A Simple Computer Design

16

- **Determine hardware requirement in design of a simple computer**
- **Discuss use of Register Transfer Language in computer design**
- **Design control unit of a simple computer**
- **Discuss how to program the simple computer in solving various problems**

In this chapter, we demonstrate how the knowledge that you gathered in this book can take you to the next higher level, where you can start designing a digital computer. A digital computer is capable of computation and taking decision based on binary coded instructions stored inside it. The central processing unit (CPU), also known as the brain of the computer sequentially fetches these instructions, decodes it and then executes it by performing some action through available hardware. In this chapter, we'll design a simple computer, which has a limited instruction set but is capable enough to solve variety of arithmetic and logic problems. The technique you learn in developing this simple machine will be useful when you go for a full-fledged computer design in some higher-level courses.

We begin the chapter by defining a small problem, which our simple computer should be able to solve. Next, we spell out different hardware components required as building blocks. We'll also discuss a simple hardware operation description language, called Register Transfer Language (RTL) useful for state machine design. Through RTL we'll describe all the operations of our simple computer. Then we'll design the control unit that will coordinate all these operations. Finally, we will discuss how to program this simple computer to solve the problem we started with and many other arithmetic and logic problems.

16.1 BUILDING BLOCKS

In Section 1.6 of Chapter 1, we have broadly seen the kind of components required for designing a computer. In this chapter, we address how to design central processing unit of a simple computer that interacts with a small memory module. Before we proceed further let us define a problem that our computer is supposed to solve. This is not the only problem it can handle. Depending on how we program it, we will be able to solve different arithmetic and logic problems and that is shown towards the end of this chapter through examples. The purpose of defining a problem is to choose specific hardware components that will serve as building block of our simple computer.

The Problem

Add 10 numbers stored in consecutive locations of memory. Subtract from this total a number, stored in 11th location of memory. Multiply this result with 2 and store it in 12th location of memory. All the numbers brought from memory lie between 0 and 9.

Memory

Since, the problem says the numbers or data to be fetched from memory and we also know that programs, i.e. binary coded instructions are also stored in memory, let us divide the memory used in our computer in two parts. One part stores the program or series of instructions the computer executes sequentially and this is known as *program memory*. The other part houses data that program uses and is also used for storing result. This is called *data memory*. From the given problem we find, we need 12 memory locations for data storage. We expect our computer won't need more than 20 instructions to complete the given task hence, a memory with 32 locations (integer power of 2) can be selected for our computer.

Now we try to decide how many bits of information we store in each address location. Usually, bits in memory locations are stored in multiple of 8 called *byte*. Let's see if our job can be done with 8 bits. Each memory location stores data between 0 and 9 on which program operates and thus require only 4 bits. The final result at most can be $10 \times 9 \times 2 = 180$ which requires 8-bit for representation. So the data memory can be of 8-bits with which we can represent decimal number up to $2^8 - 1 = 255$.

Let's now see the requirement of program memory. There, in each location, certain number of bits are allocated that defines the instruction to be executed. This is called operation code or in short, *opcode*. The rest of the bits can be used for referring the memory location from which data is to be brought or stored, if required by the instruction. Since, 32 memory locations require $\log_2 32 = 5$ bits for memory referencing we'll have $8 - 5 = 3$ bits for opcode specification giving $2^3 = 8$ different opcodes (Fig. 16.1). We'll see that 8 instructions are sufficient for the given kind of task in our limited ability computer. Hence, one important hardware component of our computer gets decided. The memory to be used is of size 32×8.

The above mode of addressing memory for data is called *direct addressing*. If the address mentioned in the instruction contains address from which actual data is to be brought it is called *indirect addressing*. If after opcode, in place of address actual data is made available, it is called *immediate addressing*. Note that, in immediate addressing data cannot be more than 5 bits as 3 bits gets used in opcode. Also note that the instruction like this is called single byte instruction. If an instruction requires 2 bytes to be fetched from program memory it is called 2-

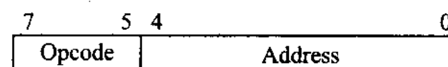

Fig. 16.1 Three Most Significant Bits (MSB) are opcode and five Least Significant Bits (LSB) are address

byte instruction. Obviously, in 2-byte instruction number of opcodes or memory addressing capability can be more than a single byte instruction.

Register Array

The computer needs a set of registers to perform its operation. Let us define them and assign task to each one of them for our simple computer. Note that, we are using a 32 × 8 memory module.

Memory Address Register (*MAR*) is a 5-bit register that stores the address of the memory location referred in a particular instruction. The output of this is fed to a 5-to-32 address decoder. Each output of the decoder points to a location in the memory. All memory referenced instruction loads memory address in *MAR*.

Memory Data Register (*MDR*) is an 8-bit register that stores the memory output when a memory read operation is performed. During memory write operation it stores the value that gets written to the memory. Thus it can also be called a memory buffer. In arithmetic or logic operation when more than one operand is required by ALU, one operand in our simple machine comes from *MDR*.

Program Counter (*PC*) is a 5-bit counter that stores the address of the memory location from which next instruction is fetched. At power on, our machine *PC* is reset so that its content is all zero. Thus location 00000 has to be a part of program memory and this is also the starting address from which program execution begins. Since, in our simple machine all the instructions are single byte instruction, every time an opcode is fetched we'll increment *PC* by one, and thus *PC* will point to location of the next opcode.

Instruction Register (*IR*) is a 3-bit register, which retains the opcode till it is properly executed in one or more clock cycles. Since all memory read and write operations are done through *MDR*, after an instruction is read from memory, 3 MSB that contains the opcode are transferred to *IR*.

Accumulator (*ACC*) is a multi-purpose register that always stores one operand of an arithmetic or logic operation. The result of this operation, i.e. ALU output is also stored in *ACC*. Functions like shifting of bits to left or right are also carried on *ACC*. Thus, in our simple computer *ACC* is a shift register with parallel load facility.

Timing Counter (*TC*) is a synchronous parallel load counter that stores and updates the timing information. The timing counter output is decoded to generate different timing signal, which in turn triggers different events in execution of an instruction. The counter is reset synchronously with clock once an instruction is fully executed. If an instruction is conceived as a *macro operation* then series of sequential steps necessary to carry out the instruction in the computer is called *micro operations*. In our simple computer, we are not expecting more than 8 micro operations for any macro-operations and hence a 3-bit counter is sufficient. Later if we see, we need more than 8 micro operations we'll change it to a 4-bit counter. Note that, a *master clock* (also called system clock) to which all the state changes of the computer are synchronized, triggers this counter. Also note, *TC* has power on reset facility, i.e. when the computer is switched on it stores 000.

Start/Stop Flag (*S*) is a flip-flop which when set, stops execution of the program. This we do in our simple computer by inhibiting the master clock. Like program counter, this also has a power-on-reset facility so that when the computer is switched on the master clock is not inhibited.

Other Important Hardware

Arithmetic Logic Unit (ALU) is a versatile combinatorial circuit that can perform a large range of arithmetic and logic operations. Since the data is 8-bit long, we use an 8-bit ALU. The control input value decides the function ALU executes at a particular time. ALU can accept up to two operands at a time, one from *ACC* and the other from *MDR*. The ALU output is stored in *ACC*. If addition operation generates a carry output from ALU, that can be stored in a flip-flop, often called carry flag (*CY*). Since, in our problem numbers are small in

magnitude the 8-bit ALU doesn't generate carry output and we don't need *CY* flag for our simple computer. Note that, ALU cannot perform multiply and division operation for which we use special hardware or some indirect technique.

Instruction Decoder (ID) is a 3-to-8 decoder, which takes input from *IR* and thereby decodes the opcode. In our simple computer there are 8 different opcodes, each one making one of the decoder output (D_0, D_1,..., D_7) high. This in turn initiates specific micro operations necessary to execute that opcode in subsequent clock cycles.

Timing Sequence Decoder (TSD) is again a 3-to-8 decoder that takes input from *TC* and provides necessary timing information in the form of decoded output (T_0, T_1,..., T_7) for a micro operation to be executed.

BUS is a group of wires that serve as a shared common path for data transfer of all the devices connected to it. With this, we do not need a separate device to device connection which increases the number of wiring specially when large number of devices are used in a system. Since, the largest group of binary data that is transferred in our computer is 8-bit, the bus used is an 8-bit bus.

BUS Selector (BS) is a multiplexer, which decides which one of all the connected devices is in transmission mode, i.e. has placed data in the BUS. Note that, if more than one device try to send data simultaneously, there will be a conflict producing erroneous result. However, in our computer we may allow more than one device connected to BUS to receive data from BUS. We'll see shortly that only *PC*, *ACC*, *MDR* and ALU want to transmit or place data on the BUS. *MAR* and *IR* only receive data and other hardware give control signal and don't do data transfer. Thus, BS has to select one of the four devices and uses eight (each one for one bit) 4-to-1 multiplexer type of device. We can also use tri-stated output for bus connection (Section 14.6 of Chapter 14) that will reduce the current loading on the device when it is not selected.

From this discussion we can draw the *data path* of our simple computer as shown in Fig. 16.2. Here, by data we mean address, opcode as well as operand and they move from/to memory, register, ALU, etc. Of course, we need another set of path to send control signal to various hardware to carry out microoperations. This is called *control path* and we'll design it when we define the instruction set for our computer.

Generally speaking, *address bus* is the group of wires that transfer address information, *data bus* is another group that transfers data and *control bus* transfers control information. Often, address information and data are transferred through a common bus and a control logic decides which is to be transferred and when. You might have noticed that in our simple computer design, we have used a common address and data bus. More about control bus will appear in Section 16.4 where we discuss the design of control unit.

Find in Fig. 16.2 the direction of arrow that shows the direction of data flow. Note that, *IR* and *MAR* can only receive data from BUS; *PC* can only send data by BUS; *ACC* and *MDR* can do both;

Memory data transfer takes place only via *MDR* and operands of ALU come from *ACC* and *MDR* and result is sent via BUS.

Example 16.1 In a particular configuration each memory location contains 16-bit data. In program memory, if 4 MSB contains opcode and rest contains address of memory locations give (a) Number of opcodes (b) Size of memory (c) Size of *PC*, *IR*, *ACC*, *MAR* and *MDR*.

Solution

(a) Number of opcodes = $2^4 = 16$ (Maximum)

(b) Number of address bits = $16 - 4 = 12$. No. of memory locations = $2^{12} = 2^2 \cdot 2^{10} = 4K$. So size of memory is $4K \times 16$.

(c) Size of *PC* and *MAR* = No. of address bits = 12. Size of *IR* = Size of opcode = 4. Size of *ACC* and *MDR* = No. of data bits = 16

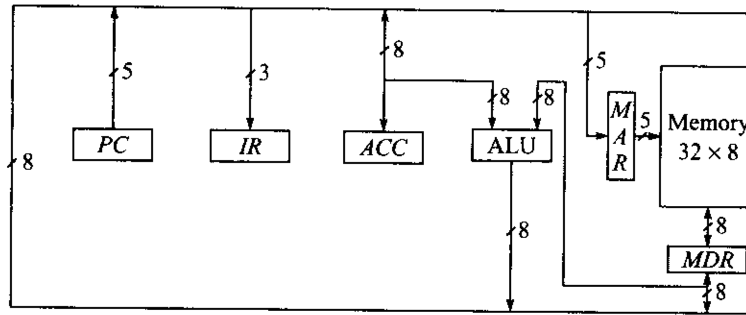

Fig. 16.2 Data path of the simple computer

SELF-TEST

1. **What is the highest integer in decimal that we can store in 16-bit data field?**
2. **What is an opcode?**
3. **What is the function of program counter?**
4. **What is indirect addressing?**

16.2 REGISTER TRANSFER LANGUAGE

Before we go for design of control path and the control unit as a whole we have to define macro operations and then we need to break up each macro operation in series of micro operations at register level. Register Transfer Language (RTL) gives a simple tool through which these micro operations can be expressed and then control unit can be designed from that. The basic structure of this language is

$$X : A \leftarrow B$$

This means, if condition X is TRUE, i.e. $X = 1$ then content of register B is transferred to register A. X can be a single logic variable or a logic expression like $xy \equiv x\&y$, $x + y \equiv x \mid y$, etc. In RTL we distinguish logic operation 'OR' from arithmetic operation 'addition' by assigning symbols ' | ' and '+' respectively. The logic AND is expressed by symbol '&'. However, if the '+' sign appears left to ':' in an RTL statement it means logical OR and '.' refers to logical AND. This is so because to left of ':' only logical operators can reside. Often AND, OR, NOT are expressed by '∧', '∨', '~' respectively. Also note, this register transfer destroys the previous content of A but not that of B. Both the register A and B now have the same value. If register transfer takes via BUS

$$A \leftarrow B \equiv BUS \leftarrow B, \quad A \leftarrow BUS$$

Since, BUS is not a register but a group of wire this means B getting access to BUS through BUS selector (BS) and the whole event takes place in one clock cycle. Figure 16.3 pictorially depicts register transfer without and with BUS.

To write anything to memory, in our simple computer we have to place the address information in MAR and the data to be written in MDR. Thus, memory write operation in RTL is expressed as

$$X : \mathrm{M}[MAR] \leftarrow MDR$$

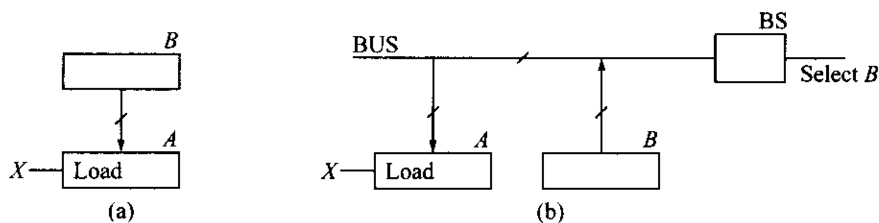

Fig. 16.3 **Register transfer A → B: (a) without BUS, (b) with BUS**

Similarly, memory read operation is also done through *MAR* and *MDR* and RTL expression is

$$X: MDR \leftarrow \mathrm{M}[MAR]$$

If certain bits of a register are to be addressed we use RTL as follows:

$$X: IR \leftarrow MDR[7{:}5]$$

The statement above refers to transfer of three most significant bits of *MDR* to *IR*, a 3-bit register when $X = 1$.

The arithmetic and logic operations of ALU that bring operands from *ACC* and *MDR* and store the result in *ACC* can be expressed in RTL in the following way

$X: ACC \leftarrow ACC \,\&\, MDR$	[logic AND]
$X: ACC \leftarrow ACC \mid MDR$	[logic OR]
$X: ACC \leftarrow ACC \oplus MDR$	[logic EX-OR]
$X: ACC \leftarrow ACC'$	[logic NOT]
$X: ACC \leftarrow ACC + MDR$	[arithmetic addition]
$X: ACC \leftarrow ACC - MDR$	[arithmetic subtraction]
$X: ACC \leftarrow ACC + 1$	[increment by 1]
etc.	

And finally if data is to be shifted in a register say by 1 bit to left we can write

$$X: ACC[7{:}1] \leftarrow ACC[6{:}0], ACC[0] \leftarrow 0$$

If such left shift occurs through carry the statement will be

$$X: ACC[7{:}1] \leftarrow ACC[6{:}0], ACC[0] \leftarrow CY$$

Normally, we come across these four kinds of micro operations namely (i) Inter-Register transfer (ii) Arithmetic operation (iii) Logic operation and (iv) Shift operation. Note that, left shift without carry can also be obtained by addition operation as shown in Example 6.14 of Chapter 6. Figure 16.4 shows how addition operation $A \leftarrow A + B$ takes place through ALU and BUS.

Note that, *TC* and *PC* can increment by 1 without taking help of ALU as they are designed to be parallel load up counters. For more complex processor unit where 2 byte, 3 byte instructions are possible we can have an adder unit accessible by *PC*.

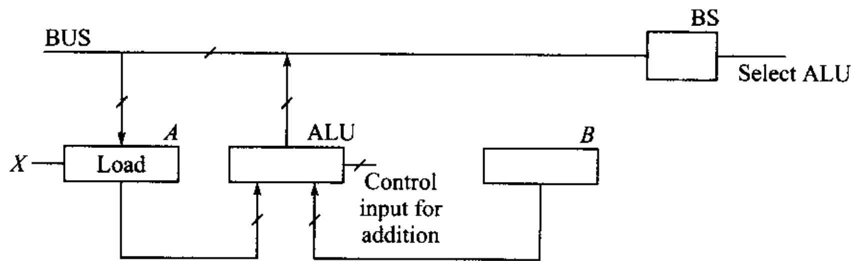

Fig. 16.4 The micro operation $A \rightarrow A + B$

Example 16.2 Explain what the following RTL statements perform

$T_1 : MDR \leftarrow ACC$

$T_2 : ACC \leftarrow ACC'$

$T_3 : ACC \leftarrow ACC \ \& \ MDR$

Solution The first statement says if $T_1 = 1$, content of *ACC* is transferred to *MDR*. The second statement says if $T_2 = 1$, content of *ACC* is complemented. The third statement says if $T_3 = 1$, bit-wise AND operation is performed on *ACC* and *MDR* and the result is stored in *ACC*. Since content of *MDR* and *ACC* were complement of one another before this statement is executed, by AND operation all the bits of *ACC* become zero, i.e. *ACC* is reset by these three statements irrespective of its initial content.

Note that, T_1, T_2 and T_3 can be output of a timing sequencer, which become active one after another in consecutive clock cycles. This way, *ACC* can be cleared in three clock cycles by above RTL statements.

SELF-TEST

5. What is RTL?
6. What is to be changed in Fig. 16.4 to perform $A \leftarrow A \ \& \ B$?

16.3 EXECUTION OF INSTRUCTIONS, MACRO AND MICRO OPERATIONS

In a computer, *execution of instructions* is carried through macro operations which again can be subdivided into micro operations. In this section, we first define the macro operations that we want to be executed in the computer we are designing. Next, we'll discuss micro operations necessary to execute each macro operation and it will be expressed through RTL. Remember that we have assigned only 3-bits as opcode and hence we can define $2^3 = 8$ instructions or macro operations with them. Table 16.1 lists all the instructions, corresponding mnemonics (easy to remember short forms), opcodes and 3-to-8 decoder (ID) output when *IR* is loaded with this opcode.

Table 16.1 Instruction Set for the Simple Computer

Macro operation performed	*Instruction mnemonic*	*Opcode*	*Instruction decoder (ID) output activated*
Load data from a specified memory location to *ACC*	LDA	000	D_0
Store *ACC* data in a specified memory location	STA	001	D_1
Halts execution of the program	HLT	010	D_2
Perform bitwise AND operation of *ACC* with data of a specified memory location and store result in *ACC*	AND	011	D_3
Perform bitwise NOT operation of *ACC*	NOT	100	D_4
Perform 1-bit left shift of *ACC* with $ACC[0] \leftarrow 0$	SHL	101	D_5
Perform addition operation of *ACC* with data of a specified memory location and store result in *ACC*	ADD	110	D_6
Subtract from *ACC*, data of a specified memory location and store result in *ACC*	SUB	111	D_7

Instruction Cycles

To carry out each instruction or macro operation the computer has to go through three distinct phases or cycles. In fetch cycle it brings the instruction or opcode from the program memory. In decoding phase it decodes the opcode and finally the execution is done in execute cycle. These cycles together known as *instruction cycle* are again repeated for next instruction. It is understandable that fetch and decode phase will be same for all instructions in our simple computer as we have only single byte directly addressed instructions. However, the execution cycle will be different for different instructions depending on the tasks the instruction wants to perform.

Fetch Cycle

An instruction cycle begins with *fetch cycle* when *TC* is reset to 0. Then, only T_0 output of TSD will be high and rest low. As told before *PC* contains the address of the location from which next instruction is to be fetched, content of *PC* is loaded into *MAR* in T_0.

At the next trigger of master clock *TC* is incremented by 1 so that T_1 becomes high and other outputs of TSD are low. In this clock cycle, content of memory from location specified by *MAR* (through 5-to-32 address decoder attached to memory) is loaded to *MDR*. *PC* now can be incremented to point to address of next location in program memory, which stores next instruction.

In the next clock cycle TC generates $T_2 = 1$ when opcode from 3 MSB of MDR is transferred to IR and 5 LSB to MAR. Content of IR is used for decoding opcode in decode phase. Content of MAR will be useful in execute phase if the opcode makes some memory reference, the address of which remain available at MAR. In RTL the above operations can be represented as

$$T_0 : MAR \leftarrow PC$$
$$T_1 : MDR \leftarrow \mathrm{M}[MAR],\ PC \leftarrow PC+1$$
$$T_2 : IR \leftarrow MDR[7{:}5],\ MAR \leftarrow MDR[4{:}0]$$

Decode Cycle

In *decode cycle* we decode the opcode fetched from program memory. Since at T_2, register IR is loaded with opcode and 3-to-8 decoder (ID) that decodes the opcode is a combinatorial circuit, we finish decoding in T_2 itself. In RTL we express it as

$$T_2 : D_0 \ldots D_7 \leftarrow \mathrm{DECODE}\ (IR)$$

Often, the 3rd statement of previously mentioned fetch cycle that loads IR with new opcode is considered a part of decode cycle or fetch-decode together is called fetch cycle.

Execute Cycle

Micro operations for each instruction are different and we list them first and then give the explanation.

LDA	$D_0T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_0T_4 : ACC \leftarrow MDR,\ TC \leftarrow 0$
STA	$D_1T_3 : MDR \leftarrow ACC$
	$D_1T_4 : \mathrm{M}[MAR] \leftarrow MDR,\ TC \leftarrow 0$
HLT	$D_2T_3 : S \leftarrow 1,\ TC \leftarrow 0$
AND	$D_3T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_3T_4 : ACC \leftarrow ACC\ \&\ MDR\ \ TC \leftarrow 0$
NOT	$D_4T_3 : ACC \leftarrow ACC',\ TC \leftarrow 0$
SHL	$D_5T_3 : ACC[7{:}1] \leftarrow ACC[6{:}0], ACC[0] \leftarrow 0, TC \leftarrow 0$
ADD	$D_6T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_6T_4 : ACC \leftarrow ACC + MDR,\ TC \leftarrow 0$
SUB	$D_7T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_7T_4 : ACC \leftarrow ACC - MDR, TC \leftarrow 0$

A quick overview of the above list shows, at the completion of each instruction cycle (fetch-decode-execute) TC is reset by which the computer goes to T_0 state and fetch cycle for next instruction begins. Note that, a detailed discussion on execution of the program at register level for every clock trigger appears in Section 16.5.

In operations like LDA, AND, ADD, SUB data is brought from memory, address of which is available in MAR. In executing STA the MAR content denotes the location where data is to be stored in memory.

Macro operations AND, NOT, ADD, SUB use ALU. When HLT is executed S flag is set which stops execution of the program. This flag is cleared through power-on-reset.

Now let's pick up one macro operation (say, LDA) and see how it gets executed through its constituent micro operations. From Table 16.1 we find instruction LDA transfers content of a specified memory location to ACC. If the opcode fetched in fetch cycle is 000 it refers to LDA operation. In decode phase, opcode 000 makes $D_0 = 1$ and the other outputs of ID are all zero. This is so till IR is refreshed or receives another opcode in the next fetch cycle, state T_2. Till then D_0 gives output 1.

The computer enters execution phase at state T_1 ($T_3 = 1$). Now as $D_0 = 1$, condition $D_0T_3 = 1$ and data is read from memory and loaded in MDR. Note that, the address of memory location from which data is to be brought was made available to MAR in state T_2. Also note that, memory content cannot directly be loaded into ACC (refer to data path shown in Fig. 16.8) and is to be done through MDR. In next clock cycle, i.e. when $D_0T_4 = 1$ the content of MDR is transferred to ACC via BUS and the macro operation is complete. We reset TC and let the computer begin a new instruction cycle. This analysis can be extended to explain execution of other instructions.

At this point we make an important observation that all the instruction executions are completed within 5 clock cycles (T_0 to T_4) and hence a 3-bit counter, which can count up to 8 is sufficient as TC in our simple computer.

SELF-TEST

7. What is a fetch cycle?
8. Why TC is reset every time an instruction is executed?

16.4 DESIGN OF CONTROL UNIT

The control unit is primarily a combinatorial circuit that supplies necessary controls inputs to all the important hardware elements of the computer. This takes timing information from computer master clock and is thus responsible for providing necessary *timing and control* information. The path through which these signals travel to reach different parts of a computer is called *control path*. Often we assign a group of wires, called *control bus* as shared path for this. The control logic is arrived at from (i) basic computer architecture we have adopted in the beginning, (ii) conditions appearing at left hand side of symbol ':' in RTL statements for our simple computer, given in previous section, and (iii) certain other issues, e.g. power-on-reset, control variables need to be activated for intended operation of a particular hardware, etc.

Loading Registers

Let us first see when parallel load control of IR is to be activated. We find from discussion of previous section, only during T_2 it is loaded. So TSD (Timing counter decoder) output T_2 can be directly connected as parallel load control input of IR. Every time T_2 is active this loads three MSBs of BUS (data path is such, refer to Fig. 16.2), which at that time holds MDR value, into IR. Obviously, at that time BUS selector (BS) should place content of MDR into BUS. This we'll discuss while designing control for BS.

What happens if we allow loading of IR say, in every clock cycle instead of above? Whenever there is some data made available in BUS by any hardware 3 MSB of that will be loaded into IR; ID (decoder) will immediately change and execution corresponding to a different opcode, not the intended one, may begin. You

can understand it'll be all chaos without any sense. Thus we return sanity to our simple machine by loading *IR* only when opcode is fetched, i.e. in T_2 and we can write logic relation

$$\text{LOAD}_{IR} = T_2$$

We see, *MDR* is loaded during T_1, D_0T_3, D_1T_3, D_3T_3, D_6T_3, D_7T_3 and corresponding condition is

$$\text{LOAD}_{MDR} = T_1 + (D_0 + D_1 + D_3 + D_6 + D_7)T_3$$

Proceeding in same manner we can write, $\text{LOAD}_{MAR} = T_0 + T_2$ and

$$\text{LOAD}_{ACC} = D_4T_3 + (D_0 + D_3 + D_6 + D_7)T_4$$

Memory Read/Write

Memory read signal is invoked by: $\text{READ}_\text{M} = T_1 + (D_0 + D_3 + D_6 + D_7)T_3$

Memory write signal is invoked by: $\text{WRITE}_\text{M} = D_1T_4$

ALU Control

Control variables of ALU activated for addition: $\text{ALU}_\text{ADD} = D_6T_4$

Control variables of ALU activated for subtraction: $\text{ALU}_\text{SUB} = D_7T_4$

Control variables of ALU activated for logic AND: $\text{ALU}_\text{AND} = D_3T_4$

Control variables of ALU activated for logic NOT: $\text{ALU}_\text{NOT} = D_4T_3$

BUS Controller

BUS controller gives access to *ACC* by $\text{BUS}_{ACC} = D_1T_3$,

to *PC* by $\text{BUS}_{PC} = T_0$,

to *MDR* by $\text{BUS}_{MDR} = T_2 + D_0T_4$

and to ALU by $\text{BUS}_\text{ALU} = D_4T_3 + (D_3 + D_6 + D_7)T_4$

Thus, selection inputs of eight 4-to-1 multiplexers that places data from one of these four devices *ACC*, *PC*, *MDR* and *ALU* on BUS should become active when corresponding conditions mentioned by above logic equations are met.

Other Control Signal

The condition for setting START/STOP flag *S* is: $\text{SET}_S = D_2T_3$ [*S* is power on reset]

The condition for shift left operation of *ACC* is: $\text{SHIFT_LEFT}_{ACC} = D_5T_3$

The signal that triggers increment of *PC*: $\text{INCREMENT}_{PC} = T_1$

Timing counter *TC* is synchronously reset by: $\text{RESET}_{TC} = (D_2 + D_4 + D_5)\, T_3 + (D_0 + D_1 + D_3 + D_6 + D_7)\, T_4$

Finally, the master clock remains enabled if flag *S* is not set. Thus $\text{ENABLE}_{CLOCK} = S'$

Based on these equations the control unit of our simple computer can be made. We show the control circuit of *ACC*, *TC* and *TSD*, BS in following three examples. Refer to problems of Section 4 of this chapter for more circuits. Together they make the control unit of our simple computer.

Example 16.3 Show using circuit diagrams the control inputs to *ACC*.

Solution The parallel load shift register *ACC* in the simple computer designed shifts data to left while serial data in is 0 (GND). It also loads parallel data from BUS. The conditions for these two operations are shown above in the form of logic equations. The corresponding diagram is shown in Fig. 16.5.

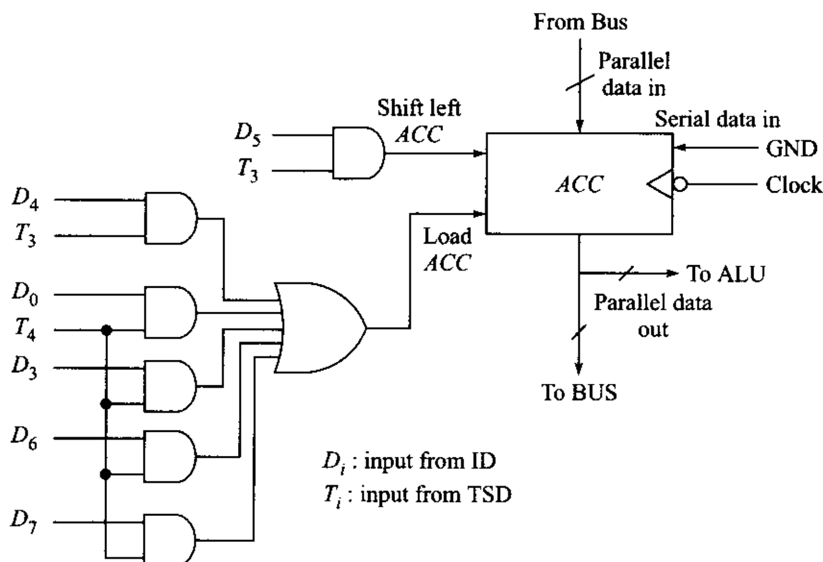

Fig. 16.5 Control for ACC

Example 16.4 Show using diagrams control inputs to *TC* and its connection to TSD.

Solution The timing counter *TC* is a mod-8 up counter with parallel load facility. When RESET_{TC} is activated according to control logic discussed in this section, 000 is synchronously loaded and up count resumes. The required circuit diagram is shown in Fig. 16.6.

Example 16.5 Show using diagram how bus controller works.

Solution The controller developed from control equations discussed in this section is shown in Fig. 16.7. This is developed on multiplexer logic like Fig. 4.5 of Chapter 5. A tri-state bus control can also be designed similar to diagram shown in Fig. 14.26 of Chapter 14. There the *DISABLE* control input will be fed by complement of respective BUS activate signal, i.e. complement of BUS_{ACC}, BUS_{PC} etc.

Note that, *PC* does not access bit 5 to 7 of BUS as it has only 5 bit binary information that is transferred to *MAR* via bit 0 to 4 of the BUS. Hence, for 3 MSB there is one AND gate less and the OR gate is of 3 input.

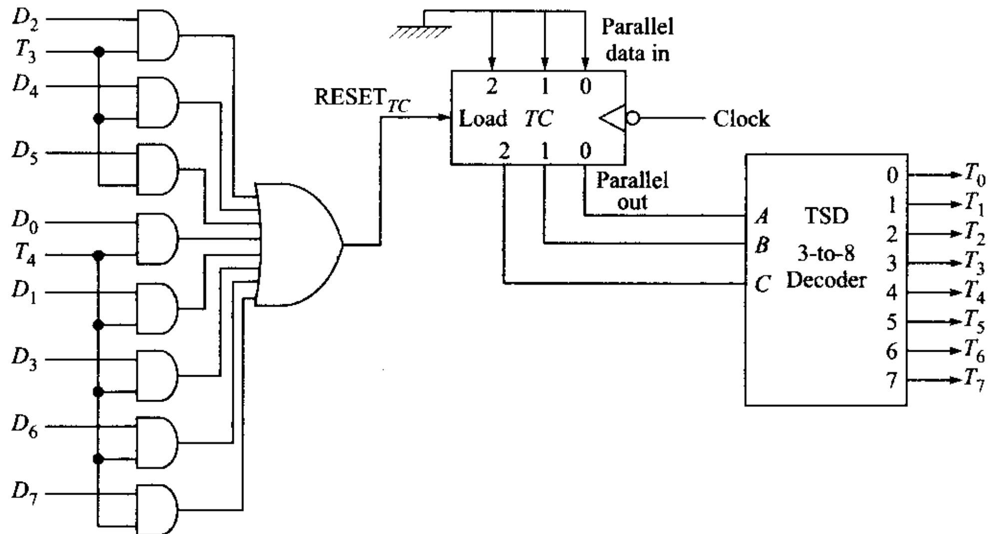

Fig. 16.6 **Control of Timing Counter, TC and its connection with Timing Sequence Detector, TSD**

SELF-TEST

9. How long instruction decoder outputs $D_7 \ldots D_0$ remain constant?
10. What are the instructions of this simple computer in which ALU places data on BUS?
11. Which instruction sets flag *F*?

16.5 PROGRAMMING COMPUTER

Now that our simple computer is ready with hardware and instruction sets let us see what computer program can solve the problem with which we started designing our simple machine. In Table 16.2 we present the program in mnemonics along with comments on job done by each instruction. Program in binary code as exists in 32 × 8 memory module will be shown after that.

Thus we need 14 instructions (Table 16.2), all single byte to solve the problem in our simple computer. We need 12 memory locations for storing numbers. So 14 + 12 = 26 bytes of our 32 byte memory are used for this problem. For bigger sized problems we need bigger memory and for more complex problem additional instruction sets and, of course, more complex computer architecture.

Now let us see how program and data remain stored in memory in binary numbers. We know that due to power-on-reset *PC* is always initialized with 00000, the first location of the memory (Refer Table 16.3) where first instruction of the program is to be stored. We use first 14 locations (address 00000 to 01101) of memory to store instructions. If we store data used in the program, i.e. 11 numbers in next consecutive locations then addresses 01110 to 11000 get filled. The location 11001, i.e. 26th location of memory can be used to store the result. Note that, multiplication is achieved by left shifting *ACC* and thus we don't need to store any multiplicand for that. If the 10 numbers to be added are say, 5, 2, 1, 3, 8, 6, 5, 2, 7, 4 and the number

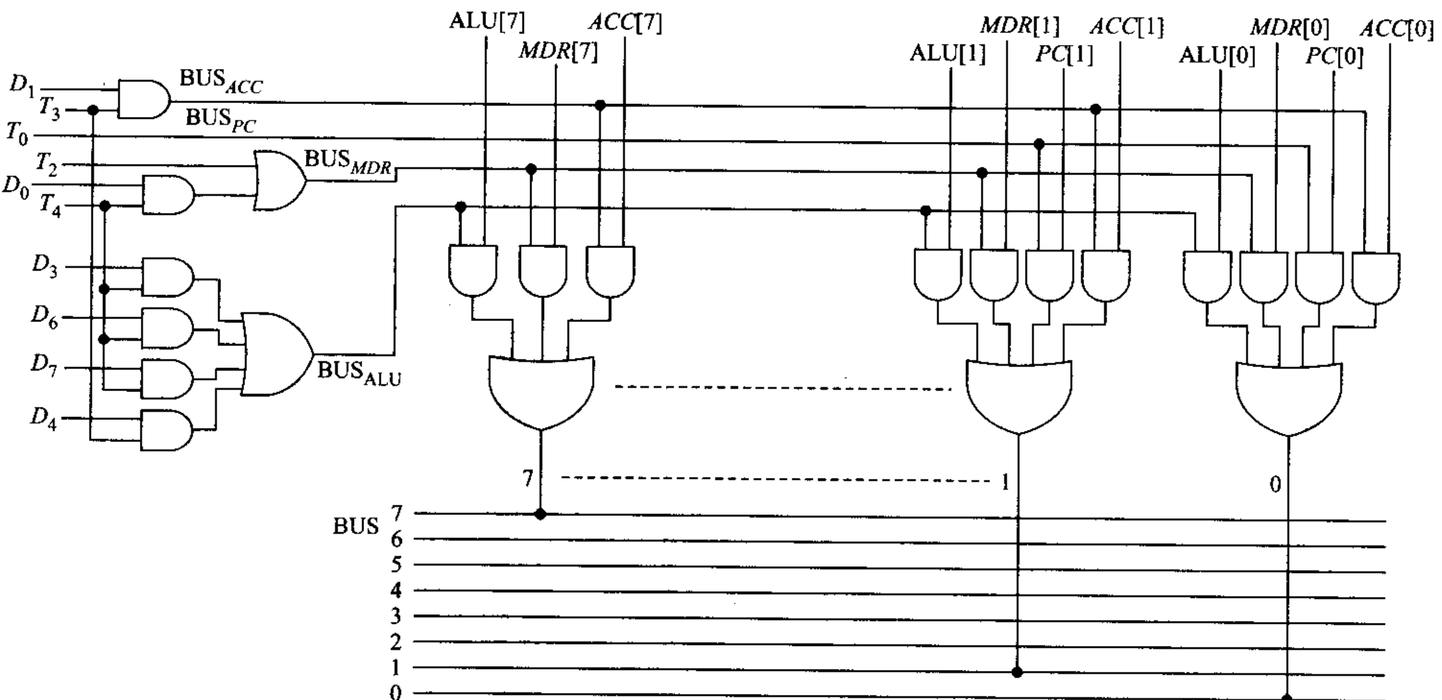

Fig. 16.7 Control over bus by different devices using multiplexer logic

Table 16.2 Program to Solve Given Problem with Comments

Instruction Number	*Instruction pnemonic*	*Comment*
1	LDA *addr*1	loads 1st number to *ACC*, the address of which follows opcode
2	ADD *addr*2	fetches 2nd number from memory and adds to 1st, stores the sum in *ACC*
3	ADD *addr*3	similarly adds 3rd number
4	ADD *addr*4	adds 4th number
5	ADD *addr*5	adds 5th number
6	ADD *addr*6	adds 6th number
7	ADD *addr*7	adds 7th number
8	ADD *addr*8	adds 8th number
9	ADD *addr*9	adds 9th number
10	ADD *addr*10	adds 10th number, now sum of 10 numbers remain available in *ACC*
11	SUB *addr*11	fetches 11th no. from memory, subtracts it from sum of 10 nos., stores result in *ACC*
12	SHL	shifts *ACC* to left by 1 bit, equivalent to multiplication by 2
13	STA *addr* 12	stores content of *ACC* in memory in the address available after opcode
14	HTL	halts the computer

subtracted is say, 9 then we can fill up first 25 locations of memory as shown in Fig. 16.14. The 26th location, before the program is run, may contain anything but after the program is run will contain the end result, i.e. 68 expressed in binary. Memory content is often shown in hexadecimal instead of binary. Refer to Problems 16.19 and 16.20 for this.

Program Execution

Now let us see how the program gets executed in first few instruction cycles. We note the change in the value of the registers along with ID and TSD in each clock cycle since the program begins. Table 16.4 shows sequential progress of our simple computer with every trigger of system clock. As told before *PC*, *TC* and *S* are power on reset. They all contain zero in the beginning when the computer is switched on.

In first clock cycle, the machine is in T_0 state given by TSD that decodes *TC*. At T_0, content of *PC* that contains the starting address of the program is copied to *MAR*. Corresponding micro operation is shown in rightmost column of Table 16.4. *TC* is incremented by 1.

In next clock cycle, *TC* and *PC* are incremented by 1, data from memory is loaded to *MDR* that contains the first instruction.

In 3rd clock cycle *TC* is incremented, *IR* gets the opcode and *MAR* gets address for first data. Note that decoding of *IR* is also done in same clock cycle that makes D_0 high, as opcode is 000 (LDA). This completes the fetch cycle, which is common for all instructions.

In executing LDA instruction the first state is T_3 state. Here, data from 15th location of Memory (*MAR* = 01110) which contains 00000101, decimal equivalent of 5 is loaded to *MDR*. In T_4 state this data is transferred to *ACC* and macro operation LDA is fully executed. This completes the first instruction cycle. Note that timing counter (*TC*) is to be reset after execution of data transfer from *MDR* to *ACC* and that begins the next instruction fetch.

Table 16.3 Program and Data Section of the Memory

Memory location number	*Memory address in binary*	*Memory content*	*Comment*
1	00000	00001110	Program section begins. Loads 1st no. from location 01110 to *ACC*. 3MSB 000: Load
2	00001	11001111	3MSB110: ADD, 5LSB 01111: Address of 2nd operand
3	00010	11010000	.
4	00011	11010001	.
5	00100	11010010	.
6	00101	11010011	First 14 locations, i.e. memory address 00000 to 01101 contain instructions.
7	00110	11010100	Here, three MSBs always refer to opcode. Five LSBs refer to memory
8	00111	11010101	address for instructions LDA, ADD, SUB, STA. For instructions
9	01000	11010110	SHL and HLT, five LSBs can be anything as they are not referred
10	01001	11010111	anywhere.
11	01010	11111000	.
12	01011	10100000	.
13	01100	00111001	.
14	01101	01000000	Halts computer. Program section ends.
15	01110	00000101	The data section starts. Stores 1st number, 5 expressed in binary
16	01111	00000010	2nd no. 2 in binary
17	10000	00000001	.
18	10001	00000011	.
19	10010	00001000	.
20	10011	00000110	.
21	10100	00000101	.
22	10101	00000010	.
23	10110	00000111	.
24	10111	00000100	.
25	11000	00001001	Stores 11th number, 9 that is subtracted from the sum of 10 nos.
26	11001	xxxxxxxx	After the program is run it becomes 01000100, i.e. 68 in decimal.
27	11010	xxxxxxxx	UNUSED
28	11011	xxxxxxxx	UNUSED
29	11100	xxxxxxxx	UNUSED
30	11101	xxxxxxxx	UNUSED
31	11110	xxxxxxxx	UNUSED
32	11111	xxxxxxxx	UNUSED

The fetch cycle is repeated in clock cycle 6 to 8. Since the instruction fetched is ADD (opcode 110) corresponding micro operations are performed in clock cycles 9 and 10 followed by next instruction fetch, starting again at 11th clock cycle. This continues till we reach 14th instruction HLT which when executed, sets *S* flag. This inhibits the system clock output in our design; thus content of all registers and memory will remain unchanged after that till the computer is switched off.

Table 16.4 Execution of the Program at Register Level

Clock Cycle	TC$	TSD	PC$	MAR	MDR	IR	ID	ACC	S	Micro operation performed after clock trigger
1	000	T_0	00000	00000	xxxxxxxx	xxx	x	xxxxxxxx	0	$MAR \leftarrow PC$
2	001	T_1	00000	00000	00001110	xxx	x	xxxxxxxx	0	$MDR \leftarrow$ M[MAR], $PC \leftarrow PC$+1
3	010	T_2	00001	01110	00001110	000	D_0	xxxxxxxx	0	$IR \leftarrow MDR$[7:5], $MAR \leftarrow MDR$[4:0]
4	011	T_3	00001	01110	00000101	000	D_0	xxxxxxxx	0	$MDR \leftarrow$ M[MAR]
5	100	T_4	00001	01110	00000101	000	D_0	00000101	0	$ACC \leftarrow MDR$, $TC \leftarrow 0$
6	000	T_0	00001	00001	00000101	000	D_0	00000101	0	$MAR \leftarrow PC$
7	001	T_1	00001	00001	11001111	000	D_0	00000101	0	$MDR \leftarrow$ M[MAR], $PC \leftarrow PC$+1
8	010	T_2	00010	01111	11001111	110	D_6	00000101	0	$IR \leftarrow MDR$[7:5], $MAR \leftarrow MDR$[4:0]
9	011	T_3	00010	01111	00000010	110	D_6	00000101	0	$MDR \leftarrow$ M[MAR]
10	100	T_4	00010	01111	00000010	110	D_6	00000111	0	$ACC \leftarrow ACC + MDR$, $TC \leftarrow 0$
11	000	T_0	00010	00010	00000010	110	D_6	00000111	0	$MAR \leftarrow PC$
12	001	T_1	00010	00010	11010000	110	D_6	00000111	0	$MDR \leftarrow$ M[MAR], $PC \leftarrow PC$+1
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...

$ *PC*, *TC* (also *TSD*) values shown are the ones before clock trigger while for other registers this is what appears after clock trigger (taking care from *TC* and *PC*).

Concluding Remark

Before we conclude our computer design exercise let us see what we have achieved and what more is needed to make this computer fully functional. We have designed a simple processor comprising register arrays, flag, small memory, BUS and control unit. In short, we have designed a *central processing unit* (CPU) that connects to a small memory module and is able to execute programs built on a small instruction set.

What we have not discussed is how data is entered into computer from an external device say, keypad and also how it displays data in some output device say, a monitor. We also have not discussed interesting and important issues like

(i) handling of jump instructions (CPU jumps to an address to fetch an instruction),
(ii) use of subroutines (program under program that is used and called many times),
(iii) memory management (slow-fast, addressing mode details),
(iv) interrupt handling (devices of different priorities asking attention and service of CPU),
(v) pipelined CPU (doing jobs in parallel if there is no conflict to enhance computer speed, e.g. execution phase of present instruction in parallel with fetch of next instruction), so on and so forth. These are covered in detail in titles related to modern computer design courses and an interested reader can refer to the same.

We shall conclude this chapter by revisiting *computer architecture* introduced in Section 1.6, carrying forward the discussions of this chapter. Figure 16.8 represents a basic 8-bit computer. The 8-bit data bus is bidirectional in nature i.e. CPU is capable of both reading and writing from/to a location defined by 16-bit address which is a total of 2^{16} = 64K locations. Individual RAM and ROM are of size 16K and this requires 14 bits for addressing. The two MSBs are sent to a 2 to 4 address decoder which generates four Chip Select (CS)

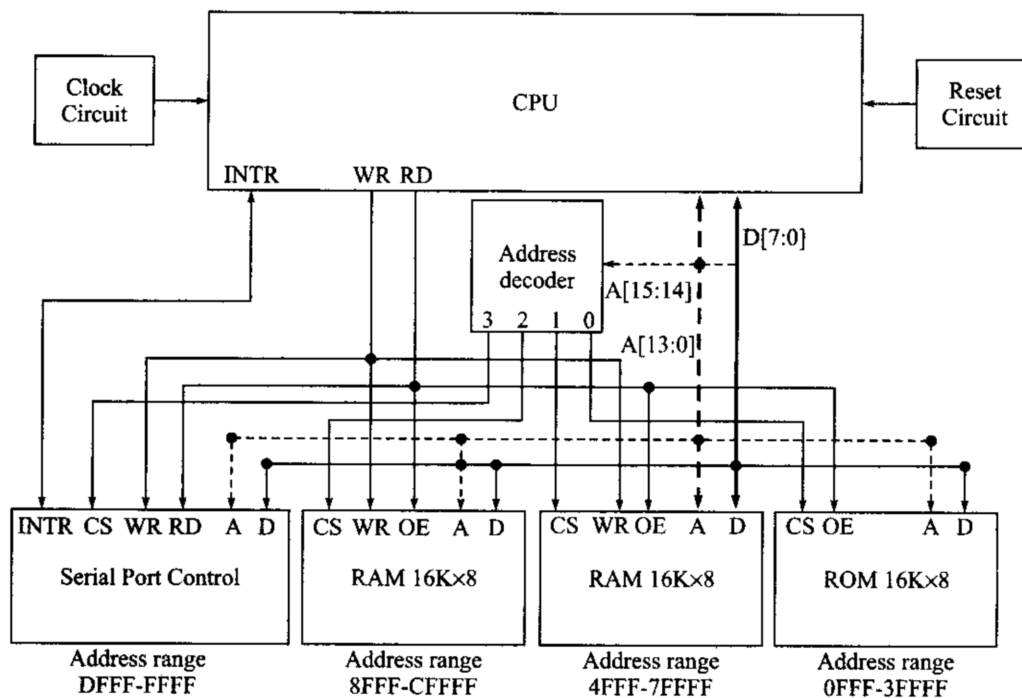

Fig. 16.8 **Basic architecture of an 8-bit computer**

signals, connected to each of the memory and output module generating unique address ranges as specified in the bottom of the figure. The calculation is as follows. For the ROM, A[15:14] is always 00 and thus all possible values of A[13:0] generate address ranges 0000 0000 0000 0000 to 0011 1111 1111 1111, i.e. 0000 to 3FFF in hex. Similarly, for the first RAM block, A[15:14] is always 01 and thus all possible values of A[13:0] generate address ranges 0100 0000 0000 0000 to 0111 1111 1111 1111, i.e. 4000 to 7FFF etc. CPU read is enabled by activating the control signal, RD (Read). This, in turn, requests outputs of the devices from which data is to be read to be enabled through OE (Output Enable) or through RD, if it is a serial input-output port, following which CPU takes the value from data bus. The control signal WR (Write) is activated to enable CPU writing to devices. Note that WR and RD should not be activated simultaneously. The timing of these control signals are also important so that data, chip select and address are properly stabilized to avoid false reading and writing operations. The ROM is not writable and usually contains sequence of instructions required for booting. This is usually used during *power on* of the computer and also in between, if the computer is asked to stop all operations and start afresh. The other time a computer may be asked to stop its usual fetch-decode-execute operations, but only temporarily, is when an *interrupt* is invoked. Then the computer's present state is stored in a designated memory space called *stack*. The computer comes back to its usual operating state once the interrupt is served usually through a *interrupt service routine* (ISR). There could be both software and hardware interrupts. The serial port control block shows how a hardware interrupt can ask service from CPU by activating INTR. Note that the *maskable interrupts* can be masked (disabled) by writing into a control register while *non-maskable interrupts* cannot be disabled. Reset is a non-maskable interrupt and care should be taken in the design of a computer so that corresponding ISR is in place before an interrupt is invoked.

Example 16.6 How many clock cycles are needed to execute the program shown in Table 16.2?

Solution The calculation of clock cycles is as follows.

One LDA	:	5
Nine ADD	:	$9 \times 5 = 45$
One SUB	:	5
One STA	:	5
One SHL	:	4
One HLT	:	4
TOTAL =		68 clock cycles

Example 16.7 Write a program for this computer that adds two positive integers, available in memory locations *addr*1 and *addr*2, multiplies the sum by 5 and stores final result in location *addr*3. Consider, the numbers are small enough not to cause any overflow, i.e. data at every stage require less than 8-bits for its representation.

Solution Addition of two numbers is straightforward and can be done using LDA and ADD instructions as done before. For multiplication with 5 we have to use an indirect technique. Two left shift give multiplication by 4 and one more addition will make it multiplication by 5. Alternatively, 5 ADD operations will also give multiplication by 5. The program can be written as follows.

```
LDA addr1
ADD addr2
STA addr3
SHL
SHL
ADD addr3
STA addr3
HLT
```

Note that, we have used *addr*3 as intermediate storage of addition result. Since in the computer designed there is no instruction to place data on a register from *ACC* (and also retrieve the same) we had to use memory. Storing intermediate results in registers speeds up the process but here we are limited by the architecture and instruction set available. Also note, we could have used any other available memory location for intermediate storage.

Example 16.8 Write a program for this computer that performs bit-wise Ex-OR operation on two numbers available in memory locations *addr*1 and *addr*2. The result is to be stored in location *addr*3.

Solution Our designed computer can perform only two kinds of logic operations AND and NOT. Therefore, we break Ex-OR logic of two numbers, say A and B in such a way that there is only AND and NOT operator.

$$Y = A \oplus B = AB' + A'B = ((AB')'.(A'B)')' \quad \text{[From DeMorgan's Theorem]}$$

Thus the program can be written as shown next. The logic operation performed by each instruction is shown as comment after semicolon.

```
LDA   addr1   ;A
NOT           ;A'
AND   addr2   ;A'B
NOT           ;(A'B)'
STA   addr3
LDA   addr2   ;B
NOT           ;B'
AND   addr1   ;AB'
NOT           ;(AB')'
AND   addr3   ;(AB')'.(A'B)'
NOT           ;((AB')'.(A'B)')'
STA   addr3
HLT
```

SELF-TEST

12. What happens to the program shown in Table 16.2 or Table 16.3 if HLT instruction is not provided?

SUMMARY

A computer stores program or binary coded instructions in its program memory. The centrai processing unit comprising set of registers and a control unit sequentially fetches this program, decodes it and executes the same. To accomplish this, an instruction, also called macro operation is broken into series of micro operations. Register Transfer Language is a very convenient tool to express each of these micro operations. A simple computer is designed in this chapter that has eight instructions and can perform logic operations like AND, NOT and arithmetic operations like addition and subtraction. It can also load data from memory and store data in memory. The data path and control unit of the computer is designed using hardware discussed in earlier chapters of the book. The programming technique for this computer for various arithmetic and logic problems is also demonstrated.

GLOSSARY

- ***accumulator*** A multipurpose register that stores one operand of all arithmetic and logic operations and also for memory referenced data transfers.
- ***address bus*** Group of wires that transfer address information.
- ***arithmetic logic unit*** A combinatorial circuit that can perform various types of arithmetic and logic functions decided by a set of selection inputs.
- ***bus*** A group of wire providing shared common path between number of devices.
- ***central processing unit*** The brain of computer that controls the operations of a computer.
- ***computer architecture*** Organization of a digital computer.

- ***control bus*** Group of wires that transfer control information.
- ***control path*** The path through which control signals travel to different devices and make them perform their assigned tasks.
- ***data bus*** Group of wires that transfer data.
- ***data memory*** The part of memory that contains data.
- ***data path*** The path through which data moves from one device to another in a computer.
- ***flag*** A single flip-flop that stores binary outcome of a certain operation.
- ***instruction register*** A register that contains the opcode or binary code of an instruction.
- ***interrupt*** An event that asks computer's immediate attention.
- ***interrupt service routine*** A set of computer instructions that serves an interrupt.
- ***macro operation*** An instruction that a computer executes in a complete instruction cycle, consists of series of micro operations.
- ***maskable interrupt*** Interrupt that can be disabled.
- ***memory address register*** A register that contains address of memory location for all memory referenced instructions.
- ***memory data register*** A register that acts as buffer between memory and rest of the circuit, storing data that moves to and from memory.
- ***micro operation*** The basic operation, a computer performs at register level.
- ***non-maskable interrupt*** Interrupt that cannot be disabled.
- ***opcode*** The binary code of an instruction.
- ***program*** Series of instructions that accomplishes a task in a computer.
- ***program counter*** A register that stores address of next instruction.
- ***program memory*** The part of memory that contains instruction.
- ***register transfer language*** A language, which expresses register transfer and condition for that.
- ***stack*** A memory block usually used for storing a computer's present state when interrupt is invoked.
- ***system clock*** The clock providing basic unit of clock cycle from which trigger of all sequential operations are derived.

PROBLEMS

Section 16.1

16.1 For memory configured as in Fig. 16.2, if immediate addressing is allowed what is the maximum value of number (in decimal) that can be loaded through instruction fetch?

16.2 What is the minimum size required for *MAR* if memory addressed has size 1K × 16?

16.3 For a more complex computer design, 75 different instructions are required. What size of *IR* would you likely choose?

16.4 Draw data path of the computer described next. The computer in addition to what is described in Section 16.2 has two more registers *P* and *Q*, which can transfer data to/from *ACC* via BUS. It also has a *CY* flag that stores ALU overflow and a *Z* flag that is set when all the bits of *ACC* are zero.

Section 16.2

16.5 What does the following statement mean $(X + Y): A \leftarrow B$

16.6 Explain the meaning of $XY: A \leftarrow B$

16.7 Give the final content of *ACC* when following statements are executed

$$T_1: ACC \leftarrow ACC \oplus MDR$$
$$T_2: ACC \leftarrow ACC'$$

16.8 State what the following statement performs for the computer described in problem 16.8

$$T_1: ACC \leftarrow ACC + MDR$$

$CY \& T_2 : P \leftarrow ACC$
$CY' \& T_2 : Q \leftarrow ACC$

Section 16.3

16.9 Show how shift left operation with carry is executed.
16.10 Show how shift right operation with carry can be executed.
16.11 Consider the first instruction of the simple computer is replaced by MVI that moves immediate data (immediate addressing) to *ACC*. Write micro operations for this instruction. What change in hardware is required for this?
16.12 Consider the first instruction of the simple computer is replaced by LDI that moves indirect data (indirect addressing) to *ACC*. Write micro operations for this instruction. Does it require any change in hardware?

Section 16.4

16.13 What does $\text{LOAD}_{MAR} = T_0 + T_2$ mean?
16.14 Explain if there will be any problem if by mistake the control unit is developed on logic equation $\text{LOAD}_{MAR} = T_0 + T_2 + T_4$.
16.15 Show using diagrams control inputs to ALU. Consider IC 74181 (Section 6.10, Chapter 6) is used as ALU.
16.16 Show using diagrams control inputs to Memory.

Section 16.5

16.17 How many clock cycles are required to execute program given for Example 16.7?
16.18 How many clock cycles are required to execute program given for Example 16.8?
16.19 If the two numbers used in Example 16.7 are 5 and 8, and data section immediately follows program section show the memory values in binary after the program is executed? How will it be represented in hexadecimal?
16.20 If the two numbers used in Example 16.8 are $F2_{16}$ and $D6_{16}$ and data section immediately follows program section show the memory values in binary after the program is executed?
16.21 Write a program that compares two binary data located in *addr*1 and *addr*2 of memory by making all the bits of *addr*3 one if two numbers are exactly equal.
16.22 Write a program that executes following where Data_{addr} refers to data corresponding to address *addr*.

$$\text{Data}_{addr7} = (\text{Data}_{addr1} + \text{Data}_{addr2} + \text{Data}_{addr3} + \text{Data}_{addr4}) \times 3 - (\text{Data}_{addr5} + \text{Data}_{addr6}) \times 2$$

Answers to Self-tests

1. $2^{16} - 1 = 65535$
2. A computer operation coded in a group of binary digits.
3. To store address from which next instruction is fetched.
4. The instruction fetches address of location in which address for operand exists.
5. Register Transfer Language.
6. Control input to ALU.
7. A part of instruction cycle that fetches instruction from memory that after decoding gets executed in execute cycle.
8. Resetting *TC* a new instruction cycle can begin.
9. Till it is loaded in next fetch cycle.
10. ADD, SUB, AND, NOT.
11. HLT.
12. It goes on executing by loading next content of memory which houses first number and since three MSB are 000 opcode decodes it as an LDA instruction.