Chapter 4 Processor Fundamentals

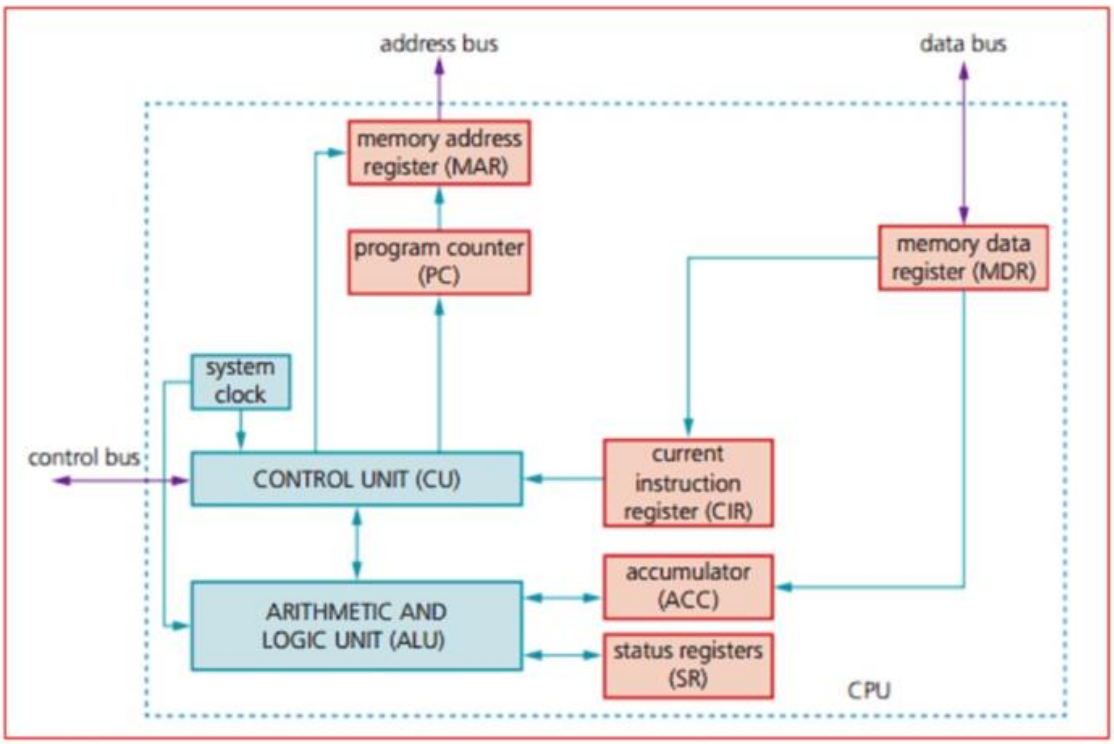
4.1 Central Processing Unit (CPU) Architecture

4 Processor Fundamentals	
4.1 Central Processing Unit (CPU) Architecture	
Candidates should be able to:	Notes and guidance
Show understanding of the basic Von Neumann model for a computer system and the stored program concept	
Show understanding of the purpose and role of registers, including the difference between general purpose and special purpose registers	Special purpose registers including: • Program Counter (PC) • Memory Data Register (MDR) • Memory Address Register (MAR) • The Accumulator (ACC) • Index Register (IX) • Current Instruction Register (CIR) • Status Register
Show understanding of the purpose and roles of the Arithmetic and Logic Unit (ALU), Control Unit (CU) and system clock, Immediate Access Store (IAS)	
Show understanding of how data are transferred between various components of the computer system using the address bus, data bus and control bus	
Show understanding of how factors contribute to the performance of the computer system	Including: • processor type and number of cores • the bus width • clock speed • cache memory
Understand how different ports provide connection to peripheral devices	Including connection to: • Universal Serial Bus (USB) • High Definition Multimedia Interface (HDMI) • Video Graphics Array (VGA)
Describe the stages of the Fetch-Execute (F-E) cycle	Describe and use 'register transfer' notation to describe the F–E cycle
Show understanding of the purpose of interrupts	Including: • possible causes of interrupts • applications of interrupts • use of an Interrupt service (ISR) handling routine • when interrupts are detected during the fetch-execute cycle • how interrupts are handled

Von Neumann Architecture

Stored Program Concept

- Programs and instructions are stored in the memory
- Programs consists of sequence of instructions • Each are fetched and then executed in a sequence.
- Instruction fetch and execute cannot occur at the same time



ALU

- Arithmetic Logic Unit or ALU is used to perform arithmetic and logical operations. The temporary result of which is stored in accumulator.

Control Unit

- Control unit sends signals through the control bus to various components to synchronize the fetch and execute cycle.

System Clock

- Generates timing signals to synchronize fetch and execute cycle.

Immediate access store (IAS)

The IAS holds all the data and programs that the processor (CPU) needs to access. The CPU takes data and programs held in the backing store and puts them into the IAS temporarily. This is done because read/write operations carried out using the IAS are considerably faster than read/write operations to backing store. Consequently, any key data needed by an application will be stored temporarily in IAS to speed up operations. The IAS is another name for primary (RAM) memory.

Registers

- Status Register
    - Status register is interpreted as flags for different reasons, such as overflow in addition calculation, or result of a calculation is zero.
- PC
    - Program Counter is used to store the address of the next current instruction to be fetched.
    - It is increased by one as soon as the address is fetched into MAR.

Register	Abbreviation	Function/purpose of register
current instruction register	CIR	stores the current instruction being decoded and executed
index register	IX	used when carrying out index addressing operations (assembly code)
memory address register	MAR	stores the address of the memory location currently being read from or written to
memory data/ buffer register	MDR/MBR	stores data which has just been read from memory or data which is about to be written to memory (sometimes referred to as MBR)
program counter	PC	stores the address where the next instruction to be read can be found
status register	SR	contain bits which can be set or cleared depending on the operation (for example, to indicate overflow in a calculation)

How special registers are used in the fetch and execute cycle

- Program Counter is used to store the address of the next current instruction to be fetched, once an instruction is fetched the program counter is incremented by one.
- The address stored in the PC is the transferred to MAR
- The content at the location in MAR is stored in MDR
- The instruction stored in MDR is then transferred to CIR, instruction is then decoded and Control unit sends signals through control bus to various components to synchronize the fetch and execute cycle.

Buses used in Von Neumann model

- Data Bus
  - Bidirectional
  - Transfer data to and from processor and i/o devices
- Address bus
  - Unidirectional
  - Used to transfer address of instruction of memory or input output device
- Control bus
  - Bidirectional
  - Used to transmit control signals to synchronize fetch and execute cycles ○

Factors affecting performance of CPU/Computer System

Number of cores

- Each core processes one instruction at a time
- Multiple cores mean the sequence of instructions can be split between them
- Multiple instructions executed at one clock pulse
- More cores decrease the time taken to complete task, fast performance of computer system

Width of Data Bus

- Width of the data bus determines the number of bits that can be transferred simultaneously
- The greater the width of the data bus, the more bits transferred at the same time, the better performance of computer system.
- Improves performance as less data transfers required, more data is transferred in one clock pulse

Clock Speed

- Clock speed determines the number of instructions executed per second.
- Each processor instruction takes specific number of clock cycles to execute
- The greater the clock speed is, the more clock cycles per unit time, instructions are fetched and executed quicker, less execution time.
- however there's a limit on clock speed because heat generated by high clock speed cannot be removed fast and can affect the performance of CPU.

Cache Memory

- When a processor reads memory, it first checks out cache and then moves
- on to main memory if the required data is not there.
- Cache memory stores frequently used instructions and data that need to be accessed faster.
- This improves processor performance.

USB / HDMI / VGA

USB ports

The Universal Serial Bus (USB) is an asynchronous serial data transmission method. It has quickly become the standard method for transferring data between a computer and a number of devices. The USB cable consists of a four-wired shielded cable, with two wires for power and the earth, and two wires used for data transmission. When a device is plugged into a computer using one of the USB ports

- the computer automatically detects that a device is present (this is due to a small change in the voltage level on the data signal wires in the cable)
- The device is automatically recognized, and the appropriate device driver is loaded up so that computer and device can communicate effectively
- if a new device is detected, the computer will look for the device driver which matches the device. If this is not available, the user is prompted to download the appropriate software

Benefits	Drawbacks
✓ Supported by many operating systems ✓ Plug and play feature, appropriate drivers are installed automatically ✓ Universal/Industrial standard so variety of support is available ✓ Backwards compatible ✓ Almost impossible to connect it incorrectly ✓ Allows power to be drawn to charge devices	✓ the present transmission rate is limited to less than 500 megabits per second ✓ the maximum cable length is presently about five meters ✓ the older USB standard (such as 1.1) may not be supported in the near future

HDMI

High-definition multimedia interface (HDMI) ports allow output (both audio and visual) from a computer to an HDMI-enabled device. They support highdefinition signals (enhanced or standard). HDMI was introduced as a digital replacement for the older Video Graphics Array (VGA) analogue system. Modern HD (high definition) televisions have the following features, which are making VGA a redundant technology:

- They use a widescreen format (16:9 aspect ratio).
- The screens use a greater number of pixels (typically 1920 × 1080).
- The screens have a faster refresh rate (such as 120Hz or 120 frames a second).
- The range of colors is extremely large (some companies claim up to four million different color variations).

This means that modern HD televisions require more data, which has to be received at a much faster rate than with older televisions (around 10 gigabits per second). HDMI increases the bandwidth, making it possible to supply the necessary data for high quality sound and visual effects. HDMI can also afford some protection against piracy since it uses high-bandwidth digital copy protection (HDCP). HDCP uses a type of authentication protocol. For example, a Blu-ray player will check the authentication key of the device it is sending data to (such as an HD television). If the key can be authenticated, then handshaking takes place and the Blu-ray can start to transmit data to the connected device.

Benefits	Drawbacks
✓ the current standard for modern televisions and monitors ✓ allows for a very fast data transfer rate ✓ improved security (helps prevent piracy) ✓ supports modern digital systems	✓ not a very robust connection (easy to break connection when simply moving device) ✓ limited cable length to retain good signal ✓ there are currently five cable/ connection standards

VGA

VGA was introduced at the end of the 1980s. VGA supports 640 × 480-pixel resolution on a television or monitor screen. It can also handle a refresh rate of up to 60Hz (60 frames a second) provided there are only 16 different colors being used. If the pixel density is reduced to 200 × 320, then it can support up to 256 colors. The technology is analogue and, as mentioned in the previous section, is being phased out.

Benefits	Drawbacks
✓ simpler technology ✓ only one standard available ✓ it is easy to split the signal and connect a number of devices from one source ✓ the connection is very secure	✓ old outdated analogue technology ✓ it is easy to bend the pins when making connections ✓ the cables must be of a very high grade to ensure good, undistorted signal

Interrupt

- An interrupt is a signal from a device or program to demand processor's attention,

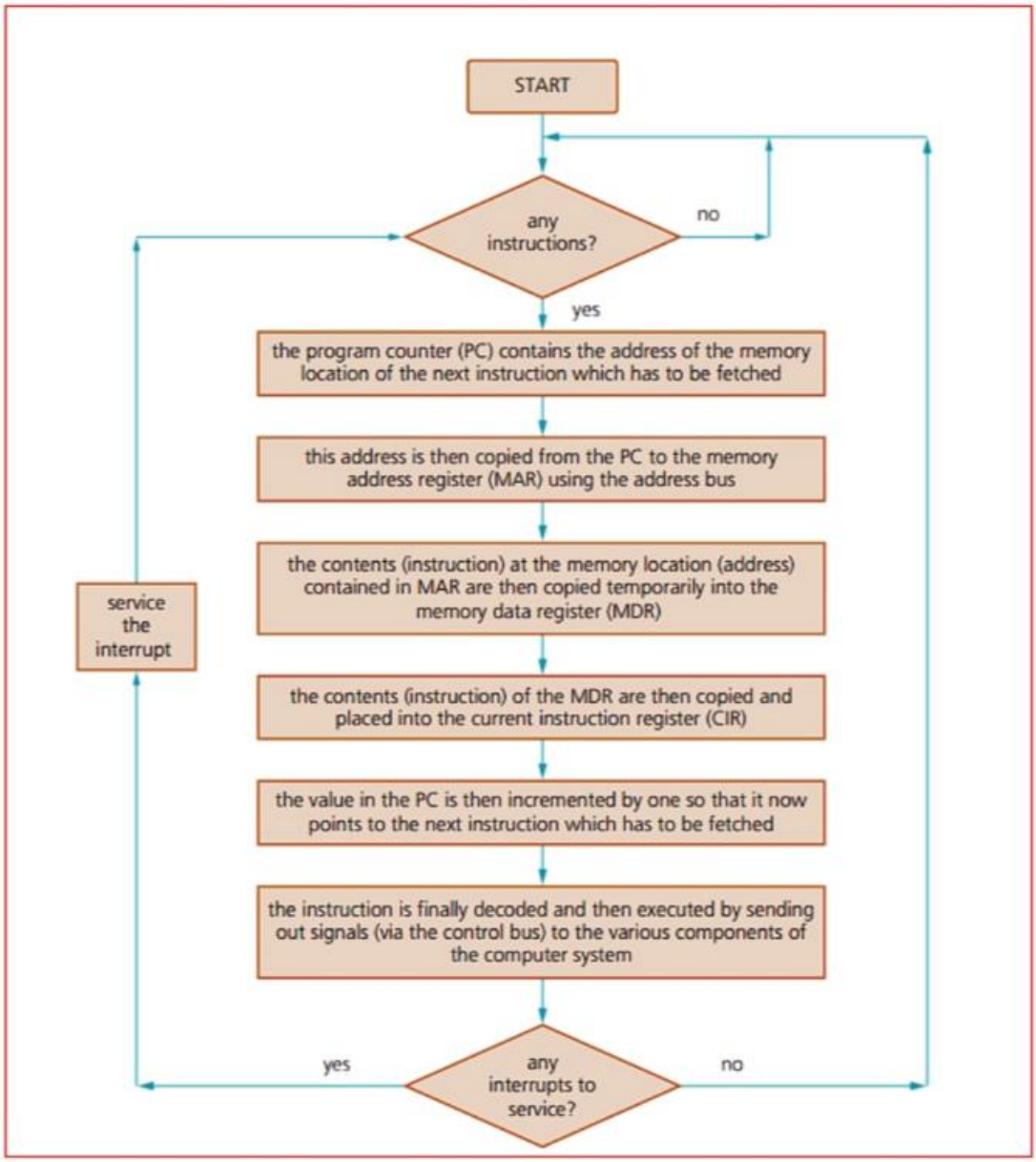
Possible causes of interrupts

- I/O Request // Printer asking for more data
- Hardware Fault // Out of toner, no paper, paper jam
- Software Fault
- Multitasking
- Pressing a key on the keyboard

How interrupt is handled during fetch and execute cycle

- At the end of each fetch and execute cycle interrupt is checked
- if the interrupt is found; interrupt flag is set
- Interruption is prioritized.
- If the priority is high enough
- the content of the registers are saved
- Processor calls interrupt service routine
- the program counter is then loaded with the address of the interrupt service routine.
- When the interrupt is serviced, contents of the register are restored
- the processor continues with the next F-E cycle.

F.E Cycle




```
MAR ← [PC]      contents of PC copied into MAR
PC  ← [PC] + 1  PC is incremented by 1
MDR ← [[MAR]]   data stored at address shown in MAR is
                copied into MDR
CIR ← [MDR]     contents of MDR copied into CIR
```

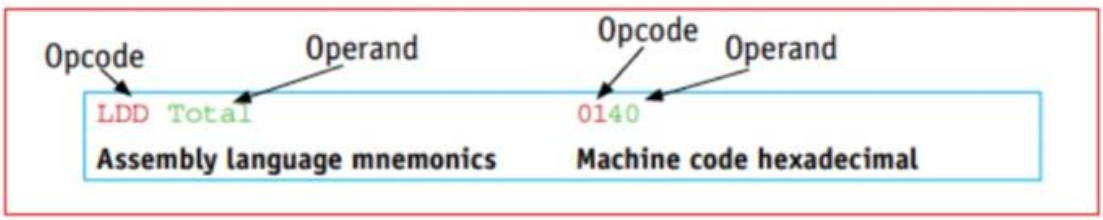
4.2 Assembly Language

4.2 Assembly Language	
Candidates should be able to:	Notes and guidance
Show understanding of the relationship between assembly language and machine code	
Describe the different stages of the assembly process for a two-pass assembler	Apply the two-pass assembler process to a given simple assembly language program
Trace a given simple assembly language program	
Show understanding that a set of instructions are grouped	Including the following groups: • Data movement • Input and output of data • Arithmetic operations • Unconditional and conditional instructions • Compare instructions
Show understanding of the different modes of addressing	Including Immediate, direct, indirect, indexed, relative

The only programming language that a CPU can use is machine code. Every different type of computer/chip has its own set of machine code instructions. A computer program stored in main memory is a series of machine code instructions that the CPU can automatically carry out during the fetch-execute cycle. Each machine code instruction performs one simple task, for example, storing a value in a memory location at a specified address. Machine code is binary, it is sometimes displayed on a screen as hexadecimal so that human programmers can understand machine code instructions more easily

The first programming language to be developed was assembly language, this is closely related to machine code and uses mnemonics instead of binary.

The structure of assembly language and machine code instructions is the same. Each instruction has an opcode that identifies the operation to be carried out by the CPU. Most instructions also have an operand that identifies the data to be used by the opcode



Stages of assembly

Before a program written in assembly language (source code) can be executed, it needs to be translated into machine code. The translation is performed by a program called assembler. An assembler translates each assembly language instruction into a machine code instruction. An assembler also checks the syntax of the assembly language program to ensure that only opcodes from the appropriate machine code instruction set are used. This speeds up the development time, as some errors are identified during translation before the program is executed.

There are two types of assembler:

single pass assemblers & two pass assemblers.

A single pass assembler puts the machine code instructions straight into the computer memory to be executed.

A two-pass assembler produces an object program in machine code that can be stored, loaded then executed at a later stage. This requires the use of another program called a loader. Two pass assemblers need to scan the source program twice, so they can replace labels in the assembly program with memory addresses in the machine code program

Pass 1

- ✓ Read the assembly language program one line at a time.
- ✓ Ignore anything not required, such as comments.
- ✓ Allocate a memory address for the line of code.
- ✓ Check the opcode in the instruction set.
- ✓ Add any new labels to the symbol table with the address, if known.
- ✓ Place the address of the labelled instruction on the symbol table.

Pass 2

- ✓ Read the assembly language program one line at a time.
- ✓ Generate object code, including opcode and operand, from the symbol table generated in Pass 1.
- ✓ Save or execute the program.

Practice past paper w17 p11, 12, 13

The second pass is required as some labels may be referred to before their address is known. For example, Found is a forward reference for the JPN instruction

```
Label       Opcode      Operand
Notfound:   LDD         200
            CMP         #0
            JPN         Found
            JPE         Notfound
Found:      OUT
```

If the program is to be loaded at memory address 100, and each memory location contains 16 bits, the symbol table for this small section of program would look like this:

Label	Address
Notfound	100
Found	104

Different types of assembly language instructions.

Data movement instructions

These instructions allow data stored at one location to be copied into the accumulator
LDM LDD LDI LDX LDR MOV

Input and output of data

These instructions allow data to be read from the keyboard or output to the screen
IN OUT

Arithmetic operations

These instructions perform simple calculations on data stored in the accumulator and store the answer the accumulator, overwriting the original data.
ADD, INC, DEC

Unconditional and conditional instructions

These instructions jump to another address if condition is true/false
JMP, JNE

Compare instructions

Compare the operand with the accumulator value
CMP

Different modes of addressing

Absolute addressing – the contents of the memory location in the operand are used. For example, if the memory location with address 200 contained the value 20, the assembly language instruction LDD 200 would store 20 in the accumulator.

Direct addressing – the contents of the memory location in the operand are used. For example, if the memory location with address 200 contained the value 20, the assembly language instruction LDD 200 would store 20 in the accumulator. Absolute and direct addressing are the same.

Indirect addressing – the contents of the contents of the memory location in the operand are used. For example, if the memory location with address 200 contained the value 20 and the memory location with address 20 contained the value 5, the assembly language instruction LDI 200 would store 5 in the accumulator.

Indexed addressing – the contents of the memory location found by adding the contents of the index register (IR) to the address of the memory location in the operand are used. For example, if IR contained the value 4 and memory location with address 204 contained the value 17, the assembly language instruction LDX 200 would store 17 in the accumulator.

Immediate addressing – the value of the operand only is used. For example, the assembly language instruction LDM #200 would store 200 in the accumulator.

Relative addressing – the memory address used is the current memory address added to the operand. For example, JMR #5 would transfer control to the instruction 5 locations after the current instruction.

Symbolic addressing – only used in assembly language programming. A label is used instead of a value. For example, if the memory location with address labelled MyStore contained the value 20, the assembly language instruction LDD MyStore would store 20 in the accumulator.

4.3 Bit Manipulation

4.3 Bit manipulation	
Candidates should be able to:	Notes and guidance
Show understanding of and perform binary shifts	logical, arithmetic and cyclic Left shift, right shift
Show understanding of how bit manipulation can be used to monitor / control a device	Carry out bit manipulation operations Test and set a bit (using bit masking)

Label	Instruction		Explanation
	Opcode	Operand	
	AND	#n/Bn/&n	Bitwise AND operation of the contents of ACC with the operand
	AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
	XOR	#n/Bn/&n	Bitwise XOR operation of the contents of ACC with the operand
	XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
	OR	#n/Bn/&n	Bitwise OR operation of the contents of ACC with the operand
	OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address> <address> can be an absolute address or a symbolic address
	LSL	#n	Bits in ACC are shifted logically n places to the left. Zeros are introduced on the right hand end
	LSR	#n	Bits in ACC are shifted logically n places to the right. Zeros are introduced on the left hand end
<label>:	<opcode>	<operand>	Labels an instruction
<label>:		<data>	Gives a symbolic address <label> to the memory location with contents <data>

AND: Checking, Testing, Resetting all to zero

OR: Setting

XOR: Inversing

Shifts

Logical shift left involves shifting bits to the left by a certain number of positions, while **logical shift right** involves shifting bits to the right by a certain number of positions. These shifts fill the empty positions with zeroes. In contrast, **arithmetic shift** involves shifting bits to the left or right while preserving the sign bit, which is the leftmost bit representing the sign (+/-) of the number.

To perform a **logical shift left** or **right** on an 8-bit binary number, we simply shift the bits in the desired direction and fill the empty positions with zeroes. For example, if we want to logical shift an 8-bit binary number 3 positions to the left, we will shift all the bits 3 places to the left and fill the empty positions with zeroes.

To perform an **arithmetic shift left** or **right** on an 8-bit binary number, we similarly shift the bits in the desired direction. However, for an arithmetic shift right, we fill the empty positions with the same bit as the sign bit, while for an arithmetic shift left, we fill the empty positions with zeroes. For example, if we want to arithmetic shift an 8-bit binary number 3 positions to the right, we will shift all the bits 3 places to the right and fill the empty positions with the sign bit.

Cyclic shift, also known as circular shift, involves shifting the bits of a binary number to the left or right, and then bringing the "extra" bits around to the other side of the number. This is different from a logical or arithmetic shift, where the extra bits are always filled with zeroes or the sign bit.

To perform a **cyclic shift** on an 8-bit binary number, you can first determine the number of positions to shift. Then, you can shift the bits in the desired direction and wrap the extra bits around to the other end of the number. For example, if you are performing a cyclic shift to the left by 3 positions on the binary number 10101110, you would get 01110101.