System/z Architecture

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 invented a revolutionary 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 support long-term software development by making the architecture of 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. 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 finally 64 bits. Current 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** (**b**inary dig**it**). A bit can hold a binary digit, 0 or 1, and is represented internally in the machine as a high or low voltage. Bits are organized into consecutive groups called **bytes**. In the System/z architecture, a byte consists of 8 bits together with a "hidden" ninth bit that is used by the machine for parity checking. Since the parity bit is unavailable to programmers, we will ignore it, and assume that there are 8 bits in every byte. Given 8 bits there are $2^8 = 256$ possible bit patterns that can be formed.

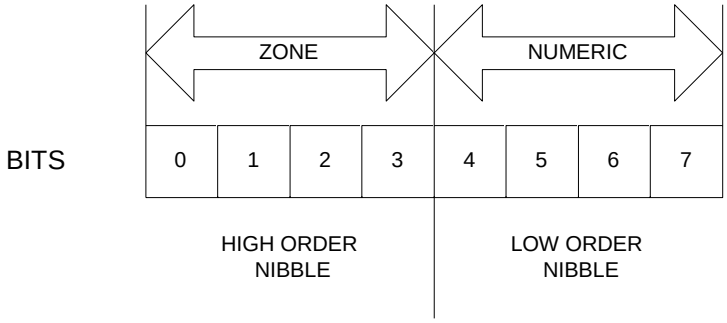
Binary Pattern / Decimal Equivalent

00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
00000100 = 4

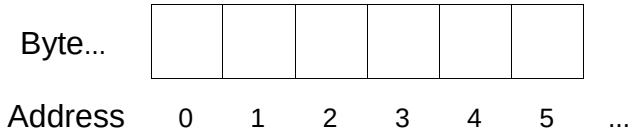
…
11111101 = 253
11111110 = 254
11111111 = 255

Each of the 256 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.

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". This is a colloquial term 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**. The idea of an address is an important: almost every reference to memory is made using 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.

A **halfword** is a 2-byte field that begins on an address that is evenly divisible by 2 (Addresses 0, 2, 4, 6 ...). Such an address is called a **halfword boundary**.

A **fullword** is a 4-byte field that begins on a **fullword boundary** (an address evenly divisible by 4). Fullwords are often referred to as "words".

A **doubleword** is an 8-byte field that begins on a **doubleword boundary** (an address that is evenly divisible by 8).

A **quadword** is a 16-byte field that begins on a **quadword boundary** (an address evenly divisible by 16).

In the diagram above which displays the first six bytes in memory, there are three halfwords (bytes 0 -1, bytes 2 - 3, and bytes 4 - 5), one fullword (bytes 0 - 3), the beginning bytes of a doubleword (bytes 0 - 7), and the beginning bytes of a quadword (bytes 0-15).

There are a host of terms listed below used to describe memory sizes. The names for memory differ based on whether you are using powers of 2 or 10.

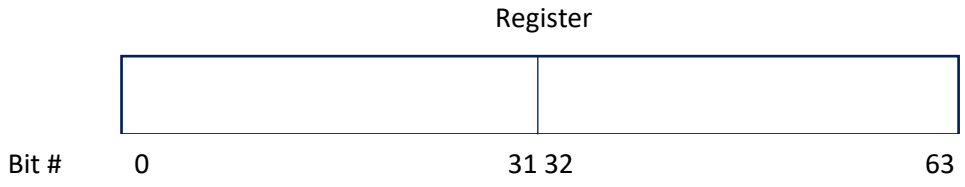
1000^1	KB	kilobyte	2^{10}	KiB	kibibyte
1000^2	MB	megabyte	2^{20}	MiB	mebibyte
1000^3	GB	gigabyte	2^{30}	GiB	gibibyte
1000^4	TB	terabyte	2^{40}	TiB	tebibyte
1000^5	KB	petabyte	2^{50}	PiB	pebibyte
1000^6	MB	exabyte	2^{60}	EiB	exbibyte
1000^7	GB	zettabyte	2^{70}	ZiB	zebibyte
1000^8	TB	yottabyte	2^{80}	YiB	yobibyte

The System/360 architecture specified a 24-bit address size when referencing bytes in main storage. With 24 bits, the maximum address that could be constructed (24 consecutive binary 1's) is equal to $2^{24} - 1$. This 24-bit address size provided the machine with $2^4 \times 2^{20} = 16$ MiB of memory. The architecture was extended in the 1970's in order to support 31-bit addresses, providing $2^1 \times 2^{30} = 2$ GiB. Today's System/z has a 64-bit address and provides for a memory space of $2^4 \times 2^{60} = 16$ EiB. While the address space is 16Eib, the physical memory supplied on actual machines is smaller than this. There is some ambiguity whenever we use a term like gigabyte as to whether we are thinking in base 2 or base 10. The terms on the right above were designed to remove this ambiguity.

REGISTERS

A **register** consists of special circuitry devoted to high speed arithmetic, addressing main storage locations, and providing control information for the operating system. There are two types of registers that require our attention as beginning assembler programmers:

1) **General purpose** - There are 16 general purpose registers numbered 0 through 15 that are available to an assembler programmer. These registers have two uses: 1) high-speed binary arithmetic, and 2) holding an address or index. Each general purpose register contains 64 bits which are numbered 0-63 moving from left to right.



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 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.

There are many other registers on Z-class machines including Access and Control registers, but in this article we are focusing only on general purpose registers - these are heavily used in business-related assembler programming.

PROGRAM STATUS WORD (PSW)

The Program Status Word or PSW is a collection of 128 bits 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.

2) **Condition Code** - This 2-bit field indicates the results of all comparison operations and a few 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     EQU  *
```

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.