**The Z80 Microprocessor**

Navigation  
[Home](http://penguicon.sourceforge.net/comphist/links/cpm/index.html)  
[email](mailto:cp-m@apexmail.com)

Background  
[History of CP/M](http://penguicon.sourceforge.net/comphist/links/cpm/history.html)  
[Architecture of CP/M](http://penguicon.sourceforge.net/comphist/links/cpm/architec.html)

Using CP/M  
[Commands and the CCP](http://penguicon.sourceforge.net/comphist/links/cpm/ccp.html)  
[Programming with BDOS](http://penguicon.sourceforge.net/comphist/links/cpm/bdos.html)  
[The BIOS interface](http://penguicon.sourceforge.net/comphist/links/cpm/bios.html)

The Z80 CPU  
[Architecture of the Z80](http://penguicon.sourceforge.net/comphist/links/cpm/z80.html)  
[The Z80 instruction set](http://penguicon.sourceforge.net/comphist/links/cpm/z80_inst.html)

Software Tools  
Installing an Emulator  
Zip, ark, crunch, and urgh  
Editors, Assemblers, Debuggers

Installing CP/M  
Assembling a BIOS  
Replacing the BDOS  
Replacing the CCP

Reference  
Memory Map  
Data Structures  
Glossary

**History**

In 1969 Intel were approached by a Japanese company called Busicom to produce chips for Busicom's electronic desktop calculator. Intel suggested that the calculator should be built around a single-chip generalized computing engine and thus was born the first microprocessor - the 4004. Although it was based on ideas from much larger mainframe and mini-computers the 4004 was cut down to fit onto a 16-pin chip, the largest that was available at the time, so that its data bus and address bus were each only 4-bits wide.

Intel went on to improve the design and produced the 4040 (an improved 4-bit design) the 8008 (the first 8-bit microprocessor) and then in 1974 the 8080. This last one turned out to be a very useful and popular design and was used in the first home computer, the Altair 8800, and CP/M.

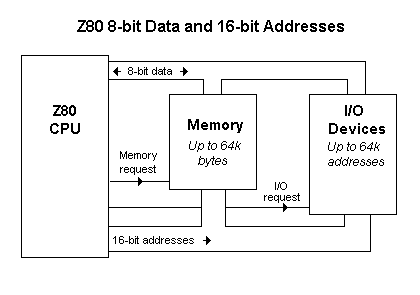
In 1975 Federico Faggin who had had worked at Intel on the 4004 and its successors left the company and joined forces with Masatoshi Shima to form Zilog. At their new company Faggin and Shima designed a microprocessor that was compatible with Intel's 8080 (it ran all 78 instructions of the 8080 in exactly the same way that Intel's chip did) but had many more abilities (an extra 120 instructions, many more registers, simplified connection to hardware). Thus was born the mighty Z80!

The original Z80 was first released in July 1976. Since then newer versions have appeared with exactly the same architecture but running at higher speeds. The original Z80 ran with a clock rate of 2.5 MHz, the Z80A runs at 4MHz, the Z80B at 6MHz, and the Z80H at 8mhz.

Many companies produced machines based around Zilog's improved chip during the 1970's and 80's and because the chip could run 8080 code without needing any changes to the code the perfect choice of operating system was CP/M.

So, let's dive into the Z80 and see what's inside...

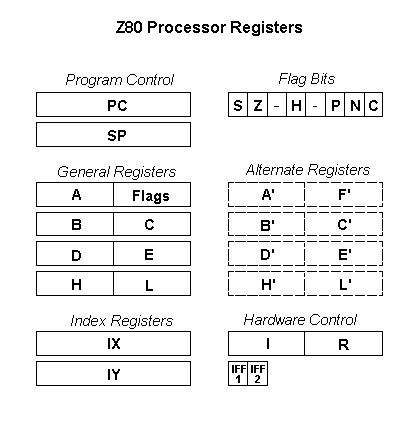
**8-bit data, 16-bit addresses**



The Z80 uses 8-bit bytes which are stored in memory. These bytes contain both the program that the processor is executing and the data items that the program is working on. The processor uses 16-bit addresses to access these bytes, so there can be anything up to 64k (65536) bytes of memory. How much memory is actually available, and how much of it is read only (ROM) or random access (RAM) varies from one machine to another: in a simple controller there might be just 4k of ROM holding the program and 1k of RAM for the data items; in a large CP/M system you would find a full 64k of RAM.

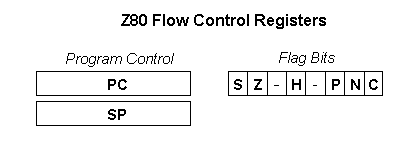
The processor also uses a separate 64k of addresses for input and output devices. This I/O address space is only used by a few instructions (called, not surprisingly, IN and OUT) to send or receive bytes from I/O devices. The addresses for I/O instructions use the same wiring as the addresses for memory and the data bytes travel through the same wiring that is used for bytes in memory so the processor uses special control signals to differentiate between memory accesses and I/O operations. These control signals have to be decoded by I/O devices together with the addresses sent by the processor so that devices do not mistakenly respond to requests that were intended to read or write bytes in memory.

**Programmer's Model**



All programming of the Z80 revolves around its internal registers. These registers are used to control the flow of the program and to operate on data items. There are 16-bit registers inside the processor that address the program bytes in memory and there are 8-bit registers into which data bytes are loaded and then operated on. These 8-bit registers can also be combined in pairs to form 16-bit addresses and these addresses can be used to index data structures in memory that span several bytes.

**Flow of Control Registers**

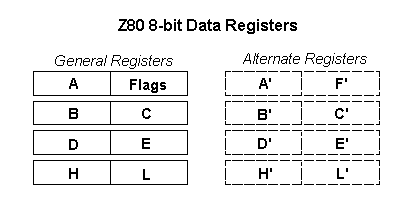


The Z80 uses a 16-bit program counter (PC) to hold the memory address of the next instruction to execute. This register gradually steps through memory as instructions are executed, but some instructions can alter it directly to cause the flow of the program to branch to a new location.

The Z80 also supports subroutines calls with a stack pointer (SP). When a subroutine is called the contents of PC are pushed into the memory location pointed to by SP and the stack pointer is decremented. When the subroutine has finished executing the stack pointer is incremented and the contents of PC popped back from the memory location. In this way calls to subroutines can be nested to any depth, as long as there is space to push the program counter into memory for each call.

The Z80 also contains flags to control the flow of a program. The flags are changed whenever an arithmetic operation is done and the flags can be tested one at a time by a jump instructions to change the flow of the program, depending on the state of the flag. The flags record the sign (S), zero (Z), half-carry (H), parity (P), and carry (C) of the result of the last arithmetic instruction and all of these can be tested by jump instructions. There is also a flag that records whether the last arithmetic operation was an addition or subtraction (N) which is only used to decimal arithmetic. These six flags are stored in a single byte (F) with bits 3 and 5 left unused.

**8-bit Data Registers**



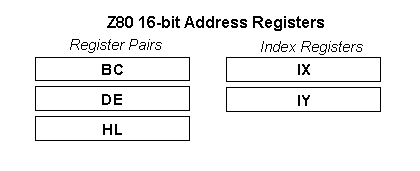
The Z80 processor contains 14 general purpose registers which can be used for holding and manipulating data bytes. Each of these registers is 8-bits wide and they are named A, B, C, D, E, H, L, A', B', C', D', E', H', L'.

As their names suggest the registers are divided into two sets. The first set are the registers A..L and the second set, called the alternate registers, are A'..L'. Most operations in the processor only work on the first set and the alternates just provide temporary space to save the contents of A..L. The Z80 has short instructions that exchange the contents of the working registers with the contents of the alternates. Of course, this temporary space could easily be allocated in memory but in applications where timing is critical it is much quicker to exchange the working registers and the alternates rather than having to save the working registers and then load new values from memory.

Once data has been loaded into the working registers the Z80 has a full range of arithmetic, logical, shifting, and rotating instructions to handle 8-bit bytes. The results of these operations generally end up in register A (the accumulator) from where they can be moved to other registers or saved back to memory locations.

In a few instructions the flags are treated as a single byte that is called the F register.

**16-bit Address Registers**



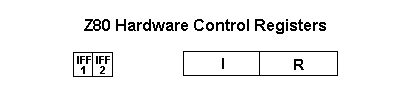
The Z80 contains two types of 16-bit addressing registers: register pairs and index registers.

Register pairs are formed from combinations of the 8-bit general purpose registers. The Z80 allows three combinations: registers B and C can be combined to form a 16-bit register that is called BC, registers D and E combined to form DE, and H and L combined to form HL.

These register pairs can be used to do some limited 16-bit arithmetic (addition, subtraction, increment, decrement) but generally they are used as pointers to data in memory. The instruction set is heavily biased towards using HL as a data pointer. Many arithmetic instructions can combine the accumulator A with the byte in memory that is pointed to by HL. In this way the byte in memory does not have to be loaded into a internal register before it is combined with the accumulator, saving instructions and register space.

The processor also contains two index registers (IX and IY) that can be used to address data in memory. These registers work very like the HL register pair in that many instructions can use a byte that is addressed by IX or IY as an operand. However, unlike HL, an offset is added to the value in the index register before addressing memory. This offset is contained in the instruction that is using IX or IY and it is an 8-bit value. This helps the programmer create fancy data structures in memory using an index register to point to the start of the structure and the 8-bit offset to select fields within the structure.

**Hardware Registers**



The Z80 also contains a few registers that let the user control its hardware in some simple ways.

Interrupts can be controlled with two flags (IFF.1 and IFF.2) and an 8-bit page pointer (I). The programmer can enable and disable some interrupts by setting and clearing IFF.1. When an interrupt does happen the current contents of IFF.1 are saved in IFF.2 and then IFF.1 is set so that further interrupts are disabled. When the program has finished handling the interrupt IFF.2 is copied back to IFF.1, restoring the flag to its original value.

**Instruction Set**

These links will take you to tables that describe the Z80 instructions in detail. They are broken down into groups of similar instructions:

[8-bit Load and Store](http://penguicon.sourceforge.net/comphist/links/cpm/z80_8ld.html) - Using the general purpose registers  
[16-bit Load and Store](http://penguicon.sourceforge.net/comphist/links/cpm/z80_16ld.html) - Using register pairs, index registers, and the stack pointer

1-bit Arithmetic - Set a bit, Clear a bit, Test a bit

8-bit Arithmetic - Add, Subtract, And, Or, etc.  
8-bit Shift and Rotate - Shift left, Shift right, Rotate, Insert carry bit, etc.

16-bit Arithmetic - Add, Subtract, Increment, Decrement

Flag Control - Set carry flag, Clear carry, Enable and Disable interrupts  
Program Flow - Jump, Conditional jump, Call subroutine

I/O - Input a byte, Output a byte  
Block Instructions - Move block of memory, Search a block of memory, Input or Output a block

3-Aug-98 22:38:42

# Z80 HARDWARE ORGANIZATION

## 2.1 INTRODUCTION

In order to program at an elementary level, it is not necessary to understand in detail the internal structure of the processor that one is using. However, in order to do efficient programming, such an understanding is required. The purpose of this chapter is to present the basic hardware concepts necessary for understanding the operation of the Z80 system. The complete microcomputer system includes not only the microprocessor unit (here the Z80), but also other components. This chapter presents the Z80 proper, while the other devices (mainly input/output) will be present in a separate chapter (Chapter 7).

We will review here the basic architecture of the microcomputer system, then study more closely the internal organization of the Z80. We will examine, in particular, the various registers. We will then study the program execution and sequencing mechanism. From a hardware standpoint, this chapter is only a simplified presentation. The reader interested in gaining detailed understanding is referred to our book ref. C201 ("Microprocessors," by the same author).

The Z80 was designed as a replacement for the Intel 8080, and to offer additional capabilities. A number of references will be made in this chapter to the 8080 design.

## 2.2 SYSTEM ARCHITECTURE

The architecture of the microcomputer system appears in [Figure 2.1](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.1). The microprocessor unit (MPU), which will be a Z80 here, appears on the left of the illustration. It implements the functions of a central-processing unit (CPU) within one chip: it includes an arithmetic-logical unit (ALU), plus its internal registers, and a control unit (CU), in charge of sequencing the system. Its operation will be explained in this chapter.

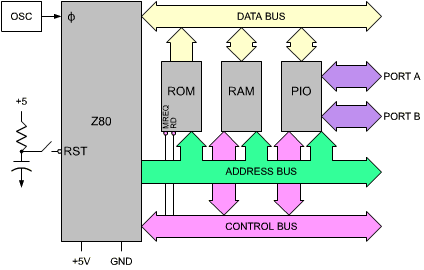
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-1.gif)

Fig. 2.1: Standard Z80 System

The MPU creates three buses: an 8-bit bidirectional data bus, which appears at the top of the illustration, a 16-bit unidirectional address bus, and a control bus, which appears at the bottom of the illustration. Let us describe the function of each of the buses.

The data bus carries the data being exchanged by the various elements of the system. Typically, it will carry data from the memory to the MPU or from the MPU to the memory or from the MPU to an input/output chip. (An input/output chip is a component in charge of communicating with an external device.)

The address bus carries an address generated by the MPU, which will select one internal register within one of the chips attached to the system. This address specifies the source, or the destination, of the data which will transit along the data bus.

The control bus carries the various synchronization signals required by the system.

Having described the purpose of the buses, let us now connect the additional components required for a complete system.

Every MPU requires a precise timing reference, whichis supplied by a clock and a crystal. In most "older" microprocessors, the clock-oscilator is external to the MPU and requires an extra chip. In most recent microprocessors, the clock-oscilator is usually incorporated within the MPU. The quartz crystal, however, because of its bulk, is always external to the system. The crystal and the clock appear on the left of the MPU box in [Figure 2.1](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.1).

Let us now turn our attention to the other elements of the system. Going from the left to right on the illustration, we distinguish:

The ROM is the read-only memory and contains the program for the system. The advantage of the ROM memory is that its contents are permanent and do not disappear whenever the system is turned off. The ROM, therefore, always contains a bootstrap or a monitor program (their function will be explained later) to permit initial system operation. In a process-control environment, nearly all the programs will reside in ROM, as they will probably never be changed. In such a case, the industrial user has to protect the system against power failure; programs must not be volatile. They must be in ROM.

However, in a hobbyist environment, or in a program-development environment (when the programmer tests his program), most of the programs will reside in RAM, so that they can be easily changed. Later, they may remain in RAM, or be transferred into ROM, if desired. RAM, however, is volatile. Its contents are lost when power is turned off.

The RAM (random-access memory) is the read/write memory for the system. In the case of a control system, the amount of RAM will typically be small (for data only). On the other hand, in a program development environment, the amount of RAM will be large, as if will contain programs plus development software. All RAM contents must be loaded prior to use from an external device.

Finally the system will contain one or more interface chips so that it may communicate with the external world. The most frequently used interface chip is the PIO or parallel input/output chip. It is the one shown on the illustration. This PIO, like all other chips in the system, connects to all three buses and provides at least two 8-bit ports for communication with the outside world. For more details on how and acutal PIO works, refer to book C201 or, for specifics of the Z80 system refer the Chapter 7 (Input/Output Devices).

All the chips are connected to all three buses, including the control bus.

The functional modules which have been described need not necessarily reside on a single LSI chip. In fact, we could use combination chips, which may include both PIO and a limited amount of ROM or RAM.

Still more components will be required to build a real system. In particular, the buses need to be buffered. Also decoding logic may be used for the memory RAM chips, and, finally, some signals may need to be amplified by drivers. These auxiliary circuits will not be described here as they are not relevant to programming. The reader interested in specific assembly and interfacing techniques is referred to book C207 "Microprocessor Interfacing Techniques."

## 2.3 INSIDE A MICROPROCESSOR

The large majority of all microprocessor chips on the market today implement the same architecture. This "standard" architecture will be described here. It is shown in [Figure 2.2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.2). The modules of this standard microprocessor will now be detailed, from right to left.

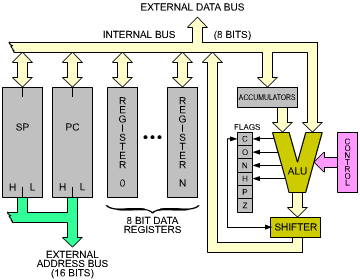
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-2.gif)

Fig. 2.2: "Standard" Microprocessor Architecture

The control box on the right represents the control unit which synchronizes the entire system. Its role will be clarified within the remainder of this chapter.

The ALU performs arithmetic and logic operations. A special register equips one of the inputs of the ALU, the left input here. It is called the accumulator. (Several accumulators may be provided.) The accumulator may be referenced as input and output (source and destination) within the same instruction.

The ALU must also provide shift and rotate facilities.

A shift operation consists of moving the contents of a byte by one or more positions to the left or to the right. This is illustrated in [Figure 2.3](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.3). Each bit has been moved to the left by one position. The details of shifts and rotations will be presented in the next chapter.

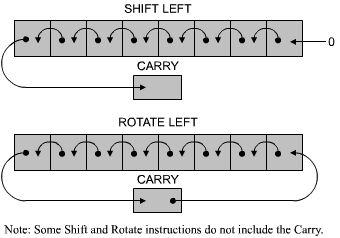
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-3.gif)

Fig. 2.3: Shift and Rotate

The shifter may be on the ALU output, as illustrated in [Figure 2.2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.2), or may be on the accumulator input.

To the left of the ALU, the flags or status register appear. Their role is to store exceptional conditions within the microprocessor. The contents of the flags registers may be tested by specialized instructions, or may be read on the internal data bus. A conditional instruction will cause the execution of a new program, depending on the value of one of these bits.

The role of the status bits in the Z80 will be examined later in this chapter.

### 2.3.1 Setting Flags

Most of the instructions executed by the processor will modify some or all of the flags. It is important to always refer to the chart provided by the manufacturer listing which bits will be modified by the instructions. This is essential in understanding the way a program is being executed. Such a chart for the Z80 is shown in Figure 4.17.

### 2.3.2 The Registers

Let us look now at [Figure 2.2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.2). On the left of the illustration, the registers of the microprocessor appear. Conceptually, one can distinguish the general-purpose registers and the address registers.

### 2.3.3 The General-Purpose Registers

General-purpose registers must be provided in order for the ALU to manipulate data at high speed. Because of restrictions on the number of bits which is reasonable to provide within an instruction, the number of (directly addressable) registers is usually limited to fewer than eight. Each of these registers is a set of eight flip-flops, connected to the bidirectional internal data bus. These eight bits can be transferred simultaneously to or from the data bus. The implementation of these registers in MOS flip-flops provide the fastest level of memory available, and their contents can be accessed within tens of nanoseconds.

Internal registers are usually labeled from 0 to n. The role of these registers is not defined in advance: they are said to be "general-purpose." They may contain any data used by the program.

These general-purpose registers will normally be used to store eight-bit data. On some microprocessors, facilities exist to manipulate two of these registers at a time. They are then called "register pairs." This arrangement facilitates the storage of 16-bit quantities, whether data or addresses.

### 2.3.4 The Address Registers

Address registers are 16-bit registers intended for the storage of addresses. They are also often called data counters or pointers. They are double registers, i.e., two eight-bit registers. Their essential characteristic is to be connected to the address bus. The address registers create the address bus. The address bus appears on the left and the bottom part of the illustration in [Figure 2.4](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.4).

The only way to load the contents of these 16-bit registers is via the data bus. Two transfers will be necessary along the data bus in order to transfer 16 bits. In order to differentiate between the lower half and the higher half of each register, they are usually labelled as L (low) or H (high), denoting bits 0 through 7, and 8 through 15, respectively. This label is used whenever it is necessary to differentiate the halves of these registers. At least two address registers are present within most microprocessors. "MUX" in [Figure 2.4](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.4) stands for multiplexer.

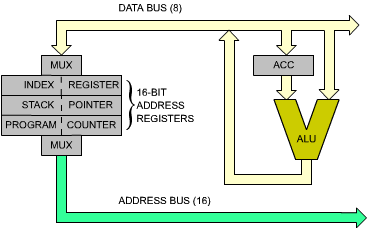
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-4.gif)

Fig. 2.4: The 16-bit Address Registers Create the Address Bus

#### 2.3.4.1 Program Counter (PC)

The program counter must be present in any processor. It contains the address of the next instruction to be executed. The presence of the program counter is indispensable and fundamental to program execution. The mechanism of program execution and the automatic sequencing implemented with the program counter will be described in the next section. Briefly, execution of a program is normally sequential. In order to access the next instruction, it is necessary to bring it from the memory into the microprocessor. The contents of the PC will be deposited on the address bus, and transmitted towards the memory. The memory will then read the contents specified by this address and send back the corresponding word to the MPU. This is the instruction.

In a few exceptional microprocessors, such as the two-chip F8, there is no PC on the microprocessor. This does not mean that the system does not have a program counter. The PC happens to be implemented directly on the memory chip, for reasons of efficiency.

#### 2.3.4.2 Stack Pointer (SP)

The stack has not been introduced yet and will be described in the next section. In most powerful, general-purpose microprocessors, the stack is implemented in "software", i.e., within the memory. In order to keep track of the top of this stack within the memory, a 16-bit register is dedicated to the stack pointer or SP. The SP contains the address of the top of the stack within the memory. It will be shown that the stack is indispensable for interrupts and for subroutines.

#### 2.3.4.3 Index Register (IX)

Indexing is a memory-addressing facility which is not always provided in microprocessors. The various memory-addressing techniques will be described in Chapter 5. Indexing is a facility for accessing blocks of data in the memory with a single instruction. An index register will typically contain a displacement which will be automatically added to a base (or it might contain a base which would be added to a displacement). In short, indexing is used to access any word within a block of data.

### 2.3.5 The Stack

A stack is formally called an LIFO structure (last-in, first-out). A stack is a set of registers, or memory locations, allocated to this data structure. The essential characteristic of this structure is that it is a chronological structure. This first element introduced into the stack is always at the bottom of the stack. The element most recently deposited in the stack is on top of the stack. The analogy can be drawn with a stack of plates on a restaurant counter. There is a hole in the counter with a spring in the bottom. Plates are piled up in the hole. With this organization, it is guaranteed that the plate which has been put first in the stack (the oldest) is always at the bottom. The one that has been placed most recently on the stack is the one which is on top of it. This example also illustrates another characteristic of the stack. In normal use, a stack is only accessible via two instructions: "push" and "pop" (or "pull"). The push operation results in depositing one element on top of the stack (two in case of the Z80). The pull operation consists of removing one element from the stack. In the case of a microprocessor, it is the accumulator that will be deposited on top of the stack. The pop will result in a transfer of the top element of the stack into the accumulator. Other specialized instructions may exist to transfer the top of the stack between other specialized registers, such as the status register. The Z80 is more versatile than most in this respect.

The availability of a stack is required to implement three programming facilities within the computer system: subroutines, interrupts, and temporary data storage. The role of the stack during subroutines will be explained in Chapter 3 (Basic Programming Techniques). The role of the stack during interrupts will be explained in Chapter 6 (Input/Output Techniques). Finally, the role of the stack in saving data at high speed will be explained during specific application programs.

We will simply assume at this point that the stack is a required facility in every computer system. A stack may be implemented in two ways:

1. A fixed number of registers may be provided within the microprocessor itself. This is a "hardware stack." It has the advantage of high speed. However, it has the disadvantage of a limited number of registers.
2. Most general-purpose microprocessors choose another approach, the software stack, in order not to restrict the stack to a very small number of registers. This is the approach chosen in the Z80. In the software approach, a dedicated register within the microprocessor, here register SP, stores the stack pointer, i.e., the address of the top element of the stack (or, sometimes, the address of the top element of the stack plus one). The stack is then implemented as an area of memory. The stack pointer will therefore require 16 bits to point anywhere in the memory.

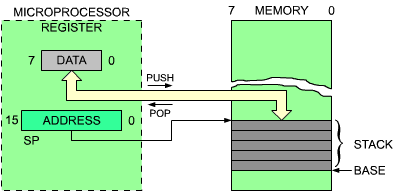
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-5.gif)

Fig. 2.5: The Two Stack-Manipulation Instructions

### 2.3.6 The Instruction Execution Cycle

Let us now refer to [Figure 2.6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.6). The microprocessor unit appears on the left, and the memory appears on the right. The memory chip may be a ROM or a RAM, or any other chip which happens to contain memory. The memory is used to store instructions and data. Here, we will fetch one instruction from the memory to illustrate the role of the program counter. We assume that the program counter has valid contents. It now holds a 16-bit address which is the address of the next instruction to fetch in the memory. Every processor proceeds in three cycles:

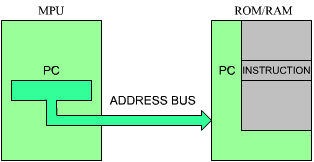
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-6.gif)

Fig. 2.6: Fetching an Instruction from the Memory

#### 2.3.6.1 Fetch

Let us now follow the sequence. In the first cycle, the contents of the program counter are deposited on the address bus and gated to the memory (on the address bus). Simultaneously, a read signal may be issued on the control bus of the system, if required. The memory will receive the address. This address is used to specify one location within the memory. Upon receiving the read signal, the memory will decode the address it has received, through internal decoders, and will select the location specified by the address. A few hundred nanoseconds later, the memory will deposit the eight-bit data corresponding to the specified address on its data bus. This eight-bit word is the instruction that we want to fetch. In our illustration, this instruction will be deposited on the data bus on top of the MPU box.

Let us briefly summarize the sequencing: the contents of the program counter are output on the address bus. A read signal is generated. The memory cycles, and perhaps 300 nanoseconds later, the instruction at the specified address is deposited on the data bus (assuming a single byte instruction). The microprocessor then reads the data bus and deposits its contents into a specialized internal register, the IR register. The IR is the instruction register: it is eight-bits wide and is used to contain the instruction just fetched from the memory. The fetch cycle is now completed. The 8 bits of the instruction are now physically in the special internal register of the MPU, the IR register. The IR appears on the left of [Figure 2.7](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.7). It is not accessible to the programmer.

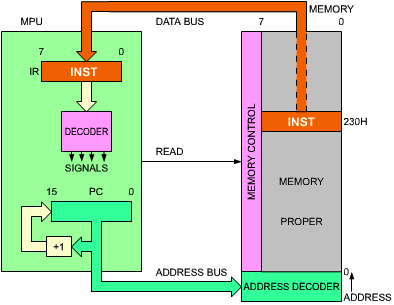
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-7.gif)

Fig. 2.7: Automatic Sequencing

#### 2.3.6.2 Decoding and Execution

Once the instruction is contained in IR, the control unit of the microprocessor will decode the contents and will be able to generate the correct sequence of internal and external signals for the execution of the specified instruction. There is, therefore, a short decoding delay followed by an execution phase, the length of which depends on the nature of the instruction specified. Some instructions will execute entirely within the MPU. Other instructions will fetch or deposit data from or into the memory. This is why the various instructions of the MPU require various length of time to execute. This duration is expressed as a number of (clock) cycles. Refer to Chapter 4 for the number of cycles required by each instruction. Since various clock rates may be used, speed of execution is normally expressed in number of cycles rather than in number of nanoseconds.

### 2.3.7 Fetching the Next Instruction

We have described now, using the program counter, an instruction can be fetched from the memory. During the execution of a program, instructions are fetched in sequence from the memory. An automatic mechanism must therefore be provided by a simple incrementer attached to the program counter. This is illustrated in [Figure 2.7](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.7). Every time that the contents of the program counter (at the bottom of the illustration) are placed on the address bus, its contents will be incremented and written back into the program counter. As an example, if the program counter contained the value "0", the value "0" would be output on the address bus. Then the contents of the program counter would be incremented and the value "1" would be written back into the program counter. In this way, the next time that the program counter is used, it is the instruction at address 1 that will be fetched. We have just implemented an automatic mechanism for sequencing instructions.

It must be stressed that the above descriptions are simplified. In reality, some instructions may be two- or even three-bytes long, so that successive bytes will be fetched in this manner from memory. However, the mechanism is identical. The program counter is used to fetch successive bytes of an instruction as well as to fetch successive instructions themselves. The program counter, together with its incrementer, provides an automatic mechanism for pointing to successive memory locations.

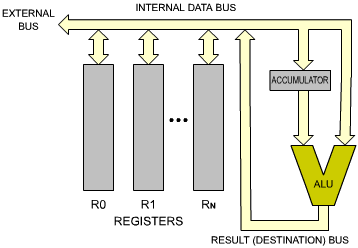
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-8.gif)

Fig. 2.8: Single-Bus Architecture

We will now execute an instruction within the MPU (see [Figure 2.8](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.8)). A typical instruction will be, for example: R0 = R0 + R1. This means: "ADD the contents of R0 and R1, and store the results in R0." To perform this operation, the contents of R0 will be read from register R0, carried via the single bus to the left input of the ALU, and stored in the buffer register there. R1 then will be selected and its contents will be read onto the bus, then transferred to the right input of the ALU. This sequence is illustrated in [Figures 2.9](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.9) and [2.10](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.10). At this point, the right input of the ALU is conditioned by R1, and the left input of the ALU is conditioned by the buffer register, containing the previous value of R0. The operation can be performed. The addition is performed by the ALU, and the result appears on the ALU output, in the lower right-hand corner of [Figure 2.11](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.11). The result will be deposited on the single bus, and will be propagated back to R0. This means, in practice, that the input latch of R0 will be enabled, so that data can be written into it. Execution of the instruction is now complete. The results of the addition are in R0. It should be noted that the contents of R1 have not been modified by this operation. This is general principle: the contents of a register, or any read/write memory, are not modified by a read operation.

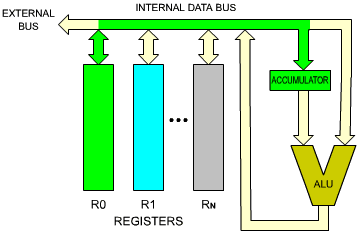
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-9.gif)

Fig. 2.9: Execution of an Addition - R0 into ACC

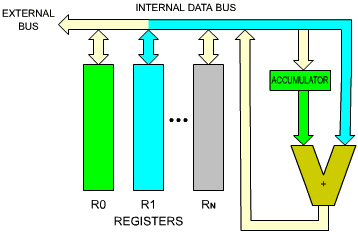
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-10.gif)

Fig. 2.10: Addition - Second Register R1 into ALU

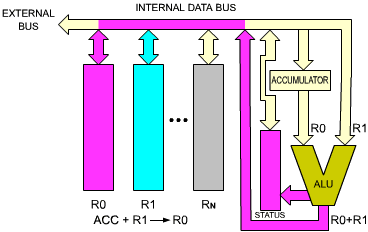
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-11.gif)

Fig. 2.11: Result is Generated and Goes into R0

The buffer register on the left input of the ALU was necessary in order to memorize the contents of R0, so that the single bus could be used again for another transfer. However, a problem remains.

### 2.3.8 The Critical Race Problem

The simple organization shown in [Figure 2.8](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.8) will not function correctly.

|  |  |
| --- | --- |
| Question: | What is the timing problem? |
|  | |
| Answer: | The problem is that the result which will be propagated out of the ALU will be deposited back on the single bus. It will not just propagate in the direction of R0, but along all of the bus. In particular, it will recondition the right input of the ALU, changing the result coming out of it a few nanoseconds later. This is a critical race. The output of the ALU must be isolated from its input (see [Figure 2.12](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.12)).  Several solutions are possible which will isolate the input of the ALU from the output. A buffer register must be used. The buffer register could be placed on the output of the ALU, or on its input. It is usually placed on the input of the ALU. Here it would be placed on its right input. The buffering of the system is now sufficient for a correct operation. It will be shown later in this chapter that if the left register which appears in this illustration is to be used as an accumulator (permitting the use of one-byte long instructions), then the accumulator will require a buffer too, as shown in [Figure 2.13](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.13).  [figure 2.12](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-12.gif)  Fig. 2.12: The Critical Race Problem  [figure 2.13](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-13.gif)  Fig. 2.13: Two Buffers Are Required (Temp Registers) |

## 2.4 INTERNAL ORGANIZATION OF THE Z80

The terms necessary in order to understand the internal elements of the microprocessor have been defined. We will now examine in more detail the Z80 itself, and describe its capabilities. The internal organization of the Z80 is shown in [Figure 2.14](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-14.gif). This diagram presents a logical description of the device. Additional interconnections may exist but are not shown. Let us examine the diagram from right to left.

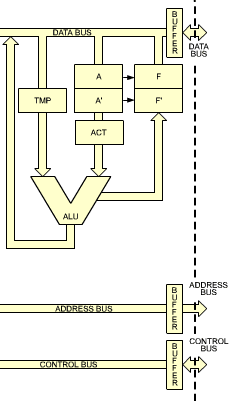
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-14c.gif)

Fig. 2.14, right side: Internal Z80 Organization (ALU and connection to the outside world)

On the right part of the illustration, the arithmetic-logical unit (the ALU) may be recognized by its characteristic "V" shape. The accumulator register, which has been described in the previous section, is identified as A on the right input path of the ALU. It has been shown in the previous section that the accumulator should be equipped with a buffer register. This is the register labeled ACT (temporary accumulator). Here, the left input of the ALU is also equipped with a temporary register, called TMP. The operation of the ALU will become clear in the next section, where we will describe the execution of actual instructions.

The flags register is called "F" in the Z80, and is shown on the right of the accumulator register. The contents of the flags register are essentially conditioned by the ALU, but it will be shown that some of its bits may also be conditioned by other modules or events.

The accumulator and the flags registers are shown as double registers labeled respectively A, A' and F, F'. This is because the Z80 is equipped internally with two sets of registers A + F, and A' + F'. However, only one set of these registers may be used at any one time. A special instruction is provided to exchange the contents of A and F with A' and F'. In order to simplify the explanations, only A and F will be shown on most of the diagrams which follow. The reader should remember that he has the option of switching to the alternate register set A' and F' if desired.

The role of each flag in the flags register will be described in Chapter 3 (Basic Programming Techniques).

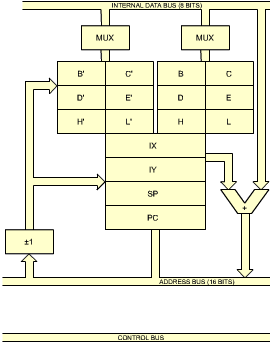
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-14b.gif)

Fig. 2.14, center: Internal Z80 Organization (Register Block)

A large block of registers is shown at the center of the illustration. On top of the block of registers, two identical groups can be recognized. Each one includes six registers labeled B, C, D, E, H, L. These are the general-purpose eight-bit registers of the Z80. There are two peculiarities of the Z80 with respect to standard microprocessors which has been described at the beginning of this chapter.

First, the Z80 is equipped with two banks of register, i.e., two identical groups of 6 registers. Only six may be used at any one time. However, special instructions are provided to switch between the two banks of registers. One bank, therefore, behaves as an internal memory, while the other one behaves as a working set of internal registers. The possible uses of the special facility will be described in the next chapter.

Conceptually, it will be assumed, for the time being, that there are only six working registers, B, C, D, E, H, and L, and the second register bank will temporarily be ignored, in order to avoid confusion.

The MUX symbol which appears above the memory bank is an abbreviation for multiplexer. The data coming from the internal data bus will be gated through the multiplexer to the selected register. However, only one of these registers can be connected to the internal data bus at any one time.

A second characteristic of these six registers, in addition to being general-purpose eight-bit registers, is that they are equipped with a connection to the address bus. This is why they have been grouped in pairs. For example, the contents of B and C can be gated simultaneously onto the 16-bits address bus which appears at the bottom of the illustration. As a result, this group of 6 registers may be used to store either eight-bit data or else 16-bit pointers for memory addressing.

The third group of registers, which appears below the two previous ones in the middle of [figure 2.14](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.14.b), contain four "pure" address registers. As in any microprocessor, we find the program counter (PC) and the stack pointer (SP). Recall that the program counter contains the address of the next instructionto be executed.

The stack pointer points to the top of the stack in the memory. In the case of the Z80, the stack pointer points to the last actual entry in the stack. (In other microprocessors, the stack pointer points just above the last entry.) Also, the stack grows "downwards" i.e. towards the lower addresses.

This means that the stack pointer must be decremented any time a new word is pushed on the stack. Conversely, whenever a word is removed (popped) from the stack, the stack pointer must be incremented by one. In the case of the Z80, the "push" and "pop" always involve two words at the same time, so that the contents of the stack pointer will be decremented or incremented by two.

Looking at the remaining two registers of this group of four registers, we find a new type of register which has not been described yet: two index registers, labeled IX (Index Register X) and IY (Index Register Y). These two registers are equipped with a special adder shown as a miniature V-shaped ALU on the right of these registers in [Figure 2.14](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.14.b). A byte brought along the internal data bus may be added to the contents of IX or IY. This byte is called the displacement, when using an indexed instruction. Special instructions are provided which will automatically add this displacement to the contents of IX or IY and generate an address. This is called indexing. It allows convenient access to any sequential block of data. This important facility will be described in Chapter 5 on addressing techniques.

Finally, a special box labeled "+/- 1" appears below and to the left of the block of registers. This is an increment/decrement. The contents of any of the register pairs SP, PC, BC, DE, HL (the "pure address" registers) may be automatically incremented or decremeneted every time they deposit an address on the internal address bus. This is an essential facility for implementing automated program loops which will be described in the next section. Using this feature it will be possible to access successive memory locations conveniently.

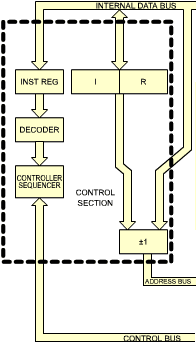
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-14a.gif)

Fig. 2.14, left side: Internal Z80 Organization (Control Section)

Let us move to the left side of the illustration. One register pair is shown, isolated on the left: I and R. The I register is called the interrupt page address register. Its rol will be described in the section on interrupts of Chapter 6 (Input/Output Techniques). It is used only in a special mode where an indirect call to a memory location is generated in response to an interrupt. The I register is used to store the high-order part of the indirect address. The lower part of the address is supplied by the device which generated the interrupt.

The R register is the memory-refresh register. It is provided to refresh dynamic memories automatically. Such a register has traditionally been located outside the microprocessor, since it is associated with the dynamic memory. It is a convenient feature which minimizes the amount of external hardware for some types of dynamic memories. It will not be used here for programming purposes, as it is essentially a hardware feature (see reference C207 "Microprocessor Interfacing Techniques" for a detailed description of memory refresh techniques). However, it is possible to use it as a software clock, for example.

Let us move now to the far left of the illustration. There the control section of the microprocessor is located. From top to bottom, we find first the instruction register IR, which will contain the instruction to be executed. The IR register is totally distinct from the "I, R" register pair described above. The instruction is received from the memory via the data bus, is transmitted along the internal data bus and is finally deposited into the instruction register. Below the instruction register appears the decoder which will send signals to the controller-sequencer and cause the execution of the instruction within the microprocessor and outside it. The control section generates and manages the control bus which appears at the bottom part of the illustration.

The three buses managed or generated by the system, i.e., the data bus, the address bus, and the control bus, propagate outside the microprocessor through its pins. The external connections are shown on the right-most part of the illustration. The buses are isolated from the outside through buffers shown in [Figure 2.14](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.14.c).

All the logical elements in the Z80 have now been described. It is not essential to understand the detailed operation of the Z80 in order to start writing programs. However, for the programmer who wishes to write efficient codes, the speed of a program and its size will depend upon the correct choice of registers as well as the correct choice of techniques. To make a correct choice, it is necessary to understand how instructions are executed within the microprocessor. We will therefore examine here the execution of typical instructions inside the Z80 to demonstrate the role and use of the internal registers and buses.

## 2.5 INSTRUCTION FORMATS

The Z80 instructioins are listed in Chapter 4. Z80 instructions may be formated in one, two, three or four bytes. An instruction specifies the operation to be performed by the microprocessor. From a simplified standpoint, every instruction may be represented as an opcode followed by an optional literal or address field, comprising one or two words. The opcode field specifies the operation to be carried out. In strict computer terminology, the opcode represents only those bits which specify the operation to be performed, exclusive of the register pointers which it might incorporate. In the microprocessor world, it is convenient to call opcode the operation code itself, as well as any register pointers which it might incorporate. This "generalized opcode" must reside in an eight-bit word for efficiency (this is the limiting factor on the number of instructions available in a microprocessor).

The 8080 uses instructions which may be one, two, or three bytes long (see [Figure 2.15](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.15)). However, the Z80 is equipped with additional indexed instructions, which require one more byte. In the case of the Z80, opcodes are, in general, one byte long, except for special instructions which require a two-byte opcode.

Some instructions require that one byte of data follow the opcode. In such a case, the instruction will be a two-byte instruction, the second byte of which is data (except for indexing, which adds an extra byte).

In other cases, the instruction might require the specification of an address. An address requires 16 bits and, therefore, two bytes. In that case, the instruction will be a three-byte or a four-byte instruction.

For each byte of the instruction, the control unit will have to perform a memory fetch, which will require four clock cycles. The shorter the instruction, the faster the execution.

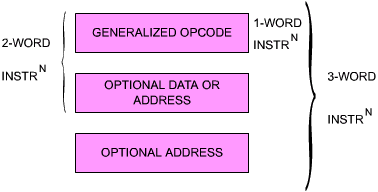
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-15.gif)

Fig. 2.15: Typical Instruction Formats

### 2.5.1 A One-Word Instruction

One-word instructions are, in principle, fastest and are favored by the programmer. A typical such instruction for the Z80 is:

LD r,r'

This instruction means: "Transfer the contents of register r' into r." This is a typical "register-to-register" operation. Every microprocessor must be equipped with such instructions, which allow the programmer to transfer information from any of the machine's registers into another one. Instructions referencing special registers of the machine, such as the accumulator or other special-purpose registers, may have a special opcode.

After execution of the above instruction, the contents of r will be equal to the contents of r'. The contents of r' will not have been modified by the read operation.

Every instruction must be represented internally in a binary format. The above representation "LD r,r' " is symbolic or mnemonic. It is called the assembly-language representation of an instruction. It is simply meant as a convenient symbolic representation of the actual binary encoding for that instruction. The binary code which will represent this instruction inside the memory is: 0 1 D D D S S S (bits 0 to 7).

This representation is still partially symbolic. Each of the letters S and D stands for a binary bit. The three D's, "D D D", represent the three bits pointing to the destination register. Three bits allow selection of one out of eight possible registers. The codes for these registers appear in [Figure 2.16](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.16). For example, the code for register B is "0 0 0", the code for register C is "0 0 1", and so on.

Similarly, "S S S" represents the three bits pointing to the source register. The convention here is that register r' is the source, and that register r is the destination. The placement of bits in the binary representation of an instruction is not meant for the convenience of the programmer, but for the convenience of the control section of the microprocessor, which must decode and execute the instruction. The assembly-language representation, however, is meant for the convenience of the programmer. It could be argued that LD r,r' should really mean: "Transfer contents of r into r'." However, the convention has been chosen in order to maintain compatibility with the binary representation in this case. It is naturally arbitrary.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exercise 2.1: | Write below the binary code which will transfer the contents of register C into register B. Consult [Figure 2.16](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.16) for the codes corresponding to C and B.   |  |  |  |  | | --- | --- | --- | --- | | **CODE** | **REGISTER** | **CODE** | **REGISTER** | | 0 0 0 | B | 1 0 0 | H | | 0 0 1 | C | 1 0 1 | L | | 0 1 0 | D | 1 1 0 | - (MEMORY) | | 0 1 1 | E | 1 1 1 | A |   Fig. 2.16: The Register Codes |

### 2.5.2 A Two-Word Instruction

ADD A, n

This simple two-word instruction will add the contents of the second byte of the instruction to the accumulator. The contents of the second word of the instruction are said to be a "literal." They are data and are treated as eight bits without any particular significance. They could happen to be a character or numerical data. This is irrelevant to the operation. The code for this instruction is:

1 1 0 0 0 1 1 0  followed by the 8-bit byte "n"

This is an immediate operation. "Immediate," in most programming languages, means that the next word, or words, within the instruction contains a piece of data which should not be interpreted (the way an opcode is). It means that the next one or two words are to be treated as a literal.

The control unit is programmed to "know" how many words each instruction has. It will, therefore, always fetch and execute the right number of words for each instruction. However, the longer the possible number of words for the instruction, the more complex it is for the control unit to decode.

### 2.5.3 A Three-Word Instruction

LD A, (nn)

The instruction requires three words. It means: "Load the accumulator from the memory address specified in the next two bytes of the instruction." Since addresses are 16-bits long, they require two words. In binary, this instruction is represented by:

|  |  |
| --- | --- |
| 0 0 1 1 1 0 1 0: | 8 bits for the opcode |
| Low address: | 8 bits for the lower part of the address |
| High address: | 8 bits for the upper part of the address |

## 2.6 EXECUTION OF INSTRUCTIONS WITHIN THE Z80

We have seen that all instructions are executed in three phases: FETCH, DECODE, EXECUTE. We now need to introduce some definitions. Each of these phases will require several clock cycles. The Z80 executes each phase in one or more logical cycles, called a "machine cycle." The shortest machine cycle lasts three clock cycles.

Accessing the memory requires three cycles for any operands, four clock cycles for the initial fetch. Since each instruction must be fetched first from memory, the fastest instruction will require four clock cycles. Most instruction will require more.

Each machine cycle is labeled M1, M2, etc., and will require three or more clock cycles, or "states," labeled T1, T2, etc.

### 2.6.1 The FETCH Phase

The FETCH phase of an instruction is implemented during the first three states of machine cycle M1; they are called T1, T2, and T3. These three states are common to all instructions of the microprocessor, as all instructions must be fetched prior to execution. The FETCH mechanism is the following:

T1 : PC OUT

The first step is to present the address of the next instruction to the memory. This address is contained in the program counter (PC). As the first step of any instruction fetch, the contents of PC are placed on the address bus (see [Figure 2.17](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.17)). At this point, an address is presented to the memory, and the memory address decoders will decode this address in order to select the appropriate location within the memory. Several hundred ns (a nanosecond is 10-9 second) will elapse before the contents of the selected memory location become available on the output pins of the memory, which are connected to the data bus. It is standard computer design to use the memory read time to perform an operation within the microprocessor. The operation is the incrementation of the program counter:

T2 : PC = PC + 1

While the memory is reading, the contents of the PC are incremented by 1 (see [Figure 2.18](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.18)). At the end of state T2, the contents of the memory are available and can be transferred within the microprocessor:

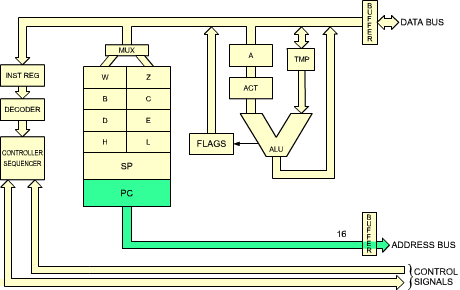
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-17.gif)

Fig. 2.17: Instruction Fetch - (PC) Is Sent to the Memory

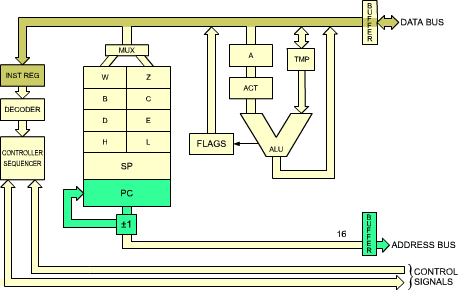
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-18.gif)

Fig. 2.18: PC Is Incremented

### 2.6.2 The DECODE and EXECUTE Phases

During state T3, the instruction which has been read out of the memory is deposited on the data bus and transferred into the instruction register of the Z80, from which point it is decoded.

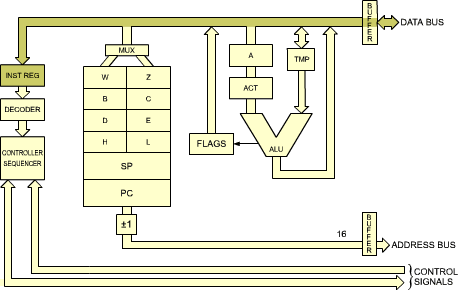
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-19.gif)

Fig. 2.19: The Instruction Arrives from the Memory into IR

It should be noted that states T4 of M1 will always be required. Once the instruction has been deposited into IR during T3, it is necessary to decode and execute it. This will require at least one machine state, T4.

A few instructions require an extra state of M1 (state T5). It will be skipped by the processor for most instructions. Whenever the execution of an instruction requires more than M1, i.e., M1, M2 or more cycles, the transition will be directly from state T4 of M1 into T1 of M2. Let us examine an example. The detailed internal sequencing for each example is shown in the tables of [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm). As these tables have not been released for the Z80, the 8080 tables are used instead. They provide an indepth understanding of the instruction execution.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 01 | |
| **MNEMONIC:** | | MOV r1,r2 | |
| **OP CODE:** | | D7D6D5D4 | D3D2D1D0 |
|  | | 0 1 D D | D S S S |
| **M1 [**[**1**](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_1)**]** | T1 | PC OUT STATUS | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 | |
| T3 | INST ->TMP/IR | |
| T4 | (SSS) ->TMP | |
| T5 | TMP ->(DDD) | |

LD D, C

This corresponds to MOV r1,r2 for the 8080. Refer to [item 1](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.27_01) of [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm).

By coincidence, the destination register in this example happens to be named "D". The transfer is illustrated in [Figure 2.20](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.20).

This instruction has been described in the previous section. It transfers the contents of register C, denoted by "C", into register D.

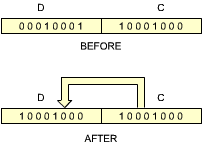
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-20.gif)

Fig. 2.20: Transferring C into D

The first three states of cycle M1 are used to fetch the instruction from the memory. At the end of T3, the instruction is in IR, the Instruction Register, from which point it can be decoded (see [Figure 2.19](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.19)).

During T4: (S S S) >TMP

The contents of C are deposited into TMP (see [Figure 2.21](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.21)).

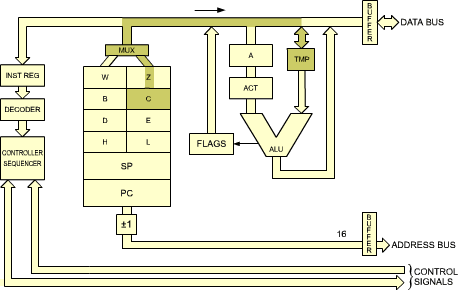
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-21.gif)

Fig. 2.21: The Contents of C Are Deposited into TMP

During T5: (TMP) >DDD

The contents of TMP are deposited into D. This is shown in [Figure 2.22](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.22).

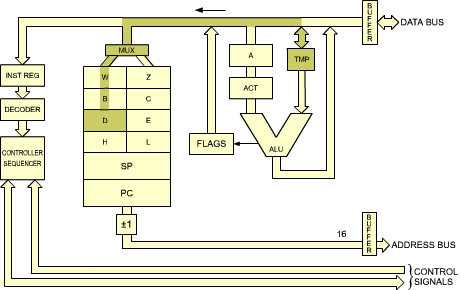
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-22.gif)

Fig. 2.22: The Contents of TMP Are Deposited into D

Execution of the instruction is now complete. The contents of register C have been transferred into the specified destination register D. This terminates execution of the instruction. The other machine cycles M2, M3, M4, and M5 will not be necessary and execution stops with M1.

It is possible to compute the duration of this instruction easily. The duration of every state for standard Z80 is the duration of the clock: 500 ns. The duration of this instruction is the duration of five states, or 5 x 500 = 2500 ns = 2.5 us. With a 400 ns clock, 5 x 400 = 2000 ns = 2.0 us.

|  |  |
| --- | --- |
| Question: | Why does the instruction require two states, T4 and T5, in order to transfer the contents C into D, rather than just one? It transfers the contents of C into TMP, and then the contents of TMP into D. Wouldn't it be simpler to transfer the contents of C into D directly within a single state? |
| Answer: | This is not possible because of the implementation chosen for the internal registers. All the internal registers are, in fact, part of a single RAM, a read/write memory internal to the microprocessor chip. Only one word may be addressed or selected at a time within an RAM (single-port). For this reason, it is not possible to both read and write into, or from, an RAM at two different locations. Two RAM cycles are required. It becomes necessary first to read the data out of the register RAM, and store it in a temporary register, TMP, then, to write it back into the destination register, here D. This is a design inadequacy. However, this limitation is common to virtually all monolithic microprocessors. A dual-port RAM would be required to solve the problem. This limitation is not intrinsic to microprocessors and it normally does not exist in the case of bit-slice devices. It is a result of the constant search for logic density on the chip and may be eliminated in the future. |

#### 2.6.2.1 Important Exercise:

At this point, it is highly recommended that the user review himself the sequencing of this simple instruction before we proceed to more complex ones. For this purpose, go back to [Figure 2.14](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-14.gif). Assemble a few small-sized "symbols" such as matches, paperclips, etc. Then move the symbols on [Figure 2.14](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-14.gif) to simulate the flow of data from the registers into the buses. For example, deposit a symbol into PC. T1 will move the symbol contained in PC out on the address bus towards the memory. Continue simulated execution in this fashion until you feel comfortable with the transfer along the buses and between the registers. At this point, you should be ready to proceed.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 15 | |
| **MNEMONIC:** | | ADD r | |
| **OP CODE:** | | D7D6D5D4 | D3D2D1D0 |
|  | | 1 0 0 0 | 0 S S S |
| **M1 [**[**1**](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_1)**]** | T1 | PC OUT STATUS | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 | |
| T3 | INST ->TMP/IR | |
| T4 | (SSS) ->TMP (A) ->ACT | |
| **M2** | T1 | [[9](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_9)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | (ACT) + (TMP) ->A | |

Progressively more complex instructions will now be studied:

ADD A, r

This instruction means: "Add the contents of register r (specified by a binary code S S S) to the accumulator (A), and deposit the result in the accumulator." This is an implicit instruction. It is called implicit as it does not explicitly reference a second register. The instruction explicitly refers only to register r. It implies that the other register involved in the operation is the accumulator. The accumulator, when used in such an implicit instruction, is referenced both as source and destination. The advantage of such an implicit instruction is that its complete opcode is only eight bits in length. It requires only a three-bit register field for the specification of r. This is a fast way to perform an addition operation.

Other implicit instructions exist in the system which will reference other specialized registers. More complex examples of such implicit instructions are, for example, the PUSH and POP operations, which will transfer information between the top of the stack and the accumulator, and will at the same time update the stack pointer (SP), decrementing it or incrementing it. They implicitly manupulate the SP register.

The execution of the ADD A,r instruction will now be examined in detail. This instruction will require two machine cycles, M1 and M2. As usual, during the first three states of M1, the instruction is fetched from the memory and deposited in the IR register. At the beginning of T4, it is decoded and can be executed. It will be assumed here that register B is added to the accumulator. The code for the instruction will then be 1 0 0 0 0 0 0 0 (the code for register B is 0 0 0). The 8080 equivalent is ADD r.

T4: (S S S) >TMP, (A) >ACT

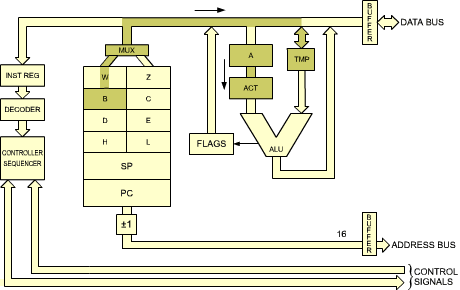
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-23.gif)

Fig. 2.23: Two Transfers Occur Simultaneously

Two transfers will be executed simultaneously. First, the contents of the specified register (here B) are transferred into TMP, i.e., to the right input of the ALU (see [Figure 2.23](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.23)). At the same time, the contents of the accumulator are transferred to the temporary accumulator (ACT). By inspecting [Figure 2.23](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.23), you will ascertain that those can occur in parallel. They use different paths within the system. The transfer from B to TMP uses the internal data bus. The transfer from A to ACT uses a short internal path independent of this data bus. In order to gain time, both transfers are done simultaneously. At this point, both the left and the right input of the ALU are correctly conditioned. The left input of the ALU is now conditioned by the accumulator contents, and the right input of the ALU is conditioned by the contents of register B. We are ready to perform the addition. We would normally expect to see the addition take place during state T5 of M1. However, this state is simply not used. The addition is not performed! We enter machine cycle M2. During state T1, nothing happens! It is only in state T2 of M2 that the addition takes place (refer to [ADD r](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.27_15) in [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm)):

T2 of M2: (ACT) + (TMP) >A

The contents of ACT are added to the contents of TMP, and the result is finally deposited in the accumulator. See [Figure 2.24](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.24). The operation is now complete.

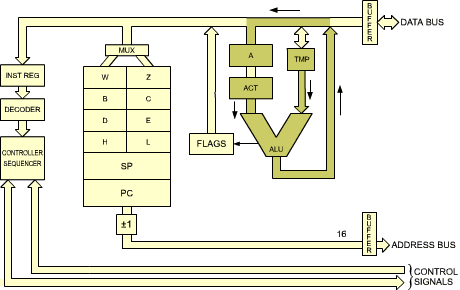
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-24.gif)

Fig. 2.24: End of ADD r

|  |  |
| --- | --- |
| Question: | Why was the completion of the addition deferred until state T2 of machine cycle M2, rather than taking place during state T5 of M1? (This is a difficult question, which requires an understanding of CPU design. However, the technique involved is fundamental to clock-synchronous CPU design. Try to see what happens.) |
| Answer: | This is a standard design "trick" used in most CPU's. It is called "fetch/execute overlap." The basic idea is the following: looking back at [Figure 2.23](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.23) it can be seen that the actual execution of the addition will only require the use of the ALU and of the data bus. In particular, it will not access the register RAM (register block). We (or the control unit) know that the next three states which will be executed after the completion of any instruction will be T1, T2, T3 of machine cycle M1 of the next instruction. Looking back at the execution of these three states, it can be seen that their execution will only require access to the program counter (PC) and the use of the address bus. Access to the program counter will require access to the register RAM. (This explains why the same trick would not be used in the instruction LD r,r'.) It is therefore possible to use simultaneously the shaded area in [Figure 2.17](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.17) and the shaded area in [Figure 2.24](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.24).  The data bus is used during state T1 of M1 to carry status information out. It cannot be used for the addition that we wish to perform. For that reason, it becomes necessary to wait until state T2 before the addition can be effectively carried out. This is what occurred in the chart: the addition is completed during state T2 of M2. The mechanism has now been explained. The advantage of this approach should now be clear. Let us assume that we had implemented a straightforward scheme, and performed the addition during state T5 of machine cycle M1.  [figure 2.25](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-25.gif)  Fig. 2.25: FETCH-EXECUTE Overlap during T1-T2  The duration of the ADD instruction would have been 5 x 500 ns = 2500 ns. With the overlap approach which has been implemented, once state T4 has been executed, the next instruction is initiated. In a manner that is invisible to this next instruction, the "clever" control unit will use state T2 to carry out the end of the addition. On the chart T2 is shown as part of M2. Conceptually, M2 will be the second machine cycle of the addition. In fact, this M2 will be overlapped, i.e., be identical to machine cycle M1 of the next instruction. For the programmer, the delay introduced by ADD will be only four states, i.e. 4 x 500 = 2000 ns, instead of 2500 ns using the "straightforward" approach. The speed improvement is 500 ns, or 20%!  The overlap technique is illustrated in [Figure 2.25](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.25). It is used whenever possible to increase the appearent execution speed of the microprocessor. Naturally, it is not possible to overlap in all cases. Required buses or facilities must be available without conflict. The control unit "knows" whether an overlap is possible. |

Courtesy: Intel Corporation

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NOTES:**   |  |  | | --- | --- | | 1. | The first memory cycle (M1) is always an instruction fetch; the first (or only) byte, containing the op code, is fetched during this cycle. | | 2. | If the READY input from memory is not high during T2 of each memory cycle, the processor will enter a wait state (TW) until READY is sampled as high. | | 3. | States T4 and T5 are present, as required, for operations which are completely internal to the CPU. The contents of the internal bus during T4 and T5 are available as the data bus; this is designed for testing purposes only. An "X" denotes that the state is present, but only used for such internal operations as instruction decoding. | | 4. | Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified. | | 5. | These states are skipped. | | 6. | Memory read sub-cycles; an instruction or data word will be read. | | 7. | Memory write sub-cycle. | | 8. | The READY signal is not required during the second and third sub-cycles (M2 and M3). The HOLD signal is accepted during M2 and M3. The SYNC signal is not generated during M2 and M3. During the execution of DAD, M2 and M3 are required for an internal register-pair add; memory is not referenced. | | 9. | The results of these arithmetic, logical or rotate instructions are not moved into the accumulator (A) until state T2 of the next instruction cycle. That is, A is loaded while the next instruction is being fetched; this overlapping of operations allows for faster processing. | | 10. | If the value of the least significant 4-bits of the accumulator is greater than 9, or if the auxiliary carry bit is set, 6 is added to the accumulator. If the value of the most significant 4-bits of the accumulator is now greater than 9, or if the carry bit is set, 6 is added to the most significant 4-bits of the accumulator. | | 11. | This represents the first sub-cycle (the instruction fetch) of the next instruction cycle. | | 12. | If the condition was met, the contents of register pair WZ are output on the address lines (A0-15) instead of the contents of the program counter (PC). | | 13. | If the condition was not met, sub-cycles M4 and M5 are skipped; the processor instead proceeds immediately to the instruction fetch (M1) of the next instruction cycle. | | 14. | If the condition was not met, sub-cycles M2 and M3 are skipped; the processor instead proceeds immediately to the instruction fetch (M1) of the next instruction cycle. | | 15. | Stack read sub-cycle. | | 16. | Stack write sub-cycle. | | 17. | CONDITION CCC  NZ - not zero (Z = 0) 000  Z - zero (Z = 1) 001  NC - no carry (CY = 0) 010  C - carry (CY = 1) 011  PO - parity odd (P = 0) 100  PE - paritty even (P = 1) 101  P - plus (S = 0) 110  M - minus (S = 1) 111 | | 18. | I/O sub-cycle: the I/O port's 8-bit select code is duplicated on address lines 0-7 (A0-7) and 8-15 (A8-15). | | 19. | Output sub-cycle. | | 20. | The processor will remain idle in the halt state until an interrupt, a reset or a hold is accepted. When a hold request is accepted, the CPU enters the hold mode; after the hold mode is terminated, the processor returns to the halt state. After a reset is accepted, the processor begins executing at memory location zero. After an interrupt is accepted, the processor executes the instruction forced onto the data bus (usually a restart instruction).   |  |  |  |  | | --- | --- | --- | --- | | **SSS or DDD** | **Value** | **rp** | **Value** | | A | 111 | B | 00 | | B | 000 | D | 01 | | C | 001 | H | 10 | | D | 010 | SP | 11 | | E | 011 |  |  | | H | 100 |  |  | | L | 101 |  |  | | |
| **NOTE:** The following abbreviations are used for operations: + addition - subtraction ^ logical AND v logical OR x logical XOR   logical NOT (underlined) |

Fig. 2.26: Intel Abbreviations

*[Fig. 2.27: Intel Instruction Formats](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm)*(stored in separate document)

|  |  |
| --- | --- |
| Question 2.1:  [solution](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80ans02.htm#ans_2.1) | Would it be possible to go further using this scheme, and to also use state T3 of M3 if we have to execute a longer instruction? |

In order to clarify the internal sequencing mechanism, it is suggested that you examine [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm), which shows the detailed instruction execution for the 8080. The Z80 includes all 8080 instructions, and more. The information represented in [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm) is not available for the Z80. It is shown here for its educational value in understanding the internal operation of this microprocessor. The equivalence between Z80 and 8080 instructions is shown in Appendices F and G.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 16 | |
| **MNEMONIC:** | | ADD M | |
| **OP CODE:** | | D7D6D5D4 | D3D2D1D0 |
|  | | 1 0 0 0 | 0 1 1 0 |
| **M1 [**[**1**](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_1)**]** | T1 | PC OUT STATUS | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 | |
| T3 | INST ->TMP/IR | |
| T4 | (A) ->ACT | |
| **M2** | T1 | HL OUT STATUS [[6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_6)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | DATA ->TMP | |
| T3 |
| **M3** | T1 | [[9](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_9)] | |
| T2 | (ACT) + (TMP) ->A | |

A more complex instruction will now be examined:

ADD A, (HL)

The opcode for this instruction is 10000110. This instruction means "add to the accumulator the contents of memory location (HL)." The memory location is specified through a rather strange system. It is the memory location whose address is contained in registers H and L. This instruction assumes that these two special registers (HL) have been loaded with contents prior to executing the instruction. The 16-bit contents of these registers will now specify the address in the memory where data resides. This data will be added to the accumulator, and the result will be left in the accumulator.

This instruction has a history. It has been supplied in order to provide compatibility between the early 8008, and its successor, the 8080. The early 8008 was not equipped with a direct-memory addressing capability! The procedure used to access the contents of the memory was to load the two registers H and L, and then execute an instruction referencing H and L. ADD A, (HL) is just such an instruction. It must be stressed that the 8080 and the Z80 are not limited in the same way as the 8008 in memory-addressing capability. They do have direct-memory addressing. The facility for using the H and L registers becomes an added advantage, not a drawback, as was the case with the 8008.

Let us now follow the execution of this instruction (it is called ADD M for the 8080 and is the [16th instruction](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.27_16) on [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm)). States T1, T2, and T3 of M1 will be used, as usual, to fetch the instruction. During state T4, the contents of the accumulator are transferred to its buffer register, ACT, and the left input of the ALU is conditioned.

Memory must be accessed in order to provide the second byte of data which will be added to the accumulator. The address of this byte of data is contained in H and L. The contents of H and L will therefore have to be transferred onto the address bus, where they will be gated to the memory. Let us do it.

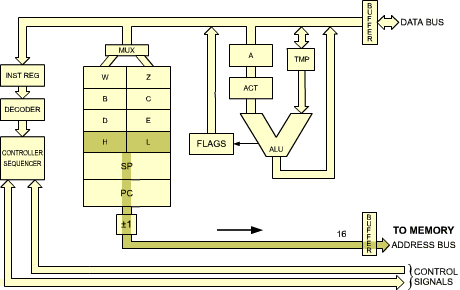
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-28.gif)

Fig. 2.28: Transfer Contents of HL to Address Bus

During machine cycle M2, we read: HL OUT. H and L are deposited on the address bus, in the same way PC used to be deposited there in previous instructions. As a remark, it has already been indicated that during state T1 status is output on the data bus, but no use of this will be made here. From a simplier standpoint, it will require two states: one for the memory to read its data, and one for the data to become available and transferred onto the right input of the ALU, TMP.

Both inputs of the ALU are now conditioned. The situation is analogous to the one we were in with the previous instruction ADD A,r: both inputs of the ALU are conditioned. We simply have to ADD as before. A fetch/execute overlap technique will be used, and, instead of executing the addition within state T4 of M2, final execution is postponed until state T2 of M3. It can be seen in [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm) that during T2 we indeed have: ACT + TMP ->A. The addition is finally performed, the contents of ACT are added to TMP, and the result deposited into the accumulator A.

|  |  |
| --- | --- |
| Question 2.2:  [solution](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80ans02.htm#ans_2.2) | What is the apparent execution time (to the programmer) for this instruction? Using a 2.5 Mhz clock, it is 3.6 us? 2.8 us? |

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 08 | |
| **MNEMONIC:** | | LDA addr | |
| **OP CODE:** | | D7D6D5D4 | D3D2D1D0 |
|  | | 0 0 1 1 | 1 0 1 0 |
| **M1 [**[**1**](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_1)**]** | T1 | PC OUT STATUS | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 | |
| T3 | INST ->TMP/IR | |
| T4 | X | |
| **M2** | T1 | PC OUT STATUS [[6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_6)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 B2 ->Z | |
| T3 |
| **M3** | T1 | PC OUT STATUS [[6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_6)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 B3 ->W | |
| T3 |
| **M4** | T1 | WZ OUT STATUS [[6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_6)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | DATA ->A | |
| T3 |

Another more complex instruction will now be examined which is a direct-memory addressing instruction using two invisible W and Z registers:

LD A,(nn)

The opcode is 00111010. The 8080 equivalent is LDA addr. As usual, states T1, T2, T3 of M1 will be used to fetch the instruction from the memory. T4 is used, but no visible result can be described. During state T4, the instruction is in fact decoded. The control unit then finds out that it has to fetch the next two bytes of this instruction in order to obtain the address from which the accumulator will be loaded. The effect of this instruction is to load the accumulator from the memory contents whose address is specified in bytes 2 and 3 of the instruction. Note that state T4 is necessary to decode the instruction. It could be considered a waste of time since only part of the state is necessary to do the decoding. It is. However, this is the philosophy of clock-synchronous logic. Because microinstructions are used internally to perform the decoding and execution, this is the penalty that has to be paid in return for the advantages of microprogramming. The structure of this instruction appears in [Figure 2.29](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.29).

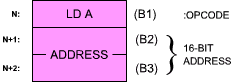
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-29.gif)

Fig. 2.29: LD A, (ADDRESS) Is a 3-Word Instruction

The next two bytes of instruction will now be fetched. They will specify an address (see [Figure 2.30](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.30)).

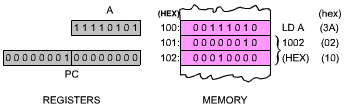
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-30.gif)

Fig. 2.30: Before Execution of LD A

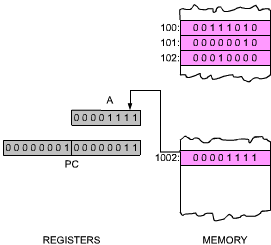
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-31.gif)

Fig. 2.31: After Execution of LD A

The effect of the instruction is shown in [Figures 2.30](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.30) and [2.31](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.31) above.

Two special registers are available to the control unit within the Z80 (but not to the programmer). The are "W" and "Z", and are shown in [Figure 2.28](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.28).

|  |  |
| --- | --- |
| Second Machine Cycle M2: | As usual, the first 2 states, T1 and T2 are used to fetch the contents of memory location PC. During T2, the program counter, PC, is incremented. Sometime by the end of T2, data becomes available from the memory, and appears on the data bus. By the end of T3, the word has been fetched from memory address PC (B2, second byte of the instruction) is available on the data bus. It must now be stored in a temporary register. It is deposited into Z: B2 ->Z (see [Figure 2.32](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.32)).  [figure 2.32](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-32.gif)  Fig. 2.32: Second Byte of Instruction Goes into Z |
| Machine Cycle M3: | Again, PC is deposited on the address bus, incremented, and finally the third byte, B3, is read from the memory and deposited into register W of the microprocessor. At this point, i.e., by the end of state T3 of M3, registers W and Z inside the microprocessor contain B2 and B3, i.e., the complete 16-bit address which was originally contained in the two words following the instruction in the memory. Execution can now be completed. W and Z contain an address. This address will have to be sent to the memory, in order to extract the data. This is done in the next memory cycle: |
| Machine Cycle M4: | This time, W and Z are output on the address bus. The 16-bit address is sent to the memory, and by the end of state T2, data corresponding to the contents of the specified memory location becomes available. It is finally deposited in A at the end of state T3. This terminates execution of this instruction. |

This illustrates the use of an immediate instruction. This instruction required three bytes in order to store a two-byte explicit address. This instruction also required four memory cycles, as it needed to go to the memory three times in order to extract the three bytes of the three-word instruction, plus one memory access in order to fetch the data specified by the address. It is a long instruction. However, it is also a basic one for loading the accumulator with specified contents residing at a know memory location. It can be noted that this instruction requires the use of W and Z registers.

|  |  |
| --- | --- |
| Question: | Could this instruction have used other registers than W, Z within the system? |
| Answer: | No. If this instruction had used other registers, for example the H and L registers, it would have modified their contents. After execution of this instruction, the contents of H and L would have been lost. It is always assumed in a program that an instruction will not modify any registers other than those it is explicitly using. An instruction loading the accumulator should not destroy the contents of any other register. For this reason, it becomes necessary to supply the extra two registers, W an Z, for the internal use of the control unit. |

|  |  |
| --- | --- |
| Question: | Would it be possible to use PC instead of W and Z? |
| Answer: | Possitively not. This would be suicidal. The reader should analyze this. |

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 54 | |
| **MNEMONIC:** | | JMP addr | |
| **OP CODE:** | | D7D6D5D4 | D3D2D1D0 |
|  | | 1 1 0 0 | 0 0 1 1 |
| **M1 [**[**1**](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_1)**]** | T1 | PC OUT STATUS | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 | |
| T3 | INST ->TMP/IR | |
| T4 | X | |
| **M2** | T1 | PC OUT STATUS [[6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_6)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 B2 ->Z | |
| T3 |
| **M3** | T1 | PC OUT STATUS [[6](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_6)] | |
| T2 [[2](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_2)] | PC = PC + 1 B3 ->W | |
| T3 |
|  | | WZ OUT STATUS [[11](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#note_11)] | |
| (WZ) + 1 ->PC | |

One more type of instruction will be studied now: a branch or jump instruction, which modifies the sequence in which instructions are executed within the program. So far, we have assumed that instructions were executed sequentially. Instructions exist which allow the programmer to jump out of sequence to another instruction within the program, or in practical terms, to jump to another area of the memory containing the program, or to another address. One such instruction is:

JP nn

This instruction appears as [item 54](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.27_54) of [Figure 2.27](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig2-27.htm) as "JMP addr." Its execution will be described by following the appropriate column of the Table. This is again a three-word instruction. The first word is the opcode, and contains 11000011. The next two words contain the 16-bit address, to which the jump will be made. Conceptually, the effect of this instruction is to replace the contents of the program counter with the 16 bits following the "JUMP" opcode. In practice, a somewhat different approach will be implemented, for reasons of efficiency.

As before, the first three states of M1 correspond to the instruction fetch. During state T4 the instruction is decoded and no other event is recorded (X). The next two machine cycles are used to fetch bytes B2 and B3 of the instruction. During M2, B2 is fetched and deposited into internal register Z, and during M3, B3 is fetched and deposited into internal register W. The next two steps will be implemented by the the processor during the next instruction-fetch, as was the case already with the addition. They will be executed instead of the usual steps for T1 and T2 of the next instruction. Let us look at them.

The next two steps will be: WZ OUT and (WZ)+1 ->PC. In other words, the contents of WZ will be used instead of the contents of PC during the next instruction-fetch. The control unit will have recorded the fact that a jump was being executed and will execute the beginning of the next instruction differently.

The effect of these extra states is the following:

The address placed on the address bus of the system will be the address contained in W and Z. In other words, the next instruction will be fetched from the address that was contained in W and Z. This is effectively a jump. In addition, the contents of WZ will be incremented by 1 and deposited in the program counter, so that the next instruction will be fetched correctly by using PC as usual. The effect is therefore correct.

|  |  |
| --- | --- |
| Question: | Why have we not loaded the contents of PC directly? Why use the intermediate W and Z register? |
| Answer: | It is not possible to use PC. If we had loaded the lower part of PC (PCL) with B2, instead of using Z, we would have destroyed PC. It would then have become impossible to fetch B3. |

|  |  |
| --- | --- |
| Question: | Would it be possible to use just Z, instead of W and Z? |
| Answer: | Yes, but it would be slower. We could have loaded Z with B2, then fetched B3, and deposited it into the high order half of PC (PCH). However, it would then have become necessary to transfer Z into PCL, before using the contents of PC. This would slow down the process. For this reason, W and Z are not transferred into PC. They are directly gated to the address bus in order to fetch the next instruction. |

|  |  |
| --- | --- |
| Question 2.3:  [solution](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80ans02.htm#ans_2.3) | (For the alert and informed reader only). What happens in the case of an interrupt at the end of M3? (If instruction execution is suspended at this point, the program counter points to the instruction following the jump, and the jump address, contained in W and Z, will be lost.) |

The detailed descriptions we have presented for the execution of typical instructions should clarify the role of the registers and of the internal buses. A second reading of the preceding section may help in gaining a detailed understanding of the internal operation of the Z80.

### 2.6.3 The Z80 Chip

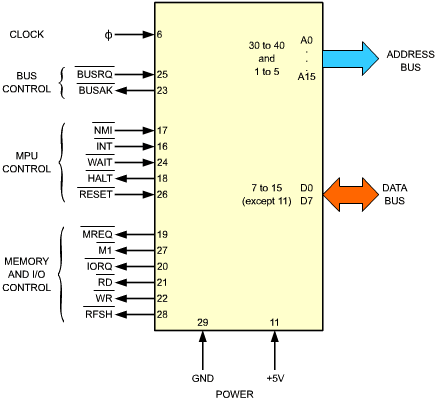
[](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/fig_2-33.gif)

Fig. 2.33: Z80 MPU Pinout

For completeness, the signals of the Z80 microprocessor chip will be examined here. It is not indispensable to understand the functions of the Z80 in order to be able to program it. The reader who is not interested in the details of the hardware may therefore skip this section.

The pinout of the Z80 appears on [Figure 2.33](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.33). On the right side of the illustration, the address bus and the data bus perform their usual role, as described in the beginning of this chapter. We will describe here the function of the signals on the control bus. They are shown on the left of [Figure 2.33](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.33).

The control signals have been partitioned in four groups. They will be described, going from the top of [Figure 2.33](http://www.msxarchive.nl/pub/msx/mirrors/msx2.com/zaks/z80prg02.htm#fig_2.33) towards the bottom.

The clock input is PHI. The Z80 requires an external 330-ohm pull-up resistor. It is connected to the PHI input and to 5 volts. However, at 4 Mhz, and external clock driver is required.

The two bus-control signals, BUSRQ and BUSAK, are used to disconnect the Z80 from its buses. They are mainly used by the DMA, but could also be used by another processor in the system. BUSRQ is the bus-request signal. It is issued to the Z80. In response, the Z80 will place its address bus, data bus and tristate output control signals in the high-impendance state, at the end of the current machine cycle. BUSAK is the acknowledge signal issued by the Z80 once the buses have been placed in the high-impendance state.

### 2.6.4 Internal Status and Sequencing

Six Z80 control signals are related to its internal status or to its sequencing:

* INT and NMI are the two interrupt signals. INT is the usual interrupt request. Interrupts will be described in Chapter 6. A number of input/output devices may be connected to the INT interrupt line. Whenever an interrupt request is present on this line, and when the internal interrupt enable flip-flop (IFF) is enabled, the Z80 will accept the interrupt (provided the BUSRQ is not active). It will then generate an acknowledge signal: IORQ (issued during the M1 state). The rest of the sequence of events is described in Chapter 6.
* NMI is the non-maskable interrupt. It is always accepted by the Z80 and it forces the Z80 to jump to location 0066 hexadecimal. It too is described in Chapter 6. (It also assumes that BUSRQ is not active.)
* WAIT is a signal used to synchronize the Z80 with slow memory or input/output devices. When active, this signal indicates that the memory or the device is not yet ready for the data transfer. The Z80 CPU will then enter a special wait state until the WAIT signal becomes inactive. It will then resume normal sequencing.
* HALT is the acknowledge signal supplied by the Z80 after it has executed the HALT instruction. In this state, the Z80 waits for an external interrupt and keeps executing NOPs to continually refresh memory.
* RESET is the signal which usually initializes the MPU. It sets the program counter, register I and R to "0". It disables the interrupt enable flip-flop and sets the interrupt mode to "0". It is normally used after power is applied to the board.

### 2.6.4 Memory and I/O Control

Six memory and I/O signals are generated by the Z80. They are: