UNIT - 4

BASIC PROCESSING UNIT

BASIC PROCESSING UNIT:

The heart of any computer is the central processing unit (CPU). The CPU executes all the machine instructions and coordinates the activities of all other units during the execution of an instruction. This unit is also called as the Instruction Set Processor (ISP). By looking at its internal structure, we can understand how it performs the tasks of fetching, decoding, and executing instructions of a program. The processor is generally called as the central processing unit (CPU) or micro processing unit (MPU).An high-performance processor can be built by making various functional units operate in parallel. High-performance processors have a pipelined organization where the execution of one instruction is started before the execution of the preceding instruction is completed. In another approach, known as superscalar operation, several instructions are fetched and executed at the same time. Pipelining and superscalar architectures provide a very high performance for any processor.

A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program. A program is a set of instructions performing a meaningful task. An instruction is command to the processor & is executed by carrying out a sequence of sub-operations called as micro-operations. Figure 1 indicates various blocks of a typical processing unit. It consists of PC, IR, ID, MAR, MDR, a set of register arrays for temporary storage, Timing and Control unit as main units

FUNDAMENTAL CONCEPTS:

Execution of a program by the processor starts with the fetching of instructions one at a time, decoding the instruction and performing the operations specified. From memory, instructions are fetched from successive locations until a branch or a jump

instruction is encountered. The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter (PC) or Instruction Pointer (IP). After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. But, when a branch instruction is to be executed, the PC will be loaded with a different (jump/branch address).

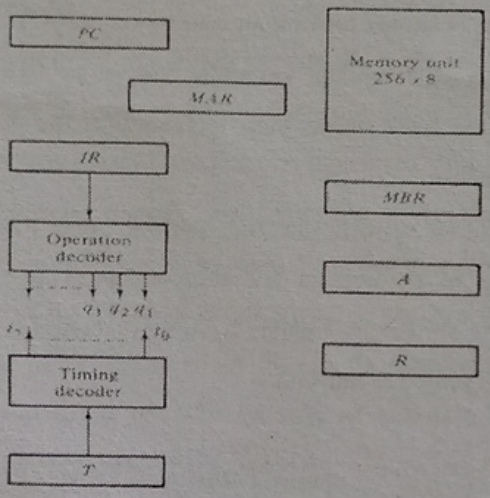


Fig-1

Instruction register, IR is another key register in the processor, which is used to hold the op-codes before decoding. IR contents are then transferred to an instruction decoder (ID) for decoding. The decoder then informs the control unit about the task to be executed. The control unit along with the timing unit generates all necessary control signals needed for the instruction execution. Suppose that each instruction comprises 2 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are interpreted as an instruction code to be executed. Hence, they are loaded into the IR/ID. Symbolically, this operation can be written as

$$IR \leftarrow [(PC)]$$

2. Assuming that the memory is byte addressable, increment the contents of the PC by 2, that is,

$PC \leftarrow [PC] + 2$

3. Decode the instruction to understand the operation & generate the control signals necessary to carry out the operation.

4. Carry out the actions specified by the instruction in the IR.

In cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction. These two steps together are usually referred to as the fetch phase; step 3 constitutes the decoding phase; and step 4 constitutes the execution phase.

To study these operations in detail, let us examine the internal organization of the processor. The main building blocks of a processor are interconnected in a variety of ways. A very simple organization is shown in **Figure 2**. A more complex structure that provides high performance will be presented at the end.

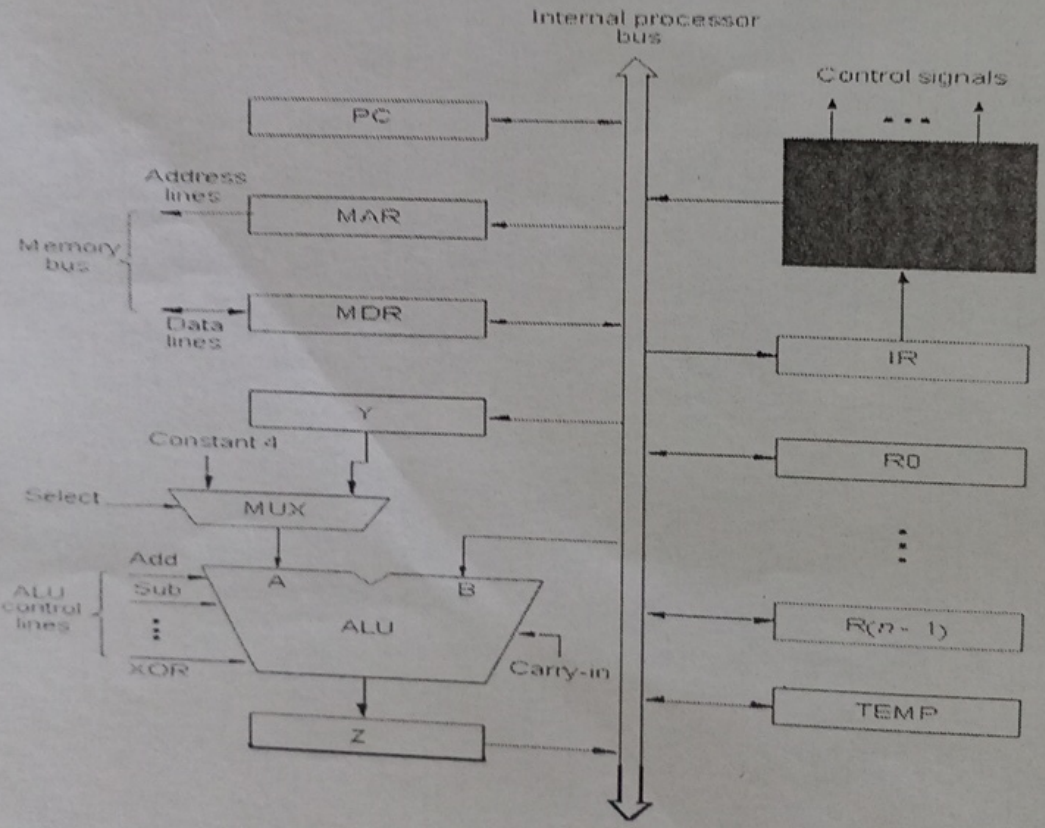


Fig 2

Figure shows an organization in which the arithmetic and logic unit (ALU) and all the registers are interconnected through a single common bus, which is internal to the processor. The data and address lines of the external memory bus are shown in Figure 7.1 connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

The number and use of the processor registers RO through R(n - 1) vary considerably from one processor to another. Registers may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as index registers or stack pointers. Three registers, Y, Z, and TEMP in Figure 2, have not been mentioned before. These registers are transparent to the programmer, that is, the programmer need not be concerned with them because they are never referenced explicitly by any instruction. They are used by the processor for temporary storage during execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction.

The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. We will refer to the two possible values of the MUX control input Select as Select4 and Select Y for selecting the constant 4 or register Y, respectively.

As instruction execution progresses, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation. The instruction decoder and control logic unit is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data. The

registers, the ALU, and the interconnecting bus are collectively referred to as the *data path*.

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

1. Transfer a word of data from one processor register to another or to the ALU
2. Perform an arithmetic or a logic operation and store the result in a processor register
3. Fetch the contents of a given memory location and load them into a processor register
4. Store a word of data from a processor register into a given memory location

We now consider in detail how each of these operations is implemented, using the simple processor model in Figure 2.

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register. This is represented symbolically in Figure 3. The input and output of register Ri are connected to the bus via switches controlled by the signals Ri_{in} and Ri_{out} respectively. When Ri_{in} is set to 1, the data on the bus are loaded into Ri. Similarly, when Ri_{out}, is set to 1, the contents of register Ri_{out} are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register RI to register R4. This can be accomplished as follows:

1. Enable the output of register $R1_{out}$ by setting RI_{out}, tc 1. This places the contents of R1 on the processor bus.
2. Enable the input of register R4 by setting $R4_{in}$ to 1. This loads data from the processor bus into register R4.

All operations and data transfers within the processor take place within time periods defined by the processor clock. The control signals that govern a particular transfer are asserted at the start of the clock cycle. In our example, RI_{out} and $R4_{in}$ are set to 1. The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock,

the flip-flops that constitute R4 will load the data present at their inputs. At the same time, the control signals $R1_{out}$ and $R4_{in}$ will return to 0. We will use this simple model of the timing of data transfers for the rest of this chapter. However, we should point out that other schemes are possible. For example, data transfers may use both the rising and falling edges of the clock. Also, when edge-triggered flip-flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as multiphase clocking.

An implementation for one bit of register Ri is shown in Figure 7.3 as an example. A two-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. When the control input Ri_{in} is equal to 1, the multiplexer selects the data on the bus. This data will be loaded into the flip-flop at the rising edge of the clock. When Ri_{in} is equal to 0, the multiplexer feeds back the value currently stored in the flip-flop.

The Q output of the flip-flop is connected to the bus via a tri-state gate. When Ri_{out} is equal to 0, the gate's output is in the high-impedance (electrically disconnected) state. This corresponds to the open-circuit state of a switch. When $Ri_{out} = 1$, the gate drives the bus to 0 or 1, depending on the value of Q.

EXECUTION OF A COMPLETE INSTRUCTION:

Let us now put together the sequence of elementary operations required to execute one instruction. Consider the instruction

Add (R3), R1

which adds the contents of a memory location pointed to by R3 to register R1. Executing this instruction requires the following actions:

1. Fetch the instruction.
2. Fetch the first operand (the contents of the memory location pointed to by R3).
3. Perform the addition.
4. Load the result into R1.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Fig 7

The listing shown in figure 7 above indicates the sequence of control steps required to perform these operations for the single-bus architecture of Figure 2. Instruction execution proceeds as follows. In step 1, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select4, which causes the multiplexer MUX to select the constant 4. This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z. The updated value is moved from register Z back into the PC during step 2, while waiting for the memory to respond. In step 3, the word fetched from the memory is loaded into the IR.

Steps 1 through 3 constitute the instruction fetch phase, which is the same for all instructions. The instruction decoding circuit interprets the contents of the IR at the beginning of step 4. This enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase. The contents of register R3 are transferred to the MAR in step 4, and a memory read operation is initiated.

Then the contents of RI are transferred to register Y in step 5, to prepare for the addition operation. When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed in step 6. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing Select Y. The sum is stored in register Z, then transferred to RI in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 7.6 except Y in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction. But, in Branch instructions the updated value of the PC is needed to compute the Branch target address. To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

Branch Instructions:

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Listing in figure 8 below gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3. The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

The offset X used in a branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Fig 8

For example, if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The reason for this can be readily appreciated from the control sequence in Figure 7. The PC is incremented during the fetch phase, before knowing the type of instruction being executed. Thus, when the branch address is computed in step 4, the PC value used is the updated value, which points to the instruction following the branch instruction in the memory.

Consider now a conditional branch. In this case, we need to check the status of the condition codes before loading a new value into the PC. For example, for a Branch-on-negative (Branch<0) instruction, step 4 is replaced with

$$\text{Offset-field-of-IR}_{out}\ \text{Add}, Z_{in}, \text{If N = 0 then End}$$

Thus, if N = 0 the processor returns to step 1 immediately after step 4. If N = 1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

MULTIPLE-BUS ORGANIZATION:

The resulting control sequences shown are quite long because only one data item can be transferred over the bus in a clock cycle. To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel.

Figure 7 depicts a three-bus structure used to connect the registers and the ALU of a processor. All general-purpose registers are combined into a single block called the register file. In VLSI technology, the most efficient way to implement a number of registers is in the form of an array of memory cells similar to those used in the implementation of random-access memories (RAMs) described in Chapter 5. The register file in Figure 9 is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU, where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z in Figure 2.

A second feature in Figure 9 is the introduction of the Incremental unit, which is used to increment the PC by 4.. The source for the constant 4 at the ALU input multiplexer is still useful. It can be used to increment other addresses, such as the memory addresses in Load Multiple and Store Multiple instructions.

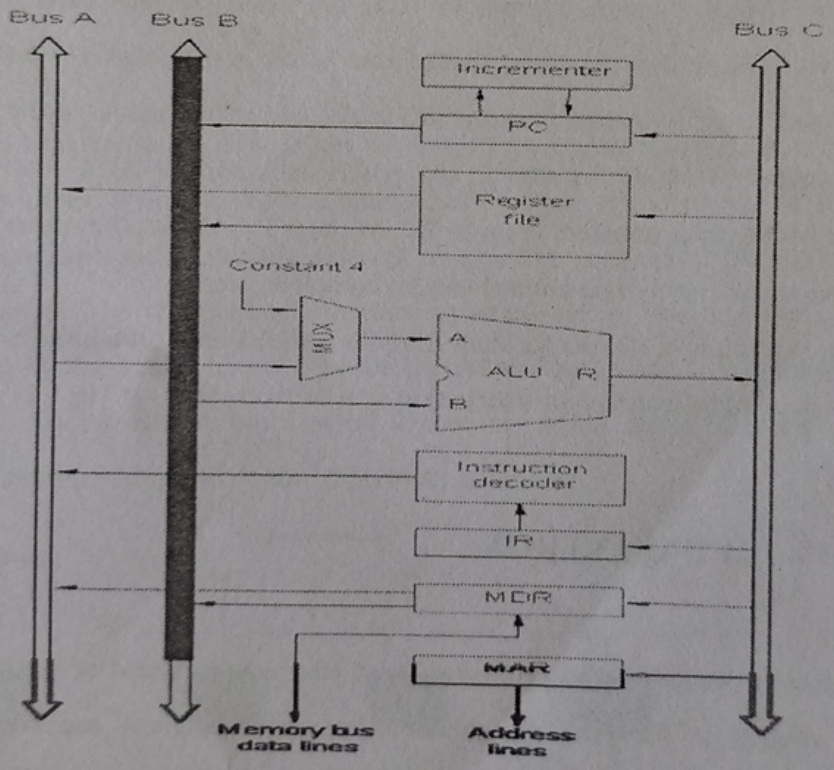


Fig 9

Consider the three-operand instruction

Add R4,R5,R6

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Fig 10

The control sequence for executing this instruction is given in Figure 10. In step 1, the contents of the PC are passed through the ALU, using the R=B control signal, and loaded into the MAR to start a memory read operation. At the same time the PC is incremented by 4. Note that the value loaded into MAR is the original contents of the PC. The incremented value is loaded into the PC at the end of the clock cycle and will not affect the contents of MAR. In step 2, the processor waits for MFC and loads the data received into MDR, then transfers them to IR in step 3. Finally, the execution phase of the instruction requires only one control step to complete, step 4.

By providing more paths for data transfer a significant reduction in the number of clock cycles needed to execute an instruction is achieved.

HARDWIRED CONTROL:

To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. Computer designers use a wide variety of techniques to solve this problem. The approaches used fall into one of two categories: hardwired control and micro programmed control. We discuss each of these techniques in detail, starting with hardwired control in this section.

Consider the sequence of control signals given in Figure 7. Each step in this sequence is completed in one clock period. A counter may be used to keep track of the control steps, as shown in Figure 11. Each state, or count, of this counter corresponds to

one control step. The required control signals are determined by the following information:

1. Contents of the control step counter
2. Contents of the instruction register
3. Contents of the condition code flags
4. External input signals, such as MFC and interrupt requests

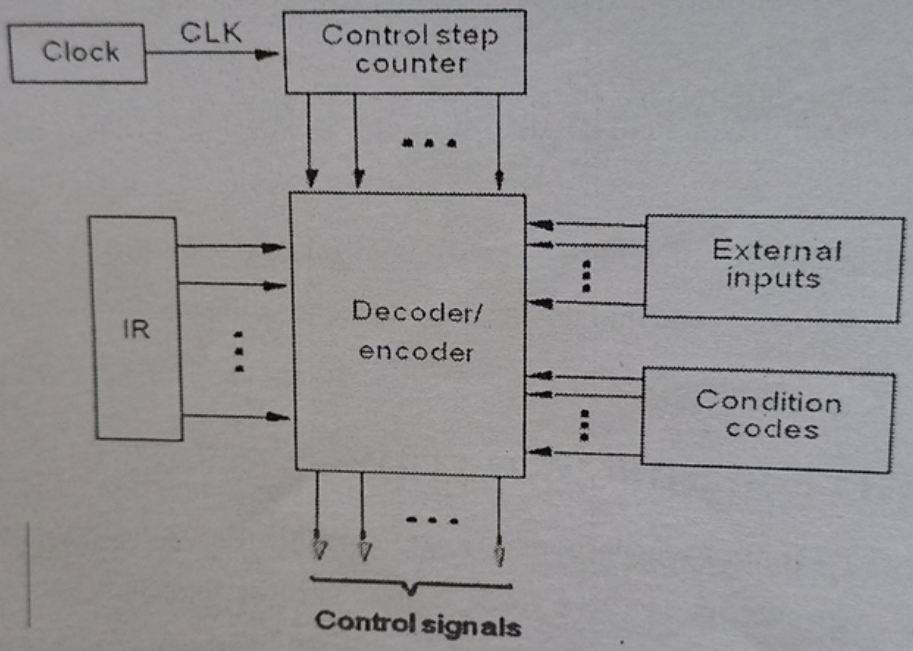


Fig 11

To gain insight into the structure of the control unit, we start with a simplified view of the hardware involved. The decoder/encoder block in Figure 11 is a combinational circuit that generates the required control outputs, depending on the state of all its inputs. By separating the decoding and encoding functions, we obtain the more detailed block diagram in Figure 12. The step decoder provides a separate signal line for each step, or time slot, in the control sequence. Similarly, the output of the instruction decoder consists of a separate line for each machine instruction. For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1, and all other lines

are set to 0. (For design details of decoders, refer to Appendix A.) The input signals to the encoder block in Figure 12 are combined to generate the individual control signals Y_{in}, PC_{out}, Add, End, and so on. An example of how the encoder generates the Z_{in} control signal for the processor organization in Figure 2 is given in Figure 13. This circuit implements the logic function

$$Z_{in}=T_1+T_6 \cdot ADD + T_4 \cdot BR+\cdots$$

This signal is asserted during time slot Ti for all instructions, during T_6 for an Add instruction, during T4 for an unconditional branch instruction, and so on. The logic function for Z_{in} is derived from the control sequences in Figures 7 and 8. As another example, Figure 14 gives a circuit that generates the End control signal from the logic function

$$End = T7 \cdot ADD + T5 \cdot BR + (T5 \cdot N + T4 \cdot N) \cdot BRN + \cdots$$

The End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value. Figure 12 contains another control signal called RUN. When

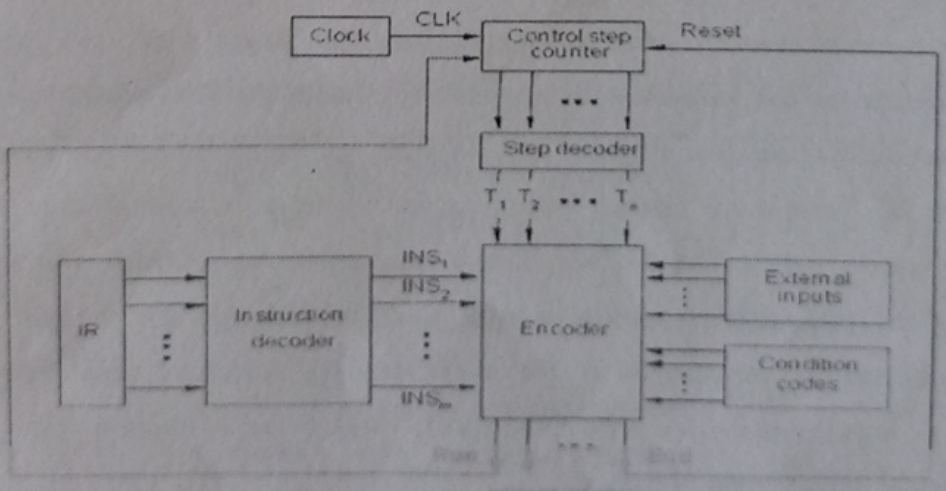


Fig 12

set to 1. RUN causes the counter to be incremented by one at the end of every clock cycle. When RUN is equal to 0, the counter stops counting. This is needed whenever the WMFC signal is issued, to cause the processor to wait for the reply from the memory.

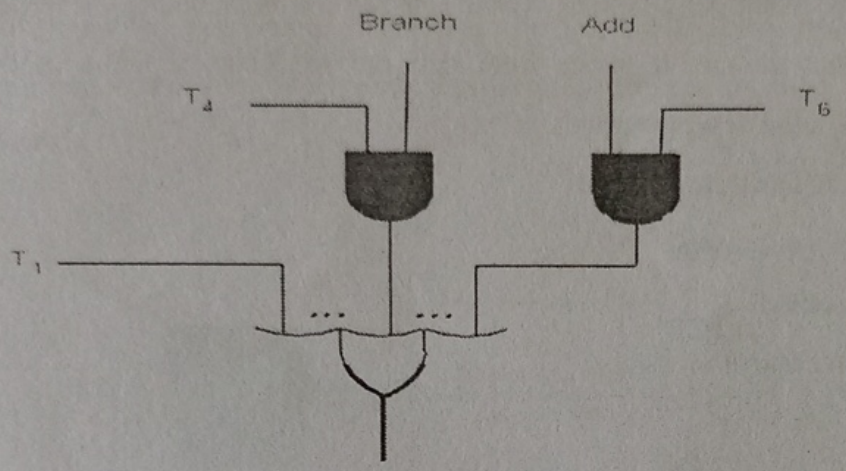


Fig 13a

The control hardware shown can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs. The outputs of the state machine are the control signals. The sequence of operations carried out by this machine is determined by the wiring of the logic elements, hence the name "hardwired." A controller that uses this approach can operate at high speed. However, it has little flexibility, and the complexity of the instruction set it can implement is limited.

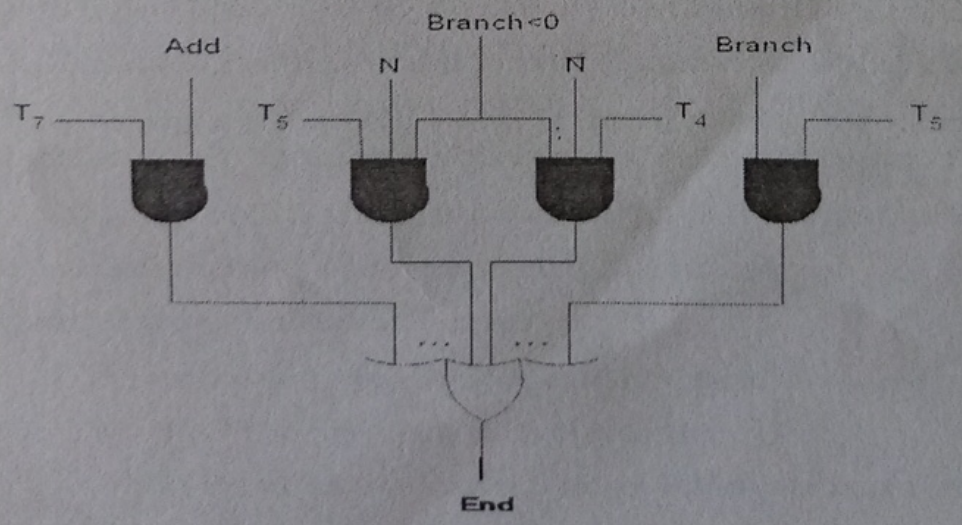
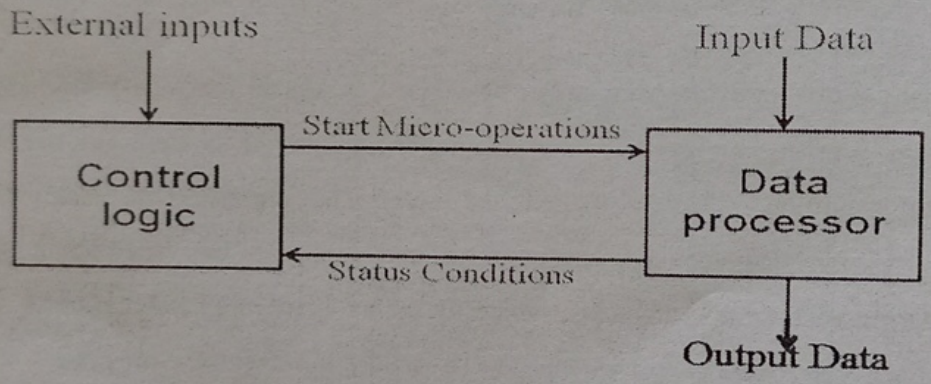


Fig 13b

MICROPROGRAMMED CONTROL:

ALU is the heart of any computing system, while Control unit is its brain. The design of a control unit is not unique; it varies from designer to designer. Some of the commonly used control logic design methods are:

- Sequence Reg & Decoder method
- Hard-wired control method
- PLA control method
- Micro-program control method



The control signals required inside the processor can be generated using a control step counter and a decoder/ encoder circuit. Now we discuss an alternative scheme, called micro programmed control, in which control signals are generated by a program similar to machine language programs.

Micro-instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	;
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	
7		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	

Fig 15

First, we introduce some common terms. A control word (CW) is a word whose individual bits represent the various control signals in Figure 12. Each of the control steps in the control sequence of an instruction defines a unique combination of Is and Os in the CW. The CWs corresponding to the 7 steps of Figure 6 are shown in Figure 15. We have assumed that Select Y is represented by Select = 0 and Select4 by Select = 1. A sequence of CWs corresponding to the control sequence of a machine instruction constitutes the micro routine for that instruction, and the individual control words in this micro routine are referred to as microinstructions.

The micro routines for all instructions in the instruction set of a computer are stored in a special memory called the control store. The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding micro routine from the control store. This suggests organizing the control unit as shown in Figure 16. To read the control words sequentially from the control store, a micro program counter (μPC) is used. Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the μPC. The μPC is then automatically incremented by the clock, causing ,successive microinstructions to be read from the control store. Hence, the control signals are delivered to various parts of the processor in the correct sequence.

One important function of the control unit cannot be implemented by the simple organization in Figure 16. This is the situation that arises when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action. In the case of hardwired control, this situation is handled by

including an appropriate logic function, in the encoder circuitry. In micro programmed control, an alternative approach is to use conditional branch microinstructions. In addition to the branch address, these microinstructions specify which of the external inputs, condition codes, or, possibly, bits of the instruction register should be checked as a condition for branching to take place.

The instruction Branch <0 may now be implemented by a micro routine such as that shown in Figure 17. After loading this instruction into IR, a branch

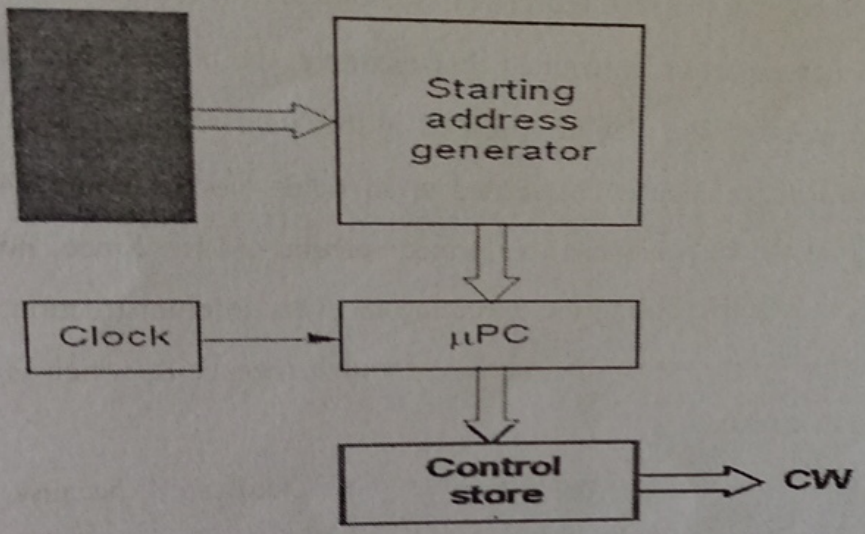


Fig 16

Address	Microinstruction
0	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
1	Z_{out} , PC_{in} , Y_{in} , WMFC
2	MDR_{out} , IR_{in}
3	Branch to starting address of appropriate microroutine
...	...
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out} , Select Y, Add, Z_{in}
27	Z_{out} , PC_{in} , End

Fig 17

microinstruction transfers control to the corresponding micro routine, which is assumed to start at location 25 in the control store. This address is the output of staring address generator block codes. If this bit is equal to 0, a branch takes place to location 0 to fetch a new machine instruction. Otherwise, the microinstruction at location 0 to fetch a new machine instruction. Otherwise the microinstruction at location 27 loads this address into the PC

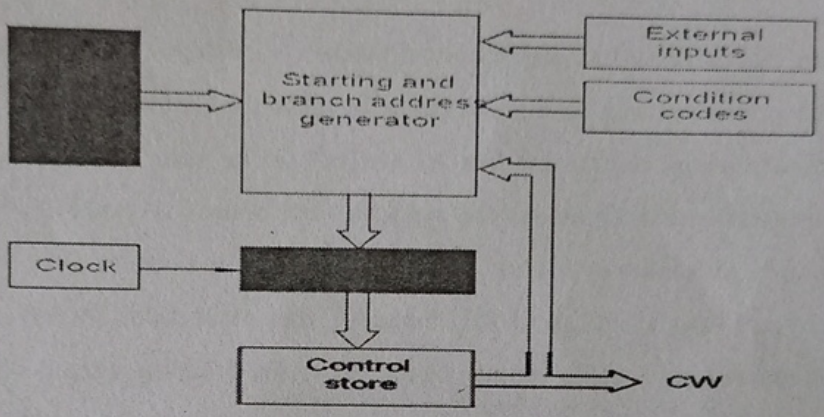


Fig 18

To support micro program branching, the organization of the control unit should be modified as shown in Figure 18. The starting address generator block of Figure 16 becomes the starting and branch address generator. This block loads a new address into the μPC when a microinstruction instructs it to do so. To allow implementation of a conditional branch, inputs to this block consist of the external inputs and condition codes as well as the contents of the instruction register. In this control unit, the μPC is incremented every time a new microinstruction is fetched from the micro program memory, except in the following situations:

1. When a new instruction is loaded into the IR, the μPC is loaded with the starting address of the micro routine for that instruction.
2. When a Branch microinstruction is encountered and the branch condition is satisfied, the μPC is loaded with the branch address.
3. When an End microinstruction is encountered, the μPC is loaded with the address

of the first CW in the micro routine for the instruction fetch cycle

Microinstructions

Having described a scheme for sequencing microinstructions, we now take a closer look at the format of individual microinstructions. A straightforward way to structure Microinstruction is to assign one bit position to each control signal, as in Figure 15.

However, this scheme has one serious drawback — assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large. Moreover, only a few bits are set to 1 (to be used for active gating) in any given microinstruction, which means the available bit space is poorly used. Consider again the simple processor of Figure 2, and assume that it contains only four general-purpose registers, R0, R1, R2, and R3. Some of the connections in this processor are permanently enabled, such as the output of the IR to the decoding circuits and both inputs to the ALU. The remaining connections to various registers require a total of 20 gating signals. Additional control signals not shown in the figure are also needed, including the Read, Write, Select, WMFC, and End signals. Finally, we must specify the function to be performed by the ALU. Let us assume that 16 functions are provided, including Add, Subtract, AND, and XOR. These functions depend on the particular ALU used and do not necessarily have a one-to-one correspondence with the machine instruction OP codes. In total, 42 control signals are needed.

If we use the simple encoding scheme described earlier, 42 bits would be needed in each microinstruction. Fortunately, the length of the microinstructions can be reduced easily. Most signals are not needed simultaneously, and many signals are mutually exclusive. For example, only one function of the ALU can be activated at a time. The source for a data transfer must be unique because it is not possible to gate the contents of two different registers onto the bus at the same time. Read and Write signals to the memory cannot be active simultaneously. This suggests that signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one *micro operation* per group is specified in any microinstruction. Then it is possible to use a

binary coding scheme to represent the signals within a group. For example, four bits suffice to represent the 16 available functions in the ALU. Register output control signals can be placed in a group consisting of PC_{out}, MDR_{out}, Z_{out}, $Offset_{out}$, $R0_{out}$ RI_{out}, $R2_{out}$, $R3_{out}$, and $TEMP_{out}$. Any one of these can be selected by a unique 4-bit code.

Further natural groupings can be made for the remaining signals. Figure 19 shows an example of a partial format for the microinstructions, in which each group occupies a field large enough to contain the required codes. Most fields must include one inactive code for the case in which no action is required. For example, the all-zero pattern in FI indicates that none of the registers that may be specified in this field should have its contents placed on the bus. An inactive code is not needed in all fields. For example, F4 contains 4 bits that specify one of the 16 operations performed in the ALU. Since no spare code is included, the ALU is active during the execution of every microinstruction. However, its activity is monitored by the rest of the machine through register Z, which is loaded only when the Z_{in} signal is activated.

Grouping control signals into fields requires a little more hardware because decoding circuits must be used to decode the bit patterns of each field into individual control signals. The cost of this additional hardware is more than offset by the reduced number of bits in each microinstruction, which results in a smaller control store. In Figure 19, only 20 bits are needed to store the patterns for the 42 signals.

So far we have considered grouping and encoding only mutually exclusive control signals. We can extend this idea by enumerating the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can

Microinstruction

F1	F2	F3	F4	F5
F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$	1111: XOR	
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	16 ALU functions	
0101: $R1_{out}$	101: $R1_{in}$			
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	...

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY 1: Select4	0: No action 1: WMFC	0: Continue 1: End

Fig 19

then be assigned a distinct code that represents the microinstruction. Such full encoding is likely to further reduce the length of micro words but also to increase the complexity of the required decoder circuits.

Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a *vertical organization*. On the other hand, the minimally encoded scheme of Figure 15, in which many resources can be controlled with a single microinstruction, is called a *horizontal organization*. The horizontal approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources. The vertical approach results in considerably slower operating speeds because more microinstructions are needed to perform the desired control functions. Although fewer bits are required for each microinstruction, this does not imply that the total number of bits in the control store is smaller. The significant factor is that less hardware is needed to handle the execution of microinstructions. Horizontal and vertical organizations represent the two organizational extremes in micro programmed control. Many intermediate schemes are also possible, in which the degree of encoding is a design parameter. The layout in Figure 19 is a horizontal organization because it groups only mutually exclusive micro operations in the

same fields. As a result, it does not limit in any way the processor's ability to perform various micro operations in parallel.

Although we have considered only a subset of all the possible control signals, this subset is representative of actual requirements. We have omitted some details that are not essential for understanding the principles of operation.