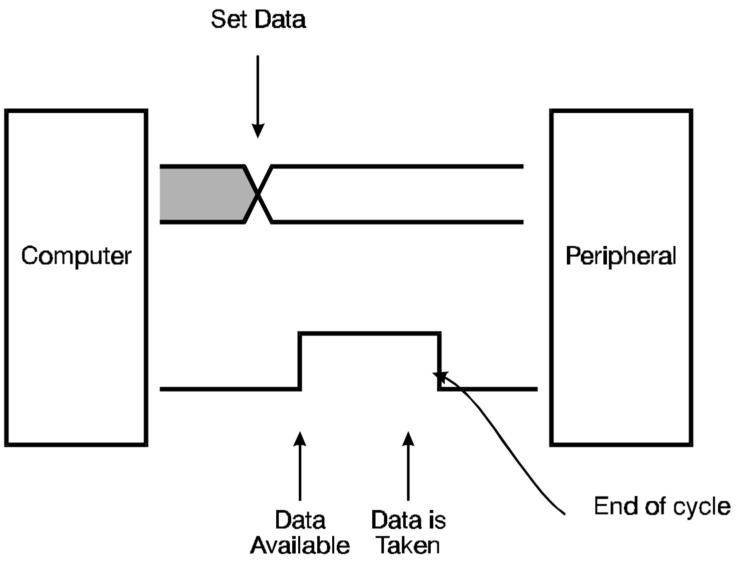
Parallel Communications

In the previous section we dealt with the situation where a single wire (or at most two) was used to communicate between systems. It was therefore necessary that the timing of the signals be used to establish the significance of the data. In parallel communications there are more wires running between the two systems and therefore both the spatial (which wire) and temporal (when) dimensions are available for the data.

Strobes, Enables and Handshakes

In a parallel communication problem there is just as great a need for a protocol and flow control as in the case of serial communications. Parallel communications however tends to have a greater emphasis on flow control or "handshaking" for a variety of reasons. The need for handshakes is really a property of the fact that two devices are running asynchronously to each other and therefore it is necessary to be able to communicate the fact that the data is ready/taken to the other device. A handshake may be "tighter" or "looser" depending upon the circumstances.

The simplest idea of a parallel handshake is a device which puts a line up when it is ready with data (Data Ready or DR) and then waits for the interrogating device to take the data. It must then immediately remove the request, otherwise the interrogator may well try to read the data again. Thus the regime is as shown in the diagram.

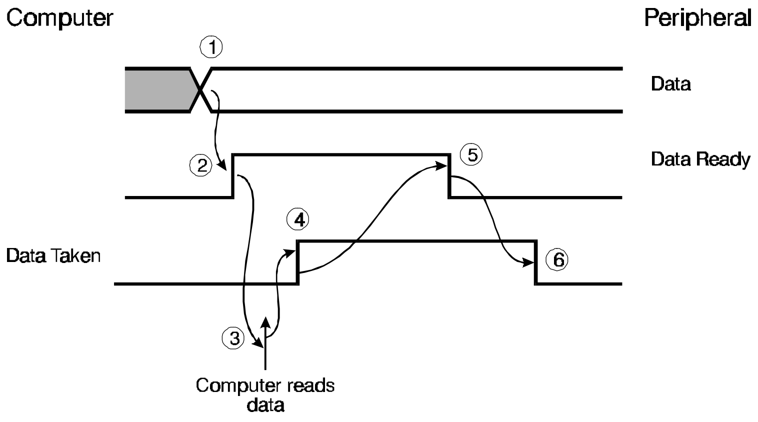


Simple Handshaking

Now in some devices it is reasonably easy to see whether the data has been read, particularly if you have access to the computer bus signals and can see whether a read has been done on the appropriate location. However if the bus signals are not available then some other means must be invoked, i.e. another communication line, "Data Taken" (DT), to tell the device that data has been taken. Thus we have the sequence:

		DR	DT
1	device places data onto data bus when DT is low	0	0
2	device raises DR	1	0
3	computer sees DR high and takes the data	1	0
4	computer raises DT	1	1
5	device sees DT high and lowers DR	0	1
6	computer sees DR low and lowers DT	0	0

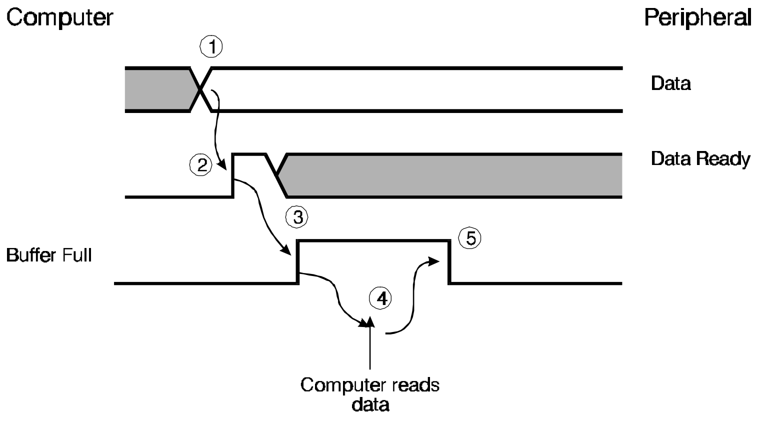
This complicated sequence of operations ensures that the two devices, each of which is probably intelligent, try to transfer data only when the other device can accept it. The sequence also ensures that each device can find out the current state of the transfer sequence from the state of the Data Ready and Data Taken lines - there is no ambiguity.



"Classic" Two-Wire Handshake

Now if you read many manuals on many devices, and in particular many peripheral interface chips such as the MC6821, NSC810, R6522 or a host of others, then you will find a variety of approaches to the problem of hardware handshaking. The reason for this is that there are numerous "speed ups" which can be done if you assume things about circuit reaction times, or build such assumptions in, and in this way things can be made a bit simpler for "dumb" devices. For instance the 6821 and 810 can both run as follows using a "Buffer Full" (BF):

		DR	BF
1	device places data on the data bus if BF is low	0	0
2	device raises DR(CA1)(STB)	1	0
3	peripheral chip raises BF(CA2)(BF)	1	1
4	computer reads peripheral chip	1	1
5	peripheral chip clears BF	1	0
6	device clears data ready	0	0



Modified Two-Wire Handshake

Notice that in this case the peripheral chip in the computer has a very fast reaction time (logic times - nS, rather than program times µS)[8] to the DR to tell the device to "hold up". Internally the peripheral chip also allows the computer to see the state of the control lines and thus to determine whether data is ready for pick-up. However since the chip has a finite reaction time from DR→BF the external device must not run too quickly - otherwise it will assume that BF has gone up and down again whereas it has not yet had time to go up at all - but this time is fairly short . In fact since the input only reacts to the edge of DR then a pulse will suffice to latch the request into the peripheral chip.

The main cause of this trouble in this handshake is the ambiguity between states 2) and 5) which have the same state of the DR and BF lines. In the absence of unambiguous state determination from the state of the lines some additional book-keeping is necessary.

[8] Actually in a 6821 it's not that short - 2µS, in the NSC 810 it's 500nS

The parallel peripheral chips therefore simplify life for devices because they allow for a very simple control structure at rather higher speeds than those allowed by the computer itself. This is in fact the major reason for having a peripheral chip to do some of the work for you rather than the "bare" provision of an input/output port. (However in reality the chips are so cheap that you can afford to buy them for the port alone)

Multiple Receivers

There are more problems when a single device is required to transmit data to a number of receivers because the "data taken" lines from all devices must go high before the next stage in the cycle can be initiated and in fact three lines are required.

We will illustrate the problem of a parallel network by discussing the "IEEE488" bus which is extensively used in instrumentation systems. It has the required three control lines called "DAV" (Data AVailable), "NFRD" (Not Ready For Data) and "NDAC" (Not Data ACcepted) and all these lines are active low[9]. DAV is controlled by the sending device and the other two lines are of the "wired-or" topology[10].

The following is an explanation of the above protocol based on 3 wires (DAV/NRFD/NDAC). As well as describing the sequences required for data transfer, it illustrates three different ways of showing such a set of events: a timing diagram, a flow chart and a state diagram. All three are equivalent descriptions which have different applications. For example the timing diagram is particularly useful if you are troubleshooting with an oscilloscope. The flow chart is useful if you are writing a software interface and the state diagram is useful if you are designing a controller using a Programmed Logic Array (PLA) with a "silicon compiler". In the diagrams the times when some of the devices have responded but others have not is represented in grey.

[9] "Active low" implies that the lines are doing what they imply when they are in the low state. Thus NRFD is low when a device is "Not Ready for Data" and conversely is high when it is "Ready For Data"

[10] Thus taking NRFD as an example: Any device can hold the NRFD line down (asserted). All devices must negate NRFD (ie be ready for data) before the line itself goes high.

List of Events for IEEE488 Handshake Process
(Numbers correspond to those on timing diagram and flow chart)

Initialisation

1.		Source initialised DAV to high (data not valid)
2.		Acceptors initialise NRFD to low (nobody ready for data) and set NDAC low (none have accepted data). They will then come ready for data one at a time until all are ready and NRFD is de-asserted. This occurs by point 5 below.

Cycle 1

3.	t_{-2}	Source checks for error condition (both NRFD and NDAC high): then sets data byte on DIO lines
4.	$t_{-2} \rightarrow t_0$	Source delays to allow data to settle on DIO lines
5.	t_{-1}	Acceptors have all indicated readiness to accept first data byte; NRFD high.
6.	t_0	Source, upon sensing NRFD high, sets DAV low to indicate that data on DIO lines are settled and valid.
7.	t_1	First acceptor sets NRFD low to indicate that it is no longer ready, then accepts that data. Other acceptors follow at their own rates.
8.	t_2	First acceptor sets NDAC high to indicate that it has accepted the data. (NDAC remains low because it requires all the acceptors to set NDAC high before going high)
9.	t_3	Last acceptor sets NDAC high to indicate that it has accepted the data; all have now accepted and NDAC goes high.
10.	t_4	Source, sensing that NDAC is high, sets DAV high. This indicates to the acceptors that data on the DIO lines must now be considered not valid.

Overlap Between Cycles

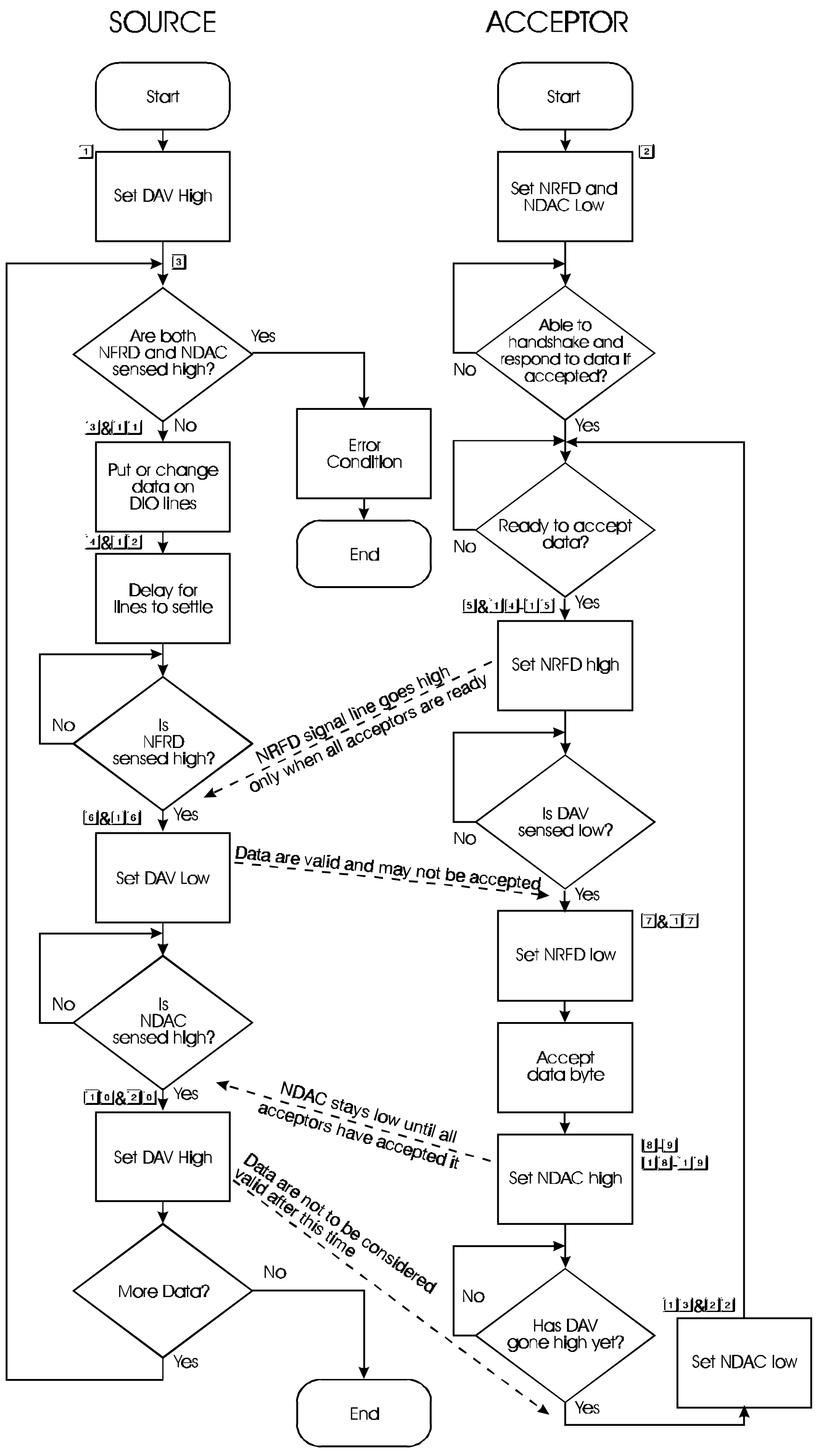
11.	$t_4 \rightarrow t_7$	Source changes data on DIO lines
12.	$t_7 \rightarrow t_9$	Source delays to allow data to settle on DIO lines
13.	t_5	Acceptors, upon sensing that DAV is high, set NDAC low in preparation for the next cycle. NDAC line goes low as the first acceptor sets the line low.
14.	t_6	First acceptor indicates that it is ready for the next byte by setting NRFD high. (NRFD remains low because other acceptors are not ready)
15.	t_8	Last acceptor indicates that it is ready for the next data byte by setting NRFD high. NRFD line goes high

Cycle 2

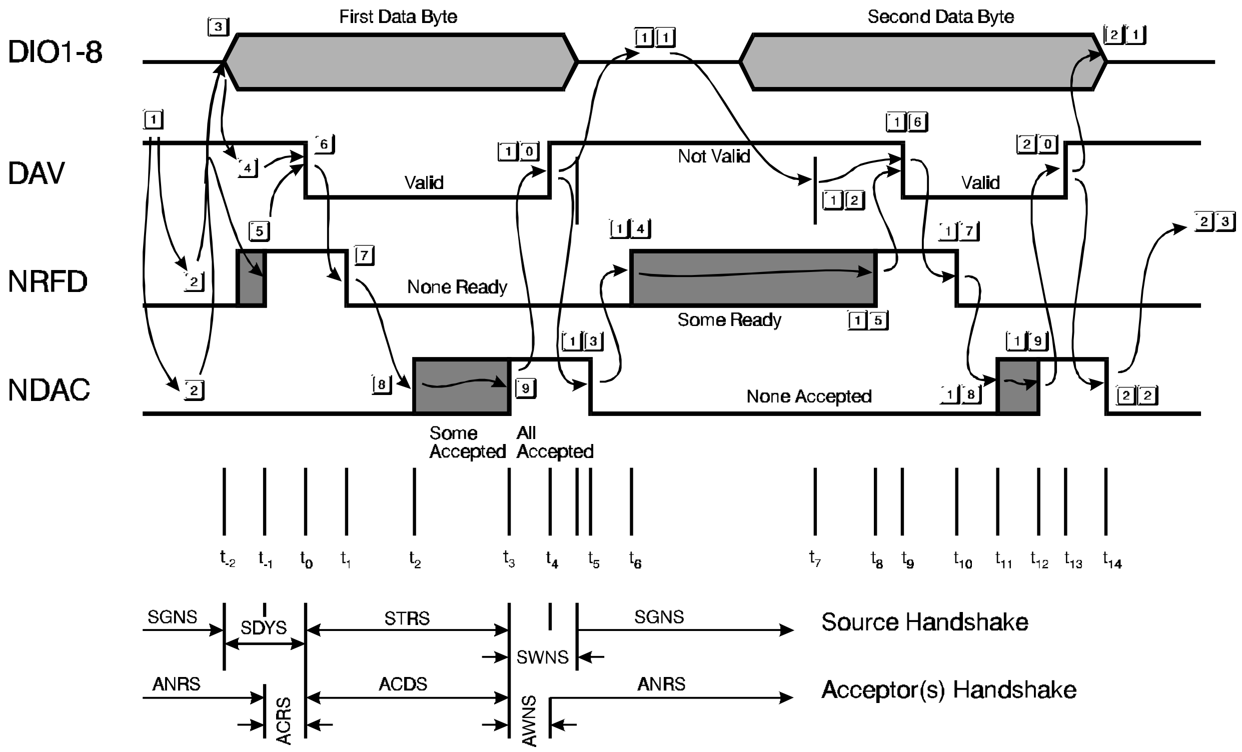
16.	t_9	Source, upon sensing NRFD high, sets DAV low to indicate that data on DIO lines are settled and valid
17.	t_{10}	First acceptor sets NRFD low to indicate that it is no longer ready, then accepts the data.
18.	t_{11}	First acceptor sets NDAC high to indicate that it has accepted the data.
19.	t_{12}	Last acceptor sets NDAC high to indicate that it has accepted the data.
20.	t_{13}	Source, having sensed that NDAC is high, sets DAV high.

EndGame

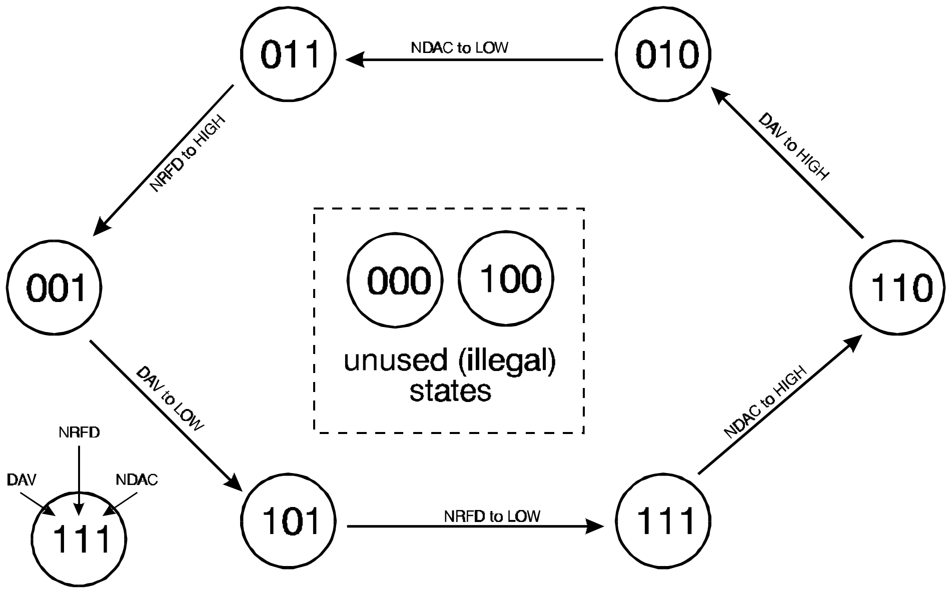
21.		Source removes data byte from DIO signal lines afeter setting DAV high.
22.	t_{14}	Acceptors, upon sensing, DAV high, set NDAC low in preparation for next cycle.
23.		Everything back in the initial state


SOURCE
ACCEPTOR
Start
Set DAV High
Are both NFRD and NDAC sensed high?
Yes
No
Error Condition
End
Put or change data on DIO lines
Delay for lines to settle
Is NFRD sensed high?
Set DAV Low
Is NDAC sensed high?
Set DAV High
More Data?
Start
Set NRFD and NDAC Low
Able to handshake and respond to data if accepted?
Ready to accept data?
Set NRFD high
Is DAV sensed low?
Set NRFD low
Accept data byte
Set NDAC high
Has DAV gone high yet?
Set NDAC low
NRFD signal line goes high only when all acceptors are ready
Data are valid and may not be accepted
NDAC stays low until all acceptors have accepted it
Data are not to be considered valid after this time

IEEE-488 Flow Chart



IEEE-488 Bus Timing Diagram



IEEE-488 Bus State Diagram

In the state diagram note that a "1" indicates assertion of the condition - the representation has no relationship to the logic level on the line. We can also see that two of the states - the ones corresponding to NRFD and NDAC being asserted - are not used and are therefore illegal. The initialisation takes us to state 011 and then the cycle proceeds anti-clockwise. Note that the first bit (DAV) is under the control of the source and the other two bits are under the control of the acceptor(s).

"Start" and "Finished" Lines

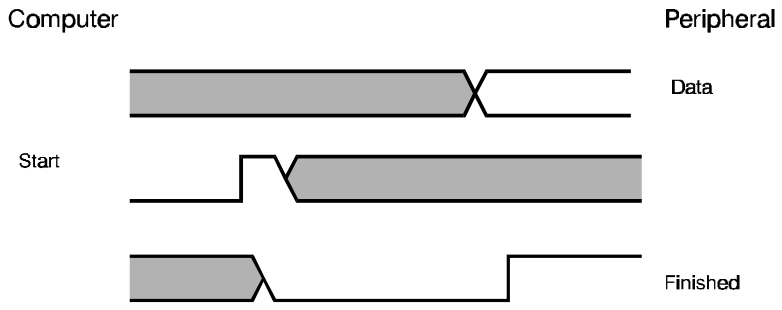
A simpler case occurs in devices which require a "start" command and then produce a "finished" line:

1) computer activates "start"
2) computer waits, interrogating "finished" until -
3) device completes and raises "finished"
4) computer reads result

notice that in this case it is assumed that the "start" command resets "finished".

Edge vs level

In most cases it is the leading edge of the "START" line which initiates the action. Beyond a minimum pulse length, the actual pulse length is immaterial - even if "START" is still high when "FINISH" becomes high a new cycle is not initiated. This is edge-triggering.



Use of START and FINISHED Lines

On the other hand, some systems are "level sensitive" in that a new cycle is initiated as soon as a "high" level is detected on the "START" line. Under these conditions it would be reasonable to assume that a new cycle would begin immediately if START were still high when FINISHED goes high. (However check the data sheets for confirmation before assuming anything!)

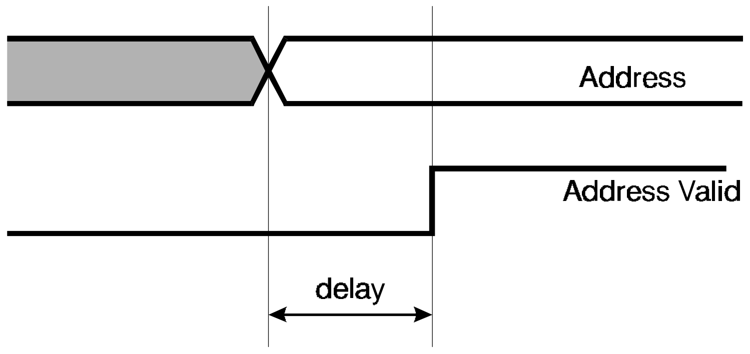
Data Ready

The concept of DATA READY requires some attention in that the raising of DATA READY implies that the data is guaranteed to be available, not that it will be available "real soon now"! A particular trap to avoid is when some data buffering is performed, perhaps to transfer data onto a heavily loaded (in the logic sense) bus. This effectively delays the data at the destination relative to the DATA READY which might not need so much buffering.

It is imperative that even with the maximum data delay and the minimum DATA READY delay, the data is still valid <u>before</u> DATA READY goes high. If this cannot be guaranteed, then DATA READY must be delayed sufficiently to guarantee the condition.[11]

Address Valid

As an example of this problem of guaranteeing that the data is valid before raising the "data valid" line, consider the case of a computer bus wishing to indicate that an address is valid.. In this system some additional decoding of part of the data (which in this case is actually the address value) is required in order to gate an appropriate "Address Valid" signal. The situation is complicated by the fact that there must be a separate "address valid" line for each memory chip in the memory to indicate to the chip that it is in fact being addressed. There is therefore a requirement for several stages of address decoding and problems with ensuring that the delays in the logic system are handled properly.



Address Validity on a Memory Bus

[11] The requirement of determining maximum and minimum delays raises problems in practical logic designs in that data sheets always show "typical" delay figures, frequently show "maximum" figures and rarely show "minimum" figures. Matching "typical" figures is not good enough as the system will "usually" work - but not necessarily always. The most stringent rule if no minimum delay is given is to assume that the minimum is zero. A better approach is to use system components which do quote minimum delays and/or to consult the manufacturer.

Two things are important here. First to ensure that the decoding of the appropriate segment of the memory (ie the particular chip) is completed before the "Address Valid" line goes high. Second to ensure that none of the possible address valid lines go high under any circumstances except the correct one of a decoded "Address Valid".

The first is addressed by ensuring that the decoding is complete before the master "Address Valid" goes high (considering maximum and minimum delay times appropriately). The second is ensured by using the final level of gating as a conditional gate using the master "Address Valid". In the example given earlier, "Address Valid" was fed into a "3→8 line decoder". The enable inputs of these decoders are designed in precisely this manner - a final level of gating to ensure that no invalid outputs are possible.

Since, in general, the complexity of address decoding increases with memory size, the delays also increase and the memory tends to become slower. In order to counteract this, there is strong motivation to use the largest memory blocks (chips) available - there are other reasons as well, such as cost and size. If a 16M x 8bits memory is required, less decoding is required with 16M x 1bit chips than with 1M x 1bit chips or even 2M x 8bit chips. Notice that if the chips are all of the same nominal access speed, then the memory constructed with 16M x 1 bit chips will tend to be faster because of the lack of decoding logic.