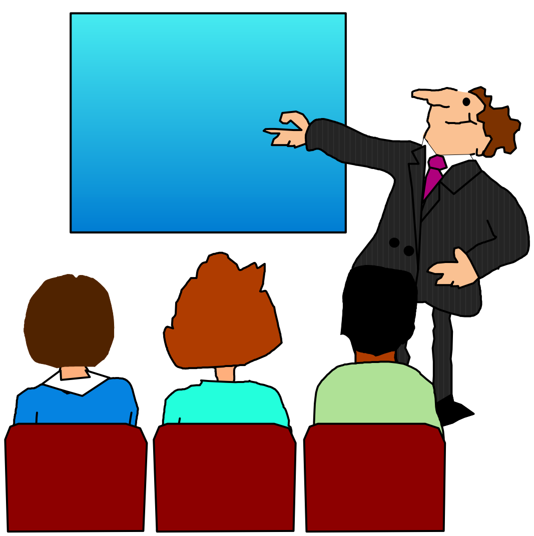
Assembler Language "Boot Camp" Part 1 - Numbers and Basic Arithmetic

SHARE in San Francisco

August 18 - 23, 2002

Session 8181



Introduction

- Who are we?
  - John Dravnieks, IBM Australia
  - John Ehrman, IBM Silicon Valley Lab
  - Michael Stack, Department of Computer Science, Northern Illinois University

Introduction

- Who are you?
  - An applications programmer who needs to write something in S/390 assembler?
  - An applications programmer who wants to understand S/390 architecture so as to better understand how HLL programs work?
  - A manager who needs to have a general understanding of assembler?

- Our goal is to provide for professionals an introduction to the S/390 assembly language

Introduction

- These sessions are based on notes from a course in assembler language at Northern Illinois University

- The notes are in turn based on the textbook, Assembler Language with ASSIST and ASSIST/I by Ross A Overbeek and W E Singletary, Fourth Edition, published by Macmillan

Introduction

- The original ASSIST (Assembler System for Student Instruction and Systems Teaching) was written by John Mashey at Penn State University

- ASSIST/I, the PC version of ASSIST, was written by Bob Baker, Terry Disz and John McCharen at Northern Illinois University

Introduction

- Both ASSIST and ASSIST/I are in the public domain, and are compatible with the System/370 architecture of about 1975 (fine for beginners)

- Both ASSIST and ASSIST/I are available at http://www.cs.niu.edu/~mstack/assist

Introduction

- Other materials described in these sessions can be found at the same site, at http://www.cs.niu.edu/~mstack/share

- Please keep in mind that ASSIST and ASSIST/I are not supported by Penn State, NIU, or any of us

Introduction

- Other references used in the course at NIU:
  - Principles of Operation
  - System/370 Reference Summary
  - High Level Assembler Language Reference
- Access to PoO and HLASM Ref is normally online at the IBM publications web site
- Students use the S/370 "green card" booklet all the time, including during examinations (SA22-7209)

Our Agenda for the Week

- Session 8181: Numbers and Basic Arithmetic
- Session 8182: Instructions and Addressing
- Session 8183: Assembly and Execution; Branching

Our Agenda for the Week

- Session 8184: Arithmetic; Program Structures
- Session 8185: Decimal and Logical Instructions
- Session 8186: Assembler Lab Using ASSIST/I

Today's Agenda

- Decimal, Binary and Hexadecimal Numbers and Conversions
- Main Storage Organization and Signed Binary Numbers
- Integer Arithmetic and Overflow
- Getting Started with ASSIST/I

Decimal, Binary and Hexadecimal Numbers and Conversions

In Which We Learn to Count All Over Again

Counting in Bases 10, 2, and 16

Dec	Bin	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dec	Bin	Hex
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

Numbers in Different Bases

- Consider how we write numbers in base 10, using the digits 0 - 9:
  - $832_{10} = 800_{10} + 30_{10} + 2_{10}$
  - $= 8 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$

- For numbers in base 2 we need only 0 and 1:
  - $1101_2 = 1000_2 + 100_2 + 00 + 1$
  - $= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

- But because it requires less writing, we <u>usually</u> prefer base 16 to base 2

Caution!

- The value of a number may be ambiguous when the base isn't indicated

- $1011 = ?_{10}$

- $1011_2 = 11_{10}$

- $1011_{16} = 4113_{10}$

- The base will usually be clear from the context, but will otherwise be provided

Converting Binary & Hexadecimal to Decimal

$$
\begin{aligned}
1011_2 &= 1 \times 2^3 = 1 \times 8 = 8 \\
&+ 0 \times 2^2 = 0 \times 4 = 0 \\
&+ 1 \times 2^1 = 1 \times 2 = 2 \\
&+ 1 \times 2^0 = 1 \times 1 = \underline{1} \\
&\qquad 11
\end{aligned}
$$

$$
\begin{aligned}
A61_{16} &= 10 \times 16^2 = 10 \times 256 = 2560 \\
&+ 6 \times 16^1 = 6 \times 16 = 96 \\
&+ 1 \times 16^0 = 1 \times 1 = \underline{1} \\
&\qquad 2657
\end{aligned}
$$

Note: numbers without subscript are base 10

Converting Decimal to Binary & Hexadecimal

- To convert a decimal number n to base b
  1. Divide n by b, giving quotient q and remainder r
  2. Write r as the rightmost digit, or as the digit to the left of the last one written
  3. If q is zero, stop; otherwise set n = q and go back to Step 1.

- Note that each digit will be in the range 0 to b - 1

Example: Convert 123_{10} to Base 16

- 123 / 16 = 7 with remainder 11, so the rightmost digit is B
- 7 / 16 = 0 with remainder 7, so the next digit to the left is 7
- Since quotient is 0, stop
- Result is $123_{10} = 7B_{16}$
- A similar process shows $123_{10} = 1111011_2$

Conversions Between Bin and Hex

- These are the easiest of the conversions, since $16 = 2^4$ and we can convert by groups of digits

- To convert from binary to hexadecimal
  1. Starting at the right, separate the digits into groups of four, adding any needed zeros to the left of the leftmost digit so that all groups have four digits
  2. Convert each group of four binary digits to a hexadecimal digit

Conversions Between Bin and Hex

- So to convert 101101 to hex,
    1. Group the digits and add zeros: 0010 1101
    2. Convert to hex digits: 2 D

- To convert from hexadecimal to binary, simply reverse the algorithm

- So $2C5_{16}$ = 0010 1100 0101 = 1011000101_2

Arithmetic with Unsigned Numbers

- Addition and subtraction of unsigned numbers is performed in hexadecimal and binary just the same as it is in decimal, with carries and borrows

- We normally use <u>signed</u> numbers, so we won't dwell on unsigned numbers

Arithmetic with Unsigned Numbers

```
 1101  <--- carries   11110  <--- carries
  FCDE                  10110
+ 9A05                +  1011
 196E3                100001

   BD+c  <--- borrows    0110+c  <--- borrows
  FCDE                 111000
 -9AE5                -  10011
  61F9                 100101
```

Main Storage Organization and Signed Binary Numbers

Main Storage Organization

- In order to understand how signed numbers are represented in a binary computer, we need to understand memory organization

- Abstractly, a binary digit (or bit) can be represented by any 2-state system: on-off, true-false, etc.

- A computer's memory is simply a collection of millions of such systems implemented using electronic switches

Main Storage Organization

- Memory is organized by grouping eight bits into a byte, then assigning each byte its own identifying number, or address, starting with zero

- Bytes are then aggregated into words (4 bytes), halfwords (2 bytes) and doublewords (8 bytes)
  - One byte = eight bits
  - One word = four bytes = 32 bits

Main Storage Organization

- Typically, each of these aggregates is aligned on an address boundary which is evenly divisible by its size in bytes
- So, a fullword (32 bits) is aligned on a 4-byte boundary (addresses 0, 4, 8, 12, 16, 20, etc.)
- Remember, memory addresses refer to bytes, not bits or words

Representation of Signed Binary Integers

- Representing unsigned binary integers was fairly simple, but how can we include a sign?

- There are three ways we might represent signed integers, using a single bit as the sign (customarily the leftmost bit)
  - Signed-magnitude
  - Ones' complement
  - Two's complement

Representation of Signed Binary Integers

- Signed-magnitude is the most familiar (**+**17, **-**391) and we will see later how this is used in S/390
  - Allocating an extra bit for the sign, since $9_{10} = 1001_2$, we would write **+**9 as 0 1001_2 and **-**9 as 1 1001_2

Representation of Signed Binary Integers

- The ones' complement of a number is found by replacing each 1 with 0 and each 0 with 1
  - If we use one bit for the sign, then since 9_{10} is 1001_2, we would write +9 as 0 1001_2 and -9 as 1 0110_2

Representation of Signed Binary Integers

- The two's complement representation is formed by taking the ones' complement and adding 1
  - In this notation, again using one bit for the sign, we write **+**9 as 0 1001_2 and **-**9 as 1 0111_2

Representation of Signed Binary Integers

- In the S/390, a negative binary integer is represented by the two's complement of its positive value
  - Note that zero is its own complement in this representation (no +0 or -0), since:

Zero	=	00000000 00000000 00000000 00000000
1s Compl	=	11111111 11111111 11111111 11111111
Plus 1	=	1
Result	=	00000000 00000000 00000000 00000000

Representation of Signed Binary Integers

- In S/390, integers are represented in a 32-bit fullword, using the first bit as the sign

- A fullword can contain non-negative integers in the range 0 to $2^{31}-1$ (with sign bit = 0)

- A negative integer in the range $-2^{31}+1$ to -1 (with sign bit = 1) is formed by taking the two's complement of its absolute value

Representation of Signed Binary Integers: Examples

- -2^{31} is represented by 1000...000 but this number is not the two's complement of any positive integer

- In two's complement representation
  - +1 = 00000000 00000000 00000000 00000001
  - -1 = 11111111 11111111 11111111 11111111

- Or, in the more commonly used hexadecimal
  - +1 = **00000001**
  - -1 = **FFFFFFFF**

Integer Arithmetic and Overflow

Arithmetic with Signed Numbers

- Let's look at examples of addition and subtraction using signed numbers in two's complement. These examples use only 4 bits, not 32, with the leftmost bit as sign.

```
+3 = 0 011
+2 = 0 010
----   -----
+5   0 101

+3 = 0 011
-2 = 1 110     (Two's complement of 0 010)
----   -----
+1   0 001     (The carry out is ignored)
```

Arithmetic with Signed Numbers

- Now, how about **-**3 plus **+**2

```
 -3 = 1 101
 +2 = 0 010
 --   -----
 -1   1 111
```

Arithmetic with Signed Numbers

- Notice that the sign is correct each time, and the result is in two's complement notation

- Also, subtraction is performed by adding the two's complement of the subtrahend to the minuend. So **+3 - +2 = +3 + (-2).**

```
    +3        +3 <- minuend
  - +2      + -2 <- subtrahend
  ----      ----
    +1        +1
```

Arithmetic with Signed Numbers

- Computer arithmetic using 32-bit fullwords is a bit more complex, and is always shown in hex. Also, we will no longer display a separate sign bit (it will be part of the leftmost hex digit):

```
 00000011        AE223464
+0000010B       +5FCA5243
 --------        --------
 0000011C        0DEC86A7
```

Arithmetic with Signed Numbers

- Subtraction is performed by adding the two's complement
- Carry bits are ignored (results are correct anyway)

```
 F89ABCDE         F89ABCDE
-6D4AFBC0   =    +92B50440
                  --------
                  8B4FC11E
```

Overflow

- What if two large numbers are added and the result is greater than $2^{31} - 1$ (or less than -2^{31})?
- And how can we tell if this happened?
- In order to understand this, we will again demonstrate with our very small "words" of four bits, the first of which is the sign
- These "4-bit words" can handle integers in the range from -8 to $+7$ (1 000 to 0 111)

Overflow

- Now let's see what happens when we try to add **+**5 to **+**4 (we'll do this in binary, using our four-bit words).

- Overflow will occur since the result is greater than **+**7.

Overflow

- This is detected by checking the carry <u>into</u> the sign position and the carry <u>out of</u> the sign position

- If they are not equal, overflow occurred and the result is invalid.

Overflow

```
Out  In  [not equal, so overflow occurred]
   \  /
    01 00   <--   carries
     0 101  =  +5
     0 100  =  +4
     -----
     1 001  =  invalid (due to overflow)
```

The program may or may not take action on overflow, but it normally should since the result is invalid

Overflow

- But be very careful! The S/390 is a binary computer, not hexadecimal, so the check for overflow must be done using the binary representation - that is, we must look at bits, not hex digits

- So, if we add as follows...

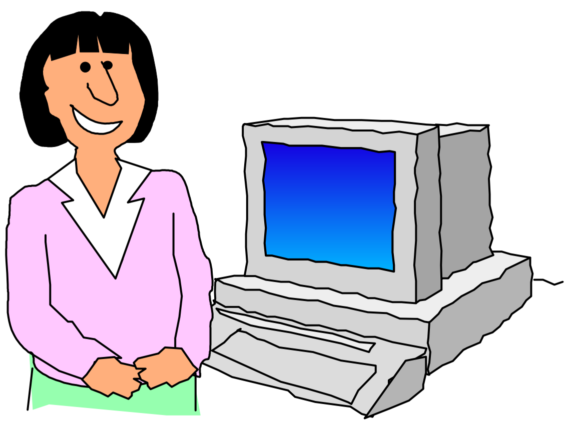
```
                          1111...
 D13BCF24        D =  1101...
+F3C12B97        F =  1111...
                      1100...
```

Overflow

- ... we can see that overflow does not occur (1 in and 1 out)
- But if we make the mistake of checking the hex digits, we see what looks like overflow

```
 10
  D1...
 +F3...
```

Getting Started With ASSIST/I



ASSIST/I Features

- ASSIST/I is an integrated assembler and instruction interpreter, plus a text editor and interactive debugger
- There are built-in functions (X-instructions) for I/O and data conversion
- Program tracing lets you watch "everything" happen

ASSIST/I Features

- It is a useful tool for getting started and "tinkering" on a PC without needing any host-system access
- A User Guide is included in the "Starter Kit" handout
- And it's free!

ASSIST/I Limitations

- ASSIST/I supports only an older, less-rich instruction set
- Modern assembler features are missing
- Programming style may be less robust than desired

ASSIST/I Limitations

- Text editor functions are rather awkward
  - It may be easier to use a simple PC editor
- System macros aren't available

Getting Started with ASSIST/I

- Easiest: run everything from the diskette
  - Change your disk drive to A: and your working directory to \BootAsst\
  - Enter CAS, and follow the prompts to run program DEMOA.ASM
  - We'll step through its execution and show how to create a .PRT file

- Try some of the other DEMO programs