Introduction to Assembler Programming

SHARE Boston 2013

Sharuff Morsa smorsa@uk.ibm.com

Session 13673 Part 1

Session 13675 Part 2

Introduction

- Who am I?
  - Sharuff Morsa, IBM Hursley Labs UK

    smorsa@uk.ibm.com
  - Material was written by Richard Cebula

Introduction to Assembler Programming

- Why assembler programming?
- Prerequisites for assembler programming on System z
- Moving data around
- Logical instructions
- Working with HLASM
- Addressing data
- Branching
- Arithmetic
- Looping
- Calling conventions
- How to read POPs

Audience

- This is an INTRODUCTION to assembler programming
- The audience should have a basic understanding of computer programming
- The audience should have a basic understanding of z/OS
- At the end of this course the attendee should be able to:
  - Understand the basics of assembler programming on System z
  - Use a variety of simple machine instructions

Why program in assembler?

- Assembler programming has been around since the very start of computer languages as an easy way to understand and work directly with machine code
- Assembler programming can produce the most efficient code possible
  - Memory is cheap
  - Chips are fast
  - So what?
- Assembler programming TRUSTS the programmer
  - Humans are smart (?)
  - Compilers are dumb (?)
- Assembler programming requires some skill
  - No more than learning the complex syntax of any high-level language, APIs (that change every few years), latest programming trends and fashions
  - Your favorite language will too become old, bloated and obsolete!

Why program in assembler?

- Misconceptions of assembler programming
  - I need a beard right?
  - It's too hard...
  - Any modern compiler can produce code that's just as efficient now days...
  - I can do that quicker using...
  - But assembler isn't portable...

Why program in assembler?

- Misconceptions of assembler programming
  - I need a beard right?
    - Assembler programmers tend to be old*er* and more experienced and typically wiser
    - Experienced programmers that have used assembler know that they can rely on it for the most complex of programming tasks
  - It's too hard...
    - Learning assembler is just like learning any other language
    - Each instruction to learn is as easy as the next
    - Syntax is consistent
    - No difficult APIs to get to grips with
  - Any modern compiler can produce code that's just as efficient now days...
    - Compilers CAN produce efficient code but that is not to say that they WILL
    - Optimization in compilers is a double-edged sword – compilers make mistakes
  - I can do that quicker using...
    - Good for you, so can I...
  - But assembler isn't portable...
    - Neither is Java, nor C, nor C++... portability depends on your definition of it

Why program in assembler?

- The assembler mindset
  - You are not writing code – you are programming the machine
  - You must be precise
  - Your assembler program is no better than your programming
- Assembler programming provides the programmer with TOTAL freedom
  - What you choose to do with that freedom is your choice and your responsibility
- The code you write is the code that will be run

Prerequisites for assembler programming on System z

- Basic programming knowledge is assumed
- Understand binary and hexadecimal notation
  - 2's complement, signed arithmetic, logical operations
- A basic knowledge of computer organisation
- Basic z/OS knowledge
  - ISPF, JCL, SDSF
- A copy of z/Architecture Principles of Operation – aka POPs
  - POPs is the processor manual
  - Optionally, a copy of the z/Architecture reference summary

Brief overview of z/Architecture

- z/Architecture – the processor architecture used for all System z Mainframes
- Processor specifications vary
  - Processor level – the physical (or virtual) chip used
  - Architecture level – the instruction specification of a chip
- System z is a 64-bit, big-endian, rich CISC (over 1000 instructions) architecture with:
  - 16 64-bit General Purpose Registers (GPRs)
  - 16 32-bit Access Registers (ARs)
  - 16 64-bit Floating Point Registers (FPRs)
  - 16 64-bit Control Registers (CRs)
  - 1 Program Status Word (PSW)
  - And other features including Cryptography, I/O dedicated channel processors
- All registers are numbered 0-15; the instructions used distinguish which 0-15 means which register
- A WORD → 32-bits, DOUBLEWORD → 64-bits, HALFWORD → 16-bits

Brief overview of z/Architecture – Understanding Registers

- GRPs – used for arithmetic, logical operations, passing operands to instructions, calling subroutines etc
- ARs – used in “Access Register” mode – provides the ability to access another address space
- FPRs – used for floating point instructions, both binary and hexadecimal arithmetic
  - DECIMAL arithmetic is performed using GPRs
- CRs – used for controlling processor operations
- PSW – provides the status of the processor consisting of 2 parts:
  - PSW Flags – these show the state of the processor during instruction execution
  - Instruction address – this is the address of the next instruction to be executed
- GPRs and FPRs can be paired
  - GPRs form even-odd pairs, i.e. 0-1, 2-3,...,14-15
  - FPRs pair evenly / oddly, i.e. 0-2, 1-3,...,13-15

Understanding Binary Numbers

Binary Numbers

- Nearly all computers today use binary as the internal "language"
  - We need to understand this language to fully understand instructions and data
  - Even decimal numbers are represented internally in binary!
- Binary numbers can get very long, so we use hexadecimal ("hex") as a shorthand
  - A hex digit is simply a group of four binary digits (bits)

Binary Numbers

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

Binary Numbers

- Consider how we write numbers in base 10, using the digits 0 - 9:
- BASE 10

$$832_{10} = 800_{10} + 30_{10} + 2_{10}$$
$$= 8 \times 100 + 3 \times 10 + 2 \times 1$$

- For numbers in base 2 we need only 0 and 1:

$$1101_2 = 1000_2 + 100_2 + 00_2 + 1_2$$

- But because it requires less writing, we usually prefer base 16 to base 2

Binary Numbers

- To convert from binary to hexadecimal
- 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
- Convert each group of four binary digits to a hexadecimal digit

0001	1000	1100	0111
1	8	C	7

Main Storage Organization

Main Storage Organization

- A computer's memory is simply a collection of billions of such systems implemented using electronic switches
- 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 = 8 bits
- One word = four bytes = 32 bits
- Double word = eight bytes = 64 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 word (32 bits) is aligned on a 4-byte boundary (addresses 0, 4, 8, 12, 16, 20, ...)
- A double word is aligned on a 8-byte boundry (0, 8, 16, 32, ...)
- Remember, memory addresses **refer to bytes**, not bits or words

Main Storage Organization

- 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

- This is potentially very confusing
  - Is $05EF_{16}$ the numeric value 1519_{10} or is it an instruction?

Main Storage Organization

- Instructions are executed one at a time
- The Program Status Word (PSW) always has the memory address of the next instruction to be executed

More on the PSW later

Base-Displacement Addressing

Base-Displacement Addressing

- 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

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

Base-Displacement Addressing

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

Base-Displacement Addressing

- For example:
  if the displacement is **4**
  and
  the base register contains **00000000 000A008C**

  The operand's effective address is

  00000000 000A0090

- When an address is coded in base-displacement form
  – it is called an explicit address

  We'll see implicit addresses later

Base-Displacement Addressing

- 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 (register zero) 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

And then there is Relative addressing -more later

Introduction to Assembler Programming
Moving Data

Moving Data – Loading from Register to Register

- The LOAD REGISTER (LR) instruction is used to load the value stored in one register to another

```
LR 1,2     LOAD REGISTER 2 INTO REGISTER 1 (32-BITS)
```

- The instruction copies 32-bits from a register to another

The copy is right to left

- The instruction has a 64-bit variant LOAD GRANDE REGISTER (LGR)

```
LGR 1,2     LOAD REGISTER 2 INTO REGISTER 1 (64-BITS)
```

- The instruction has a 16-bit variant LOAD HALFWORD REGISTER

```
LHR 1,2     LOAD REGISTER 2 INTO REGISTER 1 (16-BITS)
```

Moving Data – Loading from Memory to Register

- The LOAD (L) instruction is used to load the value stored in memory to a register

```
L 1,NUMBER     LOAD REGISTER 1 WITH THE VALUE NUMBER
```

- The instruction copies 32-bits from memory to a register

  The copy is right to left

- The instruction has a 64-bit variant LOAD GRANDE (LG)

```
LG 1,NUMBER    LOAD REGISTER 1 WITH THE VALUE NUMBER
```

- The instruction has a 16-bit variant LOAD HALFWORD REGISTER

```
LH 1,NUMBER    LOAD REGISTER 1 WITH THE VALUE NUMBER
```

Moving Data – Storing from a Register to Memory

- The STORE (ST) instruction is used to store the value in a register to memory

```
ST 1,address       STORE REGISTER 1 TO address (32-BITS)
```

- The instruction copies 32-bits from a register to memory

The copy is left to right

- The instruction has a 64-bit variant STORE GRANDE (STG)

```
STG 1,address      STORE REGISTER 1 TO address (64-BITS)
```

- The instruction has a 16-bit variant STORE HALFWORD

```
STH 1,address      STORE REGISTER 1 TO address (16-BITS)
```

Moving Data – Moving data without registers

- The MOVE (MVC) instruction can be used to move data in memory without the need for a register

```
MVC  OUTPUT,INPUT      MOVE  INPUT  TO  OUTPUT
```

- The MVC instruction can move up to 256 bytes from one area of memory to another

- The MVCL instruction can move up to 16 Meg (but uses different parameters)

- The MVCLE instruction can move up 2G (or up to 16EB in 64-bit addressing)

- In all cases, the move instruction moves 1 byte at a time <u>(left to right)</u>

Introduction to Assembler Programming
Logical Operations

Logical Instructions – EXCLUSIVE OR (X, XG, XR, XGR, XC)

- The EXCLUSIVE OR instructions perform the EXCLUSIVE OR *bit-wise* operation

```
X    1,NUMBER     XOR REGISTER 1 WITH NUMBER (32-BITS)
XG   1,NUMBER     XOR REGISTER 1 WITH NUMBER (64-BITS)
XR   1,2          XOR REGISTER 1 WITH REGISTER 2 (32-BITS)
XGR  1,2          XOR REGISTER 1 WITH REGISTER 2 (64-BITS)
XC   NUM1,NUM2    XOR NUM1 WITH NUM2 (UP TO 256-BYTES)
```

- Notice a pattern with the instruction mnemonics?
  - Rules of thumb:
    - G → 64bits (DOUBLEWORD)
    - H → 16bits (HALFWORD)
    - R → register
    - C → character (byte / memory)
    - L → logical (i.e. unsigned)

Logical Instructions – AND (N*x*), OR (O*x*)

- The AND instructions perform the AND *bit-wise* operation

```
N    1,NUMBER     AND REGISTER 1 WITH NUMBER (32-BITS)
NG   1,NUMBER     AND REGISTER 1 WITH NUMBER (64-BITS)
NR   1,2          AND REGISTER 1 WITH REGISTER 2 (32-BITS)
NGR  1,2          AND REGISTER 1 WITH REGISTER 2 (64-BITS)
NC   NUM1,NUM2    AND NUM1 WITH NUM2 (UP TO 256-BYTES)
```

- The OR instructions perform the OR *bit-wise* operation

```
O    1,NUMBER     OR REGISTER 1 WITH NUMBER (32-BITS)
OG   1,NUMBER     OR REGISTER 1 WITH NUMBER (64-BITS)
OR   1,2          OR REGISTER 1 WITH REGISTER 2 (32-BITS)
OGR  1,2          OR REGISTER 1 WITH REGISTER 2 (64-BITS)
OC   NUM1,NUM2    OR NUM1 WITH NUM2 (UP TO 256-BYTES)
```

A word on instruction choice

- In 5 basic operations (loading, storing, AND, OR, XOR) we have already seen over 25 instructions!
- How do I decide which instruction to use?
  - The instruction should be chosen for:
    - Its purpose, e.g. don't use a STORE instruction to LOAD a register – it won't work!
    - Its data, e.g. 32-bits, 16-bits, 64-bits, bytes?
- Many instructions can perform *similar* operations, e.g.

```
XR    15,15     XOR REGISTER 15 WITH REGISTER 15
L     15,=F'0'  LOAD REGISTER 15 WITH 0
```

- Different instructions NEVER do the same thing even if you think they do
  - The result does not justify the means

Introduction to Assembler Programming Working with HLASM

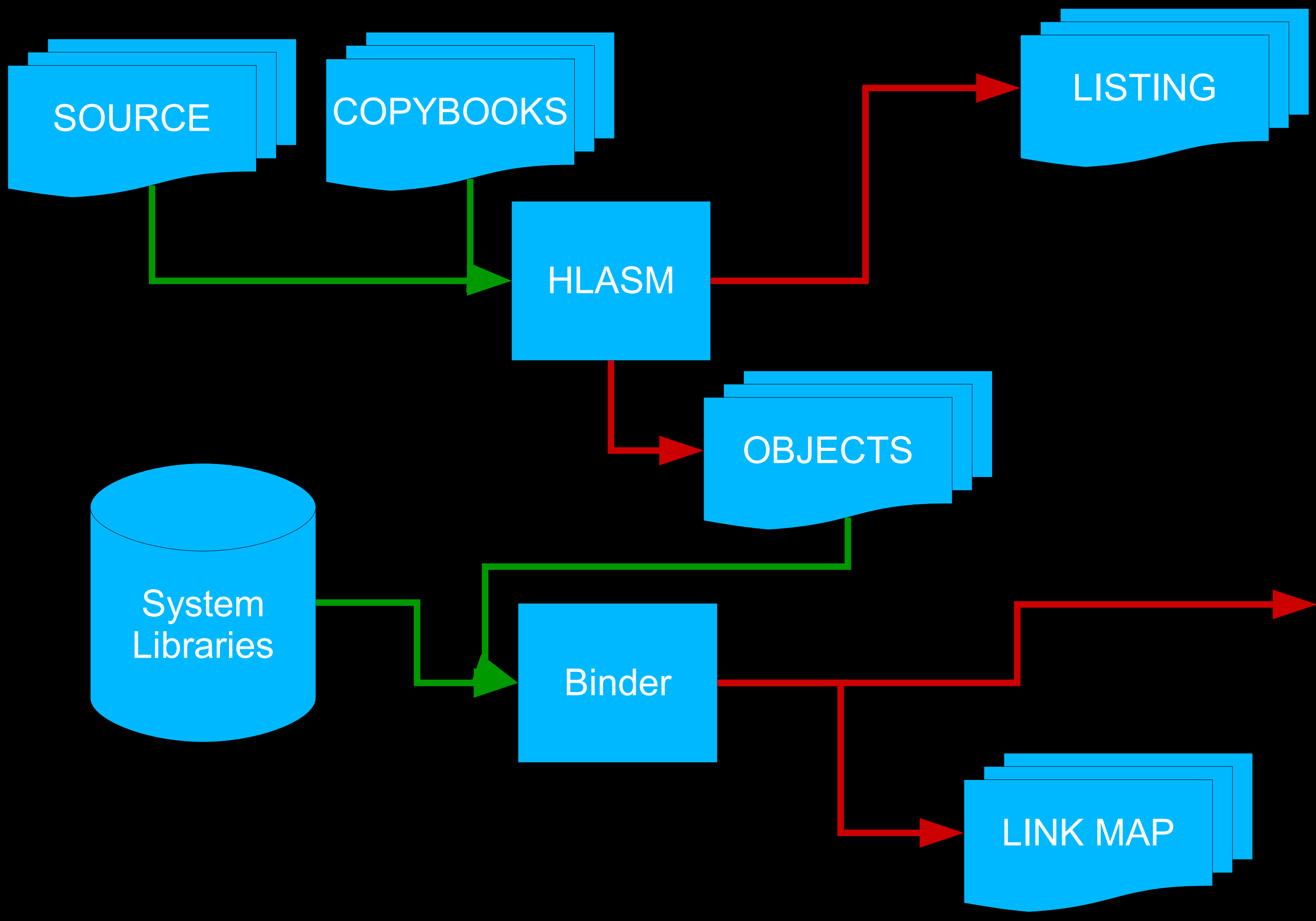
Working with HLASM

- HLASM – IBM's High Level Assembler
- Available on z/OS, z/VM, z/VSE, z/Linux and z/TPF
- *High Level Assembler???* - YES!
  - Provides a wide range of assembler *directives*
    - An assembler *directive* is not a machine instruction
    - It is an instruction to the assembler during assembly of your program
  - An incredible macro programming facility
  - Structured programming

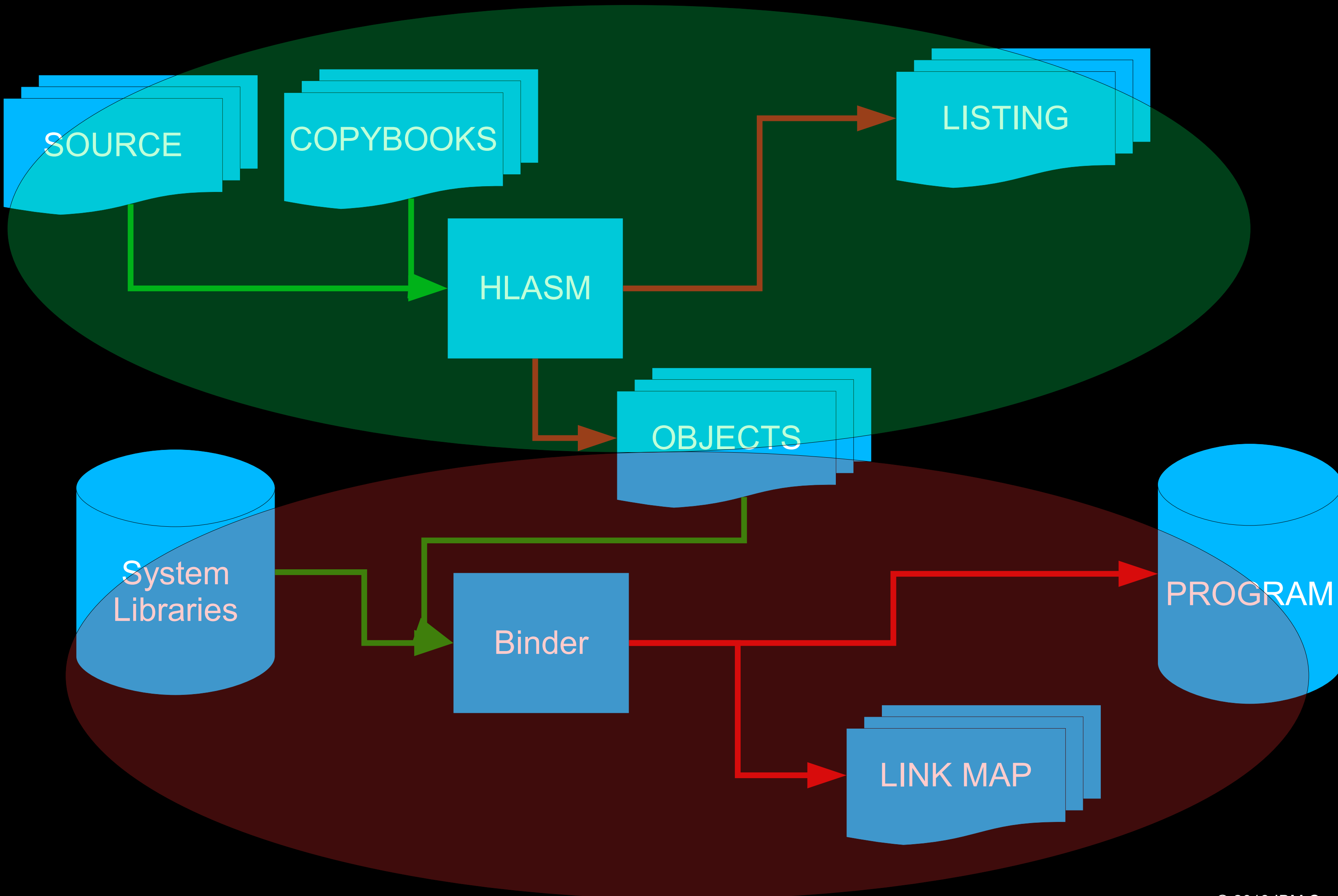
Working with HLASM – Producing a program

- Assembling is the process of changing assembler source code into OBJECT DECKS
  - To assemble, use an assembler
- The assembler produces 2 outputs
  - OBJECT DECKS – this is the object code that is used as input to binding
  - Listing – this provides shows any errors, all diagnostics and human readable output from the assemble phase
- Binding is the process of combining object code into a LOAD MODULE
  - To bind, us a Binder
- The Binder produces 2 outputs
  - LOAD MODULE – this is the bound object decks forming an executable program
  - A LOAD MAP – this is the Binder equivalent of an assembler listing
- A LOAD MODULE can be loaded into memory by the operating system and run

Working with HLASM – Assembling and Binding a program



Working with HLASM – Assembling and Binding a program



IBM

Working with HLASM – A look at syntax

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
VIEW       RCEBULA.APAR.PM76008.SOURCE(LARLLOAD) - 01.04       Columns 00001 00080
Command ===>                                                      Scroll ===> CSR
****** ******************************** Top of Data *********************************
 ***********************************************************************
 * SIMPLE DUMMY EXIT FOR HLASM
 ***********************************************************************
 *
 * MAIN PROGRAM STARTS HERE
 *
 LARLLOAD CSECT
 LARLLOAD AMODE 31
 LARLLOAD RMODE 24
 * USUAL PROGRAM SETUP
                   STM   14,12,12(13)
                   BALR  12,0          GET THE CURRENT ADDRESS
                   USING *,12          USE 12 AS THE BASE REGISTER
                   L     1,=F'12'
 LMRET             LM    14,12,12(13)
                   XR    15,15
                   BR    14
 * *********************************************************************
 * END OF PROGRAM
 * *********************************************************************
                   END
****** ******************************* Bottom of Data *******************************
```

IBM

Working with HLASM – A look at syntax

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help
VIEW       RCEBULA.APAR.PM76008.SOURCE(LARLLOAD) - 01.04         Columns 00001 00080
Command ===> ________________________________________________________ Scroll ===> CSR
****** ******************************** Top of Data ********************************
***********************************************************************
* SIMPLE DUMMY EXIT FOR HLASM
***********************************************************************
*
* MAIN PROGRAM STARTS HERE
*
LARLLOAD CSECT
LARLLOAD AMODE 31
LARLLOAD RMODE 24
* USUAL PROGRAM SETUP
         STM   14,12,12(13)
         BALR  12,0              GET THE CURRENT ADDRESS
         USING *,12              USE 12 AS THE BASE REGISTER
         L     1,=F'12'
LMRET    LM    14,12,12(13)
         XR    15,15
         BR    14
* *********************************************************************
* END OF PROGRAM
* *********************************************************************
         END
****** ******************************* Bottom of Data ******************************
```

Comments start with a * in column 1 or appear after free-form instruction operands until column 72

Working with HLASM – A look at syntax

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help

VIEW       RCEBULA.APAR.PM76008.SOURCE(LARLLOAD) - 01.04          Columns 00001 00080
Command ===> ________________________________________________________ Scroll ===> CSR
****** ********************************* Top of Data ********************************
       ***********************************************************************
       * SIMPLE DUMMY EXIT FOR HLASM
       ***********************************************************************
       *
       * MAIN PROGRAM STARTS HERE
       *
       LARLLOAD CSECT
       LARLLOAD AMODE 31
       LARLLOAD RMODE 24
       * USUAL PROGRAM SETUP
                STM    14,12,12(13)
                BALR   12,0              GET THE CURRENT ADDRESS
                USING  *,12              USE 12 AS THE BASE REGISTER
                L      1,=F'12'
       LMRET    LM     14,12,12(13)
                XR     15,15
                BR     14
       * *********************************************************************
       * END OF PROGRAM
       * *********************************************************************
                END
****** ******************************** Bottom of Data ******************************
```

Labels start in column 1

IBM

Working with HLASM – A look at syntax

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help

VIEW       RCEBULA.APAR.PM76008.SOURCE(LARLLOAD) - 01.04        Columns 00001 00080
Command ===> ________________________________________________________ Scroll ===> CSR
****** ********************************* Top of Data ******************************
       ***************************************************************************
       * SIMPLE DUMMY EXIT FOR HLASM
       ***************************************************************************
       *
       * MAIN PROGRAM STARTS HERE
       *
       LARLLOAD CSECT
       LARLLOAD AMODE 31
       LARLLOAD RMODE 24
       * USUAL PROGRAM SETUP
                STM    14,12,12(13)
                BALR   12,0            GET THE CURRENT ADDRESS
                USING  *,12            USE 12 AS THE BASE REGISTER
                L      1,=F'12'
       LMRET    LM     14,12,12(13)
                XR     15,15
                BR     14
       * *************************************************************************
       * END OF PROGRAM
       * *************************************************************************
                END
****** ******************************** Bottom of Data ****************************
```

Instructions start after column 1 or a label

Working with HLASM – A look at syntax

```
 File  Edit  Edit_Settings  Menu  Utilities  Compilers  Test  Help

VIEW      RCEBULA.APAR.PM76008.SOURCE(LARLLOAD) - 01.04      Columns 00001 00080
Command ===> _______________________________________________    Scroll ===> CSR
****** ********************************* Top of Data **********************************
       ***********************************************************************
       * SIMPLE DUMMY EXIT FOR HLASM
       ***********************************************************************
       *
       * MAIN PROGRAM STARTS HERE
       *
       LARLLOAD CSECT
       LARLLOAD AMODE 31
       LARLLOAD RMODE 24
       * USUAL PROGRAM SETUP
                STM   14,12,12(13)
                BALR  12,0             GET THE CURRENT ADDRESS
                USING *,12             USE 12 AS THE BASE REGISTER
                L     1,=F'12'
       LMRET    LM    14,12,12(13)
                XR    15,15
                BR    14
       * ********************************************************************
       * END OF PROGRAM
       * ********************************************************************
                END
****** ******************************** Bottom of Data ********************************
```

Operands start after a space after instructions and are delimited by commas and brackets

Working with HLASM – CSECTs and DSECTs

- CSECT → CONTROL SECTION (HLASM directive)
  - A CSECT contains machine instructions to be run on the machine
- DSECT → DUMMY SECTION (HLASM directive)
  - Used to define the structure of data
- Both CSECT and DSECT are terminated with the end statement

```
MYPROG   CSECT                START OF CODE
    ...awesome assembler program goes here...
MYSTRUCT DSECT                START OF DATA STRUCTURE
    ...awesome data structure goes here...
         END                  END OF PROGRAM
```

Working with HLASM – Defining Data

- Data is defined via the DC and DS HLASM directives
- DC – Define Constant
  – Defines data and initialises it to a given value
- DS – Define Storage
  – Defines storage for data but does not give it a value
- e.g.

```
NUMBER1    DC     F'12'            DEFINE A FULLWORD WITH VALUE 12
NUMBER2    DC     H'3'             DEFINE A HALFWORD WITH VALUE 3
TOTAL      DS     H                DEFINE A HALFWORD
MYSTR      DC     C'HELLO WORLD'   DEFINE A SERIES OF CHARACTERS
MYHEX      DC     X'FFFF'          DEFINE A SERIES OF HEX CHARACTERS
```

Working with HLASM – Literals

- A literal is an inline definition of data used in an instruction but the space taken for the literal is in the nearest literal pool
- A literal pool collects all previous literals and reserves the space for them
- By default, HLASM produces an implicitly declared literal pool at the end of your CSECT
- To cause HLASM to produce a literal pool, use the LTORG directive

```
          L     1,=F'1'          LOAD REGISTER 1 WITH FULLWORD OF 1
          X     1,=H'2'          XOR REGISTER 1 WITH HALFWORD OF 2
     ...more awesome assembler code here...
          LTORG ,                THE LITERAL POOL IS CREATED
```

Introduction to Assembler Programming Exercise 1

Exercise 1 – A Solution

```
****** ***************************** Top of Data ******************
**********************************************************************
* SIMPLE HELLO WORLD PROGRAM
**********************************************************************
*
* MAIN PROGRAM STARTS HERE
*
EX1      CSECT
EX1      AMODE 31
EX1      RMODE 24
* USUAL PROGRAM SETUP    <- FIX THIS COMMENT
         STM   14,12,12(13)
         BALR  12,0
         USING *,12
*
* *******************************************************************
* WRITE YOUR CODE HERE
* MOVE THE DATA IN_STRING TO OUT_STRING
* HERE...
         MVC   OUT_STRING,IN_STRING
* *******************************************************************
*
         LA    5,WTO_AR
         WTO   TEXT=(5)
LMRET    LM    14,12,12(13)
*
* *******************************************************************
* WRITE YOUR CODE HERE
* THE RETURN CODE OF THE PROGRAM IS HANDED BACK IN REGISTER 15
* PROVIDE A RETURN CODE OF 15
* HERE...
         XR    15,15
* *******************************************************************
*
         BR    14
* *******************************************************************
* END OF PROGRAM
* *******************************************************************
IN_STRING  DC    C'HELLO WORLD!'
WTO_AR     DC    AL2(L'OUT_STRING)
OUT_STRING DS    CL(L'IN_STRING)
         LTORG ,
         END
```

Move comment to column 1

Use MVC to copy the data

Set register 15 to 0

Introduction to Assembler Programming
Addressing Data

Addressing Data

- There are 2 ways to access data for manipulation
  - Base-Displacement (and index) addressing
  - Relative addressing
- Relative addressing is a new form of addressing which calculates the data's relative position from the current PSW (in half-word increments)

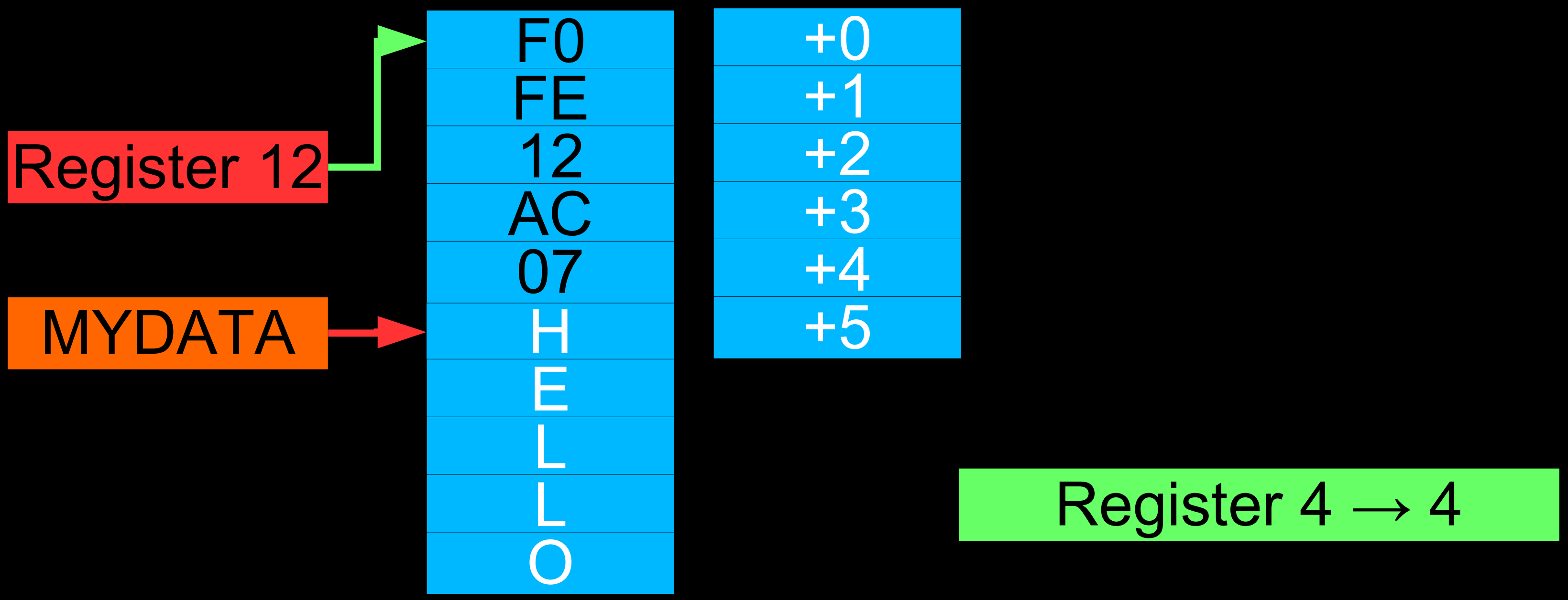
```
         LRL   1,NUMBER       LOAD RELATIVE REGISTER 1 WITH NUMBER
       ...more awesome assembler code here...
NUMBER DC    F'23'
```

Addressing Data - Base-Displacement-Index

- Base-Displacement(-index) addressing involves using a register as a pointer to memory – this is called the BASE register
- A displacement is usually between 0 and 4095 bytes allowing a single base register to address 4K of memory
- An index register is an additional register whose value is added to the base and displacement to address more memory
- Incrementing an index register allows the assembler programmer to cycle through an array whilst maintaining the same base-displacement
- Note that register 0 cannot be used as a base or index register
  - Register 0 used in this way means that the *value* 0 will be used as a base / index and NOT the contents of register 0
- Base, displacement and indexes are optionally specified on an instruction
  - Implicit default value for each is 0

Addressing Data - Base-Displacement-Index

- Address = BASE+INDEX+DISPLACEMENT

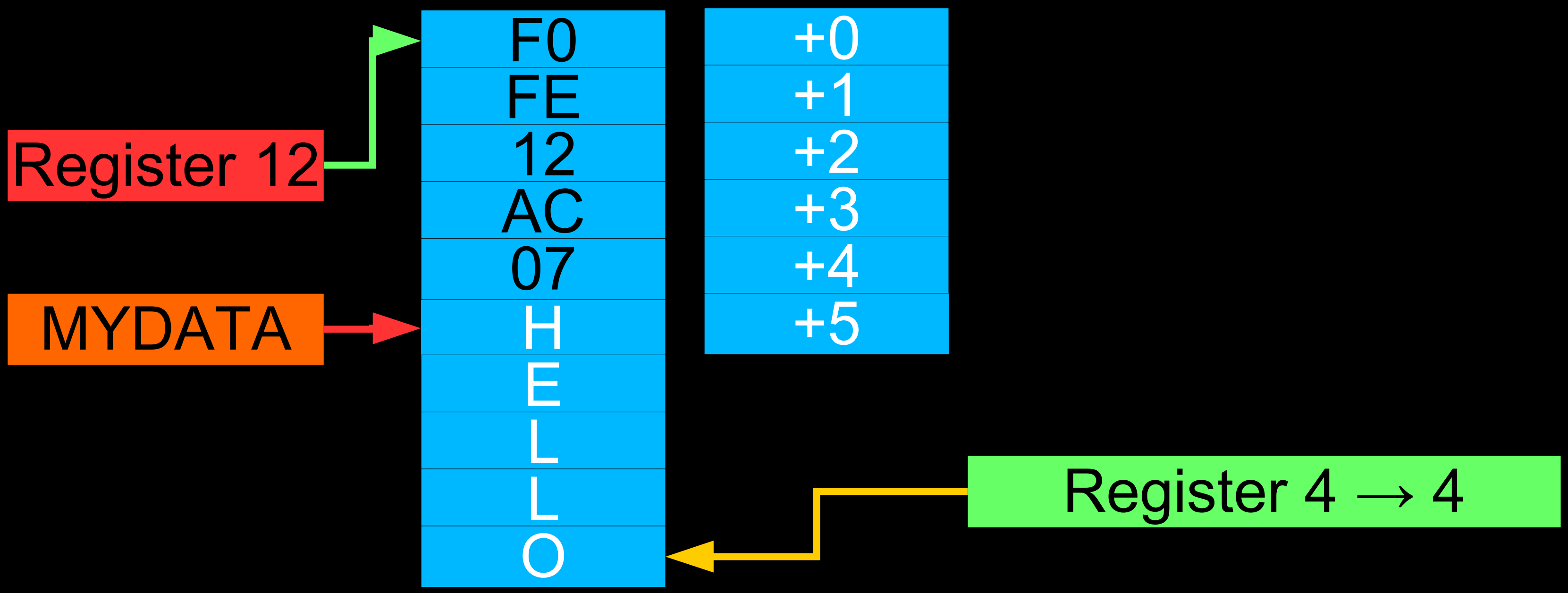


Addressing Data - Base-Displacement-Index

- Address of MYDATA = 5(0,12) → *displacement 5 + index 0 + base 12*

Addressing Data - Base-Displacement-Index

- Address of 'O' in 'HELLO' = 5(4,12) → *displacement 5 + index 4 + base 12*



Addressing Data – Loading addresses

- To load an address into a register, use the LOAD ADDRESS (LA) instruction

```
LA    1,DATA         LOAD ADDRESS OF DATA INTO REGISTER 1
```

- The LA instruction can be used to set a register to between 0 and 4095 by specifying a base and index register of 0 – these are automatically implicitly specified, e.g.

```
LA    1,12         base=0, index=0, displacement=12
```

- To store a 'O' in 'HELLO' in the previous example:

```
...some setup for REGISTER 12...
LA    4,4               LOAD ADDRESS 4 INTO REGISTER 4
L     3,=C'O'           LOAD CHARACTER 'O' INTO REGISTER 3
ST    3,MYDATA(4)       base=12, index=4, displacement=5
```

Introduction to Assembler Programming
Exercise 2

Exercise 2 – A Solution

```
*****************************************************************
* SIMPLE ADDRESSING LOOP PROGRAM
*****************************************************************
*
* MAIN PROGRAM STARTS HERE
*
EX2      CSECT
EX2      AMODE 31
EX2      RMODE 24
* USUAL PROGRAM SETUP
         STM   14,12,12(13)
         BALR  12,0
         USING *,12
*
* SAVE REGISTER 1 SOMEWHERE BECAUSE IT MAY BE USED BY WTO
         LR    3,1
*
         WTO   'HELLO'
         LA    5,WTO_AR            5 -> WTO BUFFER
*
* RESTORE THE SAVED VALUE TO REGISTER 1
         LR    1,3
*
         L     3,0(,1)             GET TO PARM LIST POINTER
*
* LOAD THE HALFWORD VALUE AT REGISTER 3 DISPLACEMENT 0 TO REGISTER 4
         LH    4,0(,3)
*
*
* LOAD THE ADDRESS AT REGISTER 3 DISPLACEMENT 2 TO REGISTER 3
         LA    3,2(,3)
*
*
* CHANGE THE WXYZ TO SPECIFY A DISPLACEMENT 0 AND BASE REGISTER 3 IN
* THE MVC INSTRUCTION BELOW.  NOTE THAT FOR THE MVC INSTRUCTION,
* THERE IS NO INDEX PARAMETER (UNLIKE IN LA)
*
LOOP     MVC   OUT_STRING(1),0(3)
         WTO   TEXT=(5)
         AHI   3,1                 BUMP 3 TO NEXT CHARACTER
         BCT   4,LOOP
LMRET    LM    14,12,12(13)
*
         XR    15,15
         BR    14
* ***************************************************************
* END OF PROGRAM
* ***************************************************************
WTO_AR     DC    H'1'
OUT_STRING DS    C
         LTORG ,
         END
```

Introduction to Assembler Programming
Branching

Branching

- Branching allows control flow in the program to move unsequentially
- Branches are performed via the BRANCH instructions
- Most branch instructions are *conditional* – i.e. they will pass control to the *branch target* if a condition is met otherwise control will continue sequentially
- The condition on which the branch will take place is called the CONDITION CODE (CC)
  - The CC is 2-bits stored in the PSW; thus the value is 0-3
  - Each instruction may (or may not) set the CC
- A branch instruction provides a *branch mask*
  - The *branch mask* instructs the processor that the branch will be taken if any of the bits in the CC match those in the branch mask
- Fortunately most code uses HLASM's branch mnemonics to provide a branch mask

Branching – Using HLASM's branch mnemonics

- B – Branch (unconditionally)
- BE – Branch on condition Equal
- BL – Branch on condition Less than
- BH – Branch on condition Higher than
- BNL – Branch Not Less
- BNH – Branch Not High
- BZ – Branch on Zero
- BNZ – Branch Not Zero
- There are also other branch mnemonics which HLASM provides

Branching – How does a branch mask work

- B – Branch (unconditionally)
  - This is translated to the BRANCH ON CONDITION (BC) instruction with a mask of 15

Condition Code	0	1	2	3
Mask value	8	4	2	1

- So, 15 → b'1111' → 8+4+2+1
- Thus the branch is taken if CC 0, 1, 2 or 3 is met, i.e. ALWAYS

Branching – How does a branch mask work

- BE – Branch on Equal
  - This is translated to the BRANCH ON CONDITION (BC) instruction with a mask of 8

Condition Code	0	1	2	3
Mask value	8	4	2	1

- So, 8 → b'1000' → 8
- Thus the branch is taken if CC 0 is met

Branching – Using a branch to form an *if* statement

```
     L   1,NUMBER   LOAD NUMBER INTO REGISTER 1
     LTR 1,1        LOAD REGISTER 1 INTO REGISTER 1 AND SET CC
     BNZ NONZERO    BRANCH TO 'NONZERO' IF REGISTER 1 IS NOT ZERO
     ...code where register 1 is zero goes here...
         B  COMMONCODE     REJOIN COMMON CODE
NONZERO DS 0H
     ...code where register 1 is non-zero goes here...
COMMONCODE DS 0H
```

Branching – Using a branch to form an *if* statement

```
if(register_1==0){
   //Code for register_1 being 0 goes here
}
else{
   //Code for register_1 being non-zero goes here
}

//Common code goes here
```

Introduction to Assembler Programming
Arithmetic

Arithmetic

- Arithmetic is performed in a wide variety ways on z/Architecture
  - Fixed point arithmetic (including logical) ← performed in GPRs
  - Packed Decimal arithmetic ← performed in memory
  - Binary and Hexadecimal Floating point arithmetic ← performed in FPRs
- Fixed point arithmetic
  - Normal arithmetic, e.g. adding the contents of 2 numbers together
  - Fixed point arithmetic is signed with numbers being stored in 2's complement form
  - Logical fixed point arithmetic is unsigned, i.e. both numbers are positive
- Pack Decimal arithmetic
  - Performed in memory
  - Numbers are in packed decimal format

Arithmetic – Fixed point arithmetic operations

- ADD instructions

```
AR    1,2          ADD REGISTER 2 TO REGISTER 1 (32-BIT SIGNED)
ALR   1,2          ADD REGISTER 2 TO REGISTER 1 (32-BIT LOGICAL)
A     1,NUMBER     ADD NUMBER TO REGISTER 1 (32-BIT SIGNED)
AL    1,NUMBER     ADD NUMBER TO REGISTER 1 (32-BIT LOGICAL)
AFI   1,37         ADD 37 TO REGISTER 1 (IMMEDIATE)
```

- Note that for immediate instructions, the operand is included in the instruction rather than needing to be obtained from memory
- At the end of the addition, the CC is updated (as specified in POPs)
  - CC → 0 → Result is 0; no overflow
  - CC → 1 → Result less than 0; no overflow
  - CC → 2 → Result greater than 0; no overflow
  - CC → 3 → Overflow occurred

Arithmetic – Fixed point arithmetic operations

- SUBTRACT instructions

```
SR    1,2           SUBTRACT REGISTER 2 TO REGISTER 1 (SIGNED)
SLR   1,2           SUBTRACT REGISTER 2 TO REGISTER 1 (LOGICAL)
S     1,NUMBER      SUBTRACT NUMBER TO REGISTER 1 (SIGNED)
SL    1,NUMBER      SUBTRACT NUMBER TO REGISTER 1 (LOGICAL)
AFI   1,-37         ADD -37 TO REGISTER 1 (IMMEDIATE)
```

- At the end of the subtraction, the CC is updated (as specified in POPs)
  - CC → 0 → Result is 0; no overflow
  - CC → 1 → Result less than 0; no overflow
  - CC → 2 → Result greater than 0; no overflow
  - CC → 3 → Overflow occurred

Arithmetic – Fixed point arithmetic operations

- MULTIPLY instructions

```
MR    2,7           MULTIPLY REGISTER 2 BY REGISTER 7
M     2,NUMBER      MULTIPLY REGISTER 2 BY NUMBER
```

- The first operand is an even-odd pair – the result of the MULTIPLY is stored in:
  - The even register (of the pair) – top 32-bits of result
  - The odd register (of the pair) – bottom 32-bits of the result
- At the end of the multiplication, the CC is UNCHANGED

Arithmetic – Fixed point arithmetic operations

- DIVIDE instructions

```
DR    2,7            DIVIDE REGISTER 2 BY REGISTER 7
D     2,NUMBER       DIVIDE REGISTER 2 BY NUMBER
```

- The first operand is an even-odd pair
  - The even register (of the pair) – top 32-bits of dividend
  - The odd register (of the pair) – bottom 32-bits of the dividend
- The result is stored in the first operand:
  - The quotient is stored in the odd register of the pair
  - The remainder in the even register of the pair
- At the end of the division, the CC is UNCHANGED

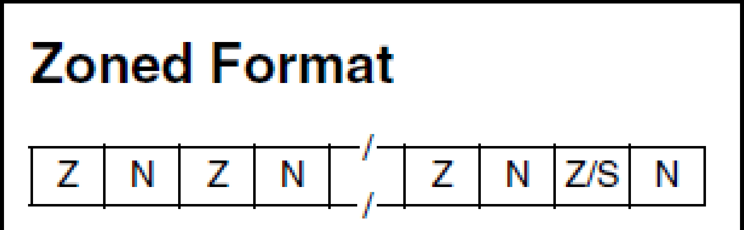
Arithmetic – Zoned and Packed Decimal

- The computations we have looked at so far have been with binary data
- This is not always satisfactory, especially when financial calculations are required
- For example, decimal percentages are inaccurate in binary (try long division on 1/1010 = 1/10102 = .000110011...)

- Lets look at decimal data types and instructions
- There are two decimal data formats
  - Zoned Decimal – good for printing and displaying
  - Packed Decimal – good for decimal arithmetic

Arithmetic – Zoned and Packed Decimal

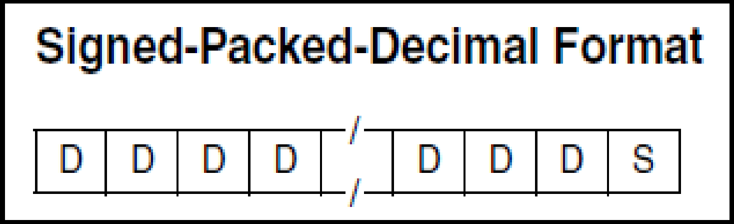
- In the zoned format, the rightmost four bits of a byte are called the numeric bits (N) and normally of a code representing a decimal digit. The leftmost four bits of a byte are called the zone bits (Z), except for the rightmost byte of a decimal operand, where these bits may be treated either as a zone or as a sign (S).



1111	0001	1111	1100	1111	0000	1111	0111
F	1	F	9	F	0	F	7

Arithmetic – Zoned and Packed Decimal

- In the signed-packed-decimal format, each byte contains two decimal digits (D), except for the rightmost byte, which contains a sign (S) to the right of a decimal digit.



0000	0000	0000	0001	1001	0000	1111	1111
0	0	0	1	9	0	7	F

Arithmetic – Zoned and Packed Decimal

- The sign for both Zoned Decimal and Packed Decimal is
  - C, A, F, or E are all +ve. C is preferred
  - D or B are -ve. D is preferred

One hundred and and eleven +ve is F1F1C1 – C being +ve sign

One hundred and and eleven -ve is F1F1D1 – D being -ve sign

Beaware... They would pring as 11A and 11J !

C1 is the character A and D1 is J

Arithmetic – Packed Decimal arithmetic operations

Decimal instructions

```
AP    a,b          ADD b to a
CP    a,b          COMPARE a to b
DP    a,b          DIVIDE a by b
MP    a,b          MULTIPLY a by b
SP    a,b          SUBTRACT b from a
ZAP   a,b          ZEROISE a and then add b
```

- At the end of the subtraction, the CC is updated (as specified in POPs)
  - CC → 0 → Result is 0; no overflow
  - CC → 1 → Result less than 0; no overflow
  - CC → 2 → Result greater than 0; no overflow
  - CC → 3 → Overflow occurred

Introduction to Equates

Equates

- You can define symbols as equates
- Use the EQU instruction to
  - Assign single absolute values to symbols
  - Assign the values of previously defined symbols or expressions to new symbols
  - Compute expressions whose values are unknown at coding time or difficult to calculate.

Equates

- Register equates examples

  R00 EQU 0,,,GR32

  R00 is the symbol

  0 is the absolute value asigned

  GR32 is the assembler type value

  GR00 EQU 0,,,GR64

  GR00 is the symbol

  0 is the absolute value assigned

  GR64 is the assembler type

Equates

Why use Assembler types ?

- Assembler option TYPECHECK(...,REGISTER)
  – Specifies that the assembler performs type checking of register fields of machine instruction operands

Equates

typeme.asm - Notepad

File Edit Format View Help

```
*/* copyright IBM 2013
*/* TYPEME   - Sample program
TYPEME CSECT
TYPEME AMODE 31
r01  EQU   1,,,,GR32
r02  EQU   2,,,,GR32
gr01 EQU   1,,,,GR64
gr02 EQU   2,,,,GR64
        LR  r01,r02                  copy 32 bit register
        LGR gr01,gr02                copy 64 bit register
        LR  gr01,gr02                copy 32 bit register
        LGR r01,r02                  copy 64 bit register
        END
```

Equates

```
00000000                  00000000 0000000C      3 TYPEME CSECT
                                                 4 TYPEME AMODE 31
                          00000001               5 r01  EQU   1,,,,GR32
                          00000002               6 r02  EQU   2,,,,GR32
                          00000001               7 gr01 EQU   1,,,,GR64
                          00000002               8 gr02 EQU   2,,,,GR64
00000000 1812                                    9           LR  r01,r02
00000002 B904 0012                              10           LGR gr01,gr02
00000006 1812                                   11           LR  gr01,gr02
** ASMA323W Symbol gr01 has incompatible type with general register field
** ASMA323W Symbol gr02 has incompatible type with general register field
** ASMA435I Record 11 in SMORSA.BOSTON.ASM.SOURCE(TYPEME) on volume: 37P003
00000008 B904 0012                              12           LGR r01,r02
** ASMA323W Symbol r01 has incompatible type with general register field
** ASMA323W Symbol r02 has incompatible type with general register field
** ASMA435I Record 12 in SMORSA.BOSTON.ASM.SOURCE(TYPEME) on volume: 37P003
                                                13           END
                               Ordinary Symbol and Literal Cross Reference
Symbol   Length   Value     Id        R Type Asm  Program   Defn References
gr01          1 00000001 00000004 A    U  GR64                 7   10M   11M
gr02          1 00000002 00000004 A    U  GR64                 8   10    11
r01           1 00000001 00000004 A    U  GR32                 5    9M   12M
r02           1 00000002 00000004 A    U  GR32                 6    9    12
TYPEME        1 00000000 00000004      J                       3    4
                                   General Purpose Register Cross Reference
```

Introduction to Assembler Programming
Looping

Looping

- A simple loop is formed by using a counter, a comparison and a branch, e.g.

```
       LA    2,0              INITIALISE COUNTER REGISTER TO 0
MYLOOP AHI   2,1              INCREMENT COUNTER
       WTO   'HELLO'          SAY HELLO
       CHI   2,10             IS THE COUNTER 10?
       BL    MYLOOP           IF IT'S LESS THAN 10, GO TO MYLOOP
```

- That's simple – but there's a better way – use BRANCH ON COUNT (BCT)

```
       LA    2,10             INITIALISE COUNTER REGISTER TO 10
MYLOOP WTO   'HELLO'
       BCT   2,MYLOOP         SUBTRACTS, COMPARES & BRANCHES
```

- There are other variants of the BCT instruction, e.g. BCTR, BXH etc...

Introduction to Assembler Programming Exercise 3

Exercise 3 – A Solution

```
**********************************************************************
* SIMPLE ADDRESSING LOOP PROGRAM
**********************************************************************
*
* MAIN PROGRAM STARTS HERE
*
EX3      CSECT
EX3      AMODE 31
EX3      RMODE 24
* USUAL PROGRAM SETUP
         STM   14,12,12(13)
         BALR  12,0
         USING *,12
         LA    5,0          INITIALISE INDEX REGISTER
         LA    6,0          INITIALISE ACCUMULATOR
LOOP     L     3,A_ARR(5)   LOAD ARRAY A ELEMENT
         L     4,B_ARR(5)   LOAD ARRAY B ELEMENT
         MR    2,4          MULTIPLY RESULT
         AR    6,3          ADD RESULT TO ACCUMULATOR
         AHI   5,4
         CHI   5,16
         BL    LOOP         BRANCH IF NOT AT END OF ARRAY
         ST    6,RESULT     STORE FINAL RESULT
LMRET    LM    14,12,12(13)
*
         XR    15,15
         LRL   15,RESULT
         BR    14
* ********************************************************************
* END OF PROGRAM
* ********************************************************************
A_ARR     DC    A(12,3,12,10)
B_ARR     DC    A(4,7,9,8)
RESULT    DC    F'0'
         LTORG ,
         END
```

Introduction to Assembler Programming
Calling conventions

Calling Conventions

- A calling convention is a convention used between programs and subroutines to call each other
- The calling convention is not enforced, but if it is disregarded undesirable and unpredictable results may occur
- In general, when programming in assembler, the *caller* will provide a *save area* and the *called* program or routine will save all GPRs into that save area.
- The subroutine will then execute its code
- To return control to the caller, the subroutine will typically:
  - Set a return code in a register
  - Prepare the register on which it should branch back on
  - Restore all other registers
  - Branch back

Calling Conventions – Typical register usage on z/OS

- Although free to do as they please, most assembler programs on z/OS use the following register convention during initialisation
  - Register 1 → parameter list pointer
  - Register 13 → pointer to register save area provided by caller
  - Register 14 → return address
  - Register 15 → address of subroutine
- Once the registers are saved, the called subroutine will:
  - Update register 13 to point to a new save area (so that it can call other programs / routines)
  - Establish register 12 as a base register for the program
- Upon termination, the called subroutine will:
  - Set a return code in register 15
  - Restore registers 14,0,1,...,12 from the save area pointed to by register 13
  - Restore register 13 to the value it was previously
  - Branch back on register 14

Calling a subroutine in code

```
MAINLINE CSECT

..code
..call internal subroutine CALC
..more code
..RETURN

CALC  DC 0H'0'  Internal routine
..subroutine code
..
..return

END
```

Calling a subroutine in code – Going in...

- The caller calls the subroutine

```
         LA    1,PARAMS          POINT TO PARAMETERS
         LA    15,SUB1           LOAD ADDRESS OF SUBROUTINE
         BALR  14,15             BRANCH TO REGISTER 15 AND SAVE RETURN
*                                IN REGISTER 14
         LTR   15,15             CHECKS RETURN CODE 0?
         ...caller code continues here...
```

Calling a subroutine in code – Going in...

- The subroutine saves the caller's registers and establishes a base register

```
STM    14,12,12(13)    STORE REGISTERS
LR     12,15           GET ENTRY ADDRESS
...subroutine code continues here...
```

Calling a subroutine in code – Getting out...

- The subroutine restores the caller's registers, sets the return code and branches back

```
         LM    14,12,12(13)    RESTORE REGISTERS
         XR    15,15           SET RETURN CODE 0
         BR    14              BRANCH BACK TO CALLER
```

- Due to this calling convention, during epilog and prologue of a program or subroutine or when calling or having control returned from a program or subroutine, avoid using registers 0, 1, 12, 13, 14, 15
- z/OS services, typically will use registers 0, 1, 14, 15
- Not sure which registers are used by a service?
  - The manuals explain in detail

Calling a subroutine in code – Going in...

- The caller calls the subroutine

```
         LA    1,PARAMS       POINT TO PARAMETERS
         LA    15,SUB1        LOAD ADDRESS OF SUBROUTINE
         BALR  14,15          BRANCH TO REGISTER 15 AND SAVE RETURN
*                             IN REGISTER 14
         LTR   15,15          CHECKS RETURN CODE 0?
         ...caller code continues here...
```

- ...but do I have to write this code ?
  - NO – use the supplied z/OS macros...
    - CALL macro
      - Documented in

Calling a subroutine in code – Going in...

- The caller calls the subroutine

```
         LA    1,PARAMS        POINT TO PARAMETERS
         LA    15,SUB1         LOAD ADDRESS OF SUBROUTINE
         BALR  14,15           BRANCH TO REGISTER 15 AND SAVE RETURN
*                              IN REGISTER 14
         LTR   15,15           CHECKS RETURN CODE 0?
         ...caller code continues here...
```

- The subroutine saves the caller's registers and establishes a base register

```
         STM   14,12,12(13)    STORE REGISTERS
         LR    12,15           GET ENTRY ADDRESS
         ...subroutine code continues here...
```

Calling a subroutine in code – Getting out...

- The subroutine restores the caller's registers, sets the return code and branches back

```
         LM    14,12,12(13)    RESTORE REGISTERS
         XR    15,15           SET RETURN CODE 0
         BR    14              BRANCH BACK TO CALLER
```

- Due to this calling convention, during epilog and prologue of a program or subroutine or when calling or having control returned from a program or subroutine, avoid using registers 0, 1, 12, 13, 14, 15
- z/OS services, typically will use registers 0, 1, 14, 15
- Not sure which registers are used by a service?
  - The manuals explain in detail

Introduction to Assembler Programming
How to read Principles of Operation

Reading POPs

- Principles of Operation (better known as POPs) is the z/Architecture manual
- It explains everything from system organisation and memory, to instructions and number formats
- It provides a useful set of appendices some of which provide good detailed examples of instruction use, including programming techniques
- The vast majority of POPs is instruction descriptions

Reading POPs – Understanding Instruction Descriptions

- Each instruction is described in exact detail including:
  - The instruction's syntax
  - Machine code
  - Operation
  - Condition code settings
  - Programming Exceptions
- There are 2 forms of syntax provided for each instruction
  - The syntax for the assembler, i.e. what is written in your assembler program
  - The machine code for the instruction, i.e. the binary code run on the processor
- The instruction's machine code is grouped together with other instructions which share a similar machine code layout called an *instruction format*

Reading POPs – Instruction Formats

- The instruction format used, is generally related to
  - The assembler syntax used to code the instruction
  - The operation that the instruction performs
- Instructions that we've used have had the following formats:
  - RR – Register-Register – this form usually manipulates registers, e.g. LR, MR, DR
  - RX – Register, Index, base displacement – usually moving data between memory and registers, e.g. L, LA, ST, A, X, S, D, M
  - SS – Storage-Storage – acts on data in memory, e.g. MVC

Reading POPs – Instruction Formats – RR – LR instruction

LOAD

Register-and-register formats:

LR R_1,R_2 [RR]

'18'	R_1	R_2
0	8	12 15

LGR R_1,R_2 [RRE]

'B904'	////////	R_1	R_2
0	16	24	28 31

LGFR R_1,R_2 [RRE]

'B914'	////////	R_1	R_2
0	16	24	28 31

Reading POPs – Instruction Formats – RX – L instruction

Register-and-storage formats:

L $R_1,D_2(X_2,B_2)$ [RX-a]

'58'	R_1	X_2	B_2	D_2
0	8	12	16	20 … 31

Introduction to Assembler Programming
Exercise 4

Exercise 4 – A solution

```
* ***********************************************************************
* WRITE YOUR CODE HERE
* CALL THE SUBROUTINE MYSUB
         LA    1,BUFLEN
         LA    2,INBUF
         LA    3,OUTBUF
         LA    15,MYSUB
         BALR  14,15
* ***********************************************************************
*
         LA    5,WTO_AR
         WTO   TEXT=(5)
LMRET    LM    14,12,12(13)
         XR    15,15
         BR    14
* ***********************************************************************
* MY SUBROUTINE
* SPECIFICATION:
*     THIS SUBROUTINE SHOULD COPY THE AMOUNT OF BYTES SPECIFIED IN
*     REGISTER 1 AT THE ADDRESS SPECIFIED IN REGISTER 2 TO THE BUFFER
*     SPECIFIED IN REGISTER 3
* INPUTS:
*     REGISTER 1  -> LENGTH OF DATA TO BE COPIED
*     REGISTER 2  -> POINTER TO INPUT BUFFER
*     REGISTER 3  -> POINTER TO OUTPUT BUFFER
*     REGISTER 14 -> RETURN ADDRESS
* OUTPUTS:
*     ALL REGISTERS ARE RESTORED EXCEPT FOR REGISTER 14
* ***********************************************************************
MYSUB STM  0,15,MYSAVEAREA
      LR   0,2
      LR   5,1
      LA   4,OUTBUF
      MVCL 4,0
      LM   0,15,MYSAVEAREA
      BR   14
* ***********************************************************************
```