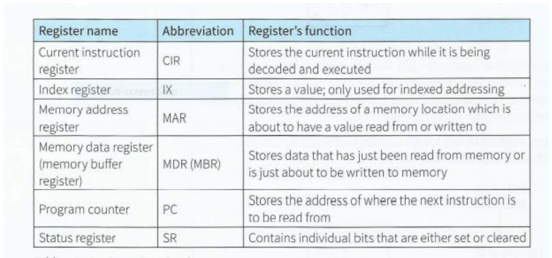
Registers:

Registers The other components of the CPU are the registers. These are storage components which, because of their proximity to the **ALU**, allow very short access times. Each register has limited storage capacity, typically 16, 32 or 64 bits. A register is either general purpose or special purpose. If there is only one general-purpose register it is referred to as the accumulator.

Accumulator: a general-purpose register that stores a value before and after the execution of an instruction by the ALU.



Register name	Abbreviation	Register's function
Current instruction register	CIR	Stores the current instruction while it is being decoded and executed
Index register	IX	Stores a value; only used for indexed addressing
Memory address register	MAR	Stores the address of a memory location which is about to have a value read from or written to
Memory data register (memory buffer register)	MDR (MBR)	Stores data that has just been read from memory or is just about to be written to memory
Program counter	PC	Stores the address of where the next instruction is to be read from
Status register	SR	Contains individual bits that are either set or cleared

Table 5.01 Registers in a simple CPU

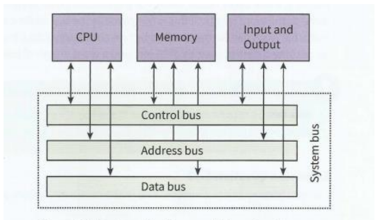


Figure 5.02 A schematic diagram of the system bus

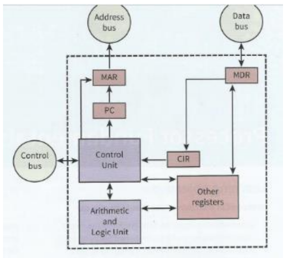


Figure 5.01 A schematic diagram of the architecture of a simple CPU

System bus:

A bus is a parallel transmission component with each separate wire carrying a single bit.   The schematic diagram of the CPU in Figure 5.01 l . shows the logical connection between each bus and a CPU component The address bus is connected to the **MAR**; the data bus to the **MDR**; and the control bus to the control unit The system bus allows data flow between the CPU, the memory, and input or output (1/0) devices as shown in the schematic diagram in Figure 5.02.

-The address bus: The sole function of the address bus is to carry an address. This address is loaded on to the bus from the **MAR** as and when directed by the control unit The address specifies a location in memory which is due to receive data or from which data is to be read. The address bus is a 'one-way street'. It can only be used to send an address to a memory controller. It cannot be used to carry an address from the memory controller back to the CPU.

-Address bus: a component that carries an address to the memory controller to identify a location in memory which is to be read from or written to. The crucial aspect of the address bus is the 'bus width', which is the number of separate wires in the bus. The number of wires defines the number of bits in the address's binary code. In the simple computer system considered here we will assume that the bus width is 16 bits allowing 65536 memory locations to be directly addressed.

-The data bus: The function of the data bus is to carry data. This might be an instruction, an address or a value. As can be seen from Figure 5.02, the data bus might be carrying the data from **CPU** to memory or from memory to CPU. However, another option is to carry data to or from an 1/0 device. The diagram does not make clear whether, for instance, data coming from an input device is carried first to the CPU or directly to the memory.

Data bus: a component that carries data to and from the processor.

Word: a small number of bytes handled as a unit by the computer system

-The control bus: The control bus is another bidirectional bus which transmits a signal from the control unit to any other system component or transmits a signal to the control unit. There is no need for extended width so the control bus typically has just eight wires. A major use of the control bus is to carry timing signals. As described in Section 5.02, **the system clock** in the control unit defines the clock cycle for the computer system. The control bus carries timing signals at time intervals dictated by the clock cycle. This ensures that the time that one component transmits data is synchronized with the time that another component reads it. **The clock speed** is the most important factor governing the processing speed of the system. The schematic diagram in Figure 5.02 slightly misrepresents the situation because it looks as if the **CPU**, the memory and the 1/0 devices have similar access to the data and control buses. The reality is different. Each 1/0 device is connected to an interface called a port. Each port is connected to the 1/0 or device controller. This controller handles the interaction between the **CPU** and an 1/0 device. A port is described as 'internal' if the connected 1/0 device is an integral part of the computer system. An external port allows the computer user to connect a peripheral 1/0 device.

The universal serial bus {USB): In the early days of the PC, the process of connecting a peripheral was time-consuming and required technical expertise. The aim of the plug-and -play concept was to remove the need for technical knowledge so that any computer user could connect a peripheral and start using it straight away. The plug-and-play concept was only fully realized by the creation of the USB.

Some information about the USB standard:

• A hierarchy of connections is supported.

• The computer is at the root of this hierarchy and can handle 127 attached devices. Devices can be attached while the computer is switched on and are automatically configured for use.

• The standard has evolved, with USB 3.0 being the latest version.

Some information about the USB standard:

• A hierarchy of connections is supported.

• The computer is at the root of this hierarchy and can handle 127 attached devices. Devices can be attached while the computer is switched on and are automatically configured for use.

• The standard has evolved, with USB 3.0 being the latest version.

The fetch-execute cycle

5.04 The fetch–execute cycle

The full name for this is the fetch, decode and execute cycle. This is illustrated by the flowchart in Figure 5.03.

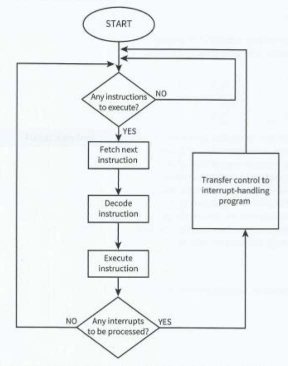

START
Any instructions to execute?
NO
YES
Fetch next instruction
Decode instruction
Execute instruction
Transfer control to interrupt-handling program
NO
Any interrupts to be processed?
YES

Figure 5.03 Flowchart for the fetch, decode and execute cycle

If we assume that a program is already running then the program counter already holds the address of an instruction. In the fetch stage, the following steps happen:

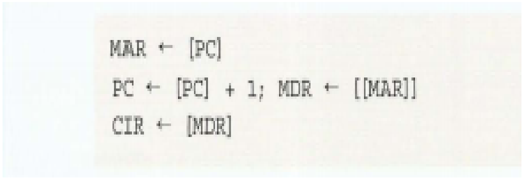
1 -This address in the program counter is transferred within the **CPU** to the **MAR**.

2 -During the next clock cycle two things happen simultaneously: the instruction held in the address pointed to by the **MAR** is fetched into the **MDR**.

• The address stored in the program counter is incremented.

3- The instruction stored in the **MDR** is transferred within the CPU to the **CIR**. For our simple system the program counter will be incremented by 1. However, it should be noted that the instruction just loaded might be a jump instruction. In this case, the program counter contents will have to be updated in accordance with the j ump condition. This can only happen after the instruction has been decoded. In the decode stage, the instruction stored in the **CIR** is received as input by the circuitry within the control unit. Depending on the type of instruction, the **control unit** will send signals to the appropriate components so that the execute stage ca n begin. At this stage, **the ALU** will be activated if the instruction requires arithmetic or logic processing.

Register transfer notation



$$\begin{aligned} &MAR \leftarrow [PC] \\ &PC \leftarrow [PC] + 1;\ MDR \leftarrow [[MAR]] \\ &CIR \leftarrow [MDR] \end{aligned}$$

Operations involving registers can be described by register transfer notation. The simplest form of this can be illustrated by the following representation of the fetch stage of the fetch - execute cycle: MAR +- [PC] PC +- [PC] + 1; MDR +- [[MAR)] CIR +- [MDR) The basic format for an individual data transfer is similar to that for variable assignment. The first item is the destination of the data. Here the appropriate abbreviation is used to identify the particular register. To the right of the arrow showing the transmission of data is the definition of this data. In this definition, the square brackets around a register abbreviation show that the content of the register is being moved possibly with some arithmetic operation being applied. When two data operations are placed on the same line separated by a semi-colon this means that the two transfers take place simultaneously. The double pair of brackets around **MAR** on the second line needs careful interpretation. The content of the **MAR** is an address; it is the content of that address which is being transferred to the **MDR.**

Interrupt handling: There are many different reasons for an interrupt to be generated. Some examples are:

• a fatal error in a program

• a hardware fault • a need for 1/0 processing to begin

• user interaction

• a timer signal. There are a number of different approaches possible for the detailed mechanisms used to hand le interrupts but the overriding principles are clearly defined. Each different interrupt. Chapter 5: Processor Fundamentals needs to be handled appropriately and different interrupts might possibly have different priorities. Therefore, the processor must have a means of identifying the type of interrupt.
One way is to have **an interrupt register** in the CPU that works like the **status register**, with each individual bit operating as a flag for a specific

type of **interrupt**. As the flowchart in Figure 5.03 shows, the existence of an interrupt is only detected at the end of a fetch-execute cycle. This allows the current program to be interrupted and left in a defined state which can be returned to later. The first step in hand ling the interrupt is to store the contents of the program counter and any other registers somewhere safe in memory. Following this, the appropriate interrupt handler or interrupt service routine (ISR) program is initiated by loading its start address into the program counter. When the **ISR** program has been executed there needs to be an immediate check to see if further interrupts need handling. If there are none, the safely stored contents of the registers are restored to the CPU and the originally running program is resumed.