hold

- instructions which are the basic units of work to the system
- data which is the information to be processed

The physical construction of memories has evolved through vacuum tubes, magnetic cores, and transistors. The one characteristic that is common to all of these physical devices is that each of their individual memory cells could be placed into two different states. This means that each individual memory cell could hold one binary digit (bit). For this reason, computing systems are built using binary arithmetic.

Typically, there is the need to work with numbers larger than 0 and 1. Therefore, several bits are grouped together to form larger data fields. Three x86 grouping are:

Name	Bits	States	Unsigned	Signed Two's Complement
Byte	8	$2^8 = 256$	0 to 255	-128 to +127
Word	16	$2^{16} = 65,536$	0 to 65,535	-32,768 to +32,767
Longword	32	$2^{32} = 4,294,967,296$	0 to 4,294,967,295	-2,147,483,648 to +2,147,483,647

There are major classifications for memory:

- Random Access Memory (RAM) is the main memory of a modern computer. It is called RAM because any byte can be directly accessed in the same amount of time. It is the normal memory for holding instructions and data. RAM can be read or written by the user's application program. RAM is volatile which means that it loses its contents when the machine is powered off. There are 2 basic technologies for building RAM.
 - Static RAM (SRAM) is very fast but very expensive.
 - Dynamic RAM (DRAM) is much slower but affordable. The main memory of a personal computer will always be DRAM because of its lower cost. Over the years faster versions of DRAM have been built and you may see these referenced in computer advertisements. These include EDO DRAM and Synchronous DRAM.
- Read Only Memory (ROM) which can only be read. Its contents are permanently burned in by the manufacturer and it does not lose its contents when the machine is powered off. It is typically used to hold the basic software needed to initialize the machine when the user turns the machine on.

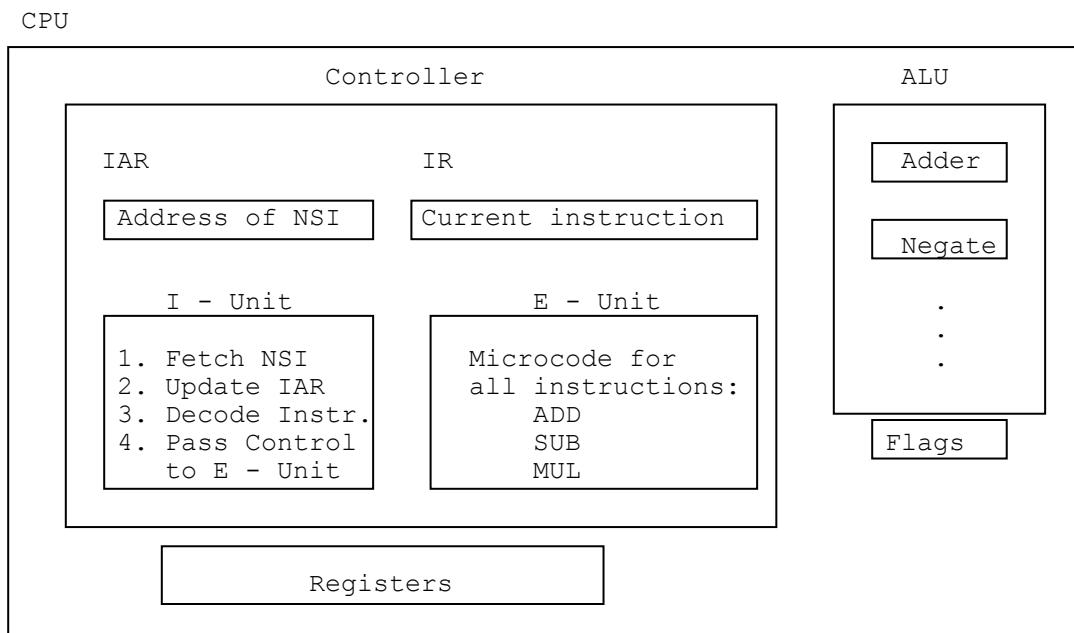
Central Processing Unit (CPU)

The CPU contains a number of functional components.

The controller manages instruction execution. To decide which instruction is to be executed next, the CPU maintains an Instruction Address Register (IAR). This is a CPU memory field that always contains the address of the next sequential instruction (NSI) to be executed.

The Instruction Decode Unit (I-Unit)

- Fetches the instruction from memory and places it into the Instruction Register (IR).
- Updates the IAR to point to the next instruction in memory.
- Decodes the bit pattern of the instruction to verify that the instruction is valid and to determine what operation is to be performed. It also locates the data needed by the instruction (effective address calculation).
- Passes control to the Execution Unit (E-Unit) to perform the desired operation.



There are different ways to build the execution portion of a CPU.

- *Hard wired* in which every instruction has its own electronic circuit, hand crafted by an engineer, to perform a specific function such as add, subtract or multiply.
- *Microprogrammed* where there are a very small number of hard wired instructions. These are called microinstructions. They control the flow of data inside the machine and they activate the real hardware in the ALU. Each assembler language instruction is built up by combining microinstructions into a microprogram. Microprogramming was very heavily used between 1965 and 1990.

What are the trade-offs between microprogramming, and a direct hard wired approach where there is a separate circuit for each instruction?

- The main advantage of microprogramming is flexibility. One can change the machine language for a computer by replacing the microprogram. One would not have to modify the actual hardware since only the most basic operations, like addition, are actually implemented in the ALU. One use for this emulation is when a new model of a computer can be used to run the software written for an older incompatible model.
- The main advantage of direct hard wiring instructions is speed since the flexibility associated with microprogrammed instructions tends to make them slower than direct hard wired instructions.

In a microprogrammed CPU, the microcode is an inner level of programming, hidden from direct view of the programmer, that controls the circuits in the CPU. For example, assume there is a machine instruction that subtracts variable *A* from variable *B*. The assembler instruction might look like SUB B,A. The microcode for that subtract instruction might look like this.

- fetch variable *A* from memory
- fetch variable *B* from memory
- calculate *TEMP* = the one's complement of variable *A*
- calculate *TEMP* = *B* + *TEMP* + 1
- route the result to variable *B*

The microcode will have a microprogram for each instruction in the machine's repertoire.

The actual hardware for performing arithmetic and logical operations is contained in the Arithmetic and Logic Unit (ALU). The various pieces of hardware in the ALU are directly controlled by the microcode. The basic components in the ALU will be an Adder (that adds two binary numbers), and a Shifter (shifts bits left or right) and Ones Complementer. There will not normally be separate circuits for subtracting since subtraction can be done by adding the two's complement of a number. Multiplication and division can be done with special hardware if speed is critical or they can be done as chains of additions or subtractions if low cost is critical.

The ALU also maintains a set of Flag bits that are set based upon the last instruction executed. These flags can be tested by the programmer to indicate whether the result overflowed, was zero, was negative, was positive.

Registers

The registers are temporary storage areas inside the CPU. Some of the registers can be controlled by the Assembler Language programmer. Some will be used only by the microcode. Very often the programmer will load data from memory into the registers. Once in the CPU's registers, the data can be used by multiple instructions. Since the data does not have to be transferred between memory and the CPU for every instruction, the system will run faster when data is kept in registers.

The number of registers is machine dependent.

Some machines have general purpose registers such as the Motorola 68000, while some machines have special purpose registers such as the Intel 8086. Some older and simpler machines may only have one register, which is usually called the Accumulator.

Buses

Buses carry information among the CPU, Memory, and I/O devices. There are three different buses.

- The Address Bus is used to select a memory location that the CPU wishes to read or write. For example, if a system supports $65,536 = 2^{16}$ bytes of memory, then the address bus will consist of 16 lines and will carry the binary number representing the desired memory address. The Address Bus is unidirectional. It carries information from the CPU out to Memory.
- The Data Bus is a bi-directional bus that carries data between the CPU and Memory. The number of lines in the Data Bus will affect system performance. It may consist of 8 lines, in which case it can carry one byte at a time, or it may consist of 16 lines, in which case it can carry one word at a time.
- The Control Bus carries control and status information. For example, it indicates whether a memory reference is for a Read or Write operation. It would also indicate when the information on the address bus is valid.

Instruction Execution: Fetch And Execute Cycle

Instruction execution consists of the following steps:

1. The next sequential instruction (which the IAR points to) is fetched from memory into the I-UNIT.
2. The IAR updated.
3. The instruction is decoded. The bit pattern is examined to ensure it is valid, and to determine what operation is to be performed. Also, the location of data used by the instruction is determined ('effective address' calculation). This is accomplished by using the 'addressing modes' encoded in the instruction. These 'addressing modes' explain how to locate the needed data.
4. Control is passed to the appropriate microcode in the E-UNIT for executing the instruction.

This process is continuously repeated as long as there are instructions to be executed and no errors are encountered. What types of errors are possible? The list is long: a branch to nonexistent memory, the attempt to execute an illegal instruction or to execute a data field, an overflow, a divide by zero, etc..

Instruction Format

Instructions have two basic parts. The first part is the Operation Code (OPCODE) which defines what function is performed by the instruction (e.g. ADD, SUB, MUL, DIV). The second part describe the operands which is the data on which the instruction will operate. Instructions always have one opcode, but they may have a different number of operands. Machines have been built that have instructions with 0,1,2 and 3 operands.

Following is an example of a machine with one operand.

- The opcode is one byte, which allows 256 different instruction types.
- In the simplest case, the operand is the binary address of the data to be used by the instruction. For a machine that supports 64 Kbytes of storage, the operand would be 2 bytes or 16 bits.

INSTRUCTION	OPCODE	OPERAND1
	1 BYTE	2 BYTES

More Detailed Example

I have a computer that

- can execute 256 different instructions
- has 8 general purpose registers and 64 Kbytes of main memory
- each instruction references one of the registers and one byte in memory
- has a clock speed of 20 MegaHertz and all instructions require 5 cycles

Terms:

- Kilo = 10^3 (Kilo = 2^{10} when referring to computer storage capacity)
 - Mega = 10^6 (Mega = 2^{20} when referring to computer storage capacity)
 - Giga = 10^9 (Giga = 2^{30} when referring to computer storage capacity)
 - Micro = 10^{-6}
 - Nano = 10^{-9}
 - MIPS = millions of instructions per second
- The instruction format for a machine instruction follows, where each letter is 1 bit position
We need 8 bits for the 256 opcodes, 3 bits for the 8 registers, 16 bits for the 64 Kbytes of memory.

Opcode Reg. Memory Address

OOOOOOOO	RRR	MMMMMMMMMMMMMMMM
----------	-----	------------------

- A clock speed of 20 Mhz. = 20×10^6 cycles/second.

$$1 \text{ cycle} = \frac{1}{20 \times 10^6 \text{ cycles/second}} \times (10^9) \text{ nanoseconds/second}$$

$$1 \text{ cycle} = .05 \times (10^{-6}) \times (10^9) = 50 \text{ nanoseconds}$$

- If an instruction requires 5 cycles then it takes $5 \times 50 = 250$ nanoseconds to execute.
- If 1 instruction takes 250 nanoseconds then the machine can execute:

$$\begin{aligned} & \frac{1}{250 \times (10^{-9}) \text{ seconds/instruction}} \\ &= .004 \times (10^9) \text{ instructions/second} = 4 \times (10^6) \text{ instructions/sec} \\ &= 4 \text{ MIPS} \end{aligned}$$

- A general formula is: **MIPS = CLOCK_SPEED / NUMBER_OF_CYCLES_PER_INSTRUCTION**

So we could also calculate $\text{MIPS} = \frac{20 \times 10^6 \text{ cycles/second}}{5 \text{ cycles/instruction}} = 4 \times 10^6 \text{ instructions/sec} = 4 \text{ MIPS}$

You can use the above formula to calculate any one of the variables if you are given the other two variables.

Intel 8086 Architecture

Intel processor timeline

In 1971, Intel introduced a 4-bit microprocessor named the 4004. The 4004 had the capabilities needed to perform basic mathematical functions. It allowed the industry to build hand-held calculators for several dollars instead of several hundred dollars. It put the slide rule manufacturers out of business and it started the personal computer revolution. The 4004 was followed by the 8008, which was used in terminals, and the 8080, which was the first general-purpose microprocessor.

In 1978, Intel introduced the 8086 as its third generation of microprocessor. The 8086 was adopted by IBM in its first Personal Computer (PC). The tremendous financial success of this venture has led to an ever-increasing powerful family of microprocessors.

The 80286, released in 1982, provided hardware assisted memory protection that supported multiprogramming; this was called *protected mode*. However, the 80286 could also run just like a fast 8086 and run any 8086 program; this was called *real mode*.

The 80386 was Intel's most elegant microprocessor. It was released in 1985. It was a true 32-bit microprocessor that could work with 32 bit data and could address 4 Gigabytes of memory. The 80386 added *virtual 8086 mode*, which allowed multiple 8086 programs to run at the same time. This feature is used when you activate multiple DOS box applications in a Windows environment.

The 80486 was released in 1989 and it continued to provide faster execution speeds and enhanced performance and a built in floating point numeric coprocessor.

In 1993, the Pentium entered the marketplace. The Pentium family contains all of the latest computer architecture enhancements, including multiple instructions being executed at the same time. These are covered in great detail in a later section these notes.

A significant contributor to the acceptance of the Intel family of processors is that each new processor is *compatible* to all the others going back as far as the 8086. This means that a customer's investment in software developed for the original IBM PC in 1981 is still valuable and that same software will run on the latest x86 processor. However, there is a price to be paid for this compatibility. To maintain compatibility, the latest Intel processors still have some of the same quirks that were used in the design of processors that are decades old. There are very powerful arguments for and against compatibility.

- Maintain compatibility with previous processors so that old software runs on the new processor.
- Redesign a new processor with the latest architectural innovations and have people redo their software.

The critical decision

It was 1978. Two companies, Intel and Motorola, had been building microprocessor chips for embedded use in devices such as calculators, ovens, cars, etc. Both were poised to introduce their third generation of microprocessors for the new emerging personal computer market.

- Motorola designed a totally new processor, the 68000. It was a state of the art processor with a 32-bit architecture and a 4 Gigabyte address space and it was completely incompatible with its predecessors.
- Intel wanted to satisfy a number of conflicting requirements. Intel wanted to keep the cost of the 8086 low. They wanted to allow old 8080 software to run on the 8086. They wanted to give the 8086 the ability to address more memory than 64 Kbytes, which was the maximum memory that the 8080 could address. These combined requirements prevented Intel from completely throwing out the whole 8080 16-bit architecture and moving to a totally new 32-bit architecture.

Segmentation

As part of Intel's compatibility goal, all the registers in the 8086 are 16 bit registers to be compatible with the 8080. Even the Instruction Pointer (IP) is 16 bits. A 16 bit IP appears to limit the memory (address space) of the 8086 to 2^{16} bytes, or 64 Kilobytes (4 hex digits). However, Intel implemented a concept called *segmentation* in which a memory (address space) of 2^{20} or 1 Megabyte (5 hex digits) can be used.

The 1 Megabyte physical memory is divided into logical blocks called segments. Each segment can have a size ranging from 16 Bytes to 64 Kbytes. At any time, a program has access to four segments: Code, Data, Stack, Extra

The 8086 has four 16 bit segment registers, one for each type of segment, which are used to access the segments.

- Code Segment is accessed via the CS register.
- Data Segment is accessed via the DS register.
- Stack Segment accessed via the SS register.
- Extra Segment accessed via the ES register.

How can a 16 bit segment register hold a 20 bit address? The answer is that all segments must start on an address that is a multiple of 16. This means the low hex digit of the segment address is zero. The Operating System must assure this fact when it creates a segment. The Operating System can then load the high 4 hex digits of the segment address into the segment register. *The zero that was thrown away will be remembered and used shortly.*

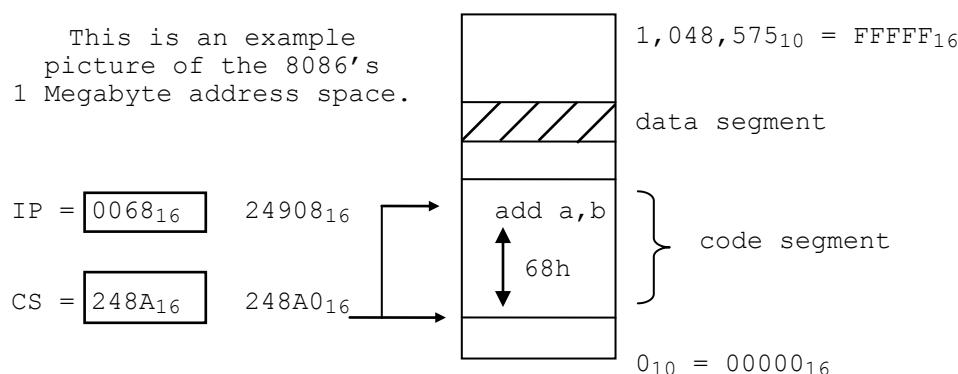
So, at any time, an 8086 program can be pointing to four segments. The contents of a segment register will be combined with an offset into the segment to produce an **absolute address** in the range 0 to 1 Megabyte as follows:

1. The value in the 16-bit segment register is multiplied by 16_{10} . Since $16_{10} = 10_{16}$ then working in hex simply requires adding a zero (the zero thrown away) to the end of the hex value in the segment register.
2. The offset into the segment is added to the value calculated above. Any overflow is ignored.
3. The result is a 20-bit number in the range of 0 to 1 Megabyte. It is the **absolute address** of the information.

$$\text{Absolute address} = \underbrace{(\text{segment_register}_{16} * 10_{16})}_{\text{Start of the segment}} + \text{Offset}_{16} \quad + \text{offset}$$

The *offset* is the location of the instruction in the Code Segment (specified by the IP register) or the location of the data item in the Data Segment (also called the effective address of the data ... see page 4-8).

For example, given a Code Segment starting at $248A0_{16}$ the Code Segment register contains $248A_{16}$. If the Instruction Pointer (IP) which points to the next sequential instruction to be executed contains 0068_{16} then the **absolute address** of the next instruction executed is $248A_{16} * 10_{16} + 0068_{16} = 24908_{16}$



The Intel CPU performs this absolute address calculation for ***every*** reference to memory. The programmer does not have to do this calculation. However, the programmer has some responsibility for maintaining the segment register.

A ***major difficulty*** arises if you have data that will ***not fit in a single segment*** (the data is > 64 Kbytes). Then the programmer must change the value in the data segment register when a segment boundary is crossed. That is a non-trivial programming exercise. In fact, it was too complicated for the early compiler writers, so languages such as PASCAL would limit the total size of code and data so that each fit within one segment.

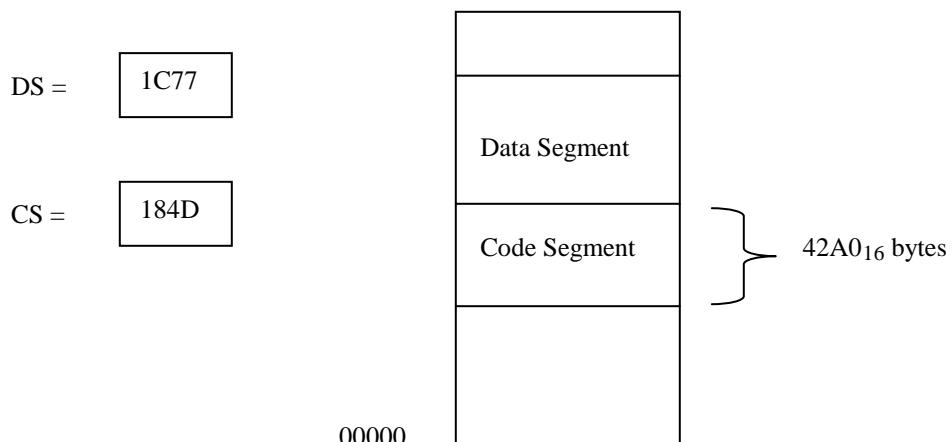
Today, compilers and assemblers have multiple memory models to let the programmer specify the segmentation option for a program. Turbo Assembler has these models: *tiny, small, medium, compact, large, huge*. The *small* model requires all code to fit in one 64K segment and all data to fit in one 64K segment.

Additional information on segments

It is not required that segments be contiguous or adjacent to each other.

However, if segments are contiguous (adjacent) then you can determine the size of the segments if you know the values loaded in the segment registers. In the example below you have the values in the CS and DS registers.

From those register values, you can determine that the Code Segment starts at $184D0_{16}$ and the adjacent Data Segment starts at $1C770_{16}$. Remember that the start of the segment is calculated by adding back the zero that was thrown away. Thus, the size of the Code Segment is $1C770_{16} - 184D0_{16} = 42A0_{16}$ bytes.



Big endian versus little endian or byte swapping

Assume you have an instruction that adds a *word* of data to a register called the accumulator. We will call it *addtoax* for this discussion. That instruction has one data argument, which is the *word* of data that you want to add. So, if you were to encounter (see or read) a request to add the *word* value 1234 to the accumulator, you would logically expect it to look something like this: addtoax 1234

Notice that as you scan (read) the instruction from its logical start to finish you see the components in this order:

- 1) the operation code *addtoax*
- 2) the high order byte of the *word* of data 12
- 3) the low order byte of the *word* of data 34

Let us only concentrate on the data. You first see the high order byte of the *word* then the low order byte of the *word*. This is quite logical and it is the way IBM hosts store *word* data and the way Motorola stores *word* data. It is called *big endian* and the derivation of that term will be covered in class.

The data size of *word* is important! The reason is that Intel does not store *word* data in memory the way we have just discussed. Intel stores *word* data in memory so that you encounter the low order byte first then the high order byte. This means the Intel instruction to add 1234 to the accumulator will be **stored in memory** as: addtoax 3412. The *word* of data is byte swapped or byte reversed. This is called *little endian* or *byte swapping* or *byte reversal*.

The storing of words in reverse order is only in main memory.

When the hardware moves the data to the CPU it will reverse the data as part of the move operation.

This means that in the CPU the data will look like 1234, which is just the way we want it to look. In summary, word size data is stored in memory with the high order and low order bytes reversed. If you define a word size variable with the hex value 1234, then in memory it will be stored as 3412. The CPU automatically makes any adjustments needed when you reference this word size variable. If you move the word into a register, the CPU swaps the bytes and the data ends up okay in the register. When you move the data from the register back to memory, the CPU again reverses the bytes.

There are two issues with storing data in little endian format:

- It may confuse the programmer when they look at data in memory.
- It is difficult to share data with a machine that stores its data in big endian format.

Why did Intel select *little endian*? Let us go back to the 1974 when Intel was designing the 8080. The 8080 could perform word size (2 byte) arithmetic. However, it could only move one byte of data at a time because its data bus was only 8 bits. Assume you have a word of data in the accumulator in the CPU with the value CDEF and you want to add the value 1234. The Intel instruction will be: addtoax 3412

The CPU, with its 8-bit data bus, can only access one byte at a time. First it accesses the operation code *addtoax*. It now knows that the two bytes that follow are to be added to the accumulator. Since we can only get 1 byte at a time, here is the big question. Which of these bytes do you add first? The answer is that you need to add the low byte first. The addition must be done in two steps. First add the low byte then add the high byte.

$$\begin{array}{r} \text{Step 1 --> } \text{CDE}\textcolor{red}{F} \\ \text{Step 2 --> } \text{CDEF} \\ \hline \text{34} \\ \text{1234} \\ \hline \text{E023} \end{array}$$

Little endian allowed a CPU with an 8-bit data bus to more easily perform 16-bit arithmetic. As bytes are fetched sequentially from memory, they are fetched in the order they are needed; low byte then high byte.

Is this needed today on a modern processor? No. The Pentium has an 8-byte data bus that moves data 8 bytes at a time. However, in order to maintain compatibility with all the processors back to the 8080, the Pentium still stores data in *little endian* format.

Programmer's model and registers

The programmer's model describes the view of the CPU as seen by the assembler language programmer or compiler writer. This is traditionally embodied in the register set for the CPU. The 8086 has the following register set.

- *Instruction Pointer*
The IP points to the next sequential instruction (NSI) in the Code Segment to be executed.
- *Segment Registers*: These point to the four active segments.
CS for the Code Segment
DS for the Data Segment
SS for the Stack Segment
ES for the Extra Segment
- *General Purpose Registers*. These can all be used as single 16 bit registers or as two 8 bit registers. Each one can be used for general arithmetic operations. Each one also has special usage in certain instructions.

Important note. When you use these registers as 8 bit registers, they are totally de-coupled. For example, a carry out of the AL register is **NOT** propagated into the AH register.

- If AX=FFFF₁₆ and you add 1 to **AX** then the new contents of AX=0000₁₆
- If AX=FFFF₁₆ and you add 1 to **AL** then the new contents of AX=FF00₁₆

Accumulator	AX when used as a 16-bit register; AH and AL when used as 8-bit registers
Base	BX when used as a 16-bit register; BH and BL when used as 8-bit registers
Count	CX when used as a 16-bit register; CH and CL when used as 8-bit registers
Data	DX when used as a 16-bit register; DH and DL when used as 8-bit registers

- *Pointer and Index Registers*: These registers are used to implement indirect addressing.

SP	Stack Pointer, which in conjunction with the SS points to the top of the stack
BP	Base Pointer, which is frequently used to point to variables stored on the Stack
SI	Source Index, which is used stepping through lists (indirect addressing)
DI	Destination Index, which is used stepping through lists (indirect addressing)

- *Flag Register*: The flags store information about the current status of the machine. These flags are often set as the result of instruction execution and allow the programmer to make decisions based upon the results of instruction execution. The main ones we will use in this class are OF, SF, ZF, CF.

OF	Overflow	1=signed overflow	0=no signed overflow
DF	Direction		
IF	Interrupt Enable		
TF	Trap (Single Step)		
SF	Sign	1=negative result	0=positive result
ZF	Zero	1=zero result	0=non zero result
AF	Auxiliary		
PF	Parity		
CF	Carry	1=unsigned overflow	0=no unsigned overflow

(Note on CF: The carry flag is set if there is a carry out of the most significant bit on an addition operation or if there is a borrow required into the most significant bit on a real subtract operation. A later chapter shows how the CF is set when doing two's complement arithmetic.)

The term *condition code* will often be used instead of flag register. For example, if an instruction modifies some of the flag bits, it will be said that the instruction modifies the condition code.

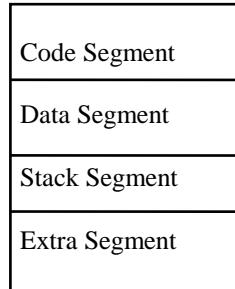
8086 programming model

Instruction pointer

Instruction Pointer

IP

Segment Registers



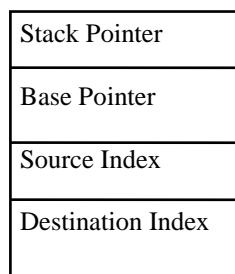
CS

DS

SS

ES

Pointer and Index Registers



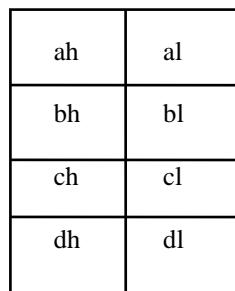
SP

BP

SI

DI

General Purpose Registers



AX Accumulator

BX Base

CX Count

DX Data

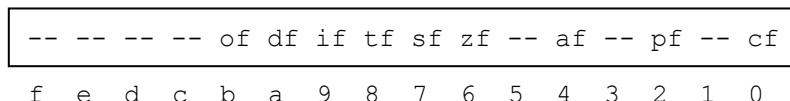
8 bits 8 bits



16 bits



Status Flags



General purpose registers versus special purpose registers

As you work with the 8086, you will find that the registers tend to be more *special purpose* as opposed to *general purpose*. This means that certain functions and instructions require the use of specific registers. For example, the Multiply instruction requires that the multiplicand must be in the AX register, and the result of the multiply is always stored in the AX register. This is different than processors such as the Motorola 68000 which has eight data registers that are all interchangeable and any one of which can be used in a multiply operation.

There are pros and cons for both approaches.

- Truly general-purpose registers make programming simpler. If the 8086 registers were truly general purpose then you could multiply the contents of any register (AX, BX, CX, DX) times another value. However, in the 8086 the AX register must be used in a multiplication. This means that if you are doing a lot of multiplies then you will be constantly loading and unloading the AX register.
- Special purpose registers, which are implied as part of an instruction, do not need to be encoded in the machine code. This means that instructions may require fewer bits thus creating smaller programs, less complicated decode logic, and possibly faster instruction execution.

Addressing Modes and Effective Address

Address Modes represent the various ways data operands for an instruction can be located. They are the instructions, provided by the programmer to the CPU, that tell the CPU where to locate data within the data segment.

For data stored in the data segment, the *effective address* of the data is the actual location of the data, within the data segment, that is calculated by following the instructions specified by the address mode.

What is the relationship between *effective address* and *absolute address*?

- Effective address is the location (0-64KB) within the data segment.
- Absolute address is the location (0-1MB) in real memory.

The effective address is the *offset* term in the equation: $\text{absolute_address} = \text{segment_register} * 10_{16} + \text{offset}$.

These are the 8086's address modes.

Register Direct: The data is held in a register. For example:

- `MOV AX,BX` moves the contents of the BX register to the AX register.

Immediate: Data is stored in the machine code of an instruction. This must be a constant value.

- `MOV AX,5` places the decimal value 5 into the accumulator.

These are the *memory reference* addressing modes. Memory references are indicated by brackets [].

The effective address is the value calculated by performing all the arithmetic within the brackets [].

In calculating the effective address, the arithmetic is performed and any carries or overflows are ignored.

Instructions with two operands do not allow both operands to be *memory references*.

Memory Direct: The user provides the actual location (offset) of a variable. This is typically done by specifying the variable's name. *The name of a variable is equated to the offset of the variable into the data segment.*

- `INC [GRADE]` Add 1 to the variable GRADE
- `INC [FIELD + 4]` Add 1 to the memory location that is 4 bytes after the variable FIELD
- `INC [BYTE PTR 32]` Add 1 to the byte at offset 32 decimal

Register Indirect: The user places the offset to a variable into one of these registers: SI, DI, BX, BP. The instruction then uses that register to indirectly access the data. References to SI, DI, BX access data in the data segment.

References to BP access data in the stack segment. High-level languages use this mode to implement pointers.

- `ADD AX,[BX]` will add the contents of memory pointed to by BX into AX

Register Indirect with Displacement: An extension of Register Indirect. A typical use is to index into a record. For example if BX points to a payroll record and the word size variable with salary information is 8 bytes into that payroll record, then the following instruction will give a \$100 raise:

- `ADD [WORD PTR BX + 8],100`

Register Indirect with Index and Displacement: This is the most general form of memory addressing that allows the user to specify two registers and an offset. One register must be either BX or BP. The other register must be SI or DI. Use of BX will get data from the data segment while use of BP will get data from the stack segment. The effective address is the sum of the contents of the two registers plus or minus the offset.

- `MOV AX,[BX + SI + 10]`

Negative displacements are okay. So these are all valid.

- `MOV AX,[SI - 100]`
- `ADD [COUNT - 5],1`
- `SUB CX,[BX + SI - 100]`

PC Assembler

The Environment

The assembler programs you write are usually *not* run totally by themselves on a clean machine. They run in a complex hardware and software environment controlled by an operating systems (DOS, WINDOWS, OS/2, LINUX, MAC). We will be using one from the dawn of the personal computer age, DOS. DOS is the acronym for Disk Operating System. It was introduced the late 1970s as the software that would control the resources of IBM PC systems.

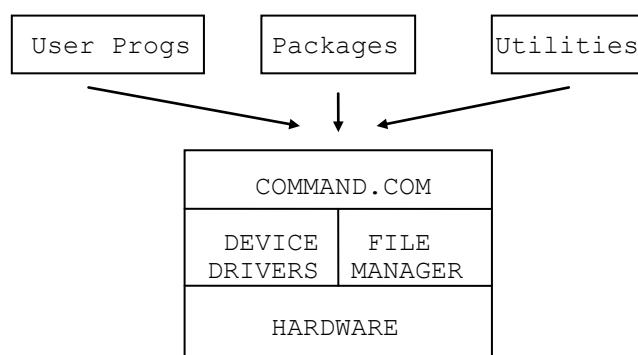
When DOS is loaded on your machine, it provides a *shell* around your computer. Other programs and commands you type pass through this shell. These programs and commands may have been written by individuals, or they may be software packages you purchased, or they may be one of the many utility programs that are shipped with DOS. The shell checks commands for correct spelling and correct parameters. The shell will load the appropriate software from disk to process the command and then pass control to that software. We need a basic understanding of this environment in order to write assembler programs for the PC.

When you power on your computer, a RESET line on the CPU will be activated. This will cause the CPU to enter a specific known state: flag register cleared; ds, es, ss segment registers set to zero; cs set to $ffff_{16}$ and ip set to zero. The CPU will then start executing instructions at the ip offset into the code segment (ffff0:0). The hardware vendor must provide a non-volatile Read Only Memory Basic Input Output System chip (ROM-BIOS) with useful code at that address. The code provided at that address is usually an unconditional jump to the *bootstrap program* written by the same ROM-BIOS vendor. This bootstrap program has the function of loading the operating system from your hard drive.

DOS has three basic components:

- COMMAND.COM that interprets commands entered by the user, and if correct then passes control to the appropriate software that will execute the command.
- The FILE MANAGEMENT system which manages the allocation of space on disks. It controls finding available space, creating files, open files for use by programs, closing files after their use, deleting files.
- DEVICE DRIVERS which are a collection of programs, each one specifically written to control the operation of a specific device such as: displays, keyboards, printers, mouse, etc.

The interaction among these components is shown below.



The way that a program or hardware device requests DOS to perform a service is through an ***interrupt***.

Examples of hardware interrupts are:

- An interrupt occurs when a user presses a key on the keyboard letting DOS know it needs to read that key stroke.
- An interrupt occurs when a disk drive has completed reading a record into memory letting DOS know that the data is now available for processing.

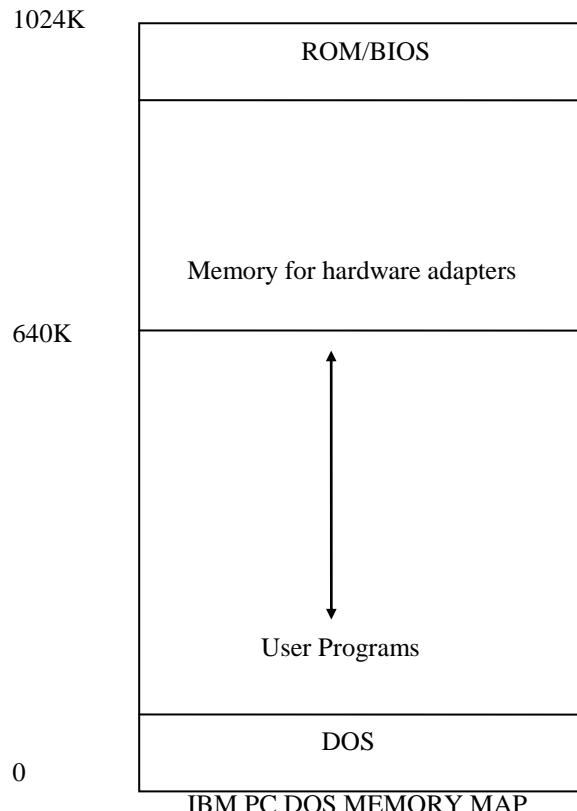
Examples of software interrupts are:

- A program generates an interrupt to request DOS write a message to the video display.
- A program generates an interrupt to tell DOS it has completed execution and can be terminated.

Software interrupts are generated by using ***int*** assembler instruction. Of special interest is interrupt 21 hex (int 21h) which is the way a program issues a function request to DOS. We will use this interrupt to request DOS provide input and output capabilities for our assembler language programs. These include: reading from the keyboard; writing to the display; reading and writing disk files.

For the sake of completeness, let me mention that the IBM PC ROM-BIOS also includes some interrupt services. These are called ROM-BIOS interrupts. BIOS stands for Basic Input Output Services. These BIOS routines are built into the non volatile Read Only Memory of the PC and therefore are available for use even if you decided to run without DOS or you wrote your own version of an operating system.

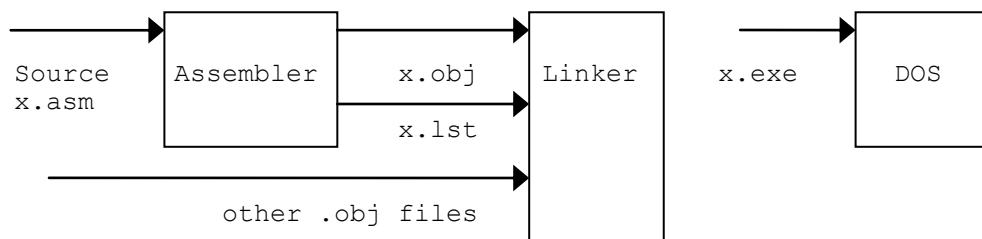
Below is a general view of how memory is laid out in a 1 Megabyte PC DOS system. Note that user programs are limited to the memory above DOS and below 640K. This typically leaves 500K Bytes to 600K Bytes for user programs depending on the version of DOS and options of DOS used.



Steps needed to develop and run a program

The process of creating an executable assembly language program requires number of steps

- Write the source program. It has a file name of your choice and file extension of *.asm*, for example myprog.asm.
- Assemble the source program. This is done by running the assembler program. It reads the source file as input and converts the symbolic assembler code into machine code. It produces an object file (myprog.obj) that contains the machine code and a listing file (myprog.lst) that shows the machine code in human readable format which can be used to help you debug the program.
- Link the object program. This is done by running the linker program. It reads the object file(s) as input and creates an executable program (myprog.exe). One of the key reasons for this linker step is that you may have written your program with a main section and a number of subroutines. Each of these parts may be contained in its own set of *.asm* and *.obj* files. The linker step will combine all these *.obj* files together and create one complete executable program.
- Load the executable program. This is done by entering the program name at the DOS prompt. COMMAND.COM will then load your program and pass control to your program.



When DOS passes control to an *.exe* program, certain registers will have been initialized.

- cs and ip are loaded with the address of the first instruction to be executed.
- ss points to the stack segment and sp points to the top of the stack.
- ds and es point to a special DOS control block that contains any command line parameters typed by the user. If the program has its own data segment then the programmer must initialize these registers in the program.

Assembler Instruction Grouping

8086 instructions are grouped into major categories. I have listed these with samples of the instructions in that category.

Data Transfer:	mov, push, pop, xchg
Arithmetic:	adds, subtracts, increment, decrement, multiply, divide, compare
Bit & Logic	not, and, or, xor, shifts and rotates
String:	string move, string compare, string scan
Transfer:	conditional jumps, call, return
Processor Control:	nop, set and clear the direction flag

Most instructions work equally well on unsigned as well as signed two's complement data. Two major exceptions exist where there are separate instructions for signed and unsigned data:

- multiply/divide operations
- conditional jumps

The conditional jumps test the state of some combination of bits in the status flag. If the condition tested is true then the jump is taken and instruction execution changes to the label specified in the operand field of the jump instruction. There are two keys to successful use of the conditional jumps.

First make sure you use the jumps that match your data (signed or unsigned).

- The jumps that are labeled above/below are used with unsigned numbers.
- The jumps that are labeled less/greater are used with signed numbers.

Second, it is important to test for an overflow as the first thing you do after arithmetic operation such as add and subtract. Remember that after an overflow the result is not valid; for example the sign of a signed number will be wrong after an overflow. This means that other tests may not be valid. The test for overflow is not required after a compare operation.

If you follow these rules you will be in good shape.

1. **Know whether you are working with unsigned or signed numbers and use the appropriate jumps.**
2. **After an arithmetic operation (such as add and subtract) always check for an overflow first. If there was an overflow then handle that condition. If there was not an overflow then make any other tests desired.**

Assembler Opcodes and Directives

General information and restrictions

- A line of assembler code has the following format. It is not case sensitive.
You must have at least one space character separating each parameter.

label: **opcode** **destination,source** **;comment**

label: is a name given to the instruction.

It is 1 to 35 alphanumeric characters. These include the letters, numbers, underscore (_), at sign (@), dollar sign (\$). The first character may not be a number. Labels terminate with a colon (:). A label is used as the target of a jump instruction; think of this as a *goto* in a high level language. Do not select a reserved word, such as the name of an instruction like *add*, as a label. This will generate an error message. You may have a label on a line by itself.

opcode specifies what instruction you want executed, such as *add* or *subtract*.

destination,source specifies the data for the instruction.

You identify the data by specifying the addressing modes for both the source and destination operands. The addressing modes do not need to be the same for both operands. For example in *mov ax,bx* both addressing modes are register direct. However, in *mov ax,[salary]* the source addressing mode is memory direct and the destination addressing mode is register direct.

;comment The semicolon (;) indicates that the rest of the line is a comment field.

You may start a line with a semicolon and that whole line will be a comment line.

- The order of the operands is significant. The left operand is the destination and the right operand is the source. For example: *mov ax,bx* copies the contents of the bx register into the ax register.
- The size of both operands must be the same. They must both be byte or they must both be word.
- When an instruction specifies two operands, they both may **not** be *memory references*. This means that it is not possible for both operands to have brackets [] which are used to indicate a memory reference.

These are okay: *mov ax,bx* *mov ax,[varx]* *mov [varx],1000*

These are not okay: *add [bx],[si]* *mov [varx],[vary]*

- An immediate operand may never be the destination operand. This is not valid: *cmp 5,ax*
- It is hard to predict which instructions set the condition code. You need to look each one up.
- The assembler assumes numbers are decimal unless you tell it otherwise. To specify a hex value, add the suffix 'h'. For example, these will accomplish the same goal: *add ax,100* *add ax,64h*
A hex value must start with a number as opposed to a letter, so that it can be distinguished from a label.
If you want to add the byte hex value ff then you would need to code this as: *add bl,0ffh*
The leading zero identifies the field as a number; the trailing 'h' identifies the number as hex. The numeric value that will be added is hex ff.
- Do not use a line continuation character to create long lines. This is unnecessary and may cause the CSC236 grading system to report incorrect instruction counts. The staff will show you alternatives to continuing lines.

Examples of all the addressing modes for data.

The following examples show how to code the various addressing modes. In these examples, the source operand is the sample being demonstrated. All of the destination addressing modes are register direct. However, the format is the same for coding any of these addressing modes as the destination.

SOURCE is sample of the address mode

mov ax,bx register direct. The contents of bx are copied into ax.

mov ax,1000 immediate. The constant 1000 (decimal) is moved to ax.

The following are considered ***memory reference*** address modes and the memory location is specified inside the brackets [].

mov ax,[varx] memory direct. The content of the variable varx is copied to ax.

mov ax,[bx] register indirect or pointer. The data pointed to by bx is copied to ax.

mov ax,[bx+10] register indirect with displacement. The effective address of the data is the contents of bx plus 10 (decimal).

mov ax,[bx+si+10] register indirect with index and displacement. The effective address of the data is the contents of bx plus the contents of si plus 10 (decimal).

As stated above, you can select any **valid pair** of addressing modes. For example: mov [varx],ax uses register direct for the source operand and memory direct for the destination operand.

Initially, you will only need to work with three addressing modes: register direct, immediate, memory direct. This yields 9 combinations of sources and destinations. Let us see which ones are legal.

Example	Legal?	Notes
mov ax,bx	Y	Sets ax equal to bx
mov ax,7	Y	Sets ax to 7
mov ax, [vara]	Y	Sets ax equal to the variable vara
mov 7,bx	N	Can never have a constant as the destination
mov 7,9	N	Can never have a constant as the destination
mov 7,[vara]	N	Can never have a constant as the destination
mov [vara],bx	Y	Sets vara equal to bx
mov [vara],7	Y	Sets vara equal to 7
mov [vara],[varb]	N	This should be legal. It makes logical sense to set vara equal to varb. However, the hardware does not support this combination.

Basic Directives

Directives are instructions to the Assembler program. They control how the Assembler processes your source code.

They do not generate executable instructions that are part of the final .exe program.
However, the db and dw directives do generate variables in the data segment.

Both the TASM and MASM versions are listed when they are different.

codeseg TASM - Start the code segment.
.code MASM

dataseg TASM - Start the data segment.
.data MASM

db Define a data byte.

dw Define a data word.

end End of the source code.
This must be the *last line* of your source code.
Anything that follows the end statement will be ignored by the assembler.

equ Equates a label to a text string.
tax equ 6 ; the assembler will replace the word tax with the number 6
loc equ var + 1 ; the assembler will replace the word loc with the phrase var + 1
accumulator equ ax ; the assembler will replace the word accumulator with the word ax

ideal Use TURBO ideal mode of the assembler. This is only used with the TURBO Assembler.

model TASM - Defines the memory model.
.model MASM
We will use *small* in CSC236 which provides 64 Kbytes of code segment and 64 Kbytes of data segment.

p8086 TASM - Only allow 8086 instructions.
.8086 MASM

stack TASM - Define the stack segment.
.stack MASM

Specific information about 8086 instructions comes from the Intel iAPX 86,88 Users's Manual (Intel Corporation 1981)

Basic Opcodes

adc	Add with carry. Adds two values together plus the carry from a previous operation. The source is added to the destination. It is used to support multi byte and multi word precision. <ul style="list-style-type: none">• adc will set the condition code. Example: <code>adc ax,bx ; ax=ax+bx+1 if the cf=1 or ax=ax+bx+0 if the cf=0</code>
add	Add. Adds two values together. The source is added to the destination. The result is stored in the first operand which is the destination. <ul style="list-style-type: none">• add will set the condition code. Example: <code>add ax,bx add [varx],1000 add si,[newval]</code>
cbw	Convert byte to word. (Note there is no operand specified, cbw always works on al) Converts the 8 bit signed value in al to a 16 bit signed value and stores the result in ax. The process involves copying the sign bit of al into all 8 bits of ah. <ul style="list-style-type: none">• cbw does not set the condition code. Example: <code>cbw</code>
clc	Clear carry flag to 0. Clears the carry flag in the condition code to zero. <ul style="list-style-type: none">• clc only affects the carry flag in the condition code. Example: <code>clc</code>
cmp	Compare. Compares two values. The compare works by subtracting the source from the destination, but not storing the result, then setting the condition code based upon result. <ul style="list-style-type: none">• compare sets the condition code. Example: <code>cmp ax,100</code> (Compare ax to decimal 100. This can be viewed as ax-100). Example: <code>cmp [varx],bx</code> (Compare the value of the variable named varx to the value in bx. This can be viewed as varx-bx)
cwd	Convert word to double. (Note there is no operand specified, cwd always works on ax) Converts the 16 bit signed value in ax to a 32 bit signed value and stores the result in the dx:ax pair. The process involves copying the sign bit of ax into all 16 bits of dx. <ul style="list-style-type: none">• cwd does not set the condition code. Example: <code>cwd</code>
dec	Decrement. Subtracts 1 from a value (register or variable). <ul style="list-style-type: none">• dec sets the condition code except the carry flag. Examples: <code>dec ax dec [varx]</code>

inc	<p>Increment.</p> <p>Adds 1 to a value (register or variable).</p> <ul style="list-style-type: none"> • inc sets the condition code except the carry flag. <p>Examples: inc ax inc [varx]</p>														
int	<p>DOS interrupt.</p> <p>Calls a DOS service routine. This is discussed in detail in a later section of the class notes.</p>														
jmp	<p>Unconditional jump.</p> <p>Jumps to the location specified by the operand. This is like a high level language ‘goto’.</p> <ul style="list-style-type: none"> • jmp does not set the condition code. <p>Example: jmp process</p>														
jxx	<p>Conditional jump.</p> <p>Jumps to the location specified by the operand if specific flags in the condition code are set.</p> <p>This is a conditional jump similar to the following in a high level language <i>if x > 0 then goto process</i>.</p> <p>The jxx instruction is covered in detail later in these class notes.</p> <ul style="list-style-type: none"> • jxx does not set the condition code. <p>Note: The conditional jump has a limited range for its jump distance. If you exceed this limit then you will get an error message indicating the jump is out of range. One technique to get around this is to re-code the test as in the following example where the <i>je to zero is out of range</i>. The jmp instruction is not limited in its jump distance.</p> <table border="0"> <tr> <td><u>Problem: the jump to zero is too far.</u></td> <td><u>The fix to that problem.</u></td> </tr> <tr> <td>cmp ax,0</td> <td>cmp ax,0</td> </tr> <tr> <td>je zero</td> <td>jne notzero</td> </tr> <tr> <td>mov [val],ax</td> <td>jmp zero</td> </tr> <tr> <td>-</td> <td>notzero: mov [val],ax</td> </tr> <tr> <td>-</td> <td>-</td> </tr> <tr> <td>zero:</td> <td>zero:</td> </tr> </table>	<u>Problem: the jump to zero is too far.</u>	<u>The fix to that problem.</u>	cmp ax,0	cmp ax,0	je zero	jne notzero	mov [val],ax	jmp zero	-	notzero: mov [val],ax	-	-	zero:	zero:
<u>Problem: the jump to zero is too far.</u>	<u>The fix to that problem.</u>														
cmp ax,0	cmp ax,0														
je zero	jne notzero														
mov [val],ax	jmp zero														
-	notzero: mov [val],ax														
-	-														
zero:	zero:														
mov	<p>Move.</p> <p>The mov instruction <i>copies</i> the source variable into the destination variable.</p> <p>The source variable is not changed or affected by the mov instruction.</p> <p>Restrictions include:</p> <ul style="list-style-type: none"> - you cannot move immediate data to a segment register - you cannot move data between two segment registers • mov does not set the condition code. <p>Examples: mov ax,[varx] mov [varx],1000</p>														
neg	<p>Negate.</p> <p>Sets a value to its two’s complement. It is the same as subtracting the value from zero.</p> <ul style="list-style-type: none"> • neg sets the condition code. <p>Examples: neg bx neg [varx]</p>														

sbb	<p>Subtract with borrow.</p> <p>Subtracts the source and the borrow from the previous operation from the destination.</p> <p>It is used to support multi byte and multi word precision.</p> <ul style="list-style-type: none"> • sbb will set the condition code. <p>Example: sbb ax,bx ; ax=ax-bx-1 if the cf=1 or ax=ax-bx-0 if the cf=0</p>
stc	<p>Set carry flag to 1.</p> <p>Sets the carry flag in the condition code to one.</p> <ul style="list-style-type: none"> • stc only affects the carry flag in the condition code. <p>Example: stc</p>
sub	<p>Subtract.</p> <p>Subtracts two values. The source is subtracted from the destination.</p> <p>The result is stored in the first operand which is the destination.</p> <ul style="list-style-type: none"> • sub will set the condition code. <p>Example: sub ax,bx sub [varx],1000 sub si,[newval]</p>

Defining data

In high level languages you can differentiate between named *constants* and named *variables*. However, in reality both are just specific examples of *data*. In assembler we use the terms *data* and *variables* to mean the same thing and a constant is just a variable that you declare but never modify.

Data is defined using the db (define byte) and dw (define word) directives.

Data declarations have 4 parts:

1. *name* Similar to a label but without the colon. The name is optional. Creating data without a name is often used when building lists or tables.
2. *directive* (db or dw) The db directive creates a 1 byte value and the dw directive creates a 2 byte value.
3. *value* Any value, signed or unsigned, that fits in a byte for db or that fits in a word for dw.
4. *comment* Same as a line comment for an instruction. It explains how the data item will be used.

Creating byte and word variables

In high-level languages you declare and initialize variables in this manner

```
unsigned char    varx=255;           //unsigned byte  
char            minus1=-1;          //signed byte  
unsigned         thousand=1000;       //unsigned word  
int              negval=-1000;        //signed word  
char            letter='A';         //unsigned byte
```

In assembler you declare and initialize those same variables in this manner:

```
varx    db    255    ; byte variable with the value FFh and should be used as unsigned  
minus1   db    -1     ; byte variable with the value FFh and should be used as signed
```

Important ... note the value -1 and 255 generate the same hex code. It is up to the programmer to know whether the value FFh is to be treated as the signed variable -1 or the unsigned variable 255

```
thousand dw    1000    ; creates a word variable named thousand with value 03E8h  
                      ; remember byte reversal stores it in memory as E803  
                      ; it can be signed or unsigned ... only the programmer knows  
  
negval    dw    -1000   ; word variable with the value FC18h and should be used as signed  
                      ; remember byte reversal stores it in memory as 18FC  
  
letter     db    'A'     ; byte variable with the value 41h and should be used as unsigned  
  
answer    db    ?       ; creates a variable but does not set its value
```

Creating a list of variables

You can create a list of variables. The name of the list is equal to the address of the first value in the list. The variables in the list are accessed through indirect addressing techniques.

In high-level languages you declare and initialize a list of variables in this manner:

```
char    mylist[ ] = { 100,0,5,50};
```

In assembler you can declare and initialize a list of variables in this manner:

```
mylist  db      100,0,5,50      ; creates 4 hex bytes 64 00 05 32
```

Creating an ASCII string of characters

In high-level languages you create a string with:

```
char message[ ] = "A0a9Zz"
```

In high-level languages there is often some magic that occurs. The compiler may actually generate 7 bytes of data by adding a string terminator, which is a binary zero, to the end of the list. In assembler there is no magic. The string is created without the string terminator being added for you.

```
message db 'A0a9Zz'           ; creates 1 hex byte for each character 41 30 61 39 5A 7A
```

If you want the message to terminate with an end of string indicator then build a list.

```
message db 'A0a9Zz',0         ; creates 1 hex byte for each character 41 30 61 39 5A 7A 00
```

If you want a message to terminate with a DOS newline and DOS end of string indicator then build this list.

```
outmsg db 'ABC',13,10,'$' ; creates this list 41 42 43 0D 0A 24
```

Note that the DOS newline is a pair of characters: Carriage Return (CR=13) and Line Feed (LF=10). The DOS string terminator is a \$.

Creating a one dimensional array

A one dimensional array is just like a list of variables except that you initialize all elements to a single value. You define an array and initialize all the values as follows.

```
buf      db    50 dup (' ')   ; creates 50 bytes all set to an ASCII blank  
nums    db    100 dup (0)     ; creates 100 bytes all set to decimal/hex/binary zero  
bignum  dw    75 dup (-1)    ; creates 75 words all set to hex FFFF
```

Again, the name of the array is equal to the address of the first byte in the list. You need to use indirect addressing techniques to access the elements of the list.

More complicated uses of *dup*

The *dup* directive will duplicate everything within the parentheses including other dup statements.

```
table    db      3 dup ( 'ABC', 2 dup ( 13, 10 ) )
```

The above creates the following hex data: 41 42 43 0D 0A 0D 0A 41 42 43 0D 0A 0D 0A 41 42 43 0D 0A 0D 0A

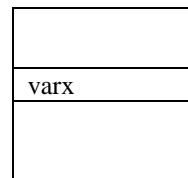
Additional important information on data and defining data

- The name of a variable is equal to the offset of the variable into the data segment.

For example, if the variable named varx is located at 12 bytes into the data segment, then all of these assembler statements are the same.

```
mov ax,[varx]  
mov ax,[12]  
mov ax,[000Ch]
```

12 bytes



- A very important concept about data is that *the Assembler generates and places data in memory in exactly the same order as you declare the data*. It does not create delimiters or in any other way mark data boundaries. The first item starts at offset zero into the data segment and each subsequent data item is placed sequentially in memory. For example, following is a set of data declarations and the hex bytes generated by the Assembler.

```
.data  
val1    db      0,1,-1          ; data generated is 00 01 FF  
           db      'Az09'         ; data generated is 41 7A 30 39      (note that a name is not required)  
val2    dw      255,-1,1234h    ; data generated is FF 00 FF FF 34 12 (words are byte reversed)
```

In memory these data items just appear as a linear string of bytes starting at offset 0 into the data segment. A total of 13 bytes are required to hold the data generated by these 3 data declarations.

val1 is located at offset 0 into the data segment



val2 is located at offset 7 into the data segment



Offset into data segment	0	1	2	3	4	5	6	7	8	9	A	B	C
Hex data	00	01	FF	41	7A	30	39	FF	00	FF	FF	34	12

- Do not use any form of line continuation character to create multiple line data definitions. This is unnecessary and may cause the CSC236 grading system to report incorrect instruction counts. The staff will show you alternatives to continuing lines.

Let us write some code

Let us write the assembler code to calculate: $c = a + b$ where a, b, c are unsigned bytes.

First we declare the variables:

```
a      db      10    ;byte variable named a  
b      db      55    ;byte variable named b  
c      db      0     ;byte variable named c
```

Next we write the instructions. Since our variables are byte size, we will use the byte registers for performing the arithmetic. We will show the full word register contents in the comment field.

```
mov    al,[a]   ;ax = ?? 0A          The ?? indicate that the value in ah is unknown  
add    al,[b]   ;ax = ?? 41          since we are performing byte operations  
mov    [c],al   ;c = a + b = 41h    in the byte register al.
```

Some notes.

1. In a real program, where we do not know the values of the data, the add operation could cause an overflow. We will learn how to check for that overflow.
2. The exact same code we wrote would also work if a, b, c were signed values.

How does it all come together for TASM

The following is a sample of a *complete* program. It includes:

- documentation header
- data segment
- code segment
- stack segment
- comments
- DOS termination code

*This is the basic model for **TASM** to follow when writing your code.*

```
;-----  
; program: ABC *TASM* version  
; function: Calculates the value of a+b and stores the result in c  
; owner: Dana Lasher  
; date: Changes  
; 05/15/2010 Original version  
;-----  
        ideal          ;use turbo ideal mode  
        model small    ;64k code and 64k data  
        p8086          ;only allow 8086 instructions  
        stack 256      ;reserve 256 bytes for the stack  
;-----  
        dataseg        ;start the data segment  
;-----  
a      db   10           ;byte variable named a  
b      db   55           ;byte variable named b  
c      db   0            ;byte variable named c  
;-----  
        codeseg        ;start the code segment  
;-----  
start:  
        ;  
        mov   ax,@data    ;establish addressability to the  
        mov   ds,ax        ;data segment for this program  
;-----  
; perform the calculation  
;-----  
        mov   al,[a]        ;get the value of a  
        add   al,[b]        ;calc a+b  
        mov   [c],al        ;store the result in c  
;-----  
; terminate program execution  
;-----  
exit:  
        ;  
        mov   ax,4c00h     ;set dos code to terminate  
        int   21h          ;return to dos  
        end   start        ;mark the end of the source code  
;.....specify where you want the  
;.....program to start execution  
;-----
```

How does it all come together for MASM

The following is a sample of a *complete* program. It includes:

- documentation header
- data segment
- code segment
- stack segment
- comments
- DOS termination code

*This is the basic model for **MASM** to follow when writing your code.*

```
;-----  
;   program:    ABD *MASM* version  
;   function:   Calculates the value of a+b and stores the result in d.  
;               'c' is changed to 'd' since 'c' is a reserved word in MASM :(  
;   owner:      Dana Lasher  
;   date:       Changes  
;   05/15/2010  Original version  
;  
;-----  
        .model      small           ;64k code and 64k data  
        .8086          ;only allow 8086 instructions  
        .stack      256            ;reserve 256 bytes for the stack  
;  
        .data          ;start the data segment  
;  
a      db      10             ;byte variable named a  
b      db      55             ;byte variable named b  
d      db      0              ;byte variable named d  
;  
        .code          ;start the code segment  
;  
start:          ;  
        mov      ax,@data          ;establish addressability to the  
        mov      ds,ax            ;data segment for this program  
;  
; perform the calculation  
;  
        mov      al,[a]           ;get the value of a  
        add      al,[b]           ;calc a+b  
        mov      [d],al           ;store the result in d  
;  
; terminate program execution  
;  
exit:          ;  
        mov      ax,4c00h          ;set dos code to terminate program  
        int      21h              ;return to dos  
        end      start            ;marks the end of the source code  
                                ;and specifies where you want the  
                                ;program to start execution  
;
```

Important concepts about the ABC / ABD program

- These directives tell the Assembler what options you want to use.

TASM	MASM	Function
ideal		use turbo ideal mode
model small	.model small	64k code and 64k data
p8086	.8086	only allow 8086 instructions
stack 256	.stack 256	reserve 256 bytes for the stack

- This directive starts the data segment.

TASM	MASM	Function
dataseg	.data	start the data segment

- This directive starts the code segment.

TASM	MASM	Function
codeseg	.code	start the code segment

- When your program is executed, the cs register and ss register will be initialized by the operating system. The programmer must initialize the ds register and es register, if they are needed.

If you have data in the data segment then you must initialize the ds register.

If you do not have data in the data segment then it is not necessary to initialize the ds register.

In CSC236 we will not use the extra segment so the es register is not initialized in our examples. These instructions initialize the ds register.

```
mov      ax, @data           ;establish addressability to the
mov      ds, ax               ;data segment for this program
```

- To terminate your program and have the prompt displayed on the screen, you must return to the operating system. This is done by executing the *int* (interrupt) instruction. The operand, 21h, specifies this interrupt as a DOS function request. The value in the ah register tells DOS what function is requested. There are about 100 different functions DOS will perform: clock functions, input and output functions, disk management. The 4C in the ah register tells DOS that this is a request to terminate the program. The value in the al register specifies the return code from this program. This is exactly the same value that you would specify in a high-level language exit statement such as *exit(0)*.

```
mov      ax, 4c00h           ;set dos code to terminate
int      21h                 ;return to dos
```

- The *end* directive tells the assembler this is the end of the source code file. Any lines after the *end* statement will be totally ignored by the assembler. The operand specifies the label of the instruction that is to be the first instruction executed by the program when DOS runs the program.

```
end      start              ;begin execution with instruction
                    ;labeled start:
```

Converting Bytes To Words ... (casting)

Let us extend the example so that **a** and **b** are bytes but the value of **c** is a word.
The idea is similar casting in a high level language.

We want to add a set of byte values, like test grades, where the sum is too big to fit in a byte. For this function we will need different code depending on whether the data is signed or unsigned.

The reason we need different code for unsigned and signed data is that:

- To convert an unsigned byte to an unsigned word we *always* pad the high byte with zeroes ($01_{16} \rightarrow 0001_{16}$).
- To convert a signed byte to a signed word we pad the high byte with the *sign value* of the data item.

This is called sign extension. For example, $+15_{10} = 0F_{16}$ and since its sign is positive we pad with zero bits ($0F_{16} \rightarrow 000F_{16}$), but $-15_{10} = F1_{16}$ and since its sign is negative we pad with 1 bits ($F1_{16} \rightarrow FFF1_{16}$).

Code to convert a byte to a word.

```
a      db      ?      ;byte variable named a whose value is not known  
b      db      ?      ;byte variable named b whose value is not known  
c      dw      0      ;word variable named c that is initialized to zero
```

In the comments below we use *aa* and *bb* to mean the hex value of the variable a and b.

Unsigned Data

```
mov    al,[a]  ;pick up a  ax=?? aa  
mov    ah,0    ;ax=00 aa  
mov    bl,[b]  ;pick up b  bx=?? bb  
mov    bh,0    ;bx=00 bb  
add    ax,bx  ;ax=a+b  
mov    [c],ax  ;c=a+b
```

Signed Data

```
mov    al,[a]  ;pick up a  ax=?? aa  
cbw  
mov    [c],ax  ;save the value in c  
mov    al,[b]  ;pick up b  ax=?? bb  
cbw  
add    [c],ax  ;c=a+b as signed words
```

The signed data example demonstrates one issue with special purpose registers.

The cbw instruction (convert byte to word) only works on the data in the al register.

This means that if we want to convert a number of signed bytes to signed words we need to continually load and unload the accumulator. We cannot convert those bytes in the other registers such as bl or cl or dl.

Jump Instructions

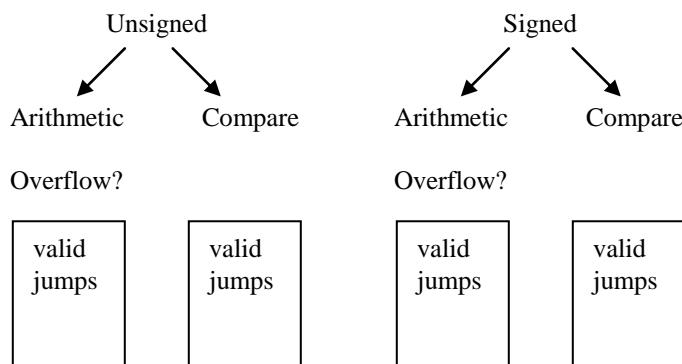
The *jump* instructions allow you to control and change the flow of instruction execution.

- An *unconditional jump* is like a high level language *goto* (goto the label specified).
 - A *conditional jump* tests flags in the status register. If the flags match the desired setting then the jump is taken. This makes it like an *if* statement: *if the flags tested have the correct values then goto the label.*

Unconditional Jump

Conditional Jumps - Used after Arithmetic Operations or Comparisons

There are four possible situations. You are working with unsigned or signed data, and you have just completed an arithmetic operation such as add or subtract or you have just completed a compare instruction.



For each of the four cases, there are a set of conditional jumps that may be used.

To use the jumps correctly, follow these rules:

1. Select the jump from the appropriate column.
 2. After an arithmetic operation such as add or subtract, first assure that an overflow did not occur. If there was an overflow then handle that condition; if there was not an overflow then make any other tests desired. *The arithmetic tests only have a valid meaning when there was not an overflow.*

Conditional Jump Summary

After an arithmetic operation such as add or subtract, first assure an overflow did not occur.

If there was an overflow handle that condition; if not then make any other tests desired.

The arithmetic tests only have a valid meaning when there was not an overflow.

Unsigned Conditional Jumps (these use the terms *above* and *below*)

<u>Instruction</u>	<u>Name</u>	<u>Condition tested</u>	<u>Jump taken if flags have these values</u>
		Arithmetic	Compare
ja	above	no overflow and result $\neq 0$	dest > source $(cf = 0) \text{ and } (zf = 0)$
jae	above or equal	no overflow	dest \geq source $(cf = 0)$
jb	below	overflow	dest < source $(cf = 1)$
jbe	below or equal	overflow or result $= 0$	dest \leq source $(cf = 1) \text{ or } (zf = 1)$
jc	carry	overflow	dest < source $(cf = 1)$
jnc	no carry	no overflow	dest \geq source $(cf = 0)$

Signed Conditional Jumps (these use the terms *greater* and *less*)

<u>Instruction</u>	<u>Name</u>	<u>Condition tested</u>	<u>Jump taken if flags have these values</u>
		Arithmetic	Compare
jg	greater	result > 0	dest > source $(sf = of) \text{ and } (zf = 0)$
jge	greater or equal	result ≥ 0	dest \geq source $(sf = of)$
jl	less	result < 0	dest < source $(sf \neq of)$
jle	less or equal	result ≤ 0	dest \leq source $(sf \neq of) \text{ or } (zf = 1)$
jo	overflow	overflow	$(of = 1)$
jno	no overflow	no overflow	$(of = 0)$
js	sign	left most bit $= 1$	$(sf = 1)$
jns	no sign	left most bit $= 0$	$(sf = 0)$

Signed and Unsigned Conditional Jumps

<u>Instruction</u>	<u>Name</u>	<u>Condition tested</u>	<u>Jump taken if flags have these values</u>
		Arithmetic	Compare
je	equal	result $= 0$	dest = source $(zf = 1)$
jne	not equal	result $\neq 0$	dest \neq source $(zf = 0)$

Conditional jump coding examples

; Calculate c=a+b where a,b,c are *unsigned* bytes.

```
a    db    ?          ; unsigned byte variable named a
b    db    ?          ; unsigned byte variable named b
c    db    ?          ; unsigned byte result named c

        mov    al,[a]      ; get the value of a
        add    al,[b]      ; calculate a+b
        jc     error       ; if an unsigned overflow occurred then goto error
        mov    [c],al      ; else set c=a+b
```

; Calculate c=a+b where a,b,c are *signed* bytes.

```
a    db    ?          ; signed byte variable named a
b    db    ?          ; signed byte variable named b
c    db    ?          ; signed byte result named c

        mov    al,[a]      ; get the value of a
        add    al,[b]      ; calculate a+b
        jo    error        ; if a signed overflow occurred then goto error
        mov    [c],al      ; else set c=a+b
```

; If grade, an unsigned byte, is zero then goto trouble.

```
grade  db    ?          ; unsigned byte with a test grade of 0...100

        cmp    [grade],0    ; grade : 0
        je     trouble       ; if grade equals zero goto trouble
```

; If speed, an unsigned byte, is greater than 55 then goto ticket.

```
speed  db    ?          ; unsigned byte with the car's speed

        cmp    [speed],55    ; speed : 55
        ja     ticket        ; if speed > 55 goto ticket
```

; If temp (in Fahrenheit) is 212 degrees or more then goto will_boil

; else if temp is 32 degrees or less then goto will_freeze

; else goto nice

;

; Since the range of temperatures can be negative through more than +212

; the only choice for temp is signed word.

;

temp dw ? ; signed word

```
        cmp    [temp],212    ; temp : 212
        jge    will_boil    ; if temp ≥ 212 then goto will_boil
        cmp    [temp],32     ; else temp : 32
        jle    will_freeze  ; if temp ≤ 32 then goto will_freeze
        jmp    nice         ; else goto nice
```

; Determine if an ASCII variable named char is an ASCII numeric in the range of '0' to '9'.
 ; If char is an ASCII numeric then convert it from ASCII to hex.
 ; Any ASCII character would be considered an unsigned byte.

```

char db ? ; an ASCII character ... an unsigned byte

cmp [char], '0' ; char : ASCII zero
jb not_num ; if char < '0' then it is not an ASCII numeric
cmp [char], '9' ; char : ASCII nine
ja not_num ; if char > '9' then it is not an ASCII numeric
sub [char], 30h ; convert ASCII to hex by subtracting 30h
  
```

; Because the jump instructions do **not** change the condition code,
 ; multiple sequential jumps are allowed.
 ; Remember that an arithmetic result is always tested relative to zero.

```

add ax,bx ; calculate ax = ax + bx where all the values are signed
jo error ; if a signed overflow occurred then goto error
je is_zero ; else if the result is equal to zero go process that condition
jg is_greater ; else if the result is greater than zero go process that condition
jl is_less_than ; else if the result is less than zero go process that condition
  
```

; Here is a looping example.
 ; What will be in the ax register when the code reaches the instruction labeled fin:

```

mov ax,8 ; ax=8
mov bx,3 ; bx=3
repit: sub ax,bx ; 8-3=5
        ja repit ; no overflow so loop → 5-3=2
fin:    ; ax = ffff ; no overflow so loop → 2-3=ffff
        ; cf=1 so fall thru
  
```

A banking application

This final example is a slightly more complicated banking application. We have three variables:

- *balance* contains a checking account balance
- *deposit* contains an amount of money to be added to the checking account
- *check* contains the amount of a check written against the account

When our code gets control the data in these variables has been set and verified as valid.

Specifically, the values in deposit and check are both greater than zero.

At our bank we allow the customer to make one overdraft at a time. Since we allow an overdraft, we need to make balance a signed word. Once making that decision, it will make life easier if we also make deposit and check signed words even though they logically could never contain negative values.

We will code two parts of the application, adding the deposit to the balance and cashing the check. When we cash the check, we allow the overdraft. That means we will cash any check as long as the current balance is positive. If the current balance is negative then we will not cash the check.

Declare the required variables

```
balance dw    ?          ; checking account balance ... signed word  
deposit dw   ?          ; amount to be added to the checking account ... signed word  
check   dw   ?          ; amount of a check written ... signed word
```

Make the deposit

```
; balance = balance + deposit  
; if an overflow occurs then go to an error routine  
  
        mov     ax,[deposit]    ; get the deposit amount  
        add     [balance],ax    ; new balance = old balance + deposit  
        jo      error1         ; goto error routine if an overflow occurred
```

Cash the check

```
; if balance < 0 then goto bounce  
; else  
; {  
;   balance = balance - check      ??? Do we need to check for an overflow after the subtraction ??? ←  
;   if new balance < 0 then goto charge_interest  
; }  
        cmp     [balance], 0      ; balance : 0  
        jl      bounce           ; if the current balance is negative then do not cash the check  
        mov     ax,[check]        ; else get the check amount  
        sub     [balance],ax      ; new balance = old balance - check  
        jl      charge_interest  ; if the new balance is negative then we had an overdraft
```

An overflow check is not needed after the subtraction.

In order to get to this instruction the smallest value in balance is 0.

The largest value in check is 32767.

Thus the worst case for this calculation is $0 - 32767 = -32767$ which did not overflow

Things to remember when designing your code

You must first decide on how to declare your variables:

- Should the size be byte or word. This will affect which registers are used to process the data. Byte size data will use the 8 bit registers such as *al* while word size data will use the 16 bit registers such as *ax*.
- Whether they are unsigned or signed. This will affect which conditional jumps are used, unsigned or signed conditional jumps.

This is **not** a new programming concept to you. You did the equivalent when you coded C/C++.

- The declaration: int varx implies a word size signed variable.
- The declaration unsigned vary; implies a word size unsigned variable.
- The declaration: unsigned char ch; implies a byte size unsigned variable.

Once *you* have declared the variable in C/C++ then the *compiler* will select the correct conditional jumps.

Here is an example:

- The variable *a* is declared as a signed word integer.
- The variable *b* is declared as an unsigned word value.

Note that the compiler selected *different* jumps.

C code	Assembler code generated by the C compiler
int a; unsigned b;	a db ? b db ?
if (a<100) a++;	cmp [a],100 ; a : 100 jge \$I167 ; if a ≥ 100 don't inc inc [a] ; a = a + 1 \$I167:
if (b<100) b++;	cmp [b],100 ; b : 100 jae \$I168 ; if b ≥ 100 don't inc inc [b] ; b = b + 1 \$I168:

In assembler:

- There is no difference in the way you declare unsigned or signed variables.
- You know how the variables will be used and you are responsible for selecting the correct conditional jumps.

Two sample sections of code for you to try to code

Convert these two short problems into 8086 assembler. All variables are already defined for you.

1. A bank customer at an Automated Teller Machine has just entered his 2 character PIN number (Personal Identification Number). The two ASCII characters that he entered are stored in the variables P1, P2. The correct PIN number for this customer is stored in the variables C1, C2. You are to determine if the PIN numbered entered matches the correct PIN number; that is does P1=C1 and P2=C2. If so then branch to the label 'MATCH'; if not branch to the label 'NOMATCH'.

```
P1    DB    ?
P2    DB    ?
C1    DB    ?
C2    DB    ?
```

2. You are to write code to help a moving company determine if there is room in a moving van to add another load of furniture. Three variables are defined for you: CAPACITY contains the capacity of the van in pounds; CURRENT contains the number of pounds of furniture already loaded in the van; NEWLOAD contains the number of pounds in the load they want to add to the van. If CURRENT + NEWLOAD is less than or equal to the CAPACITY then branch to the label 'FIT' else branch to the label 'NOFIT'. Do not change the values stored in CAPACITY, CURRENT, NEWLOAD.

```
CAPACITY    DW    ?      ;CAPACITY OF VAN IN POUNDS, UNSIGNED
CURRENT      DW    ?      ;POUNDS OF FURNITURE ALREADY ON VAN, UNSIGNED
NEWLOAD      DW    ?      ;POUNDS OF FURNITURE TO BE ADDED, UNSIGNED
```

Sample sections of code solutions

Convert these two short problems into 8086 assembler. All variables are already defined for you.

1. A bank customer at an Automated Teller Machine has just entered his 2 character PIN number (Personal Identification Number). The two ASCII characters that he entered are stored in the variables P1, P2. The correct PIN number for this customer is stored in the variables C1, C2. You are to determine if the PIN numbered entered matches the correct PIN number; that is does P1=C1 and P2=C2. If so then branch to the label 'MATCH'; if not branch to the label 'NOMATCH'.

```
P1    DB    ?
P2    DB    ?
C1    DB    ?
C2    DB    ?
```

```
mov    al,[p1]      ;get the first pin character entered
cmp    al,[c1]      ;was it correct
jne    nomatch     ;no, exit with error
mov    al,[p2]      ;yes, get the second pin character entered
cmp    al,[c2]      ;was it correct
jne    nomatch     ; no, exit with error
jmp    match        ;yes, exit without error
```

2. You are to write code to help a moving company determine if there is room in a moving van to add another load of furniture. Three variables are defined for you: CAPACITY contains the capacity of the van in pounds; CURRENT contains the number of pounds of furniture already loaded in the van; NEWLOAD contains the number of pounds in the load they want to add to the van. If CURRENT + NEWLOAD is less than or equal to the CAPACITY then branch to the label 'FIT' else branch to the label 'NOFIT'. Do not change the values stored in CAPACITY, CURRENT, NEWLOAD.

```
CAPACITY   DW    ?      ;CAPACITY OF VAN IN POUNDS, UNSIGNED
CURRENT     DW    ?      ;POUNDS OF FURNITURE ALREADY ON VAN, UNSIGNED
NEWLOAD     DW    ?      ;POUNDS OF FURNITURE TO BE ADDED, UNSIGNED
```

```
mov    ax,[current]   ;get the weight of the current load
add    ax,[newload]   ;calculate the new desired load
jc    nofit          ;an overflow definitely means desired load is too big
cmp    ax,[capacity] ;compare desired load to capacity
ja    nofit          ;if desired load is above the capacity then we have an error
jmp    fit            ;if desired load is below or equal capacity we can carry the load
```

Additional Directives

extrn Specifies that a label is defined outside this module. It is an external reference.
This example declares subr as an external procedure. extrn subr:proc

public Make a label known outside this module.
This example declares subr as available to routines in other files: public subr

Additional Opcodes

Multiplication and division (div, idiv, imul, mul) are covered in detail in the Multiply / Divide section of the notes.

String instructions (cld, std, rep, cmps, lods, movs, scas, stos) are covered in their own section of the notes.

call Subroutine call.
Push the IP register, which contains address of the next instruction, onto the stack and then jump to the target instruction.

- call does not set the condition code.

Example: call sqrt

lea Load effective address.
The lea instruction is used to do arithmetic on pointer or index registers without affecting the condition code settings. It dynamically calculates the *address* of the source operand and puts the *address* of that source operand (*not its contents*) in the destination operand.

- The source must be a memory reference. The destination must be any word register.
- lea does not set the condition code. Examples:

lea bx,[varx]; the address of varx is put into bx

The source may be *any* of the valid indirect address combinations.

The lea is used to perform mathematics on the pointer or index registers and to place the resulting *address* in the destination register.

lea di,[bx + si - 1] ; the sum of the bx+si-1 is put into di
; (if bx=1000h si=01F8h then di will be set to 11F7h)

lea si,[si + 2] ; advance the pointer value in si by 2 (si=si+2)

- Note to C++ programmers: lea does not dereference data. The result is an *address*.

loop Loop.
cx holds the count of the number of times you want to go through the loop. cx is treated as an unsigned word. cx is decremented by 1. If cx is not zero then the loop is repeated. An initial value of zero in cx will cause the loop to be repeated 65,536 times. See the jcxz instruction.

- loop does not set the condition code

Example: loop doitagain

There are two additional variations of the loop instruction ... and a special jump instruction for testing cx.

loop/equal Loop while equal (loop while zero).
cx holds the count of the number of times you want to go through the loop. cx is treated as an unsigned word. cx is decremented by 1. If cx is not zero and ZF=1 then the loop is repeated. This allows you to exit the loop if either the count goes to zero or two values compare as not equal or some arithmetic value is not zero.

loopne/notzero Loop while not equal (loop while not zero).
cx holds the count of the number of times you want to go through the loop. cx is treated as an unsigned word. cx is decremented by 1. If cx is not zero and ZF=0 then the loop is repeated. This allows you to exit the loop if either the count goes to zero or two values compare as equal or some arithmetic value is zero.

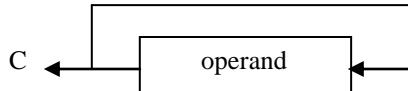
jcxz Jump if cx is zero.
This instruction will jump if the cx register is zero. This can be used to bypass a set of instructions when using the loop instruction to control the number of iterations.

- jcxz does not set the condition code

pop	<p>Pop a word item off the stack. Pop always removes a word of data.</p> <p>One word of data is removed from the stack. After the word is removed, the stack pointer is incremented by 2. The destination can be a register or variable.</p> <ul style="list-style-type: none"> • pop does not set the condition code. <p>Example: pop ax pop [varx]</p>
push	<p>Push a word item onto the stack. Push always stores a word of data.</p> <p>The stack pointer is decremented by 2, then the word size data item is placed onto the stack. The source may be a register or variable.</p> <ul style="list-style-type: none"> • push does not set the condition code. <p>Example: push ax push [varx]</p>
ret	<p>Return from subroutine.</p> <p>The return address (word size data) is popped off the stack and placed into the IP register and the program resumes execution at the instruction after the call.</p> <ul style="list-style-type: none"> • ret does not set the condition code. <p>Example: ret</p>
and	<p>Boolean <i>and</i>.</p> <p>Performs the Boolean ‘and’ of two values. The source is ‘anded’ into the destination. The result is stored in the first operand which is the destination.</p> <ul style="list-style-type: none"> • and sets the sf and zf flags based upon the result. The of and cf are cleared to zero. <p>Example: and ax,bx and [varx],0800h and si,[newval]</p>
not	<p>Boolean <i>not</i>.</p> <p>Inverts all the bits of a value.</p> <ul style="list-style-type: none"> • not does not set the condition code. <p>Examples: not ax not [varx]</p>
or	<p>Boolean <i>inclusive or</i>.</p> <p>Performs the Boolean ‘or’ of two values. The source is ‘ored’ into the destination. The result is stored in the first operand which is the destination.</p> <ul style="list-style-type: none"> • or sets the sf and zf flags based upon the result. The of and cf are cleared to zero. <p>Example: or ax,bx or [varx],0800h or si,[newval]</p>
xor	<p>Boolean <i>exclusive or</i>.</p> <p>Performs the Boolean ‘exclusive or’ of two values. The source is ‘xored’ into the destination. The result is stored in the first operand which is the destination.</p> <ul style="list-style-type: none"> • xor sets the sf and zf flags based upon the result. The of and cf are cleared to zero. <p>Example: xor ax,bx xor [varx],0800h xor si,[newval]</p>
test	<p>Boolean <i>and</i> without saving the result.</p> <p>Performs the Boolean ‘and’ of two values and updates the condition code, but does not store the result. That is, neither operand is changed. The test instruction may be followed by a <i>jnz</i>. The jump will be taken if there are any matching 1 bits in both operands.</p> <ul style="list-style-type: none"> • test sets the sf and zf flags based upon the result. The of and cf are cleared to zero. <p>Example: test ax,bx test [varx],1 test si,[newval]</p>

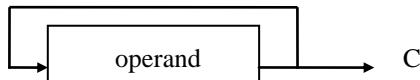
For the ROTATES and SHIFTS the only immediate value allowed is 1. For other values, the value goes in cl.

rol / rcl	Rotate left. (LSB=least significant bit MSB=most significant bit) Rotates a value left. The old MSB moves into the carry flag and becomes the new LSB. <ul style="list-style-type: none"> The source operand is an immediate value of 1 in which case the destination is rotated 1 bit. The source operand can be the cl register. In which case the rotate amount is loaded into cl before the rotate is executed. The carry flag contains the value of the last bit shifted. Example: rol ax,1 rol [varx],1 rol ax,cl (cl contains the shift count)
-----------	--



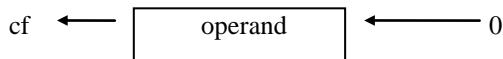
rcl is the same except the Carry Flag is treated as part of the data.
 That is, the carry flag becomes the LSB and then the MSB goes into the carry flag.

ror / rcr	Rotate right. (LSB=least significant bit MSB=most significant bit) Rotates a value right. The old LSB moves into the carry flag and becomes the new MSB. <ul style="list-style-type: none"> The source operand is an immediate value of 1 in which case the destination is rotated 1 bit. The source operand can be the cl register. In which case the rotate amount is loaded into cl before the rotate is executed. The carry flag contains the value of the last bit shifted. Example: ror ax,1 ror [varx],1 ror ax,cl (cl contains the shift count)
-----------	--

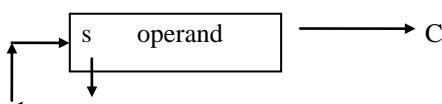


rcr is the same except the Carry Flag is treated as part of the data.
 That is, the carry flag becomes the MSB and then the LSB goes into the carry flag.

sal / shl	Shift arithmetic left / Shift logical left (these generate the same machine code). Shifts a value left. The old MSB moves into the carry flag. A zero bit replaces the old LSB. <ul style="list-style-type: none"> The source operand is an immediate value of 1 in which case the destination is shifted 1 bit. All the condition code flags are set and the 'of' indicates that the sign bit has changed value. The source operand can be the cl register. In this case the shift amount is loaded into cl before the shift is executed. In this case the sf, zf, cf flags are set but the 'of' is not defined. Example: sal ax,1 sal [varx],1 sal ax,cl (cl contains the shift count)
-----------	--



sar	Shift arithmetic right. Shifts a value right. The old LSB moves into the carry flag. The old value of the MSB (the sign bit) is copied as the new value of the MSB (sign bit). <ul style="list-style-type: none"> The source operand is an immediate value of 1 in which case the destination is shifted 1 bit. The source operand can be the cl register. In this case the shift amount is loaded into cl before the shift is executed. The shr sets the sf, zf, of, cf. (The of is undefined for multipbit shifts). Example: sar ax,1 sar [varx],1 sar ax,cl (cl contains the shift count)
-----	---



For the ROTATES and SHIFTS the only immediate value allowed is 1. For other values, the value goes in cl.

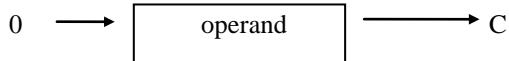
shr

Shift logical right.

Shifts a value right. The old LSB moves into the carry flag. A zero bit is copied as the new value of the MSB.

- The source operand is an immediate value of 1 in which case the destination is shifted 1 bit.
- The source operand can be the cl register. In this case the shift amount is loaded into cl before the shift is executed.
- The shr sets the sf, zf, of, cf. (The of is undefined for multipbit shifts).

Example: shr ax,1 shr [varx],1 shr ax,cl (cl contains the shift count)



xchg

Exchange

Exchanges the source and destination operands. That is, the values are swapped.

- xchg does not set the condition code.

Examples: xchg ax,bx xchg ax,[varx]

xlat

Translate-table

- The xlat instruction uses the contents of the al register as an index into a 256-byte table.
- The original byte in the al register is replaced by the byte in the table at the offset corresponding to the original value in al.
- The bx register must point to the table.
- xlat does not set the condition code

The example below reads a character from the standard input stream. It then uses xlat to translate the character. If the character read was a lower case letter then it is converted to upper case.

If the character read is anything else then it is converted to an *.

```

.data
; tran is a table with 256 entries (0 to 255)
tran db 97 dup('*') ; the 97 chars below 'a'
db 'ABCDEFGHIJKLMNPQRSTUVWXYZ' ; a-z in lowercase
db 133 dup('*') ; the 133 chars above 'z'

```

.code

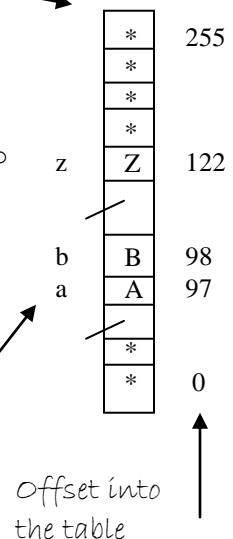
```

mov ax, @data ; set addressability
mov ds, ax ; to the data segment

mov bx, offset tran ; bx points to the table
mov ah, 8 ; code to read without echo
int 21h ; read a character
xlat ; translate the char
mov ah, 2 ; code to write char
mov dl, al ; mov character to dl
int 21h ; write the character

```

Lower case 'a' with a decimal value of 97 is replaced by the 98th entry in the table which is an upper case 'A'.



Additional Thoughts on Assembler Opcodes and Directives

This chapter contains many miscellaneous topics related to assembler.

They are listed in chapter's table of contents below.

Reserved words - do not use these as labels	2
Negative immediate source operand values are fine	2
You can use a variable as both the source and destination	2
How to put a quotation mark in an ASCII string	2
How to do multiple digit addition	3
How to do multiple digit subtraction	4
How to dynamically modify an output string	5
Coding and commenting immediate data.....	5
Using the <i>equ</i> directive to code immediate data	6
A review of how the Condition Code is set on addition and subtraction	7
The 8086 Carry Flag and how it is set (real subtraction versus two's complement addition)	9
Conditional jumps and the flags tested	11
High-level language constructs - the assembler language equivalents	13
Simple if...then...else statement.....	13
Compound if...then...else statement	13
While statement	14
Break statement to exit a while loop	15
Switch statement	16
How to use the extra segment for named variables.....	17
How to use the extra segment with pointers and indirect addressing.....	19

Reserved words - do not use these as labels

The assembler has a list of reserved words that may not be used as labels. Any attempt to use these as labels will generate weird error messages (see common bugs section). These include:

All instruction opcodes (add, adc, mov, jmp, loop, etc.)
All directives (db, dw, stack, etc.)

See the WEB for a list of the most common reserved words.

Negative immediate source operand values are fine

All of these are valid: mov ax,-1 cmp bx,-10 add [varx],-100



You can use a variable as both the source and destination

When you do, the original value is used as the source and the result is stored as the destination. For example:

```
mov al,5      ; 5          set the value in ax = ?? 05  
add al,al    ; 5+5 =10   add al to itself so that ax = ?? 0A  
mul al       ; 10*10=100  multiply al by itself so that ax = 00 64
```

How to put a quotation mark in an ASCII string

Use the double quote (") as the data delimiter.
con1 db "that's okay"

Use the single quote (') as the data delimiter
con2 db 'he said "hello" to you'

Use two single quotes together. The message will appear as: that's okay
con3 db 'that's okay'

How to do multiple digit addition

The Intel 8086 hardware only supports byte + byte or word + word additions. How can you support any unique data size? Assume that you wanted to add longword + longword (one in the ax/bx pair and one in the cx/dx pair). We will add into the cx/dx pair. You perform operations just as you would in decimal if you wanted to add multiple digit numbers. You add from low order to high order. If you get a carry from an operation you add 1 to the next higher set of digits. This is how the math looks in hex. Note bx+dx exceeds a word so a carry is required into the addition of ax+cx.

$$\begin{array}{r} 1 \\ \text{ax} \quad \text{bx} = \quad 0167 \quad \text{E348} \\ + \text{cx} \quad \text{dx} = \quad + 4055 \quad 7722 \\ \hline \text{cx} \quad \text{dx} = \quad 41BD \quad 5A6A \end{array}$$

This is the way you might try to code the above operation.

```
add    dx,bx      ; add the low words into dx, if there is a carry then cf will be set to 1
jnc    skip        ; if there was no carry the skip adding the carry
add    cx,l        ; add the carry from the previous addition into the high order sum
skip: add    cx,ax  ; add the high words in cx
```

Intel has a special add instruction called *add with carry* that should be used for adding all the columns into which a carry might occur. Using adc assures the correct propagation of the carry and reduces the number of instructions needed to perform this longword addition. The way adc works is that it adds its two operands together (just like add) but it also adds the value of the carry flag cf.

```
add    dx,bx      ; add the low word, if there is a carry then cf will be set to 1
adc    cx,ax      ; add the high word and the CF from the previous operation
```

How to do multiple digit subtraction

There is similar support for subtracting. Assume you want to subtract longword - longword (cx/dx pair minus ax/bx pair). You perform operations just as you would in decimal if you wanted to subtract multiple digit numbers. You subtract from low order to high order. If you get a borrow from an operation you subtract 1 from the next higher set of digits.

cx	dx =	1234	0000	
-	ax	bx =	- 0000	0001
-----		----	----	
cx	dx =	1233	FFFF	

From a coding point of view, you can do this two ways.

```
sub    dx,bx      ; subtract the low words, if there is a borrow then cf will be set to 1
jnc    skip        ; if there was no carry the skip subtracting the carry
sub    cx,l       ; subtract borrow from previous subtraction from the high order value
skip:   sub    cx,ax    ; subtract the high words
```

Intel has a special instruction called *subtract with borrow* that should be used for all the columns from which a borrow might occur. Using sbb assures the correct propagation of the borrow and reduces the number of instructions needed to perform the subtraction. The way sbb works is that it subtracts its two operands (just like sub) but it also subtracts the value of the carry flag cf.

```
sub    dx,bx      ; subtract the low word, if there is a borrow then cf will be set to 1
sbb    cx,ax      ; subtract the high word and the cf from the previous operation
```

How to dynamically modify an output string

Create the string.

Have a named variable within the string.

Move data into the named variable. ← *this modifies the output string*

```
databaseg ; data goes in data segment
ans      db      'The answer is ' ; part 1 of the string
val      db      ? ; part 2 of the string ... the part that will be modified
           db      13,10,'$' ; part 3 of the string

codeseg ; code goes in code segment
mov      [val],al ; al contains the answer in ASCII and this value
                  ; is stored into the middle of the output string
mov      dx, offset ans ; dx must point to the string
mov      ah,9 ; set the DOS code to write a string
int      21h ; write the string
```

Coding and commenting immediate data

When comparing a variable against an immediate ASCII value you have 3 choices. All of these compare the *al* register to the ASCII character 'A'. Even *without a comment*, the last one is easier to follow.

```
cmp al,65
cmp al,41h
cmp al,'A'
```

Now let us insert comments. The first two are not particularly useful.

```
cmp al,65      ; compare al to 65
cmp al,41h     ; compare al to test value
cmp al,'A'     ; was the character read from the keyboard an upper case letter 'A'
```

The first two of those comments do *not* really provide additional information above what can be determined from the actual instruction. The third example provides insight into what the programmer is attempting to accomplish.

Using the equ directive to code immediate data

There are many ways to code the same function.

Assume you want to compare the contents of the al register to the ASCII exclamation point '!'. Then all of these are the same. The last one is the most clear to someone reading the program.

```
cmp al,33           ; compare al to !
cmp al,21h          ; compare al to !
cmp al,'!'          ; compare al to !
```

You can also use the EQU directive. This directive equates a label to an immediate value. For example

```
exp_point equ 33      ; the word exp_point is equated to the number 33
```

You can now code our compare another way.

```
cmp al,exp_point; compare al to !
```

Finally, if you had declared the following variables...

```
ex1 db 33
ex2 db 21h
ex3 db '!'
```

... then these will also compare the al register an exclamation point.

```
cmp al,[ex1]          ; compare al to !
cmp al,[ex2]          ; compare al to !
cmp al,[ex3]          ; compare al to !
```

More uses for EQU

Going back to the use of negative immediate data, these are the same:

```
mov ax,-1            ; set ax to -1
mov ax,0ffffh         ; set ax to -1
```

Using an EQU you could also equate a word to the value -1.

```
minus_1 equ -1
mov al,minus_1
```

The EQU directive can also equate a word to a phrase. In this example we wish to change the letter X to the letter Y. This value is 7 bytes into a string.

```
msg db 'LETTER X IS RIGHT'
loc equ msg + 7

mov [msg + 7], 'Y'      ; change X to Y
mov [loc], 'Y'          ; change X to Y
```

A review of how the Condition Code is set on addition and subtraction

Addition and subtraction set all the Condition Code bits:

SF – Sign Flag
ZF – Zero Flag
OF – Overflow Flag
CF – Carry Flag

These bits are set algorithmically and it is the responsibility of the programmer to test the flags that match the way the data is being used, either as unsigned or signed,

- The Sign Flag is set to be the left most bit of the result. If the numbers are signed then this bit does correctly represent the sign of the result (0=positive, 1=negative). If the numbers are unsigned then this bit has no specific meaning, and the programmer should not be testing the Sign Flag. It is not logical to try to test the sign of an unsigned number.
- The Zero Flag is set to 1 (true) if the result is all zeroes. There is only one value of zero. An unsigned zero looks just like a two's complement zero. Thus the Zero Flag is valid for both unsigned and two's complement signed numbers.
- An overflow occurs when the result of a calculation does not fit in the data size being used. It can occur for unsigned and signed numbers when adding or subtracting. The Carry Flag is set to 1 if the operation (addition or subtraction) caused an unsigned overflow. The Overflow Flag is set to 1 if the operation (addition or subtraction) caused a signed overflow.

Since the hardware does not know if the numbers are signed or unsigned it uses one flag (the Carry Flag) to signal an unsigned overflow and another flag (the Overflow Flag) to signal a signed overflow. The hardware algorithmically sets both those flags. The programmer must test the flag that matches the type of data being used.

- Let us look at addition first. Let us assume this byte size addition is performed.

$$\begin{array}{r} 111111 \\ 10101110 \\ + 11010101 \\ \hline 10000011 \end{array}$$

If those numbers were unsigned then the top number is 174 and the bottom number is 213. The correct sum should be 387, but that value exceeds the maximum unsigned byte value of 255. So we end up with a decimal result of 131 and a carry out of the most significant bit. That carry represents the value 256. We have had an unsigned overflow that is detected by the carry out of the most significant bit and the Carry Flag will be set to 1.

If those numbers were signed then the top number is -82 and the bottom number is -43. The correct sum should be -125 and that value does fit within a signed byte whose range is -128 to +127. There are two tests for a signed overflow that will show that signed result is correct. These two tests were derived in chapter 2D-4 through 2D-7. Test 1 shows that we added two negative numbers and obtained a negative result which means the result is correct. Test 2 shows that the carry into the sign bit equals the carry out of the sign bit (both are 1) which shows the result is correct. Thus the Overflow Flag will be set to 0.

For this example there was an unsigned overflow but not a signed overflow.

- Now let us look at subtraction. Let us assume this byte size subtraction is performed.

If the numbers are unsigned we can do pure subtraction.

```

10000000
- 00000001
-----
01111111

```

If those numbers were unsigned then the top number is 128 and the bottom number is 1. The correct difference should be $128 - 1 = 127$, and that value is allowed in an unsigned byte. The value of the result is binary 01111111 which as an unsigned number with the decimal value of 127. No borrow was required into the most significant bit and thus we did not have an unsigned overflow and the Carry Flag will be set to 0.

If those numbers were signed then we would perform the subtraction $(A) - (B)$ as $A + (-B)$

```

1
10000000  ← -128
+ 11111110  { -1
   1
-----
01111111

```

The top number is -128 and the bits flipped bottom number is -1. The correct sum should be $(-128) + (-1) = -129$ but that value does not fit within a signed byte whose range is -128 to + 127. The result 01111111 is +127 and not -129.

There are two tests for a signed overflow that will show that the signed result is incorrect. These two tests were derived in chapter 2D-4 through 2D-7. Test 1 shows that we added a negative number and a negative number and the result was a positive number which signals a signed overflow. Test 2 shows that the carry into the sign bit was 0 and the carry out of the sign bit was 1 and since they do not match the result is incorrect. The Overflow Flag will be set to 1.

For this example there was a signed overflow but not an unsigned overflow.

- Here is an example where there is neither a signed nor unsigned overflow.
- Here is an example where there is both a signed and unsigned overflow.

```

00000001
+ 00000001
-----
00000010

```

SF=0 ZF=0 OF=0 CF=0

```

1
10000000
+ 10000000
-----
00000000

```

SF=0 ZF=1 OF=1 CF=1

- In summary:

The Carry Flag is set to 1 if the data is treated as unsigned and an unsigned overflow occurred.

The Overflow Flag is set to 1 if the data is treated as signed and a signed overflow occurred.

The flags are set algorithmically and the programmer must test the flag that matches the way the data is being used.

The 8086 Carry Flag and how it is set (real subtraction versus two's complement addition)

Important note.

Even if you skip reading this explanation, make sure you know the 8086 Carry Flag Rule presented at the end of this discussion.

Let us look at the following 3 bit binary example:

Case #1 done using pure subtraction

$$\begin{array}{rcl} 010_2 & = & +2_{10} \\ - 011_2 & = & - 3_{10} \\ \hline 111_2 & = & -1_{10} \end{array}$$

Look at how we *THINK* the condition code should be set.

SF=1

left most bit is 1

OF=0

positive - positive is the same as positive + negative so no signed overflow is possible

ZF=0

the result is not zero

CF=1

a borrow into the most significant digit was required

Case # 2 repeat the process but this time using two's complement. This means $A-B=A+(-B)$

$B = +3_{10} = 011_2$ so $-B = \text{flip the bits } +1 = 100 + 1 = 101_2$

$$\begin{array}{rcl} 010 & \xrightarrow{\hspace{1cm}} & 010 & = & +2 \\ - 011 & & +101 & = & + -3 \\ \hline 111 & & 111 & = & -1 \end{array}$$

Look at how we *THINK* the condition code should be set.

SF=1

left most bit is 1

OF=0

carry into sign = carry out of sign so no signed overflow occurred

ZF=0

the result is not zero

CF=?

*a carry out of the most significant digit did NOT occur
should we set CF=0 ??????*

NO ... and we will now see why CF should be set to 1

Note the numeric result was identical. We expect that. If it were not then two's complement would not be useful.
Also, the condition code flags seem to be the same except the carry flag. Let us see why.

First we review the definition of the carry flag.

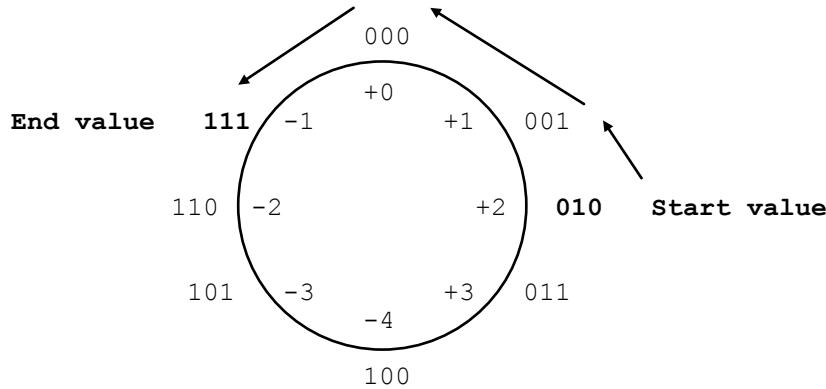
The cf indicates whether an unsigned overflow occurred and is set : 1=unsigned overflow 0=no unsigned overflow.

Another way to look at this is the carry flag is set to 1 if there is a carry out of the most significant bit on an addition operation or if there is a borrow required into the most significant bit on a subtract operation.

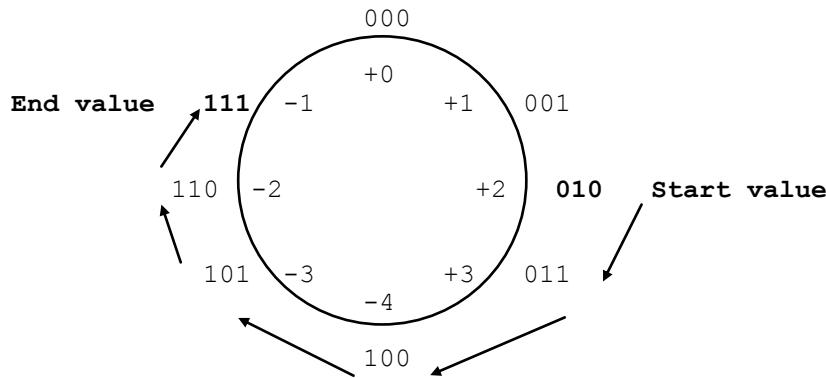
So clearly Case #1 is correct.

We now need to explain why the cf appears incorrect in Case #2 and what the 8086 does in this situation.

Case #1. In the pure subtraction case, we went from $+2_{10}$ (010_2) to -1_{10} (111_2) by subtracting the value 1 multiple times: $010_2 - 011_2 = 111_2$



Case #2. In the two's complement case, we went from $+2_{10}$ (010_2) to -1_{10} (111_2) by adding the value 1 multiple times: $010_2 + 101_2 = 111_2$



In both cases we start at the same value and end at the same value. However, we travel in *different directions*.

In the pure subtraction we cross the zero boundary causing a borrow.

In two's complement addition we do not cross the zero boundary and there is no carry.

This difference will always be the case. You can only cross the zero boundary in one direction. This means, when using two's complement addition to perform real subtraction, the carry is always the *reverse* of the real borrow value.

The 8086 Carry Flag Rule You need to know this rule.

**The 8086, on subtraction, always sets the cf bit to represent the real borrow value.
If you use two's complement addition to perform real subtraction then you must remember that
the 8086 cf flag will be the inverse of the carry you calculate.**

Going back to Case #2 where the carry out of the left most bit was 0, the 8086 will invert that 0 and set the CF to 1.

Conditional jumps and the flags tested

Each conditional jump decides whether to execute the jump or fall through to the next sequential instruction. The decision is based upon the settings of the flags in the condition code. Remember that a compare calculates the destination - source. The jump looks at the result to determine whether to jump.

dest-source	conclusion
< 0	dest < source
= 0	dest = source
> 0	dest > source

Example 1. The unsigned jump above or equal (jae)

Given the following code:

cmp [var],al	; [varx] - al
jae higher	; jump if result ≥ 0

The jae will jump if the carry flag = 0. Why this flag? The cmp instruction calculates [var] - al. If [var] \geq al then the result ≥ 0 and a borrow is not needed and so the cf = 0. So this single test assures that the jump is taken only when [var] \geq al.

Example 2. A more challenging case to understand, the signed jump greater or equal (jge)

Given the following code:

cmp [var],al	; [varx] - al
jge smaller	; jump if result ≥ 0

We might expect that the jge will jump if sign flag = 0 indicating a positive result. However, the notes show the jge will jump if the sign flag equals the overflow flag (sf = of). Why? Again, the cmp instruction calculates [var] - al. We expect the sf to tell us the relative magnitudes of the source and destination. For example, if the destination is more than the source then we expect the sign of the result of (dest - source) to be positive (e.g. $+15 - +7 = +8$). The complication is that *if an overflow occurs on the subtraction then the sign of the result will be wrong*, and we must account for that possibility.

Assume that the calculation [var] - al does not cause an overflow. Then of = 0. If [var] \geq al then the result will be positive and the sf = 0. Note that in this situation, with [var] \geq al and no overflow on the comparison, then sf = of = 0 and the test works.

Assume that the calculation [var] - al does cause an overflow. Then of = 1 and the sign of the result will be wrong. So if [var] > al then the calculated result will appear negative and the sf = 1. This is hard to picture so let me show you some real numbers.

Assume var is $+127_{10} = 0111111_2$ and al is $-2_{10} = 1111110_2$.
We expect that $(+127) - (-2) = +129$ however +129 will overflow a signed byte.

$$\begin{array}{r} \text{[var]} - \text{al} = \quad 0111111 \\ - \quad 1111110 \\ \hline 10000001 = -127 \text{ NOT } +129 \end{array}$$

In this situation, with [var] > al and an overflow on the compare, then sf = 1 and of = 1 and the test works.

{Extra aside note. In the case where an overflow occurs then [var] cannot equal al. If [var] = al then the result will be zero without an overflow.}

Example 3.

Given the following value of the condition code: sf=0 zf=0 of=1 cf=1
Which of the following conditional jumps will be taken?

- jb
- jo
- jl

To answer this type question, you always need to get the table that shows the flags tested for conditional jump.

- The table shows that *jb* will jump if the carry flag is 1. It is, so *jb* will be taken.
- The table shows that *jo* will jump if the overflow flag is 1. It is, so *jo* will be taken.
- The table shows that *jl* will jump if the sign flag \neq overflow flag. That is true, so *jl* will be taken.

The last answer is an area that can cause confusion, so let's explain the answer.

- First, assume the condition code was set by an arithmetic operation such as add or subtract.

Then *jl* (jump less than) which is a signed test seems to imply a result less than zero which would imply the sign flag would be 1. However, the sign flag is 0, so why did we jump?

The answer is there was a signed overflow as indicated by of=1 and the rule is that *the arithmetic tests only have a valid meaning when there was not an overflow*. So after an arithmetic operation that caused an overflow, we should not have used this test. We made a bad test and we took a bad jump. Programmer error!

- Second, assume the condition code was set by a *compare* instruction.

The compare instruction calculates (destination - source) and then sets the condition code.

Then *jl* (jump less than) which is a signed test seems to imply the destination is less than the source which would then imply the sign flag would be 1. However, the sign flag is 0, so why did we jump?

The answer is that the *jl* test is sf \neq of. This is more than just testing the sign flag and the reason it is needed is that we must handle the situation of having an overflow when subtracting the source from the destination.

- If the destination were less than the source and there had not been an overflow when calculating (destination – source) then the overflow flag would be 0 and sign flag would be 1. This is exactly what we expect; the destination is less than the source and sf is 1 and the *jl* is taken because sf \neq of (sf=1 and of=0).
- If the destination were less than the source and there was an overflow when calculating (destination – source) then we know the sign of the result will be wrong (that is the definition of a signed overflow). So in the presence of an overflow, although the sign flag is 0, if we had used infinite precision arithmetic then the correct result would have had the sign flag as 1 which shows the destination is less than the source and the *jl* is taken because sf \neq of (sf=0 and of=1).

Thus the *jl* test of sf \neq of always works after a compare and jumps if the destination is less than the source, whether or not an overflow occurs in performing that calculation.

High-level language constructs - the assembler language equivalents

Here are examples of high-level language control statements and how they can be translated into assembly language. Note that we use the underscore (_) on our labels to assure that we do not use a Turbo reserved word as a label.

Simple if...then...else statement

```
;if (ax==0) then (cx=cx+1) else (dx=dx+1)

if_:
    cmp     ax,0           ; test the condition
    jne     else_
then_:
    inc     cx             ; cx=cx+1
    jmp     endif_
else_:
    inc     dx             ; dx=dx+1
endif_:
```

Compound if...then...else statement

```
; if (ax==0 and bx==0) then (cx=cx+1) else (dx=dx+1)

if_:
    cmp     ax,0           ; test the compound condition
    jne     else_
    cmp     bx,0           ; - is bx == 0
    jne     else_
then_:
    inc     cx             ; cx=cx+1
    jmp     endif_
else_:
    inc     dx             ; dx=dx+1
endif_:
```

Important Note

You can always break compound statements in multiple simple statements. This may make it easier for you to see how generate the code.

Compound: if (a==b or c==d) then do {...} else do {...}

Simple: if (a==b) then do {...}
else if (c==d) then do {...}
else do {...}

For statement

We give it in generic pseudo code and in C++.
Note the ending condition is tested before the first iteration.

```
; for x=1 to 10 do {y = y + 1}      (pseudo code)
; for (x=1; x≤10; x++) {y = y + 1}  (C++ x and y declared as INT)

x      dw      0          ; x is a signed word
y      dw      0          ; y is a signed word

for_:
    mov    [x],1      ; initialize the loop counter
    jmp    test_
do_:
    inc    [y]       ; if loop not done then y = y + 1
next_:
    inc    [x]       ; increment the loop counter
test_:
    cmp    [x],10    ; test counter against end value
    jle    do_        ; repeat if counter less than end val
endfor_:

```

While statement

A *while* statement tests the condition before every iteration,
even the first one.

```
;set initial value in char to zero
;while (char≠lah) {read character}

while_:
    mov    [char],0      ; set initial value in char to 0
    jmp    test_
repeat_:
    mov    ah,8          ; set dos code to read without echo
    int    21h           ; read a character
    mov    [char],al      ; save the character
test_:
    cmp    [char],lah    ; is char the termination value
    jne    repeat_      ; no, repeat the loop
    endwhile_:

```

Break statement to exit a while loop

This is similar to a PASCAL *repeat* loop.
The loop is always executed at least once.

```
; count = 0
; while (true)
; {
;   char = read character
;   if char == eof then break
;   count = count + 1
; }

while_:
    mov    [count],0      ; set count of chars read to zero
    ;
    mov    ah,8           ; set dos code to read without echo
    int    21h            ; read a character
    mov    [char],al        ; save the character
    cmp    [char],lah       ; is char the termination value
    je     endwhile_       ; yes, break out of the loop
    inc    [count]          ; no, increment count of chars read
    jmp    while_          ; repeat the loop
endwhile_:
    ; exit
```

Switch statement

This is the actual assembler code generated by a compiler.

```
switch (x)          // C switch statement
{
    case 100:        // This case statement does not
                     // have a break.
        one++;
    case 10:         // This case statement does
                     // have a break.
        small++;
        break;
    case 'A':        // This case statement does
                     // have a break.
        let++;
        break;
    case 275:        // This case statement does
                     // have a break.
        big++;
        break;
    default:         // This case statement is executed
                     // if x does not match any of the above.
}
```

```
switch (x)          mov     ax,WORD PTR [x]      ;get x
                     jmp     $S171           ;go to test x
case 100:          $SC175:   inc     WORD PTR [one]    ;one++
case 10:           $SC176:   inc     WORD PTR [small]  ;small++
                     jmp     $SB172           ;break
case 'A':          $SC177:   inc     WORD PTR [let]    ;let++
                     jmp     $SB172           ;break
case 275:          $SC178:   inc     WORD PTR [big]    ;big++
                     jmp     $SB172           ;break
default:          $SD179:   inc     WORD PTR [err]    ;err++
                     jmp     $SB172           ;automatic break
$S171:             sub     ax,10            ;test x
                     je      $SC176           ;if 0 then case 10
                     sub     ax,55            ;x=x-55
                     je      $SC177           ;if 0 then case 65='A'
                     sub     ax,35            ;x=x-35
                     je      $SC175           ;if 0 then case 100
                     sub     ax,175           ;x=x-175
                     je      $SC178           ;if 0 then case 275
                     jmp     $SD179           ;else default
$SB172:            jmp     $S171           ;exit switch statement
```

How to use the extra segment for named variables

It is very easy to use the extra segment for an additional 64 Kbytes of data. We need to do three things:

1. Define the extra segment
2. Correctly load the es register
3. Tell the assembler that we are using the es register to access the extra segment.

This code shows how to accomplish these three steps for *named variables*. Look for the ***bold italic*** statements.

```
;-----  
; This code shows how to use the extra segment in TASM  
;  
    ideal                      ;use turbo ideal mode  
    model      small           ;64k code and 64k data  
    p8086                  ;only allow 8086 instructions  
    stack      256            ;reserve 256 bytes for the stack  
;  
        dataseg                ;start the data segment  
;  
v1    dw      1              ;three  
v2    dw      2              ; variables  
v3    dw      3              ; in the data segment  
;  
        fardata               ;start the extra segment  
;  
v4    dw      4              ;three  
v5    dw      5              ; variables  
v6    dw      6              ; in the extra segment  
;  
        codeseg                ;start the code segment  
;  
start:  
    mov      ax,@data          ;establish addressability to the  
    mov      ds,ax             ;data segment for this program  
    mov      ax,@fardata       ;establish addressability to the  
    mov      es,ax            ;extra segment for this program  
    assume   es:@fardata       ;tell the assembler that we are  
                            ; using es for the extra segment  
;  
; perform the calculation  
;  
    mov      ax,[v1]           ;add the  
    add      ax,[v2]           ; three variables  
    add      ax,[v3]           ; from the data segment  
    add      ax,[v4]           ;add the  
    add      ax,[v5]           ; three variables  
    add      ax,[v6]           ; from the extra segment  
;  
; terminate program execution  
;  
exit:  
    mov      ax,4c00h          ;set dos code to terminate program  
    int      21h               ;return to dos  
    end      start             ;end of the source code  
;
```

```

;-----  

; This code shows how to use the extra segment in MASM  

;-----  

.model small ;64k code and 64k data  

.8086 ;only allow 8086 instructions  

.stack 256 ;reserve 256 bytes for the stack  

;  

;-----  

.data ;start the data segment  

;-----  

v1 dw 1 ;three  

v2 dw 2 ; variables  

v3 dw 3 ; in the data segment  

;  

;-----  

.fardata ;start the extra segment  

;-----  

v4 dw 4 ;three  

v5 dw 5 ; variables  

v6 dw 6 ; in the extra segment  

;  

;-----  

.code ;start the code segment  

;-----  

start: ;  

    mov ax,@data ;establish addressability to the  

    mov ds,ax ;data segment for this program  

    mov ax,@fardata ;establish addressability to the  

    mov es,ax ;extra segment for this program  

    assume es:@fardata ;tell the assembler that we are  

                       ; using es for the extra segment  

;-----  

; perform the calculation  

;-----  

    mov ax,[v1] ;add the  

    add ax,[v2] ; three variables  

    add ax,[v3] ; from the data segment  

    add ax,[v4] ;add the  

    add ax,[v5] ; three variables  

    add ax,[v6] ; from the extra segment  

;-----  

; terminate program execution  

;-----  

exit: ;  

    mov ax,4c00h ;set dos code to terminate program  

    int 21h ;return to dos  

    end start ;end of the source code  

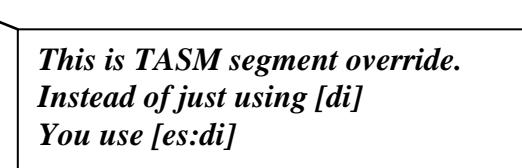
;-----  


```

How to use the extra segment with pointers and indirect addressing

When using **named variables**, the assembler knows in which segment the data was declared. When using **pointers** (indirect addressing) the system does not know in which segment the data is located. The default segment for data is the data segment. If you want to access data in the extra segment you must tell the assembler by using a **segment override**.

```
;-----  
; TASM indirect addressing in the extra segment  
;-----  
ideal          ;use turbo ideal mode  
model         small    ;64k code and 64k data  
p8086        ;only allow 8086 instructions  
stack         256     ;reserve 256 bytes for the stack  
;  
  
        database      ;start the data segment  
;-----  
listd dw 1,2,3,4,5 ;list in the data segment  
;  
  
        fardata       ;start the extra segment  
;-----  
liste dw 0,0,0,0,0 ;list in the extra segment  
;  
  
        codeseg      ;start the code segment  
;-----  
start:  
        ;  
        mov ax,@data   ;establish addressability to the  
        mov ds,ax      ;data segment for this program  
        mov ax,@fardata ;establish addressability to the  
        mov es,ax      ;extra segment for this program  
        assume es:@fardata ;tell the assembler that we are  
                           ; using es for the extra segment  
;  
; perform the calculation  
;  
        mov si, offset listd ;si points to list in data segment  
        mov di, offset liste ;di points to list in extra segment  
        mov cx,5           ;5 items to move  
;  
moveit:  
        mov ax,[si]      ;get next item from data segment  
        mov [es:di],ax  ;put next item in extra segment  
        add si,2         ;advance data segment pointer  
        add di,2         ;advance extra segment pointer  
        loop moveit     ;loop 5 times  
;  
; terminate program execution  
;  
exit:  
        mov ax,4c00h    ;  
        int 21h         ;  
        end start       ;
```



This is TASM segment override.
Instead of just using [di]
You use [es:di]

```

;-----  

; MASM indirect addressing in the extra segment  

;-----  

.model small ;64k code and 64k data  

.8086 ;only allow 8086 instructions  

.stack 256 ;reserve 256 bytes for the stack  

;  

;  

.data ;start the data segment  

;-----  

listd dw 1,2,3,4,5 ;list in the data segment  

;  

;  

.fardata ;start the extra segment  

;-----  

liste dw 0,0,0,0,0 ;list in the extra segment  

;  

;  

.code ;start the code segment  

;-----  

start:  

    mov ax,@data ;establish addressability to the  

    mov ds,ax ;data segment for this program  

    mov ax,@fardata ;establish addressability to the  

    mov es,ax ;extra segment for this program  

    assume es:@fardata ;tell the assembler that we are  

                       ; using es for the extra segment  

;  

; perform the calculation  

;  

    mov si, offset listd ;si points to list in data segment  

    mov di, offset liste ;di points to list in extra segment  

    mov cx,5 ;5 items to move  

moveit:  

    mov ax,[si] ;get next item from data segment  

    mov es:[di],ax ;put next item in extra segment  

    add si,2 ;advance data segment pointer  

    add di,2 ;advance extra segment pointer  

    loop moveit ;loop 5 times  

;  

; terminate program execution  

;  

exit:  

    mov ax,4c00h ;  

    int 21h ;  

    end start ;  

;-----
```

This is MASM segment override.
Instead of just using [di]
You use es:[di]

FILE Input/Output

*This chapter covers generic DOS functions.
Not all of these functions work in all versions of DOS or DOSBox.
Check with the instructor before using a function in your program.*

The only simple basic hardware input and output capabilities that exist are the capabilities to read a single ASCII character from keyboard and to write a single ASCII character to the display. Anything else you have done in a high level language, with respect to getting data in or out of the system, has required significant additional software that was provided by that high level language.

Single ASCII digits

You can read in a single ASCII digit by pressing one of the keys ‘0’ to ‘9’. The character that is read will come into the system with the hex value of the ASCII key pressed. The ‘0’ key will appear as 30_{16} and the ‘1’ key will appear as 31_{16} and this continues up through the ‘9’ key which will appear as 39_{16} .

To use this single ASCII digit in a numerical calculation it must be converted from ASCII to positional number system form. For a single ASCII digit, this is done by subtracting 30_{16} from the ASCII value. For example, to read in a single ASCII digit ‘4’ and double it and write it out, the following conversions are required.

- The ASCII digit ‘4’ is read as 34_{16}
- It is converted to positional form by subtracting 30_{16} yielding 04_{16}
- It is doubled yielding 08_{16}
- To write out the ASCII result, it is converted from positional form to ASCII by adding 30_{16} yielding 38_{16}

Multiple ASCII digits

This gets *much more complicated* if you want to work with multiple digit numbers. For example, in C++

- The conversion of multiple digit ASCII numbers from a string into positional format is done for you in the function *cin* which implements an algorithm called Horner’s rule.
- The conversion of positional form numbers to ASCII strings is done in the function *cout*, which implements the reduction of powers algorithm.

Both these algorithms are described in this section.

Here is the basic C++ code that declares an integer variable, reads a number, doubles it and writes out the result.

```
int value;           // declare a variable
cin >> value;       // read a number
value = value + value; // double the number
cout << value;       // output the result
```

If you run this code and type in the number 165 then these are the conversions done by the C++ routines *cin* and *cout*.

Three ASCII characters are generated when you type 165	$31_{16} \ 36_{16} \ 35_{16}$
Cin converts that string into a hex two’s complement word with the decimal value of 165	$00A5_{16}$
Doubling the value creates the hex two’s complement word with the decimal value of 330	$014A_{16}$
Cout converts the hex two’s complement word back into ASCII for printing	$33_{16} \ 33_{16} \ 30_{16}$

The only hardware support for input is the ability to read the three individual ASCII characters $31_{16} \ 36_{16} \ 35_{16}$.

The only hardware support for output is the ability to write the three individual ASCII characters $33_{16} \ 33_{16} \ 30_{16}$.

The conversion from $31_{16} \ 36_{16} \ 35_{16}$ to $00A5_{16}$ was done in software.

The conversion from $014A_{16}$ to $33_{16} \ 33_{16} \ 30_{16}$ was done in software.

DOS int 21h

DOS provides a set of function calls for these basic I/O functions.

1. Read one ASCII character from the *standard input stream*
 2. Write one ASCII character to the *standard output stream*
 3. Write an ASCII string of characters to the *standard output stream*

The *standard input stream* can either be the keyboard or an ASCII file.
The *standard output stream* can either be the display or an ASCII file.

How to select the Standard Input and Output Devices

How you start the program determines if the standard input comes from the keyboard or a file.

Without any command line parameters, these are the defaults.

- standard input is the keyboard
 - standard output is the display

If you start the program by typing its name then you will read from the keyboard and write to the display.

myprog (data is read from the keyboard, data is written to the display)

If you want to read from an ASCII text file, you use the *exact same* assembler instructions. You just redirect the standard input stream to point to a file. This is done by issuing the following DOS command when starting the program.

myprog < mydata (data is read from the file named mydata, data is written to the display)

To write to an ASCII text file instead of the display, redirect the standard output stream to the file by using this DOS command when starting the program.

myprog > myoutput (data is read from the keyboard, data is written to the file named myoutput)

To both read from an ASCII text file and write to an ASCII text file, redirect both the standard input and standard output stream with this DOS command.

```
myprog < mydata > myoutput
```

The basic input / output functions

You must use the exact same registers as shown in these examples.

You do not have a choice in register selection.

- Read one ASCII character from the standard input and echo (write) the character to the standard output.
If the standard input is the keyboard then this code will wait until the user types in a key.

```
mov ah,1      ; set ah=1 to request a character with echo
int 21h       ; call DOS and the character will be returned in al
                ; the character read and echoed is now in the al register
```

- Read one ASCII character without echo (write).
If the standard input is the keyboard then this code will wait until the user types in a key.

```
mov ah,8      ; set ah=8 to request a character without echo
int 21h       ; call DOS and the character will be returned in al
                ; the character read is now in the al register
```

- Write a single character to the standard output stream.
The character must be in the dl register. This example loads dl from a variable named char that has been defined as a byte and contains the character we wish to write.

```
mov ah,2      ; set ah=2 to request a character to be written
mov dl,[char] ; put the character in dl ... this is just an example
                ; ... dl can be loaded from any source
int 21h       ; call DOS and the character will be written
```

- Write an ASCII string to the standard output stream.
The string must be defined in the data segment. It **must** be terminated by an ASCII \$. DOS writes characters **until it finds a \$**. If you forget the \$ then DOS will keep writing bytes from the data segment, possibly forever.

```
msg db      'this is a message','$'      ; an output message
      mov ah,9      ; set ah=9 to request DOS to display a string
      mov dx, offset msg ; point dx to the string ... (dx and not dl)
      int 21h       ; call DOS to write the string
```

A DOS end of line (newline or \n for C++ programmers) requires two characters in a specific order.
The Carriage Return (CR = 13₁₀ = 0D₁₆) and the Line Feed (LF = 10₁₀ = 0A₁₆)

The following example includes the standard DOS end of line characters, carriage return and line feed. This string can be redirected to a file just like individual characters.

```
msg db  'this is a message',13,10,'$' ; message terminated with CR/LF
```

- '**Read with echo**' versus '**Read**'

When processing a file, it is sometimes necessary to use read without echo then separately use a write instruction. This allows the program to examine and process each character before that character is written to the output file.

Examples include:

- Reading a password that you do not want to show on the display
- You need to produce output **after** the input file's end of file character is read

Important note about using int 21h !!!!

This quote comes from the DOS Technical Reference Manual.

"The contents of the AX register can be altered by any of the int 21h function calls."

This means that the int 21h does not change any of the register's contents except AX.

The contents of AX may be altered by the int 21h. Do not assume AX holds its value across int 21h use.

This is especially a problem when running under Windows NT. Ignoring this warning leads to this bug, affectionately called the **NT Bug**.

```
mov ah,8          ; set up to read a character without echo
int 21h          ; read the character

mov dl,al         ; move the character to dl so it can be written
mov ah,2          ; set up to write a character
int 21h          ; write the character

cmp al,0dh      ; was the character the ENTER key
je done           ; if yes, exit
```

The above code is WRONG !!!! The contents of the al register cannot be tested. It may no longer hold the character. The correct instruction is:

```
cmp dl,0dh      ; was the character the ENTER key
```

What if I want to test to see if a character is available from the standard input stream.

When reading from the keyboard it is perfectly okay to issue a READ request and then wait for the user to press a key.

However, if you are reading from a file that does not have an End Of File (EOF) character, which signals there is no more data available, then you need to determine if there are more characters to read. If you try to read from a file and there is not any data then you will hang your program.

There is a DOS function call that will tell you if there is any more data available from the standard input stream.

Set ah=0Bh then issue the int 21h instruction. If DOS sets al=ffh then there is more data to read

```
; test for more data in the standard input stream
mov ah,0Bh        ; check if a character is available
int 21h          ; how: set ah=0bh and issue int 21h
cmp al,0ffh       ; if al = -1(dec) = ffh then a char is available to be read
jne exit          ; if none available then exit

; data is available, so read it
mov ah,1          ; if char available then read and echo it
int 21h          ; how: set ah=1 and issue int 21h

; there is no more data
exit:
```

Summary of File I/O operations

**The use of these exact registers is required.
You do not have a choice in register selection.**

Operation	Setup	Result	Example Code
Read a character without echo.	ah=8	al=char read	mov ah,8 int 21h
Write a character. The character must be in the dl register. In this example, <i>char</i> is declared as a byte and contains the character to be written	ah=2 dl=char	char is written to the standard output stream	mov ah,2 mov dl,[char] int 21h
Write a string . msg db 'xxx',13,10,'\$' The string must be terminated with a \$	ah=9 dx=offset of the string	string is written to the standard output stream	mov ah,9 mov dx,offset msg int 21h

*For CSC236 programs you should only use
Read without echo AH=8, Write a character AH=2, Write a string AH=9
The other functions will not work under DOSBox*

DOS text file format and text editors

DOS ASCII text files should have this format.

- All lines are terminated by the two characters Carriage Return (CR=0Dh) and Line Feed (LF=0Ah).
- The file is terminated by the End of File character (EOF=1Ah).

For example if you create the following two line file:

AaZz
09

That file should have this format:

A	a	Z	z	CR	LF	0	9	CR	LF	EOF
41	61	5A	7A	0D	0A	30	39	0D	0A	1A

To see exactly what your test files look like, run the program *testfile*.

The source is in the *samples* course locker.

The *testfile* program will display all the characters in a file, including those that normally do not print such as cr, lf, eof.

```

;-----  

; Program: Hello (TASM version)  

;  

; Function: Writes "Hello World" to the standard output device.  

;           It is equivalent to the famous C++ program, hello world.  

;  

;           It uses 3 constant declarations to generate the message,  

;           to show that dos writes data until it finds a '$'.  

;  

;           A single constant will also work, such as  

;           msg db 'Hello World',13,10,'$'  

;  

;           In assembler you have as much flexibility as you want.  

;  

; Owner: DAL  

;  

; Date: Changes  

; 01/26/01 original version  

;  

;-----  

ideal          ;use turbo ideal mode  

model small    ;64k code and 64k data  

p8086         ;only allow 8086 instructions  

stack 256     ;reserve 256 bytes for the stack  

;-----  

;  

;-----  

        database          ;start the data segment  

;-----  

msg    db      'Hello World'      ;the hello world message  

      db      13,10          ;cr/lf = \n in c++  

      db      '$'            ;dos end of string  

;-----  

;  

;-----  

        codeseg          ;start the code segment  

;-----  

hello:  

      mov    ax,@data        ;establish addressability to the  

      mov    ds,ax          ;data segment for this program  

;-----  

; write out the message  

;-----  

      mov    dx,offset msg  ;point to the message  

      mov    ah,9             ;set the dos code to write a string  

      int    21h              ;write the string  

;-----  

; terminate program execution  

;-----  

exit:  

      mov    ax,4c00h        ;  

      int    21h              ;set dos code to terminate program  

      end    hello          ;return to dos  

                           ;end marks the end of the source code  

                           ;.....and specifies where you want the  

                           ;.....program to start execution  

;-----
```

```

;-----  

; Program: Hello (MASM version)  

;  

; Function: Writes "Hello World" to the standard output device.  

;           It is equivalent to the famous C++ program, hello world.  

;  

;           It uses 3 constant declarations to generate the message,  

;           to show that dos writes data until it finds a '$'.  

;  

;           A single constant will also work, such as  

;           msg db 'Hello World',13,10,'$'  

;  

;           In assembler you have as much flexibility as you want.  

;  

; Owner: DAL  

;  

; Date: Changes  

; 01/26/01 original version  

;  

;-----  

.model      small          ;64k code and 64k data  

.8086        ;only allow 8086 instructions  

.stack      256            ;reserve 256 bytes for the stack  

;-----  

;  

;-----  

.data          ;start the data segment  

;-----  

msg    db      'Hello World'   ;the hello world message  

      db      13,10          ;cr/lf = \n in c++  

      db      '$'             ;dos end of string  

;-----  

;  

;-----  

.code          ;start the code segment  

;-----  

hello:          ;  

    mov      ax,@data        ;establish addressability to the  

    mov      ds,ax           ;data segment for this program  

;-----  

; write out the message  

;-----  

    mov      dx,offset msg  ;point to the message  

    mov      ah,9             ;set the dos code to write a string  

    int      21h              ;write the string  

;-----  

; terminate program execution  

;-----  

exit:          ;  

    mov      ax,4c00h        ;set dos code to terminate program  

    int      21h              ;return to dos  

    end      hello           ;end marks the end of the source code  

                           ;....and specifies where you want the  

                           ;....program to start execution  

;-----
```

```

;-----  

; Program:      COPYFILE  

;  

; Function:      This program will read characters from the standard input  

;                device.  This can be the keyboard or a file that has been  

;                redirected to the standard input.  

;  

;                It allows the software to examine each character.  

;  

;                The character is then written to the standard output device.  

;                This can be the display or a file that has been redirected  

;                to the standard output.  

;  

;                The program terminates when the character that has been  

;                read and echoed matches the character stored in the  

;                variable named 'end_char'.  

;  

;                If reading from the keyboard then  

;                set 'end_char' to be 0Dh (enter key).  

;  

;                If reading from a file then  

;                set 'end_char' to be 1Ah (DOS eof character).  

;  

;                These are the four possible combinations  

;                for reading and writing data:  

;-----  

;        copyfile           input=keyboard   output=display  

;        copyfile < input    input=file       output=display  

;        copyfile > output   input=keyboard   output=file  

;        copyfile < input > output  input=file   output=file  

;  

;        Restriction: The program hangs if you copy a file without the  

;                      an EOF character.  

;  

;        Owner:          Dana Lasher  

;  

;        Date           Reason  

;-----  

;        07/18/2000     New version  

;  

;-----  

;        ideal           ; use turbo ideal mode  

;        model          small           ; 64k data and 64k code model  

;        p8086          ; only allow 8086 instructions  

;        stack          256            ; stack size is 256 bytes
;-----
```

```

        dataseg
;-----
; The program ends when the last character
; read and echoed matches this variable.
; 1Ah=EOF    0Dh=ENTER
;-----
end_char db      0Dh          ; termination character
;-----


        codeseg
;-----
; Establish addressability to the data segment.
;-----
start:           ;
    mov      ax,@data       ; establish addressability
    mov      ds,ax           ; for the data segment
;-----
; Read a character without echo.
;-----
getloop:         ;
    mov      ah,8            ; read without echo a character
    int      21h             ; how: set ah=8 and issue int 21
;-----
; The character is now available in the
; al register for processing.
;-----

; The program may work on the data in al

;-----
; Write out the character and loop if it
; is not the terminating character.
;-----
    mov      dl,al           ; move the char to dl
    mov      ah,2            ; write the character from dl
    int      21h             ; how: set ah=2 and issue int 21
    ;
    cmp      dl,[end_char]   ; is this the terminating character
    jne      getloop         ; no, then loop
;-----
; When the terminating character has
; been read and echoed, return to DOS.
;-----
exit:           ;
    mov      ax,4C00h        ; we processed the terminating character
    int      21h             ; set correct exit code in ax
    end      start           ; int 21 will terminate program
                           ; execution begins at the label start
;-----
```

```

;-----  

;  

; Program:    SELECT  

;  

; Function:   Select prompts the user for a number (1 or 2) or q to quit.  

;             It reads the key pressed and writes an appropriate message,  

;             indicating which key was pressed.  

;  

;             The program shows some basic techniques needed to do  

;             I/O from the standard input stream.  

;  

; Owner:      DAL  

;  

; Date        Reason  

; -----  

; 01/28/2001 New version  

;  

;-----  

        ideal          ; use turbo ideal mode  

        model         small       ; 64k data and 64k code model  

        p8086         ; only allow 8086 instructions  

        stack         256        ; stack size is 256 bytes  

;  

;  

        dataseg  

;-----  

crlf    db  13,10,'$'  

prompt  db  "Enter a number (1 or 2) or 'q' to quit : ",'$'  

msgq   db  'Goodbye.',13,10,'$'  

msg1   db  'You selected 1.',13,10,13,10,'$'  

msg2   db  'You selected 2.',13,10,13,10,'$'  

msgerr db  'You did not select 1 or 2 or q.',13,10,13,10,'$'  

;  

;  

        codeseg  

;-----  

; Establish addressability to the data segment.  

;  

select:           ;  

        mov      ax,@data      ;establish addressability  

        mov      ds,ax          ;for the data segment  

;  

; Prompt the user for their input.  

; Read a key and save it.  

; Output a new line character pair.  

;  

progloop:         ;  

        mov      dx, offset prompt ;point to the prompt message  

        mov      ah,9              ;set the dos code to write a string  

        int      21h              ;write the string  

        ;  

        mov      ah,1              ;set the dos code to read a character  

        int      21h              ;read a character  

        mov      bl,al            ;save the character in bl  

        ;  

        mov      dx, offset crlf ;point to the new line character pair  

        mov      ah,9              ;set the dos code to write a string  

        int      21h              ;write the string

```

```

;-----  

; Test for 1 or 2 or q or other character.  

; - If q then the program is done.  

; - If 1 or 2 write the appropriate message.  

; - If anything else then write an error message.  

; Loop for the next input character.  

;-----  

testq:          ;  

    cmp     bl,'q'      ;test for quit  

    je      eof         ;yes, we are done  

;-----  

test1:          ;  

    cmp     bl,'1'      ;test for 1  

    jne    test2        ;no, do the next test  

    mov     dx, offset msg1  ;yes, point to correct message  

    jmp     writemsg    ;go write the message  

;-----  

test2:          ;  

    cmp     bl,'2'      ;test for 2  

    jne    error        ;no, we have an error  

    mov     dx, offset msg2  ;yes, point to correct message  

    jmp     writemsg    ;go write the message  

;-----  

error:          ;  

    mov     dx, offset msgerr  ;point to the error message  

;-----  

writemsg:  

    mov     ah,9      ;set the dos code to write a string  

    int     21h       ;write the string  

    jmp     progloop   ;repeat the process  

;-----  

;-----  

; Process the request to quit.  

; Write the goodbye message.  

;-----  

eof:            ;  

    mov     dx,offset msgq  ;point to the goodbye message  

    mov     ah,9      ;set the dos code to write a string  

    int     21h       ;write the string  

;-----  

; Terminate the program  

;-----  

exit:           ;  

    mov     ax,4c00h    ;set the exit code in ax  

    int     21h       ;int 21h will terminate program  

    end     select     ;end of source code  

;-----
```

```

;-----  

;  

; Program:    FILERW  

;  

; Function:   FILERW will read characters from a file that has been redirected  

;             to the standard input. At the end of each line it will write a  

;             message. At the end of file it will write a message.  

;  

;           It is intended to show how to use file I/O instructions.  

;  

;           It assumes that all lines end with a CR/LF (0Dh 0Ah).  

;           It assumes that all files end with a EOF      (1Ah).  

;  

;           To run it type: FILERW < in_file > out_file  

;           in_file  is the name of your input  file  

;           out_file is the name of your output file  

;  

; Owner:      Dana Lasher  

;  

; Date        Reason  

; -----  

; 07/18/2000 New version  

;  

;-----  

ideal          ; use turbo ideal mode  

model         small ; 64k data and 64k code model  

p8086          ; only allow 8086 instructions  

stack         256 ; stack size is 256 bytes
;-----
```

```

dataseg  

;-----  

linemsg db      '... end of line',13,10,13,10,'$'  

;  

; The end of line message is followed by CR/LF/CR/LF to provide double spacing.  

; It is then terminated by a $ which DOS uses to mark the end of a string.  

;  

;  

filemsg db      '... end of file',13,10,1Ah,'$'  

;  

; The end of file message is followed by CR/LF/EOF  

; which will create a valid final line with an EOF marker.
;-----
```

```

codeseg
;-----
; Establish addressability to the data segment.
;-----
start:           ;
    mov      ax,@data          ; establish addressability
    mov      ds,ax             ; for the data segment
;-----
; Read a character without echo and analyze it:
; - if the character is EOF then goto the
;   EOF code without echoing the character
; - for all other characters echo the character
;   to the output stream
; - if the character is EOL then goto EOL code
; - for all other characters loop to read another
;   character from the input stream
;-----
progloop:       ;
    mov      ah,8              ; read a character without echo
    int      21h               ; how: set ah=8 and issue int 21h
    ;
    cmp      al,1Ah            ; is the char eof
    je      eof                ; yes, go process eof
    ;
    mov      dl,al              ; no, move the char to dl
    mov      ah,2              ; echo the character from dl
    int      21h               ; how: set ah=2 and issue int 21h
    ;
    cmp      dl,0ah            ; is the char eol
    je      eol                ; yes, go process
    ;
    jmp      progloop          ; no, go process the next character
;-----
; Process the End Of Line condition
; - output the EOL message
; - go process the next line
;-----
eol:            ;
    mov      dx,offset linemsg ; set dx to point to the message
    mov      ah,9              ; set ah=9
    int      21h               ; int 21h will write the message
    ;
    jmp      progloop          ; go process the next line
;-----
; Process the End Of File condition
; - output the EOF message including an EOF marker
; - terminate the program
;-----
eof:            ;
    mov      dx,offset filemsg ; set dx to point to the message
    mov      ah,9              ; set ah=9
    int      21h               ; int 21h will write the message
;-----
; Terminate the program
;-----
exit:           ;
    mov      ax,4c00h          ; set correct exit code in ax
    int      21h               ; int 21h will terminate program
    end      start             ;
;-----
```

How can I build a null file to test my programs

Creating a NULL file, one that only contains an EOF (1Ah) character, can be difficult with a text editor. However it is easy to write a program that will generate a NULL file. The following program can be used to build a null file. That is a file with only a single EOF character. The source is in the course locker.

```
;-----  
; Program: MAKEEOF makes a file with just an eof character  
;  
; To run it type: MAKEEOF > OUT_FILE  
; where out_file is the name of your output file.  
;  
; Owner: Dana Lasher  
;  
; Note: This program does not require a data segment  
; since we have not declared any variables.  
;  
; Date Reason  
; ---- ----  
; 10/22/00 Original version  
;  
;-----  
ideal                      ; use turbo ideal mode  
model      small           ; 64k data and 64k code model  
p8086                   ; only allow 8086 instructions  
stack      256            ; stack size is 256 bytes  
;  
;  
codeseg  
;  
; Write out the DOS eof character = 1Ah  
;  
start:                   ;  
    mov      dl,1Ah          ; move eof char to dl  
    mov      ah,2             ; write eof from dl  
    int      21h             ; how: set ah=2 and issue int 21  
;  
; Return to DOS  
;  
exit:                    ;  
    mov      ax,4C00h         ; set correct exit code in ax  
    int      21h             ; int 21 will terminate program  
    end      start           ;  
;
```

Extended ASCII characters

When using int 21h to read from the keyboard, you normally get one character for each key pressed. For example pressing the upper case A key will return the hex value 41h. However, there are certain keys that do not have codes in the basic ASCII range 00h-7fh or even in the extended printable range of 80h-ffh. For example, the Function Keys F1 through F12 do not have an ASCII value.

When you press one of these keys, DOS will generate *two* characters. The first character is 00h which is returned to the program as an indicator that one of the special extended keys was pressed. The program is expected to issue a second read operation, at which time the value of the special extended key is returned.

For example, the F1 key produces the character pair 00h 3Bh. Normally the 3Bh would be interpreted as a semi-colon, but since it is preceded by the 00h flag it is interpreted as the F1 key.

We will not use any of these extended keys in this class. You may experiment with the program *testkeys.asm* that is in the samples course locker if you wish to see the values returned for the extended function keys.

Conversion algorithms

```
// Convert a valid ASCII string to hex positional number system format

determine the sign of the number
sum = 0
while current_char is a valid digit
    {sum = (sum * 10) + (current_char - 30h)}
if the sign is negative then sum = -sum

// Convert a positional binary number (word size) to an ASCII string
// Reduction of powers

if bin_num is negative then
{
    output minus sign
    bin_num = -bin_num
}
else
    output plus sign
div = 10,000
while (div > 0)
{
    calculate the quotient and remainder of bin_num / div
    output quotient + 30h
    bin_num = remainder
    div = div / 10
}
```

```

;-----  

;  

; Program: TESTFILE  

;  

; Function: This program will read ASCII characters from a file  

; redirected from the standard input stream. Each ASCII character read  

; will be converted to its hex equivalent, and then written to the  

; standard output device. For example the character 'A' will be output  

; as '41' and a CR will be output as '0D'. This allows the user to view  

; all the characters that exist in a file, including the control  

; characters. Some other examples of the conversion are:  

;  

;      Character      Internal      Converted      Output  

;      in file       representation   to          as  

;      -----  

;      A             41h           34 31         41  

;      =             3Dh           33 44         3D  

;      z             7Ah           37 41         7A  

;  

; The program does NOT require the input file have an EOF character to  

; work correctly. It uses the DOS function call to see if there are more  

; characters: set ah=0Bh, issue int 21h, if al=-1 (FFh) then a character  

; is available.  

;  

; To use, type this command: TESTFILE < INPUT > OUTPUT  

;  

; Output: The program prints the hex for 15 characters.  

;          Then it prints the 15 ASCII characters.  

;          Any control character, 00-1F, is printed as #  

;  

;  

;      <-----15 characters ----->      <----ASCII---->  

;  

;  

;      4D 5A 9B 00 02 00 01 00 20 00 11 00 FF FF 0A      MZ >##### ##YY#  

;      CD 21 3C FF 75 4B B4 08 CD 21 8A C8 3C 20 73      !<ÿuK`#!ŠÈ< s  

;      02 B0 23 88 04 46 8A D9 D1 EB D1 EB D1 EB D1      #°#^#FŠÙÑëÑëÑëÑ  

;  

;  

; Owner: Dana Lasher  

;  

; Date      Reason  

; -----  

; 06/15/97  Original version  

; 03/26/2000 Outputs ASCII characters and line break after 15 characters  

;  

;-----  

;      ideal                      ; use turbo ideal mode  

;      model small                ; 64k data and 64k code  

;      p8086                     ; only allow 8086 instructions  

;      stack 256                 ; stack size is 256 bytes  

;  

;  

;-----  

;      databeg  

;  

;-----  

; codes    db      '0123456789ABCDEF' ; codes of ASCII encoded hex digits  

;                                ;  

; break    db      15                  ; put a line break after 15 characters  

; ascii    db      5 dup(' ')        ; 5 spaces to format the output  

; ascii2   db      15 dup(' ')        ; room for 15 ASCII characters  

;                                ;  

;                                db      13,10,'$'          ; eol sequence  

;-----  


```

```

codeseg
;-----
; Establish addressability to the data segment.
; Initialize work registers.
; - bx is cleared to zero and will be used as
;   an index register in the conversion process
; - si is used to point to the string that holds
;   the last 15 characters read from input file
;-----
start:           ;
    mov      ax,@data          ; establish addressability
    mov      ds,ax              ; for the data segment
    mov      bx,0               ; clear index register
    mov      si, offset ascii2 ; point to the ASCII version
;-----
; Determine if there is another character to read
; using the DOS function 0Bh that determines if there
; are characters to be read from the standard input.
; - if not then terminate program
; - if yes then read the character
;-----
getloop:         ;
    mov      ah,0Bh            ; check if a character is available
    int      21h               ; how: set ah=0bh and issue int 21h
    cmp      al,0ffh           ; al = -1(dec) = FFh = char avail.
    jne      exit              ; if non available then exit
    ;
    mov      ah,8               ; read without echo a character
    int      21h               ; how: set ah=8 and issue int 21h
    mov      cl,al              ; save the character (cl=xy)
;-----
; We have read another character.
;
; We want to save the original character
; so it can be displayed and we also want
; to display the 2 hex digits that comprise
; each character. For example, the letter
; 'A' will be displayed as: 41 A
;
; If the character is a control character
; or a $ then it is not a printable
; character and we can not display its
; true ASCII value (e.g. tabs do not
; show on the screen). Instead, we will
; replace control characters and the $
; character with a # symbol.
;
; Store the ASCII version of the character
; in the next position of a 15 byte string.
;-----
        cmp      al,' '          ; is the character a printable char
        jae      store1            ; yes, store it
        mov      al,'#'            ; no, store a # symbol
store1:          ;
        cmp      al,'$'            ; replace the $ character with a #
        jne      store2            ; symbol also since DOS won't let
        mov      al,'#'            ; us write a $ as part of a string
store2:          ;
        mov      [si],al            ; save as ASCII data
        inc      si                ; move the ASCII pointer

```

```

;-----
; Convert each character represented by a
; hex byte into two printable ASCII characters.
; Example: character = 'A' = 41h.
; Output as ASCII '4' and ASCII '1'.
;
; Get the printable version of the most
; significant hex digit and print it.
; Example: character = 'A' = 41h
; Output an ASCII '4'
;-----
    mov     bl,cl          ; move the hex byte to bx (bx=00xy)
    shr     bx,1           ; isolate
    shr     bx,1           ; the left
    shr     bx,1           ; hex digit
    shr     bx,1           ; bx=000x
    mov     dl,[codes+bx]  ; use bx as index into ASCII codes
    mov     ah,2           ; write the ASCII code
    int     21h            ; how: set ah=2 and issue int 21h
;-----
; Get the printable version of the least
; significant hex digit and print it.
; Example: character = 'A' = 41h
; Output an ASCII '1'
;-----
    mov     bl,cl          ; move the hex byte to bx (bx=00xy)
    and     bx,000fh        ; isolate right hex digit (bx=000y)
    mov     dl,[codes+bx]  ; use bx as index into ASCII codes
    mov     ah,2           ; write the ASCII code
    int     21h            ; how: set ah=2 and issue int 21h
;-----
; Print a blank to separate characters.
;-----
    mov     dl,' '          ; write out a blank
    mov     ah,2           ; set up to write the character
    int     21h            ; how: set ah=2 and issue int 21h
;-----
; Keep track of the number of characters
; we have converted from hex to ASCII and
; displayed. After 15 characters write out
; the 15 original ASCII data characters
; and then write a line break.
;-----
    dec     [break]         ; reduce the count of characters written
    jne     getloop         ; we did not yet write 15 characters
    mov     [break],15       ; we did write 15, so reset the counter
    mov     dx, offset ascii ; point to original ASCII data
    mov     ah,9             ; get dos code to write a message
    int     21h             ; write the ASCII data and line break
    mov     si, offset ascii2 ; reset the ASCII pointer
    jmp     getloop         ; loop
;-----
; No more characters so terminate the program.
;-----
exit:               ;
    mov     ax,4c00h        ; set the correct exit code in ax
    int     21h             ; int 21h will terminate program
    end     start           ; execution begins at the label start
;-----
```


Multiply and Divide

More than any other instructions, multiply and divide have benefited from the increase in CPU chip designer's ability to put more circuits on a chip. These additional circuits have allowed architects to migrate the multiply and divide logic from lengthy shift and add operations to faster more sophisticated algorithms.

These are the number of machine cycles required for 16 bit register operations for the 8086 through Pentium.

	Unsigned multiply 16 bit register * register	Unsigned divide 32 register / 16 bit register
8086	118-133	144-162
80286	21	22
80386	9-22	22
80486	13-26	24
Pentium	11	25

In some cases, there seems to be a small increase in the number of cycles used as we move to a more advanced CPU. The instruction execution time, however, would be shorter because the clock speed of the more advanced CPU will be faster.

Different instructions are needed for unsigned and signed data

Unlike the previous arithmetic operations we have studied, such as add and subtract, the multiply and divide operations have different instructions for use with unsigned and signed numbers.

The reason for distinct instructions is that the exact same inputs may generate different results depending on whether they are signed or unsigned.

For example, look at what happens if we multiply the bytes (FFh) * (FFh).

	Decimal	Hex
Unsigned FFh * FFh	255 * 255 = 65,025	FE01
Signed FFh * FFh	-1 * -1 = +1	0001

How the size of the result is affected by the size of the source operands

When you multiply two numbers, the size of the result can be as large as the sum of the sizes of the two inputs.

	# digits in source	Result	# digits in result
9 * 9	1	81	2
99 * 99	2	9801	4
999 * 999	3	998001	6

Multiply

Terminology - The operation multiplies the *multiplicand* by the *multiplier* and generates a *product*.

$$\text{multiplicand} * \text{multiplier} = \text{product}$$

The opcode for unsigned multiplication is *mul* and the opcode for signed multiplication is *imul*.

This allows two instructions: *mul operand* or *imul operand*

- Multiply has a single operand that specifies the multiplier. That operand can be either a register, or a memory operand such as the name of a variable like [varx].
- The size of the operand determines:
 - the size and location of the multiplicand
 - the size and location of the result
- If the size of the operand is a byte then the multiplicand is the al register and the (al * byte) product is stored as a word in the ax register. The whole ax register is always modified by this operation.

$$\boxed{\text{al}} * \boxed{\text{byte}} = \boxed{\text{ax}} \quad \text{Result is a word that is stored in the ax register.}$$

- If the size of the operand is a word then the multiplicand is the ax register and the (ax * word) product is stored as two words in the dx:ax register pair. Both dx and ax are always modified by this operation.

$$\boxed{\text{ax}} * \boxed{\text{word}} = \overbrace{\boxed{\text{dx}} \quad \boxed{\text{ax}}}^{32 \text{ bit result stored in the dx : ax register pair}}$$

- You cannot multiply by an immediate value. For example: *mul 2* is not allowed.
- After a multiply, the whole result field is always set.
(byte * byte) always stores a whole word in ax
(word * word) always stores a long word in dx and ax
- An overflow can never occur. (al * byte) always fits in ax. (ax * word) always fits in the dx:ax register pair.
- The Carry Flag (cf) can be used to determine whether the result would still fit in the same size field as the source data. This works for *both signed and unsigned data*.
 - cf = 0 means the significant part of the result is the same size as the original multiplier.
That is (al * byte) correctly fits into al, or (ax * word) correctly fits into ax.
 - cf = 1 means the significant part of the result is larger than the original multiplier.
That is (al * byte) needs all of ax, or (ax * word) needs the dx:ax pair.
 - For the unsigned *mul* instruction, cf is set to 1 if the upper half of the result is not zero. The upper half of the result is ah when the two sources are bytes, and it is dx when the two sources are words.
 - For the signed *imul* instruction, cf is set to 1 if the upper half of the result is not the sign extension of the lower half of the result. The upper half of the result is ah when the two sources are bytes, and it is dx when the two sources are words.
- After the multiply, the other condition code flags are set as follows: sf and zf are undefined; of = cf.

Multiply examples (u=unsigned p=positive n=negative)

u10	db	10	; 0a	in hex
u100	db	100	; 64	in hex
n100	dw	-100	; ff9c	in hex 9cff byte reversed
p1000	dw	1000	; 03e8	in hex e803 byte reversed
p1	dw	1	; 0001	in hex 0100 byte reversed

; byte times byte with the result fitting in a byte
; even though the result will fit in a byte, the whole ax register is used

mov	al,[u10]	; ax = ?? 0a	(ah = ?? unknown al = 0a)
mul	[u10]	; ax = 00 64	cf = 0

; byte times byte with the result requiring a word

mov	al,[u100]	; ax = ?? 64	(ah = ?? unknown al = 64)
mul	[u10]	; ax = 03 e8	cf = 1

; word times word with the result requiring a double word (+1000 x -100 = -100,000 = fffe7960h)

mov	ax,[p1000]	; dx=???? unknown ax = 03e8
imul	[n100]	; dx:ax = fffe 7960 (dx = fffe ax = 7960) cf = 1

; word times word with the result fitting in a word (+1000 x +1 = +1000 = 0000 03e8h)
; note that dx is always set even if the result could fit in ax alone

mov	ax,[p1000]	; dx=???? unknown ax = 03e8
imul	[p1]	; dx:ax = 0000 03e8 (dx = 0000 ax = 03e8) cf = 0

The Carry Flag is only set to 1 if the result is too big to fit in the same size data field as the original data. Note in the following example, that even though the ah register is modified, the Carry Flag is set to 0 because ah is just the sign extension of al. The resultant value of -1 would fit in a byte.

n1	db	-1	;ff	in hex
p1	db	1	;01	in hex
mov	al,[p1]	; ax = ?? 01		
imul	[n1]	; ax = (+1) * (-1) = -1 = FFFF	CF = 0	

Remember ... you cannot multiply by an immediate value

You may be tempted to try to multiply by an immediate value. For example.

mov	ax,1000	;ax = 03e8
mul	10	;try to calculate 1000 * 10

However, that will not work.

Divide

Terminology - The operation divides the *dividend* by the *divisor* and generates a *quotient* and *remainder*.

dividend / divisor = quotient and remainder

The basic unsigned concept is

- The quotient is the number of times the magnitude of the divisor can be subtracted from the magnitude of the dividend without going negative.
- The remainder is what is left in the dividend after the quotient is calculated.

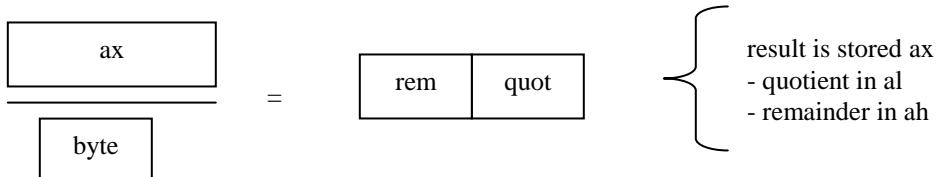
In high-level language, the operation / creates the quotient and the operation % creates the remainder.

For example: 10/3 yields quotient=3 remainder=1 7/15 yields quotient=0 remainder=7

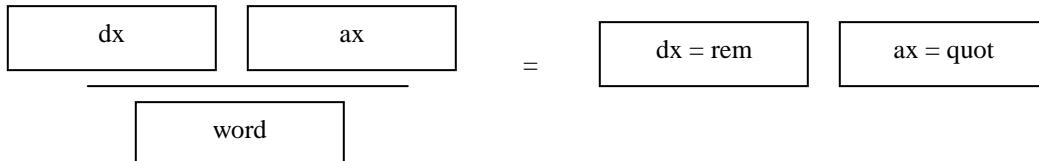
The opcode for unsigned division is *div* and the opcode for signed division is *idiv*.

This allows for two instructions: *div operand* or *idiv operand*

- Divide has a single operand that specifies the divisor. That operand can be either a register or a memory variable.
- The size of the operand determines:
 - the size and location of the dividend
 - the size and location of the quotient and remainder
- If the size of the divisor operand is a byte then the dividend is the ax register and the byte quotient is placed into the al register and the byte remainder is placed into the ah register.



- If the size of the divisor operand is a word then the dividend is the dx:ax register pair and the word quotient is placed into the ax register and the word remainder is placed into the dx register.



- You cannot divide by an immediate value. For example: *div 2* is not allowed.
- After the divide operation is completed, all the condition code flags are ***undefined***.
- If the quotient is too large to fit in the designated register (ax or al), or you divide by zero, then a divide overflow interrupt occurs and the program is terminated. You are ***not*** given the chance to test the condition code and to write error detection and recovery code. Either of those problems would be considered a programming bug.

Type of divide	Divisor size	Maximum allowed quotient in decimal (and hex)	
unsigned	word	65,535	(FFFFh)
unsigned	byte	255	(FFh)
signed	word	positive +32,767	(7FFFh) negative -32768 (8000h)
signed	byte	positive +127	(7Fh) negative -128 (80h)

Look at how the quotient and remainder are stored graphically.

- Unsigned word / byte

Calculate $10 / 3$ so that the quotient is 3 and the remainder is 1.

ax	rem	quot
ah	ah	al
00 0A	=	01 03
		03

- Signed word / byte

For signed numbers:

- The sign of the quotient is determined by the standard rules of arithmetic (like signs have a positive quotient, unlike signs a negative quotient).
- The sign of the remainder is equal to the sign of the dividend.
- The exception is a quotient or remainder of zero is always positive.

Look at the four signed combinations.

signed: $+10/+3$ ($q=+3$ $r=+1$)

ax	rem	quot
ah	ah	al
00 0A	=	01 03
		03

$+10/-3$ ($q=-3$ $r=+1$)

ax	rem	quot
ah	ah	al
00 0A	=	01 FD
		FD

signed: $-10/+3$ ($q=-3$ $r=-1$)

ax	rem	quot
ah	ah	al
FF F6	=	FF FD
		03

$-10/-3$ ($q=+3$ $r=-1$)

ax	rem	quot
ah	ah	al
FF F6	=	FF 03
		FD

You can verify the sign and magnitude of a quotient and remainder by using this verification formula: $\text{dividend} = (\text{divisor} * \text{quotient}) + \text{remainder}$

Divide examples

Examples: (u=unsigned p=positive n=negative)

n3	db	-3	;	fd	in hex
p10	dw	10	;	000a	in hex 0a00 byte reversed
p17	dw	17	;	0011	in hex 1100 byte reversed
n100	dw	-100	;	ff9c	in hex 9cff byte reversed
p1000	dw	1000	;	03e8	in hex e803 byte reversed

; A word divided by byte (+10 / -3 generating a quotient of -3 and a remainder of +1)

mov	ax,[p10]	;	ax= 00 0a +10
idiv	[n3]	;	ax= 01 fd rem=+1 quot=-3

; A chained operation. A multiply generating a double word which is divided by a word
; (+1000 * -100) / (+17) = -5882 with a remainder of -6

mov	ax,[p1000]	;	ax= 03 e8
imul	[n100]	;	dx:ax=ffff 7960 = (-100,000 decimal)
idiv	[p17]	;	rem=dx=ffffa (-6 decimal) quot=ax=e906 (-5882 decimal)

Overflow detection

For unsigned numbers, it is possible to detect, **before performing the divide**, whether the divide will cause an overflow. Let us look at this in unsigned decimal first. Assume you want to divide a 4 digit decimal number by a 2 digit decimal number, and that a valid quotient must fit in 2 digits. With the values that follow, an overflow would occur because the correct result should be 108 with a remainder of 15.

$$\begin{array}{r} \underline{27} \quad 15 \\ \underline{25} \end{array} = \quad 01 \ 08 \quad \text{Remainder} = 15$$

Since the high 2 digits of the dividend (27) is greater than the divisor (25) we can tell **before dividing** that the result will be 100 or more which does not fit in 2 digits, thus causing a divide overflow.

The same idea holds true in unsigned hex. When you do an unsigned divide of a word dividend by a byte divisor, if the magnitude of the high order byte of the dividend \geq the magnitude of the byte divisor then the result will be too big to fit in a byte quotient thus causing an error. Look at calculating $c=a/b$ where a is an unsigned word, b is an unsigned byte, cq is the byte quotient, cr is the byte remainder.

a	dw	?	; unsigned word dividend
b	db	?	; unsigned byte divisor
cq	db	?	; unsigned byte quotient
cr	db	?	; unsigned byte remainder

```
mov ax,[a] ;ax=dividend=hhll where hh is the high order byte and ll is the low order byte
cmp ah,[b] ;see if the high order byte is equal or greater than the divisor
jae error ;yes, we cannot divide
div [b] ;no, divide
mov [cq],al ;store the quotient
mov [cr],ah ;store the remainder
```

Case 1. a=1000 decimal (03e8h) and b=4 so the calculation is 1000/04 (quotient=250 (fah) remainder=0)

```
mov ax,[a] ;ax=03e8
cmp ah,[b] ;ah compared to b = 03:04
jae error ;03 is less than 04 so we do not jump
div [b] ;ah=00 al=f0
mov [cq],al ;store the quotient
mov [cr],ah ;store the remainder
```

Case 2. a=1000 decimal (03e8h) and b=2 so the calculation is 1000/02 (*expected* quotient=500 (1f4h) remainder=0)

```
mov ax,[a] ;ax=03e8
cmp ah,[b] ;ah compared to b = 03:02
jae error ;03 is greater than 02 so the result will be an error and we take the jump
div [b] ;not executed in this example
mov [cq],al ;not executed in this example
mov [cr],ah ;not executed in this example
```

- The same technique can be used if you are doing an unsigned divide of a longword dividend in the dx:ax pair by a word divisor. You cannot divide if $dx \geq$ the word size divisor.
- This test does not work for signed divides. For signed divides, convert the values to decimal and manually divide and see if the quotient fits in the allocated result size.

Examples for you to try ... solutions follow

1. Calculate $(a*b)/c$ where a,b,c are unsigned bytes
- store quotient in q
- store remainder in r
- on any error goto instruction labeled 'error'

a	db	?	;unsigned byte
b	db	?	;unsigned byte
c	db	?	;unsigned byte
q	db	?	;unsigned quotient
r	db	?	;unsigned remainder

2. Given x is a signed byte
- if x is odd goto instruction labeled 'isodd'
- if x is even goto instruction labeled 'iseven'
Solve this by coding the following design:
If remainder of $x/2 = 0$ goto iseven else goto isodd

x	db	?	;signed byte
---	----	---	--------------

3. Given y is an unsigned byte
Calc $y = 1/y$
On any error goto an instruction labeled 'error'

y	db	?	;unsigned byte
---	----	---	----------------

4. Given that 'a' is a signed byte, calculate a^4
You can leave the result in register(s)
Note that $a^4 = a * a * a * a$

a	db	?	;signed byte
---	----	---	--------------

5. Given the radius of a circle is an unsigned byte 'r'
and pi is approximated as 3
- calculate the area of the circle: $area=pi*r^2$
- if area is too big to fit in a word then goto 'error'
Additional Question: what is the largest value of 'r'
that will allow 'area' to fit in an unsigned word.

r	db	?	;unsigned byte
pi	db	3	;unsigned byte
area	dw	0	;unsigned word

solutions

1. Calculate $(a*b)/c$ where a,b,c are unsigned bytes
- store quotient in q
- store remainder in r
- on any error goto instruction labeled 'error'

a	db	?	;unsigned byte
b	db	?	;unsigned byte
c	db	?	;unsigned byte
q	db	?	;unsigned quotient
r	db	?	;unsigned remainder

mov	al,[a]	;ah=xx al=a
mul	[b]	;ax=a*b
cmp	ah,[c]	;high order byte : divisor
jae	error	;if = or > then can't divide
div	[c]	;divide
mov	[q],al	;store quotient
mov	[r],ah	;store remainder

2. Given x is a signed byte
- if x is odd goto instruction labeled 'isodd'
- if x is even goto instruction labeled 'iseven'
Solve this by coding the following design:
If remainder of $x/2 = 0$ goto iseven else goto isodd

x	db	?	;signed byte
---	----	---	--------------

mov	al,[x]	;ah=xx al=x
cbw		;ax=x (convert byte to word)
mov	bl,2	;get the value 2
idiv	bl	;calc x/2 ah=rem al=quot
cmp	ah,0	;is remainder zero
je	iseven	;yes, goto iseven
jmp	isodd	;no, goto isodd

3. Given y is an unsigned byte
Calc $y = 1/y$
On any error goto an instruction labeled 'error'

```
y    db      ?          ;unsigned byte
```

This solution does the mathematics.

```
cmp      [y],0      ;is y=0
je       error      ;yes, then we cannot divide
mov      ax,1       ;ax=0001
div      [y]        ;ah=rem  al=quot
mov      [y],al     ;set y=1/y
```

This solution uses logic.

If $y=0$ then we have an error

If $y=1$ then $1/y = 1$

If $y>1$ then $1/y = 0$... remember this is integer division

```
cmp      [y],0      ;is y=0
je       error      ;yes, can't calculate answer
cmp      [y],1      ;is y=1
je       done       ;yes, then 1/y=1
mov      [y],0       ;no, then 1/y=0
done:
```

4. Given that 'a' is a signed byte, calculate a^4
You can leave the result in register(s)
Note that $a^4 = a * a * a * a$

```
a    db      ?          ;signed byte
```

This code works.

```
mov      al,[a]      ;ah=--  al=a
imul    [a]         ;ax=a*a
imul    ax          ;dx:ax = a*a*a*a
```

This code will not work. The second and third
multiplies attempt to multiply a word * a byte.

```
mov      al,[a]      ;ah=--  al=a
imul    [a]         ;ax=a*a
imul    [a]         ;ax=a*a*a ????
imul    [a]         ;ax=a*a*a*a ????
```

5. Given the radius of a circle is an unsigned byte 'r'
 and pi is approximated as 3
 - calculate the area of the circle: $\text{area}=\pi \times r^2$
 - if area is too big to fit in a word then goto 'error'

Additional Question: what is the largest value of 'r'
 that will allow 'area' to fit in an unsigned word.

```

r      db      ?          ;unsigned byte
pi     db      3          ;unsigned byte
area   dw      0          ;unsigned word

mov     al,[r]           ;ah--- al=r
mul     [r]              ;ax=r*r
mov     bh,0             ;bh=0   bl=---
mov     bl,[pi]          ;bh=00   bl=03
mul     bx               ;dx:ax = pi*r*r
jc      error            ;error if area > word
mov     [area],ax         ;area=pi*r*r

```

For area to fit in a word then $\pi \times r^2 < 65536$.

Using $\pi=3$ then $r^2 < 65536/3$ or $r^2 < 21845$ or $r < 148$.

The largest value of r that allows the area to fit in a word is 147.
 If $r=147$ then $\text{area}=64,827$, but if $r=148$ then $\text{area}=65,712$.

Working with other data sizes

What if you need to multiply data sizes for which there is no hardware support, such as longword * longword?
You then need to develop software extensions.

I will demonstrate the basic idea using *decimal* examples.

Assume your hardware will multiply a 2 digit number times a 2 digit number yielding a 4 digit result.

For example: $17 * 15 = 02\ 55$.

You want to multiply a 4 digit number times a 4 digit number yielding an 8 digit result.

For example: $1234 * 5678 = 07,\ 006,\ 652$

This can be rewritten as: $(12*100 + 34) * (56*100 + 78)$

Now we can use standard algebra that says $(a + b) * (c + d) = (a*c) + (b*c) + (a*d) + (b*d)$.

a=12*100

b=34

c=56*100

d=78

The calculation now becomes $((12*100)*(56*100)) + ((34)*(56*100)) + ((12*100)*(78)) + ((34)*(78))$

We group multiplications of 100, yielding: $(12*56*100*100) + (34*56*100) + (12*78*100) + (34*78)$

To perform the operation, we create an 8 digit result field 00 00 00 00. We then do each component multiplication.

*However, we will not actually multiply by 100 or 100*100.*

- We simulate multiplying by 100 by just shifting that result left by 2 decimal digits when we add it into its correct position within the final 8 digit result.
- We simulate multiplying 100*100=10,000 by just shifting that result left by 4 decimal digits when we add it into its correct position within the final 8 digit result.

The calculation of $1234 * 5678$ then becomes the following. The underlined 00s indicate where we shifted one of the component results by 100 or by 10,000.

$12*56*100*100$	$\begin{array}{r} 1\ 2\ 1 \\ 06\ 72\ 00\ 00 \end{array}$	(where $12*56 = 0672$ is done by hardware)
$34*56*100$	$\begin{array}{r} 19\ 04\ 00 \\ 09\ 36\ 00 \end{array}$	(where $34*56 = 1904$ is done by hardware)
$12*78*100$	$\begin{array}{r} 09\ 36\ 00 \\ 26\ 52 \end{array}$	(where $12*78 = 0936$ is done by hardware)
$34*78$		(where $34*78 = 2652$ is done by hardware)
$07\ 00\ 66\ 52$		

This concept can be used to multiply any data size times any data size.

Indirect Addressing

The topic of indirect addressing and pointers has an interesting Computer Science political twist that we will discuss before getting into a technical discussion. If you go back to 1965 and look at the body of Computer Science knowledge, you would see something like the table on the left. The total amount of information was manageable and the way to learn about computers was to start at the bottom and work upward.

Applications
Operating Systems
High Level Languages
Assembler Language
Machine Code
Hardware

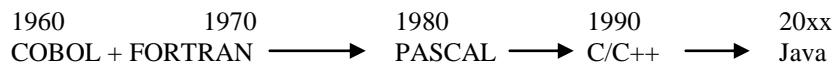
Computer Science Circa 1965

Artificial Intelligence, Networking, E-commerce, WEB development, Real time, data compression, etc.
Applications
Operating Systems
High Level Languages
Assembler Language
Machine Code
Hardware

Computer Science Circa 20xx

We are now in the 21st century and the body of Computer Science knowledge is vast, with the addition of dozens of sophisticated topics. In order to allow students to reach these advanced topics in a four-year curriculum, the education process starts at the High Level Language layer and then works upward toward the new topics and not downward toward the hardware.

In addition, the complexity of the High Level Languages has increased. The industry started with two languages, COBOL for business applications and FORTRAN for scientific applications. These merged into a combined language, PASCAL. PASCAL evolved to C/C++, which evolved to Java.



The characteristics of C/C++	The characteristics of Java
Full function and flexible with few confining rules.	More structure and rigidity, with the emphasis on code reuse, fast development, and the GUI interface.
The base language is portable but any GUI is not easily moved across platforms.	Truly platform independent and portable.
Compiled down to specific machine code for each hardware platform and thus high performance.	Compiled to Java Bytecode, which is interpreted for each hardware platform, which does not optimize performance.

As part of making Java truly portable, the language moved *further away from any specific hardware platform*. Two hardware related concepts, that have been part of previous languages, are not part of the Java language accessible by the programmer.

- Unsigned primitives
- Pointer manipulation

The question is whether it is important for a Computer Scientist to be knowledgeable about these two topics and in general be knowledgeable below the High Level Language layer. The con side argues that the realm below High Level Languages belongs to the Computer Engineers and Electrical Engineers. The pro side argues that you cannot effectively and efficiently work at or above the High Level language level without understanding how the hardware really functions.

Why is Indirect Addressing Important?

Indirect addressing, which is often referred to as *pointers* in high-level languages, allows us to easily process a list of data items.

For example, without indirect addressing it would be difficult to sum a list of numbers. Each number would need to be declared individually, and a separate add instruction would be needed for each number.

This is the way it would have to be done, which would be very tedious if the list had 1000 values.

```
;      declare the data
a    dw    ?          ; declare a
b    dw    ?          ; declare b
c    dw    ?          ; declare c

;      sum the data
mov   ax,0           ; sum = 0
add   ax,[a]         ; sum = sum + a
add   ax,[b]         ; sum = sum + b
add   ax,[c]         ; sum = sum + c
```

Basic Indirect Addressing Implementation

To use Indirect Addressing, we declare the data as a list.

```
list      dw      10 dup (0)      ; declare a list of 10 words all initialized to zero
```

Either our code or some other code would fill in or modify the contents of the list.

What we wish to do now is execute this design: **For i = 0 to 9 do sum = sum + list[i]**

In this first version of code we will be very basic and not try any optimization.

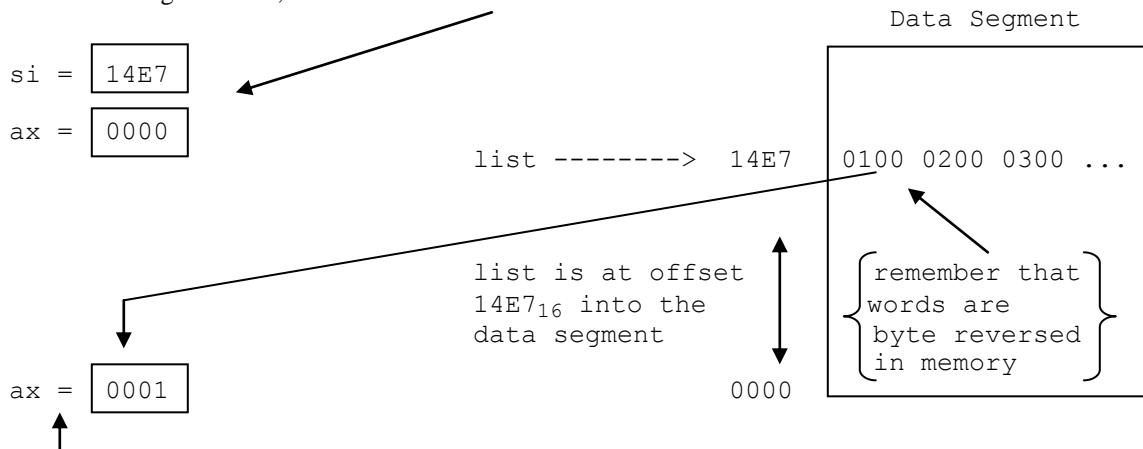
```

    mov     ax,0          ; (I1) ax contains the sum and is initialized to zero
    mov     cx,0          ;       cx represents i which is the loop counter and it is set to zero
    mov     si,offset list ; (I3) si is the index into the list and starts by pointing to the first element
calc:
    add     ax,[si]        ; (I4) sum = sum + current element
    add     si,2            ;       advance pointer to the next element ... add 2 since each element is a word
    inc     cx              ;       add 1 to the count ... i = i+1
    cmp     cx,10           ;       test for all elements 0...9 processed
    jne     calc             ;       no ... loop back
done:
```

- Let us examine the instruction (I3): *mov si, offset list*

The terms *offset* and *address* have the same meaning. They both refer to a location in the data segment. Also, remember that the name of a variable is equivalent its offset into the data segment, and the name of a list is equivalent to the offset into the data segment of the first element. So the instruction *mov si, offset list* loads the si register with the address of the first element in the list. In the example below that offset is $14E7_{16}$.

After executing I1 and I3, si and ax have these values.



- Let us examine the instruction (I4): *add ax,[si]*

The source operand is *[si]*. The brackets tell the system this is a memory reference. *si* is not the name of a variable. Instead *si* is one of the pointer registers. This tells the system that the memory referenced is located at the address contained in the *si* register. So the system goes to memory location $14E7$, gets the data at that address, byte reverses it, and adds it to *ax*. So after instruction I4 the *ax* register will contain 0001 .

Indirect Addressing Implementation Using The LOOP Instruction

Now let us do a little optimization.

Since controlling the flow through a loop is so common, Intel has a special instruction *loop* that will reduce the amount of code we need to write.

The *loop* instruction works as follows:

- Place the count of the number of times you want to go through the loop in the cx register.
This must be in the range of 1 to 65535. It should not be zero!
- The number in cx is treated as an unsigned count.
- The cx register is decremented by 1.
- If cx is not zero the loop is repeated.
- The *loop* instruction does not modify the condition code flags.
- An initial value of zero in cx will cause the loop to be repeated 65,536 times.

Using *loop* the code now becomes:

```
list    dw      10 dup (0)      ; declare a list of 10 words all initialized to zero

        mov     ax,0          ; sum = 0
        mov     cx,10         ; i = loop count = 10
        mov     si,offset list ; si = index into the list and starts by pointing to the first element
calc:   add     ax,[si]       ; sum = sum + current element
        add     si,2          ; advance the pointer to the next element
        loop   calc          ; decrement cx and loop back to calc: if cx not equal to zero
done:
```

Data Size Override

In this example we have a list of signed words. We scan the list and replace any negative numbers with zero.

```
list    dw      -3,7,100,-83,0,1000      ; list of signed words
count   dw      6                      ; number of items in the list

;      before entering the loop, make sure the count is not zero
cmp     [count],0                  ; is the count zero
je      done                     ; yes, then do not enter the loop

;      the count is not zero so we can process the list
mov     si,offset list           ; initialize a pointer to the list
mov     cx,[count]                ; get the count of items in the list
testit:
cmp     [word ptr si],0          ; compare the word in the list to which si points to zero
jge     next                     ; if it is zero or above then there is no work to do
mov     [word ptr si],0          ; if it is negative then replace the negative value with zero
next:
add     si,2                     ; move the pointer to the next element in the list
loop   testit                   ; and loop if more data to process in the list
done:
```

In this example we see a new concept called '*data size override*' in the instruction: `cmp [word ptr si],0`

The assembler **needs to know the size of the data**, byte or word, to use.

Normally it gets that size by analyzing the operands.

For example in `cmp [si],ax` it knows that ax is a word and, therefore, the data that si points to in the list must also be a word.

However in the instruction: `cmp [si],0` the assembler cannot figure out the data size.

- The source operand of zero can be either byte or word.
- The destination operand `[si]` can be either byte or word.

Since the assembler cannot figure out the data size, we **must** tell it whether the data is byte or word.

Either of these could be used depending on the actual size of the data items in the list.

- `cmp [word ptr si],0` ; compare the data word in memory pointed to by si to zero
- `cmp [byte ptr si],0` ; compare the data byte in memory pointed to by si to zero

Indirect Addressing Options

Notes

1. The four pointer registers that are allowed with the brackets [] are SI, DI, BX, BP.
2. The pointer registers are not changed by their use in the examples below.
3. In calculating the effective address, the arithmetic is performed and any carries or overflows are ignored.

- **Register Indirect:** The user places the offset to a variable into one of these registers: SI, DI, BX, BP. The instruction then uses that register to indirectly access the data.

- References to SI, DI, BX access data in the data segment.
- References to BP access data in the stack segment.

In high-level languages this addressing mode is the basis for implementing pointers.

- `add ax,[bx]` ; add the contents of memory pointed to by bx into ax

- **Register Indirect with Displacement:** The effective address of the data is the offset equal to the contents of the pointer register (SI, DI, BX, BP) plus or minus the displacement. The displacement is a constant. References to SI, DI, BX access data in the data segment. References to BP access data in the stack segment.

- `add ax, [bx + 8]` ; add the contents of memory pointed to by bx + 8 into ax
- `mov cx, [si - 1000]` ; move the contents of memory pointed to by si - 1000 into cx
- `inc [byte ptr di + 27]` ; add 1 to the byte of data pointed by di + 27

- **Register Indirect with Index and Displacement:** This is the general form of indirect addressing that allows the user to specify two pointer registers and an offset (which is a constant). One register must be either BX or BP. The other register must be SI or DI. The effective address is the sum of the contents of the two registers plus or minus any offset specified. Data is taken from the data segment if BX is used, and the stack segment if BP is used. *If you specify the offset to be the name of a data item, it will use the address of the data item and not the contents of the data item.*

- `add ax,[bx + si + 10]` ; add the data segment word pointed to by bx + si + 10 into ax
- `sub al,[bx + di - 1750], al` ; subtract al from the data segment byte pointed to by bx + di - 1750
- `mov ax,[bp + si + 1]` ; move the *stack segment* word pointed to by bp + si + 1 into ax
- `mov ax,[bx + di + varx]` ;warning: the effective address is the contents of bx + contents of di + address of varx. It does not use the contents of varx.

There are two types of *override* that the programmer may specify.

- **Data Size Override** - If the assembler cannot figure out the data size, we *must* tell it whether the data is byte or word. Either of these could be used depending on the actual size of the data in the list.

TASM	<code>cmp [word ptr si], 0</code>	<code>cmp [byte ptr si], 0</code>
MASM	<code>cmp word ptr [si], 0</code>	<code>cmp byte ptr [si], 0</code>

- **Segment Override** - If you run low on data segment pointer registers (BX, SI, DI) you can override the default use of the stack segment for the BP register. The segment override operation (ds: es: ss: cs:) tells the hardware to use a specific segment instead of the default segment. The instructions below get data from the data segment even though the primary indirect register specified is BP which normally points to the stack segment.

TASM	<code>add ax, [ds:bp + si + 10]</code> ; data in data segment at offset = bp + si + 10
TASM	<code>mov [byte ptr ds:bp], 0</code> ; move 0 to the byte of data in the data segment at offset = bp
MASM	<code>add ax, ds:[bp + si + 10]</code> ; data in data segment at offset = bp + si + 10
MASM	<code>mov byte ptr ds:[bp], 0</code> ; move 0 to the byte of data in the data segment at offset = bp

Using the segment override operation (ds: es: ss: cs:) you can force the system to go to any specific segment.

Example 1A, a few pages further on, shows you exactly how to write code using the segment override.

Summary Of All Indirect Options

Valid pointer registers allowed within brackets are
bx bp si di

Indirect Addressing Components	Format
	<ul style="list-style-type: none">• These are the only valid combinations• In calculating the effective address the arithmetic is performed and any carries or overflows are ignored
base or index	[bx] [bp] [si] [di]
base or index plus/minus displacement	[bx ± n] [bp ± n] [si ± n] [di ± n]
base and index	[bx + si] [bx + di] [bp + si] [bp + di]
base and index plus/minus displacement	[bx + si ± n] [bx + di ± n] [bp + si ± n] [bp + di ± n]

Notes.

- A common mistake is to code ... [bx + var] ... thinking the effective address will be sum of the contents of bx and the contents of var. That is not correct.

The value of n denotes a constant and *not* the contents of a variable. If you specify n to be the name of a data item, it will use the *address* of the data item and not the *contents* of the data item.

If you want to use the contents of a variable then that variable must be loaded into a pointer register and you would use the base and index format.

- The order you code parameters is *not significant*. All these are identical.
[bx + si + 1] [si + bx + 1] [1 + si + bx]
[bx + 1 + si] [si + 1 + bx] [1 + bx + si]
- Any combination that does not use bp, points to the data segment.
[bx] [si] [di] [bx ± n] [si ± n] [di ± n] [bx + si] [bx + di]
[bx + si ± n] [bx + di ± n]
- Any combination that does use bp, points to the stack segment.
[bp] [bp ± n] [bp + si] [bp + di] [bp + si ± n] [bp + di ± n]
- Using the segment override operation (ds: es: ss: cs:) you can force the system to go to any specific segment.

TASM places the override inside the brackets ... [ds:bp + si]
MASM places the override outside the brackets ... ds:[bp + si]

The Mechanics Of Indirect Addressing And Effective Address Calculations

We start with the following data declarations.

```
        databaseg
var1    dw    100
var2    db    100
list1   dw    10,20,30,-10,-20,-30
list2   db    'ABCabcXYZxyz'
list4   db    '01234567890'
```

Which generates this hex data in the data segment.

Hex Offset	8 Bytes of hex data ...
0000:	64 00 64 0A 00 14 00 1E
0008:	00 F6 FF EC FF E2 FF 41
0010:	42 43 61 62 63 58 59 5A
0018:	78 79 7A 30 31 32 33 34
0020:	35 36 37 38 39 30

The current hex values of the 8086 registers.

ax=0123 bx=0003 cx=8008 dx=FFFF
si=0011 di=000A bp=0111 sp=0100 ds=20A0

- The effective address is the offset of the data into the data segment.
- The effective address is the sum of all the items inside the brackets [].
- The absolute address would be: $ds_register_{16} * 10_{16} + effective_address_{16}$

Register contents are already in hex, but any displacement in the instruction is in decimal. The final effective address must be in hex. You must byte swap any words of data moved from memory to a register. Some of the effective addresses and absolute address calculations are given. You may do the others.

What is the hex value in al after executing: mov al,[si] al=43
Effective address = 0011h Absolute address = 20A11h

What is the hex value in al after executing: mov al,[di+1] al=EC

What is the hex value in al after executing: mov al,[bx+di+5] al=61
Effective address = 0003h + 000Ah + 0005h = 0012h
Absolute address = 20A0h * 10h + 0012h = 20A12h

What is the hex value in ax after executing: mov ax,[bx] ax=000A

What is the hex value in ax after executing: mov ax,[bx+2] ax=0014

What is the hex value in ax after executing: mov ax,[bx+29] ax=3635
Effective address = 0003h + 001Dh = 0020h
Absolute address = 20A0h * 10h + 0020h = 20A20h

What is the hex value in ax after executing: mov ax,[bx+si] ax=5863

What is the hex value in ax after executing: mov ax,[bx+di+11] ax=7978

What is the hex value in ax after executing: mov ax,[si-2] ax=4241

What is the hex value in ax after executing: mov ax,[bx+si-2] ax=6261
Effective address = 0003h + 0011h - 0002h = 0012h

Example 1: Building An Output String Dynamically

The first example we looked at went through a list and converted negative values to zero.

This example dynamically builds a list in memory that contains the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ... It then prints this list.

list+0	list+1	list+2	list+3	list+4	list+5	list+6	list+7	list+8	list+9
1	1	2	3	5	8	13			

The program stops when it builds a value that is greater than 9. The reason is that it is very easy to convert single digit numbers (0-9) to printable ASCII characters. To convert a single digit value to ASCII you just add hex 30.

The program contains examples of all the basic indirect functions. It is broken into small blocks to make it easier to follow. I trust it will help you understand indirect operations.

```
;-----
; Program: Fib
;
; Function: This program calculates the Fibonacci sequence 1,1,2,3,5,8,13,...
;           The first two values are 1
;           Each succeeding value n = (n-1) + (n-2)
;
;           It stops when it gets a value that is more than 9 (one digit)
;           It then converts the numbers from hex to ascii
;           The hex value of a digit + 30h = the ascii value
;           (04h + 30h = 34h = '4')
;           It then prints out the sequence
;
;-----
ideal                                ;use turbo ideal mode
model      small                   ;64k code and 64k data
p8086                               ;only allow 8086 instructions
stack      256                    ;reserve 256 bytes for the stack
;-----

;-----  
; declare an uninitialized list of 80 bytes  
-----  
        dataseg                ;  
list      db       80 dup(?)          ;declare 80 uninitialized bytes  
-----  
  
-----  
; initialize the ds register  
-----  
        codeseg                ;  
start:  
        mov       ax,@data          ;establish addressability to the  
        mov       ds,ax            ;data segment for this program  
-----
```

```

;-----
; The Fibonacci sequence is 1,1,2,3,5,8,13,...
; Set the first two Fibonacci numbers to 1
; Set the pointer to the 3rd Fibonacci number location
;-----
    mov      [list + 0],1          ;fib number 1 = 1
    mov      [list + 1],1          ;fib number 2 = 1
    mov      si, offset list + 2 ;point to fib number 3
;-----

;-----
; Calculate the next Fibonacci number as (n-2) + (n-1)
; Repeat until the Fibonacci number is above 9
;-----
next_fib:                      ;
    mov      al, [si - 2]        ;get n-2
    add      al, [si - 1]        ;calc (n-2) + (n-1)
    mov      [si], al           ;store that as current fib number
    inc      si                 ;advance the pointer
    cmp      al,9               ;is current fib number > 9
    jbe      next_fib          ;no, calculate next fib number
;-----


;-----
; Convert the single digit hex Fibonacci numbers to ascii
; This is done by adding 30h which is the character '0'
;-----
    mov      si, offset list      ;point to the start of the list
convert:                         ;
    cmp      [byte ptr si],9     ;is current fib number > 9
    ja       print               ;yes, go print the list
    add      [byte ptr si],'0'   ;no, convert fib number to ascii
    inc      si                 ;advance the pointer
    jmp      convert             ;convert the next fib number
;-----


;-----
; Store a CR LF (end of line) and $ (end of string) in the string
; Print the string
;-----
print:                          ;
    mov      [byte ptr si    ],13 ;CR after last single digit fib #
    mov      [byte ptr si + 1],10 ;store LF after CR
    mov      [byte ptr si + 2],'$' ;store DOS end of string after LF
    mov      dx, offset list     ;point to the list
    mov      ah,9                ;DOS request to print a string
    int      21h                 ;print
;-----


;-----
; Terminate the program
;-----
exit:                           ;
    mov      ax,4c00h            ;set dos code to terminate program
    int      21h                 ;return to dos
    end      start               ;end of the source code

```

Example 1A ... Building An Output String ... using the BP register & segment override

I have removed all block and header comments so it fits on one page.

```
ideal                                ;use turbo ideal mode
model      small                   ;64k code and 64k data
p8086
stack      256                   ;only allow 8086 instructions
                           ;reserve 256 bytes for the stack

databseg
list      db        80 dup(?)      ;declare 80 uninitialized bytes

codeseg
start:   mov      ax,@data       ;establish addressability to the
         mov      ds,ax           ;data segment for this program
;-----
; Set first two Fib numbers to 1 ... point to 3rd Fib number location
;-----
        mov      [list + 0],1      ;fib number 1 = 1
        mov      [list + 1],1      ;fib number 2 = 1
        mov      bp, offset list + 2 ;loading bp is the same as loading
                           ;any other register
;-----
; Repeat calculating the next Fibonacci number as (n-2) + (n-1)
;-----
next_fib:mov    al, [ds:bp - 2]    ;using bp inside the brackets
               add    al, [ds:bp - 1]    ; requires the use of the ds:
               mov    [ds:bp], al       ; segment override
               inc    bp                ;inc bp just like any register
               cmp    al,9              ;is current fib number > 9
               jbe    next_fib          ;no, calculate next fib number
;-----
; Convert the single digit hex Fibonacci numbers to ascii
;-----
convert:  mov      bp, offset list ;point to the start of the list
         cmp      [byte ptr ds:bp],9 ;is current fib number > 9
         ja     print             ;yes, go print the list
         add    [byte ptr ds:bp], '0' ;no, convert fib number to ascii
         inc    bp                ;advance the pointer
         jmp    convert           ;convert the next fib number
;-----
; Store a CR LF 'S' in the string and Print the string
;-----
print:   mov      [byte ptr ds:bp    ],13 ;store CR after last fib number
         mov      [byte ptr ds:bp + 1],10 ;store LF after CR
         mov      [byte ptr ds:bp + 2], '$' ;store DOS end of string after LF
         mov      dx, offset list      ;point to the list
         mov      ah,9                ;DOS request to print a string
         int      21h                ;print
;-----
; Terminate the program
;-----
exit:    mov      ax,4c00h      ;set dos code to terminate program
         int      21h                ;return to dos
         end      start             ;end of the source code
```

Example 2: Table Lookup Using Indirect Addressing

Sometimes you know all the answers to a specific question in advance.

So when asked the question, instead of calculating the answer, you just look the answer up.

For example, say we have an insurance application. Based upon the age of the person, we want to know the cost per month for 1 unit (usually \$1000) of life insurance.

We build a table:

- At offset zero into the table is the cost per month if the person is 0 to 1 year old.
- At offset 1 into the table is the cost per month if the person is older than 1 to 2 years old.
 - In general, at offset 'n' into the table is the cost per month if the person is 'n' years old. We then use the person's age as an index into the table to get the cost per month.

costtable	db	20 dup(1)	; 0 - 19 years old is \$1 per month
	db	10 dup(4)	; 20 - 29 years old is \$4 per month
	db	10 dup(5)	; 30 - 39 years old is \$5 per month
	db	10 dup(6)	; 40 - 49 years old is \$6 per month
	db	10 dup(10)	; 50 - 59 years old is \$10 per month
	db	40 dup(25)	; 60 - 99 years old is \$25 per month
	db	156 dup(100)	; 100 - 255 years old is \$100 per month

age	db	?	; the age of the person is in this field
cost	db	?	; we will place the cost per month in this field

```
-----
; Method 1 uses indirect addressing with displacement
; Put the age in the bx register.
; Use bx as a pointer register.
; Use the offset to the start of the table as a fixed displacement.
; Locate the data at the location that is the sum of bx (age) and the start of the table.
-----
```

```
        mov  bx,0          ; clear the index register
        mov  bl,[age]       ; get the age as an index value
        mov  al,[bx + costtable] ; get the correct cost
        mov  [cost],al      ; set the cost field
```

```
-----
; Method 2 uses indirect addressing with index
; Put the offset of the start of the table into the si register.
; Put the age in the bx register.
; Locate the data at the location that is the sum of bx (age) and si (start of the table).
-----
```

```
        mov  si, offset costtable ; si points to the table
        mov  bx,0          ; clear the index register
        mov  bl,[age]       ; get the age as an index value
        mov  al,[bx + si]   ; get the correct cost
        mov  [cost],al      ; set the cost field
```

Example 3: Building A List Of Addresses

Since a label (the name of a variable or instruction) is just the equivalent of its offset, you can build a list of addresses if you want. This is a very powerful concept that has many uses.

- Below we have a number of error messages. Each message is associated with a specific type of error.
- We have also built a list of pointers to these messages.

```
errormsg0    db    'Invalid input',13,10,'$'
errormsg1    db    'Number entered is too large',13,10,'$'
errormsg2    db    'Unrecognized command',13,10,'$'

msglist      dw    errormsg0, errormsg1, errormsg2
```

The declaration for msglist builds a 6 byte list with three words.

0	2	4
address of errormsg0	address of errormsg1	address of errormsg2

Let us assume that we have detected error number 2 and wish to print errormsg2.

We assume ax contains the value 0002 representing the specific error detected.

```
mov  bx,ax           ; bx contains 0002 which is the error number
add  bx,bx          ; double bx to 0004 since the addresses are words
mov  dx,[bx + msglist] ; dx will be loaded with the address of errormsg2
mov  ah,9            ; set the dos code to print a string
int  21h             ; print the string
```

Example 4: Jump Tables

Let's assume that you have some form of menu driven input. For example:

```
Enter the number representing the information you need: ___
1      list your grades
2      list your attendance record
3      list your homeworks submitted
9      exit
```

When the user types a number, you could do a series of compares and jumps to determine what number was entered.

An alternative is to use **a jump table**.

In this approach, the input value is used as an index into a table to select a specific routine *without* using compares. The following is an example of using a jump table.

```
;-----
; Program:    jumptabl
;
; Function:   This program provides an example of jump tables.
;             The program prompts the user for a number: 0 1 2 3 4
;             0 - 3 are inputs while 4 means end the program.
;             It then uses that number as an index into a jump table
;             to jump to a specific routine that processes the various
;             digits. For this sample, the processing involves just
;             writing out a message.
;
; Owner:      Dana Lasher
;
; Date:       09/30/1999 - original version
;-----
ideal                      ;use turbo ideal mode
model      small           ;64k code and 64k data
p8086                   ;only allow 8086 instructions
stack      256            ;reserve 256 bytes for the stack

dataseg
;-----
prompt db     'Enter a number 0 - 3 or 4 to terminate: ','$'
jumptbl dw     zero, one, two, three, exit      ;jump table

msg0   db     13,10,'zero' ,13,10,'$'          ;output message
msg1   db     13,10,'one'  ,13,10,'$'          ;output message
msg2   db     13,10,'two'  ,13,10,'$'          ;output message
msg3   db     13,10,'three',13,10,'$'          ;output message
;-----
```

```

codeseg                                ;start the code segment
;-----
start:                               ;
    mov      ax,@data                 ;establish addressability to the
    mov      ds,ax                   ;data segment for this program
;-----
;      Prompt the user for a number
;      and then read the number.
;-----
again:                               ;
    mov      dx, offset prompt      ;point to the prompt
    mov      ah,9                    ;set dos code
    int      21h                   ; to write the prompt
    ;
    mov      ah,1                    ;set dos code
    int      21h                   ; to read and echo a character
;-----
;      Use the number to locate the
;      corresponding routine and jump to it
;-----
        and     ax,000fh            ;clear all but the low digit
        add     ax,ax              ;double since table has words
        mov     si,ax              ;set si as the index
        jmp     [word ptr jmptbl + si];jump to address in jump table
                                ; that matches the digit entered
;-----
;      Routines corresponding to the
;      input values 0,1,2,3
;-----
zero:                               ;
    mov      dx, offset msg0       ;point to the message
    mov      ah,9                  ;set dos code
    int      21h                  ; to write a message
    jmp     again                ;go back to read another digit
one:                                ;
    mov      dx, offset msg1       ;point to the message
    mov      ah,9                  ;set dos code
    int      21h                  ; to write a message
    jmp     again                ;go back to read another digit
two:                                ;
    mov      dx, offset msg2       ;point to the message
    mov      ah,9                  ;set dos code
    int      21h                  ; to write a message
    jmp     again                ;go back to read another digit
three:                             ;
    mov      dx, offset msg3       ;point to the message
    mov      ah,9                  ;set dos code
    int      21h                  ; to write a message
    jmp     again                ;go back to read another digit
;-----
;      Exit routine corresponding to
;      the input value 4
;-----
exit:                               ;
    mov      ax,4c00h            ;dos code to terminate program
    int      21h                  ;return to dos
    end      start               ;mark the end of the source code
;-----

```

Example 5: Use Of Indirect Addressing With Index And Displacement

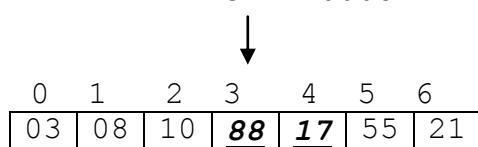
The advanced indirect addressing modes can be used to process lists.
In this example the code is sorting the list into increasing values.

The following situation exists.

- BX contains the address of a list of bytes indexed from 0 to 6. That list is at offset 1000h into the data segment.
- SI currently contains an index to element #3, which has the value 88. We have determined that we want to swap element #3 with element #4 as part of the sorting process.

BX = 1000h

SI = 0003h



These addressing modes will access the following values.

[bx] = The data is at location 1000h.

[bx + si] = The data is at location 1000h + 0003h = 1003h.

[bx + si + 1] = The data is at location 1000h + 0003h + 0001h = 1004h.

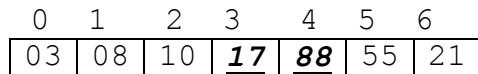
This code will swap element #3 (the value 88) and element #4 (value 17).

```
mov al,[bx+si]      ; al = 88
xchg al,[bx+si+1]   ; al = 17 ***and*** offset 1004 is set to 88
mov [bx+si],al       ; offset 1003 is set to 17
```

The list has been changed. The pointer registers have not been changed.

BX = 1000h

SI = 0003h



Some Coding Examples For You To Try. The Solutions Follow.

1. Convert this following C code to assembler. It goes through a string looking for the end of the string, which is marked with a value of 0.

```
char      buf[100];           // 100 character buffer
int       *ptr;              // a pointer

ptr = buf;                  // point to the string
while ( *ptr != 0 ) { ptr++; } // find the end of the string
```

2. Write the code that compares two ASCII character strings. The strings terminate with a null character. If they are identical then jump to an instruction labeled ‘same’. If they are different, including being different lengths, then jump to an instruction labeled ‘not_same’. Below is a sample of two strings to be compared.

```
line1    db      'this is a line of text',0
line2    db      'this is also a line of text',0
```

3. Write the code that copies an ASCII string of characters from one list to another list AND while copying replaces groups of multiple consecutive blanks with a single blank. That is, eliminate extra blanks. The list is terminated by a null character.

For example, if the input list is:

a		l		i		n		e				w		i		t		h						e		x		t		r		a						b		l		a		n		k		s		0h
---	--	---	--	---	--	---	--	---	--	--	--	---	--	---	--	---	--	---	--	--	--	--	--	---	--	---	--	---	--	---	--	---	--	--	--	--	--	---	--	---	--	---	--	---	--	---	--	---	--	----

Then the output would be:

a		l		i		n		e				w		i		t		h				e		x		t		r		a				b		l		a		n		k		s		0h
---	--	---	--	---	--	---	--	---	--	--	--	---	--	---	--	---	--	---	--	--	--	---	--	---	--	---	--	---	--	---	--	--	--	---	--	---	--	---	--	---	--	---	--	---	--	----

Here are the data definitions for these strings:

```
inline    db      'a line with extra     blanks',0
outline   db      50 dup (' ')
```

4. A Palindrome is a word or phrase that reads the same way forward or backward (ignoring blanks). Some examples are:

- a
- pup
- toot
- radar
- devil lived
- a man a plan a canal panama

Write the code to determine if a string is a palindrome. The string has no blanks or punctuation and all letters are lower case. The length of string is also defined as an unsigned byte.

```
pal      db      'amanaplanacanalpanama'
pallen  db      21
```

If the string is a palindrome goto the instruction labeled ‘ispal’ and if not goto the instruction labeled ‘notpal’.

Solutions

1. Convert this following C code to assembler. It just goes through a string looking for the end, which is assumed to be marked with a value of 0.

```
char  buf[100];           // 100 character buffer
int   *ptr;               // a pointer

ptr = buf;                // point to the string
while ( *ptr != 0 ) { ptr++; } // find the end of the string

buf    db     100 dup(?)      ; 100 character buffer

scan:  mov    bx, offset buf  ; point to the string using bx as the ptr
       cmp    [byte ptr bx],0  ; is the next character 0
       je     fin             ; yes, we are done
       inc    bx              ; no, move the pointer
       jmp    scan             ; and loop

fin:
```

2. Write the code that compares two ASCII character strings. The strings terminate with a null character. If they are identical then jump to an instruction labeled ‘same’. If they are different, including being different lengths, then jump to an instruction labeled ‘not_same’. Below is a sample of two strings to be compared.

Notes: The only way that the two strings can be equal is if all the characters up to and including the null character are identical. We terminate with an unequal condition if any pair of characters does not match. We terminate with an equal condition when the pair of matching characters is the null character.

```
line1    db      'this is a line of text',0
line2    db      'this is also a line of text',0

;-----
; set index for line1 and index for line2
; repeat
;   {
;     if char from line1 not equal to char from line2 then goto not_same
;     else if char from line is null then goto same
;     else advance index for line1 and index for line2
;   }
;-----
        mov    si,offset line1    ;si is ptr to line1
        mov    di,offset line2    ;di is ptr to line2
more_data:
        mov    al,[si]            ;get char from line1
        mov    ah,[di]            ;get char from line2
        cmp    al,ah              ;equal?
        jne    not_same          ;no, lines are not the same
        cmp    al,0                ;yes, are they the null character
        je     same               ;yes, the lines are the same
        inc    si                 ;no, move line1 index
        inc    di                 ;move line2 index
        jmp    more_data          ;continue testing
```

3. Write the code that copies an ASCII string of characters from one list to another list AND while copying replaces groups of multiple consecutive blanks with a single blank. That is, eliminate extra blanks. The list is terminated by a null character.

Notes: We copy a character from the input list to the output list. If this character is a blank, then we want to ignore any additional consecutive blanks in the input string. We do this by looking ahead in the input stream using *indirect addressing with displacement*. We need to assure we do move the final null character, and we stop after moving that null character.

```

inline db      'a line with extra blanks',0
outline db     50 dup (' ')
-----
;-----;
; set inline index
; set outline index
; repeat
;   {
;     char[outline index] = char[inline index]
;     if char[inline index] = blank
;       while char[inline index + 1] = blank
;         advance inline index
;     advance inline index
;     advance outline index
;   }
; until last character moved was the null character
;-----;

;-----;
; initialize pointers to the data
;-----;
    mov      si,offset inline ;si is ptr to input line
    mov      di,offset outline ;di is ptr to output line
;-----;
; move a character and test if it was a blank
;-----;
move_data:          ;
    mov      al,[si]           ;get char[inline index]
    mov      [di],al            ;put char[outline index]
    cmp      al,' '            ;was char moved a blank
    jne      advance           ;no, advance indices
;-----;
; remove multiple blanks
;-----;
remove:            ;
    cmp      [byte ptr si+1],' ' ;yes, is next character a blank
    jne      advance           ;no, done removing blanks
    inc      si                ;yes, advance inline index
    jmp      remove            ;and continue to remove blanks
;-----;
; advance the data pointers and
; loop if there is more data
;-----;
advance:           ;
    inc      si                ;advance inline index
    inc      di                ;advance outline index
    cmp      al,0               ;was char moved a null
    jne      move_data         ;no, loop
exit:

```

4. Determine if a string is a palindrome. Some notes:

- If the string is zero length we will call that a palindrome (arbitrary decision).
- A string length of 1 is a palindrome.
- We will divide the length by 2 and use that as a count of needed comparisons. We can truncate that division down since the middle character of an odd length string will be the same forward or backward. For example:
 string=toot len=4 count=4/2=2 string=pup len=3 count=3/2=1

```

pal      db      'amanaplanacanalpanama'
pallen  db      21

; If length = 0 or 1 then goto 'ispal'
; count = length / 2
; left_index = address of string
; right_index = address of string + length - 1
; while count > 0
;   {
;     if char[left_index] not equal char[right_index] then goto 'notpal'
;     inc left_index
;     dec right_index
;     count = count - 1
;   }
; goto 'ispal'
;-----
; init all registers
; calculate the right index location
;-----
      mov      ax,0          ; clear work register
      mov      cx,0          ; clear loop register
      mov      bl,2          ; get 2 for the divide
      mov      al,[pallen]    ; ax = length as unsigned word
      ;
      mov      si,offset pal ; set left_index
      mov      di,offset pal ; copy to right_index
      add      di,ax          ; right_index = left_index + length
      dec      di            ; right_index = left_index + length - 1
;-----
; calculate the number of compares needed
;-----
      cmp      ax,1          ; is length 0 or 1
      jbe      ispal         ; yes, we have a palindrome
      div      bl            ; no, al=count of compares needed
      mov      cl,al          ; move count to loop register
;-----
; compare current left and right character
; - if not the same then this is not a palindrome
; - if they are the same then update pointers and
;   loop again if there is more data
;-----
pal_loop:           ;
      mov      al,[si]        ; get left char
      cmp      al,[di]        ; left char : right char
      jne      notpal        ; if not equal then not palindrome
      inc      si            ; inc left index
      dec      di            ; dec right index
      loop    pal_loop       ; loop
      jmp      ispal         ; is palindrome when count exhausted

```

How do compilers use pointers ... C/C++ versus Assembler ... For Those Who Know C/C++

Function	C/C++	Assembler
Declaring a pointer	int *ptr; char *ptr;	SI, DI, BX, BP, SP are the hardware pointers
Setting a pointer	ptr = list; ptr = &list[0];	mov si, offset list
Using the pointer	x = *ptr; *ptr = 5; *ptr = *ptr + 5	mov ax, [si] or mov al,[si] mov [word ptr si], 5 or mov [byte ptr si], 5 add [word ptr si], 5 or add [byte ptr si], 5
Incrementing the pointer	ptr++;	add si,1 or add si,2 depending on data size

Replacing negative values example as generated by the C compiler

This is the same problem solved in C/C++. Along with the C/C++ code we show the actual assembler code generated by the compiler. We assume that the values in the list and count have previously been filled in correctly.

```

int list[6];           // list of 6 words = -3 7 100 -83 0 1000
int count=6;           // count of elements in the list = 6
int *index;            // a pointer used to process the list

index=&list[0];         // index points to the 1st element in the list

while (count > 0)       // go through all 6 elements
{
    if (*index < 0) *index=0; // set any negative values to zero
    count--;               // reduce the count of elements left
    index++;               // advance the index pointer
}

```

<u>C Statement</u>	<u>Assembler Code by C</u>	<u>Comments by DAL</u>
*** index=&list[0];	lea ax,list mov [index],ax	;index points ;to the list
*** while (count > 0) \$FC175:	cmp [count],0 jle \$FB176	;is count > 0 ;no, exit
*** {		
*** if (*index < 0)	mov bx,[index]	;put index in register
*index=0;	cmp [WORD PTR bx],0	;is next element < 0
	jge \$I177	;no, skip
	mov [WORD PTR bx],0	;yes, set element to 0
*** count--;	\$I177: dec [count]	;reduce count
*** index++;	add [index],2	;advance the index
*** }		
	jmp \$FC175	;repeat the loop
	\$FB176:	;done with the loop

Register Indirect with Index and Displacement in C/C++

Below is part of a C program that has an array of characters.
It establishes a pointer to the start of the array.
It then uses an index and constant to access one of the elements.

We provide the assembler code generated for the C statements so you can make the connection between your understanding of C and how the assembler and hardware implement C function.

```
int main()
{
    char list[20];                      // an array of 20 bytes
    char val;                            // a single byte
    char *base;                          // the pointer to the array
    int index1, index2;                  // index values

    base = &list[0];                    // base points to start of list
    index1 = 5;                         // set index value
    index2 = 3;                         // set index value

    val = *(base + index1 + 2);          // val = list[0 + 5 + 2]      = [7]
    val = *(base + index1 + index2 + 2); // val = list[0 + 5 + 3 + 2] = [10]
}
```

Note that with the program specifying **2** index values the compiler uses Indirect Addressing with Index and Displacement.

```
;|***  val = *(base + index1 + 2);

        mov     bx,[base]      ;bx = base
        mov     si,[index1]    ;si = index1
        mov     al,[bx+si+2]   ;index1 = 5 so ax = list[7]
        mov     [val],al       ;val
```

With the program specifying **3** index values, the compiler must perform arithmetic to get down to only using two indirect registers.

```
;|***  val = *(base + index1 + index2 + 2);

        mov     si,[index2]    ;si = index2
        add     si,[index1]    ;si = index2 + index1
        mov     bx,[base]      ;bx = base
        mov     al,[bx+si+2]   ;index1 = 3 & index2 = 5 so ax = list[10]
        mov     [val],al       ;val
```

Subroutines

A subroutine is a small self-contained piece of code that is called, executes and then returns.

There are three significant reasons for using subroutines.

1. Subroutines allow you to break a large complicated program into smaller more understandable pieces.
2. Once a program is broken into subroutines, then multiple coders can be assigned to work on the program.
3. Once a subroutine is completed, it can be re-used in the same program or other programs.

In general you will find that if subroutines are kept small and perform a single function that your overall development, debugging, and testing effort will be minimized.

Since the work associated with a program is going to be divided up among a number of subroutines, it will be required to establish a set of protocols, rules for communicating, between subroutines. These protocols must cover 2 topics.

1. How are values passed into the subroutine and how are results returned by the subroutine.
2. Who is responsible for saving and restoring registers that are modified by the subroutine.

Passing Parameters To The Subroutine

There are three ways to pass parameters to a subroutine and get back results.

1. *Pass parameters in registers.* This is very simple. It works well when you have very few parameters and they fit into registers.
2. *Share data through the use of global variables.* This is not a widely accepted technique in the computer science field. In large systems with many subroutines sharing global variables it is hard to tell which subroutine has changed a variable. This can make bug fixing very difficult. Global variables can be used in small well-controlled environments.
3. *Pass parameters on the stack.* This technique is used when the number or size of the parameters does not lend itself to passing them in registers. High-level languages use this technique extensively.

Saving And Restoring Registers

There is only one real set of hardware registers. In general, a subroutine does not know how the registers are being used by the program that called the subroutine. If the subroutine uses a register and changes its content then it may really screw up the calling program. To prevent that disaster, the standard convention is that a subroutine should save the contents of any register it uses and modifies and then restore its original value before returning to the calling program.

Which registers must be saved and restored is a design decision worked out between the coders of the calling program and the subroutine. However, the most common protocol has the subroutine save and restore any register it modifies, except those registers used for returning values to the calling program.

Overall Program Structure

After you break a program down into subroutines, you have two ways to handle the writing of the subroutines.

1. You can have the main program and all the subroutines contained in a single source file. This is okay as long as the total size of the program is relatively small and a single programmer is writing all the subroutines.
2. You can have separate source files for the main program and for each subroutine. This works well when the total effort gets large or if you have multiple programmers working on the same program and you need to share subroutines.

Code Characteristics

Code generated by a high-level language compiler or an assembler programmer can have one of these classifications

1. *Non-reusable* - This code is loaded into memory. It can then only be used once. If you need to use it again, a new copy must be loaded into memory. Programs become non-reusable when they have variables in the data segment that are initialized with a *ds* or *dw* directive and those variables are then modified by the code. After the code executes, those variables do not have their original value. This means the code cannot be executed a second time.
2. *Seriously reusable* - This code is loaded into memory. After it is completely done executing, it can be used again. Why is this important? If the code is a subroutine, such as a square root subroutine, it may be called many times. You would want it to work correctly each time it is called. Code with data in the data segment, that is modified, can be made serially reusable by having the code dynamically initialize variables with a *mov* instruction.
3. *Reentrant* - This code is loaded into memory. Multiple processes may use it simultaneously. It may be called multiple times before it finishes its first execution. Why would you want to do this? A good example is recursive code. To be reentrant, code may not have variables, that are modified, in the data segment. Since there is only one data segment, it is not possible to have multiple processes simultaneously use a variable. Instead, variables may be in registers or they may be dynamically created on the stack.
4. *Refreshable* - This is a history lesson that emphasizes that strange things can happen when you try to beat the system. Other than recursion, a major use envisioned for reentrant code was Operating System modules. For example, you only want one copy of the memory management code for an OS, but this code must be shared by all processes that need to allocate memory. However, you can only have one set of memory management variables, those that show what memory is in use and what memory is available. In other words, you need reentrant code that does share data segment global variables. In addition, you need to assure that the reentrant memory management code is not reentered while it is in the middle of updating those data segment global variables. Early Operating System developers accomplished this with a nefarious technical scheme. They made the reentrant code uninterruptible while it was updating its data segment global variables. Thus the code effectively appeared to be truly reentrant. How did they make the code uninterruptible? The two most common events that cause a process switch are an interrupt from a peripheral device or a timer interrupt. So their code masked off interrupts (stopped them from occurring) while the code was updating the data segment global variables. Now the problem occurred that had not been planned for. A hardware error would occur in main memory, some temporary memory failure. The error recovery part of the OS would look at the failing memory and see that there was a reentrant module loaded in that memory. Reentrant modules are not supposed to have data segment global variables, so the error recovery code knew it could safely reload the program, on top of itself, thus resetting any temporary failing memory locations. Unfortunately, if the code were actually this false version of reentrant code such as the memory management code, then all the data segment global variables that contain the state of memory would be destroyed when the program was reloaded. When this problem surfaced in early Operating Systems, the architects were forced to create a new type of code, called *refreshable*. *Refreshable* code can be reloaded on top of itself. In other words it really did not have any form of modified data segment global variables.

Creating Serial Reusable Code

If you write a subroutine then it must be at least serially reusable.

*If you call a subroutine more than once
then you must initialize all modifiable variables dynamically in the subroutine.
Do not rely on the directives 'db' or 'dw' for variables that are modified by the subroutine.*

Variables declared in assembler subroutines are like *static* variables in a HLL. They hold their value for the life of the whole program.

This means that you must dynamically initialize all variables that are modified by the subroutine. Do not rely on the directives *db* or *dw* for variables that are modified by the subroutine. Instead, write code in the subroutine to move the initial data values into the variables.

Failure to initialize the variables in the subroutine will mean that the variables will not have their expected value and this will cause unpredictable results.

***This version is not okay
It is non-reusable***

```
dataseg
val    dw  10      ;declare val=10

codeseg
subr:
        mov  [val],10  ;set val=10

repit:
        dec  [val]    ;val=val-1
        ja   repit   ;loop if val>0
```

***This version is okay
It is serially reusable***

```
dataseg
val    dw  ?       ;declare val

codeseg
subr:
        mov  [val],10  ;set val=10

repit:
        dec  [val]    ;val=val-1
        ja   repit   ;loop if val>0
```

See the section Common Bugs for more information.

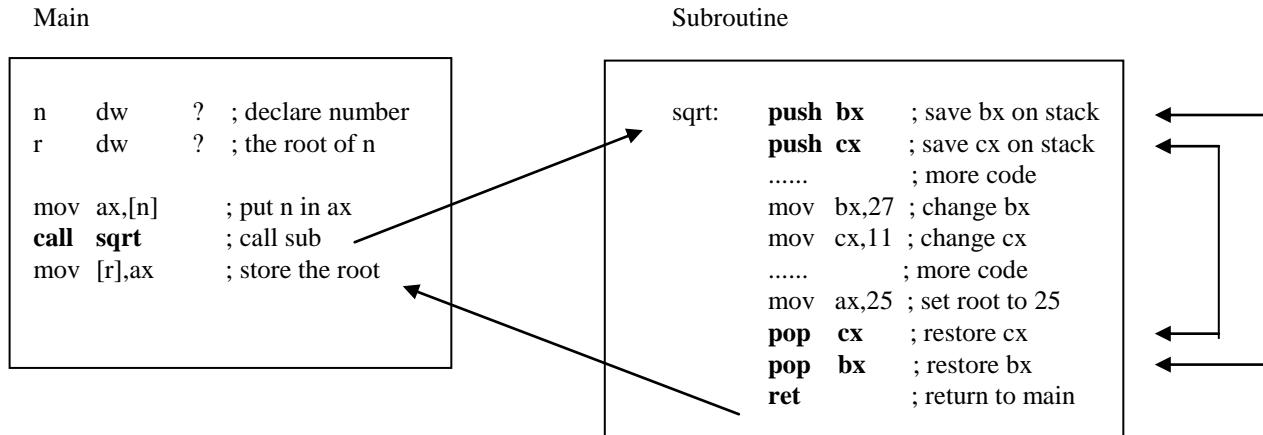
Conceptual Example

This is the conceptual structure of a main program and a subroutine that calculates the square root of a number.

It introduces four new instructions: **call** **ret** **push** **pop**

In this example we establish the following protocols between the main program and the subroutine.

- The main program passes the number whose root is desired to the subroutine in the ax register.
- The subroutine returns the calculated root in ax.
- The subroutine will save and restore, on the stack, all registers it modifies except ax.



Notes

- The **call** instruction takes us to the subroutine.
- The **ret** instruction takes us back to the main program.
- The **push** instruction saves the contents of a register on the stack. It is like *mov register to the stack*.
- The **pop** instruction restores the contents of a register from the stack. It is like *mov stack to the register*.
- The **order of pushes and pops is critical**; the first item pushed must be the last item popped.
- There must be **exactly the same number of pops as pushes**.
- (The two notes above are **really** important ... make sure you understand them)

Complete Example

Following is an example of real subroutine usage. The same problem is coded as a single file, and as separate files. Parameters are passed in registers.

Problem Statement (this is the exact same example used in the Indirect Addressing section)

Scan a list of signed word integers. Replace any negative values with zero.

For example if the list contains: -3, 7, 100, -83, 0, 1000
After processing the list will contain: 0, 7, 100, 0, 0, 1000

Input to the subroutine: cx = count of items in the list
 si = pointer to the offset to the list

The subroutine is to save and restore all registers it modifies.

```

;-----  

;   program:      SINGLE is a program for list processing  

;   purpose:       It shows a main program and subroutine in a single file  

;   processing:    go through a list and replace negative values with zero  

;-----  

        ideal          ;use turbo ideal mode  

        model         small ;64k code and 64k data  

        p8086          ;only allow 8086 instructions  

        stack         256 ;reserve 256 bytes for the stack  

;-----  

        databaseg      ;start the data segment  

;-----  

count dw      6 ;count of items in the list  

list dw      -3,7,100,-83,0,1000 ;list of words  

;-----  

        codeseg      ;start the code segment  

;-----  

start:          ;  

        mov      ax,@data ;establish addressability to the  

        mov      ds,ax  ;data segment for this program  

;-----  

; call list processing subroutine  

;-----  

        mov      cx,[count] ;place count in cx  

        mov      si, offset list ;place address of list in si  

        call     set ;call the subroutine  

;-----  

; terminate program execution  

;-----  

exit:           ;  

        mov      ax,4c00h ;set dos code to terminate program  

        int      21h ;return to dos  

;-----  

;-----  

;   list processing subroutine  

;   input: cx=count      si=pointer to the list  

;-----  

        databaseg      ;start the data segment  

;-----  

zero dw      0 ;constant zero used by the subroutine  

;-----  

        codeseg      ;start the code segment  

;-----  

set:            ;  

        push     ax ;save registers  

        push     si ;save registers  

        push     cx ;save registers  

set01:          ;  

        cmp      cx,0 ;test for length of zero  

        je      set04 ;yes, return  

        mov      ax,[zero] ;no, set up ax with zero  

set02:          ;  

        cmp      [word ptr si],0 ;test the next value  

        jge     set03 ;if >= 0 then skip  

        mov      [si],ax ;if < 0 then replace with zero  

set03:          ;  

        add      si,2 ;move pointer to next value  

        loop    set02 ;loop if more items in the list  

set04:          ;  

        pop      cx ;restore registers  

        pop      si ;restore registers  

        pop      ax ;restore registers  

        ret ;return  

;-----  

        end      start ;end of the source code  

;-----  


```

```

;-----  

;   program:      MAIN is a program for list processing  

;   purpose:       It shows a main program and subroutine in separate files  

;   processing:    go through a list and replace negative values with zero  

;-----  

        ideal                  ;use turbo ideal mode  

        model     small          ;64k code and 64k data  

        p8086                ;only allow 8086 instructions  

        extrn    set:proc        ;declare external procedure  

        stack     256            ;reserve 256 bytes for the stack  

;-----  

;  

        databaseg             ;start the data segment  

;-----  

count    dw      6           ;count of items in the list  

list     dw      -3,7,100,-83,0,1000 ;list of words  

;-----  

;  

        codeseg               ;start the code segment  

;-----  

start:                                ;  

        mov      ax,@data        ;establish addressability to the  

        mov      ds,ax           ;data segment for this program  

;-----  

; call list processing subroutine  

; input: cx=count of items in the list  

;         si=pointer to the list  

;-----  

        mov      cx,[count]      ;place count in cx  

        mov      si, offset list ;place address of list in si  

        call    set              ;call the subroutine  

;-----  

; terminate program execution  

;-----  

exit:                                ;  

        mov      ax,4c00h        ;set dos code to terminate  

        int      21h             ;return to dos  

        end      start           ;end of source code  

;-----  


```

Notes:

- The **extrn** statement declares the label **set** as a procedure that is external to this file. The Linker program will resolve this external reference.
- The **extrn** statement is exactly the same in MASM and TASM.

```

;-----  

;   program: SUB is a list processing subroutine  

;           This is the model for a subroutine created in a file that  

;           is separate from the main program file  

;   input:    cx=count of items in the list and si=pointer to the list  

;-----  

        ideal                      ;use turbo ideal mode  

        model      small           ;64k code and 64k data  

        p8086                  ;only allow 8086 instructions  

        public     set              ;define set to other modules  

;-----  

;  

        databaseg                ;start the data segment  

;-----  

zero      dw      0             ;constant used by the subroutine  

;-----  

;  

        codeseg                 ;start the code segment  

;-----  

set:  

        push      ax              ;save registers  

        push      si              ;save registers  

        push      cx              ;save registers  

set01:  

        cmp       cx,0            ;test for length of zero  

        je       set04            ;yes, return  

        mov       ax,[zero]         ;no, set up ax with zero  

set02:  

        cmp       [word ptr si],0  ;test the current value  

        jge      set03            ;if >= 0 then skip  

        mov       [si],ax           ;if < 0 then replace with zero  

set03:  

        add       si,2             ;move pointer to next value  

        loop      set02            ;loop if more items in the list  

set04:  

        pop      cx              ;restore registers  

        pop      si              ;restore registers  

        pop      ax              ;restore registers  

        ret                  ;return  

        end                  ;end of the source code  

;-----  


```

Notes:

- The **public** statement indicates that the label **set** will be used by a program that was assembled in a separate file. The **public** statement is exactly the same in MASM and TASM.
- The subroutine does not define a stack. The main program does it.
- The subroutine does not load the ds register. The main program does it.
- The end statement does not have an operand.
- These commands create the program called main.exe.

TASM	MASM
• tasm /zi /l main	ml /c /Zi /Fl main.asm
• tasm /zi /l sub	ml /c /Zi /Fl sub.asm
• tlink /v main sub	link /CO main.obj sub.obj

How to structure multiple subroutines in one file along with a main program

It is okay to have multiple and nested subroutines where one subroutine calls another subroutine. Follow the same basic rules. Just place the code sequentially in memory. Each subroutine may have its data declared with its code for clarity. However, the assembler will combine all the data into a single segment. This means that all labels and variables must be unique.

```
;-----  
;      Main header  
;-----  
        ideal          ;the main  
        model small    ;program  
        p8086          ;has these  
        stack 256       ;directives  
  
        dataseg         ;data for main  
varx      dw 0  
  
        codeseg         ;code for main  
start:  
        mov  ax,@data  ;the main program  
        mov  ds,ax      ;initializes the ds register  
  
        call sub1       ;call subroutine 1  
  
exit:  
        mov ax,4c00h    ;dos termination code  
        int 21h         ;terminate  
  
;-----  
;      Subroutine 1 header  
;-----  
        dataseg         ;data for sub1  
vary      dw 0  
  
        codeseg         ;code for sub1  
sub1:  
        push ax         ;save registers  
        call sub2       ;call nested subroutine 2  
        pop  ax         ;restore registers  
        ret             ;return  
  
;-----  
;      Subroutine 2 header  
;-----  
        dataseg         ;data for sub2  
varz      dw 0  
  
        codeseg         ;code for sub2  
sub2:  
        push ax         ;save registers  
        .....          ;perform some calculations  
        pop  ax         ;restore registers  
        ret             ;return  
  
end start      ;mark the end of the source program
```

How to structure multiple subroutines in one file that is separate from the main program file

```
ideal
model      small
p8086
public    sub1
public    sub2

;-----
;       Subroutine 1 header
;-----
        databaseg      ;data for sub1
vary      dw 0

        codeseg         ;code for sub1
sub1:
        push ax          ;save registers
        call sub2        ;call nested subroutine 2
        pop  ax          ;restore registers
        ret              ;return

;-----
;       Subroutine 2 header
;-----
        databaseg      ;data for sub2
varz      dw 0

        codeseg         ;code for sub2
sub2:
        push ax          ;save registers
        .....           ;perform some calculations
        pop  ax          ;restore registers
        ret              ;return

end        ;mark the end of the source program
```


Linking High Level Languages And Assembler

It is often desirable to mix an assembler subroutine with a program written in a High Level Language (HLL).

For example:

- you have a special hardware device controller such as a robot arm
- you need to optimize performance of a section of code

In all cases, you will need to have the specific linkage conventions required by the HLL. They will vary with each HLL and may vary with the specific vendor compiler.

We will study one specific case, C /C++ language calls to assembler.

Passing Parameters To The Subroutine

There are multiple potential ways to pass parameters between routines.

- The simplest is to pass parameters in registers. The disadvantage is that you only have a few registers and this would limit the amount of information that could be passed.
- You could create and share global variables. This is not an accepted method because if a bug causes a shared global variable to be corrupted it is hard to determine which routine was at fault.
- You can pass parameters on the stack. This is the most widely used method.

C / C++ pass parameters on the stack. You may pass either the *value* of the variable or the *address* of the variable. For byte or word size variables the programmer makes that decision. If you pass the value, then the subroutine can use it as a read only constant. If you pass the address, then the subroutine can both read and modify the variable.

- To pass the value of a variable just pass its name: `sqrt(x)`
- To pass the address of the variable, the name is prefixed with an & symbol: `sqrt(&x)`

For strings or arrays you do not want to try to copy the contents of the whole string or array onto the stack. Therefore, you always want to pass the address of the string or array. The C / C++ language always interprets the *name* of a string or array as its *address*. This means that you do not need to prefix the name with an & symbol. The call `addlist(array)` passes the address of array and not the whole contents of the array.

We will now go through a detailed. In this example we have a C/C++ program that calls an assembler subroutine. It passes to the subroutine a single character *c* and an integer count *n*. The subroutine is to write to the standard output device the character *c*, *n* times.

C Main Program

```
// Declare the subroutine asmprint as external
// asmprint will return an integer
// The input to asmprint is a character c and a count value n
// asmprint will output the character c, n times

extern int asmprint(char c, int n);

int main()
{
// declare the return code
int rc;

// call the subroutine with c=x  count=5
rc=asmprint('x', 5);

// exit the program
return(0);
}
```

Asmprint Subroutine

```
// include the input and output library functions
#include <stdio.h>

int asmprint(char c, int n)
{

// print the character n times
while (n>0)          // while n is greater than zero
{
    printf("%c", c);   // print the character c
    n--;               // reduce n by 1
}

return (0);
}
```

This is the code generated by the compiler that calls the subroutine.

```
; |*** // call the subroutine with c=x  count=5
; |***      rc=asmprint('x', 5);

    mov     ax,5
    push    ax
    mov     al,120
    push    ax
    call    _asmprint
    add    sp,4
    mov    [rc],ax
```

Notes:

1. The variables are pushed onto the stack in reverse order from the calling sequence.
This means that the count is pushed onto the stack first, and this is followed by the character.
2. Both variables are passed as *values* in this example.
The character is only 1 byte, however, it is placed into the low byte of AX and AX is pushed onto the stack.
The system always will push words onto the stack.
3. After all parameters are pushed onto the stack, SP points to the last parameter pushed.
The call instruction will then push the return address onto the stack.
When we get to the subroutine, SP will be pointing to the return address.
4. The procedure name is preceded by an underscore.
5. After the subroutine returns, the C program removes the two variables from the stack.
It does this by adding 4 to the stack pointer (2 for each variable it pushed onto the stack).
It uses this technique *instead* of popping off the variables because it does not actually need their values; it only needs to restore the stack to its original condition.

The Assembler Source For The Subroutine

```

; Program: asmprint
; Function: display an input character (c) n times
; Owner: Dana Lasher
;
; Input: the subroutine is called by asmprint(c,n);
;        the inputs are taken from the stack
;        n was pushed first and is a word
;        c was pushed second and is stored in the low byte of a word
;        ( since c and n were pushed as words they will appear
;          byte reversed on the stack)
;
;                                |           | <--stack
;                                +6      | nn nn      |
;                                +4      | cc --      |
;                                +2      | ret. addr |
; active bp as copied from sp --> +0      | saved bp |
;
; Notes: Microsoft HLLs expect that these registers will be saved
; if they are modified by the subroutine: BP, SI, DI, SS, DS
;
; Date: changes
; 07/21/96 original version
-----
;           ideal                      ;use turbo ideal mode
;           model         small          ;64k code and 64k data
;           p8086          ;only allow 8086 instructions
;           public        _asmprint     ;allow access to _asmprint
;-----


;-----          datasetg          ;start the data segment
;-----


;-----          codeseg          ;start the code segment
;-----


_asmprint:
    push    bp          ;save caller's bp
    mov     bp,sp       ;init bp so we can get the arguments
    push    si          ;save si just to show how its done
    push    di          ;save di just to show how its done
    mov     dx,[bp+4]   ;get character c (into dl) dx=0078
    mov     cx,[bp+6]   ;get count n (into cx) cx=0005
output:
    mov     ah,2         ;get DOS code to write a char
    int     21h         ;write a character from dl
    loop   output       ;go through the loop n times
exit:
    pop    di          ;restore di
    pop    si          ;restore si
    pop    bp          ;restore bp
    mov    ax,0         ;set return code
    ret              ;return
    end
;
```

Notes On The Assembler Source For The Subroutine

1. The ***first thing*** we do in the subroutine is set up the bp register to point to the parameters passed on the stack.

**These 2 instructions must be the first instructions executed by your subroutine.
If they are not, then your code will not work and will exhibit bizarre behavior.**

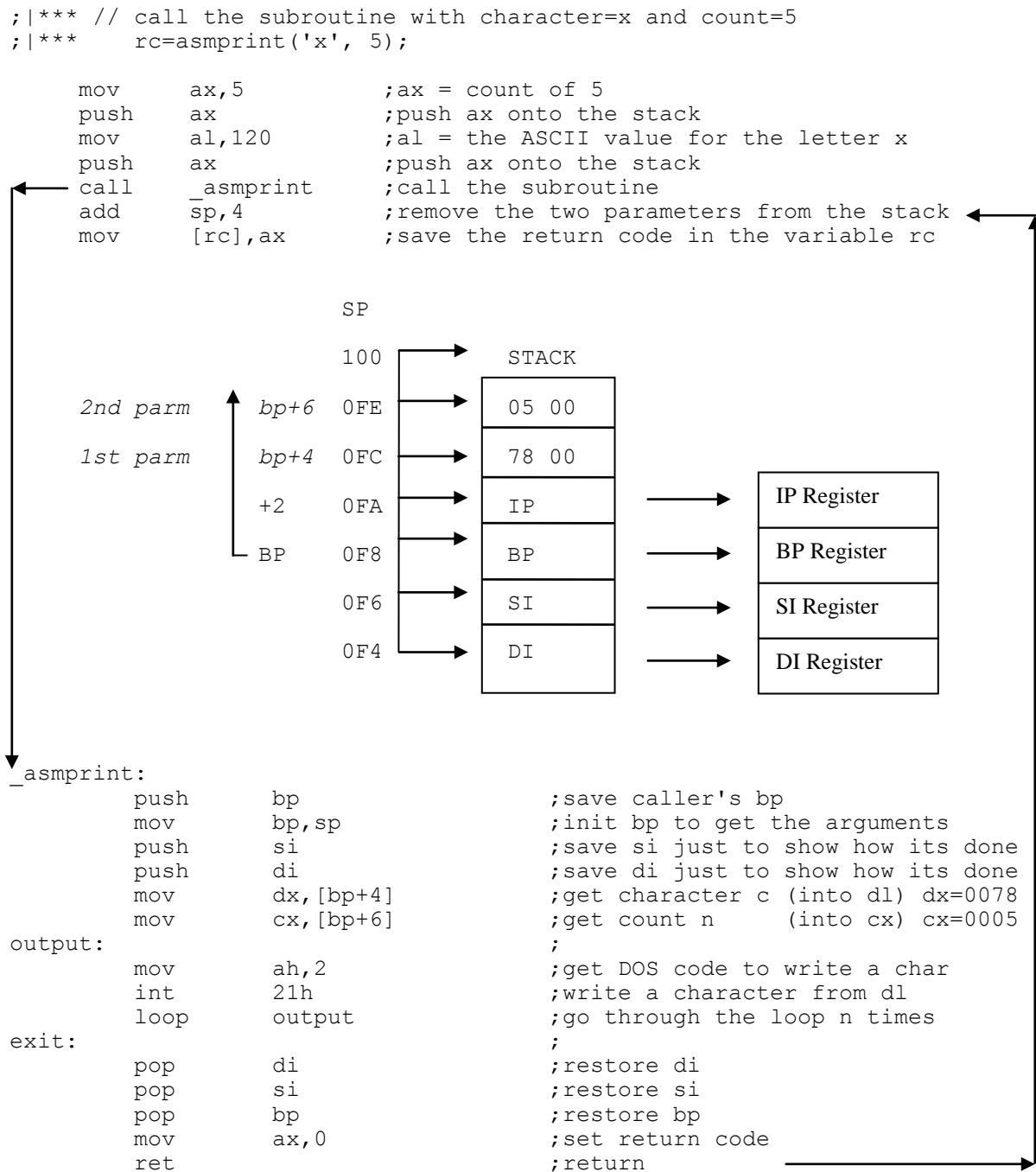
```
push bp          ; saves the caller's bp  
mov bp,sp       ; loads bp from sp
```

There are only 4 registers that can be used as pointers: si, di, bx, bp

- The si, di, bx registers assume that the data being referenced is in the data segment. The subroutine's data, however, is on the stack, which is in the stack segment.
 - The bp register assumes that the data being referenced is in the stack segment. The bp register is the register available to the programmer to get the parameters off the stack. Once we save bp we can then load it with the current value of sp. It will then remain constant for the duration of the subroutine.
2. C / C++ expects that these registers will be saved if they are modified by the subroutine:
 - bp, si, di, ss, ds
 - Any other modified registers do NOT need to be saved
 3. To pass a return code back to the calling program, it is loaded into the ax register before returning.

A Pictorial View Of What Happens

(we assume a stack size of 256 bytes which is hex 100, so the initial value in sp is 0100h)



Note. The calling sequence assures that 1st parameter is always at [bp+4], the 2nd parameter at [bp+6], the 3rd parameter at [bp+8], etc.

$\begin{matrix} \text{bp+4} & \text{bp+6} & \text{bp+8} & \text{bp+10} & \dots \end{matrix}$
 sub (parm1, parm2, parm3, parm4, ...

Compilers And Their Data

A related aspect of how compilers work is where compilers create data and how they create serially reusable subroutines.

We are used to creating data in the data segment. Compilers use both the data segment and the stack segment depending on the characteristics of the data.

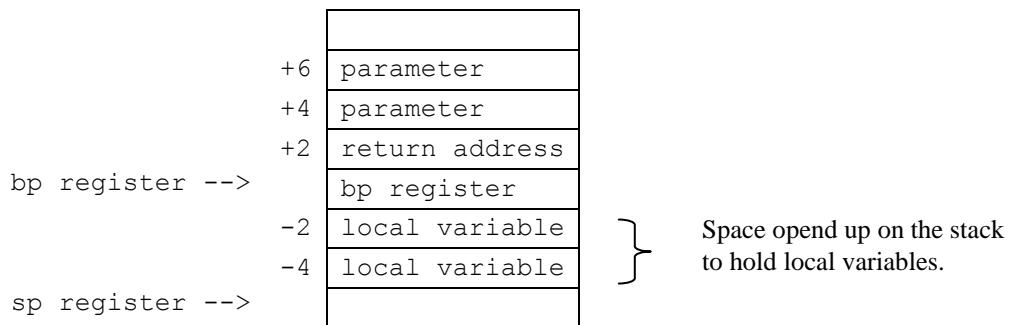
This is how most compilers generate data.

- Global or Static variables
 - Declared in the data segment.
 - They are only created once.
 - If they are given an initial value then they only initialized once, at the start of the program.
- Local variables
 - Declared on the stack.
 - They are dynamically created, using code at execution time, every time the routine is called.
 - If they are given an initial value then they initialized each time the routine is called.
 - The name given to these variables created on the stack is an ***activation record***. This process is required if the code is going to support ***recursion***.

How are variables created on the stack?

We will see that the called routine moves the stack pointer to open up as many words as necessary to hold the data.
In the example below it subtracts 4 from the stack pointer to open up space for two words.

- We have already seen that the passed parameters are referenced relative to the bp register and that they have **positive** offsets such as [bp+4].
- We will now see that the local variables are also referenced relative to the bp register and that they have **negative** offsets such as [bp-2].



An Example That Shows The Different Ways Compilers ...

1. generate data
2. initialize data
3. access data

```
*****  
// main:      Declares a global variable named count  
//             Sets count to 7  
//             Calls asmprint to print a character  
//                     (passes the character and a value)  
//  
// asmprint:   Declares two local variables, i and j  
//             i is initialized to 0  
//             j is uninitialized  
//             Sets j to the value passed plus main's global variable  
//             Print the character j times  
*****  
  
#include <stdio.h>  
  
// declare a global variable  
static int count;  
  
int main()  
{  
  
    // set the global variable  
    count=7;  
  
    // call the subroutine  
    asmprint('x', 5);  
  
    // exit the program  
    return(0);  
}  
  
  
asmprint(char c, int n)  
{  
    // local initialized variable  
    int i = 0;  
  
    // local uninitialized variable  
    int j;  
  
    // j is set to the sum of the number passed plus main's global variable  
    j = n + count;  
  
    // print the character j times  
    while (i<j)  
    {  
        printf("%c", c);  
        i++;  
    }  
  
    return;  
}
```

The Assembler Code ... Local Variable Allocation Is In *Bold Italic Type*.

```

.data
$S163_count dw ? ; main's global variable created in the Data Segment

.code
_main
; set the global variable count = 7
mov     [$S163_count],7

; call the subroutine...asmprint('x', 5)
mov     ax,5
push    ax
mov     ax,120
push    ax
call    _asmprint
add    sp,4

; exit the program...return(0)
sub    ax,ax
ret

; asmprint(char c, int n) subroutine
_asmprint: push   bp           ; save bp
            mov    bp,sp        ; set bp
            sub   sp,4         ; create space    bp → +0
            → -2
            → -4

; Subtracting 4 from sp opens up two words on the stack.
; These two words are used for the local variables i and j.
; At the time the subtraction is done, bp equals sp.
; So, these local variables have negative offsets to the
; fixed value in the bp register. i = [bp-2] j = [bp-4]

; declare and init i = 0
mov    word ptr [bp-2],0    ;i = 0

; j=n+count
mov    ax,[S163_count]      ;ax = count
add    ax,[bp+6]            ;ax = ax + n = count + n
mov    [bp-4],ax            ;j = count + n

; print the character j times
; while (i<j)
$fC173: mov    ax,[bp-2]      ;ax = i
            cmp    [bp-4],ax      ;j compared to i
            jle    $fb174          ;done if j less than or equals i

            printf("%c", c)       ;print if j > i

            inc    word ptr [bp-2]  ;i = i + 1
            jmp    $fc173          ;go back and test while condition

; return
$fb174: mov    sp,bp          ;undo declaring the local variables
            pop    bp              ;restore bp
            ret                  ;return

```

Stack	
+6	05 00
+4	'x' 00
+2	return
0	bp
-2	i
-4	j

Show me the magic incantations that

... convert symbolic assembler into machine code

... convert machine code back into assembler

This section describes the machine code, also called the Instruction Set Architecture (ISA), for the 8086.

This is the boundary between hardware and software.

All software that is written in high level languages such as C++ or Java and all software that is written in Assembler language *must be translated into machine code in order to be executed by the CPU*. The only language that a CPU understands is machine code.

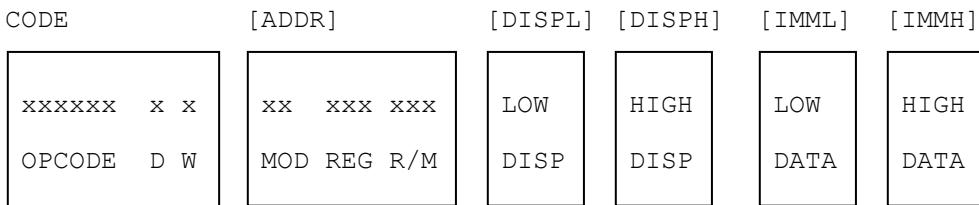
Of course, you will not write programs in machine code. You will write code in some symbolic language and a compiler or assembler will generate the machine code. This raises the question "*Why should I care about machine code?*". The answer to that question depends heavily on how you wish to use your Computer Science knowledge. If your goal is to write WEB applets in Java or spreadsheets in Excel then you will not have much interest in machine code. However, remember back to the first day of class when we discussed the purpose of this course. The course covers computer architectural topics required by professional software developers. It explains what happens underneath High Level Languages such as C++ and Java. To many in the Computer Science field, an understanding of machine code is absolutely fundamental.

Here are tasks that require a level of understanding of machine code and the ISA for a machine.

- All parts of the industry that design new computers. This can be a computer architect or hardware engineer. It can also be a software developer who works with hardware engineers to help bring up a new computer. As early versions of hardware start to function someone has to write basic code for the new computer. In this early life of a computer there is no assembler or compiler available and the first instructions written are in the machine code for the new computer.
- A software developer that writes Assemblers or Compilers.
- A technical system's support programmer. This is the person who is called upon to debug the nasty intermittent bugs that plague your company's product. This task often involves looking at raw bits, both data and machine instructions, in memory.
- A system's analyst that may need to evaluate the relative performance of different computing systems. This can include hardware performance and also the performance of the language compilers available for the system.
- An application programmer who strives for developing efficient programs and wants to be able to look at the code generated by your compiler to evaluate its quality.

Table-1 8086 Machine Code Instruction Formats

- Machine instructions are variable in length ranging from 1 to 6 bytes.
- Bytes in brackets [] are optional.
- If an optional field is not needed then it is omitted and all other fields are just moved to the left.
- CODE byte specifies the machine opcode, the size of the data, and the direction of the data movement.
- ADDR byte specifies the addressing modes for the *source* and *destination* data for the instruction.
- DISPL and DISPH specify the memory address if a memory operand is used.
- IMML and IMMH specify the value of any immediate data that is needed.



CODE Byte

OPCODE Six bits that define the opcode for the instruction.

D Direction of data movement. Only needed for 2 operand instructions (register & memory) or (register & register). See OPERANDS 5 and 6.
 d = 0 REG field in ADDR is the source operand
 d = 1 REG field in ADDR is the destination operand

W Data size

w = 0 means data is byte size w = 1 means data is word size

ADDR Byte (Simplified to show register direct and memory direct addressing only)

OPERAND (S)	MOD	REG	R/M	DISP	IMM
1. single register	11	special code	register		
2. single memory	00	special code	110	mem addr	
3. immediate to register	11	special code	register		data
4. immediate to memory	00	special code	110	mem addr	data
5. register and memory	00	register	110	mem addr	
6. register and register	11	register_1	register_2		

Register specifications for REG or R/M fields

Word	Byte	Segment Reg (sr)	Note
000 AX	000 AL	ES 00	When you see the term accumulator
001 CX	001 CL	CS 01	- it means ax register for word data
010 DX	010 DL	SS 10	- it means al register for byte data
011 BX	011 BL	DS 11	
100 SP	100 AH		
101 BP	101 CH		
110 SI	110 DH		
111 DI	111 BH		

Table-2 8086 Instruction Encoding (Assembler to Machine)

w	0	data is byte
	1	data is word
d	0	reg field specifies source operand
	1	reg field specifies destination operand
v	0	shift or rotate is 1 bit
	1	shift or rotate is specified in cl register

Each box represents 1 byte

mov

mov reg/mem to reg/mem
 mov immed to mem
 mov immed to reg
 mov mem to accumulator
 mov accumulator to mem
 mov reg/mem to seg reg
 mov seg reg to reg/mem

1000 10dw	mod reg r/m	disp-lo	disp-hi		
1100 011w	mod 000 r/m	disp-lo	disp-hi	data-lo	data-hi
1011 w reg		data-lo	data-hi		
1010 000w		disp-lo	disp-hi		
1010 001w		disp-lo	disp-hi		
1000 1110	mod 0 sr r/m	disp-lo	disp-hi		
1000 1100	mod 0 sr r/m	disp-lo	disp-hi		

push

push mem
 push reg
 push seg reg

1111 1111	mod 110 r/m	disp-lo	disp-hi		
0101 0 reg					
000 sr 110					

pop

pop mem
 pop reg
 pop seg reg

1000 1111	mod 000 r/m	disp-lo	disp-hi		
0101 1 reg					
000 sr 111					

add

add reg/mem to reg/mem
 add immed to reg/mem
 add immed to accumulator

0000 00dw	mod reg r/m	disp-lo	disp-hi		
1000 000w	mod 000 r/m	disp-lo	disp-hi	data-lo	data-hi
0000 010w		data-lo	data-hi		

adc

adc reg/mem to reg/mem
 adc immed to reg/mem
 adc immed to accumulator

0001 00dw	mod reg r/m	disp-lo	disp-hi		
1000 000w	mod 010 r/m	disp-lo	disp-hi	data-lo	data-hi
0001 010w		data-lo	data-hi		

inc

inc 8 bit reg / any size mem
 inc 16 bit reg

1111 111w	mod 000 r/m	disp-lo	disp-hi		
0100 0 reg					

sub

sub reg/mem from reg/mem
 sub immed from reg/mem
 sub immed from accumulator

0010 10dw	mod reg r/m	disp-lo	disp-hi		
1000 000w	mod 101 r/m	disp-lo	disp-hi	data-lo	data-hi
0010 110w		data-lo	data-hi		

sbb

sbb reg/mem from reg/mem
 sbb immed from reg/mem
 sbb immed from accumulator

0001 10dw	mod reg r/m	disp-lo	disp-hi		
1000 000w	mod 011 r/m	disp-lo	disp-hi	data-lo	data-hi
0001 110w		data-lo	data-hi		

dec

dec 8 bit reg / any size mem
dec 16 bit reg

1111 111w	mod 001 r/m	disp-lo	disp-hi
0100 1 reg			

neg change sign

1111 011w	mod 011 r/m	disp-lo	disp-hi
-----------	-------------	---------	---------

cmp

cmp reg/mem to reg/mem
cmp reg/mem to immed
cmp accumulator to immed

0011 10dw	mod reg r/m	disp-lo	disp-hi
1000 000w	mod 111 r/m	disp-lo	disp-hi
0011 110w	data-lo	data-hi	

multiply and divide

mul multiply (unsigned)
imul integer multiply (signed)
div divide (unsigned)
idiv integer divide (signed)

1111 011w	mod 100 r/m	disp-lo	disp-hi
1111 011w	mod 101 r/m	disp-lo	disp-hi
1111 011w	mod 110 r/m	disp-lo	disp-hi
1111 011w	mod 111 r/m	disp-lo	disp-hi

sign extend

cbw convert byte to word
 cwd convert word to double

1001 1000
1001 1001

logic

not invert
shl/sal shift left
shr shift logical right
sar shift arithmetic right
rol rotate left
ror rotate right
rcl rotate thru carry left
rcr rotate thru carry right

1111 011w	mod 010 r/m	disp-lo	disp-hi
1101 00vw	mod 100 r/m	disp-lo	disp-hi
1101 00vw	mod 101 r/m	disp-lo	disp-hi
1101 00vw	mod 111 r/m	disp-lo	disp-hi
1101 00vw	mod 000 r/m	disp-lo	disp-hi
1101 00vw	mod 001 r/m	disp-lo	disp-hi
1101 00vw	mod 010 r/m	disp-lo	disp-hi
1101 00vw	mod 011 r/m	disp-lo	disp-hi

and

and reg/mem to reg/mem
and immed to reg/mem
and immed to accumulator

0010 00dw	mod reg r/m	disp-lo	disp-hi
1000 000w	mod 100 r/m	disp-lo	disp-hi
0010 010w	data-lo	data-hi	

test

test reg/mem and reg
test immed and reg/mem
test immed and accumulator

1000 010w	mod reg r/m	disp-lo	disp-hi
1111 011w	mod 000 r/m	disp-lo	disp-hi
1010 100w	data-lo	data-hi	

or

or reg/mem to reg/mem
or immediate to reg/mem
or immed to accumulator

0000 10dw	mod reg r/m	disp-lo	disp-hi
1000 000w	mod 001 r/m	disp-lo	disp-hi
0000 110w	data-lo	data-hi	

xor

xor reg/mem to reg/mem
xor immed to reg/mem
xor immed to accumulator

0011 00dw	mod reg r/m	disp-lo	disp-hi
1000 000w	mod 110 r/m	disp-lo	disp-hi
0011 010w	data-lo	data-hi	

call

call within seg

1110 1000	ip-inc-lo	ip-inc-hi
-----------	-----------	-----------

jmp

jmp within seg

1110 1001	ip-inc-lo	ip-inc-hi
-----------	-----------	-----------

ret

ret within seg

1100 0011

jxx

ja	jump above
jae	jump above or equal
jb	jump below
jbe	jump below or equal
je	jump equal
jg	jump greater
jge	jump greater or equal
jl	jump less
jle	jump less or equal
jne	jump not equal
jno	jump no overflow
jns	jump not sign
jo	jump overflow
jpe	jump parity even
jpo	jump parity odd
js	jump sign
loop	loop cx times

0111 0111	ip-inc-8
0111 0011	ip-inc-8
0111 0010	ip-inc-8
0111 0110	ip-inc-8
0111 0100	ip-inc-8
0111 1111	ip-inc-8
0111 1101	ip-inc-8
0111 1100	ip-inc-8
0111 1110	ip-inc-8
0111 0101	ip-inc-8
0111 0001	ip-inc-8
0111 1001	ip-inc-8
0111 0000	ip-inc-8
0111 1010	ip-inc-8
0111 1011	ip-inc-8
0111 1000	ip-inc-8
1110 0010	ip-inc-8

Special notes on the jumps

- The field *ip-inc-8* is an 8 bit signed value that is added to the instruction pointer if the jump is taken to form the target address.
- The instruction pointer (ip) points to the address of the instruction following the jump. This means the instruction *here: jne here* will generate the machine code: 75 FE
- The conditional jumps are limited to a range of -128 bytes backwards and +127 bytes forward.
- The unconditional jump (jmp) has a 16 bit signed value (*ip-inc-lo* and *ip-inc-hi*) that is added to the instruction pointer when the jump is taken to form the target address. This gives the jmp instruction a range of -32768 bytes backwards and +32767 bytes forward.

string manipulators

rep	repeat
movs	move byte/word
cmps	compare byte/word
scas	scan byte/word
lod	load to al/ax
stds	store from al/ax

1111 001z
1010 010w
1010 011w
1010 111w
1010 110w
1010 101w



z=0 means repeat loop while zero flag (ZF) is clear (ZF=0)
z=1 means repeat loop while zero flag (ZF) is set (ZF=1)

Table-3 8086 Instruction Decoding (Machine to Assembler)

push seg reg	000 sr 110				
pop seg reg	000 sr 111				
add reg/mem to reg/mem	0000 00dw	mod reg r/m	disp-lo	disp-hi	
add immed to accumulator	0000 010w	data-lo	data-hi		
or reg/mem to reg/mem	0000 10dw	mod reg r/m	disp-lo	disp-hi	
or immed to accumulator	0000 110w	data-lo	data-hi		
adc reg/mem to reg/mem	0001 00dw	mod reg r/m	disp-lo	disp-hi	
adc immed to accumulator	0001 010w	data-lo	data-hi		
sbb reg/mem from reg/mem	0001 10dw	mod reg r/m	disp-lo	disp-hi	
sbb immed from accumulator	0001 110w	data-lo	data-hi		
and reg/mem to reg/mem	0010 00dw	mod reg r/m	disp-lo	disp-hi	
and immed to accumulator	0010 010w	data-lo	data-hi		
sub reg/mem from reg/mem	0010 10dw	mod reg r/m	disp-lo	disp-hi	
sub immed from accumulator	0010 110w	data-lo	data-hi		
xor reg/mem to reg/mem	0011 00dw	mod reg r/m	disp-lo	disp-hi	
xor immed to accumulator	0011 010w	data-lo	data-hi		
cmp reg/mem to reg/mem	0011 10dw	mod reg r/m	disp-lo	disp-hi	
cmp accumulator to immed	0011 110w	data-lo	data-hi		
inc 16 bit reg	0100 0 reg				
dec 16 bit reg	0100 1 reg				
push reg	0101 0 reg				
pop reg	0101 1 reg				
jo jump overflow	0111 0000	ip-inc-8			
jno jump no overflow	0111 0001	ip-inc-8			
jb jump below	0111 0010	ip-inc-8			
jae jump above or equal	0111 0011	ip-inc-8			
je jump equal	0111 0100	ip-inc-8			
jne jump not equal	0111 0101	ip-inc-8			
jbe jump below or equal	0111 0110	ip-inc-8			
ja jump above	0111 0111	ip-inc-8			
js jump sign	0111 1000	ip-inc-8			
jns jump not sign	0111 1001	ip-inc-8			
jpe jump parity even	0111 1010	ip-inc-8			
jpo jump parity odd	0111 1011	ip-inc-8			
jl jump less	0111 1100	ip-inc-8			
jge jump greater or equal	0111 1101	ip-inc-8			
jle jump less or equal	0111 1110	ip-inc-8			
jg jump greater	0111 1111	ip-inc-8			
add immed to reg/mem	1000 000w	mod 000 r/m	disp-lo	disp-hi	data-lo
or immediate to reg/mem	1000 000w	mod 001 r/m	disp-lo	disp-hi	data-lo
adc immed to reg/mem	1000 000w	mod 010 r/m	disp-lo	disp-hi	data-lo
sbb immed from reg/mem	1000 000w	mod 011 r/m	disp-lo	disp-hi	data-lo
and immed to reg/mem	1000 000w	mod 100 r/m	disp-lo	disp-hi	data-lo
sub immed from reg/mem	1000 000w	mod 101 r/m	disp-lo	disp-hi	data-lo
xor immed to reg/mem	1000 000w	mod 110 r/m	disp-lo	disp-hi	data-lo
cmp reg/mem to immed	1000 000w	mod 111 r/m	disp-lo	disp-hi	data-lo

test reg/mem and reg
 mov reg/mem to reg/mem
 mov seg reg to reg/mem
 mov reg/mem to seg reg
 pop mem
 cbw convert byte to word
 cwd convert word to double
 mov mem to accumulator
 mov accumulator to mem
 movs move byte/word
 cmps compare byte/word
 test immed and accumulator
 stds store from al/ax
 lods load to al/ax
 scas scan byte/word
 mov immed to reg
 ret within seg
 mov immed to mem
 rol rotate left
 ror rotate right
 rcl rotate thru carry left
 rcr rotate thru carry right
 shl/sal shift left
 shr shift logical right
 sar shift arithmetic right
 loop loop cx times
 call within seg
 jmp within seg
 rep repeat
 test immed and reg/mem
 not invert
 neg change sign
 mul multiply (unsigned)
 imul integer multiply (signed)
 div divide (unsigned)
 idiv integer divide (signed)
 inc 8 bit reg / any size mem
 dec 8 bit reg / any size mem
 push mem

1000 010w	mod reg r/m	disp-lo	disp-hi			
1000 10dw	mod reg r/m	disp-lo	disp-hi			
1000 1100	mod 0 sr r/m	disp-lo	disp-hi			
1000 1110	mod 0 sr r/m	disp-lo	disp-hi			
1000 1111	mod 000 r/m	disp-lo	disp-hi			
1001 1000						
1001 1001						
1010 000w		disp-lo	disp-hi			
1010 001w		disp-lo	disp-hi			
1010 010w						
1010 011w						
1010 100w		data-lo	data-hi			
1010 101w						
1010 110w						
1010 111w						
1011 w reg		data-lo	data-hi			
1100 0011						
1100 011w	mod 000 r/m	disp-lo	disp-hi	data-lo	data-hi	
1101 00vw	mod 000 r/m	disp-lo	disp-hi			
1101 00vw	mod 001 r/m	disp-lo	disp-hi			
1101 00vw	mod 010 r/m	disp-lo	disp-hi			
1101 00vw	mod 011 r/m	disp-lo	disp-hi			
1101 00vw	mod 100 r/m	disp-lo	disp-hi			
1101 00vw	mod 101 r/m	disp-lo	disp-hi			
1101 00vw	mod 111 r/m	disp-lo	disp-hi			
1110 0010	ip-inc-8					
1110 1000	ip-inc-lo	ip-inc-hi				
1110 1001	ip-inc-lo	ip-inc-hi				
1111 001z						
1111 011w	mod 000 r/m	disp-lo	disp-hi	data-lo	data-hi	
1111 011w	mod 010 r/m	disp-lo	disp-hi			
1111 011w	mod 011 r/m	disp-lo	disp-hi			
1111 011w	mod 100 r/m	disp-lo	disp-hi			
1111 011w	mod 101 r/m	disp-lo	disp-hi			
1111 011w	mod 110 r/m	disp-lo	disp-hi			
1111 011w	mod 111 r/m	disp-lo	disp-hi			
1111 111w	mod 000 r/m	disp-lo	disp-hi			
1111 111w	mod 001 r/m	disp-lo	disp-hi			
1111 1111	mod 110 r/m	disp-lo	disp-hi			

Two detailed examples

- Given that the variable named varw is located 12 (decimal) bytes into the data segment, then convert this instruction into machine code: **mov ax,[varw]**

This is a *move memory to accumulator*

Use table-2 to locate the mov instruction whose

- source is memory
- destination is the accumulator.

1010	000w	disp-lo	disp-hi
------	------	---------	---------

Ax is a 16 bit register, so we are moving a word of data, so w=1.

Varw is located 12 decimal bytes (000C hex) into the data segment. Therefore, disp-lo = 0Ch and disp-hi = 00h. Note that the address is byte reversed in the machine code.

Substituting these values yields the following hex machine code: **A1 0C 00**

- Convert the following machine code back to symbolic assembler: **80 06 0B 00 14**

Table-3 shows a CODE byte of 80 identifies this as 1 of 8 potential instructions (add, or, adc, sbb, and, sub, xor, cmp). The REG field of the ADDR byte will determine the specific instruction.

CODE	ADDR
1000 000w	mod reg r/m disp-lo disp-hi data-lo data-hi

The second byte is an ADDR byte. It is 06 hex or 00000110 in binary.

It has the form MOD REG R/M. Therefore mod = 00 reg = 000 r/m = 110.

Using table-3, a reg of 000 defines this instruction as an add *immediate to register or memory*.

Table-1 shows mod=00 and r/m=110 defines the destination address mode as memory direct.

We now know the instruction is of the form: **ADD [memory],n**

Table-3 specifies two bytes following the ADDR field are the memory address (byte reversed).

We now know the instruction is of the form: **ADD [000Bh],n**

Going back to the CODE byte, it has the form 1000 000w

The value 80h = 1000 0000 so w=0 which means the data size is byte.

This means that data-lo will hold the immediate value. The data-lo byte is 14h.

Putting this together yields the following symbolic instruction: **ADD [000Bh],14h**

More samples of conversion between symbolic and machine

add si,500 add 500 decimal (01f4h) as a word to si

add immediate to register = [1000000w] [mod 000 r/m] [data-lo] [data-hi]

si is a word size register, so w = 1 for data that is word size
destination is a register, so mod = 11 and r/m = 110 to specify the si register

[10000001] [11 000 110] [data-lo] [data-hi]

data-lo = f4 and data-hi = 01 so final machine code is: **81 C6 F4 01**

dec cx cx = cx -1 (cx is 16 bit register)

dec 16 bit register = [01001xxx]

xxx = the register = 001 to specify cx

machine code = 01001001 = **49**

mov ax,bx move the contents of bx to ax

mov reg to reg = [100010dw] [mod reg r/m] (the disp-lo and disp-hi are not needed)

w = 1 means the data size is word

The reg field will specify one of the registers, mod will be set to 11 and r/m will specify the other register. We have a choice, so I selected reg to specify ax and r/m to specify bx.
reg = 000 for ax
r/m = 011 for bx

d = 1 means data moved to ax register specified by the reg field.

machine code = [10001011] [11 000 011] = **8B C3**

Convert this machine code to symbolic: **2B 16 F8 AF**

2B = 0010 1011 = 0010 10dw = sub reg/mem from reg/mem

0010 10dw	mod reg r/m	disp-lo	disp-hi
-----------	-------------	---------	---------

ADDR = mod reg r/m = 16 hex = 00 010 110 binary

mod = 00 and r/m=110 implies memory address for source specified by disp-lo and disp-hi
reg = 010 indicates the destination register is dx or dl

d=1 means the reg field specifies the destination and w=1 so size is word
We now know we are subtracting from the word size register dx

The instruction must be: **SUB DX,[0AFF8H]**

Additional thoughts on machine code

How do you handle two register operands, for example: add bx,cx

1. Start with the register plus register instruction format.

[000000dw] [mod reg r/m]

2. Using Table-1 we see how mod reg r/m are used

mod = 11

reg = register 1

r/m = register 2

The choice of register 1 and register 2 is arbitrary, but will affect how the direction bit is set.

We will use bx for register 1 and cx for register 2. This yields ...

mod = 11

reg = register 1 = bx = 011

r/m = register 2 = cx = 001

3. The destination is bx so set d = 1 indicating the reg field is the destination
4. w = 1 indicating word size data
5. This yields...[00000011] [11 011 001]

The final hex result is: 03 D9

At step 2 we could also have made this choice:

reg = register 1 = cx = 001

r/m = register 2 = bx = 011

Now the direction must show that the reg field, representing cx, is the source, so d = 0.

This yields...[00000001] [11 001 011]

The final hex result is: 01 CB

Both answers are correct.

Program Testing

Program Development Process Stages

Any complex task, that one wishes to complete without defects and in repeatable manner, should be controlled and monitored by a written set of rules called a *process*. For example, the steps in building a house are closely controlled with inspections performed after each step to assure the house meets building code standards.

Software should be developed in a similarly controlled manner. Many large software projects do follow some form of *Software Development Process*. Although there is not one industry standard process, the following six stage process is typical. Each stage will have inputs (items to be completed before the stage starts) and outputs (results of activities during the stage).

Requirements

Input(s): A request for a new product.
Activities: Gather input from the customer. Perform competitive analysis.
Output(s): A Requirements Document describing what the product should look like.

Design

Input(s): Requirements Document.
Activities: Determine how the problem will be solved. Review the design with experts.
Output(s): A Design Specification document that describes how the product will be developed.

Code

Input(s): Design Specification.
Activities: Convert the design into a programming language. Review the code with experts.
Output(s): Executable code for each component or subroutine.

Unit Test

Input(s): Code for a component or subroutine.
Activities: Developers test their individual components. Have the test plans reviewed by experts.
Output(s): Tested code for a component or subroutine.

System Integration and Test

Input(s): Tested code for all components or subroutines.
Activities: All components are combined into a system and tested. Validate the product meets customer requirements.
Output(s): A fully tested product that can be shipped to customers.

Maintenance

Input(s): A shipped product.
Activities: Customer reported problems are fixed.
Output(s): Code fixes and new versions of the product.

A significant set of activities in this process is concerned with testing the individual components and testing the system. The rest of this section will provide you with techniques to help you test your individual component or subroutine.

Definitions

Testing: The process of executing a program with the goal of finding errors.

- *Unit Test* - tests an individual component or subroutine and is performed by the developers
- *System Test* - tests all the components combined into a system. System Test must be performed by a separate test organization, not the programming organization. The use of a separate organization assures an unbiased opinion about the quality of the code that was developed.

Test Case: A set of inputs and the expected outputs.

Black Box Testing: Functional tests derived from the specifications or a user's manual.

White Box Testing: Tests derived from examining the code.

Type of test and techniques used

Black Box

- *Boundary Value Testing* . Test all boundary conditions. Most errors occur at the allowed limits for data. This is due to the fact that the developer must perform tests and make decisions at these limits thus providing an opportunity to introduce an error.
- *Equivalence Partitioning*. Break input into similar groups and build test suites to cover all groups. For any real program it will be impossible to exhaustively test all possible combinations of inputs. You must therefore, define broad areas to be tested and develop a reasonable set of tests to cover each of these broad areas.

White Box

- *Boundary Value Testing* . Same as above.
- *Statement Coverage*. Your goal is to assure that your tests provide 100% instruction coverage. That means that during the execution of your tests, all the instructions you have written are executed. If you have instructions which have not been executed you either have code that can never be executed or you will ship untested code that will then be tested by the customer (usually not a good idea).

Test Plans

An important question to be answered is '*How do you know when code is ready to be released for system's integration or how do you know when a product is ready to be released to customers?*'. The answer is that you develop a WRITTEN test plan. The test plan describes ALL the testing that you determine needs to be completed in order to release the code. Until all the tests run successfully, the code is not ready. When all the tests do run successfully, then the code is ready since you have previously established this plan represents all the required testing. You will often find that the work associated with developing and writing a test plan and building all the test cases, is MORE than that needed to write the code.

Another important factor is that AUTOMATED testing is far superior to manual testing. With manual testing you run each test case individually and visibly verify the results. This may be very time consuming and under the pressure of completing a project on schedule you may reduce your testing effort as you near the delivery date. This is the exact opposite of what you should do and often has disastrous results. To achieve automated testing you build programs or batch files that run the tests and compare the calculated results to the expected results. Automated testing can take orders of magnitude less time to run the test cases as compared to manual testing. Of course you have to expend the resources to build the automated test system. Running all test cases efficiently is very important because EVERY time you make a change, you need to rerun all test cases to assure you did not introduce a bug. This is true even if you only changed a comment; Murphy's law says you will introduce a bug. In the real world you should always try for

automated testing; in school it may be too much work. Since I have to grade 75 copies of each program for each section I teach, all my testing is automated.

How to build an Equivalence Partition and Test Cases

This is a skill that must be developed. I can provide the basic ideas, but practice in developing test cases is the only way to learn to do this activity well. Lets go through an example.

We will develop the test cases for a weekly payroll program. This program has two inputs for each check to be written: a five digit employee number and the number of hours worked by the employee. The program will look up the employee's salary and calculate a check equal to the hourly salary * hours worked.

Notes:

- There are 100,000 employee numbers. However the company does not have 100,000 employees. This means that some employee numbers are not currently active. They represent retired employees, employees that left the company, or are reserved for new employees.
- The number of hours worked is specified as whole integers. 0-40 is regular time. 41-80 is overtime and thus paid at a higher rate. An employee is not allowed to work more than 80 hours per week.

I have chosen to break the employee number input into these groups: 5 digit active employee numbers; 5 digit inactive employee numbers; employee numbers entered with less than five digits; employee numbers entered with more than five digits; employee numbers that contain non numeric data. Only the first group represents valid input. Lets assume for this example that employee numbers 00000-02155 are active; others are inactive.

We will also assume that all employees are paid \$10/hour regular time and \$15/hour overtime.

I have chosen to break the hours worked input in these groups: 0-40 hours; 41-80 hours; greater than 80 hours; a non numeric input (which will catch an attempt to enter negative hours such as -20).

We now build an equivalence partition by listing the selected groups of inputs. This process defines a set of Test Suites by generating all the possible combinations of inputs. In this example we have $5 \times 4 = 20$ combinations of input data giving 20 test suites.

Suites 1-20

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2
Employee Number																						
5 digit active	x		x			x				x												
5 digit inactive	x			x			x				x											
> 5 digits	x			x				x			x				x							
< 5 digits		x			x				x		x				x					x		
non numeric		x			x				x		x			x			x			x		
Hours																						
0-40	x	x	x	x	x																	
41-80		x	x	x	x	x																
> 80						x	x	x	x	x												
non numeric											x	x	x	x	x							

This generates the worst case scenario. In theory, we would now generate multiple test cases for each suite. This is specially true for black box testing where we have no knowledge of the internal workings of the program. We may, however, decide that based upon our knowledge of the program to combine test suites. By talking to the programmers we may discover that once the code finds a non numeric employee number, then it stops processing. This means that suites 5,10,15,20 may be combined. Always keep in mind that any

shortcuts you take now, may reduce your initial testing effort, but these may come back to haunt you at a later time. Everything is a judgment call. For this example, lets assume once we find any invalid employee number we stop processing. This means you can combine the error suites: (2,7,12,17), (3,8,13,18), (4,9,14,19), (5,10,15,20) into the lowest numbered suite.

Remember that when we build test cases for a test suite we want to assure we cover typical values and boundary values. Test cases are numbered S.NN where S is the test suite and NN is the test case number within that suite. Inputs are Emp# and Hours. Expected output is Gross (an amount or ‘none’ if error).

Test Case	Emp.#	Hours	Gross	Explanation.
1.01	00000	0	\$0	Covers full valid range, including boundaries
1.02	00001	1	\$10	of 5 digit active numbers and 0-40 hours
1.03	01245	9	\$90	
1.04	02000	33	\$330	
1.05	02107	39	\$390	
1.06	02155	40	\$400	
2.01	02156	10	none	Covers 2, 7, 12, 17
2.02	99999	55	none	
3.01	000000	8	none	Covers 3, 8, 13,18
3.02	011111	66	none	
3.03	333333	99	none	
4.01	0	18	none	Covers 4, 9, 14, 19
4.02	333	1	none	
5.01	A	10	none	Covers 5, 10, 15, 20
5.02	0111B	40	none	
6.01	01001	41	\$415	Covers overtime hours worked
6.02	01867	72	\$880	
6.03	02155	80	\$1000	
11.01	01111	81	none	Cover too many hours worked in a week
11.02	02020	200	none	200 ₁₀ = C8 ₁₆ which may look negative to a byte counter
16.01	00567	1*	none	Covers non numeric hours
16.02	02001	-10	none	

Test Plan Contents

The test plan should be developed along with the code, not afterwards. It requires as much thought as the code. In the real world you may be fortunate enough to be on a team where a separate person spends their full time building the test plan and automated test system.

Test plans should include the following items.

1. A description of what functions are being tested (for example the Input/Output specifications for a subroutine).
2. An explanation of how the tests were developed (for example tests were developed from an equivalence partition and provide 100% instruction coverage). This part will be the hardest to develop. Each test must have a purpose. Even if you have trouble developing this section, whatever you have will be a start, and can be expanded upon as you get more information. Your goal is to assure that your tests provide 100% instruction coverage. If you have instructions which have not been executed you either have code that can never be executed or you need more tests.
3. A list of all tests (identification number, inputs, expected results). If these are defined somewhere else, like a test driver, then a reference to that location is okay.
4. The test results (date tests were executed, which tests passed/failed, percent of instructions covered by the tests, file names and dates, etc.). This would be added after testing is complete, possibly as an appendix or attachment. The key is to be able to show an objective observer that the tests were run and that you can reproduce test execution if required (you may wish to archive all files associated with a test).

Test plans should be reviewed by a lead programmer or tester. They will often make valuable suggestions.

Whenever you uncover a bug in your code that was not found by one of your test cases, you should add a test case that will find that bug. It is possible that a future change to the code may reintroduce that bug.

Conclusion

When you have a well developed test plan which has been reviewed by peer programmers or testers, then your code is ready to ship when all the tests execute correctly and 100% instruction coverage is achieved.

In the next two pages, I pulled together the test data and simulated results for the payroll program and have a model of a complete test plan. Remember that the goal of the test plan is to be able to verify to an unbiased auditor that the program was tested. An auditor's favorite words are 'show me'. So if you make a statement, be sure you have some evidence that the statement is correct.

Title: Test plan for PAYROLL program
 Owner: Dana Lasher
 Date: 08/04/96
 Revisions: 07/03/96 Original version
 08/04/96 Updated with results of running tests on 7/31/96

Page 1/2

This is the test plan for the PAYROLL program. It contains 4 sections:

1. The Input/Output specification for the PAYROLL program which defines the functions to be tested.
2. An explanation of how the tests were developed. For the PAYROLL program an Equivalence Partition is used to show how function coverage is obtained.
3. The specific list of all tests in the test suites.
4. The test results for the current version of the PAYROLL program.

1. Input/Output specification for the PAYROLL program

The PAYROLL program has two inputs: employee number and the number of hours worked this week. Only active employee numbers are valid. Employees may work 0-80 hours in a given week. Hours worked which exceed 40, are paid at an overtime rate. If all inputs are valid then a check for the gross amount is created. If any input is invalid then an error message is generated, and the gross amount is not calculated.

Input: 5 digit active employee number (exactly 5 numeric digits for a currently active employee).
 2 digit number in the range of 0-80 indicating the number of hours worked.

Output: A check for the gross amount if no errors were detected in the input values.
 An error message in an error was detected in an input value.

2. Description of the tests were developed

An Equivalence Partition is used to define the possible combinations of inputs. This Equivalence Partition is a matrix showing the possible combinations of the two inputs (employee number, hours worked). All combinations of inputs are covered. A total of 20 test suites are used to cover all the combinations of input values.

Suites 1-20

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	
Employee Number 5 digit active	x		x		x		x		x		x		x		x		x		x		
5 digit inactive	x		x		x		x		x		x		x		x		x		x		
> 5 digits	x		x		x		x		x		x		x		x		x		x		
< 5 digits	x		x		x		x		x		x		x		x		x		x		
non numeric		x		x		x		x		x		x		x		x		x		x	
Hours 0-40	x	x	x	x	x																
41-80		x	x	x	x	x															
> 80						x	x	x	x	x	x										
non numeric							x	x	x	x	x	x									

Based upon our knowledge of the program , once an invalid employee number is found, the program stops processing. We have therefore combined the error suites: (2,7,12,17), (3,8,13,18), (4,9,14,19), (5,10,15,20) into the lowest numbered suite. This reduces the work effort in testing the program.

3. Specific test case descriptions.

Page 2/2

Test cases are numbered S.NN where S is the test suite and NN is the test case number within that suite. Inputs for each test case are: employee number and hours worked. The expected output from each test case is the gross amount of the pay check (an amount or 'none' if an error was detected in the input values).

Test Case	Emp.#	Hours	Gross	Explanation.
1.01	00000	0	\$0	Covers full valid range, including boundaries
1.02	00001	1	\$10	of 5 digit active numbers and 0-40 hours
1.03	01245	9	\$90	
1.04	02000	33	\$330	
1.05	02107	39	\$390	
1.06	02155	40	\$400	
2.01	02156	10	none	Covers 2, 7, 12, 17
2.02	99999	55	none	
3.01	000000	8	none	Covers 3, 8, 13,18
3.02	011111	66	none	
3.03	333333	99	none	
4.01	0	18	none	Covers 4, 9, 14, 19
4.02	333	1	none	
5.01	A	10	none	Covers 5, 10, 15, 20
5.02	0111B	40	none	
6.01	01001	41	\$415	Covers overtime hours worked
6.02	01867	72	\$880	
6.03	02155	80	\$1000	
11.01	01111	81	none	Cover too many hours worked in a week
11.02	02020	200	none	200 ₁₀ = C8 ₁₆ which may look negative to a byte counter
16.01	00567	1*	none	Covers non numeric hours
16.02	02001	-10	none	

4. Test results

Date run: 07/31/96

Test status: All test cases ran successfully. No defects in the PAYROLL program were detected.

The following files were used or created in these tests. The output file PAYROLL.OUT contained results that matched the expected output for all test cases. These files have been archived and may be inspected to verify successful test completion.

PAYROLL.ASM	49703	07-31-96	1:15p
PAYROLL.EXE	14216	07-31-96	1:28p
PAYROLL.IN	2084	07-22-96	9:35a
PAYROLL.OUT	8056	07-31-96	2:16p

***** END OF DOCUMENT *****

Boolean Algebra, Shifts and Rotates, Hardware Circuits

Boolean Algebra was developed by the English mathematician George Boole (1815-1864). In this algebra, there are variables that have either the value 0 or 1. There are also functions, analogous to addition and subtraction, which have inputs and outputs. These functions are quite useful in applications that involve logical analysis and bit manipulation, such as data compression.

The key Boolean functions of interest are: *not*, *and*, *inclusive or*, *exclusive or*.

The tables that describe how these functions generate output values are called *truth tables*.

not

Input	Output
0	1
1	0

and

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

inclusive or

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

exclusive or

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

The 8086 has instructions that implement these Boolean functions.

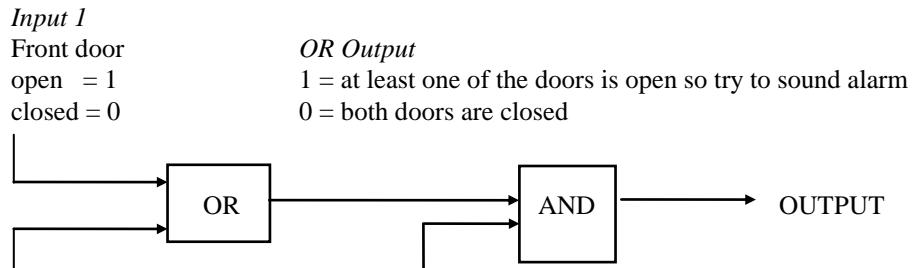
To use these functions in an application that requires logical analysis we can equate *events* with *inputs* and *outputs*.

- 0 means the event is false or the event did not happen
- 1 means the event is true or the event did happen

Example

This is an example of the use of these functions in the design of an alarm system for the home. The alarm is to sound if either the front door or the back door is opened, and the alarm has been turned on. The assumption is that if either door is open then a burglar is trying to enter the house.

A conceptual layout of this system contains an INCLUSIVE OR function with an AND function.



Input 2

Back door
open = 1
closed = 0

Input 3

Alarm turned on = 1
Alarm turned off = 0

<u>Input 1</u>	<u>Input 2</u>	<u>OR Output</u>	
0 closed	0 closed	0 safe	Only try to sound alarm
0 closed	1 open	1 burglar	when at least one door
1 open	0 closed	1 burglar	is open.
1 open	1 open	1 burglar	

<u>OR Output</u>	<u>Input 3</u>	<u>And Output</u>	
0 safe	0 inactive	0 quiet	Only sound alarm when
0 safe	1 active	0 quiet	trying to sound alarm
1 burglar	0 inactive	0 quiet	and the alarm is active.
1 burglar	1 active	1 sound alarm	

Shifts and rotates ... see chapter 6 for details on how to use these instructions

There is another set of logical operations that are very heavily used in applications that need to manipulate individual bits. These are the shift and rotate operations.

- Shifts move data left or right.
 - Shifting right effectively divides by 2
 - Shifting left effectively multiplies by 2
 - In the 8086, the carry flag always contains the value of the last bit shifted out of the destination operand.
 - Rotate is similar to shift except bits are not lost but are circled back into the other end of the operand.
 - The source operand is an immediate value of 1 in which case the destination is shifted 1 bit.
 - The source operand can be the cl register where the shift amount is loaded into cl before the shift is executed.

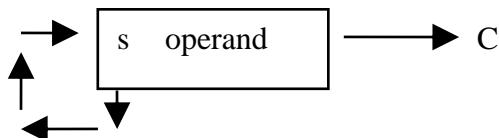
sal / shl

Shift arithmetic left / Shift logical left



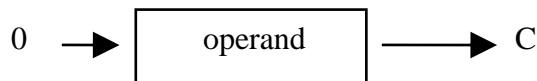
sar

Shift arithmetic right.



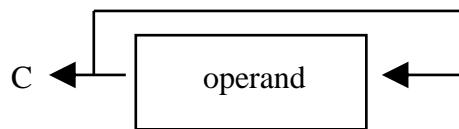
shr

Shift logical right.



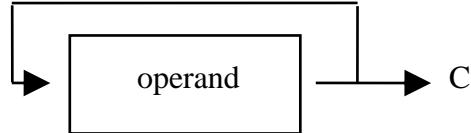
rol

Rotate left.



ror

Rotate right.



Example of code to count the number of 1 bits in a field.

4E	03	71	2A	
----	----	----	----	--



si points to a byte of data in memory

```
        mov    ax,0          ; # of 1 bits starts as zero
        mov    bl,[si]        ; get the byte to analyze
        mov    cx,8          ; 8 bits in the byte to be processed
testit:   shl    bl,1          ; shift 1 bit into the carry flag
        jnc    next          ; do not count 0 bits
        inc    ax            ; do count 1 bits
next:     loop   testit       ; loop if more bits to process
```

Use of Add With carry

```
        mov    ax,0          ; # of 1 bits starts as zero
        mov    bl,[si]        ; get the byte to analyze
        mov    cx,8          ; 8 bits in the byte to be processed
testit:   shl    bl,1          ; shift 1 bit into the carry flag
        adc    ax,0          ; count = count + 0 + the last bit shifted
next:     loop   testit       ; loop if more bits to process
```

A data compression example

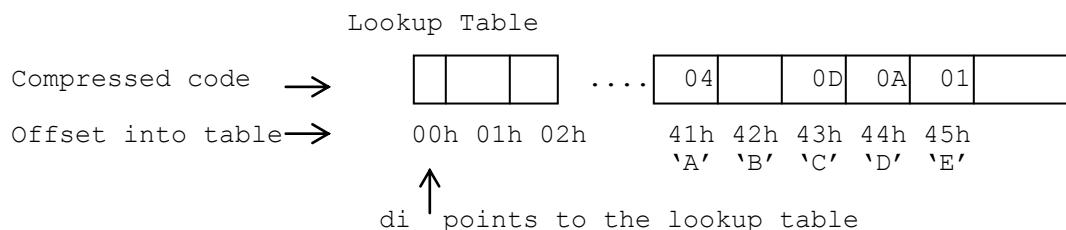
- Standard ASCII uses 1 byte per character.
 - We identify the 15 most frequently occurring characters in English text and assign each a 4 bit hex code.
- | | |
|-----------|--------------------------------|
| Char: | sp E T O A N I S R H D L M C U |
| Hex code: | 0 1 2 3 4 5 6 7 8 9 A B C D E |
- There are two modes of operation.
Uncompressed - send 1 byte per char.
Compressed - send 2 high frequency characters per byte.
 - Rules
 1. Start in uncompressed mode.
 2. When we get two high frequency characters in a row then switch to compressed mode. Use FFh as the switch indicator.
 3. When we get a non high frequency character then switch back to uncompressed mode. Use FFh as the switch indicator.
 - Statistics

Sample ASCII input data is 31 bytes:
KEY ENTERED DATA HAS REDUNDANCY

Compressed output data is 20 bytes:
K E Y FF 01 52 18 1A 0A 42 40 94 70 81 AE 5A 45 FF C Y

Uncompressed ASCII data = 8 bits/char
Compressed data = $\frac{20 \text{ bytes} * 8 \text{ bits/byte}}{31 \text{ chars}} = 5.16 \text{ bits/char}$

Example code for packing two high frequency characters into a byte.



```

mov    bx,0          ;bx = 00 00
mov    bl,[si]        ;bx = 00 45 = 'E'
mov    al,[bx+di]    ;al = 01 = coded 'E'
shl    al,4          ;al = 10 = shift code left 4 bits
mov    bl,[si+1]      ;bx = 00 44 = 'D'
or     al,[bx+di]    ;al = 1A = coded 'ED'
mov    [?],al         ;move code to output

```

Hardware logic

The hardware of a computer is built from Boolean logic circuits. This section will look at these logic circuits. We will also see that all of these different logic circuits can be built from one single primitive function. We will show how this is done by building up a number of circuits from this primitive.

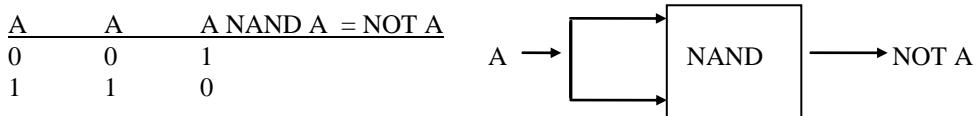
Building more complex Boolean functions from one basic primitive

All of the Boolean functions can be built from one primitive function, the NOT-AND (NAND) function.

NAND: The output is 1 unless both of the inputs are 1

INPUT1	INPUT2	OUTPUT
0	0	1
0	1	1
1	0	1
1	1	0

How are the other functions built from the NAND function? The value of 'A' NAND 'A' will be NOT 'A'.



With the NAND and the NOT we can build an AND function. $\text{NOT}(\text{A NAND B}) = \text{A AND B}$

A	B	A NAND B	NOT(A NAND B) = A AND B
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

With the NOT and NAND we can build an OR function. $(\text{NOT A}) \text{ NAND } (\text{NOT B}) = \text{A OR B}$

A	B	NOT A	NOT B	(NOT A) NAND (NOT B) = A OR B
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

EXCLUSIVE OR is equal to $((\text{NOT A}) \text{ AND B}) \text{ OR } (\text{A AND } (\text{NOT B}))$.

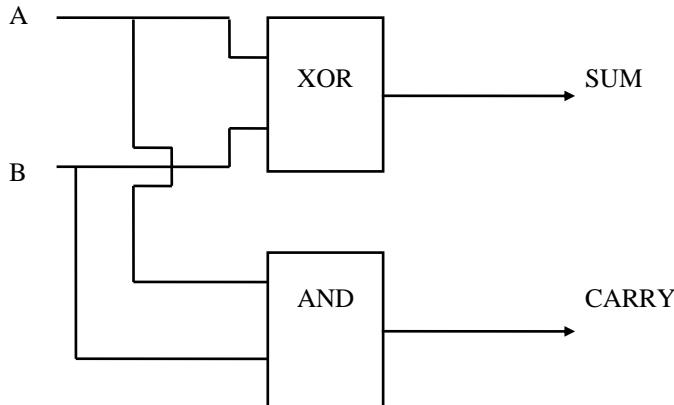
The truth table for that is left to the reader.

Building an ADDER circuit

We can build a circuit that adds two bits out of Boolean functions. The truth table for an adder is:

A	B	A + B (SUM)	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

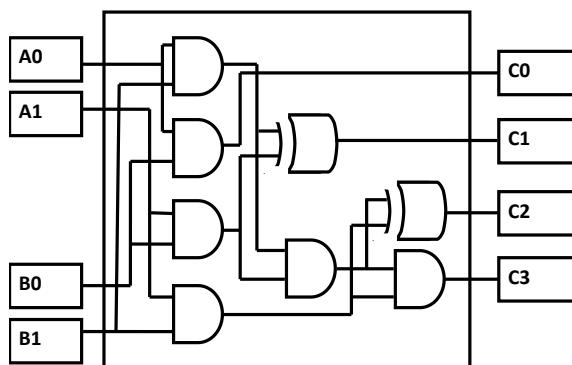
The SUM is A XOR B, and the carry is A AND B. We can thus build an adder out of two Boolean circuits.



Building a MULTIPLY circuit

Given enough circuits we can build a multiply circuit. The unsigned multiply circuit below uses 6 AND gates and 2 EXCLUSIVE OR gates and will multiply 2 bits x 2 bits creating a 4 bit product.

2 bit x 2 bit binary multiplier



To verify it works select sample values for the inputs A0, A1, B0, B1. Try A0=A1=B0=B1=1.

The inputs to the left 4 AND gates will be 1 and so their outputs will be 1 and so C0=1.

The inputs to the middle XOR gate will 1 and so its output will be 0 and so C1=0.

The inputs to the middle AND gate will 1 and so its output will be 1.

The inputs to the right XOR gate will 1 and so its output will be 0 and so C2=0.

The inputs to the right AND gate will 1 and so its output will be 1 and so C3=1.

Thus $11_2 \times 11_2 = 1001_2$

Microprogramming & String Instructions

Microprogramming is a technique used to implement a given machine instruction set architecture. It is not visible to the assembler language programmer; it is considered as part of the hardware of the computer. Each machine instruction is implemented by executing a microprogram which is comprised of one or more microinstruction. Each microinstruction controls the flow of data inside the machine and the actions of the hardware components such as the ALU. Microinstructions activate the real hardware in the ALU.

Microprogramming was very heavily used between 1965 and 1990.

The microcode defines the personality of the computer. If you can change the microcode, then you can allow the computer to execute any format of machine/assembler instruction you desire. It is very powerful.

What are the trade-offs between microprogramming, and a direct hard-wired approach where there is a separate circuit for each instruction?

- The main advantage of microprogramming is flexibility. One can change the machine language for a computer by replacing the microprogram. One would not have to modify the actual hardware since only the most basic operations, like ADD, are actually implemented in the ALU. One use for this emulation is when a new model of a computer can be used to run the software written for an older incompatible model. Another use is for the same hardware computer chip being able to run 68000 software and 8086 software.
- The main advantage of direct hard wiring instructions is speed; the flexibility associated with microprogrammed instructions tends to make them slower than direct hard wired instructions.

Programming With 8086 String Instructions

Intel used microcode to implement a set of very powerful string instructions for use in text processing applications. The five string instructions *load*, *store*, *move*, *compare*, and *scan* operate on strings of bytes or words.

- The string transfer instructions (*lod*s, *mov*s, *stos*) copy bytes or words from memory to register, from register to memory, or directly from memory to memory.

On the 8086, memory to memory addressing is unique to string instructions.

- The string inspection instructions (*cmp*s, *scas*) let you compare and scan bytes or words in a string to search for a specific value.

All string instructions use specific registers to perform their functions. When addressing memory, string instructions use **si** and **di** in this manner.

- The **si** index register specifies the offset into the **data segment** of the **source** operand.
- The **di** index register specifies the offset into the **extra segment** of the **destination** operand. The es (extra segment) register must point to the start of the extra segment in order to use a string instruction with a destination operand.

While executing their various operations, each string instruction automatically increases or decreases the index registers they use.

- Byte operations add or subtract 1 to si or di (or both).
- Word operations add or subtract 2 to si or di (or both).

The setting of the direction flag determines whether a string instruction will increment or decrement the index register(s) used. The *cld* (clear direction flag) and *std* (set direction flag) commands are detailed later.

Each string instruction has three possible formats.

- The general format, like *movs [byte di],[byte si]* must specify the operands and their size.
- The shorthand formats do not require you to specify the operands and size. Instructions ending with *b* such as *movsb* and *cmpsb* imply the use of byte operands and formats ending with *w* imply the use of word operands.
- Specialized prefixes (*rep*, *repe*, *repz*, *repne*, *repnz*) to string instructions allow you to repeat that instruction the number of times specified in the cx register or until a certain condition is met. *rep* is different from the *LOOP* instruction. If *cx=0* then the loop instruction would repeat 65536 times. However, *rep* is different and *rep* means *repeat while cx not zero*. So if *cx=0* then a *rep* operation would be performed zero times.

String Opcodes

First we cover operations that support your use of the string instructions.

The direction flag, in the Status Flag register controls whether the index registers are incremented or decremented. This allow you to process strings in the forward or backward directions.

cld	<p>Clear direction flag.</p> <p>Clears the direction flag in the condition code to zero. Use cld before a repeated string operation to automatically increment the string index registers, si and di.</p> <ul style="list-style-type: none">• cld only affects the direction flag in the condition code.• Example: cld
std	<p>Set direction flag.</p> <p>Sets the direction flag in the condition code to 1. Use std before a repeated string operation to automatically decrement the string index registers, si and di.</p> <ul style="list-style-type: none">• std only affects the direction flag in the condition code.• Example: std

The repeat prefixes allow you to repeat string operations. This is the most common way that string operations are used.

The basic rep prefix uses cx as a counter.

rep	<p>Repeat a string instruction using cx as a counter. Rep means repeat while cx not zero. That is, the instruction is repeated until cx=0. If cx starts as zero the operation will not be performed.</p> <ul style="list-style-type: none">• rep does not set the condition code.• Example: rep movsb
-----	---

These versions of the repeat prefix use **both** cx as a counter **and** the zf as terminator. This allows you to stop either at the end of a string or if the two strings become unequal or equal.

repe repz	<p>Repeats a string instruction the number of times specified in cx while string operands are equal. The repe and repz opcodes are only used with the string opcodes scas and cmps because those are the only string instructions that modify the zero flag (zf).</p> <ul style="list-style-type: none">• repe or repz do not set the condition code. <p>Example: repe cmpsb repz scasw</p>
repne repnz	<p>Repeats a string instruction the number of times specified in cx while string operands are not equal. The repne and repnz opcodes are only used with the string opcodes scas and cmps because those are the only string instructions that modify the zero flag (zf).</p> <ul style="list-style-type: none">• repne or repnz do not set the condition code. <p>Example: repne cmpsb repnz scasw</p>

The compare string instruction

cmps dest,source cmpsb cmpsw	<p>Compare strings.</p> <p>Compares the source string in data segment memory addressed by si to the destination string in extra segment memory addressed by di.</p> <p>The string compare works by subtracting the destination from the source, but not storing the result, then setting the condition code based upon result.</p> <p>The order of comparison is the reverse of the standard compare instruction. The standard compare subtracts the source from the destination.</p> <p>After the first non-matching values are found in the strings, the si and di registers point 1 unit (1 byte or 1 word depending on operand size) beyond the position of the non-matching value in the strings.</p> <ul style="list-style-type: none">• compare strings will set the condition code. <p>Example: cmps [byte di],[byte si]</p> <p>Compare the value of the source data string pointed to by si to the value of the destination data string pointed to by di. This can be viewed as [byte si] - [byte di].</p> <p>Example: cmpsb ; the shorthand version of the above cmpsw ; the shorthand that compares words</p>
------------------------------------	--

Some notes:

- Before using cmps you want to set the direction flag.
- If you do not use some form of rep prefix, then you will only compare one element. It is the rep prefix along with a count in cx that makes this a truly useful instruction.

Example that compares a 5 byte string 'str1' to another 5 byte string 'str2'.

```
        databaseg
;-----
str1    db      'will'           ;variable string 1
str2    db      'wills'          ;variable string 2
;-----

        codeseg
;-----
start:
        mov      ax,@data      ;address the
        mov      ds,ax          ;data segment
        mov      es,ax          ;extra segment

;-----
; compare 'str1' to 'str2' the basic way
;-----
        mov      si,offset str1 ;point to str1
        mov      di,offset str2 ;point to str2
        mov      cx,5            ;count is 5
compare:
        mov      al,[si]         ;get char from str1
        cmp      al,[di]         ;compare str1 to str2
        jb     less              ;jump if str1 < str2
        ja     greater           ;jump if str1 > str2
while_equal:
        inc      si              ;advance str1 pointer
        inc      di              ;advance str2 pointer
        loop   compare           ;loop
equal:                                ;end here if they are equal

;-----
; compare 'str1' to 'str2' using the string instruction
;-----
        mov      si,offset str1 ;point to str1
        mov      di,offset str2 ;point to str2
        mov      cx,5            ;count is 5
        cld
repe  cmpsb
        jb     less              ;jump if str1 < str2
        ja     greater           ;jump if str1 > str2
equal:                                ;end here if they are equal
```

The load accumulator from string instruction

lod\$	Load accumulator from string.
lod\$B	Loads a memory byte or word addressed by si in the data segment into the accumulator ax or al, then si is updated.
lod\$W	<ul style="list-style-type: none">• lod\$ does not set the condition code.
	Example: lod\$B lod\$W

An example that loads and sums the values in a string.

```
Dataseg
;-----
buffer dw      0,1,2,3,4,5,6,7,8,9 ;list of values
sum   dw      0                   ;sum of values

codeseg
;-----
start:
    mov      ax,@data           ;address the
    mov      ds,ax              ;data segment

;-----
; load 'buffer' to ax the basic way
;-----
load1:
    mov      [sum],0            ;initialize sum
    mov      si,offset buffer  ;point to buffer
    mov      cx,10              ;count is 10
    mov      ax,[si]            ;get value from buffer
    add      [sum],ax           ;sum = sum + ax
    add      si,2               ;advance buffer pointer
    loop    load1              ;loop

;-----
; load 'buffer' to ax using string instruction
;-----
load2:
    lodsw                ;load buffer value into ax
    add      [sum],ax           ;sum = sum + ax
    loop    load2              ;loop
```

The move string instruction

movs movsb movsw	Move string. Copies a byte or word from memory addressed by si in the data segment to memory addressed by di in the extra segment, then si and di are updated. <ul style="list-style-type: none">• movs does not set the condition code.• Example: movsb movsw
------------------------	--

An example that copies an 80 byte string 'a' to another 80 byte string 'b'.

```
        dataseg
;-----
a      db      80 dup('a')           ;variable a
b      db      80 dup('b')           ;variable b

        codeseg
;-----
start:
        mov      ax,@data           ;address the
        mov      ds,ax              ;data segment
        mov      es,ax              ;extra segment

;-----
; copy 'a' to 'b' the basic way
;-----
        mov      si,offset a         ;point to a
        mov      di,offset b         ;point to b
        mov      cx,80               ;count is 80
moveit:
        mov      al,[si]             ;get a char
        mov      [di],al              ;put a char
        inc      si                  ;advance pointer
        inc      di                  ;advance pointer
        dec      cx                  ;reduce count
        jne      moveit              ;loop if more data

;-----
; copy 'a' to 'b' using a string instruction
;-----
        mov      si,offset a         ;point to a
        mov      di,offset b         ;point to b
        mov      cx,80               ;count is 80
        rep      movsb               ;move the string
```

The scan string instruction

scas scasb scasw	<p>Scan string. Scans a string in memory pointed to by di in the extra segment for a value that matches the byte or word value in al or ax.</p> <p>After the search value is found in the string, the di register points 1 unit (1 byte or 1 word depending on operand size) beyond the position of the matching character.</p> <ul style="list-style-type: none"> • scas will set the condition code. <p>Example: scasb scasw</p>
------------------------	---

This is an example of code that scans a 16 byte string for the letter 'g'.

```

        dataseg
;-----
alpha    db      'abcdefghijklmnopqrstuvwxyz' ;list of chars

        codeseg
;-----
start:          ;
    mov      ax,@data           ;address the
    mov      ds,ax              ;data segment
    mov      es,ax              ;extra segment

;-----
; scan the basic way
;-----
    mov      di,offset alpha   ;point to alpha
    mov      cx,16              ;count is 16
    mov      al,'g'             ;set search value = 'g'
scan:           ;
    cmp      [di],al           ;is char = 'g'?
    je      found              ;match found
    inc      di                 ;advance alpha pointer
    loop     scan               ;loop
notfound:       ;not found

;-----
; scan using string instruction
;-----
    mov      di,offset alpha   ;point to alpha
    mov      cx,16              ;count is 16
    mov      al,'g'             ;set search value = 'g'
    cld
repne scasb    ;scan while char!='g'
    je      found              ;match found
notfound:       ;not found

```

Store string instruction

stos	Store in string.
stosb	Stores the value in al or ax to the memory location addressed by di in the extra segment, then di is updated.
stosw	• stos does not set the condition code. Example: stosb stosw

An example that stores the value in al to a 20 byte string.

```
        dataseg
;-----buffer db      20 dup(?)           ;buffer list

;-----start:                      ;
    mov      ax,@data            ;address the
    mov      ds,ax              ;data segment
    mov      es,ax              ;extra segment

;-----; store al in 'buffer' the basic way
;-----
    mov      di,offset buffer   ;point to buffer
    mov      cx,20               ;count is 20
    mov      al,55h              ;set al=55h
store:
    mov      [di],al            ;store al in buffer
    inc      di                 ;advance buffer pointer
    loop     store               ;loop

;-----; store al in 'buffer' using string instruction
;-----
    mov      di,offset buffer   ;point to buffer
    mov      cx,20               ;count is 20
    mov      al,55h              ;set al=55h
    cld
    rep     stosb              ;store al in buffer
```


Microcode Machine Example

Microprogramming is a technique used to implement a given machine instruction set architecture. It is not visible to the assembler language programmer; it is considered as part of the hardware of the computer. Each machine instruction is implemented by executing a microprogram which is comprised of one or more microinstruction. Each microinstruction controls the flow of data inside the machine and the actions of the hardware components such as the ALU. Microinstructions activate the real hardware in the ALU.

The microcode defines the personality of the computer. If you can change the microcode, then you can allow the computer to execute any format of machine/assembler instruction you desire. It is very powerful.

The idea for showing a simple microcoded machine came from Andrew Tanenbaum's Structured Computer Organization 2nd Edition Prentice-Hall, Inc 1984. We present a very simplified example of a micropogrammed machine that might be used in a vending machine or small home appliance.

The example CPU has these characteristics:

- Supports 256 bytes of main memory.
- Has two programmer accessible registers, an accumulator similar to ax and an index register similar to si.
- The machine has two formats of assembler/machine instructions. One format of instruction has only a one-byte operation code. This type instruction would be used for instructions that only use registers. In other words, the opcode bits would specify the operation to be performed and any register operands (accumulator or index).



- clear acc ;acc=0
- inc index ;index=index+1
- dec acc ;acc=acc-1
- add index,acc ;index=index+acc
- sub acc,index ;acc=acc-index

} Sample assembler instructions that could be converted into a machine instruction that only has a one byte opcode.

- The second format of instruction allows reference to one memory location or an immediate operand. It has a one-byte operation code and a one-byte memory address or immediate field. This type instruction might be used for adding a byte of data that is stored in memory to a register. Again, the opcode bits would specify the operation to be performed and any register operands (accumulator or index) and the operand would specify the memory location or immediate value.



- load acc,[var] ;acc=[var]
- add index,[var] ;index=index+[var]
- load index,25 ;index=25
- sub acc,[var] ;acc=acc-[var]

} Sample assembler instructions that could be converted into a machine instruction that has a one byte opcode and one byte operand.

The machine has eight internal registers, each one byte large. Only the Accumulator and Index Register are known to the assembler language programmer. All the other registers are internal to the machine, and are only available to the microprogrammer.

- The *Program Counter* holds the address of the next sequential instruction to be executed.
- The *Instruction Register* has two parts.
The OPCODE part holds the opcode of the instruction to be executed.
The OPERAND part holds the address of a variable or immediate data.
- There are two constants, zero and one.
- The *Accumulator* is used for arithmetic operations.
- The *Index Register* is used to step through memory and can also be used for arithmetic operations.
- The *Microcode Work Register* is available for the microprogram to hold temporary values.

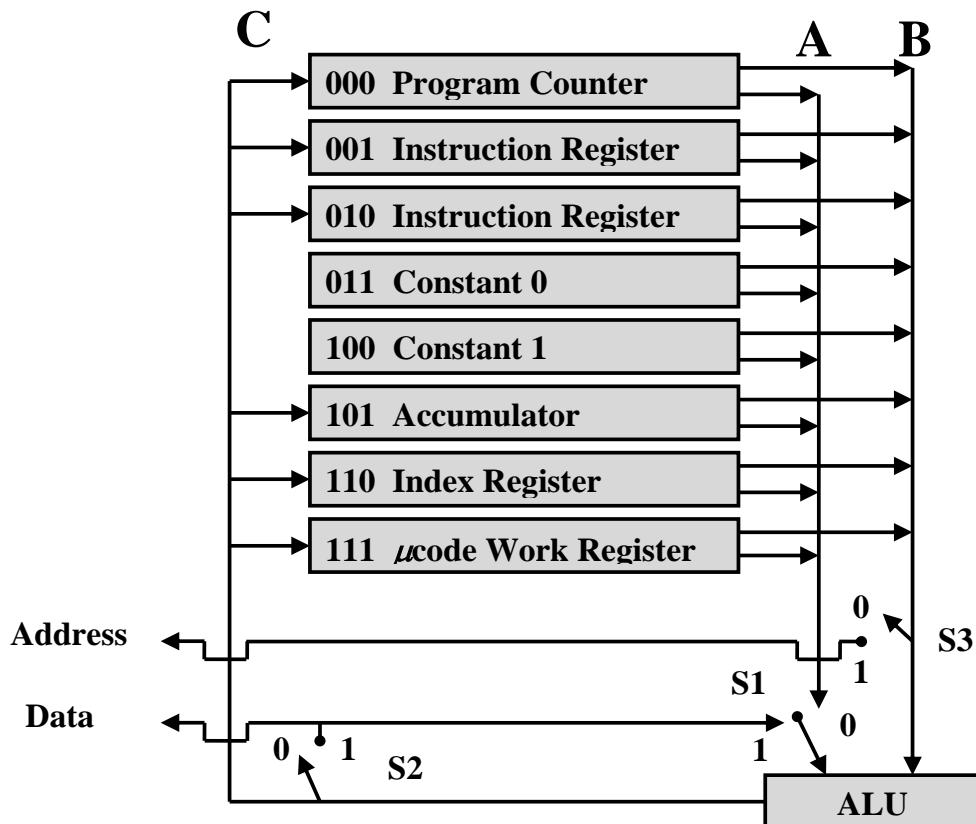
The ALU can perform four operations:

- Add two 8 bit numbers.
- Boolean-and two 8 bit numbers.
- Take the one's complement of an 8 bit number.
- Pass an 8 bit number through without change.

There are three internal buses used for moving data.

- The A and B buses feed the ALU, and the C bus carries the output of the ALU to its destination.

A set of three switches (S1, S2, S3) control the data flow.

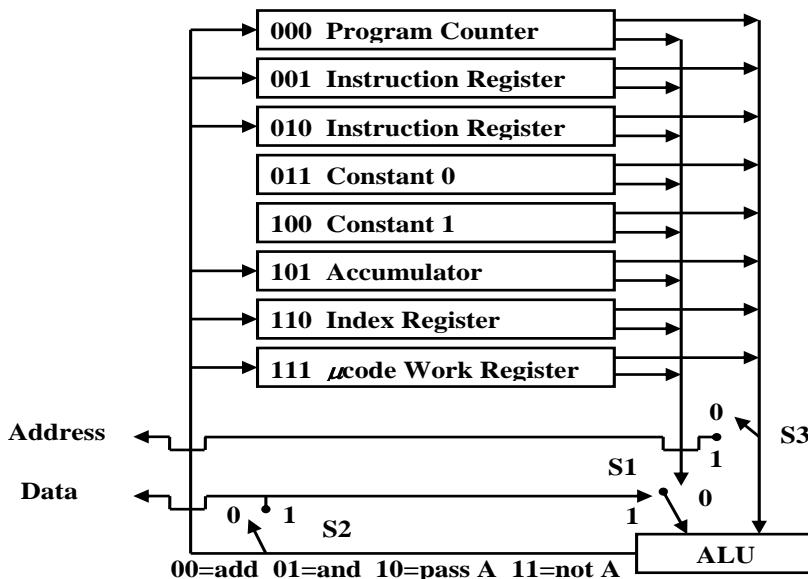


A logical view of the CPU chip

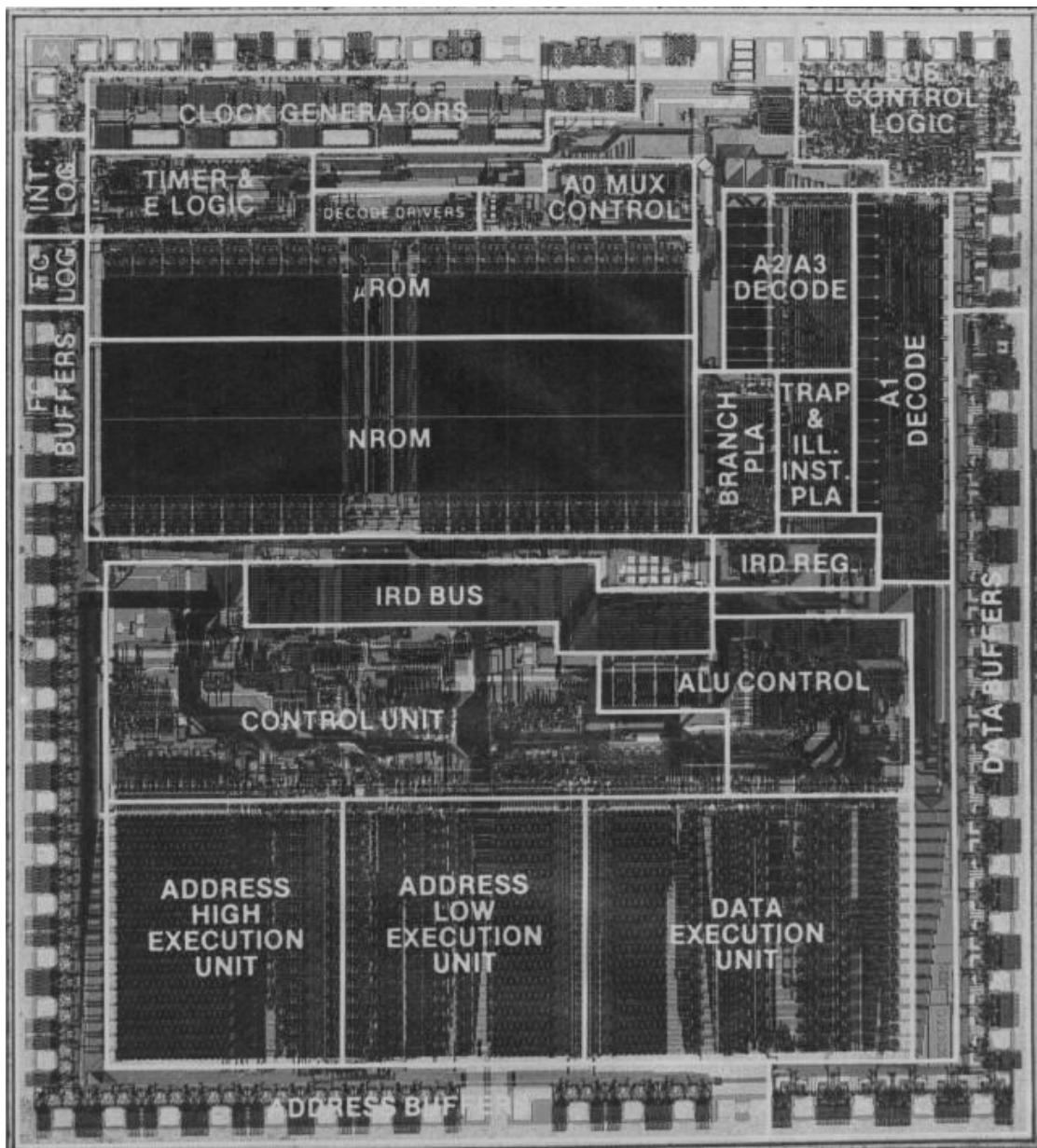
Everything is controlled by a 16 bit micro instruction with this format.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1	ALU	S2	S3	AV	R/W	A BUS		B BUS		C BUS					

- S1 Controls which is the 'A' input to the ALU
 0 = the internal 'A' bus is routed to ALU
 1 = the external data bus is routed to ALU
- ALU Controls the ALU operation (what operation the ALU performs with 'A' & 'B' inputs)
 00 = add (A+B)
 01 = Boolean *and* operation (A *and* B)
 10 = pass 'A' through without changing it
 11 = take the ones complement of 'A'
- S2 controls routing of the ALU output to the data bus
 0 = ALU output not placed on the external data bus
 1 = ALU output is placed on the external data bus
- S3 controls routing of the 'B' bus to the external address bus
 0 = 'B' bus not placed on the external address bus
 1 = 'B' bus is placed on the external address bus
- AV Address Valid tells memory if there is work for it
 0 = tells memory there is no work to perform
 1 = tells memory there is work to perform
- R/W 0 = tells memory this is a memory read operation (if AV = 1)
 1 = tells memory this is a memory write operation (if AV = 1)
- A BUS which register puts data onto the 'A' bus000 - 111 indicates the internal register source
- B BUS which register puts data onto the 'B' bus 000 - 111 indicates the internal register source
- C BUS which register receives the data on the 'C' bus 000 - 111 indicates the internal register destination



This is a picture of the Motorola 68000 chip which was a microcoded machine. You can see the Read Only memory that holds all the microinstructions ... µROM



MC68000

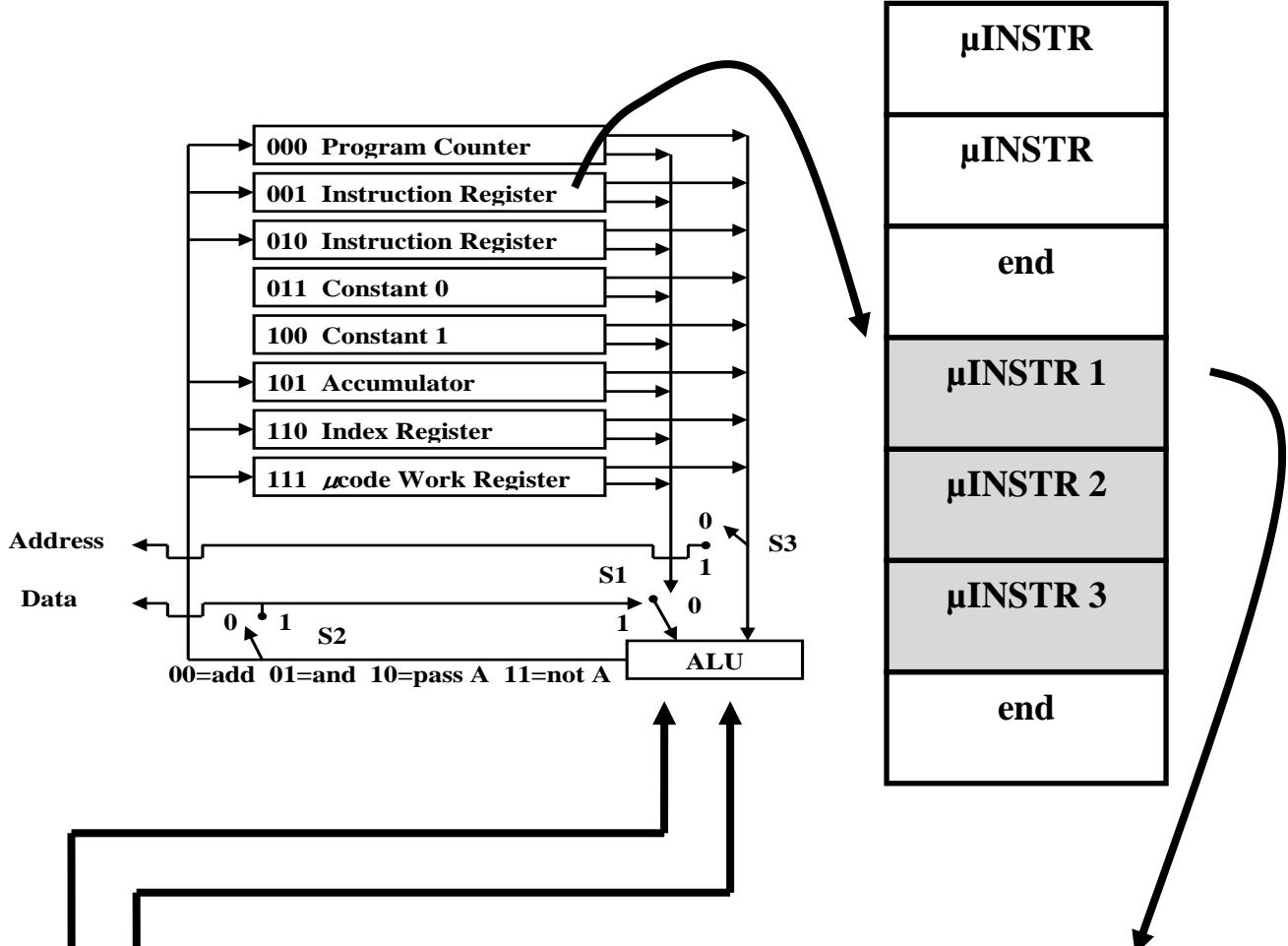
Executing a microinstruction effectively activates the control signals specified by the bits in the microinstruction. To execute the microinstruction, it is loaded into the Micro Instruction Register. The Micro Instruction Register has control lines that feed and control components such as the ALU and tell it which operation to perform.

For each assembler/machine instruction there will be a microprogram which consists of the required number microinstructions needed to implement the assembler/machine instruction.

Conceptually, the first half of the Instruction Register 001 contains the machine instruction's opcode. This would be used to index the system into the microinstruction store to get to the first microinstruction in the microprogram. Then microinstructions would be sequentially fetched into the Micro Instruction Register and executed. In a real machine it would be possible to branch in the microprogram, but for our simple example, the microinstructions in a microprogram will just be executed sequentially.

All these operations would be synchronized by the computer's clock.

μINSTR Store on chip ROM 16 bits



Micro Instruction Register

We present examples of the microinstructions used to implement assembler/machine instructions.

Example. The machine instruction *clear acc*

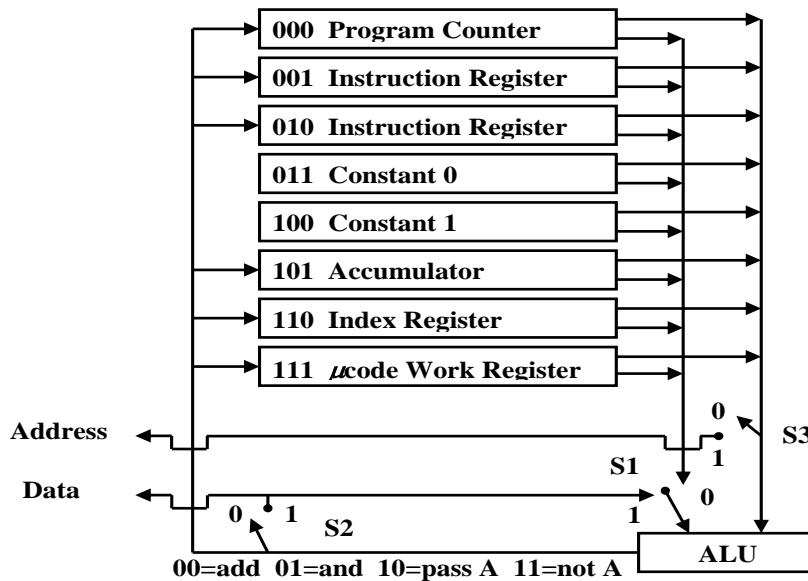
This machine instruction requires one microinstruction.

1. ALU = Constant Zero

- The constant zero (011) feeds the A bus.
- S1 is set 0 to allow the A bus to feed the ALU.
- The ALU is set to 10 to just pass the value on the A bus.
- The output of the ALU on the C bus feeds the accumulator (101).
- Memory is not involved so Address Valid is 0 which is false .
- S2, S3, R/W and the Bus are set to zero. They are not used for this instruction.

0	1 - 2	3	4	5	6	7 - 9	10 - 12	13 - 15
S1	ALU	S2	S3	AV	R/W	A Bus	B Bus	C Bus

symbolic	A to ALU	pass A	-	-	F	-	zero	---	ACC
acc = 0	0	10	0	0	0	0	011	000	101



Example. The machine instruction *add acc,[var]*

This machine instruction requires two microinstructions.

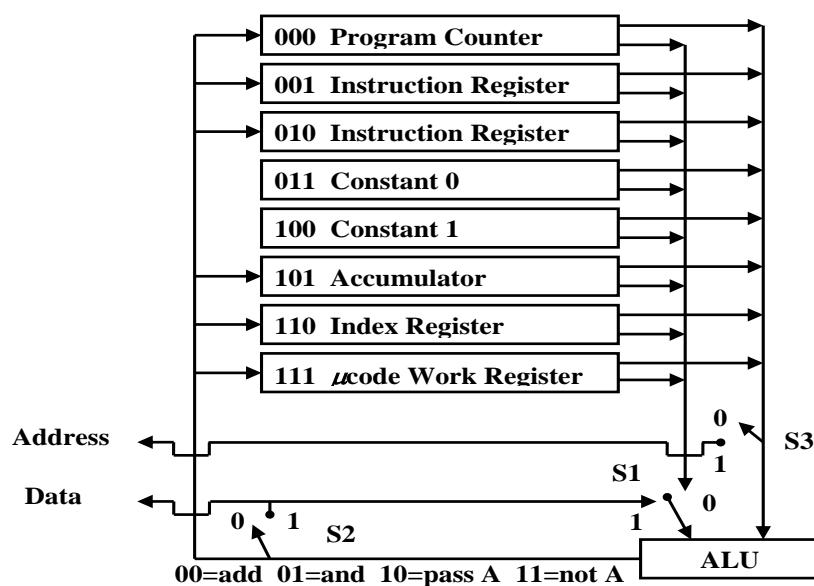
1. Fetch the variable [var] to the µcode work register.

- The second byte of the IR (010) has the address of [var] so it feeds the B bus.
- Switch 3 is set to 1 so the B bus goes onto the Address Bus
- S1 is set to 1 to the value of [var] coming in on the Data Bus feeds the ALU.
- The ALU is set to 10 to pass through the value on the A bus.
- The C bus feed the µcode work register (111).
- Address Valid is set to 1 so memory knows there is work for it.
- R/W is set to 0 to read the data.
- The A bus is set to 000 and is not used for this operation.

2. Add the µcode work register to the accumulator

- The A bus is fed by the accumulator (101).
- The B bus is fed by the µcode work register (111).
- Switch 1 is set to 0 so the A bus feeds the ALU.
- The ALU is set 00 to add the two input values.
- The C bus is set to 101 to feed the accumulator.
- S2, S3, R/W and the Bus are set to zero. They are not used for this instruction.

	0	1 - 2	3	4	5	6	7 - 9	10 - 12	13 - 15
	S1	ALU	S2	S3	AV	R/W	A Bus	B Bus	C Bus
symbolic									
work = [var]	1	10	0	1	1	0	000	010	111
acc = acc+work	0	00	0	0	0	0	101	111	101



Example. The machine instruction dec acc

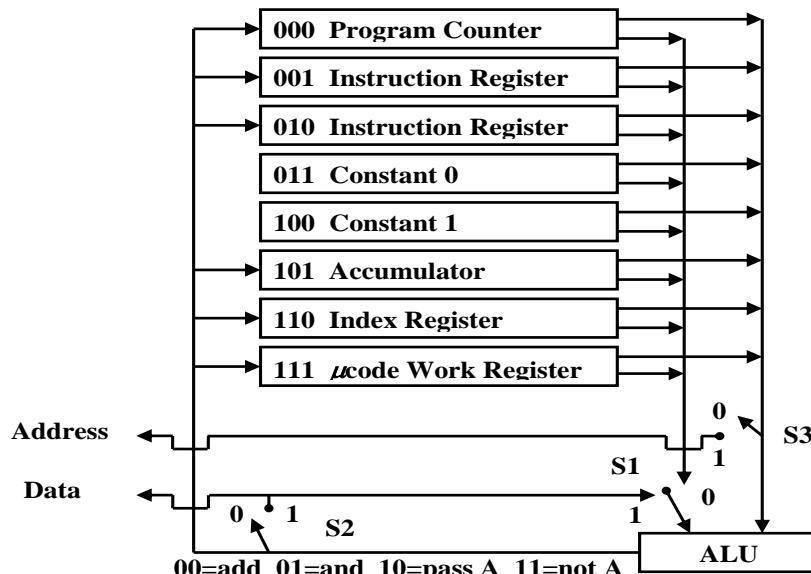
The assembler instruction `dec acc` performs the operation $acc = acc - 1$.

However, the ALU does not perform subtraction. Instead, the operation will be performed as: $acc = acc + (-1)$

Since the constant -1 is not built into a register, we need to generate it. The standard way to do that is to start with the constant 1 and invert its bits and add 1 . $-1 = (\text{flip the bits of } 1) + 1$

This process results in using 3 microinstructions to implement `dec acc`.

	0	1 - 2	3	4	5	6	7 - 9	10 - 12	13 - 15
symbolic	S1	ALU	S2	S3	AV	R/W	A Bus	B Bus	C Bus
work = not 1	0	11	0	0	0	0	100	000	111
work = work + 1 = -1	0	00	0	0	0	0	100	111	111
acc = acc + work = acc + (-1)	0	00	0	0	0	0	101	111	101



But wait ... with a little thought we can generate -1 with only a single microinstruction. If we just invert the bits of zero (00000000) it will generate -1 (11111111). This gives a more efficient implementation of `dec acc`.

	0	1 - 2	3	4	5	6	7 - 9	10 - 12	13 - 15
symbolic	S1	ALU	S2	S3	AV	R/W	A Bus	B Bus	C Bus
work = not 0 = -1	0	11	0	0	0	0	011	000	111
acc = acc + work = acc + (-1)	0	00	0	0	0	0	101	111	101

Example. The machine instruction $sub [var], acc$

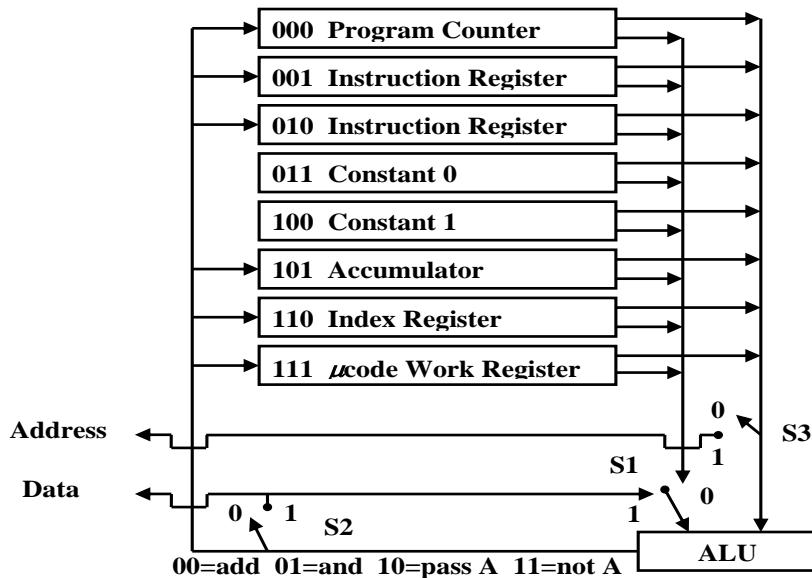
The assembler instruction $sub [var], ax$ will be done as $var = var + (-ax)$

This implementation requires two temporary work registers.

- Fetching $[var]$... needs a work register
- Negating ax ... needs a work register

However, we only have 1 work register. There are multiple potential solutions. The one selected is to use Instruction Register (001) as a temporary work register. It contains the instruction opcode and once we have used that to point to the microprogram, it is no longer needed. We cannot use the Instruction Register (010) since that contains the address of the variable $[var]$ which is needed throughout the instruction's execution. This process results in 5 microinstructions.

Symbolic Microinstruction	Value	Binary Microinstruction								
		S1	ALU	S2	S3	AV	R/W	A BUS	B BUS	C BUS
1. work = $[var]$	var	1	10	0	1	1	0	000	010	111
2. $IR_{001} = \text{not } ax$	ax	0	11	0	0	0	0	101	000	001
3. $IR_{001} = IR_{001} + 1$	-ax	0	00	0	0	0	0	001	100	001
4. work=work+IR ₀₀₁	var - ax	0	00	0	0	0	0	111	001	111
5. $[var] = \text{work}$	var - ax	0	10	1	1	1	1	111	010	111



Using the machine layout you can verify the binary microinstructions.

We analyze the last microinstruction since it is the first time we have written data to memory.

$[var] = \text{work}$

- Second byte of IR (010) with the address of $[var]$ goes on the B bus.
- S3 is set to 1 so the B bus goes out on the Address Bus.
- Address Valid is set to 1 so memory knows it has work.
- R/W is set to 1 to write data to memory.
- The μ code work register (111) with the data to be written feeds the A bus.
- Switch 1 is set to 0 so the A bus feeds the ALU.
- The ALU is set to 10 to pass the data through.
- Switch 2 is set to 1 so the data goes out on the Data Bus.
- The C bus is set to 111 to just reload the μ code work register and not modify another register.

The assembler instruction $sub [var],ax$ continued.

To demonstrate the power of human intelligence, a student was able to write the microprogram for this assembler instruction in three microinstructions.

They used the following equality.

We know that in two's complement $\overline{-X} = \overline{X} + 1$

We can now rearrange the terms to see that ... $\overline{X} = \overline{-X} - 1$



Symbolic Microinstruction	Value	Value based on equality above
<ul style="list-style-type: none"> • work = not var (bring in the value of var and invert the bits and store it in the work register) 	$\overline{\text{var}}$	$-\text{var} - 1$
<ul style="list-style-type: none"> • work = work + ax (add ax to the one's complement of var in the work register) 	$\overline{\text{var}} + \text{ax}$	$-\text{var} - 1 + \text{ax}$
<ul style="list-style-type: none"> • var = not work (set var to be the one one's complement of the work register) 	$\overline{\overline{\text{var}}} + \text{ax}$	$ \begin{aligned} & -(-\text{var} - 1 + \text{ax}) - 1 \\ & = \text{var} + 1 - \text{ax} - 1 \\ & = \text{var} - \text{ax} \end{aligned} $

Ideas for further thought.

- We did not discuss unconditional or conditional jumps.

If we had the unconditional jump instruction *jmp [lab]* then the address of the target instruction *lab* would be in Instruction Register 010 and the symbolic micro instruction to execute the jump would be Program counter = Instruction Register 010.

To implement conditional jumps we would need to add to our machine:

- A set of flags such as the Sign Flag, Zero Flag, Overflow Flag, Carry Flag.
- The hardware logic to set those flags.
- The hardware logic to test those flags.

- Getting another programmer accessible register.

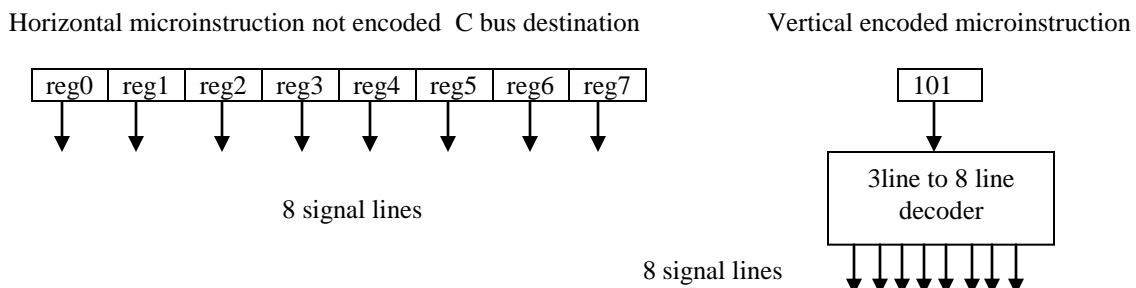
The example machine only has eight internal registers. Two are used for the constants zero and one. Instead of using a register for the constant zero, let's use that register for a second programmer accessible register. Now if we want the constant zero we can create it from the constant one and the four ALU functions (add, and, one's complement, pass).

Using 8 bits, show the symbolic microinstructions that can create the value 00000000 from the value 00000001 only using the four ALU functions.

1. μcode work register = not constant 1 = 11111110
2. μcode work register = μcode work register *and* constant 1 = 11111110 *and* 00000001 = 00000000

- There are two forms of microprogramming, *vertical* which tends to encode information and *horizontal* which tends not to encode information. Our example machine uses vertical microcode. For example, the microinstruction uses 3 bits to indicate which register is fed by the C bus. Only one register can be encoded in those 3 bits and thus only one register be fed by the C bus at a time. If we wanted to clear all the registers to zero we would need to have one microinstruction (that passes zero to the C bus) for each register to be cleared.

With horizontal microcode, we would not encode the register number (0-7) in 3 bits. Instead we would have a unique bit for each register that can be fed by the C bus. Theoretically that would be 8 bits. Now we can clear all the registers with one microinstruction. Any bit that was set to 1 would mean that register is fed by the C bus.



Vertical microcoding will take longer to execute since the signals must pass through additional logic gates to convert the encoded value to a set of signal lines.

Horizontal microcode tends to be more challenging to code but also offers more opportunity to perform operations in parallel thus improving performance.

References

- Andrew Tanenbaum's Structured Computer Organization 2nd Edition Prentice-Hall, Inc 1984.
A Brief History of Microprogramming, Mark Smotherman
Microprogramming Concepts, Alan Clements

IEEE Floating-point

This paper presents an *introduction* to Floating-point arithmetic. To fully cover Floating-point, a full term dedicated course is required. However, we can provide a limited subset of instructions and capabilities. It is enough to allow you to write basic Floating-point code. A good place to start for an external reference is Wikipedia which has a good introduction to the IEEE 754-2008 standard. If you need more detail on specific topics such as exception handling, you should obtain a full copy of the standard document.

Introduction

Real numbers are divided into two groups, rational and irrational.

- Rational numbers are those that can be represented by the ratio of two numbers.
Rational numbers include all the integers (1/1, 2/1, 3/1, ... 73/1).
Rational numbers include all the fractions (1/2, 2/3, 3/4, ...).
- Irrational numbers are those that cannot be represented by a ratio of two numbers.
Irrational numbers include PI and the square root of 2.

In CSC23x we mostly work with finite precision (byte and word) fixed-point integers (unsigned and two's complement). However there are many problems that need more than finite precision fixed-point arithmetic can provide.

- Calculating the square root of 2 to be 1.4142135623730...
- Calculating the mass of a star to be 2^{40} Kilograms.

For these types of calculations we must use a different method, Floating-point, which is able to represent an extremely wide range of rational and irrational numbers. This representation, also called scientific notation, has this generic format ...

$$\pm n \times 10^e$$

... where n is a decimal fractional number and e is an exponent. For example $+ 7.356 \times 10^3$

Since computers use binary numbers, computers use this format: $\pm n \times 2^e$ where n is a binary fractional number and e is an exponent. For example: $+ 1.01 \times 2^4$ which becomes binary 10100 which is decimal 20. That same number 1.01×2^4 can have multiple representations. For example: 10.1×2^3 $101. \times 2^2$ 1010×2^1 10100×2^0 etc..

A binary Floating-point number is said to be *normalized* if the fraction is in this range ($2 > \text{fraction} \geq 1$). So 1.01×2^4 is the *normalized* way to store the decimal value of 20.

The term *Floating-point* was adopted because the binary point floats to the position immediately after this non-zero first digit.

With a normalized number:

- You do not need to actually store the binary point.
- It is assumed that the left most bit is 1 and this is followed by the binary point which is followed by the remaining bits. So the normalized fraction 101 is assumed to be 1.01.
- Since we know the left most bit is *always* 1 we do not even have to store it. This is called *hidden bit normalization*. Using this technique, only the fraction 01 is stored. When retrieved, the fraction 01 is assumed to be 1.01 where we have added the hidden 1 bit and the binary point. This technique gives you one more bit of precision and allows more information to be stored in the data field.

However, the hidden bit format has a problem. There is no way to *actually* represent zero. If we store the fraction as four bits then the value 0000 would be assumed to be 1.0000. We will see how this is handled by the IEEE standard.

The IEEE 754-1985 standard was the first version adopted in 1985. The current version was adopted in 2008, IEEE 754-2008. This latest standard defines 32 bit Single Precision, 64 bit Double Precision and 128 bit Quadruple Precision formats.

Warning: Floating-Point arithmetic is inherently inexact. Single precision supports about 7 decimal digits. If a number requires more digits, it is approximated. This can cause unexpected results if the programmer is not very careful.

IEEE Floating-point standard 754

The precision of a Floating-point number is the number of bits used to hold the fraction (including the hidden bit). We will only look at the format for single precision numbers. These are 32-bit values with the format shown below. The IEEE standard uses the term *significand* instead of *fraction* to remind you that it is in hidden bit normalized format.

This how the 32 bits are allocated to a single precision word.

1	8	23
sign	exponent	significand

sign	0 = positive 1 = negative
exponent	The exponent is stored in <i>excess 127 format</i> . The value stored in this field is equal to the desired power plus 127. To calculate the power, you subtract 127 from the value stored. This means a number raised to the zero power will have an exponent value stored of 127. A value of 125 means that the power is $125-127 = -2$ A value of 130 means that the power is $130-127 = +3$ The exponent values of 0 and 255 are reserved for special uses. See the Miscellaneous Information section for more detail.
significand	The significand has an implied 1 bit in front of it, thus yielding an effective 24-bit significand (24 bit precision). This is called the <i>hidden bit normalized</i> . This means to store a number, it is first normalized so that it is of the form 1.xxxxx... then the 1 bit is dropped (it will be represented by the hidden bit) and xxxx... is stored as the significand.

Let's look at one example, the decimal number 511.1

1. The number is shown as a hex value and 32 bit binary value.
 2. Then it is shown in binary, separated into the three fields: sign, exponent, significand.
 3. Then the hidden 1 bit is added and precedes the significand and binary point.
The hidden 1 bit is shown in bold italic as ***I***.
 4. The exponent is then used to move the binary point and finally we calculate the decimal value created from the final binary number.

Many things to consider

- **Single Precision Digits**

With single precision floating point numbers you can handle approximately 8 decimal digits. For any number, those digits can be used for the integer portion or fraction or both. However, remember that you only have that limited number of digits. If you try to create numbers that need more than the supported digits, the assembler/compiler will create a constant that just *approximates* the desired value. The three examples below show this issue.

Data declaration	Hex constant created	Actual decimal value of the hex constant	Notes
v1 dd 0.987654321 (Too many fractional digits)	3F7CD6EA	0.98765433	Handle 7 digits, all used for the fraction
v2 dd 12345.987654321 (Too many combined integer plus fractional digits)	4640E7F3	12345.98730469	Handle 8 digits, 5 used for the integer and 3 for the fraction.
v3 dd 12345678.987654321 (Too many combined integer plus fractional digits)	4B3C614F	123456789	Handle 9 digits, all used for the integer. No digits were allocated to the fraction.

- **Decimal Fractions Versus Binary Fractions**

There are decimal fractions that cannot be exactly represented by binary fractions. For example the decimal value 0.1 has no exact binary equivalent. This combined with the fact that you only have about 8 digits will cause problems when you try to perform comparisons between numbers.

Data declaration	Hex constant created	Actual decimal value of the hex constant	Notes
a dd 1000.1	447A0666	1000.09997559	9 of the 23 significand bits and the hidden bit are used by the integer 1000. This leaves 14 significand bits to represent the fraction .1 and that is not enough and so we start to lose accuracy.
b dd 1000.0	447A0000	1000.0	No problem representing 1000.0
c dd 0.1	3DCCCCCD	0.100000001	Since all digits are used by the fraction, the result is closer to 0.1 (one tenth).
d = a - b	3DCCC000	0.099975586	Although all the significand bits are available for the fraction, we unfortunately lost accuracy when we did the calculation 1000.09997559 - 1000.0

The problem now is that both c and d should be 0.1 but they will not compare as equal!

If we run a C program that calculates $1000.1 - 1000.0$ we get the hex single precision result of 3DCCC000 which is decimal 0.099975586 and although close to 0.1 it is not exactly the same.

The comparison indicates *c is not equal to d* even when we expect both values to be 0.1 (one tenth).

C program

```
#include <stdio.h>
int main()
{
    float a, b, c, d;
    int result;

    printf("Enter 'a' floating point number x.1 : "); // read a
    result = scanf("%f",&a);

    printf("Enter 'b' floating point number x.0 : "); // Read b
    result = scanf("%f",&b);

    printf("Enter 'c' their difference      0.1 : "); // Read c
    result = scanf("%f",&c);

    printf("a = %.9f\n",a); // print a
    printf("b = %.9f\n",b); // print b
    printf("c = %.9f\n",c); // print c

    d = (float)a-b; // d = a - b
    printf("d = a - b\n");
    printf("d = %.9f\n",d); // print d

    if (d == c) printf("x.1 - x.0 == 0.1 \n"); // test whether
    else         printf("x.1 - x.0 != 0.1 \n"); // c == d

    return (0);
}
```

Results of running that C program

```
Enter 'a' floating point number x.1 : 1000.1
Enter 'b' floating point number x.0 : 1000.0
Enter 'c' their difference      0.1 : 0.1
a = 1000.099975586 ←
b = 1000.000000000
c = 0.100000001

d = a - b
d = 0.099975586
x.1 - x.0 != 0.1
```

9 of the 23 significand bits and the hidden bit are used by the integer 1000. This leaves 14 significand bits to represent the fraction .1 and that is not enough and so we start to lose accuracy.

There is a special way to handle this inequality that allows the comparison to work as desired. It is called *safe testing* and we discuss it next.

- **Safe Testing**

You have two values and you wish to determine whether they are equal.

For example A = 1000.1 and B = 1000.0 and we calculate C = A - B.

We want to know if C == 0.1 and from the last topic we saw that C will not compare equal to 0.1

Instead we establish a value, epsilon, with a small value such as 0.0001

Write the code that indicates C equals 0.1 if the difference between C and 0.1 is less than epsilon.

```
c = a - b;                                // c = a - b;
e = (c - 0.1); if (e < 0.0) e = -e;        // e = absolute value of (c - 0.1)
epsilon = 0.0001;                           // acceptable difference = 0.0001
if (e < epsilon) printf("c == 0.1 \n");    // if c is close enough to 0.1
else           printf("c != 0.1 \n");       // then decide c == 0.1
```

- **Rounding**

The standard provides five ways to round. The first is the most common and the default.

Type of rounding	Description
Round to nearest (ties to even)	Rounds to the nearest value. If the number falls exactly midway then it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time. This is the default rounding process.
Round to nearest (ties away from zero)	Rounds to the nearest value. If the number falls midway then it is rounded to the nearest value above for positive numbers or below for negative numbers.
Round toward 0	Rounds towards zero. Also known as truncation.
Round toward $+\infty$	Rounds towards positive infinity. Also known as rounding up or ceiling.
Round toward $-\infty$	Rounds towards negative infinity. Also known as rounding down or floor.

- **Special Numbers**

There are some special values defined by the standard.

Value	Sign	Exponent value stored	Significand	Hex value
zero	+/-	0	0	00000000, 80000000
+infinity $+\infty$ This represents the maximum positive real number.	+	255	0	7F800000
-infinity $-\infty$ This represents the most negative real number.	-	255	0	FF800000

If an operation overflows or an operation divides by zero then the result will be one of the infinities.

You can compare other numbers to $+\infty$. It will be greater than any finite number.

You can compare other numbers to $-\infty$. It will be less than any finite number.

	Sign	Exponent value stored	Significand	Hex value
NaN - Not a Number This is an error pattern created when an illegal operation is attempted, such as ... $\sqrt{-2}$ There are two types on NaN. Quiet NaN may be used as an operand without generating an exception. Another name for the quiet NaN is an indefinite. The result of an illegal operation will usually be a quiet NaN (square root of negative number, $0 \div 0$, $0 * \infty$, $\infty - \infty$, stack error, ...). If a quiet NaN is used as an argument, the result of the operation will be a quiet NaN. So the program can continue, but that may not be too useful. Signaling NaN will cause an exception if used as an operand. An example is an uninitialized variable may be set to a signaling NaN. Thus any attempt to use an uninitialized variable will cause an exception.	+/-	255	1000... 0100...	7FC00000 7FA00000

The matrix of results generated when using NaN or infinity as operand is large and complex.

If you need to see that matrix, you will need to find a detailed reference.

- **Exceptions**

Exception handling can very complex. For complete details you need to see the full standard.

These are the most common exceptions.

For most programs the default action taken by the hardware will suffice.

One can, however, write special exception handling code if it is needed.

Exception	Description / example	Default (masked) action <i>(This includes setting a bit in the NDP status word so the programmer can determine that a specific exception occurred.)</i>
Invalid operation	<p>These are usually a programming error.</p> <p>Examples:</p> <p>Square root of a negative number.</p> <p>Push when the stack is full.</p> <p>Pop when the stack is empty.</p> <p>An operand is NaN.</p>	Return NaN.
Denormal	Denormalized operand was fetched.	Use the denormal number and try to continue.
Division by zero	Attempt to divide by zero.	Return $\pm\infty$
Overflow	Creating a result too large to be represented correctly.	Return $\pm\infty$
Underflow	Creating a result too small to be represented correctly	You might expect that the result returned would be 0. However, this would be an <i>abrupt</i> underflow and would not allow the program to effectively continue. Instead, the system attempts a <i>gradual</i> underflow and returns the <i>denormalized</i> result. See the Miscellaneous Information section for more details.
Precision	Creating a result that cannot be exactly represented such as 0.1	Return the rounded result.

- **Miscellaneous Information**

- The exponent values stored of 0 and 255 are used for special purposes.

Thus the largest normalized number that can be represented is $1.99\dots \times 2^{127}$
 The smallest normalized number that can be represented is $1.00\dots \times 2^{-126}$

An exponent value stored of 0 is used to mean that the number is not a normalized number.
 It represents the power 2^{-126} but also that the significand does not have a hidden bit.
 When the significand is all zero then the number is defined to be zero.

An exponent value stored of 255 is used to represent infinity or NaN.
 Infinity has a zero significand while NaN has a non-zero significand.

- Denormalized results

Numbers are usually stored in normalized format. This maximizes the number of significant digits that can be stored.

An exception is when an *underflow* occurs, where the result is too small to be represented in the destination format. For example, the single precision result should be 1.25×2^{-129} which is outside the exponent range of -126 to +127.

You might expect that when an underflow occurs (the true result is too small to be correctly represented) the result returned would be 0. However, this would be an *abrupt* underflow and would not be acceptable in trying to continue the program. Instead, the system attempts a *gradual* underflow. The system goes through a process of trying to make the result fit by using some of the significand bits to hold leading zeroes, thus allowing smaller numbers to be represented. Here is an example where the true result should be 1.25×2^{-129} .

Description	Exp	Significand	Note
True desired result	-129	1.010000	Normalized significand and an exponent will not fit.
Step 1 - Denormalize	-128	0.101000	Shift binary point left & reduce exponent.
Step 2 - Denormalize	-127	0.010100	Shift binary point left & reduce exponent.
Step 3 - Denormalize	-126	0.001010	Shift binary point left & reduce exponent.
Denormal result returned $.00125 \times 2^{-126}$	-126	0.001010	Denormal significand & exponent fits.

The value usually returned for an exponent of -126 would be $(-126 + 127 = 001)$. However, we need to indicate this number is not normalized. This is done by returning the exponent value of 0. This indicates the exponent is -126 but that the significand does not have a hidden bit. This is important since any operation using this denormal number must know there is not a hidden bit. It is *possible* to continue processing using a denormal result but denormals have reduced significance and that may be an issue with the program.

If the true desired result is too small to be denormalized, then a result of zero will be returned

- Signed zeroes will compare equal. That is $+0 = -0$.
- There are two models for infinity.

Projective mode treats $+\infty$ and $-\infty$ as a single unsigned infinity.

Affine mode uses the sign of the infinity and the two values will not compare equal.

The default is projective mode.

Additional examples of how Floating-point numbers are stored

1	8	23
sign	exponent	significand

sign	0 = positive 1 = negative
exponent	<p>The exponent is stored in <i>excess 127 format</i>. The value stored in this field is equal to the desired power plus 127. To calculate the power, you subtract 127 from the value stored. This means a number raised to the zero power will have an exponent value stored of 127. A value of 125 means that the power is $125-127=-2$ A value of 130 means that the power is $130-127=+3$</p>
significand	<p>The significand has an implied 1 bit in front of it, thus yielding an effective 24-bit significand. This is called the <i>hidden bit normalized</i>. This means to store a number, it is first normalized so that it is of the form $1.\text{xxxx}$ then the 1 bit is dropped (it will be represented by the hidden bit) and xxxx is stored as the significand.</p>

Dec.	Hex	Binary
+1.0 =	3F800000	= 0011 1111 1000 0000 0000 0000 0000 0000 (float pt value)
		3F800000 = 0 01111111 00000000000000000000000000000000 (fields)
		 $+ 127-127=2^0 \quad 1.00000000000000000000000000000000$ (with hidden 1)
		$+ 1.00000000000000000000000000000000 \times 2^0$ (fields combined)
		$+ 1.00000000000000000000000000000000$ (binary number is 1.0)

Dec.	Hex	Binary
-2.0 =	C0000000	= 1100 0000 0000 0000 0000 0000 0000 0000 (float pt value)
		C0000000 = 1 10000000000000000000000000000000 (fields)
		 $- 128-127=2^1 \quad 1.00000000000000000000000000000000$ (with hidden 1)
		$- 1.00000000000000000000000000000000 \times 2^1$ (fields combined)
		$- 10.00000000000000000000000000000000$ (binary number is -2.0)

Dec.	Hex	Binary
0.333...	= 3EAAAAAB	= 0011 1110 1010 1010 1010 1010 1010 1011 (float pt value)
	3EAAAAAB = 0 01111101 010 1010 1010 1010 1010 1011 (fields)	
	+ 125-127=2 ⁻² 1.01010101010101010101011 (with hidden 1)	
	1.01010101010101010101011 x 2 ⁻² (fields combined)	
	.01010101010101010101011	(binary number)
	1/4 + 1/16 + 1/64 ... = .250000 + .0625 + .00390625 + ...	
	= 0.33333334	

With a hidden 1 bit there is no way to "actually" represent the value 0.0

This is a very very very small number, but it is not exactly zero. To bypass this problem, the IEEE standard defines that value as zero.

References

- The Perils of Floating Point, Bruce M. Bush, Lahey Company
 - IEEE-754-2008, Wikipedia
 - iAPX 86,88 User's Manual, Intel Corporation
 - Numerical Computing with IEEE Floating Point Arithmetic, Michael L. Overton, SIAM

How To Use The Floating-Point Co-Processor

Floating-point calculations are done in a *separate* CPU.

On older Intel systems this was a completely separate chip called the 8087 Numeric Data Processor (NDP). For this paper we will use the term x86 to represent the standard Intel CPU which includes the 8086 through the Core Architecture processors, and we will use the term NDP to represent the 8087 Floating-point CPU.

Starting with the 80486, the NDP was placed on the same physical chip as the x86. However, it is still logically a separate CPU and this means we need to have a set of protocols for passing information between the x86 and the NDP.

Passing data between the x86 and NDP

Information is passed between the x86 and the NDP via a Floating-point register stack. This stack contains eight registers. The current value of the top of the stack is referred to as Stack Top (ST). Remember the stack grows downward.

ST(2) --->	Current Stack Value - 2
ST(1) --->	Current Stack Value - 1
ST --->	Current Stack Value

We will use single precision 32-bit operations, so view the ST register as having that 32 bit format.

Like a classic stack machine . . .

- A push decrements the ST pointer and then places data into the register.
- A pop removes data from the current register and increments the ST pointer.
- When the system is initialized, the ST pointer will be just above the first register. This means the first operation must be a push, which will move ST to the top register in the stack.

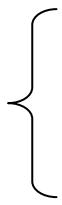
Setting the NDP Control Word

The NDP control word allows the user to set: the precision to be used; the type of rounding to be used; how you want infinity to be handled ;whether you want to take the default action when an exception occurs (exception is masked) or whether you want to provide special exception handling routines (exception is not masked).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			IC	RC	RC	PC	PC			PM	UM	OM	ZM	DM	IM

IC – Infinity Control

- 0 Projective
- 1 Affine


 There are two models for infinity.
Projective mode treats $+\infty$ and $-\infty$ as a single unsigned infinity.
Affine mode uses the sign of the infinity and the two values will not compare equal.
 The default is projective mode.

RC – Rounding Control

- 00 Round to nearest. This is the default and most common option.
- 01 Round down to $-\infty$
- 10 Round up to $+\infty$
- 11 Truncate to zero

PC – Precision Control

- | | | |
|----|-----------------------------|--|
| 00 | 32 bit Single Precision | (sign=1 exponent=08 significand=23) |
| 01 | Reserved | |
| 10 | 64 bit Double Precision | (sign=1 exponent=11 significand=52) |
| 11 | 128 bit Quadruple Precision | (sign=1 exponent=15 significand=112) |

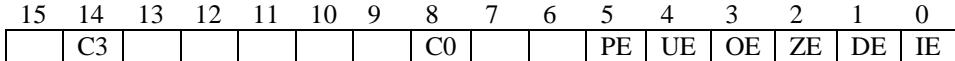
Exception Control (1=masked which means accept the default action)

Code	Exception	Description / Example	Default Action
IM	Invalid operation	Usually a programming error such as taking the square root of a negative number.	Return NaN.
DM	Denormal	Denormalized operand fetched.	Use the value and continue the program.
ZM	Division by zero	Tried to divide by zero.	Return $\pm\infty$
OM	Overflow	The result too large to be represented correctly.	Return $\pm\infty$
UM	Underflow	The result too small to be represented correctly.	The system attempts a <i>gradual</i> underflow and returns the <i>denormalized</i> result. See the IEEE handout for more detail.
PM	Precision	The result cannot be exactly represented. For example the result should be 0.1	Return the rounded result.

Testing the NDP Status Word and testing Floating-point numbers and conditional jumps

Remember that the only program being run is in the x86. The NDP is strictly executing specific Floating-point operations passed to it by the x86. This makes testing the status of Floating-point operations, and conditional jumping based on the result, challenging. We present the basic process.

There is a 16-bit status word in the NDP. We are only concerned with two bits that show the result of comparing two Floating-point numbers. One number is the value in the ST(1) register and the other number is in ST. The comparison is done as $ST(1) : ST$ which is performed as $ST(1) - ST$.



This table shows how C0 and C3 are set and what the settings mean.

These 6 bits indicate whether the specific exception has occurred. See the NDP Control Word for an explanation of each exception.

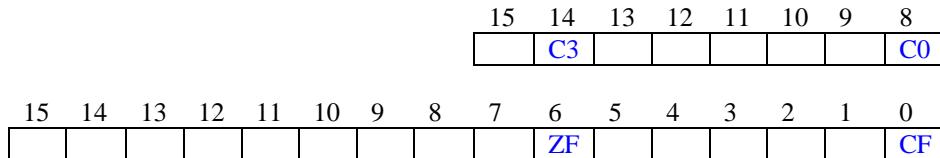
C3	C0	Meaning
0	0	$ST(1) > ST$
0	1	$ST(1) < ST$
1	0	$ST(1) = ST$
1	1	$ST(1) ? ST$

Nan and projective infinity cannot be compared. So if either is used as an operand, C3=C0=1 is returned.

Unfortunately, we have no way to *directly* test these C3 and C0 bits!

We must move these bits to the x86 condition code in order to perform conditional jumps.

If we move the upper byte of the NDP status register (bits 8-15) to the lower byte of the x86 status register (bits 0-7) then CF will equal C0 and ZF will equal C3 and we can test ZF and CF.



This move yields the following table, which allows us to use the x86 conditional jumps JA, JB, JE to test the relation between ST (1) and ST.

ZF	CF	Meaning	x86 Tests
0	0	$ST(1) > ST$	JA will be taken if $ST(1) > ST$
0	1	$ST(1) < ST$	JB will be taken if $ST(1) < ST$
1	0	$ST(1) = ST$	JE will be taken if $ST(1) = ST$

The next page and the sample programs show exactly how this test is coded.

Floating-point instructions

The NDP differs from a classic stack machine in that it allows arithmetic operations to have operands.

We will not use that feature and will treat the NDP as a pure classic stack machine.

To take advantage of the full power of the NDP you should obtain a separate reference.

This is the subset of Floating-point instructions you may use for this assignment.

In the examples below:

- [var] a single precision Floating-point variable, declared as var dd 1.414
- ST the current Floating-point Stack Top register
- [status] a binary word to hold the NDP status, declared as status dw 0

Data Movement	Explanation
FLD [var]	Load Real \equiv push var is pushed onto the stack so that ST = var
FSTP [var]	Store Real \equiv pop ST is popped off the stack and stored in var so that var = ST

Arithmetic	Explanation
FADD	Add Real Pop ST and ST(1); calculate ST(1)+ST; push the sum onto the stack
FDIV	Divide Real Pop ST and ST(1); calculate ST(1)/ST; push quotient onto the stack
FMUL	Multiply Real Pop ST and ST(1); calculate ST(1)*ST; push product onto the stack
FSUB	Subtract Real Pop ST and ST(1); calculate ST(1)-ST; push difference onto the stack

Control	Explanation
FINIT	Initialize the NDP
FWAIT	Wait for the NDP to complete the current operation
FCOMPP	Compare Real (and pop both operands off the stack)
FSTSW	Store NDP Status Flags
Sample code to compare two Floating- point variables	<pre> fld [A] ; push A onto the stack fld [B] ; push B onto the stack fcompp ; calc A - B (and pop A and B off the stack) fstsw [status] ; store the NDP flags in status fwait ; wait for the NDP to complete the store mov ax,[status] ; load ax with the NDP flags sahf ; store ah into the x86 low flag byte jb ... ; jump if A < B je ... ; jump if A = B ja ... ; jump if A > B </pre>

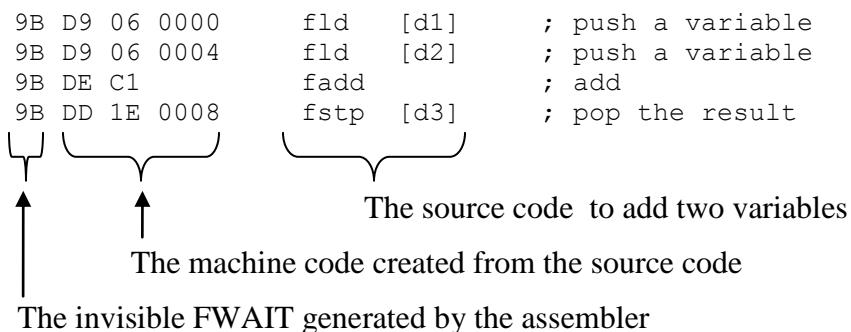
Synchronizing the x86 and the NDP

It is likely that the x86 will be able to execute instructions significantly faster than the NDP. This means that the x86 will have to wait for the NDP to complete each Floating-point operation. This wait is required in two situations.

1. The x86 cannot start another Floating-point operation until the current one is complete.
2. The x86 cannot use/save the results of the current floating point operation until it is complete.

The x86 uses the *fwait* instruction to wait until the NDP has completed its current operation. The manner in which this fwait is coded is different for the two situations described above.

1. To make the programmer's task easier, the assembler automatically generates an fwait instruction before every Floating-point operation. You will not see this fwait in your source code, but you can see it in the listing file and the debugger. This help from the assembler resolves issue #1 above. A new Floating-point operation will not be attempted until the current one is complete.



2. To resolve issue #2, the programmer (*that means you*) must insert an fwait instruction before any x86 host processor instruction that uses the result from the current Floating-point instruction. The sample code shows examples of how to code this process.

Safe Testing

You have two values and you wish to determine whether they are equal. For example $A = 1000.1$ and $B = 1000.0$ and we calculate $C = A - B$. We want to know if $C == 0.1$

However, when doing those calculations, $1000.1 - 1000.0$ yields 0.099975586 while the value created for $1/10$ is 0.100000001 and those will **not** compare equal. Instead we establish a value, epsilon, with a small value such as 0.0001 and write the code that indicates C equals 0.1 if the difference between C and 0.1 is less than epsilon.

```
c = a - b;                                // c = a - b;
e = (c - 0.1); if (e < 0.0) e = -e;          // e = absolute value of (c - 0.1)
epsilon = 0.0001;                            // acceptable difference = 0.0001
if (e < epsilon) printf("c == 0.1 \n");      // if c is close enough to 0.1
else                printf("c != 0.1 \n");      // then decide c == 0.1
```

References

- The Perils of Floating Point, Bruce M. Bush, Lahey Company
- IEEE-754-2008, Wikipedia
- iAPX 86,88 User's Manual, Intel Corporation
- Essentials of 80x86 Assembly language, Richard C. Detmer, Jones and Bartlett

Interrupts and I/O

One of the most important capabilities of modern computers is the ability to work on multiple tasks at the same time and to be able to overlap multiple operations such as executing instructions while a record is being read from a disk. The driving hardware force that supports the ability to do multiple tasks at the same time is the *interrupt*.

An interrupt is a special input to the CPU that allows the CPU to react to an asynchronous event.

Asynchronous events are events that the CPU knows can occur, but the CPU does not know exactly when they will happen.

First, a non technical example. You are working at your desk at home on your income taxes. The phone rings (an interrupt). You stop work on your taxes and answer the phone (you accept the interrupt). It is your friend who wants to know the name and phone number of your auto mechanic so she can call to arrange a service for her car. You give her the name and number (you process the interrupt request immediately). You then hang up and go back to working on your taxes. Note that additional time it will take you to complete your taxes is minuscule; yet the amount of time for your friend to complete her task may be significantly reduced.

There are two generic forms of asynchronous events:

1. a request for service
2. a notification of completion

An example of a requests for service:

- A person sitting at a keyboard presses a key on their keyboard. The CPU may be busy performing other tasks such as downloading an image file from the WEB or formatting a document in a word processing system or playing music. The interrupt tells the CPU to stop for a brief moment and read the key pressed. The CPU can then decide to either resume its previous work or perform some other work based upon the key pressed. In a well designed system, the CPU is fast enough and the interrupt time is short enough so that the interrupted tasks are not adversely affected. For example the modem does not drop the connection, and the music continues to sound good.

An example of a notification of completion:

- This is a signal to the CPU that some event which the CPU started in the past, has now completed. For example, the CPU might have passed a page of text to a printer to be printed. When the printer has completed printing the page, it would signal the CPU that it was done.

In both these cases, the interrupt has freed the processor from continually checking to see whether a key was pressed or the page has been printed. Interrupts are a very efficient way to handle these asynchronous events.

Some non critical interrupts can be *masked off*. This means that the CPU does not want to accept the interrupt at this moment. In our example, you could mask off the telephone interrupt by just not answering the phone because you are half way through adding up a column of numbers. In most cases, the source of the interrupt will just keep trying to generate the interrupt (the phone keeps ringing). The interrupt remains *pending*. Eventually you finish adding up the numbers and *enable* interrupts again. You now accept the pending interrupt (you answer the phone).

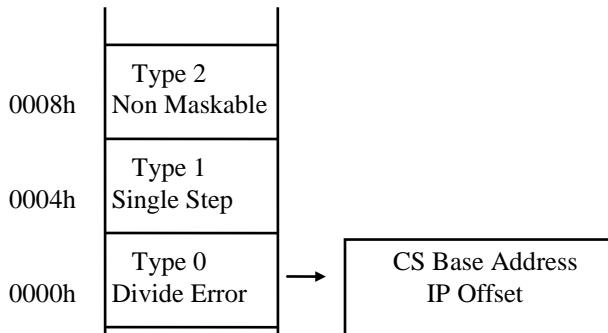
Some critical interrupts can not be masked off. These are called Non Maskable Interrupts (NMI). Instead of the phone ringing, your fire alarm goes off. Regardless of what you are doing, you will stop and handle the fire alarm.

In our computer example, the keyboard interrupt can be masked off. However, a divide by zero interrupt can not be masked off.

Intel Interrupt Processing

Intel supports multiple interrupt types. For every active interrupt type, there must be special code written to handle the interrupt. This code is called an *interrupt handler*. In most cases, the interrupt handlers come with your Operating System. However, you may always write your own version of an interrupt handler.

The link between the actual interrupt occurring and the interrupt handler code being executed is the Interrupt Vector Table. This table is located at absolute address 0 in memory. The table has multiple entries. Each entry is two words and contains the Code Segment Register value and the Instruction Pointer offset for each interrupt handler program.



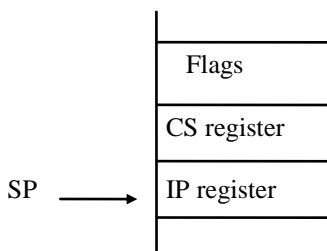
In general, when an interrupt occurs:

1. The system must save, on the stack, the relevant information concerning the currently executing program.
2. The system must activate the interrupt handler.
3. The interrupt handler executes.
4. The system must restore, from the stack, all the information for the interrupted program and must return control to the interrupted program.

Specifically, when an interrupt occurs, the following actions are taken:

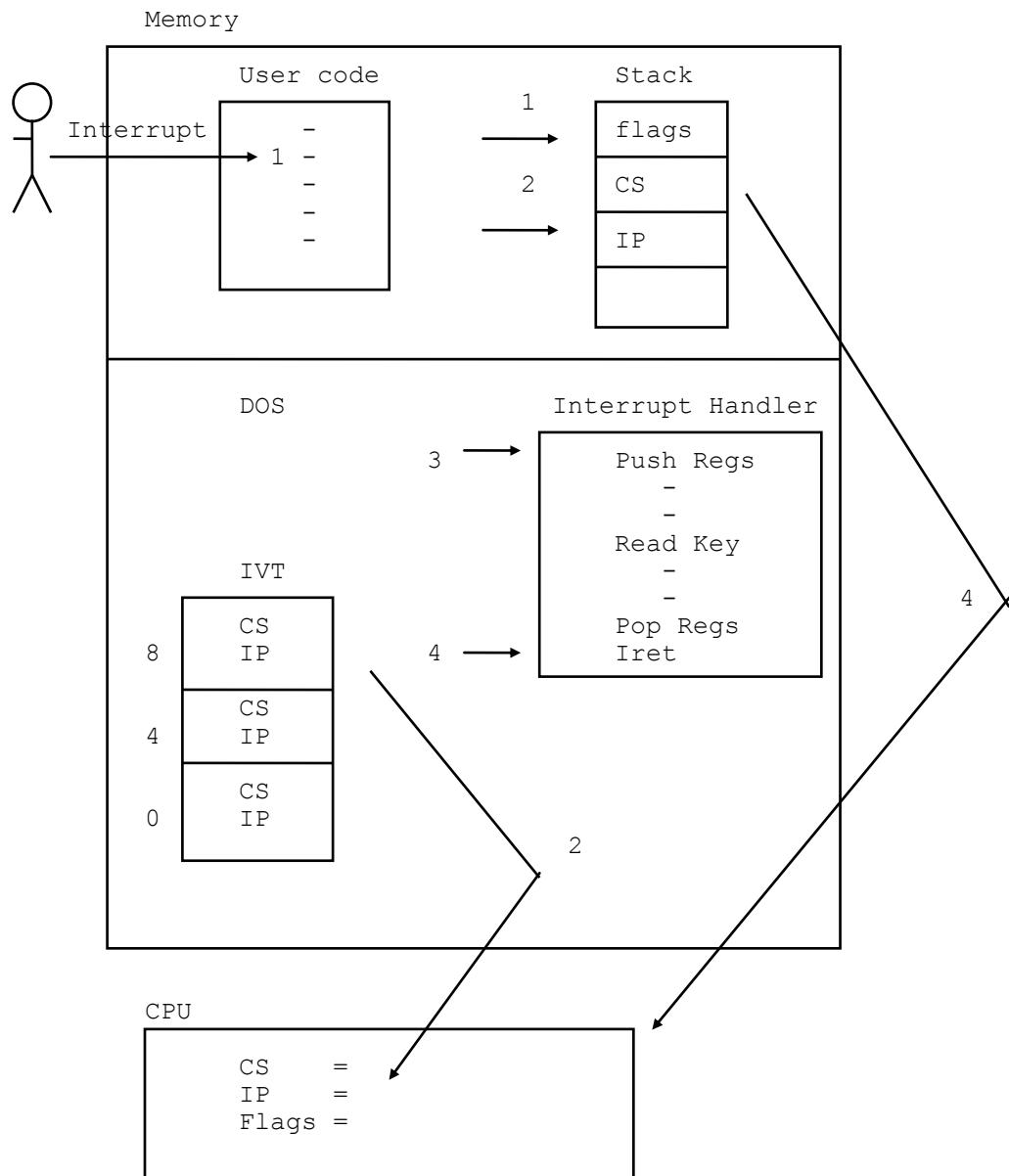
1. The CPU pushes onto the stack, the current user's Status Flag register.
2. The CPU activates the corresponding interrupt handler by executing an inter-segment call to the segment and offset specified in the corresponding interrupt vector table entry. This pushes the current CS register and IP register onto the stack. Then the CS register and IP register are then replaced by the corresponding entry in the interrupt vector table.
3. The interrupt handler code is executed. It must save and restore any registers it modifies.
4. When the interrupt handler is done, it executes an interrupt return instruction (IRET). This restores from the stack, the Status Flag register, the CS register, the IP register, and the interrupted program resumes execution.

Upon entry to the interrupt handler, the stack looks like:



An aside note. You can now understand how int 21h works for file I/O. When you issue an int 21h, the interrupt handler in entry 21h of the Interrupt Vector Table is executed.

How do the pieces work together



1. Interrupt occurs. The current user's FLAGS are pushed onto the stack.
2. The current user's CS and IP are pushed onto the stack and the active CS and IP are loaded from the IVT.
3. The interrupt handler is executed.
4. The interrupt handles terminates and the user's program is reactivated.

Notes on DOS interrupt handlers

DOS is a single user system. To install an interrupt handler, you need a setup program that loads the interrupt handler code which then remains operational *after* the setup program ends. This is called *Terminate And Stay Resident (TSR)* code. Once a TSR program is loaded, it remains active in memory until the DOS system is reloaded. A virus checker program is an example of TSR code.

You can *not* easily debug an active interrupt handler in real time. You may need special hardware and software for developing interrupt handlers.

In DOS there are two types of executable program formats.

- The one we have studied so far is the .EXE format. This is the most general type of program. It allows you to access four unique segments (CS, DS, ES, SS).
- The other is the .COM format. This is a small high performance version used for system's functions. In a .COM program, all four segments share the same memory (which is limited to 64 KBytes). Interrupt handlers are .COM format.

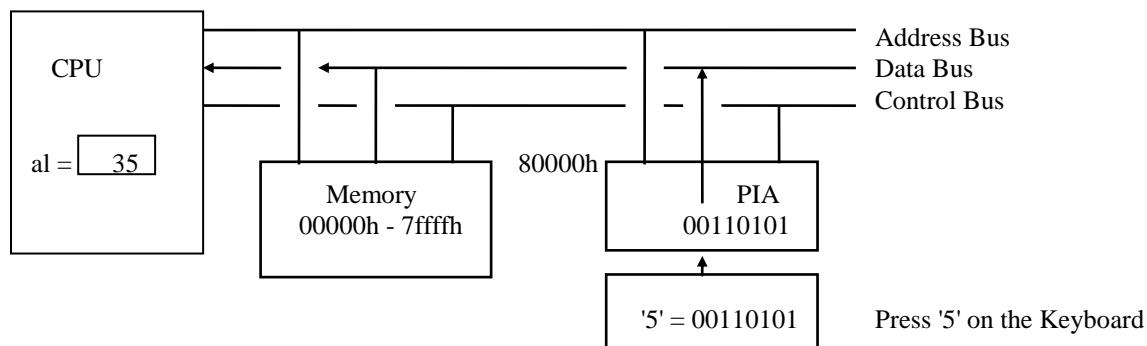
Three ways of doing Input / Output

- **Programmed I/O (PIO)**

Using PIO, external devices are connected to hardware *ports*. Each port has a unique number. There are special instructions that can read and write these hardware ports. For example: IN AL, 08h will read the byte value in hardware port 08h. This was the original and primary way of doing I/O on the 8086. It works acceptably well, but is somewhat restrictive in that there are only two instructions that can reference a port: IN and OUT.

- **Memory Mapped I/O (MMIO)**

In a memory mapped I/O system, some main memory addresses are assigned to Input/Output devices. The CPU sees these peripheral devices as memory locations in its 1 MegaByte memory space. To perform writes or reads, the program just moves data to or from these assigned memory addresses. This is much more flexible than PIO. Any instruction that references memory (mov, and, add) can be used to access an I/O device in memory space. In the following example, we have a system with 1/2 MegaByte of memory. That memory is addressed from 00000h - 7ffffh. Also attached to the system is a keyboard. The keyboard is connected to the system through a Peripheral Interface Adapter (PIA). A PIA is a circuit that is used for interfacing the system buses to the outside world. In our simplified environment the PIA looks like a one byte memory location to the CPU, and it is set to respond to memory address 80000h. The PIA contains a one byte register that is loaded with the ASCII value of any key that is pressed on the keyboard (in the diagram below the number '5' key has just been pressed). If the programmer tries to read the data at 80000h using mov al,[80000h], then the PIA will place the contents of its one byte register onto the data bus, and al will become 35h.



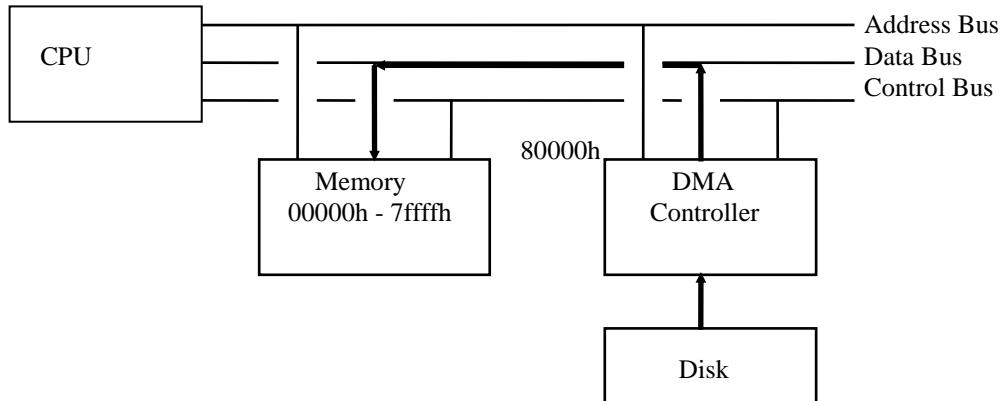
How does the CPU know when a key is pressed. The keyboard passes that information to the PIA. The PIA will generate an interrupt (request for service) to the CPU. This interrupt will be passed to the CPU on the Control Bus. The PIA will also identify itself to the CPU; that is tell the CPU which device is requesting service. This may be done by passing a code to the CPU on the Control Bus. The operating system will then have to know that to service an interrupt from the PIA, it should read the data at address 80000h.

- **Direct Memory Access (DMA)**

When using PIO or MMIO, the attention of the CPU was required for *every* keystroke transferred. There are many devices from which we want to transfer a record or collection of bytes. This is true of disks, page printers, tapes.

DMA allows us to transfer a whole record to/from a device, without having each byte of the record pass through a CPU register. The data flows directly between the I/O device and memory. This is very efficient and allows the CPU to execute a program simultaneously with a data record being read from a disk.

DMA requires that the I/O device be connected through an intelligent I/O interface, called a DMA controller. The DMA Controller, must be capable of generating memory addresses as well as providing the data transfer. A DMA Controller is in fact a small computer. DMA Controllers must be able to use the system buses just as the CPU itself does, and do it in a way that does not harmfully interfere with the CPU using the system's buses.



For example to read disk record into memory, the Operating System provides to the DMA Controller the following information:

1. The main memory address to receive the data.
2. The location of the data on the disk.
3. The direction of the data transfer. A 'read' is from the disk to main memory.
4. The number of bytes to be transferred.

Once given this information, the DMA Controller will perform the I/O operation. It will alternate with the CPU, taking control of the Address Bus, Data Bus and Control Bus.

When it has completed the data movement, it will generate an interrupt to the CPU signaling the completion of the event.

Design Trade-offs

The design of computing systems is continually changing. This makes it an exciting field of academic study and provides excellent career opportunities.

During the decades of the 1960s, 1970s and 1980s a dominant form of computer architecture was the Complex Instruction Set Computer (CISC). The basic concept of CISC machines was to make the hardware more powerful and more complex in an attempt to make software development easier for the compiler writer and the assembler programmer. CISC machines had these characteristics:

- Many forms of complicated addressing modes with many options for indirect addressing and indexing.
For example *mov ax,[bx+si-100]*
- A single machine could have instructions with a different number of operands: *ret inc ax add ax,bx*
- Multi-function instructions such as looping instructions and string instructions.
- Variable length machine instructions.
- Instructions that require many machine cycles.

Research in the 1980s showed that most compilers and assembler programmers rarely used the complex features of CISC machines. It was too hard to effectively use these complex functions.

This research led to new form of computer architecture called Reduced Instruction Set Computers (RISC). The basic concept of RISC machines was to only build high-speed primitive instructions into the hardware. All complicated functions would be built up, by the compiler or assembler programmer, from these high-speed primitive instructions. RISC machines had these characteristics:

- Simple addressing modes. All forms of indexing would be accomplished by having the software build the index value (calculate the effective address).
- Single simple machine instruction formats.
- All arithmetic operations are done using register operands only. The only operations that access memory are loads and stores.
- One instruction is executed each machine cycle.

We start our discussion of computer performance enhancements by looking at one of the choices needed to be made by early computer architects.

How many operands should be specified in an instruction?

Since we are trying to get insight into the possible trade-offs, we will look at machines that specify a different number of memory operands.

Specifically, we will look at how one might calculate $C = A + B$ on different machines that have a different number of explicit memory operands that can be specified by a programmer.

Machine With Zero Operands

Machines that do not have explicit operands are called Stack Machines. Data values are pushed onto the stack and popped off the stack as needed. Arithmetic instructions, such as addition, pop the top two values off the stack and add them together and then push the result back onto the stack. Note that push and pop do really allow an explicit operand, however, arithmetic operations such as addition or subtraction do not. For simplicity, we will not consider Stack Machines for this exercise.

Machine With One Operand

This class of machine typically has only one general-purpose register called the *accumulator*. That accumulator is an implied operand of most instructions. With the one explicit operand, you can reference the variables A, B and C and our calculation can be done as follows:

LOAD_ACC	A	;PLACE VARIABLE A INTO THE ACCUMULATOR
ADD_ACC	B	;ADD B TO THE ACCUMULATOR
STORE_ACC	C	;STORE THE ACCUMULATOR INTO C

Machine With Two Operands:

This class of machine usually has multiple general-purpose registers and more sophisticated instructions. Since two operands can be specified explicitly, we can perform the calculation as follows:

MOVE	A,C	;MOVE VARIABLE A TO VARIABLE C
ADD	B,C	;ADD VARIABLE B TO VARIABLE C

Note that with two explicit operands, one of those operands will represent both a source and destination. For example, the ADD B,C instruction can be viewed as $B=B+C$. This means that B is an input source and output destination.

Machine With Three Operands:

With three operands, it is possible to perform the calculation with a single instruction. Also note that the two sources can be independent of the destination.

ADD	A,B,C	;ADD A TO B AND STORE THE RESULT IN C
-----	-------	---------------------------------------

Can We Draw Any General Conclusions

What conclusions can be drawn from this example? As you move from fewer to more operands:

- The instructions tend to get more complex and more powerful.
- It appears that you need fewer instructions to accomplish the same task.

However, that does not necessarily make one design better than another. *Which is better, will depend on what needs to be accomplished.*

Example Of Three Different Architectures

These are machines with 1, 2 and 3 memory operands. The machines have the following characteristics.

- Timings are given in microseconds (us) where 1 us = 1/1,000,000 second.
- Each machine can reference 64 K bytes of memory via a 16 bit address bus.
- Each machine has a one byte data bus. All data is one byte. All operations are performed on one byte.
- The memory of each system can deliver 1 million bytes per second to the CPU.
- The opcode for each machine is one byte.
- All operands are 16 absolute addresses of the desired byte in memory.
- Instruction execution consists of four parts:

1. <i>fetch</i> the instruction	1 us per byte
2. <i>decode</i> the instruction	fixed at 1 us
3. <i>access the operands</i>	1 us per operand
4. <i>execute</i> the instruction	fixed at 2 us

For each machine we can determine the average time to execute an instruction, the execution rate of the machine in instructions per second, and the time needed to calculate $C = A + B$.

Characteristic	M1	M2	M3
Instruction format	opcode & 1 operand	opcode & 2 operands	opcode & 3 operands
Instruction size	3 bytes	5 bytes	7 bytes
Time to <i>fetch</i> an instruction	3 us	5 us	7 us
Time to <i>decode</i> an instruction	1 us	1 us	1 us
Time to <i>access operands</i>	1 us	2 us	3 us
Time to <i>execute</i> the instruction	2 us	2 us	2 us
Total time to execute one instruction	7 us	10 us	13 us
Instructions executed per second	142,857	100,000	76,923
Time to calculate C=A+B	21 us	20 us	13 us

From this chart we can see it is possible to draw conflicting conclusions.

- M1 is the fastest machine in instruction executions per second. However, it does not perform the calculation in the least amount of time.
- M3 is the slowest machine in instruction executions per second. However, because its instructions are the most powerful, it performs the desired calculation in the shortest amount of time.

The real conclusion to be made is that in order to compare the performance of different machines you must consider more than just instruction execution rate. You must consider:

1. The instruction execution rate of each machine.
2. The capability of the instructions of each machine.
3. The specific application to be programmed on the machine.

Improving Performance

This section discusses changes that can be made to the basic system's architecture in order to improve the performance of the system. Most of these changes were developed and implemented in large mainframe computers in the 1970s. Many have been implemented in modern microprocessors when the cost to implement the concept became affordable. In most cases, changes to improve performance are implemented so that they are transparent to the assembler language programmer. This allows compatibility among different models of machine within the same computer family.

The Problem With Building Fast Computers

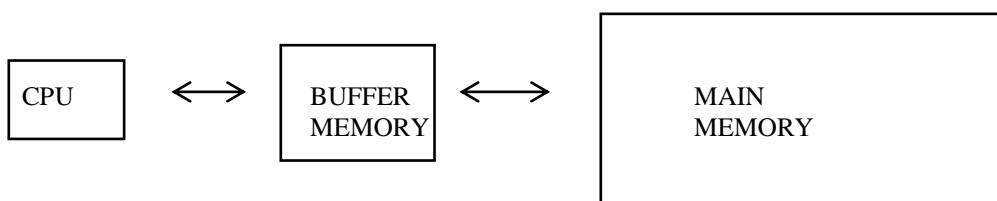
Why can't manufacturers just build faster computers with faster logic circuits? There are three basic reasons.

1. At any given time, there is a practical limit on the maximum speed of circuits that can be built by the semiconductor industry.
2. The cost of building circuits that operate near that limit is much higher than building circuits that operate at some slower speed. Today, it is possible to build a microprocessor that executes 250,000 instructions per second and sells for under \$1. Microprocessors that execute at 4 to 10 billion instructions per second are at the upper limit of performance and cost hundreds of dollars.
3. A CPU may contain 40 million transistors. If you connect 1 Gigabytes of memory to that CPU, the 1 Gigabyte of memory may contain $6 \times 8 \times 1$ billion transistors (6 per bit). In such a system, the memory contains 1000 times the circuitry of the CPU. Even if you were willing to pay the cost of fast logic for the CPU, it might be too expensive to build the very large memory for that system.

However, it is still possible to build fast and expensive CPUs and connect them to slow and inexpensive large memories, but still have a fast system.

Buffer or Cache Memory

In the 1970s IBM built a System/370 model 168 with a fast 80 nanosecond cycle time CPU and a two level storage system. A small, 16 Kilobyte, very high speed buffer / cache storage was backed by a slower large 16 Megabyte main memory. The buffer memory operated at the same speed as the 80 nanosecond cycle time CPU while the backing storage operated about five times slower.



The hardware automatically moved instructions and data that were being used from main memory to the buffer. This was transparent to the programmer who did not need to adhere to any special program structure to take advantage of the buffer. When the CPU referenced instructions or data, the hardware determined whether that referenced memory had been copied to the buffer. If so, the instruction or data was given immediately to the CPU. If not, the referenced memory and the 32 consecutive bytes that surround it were first copied from main memory into the buffer (see why below).

For a buffer/cache technique to be effective, programs must exhibit two characteristics.

1. First is *locality of reference*. This simply means programs tend to spend a lot of time executing small tight loops and programs tend to use the data variables repeatedly. Therefore, once those instructions and data are brought into the buffer, they will be used many times.
2. Second is that programs tend to use instructions and data in a sequential manner. This means that if a program uses an instruction or variable, there is a very good chance it will use the next sequential instruction or variable. That is why the hardware brings into the buffer the memory that surrounds the referenced instruction or data.

Does this buffering technique work in real machines? Yes. Let me quote from *A Guide to the IBM S/370 168*. "Sample job step executions have shown that in the Model 168 the data accessed by the CPU is in the buffer 95% of the time on the average"

Implementing buffering is not free. Some problems include:

- Programs that do not exhibit these characteristics will execute slowly.
- The control circuitry to manage the buffer can be complex and expensive.
- You need an effective page replacement algorithm to replace data in the cache when it becomes full. Lazlo Belady showed in 1970 that systems with larger caches can actually run slower than systems with smaller caches if you select a bad page replacement algorithm. This is called Belady's Anomaly.

The Instruction Unit

Instructions can be classified as those which execute very quickly such add and subtract, and those that execute more slowly such as multiply and divide.

- The bottleneck in executing the fast instructions tends to be the fetching and decoding of those instructions.
- The bottleneck in executing the slow instructions is that they require a lot of time in the execution unit.

Why not try to get better performance by overlapping the functions performed by the I-UNIT and the E-UNIT. Let the I-UNIT prefetch a certain number of instructions, decode them, and do effective address calculation. This means that when the E-UNIT is busy on a slow instruction, the I-UNIT is working ahead. When the E-UNIT starts moving through fast instructions, the I-UNIT has already prepared a number of instructions. In the case where the time to prepare an instruction is about the same as the time to execute it, there will be a performance improvement of about 2:1. This technique is called pipelining. It is very similar to the concept of how cars are built on an assembly line.

Time	No Overlap (Not Pipelined)	With Overlap (2 Stage Pipeline)	
1	Fetch & Decode Instruction 1	Fetch & Decode 1	
2	Execute Instruction 1	Fetch & Decode 2	Execute 1
3	Fetch & Decode Instruction 2	Fetch & Decode 3	Execute 2
4	Execute Instruction 2	Fetch & Decode 4	Execute 3

What happens when a conditional branch is encountered during the prefetching operation? It is not known whether the branch will actually be taken, and therefore it is not known which path of the branch should be prefetched next.

The system could:

- Stop prefetching and wait until it knows whether the branch will be taken.
- Estimate the probability of branching and then prefetch in the direction of highest probability.
- Prefetch both paths of the branch. The S/370 168 contained two instruction buffers and prefetched both paths of a branch. The sequential path goes into a primary instruction buffer and the target path goes into a secondary instruction buffer. This ensured the availability of prefetched instructions whether the branch is taken or not. The 168 could prefetch a total of 12 instructions and decode the first 4 of those. A problem associated with prefetching is that the control circuitry for prefetching is complex and expensive.
- Prefetch the sequential path and implement *delayed branching* by always executing the instruction after the branch. The instruction following the conditional branch is in the *delay slot*. There are processors with 3 stage pipelines that have one delay slot and there are processors with 4 stage pipelines that have two delay slots. Due to the difficulty of finding multiple instructions that are executed regardless of whether the branch is taken, delayed branching is less practical for machines with many stages in the pipeline and thus many delay slots..

The Execution Unit

There are different approaches to speeding up the E-UNIT.

- Add special hardware, such as a multiply unit or a floating point unit. This can speed up these complex operations by an order of magnitude.
- Provide a wider data path. For example, replace an 8 bit parallel adder with a 16 bit parallel adder. This can speed up word operations.
- Provide multiple adders and multipliers and run them simultaneously. For example, in the formula: $(A*B) + (C*D)$ each of the multiplication operations could be done at the same time if the system had two multipliers.

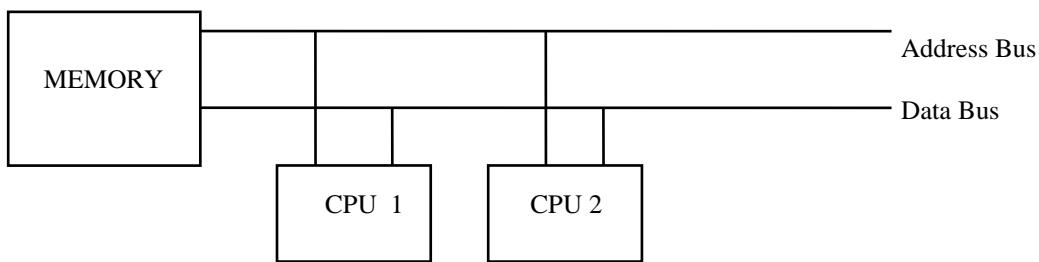
Many special purpose computers have been built. The IBM S/370 195 was a high performance scientific computer. It had multiple adders and multipliers, and could perform three simultaneous operations.

Problems with these techniques are:

- Control circuitry for managing multiple adders and multipliers can be complex.
- It is difficult to write programs that can take advantage of the potential simultaneous operations.

Multiprocessing (The term used by Intel is Multiple Cores)

Multiprocessing is a configuration where two or more CPUs share the same main storage and operate under the control of a single operating system.



Multiprocessor systems are designed to provide:

- Improved availability of the system. If one CPU fails the others can continue working.
- Improved resource utilization. All the processors share resources such as memory and I/O devices.
- Improved performance. This one is tricky. Can a single program utilize two CPUs? Performance is improved most when normally sequential operations can now be done in parallel. For example each CPU running a different weather forecasting algorithm and then comparing results, or having each CPU executing commands from different terminal users simultaneously. Studies have shown that "the internal performance of a Model 168 tightly coupled multiprocessing configuration is in the area of 1.5 to 1.9 times that of a uniprocessor Model 168. However, because of the load leveling that occurs in a tightly coupled configuration, the throughput achieved for a given job stream can be greater than that achieved when the same job stream is split and processed by two uncoupled systems".

Problems with multiprocessing include:

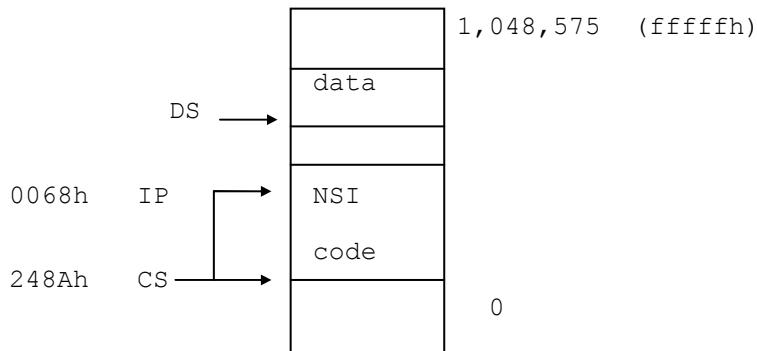
- Software control and hardware control are complex.
- The potential for simultaneously updating data can be hazardous to the health of a database.

Post 8086 Architecture

8086

The 8086, that we have studied, supports 1 Megabyte of real physical memory. It addresses this 1 MB by using a 20 bit address bus. Since the 8086 registers are only 16 bits, it uses the concept of segmentation to address the 1 MB. The 8086 has 4 segment registers (CS, DS, ES, SS). To generate a 20 bit physical address the hardware multiplies the value in the 16 bit segment register by 16 and then adds an offset such as the Instruction Pointer (IP).

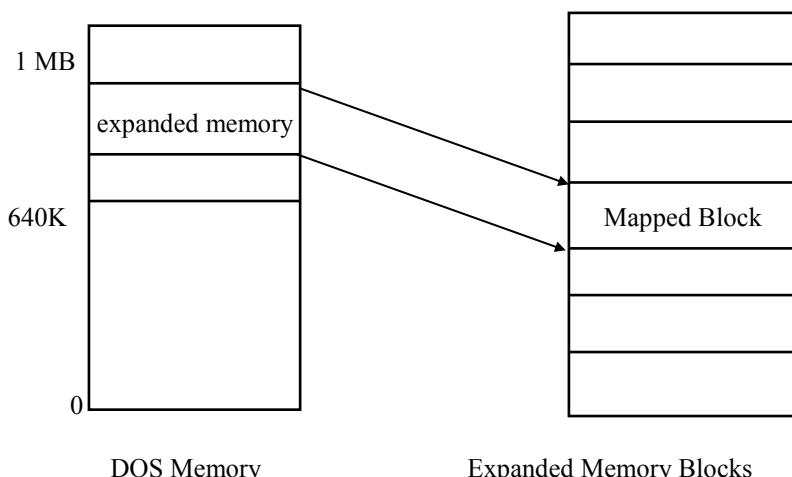
For example, given the Code Segment register contains 248Ah and the Instruction Pointer (IP) which points to the next sequential instruction (NSI) to be executed contains 0068h, then the address of the next instruction executed is $248A0h + 0068h = 24908h$. Below is a picture of the 1 MB address space.



This concept is referred to as *8086 Real Mode Addressing*.

The first attempt at supporting more than 1 MB of memory was developed by a three company (Lotus, Intel, Microsoft) task force. It was called *expanded memory system (ems)*. EMS took advantage of one of the unused 64 KB blocks of memory that resided between the top of DOS user memory (640K) and the top of memory (1 MB).

Through special software packages and hardware, an application could *map* this 64K address space into one of any of an unlimited number of expanded memory blocks. When software referenced the expanded memory location in the DOS address space, the hardware really referenced the mapped block of extra memory. The technique was slow, and did require the use of special memory management software; but it was used since it did open up the range of available memory.

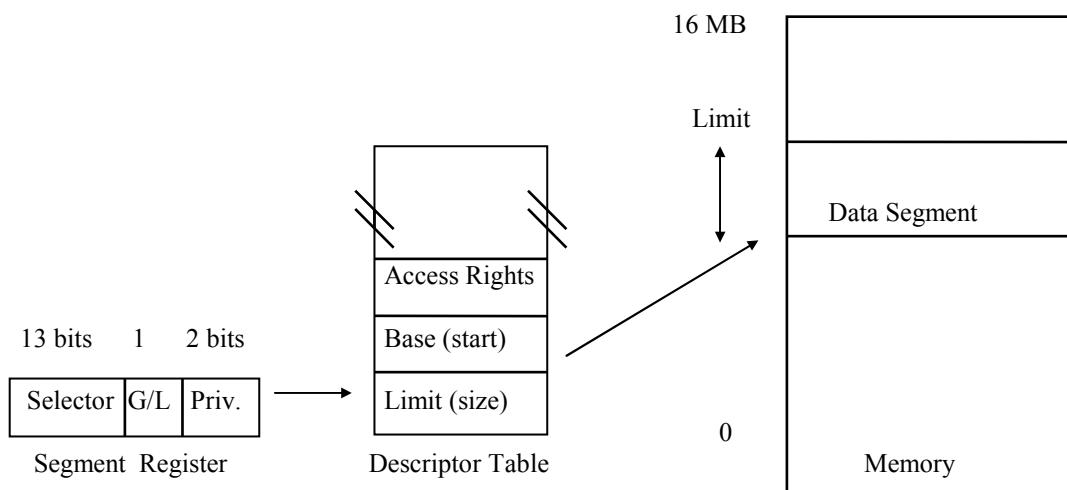


80286

The 80286, with its 24 bit address bus, allowed access to 16 MB of memory. However, Intel recognized the absolute necessity of providing machine code compatibility with enormous amount of software that existed for the 8086. So the 80286 had two modes of operation.

Real Mode. In real mode, the 80286 was 100% machine compatible with the 8086. Physical addresses were calculated as (Segment * 16 + offset) and were limited to 1 MB.

Protected Mode. In protected mode, the 80286 can access 16 MB of memory through its 24 bit address bus. This memory, above 1 MB, is called *extended memory* (xms). XMS uses a different way of calculating a physical address. The offset is still used to point to a location within a segment. However, the segment register now contains a *selector* that points to a *descriptor* which describes the segment's location, length and access attributes.



For the 80286 a segment register is still 16 bits. It uses 13 bits to point to an entry in the Global or Local Descriptor Table. Each of the 8192 entries in a Descriptor Table defines a segment. *Limit* is the 16 bit size of the segment (1-64 KB). *Base* is the 24 bit address of the start of the segment. The *Access Rights* will indicate whether this segment is read/write data, read only data, execute only code.

Global descriptors contain segments that apply to all programs; an example would be segments with operating system service routines. Local descriptors contain segments that are unique to a specific application program that is executing; examples would be its code and data segments. There is one bit in the segment register that indicates whether to use the Global or Local Descriptor Table.

Segments can also be assigned a privilege which is used to protect the operating system from being modified by a user. There are 4 levels of privilege: 0 for the most important part of the operating system such as the interrupt handlers, 1 for operating system services such as allocating memory, 2 for operating system extensions such as print routines, 3 for user applications. There are two bits in the segment register that contain the requested privilege level for a segment. (There are two special machine registers that point to actual physical addresses of the Global Descriptor Table and Local Descriptor Table. These are the GDTR and LDTR.)

The 80286 introduced a few new instructions, of note are PUSHA/POPA which will save/restore all the general purpose and pointer registers with a single instruction.

With these features, the 80286 running in protected mode could support a multi-user operating system such as Windows and support multiple applications running at the same time.

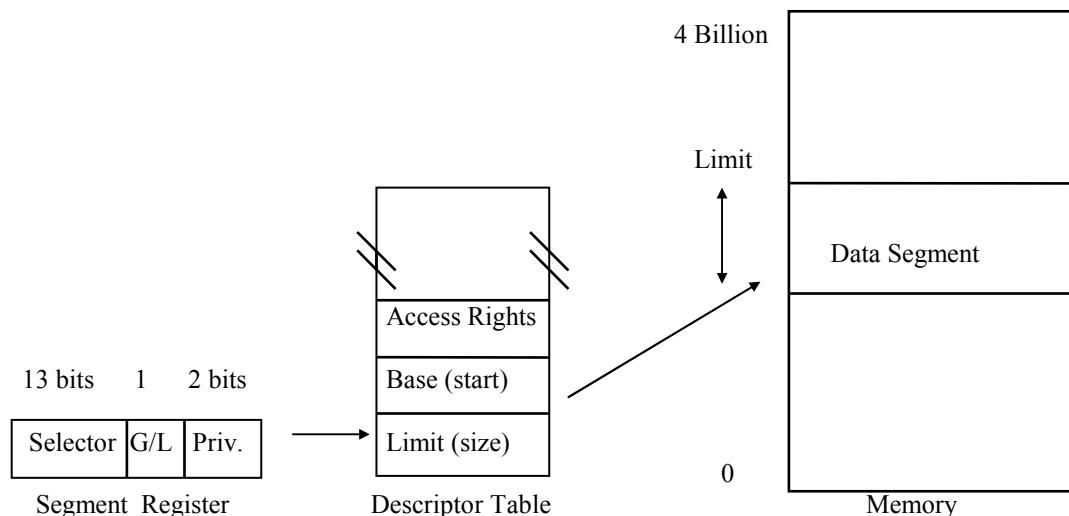
80386

The 80386 is a full 32 bit processor. It has an extended version of registers found on earlier Intel processors. For example, the Extended Accumulator (EAX) is 32 bits. The lower word can still be referenced as AX or as AH and AL. Also notice that there are two more Segment (Selector) registers, FS and GS.

	General Purpose and Pointer Registers				Segment (Selector) Registers	
	16	8	8		16	
EAX		AH	AL	AX		
EBX		BH	BL	BX	CS	
ECX		CH	CL	CX	DS	
EDX		DH	DL	DX	ES	
ESP		SP			SS	
EBP		BP			FS	
EDI		DI			GS	
ESI		SI				

The 80386 supports two modes of operation like the 80286.

- In Real Mode the chip looks like a very fast 8086.
- In Protect Mode, the chip can access 4 billion bytes of main storage through the use of selectors that point to descriptors that define segments. An enhancement for the 80386 is that now a segment can be 1 byte to 4 billion bytes.



The 80386 introduced more significant changes.

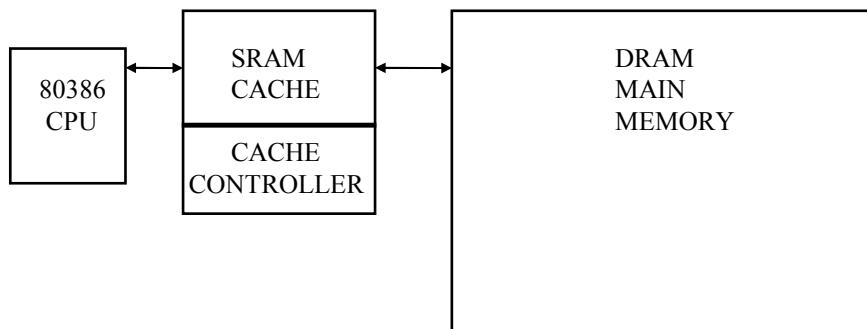
Pipelined Architecture. The 80386 has a multi stage instruction pipeline that allows the CPU to overlap the FETCH, DECODE, EXECUTION of three instructions.

Virtual 8086 (V86) Mode. In V86 mode, multiple simultaneous 8086 Real Mode environments can be created. This means that a system such as Windows can have multiple DOS boxes active, each one running a different Real Mode application. In V86 mode a ‘virtual 8086 machine’ is formed. The hardware provides a virtual set of registers and a 1 MB virtual address space. With additional software support, the system also provides the required external interfaces, such as I/O.

In V86 mode, software runs just as it would on a real 8086 with *some* exceptions. Three interesting ones are:

- Instruction clock speed will be faster and execution time will be quicker.
- Undefined 8086 Opcodes will behave differently on a real 8086 versus V86 mode.
- Segment wrapping (crossing the boundary of 65535) will not work in V86 mode .

Cache Memory Subsystem. Traditional systems are implemented using dynamic RAM (DRAM) memory which can provide a large amount of memory at an affordable price. DRAM is, however, relatively slow compared to the speed that a 80386 processor requires. Faster static RAM (SRAM) memory can meet the performance needs of a system but it is much more expensive. A system with cache memory contains a small amount of SRAM and a large amount of DRAM. The system is configured so that programs and data are loaded into the cache when needed and they execute out of the cache. If designed correctly, such a hybrid system will provide the performance of an SRAM system at the cost of a DRAM system.



The cache controller performs block fetches from main memory into the cache. The 80386 has a 32 Kbytes cache, and uses 16 byte blocks. This means that the cache can hold 8 K blocks. The cache controller attempts to bring in a referenced data location and the data locations that follow it (this is called cache lookahead).

In a cached system, two copies of the same data exist at the same time. One is in main memory and the other is in the cache. It is important to assure that both copies of the data are kept the same. The system must assure that if a program modifies data in the cache, that main memory is also modified. In a 80386 system, whenever a program modifies data, the data is changed in the cache and the cache controller moves the affected block back out to main memory (this is called cache write through).

New Instructions. The 80386 provides new instructions that support the extended registers as well as some new addressing modes.

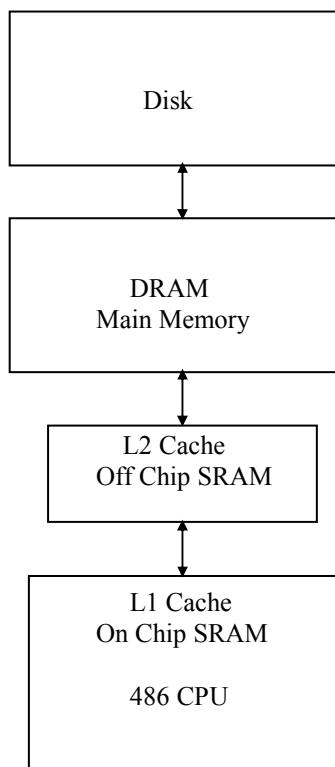
80486

The 80486 is faster and more highly integrated device than the 80386. However, from an architectural view it is almost identical to an 80386 system. The 80486 contains

- a complete 32 bit CPU that is upward compatible with the 80386
- a floating point numeric coprocessor (that required a separate 80387 chip on 386 systems)
- an on chip 8 Kbyte high speed cache

Almost half of the instructions for the 80486 execute in one clock cycle instead of two clock cycles for the 80386. This accounts for a major performance improvement between the 386 and 486.

Now that the 80486 has an on chip cache, we need to have a new picture of memory hierarchy.



Pentium

The x86 architecture is the epitome of the Complex Instruction Set Computers (CISC). CISC computers were built to take the burden off the programmer and put it on the hardware. This type of architecture was predominant in the time frame 1960 - 1990.

Key examples of CISC concepts are:

- Variable length machine instructions
- Instructions that require many machine cycles
- Many forms of complicated address modes
- String instructions

Starting in the 1990's there was a move toward Reduced Instruction Set Computers (RISC). RISC computers were built with simplicity and speed as their objective.

Key examples of RISC concepts are:

- Fixed instruction lengths
- All instructions execute in one machine cycle
- Simple address modes
- The only operations involving memory are load and store
- All mathematical and logical operations use registers

The theory behind RISC computers is that 95% of the time programs do simple things, so do those simple things very fast. If more complicated operations are needed then let the programmer or compiler generate the code to accomplish them.

The problem for Intel was how does CISC architecture take advantage of RISC concepts.

The following analysis comes from the text "Computer Organization & Design" by Patterson and Hennessy and published by Morgan Kaufmann. Even the latest Pentium carries the burden of the original 8086 complex instruction set where complicated addressing modes and string instructions mean that many instructions require many cycles to complete. Intel architecture has traditionally been implemented using a microprogrammed controller. The new challenge is to ensure that any simple RISC type instructions execute quickly and that the burden of the complexities of the instruction set penalize only the less frequently used CISC like instructions. To accomplish this goal, the Pentium uses a combination of hardwired control to handle simple instructions and microcode control to handle the more complex instructions.

In other words, the Pentium architecture provides an outer CISC layer that is seen by the user, and an inner RISC core that does the work.

The Pentium brings many significant performance enhancements to the Intel processor family. The exact numbers and sizes depend on the Pentium version. Take the following as conceptual information. Some good references are: Computer Organization & Design by Patterson and Hennessy; Computer Organization and Architecture by Stallings.

- Super scalar Architecture where multiple instructions can be executed at the same time.
- Single cycle RISC performance.
- Multi stage pipeline (5 stages for the Pentium I and 20 stages for the Pentium 4).
- Multiple parallel execution units (2 fixed ALUs, 2 load/store, 1 branch, 1 floating point).
- Out of order instruction execution.
- Dynamic branch prediction and speculative execution along predicted branches.
- 16 Kbyte instruction cache and 8 Kbyte data cache.

The Pentium attempts to prefetch instructions into the instruction cache 32 bytes at time. Instructions are executed in a multi stage pipeline:

- *Fetch stage*. This brings in the next instruction. If the instruction is a conditional branch then the fetch logic *predicts* whether the branch will be taken and continues *speculative execution* along the predicted branch line. X86 code executes a branch about every 7 instructions. Branch prediction is correct about 75% of the time. Correct predictions keep the pipeline full and maximizes performance. The first time a branch is encountered, the prediction algorithm uses this rule: conditional jumps backwards are predicted to be taken; conditional jumps forwards are predicted not to be taken. For branches that have been encountered before, branch prediction is done using history information. The Pentium I kept a 2 bit history of any branch it encountered. The Pentium II has more stages in its pipeline and so there is a greater performance hit for incorrect prediction. Therefore, the Pentium II implemented a 4 bit history.
- *Two stage decode*. This logic determines what the instruction needs to do and generates the correct entry point into the microcode for complex instructions or the correct internal RISC operations (ROPs) for basic instructions.

ROPs are fixed function internal instructions that execute in a single cycle. Instructions such as register to register adds can be done with one ROP. Each ROP has a fixed length of 118 bits. The ROP contains a field that indicates whether the instruction is *speculative* or not.

The hardware will try to translate CISC instructions into ROPS. Any CISC instruction that can be done with 1-4 ROPs will be translated. An example might be the conversion of *sub ax,[bx+si]* into 3 ROPs.

$$\text{sub ax,[bx+si]} \longrightarrow \begin{array}{lll} \text{temp1} & = [\text{bx+si}] & \text{ROP 1} \\ \text{temp2} & = \text{not}(\text{temp1}) + 1 & \text{ROP 2} \\ \text{ax} & = \text{ax} + \text{temp2} & \text{ROP 3} \end{array}$$

Note that the Pentium has introduced a set of internal registers to hold intermediate results.

Those CISC instructions that need more than 4 ROPs require the use of microcode. An example is any string instruction. These are processed by a separate microcode instruction sequencer ROM that contains the microprogram for all the complex machine instructions.

- *Execute stage*. ROPs are executed by one of the 6 execution units (2 ALUs for integer arithmetic, logic and shifts; 1 floating point unit; 2 load/store units that read and write memory operands; 1 branch unit that executes calls, returns, conditional jumps, and signals the fetch logic when the branch prediction was incorrect). Each execution unit has its own FIFO reservation station with multiple entries. ROPs are dispatched to reservation stations in program order. Up to 4 ROPs can be dispatched in every clock cycle. This means that it is possible that 4 instructions will be in execution at the same time. It is also possible that ROPs will complete out of order.
- *Results stage*. In this stage the result of the execution of a ROP (an intermediate result or part of an instruction) is forwarded to the next user of that result.
- *Retire stage*. During this stage all the results of the execution of ROPs for a given instruction are reordered to assure the total correctness of the execution of that instruction. The final result is routed to the correct final destination (such as a memory operand or register). Up to four instructions can retire per clock period.

Summary of x86 Processors

	Date	Address Bus (Bits)	Data Bus (bits)	Max. Clock Speed	Virtual Memory	Floating Point Coprocessor	Transistors
4004	1971	12	4	0.1 MHz	No	No	2,300
8008	1972	14	8	0.1 MHz	No	No	3,500
8080	1974	16	8	2.0 MHz	No	No	6,000
8086	1978	20	16	10 MHz	No	Yes	29,000
8088	1979	20	8	8 MHz	No	Yes	29,000
8087	1980						
80286	1982	24	16	12.5 MHz	Yes	Yes	134,000
80386	1985	32	32	33 MHz	Yes	Yes	275,000
80486	1989	32	32	100 MHz	Yes	On Chip	1,200,000
Pentium I	1993	32	64	200 MHz	Yes	On Chip	5,500,000
Pentium III	1999	32	64	1.0 GHz	Yes	On Chip	19,000,000
Pentium 4	2001	32	64	2.0 GHz	Yes	On Chip	40,000,000
Core i7 Quad	2008	64	64	2.7 GHz	Yes	On Chip	731,000,000

Notes:

- This chart shows the general characteristics of the x86 processors. There were multiple versions of 386 and 486 processors. Versions with a DX suffix were high performance. Versions with SX suffix were lower cost and performance, often with a smaller data bus and no math coprocessor. Versions with an SL suffix were low power chips used with laptops.
- In 1980 the 8087 Floating Point Coprocessor was announced. It worked with the 8086, 8088, 80286 and 80386.

Common Bugs

One of the most fascinating and difficult parts of software development is the process of isolating defects, affectionately called debugging. Complicated bugs will play havoc with your logical thinking process. I assure you that there will come a time when you will encounter a bug and it will absolutely appear that ‘the hardware is broken’ or the ‘operating system is defective’. Even the most experienced programmer, when under pressure, will think those thoughts. Since those are rarely true, you must work hard to determine the real problem. To paraphrase Sherlock Holmes, first eliminate the impossible then whatever you are left with, no matter how improbable, must be the cause of the problem.

General notes on how to get help debugging your program

- First, you **must** have a plan to solve the problem. This is called a *design*.

If a problem is small enough that it can be totally visualized in your mind, then it may be possible to code the solution to the problem directly from the specification. However, as the problem gets larger and more complicated, coding from the specification is not practical. Picture the following analogy. You could probably build a dog house starting from just an idea and a stack of lumber. You could not build a two story, 4 bedroom, 2 bath house without a detailed blueprint.

A program design is the link between the problem specification and your code. It is where you assure that you know how to solve the problem. It should be written, clear, detailed and correct. If you attack a program without a design, you minimize your probability of success and you will increase the amount of time needed to get a working solution.

How should you develop the design? There are three common approaches:

- A state diagram
- Pseudocode
- An actual high level language that you know the best. This option even allows you to compile your design and run it against the grading program, not for a grade but to demonstrate that the design works. You can then convert your high level language program design into assembler.

Having a design breaks the program into two more manageable pieces. One is your ability to solve the problem. Two is your ability to code the solution in assembler. The few hours spent in doing the design may save you from many painful hours debugging your non working assemble program.

- If the grading program is not happy with the format of your output, but it ‘looks’ correct to you, then you may need to use the following approach to locate the problem.
 - Manually, list character by character the *expected* output from the test case, including CR, LF, EOF.
 - Run the test case directing the output to a file.
 - Using the *testfile* program, in the course locker, manually list character by character the *actual* output generated by your program.
 - Manually, character by character compare the *expected* output to the *actual* output. Where they do not match is the bug.
 - The instructors will ask to see these two lists if you request our help with this type of problem.
- Learn how to use a debugger when it is absolutely needed.

Below are some common bugs that other students have encountered.

- Use of Unix based text editors for DOS work**

If you use an EOS editor to create source files then be careful. EOS is case sensitive. This means you can create these 4 different files: *PROG.ASM* *prog.asm* *prog.ASM* *PROG.asm*. However, DOS is not case sensitive. This means that all of those files will appear to have the same name to DOS, and you cannot have multiple files with the same name in DOS. DOS will be confused and these files will appear with unusual names like *PROG.G61*. What you assemble may not be the version that you want.

- Relative jump out of range**

You get the following error message during the assembly of your program.

```
Assembling file: prog.ASM
**Error** prog.ASM(29) Relative jump out of range by 0041h bytes
Error messages: 1
Warning messages: None
Passes: 1
Remaining memory: 354k
```

The conditional jump has a limited range for its jump distance. If you exceed this limit then you will get an error message indicating the jump is out of range. One technique to get around this is to re-code the test as in the following example where the *je to zero is out of range*. The unconditional jump (jmp) instruction is not limited in its jump distance. The net effect of both pieces of code is the same; if ax=0 then the code jumps to the label zero:

BUG	FIX
cmp ax, 0	cmp ax, 0
je zero	jne notzero
mov [val], ax	jmp zero
.	notzero: mov [val], ax
.	.
zero:	zero:

- INT 21h to read or write data does not seem to work**

A common fault is to forget the hex (h) indicator after the interrupt number (21).

<u>You coded</u>	<u>You meant to code</u>
mov ah,2	mov ah,2
int 21	int 21h

- **You get extraneous dollar signs (\$) in your output**

This can be caused by the following code. You have a character in al; you write a message; then you write the character. Remember that the contents of ax can be altered by the int 21h. Do not assume it holds its value across the first int 21h use.

You want to output: THE ANSWER IS 7

```
message db  'THE ANSWER IS ', '$'  
  
mov  al,'7'           ;set al to contain the character 7  
mov  ah,9             ;set to write a string terminated by a $  
mov  dx,offset message ;point to the string  
int  21h              ;write the string  
                           ;code above destroys the contents of al  
mov  dl,al            ;INCORRECTLY TRY TO MOVE CHARACTER TO DL  
mov  ah,2              ;set up to output a character  
int  21h              ;write the char BUT IT WILL BE A $ INSTEAD
```

- **Strings are sometimes written out as garbage**

Make sure the word register DX not DL points to the string.

```
mov  ah,9           ;set to write a string terminated by a $  
mov  dl,offset message ;YOU LOADED DL INSTEAD OF DX  
int  21h            ;write the string
```

- **Same test case works in Debugger but not when you run the program**

When you execute a program, YOUR CODE must assure that all registers are properly initialized.

However, when you use a Debugger it may initialize the registers to zero (it thinks it is helping you).

So the following code may work in a debugger but not when just executed as a program.

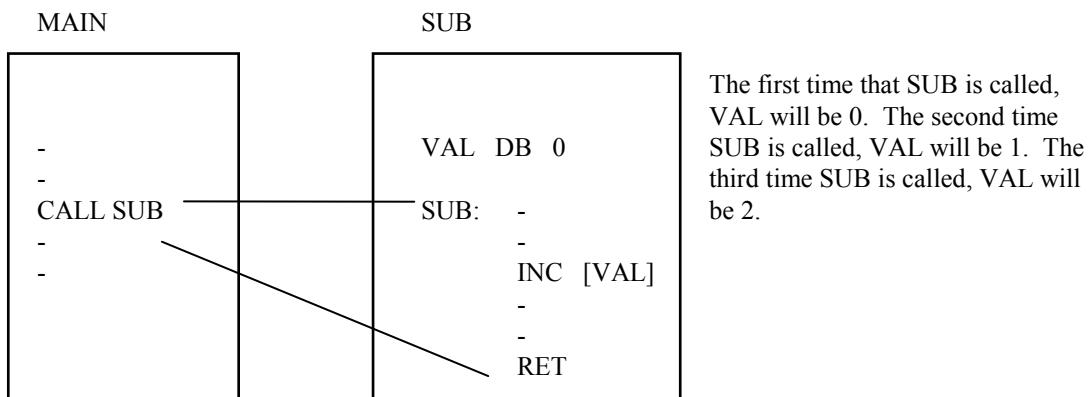
```
mov  dl,10      ;dl has 10, but dh was not initialized, so dx = ?? 0A  
cmp  dx,10      ;compare the value ?? 0A in dx to 00 0A  
je   match      ;if equal jump to match
```

In a debugger, dh may be set to zero and the jump will occur. When just executed as a program dh will have an unknown value and this code may or may not take the jump.

- Your subroutine works correctly when YOU test it but NOT when the grading program runs

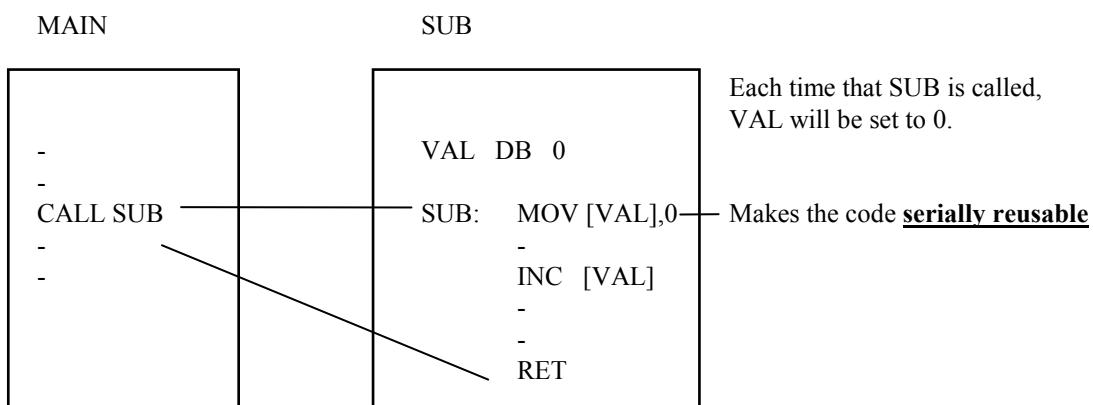
Your *declared variables are only initialized the first time a routine is called*. This especially adversely affects subroutines, which become NOT-REUSABLE. This means you cannot call a subroutine more than once, which of course is not acceptable.

Here is the problem:



The fix is simple, and **must be done for all subroutines!!!!**

You make the code **SERIALLY REUSABLE** by initializing the variables in the subroutine dynamically with move instructions.



- **Miscellaneous syntax errors generated by the *MASM Assembler***

This code has a number of problems that are described in the comment field. You have the source and listing.

```

.model      small
.stack     256

        .data
abc      dw      0
def      dw      0

        .code
start:
        mov      ax,@data
        mov      ds,ax
abc:
        jmp      abc          ;abc already used as name of a variable!
        nop
        je       def          ;def is a constant...can we jump to it?
loop:
        nop          ;word 'loop' is instruction opcode
sub:
        nop          ;word 'sub' is instruction opcode
exit:
        mov      ax,4c00h
        int      21h
        end      start

                                .model  small
                                .stack 256

0000
0000 0000      abc      dw      0
0002 0000      def      dw      0

0000
0000      .code
start:
        mov      ax,@data
        mov      ds,ax
abc:                      ;abc used as variable
label.asm(12): error A2005: symbol redefinition : abc

0005  FF 26 0000 R      jmp      abc          ;which abc does this refer to?
0009  90              nop
                    je       def          ;def is a constant
                    ;...can we jump to it?
label.asm(15): error A2077: inst does not allow NEAR indirect addressing

        loop:                  ;word 'loop' is opcode
label.asm(16): error A2081: missing operand after unary operator

000A  90              nop
        sub:                  ;word 'sub' is opcode
label.asm(18): error A2008: syntax error : in instruction

000B  90              nop
0017
0017  B8 4C00          exit:
001A  CD 21              mov      ax,4c00h
                           int      21h
                           end      start

```

Miscellaneous syntax errors generated by the TASM Assembler

This code has a number of problems that are described in the comment field. You have the source and listing.

```
ideal
model    small
stack    256

        dataseg
abc      dw      0
def      dw      0

        codeseg
start:
        mov      ax,@data
        mov      ds,ax
abc:
        jmp      abc      ;abc already used as name of a variable!
        nop
        je       def      ;def is a constant...can we jump to it?
loop:
        nop      ;word 'loop' is instruction opcode
sub:
        nop      ;word 'sub' is instruction opcode
exit:
        mov      ax,4c00h
        int      21h
        end      start

1      ideal
2      model    small
3      stack    256
4
5      dataseg
6 abc     dw      0
7 def     dw      0
8
9      codeseg
10 start:
11     mov      ax,@data
12     mov      ds,ax
13 abc:           ;abc already used as name of a variable!
**Error** label.ASM(13) Symbol already different kind: ABC

14     jmp      abc      ;which abc does this refer to?
*Warning* label.ASM(14) Pointer expression needs brackets

15     nop
16     je       def      ;def is a constant...can we jump to it?
**Error** label.ASM(16) Argument to operation or instruction has illegal size

17 loop:          ;word 'loop' is instruction opcode
**Error** label.ASM(17) Too few operands to instruction

18     nop
19 sub:           ;word 'sub' is instruction opcode
**Error** label.ASM(19) Too few operands to instruction
*Warning* label.ASM(19) Argument needs type override

20     nop
21 exit:
22     mov      ax,4c00h
23     int      21h
24     end      start
```

CODE COMPLEXITY

You can probably look at section of code and generate an intuitive feel as to how complex it is.

You can also get an intuitive feel of how hard it would be to maintain that code (fix bugs or add new function).

You would probably look at the following characteristics of the code:

- are there comments that explain WHAT the code is doing and HOW it does it
- does the code have a clean flow or does it jump around (spaghetti code)
- is it obvious what are the parameters passed to subroutines; etc.

Researchers have tried to develop formal measures of complexity. The most notable is a numeric method described by Tom McCabe. It counts 'decision points' in a section of code and generates a McCabe value.

Rule to calculate the McCabe number for a section of code

- Start with 1 for the straight path through the code.
- Add 1 for every decision point.
In high level languages this is for each occurrence of an IF, WHILE, REPEAT, FOR, AND, OR, and each case in a CASE statement (and 1 for the case default if not specified).
In assembler a decision point would be any conditional jump instruction or the loop instruction.

Analysis of the McCabe number

1-5	Okay
6-10	Start to think about simplification
10+	Break the code into two pieces. Be reasonable. A case statement with 10 entries would not have to be broken up.

Programming For Efficiency

Efficiency is defined as the completion of a task with the minimum of expense or effort.

Programming for efficiency means that you are trying to minimize the use of some specific subset of the resources needed by a program. These are the four primary resources.

1. Main storage. The amount of main storage needed is affected by these program parameters.
 - The number of instructions written.
 - The relative size of the instructions written. Machine instructions can be 1 to 6 bytes long.
 - The size of the data declared.
2. Execution time. The time to run a program is affected by these program parameters.
 - The number of instructions that are executed to solve the problem.
 - The relative speed of the instructions executed. For example, divide takes longer than addition.
3. Disk space.
4. Time to develop the program. This usually covers the time from the start of the project until the working program is delivered to the customer.

The specific resource you optimize will depend on the situation. Here are three examples.

- Size - You are writing the code to count money deposited and make change for a soda vending machine. You are using single chip that contains a CPU and 64 Kbytes of ROM. This means that your total program must fit within 64 Kbytes. A 70 Kbyte program is too big. If you cannot get the program size down to 64 Kbytes then the vending machine cannot be built.
- Speed - You are writing code to analyze radar data for air traffic controllers to use in landing passenger airplanes. Your code must handle the radar data in real time. If you cannot process the data fast enough then planes may not land safely.
- Development Time - You are writing a commercial Income Tax Program. It must be ready to ship on January 1 of the new year. If it is not ready until April 15 then you will have lost all your customers.

The net is that there will always be an application for which some resource must be optimized. The intent of CSC23x is help you understand how to proceed if you have to program that application.

Writing efficient code is an art, not a science. There is not a set of cookbook rules that will guarantee success. Instead there are *guidelines* that will aid you in writing efficient code.

Important: Your first concern should be to write a correct, well-designed, well-documented, well-tested program. An efficient program that does not work has little value.

Once this is done, then optimization can be applied as required. As changes are made to the program, it must be thoroughly re-tested to assure that no defects have been introduced. It will be desirable to know in which sections of the code the program spends a lot of time. This can be done most easily with a profiler that counts instructions executed. Once the areas of interest have been determined, you can start to apply efficiency guidelines.

The most important advice is to *know what you are doing*. Even when programming in a high level language, there are some basic underlying concepts of computer architecture it is valuable to know. Some general rules follow.

- **Integer arithmetic is faster than floating point.**

The order of speed of execution from fastest to slowest is
 1) add/sub/logical 2) multiply 3) divide 4) subroutine call

- **Store pre-computed results and then just look up the answer.**

There are many times when this can be done and it can really save execution time. For example, sin, cos, square roots all use iterative algorithms for generating results. If you have a bounded set of inputs then you can just pre-calculate all the possible answers for those inputs (e.g. you need to calculate the square root of integers between 0 and 1000) and store them in a table. Then you use the input value as an index into that table to look up the answer. This technique will increase the data space used by the program but will drastically reduce the execution time.

- **Execute the fewest instructions in loops that are executed the most often.**

This can be done through a number of techniques.

- Test for the most common cases first. If end of line occurs less often than data to be processed, then test for end of line last.
- Simplify the loop. You may be better off with multiple loops each doing a specific function, as opposed to one big loop and using flags to control the flow through the loop.

- **Do not calculate things twice.**

Assume you have a subroutine that calculates the distance between two cities, and you are searching for the shortest distance between the city in which you are located and a number of potential destinations. You could use this algorithm:

```

NEXTCITY=1           {nextcity is first candidate      }
CLOSEST=DIST(HERE,NEXTCITY) {calc distance to that city   }
FOR NEXTCITY=2 TO N {test rest of the cities       }
  IF DIST(HERE,NEXTCITY) < CLOSEST { if you calc a shorter distance  }
    THEN CLOSEST=DIST(HERE,NEXTCITY) { then save it as new closest   }

```

Note this calls the DIST subroutine twice if it finds a new closest city. A better approach only calculates the distance once:

```

NEXTCITY=1           {nextcity is first candidate      }
CLOSEST=DIST(HERE,NEXTCITY) {calc distance to that city   }
FOR NEXTCITY=2 TO N {test rest of the cities       }
  NEXTDIST=DIST(HERE,NEXTCITY) {calc distance to the next choice  }
  IF NEXTDIST < CLOSEST { if you calc a shorter distance  }
    THEN CLOSEST=NEXTDIST { then save it as new closest   }

```

- **Simplify tests and calculations.**

Maybe you have the following type of situation:

```
IF X = SQUAREROOT(Y) THEN
begin
.
.
end
```

Calculating the square root is relatively complex. You may be able to avoid that calculation by rewriting the test as:

```
IF X*X = Y THEN
begin
.
.
end
```

- **Rewrite procedures in line for short procedures to reduce calling overhead.**

This example has a short procedure that swaps two data items.

```
SWAP(I,J)           {a short procedure to swap two data items}
TEMP=I
I=J
J=TEMP
RETURN

IF A<B THEN SWAP(A,B)      {a call to that procedure}
```

It may be more efficient to write that code as:

```
IF A<B THEN
TEMP=I
I=J
J=TEMP
```

- **Unroll short loops.**

In short loops the overhead of the loop control is as much as the work performed in the loop. Given LIST1 and LIST2 contain 4 data values, then to move LIST1 to LIST2, you code the following:

using a loop

```
FOR I=0 TO 3 DO  
    LIST2[I] = LIST1[I]
```

unrolling the loop

```
LIST2[0]=LIST1[0]  
LIST2[1]=LIST1[1]  
LIST2[2]=LIST1[2]  
LIST2[3]=LIST1[3]
```

The version using the loop has fewer high level language statements.

However, when coded in C/C++, my compiler generated 7 instructions that were executed to move each element and update and test the loop variable I. This yielded a total execution of 29 instructions (1 instruction to set I to zero, and then 7 instructions executed for each of the four data items moved).

For the unrolled version, the compiler generated two instructions to move each data item. This yielded a total of 8 instructions. If you were coding in assembler, you could probably do better for both cases, but the unrolled version would probably execute fewer instructions.

- **Don't calculate constants.**

```
PI=3.14159          { pi is constant}  
CIRCUMFERENCE = 2 * PI * R      { two multiplications required }
```

Instead we should also have a constant for 2*pi.

```
PI=3.14159          { pi is constant}  
TWOPI=6.28318       { 2*pi is a constant}  
CIRCUMFERENCE = TWOPI * R      { one multiplication required }
```

- Trade **instructions written** for **instructions executed**.

Here is C/C++ code that tests two variables (*a* and *val*). It uses multiple ***if*** statements. The compiler generates 10 Instructions Written. For any combination of values there will be 8 Instructions Executed.

```
if (a < val) count--;
if (a == val) count=0;
if (a > val) count++;
```

Assembler Code Generated

```
if (a < val) count--;
    mov ax,WORD PTR [val]      ;val
    cmp WORD PTR [a],ax        ;a:val
    jge $I172                  ;a >= val so skip
    dec WORD PTR [count]      ;count--
if (a == val) count=0;
    $I172:
    cmp WORD PTR [a],ax      ;a:val
    jne $I173                  ;a != val so skip
    mov WORD PTR [count],0     ;count=0
if (a > val) count++;
    $I173:
    cmp WORD PTR [a],ax      ;a:val
    jle $I174                  ;a <= val so skip
    inc WORD PTR [count]      ;count
```

The following code does the same function, but it uses ***If ... then ... else***. The compiler generates more Instructions Written (12) but the number of Instructions Executed will now vary from 5 to 8 depending on the values of the variables. So the programmer must decide which is more important, instructions written or instructions executed.

```
if      (a < val) count--;
else if (a == val) count=0;
else if (a > val) count++;
```

Assembler Code Generated

```
if      (a < val) count--;
    $I174:
    mov ax,WORD PTR [val]      ;val
    cmp WORD PTR [a],ax        ;a:val
    jge $I175                  ;a >= val so skip
    dec WORD PTR [count]      ;count--
    jmp SHORT $I179             ;exit
else if (a == val) count=0;
    $I175:
    cmp WORD PTR [val],ax     ;a:val
    jne $I177                  ;a != val so skip
    mov WORD PTR [count],0     ;count=0
    jmp SHORT $I179             ;exit
else if (a > val) count++;
    $I177:
    cmp WORD PTR [a],ax      ;a:val
    jle $I179                  ;a <= val so skip
    inc WORD PTR [count]      ;count
$I179:
```


Elements of Program Design

What is a design and why is it important?

The function of a *design* is to convert an English language problem statement into a series of logical algorithmic steps that can be coded.

The goal of a design is to assure that the problem is understood well enough to be correctly solved. The design is done before the program is coded. A program design serves the same function as the blueprint for a house.

When coding in Assembler, more than any other language, a design is critical. This is why. One popular design technique, *pseudocode*, is very similar to actually coding in a high level language such as C/C++ or Java. Therefore, if one is coding a small program in C/C++ or Java then the design step does not *appear* to have much value. Thus it is often skipped for small C/C++ and Java programs without causing significant problems. In addition you may be used to C/C++ and Java programs where you can picture the whole solution in your mind. When this is true you can also skip the design process without causing significant problems.

These reasons for skipping the design process will not be true when programming in Assembler Language which is extremely detail oriented. Failure to design in Assembler is the first step toward failure.

Building a design fulfills three requirements:

1. It allows the programmer to think in a structured fashion.
2. It provides a solid basis for converting the logical statements into assembler code.
3. It provides an easy to read overview of the program for someone who needs to modify or fix it.

Completely skipping the design in Assembler will lead to major problems such as more inefficient solutions, longer development time, difficult debugging, and sometimes an inability to create a functioning program. It is not necessary to build a complete formal design document to improve your chances of success in Assembler; just a basic written design is guaranteed to help.

Characteristics of a good design include these two items.

- An outside observer should be able to take the problem specification and the design and verify that the design will solve the problem.
- It should be possible to implement the design in any programming language.

An excellent way to manually verify your design is to define some short sample input data and manually walk that data through your design, following exactly the instructions in the design. This process works best if an outside observer performs this task. The reason is that the outside observer *does not know what you wanted to do*; they only *know what you actually said to do*.

An excellent way to automatically verify your design (for an Assembler program) is to create the design in a high-level language and then actually run the program.

So in order of desirability (for Assembler programs) the design should be:

1. Coded in a high-level language that can be executed.
2. Written in Pseudocode.
3. Drawn as a finite state machine or flowchart.

How can you gauge your ability to design a solution?

To measure your ability to create a design, follow these steps.

- Read the problem statement for the KEY assignment.
- Map out a detailed algorithm to solve the problem. Assume that *you* will not be coding the solution. Instead, your design will be given to a novice program and they will code the program using your design. Your design should be programming language independent. Specify the error conditions the programmer should handle.
- Compare your solution to the solution given. If you need help in determining whether your solution works then see the TA or instructor.

We will present three designs techniques.

- Finite State Machines, which graphically represent the sequentially processing of one item at a time, such as reading keystrokes from a keyboard.
- Pseudocode, which is similar to programming in a high-level language.
- The program written in C and Java.

Problem statement for KEY

KEY reads printable characters (20h-7Fh) from the keyboard without echo (using ah=08h and int 21h).

The characters are processed one at a time as they are read. For each character read perform the following:

- If the character is an upper case letter (A-Z) then write it to the display (ah=02h, dl=char, int 21h).
- If the character is a lower case letter (a-z) then convert it to upper case and write it to the display. You can perform this conversion by subtracting 20h from the lower case letter.
- If the character is a blank (20h) or period (2Eh) then write it to the display.
- If the character is anything else then do not write it to the display, just throw the character away.

After processing the character, if the character is a period (2Eh) then end your program's execution.

For example if you run your program and type in this data: **Hello, testing 123.**

Then you would expect the display to show: **HELLO TESTING .**

That specification is already clearer and more structured than anything you will ever encounter in the real world.

Even so, it has some ambiguities.

For example, it does not strictly dictate the order of events. Do you read all the characters into a list and then process them, or do you read and process one character at a time?

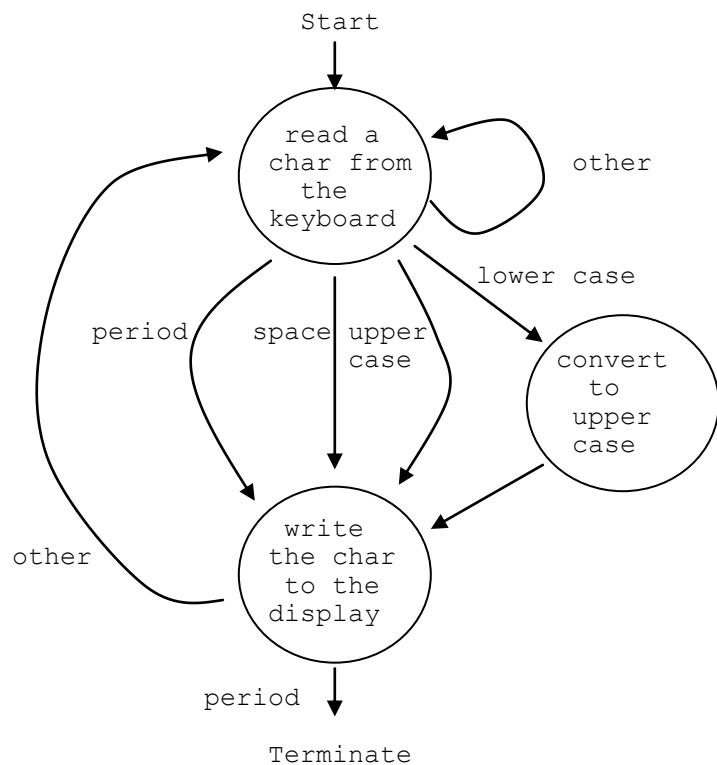
Real problem statements will have many more vagaries that need to be resolved. The design will resolve all the issues.

A finite state machine design (FSM) for KEY

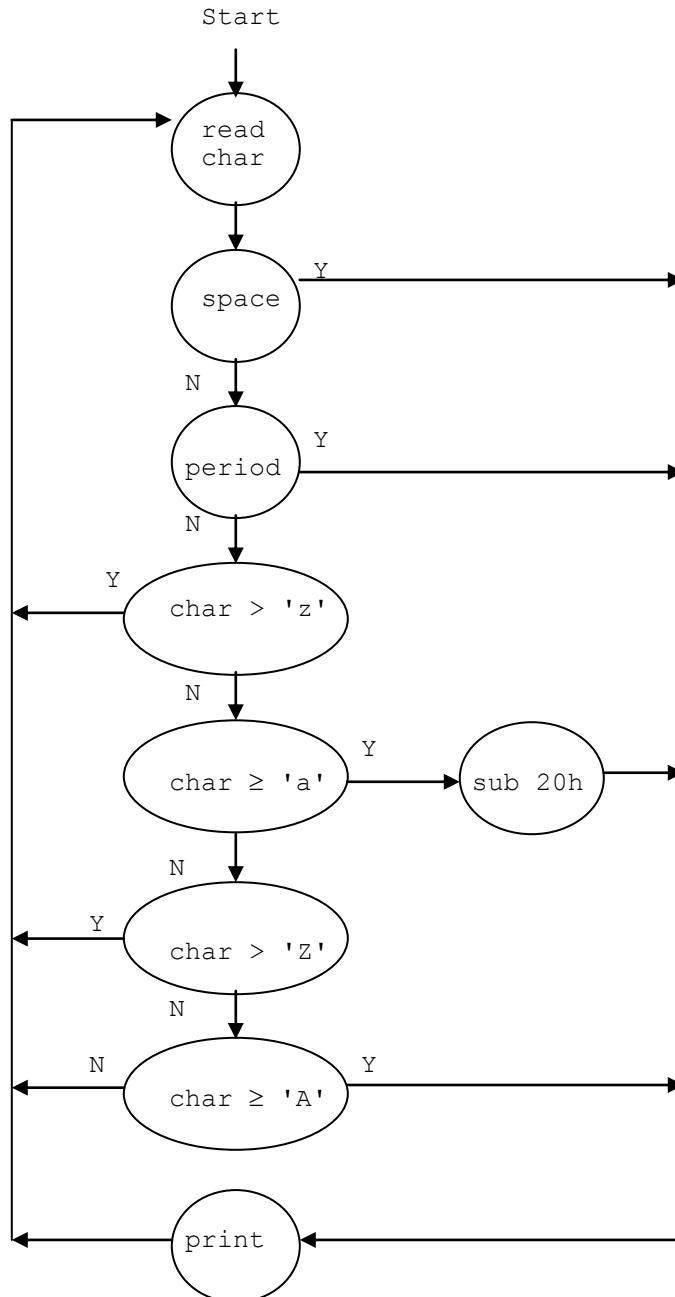
An FSM has two components.

- States, which represent the current environment of the system. States are represented by circles and the work done by the state is written inside the circle.
- Transitions that represent how the system moves from one state to another. Transitions are represented by arrows. Any decision-making information is written alongside the arrow.

FSM for KEY.
Decision information on the arrows is based upon
the character that was just read or written.



It may be desirable to break down the state that reads a character and then identifies the character. This would make the conversion to assembler language easier. We now present a design that is a meld of FSM and flowcharting with more detail than the original FSM. The order of performing the tests is important. We test all the specific equal conditions first. Then we test against the collating sequence of ASCII characters starting with the largest values.



ASCII Collating Sequence	
<u>value</u>	<u>char</u>
127	.
122	z
97	a
90	Z
65	A
46	period
32	space
0	

Pseudocode design for KEY

Pseudocode is a form of structured English. It is not intended to be executed on a machine, but it is used to organize the solution prior to coding. It uses the standard constructs of high level languages such as *if-then-else* but does not require the format of any specific language such as ending all statements with a semicolon. Since your design may be read by someone who is not familiar with some specific high-level language, it is always a good idea to specify any special conventions used in your pseudocode design.

Elements of pseudo code used in this design.

- Any statement that starts with two slashes // is a comment.
- 'Begin' and 'end' blocks are designated by braces { }
- The symbol for the test for equal is == and for not equal is !=

```
// KEY High Level Design
// Test against characters in ASCII collating sequence order.
// That means it tests for: space, period, upper case, lower case
// This will make final coding much easier.
// Declare the variable 'char' as an unsigned byte that will hold
// ASCII characters read from the keyboard

// Initialize char to hex 0
char = 0;

// Loop while character read is not a period
while ( char != '.' )
{
    read char from the keyboard
    if char is a space ( ' ' ) then write char to the display
    else if char is a period ( '.' ) then write char to the display
    else if char is an upper case letter then write char to the display
    else if char is a lower case letter then convert char to upper case
        and write converted char to the display
    else { } // for all other ASCII characters do nothing
}
terminate program.
```

We now take a major step forward by creating a low level design, with more detail. The goal of the low level design is to help us convert the design steps into actual code.

Since assembler statements perform very basic operations, you will find that it will be best to break compound design statements into individual design statements.

For example, to test if a character is an upper case character we could use the compound statement:
if ((char \geq 'A') and (char \leq 'Z')) then

However, you will see that we have broken that compound statement into multiple single statements to ease the transition from design into actual assembler code. This process is also helped by the fact that we test the character in ASCII collating sequence.

```
// KEY Low Level Design
// Design does its tests against characters in ASCII collating sequence order.
// That means it tests for: space, period, upper case, lower case
// This will make final coding much easier.
// Declare the variable 'char' as an unsigned byte that will hold
// ASCII characters read from the keyboard

// Initialize char to hex 0
char = 0;

// Loop while character read is not a period
while ( char != '.' )
{
    read char from the keyboard
    if      ( char == ' ' ) {write char}                      // write spaces
    else if ( char == '.' ) {write char}                      // write periods
    else if ( char < 'A' ) { }                                // throw away below 'A'
    else if ( char  $\leq$  'Z' ) {write char}                      // write upper case
    else if ( char < 'a' ) { }                                // throw away below 'a'
    else if ( char  $\leq$  'z' ) { char = char - 20h           // convert to upper
                            write char }                      // write upper case
    else { }                                                 // throw away anything else
}
terminate program.
```

This Low Level Design is perfectly adequate from a functional viewpoint. However, we may wish to look at improving efficiency.

From the section in the notes titled *Programming For Efficiency* we are told not to calculate the same thing more than once. In our design we have multiple occurrences of the operation *write char to the display*. So we will make one more pass over the design and isolate that calculation.

We introduce one more pseudocode convention.

The *continue* operation passes control to the next iteration of the *while loop* thus bypassing any remaining statements in the loop.

For our design, the continue statement will bypass the code that writes the character to the display.

```
// KEY Low Level Design
// Design does its tests against characters in ASCII collating sequence order.
// That means it tests for: space, period, upper case, lower case
// This will make final coding much easier.
// Declare the variable 'char' as an unsigned byte that will hold
// ASCII characters read from the keyboard

// Initialize char to hex 0
char = 0;

// Loop while character read is not a period
while ( char != '.' )
{
    read char from the keyboard
    if      ( char == ' ' ) { }                                // write spaces
    else if ( char == '.' ) { }                                // write periods
    else if ( char < 'A' ) { continue }                         // throw away below 'A'
    else if ( char ≤ 'Z' ) { }                                // write upper case
    else if ( char < 'a' ) { continue }                         // throw away below 'a'
    else if ( char ≤ 'z' ) { char = char - 20h } // convert to upper
    else { continue }                                         // throw away anything else
    write the char to the display
}
terminate program.
```

Once you have the pseudocode, you can add any language specific directives and port the pseudocode to a high-level language that can actually be executed.

A version of KEY written in C.

```
-----  
// Prog:      Key - a solution to the KEY homework written in 'C'  
//  
// Owner:     Dana Lasher  
//  
// Logic:  
//   Read characters from the keyboard  
//   If the char is uppercase then write it to the display  
//   If the char is lowercase then convert to upper and write to display  
//   If the char is blank or period write it to the display  
//   If the char is anything else then throw it away  
//   Stop after reading a period  
//  
// Date:      Reason  
// 05/14/2010 Original version  
//-----  
  
#include <stdio.h>                      // standard input routines  
#include <conio.h>                        // console routines  
  
int main()  
{  
    int ch = 0;                            // char set to zero  
  
    while ( ch != '.' )                   // loop while char read  
    {  
        ch = getch();  
        if      ( ch == ' ' )           { }          // write spaces  
        else if ( ch == '.' )           { }          // write periods  
        else if ( ch < 'A' )           { continue; } // throw away below 'A'  
        else if ( ch <= 'Z' )          { }          // write upper case  
        else if ( ch < 'a' )           { continue; } // throw away below 'a'  
        else if ( ch <= 'z' )          { ch = ch - 32; } // convert lower to upper  
        else                           { continue; } // throw away anything else  
        putchar(ch);                  // echo it  
    }  
  
    return (0);                          // terminate  
}
```

A version of KEY written in Java

Java is buffered, so the code reads a line and produces output when enter is pressed.

```
import java.io.*;
import java.lang.*;

public class Key{

    // printer: Prints to the screen whatever is passed in to it.
    public static void printer(int c){ System.out.print((char)c); }

    // smallCap: Boolean, returns true if char passed to it is lower case
    public static boolean smallCap(int c){
        if( c >= 'a' && c <= 'z') return isSmallCap = true;
        else return isSmallCap = false;      }// End smallCap

    // bigCap: Boolean, returns true if char passed to it is a upper case
    public static boolean bigCap(int c){
        if( c >= 'A' && c <= 'Z' ) return isBigCap = true;
        else return isBigCap = false;      }// End bigCap

    public static boolean isSmallCap = false;
    public static boolean isBigCap   = false;

    // Process each character and terminates with a period

    public static void main (String args[]){
        try{

            // Read characters until EOF (-1) or a period (.)
            int i = 0;

            while( (i != '.') && (i != -1) ) {

                // read a key
                i = System.in.read();

                // Convert lower case to upper case and print character
                if (smallCap(i)){ i = i-32; printer(i); }

                // Print upper case
                else if( bigCap(i) ) printer(i);

                // Print space
                else if(i == ' ') printer(i);

                // Print period
                else if(i == '.') printer(i);

            } // end while

        } // End try

        catch(Exception e){ System.out.println("Exception: " + e ); }

    } // End Main

} // End Key class
```


8086 Register Usage Summary

The following tables contain a summary of how the 8086 registers are used.

Reg	General Use	Special Purpose Use
ax	<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register called ax or two 8 bit registers called ah and al. 	<ul style="list-style-type: none"> DOS read and DOS read with echo. al receives the input character on a DOS read or a DOS read with echo. Convert signed byte to word (cbw). Converts the signed byte in al to a signed word in ax. Multiply. $ax = \text{byte} * al$ $dx:ax = \text{word} * ax$ Divide. $ax / \text{byte} \rightarrow ah = \text{rem} \quad al = \text{quot}$ $dx:ax / \text{word} \rightarrow dx = \text{rem} \quad ax = \text{quot}$
bx	<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register called bx or two 8 bit registers called bh and bl. 	<ul style="list-style-type: none"> Indirect addressing. bx assumes that the data is in the data segment.
<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register called cx or two 8 bit registers called ch and cl. 	<ul style="list-style-type: none"> Loop instruction. cx contains the loop count as an unsigned word. String instructions. cx contains the loop count as an unsigned word. 	
dx	<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register called dx or two 8 bit registers called dh and dl. 	<ul style="list-style-type: none"> DOS character write. dl contains the character for a DOS character write. DOS string write. dx contains the string offset for DOS string write. Multiply. $dx:ax = \text{word} * ax$ Divide. $dx:ax / \text{word} \rightarrow dx = \text{rem} \quad ax = \text{quot}$

Reg	General Use	Special Purpose Use
si	<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register. 	<ul style="list-style-type: none"> Indirect addressing. si assumes the data is in the data segment. String instructions. si is the index for the source string.
di	<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register. 	<ul style="list-style-type: none"> Indirect addressing. di assumes the data is in the data segment. String instructions. di is the index for the destination string.
bp	<ul style="list-style-type: none"> Arithmetic calculations. Used as one 16 bit register. 	<ul style="list-style-type: none"> Indirect addressing. bp assumes data is in the stack segment
sp		<ul style="list-style-type: none"> Stack Pointer. sp contains the offset into the stack segment of the current stack entry. Do not use it for any other function.

Indirect Addressing Components	Format
	<ul style="list-style-type: none"> • These are the only valid combinations. • In calculating the effective address, the arithmetic is performed and any carries or overflows are ignored.
base or index	[bx] [bp] [si] [di]
base or index plus/minus displacement	[bx ± n] [bp ± n] [si ± n] [di ± n]
base and index	[bx + si] [bx + di] [bp + si] [bp + di]
base and index plus/minus displacement	[bx + si ± n] [bx + di ± n] [bp + si ± n] [bp + di ± n]

The order that you code the parameters is *not significant*.

For example, all of these are identical.

- [bx + si + 1]
- [si + bx + 1]
- [1 + si + bx]
- [1 + bx + si]
- [bx + 1 + si]
- [si + 1 + bx]

- These all point to the data segment (the forms that do *not* use bp)
[bx] [si] [di] [bx ± n] [si ± n] [di ± n] [bx + si] [bx + di] [bx + si ± n] [bx + di ± n]

- These all point to the stack segment (any form that uses bp)
[bp] [bp ± n] [bp + si] [bp + di] [bp + si ± n] [bp + di ± n]

- Using the segment override operation (ds: es: ss: cs:) you can force the system to go to any specific segment.

TASM places the override inside the brackets ... [ds:bp + si]
MASM places the override outside the brackets ... ds:[bp + si]

Reg	Special Purpose Use
cs	<ul style="list-style-type: none"> • Contains the high 4 hex digits of the starting absolute address of the code segment. • Initialized by DOS.
ds	<ul style="list-style-type: none"> • Contains the high 4 hex digits of the starting absolute address of the data segment. • Initialized by the programmer.
ss	<ul style="list-style-type: none"> • Contains the high 4 hex digits of the starting absolute address of the stack segment. • Initialized by DOS.
es	<ul style="list-style-type: none"> • Contains the high 4 hex digits of the starting absolute address of the extra segment. • Initialized by the programmer.
ip	<ul style="list-style-type: none"> • Contains the offset into the code segment of the next instruction to be executed. • Initialized by DOS.

ASCII Codes

Dec	Hex		Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	space	64	40	@	96	60	`
1	01		33	21	!	65	41	A	97	61	a
2	02		34	22	"	66	42	B	98	62	b
3	03		35	23	#	67	43	C	99	63	c
4	04		36	24	\$	68	44	D	100	64	d
5	05		37	25	%	69	45	E	101	65	e
6	06		38	26	&	70	46	F	102	66	f
7	07	Bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Tab	41	29)	73	49	I	105	69	i
10	0a	LF	42	2a	*	74	4a	J	106	6a	j
11	0b		43	2b	+	75	4b	K	107	6b	k
12	0c		44	2c	,	76	4c	L	108	6c	l
13	0d	CR	45	2d	-	77	4d	M	109	6d	m
14	0e		46	2e	.	78	4e	N	110	6e	n
15	0f		47	2f	/	79	4f	O	111	6f	o
16	10		48	30	0	80	50	P	112	70	p
17	11		49	31	1	81	51	Q	113	71	q
18	12		50	32	2	82	52	R	114	72	r
19	13		51	33	3	83	53	S	115	73	s
20	14		52	34	4	84	54	T	116	74	t
21	15		53	35	5	85	55	U	117	75	u
22	16		54	36	6	86	56	V	118	76	v
23	17		55	37	7	87	57	W	119	77	w
24	18		56	38	8	88	58	X	120	78	x
25	19		57	39	9	89	59	Y	121	79	y
26	1a	EOF	58	3a	:	90	5a	Z	122	7a	z
27	1b	Esc	59	3b	;	91	5b	[123	7b	{
28	1c		60	3c	<	92	5c	\	124	7c	
29	1d		61	3d	=	93	5d]	125	7d	}
30	1e		62	3e	>	94	5e	^	126	7e	~
31	1f		63	3f	?	95	5f	_	127	7f	□

The values 00h - 1Fh are control characters. They are **not** meant to be printed or displayed. If you do so, they will appear strange.

If you get a strange symbol in your output then you have surely written a binary value such as 01h instead of an ASCII value such as 31h.

ARM Architecture

Assembler Language And ARMSim Assembler/Simulator

Introduction

ARM is a computer architecture originally designed by a British company, Acorn Computers, and originally stood for Acorn RISC Machine. It is currently developed by a successor British company, the ARM Holdings PLC* and now stands for Advanced RISC Machine.

ARM Holdings does not manufacture chips. It develops and then licenses the ARM architecture to chip manufacturers such as Advanced Micro Devices and Texas Instruments that do build chips.

ARM is the most prolific and widely used chip for modern hi-tech devices. It is used in smart phones such as Apple iPhone, digital televisions, and mobile computing devices such as Apple iPad.

ARM is an example of a Reduced Instruction Set Computer (RISC) architecture. There are many versions of the ARM architecture (ARMv1 through ARMv7 are 32 bit architectures and ARMv8 is the new 64 bit architecture). For CSC236 we will study the basic generic 32 bit ARM architecture.

The ARM architecture is very rich in function and sophistication. My copy of the ARMv7 Architecture Reference Manual has over 2700 pages. Thus we only provide an overview of the architecture and enough assembler information to write basic programs.

ARM has these characteristics:

- Load / Store architecture. Memory variables can only be accessed by instructions that load them into registers.
All arithmetic operations are done on registers. Results then must be stored back into memory.
- Machine instructions are all fixed size of 32 bits.
- Most instructions execute in a single cycle.

This document will provide an overview of the ARM architecture and then an overview of coding ARM assembler code that can run on the ARMSim assembler and simulator developed at University of Victoria.

*PLC is a *Public Limited Company* whose shares may be sold to the public.

References:

- ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition, ARM Holdings PLC
- The ARMSim# User Guide1 R. N. Horspool, W. D. Lyons, M. Serra Department of Computer Science, University of Victoria
- ARM Assembly Language, William Hohl CRC Press

Programmer's Model

The processor has 7 operating modes.

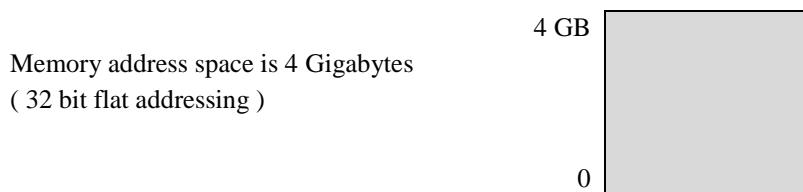
Supervisor	Used for Operating Systems services (similar to DOS int 21h)
FIQ	High priority (fast) interrupt
IRQ	Normal priority interrupt
Abort	Memory access violations e.g. accessing a word on an odd byte boundary
Undef	Handle attempted execution of undefined instructions.
System	Privilege mode ... sharing the same registers as User mode
User	Applications

Applications run in User Mode which is what we will discuss.

Three data sizes are supported. The data can be unsigned or signed two's complement.

- Byte (8 bits)
- Halfword (16 bits) - must be aligned on an address which is a multiple of 2.
- Word (32 bits) - must be aligned on an address which is a multiple of 4.

The architecture allows hardware designers to select whether data is stored as Big Endian or Little Endian. The default is Little Endian.



In User Mode there are 16 registers (r0-r15). Each register is 32 bits.

- R0-R12 are general purpose and available to the programmer.
- R13 is the Stack Pointer.
- R14 is Link Register and holds the return address for subroutines.
- R15 is the Program Counter and points the instruction being fetched.

There is a Status Register. The four left most significant bits represent the Condition Code which is similar to the x86 (N, Z, V are the same but C is different and will be explained further).

N	Z	C	V	
---	---	---	---	--	-------

- N Left most bit of the result. For signed numbers 0=positive and 1=negative
- Z 1 = The result is zero 0 = The result is not zero
- C 1 = There was a carry out of the most significant bit on an addition (same as x86)
1 = There was a carry out of msb on two's complement sub ... this is the opposite to x86
Another way to say this is a 1 means the result of an unsigned subtraction is correct
- V 1 = signed overflow 0 = no signed overflow

Let us examine the Carry Flag further.

The Carry Flag usage is clear for addition. It is set to 1 if there is a carry out of the most significant bit.

For subtraction, there are two ways architects use the Carry Flag.

- As a Borrow Flag. The x86 uses the Carry Flag to indicate if a borrow into the most significant digit was required for an unsigned subtraction. When doing a subtraction A-B by adding the complement A+(-B) the two's complement carry will always be the reverse of the real borrow value. Thus the x86 sets the Carry Flag to the inverse of the carry obtained doing the complement addition.
- As a Carry Flag. The Carry Flag is set to the value of the carry obtained doing the complement addition. ARM adopts this use of the Carry Flag. Is the opposite of what is done by the x86. In summary C=1 for an add is an unsigned overflow, but C=1 for a subtract means the result was correct.

Similar to the x86 you can test the condition code with conditional branches. These are mnemonics and flags tested by the ARM architecture.

To create a branch instruction, just precede the mnemonic with 'b' such as *breq*.

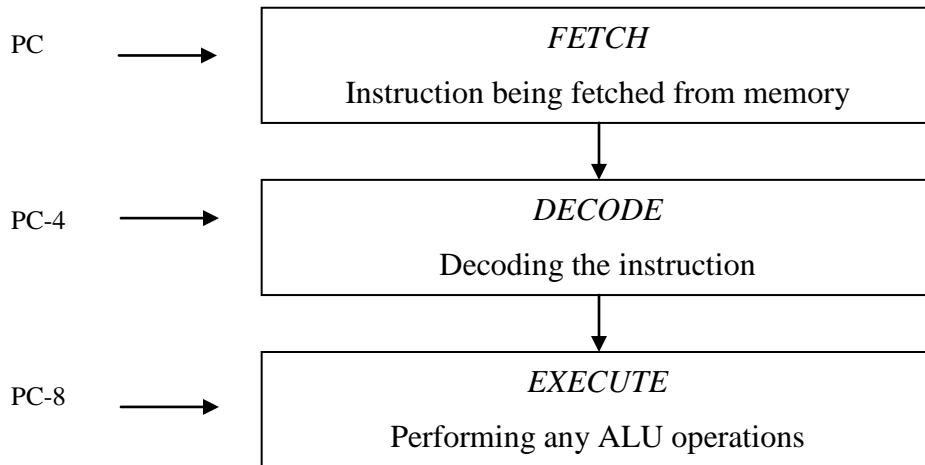
An unconditional branch would be *bal* or just *b*.

Conditional Mnemonic	Flags	Arithmetic	Compare (like a sub)
Unsigned			
CS	Carry Set	C=1	Carry Flag set - add had overflow - sub no overflow dest \geq source
CC	Carry Clear	C=0	Carry Flag cleared - add no overflow - sub had overflow dest $<$ source
HI	High	C=1 and Z=0	sub result > 0 dest $>$ source
LS	Lower or Same	C=0 or Z=1	sub result ≤ 0 dest \leq source
SIGNED			
PL	Plus	N=0	result positive
MI	Minus	N=1	result negative
VC	Overflow Clear	V=0	no signed overflow
VS	Overflow Set	V=1	signed overflow
GE	Greater or Equal	N=V	result ≥ 0 dest \geq source
GT	Greater Than	Z=0 and N=V	result > 0 dest $>$ source
LE	Less or Equal	Z=1 or N \neq V	result ≤ 0 dest \leq source
LT	Less Than	N \neq V	result < 0 dest $<$ source
BOTH UNSIGNED AND SIGNED			
EQ	Equal	Z=1	equal dest = source
NE	Not Equal	Z=0	not equal dest \neq source
AL	Always		unconditional

To create a branch instruction, just precede the mnemonic with 'b' such as *breq*.

An unconditional branch would be *bal* or just *b*.

ARMv7 and earlier versions had a three stage pipeline. The Program Counter (PC) points to the instruction being fetched. Execution of each instruction is broken into three steps: fetch, decode, execute. Another way to state that is that during any one machine cycle, three instructions are in progress. By overlapping these three stages, an instruction can be completed in each machine cycle instead of every three machine cycles/



A performance issue arises when a branch instruction is encountered in the instruction stream. The processor is fetching instructions sequentially. If in the execute stage, a branch is taken then the two instructions behind it in the pipeline are not to be executed. Thus the pipeline must be drained and the hardware must start fetching instructions from the branch target instruction. This hurts performance.

There are techniques to reduce the need to drain the pipeline that will be covered in class.

Load And Store Instructions That Access Memory

rd=destination register rn=first source register rm=second source register

Instr	Operands	Notes
LDR	rd, =variable	rd = address of the variable
	rd, =constant	rd = value of the constant
	rd, ='c'	rd = value of the ASCII character c
	rd, [rn]	rd = value pointed to by rn
	rd, [rn, rm]	rd = value pointed to by rn + rm
	rd, [rn, #n]	rd = value pointed to by rn + #n
	rd, [rn],#n	rd = value pointed to by rn (then rn = rn + #n) this is called <i>post increment</i> and is very useful for updating a pointer as the code moves through a list.
There are variations for LDR that are given below		
LDR	loads a 32 bit value into a 32 bit register	
LDRH	loads an unsigned half word into the low half of a 32 bit register and the high half word is cleared to zero	
LDRB	loads an unsigned byte into the low byte of a 32 bit register and the high 3 bytes are cleared to zero	
LDRSH	loads a signed half word into the low half of a 32 bit register and the high half word is sign extended	
LDRSB	loads a signed byte into the low byte of a 32 bit register and the high 3 bytes are sign extended	

Instr	Operands	Notes
STR	rd, [rn]	the memory pointed to by rn = rd
	rd, [rn, rm]	the memory pointed to by rn + rm = rd
	rd, [rn, #n]	the memory pointed to by rn + #n = rd
	rd, [rn],#n	the memory pointed to by rn = rd (then rn = rn + #n) this is called <i>post increment</i>)
There are variations for STR that are given below		
STR	stores the 32 bit value in rd into memory	
STRH	stores the low half word of rd into memory	
STRB	stores the low byte of rd into memory	

Move - Copy One Register To Another, Move Constant To A Register

Instr	Operands	Notes
MOV	rd, rn	copies rn into rd
MOV	rd, #n	move a limited range of constants to rd ... see note below

In the move instruction, the immediate value of #n is limited in its range. If you get an error message that the value does not fit then you need to use the LDR instruction to load #n into a register. See a later discussion of the specific limits on #n.

Add And Subtract Instructions

The programmer controls whether add and subtract set the condition code. To have the instruction set the condition code you add the suffix 's' to the instruction. So 'add' will not set the condition code and 'adds' will set the condition code.

Instr	Operands	Notes
ADD	rd, rn, rm	rd = rn + rm
ADD	rd, rn, #n	rd = rn + #n
ADC	rd, rn, rm	rd = rn + rm + carry_flag
ADC	rd, rn, #n	rd = rn + #n + carry_flag
SUB	rd, rn, rm	rd = rn - rm
SUB	rd, rn, #n	rd = rn - #n
SBC	rd, rn, rm	rd = rn - rm - carry_flag
SBC	rd, rn, #n	rd = rn - #n - carry_flag

The immediate value #n is limited in range. If you get an error message the value does not fit then you need to use LDR to load #n into a register and then do a register to register addition. See a later discussion of the specific limits on #n.

Compare

Instr	Operands	Notes
CMP	rd, rn	calculates rd - rn and sets the condition code
CMP	rd, #n	calculates rd - #n and sets the condition code

The immediate value #n is limited in range. If you get an error message the value does not fit then you need to use LDR to load #n into a register and then do a register to register compare. See a later discussion of the specific limits on #n

Conditional Execution Of Instructions

ARM instructions can be *conditionally executed*. This is accomplished by appending the conditional mnemonic to the instruction mnemonic. This allows one to eliminate some conditional branches that could cause the pipeline to drain. This example calculates the absolute value of r4 (it assumes r0=0).

Standard code <pre>cmp r4, #0 ; test the value in r4 bpl skip ; if positive, no action subr 4, r0, r4 ; if negative then r4=0-r4 skip:</pre>
--

<i>Conditional Execution</i> of the subtract instruction <pre>cmp r4, #0 ; test the value in r4 submi r4, r0, r4 ; if r4 is negative then r4=0-r4</pre>
--

Multiply And Divide

We present two basic forms of the multiply instruction in which two 32 bit registers are multiplied producing a 64 bit product. There are other forms that can be found in an ARM Functional Specification document.

Instr	Operands	Notes
UMULL	rdlo, rdhi, rm, rs	rdhi:rdlo = rm * rs This is unsigned multiply
SMULL	rdlo, rdhi, rm, rs	rdhi:rdlo = rm * rs This is signed multiply
		Notes: <ul style="list-style-type: none"> • (32 bit) * (32 bit) = 64 bit product • Do not specify rdhi or rdlo as rm • Do not specify rdhi and rdlo as the same register • Both multiplies can be conditionally executed • Both can optionally set the N and Z bits in the condition code (UMULLS, SMULS)

What about division? Many ARM cores do not provide a hardware divide instruction. A probable reason is that divide circuitry: is infrequently used; would require too many circuits; would use a lot of power. If divide is needed on a processor that does not support the hardware instruction, then it must be done in a software subroutine.

We present the divide instructions format for those who may be using a core that supports them.
The divide is a (32 bit register) divided by (32 bit register) producing a 32 bit quotient.

Instr	Operands	Notes
UDIV	rd, rm, rn	rd = rm / rn This is unsigned divide
SDIV	rd, rm, rn	rd = rm / rn This is signed divide
		Notes: <ul style="list-style-type: none"> • The condition code is not updated. • No indication of an overflow is produced, and the 32-bit result written to rd will be the bottom 32 bits of the result.

The hardware divide instructions will not produce a remainder as was the case with the x86 divide; but you can get it with software. The remainder can be calculated as $remainder = dividend - (quotient * divisor)$

15/6 yields a quot=2 ... now calculate $15 - (2*6) = 3$ which is the remainder of 15/6

As it turns out there is a version of multiply that will help with that calculation. I was not going to cover it because it seemed too special purpose; but here is a good use of it. The *mla* (multiply and add) and *mls* (multiply and subtract) instructions multiplies two registers then either adds the product to a third register or subtracts the product from a third register and stores the result in a fourth register. So here is the code to calculate the mod or remainder of a division.

```

;input -    r0 = dividend    r1 = divisor
;output -   r2 = quotient    r3 = remainder
udiv  r2,r0,r1      ;r2 = quotient of (r0/r1)
mls   r3,r2,r1,r0      ;r3 = r0 - r2*r1

```

Logical Operations

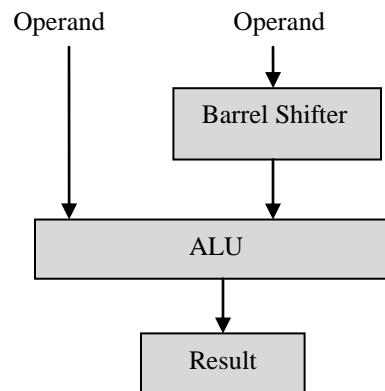
ARM supports the basic Boolean operations: and; inclusive or; exclusive or.

All can be conditionally executed and can optionally set the condition code by appending ‘s’ to the opcode.

Instr	Operands	Notes	
AND	rd, rn, rm	rd = rn AND rm	Boolean AND
AND	rd, rn, #n	rd = rn AND #n	Boolean AND
ORR	rd, rn, rm	rd = rn OR rm	Boolean INCLUSIVE OR
ORR	rd, rn, #n	rd = rn OR #n	Boolean INCLUSIVE OR
EOR	rd, rn, rm	rd = rn EOR rm	Boolean EXCLUSIVE OR
EOR	rd, rn, #n	rd = rn EOR #n	Boolean EXCLUSIVE OR

The shifts and rotates are a bit different. One of the operands that feed the ALU goes through a *barrel shifter* which is a digital circuit that can shift a data word by a specified number of bits in one clock cycle.

Thus instead of having special instructions for shifts and rotates, the system uses the mov instruction with source operand being shifted or rotated and then placed in destination operand.



Select the desired shift or rotate operation and use that as the operand in the MOV instruction.

	Notes
LSL	shift left, fill with 0, last bit shifted is in CF
LSR	shift right, fill with 0, last bit shifted is in CF
ASR	shift right, fill with sign bit, last bit shifted is in CF
ROR	rotate right, last bit shifted is in the CF
RRX	rotate right through the CF, the CF acts as the 33 bit

Replace xxx with desired shift or rotate

Instr	Operands	Notes
MOV	rd, rn, xxx #n	Perform the shift or rotate on rn (use n as the count) Place the result in rd
MOV	rd, rn, xxx rm	Perform the shift or rotate on rn (use rm as the count) Place the result in rd

Notes:

- The count can be 0 to 31.
- Coding *movs* will update the N, Z, C flags only.
- There is not an Arithmetic Shift Left. That is identical to a Logical Shift Left.
- There is not a rotate left. You can do a rol by calculating rol=ror32-m. However, when using ror to do a rol the CF will not contain the same bit as if a rol had actually been used.

Creating Immediate Values

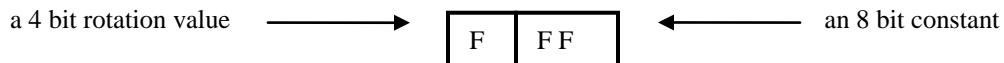
The immediate value #n in these instructions has a limited range: mov r0, #n add r0, r0, #n
 However, it is *not* simple bounds value. For example, here are some valid and invalid values:

Instruction	Valid?
add r0, r0, #1	valid
add r0, r0, #10	valid
add r0, r0, #100	valid
add r0, r0, #1000	valid
add r0, r0, #1024	valid
add r0, r0, #32768	valid

Instruction	Valid?
add r0, r0, #379	<i>invalid</i>
add r0, r0, #999	<i>invalid</i>
add r0, r0, #1001	<i>invalid</i>

So why is 1000 valid but 379 and 999 and 1001 invalid?

The immediate operand in a machine instruction consists of two parts:



The 32 bit immediate value is created by putting the 8 bit constant into a 32 bit field and rotating right that 32 bit field by $2 * \text{the rotation value}$ (e.g. 3 means rotate 6 bits)

Instruction	Operand	Rotation amount	Constant	<ul style="list-style-type: none"> • 32 bit initial constant created • below that is the rotated value
add r0, r0, #1	001	0	01	0000 0000 0000 0000 0000 0000 0000 0001
add r0, r0, #10	00A	0	0A	0000 0000 0000 0000 0000 0000 0000 1010
add r0, r0, #100	064	0	64	0000 0000 0000 0000 0000 0000 0000 0110 0100
add r0, r0, #1000	FFA	$F = 30_{10}$	FA	0000 0000 0000 0000 0000 0000 0000 1111 1010 0000 0000 0000 0000 0000 0000 0000 0011 1110 1000
add r0, r0, #1024	B01	$B = 22_{10}$	01	0000 0000 0000 0000 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0100 0000 0000
add r0, r0, #32768	902	$9 = 18_{10}$	02	0000 0000 0000 0000 0000 0000 0000 0010 0000 0000 0000 0000 0000 0000 0000 1000 0000 0000

If the assembler cannot generate the rotation value and 8 bit constant to create the desired immediate value, such as 999, then it will output an error message.

For putting an immediate value in a register, instead of using the *mov* instruction, it is recommended that you use the *ldr* instruction. The reason is that most assemblers treat the *ldr* instruction in a special way. They first try to create a rotation value and an 8 bit constant value. If successful, they generate the corresponding *mov* instruction. If unsuccessful, they create a 32 bit constant and put it into a *literal pool* in memory. They then generate the *ldr* instruction that loads the memory constant into the register using the PC register as the index register. Thus using *ldr* moves some of the burden from the programmer to the assembler.

The advantage to using this *shift* and *constant* process is that ARM can generate the most common constants (for example 0 to initialize a counter, 1 and 2 and 4 to advance pointers for byte, halfword and word lists) and you can generate large numbers, such as 32768, that may be used as mask fields in logical operations.

Subroutines And Stacks

ARM provides *multiple* options for creating and using stacks. The method presented implements standard stack operation as used by the x86, JVM and floating-point Numeric Data Processor. These are the needed instructions.

Instr	Operands	Notes
BL	label	Branch and Link <ul style="list-style-type: none"> Conditionally branches to the instruction at label (e.g. bleq ... branch and link if Z=1). Stores the address of the next instruction in the Link Register r14 to allow code to return from a subroutine.
STMDB	sp!, {regs,lr}	Store Multiple Decrement Before Use <ul style="list-style-type: none"> Stores registers in the brackets { ... } on the stack. The stack pointer is decremented before use to open slots on the stack. This is standard stack operation. The ! causes the stack pointer to be updated. The register list may contain multiple ranges. {r0-r3, r5, r7, r9-r11, lr} The last register specified is the Link Register which contains the return address.
LDMIA	sp!, {regs,pc}	Load Multiple Increment After Use <ul style="list-style-type: none"> Loads registers in the brackets { ... } from the stack. The stack pointer is incremented after use. This is standard stack operation. The ! causes the stack pointer to be updated. The last register specified is the Program Counter which is loaded with the last register stored which was the Link Register, thus returning the subroutine to the calling program.

We use the same example as we did with x86. A subroutine that calculates the square root of number passed in via r0. The resulting root is returned in r0. The subroutine uses r0, r1, r2, r3 and thus will save and restore r1, r2, r3.

```

main:
    ldr    r1, =n      ;r1 pts to n
    ldr    r0, [r1]    ;r0 = n
bl    sqrt        ;call sub
    ldr    r1, =r      ;r1 pts to r
    str    r0, [r1]    ;r = root

    .data
n: .word  625    ;n is a word
r: .word  0       ;root is a word

sqrt:
    stmdb sp!, {r1-r3,lr} ;save r1-r3 and
                           ;the link register

    ldr    r1, =27      ;change r1
    ldr    r2, =11      ;change r2
    ldr    r3, =100     ;change r3

    ldr    r0, =25      ;set root to be 25

    ldmia sp!, {r1-r3,pc} ;restore r1-r3 and
                           ;load the pc with the
                           ;original link reg

```

Summary of Instructions

This table shows which instructions can optionally set the condition code and which can be conditionally executed.

	Can optionally set the condition code by adding the suffix 's'	Can be conditionally executed
ADD / SUB	Yes	Yes
AND / ORR / EOR	Yes ... N, Z, C bits	Yes
BL (BRANCH & LINK)	No	Yes
BXX (BRANCH)	No	Yes
CMP	Always set the condition code	Yes
LDR / STR	No	Yes
MOV	Yes ... N, Z, C bits	Yes
STMDB / LDMIA	No	Yes
UMUL / SMULL	Yes ... N, Z, bits	Yes

Notes Related To The ARMSim Assembler And Simulator

These represent items encountered when coding for the ARMSim assembler and simulator. They may represent implementation decisions as opposed to ARM architectural decisions.

1. Both instruction labels and data declaration names are case sensitive.
2. The way to specify a hex value is: 0xFFFFFFFF
3. The load register instruction uses an = to identify addresses or immediate data to put in a register.

```
ldr r0, =val1           ;r0 = address of val1  
ldr r1, =1000           ;r1 = 1000 decimal  
ldr r2, =0xFFFFFFFF    ;r2 = FFFFFFFF hex
```

All other operations use a # to identify immediate data.

```
ldr r4, [r0], #4        ;post increment  
mov r5, #1000           ;mov  
add r6, r6, #1          ;add constants  
cmp r7, #0              ;compare  
cmp r8, #0xFFFFFFFF    ;compare
```

4. Note that all loads and stores of memory variables are indirect; they use a base register and second register or constant. Some assemblers will use the Program Counter, r15, as the base register and the offset of the variable into the program as the constant. However, as best I can tell, the ARMSim assembler and simulator do not support that and thus do not support directly loading a variable into a register. Instead you need to load the address of variable into a register then use that register as a pointer.

```
ldr r0, v1 ;r0=v1  
  
.data  
v1: .word 579 ;data item v1
```

```
ldr r0, =v1 ;r0=address of v1  
ldr r0, [r0] ;r0=v1  
  
.data  
v1: .word 579 ;data item v1
```

5. If an instruction can both optionally set the condition code and be conditionally executed then the mnemonic is: **opcode condition_code setting s ...** for example: addeqs

ARM Instruction Encoding

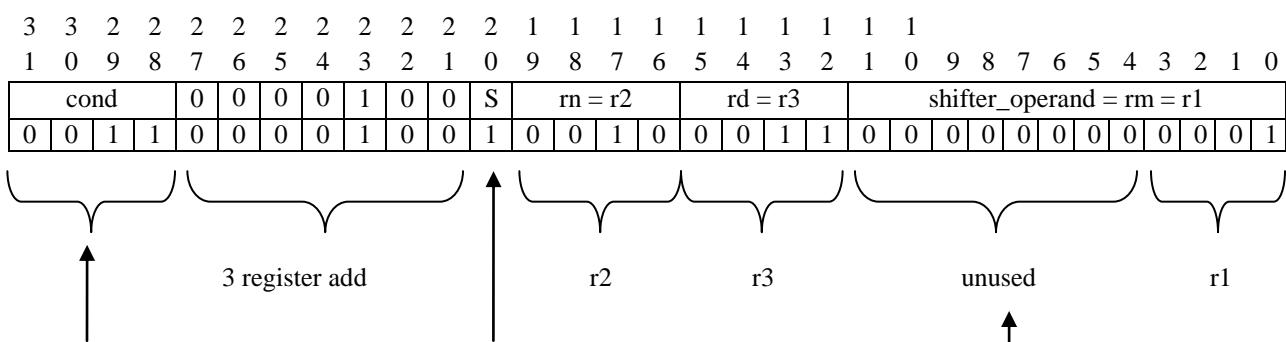
Those interested in the whole picture of encoding ARM instructions should obtain book or the ARM Architecture Reference Manual. In this section provide some examples to show some basic concepts.

Let us look at the encoding of this instruction: **addecs r3, r2, r1 ;r3 = r2 + r1**

This instruction adds r2 and r1 and stores the result in r3. It will only execute if the Carry Flag is cleared. When executed, it sets the Condition Code

ARM instructions are 32 bits. The hex value of this machine instruction is: **30 92 30 01**

This is the binary value of that machine instruction. We describe the fields left to right.



To execute an instruction conditionally, place the four bits from this table that match the desired condition into bits 28-31 of the instruction. For CC the bits are 0011

S = 1 means have the instruction sets the condition code

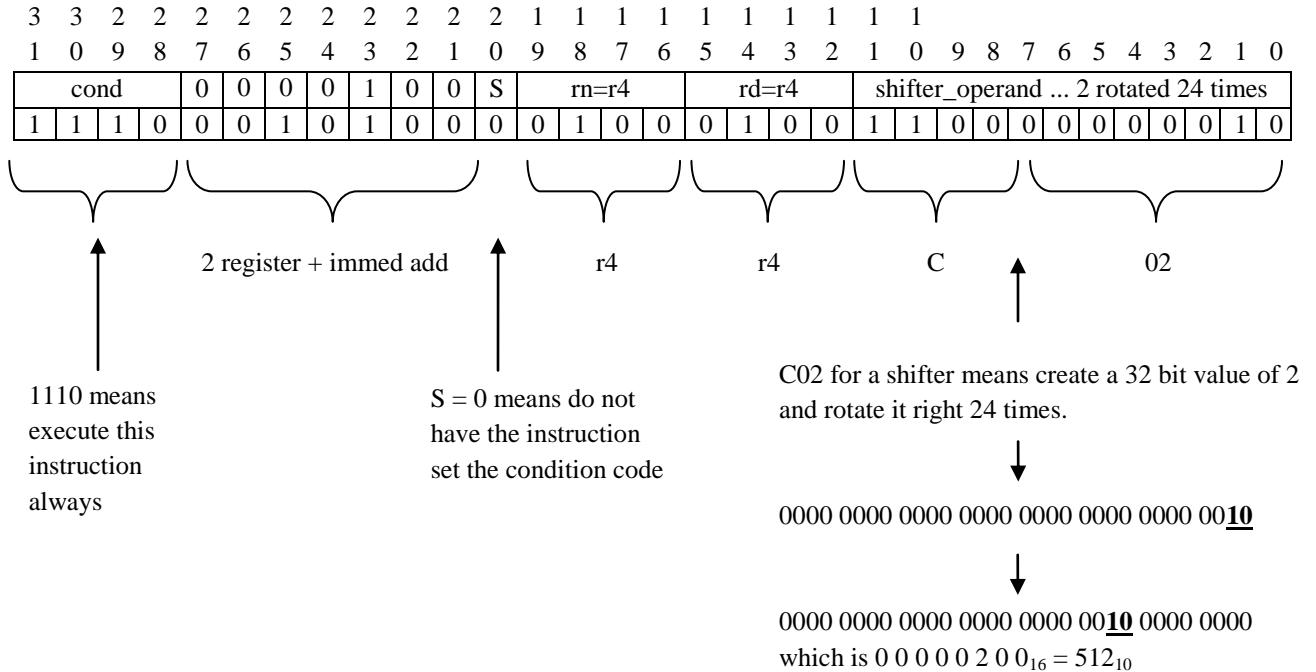
Note: For this instruction, there are 8 bits in the shifter_operand that are not used. If ARM used variable length instructions then this instruction could have been encoded in three bytes. However, RISC machines use fixed length instructions to simplify and speed up pipeline instruction fetching.

0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
1000	VC
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	Always

Let us change the add just a bit to be: `add r4, r4, #512 ;r4 = r4 + 512`

The hex value of the machine instruction is: **E2 84 4C 02**

This is the binary value of that machine instruction. We describe the fields left to right.

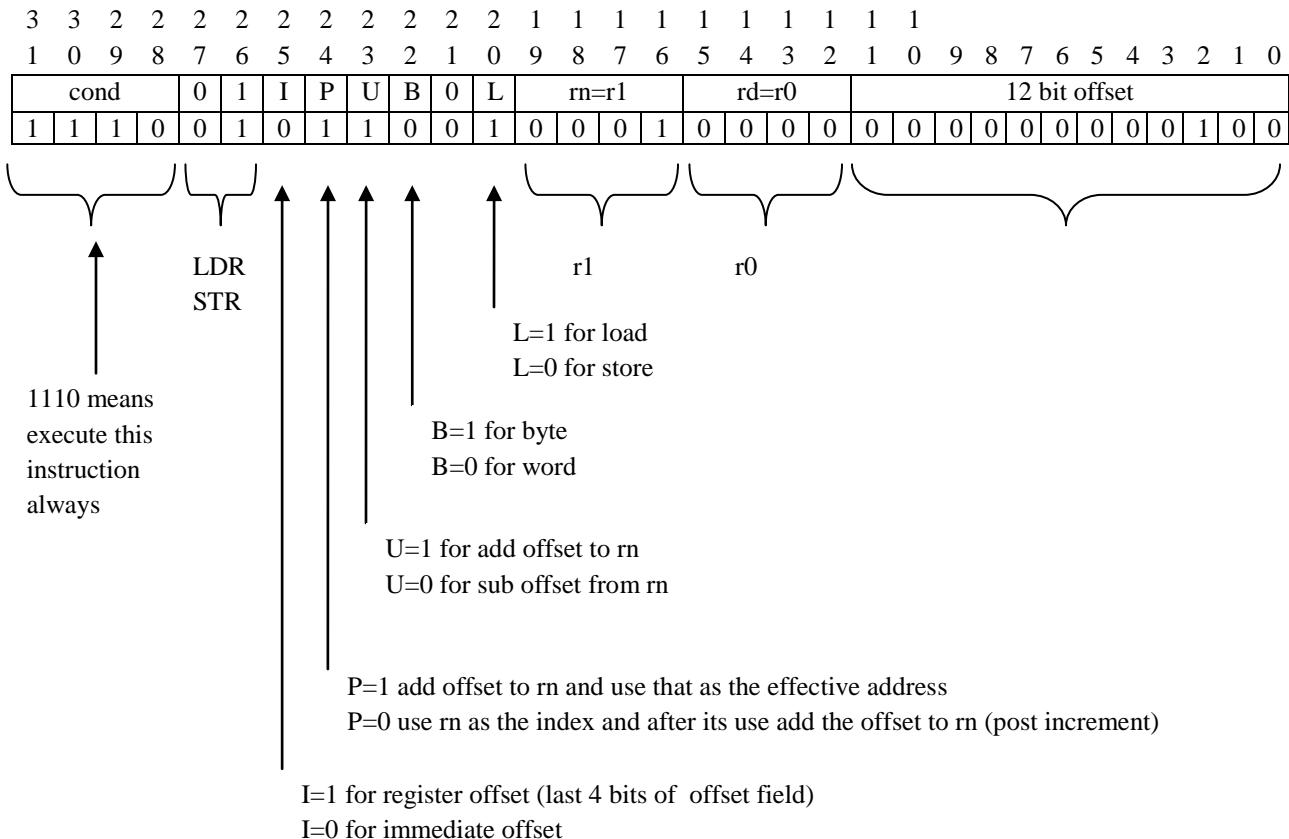


Let us look at loads and stores.

ldr r0, [r1, #4] ; r0 = the word of data pointed to by r1 + 4

The hex value of the machine instruction is: **E5 91 00 04**

This is the binary value of that machine instruction. We describe the fields left to right.



Variations:

ldr r0, [r1]	; r0 = the word of data pointed to by r1	E5 91 00 00	offset=0
ldr r0, [r1, r2]	; r0 = the word of data pointed to by r1 + r2	E7 91 00 02	I=1 (register offset) offset=2 for r2
ldrb r0, [r1, #4]	; r0 = the word of data pointed to by r1 + 4	E5 D1 00 04	B=1 for byte
lrd r0, [r1], #4	; r0 = the word of data pointed to by r1 then add 4 to r1 (post increment)	E4 91 00 04	P=0 for post increment

Note that this machine instruction only supports word or byte data and that data is unsigned.

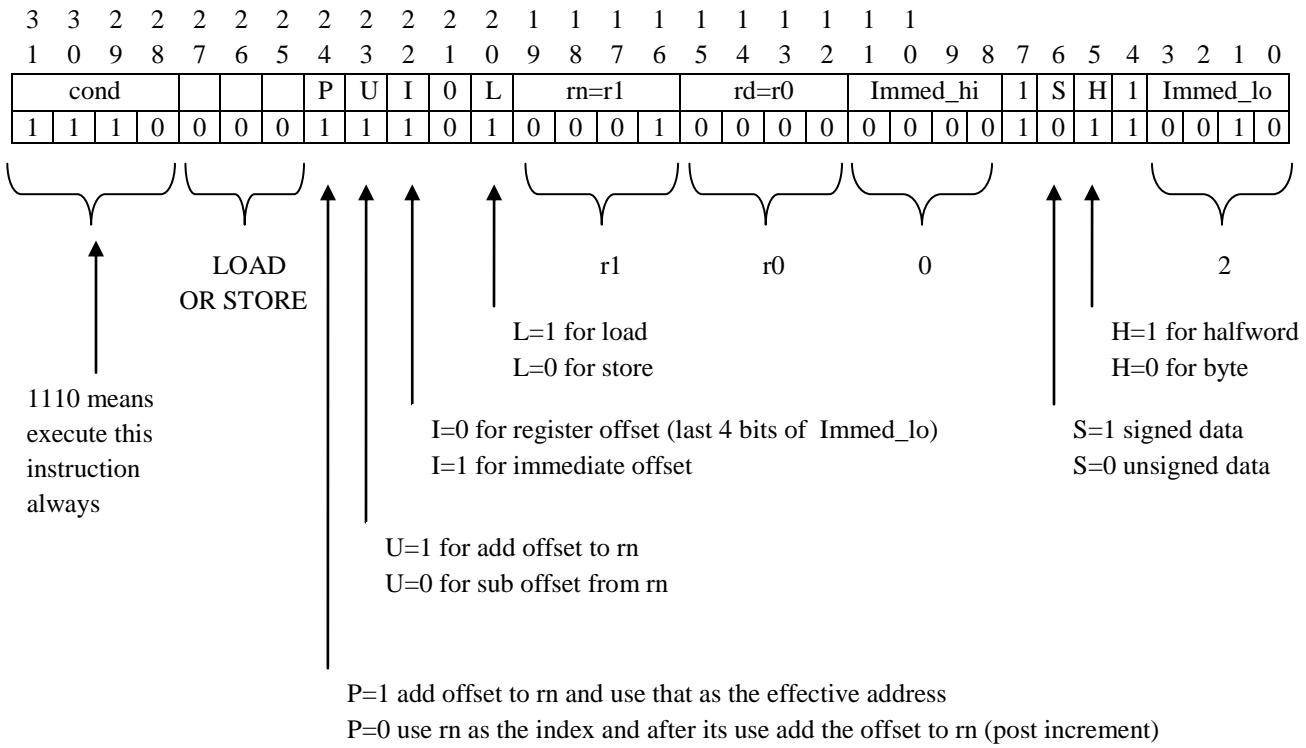
Thus a different machine instruction exists for load or store unsigned or signed halfwords and signed bytes.

This is the instruction for loading/storing unsigned or signed halfwords or bytes.

ldrh r0, [r1, #2] ; r0 = the halfword of data pointed to by r1 + 2

The hex value of the machine instruction is: **E1 D1 00 B2**

This is the binary value of that machine instruction. We describe the fields left to right.



Variations:

ldrsh r0, [r1], #2 : r0 = the signed halfword of data pointed to by r1 + 2 E1 D1 00 F2 S=1 for signed

ldr sb r0, [r1, #2] : r0 = the signed byte of data pointed to by r1 + 2 E5 D1 00 D0 S=1 Signed

$b = 1$ byte

Writing Assembler Code For The ARMSim Assembler And Simulator

ARMSim accepts ARM assembly source files that use the Gnu Assembler (GAS) syntax. ARM assembly source files can be created using any text editor and must be saved with a .s filename extension.

Basic files have this format which is *similar* to DOS MASM.

Two differences to watch for are instruction labels and data names are case sensitive and both end with a colon.

This is the format for data declarations. Note that data is declared after instructions. You can declare bytes, halfwords, words, in decimal or hex. You can declare lists in ASCII, decimal or hex.

```
val01: .skip    4          ;reserve 4 uninitialized bytes
val02: .word    1000       ;create a 32 bit word
val03: .word    -1000      ;create a 32 bit word
val04: .word    0x000003e8  ;create a 32 bit word ... specifying hex
val05: .hword   555        ;create a 16 bit half word
val06: .hword   -1234      ;create a 16 bit half word
val07: .byte    10         ;create an 8 bit byte
val08: .byte    -100        ;create an 8 bit byte
val09: .asciz   "ABC"      ;create an ASCII string terminated with 00h
val10: .ascii   "DEF"      ;create an ASCII sting without a terminating 00h
val11: .byte    0xFF, 0, 1   ;create a list of bytes
val12: .word    1, -2, 3, -4 ;create a list of word values
```

In addition to the normal ARM instructions, ARMSim supports supervisor calls to read and write files. These are the calls that we will use in CSC236. In our x86 programming we read the Standard Input and wrote the Standard Output. Those were normally the keyboard and display, but could be redirected from/to files. For our ARM programming we do something a bit more sophisticated and read and write actual files. This means we will need to open the files before use and close the files after use.

To use a file, it must be assigned a *handle*. A handle is a number that the operating system assigns temporarily to a file when it is opened. The operating system uses the file handle internally when accessing the file.

Opcode	Description	Inputs	Outputs	EQU
swi 0x11	Halt program			SWI_Exit
swi 0x66	Open a file	r0 = the address of a null terminated ASCII string with file name r1 = 0 for input 1 for output	r0 = <i>file handle</i> that will specify the file for read, write, and close	SWI_Open
swi 0x68	Close a file	r0 = file handle		SWI_Close
swi 0x69	Write string to a file	r0 = file handle r1 = the address of a null terminated ASCII string		SWI_PrStr
swi 0x6a	Read string from a file	r0 = file handle r1 = the address of a string in which to read the data r2 = maximum number of bytes to store	r0 = number of bytes stored See Notes below	SWI_RdStr
		Notes on reading a string: <ul style="list-style-type: none"> • a count of 0 means <i>end of file</i> • any <i>end of line</i> sequence (LF or CR/LF) in the file will be replaced with single hex 00 • a hex 1A will be read as a regular character 		

As part of the UNPACK.EXE file in ARM locker are three sample programs that show how ARM instructions work as well as how the Software Interrupts above work.

- hello.s Writes *hello world*. The code for this program is also on the next page.
- copystr.s Copies one line from an input file to an output file, one byte at a time.
This code would be a good starting model for the KEY program written in ARM Assembler.
- copyfile.s This code copies an input file to an output file, one line at a time.

This code opens a file named *HELLO.OUT* and writes *hello world* to that file.

```
;-----
; Software Interrupt equates
;-----
    .equ SWI_Open, 0x66      ;Open a file
    .equ SWI_Close, 0x68     ;Close a file
    .equ SWI_PrStr, 0x69     ;Write a null-ending string
    .equ SWI_RdStr, 0x6a     ;Read a string and terminate with null char
    .equ SWI_Exit, 0x11      ;Stop execution
;-----

    .global _start           ;start will be known to external programs
    .text                      ;start instructions

;-----
; open output file
; - r0 points to the file name
; - r1 1 for output
;-----
_start:                   ;
    ldr r0, =OutFileName   ;r0 points to the file name
    ldr r1, =1              ;r1 = 1 specifies the file is output
    swi SWI_Open            ;open the file ... r0 will be the file handle
    ldr r1, =OutFileHandle  ;r1 points to handle location
    str r0, [r1]             ;store the file handle
;-----

;-----
; Write the outputs string
;-----
    ldr r0, =OutFileHandle ;r0 points to the output file handle
    ldr r0, [r0]             ;r0 has the output file handle
    ldr r1, =OutString       ;r1 points to the output string
    swi SWI_PrStr           ;write the null terminated string
;-----


;-----
; Close output file and terminate the program
;-----
_exit:                   ;
    ldr r0, =OutFileHandle ;r0 points to the output file handle
    ldr r0, [r0]             ;r0 has the output file handle
    swi SWI_Close            ;close the file
    ;
    swi SWI_Exit             ;terminate the program
;-----


    .data                  ; start data
;-----
OutFileHandle: .skip 4          ;4 byte field to hold the output file handle
OutString: .asciz "hello world" ;the output string
OutFileName: .asciz "HELLO.OUT"  ;Output file name, null terminated
;-----
.end
```