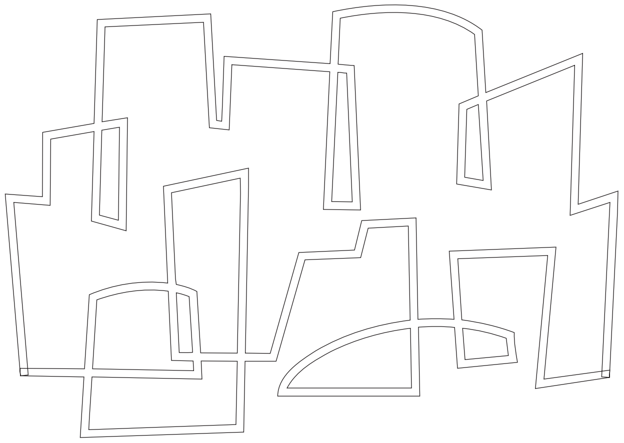
A High Performance Multiprocessor DSP System

Todd Heirs

Masters Thesis

2001, May

Computation Structures Group
Memo 439



A High Performance Multiprocessor DSP System

by

Todd C. Hiers

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 23, 2001

Author ____
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ____
Larry Rudolph
Thesis Supervisor

Certified by ____
Nat Seshan
VI-A Company Thesis Supervisor

Accepted by ____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A High Performance Multiprocessor DSP System

by

Todd C. Hiers

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 23, 2001

ABSTRACT

As DSPs have become more sophisticated so have the kinds of systems they are used in. In this thesis, a multiprocessor DSP system is built that attempts to fulfill the kinds of demands for a DSP-based special purpose system while also providing the flexibility of a general purpose system. This system is designed for low cost and high performance, and meets those goals, though imperfect in its implementation.

Thesis Supervisor: Larry Rudolph

Title: Professor, Electrical Engineering

1 Introduction

1.1 Overview

There are many high performance systems in existence that can be broadly classified by their abilities and applications. Scientific computing has vector and floating point support, enterprise computing has large memory and operating system support, and special purpose computing has customizability. Digital Signal Processors (DSPs) have traditionally been used to build systems that fit this third category, as they lacked floating point, large memory and O/S support. Recently however, more sophisticated DSPs have blurred the lines defining the se categories. Modern DSPs may have floating-point support and may have rudimentary O/S support as well.

Thus, the classification of high performance systems has shifted slightly. A better distinction may be made between systems that are latency tolerant and those that are not. The former can be built with clusters of general purpose computers connected by standard networks. The latter kind of system is where DSPs are well suited to the task. This thesis is the design and implementation of a high performance, DSP-based multiprocessor system. The have been many of kinds of these systems implemented with DSPs, ranging from on-die multiprocessor setups [Texas Instruments] to chips that are suitable for flexible memory systems {Analog Devices], and general purpose processors arranged for signal-processing type tasks [Halstead, Passmore]. This system is suited for the kinds of latency-intolerant tasks that engendered DSPs and special purpose computing, but still attempts to retain the flexibility of a general purpose computer.

1.2 Design Goals

This system endeavours to be many things: Fast, cheap, reliable, inexpensive. Many of these are conflicting goals, however, and a priority must be established. Inspiration of this project comes from the like of Bobnet [Csandy] and Memory Channel [Gillett] that built high-performance systems using commodity parts, and were largely successful.

This system is designed with performance and cost as primary objectives. By using off-the-shelf components, performance is not as good as it could be with a completely custom design, but cost is significantly reduced. Similarly, using some better performing devices when less capable ones would still provide a functional system does not serve to minimize cost, but does benefit system performance. In general, cost has priority over performance. Thus, the main goal could be stated as designing an inexpensive system that performs as well as possible without greatly increasing cost.

While the system seeks to balance cost and performance, it also desires flexibility. The design is not based on any particular application, and the hardware should not make assumptions about the applications that will use it. The system should not use application as a basis for design decisions, rather it should support as many as possible as well as possible. Reliability is also a goal, but has the lowest priority. This system is simply not intended to be fail-safe, but it should be as robust as possible given the other priorities.

1.3 Document Organization

Chapter 1 is this overview. It presents the basic premise of the system design, the project goals, and the document organization.

Chapter 2 is the hardware description. It explains the hardware platform in its entirety and shows the basics of operation

Chapter 3 is the message passing system description. It describes the concept and mechanics of a message-passing multiprocessor implementation.

Chapter 4 is the processor in memory system. It describes another use of the hardware, in which the system acts as memory in host PC with special processing ability.

Chapter 5 is the evaluation and performance of the system. It chronicles the major and minor problems with the system, and gives an idea of the performance of the message passing implementation.

2 Hardware Description

2.1 Overview

From the vague idea of building an inexpensive, yet powerful and flexible system using DSPs and off-the-shelf components, an architecture is born. This is an MIMD parallel computer according to Flynn's taxonomy [Flynn]. Processors not only have their own data and program streams, they operate completely autonomously, as there is no master control processor coordinating them.

2.1.1 Clustering

Culler and Singh present a generic scalable multiprocessor organization [Culler et al.] with a hierarchy whereby processors can be grouped and networked with nearby processors, and connected to other groups, or clusters, of processors by some potentially different means. This is meant to address scalability by allowing clusters to contain an easily handled number of processors, while still allowing the overall system to grow quite large. System availability can also be addressed by multiple processors [Kronenbeg], though this is not the focus of this project. Depending on the context of the system, the cluster interconnect may be essentially the same as the local interconnect or it may be vastly different.

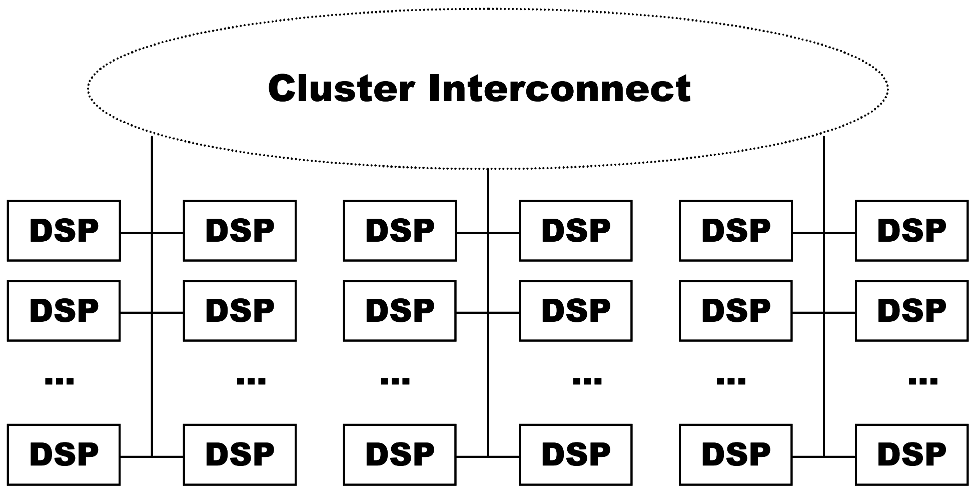


Figure 1. Local and Clustering Interconnect

In this case, clusters can be networked together to provide a scalability solution that extends beyond what the DSPs could do unclustered. This is not unconditionally true, however. Care must be taken when designing the cluster interconnect so that it does not become a system bottleneck via its complexity or performance characteristics.

2.2 System Implementation

With the abstract idea of connected groups of networked DSPs, we can set about creating an implementation. We must consider the hardware components in rough order of their importance to the system, as the devices chosen will limit the options available for designing the other parts of the system. Hoenig describes a similar multiprocessor system design [Hoenig] that shares some of the design tradeoffs of this system.

2.2.1 The Processor

It is appropriate to chose the processor first, as the choice of hardware on which the system will be implemented will influence many of the design decisions. The processor core will govern the interconnect details, as well as what kind of processor-to-processor relationships can exist. The TI TMS320C6000 family of DSPs offers a variety of DSPs that correlate well with the overall system goals: Fast, cheap, and straightforward.

The TMS320C6000 (C6x) family of DSPs comes in a variety of configurations, all of which share the same execution core and fixed-point instruction set. Some are fixed-point only cores, while others are floating-point cores that have the same fixed-point instructions, but also have a set of floating point instructions. Aside from the fixed-point versus floating-point core, they differ mainly in the peripherals available on-die, though many have identical or similar peripherals.

Common to all 6x DSPs is the explicitly parallel, very long instruction word (VLIW) core. This core is comprised of two identical banks of functional units. Each of these banks has an independent register file, with enough ports to fully service the functional units, plus provide a crossover for at most one of the functional units in the other bank. Each of these functional units is capable of executing a different subset of the C6x instruction set, and it is up to the compiler to assign an instructions to functional units. It is also up to the compiler to decide which instructions may be executed in parallel.

The execution pipeline is designed so that difficult operations (e.g. load, store, multiply, branch) have a fixed number of delay slots before their effects become visible, rather than stall the pipeline (although memory stalls may still occur). This allows a crafty compiler to pack instructions into the delay slots of other instructions, increasing the work density done by the CPU. Therefore, it is technically possible to use all functional units every clock cycle and execute eight instructions at a time. Since the C6x devices have maximum clock speeds between 200 MHz and 300 MHz, this leads to a theoretical core performance of 1600 MIPS to 2400 MIPS. In practice, however, it is impossible to keep all functional units fully engaged all the time, leading to lower actual achieved MIPS. The core is still quite capable though; in discrete-time signal processing applications where multiply-accumulates performance is critical, the C6x shines. It is capable of doing up to six additions/subtractions and initiating up to two multiplies every cycle.

The C6x is available with a variety of peripherals on-die. Some of these peripherals can be used for high-bandwidth data transfer to/from the DSP. Most have the proprietary host port interface (HPI) though recently devices have been made with other standard ports, such as PCI or ATM. These ports provide a way to directly interact with the DSP and to do massive data transfers.

From the C6x family, individual versions can be evaluated for their suitability in this multiprocessor system. There can be no differentiation by processor core, as they are identical, save for the fixed versus floating point issue. With no specific design criteria that suggests the need for floating-point capability, and with an eye towards the more numerous and peripheral-laden fixed-point cores, no determination of a particular device or even subset of devices can be made by processor core. All else being equal, a floating point core is more versatile, whereas its fixed-point cousin is faster and cheaper.

The peripherals available on-chip are the big differentiators in this design. Clearly, the most important peripheral in this design is the one through which inter-processor communication will be achieved. The different flavors of the C6x had only a few high-bandwidth ports available at design time: host port interface (HPI), and peripheral component interconnect (PCI) . Here, PCI is a winner for both performance and versatility reasons. The HPI has a maximum throughput of 50 MB/s or slower for certain types of transfers. PCI, has a maximum throughput of 132 MB/s and in practice, can sustain 50 MB/s, easily outperforming the HPI. PCI also has the flexibility advantage. Having a PCI bus as a communication medium makes the system able to interface easily to PCs. It is clear that PCI is the desirable feature to have on the device incorporated into this design. PCI also has the advantage of being recognized as a good commodity interconnect for multiprocessor systems [Walker]. This reduces the set of considered processors to just one: the TMS320C6205 (C6205)

As the only C6x with PCI, the C6205 stands out as particularly well suited to this system. Using the C6205 as the basis of the design, the rest of the system may be designed around the DSP in such a way to make high-bandwidth message-passing cheap and practical.

2.2.2 The Link

The choice of PCI as the local network medium presents some different possibilities for the inter-cluster links. PCI is directly scalable through the use of PCI-to-PCI bridges. This scaling, while straightforward and transparent, lacks the performance desired for large systems with many clusters. The PCI bridge serves to link two PCI busses; busses are linked in a tree fashion, where there is one root bus from which all others branch out. The PCI busses share the same address space, making data transfer between the two simple, as all addresses are globally unique. The specification, however, calls for a switched-connection, as opposed to switched-packet style of interconnect. This has the undesirable effect of tying up multiple PCI bus segments to do one trans-segment connection. A PCI transaction that must go from PCI bus A to PCI bus B will utilize not only busses A and B, but all busses in between as well.

This performance problem can be alleviated somewhat by doing buffering at the bridge level. Data bound for other busses can be buffered at the PCI bridge, allowing some asynchrony in cross-bus transfers. This makes the bus-to-bus transfers more resemble switched-packet style of interconnect. At best, however, the simultaneity of the transfer is relaxed, but the data must still cross each segment between source and destination. This cross-bus data still competes with local data for PCI bus bandwidth. Thus, the aggregate bandwidth available is limited by the bandwidth of a single segment.

To get around this aggregate bandwidth limit, a separate mechanism can be employed for the inter-cluster link. Systems such as Memory Channel for PCI [Gillett] demonstrate that inexpensive solutions can dramatically outperform just a PCI bus tree. We can design an inter-cluster link with higher aggregate bandwidth than PCI alone, without greatly increasing cost. Several alternatives exist; the best choice would be one that provides the transparency of the PCI bridge, while still allowing for higher throughput. Unfortunately, at the time of system design, no such devices were available off-the-shelf. The available solutions, all involved increasing the overall system complexity, as nothing could natively connect to PCI. It would be necessary to “translate” PCI cycles to some other format for link transmission. Thus, practical solutions can be evaluated on two main criteria: overall performance, and interface complexity. With current FPGA technology, there is a large initial cost and an extremely low marginal cost for custom logic.. Therefore, debugging concerns aside, a more complex interface logic that can still fit entirely into an FPGA isn’t really any more costly than a simpler logic that also would go into the same FPGA.

For this reason, we can give priority to the performance criteria when choosing this link, as long as the resulting necessary interface logic will still fit into an FPGA. Rather than explicitly minimizing complexity, we can seek to find a solution that has an acceptable level of complexity. Thus, we look to high-performance current technologies for link hardware. Gigabit serial transceivers provide some of the best throughput available at an IC level. They are cheap, available, and reliable. While there is a significant logic gap to fill between PCI interface and transceiver interface, it is ultimately a problem of just getting a big enough FPGA.

In this project, high-speed serial transceivers are used to provide fast inter-cluster links. Current high-end transceivers can achieve around 2.5 gigabits per second in full duplex. These links are really 2 half-duplex links, as the receive and transmit side can operate completely independently. Each transceiver chip then provides one input port and one output port. The question of topology follows - we need to know how many chips each cluster will have and how the link will be arranged

The serial transceiver chips chosen for this design provide an input and an output for the link hardware, however multiple transceivers could be attached to each cluster, to provide more aggregate throughput and more connection topology choices. With one chip per cluster, the problem is trivial; the only way to connect multiple clusters that have one input and one output is a ring. This resembles a token-ring network, where data put into the ring is passed along by all nodes until it reaches its destination. Multiple chips per cluster provides more choices in topology.

The topologies to choose from are bounded by the number of transceivers available to each cluster. For better interconnect and higher aggregate bandwidth, more transceivers is better, but more transceivers also increases routing complexity and consumes more power, board space, etc. In a special-purpose design, the algorithm the hardware was being designed to support may favor a certain communication topology, and the design may be build with that topology in mind. For this project, no particular algorithm is central to the design, so there is no obvious choice of topology; the system should try for a topology that is a good compromise between resources consumed and bandwidth provided.

With two transceivers, two distinct topological possibilities emerge: a full-duplex ring or half-duplex grid. From a control point of view, the full-duplex ring is preferable, as it allows simple flow control feedback, and the hypercube does not. Analyzing the performance characteristics of the ring topology, we find it much more capable than PCI. The ring is really comprised of fully independent point-to-point connections with 2.5 Gbps bandwidth each. Assuming uniform source and destination distribution, data is expected to travel halfway around the ring. Node independence will allow for four such links to run simultaneously, leading to an aggregate throughput of 10 Gb/s (1.25 GB/s), a order of magnitude more than the PCI bus alone could provide.

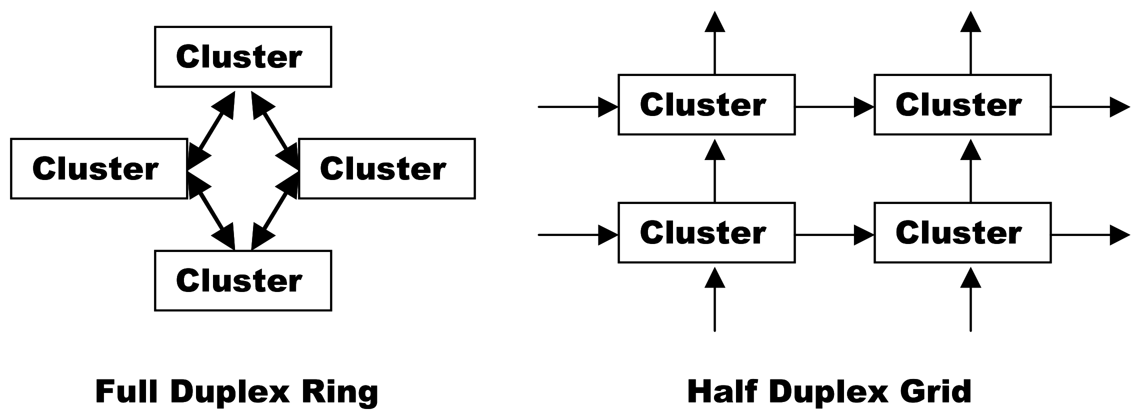


Figure 2. Possible 2-Chip Topologies

Unfortunately, this aggregate system bandwidth does not increase with the number of nodes on the link. It is not hard to imagine a system comprised of enough clusters where this intercluster link would be a performance bottleneck. However, without a specific performance demand for the link, it makes little sense to consume additional board space for additional transceivers or to use the hypercube topology, with its lack of direct feedback. For general purposes, a two-chip-per-cluster solution with full-duplex ring topology will suffice.

2.3 Physical Realization

The basic idea for this multiprocessor system has taken shape; clusters of DSPs, each with its own local memory, are linked internally via PCI and externally by a pair of high-speed gigabit serial transceivers. With this conceptual design in mind, a physical implementation can be designed. Here again, there are choices that involve design tradeoffs. Sine the networking medium is PCI, this allows the possibility of building the system to PCI expansion card specs. This would noticeable increase the flexibility and versatility of the system, as it could be unused inside of a PC, independently or in conjunction with the PC’s general purpose processor. It would also greatly simplify the required system support logic, as much of the mundane housekeeping could be offloaded to the PC.

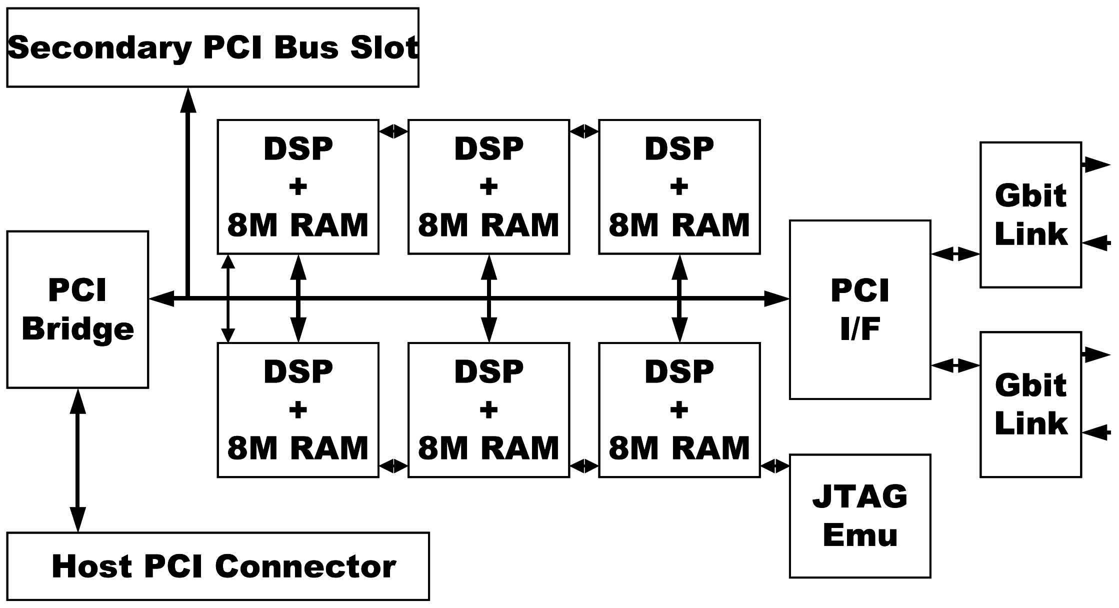


Figure 3. Basic System Block Diagram

Implementing the system on a PCI expansion card is not without its drawbacks, however. The PCI expansion card specification introduces some strict size and electrical limitations that would not be design considerations in a freestanding PCB implementation. The size of the card is cannot exceed 312 mm by 106.68 mm. Components on the front of the card may not exceed 14.48 mm in height, and 2.67 mm on the back. The card itself may be no thicker than 1.77 mm. The specification also states that only one physical device may interface to the card’s PCI connection; having multiple PCI devices, such as the multiple DSPs, requires the use of a PCI to PCI bridge as an intermediary. Power is an issue, as well. The specification allows the PCI card to draw a maximum of 25 watts from the bus. The supply voltages are +3.3, +5, +12, and -12 volts, though not all voltages are available always. Any other voltages needed must be converted on-card. These restrictions would force the addition of a PCI-to-PCI bridge, and power converter modules to the design. These necessary support chips also consume limited board space that otherwise could be used for more DSPs. Six DSPs would fit on the board, making 6 the cluster size for the system.

The advantages of building the system as an expansion card are quite compelling, and enough to outweigh the disadvantages. The PC’s BIOS can configure the system's PCI space, freeing the system from needing to do that itself. The host PC can also be used to boot, monitor, and debug the system. This implementation also leads to heterogeneous multiprocessor possibilities, where the PC's general purpose processor interacts with the DSP multiprocessor system to do computation that is impossible or inefficient with just one or the other. All of this is made much easier by having the multiprocessor system attached to a PC, which can be easily done with a PCI expansion card.

2.3.1 Description

We have now specified all of the major components for the board. We have six DSPs and a link unit that consists of two transceivers plus glue logic on a PCI bus that sits behind a PCI bridge. The support logic and uninteresting, but necessary components can the be added in to make a complete and functional system. A description of the entire system follows. Schematics and VHDL code to realize all of the following can be obtained from http://www.csg.lcs.mit.edu/publications/

A DSP unit is defined and replicated for the six DSP and Memory units. These units are identical schematically, but differ slightly in their physical placement and routing on the board for space efficiency reasons. Each unit contains one C6205 DSP, an 8MB by 32-bit SDRAM, a local clock generator, and an EEPROM for holding DSP PCI boot time configurations.

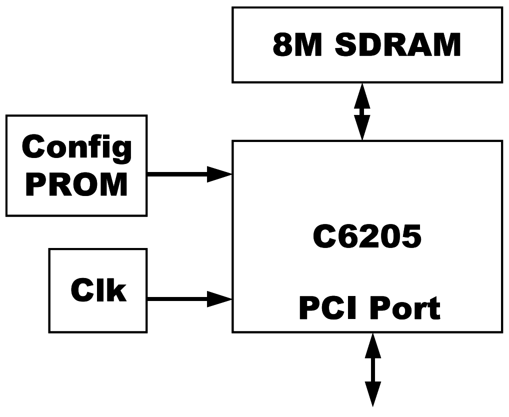


Figure 4. Block Diagram of DSP Unit

The DSP is simply the C6205 described above. The device is configured to boot from the PCI port, which means that after the PCI port is configured by the host, the peripherals of the chip are brought on-line, and the PCI port is given access to the DSP’s memory space while the core is still held in reset. Once the host has configured the device as necessary, which must include loading the boot program, the host sets an interrupt flag in one of the memory mapped registers to awake the core.

The DSP’s EMIF is connected directly to the SDRAM, with pull-up and pull-down resistors on the data lines to indicate the device’s boot mode. Since the EMIF is designed to interface directly to SDRAM or SBSRAM, no external logic is required. The EMIF requires 32-bit wide memories that may be physically comprised of multiple chips (i.e. Two 16-bit RAMs in parallel is a common design) Here, a single 32-bit wide memory is used to save space; otherwise additional room on the board would be required to place more memory chips, further reducing the cluster size. The EMIF can address up to 16MB in a single external memory space (there are four such external memory spaces available). Ideally, a 16MB device would have been used, fully utilizing the EMIF’s capability, but no such chip was available in a 32-bit width at design time. The SDRAM must be PC100 compliant, as the EMIF works at half the clock frequency of the DSP core, which has a maximum speed of 200 MHz. The EMIF is fixed as a CAS-3 latency design, which all PC100 SDRAM is capable of supporting. Thus, the Micron MT48LC2M32B2 SDRAM, an 8MB by 32-bit PC100 compliant SDRAM was chosen.

In addition to individual RAM units, each DSP receives its own clock, as they need not be synchronized with each other, save for the PCI interface, which provides its own clock. MicroClock makes an IC designed specifically for clocking C6x devices. The MK1711 is a selectable phase-locked-loop (PLL) generator that connects to a 20 MHz crystal, and outputs a clock frequency from 112 MHz to 200 MHz, based on some status pins. For this design, the device is hard wired to produce 200 MHz. The C6205 also has an internal PLL capability for slower clock inputs. Since the clock coming out of the MK1711 is at full speed, this internal PLL is hardwired to bypass.

The last piece of significant hardware in the DSP unit is the PCI port's boot EEPROM. Since in PCI boot mode, the DSP core does not have control of the device until after the PCI space has already been configured, it does not have an opportunity to set some configuration parameters that get read by the host at PCI configuration time. This includes the device ID, vendor ID, minimum grant request, and maximum latency request. These parameters all come with default values that can only be meaningfully changed by the PCI boot EEPROM. The first two merely serve as identifiers, and the default values can be used without problem. The second two, however, characterize the DSP's PCI bus usage and performance desires so that the PCI arbiter may divvy up the bus intelligently. The default values here impose the fewest performance demands on the PCI bus, but hamper the devices ability to efficiently burst large data transfers over the PCI bus; i.e. Minimum grant is set to 0 by default, so the PCI port can be stopped after every word of data transfer, effectively preventing bursting of data if there is any contention for the bus. For this reason, a PCI boot EEPROM is included in the DSP unit.

Six of these units are situated on the board to from the DSP cluster. In addition to the PCI link, the DSPs are tied together in a JTAG / Emulation scan chain. The devices are jumpered so that they can be dynamically included or bypassed in the scan chain. This scan chain exists for debugging purposes. It is an additional way to access the DSPs from the PC host, and this is how the development environment for the DSPs interacts with them.

This implementation requires a PCI-to-PCI bridge to satisfy the PCI load specification for the PCI expansion card. This is a commonplace chip made by many vendors with any one of them being no better or worse than the rest. Texas Instruments PCI2050 suits the job well. It is a buffering PCI bridge, as well as an arbiter and clock generator. Since the devices on the PCI bus require an individual clock and PCI request/grant lines, this chip provides all of the PCI support needed for the secondary bus. The primary side of the chip interfaces to the card's edge connector for connection to the host PC's PCI bus. The host PC supplies a PCI clock that the bridge buffers and makes available at ten output pins. The bridge also provides an arbiter for ten devices and IDSEL / AD multiplexing for sixteen. This is plenty of available connections for this design, and is effectively as many as the trace length and electrical load requirements of a PCI bus can support.

In addition to the link PCI interface and the DSPs, an expansion slot also resides on the secondary bus. While this slot could be used to stack cards in a multiple cluster system, the primary reason for its inclusion was debugging. The slot provides an access point on the secondary PCI bus to attach a PCI exerciser/analyzer to be able to observe and generate traffic on the secondary bus.

The link hardware is an independent unit that sits on the PCI bus. Since the hardware is not capable of routing data bound for other clusters transparently like a PCI bridge, the link occupies PCI address space and DSPs wishing to route data through the link must address it explicitly. To this end, the link hardware consists of two TLK2500 serial transceivers, a PLX 9054 PCI master interface chip, and a Xilinx Virtex 300 FPGA to control data flow between the two. Also, two ICS 525-02 clock generators provide the 40 MHz local bus clock and the 125 MHz transceiver clock.

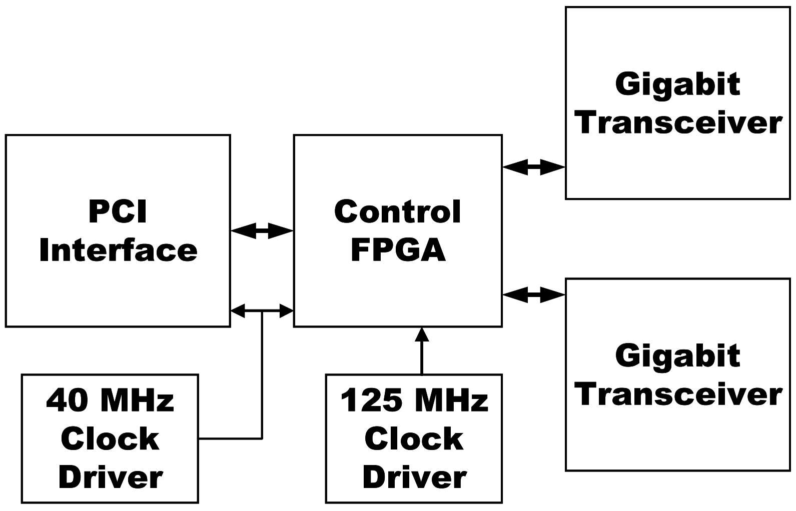


Figure 5. Block Diagram of Link Hardware

The TLK2500s simply provide a full-duplex serializer/deserializer pair that used the gigabit ethernet protocol. They accept input 16 bits at a time along with a clock in the 80 MHz to 125 MHz range. This clock is multiplied up by 20 with the internal PLL to serve as the serial clock. Data is put through 8b/10b conversion for error checking and correction, and the sent out over a differential serial output. The same serial clock is used by the receive half as a baseline clock from which the receive serial clock is derived, based on the serial input signal, This clock is divided down by twenty to produce a receive output clock that the parallel data is clocked out of the chip by, 16 bits at a time. In addition to the 2^16 possible input data sequences, the chip is able to send and receive "idle" (for when no data is being sent/received) and "comma" (for possible control usage). It can also detect received errors. The four possible types of data transmitted or received (data, idle, comma, error) are set or indicated by status pins on the device.

Similarly, the PLX 9054 merely takes PCI transactions and translates them to local bus transactions for devices behind it. The local bus protocol employed is selectable from multiplexed address/data (like PCI), separate address/data, and PowerPC native. In all modes, the bus is selectable width - 8, 16, or 32 bits wide. The PLX device is capable of buffering and address translation, and receiving as well as transmitting data over the PCI bus. The main advantage of using this chip is the relaxed specification of the local bus when compared to the PCI bus. Using the 9054 alleviates many of the electrical and timing constraints that PCI imposes. This is especially true in this case, where the 9054 and FPGA are the only devices on the local bus, whereas there are many on the PCI bus.

The FPGA, then, must do the bulk of the work in this PCI-to-serailizer conversion. It must recognize incoming PCI messages and format and address them properly so they reach their destination in the ring. Since each serial-link is point-to-point, it must also deal with incoming data from both links, and route them correctly (either as pass-through or local). For incoming link data that is bound for the local cluster, it must format and address the data correctly for the PCI bus. Complicating this are the many clock domains. The PCI-FPGA link is clocked independently by a clock that can be from 0 to 40 MHz. Each receive side of the transceivers generates its own 80 MHz to 125 MHz clock for use with the receive side, and each transmit clock requires an input clock in the 80 MHz to 125 MHz range. Since both transmit sides may be driven by the same clock, this leaves four clock domains to cope with: One for the FPGA-PLX link, two for FPGA-Transceiver receives, and one for FPGA-Transceiver transmit.

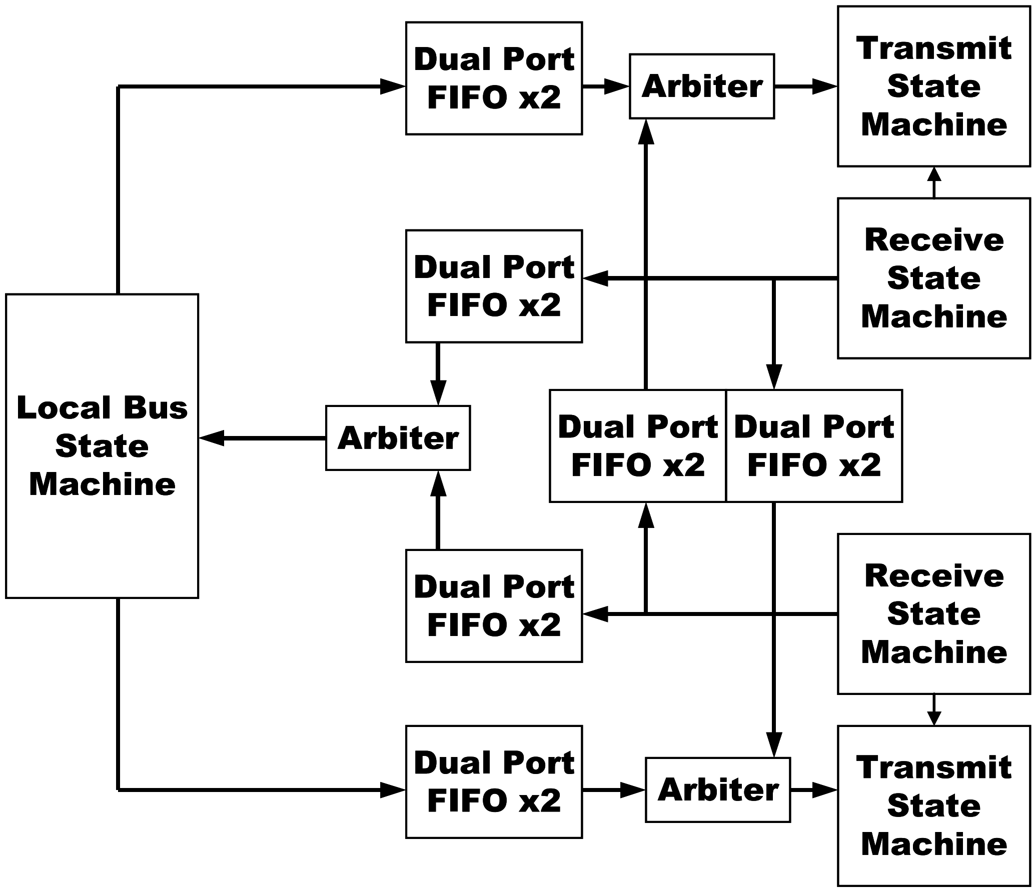


Figure 6. Block Diagram of FPGA Internals

The FPGA is capable of supporting four clock domains internally, however the issue of transferring data internally from one clock domain to another must be handled delicately. This issue is solved by the use of two-clock FIFO buffers inside the FPGA. These FIFOs serve to not only buffer data being processed by the FPGA, but to convert between clock domains, as data may be placed into the FIFO on one clock, and removed from it on another. The FPGA contains internal dual-ported RAM blocks well suited for this purpose. The fist port is clocked by the input clock, and is fed an address by address generation logic also clocked by the input clock. Similarly, the second port is clocked by the output clock, and its address is generated by logic also clocked by the output clock. Only the full and empty signals depend on data from both clock domains (namely the input and output addresses), but these may be purely combinational logic; this means that at worst the correct values for these signals will be seen later than they actually occur, due to propagation delay. This is not a problem in this design, as the full signal is used to allow data to be placed in to the FIFO and empty is used to take data out. Full and empty, however, are set by incoming and outgoing data, respectively. Therefore the system cannot erroneously see an non-full FIFO that is really full, or see a non-empty FIFO that is really empty. It may, however, believe the FIFO to be full or empty, when, in fact, it is not.

Now that the issue of data transfer between clock domains has been resolved, the general control system to control the flow of data through and on/off the serial ring must be discussed. Ultimately, 32-bit PCI bus transactions must be converted so they can be sent over a 16-bit wide channel. It is desirable that the link protocol closely resemble the PCI protocol, so that minimal conversion of data between the two must be done. This would also allow for a relatively low-latency transmission and not artificially limit the PCI transaction size, as buffering would be used for performance optimization, and not as a message store. However, PCI transactions do not map smoothly to the transceiver interface.

Converting from 32 bits to 16 bits and back is simple enough; it just involves choosing an endianness and remaining consistent. Indeed, the 32-to-16 and 16-to-32 modules use a slow clock on the 32 bit size and a fast clock (x2) on the 16 bit side. The 32-to-16 takes in a 32 bit data value and converts it into two 16 bit values using big endian ordering (high two bytes first). The 16-to-32 does the exact opposite, recreating the original 32-bit value. Thus the system is able to do the data width conversion.

PCI has additional control signals, that have nothing to map to on the transceiver interface. Namely, the protocol needs to be able to deal with address data words (handled by FRAME# on the PCI bus) and transmit/receive ready (handled by TRDY# and IRDY# on the PCI bus). The serial transceivers do not have any additional bandwidth to transmit these control signals, however, it does have the "comma" capability, that can be combined with flow control signals to achieve both address notification and flow control. Basically, a comma can be sent as a general interrupt to the receiving FPGA, the data immediately following the comma can contain the control signal and any other necessary information.

This is exactly how control information is transmitted. First, a comma is sent to alert the receiver that what follows is a control signal. The next word is a constant that defines the control signal being sent. The available signals are Ready-To-Send (RTS, which includes address), Clear-To-Send (CTS), Xon (XON), and Xoff (XOFF). In the case of RTS, the next two data words are the big endian representation of the address being sent. Thus, like PCI, the address may be set once, then followed by a potentially unlimited burst of contiguous data, with provisions for backoff if the receiver cannot handle additional data. Transmitter done or not ready is indicated by sending IDLE over the serial link; this must be followed by a new RTS if the transmitter wishes to resume transferring data. This RTS/CTS handshaking and XON/XOFF flow control feedback require that a pair of transceivers be hooked up in full-duplex, though this is not required by the transceiver silicon.

Since the address is only transmitted occasionally, the FIFO must be revisited to be able to associate an address with the data stored in it. A separate address register is added so that it may be set during an RTS by the receiving FPGA, and read from by the transmitting FPGA to get the address for an RTS. This has the undesirable affect of tying a FIFO to a specific burst. Since a new RTS requires setting the address register, only an empty FIFO may be used from a new burst. If a non-empty FIFO were used for a new burst, the address register would be overwritten, corrupting the association with the data previously in the FIFO. Since it is not desirable to have to wait to drain the FIFO before a new burst can commence, and alternating, dual-FIFO scheme is used. In this scheme, a receiver fills two FIFOs, alternating with each burst. This way, a new incoming bust can use the FIFO which was not just previously used, which should be empty. Assuming the data bursts are of roughly equal size, this scheme eliminates any delay associated with waiting for a FIFO to drain, as one would be able to be completely drained while the other is filling, providing somewhere for new data to go at all times.

Data coming in from transceivers must be routed to the appropriate place. Messages bound for the local cluster must be routed to the local PCI bus, all other traffic is sent to the other transceiver for transmission to the next spot on the ring. This requires the logic to be aware of the address range of the local cluster. The address of an incoming message is provided by the RTS control signal, and can be compared against the known local address range to determine how to route the following data. The switch point is actually before the buffers; each receive side feeds two pairs of FIFOs. One pair is for data bound for the local PCI bus, the other is for data that is to be passed through to the other transceiver. Routing like so allows the interconnect between the three units to be slightly less complicated, as only one segment reads from and only one segment writes to any given FIFO.

In addition to the two transceiver interfaces, the FPGA contains a local bus side for data coming from or going to the PCI bus. This module takes data from two pairs of FIFOs (one pair from each transceiver module) and routes it to the local bus for transmission on the PCI bus. It also takes incoming data from the PCI bus, via the local bus, and routes it to the appropriate transceiver. Since the clusters are linked in a bi-directional ring, and each node either routes the incoming data to its local bus or forwards it on to the next node, routing data form the PCI bus in either direction would be correct, because in either case, the data will reach the intended target. Routing, then, becomes a question of optimization, as the fewer hops that are required to send a message, the greater the available aggregate bandwidth of the serial transceiver ring. This performance optimization is an interesting problem unto itself, and is beyond the scope of this project. The FPGA routes all traffic statically; range resisters are provided to describe the address ranges that should be routed in each direction (one contiguous range per direction). By convention, the local range is defined to be the space between the two address ranges. This local range information is provided to the transceiver modules for their routing logic. This routing logic is smart enough to detect a “rollover” in address space, allowing the extreme high addresses and extreme low addresses to be mapped in one direction. This is necessary because the PCI addressing will be linear, so at some point on the rig the extreme high addresses are one hop away from the extreme low addresses. Using this scheme, the FPGA must be configured by an intelligent host to set up the routing. The host can then assign the address ranges such that messages get routed in the direction that requires the fewest hops.

One other complication arises in interfacing to the PCI bus. Namely, DSPs write to the FPGA through the PLX device, but due to the non-transparency of this link solution, they must write explicitly to the PLX device's PCI space. However, this is not the address the data is ultimately bound for. The DSPs must specify the true destination PCI address of the data. Data bursts may be interrupted on the PCI bus and resumed later, but there is no practical way to re-send the true destination address every time a transfer is resumed. A solution is to provide a mapping register that specifies where the true destination is in relation to the PLX’s address space, so the true destination address may be deduced each time a transfer restarts. This solution works fine when only one DSP is writing to the link at a time, there is no conflict for the map register. A problem arises, however, when multiple DSPs are using the link hardware simultaneously. A resumed transfer might not map properly to the correct destination address, because another DSP could have overwritten the map register. The solution to this is to provide a map register and address space for each DSP that writes to the link hardware. This way, the FPGA can know from which DSP incoming data is from, and know the proper map register with which to do the address translation.

Since each destination DSP has a 4 MB PCI address space to write to, this is the size of the address space allotted to each DSP in the PLX chip’s range. The host setting up the system informs each DSP of what address it should write to for the link. It does this in such a way that each of the 6 DSPs in a cluster uses a different address for the link hardware. This way, the FPGA can deduce the sender of data simply by looking at the high-order address bits. Any incoming data can then be matched with the appropriate address map register for correct destination address generation. The map register holds the upper 22 bits of address, i.e. It selects a 4 MB window in PCI space, and the lower 10 bits of address are from the incoming data’s address, i.e.

Where data is in the PLX's 4 MB window determines where the data will appear in the destination DSP's 4 MB window. To generate an address, the top 22 bits of the original destination address are masked off and replaced with the contents of the address map register. To change the contents of the address map register, the DSP writes to a known address in the PLX's space that isn't in the 24 MB link space.

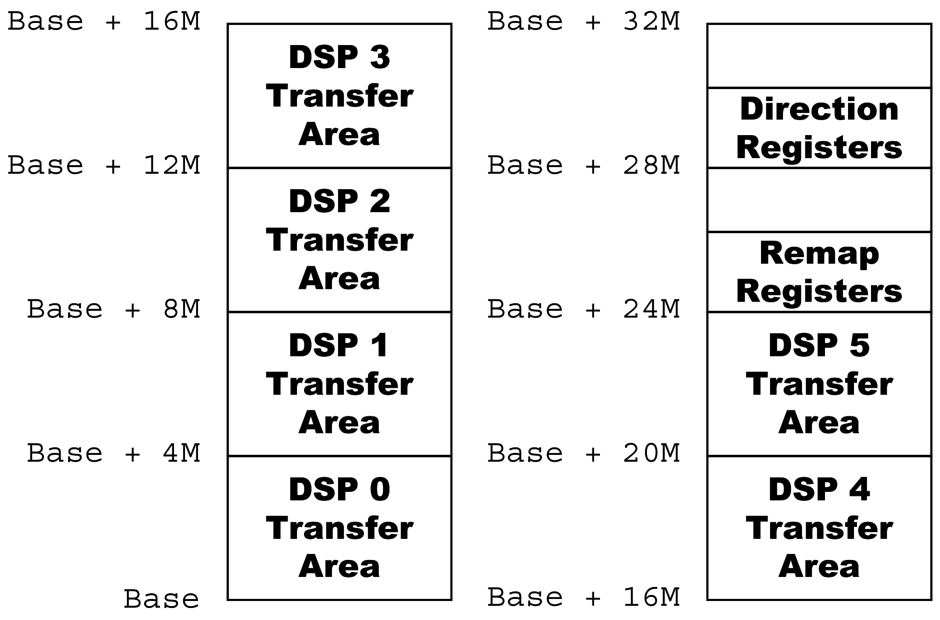


Figure 7. Address Mappings

Thus, the PCI module can correctly deduce and generate the true destination address for all incoming bursts, it can then route the data to the proper side, based on the routing ranges, and begin to fill one of the pair of FIFOs on that side for data entering the ring. The receive side of each transceiver module has two pairs of FIFOs to draw data from; some sort of arbitration rules need to be defined to chose which to draw from. Clearly, inside of a pair of transceivers, an alternating scheme should be used, to guarantee liveliness for all messages, since they are placed in the FIFOs in an alternating manner. What is not so clear is how which pair is selected. In this case, the FIFOs coming from the other transceiver unit always have priority over the PCI FIFOs . This is done to prevent deadlock. Messages already in the ring have priority, so they can work their way out before additional messages can be placed in the ring.

Thus data will always be drawn from the link FIFOs if there is data in them, and only from the PCI FIFOs when the link FIFOs are both empty.

2.3.2 Power

Components on the board use varying supply voltages. All chips require either 3.3, 2.5, or 1.5 volts. The PCI bus supplies 3.3, 5, +12, and -12 volts, not all of which are necessarily available in a system. Indeed, in a PC environment, it is common to have only 5, +12, and -12 available. Since the board is designed for a PC environment, it should be capable of running off of those voltages, and synthesizing any others it needs. To make the 3.3, 2.5, and 1.5 volt supply voltages, two power conversion modules are used. The first is a Power Trends' PT6502B that takes in a 5 volt supply voltage and makes a 1.5 v output that can supply up to 8 amps. The second is a PT6931C that also takes a 5 v input and can supply 3.3 v and 2.5 v outputs with 6 amps and 2.2 amps of current, respectively. The board features fuses on both sides of the power modules to protect from over currents, and LEDs attached to each of the power planes to indicate fuse status.

Large bypass capacitors are connected to both the inputs and outputs of the power modules to prevent noise on the power supply lines. Ferrite beads are also connected to the inputs of the modules to further reduce noise. The PT6502B is roughly 70% efficient, whereas the PT6931C is about 80% efficient. The two, in aggregate, can then draw about 12 amps from the 5 volt supply when they are both running at maximal output current. This 60 watt worst-case power burn exceeds the maximum allowed under the PCI specification. The specification says an expansion card my draw up to 25 watts combined, from the 3.3 and 5 volt supplies. Many PCI slots on PC motherboards are not even able to supply that much power. To remedy this, a standard disk drive power jack is included on the board. This jack gets ground, 5, +12, and -12 volts directly from the PC's power supply unit, and is capable of supplying much more current than is available over the PCI bus. This connector is fused separately from the PCI 5v supply, so the card can be configured to run exclusively on one or both, as desired.

2.4 Operation

Before high-level behavior can be realized on the system, it is important to understand the low-level mechanics of data transfer on the system. There are two different methods for transferring data from DSP to DSP. In the local case, both the source and destination are in the same cluster, so the transfer can stay on that PCI bus. In the remote case, the source and destination are in different clusters, so the transfer must use the link hardware. This complicates the mechanics, as the link hardware is not transparent, so it must be explicitly addressed by the sender.

2.4.1 DSP to DSP local

This is the simpler transfer mode, because the transfer happens directly over the PCI bus, without any intermediary. The sending DSPs imply sets the destination PCI address in its PCI Master Address register (PCIMA), the source data address (in DSP address space) in its DSP Master Address register (DSPMA), and the desired number of bytes to be transferred along with the start control code in its PCI Master Control register (PCIMC). The PCI hardware takes over from there, using an EDMA channel to complete the transfer without further intervention by the CPU core. The status of the transfer can be determined by polling PCIMC for a done or error code, or by waiting for a CPU interrupt to be generated by the PCI hardware, indicating the master transfer is done.

Ideally, this transfer would do the initial PCI address cycle and then burst data until the sender has sent all of the desired data. In reality, this transfer may be broken up for a variety of reasons: full FIFOs, PCI bus contention, or burst limitation of the DSP. Both the receiving and transmitting DSPs have a 32-byte FIFO buffer between the PCI module and the EDMA module that services the memory requests. This EDMA must ultimately share the memory bus with other DMA transfers and the CPU core itself, so it is possible that the PCI's memory requests do not get serviced as quickly as the PCI module is capable of generating them. If the sender's FIFO runs empty, the PCI module will simply end the current transfer on the bus and re-request the PCI bus to do another burst when data is again available. If the receiver's FIFO fills, it will disconnect the transfer, forcing the sender to retry the rest of the transfer later. It is also possible that other DSPs on the PCI bus request the bus while a transfer is going on, and the PCI arbiter grants it to them. In this case, the sending DSP must yield the bus and re-request it to finish the transfer. The sending DSP might also run into its burst limitation - due to the size of the byte count field in PCIMC, the DSP can transfer a maximum of 64k bytes in a transfer. Transfers of bigger sizes must be broken up explicitly. In all cases, the PCI module's address generation logic correctly generates the address for the resuming PCI address cycle by adding the original PCIMA with the number of bytes successfully sent, and the transfer picks up where it left off. This fragmentation of transactions can happen multiple times, all the while transparent to the DSP cores.

2.4.2 DSP to DSP remote

Unlike the case of intra-cluster DSP-to-DSP transfers, inter-cluster transfers are more complicated. Because the link hardware is not transparent, data must be explicitly routed through it. To do a transfer over the link, The sending DSP must first set its address map register in the FPGA. To do this, it sets up the control registers as above to do a 4 byte transfer to the address of the register (which it knows about from the host boot-time configuration) with the data being the remap value to be placed in the register. Once that transfer is complete, the DSP can the set up another transfer to send the actual data. This time the address in PCIMA is in the DSP's private 4 MB space in the FPGA's PCI space; the offset into that space determines the offset into the receiving DSP's window, so the lower 10 bits of the real destination address are used as the lower 10 bits in PCIMA, and the upper 22 bits are the base of the appropriate FPGA slot. As before, a pointer to the data to be sent is copied into DSPMA and the byte count and start code are put into PCIMC. The conclusion of this transfer may then be handled as before, though it should be noted that transmission and reception are divorced in this scenario due to the latency of the link hardware. Unlike the direct case, the completion of a transfer at the sending end does not guarantee reception, in whole or part, at the receiving end.

Transaction fragmentation works as before, with the sending DSP correctly generating the new address. Because the resumed transfer uses the same remap value as the first part, there is no need to re-set the link remap register. Indeed there is no way to change the remap register for a resumed transaction, so it is an important feature that the link hardware be able to correctly handle resumed transactions. Since the lower 10 bits of the generated address are the actual lower 10 bits of the true destination address, the new PCI address cycle will contain all of the information necessary for proper link transmission.

3 Message Passing

3.1 Overview

One of the metaphors the board is capable of supporting is the idea of a multiprocessor system. The physical platform of the system has the connectivity required to implement a message-passing scheme. Such a scheme is not natively supported by the hardware, however, and a set of semantics and software implementation of that behavior are required to realize the message-passing multiprocessor system. This software must be able to oversee message transmission, reception, and management of local resources necessary for message passing. The complexity is increased by the hardware's use of PCI ports for networking DSPs, and the separate high-speed interconnect for inter-board messages.

3.1.1 Message Passing on the Hardware

The message-passing semantics see the network as a queue, where messages are independent and do not interfere with each other. PCI, however, is by nature a memory-sharing design, where address space in the PCI domain is mapped to address space in the local domain through a PCI port. This is potentially problematic, as messages bound for a PCI port can possibly interfere with messages simultaneously bound for the same PCI port or messages already received at the destination processor. Software, then, must be very careful to avoid such situations, since the hardware is not capable of protecting memory in the shared region. Each destination must carefully allocate space for incoming messages, and each source must be careful to only write to the allotted destination space. The software must strictly adhere to the semantics established for memory management to ensure message integrity and independence. Similarly, care must be taken with messages bound for other boards, since these messages must first be explicitly sent to the high-speed interconnect. This leads to different message routing behavior for messages that are bound for a processor on the same PCI bus, which may be sent directly over the PCI bus, and messages that are bound for other boards, which must be explicitly sent through the serial link.

An additional complication arises in the C6205's PCI port implementation. On the device, incoming PCI memory accesses are handled by a direct memory access (DMA) channel, and thus, the processor core is not directly aware of transfers. The PCI port is, however, capable of interrupting the processor as a result of a command sent to the PCI port's I/O space. This mechanism is exploitable to make a destination processor aware of incoming data.

3.1.2 MPI

The message passing interface (MPI) is a well-established message-passing specification [MPI Forum]. It defines behaviors and a language binding to implement message-passing systems suitable for a wide variety of underlying hardware. It is designed to be flexible and portable, as well as usable in heterogeneous environments. Many public implementations exist that will run in a variety f environments [Gropp].

Under MPI, messages must be explicitly sent by the source and explicitly received by the destination. The send/receive can be buffered or not, synchronous or not, and local (function completion does not depend on operations of another processor) or not. The user of MPI may determine the mode of transmission or leave it up to the MPI implementation to decide. MPI also defines some environmental operations to control how processes are organized and talk to each other.

MPI does have limitations, especially considering this hardware; some of the environmental controls of MPI are not well suited to the limited resources of this DSP-based hardware, but instead are more suited to general-purpose computers. As such, it is not feasible to fully implement all of the MPI specification, but rather to create a messaging API based upon its behavioral semantics. A goal of this of this project is to create software that is performance tuned to the hardware platform; that is inconsistent with a fully compliant MPI implementation. Thus, for this project, MPI is used as a behavior model and inspiration for the message-passing methodologies, and not as a standard we try to achieve. The message-passing semantics work exactly as in MPI, however not all of the environmental support is implemented. While it would be possible to make a fully compliant MPI implementation, such was not practical or necessary to demonstrate the utility of the hardware. It is likely then, that a stripped down MPI-like messaging system that fulfills the needs of a message-passing application with as little overhead as possible is preferable to a larger, slower, more feature-laden MPI implementation.

3.1.3 A Messaging System

The MPI ideal that system behavior is based on views the interconnect as a set of independent FIFO buffers. That is, for each message, the sender puts the message in to a buffer, and the receiver removes it from the same buffer. This buffer is not affected by any other messages in the network, at the sender, or at the receiver. Conceptually, there are an infinite number of these buffers, and each sender/receiver pair gets its own buffer. This view of message passing interconnect dates back to early message passing machines where the hardware primitives were close to the send/receive abstraction, and did use FIFO pairs to physically connect processors [Seitz].

This idea of pairs of FIFOs has a very strict notion of independence, but acceptable ordering not so well defined. Clearly, there is no total ordering at the network or processor level, and MPI does not seek to impose such an ordering. Instead, messages must be non-overtaking, that is a message cannot be received if there is another message that matches the receive that was sent first. MPI also does not guarantee an ordering of the send and receive functions on the two different processors, save for the requirement that a send must complete before the matching receive may. MPI does offer send and receive functions that have stricter ordering, but unless the user specifically requests such functions, they may not assume a more strict ordering.

MPI also supports the idea of standard (non-buffered), buffered, and synchronous sends, as well as immediate (blocking) and background (non-blocking) communications. Standard send may be buffered or not, buffered send must be buffered if immediate transmission cannot take place, and synchronous send requires the matching receive to have started before it will complete. The idea for buffering is that it can allow total locality of functions, i.e. a buffered send can complete without the matching receive. The system will buffer the message and send it when possible. Buffering decouples the send and receive, in exchange for consuming more system memory. The idea of non-blocking sends and receives is that the they will be started by the initial call, and the communication can occur in background, while other computation is going on. Blocking send function calls do not return until after the data has actually been sent, thus indicating it is safe to overwrite the message data, whereas nonblocking sends return immediately, but may not have actually sent the message. Non-blocking functions de-couple the start of a send or receive from the completion of the send or receive.

The messaging system implemented here is somewhat simpler than MPI. Many other scaled-down versions of MPI exist, such as Ames Lab's MP_Lite [Ames Laboratory]. In the case with this messaging implementation and other scaled down versions, MPI-like behavior is desired, but we do not wish to pay the full overhead of an MPI-compliant system. It is certainly possible to build a fully compliant system by extending this scaled down version to cover the neglected portions of the MPI specification. Rather than full MPI support, a goal for the messaging system is to support simple sends and receives at the user level, while consuming as little system resources as possible.

The network is conceptually limited to a finite number of FIFO buffers. Specifically, a FIFO exits between each potential sender/receiver pair. This means the strict message independence is preserved, but the loose ordering is tightened by forcing messages to be transferred one at a time between a sender and receiver. Unlike MPI, a sender that wants to send multiple messages to the same receiver must wait for the first send to complete before the second may begin. The communications semantics also tighten up the ordering of the send and receive on the two processors. Under this scheme a receive must begin before a send can complete, in addition to requiring that a send complete before its matching receive. This means that all sends are analogous to MPI’s synchronous send. Implementing only synchronous sends simplifies the code a great deal and economizes on system resources. The buffering consumes memory and time doing the memory copy, and it does not generally make sense to buffer synchronous sends. Buffering can always be implemented in the user part of the code, if need be.

Though the messaging protocol only supports one of MPI’s send modes, it does however, support both blocking and nonblocking communications. Since the utility of both is apparent; blocking sends are useful for synchronizing computation across processors, and non-blocking sends are useful for sending data while not wasting processor time. This behavior is not so easily emulated in user code, therefore it is worthwhile for the messaging code to provide this capability.

Another difference from MPI is that the user is responsible for managing buffer space. In the case of sends this is simply a matter of not overwriting the data until the message has been sent, as the send does not buffer the data. In the case of receives, the user must allocate space in the designated receive space and is responsible for not interfering with other messages in the receive space. Arguably, the messaging code should provide receive buffer memory management. However, there exist functions that behave like standard malloc and free to assist users in dynamic, safe memory allocation. If the messaging code were to provide this memory allocation, it would use the very same functions to do it. Since for some applications such memory management may not be necessary, for simplicity and compactness, receive buffer management is left to the user. Code for this message passing implementation can be obtained from http://www.csg.lcs.mit.edu/publications/.

3.2 Implementation

The message-passing implementation is most complicated by the C6205's PCI port behavior. It becomes obvious that the PCI network is not directly analogous to the independent FIFO conception when mapping this ideal to the PCI hardware. It is, however, analogous to DMA transfers across the network. Thus the message passing becomes memory copying across the networking medium [Barton et al].

Specifically, message independence is something that the implementation must pay careful attention to. The PCI port's DMA-like memory access is ideal for sending messages; it is simple to conceptually map it to filling a FIFO and it relieves the core from data transmission almost entirely. This memory access is extremely problematic for the receive behavior, as the receiving DSP has no control of where the data goes or whether or not to accept data. This clearly does not map conceptually to emptying a FIFO, as the potential for harmful data interaction is evident. Much of the control aspect of the message-passing software, then, is dedicated to guaranteeing message independence at the receive side.

To implement message passing on this hardware, portions of the PCI-addressable memory space must be allocated for message reception. This allocation of receive space leads to a handshaking protocol where a sender must request space before actually sending a message. Since, like messages, these requests must not interfere with other requests, dedicated space must exist on each processor for each other processor to make requests to. In this fashion, requests can be made to a static and known request area in the destination's PCI space. The receiver can then respond to the source's static and known response area, indicating a memory allocation for sending the message. The sender may then send the message without interfering with other messages bound for or already at the destination. Thus, a receiver must explicitly allocate its receive space for message buffering and independence. The allocation is not enforceable by the hardware; it is the responsibility of the transmitter to strictly adhere to the space allocation.

3.2.1 DSP BIOS

This project makes use of DSP BIOS - a set of libraries built in to the C6205's development environment to efficiently implement such programmatic constructs as threads, locks, mutexes, mailboxes, heaps, memory management, and software interrupts. These extend the basic development toolset beyond the optimizing C compiler, to include these objects that typically aren't implemented on DSPs. The BIOS also allows for rudimentary multitasking by supporting a simple threading model.

This threading model is neither quite preemptive multitasking nor cooperative multitasking. The basic premise is that of prioritized threads where high priority threads preempt lower priority threads and threads that are blocked have no priority, is in preemptive multitasking. However, context switches only happen when calling certain BIOS functions. Thus, a thread can execute indefinitely so long as it does not call any of those BIOS functions, as in cooperative multitasking. If a thread does call a function where a context switch is possible, it will be pre-empted if any threads of higher priority are ready to run. Threads are not protected from each other, as every thread has full access to the memory and peripherals of the DSP

This is both advantageous and problematic at times. In much of the code, atomicity can be guaranteed by simply not calling functions that can cause context switches, eliminating the need for explicit atomic functions. The lack of hardware protection negates the need for a kernel to access the hardware, as any code that needs to can access the hardware directly. The drawbacks are that care must be taken to prevent a thread from deadlocking, and to prevent threads from interfering with each others' peripheral usage.

3.2.2 Resource Allocation

The messaging software has limited room to work with. There is 8 MB of external program /data memory, 64 KB internal program memory, and 64 KB internal data memory. This must contain the message passing code, the user-defined behavioral code, the message passing data and user data. Without a specific user application to guide the allotment of memory to the various parts, it is difficult to make an argument for a particular partitioning, so a general one is made that seems reasonable. 4 MB of external memory - an entire PCI window - is reserved for message passing use. A small portion of the rest of external memory is reserved for messaging environment data. The rest of the external and all of internal memory is used for the application, that consists of the user code and data as well as the message passing code and data. The BIOS development environment allows easy relocation of code segments. Since there is no compelling case for where to locate various parts of the system without an application in mind, all of the messaging code goes in a named memory region that gets mapped to the internal program memory. Similarly, the user code and data get mapped into internal program memory, and external memory, respectively. These mappings are moveable so that an end-user of the system could allocate resources as the application required. Since it is not unrealistic to expect to fit the message-passing code and user code in the 64 KB program memory, both are placed there for maximum performance.

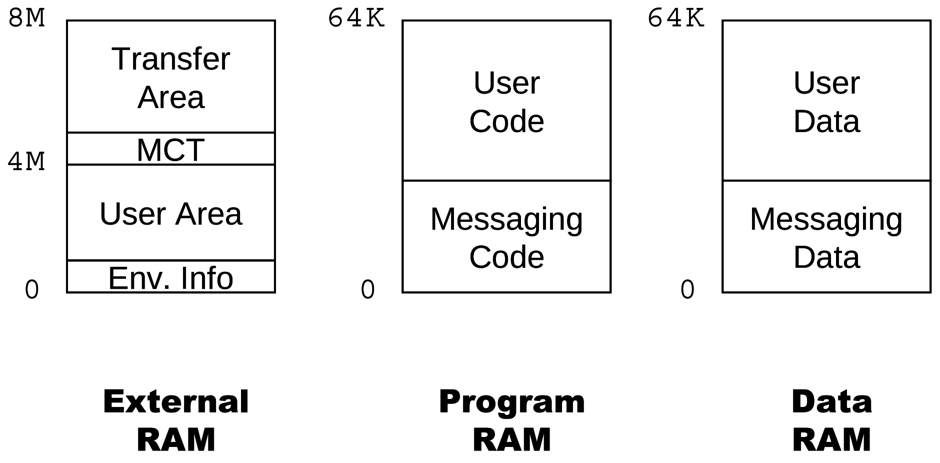


Figure 8. Memory Allocation

3.2.3 Boot Time Setup

The processors need to know information about their environment, and the wholly decentralized nature of the message-passing mechanism provides no good way for self-discovery. Specifically, each processor needs to know the PCI address of every other processor’s memory window, the PCI address of every other processor’s memory-mapped register space, the location (in-cluster or out-of-cluster) of all the other processors, the PCI address of the link hardware, the PCI address of the link hardware remap register, the processor’s own ID number, and the number of processors in the system. All of this information is assigned a fixed location in the processor’s memory, and the host boot code is responsible for filling the specified memory locations with the proper information.

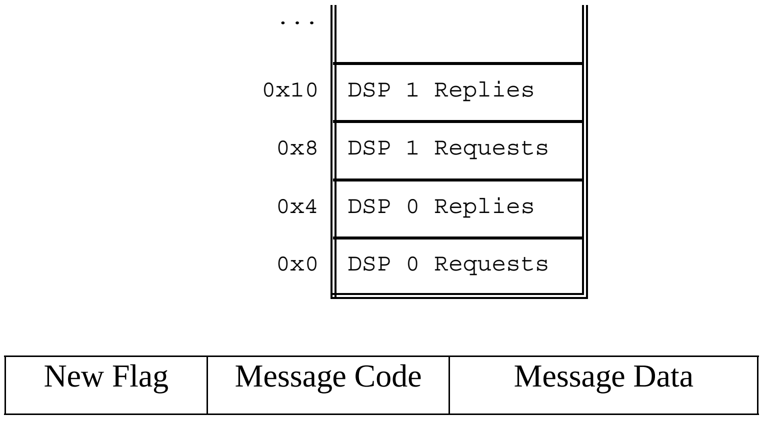
In addition to providing the messaging environment information, the host boot program must initialize the DSPs and put them in a useable state. When the board boots on the PC, the DSPs cores are held in reset, with the peripherals come online to configure the device. The host boot code must configure the EMIF to be able to interface with the attached external SDRAM, by setting configuration registers to describe the size, type, and timing parameters of the RAM. With full access to all memory, the host can then load the messaging and user code into the processor, followed by the environmental info the messaging code needs to run. Finally, the host awakens the DSP core, which starts executing the loaded program.

3.2.4 Program Data

To implement message passing, there are some data constructs that occupy fixed locations in memory so that processors know how to communicate with each other. Most important is the messaging control table or MCT. The messaging control table is a region in memory that is the space for other processors to send requests and acknowledgments to. It is always located at the base of a processor’s PCI address space, and processors use their own unique processor identification numbers (PIDs) to index into this table when writing data. The table consists of two words per processor. The first is space for making requests, and the second is space for acknowledgments. It is necessary to have both because two processor might send a message to each other simultaneously, and the messages involved in the handshaking would get confused if there were not separate request and acknowledge words. This allows a processor to have a unique, known space on every other processor in the system for communicating. All handshaking done by the messaging protocol occurs through the MCTs on the processors.

Figure 9. MCT Layout

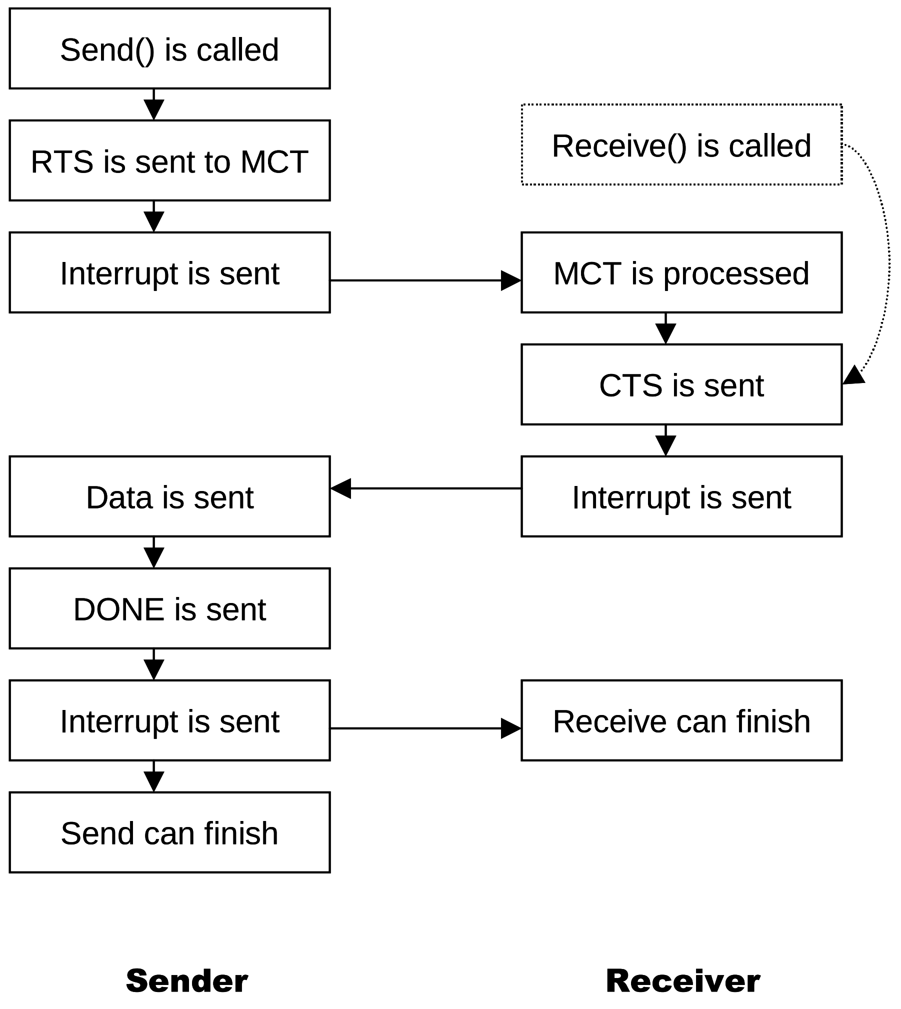
The MCT is not enough for efficient implementation, though. Due to the nature of the PCI port on the C6205, it is difficult to tell when data has arrived that needs processing. To work around this, the messaging protocol uses explicit signals to inform other processors of important new data. This explicit signal is a hardware interrupt than can be generated through the PCI port. Every processor has a register at a fixed offset from its memory mapped register PCI space that can generate a CPU interrupt.



3.2.5 Program Behavior

At a descriptive level the send and receive works as follows. The sending program writes to its space in the receiver's MCT where it can make requests. It then sends an interrupt signal to the receiver. The interrupt handler wakes up the thread responsible for processing new requests. The thread parses the MCT looking for new requests, and it finds the request just received. The receiver then allocates space in the receive buffer for the message and replies to its proper spot in the transmitter's MCT and sends the transmitter an interrupt. The interrupt awakens the transmitter's MCT processing thread, and it discovers the space allocation in the receiver's buffer. It then sends the actual message to that address, and when finished indicates so by placing a "done" indicator in the receiver's MCT. An interrupt again awakens the receiver's mailbox processing thread, where transmission completion is noted, and the local receive can complete.

Figure 10. Transmission Flow



This behavior, as described, is a complex interaction of different parts of the code. The code can be broken down into five segments: the user code, the interrupt handler, the mailbox table processor, the send code, and the receive code. The user code runs in its own thread, as does the mailbox table processor. The send and receive code may be running in separate threads, as is the case for nonblocking mode, or in the user thread directly. The hardware interrupt handler exists outside of the scope of the thread manager; conceptually it is its own thread as well, but with different preemption rules.

Since these pieces of code run in different threads and must share data with each other for proper message passing, a method of communication must exist amongst them. As an additional complication, the order of sends and receives is unknown, so the inter-thread communications method must be able to handle either happening first. This is only a complication for the receive code, as the send always happens first at the sending processor. This is because the clear-to-send (CTS) message is only sent as a response to a ready-to-send (RTS) message. Thus until the send code causes the RTS to be sent, the receive code executing on the receiving processor will not affect the sending processor in any way.

This inter thread communication is handled with BIOS mailbox objects, because they implement the data sharing and requisite thread blocking and context switching ideally. The communication is slightly different for sends and receives.. In the context of receives, a call to a receive function creates a mailbox for getting the data associated with the transfer (namely the location of the received data and the size of the message). A static array of mailbox handles is defined such that there is an element for each source processor (process IDs serve as the index) . The MCT processing thread can then put information into an existing mailbox or create a new one, depending on whether the send or receive happens first. In the send context, it works slightly differently. Instead of an array of mailbox handles, one mailbox is used to register the other mailboxes created by the send threads. Thus the MCT processing thread, scans through the registered mailboxes and selects the right one to communicate the CTS to the proper sending thread. This is slightly more space efficient than a static array of mailbox handles for system where there are a large number of processors. Since sends always happen before receives at the sending processor, there is no need for the MCT thread to create mailboxes in response to a new CTS.

The PCI hardware interrupt is the simplest of the individual pieces. It is called every time the PCI port generates an interrupt. The PCI port generates an interrupt when a remote processor requests one by setting a bit in the PCI port's memory-mapped control register, when a master transfer has completed, or for an error condition. In the case of the remote request, the interrupt handler sets a processing flag and wakes the mailbox table processing task by setting its priority to a non zero value. In the case of a completed transfer, which indicates data has been successfully sent out, the interrupt handler wakes the waiting task by posting to a semaphore created for such a purpose. If the handler is invoked due to an error condition, it halts the system and reports the error.

The messaging control table processing thread simply scans the messaging mailbox area, first looking for CTS replies. It does this by scanning through all the table locations where one might occur looking for an MSB set to '1'. If one is found, the registered send thread mailboxes are scanned until one indicating the destination PID of the send operation matches PID of the processor sending the CTS. The buffer allocation of the CTS is then placed in the send thread's mailbox, and the CTS message is marked as read by setting the MSB to '0'. The mailbox table processing continues looking for additional CTS messages. After a scan through the table, it scans through again, looking for new RTS messages in the same manner. If a new one is found the proper receive thread is notified by using the mailbox associated with the handle stored in the array at the sending processor's PID. If there is no mailbox handle in that array spot, one is created. Either way, the mailbox then gets the size of the message to be sent. The RTS message can then be marked as read just as a CTS message. When the scan of RTS messages has completed, the mailbox processing checks to see if any new interrupts have happened since it started scanning. If new interrupts were generated, the scan is started again. If not, the task goes to sleep indefinitely by setting its priority to -1. It will be awakened by the hardware interrupt routine when new data arrives.

The send code can work in either blocking or non-blocking mode. The behavior is exactly the same, but in blocking mode, the low-level send function is called directly from the thread that called the high-level send function, whereas in non-blocking mode, the high-level send function spawns a new thread that just executes the low-level send function.

The low level send function takes, as arguments, the destination PID, a pointer to the data to be sent, a count of how much data to send, and optionally, a pointer to a semaphore to alert upon completion. The semaphore alert only makes sense in the context of non-blocking sends. The send code first creates a mailbox object to communicate with the MCT processor, and registers it by placing a handle to the new mailbox object in another mailbox used by the MCT processor.

The code next sends an RTS message to the destination processor using the send to messaging control table function defined later. When the RTS has been sent, the code waits for the CTS reply by executing a pend on its mailbox. This pend effectively puts the thread to sleep until the mailbox is posted to - when the CTS reply is received. When the thread resumes execution, the CTS reply is checked and the destination address is extracted. The newly created mailbox is no longer needed, and is therefore destroyed.

With a proper address to send data to and permission granted, the send function sends the actual data using the send to buffer function defined later. As soon as this has completed, the code can then send a done message to the remote messaging control table, indicating the transfer has successfully completed. Finally the function need only alert the waiting semaphore, if one was indicated, and exit. A potential problem arises, however in this semaphore alert. Under the threading scheme BIOS implements, posting to the semaphore could wake a higher priority task and prevent the send function from ever reaching its exit code. The send task would forever be in the pre-empted, and not terminated state, so the thread delete function could cause problems (it is generally a bad idea to delete threads that are not finished executing). To get around this, the send code boosts its priority to the maximum, so that it cannot be pre-empted when it posts the semaphore. The semaphore post will then change the waiting thread to the ready state, but not transfer control there. The send code retains control and exits, where the thread scheduler marks it as terminated, and transfers control to the waiting task (or any other task, as per its scheduling schema). Thus, the waiting semaphore can be alerted of the send success and the send can be allowed to complete, so that it may be safely deleted.

Exactly like the send code, the receive code can operate in blocking and non-blocking modes. In both cases the functionality is the same, the difference is what thread the low-level code is called from.

The low-level receive code takes as arguments, the source PID, a pointer to the allocated buffer space and an optional semaphore handle.. The receive code first checks the array of mailbox handles to see if the messaging control table processor has already created a mailbox with the matching RTS request. If it has not, the receive code creates one and waits for a RTS to come in. If the mailbox message already exits, it does not need to do anything, and may immediately continue. The CTS is then sent back informing the sender of the location to send data. Currently the code requires clairvoyance on the part of the sender and receiver to match message sizes. This allows the messaging code to avoid the need to do any memory allocation. The user could implement an end-to-end solution to the potential problem of how to allocate receive buffer space, or the code could be modified to make use of the fact that the RTS request includes a word count.

Regardless, the receive code sends a RTS to the sender by using send to mailbox function in reply mode. The receive code then waits for the done message. Due to the ill-defined timing of the whole system, the receiver might not process the done before the same sender sends a new RTS. Thus the receive code must check the next message the MCT processor passes along to it. If it is a done message, the receive code can simply delete its MCT mailbox, and exit. If the done message is, in fact, a new RTS, the receive code simply leaves it in the mailbox for the next receive to process and exits. In both cases it uses the same priority boosting trick as the send code to be able to alert a potentially waiting semaphore and terminate successfully.

The send to messaging control table subroutine handles the PCI port interaction to send data to a remote processor’s messaging control table. It takes the PID of the destination, the data word to be sent, and a flag indicating if the message to be sent is part of a receive or send operation. The routine must calculate where to send data, and appropriately configure the link hardware if necessary.

The function first acquires the PCI port lock, to assure no other sends or receives happening on the processor can interfere with the PCI port. The PCI address of the destination’s message control table can then be looked up in the table. The offset into the remote MCT can also be calculated based on the local PID and whether or not the data is part of a receive or send. If the data is to be sent through the serial link, the link is set up by sending the desired base address to the address remap register in the link’s PCI space. The PCI port is then set to send to the link hardware’s base address, plus the calculated offset. In the intra-cluster case, the PCI port is simply set to send to the destination’s base address plus offset. The code then makes use of a semaphore to wait for the PCI transmission to complete.

Once the data has been sent, the code then sends an interrupt to alert the destination processor of new data. This is done in exactly the same manner as for sending the data, except the PCI address sent to is the address of the destination processor’s interrupt generating register, and the data is the interrupt message. Again, a semaphore signal is used to indicate the completion of the transfer. The function can then release the PCI port lock and exit.

The send to buffer function works much like the send to messaging control table function, but it is used to send the data of actual messages to allocated receive buffer space. The code takes as arguments the destination PID, the address of the remote buffer, a pointer to the data to be sent, and the number of words to send.

The PCI port lock is acquired, and the PCI port is then configured. The address to send the data is expressed as an offset from the remote processor’s messaging control table. As before, this address is either sent to directly or via the link hardware, with requisite remap register setup. Once the PCI port has been instructed to send the specified number of words, the code waits on a semaphore to indicate completion of the transfer. Unlike the send to messaging control table, this transfer is not followed by an remote interrupt, as it is not necessary. The code can therefore immediately release the PCI port lock and exit.

Before any messaging sends or receives can be executed, the initialization function must be called. This function simply waits for a dummy message from processor 0. Processor 0 is, by convention, the last to start and sends this dummy message to all the other processors. This serves to be sure all processors are up and running before any try to transmit messages. Otherwise, it is possible to have the MCT corrupted by the host boot program or have a system error generated by trying to write to the MCT of a processor that has not had its memory space configured.

4 Processor In Memory

4.1 Overview

Another metaphor the hardware is capable of supporting is that of a general-purpose accelerator card. In this view, the DSPs act as regions of memory with special capabilities in a host computer’s memory space. The host system can use the DSPs to process data independently of the main CPU. This allows inexpensive systems to be built that combine the flexibility of a PC with the computational power of high-performance DSPs.

4.2 Using the Hardware

Taking this view of the system, the strengths and weaknesses of the design are different than for message passing. The PCI interconnect that proved somewhat awkward for a message passing implementation becomes a virtue of the system, and the serial transceiver hardware that let a message passing implementation scale well becomes a nearly useless appendage. Like the message passing system, the processors are completely independent, but unlike the message passing system they do not interact with each other. Rather, they interact with the host PC's processor mainly, which is why the PCI port is beneficial. Since processors do not need to talk to each other, the link hardware becomes substantially less useful than it was for message passing.

The mechanics of PIM system operation are simple. The host PC configures the DSP to do the operation it wants done autonomously by loading a program into the DSP. It can then set up the PCI port to window to the external RAM on the DSP and transfer the data there. The DSP can begin processing the data, and indicate that it is done. At that point the CPU can fetch the processed data whenever it desires.

The specifics of the operation can be tailored to the application. Since the DSPs have bus master capable PCI ports, getting the data into the DSP's local RAM can either be pushed by the host CPU or the DSP itself can grab the data out of the host's main memory. If the data is placed in the DSP's memory space by the host PC, The DSP needs to know when to start processing the data. This can also be accomplished in a variety of ways; the host PC can generate an interrupt on the DSP to alert it, much like in the message passing scheme, or the DSP can poll its memory looking for the appropriate condition to begin processing. If the DSP is responsible for grabbing its own data from memory, it must know when and where to get it from.

Similarly, upon completion of data processing, the host PC must be made aware of the availability of processed data. This can be accomplished through pooling or interrupts as well, as the DSP is capable of generating an interrupt on the host PC. Another issue is completed data placement: the processed data could remain on the DSP's memory or be pushed back into the main memory of the host PC.

Since these options are more or less interchangeable, the optimal choice depends on the goal of the PIM implementation. If it is to maximize overall system efficiency, then it must be considered whether the CPU or the DSPs are better suited for doing a particular task. Another possibility for the system goal is to minimize implementation complexity, which would dictate the systems use the simpler choice in a decision. In an implementation of PIM that was designed around a specific application, the balance of overhead work done by the DSPs or host CPU would be determined by the performance demands of the application. I.e. In an application that was very host CPU intensive, and needed the DSPs less intensely, the PIM setup would want to make the DSPs do as much of the overhead work as possible. Conversely, if a system were very dependent on the DSPs and used the host CPU mainly as a coordinator, the host CPU should do most of the work.

In all cases, the interrupt method of acknowledgment is more efficient than polling for status information. It also does not require significant work from the interrupt generator. For the host CPU to generate an interrupt on one of the DSPs, it need only send a word to the correct PC address. Similarly, for the DSP to generate an interrupt on the host, it need only set a control bit in its own PCI port that activates the hardware interrupt line attached to the device. The interrupt method is, however, more complicated as it involves the creation of interrupt service routines to handle the interrupts. For the DSPs, ISRs are not particularly difficult to write, but they can be much more troublesome for the host CPU. For this reason, systems that aren't concerned with absolute maximum performance might want to use interrupts on the DSPs, but have the CPU poll where appropriate.

4.2.1 CPU Intensive

First, we consider the case where host CPU cycles are precious, so the PIM system should put as much of the burden on the DSPs as possible. In this case, it is optimal for the DSPs to handle the memory transfers. The host CPU indicates to a DSP that it has data that wants processed. The DSP would then fetch the data out of the processor's memory, perform the desired operation, push the data back into main memory, and alert the CPU of the completion.

This has the desired benefit of involving the host CPU as little as possible. Most importantly, the host CPU can do the entire transaction using only main memory, which is much faster than the PCI bus. A slight tradeoff would be to have the CPU generate a DSP interrupt to begin the procedure, to alleviate the DSP from needing to poll main memory looking for work. This would reduce significantly the memory transactions generated by the DSP, an would only cost a one-word transfer on the PCI bus for the CPU.

4.2.2 DSP Intensive

Conversely, another usage case to consider is that where the DSPs are used intensively and the host CPU is not. Here, it is optimal for the CPU to handle memory transactions. The host CPU would put the data to be processed into the DSP's memory and indicate that processing should begin. The DSP can then process the data and indicate when done, and the host CPU can then read the data out of the CPU. In this setup the DSP works only with its own memory, which is faster than accessing main memory over the PCI bus.

Since the PCI address to DSP local address mapping is controllable on the DSP, this presents some good tricks for committing the completed data. The DSP could read data out of the buffer the host wrote to, and assemble data elsewhere in its memory while working on the task. When done, rather then copy the finished data, it can simply change the PCI window mapping, and instantly give the host CPU access to the completed data. This also appears as an atomic commit to the host CPU - all of the new data shows up in the same PCI address space simultaneously. This could be an important feature for systems that do not do strict acknowledgment.

5 Evaluation and Performance

5.1 Overview

This thesis not only proposed a hardware architecture and some software for implementing various systems, but actually built some boards and implemented the message passing system. A Processor in Memory implementation was never attempted due to time constraints. Various parts of the system met with varying degrees of success. The hardware was mostly functional, though not perfectly so, and the software performance suffered somewhat from the imperfections of the hardware.

5.2 The Board

The board, as described earlier, was built and tested. It was initially nonfunctional, however, after some debugging, most of the system was made usable to at least demonstrate the feasibility and performance of the architecture.

Some of the bugs with the board were correctable with solder and wire, and others could only be corrected by respinning the board, which was not done.

Aside from the issues discussed below, there were no problems with the board. I was able to fix the boards well enough to get them to boot up in a PC, configure them over PCI, run them in the debugging environment over JTAG, and get them to talk to the rest of the system.

5.2.1 Easily Fixable Problems

When the board was first delivered from the fab, it was not only nonfunctional, but it would prevent a host PC from booting at all. Extensive testing with a PCI bus analyzer showed that the problem could be traced to DSPs not responding to configuration cycles from the host PC's PCI BIOS. Further debugging revealed that the IRDY# and TRDY# lines had been switched on some of the devices on the secondary bus. Thus the DSP would assert TRDY# to acknowledge the cycle, but the host PC would never see this, and the system would deadlock. Fortunately all of the devices that had this problem were TQFP-style packages and not BGA-style packages. This problem was fixed by carefully unsoldering and swapping the TRDY# and IRDY# pins from the PCI bridge, the PLX PCI interface chip, and the PCI slot connector. This fix completely solved the problem, and the impedance mismatch introduced by the correction wire did not seem to adversely impact the system

After the PCI fix made the board boatable, testing the DSPs on the board revealed a problem with the external memory. Though an 8 MB external RAM was attached, the DSPs ere seeing only 2 MB of it. The other 6 MB that should have been the rest of external RAM was just 3 copies of the first 2 MB. The source of this problem was that tow bits of address into the SDRAM were never connected to the DSP in the schematic. Fortunately, all of the necessary pins were accessible directly or by vias on the board, and this problem was fixed with correction wire. Again, the wire impedance mismatch and lack of serial terminating resistor did not seem to cause problems.

Once the DSPs could access their entire external RAM, further testing revealed that data was sometimes corrupted when transferring to and from the external RAM. The corruption only happened when working with C char-type variables. This corruption was caused by crossed byte enable control lines between the DSPs and the SDRAMs. Fortunately, like the PCI bug, the TQFP SDRAM package allowed this to be fixed with some careful soldering. This fix, too, did not cause problems.

These previous three fixes brought the board to a point where it worked well for developing a message passing implementation. When it came time to test the performance tuning capabilities of the system, it was discovered that the PCI minimum grant value could not be stored into the DSP's boot EEPROM. This was because the data in and data out lines of the serial EEPROM were crossed. This error could have been fixed with solder and wire, like the others, but was not, due to time constraints.

5.2.2 Not Easily Fixable Problems

The board has a JTAG scan chain for a debugging interface to the DSPs, and this scan chain followed the recommended procedures for designing a multiprocessor scan chain. I could never get all six DSPs on the scan chain at once. At best, the scan chain will work with the four closest to the connector enabled and the other two bypassed. I was unable to determine the nature of the failure, but I was able to verify that it was not the result of a schematic miswiring. This suggests some more insidious scan chain integrity issue.

Fortunately, this limitation is far from fatal. The 4-DSP scan chain is sufficient to run interesting message passing systems and observe the results. The other two DSPs are also still fully accessible through PCI, so they can participate in the system, just without the debugging functionality offered by the JTAG interface.

Another problem that is not easily fixable, but is not disastrous involved the DSP reset logic. The DSPs need to be held in reset for a certain amount of time to reset properly. On the board, the DSP resets are controlled by the FPGA, which hold the DSPs in reset as long a the PCI reset signal is active. A problem arises in the timing however. Since the FPGA must load its configuration from and EEPROM upon power up, the FPGA cannot always see the reset signal, due to its own power on latency. This is especially true since the reset only happens when the PC begins booting, which is typically not long after power is applied. As a result, the DSPs are not always reset properly. When that happens, they don’t respond to PCI configuration cycles, and the host PC refuses to boot.

A solution to this problem is simply to buffer the PCI reset signal and provide it to the DSPs. This would require adding additional logic to the board and changing traces. Since both the FPGA and DSPs are in BGA packages, this is not solvable with correction wire. This is only a sporadic issue, as the failure is intermittent at boot time. While annoying, this bug does not prevent the use of the hardware.

Unlike the other problems with the board that were easily fixable or not critical, I found a catastrophic problem with the link hardware. Tests shoed the link hardware units were not talking to each other. The problem was traced all the way to the serial gigabit transceivers, which could not establish a link between themselves. Investigation showed that a badly timed input clock was the most likely culprit. Since the transceivers use an internal PLL to multiply a 125 MHz input clock into a 2.5 GHz serial clock, they require the input clock to have very little jitter (60 ps, peak-to-peak, maximum). The specifications of the clock generator parts in use for the system do not meet that jitter specification, and observation on an oscilloscope confirmed the parts were getting clock signals with around 300 ps of jitter.

An attempt was made to find a quick fix. The clocking parts on the board were replaced with parts that did meet the jitter specification. However, the replacement parts were not pin-compatible with the existing parts, so they had to be connected to the board with correction wire. This was not sufficient for the integrity of the high-speed, ultra-accurate clock. While the new parts did reduce measured clock jitter at the transceiver, it did not lower it enough to meet the 60 ps specification or allow the transceivers to operate.

Without the transceivers capable of even so much as loopback functionality, the link hardware on this version of the board is completely non functional. It would require a respin of the board to fix the clocking issue. Time and resources did not permit this.

5.3 The Message Passing System

The message passing software as described was implemented and debugged. The system was tested and able to send and receive in all modes properly. The functional system was then put through a test program that had DSPs exchange blocks of data to test system performance. Two versions of this test program were run: one where two DSPs exchange data, and one where four DSPs exchanged data simultaneously. Performance metrics were then gathered

5.3.1 PCI Utilization

Efficient use of the networking medium is crucial to message passing performance. In this system, the PCI bus saturated easily, and a hardware bug prevented the system from optimizing bus usage. The DSP Boot EEPROM bug eliminated the ability of the system the DSPs to request minimum bus grant times. This means the PCI arbiter will potentially force a device that is using the bus to surrender it after only one word of data transfer. Thus, without minimum grants, the burst capability of the PCI bus is essentially eliminated when two or more devices have data to send.

For our message passing system, this means that any time two messages are being sent simultaneously, the transfers happen one word at a time, and the PCI bus utilization hits 100%. Without fixing the hardware bug, it is not possible to change this behavior. Thus, while the PCI bus does achieve full utilization, under heavy load, most of the cycles are for address and wait and not data; the PCI bus use is inefficient. Ideally, it would be possible to set a minimum grant of the PCI bus to allow bursts to happen even under heavy bus load, thus improving the efficiency of the PCI bus at full utilization.

5.3.2 Transfer Throughput

An important measure of the message passing system’s performance is its ability to quickly send messages and the effort required to do so. Two metrics to measure system performance by are communications bandwidth and the system's ability to hide latency [Hennessy et al.]. This system did reasonably well on both counts, but especially the latter, where nearly all of the clock time involved in communications is time that the processors can spend doing other useful work on top of the communications. This allows a potential application to mostly avoid the tradeoff between communication effort and computation effort [Naik].

The system communications throughput was tested with two-way and four-way simultaneous transfers, where the send and receives were done simultaneously in the background, and two-way sequential transfers, where the sends and receives were ordered to ensure only one was happening at a time. The tests were set up with block sizes of 1 KB, 4 KB, 16 KB, and 64 KB. In all cases, all participating DSPs sent every other participating DSP one random block of data and measured the time elapsed from the start of transfers to completion of all sends and receives. The results can be seen in the tables below.

Type / Size	1 KB	4 KB	16 KB	64 KB
2x Sequential	342,936	372,370	449,439	792,383
2x Simultaneous	396,932	431,654	516,874	857,629
4x Simultaneous	993,221	1,052,410	1,364,599	2,389,351

Figure 11. Performance Results: Average Total Transfer Time in CPU Cycles

Type / Size	1 KB	4 KB	16 KB	64 KB
2x Sequential	1,166	4,297	14,240	32,308
2x Simultaneous	1,008	3,707	12,382	29,849
4x Simultaneous	805	3,041	9,380	21,428

Figure 12. Performance Results: Average Throughput in KB/s

The absolute best case transfer throughput is determined by the PCI bus. Theoretically, the system could get up to 132 MB/s of throughput, though a more realistic maximum sustained throughput for PCI is on the order of 50 MB/s. In these tests, the best throughput obtained is about 32 MB/s, which includes the messaging protocol overhead. The marginal throughput, which removes the protocol overhead from the calculation, is much closer to PCI’s realistic maximum; it is 56 MB/s for the 64 KB block case.

This performance is impacted by the efficient use of the underlying network, so the PCI utilization problem affects this performance. This can be seen in the marginal throughput of the four-way simultaneous case, which drops to 32 MB/s. As there is more competition for the PCI bus in this case, and since the system has no way to efficiently allocate bus usage, more of the PCI transactions are prevented from bursting, lowering the throughput.

5.4 Conclusion

The DSP multiprocessor system designed for this thesis was robust, inexpensive, and scalable. At a marginal cost of about $2,000 per board, this design certainly succeeded in its cost design goal. System performance and scalability goals, however, were not met with such overwhelming success, as implemented. The actual systems built did not fully support their potential, due to implementation mistakes more so than design flaws. Major implementation blunders prevented the efficient utilization of the networking medium in high-demand situations, and prevented the use of the high-speed inter-board link at all.

Despite these flaws, the system’s capability was, nonetheless, acceptable. The system was able to fully utilize, though inefficiently, its communication network. The system's ability to do processing and communication simultaneously contributes to its high performance potential. Further revisions of the implementation would fix that which hinders the current system from realizing its full design potential.

References

Ames Laboratory. 2001. MP_Lite: A Lightweight Message Passing Library. http://cmp.ameslab.gov/MP_Lite/MP_Lite.html

Analog Devices. 1998. A New Standard for Multiprocessor DSP Systems. http://www.analog.com/publications/whitepapers/products/sp_mp_wpp/sp_mp_wpp.html

Barton, E., J. Crownie, and M. McLaren. 1994. Message Passing on the Meiko CS-2. *Parallel Computing* 20(4):497-507

Culler, D. E., and J. P. Singh. 1999. *Parallel Computer Architecture: A Hardware/Software Approach.* San Francisco: Morgan Kaufman.

Csanady, C., and P. Wyckoff. 1998. Bobnet: High-Performance Message Passing for Commodity Networking Components. Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networks

Flynn, M. J., 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computing* C-21 (September):948-960

Gillett R., 1996. Memory Channel Network for PCI. *IEEE Micro* 17(1):19-25

Gropp, W. et al. 1996. A high-performance, portable implementation of the {MPI} message passing interface standard. *Parallel Computing* 22(6):789-828

Gropp, W. and E. Lusk. 1996. User's Guide for MPICH, a Portable Implementation of MPI. Mathematics and Computer Science Division, Argonne National Laboratory. ANL-96/6.

Halstead, R. H., 1978. Multiprocessor Implementations of Message Passing Systems. Cambridge Laboratory for Computer Science at MIT.

Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach.* 2nd ed. San Francisco: Morgan Kaufmann.

Hoening, B. 1999. Building Multiprocessor DSP Systems. *Electronic Systems and Technology Design*

Kronenberg, N. P., H. Levy, and W. D. Strecker. 1986. Vax Clusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems* 4(2):130-146

MPI Forum. 1995. *MPI: A Message Passing-Interface Standard.* http://www.mcs.anl.gov/mpi/

Naik, V., 1993 *Multiprocessor tradeoffs in computation and communication*. Boston: Klauer Academic Publishers

Passmore, L. D. 1978. *A multiprocessor Computer Architecture for Signal Processing*. MIT.

Pfister, G. F., et al. 1985. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proc. Int'l Conference on Parallel Processing* (August):264-771

Seitz, C. L. 1985. The Cosmic Cube. *Communications of the ACM* 28(1):22-33

Texas Instruments. 1995. *TMS320C80 (MVP) Parallel Processor User's Guide*.

Walker, C. P., Hardware for Transputing without Transputers. *Parallel Processing Developments*(19):1-10