in the name of God
the compassionate, the merciful

notes on
MACHINE
ARCHITECTURE
&
LANGUAGE

compiled by
Jumong
A system designer should consider a microprocessor-based solution whenever an application involves making calculations, making decisions based on external stimulus, and maintaining memory of past events.

First of all, the microprocessor itself is a general-purpose device. It takes personality by the software program instructions given by the designer.

Typically, when a new microprocessor is introduced, it will have a few new software instructions available and will have some of the I/O features, previously handled by external support chips, integrated into the microprocessor chip.

They are called 8-bit microprocessors because external and internal data movement is performed on 8 bits at a time.

There are three external buses:

**Address Bus** The address bus is 16 bits wide and is generated by the microprocessor to select a particular location or IC to be active.

**Data Bus** Once the address bus is set up with the particular address that the microprocessor wants to access, the microprocessor then sends or receives 8 bits of data to or from that address via bidirectional data bus.

**Control Bus** The control bus is of varying width, depending on the microprocessor being used. It carries control signals that are tapped into by the other ICs to tell what type of operation is being performed.

The microprocessor has an 8-bit internal register called the *accumulator* that can be used for I/O.

The software used to drive microprocessor-based systems is called *assembly language*. The Intel 8080/8085 assembly language statement to load the contents of the input port into the accumulator is `LDA addr`. LDA is called a *mnemonic*, an abbreviation of the operation being performed, which in this case is “Load Accumulator”. The suffix `addr` will be replaced with a 16-bit address (4 hex digits) specifying the address of the input port.

Now, to write those data to the output port, we use the command `STA addr`. STA is the mnemonic for “Store Accumulator” and `addr` is the 16-bit address where you want the data stored.

The microprocessor takes care of the timing on the three buses. The address decoder takes care of providing chip enables to the appropriate ICs.

The 8085A is *software compatible* with the 8080A. The 8085A has few additional features not available on the 8080A.

The 8085A is an 8-bit *central processing unit* (CPU). The accumulator discussed in the previous section is connected to an 8-bit *internal data bus*. Six other *general-purpose registers* labeled B, C, D, E, H and L are also connected to the same bus.

All arithmetic operations take place in the *arithmetic logic unit* (ALU). The accumulator, along with a temporary register, are used as inputs to all arithmetic operations. The output of the operations is sent to the internal data bus and to five *flag flip-flops* that record the status of the arithmetic operation.
The instruction register and decoder receive the software instructions from external memory, interpret what is to be done, and then create the necessary timing and control signals required to execute the instruction.

The block diagram also shows interrupt control, which provides a way for an external digital signal to interrupt a software program while it is executing. Serial communication capabilities are provided via the SID and SOD I/O pins.

The register array contains the six general-purpose 8-bit registers and three 16-bit registers. Sixteen-bit registers are required whenever you need to store addresses. The stack pointer stores the last entry on stack. The stack is a data storage area in RAM used by certain microprocessor operations, which will be covered in a later chapter. The program counter contains the 16-bit address of the next software instruction to be executed. The third 16-bit register is the address latch, which contains the current 16-bit address that is being sent to the address bus.

The six general-purpose 8-bit can also be used in pairs (B–C, D–E, H–L) to store addresses or 16-bit data.

The mnemonics LDA and STA cannot be understood by the CPU as they are, they have to be assembled, or converted, into a binary string called machine code.

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Assembly language</th>
<th>Machine code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000H</td>
<td>LDA 4000H</td>
<td>3A</td>
<td>Three-byte instruction to load</td>
</tr>
<tr>
<td>2001H</td>
<td>00</td>
<td>00</td>
<td>accumulator with contents from</td>
</tr>
<tr>
<td>2002H</td>
<td>40</td>
<td>40</td>
<td>address 4000H</td>
</tr>
<tr>
<td>2003H</td>
<td>STA 4000H</td>
<td>32</td>
<td>Three-byte instruction to store</td>
</tr>
<tr>
<td>2004H</td>
<td>00</td>
<td>00</td>
<td>accumulator out to address 6000H</td>
</tr>
<tr>
<td>2005H</td>
<td>60</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
The instructions are first written in assembly language using mnemonic abbreviations and the converted to machine language so that they can be interpreted by the microprocessor. → “assembler” “hand assembly”

Assembly language is classified as a low-level language. High-level languages are much easier to write but are not as memory efficient or as fast as assembly language. The conversion from high-level languages to machine code is done by a compiler. → most streamlined, memory-efficient, and fastest programs possible

Assembly language translates directly into machine code without using a compiler. → Assembly language and its corresponding machine code differs from processor to processor.

“standard I/O” or “I/O-mapped I/O” → identify their input and output devices by giving them an 8-bit \textit{port number}. The microprocessor then accesses the I/O ports by using instructions: OUT \textit{port} and IN \textit{port}, where \textit{port} is 00H to FFH.

Special hardware external to the 8085A is required to provide the source for the IN instruction and the destination for the OUT instruction. → SSI and MSI ICs

\textbf{74LS244}. octal three-state buffer

\textbf{74LS374}. octal D\textit{-}flip-flop

Setup time is accounted for by the microprocessor timing specifications.

<table>
<thead>
<tr>
<th>8085A Assembly language</th>
<th>8085A Machine language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Label</strong></td>
<td><strong>Instruction</strong></td>
</tr>
<tr>
<td>START:</td>
<td>MVI A,09H</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP:</td>
<td>DCR A</td>
</tr>
<tr>
<td></td>
<td>JZ START</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JMP LOOP</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembly language is written using \textit{mnemonics}. The term \textit{mnemonics} is defined as “abbreviations used to assist the memory”.

\textbf{MVI}: “Move Immediate”

\textbf{JZ} → \textit{conditional jump}, the condition that it is checking for is zero condition (zero flag is set (equal to 1)).

\textbf{JMP} → \textit{unconditional jump}

Register A and the accumulator are the same.

Be careful to always enter addresses as low-order first, then high-order.

\textbf{CPI data}

Compare Immediate

sets zero flag if accumulator is equal to data, sets carry flag if accumulator is \textit{less than} byte 2 (data)

\textbf{CMA}

Complement Accumulator
MVI r, data
The easiest way to store data into a register r is replaced with one of the registers A, B, C, D, E, H, or L. data is replaced with a 1-byte data value.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVI A, data</td>
<td>3E</td>
</tr>
<tr>
<td>MVI B, data</td>
<td>06</td>
</tr>
<tr>
<td>MVI C, data</td>
<td>0E</td>
</tr>
<tr>
<td>MVI D, data</td>
<td>16</td>
</tr>
<tr>
<td>MVI E, data</td>
<td>1E</td>
</tr>
<tr>
<td>MVI H, data</td>
<td>26</td>
</tr>
<tr>
<td>MVI L, data</td>
<td>2E</td>
</tr>
</tbody>
</table>

MOV r1, r2
To move data from register to register. Register r2 is the source and r1 is the destination of the data. The data in the source register (r2) remains unchanged.

1-byte instruction

MOV C, 44H
MOV E, C

2 bytes long

MOV C, 44H
MVI E, 44H

3 bytes long

LXI rp, data16
allows to load all 16 bits of a register pair with one instruction. The rp is replaced by a single letter signifying which register pair to be loaded.

B for register pair B–C, D for register pair D–E, H for register pair H–L

3-byte instruction:
byte 1 : opcode
byte 2 : low-order byte of data → low-order register in rp
byte 3 : high-order byte of data → high-order register in rp

INR r / DCR r / INX rp / DCX rp
commands for incrementing and decrementing registers and register pairs. If you increment/decrement a single register that has an FFH/00H in it, it will “roll over” to 00H/FFH. The same happens to register pairs when they are incremented past FFFFH or decremented below 0000H.

All register pair instructions have an X in their mnemonic.

One T state is the length of one microprocessor clock period. The internal clock frequency of the microprocessor is one-half the frequency of the crystal used to drive the 8085A.

in conditional jumps
number of T states if the condition is met > number of T states if the condition is not met

loop ⇒ delay
nested loops ⇒ longer delays
inserting dummy instructions within the loop or forming a third level of nesting ⇒ even longer delays

5
NOP
no-operation
a commonly used dummy or “do nothing” instruction
takes four \( T \) states

Register pairs can also be used as our loop counter. The problem with register pairs, however, is that the DCX \( rp \) instruction does not set the zero flag when the register pair reaches 0000H.

\[ \text{DCR} \ r \text{ affects the zero flag, whereas DCX} \ rp \text{ doesn’t.} \]

There are ways to use register pairs as loop counters, however. The CMP \( r \) or the ORA \( r \) instructions, could be used for that purpose.

CALL addr
We can set up the delay routine as a subroutine and “call” it whenever we need a delay. All we need to do is to put a return statement (RET) at the end of the routine. Then, whenever another program calls it by using a CALL addr instruction, it will be executed. When execution of the subroutine is complete, the program control returns to the instruction following the CALL addr instruction. The operand, \( addr \) is the address in the memory of the subroutine, which we want to call it.

HLT
halts microprocessor execution

A single microprocessor program instruction occupies 1, 2, or 3 bytes of memory.
It always starts with a 1-byte opcode.