**Chapter 7: Input/Output**

Table of Contents

[7.1 External Devices 4](#_Toc48984311)

[7.2 I/O Modules 5](#_Toc48984312)

[7.3 Programmed I/O 7](#_Toc48984313)

[Overview of Programmed I/O 8](#_Toc48984314)

[I/O Commands 8](#_Toc48984315)

[I/O Instructions 10](#_Toc48984316)

[7.4 Interrupt-Driven I/O 12](#_Toc48984317)

[Interrupt Processing 13](#_Toc48984318)

[Design Issues 16](#_Toc48984319)

[Multiple Interrupt Lines 16](#_Toc48984320)

[Software Poll 16](#_Toc48984321)

[Daisy Chain 17](#_Toc48984322)

[Bus Arbitration 17](#_Toc48984323)

[Intel 82C59A Interrupt Controller 18](#_Toc48984324)

[Intel 8255A Programmable Peripheral Interface 20](#_Toc48984325)

[7.5 Direct Memory Access 22](#_Toc48984326)

[Drawbacks of Programmed and Interrupt-Driven I/O 22](#_Toc48984327)

[DMA Function 22](#_Toc48984328)

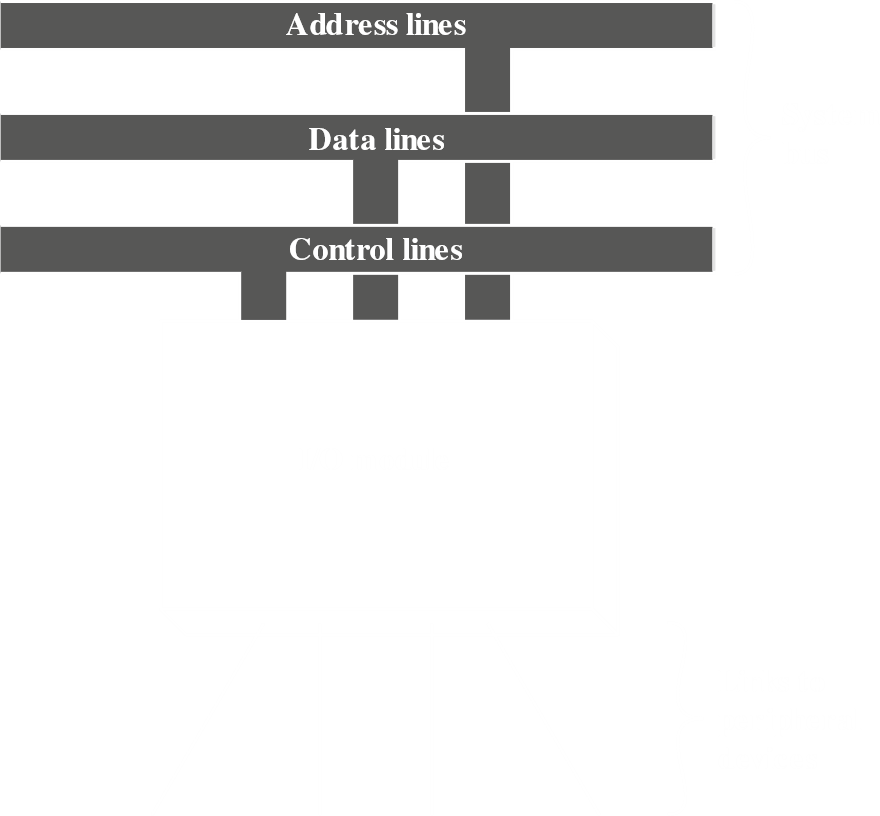
[7.6 Direct Cache Access 27](#_Toc48984329)

[DMA Using Shared Last-Level Cache 27](#_Toc48984330)

The connection between external input/output (I/O) devices and the system bus is managed by I/O modules. These modules are not simply connecting the two parts, but rather contains logic to handle the communication between them. This is required for a few reasons:

* Peripheral devices operate in a variety of methods and incorporating the necessary logic to handle these directly into the processor would be impractical.
* The data transfer rate of peripherals can be slower or faster than the memory or the processor, which means the module is required to handle the mismatch in speeds.
* Peripherals may use different data formats and word lengths.

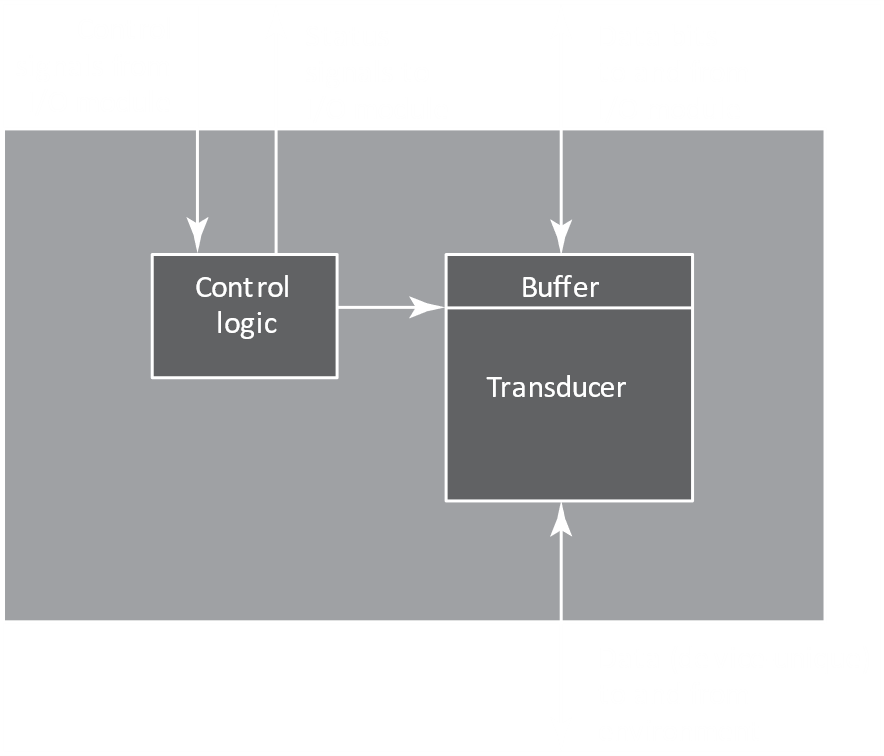
Thus, an I/O module handles the interface to the processor and memory via the system bus or central switch and the interface to one or more peripherals by tailored data links.



## 7.1 External Devices

External peripherals can be broadly classified into three categories: human readable, like displays and printers, used to communicate with users, machine readable, like magnetic tapes, used to communicate with equipment, and communication, used to communicate with remote devices.

In very general terms, the nature of an external device can be described with the following diagram. Control signals contain information about how the device should function, status signals indicate the state of the device to the module, data bits are stored in the buffer while it is being transferred and the transducer converts data from electrical signals to other forms and vice versa.



A simple example of how all this would work is taking some text as input from the keyboard and displaying it on a monitor, or reading some data from the hard disk and writing some data back. The external device sends data to the I/O module, the I/O modules sends it to the CPU for processing, the output goes back to the same or a different I/O module as required, which then passes it on the external peripheral.

## 7.2 I/O Modules

The major functions of an I/O module include:

* Control and Timing
* Processor Communication
* Device Communication
* Data Buffering
* Error Detection

At any given time, the processor can demand communication with multiple external devices in an unpredictable pattern. Since internal resources like main memory and the system bus are shared between activities, control and timing is required to coordinate the flow of data to and from external devices.

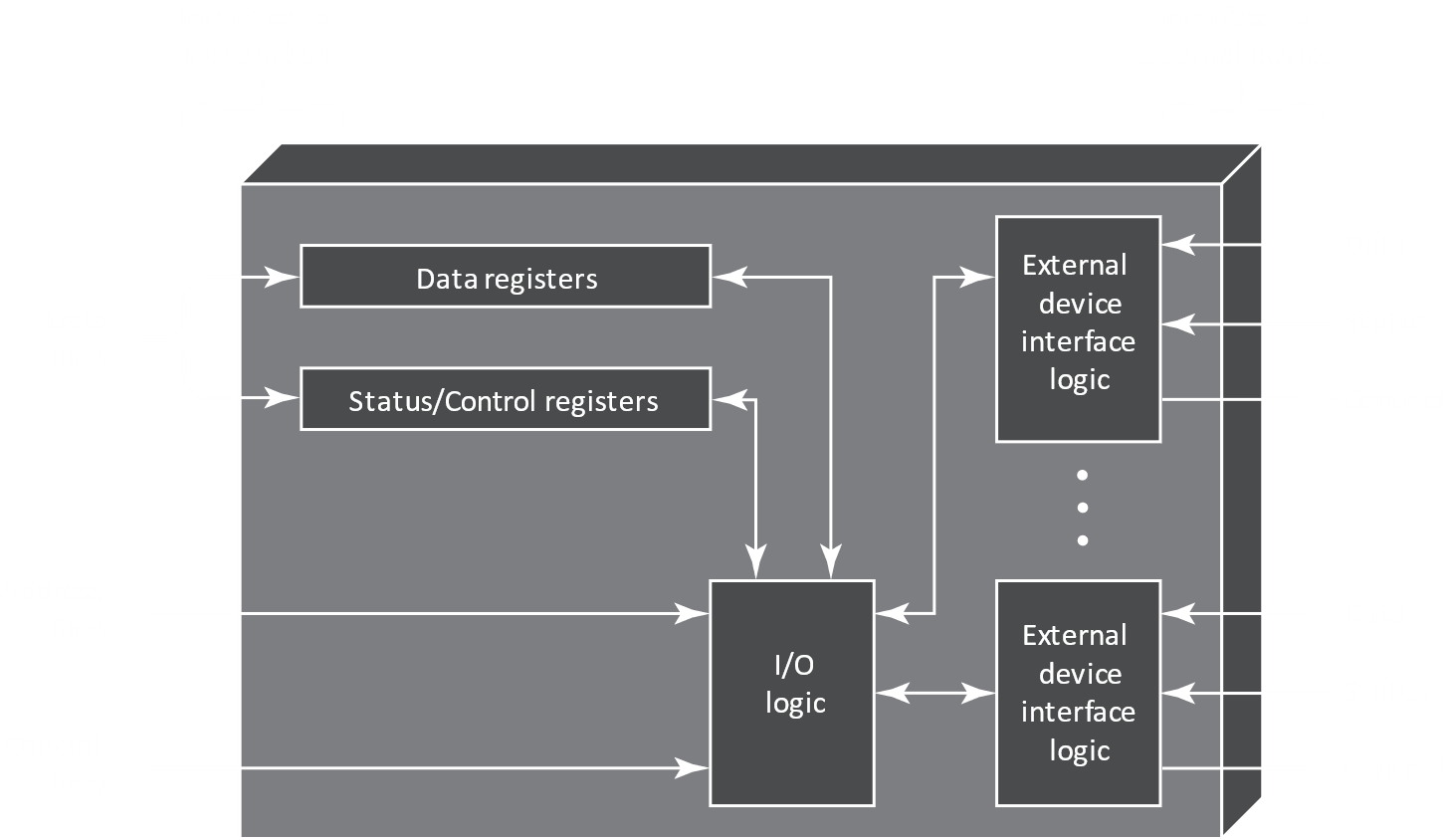
Processor communication is required since the I/O module must be able to communicate with the processor. This involves command decoding, exchanging data, reporting the status of peripherals and address recognition for the I/O devices.

Device communication, required for the I/O module to be able to communicate with external devices, involves commands, status information and data.

Data buffering is required to maintain a synchronization between the different speeds of internal and external devices, as previously discussed.

I/O modules also perform some error detection, like electrical malfunctions reported by the external devices or unintentional changes in transmitted data, detected with parity bits.

A general diagram for an I/O module is shown below. Registers store data or control and status information and the I/O logic uses control lines to receive commands from the processor. The other parts should be obvious.



An I/O module tends to simplify the view of devices from the processor’s perspective, handling the details. However, it may also leave much of the work to the processor. An I/O module that takes up most of the detailed processing burden is called a I/O channel or I/O processor and are mostly seen on mainframes. A simpler I/O module is called an I/O controller or device controller and are commonly seen on microcomputers.

## 7.3 Programmed I/O

There are three techniques possible for I/O operations, programmed I/O, interrupt-driven I/O and direct memory access (DMA). In programmed I/O, the processor is given direct control over the I/O operation. Thus, it must wait after issuing a command for the I/O operation to complete. This is a waste of processing time. With interrupt-driven I/O, the I/O module is given control of the I/O operation and is instructed to inform the processor when the operation has completed, thus allowing the processor to handle other tasks in the meantime. In DMA, the I/O module and main memory exchange data directly, without processor involvement.

I/O operations are performed through three techniques, programmed I/O, interrupt-driven I/O and direct memory access (DMA).

With programmed I/O, data is exchanged between the processor and the I/O module. The processor executes a program that gives it direct control over all I/O operations, including sensing device status, sending read or write commands and transferring data. When it issues a command, it must wait for the I/O module to complete the I/O operation. If the processor is faster than the I/O module, this results in wasted processor time.

With interrupt-driven I/O, the processor issues an I/O command and moves on to execution of other instructions. When the I/O module has completed its work, it interrupts the processor to let it know that it has finished its work and is ready for more commands.

With both programmed and interrupt-driven I/O, the processor is in charge of extracting data from the main memory for output and storing data in main memory for input. The alternative is known as direct memory access, or DMA. In DMA, the I/O module and main memory exchange data directly without any processor involvement. DMA also makes use of interrupts to let the processor know when it has completed its command.

### Overview of Programmed I/O

When the processor is executing a program and finds an instruction related to I/O, it issues a command to the appropriate I/O module. With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register. The I/O module does not do anything else to alert the processor. Specifically, it does not interrupt the processor. Thus, it is the processor’s job to periodically check the status of the I/O module until it finds that the operation has been completed.

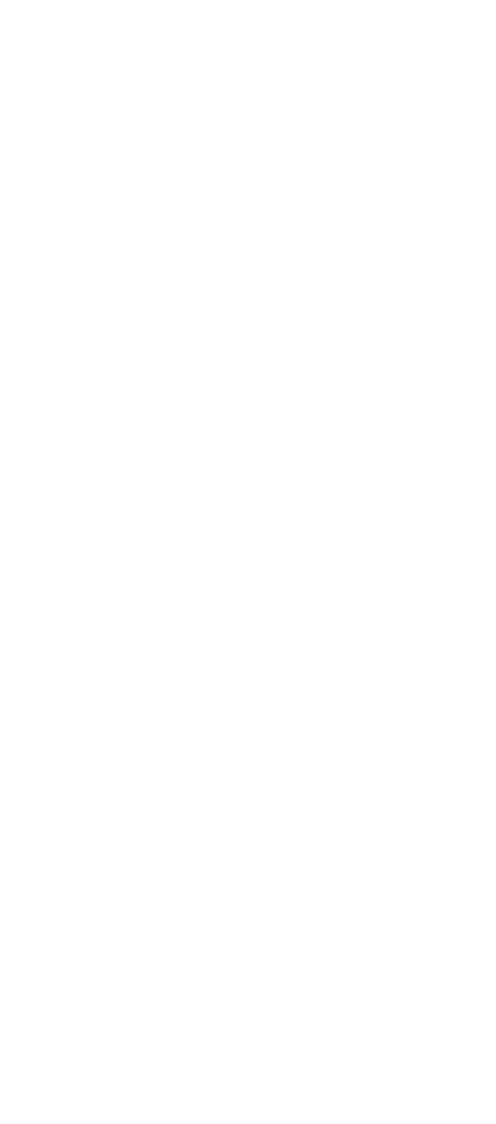
To further explain the programmed I/O technique, we will be looking first at the point of view of the I/O commands issued by the processor, and then from the point of view of the I/O instructions executed by the processor.

### I/O Commands

To execute an I/O instruction, the processor issues an address for a particular I/O module and external device, along with the I/O command. The I/O command can be of four types:

* Control – Used to activate a peripheral and tell it what to do, e.g. telling a magnetic tape to rewind some distance. Control commands are specific to the particular type of peripheral device.
* Test – Used to test status conditions of the I/O module and peripherals, such as if the peripheral is powered on and available for use, or if the last I/O operation was completed and if there were any errors.
* Read – Makes the I/O module obtain an item of data from the peripheral and place it in an internal buffer, from where the processor can obtain it by requesting the I/O module to place it on the data bus.
* Write – Makes the I/O module take an item of data from the data bus and transmit it to the peripheral.

The following figure gives us an example of the use of programmed I/O to read a block of data from a peripheral device into memory. Data is read a word at a time, and for each word read, the processor must remain in a status-checking cycle until it determines the word is available in the I/O module’s data register. Thus, it is obvious that this technique is time consuming and needlessly keeps the processor busy.



### I/O Instructions

With programmed I/O, there is a close correspondence between the instructions the processor fetches from memory and the commands it issues to the I/O module. The instructions can easily be converted into I/O commands, with there often being a one-to-one relationship.

The form of the instruction depends on the way in which external devices are addressed. Typically, there are many I/O devices connected through I/O modules to the system, with each device having a unique identifier or address. When the processor issues an I/O command, the command contains the address of the required device. Thus, the I/O modules must interpret the address line to check if the command is for the device connected to it.

When the processor, main memory and I/O share a common bus, two modes of addressing are possible: memory mapped and isolated.

With memory mapped I/O, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O modules as memory locations, using the same instructions to access both memory and I/O devices. Essentially, some part of the range of addresses is used for actual memory, and some is used to identify I/O devices. The disadvantage here, is obviously that valuable memory address space is being shared with I/O devices.

With memory mapped I/O, a single read line and a single write line are needed on the bus. The alternative is to use 4 lines, 2 for memory read and write and two for input or output commands. The command line specifies whether the address refers to a memory location or an I/O device. This allows us to use the entire range of addresses for both memory and I/O devices and is the method used in isolated I/O, given the name because the address space for I/O is isolated from that for memory. The disadvantage here is that we need completely separate sets of instructions for memory addressing and I/O device addressing.

The essential idea is, with memory mapped I/O, we are dividing the memory addresses available into two parts, one to identify the I/O devices and another to actually store data. With isolated I/O, we are using special commands to access I/O ports, which means we can use the entire range of addresses for memory when those commands are not being used, and do the same to identify I/O ports when those commands are being used.

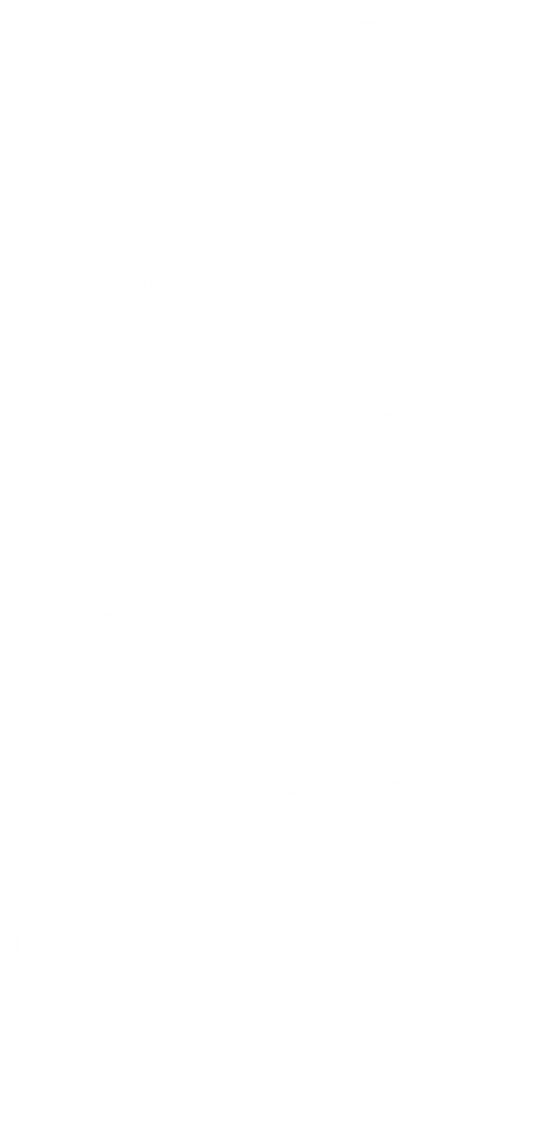
## 7.4 Interrupt-Driven I/O

The problem with programmed I/O is that the processor must wait for the I/O module to either be ready to receive or transmit data, repeatedly checking the status of the module. This degrades the performance of the entire system. The alternative is the issue a command to the module and then continue to do other work. Once the module is ready to exchange data, it can then interrupt the processor to request service. This is the method followed in interrupt-driven I/O.

From the point of view of the I/O module, it receives a READ command from the processor, and then proceeds to read data in from the appropriate peripheral. Once the data is in its data register, the module signals an interrupt to the processor over a control line. When the request is made by the processor, it places the data on the data bus.

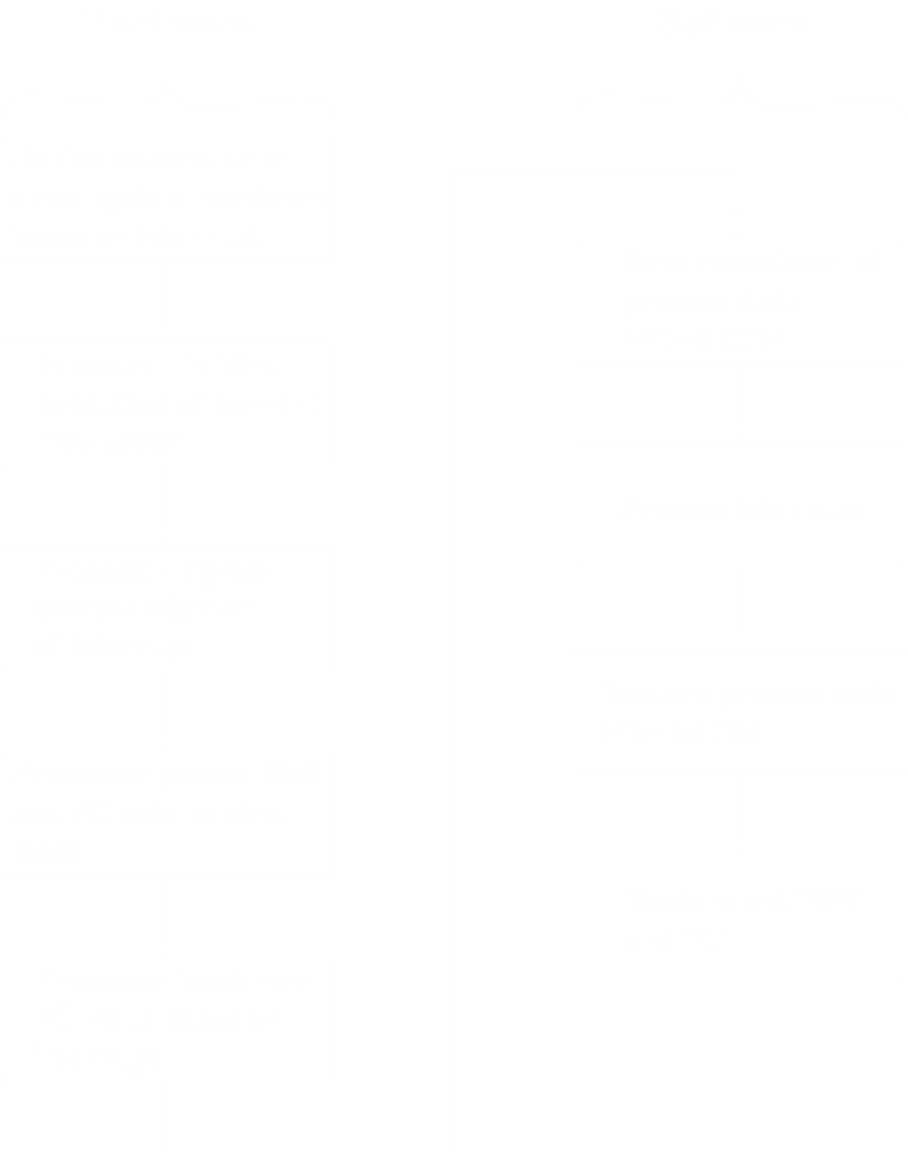
From the point of view of the processor, it issues a READ command and then proceeds to do other work. At the end of each instruction cycle, the processor checks for interrupts. Once it finds an interrupt request from the I/O module, it saves the context of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program it was working on and resumes execution.

From the figure below, we can see that interrupt driven I/O is more efficient than programmed I/O, since it eliminates needless wait times. However, it still consumes a lot of processor time, since every word of data that goes from memory to the I/O module or from the I/O module to memory must pass through the processor.



### Interrupt Processing

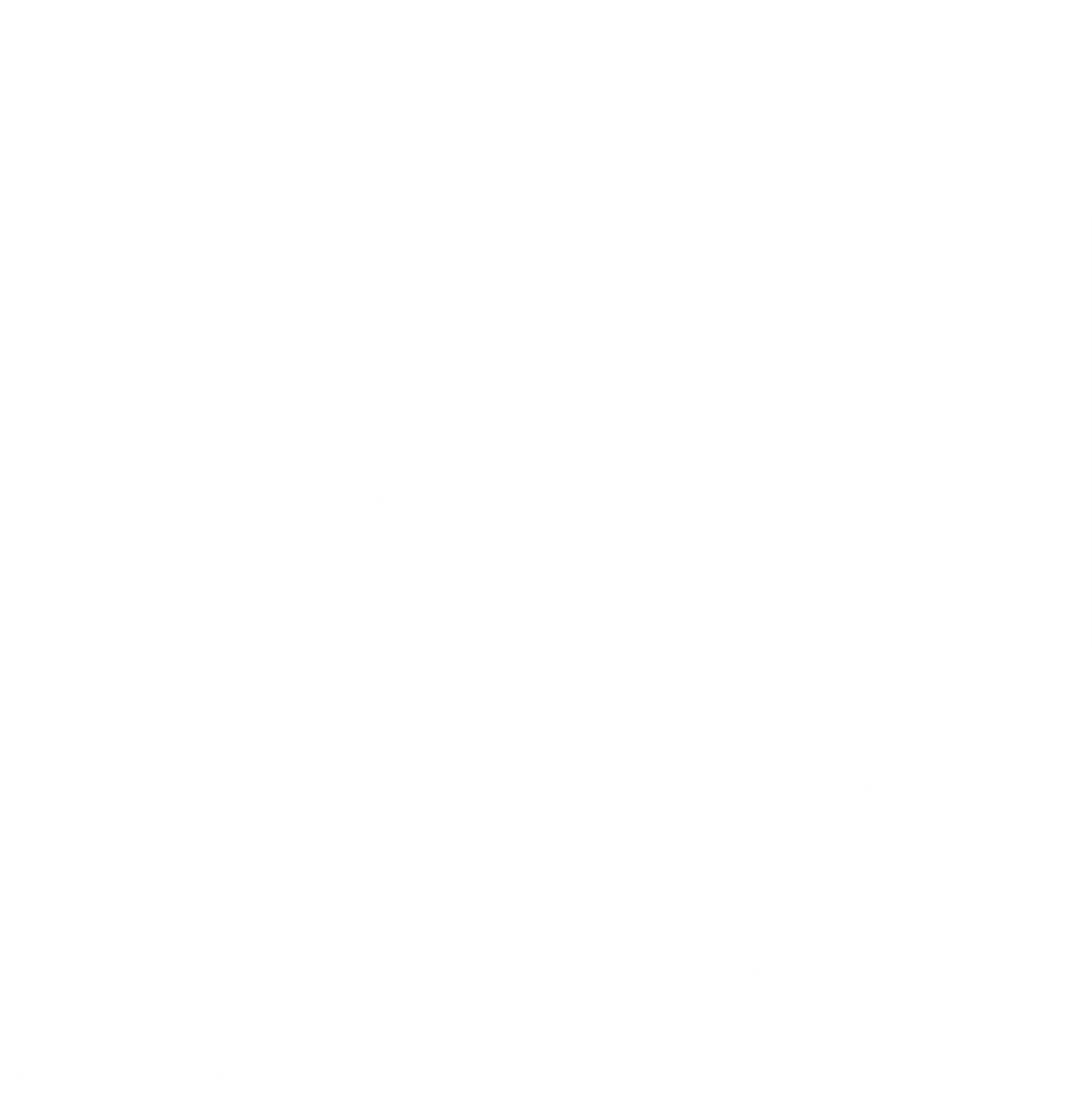
From the processor’s point of view, an interrupt will trigger multiple events, both in hardware and in software:



1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor checks for an interrupt and finds one. It sends an acknowledgement signal to the device, so that it can remove its interrupt signal.
4. The processor needs to save information needed to resume the current program from the point of interrupt before giving control to the interrupt routine. The status of the processor, contained in the program status word (PSW) and the location of the next instruction to be executed, contained in the program counter, are pushed onto the system control stack.
5. The processor loads the program counter with the entry location of the interrupt-handling program. Depending on the computer architecture and OS design, there may be one program, a separate program for each type of interrupt or even a separate program for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may already be in the original interrupt signal, or the processor may have to request the respective device for the information.

Once the program counter is loaded, the processor goes to the next instruction cycle, fetching the instruction from the program counter. The program counter contains the instructions for the interrupt-handler program at this point, so control is passed to that. This program then continues the operation.

1. There is more information, such as the processor registers, that is considered to be part of the state of the executing program and must be saved. Typically, the interrupt-handler saves the contents of the registers on the stack. This is shown below. The interrupt occurs at instruction , so the contents of all the registers and the address of the th instruction are pushed onto the stack. The stack pointer is updated to point to the new top of the stack, and the program counter is updated to point to the beginning of the interrupt service routine (ISR).
2. The interrupt is now processed by the interrupt-handler. It examines status information about the I/O operation and other events that caused the interrupt. It may also send additional commands or responses to the I/O device.
3. When the interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers, as shown below.
4. Finally, the PSW and program counter values are restored from the stack, so the next instruction executed will be from the interrupted program.



### Design Issues

There are two design issues that must be dealt with. First, if there are multiple I/O modules, how do we determine which module issued the interrupt? Second, if there are multiple interrupts, how do we decide which one to process?

For device identification, there are four general categories of techniques:

* Multiple Interrupt Lines
* Software Poll
* Daisy Chain
* Bus Arbitration

#### Multiple Interrupt Lines

Literally just have multiple lines between the processor and I/O modules. This is impractical though. Plus, even if we have multiple lines, we would most likely have multiple modules on each line which means we will need to use one of the other methods anyways.

#### Software Poll

When the processor finds an interrupt, it branches to an interrupt-service routine, and that in turn polls each I/O module to find the one that caused the interrupt. This could be in the form of a separate command line, where the processor raises TEST I/O and places the address of a particular I/O module on the address line. The corresponding I/O module responds positively if it issued the interrupt. Alternatively, each I/O module could have an addressable status register, so the processor can read the status register of each I/O module to find the correct one. Once the correct module is found, the processor branches to a device-service routine specific to the device.

#### Daisy Chain

Software polls are very time consuming. Daisy chain is more efficient, since it is essentially a hardware poll. All I/O modules have a common interrupt request line and the interrupt acknowledge line is daisy chained (strung through) the modules. The processor sends an interrupt acknowledge when it finds an interrupt, which propagates through a number of modules until it finds the correct one. The requesting module typically responds by placing a word on the data lines. The word is referred to as the vector, and it is either the address of the module or some other unique identifier. The processor can then use the vector as a pointer to the correct device-service routine. This avoids the need to execute a general interrupt-service routine first, and the technique is called the vectored interrupt.

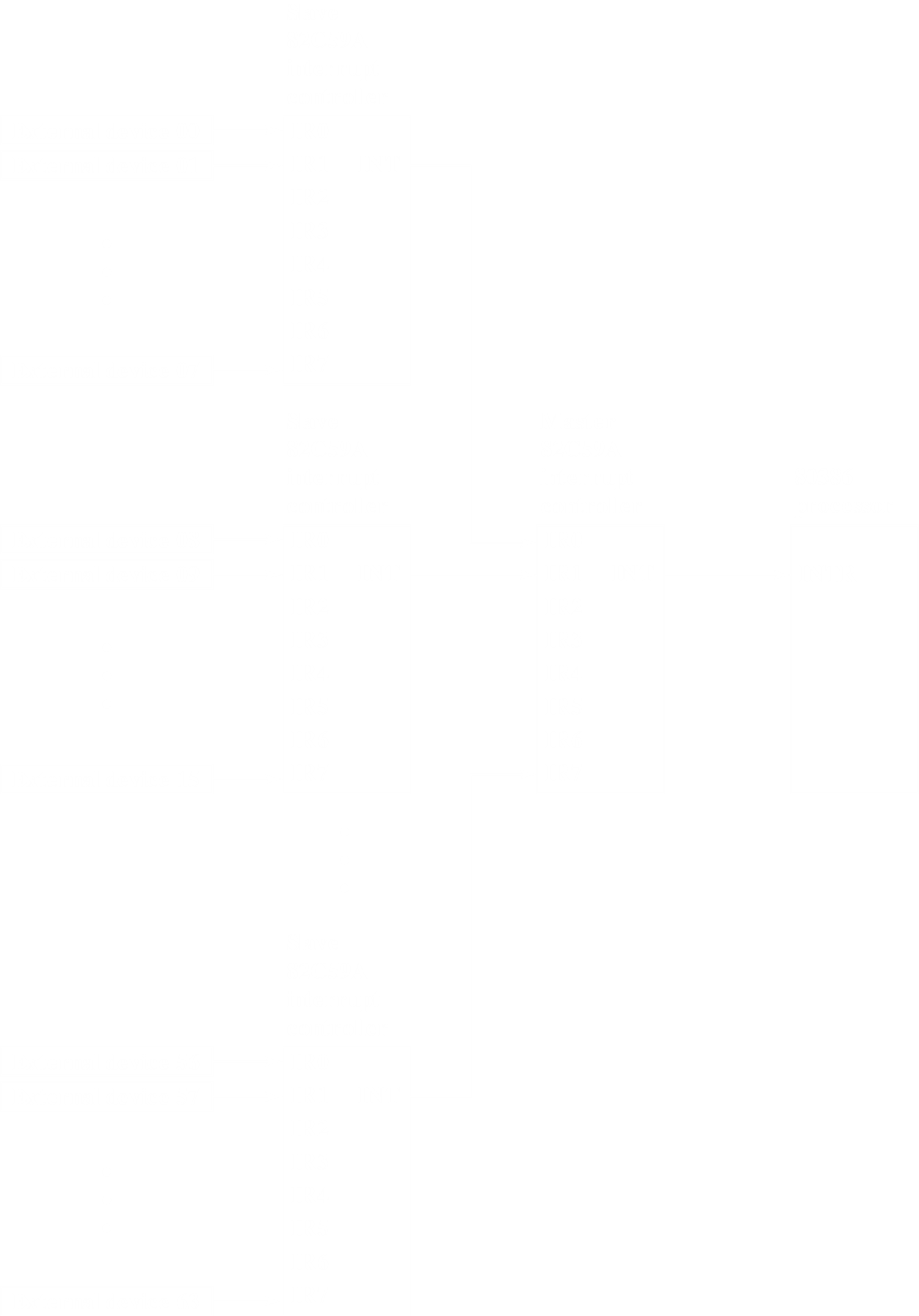
#### Bus Arbitration

This also makes use of vectored interrupts. Here, the I/O module must first gain control of the bus before it can raise an interrupt request line, meaning only one module can raise the line at a time. When the processor finds the interrupt, it responds on the interrupt acknowledge line and the requesting module places its vector on the data lines.

The techniques outlined above can also give us a way to prioritize multiple devices requesting the interrupt service. With multiple lines, the processor just picks the interrupt line with the highest priority. With software polling, the order in which modules are polled determines priority. Similarly, the order of modules on a daisy chain determines priority. With bus arbitration, a priority scheme is adopted.

### Intel 82C59A Interrupt Controller

We will now look at two interrupt structures. The first is the Intel 82C59A Interrupt Controller. The Intel 80386 provides a single Interrupt Request (INTR) and a single Interrupt Acknowledge (INTA) lie. To allow the 80386 to handle a variety of device sand priority structures, an external interrupt controller, the 82C59A, is attached to it. External devices get connected to the 82C59A. A single 82C59A can handle up to 8 modules. If more is required, other 82C59A controllers can be attached to the inputs of the first one to create a cascading arrangement as shown. In 2 levels, this can thus handle 64 modules.



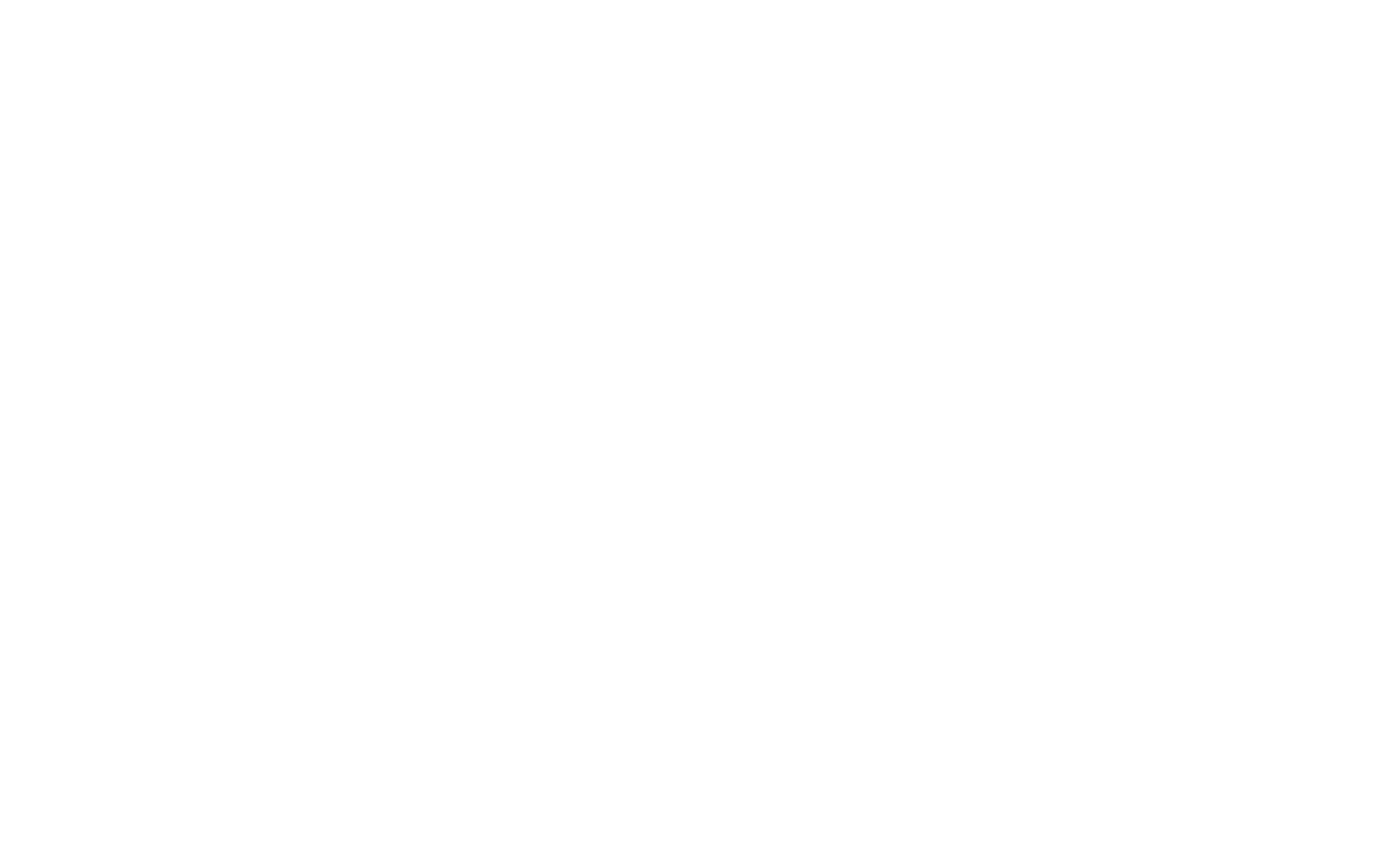
The 82C59A manages interrupts. It accepts interrupt requires from attached modules, determines which one has the highest priority, and then signals the process by raising the INTR line. The processor acknowledges via the INTA line, which prompts the 82C59A to place the appropriate vector information on the data bus. The processor can then process the interrupt and communicate directly with the I/O module to read or write data.

The 82C59A is programmable and the 80386 can determine the priority scheme to be used by setting a control word in the 82C59A. The following modes are possible

* Full Nested – Interrupt requests are ordered in priority from 0 (IR0) to 7 (IR7).
* Rotation – For applications with a number of interrupting devices of equal priority, a device that has just been serviced is given the lowest priority in the group.
* Special Mask – This allows the processor to inhibit interrupts from certain devices.

### Intel 8255A Programmable Peripheral Interface

This is an I/O module used for programmed I/O and interrupt-driven I/O. It is a single-chip, general=purpose I/O module originally designed for the 80386 processor. It has been cloned by other manufacturers and is a widely used peripheral controller chip. Uses include for simple I/O devices for microprocessors and in embedded systems, including microcontroller systems.



* D0 – D7 – Data I/O lines; all data and control information exchanged through these.
* CS (Chip Select Input) – If this line is set to 0, the module is active.
* RD (Read Input) – If this line is set to 0, the data outputs are enabled onto the system data bus.
* WR (Write Input) – If this line is set to 0, data is written onto the module from the system data bus.
* RESET – If this line is set to 1, the module is set to its reset state. All peripheral ports are set to the input mode.
* PA0 – PA7, PB0 – PB7, PC0 – PC7 – Signal lines used as 8-bit I/O ports; they can be connected to peripheral devices.
* A0, A1 – The logical combination of these two input lines decides which internal register of the module can be written or read from.

From the 24 I/O lines, groups C is subdivided into 2 4-bit groups which can be used with A and B I/O ports to carry control and status signals.

============= Skipping the rest of this section. Sir did not teach. =============

## 7.5 Direct Memory Access

### Drawbacks of Programmed and Interrupt-Driven I/O

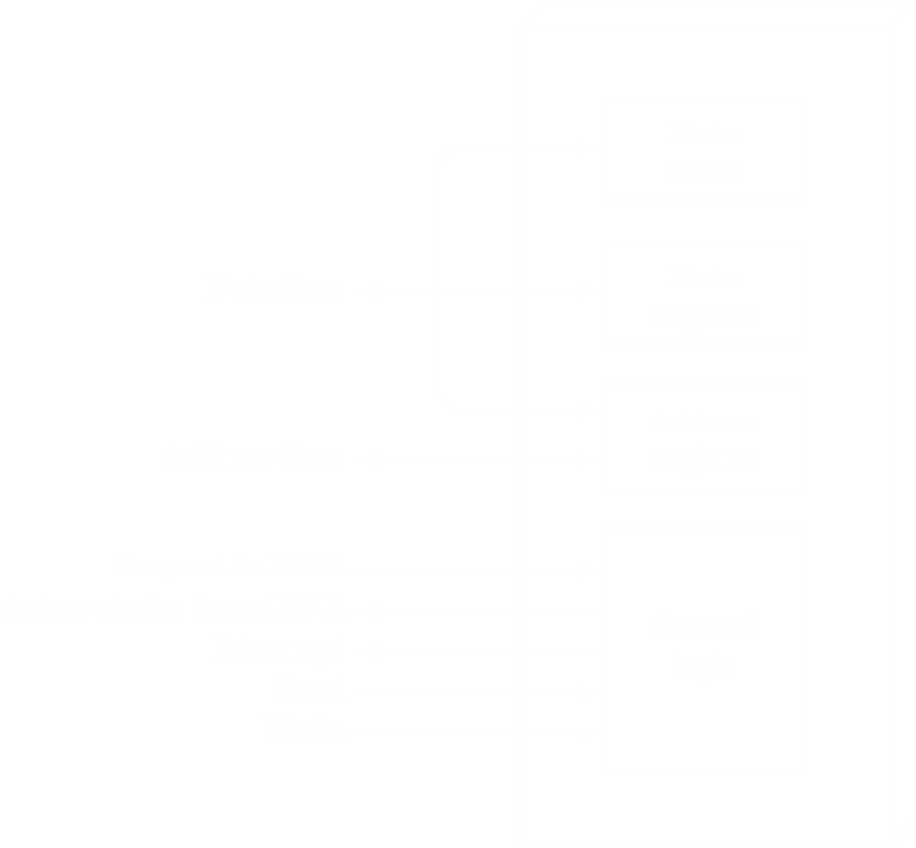
Interrupt-driven I/O is more efficient than simple programmed I/O, but it still needs active intervention of the processor to transfer data between memory and an I/O module, and any data transfers must traverse a path through the processor. Thus, both these forms of I/O have two drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

There is somewhat of a trade-off between the two drawbacks. Consider the transfer of a block of data. In simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a higher rate, at the cost of doing nothing else. In Interrupt-driven I/O, the processor is freed to some extent at the cost of the I/O transfer rate. Nevertheless, both methods negatively affect the processor activity and I/O transfer rates. With large volumes of data, a more efficient technique is direct memory access (DMA).

### DMA Function

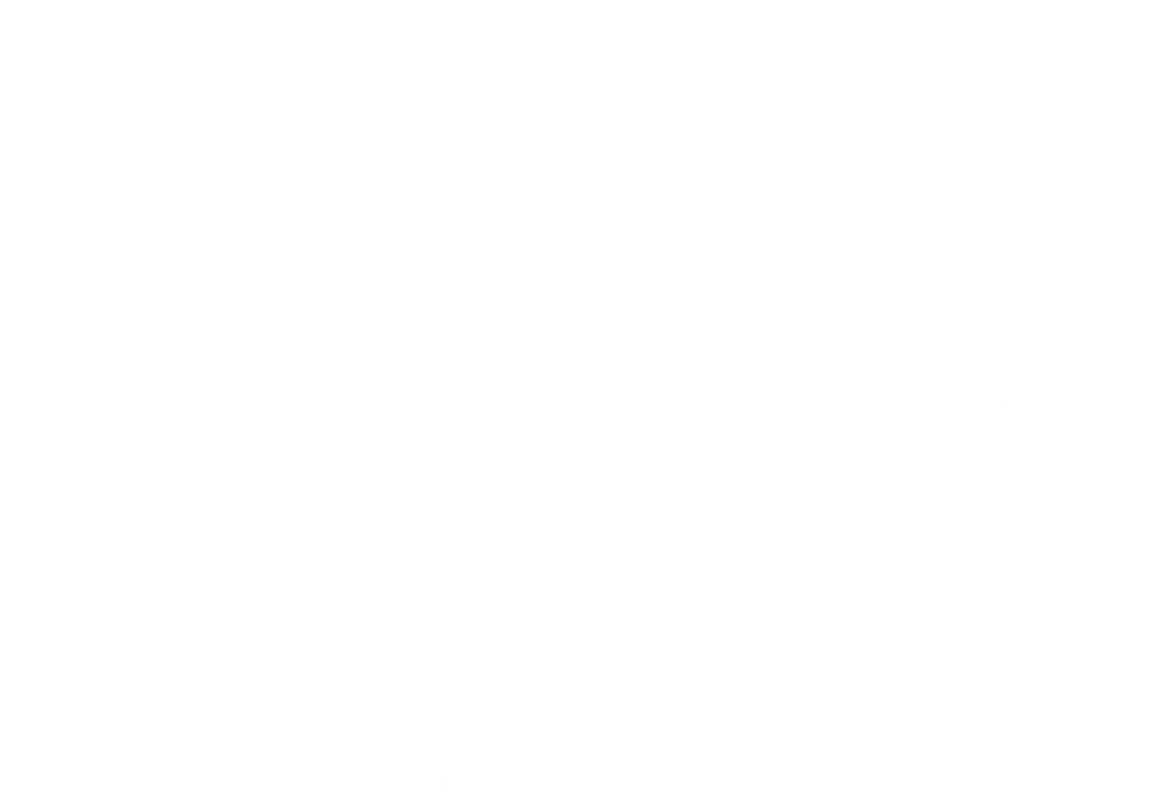
DMA involves an additional module on the system bus. This module is capable of mimicking the processor and taking over control of the system from the processor. It needs to do this to transfer data to and from memory over the system bus. The DMA module must only use the bus when the processor does not need it, or it must force the processor to suspend operations temporarily. The latter technique is more common and is called cycle stealing.



When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending the DMA module the following information:

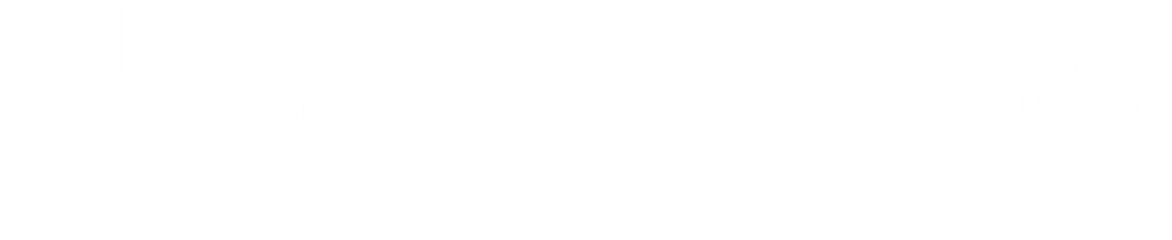
* Whether to read or write, using the read or write control line.
* The address of the I/O device, communicated on the data lines.
* The starting location in memory to read from or write to, communicated on data lines and stored by the DMA in its address register.
* The number of words to be read or written, communicated on the data lines and stored in the data count register.

The process has delegated the I/O operation to the DMA module now and can continue with other work. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor, meaning the processor is only involved at the beginning and the end of the transfer.

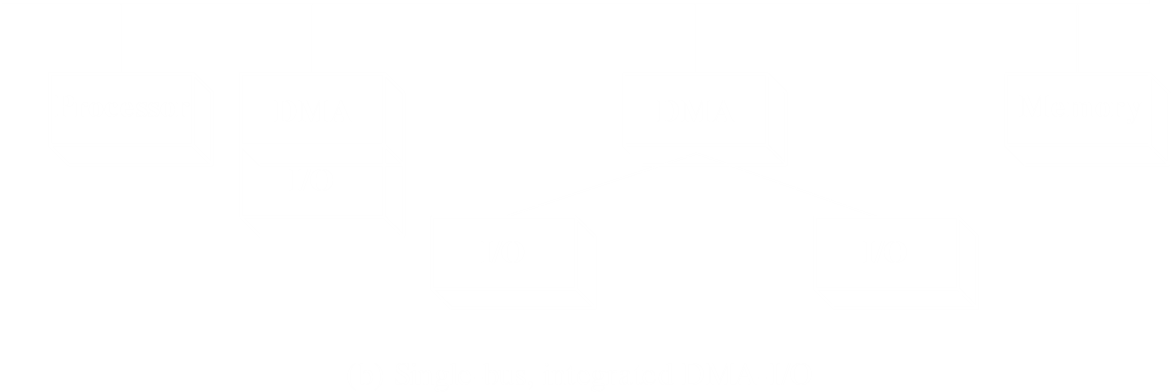


The figure above shows where in the instruction cycle the processor may be suspended. In each case, the processor is suspended just before it needs to use the bus. The DMA module then transfers one word and returns control to the processor. Note that this is not an interrupt, since the processor does not save a context and do something else. Rather, it pauses for one bus cycle. The overall effect is to cause the processor to execute more slowly. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

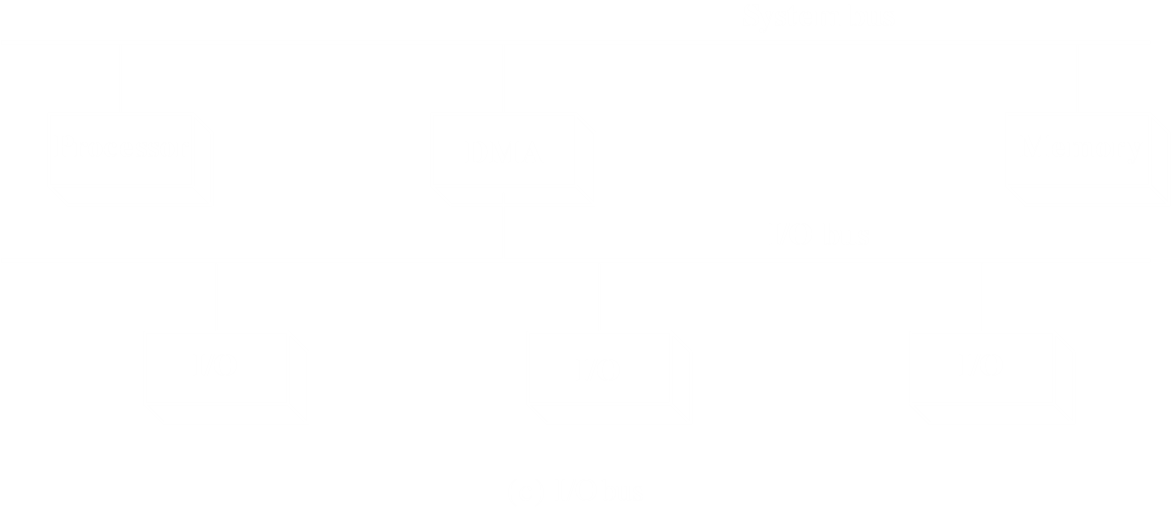
The DMA mechanism can be configured in a few ways. We will look at three here.



In the first configuration, all modules share the same system bus. The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and the I/O module through the DMA module. This configuration is cheap, but inefficient. As with processor-controlled programmed I/O, each word transfer consumes two bus cycles

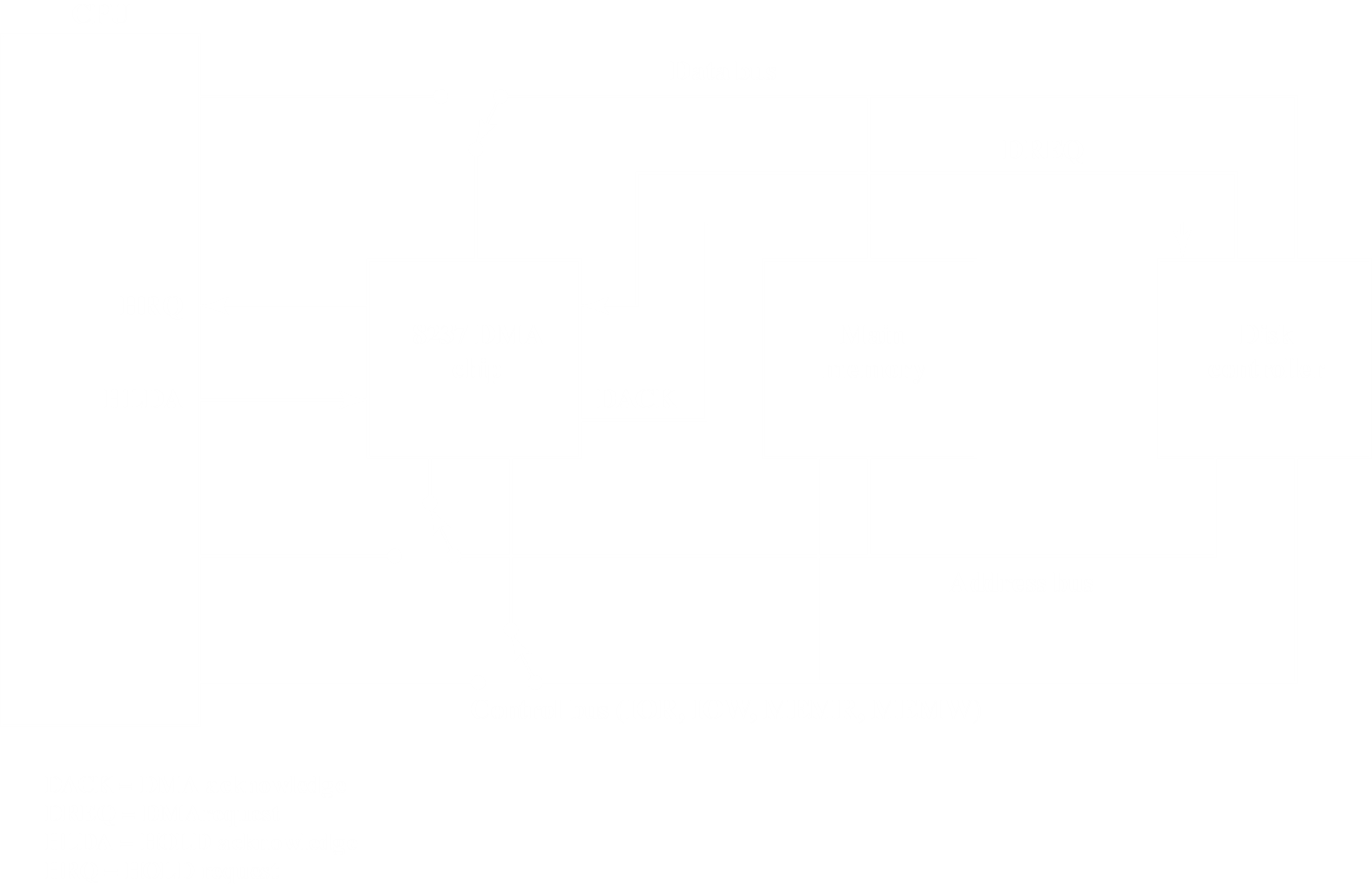


In the second configuration, the DMA and I/O functions are integrated, so the number of bus cycles are reduced significantly. There is a path between the DMA module and one or more I/O modules that does not include the system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.



The concept from the second configuration can be taken further by connecting I/O modules to the DMA module using an I/O bus, as is done in this third configuration. This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration.

In both the second and third configurations, the system bus that the DMA module shares with the processor and memory is used by the DMA module only to exchange data with memory. The exchange of data between the DMA and I/O modules takes place off the system bus.



We can have an overview of how the DMA works with the help of the Intel 8237A DMA Controller shown above. To transfer a block of data from memory to disk:

* The peripheral device (like the disk controller) will request the service of DMA by pulling DREQ (DMA request) high.
* The DMA will put a high on its HRQ (hold request), signalling the CPU through its HOLD pin that it needs to use the buses.
* The CPU finishes the present bus cycle (not necessarily the present instruction) and responds to the DMA request by pulling high on its HDLA (hold acknowledge), thus telling the 8237 DMA that it can go ahead and use the buses to perform its task. HOLD must remain active high as long as DMA is performing its task.
* DMA will activate DACK (DMA acknowledge) which tells the peripheral device that it will start to transfer the data.
* DMA starts to transfer the data from memory to peripheral by putting the address of the first byte of the block on the address bus and activating MEMR, thereby reading the byte from memory into the data bus; it then activates IOW to write it to the peripheral. Then DMA decrements the counter and increments the address pointer and repeats this process until the count reaches zero and the task is finished.
* After the DMA has finished its job it will deactivate HRQ, signalling the CPU that it can regain control over its buses.

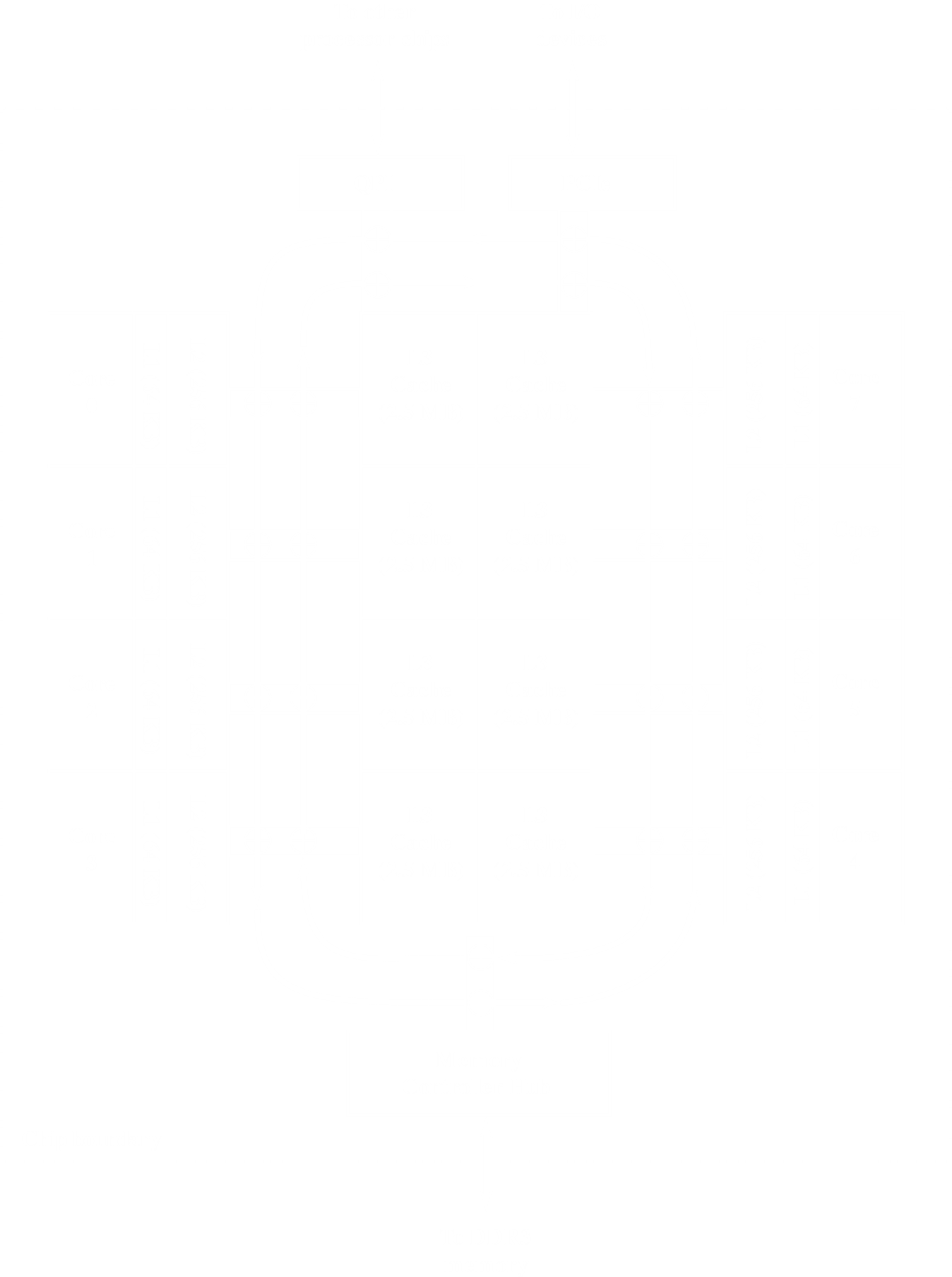
## 7.6 Direct Cache Access

DMA, though effective in enhancing performance of I/O with peripheral devices and network I/O traffic, is unable to scale to meet the dramatic increases in data rates, especially due to the large amount of traffic from Wi-Fi in the gigabit range. We shall now explore how enabling I/O function to have direct access to the cache can enhance performance. We shall only be concerned with the cache that is closest to main memory (last-level cache).

### DMA Using Shared Last-Level Cache

Contemporary multicore systems include both caches dedicated to each core and an additional level of shared cache, either L2 or L3. With the increasing size of last-level cache, DMA has been enhanced so the DMA controller can access the shared cache.

To explain how this works, consider the figure of the Intel Xeon E5-2600/4600 processor below:



If this processor used traditional DMA, an I/O driver running on a core would send an I/O command to the I/O controller (PCIe). This would issue a read request routed to the memory controller hub (MCH), which pulls the data from the DDR3 memory and puts it on the system ring for delivery to the I/O controller. The L3 shared cache is not involved at all. For a write operation, a similar process is followed, but the L3 cache must be updated in this case.

With DCA, the MCH first checks if the data is in the L3 cache. If it is, no main memory access is required. However, the data gets evicted from the cache and must be restored from main memory. Input operations are not affected. Performance is only improved for output operations.

================= Forgot to make notes for Sections 7.7 and 7.8 =================