DIGITAL MAGIC EXPOSED

Book 1 – A Simple Digital Processor

Any sufficiently advanced technology is indistinguishable from magic – Arthur C Clarke's Third Law

I. Forward

Digital devices are in use everywhere by most people. Typically included are smartphones, lap-top computers, tablets, ebooks, home appliances etc. Some users may be curious about how these actually work. The three books in this series are intended to provide an insight into the digital processing which underpins their inner working.

At the heart of all the devices is a “processor”, a machine which takes in information and performs activities according to a set of instructions. The result can appear sophisticated and complex but is actually as a consequence of doing very simple things very quickly.

Naturally, the subject matter can appear a little dry since it involves a lot of description of technical processes. These are laid out simply and clearly and should not involve long periods of concentration. The books are aimed at the curious and no initial expertise is required. Each book consists of two parts with more complex features omitted from the first part of each book and the second parts may be skipped.

The subject is described in this part by introducing a machine which can perform the essential functions of a processor. The machine is referred to as the “simple processor”. The description uses only very easy arithmetic principles and includes a set of instructions controlling the simple processor.

Book 2 describes an emulation of the simple processor. The emulator provides an animation to demonstrate the description in this part. The emulator also provides the means to write programs, assemble them and produce executable program code for the simulator described in Book 3.

Book 3 describes a simulation of an electronic design of the simple processor. The simulator can run the programs written using the instructions described in this part and produced by the emulator in Book 2.

The three books of the series can be read consecutively or concurrently with this part. A good approach is to read all the Part 1s before the Part 2s. If the Part 2s are skipped it is recommended that the “Concluding Remarks” in Book 1 Part 2 is read to complete the series.

- Book 1 (this book). The book is divided into two parts:
    - Part 1 – The concept is introduced in plain English and is intended to be understandable to anyone.
    - Part 2 – contains additional descriptions of logic, arithmetic, the simple processor control system and examples of more complex processor features omitted from this part.

- Book 2 describes a pc-based emulation of the processor described in this part. The emulator demonstrates the processes occurring step-by-step as instructions are executed and can produce loadable programs for the simulator described in Book 3. The emulator can accompany this part to aid clarity to the text. Book 2 contains two parts:

  - Part 1 – describes installation of the emulator and its operation.
  - Part 2 – provides some detail on the construction of the emulator to aid its further development.

- Book 3 describes a pc-based electronic circuit simulation of the processor described in this part. The electronic circuit tool running the simulator is freeware available online from enthusiasts and instructions are provided for access, loading and operating the tool. The processor simulator demonstrates operation step-by-step or as a running program. Book 3 consists of two parts:
  - Part 1 – describes the installation of the tool and the loading and operating of the processor circuit simulator.
  - Part 2 – provides a full description of the simulator electronic circuits.

Finally, an acknowledgement to an out-of-date and out-of-print book is in order since it first fuelled the author’s interest in digital processing:

Electronic Computers – Teach Yourself Books 1962 F. L. Westwater

This book appears dated today. It is true that much of the content relates to a bygone age. However, the original edition appeared only 14 years after the SSEM (“Baby”) – the world’s first stored program computer. The book is short and succinctly covers the subject matter. It was understandable (with some concentration) and inspiring to a 14 year-old.

II. Terms

Many technical terms enter into general use in everyday language. To avoid possible confusion or misunderstanding some of the terms used in the books are defined here.

Software – this is written text which is meaningful to a software engineer but not to a computer processor. However, through a tool called a "compiler", software generates Machine Codes which are understandable to a processor. Software is not of concern here. Applications or "Apps" are built from software.

Code – this is sometimes used on its own as a shortened form of "Software Code" i.e. software. In these books the word "code" is always preceded by a defining term or context and does not refer to software.

Instruction Code, Instruction Mnemonics, Instructions - this is written text which has a direct conversion (i.e. one to one) into the binary Machine Codes meaningful to the computer processor. The text is more readable to a person than a binary code. The Instruction Code consists of "instruction mnemonics" and any associated data which facilitate specific actions in the processor. Often the term instruction mnemonics is shortened to "instruction".

Machine Code – The numerical value in binary of the instruction mnemonics and data incorporated into and meaningful to the processor machine. This is sometimes known as "firmware".

It takes very many instructions to complete even simple tasks and it would be very tedious indeed to write Instruction Code to complete the required tasks in real life. Software provides an abstract development environment (i.e. makes the inner working of the machine unimportant) and generates very many Machine Code values from a small amount of written text. Therefore, software is much more efficient in development time and accuracy in coding tasks. Instruction Code is introduced in these books, but not software.

Part 1 – The Simple Computer Processor

1 Introduction

The following describes a simple computing system to give an insight into what is occurring in much larger machines, which only have more parts and additional features. However, essentially, the larger machines operate in a very similar manner. There are very many different forms the system could take. The form chosen for these books is intended to be simple to understand.

The heart of the computing system in the pc, mobile etc is the processor. This is sometimes called a microprocessor due to its small scale compared with physically larger computers in the past. The essential property of the processor is to change input data into output data (Figure 1). What input data, how it is to be changed and where the output is to be placed are all determined by the instructions controlling the processor. The flexibility of the machine is characterised by the ability to alter the instructions, thereby altering the processing.

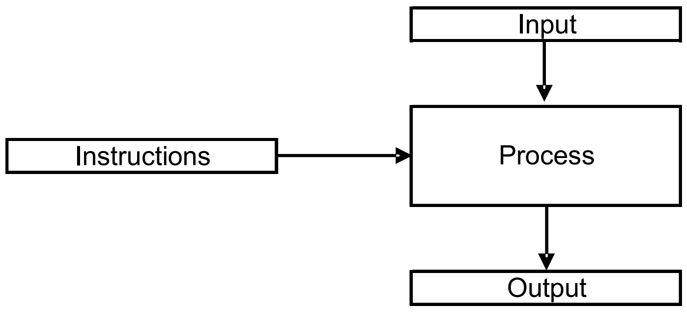


Figure 1 Process changing input to output controlled by instructions

For example, the process flow determined by the instructions could be like the following:

Take the number from location “A”
Subtract the number from location “B”
Place the result in location “C”.

The locations referred to are regarded as computer “memory”.

The true power of the computer processor is its ability to perform different process flows depending upon the data. For example:

Take the number from location “A”
Subtract the number from location “B”
If the answer is less than zero, then take the number from location “D”
Place the result in location “C”.

The number placed in “C” is either: the difference between “A” and “B” (if “A” is greater than or equal to “B”); or is “D” (if “B” is greater than “A”). Note that the part of the instruction following the “then” is skipped if the condition is false (that is, the answer is not less than zero). The altered process flow has the appearance of the processor making decisions.

Therefore, the computer must consist of

- a means to hold data,
- a means to receive instructions,
- to look at and process data held in specific locations,
- to alter the data in specific ways,
- to alter the process flow based upon data,
- to place data back into specific locations.

Before outlining how the processor achieves all this the means by which data are represented is briefly described.

2 Electrical Representation of Data

Computers and electronic memory devices hold information in the most basic form imaginable. That is, the data consists of “bits” of information where a “bit” is held in a device which has only two states: “On” or “Off”. So a bit has only two possible values.

The bits are collected into groups of eight called “bytes”. The eight bits together (i.e. the byte) can contain any combination of “on” and “off” states in any of the eight bits. The total number of states for the byte is 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 = 256 different states. Figure 2 shows four examples:

Off	Off	On	On	Off	On	Off	On

Off	On	On	Off	Off	On	On	On

On	Off	On	On	On	Off	Off	Off

Off	Off	Off	On	Off	Off	Off	On

Figure 2 Example Byte states

The two states are more usually referred to by the numbers “1” and “0”. So the examples in Figure 2 look like Figure 3:

0	0	1	1	0	1	0	1

0	1	1	0	0	1	1	1

1	0	1	1	1	0	0	0

0	0	0	1	0	0	0	1

Figure 3 Here an “Off” bit is 0 and “On” bit is 1

By convention, the byte consists of bits of increasing “significance” from the “least significant” to the “most significant”, in order along the byte. The least significant is regarded as bit b0 and most significant as bit b7, with the other bits b1 to b6 fitting in between.

The byte can be considered to be a number. All numbers can be represented using 1s and 0s in this way and is referred to as the binary system of numbers. Appendix A provides an introduction to binary numbers. Each higher significant bit along the byte regarded as a number is a power of 2 greater than the previous bit. The four examples above, as numbers, have the decimal values shown in Figure 4.

Significance:	Most							Least	Byte
Bit Number:	b7	b6	b5	b4	b3	b2	b1	b0	Value
Bit Value:	128	64	32	16	8	4	2	1	
	0	0	1	1	0	1	0	1	53
	0	1	1	0	0	1	1	1	103
	1	0	1	1	1	0	0	0	184
	0	0	0	1	0	0	0	1	17

Figure 4 Bytes as numbers

The Byte Value is found by adding the Bit Value for each of the 1 bits together.

The byte does not necessarily have to be a number but can be considered to be a code. The bytes are often arranged according to some conventions to represent something understandable to a person or machine. For example, sometimes it is regarded as an encoding of characters useful for printing text. Values for each byte include numbers, letters, punctuation, and control characters for operating printers (for example line feed and page feed). Examples of encoding systems include ASCII and EBCDIC which can be found in Wikipedia.

The eight-bit byte is used in many smaller computing systems and is used in the description of a simple processor system in the following. Larger machines that have evolved over the years are effectively multiple-byte machines (i.e. multiples of 8 bits). This is a generalisation and other examples of bit-widths that have been used include one-bit, four-bit and 12-bit machines.

It is worth noting that writing out binary representations of numbers and characters is tedious, both for writing and reading. Therefore, the bits are arranged into groups of four and represented by a single character as shown in Table 1. The equivalent numeric decimal value is also shown. So a byte contains two such groups.

Binary	Hex	Decimal
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	A	10
1 0 1 1	B	11
1 1 0 0	C	12
1 1 0 1	D	13
1 1 1 0	E	14
1 1 1 1	F	15

Table 1 Binary and Hex equivalents with Decimal value

The characters form the basis of a 16-digit number system (0 to F) called "hexadecimal". Like binary, a whole arithmetic can be studied which uses hexadecimal and some very simple examples will occur later. This is not required for a basic understanding of the machine. Hexadecimal (or Hex for short) is a convenient representation of the binary bits and will be used in most of the following descriptions.

Figure 5 shows a few examples of byte values represented by hex. The decimal value can be found from the binary as seen before or by the hex conversion demonstrated in Appendix B.

Byte	Hex	Decimal
0 1 0 0 1 0 1 1	4B	75
1 1 0 0 1 1 0 0	CC	204
0 0 0 0 0 1 0 1	05	5

Figure 5 Bytes shown as Hex values

Finally, useful information such as texts, documents, spreadsheets etc consist of very many byte characters arranged in a meaningful format, both in terms of human understanding and in a way the computer is able to perform processing. When altering such information, essentially the machine achieves the required output on the input information by processing one byte at a time under the control of the processor instructions.

3 Registers and Memory

To summarise the previous section, bits are held in groups of eight known as bytes. Each physical byte in the computer system is contained in an electronic device holding the values of eight bits.

The bytes to be processed which make-up the information must be held so as to be directly accessible by the processor. This is the computer “memory”. The maximum memory size is determined by the capability of the processor to access directly any particular byte. This is not the same as data storage, which in principle could be unlimited but requires data to be retrieved into memory for processing. For example, a “hard-disk drive” in a lap-top is a data storage device.

The processor needs to access memory and operate on data it receives. To do this, the processor contains a number of electronic devices holding bytes which perform specific functions. These are known as “registers”. For the simple processor described here the registers are shown in Figure 6 and introduced in the following.

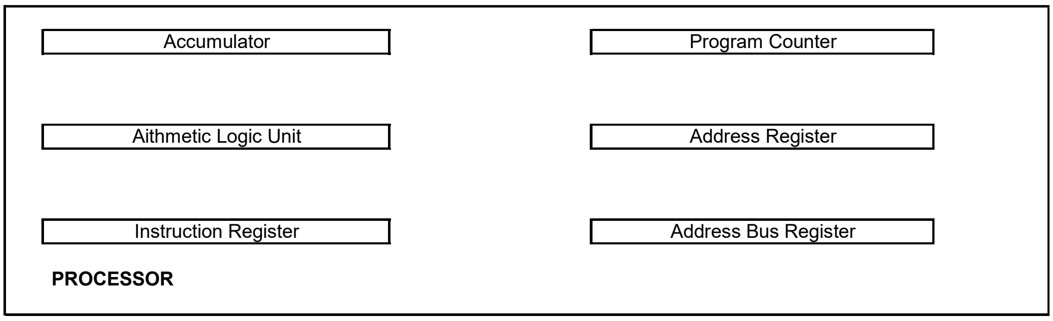


Figure 6 The Registers used in the Simple Processor

3.1 Address Bus Register

Each memory byte’s location must be uniquely accessible. The location is known as the byte “address”. The processor accesses a byte in memory through a register called the “Address Bus Register”. The output value of the Address Bus Register is decoded by an electronic circuit to select a specific memory byte from all memory bytes.

In the example described here the register consists of a single byte i.e. eight bits. This means that 256 bytes of memory is addressable (since a single byte has 256 different states). In practice, this is rather small but is sufficient for the purpose here. Although seldom used today, practical microprocessors using 8-bit data technology use 16-bit Address Bus Registers which can therefore address 256 x 256 = 65536 bytes of memory. Adding each extra bit to the address bus register doubles the memory space.

The Address Bus Register is eight bits and any one of 256 bytes of memory can be addressed. The electronic circuits decode the output of the register to access a particular byte in a manner which can be visualised in Figure 7.

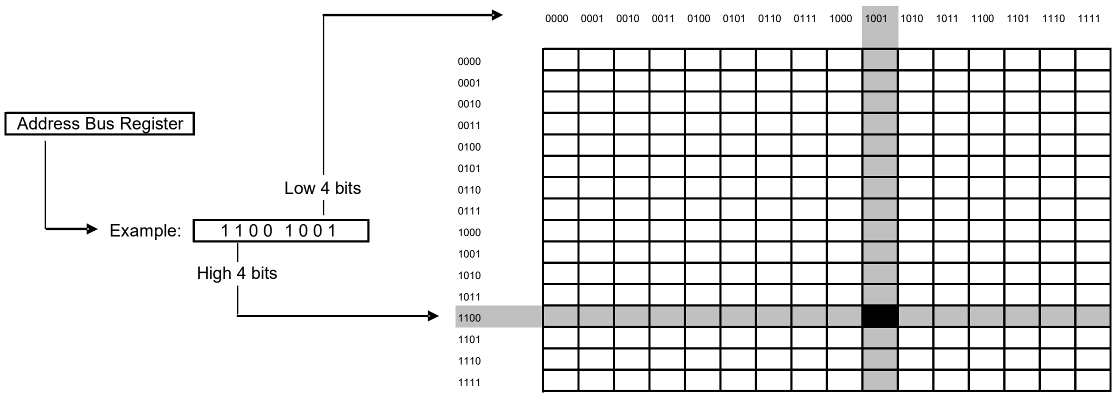


Figure 7 Memory - binary address

The grid represents all 256 bytes of memory. Each 4-bit group of the Address Bus Register decodes a particular row and column in the grid and the intersection of the two identifies the specific memory byte. In Figure 7 the binary address 11001001 identifies the specific location of the memory byte. The address is more readable in the hexadecimal form “C9” as shown in Figure 8.

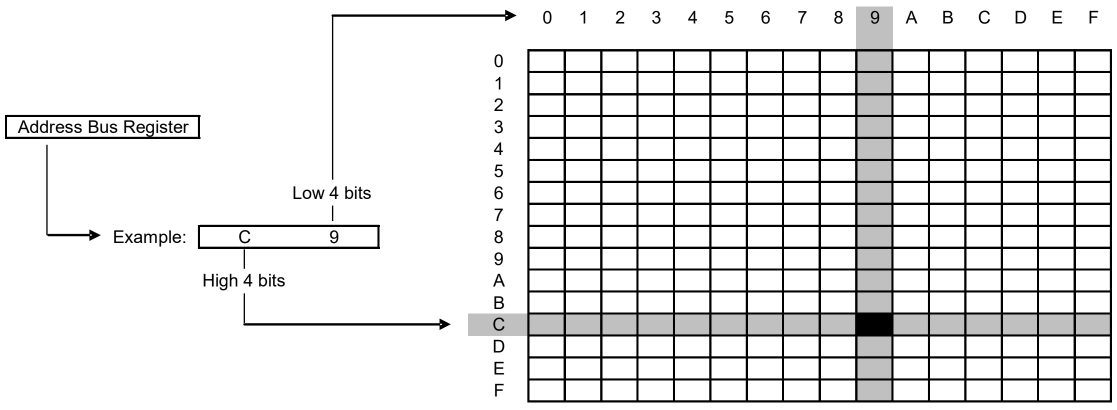


Figure 8 Memory - Hexadecimal Address

3.2 Accumulator and ALU

Information is processed one byte at a time in the processor. (Modern commercial processors are typically 32 or 64 bit machines and therefore handle multiple bytes at one time). The processor reads a byte, does something with it and places the result somewhere. A register called the “Accumulator” holds the data whilst it is in the processor. Some specific operations can occur within the Accumulator and these are described in section 6.

To visualise how the Accumulator, Address Bus Register and memory operate together the following process is considered.

Take the number from location “A”
Subtract the number from location “B”
Place the result in location “C”.

To achieve this, the processor requires something that can perform the arithmetic. This component is referred to as the “Arithmetic Logic Unit”. The Arithmetic Logic Unit (ALU) is fully described in section 5. Essentially, the ALU reads in the content of the Accumulator and a byte in memory and calculates a result. So the processor looks something like Figure 9:

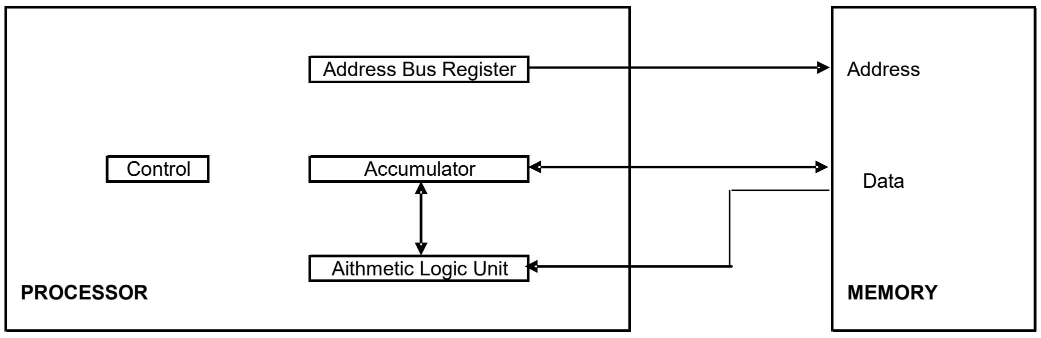


Figure 9 Rudimentary Processor

The value held in the Address Bus Register identifies a specific byte in Memory. Data may be read from or written to that location. Therefore, Memory must also be instructed whether the operation is a “read” or “write” data. This is not shown in the figure. It is operated by the box “Control”.

For the rudimentary processor Figure 9, Control needs to provide the following functionality:

- Data may be read from and written to Memory by the Accumulator.
- The ALU may read data from and write data to the Accumulator (all results pass to the Accumulator).
- Data may be read from Memory by the ALU. The ALU does not need to write to Memory (shown in the figure as separate data lines from Memory although in an electronic circuit this is unnecessary).

Understanding what Control needs to do is achieved by breaking down the steps of the process into simpler steps. This exercise is a very simple form of analysis undertaken by engineers who develop programs running on a computer. The break down is shown in Table 2.

Instruction	Steps
Take the number from location "A"	• Put the address "A" in the Address Bus Register. • Read the Memory placing the data into the Accumulator.
Subtract the number from location "B"	• Put the address "B" in the Address Bus Register. • Set the Arithmetic Logic Unit to "Subtract". • Read data from the Accumulator and Memory into the Arithmetic Logic Unit and subtract them. • Write the result from the Arithmetic Logic Unit into the Accumulator.
Place the answer in location "C".	• Put the address "C" in the Address Bus Register. • Write the data in the Accumulator into the Memory.

Table 2 Instructions broken into Steps

Each **Instruction** represents the simplest level of coding needed to instruct the processor as to what is to be achieved. However, inside the processor there is a further level of breakdown of the Instruction into individual **Steps** which control precisely what is to happen inside the machine. These steps are referred to as the "microcode" by processor designers.

Converting instructions into steps through the microcode is the function of Control. What Control does is determined by loading the instructions into the "Instruction Register".

3.3 Instruction Register

Control determines what the processor machine does next. Of course, this is under the command of the instructions given to it. So what are the instructions and where do these come from?

The instructions are byte codes (called Machine Code) which invoke specific steps (i.e. sequences of microcode) within the processor machine. Control contains a register, the Instruction Register (8 bits in the example here), into which the instructions are placed, one at a time, and from which they are processed. When the processing of an instruction is finished, the next instruction can be loaded.

The instructions form a fixed set (referred to as the "Instruction Set") for a given processor design. The simple processor described here has an instruction set which is effective but would not claim to be an efficient computing machine. However, even a simple instruction set is capable of complex calculation and all software eventually creates instructions consisting of Instruction Codes for a particular processor's instruction set. The full instruction set for the simple processor is described in Appendix C.

The instructions to complete a specific task are also contained in the Memory. This is the computer program. Computers (especially simple ones as described here) can be considered to be sequential machines. That is, they follow a sequence of instructions one at a time from the start and continue until instructed to stop. If the instructions form a processing loop then it may never stop.

The instructions are loaded and processed one after another. The loading is referred to as a "Fetch" and the processing as "Execute". These form an alternating sequence as can be seen in Figure 10.

Fetch	Execute	Fetch	Execute	Fetch	Execute	Fetch	…

Figure 10 Fetch/Execute sequence

Therefore, there is an underlying cycle running which drives the machine through the sequence. The cycle is known as the "Machine Cycle" and this in turn is driven by an electronic ticking "clock". The speed of the clock (i.e. the number of cycles per second) determines how fast the processor is in running programs.

3.4 Memory Access Registers

The processor must know where to start loading instructions from first. This can be achieved in a variety of ways, such as by arranging the electronic circuits to fetch a memory address from a specific location, or simply starting from a specific address. The simple processor here always starts the instruction load from memory location 00 (hex).

The processor must know the address of the next instruction to load from Memory via the Address Bus Register. This is achieved by using another register called the "Program Counter" which holds the address of the instruction memory location. After an instruction is loaded by placing the address held in the Program Counter into the Address Bus Register, the Program Counter is incremented to point to the next location in memory.

Furthermore, the processor must know where in memory to read and write data. Another register "Address Register" holds the address which the processor accesses data in memory via the Address Bus Register. Data access can occur when the content of the Address Register is placed in the Address Bus Register.

Together the registers described form the basis of the simple processor.

4 Simple Processor

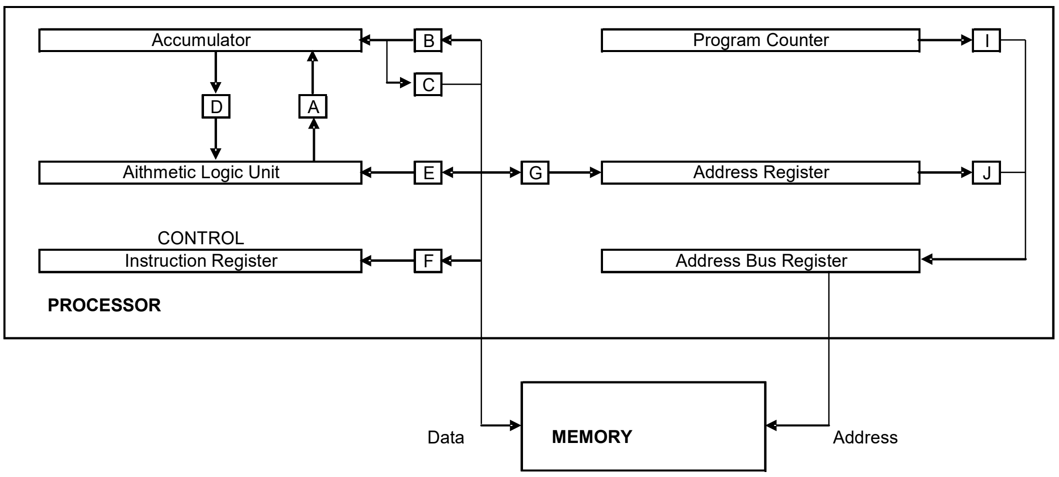


Figure 11 Simple Processor

Figure 11 shows the registers making up the processor described here. On power-up or reset the electronic circuits set the value in Program Counter to 00 (hex). The values in the other registers are set according to what happens when the processor begins to “run”. The processor only runs after the clock is running and stable and the electronic circuits have completed the required setting of preconditions needed by the processor electronics.

The figure includes boxes containing the letters A to J (H omitted and described later). These are referred to as “Gates”. When a Gate is “closed” there is no connection via the lines shown between the registers or memory. When a Gate is “open” data is copied from one register (or memory) to another in the direction of the corresponding arrow in Figure 11. The Accumulator can read and write data with memory through Gates B and C respectively (Gates only operate in one direction). Note that the connecting lines consist of eight wire connections for the bits b0 to b7 (i.e. the byte).

The eight data lines to memory are attached to various Gates which can connect memory to registers. The data lines are known collectively as a “Data Bus” and can connect many components in real devices (smartphones, tablets etc) as well as memory. This is how information is passed around within real systems. Similarly, there are Address and Control Buses in practical machines.

The function completed by each Gate when open is shown in Table 3.

A	Data from the ALU is placed in the Accumulator.
B	Data from the addressed memory is placed in the Accumulator.
C	Data in Accumulator is placed in the addressed memory. This function also sets the memory for a “write”. All other memory access functions are “read”.
D*	Data from the Accumulator is placed in the ALU.
E*	Data from the addressed memory is placed in the ALU.
F	Data from the addressed memory is placed in the Instruction Register.
G	Data from the addressed memory is placed in the Address Register.
I	Data from Program Counter is placed in the Address Bus Register.
J	Data from the Address Register is placed in the Address Bus Register.

Table 3 Gate Functions

* In fact, Gates D and E always act in unison. That is, when the ALU performs a task data is always read from the Accumulator and memory at the same time. Therefore, both Gates are triggered together and both are referred to as Gate E in the rest of these books.

The opening and closing of the Gates is a major function of the Control electronics in the processor. There is a strict logic to the operation of Gates. For example, it is not permissible to open Gates I and J at the same time as the ensuing Address would be in conflict between the Program Counter and the Address Register.

The computer operates by loading an instruction from memory, placing it in the Instruction Register (i.e. Fetch) and processing according to the instruction loaded (i.e. Execute). All Fetch cycles are the same but the Execute cycle depends upon the instruction. An example of the operation of the cycles follows.

4.1 A Basic Operation

A basic operation is now described to demonstrate the cycles and the operation of Gates by Control. The operation involves loading the Accumulator with data.

Two types of load are described. The first involves the instruction “telling” the processor immediately what data to load. That is, the instruction is saying “load this particular value”. The second type of instruction is telling the processor the memory address of the data it is to load.

Instructions create Machine Code, which are binary values in memory, loaded into the Instruction Register and interpreted by Control. To make the instructions more understandable to a person they are given letter short-forms referred to as mnemonics, presumably because whoever chose this word had a classical education. The letters chosen are arbitrary but are intended to provide a clear understanding to a person reading or writing processor instructions. Every commercial processor contains its own set of mnemonics.

Here, the first instruction mnemonic is called LDI ("Load Immediate"). The second is LDA ("Load Accumulator from an Address"). The chosen Machine Code values in this machine are "10" and "11" (hex) respectively. The Machine Code values chosen in a real processor are carefully considered to maximise the performance and minimise the cost of the electronics.

Inst	**Code**	**Description**
LDI	10	Load Accumulator, Immediate
LDA	11	Load Accumulator from Address

Table 4 Simple Load Instructions

Both the instructions have data associated with them. LDI needs to specify the value to be loaded into the Accumulator, and LDA needs to specify the address from which the processor finds the data to be loaded. Data associated with an instruction is referred to as the "operand". This is a mathematical term and is only important here in that some instructions contain an operand and some do not. The operand appears in memory immediately following the instruction mnemonic. So the Instruction Codes here are

LDI operand
LDA operand

All operands are one byte in the simple processor. Commercial processors usually contain instructions with multiple-byte operands. For the simple processor instructions with an operand consist of two bytes and instructions without an operand are one byte.

The example program following loads the number value 240 (decimal) into the Accumulator, followed by a load from memory address 240 (decimal). The value in address 240 is zero. However, the operand data appears in memory in binary, represented here in hexadecimal form. Since 240 (decimal) is F0 (hex), the program in memory (program starts at 00) is as shown in Table 5.

Memory Address	**Instruction**	**Code**	**Description**
00	LDI	10	Load Accumulator, Immediate data
01	F0	F0	Operand for LDI
02	LDA	11	Load Accumulator with data at address
03	F0	F0	Operand for LDA

Table 5 Simple Load Program

The processor runs from 00 after reset. Therefore the program is loaded into memory from location 00. Just before the processor begins, the system looks like Figure 12:

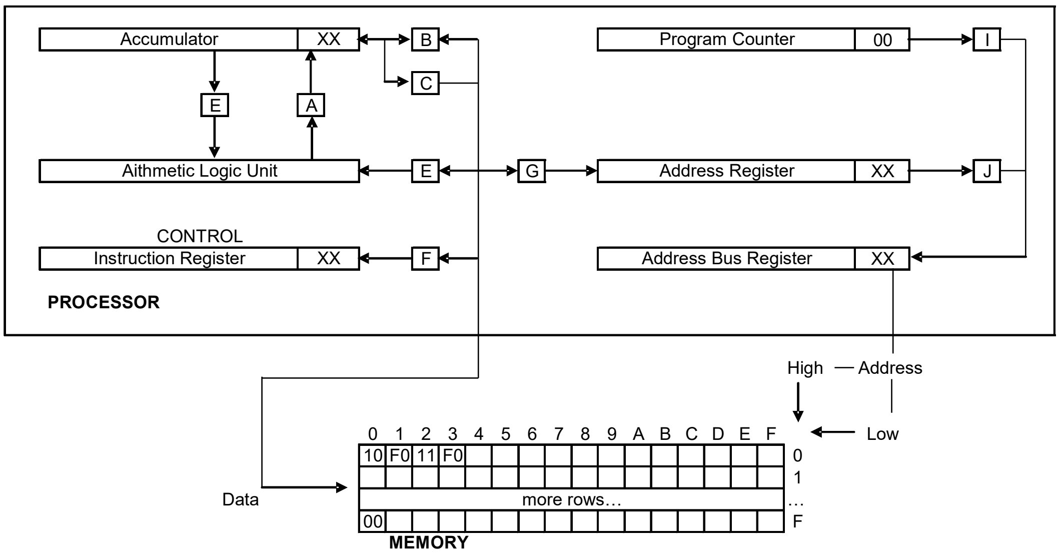


Figure 12 Load Program - Reset State

All the Gates are closed, the Program Counter (PC) is 00, the program is present in memory from location 00 and location F0 in memory is 00. “XX” indicates “Don’t Care”, which means the current value doesn’t matter as it has no impact on what is to follow.

The program starts. In the following figures, open Gates and accessed memory bytes are indicated using dark shading.[1]

[1] The following sequence is animated in the “User Guide” section of Book 2 Part 1

Fetch LDI

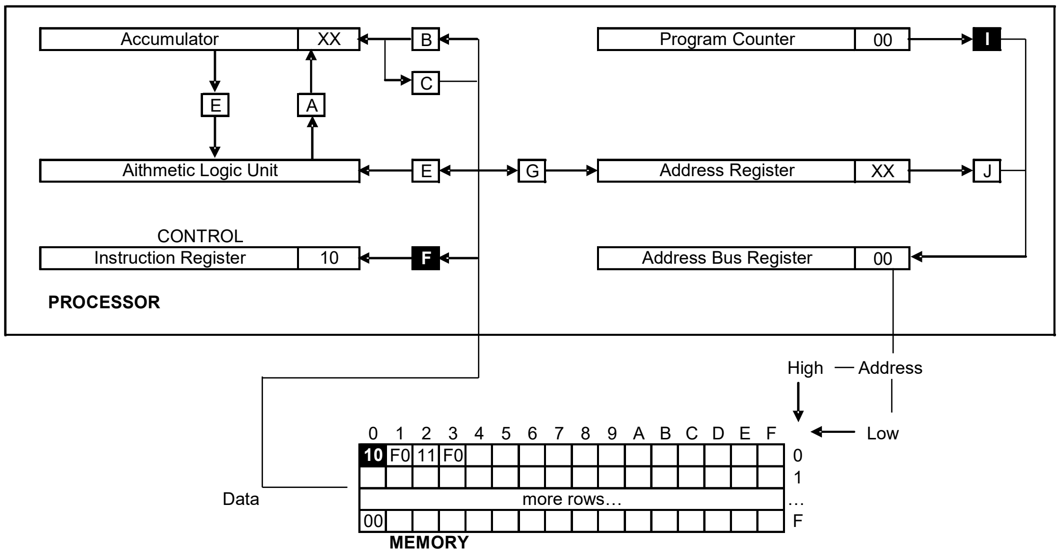


Figure 13 Load Program - Fetch LDI

The processor electronics invoke a Fetch cycle. This cycle is always the same and loads the data addressed by the PC into the Instruction Register (IR). This is achieved by Control opening Gates I and F and reading memory (Figure 13).

The PC value is placed into the Address Bus Register and location 00 is selected in memory. The value stored at the address (10) is read and placed into the IR. Once accomplished, Control closes the Gates and increments the PC (Figure 14).

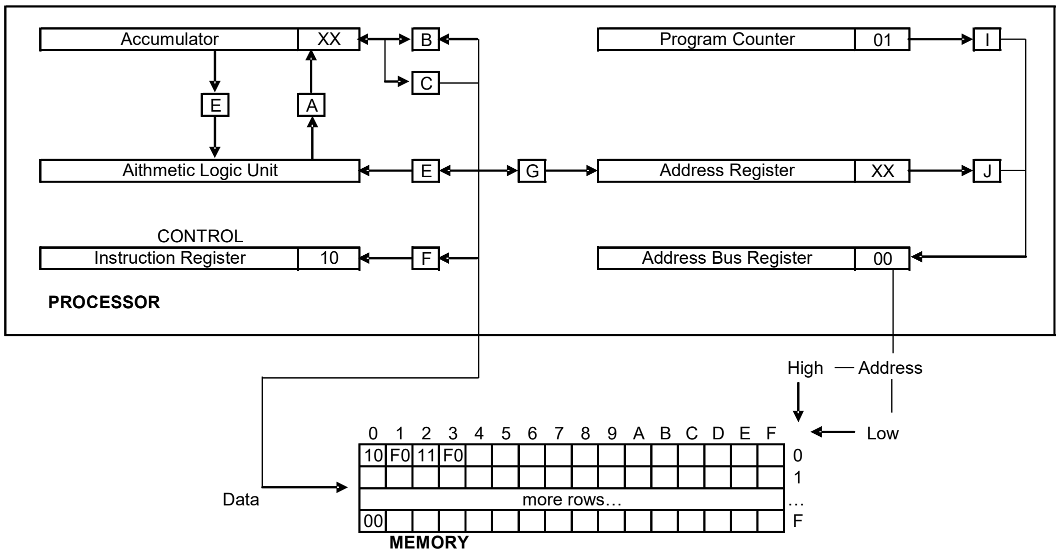


Figure 14 LDI Fetch - Increment PC

Execute LDI

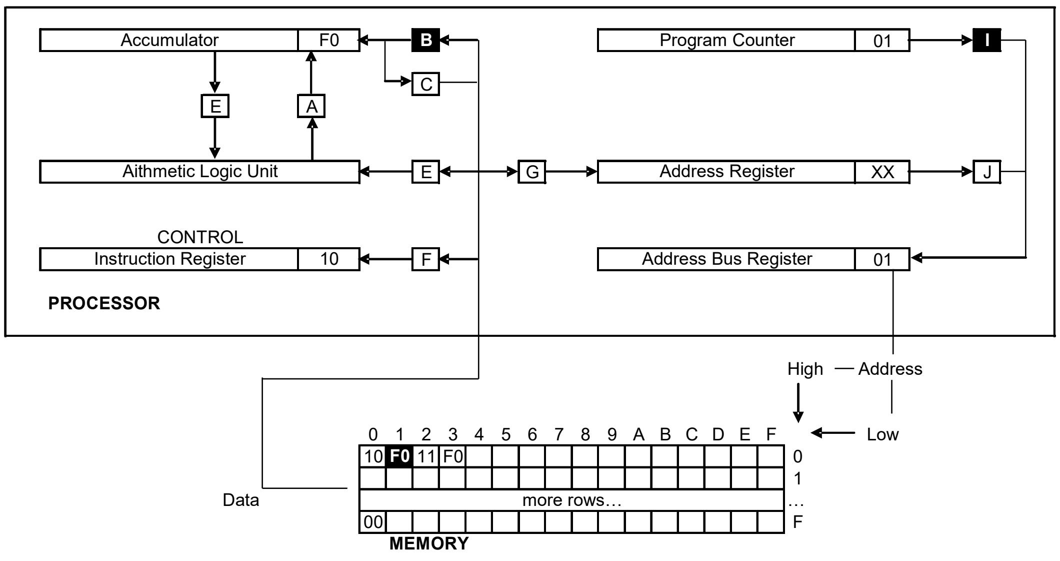


Figure 15 Load Program - Execute LDI

The processor enters the Execute cycle. The LDI instruction loads data provided immediately with the instruction into the Accumulator. I.e. the data byte pointed to by the PC. Therefore, for the LDI instruction, Control opens Gates I and B and reads memory (Figure 15).

The task is completed, so it only remains for Control to close the Gates, increment the PC (Figure 16) and the Execute cycle is completed.

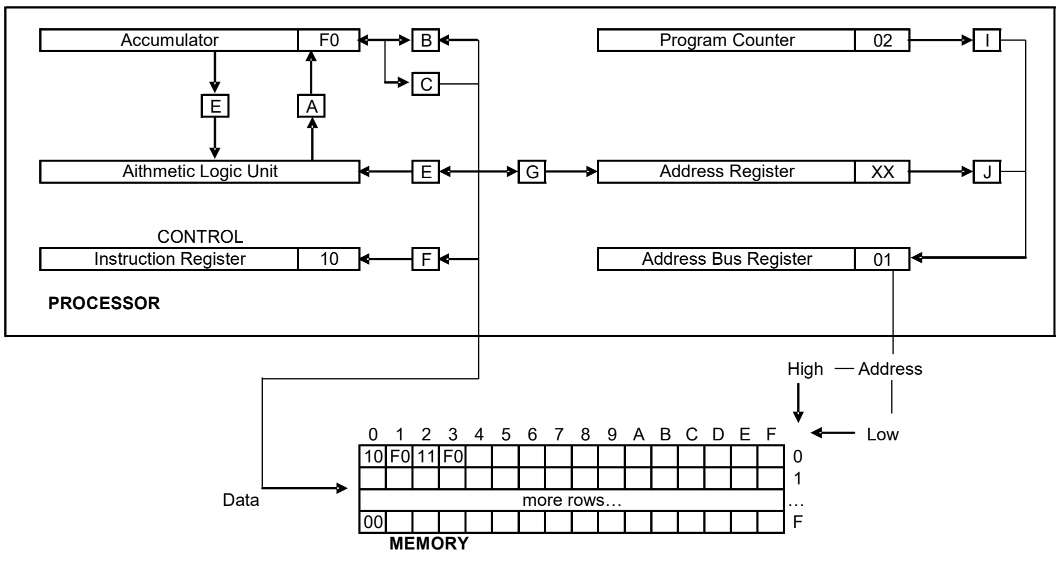


Figure 16 LDI Execute - Increment PC

Fetch LDA

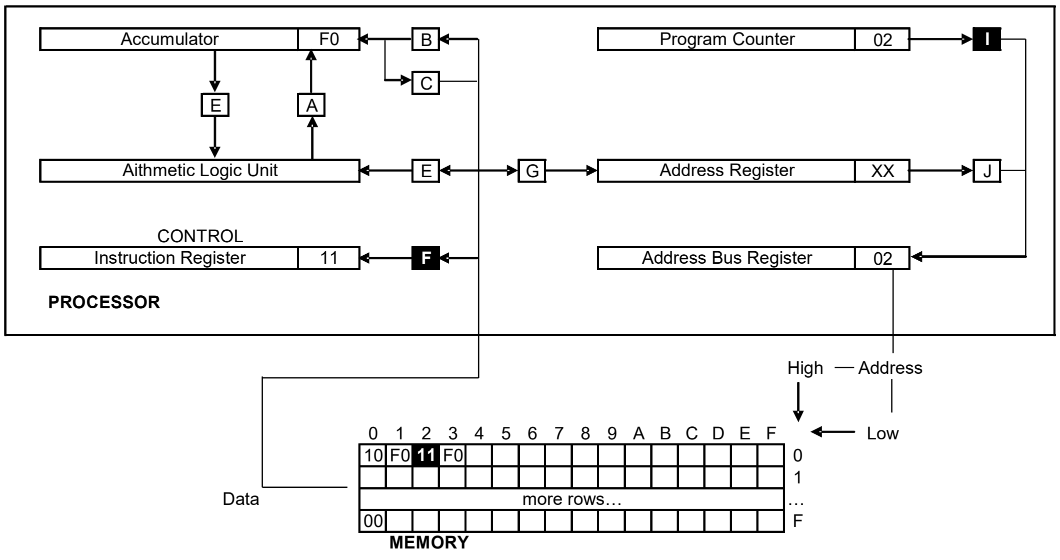


Figure 17 Load Program - Fetch LDA

The processor re-enters the Fetch cycle. The data at the address held in the PC is loaded into the IR (Figure 17).

The value 11 is read and placed into the IR. Once accomplished, Control closes the Gates and increments the PC (Figure 18).

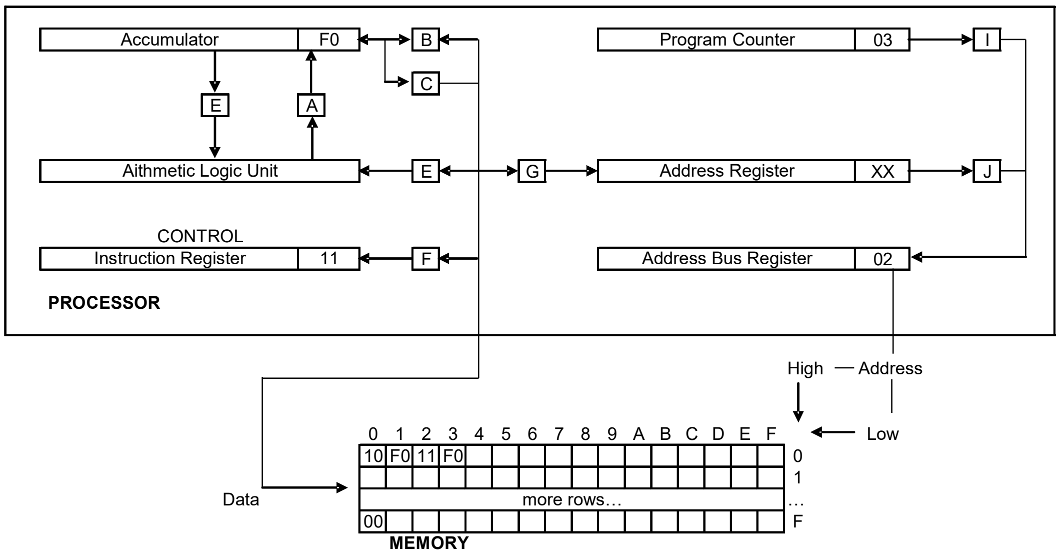


Figure 18 LDA Fetch - Increment PC

Execute LDA

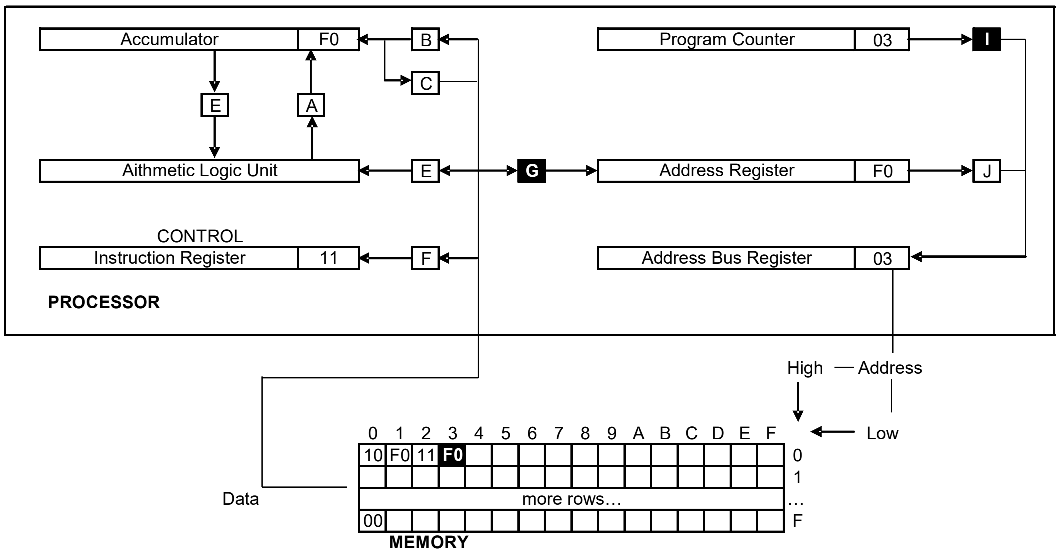


Figure 19 Load Program - Execute LDA 1

The processor enters the Execute cycle for LDA. This instruction uses the data supplied with it as an address to the actual data to load into the Accumulator. So the processor must load data pointed to by the instruction address. In this case, the address is at location 03, which contains F0. To access the address, Control loads the supplied data into the Address Register. I.e. it opens Gates I and G and reads memory (Figure 19).

To load the Accumulator with the required data, Control must now close Gates I and G, open Gates J and B and read memory (Figure 20).

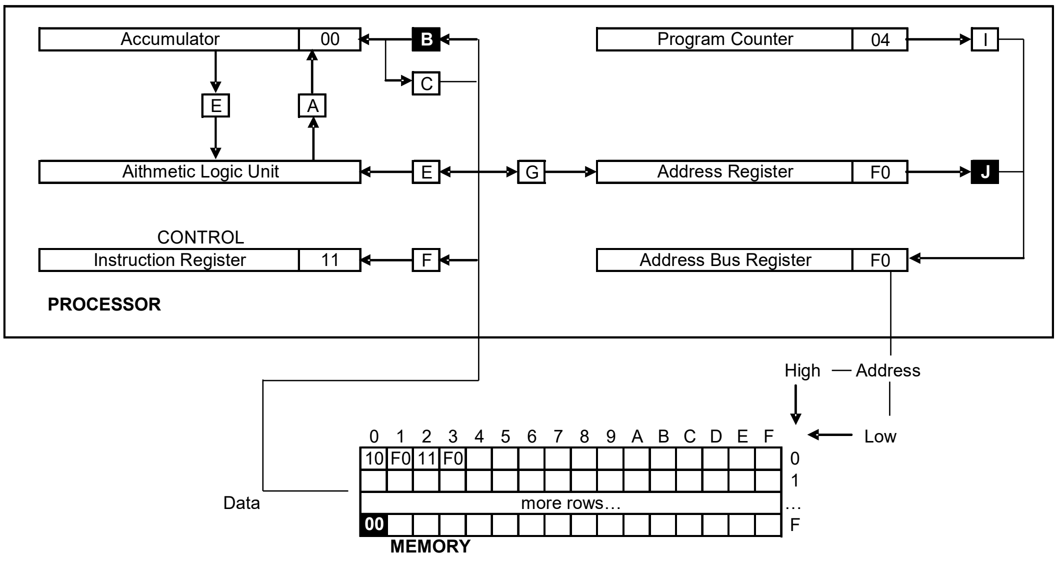


Figure 20 Load Program - Execute LDA 2

The task is accomplished, the Gates are closed and the PC incremented. The Execute cycle is completed.

It is useful to regard each phase opening and closing Gates as distinct Machine Cycles. This is because a practical machine is likely to be designed around such cycles (as is the simulator in Book 3). Therefore, it can be seen from the forgoing that the instructions described consist of the following Machine Cycles:

	Fetch	**Execute 1**	**Execute 2**
LDI	Gates I, F Increment PC	Gates I, B Increment PC	
LDA	Gates I, F Increment PC	Gates I, G Increment PC	Gates J, B

Table 6 Machine Cycles for Simple Load Instructions

So LDI consists of two Machine Cycles and LDA three. Some of the instructions described in Appendix C have five cycles. The processor microcode is developed to deliver the Gate and PC operations for each Machine Cycle in the instruction set. The processor control electronics is informed of the number of cycles required by each instruction. How this is achieved for the simulated simple processor is described in Part 2.

For LDA the final Increment PC could occur in either of the two Execute cycles and which one is determined by practical design (i.e. simplest implementation). Here it is on the first Execute cycle.

All the instructions operate by sequencing the opening/closing of Gates and incrementing the PC. In addition, the instruction may indicate a specific action to be performed. For example, the Gate sequences for ALU operations are the same for addition, subtraction and logical operations. Control sets the specific operation required in the ALU according to the instruction. A full list of the Machine Cycles for the simple processor is included in Appendix C.

The simple load accumulator instructions illustrate how the processor interacts with memory. Microcode is developed for each instruction supported for a particular processor design to operate the gates and memory to fulfil tasks set to the processor. This leaves it necessary to explain how the processor manipulates data and makes decisions. Data manipulation occurs within the following registers

- Arithmetic Logic Unit
- Accumulator

Before this, a Halt instruction is described.

4.2 Halt Instruction

Of course, in the example above, the processor would move on to fetch an instruction from location 04. If the processor is required to stop a “Halt” instruction can be used. This prevents the processor from attempting to execute instructions where no (intended) instruction exists in memory (for example the memory area could contain data).

The simple processor does not distinguish Machine Codes from data and the position of the codes in memory is critical to correct operation. A processing flow adrift from the intended path leads to malfunction usually referred to as a “crash”. Modern processors have built-in safeguards and recovery systems but even these may not be perfect.

The Halt instruction simply stops the PC from incrementing and effectively causes the processor to remain on the Halt instruction. (To this end it impacts the Fetch cycle in a manner unique to this code). The instruction mnemonic for the simple processor is defined as HLT and assigned code “00”. The instruction has no operand.

Inst	**Code**	**Description**
HLT	00	Halt processor

Table 7 Halt Instruction

An electronic hardware reset is required to clear the processor from the Halt.

5 Arithmetic Logic Unit

The processor is required to make calculations on data it is handling. The calculations occur between data held in the Accumulator and data in memory. The Arithmetic Logic Unit (ALU) can perform the following operations on the input data.

- Addition
- Subtraction
- Logical AND
- Logical OR
- Logical Exclusive OR

A summary of the functions follows. More detail is provided in Part 2.

5.1 Addition

Two input binary numbers are added. For example, if the numbers 35 and 17 are added the result is 52 (Figure 21):

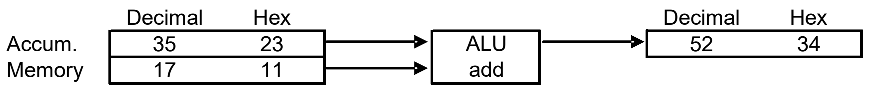


Figure 21 Simple addition

The result of adding two numbers could be greater than 255 (all bits set to “1” in the register). Therefore, there needs to be a “Carry” flag to indicate an overflow.

The instructions for addition in the simple processor always check the Carry flag and add one if the flag is set to one. Therefore, at the beginning of addition the Carry flag must be preset to zero. Figure 22 shows addition in the simple processor and an example of an overflow occurring. The Carry flag takes 256 (decimal) forward to any higher-power addition.

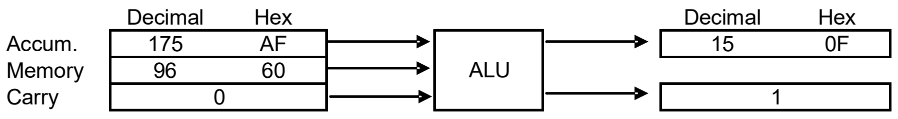


Figure 22 Overflow addition

Larger numbers can be represented over multiple bytes and the Carry flag allows multiple-byte addition to take place. Multiple-byte arithmetic is described in Part 2.

5.2 Subtraction

Two input binary numbers are subtracted. For example, if the numbers 35 and 17 are subtracted the result is 18 (Figure 23):

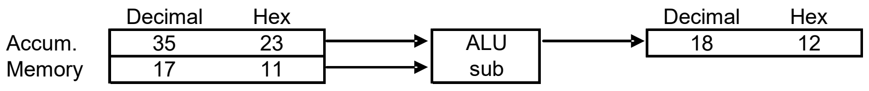


Figure 23 Simple subtraction

Answers for some sums may be less than zero. Negative answers are somewhat more complicated since their interpretation is a little trickier. Negative answers are described in Part 2.

The Carry flag indicates if the number is positive or negative and has the value one for positive answers. The Carry flag is preset to one at the beginning of subtraction (for a number to be subtracted it must be positive). Therefore, the subtraction in Figure 23 is more precisely described by Figure 24. More detail on the operation of the Carry flag is given in Part 2.

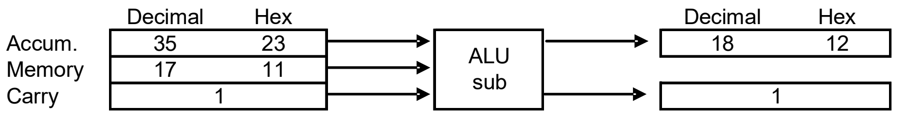


Figure 24 Simple subtraction showing Carry

For numbers greater than 255, multiple-byte subtraction is used. Part 2 describes multiple-byte subtraction.

5.3 Logical Operators

The logical operators are

- Logical AND
- Logical OR
- Logical Exclusive OR

These are “bit operations”. That is, the output of each bit of the byte from the operation is only determined by the corresponding bits in the input bytes, without influence from any other bit position. The byte from memory and the Accumulator byte feed into the operation and each corresponding bit of the two bytes is processed.

The logic applying for the three operations is shown in Table 8.

AND	For each input bit position, the output AND of the corresponding bit is “1” only if ALL of the input bits are “1”, and is “0” if ANY of the input bits is “0”.
OR	For each input bit position, the output OR of the corresponding bit is “1” if ANY of the input bits are “1”, and is ONLY “0” if ALL the input bits are “0”.
Excl OR	For each input bit position, the output Exclusive OR of the corresponding bit is “1” only if one input bit is “1” AND one input bit is “0”, and is “0” if BOTH input bits are “1” or if BOTH input bits are “0”.

Table 8 Logic Operators

The AND and OR logic on bytes in general can apply to any number of input bytes. Exclusive OR only applies to two input bytes.

The ALU only deals with two input bytes (Accumulator and Memory). Each corresponding bit of the bytes read during the operation is compared and the output bit is determined (using the logic described) as shown in Table 9.

Input		**Output**		
Memory bit	Acc bit	AND	OR	Excl OR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 9 ALU Logical operation outputs

Example bytes are shown in Table 10:

Inputs	Memory Accumulator	1 0 1 1 0 1 0 1 1 1 0 1 1 0 0 1

Outputs	AND
	OR
	Excl OR
		1 0 0 1 0 0 0 1
		1 1 1 1 1 1 0 1
		0 1 1 0 1 1 0 0

Table 10 Examples of ALU logical operation

5.4 Instruction Codes

Instructions are issued to the processor (through the Instruction Register) to perform the ALU calculations. During the processing the ALU accesses the data in the accumulator and in memory. As seen before in memory operations, the data from memory may be passed immediately by the instruction, or may be contained in another memory location accessed by an address passed in the instruction. Similarly, the ALU instructions indicate if the operand contains immediate data or a data address.

The ALU instruction mnemonics for the simple processor are chosen as shown in Table 11 and include an operand:

Inst	**Code**	**Description**
ADI	48	Add with Carry, Immediate
ADC	40	Add with Carry from Address
SBI	49	Subtract with Carry, Immediate
SBC	41	Subtract with Carry from Address
ANI	4F	Logical AND, Immediate
AND	47	Logical AND from Address
ORI	4E	Logical OR, Immediate
ORA	46	Logical OR from Address
XOI	4D	Logical Exclusive OR, Immediate
XOR	45	Logical Exclusive OR from Address

Table 11 ALU Instructions

Arithmetic instructions require the ability to manipulate the Carry flag explicitly at the beginning of calculations (Table 12). The instructions do not include an operand.

Inst	**Code**	**Description**
SEC	81	Set the Carry flag
CLC	80	Clear the Carry flag

Table 12 Carry Flag operator Instructions

5.5 ALU Layout

The ALU consists of two eight-bit inputs, an eight-bit output and the Carry flag. It is represented in the rest of the books as shown in Figure 25.

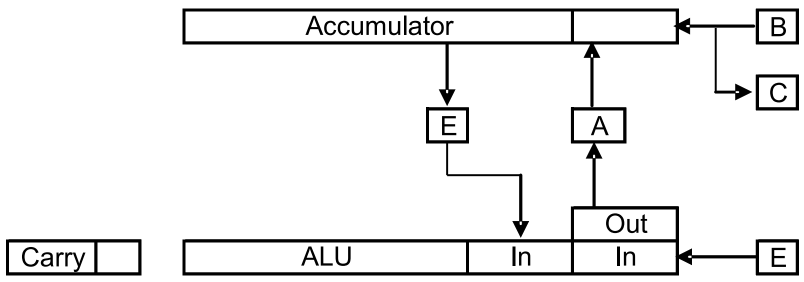


Figure 25 The complete ALU with Accumulator

6 Accumulator Operations

Processor operations move data through the Accumulator. The capability of processors to make decisions based upon data derives, in part, from the data held in the Accumulator. Specifically, a flag associated with the Accumulator indicates if the value held by the Accumulator is zero or not zero. That is, if the value held is 00 (hex), the “Zero flag” is set to “1”. For ANY other Accumulator value the flag is set to “0”. The Zero flag is included in further Accumulator diagrams in these books.

Sometimes it is useful to shift bits along a register. That is, a bit value is moved into one of its neighbouring bits. For example, each “left shift” of bits representing a number multiplies the number by two. Figure 26 shows an example of left-shifting a byte and doubling its numeric value.

	128	64	32	16	8	4	2	1	Hex	Decimal
	0	0	1	0	1	1	0	1	2D	45
left shift	0	1	0	1	1	0	1	0	5A	90
left shift	1	0	1	1	0	1	0	0	B4	180

Figure 26 Left shifts multiplies by 2

This forms a basis for the multiplication of binary numbers. In fact, “shifting” and “rotating” the bits in a register find many practical applications in programming. The register used for shifting and rotating in the simple processor is the Accumulator.

6.1 Shifting

Shifting involves moving the binary bits along the Accumulator to the left or right. The bit at the right end during a left shift, and at the left end during a right shift, is set to zero. The bits at the other end of the Accumulator fall out of the Accumulator and move into the Carry flag. After eight shifts the value in the Accumulator would be 00. After nine shifts the Accumulator and Carry flag would be all zeroes.

Figure 27 shows successive left shifts with bit b7 moving into the Carry flag until all bits are zero (nine shifts). Right shifts follow a similar pattern with bits moving the other way and bit b0 moving into the Carry flag.

	Carry	Bit b7	b6	b5	b4	b3	b2	b1	Bit b0
	0	0	0	1	0	1	1	0	1
1	0	0	1	0	1	1	0	1	0
2	0	1	0	1	1	0	1	0	0
3	1	0	1	1	0	1	0	0	0
4	0	1	1	0	1	0	0	0	0
5	1	1	0	1	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

Figure 27 Successive left shifts

6.2 Rotating

Rotating also involves moving the binary bits along the Accumulator to the left or right. The bit at the right end during a left shift, and at the left end during a right shift, is set to the value in the Carry flag. The bits at the other end of the Accumulator fall out of the Accumulator and move into the Carry flag. After the ninth rotate the value in the Accumulator and Carry flag would be back to the start values.

Figure 28 shows successive left rotates with the Carry flag moving into bit b0 and being replaced by bit b7 until all bits are restored to their original position (nine rotates). Right rotates follow a similar pattern with bits moving the other way and the Carry flag moving into bit b7 being replaced by bit b0.

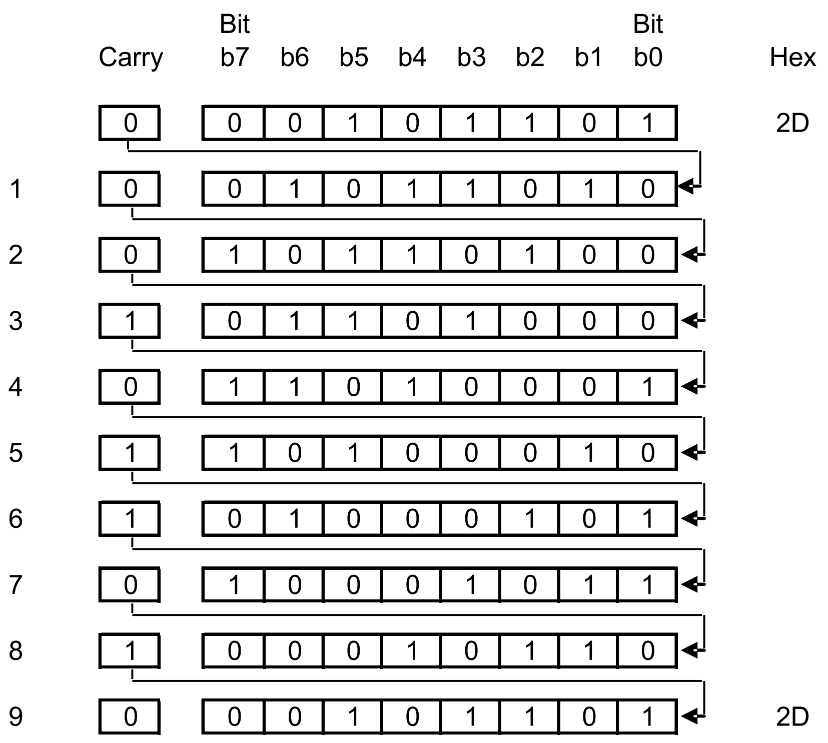


Figure 28 Successive left rotates

6.3 Instruction Codes

The Accumulator instruction mnemonics for the simple processor are chosen as shown in Table 13. The instructions do not include an operand.

Inst	**Code**	**Description**
SHR	60	Shift Accumulator Right
SHL	61	Shift Accumulator Left
ROR	62	Rotate Accumulator Right
ROL	63	Rotate Accumulator Left

Table 13 Accumulator Instructions

7 Branching and Programming

A key feature of the processor is its ability to make decisions based upon data. This really means the program path through the instructions is altered according to data. This is known as “branching”. There are many ways to achieve this based upon conditions within the processor, or by external hardware (electronic) signals. Two internal means are described here, both of which are commonly used by commercial processors.

Branch processing occurs through the issue of a Branch instruction. There are two types of branch: Unconditional and Conditional.

The Unconditional branch causes the processor program path to jump to the address supplied with the instruction. This is described in Appendix C and is useful where coding in memory requires the program path to loop or bypass data, for example.

The Conditional branch instructions also supply a memory address. However the processor program path only jumps to the address if a condition is met. If the condition is not met, then the program path is not altered and the instruction following the branch instruction is fetched.

In the simple processor the address given within the instruction is loaded into the Address Register. If the condition is met, the content of the Address Register is placed in the Program Counter and the jump occurs on the next fetch. If the condition is not met, then the Address Register is not placed in the Program Counter and the instruction following the branch in memory is fetched.

The branch conditions and the instruction mnemonics for the simple processor are shown in Table 14 and are based upon the status of the two flags associated with the ALU and the Accumulator. The instructions include an operand.

Inst	**Code**	**Description**
BCS	A0	Branch if Carry flag is set
BCN	A1	Branch if Carry flag is not set
BEZ	A2	Branch if Zero flag is set
BNZ	A3	Branch if Zero flag is not set
BRA	A4	Branch unconditionally

Table 14 Branch Instructions

The simple processor described in this book is now represented in its complete form in Figure 29.

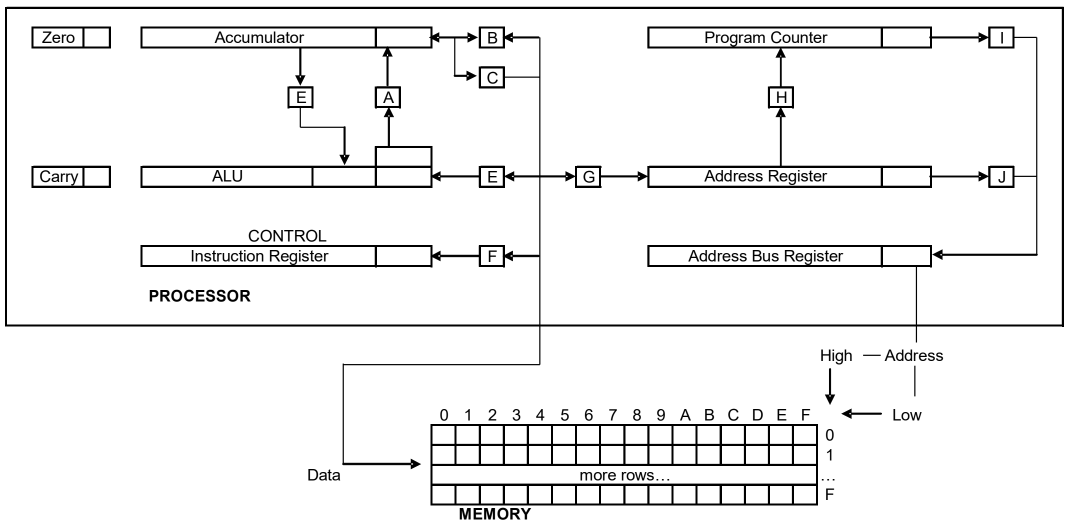


Figure 29 The complete simple processor

Gate H is described in the Table 15.

H	Data from the Address Register is placed in the Program Counter: • For a conditional branch ONLY if the branch condition is met. • Always for an unconditional branch.

Table 15 Gate H Function

7.1 A Simple Branch

The following simple example program shows how conditional branching works in the processor.[2]

The program task is

Starting with the number “2”
Subtract “1” from the number
If the result is not zero, then go back to the subtract line above
Halt the program

One thing to notice with branch instructions is that the complexity of programming has increased due to the need to know where in memory the program is jumping to. Reference names or program structures are used in real programming and Book 2 Part 1 describes a simple convention example used by the emulator. Here the branch address has to be given explicitly. This can be derived by writing out the instructions in a table with memory address references which can be seen directly. See Table 16. The program starts from address 00. The microcode for each instruction is given in Appendix C.

[2] The program is available in the emulator Book 2.

For the subtraction the Carry flag is set before the subtraction instruction indicating that the number to be subtracted at the start is positive. (See section 5.2. The full reasoning is described in Part 2).

Memory Address	**Instruction**	**Code**	**Description**
00	LDI	10	Load Accumulator, Immediate data
01	02	02	Operand for LDI: Immediate data 02
02	SEC	81	Set the Carry flag
03	SBI	49	Subtract with Carry, Immediate
04	01	01	Operand for SBI: Subtract 1
05	BNZ	A3	Branch if Not Zero to Address
06	03	03	Operand for BNZ: Branch to address 03
07	HLT	00	Halt

Table 16 Simple Branch program

The program is placed in memory.

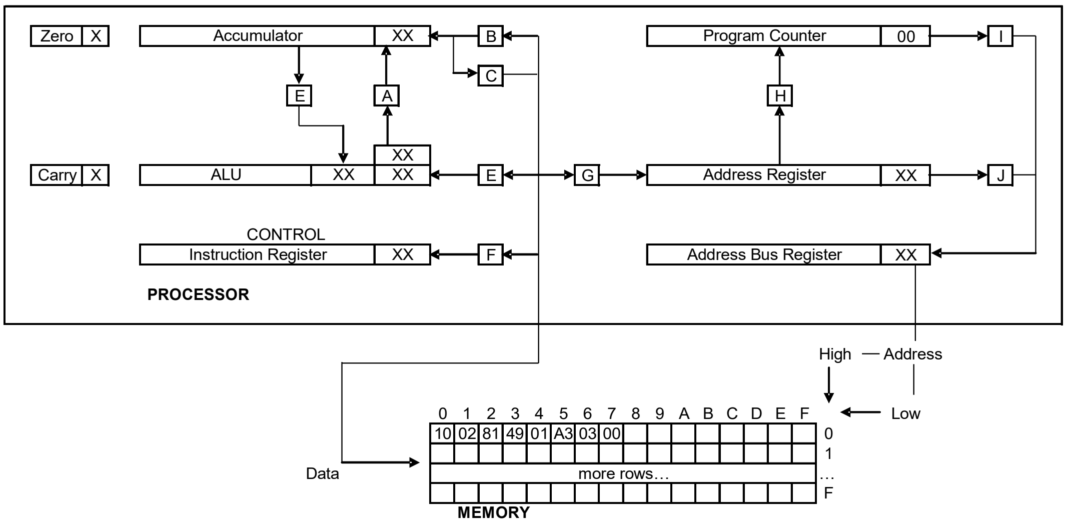


Figure 30 Branch Program - Reset State

The processor is in its reset state with the program about to start. All the Gates are closed and the PC is set to 00. Figure 30 shows the initial state.

The program starts.

Cycle 1 – Fetch LDI

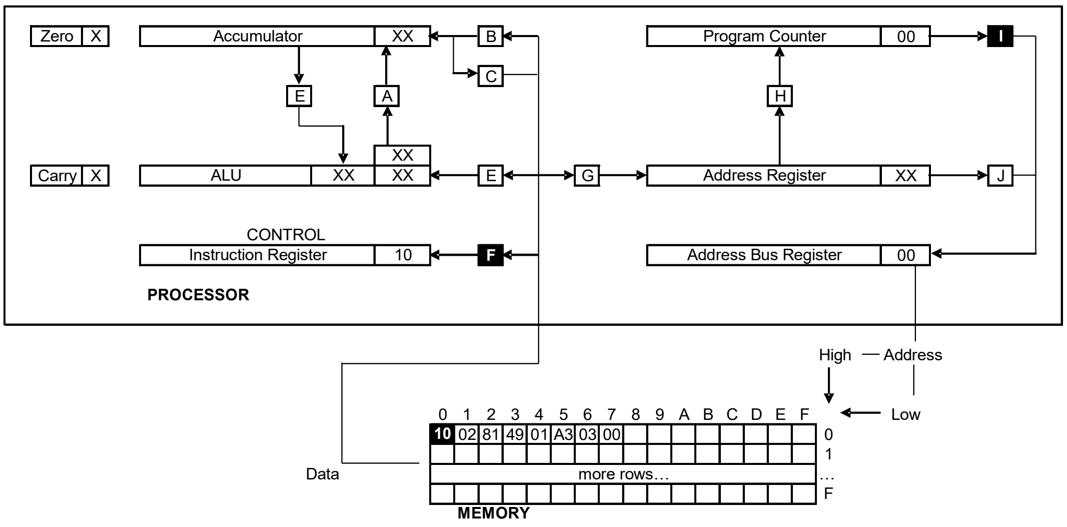


Figure 31 Branch Program - Fetch LDA

The Fetch cycle loads the first instruction into the Instruction Register (Figure 31). This is the load Accumulator with immediate data instruction.

Cycle 2 – Execute LDI

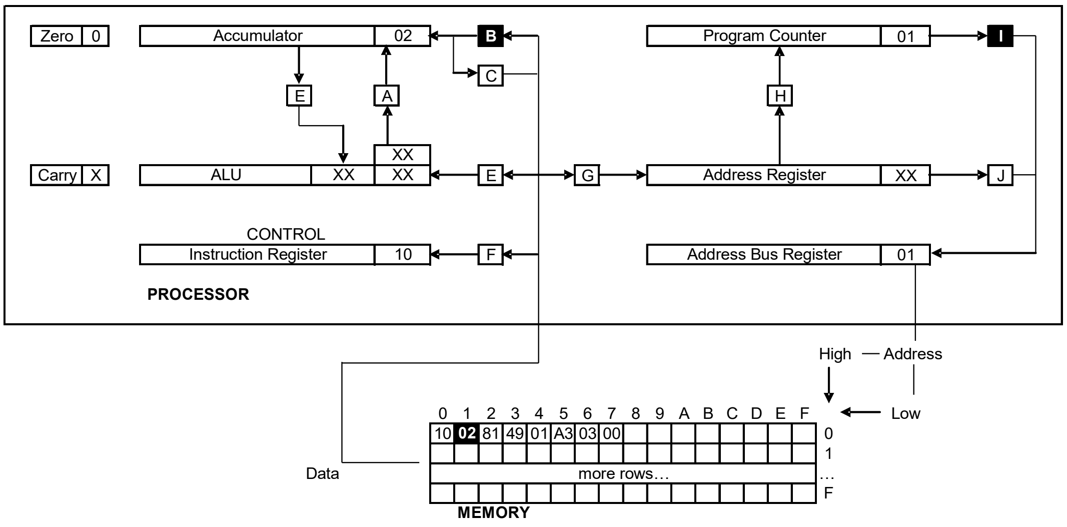


Figure 32 LDA Execute 1 - Load Address

The Execute cycle loads the Address Register with immediate data (Figure 32). The data is not zero and the Zero flag is “0”.

Cycle 3 – Fetch SEC

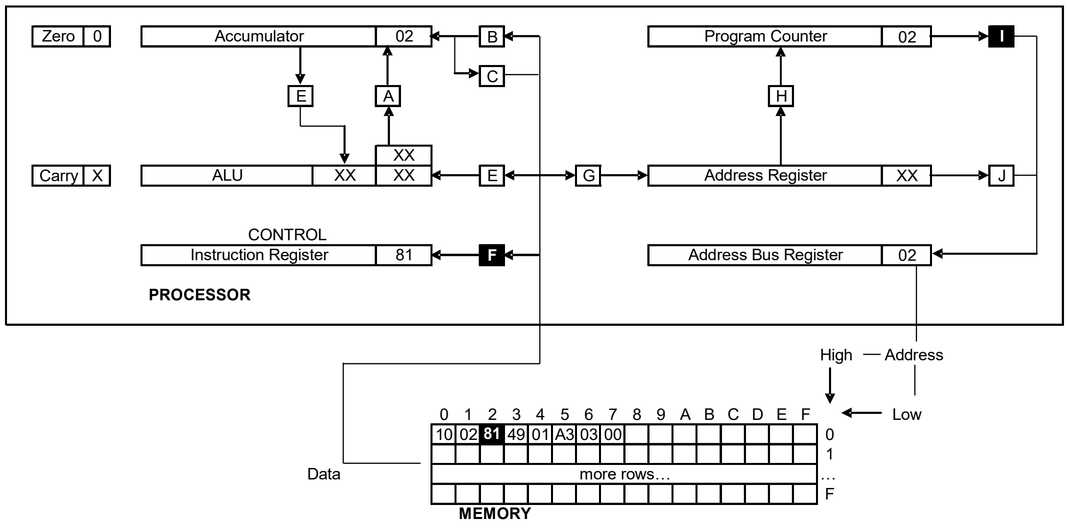


Figure 33 Branch Program - Fetch SEC

The next instruction is fetched (Figure 33). Since a subtraction is to be performed for the first time and the number to be subtracted is positive the Carry flag is set. The instruction has no operand.

Cycle 4 – Execute SEC

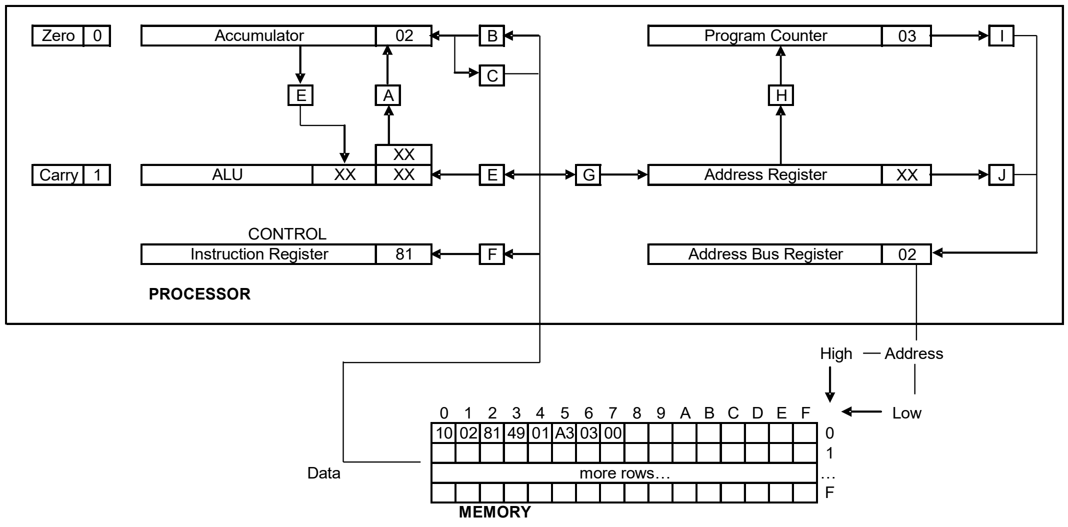


Figure 34 Execute SEC

The Set Carry instruction is executed (Figure 34). The operation is internal to the processor and no memory is used. The Execute cycle is complete.

Cycle 5 – Fetch SBI

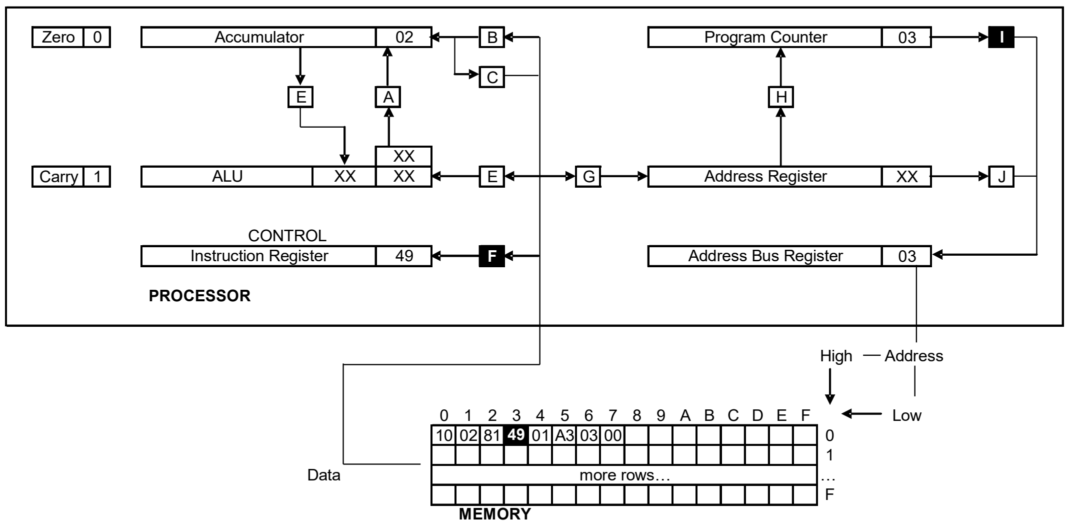


Figure 35 Branch Program - Fetch SBI

The next instruction is fetched (Figure 35).The instruction invokes a subtract using immediate data.

Cycle 6 – Execute SBI 1

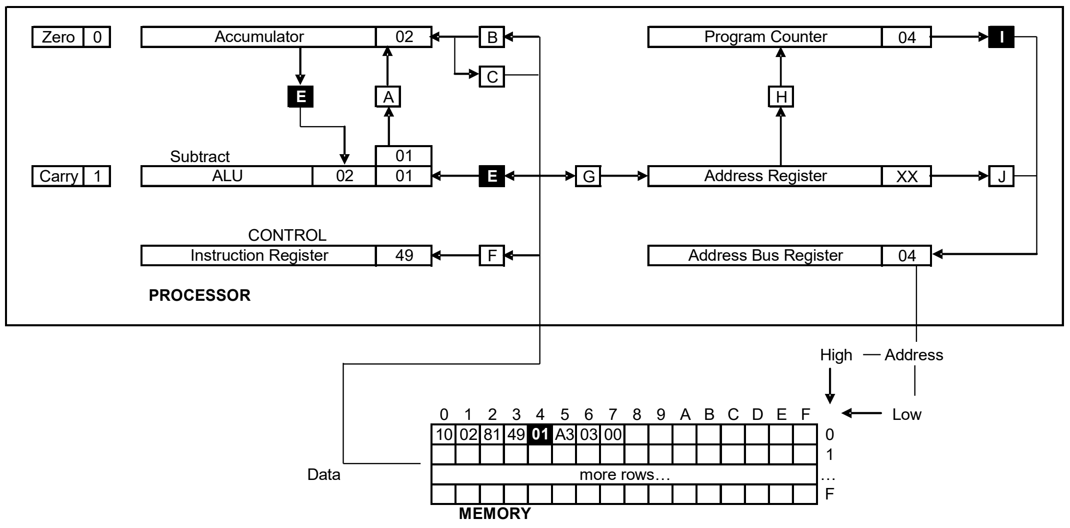


Figure 36 Execute SBI 1 - Read data to ALU

During the first Execute cycle the Accumulator and immediate data are sent to the ALU and subtracted. The answer is positive and Carry remains set (Figure 36).

Cycle 7 – Execute SBI 2

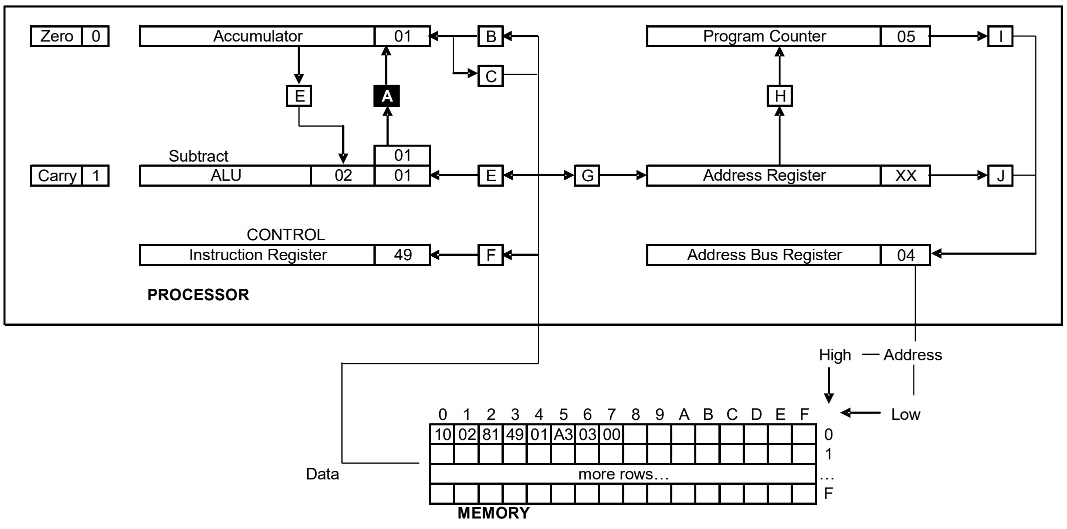


Figure 37 Execute SBI 2 – ALU result to Accumulator

The second Execute cycle provides the answer to the Accumulator and the PC is incremented ready to fetch the next instruction. The answer is not zero and the Zero flag remains “0”. The operation is internal to the processor and memory is not used (Figure 37).

Cycle 8 – Fetch BNZ

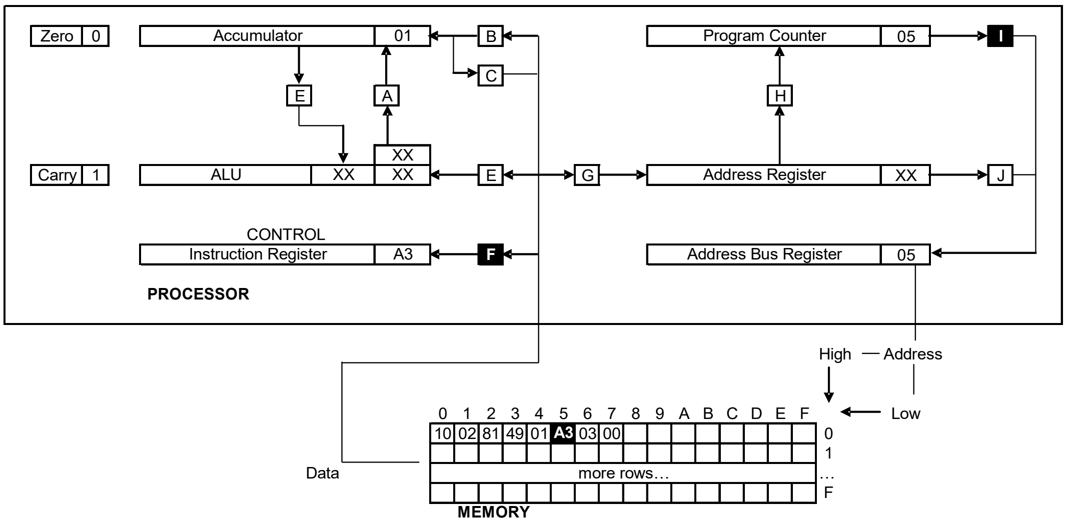


Figure 38 Branch Program - Fetch BNZ

The next instruction is fetched (Figure 38). The instruction loaded is “Branch if Not Zero”. The branch address follows the instruction.

Cycle 9 – Execute BNZ 1

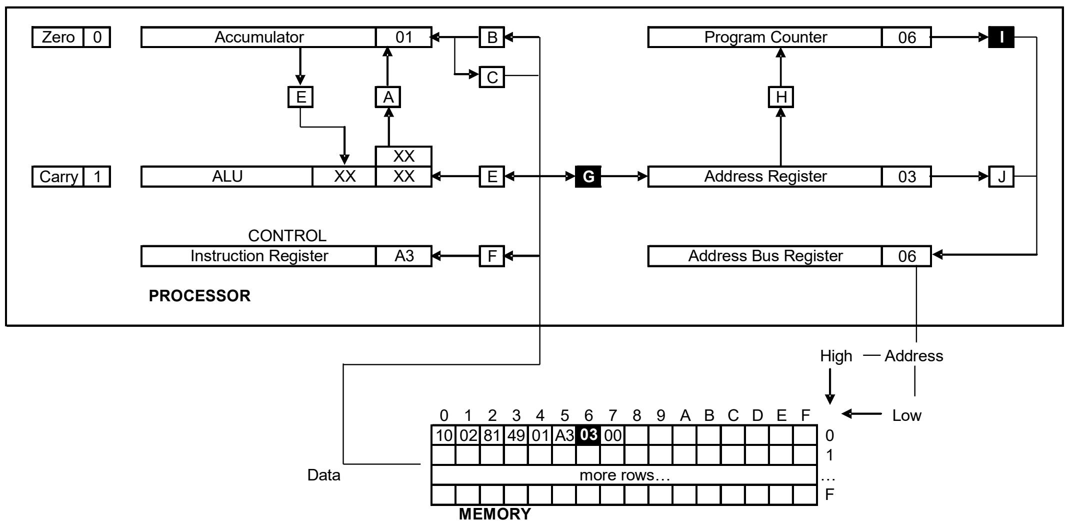


Figure 39 Execute BNZ 1 - Load branch address

During the first Execute cycle the branch address is loaded into the Address Register (Figure 39).

Cycle 10 – Execute BNZ 2

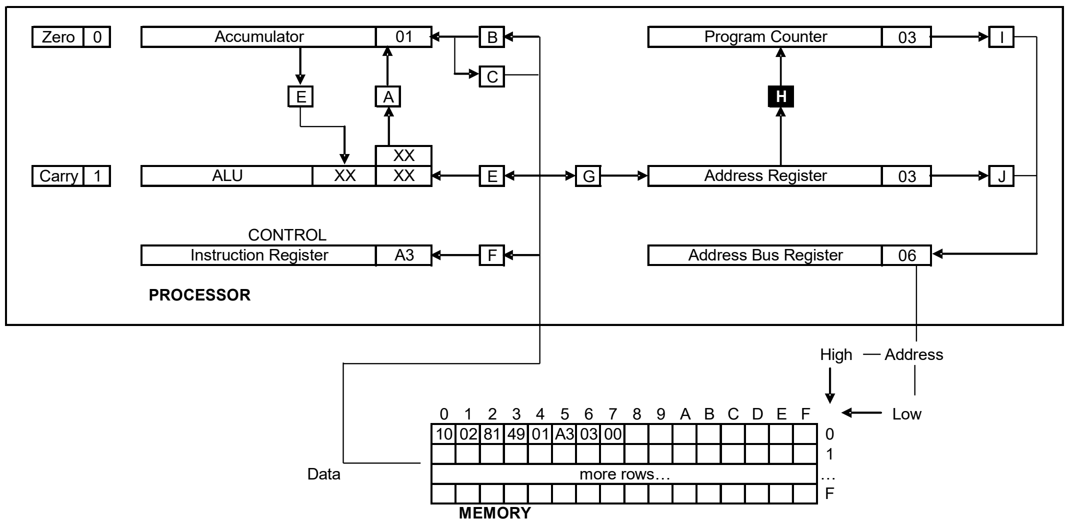


Figure 40 Execute BNZ 2 – Update PC on conditional

The second Execute cycle controls Gate H and opens the Gate if the Zero flag is not set (i.e. the Accumulator is not zero). The Gate is opened since the Accumulator value is not zero and the PC is updated from the Address Register (Figure 40). The instruction is completed and the next Fetch occurs from the updated PC address.

Cycle 11 – Fetch SBI

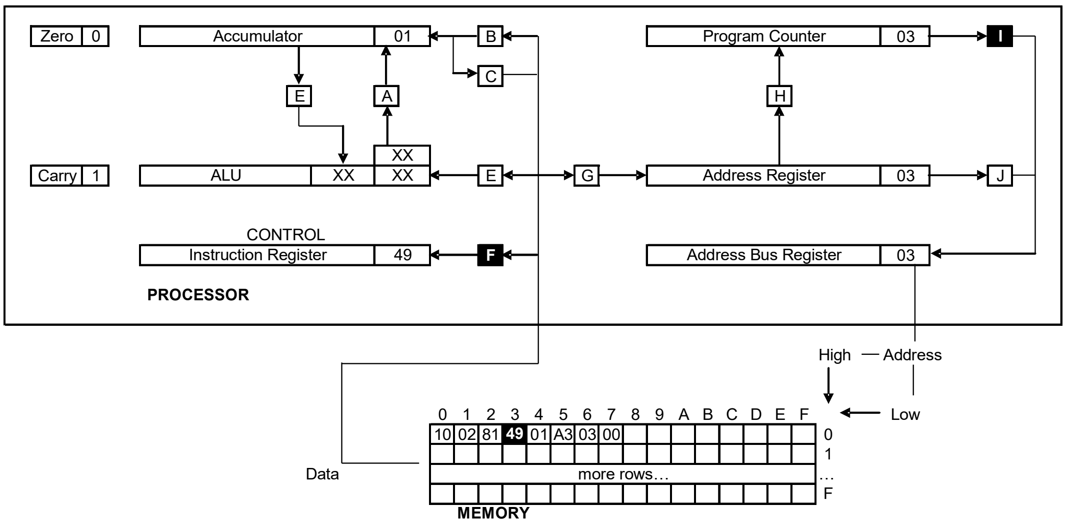


Figure 41 Branch Program - Fetch SBI again

The program loops back to reload the subtract instruction (Figure 41)

Cycle 12 – Execute SBI 1

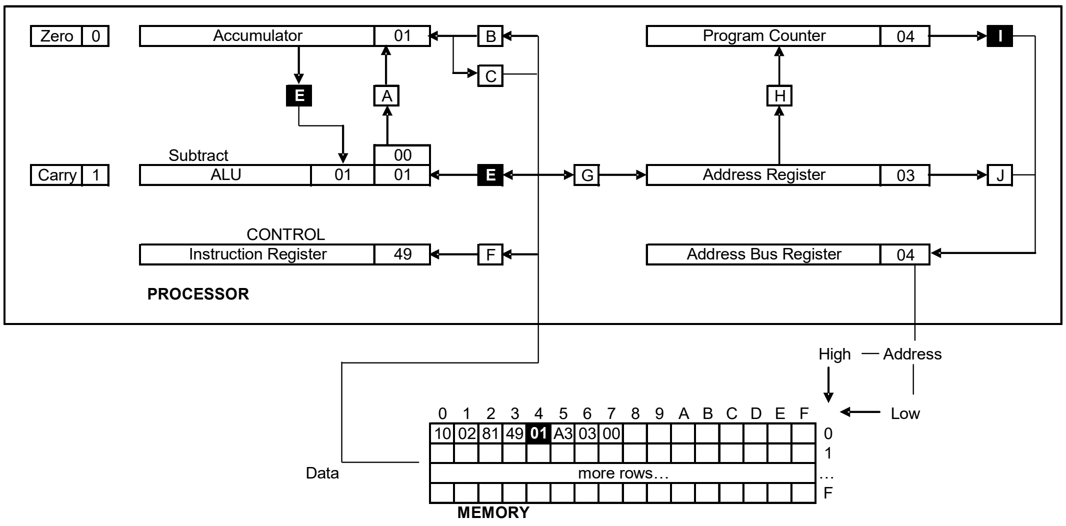


Figure 42 Execute SBI again 1 - Read data to ALU

During the first Execute cycle, the Accumulator and immediate data are sent to the ALU and subtracted. No borrow is required and Carry remains set (Figure 42).

Cycle 13 – Execute SBI 2

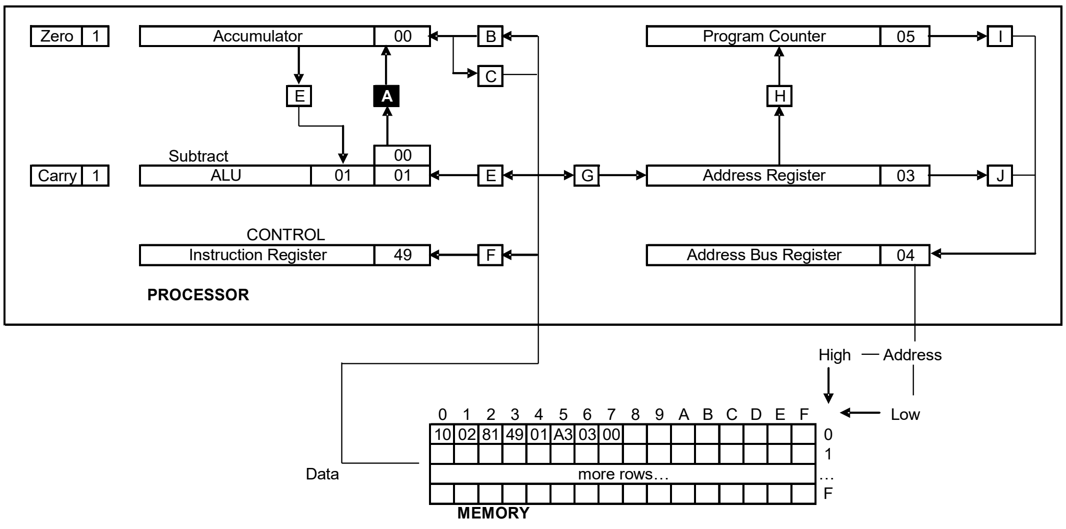


Figure 43 Execute SBI again 2 – ALU result to Accumulator

The second Execute cycle provides the answer to the Accumulator and the PC is incremented ready to fetch the next instruction. The answer is zero and the Zero flag is set to "1". The operation is internal to the processor and memory is not used (Figure 43).

Cycle 14 – Fetch BNZ

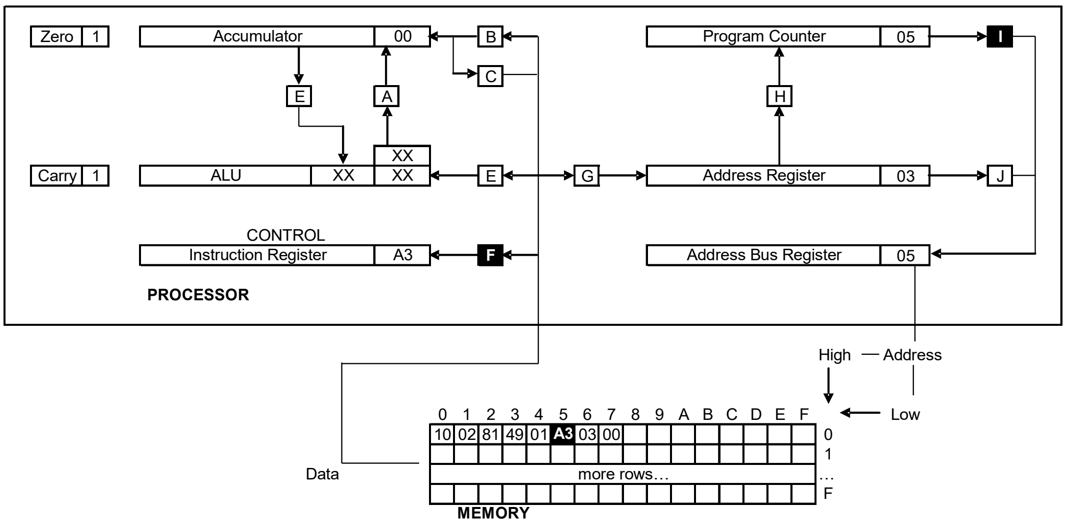


Figure 44 Branch Program - Fetch BNZ again

The instruction loaded is "Branch if Not Zero". The branch address follows the instruction (Figure 44).

Cycle 15 – Execute BNZ 1

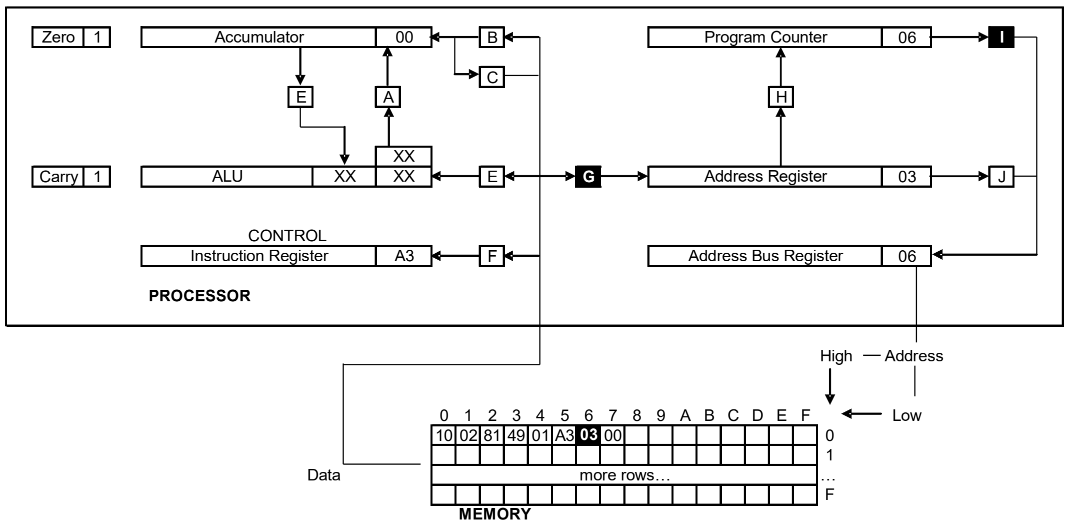


Figure 45 Execute BNZ again 1 - Load branch address

During the first Execute cycle the branch address is loaded into the Address Register (Figure 45).

Cycle 16 – Execute BNZ 2

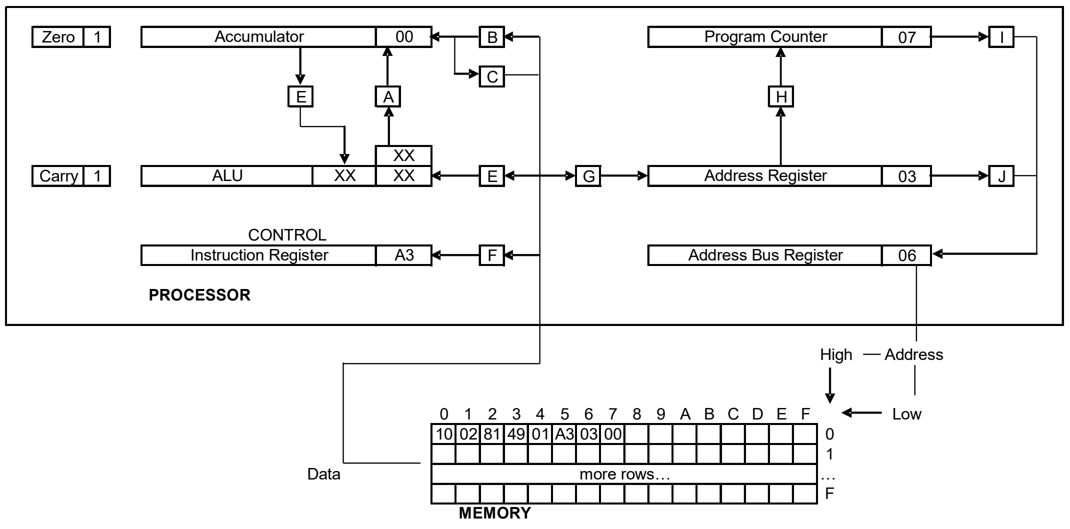


Figure 46 Execute BNZ again 2 – Update PC on conditional

The second Execute cycle controls Gate H and opens the Gate if the Zero flag is not set (i.e. the Accumulator is not zero). The Gate is not opened since the Accumulator value is zero and the Zero flag is "1". The PC is incremented to fetch the next instruction (Figure 46). The instruction is completed and the next Fetch occurs.

Cycle 17 - HLT

The processor loads the HLT (Halt) instruction “00” and the program stops.

This simple branch example demonstrates how conditional processing based upon data and results occurs.

The ability to manipulate data and change program paths can be used to solve complex problems by breaking-down the problems into elementary calculations involving the Carry and Zero flags. The skill required is in the construction of the computer program.

7.2 A Note On Programming

Programming includes some art as well as science since there are generally a number of ways to solve a problem. Some programs will be better than others depending upon how they are measured (e.g. conserving code space, maximising speed, achieving consistency of calculation across data can all lead to different programs for the same problem).

The type of problems suited to programmed solutions include data transfer (Load/Store instructions), data manipulation (ALU and Accumulator instructions) and decision-making (Branch instructions). The simple processor provides this capability.

Programming using the instruction mnemonics of a processor is referred to as “Assembler” programming. The codes that include memory addresses use the actual byte value within the instruction. Since this is generally unknown to the programmer (e.g. branch address) a tool is used which can calculate the values from a symbolic convention which uses labels to represent the memory addresses. Book 2 Part 1 contains a suitable convention and the emulator includes the tool. Book 1 Part 2 and Book 3 Part 1 include some sample programs which are also included in the emulator.

Problems need to be analysed logically so that a suitable program can be constructed. One technique is to represent the process in a flow diagram and examples are included in Book 3 Part 1. Programming is of course a complete study itself.

Although some assembler programming is still used where performance or memory size is critical, programming is normally practiced using software. Software removes the need to consider most of the processor machines characteristics. A compiler tool converts software into Machine Code routines that deliver the required functionality.

8 Concluding Remarks

It can be seen from the foregoing that the computer processor is little more than a programmable calculating machine which is also capable of handling and storing information. The speed with which a real machine can perform the tasks is immense (millions of instructions per second) and is quite capable of providing what appears to be a comprehending face to the world. However, the face is illusory and is simply a measure of the software engineer's ingenuity. Artificial Intelligence is exactly what it says: Artificial. If the instructions go wrong the machine will malfunction, although even this can be detected and systems put into play which may recover the situation.

The simple processor described does not possess any input/output facility. Of course this is how most people interact with their devices. Each facility (keyboard, display, sound system, network controller etc) may contain its own processor which communicates directly to a main processor. Information on a screen display is effectively a write memory where the data passed induces a particular display. The network controller allows the system to communicate with other systems (LAN, internet etc) for the exchange of data.

Another major characteristic of practical devices such as smartphones is that they already contain a program known as an "operating system" which provides a user-interface from power-up. The system provides the environment which supports the applications ("Apps") required by users.

In summary, the simple processor provides a platform upon which more complex devices could be designed. This can be achieved by adding more registers and additional functionality. Additional instructions can be provided (with corresponding microcode). Part 2 describes some of the additional features of practical processors, along with an introduction to electronic logic and a description of the processor control system used in Book 3.

A. Appendix: Binary Numbers

An appreciation of the binary system of numbers is best approached through a deeper understanding of the usual way in which numbers are written. That is, the decimal system.

The numbers fifteen, thirty-seven, one hundred and fifty-six and five hundred and seventy-four are represented as 15, 37, 156 and 574 respectively. What is actually meant by the numbers is the following:

	Hundreds	Tens	Units
15		1	5
37		3	7
156	1	5	6
574	5	7	4

Thousands are added when numbers increase beyond 999. Each additional digit is a power of ten greater than the previous digit and is known as "base 10". The number of tens, hundreds etc can be represented by one of the ten numbers (0 to 9) available in decimal. What is happening here is the units, tens and hundreds are individually multiplied by the number assigned as follows:

	(1 x 10)	+ 5	=	15
	(3 x 10)	+ 7	=	37
(1 x 100)	+ (5 x 10)	+ 6	=	156
(5 x 100)	+ (7 x 10)	+ 4	=	574

Binary works in the same way except only two numbers (0 and 1) are available. The powers increase as powers of two (i.e. "base 2"). So the numbers above can be obtained as follows:

	512	256	128	64	32	16	8	4	2	1
15							1	1	1	1
37					1	0	0	1	0	1
156			1	0	0	1	1	1	0	0
574	1	0	0	0	1	1	1	1	1	0

The power of two containing a one is added to the total but entries of zero are ignored (since multiplying by zero results in zero). The binary representations are therefore as follows

15	1111
37	100101
156	10011100
574	1000111110

When adding two decimal numbers individual digits may overflow the number base (10) during the summation. For example, adding seven to 14 results in the units overflowing. The overflow is carried into the tens column as one to be added to the tens addition. This can be written as follows:

$$\begin{array}{rcc} & \text{Tens} & \text{Units} \\ & 1 & 4 \\ + & & 7 \\ \hline \text{Carry} & 1 & \\ \text{Sum} & 2 & 1 \\ \hline \end{array}$$

The binary version works in the same way except the overflow occurs when two is reached rather than 10 and can be written as follows.

$$\begin{array}{rccccc} & 16 & 8 & 4 & 2 & 1 \\ & & 1 & 1 & 1 & 0 \\ + & & & 1 & 1 & 1 \\ \hline \text{Carry} & 1 & 1 & 1 & & \\ \text{Sum} & 1 & 0 & 1 & 0 & 1 \\ \hline \end{array}$$

It can be seen that the rules for binary addition are as follows.

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 and 1 to carry

When there is already one to carry 1 + 1 + Carry = 1 with 1 to carry as occurs above for the binary bit “4”.

B. Appendix: Hexadecimal Numbers

Hex is used here as a convenient way to represent binary data in bytes. The two four-bit groups are written as two hex digits. The digits also represent the numbers in a system where each additional digit increases by a power of 16 i.e. “base 16”.

Appendix A contains examples of arithmetic using decimal (base 10) and binary (base 2) numbers. This Appendix introduces hexadecimal arithmetic.

When converting the byte hex value to decimal the digits can be regarded as base 16 numbers. Therefore, the conversion of the byte values to decimal shown in Figure 5 do not need to be reduced to the binary form but can be converted directly as follows.

```
4B  =  (4 x 16)    + 11  =  75
CC =  (12 x 16)   + 12  =  204
05  =   (0 x 16)   + 5    =  5
```

When adding numbers the individual digit additions overflow when the number base is exceeded. The same is true for hexadecimal where the number base is 16. Examples follow, with each column a power of 16 greater than the previous:

	256	16	1
		2	F
		0	5
Carry		1	
Sum		3	4

	256	16	1
		C	C
		4	9
Carry	1	1	
Sum	1	1	5

C. Appendix: Instruction Set for the Simple Processor

The full instruction set for the simple processor described in these books is shown in the table. Some of the instructions have not yet been introduced and all instructions are described fully in the following.

Many of the instructions shown in the following include the microcode “Int Op”. This signifies “Internal Operation” and in the design of the simple processor (Book 3) prevents external memory being accessed since no memory operation is required. Increment PC is represented by “PC=PC+1”.

Inst	Code	Operand	MC	Task
HLT	00	N	2	Halt
LDI	10	Y	3	Load Accumulator, Immediate
LDA	11	Y	3	Load Accumulator
LDX	13	Y	5	Load Accumulator, Indirectly
STA	20	Y	3	Store Accumulator
STX	22	Y	5	Store Accumulator, Indirectly
ADC	40	Y	4	Add with Carry
SBC	41	Y	4	Subtract with Carry
XOR	45	Y	4	Exclusive Or
ORA	46	Y	4	Or
AND	47	Y	4	And
ADI	48	Y	3	Add with Carry, Immediate
SBI	49	Y	3	Subtract with Carry, Immediate
XOI	4D	Y	3	Exclusive Or, Immediate
ORI	4E	Y	3	Or, Immediate
ANI	4F	Y	3	And, Immediate
SHR	60	N	2	Shift Right
SHL	61	N	2	Shift Left
ROR	62	N	2	Rotate Right
ROL	63	N	2	Rotate Left
CLC	80	N	2	Clear Carry
SEC	81	N	2	Set Carry
BCS	A0	Y	3	Branch if Carry Set
BCN	A1	Y	3	Branch if Carry Not Set
BEZ	A2	Y	3	Branch if Acc Zero
BNZ	A3	Y	3	Branch if Acc Not Zero
BRA	A4	Y	3	Branch unconditionally

Table 17 Simple Processor Instruction Set

C.1 HLT – HALT

Description:

A HLT instruction stops the processor proceeding through memory and attempting to load more instructions. The Program Counter is prevented from incrementing during the Fetch state (a unique feature for the instruction). No activity takes place during Execute.

Instruction:

Inst	**Code**	**Operand**	**Description**
HLT	00	N	Halt processor

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
HLT	2	I, F		▨	▨	▨

Execute cycle:

No Gates or other activities become active during the Execute cycle. On completion, the processor reverts to the Fetch cycle with the Program Counter unchanged and the processor repeats the Fetch cycle on the HLT instruction indefinitely or until the processor is reset by the electronics.

C.2 LDI – LOAD IMMEDIATE

Description:

Load Immediate results in the instruction operand being placed in the Accumulator after the single Execute cycle.

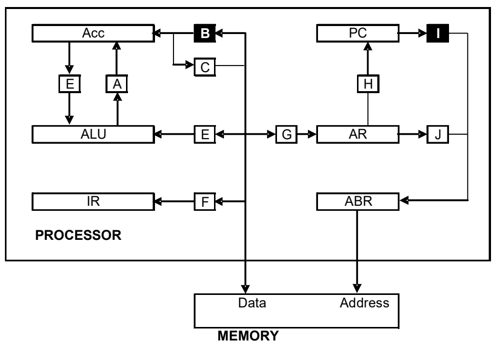
Instruction:

Inst	Code	Operand	Description
LDI	10	Y	Load Accumulator, Immediate

Microcode:

Inst	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
LDI	2	I, F PC=PC+1	I, B PC=PC+1			

Execute cycle:

 Acc B PC I C E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and B • Read memory (Instruction operand is written into Accumulator) • Close Gates • Increment Program Counter

C.3 LDA – LOAD ACCUMULATOR FROM ADDRESS

Description:

Load Accumulator from address results in the data in memory at the address given in the instruction operand being placed in the Accumulator after the second Execute cycle.

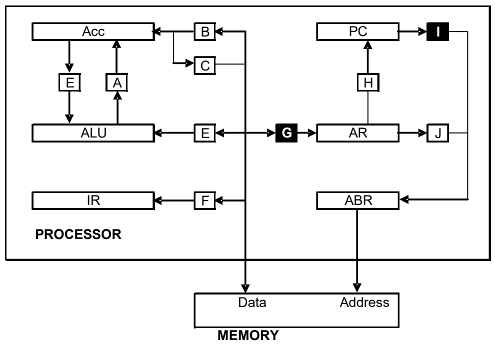
Instruction:

Inst	**Code**	**Operand**	**Description**
LDA	11	Y	Load Accumulator from Address

Microcode:

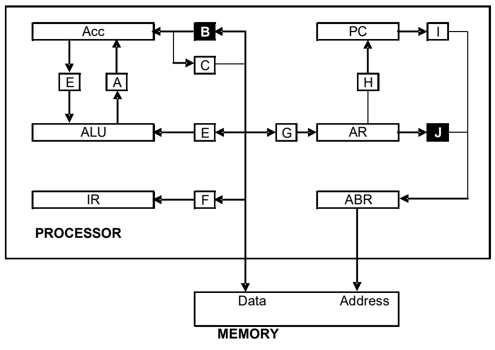
Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
LDA	3	I, F PC=PC+1	I, G PC=PC+1	J, B	▨	▨

Execute cycles:



Execute 1

- Open Gates I and G
- Read memory (Instruction operand is written into Address Register)
- Close Gates
- Increment Program Counter



Execute 2

- Open Gates J and B
- Read memory (Content of addressed memory copied into Accumulator)
- Close Gates

C.4 LDX – LOAD ACCUMULATOR FROM INDIRECT ADDRESS

Description:

Load Accumulator from indirect address results in the data loaded from the memory address given by the instruction operand is in turn used as an address to the data to be placed in the Accumulator. (Indirect addressing is particularly useful where the address to data is calculated e.g reading each byte in a message consisting of multiple bytes in memory).

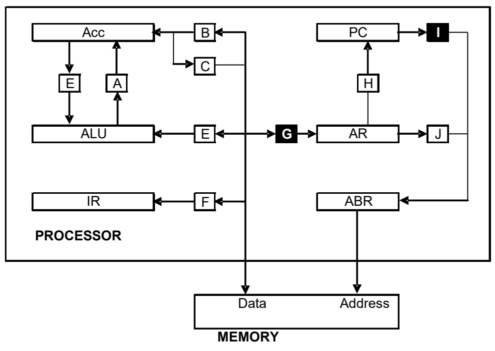
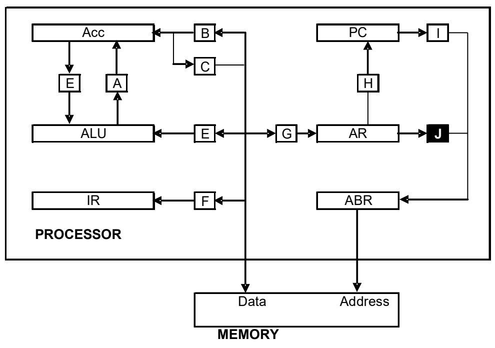
Instruction:

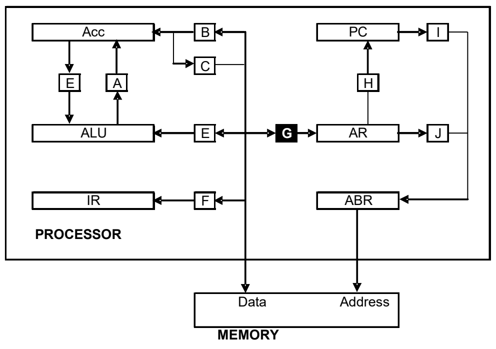
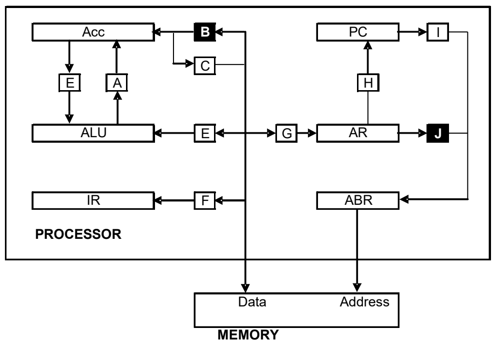
Inst	Code	Operand	Description
LDX	13	Y	Load Accumulator from Indirect Address

Microcode:

Inst	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
LDX	5	I, F PC=PC+1	I, G PC=PC+1	J	G	J, B

Execute cycles:

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and G • Read memory (Instruction operand is written into Address Register) • Close Gates • Increment Program Counter
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Open Gate J • Address Register copied into Address Bus Register • Close Gate

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 3 • Open Gate G • Read memory (Address Bus Register retains address from Execute 2 and content of addressed memory copied into Address Register) • Close Gates
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 4 • Open Gates J and B • Read memory (Content of addressed memory copied into Accumulator) • Close Gates

C.5 STA – STORE ACCUMULATOR TO ADDRESS

Description:

Store Accumulator places the data in the Accumulator in the memory at the address given by the operand.

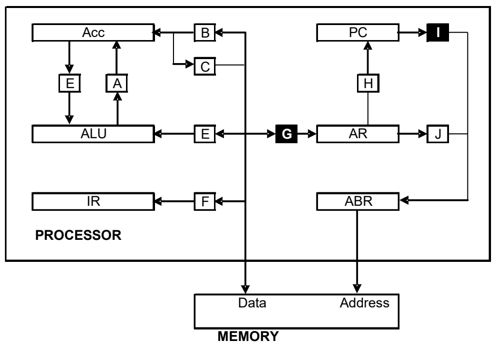
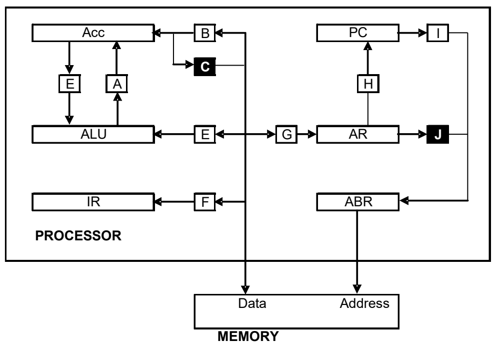
Instruction:

Inst	Code	Operand	Description
STA	20	Y	Store Accumulator to memory

Microcode:

Inst	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
STA	3	I, F PC=PC+1	I, G PC=PC+1	J, C		

Execute cycles:

 Acc B C E A ALU E G AR J PC I H IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and G • Read memory (Instruction operand is written into Address Register) • Close Gates • Increment Program Counter
 Acc B C E A ALU E G AR J PC I H IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Open Gates J and C • Write memory (Content of Accumulator copied into addressed memory) • Close Gates

C.6 STX – STORE ACCUMULATOR TO INDIRECT ADDRESS

Description:

Store Accumulator Indirect places the data in the Accumulator in the memory at the address given by the data in the memory location pointed to by the operand. (Indirect addressing is particularly useful where the address to data is calculated e.g writing a message over multiple bytes in memory).

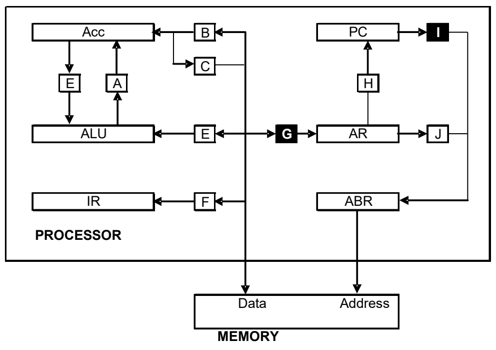
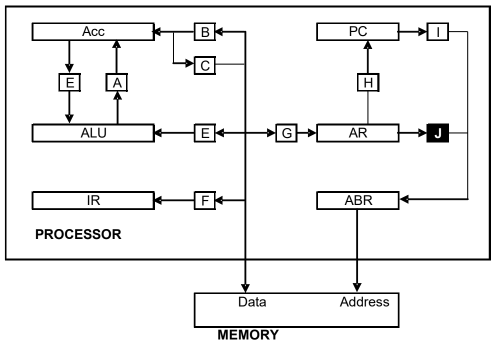
Instruction:

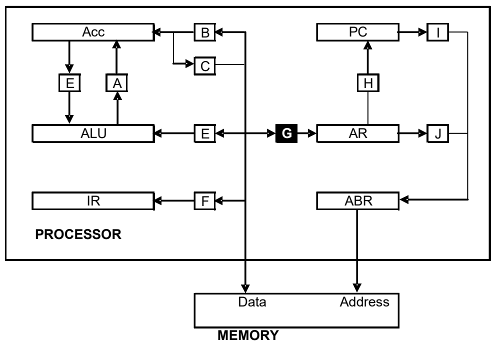
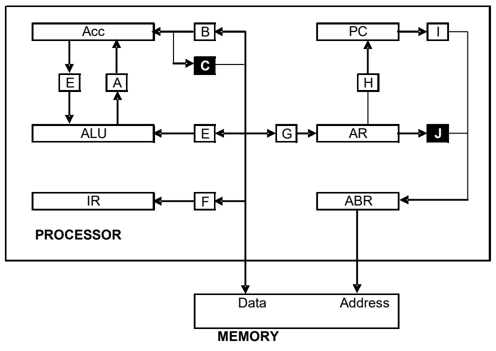
Inst	**Code**	**Operand**	**Description**
STX	22	Y	Store Accumulator to memory indirectly

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
STX	5	I, F PC=PC+1	I, G PC=PC+1	J	G	J, C

Execute cycles:

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and G • Read memory (Instruction operand is written into Address Register) • Close Gates • Increment Program Counter
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Open Gate J • Address Register copied into Address Bus Register • Close Gate

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 3 • Open Gate G • Read memory (Address Bus Register retains address from Execute 2 and content of addressed memory copied into Address Register) • Close Gate
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 4 • Open Gates J and C • Write memory (Content of Accumulator copied into addressed memory) • Close Gates

C.7 ADC – ADD WITH CARRY FROM ADDRESS

Description:

The instruction adds the content of the Accumulator, Carry and the data read from the operand memory address. The result is placed into the Accumulator and the Carry flag is updated.

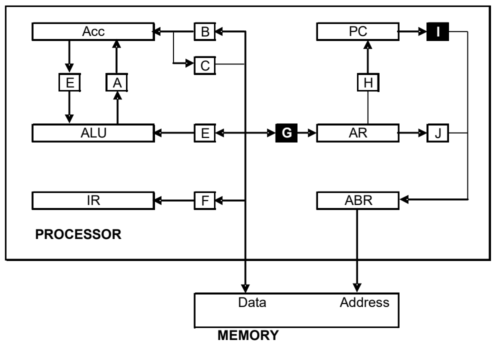
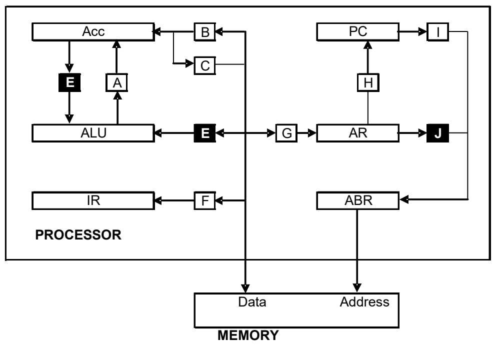
Instruction:

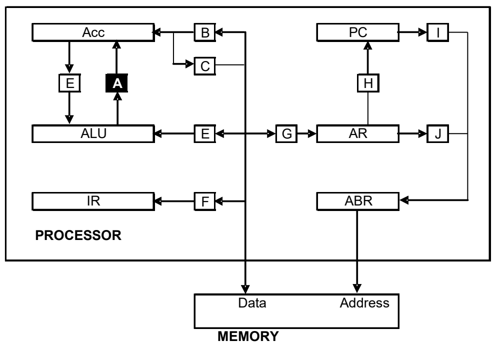
Inst	**Code**	**Operand**	**Description**
ADC	40	Y	Add with Carry

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
ADC	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	▨

Execute cycles:

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and G • Read memory (Instruction operand is written into Address Register) • Close Gates • Increment Program Counter
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Set ALU for Addition • Open Gates J and E • Read memory (Addressed memory and Accumulator data sent to ALU) • Close Gates

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 3 • Open Gate A (ALU result is sent to Accumulator) • Close Gate

C.8 SBC – SUBTRACT WITH CARRY FROM ADDRESS

Description:

The instruction subtracts with Carry the data read from the operand memory address from the content of the Accumulator. The result is placed into the Accumulator and the Carry flag is updated.

Instruction:

Inst	**Code**	**Operand**	**Description**
SBC	41	Y	Subtract with Carry

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
SBC	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	▨

Execute cycles:

Gate operation is as per ADC instruction. Only change to execution is to Execute 2 where ALU is set for Subtraction instead of Addition.

C.9 XOR – EXCLUSIVE OR FROM ADDRESS

Description:

Each bit in the Accumulator is processed in a logical Exclusive OR with its corresponding bit in the memory byte addressed by the operand and the result placed into the Accumulator.

Instruction:

Inst	**Code**	**Operand**	**Description**
XOR	45	Y	Logical Exclusive OR

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
XOR	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	▨

Execute cycles:

Gate operation is as per ADC instruction. Only change to execution is to Execute 2 where ALU is set for Logical Exclusive OR instead of Addition.

C.10 ORA – OR FROM ADDRESS

Description:

Each bit in the Accumulator is processed in a logical OR with its corresponding bit in the memory byte addressed by the operand and the result placed into the Accumulator.

Instruction:

Inst	**Code**	**Operand**	**Description**
ORA	46	Y	Logical OR

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
ORA	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	▨

Execute cycles:

Gate operation is as per ADC instruction. Only change to execution is to Execute 2 where ALU is set for Logical OR instead of Addition.

C.11 AND – AND FROM ADDRESS

Description:

Each bit in the Accumulator is processed in a logical AND with its corresponding bit in the memory byte addressed by the operand and the result placed into the Accumulator.

Instruction:

Inst	**Code**	**Operand**	**Description**
AND	47	Y	Logical AND

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
AND	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	▨

Execute cycles:

Gate operation is as per ADC instruction. Only change to execution is to Execute 2 where ALU is set for Logical AND instead of Addition.

C.12 ADI – ADD WITH CARRY IMMEDIATE

Description:

The instruction adds the content of the Accumulator, Carry and the data read from the operand. The result is placed into the Accumulator and the Carry flag is updated.

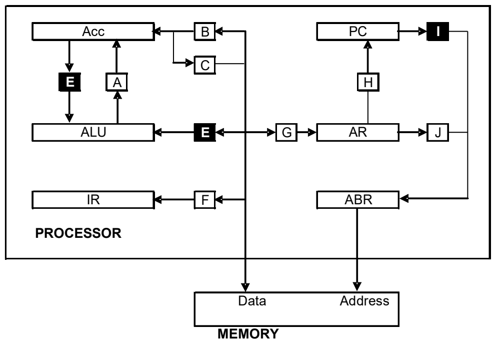
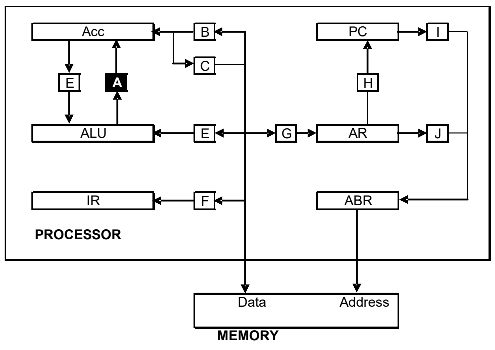
Instruction:

Inst	Code	Operand	Description
ADI	48	Y	Add with Carry, Immediate

Microcode:

Inst	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
ADI	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		

Execute cycles:

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Set ALU for Addition • Open Gates I and E • Read memory (Operand and Accumulator data sent to ALU) • Close Gates
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Open Gate A (ALU result is sent to Accumulator) • Close Gate

C.13 SBI – SUBTRACT WITH CARRY IMMEDIATE

Description:

The instruction subtracts with Carry the data read from the operand from the content of the Accumulator. The result is placed into the Accumulator and the Carry flag is updated.

Instruction:

Inst	**Code**	**Operand**	**Description**
SBI	49	Y	Subtract with Carry, Immediate

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
SBI	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op	▨	▨

Execute cycles:

Gate operation is as per ADI instruction. Only change to execution is to Execute 1 where ALU is set for Subtraction instead of Addition.

C.14 XOI – EXCLUSIVE OR IMMEDIATE

Description:

Each bit in the Accumulator is processed in a logical Exclusive OR with its corresponding bit in the operand and the result placed into the Accumulator.

Instruction:

Inst	**Code**	**Operand**	**Description**
XOI	4D	Y	Logical Exclusive OR, Immediate

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
XOI	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op	▨	▨

Execute cycles:

Gate operation is as per ADI instruction. Only change to execution is to Execute 1 where ALU is set for Logical Exclusive OR instead of Addition.

C.15 ORI – OR IMMEDIATE

Description:

Each bit in the Accumulator is processed in a logical OR with its corresponding bit in the operand and the result placed into the Accumulator.

Instruction:

Inst	**Code**	**Operand**	**Description**
ORI	4E	Y	Logical OR, Immediate

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
ORI	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op	▨	▨

Execute cycles:

Gate operation is as per ADI instruction. Only change to execution is to Execute 1 where ALU is set for Logical OR instead of Addition.

C.16 ANI – AND IMMEDIATE

Description:

Each bit in the Accumulator is processed in a logical AND with its corresponding bit in the operand and the result placed into the Accumulator.

Instruction:

Inst	**Code**	**Operand**	**Description**
ANI	4F	Y	Logical AND, Immediate

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
ANI	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op	▨	▨

Execute cycles:

Gate operation is as per ADI instruction. Only change to execution is to Execute 1 where ALU is set for Logical AND instead of Addition.

C.17 SHR – SHIFT ACCUMULATOR RIGHT

Description:

Each bit value in the Accumulator is shifted to the next lower significant bit. The Least Significant Bit value is moved to the Carry flag. The Most Significant Bit value is set to zero.

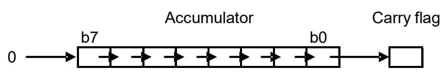
Instruction:

Inst	**Code**	**Operand**	**Description**
SHR	60	N	Shift Accumulator bits right

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
SHR	2	I, F PC=PC+1	Int Op	▨	▨	▨

Execute cycle:

	Execute 1 • Internal operation (no memory access) • Accumulator operation: ○ Carry flag set to b0 ○ b0 set to b1 ○ b1 set to b2 ○ b2 set to b3 ○ b3 set to b4 ○ b4 set to b5 ○ b5 set to b6 ○ b6 set to b7 ○ b7 set to 0

C.18 SHL – SHIFT ACCUMULATOR LEFT

Description:

Each bit value in the Accumulator is shifted to the next higher significant bit. The Most Significant Bit value is moved to the Carry flag. The Least Significant Bit value is set to zero.

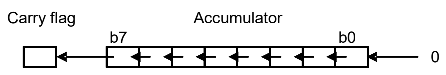
Instruction:

Inst	Code	Operand	Description
SHL	61	N	Shift Accumulator bits left

Microcode:

Inst	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
SHL	2	I, F PC=PC+1	Int Op			

Execute cycle:



Execute 1

- Internal operation (no memory access)
- Accumulator operation:
    - Carry flag set to b7
    - b7 set to b6
    - b6 set to b5
    - b5 set to b4
    - b4 set to b3
    - b3 set to b2
    - b2 set to b1
    - b1 set to b0
    - b0 set to 0

C.19 ROR –ROTATE ACCUMULATOR RIGHT

Description:

Each bit value in the Accumulator is shifted to the next lower significant bit. The Most Significant Bit value is set to the value in the Carry flag. Subsequently, the Least Significant Bit value is moved to the Carry flag.

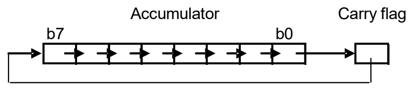
Instruction:

Inst	**Code**	**Operand**	**Description**
ROR	62	N	Rotate Accumulator bits right

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
ROR	2	I, F PC=PC+1	Int Op			

Execute cycle:

	Execute 1 • Internal operation (no memory access) • Accumulator operation: ○ b0 saved ○ b0 set to b1 ○ b1 set to b2 ○ b2 set to b3 ○ b3 set to b4 ○ b4 set to b5 ○ b5 set to b6 ○ b6 set to b7 ○ b7 set to Carry flag ○ Carry flag set to saved b0

C.20 ROL – ROTATE ACCUMULATOR LEFT

Description:

Each bit value in the Accumulator is shifted to the next higher significant bit. The Least Significant Bit value is set to the value in the Carry flag. Subsequently, the Most Significant Bit value is moved to the Carry flag.

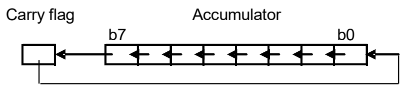
Instruction:

Inst	**Code**	**Operand**	**Description**
ROL	63	N	Rotate Accumulator bits left

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
ROL	2	I, F PC=PC+1	Int Op			

Execute cycle:

	Execute 1 • Internal operation (no memory access) • Accumulator operation: ○ b7 saved ○ b7 set to b6 ○ b6 set to b5 ○ b5 set to b4 ○ b4 set to b3 ○ b3 set to b2 ○ b2 set to b1 ○ b1 set to b0 ○ b0 set to Carry flag ○ Carry flag set to saved b7

C.21 CLC – CLEAR CARRY FLAG TO “0”

Description:

The instruction sets the Carry flag to zero.

Instruction:

Inst	**Code**	**Operand**	**Description**
CLC	80	N	Clear the Carry flag

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
CLC	2	I, F PC=PC+1	Int Op	▨	▨	▨

Execute cycle:

The Carry flag is set to 0.

C.22 SEC – SET CARRY FLAG TO “1”

Description:

The instruction sets the Carry flag to one.

Instruction:

Inst	**Code**	**Operand**	**Description**
SEC	81	N	Set the Carry flag

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
SEC	2	I, F PC=PC+1	Int Op	▨	▨	▨

Execute cycle:

The Carry flag is set to 1.

C.23 BCS – BRANCH IF CARRY FLAG SET

Description:

This is a branch instruction that sets the Program Counter to the address given in the operand, but only if the Carry flag is one.

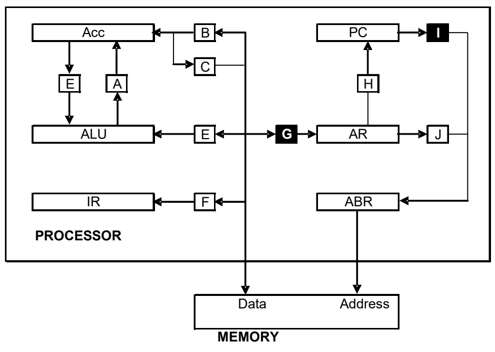
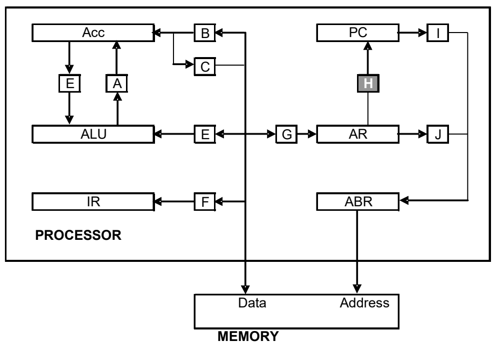
Instruction:

Inst	**Code**	**Operand**	**Description**
BCS	A0	Y	Branch if Carry flag is set

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
BCS	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		

Execute cycles:

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and G • Read memory (Instruction operand is written into Address Register) • Close Gates • Increment Program Counter
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Query: Is the Carry flag value one? • If so open Gate H (Address Register copied into Program Counter) otherwise do nothing • Close Gate

C.24 BCN – BRANCH IF CARRY FLAG NOT SET

Description:

This is a branch instruction that sets the Program Counter to the address given in the operand, but only if the Carry flag is zero.

Instruction:

Inst	**Code**	**Operand**	**Description**
BCN	A1	Y	Branch if Carry flag is not set

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
BCN	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op	▨	▨

Execute cycles:

Gate operation is as per BCS instruction. Only change to execution is to Execute 2 where query is changed to Query: Is the Carry flag value zero?

C.25 BEZ – BRANCH IF ACCUMULATOR IS ZERO

Description:

This is a branch instruction that sets the Program Counter to the address given in the operand, but only if the Zero flag is one (i.e. the Accumulator value is zero).

Instruction:

Inst	**Code**	**Operand**	**Description**
BEZ	A2	Y	Branch if Zero flag is set

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
BEZ	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op	▨	▨

Execute cycles:

Gate operation is as per BCS instruction. Only change to execution is to Execute 2 where query is changed to Query: Is the Zero flag value one?

C.26 BNZ – BRANCH IF ACCUMULATOR IS NOT ZERO

Description:

This is a branch instruction that sets the Program Counter to the address given in the operand, but only if the Zero flag is zero (i.e. the Accumulator value is not zero).

Instruction:

Inst	**Code**	**Operand**	**Description**
BNZ	A3	Y	Branch if Zero flag is not set

Microcode:

Inst	**MC**	**Fetch**	**Execute 1**	**Execute 2**	**Execute 3**	**Execute 4**
BNZ	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op	▨	▨

Execute cycles:

Gate operation is as per BCS instruction. Only change to execution is to Execute 2 where query is changed to Query: Is the Zero flag value zero?

C.27 BRA – BRANCH ALWAYS

Description:

This is a branch instruction that sets the Program Counter to the address given in the operand.

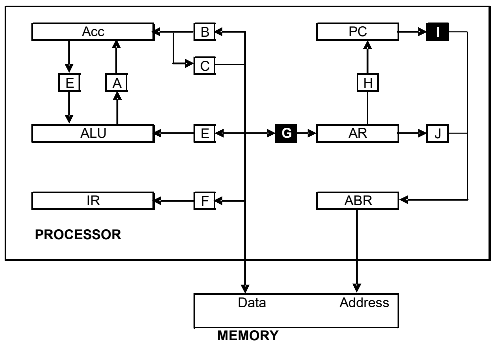
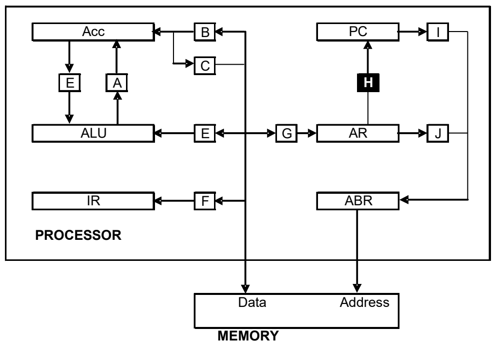
Instruction:

Inst	Code	Operand	Description
BRA	A4	Y	Branch to operand address

Microcode:

Inst	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
BRA	3	I, F PC=PC+1	I, G	H Int Op		

Execute cycles:

 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 1 • Open Gates I and G • Read memory (Instruction operand is written into Address Register) • Close Gates • Increment Program Counter
 Acc B C PC I E A H ALU E G AR J IR F ABR PROCESSOR Data Address MEMORY	Execute 2 • Open Gate H (Address Register copied into Program Counter) • Close Gate

C.28 MICROCODE SUMMARY

The instructions and their microcode are summarised in Table 18.

Inst	Code	MC	Fetch	Execute 1	Execute 2	Execute 3	Execute 4
HLT	00	2	I, F				
LDI	10	2	I, F PC=PC+1	I, B PC=PC+1			
LDA	11	3	I, F PC=PC+1	I, G PC=PC+1	J, B		
LDX	13	5	I, F PC=PC+1	I, G PC=PC+1	J	G	J, B
STA	20	3	I, F PC=PC+1	I, G PC=PC+1	J, C		
STX	22	5	I, F PC=PC+1	I, G PC=PC+1	J	G	J, C
ADC	40	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
SBC	41	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
XOR	45	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
ORA	46	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
AND	47	4	I, F PC=PC+1	I, G PC=PC+1	J, E	A Int Op	
ADI	48	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
SBI	49	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
XOI	4D	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
ORI	4E	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
ANI	4F	3	I, F PC=PC+1	I, E PC=PC+1	A Int Op		
SHR	60	2	I, F PC=PC+1	Int Op			
SHL	61	2	I, F PC=PC+1	Int Op			
ROR	62	2	I, F PC=PC+1	Int Op			
ROL	63	2	I, F PC=PC+1	Int Op			
CLC	80	2	I, F PC=PC+1	Int Op			
SEC	81	2	I, F PC=PC+1	Int Op			
BCS	A0	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BCN	A1	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BEZ	A2	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BNZ	A3	3	I, F PC=PC+1	I, G PC=PC+1	H Int Op		
BRA	A4	3	I, F PC=PC+1	I, G	H Int Op		

Table 18 Simple Processor Instruction Set microcode summary