Chapter 8. Logic Emulator Interfaces

Introduction

As pointed out in Chapter 4, logic emulation is an important part of the logic validation process. While current systems are capable of performing emulation for single-chip systems, Chapter 6 presents a method for raising these benefits to the system level. In this way, board-level designs can be prototyped, speeding time-to-market for these circuits. Chapter 7 delved into some of the issues of how to build multi-FPGA hardware for emulation and other applications. However, in order to handle chip-level and board-level emulation, the system must not only provide an efficient method for mapping the logic, but also a means for handling the external communications of the circuit under validation.

One of the strongest potential reasons for using logic emulation instead of simulation is that an emulation of a system might be capable of operating in the target environment of the prototype circuit. In this way, the emulation would be exercised with real inputs and outputs, providing a much more realistic evaluation of the circuit's functionality, while providing a fully functional prototype for further experimentation.

Unfortunately, while it is clear that placing the emulation into its target environment is valuable, it is unclear how this can be accomplished in general. Because the environment expects to be communicated with via some protocol, and while the final circuit will obey the protocol, it is not clear that the emulation will meet the protocol's requirements. This problem is twofold: some protocols have timing constraints faster than the emulator can support, and the process of emulation slows the prototype's clock.

Some protocols have timing assumptions much faster than FPGA-based prototypes can deliver. For example, while logic emulators can reach speeds in the hundreds of kilohertz, or even a few megahertz, many communication protocols operate in the tens of megahertz range. Thus, unless the protocol will automatically slow down to the speed of the slowest communicator, the logic emulation will be unable to keep pace with the communication protocol, and thus cannot be placed in the target environment.

Even if the protocol is slow enough that the logic emulator could keep pace, the emulation will still not meet the protocol's timing requirements. This is because a logic emulation does not run at the same speed as the original circuit, and there will inevitably be slowdown of the emulation's clock speed. For example, a 50 MHz circuit built to communicate on a channel that requires responses at 1 MHz will communicate once every 50 clock cycles. If we use logic emulation for this circuit, we might achieve a performance of 10 MHz from the emulation. However, the emulation will still be communicating only once every 50 clock cycles. This achieves a communication performance of 200 KHz, much too slow to respond to a 1 MHz channel. In general, the emulation cannot be altered to communicate in less clock cycles, both because it may be busy performing other work during this time period, and also because so altering the emulation would change its behavior, and we would not be testing the true system functionality.

One solution to this problem is to build a custom interface transducer, a circuit board capable of taking the high rate communication interface that the environment assumes and slow it down sufficiently for the logic emulation to keep pace. This approach has already been taken by Quickturn Design Systems, Inc. with its Picasso Graphics Emulation Adapter [Quickturn93], which takes video output from a logic emulation and speeds it up to meet proper video data rates. This board is essentially a frame buffer, writing out the last complete frame sent by the emulation while reading in the next frame. Although this board is adequate for the specific case of video output, it will be useless for many other communication protocols, or even for video input to the logic emulation. With the large number of communication protocols, it is clear that special-purpose transducers cannot be marketed to handle every need. Expecting the user to develop and fabricate these transducers is also impractical, since now to emulate one chip or circuit board, the user must develop other circuit boards to handle the interfaces. Thus, the user has to do extra development and debugging work to use a system meant to simplify the development and debugging process.

In this chapter, we explore an alternative solution for handling logic emulation interfaces: the development and use of a single generic, standardized interface transducer for most logic emulation interfaces. In the next section, we discuss some of the details of communication interfaces. We also describe a standard, generic transducer board. Then, in the sections that follow we present some case studies, describing how several communication protocols can be mapped to this structure. These include NTSC video, digital audio, PCMCIA, and VMEbus. Finally, we discuss some of the general techniques and limitations of this approach, as well as some overall conclusions.

Protocol Transducers

As shown in Figure 59, communication protocols can be considered to have three levels of requirements: electrical, timing, and logical. At the electrical level, the protocol specifies the proper signaling voltages, the amount of current driven onto signals, termination resistors, allowed capacitances, and other features that interact with the electrical properties of the signaling medium. At the timing level, the protocol specifies the minimum and maximum time delay between transitions on signal wires. At the logical level, the protocol specifies the order of transitions necessary to accomplish various actions, both on the sender’s and on the receiver’s end of the communications. To properly communicate using a given protocol a system must obey the restrictions of all levels of the specification.

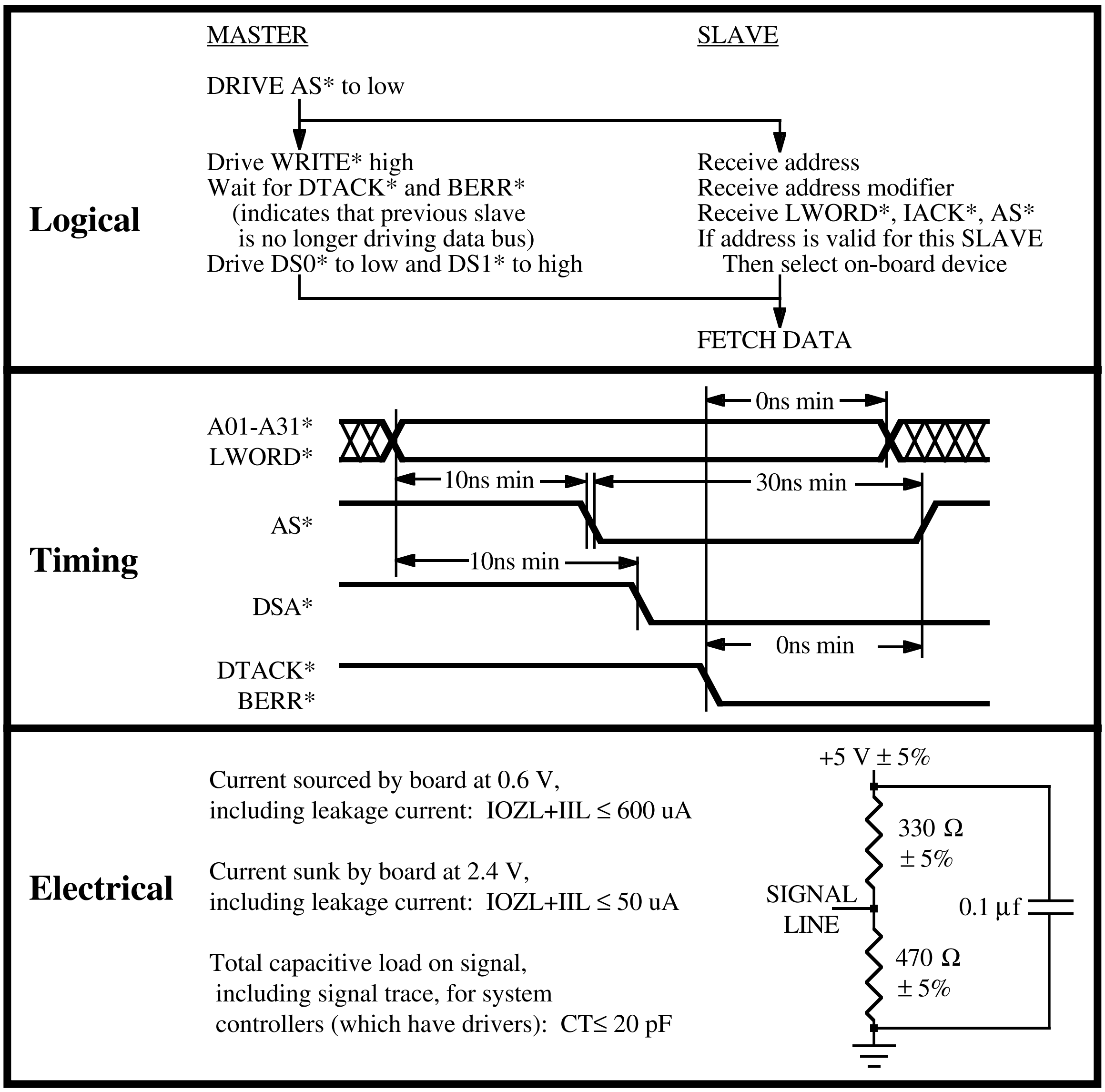

Figure 59. A communication protocol can be considered to have three levels: electrical, timing, and logical. Examples in this figure are adapted from the VMEbus specification [VMEbus85].

Most of the problems for the interfaces of logic emulations occur at the timing level. Since the circuit being emulated has been designed to communicate using the required protocols, it will already be capable of handling the logical level of the protocol. This means that the emulation will already know how to respond to any given set of input transitions, and will generate output transitions in the right order. However, the emulation will send and receive these transitions much more slowly than the final circuit is intended to, since the logic emulator will not be able to achieve the same performance as the custom hardware. The emulator may also not meet the electrical level specifications of the protocol. However, in general the electrical requirements are simple, and easily met by a small amount of interface hardware. In some cases, the protocol will perform all signaling using standard voltage values, with almost no requirements in the electrical level. This is most common when the protocol is used to communicate directly between only two systems. In this case, most programmable logic, including the logic contained in the interface transducer, will meet the electrical level specifications of the protocol. In other cases, there will be more stringent requirements on the emulation. However, most protocols will have a standard set of components that meet their electrical level specification (e.g., standard bus drivers, video input and output coders). As long as the emulator (with the help of a protocol transducer) is capable of meeting the logical and timing level specifications, the standard set of interface components can be used to meet the electrical level of the protocol.

The interface transducer for a logic emulator will be almost entirely focused on meeting the timing level of the specification. While the details of this vary greatly between protocols, there are in general two methods: slowing down the protocol, or filtering the data to reduce the amount of communication.

Many protocols are designed to allow a great deal of flexibility in the speed, cost, and quality of the systems involved in the communication. These protocols use a handshaking between the communicators, with both sides of a communication indicating both when they are ready to start or accept a communication, and also when they have finished with a communication. These protocols allow either end of a communication to slow down the actions taking place, and thus it is easy for a emulation to slow the communication sufficiently for it to keep up.

Some communications are not built with a handshake, and instead assume that any system using a given protocol is able to meet the specified timing requirements. Obviously, while the final circuit will meet these requirements, the emulation may not, and something must be done to compensate. What is necessary is to slow down the data coming into the emulation, and speed up the data leaving the emulation. In general, we wish the emulation to keep up with all incoming data streams, and not simply keep an ever increasing buffer of unreceived communications. Unfortunately, the emulation is capable of processing only a small fraction of the incoming communications. Thus, we must somehow reduce the number of communications coming into the emulation. The general approach is to throw away enough of the communications to allow the emulation to keep pace, while not throwing away any essential information. Obviously, simply throwing away all but the nth clock tick's worth of data will not normally work, since the information coming in will usually be totally corrupted. One possibility, applicable to protocols such as video input, is to throw away all but the nth packet or frame of data. In this way the emulation receives complete frames of video, though from the emulation's point of view the objects in the video are moving very fast. For other situations, we can throw away communications that do not include important data. For example, in a bus-based protocol, many of the transactions will not be intended for the emulation, and most systems can safely ignore them. Alternatively, for a voice recognition system an interface transducer can throw away the "dead air", and retain only those portions of the input surrounding high volume levels.

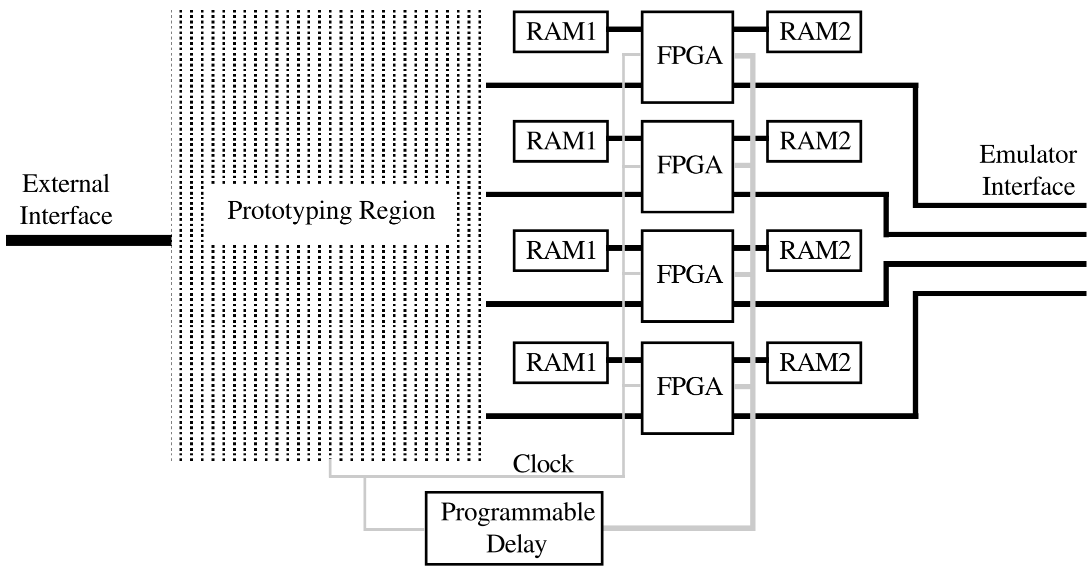

Figure 60. Interface transducer board.

The details of meeting the timing requirements of different protocols can vary greatly. However, the hardware necessary to do so can be generic and simple (see Figure 60): reconfigurable logic to perform filtering and meet timing delays, external RAM for holding intermediate results, and programmable delay lines to generate timing signals. Also important is a small prototyping area to accommodate any required standard components for meeting the electrical level requirements of the protocol. Note that there will often need to be communication between the FPGAs in the transducer so they can coordinate their actions. This can be accomplished by connecting together some of the unused wires leading from the prototyping region to the FPGAs, or from the FPGAs to the emulator interface. In general, the number of such connections will be small, since the number of actions the FPGAs will perform will be very limited. Flexibility is not only important in the number of interconnections between FPGAs, but also in the number and type of components used for a given mapping. Specifically, some interfaces will require only a single small FPGA, while others may need several large FPGAs plus a significant amount of RAM. This flexibility can be handled by using standard sockets for all chips. For example, a socket for the PQFP208 package can accommodate chips ranging from the Xilinx XC4003H (a high-I/O chip with relatively little internal logic) to the XC4025 (the largest capacity Xilinx FPGA) [Xilinx94]. In this way, a single board can be built to handle wide varieties of resource demands.

Example Mappings

In the following sections, we consider several different communication protocols, and describe how they are mapped onto the interface transducer board detailed above. This includes continuous streams of data (video and audio), as well as packetized protocols (PCMCIA and VMEbus).

Video

Video data is transmitted in a steady stream of video frames, arriving at a continuous rate (for the US's NTSC video the rate is 30 frames a second). Because the video arrives at a steady rate we must ignore some frames, throwing away enough data so that the remainder can be handled by the emulator. What remains are complete, sequential frames of video, though any motion in the pictures will seem artificially sped up to the emulator. This should be sufficient for most applications.

As mentioned earlier, to handle the constraints of a protocol we must handle the electrical, timing, and logical levels of that protocol. For video, there are standard chips available for handling the electrical characteristics of NTSC video [Philips92]. These chips take in the analog waveform containing the video data and convert it into a stream of digital data. Most video applications will use these or similar chips, and a transducer can rely on these chips to handle the electrical level of the interface.

The digital video data coming from the standard chipset arrives at a rate of 13.5 MHz, faster than most emulators can handle. To slow the video down, the protocol transducer maintains a buffer of video frames in the memories, filling up buffers as the logic emulator consumes them. In the simplest case, there needs to be only two frames worth of buffer space, one for the frame being sent to the emulator, and one for the frame being read from the external interface. In cases such as motion detection, the interface transducer may need to store several consecutive frames of video, increasing the memory demands. Because of the protocol transducer's socketed design the memory capacity can easily be increased, and the controlling logic in the FPGAs adjusted accordingly. Note that since the transducer will be writing arriving data to the memories at the same time as it is reading data to send to the emulator, there could be memory access conflicts with a single memory system. With the two memories of the proposed transducer interleaving or double-buffering can be used to avoid conflicts.

A similar mapping can handle outgoing video data as well. The primary difference is that since there is less data coming out of the emulator than the external interface expects, we must fill in the time between frames. A solution is simply to repeatedly output the last complete frame of data from the logic emulator until the next frame has been fully received.

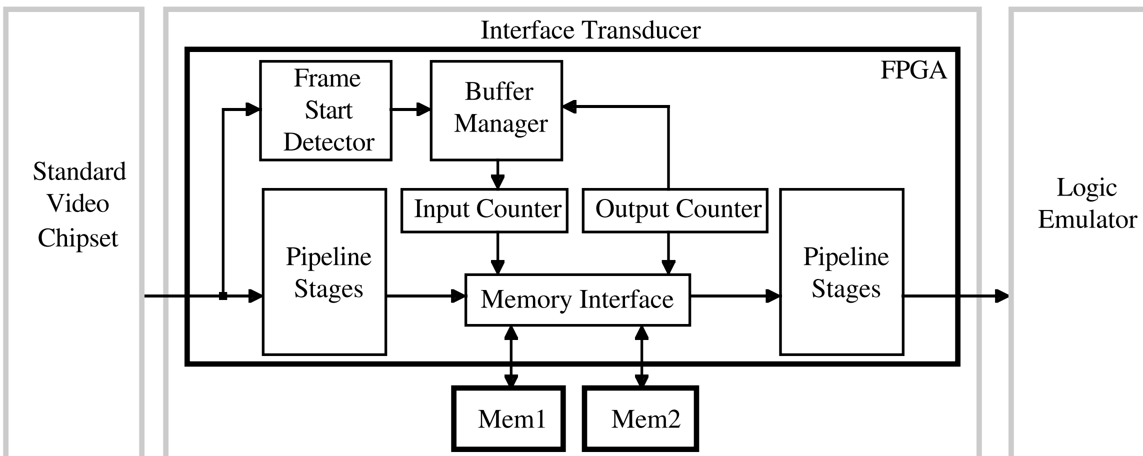

Figure 61. Logic diagram of an interface transducer mapping for incoming video data.

As we have just shown, there is relatively little logic necessary in an interface transducer to handle video data. A diagram of the logic for the input side is given in Figure 61. The logic contains a detector to find the beginning of an incoming frame. This information is passed to a buffer manager, which checks for an empty buffer in memory, and if so the incoming data frame is routed into this buffer. If there are no empty buffers, the frame is discarded. Two counters, one for incoming and one for outgoing data, index into the memory buffers. This is fed to a simple memory interface, which routes requests to the proper memory, and generates the proper read and write signals. To achieve the best performance these steps are all heavily pipelined. Since the data comes in at a steady, predictable pace, usually with no latency constraint, heavy pipelining can easily be achieved.

We implemented the interface transducer mapping as described above by specifying it in Verilog (a high-level design language). It was converted to a Xilinx 4000 mapping with completely automatic mapping tools, and easily achieved the required performance.

Audio

Even though audio is slower than video, and is well within the performance constraints of logic emulators, audio is actually more difficult to handle than video. Even though a mapping running on a logic emulator could meet the performance requirements for most audio applications, the problem is that the system under emulation was not designed to run on the logic emulator, but was in fact designed to run at a higher clock rate. Thus, it will accept or generate audio data on every nth clock cycle. Since the clock cycle on the logic emulator is slower, the system will not be keeping up with the external world.

Since the prototype on the logic emulator will not keep up with the required data rates of audio I/O, it will need an interface transducer to fix the problem. Unfortunately, unlike video, audio is not broken into frames, and there is no clear way to find data that can be ignored. For some situations, such as the input to voice recognition systems, the signal can be expected to have large gaps of silence, and a transducer could look for the "noisy" periods of time. The mapping takes all incoming data and stores it in one of the transducer's memories. The transducer looks for noisy periods, places where the signal amplitude is above some threshold. When such a period is discovered the transducer transfers the noisy period, along with a second or more of data on either side, to the next memory. We need this extra time period to catch the less noisy start and end of the signal, which is why we store all incoming data in the first memory. The data from the second memory is sent to the logic emulator, and can be padded with null data if necessary. In this way we can detect uninteresting data and throw it away.

Other, more common situations are harder to handle. In most cases we wish to preserve all of the audio data in the system. For example, if the system is generating or recording music there will always be interesting data in the audio signal, and there is no "dead air" to throw away. In such situations it is necessary is to save all of the data, perhaps to tape or disk, and use this storage as a buffer to change the speed of the data. For audio this is obviously feasible, since we already store large amounts of audio on today's systems. Thus, it may be worthwhile to augment the transducer hardware with a hard disk interface. Even in protocols and mappings where this feature is not necessary for buffering, the disk could still be used to record the data passing on the channel, serving as a debugging aid. However, for higher data rate signals (i.e., video), there may be too much data to store, and using secondary storage would not be sufficient.

PCMCIA

PCMCIA [Mori94] is a card format and bus protocol used to add peripherals to computer systems. It allows standardized memory cards, modems, disk drives, and other computer cards to be added to any compatible host, including systems from portable computers to Newton MessagePads. The interface can be moderated by standard chips such as the Intel 82365SL DF PC Card Interface Controller (PCIC) [PCIC93], which connects two PCMCIA card slots to an ExCA/ISA bus (see Figure 62).

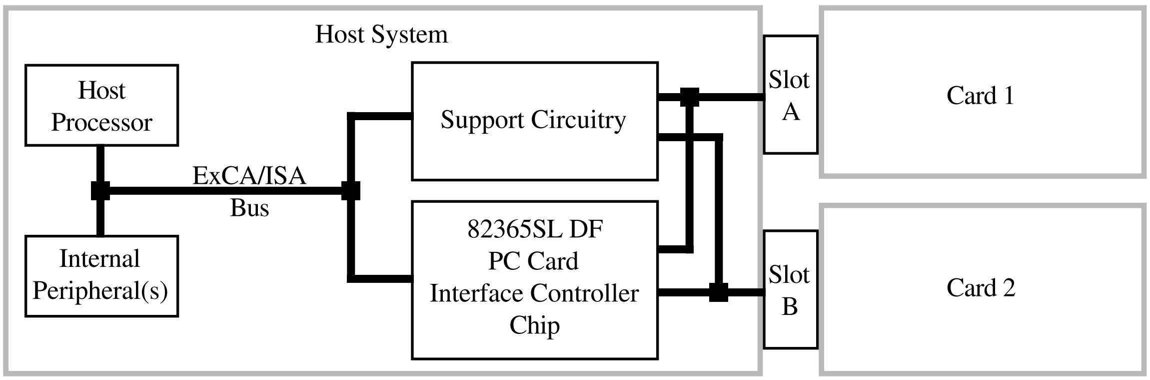

Figure 62. PCMCIA interface built with the Intel 82365SL DF PCIC chip.

Communications to the PCMCIA cards are initiated by the host processor. The PCIC chip routes these messages to the proper card slot based on the address sent by the processor. This card then has the option of asserting a WAIT signal to indicate that it needs extra time to process the current cycle. Once the card has finished processing the current cycle, either by reading in data sent by the host or by writing out requested data, it will deassert the WAIT signal. All signals are then returned to their prior (idle) state, and the system is available for another bus cycle.

Implementing a transducer for either a PCMCIA card or a PCMCIA host system is quite simple. The electrical level of the specification is handled by the Intel 82365SL DF PCIC chip plus some standard support hardware for the host side, while the card side requires little electrical adaptation. The timing level is also easy to meet because it is always obvious what subsystems are involved in a given communication, only one communication can be in progress at a time, and almost all timing constraints on the host system and the PCMCIA cards are minimum delays, not maximums. Note that conversely, the timing constraints on the PCIC chip itself are mostly maximums, making it difficult to emulate the PCIC chip itself. On the card side of the protocol, the transducer can use the WAIT signal to slow down the communication. Performing these actions inside the transducer's FPGAs is quite easy, and is well within the performance capabilities of current devices.

One important concern raised by examining the PCMCIA specification is that while it is trivial to handle the interfaces for the host system and for PCMCIA cards, it is much harder to meet the timing constraints on the controller chip itself. While most of the delays on the host and the cards are minimum delays, there are numerous maximum allowable delays on the controller chip, and these delays are on the order of tens of nanoseconds. The emulator will most likely be unable to meet the required delays, and it is unclear if there is any way to put an emulation of the controller chip into its target environment. The reason for this difference is quite simple. When we design protocols to interconnect many types of circuit boards we must realize that some of the boards will be much more complicated, or made from low-cost parts, slowing down their communications. Thus, inter-board protocols are designed to tolerate slower cards. However, when we design the interfaces to integrated circuits we expect a certain level of performance. The scope of the tasks required from these ICs are often limited and well defined, and thus we can require it to respond to external communications within a certain amount of time. Thus, meeting the external interface timings for system-level prototyping is in general simpler than for chip-level prototyping.

VMEbus Slave

The VMEbus [VMEbus85] is a backplane bus capable of supporting multiple Master (controller) boards, as well as Slave boards. One of the features that makes this system different than the others discussed so far is the fact that communication is bus-based, meaning that many of the transactions a component sees are intended for some other subsystem, and thus that component has no control over the processing of that transaction.

A VMEbus Master requests control over the bus via an arbitration process. Once it has gained control of the bus, it writes an address out onto the bus. Boards in the system are mapped into a common address space, and the address written to the bus will uniquely determine with which board the Master wishes to communicate. This board will respond to the Master's request, either reading or writing a data value onto the bus. The Master may then request additional data transfers from the same target board, either communicating several data values in the same direction at once, or performing a read-modify-write operation.

Slowing down the bus communications for an emulated slave can be difficult, primarily because the emulation will not be involved in all transactions that occur on the bus, and thus many of these communications will not give the emulation a chance to slow them. For transfers actually involving the emulation we can simply delay the interface signals to re-establish the proper minimum delays from the emulation's perspective. Communications that do not involve the emulation cannot be handled so simply, because the system does not wait for the emulation to process a transaction that is not intended for it. However, the emulation may not function correctly if these transactions proceed faster (in its slowed frame of reference) than the required minimum delays. There are two solutions to this problem. First, since the ranges of addresses a board responds to are fixed, we can program the interface transducer to simply ignore all transactions that are not intended for the emulation. A second choice is to allow the interface transducer to present a transaction to the emulation regardless of what board it is destined for, and delay it sufficiently to meet the minimum delays in the emulation's time frame. If the transaction is not destined for the emulation, this and other transactions can complete while the emulation is still receiving the original transaction. These transactions are ignored, and the interface only presents a new transaction when the last transaction has been completely received by the emulation. Since the transactions are very simple, requiring less than 100 bits, the data can easily be cached inside the FPGAs in the transducer. The emulation will not miss any transactions that are intended for it, since such a transaction would still be waiting for it when it completes the previous transaction. This second solution has the advantage that the emulation experiences transactions that are not intended for it, thus allowing the designer to test that the board does indeed ignore communications not intended for it. Note however that a emulation cannot "snoop" on the bus, attempting to overhear and capture communications not intended for it, since there will inevitably be transactions communicated on the bus that must be filtered out.

One significant problem with the VMEbus specification is that it contains several built-in time-outs. Specifically, if a board does not respond in a reasonable amount of time to a bus communication or arbitration, the system may decide to rescind the transaction. While these delays should be quite large, since these time-outs are only meant to free the bus in exceptional situations, it is possible that the emulation will operate too slowly to meet these upper bounds. In such a case, the logic that administers these time-outs will need to be modified or removed to allow proper operation of the emulation.

Conclusions

As we have shown, many logic emulator interfaces can be handled by a simple, generic interface transducer board. The board consists of FPGAs, memories, programmable delays, and perhaps an interface to a secondary storage device. This board is responsible for meeting the timing level specifications of the protocol. The electrical level of the protocol is met by standard chips where necessary, and the logical level is met in the emulator itself. The FPGAs on the transducer perform filtering and padding on the data stream, and make sure all required timing relationships are upheld. The memories are used for temporary storage and buffering, with two memories per FPGA to avoid read/write conflicts. The programmable delays are used to generate timing signals, both for the interfaces to the memories, and to help the FPGA meet the protocol's timing constraints. While the protocol transducer mappings can be somewhat involved, we have found that they can be quickly developed in a high-level language, and automatically translated into FPGA logic.

In this chapter, we described how this generic board can be used to handle NTSC video, digital audio, PCMCIA, and VMEbus. From these examples several general techniques have emerged, techniques that are applicable to many other situations. Interface transducers must somehow slow the external interface down to the speed of the logic emulator, either by delaying the communication, or filtering away data. Many protocols obey a strict handshaking, or have explicit wait or delay signals. In these cases, the incoming data rate can be slowed down to meet the emulator's speed simply by properly delaying these signals. In some cases (such as the VMEbus) data will be sent on a common medium between many subsystems, but only the sender and receiver of this data have control of the handshaking. In these situations the interface transducer can simply ignore data not intended for the emulation.

Other protocols do not have an explicit method for slowing the incoming data. Video and audio are good examples of this. In these cases it is necessary to filter away some of the data that is arriving, throwing away less interesting data, while storing the more important portions. In some cases this may require a large amount of buffer space, possibly even the use of secondary storage.

Timing constraints on individual signals must be respected by the protocol transducers. However, it is in general true for system-level prototyping that most of the timing constraints on the system are minimums, meaning that the emulator should not respond too quickly. Obviously this is not a problem. Note that if the emulation is of an individual chip, meeting the timing constraints can be much more difficult. This is because protocols are more likely to impose maximum response-time constraints on individual chips than on complete systems. The one exception to this is fault-tolerance features, features that impose a large maximum response time constraint on the system, reasoning that only a defective system will ever exceed these delays. In many cases, the emulation will meet these delays; in others, there is no choice but to disable the fault-tolerance features.

Although we did not encounter such protocols in our study, there are some types of interfaces that simply cannot be handled by any protocol transducer. Specifically, the logic emulation will run more slowly than the target system, and this will significantly increase the response-time of the system. Thus, the protocol must be able to tolerate large communication latencies. Also, there must be some method of reducing the incoming data rate. If the interface delivers data at a steady rate, and all data must be received by the system, there will be no way to reduce the data rate.

As we have shown, the design of a generic interface transducer is possible, and can meet most of the interface demands of current and future emulation systems. With a small amount of design effort, which can in general be carried out in a high-level design language, such systems can be run in their target environment, greatly increasing their utility.