2012

The British University in Egypt - BUE
Faculty of Informatics and Computer Science
Professor Dr. Samir Abou El-Seoud

ADDRESSING MODES

To perform any operation, we have to give the corresponding instructions to the microprocessor. In each instruction, programmer has to specify 3 things: Operation to be performed, Address of source of data, Address of destination result. The method by which the address of source of data or the address of destination of result is given in the instruction is called Addressing Modes.

Introduction

Address:

Generally speking, an address is a location of data, usually in main memory or on a disk. You can think of computer memory as an array of storage boxes, each of which is one byte in length. Each box has an address (a unique number) assigned to it. By specifying a memory address, programmers can access a particular byte of data. Disks are divided into *tracks* and *sectors*, each of which has a unique address. Usually, you do not need to worry about addresses unless you are a programmer.

- With computer data storage: An **address** is the location pointing to where data can be accessed.
- With computer networks: An **address** refers to a network **IP address** or other unique network identification. On the Internet: An address is a synonym for a **web address**.

Absolute Address:

Alternatively known as a **direct address**, machine address or real address, an absolute address is an exact memory address. An absolute address is a fixed address in memory. The term *absolute* distinguishes it from a *relative address*, which indicates a location by specifying a distance from another location. Absolute addresses are also called *real addresses* and *machine addresses*.

Relative Address:

An address specified by indicating its distance from another address, called the *base address*. For example, a relative address might be B+15, B being the base address and 15 the distance (called the *offset*).

There are two types of addressing: *relative addressing* and *absolute* addressing. In absolute addressing, you specify the actual address (called the *absolute address*) of a memory location.

Relative and absolute addressing are used in a variety of circumstances. In programming, you can use either mode to identify locations in main memory or on mass storage devices. In spreadsheet applications, you can use either mode to designate a particular cell.

Effective Address

An effective address is the value which is used by a fetch or store operation to specify which memory location is to be accessed by the operation from the perspective of the entity (i.e. process, thread, interrupt handler, kernel component, etc) issuing the operation. It is the address that is obtained by applying any specified indexing or indirect addressing rules to the specified address; the effective address is then used to identify the current operand.

For example:

- if a register contains the value 10000 and an instruction is executed which fetches the contents of the memory location specified by the contents of the register then the effective address for the fetch operation is 10000.
- if a register contains the value 10000 and an instruction is executed which stores a value into the memory location which is 120 memory locations beyond the location specified by the contents of the register then the effective address for the store operation is 10120.

Addressing Mode: An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. When a microprocessor accesses memory, to either read or write data, it must specify the memory address it needs to access.

Addressing Modes

On **ARM** processors you have 16 registers. Actually, that is not entirely true. ARM processors have 32 registers, each 32 bits wide. ARM processors have different programming modes to distinguish user-level and system-level access. Only some registers are visible in each mode. In the user-level mode you can access 16 registers. This is the mode you will use most frequently, so you can ignore the entire mode stuff for now. The 16 user-accessible general-purpose registers are labeled R0 – R15. The last four registers are special:

R12: IP, or ***Intra-Procedure*** call stack register. This register is used by the linker as a scratch register between procedure calls. A procedure must not modify its value on return.

R13: SP, or ***Stack Pointer***. This register points to the top of the stack. The stack is area of memory used for local function-specific storage. This storage is reclaimed when the function returns. To allocate space on the stack, we subtract from the stack register. To allocate one 32-bit value, we subtract 4 from the stack pointer.

R14: LR, or ***Link Register***. This register holds the return value of a subroutine. When a subroutine is called, the LR is filled with the program counter.

R15: PC, or ***Program Counter***. This register holds the address of memory that is currently being executed.

There is one more register, the **Current Program Status Register (CPSR)** that contains values indicating some flags like Negative, Zero, Carry, etc. you can't read and write it like a normal register anyway.

There is a method in which the instructions address the data to be operated. The method of specifying the data to be operated by the instruction is known as **addressing**.

Now we know what is addressing. But what are addressing modes?

The way by which the microprocessor identifies the operands for a particular instruction is known as **Addressing mode**. An "addressing mode" refers to how you are addressing a given memory location.

To perform any operation, we have to give the corresponding instructions to the microprocessor. In each instruction, programmer has to specify 3 things: Operation to be performed, Address of source of data, Address of destination result. The method by which the address of source of data or the address of destination of result is given in the instruction is called Addressing Modes.

The term addressing mode refers to the way in which the operand of instruction is specified. Intel 8085 uses the following addressing modes:

1. Immediate Addressing Mode
2. Direct Addressing Mode
3. Register Indirect Addressing Mode
4. Register Addressing Mode
5. Implicit Addressing Mode
6. Indexed Addressing Mode

1. Immediate Addressing Mode:

- One of the simplest modes is immediate addressing, where the operand itself is accessed. Immediate addressing is so-named because the value of constant to be stored in memory immediately follows the operation code in memory. That is to say, the instruction itself dictates what value will be stored in memory. In this mode, the operand is specified within the instruction itself. In other words, the operand specified is not address; it is the actual data to be used.

LDR R1, #1999 **R1 $\leftarrow$ 1999**

a. **LDR is the operation**
b. **1999 is the immediate data (source)**
c. **R1 is the destination**

In this example, the data value 1999 is loaded into R1.

Here is another example:

MOV A, #20H **A $\leftarrow$ 20**

This instruction uses Immediate Addressing because the Accumulator will be loaded with the value that immediately follows; in this case 20 (hexadecimal).

Here is another example:

ADD R1, #5 **R1 $\leftarrow$ R1 + 5**

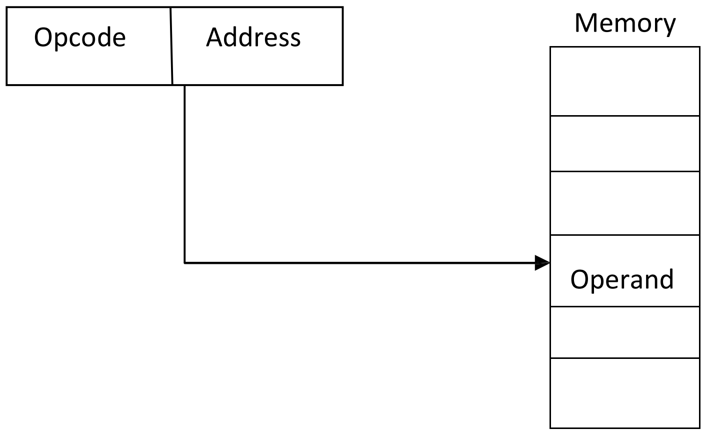
a. **ADD is the operation**
b. **R1+5 is the immediate data (source)**
c. **R1 is the destination**

Immediate addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible.

- This mode is a good way to specify initial values for registers.
- We've already used immediate addressing several times.
  – It appears in the string conversion program you just saw.

2. Direct Addressing Mode

- Another possible mode is direct addressing, where the operand is a constant that represents a memory address. Direct addressing is so-named because the value to be stored in memory is obtained by directly retrieving it from another memory location. In other words, the instruction in this mode includes a memory address; the CPU accesses that location in memory.



LDR R1, 500 **R1 $\leftarrow$ M[500]**

a. **LDR is the operation**
b. **500 is the effective address of source**
c. **R1 is the destination**

- Here the effective address is 500, the same as the operand.
- This is useful for working with pointers.
  – You can think of the constant as a pointer.
  – The register gets loaded with the data at that address.

In the above example, the instruction LDR R1, 500 reads the data from memory location 500 and stores the data in the register R1. This mode is typically used to load operands and values of variables into the CPU.

Here is another example:

MOV A, 30H

This instruction will read the data out of Internal RAM address 30 (hexidecimal) and store it in the Accumulator.

Direct addressing is generally fast since, although the value to be loaded isnt included in the instruction, it is quickly accessible since it is stored in the 8051s Internal RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address--which may be variable.

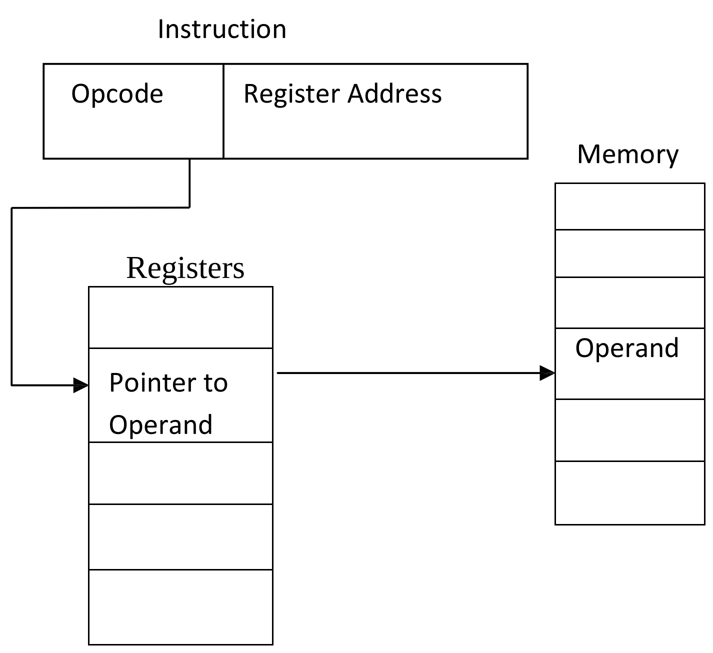
Also, it is important to note that when using direct addressing any instruction which refers to an address between 00h and 7Fh is referring to Internal Memory. Any instruction which refers to an address between 80h and FFh is referring to the SFR control registers that control the 8051 microcontroller itself.

The obvious question that may arise is, "If direct addressing an address from 80h through FFh refers to SFRs, how can I access the upper 128 bytes of Internal RAM that are available on the 8052?" The answer is: You can’t access them using direct addressing. As stated, if you directly refer to an address of 80h through FFh you will be referring to an SFR. However, you may access the 8052s upper 128 bytes of RAM by using the next addressing mode, "indirect addressing."

3. Register Indirect Addressing Mode:

- We already saw register indirect mode, where the operand is a register that contains a memory address. Indirect addressing is a very powerful addressing mode which in many cases provides an exceptional level of flexibility. Indirect addressing is also the only way to access the extra 128 bytes of Internal RAM found on an 8052 processor.

This mode is similar to indirect addressing. The address field of the instruction refers to a register. The register contains the effective address of the operand. This mode uses one memory reference to obtain the operand. The address space is limited to the width of the registers available to store the effective address.



LDR R1, [R0] **R1 ← M[R0]**

a. **LDR is the operation**
b. **R0 contain the effective address of the source of data**
c. **R1 is the destination**

The above instruction moves the contents of data pointed to by R0 to R1. This achieves R1 = *(R0). This addressing mode is using a register for a base address

Here is another example

MOV A, [R0] **A ← M[R0]**

This instruction causes the processor to analyze the value of the R0 register. The processor will then load the accumulator with the value from Internal RAM which is found at the address indicated by R0.

For example, let's say R0 holds the value 40h and Internal RAM address 40h holds the value 67h. When the above instruction is executed the processor will check the value of R0. Since R0 holds 40h the processor will get the value out of Internal RAM address 40h (which holds 67h) and store it in the Accumulator. Thus, the Accumulator ends up holding 67h.

- The effective address would be the value in R0.
- This is also useful for working with pointers. In the example above,
  – R0 is a pointer, and R1 is loaded with the data at that address.
  – This is similar to R1 = *R0 in C or C++.

- So what's the difference between direct mode and this one?
  – In direct mode, the address is a constant that is hard-coded into the program and cannot be changed.
  – Here the contents of R0

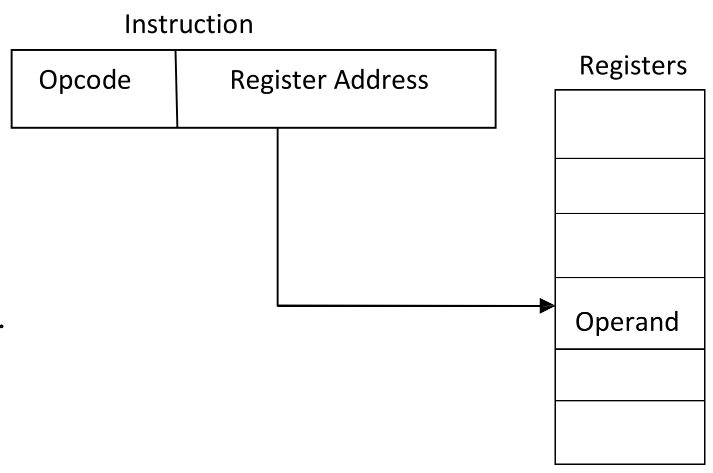
Consider the following instruction

LDR R1, [360] **R1 ← M[M[360]]**

- The operand is a constant that specifies a memory location which refers to another location, whose contents are then accessed.
- The effective address here is M[360].
- Indirect addressing is useful for working with multi-level pointers, or "handles."
  – The constant represents a pointer to a pointer.
  – In C, we might write something like **R1 = **ptr**.

4. Register Addressing Mode:

- In this mode, the operand is in general purpose register. Register addressing mode is similar to direct addressing. The only difference is that the address field of the instruction refers to a register rather than a memory location 3 or 4 bits are used as address field to reference 8 to 16 generate purpose registers. The advantages of register addressing are Small address field is needed in the instruction.



MOV R1, R0 **R1 ← R0**

a. **MOV is the operation**
b. **R0 is the source data**
c. **R1 is the destination**

Here is another example

ADD R1, R0 **R1 ← R1 + R0**

a. **ADD is the operation**
b. **R0+R1 are the source data**
c. **R1 is the destination**

Stepping through arrays

Register indirect mode makes it easy to access contiguous locations in memory, such as elements of an array.

- If R0 is the address of the first element in an array, we can easily access the second element too:

```
LDR     R1, [R0]        // R1 contains the first element
ADD     R0, R0, #1
LDR     R2, [R0]        // R2 contains the second element
```

- This is so common that some instruction sets can automatically increment the register for you:

```
LDR     R1, [R0]+       // R1 contains the first element
LDR     R2, [R0]+       // R2 contains the second element
```

- Such instructions can be used within loops to access an entire array.

5. Indexed Addressing Mode

Operands with indexed addressing include a constant and a register. **Indexed addressing (indexing)** is a method of generating an effective address that modifies the specified address given in the instruction by the contents of a specified index register. The modification is usually that of addition of the contents of the index register to the specified address. The automatic modification of index-register contents results in an orderly progression of effective addresses being generated on successive executions of the instruction containing the reference to the index register. This progression is terminated when the index register reaches a value that has been specified in an index-register handling instruction.

LDR R1, 500[R0] **R1 ← M[R0 + 500]**

- The effective address is the register data plus the constant. For instance, if R0 = 25, the effective address here would be 525.
- We can use this addressing mode to access arrays also.
  – The constant is the array address, while the register contains an index into the array.
  – The example instruction above might be used to load the 25th element of an array that starts at memory location 500.
- It's possible to use negative constants too, which would let you index arrays backwards.

6. PC-relative addressing

We've seen PC-relative addressing already. The operand is a constant that is added to the
program counter to produce the effective memory address.

200: LDR R1, $30 **R1 ← M[201 + 30]**

- The PC usually points to the address of the next instruction, so the effective address here is 231 (assuming the LDR instruction itself uses one word of memory).
- This is similar to indexed addressing, except the PC is used instead of a regular register.
- Relative addressing is often used in jump and branch instructions.
  – For instance, **JMP $30** lets you skip the next 30 instructions.
  – A negative constant lets you jump backwards, which is common in writing loops.

Addressing mode summary

Mode	**Notation**	**Register transfer equivalent**
Immediate	**LDR R1, #CONST**	**R1 ← CONST**
Direct	**LDR R1, CONST**	**R1 ← M[CONST]**
Register indirect	**LDR R1, [R0]**	**R1 ← M[R0]**
Indexed	**LDR R1, CONST[R0]**	**R1 ← M[R0 + CONST]**
Relative	**LDR R1, $CONST**	**R1 ← M[PC + CONST]**
Indirect	**LDR R1, [CONST]**	**R1 ← M[M[CONST]]**

Consider the following instructions:

LDR R1, [R0, #4] R1 ← M[R0 + 4]

This instruction loads the memory contents at address (R0+4) into R1.

LDR R0, [R1, R2] R0 ← M[R1 + R2]

This instruction loads the memory contents at address (R1+R2) into R0.

LDR R0, [R1, R2, LSL#2]

First, the contents of register R2 is left shifted with 2 bits then added to the contents of R1 to form the target R1 to form the target address to retrieve the values from.

LDR R0, [R1], #4

First, the memory contents at address R1 is loaded into R0, then the value of R1 is incremented by #4

LDR R0, [R1], R2

First, the memory contents at address R1 is loaded into R0, then the value of R1 is incremented by R2

LDR R0, [R1], R2, LSL #2

First, the memory contents at address R1 is loaded into R0, then the value of R1 is updated by adding the value of shifting R2 contents by 2 bits.

Number of operands

Another way to classify instruction sets is according to the number of operands that each data manipulation instruction can have.

• Our example instruction set had three-address instructions, because each one had up to three operands—two sources and one destination.

Operation	operands			Register transfer instruction
ADD	R0,	R1,	R2	R0 ← R1 + R2
	↑	↑	↑	
	destination	sources		

• This provides the most flexibility, but it’s also possible to have fewer than three operands.

Two-address instructions

Operation	operands		Register transfer instruction
ADD	R0,	R1,	R0 ← R0 + R1
	↑	↑	
	destination and source 1	source 2	

- Some other examples and the corresponding C code:

ADD	**R3, #1**	**R3 ← R3 + 1**	≡	**R3++;**
MUL	**R1, #5**	**R1 ← R1 * 5**	≡	**R1 *= 5;**
NOT	**R1**	**R1 ← R1′**	≡	**R1 = ~R1;**