z/OS Assembler Programming Part 1: Beginnings

z/OS Assembler Programming Part 1: Beginnings - Course Objectives

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

1. Code a program in Assembler language that uses the following techniques:
   a. Use standard save area linkage techniques
   b. Define and process sequential files with fixed length records, including
      * Reading and writing records from / to DASD files
      * Reading and writing records from / to tape files
      * Writing records to print files, including formatting detail lines, but not using carriage control characters or other report management techniques
   c. Perform calculations using packed decimal arithmetic, including formatting results with edit patterns and half-adjusting results
   d. Perform calculations using binary integer arithmetic
   e. Work with data in tables, including defining and accessing the elements in a table
   f. Use DSECTs to describe structures
   g. Use multiple base registers

2. Document the program listing with comments to assist in maintenance and understanding of the code

3. Debug the resulting code of program-check type errors

Note: This course supports OS/390 and z/OS operating system environments.

z/OS Assembler Programming Part 1: Beginnings - Topical Outline

Day One

Day Two

Day Two, continued

Packed Decimal Arithmetic
- Zoned Decimal Format
- Packed Decimal Format
- DC and DS for Zoned and Packed Type Data
- Packed Decimal Instruction Set: PACK, UNPK, ZAP, CP, AP, SP, MP, DP
- Arithmetic Concerns: Significant digits and Keeping Track of Decimal Points
- Computer Exercise: Packed Decimal Calculations 145

More Assembler and Arithmetic Concepts
- Redefining Storage
- Creating Data Structures
- The Assembly Listing Components
- Assembly Listing Control
- Introduction to Debugging
- Rounding
- MVO - Move With Offset
- SRP - Shift and Round Packed
- SRP vs MVO
- Computer Exercise: Half-Adjusting Data 177

Day Three

Editing Packed Decimal Fields
- DS / DC for Hexadecimal Data
- "ED" Instruction
- Edit Patterns
- Computer Exercise: Edit Packed Decimal Data 190

A Deeper Look at Instruction Formats
- DC / DS for Binary Data Type
- Addresses in Instructions
- Tables
- LA Instruction
- Instruction Formats (SS, RR, RX)
- MVI, CLI Instructions
- Instruction Formats (SI)
- MVN, MVZ Instructions
- Computer Exercise: Using Immediate Instructions 214

Day Three, continued

Binary Integer Data
- Binary Integer Data Formats
- Two's Complement
- DC/DS for Fullword, Halfword, and Doubleword Binary Data
- Boundary Alignment
- CVB, CVD Instructions
- L, LR, ST Instructions
- A, AR, S, SR, C, CR, MR, M, DR, D Instructions
- Concerns Of Working With Binary Integers
- Working With Binary Numbers - An Example
- Computer Exercise: Binary Arithmetic Computations 250

Day Four

More Binary Instructions
- Compare Instrucions
- LPR, LNR, LCR Instructions
- "Logical Arithmetic": AL, ALR, CL, CLR, SL, SLR
- Halfword Instructions: AH, CH, LH, MH, SH, STH

EDMK
- EDMK Instruction
- Computer Exercise: Floating Dollar Sign 260

Loops and Tables
- Literals
- LTORG
- Address Constants
- EQU - Equate Symbol
- Loop Control
- Tables
- BCT, BCTR, BXLE, BXH, IC, STC
- Computer Exercise: Table Processing 282

Day Five

Multiple base registers, DSECTS, ORG
- STM, LM
- Multiple Base Registers
- CNOP
- Dummy Sections - DSECTs
- ORG
- Computer Exercise: Using DSECTs.....300

Working With Bits
- O, OC, OR, OI, N, NC, NR, NI, X, XC, XR, XI Instructions
- Sorting Tables
- LTR, TM Instructions
- More on EQU
- Computer Exercise: Sorting a Table 318

Shift Instructions
- SRL, SRA, SLL, SLA, SRDL, SRDA, SLDL, SLDA Instructions

Translate
- Instruction Set: TR
- Code Fragments: Display Hex String and Direct Access to a Table
- Computer Exercise: Build a Table Dynamically 337

TRT and EX
- TRT - Translate and Test
- EX - Execute
- TRT and EX

Strings
- ICM, CLM, STCM Instructions
- MVCL, CLCL MVCIN Instructions

Setting Addressing Mode
- Addressing Mode
- AMODE and RMODE
- BASSM - Branch And Save And Set Mode
- BSM - Branch and Set Mode

Getting your thoughts together: strategies in code design

Section Preview

- **Fundamentals**

  - **Programming Concepts**
  - **Source, Object, and Load Modules**
  - **Memory and Data Representation**
  - **Addresses**
  - **The CPU**
  - **Setting Up For Programming (Machine Exercise)**
  - **Machine Instruction Formats**
  - **Base / Displacement Addresses**
  - **Assembler Language and the High Level Assembler (HLASM)**
  - **Basic Program Structure Requirements**

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)

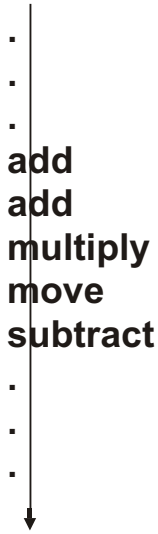
Transformation (move, manipulate, change)

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:**

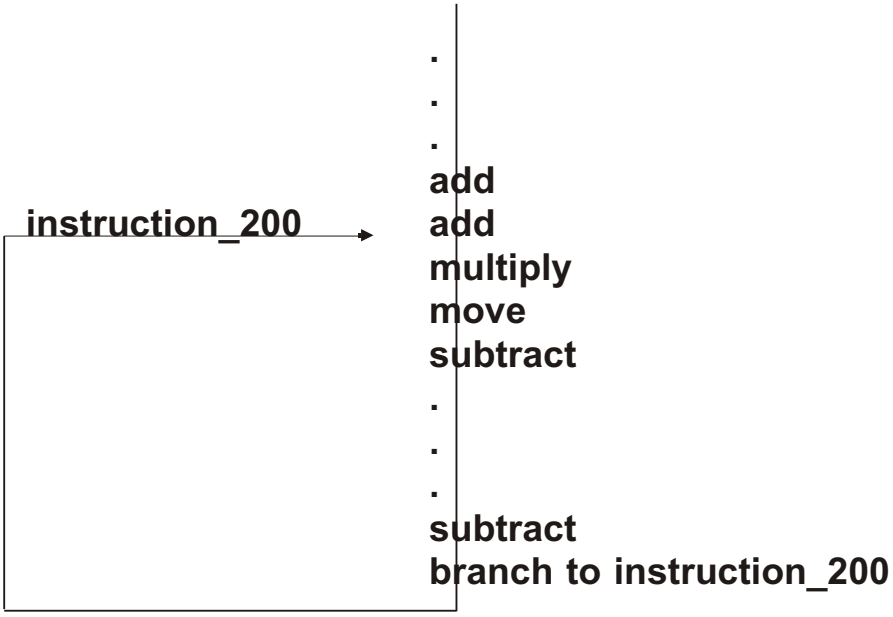


- **So the order in which you code (write) the instructions is the order in which they will execute**

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

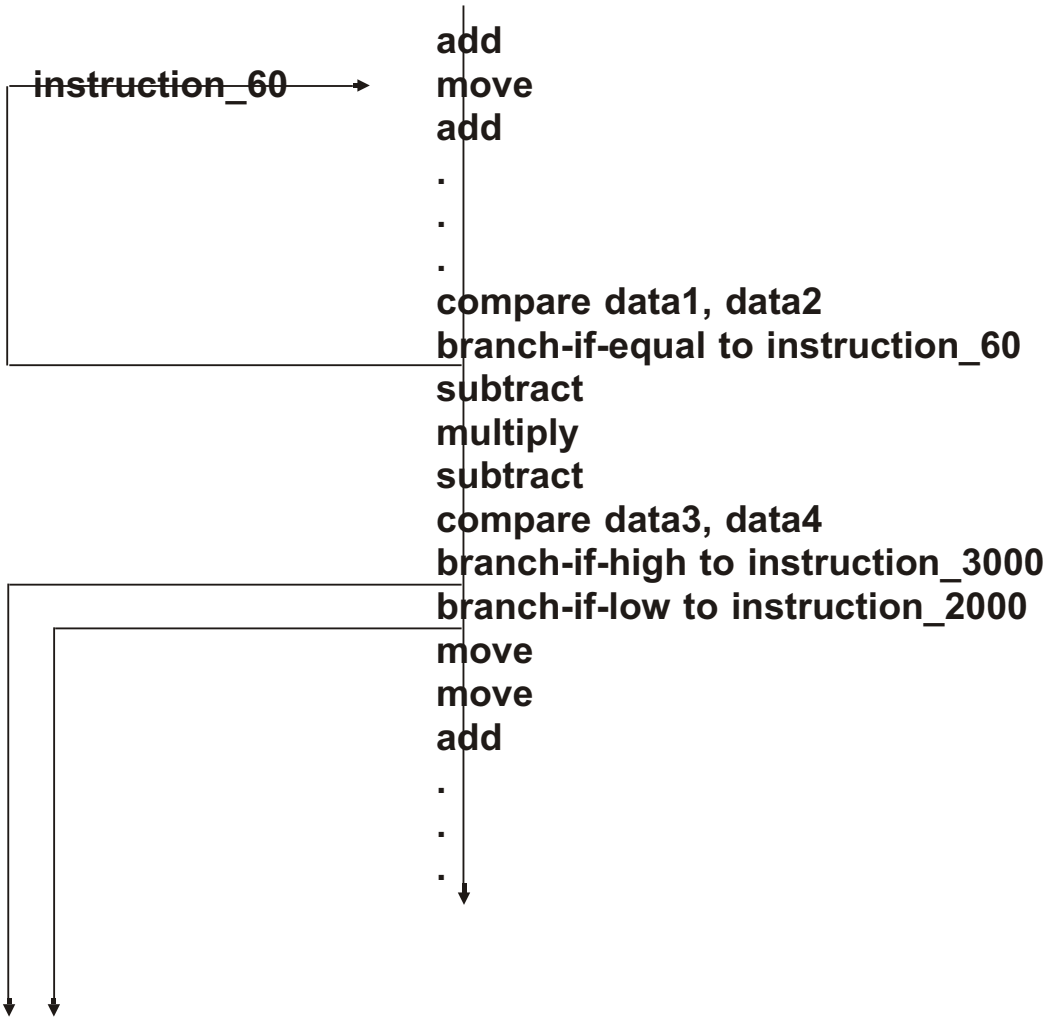
- This enables the computer to repeat a set of instructions as often as necessary:



- 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

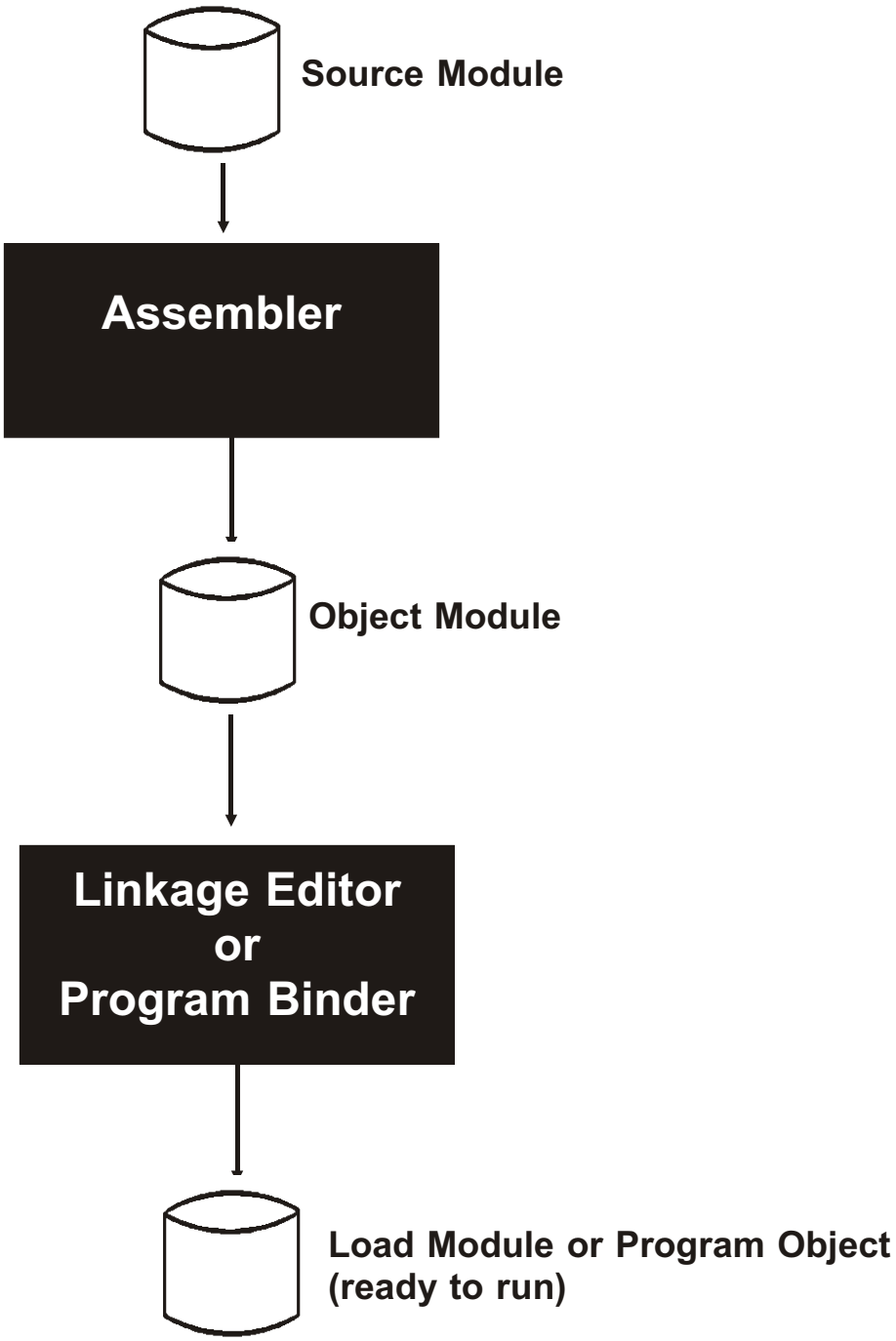
- **Feed one or more object modules into an IBM-supplied program called the Linkage Editor**

  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 replaced 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

operation

A word, abbreviation, or mnemonic that describes the actual operation the computer is to perform (for example: add, move)

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 string data because the hardware does not have any predetermined length or maximum size for this kind of data:**

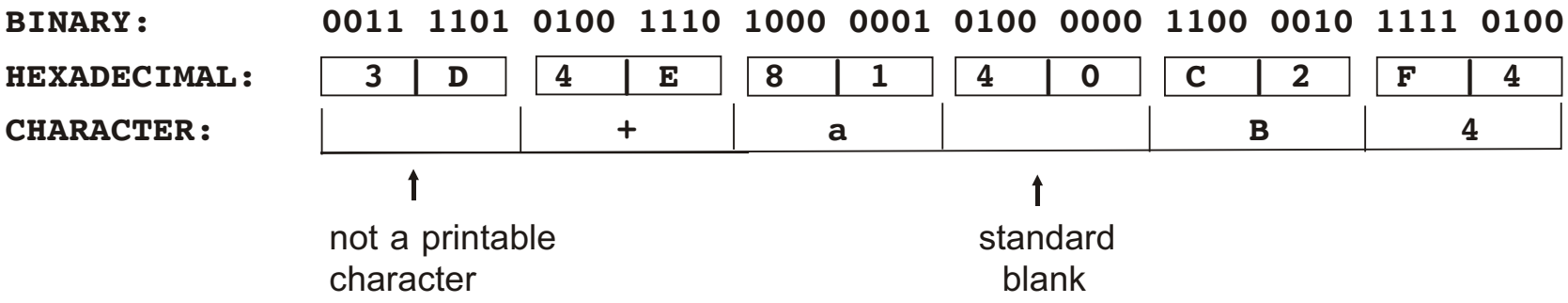
✗ **You can string characters together as long as you like**

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**

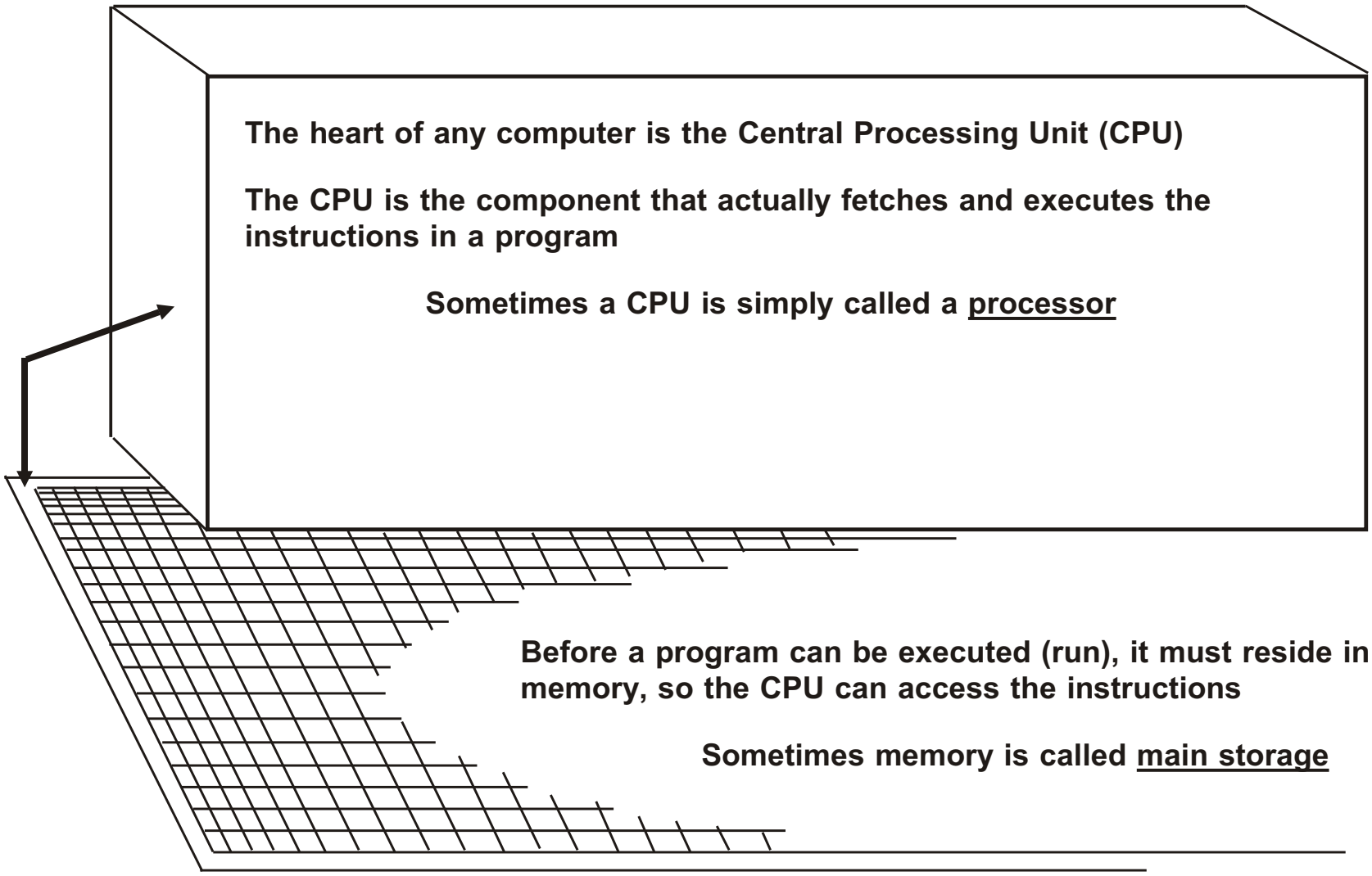
  A short-hand: one hex digit for each four bits (half-byte or nybble)

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)

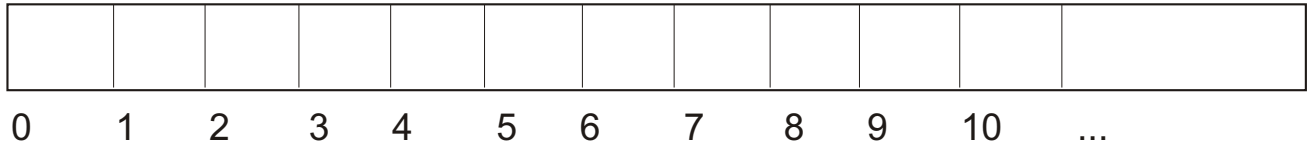
A CPU and Memory (Main Storage)



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

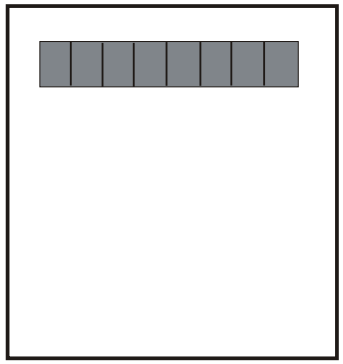
Address Registers

- **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

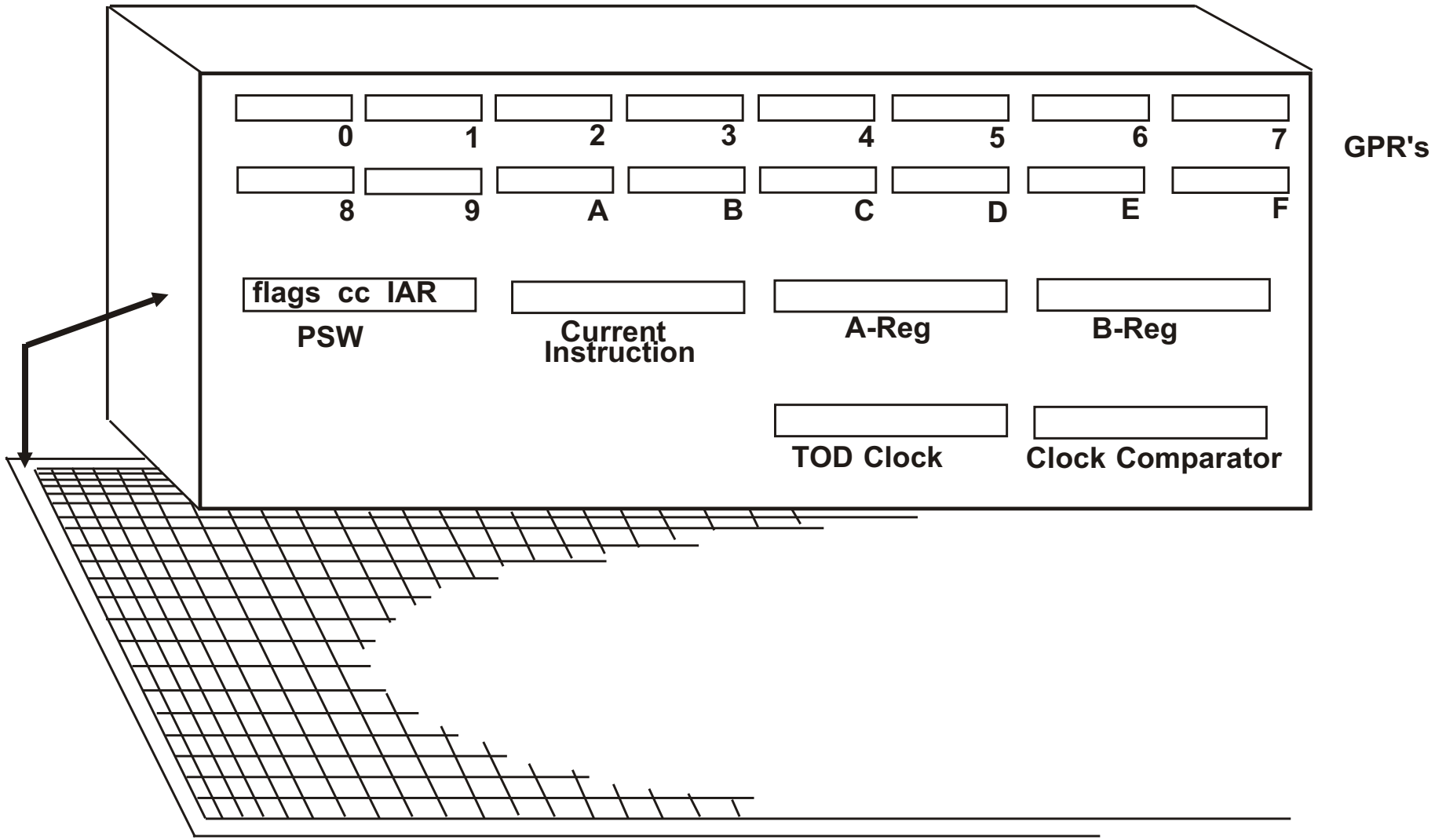
❒ **Here are some sample addresses when the CPU is using 31-bit addressing mode**

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)

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 if treated as unsigned binary integer
+ Decimal range -2,147,483,648 to +2,147,483,647 as signed integer

16 Floating Point Registers (not shown)

+ Eight bytes each, work with data in "excess-64 floating point" (S/360) format or in "binary floating point" (IEEE) format

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 indicate result of operation

Addressing Mode Indicator

+ 1 bit, indicates if 24-bit mode (bit is 0) or 31-bit mode (bit is 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 the 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 Circuitry
  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 (optional)
  Includes registers and instructions to perform operations on arrays of data simultaneously

+ Cryptographic facility (optional)
  Provides encoding and decoding services

Notes

The S/390 floating point registers are designated 0, 2, 4, and 6

Newer machines have 16 floating point registers; also, the registers can work with data in traditional IBM floating point format (also called "hexadecimal floating point") or IEEE floating point format (also called "binary floating point")

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

- **This course covers the original instruction set and instructions that were introduced in the first 20 years of the product line (roughly 1963~1981)**
  - ✗ Probably 90% of Assembler applications are written using only these instructions

  And developers are loathe to use instructions that might not be present on a machine when a customer installs their product

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

  We will use many of these improvements, since the Assembly process isn't limited by the target machine hardware

  If you have to maintain older code, it was obviously developed using the earlier Assemblers and so couldn't take advantage of the new features

Computer Exercise: Setting Up For Programming

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(c410strt)' 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 three libraries for you, one to hold your source code, one to hold your JCL, and one to hold load modules you will be creating throughout the class.

The library names are <hlq>.TR.SOURCE, <hlq>.TR.CNTL, and <hlq>.TR.LOAD, where "<hlq>" is replaced by the high level qualifier you entered in response to the setup's prompt.

Machine Instruction Formats

- **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:**

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)

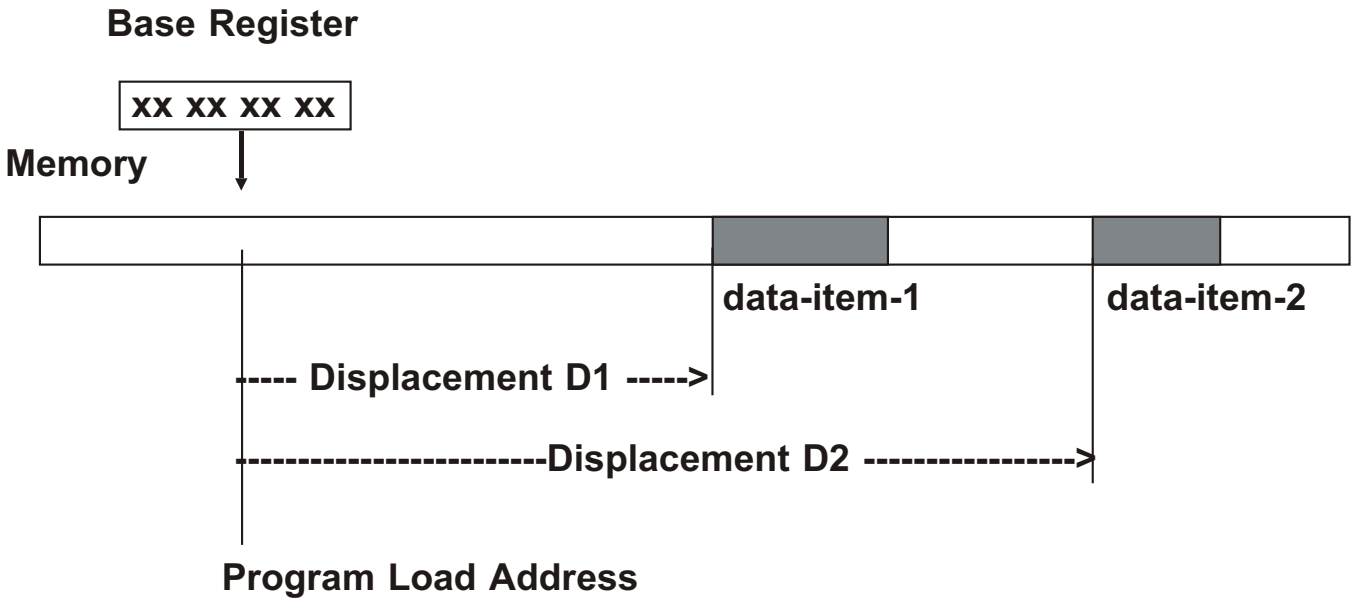
  This is how a program establishes its Base Address

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

1 Byte 1 Byte

❒ **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 (not discussed in this course)

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/390 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 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

Assembler Rules and Conventions

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

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 (optional)

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

Blank lines are allowed (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 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                Source                          (storage size,
Counter                 Instruction                      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

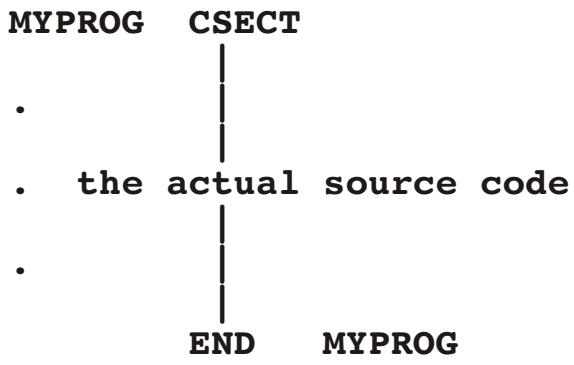
"starting-location" represents where in the program execution should begin when the program is actually run (the Entry Point)

✗ 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:**



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 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 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**

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

- One last 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
```

Computer Exercise: Coding, Assembling, Linking, Running

Now you are ready to code a simple Assembler program. Really simple.

1. Code a program called ASMEX1. This program simply enters and returns, and is based on the code on pages 50 to 52 (without the ellipsis (three dots) of course).

   Code this program as a member in your TR.SOURCE library.

2. To Assemble, link, and run this program, use the procedure called ASM1RUN1 in your TR.CNTL library. This JCL will work for most of the remaining exercises, with only minor modifications.

This program will not produce any output yet, although you will get listings from the Assembler and the Program Binder.

The purpose of this exercise is to get the basic logistics of coding, assembling, linking and running programs for the class environment taken care of. Also, you will now have a prototype program that follows basic linkage conventions as a model for future programs.