Hierarchical indexing data structure method for verifying the functionality of the STI-to-PCI bridge chips of the IBM eServer z900

by J. F. Silverio
Y. M. Ng
D. F. Anderson

The IBM eServer z900 has an overall system I/O bandwidth which is three times that of IBM S/390® G5/G6 servers, necessitating the use of Self-Timed-Interface (STI)-to-Peripheral-Component-Interface (PCI) bridge chips to exploit this bandwidth. The chips are used to form a layer between the networking attachments of the z900 and its main storage complex. The layer adapts the high-speed point-to-point packet-oriented STI interface of the z900 to its multi-drop PCI bus structure. This paper describes a method for verifying the functionality of the STI-to-PCI bridge chips by implementing a hierarchical indexing method to support all address dispatching, data management, and data integrity checking. The method is at the core of the random-element-level verification methodology to support all data movement mainline testing of the z900. Monitors, checkers, and drivers were developed and integrated as part of the overall methodology to verify all external interfaces.

Introduction

The IBM eServer z900 can accommodate 256 channels, network adapters or Parallel Sysplex* attachments [1]. It is through the use of a robust I/O subsystem based on new chips exploiting the 24 1GB/s self-timed interface (STI) connections of the z900 that its large 16-way processor

0018-8646/02/$5.00 © 2002 IBM

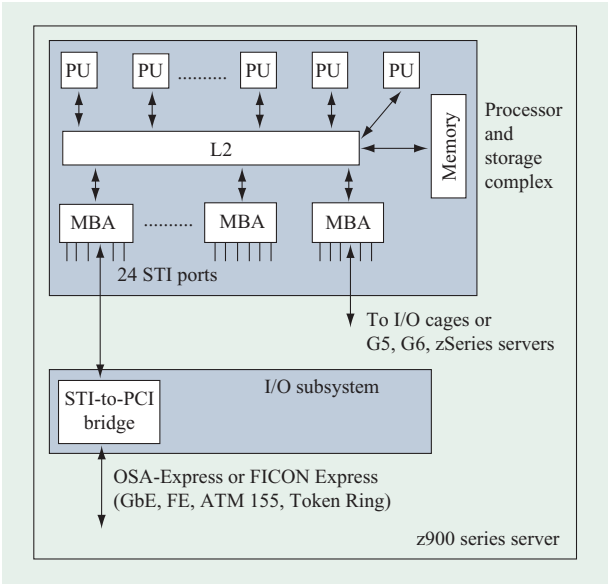

Figure 1

Overall structure of a zSeries server. Its microprocessors (PUs) can be CPs, system assist processors (SAPs), integrated facilities for Linux, or integrated coupling facilities.

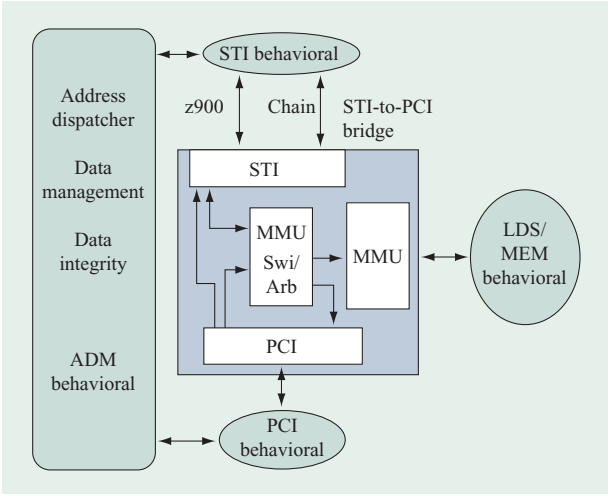

Figure 2

STI-to-PCI bridge chip design verification environment structure.

complex can efficiently run applications. **Figure 1** shows the overall structure of a zSeries server, depicting the processor complex and the STI-to-PCI bridge chip integrated in the I/O subsystem. New complex chips used for mainframe I/O channel, network, and Parallel Sysplex attachments had to have their design and internal function verified. Tight development schedules and cost efficiencies associated with reducing the number of chip design passes, or "RITS," required new methods for verifying interfaces. Mainline function of interfaces between the STI and attachments and internal address and data management functions were verified using innovative methods and tools. Extensive simulation coverage of the new z900 chips enabled the verification team to support the relatively short z900 development schedule to bring its leading-edge performance and function to our customers. Verifying function used in the IBM OSA-Express GbE feature and the high-performance FICON* Express channel was a challenge. These high-bandwidth adapters were respectively available in the z900 in December 2000 and October 2001. Thus, the challenge was met and led to the establishment of a method for verifying fast and complex I/O interfaces and functions.

Element-level verification environment

The environment structure shown in **Figure 2** was implemented for all mainline functional simulation of the STI-to-PCI bridge chips. The structure consists of four behaviorals: three interface behaviorals and an address and data management (ADM) behavioral.

Each of the interface behaviorals [the STI behavioral, the local data storage (LDS) behavioral, and the PCI behavioral] is further subdivided into two other behaviorals: a monitor behavioral, which performs all interface protocol monitoring and validation according to the design specifications, and a driver behavioral, which is responsible for driving the data and address information on the interface buses. Basically the driver behavioral performs two functions: One is to issue commands to the chip, and the other is to respond to commands coming from the chip.

All communication between the interface behaviorals is achieved through the ADM behavioral. The ADM behavioral is an event handler providing all address dispatching and data management functions. First, events occurring at the interfaces trigger the interface behaviorals, and methods are subsequently invoked, thus allowing the behaviorals to communicate with one another. The ADM behavioral is also responsible for handling all data integrity checking. A more detailed description of all environment behaviorals is provided later.

Integrated in the overall environment structure is the device under test (DUT), which in this case is the STI-to-PCI bridge chip.

Device under test

The STI-to-PCI bridge chip, the DUT shown in Figure 2, is a high-performance interconnect chip which serves as a bridge between the eServer zSeries STI bus and the

industry-standard PCI bus. This chip was designed to support an environment in which a processor is attached via the PCI interface. This processor is referred to as the channel processor. Various networking and channel attachments, such as ATM, Fast Ethernet, Gigabit Ethernet, and FICON, are also supported via the PCI interface.

The STI-to-PCI bridge chip contains four basic interfaces, of which two are STI interfaces. One interface provides all data movement to and from the z900 memory, and the other provides the STI chaining capability, making it possible to attach another STI-to-PCI bridge chip in a similar configuration for additional connectivity and bandwidth. An SDRAM memory interface, also known as LDS, provides a temporary memory staging area for data movements in and out of the PCI-attached devices. The LDS is controlled and protected with ECC, single-error correction and double-error detection, by the on-chip memory management unit (MMU). This unit provides a high-speed memory access to the elements inside the chip. The PCI interface is 64-bit, 33-MHz or 66-MHz, which allows the z900 I/O subsystem to support the various PCI adapters.

The data movement between the STI and PCI interfaces is accomplished by a register operation (RegOp) engine and by two data mover queue (DMQ) engines. These DMA engines can be controlled by the channel processor via the PCI interface or by the system assist processor (SAP) [2] via the STI interface.

The MMU switch and arbiter function within the chip provides the internal data path between the data movement functional units.

Verification methodology

Verification of the STI-to-PCI bridge chip was performed with an IBM-designed cycle simulator called ZFS [3]. With a cycle simulator such as ZFS, the detailed timing of the logic circuits is ignored, and the state of the logic is evaluated on clock cycle boundaries. Timing was checked and verified with tools other than ZFS. ZFS provides the performance needed to support the large complex STI-to-PCI bridge chip model. All mainline functional implementation testing was done using ZFS as a single-cycle simulator using the L2 portion of the L1/L2 SRL structure.

All test-case coding was done using SimAPI [4]. SimAPI is a common programming interface for simulation; it provides a set of functions implemented by the user programs and test cases to drive the simulation model.

Because of the structural nature of the behaviorals shown in Figure 2, an object-oriented conceptual language was used. Based on the availability of resources and skills, C++ was the language implemented.

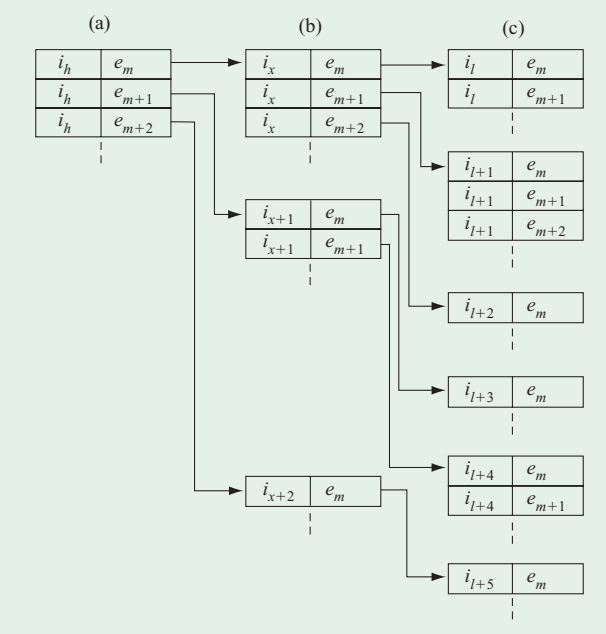

Figure 3

Hierarchical indexing data structure: (a) Higher-level index [i] and entry [e] pairs corresponding to a given operation control block. (b) Intermediate-level index and entry pairs with pointers to lower levels down the hierarchy. (c) Lower-level index and entry pairs. Indices h, x, and l represent any integer value; m is an integer value associated with an entry.

ADM behavioral

Hierarchical indexing data structure

As shown in Figure 2, the address and data management (ADM) behavioral is at the heart of the overall element-level design-verification structure. The ADM behavioral is an event handler acting as an address dispatcher for all chip interface behaviorals. It supports all address and data management functions and, in addition, it handles all data integrity checking. For this purpose, the ADM behavioral implements a hierarchical indexing data structure, as depicted in **Figure 3**.

Each level within the hierarchy associates a given index, which can be any integer value, with a set of entries. Each entry at the highest and intermediate levels contains a reference or pointer to the lower-level indices and so on down the hierarchy. This reference is a key or an address pointer. The entries at the lowest level of the hierarchy contain the addresses and the source or target data allocated for a given operation.

The control-block entries for a given operation are indexed at the highest level of the hierarchy. The control block contains information related to the operation setup

$$K(i_{0-h}, e_{0-m}); \quad \text{(a)}$$

$$K[(i_{0-h}, e_{0-m}); (i_{0-x}, e_{0-m}); \cdots; (i_{0-y}, e_{0-m})] \quad \text{(b)}$$

$$K[(i_{0-h}, e_{0-m}); (i_{0-x}, e_{0-m}); \cdots; (i_{0-y}, e_{0-m}); (i_{0-l}, e_{0-m})] \quad \text{(c)}$$

Figure 4

Key or address pointer index and entry pair representation, where K is the key or address pointer, i is the index, and e is the entry: (a) Highest-level pair; (b) intermediate-level pairs; (c) lowest-level pairs.

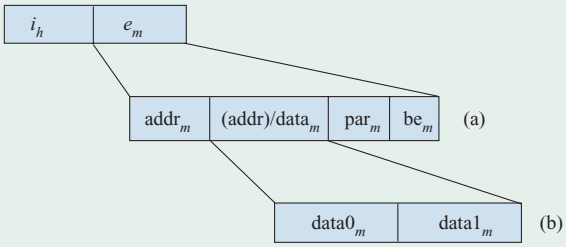

Figure 5

An index and entry pair with the entry portion expanded: (a) Each entry contains an address, and the location in memory to which this address is pointing may contain another address or data. (b) The data portion may contain 32 bits or 64 bits of data.

as well as all its associated controls, such as data movement parameters, source address pointers, target address pointers, data counts, and ending status. At the intermediate levels, the index and entry pairs contain keys or address pointers to lower indices down the hierarchy. At this level, the entry blocks contain keys or address pointers which are used to reference other addresses or memory locations. The indices and entry pairs at the lowest level are associated with actual data blocks in memory.

The keys or address pointers at each level within the hierarchy are represented as shown in **Figure 4**. The highest-level key or address contains a single index and entry pair, as depicted in Figure 4(a). Figure 4(b) shows that intermediate levels contain two or more index and entry pairs associated with a given key or address. A lowest-level key or address which contains the maximum number of index and entry pairs is represented in Figure 4(c).

Figure 5 shows a single index and entry pair in which the entry portion is expanded. The contents of each entry is

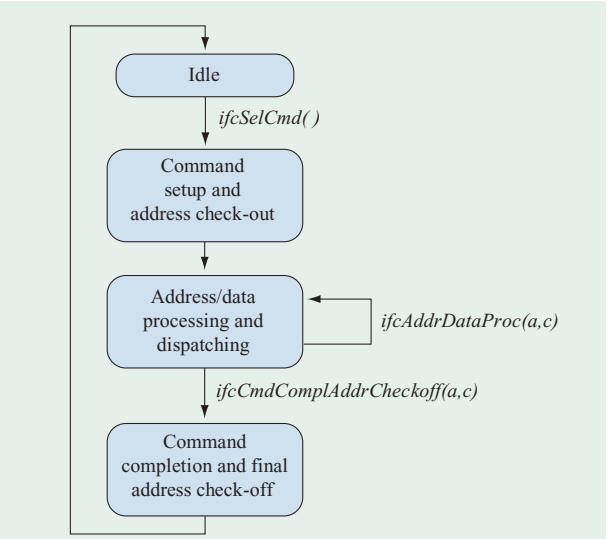

Figure 6

ADM behavioral-state machine structure.

dependent on the level at which the entry is positioned within the hierarchy. At the highest and intermediate levels, the entry contains the address for each entry or memory location. The actual contents of this memory location are an address or a pointer to another address location down the hierarchy. An entry at the lowest level of the hierarchy also contains an address for each entry or memory location, but the memory content to which this address is pointing is the actual data and associated information such as data parity and byte enables. Byte enables define which bytes of data are considered valid. The data parameter for each entry may contain 32 bits or 64 bits of data.

ADM behavioral structure

The structure of the ADM behavioral includes a state machine with four distinct states: an idle state, a command- or operation-setup and address-check-out state, a processing and address-dispatching state, and an address-check-off and operation-completion state. This state machine structure is shown in **Figure 6**. The ADM behavioral makes the transition from an idle state when it is invoked with a command selection function call from an interface behavioral [*ifcSelCmd( )*]. This function call implies that the interface behavioral is ready to process or initiate an operation. After a supported command or operation is selected at random, the ADM behavioral transitions to the command-setup and address-check-out state. In this state, the ADM behavioral will set up the operation and check out all necessary addresses to support

the selected command or operation. When this setup is complete, the transition to the address, data processing, and dispatching state occurs. The ADM behavioral remains in this state while it receives function calls from the interface behaviorals [*ifcAddrDataProc(a,c)*] to dispatch or process an address and its corresponding data. The function is invoked with an address (*a*) parameter and a command (*c*) parameter. If the command is a read from the perspective of the chip, meaning that the chip is performing a read operation, the ADM behavioral dispatches the corresponding data associated with the address parameter. In other words, the data is made available to the interface driver so that it can be driven on the input bus to the chip. If the command is a write, which implies that the chip is performing a write, the ADM behavioral will process the data for the interface monitor or checker behavioral. This is depicted in more detail in the flowcharts shown in Figures 7 through 9. Finally, the ADM behavioral enters the final state when it invokes the functional call to perform the command-completion and final address and data-check-off routine. This routine is called when the ADM behavioral receives an *ifcAddrDataProc(a,c)* call in which, for this case, address (*a*) corresponds to ending status data. This triggers the ADM behavioral to invoke the *ifcCmdComplAddrCheckoff(a,c)* function passing the ending-status address and the write-command parameters.

The flowchart diagram shown in **Figure 7** provides a more detailed description of the command-setup and address-check-out state implemented in the ADM behavioral. The PCI and STI interface driver behaviorals can invoke function *ifcSelCmd( )* simultaneously, forcing the ADM behavioral to transition from an idle state to the command-setup and address-check-out state. In this state, the ADM behavioral selects a command or operation at random from a list of supported commands. The commands or operations supported by the STI-to-PCI bridge chip are described later. After the command selection, the setup begins for all addresses and their corresponding index and entry pairs. These addresses are selected at random from a predefined range allocated during chip configuration time. The address space allocated to the STI behavioral is considered main storage space, while the PCI behavioral will have its space allocated in PCI-device memory-mapped addressable range.

First, the highest level of addresses and corresponding index and entry pairs, represented as $K(i_h, e_m)$;, is set up for the command control block, which includes the ending response or status entry. This block contains address pointers to destination and source address lists of data blocks, as well as other miscellaneous control information necessary to execute the operation. Index h can be any integer value, meaning that several operations can be set

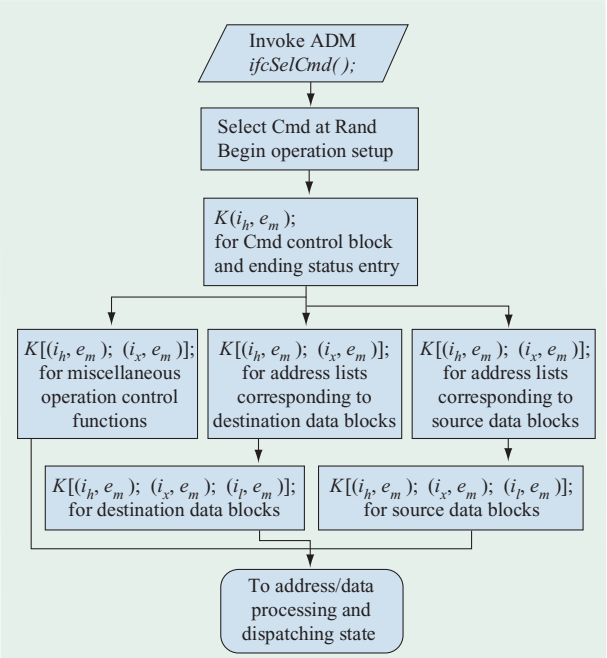

Figure 7

Flowchart diagram for the command- or operation-setup and address-check-out state.

up at any time. Subsequently, the intermediate index and address entry pairs are also set up. As shown in the flowchart diagram, these are represented as $K[(i_h, e_m); (i_x, e_m); \cdots; (i_y, e_m)]$;. The destination and source data blocks can be partitioned at random into smaller blocks of data. Therefore, the lists of address pointers to the smaller data blocks are grouped and set up with intermediate indices and address entry pairs. The value of index x can be any integer value, indicating that there could be several indices and entry pairs associated with a higher-level index h. Also, other miscellaneous control functions are represented at the same intermediate level, as are the lists of address pointers. For instance, LRC and CRC starting and ending pointers are set up and represented at this level of the hierarchy. At the lowest level of the index and entry pair hierarchy, all data blocks for the destination are set up, as well as the source data blocks. The index and address entry pair representation for these data blocks is $K[(i_h, e_m); (i_x, e_m); \cdots; (i_y, e_m); (i_l, e_m)]$;. An address at this level is represented by the maximum number of indices and entry pairs. The lowest-level index l can also be any integer value, and several indices at this level are associated with a single index x at the previous level within the hierarchy. Since any given index may contain several entries, m can be any integer value as well. Therefore, any index and entry pair at a

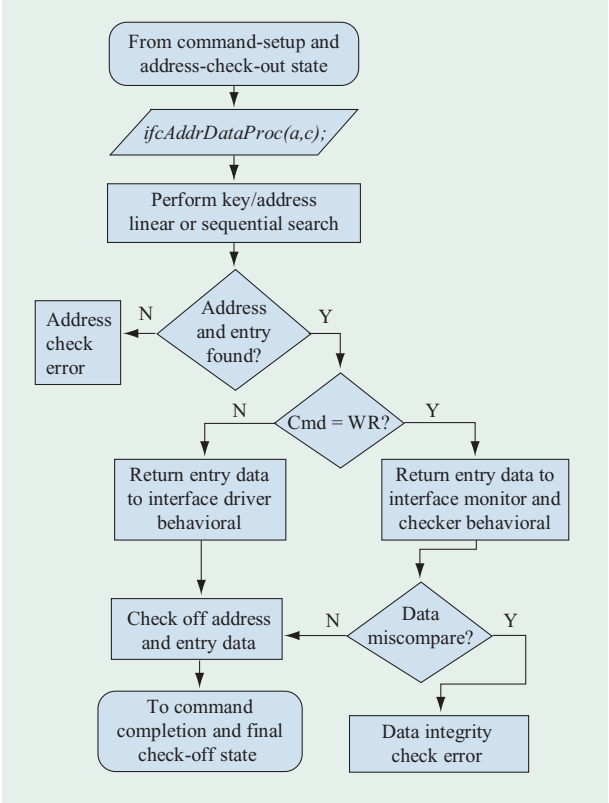

Figure 8

Flowchart diagram for the address/data processing and dispatching state.

lower level within the hierarchy can be associated with only a single index at a higher level, and so on, as the hierarchy structure is traversed from the lowest level to the highest.

The ADM behavioral transitions unconditionally to the next state when all necessary address index and entry structures associated with and necessary to support the operation are properly set up. This may take place during the next machine cycle or several machine cycles later, depending on when the event occurs at the interfaces. Then this event or command triggers the interface behaviorals to call the ADM behavioral. The next state to which the ADM behavioral transitions is described in the flowchart diagram of **Figure 8**. While in this state (the address, data processing and dispatching state), the ADM behavioral waits for the interface behaviorals to invoke it by the interface monitors and checkers or by the interface drivers. This is done via the *ifcAddrDataProc(a,c)* function call. The parameters being passed are simply an address (*a*) and a command type (*c*). A read command parameter is passed by the interface driver, since from the perspective of the chip, a read command means that the dataflow is from the interface driver behavioral to the chip inputs. On the other hand, a write command parameter is passed by the interface monitor, since from the point of view of the chip, the data movement is from the chip outputs to the interface monitor behavioral. Whether it is the interface driver or the interface monitor invoking *ifcAddrDataProc(a,c)*, the ADM behavioral will perform a linear or sequential search of the address and entry information [5]. This simple and straightforward data search approach is implemented to facilitate the communication between the behaviorals within the overall structure. The interface behaviorals have no knowledge of any of the chip operations or command information and controls associated with a given address. The only information the interface behaviorals have associated with the address is whether that address location is being read or written. The burden of checking and testing is shifted to the ADM behavioral, which has the responsibility for checking and testing all conditions and information related to the address being passed by the interface behaviorals.

If the address being searched, with its associated data information, has not been found after the linear or sequential search has been performed, an error check is flagged. In the event that the address and data information is found, all indices and entry pairs associated with this address are composed, and the data information is returned to the interface driver in the case of a read, or to the interface monitor in the case of a write. Subsequently, this address and its associated indices and entry pairs are checked off. Therefore, this address has been visited, and its associated data and control information has been properly checked by the interface monitor behaviorals in the case of a write-type operation, or it has been visited and its data provided to the interface behavioral drivers in the case of a read operation. Also, in the case of a write command, a data integrity check is performed at this point. The entry information returned to the interface monitor is compared with the outgoing data. If a miscompare is present, an error check is flagged, as shown in Figure 8.

The transition to the command-completion and final check-off state, for any given operation, may occur at any point in time if the interface behavioral invokes the *ifcAddrDataProc(a,c)* function for one final time. The ADM behavioral performs a linear or sequential search, and if an ending status write is found, the ADM behavioral calls the *ifcCmdComplAddrCheckoff(a,c)* function. What constitutes an ending status write is determined by the fact that it is a write operation to an address entry with the highest-level index and entry pair associated with it. The flowchart diagram representing this state is shown in **Figure 9**. When this function is invoked, it is an indication that the hardware model is completing the operation by updating or writing to the response or ending status

address location. Once again the ADM behavioral performs a linear or sequential search of this ending status address. If it is not found, an error check is displayed; if it is found, the index and entry pair for this address is composed, and it is checked off. Subsequently, the ADM behavioral performs a hierarchical indexing data search of all addresses and their associated indices and entry pairs from the highest level all the way down to the lowest level. This search begins with the index and entry pair, $K(i_h, e_m)$;, for the ending status address entry. Then the search continues with all addresses for all the intermediate-level and entry pairs, $K[(i_h, e_m); (i_x, e_m); \cdots; (i_y, e_m)]$;, associated with this higher-level index. The search then moves on to the lowest-level addresses, represented as $K[(i_h, e_m); (i_x, e_m); \cdots; (i_y, e_m); (i_l, e_m)]$;. All addresses at this level, which are associated with a given index at the previous level, are also visited. These addresses refer to memory locations with actual data. If any of the addresses or associated indices and entries are not checked off, this operation is deemed unsuccessful, and the addresses that were not visited will be displayed for further debugging. This hierarchical search from the highest level through the intermediate levels and on down to the lowest level is critical to the overall mainline dataflow testing and debugging, since many operations may be outstanding at any particular time.

Supported commands or operations

The set of supported commands can be subdivided into two subsets: commands from the memory bus adapter (MBA) targeting the CCA registers internal to the chip or to access other chained STI-to-PCI chips, or commands targeted to the MBA. The commands from the MBA targeting the CCA registers are initiated from the SAP or PUs in the main storage complex. These can be referred to as outgoing commands. The commands targeted to the MBA are initiated from the STI-to-PCI bridge chip itself or from a processor attached via the PCI interface. These commands are considered to be incoming commands. The outgoing commands are initiated from the STI behavioral driver, and the incoming commands are emulated by the PCI interface driver behavioral.

The outgoing commands from the main storage complex via the MBA are called disconnected senses and controls (D-S/C). These are implemented to support reads and writes of internal CCA registers, logging, and maintenance registers. In addition, they are used to perform similar functions on a chip attached via the chained STI interface. Disconnected senses are considered to be read commands, while disconnected controls are thought of as write commands.

The set of incoming commands are further subdivided into two groups: storage commands and sense/control commands. Storage commands can be initiated from the SAP or from a PCI-attached processor device. A fetch storage command involves the transfer of data from the main storage complex area to the peripheral device attached to the PCI interface. For a store storage-type command, the flow of data is to the main storage complex. In the context of the STI-to-PCI bridge chip, the operations which involve the movement of data are controlled by an internal data mover queue (DMQ) engine or by an internal register operation (RegOp) engine. The hardware structure for the STI-to-PCI bridge chip integrates two DMQ engines with the capability to execute two queue entries simultaneously. It also contains one register-operation engine. Register operations are serialized, but they can be executed simultaneously with DMQ operations. As part of an IP packet, the amount of data movement supported, in terms of bytes, is a variable number between 1 and 64. In addition, a maximum of 128 bytes is also supported. Internally, the STI-to-PCI bridge chip can initiate sense- and control-type commands as well. Control commands are used to modify bit vectors in the MBA. Sense commands are implemented to communicate channel subsystem timer values to the MBA.

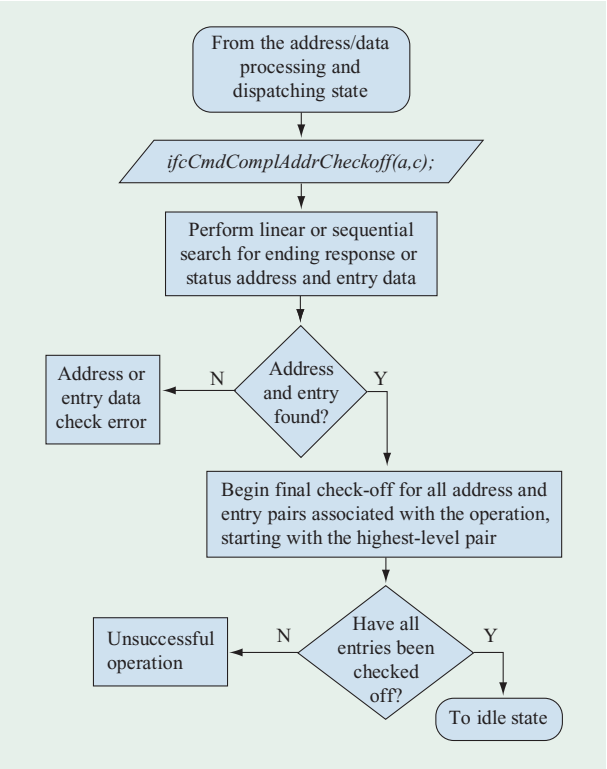

Figure 9

Flowchart diagram for the command-completion and final address-check-off state.

Table 1 Format for DMQ entries.

Hardware response word	— Valid bit and response code
DMQ control word	— DMQ data movement operational parameters
PCI space address lists	— Pointer to top of PCI address list
z900 space address lists	— Pointer to top of z900 address list
Address list sizes	— Valid PCI/z900 space address/count pairs
External address/zone ID	— Used to support 35-, 36-, or 48-bit addressing
CRC start pointer	— Address pointer to initial CRC seed
CRC end pointer	— Address pointer to ending CRC
LRC start pointer	— Address pointer to initial LRC seed
LRC end pointer	— Address pointer to ending LRC
Duplicate z900 address pointer	— Bit-inverted copies of z900 addresses
Pad bytes	— Doubleword multiples

Sense and control commands involve small movements of data, normally 32 bits or 64 bits. Therefore, the address being accessed by these sense and control commands can be represented with a single index and entry pair. Following the nomenclature to represent addresses and their associated indices and entry pairs presented previously, senses and controls can be represented as $K(i_h, e_m)$;. The representation used for sense- and control-type commands is a subset of the hierarchical indexing data structure presented before. On the other hand, storage commands, which may involve large movements of data, require several levels of indices and entry pairs in order to be represented. For DMQ- and RegOp-type operations, the hierarchical indexing data structure implemented in the ADM behavioral is necessary to set up, process, and dispatch all addresses and data to support the operation. The following two sections provide a more detailed description of DMQ and RegOp operations and how the hierarchical indexing data structure is implemented to support the testing and debugging of these operations.

Data mover queue (DMQ) operations

The DMQ operation supports the transfer of large blocks of data between main storage (z900) and a PCI-address-mapped space, LPS or LDS. The generation of the command streams to perform the data movement is defined by the contents of the DMQ entry. This DMQ entry can be initiated or set up by a PCI-attached channel processor or by a z900 processor configured as a SAP. A series of DMQ entry queues are stored contiguously in a predefined portion of LDS or LPS space. Each DMQ entry represents multiple movements of blocks of data. The total length of a DMQ entry is configured by the channel microcode at initialization time. A minimum of twelve 32-bit words are required per DMQ entry. **Table 1** shows the format of each DMQ entry. The configuration time and initialization time mentioned here are emulated by setting specific hardware facilities in the model or by explicitly performing control commands targeted to the chip configuration registers. A detailed explanation of the initialization and configuration sequence implemented for the STI-to-PCI bridge chip and the bit definitions for the contents of the configuration registers is beyond the scope of this paper.

The hardware response word can be read or written by the internal DMQ engine. When it is read, a valid bit indicates whether or not the entry can be executed by the DMQ engine. The time at which this entry is written by the DMQ engine is an indication that the engine is completing the operation. The contents of this address entry, referred to as the response code, indicate whether the operation ended successfully or with a particular error. The DMQ control word is used to control the operation. The direction of the storage command, the LRC and CRC controls, operation byte count, and several other controls are defined in the DMQ control word. The address pointers to the list of partitioned address and data blocks, in both LPS and main-storage space, are respectively defined by the PCI address list pointer word and by the z900 address list pointer. The actual sizes of these lists are defined in the fifth word of the DMQ entry. The extended address and zone ID word are used to support 35-, 36-, or 48-bit addressing modes of operation. The following words contain address pointers for initial and final or generated CRC and LRC values. The following word is an address pointer to the top of the duplicate of the z900 address list. This block contains a bit-inverted copy of the addresses from the normal list. As the data movement to and from main storage takes place, the normal address list contents

are compared with this bit-inverted list. A miscompare results in an error condition which causes the DMQ to terminate. The last of the required words is used only for padding purposes.

Implementing the hierarchical indexing structure discussed before, the address for each one of these words in the DMQ entry is referenced by a higher-level index. Once again, following the nomenclature presented before, each address, starting with the first word, is represented as $K(i_h, e_0)$;, followed by $K(i_h, e_1)$; for the second word address, and so on down to $K(i_h, e_{11})$; for the last required word address. K corresponds to the address for each one of the words in the DMQ entry; (h) is a single integer value to represent this particular DMQ queue entry. Each entry (e) contains the addresses and the corresponding data entries. In this case, the data entries are actual address pointers to other locations in memory where other address list pointers (the z900 and the LPS address list pointers) reside. These blocks containing the address list pointers are represented at the intermediate levels of the hierarchy as $K[(i_h, e_2); (i_x, e_m)]$; for the block of LPS addresses and as $K[(i_h, e_3); (i_x, e_m)]$; for the block of z900 addresses, where (x) is a single unique integer value referencing the block of address pointers to the actual data words or doublewords in memory. For instance, an address pointing to the top or first entry of a list of LPS addresses is represented as $K[(i_h, e_2); (i_x, e_0)]$;. An address pointing to the actual data entry block corresponding to this index and entry pair representation can in turn be represented as $K[(i_h, e_2); (i_x, e_0); (i_l, e_m)]$;, where (l) is a unique integer value referencing this particular block of data. For example, the address of the data in the fifth entry of this data block is represented as $K[(i_h, e_2); (i_x, e_0); (i_l, e_4)]$;. An address pointing to the fourth entry of a list of z900 addresses can be represented as $K[(i_h, e_3); (i_x, e_3)]$;. Subsequently, for instance, the address of the data in the sixth entry of this data block is represented as $K[(i_h, e_3); (i_x, e_3); (i_l, e_5)]$;. The addresses for the entries pointed to by the rest of word entries in the DMQ queue entry block can be represented by two levels of index and entry pairs. This representation has previously been shown as $K[(i_h, e_m); (i_x, e_m)]$;. For example, the addresses of the actual CRC start value, CRC ending value, LRC start value, and LRC ending value would be represented respectively as $K[(i_h, e_6); (i_x, e_0)]$;, $K[(i_h, e_7); (i_x, e_0)]$;, $K[(i_h, e_8); (i_x, e_0)]$;, and $K[(i_h, e_9); (i_x, e_0)]$;.

When each address is visited, as a result of the linear or sequential search when the interface behaviorals invoke the *ifcAddrDataProc(a,c)* function call, a check-off parameter is tagged with this address, indicating that there is a match for the associated data and that the address was found.

When the ending status or response word is written, indicating the end of the operation, the ADM behavioral is triggered to perform the final completion check-off. All entries associated with highest-level index are visited to make sure that the check-off parameter has been set for all of them. Then all indexed entries referenced by the address pointers in the DMQ entry are also visited. These are the intermediate indexed entries within the hierarchy. Finally, all indexed entries at the lowest level of the hierarchy which are associated with the previous levels are also visited. Hence, the ADM behavioral performs the checking by traversing the hierarchical indexing structure from the highest level to the lower levels.

Register operations

Register operations are used to pass control information between the channel processor and the z900 storage complex. In addition, these operations are used to generate single storage commands. A register operation can be initiated from the channel processor via the PCI interface or from the SAP via the STI interface with a D-S/C sense command.

Register operations are set up by microcode in PCI-defined space as a control block. This control-block format, shown in **Table 2**, defines all data movement as well as all control parameters. The register operation is started with a write operation to the blocked register operation pointer. The nomenclature shown above to represent addresses and their associated index and entry pairs is also used for the RegOp commands. Therefore, the address containing the blocked RegOp address pointer is represented as $K(i_h, e_m)$;. This is the highest-level index and entry pair representation. The content of this blocked-register operation pointer points to an address at the top of the location where the actual register operation block resides in memory. A block of data is fetched starting at the address location, and the register operation begins. The format for this block of data is shown in Table 2.

The addresses and associated index and entry pairs are considered at the intermediate level and are represented as $K[(i_h, e_m); (i_x, e_m)]$;. The first four words within this block of data contain the header and extended data header information as required by STI IP packet transfer protocol [4]. The extended data header word 1 contains an address pointer to the main-storage location for the destination of the data. The following word is an address pointer to the PCI address space where the source of the data is located. All addresses associated with these entries are at the lowest level of the indexing structure hierarchy and can be represented as $K[(i_h, e_m); (i_x, e_m); (i_l, e_m)]$;. The addresses for the rest of the words in the RegOp block are represented with the intermediate-level index and entry pair. At the completion of this operation, the STI status register word is written with the proper status information, such as IP control or interface error information, or simply the information received in the response IP packet. Just as with DMQ-type operations,

Table 2 Format for RegOp entries.

Header word 0	— STI IP header word 0
Header word 1	— STI IP header word 1
Extended header word 0	— STI IP extended data header word 0
Extended header word 1	— STI IP extended data header word 1
PCI space address	— Pointer to PCI space address block of data
Register op control word	— Control word to support RegOp execution
Data register word 0	— Data word 0; used as source/destination data transfer
Data register word 1	— Data word 1; used as source/destination data transfer
LRC start pointer	— Address pointer to initial LRC value/seed
LRC end pointer	— Address pointer to final LRC value
CRC start pointer	— Address pointer to initial CRC value/seed
CRC end pointer	— Address pointer to final CRC value
STI status register word	— Ending status word

the final check-off and operation-completion state in the ADM behavioral structure performs a hierarchical indexing data structure search starting at the highest level of the hierarchy to make sure that all addresses related to this RegOp command are properly checked off and their associated data processed correctly. The ADM behavioral proceeds with the intermediate levels all the way down to the lowest level.

STI behavioral

The STI-to-PCI bridge chip implements two STI interfaces; one provides the connection with the MBA and the other allows the connection to other chained STI-to-PCI bridge chips. The STI behavioral emulates both the MBA interface and any other attachments to the chained STI interface. In this paper, the focus is on the dataflow of IP packets. The following section discusses the STI behavioral structure in the context of movement of IP packets for mainline testing and debugging purposes. These IP packets can be in the form of commands, also known as outgoing packets, or responses, which are considered to be incoming packets.

The STI interface is a full-duplex link structure which comprises two 10-wire connection links. One link is used for receiving IP packets and the other is used for sending IP packets. Each 10-wire connection contains an 8-bit serial data interface, one wire is used as a parity or control bit, and the tenth wire is implemented as the clock which travels with the data. Since the clock travels with the data, the physical receive layer of the STI link handles skew and jitter in the transmission link by retiming each data bit with the transmitted clock. Thus, this interface is known as a self-timed interface [6].

The verification of the timing constraints, clock jitter, and skew on the STI link are not addressed in this random single-cycle simulation environment. This environment focuses on functional testing, and it does not handle timing issues. Therefore, the STI interface is modeled such that each bit of data is clocked with a single pulse. This is true for all eight data wires. Thus, after eight clock pulses, a byte of data would be transmitted over each wire. With eight wires, an IP packet containing a minimum of 64 bits would be transmitted after eight clock pulses. Each z900 main-storage 32-bit address points to a memory location with 64 bits of data.

STI behavioral structure for dataflow testing

The STI behavioral interfaces with the STI via the 8-bit serial interface, and it communicates with the ADM behavioral using a 32-bit z900 main-storage address. In addition, the other parameter used to communicate with the ADM behavioral is the command type, whether the command is a read from main memory or a write to main memory.

The structure of the STI behavioral includes a monitor behavioral and a driver behavioral. The STI monitor behavioral also includes several checking routines to handle all data-integrity checking as well as controls and protocol checking. When the STI behavioral receives an IP packet for a write request to main memory (in other words, an incoming packet where the data is flowing out of the chip and into the STI behavioral), it will invoke the ADM behavioral with function call *ifcAddrDataProc(a,c)*, where (*a*) is the 32-bit z900 main-storage address and (*c*) indicates that the command is a write operation. This is handled by the receive state machine internal to the STI

monitor behavioral. The ADM behavioral returns the data associated with this address, and the STI monitor invokes a checker routine to perform the data-integrity checking. If a mismatch occurs, this routine will flag a check error.

The driver behavioral integrated in the STI interface behavioral communicates with the ADM behavioral via two functional calls: the *ifcSelCmd( )* call, which is used when the STI driver wants to initiate a command, and the *ifcAddrDataProc(a,c)* function call, which is invoked when the STI driver receives a read request operation. When *ifcSelCmd( )* is called, a SAP or MBA command is selected at random from a list of supported commands. As discussed before, the ADM behavioral sets up all address indices and entry pairs for the selected operation. The state machine in the STI driver behavioral responsible for handling SAP- or MBA-initiated commands is called the initiate state machine. This state machine also supports the testing of commands initiated via the chained STI interface.

The function *ifcAddrDataProc(a,c)* is invoked when the STI driver receives a read request operation—for instance, a fetch storage command. The ADM behavioral returns the 64-bit data associated with the z900 32-bit address, and the STI driver places it on the STI interface transmit side.

PCI behavioral

PCI is a multi-drop bus which supports multiple devices on the bus through a bus arbiter. The PCI interface behavioral emulates the activities of the PCI devices on the bus. Each PCI device on the interface is represented by a class instance of the PCI interface behavioral. The bridge chip internal PCI arbiter can support four PCI devices on the interface. One of the PCI devices is connected to a microprocessor which handles the PCI configuration as well as setting up the controls for the DMA engines. The PCI behavioral can be operated at a clock rate of 33 MHz or 66 MHz. Each class instance can be programmed independently to support 32-bit or 64-bit transactions. The PCI interface behavioral has three basic operation states: the idle state, the command/address state, and the data state used for both the monitor and driver function of the behavioral. Protocols, data, parity, and PCI arbitration are checked by the monitor function within each operating state. The monitor also helps to test the tri-state bus requirements in a two-value (0 and 1) simulator environment by monitoring the output-enable signal of all of the bidirectional drivers. The driver function of the PCI interface behavioral is implemented with two independent state machines to handle master and target operations. The driver stress-tests the interface by randomly driving all of the control signals, including the bus requests, as long as they are permitted by the specification. In the case of a master operation, during the command/address state, the PCI interface behavioral calls the ADM behavioral with *ifcSelCmd( )* to get a command and address to initiate a transaction on the bus. The transaction can be for any PCI device, including the bridge chip on the bus. The data obtained from the ADM behavioral for a write operation is sent to the bus, while the data for a read operation is used to compare with the received data from the bus. When responding to a target operation during the data state, the PCI interface behavioral calls the ADM behavioral with *ifcAddrDataProc(a,c)* to obtain information to validate the receiving data for a write operation or return data to the requester for a read operation. With the help of the ADM behavioral, the PCI interface behavioral need only process and check a piece of data at a time, leaving the more complex job of verifying the full transaction completion to the ADM behavioral.

LDS behavioral

The LDS interface behavioral can be programmed to represent a variety of SDRAMs with different latency requirements and sizes supported by the MMU. It can be configured to contain up to four external banks of memory on the bus. The LDS interface behavioral does not contain any real storage area. It depends on the ADM behavioral to keep track of the data being moved into and out of the SDRAM. The structure of the LDS interface behavioral is similar to that of the PCI interface behavioral, with both monitor and driver function. Each bank of memory is represented by a class instance of the LDS interface behavioral. The LDS interface behavioral monitors the interface, checks protocols, and provides responses to the bus according to the SDRAM specification. It checks the power-on and wake-up sequences required by the SDRAM. It also verifies the refresh, scrubbing, and ECC function of the MMU with the information collected from the memory interface. The LDS behavioral has only the target states to handle requests from the memory bus. It is not required to perform any master operation such as initiating commands. Upon receiving a data-moving command from the memory bus, the LDS behavioral uses *ifcAddrDataProc(a,c)* to call the ADM behavioral to obtain data to prepare for the bus operation. The LDS interface behavioral utilizes an ECC function to generate the check bits for the data. In a memory read operation, data from the ADM behavioral along with ECC check bits are sent to the bus. In a memory write operation, data from the ADM behavioral together with ECC check bits are used to verify the received data from the bus. With just a simple communication path to the ADM behavioral, the task of handling the data for memory operations becomes much simpler for the LDS interface behavioral.

Concluding remarks

The verification method described here, which utilized three interface behaviorals and an address and data management behavioral, was first developed to support the verification of the STI-to-PCI bridge chip integrated in the I/O subsystem of the Multiprise* 3000 and G5/G6 S/390 series servers. During the verification-environment planning stages of the STI-to-PCI bridge chip for the I/O subsystem of the eServer z900, it became more cost-effective, from a resources and skills point of view, to reuse an existing environment structure with some modifications to support new or added functional logic requirements. As a result, the use of the environment structure described in this paper contributed to a two-pass (RIT) design and on-time delivery of the most complex I/O chip integrated in the eServer I/O subsystem to date. In addition, this environment structure, with its verification implementation and tools methodology, should be useful for future derivative chips.

Industry-standard buses are used to provide the high-performance "line speed" attachments for mainframe channels (FICON Express) and network adapters (OSA-Express GbE) to the STI buses. Chip-verification methods such as the hierarchical indexing of data structures have made it possible to achieve excellent simulation coverage and team success for state-of-the-art designs implemented in leading-edge technologies. The methods, successfully used on the z900 chips used in the IBM OSA-Express GbE feature and the associated FICON Express channel should be useful for other eServers as they evolve and require more I/O bandwidth support for their powerful microprocessor complexes.

Acknowledgments

The authors would like to thank Bruce Wile and David Fox for their valuable comments regarding this manuscript.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. C. L. Rao, G. M. King, and B. A. Weiler, "Integrated Cluster Bus Performance for the IBM S/390 Parallel Sysplex," *IBM J. Res. & Dev.* **43,** No. 5/6, 855–862 (1999).
2. T. A. Gregg, "S/390 CMOS Server I/O: The Continuing Evolution," *IBM J. Res. & Dev.* **41,** No. 4/5, 449–462 (1997).
3. B. Wile, M. P. Mullen, C. Hanson, D. G. Bair, K. M. Lasko, P. J. Duffy, E. J. Kaminski, Jr., T. E. Gilbert, S. M. Licker, R. G. Sheldon, W. D. Wollyung, W. J. Lewis, and R. J. Adkins, "Functional Verification of the CMOS S/390 Parallel Enterprise G4 System," *IBM J. Res. & Dev.* **41,** No. 4/5, 549–566 (1997).
4. G. G. Hallock, E. J. Kaminski, K. M. Lasko, and M. P. Mullen, "SimAPI—A Common Programming Interface for Simulation," *IBM J. Res. & Dev.* **41,** No. 4/5, 601–610 (1997).
5. Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1997.
6. J. M. Hoke, P. W. Bond, T. Lo, F. S. Pidala, and G. Steinbrueck, "Self-Timed Interface for S/390 I/O Subsystem Interconnection," *IBM J. Res. & Dev.* **43,** No. 5/6, 829–846 (1999).

Received October 17, 2001; accepted for publication February 12, 2002

Jose F. Silverio *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (silverio@us.ibm.com).* Mr. Silverio received a B.S. degree in electrical engineering from the University of Hartford, Connecticut, in 1990. That same year he joined IBM as a logical designer on an advanced I/O processor project. He received an M.S. degree in computer engineering from Syracuse University in 1995. In 1996, he received an IBM Team Award for his work on the IBM S/390 G3 fast internal bus design. Mr. Silverio joined the Advanced I/O Connectivity Hardware Verification team in 1996; he is currently an Advisory Engineer. He has received IBM Outstanding Technical Achievement Awards for his design verification work on the STI-to-PCI bridge chip for the S/390 G4 (1999), Multiprise 3000 (2000), and z900 eServer (2001).

Y. Ming Ng *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (mingng@us.ibm.com).* Mr. Ng is an Advisory Engineer in the eServer Connectivity Solutions Development Group. He received B.S. and M.S. degrees in electrical engineering from the New Jersey Institute of Technology in 1976 and 1978, respectively. In 1978 he joined IBM in Endicott, New York, where he worked as a logic designer in processor development. In 1992, he moved to Poughkeepsie, New York, to work on hardware simulation in I/O development. He has received an IBM Invention Achievement Award and several IBM Outstanding Technical Achievement Awards.

David F. Anderson *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (dfa@us.ibm.com).* Mr. Anderson received a B.S. degree in engineering from the United States Military Academy, West Point, New York, in 1976. He also received an M.B.A. degree from the University of Puget Sound in 1979 and an M.S. degree in engineering science from the Rensselaer Polytechnic Institute in 1993. Mr. Anderson joined IBM in 1981 to work on resource planning and has worked on almost all of the large systems which have been designed and manufactured in Poughkeepsie in the past 20 years, including the IBM 3033, 3081, 3090 E/S/J, ES/9000, 9121, 9672 CMOS Parallel Enterprise Servers through Generation 6, and, recently, the zSeries. In July 1999, he took an assignment managing a department responsible for the logic design and simulation of advanced I/O hardware (chips). The department designed, developed and debugged channels (ESCON, FICON and FICON Express) and OSA-Express network adapters (such as GbE) in the IBM z900. Mr. Anderson received an IBM Excellence Award for his work on the zSeries I/O chip development. He is currently a Senior Engineer educating and consulting with customers approximately 80% of his time in the Poughkeepsie eServer Briefing Center. He spends the remainder of his time working with eServer teams across IBM on future products and announcements.