5.1.3 Register Usage

There will be 512 registers available. The first 256 will be used for process contexts. The remaining 256 will be split between the 3 sequencers as follows:
257 – 320: 64 for RCV general processing / main loop.
321 - 384: 64 for XMT general processing / main loop.
385 – 512: 128 for 3rd sequencer use.

5.2 Receive Processing

5.2.1 Main Loop

The following is a summary of the main loop of Receive:

```
forever {
        while there are any Receive events {
                if (a new event) {
                        if (no new context available)
                                ignore the event;
                }
                call appropriate event handler to service the event;
                this may make a waiting process runnable or set up
                a new process to be run (get free context, hddr buffer,
                TCB buffer, set the context up).
        }
        while any process contexts are runable {
                run them by jumping to the start/resume address;
                if (process complete)
                        free the context;
        }
}
```

5.2.2 Receive Events

The events that will be processed on a given context are:

- accept a context
- release a context command (from the host via Transmit)
- release a context request (from Transmit)
- receive a valid frame; this will actually become 2 events based on the received frame
- receive an ACK, receive a segment
- receive an "invalid" frame i.e. one that causes the TCB to be flushed to the host
- a valid ACK needs to be sent (delayed ACK timer expiry).
- There are expected to be the following sources of events:
  1. Receive input queue: it is expected that hardware will automatically DMA arriving frames into frame buffers and queue an event into a RCV-event queue.
  2. Timer event queue: expiration of a timer will queue an event into this queue.
  3. Transmit sequencer queue: for requests from the transmit processor.

09/748,936 (approx. stamp "60051809.101497")
WISTRON CORP. EXHIBIT 1031.056

For the sake of brevity the following only discusses receive-frame processing .

5.2.3 Receive Details – Valid Context

The base for the receive processing done by the INIC on an existing context is the fast-path or "header prediction" code in the FreeBSD release. Thus the processing is divided into 3 parts: header validation and checksumming, TCP processing and subsequent SMB processing.

5.2.3.1 Header Validation

There is considerable hardware assist here. The first step in receive processing is to dma the frame header into an SRAM header buffer. It is useful for header validation to be implemented in conjunction with this dma by scanning the data as it flies by. The following tests need to be "passed":

- MAC header: destination address is our MAC address (not MC or BC too), the Ethertype is IP.
- IP header: header checksum is valid, header length = 5, IP length > header length, protocol = TCP, no fragmentation, destination IP is our IP address.
- TCP header: checksum is valid (incl. pseudo-header), header length = 5 or 8 (timestamp option), length is valid, dest port = SMB or FTP data, no FIN/SYN/URG/PSH/RST bits set, timestamp option is valid if present, segment is in sequence, the window size did not change, this is not a retransmission, it is a pure ACK or a pure receive segment, and most important, a valid context exists. The valid-context test is non-trivial in the amount of work involved to determine it. Also note that for pure ACKs, the window-size test will be relaxed. This is because initially the output PERSIST state is to be handled on the INIC.

Many but perhaps not all of these tests will be performed in hardware – depending upon the embodiment

5.2.3.2 TCP Processing

Once a frame has passed the header validation tests, processing splits based on whether the frame is a pure ACK or a pure received segment.

5.2.3.2.1 Pure RCV Packet

The design is to split off headers into a small header buffer and pass the aligned data in separate large buffers. Since a frame has been received, eventually some receiver process on the host will need to be informed. In the case of FTP, the frame is pure data and it is passed to the host immediately. This involves getting large buffers and dmaing the data into them, then setting the appropriate details in a small buffer that is used to notify the host. However for SMB, the INIC is performing reassembly of data when the frame consists of headers and data. So there may not yet be a complete SMB to pass to the host. In this case, a small buffer will be acquired and the header moved into it. If the received segment completes an SMB, then the procedures are pretty much as for FTP. If it does not, then the scheme is to at least move the received data (not the headers) to the host to free the INIC buffers and to save latency. The list of in-progress host buffers is maintained in the TCB and moved to the header buffer when the SMB is complete.

09781097 . 01101

WISTRON CORP. EXHIBIT 1031.057

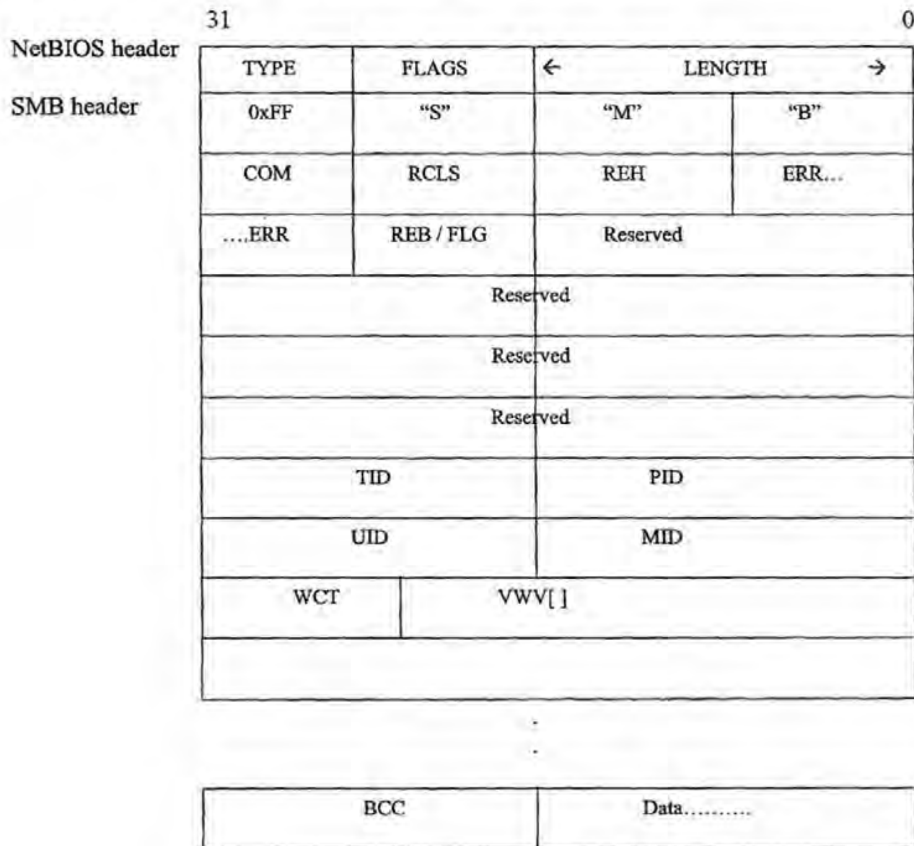
The final part of pure-receive processing is to fire off the delayed ACK timer for this segment.

5.2.3.2.2 Pure ACK

Pure ACK processing implies this TCB is the sender, so there may be transmit buffers that can be returned to the host. If so, send an event to the Transmit processor (or do the processing here). If there is more output available, send an event to the transmit processor. Then appropriate actions need to be taken with the retransmission timer.

5.2.3.3 SMB Processing

The following is the format of the SMB header of an SMB frame:



Notes (interesting fields):

LENGTH	17 bit Length of SMB message (0 – 128K)
COM	SMB command
WCT	Count (16 bit) of parameter words in VWV[]
VWV	Variable number of parameter words
BCC	Bytes of data following

The LENGTH field of the NetBIOS header will be used to determine when a complete SMB has been received and the header buffer with appropriate details can be posted to the host.
The interesting commands are the write commands: SMBwrite (0xB), SMBwriteBraw (0x1D), SMBwriteBmpx (0x1E), SMBwriteBs (0x1F), SMBwriteclose (0x2C), SMBwriteX (0x2F), SMBwriteunlock (0x14). These are interesting because they will have data to be aligned in host memory. The point to note about these commands is that they each have a different WCT field, so that the start offset of the data depends on the command type. SMB processing will thus need to be cognizant of these types.

5.2.4 Receive Details – No Valid Context

The design here is to provide as much assist as possible. Frames will be checksummed and the TCPIP headers may be split off.

5.2.5 Receive Notes

1. PRU_RCVD or the equivalent in Microsoft language: the host application has to tell the INIC when he has accepted the received data that has been queued. This is so that the INIC can update the receive window. It is an advantage for this mechanism to be efficient. This may be accomplished by piggybacking these on transmit requests (not necessarily for the same TCB).
2. Keepalive Timer: for a INIC-controlled TCB, the INIC will not maintain this timer. This leaves the host with the job of determining that the TCB is still active.
3. Timestamp option: it is useful to support this option in the fast path because the BSD implementation does. Also, it can be very helpful in getting a much better estimate of the round-trip time (RTT) which TCP needs to use.
4. Idle timer: the INIC will not maintain this timer (see Note 2 above).
5. Frame with no valid context: The INIC may split TCP/IP headers into a separate header buffer.

60019809.101497

WISTRON CORP. EXHIBIT 1031.059

5.3 Transmit Processing

5.3.1 Main Loop.

The following is a summary of the main loop of Transmit:

```
forever {
        while there are any Transmit events {
                if (a new event) {
                        if (no new context available)
                                ignore the event;
                }
                call appropriate event handler to service the event;
                this may make a waiting process runnable or set up
                a new process to be run (get free context, hddr buffer,
                TCB buffer, set the context up).
        }
        while any process contexts are runable {
                run them by jumping to the start/resume address;
                if (process complete)
                        free the context;
        }
}
```

5.3.2 Transmit Events

The events that will be processed on a given context and their sources are:

- accept a context (from the Host).
- release a context command (from the Host).
- release a context command (from Receive).
- valid send request and window > 0 (from host or RCV sequencer).
- valid send request and window = 0 (from host or RCV sequencer).
- send a window update (host has accepted data).
- persist timer expiration (persist timer).
- context-release event e.g. window shrank (XMT processing or retransmission timer).
- receive-release request ACK(from RCV sequencer).

5.3.3 Transmit Details – Valid Context

The following is an overview of the transmit flow:

The host posts a transmit request to the INIC by filling in a command buffer with appropriate data pointers etc and posting it to the INIC via the Command Buffer Address register. Note that there is one host command buffer queue, but there are 4 physical transmit lines. So each request needs to include an interface number as well as the context number. The INIC microcode will dma the command in and place it in 1 of 4 internal command queues which the transmit sequencer will work on. This is so that transmit processing can round-robin service these 4 queues to keep all 4 interfaces busy, and not

09/464283 [side stamp, rotated]
WISTRON CORP. EXHIBIT 1031.060

let a highly-active interface lock out the others (which would happen with a single queue).

The transmit request may be a segment that is less than the MSS, or it may be as much as a full 64K SMB READ. Obviously the former request will go out as one segment, the latter as a number of MSS-sized segments. The transmitting TCB must hold on to the request until all data in it has been transmitted and acked. Appropriate pointers to do this will be kept in the TCB. A large buffer is acquired from the free buffer fifo, and the MAC and TCP/IP headers are created in it. It may be quicker/simpler to keep a basic frame header set up in the TCB and either dma directly this into the frame each time. Then data is dmad from host memory into the frame to create an MSS-sized segment. This dma also checksums the data. Then the checksum is adjusted for the pseudo-header and placed into the TCP header, and the frame is queued to the MAC transmit interface which may be controlled by the third sequencer. The final step is to update various window fields etc in the TCB. Eventually either the entire request will have been sent and acked, or a retransmission timer will expire in which case the context is flushed to the host. In either case, the INIC will place a command response in the Response queue containing the command buffer handle from the original transmit command and appropriate status.

The above discussion has dealt how an actual transmit occurs. However the real challenge in the transmit processor is to determine whether it is appropriate to transmit at the time a transmit request arrives. There are many reasons not to transmit: the receiver's window size is <= 0, the Persist timer has expired, the amount to send is less than a full segment and an ACK is expected / outstanding, the receiver's window is not half-open etc. Much of the transmit processing will be in determining these conditions.

5.3.4 Transmit Details – No Valid Context

The main difference between this and a context-based transmit is that the queued request here will already have the appropriate MAC and TCP/IP (or whatever) headers in the frame to be output. Also the request is guaranteed not to be greater than MSS-sized in length. So the processing is fairly simple. A large buffer is acquired and the frame is dmad into it, at which time the checksum is also calculated. If the frame is TCP/IP, the checksum will be appropriately adjusted if necessary (pseudo-header etc) and placed in the TCP header. The frame is then queued to the appropriate MAC transmit interface. Then the command is immediately responded to with appropriate status through the Response queue.

5.3.5 Transmit Notes

1. Slow-start: the INIC will handle the slow-start algorithm that is now a part of the TCP standard. This obviates waiting until the connection is sending a full-rate before passing it to the INIC.
2. Window Probe vs Window Update: an explanation for posterity….

A Window Probe is sent from the sending TCB to the receiving TCB, and it means the sender has the receiver in PERSIST state. Persist state is entered when the receiver advertises a zero window. It is thus the state of the transmitting TCB. In this state, he sends periodic window probes to the receiver in case an ACK from the receiver has been lost. The receiver will return his latest window size in the ACK.

09/801,497
WISTRON CORP. EXHIBIT 1031.061

A Window Update is sent from the receiving TCB to the sending TCB, usually to tell him that the receiving window has altered. It is mostly triggered by the upper layer when it accepts some data. This probably means the sending TCB is viewing the receiving TCB as being in PERSIST state.

3. Persist state: it is designed to handle Persist state on the INIC. It seems unreasonable to throw a TCB back to the host just because its receiver advertised a zero window. This would normally be a transient situation, and would tend to happen mostly with clients that do not support slow-start. Alternatively, the code can easily be changed to throw the TCB back to the host as soon as a receiver advertises a zero window.
4. MSS-sized frames: the INIC code will expect all transmit requests for which it has no TCB to not be greater than the MSS. If any request is, it will be dropped and an appropriate response status posted.
5. Silly Window avoidance: as a receiver, the INIC will do the right thing here and not advertise small windows – this is easy. However it is necessary to also do things to avoid this as a sender, for the cases where a stupid client does advertise small windows. Without getting into too much detail here, the mechanism requires the INIC code to calculate the largest window advertisement ever advertised by the other end. It is an attempt to guess the size of the other end's receive buffer and assumes the other end never reduces the size of its receive buffer. See Stevens Vol. 1 pp. 325-326.

6 The Utility Processor

6.1 Summary

The following is a summary of the main functions of the utility sequencer of the microprocessor:

- look at the event queues: Event13Type & Event23Type (we assume there will be an event status bit for this - USE_EV13 and USE_EV23) in the events register; these are events from sequencers 1 and 2; they will mainly be XMIT requests from the XMT sequencer. Dequeue request and place the frame on the appropriate interface.
- RCV-frame support: in the model, RCV is done through VinicReceive() which is registered by the lower-edge driver, and is called at dispatch-level. This routine calls VinicTransferDataComplete() to check if the xfer (possibly DMA) of the frame into host buffers is complete. The latter rtne is also called at dispatch level on a DMA-completion interrupt. It queues complete buffers to the RCV sequencer via the normal queue mechanism.
- Other processes may also be employed here for supporting the RCV sequencer.
- service the following registers: (this will probably involve micro-interrupts)

  Header Buffer Address register:

    buffers are 256 bytes long on 256-byte boundaries.

    31-8 - physical addr in host of a set of
      contiguous hddr buffers

    7-0 - number of hddr buffers passed.

    Use contents to add to SmallHType queue

60061809.101497

WISTRON CORP. EXHIBIT 1031.062

Data Buffer Handle & Data Buffer Address registers:
buffers are 4K long aligned on 4K boundaries...
Use contents to add to the FreeType queue.

Command Buffer Address register:
buffers are multiple of 32 bytes up to 1K long (2**5 * 32)
31-5 - physical addr in host of cmd buffer
4-0 - length of cmd in bytes/32
(i.e. multiples of 32 bytes)
Points to host cmd; get FreeSType buffer and move
command into it; queue to Xmit0-Xmit3Type queues.

Response Buffer Address register:
buffers are 32 bytes long on 32-byte boundaries
31-8 - physical addr in host of a set of
contiguous resp buffers
7-0 - number of resp buffers passed.
Use contents to add to the ResponseType queue.

- low buffer threshold support: set approp bits in the ISR when the available-buffers count in the various queues filled by the host falls below a threshold.

6.2 Further Operations of the Utility Processor

The utility processor of the microprocessor housed on the INIC is responsible for setting up and implementing all configuration space and memory mapped operations, and also as described below, for managing the debug interface.

All data transfers, and other INIC initiated transfers will be done via DMA. Configuration space for both the network processor function and the utility processor function will define a single memory space for each. This memory space will define the basic communication structure for the host. In general, writing to one of these memory locations will perform a request for service from the INIC. This is detailed in the memory description for each function. This section defines much of the operation of the Host interface, but should be read in conjunction with the Host Interface Strategy for the Alacritech INIC to fully define the Host/INIC interface.

Two registers, DMA hardware and an interrupt function comprise the INIC interface to the Host through PCI. The interrupt function is implemented via a four bit register (PCI_INT) tied to the PCI interrupt lines. This register is directly accessed by the microprocessor.

THE MICROPROCESSOR uses two registers, the PCI_Data_Reg and the PCI_Address_Reg, to enable the Host to access Configuration Space and the memory space allocated to the INIC. These registers are not available to the Host, but are used by THE MICROPROCESSOR to enable Host reads and writes. The function of these two registers is as follows.

60061809.101497

WISTRON CORP. EXHIBIT 1031.063

PCI_Data_Reg

This register can be both read and written by THE MICROPROCESSOR. On write operations from the host, this register contains the data being sent from the host. On read operations, this register contains the data to be sent to the host.

PCI_Address_Reg

This is the control register for memory reads and writes from the host. The structure of the register is as follows:

Bit 31 – 24 Byte enable 7 – 0. Only the low order four bits are valid for 32 bit addressing mode.
Bit 23 – 0 Memory access
1 Configuration access
Bit 22 – 0 Read (to Host)
1 Write (from Host)

1 Bit 21 – 1 Data Valid

Bit 20 – 16 Reserved
Bit 15 – 0 Address

During a write operation from the Host the PCI_Data_Reg contains valid data after Data Valid is set in the PCI_Address_Reg. Both registers are locked until THE MICROPROCESSOR writes the PCI_Data_Reg, which resets Data Valid.

All read operations will be direct from SRAM. Memory space based reads will return 00. Configuration space reads will be mapped as follows:

Configuration Space 1	SRAM Address Offset
00	00
04	04
08	08
0C	0C
10	10
3C	14
Configuration Space 2	
00	00
04	18
08	08
0C	1C
10	20
3C	24

All other reads to configuration space will return 00.

09781609.101497

WISTRON CORP. EXHIBIT 1031.064

6.2.1 CONFIGURATION SPACE

The INIC is implemented as a multi-function device. The first device is the network controller, and the second device is the debug interface. An alternative production embodiment may implement only the network controller function. Both configuration space headers will be the same, except for the differences noted in the following description.

Vendor ID – This field will contain the Alacritech Vendor ID. One field will be used for both functions. The Alacritech Vendor ID is hex 139A.

Device ID – Chosen at Alacritech on a device specific basis. One field will be used for both functions.

Command – Initialized to 00. All bits defined below as not enabled (0) will remain 0. Those that are enabled will be set to 0 or 1 depending on the state of the system. Each function (network and debug) will have its own command field.

Bit 0 – 0 I/O accesses are not enabled
Bit 1 – 1 Memory accesses are enabled
Bit 2 – 1 Bus master is enabled
Bit 3 – 0 Special Cycle is not enabled
Bit 4 – 1 Memory Write and Invalidate is enabled
Bit 5 – 0 VGA palette snooping is not enabled
Bit 6 – 1 Parity checking is enabled
Bit 7 – 0 Address data stepping is not enabled
Bit 8 – SERR# is enabled
Bit 9 – 0 Fast back to back is not enabled

Status – This is not initialized to zero. Each function will have its own field. The configuration is as follows:
Bit 5 – 1 66 MHz capable is enabled. This bit will be set if the INIC Detects the system running at 66 MHz on reset
Bit 6 – 0 User Definable Features is not enabled
Bit 7 – 1 Fast Back-to-Back slave transfers enabled
Bit 8 – 1 Parity Error enabled – This bit is initialized to 0
Bit 9,10 – 00 – Fast device select will be set if we are at 33 MHz
01 – Medium device select will be set if we are at 66 MHz
Bit 11 – 1 Target Abort is implemented. Initialized to 0.
Bit 12 – 1 Target Abort is implemented. Initialized to 0.
Bit 13 – 1 Master Abort is implemented. Initialized to 0.
Bit 14 – 1 SERR# is implemented. Initialized to 0.
Bit 15 – 1 Parity error is implemented. Initialized to 0.

Revision ID – The revision field will be shared by both functions.

Class Code – This is 02 00 00 for the network controller, and for the debug interface. The field will be shared.

60061809.101497
WISTRON CORP. EXHIBIT 1031.065

Cache Line Size – This is initialized to zero. Supported sizes are 16, 32, 64 and 128 bytes. This hardware register is replicated in SRAM and supported separately for each function, but THE MICROPROCESSOR will implement the value set in Configuration Space 1 (the network processor).

Latency Timer – This is initialized to zero. The function is supported. This hardware register is replicated in SRAM. Each function is supported separately, but THE MICROPROCESSOR will implement the value set in Configuration Space 1 (the network processor).

Header Type – This is set to 80 for both functions, but will be supported separately.

BIST – Is implemented. In addition to responding to a request to run self test, if test after reset fails, a code will be set in the BIST register. This will be implemented separately for each function.

Base Address Register – A single base address register is implemented for each function. It is 64 bits in length, and the bottom four bits are configured as follows:

Bit 0 – 0 Indicates memory base address
Bit 1,2 – 00 Locate base address anywhere in 32 bit memory space
Bit 3 – 1 Memory is prefetchable

CardBus CIS Pointer – Not implemented—initialized to 0.

Subsystem Vendor ID – Not implemented—initialized to 0.

Subsystem ID – Not implemented—initialized to 0.

Expansion ROM Base Address – Not implemented—initialized to 0.

Interrupt Line – Implemented—initialized to 0. This is implemented separately for each function.

Interrupt Pin – This is set to 01, corresponding to INTA# for the network controller, and 02, corresponding to INTB# for the debug interface. This is implemented separately for each function.

Min_Gnt – This can be set at a value in the range of 10, to allow reasonably long bursts on the bus. This is implemented separately for each function.

Max_Lat – This can be set to 0 to indicate no particular requirement for frequency of access to PCI. This is implemented separately for each function.

6.2.2 MEMORY SPACE

Because each of the following functions may or may not reside in a single location, and may or may not need to be in SRAM at all, the address for each is really only used as an identifier (label). There is, therefore, no control block anywhere in memory that represents this memory space. When the host writes one of these registers, the utility

60061809.101497

WISTRON CORP. EXHIBIT 1031.066

processor will construct the data required and transfer it. Reads to this memory will generate 00 for data.

6.2.2.1 Network Processor

The following four byte registers, beginning at location h00 of the network processor's allocated memory, are defined.

00 – Interrupt Status Pointer -- Initialized by the host to point to a four byte area where status is stored

04 – Interrupt Status – Returned status from host. Sent after one or more status conditions have been reset. Also an interlock for storing any new status. Once status has been stored at the Interrupt Status Pointer location, no new status will be stored until the host writes the Interrupt Status Register. New status will be ored with any remaining uncleared status (as defined by the contents of the returned status) and stored again at the Interrupt Status Pointer location. Bits are as follows:
Bit 31 – ERR -- Error bits are set
Bit 30 – RCV – Receive has occurred
Bit 29 – XMT – Transmit command complete
Bit 25 – RMISS – Receive drop occurred due to no buffers

08 – Interrupt Mask – Written by the host. Interrupts are masked for each of the bits in the interrupt status when the same bit in the mask register is set. When the Interrupt Mask register is written and as a result a status bit is unmasked, an interrupt is generated. Also, when the Interrupt Status Register is written, enabling new status to be stored, when it is stored if a bit is stored that is not masked by the Interrupt Mask, an interrupt is generated.

0C – Header Buffer Address – Written by host to pass a set of header buffers to the INIC.

10 – Data Buffer Handle – First register to be written by the Host to transfer a receive data buffer to the INIC. This data is Host reference data. It is not used by the INIC, it is returned with the data buffer. However, to insure integrity of the buffer, this register must be interlocked with the Data Buffer Address register. Once the Data Buffer Address register has been written, neither register can be written until after the Data Buffer Handle register has been read by THE MICROPROCESSOR.

14 – Data Buffer Address – Pointer to the data buffer being sent to the INIC by the Host. Must be interlocked with the Data Buffer Handle register.

18 – Command Buffer Address XMT0 – Pointer to a set of command buffers sent by the Host. THE MICROPROCESSOR will DMA the buffers to local DRAM found on the FreeSType queue and queue the Command

09/601,4897 (side stamp: 09801497.100605)
WISTRON CORP. EXHIBIT 1031.067

Buffer Address XMT0 with the local address replacing the host Address.

1C – Command Buffer Address SMT1

20 – Command Buffer Address SMT2

24 – Command Buffer Address SMT3

28 – Response Buffer Address -- Pointer to a set of response buffers sent by the Host. These will be treated in the same fashion as the Command Buffer Address registers.

6.2.2.2 Utility Processor

Ending status will be handled by the utility processor in the same fashion as it is handled by the network processor. At present two ending status conditions are defined B31 – command complete, and B30 – error. When end status is stored an interrupt is generated.

Two additional registers are defined, Command Pointer and Data Pointer. The Host is responsible for insuring that the Data Pointer is valid and points to sufficient memory before storing a command pointer. Storing a command pointer initiates command decode and execution by the debug processor. The Host must not modify either command or Data Pointer until ending status has been received, at which point a new command may be initiated. Memory space is write only by the Host, reads will receive 00. The format is as follows:

00 – Interrupt Status Pointer -- Initialized by the host to point to a four byte area where status is stored

04 – Interrupt Status – Returned status from host. Sent after one or more status conditions have been reset. Also an interlock for storing any new status. Once status has been stored at the Interrupt Status Pointer location, no new status will be stored until the host writes the Interrupt Status Register. New status will be ored with any remaining uncleared status (as defined by the contents of the returned status) and stored again at the Interrupt Status Pointer location. Bits are as follows:
Bit 31 – CC – Command Complete
Bit 30 – ERR -- Error
Bit29 – Transmit Processor Halted
Bit28 – Receive Processor Halted
Bit27 – Utility Processor Halted

08 – Interrupt Mask – Written by the host. Interrupts are masked for each of the bits in the interrupt status when the same bit in the mask register is set. When the Interrupt Mask register is written and as a result a status bit is unmasked, an interrupt is generated. Also,

09/801497.002609

WISTRON CORP. EXHIBIT 1031.068

when the Interrupt Status Register is written, enabling new status to be stored, when it is stored if a bit is stored that is not masked by the Interrupt Mask, an interrupt is generated.

0C – Command Pointer – Points to command to be executed. Storing this pointer initiates command decode and execution.

10 – Data Pointer – Points to the data buffer. This is used for both read and write data, determined by the command function.

7 Debug Interface

In order to provide a mechanism to debug the microcode running on the microprocessor sequencers, a debug process has been defined which will run on the utility sequencer. This processor will interface with a control program on the host processor over PCI.

7.1 PCI Interface

This interface is defined in the combination of the Utility Processor and the Host Interface Strategy sections, above.

7.2 Command Format

The first byte of the command, the command byte, defines the structure of the remainder of the command. The first five bits of the command byte are the command itself. The next bit is used to specify an alternate processor, and the last two bits specify which processors are intended for the command.

7.2.1 Command Byte

7 – 3	2	1 – 0
Command	Alt. Proc.	Processor

7.2.2 Processor Bits

00 – Any Processor
01 – Transmit Processor
10 – Receive Processor
11 – Utility Processor

WISTRON CORP. EXHIBIT 1031.069

60051809.101497

7.2.3 Alternate Processor

This bit defines which processor should handle debug processing if the utility processor is defined as the processor in debug.

0 – Transmit Processor
1 – Receive Processor

7.2.4 Single Byte Commands

00 – Halt

This command asynchronously halts the processor.

08 – Run

This command starts the processor.

10 – Step

This command steps the processor.

7.2.5 Eight Byte Commands

18 – Break

0	1	2 - 3	4 – 7
Command	Reserved	Count	Address

This command sets a stop at the specified address. A count of 1 causes the specified processor to halt the first time it executes the instruction. A count of 2 or more causes the processor to halt after that number of executions. The processor is halted just before executing the instruction. A count of 0 does not halt the processor, but causes a sync signal to be generated. If a second processor is set to the same break address, the count data from the first break request is used, and each time either processor executes the instruction the count is decremented.

20 – Reset Break

0	1 - 3	4 – 7
Command	Reserved	Address

60061809.101497

WISTRON CORP. EXHIBIT 1031.070

This command resets a previously set break point at the specified address. Reset break fully resets that address. If multiple processors were set to that break point, all will be reset.

28 – Dump

0	1	2 - 3	4 – 7
Command	Descriptor	Count	Address

This command transfers to the host the contents of the descriptor. For descriptors larger than four bytes, a count, in four byte increments is specified. For descriptors utilizing an address the address field is specified.

7.2.6 Descriptor

00 – Register

This descriptor uses both count and address fields. Both fields are four byte based (a count of 1 transfers four bytes).

01 – Sram

This descriptor uses both count and address fields. Count is in four byte blocks. Address is in bytes, but if it is not four byte aligned, it is forced to the lower four byte aligned address.

02 – Dram

This descriptor uses both count and address fields. Count is in four byte blocks. Address is in bytes, but if it is not four byte aligned, it is forced to the lower four byte aligned address

03 – Cstore

This descriptor uses both count and address fields. Count is in four byte blocks. Address is in bytes, but if it is not four byte aligned, it is forced to the lower four byte aligned address

Stand-alone descriptors:

The following descriptors do not use either the count or address fields. They transfer the contents of the referenced register.

04 – CPU_STATUS

05 – PC

60918509.101497

WISTRON CORP. EXHIBIT 1031.071

06 – ADDR_REGA

07 – ADDR_REGB

08 – RAM_BASE

09 – FILE_BASE

0A – INSTR_REG_L

0B – INSTR_REG_H

0C – MAC_DATA

0D – DMA_EVENT

0E – MISC_EVENT

0F – Q_IN_RDY

10 – Q_OUT_RDY

11 – LOCK STATUS

12 – STACK - This returns 12 bytes

13 – Sense _ Reg

This register contains four bytes of data. If error status is posted for a command, if the next command that is issued reads this register, a code describing the error in more detail may be obtained. If any command other than a dump of this register is issued after error status, sense information will be reset.

30 – Load

0	1	2 - 3	4 – 7
Command	Descriptor	Count	Address

This command transfers from the host the contents of the descriptor. For descriptors larger than four bytes, a count, in four byte increments is specified. For descriptors utilizing an address the address field is specified.

7.2.7 Descriptor

00 – Register

This descriptor uses both count and address fields. Both fields are four byte based.

60081005 . 101497

WISTRON CORP. EXHIBIT 1031.072

01 – Sram

This descriptor uses both count and address fields. Count is in four byte blocks. Address is in bytes, but if it is not four byte aligned, it is forced to the lower four byte aligned address.

02 – Dram

This descriptor uses both count and address fields. Count is in four byte blocks. Address is in bytes, but if it is not four byte aligned, it is forced to the lower four byte aligned address

03 – Cstore

This descriptor uses both count and address fields. Count is in four byte blocks. Address is in bytes, but if it is not four byte aligned, it is forced to the lower four byte aligned address. This applies to WCS only.

Stand-alone descriptors:

The following descriptors do not use either the count or address fields. They transfer the contents of the referenced register.

04 – ADDR_REGA

05 – ADDR_REGB

06 – RAM_BASE

07 – FILE_BASE

08 – MAC_DATA

09 – Q_IN_RDY

0A – Q_OUT_RDY

0B – DBG_ADDR

38 – Map

This command allows an instruction in ROM to be replaced by an instruction in WCS. The new instruction will be located in the Host buffer. It will be stored in the first eight bytes of the buffer, with the high bits unused. To reset a mapped out instruction, map it to location 00.

0	1 – 3	4 – 7
Command	Address to Map To	Address to Map Out

60061809 .101497

WISTRON CORP. EXHIBIT 1031.073

8 HARDWARE SPECIFICATION

FEATURES

• Peripheral Component Interconnect (PCI) Interface

- Universal PCI interface supports both 5.0V and 3.3V signaling environments.
- Supports both 32-bit and 64 bit PCI interface.
- Supports PCI clock frequencies from 15MHz to 66MHz
- High performance bus mastering architecture.
- Host memory based communications reduce register accesses.
- Host memory based interrupt status word reduces register reads.
- Plug and Play compatible.
- PCI specification revision 2.1 compliant.
- PCI bursts up to 512 bytes.
- Supports cache line operations up to 128 bytes.
- Both big-endian and little-endian byte alignments supported.
- Supports Expansion ROM.

• Network Interface

- Four internal 802.3 and ethernet compliant Macs.
- Media Independent Interface (MII) supports external PHYs.
- 10BASE-T, 100BASE-TX/FX and 100BASE-T4 supported.
- Full and half-duplex modes supported.
- Automatic PHY status polling notifies system of status change.
- Provides SNMP statistics counters.
- Supports broadcast and multicast packets.
- Provides promiscuous mode for network monitoring or multiple unicast address detection.
- Supports "huge packets" up to 32KB.
- Mac-layer loop-back test mode.
- Supports auto-negotiating Phys.

60971808.101497

WISTRON CORP. EXHIBIT 1031.074

- **Memory Interface**

- External Dram buffering of transmit and receive packets.
- Buffering configurable as 4MB, 8MB, 16MB or 32MB.
- 32-bit interface supports throughput of 224MB/s
- Supports external FLASH ROM up to 4 MB, for diskless boot applications.
- Supports external serial EEPROM for custom configuration and Mac addresses.

- **Protocol Processor**

- High speed, custom, 32-bit processor executes 66 million instructions per second.
- Processes IP, TCP and NETBIOS protocols.
- Supports up to 256 resident TCP/IP contexts.
- Writable control store (WCS) allows field updates for feature enhancements.

- **Power**

- 3.3V chip operation.
- PCI controlled 5.0V/3.3V I/O cell operation.

- **Packaging**

- 272-pin plastic ball grid array.
- 91 PCI signals.
- 68 MII signals.
- 58 external memory signals.
- 1 clock signal.
- 54 signals split between power and ground.
- 272 total pins.

60061809 . 101497

WISTRON CORP. EXHIBIT 1031.075

GENERAL DESCRIPTION

The microprocessor is a 32-bit, full-duplex, four channel, 10/100-Megabit per second (Mbps), Intelligent Network Interface Controller, designed to provide high-speed protocol processing for server applications. It combines the functions of a standard network interface controller and a protocol processor within a single chip. Although designed specifically for server applications, The microprocessor can be used by PCs, workstations and routers or anywhere that TCP/IP protocols are being utilized.

When combined with four 802.3/MII compliant Phys and Synchronous Dram (SDram), the INIC comprises four complete ethernet nodes. It contains four 802.3/ethernet compliant Macs, a PCI Bus Interface Unit (BIU), a memory controller, transmit fifos, receive fifos and a custom TCP/IP/NETBIOS protocol processor. The INIC supports 10Base-T , 100Base-TX, 100Base-FX and 100Base-T4 via the MII interface attachment of appropriate Phys.

The INIC Macs provide statistical information that may be used for SNMP. The Macs operate in promiscuous mode allowing the INIC to function as a network monitor, receive broadcast and multicast packets and implement multiple Mac addresses for each node.

Any 802.3/MII compliant PHY can be utilized, allowing the INIC to support 10BASE-T, 10BASE-T2, 100BASE-TX, 100Base-FX and 100BASE-T4 as well as future interface standards. PHY identification and initialization is accomplished through host driver initialization routines. PHY status registers can be polled continuously by the INIC and detected PHY status changes reported to the host driver. The Mac can be configured to support a maximum frame size of 1518 bytes or 32768 bytes.

The 64-bit, multiplexed BIU provides a direct interface to the PCI bus for both slave and master functions. The INIC is capable of operating in either a 64-bit or 32-bit PCI environment, while supporting 64-bit addressing in either configuration. PCI bus frequencies up to 66MHz are supported yielding instantaneous bus transfer rates of 533MB/s. Both 5.0V and 3.3V signaling environments can be utilized by the INIC. Configurable cache-line size up to 256B will accommodate future architectures, and Expansion ROM/Flash support allows for diskless system booting. Non-PC applications are supported via programmable big and little endian modes. Host based communication has been utilized to provide the best system performance possible.

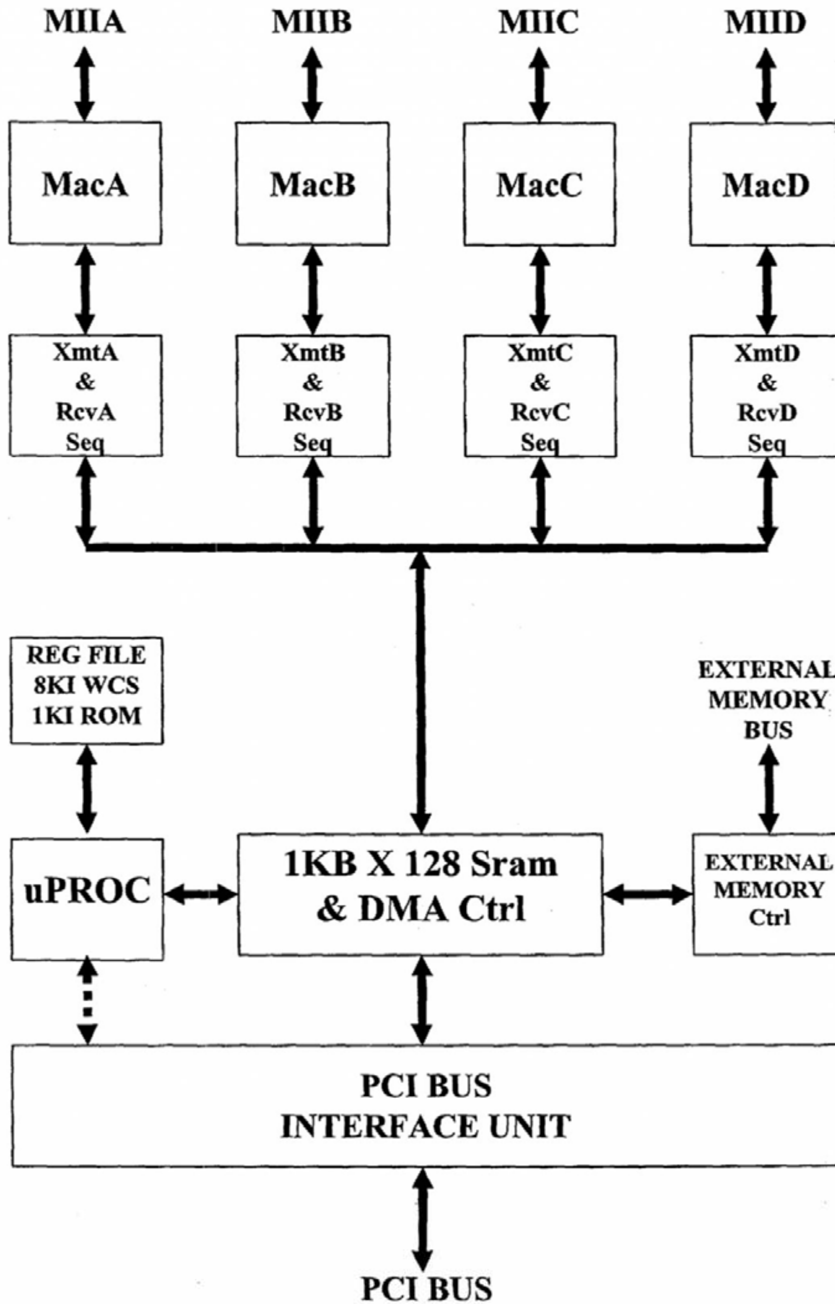
The INIC supports Plug-N-Play auto-configuration through the PCI configuration space. External pull-up and pull-down resistors, on the memory I/O pins, allow selection of various features during chip reset. Support of an external eeprom allows for local storage of configuration information such as Mac addresses.

External SDram provides frame buffering, which is configurable as 4MB, 8MB, 16MB or 32MB using the appropriate SIMMs. Use of -10 speed grades yields an external buffer bandwidth of 224MB/s. The buffer provides temporary storage of both incoming and outgoing frames. The protocol processor accesses the frames within the buffer in order to implement TCP/IP and NETBIOS. Incoming frames are processed, assembled then transferred to host memory under the control of the protocol processor. For transmit, data is moved from host memory to buffers where various headers are created before being transmitted out via the Mac.

09461809 . 101497

WISTRON CORP. EXHIBIT 1031.076

BLOCK DIAGRAM



60061809 . 101497

WISTRON CORP. EXHIBIT 1031.077

OUTLINE

• Cores/Cells

LSI Logic **Ethernet-110 Core**, 100Base & 10Base Mac with MII interface.

LSI Logic **single port Sram, triple port Sram and ROM** available.

LSI Logic **PCI 66MHz, 5V compatible I/O cell.**

LSI Logic **PLL**

• Die Size / Pin Count

LSI Logic G10 process.

MODULE	DESCR	SPEED	AREA
Scratch RAM,	1Kx128 sport,	4.37 ns nom.,	06.77 mm^2
WCS,	8Kx49 sport,	6.40 ns nom.,	18.29 mm^2
MAP,	128x7 sport,	3.50 ns nom.,	00.24 mm^2
ROM,	1Kx49 32col,	5.00 ns nom.,	00.45 mm^2
REGs,	512x32 tport,	6.10 ns nom.,	03.49 mm^2
Macs,	.75 mm^2 x 4 =		03.30 mm^2
PLL,	.5 mm^2 =		00.55 mm^2
MISC LOGIC,	117,260 gates / (5035 gates / mm^2) =		23.29 mm^2
TOTAL CORE			56.22 mm^2

(Core side)^2		=	56.22 mm^2
Core side		=	07.50 mm
Die side	= core side + 1.0 mm (I/O cells)	=	08.50 mm
Die area	= 8.5 mm x 8.5 mm	=	**72.25 mm^2**
Pads needed	= 220 signals x 1.25 (vss, vdd)	=	**275 pins**
LSI PBGA		=	**272 pins**

WISTRON CORP. EXHIBIT 1031.078

• **Datapath Bandwidth**

(10MB/s/100Base) x 2 (full duplex) x 4 connections	=	80 MB/s
Average frame size	=	512 B
Frame rate = 80MB/s / 512B	=	156,250 frames / s
Cpu overhead / frame = (256 B context read) + (64B header read) + (128B context write) + (128B misc.)	=	512B / frame
Total bandwidth = (512B in) + (512B out) + (512B Cpu)	=	1536B / frame
Dram Bandwidth required = (1536B/frame) x (156,250 frames/s)	=	**240MB/s**
Dram Bandwidth @ 60MHz = (32 bytes / 167ns)	=	**202MB/s**
Dram Bandwidth @ 66MHz = (32 bytes / 150ns)	=	**224MB/s**
PCI Bandwidth required	=	**80MB/s**
PCI Bandwidth available @ 30 MHz, 32b, average	=	**46MB/s**
PCI Bandwidth available @ 33 MHz, 32b, average	=	**50MB/s**
PCI Bandwidth available @ 60 MHz, 32b, average	=	**92MB/s**
PCI Bandwidth available @ 66 MHz, 32b, average	=	**100MB/s**
PCI Bandwidth available @ 30 MHz, 64b, average	=	**92MB/s**
PCI Bandwidth available @ 33 MHz, 64b, average	=	**100MB/s**
PCI Bandwidth available @ 60 MHz, 64b, average	=	**184MB/s**
PCI Bandwidth available @ 66 MHz, 64b, average	=	**200MB/s**

• **Cpu Bandwidth**

Receive frame interval = 512B / 40MB/s	=	12.8us
Instructions / frame @ 60MHz = (12.8us/frame) / (50ns/instruction)	=	**256 instructions/frame**
Instructions / frame @ 66MHz = (12.8us/frame) / (45ns/instruction)	=	**284 instructions/frame**
Required instructions / frame (per Clive)	=	**250 instructions/frame**

09781060 . 101497

WISTRON CORP. EXHIBIT 1031.079

- **Performance Features**

- 512 registers improve performance through reduced scratch ram accesses and reduced instructions.
- Register windowing eliminates context-switching overhead.
- Separate instruction and data paths eliminate memory contention.
- Totally resident control store eliminates stalling during instruction fetch.
- Multiple logical processors eliminate context switching and improve real-time response.
- Pipelined architecture increases operating frequency.
- Shared register and scratch ram improve inter-processor communication.
- Fly-by state-Machine assists address compare and checksum calculation.
- TCP/IP-context caching reduces latency.
- Hardware implemented queues reduce Cpu overhead and latency.
- Horizontal microcode greatly improves instruction efficiency.
- Automatic frame DMA and status between Mac and dram buffer.
- Deterministic architecture coupled with context switching eliminates processor stalls.

60061809.101497

WISTRON CORP. EXHIBIT 1031.080

PROCESSOR

The processor is a convenient means to provide a programmable state-machine which is capable of processing incoming frames, processing host commands, directing network traffic and directing PCI bus traffic. Three processors are implemented using shared hardware in a three-level pipelined architecture which launches and completes a single instruction for every clock cycle. The instructions are executed in three distinct phases corresponding to each of the pipeline stages where each phase is responsible for a different function.

The first instruction phase writes the instruction results of the last instruction to the destination operand, modifies the program counter (Pc), selects the address source for the instruction to fetch, then fetches the instruction from the control store. The fetched instruction is then stored in the instruction register at the end of the clock cycle.

The processor instructions reside in the on-chip control-store, which is implemented as a mixture of ROM and Sram. The ROM contains 1K instructions starting at address 0x0000 and aliases each 0x0400 locations throughout the first 0x8000 of instruction space. The Sram (WCS) will hold up to 0x2000 instructions starting at address 0x8000 and aliasing each 0x2000 locations throughout the last 0x8000 of instruction space. The ROM and Sram are both 49-bits wide accounting for bits [48:0] of the instruction microword. A separate mapping ram provides bits [55:49] of the microword (MapAddr) to allow replacement of faulty ROM based instructions. The mapping ram has a configuration of 128x7 which is insufficient to allow a separate map address for each of the 1K ROM locations. To allow re-mapping of the entire 1K ROM space, the map ram address lines are connected to the address bits **Fetch**[9:3]. The result is that the ROM is re-mapped in blocks of 8 contiguous locations.

The second instruction phase decodes the instruction which was stored in the instruction register. It is at this point that the map address is checked for a non-zero value which will cause the decoder to force a **Jmp** instruction to the map address. If a non-zero value is detected then the decoder selects the source operands for the Alu operation based on the values of the **OpdASel, OpdBSel** and **AluOp** fields. These operands are then stored in the decode register at the end of the clock cycle. Operands may originate from **File**, **Sram**, or flip-flop based registers. The second instruction phase is also where the results of the previous instruction are written to the **Sram**.

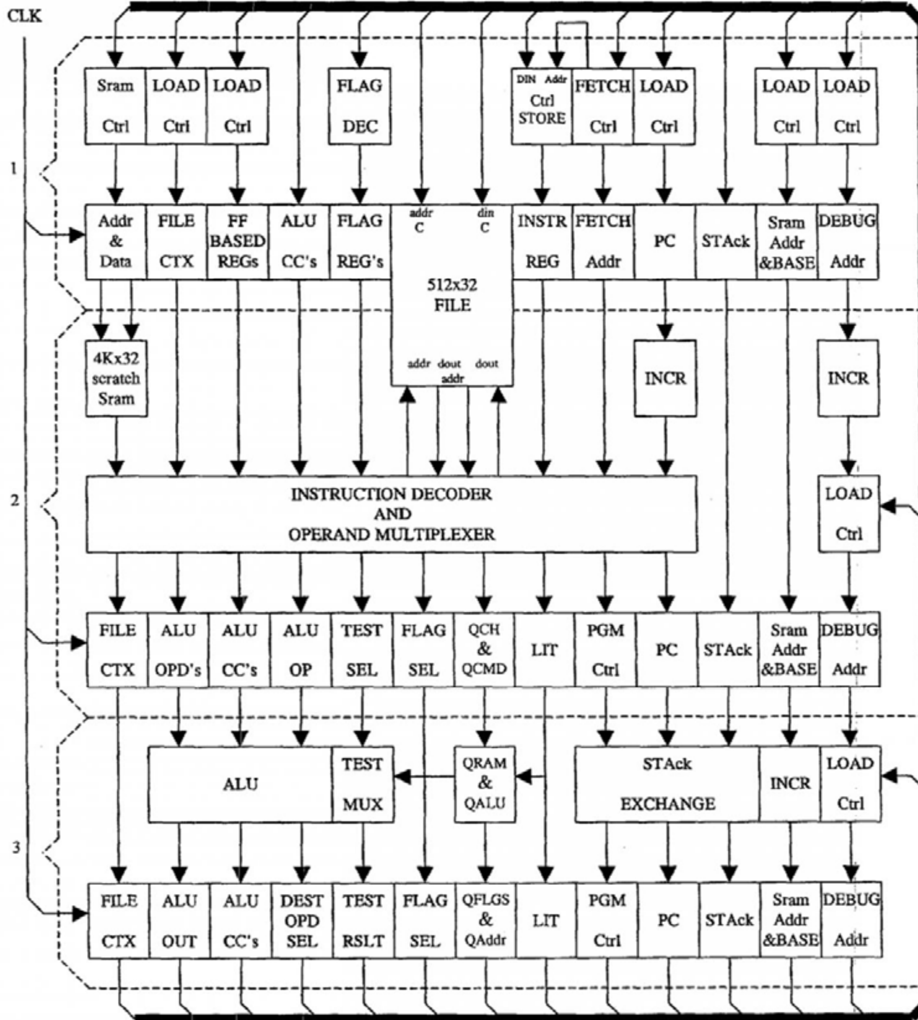
The third instruction phase is when the actual Alu operation is performed, the test condition is selected and the Stack push and pop are implemented. Results of the Alu operation are stored in the results register at the end of the clock cycle.

Following is a block diagram which shows the hardware functions associated with each of the instruction phases. Note that various functions have been distributed across the three phases of the instruction execution in order to minimize the combinatorial delays within any given phase.

60061803.101497

WISTRON CORP. EXHIBIT 1031.081

Cpu BLOCK-DIAGRAM



60061809.101497

INSTRUCTION SET

The micro-instructions are divided into six types according to the program control directive. The micro-instruction is further divided into sub-fields for which the definitions are dependent upon the instruction type. The six instruction types are listed below.

INSTRUCTION-WORD FORMAT

TYPE	[55:49]	[48:47]	[46:42]	[41:33]	[32:24]	[23:16]	[15:00]
Jcc	0b0000000	0b00,	AluOp,	OpdASel,	OpdBSel,	TstSel,	Literal
Jmp	0b0000000	0b01,	AluOp,	OpdASel,	OpdBSel,	FlgSel,	Literal
Jsr	0b0000000	0b10,	AluOp,	OpdASel,	OpdBSel,	FlgSel,	Literal
Rts	0b0000000	0b11,	AluOp,	OpdASel,	OpdBSel,	0hff,	Literal
Nxt	0b0000000	0b11,	AluOp,	OpdASel,	OpdBSel,	FlgSel,	Literal
Map	MapAddr	0bXX,	0bXXXXX,	0bXXXXXXXXX,	0bXXXXXXXXX,	0hXX,	0hXXXX

All instructions include the Alu operation (**AluOp**), operand "A" select (**OpdASel**), operand "B" select (**OpdBSel**) and **Literal** fields. Other field usage depends upon the instruction type.

The "jump condition code" (**Jcc**) instruction causes the program counter to be altered if the condition selected by the "test select" (**TstSel**) field is asserted. The new program counter (**Pc**) value is loaded from either the **Literal** field or the **AluOut** as described in the following section and the **Literal** field may be used as a source for the Alu or the ram address if the new **Pc** value is sourced by the Alu.

The "jump" (**Jmp**) instruction causes the program counter to be altered unconditionally. The new program counter (**Pc**) value is loaded from either the **Literal** field or the **AluOut** as described in the following section. The format allows instruction bits 23:16 to be used to perform a flag operation and the **Literal** field may be used as a source for the Alu or the ram address if the new **Pc** value is sourced by the Alu.

The "jump subroutine" (**Jsr**) instruction causes the program counter to be altered unconditionally. The new program counter (**Pc**) value is loaded from either the **Literal** field or the **AluOut** as described in the following section. The old program counter value is stored on the top location of the Pc-Stack which is implemented as a LIFO memory. The format allows instruction bits 23:16 to be used to perform a flag operation and the **Literal** field may be used as a source for the **Alu** or the ram address if the new **Pc** value is sourced by the **Alu.**

The "**Nxt**" (**Nxt**) instruction causes the program counter to increment. The format allows instruction bits 23:16 to be used to perform a flag operation and the **Literal** field may be used as a source for the **Alu** or the ram address.

The "return from subroutine" (**Rts**) instruction is a special form of the **Nxt** instruction in which the "flag operation" (**FlgSel**) field is set to a value of 0hff. The current **Pc** value is replaced with the last value stored in the stack. The **Literal** field may be used as a source for the **Alu** or the ram address.

The **Map** instruction is provided to allow replacement of instructions which have been stored in **ROM** and is implemented any time the "map enable" (**MapEn**) bit has been set and the content of the "map address" (**MapAddr**) field is non-zero. The instruction decoder forces a jump instruction with the **Alu** operation and destination fields set to pass the **MapAddr** field to the program control block.

The program control is determined by a combination of **PgmCtrl**, **DstOpd**, **FlgSel** and **TstSel**. The behavior of the program control is defined with the following "C-like" description.

09/801497
WISTRON CORP. EXHIBIT 1031.083

SEQUENCER BEHAVIOR

```
if (MapEn & (MapAddr != 0b0000000)){                        //re-map instr
          Stackc = Stackc;
          StackB = StackB;
          StackA = StackA;
       InstrAddr = 0h8000 | Pc[2:0] | (MapAddr << 3);
              Pc = InstrAddr + (Execute & ~DbgMd);
           Fetch = DbgMd ? DbgAddr:InstrAddr;
         DbgAddr = DbgAddr + (Execute & DbgMd);}

else if (PgmCtrl == Jcc){                                    //conditional jump
          Stackc = Stackc;
          StackB = StackB;
          StackA = StackA;
       InstrAddr = ~Tst@TstSel ? Pc:(AluDst==Pc) ? AluOut:Literal;
              Pc = InstrAddr + (Execute & ~DbgMd)
           Fetch = DbgMd ? DbgAddr:InstrAddr;
         DbgAddr = DbgAddr + (Execute & DbgMd);}

else if (PgmCtrl == Jmp){                                    //jump
          Stackc = Stackc;
          StackB = StackB;
          StackA = StackA;
       InstrAddr = (AluDst == Pc) ? AluOut:Literal;
              Pc = InstrAddr + (Execute & ~DbgMd)
           Fetch = DbgMd ? DbgAddr:InstrAddr;
         DbgAddr = DbgAddr + (Execute & DbgMd);}

else if (PgmCtrl == Jsr){                                    //jump subroutine
          Stackc = StackB;
          StackB = StackA;
          StackA = Pc;
       InstrAddr = (AluDst == Pc) ? AluOut:Literal;
              Pc = InstrAddr + (Execute & ~DbgMd)
           Fetch = DbgMd ? DbgAddr:InstrAddr;
         DbgAddr = DbgAddr + (Execute & DbgMd);}

else if (FlgSel == Rts){                                     //return subroutine
       InstrAddr = StackA;
          StackA = StackB;
          StackB = Stackc;
          Stackc = ErrVec;
              Pc = InstrAddr + (Execute & ~DbgMd)
           Fetch = DbgMd ? DbgAddr:InstrAddr;
         DbgAddr = DbgAddr + (Execute & DbgMd);}

else    {
       InstrAddr = Pc;                                       //continue
          StackA = StackA;
          StackB = StackB;
          Stackc = Stackc;
              Pc = InstrAddr + (Execute & ~DbgMd)
           Fetch = DbgMd ? DbgAddr:InstrAddr;
         DbgAddr = DbgAddr + (Execute & DbgMd);}
```

09081097

WISTRON CORP. EXHIBIT 1031.084

ALU OPERATIONS

```
AluOp      OPERATION

0b00000    A = (A & ~(1 << B));                                  //bit clear
           C = 0; V = (B >= 32) ? 1:0;

0b00001    A = (A & B);                                          //logical and
           C = 0; V = 0;

0b00010    A = (Literal & B);                                    //logical and
           C = 0; V = 0;

0b00011    A = (~Literal & B);                                   //logical and not
           C = 0; V = 0;

0b00100    A = (A | (1 << B));                                   //bit set
           C = 0; V = (B >= 32) ? 1:0;

0b00101    A = (A | B);                                          //logical or
           C = 0; V = 0;

0b00110    A = (Literal | B);                                    //logical or
           C = 0; V = 0;

0b00111    A = (~Literal | B);                                   //logical or not
           C = 0; V = 0;

0b01000    for (i=31; i>=0; i--) if B[i] continue; A=i;          //priority enc
           C = 0; V = (B) ? 0:1;

0b01001    A = (A ^ B);                                          //logical xor
           C = 0; V = 0;

0b01010    A = ({Literal} ^ B);                                  //logical xor
           C = 0; V = 0;

0b01011    A = ({~Literal} ^ B);                                 //logical xor not
           C = 0; V = 0;

0b01100    A = B;                                                //move
           C = 0; V = 0;

0b01101    A = B[31:24] ^ B[23:16] ^ B[15:08] ^ B[07:00];//hash
           C = 0; V = 0;

0b01110    A = {B[23:16],B[31:24],B[07:00],B[15:08]};            //swap bytes
           C = 0; V = 0;

0b01111    A = {B[15:00], B[31:16]};                             //swap doublets
           C = 0; V = 0;
```


WISTRON CORP. EXHIBIT 1031.085

AluOp	FUNCTION	
0b10000	`A = (A + B);` `C = (A + B)[32]; V = 0;`	`//add B`
0b10001	`A = (A + B + C);` `C = (A + B + C)[32]; V = 0;`	`//add B, carry`
0b10010	`A = (Literal + B);` `C = (Literal + B)[32]; V = 0;`	`//add constant`
0b10011	`A = (-Literal + B);` `C = (-Literal + B)[32]; V = 0;`	`//sub constant`
0b10100	`A = (A - B);` `C = (A - B)[32]; V = 0;`	`//sub B`
0b10101	`A = (A - B - ~C);` `C = (A - B - ~C)[32]; V = 0;`	`//sub B, borrow`
0b10110	`A = (-A + B);` `C = (-A + B)[32]; V = 0;`	`//sub A`
0b10111	`A = (-A + B - ~C);` `C = (-A + B - ~C)[32]; V = 0;`	`//sub A, borrow`
0b11000	`A = (A << B);` `C = A[31]; V = (B >= 32) ? 0:1;`	`//shift left A`
0b11001	`A = (B << Literal);` `C = B[31]; V = (Literal >= 32) ? 0:1;`	`//shift left B`
0b11010	`A = (B << 1);` `C = B[31]; V = 0;`	`//shift left B`
0b11011	`n = (A - B);` `C = (A - B)[32]; V = 0;`	`//compare`
0b11100	`A = (A >> B);` `C = A[0]; V = (B >= 32) ? 1:0;`	`//shift right A`
0b11101	`A = (B >> Literal);` `C = A[0]; V = (Literal >= 32) ? 1:0;`	`//shift right B`
0b11110	`A = (B >> 1);` `C = A[0]; V = 0;`	`//shift right B`
0b11111	`n = (B - A);` `C = (B - A)[32]; V = 0;`	`//compare`

60091809.101497

WISTRON CORP. EXHIBIT 1031.086

OpdSel	SELECTED OPERANDs	
0b0000aaaaa	**File**	**File**@(**OpdSel**[4:0] | **FileBase**); Allows paged access to any part of the register file.
0b0001aaaaa	**CpuReg**	**File**@{2'b11, **CpuId**, **OpdSel**[4:0]}; Allows direct access to Cpu specific registers.
0b001XXXXXX	**reserved**	Reserved for future expansion.
0b0100000XX	**CpuStatus**	0b00000000000000BHD00000000000000CC This is a read-only register providing information about the Cpu executing (**OpdSel**[1:0]) cycles after the current cycle. "CC" represents a value indicating the Cpu. Currently, only **CpuId** values of 0, 1 and 2 are returned. "H" represents the current state of **Hlt**, "D" indicates **DbgMd** and "B" indicates **BigMd**. Writing this register has no effect.
0b0100001XX	**reserved**	Reserved for future expansion.
0b0100010XX	**Pc**	0x0000AAAA Writing to this address causes the program control logic to use **AluOut** as the new **Pc** value in the event of a **Jmp**, **Jcc** or **Jsr** instruction for the Cpu executing during the current cycle. If the current instruction is **Nxt**, **Map**, or **Rts**, the register write has no effect. Reading this register returns the value in **Pc** for the Cpu executing (**OpdSel**[1:0]) cycles after the current cycle.
0b0100011XX	**DbgAddr**	0xD000AAAA Writing to this register alters the contents of the debug address register (**DbgAddr**) for the Cpu executing (**OpdSel**[1:0]) cycles after the current cycle. **DbgAddr** provides the fetch address for the control-store when **DbgMd** has been selected and the Cpu is executing. **DbgAddr** is also used as the control-store address when performing a **WrWcs@DbgAddr** or **RdWcs@DbgAddr** operation. "D" represents bit 31 of the register. It is a general purpose flag that is used for event indication during simulation. Reading this register returns a value of 0x00000000.
0b01001XXXX	**reserved**	Reserved for future expansion.

0b010100000 **RamAddr** {0b1CCC, 0x000, 0b1, AAAA}

RamAddr = **AluOut**[15] ? **AluOut** : (**AluOut** | **RamBase**);
PrevCC = **AluOut**[31] ? CCC : **AluCC**;

A read/write register. When reading this register, the Alu condition codes from the previous instruction are returned together with **RamAddr**.

bit	name	description
31		Always 1.
30	**PrevC**	Previous Alu Carry.
29	**PrevV**	Previous Alu Overflow.
28	**PrevZ**	Previous Alu Zero.
27:16		Always 0.
15		Always 1.
14:0	**RamAddr**	Contents of last Sram address used.

When writing this register, if alu_out[31] is set, the previous condition codes will be overwritten with bits 30:28 of **AluOut**. If **AluOut**[15] is set, bits 14:0 will be written to the **RamAddr**. If **AluOut** [15] is not set, bits 14:0 will be ored with the contents of the **RamBase** and written to the **RamAddr**.

WISTRON CORP. EXHIBIT 1031.087

60061801 "60087 6009

OpdSel	SELECTED OPERANDs
0b010100001	**AddrRegA** 0x0000AAAA **AddrRegA = AluOut;** A read/write operand which loads **AddrRegA** used to provide the address for read and write operations. When **AddrRegA**[15] is set, the contents will be presented directly to the ram. When **AddrRegA**[15] is reset, the contents will first be ored with the contents of the **RamBase** register before presentation to the ram. Writing to this register takes priority over Literal loads using **FlgOp**. Reading this register returns the current value of the register.
0b010100010	**AddrRegB** 0x0000AAAA **AddrRegB = AluOut;** A read/write operand which loads **AddrRegB** used to provide the address for read and write operations. When **AddrRegB**[15] is set, the contents will be presented directly to the ram. When **AddrRegB**[15] is reset, the contents will first be ored with the contents of the **RamBase** register before presentation to the ram. Writing to this register takes priority over Literal loads using **FlgOp**. Reading this register returns the current value of the register.
0b010100011	**AddrRegAb** 0x0000AAAA **AddrRegA = AluOut; AddrRegB = AluOut;** A destination only operand which loads **AddrRegB** and **AddrRegA** used to provide the address for read and write operations Writing to this register takes priority over Literal loads using **FlgOp**. Reading this register returns the value 0x00000000.
0b010100100	**RamBase** 0x0000AAAA **RamBase = AluOut;** A read/write register which provides the base address for ram read and write cycles. When **RamAddr[15]** is set, the contents will not be used. When **RamAddr[15]** is reset, the contents will first be ored with the contents of the **RamBase** register before presentation to the ram. Reading this register returns the value for the current Cpu.
0b010100101	**FileBase** 0b00000000000000000000000AAAAAAAAA **FileBase = AluOut;** **FileAddr = OpdSel[8] ? OpdSel:(OpdSel + FileBase);** A read/write register which provides the base address for **file** read and write cycles. When **OpdSel[8]** is set, the contents will not be used and **OpdSel** will be presented directly to the address lines of the file. When **OpdSel**[8] is reset, the contents will first be ored with the contents of the **FileBase** register before presentation to the file. Reading this register returns the value for the current Cpu.
0b010100110	**InstrRegL** 0xIIIIIIII This is a read-only register which returns the contents of **InstrReg**[31:0]. Writing to this register has no effect.
0b010100111	**InstrRegH** 0x00IIIIII This is a read-only register which returns the contents of **InstrReg**[55:32]. Writing to this register has no effect.

09781097 . 101497

WISTRON CORP. EXHIBIT 1031.088

OpdSel	SELECTED OPERANDs	
0b010101000	**Minus1**	0xffffffff This is a read-only register which supplies a value 0xffffffff.. Writing to this register has no effect.
0b010101001	**FreeTime**	A free-running timer with a resolution of 1.00 microseconds and a maximum count of 71 minutes. This timer is cleared during reset.
0b010101010	**LiteralL**	**Instr**[15:0] A read-only register. Writing to this register has no effect
0b010101011	**LiteralH**	**Instr**[15:0] < < 16; A read-only register. Writing to this register has no effect

0b010101100 **MacData** - Writing to this address loads the **AluOut** data into the **MacData** register for use during Mac operations. The Mac operation, resulting from writing to the **MacOp** register, determines the definition of the **MacData** register contents as follows.

MacOp	MacData definition
Mstop	0bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX **MacData** is not used for the **StopM** operation.
WrMcfg	**hrstl**, rsvd, rsvd, **ercen, fulld, hrstl, hugen, nopre, paden, prtyl, xdl10, ipgr1[6:0], ipgr2[6:0], ipgt[6:0].** Loads the **MacCfg** register with the contents of the **MacData** register. Refer to LSI Logic's *Ethernet-110 Core Technical Manual* for detailed definitions of these bits.
WrMrng	0bXXXXXXXXXXXXXXXXXXXXXSSSSSSSSSSS Loads **seed**[10:0] into the Mac's random number generator.
RdPhy	0bXXXXRRRRXXXXPPPPXXXXXXXXXXXXXXXX Reads register[R] of phy[P].
WrPhy	0bXXXXRRRRXXXXPPPPDDDDDDDDDDDDDDDD Writes register[R] of phy[P] with **MacData**[15:0].

Reading this register returns **prsd[15:0]** of **Mac0** which contains phy status data returned to the Mac at the completion of a **RdPhy** command. This data is invalid while **MacBsy** is asserted as a result of a **RdPhy** command. Refer to the appropriate phy technical manual for a definition of the phy register contents.

60061809.101497

WISTRON CORP. EXHIBIT 1031.089

OpdSel	SELECTED OPERANDs

0b010101101 **MacOp** - A write only register. Writing to this address loads the **MacSel** register and staRts execution of the specified operation as follows.

AluOut	description
0xXXXXX0XM	**Mstop** - Halts execution of a **MacOp** for **Mac[M]**. The user must wait for **MacBsy** to be deasserted before issuing another command or changing the contents of **MacData**.
0xXXXXX1XM	**WrMcfg** - Writes the contents of **MacData** to the **MacCfg** register of **Mac[M]**. The user must wait for **MacBsy** to be deasserted before issuing another command or changing the contents of **MacData**.
0xXXXXX2XM	**WrMrng** - Writes the contents of **MacData** to the **seed** register of **Mac**[M]. The user must wait for **MacBsy** to be deasserted before issuing another command or changing the contents of **MacData**.
0xXXXXX3XM	**RdPhy** - Reads the contents of reg[R] for phy[P] on the MII management bus of Mac[M]. The contents may be read from **MacData** after **MacBsy** has been de-asserted.
0xXXXXX4XM	**WrPhy** - Writes the contents of **MacData**[15:0] to the reg[R] of phy[P] on the MII management bus of **Mac[M]**. The user must wait for **MacBsy** to be deasserted before issuing another command or changing the contents of **MacData**.
0xXXXXX8XM	**WrAddrAL** - Writes the contents of **MacData**[15:0] to **MacAddrA[15:0]** for Mac[M].
0xXXXXX9XM	**WrAddrAH** - Writes the contents of **MacData**[11:0] to **MacAddrA[47:16]** for Mac[M].
0xXXXXXaXM	**WrAddrBL** - Writes the contents of **MacData**[15:0] to **MacAddrB[15:0]** for Mac[M].
0xXXXXXbXM	**WrAddrBH** - Writes the contents of **MacData**[11:0] to **MacAddrB[47:16]** for **Mac[M]**.

0b010101110 **ChCmd** A write-only register.

bit	name	description
31:11	reserved	Data written to these bits is ignored.
10:8	**command**	0 - Stops execution of the current operation and clears the corresponding event flag. 1 - Transfer data from ExtMem to ExtMem. 2 - Transfer data from Pci to ExtMem. 3 - Transfer data from ExtMem to Pci. 4 - Transfer data from Sram to ExtMem. 5 - Transfer data from ExtMem to Sram. 6 - Transfer data from Pci to Sram. 7 - Transfer data from Sram to Pci.
07:05	reserved	Data written to these bits is ignored.
04:00	**ChId**	Provides the channel number for the channel command.

0b010101110 **ChEvnt** A read-only register.

bit	name	description
31:00	**ChDn**	Each bit represents the done flag for the respective dma channel. These bits are set by a dma sequencer upon completion of the channel command. Cleared when the processor writes 0 to the corresponding **ChCmd** register.

0b010101111 **GenEvnt** A read-only register.

bit	name	description
31	**PciRdEvnt**	Indicates that a PCI initiator is attempting to read a µproc. register.
30	**PciWrEvnt**	Indicates that a PCI initiator has posted a write to a µproc. register.
29	**TimeEvnt**	An event which occurs once every 2.00 milliseconds.
28:00	**reserved**	Reserved for future use.

09061497.101097

WISTRON CORP. EXHIBIT 1031.090

0b010110000 **QCtrl** A write-only register used to select and manipulate a **Q**.

bit	name	description
31:11	reserved	Data written to these bits are ignored.
10:8	**QSz**	Used only during **InitQ** operations to specify the size of the **QBdy** in Dram. 7 – Queue depth is 32K entries (128KB). 6 – Queue depth is 16K entries (64KB). 5 – Queue depth is 8K entries (32KB). 4 – Queue depth is 4K entries (16KB). 3 – Queue depth is 2K entries (8KB). 2 – Queue depth is 1K entries (4KB). 1 – Queue depth is 512 entries (2KB). 0 – Queue depth is 256 entries (1KB).
7:5	**QOp**	Specifies the queue operation to perform. 7 – **DblQ** Disables all queues. 6 – **EnQ** Enables all queues. 5 – **RdBdy** Increments the **QBdyRdPtr** and increments the **QTlWrPtr**. 4 – **WrBdy** Decrements the **QBdyWrPtr** and increments the **QHdRdPtr**. 3 – **RdQ** Returns a queue entry in register **QData**.. 2 – **rsvd** Reserved. Not to be used. 1 – **InitQ** Set the queue status to empty and initializes **QSz**. 0 – **SelQ** Selects the **QId** to be utilized during writes to **QData**.
4:0	**QId**	Specifies the queue on which to perform all operations except **DblQ** or **EnQ**.

0b010110001 **QData** A read/write register. Writing this register will result in the data being pushed on to the selected queue. Reading this register fetches queue data popped off during the previous **RdQ** operation.

0b010110010 **reserved** Reserved for future expansion.

0b010110011 **XcvCtrl** A write-only register used to enable and disable Mac transmit and receive sub-channels.

bit	name	description
31:09	reserved	Data written to these bits are ignored.
8	**enable**	When set, indicates to the Mac transmit or receive sequencer that the subchannel contains a transmit or receive descriptor.
07:05	reserved	Data written to these bits is ignored.
04	**RcvCh**	Selects a Mac receive subchannel when set. Selects a Mac transmit subchannel when cleared.
03	reserved	Data written to this bit are ignored.
02	**SubCh**	Selects subchannel B when set or A when reset.
01:00	**MacId**	Provides the Mac number for the subchannel enable bit.

0b010110100 **Lru** 0x0000000A

A read/write operand indicating which of the 16 entries is least recently used. When Reading This register the least recently used entry is returned, after which it is automatically made the most recently used entry. This register should only be read in conjunction with a 'Move' operation of the ALU, else the results are unpredictable. Writing to this register forces the addressed entry to become the least recently used entry.

0b010110101 **Mru** 0x0000000A

A write only operand forcing the addressed entry to become the most recently used entry.

WISTRON CORP. EXHIBIT 1031.091

0b010111000	**QInRdy**	A read-only register comprising **QHd** not full flags for each of the 32 queues.
0b010111001	**QOutRdy**	A read-only register comprising **QTl** not empty flags for each of the 32 queues.
0b010111010	**QEmpty**	A read-only register comprising **QEmpty** flags for each of the 32 queues.
0b010111011	**QFull**	A read-only register comprising **QFull** flags for each of the 32 queues.
0b0101111XX	**reserved**	Reserved for future expansion.
0b0110XXXXX	**Constants**	{0b000, **OpdSel**[4:0]}
0b01110XXXX	**reserved**	Reserved for future expansion.

60061809.101497

OpdSel	SELECTED OPERANDs
0b01111XXXX	Sram OPERATIONS

OpdSel[3]	PostAddrOp
0	nop
1	RamAddr = RamAddr + (OpdSel[1:0]);

OpdSel[2]	transpose_Ctrl
0	don't transpose
1	transpose bytes

OpdSel[1:0]	RamOpdSz
0	quadlet
1	triplet
2	doublet
3	byte

RAM READ ATTRIBUTES				SOURCE OPERAND			
endian mode	trans-pose	byte offs	Sram data	sz=Q	sz=T	sz=D	sz=B
little	0	0	abcd	abcd	0bcd	00cd	000d
little	0	1	abcX	trap	0abc	00bc	000c
little	0	2	abXX	trap	trap	00ab	000b
little	0	3	aXXX	trap	trap	trap	000a
little	1	0	abcd	dcba	0dcb	00dc	000d
little	1	1	abcX	trap	0cba	00cb	000c
little	1	2	abXX	trap	trap	00ba	000b
little	1	3	aXXX	trap	trap	trap	000a
BIG	0	0	abcd	abcd	0abc	00ab	000a
BIG	0	1	Xbcd	trap	0bcd	00bc	000b
BIG	0	2	XXcd	trap	trap	00cd	000c
BIG	0	3	XXXd	trap	trap	trap	000d
BIG	1	0	abcd	dcba	0cba	00ba	000a
BIG	1	1	Xbcd	trap	0dcb	00cb	000b
BIG	1	2	XXcd	trap	trap	00dc	000c
BIG	1	3	XXXd	trap	trap	trap	000d

RAM WRITE ATTRIBUTES				SOURCE OPERAND			
endian mode	trans-pose	Opd size	Alu out	OF=0	OF=1	OF=2	OF=3
little	0	Q	abcd	abcd	trap	trap	trap
little	0	T	Xbcd	-bcd	bcd-	trap	trap
little	0	D	XXcd	--cd	-cd-	cd--	trap
little	0	B	XXXd	---d	--d-	-d--	d---
little	1	Q	abcd	dcba	trap	trap	trap
little	1	T	Xbcd	-dcb	dcb-	trap	trap
little	1	D	XXcd	--dc	-dc-	dc--	trap
little	1	B	XXXd	---d	--d-	-d--	d---
big	0	Q	abcd	abcd	trap	trap	trap
big	0	T	Xbcd	bcd-	-bcd	trap	trap
big	0	D	XXcd	cd--	-cd-	--cd	trap
big	0	B	XXXd	d---	-d--	--d-	---d
big	1	Q	abcd	dcba	trap	trap	trap
big	1	T	Xbcd	dcb-	-dcb	trap	trap
big	1	D	XXcd	dc--	-dc-	--dc	trap
big	1	B	XXXd	d---	-d--	--d-	---d

0b1aaaaaaaa **File** File@OpdSel[8:0];
Allows direct, non-paged, access to the top half of the register file.

60061801 . 101497

TstSel	SELECTED TEST	
0bX00XXXXX	**Tst** = **TstSel**[7] ^ **AluOut**[**TstSel**[4:0]]	//Alu bit
0bX0100000	**Tst** = **TstSel**[7] ^ **C**	//carry
0bX0100001	**Tst** = **TstSel**[7] ^ **V**	//error
0bX0100010	**Tst** = **TstSel**[7] ^ **Z**	//zero
0bX0100011	**Tst** = **TstSel**[7] ^ (**Z** | ~**C**)	//less or equal
0bX0100100	**Tst** = **TstSel**[7] ^ **PrevC**	//previous carry
0bX0100101	**Tst** = **TstSel**[7] ^ **PrevV**	//previous error
0bX0100110	**Tst** = **TstSel**[7] ^ **PrevZ**	//previous zero
0bX0100111	**Tst** = **TstSel**[7] ^ (**PrevZ** & **Z**)	//64b zero
0bX0101000	**Tst** = **TstSel**[7] ^ **QOpDn**	//queue op okay
0bX0101001	**Tst** = reserved	
0bX010101X	**Tst** = reserved	
0bX01011XX	**Tst** = reserved	
0bX0110XXX	**Tst** = **TstSel**[7] ^ **Lock[TstSel[2:0]]** **Lock(TstSel[2:0])** = 1;	//tests the current value of //the Lock then set it.
0bX0111XXX	**Tst** = **TstSel**[7] ^ **Lock[TstSel[2:0]]**	//tests the value of Lock.
0bX01XXXXX	**Tst** = reserved	
0bX1XXXXXX	**Tst** = reserved	

FlgSel	FLAG OPERATION	
0b00000000	No operation.	
0b00000001	**SelfRst**	Forces a self reset for the entire chip excluding the PCI configuration registers
0b00000010	**SelBigEnd**	Selects big-endian mode for ram accesses for the current Cpu.
0b00000011	**SelLitEnd**	Selects little-endian mode for ram accesses for the current Cpu.
0b00000100	**DblMap**	Disable instruction re-mapping for the current Cpu.
0b00000101	**EnbMap**	Enable instruction re-mapping for the current Cpu.
0b0000011X	reserved	
0b00001XXX	reserved	
0b00010XXX	**ClrLck**	**Lock[FlgSel[2:0]]** = 0; Clears the semaphore register bit for the current Cpu only.
0b00011XXX	reserved	

60061809 . 101497

WISTRON CORP. EXHIBIT 1031.094

FlgSel	FLAG OPERATION

0b0010XXXX **AddrOp**

FlgSel[3:2]	AddrSelect
0	RamAddr = Literal[15] ? Literal : (Literal | RamBase);
1	RamAddr = AddrRegA[15] ? AddrRegA : (AddrRegA | RamBase);
2	RamAddr = AddrRegB[15] ? AddrRegB : (AddrRegB | RamBase);

3	if (OpdA = = RamAddr)
	RamAddr = AluOut[15] ? AluOut : (AluOut | RamBase);
	else if (OpdA = = ram)
	RamAddr = AddrRegB[15] ? AddrRegB : (AddrRegB | RamBase);
	else
	RamAddr = AddrRegA[15] ? AddrRegA : (AddrRegA | RamBase);

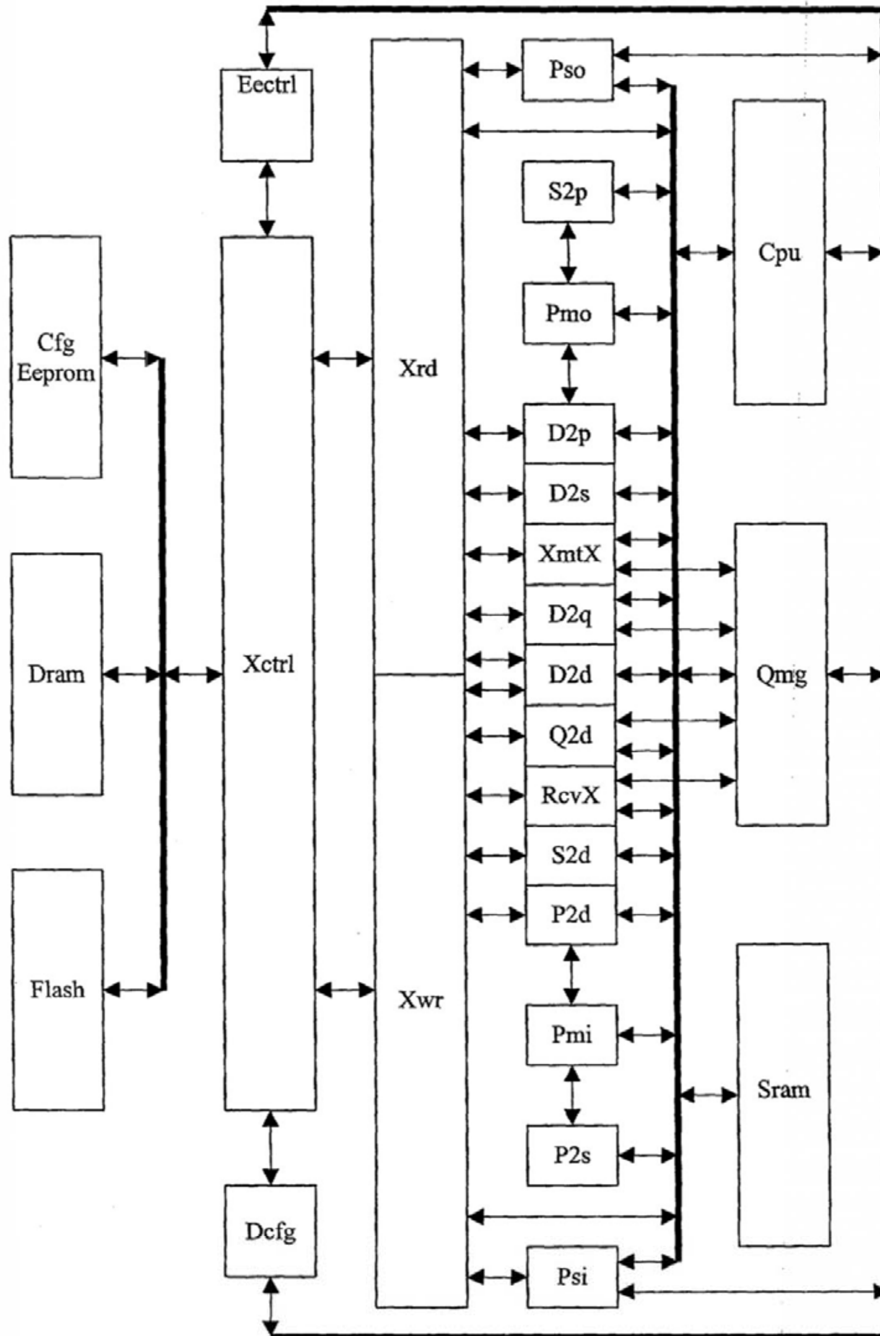
FlgSel[1:0]	addr reg load
0	nop
1	AddrRegA = Literal;
2	AddrRegB = Literal;
3	AddrRegA = Literal; AddrRegB = Literal;

note: When specifying the same register for both the load and select fields, the current value of the register, before it is loaded with the new value, will be used for the ram address.

0b0011XXXX	reserved	
0b01000000	**WrWcsL@Dbg**	Causes the bits [31:0] of the control-store at address **DbgAddr** to be written with the current **AluOut** data.
0b01000001	**WrWcsH@Dbg**	Causes the bits [63:32] of the control-store at address **DbgAddr** to be written with the current **AluOut** data then increments **DbgAddr**.
0b01000010	**RdWcsL@Dbg**	Causes the bits [31:0] of the control-store at address **DbgAddr** to be moved to file address 0x1ff.
0b01000011	**RdWcsH@Dbg**	Causes the bits [63:32] of the control-store at address **DbgAddr** to be moved to file address 0x1ff then increments **DbgAddr**.
0b01000100	reserved	
0b010001XX	**Step**	Allows the Cpu (**FlgSel**[1:0]) cycles after the current cycle to execute a single instruction. There is no effect if the Cpu is not halted. An offset of 0 is not allowed.
0b010010XX	**PcMd**	Selects the **Pc** as the address source for the control-store during instruction fetches for the Cpu (**FlgSel**[1:0]) cycles after the current cycle.
0b010011XX	**DbgMd**	Selects the **DbgAddr** address register as the address source for the control-store during instruction fetches for the Cpu (**FlgSel**[1:0]) cycles after the current cycle.
0b010100XX	**Hlt**	Halts the Cpu (**FlgSel**[1:0]) cycles after the current cycle.
0b010101XX	**Run**	Clears Halt for the Cpu (**FlgSel**[1:0]) cycles after the current cycle.
0b01011XXX	reserved	
0b011XXXXX	reserved	
0b1XXXXXXX	reserved	

09791101.081809

DATA FLOW



09619809 . 101497

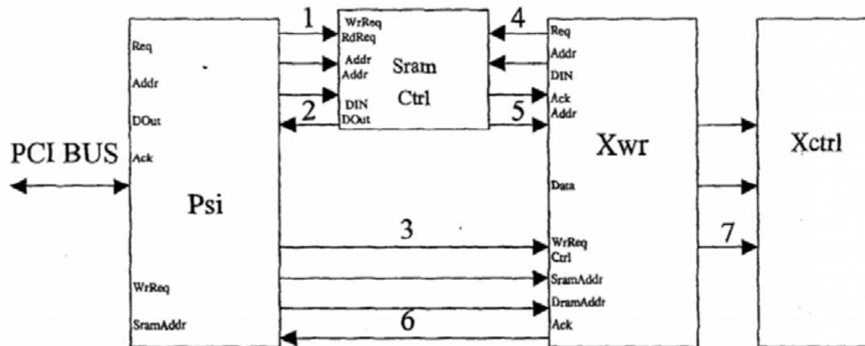
WISTRON CORP. EXHIBIT 1031.096

SRAM CONTROL SEQUENCER (SramCtrl)

Sram is the nexus for data movement within the INIC. A hierarchy of sequencers, working in concert, accomplish the movement of data between dram, Sram, Cpu, ethernet and the Pci bus. Slave sequencers, provided with stimulus from master sequencers, request data movement operations by way of the Sram, Pci bus, Dram and Flash. The slave sequencers prioritize, service and acknowledge the requests

The preceding block diagram shows all of the master and slave sequencers of the INIC product. Request information such as r/w, address, size, endian and alignment are represented by each request line. Acknowledge information to master sequencers include only the size of the transfer being acknowledged.

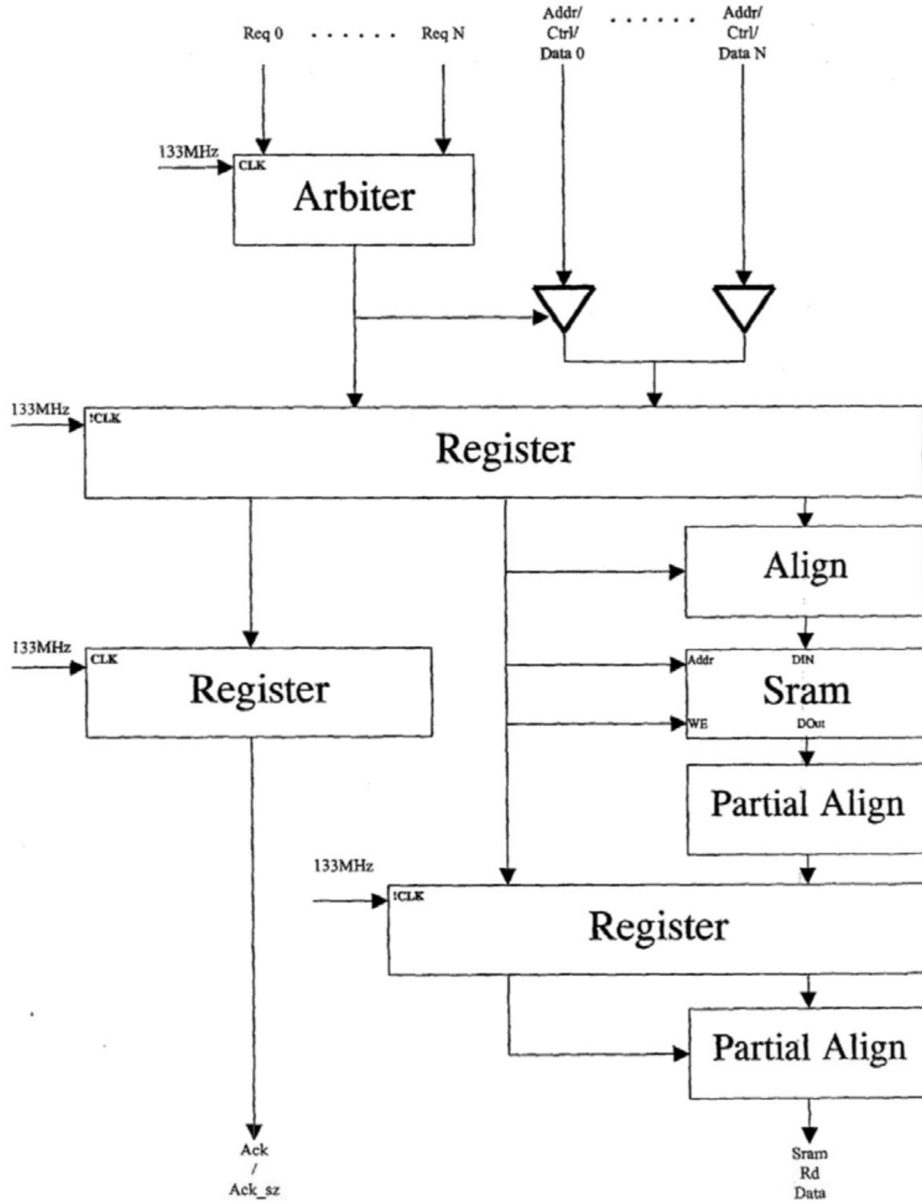
The following block diagram illustrates how data movement is accomplished for a Pci slave write to Dram. Note that the **Psi** (Pci slave in) module functions as both a master sequencer. **Psi** sends a write request to the **SramCtrl** module. **Psi** requests **Xwr** to move data from Sram to dram. **Xwr** subsequently sends a read request to the **SramCtrl** module then writes the data to the dram via the **Xctrl** module. As each piece of data is moved from the Sram to **Xwr**, **Xwr** sends an acknowledge to the **Psi** module.



09018009.101497

WISTRON CORP. EXHIBIT 1031.097

SRAM CONTROL SEQUENCER (SramCtrl)



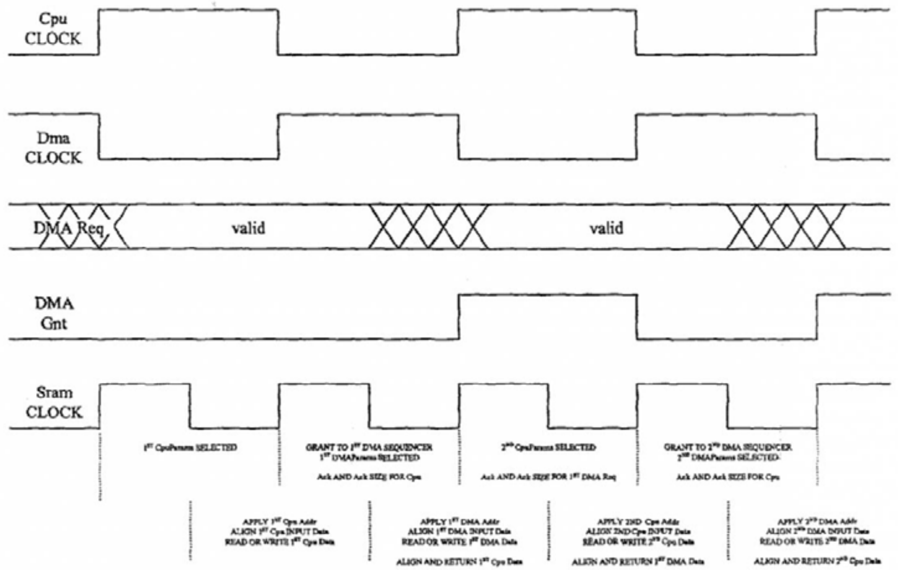
60061809.101497

WISTRON CORP. EXHIBIT 1031.098

The Sram control sequencer services requests to store to, or retrieve data from an Sram organized as 1024 locations by 128 bits (16KB). The sequencer operates at a frequency of 133MHz, allowing both a Cpu access and a dma access to occur during a standard 66MHz Cpu cycle. One 133MHz cycle is reserved for Cpu accesses during each 66MHz cycle while the remaining 133MHz cycle is reserved for dma accesses on a prioritized basis.

The preceding block diagram shows the major functions of the Sram control sequencer. A slave sequencer begins by asserting a request along with r/w, ram address, endian, data path size, data path alignment and request size. **SramCtrl** prioritizes the requests. The request parameters are then selected by a multiplexer which feeds the parameters to the Sram via a register. The requestor provides the Sram address which when coupled with the other parameters controls the input and output alignment. Sram outputs are fed to the output aligner via a register. Requests are acknowledged in parallel with the returned data.

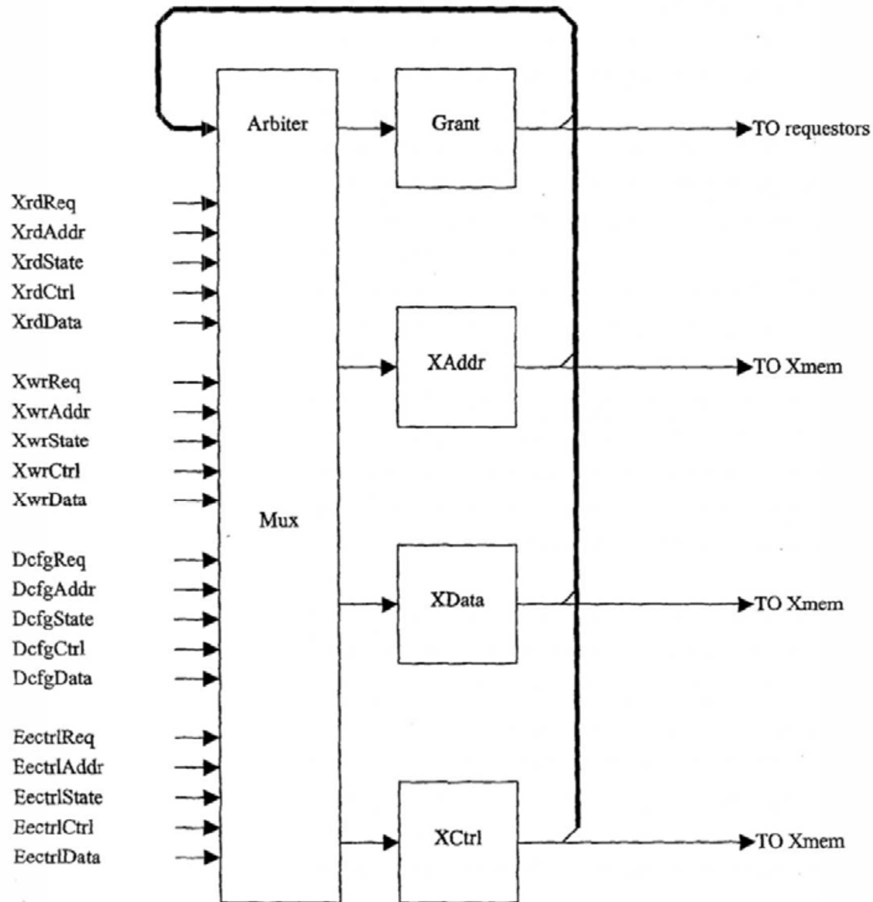
Following is a timing diagram depicting two ram accesses during a single 66MHz clock cycle.



WISTRON CORP. EXHIBIT 1031.099

EXTERNAL MEMORY CONTROL (Xctrl)

Xctrl provides the facility whereby **Xwr**, **Xrd**, **Dcfg** and **Eectrl** access external **Flash** and **Dram**. **Xctrl** includes an arbiter, i/o registers, data multiplexers, address multiplexers and control multiplexers. Ownership of the external memory interface is requested by each block and granted to each of the requesters by the arbiter function. Once ownership has been granted the multiplexers select the address, data and control signals from owner, allowing access to external memory.



09061809.101497

WISTRON CORP. EXHIBIT 1031.100

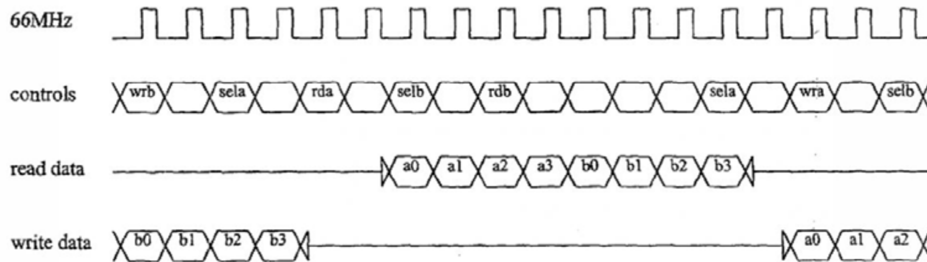
EXTERNAL MEMORY READ SEQUENCER (Xrd)

The **Xrd** sequencer acts only as a slave sequencer. Servicing requests issued by master sequencers, the **Xrd** sequencer moves data from external sdram or flash to the **Sram**, via the **Xctrl** module, in blocks of 32 bytes or less. The nature of the sdram requires fixed burst sizes for each of it's internal banks with ras precharge intervals between each access. By selecting a burst size of 32 bytes for sdram reads and interleaving bank accesses on a 16 byte boundary, we can ensure that the ras precharge interval for the first bank is satisfied before burst completion for the second bank, allowing us to re-instruct the first bank and continue with uninterrupted dram access. Sdrams require a consistent burst size be utilized each and every time the sdram is accessed. For this reason, if an sdram access does not begin or end on a 32 byte boundary, sdram bandwidth will be reduced due to less than 32 bytes of data being transferred during the burst cycle.

The following block diagram depicts the major functional blocks of the **Xrd** sequencer. The first step in servicing a request to move data from sdram to Sram is the prioritization of the master sequencer requests. Next the **Xrd** sequencer takes a snapshot of the dram read address and applies configuration information to determine the correct bank, row and column address to apply. Once sufficient data has been read, the **Xrd** sequencer issues a write request to the **SramCtrl** sequencer which in turn sends an acknowledge to the **Xrd** sequencer. The **Xrd** sequencer passes the acknowledge along to the level two master with a size code indicating how much data was written during the Sram cycle allowing the update of pointers and counters. The dram read and Sram write cycles repeat until the original burst request has been completed at which point the **Xrd** sequencer prioritizes any remaining requests in preparation for the next burst cycle.

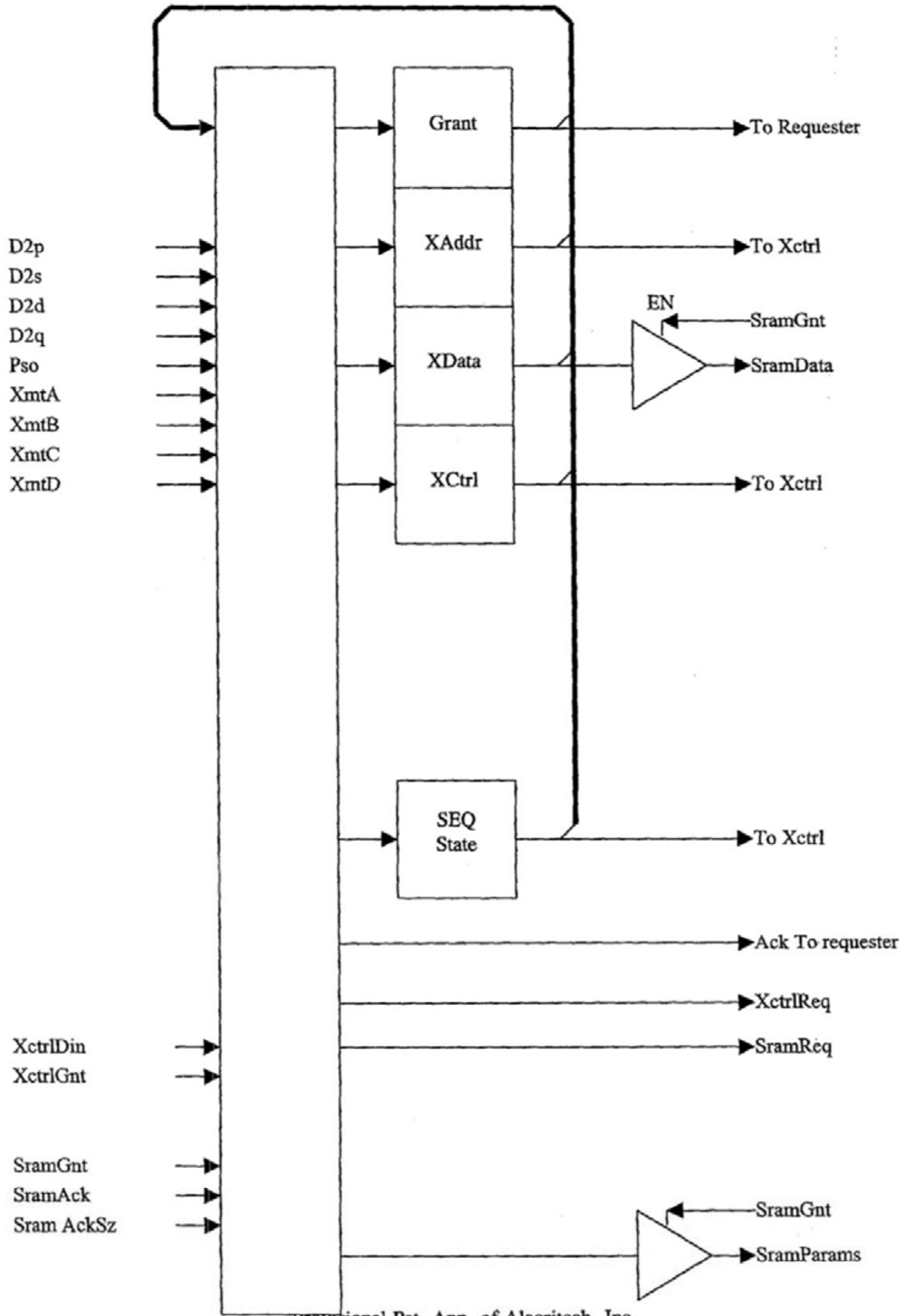
Contiguous dram burst cycles are not guaranteed to the **Xrd** sequencer as an algorithm is implemented which ensures highest priority to refresh cycles followed by flash accesses, dram writes then dram reads.

Following is a timing diagram illustrating how data is read from sdram. The dram has been configured for a burst of four with a latency of two clock cycles. Bank A is first selected/activated followed by a read command two clock cycles later. The bank select/activate for bank B is next issued as read data begins returning two clocks after the read command was issued to bank A. Two clock cycles before we need to receive data from bank B we issue the read command. Once all 16 bytes have been received from bank A we begin receiving data from bank B.



WISTRON CORP. EXHIBIT 1031.101

EXTERNAL MEMORY READ SEQUENCER (Xrd)



60018009.101497

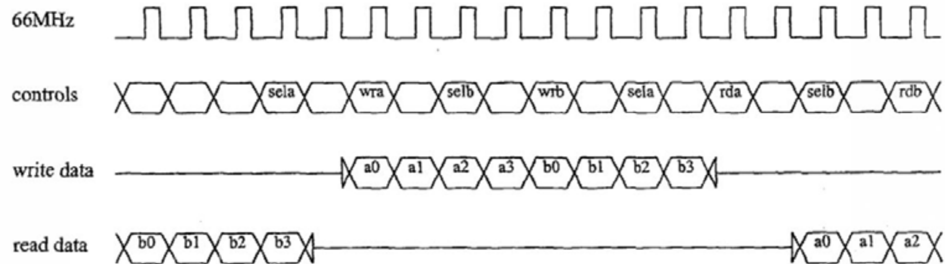
EXTERNAL MEMORY WRITE SEQUENCER (Xwr)

The **Xwr** sequencer is a slave sequencer. Servicing requests issued by master sequencers, the **Xwr** sequencer moves data from Sram to the external sdram or flash, via the **Xctrl** module, in blocks of 32 bytes or less while accumulating a checksum of the data moved. The nature of the sdram requires fixed burst sizes for each of it's internal banks with ras precharge intervals between each access. By selecting a burst size of 32 bytes for sdram writes and interleaving bank accesses on a 16 byte boundary, we can ensure that the ras prechage interval for the first bank is satisfied before burst completion for the second bank, allowing us to re-instruct the first bank and continue with uninterrupted dram access. Sdrams require a consistent burst size be utilized each and every time the sdram is accessed. For this reason, if an sdram access does not begin or end on a 32 byte boundary, sdram bandwidth will be reduced due to less than 32 bytes of data being transferred during the burst cycle.

The following block diagram depicts the major functional blocks of the **Xwr** sequencer. The first step in servicing a request to move data from Sram to sdram is the prioritization of the level two master requests. Next the **Xwr** sequencer takes a Snapshot of the dram write address and applies configuration information to determine the correct dram, bank, row and column address to apply. The **Xwr** sequencer immediately issues a read command to the Sram to which the Sram responds with both data and an acknowledge. The **Xwr** sequencer passes the acknowledge to the level two master along with a size code indicating how much data was read during the Sram cycle allowing the update of pointers and counters. Once sufficient data has been read from Sram, the **Xwr** sequencer issues a write command to the dram starting the burst cycle and computing a checksum as the data flies by. The Sram read cycle repeats until the original burst request has been completed at which point the **Xwr** sequencer prioritizes any remaining requests in preparation for the next burst cycle.

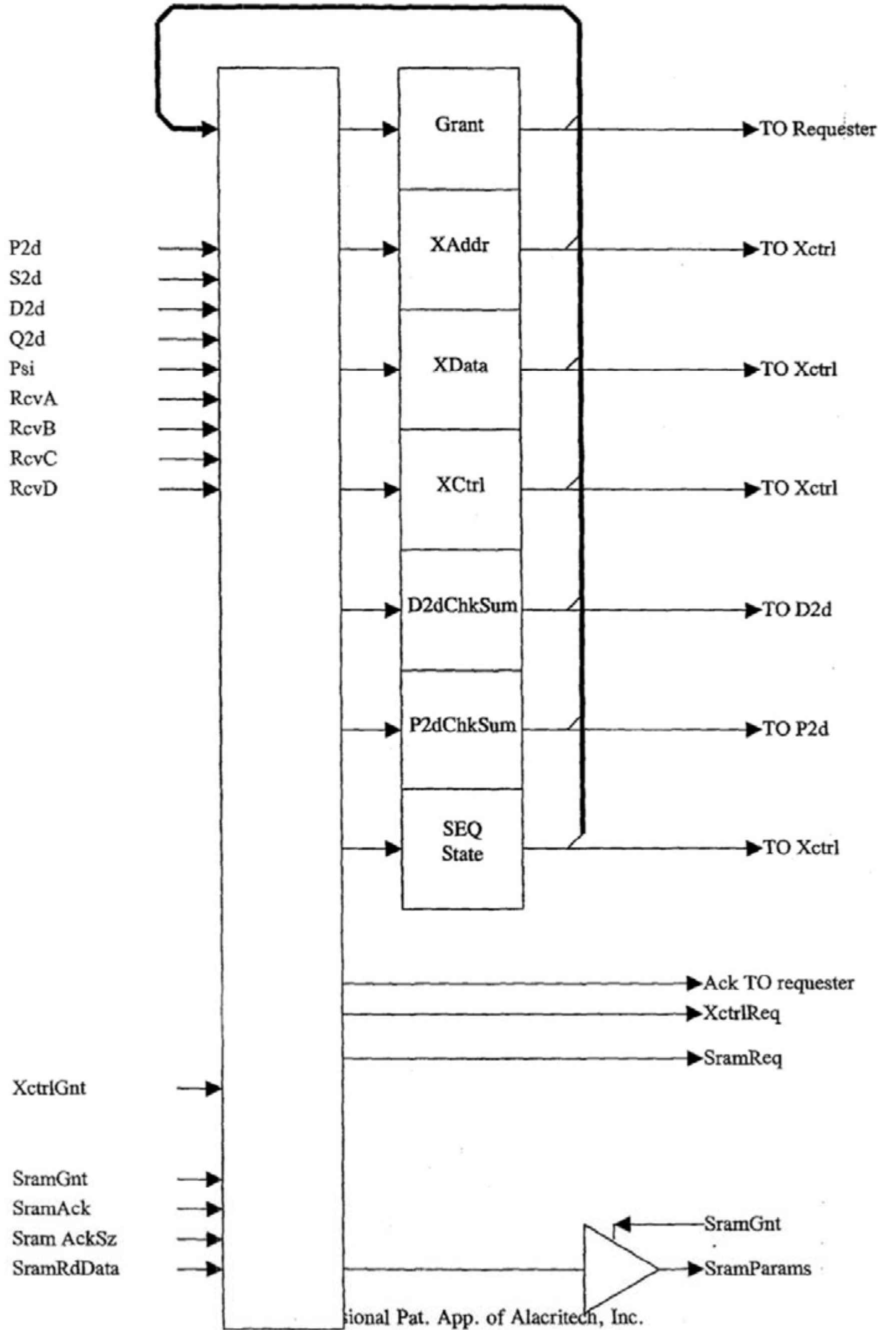
Contiguous dram burst cycles are not guaranteed to the **Xwr** sequencer as an algorithm is implemented which ensures highest priority to refresh cycles followed by flash accesses then dram writes.

Following is a timing diagram illustrating how data is written to sdram. The dram has been configured for a burst of four with a latency of two clock cycles. Bank A is first selected/activated followed by a write command two clock cycles later. The bank select/activate for bank B is next issued in preparation for issuing the second write command. As soon as the first 16 byte burst to bank A completes we issue the write command for bank B and begin supplying data.



WISTRON CORP. EXHIBIT 1031.103

EXTERNAL MEMORY WRITE SEQUENCER (Xwr)

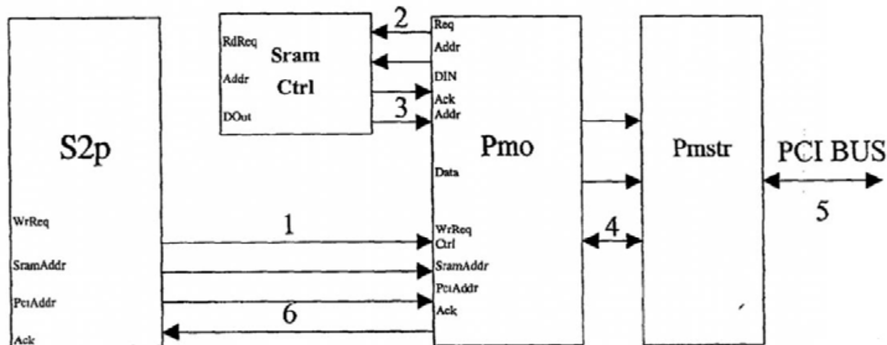


WISTRON CORP. EXHIBIT 1031.104

PCI MASTER-OUT SEQUENCER (Pmo)

The **Pmo** sequencer acts only as a slave sequencer. Servicing requests issued by master sequencers, the **Pmo** sequencer moves data from an Sram based fifo to a Pci target, via the **PciMstrIO** module, in bursts of up to 256 bytes. The nature of the PCI bus dictates the use of the write line command to ensure optimal system performance. The write line command requires that the **Pmo** sequencer be capable of transferring a whole multiple (1X, 2X, 3X, ...) of cache lines of which the size is set through the Pci configuration registers. To accomplish this end, **Pmo** will automatically perform partial bursts until it has aligned the transfers on a cache line boundary at which time it will begin usage of the write line command. The Sram fifo depth, of 256 bytes, has been chosen in order to allow **Pmo** to accommodate cache line sizes up to 128 bytes. Provided the cache line size is less than 128 bytes, **Pmo** will perform multiple, contiguous cache line bursts until it has exhausted the supply of data.

Pmo receives requests from two separate sources; the dram to Pci (**D2p**) module and the Sram to Pci (**S2p**) module. An operation first begins with prioritization of the requests where the **S2p** module is given highest priority. Next, the **Pmo** module takes a Snapshot of the Sram fifo address and uses this to generate read requests for the **SramCtrl** sequencer. The **Pmo** module then proceeds to arbitrate for ownership of the Pci bus via the **PciMstrIO** module. Once the **Pmo** holding registers have sufficient data and Pci bus mastership has been granted, the **Pmo** module begins transferring data to the Pci target. For each successful transfer, **Pmo** sends an acknowledge and encoded size to the master sequencer, allow it to update it's internal pointers, counters and status. Once the Pci burst transaction has terminated, **Pmo** parks on the Pci bus unless another initiator has requested ownership. **Pmo** again prioritizes the incoming requests and repeats the process.

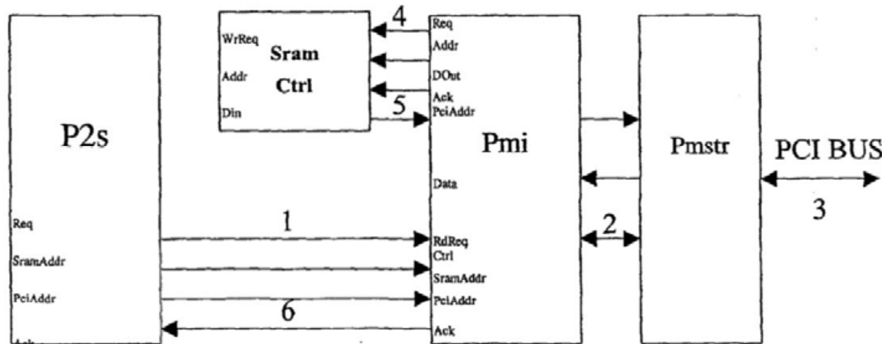


WISTRON CORP. EXHIBIT 1031.105

PCI MASTER-IN SEQUENCER (Pmi)

The **Pmi** sequencer acts only as a slave sequencer. Servicing requests issued by master sequencers, the **Pmi** sequencer moves data from a Pci target to an Sram based fifo, via the **PciMstrIO** module, in bursts of up to 256 bytes. The nature of the PCI bus dictates the use of the read multiple command to ensure optimal system performance. The read multiple command requires that the **Pmi** sequencer be capable of transferring a cache line or more of data. To accomplish this end, **Pmi** will automatically perform partial cache line bursts until it has aligned the transfers on a cache line boundary at which time it will begin usage of the read multiple command. The Sram fifo depth, of 256 bytes, has been chosen in order to allow **Pmi** to accommodate cache line sizes up to 128 bytes. Provided the cache line size is less than 128 bytes, **Pmi** will perform multiple, contiguous cache line bursts until it has filled the fifo.

Pmi receive requests from two separate sources; the Pci to dram (**P2d**) module and the Pci to Sram (**P2s**) module. An operation first begins with prioritization of the requests where the **P2s** module is given highest priority. The **Pmi** module then proceeds to arbitrate for ownership of the Pci bus via the **PciMstrIO** module. Once the Pci bus mastership has been granted and the **Pmi** holding registers have sufficient data, the **Pmi** module begins transferring data to the Sram fifo. For each successful transfer, **Pmi** sends an acknowledge and encoded size to the master sequencer, allowing it to update it's internal pointers, counters and status. Once the Pci burst transaction has terminated, **Pmi** parks on the Pci bus unless another initiator has requested ownership. **Pmi** again prioritizes the incoming requests and repeats the process.

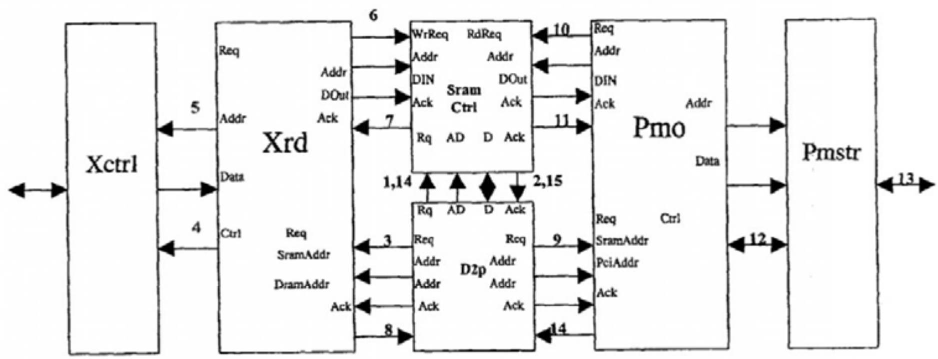


WISTRON CORP. EXHIBIT 1031.106

Dram TO PCI SEQUENCER (D2p)

The **D2p** sequencer acts is a master sequencer. Servicing channel requests issued by the Cpu, the **D2p** sequencer manages movement of data from dram to the Pci bus by issuing requests to both the **Xrd** sequencer and the **Pmo** sequencer. Data transfer is accomplished using an Sram based fifo through which data is staged.

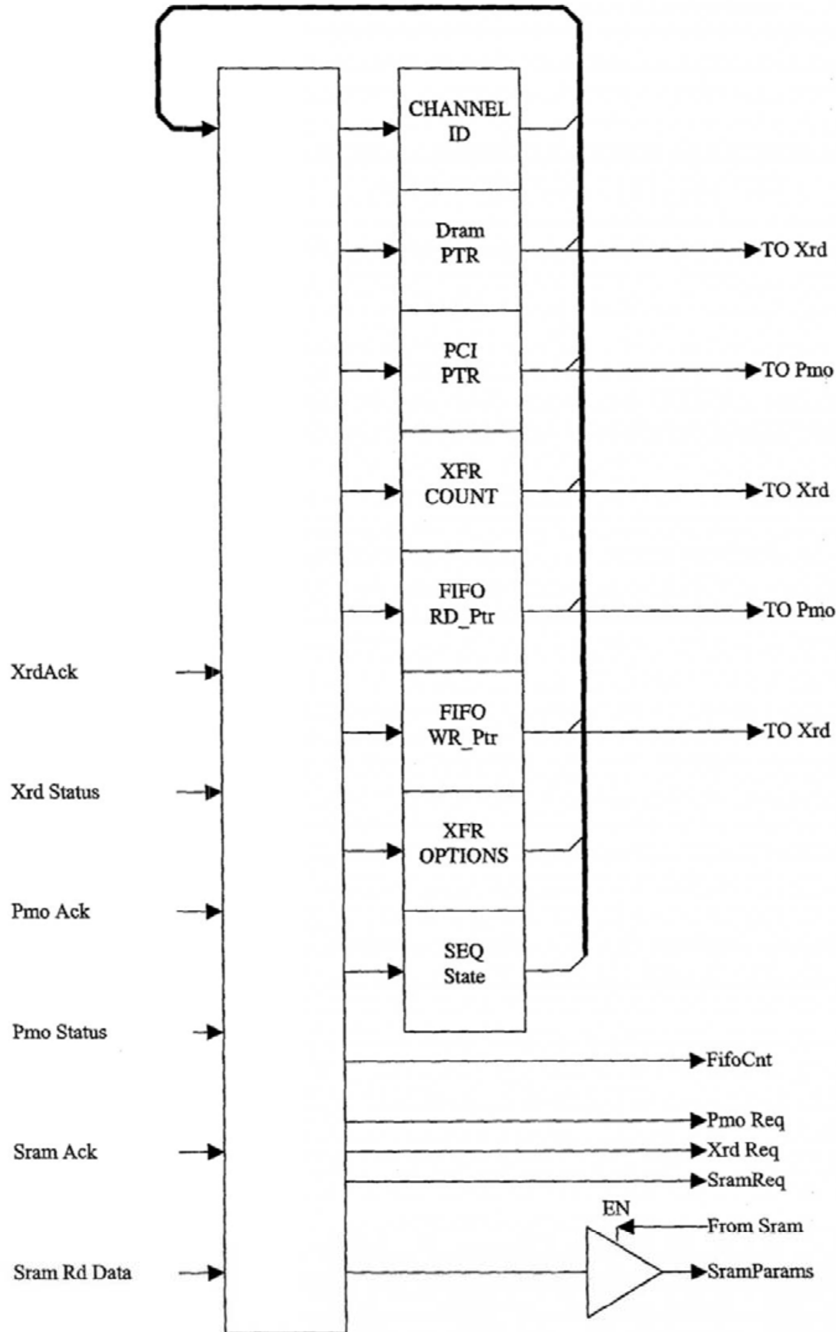
D2p can receive requests from any of the processor's thirty-two dma channels. Once a command request has been detected, **D2p** fetches a dma descriptor from an Sram location dedicated to the requesting channel which includes the dram address, Pci address, Pci endian and request size. **D2p** then issues a request to the **D2s** sequencer causing the Sram based fifo to fill with dram data. Once the fifo contains sufficient data for a Pci transaction, **D2s** issues a request to **Pmo** which in turn moves data from the fifo to a Pci target. The process repeats until the entire request has been satisfied at which time **D2p** writes ending status in to the Sram dma descriptor area and sets the channel done bit associated with that channel. **D2p** then monitors the dma channels for additional requests. Following is an illustration showing the major blocks involved in the movement of data from dram to Pci target.



WISTRON CORP. EXHIBIT 1031.107

60981001.49

Dram TO PCI SEQUENCER (D2p)



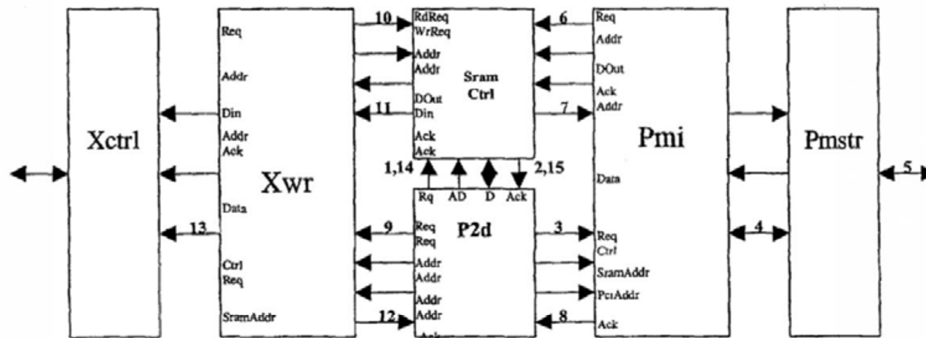
60061809.101497

WISTRON CORP. EXHIBIT 1031.108

PCI TO DRAM SEQUENCER (P2d)

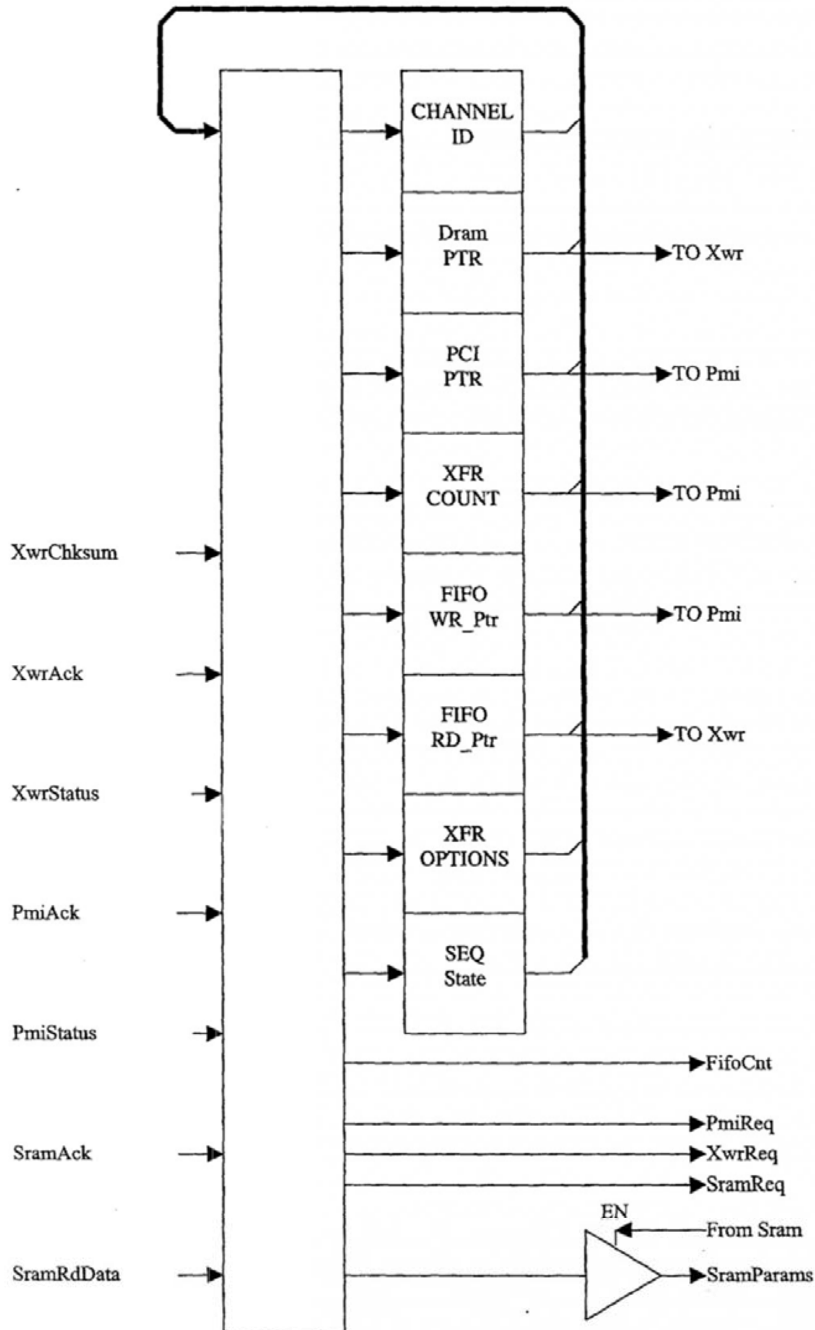
The **P2d** sequencer acts as both a slave sequencer and a master sequencer. Servicing channel requests issued by the Cpu, the **P2d** sequencer manages movement of data from Pci bus to dram by issuing requests to both the **Xwr** sequencer and the **Pmi** sequencer. Data transfer is accomplished using an Sram based fifo through which data is staged.

P2d can receive requests from any of the processor's thirty-two dma channels. Once a command request has been detected, **P2d**, operating as a slave sequencer, fetches a dma descriptor from an Sram location dedicated to the requesting channel which includes the dram address, Pci address, Pci endian and request size. **P2d** then issues a request to **Pmo** which in turn moves data from the Pci target to the Sram fifo. Next, **P2d** issues a request to the **Xwr** sequencer causing the Sram based fifo contents to be written to the dram. The process repeats until the entire request has been satisfied at which time **P2d** writes ending status in to the Sram dma descriptor area and sets the channel done bit associated with that channel. **P2d** then monitors the dma channels for additional requests. Following is an illustration showing the major blocks involved in the movement of data from a Pci target to dram.



WISTRON CORP. EXHIBIT 1031.109

PCI TO DRAM SEQUENCER (P2d)



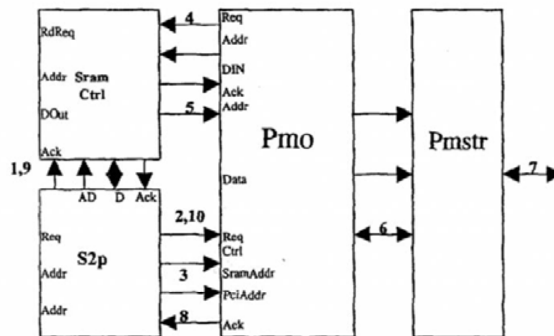
09851809.101497

WISTRON CORP. EXHIBIT 1031.110

SRAM TO PCI SEQUENCER (S2p)

The **S2p** sequencer acts as both a slave sequencer and a master sequencer. Servicing channel requests issued by the Cpu, the **S2p** sequencer manages movement of data from Sram to the Pci bus by issuing requests to the **Pmo** sequencer

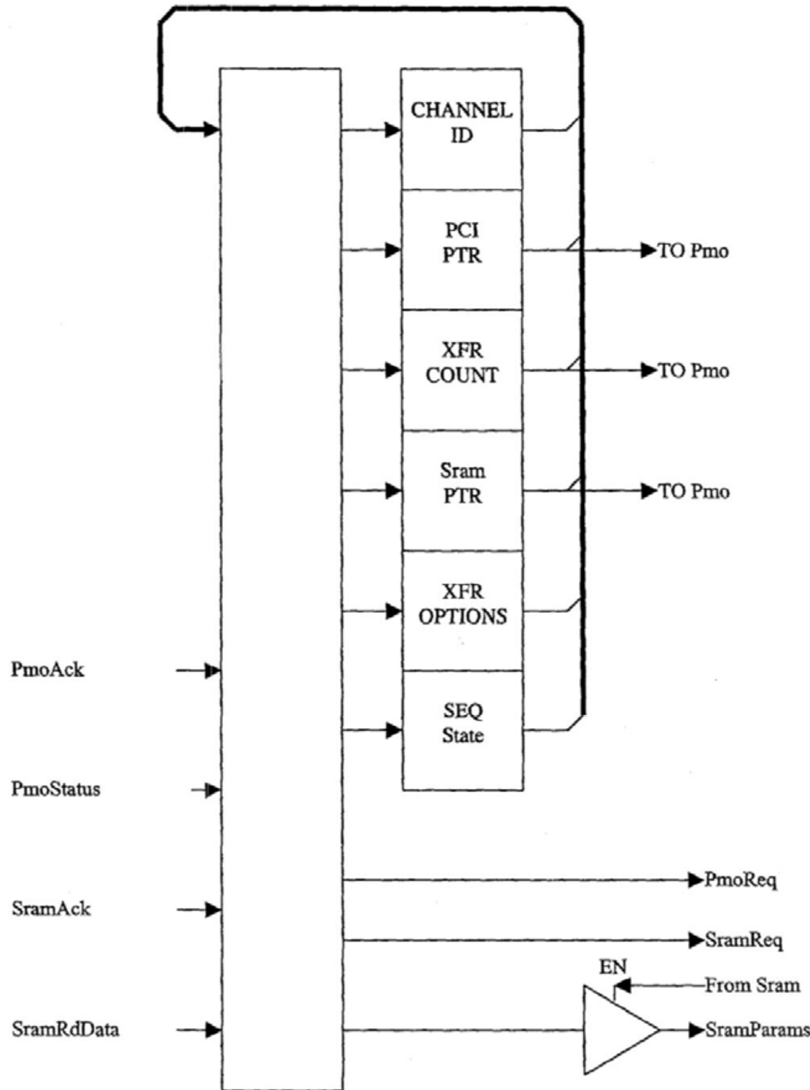
S2p can receive requests from any of the processor's thirty-two dma channels. Once a command request has been detected, **S2p**, operating as a slave sequencer, fetches a dma descriptor from an Sram location dedicated to the requesting channel which includes the Sram address, Pci address, Pci endian and request size. **S2p** then issues a request to **Pmo** which in turn moves data from the Sram to a Pci target. The process repeats until the entire request has been satisfied at which time **S2p** writes ending status in to the Sram dma descriptor area and sets the channel done bit associated with that channel. **S2p** then monitors the dma channels for additional requests. Following is an illustration showing the major blocks involved in the movement of data from Sram to Pci target.



WISTRON CORP. EXHIBIT 1031.111

60091809 . 101497

SRAM TO PCI SEQUENCER (S2p)



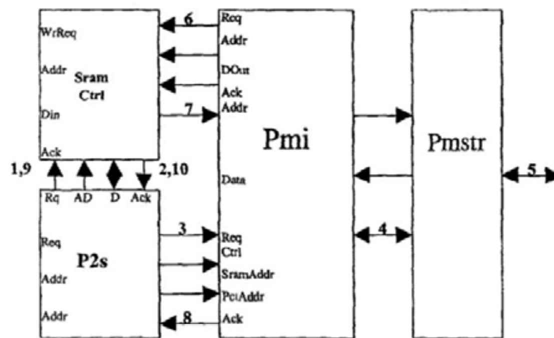
60061809.101497

WISTRON CORP. EXHIBIT 1031.112

PCI TO SRAM SEQUENCER (P2s)

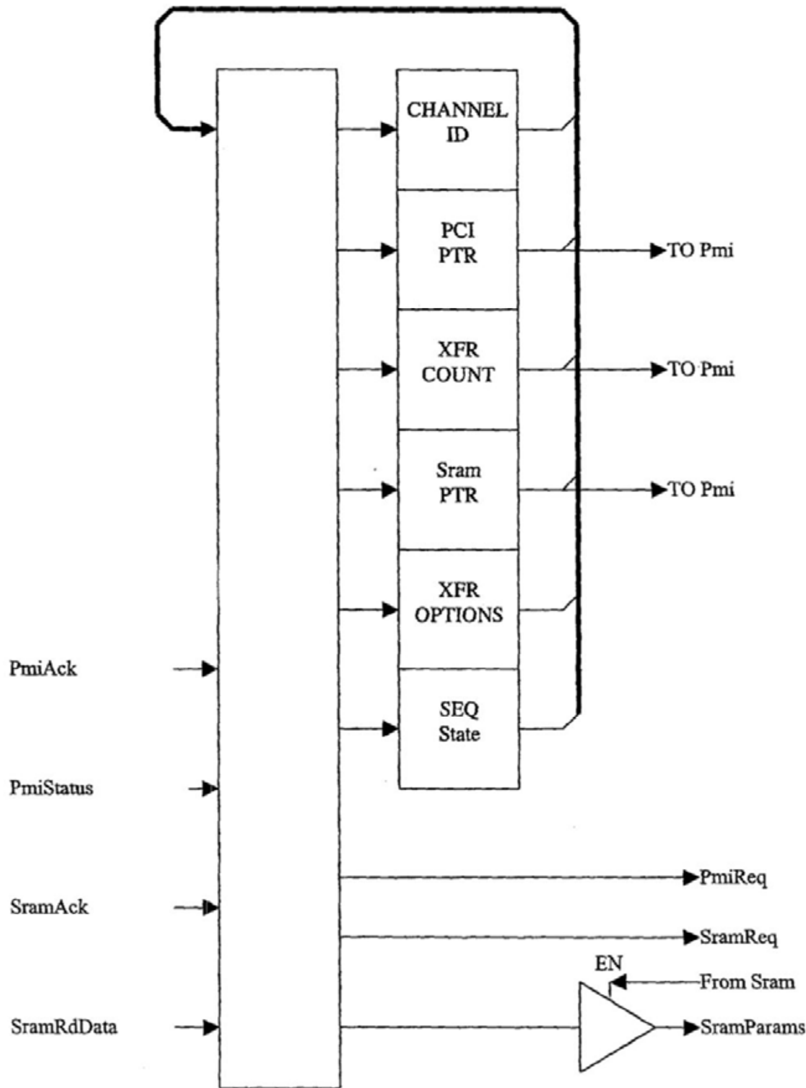
The **P2s** sequencer acts as both a slave sequencer and a master sequencer. Servicing channel requests issued by the Cpu, the **P2s** sequencer manages movement of data from Pci bus to Sram by issuing requests to the **Pmi** sequencer.

P2s can receive requests from any of the processor's thirty-two dma channels. Once a command request has been detected, **P2s**, operating as a slave sequencer, fetches a dma descriptor from an Sram location dedicated to the requesting channel which includes the Sram address, Pci address, Pci endian and request size. **P2s** then issues a request to **Pmo** which in turn moves data from the Pci target to the Sram. The process repeats until the entire request has been satisfied at which time **P2s** writes ending status in to the dma descriptor area of Sram and sets the channel done bit associated with that channel. **P2s** then monitors the dma channels for additional requests. Following is an illustration showing the major blocks involved in the movement of data from a Pci target to dram.



WISTRON CORP. EXHIBIT 1031.113

PCI TO SRAM SEQUENCER (P2s)



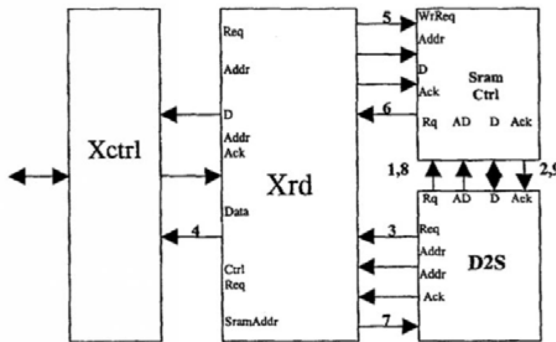
60061809 . 101497

WISTRON CORP. EXHIBIT 1031.114

DRAM TO SRAM SEQUENCER (D2s)

The **D2s** sequencer acts as both a slave sequencer and a master sequencer. Servicing channel requests issued by the Cpu, the **D2s** sequencer manages movement of data from dram to Sram by issuing requests to the **Xrd** sequencer.

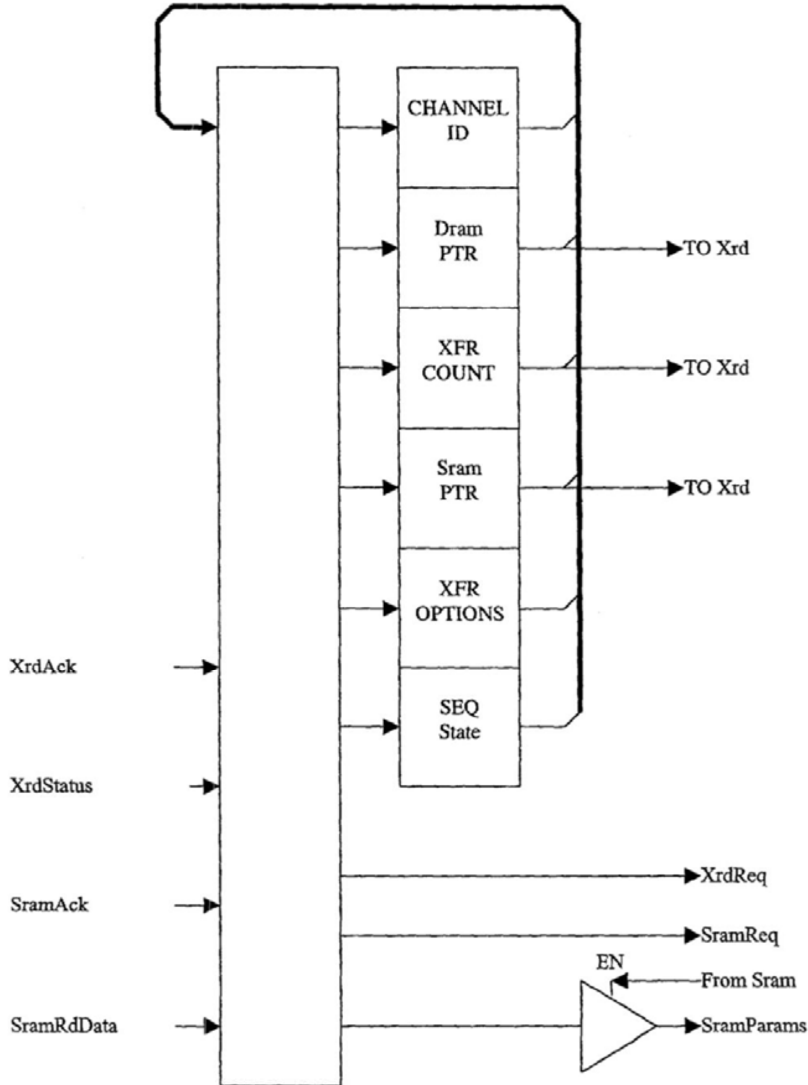
D2s can receive requests from any of the processor's thirty-two dma channels. Once a command request has been detected, **D2s**, operating as a slave sequencer, fetches a dma descriptor from an Sram location dedicated to the requesting channel which includes the dram address, Sram address and request size. **D2s** then issues a request to the **Xrd** sequencer causing the transfer of data to the Sram. The process repeats until the entire request has been satisfied at which time **D2s** writes ending status in to the Sram dma descriptor area and sets the channel done bit associated with that channel. **D2s** then monitors the dma channels for additional requests. Following is an illustration showing the major blocks involved in the movement of data from dram to Sram.



60961809 . 101497

WISTRON CORP. EXHIBIT 1031.115

DRAM TO SRAM SEQUENCER (D2s)



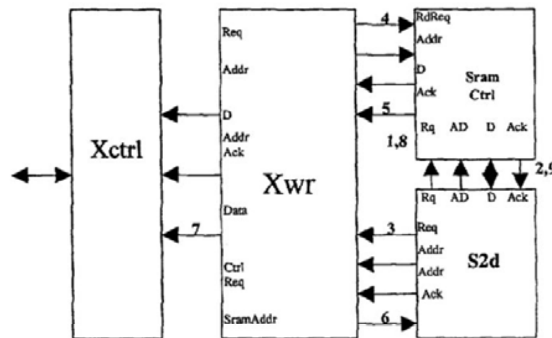
09741809 . 10149?

WISTRON CORP. EXHIBIT 1031.116

SRAM TO DRAM SEQUENCER (S2d)

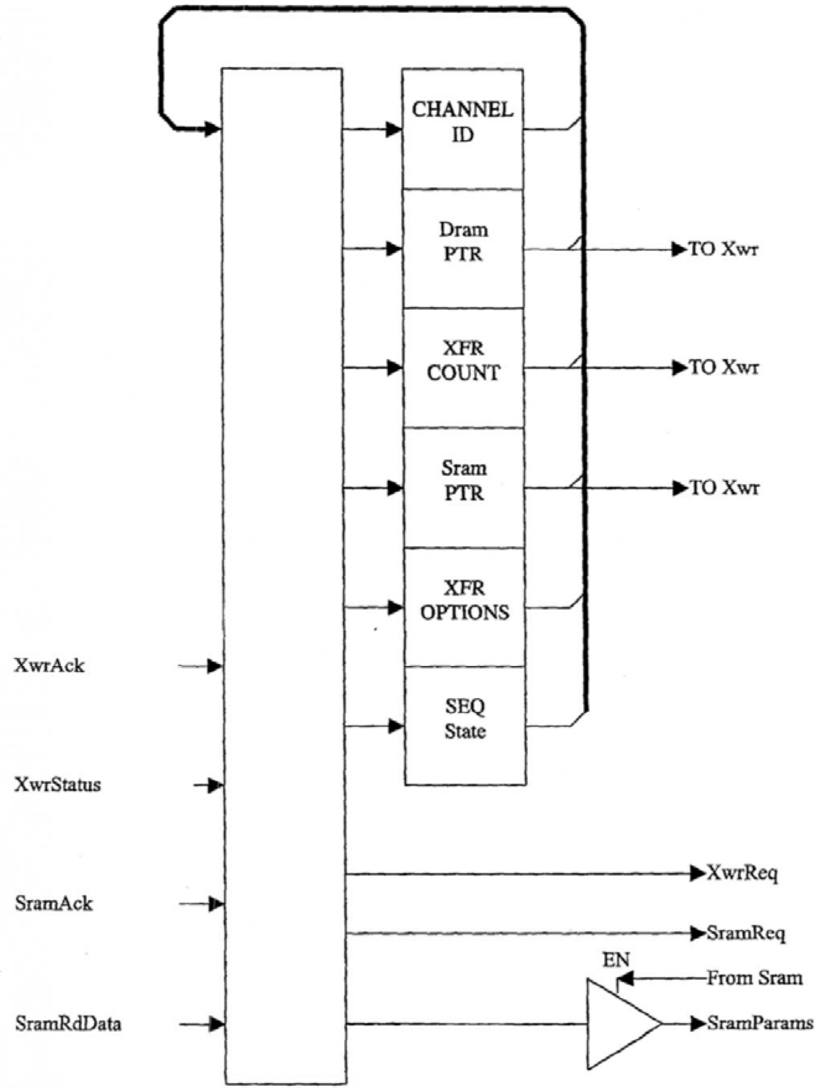
The **S2d** sequencer acts as both a slave sequencer and a master sequencer. Servicing channel requests issued by the Cpu, the **S2d** sequencer manages movement of data from Sram to dram by issuing requests to the **Xwr** sequencer.

S2d can receive requests from any of the processor's thirty-two dma channels. Once a command request has been detected, **S2d**, operating as a slave sequencer, fetches a dma descriptor from an Sram location dedicated to the requesting channel which includes the dram address, Sram address, checksum reset and request size. **S2d** then issues a request to the **Xwr** sequencer causing the transfer of data to the dram. The process repeats until the entire request has been satisfied at which time **S2d** writes ending status in to the Sram dma descriptor area and sets the channel done bit associated with that channel. **S2d** then monitors the dma channels for additional requests. Following is an illustration showing the major blocks involved in the movement of data from Sram to dram.



WISTRON CORP. EXHIBIT 1031.117

SRAM TO DRAM SEQUENCER (S2d)



60091509.101497

WISTRON CORP. EXHIBIT 1031.118

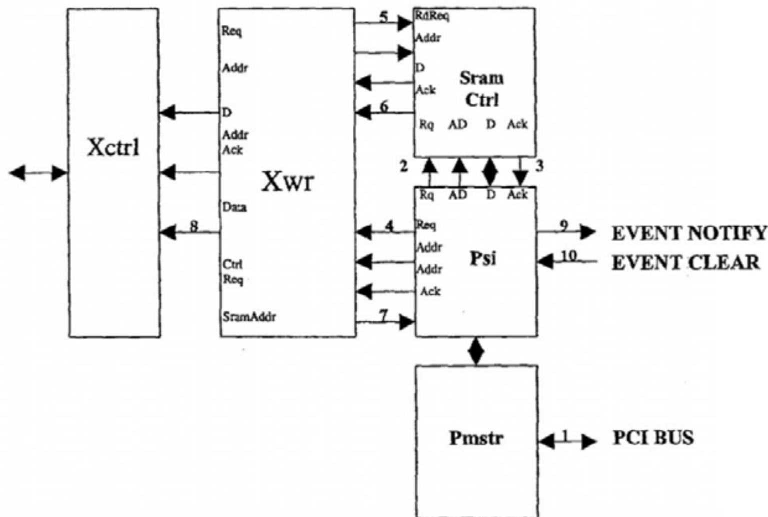
PCI SLAVE INPUT SEQUENCER (Psi)

The **Psi** sequencer acts as both a slave sequencer and a master sequencer. Servicing requests issued by a Pci master, the **Psi** sequencer manages movement of data from Pci bus to Sram and Pci bus to dram via Sram by issuing requests to the **SramCtrl** and **Xwr** sequencers.

Psi manages write requests to configuration space, expansion rom, dram, Sram and memory mapped registers. **Psi** separates these Pci bus operations in to two categories with different action taken for each. Dram accesses result in **Psi** generating write request to an Sram buffer followed with a write request to the **Xwr** sequencer. Subsequent write or read dram operations are retry terminated until the buffer has been emptied. An event notification is set for the processor allowing message passing to occur through dram space.

All other Pci write transactions result in **Psi** posting the write information including Pci address, Pci byte marks and Pci data to a reserved location in Sram, then setting an event flag which the event processor monitors. Subsequent writes or reads of configuration, expansion rom, Sram or registers are terminated with retry until the processor clears the event flag. This allows the INIC to keep pipelining levels to a minimum for the posted write and give the processor ample time to modify data for subsequent Pci read operations.

The following diagram depicts the sequence of events when **Psi** is the target of a Pci write operation. Note that events 4 through 7 occur only when the write operation targets the dram.



60061809.101497

WISTRON CORP. EXHIBIT 1031.119

PCI SLAVE OUTPUT SEQUENCER (Pso)

The **Pso** sequencer acts as both a slave sequencer and a master sequencer. Servicing requests issued by a Pci master, the **Pso** sequencer manages movement of data to Pci bus form Sram and to Pci bus from dram via Sram by issuing requests to the **SramCtrl** and **Xrd** sequencers.

Pso manages read requests to configuration space, expansion rom, dram, Sram and memory mapped registers. **Pso** separates these Pci bus operations in to two categories with different action taken for each. Dram accesses result in **Pso** generating read request to the **Xrd** sequencer followed with a read request to Sram buffer. Subsequent write or read dram operations are retry terminated until the buffer has been emptied.

All other Pci read transactions result in **Pso** posting the read request information including Pci address and Pci byte marks to a reserved location in Sram, then setting an event flag which the event processor monitors. Subsequent writes or reads of configuration, expansion rom, Sram or registers are terminated with retry until the processor clears the event flag. This allows the INIC to use a microcoded response mechanism to return data for the request. The processor decodes the request information, formulates or fetches the requested data and stores it in Sram then clears the event flag allowing **Pso** to fetch the data and return it on the Pci bus.

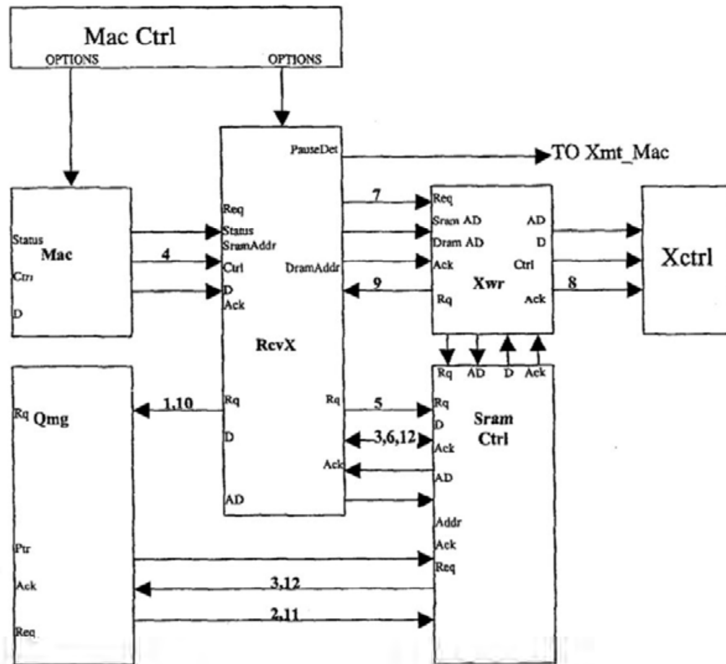
The following diagram depicts the sequence of events when **Pso** is the target of a Pci read operation.

WISTRON CORP. EXHIBIT 1031.120

FRAME RECEIVE SEQUENCER (RcvX)

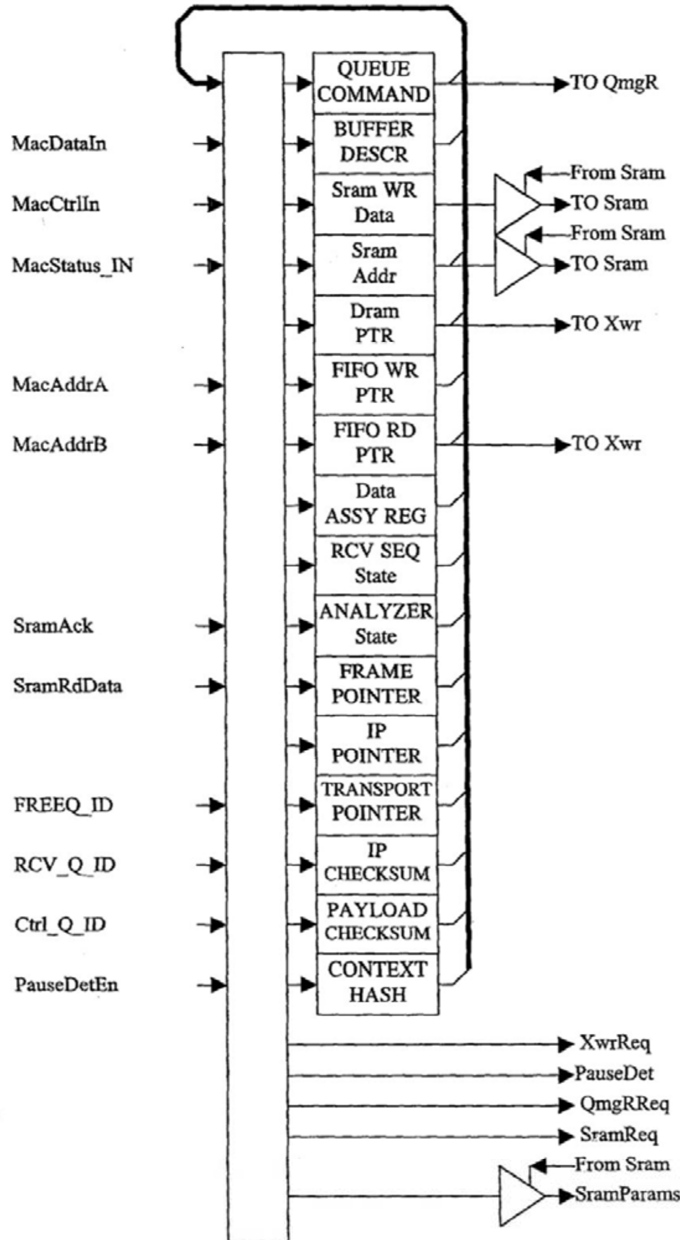
The receive sequencer **(RcvSeq)** analyzes and manages incoming packets, stores the result in dram buffers, then notifies the processor through the receive queue **(RcvQ)** mechanism. The process begins when a buffer descriptor is available at the output of the **FreeQ. RcvSeq** issues a request to the **Qmg** which responds by supplying the buffer descriptor to **RcvSeq. RcvSeq** then waits for a receive packet. The Mac, network, transport and session information is analyzed as each byte is received and stored in the assembly register **(AssyReg)**. When four bytes of information is available, **RcvSeq** requests a write of the data to the Sram. When sufficient data has been stored in the Sram based receive fifo, a dram write request is issued to **Xwr**. The process continues until the entire packet has been received at which point **RcvSeq** stores the results of the packet analysis in the beginning of the dram buffer. Once the buffer and status have both been stored, **RcvSeq** issues a write-queue request to **Qmg. Qmg** responds by storing a buffer descriptor provided by **RcvSeq**. The process then repeats. If **RcvSeq** detects the arrival of a packet before a free buffer is available, it ignores the packet and sets the **FrameLost** status bit for the next received buffer.

The following diagram depicts the sequence of events for successful reception of a packet followed by a definition of the receive buffer and the buffer descriptor as stored on the **RcvQ.**



WISTRON CORP. EXHIBIT 1031.121

FRAME RECEIVE SEQUENCER (RcvX)



60091018 .101497

WISTRON CORP. EXHIBIT 1031.122

RECEIVE BUFFER DESCRIPTOR

bit	name	description
31:30	reserved	
29:28	size	A copy of the bits in the **FreeBufDscr.**
27:00	address	Represents the last address +1 to which frame data was transferred. The address wraps around at the boundary dictated by the **S** bits. This can be used to determine the size of the frame received.

RECEIVE BUFFER FORMAT

FRAME Status A **OFFSET 0x0000:0x0003**

bit	name	description			
31	**attention**	Indicates one or more of the following: **CompositeErr, !IpDn, !MacADet & !MacBDet, IpMcst, IpBcst, !ethernet & !802.3Snap, !Ip4, !Tcp .**			
30	**CompositeErr**	Set when any of the error bits of **ErrStatus** are set or if frame processing stops while receiving a Tcp or Udp header.			
29	**CtrlFrame**	A control frame was received at our unicast or special MltCst address.			
28	**IpDn**	Frame processing Hlted due to exhaustion of the IP4 length counter.			
27	**802.3Dn**	Frame processing Hlted due to exhaustion of the 802.3 length counter.			
26	**MacADet**	Frame's destination address matched the contents of **MacAddrA.**			
25	**MacBDet**	Frame's destination address matched the contents of **MacAddrB.**			
24	**MacMcst**	The Mac detected a MltCst address.			
23	**MacBcst**	The Mac detected a BrdCst address.			
22	**IpMcst**	The frame processor detected an IP MltCst address.			
21	**IpBcst**	The frame processor detected an IP BrdCst address.			
20	**Frag**	The frame processor detected a Frag IP datagram.			
19	**IpOffst**	The frame processor detected a non-zero IP datagram offset.			
18	**IpFlgs**	The frame processor detected flags within the IP datagram.			
17	**IpOpts**	The frame processor detected a header length greater than 20 for the IP datagram.			
16	**TcpFlgs**	The frame processor detected an abnormal header flag for the TCP segment.			
15	**TcpOpts**	The frame processor detected a header length greater than 20 for the TCP segment.			
14	**TcpUrg**	The frame processor detected a non-zero urgent pointer for the TCP segment.			
13	**CarrierEvnt**	Refer to *E110 Technical Manual.*			
12	**LongEvnt**	Refer to *E110 Technical Manual.*			
11	**FrameLost**	Set when an incoming frame could not be processed as a result of an outstanding frame completion event not yet serviced by the utility processor.			
10	**reserved**				
10	**NoAck**	The frame processor detected a			
09:08	**FrameTyp**	00 - Reserved.	01- ethernet.	10 - 802.3.	11 - 802.3 Snap.
07:06	**NwkTyp**	00 - Unknown.	01- Ip4.	10 - Ip6	11 - ip other.
05:04	**TrnsptTyp**	00 - Unknown.	01- reserved.	10 - Tcp	11 - Udp
03	**NetBios**	A NetBios frame was detected.			
02	**reserved**				
01:00	**channel**	The Mac on which this frame was received.			

60051809.101497

FRAME Status B **OFFSET 0x0004:0x0007**

bit	name	description
31	**802.3Shrt**	End of frame was encountered before the 802.3 length count was exhausted.
30	**BufOvr**	The frame length exceded the buffer space available.
29	**BadPkt**	Refer to *E110 Technical Manual*.
28	**InvldPrmbl**	Refer to *E110 Technical Manual*.
27	**CrcErr**	Refer to *E110 Technical Manual*.
26	**DrblNbbl**	Refer to *E110 Technical Manual*.
25	**CodeErr**	Refer to *E110 Technical Manual*.
24	**IpHdrShrt**	The IP4 header length field contained a value less than 0x5.
23	**IpIncmplt**	The frame terminated before the IP length counter was exhausted.
22	**IpSumErr**	The IP header checksum was not 0xffff at the completion of the IP header read.
21	**TcpSumErr**	The session checksum was not 0xffff at the termination of session processing.
20	**TcpHdrShrt**	The TCP header length field contained a value less than 0x5.

19:16	**PrcssCd**	The state of the frame processor at the time the frame processing terminated.
		`0b0000 Processing Mac header.`
		`0b0001 Processing 802.3 LLC header.`
		`0b0010 Processing 802.3 SNAP header.`
		`0b0011 Processing unknown network data.`
		`0b0100 Processing IP header.`
		`0b0101 Processing IP data (unknown transport).`
		`0b0110 Processing transport header (IP data).`
		`0b0111 Processing transport data (IP data).`
		`0b1000 Processing IP processing complete.`
		`0b1001 Reserved.`
		`0b101x Reserved.`
		`0b11xx Reserved.`

15:08	**MacHsh**	The Mac destination-address hash. Refer to *E110 Technical Manual*.
07:00	**CtxHsh**	The 8-bit context-hash generated by exclusive-oring all bytes of the IP source address, IP destination-address, transport source port and the transport destination port.

TIME STAMP **OFFSET 0x0008:0x000B**

bit	name	description
31:00	**RcvTime**	The contents of **FreeClk** at the completion of the frame receive operation.

CHECKSUM **OFFSET 0x000C:0x000F**

bit	name	description
31:16	**IpChksum**	Reflects the value of the IP header checksum at frame completion or IP header completion. If an IP datagram was not detected, the checksum provides a total for the entire data portion of the received frame. The data area is defined as those bytes received after the type field of an ethernet frame, the LLC header of an 802.3 frame or the SNAP header of an 802.3-SNAP frame.
15:00	**TcpChksum**	Reflects the value of the transport checksum at IP completion or frame completion. If IP was detected but session was unknown, the checksum will not include the psuedo-header. If IP was not detected, the checksum will be 0x0000.

RESERVED **OFFSET 0x0010:0x0011**

FRAME Data **OFFSET 0x0012:END OF BUFFER**

60091808 . 101497

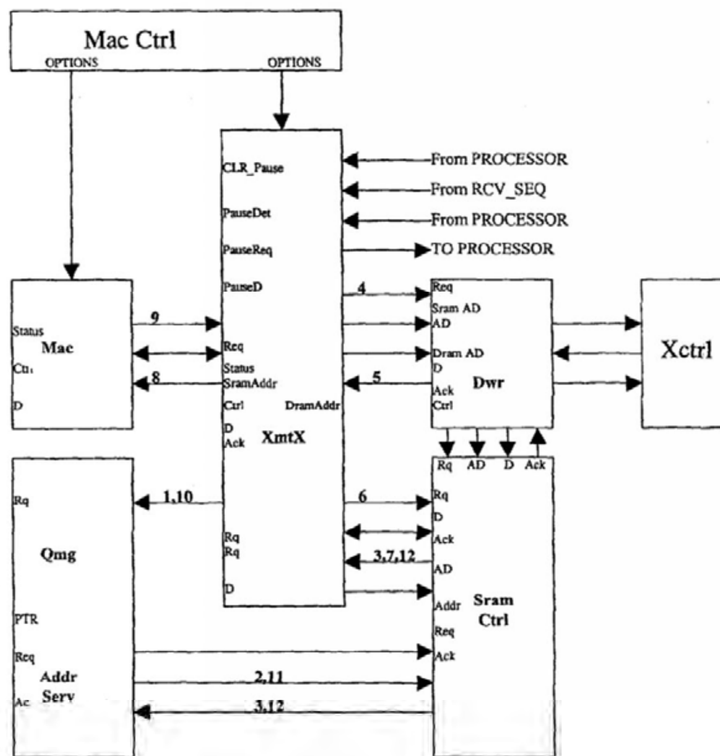
WISTRON CORP. EXHIBIT 1031.124

FRAME TRANSMIT SEQUENCER (XmtX)

The transmit sequencer (**XmtSeq**) analyzes and manages outgoing packets, using buffer descriptors retrieved from the transmit queue (**XmtQ**) then storing the descriptor for the freed buffer in the free buffer queue (**FreeQ**). The process begins when a buffer descriptor is available at the output of the **XmtQ**. **XmtSeq** issues a request to the **Qmg** which responds by supplying the buffer descriptor to **XmtSeq**. **XmtSeq** then issues a read request to the **Xrd** sequencer. Next, **XmtSeq** issues a read request to **SramCtrl** then instructs the Mac to begin frame transmission. The Mac accepts data from **XmtSeq** which analyzes the packet as it flys-by in order to generate checksums to insert in the data stream. Once the frame transmission has completed, **XmtSeq** stores the buffer descriptor on the **FreeQ** thereby recycling the buffer.

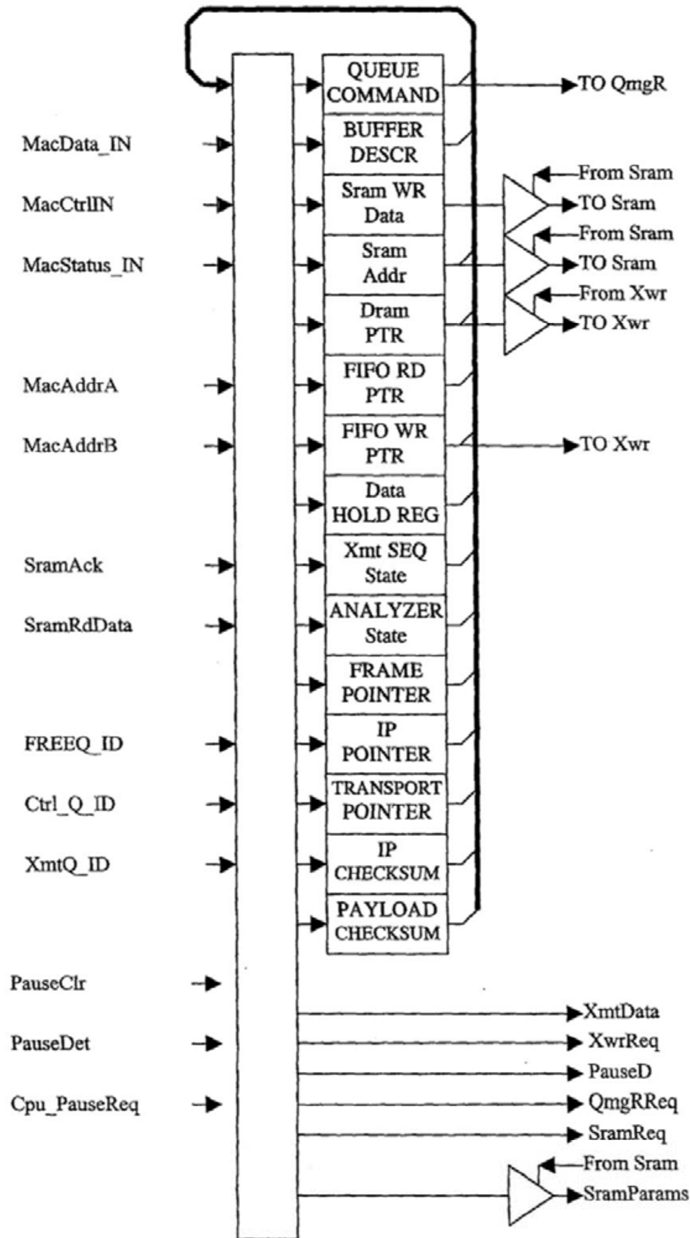
The following diagram depicts the sequence of events for successful transmission of a packet followed by a definition of the receive buffer and the buffer descriptor as stored on the **XmtQ**.

60081009 . 101497



WISTRON CORP. EXHIBIT 1031.125

FRAME TRANSMIT SEQUENCER (XmtX)



60061809.101497

WISTRON CORP. EXHIBIT 1031.126

TRANSMIT BUFFER DESCRIPTOR

bit	name	description
31	**ChksumEn**	When set, **XmtSeq** will insert a calculated checksum. When reset, **XmtSeq** will not alter the outgoing data stream.
30	**reserved**	
29:28	**size**	Represents the size of the buffer by indicating at what boundary the buffer should start and terminate. This is used in combination with **EndAddr** to determine the starting address of the buffer : **S** = 0 256B boundary. **A[7:0]** ignored. **S** = 1 2KB boundary. **A[10:0]** ignored. **S** = 2 4KB boundary. **A[11:0]** ignored. **S** = 3 32KB boundary. **A[14:0]** ignored.
27:00	**EndAddr**	The address of the last byte to transmit plus one.

TRANSMIT BUFFER FORMAT

CHECKSUM PRIMER OFFSET 0x0000:0x0003

bit	name	description
31:00	**Primer**	A value to be added during checksum accumulation. For IPV4, this should include the psuedo-header values, protocol and Tcp-length.

RESERVED OFFSET 0x0004:0x0005

FRAME Data OFFSET 0x0006:END OF BUFFER

TRANSMIT Status VECTOR

bit	name	description
31	**LnkErr**	Indicates that a link status error occured before or during transmit.
30:15	**reserved**	
14	**ExcessDeferral**	Refer to *E110 Technical Manual*.
13	**LateAbort**	Refer to *E110 Technical Manual*.
12	**ExcessColl**	Refer to *E110 Technical Manual*.
11	**UnderRun**	Refer to *E110 Technical Manual*.
10	**ExcessLgth**	Refer to *E110 Technical Manual*.
09	**Okay**	Refer to *E110 Technical Manual*.
08	**deferred**	Refer to *E110 Technical Manual*.
07	**BrdCst**	Refer to *E110 Technical Manual*.
06	**MltCst**	Refer to *E110 Technical Manual*.
05	**CrcErr**	Refer to *E110 Technical Manual*.
04	**LateColl**	Refer to *E110 Technical Manual*.
03:00	**CollCnt**	Refer to *E110 Technical Manual*.

09011809 "101497

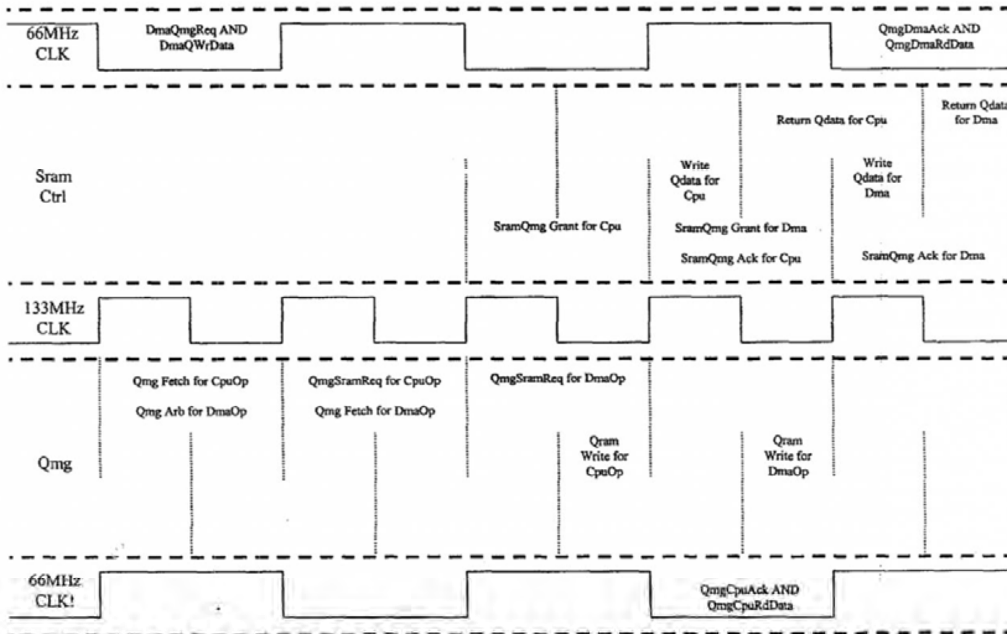
WISTRON CORP. EXHIBIT 1031.127

QUEUE MANAGER (Qmg)

The INIC includes special hardware assist for the implementation of message and pointer queues. The hardware assist is called the queue manager (**Qmg**) and manages the movement of queue entries between Cpu and Sram, between dma sequencers and Sram as well as between Sram and dram. Queues comprise three distinct entities; the queue head (**QHd**), the queue tail (**QTl**) and the queue body (**QBdy**). **QHd** resides in 64 bytes of scratch ram and provides the area to which entries will be written (pushed). **QTl** resides in 64 bytes of scratch ram and contains queue locations from which entries will be read (popped) . **QBdy** resides in dram and contains locations for expansion of the queue in order to minimize the Sram space requirements. The **QBdy** size depends upon the queue being accessed and the initialization parameters presented during queue initialization.

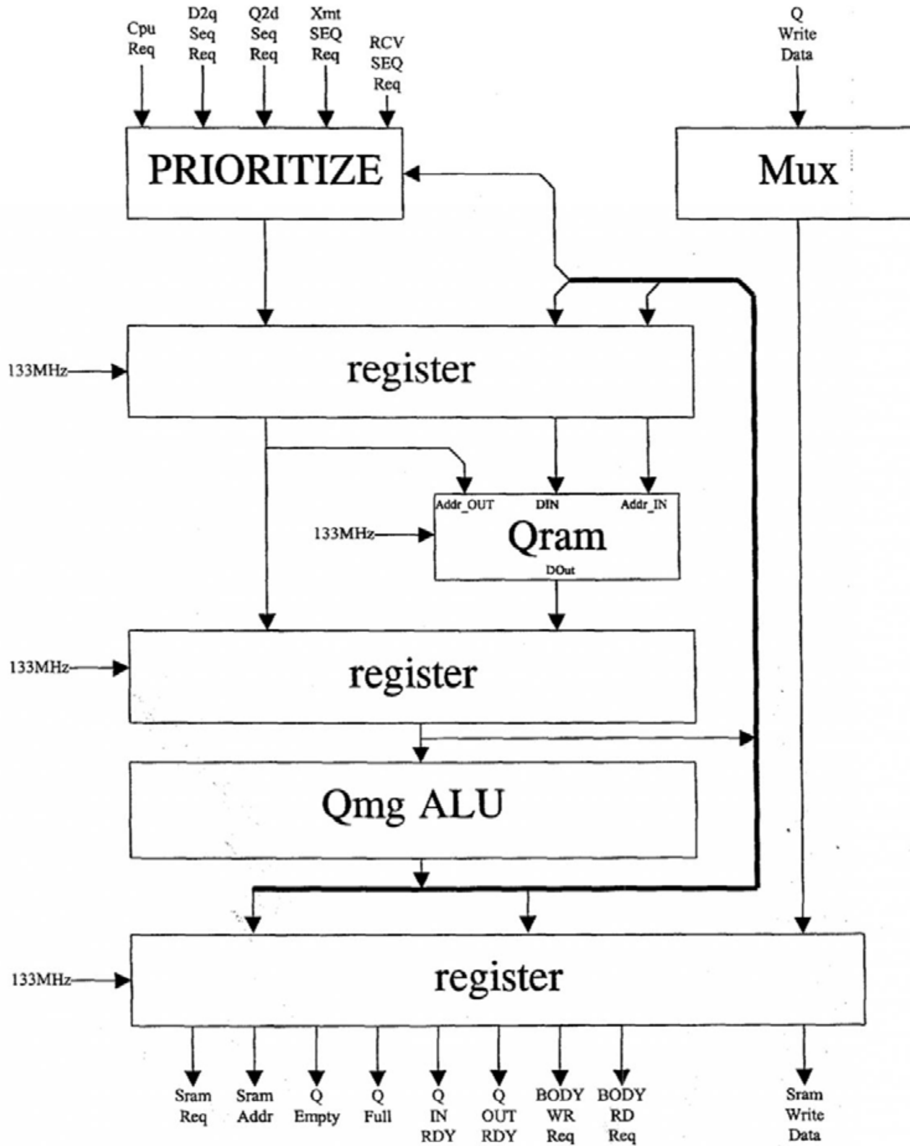
Qmg accepts operations from both Cpu and dma sources. Executing these operations at a frequency of 133MHz, **Qmg** reserves even cycles for dma requests and reserves odd cycles for Cpu requests. Valid Cpu operations include initialize queue (**InitQ**), write queue (**WrQ**) and read queue (**RdQ**). Valid dma requests include read body (**RdBdy**) and write body (**WrBdy**). **Qmg** working in unison with **Q2d** and **D2q** generate requests to the **Xwr** and **Xrd** sequencers to control the movement of data between the **QHd**, **QTl** and **QBdy**.

The preceding block diagram shows the major functions of **Qmg**. The arbiter selects the next operation to be performed. The dual-ported Sram holds the queue variables **HdWrAddr, HdRdAddr, TlWrAddr, TlRdAddr, BdyWrAddr, BdyRdAddr** and **QSz. Qmg** accepts an operation request, fetches the queue variables from the queue ram (**Qram**), modifies the variables based on the current state and the requested operation then updates the variables and issues a read or write request to the Sram controller. The Sram controller services the requests by writing the tail or reading the head and returning an acknowledge.



WISTRON CORP. EXHIBIT 1031.128

QUEUE MANAGER (Qmg)



WISTRON CORP. EXHIBIT 1031.129

DMA OPERATIONS

DMA operations are accomplished through a combination of thirtytwo dma channels (**DmaCh**) and seven dma sequencers (**DmaSeq**). Each dma channel provides a mechanism whereby a Cpu can issue a command to any of the seven dma sequencers. Where as the dma channels are multi-purpose, the dma sequencers they command are single purpose as follows.

dma seq #	name	description
0	none	This is a no operation address.
1	**D2dSeq**	Moves data from ExtMem to ExtMem.
2	**D2sSeq**	Moves data from ExtMem bus to sram.
3	**D2pSeq**	Moves data from ExtMem to Pci bus.
4	**S2dSeq**	Moves data from sram to ExtMem.
5	**S2pSeq**	Moves data from sram to Pci bus.
6	**P2dSeq**	Moves data from Pci bus to ExtMem.
7	**P2sSeq**	Moves data from Pci bus to sram.

The processors manage dma in the following way. The processor writes a dma descriptor to an Sram location reserved for the dma channel. The format of the dma descriptor is dependent upon the targeted dma sequencer. The processor then writes the dma sequencer number to the channel command register.

Each of the dma sequencers polls all thirtytwo dma channels in search of commands to execute. Once a command request has been detected, the dma sequencer fetches a dma descriptor from a fixed location in Sram. The Sram location is fixed and is determined by the dma channel number. The dma sequencer loads the dma descriptor in to it's own registers, executes the command, then overwrites the dma descriptor with ending status. Once the command has halted, due to completion or error, and the ending status has been written, the dma sequencer sets the done bit for the current dma channel.

The done bit appears in a dma event register which the Cpu can examine. The Cpu fetches ending status from Sram, then clears the done bit by writing zeroes to the channel command (**ChCmd**) register. The channel is now ready to accept another command.

The format of all channel command registers is as follows.

bit	name	description
31:11	reserved	Data written to these bits is ignored.
10:8	**ChCmd**	0 - Stops execution of the current operation and clears the corresponding event flag. 1 - Transfer data from ExtMem to ExtMem. 2 - Transfer data from ExtMem bus to sram. 3 - Transfer data from ExtMem to Pci bus. 4 - Transfer data from sram to ExtMem. 5 - Transfer data from sram to Pci bus. 6 - Transfer data from Pci bus to ExtMem. 7 - Transfer data from Pci bus to Sram.
07:05	reserved	Data written to these bits is ignored.
04:00	**ChId**	Provides the channel number for the channel command.

60061809 . 101497

WISTRON CORP. EXHIBIT 1031.130

The format of the **P2d** or **P2s** descriptor is as follows.

bit	name	description
127:96	**PciAddrH**	Bits [63:32] of the Pci address.
95:64	**PciAddrL**	Bits [31:00] of the Pci address.
59:32	**MemAddr**	Bits [27:00] of the ExtMem address or bits [15:00] of the Sram address.
31	**PciEndian**	When set, selects big endian mode for Pci transfers.
30	**WideDbl**	When set, disables Pci 64-bit mode.
22	**DstFlash**	Selects Flash for the external memory destination of **P2d.**
15:00	**XfrSz**	Bits [15:00] of the requested dma size expressed in bytes.

The format of the **S2p** or **D2p** descriptor is as follows.

bit	name	description
123:96	**MemAddr**	Bits [27:00] of the ExtMem address or bits [15:00] of the Sram address.
95:64	**PciAddrH**	Bits [63:32] of the Pci address.
63:32	**PciAddrL**	Bits [31:00] of the Pci address.
30	**SrcFlash**	Selects Flash for the external memory source of **D2p.**
23	**PciEndian**	When set, selects big endian mode for Pci transfers.
22	**WideDbl**	When set, disables Pci 64-bit mode.
15:00	**XfrSz**	Bits [15:00] of the requested dma size expressed in bytes.

The format of the **S2d, D2d** or **D2s** descriptor is as follows.

bit	name	description
127:124	**reserved**	Reserved for future use.
123:96	**SrcAddr**	Bits [27:00] of the ExtMem address or bits [15:00] of the Sram address.
95:60	**reserved**	Reserved for future use.
59:32	**DstAddr**	Bits [27:00] of the ExtMem address or bits [15:00] of the Sram address.
30	**FlashSel**	Selects Flash for the external memory source of **D2d** or **D2s.**
22	**FlashSel**	Selects Flash for the external memory destination of **S2p** or **D2d.**
15:00	**XfrSz**	Bits [15:00] of the requested dma size expressed in bytes.

The format of the ending status or all channels is as follows.

bit	name	description
127:64	**reserved**	Not used.
63:32	**ChkSum**	Represents the 1's compliment sum of all halfwords transferred during a **P2d** or **D2d** operation only.
31:24	**reserved**	Reserved for future use.
23:20	**SrcStatus**	TBD.
19:16	**DstStatus**	TBD.
15:00	**XfrSz**	Bits [15:00] of the residual dma size expressed in bytes. This value will be zero if the dma operation was successful

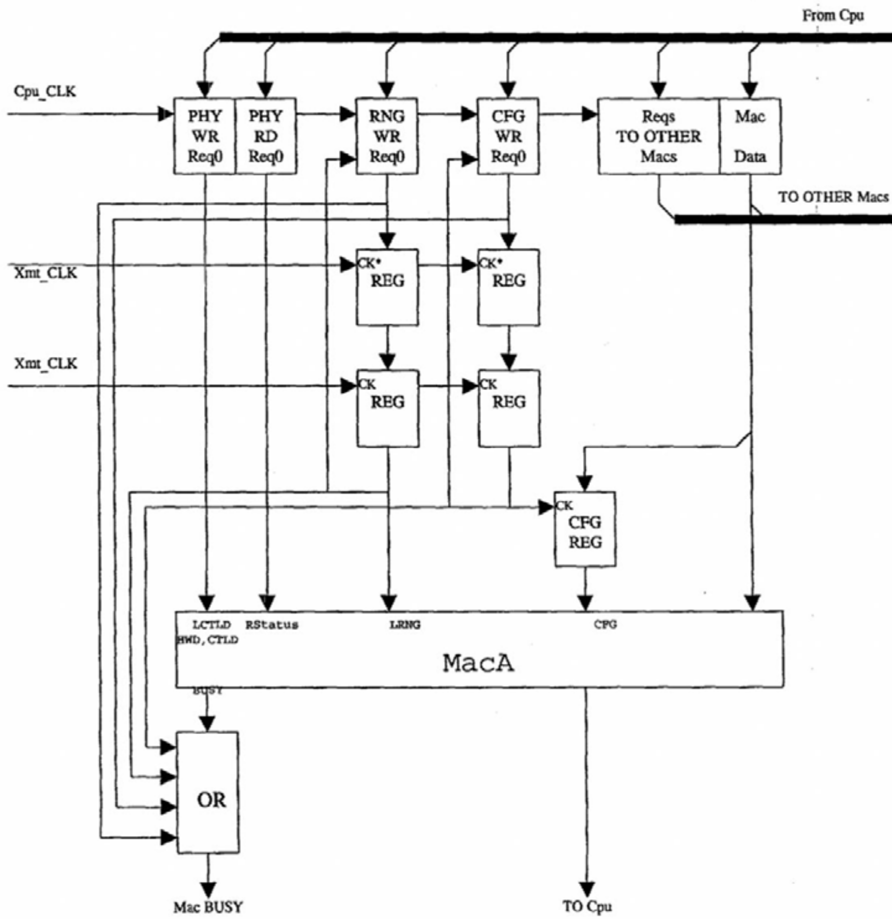
The format of the **ChEvnt** register is as follows.

bit	name	description
31:00	**ChDn**	Each bit represents the done flag for the respective dma channel. These bits are set by a dma sequencer upon completion of the channel command. Cleared when the processor writes 0 to the corresponding **ChCmd** register **ChCmdOp** field.

60061809 . 101497

WISTRON CORP. EXHIBIT 1031.131

MAC CONTROL (Macctrl)



WISTRON CORP. EXHIBIT 1031.132

Appendix A

The following load calculations are based on the following basic formulae:

N = X * R (Little's Law) where
N = number of jobs in the system (either in progress or in a queue),
X = system throughput,
R = response time (which includes time waiting in queues).

U = X * S (from Little's Law) where
S = service time,
U = utilization.

R = S / (1-U) for exponential service times (which is the worst-case assumption).

A 256 byte frame at 100Mb/sec takes 20 μsec per frame.
4 * 100 Mbit ethernets receiving at full frame rate is:
51200 (4 * 12800) frames/sec @ 1024 bytes/frame
102000 frames/sec @ 512 bytes/frame
204000 frames/sec @ 256 bytes/frame.

The following calculations assume 250 instructions/frame, 45nsec clock. Thus
S = 250 * 45 nsecs = 11.2 μsecs.

Av. Frame Size	**Thruput (X)**	**Utilization (U)**	**Response (R)**	**Nbr. in system (N)**
1024	51200	.57	26 usecs	1.3
512	102000	> 1	--	--
256	204000	> 1	--	--

Lets look at it for varying instructions per frame assuming 512 bytes per frame average.

Instns Per Frame	**Service Time (S)**	**Thruput (X)**	**Utilization (U)**	**Response (R)**	**Nbr. in system (N)**
250	11.2 usec	102000	> 1	--	--
250	11.2	85000 (*)	.95	224 usecs	19
250	11.2	80000 (**)	.89	101	8
225	10	102000	1.0	--	--
225	10	95000 (*)	.95	200	19
225	10	89000 (**)	.89	90	8
200	9	102000	.9	90	9
150	6.7	102000	.68	20	2

(*) shows what frame rate can be supported to get a utilization of less than 1.
(**) shows what frame rate can be supported with 8 SRAM TCB buffers and at least 8 process contexts.

09061809 . 101497

WISTRON CORP. EXHIBIT 1031.133

If 100 instructions / frame is used, S = 100 * 45 nsecs = 4.5 usecs, and we can support 256 byte frames:

100	4.5	204000	.91	50	10

Firstly note that these calculations assume that response times increase exponentially as utilization increases. This is the worst-case assumption, and probably may not be true for our system.

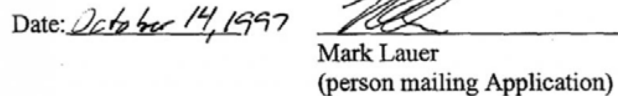
The figures show that to support a theoretical full 4 * 100 Mbit receive load with an average frame size of 512 bytes, there will need to be 19 active "jobs" in the system, assuming 250 instructions per frame. Due to SRAM limitations, the current design specifies 8 SRAM buffers for active TCBs, and not to swap a TCB out of SRAM once it is active. So under these limitations, the INIC will not be able to keep up with the full frame rate. Note that the initial implementation is trying to use only 8KB of SRAM, although 16KB may be available, in which case 19 TCB SRAM buffers could be used. This is a cost trade-off.

The real point here is the effect of instructions/frame on the throughput that can be maintained. If the instructions/frame drops to 200, then the INIC is capable of handling the full theoretical load (102000 frames/second) with only 9 active TCBs. If it drops to 100 instructions per frame, then the INIC can handle full bandwidth at 256 byte frames (204000 frames/second) with 10 active TCBs. The bottom line is that ALL hardware-assist that reduces the instructions/frame is really worthwhile. If header-assist hardware can save us 50 instructions per frame then it goes straight to the throughput bottom line.

CERTIFICATE OF MAILING UNDER 37 CFR 1.10

I hereby certify that this Provisional Patent Application is being deposited with the United States Postal Service as "Express Mail Post Office to Addressee", label number EH756230105US, in an envelope addressed to: Assistant Commissioner for Patents, Washington, D.C. 20231, on October 14, 1997.

Date: October 14, 1997



Mark Lauer
(person mailing Application)

WISTRON CORP. EXHIBIT 1031.134