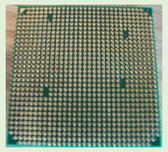
# Computer Architecture

## Internal Hardware Components

**Motherboard**
The **motherboard** is a circuit board that enables all the other components to be connected to it either directly via slots or cables. Normally the graphics and sound cards are integrated into motherboard allowing you to access then via hardware ports.

**Hard Disk**
The **hard disk** allows you to store and retrieve documents, music, pictures and any other files. Even when there is no power, the data remain unchanged and can be accessed once again once power has been restored.

**Processor**
The **processor** fetches, decodes and executes instructions and performs logical and arithmetic operations.



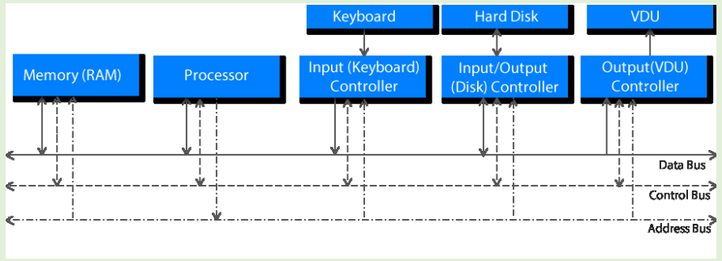
**Main Memory**
The purpose of **main memory** is to store the instructions from the programs currently running on a computer system and to temporarily hold the data needed by those programs.

**Random Access Memory RAM** memory is volatile. This means that when the computer is turned off the contents of volatile memory is lost. When there is no power, volatile memory is erased When you create a document that you do not save, if the computer crashes, the document will be lost because it is only stored in RAM which is volatile memory. If you have saved the document to the hard disk drive then you will be able to retrieve the document once power has returned. Every time the computer is powered up the operating system needs to be copied into the RAM from secondary storage. This is what happens during the boot sequence.

**Input / Output Controller**
The input and output controller manages hardware devices such as hard disk, monitor and keyboard. This is helpful for the processor because the processor does no need to control each device. In any case each device will have different control signals, so the processor can delegate the task of managing the input and output devices to the input / output controller.

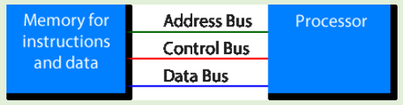
- Input controller – eg manages mouse, keyboard
- Output controller – eg manages monitor, printer
- Input / Output disk controller – eg manages the secondary storage devices.

**Buses**
- The **data bus** is a set of wires that allows the transfer of data between different computer components including the main memory, processor and input/output controllers. The data bus sends data in both directions for the processor, and memory, but input controllers send data only and output controllers receive data only.
- The **address bus** specifies the physical memory address that we wish to write data to or read from. The processor sends address requests and all the other components receive requests.
- The **control bus** is used to send commands from the processor to other components and to receive status signals from the components. The control bus sends signals in both directions for all components.



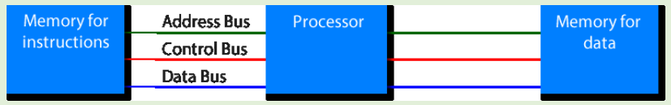

**The Stored Program Concept**
The instructions are stored in main memory. The instructions are fetched, decoded and executed in the processor.

**von Neumann Architecture**
- For systems using von Neumann architecture the program instructions and the data to be processed are stored in the same memory.
- Instructions and data have to be retrieved one at a time.
- General purpose computers are based on von Neumann architecture.




*von Neumann bottleneck*
The von Neumann bottleneck refers to the latency for transferring data and instructions between memory and the processor. Even if the speed of memory and the processor are improved, in the von Neumann architecture we are still limited by the speed of transfer of data between the two components. This is because both the data and instructions use the same buses.

**Harvard Architecture**
With Harvard architecture there are separate memory storage for the instructions and the data. This allows simultaneous retrieval both data and instructions on each processor cycle therefore speeding up the throughput of instructions and data and reducing latency. Harvard Architecture is typically used in embedded systems and used for signal processing. The program is stored onboard the processor in ROM.




**Addressable Memory**
- Each location in memory has an address associated with it. The data refers to the content at each location. We can access the data in memory by referencing to the memory location and writing to or reading the values in these memory locations.
- With word addressing the typical size of each memory location is the word length of the system (typically 32 or 64 bits)
- With byte addressing each memory location can hold one byte.
- The number of memory addresses is determined by the address bus width. For instance, if the bus address bus width is 32 bits then $2^{32}$ address locations can be accessed.

## Structure and role of the processor and its components

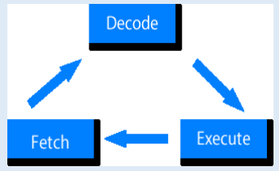
**Processor (CPU) Components**

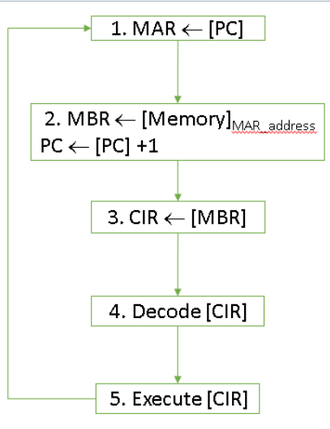
- **Arithmetic logic unit** – The arithmetic and logic operations take place here.
- **Control unit** – Decodes the instruction, splits it into the opcode and operand and identifies the specific arithmetic operation.
- **Clock** – The purpose is to keep all the processor components synchronised. The clock controls the processor so that it can process the instructions one-by-one and in the correct order.
- **Registers** – Memory locations on the CPU. Much faster than cache and RAM however also much smaller. Used to store frequently accessed data and instructions. There are two types of registers general purpose and registers dedicated to a specific purpose.

*Dedicated Registers*
- **Memory Address Register (MAR)** – The address in main memory that we wish to fetch the instruction from
- **Memory Buffer Register (MBR)** – Holds the instruction that has been fetched from memory
- **Current instruction register (CIR)** – Holds the contents of MBR, which is the instruction to be processed
- **Program Counter (PC)** – This holds the memory address of the location to fetch the next instruction from.
- **Status Register (SR)** – Gives information on such things as overflow/underflow, interrupts, parity. The status register allows an instruction to depend the outcome of the previous instruction.
- **Accumulator** – Holds the results of arithmetic and logic computations

**Fetch Decode Execute Cycle**

1. The instructions are loaded into memory
2. The processor fetches the instruction from the main memory
3. The instruction is decoded so the CPU knows what to do with the instruction
4. The processor then executes the instruction
5. The result of the instruction can be stored in memory
6. The next instruction is then fetched from main memory and the cycle repeats itself

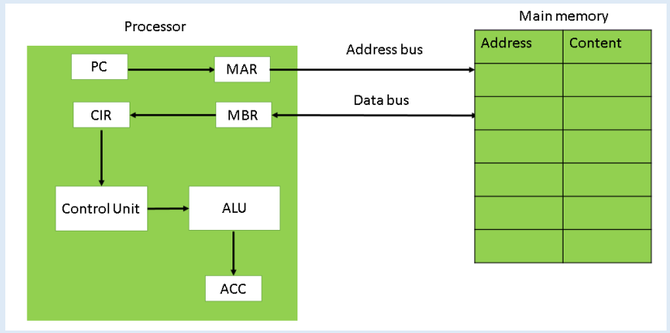



1. Transfer the contents of the PC to the MAR
2. The contents at that address is transferred back to the MBR along the data bus At the same time the PC is incremented by one to get ready for the next cycle.
3. Transfer the contents of the MAR to the CIR ready to be decoded.
4. Control unit decodes instruction held in CIR into operand and opcode

| | |
|---|---|
| | 5. Instruction executed by ALU for instance and result stored in Accumulator. Status register updated |

**Processor Schematic**




**Fetch**

Processor

| Register | Content |
|---|---|
| PC | 0 |
| MAR | |
| MBR | |
| CIR | |

Main memory

| Address | Content |
|---|---|
| 0 | 00101001 |
| 1 | 11101010 |
| 2 | 11100101 |
| 3 | 10101110 |

**1. Fetch: MAR ← [PC]**

Processor

| Register | Content |
|---|---|
| PC | 0 |
| MAR | 0 |
| MBR | |
| CIR | |

Address bus

Main memory

| Address | Content |
|---|---|
| 0 | 00101001 |
| 1 | 11101010 |
| 2 | 11100101 |
| 3 | 10101110 |

- Transfer the contents of the PC to the MAR
- The address in MAR is transferred to main memory along the address bus

**2. Fetch: MBR ← [Memory]$_{MAR\_address}$ ; PC ← [PC] +1**

Processor

| Register | Content |
|---|---|
| PC | 1 |
| MAR | 0 |
| MBR | 00101001 |
| CIR | |

Address bus

Data bus

Main memory

| Address | Content |
|---|---|
| 0 | 00101001 |
| 1 | 11101010 |
| 2 | 11100101 |
| 3 | 10101110 |

- The contents at that address is transferred back to the MBR along the data bus
- At the same time the PC is incremented by one to get ready for the next cycle. This can be done because the two steps do not rely on one another.

**3. Fetch: CIR ← [MBR]**

Processor

| Register | Content |
|---|---|
| PC | 1 |
| MAR | 0 |
| MBR | 00101001 |
| CIR | 00101001 |

Address bus

Data bus

Main memory

| Address | Content |
|---|---|
| 0 | 00101001 |
| 1 | 11101010 |
| 2 | 11100101 |
| 3 | 10101110 |

- Transfer the contents of the MAR to the CIR ready to be decoded.
- This allows the MBR to be used to store the next instruction during the decode and execution stages.

**4. Decode [CIR]**

- The control unit decodes the instruction held by the current instruction register.
- The instruction is divided into the opcode and the operand

**5. Execute [CIR]**

- The instruction is executed in the arithmetic logic unit.
- The result is stored in the accumulator register
- The status register is updated

**Processor Instruction Set**

The instruction set is a set of machine language commands for the processor. Each processor architecture will have its own unique instruction set. Thus, any code written in machine code or assembly language will be processor specific. Nevertheless, all processor architectures will have a common set of operations that include:

- Data transfer between internal components
- arithmetic operations
- Logical operations
- Shift operations
- Branching
- Comparison

**Instructions**

- Instructions are made up of two parts: the opcode and the operand.
- The opcode determines the operation that is to be carried out (eg. ADD, LOAD).
- The operand is the value or memory address that that instruction is to be operated on.
- The opcode in turn is split into two parts: The basic machine operation and the addressing mode.

**Addressing Mode**

The addressing mode determines how the operand is interpreted. There are many addressing modes but we are only interested in two: The immediate addressing mode and the direct addressing mode.

- **Immediate addressing mode** – The data value in the operand is part of the instruction. In other words this is the actual value we apply the operation to.
- **Direct addressing mode** – The operand specifies the address of the data in its memory location

**Opcode and Operand**

| Opcode | | | | Operand | | | |
|---|---|---|---|---|---|---|---|
| Basic Machine operation | | | Addressing mode | | | | |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

In the example we have allocated 8 bits to each instruction. Four bits are allocated to the operand thus 16 ($2^4$) possible addresses in memory are accessible or data values. Four bits are allocated to the opcode 1 of which is 1 given over to the addressing mode. Thus we have 2 ($2^1$) addressing modes, and 8 ($2^3$) machine operations.

*Examples*

| Opcode | Assembler instruction | Description |
|---|---|---|
| 010 0 0101 | LOAD #5 | 010 - Load the value 5 into accumulator register<br>0 - direct addrssing mode (# on the operand refers to direct addressing mode, its absence implies immediate addressing mode<br>101 – binary value 5 |
| 100 0 0100 | ADD #4 | 100 - ADD to value already in accumulator register<br>0 - Direct addressing<br>100 – binary value 4 |
| 101 1 1111 | STORE 15 | 101 - Copy the contents of the accumulator register<br>1 – Immediate addressing<br>1111 - binary value 15 (memory location) |

## Assembly Language

**Data transfer operations**

| Instruction | Description | Example | Example description |
|---|---|---|---|
| LDR R, M | Value in main memory address M loaded into register R | LDR R1, 100 | Load value at memory location 100 into register R1 |
| STR R, M | Value in register R stored in main memory address location M | STR R1, 100 | Store value in R1 into main memory location 100 |
| MOV R, #V | Copy data value #V register R | MOV R1, #12 | Copy the number 12 into register R1. |

The # refers to immediate addressing ie the value is the data.

**Arithmetic operations**

| Instruction | Description | Example | Example description |
|---|---|---|---|
| `ADD Ra, Rb, <operand>` | Add values in registers in Rb and Rc and load result in register Ra | `ADD R1, R1, #102`<br>`ADD R1, R1, R2` | Add 102 to the value in register `R1` and store the value in register `R1`<br>Add the value stored in R2 and add to the value stored in R1 and output the result. |
| `SUB Ra, Rb, <operand>` | Subtract value in registers in Rc from Rb and load result in register Ra | `SUB R2, R1, #102`<br>`SUB R2, R1, R3` | Subtract 102 from the value in register `R1` and store the result in register `R2`<br>subtract the value stored in R3 from the value in R1 and store the result in R2 |

The <operand> can be a register or a data value. The register is indicated by R and a data value is preceded by a #.

**Logical shift operations**

| Instruction | Description | Example | Example description |
|---|---|---|---|
| `LSL Ra, Rb, <operand>` | Logical shift left value in register `Ra` by `<operand>` value and store in register `Ra` | `LSL R1, R1, #2`<br>`LSL R1, R1, R2` | Logical shift left value in `R1` by 2 and store in register `R1`<br>Logical shift left value in `R1` by value in `R2` and store in register `R1` |

| | | | |
|---|---|---|---|
| `LSR Ra, Rb, <operand>` | Logical shift left value in register `Ra` by `<operand>` value and store in register `Ra` | `LSR R1, R1, #2`<br>`LSR R1, R1, R2` | Logical shift right value in `R1` by 2 and store in register R1<br>Logical shift right value in `R1` by value in `R2` and store in register `R1` |

Eg $3_{10} << 2_{10}$; $3_{10} = 011_2$; $01100_2 = 12_{10}$

**Logical operations**

| Instruction | Description | Example | Example description |
|---|---|---|---|
| AND `Ra, Rb, <operand>` | Bitwise AND operation between value in register `Rb` and `<operand>` and store result in `Ra` | `AND R1, R1, #8` | Bitwise AND operation between value in `R1` and $8_{10}$ ($00001000_2$) and store result in `R1` |
| `ORR Ra, Rb, <operand>` | Bitwise OR operation between value in register `Rb` and `<operand>` and store result in `Ra` | `ORR R1, R1, #8` | Bitwise OR operation between value in `R1` and $8_{10}$ ($00001000_2$) and store result in `R1` |
| `EOR Ra, Rb, <operand>` | Bitwise XOR operation between value in register `Rb` and `<operand>` and store result in `Ra` | `EOR R1, R1, #8` | Bitwise XOR operation between value in `R1` and $8_{10}$ ($00001000_2$) and store result in `R1` |
| `MVN R, <operand>` | Bitwise NOT operation on `<operand>` and store in `R` | `MVN R1, #8` | Bitwise NOT operation on $8_{10}$ ($00001000_2$) and store result in `R1` ($11110111_2$) |

**Control**

| Instruction | Description |
|---|---|
| `CMP R, <operand>` | Compare value in register R with `<operand>` value |
| `B <label>` | Branch to position `<label>` |
| `BEQ <label>` | Branch to position `<label>` if result of last comparison between `R` and `<operand>` was equal |
| `BNE <label>` | Branch to position `<label>` if result of last comparison was not equal between `R` and `<operand>` |
| `BGT <label>` | Branch to position `<label>` if `R` was greater than `<operand>` in the last comparison comparison |
| `BLT <label>` | Branch to position `<label>` if `R` was less than `<operand>` in the last comparison |
| `HALT` | Terminate execution of program |
| `CMP R, <operand>` | Compare value in register R with `<operand>` value |

*Example selection (if .. else ..)*

```
LDR R1, 100
LDR R2, 102
ADD R1, R1, R2
CMP R1, #10
BEQ IF
STR R1, 101
B ELSE
IF:
STR R1, 102
ELSE:
HALT
```

*Example selection (if …)*

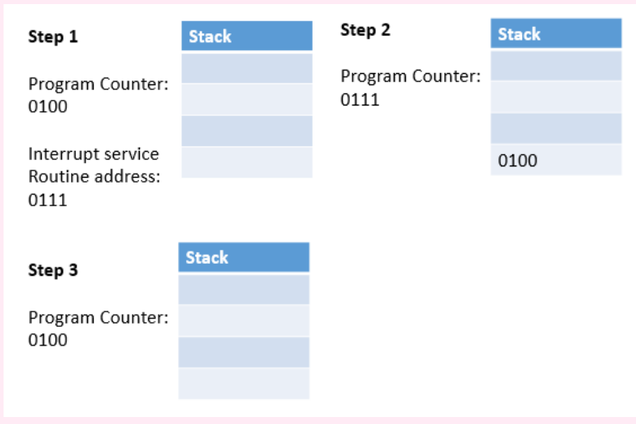
```
LDR R1, 100
LDR R2, 102
ADD R1, R1, R2
CMP R1, #10
BNE end
STR R1, 102
end:
HALT
```

*Example iteration*

```
MOV R0, #0
loop:
ADD R0 R0 #1
CMP R0 #4
BNE loop
STR R0 0
HALT
```

# Interrupts

- Interrupts are messages from software applications or hardware that request processing resources from the processor.
- If the processor is performing operations already then those processes can be interrupted and suspended and the CPU processes the interrupt request
- An interrupt service routine will be executed depending on the type of interrupts.
- When an interrupt occurs, the data and instruction in the program counter from the interrupted job are stored in the stack so that the job can be restored after the interrupt has been dealt with.
- Once the interrupt has been completed the processor resumes the job it was currently processing from where it left off. It can do this because the instructions and data are loaded back into program counter from the stack.




**Examples of interrupts**

- **Hardware commands** (computer reset)
- **Software** (error detection, Overflow error)
- **Timer**
- **Input/output** (Pressing a key on a keyboard, Moving the mouse)

# Factors affecting processor performance

**Clock speed**

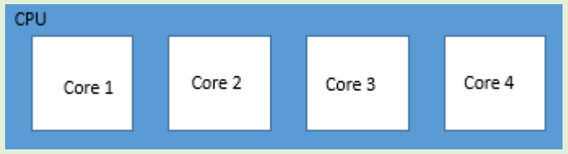
- This is the number of cycles that a processor carries out in an second.
- Nowadays a typical processor on a PC can have a clock speed ranging between 2GHz and 4GHz. That means there can be 2-4 billion cycles per second.
- Each cycle of the CPU allows a single instruction to be carried out.
- Thus the greater the clock speed, the greater the number of operations and the faster the computer will run.

*Moore's law*

- Moore's law states that the number of transistors on a processors doubles every 18 months.
- The number of transistors is related to the clock speed of a processor.
- There is evidence that this trend is slowing and it will reach a physical limit where there will be so many transistors that they produce too much heat.

**Number of processor cores**

A core is processor. Instead of making the processors with a higher clock speed (which will result in the processors getting too hot), manufacturers add more cores. Processors will have at least one core. Nowadays, most processors have dual (2) or even quad (4) cores to boost performance. Having multiple cores allows instructions to be carried out in parallel (at the same time), whereas a single core will only allow carry out instructions in serial (one at a time).




**Cache size**

Cache is a volatile memory store on the processor, and is not to be confused with RAM and registers. Cache is much faster but smaller that RAM. Frequently used data and instructions within an application can be stored in cache instead of fetching from RAM which is quite slow. In other words there is greater latency - Delay in transfer of data after the execution of an instruction. The bigger the cache the greater the volume of data and instructions that can be stored at a time and consequently increases the performance of the CPU as latency is reduced.

**Cache type**

Cache Level is a trade off between size and speed

- *Level 1 Cache* - This is closest to the CPU and is the fastest cache because of the lowest latency, but does not have much capacity
- *Level 2 Cache* – is slower and further away from the CPU than Level 1 cache because of increased latency, but has more storage capacity.
- *Level 3 Cache* – is the slower than L1 and L2 cache but still much faster than RAM, but has greater capacity than L1 and L2.

**Word length**

- The word length is the number of bits that can processed in one go.
- Typical word length of most processor architectures are 32 or 64 bits.
- The greater the word length the more bits that can be processed in one cycle.
- Address bus width

The address bus width determines the number of locations in main memory that can be accessed.

For instance if the bus address bus width is 32 bits then $2^{32}$ address locations can be accessed.

**Data bus width**

The data bus width refers to the number of bits that can be transferred at a time between the main memory and the processor. Increasing the data bus width can reduces the number of read write operations.

- If the data bus width is the same as the word length then one instruction can be transferred at a time.
- It the data bus width is twice the word length they two instruction can be transferred at a time.
- If the data bus width is half the word length the two read requests will need to be made for a single instruction.