Assembler Language
"Boot Camp"
Part 2 - Instructions and Addressing
SHARE 117 in Orlando
Session 9211
August 8, 2011
Introduction

Who are we?

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

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

- Basic z/Architecture and Program Execution
- General-Purpose Registers; Addressing using a Base Register and a Displacement
- Basic Instruction Formats
- Some Conventions and Standards
- A Complete Program
Basic z/Architecture and Program Execution
There's more to a computer than just memory

We need to understand the architecture in order to understand how instructions execute

We will need to understand how instructions execute in order to understand how programs accomplish their goals

Assembler Language provides the capability to create machine instructions directly
In addition to memory, there are (at least):

- A Central Processing Unit (CPU)
- A Program Status Word (PSW)
- Sixteen general-purpose registers
- Floating-point registers
- Many other elements beyond our scope
Common, Shared Memory for Programs and Data

One of the characteristics of z/Architecture is that programs and data share the same memory (this is very important to understand).

The effect is that:
- Data can be executed as instructions
- Programs can be manipulated like data
Common, Shared Memory for Programs and Data

This is potentially very confusing

Is $05EF_{16}$ the numeric value $1519_{10}$ or is it an instruction?

It is impossible to determine the answer simply by inspection

Then how does the computer distinguish between instructions and data?
The Program Status Word (PSW) always has the memory address of the next instruction to be executed.

It is this fact which defines the contents of that memory location as an instruction.

We will see the format of the PSW in Part 4, but for now, we look at how it is used to control the execution of a program (a sequence of instructions in memory).
The Execution of a Program

In order to be executed by a CPU, an assembler language program must first have been

1. Translated ("assembled") to machine language "object code" by the assembler
2. Placed ("loaded") into the computer memory

Once these steps are complete, we can begin the execution algorithm
The Execution of a Program

- Step 1 - The memory address of the first instruction to be executed is placed in the PSW

- Step 2 - The instruction pointed to by the PSW is retrieved from memory by the instruction unit

- Step 3 - The PSW is updated to point to the next instruction in memory
The Execution of a Program

- Step 4 - The retrieved instruction is executed
  - If the retrieved instruction did not cause a Branch (GoTo) to occur, go back to Step 2
  - Otherwise, put the memory address to be branched to in the PSW, then go back to Step 2

- This leaves many questions unanswered (How does the algorithm stop?) but provides the basic ideas
General-Purpose Registers and Base-Displacement Addressing
General-Purpose Registers

- z/Architecture has sixteen General Purpose registers
- Each register is 64 bits in size
- Each register is identified by a unique number: 0, 1, ..., 15 (0–F in hexadecimal)
- Registers have faster access than memory, and are used both for computation and for addressing memory locations
Base-Displacement Addressing

- Recall that every byte of a computer's memory has a unique address, which is a non-negative integer.

- This means that a memory address can be held in a general purpose register.

- When it serves this purpose, a register is called a base register.
Base-Displacement Addressing

- The contents of the base register (the base address of the program) depends on where in memory the program is loaded.

- But locations relative to one another within a program don't change, so displacements are fixed when the program is assembled.
Base-Displacement Addressing

- z/Architecture uses what is called base-displacement addressing for many instruction operands.

- A relative displacement is calculated at assembly time and is stored as part of the instruction, as is the base register number.

- The base register's contents are set at execution time, depending upon where in memory the program is loaded.
Base-Displacement Addressing

The sum of the base register contents and the displacement gives the operand's effective address in memory.

For example, if the displacement is 4 and the base register contains 00000000 0000007C, the operand's effective address is 000080 (written intentionally as 24 bits).
When an address is coded in base-displacement form, it is called explicit (we will see implicit addresses shortly).

When coding base and displacement as part of an assembler instruction, the format is often D(B), depending on the instruction.

- D is the displacement, expressed as a decimal number in the range 0 to 4095 (hex 000-FFF).
- B is the base register number, except that 0 means "no base register," not "base register 0."
Base-Displacement Addressing

- Some examples of explicit addresses: 4(1) 20(13) 0(11)

- In 0(11), the base register gives the desired address without adding a displacement.

- When the base register is omitted, a zero is supplied by the assembler.
  - So coding 4 is the same as coding 4(0)
Base-Displacement Addressing

- Some instructions allow for another register to be used to compute an effective address

- The additional register is called an index register

- In this case, the explicit address operand format is D(X,B) (or D(,B) if the index register is omitted)
  - D and B are as above
  - X is the index register number
Base-Displacement Addressing

For example, 4(7,2) is computed as an effective address by adding 4 plus the contents of index register 7 plus the contents of base register 2.

Again, 0 means "no register" rather than "register 0"
- This applies to the index register position of an RX instruction (just as for the base register position) in any instruction that has one.
Base-Displacement Addressing

- We will see next how the assembler encodes instructions, converting them to a string of bits called object code.

- As a preview, for D(B) format operands the conversion is to \( h_B h_D h_D h_D \), thus taking two bytes (each \( h \) represents a hex digit, two per byte).
Base-Displacement Addressing

This explains why the displacement DDD is limited to a maximum of 4095 (hex FFF)

Some recent instructions are called "relative" instructions and need no base register, and some use 20-bit displacements

These are beyond our scope

Also beyond our scope are instructions which use all 64 bits of a register
A Note on 64-bit Registers

- All sixteen registers in z/Architecture are 64 bits long (two fullwords)
  - The first (left) fullword has bits 0-31
  - The second (right) fullword has bits 32-63

- The second fullword is the only one we will see for the rest of the week
  - It is the only one referenced by the instructions we will see
  - So, from this point on, only the second fullword will be shown
Basic Instruction Formats
Instruction Formats

The process of "assembling" includes encoding programmer-written symbolic instructions. These are then converted by the assembler to machine instructions (which are strings of bits).

The assembler can also create data areas as part of a program.
Instruction Formats

- A program is a combination of instructions and data areas whose relative locations are fixed at assembly time.

- This point is very important to understand - it is part of what makes assembler language difficult to learn.

- Assembler language has no "variables," just data areas.
Instruction Formats

- There are five basic machine instruction formats we will need to understand

- They are similar, but different in their operands

- Each machine instruction requires 2, 4, or 6 bytes of memory (usually referred to as 1, 2, or 3 halfwords because all instructions are halfword aligned)
Instruction Formats

- Each machine instruction that we will see begins with a one-byte operation code.
- The five formats are named according to the types of operand each has.
Instruction Formats

- **RR - Register-Register**
  - Occupies one halfword and has two operands, each of which is in a register (0 - 15)

- **RX - Register-index register**
  - Occupies two halfwords and has two operands; the first is in a register, the second is in a memory location whose address is of the form D(X,B)
Instruction Formats

- **RS - Register-Storage**
  - Occupies two halfwords and usually has three operands: two register operands and a memory address of the form D(B)

- **SI - Storage-Immediate**
  - Occupies two halfwords and has two operands: a byte at memory address D(B) and a single "immediate" data byte contained in the instruction
Instruction Formats

SS - Storage-Storage
- Occupies three halfwords and has two memory operands of the form D(B) or D(L,B); each operand may have a length field - this depends on the specific instruction

There are variations of these formats, including many less frequently executed operations whose op codes are two bytes long instead of one
**RR Instructions**

- Our first machine instruction is type RR and will add the contents of two registers, replacing the contents of the first register with the sum.

- This instruction is called ADD, and is written symbolically as $AR \ R_1, R_2$.

- An example is $AR \ 2, 14$ which adds the contents of register 14 to the contents of register 2; the sum replaces the contents of register 2.
**RR Instructions**

- Note that the "direction" of the add is right to left; this is a consistent rule for all but a few instructions.

- The assembly process will convert the mnemonic AR to the operation code **1A**.

- It will also convert each of the two register values to hexadecimal (2 and E).
RR Instructions

- The instruction would then be assembled as the machine instruction 1A2E at the next available location in the object code.

- In bits this is: 0001101000101110

- All RR instructions assemble as $h_{OP}h_{OP}h_{R1}h_{R2}$

- Another instruction is SUBTRACT, which is written symbolically as $SR_{R1,R2}$
RR Instructions

- For example, $\text{SR 2,14}$ would subtract the contents of R14 from R2, replacing the contents of R2 with the difference.

- Note the "Rn" shorthand convention for "register n".

- The op code for $\text{SR}$ is $\text{1B}$.

- Both ADD and SUBTRACT can cause overflow - we must be able to cope with this.
Our final (for now) RR instruction is LOAD, written symbolically as $\text{LR} \ R_1, R_2$

The contents of the first operand register are replaced by the contents of the second operand register ($R_2$ contents are unchanged)

The op code for LR is 18

LOAD cannot cause overflow
**RR Instructions**

**Exercises:**
- Encode **AR 1,15** and **SR 0,0**
- Decode **1834**

If \( c(R0) = 001A2F0B \), \( c(R1) = FFFFA21C \), and \( c(R6) = 000019EF \) for each instruction:
- After **LR 6,0**, \( c(R6) = ? \)
- After **AR 1,6**, \( c(R1) = ? \)
- After **SR 1,6**, \( c(R1) = ? \)

**001A2F0B, FFFFB0C0B, FFFF882D**
RX Instructions

- This format has a register operand and a memory address operand (which includes an index register - thus, the "RX" notation)

- The RX version of LOAD is \( L R_1, D_2(X_2, B_2) \) which causes the fullword at the memory location specified by \( D_2(X_2, B_2) \) to be copied into register \( R_1 \), replacing its contents

- Note: the mnemonics (LR and L) determine the format (RR vs RX) of the instruction
RX Instructions

Although z/Architecture doesn't require it, the second operand's effective address should also be on a fullword boundary (thus ending in ...0, ...4, ...8, or ...C)

This is a good habit, and ASSIST/I does require it

The encoded form of an RX instruction is:

\[ h_{\text{OP}} h_{\text{OP}} h_{\text{R1}} h_{\text{X2}} h_{\text{B2}} h_{\text{D2}} h_{\text{D2}} h_{\text{D2}} \]
The opcode for LOAD is 58, so the encoded form of \( \text{L 2,12 (1,10)} \) is 5821A00C.

The reverse of LOAD is STORE, coded symbolically as \( \text{ST R}_1, D_2 (X_2, B_2) \), and which causes the contents of \( \text{R}_1 \) to replace the contents of the fullword at the memory location specified by \( \text{D}_2 (X_2, B_2) \) (violates the "right to left" rule of thumb).

The opcode for ST is 50.
RX Instructions

Exercises:
- Encode \texttt{ST 2,10 (14,13)}
- Decode \texttt{5811801C}

If c(R2) = 000ABC10, c(R3) = 0000000B, and c(R4) = 000C1F11, what is the effective address of the second operand?
- \texttt{L 0,16 (,2)}
- \texttt{ST 15,20 (3,4)}
- \texttt{L 8,0 (2,4)}
RX Instructions

- We have seen two RR instructions, AR and SR (ADD and SUBTRACT)

- Each has an RX counterpart
  - A  $R_1, D_2 (X_2, B_2)$  [ADD]
  - S  $R_1, D_2 (X_2, B_2)$  [SUBTRACT]

- We now have almost enough instructions for a complete program
Some Coding Conventions and Standards

Assembler Statement Coding Conventions and Program Entry and Exit Rules
Coding Assembler Statements

- Recall the two ways we can view an instruction
  - Symbolic: AR 3,2
  - Encoded: 1A32

- The encoded form is easily the most important
  - "Object Code - Nothing Else Matters"

- But we write programs using the symbolic form
Format of a Symbolic Instruction

- Label (optional)
  - Begins in Column 1
  - 1 to 63 characters (1 to 8 in ASSIST/I)
  - First character is alphabetic
  - Other characters may be 0 - 9 (or _, except in ASSIST/I)

- Mixed case not allowed in ASSIST/I
Format of a Symbolic Instruction

- Operation code mnemonic (required)
  - May begin in column 2 or after label (at least one preceding blank is required)
  - Usually begins in column 10

- Operands (number depends on instruction)
  - Must have at least one blank after mnemonic
  - Separated by commas (and no blanks)
  - Usually begins in column 16
Format of a Symbolic Instruction

Continuation (Optional)

- Non-blank in column 72 means the next statement is a continuation and must begin in column 16!
- Also, columns 1 - 15 of the next statement must be blank
Format of a Symbolic Instruction

- **Line comments (Optional)**
  - Must have at least one blank after operands
  - Usually begin in column 36, cannot extend past column 71
  - Some begin the comment with // or ; to be consistent with other languages

- **Comment Statements**
  - Asterisk (*) in column 1 means the entire statement is a comment
  - These also cannot extend past column 71
In addition to symbolic instructions which encode to machine instructions, there are also assembler instructions or directives which tell the assembler how to process, but which may not generate object code.

The CSECT instruction (Control SECTion) is used to begin a program and appears before any executable instruction.

- label           CSECT
Assembler Instructions (Directives)

- The END instruction defines the physical end of an assembly, but *not* the logical end of a program
  
  ```
  END label
  ```

- The **logical** end of our program is reached when it returns to the program which gave it control
The DC instruction reserves storage at the place it appears in the program, and provides an initial value for that memory

- **label DC mF'n'**
  - where \( m \) is a non-negative integer called the duplication factor, assumed to be 1 if omitted
  - Generates \( m \) consecutive fullwords, each with value \( n \)

IBM calls DC "define constant" but a better choice is "define storage with initial value"
Assembler Instructions (Directives)

- What's generated by `TWELVES DC 2F'12'

- `0000000C0000000C`

- There are many other data types besides fullword F

- A variation is provided by the `DS` (Define Storage) instruction, which also reserves storage but does not give it an initial value (so contents are unpredictable)
Entry Conventions

- There are two registers which, by convention, have certain values at the time a program begins.

- Register 15 will have the address of the first instruction to be executed.
Register 14 will have the address of the instruction to be given control when execution is complete

- To get there, execute a "branch":
  - **BCR B'1111',14**
  - This instruction will be explained shortly
A Complete Program

- This is the first demo program in the materials provided for these sessions

- It has only five executable instructions and reserves three fullwords of storage for data, the first two of which have an initial value

- In the next session we will analyze the program thoroughly, but for today, we end with just a list of the assembler statements
* This program adds two numbers that are taken
* from the 5th and 6th words of the program.
* The sum is stored in the 7th word.

ADD2   CSECT
L      1,16(,15)   Load 1st no. into R1
L      2,20(,15)   Load 2nd no. into R2
AR     1,2         Get sum in R1
ST     1,24(,15)   Store sum
BCR    B'1111',14  Return to caller
DC     F'4'        Fullword initially 4
DC     F'6'        Fullword initially 6
DS     F            Rsrvd only, no init
END    ADD2
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>CODE</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>ADD2</td>
<td>CSECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>5810 F010</td>
<td>L</td>
<td>1,16(,15)</td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td>5820 F014</td>
<td>L</td>
<td>2,20(,15)</td>
<td></td>
</tr>
<tr>
<td>000008</td>
<td>1A12</td>
<td>AR</td>
<td>1,2</td>
<td></td>
</tr>
<tr>
<td>00000A</td>
<td>5010 F018</td>
<td>ST</td>
<td>1,24(,15)</td>
<td></td>
</tr>
<tr>
<td>00000E</td>
<td>07FE</td>
<td>BCR</td>
<td>B'1111',14</td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td>00000004</td>
<td>DC</td>
<td>F'4'</td>
<td></td>
</tr>
<tr>
<td>000014</td>
<td>00000006</td>
<td>DC</td>
<td>F'6'</td>
<td></td>
</tr>
<tr>
<td>000018</td>
<td></td>
<td>DS</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

END ADD2
A Complete Program

Now that we have assembled the program,
- What does that stuff on the left mean?
- How did we get there?
- And what do we do with it, now that it's assembled?

Tune in tomorrow!