

CHAPTER 2: DO YOU Zee WHAT I Zee?

In which we examine the machine components that affect the programs we write.

Where the Rubber Meets the Road

The term “machine architecture” refers to the design of the electronic components that comprise a computer. Since we are studying assembly language, we are most interested in the design of the components that can be seen by an assembler programmer and that affect the programs we write. These components include memory, registers, the program status word, and the CPU. In the early 1960’s IBM adapted the architecture of its IBM 1400 series to produce a durable architecture known as the System/360. The “360” in the name referred to 360 degrees - the number of degrees in a circle. (System/360 machines were marketed as “all-purpose” machines, capable of handling business as well as scientific applications.) IBM also committed to supporting long-term software development by making the architecture of its future machines “backward compatible” with the System/360.

In the early 1970’s the architecture was modified to allow access to larger amounts of memory and was renamed the System/370 and System/370 XA architecture. IBM machine architecture has continued to evolve through many versions and today is known as the System/z. One of the major changes to the architecture has been the number of bits used to create an address. This has increased from the original 24 to 31 and is currently 64 bits. Today’s machines can operate using tri-modal addressing, switching between addressing modes when needed.

Throughout this evolution, the architecture has remained backward compatible. In fact, programs that were written to run on 1960’s era machines can run on current machines with little or no change in the code. The IBM mainframe has been an incredibly stable platform to develop on, and this fact has accounted for much of the success of this family of machines. We begin our discussion of the architecture by first looking at the main storage memory of these machines.

MEMORY

The smallest unit of storage memory is a **bit (binary digit)**. A bit can hold a binary digit, 0 or 1, and is represented internally in the machine as low or high voltage respectively. Bits are organized into consecutive groups called **bytes**. Given 8 bits, there are $2^8 = 256$ possible bit patterns that can be formed.

Binary Pattern / Decimal Equivalent

00000000 = 0

00000001 = 1

00000010 = 2

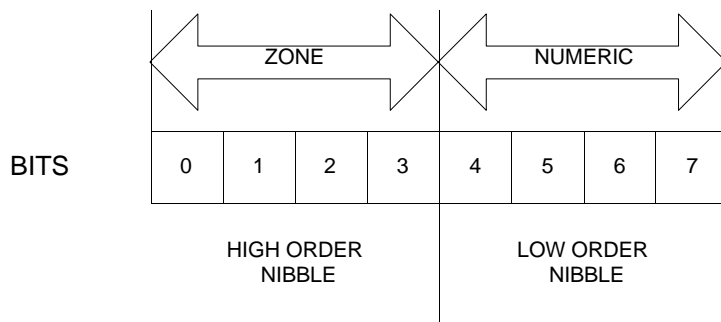
00000011 = 3

00000100 = 4

...
 11111101 = 253
 11111110 = 254
 11111111 = 255

Each of these two hundred-and-fifty-six-bit patterns is used to represent one character of data in the EBCDIC encoding sequence. EBCDIC is an acronym for “Extended Binary Coded Decimal Interchange Code”. This is a fancy name for a code created by IBM to represent characters in memory. For instance, 11000001 represents a character “A”, and 11110001 represents a character ‘1’ in the EBCDIC encoding sequence. ASCII is another binary code which is commonly used on non-IBM machines. While IBM mainframes have traditionally used EBCDIC, modern machines can work with many different encodings.

Each byte can be divided into two 4-bit areas called the **zone** and **numeric** portions as pictured below. Programmers will refer to a half-byte as a “nibble” or sometimes “nybble”. These are colloquial terms not found in the IBM documentation. Nevertheless, we will sometimes refer to bits 0 - 3 as the “High Order Nibble” and bits 4 - 7 as the “Low Order Nibble”.



Bytes are arranged consecutively and numbered, starting with the first byte, which is numbered 0 as indicated in the diagram below. The number assigned to each byte is called an **address**.



A byte is the smallest unit of storage on the machine that has an address. Although we can manipulate bits with a variety of instructions, bits do not have addresses.

Consecutive bytes are arranged into groups called **fields**. The address of a field is denoted by the address of the first byte that is in the field. We will revisit this point when we talk about base/displacement addressing.

For backward compatibility, the original System/360 instructions that operated on 32-bit registers now use bits 32-63, leaving bits 0-31 unchanged. Some instructions operate on bits 0- 31 as a 32-bit register, leaving bits 32-63 of a 64-bit register unchanged. There are also a large number of instructions that operate on all 64 bits as a single register value.

2) Floating-point - There are 16 floating-point registers available. These registers are used for scientific data processing where the values they denote can be quite large or small, and typically require a large number of decimal places. Values are expressed in a scientific notation. Each floating-point register is 64 bits in length. Current machine support three types of floating-point arithmetic: binary, hexadecimal and decimal. Binary and hexadecimal floating-point arithmetic is wonderful for scientific work but inappropriate for business arithmetic because it is not exact. Decimal floating-point, on the other hand, is exact for decimal calculations. Since this book is targeted at beginning assembler programmers, I have not written a chapter covering decimal floating-point arithmetic. I hope to add that later.

There are many other registers on Z-class machines (Access and Control registers), but in this book we are focusing on general-purpose registers - these are heavily used in business-related assembler programming. For the rest of the book we will assume that the term “register” refers to a general-purpose register unless otherwise specified.

PROGRAM STATUS WORD (PSW)

The Program Status Word or PSW is a collection of 128 bits (not a word) that describe the current status of the machine. There are numerous fields in the PSW and two of these fields are of special importance for programmers:

1) **Instruction Address** - This 64-bit field contains the address of the next instruction that the machine will execute. The machine uses this field to provide sequential execution of instructions and branching. (Technical note – IBM/z-class machines are capable of out-of-order and parallel execution of instructions in order to speed things along, but from a programmer’s viewpoint, it seems as if instructions are sequentially executed.)

2) **Condition Code** - This 2-bit field indicates the results of all comparison operations and most arithmetic operations as well. The condition code has four settings:

Equal	- 00
Low	- 01
High	- 10
Overflow	- 11

After the condition code is set, it can be tested with branch and jump instructions as illustrated in the example below.

```
CLC    CUSTTYPE,NEWCUST    COMPARE AND SET THE CONDITION CODE
BE     THERE               TEST THE CONDITION CODE
...
THERE DS    0H
```

CLC is the mnemonic for a Compare Logical Character instruction which is used to compare the two fields CUSTTYPE and NEWCUST, byte-by-byte. This instruction sets the two-bit condition code to reflect how the contents of the two fields compare. Subsequently, the condition code is tested with BE, a Branch Equal instruction. This instruction examines the condition code and causes execution to continue at the label THERE (by modifying the PSW instruction address) if the condition code is set to 00. Otherwise, flow of control continues with the next sequential instruction following the branch. The condition code remains set until it is changed by executing another comparison or arithmetic operation. By coding a sequence of comparisons and branches, we are able to build the logic of our program.

THE FETCH/DECODE/EXECUTE CYCLE

The PSW instruction address field is used in a process called the “Fetch/Decode/Execute” cycle. This process describes, algorithmically, how a computer operates. Conceptually, it is quite simple, since the Fetch/Decode/Execute cycle is just a “loop” which the CPU continually executes.

The cycle consists of the following 5 steps:

- 1) Fetch the instruction whose address is in the PSW instruction address field.
- 2) Decode the instruction which was fetched. (Among other things, the length of the current instruction is determined, and the addresses of operands are computed.)
- 3) Update the PSW instruction address field (by adding the current instruction length) so that it points at the next sequential instruction in the program.
- 4) Execute the decoded instruction.
- 5) Go back to step 1.

Cycling through the Fetch/Decode/Execute cycle, the machine processes instructions in a sequential fashion until execution of a branch or jump instruction modifies the instruction address to contain a non-sequential location in the program. Changing the PSW address causes the CPU to fetch an instruction other than the next sequential one.

Branch instructions use two different techniques for altering the PSW instruction address. For older instructions like the BE above, the current instruction address is simply replaced with the address specified in the branch instruction. Newer instructions provide for relative branching. In this case, a two’s complement integer inside the instruction is multiplied by two and the result is added to the PSW instruction address to determine the new PSW instruction address. This technique allows us to branch forward or backward relative to the current instruction address. In both types of branching, the flow of execution is altered by a change in the instruction address.

You may wonder why the two’s complement integer is multiplied by two in all relative branches. Since there are no odd length instructions, if we didn’t multiply by two, all the odd numbered values would be wasted. In multiplying by two, we effectively double the distances we can jump forward or backward from the current location.

VISUALIZING THE MACHINE COMPONENTS

VisibleZ is a software tool that I designed to help visualize the components and processes that are used by assembler programmers: memory, registers, the PSW, the fetch/decode/execute cycle, base/ displacement addressing. It comes with over 400 object code programs that can be executed one step at a time to help you experiment with the assembler instruction set. To get the most out of this book, you can download it at my website:

<https://punctiliousprogrammer.com/>

PROBLEMS

- 1-1. What is the smallest unit of storage?
- 1-2. What is the smallest unit of storage with an address?
- 1-3. How many bits in a byte?
- 1-4. What is a fullword?
- 1-5. How many patterns can a single byte represent?
- 1-6. What encoding sequence is commonly used on System/z machines?
- 1-7. What is the zone portion of a byte?
- 1-8. What is the numeric portion of a byte?
- 1-9. How many bytes are contained in a doubleword? Quadword?
- 1-10. What is the address of the first byte on a System/z machine?
- 1-11. What is the address of a field in memory?
- 1-12. How many general-purpose registers exist on a System/z machine?
- 1-13. How many bits are in a general-purpose register?
- 1-14. How are the bits in a general-purpose register numbered?
- 1-15. The System/360 had 32-bit registers. Where are those bits located in a modern register?
- 1-16. For an assembler programmer, what are the two most important fields in the PSW?
- 1-17. What address is contained in the instruction address field of the PSW?
- 1-18. How large is the condition code? What values can it contain?
- 1-19. During the fetch/execute cycle, what is contained in the instruction address field?
- 1-20. How does a branch instruction affect the fetch/execute cycle?