University of Washington, Dept. of CS&E TR #95-04-04, 1995.

Achieving High-Latency, Low-Bandwidth Communication: Logic Emulation Interfaces

Scott Hauck, Gaetano Borriello, Carl Ebeling
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

There is a large amount of interest in using multi-FPGA systems for logic emulation and rapid-prototyping of digital systems. One difficulty with this approach is the handling of the external interfaces of the system. In this paper we describe a generic interface transducer, a board capable of handling the external interfaces of the system under prototype, allowing the emulation to operate in the target environment. We also describe how several protocols can be implemented on this board, including NTSC video, digital audio, PCMCIA, and VMEbus.

Introduction

Logic emulation with FPGAs (the mapping of circuitry to be tested onto a multi-FPGA system) holds the promise of greatly decreasing development times for both ASICs and complete systems. Emulation systems [Tessier94, Thomae91, Varghese93, Yamada94, Zycad94] offer orders of magnitude speedup over software simulation for ASIC designs. There is also work being done to bring similar benefits to board-level and system-level emulation [Aptix93, Hauck94, Koch94].

One of the strongest potential justifications for using logic emulation instead of simulation is that a logic emulation of a system might be capable of operating in the target environment of the prototype circuit. In this way, the prototype would be exercised with real inputs and outputs, providing a much more realistic evaluation of the circuit's functionality, while hopefully providing a fully functional prototype for further experimentation. This is important, because ideally a prototype will not only be used to determine whether the circuit implementation meets its specification, but also whether what was specified is actually what is wanted, and if what is wanted will actually be useful. Specifically, while working with a prototype the implementers, as well as the eventual users, may be able to determine not only whether there is faulty logic in the implementation, but also whether the circuit meets their expectation. If we only use simulation, the interaction with the simulation is much more restricted than with an emulation, which greatly limits how much and what kinds of testing can be performed.

Unfortunately, while it is clear that placing the prototype into its target environment is valuable, it is unclear how in general this can be accomplished. This is because the environment expects to be connected via some protocol, and while the final circuit will obey the protocol, it is not clear that the prototype will meet the protocol's requirements. This problem is twofold. First of all, 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.

The second issue is that even if the protocol is slow enough that the logic emulator could keep pace, the prototype 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 some slowdown of the prototype's clock speed. For example, a 50 MHz circuit built to communicate on a channel that requires responses at 1 MHz will be constructed to 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 prototype 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 prototype 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 prototype would change its behavior, and we would no longer 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 prototype 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 existing communication protocols, it is clear that special-purpose transducers cannot be marketed to handle every user's needs. Expecting the user to develop and fabricate these transducers is also impractical, since now in order to prototype one chip or circuit board, the user must develop other circuit boards to handle the interfaces. Thus, we now have to do extra development and debugging work in order to use a system meant to simplify the development and debugging process.

In this paper 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 a communication interface. We also describe a standard, generic transducer board for these interfaces. 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.

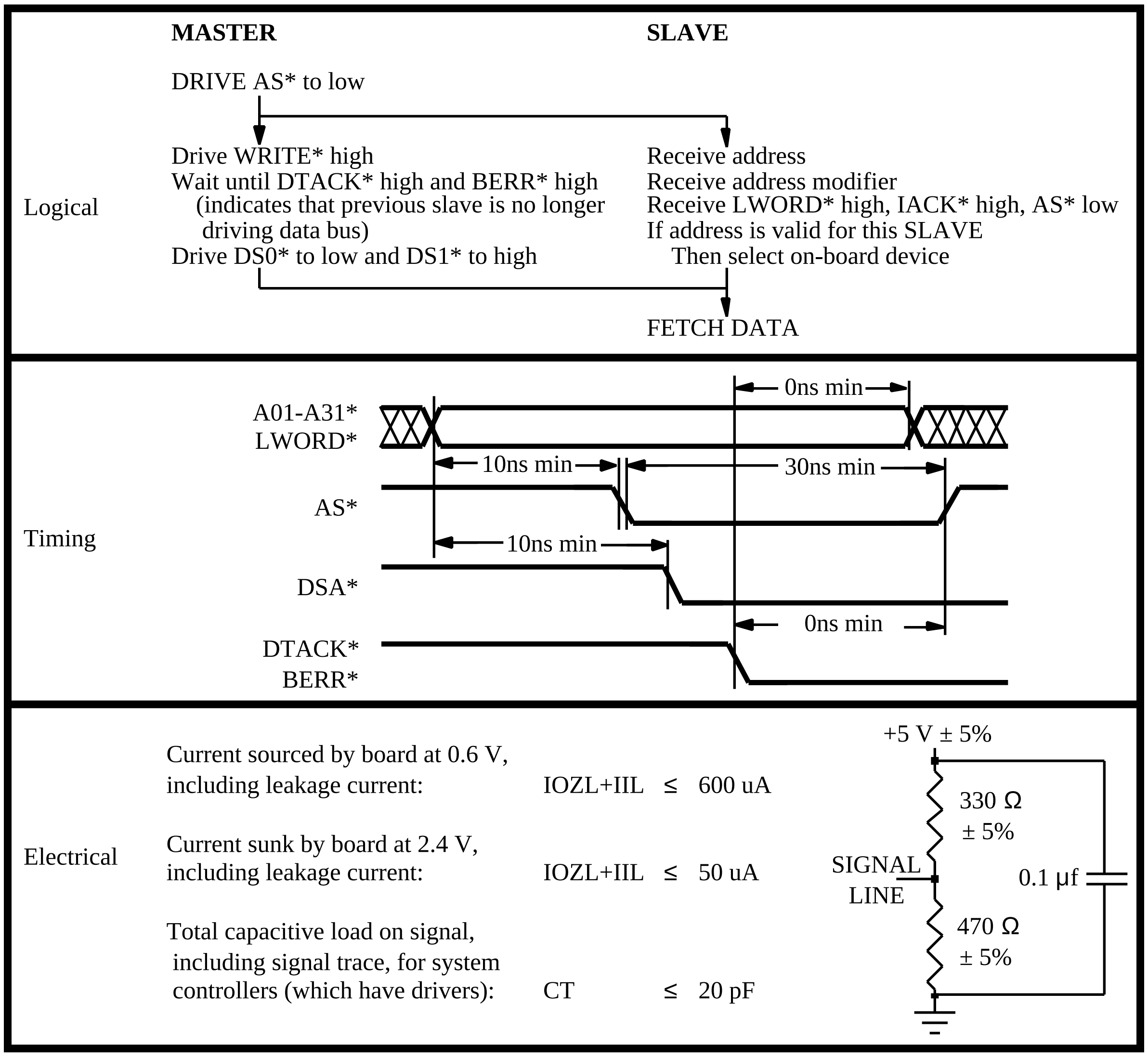
MASTER
SLAVE
Logical
DRIVE AS* to low
Drive WRITE* high
Wait until DTACK* high and BERR* high
(indicates that previous slave is no longer driving data bus)
Drive DS0* to low and DS1* to high
Receive address
Receive address modifier
Receive LWORD* high, IACK* high, AS* low
If address is valid for this SLAVE
Then select on-board device
FETCH DATA
Timing
A01-A31*
LWORD*
AS*
DSA*
DTACK*
BERR*
0ns min
10ns min
30ns min
10ns min
0ns min
Electrical
Current sourced by board at 0.6 V, including leakage current:
IOZL+IIL ≤ 600 uA
Current sunk by board at 2.4 V, including leakage current:
IOZL+IIL ≤ 50 uA
Total capacitive load on signal, including signal trace, for system controllers (which have drivers):
CT ≤ 20 pF
+5 V ± 5%
330 Ω ± 5%
SIGNAL LINE
0.1 μf
470 Ω ± 5%

Figure 1. A communication protocol can be considered to have three levels: electrical, timing, and logical. Examples in this figure adapted from the VMEbus specification [VMEbus85].

Protocol Transducers

As shown in figure 1, 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. While each of these levels is conceptually different, to properly communicate using a given protocol a system must obey the restrictions of all levels of the specification.

It turns out that for logic emulation interfaces most of the problems occur at the timing level. Since the circuit being prototyped 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 prototype will already know how to respond to any given set of input transitions, and will generate output transitions in the right order. However, the prototype will send 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 also may 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 prototype. However, most protocols will have a standard set of components that meet their electrical level specification. 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.

Since the logic emulation of the circuit will meet the logical level of the protocol specification, and since the electrical level of the specification can be handled by at most a small set of standard components, the interface transducer will be almost entirely focused on meeting the timing level of the specification. While the details of meeting the timing level of the specification 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 prototype to slow the communication sufficiently for it to keep up. However, as we will see in the examples that follow, there are still some issues to be dealt with even in these systems.

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 prototype may not, and something must be done to compensate. What is necessary is to somehow slow down the data coming into the prototype, and speed up the data leaving the prototype. In general we wish the prototype to keep up with all incoming data streams, and not simply keep an ever-increasing buffer of unreceived communications. Unfortunately the prototype is capable of processing only a small fraction of the incoming communications. Thus we must somehow reduce the number of communications coming into the prototype. How this is done differs from protocol to protocol, but the general approach is to throw away enough of the communications to allow the prototype 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 work, as the information coming in will be totally corrupted. One possibility, applicable to protocols such as video input, is to throw away all but the nth frame of data. In this way the prototype receives complete frames of video, though from the prototype's point of view the objects in the video are moving very fast. For other situations, we can throw away uninteresting communications, such as those not intended for the prototype, or those that do not include important data. For example, in a bus-based protocol, many of the communications will not be intended for the prototype, 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.

Although there are only a few general methods for meeting the timing requirements of a protocol, the details of doing so for different protocols can very greatly. However, the hardware necessary to do so can be generic and simple (see figure 2): reconfigurable logic to perform filtering and meet timing delays, external RAM for holding intermediate results, and possibly 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 system so they can coordinate their actions. This can be accomplished by connecting together some of the 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 small amount of logic in a single 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. Also, at least in the case of the Xilinx FPGAs [Xilinx94], a single socket type can accommodate many different capacities of FPGAs. For example, a socket for the PQFP208 package can accommodate chips from the Xilinx XC4003H (a high-I/O chip with relatively little internal logic) to the XC4025 (the largest capacity Xilinx FPGAs). In this way, a single board can be built to handle wide varieties of resource demands.

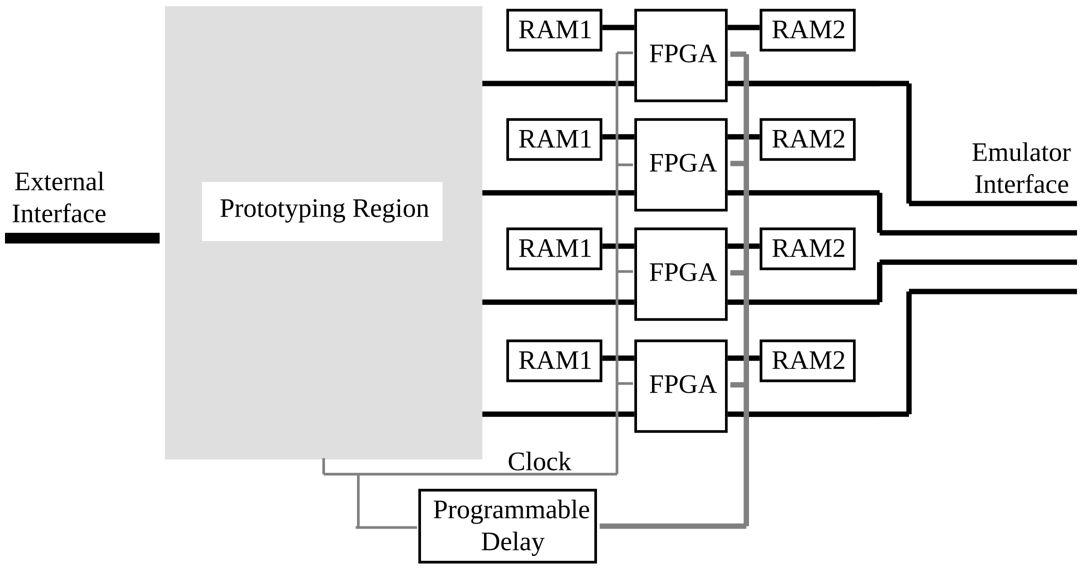

Figure 2. Interface transducer board.

Example Mappings

In the following sections, we consider several different communication protocols, and describe how they are mapped onto the interface transducer board described 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 NTSC video, the standard used for broadcast video in the US, the rate is 30 frames a second, 2 interlaced fields per frame). 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 be artificially sped up (that is, if we view the remaining frames at the standard rate). 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 the 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 after the logic emulator has consumed 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 FPGAs in the protocol transducer, as well as its socketed design, the transducer's memory capacity can be easily increased. Note that since the transducer will be writing arriving data to the memories at the same time as it is reading data from the memories to send to the emulator, there could be memory access conflicts if we were using only one memory at a time, or a single memory for a given set of frames. The solution we have adopted is to interleave storage of the data to the two memories, with all data from one timestep residing in a different memory from the next timestep. If we assume the emulator runs at least twice as slow as the target system, then the transducer will have at least two clock cycles (where the clock is the 13.5 MHz clock of the incoming data) to supply the emulator with the next data value. Since the transducer will not write data to the same memory in two successive cycles, there will always be an opportunity to read data from the memories to supply it to the emulator.

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 somehow fill in the time between frames. The solution is simply to repeatedly output the last complete frame of data from the logic emulator until the next frame has been fully received.

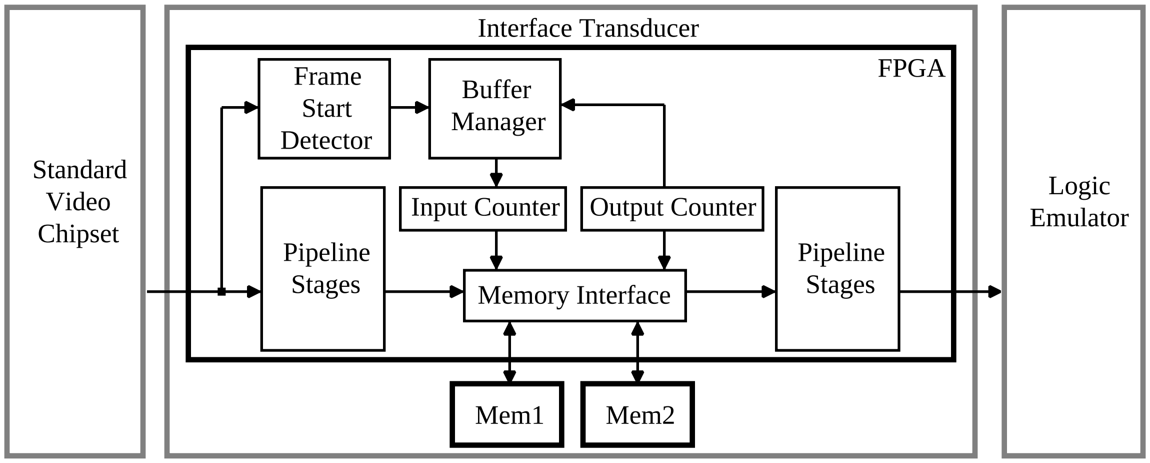

Figure 3. Logic diagram of a interface transducer mapping for incoming video data.

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 3. The logic contains a detector to find the beginning of an incoming frame. This information is passed to a buffer manager, which simply determines if there is an empty buffer in memory, and if so the incoming data frame is routed into this buffer. If there is not an empty buffer, the frame is discarded. Two counters, one for incoming data from the external interface and one for outgoing data to the emulator, index into the memory buffers. The low-order bit from these counters are used to decide in which memory the current data should reside. Thus, this interleaves accesses to the two memories. This is fed to a simple memory interface, which routes input and output requests to the proper memory, and generates the proper read and write signals. To achieve the best performance all of these steps are heavily pipelined. Since the data comes in at a steady, predictable pace, and since there usually is 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), and used completely automatic mapping tools to convert it to a Xilinx 4000 mapping. By simply making sure all logic had at most five inputs, with a register on the output, and by using a counter design from the Xilinx Application Briefs [Xilinx94], we were easily able to achieve the required performance. The performance could easily be increased by the specification of timing hints to the tools, as well as by the hand placement of individual logic blocks.

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 prototype was not built to run at the speeds achieved on the logic emulator, but was in fact designed to run at a higher clock rate. Thus, it will accept or output 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. While it might be possible to alter this, and adjust the mapping to account for the slowdown, in general this will not be feasible, and may distort the system.

Since the prototype on the logic emulator will not keep up with the required data rates of its 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 would take all incoming data and store it in one of the transducer's memories. Since the data rates are so low, there are no concerns over read/write conflicts on the memories, and only one memory is necessary for this. The transducer would look for noisy periods, places where the signal amplitude is above some threshold. When such a period is discovered the transducer would transfer 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, and this 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 (that is, during long quiet stretches). 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 is too much data to store, and using secondary storage would not be sufficient. It is unclear if there is any way to handle high data rate signals that have no uninteresting periods, no periods where the data can be discarded. However, we have yet to encounter such a protocol.

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 4).

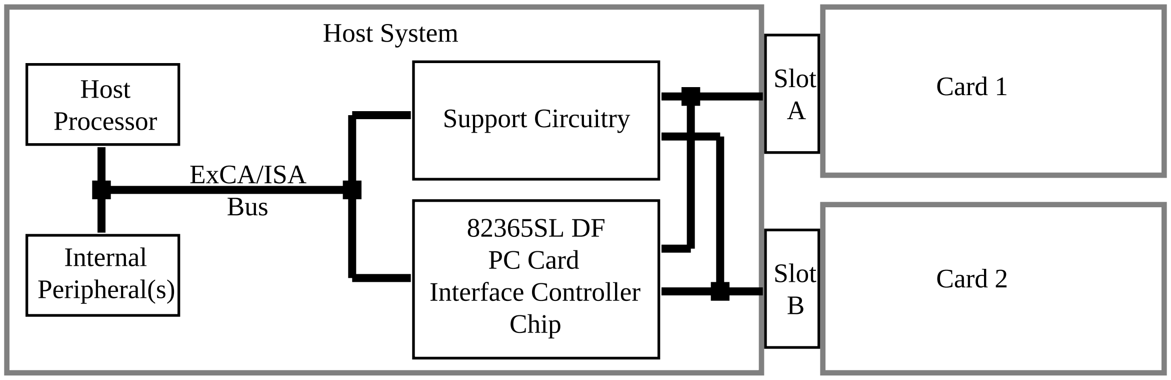

Figure 4. PCMCIA interface built with the Intel 82365SL DF PCIC chip.

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. Cards can generate interrupt requests to the host processor via special interrupt lines. Note that the PCMCIA specification also contains provisions for digital audio to be sent from a card to the host system; the details of this portion of the protocol are ignored here, and can be handled as described in the audio section earlier.

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 very little special processing. There are some constraints on the physical properties of the PCMCIA cards, which translate into electrical and timing properties of the protocol. However, these primarily involve the order of certain events, such as when power lines connect and disconnect relative to other signals, requirements that are easily met in a transducer mapping. 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 any 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 prototype the PCIC chip itself. The reason why it is always clear who is involved in this communication is that the Host always initiates all bus communications, and the PCIC chip takes care of activating only the slot involved in the current communication. Thus, meeting the timing constraints of the host system involve simply resynching a few signals to the clock lines, stretching some interrupt signals coming into the host system to meet their minimum delays, and tristating the bi-directional data lines quickly at the end of a bus cycle. Note that in general bi-directional buses take some extra work inside an FPGA-based transducer, since internal to the FPGAs these bi-directional signals must be carried by two unidirectional signal lines, with only one of the directions active at a given time. However, the direction of these lines is determined easily from the state of a few bus signals. Meeting the PCMCIA timing specification on the card side is similar to the host side, except that the interface logic must also use the WAIT signal automatically to slow the current bus cycle. It does this by asserting the WAIT signal near the beginning of all bus cycles. If the card decides to assert its own WAIT signal, this serves to sustain the WAIT signal generated by the transducer; if the card does not assert its WAIT signal within the required time period, where this time period has been adjusted to compensate for the slowdown caused by emulation, the transducer lowers the WAIT signal and allows the cycle to complete. Performing all of 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 considering the PCMCIA specification is that while it is trivial to handle the interfaces for the host system and for a PCMCIA card, 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 events generated by the controller chip, and these delays are on the order of tens of nanoseconds. It is unclear if there is any way to prototype the controller chip itself on a logic emulator while running it in its target environment. The emulator will most likely be unable to meet the required delays, and it is difficult for a transducer to speed up the events without implementing most of the logical portion of the protocol, which means implementing most of the controller chip in the transducer. The reason for this discrepancy is quite simple. When protocols are designed to interconnect many types of circuit boards, it must take into account the fact that some of the boards will be much more complicated, or made from low-cost parts, slowing down their responses to communication events. Thus, the protocols are designed to tolerate slower cards. However, when we design the interfaces to integrated circuits, we realize that there will be a smaller number of different ICs for a given task, and 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 on the bus are intended for some other subsystem, and thus that component has no control over the processing of that transaction.

The VMEbus is broken into a Data Transfer Bus, an Arbitration Bus, an Interrupt Bus, and a Utility Bus. Data communication is accomplished over the Data Transfer Bus. A Master, after it has gained control of the bus, writes an address out onto bus. Boards in the system are mapped into a common address space, and the address written to the bus will uniquely determine which board the Master wishes to communicate with. This board will respond to the Master's request, either reading or writing a data value onto the data lines of 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 data transfer bus communications for a prototype slave can be difficult, primarily because the prototype will not be involved in all transactions that occur on the bus, and thus many of these communications will not give the prototype a chance to slow down the communications. For transfers actually involving the prototype, almost all of the required timing relationships are minimums, not maximums. This means that we can simply delay the signals in the interface to re-establish the proper minimum delays from the prototype's perspective. Communications that do not involve the prototype cannot be handled so simply, because the system does not wait for the prototype to process a transaction that is not intended for it. However, the prototype 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 of all, since the ranges of addresses a board responds to are fixed, we can program the interface to ignore all transactions that are not intended for the prototype, and the issue is avoided. A second choice is to allow the interface to present a transaction to the prototype regardless of whom it is destined for, and delay it sufficiently to meet the minimum delays in the prototype's time frame. If the transaction is not destined for the prototype, this and other transactions can complete while the prototype 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 prototype. Since the transactions are very simple, requiring less than 100 bits, the data can easily be cached inside the FPGAs in the interface. The prototype 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 prototype board 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 prototype 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.

The Arbitration and Interrupt buses are much easier to handle than the Data Transfer Bus, since most of the processing is handled by a daisy-chain of point-to-point communications. The interface need only slow portions of the processing down enough to meet the minimum delays guaranteed to all boards by the VMEbus specification. The Utility bus is also not a problem for the prototype, since it provides system signals such as power-up and clock signals, and failure indication lines, signals whose behaviors can easily be retained by the interface.

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 an arbitration or Data Transfer Bus communication, the system may decide to rescind the transaction. While these delays should be quite large, since these systems are only meant to free the bus in exceptional situations, it is possible that the prototype 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 prototype.

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 required 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 paper, we described how this generic board can be used to handle NTSC video, digital audio, PCMCIA, and VMEbus. Beyond the details of the individual mappings, there are several general techniques that 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 prototype.

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, at least in the case of 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 make maximum response-time constraints on individual chips, but not on complete systems. The one exception to this is fault-tolerant 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.

Acknowledgments

This research was funded in part by the Advanced Research Projects Agency under Contract N00014-J-91-4041. Scott Hauck was supported by an AT&T Fellowship.

References

[Aptix93] Aptix Corporation, *Data Book*, San Jose, CA, February 1993.

[Hauck94] S. Hauck, G. Borriello, C. Ebeling, "Springbok: A Rapid-Prototyping System for Board-Level Designs", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Koch94] G. Koch, U. Kebschull, W. Rosenstiel, “A Prototyping Environment for Hardware/Software Codesign in the COBRA Project”, *Third International Workshop on Hardware/Software Codesign*, September, 1994.

[Mori94] M. T. Mori, *The PCMCIA Developer’s Guide*, Sunnyvale, CA: Sycard Technology, 1994.

[PCIC93] *82365SL DF PC Card Interface Controller (PCIC)*, Santa Clara, CA: Intel Corporation, April 1993.

[Philips92] *Desktop Video Data Handbook*, Sunnyvale, CA: Philips Semiconductors, 1992.

[Quickturn93] Quickturn Design Systems, Inc., “Picasso Graphics Emulator Adapter”, 1993.

[Tessier94] R. Tessier, J. Babb, M. Dahl, S. Hanono, A. Agarwal, “The Virtual Wires Emulation System”, *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Thomae91] D. A. Thomae, T. A. Petersen, D. E. Van den Bout, “The Anyboard Rapid Prototyping Environment”, *Advanced Research in VLSI 1991: Santa Cruz*, pp. 356-370, 1991.

[Varghese93] J. Varghese, M. Butts, J. Batcheller, "An Efficient Logic Emulation System", *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, pp. 171-174, June 1993.

[VMEbus85] *The VMEbus Specification*, Tempe, Arizona: Micrology pbt, Inc., 1985.

[Xilinx94] The Programmable Logic Data Book, San Jose, CA: Xilinx, Inc., 1994.

[Yamada94] K. Yamada, H. Nakada, A. Tsutsui, N. Ohta, “High-Speed Emulation of Communication Circuits on a Multiple-FPGA System”, *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Zycad94] *Paradigm RP*, Fremont, CA: Zycad Corporation, 1994.