5.11 A given design with cache implemented has a main memory access cost of 20 cycles on a miss and two cycles on a hit. The same design without the cache has a main memory access cost of 16 cycles. Calculate the minimum hit rate of the cache to make the cache implementation worthwhile.

5.12 Design your own 8K × 32 PSRAM using an 8K × 32 DRAM, by designing a refresh controller. The refresh controller should guarantee refresh of each word every 15.625 microseconds. Because the PSRAM may be busy refreshing itself when a read or write access request occurs (i.e., the enable input is set), it should have an output signal ack indicating that an access request has been completed. Make use of a timer. Design the system down to complete structure. Indicate at what frequency your clock must operate.

CHAPTER 6: *Interfacing*

6.1 Introduction

As stated in the Chapter 5, we use processors to implement processing, memory to implement storage, and buses to implement communication. The earlier chapters described processors and memory. This chapter describes implementing communication with buses, known as interfacing. Communication is the transfer of data among processors and memories. For example, a general-purpose processor reading or writing a memory is a common form of communication. A general-purpose processor reading or writing a peripheral's register is another common form.

We begin by defining some basic communication concepts. We then introduce several issues relating to the common task of interfacing to a general-purpose processor: addressing, interrupts, and direct memory access. We also describe several schemes for arbitrating among multiple processors attempting to access a single bus or memory simultaneously. We show that many systems may include several hierarchically organized buses. We then discuss some more advanced communication principles and survey several common serial, parallel, and wireless communication protocols.

6.2 Communication Basics

Basic Terminology

We begin by introducing a very basic communication example between a processor and a memory, shown in Figure 6.1. Figure 6.1(a) shows the bus structure, or the wires connecting the processor and the memory. A line *rd'/wr* indicates whether the processor is reading or writing. An *enable* line is used by the processor to carry out the read or write. Twelve address lines *addr* indicate the memory address that the processor wishes to read or write. Eight data lines *data* are set by the processor when writing or set by the memory when the processor is reading. Figure 6.1(b) describes the read protocol over these wires: the processor sets *rd'/wr* to 0, places a valid address on *addr*, and strobes *enable*, after which the memory will place valid data on the *data* lines. Figure 6.1(c) shows a write protocol: the processor sets *rd'/wr* to 1, places a valid address on *addr*, places data on *data*, and strobes *enable*, causing the memory to store the data. This very simple example brings up several points that we now describe.

Wires may be unidirectional, meaning they transmit in only one direction, as did *rd'/wr*, *enable*, and *addr*, or they may be bidirectional, meaning they transmit in two directions, though in only one direction at a time, as did *data*. A set of wires with the same function is typically drawn as a thick line and/or as a line with a small angled line drawn through it, as was the case with *addr* and *data*.

The term *bus* can refer to a set of wires with a single function within a communication. For example, we can refer to the "address bus" and the "data bus" in the above example. The term *bus* can also refer to the entire collection of wires used for the communication (e.g., *rd'/wr*, *enable*, *addr*, and *data*) along with the communication protocol over those wires. Both uses of the term are common and are often used in conjunction with one another. For example, we may say that the processor's bus consists of an address bus and a data bus. A *protocol* describes the rules for communicating over those wires. We deal primarily with low-level hardware protocols in this chapter, while higher-level protocols, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.

The bus connects to ports of a processor (or memory). A *port* is the actual conducting device, like metal, on the periphery of a processor, through which a signal is input to or output from the processor. A port may refer to a single wire, or to a set of wires with a single function, such as an address port consisting of twelve wires. A related term is *pin*. When a processor is packaged as its own IC, there are actual pins extending from the package, and those pins are often designed to be plugged into a socket on a printed-circuit board. Today, however, a processor commonly coexists on a single IC with other processors and memories. Such a processor does not have any actual pins on its periphery, but rather "pads" of metal in

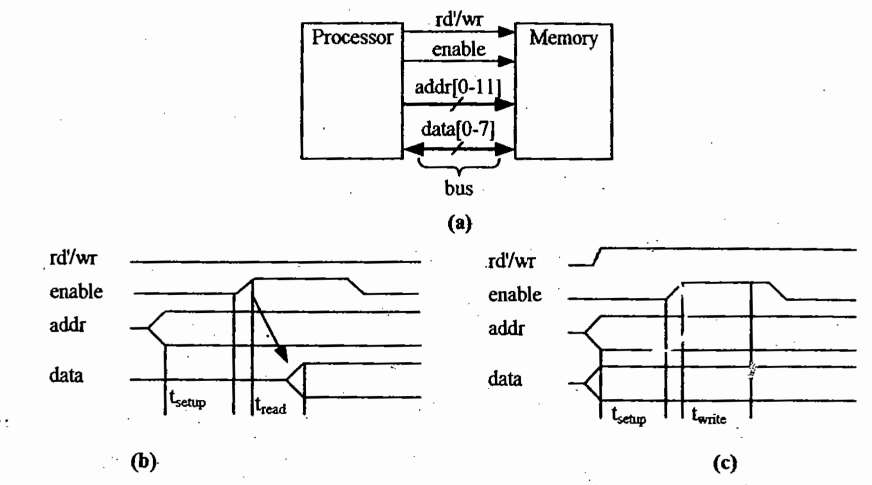


Figure 6.1: A simple bus example: (a) bus structure, (b) read protocol, (c) write protocol.

the IC. In fact, even for a processor packaged in its own IC, alternative packaging-techniques may use something other than pins for connections, such as small metallic balls. However, we can still use the term *pin* to refer to a port on a processor.

The distinction between a bus and a port is similar to the distinction between a street and a driveway — the bus is like the street, which connects various driveways. A processor's port is like a house's driveway, which provides access between the house and the street.

The most common method for describing a hardware protocol is a timing diagram, as was used in Figure 6.1(b) and (c). In the diagram, time proceeds to the right along the *x*-axis. The diagram shows that the processor must set the *rd'/wr* line low for a read to occur. The diagram also shows, using two vertical lines, that the processor must place the address on *addr* for at least t_{setup} time before setting the *enable* line high. The diagram shows that the high *enable* line triggers the memory to put data on the *data* wires after a time t_{read}. Note that a timing diagram represents control lines, like *rd'/wr* and *enable*, as either being high or low, while it represents data lines, like *addr* and *data*, either as being invalid or valid, using a single horizontal line or two horizontal lines, respectively. The actual value of data lines is not normally relevant when describing a protocol, so that value is typically not shown.

In the above protocol, the control line *enable* is active high, meaning that a 1 on the *enable* line triggers the data transfer. In many protocols, control lines are instead active low, meaning that a 0 on the line triggers the transfer. Such a control line's name is typically written with a bar above it, a single quote after it (e.g., *enable'*), a forward slash before it (e.g., */enable*), or the letter *L* after it (e.g., *enable_L*). To be general, we will use the term *assert* to mean setting a control line to its active value, such as to 1 for an active high line, and to 0 for an active low line. We will use the term *deassert* to mean setting the control line to its inactive

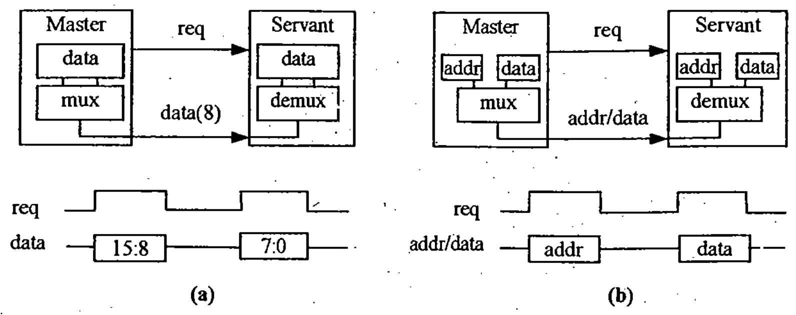


Figure 6.2: Time-multiplexed data transfer: (a) data serializing, (b) address/data muxing.

value. Notice that the *rd'/wr* of our earlier example merges two control signals into one line, so we accomplish a read by setting *rd'/wr* to 0 and a write by setting *rd'/wr* to 1.

A protocol typically consists of several possible subprotocols, such as a read protocol and a write protocol. Each subprotocol is known as a *transaction* or a *bus cycle*. A bus cycle may consist of several clock cycles.

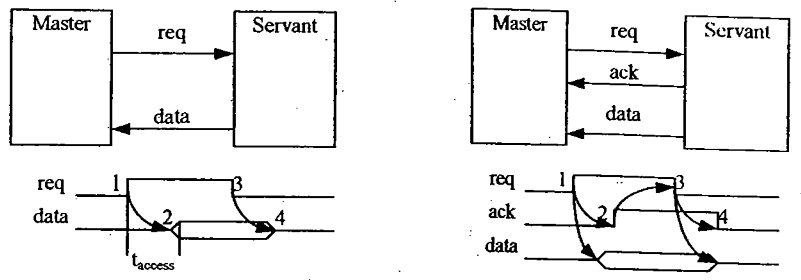
Basic Protocol Concepts

The processor-memory protocol described above was a simple one. Hardware protocols can be much more complex. However, we can understand them better by defining some basic protocol concepts. These concepts are: actors, data direction, addresses, time-multiplexing, and control methods.

An *actor* is a processor or memory involved in the data transfer. A protocol typically involves two actors: a master and a servant. A master initiates the data transfer. A servant, commonly called a slave, responds to the initiation request. In the example of Figure 6.1, the processor is the master, and the memory is the servant (i.e., the memory cannot initiate a data transfer). The servant could also be another processor. Masters are usually general-purpose processors, and servants are usually peripherals and memories.

Data direction denotes the direction that the transferred data moves between the actors. We indicate this direction by denoting each actor as either receiving or sending data. Note that actor types are independent of the direction of the data transfer. In particular, a master may either be the receiver of data, as in Figure 6.1(b), or the sender of data, as shown Figure 6.1(c).

Addresses represent a special type of data used to indicate where regular data should go to or come from. A protocol often includes both an address and regular data, as did the memory access protocol in Figure 6.1, where the address specified the location where the data should be read from or written to in the memory. An address is also necessary when a general-purpose processor communicates with multiple peripherals over a single bus; the address not only specifies a particular peripheral, but also may specify a particular register within that peripheral.



1. Master asserts *req* to receive data
2. Servant puts data on bus within time t_{access}
3. Master receives data and deasserts *req*
4. Servant ready for next request

(a)

1. Master asserts *req* to receive data
2. Servant puts data on bus and asserts *ack*
3. Master receives data and deasserts *req*
4. Servant ready for next request

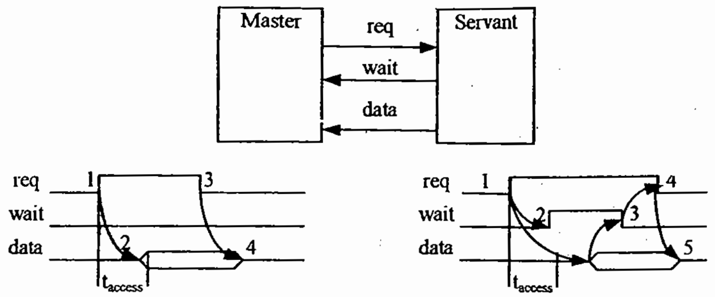
(b)

Figure 6.3: Two protocol control methods: (a) strobe, (b) handshake. The main differences are underlined.

Another protocol concept is *time multiplexing*. To multiplex means to share a single set of wires for multiple pieces of data. In time multiplexing, the multiple pieces of data are sent one at a time over the shared wires. For example, Figure 6.2(a) shows a master sending 16 bits of data over an 8-bit bus using a strobe protocol and time-multiplexed data. The master first sends the high-order byte and then the low-order byte. The servant must receive the bytes and then demultiplex the data. This serializing of data can be done to any extent, even down to a 1-bit bus, in order to reduce the number of wires. As another example, Figure 6.2(b) shows a master sending both an address and data to a servant, such as a memory. In this case, rather than using separate sets of lines for address and data, as was done in Figure 6.1, we can time multiplex the address and data over a shared set of lines *addr/data*.

Control methods are schemes for initiating and ending the transfer. Two of the most common methods are strobe and handshake. In a *strobe* protocol, the master uses one control line, often called the *request* line, to initiate the data transfer, and the transfer is considered to be complete after some fixed time interval after the initiation. For example, Figure 6.3(a) shows a strobe protocol with a master wanting to receive data from a servant. The master first asserts the request line to initiate a transfer. The servant then has time t_{access}, to put the data on the data bus. After this time, the master reads the data bus, believing the data to be valid. The master than deasserts the request line, so that the servant can stop putting the data on the data bus, and both actors are then ready for the next transfer. An analogy is a demanding boss who tells an employee "I want that report (the data) on my desk (the data bus) in one hour (t_{access})." and merely expects the report to be on the desk in one hour.

The second common control method is a *handshake* protocol, in which the master uses a request line to initiate the transfer, and the servant uses an *acknowledge* line to inform the master when the data is ready. For example, Figure 6.3(b) shows a handshake protocol with a



1. Master asserts *req* to receive data
2. Servant puts data on bus within time t_{access} (*wait* line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

(a)

1. Master asserts *req* to receive data
2. Servant can't put data within t_{access}, asserts *wait*
3. Servant puts data on bus and deasserts *wait*
4. Master receives data and deasserts *req*
5. Servant ready for next request

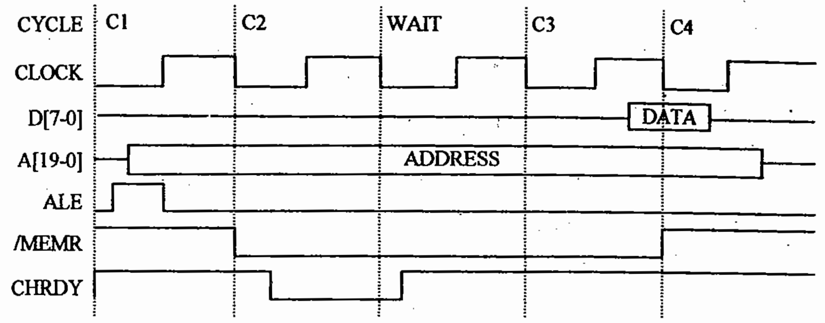
(b)

Figure 6.4: A strobe/handshake compromise: (a) fast-response, (b) slow-response. The differences are underlines.

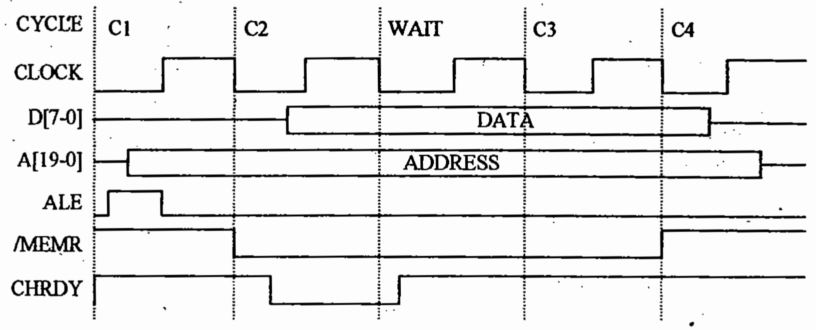
receiving master. The master first asserts the request line to initiate the transfer. The servant takes as much time as necessary to put the data on the data bus, and then asserts the acknowledge line to inform the master that the data is valid. The master reads the data bus and then deasserts the request line so that the servant can stop putting data on the data bus. The servant deasserts the acknowledge line, and both actors are then ready for the next transfer. In our boss-employee analogy, a handshake protocol corresponds to a more tolerant boss who tells an employee "I want that report on my desk soon; let me know when it's ready." A handshake protocol can adjust to a servant, or servants, with varying response times, unlike a strobe protocol. However, when response time is known, a handshake protocol may be slower than a strobe protocol, since it requires the master to detect the acknowledgment before getting the data, possibly requiring an extra clock cycle if the master is synchronizing the bus control signals. A handshake also requires an extra line for acknowledge.

To achieve both the speed of a strobe protocol and the varying response time tolerance of a handshake protocol, a compromise protocol is often used, as illustrated in Figure 6.4. In this case, when the servant can put the data on the bus within time t_{access}, the protocol is identical to a strobe protocol, as shown in Figure 6.4(a). However, if the servant cannot put the data on the bus in time, it instead tells the master to wait longer, by asserting a line we've labeled *wait*. When the servant has finally put the data on the bus, it deasserts the *wait* line, thus informing the master that the data is ready. The master receives the data and deasserts the *request* line. Thus, the handshake only occurs if it is necessary. In our boss-employee analogy, the boss tells the employee "I want that report on my desk in an hour; if you can't finish by then, let me know that and then let me know when it's ready."

Perhaps the most common communication situation in embedded systems is the input and output (I/O) of data to and from a general-purpose processor, as it communicates with its



(a)



(b)

Figure 6.5: ISA bus protocol: (a) read bus timing, (b) write bus timing.

peripherals and memories. I/O is relative to the processor: input means data comes into the processor, while output means data goes out of the processor. In the next three sections, we will discuss three microprocessor-interfacing issues: addressing, interrupts, and direct memory access. We'll use the term *microprocessor* to refer to a general-purpose processor.

Example: The ISA Bus Protocol — Memory Access

The Industry Standard Architecture (ISA) bus protocol is common in systems using an 80x86 microprocessor. Figure 6.5(a) illustrates the bus timing for performing a memory read operation, referred to as a *memory read cycle*. During a memory read cycle, the microprocessor drives the bus signals to read a byte of data from memory. Note that in Figure 6.5(a), several other control signals that are inactive during a memory read cycle are not included in the timing diagram. The operation works as follows. In clock cycle *C*1, the microprocessor puts a 20-bit memory address on the address lines *A* and asserts the address latch enable signal *ALE*. During clock cycles *C*2 and *C*3, the processor asserts the memory read signal *MEMR* to request a read operation from the memory device. After *C*3, the memory device holds the data on data lines *D*. In cycle *C*4, all signals are deasserted.

The ISA read bus cycle uses a compromise strobe/handshake control method. The memory device deasserted the channel ready signal *CHRDY* before the rising clock edge in *C*2, causing the microprocessor to insert wait cycles until *CHRDY* was reasserted. Up to six wait cycles can be inserted by a slow device.

Figure 6.5(b) illustrates the bus timing for performing a memory write operation, referred to as a *memory write cycle*. During a memory write bus cycle, the microprocessor drives the bus signals to write a byte of data to memory. The operation works as follows. In clock cycle *C*1, the processor puts the 20-bit memory address to be written on the address lines and asserts the *ALE* signal. During cycles *C*2 and *C*3, the processor puts the write data on the data lines and asserts the memory write signal *MEMW* to indicate a write operation to the memory device. In cycle *C*4, all signals are deasserted. The write cycle also uses a compromise strobe/handshake control method.

6.3 Microprocessor Interfacing: I/O Addressing

Port and Bus-Based I/O

A microprocessor may have tens or hundreds of pins, many of which are control pins, such as a pin for clock input and another input pin for resetting the microprocessor. Many of the other pins are used to communicate data to and from the microprocessor, which we call processor I/O. There are two common methods for using pins to support I/O: port-based I/O and bus-based I/O.

In *port-based I/O*, also known as *parallel I/O*, a port can be directly read and written by processor instructions, just like any other register in the microprocessor; in fact, the port is usually connected to a dedicated register. For example, consider an 8-bit port named *P*0. A C-language programmer may write to *P*0 using an instruction like: *P*0 = 255, which would set all eight pins to 1s. In this case, the C compiler manual would have defined *P*0 as a special variable that would automatically be mapped to the register *P*0 during compilation. Conversely, the programmer might read the value of a port *P*1 being written by some other device by typing something like *a* = *P*1. In some microprocessors, each bit of a port can be configured as input or output by writing to a configuration register for the port. For example, *P*0 might have an associated configuration register called *CP*0. To set the high-order four bits to input and the low-order four bits to output, we might say: *CP*0 = 15. This writes 00001111 to the *CP*0 register, where a 0 means input and a 1 means output. Ports are often bit-addressable, meaning that a programmer can read or write specific bits of the port. For example, one might say: *x* = *P*0.2, giving *x* the value of the number 2 pin of port *P*0.

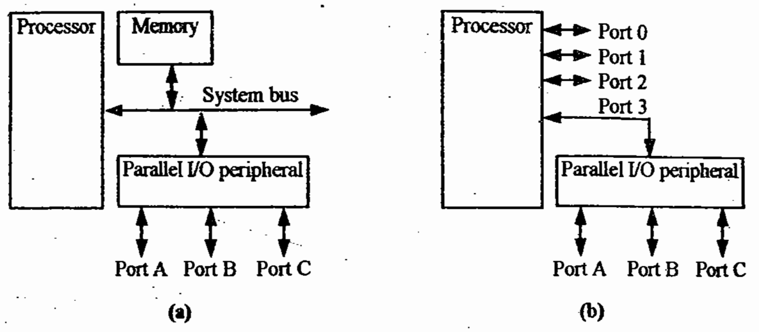


Figure 6.6: Parallel I/O: (a) adding parallel I/O to a bus-based I/O processor, (b) extended parallel I/O.

In *bus-based I/O*, the microprocessor has a set of address, data, and control ports corresponding to bus lines, and uses the bus to access memory as well as peripherals. The microprocessor has the bus protocol built in to its hardware. Specifically, the software does not implement the protocol but merely executes a single instruction that in turn causes the hardware to write or read data over the bus. We normally consider the access to the peripherals as I/O, but don't normally consider the access to memory as I/O, since the memory is considered more as a part of the microprocessor.

A system may require parallel I/O (port-based I/O), but a microprocessor may only support bus-based I/O. In this case, a parallel I/O peripheral may be used, as illustrated in Figure 6.6(a). The peripheral is connected to the system bus on one side, with corresponding address, data, and control lines, and has several ports on the other side, consisting just of a set of data lines. The ports are connected to registers inside the peripheral, and the microprocessor can read and write those registers in order to read and write the ports.

Even when a microprocessor supports port-based I/O, we may require more ports than are available. In this case, a parallel I/O peripheral can again be used, as illustrated in Figure 6.6(b). The microprocessor has four ports in this example, one of which is used to interface with a parallel I/O peripheral, which itself has three ports. Thus, we have extended the number of available ports from four to six. Using such a peripheral in this manner is often referred to as *extended parallel I/O*.

Memory-Mapped I/O and Standard I/O

In bus-based I/O, there are two methods for a microprocessor to communicate with peripherals, known as memory-mapped I/O and standard I/O.

In *memory-mapped I/O*, peripherals occupy specific addresses in the existing address space. For example, consider a bus with a 16-bit address. The lower 32K addresses may correspond to memory addresses, while the upper 32K may correspond to I/O addresses.

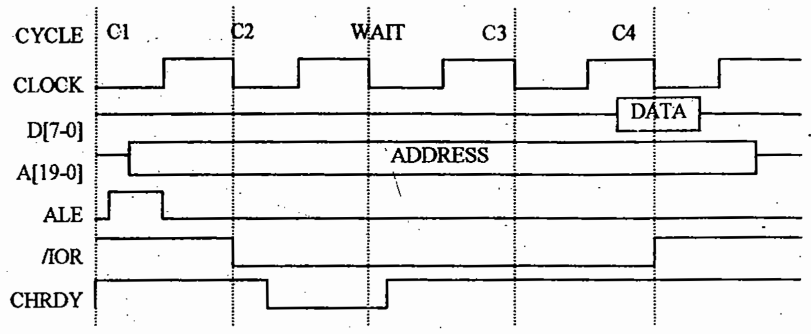


Figure 6.7: ISA bus protocol for standard I/O.

In *standard I/O* (also known as *I/O-mapped I/O*), the bus includes an additional pin, which we label *M/IO*, to indicate whether the access is to memory or to a peripheral (i.e., an I/O device). For example, when *M/IO* is 0, the address on the address bus corresponds to a memory address. When *M/IO* is 1, the address corresponds to a peripheral.

An advantage of memory-mapped I/O is that the microprocessor need not include special instructions for communicating with peripherals. The microprocessor's assembly instructions involving memory, such as MOV or ADD, will also work for peripherals. For example, a microprocessor may have an ADD A, B instruction that adds the data at address B to the data at address A and stores the result in A. A and B may correspond to memory locations, or registers in peripherals. In contrast, if the microprocessor uses standard I/O, the microprocessor requires special instructions for reading and writing peripherals. These instructions are often called IN and OUT. Thus, to perform the same addition of locations A and B corresponding to peripherals, the following instructions would be necessary:

```
IN R0, A
IN R1, B
ADD R0, R1
OUT A, R0
```

Advantages of standard I/O include no loss of memory addresses to the use as I/O addresses, and potentially simpler address decoding logic in peripherals. Address decoding logic can be simplified with standard I/O if we know that there will only be a small number of peripherals, because the peripherals can then ignore high-order address bits. For example, a bus may have a 16-bit address, but we may know there will never be more than 256 I/O addresses required. The peripherals can thus safely ignore the high-order 8 address bits, resulting in smaller and/or faster address comparators in each peripheral. Note that we can build a system using both standard and memory-mapped I/O, since peripherals in the memory space act just like memory themselves.

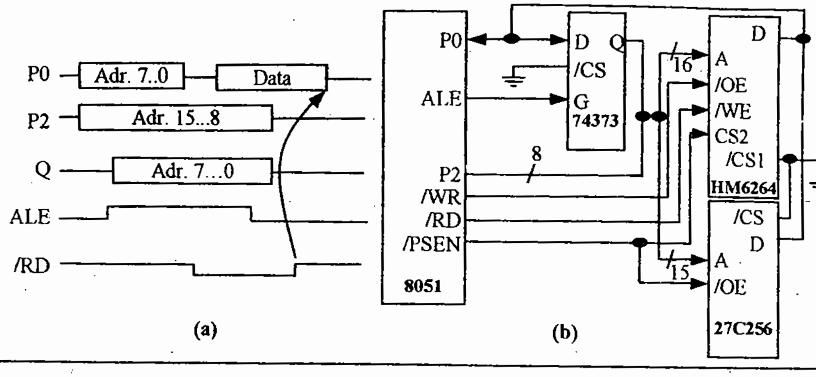


Figure 6.8: A basic memory protocol: (a) timing diagram for a read operation, (b) interface schematic.

Example: The ISA Bus Protocol — Standard I/O

The ISA bus protocol introduced earlier supports standard I/O. The I/O read bus cycle is depicted in Figure 6.7. During this bus cycle, the microprocessor drives the bus signals to read a byte of data from a peripheral, according to the timing diagram shown. Note that the cycle uses a control line distinct from */MEMR*, namely */IOR*, which is consistent with the standard I/O approach. The I/O device address space is limited to 16 bits, as opposed to 20 bits for memory devices. The I/O write bus cycle is similar to the memory write bus cycle but uses a control signal */IOW* and again limits the address to 16 bits. The I/O read and write bus cycles use the compromise strobe/handshake control method, as did the memory bus cycles.

Example: A Basic Memory Protocol

In this example, we illustrate how to interface 8K of data and 32K of program code memory to a microcontroller, specifically the Intel 8051. The 8051 uses separate memory address spaces for data and program code. Data or code address space is limited to 64K, hence, addressable with 16 bits through ports *P0* (least significant bits) and *P2* (most significant bits). A separate signal, called *PSEN* (program strobe enable), is used to distinguish between data/code. For the most part, the 8051 generates all of the necessary signals to perform memory I/O, however, since port *P0* is used both for the least significant address bits and for data, an 8-bit latch is required to perform the necessary multiplexing. The timing diagram depicted in Figure 6.8(a) illustrates a memory read operation. A memory write operation is performed in a similar fashion with data flow reversed and *RD* (read) replaced with *WR* (write). The memory read operation proceeds as follows. The microcontroller places the source address (i.e., the memory location to be read) on ports *P2* and *P0*. *P2*, holding the eight most significant address bits, retains its value throughout the read operation. *P0*, holding the eight least-significant address bits, is stored inside an 8-bit latch. The *ALE* signal (address

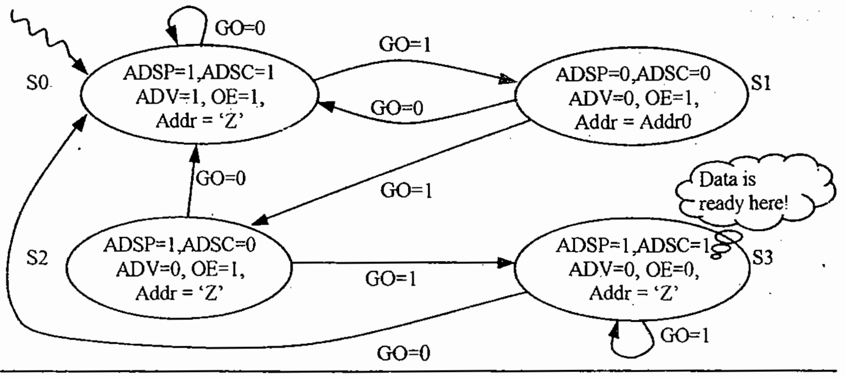


Figure 6.9: A complex memory protocol.

latch enable) is used to trigger the latching of port *P*0. Now, the microcontroller asserts high impedance on *P*0 to allow the memory device to drive it with the requested data. The memory device outputs valid data as long as the *RD* signal is asserted. Meanwhile, the microcontroller reads the data and deasserts its control and port signals. Figure 6.8(b) illustrates the interface schematic.

Example: A Complex Memory Protocol

In this example, we will build a finite-state machine (FSM) controller that will generate all the necessary control signals to drive the TC55V2325FF memory chip in burst read mode (i.e., pipelined read operation), as described in Chapter 5. Our specification for this FSM is the timing diagram presented in the earlier example from Chapter 5. The input to our machine is a clock signal (*CLK*), the starting address (*Addr0*) and the enable/disable signal (*GO*). The output of our machine is a set of control signals specific to our memory device. We assume that the chip's *enable* and *WE* signals are asserted. Figure 6.9 gives the FSM description. From the state machine description, we can derive the next-state and output truth tables. From these truth tables, we can compute next-state and output equations. By deriving the next-state transition table, we can solve and optimize the next-state and output equations. These equations can be implemented using logic components. (See Chapter 2 for details.) Any processor that is to be interfaced with one of these memory devices must implement, internally or externally, a state machine similar to the one presented in this example.

6.4 Microprocessor Interfacing: Interrupts

Another microprocessor I/O issue is that of interrupt-driven I/O. To introduce this issue, suppose the program running on a microprocessor must, among other tasks, read and process

data from a peripheral whenever that peripheral has new data; such processing is called *servicing*. If the peripheral gets new data at unpredictable intervals, how can the program determine when the peripheral has new data? The most straightforward approach is to interleave the microprocessor's other tasks with a routine that checks for new data in the peripheral, perhaps by checking for a 1 in a particular bit in a register of the peripheral. This repeated checking by the microprocessor for data is called *polling*. Polling is simple to implement, but this repeated checking wastes many clock cycles, so it may not be acceptable in many cases, especially when there are numerous peripherals to be checked. We could check at less-frequent intervals, but then we may not process the data quickly enough.

To overcome the limitations of polling, most microprocessors come with a feature called *external interrupt*. A microprocessor with this feature has a pin, say, *Int*. At the end of executing each machine instruction, the processor's controller checks *Int*. If *Int* is asserted, the microprocessor jumps to a particular address at which a subroutine exists that services the interrupt. This subroutine is called an *interrupt service routine*, or ISR. Such I/O is called *interrupt-driven I/O*.

One might wonder if interrupts have really solved the problem with polling, namely of wasting time performing excessive checking, since the interrupt pin is "polled" at the end of every microprocessor instruction. However, in this case, the polling of the pin is built right into the microprocessor's controller hardware, and therefore can be done simultaneously with the execution of an instruction, resulting in no extra clock cycles.

There are two methods by which a microprocessor using interrupts determines the address, known as the *interrupt address vector*, at which the ISR resides. These two methods are *fixed* and *vectored* interrupt. In *fixed interrupt*, the address to which the microprocessor jumps on an interrupt is built into the microprocessor, so it is fixed and cannot be changed. The assembly programmer either puts the ISR at that address, or if not enough bytes are available in that region of memory, merely puts a jump to the real ISR there. For C programmers, the compiler typically reserves a special name for the ISR and then compiles a subroutine having that name into the ISR location, or again just a jump to that subroutine. In microprocessors with fixed ISR addresses, there may be several interrupt pins to support interrupts from multiple peripherals.

Figure 6.10 provides a summary of the flow of actions for an example of interrupt-driven I/O using a fixed ISR address. Figure 6.11 illustrates this flow graphically for the example. In this example, data received by *Peripheral*1 must be read, transformed, and then written to *Peripheral*2. *Peripheral*1 might represent a sensor, and *Peripheral*2, a display. Meanwhile, the microprocessor is running its main program, located in program memory starting at address 100. When *Peripheral*1 receives data, it asserts *Int* to request that the microprocessor service the data. After the microprocessor completes execution of its current instruction, it stores its state and jumps to the ISR located at the fixed program memory location of 16. The ISR reads the data from *Peripheral*1, transforms it, and writes the result to *Peripheral*2. The last ISR instruction is a return from interrupt, causing the microprocessor to restore its state and resume execution of its main program, in this case executing instruction 101.

Other microprocessors use *vectored interrupt* to determine the address at which the ISR resides. This approach is especially common in systems with a system bus, since there may be

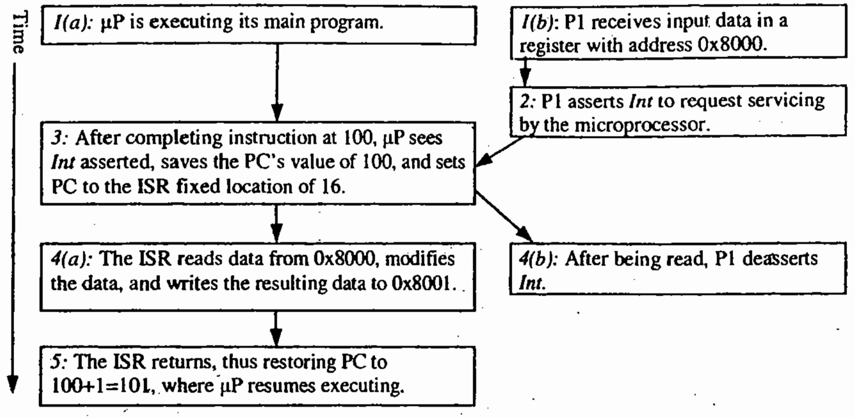

Time
1(a): µP is executing its main program.
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts Int to request servicing by the microprocessor.
3: After completing instruction at 100, µP sees Int asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.
4(b): After being read, P1 deasserts Int.
5: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

Figure 6.10: Interrupt-driven I/O using fixed ISR location: summary of flow of actions.

numerous peripherals that can request service. In this method, the microprocessor has one interrupt pin, say, *Int*, which any peripheral can assert. After detecting the interrupt, the microprocessor asserts another pin, say, *Inta*, to acknowledge that it has detected the interrupt and to request that the interrupting peripheral provide the address where the relevant ISR resides. The peripheral provides this address on the data bus, and the microprocessor reads the address and jumps to the corresponding ISR. We discuss the situation where multiple peripherals simultaneously request servicing in a later section on arbitration. For now, consider an example of one peripheral using vectored interrupt. The flow of actions is shown in Figure 6.12, which represents an example very similar to the previous one. Figure 6.13 illustrates the example graphically. In contrast to the earlier example, the ISR location is not fixed at 16. Thus, *Peripheral*1 contains an extra register holding the ISR location. After detecting the interrupt and saving its state, the microprocessor asserts *Inta* in order to get *Peripheral*1 to place 16 on the data bus. The microprocessor reads this 16 into the PC and then jumps to the ISR, which executes and completes in the same manner as the earlier example.

As a compromise between the fixed and vectored interrupt methods, we can use an *interrupt address table*. In this method, we still have only one interrupt pin on the processor, but we also create in the processor's memory a table that holds ISR addresses. A typical table might have 256 entries. A peripheral, rather than providing the ISR address, instead provides a number corresponding to an entry in the table. The processor reads this entry number from the bus, and then reads the corresponding table entry to obtain the ISR address. Compared to the entire memory, the table is typically very small, so an entry number's bit encoding is small. This small bit encoding is especially important when the data bus is not wide enough to hold a complete ISR address. Furthermore, this approach allows us to assign each peripheral a

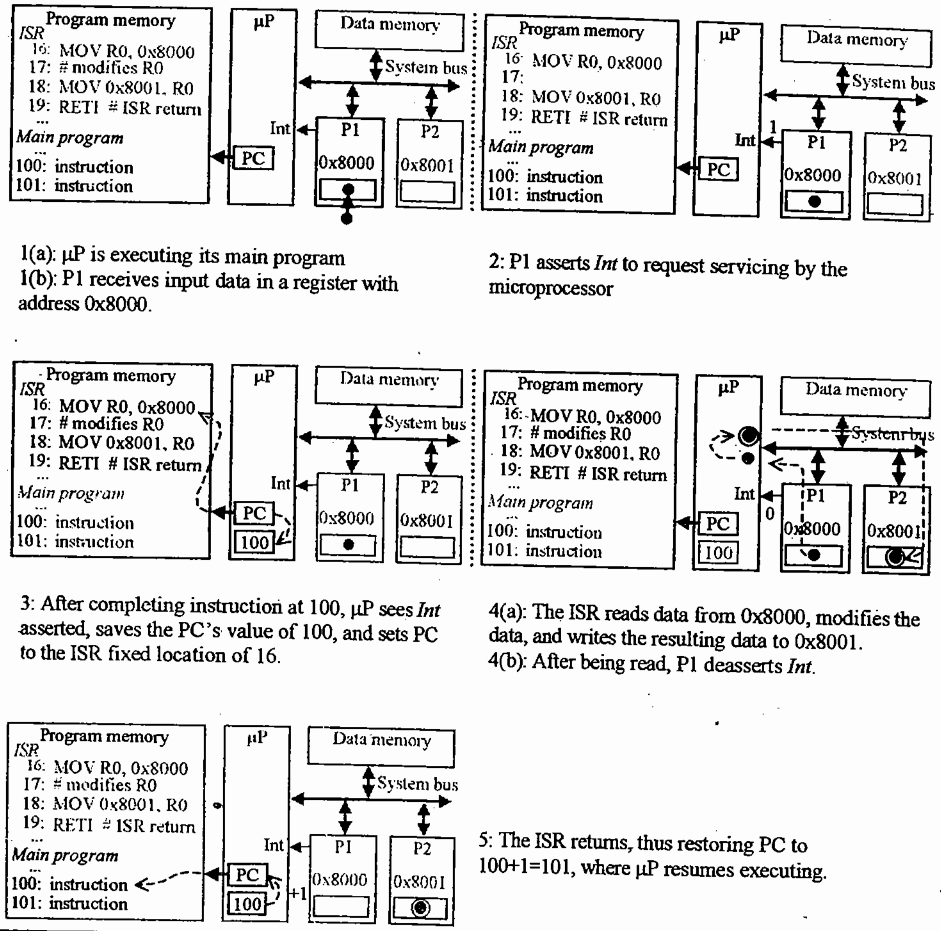
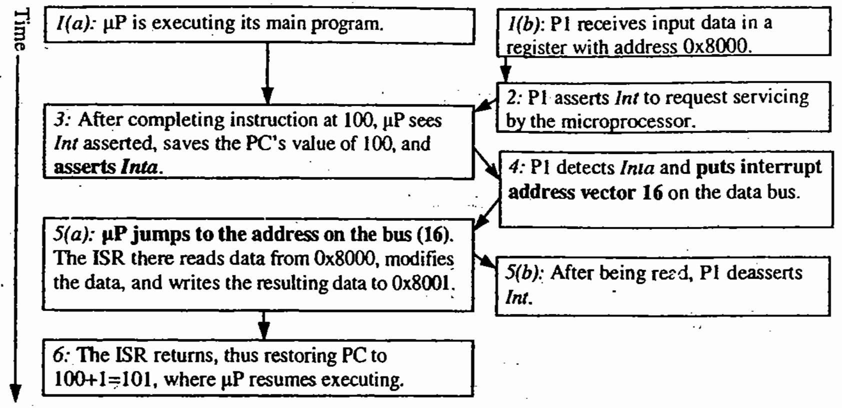

Program memory
ISR
16: MOV R0, 0x8000
17: # modifies R0
18: MOV 0x8001, R0
19: RETI # ISR return
Main program
100: instruction
101: instruction
µP
PC
Int
Data memory
System bus
P1
P2
0x8000
0x8001
1(a): µP is executing its main program
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts Int to request servicing by the microprocessor
3: After completing instruction at 100, µP sees Int asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.
4(b): After being read, P1 deasserts Int.
5: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.
100
+1

Figure 6.11: Interrupt-driven I/O using fixed ISR location: flow of actions.

unique number independent of ISR locations, meaning that we could move the ISR location without having to change anything in the peripheral.

External interrupts may be maskable or nonmaskable. In *maskable* interrupt, the programmer may force the microprocessor to ignore the interrupt pin, either by executing a specific instruction to disable the interrupt or by setting bits in an interrupt configuration register. A situation where a programmer might want to mask interrupts is when there exist time-critical regions of code, such as a routine that generates a pulse of a certain duration. The



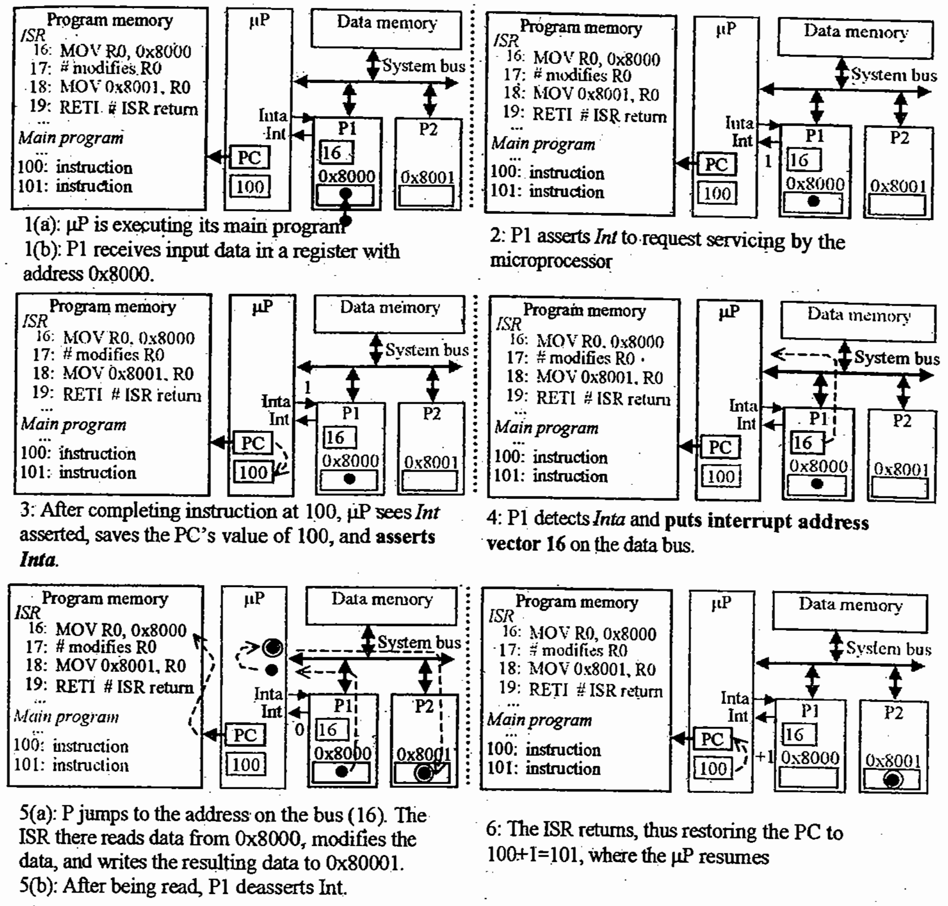
Time
1(a): µP is executing its main program.
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts Int to request servicing by the microprocessor.
3: After completing instruction at 100, µP sees Int asserted, saves the PC's value of 100, and asserts Inta.
4: P1 detects Inta and puts interrupt address vector 16 on the data bus.
5(a): µP jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.
5(b): After being read, P1 deasserts Int.
6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

Figure 6.12: Interrupt-driven I/O using vectored interrupt: summary of flow of actions.

programmer may include an instruction that disables interrupts at the beginning of the routine, and another instruction reenabling interrupts at the end of the routine. *Nonmaskable* interrupt cannot be masked by the programmer. It requires a pin distinct from maskable interrupts. It is typically used for very drastic situations, such as power failure. In this case, if power is failing, a nonmaskable interrupt can cause a jump to a subroutine that stores critical data in nonvolatile memory, before power is completely gone.

In some microprocessors, the jump to an ISR is handled just like the jump to any other subroutine, meaning that the state of the microprocessor is stored on a stack, including contents of the program counter, datapath status register, and all other registers. The state is then restored upon completion of the ISR. In other microprocessors, only a few registers are stored, like just the program counter and status registers. The assembly programmer must be aware of what registers have been stored, so as not to overwrite nonstored register data with the ISR. These microprocessors need two types of assembly instructions for subroutine return. A regular return instruction returns from a regular subroutine, which was called using a subroutine call instruction. A return from interrupt instruction returns from an ISR, which was jumped to not by a call instruction but by the hardware itself, and which restores only those registers that were stored at the beginning of the interrupt. The C programmer is freed from having to worry about such considerations, as the C compiler handles them.

The reason we used the term *external interrupt* is to distinguish this type of interrupt from internal interrupts, also called *traps*. An internal interrupt results from an exceptional condition, such as divide-by-0, or execution of an invalid opcode. Internal interrupts, like external ones, result in a jump to an ISR. A third type of interrupt, called *software interrupts*, can be initiated by executing a special assembly instruction.



Program memory
ISR
16: MOV R0, 0x8000
17: # modifies R0
18: MOV 0x8001, R0
19: RETI # ISR return
Main program
100: instruction
101: instruction
µP
Data memory
System bus
Inta
Int
PC
100
P1
P2
16
0x8000
0x8001
1
0
+1
1(a): µP is executing its main program.
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts Int to request servicing by the microprocessor
3: After completing instruction at 100, µP sees Int asserted, saves the PC's value of 100, and asserts Inta.
4: P1 detects Inta and puts interrupt address vector 16 on the data bus.
5(a): P jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x80001.
5(b): After being read, P1 deasserts Int.
6: The ISR returns, thus restoring the PC to 100+1=101, where the µP resumes

Figure 6.13: Interrupt-driven I/O using vectored interrupt: flow of actions.

6.5 Microprocessor Interfacing: Direct Memory Access

Commonly, the data being accumulated in a peripheral should be first stored in memory before being processed by a program running on the microprocessor. Such temporary storage of data that is awaiting processing is called *buffering*. For example, packet data from an

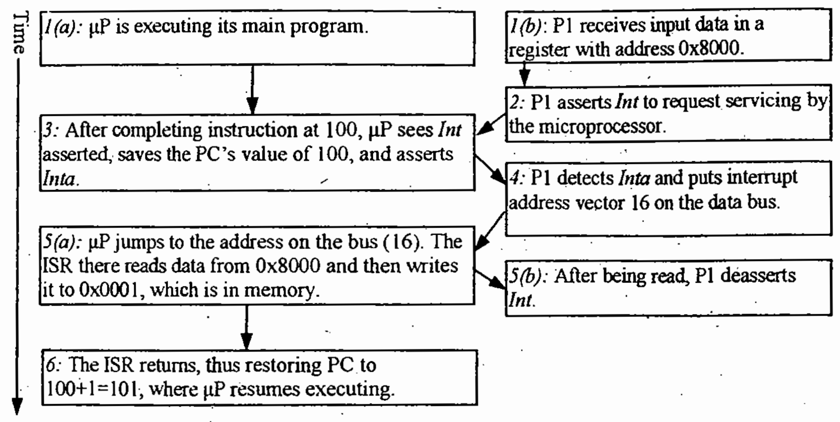

Time
1(a): µP is executing its main program.
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts Int to request servicing by the microprocessor.
3: After completing instruction at 100, µP sees Int asserted, saves the PC's value of 100, and asserts Inta.
4: P1 detects Inta and puts interrupt address vector 16 on the data bus.
5(a): µP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.
5(b): After being read, P1 deasserts Int.
6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

Figure 6.14: Peripheral to memory transfer without DMA, using vectored interrupt: summary of flow of actions.

Ethernet card is stored in main memory and is later processed by the different software layers (e.g., Internet Protocol stacks). We could write a simple interrupt service routine on the microprocessor, such that the peripheral device would interrupt the microprocessor whenever it had data to be stored in memory. The ISR would simply transfer data from the peripheral to the memory, and then resume running its application. For example, Figure 6.14 provides a summary of the flow of actions for an example in which peripheral *P1* interrupts the microprocessor when receiving new data. Figure 6.15 illustrates the example graphically. In this example, the microprocessor jumps to ISR location 16, which moves the data from 0x8000 in the peripheral to 0x0001 in memory. Afterward, the ISR returns. However, recall that jumping to an ISR requires the microprocessor to store its state (i.e., register contents), and then to restore its state when returning from the ISR. This storing and restoring of the state may consume many clock cycles, and is thus somewhat inefficient. Furthermore, the microprocessor cannot execute its regular program while moving the data, resulting in further inefficiency.

The I/O method of *direct memory access* (DMA) eliminates these inefficiencies. In DMA, we use a separate single-purpose processor, called a *DMA controller*, whose sole purpose is to transfer data between memories and peripherals. Briefly, the peripheral requests servicing from the DMA controller, which then requests control of the system bus from the microprocessor. The microprocessor merely needs to relinquish control of the bus to the DMA controller. The microprocessor does not need to jump to an ISR, and thus the overhead of storing and restoring the microprocessor state is eliminated. Furthermore, the microprocessor can execute its regular program while the DMA controller has bus control, as long as that regular program doesn't require use of the bus (at which point the microprocessor would then have to wait for the DMA to complete).

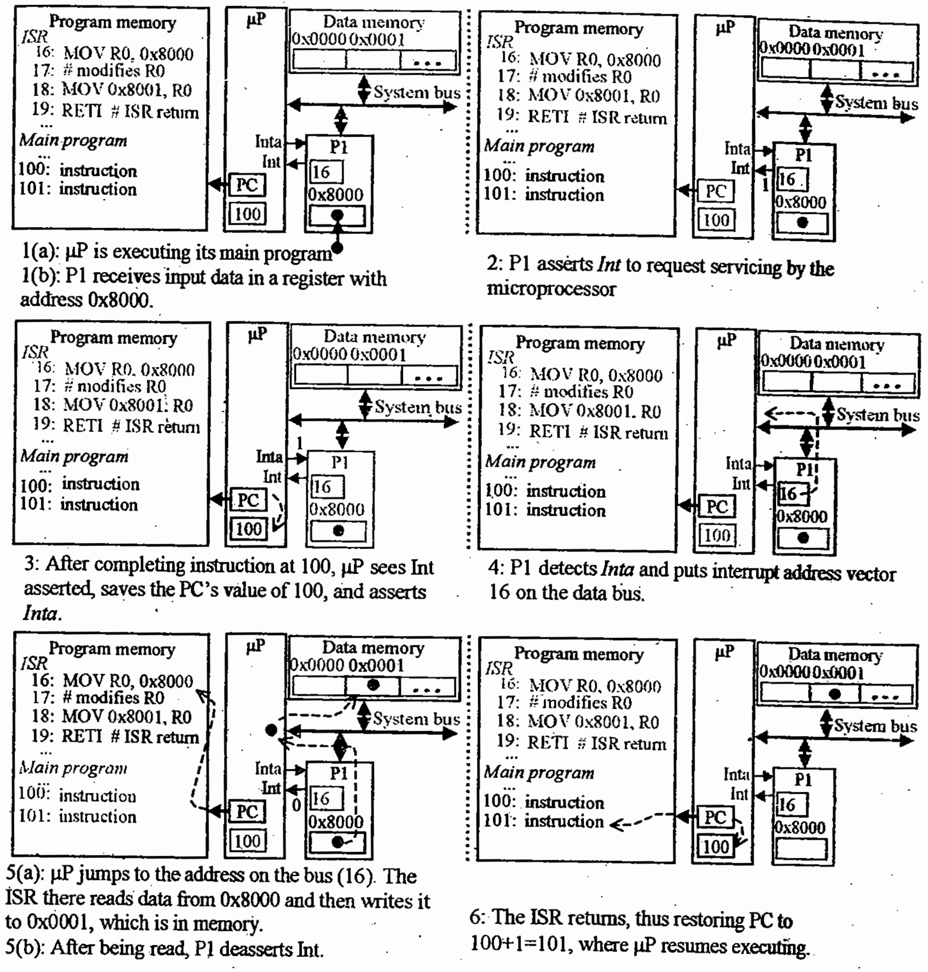

Program memory
ISR
16: MOV R0, 0x8000
17: # modifies R0
18: MOV 0x8001, R0
19: RETI # ISR return
Main program
100: instruction
101: instruction
µP
Data memory
0x0000 0x0001
System bus
Inta
Int
PC
100
P1
16
0x8000
1(a): µP is executing its main program
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts Int to request servicing by the microprocessor
3: After completing instruction at 100, µP sees Int asserted, saves the PC's value of 100, and asserts Inta.
4: P1 detects Inta and puts interrupt address vector 16 on the data bus.
5(a): µP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.
5(b): After being read, P1 deasserts Int.
6: The ISR returns, thus restoring PC to 100+1=101, where µP resumes executing.

Figure 6.15: Peripheral to memory transfer without DMA, using vectored interrupt: flow of actions.

A system with a separate bus between the microprocessor and cache may be able to execute for some time from the cache while the DMA transfer takes place.

Figure 6.16 summarizes the flow of actions for an example transfer using DMA, and Figure 6.17 depicts the example graphically. As seen in Figure 6.17, we connect the peripheral to the DMA controller rather than the microprocessor. Note that the peripheral does not recognize any difference between being connected to a DMA controller device or a

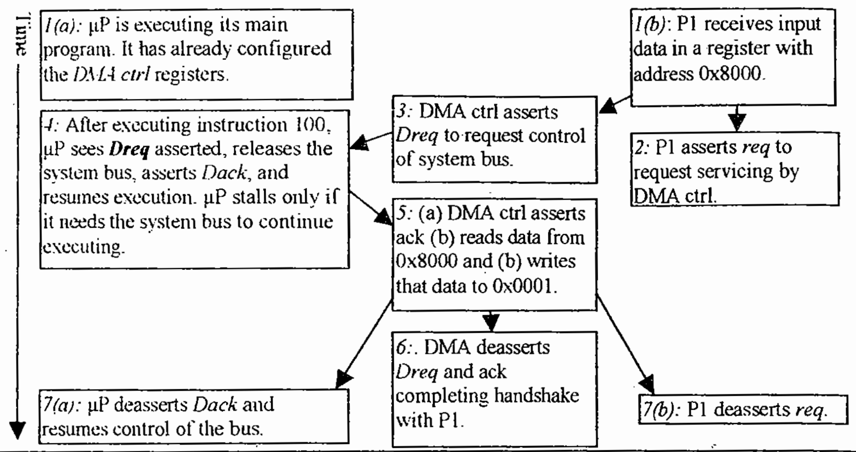

Time
1(a): µP is executing its main program. It has already configured the DMA ctrl registers.
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts req to request servicing by DMA ctrl.
3: DMA ctrl asserts Dreq to request control of system bus.
4: After executing instruction 100, µP sees Dreq asserted, releases the system bus, asserts Dack, and resumes execution. µP stalls only if it needs the system bus to continue executing.
5: (a) DMA ctrl asserts ack (b) reads data from 0x8000 and (b) writes that data to 0x0001.
6:. DMA deasserts Dreq and ack completing handshake with P1.
7(a): µP deasserts Dack and resumes control of the bus.
7(b): P1 deasserts req.

Figure 6.16: Peripheral to memory transfer with DMA: summary of flow of actions.

microprocessor device: all the peripheral knows is that it asserts a request signal on the device, and then that device services the peripheral's request. We connect the DMA controller to two special pins of the microprocessor. One pin, which we'll call *Dreq*, is used by the DMA controller to request control of the bus. The other pin, which we'll call *Dack*, is used by the microprocessor to acknowledge to the DMA controller that bus control has been granted. Thus, unlike the peripheral, the microprocessor must be specially designed with these two pins in order to support DMA. The DMA controller also connects to all the system bus signals, including address, data, and control lines.

To achieve this we must have configured the DMA controller to know what addresses to access in the peripheral and the memory. Such setting of addresses may be done by a routine running on the microprocessor during system initialization. In particular, during initialization, the microprocessor writes to configuration registers in the DMA controller just as it would write to any other peripheral's registers. Alternatively, in an embedded system that is guaranteed not to change, we can hardcode the addresses directly into the DMA controller. In the example of Figure 6.17, we see two registers in the DMA controller holding the peripheral register address and the memory address.

During its control of the system bus, the DMA controller might transfer just one piece of data, but more commonly will transfer numerous pieces of data (called a *block*), one right after other, before relinquishing the bus. This is because many peripherals, such as any peripheral that deals with storage devices (e.g., CD-ROM players or disk controllers) or that deals with network communication, send and receive data in large blocks. For example, a particular disk controller peripheral might read data in blocks of 128 words and store this data in a 128-word internal memory, after which the peripheral requests servicing (i.e., requests that this data be buffered in memory).

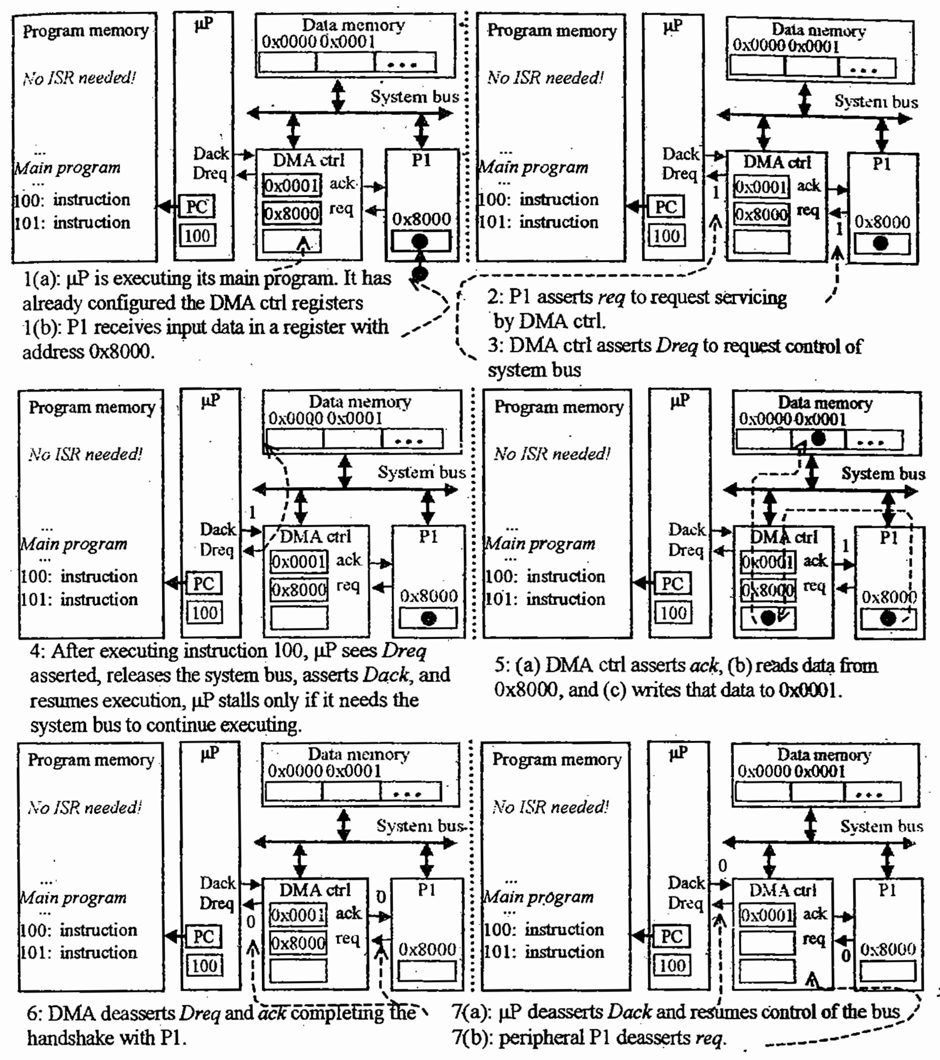

Program memory
µP
Data memory
0x0000 0x0001
No ISR needed!
Main program
100: instruction
101: instruction
System bus
Dack
Dreq
PC
100
DMA ctrl
0x0001
0x8000
ack
req
P1
0x8000
1(a): µP is executing its main program. It has already configured the DMA ctrl registers
1(b): P1 receives input data in a register with address 0x8000.
2: P1 asserts req to request servicing by DMA ctrl.
3: DMA ctrl asserts Dreq to request control of system bus
4: After executing instruction 100, µP sees Dreq asserted, releases the system bus, asserts Dack, and resumes execution, µP stalls only if it needs the system bus to continue executing.
5: (a) DMA ctrl asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.
6: DMA deasserts Dreq and ack completing the handshake with P1.
7(a): µP deasserts Dack and resumes control of the bus.
7(b): peripheral P1 deasserts req.

Figure 6.17: Peripheral to memory transfer with DMA: flow of actions.

For the example just given, the DMA controller works as follows. The DMA controller gains control of the bus, makes 128 peripheral reads and memory writes, and only then relinquishes the bus. We must therefore configure the DMA controller to operate in either

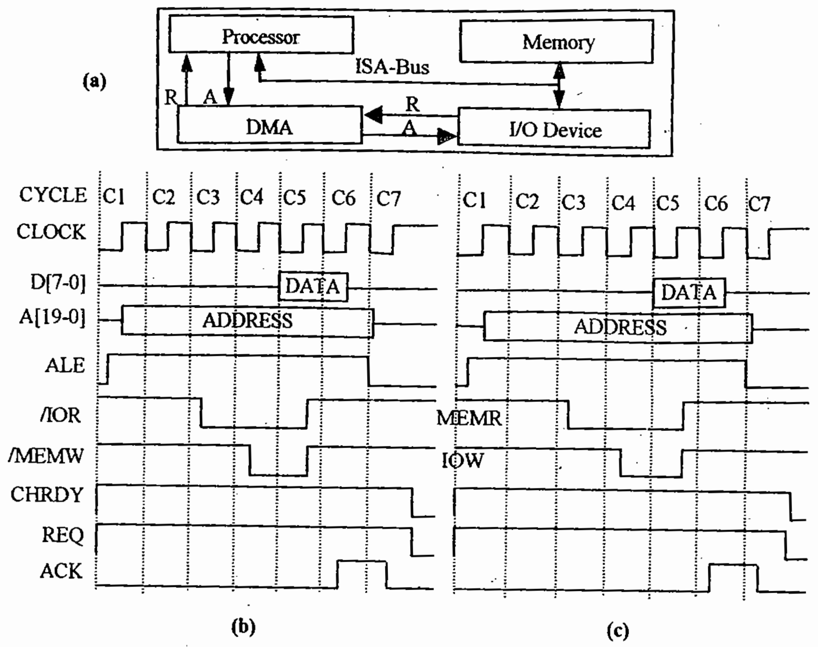


Figure 6.18: DMA using the ISA bus protocol: (a) system architecture, (b) DMA write cycle, (c) DMA read cycle.

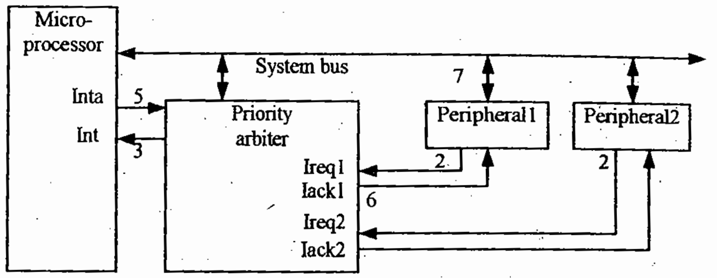
single transfer mode or block transfer mode. For block transfer mode, we must configure a base address as well as the number of words in a block.

DMA controllers typically come with numerous channels. Each channel supports one peripheral. Each channel has its own set of configuration registers. Some modern peripherals come with DMA capabilities built into the peripheral itself.

Example: DMA I/O and the ISA Bus Protocol

In an earlier example, we introduced the basic ISA memory and peripheral I/O read and write bus cycles. In this example, we will introduce the DMA related bus cycles. Our sample architecture is extended now to include a DMA controller as shown in Figure 6.18(a). In this figure, *R* denotes the DMA request signal and *A* denotes the DMA acknowledge signal.

DMA is used to perform memory writes/reads to/from I/O devices directly without the intervention of the processor. Let us first look at the DMA memory write bus cycle. A DMA write bus cycle proceeds as follows. First, the processor programs the DMA controller to monitor a particular I/O device for available data. The processor also programs the DMA with the starting memory address where the data item is to be written to. Once the I/O device has available data, it generates a DMA request by asserting its DMA request line (*DRQ*). In



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq*1. Peripheral2 also needs servicing so asserts *Ireq*2.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor asserts *Inta*.
6. Priority arbiter asserts *Iack*1 to acknowledge Peripheral1.
7. Peripheral1 puts its interrupt address vector on the system bus.
8. Microprocessor jumps to the address of the ISR read from the data bus, ISR executes and returns (and completes handshake with arbiter).
9. Microprocessor resumes executing its program.

Figure 6.19: Arbitration using a priority arbiter.

response to this, the DMA controller will assert its *DRQ* to signal the processor. The processor, then, relinquishes the bus control signals and signals to the DMA controller with an acknowledgment (*DACK*). In response, the DMA will acknowledge the I/O device's *DRQ* by asserting its *DACK*. At this point, the actual transfer of data from the device to memory is initiated. Note that the actual DMA signals (*DACK*s and *DRQ*s) are not part of the ISA protocol. The ISA protocol merely provides a scheme for performing an I/O read and a memory write in the same bus cycle. The DMA memory write bus cycle is shown in Figure 6.18(b).

Let us now look at the DMA memory read bus cycle. The DMA memory read bus cycle is almost identical to a DMA memory write bus cycle. The only difference is that *IOW* is replaced with *IOR* and *MEMW* is replaced with *MEMR*. In addition, the order in which the I/O write and memory read signals are asserted is reversed. The DMA memory read bus cycle is shown in Figure 6.18(c).

6.6 Arbitration

In our earlier discussions, several situations existed in which multiple peripherals might request service from a single resource. For example, multiple peripherals might share a single microprocessor that services their interrupt requests. As another example, multiple peripherals might share a single DMA controller that services their DMA requests. In such situations, two or more peripherals may request service simultaneously. We therefore must have some method to arbitrate among these contending requests. Specifically, we must decide which one of the contending peripherals gets service, and thus which peripherals need to wait. Several methods exist, which we now discuss.

Priority Arbiter

One arbitration method uses a single-purpose processor, called a *priority arbiter*. We illustrate a priority arbiter arbitrating among multiple peripherals using vectored interrupt to request servicing from a microprocessor, as illustrated in Figure 6.19. Each of the peripherals makes its request to the arbiter. The arbiter in turn asserts the microprocessor interrupt, and waits for the interrupt acknowledgment. The arbiter then provides an acknowledgment to exactly one peripheral, which permits that peripheral to put its interrupt vector address on the data bus (which, as you'll recall, causes the microprocessor to jump to a subroutine that services that peripheral).

Priority arbiters typically use one of two common schemes to determine priority among peripherals: fixed priority or rotating priority. In *fixed priority* arbitration, each peripheral has a unique rank among all the peripherals. The rank can be represented as a number, so if there are four peripherals, each peripheral is ranked 1, 2, 3, or 4. If two peripherals simultaneously seek servicing, the arbiter chooses the one with the higher rank.

In *rotating priority* arbitration (also called *round-robin*), the arbiter changes priority of peripherals based on the history of servicing of those peripherals. For example, one rotating priority scheme grants service to the least-recently serviced of the contending peripherals. This scheme obviously requires a more complex arbiter.

We prefer fixed priority when there is a clear difference in priority among peripherals. However, in many cases the peripherals are somewhat equal, so arbitrarily ranking them could cause high-ranked peripherals to get much more servicing than low-ranked ones. Rotating priority ensures a more equitable distribution of servicing in this case.

Notice that the priority arbiter is connected to the system bus, since the microprocessor can configure registers within the arbiter to set the priority schemes and/or the relative priorities of the devices. However, once configured, the arbiter does not use the system bus when arbitrating.

Priority arbiters represent another instance of a standard single-purpose processor. They are also often found built into other single-purpose processors like DMA controllers. A common type of priority arbiter arbitrates interrupt requests; this peripheral is referred to as an *interrupt controller*.

Daisy-Chain Arbitration

The *daisy-chain* arbitration method builds arbitration right into the peripherals. A daisy-chain configuration is shown in Figure 6.20(a), again using vectored interrupt to illustrate the method. Each peripheral has a request output and an acknowledge input, as before. But now each peripheral also has a request input and an acknowledge output. A peripheral asserts its request output if it requires servicing or if its request input is asserted; the latter means that one of the "upstream" devices is requesting servicing. Thus, if any peripheral needs servicing,

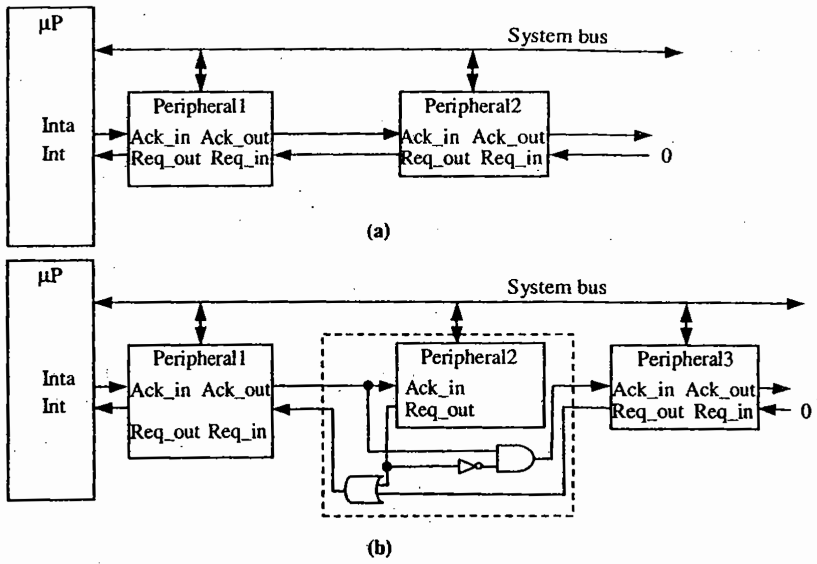


Figure 6.20: Arbitration using a daisy-chain configuration: (a) Daisy-chain aware peripherals, (b) adding logic to make a peripheral daisy-chain aware; more complex logic will typically be necessary, however.

its request will flow through the downstream peripherals and eventually reach the microprocessor. Even if more than one peripheral requests servicing, the microprocessor will see only one request. The microprocessor acknowledge signal connects to the first peripheral. If this peripheral is requesting service, it proceeds to put its interrupt vector address on the system bus. But if it doesn't need service, then it instead passes the acknowledgment upstream to the next peripheral, by asserting its acknowledge output. In the same manner, the next peripheral may either begin being serviced or may instead pass the acknowledgment along. Obviously, the peripheral at the front of the chain, i.e., the one to which the microprocessor acknowledge is connected, has highest priority, and the peripheral at the end of the chain has lowest priority.

We prefer a daisy-chain priority configuration over a priority arbiter when we want to be able to add or remove peripherals from an embedded system without redesigning the system. Although conceptually we could add as many peripherals to a daisy chain as we desired, in reality the servicing response time for peripherals at the end of the chain could become intolerably slow. In contrast to a daisy chain, a priority arbiter has a fixed number of channels; once they are all used, the system needs to be redesigned in order to accommodate more peripherals. However, a daisy chain has the drawback of not supporting more advanced priority schemes, like rotating priority. A second drawback is that if a peripheral in the chain stops working, other peripherals may lose their access to the processor.

Although it appears from Figure 6.20(a) that each peripheral must be daisy-chain aware, in fact logic external to each peripheral can be used to carry out the daisy-chain logic. Figure 6.20(b) illustrates a simple form of such logic. *Peripheral*1 and *Peripheral*3 are both daisy-chain aware, whereas *Peripheral*2 is not. In order to incorporate *Peripheral*2 into the daisy chain configuration, we must extend it to take care of requests and acknowledgments. Regarding requests, if *Peripheral*3 requests service or *Peripheral*2 requests service, then *Peripheral*1's *req_in* needs to be asserted. To accomplish this, we OR *Peripheral*2's *req_out* and *Peripheral*3's *req_out* and input the result to *Peripheral*1. Regarding acknowledgments, if *Peripheral*1's *ack_out* is asserted, then if *Peripheral*2 requested service, it should not pass this acknowledgment to *Peripheral*3, per the daisy-chain protocol. However, if *Peripheral*2 did not request service, then it should pass the acknowledgment to *Peripheral*3. To accomplish this, we use an inverter and an AND gate, as shown in the figure. Only if *Peripheral*1's *ack_out* is high and *Peripheral*2's *req_out* is low do we assert *Peripheral*3's *ack_in*. However, note that this logic is very simple in this case, whereas most peripherals will require more complex logic, even implementing a state machine, to convert the peripheral to a daisy-chain aware device.

Network-Oriented Arbitration Methods

The arbitration methods described are typically used to arbitrate among peripherals in an embedded system. However, many embedded systems contain multiple microprocessors communicating via a shared bus; such a bus is sometimes called a network. Arbitration in such cases is typically built right into the bus protocol, since the bus serves as the only connection among the microprocessors. A key feature of such a connection is that a processor about to write to the bus has no way of knowing whether another processor is about to simultaneously write to the bus. Because of the relatively long wires and high capacitances of such buses, a processor may write many bits of data before those bits appear at another processor. For example, Ethernet and I^{2}C use a method in which multiple processors may write to the bus simultaneously, resulting in a collision and causing any data on the bus to be corrupted. The processors detect this collision, stop transmitting their data, wait for some time, and then try transmitting again. The protocols must ensure that the contending processors don't start sending again at the same time, or must at least use statistical methods that make the chances of them sending again at the same time small.

As another example, the CAN bus uses a clever address encoding scheme such that if two addresses are written simultaneously by different processors using the bus, the higher-priority address will override the lower-priority one. Each processor that is writing the bus also checks the bus, and if the address it is writing does not appear, then that processor realizes that a higher-priority transfer is taking place and so that processor stops writing the bus.

Example: Vectored Interrupt Using an Interrupt table

This is an example of a system using vectored interrupts as well as a vectored interrupt table. We will describe the software programming required to handle the interrupt requests. The relevant portions of the system architecture are shown in Figure 6.21. Here, two peripheral

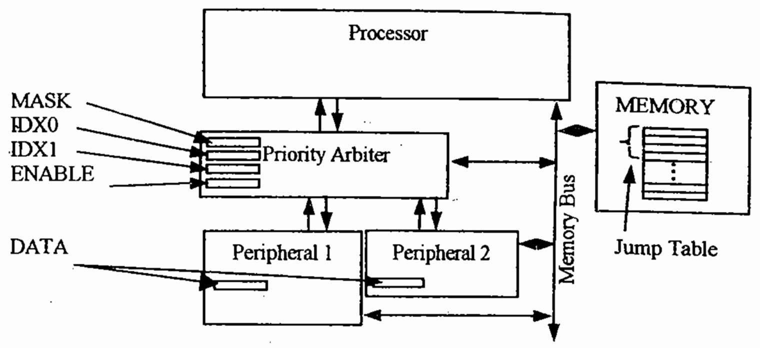


Figure 6.21: Architecture of a system using vectored interrupt and an interrupt table.

devices are connected to a two-channel priority arbiter with fixed priority scheme (i.e., *Peripheral*1 has higher priority than *Peripheral*2). Both the peripherals and the arbiter are connected to the processor's memory bus and communicate with it using memory-mapped I/O. The interrupt table index placed on the memory bus (a.k.a. system bus) by the arbiter is software programmable through two memory-mapped registers. Both peripheral devices receive data from the external environment and raise their interrupt accordingly.

The software to initialize the peripherals and the priority arbiter, and to process the data received by our peripherals, is given in Figure 6.22. Let us now study the code. First, we define a number of variables that correspond to the registers inside the priority arbiter and peripheral devices. However, unlike defining ordinary variables in a program, these variables must refer to specific memory locations, namely, those that are mapped to the peripheral's register. Normally, a compiler will place a variable somewhere in memory where storage for that variable's data is available. By using special keywords, we can force the compiler to place these variables at specific memory locations (e.g., in our compiler the keyword *_at_* followed by a memory location is used to accomplish this). The priority arbiter, thus, has four registers located at memory locations 0xfff0 through 0xfff3. Note that our processor has a 16-bit memory address.

Next, we define two procedures, *Peripheral2_ISR* and *Periperhal2_ISR*, that handle the interrupts generated by the peripherals. Since we are using an interrupt jump table, these ISRs can be ordinary C procedures. Each ISR, of course, must perform necessary processing. Often, an ISR merely reads the data from the peripheral, places the data into a buffer, sets a flag indicating to the main program that the buffer was updated.

Finally, we define the procedure *InitializePeripherals*. The procedure first configures the priority arbiter. We can select, in software, which interrupts we are willing to handle. This is done through the mask register. In our case, we set the first two bits of the mask register, indicating that we are to handle interrupts generated by both peripherals. Next, we program the priority arbiter with the indices into the jump table where the location of the ISR is stored. We have chosen to place these in locations 13 and 17, but this choice is arbitrary. The

```
unsigned char ARBITER_MASK_REG             _at_ 0xfff0;
unsigned char ARBITER_CH0_INDEX_REG        _at_ 0xfff1;
unsigned char ARBITER_CH1_INDEX_REG        _at_ 0xfff2;
unsigned char ARBITER_ENABLE_REG           _at_ 0xfff3;
unsigned char PERIPHERAL1_DATA_REG         _at_ 0xffe0;
unsigned char PERIPHERAL2_DATA_REG         _at_ 0xffe1;
unsigned void* INTERRUPT_LOOKUP_TABLE[256] _at_ 0x0100;
void Peripheral1_ISR(void) {
        unsigned char data;
        data = PERIPHERAL1_DATA_REG;
        // do something with the data
}
void Peripheral2_ISR(void) {
        unsigned char data;
        data = PERIPHERAL2_DATA_REG;
        // do something with the data
}
void InitializePeripherals(void) {
        ARBITER_MASK_REG = 0x03; // enable both channels
        ARBITER_CH0_INDEX_REG = 13;
        ARBITER_CH1_INDEX_REG = 17;
        INTERRUPT_LOOKUP_TABLE[13] = (void*)Peripheral1_ISR;
        INTERRUPT_LOOKUP_TABLE[17] = (void*)Peripheral2_ISR;
        ARBITER_ENABLE_REG = 1;
}
void main() {
        InitializePeripherals();
        for(;;) {} // main program goes here
}
```

"bus" "Gus"

Figure 6.22: Software for a system using vectored interrupt and an interrupt table.

procedure then places the ISRs into the lookup table at locations 13 and 17, as shown in the code. Last, the procedure enables interrupts by setting the arbiter's interrupt enable register.

6.7 Multilevel Bus Architectures

A microprocessor-based embedded system will have numerous types of communications that must take place, varying in their frequencies and speed requirements. The most frequent and high-speed communications will likely be between the microprocessor and its memories. Less frequent communications, requiring less speed, will be between the microprocessor and its peripherals, like a UART. We could try to implement a single high-speed bus for all the communications, but this approach has several disadvantages. First, it requires each peripheral

Offlad offload

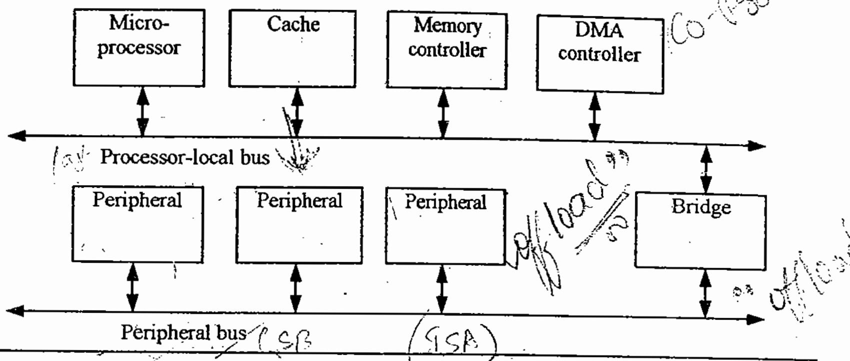


Figure 6.23: A two-level bus architecture.

"off load"

to have a high-speed bus interface. Since a peripheral may not need such high-speed communication, having such an interface may result in extra gates, power consumption and cost. Second, since a high-speed bus will be very processor-specific, a peripheral with an interface to that bus may not be very portable. Third, having too many peripherals on the bus may result in a slower bus. 3

Therefore, we often design systems with two levels of buses: a high-speed processor local bus and a lower-speed peripheral bus, as illustrated in Figure 6.23. The *processor local bus* typically connects the microprocessor, cache, memory controllers, and certain high-speed coprocessors, and is processor specific. It is usually wide, as wide as a memory word.

The *peripheral bus* connects those processors that do not have fast processor local bus access as a top priority, but rather emphasize portability, low power, or low gate count. The peripheral bus is typically an industry standard bus, such as ISA or PCI, thus supporting portability of the peripherals. It is often narrower and/or slower than a processor local bus, thus requiring fewer pins, fewer gates and less power for interfacing. 20 10 10 10

A bridge connects the two buses. A *bridge* is a single-purpose processor that converts communication on one bus to communication on another bus. For example, the microprocessor may generate a read on the processor local bus with an address corresponding to a peripheral. The bridge detects that the address corresponds to a peripheral, and thus it then generates a read on the peripheral bus. After receiving the data, the bridge sends that data to the microprocessor. The microprocessor thus need not even know that a bridge exists — it receives the data, albeit a few cycles later, as if the peripheral were on the processor local bus. 5

A three-level bus hierarchy is also possible, as proposed by the VSI Alliance. The first level is the processor local bus, the second level a system bus, and the third level a peripheral bus. The system bus would be a high-speed bus, but would offload much of the traffic from the processor local bus. It may be beneficial in complex systems with numerous coprocessors. 8

"8" bus pro "8"

6.8 Advanced Communication Principles

In the preceding sections, we discussed basic methods of interfacing. Those interfacing methods could be applied to interconnect components within an IC via on-chip buses, or to interconnect ICs via on-board buses. In the remainder of the chapter, we study more advanced interfacing concepts and look at communication from a more abstract point of view. In particular, we study parallel, serial, and wireless communication. We also describe some advanced concepts, such as layering and error detection, which are part of many communication protocols. Furthermore, we highlight some of the popular parallel, serial, and wireless communication protocols in use today.

Communication can take place over a number of different types of media, such as a single wire, a set of wires, radio waves, or infrared waves. We refer to the medium that is used to carry data from one device to another as the *physical layer*. Depending on the protocol, we may refer to an actor as a *device* or *node*. In either case, a device is simply a processor that uses the physical layer to send or receive data to and from another device.

In this section, we provide a general description of serial communication, parallel communication, and wireless communication. In addition, we describe communication principles such as layering, error detection and correction, data security, and plug and play.

Parallel Communication

Parallel communication takes place when the physical layer is capable of carrying multiple bits of data from one device to another. This means that the data bus is composed of multiple data wires, in addition to control and possibly power wires, running in parallel from one device to another. Each wire carries one of the bits. Parallel communication has the advantage of high data throughput, if the length of the bus is short. The length of a parallel bus must be kept short because long parallel wires will result in high capacitance values, and transmitting a bit on a bus with a higher capacitance value will require more time to charge or discharge. In addition, small variations in the length of the individual wires of a parallel bus can cause the received bits of the data word to arrive at different times. Such misalignment of data becomes more of a problem as the length of a parallel bus increases. Another problem with parallel buses is the fact that they are more costly to construct and may be bulky, especially when considering the insulation that must be used to prevent the noise from each wire from interfering with the other wires. For example, a 32-wire cable connecting two devices together will cost much more and be larger than a two-wire cable.

In general, parallel communication is used when connecting devices that reside on the same IC, or devices that reside on the same circuit board. Since the length of such buses is short, the capacitance load, data misalignment and cost problems mentioned earlier do not play an important role.

Serial Communication

Serial communication involves a physical layer that carries one bit of data at a time. This means that the data bus is composed of a single data wire, along with control and possibly power wires, running from one device to another. In serial communication, a word of data is transmitted one bit at a time. Serial buses are capable of higher throughputs than parallel buses when used to connect two physically distant devices. The reason for this is that a serial bus will have less average capacitance, enabling it to send more bits per unit of time. In addition, a serial bus cable is cheaper to build because it has fewer wires. The disadvantage of a serial bus is that the interfacing logic and communication protocol will be more complex. On the sending side, a transmitter must decompose data words into bits and on the receiving side, and the receiver must compose bits into words.

Most serial bus protocols eliminate the need for extra control signals, such as read and write signals, by using the same wire that carries data for this purpose. This is performed as follows. When data is to be sent, the sender first transmits a bit called a *start bit*. A start bit merely signals the receiver to wakeup and start receiving data. The start bit is then followed by N data bits, where N is the size of the word, and a *stop bit*. The stop bit signals to the receiver the end of the transmission. Often, both the transmitter and the receiver agree on the transmission speed used to send and receive data. After sending a start bit, the transmitter sends all N bits at the predetermined transmission speed. Likewise, on seeing a start bit, a receiver simply starts sampling the data at a predetermined frequency until all N bits are assembled. Another common synchronization technique is to use an additional wire for clocking purposes (see the I^2C bus protocol). Here, the transmitter and receiver devices use this clock line to determine when to send or sample the data.

Wireless Communication

Wireless communication eliminates the need for devices to be physically connected in order to communicate. The physical layer used in wireless communication is typically either an infrared (IR) channel or a radio frequency (RF) channel.

Infrared uses electromagnetic wave frequencies that are just below the visible light spectrum, thus undetectable by the human eye. These waves can be generated by using an infrared diode and detected by using an infrared transistor. An infrared diode is similar to a red or green diode except that it emits infrared light. An infrared transistor is a transistor that conducts (i.e., allows current to flow from its source to its drain), when exposed to infrared light. A simple transmitter can send 1s by turning on its infrared diode and can send 0s by turning off its infrared diode. Correspondingly, a receiver will detect 1s when current flows through its infrared transistor and 0s otherwise. The advantage of using infrared communication is that it is relatively cheap to build transmitters and receivers. The disadvantage of using infrared is the need for line of sight between the transmitter and receiver, resulting in a very restricted communication range.

Radio frequency (RF) uses electromagnetic wave frequencies in the radio spectrum. A transmitter here will need to use analog circuitry and an antenna to transmit data. Likewise, a receiver will need to use an antenna and analog circuitry to receive data. One advantage of using RF is that, generally, a line of sight is not necessary and thus longer distance communication is possible. The range of communication is, of course, dependent on the transmission power used by the transmitter.

Typically, RF transmitters and receivers must agree on a specific frequency in order to send and receive data. Using *frequency hopping*, it is possible for the transmitter and receiver to communicate while constantly changing the transmission frequency. Of course, both devices must have a common understanding of the sequence for frequency hops. Frequency hopping allows more devices to share a fixed set of frequencies and is commonly used in wireless communication protocols designed for networks of computers and other electronic devices.

Layering

Layering is a hierarchical organization of a communication protocol where lower levels of the protocol provide services to the higher levels. We have already discussed the physical layer. The physical layer provides the basic service of sending and receiving bits or words of data. The next higher-level protocol uses the physical layer to send and receive packets of data, where a packet of data is composed of possibly multiple bytes. The next higher level uses the packet transmission service of its lower level to perhaps send different type of data such as acknowledgments, special requests, and so on. Typically, the lowest level consists of the physical layer and the highest level consists of the application layer. The application layer provides abstract services to the application such as ftp or http.

Layering is a way to break the complexity of a communication protocol into independent pieces, thus making it easier to design and understand, much like a programmer abstracting away complexities of a program by creating objects or libraries of routines. In communication and networking, the concept of layering is very fundamental.

Error Detection and Correction

Error detection is the ability of a receiver to detect errors that may occur during the transmission of a data word or packet. The most common types of errors are bit errors and burst of bit errors. A *bit error* occurs when a single bit in a word or data packet is received as its inverted value. A *burst of bit error* occurs when consecutive bits of a word or data packet are received incorrectly. Given that an error is detected, *error correction* is the ability of a receiver and transmitter to cooperate in order to correct the problem. The ability to detect and correct errors is often part of a bus protocol. We will next discuss parity and checksum error detection algorithms, which are commonly used in bus protocols.

Parity is a single bit of information that is sent along with a word of data by the transmitter to give the receiver some additional knowledge about the data word. This additional knowledge is used by the receiver to detect, to some degree, a bit or burst of bit error in receiving a word. Common types of parity are odd or even. Odd parity is a bit that if set indicates to the receiver that the data word bits plus parity bit contain an odd number of 1s. Even parity is a bit that if set indicates to the receiver that the data word bits plus parity bit contain an even number of 1s. Prior to sending a word of data, the transmitter will compute the parity and send that along with the data word to the receiver. On reception of the data word and parity bit, the receiver will compute the parity of the data and make sure that it agrees with the parity bit received from the transmitter. If a parity check fails, it indicates with

certainty that there was at least one transmission error. Parity checks will always detect a single bit error. However, burst bit errors may or may not be detected by parity checking — an even number of errors, for example, will not be detected.

As an example of parity-based error checking, consider wanting to transmit the following 7-bit word: 0101010. Assuming even parity, we would actually transmit the 8-bit word: 01010101, where the least-significant bit is the parity bit. Now, suppose during transmission, a bit gets flipped, so that a receiver receives the following 8-bit word: 11010101. The receiver detects that this word has odd parity; knowing that the word was supposed to have even parity, the receiver determines that this word has an error. Instead, suppose the receiver receives: 11110101. This word has even parity, and so the receiver thinks the word is correct, even though it contains two errors.

Checksum is a stronger form of error checking that is applied to a packet of data. A packet of data will contain multiple words of data. Using parity checking, we used one extra bit per word to help us detect errors. Using checksum, we use an extra word per packet for the same purpose. For example, we may compute the XOR sum of all the data words in a packet and send this value along with the data packet. Upon receiving the data packet words and the checksum word, the receiver will compute the XOR sum of all the data words it received. If the computed checksum word equals the received checksum word, the data packet is assumed to be correct. Otherwise, it is assumed to be incorrect. Again, not all error combinations can be detected. We can of course use both parity and checksum for stronger error checking.

As an example, suppose a packet consists of four words: 0000000, 0101010, 1010101, and 0000000. The XOR checksum of these four words is 1111111. A transmitter can thus send that checksum word at the end of the packet. Now, suppose the receiver receives 1000001, 0101010, 1010101, and 0000000. Note that two bits have switched in the first word, and that parity-based error checking would not detect this error. The receiver computes the checksum of this packet and obtains 0111110. This differs from the received checksum of 1111111, and thus the receiver determines that an error has occurred.

Note that errors can also occur in the parity bit or the checksum word itself.

When using parity or checksum error detection, error correction is typically done by a retransmission and acknowledgment protocol. Here, the transmitter sends a data packet and expects to receive an acknowledgment from the receiver indicating that the data packet was received correctly. If an acknowledgment is not received, the transmitter retransmits the data packet and waits for a second acknowledgment.

I^2C Inter-IC I^2C

6.9 Serial Protocols

inter-IC

In this section we describe four popular serial protocols, namely the I^2C protocol, the CAN protocol, the FireWire protocol, and the USB protocol.

I^2C

Philips Semiconductors developed the *Inter-IC*, or I^2C, bus nearly 20 years ago. I^2C is a two-wire serial bus protocol. This protocol enables peripheral ICs in electronic systems to

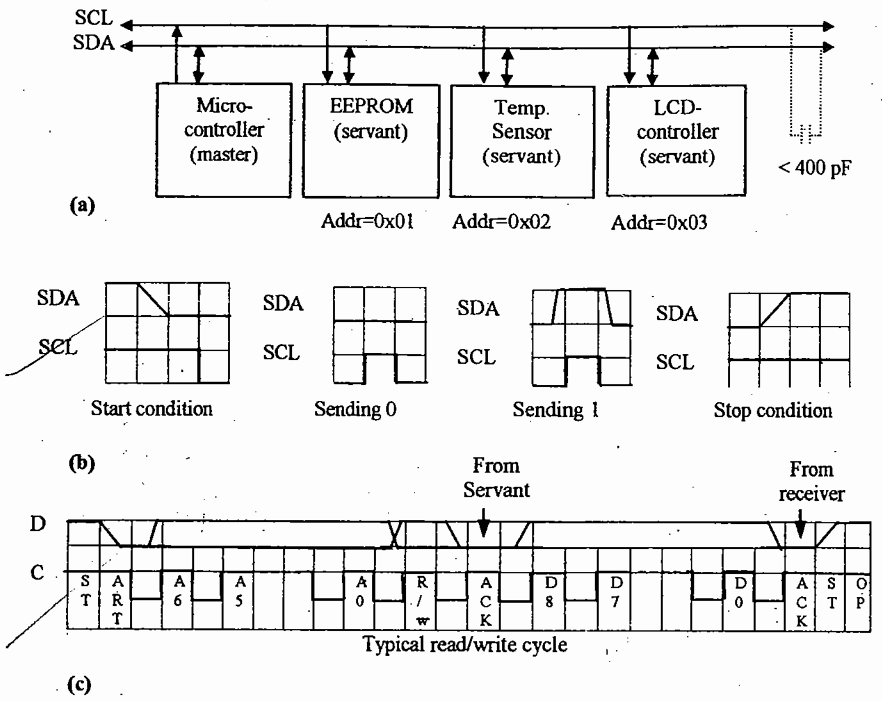


Figure 6.24: I^2C bus structure.

communicate with each other using simple communication hardware. Based on the original specification of the I^2C, data transfer rates of up to 100 kbits/s and 7-bit addressing are possible. Seven-bit addressing allows a total of 128 devices to communicate over a shared I^2C bus. With increased data transfer rate requirements, the I^2C specification has been recently enhanced to include fast-mode, 3.4 Mbits/s, with 10-bit addressing. Common devices capable of interfacing to an I^2C bus include EPROMs, Flash and some RAM memory devices, real-time clocks, watchdog timers, and microcontrollers.

A sample I^2C network is depicted in Figure 6.24(a). The bus consists of two wires; a data wire called serial-data-line (*SDA*) and a clock wire called serial-clock-line (*SCL*). The I^2C specification does not limit the length of the bus wires, as long as the total capacitance of the bus remains under 400 pF. In this example, there are four devices attached to the bus. One of these devices, the microcontroller, is a master. The other three devices, a temperature sensor, an EEPROM, and a LCD-controller, are servants. Each of these servant devices is assigned a unique address, as shown in Figure 6.24(a). Only master devices can initiate a data transfer on

an I^2C bus. The protocol does not limit the number of master devices on an I^2C bus, but typically, in a microcontroller-based system, the microcontroller serves as the master. Both master and servant devices can be senders or receivers of data. This will depend on the function of the device. In our example, the microcontroller and EEPROM send and receive data, while the temperature sensor sends data and the LCD-controller receives data. In Figure 6.24(a), arrows connecting the devices to the I^2C bus wires depict the data movement direction. Normally, all servant devices residing on an I^2C assert high-impedance on the bus while the master device maintains logic high, signaling an idle condition.

All data transfers on an I^2C bus are initiated by a start condition. A start condition is shown in Figure 6.24(b). A high to low transition of the *SDA* line while the *SCL* signal is held high signals a start condition. All data transfers on an I^2C bus are terminated by a stop condition. A stop condition is shown in Figure 6.24(b). A low to high transition of the *SDA* line while the *SCL* signal is held high signals a stop condition. Actual data is transferred in between start and stop conditions. A typical I^2C byte write cycle works as follows. The master device initiates the transfer by a start condition. Then, the address of the device that the byte is being written to is sent starting with the most significant down to the least significant bit. Ones and zeros are sent as shown in Figure 6.24(b). Here, the bit value is placed on the *SDA* line by the master device while the *SCL* line is low and maintained stable until after a clock pulse on *SCL*. If performing a write, right after sending the address of the receiving device, the master sends a zero. The receiving device in return acknowledges the transmission by holding the *SDA* line low during the first *ACK* clock cycle. Following the acknowledgment, the master device transmits a byte of data starting with the most significant down to the least significant bit. The receiving device, in this case the servant, acknowledges the reception of data by holding the *SDA* line low during the second *ACK* clock cycle. If performing a read operation, the master initiates the transfer by a start condition, sends the address of the device that is being read, sends a one (logic high on *SDA* line) requesting a read and waits to receive an acknowledgment. Then, the sender sends a byte of data. The receiver, master device in this case, acknowledges the reception of data and terminates the transfer by generating a stop condition. The timing diagram of a typical read/write cycle is depicted in Figure 6.24(c).

CAN

40, 50 11-bit High integrity SDC

The *controller area network* (CAN) bus is a serial communication protocol for real-time applications, possibly carried over a twisted pair of wires. This protocol was developed by Robert Bosch GmbH ~~to enable communication among various electronic components of cars as an alternative to expensive and cumbersome wiring harnesses. The robustness of the protocol has expanded its use to many other automation and industrial applications.~~ Some characteristics of the CAN protocol include high-integrity serial data communications, real-time support, data rates of up to 1 Mbit/s, 11-bit addressing, error detection, and confinement capabilities. ~~The CAN protocol is documented in ISO 11898 (for high-speed applications) and ISO 11519-2 (for lower speed applications).~~ Common applications, other than automobiles, using CAN include elevator controllers, copiers, telescopes, production-line control systems, and medical instruments. Among devices that incorporate a CAN interface

RTD

90

20

are the 8051-compatible 8592 processor, and a variety of standalone CAN controllers, such as the 80C200, from Philips.

The CAN specification does not specify the actual layout and structure of the physical bus itself. Instead, it requires that a device connected to the CAN bus is able to transmit, or detect, on the physical bus, one of two signals called dominant or recessive. For example, a dominant signal may be represented as logic '0' and recessive as logic '1' on a single data wire. Furthermore, the physical CAN bus must guarantee that if one of two devices asserts a dominant signal and another device simultaneously a recessive signal, the dominant signal prevails. Given a physical CAN bus with the above-mentioned properties, the protocol defines data packet format and transmission rules to prioritize messages, guarantee latency times, allow for multiple masters, handles transmission errors, retransmit corrupted messages, and distinguish between a permanent failure of a node versus temporary errors.

FireWire

The *FireWire* (a.k.a. *I-Link* or *Lynx*) is a high-performance serial bus developed by Apple Computer Inc. Because the specification of the FireWire protocol is given by the 1394 IEEE designation, many refer to it as the IEEE 1394, or simply 1394. The need for FireWire is driven by the rapidly growing need for mass information transfer. Typical local or wide area networks (LANs/WANs) are incapable of providing cost-effective connection capabilities and do not guarantee bandwidth for real-time applications. Some characteristics of the FireWire protocol include transfer rates of 12.5 to 400 Mbits/s, 64-bit addressing, real-time connection/disconnect and address assignment (a.k.a., plug-and-play capabilities), and packet-based layered design structure.

While I^2C and CAN bus protocols are designed mostly for interfacing ICs, FireWire is designed for interfacing among independent electronic devices (e.g., a desktop computer and a digital scanner). Moreover, FireWire is capable of supporting an entire local-area network similar to one based on Ethernet. The 64-bit wide address space of FireWire is partitioned as 10 bits for network identifiers, 6 bits for node identifiers, and 48 bits for memory address. A local-area network based on FireWire can consist of 1,023 subnetworks, each consisting of 63 nodes, with each node, in turn, addressable by 281 terabytes of distinct locations! FireWire is feasible for applications such as disk drives, printers, scanners, cameras, and many other consumer electronics devices.

USB

The *Universal Serial Bus* (USB) protocol is designed to make it easier for PC users to connect monitors, printers, digital speakers, modems and input devices like scanners, digital cameras, joysticks, and multimedia game equipment. USB has two data rates, 12 Mbps for devices requiring increased bandwidth, and 1.5 Mbps for lower-speed devices like joysticks and game pads. USB uses a tiered star topology, which means that some USB devices, called USB hubs, can serve as connection ports for other USB peripherals. Only one device needs to be plugged into the PC. Other devices can then be plugged into the hub. USB hubs may be embedded in such devices as monitors, printers and keyboards. Standalone hubs could also be made

available, providing a handful of convenient USB ports right on the desktop. Hubs feature an upstream connection (pointed toward the PC) as well as multiple downstream ports to allow the connection of additional peripheral devices. Up to 127 USB devices can be connected together in this way.

USB host controllers manage and control the driver software and bandwidth required by each peripheral connected to the bus. Users don't need to do a thing, because all the configuration steps happen automatically. The USB host controller even allocates electrical power to the USB devices. Like USB host controllers, USB hubs can detect attachments and detachments of peripherals occurring downstream and supply appropriate levels of power to downstream devices. Since power is distributed through USB cables, with a maximum length of 5 meters, you no longer need a clunky AC power supply box for many devices.

6.10 Parallel Protocols

Peripheral Component Inter

In this section, we briefly describe two popular parallel protocols, namely the PCI bus protocol and the ARM Bus protocol.

PCI Bus

The *Peripheral Component Interconnect* (PCI) bus is a high-performance bus for interconnecting chips, expansion boards (e.g., a video card that plugs into a main board like a Pentium mother board), and processor memory subsystems. The PCI bus originated at Intel in the early 1990s, was then adopted by the industry as a standard and administered by the PCI Special Interest Group (PCISIG), and was first used in personal computers in 1994 along with Intel 486 processors. The PCI bus has since largely replaced the earlier bus architectures such as the ISA/EISA bus described earlier, and Micro Channel bus protocols. Some characteristics of the PCI bus protocol include transfer rates of 127.2 to 508.6 Mbits/s, 32-bit addressing, synchronous bus architecture (i.e., all transfers take place with respect to a clock signal), and multiplexed 32-bit data/address lines. It must be noted that later additions to the specification of the PCI bus extend the protocol to allow 64-bit data and addressing while maintaining compatibility with the 32-bit schemes.

t6 32-b (SPA)

ARM Bus

While PCI is a widely used industry standards, many other bus protocols are predominantly designed and used internally by various IC design companies. One such bus is the ARM bus designed by the ARM Corporation and documented in ARM's application note 18v. This bus is designed to interface with the ARM line of processors. The ARM bus supports 32-bit data transfer and 32-bit addressing and, similar to PCI, is implemented using synchronous data transfer architecture. The transfer rate on an ARM bus is not specified and instead is a function of the clock speed used in a particular application. More specifically, if the clock speed of the ARM bus is denoted as X, then the transfer rate is $16 \times X$ bits/s.

Y 16xX bits

6.11 Wireless Protocols

In this section, we briefly introduce three new and emerging wireless protocols, namely IrDA, Bluetooth, and the IEEE 802.11.

IrDA

The Infrared Data Association (IrDA) is an international organization that creates and promotes interoperable, low-cost, infrared data interconnection standards that support a walk-up, point-to-point user model. Their protocol suite, also commonly referred to as IrDA, is designed to support transmission of data between two devices over short-range point-to-point infrared at speeds between 9.6 kbps and 4 Mbps. IrDA is that small, semitransparent, red window that you may have wondered about on your notebook computer. Over the last several years, IrDA hardware has been deployed in notebook computers, printers, personal digital assistants, digital cameras, public phones, and even cell phones. One of the reasons for this has been the simplicity and low cost of IrDA hardware. Unfortunately, until recently, the hardware has not been available for applications programmers to use because of a lack of suitable protocol drivers.

Microsoft Windows CE 1.0 was the first Windows operating system to provide built-in IrDA support. Windows 2000 and Windows 98 now also include support for the same IrDA programming APIs that have enabled file sharing applications and games on Windows CE. IrDA implementations are becoming available on several popular embedded operating systems.

Bluetooth

Bluetooth is a new and global standard for wireless connectivity. This protocol is based on a low-cost, short-range radio link. The radio frequency used by Bluetooth is globally available. When two Bluetooth-equipped devices come within 10 meters of each other, they can establish a connection. Because Bluetooth uses a radio-based link, it doesn't require a line-of-sight connection in order to communicate. For example, your laptop could send information to a printer in the next room, or your microwave oven could send a message to your cordless phone telling you that your meal is ready. In the future, Bluetooth is likely to be standard in tens of millions of mobile phones, PCs, laptops and a whole range of other electronic devices.

IEEE 802.11

IEEE 802.11 is an IEEE-proposed standard for wireless local area networks (LANs). There are two different ways to configure a network: ad-hoc and infrastructure. In the ad-hoc network, computers are brought together to form a network on the fly. Here, there is no structure to the network, there are no fixed points, and usually every node is able to communicate with every other node. Although it seems that order would be difficult to maintain in this type of network, special algorithms have been designed to elect one machine as the master station of the network with the others being servants. Another algorithm in ad-

hoc network architectures uses a broadcast and flooding method to all other nodes to establish who's who. The second type of network structure used in wireless LANs is the infrastructure. This architecture uses fixed network access points with which mobile nodes can communicate. These network access points are sometime connected to landlines to widen the LAN's capability by bridging wireless nodes to other wired nodes. If service areas overlap, handoffs can occur. This structure is very similar to the present day cellular networks around the world.

The IEEE 802.11 protocol places specifications on the parameters of both the physical PHY and medium access control MAC layers of the network. The PHY layer, which actually handles the transmission of data between nodes, can use direct sequence spread spectrum, frequency-hopping spread spectrum, or infrared pulse position modulation. IEEE 802.11 makes provisions for data rates of either 1 Mbps or 2 Mbps, and calls for operation in the 2.4 to 2.4835 GHz frequency band, which is an unlicensed band for industrial, scientific, and medical applications, and 300 to 428,000 GHz for IR transmission. Infrared is generally considered to be more secure to eavesdropping, because IR transmissions require absolute line-of-sight links (no transmission is possible outside any simply connected space or around corners), as opposed to radio frequency transmissions, which can penetrate walls and be intercepted by third parties unknowingly. However, infrared transmissions can be adversely affected by sunlight, and the spread-spectrum protocol of IEEE 802.11 does provide some rudimentary security for typical data transfers.

The MAC layer is a set of protocols, which is responsible for maintaining order in the use of a shared medium. The IEEE 802.11 standard specifies a carrier sense multiple access with collision avoidance CSMA/CA protocol. In this protocol, when a node receives a packet to be transmitted, it first listens to ensure no other node is transmitting. If the channel is clear, it then transmits the packet. Otherwise, it chooses a random backoff-factor, which determines the amount of time the node must wait, until it is allowed to transmit its packet. During periods in which the channel is clear, the transmitting node decrements its backoff counter. When the backoff counter reaches zero, the node transmits the packet. Since the probability that two nodes will choose the same backoff factor is small, collisions between packets are minimized. Collision detection, as is employed in Ethernet, cannot be used for the radio frequency transmissions of IEEE 802.11. The reason for this is that when a node is transmitting it cannot hear any other node in the system, which may be transmitting, since its own signal will drown out any others arriving at the node.

Whenever a packet is to be transmitted, the transmitting node first sends out a short ready-to-send RTS packet containing information on the length of the packet. If the receiving node hears the RTS, it responds with a short clear-to-send CTS packet. After this exchange, the transmitting node sends its packet. When the packet is received successfully, as determined by a cyclic redundancy check, the receiving node transmits an acknowledgment ACK packet.

6.12 Summary

Interfacing processors and memory represents a challenging design task. Timing diagrams provide a basic means for us to describe interface protocols. Thousands of protocols exist, but they can be better understood by understanding basic protocol concepts like actors, data direction, addresses, time multiplexing, and control methods. A general-purpose processor typically has either a bus-based I/O structure or a port-based I/O structure for interfacing. Interfacing with a general-purpose processor is the most common interfacing task and involves three key concepts. The first is the processor's approach for addressing external data locations, known as its I/O addressing approach, which may be memory-mapped I/O or standard I/O. The second is the processor's approach for handling requests for servicing by peripherals, known as its interrupt handling approach, which may be fixed or vectored. The third is the ability of peripherals to directly access memory, known as direct memory access. Interfacing also leads to the common problem of more than one processor simultaneously seeking access to a shared resource such as a bus, requiring arbitration. Arbitration may be carried out using a priority arbiter or using daisy chain arbitration. A system often has a hierarchy of buses, such as a high-speed processor local bus and a lower-speed peripheral bus. Communication protocols may carry out parallel or serial communication, and may use wires, infrared, or radio frequencies as the transmission medium. Communication protocols may include extra bits for error detection and correction, and typically involve layering as an abstraction mechanism. Popular serial protocols include I^2C, CAN, FireWire, and USB. Popular parallel protocols include PCI and ARM. Popular serial wireless protocols include IrDA, Bluetooth, and IEEE 802.11.

6.13 References and Further Reading

- VSI Alliance. On-Chip Bus Development Working Group, Specification 1 version 1.0, "On-Chip Bus Attributes," August 1998, http://www.vsi.org.
- L. Eggebrecht. *Interfacing to the IBM Personal Computer.* Indianapolis, IN: SAMS, Macmillan Computer Publishing, 1990.
- Peter W. Gofton. *Mastering Serial Communications.* Alameda, CA: SYBEX Inc., 1994.
- Bob O'Hara and Al Petrick. *IEEE 802.11 Handbook — A Designer's Companion.* Piscataway, NJ: Standards Information Network, IEEE Press, 1999.
- John Hyde. *USB Design by Example.* New York: John Wiley & Sons, Inc., 1999.

6.14 Exercises

6.1 Draw the timing diagram for a bus protocol that is handshaked, nonaddressed, and transfers 8 bits of data over a 4-bit data bus.

6.2 Explain the difference between port-based I/O and bus-based I/O.

6.3 Show how to extend the number of ports on a 4-port 8051 to 8 by using extended parallel I/O. (a) Using block diagrams for the 8051 and the extended parallel I/O device, draw and label all interconnections and I/O ports. Clearly indicate the names and widths of all connections. (b) Give C code for a function that could be used to write to the extended ports.

6.4 Discuss the advantages and disadvantages of using memory-mapped I/O versus standard I/O.

6.5 Explain the benefits that an interrupt address table has over fixed and vectored interrupt methods.

6.6 Draw a block diagram of a processor, memory, and peripheral connected with a system bus, in which the peripheral gets serviced using vectored interrupt. Assume servicing moves data from the peripheral to the memory. Show all relevant control and data lines of the bus, and label component inputs/outputs clearly. Use symbolic values for addresses. Provide a timing diagram illustrating what happens over the system bus during the interrupt.

6.7 Draw a block diagram of a processor, memory, peripheral, and DMA controller connected with a system bus, in which the peripheral transfers 100 bytes of data to the memory using DMA. Show all relevant control and data lines of the bus, and label component inputs/outputs clearly. Draw a timing diagram showing what happens during the transfer; skip the 2nd through 99th bytes.

6.8 Repeat problem 6.7 for a daisy-chain configuration.

6.9 Design a parallel I/O peripheral for the ISA bus. Provide: (a) a state-machine description and (b) a structural description.

6.10 Design an extended parallel I/O peripheral. Provide: (a) a state-machine description and (b) a structural description.

6.11 List the three main transmission mediums described in the chapter. Give two common applications for each.

6.12 Assume an 8051 is used as a master device on an I2C bus with pin P1.0 corresponding to I2C_Data and pin P1.1 corresponding to I2C_Clock. Write a set of C routines that encapsulate the details of the I2C protocol. Specifically, write the routines called StartI2C/StopI2C, that send the appropriate start/stop signal to slave devices. Likewise, write the routines ReadByte and WriteByte, each taking a device Id as input and performing the appropriate I/O actions.

6.13 Select one of the following serial bus protocols, then, perform an internet search for information on transfer rate, addressing, error correction (if applicable), and plug-and-play capability (if applicable). Then give timing diagrams for a typical transfer of data (e.g., a write operation). The protocols are USB, I2O, Fibre Channel, SMBus, IrDA, or any other serial bus in use by the industry and not described in this book.