A VLSI design of the ActiveBus Interface Unit

James Van Peursem
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
A VLSI design of the ActiveBus Interface Unit

by

James Edward Van Peursem

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Iowa State University
Ames, Iowa
1991
DEDICATION

To my wife Denise, without whose unbounded patience and support
the completion of this work would not have been possible.

Special thanks also go to Herman Stallbaum, who inspires me daily.
TABLE OF CONTENTS

ABSTRACT........................................................................................................... viii
CHAPTER 1: INTRODUCTION.................................................................................... 1
CHAPTER 2: ABIU OPERATIONAL CHARACTERISTICS................................. 5
  2.1 Slave Transactions ....................................................................................... 5
  2.2 Master Transactions .................................................................................... 6
  2.3 Design Strategy ......................................................................................... 7
  2.4 Pin Description ........................................................................................... 9
CHAPTER 3: ACTIVEBUS PROTOCOL................................................................. 15
  3.1 Transactions Supported ............................................................................ 15
    3.1.1 Arbitration ......................................................................................... 15
    3.1.2 Write transactions ............................................................................ 18
    3.1.3 Read transactions ............................................................................ 20
    3.1.4 Error signaling ................................................................................ 20
CHAPTER 4: BACKPLANE INTERFACE ............................................................... 24
  4.1 Status State Machine .............................................................................. 25
  4.2 Master State Machine ............................................................................. 27
  4.3 Slave State Machine .............................................................................. 28
  4.4 Arbitration State Machine .................................................................... 30
  4.5 Schematics Description .......................................................................... 32
  4.6 Arbitration logic ..................................................................................... 37
  4.7 Bus Pads ................................................................................................. 38
LIST OF TABLES

Table 2.1. Description of the ABIU pins corresponding to bus signals .........................9
Table 2.2. Description of the ABIU pins corresponding to module signals ...................11
Table 2.3. Description of the other ABIU pins ..........................................................14
LIST OF FIGURES

Figure 1.1: Passive backplane ................................................................. 2
Figure 1.2: Active backplane ................................................................. 3
Figure 2.1: Block diagram of the ABIU ...................................................... 7
Figure 2.2: ActiveBus signal levels ......................................................... 8
Figure 3.1: Arbitration flow chart ........................................................... 17
Figure 3.2: Write transaction ............................................................... 19
Figure 3.3: Read transaction ............................................................... 21
Figure 3.4: Error transaction ............................................................... 23
Figure 4.1: Status state machine ........................................................... 26
Figure 4.2: Master state machine ......................................................... 27
Figure 4.3: Slave state machine ........................................................... 29
Figure 4.4: Arbitration state machine ................................................... 31
Figure 4.5: Bus interface schematic ....................................................... 33
Figure 4.6: Decode_cw schematic ......................................................... 36
Figure 4.7: Arbitration circuitry ........................................................... 38
Figure 4.8: Bus transceiver schematic ................................................ 39
Figure 5.1: Buffer analysis model ................................................................. 41
Figure 5.2: Outgoing buffer no. 1 full (before data transferred on bus) ........... 43
Figure 5.3: Incoming data buffer no. 2 full (after data transferred) ............... 44
Figure 5.4: Fast RAM buffer ................................................................. 46
Figure 5.5: FIFO with multiplexed outputs .............................................. 47
Figure 5.6: General buffer structure ....................................................... 48
Figure 5.7: Down buffer schematic ....................................................... 50
Figure 5.8: Clock generation circuitry ..................................................... 52
Figure 5.9: Basic storage element, logical view ....................................... 54
Figure 5.10: Basic storage element, transistor view .................................... 54
Figure 5.11: Basic storage element layout .............................................. 55
Figure 5.12: Each bit being stored requires two storage elements ............... 56
Figure 5.13: Layout of clock driver ....................................................... 58
Figure 5.14: Load buffers state machine ............................................... 60
Figure 5.15: Unload buffers state machine ............................................. 61
Figure 5.16: Up buffer schematic ........................................................... 62
Figure 6.1: Master write ....................................................................... 64
Figure 6.2: Master read ....................................................................... 66
Figure 6.3: Slave write ....................................................................... 67
Figure 6.4: Slave read ....................................................................... 69
Figure 6.5: Slave read ending in error ................................................... 70
Figure 6.6: Clock generation circuit ....................................................... 72
Figure 6.7: Up buffer ....................................................................... 73
Figure 6.8: Layout of final chip ............................................................. 74
ABSTRACT

To achieve the performance necessary in multiprocessor systems, an efficient communications medium must exist between processors. A popular and fairly inexpensive connection scheme is the communications bus. In most systems though, it is also usually the bottleneck of the system. To improve system performance, an improvement in the bus capacity must be made. This thesis describes the bus interface unit for the ActiveBus, a bus designed at Iowa State University.

The ActiveBus Interface Unit (ABIU) is the chip that is placed between the physical backplane and the modules plugged in the bus slots. Since the ActiveBus is an active backplane, a driver chip is necessary at each shot to maintain the capacitance, and therefore the characteristic impedance of the bus.

In order for the bus to operate at a higher speed than the module, and to relieve the module from some of the protocol issues, some functionality and data storage must be included in the ABIU. The ABIU designed here implements the bus drivers and data buffering as well as the basic read and write protocols of the ActiveBus. The goal of this project was to provide proof that some of the concepts of the ActiveBus design will perform as expected.
CHAPTER 1: INTRODUCTION

To achieve the performance necessary in multiprocessor systems, an efficient communications medium must exist between processors. A popular and fairly inexpensive connection scheme is the communications bus. In most systems though, it is also usually the bottleneck of the system. To improve system performance, an improvement in the bus capacity must be made. This thesis describes the bus interface unit for the ActiveBus—a high speed, word serial, multiple bus system designed at Iowa State University. This bus is designed to operate at speeds of 100 MHz or more in 1.2µm or smaller CMOS technologies.

Previous work that was done with respect to the ActiveBus includes the development of some of the initial concepts by Armstrong [1]. He identified some key concepts that a high speed word serial bus must have, such as a robust method for arbitration.

This led to an organized project, funded by Control Data Corporation of Minneapolis, Minnesota, where A.V. Pohm, J.A. Davis, M.M. Hassoun, S.A. Irwin, S.W. Kenkare, S.C. Ng, and I designed the ActiveBus.

The physical layers of the bus, including the bus transceiver pads used in this design, were described in [14]. The logical layers and protocol, including a new arbitration method called Previous Priority First, were described in [16]. A design which bridges the ActiveBus interface to the AN/AYK-14, a military bus used by Control Data Corporation [5, 6, 7], was described in [18].
A typical passive backplane bus is shown in Figure 1.1. The bus drivers are contained in the module. Therefore the capacitance at each shot is dependant on whether or not the module is present in the slot. The ActiveBus, because it is an active backplane, has bus driver chips at every slot as shown in Figure 1.2. Therefore, modules communicate with the driver chips instead of the backplane. The advantage of this configuration is that the capacitance is known and therefore the characteristic impedance of the bus is known and the bus can be properly terminated at each end to eliminate reflections. This provides a cleaner environment for signal transfer, and higher speeds can be achieved.

![Figure 1.1: Passive backplane](image)
The ActiveBus is a 32 bit wide, word synchronous active backplane. The advantage of being an active backplane is that, since the modules don’t physically drive the backplane directly, an interface unit is always present whether a module is present or not. Therefore the characteristic impedance of the communications path is known whether or not there are modules in the slots. This allows proper termination of the ends of the bus which eliminates reflections. Through proper design of the protocol, higher clock speeds can be achieved.
This thesis describes the design of a prototype ActiveBus Interface Unit (ABIU) and its implementation in 2.0μm CMOS. This prototype is not intended for commercial use because of the limited transaction set and data size it supports. Chapter two gives an overview of the operational details of this prototype.

The ABIU designed for this thesis only supports a subset of the transactions of the complete ActiveBus instruction set. Supported transactions are the read and write transactions with error responses possible. The transactions not supported include the cache transactions and synchronization transactions. The protocol of the transactions supported by this ABIU is described in chapter three. This information was used to design the bus interface, which is described in chapter four.

Further analysis revealed that the ABIU must be able to store complete transactions internally. Chapter five presents the buffers designed for this prototype. These buffers only support transaction sizes of less than or equal to four words. This is a severe limitation in a commercial application, but is sufficient for purposes of testing this prototype.

Chapter six presents a set of simulations demonstrating the operation of the ABIU and some of its internal components. Conclusions are presented in chapter seven.
CHAPTER 2: ABIU OPERATIONAL CHARACTERISTICS

The ABIU is the interface point for modules to use the ActiveBus. This chip translates the non-TTL logic levels on the backplane to TTL levels for the module, as well as following and generating the ActiveBus protocol. Because the ABIU is the module’s interface to the bus, and the bus is operating at a very high speed, the ABIU should accommodate modules operating at any speed. Since the module could be either a master or slave, the ABIU must be able to handle both ends of a transaction. The transactions supported in this project are the normal write, normal read, and error responses, including split transactions.

2.1 Slave Transactions

Slave transactions are transactions in which the module (and ABIU) are being asked to perform a function. This function is defined by the abilities of the module, and may be anything from reading the value of a memory location, to moving the ailerons on an F-16.

- The ABIU constantly watches the bus and the module interface. When there is a bus transaction which was not originated by its module, a state machine in the backplane interface which handles being a slave is activated.
- The backplane interface decodes the control word, the first word transferred on the bus, to see if the transaction is a read or write, and the length of the transaction.
- The control word is passed to the buffers along with a start signal, with the address word immediately following, since it also immediately follows on the backplane.
• If the transaction is a write, the data also immediately follows.

• If the transaction is a read, the backplane interface passes the control and address words to the module and waits for it to respond with either data or a split response request. Data is signaled by the assertion of the load pin of the ABIU.

• If an error occurs in either transaction possibility, the module puts a status word on the module interface and asserts the AI line, and one clock cycle later releases the AK line. This is the proper protocol for an error condition on the ActiveBus. The status word is latched and put on the bus with the corresponding AI assertion and AK release one \( s_{\text{clk}} \) cycle later, which is echoed to the backplane.

2.2 Master Transactions

Master transactions are transactions in which the module (and ABIU) asks another module to perform a function. This function is defined by the abilities of the slave.

• When the module asserts the bus request line, this starts the master state machine in the backplane interface. The control word, address, and data must have already been loaded into the buffer by this time.

• Arbitration for the bus begins at the end of the current transaction.

• If the bus is won (see chapter 3 for more details on arbitration) the backplane interface asserts the strobe line on the bus signaling the transfer of the control word, because it is already valid on the bus.

• The address is similarly strobed onto the bus and if the transaction is a write, the data immediately follows.

• If the transaction is a read, the backplane interface waits for the slave to respond with data and strobes, and passes them up to the buffer as they arrive with the strobe line on the bus.
2.3 Design Strategy

A block diagram of the ABIU is shown below in Figure 2.1. The ABIU is basically broken up into two separate blocks, the module interface and the backplane interface. The dashed line horizontally through the center is the basic splitting location between the two portions. The top half is the module interface and buffers, and the bottom half is the backplane interface.

The module interface logic block includes the combinational logic and the bidirectional pads used by the module interface control block for communication with the module and the protocol defined there.

Figure 2.1: Block diagram of the ABIU
The buffers are used for temporary storage of the control word, address word, and data words while the transaction is taking place. These buffers are necessary because of the timing differences between the backplane and the module.

The backplane interface control block contains the state machines used to control the backplane interface. The arbitration circuitry and the transceiver pads are also controlled by the backplane interface control block.

The ActiveBus backplane logic levels are non-TTL signals asserted low as shown in Figure 2.2. The idle voltage on the bus is 2.5 V, which represents a logic 0. When an ABIU asserts a line on the bus, it sinks 20 ma of current. Since the bus has an experimental characteristic impedance of 50 Ω, the current sink produces a 0.5 V drop in the line voltage (50Ω in parallel with 50Ω = 25Ω : 25Ω x 20 ma = 0.5 V). Therefore, if a reference voltage of 2.25 V is established, any voltage below this reference is read as a logic 1.

![Figure 2.2: ActiveBus signal levels](image)

It is also important in some places to know if more than one module is asserting a line, such as the BB line described below. This is called a collision and is detected by reading the input line on the transceiver pad [14] while asserting the output line. When
there is a collision, the voltage will drop 0.5 V for every ABIU asserting this line. A reference voltage of 1.75 V is used to detect collisions. When an output is asserted, the voltage comparator's reference voltage in the transceiver pad is lowered 0.5 V to give the 1.75 V reference so collisions can properly be detected.

2.4 Pin Description

The prototype ABIU is currently being fabricated through the MOSIS (Metal Oxide Semiconductor Information Systems) program and will be put in a 108 pin grid array package. There was room for only 106 pads within the 4.6 x 6.8 millimeter die specified so two pins were left unconnected. The connected pins are described below, split into three groups: the bus interface, module interface, and other.

Table 2.1. Description of the ABIU pins corresponding to bus signals

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD[31:0]</td>
<td>Address/data transfer</td>
<td>32</td>
</tr>
<tr>
<td>ST</td>
<td>Strobe</td>
<td>1</td>
</tr>
<tr>
<td>BB</td>
<td>Bus busy</td>
<td>1</td>
</tr>
<tr>
<td>AK</td>
<td>Acknowledge</td>
<td>1</td>
</tr>
<tr>
<td>AI</td>
<td>Acknowledge inverse</td>
<td>1</td>
</tr>
<tr>
<td>FL</td>
<td>Flag</td>
<td>1</td>
</tr>
<tr>
<td>m_clk</td>
<td>High speed clock</td>
<td>1</td>
</tr>
<tr>
<td>s_clk</td>
<td>Synchronization clock</td>
<td>1</td>
</tr>
</tbody>
</table>

Number of pins in the bus interface 39
The AD[31:0] lines are used for normal data transfer of 32 bit words, as well as arbitration, and error status return. There are AD[31:0] on both the bus side and on the module side.

The strobe line is used to strobe the control word, address, and data across the bus. Since the rate at which data is transferred on the bus is higher than the propagation delays of the signals at the speed of light, there is the possibility of clock skewing. To eliminate clock skewing, the sending party, also sends a strobe with each word transferred. This way, the data and strobe signal propagate down the bus at the same rate. The receiving ABIU uses this strobe signal as the receiving clock for the data, rather than the system clocks.

The bus busy line is asserted while there is a transaction in progress. It is also used to detect collisions during arbitration. While the bus busy signal is asserted, only the master module may control the AD[31:0] lines.

The AK and AI lines are handshaking lines used by the slaves to signal different things throughout the transaction, including decoding the address, normal termination, and termination in error.

The flag line is a multiple purpose line. It is used during the arbitration process for fairness, in the previous priority first algorithm [16]. Its second use is to mark the end of data when the data length is unknown at the time of arbitration. During cache transactions the flag line is asserted by all modules participating in the transaction. This is used for the MOESI model used on the ActiveBus. For this prototype, it is used during arbitration and to mark the end of data.
The s_clk signal is the slow clock on the bus used for arbitration and handshaking. The frequency of this clock signal is dependant on the physical length of the bus. The frequency must be low enough to allow round trip propagation delays during each cycle, and be an integer fraction of m_clk cycles.

The m_clk operates at the highest speed possible, determined by the technology of the ABIU. This prototype is being fabricated in 2.0μm CMOS, thus achieving a speed of approximately 10 MHz because of the standard cells used.

Table 2.2. Description of the ABIU pins corresponding to module signals

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Reset</td>
<td>1</td>
</tr>
<tr>
<td>AD[31:0]</td>
<td>Address/data transfer</td>
<td>32</td>
</tr>
<tr>
<td>BusRQ</td>
<td>Bus request</td>
<td>1</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load buffer</td>
<td>1</td>
</tr>
<tr>
<td>ST_NOW</td>
<td>Strobe now</td>
<td>1</td>
</tr>
<tr>
<td>EOD</td>
<td>End of data</td>
<td>1</td>
</tr>
<tr>
<td>FULL</td>
<td>Buffer is full</td>
<td>1</td>
</tr>
<tr>
<td>MOD_STROBE</td>
<td>Buffer loading strobe</td>
<td>1</td>
</tr>
<tr>
<td>DATA_AVAIL</td>
<td>Data is available</td>
<td>1</td>
</tr>
<tr>
<td>LASTWORD</td>
<td>Last word in the buffer</td>
<td>1</td>
</tr>
<tr>
<td>MAI</td>
<td>Module AI</td>
<td>1</td>
</tr>
<tr>
<td>MAK</td>
<td>Module AK</td>
<td>1</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Unload buffer</td>
<td>1</td>
</tr>
<tr>
<td>!RD/WR</td>
<td>Read/Write</td>
<td>1</td>
</tr>
</tbody>
</table>

Number of pins in the module interface 45
In the ActiveBus specifications there is one clock signal, which is a dual signal line. It contains the sub-harmonic s_clk signal superimposed on the high speed m_clk signal. The same “collision detection” method is used to separate the two clock signals from the one line. For this prototype, the clock pins on the chip are actually two TTL input pins, to make testing easier.

The reset line is used by the module to reset the ABIU.

The AD[31:0] lines are used for normal data transfer of 32 bit words, as well as arbitration, and error status return. There are AD[31:0] on both the bus side and on the module side.

The bus request line is used by the module to tell the ABIU to arbitrate for the bus at the next available time. The ABIU will keep arbitrating for the bus until it is either won or the module releases this signal.

When the module wants to master a transaction, it must first load data into the ABIU buffers. This is started by putting the first word on the AD[31:0] lines, asserting the LOAD signal, and cycling MOD_STROBE once. The address and data are then loaded into the buffer with each MOD_STROBE cycle.

When the module is a slave and a read request has been done to it, it loads the requested data into the buffer and asserts the ST_NOW line.

The module tells the ABIU when the last word is being written into the buffer by asserting the EOD signal.

The full signal tells the ABIU that there is an unfinished transaction still stored in the buffer. The module may choose to overwrite this transaction if it wants, there is no
internal protection to prevent this. One example where this is desirable is as follows. The module has control of four busses. It doesn’t know which bus will be free when it is finished writing the data to the buffers so it writes the transaction to all four ABIUs. It then decides which bus it is going to use, and disregards the transactions in the other three ABIU’s. At this point the full signal is asserted by the other three ABIUs, but the module would want to write over them since that transaction has been completed somewhere else.

The mod_strobe signal is the clock signal between the ABIU and the module. Any communication between these two is performed at the mod_strobe rate, which is generated by the module.

When the ABIU receives data from another ABIU across the bus, this data is to be given to the module. The ABIU signals the module of this by asserting the data_avail signal.

When the module is reading the data from the ABIU buffer, the ABIU asserts the lastword signal when the module is receiving the last word in the buffer.

The MAI and MAK signals are used by the module to signal an error, similar to the AI and AK signals on the bus. For more information, see the bus interface chapter.

The receive signal is used by the module to tell the ABIU to send the buffer contents to the module.

R/W is used to tell which direction the AD[31:0] lines are.
Table 2.3. Description of the other ABIU pins

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTMODE</td>
<td>Enable Test Mode</td>
<td>1</td>
</tr>
<tr>
<td>TEST[7:0]</td>
<td>Test Pins</td>
<td>8</td>
</tr>
<tr>
<td>CAT[7]</td>
<td>Category 7 enable</td>
<td>1</td>
</tr>
<tr>
<td>Vdd</td>
<td>+5 V</td>
<td>6</td>
</tr>
<tr>
<td>Vss</td>
<td>Ground</td>
<td>6</td>
</tr>
</tbody>
</table>

| Number of pins for testing and power | 22     |
| Total Number of pins on the prototype ABIU | 106    |

To make testing easier, eight test pins were included in the ABIU. The testmode pin determines whether these pins are inputs or outputs. Normally they would be outputs so you can see the internal operation of the ABIU. When they are inputs, the test[7:0] pins drive the internal signals from these pins. The test[7:0] are some signals from AD[31:0]

The ActiveBus has the concept of categories [15, 17]. A module will participate in some category transactions, and not in others. There are eight categories in the specifications. Because of the limited number of pins on the package in which this prototype ABIU was fabricated, it was hardwired to participate in all even category transactions, and no odd category transactions with the exception of category seven. This category is controlled by the external CAT[7] pin.
CHAPTER 3: ACTIVEBUS PROTOCOL

The main goal of this prototype of the ABIU is to test some basic concepts of the bus, therefore only a subset of the full protocol was implemented. The transactions supported by this project are described next. For a full description of the protocol, see [15, 17].

3.1 Transactions Supported

For this prototype, only basic read and write transactions are supported, including split transactions and error responses. Before these transactions are described, there is a common denominator to all transactions that will be described first. This common denominator is arbitration.

3.1.1 Arbitration

Each transaction begins with an arbitration period, during which time all modules that want control of the bus compete for it. The winner is determined by comparing the priority levels of their transactions. If one module has a transaction priority higher than the others, it wins arbitration and proceeds with the transaction. All other modules release the bus until the next arbitration period. If two or more modules have the same priority level, the next stage of selection falls to a geographical address. Each module is assigned a fixed geographical address. Of the modules that tied for the highest priority, the one with the highest geographical address wins.

If a module ties for highest priority but does not win the arbitration because of a lower geographical address than the winner, it gets the first chance for the bus at the next
arbitration time. This state is signaled by the assertion of the FL line when BB is
deasserted at the end of the current transaction. All modules of equal priority to the last
transaction that lost the arbitration last time are guaranteed to win the bus before any
other new bus requests are considered. This arbitration method is called Previous Priority
First [16] and shows great results with little complexity.

So the arbitration formally goes as shown in the flow chart in Figure 3.1. After the
ABIU receives a bus request, it waits until BB is not asserted. It then waits one s_clk
cycle and checks if FL is asserted. If it is, the ABIU waits until the end of the next
transaction and begins arbitration again. If FL is not asserted, the ABIU asserts BB and
puts its transaction priority and geographical address on the bus. This information is
contained in bits 16-31 of the control word. It waits one s_clk cycle for the information to
propagate across the bus and then checks for a collision on the BB line. If no collision is
detected, then it is the only ABIU that is arbitrating for the bus so it wins.

If there is a collision, it must deassert FL and wait two cycles for the arbitration logic
to determine the winner. The arbitration logic is described in the bus interface chapter on
page 37. If, after the two s_clk cycle wait, the arbitration logic determines it has the
highest priority, it proceeds with the transaction. If it did not win the arbitration, it
releases BB and removes the control word from the bus. If it did not have the same
priority as the winner, it begins the arbitration process from the top again at the beginning
of the next transaction. If it tied the highest priority, instead of waiting until BB is
deasserted, it waits until AK is deasserted. When this happens, it asserts FL, telling the
other ABIUs that it tied priorities with the last transaction. It then waits one s_clk cycle
after the release of BB, and then asserts BB, and puts its control word on the bus again.
Figure 3.1: Arbitration flow chart
Once the bus has been won, the rest of the transaction can continue. As mentioned before, this prototype only supports the write transaction, read transaction, and status signaling.

### 3.1.2 Write transactions

The timing diagram for a write transaction is shown in Figure 3.2. The diagram begins after the bus has been won in arbitration. The CL signal is the clock signal on the bus. The s_clk and m_clk are not bus signals but are derived from CL. In the diagrams, s_clk and m_clk are shown separately for clarity.

Once the bus has been won, the control word is already on the bus, and therefore the master simply strobes the ST line once to tell the slaves that it is valid. The master then immediately proceeds to write the address and data if there is any (address only transactions are supported), cycling ST every word. The rising edge of ST signifies a valid word on the bus. This signal is used internally by the slave to latch the words into its buffer.

The handshaking between master and slave is as follows. In the process of arbitrating for the bus, the master asserts BB. Once the bus is won, the master starts the transaction, keeping BB asserted. When the other modules on the bus, all potential slaves, detect BB asserted, they watch for the rising edge of ST. They latch in the control word and start decoding it. Those that determine they are to participate in this transaction, assert AK and release AI. Those that determine they are not to participate, release both AI and AK. The master knows all slaves have decoded the control word when the AI line goes idle. This is not used in the write transaction, since the master operates in a no-news-is-good-news policy. It assumes that the transaction is successful unless it detects an error response.
Figure 3.2: Write transaction
When the slave(s) have received the last data word and determined that there were no errors, they release AK, and one s_clk cycle later, they assert AI. The master then knows that the transaction was successful and therefore releases BB.

3.1.3 Read transactions

The timing diagram for a read transaction is shown in Figure 3.3. Once the bus has been won, the control word is already on the bus, and therefore the master simply strobes the ST line once to tell the slaves that it is valid. The master then immediately proceeds to write the address, cycling ST again. The rising edge of ST signifies a valid word on the bus. This signal is used internally by the slave to latch the words into its buffer.

When the other modules on the bus have successfully decoded the control word and decided who the slave is, they pass the control word and address to their module. The module does the read internally and returns the requested data to the ABIU. The ABIU writes this data to the bus cycling ST with every word. The master uses the ST signal to strobe the data into its buffer. When the slave is done writing the data, it releases AK, and one s_clk cycle asserts AI. The master knows this is the end of the data and no error has been signaled so it releases BB.

3.1.4 Error signaling

In the case of an error or if the slave wants a split response, the error signaling mechanism begins. The timing diagram for an error response is shown in Figure 3.4. This particular error response is occurring during a write transaction, but it would act similarly in a read transaction.
Figure 3.3: Read transaction
The error signal is raised by asserting AI before releasing AK. When this pattern is recognized on the bus, the master knows that an error has occurred and suspends the transmission of data on the bus, releasing all AD[31:0] lines. The ABIU that detected the error and asserted AI before releasing AK puts a status word on the bus when asserting AI. The releasing of AK signals this word as being valid on the bus. The master latches this status word and passes it immediately to its module. When the master has latched the status word, it releases BB on the next s_clk cycle signaling the end of the transaction. The slave module that is driving AD[31:0] with the status word releases AD[31:0] when BB is released.

Since all ABIUs recognize the error condition and see the status word, an error logging module could be designed to capture the status words and keep a log of all occurrences with date and time stamps. This could prove to be useful in debugging bus problems.
Figure 3.4: Error transaction
CHAPTER 4: BACKPLANE INTERFACE

The backplane interface of the ABIU, Figure 2.1, has five main responsibilities. This chapter describes these five responsibilities and describes the solutions to them. These responsibilities are:

1) Generate and follow the ActiveBus backplane protocol.
2) Decode control word for r/w, data length, etc.
3) Count number of words transferred and generate an end of data signal.
4) Inform the module interface of the status of the bus.
5) Translate the TTL logic levels to the ActiveBus backplane logic levels.

To generate and follow the ActiveBus backplane protocol, the protocol and timing diagrams presented in the previous chapter were studied. The design appears to be intuitively split into three state machines - one to control the arbitration, one to control the master transactions, and one to control the slave transactions. After further study, it was discovered that there was some repetition in each state machine. This repetitive portion was broken out into a separate state machine, called the status state machine, which monitors the normal progression of all transactions. It generates signals when it detects normal termination, or termination in error. It also generates a signal when there is a status word valid on the backplane after an error has been detected.

To decode the control word, a combinational logic circuit was designed to latch the control word on its transfer and decode it. The signals produced by this block are
read/write, category match, data length, and end of data (EOD).

To count the number of words transferred on the bus, the control word decoder block has a 12 bit counter that counts the number of strobe pulses going to the bus or coming from the bus, depending on the transaction. When the number of words transferred equals the transaction length, as specified in the control word, the decoder block asserts an end of data signal.

To inform the module interface of the status of the bus, the state machines communicate with the buffers and generate the necessary signals for the buffers and the module.

To translate the TTL logic levels to the ActiveBus backplane logic levels, a transceiver pad was designed [14], which serves as a logic level translator. It also has the ability to detect collisions.

Therefore the backplane interface consists of four state machines and one major block of combinational logic. The four state machines, status, master, slave, and arbitration, were derived from the timing diagrams described in the last chapter. The following sections describe each state machine in detail. After the state machine descriptions, the schematics of the bus interface are described.

4.1 Status State Machine

The status state machine, shown in Figure 4.1, is in charge of watching the natural progression of transactions on the bus and generating signals that are used by the rest of the state machines and the combinational logic.
The state machine starts in an idle state and watches for a transaction to start, which is signaled by a BB assertion. At that point, it goes to a state which watches for the slave to successfully decode the address and participate in the transaction, AK&!AI. It then watches for either an error to be signaled or a successful completion of the transaction. A successful completion is signaled by the release of AK before the assertion of AI. An error is signaled by the assertion of AI before the release of AK. At the recognition of an error condition, the status state machine waits for the status word to become valid on the bus (AI&!AK) and asserts a signal which tells the module interface to latch the status word now.
The responsibility of the master state machine, shown in Figure 4.2, is to produce the required signals, internal and external, for a normal transaction when the module driving the ABIU is the master.
The master remains in an idle state until the arbitration state machine asserts a signal telling the master state machine that it has won the bus. The control word is then strobed on the backplane (for the benefit of the slaves) and the address is also strobed on. At this point, the master checks the decode_cw logic block to see if the transaction is a read or write, and if the data length is greater than one word. If the transaction is an address only transaction, the master immediately goes to the done state and waits for the slaves to signal the completion of the transaction.

If the transaction is a write with data length greater than one word, the master slave machine goes to the write state and strobes data out until end of data is asserted, or an error is detected by the status state machine. When either of these conditions is true, the master state machine waits for the end of the transaction.

If the transaction is a read, the master state machine goes to the read state and passes the strobe signal of the backplane to the buffers so they can latch in the data when it arrives. It does this until end of data is asserted, and finishes the same as above.

4.3 Slave State Machine

The slave state machine, shown in Figure 4.3, is responsible for handling the slave side of the ActiveBus backplane protocol. This includes reading and writing data, and controlling the AI and AK lines on the backplane.

The slave state machine starts in an idle state and waits for BB to be asserted. When it is, and the ABIU is not the master, the control word and address word are passed with strobe to the buffers. If the transaction is a write, determined by checking the decode_cw logic block WR signal, the data is strobed to the buffers until end of data is asserted (remember, a write transaction means the slave must read the data).
Figure 4.3: Slave state machine
When end of data is asserted, the slave waits one cycle for the last data word to be strobed up, and releases AI and AK. It then asserts AI one s_clk later to signal successful completion of the transaction and waits for BB to be released and goes back to its idle state.

If the transaction is a read, the slave state machine waits for the module to assert the eod signal. When eod is asserted, it strobes data from the buffers to the bus until end of data is asserted, and finishes up the same way the write transaction did.

If in either the read or write, the module wants to generate an error or split response, it asserts the MAI line. When the slave state machine detects that, it asserts AI on the backplane, strobes the status word down from the buffers to the backplane, and waits for the module to release MAK. When it has released MAK, AK is released on the backplane and the state machine waits for BB to be released by the master, and then goes to the idle state to start over again.

### 4.4 Arbitration State Machine

The arbitration state machine, shown in Figure 4.4, is responsible for gaining control of the bus when the module wants to be master. If it cannot gain control of the backplane, it continues to try at the beginning of each new transaction until it either wins, or the module gives up by deasserting BusRQ. When the bus was won, the arbitration state machine tells the master state machine that it has won and the transaction proceeds.

The arbitration state machine was derived from the arbitration flow chart [16]. The arbitration flow chart is shown on page 17.
Arbitration

Figure 4.4: Arbitration state machine
The state machine begins in an idle state and looks for a BusRQ from the module. It then looks for BB to be released at the end of the current transaction. When the current transaction ends, it checks to see if flag is asserted. If it is, it waits for the next transaction completion, otherwise it asserts BB and puts the top 16 bits of the control word on the bus for arbitration. Next cycle, it checks to see if a collision has occurred on the BB line, detected by BB_IN asserted when BB_OUT is asserted in Irwin’s transceiver pads. If not, the bus was won and the arbitration state machine asserts a signal telling the master state machine to begin. If a collision occurred, the state machine waits two s-clk cycles and checks to see if the arbitration was won. This gives enough time for logic delays plus two end to end propagation delays on the bus. If it was won, it informs the master, otherwise it releases the priority word from the bus and waits until the end of the current transaction. Once BB is released at the end of the transaction, it asserts the flag signal. This gives this module, and other modules that were the same priority during the last arbitration, priority in this transaction. New modules do not enter arbitration until all previous modules of the same priority have completed their transactions.

4.5 Schematics Description

The schematic for the bus interface is shown in Figure 4.5. It consists of the four state machines described above, the block labeled decode_cw, and some other glue logic needed to generate the necessary signals.

The design requires the decoding of the control word and counting the number of data words transferred on the bus so that control signals can be generated. The decode_cw logic block was designed to accomplish this task. The logic generates six signals. All are derived from the contents of the control word.
Figure 4.5: Bus interface schematic
The six signals that the decode_cw logic generates are:

1) CATEGORY_MISS - The current transaction is not a member of the classes supported by the module attached.
2) DLEQ0 - Data length equals zero (Address only transaction)
3) EOD - (end of data) - Asserted when the word being transferred is the last one for this transaction.
4) FLTOBUS - Used to signal end of data on the bus.
5) RD - Current transaction is a read.
6) WR - Current transaction is a write.

In order to generate these signals the following inputs were needed:

1) AD[31:0] - The address and data bus
2) CAT[7:1] - The category support flag bus
3) BUSFL - The flag signal input from the bus
4) CW_ENABLE - A signal from the buffers asserted when the control word being output to the bus is valid
5) LD_CNT - A signal from the slave state machine
6) CW_VALID - A signal indicating the time when the control word coming from the bus is valid
7) BB - Input from the bus
8) GST - The global internal clock controlled by the state machines

Inside of decode_cw, a 4:16 decoder and 12-bit counter are required, along with some other logic (see schematic in Figure 4.6). The input to the 4:16 decoder comes from the data length field of the control word. One output of 16 is asserted low when the control
word is valid. The 12-bit counter loads the ones complement of the count value plus one. The counter then counts down the number of strobes on the bus. This includes the address word, since it is the first word strobed to the bus after the control word. Therefore the counter must count the number of data words transferred plus one. The output of the counter is fed into an OR structure. So if the count equals one or zero, end of data is asserted. The EOD signal indicates the last valid word incoming, but since there is a measurable delay to generate this signal, EOD is asserted one data item before the actual end of data. This signal is only used at the rising edge of the m_clk cycles, so the early generation of the signal is not a problem.

If the 0th output of the encoder is asserted, the data length of this transaction is zero and the DLEQ0 line is asserted and latched. In this case, the end of data line is also asserted since the address word is the last word transferred. Remember, the end of data signal is asserted at the beginning of the last word transferred. If the 15th output of the encoder is asserted, the data length is unknown so the flag line on the bus marks the end of data. Therefore the flag line on the backplane is passed to the end of data output. The flag line is asserted at the beginning of the transfer of the last data word.

The read/write signals are generated by latching the 9th bit of the control word (0=RD, 1=WR). This is the least significant bit of the command field. Since all commands in the ActiveBus protocol use this bit as the read/write bit, that is the only bit that needs decoding for this prototype. Additional command word bits must be decoded for a commercial application, but in this project only basic read and write transactions are supported.
Figure 4.6: Decode cw schematic
The contents of the counter and latches are latched on the first strobe from the bus when the ABIU is the slave. When the ABIU is the master, the control word is valid when the master state machine asserts the cw_enable signal, or when the buffers signal the transferral of the control word.

The CATEGORY_MISS signal is produced by comparing the category of the current transaction with those categories that the module can participate in. This prototype is hardwired to participate in all even categories and not participate in any odd categories except for category seven. Category seven is an external input to the circuit. When wired to Vdd, the ABIU will participate in CAT[7] transactions, when wired to Vss, it will not.

4.6 Arbitration logic

The arbitration process requires some dedicated logic to determine the winner. This circuitry must report the results to the bus interface for use in the arbitration algorithm. Figure 4.7 shows a gate level equivalent schematic for the arbitration circuitry used. This circuitry was actually implemented in full custom CMOS for maximum speed [14]. Sixteen of these blocks are stacked on top of each other for the sixteen bits used in arbitration.

The arbitration process starts with the most significant bit of the control word put on the bus and if there is no collision, the next stage is enabled. This ripples through the priority bits (31:16) of the control word. If the last bit has an asserted output, the bus was won.
The level translation between CMOS TTL and the bus logic levels is done in the transceiver pads [14]. These pads have one input and one output as shown in Figure 4.8. The output is used to assert and not assert the signal on the bus. The input tells you if the bus signal is asserted, and if there was a collision on that pin. A collision is detected when you are asserting the output pin and the input pin is also asserted. Since it is only necessary to detect a collision if we are a part of that collision, this is sufficient.

Figure 4.7: Arbitration circuitry

4.7 Bus Pads
Figure 4.8: Bus transceiver schematic
CHAPTER 5: BUFFERS

Because the bus clock speed is typically at least twice as fast as a module’s clock speed, the ABIU must store a complete transaction internally before it starts arbitrating for the bus. The ABIU must wait until it knows all of the data is available to be transferred before it commits the bus to this transaction. This will provide the most efficient use of the bus. It is the module’s responsibility to provide all of the data to the ABIU before asserting the bus request line.

The same is true for the case of an incoming bus transaction going to a module. The ABIU must store the transaction data until the module is ready to receive it. The module is typically much slower than the bus speed, so it cannot handle data as fast as the bus can provide it. The ABIU would also have to store the data if the module was busy doing something else.

As mentioned before, the module and bus typically run at different speeds, so there must be a mechanism where synchronization occurs between these two clocks. The data buffers offer a convenient place to put this synchronization point. In a typical transaction, the module writes the control word, address, and complete data into the ABIU at the module clock rate. When it is done with this, it asserts the bus request signal. This signal tells the ABIU to begin arbitration, at the high speed bus clock rate. The ABIU completes the transaction at the bus speed.
The above arguments show that the ABIU must indeed buffer complete transactions in both of its operational directions. Therefore two buffers were included. The buffer design will be described next, followed by schematics of the actual buffers designed.

5.1 Buffer Design

In designing the buffers in the ABIU, a model was developed to analyze the effectiveness of each design. The model consists of two modules communicating across the bus through two ABIUs. Propagation delays on the bus are ignored for this model. A picture of the basic model is shown in Figure 5.1.

Figure 5.1: Buffer analysis model
When the needs of the system are analyzed, it is apparent that the following features are important:

- Scalability of design. More storage added easily?
- Speed of operation. Must be able to operate at bus speed.
- Size of complete buffers. Smaller is better.
- Must not impose any time penalty on the transaction
- Must be able to operate at two speeds.
- Must be able to store the data when no clock is present (static storage).

5.1.1 Buffer is a FIFO

The simplest and most intuitive solution for storing the transactions is a simple synchronous FIFO. The master module shifts down the control word, address, and data. The transaction data is shifted to the bottom of the FIFO. At this point, the bus arbitration can begin.

The problem with this solution is the delay necessary to always shift the data to the bottom of the FIFO. If transactions of length 256 would like to be supported, all transactions, including address only transactions, would require 256 shifts to reach the bottom of the FIFO. The example below illustrates this problem.

*Example: Master read from Slave*

1) The master module shifts the control word into the ABIU buffer, then the address, and then shifts these to the bottom of the FIFO. This requires 256 shifts and therefore 256 module clock periods.
2) When the data is shifted down to the bottom of the FIFO, as shown in Figure 5.2, the module asserts the bus request line, indicating that it wants control of the bus.

3) The ABIU then arbitrates for the bus. When the bus is won, the ABIU shifts the control word, address, and data onto the bus. One word per bus clock cycle. Therefore this requires 256 bus clock cycles.

![Diagram](image)

Figure 5.2: Outgoing buffer no. 1 full (before data transferred on bus)

4) When the slave ABIU receives this read request, it shifts the control and address into the buffer, and to the top, as shown in Figure 5.3. This requires 256 bus clock cycles.
5) The module shifts the control and address out of the buffer and does the read internally.

6) It shifts the resulting data into the ABIU, and to the bottom of the FIFO. This requires 256 module clock cycles.

7) The data is shifted across the bus without a new arbitration necessary as the bus is held during this time.

8) The data arrives at the master ABIU and is shifted into the buffer. At this point the bus can be released.

9) The data is shifted to the top of the buffer and into the master module. This requires 256 module clock cycles.

Figure 5.3: Incoming data buffer no. 2 full (after data transferred)
As can be seen, the total delay for this worst case transaction is quite large. An analysis of this delay is shown below. If the bus clock period, B, is 10 ns and the module clock period, M, is 20 ns, the delay for this transaction follows.

- Delay in shifting control and address to bottom of master buffer: 256M.
- Shifting control and address to slave module: 256B.
- Shifting slave data into buffer: 256M.
- Shifting data onto bus and into master buffer: 256B.
- Shifting data from master buffer to master module: 256M.

Total delay = 768M + 512B = 20.48 μs!!

Total time holding bus = 256M + 512B = 10.24 μs!!

This buffer strategy clearly imposes too much of a time penalty on the transactions. One of the criterion of a good buffer is to impose no delay. Another strategy must be found.

5.1.2 Buffer is a RAM Queue

The next most intuitive solution is to emulate the FIFO with a RAM FIFO queue, as shown in Figure 5.4.

The advantages of this design is that the data does not have to be "shifted" down to the bottom of the FIFO. This eliminates the time penalty imposed by the first strategy. Another possible advantage to this strategy is that it would be possible for the module to simultaneously read data from the ABIU buffer while the buffer is still being filled from the bus. This would require a dual port RAM and some other control logic to guarantee that the rules are not violated, but this would also speed the transactions because of the overlap of functions.
The disadvantages of the RAM are that it is more complex in design and adds a lot of wiring to the buffer. It is also not as scalable as the FIFO, in that the head, tail, and count pointers would have to be made the right size to address the full space. This would also require a very fast RAM. It would have to be fast enough to read the head pointer and use it to find the data, write the data, modulo increment the head pointer, and increment the count field within each bus cycle. In the example above of a 10 ns bus clock period, the RAM would have to be at least 5 ns if not faster. This gets quite costly and complex.

5.1.3 Buffer is a hybrid FIFO

The ideal solution seems to lie somewhere in between those two strategies. The chosen solution was to have a FIFO in which the output could be taken from every N cells, as shown in Figure 5.5. The outputs would go into a MUX which would select
which output to route to the bus. This solves the time delay problem of the long shift register, and avoids the problems of the RAM solution. Now the maximum number of shifts required to get the data to the bottom of the FIFO is \( N \), where \( N \) is the number of FIFO cells between successive outputs to the MUX.

The disadvantages of this solution are that there is still a small delay imposed, although that will be proven to not be a problem in most cases. The FIFO is not dual ported as the RAM is, and there is still a lot of extra wiring more than the straight FIFO.

The reason that the delay imposed is not a problem, is that the delay of \( N \) is overlapped with other operations so this delay is only seen in one case. For this design, \( N \) was chosen to be four words. Now the maximum number of shifts required is four. In a
transaction, the module loads all of the data for the transaction into the ABIU. It then asserts the bus request signal which signals the ABIU to start arbitrating for the bus. The ABIU uses only the control word for this arbitration, and the minimum time for arbitration is three bus clock cycles. Therefore to have no time penalty the control word and address word are stored in a two word FIFO, separate from the data FIFO as shown in Figure 5.6. This way the control word is available immediately, and if the arbitration only takes three cycles, the address word is also immediately available to fill in the fourth clock cycle. By this time the data is guaranteed to have reached an even N boundary.

Figure 5.6: General buffer structure
The two buffers are nearly identical except for a few things. Therefore the down buffer will be presented first and the up buffer will be presented after that explaining only the differences between the two.

5.2 Down Buffer

The down buffer stores the data which comes from the module and is going toward the bus. The following circuitry, is necessary to accomplish this task (see Figure 5.7).

There are two storage units in the down buffer. One storage unit stores the control word and address, and is a fixed length two word storage unit. The other storage unit stores the data for the transaction. In this chip, it is a four word storage FIFO, but in general it could be any length with an output every four words. The tag output (described later) of each block would be used to select the multiplexed output to choose. The prototype chip only contains four words of storage because of size constraints. The control and address words are stored separately because they are always present in all transactions. By storing these separately, more time is given to shifting the data words to the bottom of the data storage unit.

The outputs of the two storage units are multiplexed to the output of the buffer. The output MUX is simply a full custom 2:1 multiplexer implemented using pass transistors.

There is a D flip flop used to signal when the buffer contains data and when it is empty. The state machines control this signal. The signal is not used internally but supplies the signal to the module and bus interface. If the buffer is full, another transaction should not be started unless the previous transaction is being cancelled.
Figure 5.7: Down buffer schematic
5.2.1 Clock generation circuitry

The trick in this scheme is the clocking. The clock generation circuit is shown in Figure 5.8. It is broken into three main parts:

- Asynchronous state machine to enforce clock switches only when both clocks are low.
- Clock Enable
- Two phase clock generator

The left most group of logic is an asynchronous state machine which forces clock switching only when both the source and destination clocks are both low. It takes the three clock sources and three clock enable signals. Whenever the clock source is to be changed, signaled by an assertion of one of the enable signals, the asynchronous state machine delays the clock change until both the original clock and the new clock are both in the low state. This insures glitch free clock switching.

The vertical logic near the center of the schematic is the clock enable signal. The clock is only enabled when the enable signal is asserted and when either the tag is asserted or the e_tag signal is asserted. These signals are controlled by the state machines and the tag output of the data buffers.
Figure 5.8: Clock generation circuitry
The circuitry in the lower right corner of the schematic is the two phase clock generation circuitry. It generates the two phase clock required by the data storage elements described below.

5.2.2 State machines

The two state machines are responsible for controlling the signals in the buffers. Enabling the shift elements, asserting the tag signal, controlling the output MUX, etc. The description of the contents of these state machines is given later.

5.2.3 Reset

The reset circuitry in the north-west corner of the schematic is used to guarantee the correct initialization of the tag storage in the storage units. The reset block activates on the assertion of the reset signal and enables the fast bus clock for the rest of the logic internally until four clock cycles have completed. This will successfully initialize the tag storage in the storage units.

5.2.4 Storage unit descriptions

The goal for this design was to provide a high speed buffer whose internal circuitry was simpler than a RAM, but provided no performance degradation to the ActiveBus. This meant that the control word had to be available to the ABIU whenever the module requested the bus, and the data would be available by the time the ABIU was done with the bus arbitration. This was accomplished by providing two buffer paths. A two word buffer which stored the control and address words, and a FIFO chain to store the data. The two word buffer was included so the control word is immediately available with no shifting required.

The storage elements had to be static storage because not only is it not known at what
speed the module is operating at, but the module is allowed to load the buffer and delay for an indeterminate amount of time before it tells the ABIU to use the data in the buffer.

The basic storage element for all of the data storage is a pair of cross coupled inverters as shown in Figure 5.9. The transistor schematic of this cell is shown in Figure 5.10. The transmission gate is on the beginning of the storage cell so this implies that the clock signal allows the data to flow into this cell, as apposed to allowing the data to flow out of the cell if the transmission gate was on the output of the cell. This decision was made to allow the output of the last storage element in each bit row to be enabled at all times. This was important because the tag bit output must be valid at all times. The layout of the cell is shown in Figure 5.11. Much time was spent optimizing this cell to reduce its size, as it was going to be replicated many times. The final size was 50.5 by 42.5.

![Figure 5.9: Basic storage element, logical view](image)

![Figure 5.10: Basic storage element, transistor view](image)
Figure 5.11: Basic storage element layout
Each storage unit contains a number of shift cells to store the data. Each bit of storage requires two storage shift cells. This is because of the necessary two phase clock.

Figure 5.12 below shows how four data items are stored into the two phase FIFO. Each data element requires two storage elements because of the two phase clock. But the two phase clock is necessary to guard against a race condition. The clock signals open the gates at the front of each storage cell, so on phase 1, the first data element is allowed into the first storage cell. At phase 2, that data element is allowed into storage cell 2. This cycle continues and the figure shows that $2 \times N$ storage cells are required to store $N$ data elements.

![Figure 5.12: Each bit being stored requires two storage elements](image)
When 33 of these storage elements were placed end to end, the capacitance on the clock lines became quite large (~1 pf each). Therefore clock drivers were designed to increase the rise and fall times of the clock signals. Figure 5.13 shows the layout of the clock driver.

If one big shift register was made to store 256 words, 32 bits wide, the worst case delay through the buffer would be 256 shifts. By breaking down the shift register into multiple, four word by 32 bit shift registers, and allowing the output of the buffer to come from the output of any of these four word blocks, the worst case delay has been reduced to three shifts.

Since, upon assertion of the bus request line, the ABIU needs the control word for arbitration, the control word must be available immediately, a separate FIFO was included to hold the control word and address word. The outputs of the control word FIFO and the data FIFO were multiplexed for the output of the buffer unit. The first two words stored in the buffer, control and address word, go to the control word FIFO, all others go into the data FIFO. Likewise, the first two words out of the buffer are from the control word FIFO, all others from the data FIFO.

While the ABIU is using the control word for bus arbitration, there is time before the data is needed. This time delay is used to shift the data words to the bottom of the FIFO. The best case delay for arbitration is two s_clk cycles. The worst case delay for shifting the FIFO to a valid output is three clock cycles. Therefore, by having the control word and address word stored separately, the buffer will add no delay to the data transfer of the ABIU.
Figure 5.13: Layout of clock driver
Some extra control was necessary at this point since the data always needs to be shifted to the bottom of the four word shift bank. The module can load any length data into the buffer, so a method was needed to keep track of the first word in the data FIFO and shift this word to an even N word boundary. This was implemented using one more bit of storage. The 33rd bit is a tag bit, which is initialized to all ones at reset, and the state machines load in a zero when the first data word is written to the FIFO. Then, when the module has loaded in its last word, the data FIFO keeps shifting at the bus clock speed until the tag output equals zero. This signals the front of the FIFO contains the first data word. The tag is also used to select which four word FIFO to use the output from.

When shifting data out of the buffer, the bus interface decodes the data length field in the control word and shifts that many words from the data FIFO.

5.2.5 State Machine Descriptions

Two state machines are necessary to control each buffer; one to control loading the buffer, and one to control unloading the buffer. The state machines are similar for each buffer but they have slight differences.

The loading state machine, shown in Figure 5.14, has the responsibility of storing the first two words into the control word buffer and all subsequent words into the data buffer. When the first word is stored into the data buffer, the state machine must force a zero into the tag field so this word is marked as the first in the buffer.
The unloading state machine, shown in Figure 5.15, has the responsibility of retrieving the control word, and address word from the cwaddr buffer. All subsequent reads come from the data buffer until the end of data is reached. This is signaled by the bus interface.

For more detail see the VTI textual state machine descriptions in Appendix A.
The only additional logic added to the up buffer is that the eod signal is not an input but rather an output, see Figure 5.16. The tag output of the buffers is used to signal the last word in the up buffer. This signal is latched in an asynchronous state machine which is just an edge triggered D type flip flop. The output goes to the up_out state machine and to the module.
Figure 5.16: Up buffer schematic
CHAPTER 6: RESULTS

This chapter presents results of simulating the ABIU presented in this thesis. Each stage of development was fully tested before being used in the ABIU, and the simulations presented here are a representative set of some of the key simulations.

The development of this ABIU was done using VLSI Technology’s VTI tool set version v7r5. This package was running on HP 9000/425 workstations.

6.1 Bus Interface

The bus interface was developed first. Using the timing diagrams of the protocol [15, 17], the state machines in chapter four were developed. This circuitry was tested in a master read transaction, a master write transaction, slave read transaction, a slave write transaction, and a slave read transaction which terminates in an error response. Although these transactions were tested with various data lengths, the simulations presented here are with data lengths of four for consistency. When these simulations are compared to the waveforms in chapter three, some similarities should be noted. Each simulation is described below.

6.1.1 Master write

The simulation of a master write transaction is shown in Figure 6.1. The simulation starts by the assertion of the reset signal. This brings the ABIU to a known condition for the start of the simulation. All simulations will start with the reset signal.
Figure 6.1: Master write
The first thing to happen after the reset is the assertion of BusRQ. This tells the ABIU that the module wants control of the bus. The ABIU checks the BB signal and waits one s_clk cycle and begins arbitration. The ARB signal is asserted internally while arbitration is taking place. It knows it has won the arbitration because there was no collision on BB. The WON_ARB signal is then asserted. One s_clk cycle later, the ABIU strobes the control word, address word, and data to the bus. The decode_cw logic block correctly counts the number of words for the transaction and asserts EOD when the last word is reached. The BB signal is then released. This marks the end of the transaction.

6.1.2 Master read

The simulation of a master read transaction is shown in Figure 6.2. After the reset the next thing to happen is the assertion of BusRQ. This tells the ABIU that the module wants control of the bus. The ABIU checks the BB signal and arbitrates for the bus as in the master write transaction. It knows it has won the arbitration because there was no collision on BB. The ABIU strobes the control and address words to the bus and waits for the requested data to return from the slave. The data is returned from the slave with the ST signal being cycled. The master ABIU watches this signal (ST_FROMBUS) and latches data into the buffer when it occurs. The decode_cw logic block correctly counts the number of words for the transaction and asserts EOD when the last word is received. The BB signal is then released which marks the end of the transaction.

6.1.3 Slave write

The simulation for a slave write transaction is shown in Figure 6.3. A slave write transaction is one in which the ABIU is currently a slave and is being written to by the slave.
Figure 6.2: Master reads
Figure 6.3: Slave write
After the typical reset for the simulation, the transaction starts. A slave transaction is started by the BB signal on the bus being asserted when the ABIU is not asserting it. All ABIUs not mastering a transaction are potential slaves. The slave ABIU receives the control word and address and decode their contents. When this process is complete and it recognizes that it is a potential participant, it releases AI and asserts AK. It continues to shift the data from the bus to the buffer as long as the EOD signal is not asserted. The EOD signal is asserted on the last word indicating the end of the data transfer. The slave ABIU then releases AK and asserts AI one s_clk cycle later indicating a normal transaction completion.

6.1.4 Slave read

The simulation for a slave read transaction is shown in Figure 6.4. It starts with the reset of the ABIU and watching BB as with the slave write transaction. When the control word and address are read, the ABIU releases AI and asserts AK and waits for the module to respond with data indicated by the assertion of ST_NOW. In this simulation, the module asserts ST_NOW immediately, but it typically will come some time later. The ABIU then strobes the data to the bus and finishes up the transaction with the release of AK and one s_clk cycle later the assertion of AI.

6.1.5 Master write error

The simulation of an error transaction is shown in Figure 6.5. This transaction starts out as a write transaction. The slave detects an error, so signals this to the master by asserting AI before releasing AK. The master ABIU detects this and stops strobing data to the bus. It waits until AK is released and then latches the status word from the bus. Once it has latched the status word, it releases BB, which ends the transaction.
Figure 6.4: Slave read
Figure 6.5: Slave read ending in error.
6.2 Buffers

Once the bus interface was complete and working, the buffers were designed. The basic storage element was designed and optimized for speed and size. Then it was stacked together to make the full storage unit size (32x4 etc.). When they were simulated, severe clock loading was noted. Clock drivers were then designed and simulated.

Once the storage units were complete, the clock generation circuitry was developed and tested. A simulation of this is shown in Figure 6.6. Notice how the clock enable signals do not change until all of the clock signals are not-asserted. This gives the cleanest switches and is due to the asynchronous state machine developed for this task.

The buffer was tested as a whole and the simulation is presented in Figure 6.7. This simulation shows two complete write-read cycles of the up buffer. The simulation starts with the assertion of reset. This brings the buffers to a known state to start from. The bus interface then asserts the load signal and strobes the bus_strobe to shift in the control, address, and data. Notice how the cwaddr clocks are cycled only twice and the buffers clocks are cycled throughout the whole write. This is because the cwaddr holds the control word and address word. The buffers clocks control the loading of the 32x4 buffer to store the data. After the bus interface asserts eod signaling the end of data, the buffer switches to the high speed m_clk to shift the data to the top of the FIFO. The shifting continues automatically until tagout is asserted. The module then asserts RECEIVE to indicate that it is ready to receive the data. It cycles mod_strobe to receive each control, address, and data word. The m_eod signal is asserted when the last data word is shifted out. This is repeated a second time to verify that all internal signals are properly reset between transactions.
Figure 6.6: Clock generation circuit
Figure 6.7: Up buffer
Figure 6.8 shows a plot of the layout of the ActiveBus Interface Unit described in this thesis. The chip is 4.6mm by 6.8mm and is being fabricated through the MOSIS (Metal Oxide Semiconductor Information Services) program. The chip will be put in a pin grid array package with 108 pins, of which only 106 are used. Two pins were left unconnected because of the lack of room for two more pads in the core.
CHAPTER 7: CONCLUSIONS

This thesis presented the design and implementation of a prototype ActiveBus Interface Unit. The ABIU presented here is only a prototype in that it does not implement the complete ActiveBus protocol. It was designed in full custom and standard cell 2.0µm CMOS. To achieve the target speed of 100 MHz, a smaller technology should be used like 1.2µm or 0.8µm CMOS.

7.1 Contributions

The contributions of this research are as follows:

1. The design and implementation of the first ActiveBus Interface Unit.

2. A slight modification to the Previous Priority First arbitration algorithm to make implementation easier. The strobe line is cycled once by the master when it knows it has exclusive control of the bus. This makes the storage of the control word by the slaves easier.

3. Developing the first state machines to handle the ActiveBus protocol. This includes four state machines which interact with each other.

4. The design of a scalable transaction storage unit. This buffer design is both scalable and imposes no time penalty on the transaction. It is a static storage design which can be packed quite dense.
5. Developing a simple yet effective module interface. This interface is simple to implement on the module because it operates at the module’s clock speed.

6. Providing some feasibility in the ActiveBus design. This ABIU was successfully simulated, and no problems are anticipated in testing the chip when it returns from fabrication.

7.2 Future Directions

When the chip arrives from the foundry, it should be thoroughly tested. This includes placing two or more ABIUs on a backplane and testing some transactions. These tests will not be able to run at the target speed of 100 MHz, but at a much lower speed near 10 MHz. For higher speeds, an ABIU must be fabricated in 1.2µm or 0.8µm CMOS.

Then a complete prototype system should be built, including some modules. This system should be tested under extreme conditions such as heavy loading and see what modifications need to be made in the protocol or the design to improve performance.

It is possible to tune the system according to the application. The designer of the final bus should decide what the priorities are, and optimize the protocol and algorithms to meet these needs. For example, adding caches to each module might reduce the number memory requests to make the bus, but cache transactions on the bus take more time because of cache synchronization. This situation needs to be looked at very carefully.
BIBLIOGRAPHY


[8] Conway, L; and Mead, C. Introduction to VLSI Systems, Addison Wesley


I would like to thank Dr. James Davis, Dr. Marwan Hassoun, and Dr. Arthur Pohm for their guidance throughout this project, and for giving me the opportunity for working on it.

I would also like to thank Dr. Scott Irwin and Dr. Sagar Kenkare for their help in understanding the intricacies of the ActiveBus protocol.

Special thanks also go the Control Data Corporation for sponsoring this project.
APPENDIX A: VTI STATE MACHINE DESCRIPTIONS

sm arbitrate;
clock s_clk;
reset !rst --> idle;
inputs busrq bbin flin bbcdin wonin samein akin;
latched outputs bb drivead fl mst;
state idle
  busrq&bbin --> a1 bb=0 drivead=0 fl=0 mst=0,
  busrq&!bbin --> a2 bb=0 drivead=0 fl=0 mst=0,
  !busrq --> idle bb=0 drivead=0 fl=0 mst=0;
state a1
  !busrq --> idle bb=0 drivead=0 fl=0 mst=0,
  !bbin --> a2 bb=0 drivead=0 fl=0 mst=0,
  bbin --> a1 bb=0 drivead=0 fl=0 mst=0;
state a2
  flin --> a1 bb=0 drivead=0 fl=0 mst=0,
  !flin --> a3 bb=1 drivead=1 fl=0 mst=0;
state a3
  !bbcdin --> m0 bb=1 drivead=0 fl=0 mst=1,
  bbcdin --> a4 bb=1 drivead=1 fl=0 mst=0;
state a4
  --> a5 bb=1 drivead=1 fl=0 mst=0;
state a5
  wonin --> m0 bb=1 drivead=0 fl=0 mst=1,
  !wonin&samein --> a6 bb=0 drivead=0 fl=0 mst=0,
  !wonin&!samein --> a1 bb=0 drivead=0 fl=0 mst=0;
state a6
  !akin --> a6 bb=0 drivead=0 fl=0 mst=0,
  bbin&akin --> a7 bb=0 drivead=0 fl=0 mst=0;
state a7
  akin --> a7 bb=0 drivead=0 fl=0 mst=0,
  !akin --> a8 bb=0 drivead=0 fl=1 mst=0;
state a8
  bbin --> a8 bb=0 drivead=0 fl=1 mst=0,
  !bbin --> a9 bb=0 drivead=0 fl=1 mst=0;
state a9
  --> a3 bb=1 drivead=1 fl=1 mst=0;
state m0
  --> idle bb=1 drivead=0 fl=0 mst=0;
end
sm down_in;
clock mod_strobe;
reset !rst --> reset;
#define ctrl[4:0] e_m_clk e_mod e_tag tagin enable
inputs load eod;
latched outputs ctrl[4:0] full;
state reset
   --&gt; reset2  ctrl=10110 full=0;
state reset2
   --&gt; idle    ctrl=10110 full=0;
state idle
   load --&gt; loadcw  ctrl=01111 full=1,
   !load --&gt; idle    ctrl=10010 full=0;
state loadcw
   --&gt; loadaddr ctrl=01111 full=0;
state loadaddr
   eod --&gt; idle    ctrl=01000 full=0,
   !eod --&gt; loaddata ctrl=01100 full=0;
state loaddata
   eod --&gt; idle    ctrl=01010 full=0,
   !eod --&gt; loaddata ctrl=01010 full=0;
end
sm down_out;
clock bus_strobe;
reset !rst --> reset;
define ctrl[10:5] e_m_clk e_bus e_tag enable_data enable_data enable_out_select;
# out_select 1=cwaddr 0=data
inputs busrq b_eod;
latched outputs ctrl[10:5] empty;
state reset
  --&gt; idle ctrl=100101 empty=1;
state idle
  busrq --&gt; loadcw ctrl=010111 empty=0,
  !busrq --&gt; idle ctrl=100101 empty=0;
state loadcw
  b_eod --&gt; idle ctrl=011101 empty=1,
  !b_eod --&gt; loadaddr ctrl=010111 empty=0;
state loadaddr
  b_eod --&gt; idle ctrl=011100 empty=1,
  !b_eod --&gt; loaddata ctrl=011100 empty=0;
state loaddata
  b_eod --&gt; idle ctrl=011100 empty=1,
  !b_eod --&gt; loaddata ctrl=011100 empty=0;
end
sm master;
clock m_clk;
reset !rst --> idle;
inputs mst wr rd eod dleq0 error ok ak;
latched outputs gst passst bb_out ld_e;
state idle
  !mst --> idle gst=0 passst=0 bb_out=0 ld_e=0,
  mst --> control gst=1 passst=0 bb_out=1 ld_e=1;
state control
  --> address gst=1 passst=0 bb_out=1 ld_e=0;
state address
  wr!dleq0&!eod --> write gst=1 passst=0 bb_out=1 ld_e=0,
    dleq0 --> done gst=0 passst=0 bb_out=1 ld_e=0,
    eod --> done gst=1 passst=0 bb_out=1 ld_e=0,
  rd!dleq0&!eod --> read gst=0 passst=1 bb_out=1 ld_e=0;
state write
  !eod&&!error&&!ok --> write gst=1 passst=0 bb_out=1
    ld_e=0,
  eod&&!error&&!ok --> done gst=1 passst=0 bb_out=1
    ld_e=0,
  error|ok --> done gst=0 passst=0 bb_out=1 ld_e=0;
state read
  !eod&&!error&&!ok --> read gst=0 passst=1 bb_out=1
    ld_e=0,
  eod&&!error&&!ok --> done gst=0 passst=1 bb_out=1
    ld_e=0,
  error|ok --> done gst=0 passst=0 bb_out=1 ld_e=0;
state done
  ak --> done gst=0 passst=0 bb_out=1 ld_e=0,
  !ak --> idle gst=0 passst=0 bb_out=0 ld_e=0;
end
sm slave;
clock m_clk;
reset !rst --> idle;
inputs bb cwvalid wr rd eod mai stnow mak terminate;
latched outputs ldcnt passst gst ai ak;
state idle
    !bb --> idle ldcnt=0 passst=0 gst=0 ai=1 ak=0,
    bb --> cw ldcnt=1 passst=1 gst=0 ai=1 ak=0;
state cw
    !cwvalid --> cw ldcnt=1 passst=1 gst=0 ai=1 ak=0,
    cwvalid --> address ldcnt=0 passst=1 gst=0 ai=1 ak=0;
state address
    wr&!terminate --> write ldcnt=0 passst=1 gst=0 ai=0
ak=1,
    rd&!terminate --> read1 ldcnt=0 passst=0 gst=0 ai=0
ak=1,
    terminate --> done ldcnt=0 passst=0 gst=0 ai=1
ak=0;
state write
    !eod&!mai --> write ldcnt=0 passst=1 gst=0 ai=0 ak=1,
    eod&!mai --> finishwr ldcnt=0 passst=1 gst=0 ai=0
ak=1,
    mai --> split1 ldcnt=0 passst=1 gst=0 ai=1 ak=1;
state finishwr
    --> done ldcnt=0 passst=0 gst=0 ai=0 ak=0;
state read1
    !stnow&!mai --> read1 ldcnt=0 passst=0 gst=0 ai=0
ak=1,
    stnow&!mai --> read2 ldcnt=0 passst=0 gst=1 ai=0
ak=1,
    mai --> split1 ldcnt=0 passst=0 gst=1 ai=1 ak=1;
state read2
    !eod --> read2 ldcnt=0 passst=0 gst=1 ai=0 ak=1,
    eod --> finishrd ldcnt=0 passst=0 gst=1 ai=0
ak=1;
state finishrd
    --> done ldcnt=0 passst=0 gst=0 ai=0 ak=0;
state split1
    mak --> split1 ldcnt=0 passst=0 gst=0 ai=1 ak=1,
    !mak --> split2 ldcnt=0 passst=0 gst=0 ai=1 ak=0;
state split2
    bb --> split2 ldcnt=0 passst=0 gst=0 ai=1 ak=0,
    !bb --> done ldcnt=0 passst=0 gst=0 ai=1 ak=0;
state done
    bb --> done ldcnt=0 passst=0 gst=0 ai=1 ak=0,
    !bb --> idle ldcnt=0 passst=0 gst=0 ai=1 ak=0;
end
sm status;
clock m_clk;
reset !rst --> idle;
inputs bb ai ak;
latched outputs ok error latchsw;

state idle
  !bb --> idle ok=0 error=0 latchsw=0,
  bb --> busy ok=0 error=0 latchsw=0;

state busy
  ai!!ak --> busy ok=0 error=0 latchsw=0,
  ak&!ai --> decoded ok=0 error=0 latchsw=0;

state decoded
  ak&ai --> decoded ok=0 error=0 latchsw=0,
  !ak --> done ok=1 error=0 latchsw=0,
  ak&ai --> error1 ok=0 error=1 latchsw=0;

state error1
  ak --> error1 ok=0 error=1 latchsw=0,
  !ak --> error2 ok=0 error=1 latchsw=1;

state error2
  bb --> error2 ok=0 error=1 latchsw=1,
  !bb --> idle ok=0 error=1 latchsw=0;

state done
  bb --> done ok=1 error=0 latchsw=0,
  !bb --> idle ok=1 error=0 latchsw=0;

end
sm up_in;
clock bus_strobe;
reset !rst --> reset;
#define ctrl[4:0] e_m_clk e_mod e_tag tagin enable
inputs load eod;
latched outputs ctrl[4:0] full;
state reset --> reset2 ctrl=10110 full=0;
state reset2 --> idle ctrl=10110 full=0;
state idle
  load --> loadcw ctrl=01111 full=0,
  !load --> idle ctrl=10010 full=0;
state loadcw --> loadaddr ctrl=01111 full=0;
state loadaddr
  eod --> idle ctrl=01000 full=1,
  !eod --> loaddata ctrl=01100 full=0;
state loaddata
  eod --> idle ctrl=01000 full=1,
  !eod --> loaddata ctrl=01010 full=0;
end
sm up_out;
clock mod_strobe;
reset !rst --> reset;
#define ctrl[10:5] e_m_clk e_bus e_tag enable_data enable_out_select;
# out_select 1=cwaddr 0=data
inputs busrq b_eod;
latched outputs ctrl[10:5] empty;
state reset
    --> idle      ctrl=100101 empty=1;
state idle
    busrq  --> loadcw   ctrl=010111 empty=0,
    !busrq --> idle     ctrl=100101 empty=0;
state loadcw
    b_eod  --> idle     ctrl=011101 empty=1,
    !b_eod --> loadaddr ctrl=010111 empty=0;
state loadaddr
    b_eod  --> idle     ctrl=011100 empty=1,
    !b_eod --> loaddata ctrl=011100 empty=0;
state loaddata
    b_eod  --> idle     ctrl=011100 empty=1,
    !b_eod --> loaddata ctrl=011100 empty=0;
end