**Chapter 3: Top-Level View of Computer**

Table of Contents

[3.1 Computer Components 3](#_Toc48981500)

[3.2 Computer Function 5](#_Toc48981501)

[Instruction Fetch and Execute 5](#_Toc48981502)

[Interrupts 6](#_Toc48981503)

[Multiple Interrupts 10](#_Toc48981504)

[I/O Function 12](#_Toc48981505)

[3.3 Interconnection Structures 13](#_Toc48981506)

[3.4 Bus Interconnection 15](#_Toc48981507)

[3.5 Point-To-Point Interconnect 18](#_Toc48981508)

[3.6 PCI Express 21](#_Toc48981509)

At a top-level, a computer consists of a CPU, memory and I/O components. These components are interconnected in some way to achieve the basic functions of the computer, which is to execute programs. At a top-level, we can characterize a computer system by describing the external behaviour of each component (the data and control signals it exchanges with other components) and the interconnection structure and controls required to manage the use of the interconnection structure.

## 3.1 Computer Components

Almost all computer designs are based on the concepts of von Neumann, referred to as von Neumann architecture. The three key concepts are:

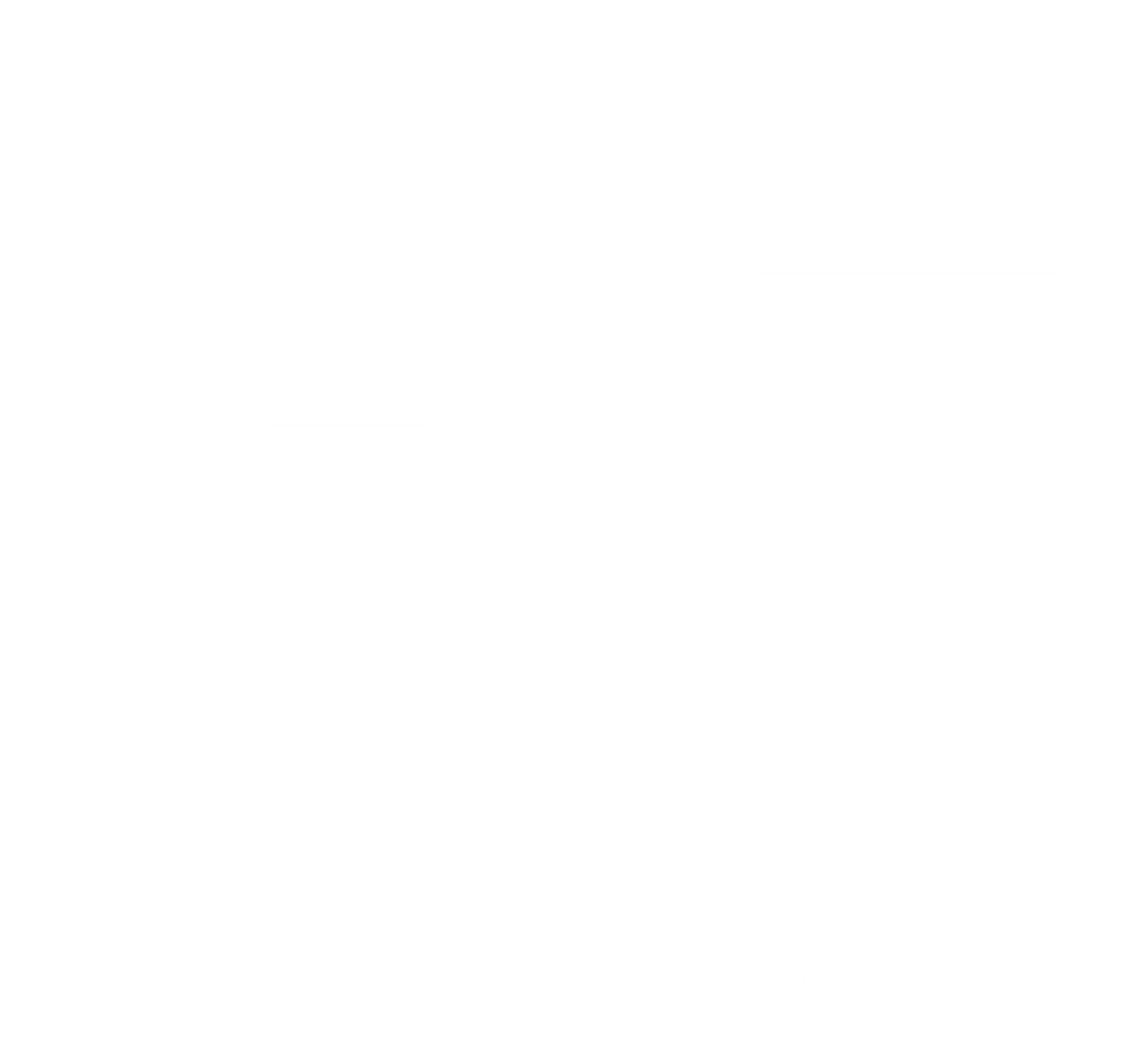
* Data and instructions are stored in a single read-write memory
* The contents of this memory are addressable by location, without regard to the type of data contained there
* Execution occurs sequentially unless explicitly modified

The reasons behind these concepts are:

* There is a small set of basic logic components that can be combined in different ways to store binary data and perform arithmetic and logical operations on that data.
* If there is a specific computation to be performed, a configuration of logic components can be designed into a *hardwired program* designed specifically for that computation.

Now consider an alternative approach. We create a general-purpose configuration of arithmetic and logic functions. This hardware can perform various functions on data depending on control signals applied. Unlike the previous case, where the configuration would accept some data and produce the results, this configuration would accept data and control signals and then produce results. Instead of having to rewire the hardwire for each new program, one would merely need to supply new control signals. This new method of programming, with a sequence of codes or instructions, is called software.

To be able to create this configuration, we essentially need two components, an instruction interpreter and a module of general-purpose arithmetic and logic functions. These constitute the CPU which does the main work. Other than this, we will also need input modules to get the data and instructions, along with components to convert the external data into some internal form of signals usable by the system, and we will need an output module to report the results. Together, these form the I/O components. We must also be able to store data and instructions, for which we need a module called memory.



The diagram above illustrates these components nicely. It also adds a few more details.

* MAR – Memory Address Register for address in memory on which to read or write
* MBR – Memory Buffer Register with data to be written or data received
* I/OAR – Input/Output Address Register to specify I/O device
* I/OBR – Input/Output Buffer Register to exchange data between I/O module and CPU
* I/O Module – Transfers data from external device to CPU and memory. Also contains internal buffer to temporarily hold this data.
* Memory Module – Contains locations with sequentially numbered addresses, each containing a binary number that can be interpreted as an instruction or data.

## 3.2 Computer Function

The basic function of a computer is to execute a program, which is just a set of instructions stored in memory. There are two key steps in program execution, *fetching* the instructions from memory one at a time, and *executing* them. A third step, decoding the instruction, can be added before the execution step. Thus, program execution is just a repeated process of instruction fetch and instruction execute.

The processing required for a single instruction is called an instruction cycle. This consists of the two steps given above, called the fetch cycle and the execute cycle. Program execution keeps repeating these cycles, until it is halted, either by a loss of power, some unrecoverable error, or an instruction to halt.

### Instruction Fetch and Execute

At the beginning of each instruction cycle, the processor fetches an instruction form memory. Typically, a register called the program counter (PC) holds the address of the next instruction to be fetched. Normally, the PC is incremented after each instruction fetch, so that it points to the next instruction. The fetched instruction is loaded into a register called the instruction register (IR). The instruction contains bits that tell the processor what action must be taken. This can be of four types:

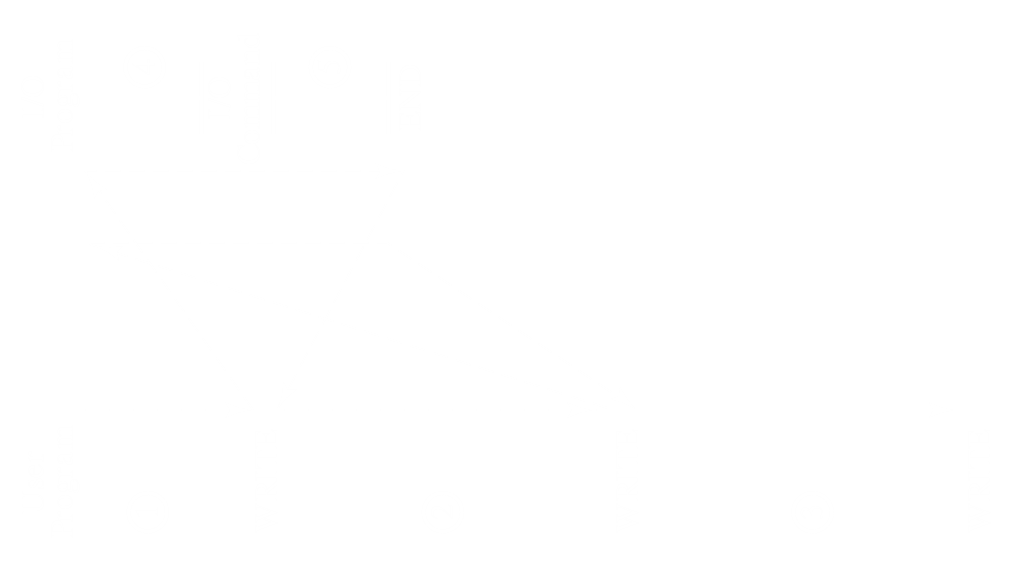
* Processor Memory – Data can be transferred from processor to memory or vice versa
* Processor I/O – Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module
* Data Processing – The processor may perform some arithmetic or logic operation on data
* Control – Alteration of the sequence of execution. This edits the location to which PC is pointing.

Other than this it is also important to know that the instruction currently being executed is stored in a data register called the accumulator (AC). The results of any arithmetic or logic operations are also stored here before being taken to main memory. Some execution cycles may require references to more than one memory, in which case more data registers will be needed to store the intermediate results and values.

A deeper look at the cycle could be instruction address calculation instruction fetch instruction operation decoding operand address calculation operand fetch data operation result storage.

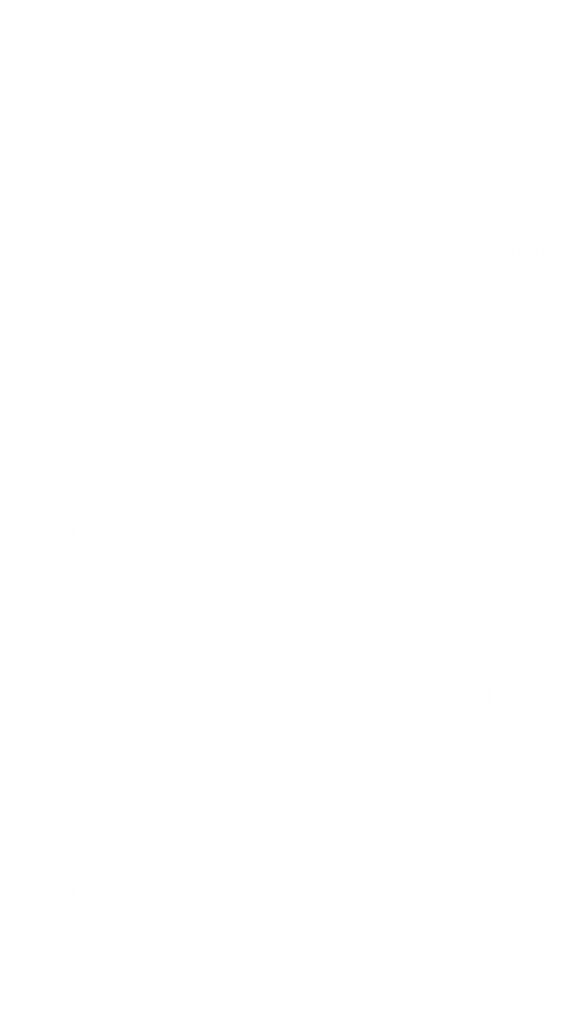
### Interrupts

Almost all computers provide a way for modules such as I/O modules and memory modules to interrupt the normal processing of the processor. Interrupts are primarily used to improve processing efficiency. For example, taking input from the user or printing something takes up a significant amount of time when we consider that a computer’s CPU has working times of nanoseconds. Without interrupts, the CPU would remain idle for this amount of time, which is a loss of hundreds if not thousands of potential instruction cycles. The following diagram shows what this process would look like:



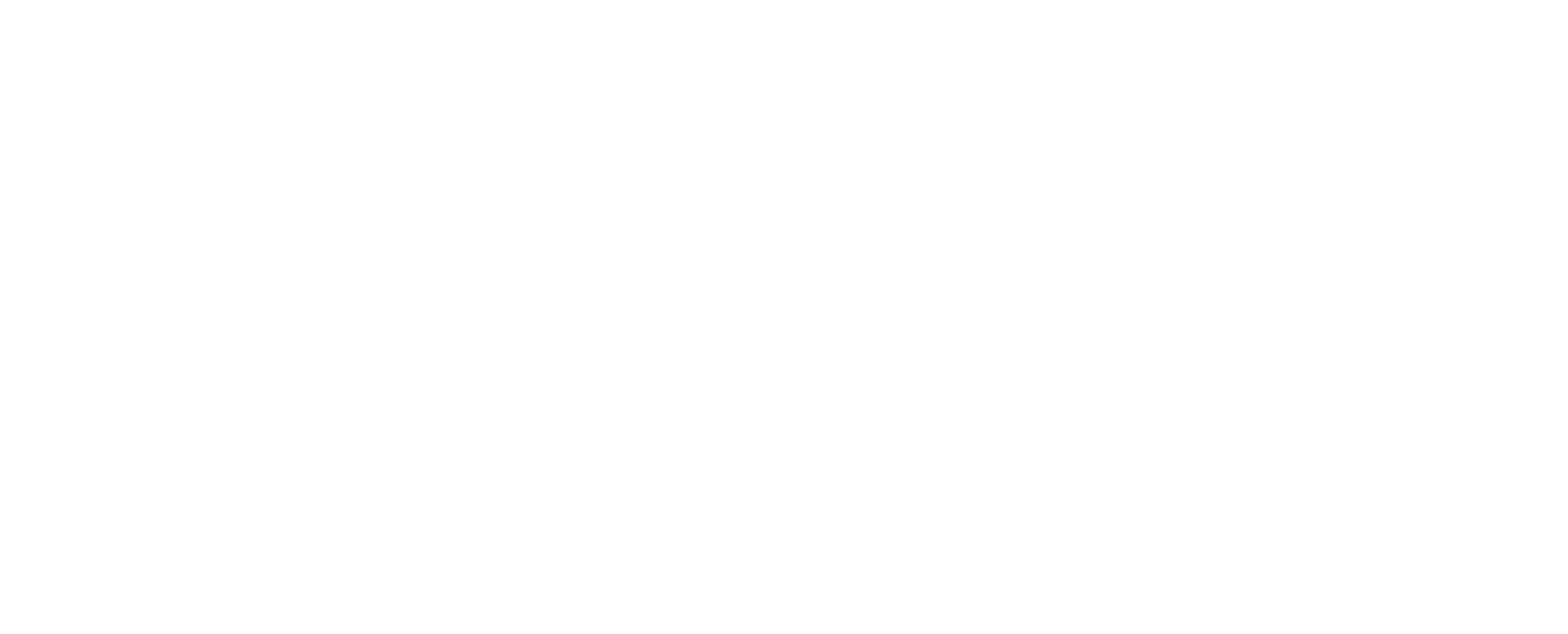
Code segments 1, 2 and 3 are sets of instructions that do not require user input. Between each segment, control passes to the I/O program, which executes code segment 4, preparing the I/O device, takes input or displays output, and then executes code segment 5, closing the I/O device, before returning control to the user program. The space between segment 4 and 5 is the part where a large amount of time is wasted.

If interrupts are used, the processor can continue executing other instructions while the I/O operation is progress. This is what that looks like:



When an I/O operation is found, the processor executes the instructions needed to set it up, and moves on to the next instruction. Once the I/O operation has finished, the I/O program sends an interrupt request signal to the processor. The processor finishes the single instruction it was executing at that moment, and then goes over to the interrupt handler program where it finishes the I/O programs instructions before moving back to the user program to the point where it was interrupted, executing the instruction that comes after the instruction it was executing when it was interrupted. If the I/O operation takes a long time however, and another WRITE call is reached before the first one has finished executing, the user program will be forced to stop, wait for the first one to finish, start the second one and then continue execution of other instructions.

The following is an edited version of the instruction cycle, made to accommodate interrupts:



After every instruction the processor checks for interrupt requests. If there is one, it suspends execution of the current program and saves the address of the next instruction to be executed and any other data relevant to the processor’s current activity. It then sets the program counter to the starting address of the interrupt handler routine. The interrupt handler program is typically a part of the operating system and determines the nature of the interrupt and what actions are needed. This is also known as the Interrupt Service Routine (ISR).

Interrupts can be of a few types:

* Program Interrupt – Generated by some condition that occurs as a result of an instruction execution, like an arithmetic overflow or reference to memory space outside the user’s scope
* Timer Interrupt – Generated by a timer within the processor that allows the operating system to perform certain activities on a regular basis
* I/O Interrupt – Generated by an I/O controller
* Hardware Failure – Generated due to some error on the hardware side such as power failure

The user program is not involved in any way with interruptions and does not include any special code for it. The OS and processor are responsible for suspension of the user program during interrupts and resuming them at the same point.

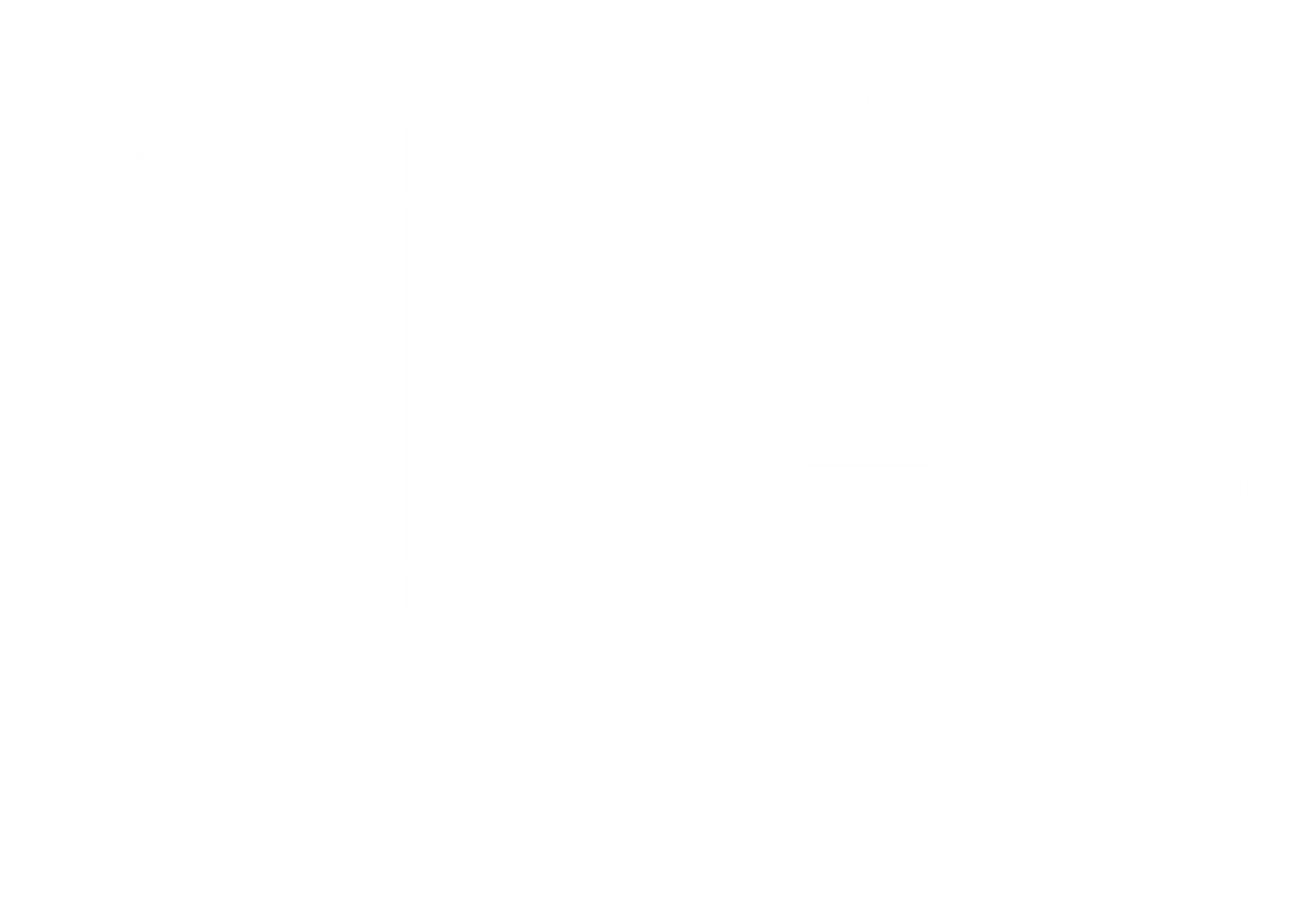
### Multiple Interrupts

It is possible for more interrupts to occur while a single interrupt is still processed. For example, a communications interrupt may occur in the middle of a printer interrupt. There are two ways to handle this situation, disable the interrupts or prioritize them.

A disabled interrupt simply means that the process will ignore further interrupts while one interrupt is being processed. Those interrupts remain pending. When the current interrupt has finished, control is passed back to the user program, where it is seen that there are pending interrupts, and control is immediately passed to them. This is the approach taken by the processor when there is an interrupt in the middle of executing a single instruction in a user program. The interrupt has to wait until the end of that particular instruction to be given control. It is sequential interrupt processing.

The disabled interrupt approach has a major problem though. It does not recognize relative priority or time-sensitive needs. For example, if a user is trying to forcefully exit a program, that interrupt needs to be given control immediately, without having to wait for other interrupts to finish or the current user program instruction ending. For communications interrupts, the data being given needs to be stored immediately to make room for more data.

This takes us to the second approach, which takes interrupt priority into account, allowing interrupts of higher priority to cause the current interrupt itself to be interrupted. Once the interrupt of higher priority has finished executing, control is passed back to the previous interrupt. If there is another interrupt in the middle of the second interrupt, which has a lower priority than the second interrupt but a higher priority than the first one, then it has to wait for the second interrupt to finish. One it has, control is passed back to the first interrupt, and from there to the third interrupt. This is nested interrupt processing. It looks a little like this:



In the middle of the user program, there was a printer interrupt, and in the middle of the printer interrupt, there was a communications interrupt. Since the communications interrupt had a higher priority than the printer interrupt, control was immediately given to the communications interrupt. During the communications interrupt, there was a disk interrupt, but since it had a lower priority it had to wait. At the end of the communications interrupt, control was given to the printer interrupt, from where it was seen that there was a disk interrupt pending, and control was immediately given to that.

When an interrupt does occur, we need to store the data from the current program or interrupt. This includes all of the MAR, MBR and other registry values. The reason for this is that the program still needs that data, but switching to the interrupt with cause them to be overwritten. The process of storing this old data is called context switching.

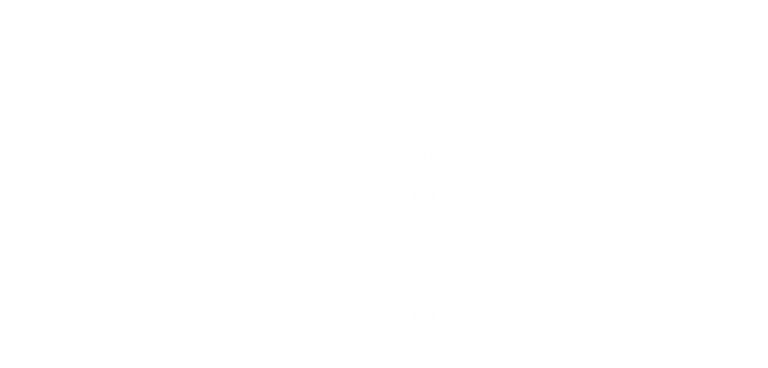
### I/O Function

The I/O module exchanges data with the processor, similar to how the processor can initiate a read or write with memory, with the processor identifying a specific device that is controlled by a particular I/O module. An instruction cycle similar to the ones we have already seen can be formed for this, with I/O instructions instead of memory-referencing instructions.

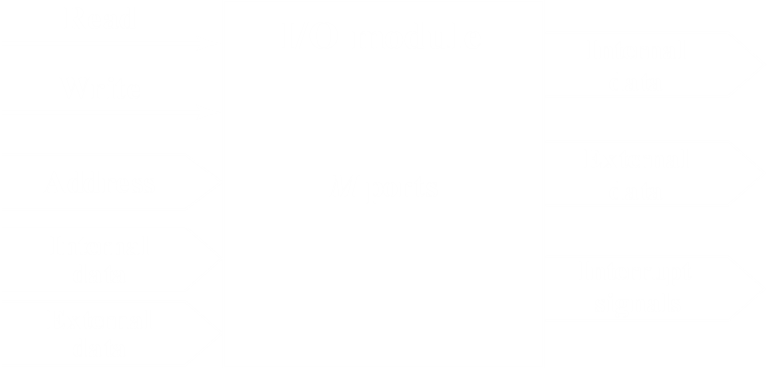
In some cases, the I/O module for a single device may need to exchange a very large amount of data. This would cause too many interrupts for the processor, which would be prevented from performing its normal tasks. In these scenarios, the I/O module is allowed to bypass the processor and directly issue read and write instructions to memory. This operation is known as direct memory access (DMA).

## 3.3 Interconnection Structures

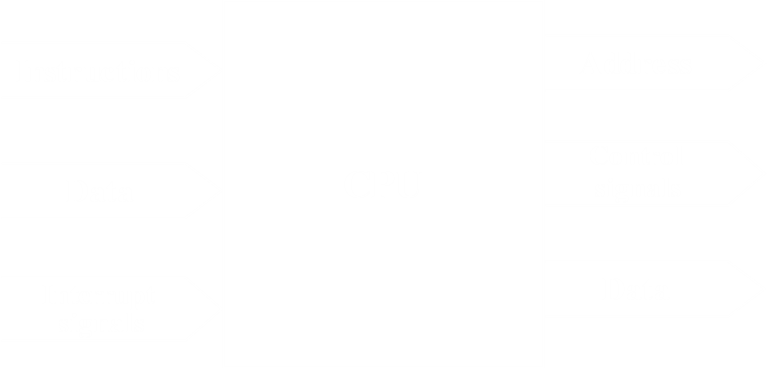
A computer consists of a set of components or modules of three basic types, processor, memory and I/O, that communicate with each other. A computer is thus a network of basic modules. The paths connecting the various modules is called the interconnection structure.



The memory module usually consists of words of equal length, each with a unique numerical address. A word of data can be read from or written into the memory. The nature of the operation is specified by the read or write input signals. The location for the operation is indicated by the address input signal.



The I/O module has two operations, read or write, each accompanied by an address for the operation. It can control external devices, each with a unique address. It can read or write data from internal devices or external devices and can send interrupt signals to the processor.



The processor reads instructions and data, and gives out data. It controls the overall operations of the system with control signals, along with addresses for the operations. It can also take in interrupt signals.

Thus, we can see that the interconnection structure needs to support transfers from:

* Memory to Processor: The processor reads instruction or data from memory.
* Processor to Memory: The processor writes data to memory.
* I/O to Processor: The processor reads data from I/O modules.
* Processor to I/O: The processor sends data to I/O devices.
* I/O to and from Memory: The I/O module is allowed to exchange data with the memory directly, using direct memory access.

Two interconnection structure in particular will be discussed, the bus structure and the point-to-point interconnection structure.

## 3.4 Bus Interconnection

Bus was the main means of component interconnection for a very long time, but has now given way to point-to-point interconnections for general-purpose computers. However, they are still widely used in embedded systems, particularly microcontrollers.

A bus is a communication pathway connecting two or more devices. The key feature is that the bus is a shared transmission medium. Multiple devices connect to the bus, and anything transmitted by any device is available for reception by all other devices. However, the sender attaches an address to the data, so that the receiver can identify which pieces of data to take.

The order of data in a bus is maintained by a timestamp. However, this means that if two devices were to transmit data at the exact same time, their signals would get mixed up, destroying both pieces of data. In reality, components do not directly send data. They make a request to a controlling component, the bus master, which has the job of directing which pieces of data go when. It does not use the bus itself. It simply manages everything.

Typically, a bus is made up of several communication pathways, or wires, with each wire transmitting a signal representing either a binary or . Thus, a sequence of binary digits can be transmitted across a single line, while taken together, several bits can be sent parallelly by using multiple lines. For example, bits of data could be transmitted together using 8 bus lines.

A typical computer system will contain a number of different buses that provide pathways between components at different levels of hierarchy. Most major components like the processor, memory and I/O, are connected using the system bus. Most common computer interconnection structures are based on the use of one or more system buses.

A system bus typically has 50 to 100 separate lines, each line being associated with a separate action. These lines can be grouped into the categories data lines, address lines, control lines and perhaps power distribution lines.

Data lines, collectively known as the data bus, provide paths for moving data. They may consist of 32, 64, 128 or more lines, this number being referred to as the width of the data bus. As we previously saw, the number of lines represents the number of bits that can be transferred at a time. The width of the data bus is important to the performance of the system. For example, if each instruction is 64 bits line but the data bus is only 32 bits wide, then the processor must access the memory module twice for every instruction cycle, which makes the processor slower than if the data bus had been 64 or more bits wide.

Address lines are used to identify the source or destination of the data on the data bus. The width of the address bus determines the maximum possible memory capacity of the system. If we have a 32-bit address bus, then we can have addresses, since a single bit can be used for 2 addresses. This limits us to gigabytes of memory. The address bus is also used to identify I/O ports. Typically, the lower order bits (for example 01111111 and below) are used to identify memory locations, while the higher order bits (for example 10000000 and above) are used to identify devices attached to an I/O module.

The control lines are used to send commands, thus controlling access and use of the data and address lines. They also transmit timing information, which validates data and address information. Typically, we can have a few types of control lines:

* Memory write, to cause data on the bus to be written into the address location
* Memory read, to cause data from the address location to be placed on the bus
* I/O write, to cause data on the bus to be outputted to the addressed I/O port
* I/O read, to cause data from the addressed I/O port to be placed on the bus
* Transfer ACK, to indicate that data has been accepted from or placed on the bus
* Bus requests from modules to gain control of the bus
* Bus grants, to grant control of the bus to a requesting module
* Interrupt requests, indicating pending interrupts
* Interrupt ACK, indicating pending interrupts have been recognized
* Clock, used to synchronize operations
* Reset, used to initialize all modules

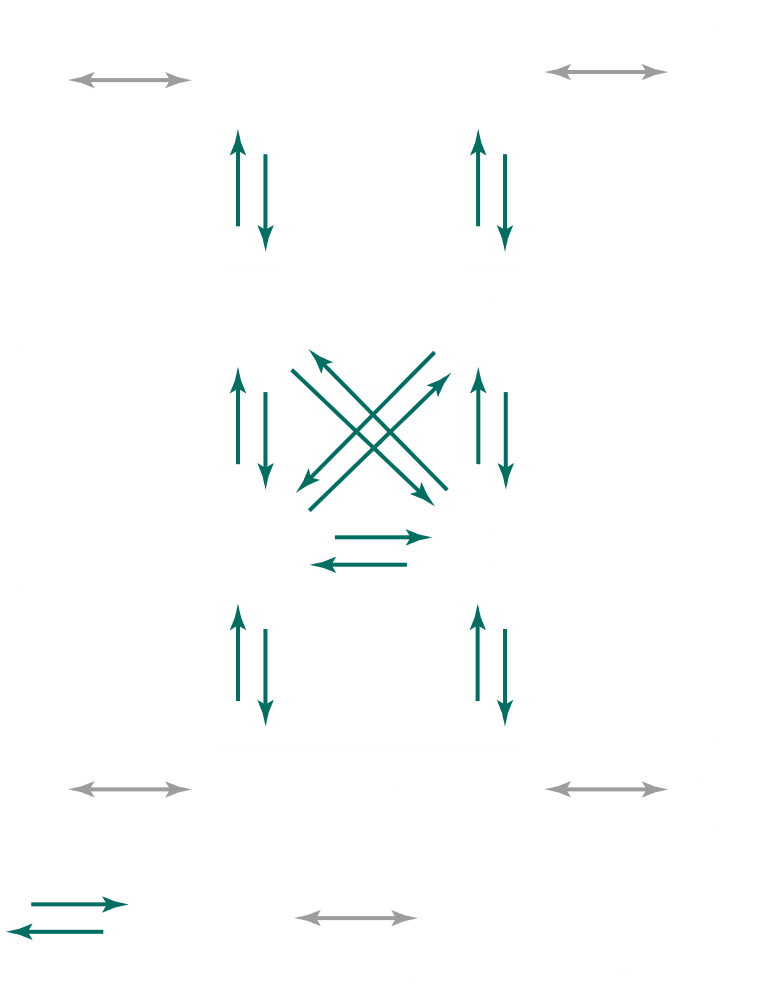
To send data, a module must obtain use of the bus, and then transfer the data via the bus. To obtain data, a module must obtain use of the bus, transfer a request to the other module using appropriate control and address lines, and then wait for the data to be transferred by the other module.

## 3.5 Point-To-Point Interconnect

Nowadays, systems have increasingly switched to point-to-point connections. The reason behind this is the electrical constraints encountered with using higher and higher data rates, which makes performing synchronization and arbitration (breaking up fights) tasks in a timely manner difficult. Furthermore, multicore chips, multiple processor and significant memory on a single chip has magnified the difficulties of increasing bus data rate and reducing bus latency to keep up with the processor. Point-to-point interconnections have lower latency, higher data rate and better scalability.

We will be looking that an important example of the point-to-point interconnect approach, Intel’s QuickPath Interconnect (QPI), introduced in 2008. It allowed:

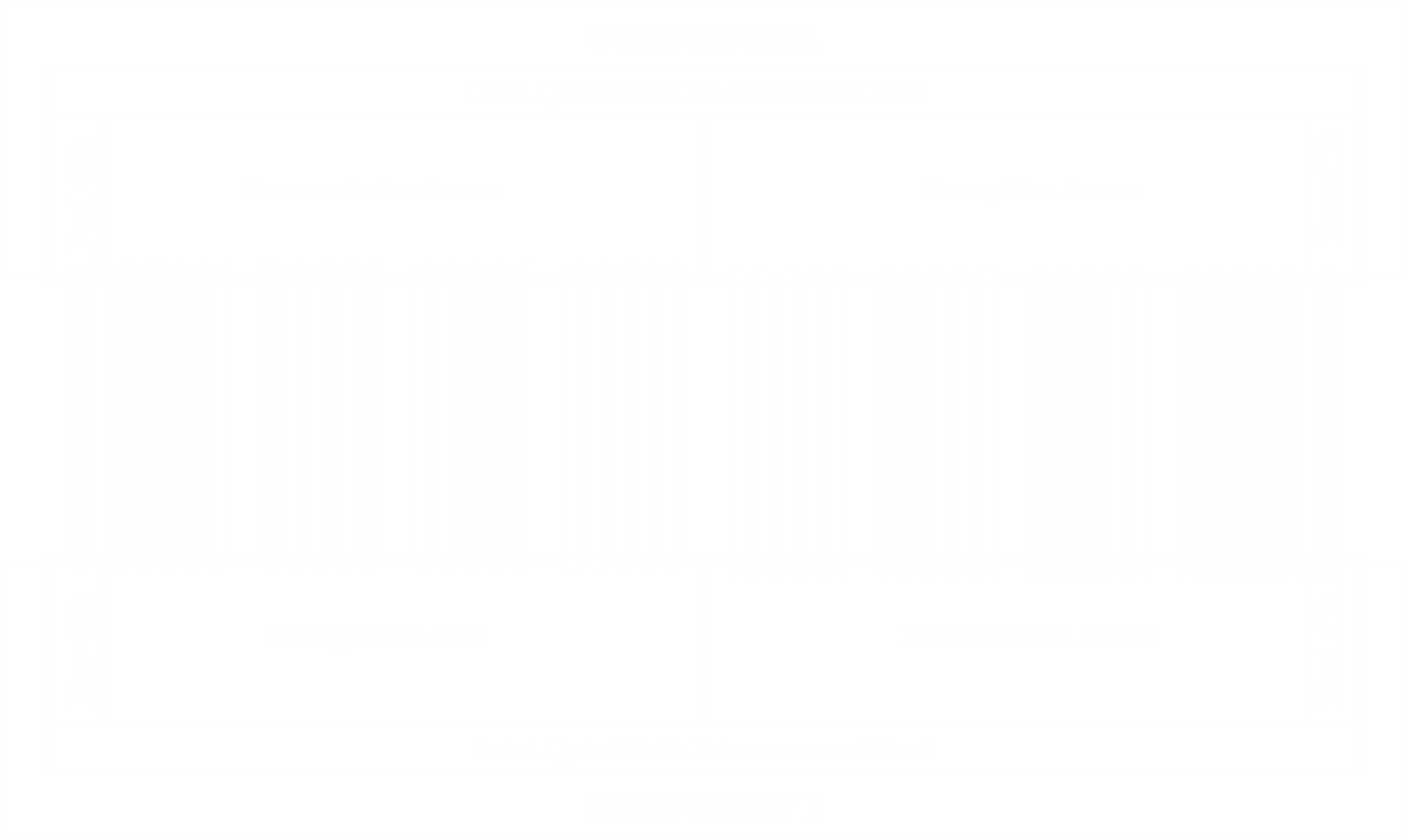
* Multiple direct connections, so multiple components within the system could enjoy direct pairwise connections to other components, eliminating the need for arbitration
* Packetized data transfer, so data was not sent as a raw bit stream but rather as a sequence of packets, each including control headers and error control codes
* Layered protocol architecture, used at processor-level interconnects instead of simple control signals



In the above QPI configuration, notice that each core is connected to the others using a direct QPI connection. QPI is also used to connect to an I/O module, called the I/O Hub, which acts as a switch, directing traffic to and from I/O devices. In newer devices, the link from the I/O Hub to the I/O devices is made with an interconnect technology called PCI Expressed (PCIe), which will be discussed further below. Each core is also linked to a main memory module using a dedicated memory bus. The memory typically uses dynamic random-access memory (DRAM) technology.

QPI is a four-layer protocol architecture, consisting of the physical layer, the link layer, the routing layer and the protocol layer.

The physical layer consists of the actual wires being used to carry signals.



Each data path consists of a pair of wires, called a lane, that transmit data one bit at a time. There are 20 data lanes in each direction, which means the QPI can transmit 20 bits parallelly in each direction. The 20-bit unit is called a phit. There is also a clock lane in each direction, which brings the total up to 84 individual links.

The link layer is responsible for reliable transmission and flow control. It transfers data in units of 80 bits, called a flit. 72 bits consist of data, while 8 bits consist of an error control code. This is the reliable transmission part. The flow control part ensures that the receiving QPI entity is not receiving data faster that it can process data.

The routing layer determines the course that a packet will traverse.

The protocol layer consists of a high-level set of rules for exchanging packets of data between devices. A unit of data here is a packet, which consists of an integral number of flits.

When data is being sent, the protocol layer first creates the packets, the routing layer maps out the path, the link layer distributes the data, and the physical layer does the transfer. When data is being received, the physical layer first transfers data, the link layer retrieves the data, the routing layer plots out a path back, and the protocol layer unpacks the data.

## 3.6 PCI Express

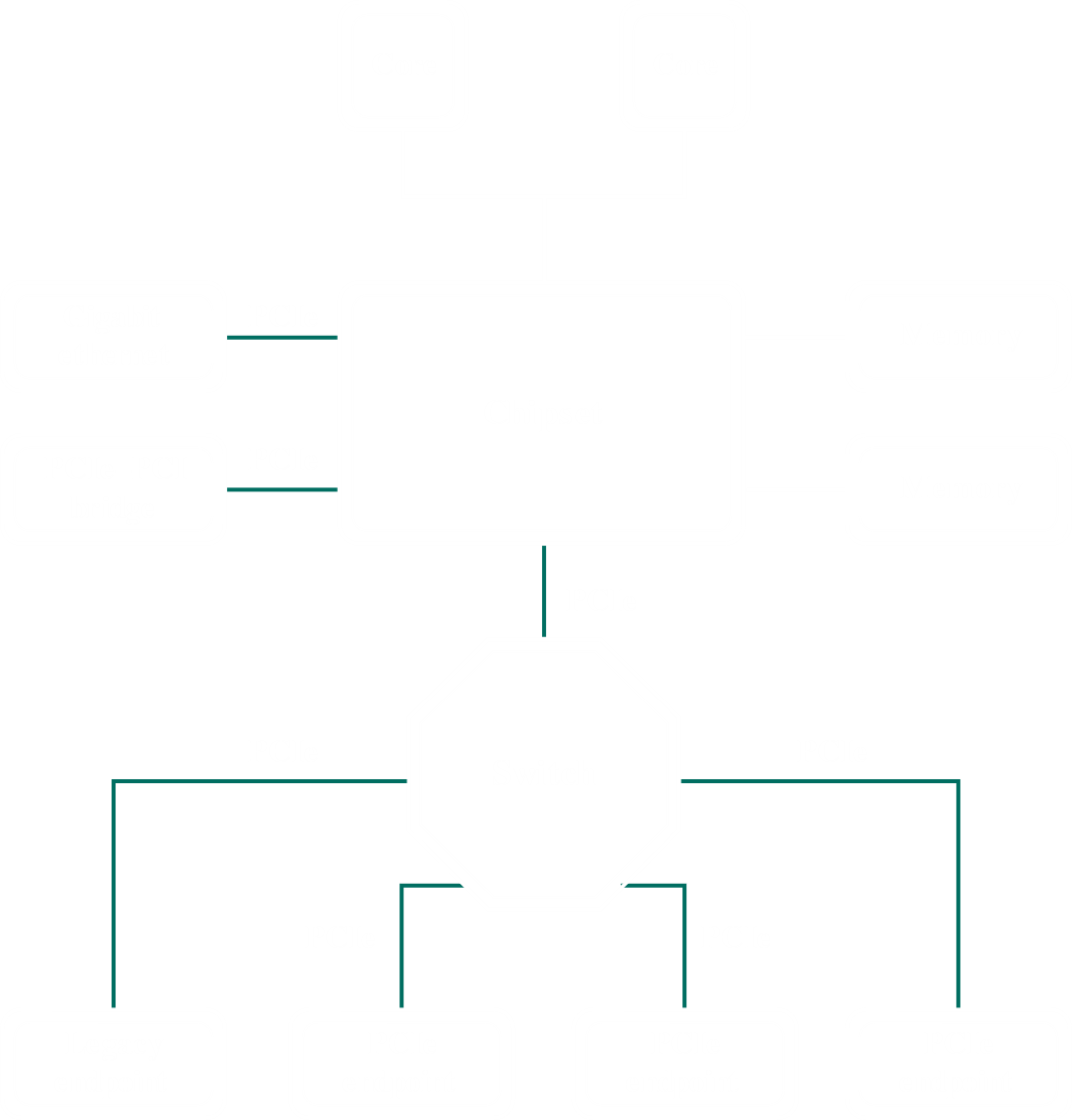
The peripheral component interconnect (PCI) is a popular, high-bandwidth, processor-independent bus that can function as a peripheral bus. Intel began working on it in 1990. Compared with other common bus specifications, PCI delivers better system performance for high-speed I/O subsystems, like graphics display adapters, network interface controllers and disk controllers. Intel released all patents to the public domain which led to PCI being widely adopted and products built by different vendors being compatible.

As with the system bus, the PCI scheme was unable to keep up with data rate demands of devices, which led to the development of a new version, PCI Express (PCIe). It is a point-to-point interconnect scheme intended to replace bus-based schemes like PCI.

Key requirements of PCIe include a high capacity to support higher data rate I/O devices, and the ability to support time-dependent data streams. Applications like video-on-demand and audio redistribution put real-time constraints on servers. Many communication applications and embedded PC control systems also process data in real-time. Multiple concurrent transfers at ever-increasing data rates must also be dealt with. Data needs to be tagged so that an I/O system can prioritize its flow through the platform, since things like streaming data will become useless if they are not delivered in real-time.

In PCIe configuration, a chipset, also called a host bridge, connects the processor and memory subsystems to the PCIe switch fabric, which connects to one or more PCIe or PCIe switch devices. The chipset also acts as a buffering device, supporting the different data rates from components, and as a translator between processor, memory and I/O modules. It typically supports multiple PCIe ports to connect PCIe devices or switches. It may connect with:

* Switches, which manage multiple PCIe streams
* PCIe devices, an I/O device with PCIe support
* Legacy devices, that have been migrated to PCIe but still allows legacy behaviour
* PCIe/PCI bridge, to allow older PCI devices to be connected



PCIe interactions are defined using a protocol architecture, that consists of the following layers:

* Physical layer, consisting of the actual wires carrying signals and circuitry and logic to support necessary features.
* Data link layer, responsible for reliable transmission and flow control. Data packets generated and consumed here are called Data Link Layer Packets (DLLPs).
* Transaction layer, which generates and consumes data packets use to implement load and store data transfer mechanisms, and manages the flow control of the packets between two components on a link. Data packets generated and consumed here are called Transaction Layer Packets (TLPs).

Above all this, there is the software layer, which generates read and write requests.