An Interrupt Controller for the RAW Processor
by
Anant K. Saraswat
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2003
© Massachusetts Institute of Technology 2003. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
February 6, 2003

Certified by
Anant Agarwal
Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students
An Interrupt Controller for the RAW Processor

by

Anant K. Saraswat

Submitted to the Department of Electrical Engineering and Computer Science on February 6, 2003, in partial fulfillment of the requirements for the degree of Master of Engineering

Abstract

The Reconfigurable Architecture Workstation (RAW) chip is an experimental parallel multi-processor that consists of a matrix of identical RISC processors. The chip has a dynamic network to facilitate communication between processors and the off-chip DRAM or external devices. Because external interrupts are by nature unscheduled and unsolicited, special care must be taken when designing an interrupt delivery system to ensure that delivery of interrupts does not interfere with cache traffic or deadlock the shared network. Two competing designs for the interrupt controller emerged during the course of this research; this paper describes and evaluates each design and gives justification for the choice that was implemented in hardware.

Thesis Supervisor: Anant Agarwal
Title: Professor
Acknowledgments

This research would not have been possible without the dedication and hard work of the entire RAW team. I would like to acknowledge Michael Taylor, Walter Lee, David Wentzlaff, Fataneh Ghodrat, Jason Miller, and Jason Lee for their work in designing and implementing the RAW architecture. Nathan Shnidman implemented the DRAM controller. Volker Strumpen, Paul Johnson, and Ben Greenwald provided valuable insights into the nature of the RAW operating system. David Wentzlaff designed the hierarchical routing system and, in conjunction with Patrick Griffin and Judy Chen, the I/O Gasket and the PCI bridge. I would also like to especially acknowledge David Wentzlaff for his invaluable guidance in navigating the often perilous world of Verilog synthesis. I would also like to thank my family, friends, and the members of the MIT Tae Kwon Do Club for their support during the course of this research.

Lastly, I would like to thank Professor Anant Agarwal, Associate Director of the Lab for Computer Science, for his guidance, his feedback, and for granting me the honor of being a part of the RAW project.
Contents

1 Introduction 13
  1.1 Organization ........................................ 14

2 RAW Overview 15
  2.1 Off-chip Routing .................................... 19
  2.2 The RAW Deadlock Problem ........................... 21

3 Overview of the Interrupt Problem 25
  3.1 Interrupt Control in a Multi-Pentium System: The Advanced Programmable Interrupt Controller (APIC) .................... 26
  3.2 Challenges on the RAW Platform .................... 29

4 Initial Design 33
  4.1 Interrupt Notification via the MDN Interrupt Header ........ 34
  4.2 Vector Queuing on the Interrupt Controller .......... 35
    4.2.1 Notification and Vector Request ............... 35
    4.2.2 Vector Delivery ............................... 37
    4.2.3 Vector Processing and Prioritization .......... 38
  4.3 Vector Storage in the DRAM .......................... 39
    4.3.1 Queuing and Tile Notification .................. 40
    4.3.2 Interrupt Processing .......................... 41
  4.4 Evaluation of Designs ............................... 43
    4.4.1 Testing Methodology ......................... 43
4.4.2 Results .................................................. 45
4.5 Performance Analysis ..................................... 45
4.5.1 Scalability ............................................... 47
4.6 Conclusion .................................................. 50

5 Hardware Implementation and Software Conventions 51
5.1 Interrupt Controller Hardware Implementation .......... 51
  5.1.1 I/O unit ................................................. 52
  5.1.2 Receive FSM ............................................. 56
  5.1.3 Interrupt Storage Unit .................................. 61
  5.1.4 Tile Information Module .................................. 62
  5.1.5 Controller Information Module ....................... 65
  5.1.6 Transmit FSM ............................................ 66
  5.1.7 Send Buffer .............................................. 69
5.2 Design of External Interrupt Transmitter ............... 71

6 Implementation and Testing .................................. 75

7 Conclusions Future Research 79
  7.1 Conclusion .................................................. 81
List of Figures

2-1 Dynamic Network Header ........................................ 16
2-2 Extended Header .................................................. 20
2-3 A Device Related Message with an ACK requested ............. 20
2-4 A General ACK ..................................................... 20

4-1 Logical Block Diagram of the “On-board” Controller ........... 36
4-2 Logical Block Diagram of DMA Version .......................... 39

5-1 Overall Block Diagram of IC Hardware .......................... 53
5-2 I/O unit ............................................................ 54
5-3 Receive FSM ....................................................... 57
5-4 Device ID request .................................................. 58
5-5 Device Location Programming Message .......................... 59
5-6 Buffer Space Allocation Message ................................ 59
5-7 Store ACK from DRAM ............................................. 60
5-8 Tile Information Programming Message .......................... 60
5-9 Interrupt Request Message ........................................ 61
5-10 Interrupt Storage Unit .......................................... 63
5-11 Tile Information Module ......................................... 65
5-12 Transmit FSM ...................................................... 67
5-13 Send Buffer ....................................................... 70
List of Tables

2.1 FBit Encoding .................................................. 17
2.2 User Field Encoding ............................................. 17
4.1 Stream Latency, Single Tile .................................. 45
4.2 Stream Latency, Multiple Tiles ............................... 45
5.1 Incoming Message Opcodes ................................. 58
Chapter 1

Introduction

The RAW (Reconfigurable Architecture Workstation) chip is an experimental Symmetric Multi-Processing platform. It consists of a 2-dimensional matrix of RISC processors, known as “tiles”. These “tiles” are connected to each other by 4 independent mesh networks, which are themselves controllable by the programmer. Unlike many conventional architectures, which aim to improve performance by building increasingly complex hardware, RAW provides a scalable fabric of very simple computational units and extracts performance by allowing the wiring between these units to be optimized for different applications. In addition, because three of the four networks go off-chip, the RAW architecture provides an unprecedented level of connectivity with the outside word.

Although the “extroverted” nature of the RAW architecture presents significant new opportunities for real-time applications, there are numerous challenges involved in managing the communications between the RAW chip and its environment. One of the most significant problems is the potential for deadlock caused by large, unpredictable streams of data. Because it is impossible to predict all of the communication that can occur in the system, two of the four networks on the RAW chip are dynamic networks. One of these networks is used for memory transactions between the caches and DRAM and for communication with external devices. Unfortunately, the need for dynamic communication introduces the potential for deadlock of the processor. The problem of deadlock is a potentially serious issue, since it can impede the
forward progress of certain processes being run by the chip. Previous research has effectively solved the problem of deadlock in memory accesses, but we must also have a mechanism for deadlock-free, dynamic communication between RAW and external entities. In order to achieve this goal, one of the core problems that must be solved is the problem of external interrupt delivery. Since the RAW compiler cannot schedule the occurrence of external events, we must have a way to allow external devices to "interrupt" the RAW chip to notify it that some event has occurred.

The goal of this thesis is to find a solution to the interrupt delivery problem that both avoids deadlock in the dynamic network and provides the flexibility to deal with a large number of external devices. It involves the design and implementation of special hardware that allows interrupts to be injected into the RAW fabric and routed to the appropriate tile in a deadlock-free manner. In addition, the thesis requires the development of software conventions that the RAW Operating System, will have to obey in order to interface correctly with the interrupt control hardware.

1.1 Organization

This thesis is structured as follows. Section II gives a brief overview of the RAW chip and its communication networks. Section III discusses some general issues in interrupt processing and describes how the problem is dealt with in other existing architectures. Section IV discusses various schemes that can be used for addressing the interrupt-handling problem on the RAW chip, and section V describes the design and implementation of the special hardware and software used for this purpose. Finally, section VI proposes ideas for further research.
Chapter 2

RAW Overview

The current version of the RAW chip is composed of 16 identical "tiles" in a 4-by-4 array. Each tile has a RISC processor, which is based on the MIPS R4000 architecture, and special routing hardware to interface with the communication system. The tile processors have 32K of instruction memory and a 32K non-write-allocate data cache. The tiles are linked to each other, and to the outside world, by four 32-bit communication networks. Each tile has crossbars that can route traffic on these networks in any direction. Signals in these networks traverse a tile in one clock cycle, with a maximum throughput of one word per cycle per network. Two of the networks are "static networks" that are programmed with routing instructions generated by the compiler. The other two networks are the Memory Dynamic Network (MDN), which is used for memory accesses and communication with external devices, and the general dynamic network (GDN), which is used for user-level dynamic communication. All four of the networks are mapped to registers, making them accessible to normal instructions, and, like normal registers, are fully bypassed.

Routing in the static networks is controlled by a special "switch processor" on each tile. The switch processor has four general-purpose registers and uses a 64-bits instruction word. The 4 high-order bytes of the instruction word are used to implement a limited set of flow control instructions and a "move" instruction that transfers data between the static networks, the switch processor's register file, and the tile processor. The 4 low-order bytes are used to set up the routing in the static network.
crossbar. On a single clock cycle, the switch processor executes the instruction in the 4 high-order bytes and routes single words according to the directives in the low-order bytes.

Unlike the static networks, the dynamic networks have no programmable router. Furthermore, data is not routed word-by-word, but is instead routed at the packet level. Every message that is sent on the dynamic network is preceded by a header word. The format of the header is shown in Figure 2-1. The first three bits of the header word, the “Funny” bits, are used to route data off of the chip once a message reaches the last tile in its path (see Table 2.1). The next five bits indicate the length of the subsequent message, up to a maximum of 31 words (Note: in this document, whenever the length of a message on the dynamic network is referred to, the number does not include the header unless otherwise specified). The “user” field is used to distinguish between the different types of messages that can be carried by the MDN. It is not used by the routing hardware of the tiles in the path, but is examined at the final destination of the message (see Table 2.2). The last 20 bits of the message encode the y- and x- coordinates of the tile that originated the message and the tile for which the message is destined. In the case where either the sender or the target is an external device, the coordinates used in the header are the coordinates of the tile to which the device is directly connected.

<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>28</th>
<th>24</th>
<th>23</th>
<th>20</th>
<th>19</th>
<th>14</th>
<th>10</th>
<th>9</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBits</td>
<td>Length</td>
<td>User</td>
<td>Origin Y</td>
<td>Origin x</td>
<td>Dest. Y</td>
<td>Dest. X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-1: Dynamic Network Header

The dynamic crossbar on each tile sets up the routing of data based on the coordinates of the destination tile. The dynamic networks use a dimension-ordered wormhole routing scheme [3]. Data is routed first in the X-direction, and then in the Y-direction. Whenever an idle dynamic router reads a header word, it sets up the crossbar on that tile to route the rest of the data in the message in the appropriate direction. The dynamic router will not attempt to schedule a new route until the
<table>
<thead>
<tr>
<th>Numeric</th>
<th>Final Route Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None (Tile)</td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td>2</td>
<td>West</td>
</tr>
<tr>
<td>3</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>East</td>
</tr>
<tr>
<td>5</td>
<td>North</td>
</tr>
<tr>
<td>6</td>
<td>Invalid</td>
</tr>
<tr>
<td>7</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Table 2.1: FBit Encoding

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cache Line Read</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Cache Line Write</td>
</tr>
<tr>
<td>5</td>
<td>Device Related Message 0</td>
</tr>
<tr>
<td>6</td>
<td>Device Related Message 1</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>Memory Related Message 0</td>
</tr>
<tr>
<td>10</td>
<td>Reserved 1</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>MDN Relay</td>
</tr>
<tr>
<td>14</td>
<td>Store Acknowledgement</td>
</tr>
<tr>
<td>15</td>
<td>Interrupt</td>
</tr>
</tbody>
</table>

Table 2.2: User Field Encoding
correct number of words, as determined by the length field of the header, has been routed through the current configuration. If a second message needs to be routed through a tile while the tile's dynamic crossbar is busy, that message will be stalled at the crossbar's input buffer (and possibly in other buffers on the chip, depending on the length of the message), until the current message has been routed. When the header word of a message reaches the destination tile, it is automatically pulled out of the input buffer, since it is used solely for routing purposes (there are certain exceptions, which will be discussed below). Thus, for buffer management purposes, a header word can be considered a "free" word, that is, a word that takes up no space in the network.

The tile processor supports 7 different types of interrupts at two different priority levels, system and user. The processor has a special-purpose register called the EX_BITS register, with bits 6 down through 0 each representing one of the 7 interrupt types. Bit 3 of the register corresponds to external interrupts. In addition to the EX_BITS register, the tile processor also has an SPR called the EX_MASK register, which masks the interrupts. If a bit in the EX_BITS register and the corresponding bit in the EX_MASK register are both set, it indicates to the processor that an interrupt of that type has occurred. The tile hardware is designed in such a way that, if a header is received off of the MDN with the user field set to 0xF, the bit corresponding to external interrupts in the EX_BITS register will be set. This feature is the centerpiece of the RAW interrupt delivery system, as shown in Chapter 4.

Communication over all networks is governed by the Static Input Buffer (SIB) protocol. In the SIB protocol, every sending element is responsible for keeping track of the amount of buffer space available in the target. If a sender has data to send, and it knows that its target has space, it will present the data on the target's DataIn port and assert ValidIn to indicate that data is available. There is no need to acknowledge the data, because, assuming that both parties are obeying the SIB protocol, the sender knows that there is space available in the target's buffer. Whenever a data word is consumed from the target's buffer, the target asserts the "Yummy" signal back to the sender. By keeping track of how many data words it has sent and how
many yummy signals it has received, the sender knows whether or not the target has space on a given cycle. If the sender has data available, but knows that the target has no buffer space, the sender will stall until space becomes available. In addition, if a tile processor or other element attempts to read from one of the networks, or a crossbar attempts to route data, and no data is available, the element trying to read data will stall on the read operation.

2.1 Off-chip Routing

A message can be sent off the chip at the edges by setting the "Funny" bits in the header word. However, using funny bits alone to do external routing restricts us to only 16 destinations off the chip. Therefore, to extend the number of devices that can be attached to the chip, an additional hierarchical routing scheme is used on each RAW port. As shown in Table 2.2, devices can be classified as type 0 or 1, depending on the user field used in the MDN messages used to communicate with them. For additional routing, however, an extra "extended header" word is used. The format of this extended header word is shown in Figure 2-2. The "tag" field of the header word is not used for routing, but can be used by the software to identify individual messages. The "sub-port ID" field is used to do additional routing. Thus, using the sub-port IDs and the two device classes in the MDN header, we can attach, in theory, up to 256 devices per port of the RAW chip. The ACK bit in the extended header is set if the sending element (either a device or a tile) wishes to receive an acknowledgment to the message. If the ACK bit is set, then the sender must include, at the end of the message, a "from header" that allows the target to locate the sending device. All external devices, including DRAMs, are required to be able to process extended headers and generate acknowledgments. Examples of ACK requests and general ACKs are shown in Figures 2-3 and 2-4, respectively. The use of the extended headers for deadlock purposes is discussed below.
Figure 2-2: Extended Header

Figure 2-3: A Device Related Message with an ACK requested

Figure 2-4: A General ACK
2.2 The RAW Deadlock Problem

As mentioned above, there are conditions under which elements in the network will stall. Since a processor itself is a network element (it can write into and read from any network), the processor pipeline itself can stall under these conditions. Unfortunately, this stall behavior can lead to deadlock in the processor. There are numerous situations in which this deadlock is possible. Consider, for example, the following scenario. Tile A has a cache miss on a load instruction, so it sends a cache line read request message (see Table 2.2) to the DRAM controller. At the same time, however, other tiles in the system are busy sending a very large stream of store requests to the same DRAM port, and as a result, the input buffers of the DRAM port overflow. At this point, a read request from tile B arrives at the crossbar of the tile connected to the DRAM port. Because the buffers leading into the DRAM are full, the read request cannot be routed into the DRAM. However, because the crossbar on that tile to which the DRAM is connected is locked into trying to route data into the DRAM controller, it is impossible for the controller to send replies back out to the chip. Thus, tiles A and B are deadlocked, since both tiles will stall permanently while waiting for the reply to their load requests. Numerous other examples can be constructed of deadlock in either of the dynamic networks. In addition, it is possible to have deadlock in the static networks, although it is assumed that, if the compiler can schedule all routes correctly, this will not occur. Thus we are primarily concerned with deadlock in the dynamic networks.

If we step back for a moment, we realize that we can attribute any deadlock in any of the networks to two basic causes: unsolicited communication and "un-sinkable" communication. The term "unsolicited communication" refers to any message that the receiver did not request, and, hence, will not try to read off the network. "Unsinkable communication" refers to any message for which there is no space in the target’s input buffer. Such messages will back up into the network and could potentially lock the crossbar into a certain route, preventing any other messages from traversing that tile. The two problems are related; for example, a message is likely to
be un-sinkable if previous messages to the same target were unsolicited and occupy buffer space because the target never reads them. If we can somehow ensure that these two conditions never arise, or if we can eliminate them when they occur, we can prevent a permanent deadlock of the chip.

There are two general approaches to dealing with deadlock. One is to try to determine when deadlock has occurred and then run some special routines to eliminate it, and the other is to simply avoid deadlock altogether. Currently, we use a deadlock detection and recovery scheme on the GDN. Each tile has a watchdog timer that is decremented on every cycle in which the processor has stalled while there is data available on the GDN input port. If the watchdog timer goes to 0, we assume that deadlock has occurred, and a timer interrupt occurs in the tile processor. The interrupt handler for the timer interrupt reads all available data out of the tile’s GDN input buffers and stores it in memory. It then sets the GDN_REFIL flag in a special-purpose register. If this flag is set, then any attempt to read from the GDN causes a GDN_REFILL interrupt, and the handler for this interrupt causes data to be read from the queue in memory rather than from the network itself. By un-clogging the network, we allow continued forward progress on the chip. Even though this scheme does nothing to address the problems in the user-level software that cause the deadlock in the first place, it does ensure that all messages on the GDN will eventually be sunk (by the interrupt handler, if necessary).

Because the MDN is used for such high-priority communication, however, we cannot afford to use a deadlock detection scheme like the one used in the GDN. We have therefore developed a scheme to avoid deadlock on the MDN entirely. As suggested above, this scheme relies on ensuring that no communication over the MDN is ever unsolicited and that all communication over the MDN is sinkable. Since the DRAM controller is designed to always listen for messages from the chip, any read or write request sent to the DRAM is by default a solicited message. However, deadlock can still occur if messages are sent to the DRAM at a faster rate than the rate at which they can be processed. Therefore, some kind of rate limit must be imposed. The rate limit on cache line read requests is automatically set at one request per tile,
since when a tile issues a cache line read request, it stalls until the response comes back. However, because the cache is non-write allocate, a limit must be imposed on the number of stores each tile is allowed to have outstanding at a time.

In order to achieve this, a system of store acknowledgments (store ACKs) is used. The cache control hardware on each tile maintains a counter that is decremented whenever a store is issued. If the counter ever reaches 0, the processor will stall on a store instruction. When the DRAM processes a store request, it sends a store acknowledgment header (see Table 2.2) back to the tile. When the header arrives at the tile, the store counter is incremented. Each tile can have up to 4 stores outstanding at once. Clearly, a cache line read response is both solicited and sinkable, since it is issued in response to a read request by the tile. And, although the tile software does not need to explicitly listen for a store acknowledgment, because the cache control hardware is designed to listen for one, store ACKs are also solicited.

In addition to avoiding deadlock in communication with the memory, we must also avoid deadlock when communicating with external devices. Fortunately, we can do this very easily by using the acknowledgment system that is made available through the use of extended headers. Suppose some element A (either a device or a tile) is communicating with external device B. A has a certain amount of space dedicated to it in B’s input buffers (these allocations are assigned by the operating system). By requesting acknowledgments for messages sent to B, A can keep track of how many spaces it has available in B. Furthermore, because messages are guaranteed to be delivered in order, it is not necessary to request an acknowledgment for every message. Instead, we can use what is known as a “Lazy ACK” system. Assume that A has space for 5 messages in B’s input buffer. A can send 2 messages, then send a third with an ACK requested, then send two more. When the ACK finally arrives at A, A knows that there are 3 spaces available in B, since the ACK would not have come back until the third message had been read. If we further impose the condition that all external devices must always be in “receive” mode (unless their buffers are full), we can be assured that all communication with external devices is both solicited and sinkable.
The model mentioned above works for external devices only because we are assuming that all external devices are permanently in "receive" mode. This guarantees that all messages to external devices are solicited. Furthermore, because a device always solicits messages, we can determine whether or not there is buffer space available in the device simply by interrogating it using the Lazy ACK system. However, this model clearly breaks down in the case of messages to tiles, because the tile processor will not attempt to read a message from the network unless it encounters an instruction to do so. If a tile initiates communication with a device, we can assume that the device driver is soliciting a response from the device, and hence that the software on the tile will read all of the data out of the network. However, for a device to initiate communication with a tile, it must send an interrupt, which is by its very nature an unsolicited message. Therefore, we must find some way to take an unsolicited message to the target tile without deadlocking the network.

Fortunately, the MDN interrupt header allows us to do exactly that. Not only is it automatically sinkable, since it is a header word, but because the hardware on the tile is designed to respond to it automatically, it is effectively solicited as well. The MDN interrupt represents the core of the RAW external interrupt system, although it is by no means sufficient for the task. After all, the MDN interrupt header can only tell the tile that some event has occurred; it carries no information about which event it was. We cannot simply have a device send an interrupt along with the header, because this scheme would be prone to deadlock; if external interrupts were disabled on the target tile, then although the header word would be read off the network, the interrupt itself would still occupy buffer space. It is therefore necessary to build a separate device, the Interrupt Controller, which can somehow get an interrupt to a tile in a deadlock-free fashion.
Chapter 3

Overview of the Interrupt Problem

In order for a processor to interact with external devices such as keyboards, mice, or PCI devices, the processor must have a mechanism for knowing when an external device wishes to communicate with it. In most cases, it is impossible to know in advance when an external device will need to send data to a processor. Data from a keyboard or mouse, for example, will be sent only when the user presses a key or clicks a button, and an Ethernet card will only have a new data packet available if some other entity on the network chooses to send data. Even in cases where the processor requests data from some device, it is often difficult to predict the precise moment at which the data will be returned. This is especially true in a multi-processor environment like RAW, where the latency of a request/reply sequence for one processor is a function of how many different processors have made requests. While it is possible, and perhaps necessary in some cases, to have the processor simply sit and wait for the reply, such a scheme uses processor time that might be better used for other tasks.

The two obvious solutions to this problem are periodic polling and external interrupts. Under a polling scheme, the processor would interrogate its external devices at regular intervals to check if the devices have had an event to which the processor needs to respond. However, a polling scheme is prohibitively expensive, especially in a system like RAW. In addition to the processor overhead required to periodically switch to kernel mode and execute the device interrogation routines, a polling scheme on RAW would cause a severe spike in traffic on the MDN as processors send messages
to, and wait replies from, external devices to see if they need to be serviced. The best solution, therefore, is a system of external interrupts, in which a device can inform the processor whenever a condition occurs that requires the processor's attention.

3.1 Interrupt Control in a Multi-Pentium System: The Advanced Programmable Interrupt Controller (APIC)

In order to develop a sense of the issues that would need to be addressed in the RAW interrupt controller, it was useful to study how the problem has been handled in existing multi-processor systems. Particular attention was paid to the Intel Advanced Programmable Interrupt Controller (APIC). There are a number of reasons for this choice. Firstly, documentation on the APIC is readily available, so it was the most convenient starting point for researching the basic problems of this thesis. In addition, the Pentium family of processors is the most widely used family in the world, and there is extensive documentation available on how the APIC is used in practice. In a multi-Pentium system, there is one local APIC for each processor in the system, and a single I/O APIC that links the processors to shared external devices. The local and I/O APICs are linked to each other by a dedicated 3-bit APIC bus. Devices request interrupts via the Interrupt Request (IRQ) pin by which they are connected to an APIC. Both the local APICs and the I/O APIC have IRQ lines that can be connected to external devices; thus, devices that only need to send interrupts to a specific processor in the system can be connected directly to that processor's local APIC, bypassing the I/O APIC entirely [4].

Devices request interrupts by sending the APIC a signal on an IRQ line. Signaling on an IRQ line can be accomplished in one of two ways. In one scheme, known as "edge-triggering," the external device requests an interrupt by pulsing the IRQ line high for one clock cycle. This signaling method normally used in the case where a single device is occupying an IRQ line. The second scheme, known as "level"
triggering, is commonly used when devices share an IRQ line. In many motherboards, for example, the PCI bridge is assigned only one IRQ line, which it must share among all the PCI devices. In this case, the PCI device will signal that one or more interrupts has occurred by pulling the IRQ line high. If the processor finishes processing one PCI interrupt but sees that the line is still low afterwards, it knows that there might be another device that needs to be serviced, and communicates with the PCI bridge to determine which device to attend to next [4, p. 7-19].

In order to distinguish between different interrupts, each IRQ is mapped to a number from 0 through 255. This number is known as the interrupt “vector”. When an interrupt is requested on a local APIC’s IRQ line, the APIC sets the bit corresponding to that vector in a 256-bit register known as the Interrupt Request Register (IRR). Periodically, the processor will issue an Interrupt Acknowledge (INTA) message to the APIC, which causes the APIC to scan the IRR, find the highest-priority bit that is currently set, clear the bit, and set the same bit in the In Service Register (ISR). The processor then reads the ISR to find out which vector it is expected to handle. Once the processor is finished handling the interrupt, it sends an End of Interrupt (EOI) message to the APIC, which clears the bit in the ISR [4, p. 7-17].

If the interrupt is requested on one of the I/O APIC’s IRQs, the I/O APIC must route the corresponding interrupt vector to the appropriate processor. Routing can be done either “statically” or “dynamically”, depending on how that particular vector is configured. In the static routing scheme, the APIC sends a message over the APIC bus to a specific, pre-programmed processor. The I/O APIC targets the delivery message to a specific processor by broadcasting the ID of the APIC to which this message is to be delivered. The I/O APIC can use either a physical ID, which is a unique, hard-coded value for each APIC, or a logical ID, which can be programmed by software. In the dynamic routing scheme, the I/O APIC chooses the lowest-priority processor and sends the vector to that processor. Priorities among processors are determined by the OS, and are generally based on processor load, with the lowest-priority processor being the least busy [4, p. 7-28].

One the I/O APIC has determined the target tile or tiles, it sets the bit corre-
sponding to that vector in the ISR and sends the vector on the data bus that links the local APICs and the I/O APIC. When the target APIC receives the message, it sets the corresponding bit in the IRR. Then, as in the case of interrupts signaled directly to it, the APIC delivers the interrupt to the processor to be serviced by setting the corresponding bit in the ISR. Once again, when the processor has finished handling the interrupt, it sends an EOI message back to the local APIC, which clears the bit for that vector in the ISR. At this point, if the interrupt was a level-triggered interrupt, the local APIC must send an EOI message over the interrupt bus back to the I/O APIC. This is to let the I/O APIC know that one of the interrupts signaled on that IRQ line has been serviced, and that the I/O APIC should check for additional interrupts on that line. It is not necessary to send an EOI in the case of an edge-triggered interrupt since these interrupts cannot be shared. The I/O APIC will always forward edge-triggered interrupts; if these interrupts arrive at an I/O APIC while the bit for the corresponding vector is set in the IRR, the APIC will simply ignore the message [1, p. 334-335]

Vectors are split into 16 priority ranges based on their number, with vectors 0-15 being the highest-priority group. By choosing the mapping of IRQ lines to vectors, the OS can establish a priority among the different devices in the system. The priority of a vector is used to determine the order in which they are delivered to the processor in the case that multiple interrupts are pending. If more than one interrupt is pending in the IRR of a processor’s local APIC, the bit corresponding to the highest-priority vector is set in the ISR when the processor sends an INTA message to the local APIC. Once a bit is set in the ISR, no bit corresponding to a vector of the same or lower priority can be set in the ISR. If, however, a higher-priority vector becomes available, that bit will be set. When the processor services an interrupt, it selects the highest-priority vector of all vectors set in the ISR [4, p. 7-17].

In addition to interrupts from external devices, the Intel interrupt system also handles a special class of interrupts called Inter-Processor-Interrupts (IPI). IPI are used when one processor in the system needs to alert another processor of an event. For example, an IPI could be used by one processor to indicate to another processor
that some data is available in a certain memory location, or to allow one processor to activate or de-activate threads on another processor. Inter-Processor-Interrupts are handled by the local APICs themselves, and do not rely on the I/O APIC. If processor A wishes to interrupt processor B, it issues the appropriate instructions to its local APIC, which sends an IPI message over the APIC bus to the target processor [4, p. 7-32].

Depending on the OS and on processor load, there could be a significant delay between when an interrupt occurs and when the processor is able to service it. During this time, it is quite possible that the same device has had another event and needs to interrupt the processor again. Rather than try to register all of these events independently, which would require an un-bounded amount of storage, the APIC simply ignores the extra interrupts by making all interrupts “sticky”. As soon as the local APIC receives a certain vector, the corresponding bit in the Interrupt Request Register is set. If, while that vector is waiting to be serviced, additional interrupts with that same vector are received by the local APIC, they are simply ignored, since a record already exists (in the IRR) that one or more interrupts with that vector have been requested. When the processor services the interrupt, it will communicate with the device using the appropriate driver. It is the responsibility of the device and the driver to figure out how many events actually occurred, and deal with them accordingly. Note that stickiness is irrelevant for level-triggered interrupts, since a level-triggered interrupt is signaled by pulling a line high, rather than simply pulsing it.

3.2 Challenges on the RAW Platform

The study of the Intel I/O APIC helped identify the 4 major issues that must be addressed in any interrupt control system for a multi-processor environment: interrupt/device identification, routing, queuing, and prioritization. The first problem refers to the need to figure out which event occurred on which device. The second issue, routing, refers to the task of informing the appropriate tile that a certain event
has occurred. The third issue, queuing, refers to the problem of storing interrupt vectors until they can be processed by the target processor. The fourth issue, prioritization, refers to the need to establish a priority among interrupts. In addition to solving these problems, any interrupt control system in a multi-processor environment must have a mechanism that allows the mappings between devices, vectors, and tiles to be programmed easily by the OS.

Although the study of the Intel APIC was useful for identifying the problems mentioned above, it proved to be of very little value in guiding the approach to these issues on the RAW processor. This is primarily because of the radical differences between the RAW processor and a multi-Pentium system. The most glaring issue is the difference in topology between the RAW chip and a multi-Pentium system. In the Intel system, the local and I/O APICs are all linked to each other by a single interrupt bus. Thus, incoming interrupt vectors can be broadcast, and a local APIC can simply choose to ignore the message if the processor to which it is attached is not the intended target. In contrast, the RAW communication network uses a mesh topology, which means that a broadcast from the interrupt controller would require the transmission of a separate message to each tile. Therefore, unlike in the Intel scheme, it is necessary for the IC to route a vector directly to the tile that is responsible for handling it. Secondly, the individual processors in the RAW fabric do not have local interrupt controllers, unlike the Pentium. In fact, the only hardware support for external interrupts that exists on the tile processors is the ability to respond to an MDN interrupt header. Thus, any prioritization of interrupts, mapping of vectors to devices, and assignment of vectors to tiles must be done either in software, in the interrupt controller, or in the devices themselves. Thirdly, the RAW architecture presents the added complication of having the account for deadlock in interrupt delivery.

Finally, in addition to external interrupts, the RAW interrupt controller must also handle Inter-Processor Interrupts, since there are no local APICs in RAW that can handle them. In RAW, IPIs are needed to allow unscheduled messages to be passed between processors at the system level. We could, in theory, have used the GDN for this purpose, since the GDN has certain hardware mechanisms that could
be used to achieve deadlock-free communication. In addition to the watchdog timer system mentioned above, the GDN also supports another interrupt, the GDN AVAIL interrupt. This interrupt occurs whenever a new message is available on the GDN, and can be used to ensure that tiles will read every message that is sent to them on the GDN. However, because we have elected to dedicate the GDN entirely to user-level dynamic communication, it would be inappropriate to use it for system-level dynamic communication. Thus, we must find another mechanism to support IPI, and the most logical choice seems to be to use the Interrupt Controller for this purpose. Although a signal from another tile is not an external event, it makes no difference from the standpoint of deadlock avoidance. An unsolicited message from a tile can cause deadlock just as easily as an unsolicited message from a device. Therefore, since the constraints on delivery of IPIs are nearly identical to the constraints on external interrupts, it makes sense to design the Interrupt Controller so that it can handle IPI, as well.
Chapter 4

Initial Design

As mentioned in Section 2.2, the interrupt control system in RAW is designed around the MDN interrupt header. Using the MDN header as the core, we can develop a mechanism for delivering interrupt vectors to tiles in a deadlock-free manner. The interrupt delivery process can be broken down into 5 stages: interrupt request, queuing, notification, vector delivery, and interrupt processing. Over the course of this research, two competing mechanisms were designed. The initial scheme was known as the “on-board storage” scheme, since all vectors would be stored on the controller itself. This idea was a natural consequence of our research into the Intel controller, which queues all vectors on the I/O APIC before delivering them to the local APICs. However, the RAW controller must support a vastly larger number of devices and interrupts than the Intel controller (at least in theory). Furthermore, since there is no local APIC on a RAW tile, the Interrupt Controller may need to store vectors for long time before they can be read by their target tiles. This means that the limited storage space available to the interrupt controller could be a serious issue. Therefore, a second scheme, known as the “DMA scheme,” was developed. In this model, the Interrupt Controller does not store vectors, but rather routes them to the DRAM. The two schemes were both evaluated using software simulations. Eventually the DMA scheme was chosen because of its scalability and ease of implementation in hardware. The two schemes, and the techniques used to evaluate them, are described below.
4.1 Interrupt Notification via the MDN Interrupt Header

The interrupt request phase refers to the act of getting the interrupt request from the device to the controller. This step includes any processing that must be done to map devices to interrupt vectors. Because the two designs differ only in how and where vectors are queued after they reach the controller, the interrupt notification stage works identically in both versions.

The current version of the RAW motherboard has no physical IRQ wires connected directly to the interrupt controller. Instead, a device on the RAW motherboard requests an interrupt by sending a message over the MDN to the IC. In this case, a device requesting an interrupt will send the IC a message with the target tile number and the vector number for the interrupt. Even though the in-band delivery scheme increases the delay between when an event occurs and when the request is received by the IC, the overall effect on the total time to deliver a vector to a tile is small. This is because, as shown in section 4.5, the bulk of the overall taken is still in other stages of the process. In addition, using the MDN to request interrupts allows us to encode vastly more information about the event than we can using just a physical IRQ wire. Since an entire 32-bit word is used as the vector.

In order to allow commercially available devices to send interrupt requests over the MDN, a special device called an External Interrupt Transmitter (EIT) has been designed to receive interrupt requests from devices and forward them to the IC over the MDN. The EIT has four “sticky” IRQ pins, similar to the IRQ pins on the Intel I/O APIC. Any device can be made to request interrupts through the EIT on a port by wiring its IRQ line to the one of the EIT’s IRQ pins. Interrupt identification and assignment of vectors to tiles is handled in the External Interrupt Transmitter itself. For each of the four devices that can connect to the EIT, the operating system programs a vector and the target tile for interrupts from that device. In addition, the operating system also programs the headers used by the EIT to send an interrupt request to the IC. Thus, all of the information needed to allow the EIT to send
interrupts is re-configurable by software. When the EIT receives an interrupt request on one of its IRQ pins, it sends a message to the interrupt controller containing the target tile number and the vector.

4.2 Vector Queuing on the Interrupt Controller

Once the interrupt request arrives at the Controller, it is queued pending delivery to the target tile. The controller can be broken up into three main modules: the receiver, the transmitter, and the 16 Vector Storage Units (VSU) that are responsible for queuing vectors for each tile. A logical block diagram of the controller is shown in Figure 4-1. The VSU for each tile maintains a state counter that indicates the status of interrupt delivery for each tile. It also stores the MDN interrupt header used to interrupt the tile. When the queue is empty, the VSU for a tile is in state 0. When the controller receives an interrupt request for a tile, the requested vector is written into the queue in the VSU. The VSU is now in state 1. When the round-robin scheduler next focuses on this tile, the scheduler will see that there are interrupts pending but that the notification has not yet been sent. At this point, the transmitter schedules the tile’s notification header for transmission and puts the VSU into state 2.

4.2.1 Notification and Vector Request

To notify the tile that an interrupt is pending, the controller simply reads the tile’s interrupt header from the VSU and transmits it over the MDN. At this point, the VSU is now in state 2 - a notification has been sent, but the tile has not yet responded with a vector request. During this time, it is possible that additional interrupts will be requested for the tile. In this case, the new interrupts are added to the queue, but it is not necessary to send an additional notification to the tile. When the tile requests a vector (step 4), all pending vectors will be returned.

As described in section 2, when the MDN header reaches the tile, the External Interrupt bit is set in the tile’s EX.BITS register. If this bit is not masked, the processor will take an interrupt and enter the external interrupt handler. At this
Figure 4-1: Logical Block Diagram of the “On-board” Controller
point, the interrupt issues a vector read request from the tile. In order for the vector request to be processed properly, the interrupt handler must be written in such a way as to ensure that, once the vector read request is made, no cache misses can occur until the all vectors have been received from the controller. The reason for this is that, if a cache miss occurs after the request is made, there is no guarantee that the data from the cache will arrive before the data from the interrupt controller. If this happens, the wrong data will be written into the cache. The interrupt handling software is allowed to execute any number of operations between the time it issues the request and the time it attempts to read the vectors out of the MDN, provided none of them have the potential for a cache miss.

4.2.2 Vector Delivery

When the receiver receives a vector request from a tile whose VSU is in state 2, it signals the VSU to change to state 3. This state indicates that a request has been received and that the vectors should be transmitted. When the scheduler focuses on this tile again, it reads the stub of the interrupt vector delivery message from the VSU. This stub already has all of the coordinates needed to send the vectors to the tile. The transmitter computes the length field by taking the number of queued interrupts and adding 1, since the first word of the message is used to indicate how many vectors will be sent. In order to ensure fairness, only one vector delivery message can be sent per tile per round, so the maximum number of vectors that can be sent at any time is 30. Vectors are packaged into the message in the order in which they were received, and the entire message is then queued for transmission. If all vectors have been read from the queue, the VSU returns to state 0. If not, the FSM goes back to state 1. It is assumed that the interrupt handler is written in such a way as to read all of the vectors that are sent in the message.
4.2.3 Vector Processing and Prioritization

Once a vector has been transmitted to the tile, the interrupt controller itself is no longer responsible for it. Any further processing of the vector must take place in software. Although the Raw OS is still in its early stages, it is expected that the external interrupt handler will read the vectors off the network, write them into some location in memory, and then process them at some later point. It would be inefficient to process them as they are read - since the rest of the vectors will still be waiting on the input buffer, no cache transactions are allowed during this period. Note that the restriction on cache transactions during vector reads complicates the process of storing vectors for later use. There are not enough general-purpose registers to store all the incoming vectors. This means that either the switch memory will have to be used, or the cache tags will have to be re-programmed so that storing the vectors does not cause a cache miss.

Note that unlike the Intel APIC, the RAW interrupt controller does not attempt to establish a priority among the different interrupts. This does not mean that RAW does not use a prioritization scheme. However, it seems more efficient to handle that task in software than in hardware. The primary reason for this is that, even if multiple interrupts are pending for a tile, they are all delivered in the same vector delivery message. Therefore, the most logical way to handle prioritization is to allow the software on the tile to establish a priority after it has read all of the vectors from the message.

After the interrupt has been processed, the tile must indicate to the source of the interrupt that the interrupt has been processed. In the case of IPI, it is assumed that the software used to communicate between two tiles will perform whatever notification is necessary. In the case of a device, however, the tile must send an End of Interrupt (EOI) message to the EIT that sent the message. Once the EIT receives an EOI that corresponds to one of the 4 devices connected to it, it clears the interrupt pending bit for that device. The next time that device requests an interrupt, the vector will be sent to the controller again, since any interrupts that were requested prior to this
Figure 4-2: Logical Block Diagram of DMA Version

will already have been processed.

4.3 Vector Storage in the DRAM

Figure 4-2 shows a basic block diagram of the DRAM version of the interrupt controller. Note is that, rather than separate VSUs for each tile, there is simply a single Interrupt Storage Unit. For each tile, certain information, such as the MDN interrupt header used or the address in memory to which the vector should be written, is stored in holding registers. The function of this version of the controller is, in essence, simply to translate interrupt requests into DMA writes.
4.3.1 Queuing and Tile Notification

In this version of the controller, the interrupt request stage is identical to that of the on-board storage version. When interrupt requests arrive at the controller, however, they are buffered for translation. The translation step takes the vector from an interrupt request and packages it into a message that can be used to write it into the appropriate location in the memory of the tile. This translation requires 4 pieces of information for each tile: the MDN interrupt header, the header used to do write data into DRAM at the correct port, the base address of the vector queue in the tile’s address space, and a mask used to set the size of the queue.

When the transmitter reads a new request out of the receive buffer, the first item read is the tile number. The transmitter uses this number to look up that tile’s DMA write header (see Table 2.2). The first word that must be sent as part of the DMA write message is the address to which data is being written. To compute this address, the IC maintains a counter for each tile that is updated by 8 whenever a write occurs. This counter is masked, to limit the size of the queue, and then added to the base address to determine the write address.

Although the RAW memory controller allows DMA writes to any location in memory, the interrupt controller must align all writes with cache lines. Furthermore, every single vector must be written on its own cache line. The reason for this is that, once the interrupt controller dispatches a vector to the DRAM, it has no way of knowing when that vector is going to be read from the DRAM by the tile. When that vector is read from the DRAM, the entire 8-word cache line on which the vector resides is also read into the tile’s cache. If the interrupt controller were to attempt to write a subsequent vector to a location on the same cache line, it would cause a cache coherency problem.

After the address, the next word to be sent in the message is a mask indicating which of the subsequent bytes is actually to be written into DRAM. In this case, the mask is set to 0x000000FF, since only the first two bytes of the cache line are necessary. The next word to be sent is the value 1, which indicates that this cache line
contains a valid vector. As shown below, the tile needs some way of knowing when it has reached the end of the vector queue in DRAM, since it cannot receive that information a priori. It finds the end of the queue by reading cache lines until it finds one without the valid word set. Following the valid word, the vector itself is sent. Next, the IC sends the header for the EOI message that must be sent back to the EIT by the tile when the interrupt has been processed. Finally, in order to complete the DMA packet, 5 0’s are sent. Note that, in the final version of the controller, the EOI is not sent with the cache-line write. Rather, the OS is responsible for calculating what this message should be. For testing purposes, however, it was easier to just include it as part of the cache line.

Once the vector is sent to the memory, the tile must somehow be notified that the vector has been queued. The MDN interrupt header cannot be sent directly to the tile from the controller, because it is possible that the MDN interrupt header will reach the target tile before the DMA write message reaches the target port in DRAM. If this occurs, then there is a chance that the tile will try to read the vector out of DRAM before it is even written. In order to ensure that the write occurs before the read request from the tile arrives, the MDN interrupt header is relayed off of the DRAM itself back into the network. An opcode of 1110 in the MDN indicates that this is a relay message, and that all subsequent words in the message are to be injected back into the network. The subsequent words must be properly formatted messages, including headers, and like any other message type, the relay message can only carry 31 words. Because the DMA write and the relay message go to the same port, the Interrupt controller constructs the relay header by taking the DMA write header and changing the opcode and length fields. It then sends a relay message containing the tile’s interrupt header to the DRAM port.

### 4.3.2 Interrupt Processing

As with the previous design, the external interrupt handler on the tile is responsible for reading the vectors out of the queue and processing them. Now, however, the process of reading vectors is abstracted as a memory read. The tile is responsible for
keeping track of the address of the cache line that stores the next vector. When it enters the interrupt handler, it simply reads the first word of that cache line, which is the valid word, which will cause the rest of the line to be allocated into the cache. If the valid word is set to 1, the tile knows that this cache line has a valid vector on it, and it can read the vector from the next word in the cache line. The tile keeps reading successive cache lines out of memory until it finds a cache line whose valid word is set to 0. This is the only way that the tile can know when it has reached the end of the queue, since, unlike in the other design, there is no way to inform the tile beforehand how many vectors are pending for delivery.

Because all vector reads are now identical to memory accesses, there is no need to constrain the interrupt handler from taking a cache miss. Therefore, if the tile decides to write the vectors into some other queue in memory for processing at a later point, it can store them as they are read and not worry about whether or not the target address is in the cache. However, in order to ensure that the delivery process works correctly, certain restrictions are placed on accesses to the interrupt queue. Firstly, the tile must set the valid words of all cache lines in the interrupt queue to 0 on startup. Secondly, except for when a vector is being read out of memory, no address in the memory space allocated for the queue can ever be allowed to reside in the cache. This is to ensure that there is never any incoherency between the data in the cache and the data in the DRAM. Finally, once a vector has been read out of the vector queue in memory, the tile must re-set the valid word in that cache line back to 0. This is because the write address will eventually wrap around and re-use old locations in the queue. If, as expected, the interrupt handler takes vectors out of the vector queue and stores them for later use, the vectors must be stored in a different location - old vectors cannot be allowed to take up space in the delivery queue.

Finally, as in the previous design, the EIT or tile that sent an interrupt vector must be notified when the vector has been read from the queue, so that new requests can be made. Since the EIT is used in both schemes, the acknowledgment messages sent to the EIT are the same in both designs. In the tests used, the EOI messages were also used as store acknowledgments. Recall from section 2 that, as part of the
deadlock avoidance system in the memory dynamic network, all store requests from tiles must be acknowledged. In addition, although DMA writes are not automatically acknowledged, a device that uses DMA is responsible for actively limiting its own sending rate. Since interrupt requests are destined for the DRAM, we must treat them as store requests and rate limit them. Fortunately, the fact that interrupts are sticky imposes an automatic rate limit for each device that wishes to send interrupts. When a tile sends an interrupt acknowledgment to an EIT, it is in effect acknowledging that the write of the vector to DRAM has been completed. In the final version, however, the store acknowledgment is a separate message from the EOI.

4.4 Evaluation of Designs

4.4.1 Testing Methodology

In order to evaluate the two different options, both designs were simulated using models written in bC, a C-like simulation language designed for RAW. In addition to comparing the performance of each design, we also considered the scalability and complexity of the design when deciding which one to implement on the FPGA. Note that, because the hierarchical routing scheme mentioned in section 2 has only recently been developed, the test models used do not incorporate the hierarchical routing scheme. Therefore, any additional traffic that might be generated due to buffer management techniques was not accounted for in these tests.

Because there is no way to incorporate a physical device into the simulation environment, a bC model of a device/EIT combination was used to send interrupt requests to the interrupt controller via the MDN. The virtual EIT was attached to the MDN on port 0, the northern port of the tile in the northeast corner of the chip. The IC was attached to port 8, southern port on the southeastern tile of the chip. The choice of port 8 as the default port for the IC is a remnant of the previous deadlock recovery scheme for the MDN. Before the store acknowledgment system currently in use was developed, memory requests and vector read requests could only be sent to the east
or south, so port 8 was the only port from which the IC could serve all tiles. There was no reason to change the choice of port for the on-board version of the controller, and for the DMA version, latency should decrease if the IQD is closer to the DRAM. The fact that a single interrupt source is being used, rather than having multiple test devices, was assumed to have little bearing on the test results, since in either case the primary bottleneck in communication with the IC is the dynamic crossbar on tile 15, the southeastern tile.

All of the tiles on the chip, except for tile 0, had identical software. Each tile began by programming its information into the IC, and then went into a loop that accessed memory in such a way as to cause periodic cache misses. All tiles had an identical interrupt handler, which was written to read interrupts in a manner appropriate to the design being tested. Tile 0 was used as the control tile, and was responsible for starting and stopping the simulation at the appropriate times.

The tests varied both the number of tiles used and the number of interrupts sent to each tile. The primary quantitative measure of performance in our case is the average latency of interrupt delivery, defined as the time between when an external event occurs and when the target tile is aware of that specific event. In addition to measuring the latency of delivering single interrupts, we also simulated high-stress conditions by sending large streams of interrupts to the controller. In these situations, performance was compared by measuring the amount taken to deliver the entire stream of interrupts to the target. Qualitatively, we considered how much memory each scheme was likely to consume on the FPGA, in the worst case. For the on-board design, this is an important consideration because interrupt vectors must be queued on the FPGA itself. Yet even for the DMA version of the controller, some storage space must be used to buffer interrupt requests for translation into DMA writes. Finally, we also present a qualitative analysis of the potential for each design to be scaled for use on larger RAW fabrics, such as the 8x8 fabric that has recently come on-line (in simulation).
4.4.2 Results

In the first test we conducted, we examined the amount of time it took to send varying numbers of interrupts to a single tile while the entire chip was active. We simulated "activity" on the chip by having all chips execute code that would periodically cause cache misses. The results of the test are shown in Table 4.1.

<table>
<thead>
<tr>
<th># Interrupts</th>
<th>Stream Latency - DMA version (# cycles)</th>
<th>Stream Latency - On-board version (# cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>211</td>
<td>210</td>
</tr>
<tr>
<td>5</td>
<td>400</td>
<td>384</td>
</tr>
<tr>
<td>10</td>
<td>674</td>
<td>570</td>
</tr>
<tr>
<td>20</td>
<td>2844</td>
<td>1176</td>
</tr>
<tr>
<td>50</td>
<td>6011</td>
<td>2957</td>
</tr>
<tr>
<td>100</td>
<td>11422</td>
<td>5380</td>
</tr>
<tr>
<td>200</td>
<td>22256</td>
<td>10329</td>
</tr>
<tr>
<td>300</td>
<td>33077</td>
<td>15303</td>
</tr>
<tr>
<td>400</td>
<td>43951</td>
<td>20269</td>
</tr>
</tbody>
</table>

Table 4.1: Stream Latency, Single Tile

In addition to conducting a latency test with a single tile, we also examined stream latency when all 15 tiles were active. The results of these tests are shown in Table 4.2.

<table>
<thead>
<tr>
<th># Interrupts</th>
<th>Stream Latency - DMA version (# cycles)</th>
<th>Stream Latency - On-board version (# cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1002</td>
<td>605</td>
</tr>
<tr>
<td>5</td>
<td>1733</td>
<td>1168</td>
</tr>
<tr>
<td>10</td>
<td>3473</td>
<td>1902</td>
</tr>
<tr>
<td>20</td>
<td>6537</td>
<td>3361</td>
</tr>
<tr>
<td>50</td>
<td>15736</td>
<td>7772</td>
</tr>
</tbody>
</table>

Table 4.2: Stream Latency, Multiple Tiles

4.5 Performance Analysis

It is clear from looking at the data that the on-board storage version has an advantage in terms of interrupt latency. Of course, it is to be expected that the on-board version
will have this advantage. Not only does the DMA version translate a 2-word interrupt request message into an 12-word DMA write and interrupt header relay message, but the notification cannot be sent to the tile until after the entire cache line has been written to DRAM. In an attempt to improve the performance somewhat, the format of the vector write message was changed so that both the interrupt header and the DMA write were encapsulated in one relay message, with the interrupt header in front of the DMA write. But even this change improved the latency only marginally. In contrast, the on-board version of the IC can send a notification directly to the tile as soon as a vector has arrived, and can send 7 vectors straight to the tile in the amount of time it takes the DRAM to send a single cache line to the tile.

However, while it is useful to compare the performance of the two schemes under a variety of circumstances, we should note that, under most operating conditions, it is unlikely that we will encounter a situation in which large numbers of interrupts are outstanding at any one time. This is a reasonable assumption to make for a number of reasons. First of all, most external devices run at a much lower speed than the RAW chip itself; the PCI bus, for example, can run at a maximum speed of 66 MHz. This fact, combined with the "stickiness" of external interrupts, means that in the average case external interrupts will occur fairly infrequently relative to the speed of the processor. Even if we factor in IPI, and assume there is a device connected to every RAW port not dedicated to DRAM, it would require that every one of those elements request an interrupt almost simultaneously to produce the kind of load that would cause the performance of the on-board version to be double that of the DMA version. Secondly, it is unlikely that every device will be required to deal with external interrupts. Instead, the OS will probably designate a small set of tiles to handle communication with external devices.

One could argue that, because RAW is an extroverted architecture, we should expect a much larger volume of interrupt traffic that we would would in a conventional architecture. However, given the nature and capabilities of RAW, this argument seems flawed. The fact that more data will be flowing into the RAW chip does not necessarily mean that more interrupts will be occurring. Recall that, once a tile has been notified
that an external event has occurred, all subsequent communication related to that event is done directly between the tile and the device. If a device has a large stream of data for a tile, it does not need to send an interrupt for every individual word of the stream; it just needs to let the tile know that the stream is coming. In fact, if we have a configuration in which we know that certain devices will constantly be sending large streams of data, it would probably make more sense to bypass the interrupt controller and simply dedicate certain tiles to interfacing with those devices. The fact that RAW tiles are so cheap and easy to replicate makes such a scheme feasible.

Another issue to consider is the question of how much latency will be added by the OS itself. Although in our tests, we treat latency as the time between the event and the delivery of the vector, a more common measure of latency is the time between the event and when the OS can service the vector. Because the RAW OS has not yet been written, we have numeric data on the latency contributed by the OS. However, a brief study of existing architectures suggests that this added delay could dwarf the latency due to vector delivery. For example, in one analysis of interrupt latency, an 800 MHz Pentium-III processor running the Windows XP OS with real-time extensions had a best-case latency of 2 microseconds, which amounts to 1600 clock cycles [2]. Given that the hardware itself takes less than 40 cycles to deliver a vector to the local APIC, this data suggests that the differences in between the two schemes we studied may not be significant [4, p. F-3]. Rather, it seems that the bulk of the potential performance improvements are in the OS rather than in the IC itself.

4.5.1 Scalability

As the size of the raw fabric increases, the number of interrupts with which the interrupt controller has to deal increases dramatically. Therefore, scalability is at least as important a factor as performance, since a design that does not scale properly cannot guarantee reliable delivery of interrupts. With scalability taken into account, the DMA version of the controller may have certain advantages. The first claim we can make about the DMA version is that, for larger fabrics, it is much better able to deal with the increased storage requirement. It is true that, in theory, both
versions have a storage requirement with $O(n)$ growth in the number of tiles. This is because, at the minimum, we must allocate enough space on the IC for one interrupt message from every single tile and every single EIT in the system. However, the storage requirement for the DMA version is guaranteed to be less than that of the on-board version. Assume there are a total of $N$ elements (either tiles or EITs) that can send interrupts. Since the DMA version maintains one unified queue for all interrupt messages, it needs to allocate, at a minimum, enough space for $N$ interrupts. The on-board version, however, must maintain a separate queue for each tile. Furthermore, we cannot know a priori which devices will be assigned to interrupt which tiles. Therefore, assuming we have $M$ tiles and that no tile will attempt to interrupt itself, we must have enough storage space for $M(N-1)$ devices. In an 8x8 fabric, this would amount to a roughly 63-fold disparity in storage requirements.

Furthermore, in the DMA version, the rate at which new interrupts can be sent to the controller is a function of how quickly vectors can be written into the DRAM, rather than of how quickly they can be delivered to the target tile. Therefore, in a situation where external interrupts are disabled for an extended period of time, the DMA version would still be able to accept new interrupt requests and translate them into DMA writes, while the on-board version would be unable to do so. In fact, assuming no significant traffic backups occur at the DRAM ports, the DMA version could write all of the interrupts that occur during this period into the DRAM. This means that, when external interrupts are re-enabled, the OS will be able to read all interrupts that had occurred during this period in one execution of the interrupt handling routine. Of course, reading the vectors will in itself be a very high-latency operation on the DRAM version; one test showed that it takes 16,855 cycles to read a stream of 300 interrupts. However, in the on-board version, only a (potentially small) subset of the interrupts that may have occurred during this period will have reached the IC. This means that the OS will have to undergo multiple executions of the external interrupt handler to read all of the interrupts that may have occurred. This may significantly increase latency.

In addition, although the distributed nature of the interrupt storage did not appear
to make the DMA version competitive with the on-board version in terms of latency, there is great reason to believe that it will be a great advantage in larger fabrics. This is because, in an 8x8 fabric, for example, there are twice as many ports available on the edges of the RAW chip. It is likely that many of these extra ports will be used as additional DRAM ports, since the model of having only one DRAM port responsible for every row of the chip becomes increasingly strained as the edge length of the RAW chip increases. As the number of DRAM ports increases, and as the ports are more distributed over the perimeter of the chip, there will be fewer major bottlenecks in the delivery system, and the latency should benefit as a result.

Not only will the DMA version benefit from reduced bottlenecks as a result of the distribution of memory, it will also benefit from the fact that tiles will be physically closer to the DRAM ports from which they read their vectors. Although the interrupt controller is the single point for interrupt requests in the RAW system, in the DMA version it is not the only point of vector storage. Since a tile is mated with the DRAM port that is closest to it (using the Manhattan metric), most tiles will be closer to their mated DRAM port than they will be to the interrupt controller. While in the 4x4 case this difference in distance may not be of any importance, the difference can grow much larger as the edge length increases and as DRAM ports become distributed over more portions of the chip. In the on-board version, however, there is still only one point from which tiles can read their vectors, so tiles very far from the interrupt controller will pay a significant penalty.

Lastly, in terms of the hardware involved, the complexity of the DMA version is much less than that of the on-board version. The on-board version must maintain a separate queue and state machine for each tile, and must be able to receive and process the vector read requests in addition to interrupt requests, programming messages, etc. In contrast, the DMA version maintains one unified queue, and stores no state for any tile beyond some header words and a write address. If the RAW clock speed increases in the future, speeding up the DMA version to keep up will be easier than speeding up the on-board version. Most importantly, however, is the possibility that the interrupt controllers themselves can be replicated and distributed on larger RAW chips. By
distributing interrupt controllers as well as DRAMs, it will be possible to place a tile, the devices that need to interrupt it, its DRAM, and its mated interrupt controller close to each other, which would greatly improve latency. Although replication could in theory be used with either scheme, the comparative simplicity of the hardware in the DMA version, combined with the smaller storage requirement, make it the more attractive option for larger fabrics.

4.6 Conclusion

Although the on-board version of the interrupt controller exhibited better performance in the tests used, the potential scalability problems posed by that design led to the choice of the DMA version of the controller as the current design. The DMA version was not only easier and faster to implement in the near term, it was also the version more likely to remain useful to us in the long term. If in the future a method is found to overcome the scalability problems encountered in the on-board version it will most likely be prudent to abandon the DMA design and switch to the on-board design. As of the writing of this paper, however, the raw OS has not yet been implemented, and RAW motherboard has only recently come online. Therefore, given that the requirements of the system may vary considerably as more progress is made in other areas of the RAW chip, it makes more sense to use the design that has smaller resource requirements and can be more easily changed.
Chapter 5

Hardware Implementation and Software Conventions

As mentioned above, the version of the Interrupt Controller selected for implementation in hardware is the so-called DMA version. This overall scheme depends on the Interrupt Controller and the External Interrupt Transmitter, both of which are implemented on a Xilinx Virtex-E FPGA, and a set of software conventions that are used by the RAW OS to program the hardware and read vectors correctly. This section describes the design, implementation, and testing of these three components.

5.1 Interrupt Controller Hardware Implementation

The overall block diagram for the Interrupt Controller is shown in Figure 5.1. The controller has seven major units: the I/O unit, the transmission FSM, the reception FSM, the tile information modules, the controller information module, and the interrupt storage unit. The I/O unit is responsible for communication with the RAW port using the SIB protocol described in section 2. The reception FSM is responsible for processing any incoming messages, such as programming messages and interrupt requests. The transmission FSM is responsible for translating interrupt requests into DMA write packets and queuing the new messages in the send buffer. The tile information modules hold the DMA write headers and write address infor-
mation for each tile, while the storage unit is responsible for storing the interrupt requests that are to be converted by the translation unit. Finally, the controller information module stores information specific to the controller, including its location (coordinates and sub-port ID), device ID, etc. Although the RAW chip itself runs at 300 MHz, at present the Interrupt Controller is only required to run at 150 MHz. The controller connects to the chip through a special interface known as an "I/O Gasket," which takes messages from the RAW chip and steps them down to 150 MHz, and likewise takes signals from the Interrupt Controller and the PCI controller and steps them up to 300 MHz.

5.1.1 I/O unit

A block diagram of the I/O unit is shown in Figure 5.1. The I/O receives a data word, a yummy signal, a thanks signal, and a select signal from the RAW port. The 2-bit select signal serves the purpose of the ValidIn bit in the SIB protocol. Because the GDN, MDN, and first static network are all multiplexed onto the same set of I/O pins, the select signal is used to indicate which of the three networks is being used to transmit. A value of 2 on the select input indicates that the word on the output port is meant for the MDN, and a value of 0 indicates that no data is available. The I/O multiplexor has separate yummy and thanks signals for each of the three different networks. The thanks signal is used to signal to the IC that the value currently being presented has been read, while the yummy signal allows the IC to keep track of how much space is left in the I/O multiplexor's MDN input buffer. The I/O unit presents the RAW port with a data word to be sent into the MDN, a select signal to indicate that data is available for the MDN, a yummy signal, and a thanks signal.

When a data value is presented to the I/O unit by the RAW chip, both the value and the 2nd bit of the select input are registered. The "thanks" output to the RAW port is simply the second bit of the SELECT input, since the I/O unit automatically reads any data value presented by the multiplexor. The thanks signal is generated by taking the AND of the registered thanks signal and the SPACE_AVAIL signal provided by the storage unit. This is because, if there is space available in the storage unit,
Figure 5-1: Overall Block Diagram of IC Hardware
Figure 5-2: I/O unit
it is guaranteed that the reception FSM will consume the data value stored in the
INDATA_HOLDING_REG register. In order to keep track of how many spaces are
available in the I/O multiplexor's MDN input buffer, the I/O unit maintains a counter
(the CHIP_BUFFER_COUNTER signal in figure 9) that is decremented whenever a
data value is transmitted and incremented whenever the I/O unit receives a yummy
signal from the RAW port.

When the send buffer (see section 5.1.6) has data to be sent over the MDN to
the RAW port, it asserts the INTSAVAIL signal. When INTSAVAIL is asserted, the
value that is to be sent over the MDN is the value currently on the output pins of the
RAM in the send buffer. This signal is registered (INSTAVAIL_REG in figure 9) and
then used to compute the 2nd bit of the SELOUT signal, which is the select input
of the RAW port. The INTSAVAIL_REG signal holds its value until the CONSUME
signal is asserted, to ensure that, if the value cannot be sent on this cycle, it is not
lost. As with the select output signal from the RAW port, the select input is 2 if there
is data available for the MDN on the input pins. In this case, the data presented to
the RAW port by the I/O unit should be read if and only if the INTSAVAIL_REG
signal and high and there is space available in the RAW port's input buffer.

In order to determine whether or not there is space available in the MDN in-
put buffer, the value of the CHIP_BUFFER_COUNTER register is used. This reg-
ister holds the number of buffer spaces the I/O unit perceives as being available
in the target's queue. However, because of timing constraints, we cannot simply
compare the value of CHIP_BUFFER_COUNTER to 0 and then AND it with the
INTSAVAIL_REG signal to determine the value of SELOUT[1]. The reason is that
the combined propagation delay of the CHIP_BUFFER_COUNTER register, the
comparison unit, and the AND gate is greater than the minimum clock period.
Instead, we must speculate, based on signals that arrived on the previous cycle,
whether or not the counter will be 0 on this cycle. The DOWN_CHOICE_REG
register is high if the I/O unit decided, on the previous cycle, to decrement the
counter, while the CHIP_COUNTER_DOWN_REG signal is high if the old value
of the counter on the previous cycle was 1. By taking the AND of these two sig-
nals, we can determine whether or not the counter is 0 on this cycle in less time than it would take to compare the value of the counter to 0. In addition, if the counter was 0 on the previous cycle and will stay 0 on this cycle, we must know this in advance. We can calculate this, however, by checking if, on the previous cycle, the counter was 0 and no yummy signal was received. This value is registered as the signal COUNTER_STAYS_0.REG. Thus, we can compute SELOUT[1] in the required amount of time by computing the function (INTSAVAIL_REG) & !(DOWN_CHOICE_REG & CHIP_COUNTER_DOWN_REG) & !(COUNTER_STAYS_0_REG), as shown in the figure.

In addition to informing the RAW chip that a word is available, the I/O unit must also request the next word to be sent from the send buffer. In theory, the conditions under which we would want to request a new word to be sent are the same as the conditions under which we send the current word out into the RAW chip. However, because the DRAM does reads synchronously, we must account for the extra delay of one clock cycle when requesting a word. If the I/O unit requests that the read address of the send buffer be incremented, it will take one clock edge to increment the address, and a second clock edge to actually get the new data. Therefore, to ensure that there are no “bubbles” in the transmission of words, the SEND-word signal, which is used to request a new word from the send buffer, is computed in the same manner as SELOUT[1], with the exception that INTSAVAIL itself, rather than INTSAVAIL_REG, is used. This way, the SELOUT[1] signal can be asserted on the same clock edge that changes the read address, so that on the next clock edge, the old data can be read by the RAW port and the new data can be presented on the output of the memory.

5.1.2 Receive FSM

The data word and valid signal (INDATA_HOLDING_REG and VALID_HOLDING_REG, respectively, in Figure 5-2) are registered in the I/O unit are presented to the Receive FSM, where they are again registered before being used as FSM inputs. A state transition diagram for the Receive FSM is shown in Figure 5-3. The basic function of the...
Note: in this figure, the inputs to the FSM are shown as \{data_{avail}, opcode, AREQ, and ACK\}. The opcode value is taken from bits \{13:7\} of datain. AREQ and ACK are taken from bits 15 and 14, respectively.

Also note that only those outputs which are high in a given state are shown.

Figure 5-3: Receive FSM

receive FSM is to listen for different message types and then process them as necessary. There are 6 different types of messages that can be processed: tile information programming messages, ID request, location programming, buffer allocation, store ACKS from DRAM, and interrupt request. The FSM recognizes different messages based on the opcode field of the extended header (see Table 5.1).

While the receive FSM is in the state 0, reset state, it simply waits until a new message is available. If, while in the reset state, the FSM sees that data is available, it assumes that this word is an MDN header. Since the MDN header itself is not
used to determine message type, the FSM simply moves to state 1 and waits for the extended header. When the extended header arrives, the FSM examines the value of the opcode field. If the opcode is 32, it indicates that this message was a request for device identification (see Figure 5-4). The FSM then moves into state 2, where it waits for the “from” header to arrive. Any tile requesting a device identification must send a “from” header as part of the request, since the ID request is a message that will be acknowledged. When the header arrives, the FSM loads it into the ID.REPLY register (shown in figure 12) by asserting ID.REPLY_WE, and then sets the PING.REQUEST register (see figure 12) by asserting the SET.PING signal. It then moves back into the reset state. The PING.REQUEST signal the Transmit FSM (see section 5.1.6) that it should send an ID message to the requesting tile. We assume that the OS will request IDs on a very infrequent basis - therefore, no attempt is made to keep successive ID requests from overwriting each other.

If the opcode in the extended header is 33, then the FSM knows that the next word in the message will be the Locator word (see Figure 5-5). The format of this word is identical to that of a “from” header, and includes all information needed to
uniquely locate the device on the chip. When the locator word arrives, the FSM writes it into the IC information module by asserting the LOCATION_WE signal, and then returns to the reset state. If the opcode is 34, the message is a buffer allocation message (see Figure 5-6). The next word in the message will be a number that indicates the total number of DMA write messages, across all DRAM ports, for which the OS has allocated space. The FSM writes this into the MAX_BUFFER register by asserting BUFFER_WE, the returns to the reset state. If the opcode is 37, the message is a store acknowledgment from one of the DRAMs (see Figure 5-7). Although there will be another word in the message (the from header), we disregard this word, since we use a unified store counter. Therefore, when the next word arrives, the FSM just increments the store counter by asserting the STORE_COUNTER_INC signal, and then returns to the reset state.

![Device Location Programming Message](image)

**Figure 5-5: Device Location Programming Message**

![Buffer Space Allocation Message](image)

**Figure 5-6: Buffer Space Allocation Message**

If the opcode is 35, the FSM moves into state 5. An opcode of 35 indicates a tile information programming message (see Figure 5-8). This message type provides all of the information necessary to route interrupts to a tile. In state 5, the IC loads the number of the tile that is being programmed into the Tile Receive Select Register.
Figure 5-7: Store ACK from DRAM

(TRSR) by asserting the TRSR_WE signal. This value is used as the selector for a 16-way decoder, with each output bit of the decoder tied to the global enable of the Tile Information Module (TIM) for one of the tiles. For any field to be written into a holding register in a TIM, both the enable signal for that field and the global enable for that TIM must be high. In state 6, the IC loads the next word of the message, which should be the interrupt header for the selected tile, into the INT_HEADER_REG register (see figure 13) in the TIM by asserting the HEADER_WE signal. Then, in states 7 through 9, it writes the DMA header, base address, and base mask by asserting CACHE_WE, BASE_WE, and MASK_WE, respectively. After writing the mask, the FSM moves back into the reset state.

Figure 5-8: Tile Information Programming Message

The final message type that the FSM can process is an interrupt request mes-
sage, indicated by an opcode of 36 (see Figure 5-9). When a header with this opcode arrives, the FSM writes the header into the ISU by asserting the write enable (STORAGE_WE) and write address increment signal (STORAGE_INC). The header will be needed by the Transmit FSM (see section 5.1.6). The FSM then moves to state 10, the tile number write state, where it writes the next word of the message, which should be the number of the tile being targeted, into the ISU. Finally, in state 12, the FSM writes the vector itself into the ISU. Since the vector is the last word of the interrupt request message, the FSM also asserts the INTS_INC signal, which indicates that an additional vector has been written to memory. The FSM then moves back into the reset state.

![Figure 5-9: Interrupt Request Message](image)

### 5.1.3 Interrupt Storage Unit

The Interrupt Storage Unit (ISU) (see Figure 5-10) is where interrupt requests received from the MDN are buffered for translation. The core of the ISU is a 4 Kb dual-ported RAM, generated from the available selectRAM blocks on the Virtex-E FPGA. The RAM has two independent ports, a write-only port and a read-only port. The RADDR and WADDR registers are used as the read and write addresses, respectively. The signals RADDR_INC and WADDR_INC are used to increment the read and write addresses of the memory. The RADDR_INC signal comes from the transmit FSM, while the WADDR_INC signal comes from the receive FSM. The ISU
checks to see if the buffer is full by comparing WADDR to (RADDR - 1). This is actually a conservative estimate, since in reality the buffer is only full when the write address has wrapped around to be equal to the read address. However, asserting that the buffer is full when there is actually a word to spare means that there will be space available for whatever word was last read out of the RAW port. The hardware cannot check for a full buffer by directly comparing WADDR with the signal (RADDR -1), since the sum of the propagation delay of the registers, the propagation delay of the decrement circuit, and the tpd of the comparator circuit will be greater than the necessary minimum clock period. Hence, the hardware keeps track of the previous read address (RADDR_OLD_REG in Figure 5-10). Because the read addresses are monotonically increasing (modulo the size of the buffer), the last read address used must be the current read address minus 1. To assure that the full detection works properly on reset, the register RADDR_OLD_REG is initialized to 0xFFF instead of to 0. Finally, the ISU maintains a counter of the number of interrupts that remain in the buffer for translation. This counter is incremented by the receive FSM whenever it has finished writing a request into the buffer, and decremented by the transmit FSM whenever it has finished translating a request. The NUM_INTS signal is compared to 0 to generate the INTSLEFT signal, which is used to let the transmit FSM know that there are requests that must be transmitted.

5.1.4 Tile Information Module

For each tile in the chip, the interrupt controller maintains a Tile Information Module (see Figure 5-11). The TIM has holding registers for a tile's interrupt header, DMA write header, base address, and base mask, each of which has its own independent write enable signal. In addition, each TIM has a global write enable signal that is generated by using the TRSR as the input to a 16-way decoder, and global read enable that is similarly generated by the Tile Send Select Register (see below). To write any piece of data to the TIM, both global enable and the enable for that item must be high. Although the TIM's holding registers do not contain all of the information necessary to send an interrupt, they are sufficient to generate any additional values needed.
Figure 5-10: Interrupt Storage Unit
For example, the relay uses the same funny bits and destination coordinates as the DMA write header. Therefore, the TIM generates the relay header by substituting the opcode and length fields from the DMA header with the appropriate values (14 and 1, respectively).

In addition to the holding registers, the TIM contains a counter (ADDR_COUNTER in Figure 5-11) that is used to generate the write address. This counter is incremented by 1 whenever the transmit FSM asserts the WADDR_INC signal (see below). Three 0's are appended to right-hand end of the counter, forming a counter that increments by 8. Assuming that the base address is aligned to a cache-line boundary, this ensures that all DMA writes are aligned as well. The counter is ANDed with the base mask, and this value is then added to the base address to form the write address. As shown in figure 14, the process generating the new write address is pipelined, because the FPGA cannot do a 32-bit add in one clock cycle. In addition, a special operation is needed to compute a new write address whenever the tile information registers are reprogrammed. To do this, the MASK_WE signal is registered as a special control signal called WRITE_ADDRESS_INITIAL_PROGRAM. When this signal is high, the write new write address is registered. Note that, with this design, it is assumed that the information registers will not need to be programmed more than once, which is why the address counter is initialized to 0x7f instead of 0. When the write address is computed after the TIM is programmed initially, this value will be incremented to 0x00.

The four signals that are needed for the translation process (the interrupt header, DMA header, relay header, and write address) are multiplexed to a single output bus. The selector for the multiplexor (TDSEL in figure Figure 5-11) is generated by the Transmit FSM. The data outputs from all 16 TIMs are then connected to the Tile Information Bus through tristate drivers. The TIM read enable signal, also generated by the transmit FSM, acts as the enable for the bus drivers. Because the enable signals for each tile are taken from the output bits of a 16-way decoder (see below), there is no possibility for bus contention.
Figure 5-11: Tile Information Module

5.1.5 Controller Information Module

A block diagram of the Controller Information Module is shown in figure 15. The CIM stores information needed to identify and locate the interrupt controller. For identification purposes, 6 words are hard-coded: the device class, vendor ID, device ID, major revision number, minor revision number, and comment. These 6 words are selected using the IDSEL multiplexor, as shown in figure 15. In addition, the CIM has a holding register (LOCATION.REG in figure 15) that contains the controller’s X and Y coordinates, MDN header type, and sub-port ID. This information is programmed into the IC by the OS (see section 5.1.2). The location word is used to generate
a "from header" that is used for acknowledgments and for messages that have the AREQ bit set in the extended header (see section 2). Finally, the TAG.COUNTER register is used to generate a unique tag for store ACK requests. Although at present we cannot see any way in which the tag will be necessary, by convention it is best to use a new tag for each message.

5.1.6 Transmit FSM

The state transition diagram for the transmit FSM is shown in Figure 5-12. The inputs and outputs for the FSM are shown in tables X and Y below:

The FSM assigns a higher priority to ID requests than it gives to interrupt requests, since we assume that ID requests will happen only at startup time. If the PING_REQUEST bit is high, the FSM writes the header MDN header used in the response into the send buffer. It does this by setting SBWDSEL to 10, which selects the CIM, setting IDSEL to 0, and by asserting SBWE. Next, the FSM must write all of the identification data in the CIM into the send buffer. To do this, it simply sets changes the value of IDSEL as appropriate, writing each of the 6 identification words (device class, vendor ID, device ID, major rev., minor rev., and comment) into the send buffer.

If PING_REQUEST is low, the FSM can send DMA writes to the DRAM. The FSM will attempt to do so if INTSLEFT is high and if STORE_AVAIL is also high. If STORE_AVAIL is low, this means that there is no buffer space available in the DRAM for any new DMA writes, so the FSM will not bother to put new messages into the send buffer. The FSM first reads the extended header of the current interrupt request out of memory and writes it into the Extended Header Register by asserting EHR.WE. The FSM also increments the read address of the ISU to ensure that, when it comes time to read the target tile number, the proper value will appear on the output of the memory. In the next stage, the FSM loads the MDN header and extended header that will be used to send the ACK into the appropriate holding registers by asserting AH.WE and AEH.WE, respectively. These values may or may not be used later, depending on the value of the AREQ bit in the extended header.
Inputs are (INTSLEFT, SET_PING, ACK_REQUESTED, SENDBLOCK, DRAM_AVAIL)
Outputs are (SBWDSEL[3:0], TDSEL[1:0], IDSEL[2:0], TSSR_WE, SBWE, WADDR_INC, MEMADDR_INC, INTS_DEC, DRAM_INC, EHR_WE, AH_WE, AEH_WE, ARHR_WE, PING_WE)

Note: The SENDBLOCK and DRAM_AVAIL inputs are not shown. If the SENDBLOCK input is ever high, the FSM will always stall in the current state regardless of the other inputs. If DRAM_AVAIL is low, the FSM will not attempt to send vectors.

Figure 5-12: Transmit FSM
read from memory.

The next step is to load the target tile number into the Tile Send Select Register. Since the read address of the ISU was incremented on the previous clock cycle, we assume that on the current cycle, the target tile number is present on the output of the ISU. The FSM loads this value into the TSSR by asserting the TSSR.WE signal. It also asserts the MEM_ADDR_INC signal so that the vector can be read later. As with the TRSR, the value from the TSSR is fed into a 16-bit decoder, and each output of the decoder is fed into the read enable of one of the TIM's. This allows the subsequent translation steps to read data from the correct tile.

The FSM then moves to state 4, in which it looks up the DMA write header for the target tile. It does this by setting the TDSEL signal to 2, which causes the multiplexor in the Tile Information Module to output the DMA write header. To write this value into the send buffer, the FSM sets SBWDSEL to 2, which causes the send buffer to select the Tile Information Bus as its data source. In the next state, the FSM sets TDSEL to 1 instead of 2, allowing the TIM to output the address in DRAM to which the vector should be written. The next item that must be sent is the mask for the DMA write. Because this value is constant in every message, the 4th input of the SBWDSEL multiplexor is hard-wired to 0xFFFFFFFF. Therefore, the FSM simply asserts SBWE and SBRE and sets the SBWDSEL value to 4. The next word to be written is the valid word. This, again, is a hard-coded value, so the FSM just changes the SBWDSEL value from 4 to 3. The next word to be written, is the vector itself. Since this value is coming from the ISU, the FSM changes the value of SBWDSEL from 4 to 0, and again asserts the RADDR_INC signal. Finally, to complete the DRAM write message, 6 0's are written. This is done by asserting the SBWE and SBRE signals and setting SBWDSEL to 2 for 6 successive stages.

After the cache line has been written, the interrupt header for the tile must be bounced off of the DRAM controller using the MDN relay mechanism. The FSM writes the relay header by setting TDSEL to 3 and setting SBWDSEL to 1. To write the interrupt header, the FSM changes TDSEL to 0. Finally, it is required that a store ACK request be sent with every DMA write. To send the ACK request, the FSM
must write the ACK request MDN header, the ACK request extended header, and the
"from" header, which are accessed by setting SBWDSEL to 6, 7, and 5, respectively.
When the FSM writes the "from" header, it also asserts the DRAM_DEC signal, which decrements the store counter.

At this point, the FSM examines the AREQ bit of the extended header that was
loaded in state X. If this bit is 0, it means that the EIT that sent the interrupt does
not need an acknowledgment, so the FSM returns to the reset state. If the bit is
set, however, the FSM must send an ACK to the EIT. Fortunately, all of necessary
words of the ACK message have already been generated. The "from" header is a
constant, and the MDN header and extended header were generated by the FSM in
state X. Therefore, the FSM simply writes these words into the send buffer, by using
the appropriate values of SBWDSEL, and then returns to the reset state.

5.1.7 Send Buffer

A diagram of the send buffer is shown in Figure 5-13. The send buffer is responsible for
queuing the messages generated by the Transmit FSM. The core of the Send Buffer
is a 4 Kb dual-ported synchronous RAM with a read-only port and a write-only
port. The send buffer takes as input a write enable signal (SBWE) and a data input
(INDATA). It outputs data (OUTDATA) and a data available signal (DAV) to the
I/O unit, and outputs a signal indicating that the buffer has filled (SENDINBLOCK) to the Transmit FSM.

The structure of the send buffer is very similar to that of the Interrupt Storage
Unit. Since the memory is being used simply as a FIFO, two counters, RADDR and
WADDR, are used as the read and write addresses, respectively. When SBWE is
asserted by the Transmit FSM, the value on INDATA is written into the memory and
WADDR is incremented. When SBRE signal is asserted by the IO unit, the value at
the current location in memory is presented on the output port of the RAM, and the
read address is incremented. The DAV signal is high if the read address and write
address are not equal. However, because of timing constraints, this value must be
computed speculatively. The signal AVAIL_SPECULATE is calculated by checking
Figure 5-13: Send Buffer
if, on the next clock cycle, the two addresses will be unequal. This signal is then registered and output as the DAV signal, ensuring that it can be used properly by the I/O unit.

5.2 Design of External Interrupt Transmitter

Figure 16 shows a diagram of the External Interrupt Transmitter. The EIT has four “sticky” IRQ pins, each of which is mapped to a vector and a tile number. Each pin does not necessarily have to map to a different device - it is possible that a single device will use multiple IRQ pins. In addition, the EIT has a “mode” pin, which sets the EIT to slave mode (0) or master mode (1). In addition, for each IRQ pin, the EIT also takes as input a target tile number, a vector, and a header to be used to send interrupt requests to the interrupt controller. The EIT interfaces to the RAW chip through a standard MDN port; its I/O module is identical to the I/O module of the IC, as shown in section 5.1.1.

In order to correctly convert a request on an IRQ pin to a request message for the Interrupt Controller, the EIT requires the following information: the header and extended header for the interrupt request message, the target tile number, and the vector. This information can be taken from the input pins, as described above, or can be programmed into the EIT. The MODE input is used to select between the different sources for this information. If the EIT is in master mode, then this information must be programmed by the OS, and the programmed information will be used when generating messages. If the EIT is in slave mode, the information provided on the input pins will be used. It is assumed that the OS knows, at startup time, whether each EIT is in master mode or slave mode. If the EIT is in master mode, the OS programs it by sending it a programming message. The format of the message is shown in figure 17. The message contains a total of 12 words and consists of the interrupt request header, target tile number, and vector, sequentially, for each of the four IRQ lines. These values are programmed into holding registers (HEADER, TILE, and VECTOR in figure 17) for each IRQ. As shown in the figure, for each of
these parameters, the value from the internal holding register is multiplexed with the value provided by on the input pins, with the mode pin acting as the selector.

It is expected that, in the common case, the EIT will operate in slave mode. The reason for this is that, instead of connecting directly to the RAW chip, the majority of external devices that we will use will interface with the chip through a special bridge designed for that class of device. For example, the first revision of the RAW motherboard contains a PCI bridge that connects all PCI devices to the RAW chip. Since all devices, regardless of type, must use the common interrupt interface described in this thesis, we simply allow the PCI bridge to use one or more EITs as slave devices to send interrupts. The PCI bridge distributes the available IRQ pins on the EIT among the different devices. The bridge is responsible for figuring out which tile number, vector, and interrupt controller (since there may be multiple controllers) each pin is mapped to, and presents this data on the VECTOR_IN, TILE_IN, and HEADER_IN ports of each IRQ (see figure 17).

Each IRQ line is fed into a holding register and then presented to the transmit FSM. A state transition diagram for the FSM is shown in figure 18. The IRQs are prioritized based on their number, with 0 being the highest-priority IRQ. When the FSM is in the reset state, it waits until one of the IRQs is asserted. It then selects the highest-priority IRQ and, using the data provided either by the data registers or by the master device, depending on the mode, constructs an interrupt request message and writes it into the send buffer. The send buffer in the EIT is identical in function to that of the send buffer in the Interrupt Controller, with the exception that the buffer is only 200 bytes in length. When the message has been written into the send buffer, the FSM clears and disables the holding register for that IRQ, and decrements the global buffer counter (BUFFER_COUNTER_REG in figure 17). The receiver FSM is designed to simply listen for EOI messages and programming messages. A state transition diagram for the FSM is shown in figure 19. When an EOI arrives, the receive FSM re-enables the holding register of the IRQ corresponding to that message. When a programming message arrives, the FSM simply programs all of the holding registers for each IRQ.
Finally, like any other device, the EIT must be able to respond to a device identification request from the OS, and must receive its location information from the OS. The mechanism by which this is accomplished is identical to the mechanism used in the Interrupt Controller. For details on this process, see section 5.1.2.
Chapter 6

Implementation and Testing

The Interrupt Controller and External Interrupt Transmitter hardware were both implemented using the Verilog hardware description language. The IC and EIT will both be programmed onto a Xilinx Virtex-II FPGA on the RAW motherboard, although for testing purposes, the target was a Xilinx Viretx-E FPGA. The software on the RAW chip used in the tests of the controller was written in a combination of C and RAW assembly.

Once the hardware was implemented, testing of the system was conducted in three phases - module testing, timing tests, and hardware simulation. The first phase, module testing, involved individually testing each module of the hardware using a test harness written in Verilog. The Verilog language has non-synthesizeable language features that facilitate testing, such as the ability to print data about the circuit to the screen, program a sequence of inputs over time, and examine any signal in the entire hierarchy of nodes. Once a module and its test harness were written, the tests were executed using the VCS simulation tool. By using these test harnesses, we were able to check whether or not the logic of the circuit was correct, at least for the cases tested. However, these tests yielded no information about how fast the circuit ran, and the test cases used were estimates of the conditions that would actually be encountered on the RAW chip. Therefore, additional testing, both of the timing and of the logic, was necessary.

The most important, but also the most difficult, aspect of the testing phase was the
timing test. In order to ensure that the hardware would run at the desired clock speed on the target FPGA, the design was synthesized and analyzed using tools provided by Xilinx. Even though the design is only required to run at 150 MHz, rather than the 300 MHz required on the RAW chip itself, meeting the timing requirements on the FPGA platform proved to be an extremely difficult and time-consuming task for a number of reasons. Firstly, the propagation delay of the standard flip-flop and the delay through a single lookup table (LUT) - the basic computational unit on an FPGA - were both great enough such that, in most cases, it was impossible to put more than three or four logic levels in the critical path. This turned out to be a particular problem in the implementation of complex finite state machines, such as the receive and transmit FSMs in the IC, because just computing which state the FSM is currently in can involve a multi-level computation.

A number of solutions were found to the timing problems mentioned above. The problem of timing in the receive and transmit FSMs in the IC was solved by switching to a unary encoding for the states. Although such an encoding is the most mathematically inefficient encoding possible, it allows for the fastest possible determination of the current state. In addition to using unary encoding, case statements were abandoned in favor of hand-written combinational assignments for each individual state bit. This approach, while tedious, assured the most efficient possible implementation of the FSM.

Another approach, which has been mentioned in section 5, was to use speculation wherever possible. A situation that arises in a number of places in the hardware is that a certain signal’s value is dependent on whether some counter is equal to a certain value (normally 0), but that the logic to check the value of the counter introduces too much delay. This problem was solved by predicting, based on information available in the current cycle, whether or not the counter would be 0 on the next cycle, and then register the value of this prediction. In doing so, we eliminate the delay from checking the value of the counter.

Although pipelining is a common technique to increase clock frequency, we avoided pipelining as much as possible. Pipelining would have considerably increased the
complexity of the circuit, and since using speculation or changing certain data formats was sufficient in most cases, we did not consider the added complexity to be warranted. Nevertheless, in a few cases the only option was to pipeline certain data paths. As shown in Figure X, for example, the data input path to the send buffer is pipelined, because the combined delay of the output multiplexor in the TIM and the SBWDSEL multiplexor is too great. In addition, the construction of the write address in the TIM must also be pipelined, since the 32-bit addition of the address counter and the base address cannot be accomplished in one clock cycle. Fortunately, because there is a guaranteed minimum of 13 clock cycles between when the address is incremented and when it might be read again, we could simply use a 2-cycle adder and not worry about any control problems. It is very likely, however, that in the future, if the IC must be made to run at 300 MHz, the hardware will have to be pipelined.

Finally, a few cases were encountered in which the hardware generated by the synthesizer was simply too slow for our purposes. The most common example of this problem was the unary multiplexor structure generated by synthesizing a case statement. In these cases, the solution was to use custom-generated modules that performed exactly the operation we wanted. For example, rather than using a case statement like the one shown in Figure 20, we instantiated an actual multiplexor in its place. These custom modules were generated using the Xilinx Core Generator tool, which generates Verilog source files and other design files for components such as multiplexors, adders, registers, and RAM. The SBWDSEL multiplexor, the output multiplexor in the TIM, and the RAMs in the ISU and send buffer were all generated using the Core Generator.

By using a combination of speculation, unary encoding, pipelining, and custom modules, we were able to achieve a maximum clock period of 156 MHz on the Virtex-E. The next stage in the testing process was to attach the IC to the RAW chip and test the logic again. By using VCS, we can conduct simulations of the RAW hardware and any external devices that are attached to it. For these tests, we used the same software that was used in the bC tests (See section 3). Rather than attaching a bC model of the Interrupt Controller to port 8, we attached the model generated from
the Verilog source code. However, we were forced to use the "dummy" model of
the EIT on port 0 to generate interrupt request messages, since it is not possible to
provide external signals such as IRQs to devices in the VCS simulation. An additional
caveat is that the tests in VCS could not be done with the core-generated DRAMs.
The reason for this was that VCS does not include any contamination or propagation
delays in its models. These delays, however, are critical to the design of the Interrupt
Controller, because it allows us to change the address and data inputs to a RAM on
a clock edge and still ensure that, because of contamination delay, the old data is
written to the old address. Therefore, for the purpose of the full-system tests in VCS,
the RAMs were replaced with large register files whose behavior modeled that of the
RAMs.

The full-system tests exposed additional problems with the logic that were not
discovered in the module tests. This was to be expected, however, since the full-
system tests were bound to produce conditions that were not predicted when the
module tests were written. Most of the problems encountered were in the I/O unit.
Again, this was to be expected - the module tests had verified that the translation
logic was working properly, so it was likely that most additional problems would be
found when trying to get messages onto and off of the chip itself. It was found, as
a result of this testing, that the original design of the I/O unit had failed to take
into account some of the timing issues that arose from the fact that the RAMs used
synchronous reads. In addition, a problem arose if the last word of the send buffer
was waiting to be sent but the network was jammed; the word would be presented on
the output of the send buffer, but on the next cycle, the send buffer would indicate
that it was empty, and the INTSAVAIL.REG signal in the I/O unit would be cleared,
causing the last word to be lost. The current design of the I/O unit reflects changes
that needed to be made to solve these problems. Finally, to ensure that the changes
to the logic did not violate the timing requirements, every revision of the design of
the hardware was again subject to a timing analysis using the Xilinx synthesis tools.
For various reasons, we have not yet had an opportunity to test the Interrupt Controller in its expected operating environment. The RAW Operating System has not yet been deployed, so it is impossible to evaluate RAW's overall interrupt handling performance. Additionally, all of the simulations have been conducted with the Interrupt Controller running at the same speed as the RAW chip, since at this point the hardware to interface between the IC and the chip has not been developed. Once the operating system and RAW motherboard have been completed, a re-evaluation of the controller will be necessary. It is possible that, as a result of this re-evaluation, we will find it necessary to try to improve the performance of the existing design, or possibly even switch to an alternate design.

The literature available about existing systems indicates that the latency of the interrupt delivery process, which is what we used to evaluate the controller, is a relatively small fraction of the overall delay between when an interrupt occurs and when it is processed. It was partly for this reason that we were able to choose the DMA design of the controller over the on-board version; a difference in latency of a few hundred clock cycles makes little difference if the average number of cycles needed just to start executing the handler is in the thousands. However, the fact that other operating systems on other architectures take a long time to process interrupts does not mean that this is a model we wish to emulate on RAW. On the contrary, since RAW is intended as an “extroverted” architecture, it is likely that considerable
effort will be made to allow the OS to respond to interrupts as fast as possible. As improvements in delay are made in the Operating System, reducing the delay due to the Interrupt Controller will become more and more critical.

One obvious way to reduce the delay would be to try to get the Interrupt Controller to run at 300 MHz. The ease with which this can be accomplished will depend in large part on the platform that will be used in the future. Any attempt to increase the speed will require a significant effort to pipeline or interleave certain operations in the hardware, which will add significant complexity. It may not even be possible to achieve a 300 MHz clock frequency on an FPGA - if that is the case, it may be necessary to build the IC onto a chip.

Another option, mentioned in section 3, would be to replicate the interrupt controllers. From a hardware point of view, scaling the current controller to handle larger fabrics is fairly simple - additional Tile Information Modules will have to be added, but no other hardware changes will be necessary. A more likely solution, however, would be to replicate the controllers. Not only is replication very feasible, especially under the new hierarchical I/O scheme, but also it is a more attractive option from a latency standpoint than having a single controller in one corner of the chip. In fact, the best solution might be to both scale and replicate the controllers. By increasing the number of tiles that a controller can handle and replicating the controllers around the edge of the chip, any device could interrupt any tile, regardless of where it was positioned on the edge, and the vector would still arrive at the DRAM closest to the target.

However, even if the controllers were scaled and replicated in this fashion, or if the controller were to be implemented at 300 MHz, the sheer number of words that need to be sent as part of every interrupt will still be a major bottleneck. Barring some change to the memory controller that allows us to dispense with the extra 0's in the cache line write, or a change in the message formats to reduce the number of words, the number of words of overhead for each vector delivered is over 20. Furthermore, the fact that vectors must be written into DRAM adds another layer of acknowledgments to the protocol.
It may, in the end, turn out that the best solution to this problem will be to revert back to the older, on-board storage design for the interrupt controller. Although this design was originally discarded because of problems with the storage requirement, these problems could be eliminated if, as proposed above for the DMA design, the controllers were replicated around the edge of the chip. If each controller were tasked to handle some constant number of tiles on the chip, then the storage requirement would scale linearly, rather than quadratically, with the number of tiles in the chip. Furthermore, because the EIT can be programmed with the coordinates of the target interrupt controller, it would still be possible for any device to interrupt any tile. Of course, the design mentioned in section 2 will have to be modified to deal with the problem of store acknowledgments. The most obvious solution would be to send each vector as a separate message and then send the store acknowledgment along with the vector. This would, of course, increase the amount of overhead per vector sent, although the overhead would still be less than what it would be in the DMA version.

7.1 Conclusion

With neither an operating system, nor a motherboard, nor any external devices available for testing, it is impossible to say with any certainty how the current design of the Interrupt Controller will perform under true operating conditions. However, although we can make no definite predictions about performance, we can say with absolute certainty that we have achieved correctness with this design of the interrupt controller. We can be certain that all communication involved in the interrupt delivery process is solicited. We can be certain that all outstanding interrupts in the system can be stored without loss. And we can be certain that, because of the stickiness of interrupts, we will never cause traffic jams in the network by sending more messages than the system can handle. Thus, assuming that the OS obeys the necessary software conventions, this controller never cause deadlock, will never interfere with cache transactions, and will never lose any interrupts. Furthermore, as a result of the tests conducted on the hardware, we are reasonably certain that the current
implementation of the chosen interrupt control scheme will work correctly. Thus, as
other components of the RAW system are activated, we know that we at least have
an interrupt control system that works. As time passes and more research is done
into actual applications of the RAW chip, there is a very high chance that the design
of the interrupt controller will need to be revisited. Hopefully, the ideas for future
research mentioned in section 7 will prove useful in any subsequent examination of
the interrupt control problem.
Bibliography


