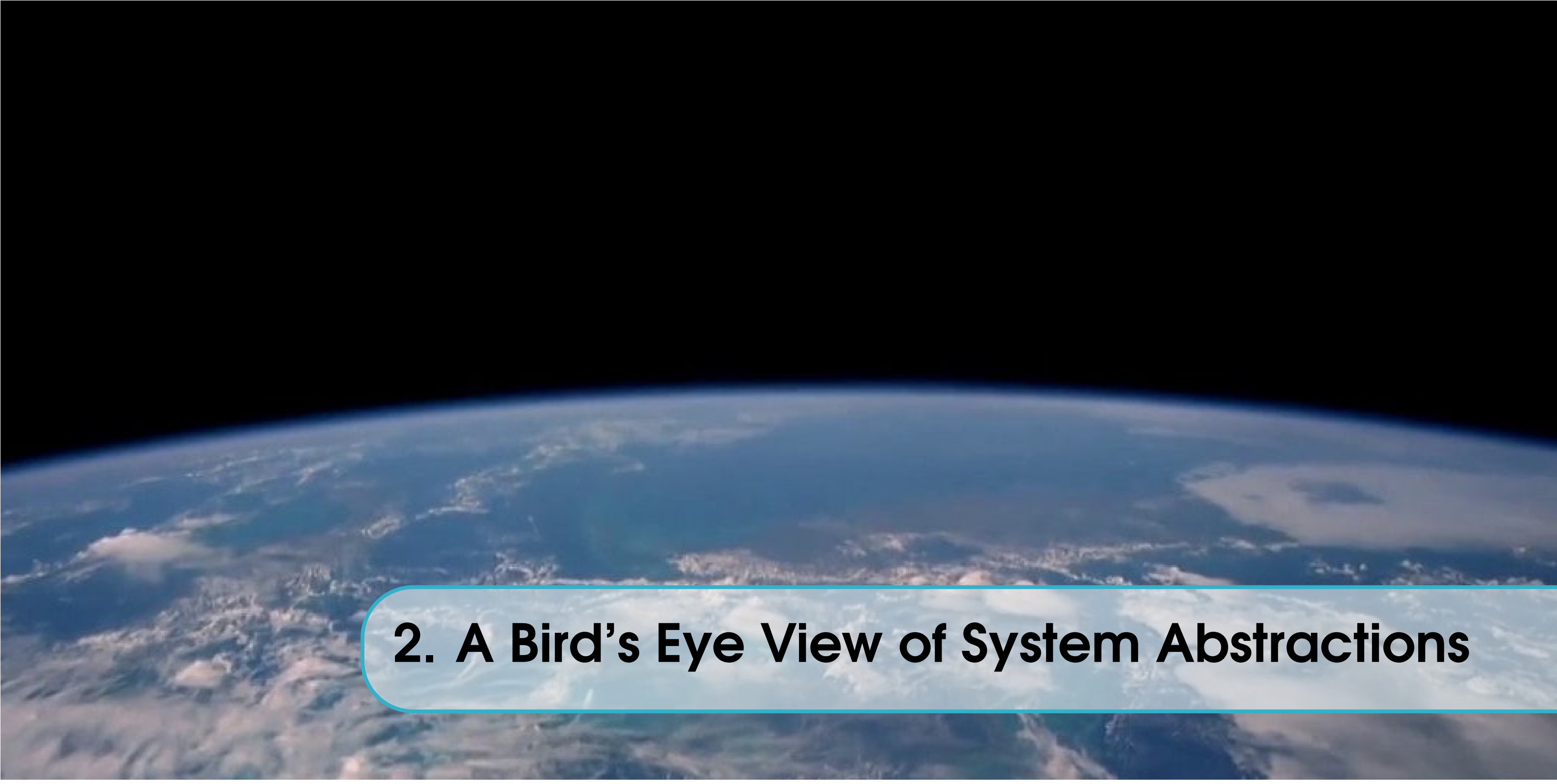

2. A Bird’s Eye View of System Abstractions

In this chapter, we will present a bird-eye’s view of computer organization in order to understand how issues of concurrency and synchronization arise in computing systems, and in order to appreciate the importance of judicious resource allocation.

While doing so, we will also examine two overarching features of modern computer systems in general: (1) the interplay between the subsystems responsible for **data flow** and those responsible for **controlling the system’s operation**, and (2) the use of **layering to hide complexity** in system design and implementation.

2.1 Computer Hardware = Data Path + Controller

If one looks at a bird’s-eye view of the organization of a processor, it will have two parts to it: a *Data Path* and a *Control Path*.

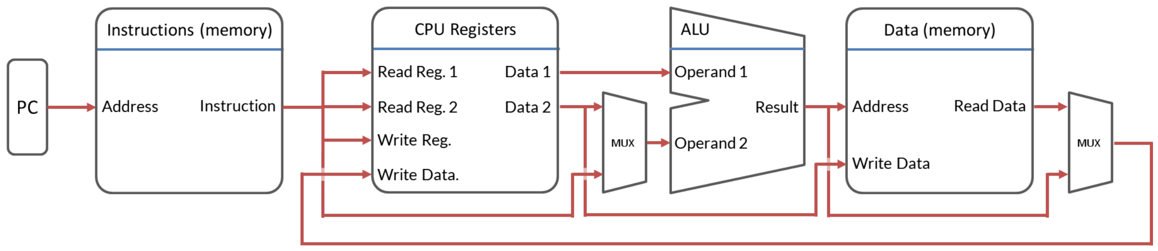

Figure 2.1: A simplified single-cycle processor with data path bus lines (highlighted in red) and data path components.

Figure 2.1 depicts a (overly) simplified representation of a single-cycle processor. The **Data Path** consists of *storage* units such as CPU Registers and Memory (for instructions and data) +

compute elements such as ALU and multiplexers (MUX) + *transfer connections* or buses highlighted in red in the figure. These are all components that deal directly with data by either manipulating them (e.g. by performing an addition in the ALU), by storing them, or by moving them inside and outside a processor.

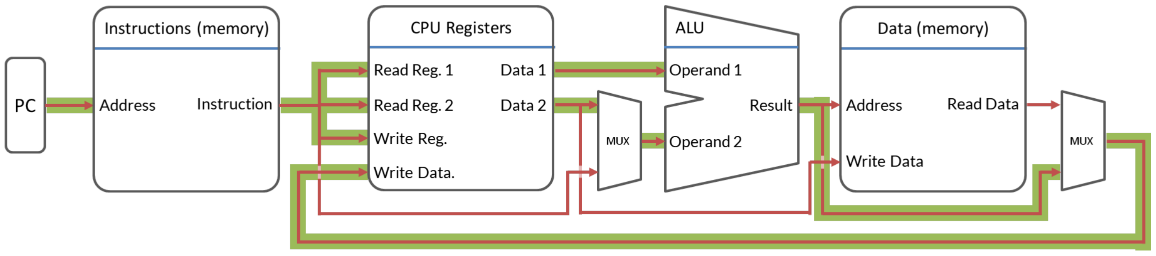

Figure 2.2: Route followed by data when addition between two registers R1 and R2 is selected by an instruction. The route is highlighted in green.

Data processing is the result of "routing" data from registers through the ALU and back to registers. For example, consider Figure 2.2 where the path followed by data to perform the addition between two registers (say R1 and R2) is performed. The result is put in R3. In this case, the data path needs to be controlled in such a way that the data contained in R1 and R2 are routed to the two inputs of the ALU unit as Operand 1 and 2, respectively. Also, we need to control the ALU so that it performs an addition operation. Furthermore, we need to control the data bus in such a way that the result of the ALU (i.e. the sum of R1 and R2) is routed back to the CPU registers block. Finally, we need to control the CPU registers block so that it will store the value of the sum between R1 and R2 in R3.

If you read the previous paragraph one more time, it is easy to realize that performing the operation R3 $\leftarrow$ R1 + R2 involved providing specific **Control Signals** to the various components of the data path. Let us introduce appropriate control signals for each of the decisions taken to perform our operation. First, the MUX before ALU's Operand 2 was instructed to select its first (top) input. Call this control signal `ALUSrc`. Next, the ALU was instructed to perform an addition, out of the various operations it can perform. Once again, this was instructed via a control signal, say `ALUOp`. Next, the result from the ALU is a pure number, and not necessarily a valid memory address. Hence, we need to instruct the data memory to not attempt a memory read with an appropriate control signal, say `MemRead`. At the same time, and for the same reason, no memory write operation should be performed by the data memory block. This is expressed via the control signal `MemWrite`. Next, what goes back to the CPU registers block should be the result of the ALU operation, not the result of a memory read. This means that the MUX on the right-hand side of the figure should be instructed to select in output its second (bottom) input via the control signal `MemToReg`. Finally, the CPU registers block should be instructed to store the forwarded value into the write register (in this case R3) originally selected by the considered instruction. For this purpose, a control signal is used, say `RegWrite`.

Figure 2.3 depicts in blue the control signals described above and directed to the various blocks of the data path to enforce the desired behavior. Clearly, these signals need to be generated somehow. The logic responsible for the generation of these signals goes under the name of **Controller**, once again reported in the figure. The controller uses a portion of the fetched instruction (the *opcode*) to

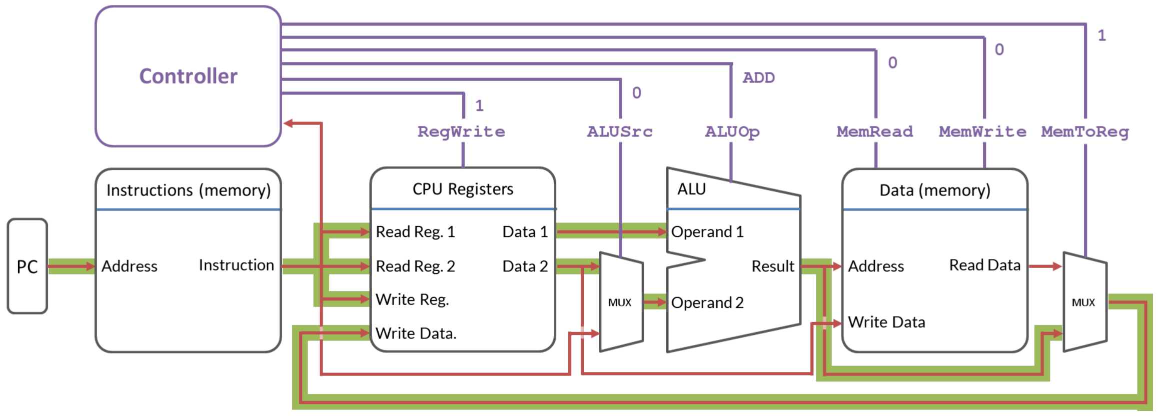

Figure 2.3: Route followed by data when addition between two registers R1 and R2 is selected by an instruction. The route is highlighted in green, and the control signals used to enforce this route are depicted above each controlled component.

generate such signals.

At any point in time, the state of a data path consists of the state of all storage in the data path (that includes registers, status bits, etc.) The state of the data path changes synchronously every clock cycle. The change in the state of the data path is controlled through a (potentially large) number of Control Signals.

The Controller is in charge of specifying the values of these control signals for every clock cycle, and in this way it is able to *control* data path state transitions. In the figures used so far, we have considered a single-cycle CPU where each machine instruction is executed within a single clock cycle. In modern CPUs (multi-cycle and pipelined CPUs), in general, a machine instruction is executed over a sequence of clock cycles, whereby in each clock cycle, the data path is "instructed" to perform part of the instruction (or what is often termed a micro operation) through the generation of appropriate control signals. A micro operation is typically a simple "register transfer" operation that results in data being moved between registers and possibly through functional units to do ALU operations (e.g., moving data between two registers, or storing the output of an ALU into a register, etc.). In the example we used for adding R1 and R2, "routing" the contents of R1 and R2 to the ALU are examples of micro operations, and so is the routing of the ALU output back into R3.

For simple CPUs, the Controller can be thought as a simple lookup table that translates opcodes into required control signals. For instance, looking at Figure 2.3, if the opcode for register-to-register addition is encountered, the following control signals are produced: `ALUSrc = 0, ALUOp = ADD, MemRead = 0, MemWrite = 0, MemToReg = 1, RegWrite = 1`. In more complex CPUs, the Controller is a *Finite State Machine* (FSM) that has different sequences of micro operations for different machine instructions. For example, the sequence of values for the control points necessary to perform an addition (described above) can be thought of as the sequence of micro operations for an addition. Similarly, one can think of a sequence of micro operations for fetching a word from memory into a given register, or for incrementing or decrementing a register, etc. Given a machine instruction (or even part of a machine instruction), the controller selects the right sequence of micro operations and executes the instruction by applying the sequence (of micro operations) to the data path.

The relationship between the data flow plane (the data path) and the control flow plane (the controller) is through a continuous interplay whereby the two planes cause constant changes in each other. The controller "interprets" the state of the data path and then "controls" the movement of data in the data path for the next clock cycle. The new state of the data path is interpreted and a new set of controls are applied to move data for the following cycle, etc.

Box 2.1.1 Data versus Control Planes

A recurring theme in the design of computing systems (whether such systems are hardware systems, such as CPUs, or software systems, such as protocols for networking or database systems) is that the architecture of such systems is best described along two dimensions or "planes": the *data flow plane* and the *control flow plane*.

The data flow plane of system architecture concerns itself with the interconnectivity of various elements of the systems. Data (or content) can move between two elements in a system if a "path" exists between these elements. In a hardware setting, such a path will manifest itself as wires (e.g., the bus connecting a register to the ALU, or the network link connecting a router in the network to another). In a software setting, such a path will manifest itself as data communicated between programs (e.g., one program consuming some other program's output, giving rise to relationships between software entities known as producer/consumer, reader/writer, publisher/subscriber, and so on.)

The control flow plane of system architecture concerns itself with how the interconnections that exist between entities are used to actually result in a data transfer. It concerns itself with issues of timing, coordination, synchronization, etc. In a hardware setting, this "control" will manifest itself through the timing and synchronization of various switching operations (e.g., the use of multiplexers and de-multiplexers to affect various patterns of communication or computation in a CPU). In a software setting, such control will manifest itself through the timing and synchronization (a.k.a. scheduling) of data exchanges between programs or processes (e.g., the use of special protocols to regulate access to shared data).

To make an analogy, consider how one would describe the "architecture" of a railroad (or subway) system. Clearly, one needs to describe which stations there are (e.g., the MBTA in Boston has stations including Government Center, Park Street, BU East and BU Central) and how these stations are interconnected (e.g., there is a direct path between BU East and BU Central, between BU Central and BU East, between Government Center and Park Street, etc.) This description (of stations and their direct adjacencies) is the MBTA "data plane." It is on that plane that "data" (i.e. trains) move. This is precisely what the MBTA "map" would tell us. Clearly, however, this map is not enough to describe the operation of the subway system. To do so requires a description of how the train movement is coordinated, timed, and synchronized. This would constitute the "control plane" of the MBTA.

In this chapter we will examine an instance of these data and control planes in computing system architectures. While we will focus on computer organization as an example, it should be noted that this same guiding principle – of two "planes" of data flow and control flow – governs how we "architect" systems in general, whether these systems are CPUs, database systems, operating systems, networking systems, etc. (see Box 2.1.2).

Box 2.1.2 Example of Data and Control Planes in the Internet

As a second example of how computing systems often constitute two planes–one for moving data between various components and the other for controlling such movement–we consider data communication from a source to a destination on the Internet. At a high enough level, one may describe the Internet as a graph, where nodes of the graph represent processing elements such as web servers, client machines, and routers, while edges of the graph represent physical communication links such as cables, wireless links, etc. Now consider a specific source of data (e.g., a web server) and a specific destination for that data (e.g., a client machine requesting (say) a web page from the server). These two entities (the source and destination) would correspond to two specific nodes in the "gigantic" graph that underlies the Internet. Clearly, one would not expect these two nodes to be directly connected, but rather that data would flow from the source to the destination by going through a number of intermediate nodes (routers) or "hops." One such set of direct links (identified by a particular set of intermediate nodes) is called an Internet path. Clearly, one would expect a huge number of paths to exist between a source and a destination on the Internet. Which one of these paths will be selected for the delivery of data from the source to the destination? This is precisely what routing protocols (which are implemented at multiple "layers" of the Internet) are designed to provide.

In the above example, the set of nodes and links that constitute the Internet graph could be construed as the "Data Plane" of the Internet, whereas the set of protocols that are used for routing on that data plane could be construed as (one of) the "Control Planes" of the Internet.

It is important to note that associated with the same data plane, there might be a number of control planes, each of them concerned with the "control" of a specific aspect of the data plane. In the above two examples (namely, processor organization and data communication over the Internet), we have seen instances of control planes used primarily for "routing"–whether of data between registers, storage, and functional units in a CPU, or of content from a sender to a receiver on the Internet.

To exemplify the need for multiple control planes, we note that controlling the "routing" from a source to a destination says nothing about the rate with which data is to be communicated. This is an important aspect since sending data at rates higher than what the data plane is capable of handling would result in loss of data, and alternately, sending data at rates much lower than what the data plane is capable of handling would result in an inefficient use of underlying resources. To regulate the data communication rates, a separate "transmission control" plane is used. To that end, a widely used protocol (called TCP) is used to implement this transmission rate control plane (among other functionalities, such as reliable and in-order delivery of data packets).

2.2 Program Execution

The abstraction of a "program" is presented to the programmer by enabling the storage of a set of machine instructions in memory. The execution of an instruction proceeds in two phases:

1. **Fetch** the instruction stored in memory at the location determined by the PC;
2. **Execute** the fetched instruction by dispatching the controller to the appropriate sequence of micro-operations. Also, adjust the PC to either point to the next instruction sequentially, or to another instruction (if the current instruction is a Branch/Jump/etc.)

The above two phases are typically called the Fetch and Execute cycles. As such, the entire lifetime of a program can be abstracted with the diagram depicted in Figure 2.4.

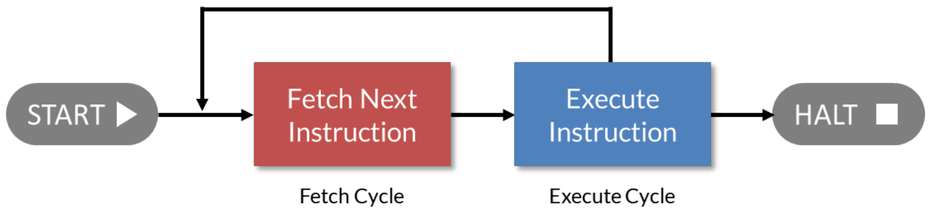

Figure 2.4: Basic instruction cycle.

2.3 Interrupts

The operation of a computing system depicted according to Figure 2.4 abstracts away an important aspect: computation can affected by events that are external to a program's instruction sequence. In other words, it can (and should) happen that every now and then the basic instruction cycle is *interrupted* to handle a number of "events". For this purpose, processors include logic to handle **Interrupts**.

In a nutshell, an interrupt is simply an input into the CPU's circuitry that, if asserted, drives the processor status into a special *interrupt handling* state. How and why an interrupt line can be asserted varies according to the type of the considered interrupt. Generally speaking interrupts are needed to handle events that arise when we interface a computer with other devices (e.g., disk, NIC, I/O, etc.), when the same CPU is used to execute multiple interleaved programs, or when we want to allow communication between various entities of a computing system (e.g., different programs).

Modern processors support a wide variety of classes of interrupts, summarized as follows:

1. **Exceptions:** synchronous interrupts that are triggered by an unexpected event within the instruction cycle. For instance, an exception in the instruction fetch phase is generated if the

address in the PC is invalid. Similarly, an exception can be generated by the ALU if a division by zero was attempted by the program.

2. **Software Interrupts:** interrupts generated synchronously by a program via a special instruction. These interrupts are commonly used to invoke routines outside the space of the executing program, for instance to implement calls to routines within the Operating System (system calls). In multi-core processors, software interrupts can be used by core A to deliver messages or to invoke a routine on a different core B.
3. **Hardware Interrupts:** interrupts generated by circuitry that is external to the processor itself. Depending on the importance of the interrupt, these can be further classified as:
   - **Maskable Interrupts:** interrupts associated to typical hardware I/O devices. For instance, the arrival of a new packet at a network interface triggers an interrupt. Another example are interrupts triggered by hardware timers to enforce periodic execution of operating system routines. Similarly the press of a key by the user in a keyboard may trigger an interrupt to handle the keystroke. All these interrupts are said to be "maskable" because they can be safely ignored by the system if needed.
   - **Non-maskable Interrupts:** interrupts that are triggered by internal circuitry responsible for checking and managing power, temperature, or logic correctness of a machine. These interrupts are generally triggered if something unusual that requires immediate attention is detected with the physical state of the machine. For instance, a core temperature which exceed the safety threshold could trigger a non-maskable interrupt. Similarly, if a corruption of the processor's logic (or memory) is detected, a non-maskable interrupt may be raised. Given how crucial is the timely handling of these interrupts, they cannot be ignored, or "masked".

2.3.1 Interrupts vs. Busy-waiting

In order to appreciate the role that interrupts play, one should consider the alternative. The alternative to interrupts is "polling" (also known as "busy waiting"). We can explain this with an example. Consider a program that needs to interact with an I/O device via I/O operations. Each I/O operation has two phases:

1. Prepare for the I/O and issue the I/O command (e.g., move data and command parameters to device memory-mapped memory and signal device);
2. Once the I/O is done then process results and cleanup (e.g., move data around, do another I/O, etc.).

To support the above without interrupts, we would need to do what is called "busy waiting." Basically, after the first phase we stay in a tight (idle) loop checking to see when (and if) the I/O completed in order to do the second phase. This process is shown in the diagram reported in Figure 2.5.

Busy waiting is very inefficient because it results in the system going at the speed of the slowest entity. Using interrupts enables us to avoid busy waiting by requesting that the I/O device (or whatever else the program is waiting for – possibly including results from another program on the same processor) would send a "signal" (the interrupt) when it needs the CPU's attention.

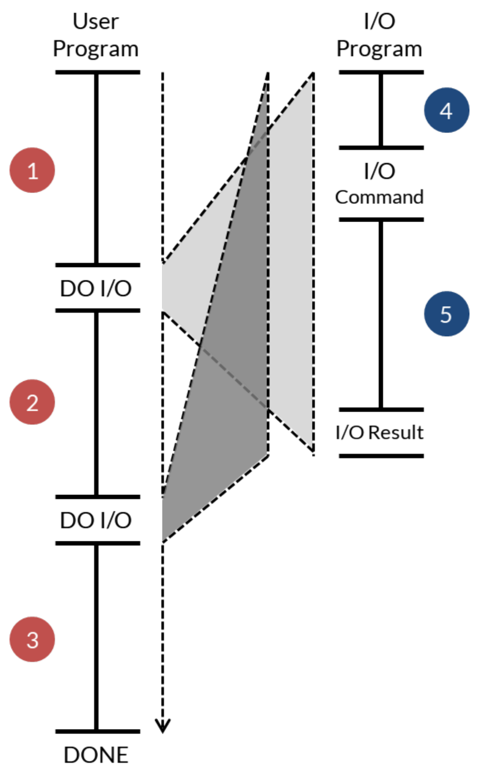

Figure 2.5: I/O operations performed synchronously via busy-waiting in a program flow.

To support interrupts, we structure I/O functions (or whatever else) to consist of two pieces of code corresponding to the two phases above: a phase to "request service", i.e. preparing and issuing an I/O command; and a phase to "complete service", i.e. receiving and interpreting the result of the previously issued I/O command. The latter of these two phases involves "handling the interrupt" that signals the completion of the request – hence the terms "Interrupt Handler" or "Interrupt Service Routine" (ISR), which are used to describe the processing necessary to respond to an interrupt. These two phases are necessary for supporting interrupts. Figure 2.6 shows the various phases of an interrupt-enabled program.

It follows that in order to process an interrupt, we need to "shift" the attention of the CPU from whatever it is doing to the task of "servicing" the interrupt, i.e. the interrupt handler. This introduces the need for interleaving program executions (i.e. switching between programs). To do this safely, we introduce a new (third) phase in the instruction cycle (after the execution phase) to check for interrupts. The updated instruction cycle is depicted in Figure 2.7.

If there is an interrupt, then we have to do some housekeeping to prepare for servicing it. This housekeeping is done by both hardware and software, as we will discuss later. Before we review what the responsibilities of hardware and software are with respect to handling interrupts, we need to address one issue: since the timing of an interrupt cannot be predicted, it is possible that an interrupt could occur while we are servicing a previous interrupt! Thus, it is imperative to figure out what needs to be done to support multiple interrupts.

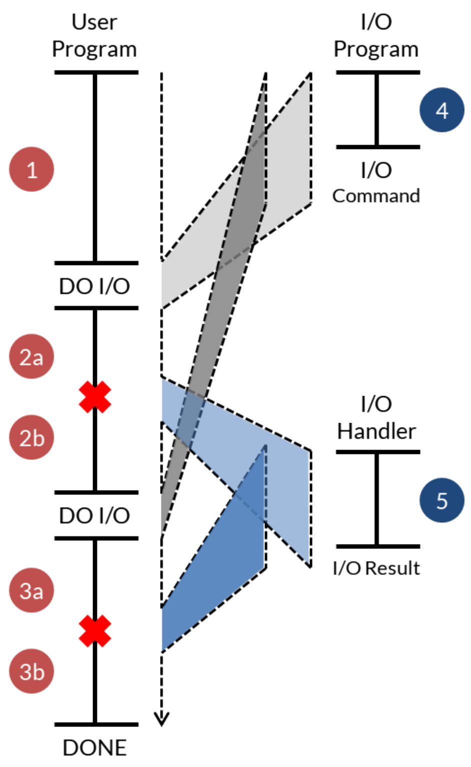

Figure 2.6: I/O operations performed asynchronously via interrupt handling in a program flow.

2.3.2 Supporting Multiple Interrupts

To support multiple interrupts, we can simply force the interrupts to be “sequenced” by disabling the processing of interrupts until the previous one has been handled. An example of this approach is provided in Figure 2.8, inset (a). Assume that while handling interrupt X, interrupt Y arrives. In this case, handling of interrupt Y is deferred until completion of the interrupt handler for X. This approach, albeit easier to implement, is not ideal for many reasons, including the possibility of “missing” interrupts, and because such an approach would result in “priority inversion”. Priority inversion refers to the condition in which handling of an important event is deferred due to a less important or unimportant even being currently handled. As we will extensively discuss later in this course, timely processing of I/O events is key in computing systems that are in charge of safety-critical processes.

An alternative solution is depicted in Figure 2.8, inset (b). In this case, some interrupts are allowed to interrupt the service of others through the use of priorities. It is important to note that we still need to disable interrupts (for a minimal amount of time) to enable the appropriate management of the “stack abstraction” for proper nesting of interrupt processing.

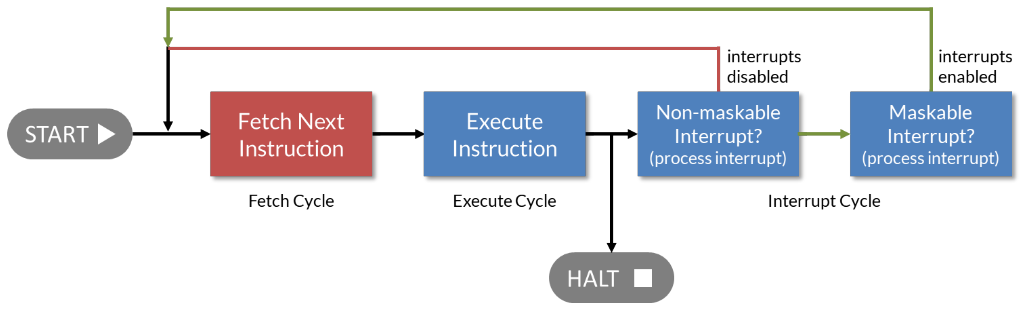

Figure 2.7: Instruction cycle with interrupt handling.

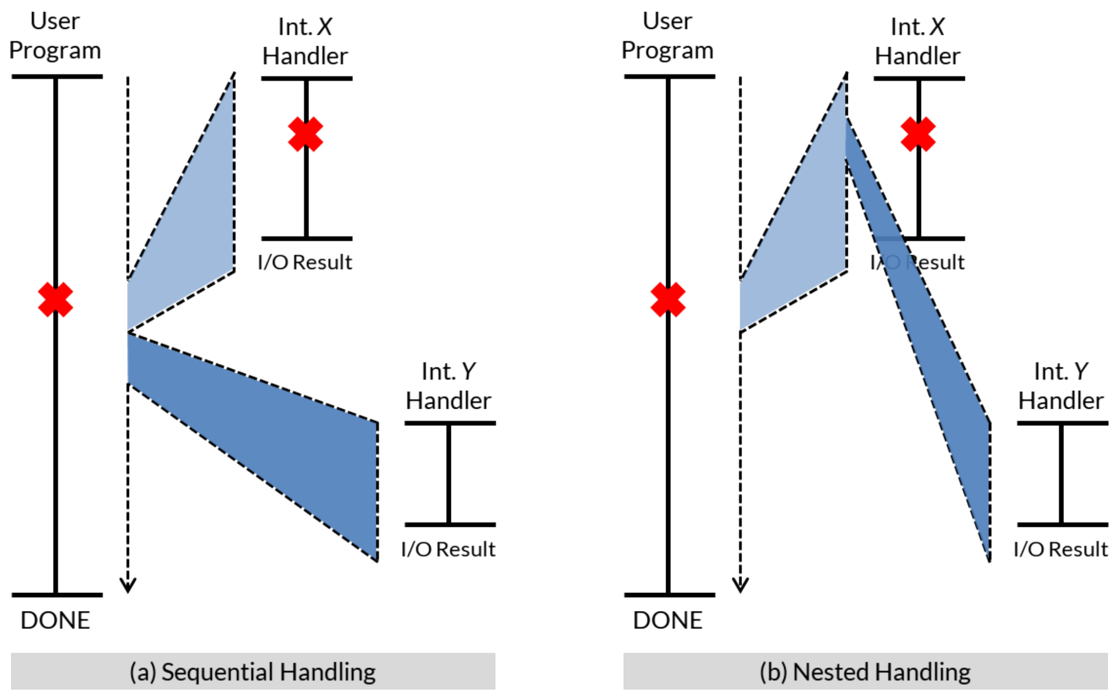

Figure 2.8: Two strategies for interrupt handling: sequential handling (a), and nested handling (b).

2.3.3 Interrupt Processing

Interrupt processing is first done through hardware support. This includes:

1. Complete current instruction and check interrupt line(s);
2. Storage of the processor's main control and status registers, including the PC;
3. Mask/disable all the other maskable interrupts.
4. Deposit of the Interrupt Service Routine (ISR) address into the PC, hence launching the execution of the corresponding interrupt handler;

Once this is done, the processor resumes its normal execution of "software." Obviously the instructions that will be executed at this point are from the ISR, since the PC was loaded with the address of the ISR as described in Step 4. An overview of the entire process is depicted in Figure 2.9.

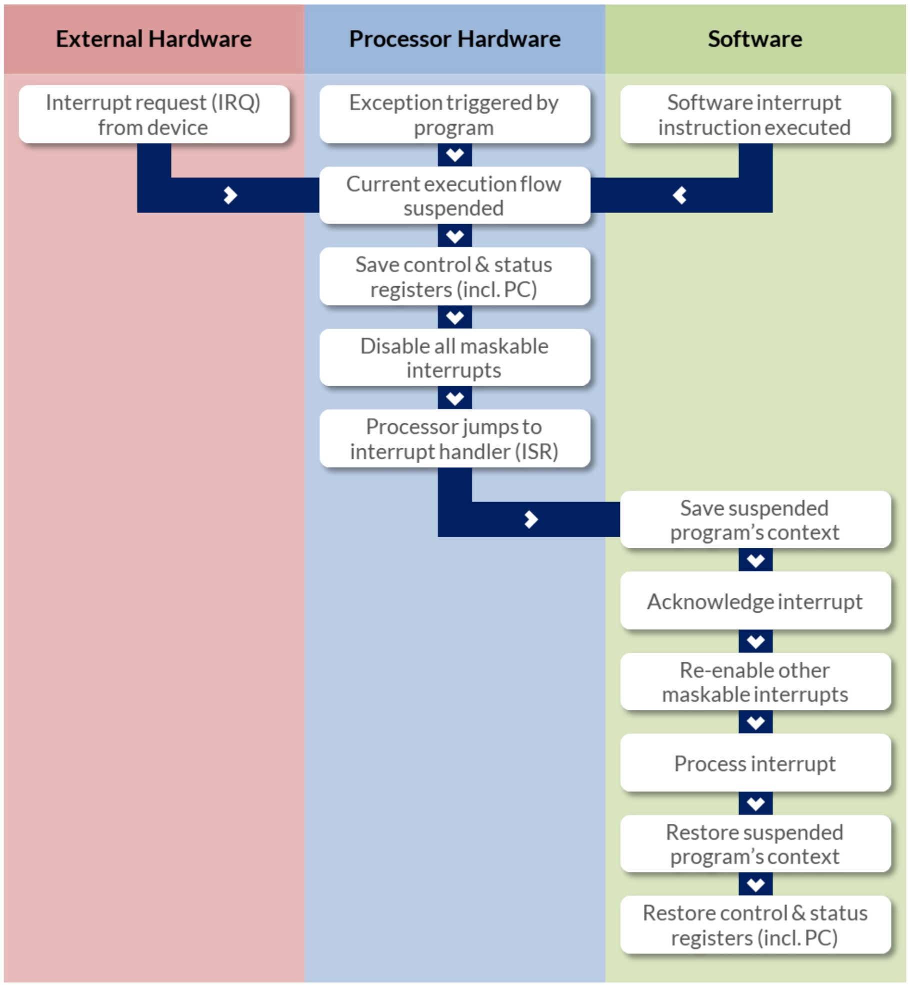
External Hardware
Processor Hardware
Software
Interrupt request (IRQ) from device
Exception triggered by program
Software interrupt instruction executed
Current execution flow suspended
Save control & status registers (incl. PC)
Disable all maskable interrupts
Processor jumps to interrupt handler (ISR)
Save suspended program's context
Acknowledge interrupt
Re-enable other maskable interrupts
Process interrupt
Restore suspended program's context
Restore control & status registers (incl. PC)

Figure 2.9: Interplay between external hardware, processor hardware and software in an interrupt receipt/handling flow.

As shown in the figure, the first thing that the ISR "software" does is to save the state of the previously executing program so that it will be possible to restore that state once the interrupt processing is done. Once the state of the previously executing program is "saved," then it is safe to re-enable interrupts. At this point a higher-priority interrupt can actually "interrupt" the execution of the ISR's execution. This is shown in the diagram in Figure 2.10, in which a network ISR interrupts a printer ISR. When a lower priority interrupt occurs at a time when a higher priority interrupt is being serviced (or when interrupts are disabled), then the servicing of the lower priority interrupt occurs immediately after the conclusion of the ISR for the higher priority interrupt (or when interrupts are enabled). This is shown in Figure 2.10 in which the disk ISR is executed when the network ISR concludes.

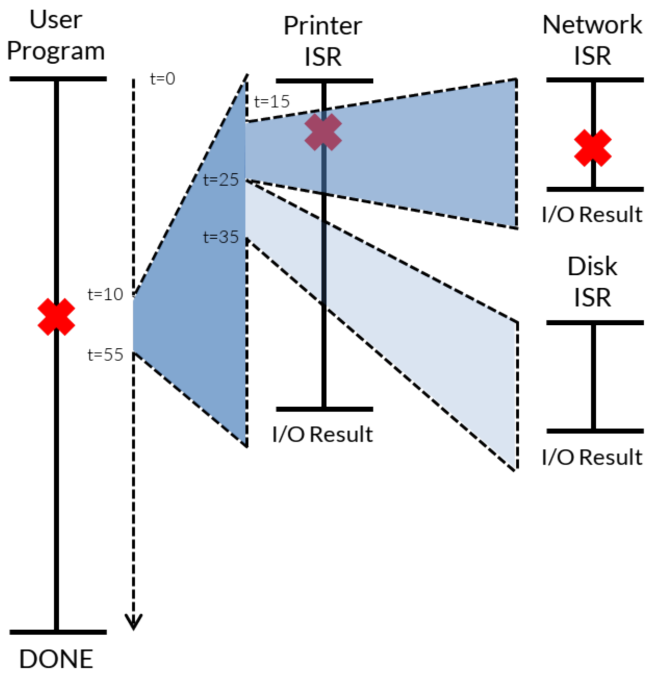

Figure 2.10: Nested execution of various ISR's (for a printer, a network interface and a disk) with different priority levels.

Box 2.3.1 Hiding System Complexity through Layering

Another recurring architectural feature of computing systems is the use of layering to hide complexity.

So far, in our discussion of computer organization for example, we saw how the CPU architecture can be described by a data flow plane (the data path) and a control flow plane (the controller). To the programmer, however, the data path and controller are too complex (and too low level) to think about (and to manage).

Think about what it would take to write a simple program to multiply (or better divide) two floating point numbers, if you had to manage/control every multiplexer, de-multiplexer, accumulator, etc. Not only would this be too complex to manage, debug, and think about, it would require programmers to master quite a bit of hardware know-how! To that end, we *hide* the complexity of the CPU data path and controller by presenting them to the programmer as an "Instruction Set Architecture" (ISA). The ISA implements a set of basic operations (e.g., add, subtract, load, store, . . .) by providing the necessary programs (or sequences of controls) to implement these operations. The ISA layer hides the intricate designs of the CPU (buses, multiplexers, ALU controls, memory controls, . . .) from the programmer, who now thinks in terms of registers and memory and assembly language instructions.

Still, the hiding of CPU architectural features under the ISA layer leaves a lot to be desired! The ISA layer itself is too complex and too low level for most programming tasks. To that end, we hide the complexities of the ISA layer from the programmer by developing compilers and interpreters that allow programmers to use higher-level programming languages (e.g., C or C++) which present the computer to the programmer as a virtual machine for that language.

This process of hiding complexity through layering is very pervasive in computing system design! However, we don't stop with a high-level programming language (like C, C++, or Java) we go on and introduce libraries (a.k.a. APIs) to hide various functionalities (e.g., I/O).

The use of layering to hide complexity is not unique to programming systems, but it is common in other systems as well, including networking with its physical, IP, TCP, and application layers (see Box 2.3.2).

Box 2.3.2 The Layering Principle in Network Design

An excellent illustration of the Layering Principle is how it is used in networking systems. Networking systems must deal with (and solve) many problems that range from fairly low level problems (e.g., how to send bits between computers that have a direct physical connection–this is called "signaling") to much more elaborate problems (e.g., how to ensure that a message is transmitted reliably, or how to ensure that the content of a message is secure, etc.)

To require developers of every networked application to solve all of these problems is not practical. It would be akin to asking a programmer to implement an inventory control application in machine language! Rather, programmers of networked applications think of the network at a much higher level of abstraction.

For example, a programmer writing an application may think of the network as allowing two HTTPS agents (one on a client computer and one a server computer) to communicate directly. All that such a programmer needs to know would be the primitives of the HTTPS "layer." In reality, , the HTTPS layer is nothing but a piece of software that uses a lower level protocol: the HTTP protocol, which is conceptually at a layer below the HTTPS protocol.

The HTTP layer on the client/server sees the network as allowing two agents to communicate directly using HTTP primitives such as "get" and "put." In reality, the HTTP protocol uses functionality at a lower level, namely TCP.

TCP provides the illusion of (among other things) the reliable delivery of messages between two (TCP) agents. TCP itself relies on yet another software layer–the IP layer.

The IP layer provides the functionality necessary to route one packet from one (IP) network address to another. To do so, the IP layer is built on top of yet another layer–the Link layer.

The Link layer provides functionality that enables a packet to be reliably transmitted between two computers with a direct physical (wired or wireless) connection. Even the link layer uses lower level functionality–the Physical layer, where bits are actually put on the wire!

The diagram below shows this layering "in action." Notice that in the example below only the client and server computers have agents at the Application, HTTPS, HTTP, and TCP layers. This is only to simplify the example. In particular, if the client is communicating with the server via a proxy, then that proxy (not shown in the diagram below) would have to include TCP and HTTP layer functionality.

The take-home message from this example is that, to a piece of software, what constitutes a network depends on the layer where that piece of software resides.

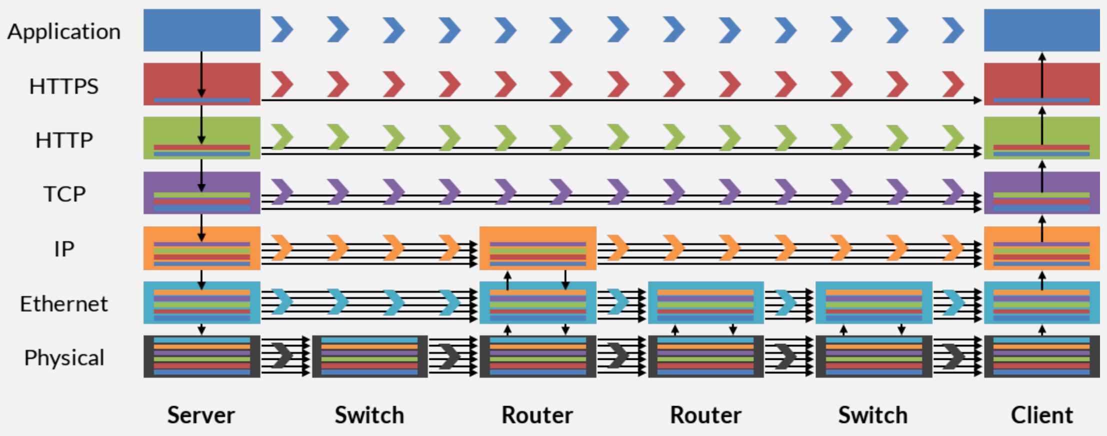

Figure 2.11: Illustrative example of layering in a network setting, with server-to-client message exchange.

Because network protocol layers are "stacked" atop each other, they are typically referred to as the "network stack." A popular network stack model is the TCP/IP stack model, illustrated in Figure 2.12 which is the de-facto standard of the Internet today. At the top of this stack there are a number of application-level protocols (e.g. telnet, FTP, SMTP, and so on). These are *transported* by either TCP or UDP, which in turn use IP at the network level. IP can then be carried over a number of data-link level protocols.

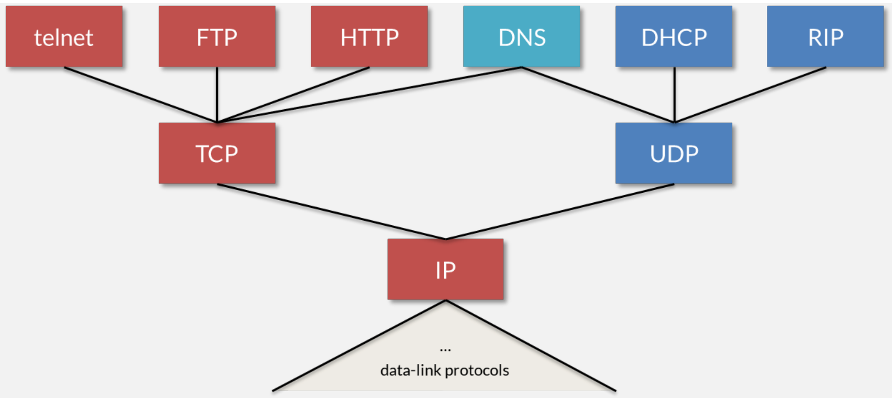

Figure 2.12: The TCP/IP networking stack.

2.4 Operating Systems in a Nutshell

If we have to give a description of the Operating System (OS) in the minimum number of words, then the following is perhaps the most concise and yet accurate description: an OS is a collection of service calls and interrupt handlers that define layers of abstraction for programmers. Indeed, for the purposes of this course, it will suffice to view the operating system as a collection of programs or utilities that coordinate/manage the use of system resources (e.g., CPUs, memory, and devices), among various programs and users.

In particular, an operating system provides two types of programs (or utilities). The first type are programs that are called "synchronously" from within a user program via software interrupts to achieve some desired function (e.g., service call to open a file, or send a packet to an Ethernet, etc.). These also go under the name of "system calls". The second type are programs (namely ISR's) that are called "asynchronously" through the occurrence of events – e.g., process a packet received on the network interface, process a keystroke, respond to completion of a disk transfer, and so on.

A keyword in the above view of an operating system is the word **coordinate**. This word implies a multiplicity of entities (programs, users, etc.) with a need to communicate and share information and resources (e.g., memory, devices, etc.). The correct and efficient management of these issues related to concurrency and resource management is at the heart of any computing system (e.g., database system, Internet application, etc.) and they are at the heart of the materials we intend to cover in this course.