Module 10 – Addressing modes

Overview

In programming microprocessors there are several different ways that one can specify a memory address. These different ways are known as addressing modes.

Stored program design (concept map)

Objectives

At the completion of this module you will be able to:

- list the techniques used in programming microprocessors through the use of addressing modes
- give examples of the addressing modes available on the MC68HC11 microprocessor
- show how a typical fetch and execute cycle operates in a CPU.

10.1 Programming techniques

A chosen address mode depends upon one or more of the following:

- The specified address needs to be as short as possible for the job because it takes less memory space to program and less time to operate.
- Sometimes tables of data need to be accessed by a program. There are ways to speed up this process which depend upon the mode chosen.

10.2 Addressing modes

Several addressing modes have been derived for microprocessors and typical ones are:

- **direct** addressing
- **extended** addressing
- **indirect** addressing
- **indexed** addressing
- **relative** addressing

- **stack** addressing
- **inherent** addressing

10.2.1 Direct addressing

In the simple example of programming given in module 8 we used the mode known as direct addressing. Direct addressing means **that the operand address field contains the address of the operand**. For the whole instruction LDA 00, it means load the accumulator with the value stored at location 00.

In a program we have:

Location address	03	OPCODE FIELD	REGISTER FIELD	ADDRESS MODE FIELD	(LDA)
Location address	04	OPERAND ADDRESS FIELD			(00)

which is two 8 bit bytes stored in memory as:

LOCN 03	1 0 0 1 0 1 1 0	(96)
LOCN 04	0 0 0 0 0 0 0 0	(00)

Direct addressing has one important limitation:

- Direct addressing is not efficient for processing data arrays or tables, because it can only access a single fixed address at one time. In many situations we need to scan a whole table of data for one particular value. In this case it is better to use a loop of the one instruction to step through the table stored in memory at many addresses.

Direct addressing can be improved by a technique known as ‘page addressing’. This means memory is divided into imaginary 1000 word pages and part of the address can be kept in a page register and the remainder with the instruction. The program will only be shorter if it seldom requires changing pages. This means we could have 2 registers for the effective address, one for the instruction and one for the page address.

Consider the instruction ADD 2007.

ADD 7	2
Instruction register	**Page register in thousands**

The effective address = 2007 and following execution of this instruction we have:

(new contents of ACC) = (previous contents of ACC) + (contents of LOCN 2007)

In a further development of this technique, by restricting the number of pages the CPU can access at one time, the page register can be eliminated. For instance if all information is in the first page of memory, this is called page zero addressing. Other techniques are then required to access other pages. Another technique is to restrict the CPU to pages with the same number as the instruction MSB. This is called **current page** addressing. These methods may be combined by including a single bit with each instruction (called the page bit) to indicate which option is being used.

Instruction	Page bit
	0 or 1

The page bit is zero for page zero addressing and one for current page addressing.

As an example of page zero and current page addressing, consider the following:

Address	Instruction	Page bit
4010	ADD 38	0

The effective address is 0038 and the result is that the accumulator will have the contents of location 38 added to it. This is page zero addressing. Consider now:

Address	Instruction	Page bit
4010	ADD 38	1

The effective address is 4038 since the instruction is on page 4 and the result is the contents of location 4038 is added to the accumulator. This is called current page addressing.

10.2.2 Extended addressing

Extended addressing is used to access data stored in any address in the 64Kbyte range of memory addresses (0000_{16} to $FFFF_{16}$) using a 16 bit address as follows:

e.g. B6 10 20 LDAA $1020
where the memory address of the required data is $1020.

10.2.3 Indirect addressing

This means that the address of another location which holds the data, is part of the instruction.

The operand address field gives the address of a memory location which contains the address of the operand.

We will note in passing that the Motorola 6800 series microprocessor does not have indirect addressing. However, several processors do and we will consider it here. An example will explain the use of indirect addressing.

Consider the instruction: ADD @ 100

↑

Indirect symbol

This means go to location 100, get its contents and use it as an address, go to that address and add its contents to the accumulator. Suppose we examine another example looking at the actual registers. Initially we have:

Address	Memory contents	
0	LDA @ 50	Program
1	ADD @ 51	
2	STR @ 52	
3	HLT	
20	10	Data
50	20	
51	100	
52	200	
100	30	
200	(40)	

0 — Program Counter (PC)

0 — Accumulator (ACC)

The computer is told to start at address 0 by putting 0 into the program counter. The sequence of events is:

- After execution of the instruction LDA @ 50 at address 0 the computer goes to address 50, gets an address of 20, goes to 20 and gets the data 10 which it loads into the accumulator (ACC).

The program counter (PC) increments itself ready for the next instruction.

The registers are now as follows:

1	Program counter (PC)
10	Accumulator (ACC)

- After execution of the instruction ADD @ 51, the computer goes to address 51, gets address 100, goes to 100, gets data 30 which it adds to the ACC giving 40. The registers are now:

2	Program counter (PC)
40	Accumulator (ACC)

- After STA @ 52, the computer goes to 52, gets 200, goes to 200 and stores the ACC value of 40 there. The PC increments to 3 and the processor halts. The ACC also retains the value of 40 as well.

  This appears more confusing and is slower than direct addressing, however its use in handling tables and arrays is more convenient than direct addressing.

10.2.4 Immediate addressing

This means: **The operand address field contains the operand**. That is the actual data is part of the instruction. Consider the instruction: ADD #4

Immediate symbol

This means add the number 4 immediately to the accumulator's contents. This is handy for introducing constants into a program. For example: LDA #4 puts the number 4 immediately into the ACC which would be faster than using direct addressing for example:

```
M = 4
LDA M
```

Note that for:

d**irect addressing**
extended addressing
indirect addressing
immediate addressing
inherent implied addressing

no additional CPU hardware, i.e. registers or counters are required. For example all of these addressing modes could be implemented on the CPU architecture shown in figure 8.1.

The addressing modes remaining, namely

indexed addressing
relative addressing
stack addressing

all require the use of 'special registers' in addition to the **'Operand address field'** to specify a unique memory address.

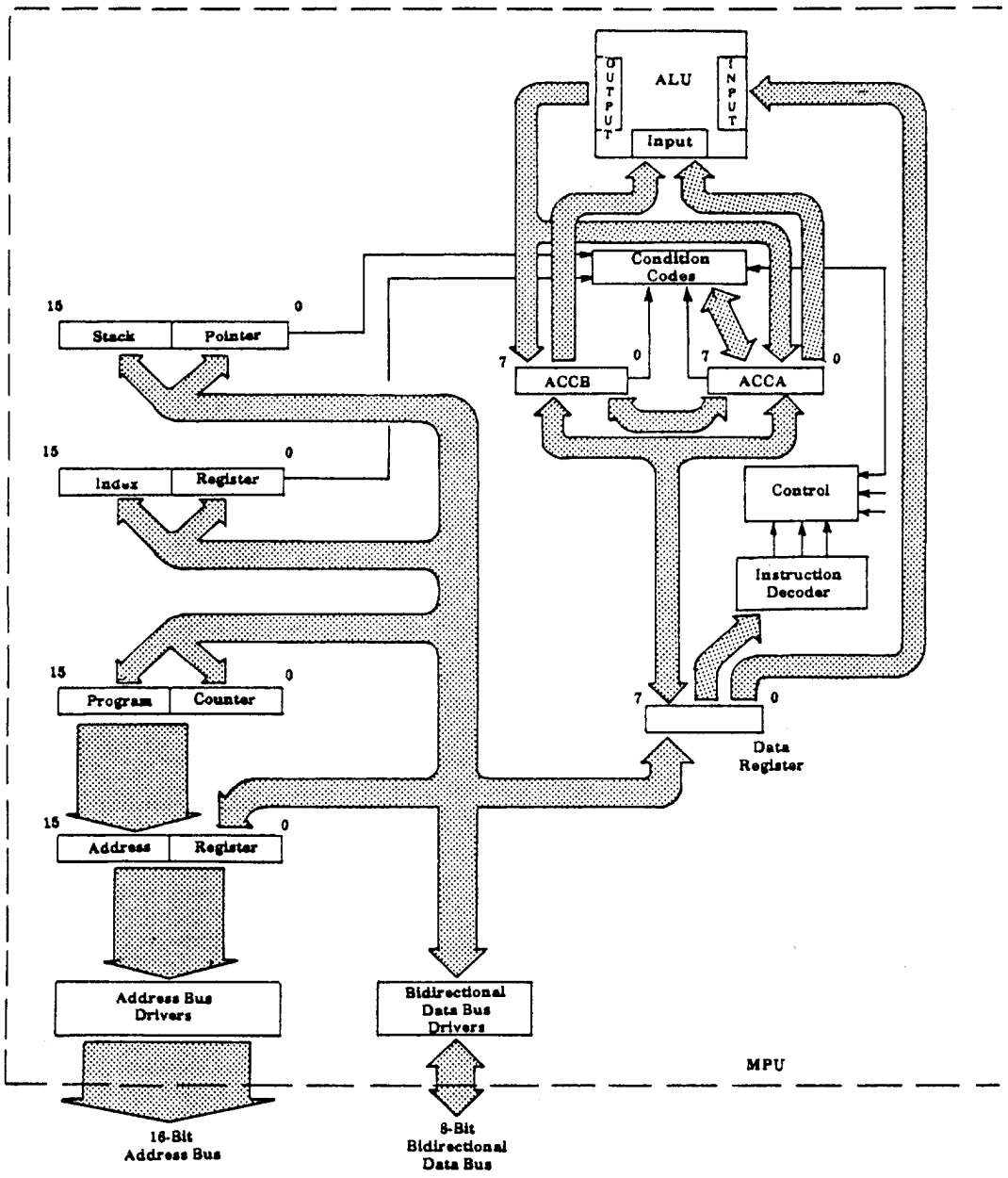
Therefore, to study these modes additional registers have to be added to the general CPU architecture of figure 8.1. These registers are

'The **index** register'
'The **stack pointer**'.

Conveniently, however, the architecture of the M6800 series microprocessor provides basically these features. As it is to be used extensively in subsequent sections it will be used here in preference to a modified CPU based on figure 8.1.

Consider the CPU architecture of the M6800 series microprocessor as shown in figure 10.1.

Figure 10.1: Simplified block diagram of the 6800 MPU

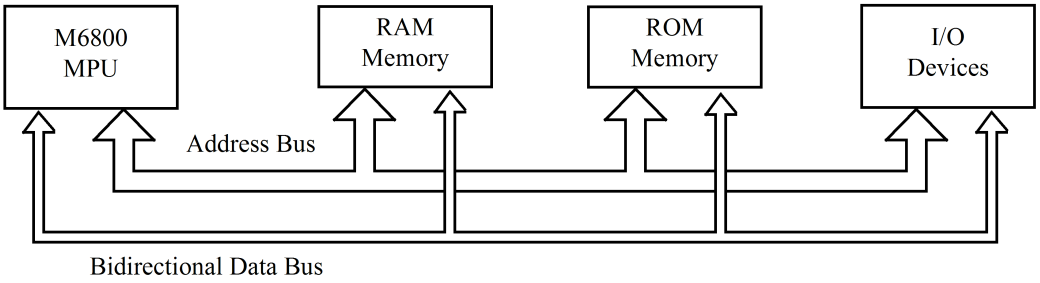


(Source: Motorola Australia Pty Ltd, M6800 Application notes. Reproduced with permission from Motorola Semiconductor Products.)

By comparison with the simplified model in figure 8.1 there are several new additions and modifications. These are:

- There are ‘bus drivers’ which buffer the address bus and the address register, and the data bus and the internal data highways of CPU. The data bus needs to be bidirectional for ‘loading from’ and ‘storing in’ memory.

Note: The normal configuration is as follows:



The ‘Address bus’ consists of 16 lines, giving a 16 bit binary address, which can be connected to all memory chips and input/output (I/O) devices. All I/O devices on the M6800 series microprocessor are addressed exactly the same as memory locations.

The 8-bit data bus is used for the transfer of information between the CPU and the memory or I/O devices. All the devices, which place output data onto the data bus are controlled by tri-state gates and all devices that take data from the bus use clocked latches.

More detail on the system configuration will be considered in a later module. The point to note is that the ‘bus drivers’ buffer the CPU from the heavy electrical load presented by having a large number of external devices connected to the bus as part of the system.

- The program counter (PC) and address register are now 16 bits wide, which means the M6800 series microprocessor can address 0 to 65535 locations. This is normally referred to as a 64K memory address capability.
- The M6800 series microprocessor has 2 working registers known as ‘Accumulator A’ and ‘Accumulator B’. Both perform the same function as the ‘accumulator’ in the earlier architecture. The ‘Register Field’ content of the instruction will determine which accumulator will be used.
- Two new registers called the ‘index register’ and the ‘stack pointer’ are included. Both these are used for specific addressing modes and will be discussed in the following.

10.2.5 Indexed addressing

This addressing mode requires the use of the special register, the **index register**, to calculate the address of the operand.

Address of operand = (index register) + (operand address field)

i.e. In indexed addressing, the ‘address of the operand’ is found by adding the ‘contents of the operand address field’ to the ‘index register’.

Special instructions are provided to operate on the contents of the index register, i.e., to set its value prior to an instruction using indexed addressing.

e.g. LDX load the index register.

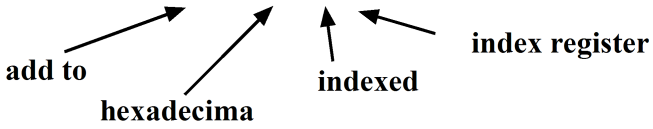
INX increment the index register.

DEX decrement the index register.

STX store the contents of index register in memory.

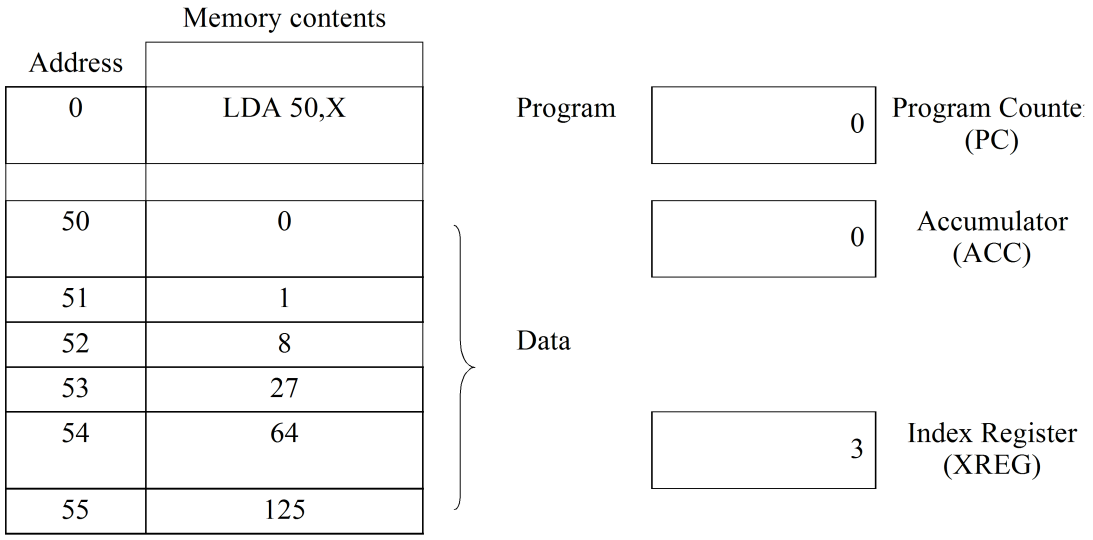
Indexed addressing may also be illustrated through examples.

Consider the instruction: ADDA $15 , X



This instruction means ‘add the contents of the location whose address is (15_{16} + index register contents) into accumulator A’.

Indexed addressing is excellent for accessing a table of values stored in memory. Suppose we wish to obtain the cube of a number. We could store a table of cubes in memory and access it through indexed addressing. Consider the registers:



The number to be cubed is placed in the index register (e.g. 3). When the program is started at address 0, it loads the contents of the location whose address is 50 + 3 i.e. 53 into the accumulator. Hence 27 is placed in ACC which is the desired answer.

10.2.6 Relative addressing

Address of operand = (PC) + (operand address field)

i.e. for relative addressing the address of the operand is found by adding the contents of the ‘program counter’ to the operand address field.

Normally the ‘relative addressing’ in most microprocessors is confined to **branch** instructions.

e.g. BRA * + 5

This would cause the program counter to be incremented by 5 and its contents would then be used as the address in the next instruction fetch. The number 5 is called the OFFSET.

In relative addressing the ‘operand address field’ contains the branch distance relative to the current value of the PC. Two very important factors must be remembered when using relative addressing.

- Negative branch distances are held in memory as 2’s complement numbers i.e. to branch backwards.
- The **value in the PC** will have been incremented **to point to the next instruction before** the branch addition takes place. This must be remembered when calculating branch distances.

As examples of relative addressing being used in branching, consider the following:

Program 1

	Address				
	0		BRA	LOOP	BRA = Branch always Branch to location of LOOP which in this case in address 4.
	1		OFFSET		
PC =>	2		ROLA		
	3		PSHB		
	4	LOOP	ADD		
	5				

This is a **forward** branch problem and remember the PC will have been incremented as soon as it has read the BRA instruction. Therefore, the PC is shown at address 2. Now we wish to branch to the instruction at address 4, so we need to go forward 2 more locations to get there. So we add +2 to the PC, i.e. the OFFSET is 02 in hexadecimal.

Program 2

	Address				
	0	LOOP	CLR		This is a backward branch problem. The PC has read the instruction to branch at address 6 and incremented itself to the next instruction at 8. We wish to go back to address 0, which is back 8 addresses. i.e. negative 8 which is F8 in 8 bit twos complement arithmetic, i.e. the offset is F8 in hexadecimal.
	1		ASRA		
	2		ROLA		
	3		PULA		
	4		NOP		
	5		NEGA		
	6		BRA	LOOP	
	7		OFFSET		
PC =>	8				

10.2.7 Stack addressing

This is a special form of Register Indirect addressing. It is special in that:

- The register is automatically incremented before a '**Load** from memory' operation.
- The register is automatically decremented after a '**Store** into memory' operation.

Usually a special register is set aside for this type of addressing and is called the '**stack pointer**'. The instructions for use with this type of 'register-indirect auto increment/decrement addressing' are also limited to the

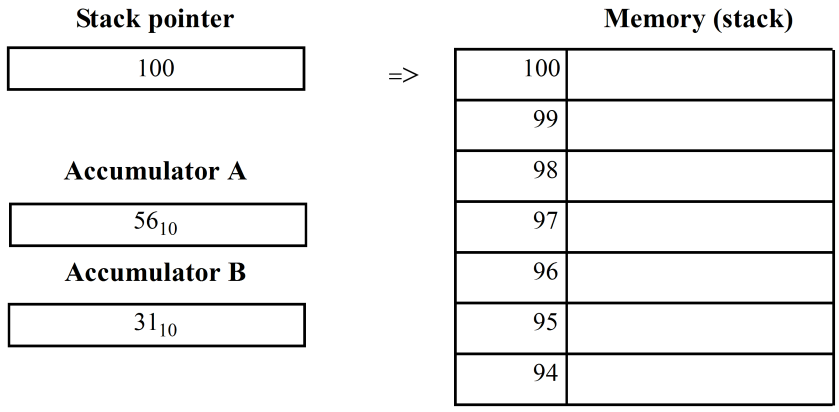
- **Load** from memory called a **pop** or **pull** instruction.
- **Store** into memory called a **push** instruction.

Since repeated use of the same instruction, **i.e. push**, will cause data to be stored in sequential locations in memory then a special area of memory must be set aside for its use. This area is called the **stack**. The stack pointer, (SP), can be loaded etc. to point to a certain section reserved for the 'stack' by special instructions such as:

LDS load the stack pointer

STS store the stack pointer

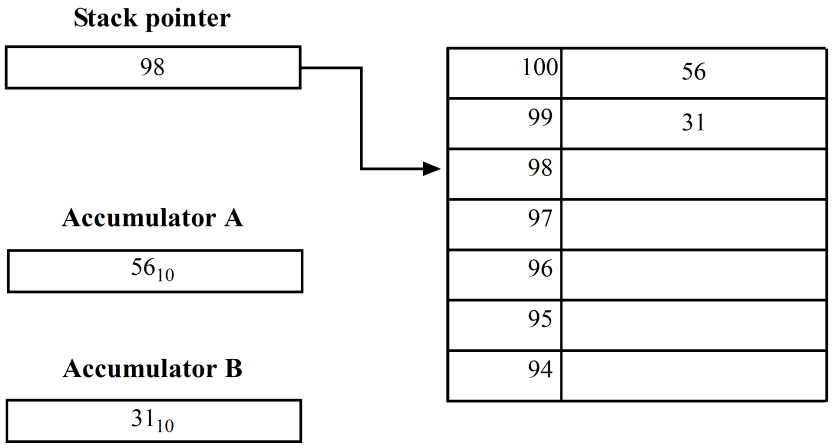
Use of the stack is illustrated as follows.



Initial conditions, are as shown in this diagram.

Following a '**Push A**' instruction, location 100 will = 56, as the contents of accumulator A will be stored in 100. The stack pointer will automatically be decremented to the **next empty location**, 99.

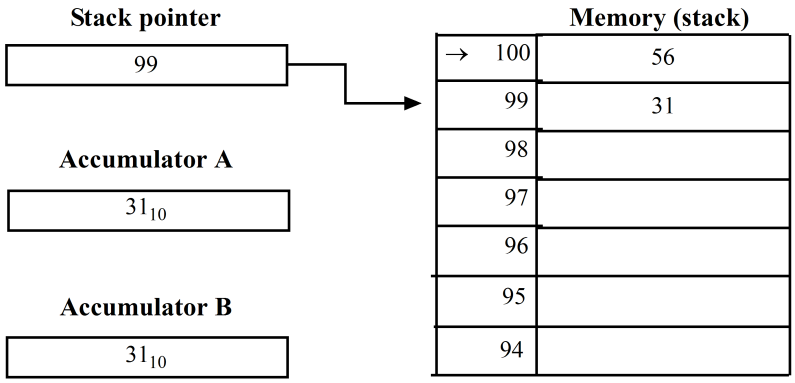
Following a '**Push B'** instruction the contents of the memory, SP, etc. are as shown in the following diagram.



After 'Push A' then **'Push B'** instructions.

Note: For the **push** operation, e.g. **Push A** or **Push B**, the (SP) initially provides the address, after which it is decremented to the next free location automatically.

For a **pull** (sometimes called **pop**) operation, e.g. **Pull A**, the (SP) will first be incremented to point to the last location and its contents used to fetch the data into the accumulator.



After Push A, Push B, then **Pull A.**

Note that the stack pointer, SP, remains pointing at 99 since the contents of this location have been 'pulled off the stack' leaving it as the 'next empty location'.

The **stack** and **stack pointer** are used widely for handling subroutines, interrupts and temporary data storage. When using the stack take care to ensure the contents of the SP are not inadvertently modified and remember it is a last on/first off operation.

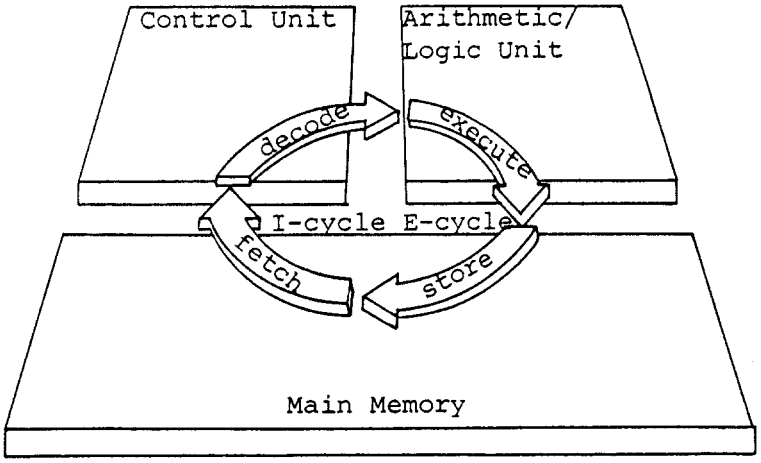
This completes the list of general addressing modes found in computers. Not all processors will have all the addressing modes, the architecture will vary with different makes of CPU. The choice of which CPU provides the best advantages depends on the proposed use of the computer.

10.2.8 Inherent (implied) addressing

Inherent instructions are those that operate on data in the microprocessor registers only. Instructions using this mode of addressing consist of an opcode only. That is, the opcode alone specifies the operation to be performed on a particular register, or registers, e.g. CLRB (clears the contents of accumulator B); ABA (adds the contents of accumulator B to accumulator A).

10.3 Fetch and execute cycle

Typically, the fetch and execute cycle has 4 steps in a computer system. This is illustrated by the diagram shown below. One 'full cycle' is made up of these four steps, which include the 'instruction cycle' (I – cycle) and the 'execution cycle' (E – cycle). The instruction cycle refers to the fetching of the program code from memory and decoding it into its meaning. The execution cycle refers to carrying out the decoded instruction and storing the results in memory.



The Full Cycle

In module 8, you studied the simplified CPU architecture and the instruction format. Although the model of the M6800 microprocessor given in figure 10.1 is still simplified you should now be able to extend your knowledge to understand the data flow sequence for the **'fetch'** and **'execute'** cycle for an instruction such as:

ADD A	$15, X

which is the mnemonic expression for the instruction which will 'add the contents of the location whose address is [15_{16} + (**Index Reg**)] into the accumulator A'.

The content of the instruction as it is held in memory will be as follows:

Address n

OPCODE	Register field	Address mode
ADD	Accumulator A	Indexed

Address n + 1

Operand address field	
01010101	(I.e. 15_{16})

Let this instruction be held in the memory locations with addresses 0127_{16} and 0128_{16} and the initial conditions of all CPU registers are shown in figure 10.2.

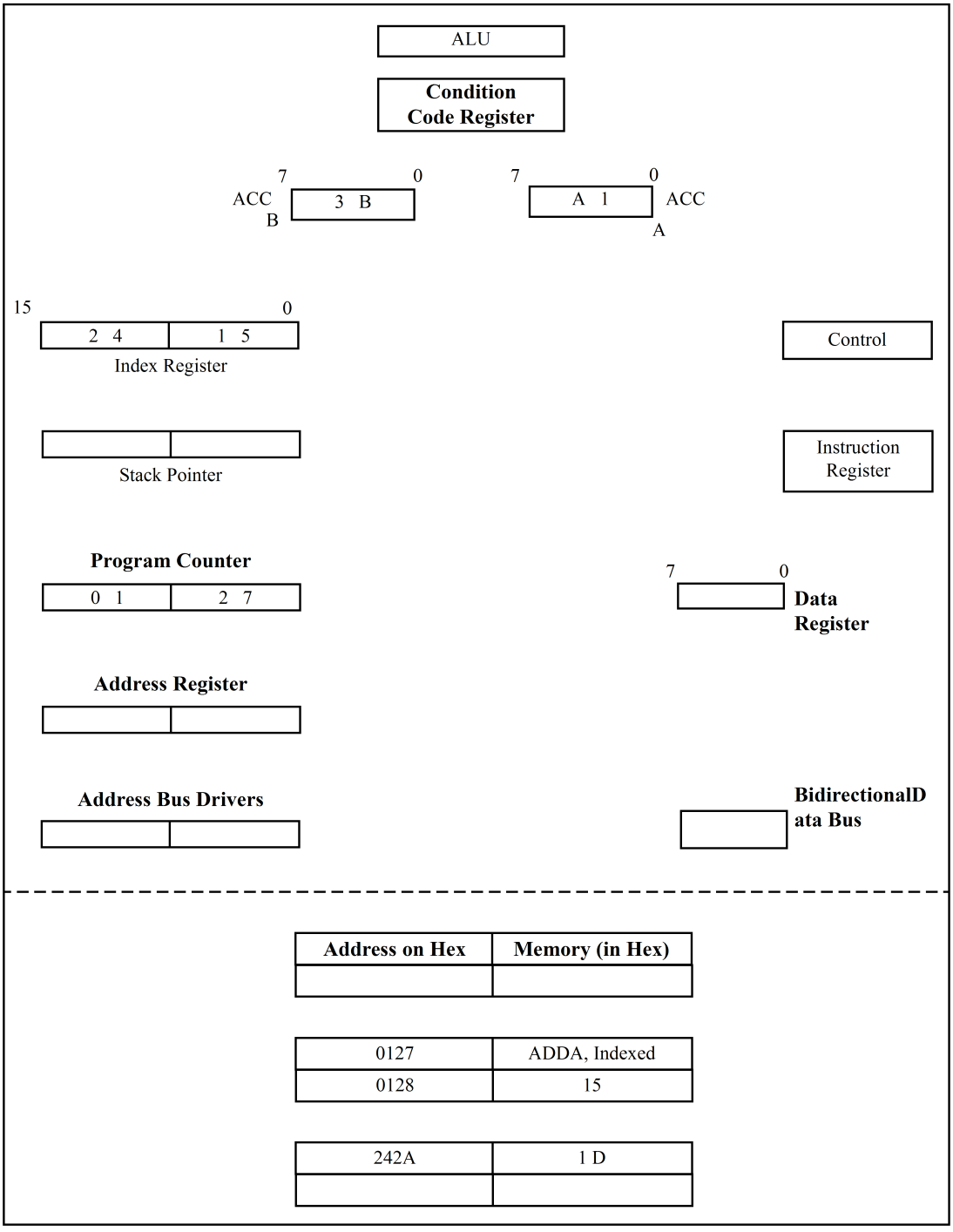


Figure 10.2: Initial conditions

The ‘fetch and execute’ operation for this instruction is then:

- The first byte of the instruction is fetched.
  (PC) → address register. (address reg) = 0127
  (PC) + 1 → PC (new PC = 0128)
  (address register) → address bus
  N.B. The contents of ‘0127’ are the machine code for ‘ADD A, indexed’.
  (data bus) → data register
  (data register) → instruction register
  decode instruction

This is summarised in figure 10.3.

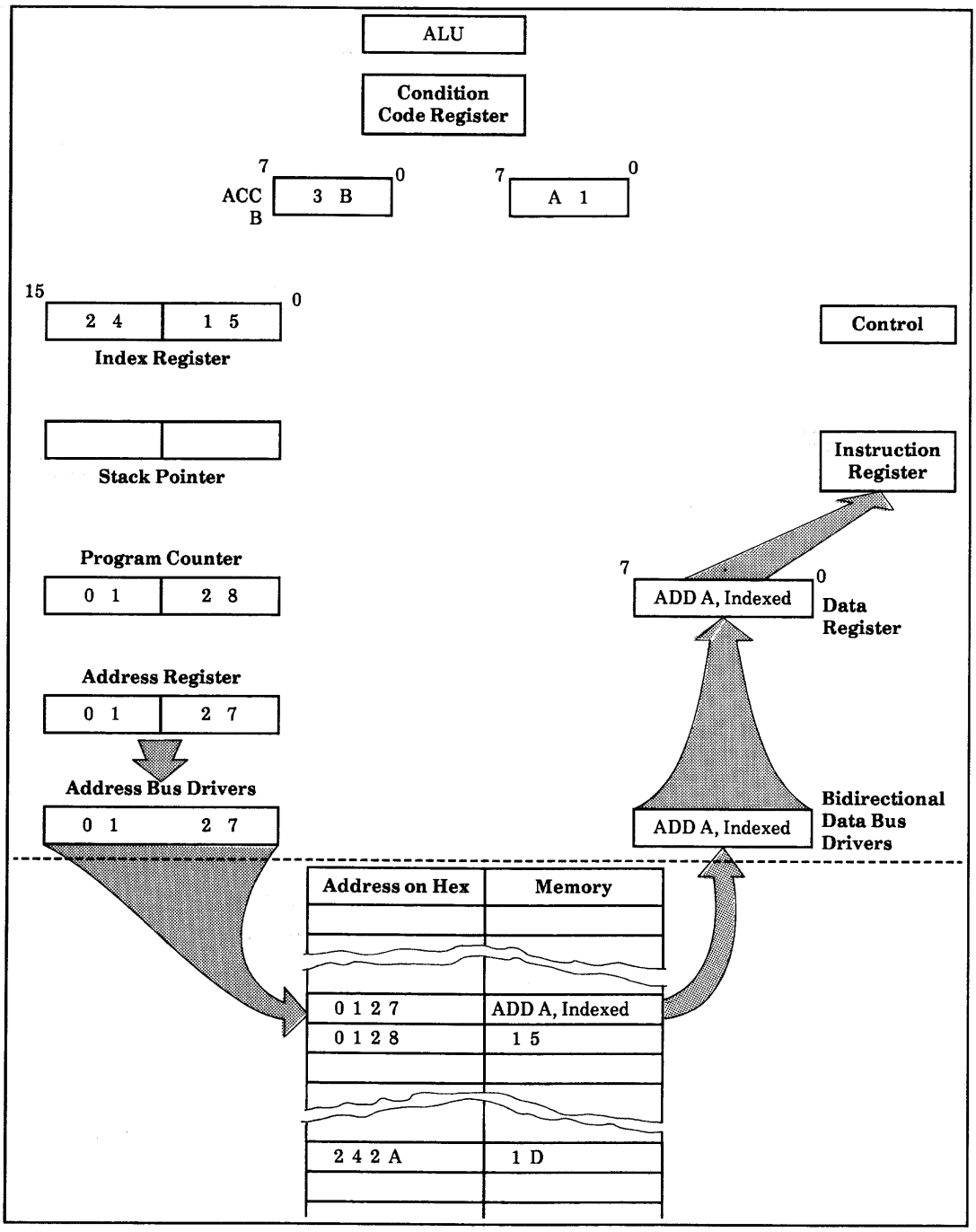


Figure 10.3: Fetch the first byte of the instruction.

- Fetch the second byte of the instruction, the ‘Operand address field’, and calculate the address of the operand.

  Refer to figure 10.4.

  (PC) → address register

  (PC) + 1 → PC (PC) = 0129

  (address register) → address bus

  Memory Read

  (data bus) → data register

  (data register) → one ALU Input

  (index register) → the other ALU Input

  ALU ADD

  (output of ALU) → address register.

i.e. The address of the operand = (**index reg** + 15).

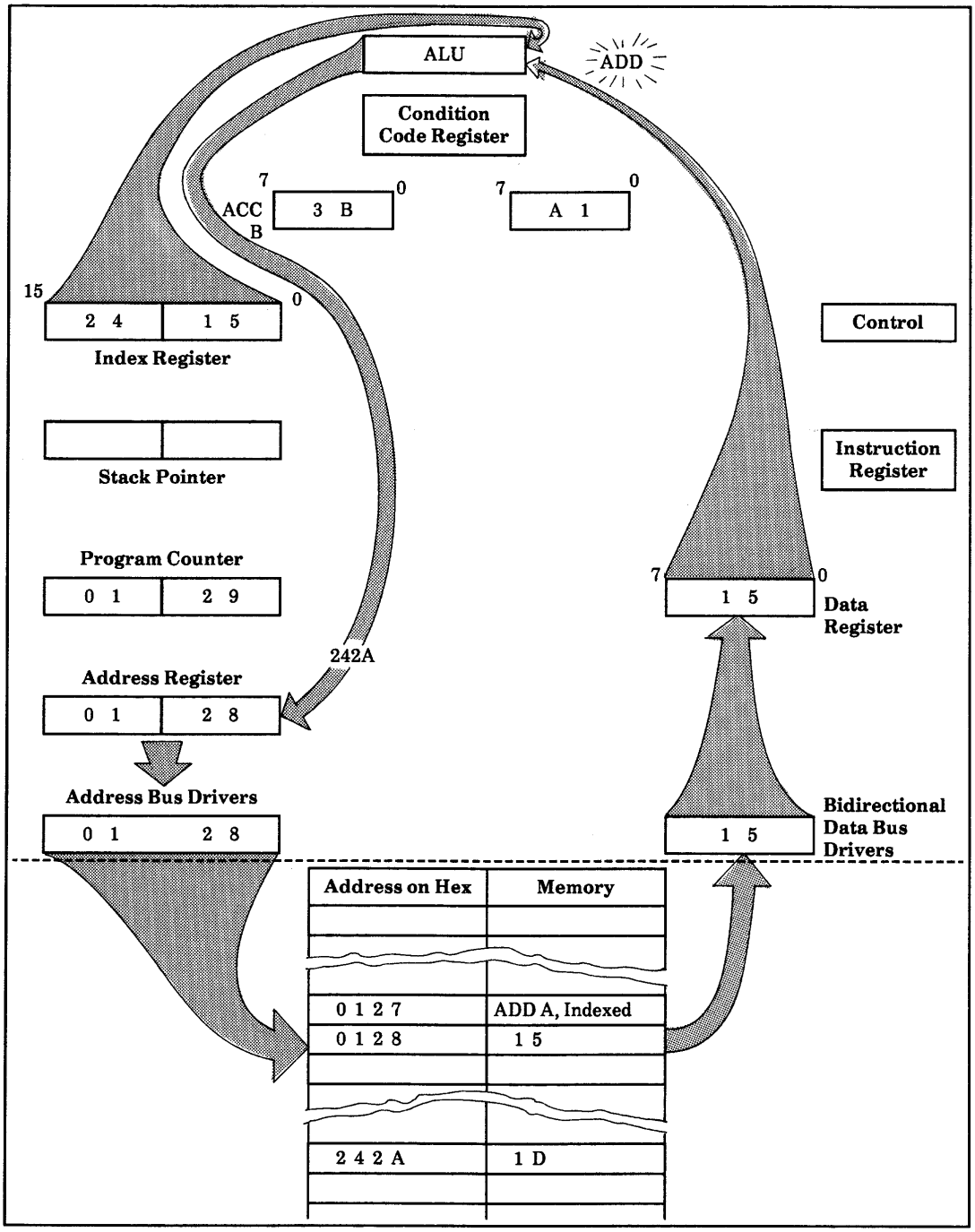


Figure 10.4: Fetch the 'Operand address field' and add it to the index register to determine the Address of the operand

- The operand is fetched and the ‘Add’ operation performed.

  Refer to figure 10.5.

  (address register) → address bus

  memory read

  (data bus) → data register

  (data register) → one ALU Input

  (Accumulator A) → The other ALU Input

  ALU ADD

  (Output of the ALU) → Accumulator A

i.e. At the end of instruction Accumulator A = ‘BE_{16}’

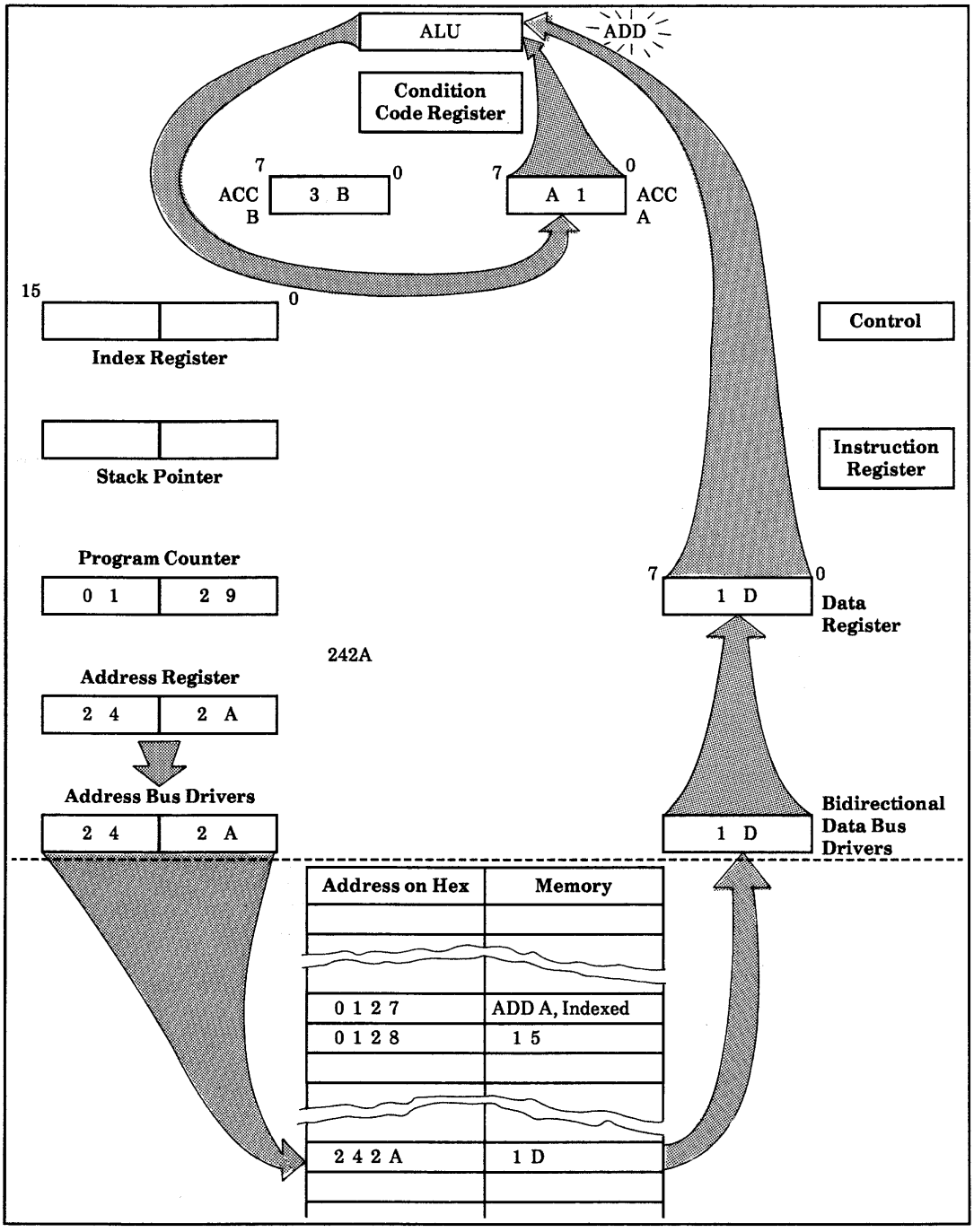
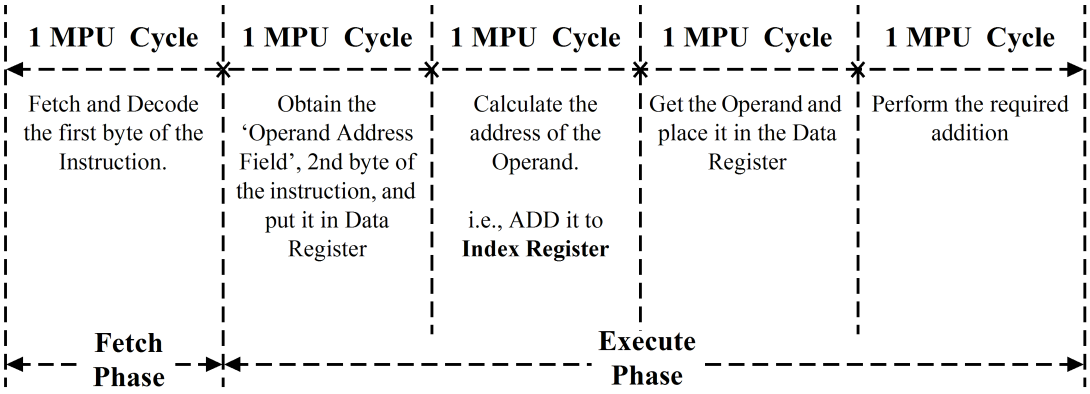


Figure 10.5: Fetch the operand and add it to the Accumulator A

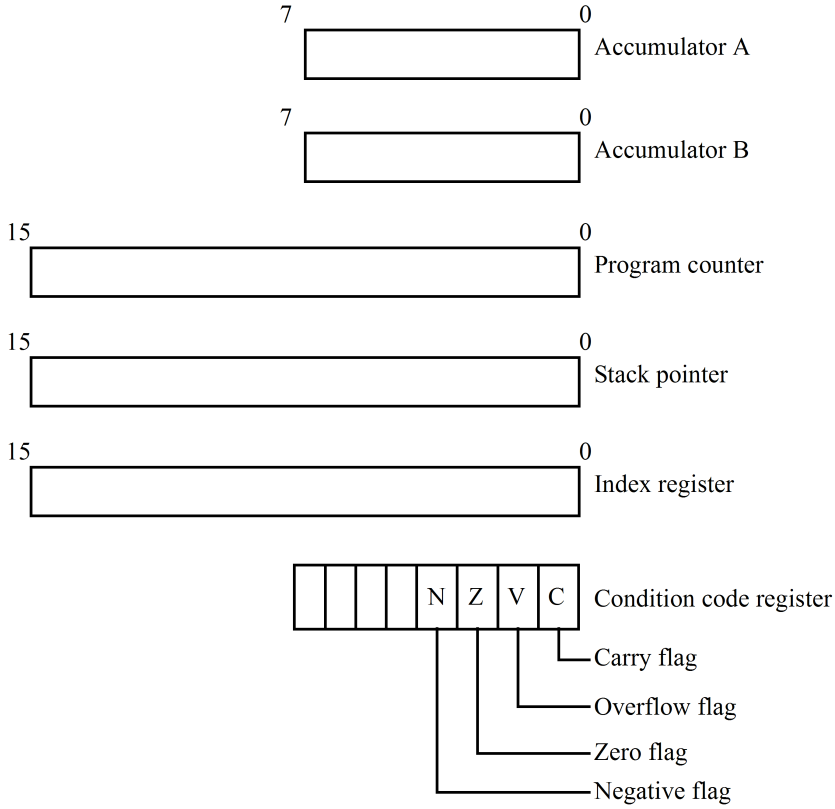
The time taken to ‘fetch and execute’ this instruction is five MPU cycles for the MC6800 (4 for the MC68HC11). This is illustrated as follows.



10.4 Programming Model of the CPU Architecture

Having grasped the concept of the ‘fetch and execute’ operation and also having a basic knowledge of the meaning of the various addressing modes it is often convenient and sufficient to represent the architecture of a CPU as a programming model only.

e.g.



This does not show the interconnection of the registers inside the CPU but it clearly indicates which registers are available for the programmer to use and the size of each.

This model requires the programmer to 'know the function of the various registers and the operation of the addressing mode'. It merely serves as a quick remainder of which registers are available and their size. When programming, this is all that is necessary since all the programmer needs is an understanding of the instruction set and the results obtained from the execution of each instruction.

Self assessment

1. Explain, using examples, why a microprocessor has a range of addressing modes.
2. List five (5) address modes used by Motorola in the MC68HC11 microprocessors. Select two modes from the list and, giving examples, describe their use in detail.
3. For the instruction CLRA ($4F), list the fetch and execute steps carried out by a typical microprocessor.

	OP Code	Register field	Address mode
Address $1020	CLR	A	Inherent