Chapter 6 *Interfacing*

6.1 Introduction

As stated in the previous chapter, we use *processors* to implement processing, *memories* to implement storage, and *buses* to implement communication. The earlier chapters described processors and memories. This chapter describes implementing communication with buses, i.e., interfacing.

Buses implement communication among processors or among processors and memories. Communication is the transfer of data among those components. 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.

A bus consists of wires connecting two or more processors or memories. Figure 6.1(a) shows the wires of a simple bus connecting a processor with a memory. Note that each wire may be uni-directional, as are *rd/wr*, *enable*, and *addr*, or bi-directional, as is *data*. Also note that a set of wires with the same function is typically drawn as a thick line (or a line with a small angled line drawn through it). *addr* and *data* each represent a set of wires; the *addr* wires transmit an address, while the *data* wires transmit data. The bus connects to "pins" of a processor (or memory). A pin is the actual conducting device (i.e., metal) on the periphery of a processor through which a signal is input to or output from the processor. When a processor is packaged as its own IC, there are actual pins extending from the package, designed to be plugged into a socket on a printed-circuit board. Today, however, a processor commonly co-exists 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 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. For consistency, though, we shall use the term pin in this chapter regardless of the packaging situation.

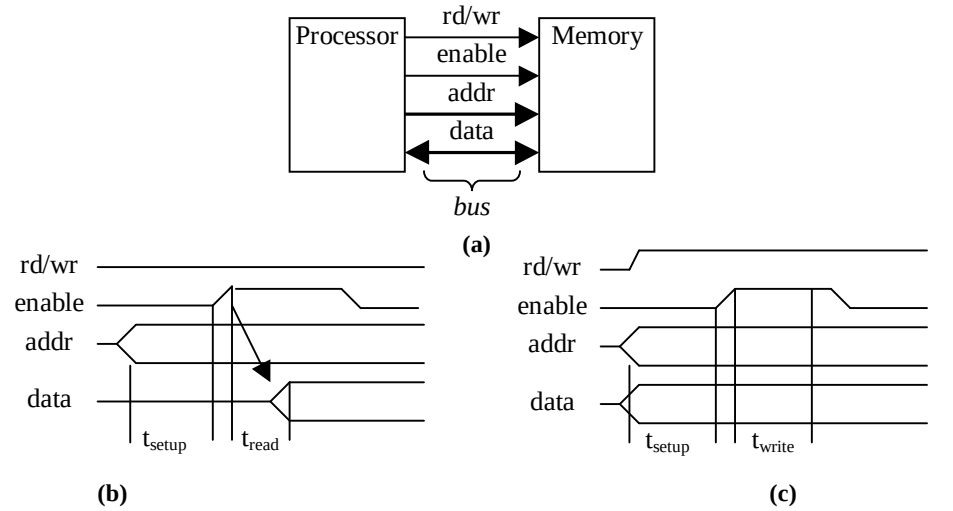
A bus must have an associated *protocol* describing the rules for transferring data over those wires. We deal primarily with low-level hardware protocols in this chapter, while higher-level protocols, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.

Interfacing with a general-purpose processor is extremely common. We describe three issues relating to such interfacing: addressing, interrupts, and direct memory access.

When multiple processors attempt to access a single bus or memory simultaneously, resource contention exists. This chapter therefore describes several schemes for arbitrating among the contending processors.

6.2 Timing diagrams

Figure 6.1: Timing diagrams: (a) bus structure, (b) read, (c) write.



The most common method for describing a hardware protocol is a timing diagram. Consider the example processor-memory bus of Figure 6.1(a). Figure 6.1(b) uses a timing diagram to describe the protocol for reading the memory over the bus. 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*, as being either invalid (a single horizontal line) or valid (two horizontal lines); the value of data lines is not normally relevant when describing a protocol.

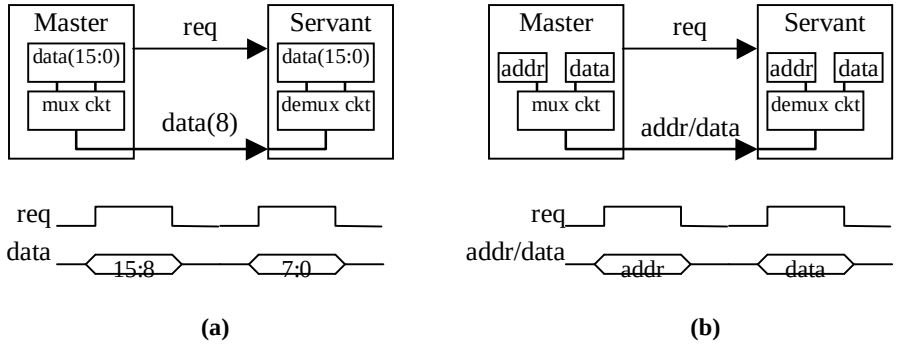
In the above protocol, the control line *enable* is active high, meaning that a 1 on the enable line triggers the data transfer. In many protocols, control lines are instead active low, meaning that a 0 on the line triggers the transfer. Such a control line is typically written with a bar above it, a single quote after it (e.g., *enable'*), or an underscore 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 (i.e., to 1 for an active high line, to 0 for an active low line), and the term "deassert" to mean setting the control line to its inactive value.

6.3 Hardware protocol basics

6.3.1 Concepts

The 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

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



concepts. These concepts include: actors, data direction, addresses, time-multiplexing, and control methods.

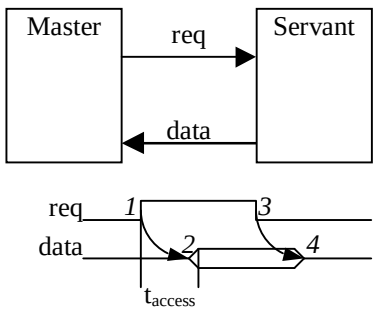
An *actor* is a processor or memory involved in the data transfer. A protocol typically involves two actors: a master and a servant. A *master* initiates the data transfer. A *servant* (usually called a slave) responds to the initiation request. In the example of Figure 6.1, the processor is the master 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, while 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 in 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 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.

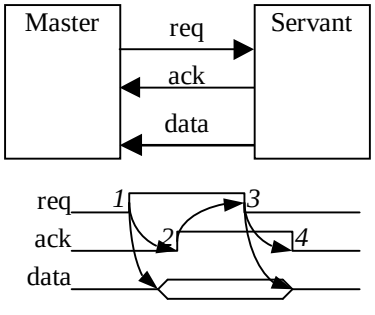
Another protocol concept is *time multiplexing*. To multiplex means to share a single set of wires for multiple pieces of data. In time multiplexing, the multiple pieces of data are sent one at a time over the shared wires. For example, Figure 6.2(a) shows a master sending 16 bits of data over an 8-bit bus using a strobe protocol and time-multiplexed data. The master first sends the high-order byte, 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 (probably a memory). In this case, rather than using separate sets of lines for

Figure 6.3: Two protocol control methods: (a) strobe, (b) handshake.



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

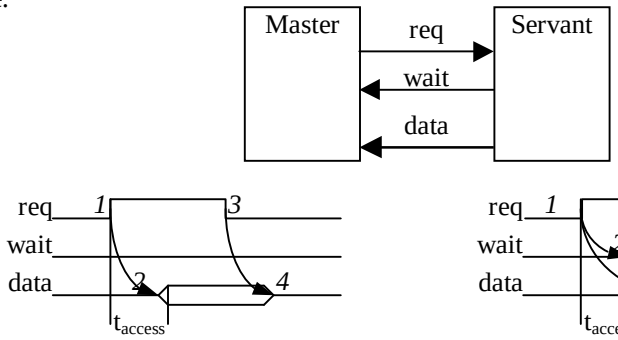
(b)

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 receiving master. The master first asserts the request line to initiate the transfer. The servant takes however much time is 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

Figure 6.4: A strobe/handshake compromise: (a) fast-response case, (b) slow-response case.



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 request
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 finally puts data and ***deasserts wait***
4. Master receives data and deasserts request
5. Servant ready for next request

(b)

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 acknowledgement 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 the 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."

Example: A simple bus protocol

```
A protocol for a simple bus (ISA) will be described.
```

6.4 Interfacing with a general-purpose processor

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 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. We will describe three processor I/O issues: addressing, interrupts, and direct memory access. We'll use the term microprocessor in this section to refer to a general-purpose processor.

6.4.1 I/O addressing

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: ports, and system buses.

A *port* is a set of pins that can be read and written just like any register in the microprocessor; in fact, the port is usually connected to a dedicated register. For example, consider an 8-bit port named P0. A C-language programmer may write to P0 using an instruction like: P0 = 255, which would set all 8 pins to 1's. In this case, the C compiler manual would have defined P0 as a special variable that would automatically be mapped to the register P0 during compilation. Conversely, the programmer might read the value of a port P1 being written by some other device, by saying something like a=P1. 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, P0 might have an associated configuration register called CP0. To set the high-order four bits to input and the low-order four bits to output, we might say: CP0 = 15. This writes 00001111 to the CP0 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 = P0.2, giving x the value of the number 2 connection of port P0. Port-based I/O is also called *parallel I/O*.

In contrast to a port, a *system bus* is a set of pins consisting of address pins, data pins, and control pins (for strobing or handshaking). The microprocessor uses the bus to access memory as well as peripherals. 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 microprocessor may use one of two methods for communication over a system bus: standard I/O or memory-mapped 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.

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.

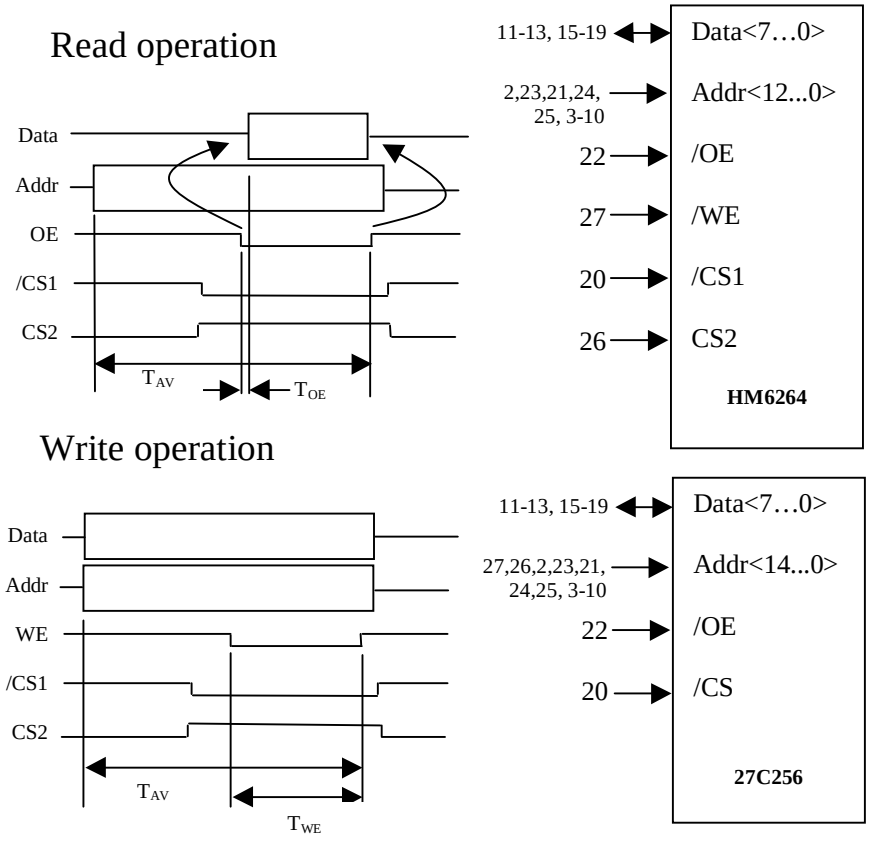
Example: HM6264 and 27C256 RAM/ROM memory devices

In this example, we introduce a pair of low-cost low-capacity memory devices commonly used in 8-bit micro-controller based embedded systems. The first two numeric digits in these devices indicate weather the device is *random-access memory* (RAM), 62, or *read-only memory* (ROM), 27. Subsequent digits give the memory capacity in K bits. Both these devices are available in 4, 8, 16, 32, and 64K bytes, i.e., part numbers 62/27 followed by 32, 64, 128, 256, or 512. The following table summarizes some of the characteristics of these devices.

Device	Access Time (ns)	Standby Pwr. (mw)	Active Pwr. (mw)	Vcc Voltage (V)
HM6264	85-100	.01	15	5
27C256	90	.5	100	5

Source Hitachi HM6264B and Microchip 27C256 datasheets.

Memory access to and from these devices is performed through an 8-bit parallel protocol. Placing a memory address on the address-bus and asserting the read signal *output-enable* (OE) performs a read operation. Placing some data and a memory address on the data/address busses and asserting the write signal *write-enable* (WE) performs a write operation. Read and write timing diagrams are presented here.



In the next example, we demonstrate interfacing of these devices with an Intel 8051 micro-controller.

*T_{WE} =40ns, T_{OE} =75, T_{AV} =40-148

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 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.

Situations often arise in which an embedded system requires more ports than available on a particular microprocessor. For example, one may desire 10 ports, while the microprocessor only has 4 ports. An *extended parallel I/O* peripheral can be used to achieve this goal.

Similarly, a system may require parallel I/O but the microprocessor may only have a system bus. In this case, a *parallel I/O* peripheral may be used. The peripheral is connected to the system bus on one side, and has several ports on the other side. 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.

Example: memory mapped and standard I/O

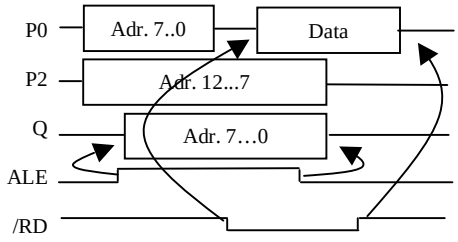
`Two examples of real processor I/O to be added.`

6.4.2 Interrupts

Another microprocessor I/O issue is that of interrupt-driven I/O. In particular, 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, then 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

Example: A basic memory protocol

In this example, we illustrate how to interface 8K of data and 32 K of program code memory to a micro-controller, specifically the *Intel 8051*. The Intel 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 (LSBs) and P2 (MSBs). A separate signal, called PSEN (program strobe enable), is used to distinguish between data/code. For the most part, the I8051 generates all of the necessary signals to perform memory I/O, however, since port P0 is used for both LSB address bits and data flow into and out of the RAM an 8-bit latch is required to perform the necessary multiplexing. The following timing diagram illustrates a memory read operation. Memory write operation is performed in a similar fashion with data flow reversed and RD (read) replaced with WR (write).



Memory read operation proceeds as follows. The micro-controller places the source address, i.e., the memory location to be read, on ports P2 and P0. P2, holding the 8-MSB bits of the address, retains its value throughout the read operation. P1, holding the 8-LSB bits of the address is stored inside an 8-bit latch. The ALE signal (address latch enable), is used to trigger the latching of port P0. Now, the micro-controller asserts high impedance on P0 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 micro-controller reads the data and de-asserts its control and port signals. The following figure gives the interface schematic.

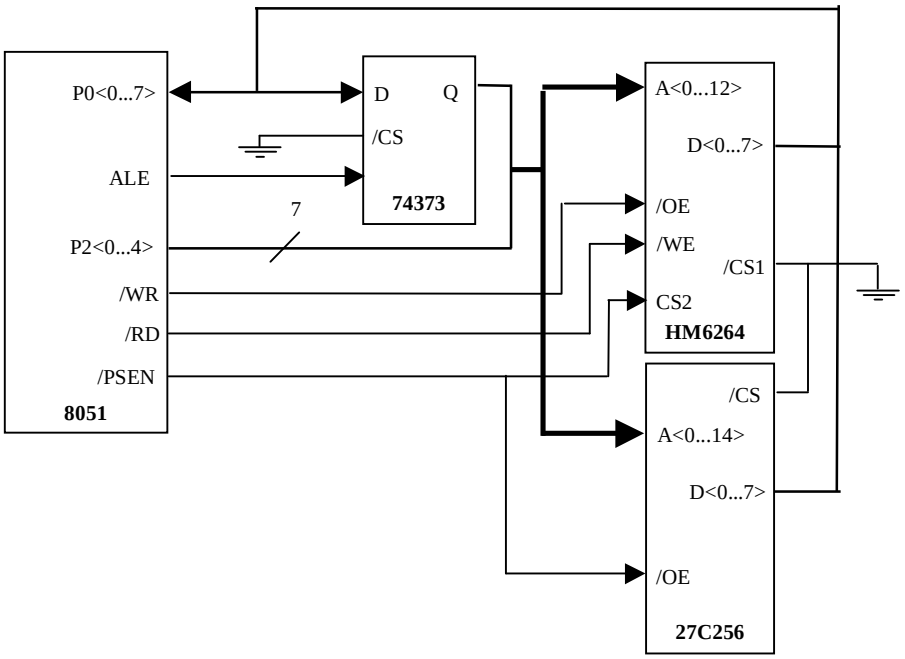
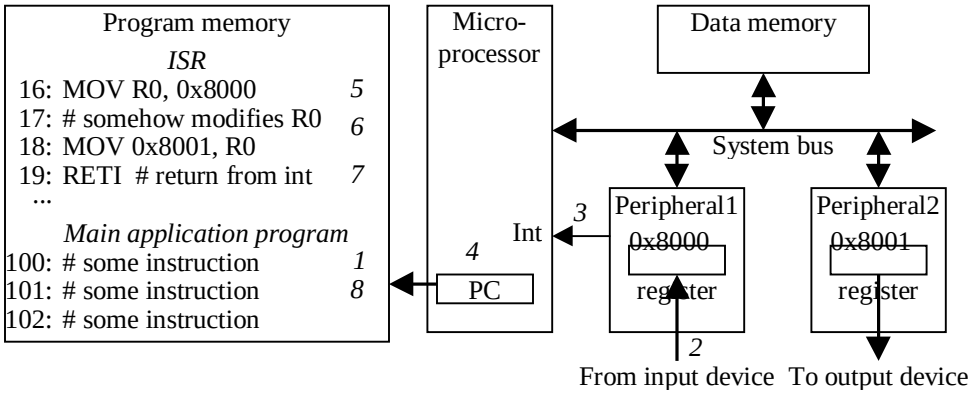


Figure 6.5: Interrupt-driven I/O using fixed ISR location.



1. *Microprocessor is executing its program, say current program counter (PC)=100.*
2. *Peripheral1's input device stores new data in register at 0x8000.*
3. *Peripheral1 asserts Int to request servicing by microprocessor of the new data.*
4. *Microprocessor stops executing its program and stores state, including PC.*
5. *Microprocessor jumps to fixed location of ISR at program memory location 16 (sets PC=16)*
6. *ISR reads data from 0x8000, modifies the data, and writes to 0x8001.*
7. *ISR returns, causing restoration of the microprocessor state, thus setting PC= 100+1 = 101*
8. *Microprocessor resumes executing its program.*

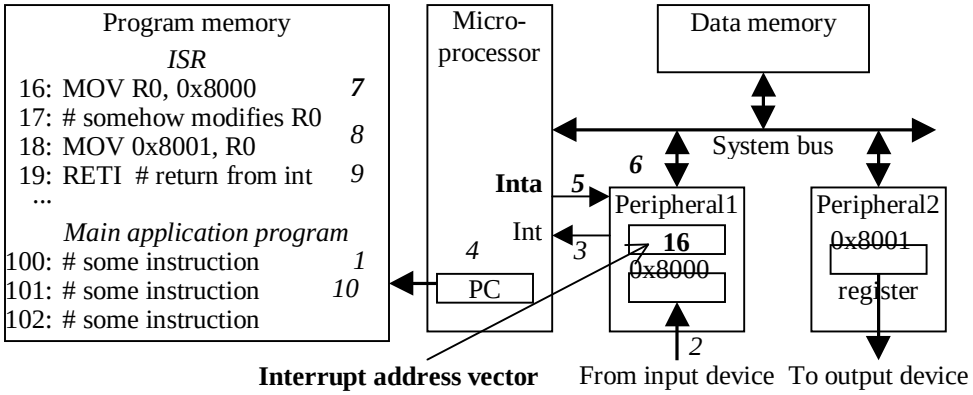
the peripheral. This repeated checking by the microprocessor for data is called *polling*. Polling is simple to implement, but this repeated checking wastes many clock cycles, so may not be acceptable in many cases, especially when there are numerous peripherals to be checked. We could check at less-frequent 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. In some processors, the address to which the microprocessor jumps on an interrupt is *fixed*. The assembly programmer either puts the ISR there, or if not enough bytes are available in

Figure 6.6: Interrupt-driven I/O using vectored interrupt.



1. *Microprocessor is executing its program, say current program counter (PC)=100.*
2. *Peripheral1's input device stores new data in register at 0x8000.*
3. *Peripheral1 asserts Int to request servicing by microprocessor of the new data.*
4. *Microprocessor stops executing its program and stores state, including PC.*
5. ***Microprocessor asserts Inta***
6. ***Peripheral1 puts its interrupt address vector 16 on the system bus***
7. ***Microprocessor jumps to the address of ISR read from the data bus, in this case 16.***
8. *ISR reads data from 0x8000, modifies the data, and writes to 0x8001.*
9. *ISR returns, causing restoration of the microprocessor state, thus setting PC= 100+1 = 101*
10. *Microprocessor resumes executing its program.*

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. In microprocessors with fixed ISR addresses, there may be several interrupt pins to support interrupts from multiple peripherals. For example, Figure 6.5 provides an example of interrupt-driven I/O using a fixed ISR address. In this example, data received by Peripheral1 must be read, transformed, and then written to Peripheral2. Peripheral1 might represent a sensor and Peripheral2 a display. Meanwhile, the microprocessor is running its main program, located in program memory starting at address 100. When Peripheral1 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 Peripheral1, transforms it, and writes the result to Peripheral2. 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 numerous peripherals that can request service. In this method, the

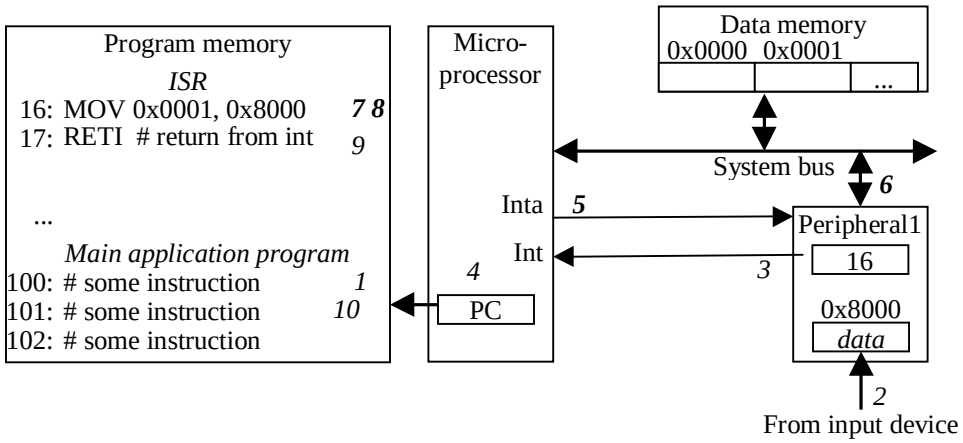
microprocessor has one interrupt pin, say *IntReq*, which any peripheral can assert. After detecting the interrupt, the microprocessor asserts another pin, say *IntAck*, to acknowledge that it has detected the interrupt and to request that the interrupting peripheral provide the address where the relevant ISR resides. The peripheral provides this address on the data bus, and the microprocessor reads the address and jumps to the 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, shown in Figure 6.6, which represents an example very similar to the previous one. In this case, however, the ISR location is not fixed at 16. Thus, Peripheral1 contains an extra register holding the ISR location. After detecting the interrupt and saving its state, the microprocessor asserts *Inta* in order to get Peripheral1 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 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 *non-maskable*. 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 exists time-critical regions of code, such as a routine that generates a pulse of a certain duration. The programmer may include an instruction that disables interrupts at the beginning of the routine, and another instruction re-enabling interrupts at the end of the routine. Non-maskable 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 non-maskable interrupt can cause a jump to a subroutine that stores critical data in non-volatile 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, and 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 non-stored register data with the ISR. These microprocessors need two types of assembly instructions for subroutine return. A regular return instruction returns from a regular

Figure 6.7: Peripheral to memory transfer *without* DMA.



1. Microprocessor is executing its program, say current program counter (PC)=100.
2. Peripheral1's input device stores new data in register at 0x8000.
3. Peripheral1 asserts *Int* to request servicing by microprocessor of the new data.
4. Microprocessor stops executing its program and stores its state, including PC.
5. **Microprocessor asserts *Inta***
6. **Peripheral1 puts its interrupt address vector 16 on the system bus.**
7. **Microprocessor jumps to the address of ISR read from the data bus, in this case 16.**
8. ISR reads data from 0x8000 and writes that data to 0x0001 in data memory.
9. ISR returns, causing restoration of the microprocessor state, thus setting PC= 100+1 = 101, and terminating handshake with peripheral.
10. Microprocessor resumes executing its program.

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.

Example: Interrupts

`Two examples of real processor interrupt handling to be added, one using fixed interrupt, the other vectored.`

6.4.3 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 to await processing is called buffering. For example, packet-data from an Ethernet card is stored in main memory and is later processed by the different software layers (such as IP 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 6.7 shows an example in which Peripheral1 interrupts the microprocessor when receiving new data. The microprocessor jumps to ISR location 16, which moves the data from 0x8000 in the peripheral to 0x0001 in memory. Then 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). 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 takes place.

We set up a system for DMA as follows. As shown in Figure 6.8, 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 microprocessor device; all it 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 Hreq (bus Hold REQuest), is used by the DMA controller to request control of the bus. The other pin, which we'll call Hlda (HoLD Acknowledge), 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.

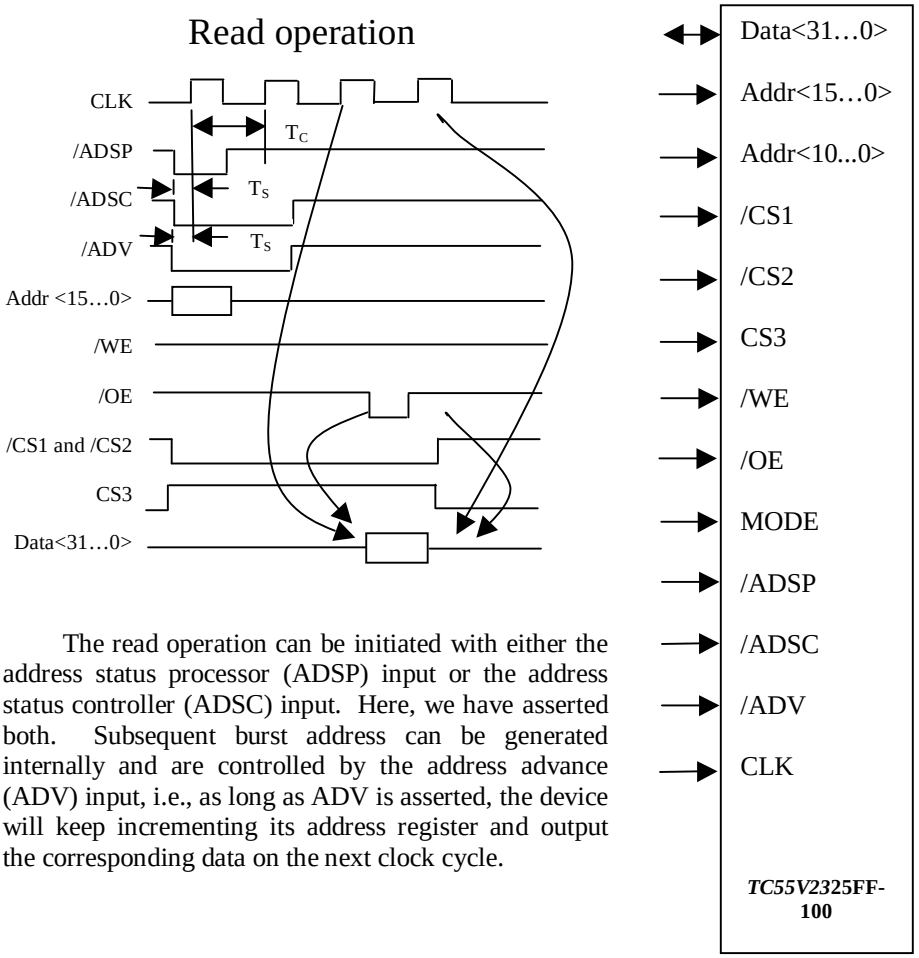
Example: TC55V2325FF-100 memory device

In this example, we introduce a 2M bit synchronous pipelined burst SRAM memory device designed to be interfaced with 32 bit processors. This device, made by Toshiba Inc., is organized as 64K × 32 bits. The following table summarizes some of its characteristics.

Device	Access Time (ns)	Standby Pwr. (mw)	Active Pwr. (mw)	Vcc Voltage (V)
TC55V2325FF-100	10	na	1200	3.3

Source Toshiba TC59S6432CFT datasheets.

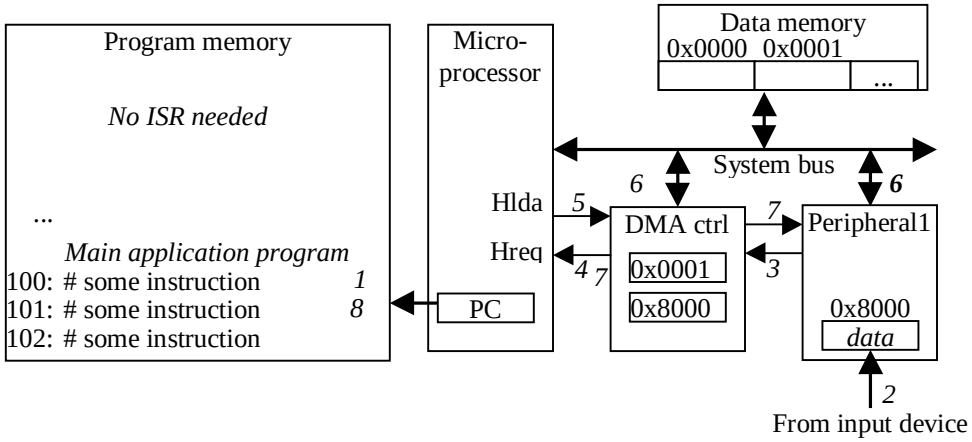
Here, we present the block and timing diagram for a single read operations. Write operation is similar. This device is capable of sequential, fast, reads and writes as well as singles byte I/O. Interested reader should refer to the manufacturer's datasheets for complete pinout and complete timing diagrams.



The read operation can be initiated with either the address status processor (ADSP) input or the address status controller (ADSC) input. Here, we have asserted both. Subsequent burst address can be generated internally and are controlled by the address advance (ADV) input, i.e., as long as ADV is asserted, the device will keep incrementing its address register and output the corresponding data on the next clock cycle.

*T_C =10ns, T_S =3ns

Figure 6.8: Peripheral to memory transfer *with* DMA.

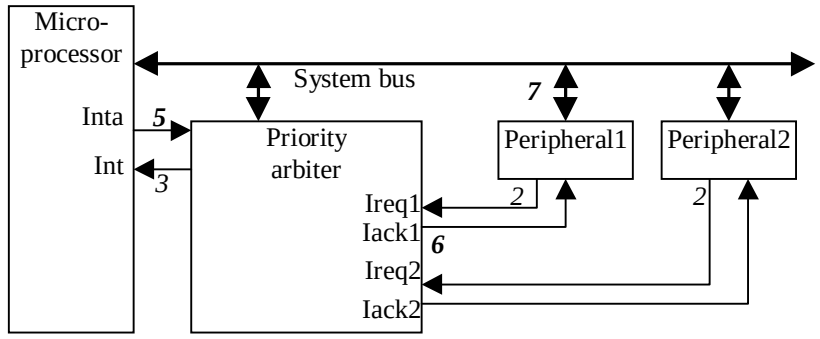


1. Microprocessor is executing its program, say current program counter (PC)=100.
2. Peripheral1's input device stores new data in register at 0x8000.
3. Peripheral1 asserts *Dreq* to request servicing of the new data.
4. DMA controller asserts *Hreq* to request control of the system bus.
5. Microprocessor relinquishes system bus, perhaps stopping after executing statement 100.
6. **DMA controller reads data from 0x8000 and writes that data to 0x0001 in data memory.**
7. DMA controller deasserts Hreq and completes handshake with peripheral
8. Microprocessor resumes executing its program, perhaps starting with statement 101.

To achieve the above, 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 6.8, 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 (like 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 128 internal registers, after which the peripheral requests servicing, i.e., requests that this data be buffered in memory. The DMA controller gains

Figure 6.9: Arbitration using a priority arbiter.



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
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 *Iack1* to acknowledge Peripheral1.**
7. **Peripheral1 puts its interrupt address vector on the system bus**
8. **Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).**
9. Microprocessor resumes executing its program.

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 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.

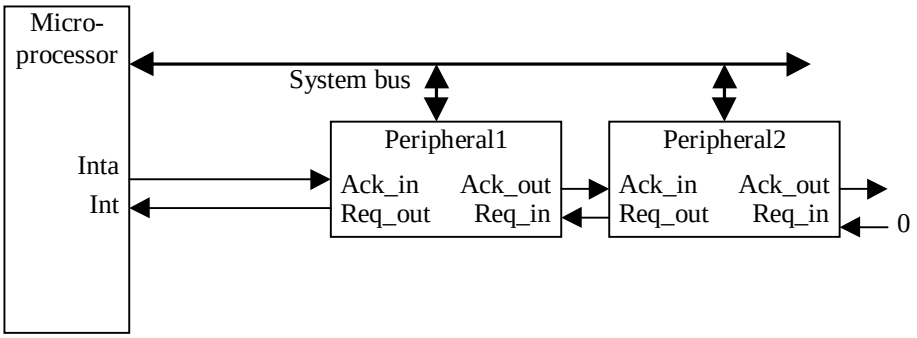
Example: Intel 8237

`Description of the Intel 8237 DMA controller and an example of its use.`

6.5 Arbitration

In our discussions above, 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,

Figure 6.10: Arbitration using a daisy-chain configuration.



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, i.e., to decide which one of the contending peripherals gets service, and thus which peripherals need to wait. Several methods exist, which we now discuss.

6.5.1 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 6.9. 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 acknowledgement 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).

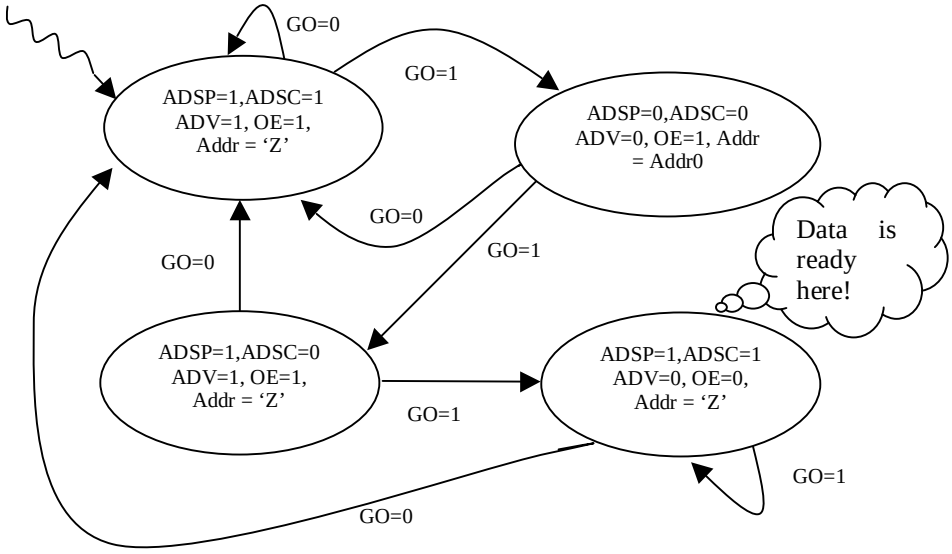
Priority arbiters typically use one of two common schemes to determine priority among peripherals: fixed priority or rotating priority. In *fixed priority* arbitration, each peripheral has a unique rank among all the peripherals. The rank can be represented as a number, so if there are four peripherals, 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.

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 the previous example. Our specification for this FSM is the timing diagram presented in the previous example. 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. Here is the finite-state machine description.



From the above state machine description we can derive the next state and output truth tables. From these truth tables, we can compute output and next state equations. By deriving the next state transition table we can solve and optimize the next state and output equations. These equations, then, can be implemented using logic components. (See chapter 4 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.

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*.

Example

`Description of a real Intel 8259 priority arbiter and an example of its use.`

6.5.2 Daisy-chain arbitration

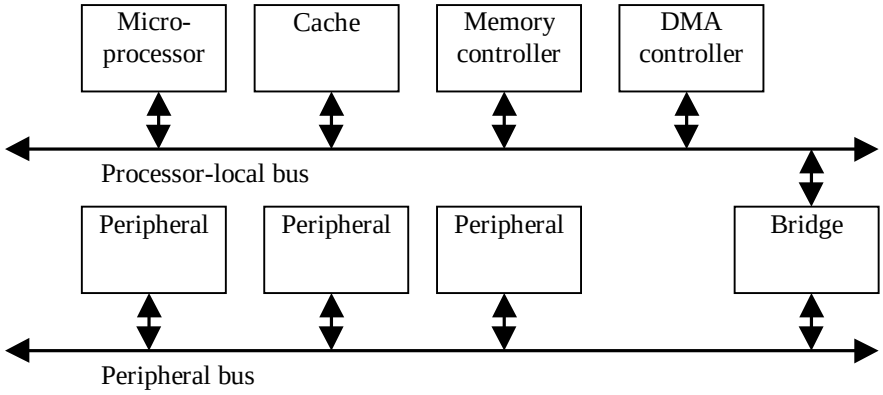
The daisy-chain arbitration method builds arbitration right into the peripherals. A daisy-chain configuration is shown in Figure 6.10, again using vectored interrupt to illustrate the method. Each peripheral has a request output and an acknowledge input, as before. But now each peripheral also has a 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, 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 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 acknowledgement 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 acknowledgement 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.

6.5.3 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

Figure 6.11: A two-level bus architecture.



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 I2C 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.

6.6 Multi-level 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 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.

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 6.11. The processor local bus typically connects the microprocessor, cache, memory controllers, certain high-speed co-processors, and is highly 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 gates and less power for interfacing.

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.

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 co-processors.

6.7 Summary

Interfacing processors and memories 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. 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 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, like 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.

6.8 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.

6.9 Excercises

1. Draw the timing diagram for a bus protocol that's handshaked, non-addressed, and transfers 8 bits of data over a 4-bit data bus.

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

3. (a) 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. (b) Draw a timing diagram showing what happens during the transfer; skip the 2nd through 99th bytes.

4. (a) Draw a block diagram of a processor, memory, two peripherals and a priority arbiter, in which the peripherals request servicing using vectored interrupt. Show all relavant control and data lines of the bus, and label component inputs/outputs clearly. (b) List the steps that occur during for such an interrupt.

5. Repeat 4(a) and (b) for a daisy chain configuration.