Module No-3: Basics of Control Unit Design and Pipelining:

- Design of a control unit: Data path design.
- Single Cycle Datapath for : ALU design / Data Movement Instructions / Control Unit Design
- Multi cycle microarchitecture; concept of states and transitions;
- Hardwired and Microprogrammed control.
- The state machine; Horizontal and Vertical micro instruction, Micro programmed control design techniques;
- Pipelining: Basic concepts, Instruction and arithmetic pipeline; Elementary concepts of hazards in pipeline and techniques for their removal.

CPU Design

1. **Introduction to CPU Design**
2. **Processor Organization**
3. **Execution of complete Instruction**
4. **Design of control unit**
5. **Microprogrammed control - I**
6. **Microprogrammed control - II**

Introduction to CPU

The operation or task that must perform by CPU are:

- **Fetch Instruction:** The CPU reads an instruction from memory.
- **Interpret Instruction:** The instruction is decoded to determine what action is required.
- **Fetch Data:** The execution of an instruction may require reading data from memory or I/O module.
- **Process data:** The execution of an instruction may require performing some arithmatic or logical operation on data.
- **Write data:** The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is beign executed. In other words, the CPU needs a small internal memory. These storage location are generally referred as registers.

The major components of the CPU are an arithmatic and logic unit (ALU) and a control unit (CU). The ALU does the actual computation or processing of data. The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU.

The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components:

Data Bus:
Data bus is used to transfer the data between main memory and CPU.

Address Bus:
Address bus is used to access a particular memory location by putting the address of the memory location.

Control Bus:
Control bus is used to provide the different control signal generated by CPU to different part of the system. As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

There are three basic components of CPU: register bank, ALU and Control Unit. There are several data movements between these units and for that an internal CPU bus is used. Internal CPU bus is needed to transfer data between the various registers and the ALU.

The internal organization of CPU in more abstract level is shown in the Figure 5.1 and Figure 5.2.

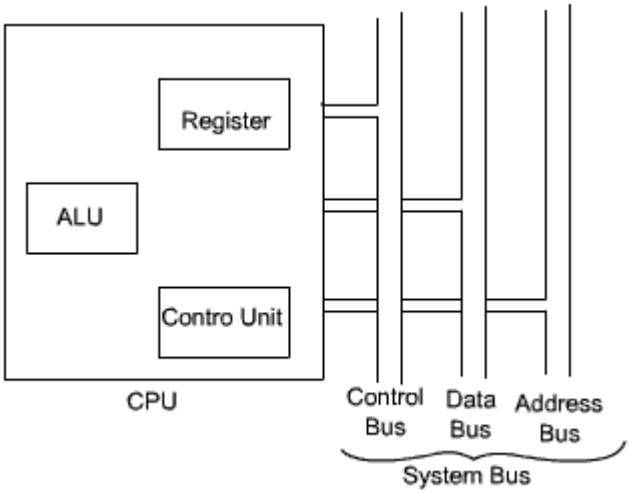


Figure 5.1 : CPU with the system Bus

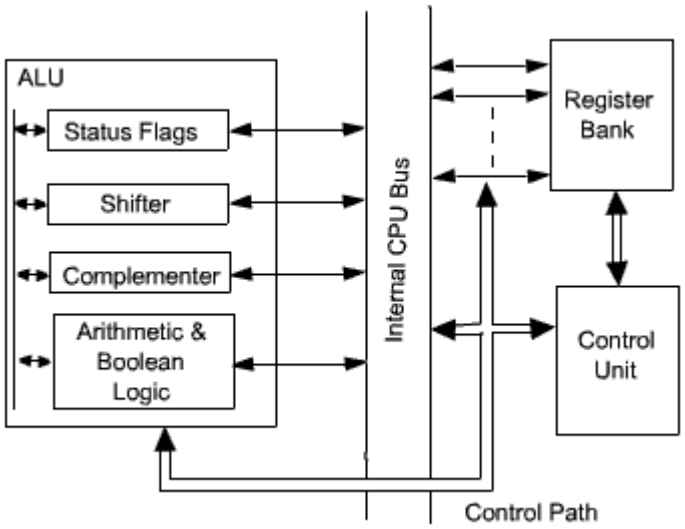


Figure 5.2 : Internal Structure of the CPU

Register Organization

A computer system employs a memory hierarchy. At the highest level of hierarchy, memory is faster, smaller and more expensive. Within the CPU, there is a set of registers which can be treated as a memory in the highest level of hierarchy. The registers in the CPU can be categorized into two groups:

- **User-visible registers:** These enables the machine - or assembly-language programmer to minimize main memory reference by optimizing use of registers.
- **Control and status registers:** These are used by the control unit to control the operation of the CPU. Operating system programs may also use these in privileged mode to control the execution of program.

User-visible Registers:

The user-visible registars can be categorized as follows:

- General Purpose Registers
- Data Registers
- Address Registers
- Condition Codes

General-purpose registers can be assigned to a variety of functions by the programmer. In some cases, general- purpose registers can be used for addressing functions (e.g., register indirect, displacement).

In other cases, there is a partial or clean separation between data registers and address registers.

Data registers may be used to hold only data and cannot be employed in the calculation of an operand address.

Address registers may be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointer:** In a machine with segment addressing, a segment register holds the address of the base of the segment. There may be multiple registers, one for the code segment and one for the data segment.
- **Index registers:** These are used for indexed addressing and may be autoindexed.
- **Stack pointer:** If there is user visible stack addressing, then typically the stack is in memory and there is a dedicated register that points to the top of the stack.

Condition Codes (also referred to as flags) are bits set by the CPU hardware as the result of the operations. For example, an arithmatic operation may produce a positive, negative, zero or overflow result. In addition to the result itself beign stored in a register or memory, a condition code is also set. The code may be subsequently be tested as part of a condition branch operation. Condition code bits are collected into one or more registers.

Register Organization

There are a variety of CPU registers that are employed to control the operation of the CPU. Most of these, on most machines, are not visible to the user.

Different machines will have different register organizations and use different terminology. We will discuss here the most commonly used registers which are part of most of the machines.

Four registers are essential to instruction execution:

Program Counter (PC): Contains the address of an instruction to be fetched. Typically, the PC is updated by the CPU after each instruction fetched so that it always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC.

Instruction Register (IR): Contains the instruction most recently fetched. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed.

Memory Address Register (MAR): Containts the address of a location of main memory from where information has to be fetched or information has to be stored. Contents of MAR is directly connected to the address bus.

Memory Buffer Register (MBR): Contains a word of data to be written to memory or the word most recently read. Contents of MBR is directly connected to the data bus.It is also known as Memory Data Register(MDR).

Apart from these specific register, we may have some temporary registers which are not visible to the user. As such, there may be temporary buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user visible registers.

Processor Status Word

All CPU designs include a register or set of registers, often known as the processor status word (PSW), that contains status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmatic operation.
- **Zero:** Set when the result is zero.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high order bit.
- **Equal:** Set if a logical campare result is equal.
- **Overflow:** Used to indicate arithmatic overflow.
- **Interrupt enable/disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicate whether the CPU is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

Apart from these, a number of other registers related to status and control might be found in a particular CPU design. In addition to the PSW, there may be a pointer to a block of memory containing additional status information (e.g. process control blocks).

Concept of Program Execution

The instructions constituting a program to be executed by a computer are loaded in sequential locations in its main memory. To execute this program, the CPU fetches one instruction at a time and performs the functions specified. Instructions are fetched from successive memory locations until the execution of a branch or a jump instruction.

The CPU keeps track of the address of the memory location where the next instruction is located through the use of a dedicated CPU register, referred to as the program counter (PC). After fetching an instruction, the contents of the PC are updated to point at the next instruction in sequence.

For simplicity, let us assume that each instruction occupies one memory word. Therefore, execution of one instruction requires the following three steps to be performed by the CPU:

1. Fetch the contents of the memory location pointed at by the PC. The contents of this location are interpreted as an instruction to be executed. Hence, they are stored in the instruction register (IR). Symbolically this can be written as:
   IR = [[PC]]
2. Increment the contents of the PC by 1.
   PC = [PC] + 1
3. Carry out the actions specified by the instruction stored in the IR.

The first two steps are usually referred to as the fetch phase and the step 3 is known as the execution phase. Fetch cycle basically involves read the next instruction from the memory into the CPU and along with that update the contents of the program counter. In the execution phase, it

interpretes the opcode and perform the indicated operation. The instruction fetch and execution phase together known as instruction cycle. The basic instruction cycle is shown in the Figure 5.3.

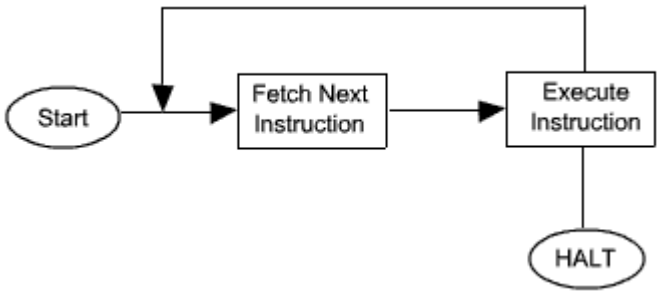


Figure 5.3: Basic Instruction cycle

In cases, where an instruction occupies more than one word, step 1 and step 2 can be repeated as many times as necessary to fetch the complete instruction. In these cases, the execution of a instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory access are required.

The fetched instruction is loaded into the instruction register. The instruction contains bits that specify the action to be performed by the processor. The processor interpretes the instruction and performs the required action. In general, the actions fall into four categories:

- **Processor-memory:** Data may be transfrred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmatic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered.

The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.

The execution cycle of a perticular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations the basic instruction cycle can be expanded with more details view in the Figure 5.4. The figure is in the form of a state diagram.

Interrupts

Virtually all computers provide a mechanism by which other module (I/O, memory etc.) may interrupt the normal processing of the processor. The most common classes of interrupts are:

Program: Generated by some condition that occurs as a result of an instruction execution, such as arithmatic overflow, division by zero, attempt to execute

	an illegal machine instruction, and reference outside the user's allowed memory space.
Timer:	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O:	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure:	Generated by a failure such as power failure or memory parity error.

The issue of interrupts is discussed in later module, but we need to introduce the concept of interrupt now to understand more clearly the nature of instruction cycle.

Interupts are provided primarily as a way to improve processing effeciency. For example, most external devices are much slower than the processor. With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

For I/O operation, say an output operation, like printing some information by a printer. Printer is much slower device than the CPU. The CPU puts some information on the output buffer. While printer is busy printing these information from output buffer, CPU is lying idle. During this time CPU can perform some other task which does not involve the memory bus.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service the particular I/O device (known as an interrupt handler), and resuming the original execution after the device is serviced.

From the point of view of the user program, an interrupt is just that : ***an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes***.

To accommodate interrupts, an interrupt cycle is added to the instruction cycle, which is shown in the Figure 5.5. In the interrupt cycle, the processor checks if any interrupt have occurred, indicated by the presence of an interrupt signals. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

1. It suspends the execution of the current program being executed and saves its contents. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.

2. It sets the program counter to the starting address of an interrupt handler routine.

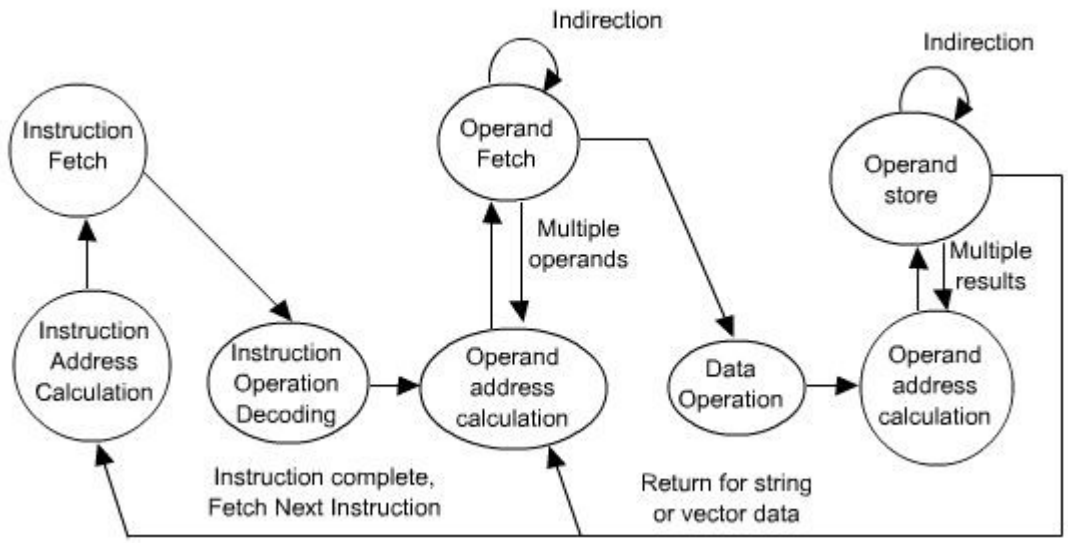


Figure 5.4: Instruction cycle state diagram.

Processor Organization

There are several components inside a CPU, namely, ALU, control unit, general purpose register, Instruction registers etc. Now we will see how these components are organized inside CPU. There are several ways to place these components and inteconnect them. One such organization is shown in the Figure 5.6.

In this case, the arithmatic and logic unit (ALU), and all CPU registers are connected via a single common bus. This bus is internal to CPU and this internal bus is used to transfer the information between different components of the CPU. This organization is termed as single bus organization, since only one internal bus is used for transferring of information between different components of CPU. We have external bus or buses to CPU also to connect the CPU with the memory module and I/O devices. The external memory bus is also shown in the Figure 5.6 connected to the CPU via the memory data and address register **MDR** and **MAR**.

The number and function of registers R0 to R(n-1) vary considerably from one machine to another. They may be given for general-purpose for the use of the programmer. Alternatively, some of them may be dedicated as special-purpose registers, such as **index register** or **stack pointers**.

In this organization, two registers, namely Y and Z are used which are transperant to the user. Programmer can not directly access these two registers. These are used as input and output buffer to the ALU which will be used in ALU operations. They will be used by CPU as temporary storage for some instructions.

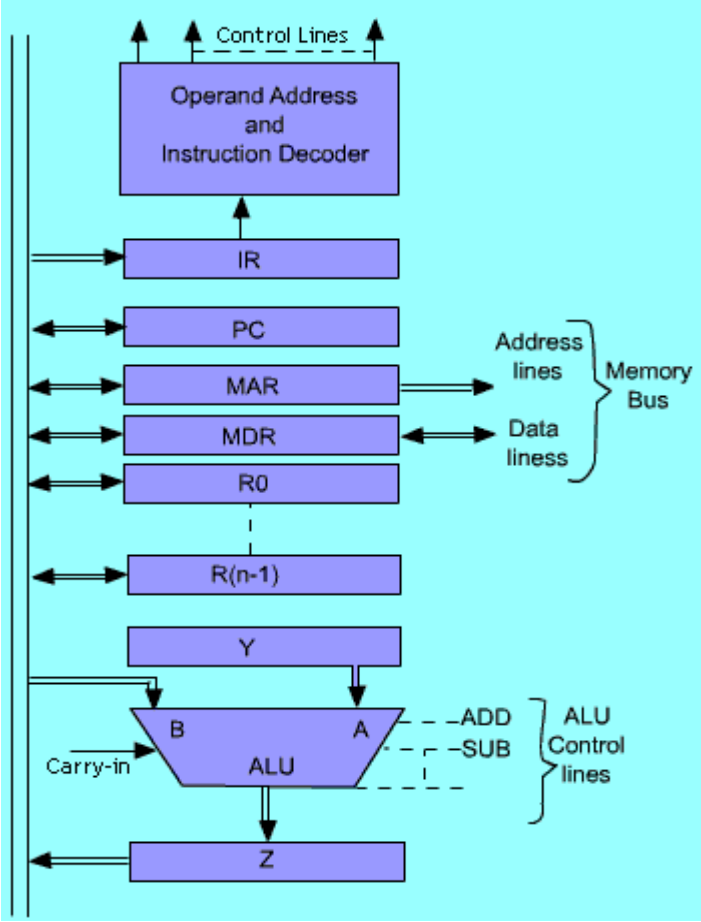


Figure 5.6 : Single bus organization of the data path inside the CPU

For the execution of an instruction, we need to perform an instruction cycle. An instruction cycle consists of two phase,

- Fetch cycle and
- Execution cycle.

Most of the operation of a CPU can be carried out by performing one or more of the following functions in some prespecified sequence:

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

Now we will examine the way in which each of the above functions is implemented in a computer.
Fetching a Word from Memory:

Information is stored in memory location indentified by their address. To fetch a word from memory, the CPU has to specify the address of the memory location where this information is stored and request a Read operation. The information may include both, the data for an operation or the instruction of a program which is available in main memory.

To perform a memory fetch operation, we need to complete the following tasks:

The CPU transfers the address of the required memory location to the Memory Address Register (MAR).

The MAR is connected to the memory address line of the memory bus, hence the address of the required word is transfered to the main memory.

Next, CPU uses the control lines of the memory bus to indicate that a Read operation is initiated. After issuing this request, the CPU waits until it receives an answer from the memory, indicating that the requested operation has been completed.

This is accomplished by another control signal of memory bus known as Memory-Function-Complete (MFC).

The memory set this signal to 1 to indicate that the contents of the specified memory location are available in memory data bus.

As soon as MFC signal is set to 1, the information available in the data bus is loaded into the Memory Data Register (MDR) and this is available for use inside the CPU.

As an example, assume that the address of the memory location to be accessed is kept in register R2 and that the memory contents to be loaded into register R1. This is done by the following sequence of operations:

1. MAR ← [R2] 2. Read
3. Wait for MFC signal 4. R1 ← [MDR]

The time required for step 3 depends on the speed of the memory unit. In general, the time required to access a word from the memory is longer than the time required to perform any operation within the CPU.

The scheme that is used here to transfer data from one device (memory) to another device (CPU) is referred to as an asynchronous transfer.

This asynchronous transfer enables transfer of data between two independent devices that have different speeds of operation. The data transfer is synchronised with the help of some control signals. In this example, Read request and MFC signal are doing the synchronization task.

An alternative scheme is synchronous transfer. In this case all the devices are controlled by a common clock pulse (continously running clock of a fixed frequency). These pulses provide common timing signal to the CPU and the main memory. A memory operation is completed during every clock period. Though the synchronous data transfer scheme leads to a simpler implementation, it is difficult to accommodate devices with widely varying speed. In such
cases, the duration of the clock pulse will be synchronized to the slowest device. It reduces the speed of all the devices to the slowest one.

Storing a word into memory

The procedure of writing a word into memory location is similar to that for reading one from memory. The only difference is that the data word to be written is first loaded into the MDR, the write command is issued.

As an example, assumes that the data word to be stored in the memory is in register R1 and that the memory address is in register R2. The memory write operation requires the following sequence:

1. MAR ← [R2]
2. MDR ← [R1]
3. Write
4. Wait for MFC

- In this case step 1 and step 2 are independent and so they can be carried out in any order. In fact, step 1 and 2 can be carried out simultaneously, if this is allowed by the architecture, that is, if these two data transfers (memory address and data) do not use the same data path.

In case of both memory read and memory write operation, the total time duration depends on wait for the MFC signal, which depends on the speed of the memory module.

There is a scope to improve the performance of the CPU, if CPU is allowed to perform some other operation while waiting for MFC signal. During the period, CPU can perform some other instructions which do not require the use of MAR and MDR.

Register Transfer Operation

Register transfer operations enable data transfer between various blocks connected to the common bus of CPU. We have several registers inside CPU and it is needed to transfer information from one register another. As for example during memory write operation data from appropriate register must be moved to MDR.

Since the input output lines of all the register are connected to the common internal bus, we need appropriate input output gating. The input and output gates for register R_i are controlled by the signal $R_{i\,in}$ and $R_{i\,out}$ respectively.

Thus, when $R_{i\,in}$ set to 1 the data available in the common bus is loaded into R_i . Similarly when, $R_{i\,out}$ is set to 1, the contents of the register R_i are placed on the bus. To transfer data from one register to other register, we need to generate the appropriate register gating signal.

For example, to transfer the contents of register R_1 to register R_2, the following actions are needed:

- Enable the output gate of register R_1 by setting R_{1out} to 1.
  -- This places the contents of R_1 on the CPU bus.
- Enable the input gate of register R_2 by setting $R_{2\,in}$ to 1.
  -- This loads data from the CPU bus into the register R_2.

Performing the arithmetic or logic operation:

- Generally ALU is used inside CPU to perform arithmetic and logic operation. ALU is a combinational logic circuit which does not have any internal storage.

Therefore, to perform any arithmetic or logic operation (say binary operation) both the input should be made available at the two inputs of the ALU simultaneously. Once both the inputs are available then appropriate signal is generated to perform the required operation.

We may have to use temporary storage (register) to carry out the operation in ALU .

The sequence of operations that have to carried out to perform one ALU operation depends on the organization of the CPU. Consider an organization in which one of the operand of ALU is stored in some temporary register Y and other operand is directly taken from CPU internal bus. The result of the ALU operation is stored in another temporary register Z. This organization is shown in the Figure 5.7.

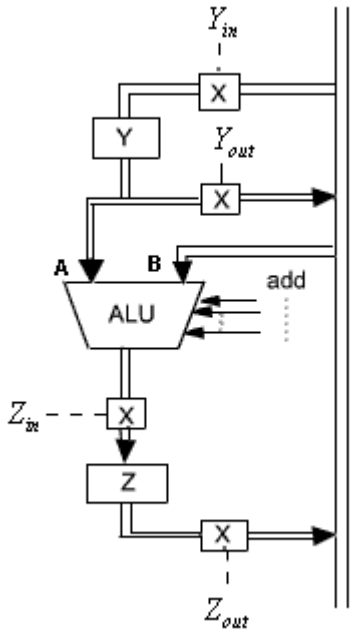


Figure 5.7 : Organization for Arithmatic & Logic Operation.

Therefore, the sequence of operations to add the contents of register R_1 to register R_2 and store the result in register R3 should be as follows:

1. R_{1out}, Y_{in}
2. R_{2out}, Add, Z_{in}
3. Z_{out}, $R3_{in}$

In step 2 of this sequence, the contents of register R_2 are gated to the bus, hence to input ?B of the ALU which is directly connected to the bus. The contents of register Y are always available at input A of ALU. The function performed by the ALU depends on the signal applied to the ALU control lines. In this example, the Add control line of ALU is set to 1, which indicate the addition operation and the output of ALU is the sum of the two numbers at input A and B. The sum is loaded into register Z, since the input gate is enabled (Z_{in}). In step 3, the contents of register Z are transferred to the destination register R_3.

Question: Whether step 2 and step 3 can be carried out simultaneously.

Multiple Bus Organization

Till now we have considered only one internal bus of CPU. The single-bus organization, which is only one of the possibilities for interconnecting different building blocks of CPU.

An alternative structure is the two bus structure, where two different internal buses are used in CPU. All register outputs are connected to bus A, add all registered inputs are connected to bus B.

There is a special arrangement to transfer the data from one bus to the other bus. The buses are connected through the bus tie G. When this tie is enabled data on bus A is transfer to bus B. When G is disabled, the two buses are electrically isolated.

Since two buses are used here the temporary register Z is not required here which is used in single bus organization to store the result of ALU. Now result can be directly transferred to bus B, since one of the inputs is in bus A. With the bus tie disabled, the result can directly be transferred to destination register. A simple two bus structure is shown in the Figure 5.8.

For example, for the operation, [R3] $\leftarrow$ [R1] + [R2] can now be performed as

1. $R1_{out}$, G_{enable}, Y_{in}
2. $R2_{out}$, Add, ALU_{out}, $R3_{in}$

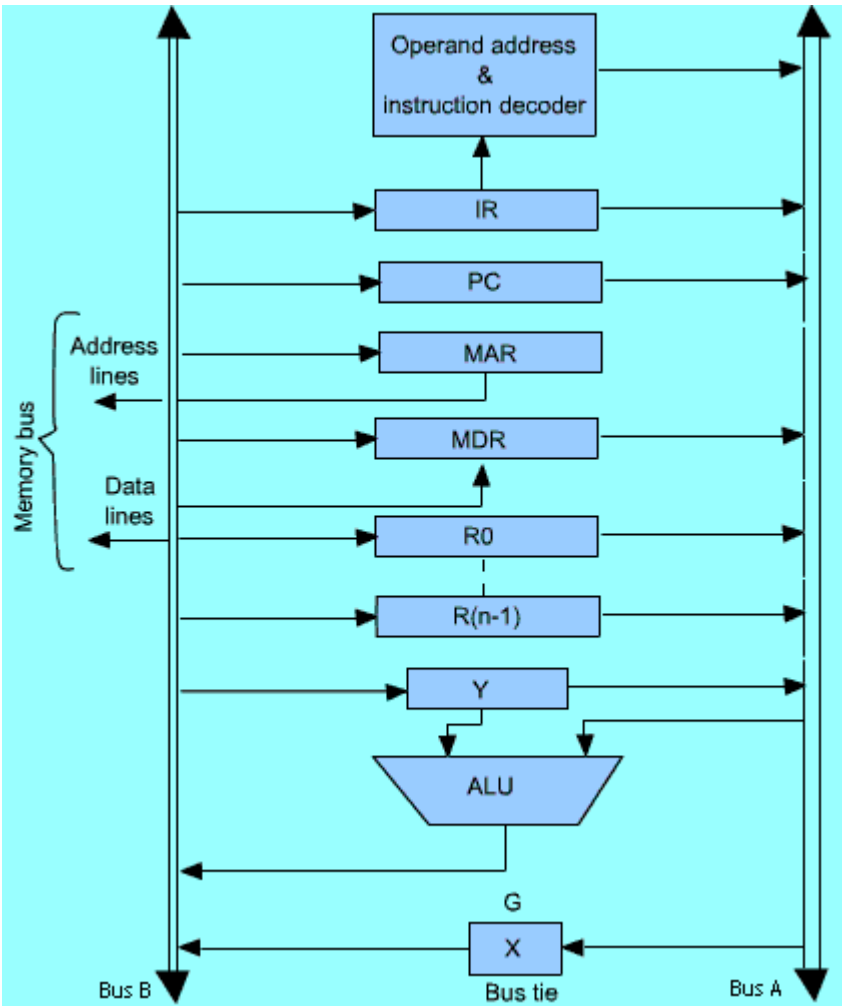


Figure 5.8 : Two bus structure

In this case source register R2 and destination register R3 has to be different, because the two operations $R2_{in}$ and $R2_{out}$ can not be performed together. If the registers are made of simple latches then only we have the restriction.

We may have another CPU organization, where three internal CPU buses are used. In this organization each bus connected to only one output and number of inputs. The elimination of the need for connecting more than one output to the same bus leads to faster bus transfer and simple control.

A simple three-bus organization is shown in the figure 5.9.

A multiplexer is provided at the input to each of the two working registers A and B, which allow them to be loaded from either the input data bus or the register data bus.

In the diagram, a possible interconnection of three-bus organization is presented, there may be different interconnections possible.

In this three bus organization, we are keeping two input data buses instead of one that is used in two bus organization.

Two separate input data buses are present ? one is for external data transfer, i.e. retrieving from memory and the second one is for internal data transfer that is transferring data from general purpose register to other building block inside the CPU.

Like two bus organization, we can use bus tie to connect the input bus and output bus. When the bus tie is enable, the information that is present in input bus is directly transferred to output bus. We may use one bus tie G1 between input data bus and ALU output bus and another bus tie G2 between register data bus and ALU output data bus.

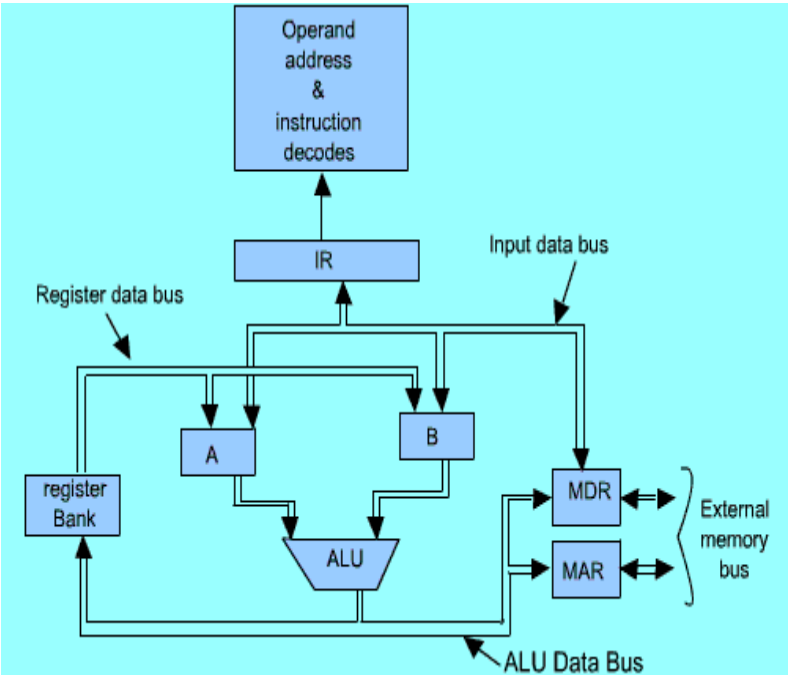


Figure 5.9 : Three Bus structure

Execution of a Complete Instructions:

We have discussed about four different types of basic operations:

- Fetch information from memory to CPU
- Store information to CPU register to memory
- Transfer of data between CPU registers.
- Perform arithmetic or logic operation and store the result in CPU registers.

To execute a complete instruction we need to take help of these basic operations and we need to execute these operation in some particular order.

As for example, consider the instruction : "Add contents of memory location NUM to the contents of register R1 and store the result in register R1." For simplicity, assume that the address NUM is given explicitly in the address field of the instruction .That is, in this instruction, direct addressing mode is used.
Execution of this instruction requires the following action :

1. Fetch instruction

2. Fetch first operand (Contents of memory location pointed at by the address field of the instruction)
3. Perform addition
4. Load the result into R1.

Following sequence of control steps are required to implement the above operation for the single-bus architecture that we have discussed in earlier section.

Steps	Actions
1.	PC_{out}, MAR_{in}, Read, Clear Y, Set carry -in to ALU, Add, Z_{in}
2.	Z_{out}, PC_{in}, Wait For MFC
3.	MDR_{out}, Ir_{in}
4.	Address-field- of-IR_{out}, MAR_{in}, Read
5.	$R1_{out}$, Y_{in}, Wait for MFC
6.	MDR_{out}, Add, Z_{in}
7.	Z_{out}, $R1_{in}$
8.	END

Instructruction execution proceeds as follows:

In Step1:

The instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a read request to memory.

To perform this task first of all the contents of PC have to be brought to internal bus and then it is loaded to MAR.To perform this task control circuit has to generate the PC_{out} signal and MAR_{in} signal.

After issuing the read signal, CPU has to wait for some time to get the MFC signal. During that time PC is updated by 1 through the use of the ALU. This is accomplished by setting one of the inputs to the ALU (Register Y) to 0 and the other input is available in bus which is current value of PC.

At the same time, the carry-in to the ALU is set to 1 and an add operation is specified.

In Step 2:

The updated value is moved from register Z back into the PC. Step 2 is initiated immediately after issuing the memory Read request without waiting for completion of memory function. This is possible, because step 2 does not use the memory bus and its execution does not depend on the memory read operation.

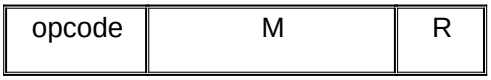
In Step 3:

Step3 has been delayed until the MFC is received. Once MFC is received, the word fetched from the memory is transfered to IR (Instruction Register), Because it is an instruction. Step 1 through 3 constitute the instruction fetch phase of the control sequence.

The instruction fetch portion is same for all instructions. Next step onwards, instruction execution phase takes place.

As soon as the IR is loaded with instruction, the instruction decoding circuits interprets its contents. This enables the control circuitry to choose the appropriate signals for the remainder of the control sequence, step 4 to 8, which we referred to as the execution phase. To design the control sequence of execution phase, it is needed to have the knowledge of the internal structure and instruction format of the PU. Secondly , the length of instruction phase is different for different instruction.

In this example , we have assumed the following instruction format :



i.e., **opcode:** Operation Code
M: Memory address for source
R: Register address for source/destination

In Step 5 :

The destination field of IR, which contains the address of the register R1, is used to transfer the contents of register R1 to register Y and wait for Memory function Complete. When the read operation is completed, the memory operand is available in MDR.

In Step 6 :

The result of addition operation is performed in this step.

In Step 7:

The result of addition operation is transfered from temporary register **Z** to the destination register **R1** in this step.

In step 8 :

It indicates the end of the execution of the instruction by generating End signal. This indicates completion of execution of the current instruction and causes a new fetch cycle to be started by going back to step 1.

Branching

With the help of branching instruction, the control of the execution of the program is transfered from one particular position to some other position, due to which the sequence flow of control is broken. Branching is accomplished by replacing the current contents of the PC by the branch address, that is, the address of the instruction to which branching is required.

Consider a branch instruction in which branch address is obtained by adding an offset **X**, which is given in the address field of the branch instruction, to the current value of PC.

Consider the following unconditional branch instruction

JUMP **X**

i.e., the format is

op- code	offset of jump

The control sequence that enables execution of an unconditional branch instruction using the single - bus organization is as follows :

Steps	Actions
1.	PCout, MAR_{in}, Read, Clear Y, Set Carry-in to ALU, Add ,Z_{in}
2.	Z_{out}, PC_{in}, Wait for MFC
3.	MDR_{out}, IR_{in}
4.	PC_{out}, Y_{in}
5.	Address field-of IR_{out}, Add, Z_{in}
6.	Z_{out}, PC_{in}
7.	End

Execution starts as usual with the fetch phase, ending with the instruction being loaded into the IR in step 3. To execute the branch instruction, the execution phase starts in step 4.

In Step 4

The contents of the PC are transferred to register Y.

In Step 5

The offset **X** of the instruction is gated to the bus and the addition operation is performed.

In Step 6

The result of the addition, which represents the branch address is loaded into the PC.

In Step 7

It generates the End signal to indicate the end of execution of the current instruction.

Consider now the conditional branch instruction instead of unconditional branch. In this case, we need to check the status of the condition codes, between step 3 and 4. i.e., before adding the offset value to the PC contents.

For example, if the instruction decoding circuitry interprets the contents of the IR as a branch on Negative(BRN) instruction, the control unit proceeds as follows:First the condition code register is checked. If bit N (negative) is equal to 1 , the control unit proceeds with step 4 trough step 7 of control sequence of unconditional branch instruction.

If , on the other hand , N is equal to 0, and End signal is issued .

This in effect , terminates execution of the branch instruction and causes the instruction immediately following in the branch instruction to be fetched when a new fetch operation is performed.

Therefore , the control sequence for the conditional branch instruction BRN can be obtained from the control sequence of an unconditional branch instruction by replacing the step 4 by

4. If $\overline{N}$ then End
   If N then PC_{out}, y_{in}

Most commonly need conditional branch instructions are

BNZ : Branch on not Zero

BZ : Branch on positive

BP : Branch on Positive

BNP : Branch on not Positive

BO : Branch on overflow

Design of Control Unit

To execute an instruction, the control unit of the CPU must generate the required control signal in the proper sequence. As for example, during the fetch phase, CPU has to generate PC_{out} signal along with other required signal in the first clock pulse. In the second clock pulse CPU has to generate PC_{in} signal along with other required signals. So, during fetch phase, the proper sequence for generating the signal to retrieve from and store to PC is PC_{out} and PC_{in}.

To generate the control signal in proper sequence, a wide variety of techniques exist. Most of these techniques, howeve, fall into one of the two categories,

1. **Hardwired Control**
2. **Microprogrammed Control.**

Hardwired Control

In this hardwired control techniques, the control signals are generated by means of hardwired circuit. The main objective of control unit is to generate the control signal in proper sequence.

Consider the sequence of control signal required to execute the ***ADD*** instruction that is explained in previous lecture. It is obvious that eight non-overlapping time slots are required for proper execution of the instruction represented by this sequence.

Each time slot must be at least long enough for the function specified in the corresponding step to be completed. Since, the control unit is implemented by hardwire device and every device is having a propagation delay, due to which it requires some time to get the *stable output signal* at the output port after giving the input signal. So, to find out the time slot is a complicated design task.

For the moment, for simplicity, let us assume that all slots are equal in time duration. Therefore the required controller may be implemented based upon the use of a counter driven by a clock.

Each state, or count, of this counter corresponds to one of the steps of the control sequence of the instructions of the CPU.

In the previous lecture, we have mentioned control sequence for execution of two instructions only (one is for add and other one is for branch). Like that we need to design the control sequence of all the instructions.

By looking into the design of the CPU, we may say that there are various instruction for add operation. As for example,

ADD	***NUM***	R_1	Add the contents of memory location specified by ***NUM*** to the contents of register R_1 . $R_1 \leftarrow R_1 + [NUM]$
ADD	R_2	R_1	Add the contents of register R_2 to the contents of register R_1.

$$R_1 \leftarrow R_1 + R_2$$

The control sequence for execution of these two ***ADD*** instructions are different. Of course, the fetch phase of all the instructions remain same.

It is clear that control signals depend on the instruction, i.e., the contents of the instruction register. It is also observed that execution of some of the instructions depend on the contents of condition code or status flag register, where the control sequence depends in conditional branch instruction.

Hence, the required control signals are uniquely determined by the following information:

- **Contents of the control counter.**
- **Contents of the instruction register.**
- **Contents of the condition code and other status flags.**

The external inputs represent the state of the CPU and various control lines connected to it, such as *MFC* status signal. The condition codes/ status flags indicates the state of the CPU. These includes the status flags like carry, overflow, zero, etc.

Control Unit Organization

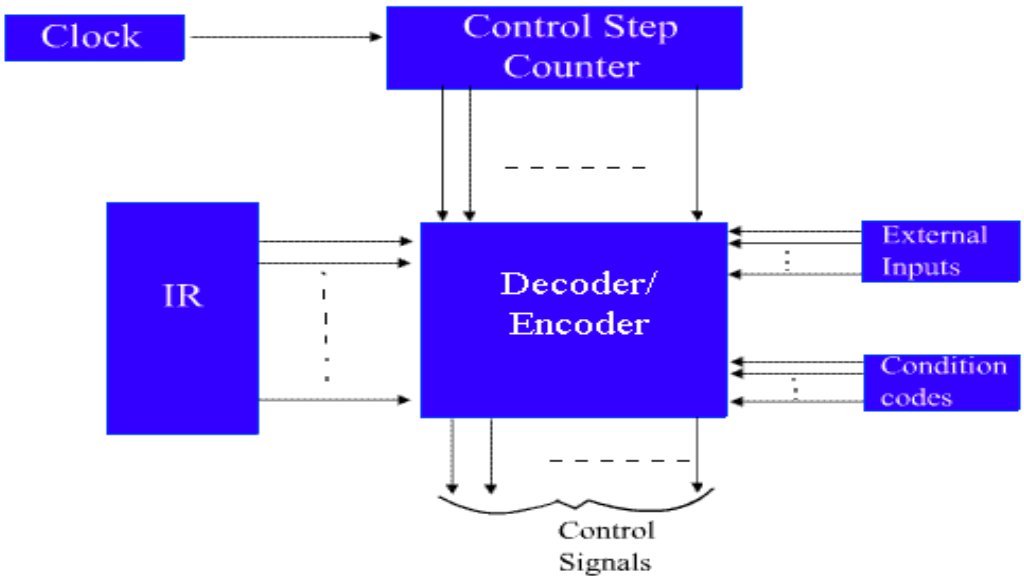


Figure 5.10: Organization of control unit

The structure of control unit can be represented in a simplified view by putting it in block diagram. The detailed hardware involved may be explored step by step. The simplified view of the control unit is given in the Figure 5.10.

The decoder/encoder block is simply a combinational circuit that generates the required control outputs depending on the state of all its input.

The decoder part of decoder/encoder part provide a separate signal line for each control step, or time slot in the control sequence. Similarly, the output of the instructor decoder consists of a separate line for each machine instruction loaded in the IR, one of the output line INS_1 to INS_m is set to 1 and all other lines are set to 0.

The detailed view of the control unit organization is shown in the Figure 5.11.

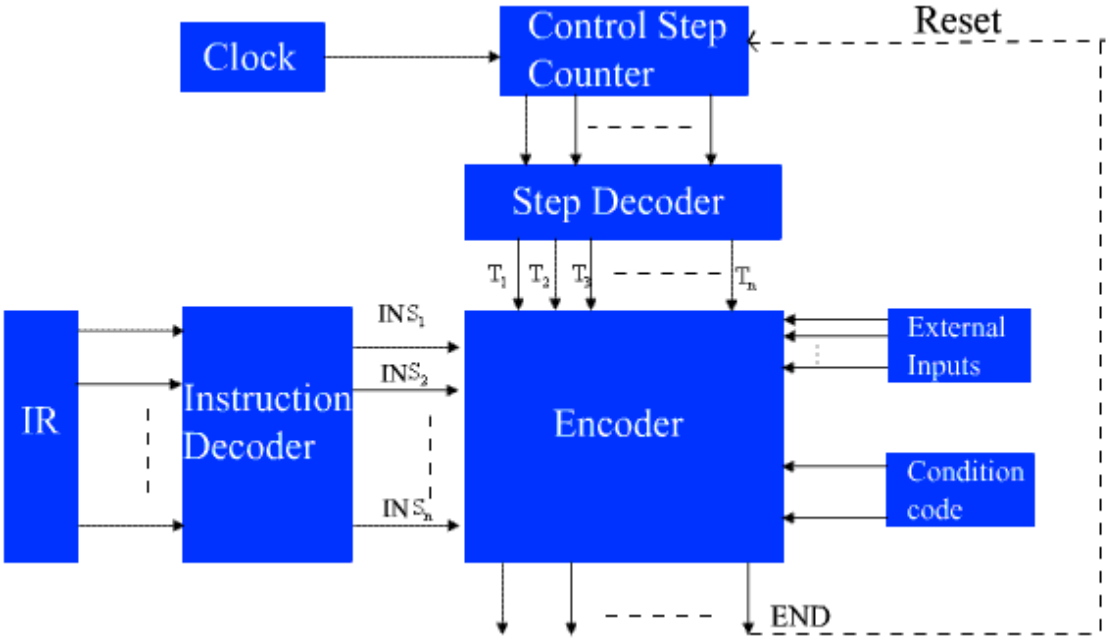


Figure 5.11: Detailed view of Control Unit organization

All input signals to the encoder block should be combined to generate the individual control signals.

In the previous section, we have mentioned the control sequence of the instruction,

"Add contents of memory location address in memory direct made to register R_1 (*ADD_MD*)",

"Control sequence for an unconditional branch instruction (*BR*)",

also, we have mentioned about Branch on negative (*BRN*).

Consder those three CPU instruction *ADD_MD*, *BR*, *BRN*.

It is required to generate many control signals by the control unit. These are basically coming out from the encoder circuit of the control signal generator. The control signals are: PC_{in}, PC_{out}, Z_{in}, Z_{out}, MAR_{in}, *ADD*, *END*, etc.

By looking into the above three instructions, we can write the logic function for Z_{in} as :

$$Z_{in} = T_1 + T_6 . ADD\_MD + T_5 . BR + T_5 . BRN + \ldots\ldots\ldots\ldots$$

For all instructions, in time step1 we need the control signal Z_{in} to enable the input to register Z_{in} time cycle T_6 of *ADD_MD* instruction, in time cycle T_5 of *BR* instruction and so on.

Similarly, the Boolean logic function for ADD signal is

$$ADD = T_1 + T_6 . ADD\_MD + T_5 . BR + \ldots\ldots\ldots\ldots$$

These logic functions can be implemented by a two level combinational circuit of AND and OR gates.

Similarly, the *END* control signal is generated by the logic function :

$$END = T_8 . ADD\_MD + T_7 . BR + (T_7 . N + T_4 . \overline{N}) . BRN + \ldots\ldots\ldots\ldots$$

This *END* signal indicates the end of the execution of an instruction, so this *END* signal can be used to start a new instruction fetch cycle by resetting the control step counter to its starting value.

The circuit diagram (Partial) for generating Z_{in} and *END* signal is shown in the Figure 5.12 and Figure 5.13 respectively.

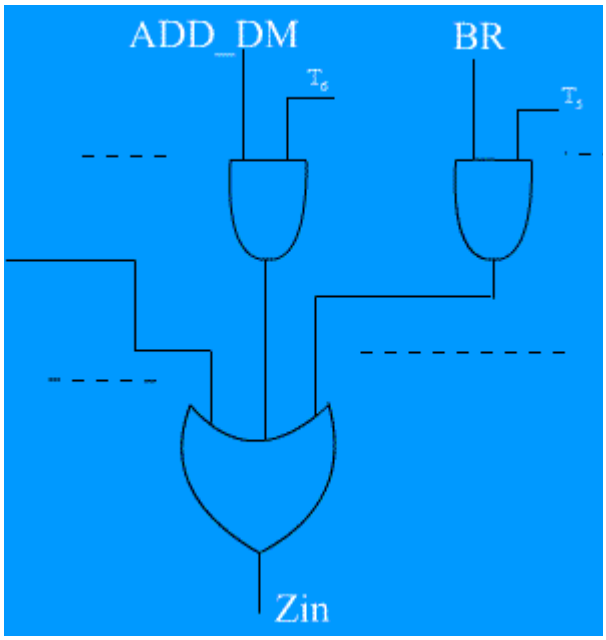


Figure 5.12: Generation of Z_{in} Control Signal

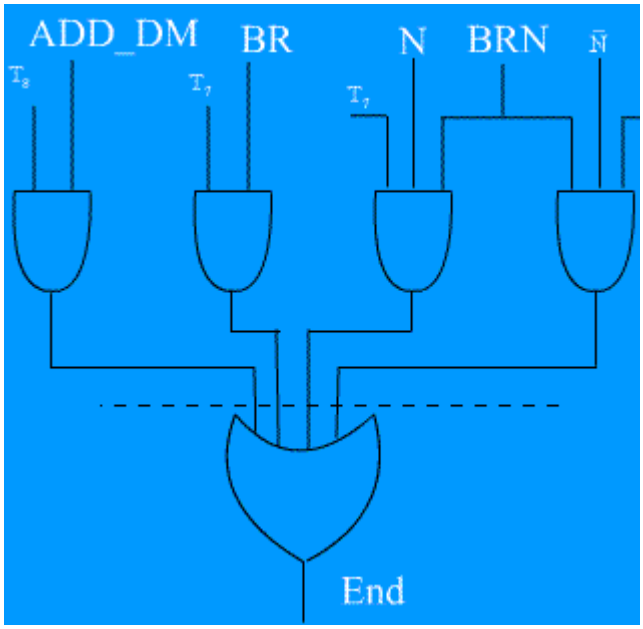


Figure 5.13: Generation of the *END* Control Signal

The signal *ADD_MD*, *BR*, *BRN* etc. are coming from instruction decoder circuits which depends on the contents of IR.

The signal T_1, T_2, T_3 etc are coming out from step decoder depends on control step counter.

The signal *N* (Negative) is coming from condition code register.

When wait for *MFC* (*WMFC*) signal is generated, then CPU does not do any works and it waits for an *MFC* signal from memory unit. In this case, the desired effect is to delay the initiation of the next control step until the *MFC* signal is received from the main memory. This can be incorporated by inhibiting the advancement of the control step counter for the required period.

Let us assume that the control step counter is controlled by a signal called ***RUN.***

By looking at the control sequence of all the instructions, the *WMFC* signal is generated as:

$$WMFC = T_2 + T_5 . ADD\_MD + \ldots\ldots\ldots\ldots$$

The *RUN* signal is generated with the help of *WMFC* signal and *MFC* signal. The arrangement is shown in the Figure 5.14.

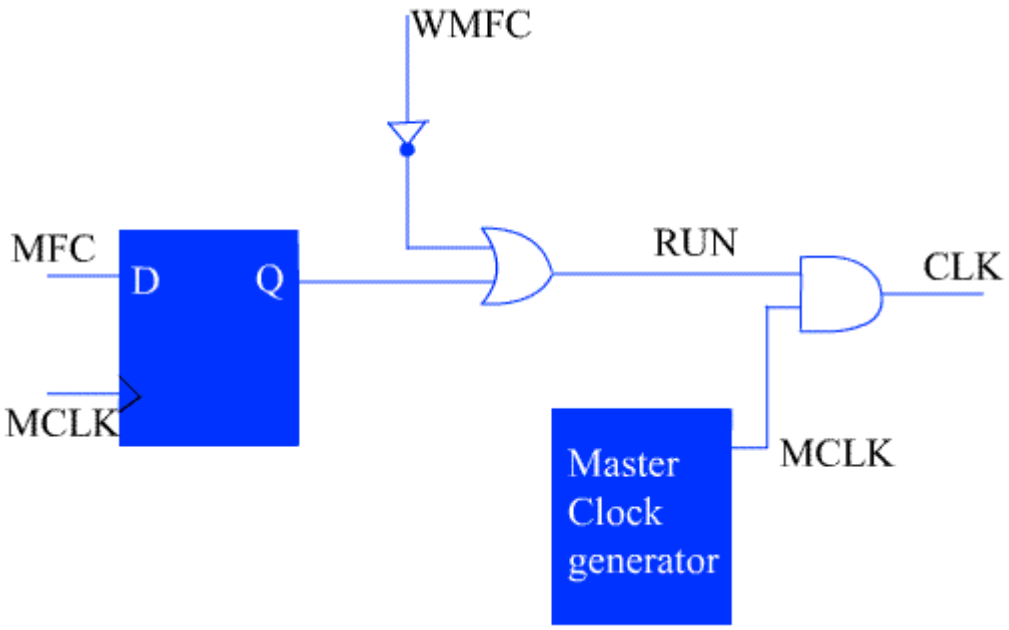


Figure 5.14: Generation of RUN signal

The *MFC* signal is generated by the main memory whose operation is independent of CPU clock. Hence *MFC* is an asynchronous signal that may arrive at any time relative to the CPU clock. It is possible to synchronized with CPU clock with the help of a D flip-flop.

When *WMFC* signal is high, then *RUN* signal is low. This run signal is used with the master clock pulse through an AND gate. When *RUN* is low, then the *CLK* signal remains low, and it does not allow to progress the control step counter.

When the *MFC* signal is received, the run signal becomes high and the *CLK* signal becomes same with the *MCLK* signal and due to which the control step counter progresses. Therefore, in the next control step, the *WMFC* signal goes low and control unit operates normally till the next memory access signal is generated.

The timing diagram for an instruction fetch operation is shown in the Figure 5.15.

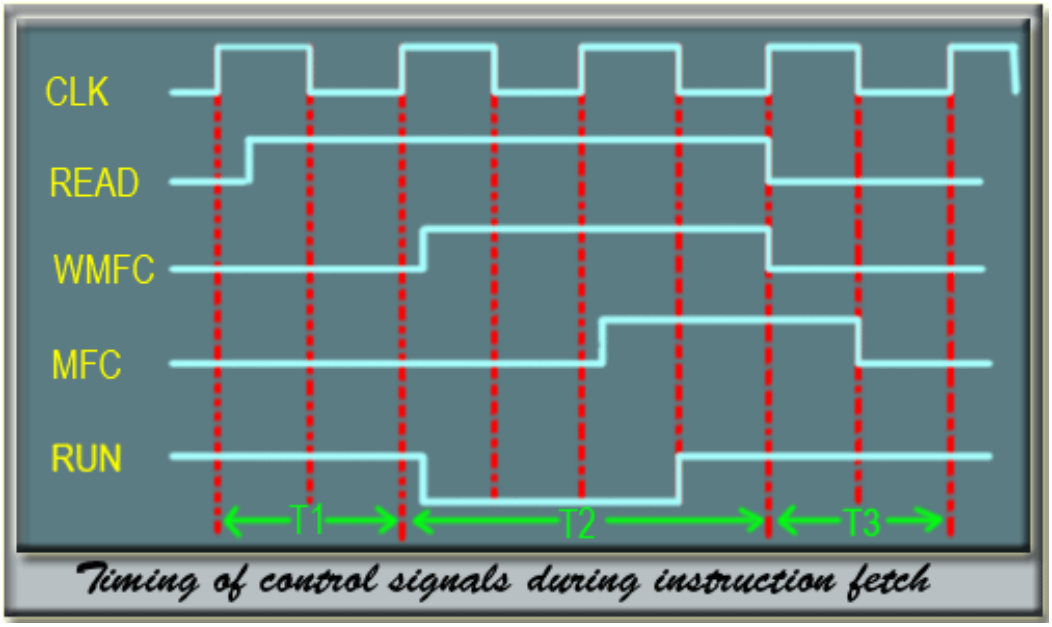


Figure 5.15: Timing of control signals during instruction fetch

Programmable Logic Array

In this discussion, we have presented a simplified view of the way in which the sequence of control signals needed to fetch and execute instructions may be generated.

It is observed from the discussion that as the number of instruction increases the number of required control signals will also increase.

In VLSI technology, structure that involve regular interconnection patterns are much easier to implement than the random connections.

One such regular structure is PLA (programmable logic array). PLAs are nothing but the arrays of AND gates followed by array of OR gates. If the control signals are expressed as sum of product form then with the help of PLA it can be implemented.

The PLA implementation of control unit is shown in the Figure 5.16.

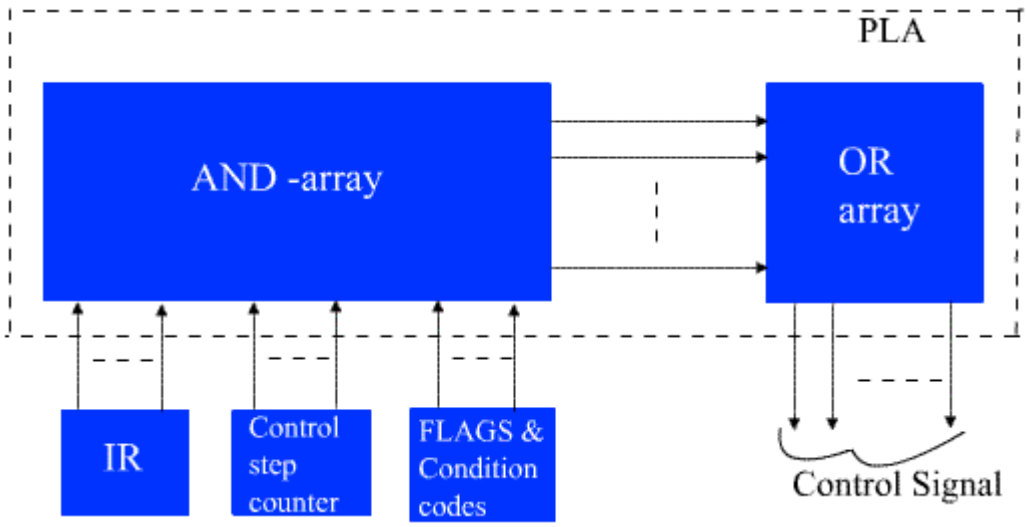


Figure 5.16 : Instruction cycle state diagram with interrupt

Microprogrammed Control

In hardwired control, we saw how all the control signals required inside the CPU can be generated using a state counter and a PLA circuit.

There is an alternative approach by which the control signals required inside the CPU can be generated . This alternative approach is known as microprogrammed control unit.

In microprogrammed control unit, the logic of the control unit is specified by a microprogram.

A microprogram consists of a sequence of instructions in a microprogramming language. These are instructions that specify microoperations.

A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.

The concept of microprogram is similar to computer program. In computer program the complete instructions of the program is stored in main memory and during execution it fetches the instructions from main memory one after another. The sequence of instruction fetch is controlled by program counter (PC) .

Microprogram are stored in microprogram memory and the execution is controlled by microprogram counter (μPC).

Microprogram consists of microinstructions which are nothing but the strings of 0's and 1's. In a particular instance, we read the contents of one location of microprogram memory, which is nothing but a microinstruction. Each output line (data line) of microprogram memory corresponds to one control signal. If the contents of the memory cell is 0, it indicates that the signal is not generated and if the contents of memory cell is 1, it indicates to generate that control signal at that instant of time.

First let me define the different terminologies that are related to microprogrammed control unit.

Control Word (CW) :

Control word is defined as a word whose individual bits represent the various control signal. Therefore each of the control steps in the control sequence of an instruction defines a unique combination of 0s and 1s in the CW.

A sequence of control words (CWs) corresponding to the control sequence of a machine instruction constitutes the microprogram for that instruction.

The individual control words in this microprogram are referred to as microinstructions.

The microprograms corresponding to the instruction set of a computer are stored in a aspecial memory which will be referred to as the microprogram memory. The control words related to an instructions are stored in microprogram memory.

The control unit can generate the control signals for any instruction by sequencially reading the CWs of the corresponding microprogram from the microprogram memory. To read the control word sequentially from the microprogram memory a microprogram counter (μPC) is needed.

The basic organization of a microprogrammed control unit is shown in the Figure 5.17.

The "starting address generator" block is responsible for loading the starting address of the microprogram into the μPC everytime a new instruction is loaded in the IR.

The μPC is then automatically incremented by the clock, and it reads the successive microinstruction from memory.

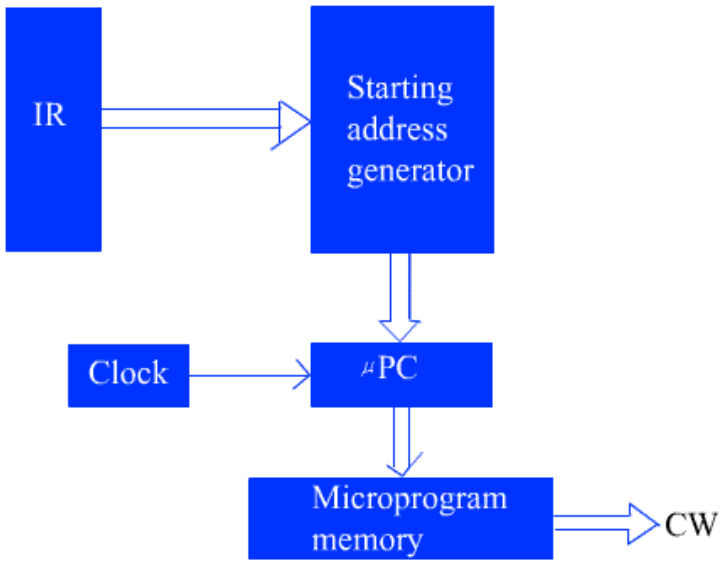


Figure 5.17: Basic organization of a microprogrammed control

Each microinstruction basically provides the required control signal at that time step. The microprogram counter ensures that the control signal will be delivered to the various parts of the CPU in correct sequence.

We have some instructions whose execution depends on the status of condition codes and status flag, as for example, the branch instruction. During branch instruction execution, it is required to take the decision between the alternative action.

To handle such type of instructions with microprogrammed control, the design of control unit is based on the concept of conditional branching in the microprogram. For that it is required to include some conditional branch microinstructions.

In conditional microinstructions, it is required to specify the address of the microprogram memory to which the control must direct. It is known as branch address. Apart from branch address, these microinstructions can specify which of the states flags, condition codes, or possibly, bits of the instruction register should be checked as a condition for branching to take place.

To support microprogram branching, the organization of control unit should be modified to accommodate the branching decision.

To generate the branch address, it is required to know the status of the condition codes and status flag. To generate the starting address, we need the instruction which is present in IR. But for branch address generation we have to check the content of condition codes and status flag.

The organization of control unit to enable conditional branching in the microprogram is shown in the Figure 5.18.

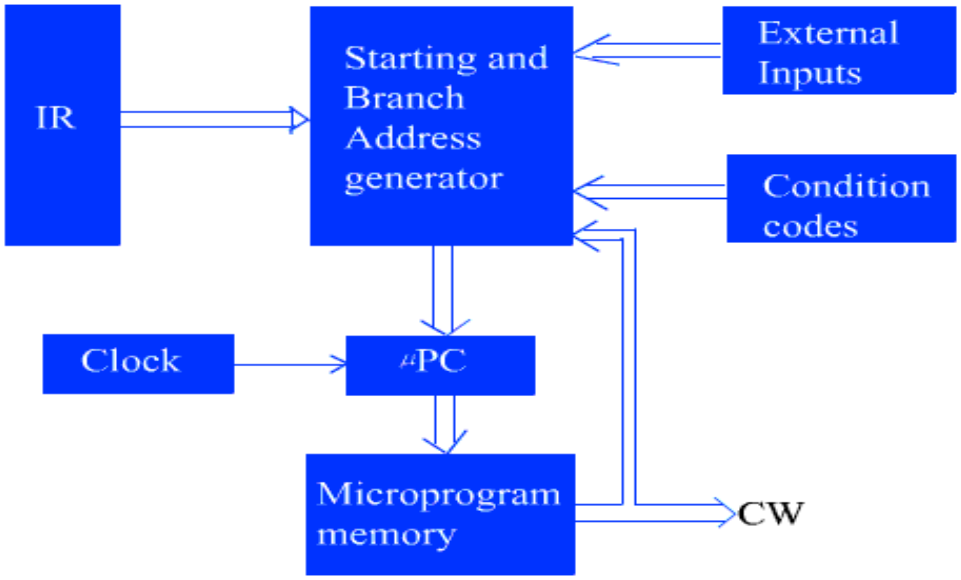


Figure 5.18: Organization of microprogrammed control with conditional branching.

The control bits of the microinstructions word which specify the branch conditions and address are fed to the "Starting and branch address generator" block.

This block performs the function of loading a new address into the μPC when the condition of branch instruction is satisfied.

In a computer program we have seen that execution of every instruction consists of two part - fetch phase and execution phase of the instruction. It is also observed that the fetch phase of all instruction is same.

In microprogrammed controlled control unit, a common microprogram is used to fetch the instruction. This microprogram is stored in a specific location and execution of each instruction start from that memory location.

At the end of fetch microprogram, the starting address generator unit calculate the appropriate starting address of the microprogram for the instruction which is currently present in IR. After the μPC controls the execution of microprogram which generates the appropriate control signal in proper sequence.

During the execution of a microprogram, the μPC is always incremented everytime a new microinstruction is fetched from the microprogram memory, except in the following situations :

1. When an End instruction is encountered, the μPC is loaded with the address of the first CW in the microprogram for the instruction fetch cycle.
1. When a new instruction is loaded into the IR, the μPC is loaded with the starting address of the microprogram for that instruction.
2. When a branch microinstruction is encountered, and the branch condition is satisfied, the μPC is loaded with the branch address.

Let us examine the contents of microprogram memory and how the microprogram of each instruction is stored or organized in microprogram memory. Consider the two example that are used in our previous lecture . First example is the control sequence for execution of the instruction "Add contents of memory location addressed in memory direct mode to register R1".

Steps	Actions
1.	PC_{out}, MAR_{in}, Read, Clear Y, Set carry-in to ALU, Add, Z_{in}
2.	Z_{out}, PC_{in}, Wait For MFC
3.	MDR_{out}, IR_{in}
4.	Address-field-of-IR_{out}, MAR_{in}, Read
5.	$R1_{out}$, Y_{in}, Wait for MFC
6.	MDR_{out}, Add, Z_{in}
7.	Z_{out}, $R1_{in}$
8.	END

Control sequence for Conditional Branch instruction (BRN) Branch on negative)

Steps	Actions
1.	PC_{out}, MAR_{in}, Read, Clear Y, Set Carry-in to ALU, Add , Z_{in}
2.	Z_{out}, PC_{in}, Wait for MFC
3.	MDR_{out}, IR_{in}
4.	PC_{out}, Y_{in}
5.	Address field-of IR_{out}, Add, Z_{in}
6.	Z_{out}, PC_{in}
7.	End

First consider the control signal required for fetch instruction , which is same for all the instruction, we are listing them in a particular order.

PC_{out}	MAR_{in}	Read	Clear Y	Set Carry to ALU	Add	Z_{in}	Z_{out}	PC_{in}	WMFC	MDR_{out}	IR_{in}

The control word for the first three steps of the above two instruction are : (which are the fetch cycle of each instruction as follows):

Step1	1	1	1	1	1	1	1	0	0	0	0	0	- - -
Step2	0	0	0	0	0	0	0	1	1	1	0	0	- - -
Step3	0	0	0	0	0	0	0	0	0	0	1	1	- - -

We are storing this three CW in memory location 0, 1 and 2. Each instruction starts from memory location 0. After executing upto third step, i.e., the contents of microprogram memory location 2, this control word stores the instruction in **IR**. The starting address generator circuit now calculate the starting address of the microprogram for the instruction which is available in **IR**.

Consider that the microprogram for add instruction is stored from memory location 50 of microprogram memory. So the partial contents from memory location 50 are as follows :

Location 50	0	1	1	0	0	0	0	0	0	0	0	0	- -	- -	- -
51	0	0	0	0	0	0	0	0	0	1	0	0	- -	- -	- -

and so on . . . .

The contents of the compile instruction is given below:

1 - PC_{in}, 2 - PC_{out}, 3 - MAR_{in}, 4 - Read, 5 - MDR_{out}, 6 - IR_{in}, 7 - $address_{out}$, 8 - Y_{in}, 9 - Clear Y, 10 - Carry-in, 11 - add, 12 - Z_{in}, 13 - Z_{out}, 14 -RI_{out}, 15 - RI_{in}, 16 - WMFC, 17 - END.

Memory Location	---------	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
0	----------	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	----------
1		1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	----------
2		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	---------
| |																			
50		0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	
51		0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	
52		0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	
53		0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	
54		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

When the microprogram executes the End microinstruction of an instruction, then it generates the End control signal. This End control signal is used to load the μPC with the starting address of fetch instruction (In our case it is address 0 of microprogram memory). Now the CPU is ready to fetch the next instruction from main memory.

From the discussion, it is clear that microprograms are similar to computer program, but it is in one level lower, that's why it is called microprogram.

For each instruction of the instruction set of the CPU, we will have a microprogram.

While executing a computer program, we fetch instruction by instruction from main memory which is controlled by program counter(PC).

When we fetch an instruction from main memory, to execute that instruction , we execute the microprogram for that instruction. Microprograms are nothing but the collection of microinstrctions. These microinstructions will be fetched from microprogram memory one after another and its

sequence is maintained by μPC. Fetching of microinstruction basically provides the required control signal at that time instant.

In the previous discussion, to design a micro programmed control unit, we have to do the following:

- For each instruction of the CPU, we have to write a microprogram to generate the control signal. The microprograms are stored in microprogram memory (control store). The starting address of each microprogram are known to the designer
- Each microprogram is the sequence of microintructions. And these microinstructions are executed in sequence. The execution sequence is maintained by microprogram counter.
- Each microinstructions are nothing but the combination of 0?s and 1?s which is known as control word. Each position of control word specifies a particular control signal. 0 on the control word means that a low signal value is generated for that control signal at that particular instant of time, similarly 1 indicates a high signal.
- Since each machine instruction is executed by a corresponding micro routine, it follows that a starting address for the micro routine must be specified as a function of the contents of the instruction register (IR).
- To incorporate the branching instruction, i.e., the branching within the microprogram, a branch address generator unit must be included. Both unconditional and conditional branching can be achieved with the help of microprogram. To incorporate the conditional branching instruction, it is required to check the contents of condition code and status flag.

Microprogramed controlled control unit is very much similar to CPU. In CPU the PC is used to fetch instruction from the main memory, but in case of control unit, microprogram counter is used to fetch the instruction from control store.

But there are some differences between these two. In case of fetching instruction from main memory, we are using two signals MFC and WMFC. These two signals are required to synchronize the speed between CPU and main memory. In general, main memory is a slower device than the CPU.

In microprogrammed control the need for such signal is less obvious. The size of control store is less than the size of main memory. It is possible to replace the control store by a faster memory, where the speed of the CPU and control store is almost same.

Since control store are usually relatively small, so that it is feasible to speed up their speed through costly circuits.

If we can implement the main memory by a faster device then it is also possible to eliminate the signals MFC & WMFC. But, in general, the size of main memory is very big and it is not economically feasible to replace the whole main memory by a faster memory to eliminate MFC & WMFC.

Grouping of control signals:

It is observed that we need to store the information of each control signal in control store. The status of a particular control signal is either high or low at a particular instant of time.

It is possible to reserve one bit position for each control signal. If there are *n* control signals in a CPU, them the length of each control signal is *n* . Since we have one bit for each control signal, so a

large number of resources can be controlled with a single microinstruction. This organization of microinstruction is known as horizontal organization.

If the machine structure allows parallel uses of a number of resources, then horizontal organization has got advantage. Since more number of resources can be accessed parallel, the operating speed is also more in such organization. In this situation, horizontal organization of control store has got advantage.

If more number of resources can be accessed simultaneously, than most of the contents of control store is 0. Since the machine architecture does not provide the parallel access of resources, so simultaneously we cannot generate the control signal. In such situation, we can combine some control signals and group them together. This will reduce the size of control word. If we use compact code to specify only a small number of control functions in each microinstruction, then it is known as vertical organization of microinstruction.

In case of horizontal organization, the size of control word is longer, which is in one extreme point and in case of vertical organization, the size of control word is smaller, which is in other extreme.

In case of horizontal organization, the implementation is simple, but in case of vertical organization, implementation complexity increases due to the required decoder circuits. Also the complexity of decoder depends on the level of grouping and encoding of the control signal.

Horizontal and Vertical organization represent the two organizational extremes in microprogrammed control. Many intermediate schemes are also possible, where the degree of encoding is a design parameter.

We will explain the grouping of control signal with the help of an example. Grouping of control signals depends on the internal organization of CPU.

Assigning individual bits to each control signal is certain to lead to long microinstruction, since the number of required control signals is normally large.

However, only a few bits are set to 1 and therefore used for active gating in any given microinstructions. This obviously results in low utilization of the available bit space.

If we group the control signal in some non-over lapping group then the size of control word reduces.

The single bus architecture of CPU is shown in the Figure 5.19.

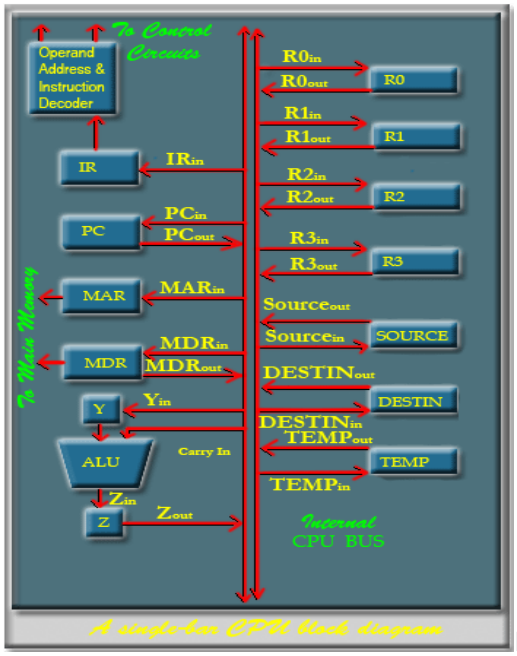


Figure 5.19: Single bus architecture of CPU

Click on Image To View Large Image

This CPU contains four general purpose registers *R0* , *R1* , *R2* and *R3* . In addition there are three other register called SOURCES, DESTIN and TEMP. These are used for temporary storage within the CPU and completely transparent to the programmer. A computer programmer cannot use these three registers.

For the proper functioning of this CPU, we need all together 24 gating signals for the transfer of information between internal CPU bus and other resources like registers.

In addition to these register gating signals, we need some other control signals which include the Read, Write, Clear Y, set carry in, WMFC, and End signal. (Here we are restricting the control signal for the case of discussion in reality, the number of signals are more).

It is also necessary to specify the function to be performed by ALU. Depending on the power of ALU, we need several control lines, one control signal for each function. Assume that the ALU that is used in the design can perform 16 different operation such as ADD, SUBSTRACT, AND, OR, etc. So we need 16 different control lines.

The above discussion indicates that 46(24+6+16) distinct signals are required. This indicates that we need 46 bits in each micro instructions, therefore the size of control word is 46.

Consider the microprogram pattern that is shown for the Add instruction. On an average 4 to 5 bits are set to 1 in each micro instruction and rest of the bits are 0. Therefore, the bit utilization is poor, and there is a scope to improve the utilization of bit.

If is observed that most signals are not needed simultaneously and many signals are mutually exclusive.

As for example, only one function of the ALU can be activated at a time. In out case we are considering 16 ALU operations. Instead of using 16 different signal for ALU operation, we can group

them together and reduce the number of control signal. From digital logic circuit, it is obvious that instead of 16 different signal, we can use only 4 control signal for ALU operation and them use a 4 X 16 decoder to generate 16 different ALU signals. Due to the use of a decoder, there is a reduction in the size of control word.

Another possibilities of grouping control signal is: A sources for data transfer must be unique, which means that it is not possible to gate the contents of two different registers onto the bus at the same time. Similarly Read Write signals to the memory cannot be activated simultaneously.

This observation suggests the possibilities of grouping the signals so that all signals that are mutually exclusive are placed in the same group. Thus a group can specify one micro operation at a time.

At that point we have to use a binary coding scheme to represent a given signal within a group. As for example, for 16 ALU function, four bits are enough to decode the appropriate function.

A possible grouping of the 46 control signals that are required for the above mention CPU is given in the Table 5.1.

Table 5.1: Grouping of the control signals

F1 (4 bits)	F2 (3 bits)	F3 (2 bits)	F4 (2 bits)	F5 (4 bits)
0000: No Transfer	**000:** No Transfer	**00:** No Transfer	**00:** No Transfer	**0000:** Add
0001: PCout	**001:** PCin	**01:** MARin	**01:** Yin	**0001:** Sub
0010: MDRout	**001:** IRin	**10:** MDRin	**10:** SOURCEin	**0010:** MULT
0011: Zout	**011:** Zin	**11:** TEMPin	**11:** DESTINin	**0011:** Div
0100: R0out	**100:** R0in			|
0101: R1out	**101:** R1in			|
0110: R2out	**110:** R2in			|
0111: R3out	**111:** R3in			|
1000: SOURCEout				|
1001: DESTINout				|
1010: TEMPout				|

1011: ADDRESSout				**1111:** XOR

F6 (2 bits)	F7 (1 bit)	F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
00: no action	**0:** no action	**0:** carry-in=0	**0:** no action	**0:** continue
01: read	**1:** clear Y	**1:** carry-in=1	**1:**WMFC	**1:** end
10: write				

A possible grouping of signal is shown here. There may be some other grouping of signal possible. Here all out- gating of registers are grouped into one group, because the contents of only one bus is allowed to goto the internal bus, otherwise there will be a conflict of data.

But the in-gating of registers are grouped into three different group. It implies that the contents of the bus may be stored into three different registers simultaneously transfer to MAR and Z. Due to this grouping, we are using 7 bits (3+2+2) for the in-gating signal. If we would have grouped then in one group, then only 4 bits would have been enough; but it will take more time during execution. In this situation, two clock cycles would have been required to transfer the contents of PC to MAR and Z.

Therefore, the grouping of signal is a critical design parameter. If speed of operation is also a design parameter, then compression of control word will be less.

In this grouping, 46 control signals are grouped into 10 different groups (*F1* , *F2* ,???., *F10*) and the size of control word is 21. So, the size of control word is reduced from 46 to 21, which is more than 50%.

For the proper decoding, we need the following decoder:

For group *F1* & *F5* : 4 X 16 decoder,

group *F2* : 3 X 8 decoder

group *F3,F4* & *F6* : 2 X 4 decoder

Microprogram Sequencing:

In microprogrammed controlled CU,

- Each machine instruction can be implemented by a microroutine.

- Each microroutine can be accessed initially by decoding the machine instruction into the starting address to be loaded into the PC.

Writing a microprogram for each machine instruction is a simple solution, but it will increase the size of control store.

We have already discussed that most machine instructions can operate in several addressing modes. If we write different microroutine for each addressing mode, then most of the cases, we are repeating some part of the microroutine.

The common part of the microroutine can be shared by several microroutine, which will reduse the size of control store. This results in a considerable number of branch microinstructions being needed to transfer control among various parts. So, it introduces branching capabilities within the microinstruction.

This indicates that the microprogrammed control unit has to perform two basic tasks:

- Microinstruction sequencing: Get the next microinstruction from the control memory.
- Microinstruction execution: Generate the control signals needed to execute the microinstruction.

In designing a control unit, these tasks must be considered together, because both affect the format of the microinstruction and the timing of control unit.

Design Consideration:

Two concerns are involved in the design of a microinstruction sequencing technique: the size of the microinstruction and the address generation time.

In executing a microprogram, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

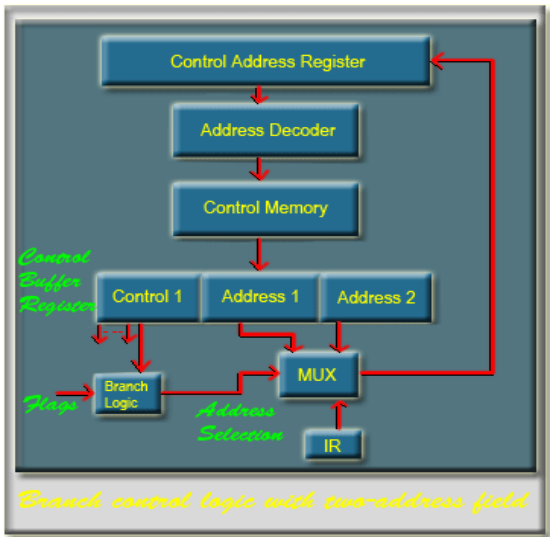
Sequencing Techniques:

Based on the current microinstruction, condition flags and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used and can be grouped them into three general categories:

- Two address fields
- Single address field
- Variable format.

Two Address fields:

The branch control logic with two-address field is shown in the Figure 5.20.



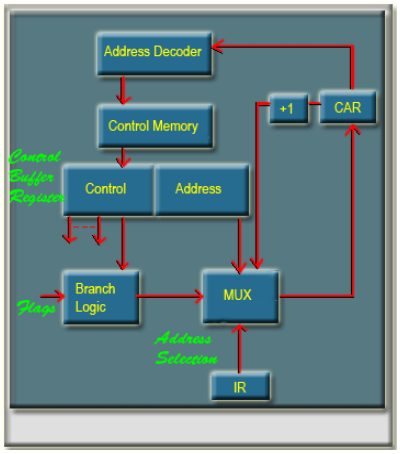
Click on Image To View Large Image

Figure 5.20: Branch control logic with two address field

A multiplier is provided that serves as a destination for both address fields and the instruction register. Based on an address selection input, the multiplexer selects either the opcode or one of the two addresses to the control address register (CAR). The CAR is subsequently decoded to produce the next microinstruction address. The address selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.

Single address field:

The two address approach is simple but it requires mere bits in the microinstruction. With some additional logic, savings can be achieved. The approach is shown in the Figure 5.21



Click on Image To View Large Image

Figure 5.21: Branch control logic with one address field

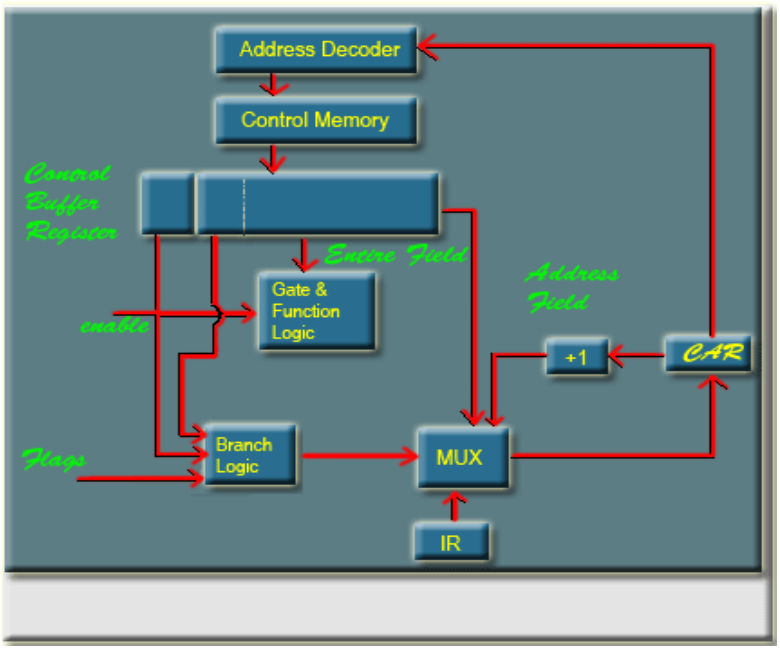
In this single address field branch control logic, the options for next address are as follows:

- Address field
- Instruction register code
- Next sequential address

The address selection signals determine which option to be selected. This approach reduce the number of address fields to one.

Variable format:

In variable format branch control logic one bit designates which format is being used. In one format, the remaining bits are used to active control signals. In the other format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is being specified. The approach is shown in the Figure 5.22.



Click on Image To View Large Image

Figure 5.22: Branch control logic with variable format

Address Generation:

We have looked at the sequencing problem from the point of view of format consideration and general logic requirements. Another viewpoint is to consider the various ways in which the next address can be derived or computed.

Various address generation Techniques	
Explicit	*Implicit*
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control

The address generation technique can be divided into two techniques: explicit & implicit.

In explicit technique, the address is explicitly available in the microinstruction.

In implicit technique, additional logic circuit is used to generate the address.

In two address field approach, signal address field or a variable format, various branch instruction can be implemented with the explicit approaches.

In implicit technique, mapping is required to get the address of next instruction. The opcode portion of a machine instruction must be mapped into a microinstruction address.

Pipeline Processor

In this Module, we have three lectures, viz.

1. **Introduction to Pipeline Processor**
2. **Performance Issues**
3. **Branching**

Pipelining

It is observed that organization enhancements to the CPU can improve performance. We have already seen that use of multiple registers rather than a single a accumulator, and use of cache memory improves the performance considerably. Another organizational approach, which is quite common, is instruction pipelining.

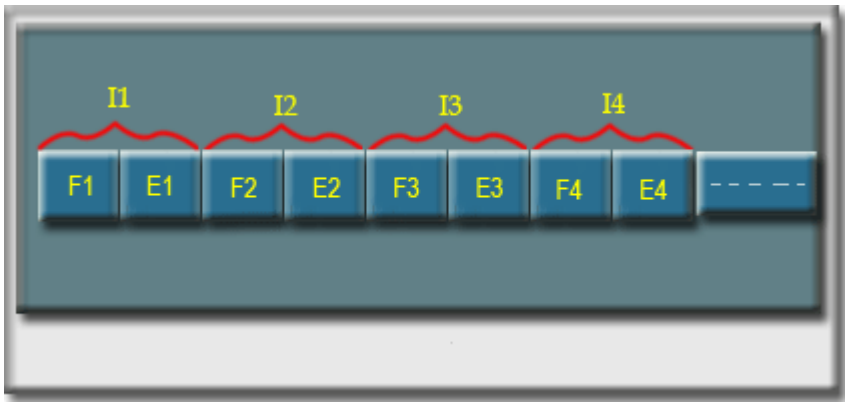
Pipelining is a particularly effective way of organizing parallel activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly line operation.

By laying the production process out in an assembly line, product at various stages can be worked on simultaneously. This process is also referred to as pipelining, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply the concept of instruction execution in pipeline, it is required to break the instruction in different task. Each task will be executed in different processing elements of the CPU.

As we know that there are two distinct phases of instruction execution: one is instruction fetch and the other one is instruction execution. Therefore, the processor executes a program by fetching and executing instructions, one after another.

Let F_i and E_i refer to the fetch and execute steps for instruction I_i. Execution of a program consists of a sequence of fetch and execute steps is shown in the Figure.



Now consider a CPU that has two separate hardware units, one for fetching instructions and another for executing them.

The instruction fetch by the fetch unit is stored in an intermediate storage buffer $B1$

The results of execution are stored in the destination location specified by the instruction.

For simplicity it is assumed that fetch and execute steps of any instruction can be completed in one clock cycle.

The operation of the computer proceeds as follows:

- In the first clock cycle, the fetch unit fetches an instruction (instruction I_1 , step F_1) and stored it in buffer B_1 at the end of the clock cycle.
- In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2).
- Meanwhile, the execution unit performs the operation specified by instruction I_1 which is already fetched and available in the buffer B_1 (step E_1).

- By the end of the second clock cycle, the execution of the instruction I_1 is completed and instruction I_2 is available.
- Instruction I_2 is stored in buffer B_1 replacing I_1, which is no longer needed.
- Step E_2 is performed by the execution unit during the third clock cycle, while instruction I_3 is being fetched by the fetch unit.
- Both the fetch and execute units are kept busy all the time and one instruction is completed after each clock cycle except the first clock cycle.
- If a long sequence of instructions is executed, the completion rate of instruction execution will be twice that achievable by the sequential operation with only one unit that performs both fetch and execute.

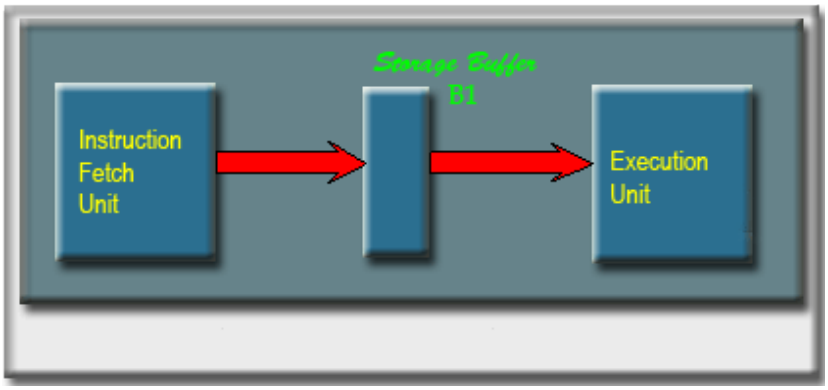


Figure 2: Hardware Organization

The pipeline execution of fetch and execution cycle is shown in the Figure 3.

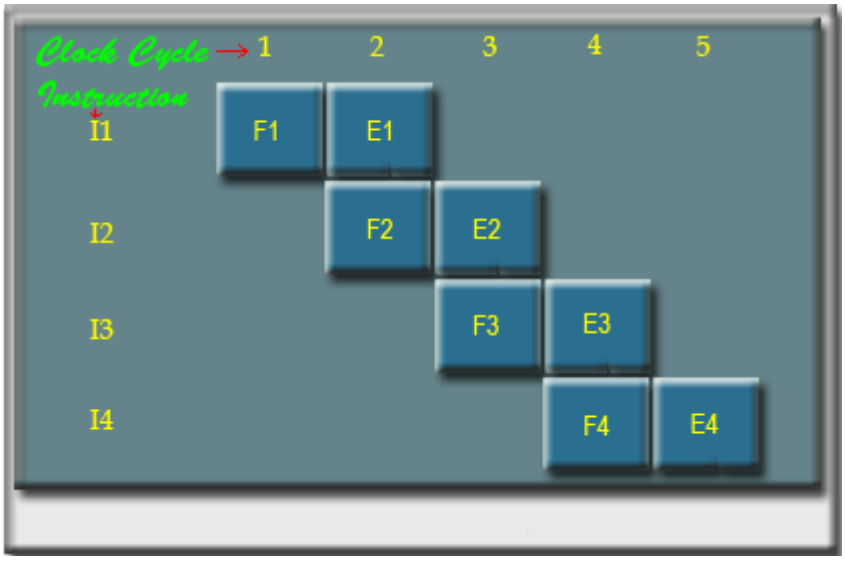


Figure 3: Pipeline Execution.

The processing of an instruction need not be divided into only two steps. To gain further speed up, the pipeline must have more stages.

Let us consider the following decomposition of the instruction execution:

- Fetch Instruction (FI): Read the next expected instruction into a buffer.
- Decode Instruction ((DI): Determine the opcode and the operand specifiers.
- Calculate Operand (CO): calculate the effective address of each source operand.
- Fetch Operands(FO): Fetch each operand from memory.
- Execute Instruction (EI): Perform the indicated operation.
- Write Operand(WO): Store the result in memory.

There will be six different stages for these six subtasks. For the sake of simplicity, let us assume the equal duration to perform all the subtasks. It the six stages are not of equal duration, there will be some waiting involved at various pipeline stages.

The timing diagram for the execution of instruction in pipeline fashion is shown in the Figure 4

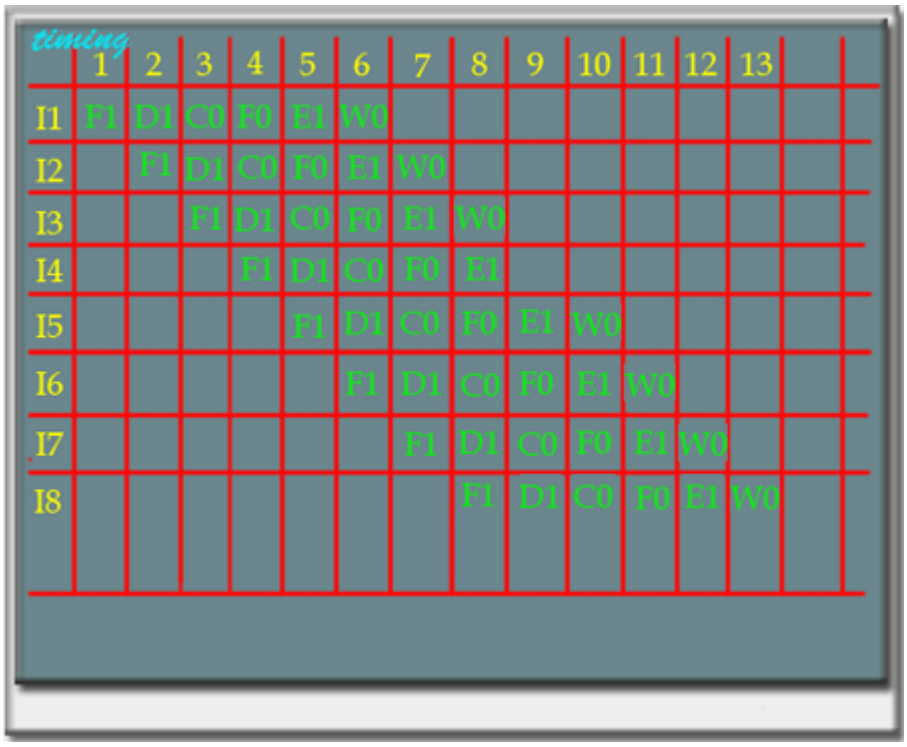


Figure 4: Timing sequence in pipeline execution.

From this timing diagram it is clear that the total execution time of 8 instructions in this 6 stages pipeline is 13-time unit. The first instruction gets completed after 6 time unit, and there after in each time unit it completes one instruction.

Without pipeline, the total time required to complete 8 instructions would have been 48 (6 X 8) time unit. Therefore, there is a speed up in pipeline processing and the speed up is related to the number of stages.

Pipeline Performance :

The cycle time τ of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline. The cycle time can be determined as

$$\tau = \max[\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

where

$\tau_m =$ maximum stage delay (delay through stage which experience the largest delay)
$k =$ number of stages in the instruction pipeline.
$d =$ time delay of a latch, needed to advance signals and data from one stage to the next.
In general, the time delay d is equivalent to a clock pulse and $\tau_m \gg d.$

Now suppose that *n* instructions are processed and these instructions are executed one after another. The total time required τ_k to execute all *n* instructions is

A total of *k* cycles are required to complete the execution of the first instruction, and the remaining *(n-1)* instructions require *(n-1)* cycles.

The time required to execute *n* instructions without pipeline is
$T_1 = nk\tau$
because to execute one instruction it will take $n\tau$ cycle.

The speed up factor for the instruction pipeline compared to execution without the pipeline is defined as:

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{\left[k+(n-1)\right]\tau} = \frac{nk}{k+(n-1)} = \frac{nk}{(k-1)+n}$$

In general, the number of instruction executed is much more higher than the number of stages in the pipeline So, the *n* tends to α , we have
$S_k = k,$

i.e. We have a *k* fold speed up, the speed up factor is a function of the number of stages in the instruction pipeline.

Though, it has been seen that the speed up is proportional to number of stages in the pipeline, but in practice the speed up is less due to some practical reason. The factors that **affect the pipeline performance is discussed next.**

Effect of Intermediate storage buffer:

Consider a pipeline processor, which process each instruction in four steps;

F: Fetch, Read the instruction from the memory

D: Decode, decode the instruction and fetch the source operand (S)

O: Operate, perform the operation

W: Write, store the result in the destination location.

The hardware organization of this four-stage pipeline processor is shown in the Figure 5.

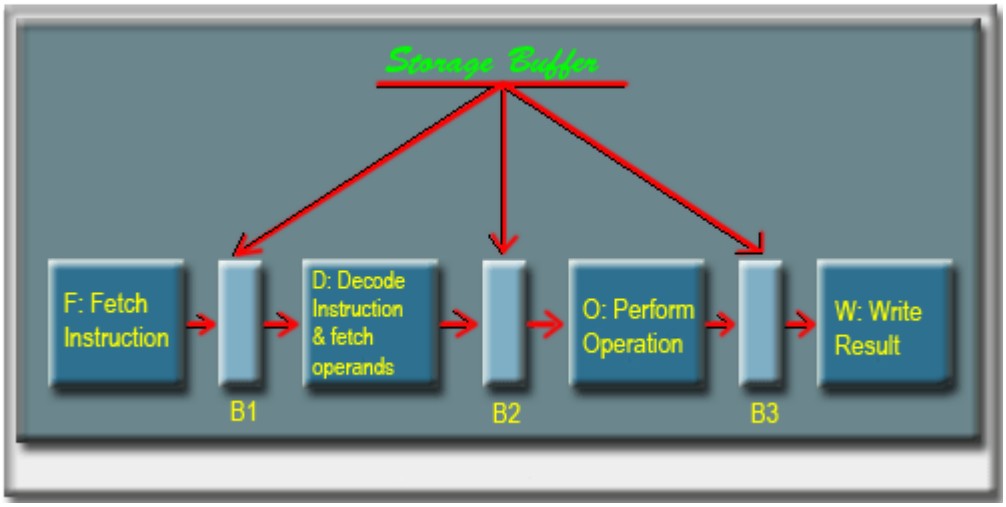


Figure 5: Four stage pipeline processor

In the preceding section we have seen that the speed up of pipeline processor is related to number of stages in the pipeline, i.e, the greater the number of stages in the pipeline, the faster the execution rate.

But the organization of the stages of a pipeline is a complex task and if affects the performance of the pipeline.

The problem related to more number of stages:

At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction.

The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages.

Apart from hardware organization, there are some other reasons which may effect the performance of the pipeline.

(A) Unequal time requirement to complete a subtask:

Consider the four-stage pipeline with processing step Fetch, Decode, Operand and write.

Although this may be sufficient for most operations, but some operations like divide may require more time to complete.
Figure 9.6 shows the effect of an operation that takes more than one clock cycle to complete an operation in operate stage.

Clock Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction									

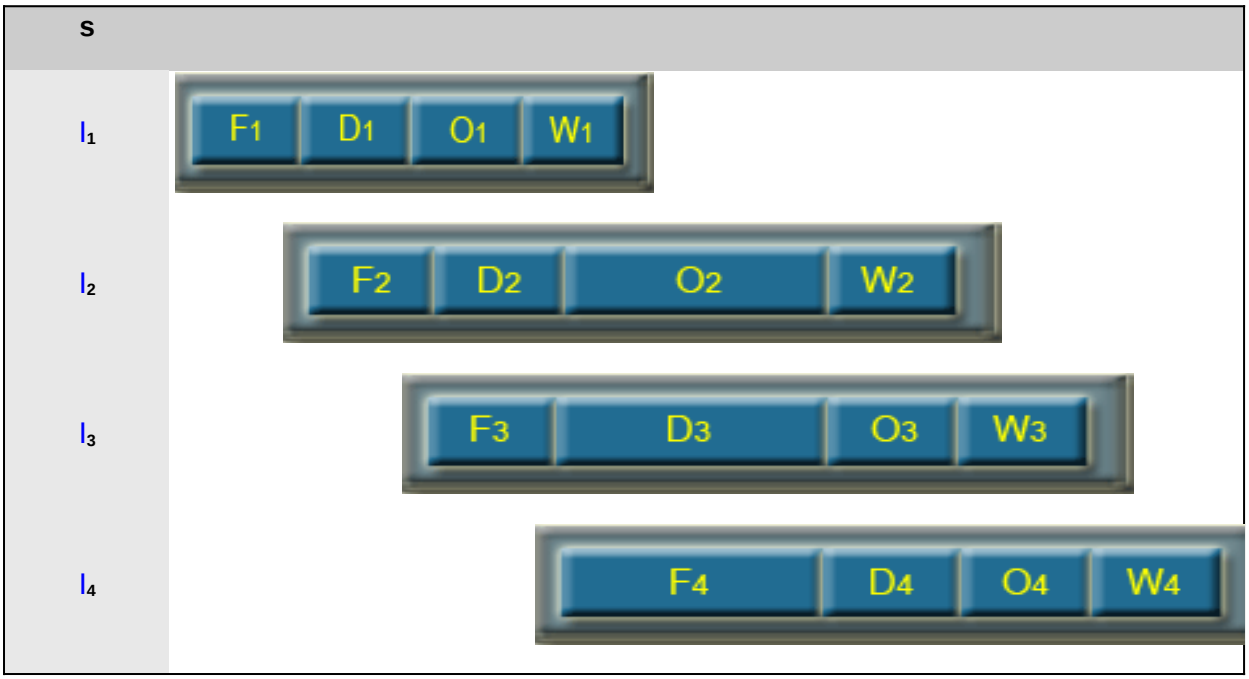


Figure 6: Operation takes more than one clock cycle

The operate stage for instruction I_2 takes 3 clock cycle to perform the specified operation. Clock cycle 4 to 6 required to perform this operation and so write stage is doing nothing during the clock cycle 5 and 6, because no data is available to write.

Meanwhile, the information in buffer B2 must remain intake until the operate stage has completed its operation.

This means that stage 2 and stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten by a new fetch instruction.

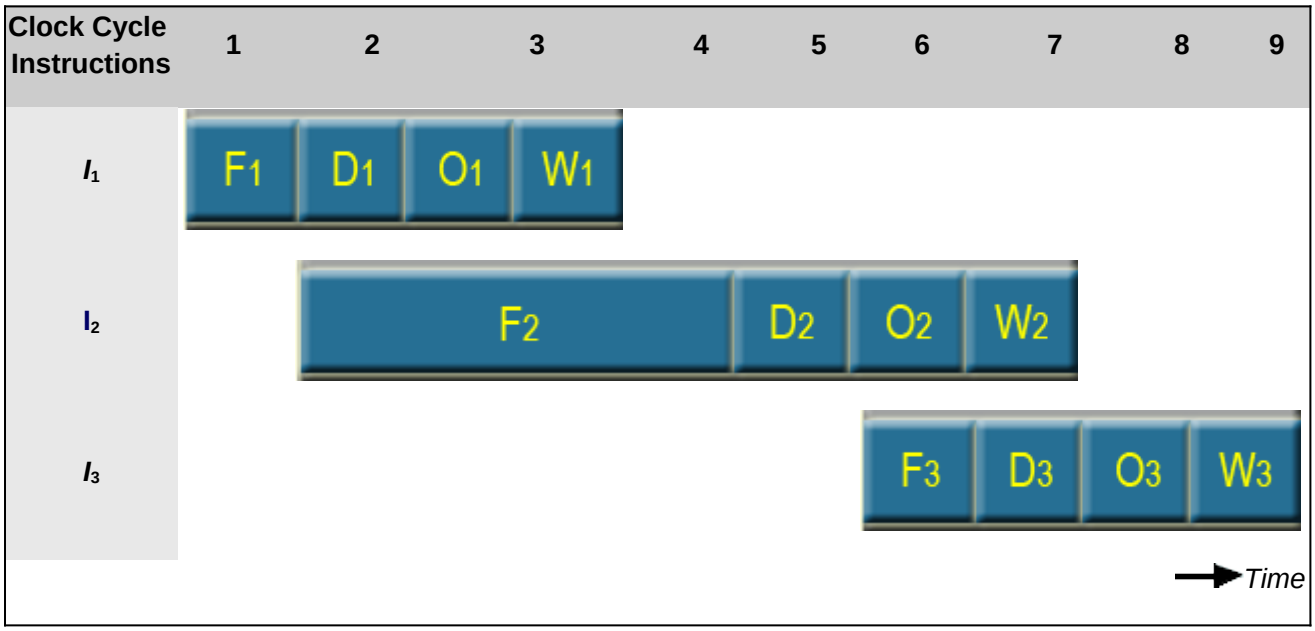
The contents of B1, B2 and B3 must always change at the same clock edge.

Due to that reason, pipeline operation is said to have been stalled for two clock cycle. Normal pipeline operation resumes in clock cycle 7.

Whenever the pipeline stalled, some degradation in performance occurs.

Role of cache memory:

The use of cache memory solves the memory access problem. issued the memory request to take much longer time to complete its task and in this case the pipeline stalls. The effect of cache miss in pipeline processing is shown in the Figure 7.



Occasionally, a memory request results in a cache miss. This causes the pipeline stage that

Figure 7: Effect of cache miss in pipeline processing

Clock Cycle Stages	1	2	3	4	5	6	7	8	9	10
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_4	F_3			
D: Decode			D_1	*idle*	*idle*	*idle*	D_2	D_3		
O: Operate				O_1	*idle*	*idle*	*idle*	O_2	O_3	

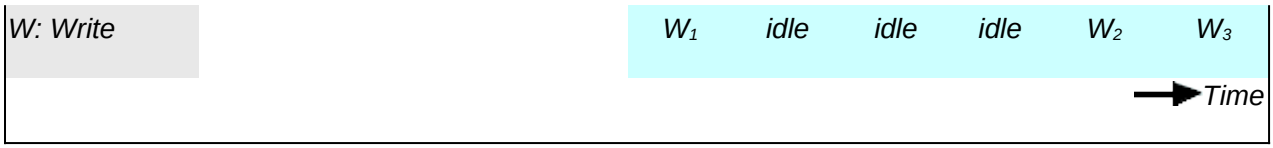


Figure 9.8: Function performed by each stage as a function of time.

Function performed by each stage as a function of time ias shown in Figure 9.8.

In this example, instruction I_1 is fetched from the cache in cycle 1 and its execution proceeds normally.

The fetch operation for instruction I_2 which starts in cycle 2, results in a cache miss.

The instruction fetch unit must now suspend any further fetch requests and wait for I_2 to arrive.

We assume that instruction I_2 is received and loaded into buffer *B1* at the end of cycle 5, It appears that cache memory used here is four time faster than the main memory.

The pipeline resumes its normal operation at that point and it will remain in normal operation mode for some times, because a cache miss generally transfer a block from main memory to cache.

From the figure, it is clear that Decode unit, Operate unit and Write unit remain idle for three clock cycle.

Such idle periods are sometimes referred to as bubbles in the pipeline.

Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit. A pipeline can not stall as long as the instructions and data being accessed reside in the cache. This is facilitated by providing separate on chip instruction and data caches.

Dependency Constraints:

Consider the following program that contains two instructions, I_1 followed by I_2

$$I_1 : A \leftarrow A + 5$$

$$I_2 : B \leftarrow 3 * A$$

When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 completes. The pipeline execution is shown in Figure 9.9.

Clock Cycle Stages	1	2	3	4	5	6

F: Fetch	F_1	D_1	O_1	W_1		
D: Decode		F_2	D_2	O_2	W_2	

Figure 9.9: Pipeline execution of two instructions

In clock cycle 3, the specific operation of instruction I_1 i.e. addition takes place and at that time only the new updated value of *A* is available. But in the clock cycle 3, the instruction I_2 is fetching the operand that is required for the operation of I_2. Since in clock cycle 3 only, operation of instruction I_1 is taking place, so the instruction I_2 will get the old value of *A* , it will not get the updated value of *A* , and will produce a wrong result. Consider that the initial value of *A* is 4. The proper execution will produce the result as

B=27

$$I_1 : A \leftarrow A + 5 = 4 + 5 = 9$$
$$I_2 : B \leftarrow 3 \times A = 3 \times 9 = 27$$

But due to the pipeline action, we will get the result as

$$I_1 : A \leftarrow A + 5 = 4 + 5 = 9$$
$$I_2 : B \leftarrow 3 \times A = 3 \times 4 = 12$$

Due to the data dependency, these two instructions can not be performed in parallel.

Therefore, no two operations that depend on each other can be performed in parallel. For correct execution, it is required to satisfy the following:

- The operation of the fetch stage must not depend on the operation performed during the same clock cycle by the execution stage.
- The operation of fetching an instruction must be independent of the execution results of the previous instruction.
- The dependency of data arises when the destination of one instruction is used as a source in a subsequent instruction.