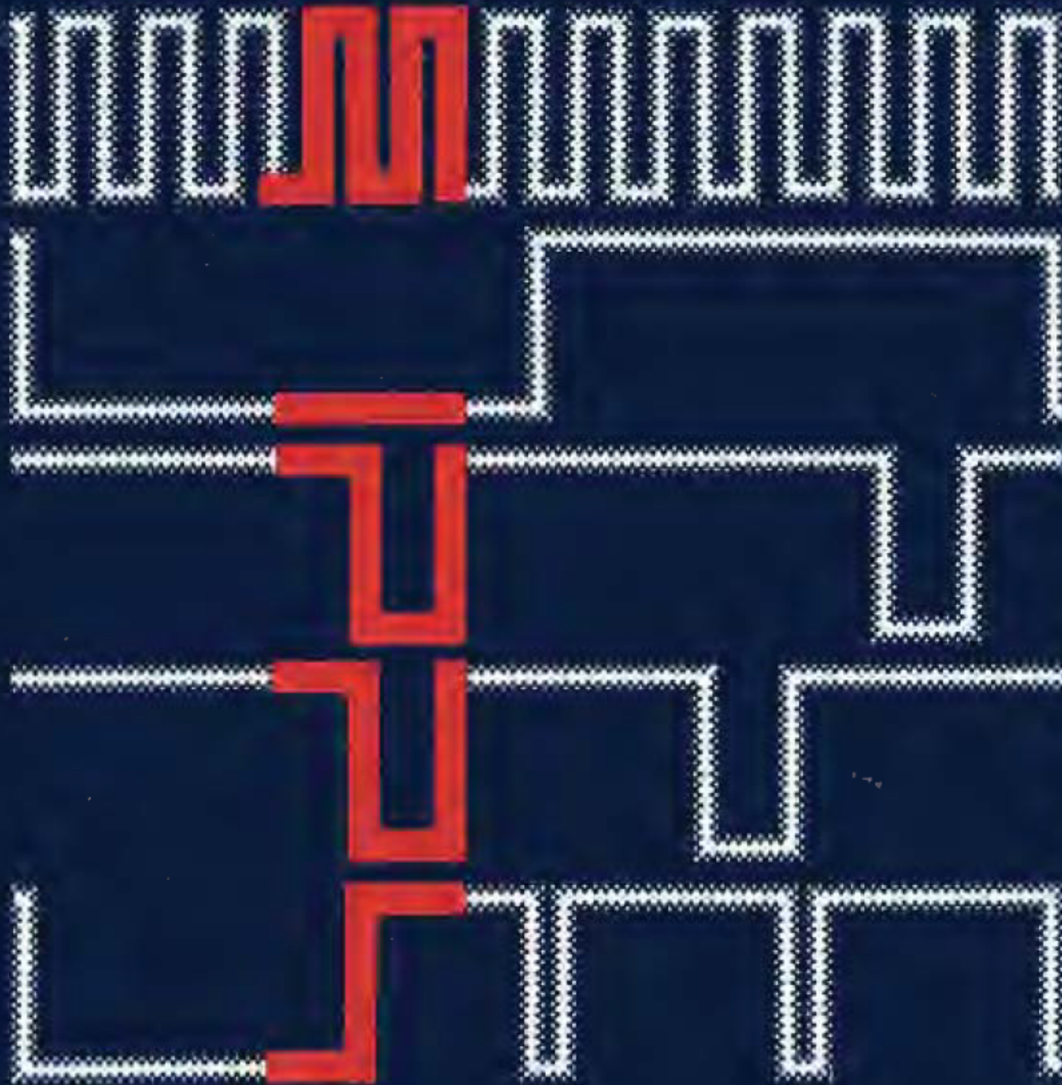
MICROCOMPUTER INTERFACING

Harold S. Stone



Samsung Exhibit 1020

Microcomputer Interfacing

HAROLD S. STONE
UNIVERSITY OF MASSACHUSETTS, AMHERST

ADDISON-WESLEY PUBLISHING COMPANY
READING, MASSACHUSETTS
MENLO PARK, CALIFORNIA
LONDON
AMSTERDAM
DON MILLS, ONTARIO
SYDNEY

This book is in the **ADDISON-WESLEY SERIES IN ELECTRICAL ENGINEERING**

SPONSORING EDITOR: *Tom Robbins*
PRODUCTION EDITOR: *Marilee Sorotskin*
TEXT DESIGNER: *Herb Caswell*
ILLUSTRATOR: *Jay's Publishers Service Inc.*
COVER DESIGN AND ILLUSTRATOR: *T. A. Philbrook*
ART COORDINATOR: *Joseph Vetere*
PRODUCTION MANAGER: *Sue Zorn*
PRODUCTION COORDINATOR: *Helen Wythe*

The text of this book was composed in Times Roman on a Mergenthaler 202 by Information Sciences Corporation and was printed by R. R. Donnelley and Sons.

Library of Congress Cataloging in Publication Data

Stone, Harold S., 1938–
Microcomputing interfacing.
Bibliography: p.
1. Interface circuits. 2. Microcomputers—Circuits.
I. Title. TK7868.158S76 621.3819'5835 81-17619
ISBN 0-201-07403-6 AACRZ

Reprinted with corrections, February 1983

Copyright © 1982 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Printed simultaneously in Canada.

ISBN 0-201-07403-6
BCDEFGHIJK-DO-89876543

3 / BUS INTERCONNECTIONS

Now that we have covered both the high-level functional description of microcomputers and have looked as well at the lowest-level of implementation details, we will work our way quickly into practical interfacing techniques. This chapter treats the data paths that tie together the processor, memory, and I/O modules of a microcomputer. The strategy followed for these interconnections is similar across all microcomputers; they make use of a general structure that we call a *bus*. A bus is a collection of signal lines that carry module-to-module communications in a microcomputer. In almost all cases bus lines are unbroken, and modules simply tap onto a bus by connecting their respective inputs and outputs directly to corresponding bus signal lines. (The only exception to this rule is for signal lines used for priority resolution, as described later in this chapter.)

For high-performance applications, buses must be restricted in length, thus limiting their use to the short module-to-module connections within a computer chassis. Although these buses can be extended from one chassis to another, performance and reliability suffer as bus length increases. For the longer and lower-performance interconnections, most microcomputer systems rely on special buses, quite separate from their high-speed internal buses, or on other point-to-point connections in order to isolate the high-speed buses from the long physical buses, thereby reducing the degradation caused by excessive bus length. Exceptions to this practice occur in low-speed applications where the internal bus runs slow enough to be extended to a second chassis with little or no performance penalty. With just one type of bus, the system avoids an additional burden of integrating two distinct bus systems and protocols.

3.1 BUS FUNCTIONS

The signal lines that collectively form a bus break naturally into three groups as shown in Fig. 3.1. One group of signals carries the basic information to be communicated on the bus; the other two signal groups guarantee that the information is delivered during a bus transaction. From the earlier discussion of the functional behavior of a microprocessor, we know that the first group of signals carries such information as

1. memory address (or port ID),
2. data, and
3. command type (READ, WRITE, DATA, STATUS).

Since there are a vast number of different buses in use, there is a wide variation in just what information is carried on the first group of lines. Generally speaking, this group carries information that one module needs to convey to another in order to invoke a remote

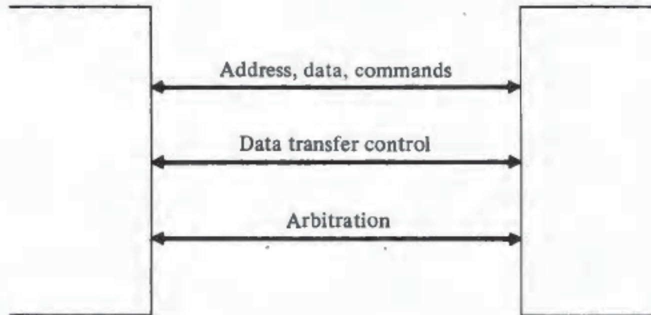


FIGURE 3.1 Bus signal and control lines.

function, response, or change of state in the remote module. In order to pass the information, the bus itself has to be controlled and operated correctly. The other two groups are dedicated to different aspects of the latter function.

The second group of signal lines controls the timing of the data transfer. This group is often called the *data handshake lines*, and contains the signals that dictate when each individual data transfer begins and ends. The handshake lines have a role analogous to traffic lights on a roadway. The handshakes start and stop transactions, and they exert the same functional control on all transactions regardless of transaction type.

The type of transaction comes into play on the third group of lines, the arbitration lines, which give critical transactions priority over less critical ones when deciding what transactions shall access the bus. This third group of lines arbitrates which module gains access to the bus. The necessity for arbitration is due to the inherent problem that occurs when two or more modules attempt to transmit information simultaneously. If module A spews forth a logic 0 while module B attempts to transmit a logic 1 at the same instant of time, we say that there is a *bus conflict*. The signal actually delivered depends on the logic family that drives the bus. A line driven by open-collector drivers moves to the 0 state during any conflict, so that in the given example, the logic 1 output by module B is lost. Then one or both modules lose data at the point of conflict, and what data are lost is unpredictable. Hence, conflicts almost certainly result in a communications failure on the bus. To ensure reliable communication, as a general rule only one module at a time can transmit on the bus, although potentially all other modules can accept the transmission and change state in response to it.

A bus conflict can be more disastrous than portrayed here. For example, what happens when tri-state drivers engage in a bus conflict? In this case, there is a possibility of damaging the bus drivers because the conflict creates a low impedance path from V_{CC} to ground through the output stages of the conflicting gates. The high current through this

path can burn out both driving gates. If either gate fails in a shorted condition, the failure could be in conflict with other driving gates on the same signal line, and burn them out as well. If bus conflicts occur during an instruction-fetch cycle, the instruction received by the processor is a corrupted version of its original form, and the incorrect version almost inevitably wreaks havoc in the program.

The role of the arbitration lines is then very clearly defined. They guarantee that, at most, one module at a time transmits on the bus. The first two groups of signals, the information and handshake groups, are thus protected from conflict by the arbitration group. The arbitration group has inherent conflicts because all potential transmitting devices use these lines concurrently as part of the arbitration process. Therefore, in many buses the arbitration lines are driven with open-collector devices, and the arbitration protocol depends on the OR-function logic of the open-collector gate.

Later sections of this chapter treat various methods for implementing both the handshake and arbitration protocols. Even though the context of the discussion is buses, the protocols have a use that extends into other areas of microcomputers as well. For example, an arbitration protocol for selecting one of several potential bus transmitters is also suitable for selecting one of several I/O ports in an interrupt-priority resolver.

3.2 THE BUS HANDSHAKE

Handshake protocols fall generally into three broad classes:

1. synchronous (clocked transfer, one clock period per transfer),
2. asynchronous (unclocked), and
3. semisynchronous (clocked transfer, one or more clock periods per transfer).

Since the specific function of the handshake lines is to indicate the beginning and end of a data transfer, the handshake lines must somehow mark these points through voltage changes in the handshake signals. Some buses have very complex, sequential timing for each data transfer, perhaps requiring a number of different data to pass along the bus during a single transaction. For these buses, the handshake lines signal the beginning and end of each subcycle within the full cycle, as well as identifying the start and end of the full cycle.

The three generic handshake techniques span a spectrum of different approaches from complete control by a clock to no clock control whatsoever. Synchronous protocols are among the easiest to implement because the only control signal is a clock oscillator. The rising and falling edges of the clock signify, respectively, the beginning and end of a bus cycle. All memories, peripherals, and processors on the bus are controlled by the same clock oscillator so that modules operate in "lock-step," advancing cycle by cycle as the clock line ticks away. Not only are synchronous protocols the least complex of the three protocols, but they also, in general, lead to the fastest transactions (provided that the responding devices are fast enough to operate at the bus-clock speed).

Synchronous Buses

The timing of a typical synchronous protocol is illustrated in Fig. 3.2. The top waveform is the bus clock, which synchronizes all modules to a common time base. (It is shown here with a 50% duty cycle, but the actual duty cycle differs for various synchronous buses.) Address and data lines are shown on the next two waveforms. The addresses and data reach their stable values at the beginning of the shaded area, retain their values through the high half-cycle of the clock, and fall at the end of the trailing shaded area. Although the address and data lines are shown in the high-state during the active portion of the clock, they actually can be in either a high or a low state, depending on the information they convey. The figure actually shows the period during which the address and data lines are *stable*, and does not show their logic values.

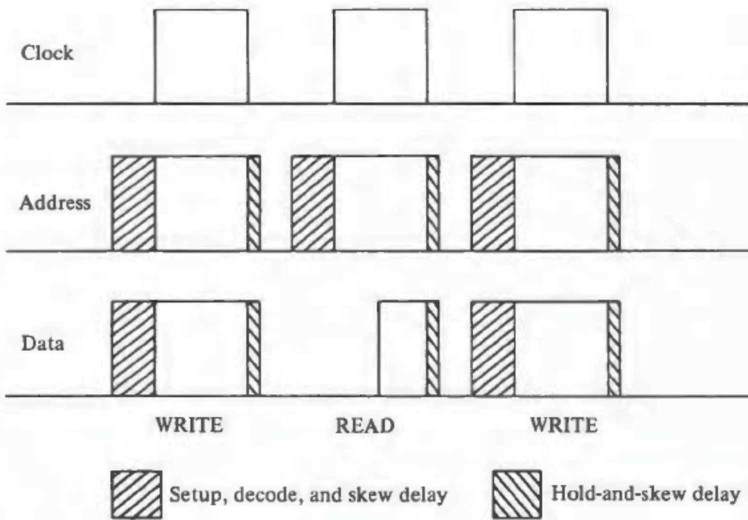


FIGURE 3.2 Timing for a synchronous bus.

There are several different reasons for the shaded area of the waveforms. Fig. 3.3 shows one source of logic delay in the address decoder of a receiving module on the bus. The figure shows a bus transmitter, hereafter called a *bus master*, transmitting to a receiver labeled *bus slave*. There are potentially many slaves on the bus, and the purpose of the address lines is to select a single slave to respond to the bus transaction. Therefore, the figure shows the address lines entering a decoder that detects the slave's address, which then selects this specific slave by producing a signal that forces the slave to load data from the bus when the clock reaches the active phase of its cycle. The decoder has to produce its signal in advance of the rising of the edge of the clock, so that the address lines must be stable for at least the duration of the logic delay through the decoder.

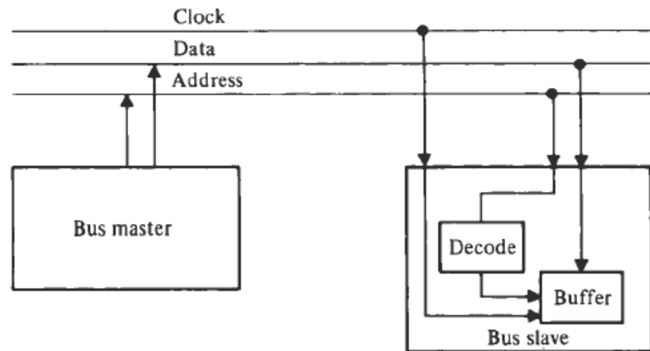


FIGURE 3.3 Typical slave internal structure.

Another related effect that cannot be ignored is the setup time and hold time of logic in the buffer. *Setup time* is the minimum amount of time that a control signal has to be present on an input of a memory device before the clock triggers a transfer into the device. *Hold time* is the minimum time that data has to be held stable on the inputs of a memory device after a clock change triggers a transfer into that device. The setup time for the diagram in Fig. 3.3 for a WRITE into the slave is the time required for the address lines to be stable after they reach the buffer, but before applying a clock to the buffer. The hold time in Fig. 3.2 depends on whether the bus operation is READ or WRITE. For a WRITE, the hold time is the hold time of the buffer in the slave. For a READ, the hold time is the hold time of the equivalent buffer in the master. In both cases, the addresses and data must be stable for at least the duration of the hold time after the clock changes state. Address and data lines need to have identical setup and hold times. If they are not identical, the bus protocol must incorporate setup and hold times that are long enough to satisfy the maximum of the address and data requirements.

In the light of the information on setup and hold times, let us return to Fig. 3.2 to consider how these times are represented in the figure. For a WRITE operation, the master transmits both addresses and data in advance of the rising edge of the clock. During this time the slave decodes the address, and the data lines stabilize at the buffer. When the clock rises, the selected slave initiates an internal WRITE operation, during which it copies the data on its data lines into an address or register identified by the address lines. If the slave is a memory chip, the subsequent delay accounts for the write-access time to memory, usually on the order of 100 to 200 ns for moderate-speed metal-oxide silicon (MOS) devices. Other devices can be bus slaves as well, including I/O ports and discrete registers. Some devices respond faster than the moderate-speed memory, but the fixed-cycle time of the synchronous bus cannot take advantage of the faster response. The falling edge of the clock signifies the end of the bus cycle. At this time, the WRITE operation is complete, and the slave can disconnect logically from the data lines.

The READ operation is similar to WRITE for the address lines, but data lines behave differently. In this case the rising edge of the clock initiates a memory READ in the slave. Some time after the clock rises, the data reaches the output buffer of the slave, which in turn places the data on the bus. The data has to be on the bus at least one setup time before the falling edge of the clock, where the setup time in question is the setup time of the master's data buffer. The slave holds the data on the bus at least one hold time after the falling edge of the clock in order to satisfy the hold-time requirements of the master.

Reexamination of Fig. 3.2 shows we have accounted for the general form of the shaded area, although we have not accounted for exact lengths of time. Note that the setup time is shown much longer than the hold time because the setup time includes the decoding delay in the slave, as well as other factors we now examine.

Among the other sources of timing delay accounted in the shaded area is *signal skew*, which is explained more fully in Fig. 3.4. The top two waveforms show the signals on two address lines as they appear at the bus master. Both signals are assumed to change at exactly the same instant for the purposes of this discussion although, in reality, the master itself may produce these signals displaced slightly with respect to each other. The master transmits the signals over the bus to the slave, which sees the signals as shown in the lower two waveforms of the figure. Note that the signals no longer change at the same instant of time, but now one changes D time units later than the other. This change in relative timing produced somewhere in the bus system is what we mean by *skew*.

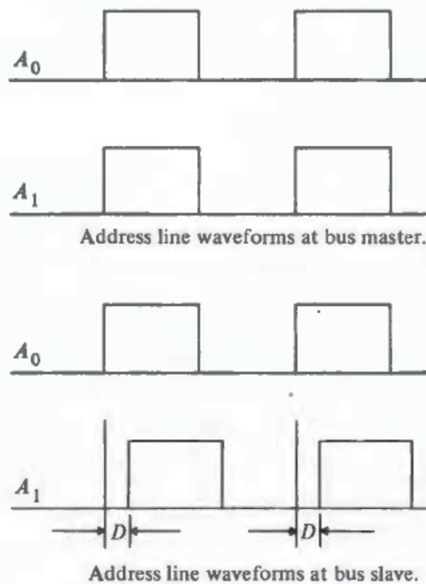


FIGURE 3.4 Skew in signal transmission. The delay D is the skew in the signals.

Several different sources of skew account for the delay. One source is a difference in the propagation delays of the two signals because the signals follow slightly different paths in going from master to slave. Propagation delays usually influence skew less than the varying logic delays through gates on the path from master to slave. Gate delays may vary from chip to chip by 10 to 20 ns, depending on the chip family. Since each of the bus signals travels through a different set of gates, the end-to-end propagation time is rarely the same for all bus signals. The rise time and fall time of a signal also affect skewing delays. A gate recognizes a change in a signal when the signal voltage passes the gate threshold. If capacitive effects stretch out the rise or fall time, there is an apparent increase in the delay between the start of a signal transition and the time when the transition is recognized. Since this time also depends on the gate threshold, differences in gate thresholds contribute to differences in skew in much the same way that rise and fall times impact skew.

To compensate for skew, addresses must stabilize at least one maximum skew time earlier than in the absence of skew, just in case some address line is delayed by skew relative to the rising edge of the clock. Hence the shaded area said to be setup time in Fig. 3.2 includes this skew time plus the decoding time and address setup time. Note also that the hold time for data written includes skew to protect against problems caused by clock skew. If the clock were delayed relative to the data during the propagation of the signals from master to slave, then the apparent hold time of data at the slave is diminished by the amount of the skew. Hence, for a WRITE cycle the master has to assert data for at least one hold time plus one skew time after the clock edge falls.

It is interesting to consider the effects of propagation delays on hold time. For the READ operation, propagation delays actually reduce the hold time somewhat, whereas for WRITE operations they have no effect unless increases in propagation delay tend to increase clock skew also. Consider the READ in Fig. 3.2, for example, and observe what happens if there are significant propagation delays between master and slave. When the master drops the clock signal at the end of the cycle, the output data at the slave remains stable at the master's input buffer for at least one round-trip propagation time between master and slave. This is true because the clock edge change has to propagate from master to slave, and the resulting changes on the data lines then propagate back to the master. Technically speaking, a slave can reduce hold time by the amount of a propagation delay, but in practice it is very difficult to do so. The propagation time delay depends on the relative positions of the master and slave on the bus, and this varies from configuration to configuration. Yet the slave module has to be engineered to work in every configuration, so that at best the slave can take advantage of the shortest propagation delay that can occur in any configuration. This delay is so unpredictable, and likely to be very small in any event, that it is rarely worthwhile to consider.

This brings us to the end of our discussion of the details of Fig. 3.2. To summarize the effects that limit the bus bandwidth, we have

1. setup time of data and control signals before clocking data into a buffer,
2. address decode delay,

3. skew time of address and data signals relative to a rising and to a falling clock edge,
4. hold time of data at a buffer input, after clocking data into the buffer, and
5. one round-trip propagation delay (for the READ operation).

The bus cycle time cannot be smaller than

$$T_{\mathrm{SETUP}} + T_{\mathrm{DECODE}} + 2T_{\mathrm{SKEW}} + \mathrm{MAX}(T_{\mathrm{HOLD}}, T_{\mathrm{RT\text{-}PROP}}),$$

where the MAX operation recognizes that propagation delay can be overlapped with hold time. If the cycle time of a bus is shorter than the time given here, the signaling rate fails to meet the signal specifications for modules that connect to that bus, so that incorrect or unreliable computations may result. Even this upper limit on bandwidth is overly optimistic. In practical situations, the master itself has an internal delay between transactions, and the slave has a nonzero access time, both of which increase the minimum cycle time and decrease realizable bandwidth.

The primary advantage of the synchronous system is simplicity. Data transfers are controlled through a single signal, and the data transfers run with minimal overhead in terms of skew, setup, hold, and propagation delays. However, the synchronous bus has a serious problem in dealing with slow slaves connected to the bus. The synchronous bus described thus far cannot accommodate devices whose access time is greater than the time available during a clock period. With the given bus protocol, the clock rate has to be set slow enough to satisfy the slowest device on the bus where the device's response time includes the effect of propagation delays due to physical separation. But this reduces the bandwidth for all transactions, and the slow device has thereby decreased the potential system performance even though the slow device is rarely accessed.

Asynchronous Buses

For the computer that drives a mix of devices with widely varying access times, the synchronous protocol may be inappropriate because the bus runs at the speed of the slowest device. Intuitively speaking, it is advantageous to have fast transactions for fast devices and slow transactions for slow or distant devices, so that transaction time varies with the device rather than being fixed for all time by a system clock. The timing and control signals for a typical asynchronous bus that has these characteristics appear in Fig. 3.5.

This bus is said to be a *fully interlocked asynchronous bus*, and is by far the most popular asynchronous protocol in use today. The DEC Unibus for the PDP-11 family is one notable implementation of this protocol (Digital Equipment, 1979). The term *fully interlocked* stems from the way the two control signals work together during a bus transaction. The control signals in the figure are called MASTER and SLAVE, and take the name of the module that produces their respective signals. The interlocked protocol requires changes to alternate between the control signals and to occur sequentially, with a change in one signal arming the other for its subsequent change. By interlocking in this manner, the information on address and data lines is guaranteed to be transmitted without conflict and without loss or duplication by the bus.

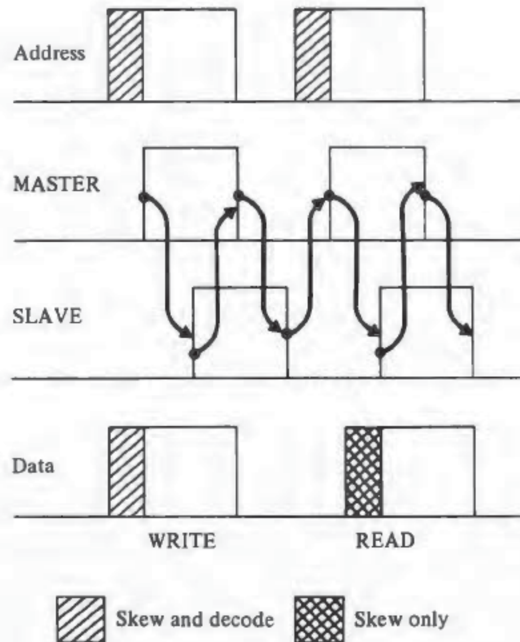


FIGURE 3.5 Timing for a fully interlocked asynchronous bus.

For the WRITE transaction, the bus master places address and data on the bus. After a delay to allow for skew, decoding, and setup time, the bus master raises MASTER, which signifies to the slave that the data can be accepted. Raising MASTER thus triggers a slave memory to initiate a WRITE cycle, and latches data into a slave buffer register. In any case, action at the slave takes place only after MASTER is asserted.

While the slave is busy copying data in response to MASTER, the SLAVE signal remains low. When copying is completed, the slave module raises SLAVE to signify, "I've got it." The handshake continues with MASTER going low ("I see you've got it"), and SLAVE going low ("I see you see I've got it"). The last two transitions are part of a sequence to guarantee that neither MASTER nor SLAVE changes too quickly. SLAVE stays high as shown in the figure until the MASTER signal goes low, thus ensuring that the high SLAVE signal has been observed and acted upon. Only then does SLAVE go low. Similarly, a new transaction cannot be initiated until SLAVE goes low signifying the end of the present transaction. Hence the rising edge of MASTER (and the transitions on the address and data lines) are interlocked to the fall of SLAVE.

A READ transaction is very similar to a WRITE, with the high value of MASTER initiating the operation at the slave after the bus master places an address on the bus. SLAVE goes high after the slave module accesses the datum requested and places it on the bus. In this context, a high value on SLAVE signifies, "The READ is complete."

This triggers the master to load its buffer from the bus. During this period SLAVE must remain high, and the data lines must be stable. If the slave were to change these signals prematurely, the master could read incorrect information. When the master has completed its acceptance of data, it drops MASTER ("I've got it"), and then SLAVE drops ("I see you've got it").

The reasons for the interlocking become clear when we consider how a partially interlocked protocol can fail. Consider the two situations shown in Fig. 3.6. In Fig. 3.6(a), we permit SLAVE to drop a fixed time after it rises, without waiting for MASTER to drop. Likewise, we also remove the interlock between the falling edge of SLAVE and the leading edge of MASTER. Also in Fig 3.6(a), SLAVE goes down well before MASTER does, and we see that the transfer is done safely. The dotted lines show SLAVE delayed somewhat with respect to MASTER, possibly because of long propagation delays or signal skew. In this case, if MASTER drops and rises again while SLAVE is high, it may mistake the high value of SLAVE for a response to the next transfer. This situation is shown in Fig. 3.6(b). Now the master may remove data and addresses too quickly from the bus for the slave to accept the new data. As a result, one transaction is lost.

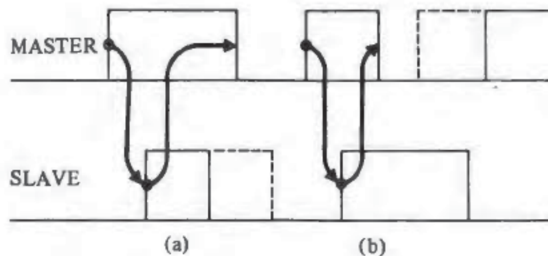


FIGURE 3.6 Examples of signaling with a partially interlocked asynchronous protocol.

Although partial interlocking as shown in Fig. 3.6 lacks the safety of full interlocking, it can be made safe provided that master and slaves adhere to a strict set of timing constraints on the noninterlocked transitions. The advantage of eliminating part of the interlocking is that the bus transaction can be made a little faster so that the bus bandwidth can be greater than it would be in a fully interlocked protocol. But tight constraints usually result in higher manufacturing cost, making the partially interlocked protocols less desirable in general, although useful in specific applications where the extra expense is justified.

Returning to Fig. 3.5, we note in this protocol shaded areas that represent roles similar to those represented by the shaded areas of the synchronous protocol. Addresses have to be raised before MASTER at least early enough to permit address decoding and buffer setup, and to protect against skew on the address lines relative to MASTER. Hold times are not shown specifically, but exist nevertheless. Hold time is usually incorporated into

the slave by delaying the SLAVE signal one hold time after a WRITE is completed, or after presenting data on the bus for a READ. Obviously, the hold time can equally well be incorporated into the master, with the SLAVE signal being presented concurrently with an event while the master delays its actions one hold time after receiving a transition on SLAVE. Whichever of these techniques is used in a protocol, that technique has to be used consistently for all slave and master modules, for otherwise the protocol will not work correctly. Deskewing data and address signals relative to MASTER and SLAVE can normally be combined with hold time, since skew effects are treated by inserting delays in the protocol, in much the way that delays for hold time are inserted into a protocol.

The wide acceptance of the fully interlocked asynchronous protocol is largely due to its reliability and its general efficiency in dealing with devices that have a broad range of response times over long buses. But the protocol is inherently slower than the synchronous protocol because of extra propagation delays. The minimum cycle time for a READ operation must account for

1. deskew (and setup time) of addresses to slave,
2. address decode at slave,
3. deskew (and hold time) of data returned by slave, and
4. two round-trip propagation delays of MASTER and SLAVE signals.

The first three items in this list are comparable to those for synchronous buses, but the propagation delay for the fully interlocked handshake is double that of a clocked bus. Information is passed up and down the bus twice per transaction for asynchronous buses, but only once for synchronous buses. The second round trip is omitted for synchronous protocols because the devices are known in advance to respond within a fixed maximum time. The purpose of the second round trip for an asynchronous bus is to convey completion information that is not bounded in advance.

Semisynchronous Buses

Because the propagation delays of the asynchronous bus severely limit maximum bandwidth, many bus designers have turned to "hybrid" buses that combine the advantages of synchronous and asynchronous buses. One such bus is the *semisynchronous bus* that appears in Fig. 3.7. This bus has two control signals, CLOCK (from the master) and WAIT (from the slave). In some sense the signals play the role of MASTER and SLAVE for the asynchronous bus, but the propagation delays are half those of the asynchronous bus because a single round trip is all that is necessary for a successful handshake. For fast devices, the bus is essentially a synchronous bus controlled by the clock alone. If a slave is fast enough to respond in one clock cycle, it does not raise WAIT, and the semisynchronous bus behaves like a synchronous bus. If the slave cannot respond in one cycle, it raises the WAIT signal, and the master halts. Subsequent clock cycles find the master idle as long as WAIT is asserted. When the slave can respond, it drops WAIT, and the master accepts the slave response using the timing of the standard synchronous protocol. The semisynchronous bus thus has the speed of the synchronous bus and versatility of the

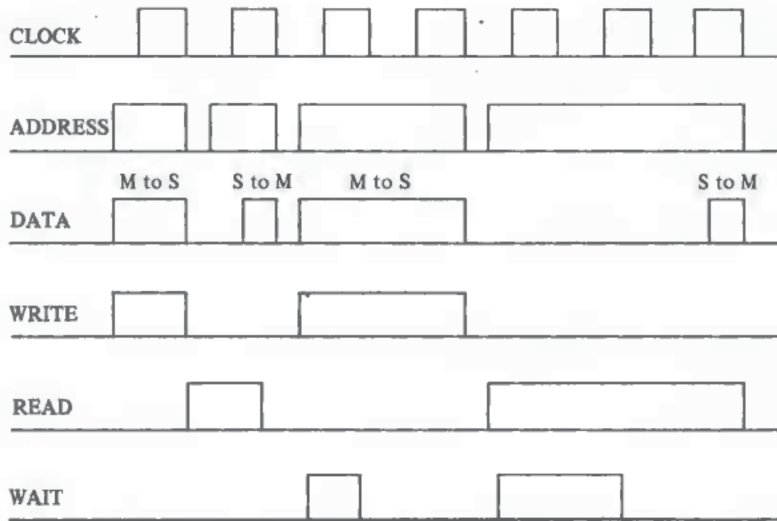


FIGURE 3.7 A semisynchronous bus with cycle times increased by a WAIT signal.

asynchronous bus. However, the length of the semisynchronous bus is limited by the requirement that WAIT must be asserted within a fixed period of time. So these buses cannot have an indefinitely long length, but there is no equivalent timing constraint for asynchronous buses.

Another way to retain the advantage of the fast synchronous protocol while accommodating slow devices is with the use of a "split-cycle" protocol as shown in Fig. 3.8. In this case a READ is split into two separate transactions. During the first transaction, the bus master transmits an address to a slave, and then disconnects from the bus. Other masters then use the bus until the slave is able to return the requested data. At this point, the slave initiates the second part of the split cycle by accessing the bus *as a master* and transmitting the data to the requesting module, which responds *as a slave*. The split cycle places a greater burden on the master and slave modules because each type of module must have the logic to assume both master and slave roles. Moreover, the bus protocol assumes that many different bus masters access the bus at different times, so that every module must also contain the logic for bus arbitration protocol in order to gain access to the bus as a master.

The split-cycle protocol differs slightly in the information passed on the bus from the protocols studied above. For a READ transaction the master supplies a unique identifier for itself together with the address of the requested data so that the slave can return the requested data to the master. In fact, the master identifier is the address used during the second part of the split cycle, and both halves of a READ follow the protocol of a WRITE cycle.

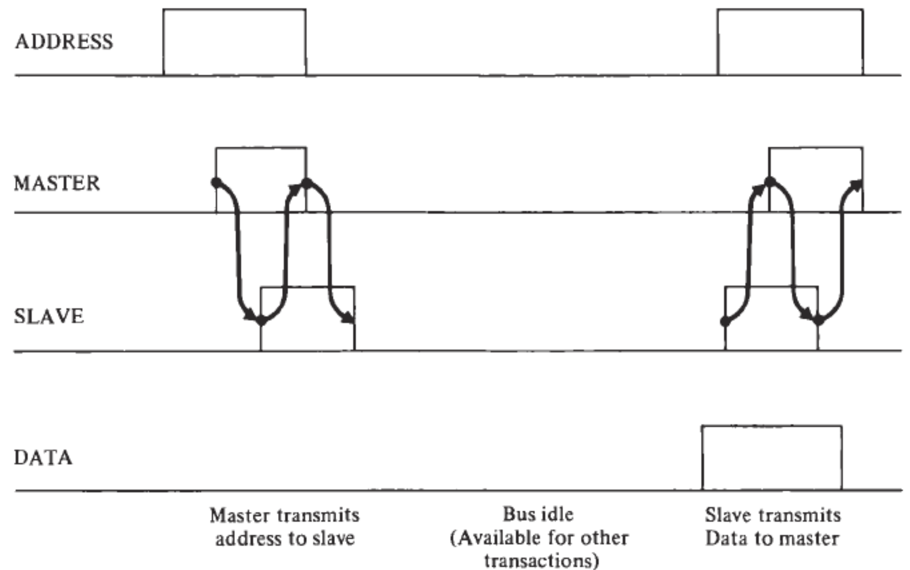


FIGURE 3.8 A split-cycle protocol READ transaction.

Clearly the performance of a split-cycle protocol depends on being able to use the bus time between the cycle halves for other transactions. Thus the protocol is most suitable for systems with multiple processors or multiple DMA devices on the bus; it makes little sense for low-performance systems. This type of protocol is used in high-performance minicomputers such as DEC's VAX-11/780, but has rarely been used in microcomputers until the introduction of the Intel iAPX-432 in 1981.

3.3 ARBITRATION PROTOCOLS

The purpose of arbitration has been discussed earlier, namely to guarantee conflict-free access to a bus. Bus arbitration is absolutely essential in systems that have two or more bus masters, and is not necessary for systems that have but a single master. But even in the latter case, the lines and logic required for arbitration are normally included in general-purpose modules so that these modules can be used in both contexts. Also, a system initially configured without DMA can be upgraded to a system with DMA, with the required arbitration facilities already in place.

One of the simplest possible arbitration techniques is called a *daisy chain*, and is shown in Fig. 3.9. The idea is that a single arbiter (the microprocessor itself in a single-processor system with DMA) has exclusive access to the bus until a request for access comes from a DMA device or other processor (identified as the small modules in the figure.) In response to a REQUEST signal, the arbiter issues a GRANT. This signal passes

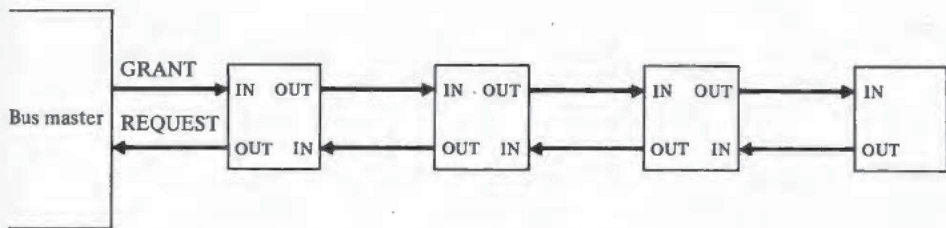


FIGURE 3.9 Daisy-chained bus arbitration (2 wires).

sequentially through the other potential bus masters. The first requesting module that receives GRANT takes control of the bus for one transaction. While that module has control of the bus, it does not pass the GRANT to the next module on the bus. Consequently, no other module has access to the bus.

Typical timing of this protocol appears in Fig. 3.10. The first transaction shows a particular module generating a REQUEST, and eventually receiving a GRANT. For this transaction, there is no REQUEST into the module, and no GRANT is passed on by the module. For the next transaction the module is inactive. It receives a REQUEST, which it repeats. When a GRANT appears later, it passes this on to the lower-priority modules. For the last transaction the module both generates a REQUEST and receives a REQUEST. The module maintains an active output on REQUEST through its bus transaction and through that of the lower priority device. When the GRANT reaches the module in response to the REQUEST, the module takes control of the bus and maintains an inactive output on GRANT. At the conclusion of its transaction, it passes GRANT down the daisy chain because the REQUEST input is still active.

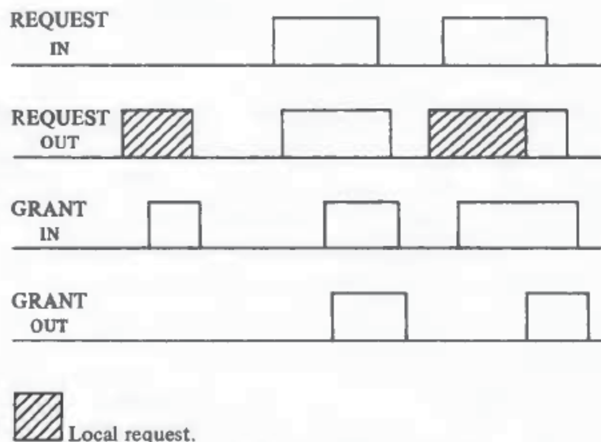


FIGURE 3.10 Daisy-chain timing.

The sequential flow of GRANT is crucial in this case. The protocol attempts to grant high-priority requests before low-priority requests, so that GRANT is routed to the modules in the order of priority. In essence, each module in succession is offered access to the bus; and the first that has a pending request accepts access.

Although the protocol appears to have all the desired characteristics, it has timing hazards that can lead to bus failures unless special precautions are taken to eliminate the hazards. Let us first investigate what problems exist, then show a popular 3-wire protocol that is hazard-free.

The arbitration protocol we have described appears to work correctly, because one module at a time receives the opportunity to take control, and therefore only one module can be granted bus control for any specific cycle. Once a module begins a cycle, that cycle must run to completion. If a higher-priority module wrests control of the bus away from a lower-priority module in the midst of a cycle, the aborted cycle may appear to be correct to the bus master or bus slave and will result in a communication failure. Therefore, a high-priority module must recognize that the lower-priority modules can be in one of three states, namely,

1. idle,
2. holding a pending request, or
3. actively controlling the bus.

The high-priority module can take control only if no lower-priority module is in the third state. The REQUEST/GRANT protocol, however, passes only one bit of information from the lower-priority modules. This bit by itself cannot distinguish among three different states. The critical distinction is between a pending request and active control of the bus. Therefore, to make the protocol safe each module uses the GRANT as well as the REQUEST signal to determine the state of the lower-priority part of the bus. Specifically,

1. if GRANT is low and REQUEST is high, there is at least one pending request, but no lower-priority module has active control of the bus, and
2. if GRANT is high and REQUEST is high, then a requesting module has been granted bus control and is currently conducting a transaction.

If a high-priority module generates a bus request while GRANT is high, it cannot take control of the bus. Safe arbitration requires that the module must see GRANT change from low to high *after* REQUEST is raised, and thus the leading edge of GRANT triggers the bus-control decision as the GRANT signal passes down the arbitration lines.

Edge-triggering on GRANT is necessary, but in itself does not provide complete protection from timing hazards. The protocol must ensure that the decision to take control of the bus is made sequentially, one module after another, and propagates in one direction on the bus. To see what happens when this rule is broken, consider what happens when the protocol in Fig. 3.10 is changed ever so slightly to violate the rule. Assume that when a module in control of the bus completes its transaction, that module passes GRANT on to the next lower-priority module, whether or not there is a REQUEST pending from that part of the bus. This protocol appears to be reasonable because a REQUEST from a

lower-priority module may be propagating up the bus at this very instant of time, and the requesting module may be able to take control when the GRANT arrives without having to wait until the next arbitration cycle. The timing hazard in this protocol appears in Fig. 3.11. Assume that the controllers are numbered 1, 2, and 3 in descending order of priority, and that we observe the bus with Controller 2 performing a transaction. Let us also assume that while Controller 2 is active, Controllers 1 and 3 are both inactive, and that both generate REQUEST signals shortly after Controller 2 terminates. Signals propagate in both directions from Controller 2. GRANT continues down the bus to the lower-priority modules, while REQUEST drops low and propagates towards the bus arbiter. As GRANT propagates to Controller 3, if this controller generates a request before the leading edge arrives, it will take control of the bus when it sees GRANT go high. Meanwhile, Controller 1 sees a high GRANT from the arbiter and a low REQUEST from the lower-priority modules on the bus. In this condition Controller 1 can assume that it is safe to take control of the bus; or, to be sure that it sees the leading edge of GRANT, the controller can output a low on REQUEST (repeating the input condition), then raise REQUEST (reporting its local bus request). The latter situation results in GRANT dropping in response to the low REQUEST and rising again in response to the high REQUEST. In either situation, Controller 1 has taken control of the bus while Controller 3 has control. A bus failure occurs.

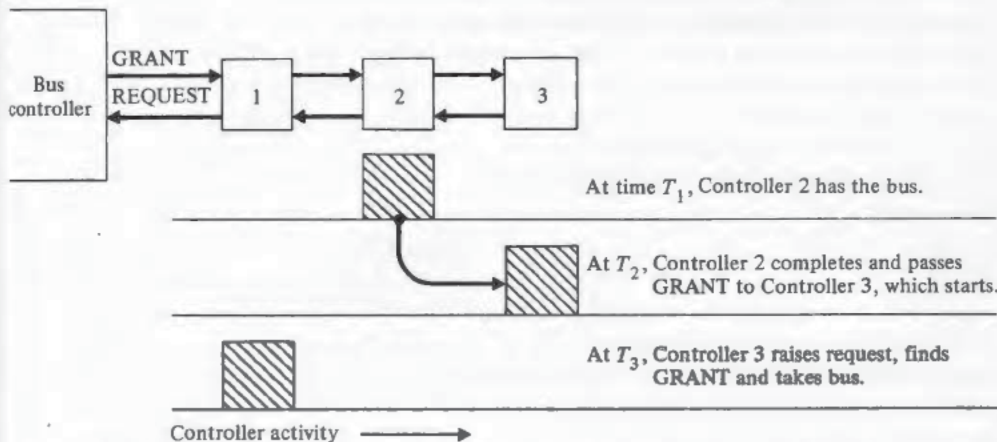


FIGURE 3.11 Possible timing hazards in an unsafe 2-wire protocol: If Controller 2 passes an inactive signal on REQUEST to Controller 1, Controller 1 can take the bus away from Controller 3.

The reason for the failure is that in this protocol Controller 2 issues bus grant signals in both directions on the bus when it completes its transaction. With bus grants going in both directions, the protocol violates the basic rule that control decisions have to be made one module at a time, progressing from module to module down the bus.

The protocol whose timing is illustrated in Fig. 3.10 is safe from this timing hazard because a controller does not pass GRANT to lower-priority modules unless that con-

troller sees an active REQUEST from these modules. Therefore one of two mutually exclusive conditions holds when Controller 2 completes its bus transaction in Fig. 3.11. Either Controller 3 has a REQUEST raised or it does not. The two different responses to these conditions are

1. When REQUEST is high from Controller 3, Controller 2 asserts GRANT to Controller 3 while also asserting REQUEST high to Controller 1.
2. When REQUEST is low from Controller 3, Controller 2 deasserts GRANT to Controller 3, and also deasserts REQUEST to Controller 1.

Since these two events are mutually exclusive, the protocol is safe. Nevertheless, poor logic design can cause short glitches on the REQUEST line output to Controller 1 if Controller 3 has its REQUEST high when Controller 2 completes its transaction. The REQUEST from Controller 2 in this case was formerly generated by Controller 2's local REQUEST for the bus. When Controller 2 has completed its activity, the output value of REQUEST changes over to the condition of being generated by the REQUEST from Controller 3. As this state change occurs, a short glitch on the REQUEST line output to Controller 1 proves disastrous. (Additional timing problems caused by Controller 3's changing its REQUEST OUT state almost concurrently with the completion of Controller 2's transaction exacerbate the hazard.) The glitch propagates toward Controller 1, which will take control of the bus when the glitch arrives. Meanwhile, the GRANT propagates from Controller 2 to Controller 3, which takes the bus when it receives GRANT. Should both controllers elect to take the bus, a conflict and bus failure is inevitable. While the conditions mentioned here appear to be somewhat contrived, they are quite realistic and demonstrate the pitfalls of careless logic design and failure-prone arbitration protocol.

One of the most popular methods for arbitration is a 3-wire method shown in Fig. 3.12. This scheme is similar to the arbitration scheme of the DEC PDP-11 Unibus (Digital Equipment, 1979). Two of the three lines are continuous bus lines, with modules having the ability to inject signals onto the lines or to read signals from the lines. One line is the GRANT line, which threads the modules sequentially, and is not a continuous bus line. The REQUEST line of the 2-wire daisy chain becomes a REQUEST and a BUSY line on the 3-wire daisy chain. With the two lines REQUEST and BUSY, we are able to distinguish among the three states mentioned earlier— idle, request pending, and active control.

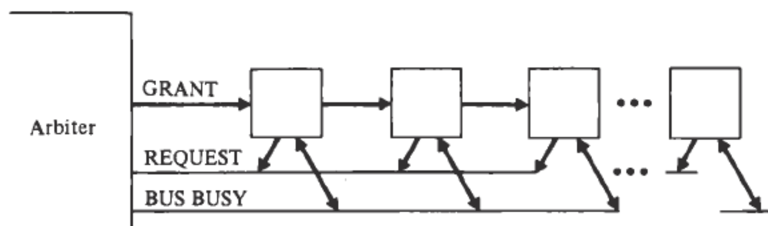


FIGURE 3.12 Safe daisy-chain arbitration protocol (3 wires).

The operation of this protocol is as follows:

1. When a controller has active control of a bus, it asserts BUS BUSY.
2. When a controller requires a bus cycle, it asserts REQUEST.
3. The arbiter transmits a GRANT signal when it detects a pending REQUEST and an inactive state on BUS BUSY. (If the arbiter is itself a bus master, such as a microprocessor, the arbiter can take one or more bus cycles when BUS BUSY falls before responding with a GRANT signal.)
4. A controller passes GRANT to the next controller if GRANT is received when the controller has no REQUEST pending.
5. A controller takes over the bus when
   i) it has a local request pending,
   ii) BUS BUSY is inactive, and
   iii) it detects the rising edge of GRANT.

For both BUS BUSY and REQUEST, we assume that the bus forms the logical OR of the outputs from the controllers. Most implementations use open-collector drivers for driving the bus, so that the low state must be the active state because this is the state that produces an OR function.

Typical bus timing appears in Fig. 3.13. We see here the interplay of the requests from two controllers, with Controller 1 having priority over Controller 2. In the first transaction, the bus is not busy when Controller 1 makes a request. At a subsequent point in time Controller 1 receives a GRANT, and then takes control of the bus without passing GRANT to the next controller in line. When taking over the bus, Controller 1 raises BUS BUSY to signify that it has the bus. At that point the arbiter removes GRANT. Now the bus is in use, and neither REQUEST nor GRANT is high.

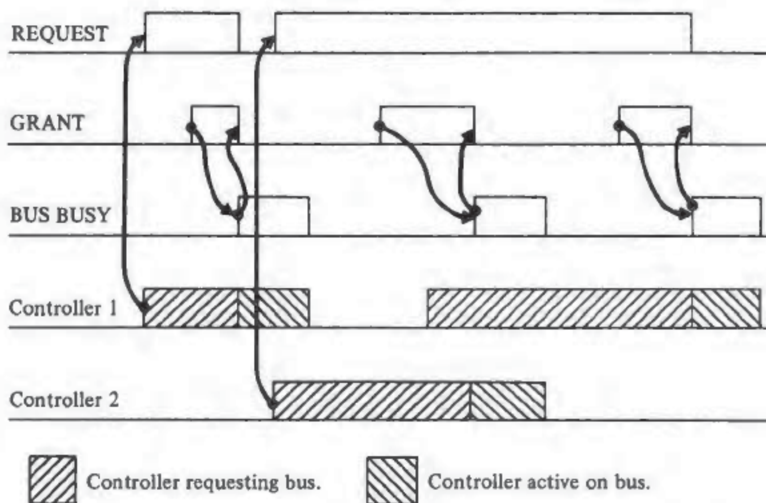


FIGURE 3.13 Timing for the 3-wire daisy-chain arbitration protocol.

Subsequently Controller 2 requires access to the bus and raises REQUEST, which propagates to the arbiter. The arbiter cannot grant access to the bus immediately because BUS BUSY is high. After BUS BUSY falls, the arbiter transmits GRANT to the controllers along the bus. Controller 1 passes this GRANT to Controller 2 because Controller 1 has no REQUEST pending. When Controller 2 receives the GRANT signal, the GRANT goes no further. Controller 2 raises BUS BUSY, lowers REQUEST, and takes over the bus. Meanwhile the arbiter removes GRANT and the arbitration cycle repeats.

Note what happens when Controller 1 requests the bus after passing GRANT to Controller 2. Because Controller 1 raises REQUEST *after* passing GRANT to the next controller in line, it does not remove GRANT when it raises REQUEST; moreover, Controller 1 does not take over the bus even though it sees GRANT high. The protocol prevents Controller 1 from taking the bus because the controller has missed the leading edge of GRANT. If Controller 1 responds to a level signal on GRANT instead of the leading edge, then Controller 1 could wrest control of the bus away from Controller 2 after Controller 2 initiates a transaction. This is an unsafe condition. Hence, the edge-sensitivity to the GRANT signal is essential for the protocol to be safe. Controller 1 eventually receives control of the bus, but this happens on the next arbitration cycle. When Controller 1 eventually takes control, the BUS BUSY signal first drops (Controller 2 completes its transaction), and then the arbiter issues a new GRANT that is accepted by Controller 1.

This protocol is safer than the 2-wire protocol for a number of reasons, although it can fail (as can any protocol) if events on the bus occur too closely in time. We take up this particular type of failure later in this chapter, but for the present we focus on the safety of the 3-wire arbitration protocol. To show the higher reliability of the 3-wire protocol, consider the failure of the 2-wire protocol. Recall that even though the 2-wire protocol is safe when glitch-free, with glitches present it becomes unsafe because Controller 2 can pass contrary information in opposite directions on the bus, and this information can enable modules on either side of Controller 2 to take control of the bus. For the 3-wire protocol, in order for both Controller 1 and Controller 3 to take control of the bus, they both have to see a low on BUS BUSY and a rising edge on GRANT. But GRANT is produced only after the low on BUS BUSY propagates forward to the arbiter. GRANT is not passed directly to Controller 3 from Controller 2. In the 3-wire protocol, the GRANT passes through all of the modules sequentially until the arbitration winner stops the propagation of GRANT. Hence, Controller 3 cannot elect to take control of the bus unless Controller 1 has first had the opportunity to do so.

The 3-wire protocol is insensitive to glitches in many instances. Suppose, for example, that Controller 1 raises its REQUEST just soon enough before GRANT passes through the controller to see the leading edge of GRANT, but the timing is so critical that a momentary pulse on GRANT propagates down the bus to the lower-priority modules. Note that GRANT passes through logic gates as it propagates down the bus, while BUS BUSY propagates in the same direction along a bus wire. In all likelihood BUS BUSY reaches the lower-priority modules before GRANT does. Hence, a requesting module that sees the brief GRANT pulse is likely to do so when BUS BUSY is high, which is an ille-

gal condition. Good logic design dictates that this condition be checked, and that a module that observes this condition be prevented from taking the bus.

The reliability of the 3-wire protocol is the reason behind its general acceptance in the industry. Almost all bus arbitration protocols for high-speed bus systems use the 3-wire system or a variation of it.

3.4 ASYNCHRONOUS TIMING DIFFICULTIES

In our discussion of arbitration we hinted at inherent difficulties in resolving asynchronous signals. The arbitration protocol relies on being able to tell if GRANT occurs before or after REQUEST. Is it always possible to make this decision correctly? If not, what is the failure mode?

The basic problem in resolving timing differences in two signals reduces to that of latching a single datum into a flip-flop. If the datum is present before the clock, the datum is latched successfully. If the datum is not present, the flip-flop latches a quiescent value (presumably a 0), and misses the datum. Every flip-flop and, in fact, every memory element must observe a datum on its input for at least a minimum time in order to copy that datum into memory in response to a clock pulse. This is equivalent to saying that the input signal must contain a minimum amount of energy in order to raise some input buffer above a threshold. The minimum time is usually expressed as a setup and hold time.

It is clear enough that a signal that satisfies setup and hold time constraints can be recognized successfully, but what happens if the constraints are violated? The results are unpredictable and disastrous. Chaney and Molnar (1973) show photos of oscilloscope traces of real devices whose input signals violate setup and hold times. Among the interesting results shown are as follows:

1. Flip-flops enter a ''metastable'' state in which the output lies about midway between a logic 0 and logic 1. The output stays in this condition for a variable and unpredictable amount of time, then relaxes unpredictably to either a logic 0 or logic 1.
2. Flip-flop outputs oscillate in phase with each other (violating the rule that the outputs are complementary) until they relax at complementary logic values an unpredictable time later.
3. Flip-flops stay in a stable state for a time longer than the worst-case transition time, then switch to their final state.
4. Flip-flops output a spurious brief pulse of unpredictable duration before assuming a steady state.

A typical failure mode is shown in Fig. 3.14. Mead and Conway (1980) address the same type of problem in the context of VLSI. Synchronization is a fundamental problem no matter how small or how fast the gate.

The implication of all of these problems is that we cannot depend on the flip-flop falling to a correct and consistent state, nor can we depend on the settling time to be within a maximum period of time. How then is it possible to design a safe, asynchronous system?

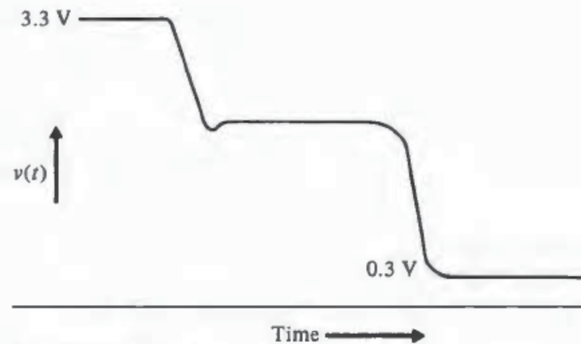


FIGURE 3.14 Output voltage as a function of time for a flip-flop in the metastable state.

The only way to be sure that a system is free from the clocking difficulties we have discussed is to use a single master clock from which all other timing is derived. At each flip-flop we can enforce the setup and hold-time constraints to be met by permitting signals to change only within certain safe periods of time. These safe periods are derived from the master clock so that each module that produces changes on its signals derives the safe period for those changes from the same time base as every other module. Because of propagation delays and skew, there are uncertainties in the edges of the master clock, but these can be taken into account when producing the windows that gate signals during safe periods. One way of gating signals safely is shown in Fig. 3.15. In this case, signal X changes within the setup and hold-time constraints for the Phase 2 clock but is known to be stable during the change of an earlier clock (Phase 1). So X drives a flip-flop gated from Phase 1. This guarantees that X is stable when the first flip-flop latches X, and the first flip-flop is stable when the second latches its value. As long as all signal changes can be gated from a single master clock, in principle every flip-flop change can be made safe from asynchronous timing hazards. But if signals can change in truly asynchronous fashion, as is the case at interfaces between two separate, individually clocked systems, then timing hazards are inherent. In this case, the designer can at best reduce the problem, but not eliminate it.

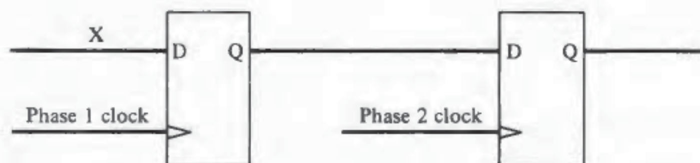


FIGURE 3.15 A safe procedure for gating a signal that is unstable during the Phase 2 clock setup period.

The method of Fig. 3.15 for guaranteeing that $\dot{X}$ is read only while X is stable is also used to synchronize X to the system when X can change at any time whatsoever. No problems occur if X is stable for the flip-flop setup and hold times of Phase 1. If X should change within the constraint times, the output of the flip-flop is unpredictable, and the flip-flop may enter the metastable state for an indefinite period of time. The idea is to have the edges of the Phase 1 and Phase 2 clocks far enough apart that the Phase 1 flip-flop is almost certainly in a stable state when Phase 2 triggers the second flip-flop. Whereas there is no guarantee that the first flip-flop will be stable at this time, the longer the period of time between Phase 1 and Phase 2, the more likely it will be that the first flip-flop will stabilize before Phase 2 occurs. For example, if the flip-flop is a 74S74, the stabilizing time should be at least 250 ns. For the bus systems we describe, the raising of REQUEST can occur at any time relative to GRANT. To avoid timing difficulties, both GRANT and REQUEST should be clocked relative to a master clock to guarantee that they cannot change within a critical time of each other.

3.5 INTERRUPT-REQUEST ARBITRATION

The arbitration schemes used to control access to a bus may be used for other purposes as well. Earlier in this textbook we described the functional details of interrupt-device identification, and we mentioned that some microcomputer systems incorporate a priority mechanism to resolve conflicts among interrupting devices. That priority mechanism is often implemented as a 2-wire or 3-wire daisy chain, using the essentially similar bus protocols, which we described, for gaining access to a bus.

The simplest form of device identification is shown in Fig. 3.16. This is a party-line system in which the interrupt request (IRQ) is the logical OR of the request signals generated by the devices. In responding to the IRQ request, the processor must first query

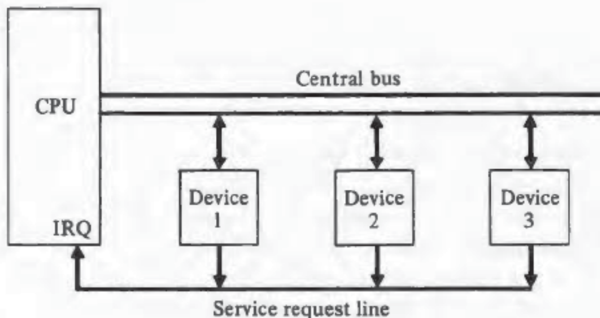


FIGURE 3.16 A party-line interrupt facility.

each device with a polling program to determine which is ready for service. Devices are normally polled in order of their priority, from the highest-priority device to the lowest. Because device polling is so time consuming, the vectored interrupt method has gained in popularity and acceptance. This scheme appears in Fig. 3.17. The scheme shown is essentially the 2-wire system, but it could as easily be the 3-wire system. The REQUEST line is the interrupt request line, and the GRANT line is the status line that indicates when the interrupt is honored by the processor. When the GRANT signal goes high, the device winning of the arbitration cycle places its device code (or some uniquely identifying integer) on the bus. The processor accepts the device code, and computes a starting address for the device-handler software from its device code. Thus, within a few machine cycles of honoring the interrupt, the processor can transfer control to software dealing specifically with that interrupting device. Polling takes a good deal of time, and severely degrades the performance of systems with many devices.

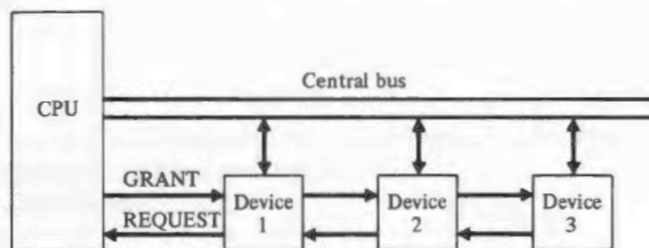


FIGURE 3.17 A daisy-chain priority scheme for a vectored-interrupt system.

3.6 EXAMPLES OF EXISTING BUS PROTOCOLS

In this section we examine implementations of the handshaking and arbitration protocols. The example of a synchronous protocol is drawn from the 6800 family of microprocessors. The PDP-11 and LSI-11 families use asynchronous protocols, with differences between the two families to account for differences in the number of wires in their buses. The 8085/8086 family of microprocessors provides examples of semisynchronous bus protocols.

The 6800 family of processors uses the synchronous protocol illustrated in Fig. 3.18. The example in the figure assumes a 1 MHz clock rate, but within the 6800 family are processors that operate up to 2 MHz, with corresponding reductions in the times shown. The 1 MHz clock has roughly a 50% duty cycle, with the clock's leading and trailing edges providing the timing points for latching or reading all bus signals. Note that the bus master places address and data on the bus at least 200 ns before the rising clock edge, and holds them for a least 40 ns after the falling clock edge. The setup time for READ and WRITE transactions requires that data be valid at least 100 ns before the falling edge of the clock. Among the signals that have the timing of address lines is the signal R/W L,

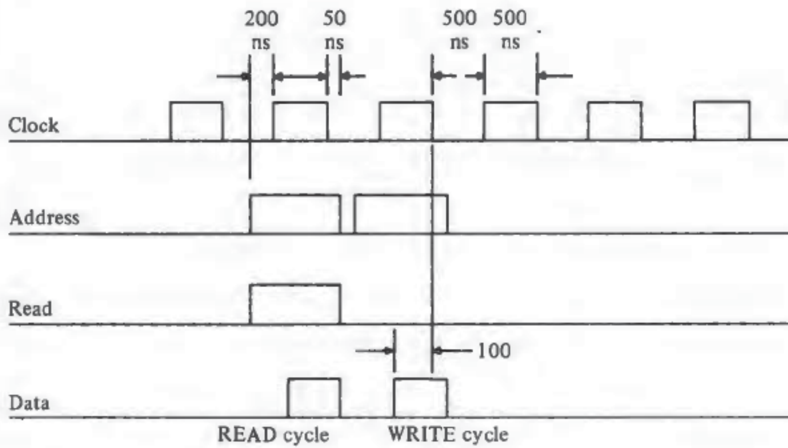


FIGURE 3.18 The 6800 bus timing: A synchronous bus.

which determines if the bus transaction is a READ or WRITE. Hence, at least 200 ns before the rising edge of the clock, the type of transaction is broadcast to all memories and ports on the bus. As we see later for the 8085 microprocessors and its descendents, the 8085 family follows a different procedure and uses separate READ or WRITE signals both to determine the direction of a transaction and to trigger it.

The PDP-11 Unibus provides an example of a purely asynchronous protocol. Figure 3.19 shows the timing for this bus, and gives minimum times for deskewing and decoding. Addresses must be placed on the bus at least 150 ns before raising MASTER, to allow for 75 ns decoding and 75 ns deskewing delays. The slave responds by raising SLAVE as soon as the data lines hold valid information (for READ) or have been latched (for

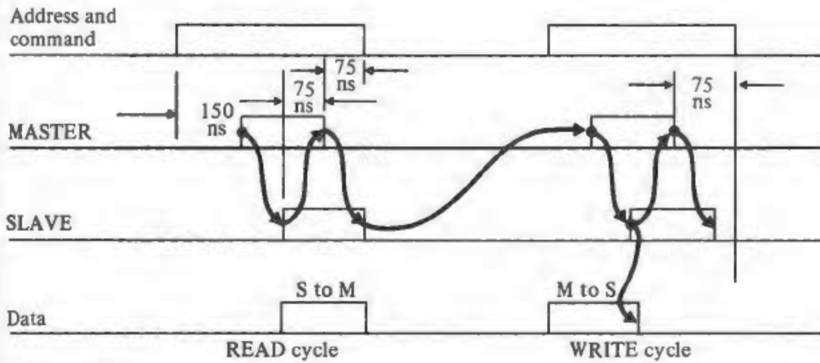
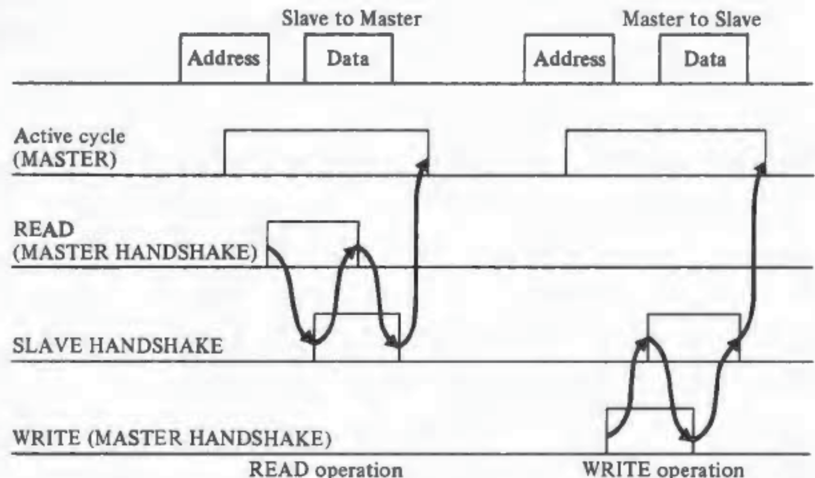


FIGURE 3.19 PDP-11 unibus timing; An asynchronous bus.

WRITE). Thereafter most transitions are spaced at least 75 ns apart to allow for deskewing. For a READ, the master waits 75 ns after seeing SLAVE high before latching data, and waits 75 ns after dropping MASTER before removing the addresses from the bus. For a WRITE, the master drops MASTER without a deskewing delay when it sees SLAVE high, but holds address and data valid on the bus for 75 ns after dropping MASTER.

The general principles of asynchronous handshakes are easily extended for more complex situations. Figure 3.20 illustrates how to multiplex addresses and data on one set of wires using an asynchronous handshake with one extra control wire. This is the technique used in the DEC LSI-11 Q-bus. Each bus transaction consists of two cycles, one in which an address is transmitted from master to slave, and the second in which data are transferred, either from master to slave or from slave to master for, respectively, WRITE and READ transactions. The beginning of a cycle occurs when addresses are placed on the bus, followed by one deskew and decode time later (150 ns) with the raising of the MASTER signal. This signal stays high through both cycles of the transaction. The bus master leaves the address on the bus for at least 100 ns to permit the selected slave to prepare for the subsequent data cycle. There is no handshake on this part of the cycle, so all slaves must respond to address broadcasts within a fixed minimum time, much the way that slaves on synchronous buses have to respond within one clock cycle. The asynchronous portion of the cycle depends on whether the cycle is a READ or WRITE. For a READ the master raises the READ HANDSHAKE signal, and thereafter the SLAVE HANDSHAKE and READ HANDSHAKE perform a fully interlocked handshake. The falling edge of SLAVE HANDSHAKE drops the MASTER signal, and the cycle ends. (Names of signals used here do not follow the naming conventions of the manufacturers, but the reader can easily identify similar names and functions.)



A WRITE cycle is similar to the READ cycle, with the exception that the master places the data on the bus first, and then raises WRITE HANDSHAKE. All subsequent transitions are fully interlocked, and the transaction closes by dropping MASTER in response to the fall of SLAVE HANDSHAKE. The figure does not show the delays inserted in the protocol for deskewing. These are all typically between 75 and 200 ns, depending on the transition. The exact details are unimportant for this discussion, but are given in detail in the *PDP-11 Bus Handbook* (1979).

The 8085 timing shown in Fig. 3.21 is an example of a semisynchronous bus that exhibits bus multiplexing similar to that of the LSI-11 Q-bus. The timing of the bus is somewhat unusual in that the clocking information is contained on the control lines. All bus timing is relative to the rising and falling edges of the control signals, so that the clock itself does not necessarily have to be transmitted on the bus. It is available, however, and can be used or ignored as the designer chooses.

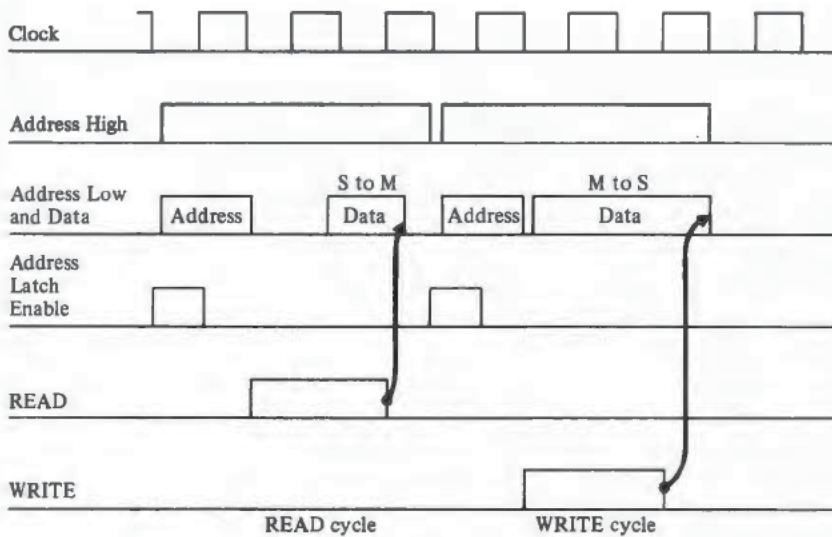


FIGURE 3.21 The 8085 bus timing: A semi-synchronous multiplexed bus.

For this bus, a cycle begins on a falling clock edge with the master raising the signal ALE (Address Latch Enable) while outputting the addresses onto the bus. ALE falls with the rising edge of the bus clock, and triggers external circuitry to latch the address currently present on the bus. The address lines remain stable until the next falling edge of the clock, providing the necessary hold time. The low byte of the address is removed from the multiplexed bus lines when the clock falls to complete its first cycle. This brings us to one clock cycle after the beginning of the transaction. The activity for the next clock depends on whether the transaction is a READ or WRITE.

For a READ, the master asserts the READ signal, holds this signal steady for 1½ clock cycles, then latches data from the slave with the falling edge of READ (approximately concurrent with the rising edge of the clock during the third clock cycle). Hence the slave has 1½ clock cycles to respond to the bus command, and the specific operation is triggered by the active edge of READ. Data should be stable on the bus for one hold time after the fall of READ, which falls in the last ½ cycle of the third clock period. A new memory operation then begins with the falling edge of the clock at the close of the third cycle. Note that only the low address is multiplexed with data. The high address is stable during the entire memory operation. Note also how the clock transitions encoded on the control lines provide separate and distinct times for each bus activity. This guarantees that there is a brief period between the disabling of one set of bus drivers and the enabling of another set.

The WRITE transaction differs from READ at the beginning of the second clock cycle. Here the bus master removes the low address and replaces it with data while asserting the WRITE signal. The leading edge of WRITE triggers a bus transaction that lasts 1½ clock cycles. WRITE drops approximately at the rising edge of the third clock, which notifies the slave that the master has completed its portion of the cycle. The memory operation ends with the falling edge of the third clock. Data and addresses remain on the bus for at least one hold time after the fall of WRITE.

As shown in the figure, both READ and WRITE take three clock cycles to complete. However, these are minimum times. Operations are extended an integral number of cycles depending on the state of the slave signal READY. The bus master reads the READY signal (not shown in the figure) ½ cycle after asserting READ or WRITE, which is ½ cycle before the last full clock period of the transaction. If the READY signal is asserted, then the next cycle is the last. If READY is low, then the master extends the cycle by one WAIT cycle, and reads READY again one cycle later. Because of setup time delays, the slave has to raise or lower READY early enough to reach the master about 100 ns before the clock edge rises in the second clock period. To lengthen a transaction, the slave can hold READY low just long enough to give time for the slave to respond to the command. If READY is used, a separate clock line on the bus simplifies the interface between the slave and the bus because it provides a recurrent edge from which READY timing can be derived.

The difference in philosophy between the 6800 and 8085 families is rather interesting and disconcerting when building "hybrid" systems containing chips from both families. Peripherals in the 6800 family need to have READ/WRITE information stable before the rising clock edge triggers a transaction. These peripherals will not operate correctly with 8085-type processors unless WAIT cycles are inserted. When an 8085 issues READ or WRITE, the signal itself is intended to trigger a transaction. But the 6800 peripheral has to use this signal to set up the direction of the transaction, and uses a delayed version of the signal to trigger the transaction. The delay forces a WAIT cycle.

But the situation depicted here is not symmetric. An 8085 peripheral can easily be connected to a 6800-type system. The READ or WRITE produced by the 6800 is ANDed with the clock to provide the trigger required by the peripheral. The unexpected difficulty

in interfacing one type of peripheral to another type of microprocessor stems from using READ and WRITE to perform two functions in the 8085 family, but only one in the 6800 family. The polarity of the signal provides the direction of the transaction in both families, but the timing of the signal triggers a transaction only in the 8085 family.

The problem described here has been pointed out by Wakerly (1979) and Borrill (1981). It is possible to design around the problem, as Wakerly points out, by using an address bit to distinguish a READ from a WRITE, since the address becomes stable well in advance of the READ/WRITE control-line activation. Thus a peripheral-chip register can have two addresses, one even and one odd. The even address results in a READ of the register, and the odd address results in a WRITE to the register. Since the address bit replaces the READ/WRITE control line at the peripheral chip, it is entirely possible to issue such nonsensical commands as LOAD from the WRITE address and STORE to the READ address. The latter case is particularly troublesome because the microprocessor and the peripheral will both attempt to put data on the bus concurrently, and the bus interface logic may be damaged by this action.

Not only has the industry not settled on one timing method or the other as a standard, but Intel and Zilog use both. Intel uses both methods in the 8048 family. Zilog's Z80 follows the 8085 method, but the later Z8000 follows the 6800. It is unlikely that this switch is indicative of a move to standardize across the industry, and we expect to see differences in READ and WRITE methodology throughout the 1980s.

3.7 EXAMPLES OF BUS ARBITRATION

Bus arbitration for the synchronous 6800 bus is undoubtedly the simplest arbitration to implement. Timing considerations are shown in Fig. 3.22. A DMA request results in the processor entering the HALT state and issuing a GRANT. The GRANT remains high until the halt request is removed. Since the halt request is sampled by the processor just be-

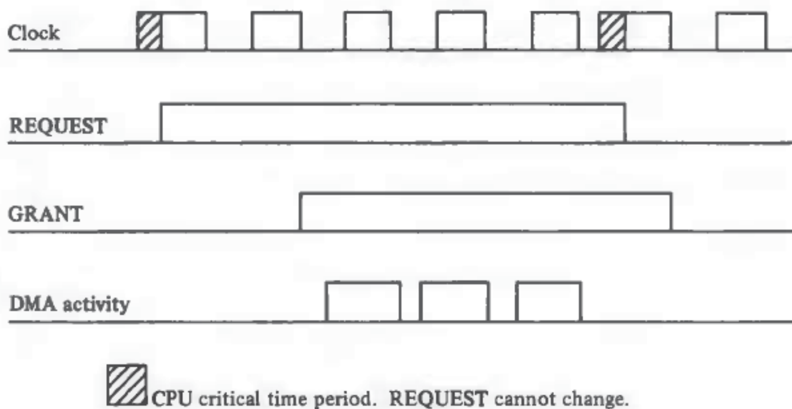


FIGURE 3.22 The 6800 arbitration timing: A 2-wire daisy-chain protocol.

fore the clock edge rises, the REQUEST signal must be stable during this period. The figure shows that period as a forbidden transition region, with REQUEST changing immediately after the clock rises. The processor honors the REQUEST at the conclusion of a clock cycle, but not necessarily the one following the REQUEST. The processor completes whatever instruction is in progress when it discovers REQUEST high, and raises GRANT at the end of the last clock cycle of that in-progress instruction. The figure shows three cycles dedicated to DMA activity with REQUEST dropping in the fourth cycle. After the fourth cycle, GRANT drops and the processor resumes control.

Several methods are available to implement an arbitration scheme for this processor. The processor requires only two wires, but a 3-wire protocol is compatible with these two control lines. Obviously, REQUEST should be the OR of the individual master requests. To facilitate this capability, REQUEST is actually active in the low state for the 6800, so that it can easily be produced with open-collector drivers. GRANT is intended to thread the masters in order of priority. For the 3-wire protocol, a third signal for BUS BUSY visits all masters, but is not used by the 6800 itself. BUS BUSY protects the processor from being in the HALT state indefinitely. A timer or one-shot connected to BUS BUSY can remove the REQUEST signal if BUS BUSY is inactive for a sufficiently long period. Otherwise BUS BUSY serves the functions described earlier in this chapter.

The DEC PDP-11 Unibus improves upon the arbitration protocol described in this textbook by adding a fourth wire that enables arbitration for the next cycle to be overlapped with the bus transaction for the present cycle. The idea behind this form of arbitration is illustrated in Fig. 3.23. Arbitration times appear on the upper line; data transfer times appear on the lower line. Note that while the data transfer for the first transaction is active, the arbitration for the second transfer takes place. Similarly, the third arbitration is completed during the second transfer. But the 3-wire daisy-chain timing that we described earlier does not permit this overlap, because GRANT does not rise unless the bus is free. The Unibus protocol eliminates this difficulty by splitting the BUS BUSY function into two lines. One signal line in this protocol, also called BUS BUSY, indicates whether a transaction is active on the bus, and serves only this purpose. The second line, ACKNOWLEDGE, is used to respond to the GRANT signal. Both of these functions are served by the one line BUS BUSY in the 3-wire protocol because the response to a GRANT is an active BUS BUSY in that protocol. Dividing the two functions into two separate lines permits the overlapping arbitration and transfer activities.

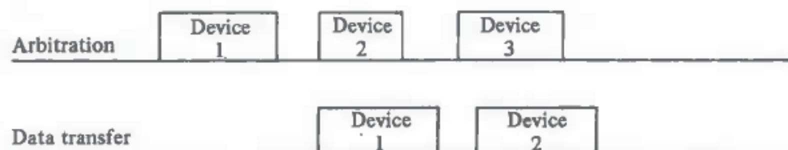


FIGURE 3.23 Overlapping of arbitration and data transfer.

Details of the Unibus arbitration methodology are shown in Fig. 3.24. Note that REQUEST starts an arbitration cycle, and that the REQUEST for the second cycle can begin during the first transaction. A GRANT appears at later time, and propagates down the bus. The winner of the arbitration responds with ACKNOWLEDGE, and removes its REQUEST (although other REQUESTs may still be present). This winner cannot take control of the bus immediately, but must wait until BUS BUSY goes low. To prevent another arbitration from beginning at this point, ACKNOWLEDGE remains high until the master is able to take control of the bus. When BUS BUSY falls, the master for the second transaction drops ACKNOWLEDGE and raises BUS BUSY as the master initiates the second transaction. The fall of ACKNOWLEDGE triggers a new arbitration cycle. A GRANT is issued by the arbiter if any REQUEST is observed on the bus, and the cycle repeats. Note that in Fig. 3.24 three different masters assert signals on REQUEST, ACKNOWLEDGE, and BUS BUSY lines, with each signal identifying the master that asserted it.

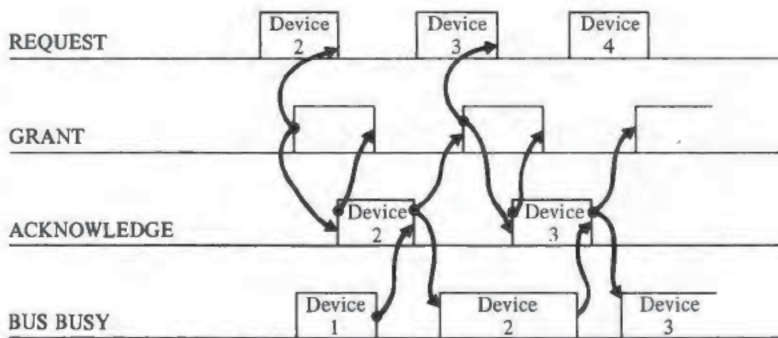


FIGURE 3.24 PDP-11 Unibus arbitration timing: A 4-wire daisy chain.

The arbitration protocols for most microprocessors are designed to be used with tri-state drivers on the address and data lines. To protect these drivers from "bus-fights," when two or more drivers in opposite polarity drive a single line, it is necessary to provide a "guard" time after one set of drivers turns off and another set turns on. The guard time is incorporated in the handshakes for the various microprocessors by providing a brief period between READ and WRITE transactions, and between a master's use of a bus and a slave's access to a bus for a READ transaction. For example, the timing for the 6800 microprocessor raises addresses and data about ¼ cycle after the beginning of a clock period. This provides about ¼ cycle after the last clock period for the transition between sets of drivers on the bus. Similarly, the 8085 protocol provides about ½ clock period between a slave's access to the bus and the master's access on either side of the slave.

Arbitration protocols must account for tri-state drivers as well as for controlling the access to a bus. The IEEE-796 bus (Intel Multibus) arbitration protocol is an interesting

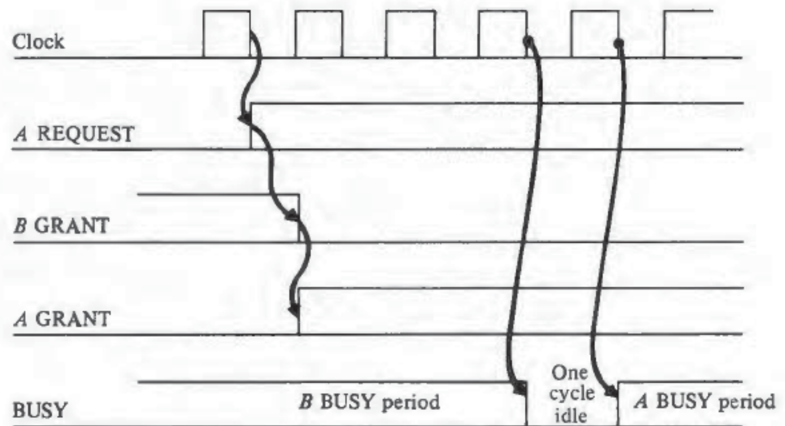


FIGURE 3.25 An IEEE-796 bus arbitration cycle. The idle cycle protects the tri-state drivers.

example of how this can be done. Figure 3.25 shows the protocol with two masters on the bus, Master *B* (which currently is active on the bus) and Master *A* (which currently requests the bus). We assume that *B* has lower priority than *A* and holds the bus, because GRANT is high in the daisy-chain connection to *B*. When *A* raises its REQUEST (which appears on the bus at the same time as the clock signal falls), the GRANT signals on the bus change so that GRANT propagates only up to the highest priority requester. In this case *A* is that requester, so it receives the GRANT signal, and *B*'s GRANT goes low. Then *B* has to relinquish control of the bus at the earliest opportunity. If *B* is a processor, it may complete the several clock cycles of a memory-cycle transaction. Other types of masters may need to complete several memory cycles before reaching a stopping point. At the trailing edge of the last clock in its activity cycle, *B* removes BUSY, and BUSY becomes inactive for one full cycle. Then *A* takes control of the bus, asserts BUSY, and arbitration has been completed. The full clock cycle of inactivity provides the dead time required between sets of bus drivers.

In this protocol, GRANT may be changed several times before a bus is relinquished. As each new REQUEST is posted, the chain of bus masters update their GRANT outputs on the daisy chain. The update has to take place within one cycle, and the master with GRANT asserted when BUSY drops to 0 is the master that takes control of the bus.

In closing, we present one final interesting method for arbitration and device identification. This arbitration method not only selects the highest priority requester when multiple requests are present, but it returns the ID of that requester on the bus back to the arbiter (usually the processor). Hence this method of arbitration is quite suitable for device-identification after a processor interrupt. An additional property of this arbitration method is that all arbitration signals are open-collector, and control lines are continuous conduc-

tors to which the bus slaves attach. This is in sharp contrast to the daisy-chain arbitration scheme that requires the bus grant to be discontinuous at each slave so that each slave makes an independent decision about whether to pass the grant to the next in line or to hold the grant for its own bus transaction. The IEEE Standard version of the S-100 bus (IEEE-696.1/D2, 1979) uses this technique for the arbitration of DMA transfer requests.

The bus interface for this application is shown in Fig. 3.26, and is adapted from the IEEE-696 Standard and Borrill (1981). (The notation OC indicates an open-collector gate.) The idea of the interface is to assert HOLD L to request access to the bus, and to await a response on pHLDA, which is the acknowledge signal returned by the processor or bus arbiter. The interface asserts a request by setting the HOLD flip-flop when three conditions occur simultaneously:

1. The device needs access to the bus (BREQ is high).
2. The bus grant has not propagated past this device (pHLDA is low).
3. No other request is active on the bus (the control line HOLD L is passive high).

The three-input NAND that drives the S input of the HOLD flip-flop generates an active low signal when all three conditions hold. If a high-priority request comes *after* a low-priority request, the high-priority request must wait for the low-priority request to be removed before it has access to the HOLD L control line. This guarantees that new requests cannot come at arbitrary times. (Arbitrary changes can produce brief glitches and metastable states.) At the beginning of a new arbitration cycle there are, in general, several outstanding requests, all of which may gain access to HOLD L. The priority-arbitration logic we now describe selects the highest-priority request of those that actively assert HOLD L.

When a device gains access to HOLD L by setting the HOLD flip-flop, it simultaneously sets the ASSERT ID flip-flop. This flip-flop energizes the four ID lines. These lines are compared to the open-collector priority lines to determine whether there is a higher-priority requester. The logic is essentially that of a borrow look-ahead circuit of a binary subtracter. Each ID is a 4-bit binary ID, and high numbers have priority over low numbers. The four lines DMA0 L through DMA3 L carry the complement of the winning ID. Each interface generates four active-low borrows, B0 L through B3 L, by subtracting its ID from the ID currently on the bus. The 4-input NAND that generates HIGH PRIORITY L requires that all borrow signals be inactive high, which is satisfied only by the requesting interface with the highest ID. HIGH PRIORITY L is gated together with HOLD L and pHLDA L at a 3-input NOR to be sure that the request and the bus grant are still active before signaling TAKE BUS to the device.

Losing requesters generate a 0 on the D-input of the HOLD flip-flop. This value is gated into HOLD when the rising edge of pHLDA reaches the interface. Hence, only the winner continues to assert HOLD L after the grant propagates the length of the bus. After the winner completes its bus activity, it removes BREQ, which resets the HOLD flip-flop. Since this is the only HOLD outstanding, HOLD L becomes inactive, and the processor regains control of the bus. Immediately after pHLDA is deasserted, the losing requests and any new requests bid for the bus through another arbitration cycle.

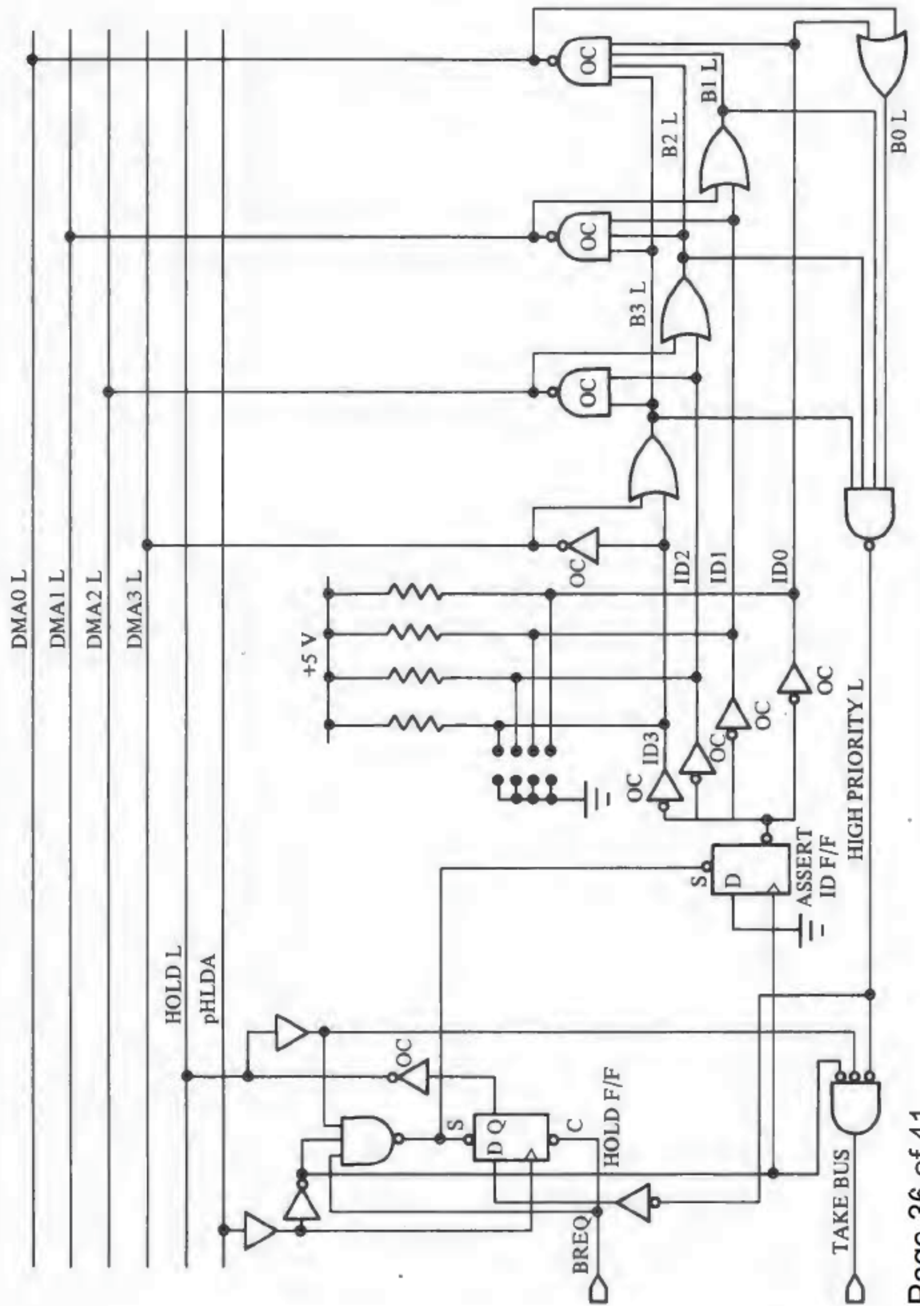


FIGURE 3.26 Priority arbitration logic for the IEEE-696 (S-100) bus.

OTHER READING

A truly comprehensive study of busing that greatly influenced this chapter is the work by Thurber *et al.*, (1972). This treatment covers a very large variety of bus implementations, including many handshake and arbitration protocols not covered in this chapter. Readers will find this to be an excellent source of background information.

The importance of buses and bus standards has only gradually been felt by the computer industry. In the infancy of computers the majority of manufacturers supplied all of the devices that attached directly to the processor/memory bus. Thus this interface was usually proprietary to the manufacturer, and it was not possible to connect "foreign" devices to the bus. Foreign devices, if attached at all, were mostly connected to DMA channels.

Digital Equipment Corporation played a dominant role in fostering the user's ability to interface user-owned devices to its computers. The DEC PDP-1, and later the PDP-8, were its early offerings that supported user interfacing. The introduction of the PDP-11 and the Unibus with memory-mapped I/O had a very a large impact on the idea of using a well-defined bus as a standard digital interface. Eventually the microprocessor industry moved toward the bus as a common interface for microprocessors, memories, and peripheral chips.

The S-100 bus was introduced by Altair Corporation in one of the first personal microcomputers. This bus provided a means for interconnecting processor, memory, and I/O boards in a very general and flexible manner. Almost immediately after Altair's computer was introduced in the mid-70s, dozens of manufacturers marketed what were intended to be "compatible" board-level products that connected directly to the S-100 bus. But the bus had not been standardized, and many products from different manufacturers simply did not work with each other. As part of the IEEE Computer Society standardization effort, a group of volunteers took on the difficult problem of standardizing the S-100 bus. The culmination of this work is the draft standard IEEE Task 696.1/D2 (1979) that incorporates extensions of the bus to 16-bit processors and includes the interesting arbitration protocol described in this chapter.

The need for bus standards is now well-recognized, and the bus literature abounds with information. The Unibus and LSI-11 bus (Q-bus) are thoroughly described in the PDP-11 Bus Handbook (Digital Equipment, 1979). Levy's chapter on busing (1978) brings the discussion up to date with respect to buses on the VAX computer. The references about the DEC bus standards are an excellent source of information on practical implementations of buses that complement the general treatment of this chapter. Borrill (1981) analyzes several concurrent efforts concerning bus definitions and standards.

In dealing with the problem of synchronization, we cited the article by Chaney and Molnar (1973). Until that article appeared, very little concrete evidence existed about the metastable state. It is very difficult to observe the metastable state in a repeatable way, so that when synchronization failures occur it is often difficult to analyze what happened. Fortunately, the problem is much better understood today. Mead and Conway (1980) review the problem for VLSI designers, and show ways of designing safe VLSI chips.

TABLE E–3.1 Test Programs for Bus Analysis

68XX Family			
	LDX	#$1000	SET THE X-REGISTER TO POINT TO MEMORY.
	CLR A		CLEAR THE ACCUMULATOR.
LOOP	COM A		COMPLEMENT THE ACCUMULATOR.
	STA A	0,X	STORE THE ACCUMULATOR.
	LDA A	0,X	RELOAD THE ACCUMULATOR.
	BRA	LOOP	

8080 and 8085			
	LXI	H,1000H	SET THE H-L REGISTER TO POINT TO MEMORY.
	XRA	A	CLEAR THE ACCUMULATOR (A := A EXCLUSIVE OR A).
LOOP:	CMA		COMPLEMENT THE ACCUMULATOR.
	MOV	M,A	STORE THE ACCUMULATOR.
	MOV	A,M	RELOAD THE ACCUMULATOR.
	JMP	LOOP	

8086 and 8088			
	MOV	BX,1000H	SET THE B REGISTER TO POINT TO MEMORY.
	MOV	AL,00H	CLEAR THE ACCUMULATOR
LOOP:	NOT	AL	COMPLEMENT THE ACCUMULATOR.
	MOV	BX,AL	STORE THE ACCUMULATOR.
	MOV	AL,BX	RELOAD THE ACCUMULATOR.
	JMP	LOOP	

65XX Family			
	LDA	$00	CLEAR THE ACCUMULATOR.
LOOP	EOR	$FF	COMPLEMENT THE ACCUMULATOR.
	STA	$1000	STORE THE ACCUMULATOR.
	LDA	$1000	RELOAD THE ACCUMULATOR.
	JMP	LOOP	

Z80			
	LD	HL,1000H	SET H-L TO POINT TO MEMORY.
	XOR	A	CLEAR THE ACCUMULATOR.
LOOP:	CPL		COMPLEMENT THE ACCUMULATOR.
	LD	(HL),A	STORE THE ACCUMULATOR.
	LD	A,(HL)	RELOAD THE ACCUMULATOR.
	JP	LOOP	

EXPERIMENTS

3.1 Table E–3.1 gives a very brief program for many different microprocessors. After initialization this program loops through a sequence of instructions that write to and read from a selected memory location. Data written alternate between 0s and 1s on successive passes through the loop.

a) Select a microprocessor with a synchronous or semisynchronous bus. (A microprocessor from the 68XX, 808X, Z80, or 65XX families is satisfactory.)

b) Describe the bus activity on a clock-cycle–by–clock-cycle basis as shown for the MC6800 in Table E–3.2. The table gives the value of each memory bus signal as a function of time, and indicates in the comments what happens at each cycle. The analysis should be made on the four instructions within the loop only, and should not include loop-initialization instructions. Use the manufacturer's reference material to determine what happens during each cycle of the execution of each instruction; or, based on your understanding of the microprocessor, estimate what the bus activity is. In Table E–3.1, the first instruction of the loop starts at address 2004_{16}.

TABLE E–3.2 Bus Transaction Timing

CYCLE	ADDRESS	DATA	R/W	VMA	COMMENTS
1	2004	43	1	1	Read opcode for COM A.
2	2005	A7	1	1	Read next instruction.
3	2005	A7	1	1	First cycle of STA 0,X, read opcode.
4	2006	00	1	1	Read offset.
5	1000	XX	1	0	No memory cycle; no data on bus.
6	1000	XX	1	0	No memory cycle; no data on bus.
7	1000	XX	1	0	No memory cycle; no data on bus.
8	1000	DD	0	1	Store data, changing every other cycle.
9	2007	B6	1	1	Start of LDA 0,X; read opcode.
10	2008	00	1	1	Read the offset.
11	1000	XX	1	0	No memory cycle; no data on bus.
12	1000	XX	1	0	No memory cycle; no data on bus.
13	1000	DD	1	1	Read data.
14	2009	20	1	1	First cycle of BRA LOOP; read opcode.
15	200A	F9	1	1	Read offset.
16	200B	XX	1	0	No memory cycle; no data on bus.
17	2004	XX	1	0	No memory cycle; no data on bus.

c) Connect a dual-channel oscilloscope to the microprocessor. The external sync signal should be connected to the microprocessor control line that asserts WRITE. This signal is asserted once per loop. One channel of the oscilloscope

should be connected to the bus clock that is to be used as a timing reference. The second channel is left disconnected, and will be used to probe various bus signals.

d) Load the program in the microprocessor and initiate execution. Set the time scale to display two to three full executions of the loop. If the oscilloscope has a continuously variable time scale, you can expand the display so that one full execution of the loop just fits the display area. Probe each bus control line and one address and one data line. As each line is probed, identify the beginning of the program loop, and verify that the bus signal follows your calculations. Explain any discrepancy.

e) Expand the timing of the display so that one memory cycle just fills the screen. Select a READ cycle, and then select a WRITE cycle. For each of these cycles find the address setup time and the address and data hold times. Compare these times with the specifications for the microprocessor bus.

3.2 Repeat Experiment 3.1 for a microprocessor or minicomputer with a fully interlocked asynchronous bus. Suitable candidates are the PDP-11, LSI-11, or MC68000.

PROBLEMS

3.1 Analyze the Unibus arbitration protocol under the following conditions: The bus has three devices on the line and Device 2 (the second on the line) is currently active on the bus. Assume that sometime between this point in time and the conclusion of the next arbitration cycle, both Devices 1 and 3 raise requests. An arbitration error will occur if Device 3 is granted the bus and initiates a transaction, and subsequently the request is aborted by Device 1 when it takes over the bus. If Device 3 requests sufficiently earlier than Device 1, it should obtain the bus. If Device 3 requests sufficiently late with respect to Device 1, Device 1 should obtain the bus. In between, the protocol must grant the bus either to Device 3 or Device 1, but it must be a safe protocol so that no matter which device gets the bus, that device is assured that it will hold the bus to the completion of its request.

a) Give a timing analysis that shows the bus arbitration is safe if the devices must observe the change in bus grant signal from NO GRANT to GRANT in order to gain access to the bus.

b) Give a timing analysis that shows the arbitration to be unsafe if the devices can gain access to the bus simply by observing GRANT without necessarily observing the change from NO GRANT to GRANT.

3.2 There are slightly different timing requirements for a tri-state bus than for an open-collector bus: A tri-state bus cannot have two or more tri-state drivers active simultaneously, whereas there is no corresponding problem for the open-collector bus. Consider a generic, fully interlocked asynchronous bus, and observe what happens for sequences of operations such as READ/READ, READ/WRITE, etc. For what sequences is it necessary to separate the operations with some idle time on a tri-state bus, when no such time is required for an open-collector bus?

3.3 The Unibus has separate arbitration lines for DMA requests and interrupt requests. There is a single daisy-chain priority for DMA, but there are four distinct daisy-chain priority lines for interrupts. The four lines are prioritized so that if interrupt requests occur on two or more chains, a request on the chain with the highest priority is the one that is recognized. On any daisy chain, the requests are prioritized by the electrical connection because the interrupt acknowledge signal is passed from device to device. All four interrupt daisy chains and the DMA daisy chain pass through each unibus interface.

a) Give a concrete example in which the four daisy-chain system of interrupt priority is more versatile or powerful than a single daisy-chain interrupt system. Note that at any given time the pending requests on the four daisy chains are totally ordered, so that those requests can be acknowledged in precisely the same order if the four daisy chains are connected to form a single daisy chain by linking them end-to-end from highest to lowest. (*Hint*: all four interrupt chains visit each interface. Would this be the case if there were only one interrupt chain? How might an interface make use of the multiple chains?)

b) Consider a typical application for a small computer that has two DMA controllers that operate at 300 K-bytes per second, and six low-speed devices that operate under interrupt control. A low-speed device operates at a maximum speed of 1 K-byte per second. The DMA channels transfer data in 4 K-byte bursts, and a bus arbitration is required for each byte transferred. The channels post interrupt requests at the end of each burst. When the computer is operating at maximum I/O capacity, how many bus arbitrations and interrupt arbitrations occur per second? Are these rates consistent with the fact that the interrupt-request daisy chains together have about the four times the bandwidth of the one DMA daisy chain?

c) Sketch the layout of an alternative approach to the four daisy-chain system that interposes an intelligent interrupt-request arbiter between the computer and the I/O interfaces. This arbiter interfaces the processor with four daisy-chain interrupt-request lines, as required by the Unibus. However, only a single daisy chain that visits all devices extends from this arbiter. The idea is that the devices post interrupt requests on the single daisy chain, and the arbiter somehow converts the requests to four different levels of request. Describe how to construct an arbiter that mimics the functions and timing of the four daisy-chain system so well that the processor cannot easily tell that the devices are connected to a single daisy chain.