Assembler Language
"Boot Camp"
Part 4 - Program Structures; Arithmetic
SHARE 117 in Orlando
Session 09213
August 10, 2011
Introduction

Who are we?

- John Ehrman, IBM Software Group
- Dan Greiner, IBM Systems & Technology Group
Introduction

Who are you?

- An applications programmer who needs to write something in mainframe assembler?
- An applications programmer who wants to understand z/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 z/Architecture assembler language
Introduction

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

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)

- Everything we discuss here works the same in z/Architecture

- Both ASSIST and ASSIST/I are available at http://www.kcats.org/assist
Introduction

- ASSIST-V is also available now, at http://www.kcats.org/assist-v

- Other materials described in these sessions can be found at the same site, at http://www.kcats.org/share

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

Other references used in the course at NIU:
- Principles of Operation (PoO)
- 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

- Assembler Boot Camp (ABC) Part 1: Numbers and Basic Arithmetic (Monday - 9:30 a.m.)

- ABC Part 2: Instructions and Addressing (Monday - 1:30 p.m.)

- ABC Part 3: Assembly and Execution; Branching (Tuesday - 9:30 a.m.)

- ABC Lab 1: Hands-On Assembler Lab Using ASSIST/I (Tuesday - 6:00 p.m.)
Our Agenda for the Week

- ABC Part 4: Program Structures; Arithmetic (Wednesday - 9:30 a.m.)
- ABC Lab 2: Hands-On Assembler Lab Using ASSIST/I (Wednesday - 6:00 p.m.)
- ABC Part 5: Decimal and Logical Instructions (Thursday - 9:30 a.m.)
Agenda for this Session

- EQUate and Extended Branch Mnemonics
- Literals, LOAD ADDRESS, and Looping
- Internal Subroutines
- The MULTIPLY and DIVIDE Instructions
Register EQUates & Extended BRANCH Mnemonics

In Which We Find More Than One Way to Say the Same Thing
Register EQUates

- It is possible to define symbols using the EQU instruction

```
label EQU expression
```

- Then, when the assembler encounters `label` elsewhere, it will substitute the value of `expression`

- We will use `expression` only as integer values, but it can be written in other ways
Register EQUates

- EQU lets us define symbolic register names
  - R0   EQU   0
  - R1   EQU   1
  - ...  
  - R15  EQU   15

- Many programmers use these to cause register references to appear in the symbol cross-reference listing (although HLASM has an option to produce a much better "register cross-reference" listing)
Register EQUates

Be careful how you think about the symbols, though - all the assembler does is substitute values.

For example, assuming that register equates are available, consider the object code for:

- \( \text{L R3,R4 (!)} \quad (58300004) \)
- \( \text{L R3,R4 (R5,R6)} \quad (58356004) \)

Because they can be confusing to learners, it may be best to not use them.
Extended BRANCH Mnemonics

The BRANCH ON CONDITION instructions (BC, BCR) require a branch mask

We have so far given the mask as B'xxxx'

There are, however, special mnemonics which incorporate the mask into the "op code"
Extended BRANCH Mnemonics

- So, for example, after a compare instruction, the extended mnemonic
  \[ \text{BE} \ \text{addr} \ (\text{Branch on Equal}) \]
  can be used in place of
  \[ \text{BC} \ \text{B}'1000',\text{addr} \]

- The \text{BE} extended mnemonic, for example, can be thought of as opcode 478

- Most of the extended mnemonics can be found in the "green card" on p. 21-22
Literals, LOAD ADDRESS, and Looping

In Which We Face a Most Difficult But Very Important Concept: Addresses
Recall that the DC instruction defines an area of storage within a program, with an initial value.

Since that value is only initial, it can easily be changed (and very often is).

For example, a counter to be incremented may be initially defined as:

```
COUNT    DC    F'0'
```
There is also a need for a "constant," a value in storage which is intended to retain that value.

```plaintext
A 4,ONE
...
ONE DC F'1'
```

We can instead code the constant as part of the instruction, in place of the usual second operand memory address:

```plaintext
A 4,=F'1'
```
In this case, the second operand is coded as a literal, which is indicated by the preceding equal sign.

This is also good documentation, as the value is seen immediately, rather than after searching the program listing for a data area.

But where will the storage for this literal be? With a DC (or DS) statement, the location is exactly where the DC or DS occurs.
A literal, on the other hand, will be located in a "pool" of literals whose location is defined by using the LTORG (LiTeral pool ORiGin) instruction. As many LTORGs as needed may be used, and each creates a pool for all previous "unpooled" literals. This means the same literal (e.g., \( \texttt{F'1'} \)) may appear in multiple pools.
The LOAD ADDRESS Instruction

- This is very simply stated:
  \[ \text{label} \quad \text{LA} \quad R_1, D_2(X_2, B_2) \]
  Replaces the contents of register \( R_1 \) with the effective address of the second operand, \( D_2(X_2, B_2) \)

- Here may be a help to understanding:
  \[ \text{L} \quad 5, \text{WORD} \]
  is the same as the pair:

  \[ \text{LA} \quad 5, \text{WORD} \]
  followed by

  \[ \text{L} \quad 5, 0(,5) \]
The LOAD ADDRESS Instruction

- Sometimes only $d_2$ is specified - that is, $x_2$ and $b_2$ are zero:

  \[ \text{LA} \ 5, 4 \ (0 \leq d_2 \leq 4095) \]

  Note: this is the same as $\text{LA} \ 5, 4 \ (0, 0)$

- This is a common method of placing a small number in a register without accessing memory
The LOAD ADDRESS Instruction

- LA may also be used to increment (but not decrement) a non-negative value in a register

  \textbf{LA} \ 6,1 (,6) \quad \text{Increase } c(\text{R6}) \text{ by } 1

- N.B. The high order bit or byte of R6 is set to zero, depending on something called \textit{addressing mode} (beyond our scope)
The following program builds a table of fullwords and then exits the completed program. It reads one number off each input card, recognizing the end of input when a trailer card containing 999999 is encountered. The logic of the program is:

* Step 1. Initialize R3 to point to the first entry in the table. R4, which is used to count the entries in the table, is initially set to 0.

* Step 2. Read the first card.

* Step 3. Check for the trailer. If it is the trailer go to Step 6 to exit the program.

* Step 4. Put the number into the table (adding 1 to the count of entries and incrementing the pointer to the next entry).

* Step 5. Read the next card and go back to Step 3.

* Step 6. Exit the program.
***<Step 1> Initialize counter and pointer to next entry
* 
SR 4,4 Set count of entries to 0
LA 3, TABLE Point at first entry (next one to fill)
* 
***<Step 2> Read the first card
* 
XREAD CARD, 80 It is assumed that there is a card and that it contains a number
*  
XDECI 2, CARD Convert input number to binary in R2
* 

Demo Program to Build a Table - 2

TABUILD CSECT
  USING TABUILD, 15

***<Step 1> Initialize counter and pointer to next entry
* 
SR 4,4 Set count of entries to 0
LA 3, TABLE Point at first entry (next one to fill)
* 
***<Step 2> Read the first card
* 
XREAD CARD, 80 It is assumed that there is a card and that it contains a number
*  
XDECI 2, CARD Convert input number to binary in R2
*
**Demo Program to Build a Table - 3**

***<Step 3>    Check for trailer
**
TRAILCHK C 2,TRAILER  Check for trailer 999999
    BE   ENDINPUT
**
***<Step 4>    Add the number to the table
**
    ST  2,0(,3)  Put number into current slot in the table
    LA  4,1(,4)  Add 1 to count of entries in the table
    LA  3,4(,3)  Move entry pointer forward 1 entry
**
***<Step 5>    Read the next card and get the number into R2
**
    XREAD CARD,80
    XDECI 2,CARD         The next number is now in R2
    B    TRAILCHK
**
***<Step 6>    Return to the caller
**
ENDINPUT BR 14         Exit from the program
**Demo Program to Build a Table - 4**

<table>
<thead>
<tr>
<th>LTORG</th>
<th>CARD</th>
<th>DS</th>
<th>CL80</th>
<th>Card input area</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE</td>
<td>DS</td>
<td>50F</td>
<td></td>
<td>Room for 50 entries</td>
</tr>
<tr>
<td>TRAILER</td>
<td>DC</td>
<td>F'999999'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>TABUILD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$ENTRY
123
456
789
234
567
890
345
999999
Looping Using BCT and BCTR

The loop we saw in the demo is controlled by the number of records in the input file.

Sometimes, a loop is to be executed n times:

1. Set $I$ equal to $n$
2. Execute the body of the loop
3. Set $I$ to $I-1$
4. If $I \neq 0$, go back to 2
5. Otherwise, continue (fall through)

This loop is always executed at least once.
Looping Using BCT and BCTR

There is a single instruction which implements this loop control - BRANCH ON COUNT

\[
\text{label} \quad \text{BCTR} \quad R_1, R_2 \\
\text{label} \quad \text{BCT} \quad R_1, D_2(X_2, B_2)
\]

The logic is

1. Decrement $R_1$ by one
2. If $c(R_1) \neq 0$, branch; otherwise, continue
Looping Using BCT and BCTR

In BCTR, if \( R_2 = 0 \), no branch is taken, although the \( R_1 \) register is still decremented.

```
LOOP  LA   12,200
      ...  
      BCT  12,LOOP  How many times?
************************
      LA   10,LOOP  
      LA   11,413
LOOP  ...  
      BCTR 11,10  How many times?
************************
      LA   0,LOOP  
      LA   1,10
LOOP  ...  
      BCTR 1,0   How many times?
```
Internal Subroutines

In Which We Show That You Can Go Home Again, and How
The Program Status Word (PSW)

- The PSW is an eight-byte aggregation of a number of important pieces of information, including:
  - The address of the next instruction
  - The Interruption Code
  - The Condition Code (CC)
  - The Program Mask
  - The Instruction Length Code (ILC) (in ASSIST only, not in z-Architecture)
The Program Status Word (PSW)

N.B. - The "basic" PSW format used in ASSIST dates to the 1960s and is not current; even so, it does have some fields which will help us.

The PSW fields in ASSIST that we want are:
- Bits 16-31: Interruption Code
- Bits 32-33: Instruction Length Code
- Bits 34-35: Condition Code
- Bits 36-39: Program Mask
- Bits 40-63: Next Instruction Address
The Program Status Word (PSW)

The Instruction Length Code (ILC) has the following meaning for its four possibilities:

<table>
<thead>
<tr>
<th>ILC (Dec)</th>
<th>ILC (Bin)</th>
<th>Instr types</th>
<th>Op Code Bits 0-1</th>
<th>Instr Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td></td>
<td></td>
<td>Not Available</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RR</td>
<td>00</td>
<td>One halfword</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>RX, RS, SI</td>
<td>01</td>
<td>Two halfwords</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>RX, RS, SI</td>
<td>10</td>
<td>Two halfwords</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>SS</td>
<td>11</td>
<td>Three halfwords</td>
</tr>
</tbody>
</table>
The Program Status Word (PSW)

- The Instruction Length Code can be used to determine the address of the current instruction
  1. Multiply by two to get the number of bytes in the current instruction
  2. Subtract it from the address of the next instruction

- This is very important in analyzing a dump from a program problem
There is a very important instruction which is used to control access to subroutines, BRANCH AND LINK

The RR and RX formats are

- `label  BALR  R₁, R₂`
- `label  BAL  R₁, D₂(X₂, B₂)`
Their operation is simple

1. Copy the 2nd word of the PSW to register \( R_1 \)
2. Branch to the address given by the second operand

Step 1 of the operation of BAL/BALR copies to register \( R_1 \) the address of the next instruction (this is very important)

- If in 24-bit addressing mode, the ILC, CC, and Pgm Mask are also copied
This operation means that, if we want to execute a subroutine called SORT, we can
1. Use `BAL 14,SORT` in the main routine, to place in R14 the address of the instruction following the BAL, then branch to SORT
2. Use `BR 14` at the end of the SORT routine to return and resume the main routine

The RR form, BALR, is very common, especially `BALR 14,15` for "external" subroutines
A special use of BALR occurs when $R_2 = 0$; then no branch occurs after placing the address of the next instruction in $R_1$:

```
BALR  12,0
USING NEWBASE,12
NEWBASE  ...  (next instruction)
```

This can be used to establish a base register when the current location is unknown.
The STM and LM Instructions

Having subroutines is all very nice, but with a limited number of registers, it is useful for subroutines to save registers at entry, then restore them at subroutine exit.

A third instruction format, RS, is used. Our first RS instruction is STORE MULTIPLE

```
label STM R1, R3, D2 (B2)
```

Copies the contents of the range of registers R1 through R3 to consecutive words of memory beginning at D2 (B2)
The STM and LM Instructions

Thus, **STM 4,8,SAVE** would copy the contents of R4, R5, R6, R7, and R8 to the five fullwords beginning at location SAVE.

And **STM 14,1,SAVE** would copy R14, R15, R0, and R1 to four consecutive fullwords at location SAVE (note the register-number wrap-around).

\[h_{OP}h_{OP}h_{R1}h_{R3}h_{B2}h_{D2}h_{D2}h_{D2}\] is the encoded form of an RS instruction.
The inverse operation is LOAD MULTIPLE

\[ \text{label} \quad \text{LM} \quad R_1, R_3, D_2(B_2) \]

Copies the contents of the consecutive words of memory beginning at \( D_2(B_2) \) to the range of registers \( R_1 \) through \( R_3 \).

Since one of the responsibilities of a subroutine is to assure that registers contain the same contents at exit that they did at entry, we can use STM and LM to save and restore them.
Saving and Restoring Registers

ROUTINE STM R0,R15,SAVEAREA Save all regs
... body of routine ...
LM R0,R15,SAVEAREA Restore all regs
BR R14 Return to caller ...
SAVEAREA DS 16F Store regs here
Type A Data: Addresses

- `label DC A(exp)` [or `DS A`]

If `exp` is an integer (non-negative in ASSIST), the generated fullword will have the binary representation of the integer (same as `F'exp'`)

If `exp` is the label of an instruction or a data area, or is of the form `label+n` or `label-n`, the generated fullword will contain the appropriate address
Type A Data: Addresses

- label  DC  A(exp)  [or  DS  A]

- If `exp` is the label of an EQU of a non-negative integer, then the symbol is interpreted as a non-negative integer, and the generated fullword will have the binary representation of the integer.
Type A Data: Addresses

- The following are examples
  - DC A(123) generates 0000007B
  - DC A(R12) generates 0000000C
  - DC A(SAVE) generates addr of SAVE

- **Very important**: All that is known at assembly time is the relative location, not the memory address; it is left to the "program loader" to adjust the relative location at execution time, to make it the address in memory
Type A Data: Addresses

But why bother? Why not just use LA? The answer is that a single base register can address only 4096 bytes (000–FFF).

LA    R4,TABLE3   This fails!
*
L     R4,ATABLE3  This works!
...
ATABLE3    DC    A(TABLE3)
TABLE1     DS    1024F  (= 4096 bytes)
TABLE2     DS    1024F
TABLE3     DS    1024F
...

The MULTIPLY and DIVIDE Instructions

In Which We Encounter "Higher" Math
Multiplication

- MULTIPLY, like ADD and SUBTRACT, comes in two flavors: RR and RX

- Both RR and RX require the first operand to be an even/odd pair of registers, implicitly specified by the even-numbered register

- The RR and RX formats are
  
  \[
  \begin{align*}
  \text{label} & \quad \text{MR} & \quad R_1, R_2 \\
  \text{label} & \quad M & \quad R_1, D_2 (X_2, B_2)
  \end{align*}
  \]
Multiplication

- The **multiplicand** is the word in register $R_{1+1}$ (the 2nd of the pair)

- The **multiplier** is either the word in register $R_2$ or the word whose address is $D_2(X_2,B_2)$

- The **product** will be two words long in the even/odd pair $R_1/R_{1+1}$

- The Condition Code is not changed by MULTIPLY
Multiplication

Multiplicand

\( R_1 \) (even)

\( R_{1+1} \) (odd)

(R_2 \text{ or } D_2(X_2, B_2)

Multiplier

(Product)

Word in reg or word in memory

Even

Odd
For example: if \( c(R9) = 00000003 \), \( c(R7) = \text{FFFFFFFFFFD} \) \((-3)\), and \( c(R6) \) is anything

Then \( \text{MR 6, 9} \) leaves \( R9 \) unchanged and the result in \( R6/R7 = \text{FFFFFFFF FFFFFFFF7} \) (which is -9 in decimal)

Note that the magnitude of the result must be very large before the even-numbered register has anything besides sign bits
Multiplication Examples

- Note that **MR 8,9** squares the value in R9, with the result in R8/R9
  - What does **MR 8,8** do?

- Some examples follow, all of which assume:
  - c(R0)=F01821F0, c(R1)=FFFFFFFF
  - c(R2)=00000003, c(R3)=00000004
  - **WORD1**  DC  F'10'
  - **WORD2**  DC  F'-2'
## Multiplication Examples

- **MR 2,1:**  \[ R2/R3 = \text{FFFFFFFC} \]
- **MR 2,2:**  \[ R2/R3 = \text{0000000C} \]
- **MR 2,3:**  \[ R2/R3 = \text{00000010} \]
- **M 2,WORD1:**  \[ R2/R3 = \text{00000028} \]
- **M 0,WORD2:**  \[ R0/R1 = \text{00000002} \]
Division

- DIVIDE also comes in RR and RX formats

- Both RR and RX require the first operand to be an even/odd pair of registers, implicitly specified by the even register

- The RR and RX formats are
  
  \[
  \text{label} \quad \text{DR} \quad R_1, R_2 \\
  \text{label} \quad D \quad R_1, D_2 (X_2, B_2)
  \]
Division

- The dividend (numerator) is two words long in the even/odd pair $R_1/R_{1+1}$

- The divisor is either the word in register $R_2$ or the word whose address is $D_2(X_2, B_2)$

- The remainder will be in register $R_1$ with sign the same as the dividend's

- The quotient will be in register $R_{1+1}$ with sign following the usual rules of algebra
**Division**

\[ \text{Dividend} \]

\[ \begin{array}{c|c}
\text{(even)} & s & R_1 \\
\hline
\text{(odd)} & & R_1 + 1 \\
\end{array} \]

\[ \text{Divisor} \]

\[ \begin{array}{c|c}
\text{R}_2 \text{ or } \text{D}_2(X_2, B_2) \\
\hline
\text{(Word in reg or word in memory)} \\
\end{array} \]

\[ \text{Remainder} \]

\[ \begin{array}{c|c}
s & R_1 \\
\hline
\end{array} \]

\[ \text{Quotient} \]

\[ \begin{array}{c|c}
s & R_1 + 1 \\
\hline
\end{array} \]
Division

- If the quotient cannot be represented as a 32-bit signed integer, a "fixed-point divide" exception occurs; this also happens if the divisor is zero.

N.B. The dividend will often fit into a single register, but the sign must be correct in both registers of the pair.

This can be assured by first multiplying by 1 (product is then in the register pair).
Division Examples

Some examples, all of which assume:
- \(c(R2) = 00000000\), \(c(R3) = 00000014\) (20)
- \(c(R4) = FFFFFFFFF\), \(c(R5) = FFFFFFF10\) (-240)
- \(c(R1) = 00000003\)
- `WORD1` DC F' -4'
- `WORD2` DC F'14'

**DR 2,1** (so dividend is in R2/R3)
\[R2/R3 = 00000002 \ 00000006\] (2,6)
Division Examples

- **DR 2, 4**
  \[ \frac{R2}{R3} = 00000000 \text{ FFFFFFFEC} \quad (0,-20) \]

- **DR 2, 5**
  \[ \frac{R2}{R3} = 00000014 \ 00000000 \quad (20,0) \]

- **DR 4, 1** (so dividend is in R4/R5)
  \[ \frac{R4}{R5} = 00000000 \text{ FFFFFFFB0} \quad (0,-80) \]

- **D 2, WORD1**
  \[ \frac{R2}{R3} = 00000000 \text{ FFFFFFFFB} \quad (0,-5) \]
Division Examples

D 2, WORD2
R2/R3 = 00000006 00000001 (6,1)

D 4, WORD1
R4/R5 = 00000000 0000003C (0,60)

D 4, WORD2
R4/R5 = FFFFFFFFFF FFFFFFFFFF (2,17)
Next Time

- Tomorrow, we will look at how decimal arithmetic is performed, and how numbers are converted from binary to decimal to character (and the reverse)

- Accurate decimal arithmetic is an important characteristic of z/Architecture, particularly for financial applications

- We'll also cover the instructions which perform the operations AND, OR, and XOR