Introduction to Assembler Programming
Sessions 17690, 17691

Richard Cebula (riccebu@uk.ibm.com) IBM HLASM
Who am I?

- Richard Cebula – HLASM, IBM Hursley, UK
- riccebu@uk.ibm.com

Develop and support the following products:
- HLASM
- SuperC
- XREF
- IDF
- DISASM
- Structured Programming Macros
Audience

- This course aims to provide the grounding knowledge for programming in assembler
- The audience should have a basic understanding of computers
- The audience should be new to the world of z Systems Assembler programming
- At the end of this course the attendee should be able to:
  - Understand the basics of assembler programming on z Systems
  - Understand a variety of simple machine instructions
  - Understand how to Assemble, Bind and run simple assembler programs
Introduction to Assembler Programming

- Why program in assembler?
- Computer Organisation – Overview of z/Architecture
- Assemblers, Compilers, Binders – Building programs on z Systems
- Working with the High Level Assembler (HLASM)
  - Using HLASM to assemble your program
  - Syntax
  - Machine vs Assembler instructions
- Programming in Assembler
  - Moving data around
    • Loading, storing and moving data
  - Manipulating Data
    • Logical operations
    • Arithmetic
  - Making Decisions
    • Comparing
    • Branching
Introduction to Assembler Programming

- Programming in Assembler
  - Forming High Level Language constructs
    - If...then...else
    - Looping
  - Addressing Data and some of its subtleties
  - Basics of Calling Conventions
  - Reading Principles of Operation

- Reading the HLASM listing

- The PSW and an introduction to Debugging Assembler Programs
Why program in assembler?
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 older 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

- WYWIWYG – What you write is what you get – even if you think you wrote something else...
Why program in assembler? - I thought this was the 21st century...

- Well there are a number of good reasons to program in assembler still:
  - Precise machine control – trust me...your compiler doesn’t do what you think it does...
  - Precise definition of data – we have more data types than other languages
  - Self-writing code – the world's best macro facility
  - When someone's program goes wrong – finding the cause of the problem is much easier in assembler
  - Assembler is high-tech – no need to wait for compilers to catch up to what your chip can do
Computer Organisation
The main parts of a computer when discussing programming are:

- Processor – z Systems is a multiprocessor computer
- Storage – (don't call it RAM on an EC12 and later it's RAIM!!)
- Disks (more often referred to as DASD in z Systems)

Programs are stored on disk since disks are non-volatile media, i.e. they do not lose their contents when the computer is not running.

A program is fetched from disk and placed into storage from where it can be executed by the processor.

The processor is the brain of the computer and is responsible for actually executing programs.

Operations inside a computer such as loading programs from disk are performed by a piece of software called an Operating System (OS).

z Systems have 5 operating systems available z/OS, z/VM, z/VSE, z/TPF and z/Linux.
Computer Organisation

- DASD (Disk)
- Processor
- Processor
- Processor

Storage
Computer Organisation

- The closer that data gets to the processor, the quicker it is to access the data both in terms of type of storage used and also due to the physical distance to the processor.

- In order to improve the speed at which the program is accessed from storage, a special form of storage called cache is built into the processor.

- Data and instructions are fetched from storage into the cache on the processor and then accessed from there.

- Depending on how often the data requires to be accessed, how much data is to be accessed and whether the data needs to be shared between different processors or not, depends on which cache level the data is placed into on the processor with level 1 being the smallest and fastest and level 4 being the slowest but largest.

- The implementation of cache is dependent on not only the processor architecture, e.g. Intel vs z/Architecture vs ARM etc. but also the model of processor itself, e.g. EC12 cache structure is different to z196.
Computer Organisation – z13

- Each processor in z Systems is split into a number of processor cores called PUs (processing units).
- Each PU (core) has its own L1 (96KB I, 128KB D) and L2 (2MB I, 2MB D) instruction and data caches.
- Each processor chip contains up to 8 processing cores which share a 64MB L3 cache.
- PU chips and storage control (SC) chips are packaged as single chip modules (SCMs). 3 PU chips and 1 SC chip form a CPC drawer node. Each SC has a 480MB L4 cache.
- 2 nodes are contained on a CPC drawer. Each system can have up to 4 drawers which contain all the storage for the system.
- All of the highly complex cache structure is transparent to the programmer.
  - Experienced programmers will often change their code in order to improve cache performance.
- Depending on how the machine is configured, each processor may run a different level of microcode which governs how it functions changing the PU into either a CP, zIIP or IFL.
- Additionally, z Systems has specialist I/O processors called System Assist Processors (SAP) used to send signals to devices attached to the machine.
Computer Organisation – z/Architecture
In order for data to be manipulated by the processor, it needs to be loaded into a register.

A register is the fastest storage available to the processor and is located inside each processor core.

In z/Architecture, each processor core has:
- 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)
- 32 128-bit Vector Registers (VRs)*
- 1 Program Status Word (PSW)

Note that all registers are numbered 0-15 (or 0-31) in machine instructions – the instruction itself determines which type of register is being used.

z/Architecture – the processor architecture used for all z Systems Mainframes.

Processor specifications vary
- Processor level – the physical (or virtual) chip used
- Architecture level – the instruction specification of a chip

* Vector registers 0-15 bits 0-63 are mapped over the FPRs.
Computer Organisation – z/Architecture

- z/Architecture is a big-endian, 64-bit, rich CISC processor architecture
  - Big-endian
    - Data is organised such that the most significant byte of a piece of data is stored in the lowest address of memory
  - 64-bit
    - The size of general purpose registers is 64-bits in length
  - CISC
    - Complex Instruction Set Computer
    - A single instruction comprises a number of micro-instructions which are executed by the processor
    - This scheme allows for a single machine instruction to perform a number of complex tasks

- For historical reasons, the size of data in z/Architecture is measured as:
  - 4 bits = 1 nibble
  - 8 bits = 2 nibbles = 1 byte
  - 16 bits = 2 bytes = a halfword
  - 32 bits = 4 bytes = 2 halfwords = a word
  - 64 bits = 8 bytes = 2 words = a doubleword
  - 128 bits = 16 bytes = 2 doublewords = a quadword
Computer Organisation – Understanding Registers

- GPRs – 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, binary, decimal and hexadecimal floating-point arithmetic
  - Do not confuse decimal floating-point with packed decimal arithmetic – the latter is performed in storage not in registers
- VRs – used for SIMD (Single Instruction Multiple Data) operations including integer, string, floating-point and general operations
- 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 are sometimes operated on in pairs by certain instructions
  - 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)
Assemblers, Compilers and Binders
Building programs on z Systems
Assemblers, Compilers and Binders

- Typing in the code for a computer program will not mean that the computer can load and then execute the program

- The written code must be changed into binary form – this is done for the programmer by a compiler or assembler
  - Compilers are used for HLLs and often attempt to optimise the code written by the programmer
  - Assemblers are used for assembly language and do NOT optimise any code written by the programmer – assembler is a WYWIWYG language – What You Write Is What You Get

- Each language will require its own compiler / assembler to process it and change it into binary code

- Cross compiling is where a compiler runs on one machine architecture and produces machine code for another architecture
Assemblers, Compilers and Binders

- The machine code produced by the assembler is called **object code**

- It is the job of a *binder* (sometimes called a *linker*) to create a complete program which can then be loaded by the operating system's *loader* into storage and then executed by the processor

- The binder works by ordering a set of objects and resolving any references between them
  - The bound object on z/OS is called a *program object* (for GOFF format) or a *load module* (for OBJ format)
  - Program objects and load modules vary in their capability – it is recommended that new programs use the GOFF format

- Some references are unable to be resolved by the binder at bind time and can only be resolved when the program is loaded into storage by the operating system's loader
  - In order to do this, the binder creates a list of these references and notes their location in the program object / load module
  - Examples of such unresolvable references are various operating system services and shared libraries

- The terms “load module” and “program object” are used interchangeably in this presentation
Assemblers, Compilers and Binders – 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
- A very powerful macro programming facility
- Structured programming
Assemblers, Compilers and Binders – Working with HLASM

- HLASM produces 2 primary outputs
  - OBJECT DECKS – this is the object code that is used as input to binding
  - Listing – this shows any errors, all diagnostics and human readable output from the assemble phase

- The binder produces 2 primary 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
Assemblers, Compilers and Binders – Working with HLASM
Assemblers, Compilers and Binders – Working with HLASM

Diagram:
- SOURCE
- COPYBOOKS
- HLASM
- OBJECTS
- LISTING
- System Libraries
- Binder
- LINK MAP
- PROGRAM
Working with the High Level Assembler
HLASM
Working with the High Level Assembler HLASM

- These slides explain about using HLASM specifically for z/OS
  - The same principles apply on other platforms although HLASM is started differently, e.g. different command lines on z/VM and z/Linux, VSE JCL for z/VSE

- HLASM is started on z/OS via JCL and is shipped with some JCL PROCs to make using HLASM easier
  - ASMAC – Assembles a program
  - ASMACG – Assembles a program and invokes the loader to bind, load and execute the program (no load module is retained)
  - ASMACL – Assembles and invokes the binder to bind the program producing a load module
  - ASMACLG – Assembles and binds the program then runs the produced load module
The following JCL invokes ASMACL to assemble and bind the program into a load module:

```plaintext
// SET OPTLIB=&SYSUID..AOPT
// SET SRCLIB=&SYSUID..SOURCE
// SET LSTING=&SYSUID..LISTINGS
// SET LODLIB=&SYSUID..LOAD
//*
// SET PRGNM=MYPROG
//*
//ASMMSAMP EXEC ASMACL,PARM.C=(OBJ,ADATA)
//C.ASMAOPT  DD DSN=&OPTLIB,DISP=SHR
//C.SYSIN    DD DSN=&SRCLIB(&PRGNM),DISP=SHR
//C.SYSPRINT DD DSN=&LSTING(&PRGNM),DISP=OLD
//L.SYSLMOD  DD DSN=&LODLIB(&PRGNM),DISP=SHR
```
Working with the High Level Assembler HLASM

- The ASMACL JCL PROC has two steps:
  - C – Assemble the program
  - L – Bind the program
  - Using the notation \textit{step.ddname} allows multiple DDNAMEs to be specified for the JCL steps

- DDNAMEs used by the Assemble step (C)
  - SYSIN → specifies the program's source
  - SYSLIN → specifies copybooks / macro libraries
  - SYSPRINT → specifies the destination of the program's \textit{listing}
  - ASMAOPT → specifies a data set which contains assembly options*
  - PARM → specifies some options for HLASM*

- DDNAMEs used by the Bind step (L)
  - SYSLMOD → specifies the destination of the bound load module

- * Options may be specified in up to 6 different locations when assembling a program – allowing for a hierarchy of options to be specified.
Working with HLASM – Assembling and Binding a program
Working with HLASM – Assembling and Binding a program

SOURCE -> COPYBOOKS -> HLASM -> OBJECTS -> LISTING

C.SYSIN -> C.SYSLIB

ASMAOPT -> C.ASMAOPT

C.SYSLIB

C.SYSLIN

C.SYSPRINT

C.ASMAOPT
Working with HLASM – Examining the listing

- The program listing produced by HLASM is an invaluable source of information for the programmer.
- It lists in detail everything that HLASM has done when assembling a program including:
  - All the options that were used when the program was assembled
  - Any external references that the program produces or relies upon in the External Symbol Dictionary (ESD)
  - Each line of source code and the machine code that was produced
  - Any symbols in the code that require relocation in the Relocation Dictionary (RLD)
  - A summary of all lines in error in the program
  - A list of all data sets used
- The assembler uses the following return codes whenever it issues a message:
  - 0 – Success / Information
  - 2 – Notice – A condition which may need correcting – program appears to be correct
  - 4 – Warning – Program may not work as expected
  - 8 – Error – Error in program
  - 12 – Severe error – Unlikely that the program assembles as expected or runs
  - 16 – Critical – Unlikely that the program runs
  - 20 – Unrecoverable – Unable to continue
Working with HLASM – A look at syntax

```assembly
/* File Edit Edit_Settings Menu Utilities Compilers Test Help */

VIEW
Command ===>

Columns 00001 00080
Scroll ===>

******************************************************************************
000001 ******************************************************************************
000002  * SIMPLE DUMMY EXIT FOR HLASM
000003 ******************************************************************************
000004  *
000005  * MAIN PROGRAM STARTS HERE
000006  *
000007  LARLOAD CSECT
000008  LARLOAD AMODE 31
000009  LARLOAD RMODE 24
000010  * USUAL PROGRAM SETUP
000011    STM 14,12,12(13)
000012    BALR 12,0       GET THE CURRENT ADDRESS
000013    USING *,12       USE 12 AS THE BASE REGISTER
000014    L 1,=F'12'
000015  LMRET LM 14,12,12(13)
000016    XR 15,15
000017    BR 14
000018  *
000019  * END OF PROGRAM
000020  *
000021    END

******************************************************************************

```

Top of Data

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

Labels start in column 1
Instructions start after column 1 or a label
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
  ```
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
```
There are two distinct types of instruction accepted by HLASM – Machine instructions and Assembler instructions.

Machine instructions are changed into machine code by the assembler – these are the instructions that will be used when your program is executed.

Assembler instructions are instructions which affect the behavior of the assembler itself as it is assembling a program.

Some assembler instructions such as DC will cause the resultant program to contain certain pieces of data although the majority of assembler instructions do not.

The assembler also includes a macro language – macros are code which are used to generate other code programmatically and therefore can shorten development time.
Introduction to Assembler Programming

Moving Data
Moving Data

- Moving data is the most common operation that ANY computer program performs

- Before any computations can be performed, data must be moved to the correct places
  - Data is moved to the processor (from disk, memory, networks, devices etc)
  - Data is manipulated by the processor
  - The result is stored somewhere (back to disk, memory, networks, devices etc)

- Data is LOADed into the processor's registers via LOAD instructions

- Data is STOREd to memory via STORE instructions
Moving Data – Loading from Register to Register

- The LOAD REGISTER (LR) instruction is used to load the value stored in one register to another.
  
  \[ \text{LR 1,2} \quad \text{LOAD REGISTER 2 INTO REGISTER 1 (32-BITS)} \]

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

- The instruction has a 64-bit variant LOAD GRANDE REGISTER (LGR).
  
  \[ \text{LGR 1,2} \quad \text{LOAD REGISTER 2 INTO REGISTER 1 (64-BITS)} \]

- The instruction has a 16-bit variant LOAD HALFWORD REGISTER.
  
  \[ \text{LHR 1,2} \quad \text{LOAD REGISTER 2 INTO REGISTER 1 (16-BITS)} \]
Moving Data – Loading from Memory to Register

- The LOAD (L) instruction is used to load the value stored in memory to a register
  
  \[
  \text{L 1,MY_NUM} \quad \text{LOAD REGISTER 1 WITH THE VALUE MY_NUM (32-BITS)}
  \]

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

- The instruction has a 64-bit variant LOAD GRANDE (LG)
  
  \[
  \text{LG 1,MY_NUM} \quad \text{LOAD REGISTER 1 WITH THE VALUE MY_NUM (64-BITS)}
  \]

- The instruction has a 16-bit variant LOAD HALFWORD
  
  \[
  \text{LH 1,MY_NUM} \quad \text{LOAD MY_NUM INTO REGISTER 1 (16-BITS)}
  \]
Moving Data – Storing from a Register to Memory

- The STORE (ST) instruction is used to store the value in a register to memory
  
  ```assembly
  ST 1,MY_NUM     STORE REGISTER 1 TO MY_NUM (32-BITS)
  ```

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

- The instruction has a 64-bit variant STORE GRANDE (STG)
  
  ```assembly
  STG 1,MY_NUM    STORE REGISTER 1 TO MY_NUM (64-BITS)
  ```

- The instruction has a 16-bit variant STORE HALFWORD
  
  ```assembly
  STH 1,MY_NUM    STORE REGISTER 1 TO MY_NUM (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 MY_OUTPUT, MY_INPUT  MOVE MY_INPUT TO MY_OUTPUT

- The MVC instruction can move up to 256B from one area of memory to another
- The MVCL instruction can move up to 16M (but uses different parameters)
- The MVCLE instruction can move up to 2G (or up to 16EB in 64-bit addressing)
- In all cases, the move instruction moves 1 byte at a time (left to right in memory)
Why no one writes assembler like this...

- Many instructions may appear very similar but may have very different (and unintended) consequences, e.g.:
  
  \[
  \text{LR\ 1,2}\quad \text{Load register 1 with the value of register 2}
  \]

- Unless you know what you're doing, don't do this:
  
  \[
  \text{L\ 1,2}\quad \text{Load register 1 with the value at memory offset 2}
  \]

- The very simple example above shows how confusing assembler programming can be as there is no distinction between writing the value of a number and the value of a register in an instruction and instead the distinction is made by which instruction was written by the programmer.

- Many programmers often name their registers r0-r15 to make it clearer to understand which values are registers and which values are numbers in an instruction. To do this, the best solution is to start a program with the statement `ASMDREG` which will include all the register names for your program to use.
  
  - From now on, this material will refer to GPRs registers as r0-r15

- Using r0-r15 also means that type checking can be performed by HLASM and they will also appear in the cross-reference section of the listing making it easier to find that rogue register...
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  r1,MY_NUM    XOR REGISTER 1 WITH MY_NUM (32-BITS)
  XG r1,MY_NUM    XOR REGISTER 1 WITH MY_NUM (64-BITS)
  XR r1,r2        XOR REGISTER 1 WITH REGISTER 2 (32-BITS)
  XGR r1,r2       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 (Nx), OR (Ox)

- The AND instructions perform the AND bit-wise operation
  
  N   r1,MY_NUM    AND REGISTER 1 WITH MY_NUM (32-BITS)
  NG  r1,MY_NUM    AND REGISTER 1 WITH MY_NUM (64-BITS)
  NR  r1,r2        AND REGISTER 1 WITH REGISTER 2 (32-BITS)
  NGR r1,r2        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   r1,MY_NUM    OR REGISTER 1 WITH MY_NUM (32-BITS)
  OG  r1,MY_NUM    OR REGISTER 1 WITH MY_NUM (64-BITS)
  OR  r1,r2        OR REGISTER 1 WITH REGISTER 2 (32-BITS)
  OGR r1,r2        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 r15,r15 XOR REGISTER 15 WITH REGISTER 15
  - L r15,=F'0' LOAD REGISTER 15 WITH 0
  - LA r15,0 LOAD REGISTER 15 WITH ADDRESS 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 17690, 17691 – SHARE – Orlando 2015

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 r1,r2  ADD REGISTER 2 TO REGISTER 1 (32-BIT SIGNED)
  - ALR r1,r2 ADD REGISTER 2 TO REGISTER 1 (32-BIT LOGICAL)
  - A r1,MY_NUM ADD MY_NUM TO REGISTER 1 (32-BIT SIGNED)
  - AL r1,MY_NUM ADD MY_NUM TO REGISTER 1 (32-BIT LOGICAL)
  - AFI r1,37 ADD 37 TO REGISTER 1 (IMMEDIATE 32-BIT SIGNED)

- 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**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR r1,r2</td>
<td>SUBTRACT REGISTER 2 FROM REGISTER 1 (SIGNED)</td>
</tr>
<tr>
<td>SLR r1,r2</td>
<td>SUBTRACT REGISTER 2 FROM REGISTER 1 (LOGICAL)</td>
</tr>
<tr>
<td>S r1,MY_NUM</td>
<td>SUBTRACT MY_NUM FROM REGISTER 1 (SIGNED)</td>
</tr>
<tr>
<td>SL r1,MY_NUM</td>
<td>SUBTRACT MY_NUM FROM REGISTER 1 (LOGICAL)</td>
</tr>
<tr>
<td>AFI r1,-37</td>
<td>ADD -37 TO REGISTER 1 (IMMEDIATE 32-BIT SIGNED)</td>
</tr>
</tbody>
</table>

- 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 r2,r7  MULTIPLY REGISTER 2 BY REGISTER 7
  M r2,MY_NUM  MULTIPLY REGISTER 2 BY MY_NUM

- 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  r2,r7       DIVIDE REGISTER 2 BY REGISTER 7  
  D   r2,MY_NUM   DIVIDE REGISTER 2 BY MY_NUM
  ```

- 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
Introduction to Assembler Programming

Branching
Branching

- Branching allows control flow in the program to move nonsequentially.
- 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, HLASM provides extended-mnemonics which provide branch masks for most branch instructions.
Branching – Using HLASM's extended-mnemonics

- B – Branch (unconditionally)
- BE – Branch on condition Equal
- BL – Branch on condition Lower than
- BH – Branch on condition Higher than
- BNL – Branch Not Low
- BNH – Branch Not High
- BZ – Branch on Zero
- BNZ – Branch Not Zero
- There are also other extended-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

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask value</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask value</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- So, 8 → b'1000' → 8
- Thus the branch is taken if CC 0 is met
Branching – Using a branch to form an \textit{if} statement

\begin{verbatim}
LT   r1,MY_NUM  LOAD MY_NUM 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
\end{verbatim}
Branching – Using a branch to form an *if* statement

//Example C-like equivalent
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

Looping
Looping

- A simple loop is formed by using a counter, a comparison and a branch, e.g.
  
  ```
  LA    r2,0           INITIALISE COUNTER REGISTER TO 0
  MYLOOP AHI   r2,1           INCREMENT COUNTER
  WTO   'HELLO'        SAY HELLO
  CHI   r2,10          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    r2,10           INITIALISE COUNTER REGISTER TO 10
  MYLOOP WTO   'HELLO'
  BCT   r2,MYLOOP       SUBTRACTS, COMPARES & BRANCHES
  ```

- There are other instructions similar to BCT that subtract/add values and then branch depending on the result, e.g. BCTR, BXH etc...
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 r1,MY_NUM       LOAD RELATIVE REGISTER 1 WITH MY_NUM
  ...more awesome assembler code here...
  MY_NUM 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
  - Base (and index) registers, are GPRs and the term base and index are used purely for referring to the different usage in a particular set of instructions

- 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(register) + INDEX(register) + DISPLACEMENT

![Diagram showing Address calculation and registers](image-url)
Addressing Data - Base-Displacement-Index

- Address of MYDATA = 5(r0,r12) → displacement 5 + index (register) 0 + base (register) 12
Addressing Data - Base-Displacement-Index

- Address of 'L' in 'HELLO' = 5(r4,r12) → displacement 5 + index (register) 4 + base (register) 12

```
 Register 12
    F0
    FE
    12
    AC
    07
    H
    E
    L
    L
    O
```

```
 Register 4 → 3
    +0
    +1
    +2
    +3
    +4
    +5
```

```
MYDATA
```
Addressing Data – Loading addresses

- To load an address into a register, use the LOAD ADDRESS (LA) instruction
  
  LA   r1,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   r1,12       base=0, index=0, displacement=12

- To store a 'L' in 'HELLO' in the previous example:
  
  ...some setup for REGISTER 12...
  
  LA   r4,3       LOAD ADDRESS 3* INTO REGISTER 4
  IC   r3,=C'L'   LOAD CHARACTER 'L' INTO REGISTER 3
  STC  r3,MYDATA(r4)  base=12, index=4, displacement=5
Addressing Data – Base VS Index registers

- Both base and index registers can be dropped from an instruction and can almost be used interchangeably. Consider the following:

  ```
  LA  r4,MYDATA       LOAD ADDRESS OF MYDATA INTO REGISTER 4
  IC  r3,=C'H'        LOAD CHARACTER 'H' INTO REGISTER 3
  STC r3,0(,r4)       STORE H AT 0 DISP + (0 INDEX) + BASE 4
  ```

- Contrast with:

  ```
  LA  r4,MYDATA       LOAD ADDRESS OF MYDATA INTO REGISTER 4
  IC  r3,=C'H'        LOAD CHARACTER 'H' INTO REGISTER 3
  STC r3,0(r4)        STORE H AT 0 DISP + 4 INDEX + (BASE 0)
  ```

- Both store the character H at the same address, i.e. at a displacement 0 from the start of MYDATA.

- However, if the program is running in Access Register Mode, then using a base register will cause its corresponding access register to be involved in the address calculation too!
Addressing Data – When a base register is not a base register...

- Certain instructions, e.g. various shift instructions, specify a base-displacement operand but do not address storage

- Instead, the calculated address forms a value for the operand which is used by the instruction

- The advantage of instructions such as this is that by varying the contents of a register, the same instruction in a program can be used repeatedly but it will modify varying amounts of data
Addressing Data – the HLASM USING instruction

- HLASM can be used to generate base-displacement values for your program automatically.
- In order to do this, the USING instruction is used to specify a base register for program / data addressability.
- HLASM will then calculate any displacements necessary in order to address parts of the program or data, e.g.:
  
  ```assembly
  BALR r12,0  ; LOAD ADDRESS OF NEXT INSTRUCTION INTO R12
  USING *\r12  ; USE R12 AS A BASE REGISTER FROM HERE
  ```

- If a register that is being used as a base register by HLASM needs to be used for another purpose, it is good practice to use the DROP instruction to cancel the previous USING instruction for that register, e.g.:
  
  ```assembly
  DROP r12  ; DROP THE USE OF R12 AS A BASE
  ```

- Note that HLASM has different forms for the USING instruction – the HLASM Language Reference should be consulted for more information.
Addressing Data – using multiple base registers

- For most instructions, the displacement field is only 12-bits in length and therefore a maximum displacement value of 4095 can be achieved.

- If a section of code or data is bigger than 4096-bytes in size, then the programmer must use another base register to address the data further into the code. This can be done by issuing another USING instruction and assigning another register as a base register.
  - It is important that the other base register's contents point at the correct place in the code which it will be addressing

- A common use of multiple base registers is to assign one set of base registers for a piece of code and another for the data in the program.

- The HLASM PUSH instruction can be used to save the current state of the USINGs for a program and the POP instruction can be used to restore the previous USING state.
Addressing Data – using multiple base registers

* Call the subroutine MYSUB

LA r15,MYSUB Prepare to call subroutine
BALR r14,r15 Call it
LTR r15,r15 Check return code

* *************************************************************************
* START OF MY SUBROUTINE MYSUB
* This subroutine starts with the contents of register 15 pointing to the
* start of the code. The mainline code of the program uses register 12 as
* its base register.
* *************************************************************************

MYSUB DC 0H
* Subroutine code goes here...
L r15,RETURN_CODE Set return code
BR r14 Branch back to caller
RETURN_CODE DC F'1'
Addressing Data – using multiple base registers

Register 12 is being used as a base register for both the main program and the subroutine.
Addressing Data – using multiple base registers

* Call the subroutine MYSUB
  LA    r15, MYSUB        Prepare to call subroutine
  BALR  r14, r15          Call it
  LTR   r15, r15          Check return code

* *****************************************************
* START OF MY SUBROUTINE MYSUB
* This subroutine starts with the contents of register 15 pointing to the
* start of the code. The mainline code of the program uses register 12 as
* its base register.
* *****************************************************
  MYSUB    DC  0H
  * Subroutine code goes here...
  PUSH USING
  USING *, r15      Set new base register*
  L     r15, RETURN_CODE  Set return code
  BR    r14          Branch back to caller
  POP   USING
  RETURN_CODE DC  F'1'

© 2009 IBM Corporation
Addressing Data – using multiple base registers

- Register 12 is being used as a base register for the main program
- The current state of the USINGs is saved via the PUSH USING statement
- Register 15 is established as the base register for the subroutine
Addressing Data – Addressing beyond 4096 bytes

- There are a number of methods for addressing code and data beyond 4096 bytes – we have already see how we could use more than one base register to do this.

- The LONG DISPLACEMENT family of instructions can address up to 512KB (both positive and negative from the base register) of data using 20-bits. This is only available if the long-displacement facility is installed.

- Using relative instructions, an immediate field is encoded into the instruction which specifies the number of half-words that the target of the instruction is from the instruction itself.
  - Relative instructions can be used to eliminate the need for base registers altogether – so long as your code has been designed for it
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 register 13 to the value it was previously
  - Restore registers 14,0,1,...,12 from the save area pointed to by register 13
  - Branch back on register 14
Calling a subroutine in code – Going in...

- The caller calls the subroutine

  LA r1, PARAMS       POINT TO PARAMETERS
  LA r15, SUB1        LOAD ADDRESS OF SUBROUTINE
  BALR r14, r15       BRANCH AND LINK
  LTR r15, r15        CHECKS RETURN CODE 0?
  ...caller code continues here...

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

  STM r14, r12, 12(r13) STORE REGISTERS
  LR r12, r15         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
  
  \[
  \begin{align*}
  &\text{LM } r14, r12, 12(r13) & \text{RESTORE REGISTERS} \\
  &\text{XR } r15, r15 & \text{SET RETURN CODE 0} \\
  &\text{BR } r14 & \text{BRANCH BACK TO CALLER}
  \end{align*}
  \]

- 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
What exactly is a “save area”?

- As with most things in software, the answer is, “it depends”
  - On z Systems, there are different types of save area which are used differently depending on the calling convention in use.
  - Your program must use the correct type of save area

- The standard z/OS linkage save area has the following format:

<table>
<thead>
<tr>
<th>Offset in save area</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Used by language products</td>
</tr>
<tr>
<td>4</td>
<td>Address of previous save area</td>
</tr>
<tr>
<td>8</td>
<td>Address of next save area</td>
</tr>
<tr>
<td>12</td>
<td>Register 14</td>
</tr>
<tr>
<td>16</td>
<td>Register 15</td>
</tr>
<tr>
<td>20</td>
<td>Registers 0-12</td>
</tr>
</tbody>
</table>
Chaining Save areas

Register 13

Lang
Prev
Next
Regs

Lang
Prev
Next
Regs

Lang
Prev
Next
Regs
Chaining save areas – Going in...

- The caller calls the subroutine

  LA    r1,PARAMS       POINT TO PARAMETERS
  LA    r15,SUB1        LOAD ADDRESS OF SUBROUTINE
  BALR  r14,r15         BRANCH AND LINK
  LTR   r15,r15         CHECKS RETURN CODE 0?
  ...caller code continues here...

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

  STM   r14,r12,12(r13)   STORE REGISTERS
  GETMAIN RU,LV=72        Get storage for save area
  ST    r13,4(,r1)       Chain previous save area to new
  ST    r1,8(,r13)       Chain new to previous
  LR    r13,r1           Set r13 to new save area
  LR    r12,r15          GET ENTRY ADDRESS
  ...subroutine code continues here...
Chaining save areas – Getting out...

- The caller calls the subroutine

```assembly
LA  r1,PARAMS       POINT TO PARAMETERS
LA  r15,SUB1        LOAD ADDRESS OF SUBROUTINE
BALR r14,r15        BRANCH AND LINK
LTR  r15,r15        CHECKS RETURN CODE 0?
...caller code continues here...
```

- The subroutine restores frees its save area, restores the caller's registers, sets a return code and branches back

```assembly
LR   r1,r13        Address of save area to free
LA   r0,72         Length of save area
L    r13,4(,r13)   Point at previous save area
FREEMAIN R,LV=(0),A=(1) Free save area
LM   r14,r12,12(13) Restore registers
XR   r15,r15       Set return code
BR   r14           Branch back to caller
```
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
  - Hint – Appendix A contains examples of instructions
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
LOAD

Register-and-register formats:

\[
\begin{array}{ccc}
\text{LR} & \text{R}_1,\text{R}_2 & \text{[RR]} \\
\hline
'18' & \text{R}_1 & \text{R}_2 \\
0 & 8 & 12 & 15 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{LGR} & \text{R}_1,\text{R}_2 & \text{[RRE]} \\
\hline
'B904' & \hline & \text{R}_1 & \text{R}_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{LGFR} & \text{R}_1,\text{R}_2 & \text{[RRE]} \\
\hline
'B914' & \hline & \text{R}_1 & \text{R}_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]
Reading POPs – Instruction Formats – RX – L instruction

Register-and-storage formats:

<table>
<thead>
<tr>
<th>'58'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

[RX-a]
Looking at Instruction Formats

- Why do I need to know an instruction's format?
  - You don't...but it might come in useful when debugging...

- Consider having a dump and the failing instruction was 5810 0004

- Examining the “Principles of Operation” z/Architecture manual tells me:
  - 58 = LOAD Instruction
  - Format = RX-a

### Register-and-storage formats:

<table>
<thead>
<tr>
<th>'58'</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

© 2015 IBM Corporation
Looking at Instruction Formats

**Register-and-storage formats:**

<table>
<thead>
<tr>
<th>L</th>
<th>R₁, D₂(X₂, B₂)</th>
<th>[RX-a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'58'</td>
<td>R₁</td>
<td>X₂</td>
</tr>
</tbody>
</table>

| 0 | 8 | 12 | 16 | 20 | 31 |

5810 0004

**Register-and-storage formats:**

<table>
<thead>
<tr>
<th>L</th>
<th>R₁, D₂(X₂, B₂)</th>
<th>[RX-a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'58'</td>
<td>R₁</td>
<td>X₂</td>
</tr>
</tbody>
</table>

| 0 | 8 | 12 | 16 | 20 | 31 |
Looking at Instruction Formats

### Register-and-storage formats:

<table>
<thead>
<tr>
<th>L</th>
<th>R₁, D₂(X₂, B₂)</th>
<th>[RX-a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'58'</td>
<td>R₁</td>
<td>X₂</td>
</tr>
</tbody>
</table>

| 0 | 8 | 12 | 16 | 20 | 31 |

\[
\text{58 100004}
\]
Looking at Instruction Formats

58 100004

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

L  R_1, D_1(X_2, B_2)  [RX-a]

L  1,4(0,0)
Reading HLASM Listings
Reading HLASM Listings

- One of the outputs produced by HLASM is the “listing” - it explains in detail all that transformations that have happened to change your source code into the produced object code.

- In certain programs, it is not possible to use a debugger and it is in these circumstances where relying on a dump and the assembler listing proves invaluable.
  - Examining a dump and a listing can also be much quicker to solve the problem than trying to work through the program with a debugger

- Unlike a lot of compilers, the HLASM listing contains much more information than just error messages that were produced when attempting to assemble your source code:
Reading HLASM Listings

- The HLASM listing is divided into:
  - High Level Assembler Option Summary
  - External Symbol Dictionary
  - Source Statements
  - Relocation Dictionary
  - Ordinary Symbol and Literal Cross Reference
  - Unreferenced Symbols Defined in CSECTs
  - Macro and Copy Code Source Summary
  - Macro and Copy Code Cross Reference
  - DSECT Cross Reference
  - Using Map
  - General Purpose Register Cross Reference
  - Diagnostic Cross Reference and Assembler Summary
Specifying options

- HLASM options can be specified in:
  - *PROCESS OVERRIDE
  - ASMAOPT
  - Invocation Parms
  - *PROCESS
  - Installation Defaults

- The location in which an option is specified since the location of each option specifies its precedence over other options.
  - Also some options cannot be specified in certain places, e.g. *PROCESS OVERRIDE VS *PROCESS
Specifying options

Our example specifies:
In source:
*process override(adata,mxref(full))
*process align
*process nodbcs
*process mxref(full),nolibmac
*process flag(0)
*process nofold,language(ue)
*process nora2
*process nodbcs
*process xref(full)

In JCL procedure - parms:
OPTS1='NOOBJECT,language(en),size(4meg)',
OPTS2='xref(short,unrefs)',
OPTS3='nomxref,norxref,adata,noadata'
//C   EXEC PGM=ASMA90,
//    PARM='&OPTS1,&OPTS2,&OPTS3'

In ASAMOPT DD:
//ASAMOPT DD *
* My ASMAOPTS Overides
sysparm(thisisatestsysparm)       pass this to assembler
gooff                            create GOFF object code
/*
High Level Assembler Option Summary

Overriding ASMAOPT Parameters -
>* My ASMAOPTS Overides
>sysparm(thisisatestsysparm) pass this to assembler
>goff create GOFF object code

Overriding Parameters- NOOBJECT,language(en),size(4meg),xref(short,unrefs),nomxref,norxref,adata,nodata
Process Statements- override(adata,mxref(full))
align	nodbc
mxref(full),nolibmac
flag(0)	nofold,language(ue)
nora2	nodbc
xref(full)

** ASMA434N GOFF/XOBJECT option specified, option LIST(133) will be used
** ASMA400W Error in invocation parameter - size(4meg)
** ASMA423N Option adata, in a *PROCESS OVERRIDE statement conflicts with invocation or default option. Option is not permitted in a *PROCESS statement and has been ignored.
** ASMA422N Option language(ue) is not valid in a *PROCESS statement.
** ASMA437N Attempt to override invocation parameter in a *PROCESS statement. Suboption full of xref option ignored.
Options for this Assembly

3 OverridingParms NOADATA
5 *Process ALIGN
   NOASA
   NOBATCH
   CODEPAGE(047C)
   NOCOMPAT
5 *Process NODBCS
   NODECK
   DXREF
   ESD
   NOEXIT
5 *Process FLAG(0,ALIGN,NOCONT,EXLITW,NOIMPLEN,NOPAGE0,PUSH,RECORD,NOSUBSTR,USING0)
5 *Process NOFOLD
2 ASMAOPT GOFF(NOADATA)
NOINFO

3 OverridingParms LANGUAGE(EN)
5 *Process NOLIBMAC
   LINECOUNT(60)
   LIST(133)
   MACHINE(,,NOLIST)
1 *Process Override MXREF(FULL)

3 OverridingParms NOOBJECT
   OPTABLE(UNI,NOLIST)
   NOPCONTROL
   NOPESTOP
   NOPROFILE
5 *Process NORA2
   NORENT
   RLD

3 OverridingParms NORXREF
   SECTALGN(8)
   SIZE(MAX)
   TYPECHECK(MAGNITUDE,REGISTER)
   USING(NOLIMIT,MAP,NOWARN)
   NOWORKFILE
3 OverridingParms XREF(SHORT,UNREFS)
### External Symbol Dictionary (ESD)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Id</th>
<th>Address</th>
<th>Length</th>
<th>Owner Id</th>
<th>Flags</th>
<th>Alias-of</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTINGB</td>
<td>SD</td>
<td>00000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_IDRL</td>
<td>ED</td>
<td>00000002</td>
<td></td>
<td></td>
<td>00000001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_PRV</td>
<td>ED</td>
<td>00000003</td>
<td></td>
<td></td>
<td>00000001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_TEXT</td>
<td>ED</td>
<td>00000004</td>
<td>00000000</td>
<td>00000084</td>
<td>00000001</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>LISTINGB</td>
<td>LD</td>
<td>00000005</td>
<td>00000000</td>
<td></td>
<td>00000004</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>FUNCTION</td>
<td>ER</td>
<td>00000006</td>
<td></td>
<td></td>
<td>00000001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUNCY</td>
<td>ER</td>
<td>00000007</td>
<td></td>
<td></td>
<td>00000001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>listme</td>
<td>ER</td>
<td>00000008</td>
<td></td>
<td></td>
<td>00000001</td>
<td>LISTINGZ</td>
<td></td>
</tr>
<tr>
<td>COMMON_DATA</td>
<td>SD</td>
<td>00000009</td>
<td></td>
<td></td>
<td>00000009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_IDRL</td>
<td>ED</td>
<td>0000000A</td>
<td></td>
<td></td>
<td>00000009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_PRV</td>
<td>ED</td>
<td>0000000B</td>
<td></td>
<td></td>
<td>00000009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B_TEXT</td>
<td>ED</td>
<td>0000000C</td>
<td>00000000</td>
<td>00000018</td>
<td>00000009</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>COMMON_DATA</td>
<td>CM</td>
<td>0000000D</td>
<td>00000000</td>
<td></td>
<td>0000000C</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

This section of the listing contains the External Symbol Dictionary information passed to the Binder.
External Symbol Dictionary

- Each entry in the ESD has a particular type:

- **SD** – Section Definition
  - The symbol appeared in the name field of a START, CSECT or RSECT instruction

- **LD** – Label Definition
  - The symbol appeared as the operand of an ENTRY statement. When you specify the GOFF assembler option on z/OS or CMS, the assembler generates an entry type of LD for each CSECT and RSECT name.

- **ER** – External Reference
  - The symbol appeared as the operand of an EXTRN statement or appeared as an operand of a V-type address constant.

- **CM** – Common control section definition
  - The symbol appeared in the name field of a COM statement
Source Statement

- This section of the listing documents source statements of the module and the resulting object code.

- The TITLE, CEJECT and EJECT assembler instructions can be used to control when the page title is printed.

```
00000034 9836 2014 00000014 360 * 00000034 9836 2014 00000014 361 continue_code_again dc 0h'0' define a label
00000038 58B0 A038 00000080 362          lm  r3,r6,l_regs       reload registers
0000003C 50B0 3014 00000014 363          l   r11,=f'1504'       load constant
00000040 58C0 A028 00000070 364          st  r11,comm_country   ... and save in common
00000044 07FE                                365          l   r12,pDateFormat    load address constant
00000046 0000 366          br  r14                and exit program
00000048                                     367          lm  r3,r6,l_regs       reload registers
00000048 839697A899898788                    369 static_data dc 0d'0'            start of static
00000062 00000000                            370 copyright   dc c'copyright IBM(UK) Ltd 2013'
00000068 E8E8E8E8D4D4C4C4                    371 listingx    dc v(external_function)     declare external
00000068 E8E8E8E8D4D4C4C4                    372 dateFormat  dc cl(date_len)'YYYYMMDD'   define format
```
**Source Statement – Location Counter and Statement number**

<table>
<thead>
<tr>
<th>Active Using</th>
<th>Linkage Data</th>
<th>R2</th>
<th>Common Data</th>
<th>R3</th>
<th>Static Data</th>
<th>R10</th>
<th>R-Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
<th>HLASM R6.0 2013/11/01 15.23</th>
</tr>
</thead>
<tbody>
<tr>
<td>static_data dc 0d'0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>369</td>
<td>static_data dc 0d'0' start of static</td>
<td></td>
</tr>
<tr>
<td>copyright dc c'copyright IBM(UK) Ltd 2013'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>370</td>
<td>copyright dc c'copyright IBM(UK) Ltd 2013'</td>
<td></td>
</tr>
<tr>
<td>listingx dc v(external_function) declare external</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>371</td>
<td>listingx dc v(external_function) declare external</td>
<td></td>
</tr>
<tr>
<td>dateFormat dc c1(date_len)'YYYYMDDD' define format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>372</td>
<td>dateFormat dc c1(date_len)'YYYYMDDD' define format</td>
<td></td>
</tr>
</tbody>
</table>
### Source Statement – Location Counter and Statement number

<table>
<thead>
<tr>
<th>R-Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000074</td>
<td>00000000</td>
<td></td>
<td></td>
<td>374</td>
<td>pZERO dc a(0)</td>
</tr>
<tr>
<td>00000078</td>
<td>00000000</td>
<td></td>
<td></td>
<td>375</td>
<td>ListData dc v(FUNCY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>376</td>
<td>extrn listingz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>377</td>
<td>listingz alias c'listme'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>378</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>379</td>
<td>ltorg f'1504'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>380</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>381</td>
<td>static_data_end equ *</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>382</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>383</td>
<td>drop r2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>384</td>
<td>*</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000018</td>
<td></td>
<td>385</td>
<td>drop r3</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
<td>00000018</td>
<td></td>
<td>386</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>A8A8A8A894948484</td>
<td></td>
<td></td>
<td>387</td>
<td>common_data com</td>
</tr>
<tr>
<td>00000000</td>
<td>A8A8A8A894948484</td>
<td></td>
<td></td>
<td>388</td>
<td>comm_date dc cl(date_len)'yyyy....'</td>
</tr>
<tr>
<td>00000008</td>
<td>A8A8A8A894948484</td>
<td></td>
<td></td>
<td>389</td>
<td>comm_user dc cl(10)'yyyyymmdd'</td>
</tr>
<tr>
<td>00000012</td>
<td>0000</td>
<td></td>
<td></td>
<td>390</td>
<td>comm_country dc f'0000'</td>
</tr>
<tr>
<td>00000014</td>
<td>00000000</td>
<td></td>
<td></td>
<td>391</td>
<td>*</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000018</td>
<td></td>
<td>392</td>
<td>linkage_data dsect</td>
</tr>
<tr>
<td>00000000</td>
<td>c140d58194854040</td>
<td></td>
<td></td>
<td>393</td>
<td>l_name dc cl(10)'A Name'</td>
</tr>
<tr>
<td>0000000A</td>
<td>A8A8A8A894948484</td>
<td></td>
<td></td>
<td>394</td>
<td>l_date dc cl(date_len)'yyyyymmdd'</td>
</tr>
</tbody>
</table>
Source Statement – Location Counter and Statement number

19          print on,gen               print statements
20          sysstate archlvl=1         set arch level
21+*        THE VALUE OF SYSSTATE IS NOW SET TO ASCENV=P AMODE64=NO
            +               L=I OSREL=00000000
22          ieabrcx  DEFINE            use relative branching
00000048    341          larl  r10,static_data
342          using (static_data,static_data_end),r10
343 *

The column following the statement number contains one of these values:

A space ( ) indicates open source

A plus sign (+) indicates that the statement was generated as the result of macro call processing.

An unnumbered statement with a plus sign (+) is the result of open code substitution.

A minus sign (-) indicates that the statement was read by a preceding AREAD instruction.

An equals sign (=) indicates that the statement was included by a COPY instruction.

A greater-than sign (>) indicates that the statement was generated as the result of a preceding AINSERT instruction. If the statement is read by an AREAD instruction, this takes precedence and a minus sign is printed.
### Source Statement - Addr1 and Addr2 Fields and USING Statements

<table>
<thead>
<tr>
<th>R-Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>C0A0 0000 0024</td>
<td>00000048</td>
<td></td>
<td>21+*</td>
<td>THE VALUE OF SYSSTATE IS NOW SET TO ASCENV=P..</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22</td>
<td>ieabrcx DEFINE use relative branching</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>341</td>
<td>larl r10.static_data</td>
</tr>
<tr>
<td>R:A</td>
<td>00000048</td>
<td></td>
<td></td>
<td>342</td>
<td>using (static_data,static_data_end),r10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>343</td>
<td></td>
</tr>
<tr>
<td>00000006</td>
<td>17FF</td>
<td></td>
<td></td>
<td>344</td>
<td>xr r15,r15 initialise register</td>
</tr>
<tr>
<td>R:2</td>
<td>00000000</td>
<td></td>
<td></td>
<td>345</td>
<td>using (linkage_data,l_end),r2 set using scope</td>
</tr>
<tr>
<td>R:3</td>
<td>00000000</td>
<td></td>
<td></td>
<td>346</td>
<td>using common_data,r3 set using scope</td>
</tr>
<tr>
<td>00000008</td>
<td>C040 0000 000A</td>
<td>0000001C</td>
<td></td>
<td>347</td>
<td>larl r4,address_constant address of....</td>
</tr>
<tr>
<td>0000000E</td>
<td>D207 3000 200A</td>
<td>00000000</td>
<td>0000000A</td>
<td>348</td>
<td>mvc comm_date,l_date copy</td>
</tr>
<tr>
<td>00000014</td>
<td>9036 2014</td>
<td>00000014</td>
<td>00000014</td>
<td>349</td>
<td>stm r3,r6,l_regs save a copy of the reg..</td>
</tr>
<tr>
<td>00000018</td>
<td>A7F4 0008</td>
<td></td>
<td>00000028</td>
<td>350</td>
<td>j continue_code jump over constant</td>
</tr>
<tr>
<td>0000001C</td>
<td>C1C4C4D9C5E2E240</td>
<td>00000048</td>
<td></td>
<td>351</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>352 address_constant dc cl12'ADDRESS' constant value</td>
</tr>
</tbody>
</table>
Relocation Dictionary (RLD)

<table>
<thead>
<tr>
<th>Pos.Id</th>
<th>Rel.Id</th>
<th>Address</th>
<th>Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000004</td>
<td>00000004</td>
<td>00000070</td>
<td>A 4</td>
<td>+</td>
</tr>
<tr>
<td>00000004</td>
<td>00000006</td>
<td>00000064</td>
<td>V 4</td>
<td>ST</td>
</tr>
<tr>
<td>00000004</td>
<td>00000007</td>
<td>00000078</td>
<td>V 4</td>
<td>ST</td>
</tr>
</tbody>
</table>

Location counter 70, and ADCON symbol pDateFormat has a value of 00000068 – which is the location counter for symbol dateFormat

- This section of the listing describes the relocation dictionary information passed to the Binder.
- The entries describe the address constants in the assembled program that are affected by relocation.
- This section helps you find relocatable constants in your program.
Ordinary Symbol and Literal Cross Reference

<table>
<thead>
<tr>
<th>Symbol name</th>
<th>Length</th>
<th>Value</th>
<th>Id</th>
<th>R Type</th>
<th>Asm</th>
<th>Program</th>
<th>Defn</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_constant</td>
<td>12</td>
<td>0000001c</td>
<td>00000004</td>
<td>C</td>
<td>C</td>
<td></td>
<td>352</td>
<td>347</td>
</tr>
<tr>
<td>comm_country</td>
<td>4</td>
<td>00000014</td>
<td>0000000c</td>
<td>F</td>
<td>F</td>
<td></td>
<td>390</td>
<td>364M</td>
</tr>
<tr>
<td>comm_date</td>
<td>8</td>
<td>00000000</td>
<td>0000000c</td>
<td>C</td>
<td>C</td>
<td></td>
<td>388</td>
<td>348M</td>
</tr>
<tr>
<td>common_data</td>
<td>1</td>
<td>00000000</td>
<td>0000000c</td>
<td>J</td>
<td></td>
<td></td>
<td>387</td>
<td>346U</td>
</tr>
<tr>
<td>continue_code</td>
<td>2</td>
<td>00000028</td>
<td>00000004</td>
<td>H</td>
<td>H</td>
<td></td>
<td>354</td>
<td>350B</td>
</tr>
<tr>
<td>continue_code_again</td>
<td>2</td>
<td>00000034</td>
<td>00000004</td>
<td>H</td>
<td>H</td>
<td></td>
<td>361</td>
<td>357B</td>
</tr>
<tr>
<td>date_len</td>
<td>1</td>
<td>00000008</td>
<td>00000004</td>
<td>A</td>
<td>U</td>
<td></td>
<td>362</td>
<td>388 394</td>
</tr>
<tr>
<td>r10</td>
<td>1</td>
<td>0000000a</td>
<td>00000004</td>
<td>A</td>
<td>U</td>
<td>GR32</td>
<td>12</td>
<td>341M 342U</td>
</tr>
<tr>
<td>r11</td>
<td>1</td>
<td>0000000b</td>
<td>00000004</td>
<td>A</td>
<td>U</td>
<td>GR32</td>
<td>13</td>
<td>363M 364</td>
</tr>
<tr>
<td>r12</td>
<td>1</td>
<td>0000000c</td>
<td>00000004</td>
<td>A</td>
<td>U</td>
<td>GR32</td>
<td>14</td>
<td>365M</td>
</tr>
<tr>
<td>r14</td>
<td>1</td>
<td>0000000e</td>
<td>00000004</td>
<td>A</td>
<td>U</td>
<td>GR32</td>
<td>16</td>
<td>366B</td>
</tr>
<tr>
<td>r15</td>
<td>1</td>
<td>0000000f</td>
<td>00000004</td>
<td>A</td>
<td>U</td>
<td>GR32</td>
<td>17</td>
<td>344M 344</td>
</tr>
</tbody>
</table>

Symbol name, length, value, Assembler type, Definition

References – no suffix, Branch, Modified, Using, Drop and eXecution

This section of the listing concerns symbols and literals that are defined and used in the program.
Unreferenced Symbols Defined in CSECTs

Unreferenced Symbols Defined in CSECTs

<table>
<thead>
<tr>
<th>Defn</th>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>370</td>
<td>copyright</td>
<td>370</td>
</tr>
<tr>
<td>375</td>
<td>ListData</td>
<td>375</td>
</tr>
<tr>
<td>371</td>
<td>listingx</td>
<td>371</td>
</tr>
<tr>
<td>374</td>
<td>pZERO</td>
<td>374</td>
</tr>
<tr>
<td>6</td>
<td>r0</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>r13</td>
<td>15</td>
</tr>
</tbody>
</table>

This section of the listing shows symbols that have been defined in CSECTs but not referenced. This may help you to remove unnecessary data definitions, and reduce the size of your program.
### Macro and Copy Code Source Summary

<table>
<thead>
<tr>
<th>Con Source</th>
<th>Macro and Copy Code Source Summary</th>
<th>HLASM R6.0</th>
<th>2013/11/01 15.23</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMARY INPUT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>BAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAS</td>
<td>BC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>BCT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BE</td>
<td>BH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td>BM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>BNH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNL</td>
<td>BNM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNM</td>
<td>BNO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>BXH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BXH</td>
<td>BXLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>SYS1.MACLIB</td>
<td>37SY01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IEABRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IEABRCX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSSTATE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**In section 'Diagnostic Cross Reference and Assembler Summary’**

**Data Sets Allocated for this Assembly**

<table>
<thead>
<tr>
<th>Con DDname</th>
<th>Data Set Name</th>
<th>Volume</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>ASMAOPT SMORSA.LISTINGC.JOB63822.D0000101.?</td>
<td>37P001</td>
<td>LISTINGC</td>
</tr>
<tr>
<td>P1</td>
<td>SYSIN SMORSA.ASM.ASM</td>
<td>37P001</td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>SYSLIB SMORSA.ASM.ASM</td>
<td>37P001</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>SYSLIN SYS13305.T152320.RA000.LISTINGC.OBJ.H01</td>
<td>37SY01</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>SYSPRINT SMORSA.LISTINGC.JOB63822.D0000102.?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This section of the listing shows the names of the macro libraries from which the assembler read macros or copy code members, and the names of the macros and copy code members that were read from each library.
## Macro and Copy Code Cross Reference

<table>
<thead>
<tr>
<th>Macro</th>
<th>Con</th>
<th>Called By</th>
<th>Defn</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>PRIMARY INPUT</td>
<td>PRIMARY INPUT</td>
<td>117</td>
<td>355</td>
</tr>
<tr>
<td>BAL</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>BAS</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>297</td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>PRIMARY INPUT</td>
<td>PRIMARY INPUT</td>
<td>177</td>
<td>356</td>
</tr>
<tr>
<td>BCT</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>BE</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>BM</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>BXLE</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>BZ</td>
<td></td>
<td>PRIMARY INPUT</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>IEABRC</td>
<td>L2</td>
<td>IEABRCX</td>
<td>-</td>
<td>22C</td>
</tr>
<tr>
<td>IEABRCX</td>
<td>L2</td>
<td>PRIMARY INPUT</td>
<td>-</td>
<td>22</td>
</tr>
<tr>
<td>SYSSTATE</td>
<td>L2</td>
<td>PRIMARY INPUT</td>
<td>-</td>
<td>20</td>
</tr>
</tbody>
</table>

This section of the listing shows the names of macros and copy code members and the statements where the macro or copy code member was called.
DSECT Cross Reference

<table>
<thead>
<tr>
<th>Dsect</th>
<th>Length</th>
<th>Id</th>
<th>Defn</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkage_data</td>
<td>00000054</td>
<td>FFFFFFFF</td>
<td>392 LINKAGE_DATA DSECT, data passed in to me</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>391 *</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>00000054 LINKAGE_DATA DSECT, data passed in to me</td>
</tr>
<tr>
<td></td>
<td>c140d58194854040</td>
<td>A8A8A8A8A894948484</td>
<td>393 L_NAME DC CL(10)'A_Name' users name</td>
</tr>
<tr>
<td></td>
<td>0000000a</td>
<td>a8a8a8a894948484</td>
<td>394 L_DATE DC CL(date_len)'yyyymmdd' users date</td>
</tr>
<tr>
<td></td>
<td>00000012</td>
<td>0000</td>
<td>395 L_REGS DC 4F'0,0,0,0' return registers values</td>
</tr>
<tr>
<td></td>
<td>00000014</td>
<td>0000000000000000</td>
<td>396 L_END EQU * end</td>
</tr>
<tr>
<td></td>
<td>00000054</td>
<td></td>
<td>397 *</td>
</tr>
</tbody>
</table>

This section of the listing shows the names of all internal or external dummy sections defined in the program, and the number of the statement where the definition of the dummy section began.
This section of the listing shows a summary of the USING, DROP, PUSH USING, and POP USING instructions used in your program.
## General Purpose Register Cross Reference

<table>
<thead>
<tr>
<th>Register</th>
<th>References (M=modified, B=branch, U=USING, D=DROP, N=index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(0)</td>
<td>(no references identified)</td>
</tr>
<tr>
<td>1(1)</td>
<td>(no references identified)</td>
</tr>
<tr>
<td>2(2)</td>
<td>345U 384D</td>
</tr>
<tr>
<td>3(3)</td>
<td>346U 349 362M 385D</td>
</tr>
<tr>
<td>4(4)</td>
<td>347M 349 362M</td>
</tr>
<tr>
<td>5(5)</td>
<td>349 362M</td>
</tr>
<tr>
<td>6(6)</td>
<td>349 362M</td>
</tr>
<tr>
<td>7(7)</td>
<td>(no references identified)</td>
</tr>
<tr>
<td>8(8)</td>
<td>(no references identified)</td>
</tr>
<tr>
<td>9(9)</td>
<td>(no references identified)</td>
</tr>
<tr>
<td>10(A)</td>
<td>341M 342U</td>
</tr>
<tr>
<td>11(B)</td>
<td>363M 364</td>
</tr>
<tr>
<td>12(C)</td>
<td>365M</td>
</tr>
<tr>
<td>13(D)</td>
<td>(no references identified)</td>
</tr>
<tr>
<td>14(E)</td>
<td>366B</td>
</tr>
<tr>
<td>15(F)</td>
<td>344M 344</td>
</tr>
</tbody>
</table>

### Register 10 – no DROP statement?

<table>
<thead>
<tr>
<th>Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>C0A0</td>
<td>0000</td>
<td>0024</td>
<td>341</td>
<td>larl  r10,static_data</td>
</tr>
<tr>
<td></td>
<td>R:A</td>
<td>00000048</td>
<td></td>
<td>342</td>
<td>using (static_data,static_data_end),r10</td>
</tr>
</tbody>
</table>

### Register 5?

<table>
<thead>
<tr>
<th>Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000014</td>
<td>9036</td>
<td>2014</td>
<td>000014</td>
<td>349</td>
<td>stm r3,r6,l_regs</td>
</tr>
<tr>
<td>00000034</td>
<td>9836</td>
<td>2014</td>
<td>000014</td>
<td>362</td>
<td>lm  r3,r6,l_regs</td>
</tr>
</tbody>
</table>

This section of the listing shows all references in the program to each of the general registers. Additional flags indicate the type of reference.
Diagnostic Cross Reference and Assembler Summary

Statements Flagged
24(P1,24)

1 Statement Flagged in this Assembly 8 was Highest Severity Code

HIGH LEVEL ASSEMBLER, 5696-234, RELEASE 6.0, PTF R160
SYSTEM: z/OS 01.13.00 JOBNAME: LISTINGB STEPNAME: B PROCSTEP: C

R-Loc Object Code Addr1 Addr2 Stmt Source Statement
0000000C 50B0 3014 00000014 22 st r11,comm_country
00000010 58C0 F040 00000040 23 l r12,pDateFormat
00000014 0000 0000 00000000 24 l r7,someData
** ASMA044E Undefined symbol - r7
** ASMA029E Incorrect register specification - r7
** ASMA044E Undefined symbol - someData
** ASMA435I Record 24 in SMORSA.ASM.ASM(LISTINGB) on volume: 37P001
00000018 07FE 25 br r14
26 *

Data Sets Allocated for this Assembly

<table>
<thead>
<tr>
<th>Con</th>
<th>DDname</th>
<th>Data Set Name</th>
<th>Volume</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>ASMAOPT</td>
<td>SMORSA.LISTINGB.JOB63825.D0000101.?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>SYSLIN</td>
<td>SMORSA.ASM.ASM</td>
<td>37P001</td>
<td>LISTINGB</td>
</tr>
<tr>
<td>L1</td>
<td>SYSLIB</td>
<td>SMORSA.ASM.ASM</td>
<td>37P001</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>SYSLIN</td>
<td>SYS1.MACLIB</td>
<td>37SY01</td>
<td></td>
</tr>
</tbody>
</table>

This section of the listing summarises the error diagnostic messages issued during the assembly, and provides statistics about the assembly.
Diagnostic Cross Reference and Assembler Summary

No Statements Flagged in this Assembly

HIGH LEVEL ASSEMBLER, 5696-234, RELEASE 6.0, PTF R160

SYSTEM: z/OS 01.13.00            JOBNAME: LISTINGC      STEPNAME: B          PROCSTEP: C

Data Sets Allocated for this Assembly

<table>
<thead>
<tr>
<th>Con</th>
<th>DDname</th>
<th>Data Set Name</th>
<th>Volume</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>ASMAOPT</td>
<td>SMORSA.LISTINGC.JOB63822.D0000101.?</td>
<td>37P001</td>
<td>LISTINGC</td>
</tr>
<tr>
<td>P1</td>
<td>SYSIN</td>
<td>SMORSA.ASM.ASM</td>
<td>37P001</td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>SYSLIB</td>
<td>SMORSA.ASM.ASM</td>
<td>37P001</td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>SYSLIB</td>
<td>SYS1.MACLIB</td>
<td>37SY01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSLIN</td>
<td>SYS13305.T152320.RA000.LISTINGC.OBJ.H01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSPRINT</td>
<td>SMORSA.LISTINGC.JOB63822.D0000102.?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4096K allocated to Buffer Pool     Storage required     200K
76 Primary Input Records Read    1093 Library Records Read    0 Work File Reads
2 ASMAOPT Records Read           312 Primary Print Records Written 0 Work File Writes
23 Object Records Written        0 ADATA Records Written
Assembly Start Time: 15.23.20 Stop Time: 15.23.20 Processor Time: 00.00.00.0044
Return Code 000
The PSW and an introduction to debugging assembler programs
Program Status Word (PSW)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0</td>
<td>Return code</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>Task tag</td>
</tr>
<tr>
<td>IE</td>
<td>0</td>
<td>Exception enable</td>
</tr>
<tr>
<td>OX</td>
<td>0</td>
<td>Operating system</td>
</tr>
<tr>
<td>Key</td>
<td>0</td>
<td>Key number</td>
</tr>
<tr>
<td>MWP</td>
<td>0</td>
<td>Maximal word pack</td>
</tr>
<tr>
<td>AS</td>
<td>1</td>
<td>Access status</td>
</tr>
<tr>
<td>CC</td>
<td>2</td>
<td>Condition code</td>
</tr>
<tr>
<td>Prog. Mask</td>
<td>3</td>
<td>Program mask</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>Instruction mask</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>End of instruction</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>Auxiliary function</td>
</tr>
</tbody>
</table>

The instruction address is located at bytes 64 to 95.

Instruction Address (Continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>96</td>
<td>Instruction address (continued)</td>
</tr>
</tbody>
</table>

© 2015 IBM Corporation
Program Status Word (PSW)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2</td>
<td>Reserved</td>
</tr>
<tr>
<td>5-7</td>
<td>OX</td>
</tr>
<tr>
<td>8</td>
<td>Key</td>
</tr>
<tr>
<td>12-16</td>
<td>MWP</td>
</tr>
<tr>
<td>20</td>
<td>Prog. Mask</td>
</tr>
<tr>
<td>24-25</td>
<td>P</td>
</tr>
<tr>
<td>31</td>
<td>EA</td>
</tr>
</tbody>
</table>

Instruction Address

Instruction Address (Continued)
Program Status Word (PSW)

- The Program Status Word (PSW) is a register in the processor which includes control information to determine the state of the CPU.

- The z/Architecture PSW is 128-bits in length
  - Bits 0-32 contain flag bits indicating control information for the CPU
  - Bits 33-63 are 0
  - Bits 64-127 contain the instruction address

- EPSW – Extract PSW
  - Obtain bits 0-63 of the PSW and place them into operands of the instruction

- LPSW(E) – Load PSW (Extended)
  - Replace the entire PSW with the contents of storage
  - This means that the instruction branches – well might do...
Program Status Word (PSW) – Addresses

Instruction Address

Instruction Address (Continued)
Program Status Word (PSW) – Addresses

Since the z/Architecture can run in a number of addressing modes, the instruction address is determined by a variable number of bits in the PSW. The current addressing mode is determined by bits 31-32 of the PSW with the following combinations:
- 00 → 24-bit mode
- 01 → 31-bit mode
- 10 → invalid
- 11 → 64-bit mode

Bits 64-127 are used to determine the address of the next instruction to be executed
- However, some instructions may be interrupted and therefore the PSW may point at the same instruction which was being executed so that it is redriven
Program Status Word (PSW) – Instruction Address

Determine Addressing mode

64-bit address

31-bit

24-bit address
Program Status Word (PSW) – Flag bits

Instruction Address

Instruction Address (Continued)
Program Status Word (PSW) – Flag bits

(A bit value of 1 indicates that the CPU is enabled for a function unless stated otherwise)

- **Bit 1 – Program Event Recording (PER) Mask**
  - Controls whether the CPU is enabled for interrupts associated with PER

- **Bit 5 – DAT Mode**
  - Controls whether the CPU has Dynamic Address Translation turned on

- **Bit 6 – I/O Mask**
  - Controls whether the CPU is enabled for interrupts

- **Bit 7 – External Mask**
  - Controls whether the CPU is enabled for external interrupts
Program Status Word (PSW) – Flag bits

Instruction Address

Instruction Address (Continued)
Program Status Word (PSW) – PSW Key

Instruction Address

Instruction Address (Continued)
Program Status Word (PSW) – PSW Key

- Bits 8-11 are used to represent the PSW Key (value of 0-15)
- PSW Keys are used to provide a security mechanism over various regions of memory with key 0 being the most secure
- Whenever an instruction attempts to access a storage location that is protected against that type of reference (read/write of storage) and the storage key does not match the access key, a protection exception is recognised.
- Programs running in PSW key 0 have read write access to storage in every storage key
- Programs in keys 1 – 15 have read access to:
  - Storage which matches their PSW key
  - Storage (in any key) that's not fetch protected
  - Storage in key 9 if the hardware feature “subsystem storage protection override” is installed
- Programs in keys 1-15 have write access to:
  - Storage whose key matches their PSW key
  - Storage in key 9 if subsystem storage protection override is installed
Program Status Word (PSW) – PSW Key – Manipulation

- **IPK – Insert PSW Key**
  - Used to *insert* the PSW Key *into* register 2
  - Used to store a copy of the current PSW Key typically before a switch to another key.
  - Bits 56-59 of register 2 are updated to contain the PSW Key, bits 60-63 are set to 0 and all other bits remain unmodified.

- **IPK** cannot be used when bit 36 of CR0 is set to 0 and in problem state.

- **SPKA – Set PSW Key from Address**
  - Used to set the PSW Key from an *address value*
  - Bits 56-59 of the 2\textsuperscript{nd} operand are inserted into the PSW Key.

- **SPKA** can only be used to set a key to which the current task is allowed to set a key determined by the PSW Key mask in CR3.

- Both IPK and SPKA are privileged instructions.
Program Status Word (PSW) – PSW Key – Manipulation

- **MVCK – Move with Key**
  - Moves an operand with an access key specified as part of the instruction
  - If the program is not enabled to use that access key, then a privileged operation exception is raised
  - Can be a slow instruction

- **BSA – Branch and Set Authority**
  - Used to branch to another place in code and set the PSW key at the same time
  - Works as a flip-flop branching from “base authority” state to “reduced authority” state
Program Status Word (PSW) – PSW Key – BSA Operation

- BSA – Branch and Set Authority – example scenario
  - A service routine, e.g. a middleware service begins in the base authority state
  - The routine issues a BSA to switch to running a user routine
  - The user routine runs in reduced authority state
  - When the user routine wants to invoke the middleware service, it issues a BSA which branches back to a fixed location in the middleware and the state is returned to running in base authority state

- The control of the states is determined by the Dispatchable Unit Control Table (DUCT)
  - The BSA instruction uses words 5, 8 and 9 of the DUCT.
Program Status Word (PSW) – PSW Key – BSA Operation - DUCT

5

<table>
<thead>
<tr>
<th>PSW-Key Mask</th>
<th>PSW Key</th>
<th>R</th>
<th>A</th>
<th>P</th>
</tr>
</thead>
</table>

0 16 24 28 31

In the 24-Bit or 31-Bit Addressing Mode

8

<table>
<thead>
<tr>
<th>All Zeros</th>
</tr>
</thead>
</table>

0 31

9

<table>
<thead>
<tr>
<th>Bits 33-63 of Return Address</th>
</tr>
</thead>
</table>

32 33 63

In the 64-Bit Addressing Mode

8

<table>
<thead>
<tr>
<th>Bits 0-31 of Return Address</th>
</tr>
</thead>
</table>

0 31

9

<table>
<thead>
<tr>
<th>Bits 32-63 of Return Address</th>
</tr>
</thead>
</table>

32 63
Program Status Word (PSW) – PSW Key – BSA Operation

Problem state bit
Reduced Authority bit

In the 24-Bit or 31-Bit Addressing Mode

All Zeros

In the 64-Bit Addressing Mode

Bits 0-31 of Return Address

Bits 32-63 of Return Address
When the BSA instruction is used in base authority, the following is stored in the DUCT:
- The PSW-key Mask (from CR3)
- The current PSW Key
- Problem state bit
- The return address

BSA then sets the Reduced Authority bit (RA) to 1 and loads:
- The PSW-key Mask into CR3 from operand 1
- The PSW Key from operand 1
- The branch address into the PSW
When the BSA instruction is used in reduced authority, the following is restored from the DUCT:

- The PSW-key Mask (to CR3)
- The current PSW Key (to the PSW)
- Problem state bit (to the PSW)
- The return address (to the PSW – therefore the machine branches...)

BSA then sets the Reduced Authority bit (RA) to 0
Program Status Word (PSW) – More flag bits

Instruction Address

Instruction Address (Continued)
Program Status Word (PSW) – More flag bits

- Bit 13 – Machine Check Mask
  - Controls whether the CPU is enabled for interrupts by machine check conditions

- Bit 14 – Wait State
  - If on, the machine is waiting and no instructions are processed but interrupts may take place.

- Bit 15 – Problem State
  - The machine operates in two states – problem state (used for user code) and supervisor state (used for privileged code)
  - If an attempt is made to execute a privileged instruction in problem state, then a privileged operation exception occurs. Some instructions are *semi-privileged* and may or may not be permitted to execute in problem state depending on the outcome of other flags
  - All instructions are valid in supervisor state
Program Status Word (PSW) – More flag bits

- **Bit 16-17 – Address-Space Control**
  - Determines how addresses are handled in conjunction with bit 5 (DAT) via the following table:

<table>
<thead>
<tr>
<th>5</th>
<th>16</th>
<th>17</th>
<th>DAT</th>
<th>Mode</th>
<th>Instruction Addresses</th>
<th>Logical Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Off</td>
<td>Real Mode</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Off</td>
<td>Real Mode</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Off</td>
<td>Real Mode</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Off</td>
<td>Real Mode</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>On</td>
<td>Primary-space Mode</td>
<td>Primary virtual</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>On</td>
<td>Access-register Mode</td>
<td>Primary virtual</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>On</td>
<td>Secondary-space Mode</td>
<td>Primary virtual</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>On</td>
<td>Home-space Mode</td>
<td>Home virtual</td>
</tr>
</tbody>
</table>
Program Status Word (PSW) – More flag bits

- Address Space Control
- Machine Check
- Condition Code
- Problem state
- Wait
- Instruction Address
- Instruction Address (Continued)
### Program Status Word (PSW) – PSW Key – Program Mask

![Diagram of Program Status Word](https://via.placeholder.com/150)

The Program Status Word (PSW) is a register that contains various status bits and flags. The key components are:

- **Program Mask (PM)**: A field in the PSW that determines which status flags are active. It is a 32-bit field, with bits 32 to 63 representing the mask. A bit set to 1 enables the corresponding status flag.

---

**Diagram Explanation**

- **Bits 0-3**: Contains flags `R`, `O`, `E`.
- **Bits 4-7**: Contains flags `I`, `O`, `X`.
- **Bits 8**: Contains the **Key** field.
- **Bits 12-16**: Contains flags `MWP`.
- **Bits 17-20**: Contains flags `AS`.
- **Bits 21-28**: Contains flags `CC`.
- **Bits 29-31**: Contains flags `RL`.
- **Bits 32-63**: Program Mask (PM), a 32-bit field used to enable or disable the corresponding status flags.

---

**Program Mask (PM)**: A blue box highlighting the 32-bit field from bit 32 to 63, indicating its role in enabling or disabling status flags.
Program Status Word (PSW) – Program Mask

- Bits 20-23 – Program Mask
  - Controls a set of program exceptions
  - When the corresponding bit is on, the exception results in an interrupt

<table>
<thead>
<tr>
<th>Program Mask PSW bit</th>
<th>Program Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Fixed-point overflow</td>
</tr>
<tr>
<td>21</td>
<td>Decimal overflow</td>
</tr>
<tr>
<td>22</td>
<td>HFP exponent underflow</td>
</tr>
<tr>
<td>23</td>
<td>HFP significance</td>
</tr>
</tbody>
</table>

- The Program Mask can be manipulated by using the instruction SET PROGRAM MASK (SPM)
- The contents of the Program Mask can be examined using the instruction INSERT PROGRAM MASK (IPM)
The PSW stores invaluable information about the general state of the machine during a program's execution.

The most interesting time to examine a PSW is when something goes wrong. Even a summary dump will provide the programmer with:

- The contents of the PSW
- The contents of the general purpose registers
- The next instruction to be executed
Running a job has resulted in an 0C4 ABEND occurring. The summary dump in the job may be enough information to work out what has gone wrong.
Program Status Word (PSW) – Something has gone wrong

First, look at the active load module
- In this example, the load module name is GO since the LKEDG JCL procedure was used. From this we already know that the error occurred in our load module and not in either the assembler, linkage editor nor other part of z/OS
Next, double check the information in the PSW against the other information in the summary dump.

- The PSW shows that the next instruction address to be executed is X'7FF6'.
- This agrees with the data in the dump showing the address of the load module (X'7FF0') and the offset into the load module (X'0006').
Program Status Word (PSW) – Something has gone wrong

- The data at the PSW shows the instructions which were, are being, and will be executed
Examining the program listing at offset 6 shows where the error occurred. Using the data at the PSW and looking at the machine code generated by HLASM in the listing confirms this and that so far our diagnosis of the problem is correct.
Program Status Word (PSW) – Something has gone wrong

Examining the program listing at offset 6 shows where the error occurred. Using the data at the PSW and looking at the machine code generated by HLASM in the listing confirms this and that so far our diagnosis of the problem is correct.

We now know the instruction which caused the error was:

58F0 C006 → L R15,RET_CODE
Program Status Word (PSW) – Something has gone wrong

At this stage of our debugging we know:
- The load module name that caused the error
- The offset into the load module at which the error occurred

We have also double-checked that what was printed in the summary dump is confirmed by the data at the PSW.

Examining the instruction at fault, we determine the following:
- 58F0 C006 → L R15,RET_CODE
- 58 – OPCODE = LOAD
- F – Register 15, the register to be loaded
- 0 – Index register (unused since it has a value of 0)
- C – Base register is register 12
- 006 – Displacement from the base register from which the data will be loaded

So, the instruction is attempting to load register 15 with the contents of memory at an offset of 6 from register 12.
The summary dump also shows the contents of the general purpose registers.

The value in register 12 is X'FD000008'.

The instruction at fault is attempting to load a value from address X'FD00000E' – which is unaddressable by our program and therefore the cause of the error.

Note that the value of register 12 is the same as the value of register 0...
Looking back through the program, we can see that register 12 was loaded with the value of register 0 during program startup.

It looks as if the programmer made a typo and instead of using LR 12,0 should have used BALR 12,0 in order to load the address of the next instruction into register 12. This would make sense since they are using register 12 to establish addressability to the program's data.

Correcting this mistake fixes the program.
Summary

- Introductory topics
  - Computer organisation and z/Architecture
  - Building programs on z Systems
  - Working with HLASM

- Programming in Assembler
  - Loading, storing and moving data
  - Manipulating data – logic and arithmetic
  - Making decisions
  - Branching and looping
  - Reading Principles of Operation

- Reading the HLASM Listing

- The PSW and an introduction to debugging assembler programs
Where can I get help?

- z/OS V2R1 Elements and Features
  http://www.ibm.com/systems/z/os/zos/bkserv/v2r1pdf/#IEA

- HLASM Programmer's Guide (SC26-4941-06)

- HLASM Language Reference (SC26-4940-06)

- z/Architecture Principles of Operation
  http://www.ibm.com/support/docview.wss?uid=isg2b9de5f05a9d57819852571c500428f9a
Putting it all together
Putting it all together

- The following slides show a small demo program which determines whether or not an employee is eligible for a pay increase

- The slides are ordered as:
  - JCL to assemble, bind and run the program called SALARY
  - Assembler source code for the SALARY program
  - Job output from the program
Putting it all together

JCL to assemble, bind and run SALARY program

```assembly
//XXXXXXX JOB NOTIFY=&SYSUID
//S1 EXEC PGM=ASMA90
// SET PRGNM=SALARY
/*****************************************************************************/
// SET SRCLIB=&SYSUID..STAGE1.ASM0.ANSWERS
//SYSPRINT DD SYSOUT=* 
//SYSLIN DD DSN=&&TEMP,DISP=,(PASS),SPACE=(CYL,1)
//SYSLIN DD DSN=&SRCLIB(&PRGNM),DISP=SHR
//SYSLIB DD DSN=SYS1.MACLIB,DISP=SHR
// DD DSN=PP.HLASM.ZOS201.SASMMAC1,DISP=SHR
// DD DSN=PP.HLASM.ZOS201.SASMMAC2,DISP=SHR
//S2 EXEC LKEDG,COND=(8,LE),
// PARM.LKED='XREF,LIST,NCAL,MAP'
//SYSLIN DD DISP=OLD,DSN=*.
//SYSPRINT DD SYSOUT=* 
//GO.SYSUDUMP DD SYSOUT=* 
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/
```
Putting it all together

Assembler source code for SALARY program (1)

**********************************************************************
* PUTTING IT ALL TOGETHER PROGRAM
**********************************************************************

* The purpose of this small demo program is to demonstrate some small
* parts of assembler programming.
* The demo pretends that it has been passed an employee record via
* register 1.
* It will copy this record to some working storage and then proceed
* to determine whether or not the employee is eligible for a pay
* increase by comparing the employee's annual salary to the target
* salary.
*
* The employee's annual salary is calculated as:
* 12 x (MONTHLY_PAY-BENEFITS) + BONUS
*
* ASMDREG ,
SALARY CSECT
SALARY AMODE 31
SALARY RMODE 24
* USUAL PROGRAM SETUP
STM 14,12,12(13)
BALR 12,0
USING *,12
* Point register 1 at the first employee to process...
la r1,employee_id_1
Putting it all together

Assembler source code for SALARY program (2)

* The data for the employee record is passed in from register 1.
* This small program will determine whether or not the employee
  * is eligible for a pay rise or not.
*  
  xc    employee(employee_rec_len),employee         Clear WS
  mvc   employee(employee_rec_len),0(r1)           Copy record to WS

  * Output the name of the employee that is being processed
  *
  xc    wto_text,wto_text                             Clear text buffer
  mvc   wto_text(l'process_text),process_text
  mvc   wto_text+l'process_text(l'employee_name),employee_name

  * Calculate the amount of text to output
  lhi   r5,l'process_text+l'employee_name
  sth   r5,wto_buf                                     Store length in buffer
  la    r5,wto_buf                                     Load address of buffer
  WTO   TEXT=(5)                                       Output text

  * Calculate employee's yearly pay as 12*(MONTHLY_PAY-BENEFITS)+BONUS
  * We will use register 3 as a work register
  *  
  l     r3,employee_monthly_pay
  s     r3,employee_benefits                          MONTHLY-BENEFITS
  m     r2,=f'12'                                     Multiply by 12
  ah    r3,employee_bonus                             Add yearly bonus
  c     r3,target_salary                              Compare total with target
  bl    deserves_increase                             Deserves increase
Putting it all together

Assembler source code for SALARY program (3)

```
WTO   'Employee has matched or exceeded target salary'
b     resume_code
deserves_increase dc 0h
WTO   'Employee deserves a pay increase'
*
* Return to the caller of the program
*
resume_code dc 0h
LM    14,12,12(13)
*
XR    15,15
BR    14
```

* *********************************************************************************
* END OF PROGRAM - DATA follows
* *********************************************************************************

```
WTO_BUF              DC H'0'
WTO_TEXT             DS CL256
PROCESS_TEXT         DC C'Processing employee '
* SALARY SCHEME DATA
TARGET_SALARY        DC F'24000'             TARGET SALARY FOR COMPANY
* EMPLOYEE RECORD STRUCTURE
EMPLOYEE             DC 0F
EMPLOYEE_NAME        DS CL40                    EMPLOYEE'S NAME
EMPLOYEE_MONTHLY_PAY DS F                       VALUE OF MONTHLY PAY
EMPLOYEE_BONUS       DS H                       YEARLY BONUS AMOUNT
EMPLOYEE_BENEFITS    DS F                       MONTHLY BENEFITS
EMPLOYEE_REC_LEN     EQU *-EMPLOYEE              SIZE OF EMPLOYEE RECORD
```
Putting it all together

Assembler source code for SALARY program (4)

* EMPLOYEE EXAMPLE DATA

EMPLOYEE_ID_1        DC    0F
DC CL40'BOB SMITH'
DC F'2000'
DC H'1000'
DC F'50'

LTORG ,
END
# Putting it all together

Job output for assembling, binding and running SALARY program (1)

<table>
<thead>
<tr>
<th>JOB</th>
<th>JOBNO</th>
<th>JOBCLASS</th>
<th>STEP</th>
<th>NAME</th>
<th>PROGRAM</th>
<th>RC</th>
<th>RELR</th>
<th>CPU</th>
<th>SRB</th>
<th>CLOCK</th>
<th>SERV</th>
<th>PG</th>
<th>PAGE</th>
<th>SWAP</th>
<th>VIO</th>
<th>SWAPS</th>
<th>STEPNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB47759</td>
<td>S1</td>
<td>INIT</td>
<td>00</td>
<td>95</td>
<td>.00</td>
<td>.00</td>
<td>615</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOB47759</td>
<td>S2</td>
<td>LKED</td>
<td>00</td>
<td>28</td>
<td>.00</td>
<td>.00</td>
<td>191</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOB47759</td>
<td>S2</td>
<td>GO</td>
<td>00</td>
<td>4</td>
<td>.00</td>
<td>.00</td>
<td>51</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21.20.35 JOB47759 +Processing employee BOB SMITH
21.20.35 JOB47759 +Employee has matched or exceeded target salary

21.20.35 JOB47759 -XXXXXX S2 GO 00 4 .00 .00 .00 51 0 0 0 0 0 3

TOTAL CPU TIME= .00 TOTAL ELAPSED TIME= .00

0------ JES2 JOB STATISTICS ------
- 29 APR 2014 JOB EXECUTION DATE
- 24 CARDS READ
- 669 SYSOUT PRINT RECORDS
- 0 SYSOUT PUNCH RECORDS
- 37 SYSOUT SPPOOL KBYTES
- 0.00 MINUTES EXECUTION TIME

1 //XXXXXX JOB NOTIFY=&SYSUID
IEF653I SUBSTITUTION JCL - NOTIFY=XXXXXX
2 //S1 EXEC PGM=ASMA90

/**** CHANGE THE FOLLOWING LINE TO REFLECT THE PROGRAM NAME
*/
Putting it all together

Job output for assembling, binding and running SALARY program (2)

3  // SET PRGNM=SALARY
   /// ************************************************************************
   /// ************************************************************************
   /// ************************************************************************
4  // SET   SRCLIB=&SYSUID..STAGE1.ASM0.ANSWERS
   IEFC653I SUBSTITUTION JCL - SRCLIB=XXXXXXX.STAGE1.ASM0.ANSWERS
5  //SYSPRINT DD SYSOUT=*  
6  //SYSLIN DD DSN=&&TEMP,DISP=(,PASS),SPACE=(CYL,1)
7  //SYSLIB DD DSN=&SRCLIB(&PRGNM),DISP=SHR
   IEFC653I SUBSTITUTION JCL - DSN=XXXXXXX.STAGE1.ASM0.ANSWERS(SALARY),DISP=SHR
8  //SYSLIB DD DSN=SYS1.MACLIB,DISP=SHR
   //                DD DSN=PP.HLASM.ZOS201.SASMMAC1,DISP=SHR
9  //                DD DSN=PP.HLASM.ZOS201.SASMMAC2,DISP=SHR
10 //                EXEC LKEDG,COND=(8,LE),
11 //                     PARM.LKED='XREF,LIST,NCAL,MAP'
12 XXLKED EXEC PGM=HENLH096,PARM='SIZE=(384K,96K),XREF,LIST,NCAL',  00033302
   XX REGION=512K             00100000
13 //SYSPRINT DD SYSOUT=*  
14 //SYSSPRINT DD SYSOUT=A   00100000
15 //SYSLIN DD DISP=OLD,DSN=*.S1.SYSLIN
   // SYSLIN DD DDNAME=SYSLIN
16 //XXSYSMOD DD DSN=&&GOSET(GO),SPACE=(1024,(50,20,1)),
   //                UNIT=SYSDA,DISP=(MOD,PASS)
17 //XXSYSUT1 DD UNIT=(SYSDA,SEP=(SYSMOD,SYSLIN)),
   //                SPACE=(1024,(200,20))
18 //XXGO EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED)  00045000
19 //GO.SYSUDUMP DD SYSOUT=*  
   /// ************************************************************************
   /// ************************************************************************
   /// ************************************************************************
STMT NO.  MESSAGE
Putting it all together

Job output for assembling, binding and running SALARY program (3)

11 IEFC001I PROCEDURE LKEDG WAS EXPANDED USING SYSTEM LIBRARY SYS1.PROCLIB
14 IEFC648I INVALID DISP FIELD - PASS SUBSTITUTED
ICH70001I XXXXXXX LAST ACCESS AT 21:00:58 ON TUESDAY, APRIL 29, 2014
IEF236I ALLOC. FOR XXXXXXX S1
IEF237I JES2 ALLOCATED TO SYSPRINT
IGD101I SMS ALLOCATED TO DDNAME (SYSIN )
   DSN (SYS14119.T212035.RA000.XXXXXX.TEMP.H01 )
   STORCLAS (STANDARD) MGMTCLAS ( ) DATACLAS ( )
   VOL SER NOS= V10
IGD103I SMS ALLOCATED TO DDNAME SYSLIB
IEF237I ADA1 ALLOCATED TO SYSLIB
IEF237I ADA1 ALLOCATED TO
IEF237I ADA1 ALLOCATED TO
IEF237I ADA1 ALLOCATED TO
IEF142I XXXXXXX S1 - STEP WAS EXECUTED - COND CODE 0000
IEF285I XXXXXXX.XXXXXX.JOB47759.D0000101.? SYOUT
IGD106I SYS14119.T212035.RA000.XXXXXX.TEMP.H01 PASSED, DDNAME=SYSLIN
IGD104I XXXXXXX.STAGE1.ASM0.ANSWERS RETAINED, DDNAME=SYSLIB
IEF285I SYS1.MACLIB KEPT
IEF285I PP.HLASM.ZOS201.SASMMAC1 KEPT
IEF285I PP.HLASM.ZOS201.SASMMAC2 KEPT
IEF285I PP.HLASM.ZOS201.SASMMAC3 KEPT
IEF373I STEP/S1 /START 2014119.2120
IEF032I STEP/S1 /STOP 2014119.2120
CPU: 0 HR 00 MIN 00.01 SEC SRB: 0 HR 00 MIN 00.00 SEC
ATB- REAL: 36K SLOTS: OK
VIRT- ALLOC: 6M SHRD: 0M
IEF236I ALLOC. FOR XXXXXXX LKED S2
IEF237I JES2 ALLOCATED TO SYSPRINT
Putting it all together

Job output for assembling, binding and running SALARY program (4)

IGD103I SMS ALLOCATED TO DDNAME SYSLIN
IGD101I SMS ALLOCATED TO DDNAME (SYSLMOD )
  DSN (SYS14119.T212035.RA000.XXXXX.XXGOSET.H01 )
  STORCLAS (STANDARD) MGMTCLAS ( ) DATACLAS ( )
  VOL SER NOS= V10
IGD101I SMS ALLOCATED TO DDNAME (SYST1 )
  DSN (SYS14119.T212035.RA000.XXXXX.XXGOSET.H01 )
  STORCLAS (STANDARD) MGMTCLAS ( ) DATACLAS ( )
  VOL SER NOS= V10
IEF142I XXXXXXX LKED S2 - STEP WAS EXECUTED - COND CODE 0000
IEF285I   XXXXXXX.XXXXXXX.JOB47759.D0000102.?          SYSOUT
IGD101I SYS14119.T212035.RA000.XXXXX.XXGOSET.H01      PASSED,   DDNAME=SYSLIN
IGD101I SYS14119.T212035.RA000.XXXXX.XXGOSET.H01      PASSED,   DDNAME=SYSLMOD
IGD105I SYS14119.T212035.RA000.XXXXX.XXR10118176      DELETED,   DDNAME=SYST1
IEF373I STEP/LKED    /START 2014119.2120
IEF032I STEP/LKED    /STOP  2014119.2120
CPU:     0 HR  00 MIN  00.00 SEC    SRB:     0 HR  00 MIN  00.00 SEC
VIRT:   100K  SYS:   268K  EXT:     1772K  SYS:    10376K
ATB- REAL:                     0K  SLOTS:                     0K
VIRT- ALLOC:       0M SHRD:       0M
IEF236I ALLOC. FOR XXXXXXX GO S2
IGD103I SMS ALLOCATED TO DDNAME PGM=*.*.DD
IEF237I JES2 ALLOCATED TO SYSUDUMP
Processing employee BOB SMITH
Employee has matched or exceeded target salary
IGD104I SYS14119.T212035.RA000.XXXXX.XXGOSET.H01      RETAINED, DDNAME=PGM=*.*.DD
IEF285I   XXXXXXX.XXXXXXX.JOB47759.D0000103.?          SYSOUT
IEF373I STEP/GO      /START 2014119.2120
IEF032I STEP/GO      /STOP  2014119.2120
CPU:     0 HR  00 MIN  00.00 SEC    SRB:     0 HR  00 MIN  00.00 SEC
Putting it all together

Job output for assembling, binding and running SALARY program (5)

| VIRT:     8K | SYS: 252K | EXT: OK | SYS: 10440K |
| ATB- REAL: OK | SLOTS: OK | |
| VIRT- ALLOC: OM | SHRD: OM |

IGD105I SYS14119.T212035.RA000.XXXXXXX.TEMP.H01 DELETED, DDNAME=SYSLIN
IGD105I SYS14119.T212035.RA000.XXXXXXX.GOSET.H01 DELETED, DDNAME=SYSLMOD
IEF375I JOB/XXXXXXX /START 20141119.2120
IEF033I  JOB/XXXXXXX /STOP  20141119.2120

CPU: 0 HR 00 MIN 00.01 SEC SRB: 0 HR 00 MIN 00.00 SEC

Options for this Assembly

0  NOADATA
   ALIGN
   NOASA
   NOBATCH
   CODEPAGE(047C)
   NOCOMPAT
   NODBCS
   NODECK
   DXREF
   ESD
   NOEXIT
   FLAG(0,ALIGN,NOCONT,EXLITW,NOIMPLEN,NOPAGE0,PUSH,RECORD,NOSUBSTR,USING0)
   NOFOLD
   NOGOFF
   NOINFO
Putting it all together

Job output for assembling, binding and running SALARY program (6)

```
LANGUAGE(EN)
NOLIBMAC
LINECOUNT(60)
LIST(121)
MACHINE(,NOLIST)
MXREF(SOURCE)
OBJECT
OPTABLE(UNI,NOLIST)
NOPCONTROL
NOPESTOP
NOPROFILE
NORA2
NORENT
RLD
RXREF
SECTALGN(8)
SIZE(MAX)
NOSUPRWARN
SYSPARM()
NOTERM
NOTEST
THREAD
NOTRANSLATE
TYPECHECK(MAGNITUDE,REGISTER)
USING(NOLIMIT,MAP,NOWARN)
NOWORKFILE
XREF(SHORT,UNREFS)
```

No Overriding DD Names

---

External Symbol Dictionary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Id</th>
<th>Address</th>
<th>Length</th>
<th>Owner Id</th>
<th>Flags</th>
<th>Alias-of</th>
</tr>
</thead>
</table>

Page 2

HLASM R6.0 2014/04/29 21.20

© 2015 IBM Corporation
Putting it all together

Job output for assembling, binding and running SALARY program (7)

0          PC 00000001 00000000 00000000             00
SALARY    SD 00000002 00000000 000002C4             02

Active Usings: None

0  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                  HLASM R6.0  2014/04/29 21.20
0                                      1 **********************************************************************
2 * PUTTING IT ALL TOGETHER PROGRAM
3 ********************************************************************************
4 *
5 * The purpose of this small demo program is to demonstrate some small
6 * parts of assembler programming.
7 * The demo pretends that it has been passed an employee record via
8 * register 1.
9 * It will copy this record to some working storage and then proceed
10 * to determine whether or not the employee is eligible for a pay
11 * increase by comparing the employee's annual salary to the target
12 * salary.
13 *
14 * The employee's annual salary is calculated as:
15 *    12 x (MONTHLY_PAY-BENEFITS) + BONUS
16 *
17 *
18 * ASMDREG ,
19+         PUSH  PRINT 01-ASMDR
119+         POP   PRINT 01-ASMDR
000000 00000 002C4 120 SALARY CSECT 120 SALARY CSECT
121 SALARY AMODE 31
122 SALARY RMODE 24
123 * USUAL PROGRAM SETUP
000000 90EC D00C 000C 124 STM 14,12,12(13)
000004 05C0 125 BALR 12,0
Putting it all together

Job output for assembling, binding and running SALARY program (8)

R:C 00006

126  USING *,12
127  * Point register 1 at the first employee to process...
128       la  rl,employee_id_1
129  *
130  * The data for the employee record is passed in from register 1.
131  * This small program will determine whether or not the employee
132  * is eligible for a pay rise or not.
133  *
134     xc  employee(employee_rec_len),employee               Clear WS
135  mvc employee(employee_rec_len),0(r1)   Copy record to WS
136  *
137  * Output the name of the employee that is being processed
138  *
139     xc  wto_text,wto_text                                   Clear text buffer
140  mvc wto_text(l'process_text),process_text               Copy text buffer
141  mvc wto_text+l'process_text(l'employee_name),employee_name
142  * Calculate the amount of text to output
143     lhi  r5,l'process_text+l'employee_name
144     sth  r5,wto_buf                                         Store length in buffer
145     la  r5,wto_buf                                          Load address of buffer
146     WTO TEXT=(5)                                         Output text

148+     CNOP 0,4                                         01-WTO
149+     BRAS 1,IHB0002A                                     BRANCH AROUND MESSAGE  @LCC 01-WTO
150+     DC  AL2(8)                                        TEXT LENGTH  @YA17152 01-WTO
151+     DC  B'00000000000001000'                        MCSFLAGS  @L5A 01-WTO
152+     DC  AL1(2)                                        MESSAGE TEXT ADDRESS  @L5A 01-WTO
153+     DC  AL1(2)                                        VERSION LEVEL  @PJC 01-WTO
154+     DC  B'0000000000010000'                        MISCELLANEOUS FLAGS  @L2A 01-WTO
155+     DC  AL1(0)                                       REPLY LENGTH  @L2A 01-WTO

Active Usings: SALARY+X'6',R12
### Putting it all together

Job output for assembling, binding and running SALARY program (9)

<table>
<thead>
<tr>
<th>Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
<th>HLASM R6.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td>DC AL1(104)</td>
<td>2014/04/29</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>156+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>068</td>
<td>0080</td>
<td></td>
<td></td>
<td>DC AL2(0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>157+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00048</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC AL4(0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>159+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00040</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC AL4(0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>160+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000050</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC AL4(0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>161+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000054</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC AL4(0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>162+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000056</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC AL4(0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>163+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000058</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>164+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000060</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>165+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000068</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>166+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00006A</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>167+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00006C</td>
<td>4040404040404040</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>168+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000074</td>
<td>4040404040404040</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>169+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00007C</td>
<td>4040404040404040</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>170+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000084</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>171+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000088</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>172+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00008C</td>
<td>4040404040404040</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>173+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000094</td>
<td>4040404040404040</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>174+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00009C</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>175+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000A0</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>176+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000A4</td>
<td>0000000000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>177+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000A8</td>
<td>18E1</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>178+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000AA</td>
<td>18FF</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>179+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000AC</td>
<td>4AF1  0000</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>180+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000B0</td>
<td>1AEF</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>181+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000B2</td>
<td>5050  1004</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>182+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000B6</td>
<td>0A23</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>183+</td>
<td></td>
<td></td>
<td>DC B'0000000010000000'</td>
<td></td>
</tr>
</tbody>
</table>

© 2015 IBM Corporation
Putting it all together

Job output for assembling, binding and running SALARY program (10)

185  * Calculate employee's yearly pay as 12*(MONTHLY_PAY-BENEFITS)+BONUS
186  * We will use register 3 as a work register
187  *

0000B8 5830 C27A 00280 188          l     r3,employee_monthly_pay
0000BC 5B30 C282 00288 189          s     r3,employee_benefits       MONTHLY-BENEFITS
0000C0 5C20 C2BA 002C0 190          m     r2,=f'12'                  Multiply by 12
0000C4 4A30 C27E 00284 191          ah    r3,employee_bonus          Add yearly bonus
0000C8 5930 C24E 00254 192          c     r3,target_salary           Compare total with target
0000CC 4740 C106 0010C 193          bl    deserves_increase

0000D0 0032 194          WTO   'Employee has matched or exceeded target salary'
0000D0 A715 001B 196+         CNOP  0,4                                                      01-WTO
0000D4 0032 197+         BRAS  1,IHB0004A               BRANCH AROUND MESSAGE      @LCC 01-WTO
0000D6 0000 198+         DC    AL2(50)                  TEXT LENGTH            @YA17152 01-WTO
0000D8 C594979396A88585 199+         DC    B'0000000000000000'      MCSFLAGS                        01-WTO
0000E0 408881A2409481A3 200+         DC    C'Employee has matched or exceeded target salary'       X01-WTO

000106 0A23 201+IHB0004A DS  OH 202+         SVC   35                       ISSUE SVC 35               @L6A 01-WTO
00010B 47F0 C130 00136 203          b     resume_code
000110 0024 204 deservies_increase dc  0h
000112 0000 205          WTO   'Employee deserves a pay increase'

Active Usings: SALARY+X'6',R12

0  Loc  Object Code Addr1 Addr2 Stmt Source Statement       HLA MS R6.0  2014/04/29 21.20
0000114 C594979396A88585 211+         DC    C'Employee deserves a pay increase'       X01-WTO
000011C 408485A28599A585 212+IHB0006A DS  OH                  MESSAGE TEXT       @L6C 01-WTO
Putting it all together

Job output for assembling, binding and running SALARY program (11)

000134 0A23 213+ SVC 35 ISSUE SVC 35 @L6A 01-WTO
000136 214 *
000136 215 * Return to the caller of the program
216 *
000136 217 resume_code dc 0h
000136 98EC D00C 0000C 218 LM 14,12,12(13)
000136 219 *
00013A 17FF 220 XR 15,15
00013C 07FE 221 BR 14
00013E 0000 222 *----------------------------------------------------------------------
000140 223 * END OF PROGRAM - DATA FollowS
000240 D799968385A2A289 224 *----------------------------------------------------------------------
000254 00005DC0 225 *
000258 226 WTO_BUF DC H'0'
000258 227 WTO_TEXT DS CL256
000240 228 PROCESS_TEXT DC C'Processing employee '\n000254 229 * SALARY SCHEME DATA
000254 230 TARGET_SALARY DC F'24000' TARGET SALARY FOR COMPANY
000254 231 * EMPLOYEE RECORD STRUCTURE
000258 232 EMPLOYEE DC 0F
000258 233 EMPLOYEE_NAME DS CL40
000254 234 EMPLOYEE_MONTHLY_PAY DS F
000254 235 EMPLOYEE_BONUS DS H
000258 236 EMPLOYEE_BENEFITS DS F
000254 237 EMPLOYEE_REC_LEN EQU -*EMPLOYEE SIZE OF EMPLOYEE RECORD
000254 238 * EMPLOYEE EXAMPLE DATA
000258 239 EMPLOYEE_ID_1 DC 0F
000258 C2D6C240B2D4C9E3 240 DC CL40'BOB SMITH'
0002B4 000007D0 241 DC F'2000'
0002B8 03E8 242 DC H'1000'
0002BA 0000
Putting it all together

Job output for assembling, binding and running SALARY program (12)

```
0002BC 00000032 243 DC F'50'
0002CD 0000000C 244 LTORG ,
0002C0 0000000C 245 =f'12'
246 END
```

Table: Ordinary Symbol and Literal Cross Reference

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Length</th>
<th>Value</th>
<th>Id</th>
<th>R</th>
<th>Type</th>
<th>Asm</th>
<th>Program</th>
<th>Defn</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>deserves_increase</td>
<td>2</td>
<td>00000010C</td>
<td>00000002</td>
<td>H</td>
<td>H</td>
<td>204</td>
<td>193B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>4</td>
<td>00000258</td>
<td>00000002</td>
<td>F</td>
<td>F</td>
<td>232</td>
<td>134M 134</td>
<td>135M</td>
<td>237</td>
</tr>
<tr>
<td>EMPLOYEE_BENEFITS</td>
<td>4</td>
<td>00000288</td>
<td>00000002</td>
<td>F</td>
<td>F</td>
<td>236</td>
<td>189</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_BONUS</td>
<td>2</td>
<td>00000284</td>
<td>00000002</td>
<td>H</td>
<td>H</td>
<td>235</td>
<td>191</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_ID_1</td>
<td>4</td>
<td>0000028C</td>
<td>00000002</td>
<td>F</td>
<td>F</td>
<td>239</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_MONTHLY_PAY</td>
<td>4</td>
<td>00000280</td>
<td>00000002</td>
<td>F</td>
<td>F</td>
<td>234</td>
<td>188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_NAME</td>
<td>40</td>
<td>00000258</td>
<td>00000002</td>
<td>C</td>
<td>C</td>
<td>233</td>
<td>141 141</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_REC_LEN</td>
<td>1</td>
<td>00000034</td>
<td>00000002</td>
<td>A</td>
<td>U</td>
<td>237</td>
<td>134 135</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHB0002A</td>
<td>2</td>
<td>000000A8</td>
<td>00000002</td>
<td>H</td>
<td>H</td>
<td>177</td>
<td>149B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHB0004A</td>
<td>2</td>
<td>00000106</td>
<td>00000002</td>
<td>H</td>
<td>H</td>
<td>201</td>
<td>197B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHB0006A</td>
<td>2</td>
<td>00000134</td>
<td>00000002</td>
<td>H</td>
<td>H</td>
<td>212</td>
<td>208B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCESS_TEXT</td>
<td>20</td>
<td>00000240</td>
<td>00000002</td>
<td>C</td>
<td>C</td>
<td>228</td>
<td>140 140</td>
<td>141M</td>
<td>143</td>
</tr>
<tr>
<td>resume_code</td>
<td>2</td>
<td>00000136</td>
<td>00000002</td>
<td>H</td>
<td>H</td>
<td>217</td>
<td>203B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>00000001</td>
<td>00000001</td>
<td>A</td>
<td>U</td>
<td>28</td>
<td>128M 135</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>00000002</td>
<td>00000001</td>
<td>A</td>
<td>U</td>
<td>29</td>
<td>190M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>00000003</td>
<td>00000001</td>
<td>A</td>
<td>U</td>
<td>30</td>
<td>188M 189M</td>
<td>191M</td>
<td>192</td>
</tr>
</tbody>
</table>
### Job output for assembling, binding and running SALARY program (13)

```
0002BC 00000032 243                      DC F'50'
0002CD 00000000C 244          LTORG ,
0002CD 00000000C 245                =f'12'
0002CD 00000000C 246          END
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Length</th>
<th>Value</th>
<th>Id</th>
<th>R Type</th>
<th>Asm</th>
<th>Program</th>
<th>Defn References</th>
</tr>
</thead>
<tbody>
<tr>
<td>deserves_increase</td>
<td>2</td>
<td>0000010C</td>
<td>00000002</td>
<td>H  H</td>
<td>204</td>
<td>193B</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>4</td>
<td>00000258</td>
<td>00000002</td>
<td>F  F</td>
<td>232</td>
<td>134M 134 135M 237</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_BENEFITS</td>
<td>4</td>
<td>00000288</td>
<td>00000002</td>
<td>F  F</td>
<td>236</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_BONUS</td>
<td>2</td>
<td>00000284</td>
<td>00000002</td>
<td>H  H</td>
<td>235</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_ID_1</td>
<td>4</td>
<td>0000028C</td>
<td>00000002</td>
<td>F  F</td>
<td>239</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_MONTHLY_PAY</td>
<td>4</td>
<td>00000280</td>
<td>00000002</td>
<td>F  F</td>
<td>234</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_NAME</td>
<td>40</td>
<td>00000258</td>
<td>00000002</td>
<td>C  C</td>
<td>233</td>
<td>141 141 141 143</td>
<td></td>
</tr>
<tr>
<td>EMPLOYEE_REC_LEN</td>
<td>1</td>
<td>00000034</td>
<td>00000002</td>
<td>A  U</td>
<td>237</td>
<td>134 135</td>
<td></td>
</tr>
<tr>
<td>IHB0002A</td>
<td>2</td>
<td>000000A8</td>
<td>00000002</td>
<td>H  H</td>
<td>177</td>
<td>149B</td>
<td></td>
</tr>
<tr>
<td>IHB0004A</td>
<td>2</td>
<td>00000106</td>
<td>00000002</td>
<td>H  H</td>
<td>201</td>
<td>197B</td>
<td></td>
</tr>
<tr>
<td>IHB0006A</td>
<td>2</td>
<td>00000134</td>
<td>00000002</td>
<td>H  H</td>
<td>212</td>
<td>208B</td>
<td></td>
</tr>
<tr>
<td>PROCESS_TEXT</td>
<td>20</td>
<td>00000240</td>
<td>00000002</td>
<td>C  C</td>
<td>228</td>
<td>140 140 141M 143</td>
<td></td>
</tr>
<tr>
<td>resume_code</td>
<td>2</td>
<td>00000136</td>
<td>00000002</td>
<td>H  H</td>
<td>217</td>
<td>203B</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>00000001</td>
<td>00000001</td>
<td>A  U</td>
<td>28</td>
<td>128M 135</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>00000002</td>
<td>00000001</td>
<td>A  U</td>
<td>29</td>
<td>190M</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>00000003</td>
<td>00000001</td>
<td>A  U</td>
<td>30</td>
<td>188M 189M 191M 192</td>
<td></td>
</tr>
</tbody>
</table>
```
## Putting it all together

Job output for assembling, binding and running SALARY program (14)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>1 00000005 00000001 A U</td>
<td>32 143M 144 145M</td>
<td></td>
</tr>
<tr>
<td>SALARY</td>
<td>1 00000000 00000002 J</td>
<td>120 121 122</td>
<td></td>
</tr>
<tr>
<td>TARGET_SALARY</td>
<td>4 0000254 0000002 F F</td>
<td>230 192</td>
<td></td>
</tr>
<tr>
<td>WTO_BUF</td>
<td>2 000013E 0000002 H H</td>
<td>226 144M 145</td>
<td></td>
</tr>
<tr>
<td>WTO_TEXT</td>
<td>256 0000140 0000002 C C</td>
<td>227 139M 139 140M 141M</td>
<td></td>
</tr>
<tr>
<td>=f'I2'</td>
<td>4 00002C0 0000002 F</td>
<td>245 190</td>
<td></td>
</tr>
</tbody>
</table>

Unreferenced Symbols Defined in CSECTs

- Defn Symbol

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR0</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>AR1</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>AR10</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>AR11</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>AR12</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>AR13</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>AR14</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>AR15</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>AR2</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>AR3</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>AR4</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>AR5</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>AR6</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>AR7</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>AR8</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>AR9</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>CR0</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>CR1</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>CR10</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>CR11</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>CR12</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>CR13</td>
<td>97</td>
<td></td>
</tr>
</tbody>
</table>

Page 7
Putting it all together

Job output for assembling, binding and running SALARY program (15)

98  CR14
99  CR15
86  CR2
87  CR3
88  CR4
89  CR5
90  CR6
91  CR7
92  CR8
93  CR9
46  FR0
47  FR1
56  FR10
57  FR11
58  FR12
59  FR13
60  FR14
61  FR15
48  FR2
49  FR3
50  FR4
51  FR5
52  FR6
53  FR7
54  FR8
55  FR9
27  R0
37  R10
38  R11
39  R12
40  R13
Putting it all together

Job output for assembling, binding and running SALARY program (16)
Putting it all together

Job output for assembling, binding and running SALARY program (17)

41  R14
42  R15
1 - Defn  Symbol
-  Defn  Symbol
0  31  R4
33  R6
34  R7
35  R8
36  R9
65  VR0
66  VR1
75  VR10
76  VR11
77  VR12
78  VR13
79  VR14
80  VR15
67  VR2
68  VR3
69  VR4
70  VR5
71  VR6
72  VR7
73  VR8
74  VR9
- -
Putting it all together

Job output for assembling, binding and running SALARY program (18)
Putting it all together

Job output for assembling, binding and running SALARY program (19)

L3 PP.HLASM.ZOS201.SASMMAC2 33SY02
SYSLIN SYS14119.T212035.RA000.XXXXXX.TEMP.H01
SYSPRINT Xxxxxx.Xxxxxx.JOB47759.D0000101.?

64840K allocated to Buffer Pool       Storage required     384K
91 Primary Input Records Read         3707 Library Records Read                  0 Work File Reads
0 ASMAOPT Records Read                384 Primary Print Records Written         0 Work File Writes
10 Object Records Written                0 ADATA Records Written

0Assembly Start Time: 21.20.35 Stop Time: 21.20.35 Processor Time: 00.00.00.0068
Return Code 000

1z/OS V2 R1 BINDER     21:20:35 TUESDAY APRIL 29, 2014
BATCH EMULATOR  JOB(XXXXX ) STEP(S2      ) PGM= HEWLH096  PROCEDURE(LKED    )
IEW2278I B352 INVOCATION PARAMETERS - XREF,LIST,NCAL,MAP
IEW2650I 5102 MODULE ENTRY NOT PROVIDED. ENTRY DEFAULTS TO SECTION SALARY.

1                         *** M O D U L E  M A P ***

---------------
CLASS  B_TEXT            LENGTH =      2C4  ATTRIBUTES = CAT,   LOAD, RMODE= 24
OFFSET =        0 IN SEGMENT 001     ALIGN = DBLWORD
---------------

SECTION CLASS                                      ------- SOURCE --------
OFFSET OFFSET  NAME                TYPE    LENGTH  DDNAME   SEQ  MEMBER
0  SALARY             CSECT       2C4  SYSLIN    01  **NULL**

1                         *** D A T A  S E T  S U M M A R Y  ***

DDNAME CONCAT FILE IDENTIFICATION
Putting it all together

Job output for assembling, binding and running SALARY program (20)

SYSLIN 01 SYSLIN.T212035.RA000.XXXXXX.TEMP.H01

*** END OF MODULE MAP ***

1 CROSS-REFERENCE TABLE

TEXT CLASS = B_TEXT

------------------------ REFERENCE -------------------------- TARGET --------------------------
CLASS OFFSET SECT/PART(ABBREV) OFFSET TYPE | SYMBOL(ABBREV) SECTION (ABBREV) OFFSET CLASS NAME |
*** NO ADDRESS CONSTANTS FOR THIS CLASS ***
*** END OF CROSS REFERENCE ***

*** OPERATION SUMMARY REPORT ***

1PROCESSING OPTIONS:

ALIASES NO
ALIGN2 NO
AMODE UNSPECIFIED
CALL NO
CASE UPPER
COMPAT UNSPECIFIED
COMPRESSION AUTO
DCBS NO
## Putting it all together

### Job output for assembling, binding and running SALARY program (21)

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNAM</td>
<td>NO</td>
</tr>
<tr>
<td>EXATTR</td>
<td>UNSPECIFIED</td>
</tr>
<tr>
<td>EXITS:</td>
<td>NONE</td>
</tr>
<tr>
<td>FILL</td>
<td>NONE</td>
</tr>
<tr>
<td>GID</td>
<td>UNSPECIFIED</td>
</tr>
<tr>
<td>HOBSET</td>
<td>NO</td>
</tr>
<tr>
<td>INFO</td>
<td>NO</td>
</tr>
<tr>
<td>LET</td>
<td>04</td>
</tr>
<tr>
<td>LINECT</td>
<td>060</td>
</tr>
<tr>
<td>LIST</td>
<td>SUMMARY</td>
</tr>
<tr>
<td>LISTPRIV</td>
<td>NO</td>
</tr>
<tr>
<td>LONGPARM</td>
<td>NO</td>
</tr>
<tr>
<td>MAP</td>
<td>YES</td>
</tr>
<tr>
<td>MAXBLK</td>
<td>032760</td>
</tr>
<tr>
<td>MODMAP</td>
<td>NO</td>
</tr>
<tr>
<td>MSGLEVEL</td>
<td>00</td>
</tr>
<tr>
<td>OVLY</td>
<td>NO</td>
</tr>
<tr>
<td>PRINT</td>
<td>YES</td>
</tr>
<tr>
<td>RES</td>
<td>NO</td>
</tr>
<tr>
<td>REUSABILITY</td>
<td>UNSPECIFIED</td>
</tr>
<tr>
<td>RMODE</td>
<td>UNSPECIFIED</td>
</tr>
<tr>
<td>SIGN</td>
<td>NO</td>
</tr>
<tr>
<td>STORENX</td>
<td>NOREPLACE</td>
</tr>
<tr>
<td>STRIPCL</td>
<td>NO</td>
</tr>
<tr>
<td>STRIPSEC</td>
<td>NO</td>
</tr>
<tr>
<td>SYMTRACE</td>
<td></td>
</tr>
<tr>
<td>TERM</td>
<td>NO</td>
</tr>
<tr>
<td>TRAP</td>
<td>ON</td>
</tr>
<tr>
<td>UID</td>
<td>UNSPECIFIED</td>
</tr>
<tr>
<td>UPCASE</td>
<td>NO</td>
</tr>
<tr>
<td>WKSPACE</td>
<td>000000K,000000K</td>
</tr>
</tbody>
</table>
Putting it all together

Job output for assembling, binding and running SALARY program (22)

XCAL     NO
XREF     YES
***END OF OPTIONS***

1SAVE OPERATION SUMMARY:
MEMBER NAME         GO
LOAD LIBRARY        SYS14119.T212035.RA000.XXXXXXXXX.GOSET.H01
PROGRAM TYPE        LOAD MODULE
VOLUME SERIAL
MAX BLOCK           32760
DISPOSITION         ADDED NEW
TIME OF SAVE        21.20.35  APR 29, 2014

1SAVE MODULE ATTRIBUTES:
AC                  000
AMODE                31
COMPRESSION         NONE
DC                    NO
EDITABLE             YES
EXCEEDS 16MB         NO
EXECUTABLE           YES
LONGPARM             NO
MIGRATABLE           YES
OL                    NO
OVLY                  NO
Putting it all together

Job output for assembling, binding and running SALARY program (23)

PACK,PRIME      NO,NO
PAGE ALIGN      NO
REFR           NO
RENT           NO
REUS           NO
RMODE          24
SCTR           NO
SIGN           NO
SSI            
SYM GENERATED  NO
TEST          NO
XPLINK         NO
MODULE SIZE (HEX) 000002C8

1 ENTRY POINT AND ALIAS SUMMARY:

NAME:            ENTRY TYPE AMODE C_OFFSET CLASS NAME        STATUS
SALARY            MAIN_EP      31 00000000 B_TEXT

*** END OF OPERATION SUMMARY REPORT ***

1z/OS V2 R1 BINDER     21:20:35 TUESDAY APRIL 29, 2014
BATCH EMULATOR JOB(XXXXXXX ) STEP(S2 ) PCM= HEWLH096 PROCEDURE(LKED )
IEW2008I 0F03 PROCESSING COMPLETED. RETURN CODE = 0.
Putting it all together

Job output for assembling, binding and running SALARY program (24)

1----------------------
MESSAGE SUMMARY REPORT
----------------------
TERMINAL MESSAGES (SEVERITY = 16)
NONE

SEVERE MESSAGES (SEVERITY = 12)
NONE

ERROR MESSAGES (SEVERITY = 08)
NONE

WARNING MESSAGES (SEVERITY = 04)
NONE

INFORMATIONAL MESSAGES (SEVERITY = 00)
2008 2278 2650

**** END OF MESSAGE SUMMARY REPORT ****