z/OS Assembler Programming - Part 3
z/Architecture and z/OS

z/OS Assembler Programming: z/Architecture and z/OS - Course Objectives

On successful completion of this course, the student, with the aid of the appropriate reference materials, should be able to:

1. Describe the major architectural changes introduced with the z/Architecture class of mainframes

2. Write Assembler programs that use the new instructions, particularly
   * Long displacement instructions
   * Relative branch instructions
   * Instructions to set and test current addressing mode
   * Instructions to load all or parts of 64-bit registers, and to shift and rotate bits within registers
   * Instructions to perform 64-bit binary arithmetic
   * Instruction to test packed decimal data for validity
   * New instructions that can set and test bits in storage or registers

3. Work with files encoded in EBCDIC, ASCII, or Unicode

4. Code, assemble, bind, and run programs that run AMODE64

5. Use new features of DROP, EQU, ORG, and CNOP

Note: the instructor may elect to skip over content in this course that is not relevant to your installation. This may change the anticipated day breaks, or even shorten the course.

But any omitted content remains in the handout for your later study, should you so wish.

z/OS Assembler Programming: z/Architecture and z/OS - Topical Outline

Day One

Day Two

Day Two, continued

Macro extensions and debugging
- Macro support (SYSSTATE, CALL, IAVR64)
- Debugging information
- Symptom dumps
- SYSUDUMPS
- Computer Exercise: Using Memory Objects 224

Day Three

Decimal Data
- Numeric characters - EBCDIC
- Numeric characters - ASCII
- Numeric characters - Unicode
- Zoned decimal data and signs

Decimal Data, continued
- Packed decimal
- PACK, PKA, PKU, UNPK, UNPKA, UNPKU, TP
- CVB, CVBY, CVBG, CVD, CVDY, CVDG
- Computer Exercise: Packed Decimal Data 245

Binary Arithmetic
- Halfword Binary Arithmetic
- Fullword and doubleword binary arithmetic
- Logical binary loads
- Other binary loads
- Logical binary arithmetic
- Computer Exercise: Binary Arithmetic 274

Day Three, continued

Boolean Instructions
- Working with bits
- OR instructions
- AND instructions
- Exclusive OR instructions
- Test Under Mask instructions
- Halfword Immediate Test instructions
- Load and Test instructions
- Zero Out Parts of a Register

Shifting and Rotating - Bits in Registers
- Shift Instructions
- Grande Shifts
- Shift Instruction Processing
- Rotate Instructions

Working With Character Strings
- Working With Character Strings in z/Architecture
- Interruptible Instructions
- CPU-Determined Unit of Processing
- More Instructions for Working With Character Strings in z/Architecture
- Additional long displacement instructions
- TRTR

Working With ASCII Data in z/OS
- Encoding Schemes
- Working With ASCII Data in z/Architecture
- Big Endian and Little Endian
- Load Reversed
- Store Reversed
- Working With ASCII Data, continued
- Computer Exercise: Supporting ASCII Data 324

Day Four

Introduction to Unicode
- Characters, Glyphs, and Fonts
- Coding Schemes and Codepages
- Unicode

Working With Unicode Strings in z/Architecture
- CUUTF, CUTFU, CLCLU, MVCLU, TROO, TROT, TRTO, TRTT
- CU24, CU21, CU42, CU41, CU12, CU14, SRSTU

The Extended-immediate Facility
- AF, AGFI, ALFI, ALGFI, CFI, CGFI, CLFI, CLGFI, SLFI, SLGFI, NIHF, NILF, XIHF, XILF, OIHF, OILF, IIHF, IILF, LGFI, LT, LTG, LBR, LGBR, LHR, LGHR, LLCR, LLGCR, LLC, LLHR, LLGHR, LLH, LLIHF, LLILF

The Dead Zone, and More
- The z/OS Address Space
- The Dead Zone
- The IEABRC macro and IEABRCX copy book
- Computer Exercise: Using IEABRCX 376

More Thoughts on AMODE64
- The Assembler and the Binder
- Program Fetch
- More Linkage Issues
- AMODE64 Linkages
- Macro support for AMODE64 programs

HLASM update
- Reference external symbols on RI instructions
- Enhancements to EQU, ORG, CNOP
- Extended mnemonics
- Address space partitioning

Day Four,continued

Newer hardware instructions
- Overview
- Load and Store instructions
- Compare and Branch, Compare and Trap, other New Compare Instructions
- New Execute Instruction
- Working with binary data - add, multiply, rotating bits in registers
- New Move instructions
- New Translate and Test instructions
- Computer Exercise: Using Compare and Branch 436

More High Level Assembler
- Additional extended mnemonics
- Mnemonic tagging

More Instructions by Facility
- Overview
- High-Word Facility instructions
- Interlocked-Access Facility instructions
- Load/Store-on-condition Facility instructions
- Distinct Operands Facility instructions

Still more instructions
- The Miscellaneous-instruction-extensions facilty
- The Load-and-trap facility
- The Interlock-access facility, 1 and 2
- The Load/Store-on-Condition facilty
- The Execution-hint facility
- The Processor-assist facility
- The Transaction-execution facilty

Appendix: Listing of ASMREPT code as supplied

Index

A Note From the Course Developer

❒ **Covering over 500 machine instructions in just a few days, <u>this could be a very boring, dull, soul-crushing course</u>**

... UNLESS ...

✗ **You <u>take an active approach</u> to absorbing the content; for example:**

- ➢ Whenever instructions are discussed, you make sure you can describe how each one works and how you might use it in your coding, personally
- ➢ Make a point of noting the instructions likely to be most useful for you; actually write out examples of how you might code them
- ➢ During the labs (machine exercises), feel free to add additional code to test instructions

❒ **IBM's mainframes have gone through immense changes over the last 60 years, getting richer - more complex - with each iteration**

For example, there are now over 1,000 machine instructions available, including floating point, timer / clock, privileged instructions, and more!

❒ **And even so, backward compatability has been a central tenet:**

Continue to improve while preserving past investments in application development

A Note From the Course Developer, 2

- **This couse is designed for experienced Assembler programmers**

  From two months after taking the intro courses, to 30+ years of application programming experience in Assembler

- **The focus of this course is on describing features introduced with z/Architecture hardware and the latest version of z/OS software that allow and enhance application development**

  It is an exploration of how to continue application development in the new world

  - ✗ New hardware instructions that are useful for application programmers
  - ✗ Expanded / enhanced Assembler features
  - ✗ Managing storage
  - ✗ Linkage conventions
  - ✗ Miscellaneous changes of interest

- **Note that the instructor may choose to skip content judged not to be relevant to your installation; the content remains in the handout as possible reference materials if you are interested**

- **To break things up a bit, we intermingle Assembler statements and / or system macros between sections about machine instructions**

- **Some things I've learned while developing this course:**

  Running code above the bar is seldom (if ever) needed

  Working with data above the bar is not always best for performance

  Using reentrant code is counter productive in non-multi-tasking environments

  - ✗ Nonetheless, these topics are each addressed to some degree in this course

IBM Reference Documents

❒ **The content of this course draws upon these IBM publications**

While the course content is focused on the details we consider most important / relevant, these publications contain all the details

You may wish to locate / download these document:

https://www.ibm.com/docs/en/module_1678991624569/pdf/SA22-7832-14.pdf
z/ Architecture Principles of Operation

https://www.ibm.com/docs/en/module_1678991624569/pdf/SA22-7871-12.pdf
Reference Summary

https://www.ibm.com/docs/en/SSENW6_1.6.0/pdf/asmg1025_pdf.pdf
HLASM 1.6 General Information

https://www.ibm.com/docs/en/SSENW6_1.6.0/pdf/asmp1024_pdf.pdf
HLASM Programmer's Guide

https://www.ibm.com/docs/en/SSENW6_1.6.0/pdf/asmr1024_pdf.pdf
HLASM Language Reference

https://www.ibm.com/docs/en/SSLTBW_3.2.0/pdf/ieaa600_v3r2.pdf
Assembler Services Guide

https://www.ibm.com/docs/en/SSLTBW_3.2.0/pdf/ieaa700_v3r2.pdf
Assembler Services Reference, Vol. 1

https://www.ibm.com/docs/en/SSLTBW_3.2.0/pdf/ieaa900_v3r2.pdf
Assembler Services Reference, Vol. 2

https://www.ibm.com/docs/en/SSLTBW_3.2.0/pdf/ieab100_v3r2.pdf
Program Management: User's Guide and Reference

https://www.ibm.com/docs/en/SSLTBW_3.2.0/pdf/idad500_v3r2.pdf
Macro Instructions for Data Sets

Section Preview

- ❒ **Introduction to the Course**
  - **General Introduction**
  - **z/Architecture**
  - **Numbers and Numeric Terms**
  - **Context and Course Flow**

General Introduction

❒ **This course is designed to introduce Assembler programmers to the 64-bit based hardware and software from IBM**

Emphasis is on the impact to applications and application programmers

✗ That is, little or no examination of system, control, I/O, or floating point facilities and instructions

✗ No examination of instructions and features used for system internals, message- or data-encryption, neural network processing, and the like

✗ No- to minimal-examination of instructions designed to be used by compilers and special runtime environments (*e.g.* Java, etc.)

✗ No examination of operational impacts

Focus is on z/OS

✗ That is, no examination of z/VM, z/VSE, z/TPF, or Linux on z

❒ **We assume the student is an experienced OS/390 or z/OS Assembler applications programmer**

Systems programmers may find this a useful introduction, but they may well need more depth than is presented here

z/Architecture

- **The IBM 64-bit mainframe has been named "z/Architecture" to contrast it to earlier mainframe architectures such as**

  S/360

  S/370

  370-XA

  ESA/370

  ESA/390

- **Although there is a clear continuity, z/Architecture also brings significant changes...**

  16 64-bit General Purpose Registers (previously 32-bit)

  16 64-bit Control Registers (well, not really new)

  16 64-bit Floating Point Registers (previously just 4 registers)

  32 128-bit Vector Registers (previously optional)

  128-bit PSW (previously 64-bit)

  Tri-modal addressing (24-bit, 31-bit, 64-bit)

  Access to very large address spaces

- **There is tremendous complexity behind the operation of z/OS, but we will use the illusion that your application program is running under one CPU in a single virtual storage**

  We begin with some basic terms about numbers ...

Numbers and Numeric Terms

❒ **The numbers possible with 64-bit integers and addresses are large, so we find we may need to refresh / introduce the proper numeric terms, at least as far as American English goes**

For counting ...

For this many digits	We use the term
9	units
99	tens
999	hundreds
9,999	thousands
9,999,999	millions
9,999,999,999	billions
9,999,999,999,999	trillions
9,999,999,999,999,999	quadrillions
9,999,999,999,999,999,999	quintillions

❒ **The largest binary number in 64 bits is, in decimal:**

0 - 18,446,744,073,709,551,615 if unsigned

-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 if signed

Numbers and Numeric Terms, 2

- **But, in terms of measurements (for example number of bits or bytes of memory or disk space) the language has gotten trickier**

- **First, we have historically used these terms**

  kilobit, kilobyte (kb, kB) - 1,024 bits or bytes

  Megabit, Megabyte (Mb, MB) - 1,048,576 bits or bytes

  Gigabit, Gigabyte (Gb, GB) - 1,073,741,824 bits or bytes

  Terabit, Terabyte (Tb, TB) - 1,099,511,627,776 bits or bytes

  Petabit, Petabyte (Pb, PB) - 1,125,899,906,842,624 bits or bytes

  Exabit, Exabyte (Eb, EB) - 1,152,921,504,606,846,976 bits or bytes

- **But these numbers are based on powers of 2, while engineers have historically worked with powers of 10 and use these terms:**

  kilobit, kilobyte (kb, kB) - 1,000 bits or bytes

  Megabit, Megabyte (Mb, MB) - 1,000,000 bits or bytes

  Gigabit, Gigabyte (Gb, GB) - 1,000,000,000 bits or bytes

  Terabit, Terabyte (Tb, TB) - 1,000,000,000,000 bits or bytes

  Petabit, Petabyte (Pb, PB) - 1,000,000,000,000,000 bits or bytes

  Exabit, Exabyte (Eb, EB) - 1,000,000,000,000,000,000 bits or bytes

- **These differences can cause real incompatibility problems in designs or standards, and thus manufacturing, costing, and pricing**

Numbers and Numeric Terms, 3

- **In 1998, the International Electrotechnical Commission (IEC) defined a standard set of terms and prefixes to be used to distinguish powers of two from powers of ten**

- **So these are the terms one should use when referencing numbers based on powers of two that describe quantities:**

  Kibibit, Kibibyte (Kib, KiB) - 1,024 bits or bytes

  Mebibit, Mebibyte (Mib, MiB) - 1,048,576 bits or bytes

  Gibibit, Gibibyte (Gib, GiB) - 1,073,741,824 bits or bytes

  Tebibit, Tebibyte (Tib, TiB) - 1,099,511,627,776 bits or bytes

  Pebibit, Pebibyte (Pib, PiB) - 1,125,899,906,842,624 bits or bytes

  Exbibit, Exbibyte (Eib, EiB) - 1,152,921,504,606,846,976 bits or bytes

- **The point of all this is that, for example, 65,538 bytes is 64KiB and 18,446,744,073,709,551,616 bytes is 16EiB, not 18EiB**

- **It is recommended that the second syllable ("bi") be pronounced as "bee"; the "bi" indicates "binary" - powers of two**

- **It is not clear if these standards will be widely adopted or used outside of technical areas, and we may mix the new with the old while we go through a period of transition**

Context and Course Flow

- **We'll start by doing a quick review of the z/OS world before 64-bit architecture**

- **Then describe the environment in the 64-bit world**

  To demonstrate the big picture of the essential differences

- **So think of the upcoming section as "before-64" followed by "after-64"**

  So we can clearly see the transitions to make

This page intentionally left almost blank.

Section Preview

- Fundamentals - A Quick Review
  - Programming Concepts
  - Source, Object, and Load Modules
  - Memory and Data Representation
  - Addresses
  - The CPU - pre-z/Architecture
  - Setting Up For Labs (Machine Exercise)
  - Machine Instruction Formats
  - Operand Addresses
  - Assembler Language and the High Level Assembler (HLASM)
  - Basic Program Structure
  - Assembling, Linking, Running (Machine Exercise)

Computer Programs

❒ **A computer program is a series of instructions that specifies the operations a computer should perform**

❒ **Some instructions may be used by all programs**

These instructions are called unprivileged, and these are the instructions discussed in this course

✗ We omit discussion of floating point and vector instructions

Most application programs use only these instructions

❒ **Some instructions may only be used by authorized programs**

These instructions are called privileged instructions, and they may only be issued by programs such as the Operating System

✗ There is also a category of instructions called semi-privileged, which we lump together with the privileged instructions

Application programs may request the Operating System to issue privileged instructions through special interfaces

✗ Most commonly, application programs request data transfer between memory and external devices through interfaces with names like READ and WRITE, GET and PUT, and so on

Types of Operations

❒ **Unprivileged instructions are very elementary and can be grouped into only a few types of operations. The most common types of operations are**

Arithmetic (add, subtract, multiply, divide)

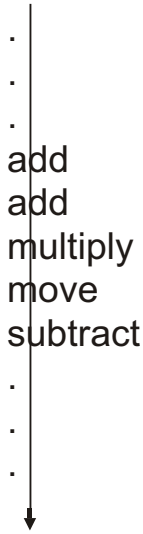
Transformation (move, manipulate, change)

Comparison (compare, test)

Order changing (change the sequence of instruction execution)

Instruction Execution

- **The instructions in a computer program are normally executed in a sequential manner, from first to last:**

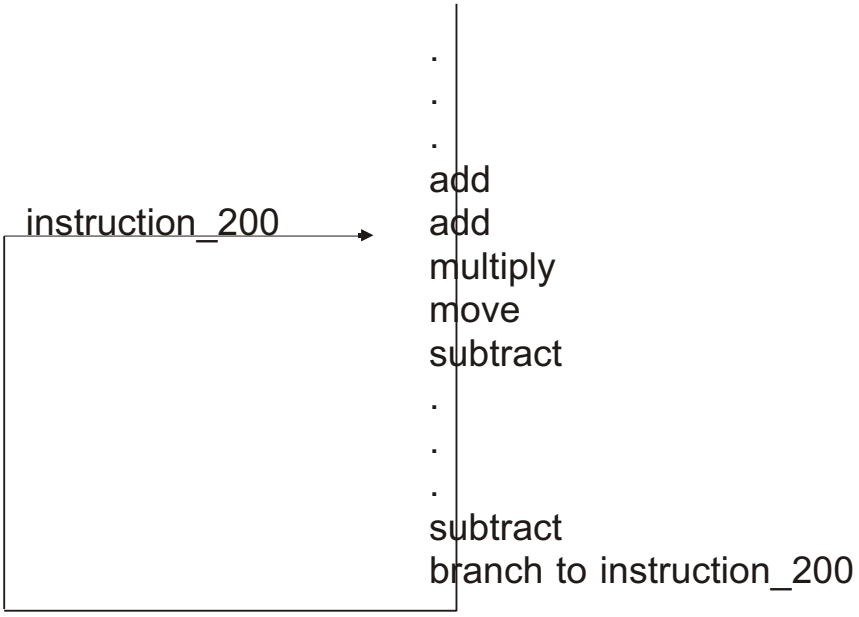


- **So the order in which you code (write) the instructions is the order in which they will execute**

Branching

- **An order changing instruction (usually called a "Branch" in Assembler and "Go To" or "Jump" in other languages) tells the computer to proceed to an instruction not in the normal sequence**

- **This enables the computer to repeat a set of instructions as often as necessary:**



- **This structure is called a <u>loop</u>**

Conditional Branching

- **Combining a compare or test instruction with an instruction that branches or not depending on the result of the compare or test, we can tell the computer to execute various sets of instructions under different situations:**

- **On a conditional branch, if the branch is not taken, execution continues normally, to the next sequential instruction (nsi)**

Modules

- **A computer program is written in a particular programming language - Assembler language in this course**

- **Code the program following the rules for the language, and then key the program into the system**

  This initial format is called a source module

  Source modules are not executable by the computer

- **Feed the source program into an IBM-supplied program called the Assembler**

  The Assembler reads source code and converts it to a machine-readable format called an object program or object module

  Object modules are not executable by the computer

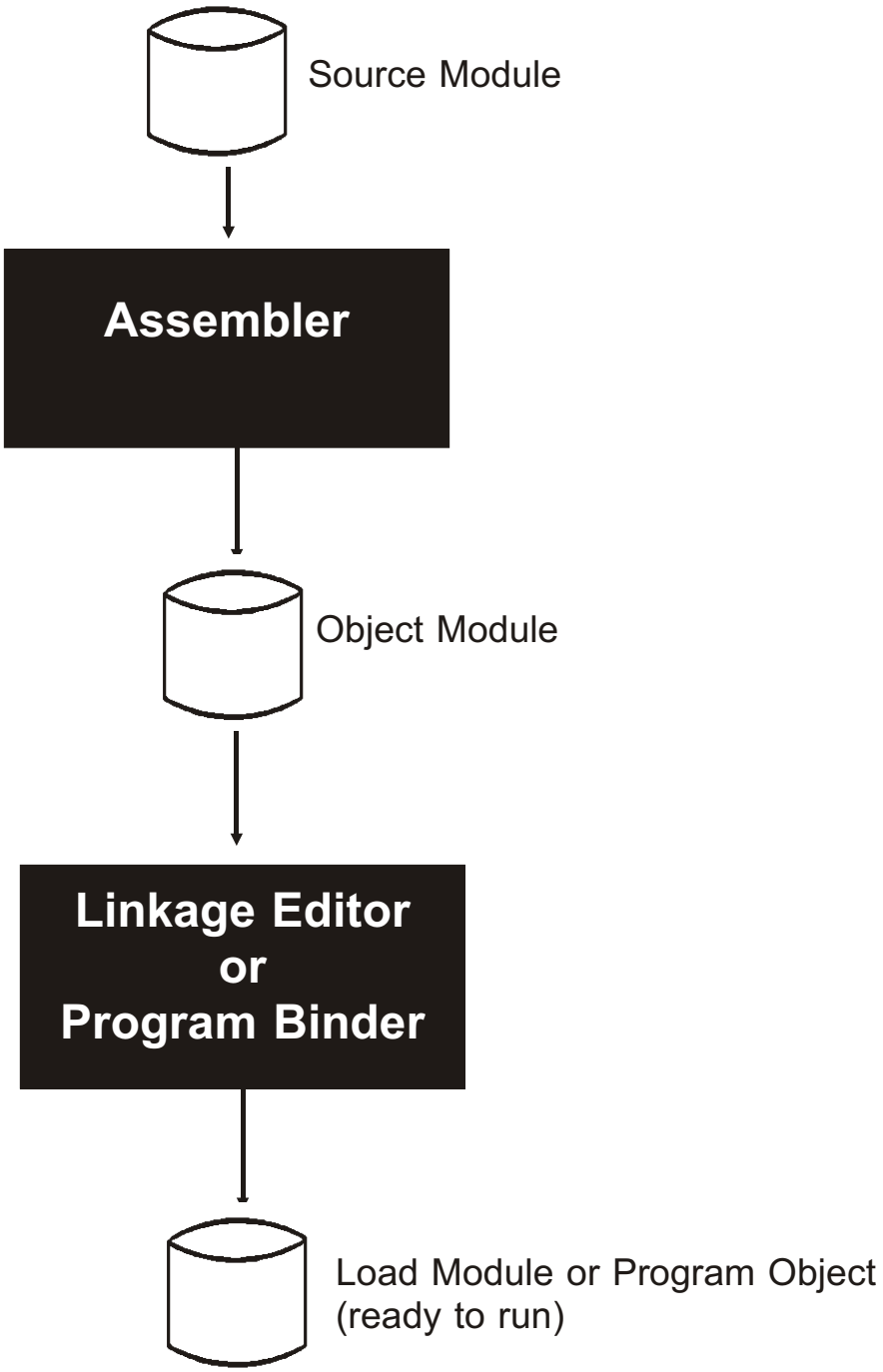
- **Feed one or more object modules into an IBM-supplied program called the Linkage Editor**

  The Linkage Editor produces an executable, machine readable format called a load module

  Load modules are stored in libraries, ready to run whenever they are invoked

  Most recently, the Linkage Editor has been suplemented by a program called the Program Binder, which serves the same role but can produce load modules or program objects (a specially formatted version of load modules)

Module Translations



❒ **In this class, we concentrate on writing source code; procedures to do the necessary Assembles, Link Edits, and runs will be provided to you**

Source Instruction Format

❒ **An imperative instruction in Assembler source format has three components to it:**

label

Optional; only needed if an instruction is to be referenced by a branch instruction; if present, must begin in column 1

operation

A word, abbreviation, or mnemonic that describes the actual operation the computer is to perform (for example: add, move); begins after at least one space after any label

operands

A description that identifies where the data to be operated on is located; for a branch instruction, this is the label of the instruction to be branched to

Most instructions require two operands; the result of arithmetic and transformation instructions generally replaces one of the operands (usually the first)

✗ For example,

```
ADD     FLDA,FLDB
```

would add the contents of FLDA and FLDB and place the sum into FLDA

To talk about operands, we need to talk about data representation, memory organization, and addressing ...

Computer Memory

Is a string of Bits (Binary digits: objects which can only have a value of 0 or 1)

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...

- **Organized into bytes of 8 bits each**

|xxxx xxxx|xxxx xxxx|xxxx xxxx| ...

Data and programs are represented in memory as strings of bits

Data Representation

- Different types of data are represented in different ways

- Primary data types:
  - Character string
  - Packed decimal
  - Binary integer
  - Floating point (Short, long, extended formats)
  - Instructions

Character String Data

- **Character string data is concerned with representing the characters used in a natural language internally in computer memory**
  - ✗ **Basically, the question is, how to represent the data from a keyboard in bit patterns**
- **This is done by what is called a "coding scheme": each character you want to represent is assigned a particular pattern of bits**
  - ✗ **S/370 family machines use a coding scheme called "EBCDIC" (Extended Binary Coded Decimal Interchange Code)**
- **Some sample EBCDIC character coding assignments:**

Character	Assigned to bit pattern	
'+'	01001110	
'a'	10000001	
' '	01000000	(Space, or Blank)
'B'	11000010	
'4'	11110100	

Character String Data, 2

❒ **Note that not every possible bit pattern in 8 bits is assigned to a printable character**

For example:

00000000
00111101
10000000
11011010

❒ **We talk about character <u>string</u> data because the hardware does not have any predetermined length or maximum size for this kind of data:**

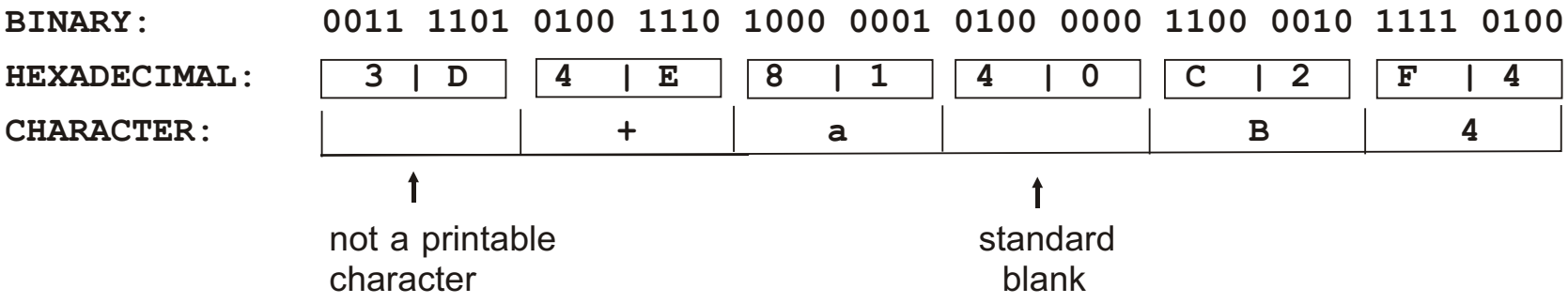
✗ **You can string characters together as long as you like**

HEXADECIMAL

- **Because binary is difficult to read and write**
- **And because not all bit patterns represent printable characters**
- **We usually use Hexadecimal to represent the contents of memory**

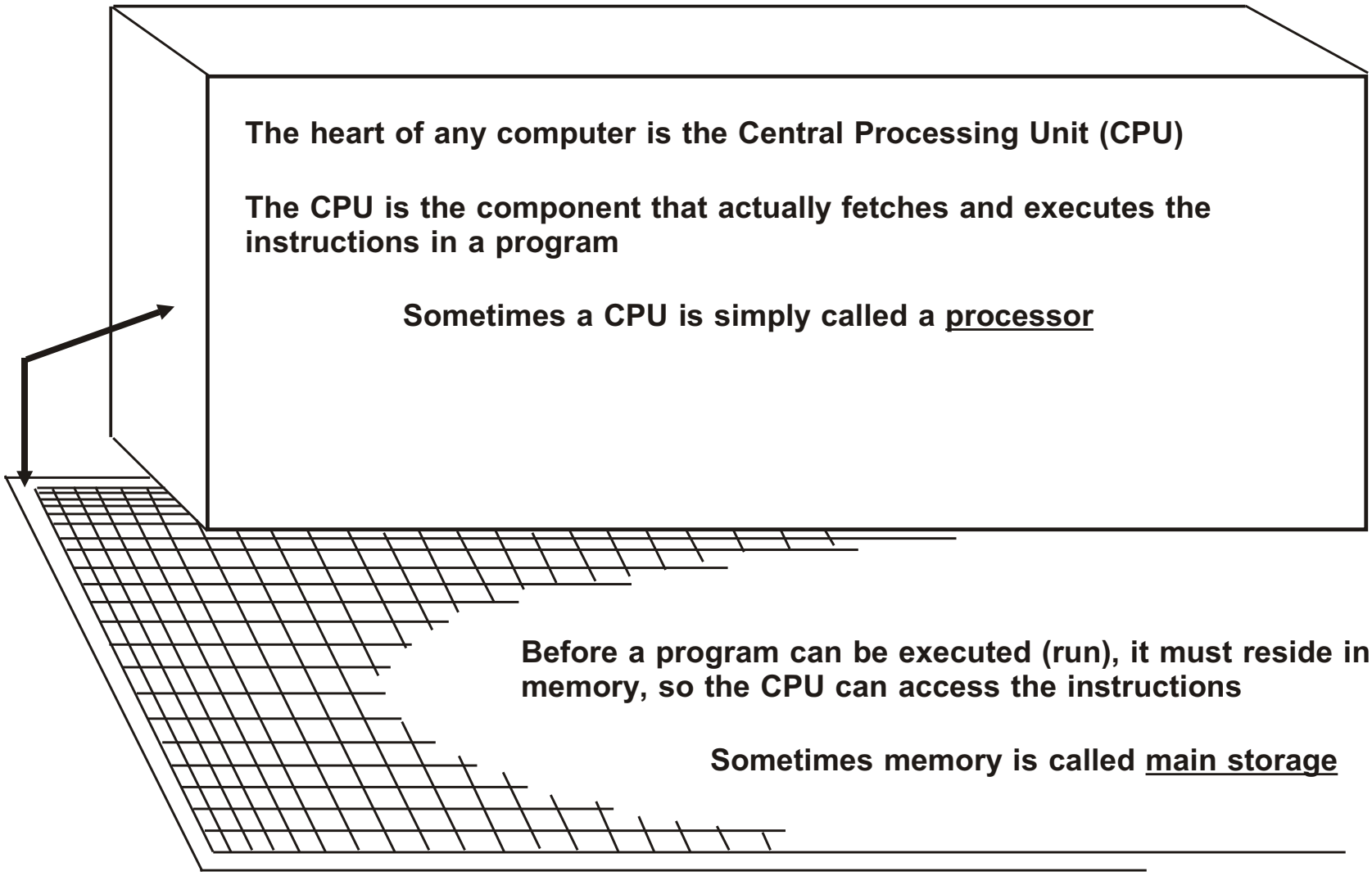
  A short-hand: one hex digit for each four bits (half-byte or nybble)

HEX	BIN	DEC
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15



HEXADECIMAL is number base 16 (HEX + DECIMAL = 6 + 10)

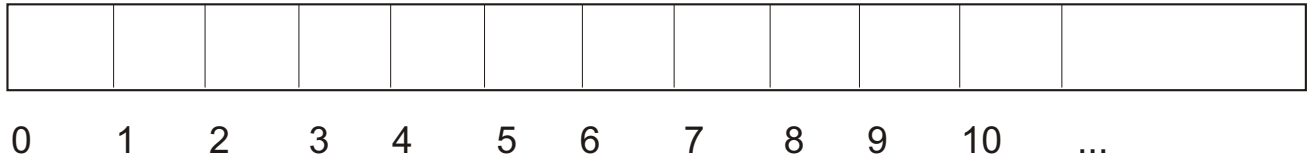
A CPU and Memory (Main Storage)



Since unprivileged instructions may only access data in memory, these instructions must specify, as their operands, locations in memory: the instructions point to the data

Memory Addressing

- **Each byte of memory is numbered:**



- **The number which uniquely locates each byte of memory is called its** address

To reference the data at a memory location in an instruction, you specify (in the instruction) the byte number, or address, of that memory location

The CPU will then fetch the data at that location for processing, or use that location as the target location for storing the result of an instruction

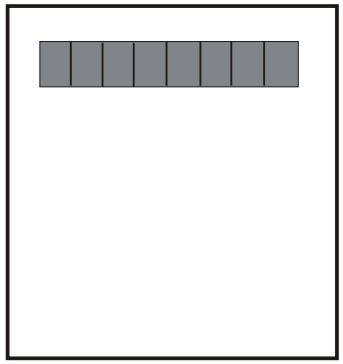
Address Registers

- **Instructions and data, then, are located in memory by their addresses**

- **The CPU contains several address registers it uses to hold memory addresses (point to locations in memory)**

  A register is a small scratch pad of memory in the CPU itself, often used for holding addresses or data, and for doing calculations

  - ✗ Think of the display on a calculator:



  - ✗ Address registers are 32 bits long, but addresses are either 24 bits long or 31 bits long, depending on the addressing mode currently being used by the CPU

24-Bit Memory Addresses

❒ **Here are some sample addresses when the CPU is using 24-bit addressing mode**

Decimal	Hex	Binary
0	000000	0000 0000 0000 0000 0000 0000
1	000001	0000 0000 0000 0000 0000 0001
2	000002	0000 0000 0000 0000 0000 0010
3	000003	0000 0000 0000 0000 0000 0011
512	000200	0000 0000 0000 0010 0000 0000
1024	000400	0000 0000 0000 0100 0000 0000
2048	000800	0000 0000 0000 1000 0000 0000
4096	001000	0000 0000 0001 0000 0000 0000
8192	002000	0000 0000 0010 0000 0000 0000
1048576	100000	0001 0000 0000 0000 0000 0000
2097152	200000	0010 0000 0000 0000 0000 0000
16777213	FFFFFD	1111 1111 1111 1111 1111 1101
16777214	FFFFFE	1111 1111 1111 1111 1111 1110
16777215	FFFFFF	1111 1111 1111 1111 1111 1111

❒ **Each address can fit in three bytes; in 24-bit addressing mode, the leftmost byte in an address register is ignored**

31-Bit Memory Addresses

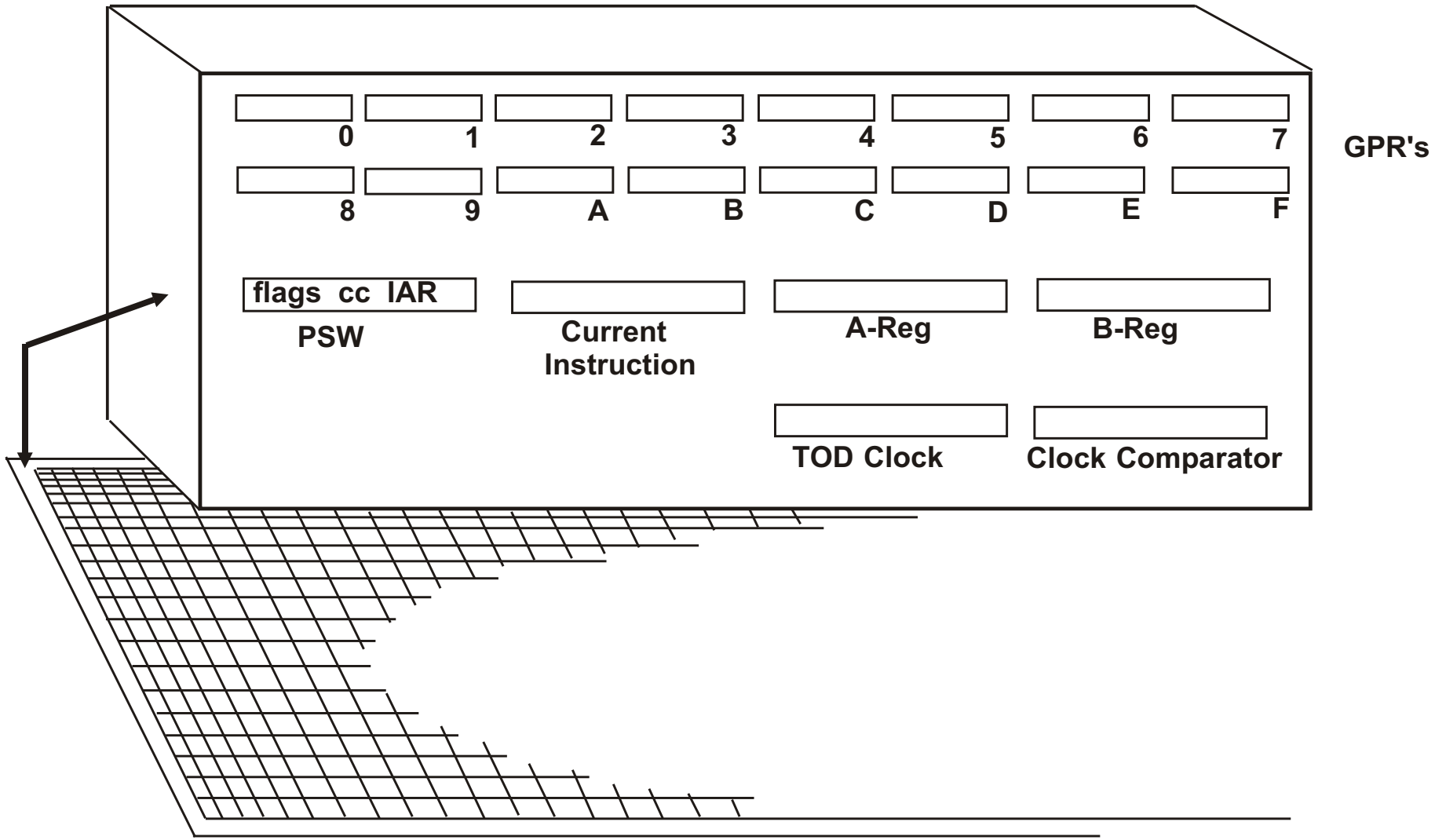
❒ **Here are some sample addresses when the CPU is using 31-bit addressing mode**

Decimal	Hex	Binary
0	00000000	x000 0000 0000 0000 0000 0000 0000 0000
1	00000001	x000 0000 0000 0000 0000 0000 0000 0001
2	00000002	x000 0000 0000 0000 0000 0000 0000 0010
3	00000003	x000 0000 0000 0000 0000 0000 0000 0011
4096	00001000	x000 0000 0000 0000 0001 0000 0000 0000
8192	00002000	x000 0000 0000 0000 0010 0000 0000 0000
1048576	00100000	x000 0000 0001 0000 0000 0000 0000 0000
2097152	00200000	x000 0000 0010 0000 0000 0000 0000 0000
16777215	00FFFFFF	x000 0000 1111 1111 1111 1111 1111 1111
67108863	03FFFFFF	x000 0011 1111 1111 1111 1111 1111 1111
1073741823	3FFFFFFF	x011 1111 1111 1111 1111 1111 1111 1111
2147483647	7FFFFFFF	x111 1111 1111 1111 1111 1111 1111 1111

❒ **In 31-bit addressing mode, the leftmost bit in an address register is ignored**

The leading "x" indicates the bit is ignored for address calculations

CPU, Registers, and Main Storage



Central Processing Unit (CPU) - pre-z/Architecture

16 General Purpose Registers (GPRs)
+ Four bytes each, numbered 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
+ Used to hold addresses, integers, any data
+ Decimal range 0 to 4,294,967,295 as unsigned binary integer
+ Decimal range -2,417,483,648 to +2,417,483,647 as signed integer

16 Floating Point Registers (not shown)
+ Eight bytes each, work with data in "Hexadecimal floating point" (S/360) format (also HFP) or in "binary floating point" (IEEE) format (also 'BFP')

16 Control Registers (not shown)
+ Four bytes each; each register serves some pre-defined purpose

16 Access Registers (not shown)
+ Four bytes each; used to access multiple address / data spaces

Program Status Word

(PSW)

Flags: Status Information
+ Supervisor / Problem state
+ Wait / Executing state
+ ... other control states ...

Flags: Storage Protect Key
+ Controls fetch / store access to parts of memory

Condition Code
+ 2 bits, set when some instructions are executed, to indiate result of operation

Addressing Mode Indicator
+ 1 bit, indicate 24-bit mode ('0') or 31-bit mode ('1')

Instruction Address Register (IAR)
+ Memory address of next instruction to be executed

Central Processing Unit (CPU), continued

Other Components

+ Instruction fetch / decode logic and registers
  Get the current instruction, point to the next instruction, determine instruction type of current instruction

+ Address calculation logic and registers (A-Reg and B-Reg)
  Compute the address of operands in memory, if needed

+ Instruction execution logic
  Perform the instruction

+ Interruption handling circutry
  Save status when an interrupt occurs, branch to interrupt handling routine

+ Clock and timers (TOD clock, Clock Comparator, CPU Timer)
  For determining current date / time, allowing time intervals to elapse, etc.

+ Vector facility
  Includes registers and instruction to perform operations on arrays of data simultaneously

+ Cryptographic facility
  Provides encoding and decoding services

Notes

The S/390 had only four floating point registers, designated 0, 2, 4, and 6

✗ Current machines have 16 floating point registers; and the registers can work with data in traditional IBM floating point format (also called "hexadecimal floating point", HFP) or IEEE floating point format (also called "binary floating point", BFP) or decimal floating point (DFP)

Central Processing Unit (CPU), continued

- **Over time, what was originally the IBM S/360 has gone through many enhancements and changes to get to the point it is now at**
  - ✗ For example, IBM mainframes did not always have Vector facilities or Cryptographic facilities

  Although the core design has remained, new capabilities and new instructions have been added

- **In addition, the Assembler itself has gone through extensive improvements**

  Not just to support the new instructions (although, of course that is included)

  But there's also been enhancements to support new features such as labled USINGs and lots more

Computer Exercise: Setting Up For Hands On Lab Exercises

At this point, we'll take a little break from lecture to prepare for our later labs.

Using ISPF option 6, enter the following command

===> exec '_____.train.library(c510strt)' exec

and press <Enter>

This will cause the setup process to run. You will be prompted for a high level qualifier for your data sets. Unless the instructor tells you otherwise, use your TSO userid (the process is set up to use this as a default anyway). Press <Enter>.

The setup process will create four libraries for you:

<hlq>.TR.CNTL - for JCL

<hlq>.TR.SOURCE - for your Assembler code

<hlq>.TR.LOAD - for load modules

<hlq>.TR.PDSE - for program objects
Note: some shops will omit the PDSE data set

where "<hlq>" is replaced by the high level qualifier you entered in response to the setup's prompt.

The process also places some members you'll need for various machine exercises in some of these libraries

Machine Instruction Formats

- Here we do some more review to set the stage for what's coming

- The CPU expects to find instructions represented in a binary form: the CPU does not recognize the word "ADD", for example, but it recognizes a binary operation code that means "ADD" to it

- For those instructions where both operands are in memory, the machine format expected by IBM mainframe computers is like this:

Operation Code	Data Length	1st Operand Address	2nd Operand Address

- Note that for some storage to storage instructions, the data length byte actually has the first hex digit represent the length of the first operand and the second hex digit represent the length of the second operand

  In the other storage to storage instructions the data length byte is a single number and the length applies to both operands

Operand Addresses

- **Machine instructions do not contain operand addresses as 24-bit or 31-bit memory addresses**

  The reason for this is that a program may be loaded into any consecutive range of memory addresses when it is to be run (executed)

  If operand addresses were stored as absolute memory locations, you would need to re-Assemble a program every time it was to execute from a different place in memory

- **Instead, every program is expected to establish a Base Address (starting address) in memory, and the location of operands is specified by how many bytes away the operand is from this base address**

  This distance is called the Displacement

- **Every time a program is loaded into memory, it finds out the address it is loaded at and uses that value for the Base Address, and operands are located relative to this starting point**

  A machine instruction is used to place a memory address into a General Purpose Register (GPR)

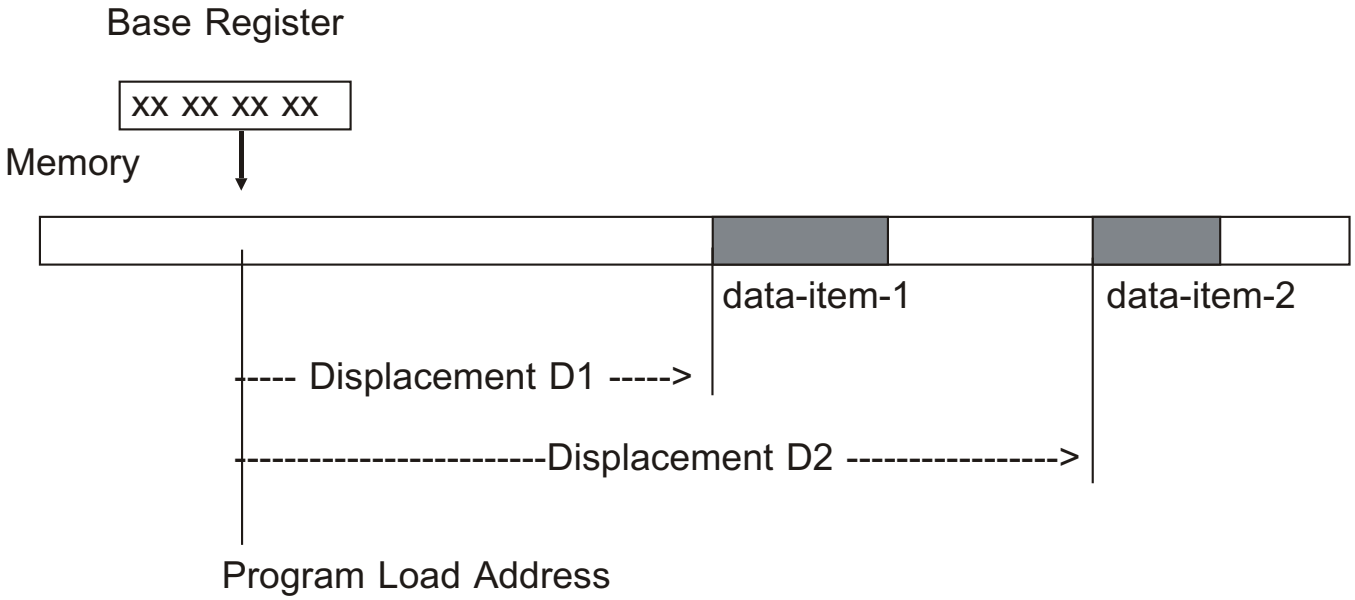
  This is how a program establishes its Base Address

Base / Displacement

- **This way, no matter what address in memory a program is loaded at, once you get that load address into a General Purpose Register, you locate data items (or instructions) by specifying**

  you are using that particular GPR as a Base Register

  and how many bytes of displacement should be used to calculate the correct address for the data:



Operand Address= Contents of Base Register + Displacement

= C(Base Reg) + Displacement

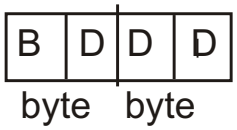
Operand Addresses, 2

❒ **So, in a machine instruction that references a memory address, the address is actually stored in Base / Displacement form, in two bytes:**

The first half-byte identifies which General Purpose Register is being used as a Base Register (a hexadecimal digit 0-9 or A-F)

✗ NOTE: if register 0 is used, a **value** of zero is used for the base, **not** the **contents** of register 0

The last three hex digits (one and a half bytes) specifies the displacement to be used in locating the data item



❒ **The range of values for the displacement is:**

	000 - FFF	**in hexadecimal**
or	**0000 - 4095**	**in decimal**

In other words, a single base register can support a program up to 4096 bytes (4K) long

Larger programs require using two (or more) different GPRs for base registers (discussed later), or breaking the program up into subroutines

Instruction Formats

- **There are several instruction formats from the original design (in all cases, the opcode is the first byte of the instruction):**

  Register-Register (RR): both operands are register numbers; the insruction is 2 bytes in size; sometimes the 1st op. is a bit mask

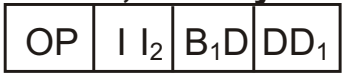
  Register-Index (RX): 1st op. is a register, 2nd op. is a base, displacement and index register; the value in the index register is added to the second operand address to get the starting address; instruction is 4 bytes long; sometimes the 1st op. is a bit mask

  Storage-Storage (SS): first and second operands are both of the base+displacement form (so two bytes each); the second byte of the instruction is either a single length code that applies to both operands (so length from 0 to 255), or a half byte length for each of the operands (so length from 0 to 15 for each); 6 bytes long

  Storage-Immediate (SI); the first operand is one byte of data that is actually the second operand; the first operand is a base and displacement; four byte instruction

- **We study this because it will help later when we look more deeply at z/Architecture**

Instruction Formats, 2

- **All instructions are two bytes, four bytes, or six bytes long; the opcode tells the story in its first two bits:**

first two bits	instruction length
00	2 bytes
01	4 bytes
10	4 bytes
11	6 bytes

- **The value x'00' will never be assigned as an operation code**

- **Additional instruction formats have been introduced at many points along the way, and we'll discuss them as the opportunity arises**

Machine Language

- **The CPU only understands instructions expressed in binary (or, of course, hex) in the form we've been looking at:**

  operation code / data length / base-displacement memory addresses

  This is called "machine language"

- **In some instructions, one or more operands may be in registers, or even included in the instruction itself**

- **Machine instructions in the S/360 and its successors are either two bytes, four bytes, or six bytes long, depending on the instruction and where the operands are located**

  Instructions with operands in registers, for example, do not need to contain memory addresses or data lengths

- **Fortunately, we do not have to code in machine language to write Assembler Language programs (although it is sometimes useful to know how to interpret machine instructions in a memory dump)**

Assembler Language

❒ **Assembler Language is a computer programming language designed to allow the programmer to specify a series of machine instructions**

❒ **This language has its own vocabulary, grammar, and rules of syntax**

❒ **Assembler Language simplifies coding machine instructions by:**

Allowing the use of mnemonics to specify an instruction, instead of the hexadecimal or binary machine language representation

✗ for example: write "MVC" for "move characters" instead of a hexadecimal "D2"

Allowing the use of symbols (names, labels) instead of forcing us to keep track of base and displacements

✗ the Assembler will determine the correct base and displacement values, based upon information we supply

The Assembler

❒ **The Assembler is a program already in executable form that converts our Assembler Language source programs into actual machine code**

This course is based on the most recent version of the High Level Assembler (HLASM)

✗ Pointing out differences from earlier versions of Assemblers where relevant

❒ **The Assembler works with three kinds of statements:**

Machine Instructions: mnemonic representations of the machine instructions we want our program to contain

✗ Machine Instructions are converted one for one into actual machine instructions in binary format

Assembler Instructions: that tell the Assembler to do something (allow room for data, start a new page on the listing, use a particular register for a base, and so on)

✗ Sometimes Assembler Instructions result in object code being generated, but often these instructions simply give the Assembler information

Macro Instructions: IBM- (or user-) defined instructions; the definitions, in turn, are composed of Machine, Assembler, and other Macro Instructions

Program Development

❒ **We use the following process when writing Assembler Language programs:**

Code the program in Assembler Language, using a text editor such as the ISPF/PDF editor

Submit a job that invokes the Assembler to convert our source code into object code and then invokes the Linkage Editor to convert our object code into executable format (a load module)

✗ This job can also test the resulting program (our approach in this class), or you can set up a separate job to test the program

✗ In some environments the invocation of the Linkage Editor is replaced by using the Binder; the output can be a load module or the newer program object

➢ This course was developed with preference to the Binder, but all the labs will work if you use the Linkage Editor

❒ **The following content is a review of classic Assembler program structure mixed with current Assembler style**

Assembler Rules and Conventions - V1R6 and later

Names (instruction labels and data labels)

1-63 characters from A-Z, 0-9, $, #, @, _ (underscore)

- ✗ Lower case alpha (a-z) are supported as equivalent to upper case alpha
- ✗ First character must not be numeric
- ✗ Names must be unique within a program
- ✗ Earlier assemblers only supported upper case names with a maximum of eight characters, and no underscores, $, #, or @

Coding Rules (columns 1 -71)

Name, if present, begins in column 1

- ✗ Followed by one or more blanks

Operation Code (Machine, Assembler, or macro instruction)

- ✗ Followed by one or more blanks

Zero or more operands

- ✗ If multiple operands, separated by commas, no extra spaces
- ✗ Followed by one or more blanks

Remarks / comments (optional)

To continue a line, enter a non-blank character in column 72

- ✗ The continuation line must have spaces in columns 1 - 15 then the continuation text begins in column 16

Comment lines are coded with an asterisk (*) in column 1

Blank lines are allowed anywhere (not so in older Assemblers)

Control Sections

- **Programs are organized into "chunks" of code (instructions and / or data areas called Control Sections, or CSECTs)**

- **The beginning of a CSECT is indicated by the appearance of either a START or CSECT Assembler instruction:**

```
  csectname  START  value
```

or

```
  csectname  CSECT
```

Assembler Instructions

Notes

The "csectname" must follow the rules for names in Assembler, with the restriction that it may only be 8 characters long, maximum

There may only be one START statement in a source program; there may be any number of CSECT statements (although in this course we will normally have only one CSECT per program)

"value" specifies a starting value for the Assembler's location counter (default: 0) in decimal or hex

Each time a new CSECT statement is encountered, the Assembler sets that control sections's location counter to 0 (zero)

The Location Counter

- As the assembler processes your source code, generating machine format instructions, it maintains an internal counter called the "location counter" that contains the number of bytes of storage assembled in the current CSECT so far

- The location counter starts out at zero for each CSECT, and as each instruction is assembled, the location counter is incremented by the size of the resulting machine instruction

- The location counter is used only during the Assembly process

- You can reference the location counter in an instruction operand by coding an asterisk (*)

  The value is the address of the first byte of the instruction containing the reference

  More on this later

Location Counter Example

Location Counter	Source Instruction			(storage size, in bytes)
000000	MYPROG	CSECT		---
000000		STM	14,12,12(13)	(4)
000004		LR	12,15	(2)
000006		USING	MYPROG,12	---
000006		ST	13,SAVE+4	(4)
00000A		LA	13,SAVE	(4)
00000E	...			

"---" indicates an instruction does not generate any space in the object module

- **So we see that the CSECT instruction just indicates the beginning of the control section and then,**

  the STM instruction is at location 0 in the object module

  the LR instruction is at location 4 in the object module

  the USING instruction is an Assembler instruction that does not take up any space in the object module

  the ST instruction begins at location 6

  the LA instruction begins at location 10 (decimal; 'A' in hexadecimal)

  and so on ...

END

❒ **A control section begins with a START or CSECT statement and continues until ...**

A new CSECT is begun

Or a DSECT is encountered

Or an END statement is encountered:

```
END    [starting-location]
```

Assembler Instruction

Notes

The END statement must be the last statement in your program: it denotes the end of the source module and any statements following it are discarded

"starting-location" represents where in the program execution should begin when the program is actually run (the Entry Point)

✗ The brackets ([ ]) around "starting-location" are typical IBM syntax style, indicating an operand is optional (you never key in the brackets)

✗ The default "starting-location" is the first byte of the program

Program Structure, I

❒ **So, the basic structure of Assembler programs is as follows:**

```
MYPROG  CSECT
          |
.         |
          |
.  the actual source code
          |
.         |
          |
         END   MYPROG
```

In this example, "MYPROG" is the CSECT name (this is chosen by the programmer, or by a standards convention in some installations)

Usually, this is also the name of the source library member that contains the program as well as the name of the load library member that contains the resulting load module

✗ Although these conventions do not have to be followed, and in some cases they can't be followed, we shall follow them in this class

✗ Thus, in the example above, you would edit member MYPROG in your source library, typing in the lines above

Saving Registers

❒ **In OS/390 and z/OS, every Assembler program must first save the current General Purpose Register values into a save area provided by the operating system**

This is because almost every program needs to use the GPRs, so a convention has been established for saving and restoring register values:

- ✗ Every program will provide a save area for the registers, and the address of this save area is placed into register 13
- ✗ When a program invokes another program, the invoked program will save the register values as received from the invoking program in the invoking program's save area
  - ➢ The invoked program will then provide its own save area so that if it invokes another program that program will have a place to save the calling program's register values
  - ➢ This save area will be pointed at by register 13
  - ➢ When a program returns to the program that invoked it, it must first restore the registers as they existed on entry, so the invoking program is guaranteed its register values are the same as when it invoked the lower level program

We discuss this in detail later, for now accept that our program must first issue the machine instruction:

```
STM   14,12,12(13)
```

Addressability

❒ **The next thing to do in an assembler program is to establish "addressability"; this requires two instructions:**

A machine instruction that will load a memory address into a general purpose register; this establishes the base address

An assembler instruction that will inform the Assembler that this is the register that should be used as the base register for the program, and from what point displacements should be calculated

✗ Note that it is your responsibility, as the programmer, to make sure the value in your base register is not "clobbered" by any code you write later in the program

❒ **Almost any GPR may be used, but a common convention (and one we will follow in this class) is to use GPR 12 for the first base register for a program**

So we have this kind of common convention:

R15: load address of module on entry
R14: return address to invoking program
R13: address of register save area
R12: first base register
...
R1: address of passed parameters
R0: varies

Addressability, Continued

- **Several machine instructions exist for getting a memory address into a register, but for now we shall use this one:**

```
LR  12,15
```

This will place into register 12 the contents of register 15

This takes advantage of the fact that in OS/390 and z/OS when a program is invoked, the address the program has been loaded at is placed into register 15 before passing control to the program

- **To tell the Assembler how to calculate displacements, code:**

```
USING MYPROG,12
```

This Assembler instruction says: use register 12 as the base register, and calculate displacements from the beginning of the CSECT named MYPROG

Providing a Save Area

- Following the convention mentioned earlier, the next thing an Assembler program must do is to save the pointer to the save area provided by the invoking program (the operating system in this case):

```
ST     13,SAVE+4
```

- Then, the address of this program's save area must be placed into register 13:

```
LA     13,SAVE
```

- Now the Assembler program has completed following standard "linkage conventions" for the OS/390 and z/OS environments

- But the program needs to contain a definition for the label "SAVE" referenced in the two instructions above

  This is done by coding:

```
SAVE     DC     18F'0'
```

  This Assembler instruction reserves memory for the save area

  18 fullwords, or 72 bytes

  The instruction may be placed almost anywhere in the program, but it is generally placed near the end of the program

Program Structure, II

- So, the basic structure of an Assembler program in the OS/390 and z/OS environments, as far as we know now, looks like this:

```
MYPROG   CSECT
         STM   14,12,12(13)
         LR    12,15
         USING MYPROG,12
         ST    13,SAVE+4
         LA    13,SAVE
.    code the actual work beginning here
          |
.         |
          |
.         |
SAVE     DC    18F'0'
         END   MYPROG
```

- The only thing remaining for a general structure is: how to terminate an Assembler program

Terminating An Assembler Program

❒ **When a program has run to completion, it must perform three final tasks:**

Restore the registers to their state before the program was run by issuing these machine instructions

```
L       13,SAVE+4
LM      14,12,12(13)
```

Place a return value in register 15 (the program that invoked this program can thus get some feedback about how things went); typically, a return code of zero is passed back by:

```
SR      15,15
```

And then return to the invoking program

```
BR      14
```

✗ Another OS/390 and z/OS convention: when a program is invoked, register 14 contains the address to return to

✗ This machine instruction branches to the address in register 14

❒ **This completes the basic structure or "skeleton" of an Assembler program designed to run in the OS/390 or z/OS environments**

❒ **The result of this structure is shown on the following page ...**

Program Structure, III

- So this is the basic structure of an Assembler program in the OS/390 and z/OS environments:

```
MYPROG   CSECT
         STM   14,12,12(13)
         LR    12,15
         USING MYPROG,12
         ST    13,SAVE+4
         LA    13,SAVE
         .
         .
         .
         L     13,SAVE+4
         LM    14,12,12(13)
         SR    15,15
         BR    14
SAVE     DC    18F'0'
         END   MYPROG
```

- We will discuss all these conventions and instructions in greater detail during the course

  This provides you with the minimum amount of information to code the initialization and termination routines in an Assembler program

- Another thought along these lines: it would be a good idea to comment the code, in order to simplify maintenance later ...

Program Structure, IV

❒ **Commenting the code may be done in a variety of styles**

This is pretty basic

✗ You may prefer your own style

✗ Or your installation may have specific standards on comments

```
MYPROG   CSECT
         STM   14,12,12(13)   SAVE REGISTERS
         LR    12,15          ESTABLISH
         USING MYPROG,12         ADDRESSABILITY
*  SAVE POINTER TO CALLING PROGRAMS REGISTERS
         ST    13,SAVE+4
*  POINT TO OWN SAVE AREA
         LA    13,SAVE
*******************************************
          .
          .
          .
*******************************************
*  PICK UP ADDRESS OF CALLING PROGRAMS SAVE AREA
         L     13,SAVE+4
         LM    14,12,12(13)   RESTORE REGISTERS
         SR    15,15          RETURN CODE = 0
         BR    14             RETURN TO SYSTEM
*******************************************
*
*          CONSTANTS AND DATA AREAS
*
*******************************************
SAVE     DC    18F'0'
         END   MYPROG
```

Program Structure, V

- **Finally, a word about capitalization**

  Labels, instructions, and operands may be coded in mixed case

  Remarks and comment lines may contain any EBCDIC character

  So comments might show up better if coded in upper and lower case:

```
MYPROG   CSECT
         STM   14,12,12(13)   Save registers
         LR    12,15          Establish
         USING MYPROG,12         addressability
*  Save pointer to calling programs registers
         ST    13,SAVE+4
*  Point to own save area
         LA    13,SAVE
*****************************************
         .
         .
         .
*****************************************
*  Pick up address of calling programs save area
         L     13,SAVE+4
         LM    14,12,12(13)   Restore registers
         SR    15,15          Return code = 0
         BR    14             Return to system
*****************************************
*
*         Constants and data areas
*
*****************************************
SAVE     DC    18F'0'
         END   MYPROG
```

  And, you may recall, z/OS provides two macros to help with this: SAVE and RETURN ...

The SAVE Macro

Samples

```
SAVE   (14,12)

SAVE   (14,12),,'Entry to first routine'

SAVE   (14,12),,*
```

Working

Generates the STM instruction of standard linkage conventions

If third operand is specified, the macro generates a DC with the constant and a branch around the constant

- ✗ An asterisk (*) implies the constant to use is the name on the SAVE macro; if no name on the SAVE macro use the name of the current CSECT

The second operand is intended for non-standard register saving

- ✗ In particular, if you don't specify (14,12) in the first operand, coding a 'T' in the second operand ensures registers 14 and 15 are saved in the appropriate place in the save area; for example:

  SAVE (3,7),T

- ✗ Not used much anymore, but you may see old code that uses this

The RETURN Macro

Samples

```
RETURN (14,12)
RETURN (14,12),,RC=n
RETURN (14,12),,RC=OK
RETURN (14,12),,RC=(15)
```

Working

Generates the LM and BR instructions

✗ But not the "L 13,4(13)"

If RC= operand specified, the macro generates the code to place return code in R15

✗ 'n' is an integer between 0 and 4095

✗ 'OK' is an example of using a symbol; 'OK' must be defined something like this:

```
OK          EQU    12
```

✗ If you code RC=(15), that says the return code is already in R15 and the RETURN macro generated code should not disturb it

➢ Only Register 15 may be used in this way

Same remarks about the second operand as for SAVE

Standard Linkages Using SAVE and RETURN

- Applying these new macros yields:

```
MYPROG   CSECT
         SAVE  (14,12)         Save registers
         LR    12,15           Establish
         USING MYPROG,12          addressability
*  Save pointer to calling programs registers
         ST    13,SAVE+4      Store backward ptr
*  Point to own save area
         LA    13,SAVE        Establish own SA
*********************************************
         .
         .
         .
*********************************************
*  Pick up address of calling programs save area
*    and return to z/OS with a zero return code
         L      13,4(13)
         RETURN (14,12),,RC=0
*********************************************
*
*         Constants and data areas
*
*********************************************
SAVE     DC    18F'0'
         END   MYPROG
```

- Many installations have their own home-grown linkage macros, usually named something like INIT, EXIT, ENTER, LEAVE, and so on

  Typically they also have options for establishing multiple base registers and other useful functions

  Find out what your installation uses if you're not sure

Computer Exercise: Assembling, Linking, Running

Now we have an opportunity to test our lab setup and prepare for later exercises.

1. In your TR.SOURCE PDS is a member named ASMREPT

   To give you some perspective, the program

   * reads each record in an inventory file
   * creates a print line and prints each record
   * extracts information from the record and updates a table
   * after the file has been read:
     sort the table by category name and print the table in that order
     sort the table by number of entries for each category
     and print the table in that order

   There is a listing of the program source in the Appendix to this book

2. To Assemble, bind, and run this program, use the procedure called C510JOB1 in your TR.CNTL library. Submit this job and examine the output from the Assembler, the binder, and the program itself.

The purpose of this exercise is to test the results of our setup and also to get a working program to modify later.