ADAPTIVELY NETWORKED EXTENDABLE MULTIPROCESSOR ARCHITECTURE

A Design Project Report
Presented to the Engineering Division of the Graduate School
Cornell University
in Partial Fulfillment of the Requirements of the Degree of
Master of Engineering (Electrical)

by
Idan Beck
Project Advisor: Bruce Land
Degree Date: January 2008

Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

Project Title:
Adaptively Network Extendible Multiprocessor Architecture

Author:
Idan Beck

Abstract:
This project consisted of the design of a fully integrated extendible multiprocessor architecture and system. This included the design of the compiler, assembler, linker, boot loader, and handshake interfacing protocols for peripheral modules which all may or may not lie on differing clock domains. The architecture is a single simple cycle RISC CPU with independent data and instruction memories such that it is as close to pure RISC as possible. The network layer works under the CPU transparently in a self-timed fashion utilizing a handshake. Network communication is solely based on instruction dispatching where each CPU is essentially capable of writing to another CPU's instruction memory. The network is also adaptive utilizing a unique register-thru design and message passing algorithm which allows each node on the network to behave either as a packet switched network or an evolved circuit switched network and a broadcast can be used to reset the network.

Report Approved by
Project Advisor:_____ Date:_____

Executive Summary

This project aims to try and provide a new approach to multiprocessor systems, which are known to be hard to design as well as program. One of the biggest problems with multiprocessor systems is the "memory wall" where multiple processors must share a certain amount of bandwidth to memory. My project provides a viable alternative to distributed shared memory models with a more cluster oriented design where each node in a multiprocessor network owns its own data and instruction memory.

Additionally a different approach was provided to the network through the design of a self optimizing asynchronous network, which can operate transparently below a synchronous CPU through a handshake. This same handshake protocol is also implemented in the system on a whole where it would be possible to interface any of the different modules designed for this project through this interface and it would work correctly. The network additionally implements a message passing algorithm, which sends out a message in the same fashion as a wave "crest" and in this way the network optimizations actually will burn in the paths over time to optimize commonly used path ways.

The design of a complete system on chip integration chain was accomplished. A new CPU architecture was designed, which implements all of the above as well as being as RISC as possible. The compiler, assembler/linker and bootloader software and hardware was designed for the CPU as well. Although minor bugs still exist in the system it was demonstrated to operate for significantly complex programs and test the functionality of the whole system including the CPU, IO, and software supporting systems.

A large portion of this project was oriented around making programming of multiprocessor environments easier by utilizing this new architecture and system topology. This involved the design of new programming conventions and methodologies that had to be supported both on the hardware and software side.

The design was then implemented in simulation first to validate the functionality and claims but also it was implemented onto an Altera CycloneII FPGA on a Terasic DE2 board. This configuration allowed access to the crucial hardware needed to validate the design, such as a VGA monitor, PS2 keyboard, and RS232 serial connection. The design also included modules that would have enabled the use of the LCD display and audio codec.

Contents

1 Introduction

1.1 Project Overview and Specifications

This project consisted of the design of several components which, when put together, constitute a complete computer system. The approach used was modular in such a way that it could be possible to replicate these components to design a scalable system. Each component employs a uniform set of protocols which then allows the components to be connected in any assortment and allow the system to be scalable and parallel.

This was done through a new RISC CPU architecture complete with the compiler and assembler-linker required to run decently complex programs on the CPU. Also a boot loader was designed as to allow the CPU to be programmed and used in a variety of situations. However, security was not a main focus since this was beyond the scope of my project.

The CPU design is a new network driven architecture (NDMA) which allows each CPU to network with others in such a way that allows scalable multiprocessing. This would allow systems to be designed with parallelism in mind. The architecture was designed in such a way that the network component could then be used to drive the CPU itself. Many designs use a network by passing messages which are then interpreted by the CPU. This has the advantage of seamlessly passing data quickly between different processors. However, send a packet of data to a CPU which doesn't know what to do with it and it will simply look like noise. The design implemented in this project does not send data but rather sends instructions where all data must be encoded in pieces of code which the receiving CPU will then execute allowing increased flexibility in how parallel programs are designed.

Several other systems had to be designed for it to be possible for the observation that the project is actually working. This included the integration chain for the NDMA processor which I called the NDMA Suite. The Suite consists of the NDMA compiler back end for LCC, NDMA Merge which allows multiple assembly files to be integrated, the NDMA Assembler-Linker, and the NDMA Bootloader. Also some APIs were written for the GPU, PS2 Input Buffer, and network layer. Throughout the system interfaces must be upheld for asynchronous operation since many of the systems lived in different clock domains. In fact, the network layer was designed in such a way that it was self timed while the data that it received would be put through a controller which would clock the data so that it could be correctly passed

to the CPU. This was possible through the use of the same handshaking protocol utilized system wide.

It can be noticed in the design that over time the design tended to evolve. This resulted in some redundancy in the design but it was deemed negligible and if needed this could be very quickly cleaned up. Cleaning up this redundancy would simplify the hardware design and make the design quicker to implement on an FPGA or increased simplicity of this design ever went to layout.

Since most of this design project was original there were not many alternatives to the design choices I made and their implementation details. On the other hand it would have been best if I could have found an open CPU core that provided the integration chain as well (such as bootloader, assembler etc). This was not easy to find and I justified my original design by the fact that I am in fact creating open source intellectual property through this project which is potentially useful for others.

With the above said I did not go ahead and write the compiler from scratch but rather used LCC which is an open source retargetable ANSI C compiler. Since this was open source the decision was valid. The other option would have been to write my own compiler and this combined with all of the other work I had to do with not a realistic goal even in the time that I had to work on this project.

Overall looking back at the design decisions I think that considering the scope and size of this project that it was very well planned out and that all of the requirements and specifications were met that I originally set out to accomplish. The hardware and simulations very much agree while the only thing that was not achieved in hardware was not originally speculated in the original design and only realized in simulation. This is explained more in the following report and specifically in the last part of the Results and Analysis.

1.2 Rationale and Motivations

The original idea behind this project originated in the realization that current computer models today do not take much advantage of parallelism yet being parallel systems. The truth is that a common computer system consists of a main CPU but many of the other components and ASICs in a computer system also have a certain amount of computing power that is not utilized during the normal average operation of the machine.

One of the best examples of this is of a theoretical system that consists

of a video card, a sound card, and a central CPU. If the demands of the system are such that the sound card or the video card are only being used to their full potential half of the time then we can clearly see that there could be a similar system designed that could somehow parallelize the resources such that when the system does not require the video card for its designated purpose it could then the resource can be shifted and used for a different purpose. For example, using this scheme, it would be possible to design audio cards that expose their DSP capabilities and video cards that expose their SIMD capabilities through some kind of interface which allows the system to reconfigure itself on the fly.

With this kind of system in mind I designed a CPU Architecture that was network driven. This CPU is then capable of either executing it's instruction memory as is initialized by a standard boot loader or it can also dispatch instructions to other CPUs and likewise receive instructions from other CPUs. The idea behind this is that a CPU can then be "told" what to do and then reply with actions as well. These capabilities were also built into the assembler such that through standard coding practice these capabilities can be accessed although most of the APIs must be written in assembly since the data sent through the pipes should be instruction based since the network layer and the CPU are completely transparent to one another and the only way communication can occur is through full 32 bit instructions.

2 Hardware

2.1 NDMA CPU Architecture

The project did not require a completely new architecture however it was noticed that the amount of changes required to a currently existing would warrant the development of a new CPU architecture and the design of the CPU from scratch. Also it was exciting to be able to decide on the design and to be able to integrate the network driven operation all the way into the design instead of a surface based approach. Another advantage of this was to be able to call the CPU original intellectual property. While the initial design of the CPU was motivated by the MIPS architecture only the OP code pairings are really preserved. There is no branch or load delay slot but the register designations are also preserved since it simplified the development of the compiler. These design decisions are also supported by the assembler

although these changes are minimal from the assembler's perspective.

2.1.1 First Pass

The NDMA CPU Architecture is a 32 bit MIPS motivated single cycle CPU with individual data and instruction memories. The instruction memory is a dual port memory to allow for reading and writing on the same cycle and the data memory is single port. Also no cache is designed.

The most noticeable thing about this design is that it is a single cycle CPU. The idea behind this was to design a CPU that was as close to RISC as possible. For this reason there exists the individual data and instruction memories. The point of the architecture was not to make it fast but rather to make it RISC and easy to manipulate as to demonstrate the network side of things. However it was seen that this design, from an embedded standpoint, was rather sufficiently well performing.

2.1.2 Memory Hierarchy

There were a few issues that arose regarding the memory hierarchy. First of all it can be noticed that there is no cache. The scope of this project did not require the design of a full cache as the design of one is non-trivial and would have required a large amount of debugging and development. Regardless of this fact since the CPU was going to be implemented on an FPGA where it was likely that the memory would be able to be clocked faster than the CPU itself it was unnecessary to develop a cache.

There was a large issue regarding the memory as well regarding the block sizes of the memory. The CPU designed was a 32 bit architecture and in turn required 32 bit memory line sizes. The way that these were implemented was through Altera M4K blocks of 32 bit width. This meant that it was impossible to allow unaligned memory access and required a good amount of compiler/assembler cooperation which is explained below. The reason that 8 bit width was not used, which would have enabled unaligned memory access, was because only 8 bits per cycle could have been accessed. It would have been possible to design a more complex memory module that would have consisted of 4 8 bit memory banks which would be striped but it was deemed enough to have 32 bit blocks and revisit and fix the inefficiencies at a later time.

A large difference in the memory hierarchy of this CPU and a MIPS CPU

for example is that there are independent data and instruction memories. The reason for this lies in a concept of resource redistribution over a network and system on chip design methodology. Also this was an idea motivated by finding a different approach to distributed shared memory. Each CPU has it's own instruction memory that can be manipulated by external CPUs as well as it's own data memory that only it can manipulate. Otherwise external CPUs may destroy data unknowingly by overwriting it. In this way each CPU has at least one resource that it owns exclusively.

2.1.3 Datapath

Below is shown a simplified datapath diagram of the NDMA architecture. Figure 1 is more for reference than an accurate design diagram (since it is missing many of the signal names) but it shows how this is a general single cycle CPU datapath. This looks very much like a normal single cycle CPU datapath except for the Input / Output module and the network driven side. It is not completely shown in this diagram how the network side is integrated into the instruction memory using an network instruction memory controller and how this is also integrated into the register file. These design details are described in the network portion of this section.

This is a single cycle CPU for the reasons mentioned above. At any moment the network side can interrupt the operation of the CPU and instead of focusing this project on how to properly manage the pipeline when dealing with such collisions I thought it would be better to focus on the systems development (compiler, assembler, bootloader) and on the network driven operation. It was also decided that the implementation of pipelining could always be done later and would be easier to integrate once the network driven architecture was more set in stone.

2.1.4 Instruction Set

There is nothing revolutionary or new about the basis of the core design. The instruction set and core architecture was at first based on the MIPS instruction set. Not to mention that most of the core instructions in a CPU look very similar such as the ADD instruction. For this reason I used the MIPS base set as a launching point since I had some supporting examples that were a nice reference. However, there are many outstanding differences between the NDMA architecture and the MIPS architecture which qualify it

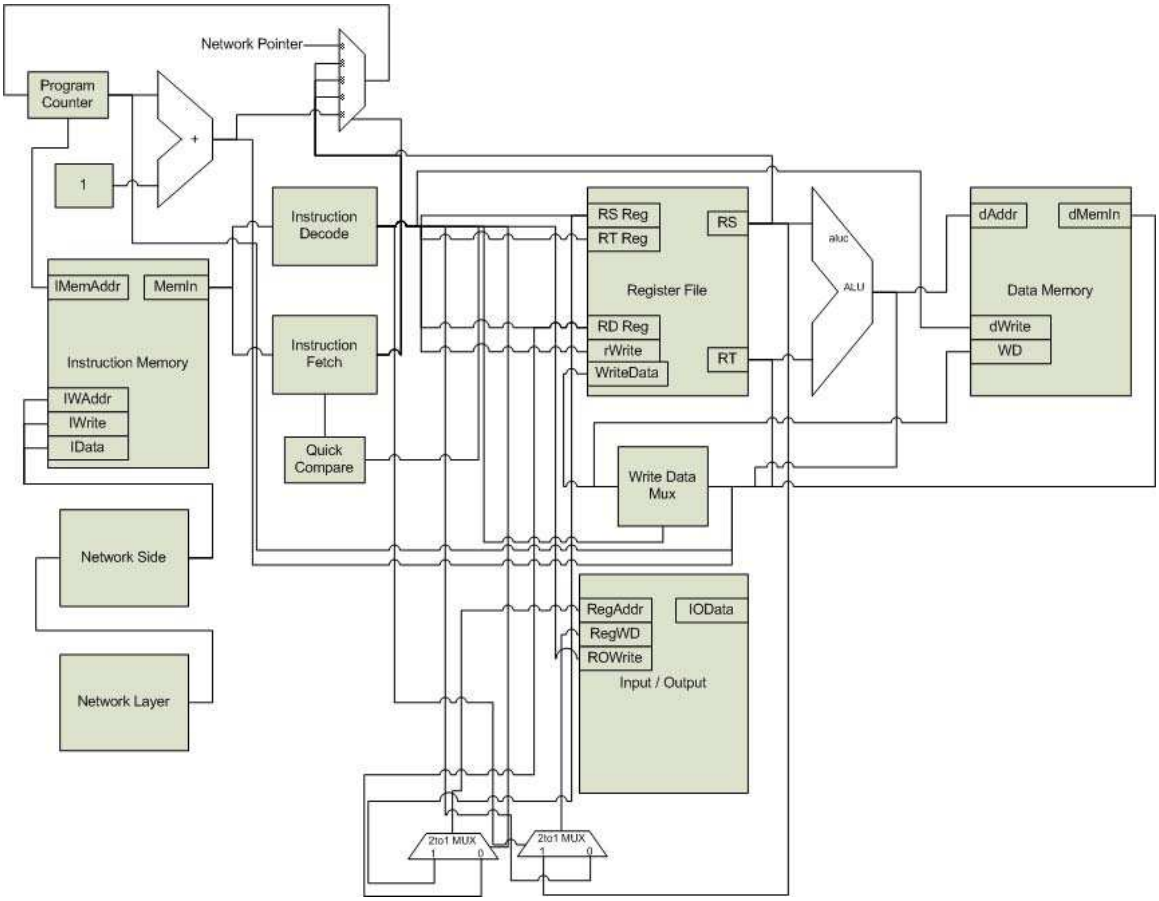

Figure 1: Network Driven Microprocessor Architecture Datapath

as a different architecture.

First of all it can be noticed that there are no branch delay slots. The reason behind eliminating branch delay slots was the fact that network based CPUs would need to strictly control the operation of the dispatched code on a different CPU and that collisions between branch delay slot instructions and network driven instructions may become an issue. It is likely possible to reintroduce the branch delay slot but the compiler/assembler would need to be smarter in this regard and since most of the network driven code is being hand written in assembly at this time it seemed better to get rid of the branch delay slot completely.

Another similar difference is the lack of load delay slots. There is some

redundant hardware currently that forces a NOP after a load but removing this does not affect the operation what so ever. The lack of load delay slots is motivated by a similar reason to the lack of branch delay slots. However it is also a result of independent instruction and data memories. The motivation of the independent data / instruction memories was an idea motivated by resource distribution on a network. This is explained above in the memory hierarchy section. This actually ends up being a positive effect of the memory hierarchy which simplifies the compiler design.

At first the instruction set was very similar to the core MIPS instruction set but over time new instructions were introduced and old ones taken out to make way for network oriented instructions. I wanted to make sure that although network and CPU instructions should never be decoded in the same space they should still not overlap as much as possible. This was more for debugging purposes but since there are so few exclusively network driven instructions there was not much issue with this.

Of the new instructions introduced into the ISA are the input / output instructions: IN, OUT, and OUTI. Also the CPU side network instructions: SID, BCST, SMSG, BCSTR, SMSGR. Also there are the network side network instructions: SNIP, JALNET, and NDJR. There exist two network driven instructions that were never implemented completely as well: NACK and RMSG. These were planned for originally but deemed redundant especially since their operation proved somewhat complex as well. The multiple and divide instructions were deprecated since they were implemented brute force with Altera megafunctions for a brief time. They were tested and working fine but were not too great for timing. They were taken out with the plans to implement a dedicated multiply and divide unit when time was available.

2.2 Network Layer

The Figure 2 below shows the basic overview of the Network Layer structure.

The idea behind the network layer is that it should function transparently to the CPU. This means that if the network layer is passing a message it will not affect the functionality of the CPU. The network layer is not clocked except for the send portion. It will correctly react to incoming messages regardless of the time at which those messages are received. The send portion, since it is triggered by a send message or broadcast message instruction from the CPU, must be clocked however. This is not clocked by an actual clock but rather self-timed due to the send flag, dxFlag, which is clocked by the

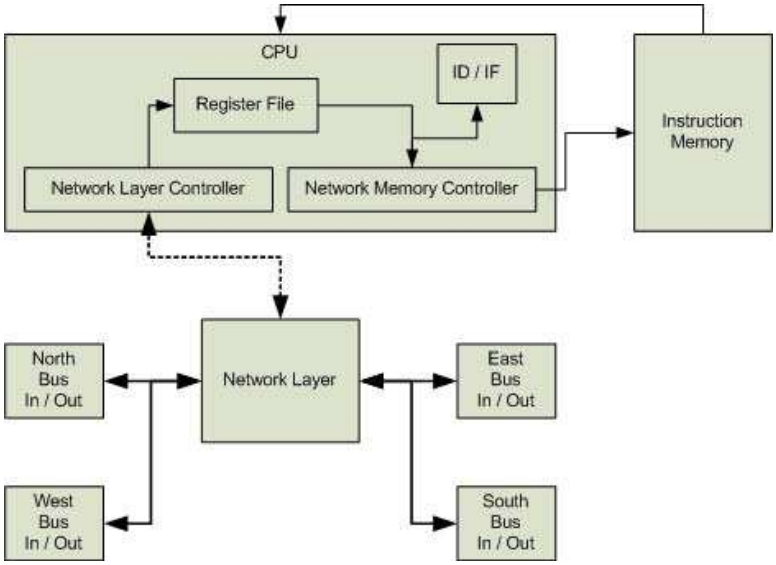

Figure 2: Network Layer Architecture Overview

network controller module.

The network layer is driven by a network controller which decodes the CPU side network instructions such as set ID, and send message. For the case of a send message instruction the network controller module will then decode the instruction on the negative edge of the clock and set the dxFlag to high. The dxFlag then is used by the network layer to set the output busses to the correct values. This is done on the positive edge of the clock to allow all of the send values to be correctly set before the message is set out on the network layer busses. The dxFlag is pipelined into a register to avoid from sending the same message twice or in the case that the same message is required to be sent twice and ensure that there will be a NULL message on the bus between messages.

2.2.1 Message Passing

The messages are sent out on a 32 bit bus and each message contains the 7 bit destination CPU's ID, 8 bits of the message data, the 7 bit origination CPU's ID, a 4 bit message age, a 2 bit origination direction, and a 2 bit last taken direction. Each one of these values is used to help navigate the message to its correct destination. Figure 3 shows the composition of a message.

Figure 3: Message Composition

The way that the network layer propagates a messages gets its motivation from wave propagation. This was demonstrated with a C++ based simulation before implementation to prove that this algorithm will work correctly since it is much harder to debug and prove in Verilog. The overview of the algorithm is shown in Figure 4.

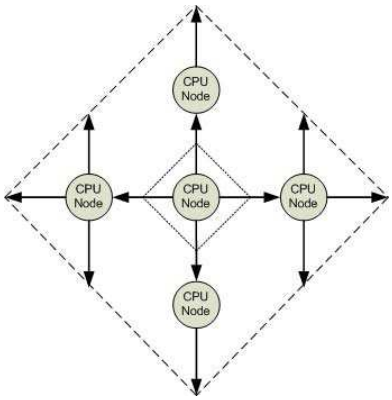

Figure 4: Message Passing Algorithm

The idea behind this algorithm is to always maintain a wave "crest" of the message and try to keep this crest as coherent as possible. A big problem is when multiple message crests collide and this must be dealt with by using a message buffer. This was not yet implemented in the hardware design but it was simulated in the simulation program.

The algorithm works by the original message source sending out the message in all directions setting the appropriate data values to the different fields of the message. In the case that the message is a broadcast message (which should be received by everyone) then the message's destination ID field is replaced with 0xFF which when read by the Network Layer will be seen as a broadcast message. Note that the IDs 0xFF and 0x00 are reserved for special cases such as broadcast and null IDs. This is important for the correct

operation of the network layer and also disallows an ID to be changed once it is set.

When a message is received by a node first the destination ID is compared to the ID of that node. This is why it is extremely important to provide a ROM image initially in a CPU node's memory that will set the ID to the correct ID. The conventions of how to use this system regarding writing actual programs is described in the Results and Analysis section. After checking the destination ID the network layer then decodes the origination direction and deals with the message in different ways based on the bus that the message was incoming on.

The general idea is that "horizontally" traveling messages will produce new messages on in the same direction as they were originally traveling as well as the directions perpendicular to their originating direction. However, messages traveling in "vertical" directions will simply continue to propagate in the vertical directions. This can be compared to dragging one's finger across the face of water.

This was shown to correctly work in the Visual Nodes simulation program. This program is further discussed in Appendix A. Figure 5 shows two blue nodes sitting on an arbitrary network of other CPU nodes. These nodes' CPUs have received instructions to send messages to two other nodes on the network. Notice that these nodes have no idea where the destination node is in the network. This constitutes a packet switched network where the message propagated contains very simple routing information.

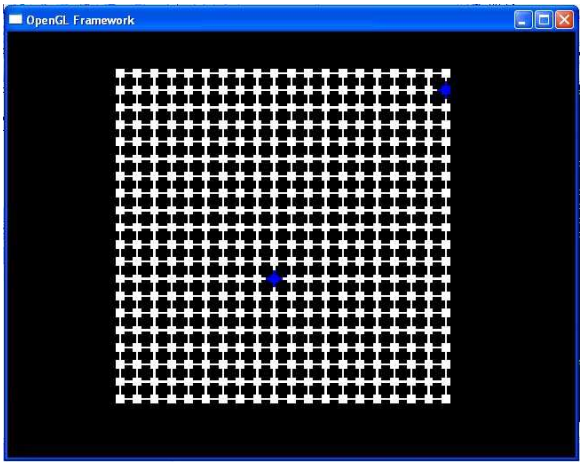

Figure 5: CPU Nodal Network at Initial State

Next we see how the algorithm described before effectively creates two message crests. Each message is identified by it's own individual color as to differentiate between the two and show that no lossiness of message occurs since each node has a message queue in the network layer. Simulation shows this but this was not implemented in hardware. Figure 6 shows the simulation of this scenario. Notice that when the messages reach the boundary of the network they are lost. This is OK to do since we know that the message must have arrived if all the connections were made correctly and the messages passed according to the algorithm.

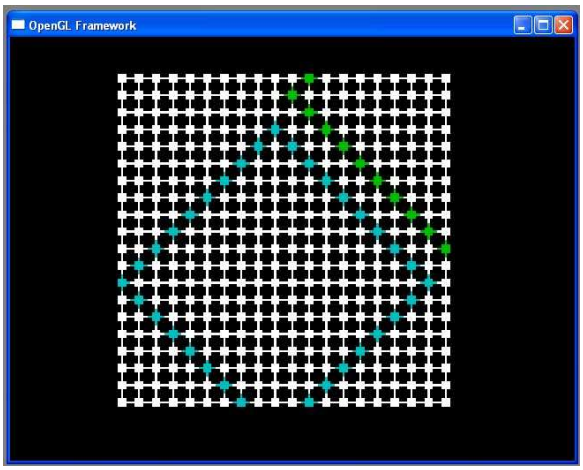

Figure 6: Message crests formed after two nodes send out a message to an unknown ID on the network

Eventually the messages are received by the destination CPUs as shown by the neon green node in Figure 7. Notice that we do not need to worry about the rest of the messages on the network since we know that only one CPU on the network has the unique ID being sent to. So once the message arrives we no longer need to worry about the coherency of the wave. Notice there is a hole "horizontally" behind the receiver node. This is because at this node the message is not propagated.

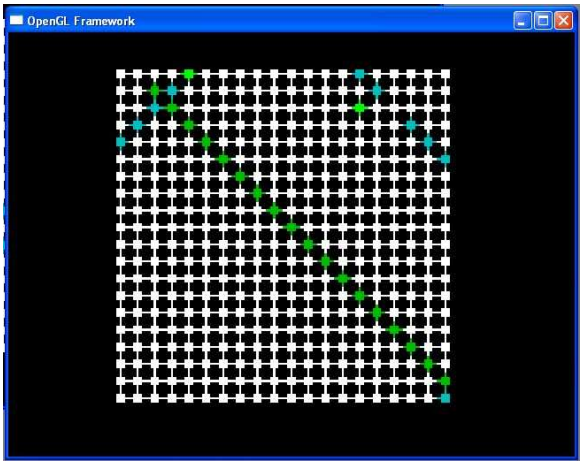

Figure 7: The messages are received by the correct CPU nodes on the network

In the simulation the node will initially pend for a few cycles. Then to prove that the data was received correctly, as encoded by color, it will send back the same message to the original sender of the message. This is shown in Figure 8. In the real implementation of this the way that CPUs would communicate is much different as will be explained in further detail. Notice that the origination direction of the message is taken into account went the reply is sent. This was never done for the hardware as well but could be in the future with some expanded instructions.

The only current open issue is that in the real implementation the message queue has not been implemented so that collisions are a real threat to the system. Collisions can only truly occur when multiple nodes are communicating on the network at the same time. It was decided that to prove the validity of the system the message queue would be something extra to strive for. It would be relatively easy to implement especially considering that the functional model has already been outlined in the Visual Nodes simulation.

2.2.2 Adaptive Network

The network layer explained up until now has been a packet switching network. However, it is obvious that this can be rather inefficient if two nodes on the network tend to communicate much more often than other nodes on the network. It would be much more efficient if the network could somehow evolve to treat commonly used paths as circuit switched networks rather than

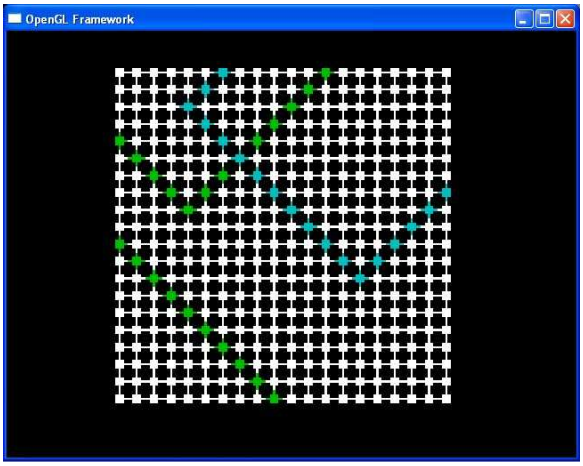

Figure 8: Message replied to original sender with same data

packet switched networks so that routing occurs automatically after a certain fitness threshold.

Exactly this was done in the network layer through the implementation of what I call a register-thru design. Essentially each directional bus also owns a certain threshold counter which is initialized to zero. Each time that a network message passes through the node this counter is incremented by one while each time a message is received this counter is decremented by one. If the counter reaches a certain threshold value it is no longer incremented but at this point the node begins to pass messages along this bus without holding on to them. However, the node does catch a glimpse of the values so it is possible for the CPU to still receive messages on this connection while the message will still pass through the node. This has no real consequences, however, since nodes need to have unique IDs. When a message is passed with matching destination ID the cut off node will "awaken" but needs to continue to get multiple messages as to not shut off once it receives a different message of mismatched ID. Also a broadcast message will reset this counter to zero.

The way that the register-thru design is implemented is by multiplexing a wire and register design. When the counter is below the threshold value the bus will first pass the message by setting the output register to the input value. This requires a certain delay and is less optimal than a pure wire. When the counter reaches a certain value the output is multiplexed to the

input and hardwired so that the input is seen on the output at the same time neglecting the actual delay of the wire. Using this register-thru design we can extremely optimize a network which may have an arbitrary node delay. The register-thru design is shown in Figure 9.

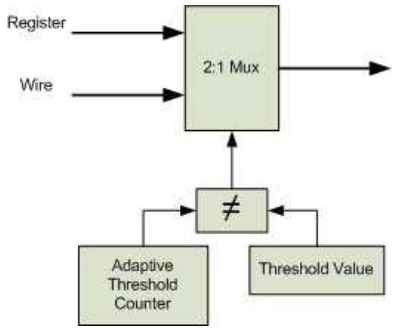

Figure 9: Register-Thru Design Implementation

To demonstrate some of the functionality of this register thru design the following print out of a command line simulation is shown in Figure 10. A more conclusive result is shown in the Results section but this is a quick demonstration in simulation of the design in action. Notice that messages are being passed back and forth with a certain delay. Once the threshold count is arrived at for a few nodes the connection is cut and adaptively converted to circuit based connections. The delay is then reduced to a delay of two units where the original delay was 5 delay units. Note, however, that this is a nodal network of only 9 nodes. The potential improvement can be rather scalable.

2.2.3 Network Driven Operation CPU Side

There are two sides to the network driven operation of the CPU. The CPU side which is dictated by instructions coming out of the instruction memory and Network side which is dictated by the network literally driving the CPU through instructions dispatched to it through the network. This section deals with the CPU side of sending and broadcasting messages.

You will notice that in the design there are remnants of other instructions and other such issues that need to be re factored. Due to time constraints these are being ignored since they are not affecting anything but they will be described and considered here since they may in the future be implemented.

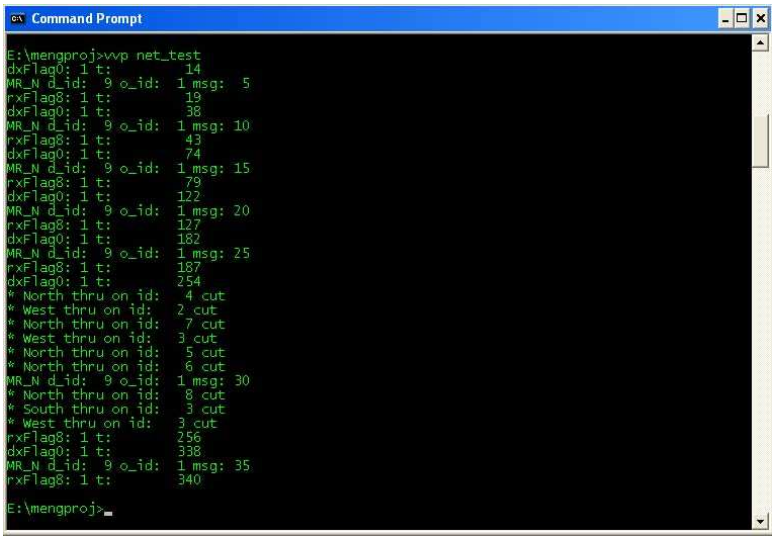

Figure 10: Adaptive network command line simulation

The receive message instruction RMSG was originally going to be used to allow a CPU to pend on a receive. However, with the way that the project has shifted since the original introduction of this instruction this never had to be implemented. The only place this instruction is seen is in the ndmaref.h file which defines the OP-Codes for the CPU. Another instruction that falls into the same category is the NACK instructions (although Network Side). In the end a different approach was taken to the NACK functionality instead of a dedicated instruction which would have been somewhat inefficient.

The only CPU side instructions that occur consist of send message (SMSG), broadcast (BCST) and their register based counter parts. The usage of these instructions is as follows:

```
smsg $destination_ID, message
smsgr $destination, $message, byte_offset
bcst message
bcstr $message, byte_offset
```

The normal versions of SMSG uses the value inside of the register $destination as the destination ID of the message being sent. The message is then sent as described at the beginning of this section. The normal version works the same way, however it requires no ID since the ID sent in the message is

0xFF which notifies the network layer that this is a broadcast message. A broadcast message will also "wake" up all nodes in it's path.

After the BCST and SMSG instructions were implemented and some of the system had been implemented it was obvious that it would be needed to have the same instructions except register based. This would allow the sending of the whole contents of a register. Since the message sent can only be 8 bits 4 messages must be used to send a full 32 bit package. Doing this by hand through the use of hexadecimal digits or decimal digits would be extremely cumbersome. Instead register versions of the instructions were designed which allow you to send an 8 bit piece of a 32 bit register through the use of the byte offset. The byte offset designates 3 as the most significant byte. In this was the following series of instructions can be used to broadcast an instruction to another CPU:

```
ori $a0, $0, 0
lui $a0, Instruction First Half
ori $a0, $a0, Instruction Second Half
bcstr $a0, 3
bcstr $a0, 2
bcstr $a0, 1
bcstr $a0, 0
```

Using this convention it was possible allow a CPU to initialize a register with an instruction that it would like to dispatch to another CPU and then utilize the bcstr / smsgr instructions to piece wise send the instruction. Note that the first three instructions above would be commonly repeated often and complex to program by hand. These were integrated into a pseudo instruction called SRI or set register instruction. This pseudo instruction is further explained in the assembler section.

2.2.4 Network Driven Operation Network Side

Once the instruction reaches the point of being in the pipes it is considered a network side instruction. The basic principles of the system network driven operation is shown in Figure 11. What is being shown in the diagram is that CPU 1's instruction memory is originally initialized through the use of the NDMA Bootloader. This memory will then drive CPU 1 and allow CPU 1 to dispatch instructions to other CPUs. Only a two CPU system is shown

here for simplicity. This is effectively enabling CPU 1 to boot load any of the other CPUs on the system. The way that this is done will be explained in this section. Once a different CPU on the network is boot loaded and running it can then return the favor and replace or re-boot load CPU 1 or any other CPU on the network. If wanted this can create an infinite chain of CPUs not only passing data messages but also passing the code required to parse those messages so that you can design programs with dynamic parallelism.

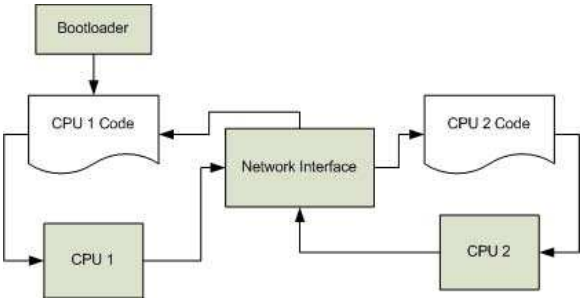

Figure 11: Network side system operation and communication

For this to be possible a few different specialized instructions need to exist which only manipulate the network layer. These instructions consist of the following:

```
snip pointer_value
jalnet
ndjr $target_register
```

The way that the network layer receives information has two parts. When a message is received by the network layer it is then saved to a specialized register in the register file indexed by a network message counter. This counter is incremented each time a message is received and saves that 8 bit message into the register indexed by the counter. Once 4 messages have been received then that constitutes a full 32 bits received.

When the instruction received from the network is valid the register file sends the instruction to the network memory controller which will then look at the instruction and either write it to a specific location in instruction memory or decode it if it is a special instruction. The way this works is that the network memory controller contains a pointer into memory which defines the network memory ”space” in instruction memory. This pointer

can be changed with one of the above SNIP instruction which stands for Set Network Instruction Pointer. It should have been SNIMP since it's an Instruction Memory pointer but SNIP tends to have a ring about it's functionality. When a SNIP instruction is decoded the CPU is not driven with any specific functionality but rather the instruction memory pointer is set to the value passed by pointer_value. Notice that when a special instruction is received by the network instruction memory controller that it is not written to memory.

There also needs to be a way to have the target CPU start executing the instructions in the network portion of the instruction memory as well as a way to force them into their previous operation. The instructions JALNET and NDJR do exactly that. JALNET is a network driven Jump and Link. This works the same way a normal JAL would work except the location to jump to is taken as the instruction memory pointer. When a JALNET is received it will be decoded by the network instruction memory but also passed to the Instruction Fetch and Instruction decode modules in the CPU data path. The CPU will then be driven by this instruction and jump to the correct location. A NDJR instruction works the same way where all of the registers initialized by the ID and IF stages in the CPU. The NDJR instruction stands for Network Driven Jump Register and works exactly the same as a normal Jump Register instruction except that it has precedence and is network driven.

One issue regarding network driven instructions was the question of what of the current instruction? The decision was made to give precedence to the network driven instructions, or specifically JALNET which should be the only one that has any problems, since with JALNET the current pointer is saved into a register and while a normal JAL will save PC + 1 to the register JALNET was changed to save the pure PC to the register because when a JALNET occurs we are consistently blocking an instruction that must be preserved.

2.2.5 Network Programming Conventions

The meat of this topic will be discussed more in the assembler section since most of the conventions applied will be done on the software side. In fact the design architecture is good to go as is, however it is rather cumbersome to program. The way that this is done is through specialized assembler side instructions which will dispatch instructions through an API. The idea

behind this is that a node cannot actually dispatch data to another processor but rather must send an instruction with that data encoded within it.

Although this may convolute the process of sending data from one processor to the other it also ensures that the interfaces are standard and that data is passed in a very interface oriented way. This allows processors to literally tell each other what to do rather than simply pass data back and forth. If a node is smart about the code that it dispatches to another processor than it is quite easy to design a system with rather complex parallelism.

Since for every instruction sent at least 4 instructions must be executed the instructions sent must be parallelized functions as is. So instead we could design a system which dispatches out a few different pieces of code to different processors which loop. These loop until completion at which point they send back instructions to the dispatcher CPU with their results where the dispatcher CPU was the one to tell them to do that. Until this occurs the dispatcher CPU is sitting on a loop waiting for the results from the dispatched process. Once this occurs the dispatcher executes the network memory, returns to a point outside of the loop (or sets a branch condition false) and then retrieves the result out of the point in which it was stored.

This is a simple scenario but imagine multiple CPUs processing different programs at the same time all the while the dispatcher CPU is sitting on a useful loop of some sort. As long as the code dispatched will run for at least 4 iterations on average we can see that this parallelism will benefit us. Especially as you increase nodes in the system since as soon as you create code that can create more code each node can then dispatch to three other nodes (not including the node it received the code from in the first place). Although my project was more of an infrastructure project this architecture seems much more realistic for parallel programming than the current methods which currently use distributed memory sharing with the very serious memory bandwidth bottleneck.

Another very important possible use of this architecture is to enable resource sharing and system reconfiguration. It would be possible to have one processor connected to one resource and have all processors relay their information to a centralized processor which would then organize all of the resources. However, when the resources are not needed it would be possible to better utilize the CPUs for other uses such as computation. This dynamic parallelism could be very useful in consumer type computers where resource need is very diverse rather than in computationally intensive computers where data is the main thing that matters.

3 Software

Apart from the hardware design of this project a variety of software had to be written. This section describes these software systems.

3.1 LCC NDMACompiler Back end

The NDMA architecture needed to have a compiler so that decently complex programs could be written both to test the system as well as make the system usable. The compiler used was LCC which is a retargetable ANSI compatible C compiler. The back-end was written such that the assembly language the compiler generated was specific to the NDMA architecture and would interface cleanly with the assembler.

A big problem that I encountered while using LCC was build related due to my Windows XP operating system. I could not quite get LCC to build correctly and in turn compile my code into NDMA specific assembly. Instead I discovered RCC which is the program that actually generates the assembly. In the case of a standard C error RCC will output a message however it will not stop generating assembly. This, for my purposes, was enough and since LCC utilizes RCC I am sure that on a system which can build LCC correctly (e.g. Linux) that LCC would work fine for my compiler back end.

The way that LCC's back end works is through an interpreted definitions file which is interpreted by a program named lburg. The file is a long listing of function calls and instruction definitions. A rather large amount of the file is standard for all code targets so I started off using a specification for the MIPS R3000 that is provided along with the source code for LCC. After this I went through the file changing all of the different instructions so that they will be compliant with my NDMA assembler as well as compiler conventions. Since I did not write LCC it was occasionally easier to keep certain conventions and then deal with them in the assembler.

3.1.1 Pseudo Instructions

There were a few pseudo instructions that were used in the compiler which I could not completely eliminate from the back-end and had to implement in the assembler. Most significantly was the pseudo instruction LA which stands for Load Address. In most cases this instruction would be used as

```
la $destination, offset($source)
```

The use of this instruction would be used to load in the effective address of the offset + $source into the $destination register. It would have then been very easy to simply replace every LA instruction with

```
addi $destination, $source, offset
```

This was not possible since LCC uses the LA instruction in a variety of ways all the while only having one specification for it. The LA specification in the lburg file is as such:

```
reg: addr  "la $%c, %0\n"  1
```

This indicates that the field filled by %0 can be anything as provided by LCC. This is in fact true and was found to be of any of the following:

```
offset($source)
offset
Label
```

The label in the above snippet means that not only is the LA instruction used to initialize registers with effective addresses as to be used for the purpose of initializing them with numerical constants but it was also used to load registers with values designating pointers to instruction memory. This became especially prevalent when dealing with resource strings in programs where LA was used as a pointer to instruction memory so that strings could be read out of the program's code.

NDMA has an independent instruction and data memory. So changing the lburg file for LA was an impossibility. Instead the assembler was designed in a much more clever way as to perceive which form of LA it was receiving and as such decode and assemble it so that it would work correctly. This became somewhat complex for string resources since it required a collaboration on both parts of the assembler and compiler as will be described below.

3.1.2 String Resources

As mentioned before the NDMA has a separate instruction and data memory. This allows for network driven operations as well as maintain more of a RISC oriented operation. However it has rather big issues regarding how LCC would process string resources in a program. Traditionally when a programmer defines a string in a program the string is saved at the top of the assembly code using .byte directives:

```
.SetStack 255
.Boot main
.rdata
STR_LABEL1:
.byte 72
.byte 45
.byte 44
.text
.text
main:
... code ...
la $3, STR_LABEL1
lb $4, 0($3)
... code ...
```

Then when the string was encountered in the code it would use the STR_LABEL1 to designate a point in memory and load the effective address into a register which would then be used to load the memory from. This would not work using the hardware architecture. Also a big point to note is that the memory used for the CPU had 32 bit wide blocks as mentioned above. It was then important to change the way data was loaded and stored into memory when it's size was less than 3 words.

To deal with these issues the code that would load and store data in LCC was altered to the following:

```
static void blkfetch(int size, int off, int reg, int tmp) {
        int i;
        assert(size == 1 || size == 2 || size == 4);
        if (size == 1)
                print("lbu $%d,%d($%d)\n",  tmp, off, reg);
        else if (salign >= size && size == 2)
                print("lhu $%d,%d($%d)\n",  tmp, off, reg);
        else if (salign >= size)
                print("lw $%d,%d($%d)\n",   tmp, off, reg);
        else if (size == 2)
        {
                // LH for .Byte here
                print("ori $%d, $0, 0\n", tmp);
```

```
                // Second byte
                print("lw $1, %d($%d) \n", off + 1, reg);
                print("sll $1, $1, 8\n");
                print("or $%d, $1, $%d\n", tmp, tmp);
                // First byte
                print("lw $1, %d($%d) \n", off, reg);
                print("or $%d, $1, $%d\n", tmp, tmp);

                //print("lhu $%d, %d($%d)\n", tmp, off, reg);
        }
        else
        {
                // LW for .Byte here
                print("ori $%d, $0, 0\n", tmp);
                for (i = 0; i < 4; i += 1)
                {
                    print("lw $1, %d($%d) \n", off + (3 - i), reg);
                    print("sll $1, $1, %d\n", (3 - i)*8);
                    print("or $%d, $1, $%d\n", tmp, tmp);
                }
        }
}

static void blkstore(int size, int off, int reg, int tmp) {
        int i = 0;
        if (size == 1)
                print("sb $%d,%d($%d)\n",  tmp, off, reg);
        else if (dalign >= size && size == 2)
                print("sh $%d,%d($%d)\n",  tmp, off, reg);
        else if (dalign >= size)
                print("sw $%d,%d($%d)\n",  tmp, off, reg);
        else if (size == 2)
        {
                // First
                print("sw $%d, %d($%d)\n", tmp, off, reg);
                print("srl $%d, $%d, 8\n", tmp, tmp);
                // Second
```

```
                print("sw $%d, %d($%d)\n", tmp, off + 1, reg);
                print("srl $%d, $%d, 8\n", tmp, tmp);   // need this??
        }
        else
                // SW for .Byte Here!
                for(i = 0; i < 4; i += 1)
                {
                    print("sw $%d, %d($%d)\n", tmp, off + i, reg);
                    print("srl $%d, $%d, 8\n", tmp, tmp);
                }
}
```

What this effectively does is when loading bytes the compiler will put out assembly instructions that treat each byte as being on an individual line of memory and then pack them into a register so that when the CPU deals with the data in the register it would do so as it would normally would. The same thing is done for stores which will unpack the data and store each byte on an individual line in memory.

So after this translation the compiler can deal with the bytes as it normally would since they are packed into the register. The only other issue is to ensure that when the resource is accessed the data is stored into data memory correctly. This is done mostly by the assembler as is described in the next section. However, it was also important to make sure that the correct data location was used by the .byte directives. This was done by appending the following after the string label:

```
print("la $1, 0($0)\n");
```

This would load 0 into the $1 register which is the temporary compiler register so that when the .byte directives are "run" and exited that the correct location in memory is accessed. Notice that this instruction has the possibility of being used with differential locations in memory. Currently the code will write only to a specific point in memory opposite to the Stack Pointer and will hopefully not have problems regarding collisions. I could not find a way to extract size information from the labels so for the time being the location must default to 0.

So with these precautions taken and with the assembler dealing correctly with the LA instruction and the .byte directives, string resources will be dealt

with correctly although through a somewhat convoluted process. They are first unpacked into memory by running a snippet of code which is derived by the assembler from the .byte directives. Then the loads will pack the data memory into registers and when storage is required they are unpacked into memory. There are a few issues regarding the default locations in memory since string manipulation may not correctly work, this has not been fully tested although multiple strings have been tested and work correctly.

3.1.3 Unsupported Instructions

The NDMA supports a variety of instructions but for simplicity of testing and ensuring that the system worked many of these instructions were disabled. This included more complex instructions like multiply and divide. However, there were a few instructions that simply required validation time that have yet to be integrated enabled on the CPU which were not deemed that important that the difference between them being one instruction or two mattered.

For one only BNE and BEQ branch instructions were fully tested and so any form of branch was some permutation of SLT, BNE, or BEQ. This is shown below:

```
stmt: EQI4(reg,reg)  "beq $%0,$%1,%a\n"   1
stmt: EQU4(reg,reg)  "beq $%0,$%1,%a\n"   1
stmt: GEI4(reg,reg)  "slt $1, $%0, $%1 \nbeq $1,$0,%a\n"   1
stmt: GEU4(reg,reg)  "sltu $1, $%0, $%1 \nbeq $1,$0,%a\n"  1
stmt: GTI4(reg,reg)  "beq $%0, $%1, 2 \nslt $1, $%0, $%1 \nbeq $1,$0, %a\n"   1
stmt: GTU4(reg,reg)  "beq $%0, $%1, 2 \nsltu $1, $%0, $%1 \nbeq $1,$0, %a\n"  1
stmt: LEI4(reg,reg)  "beq $%0, $%1, %a \nslt $1, $%0, $%1 \nbne $1, $0, %a\n" 1
stmt: LEU4(reg,reg)  "bleu $%0,$%1,%a\n"  1
stmt: LTI4(reg,reg)  "slt $1, $%0, $%1 \nbne $1, $0, %a\n"   1
stmt: LTU4(reg,reg)  "sltu $1, $%0, $%1 \nbeq $1, $0, %a\n"  1
stmt: NEI4(reg,reg)  "bne $%0,$%1,%a\n"   1
stmt: NEU4(reg,reg)  "bne $%0,$%1,%a\n"   1
```

Also NOT instructions were not implemented on the CPU so a NOT was interpreted as a combination of addi and XOR to first initialize a register to all ones and then xor that with the register in question.

```
reg: BCOMI4(reg)  "addi $1, $0, -1\nxor $%c, $1, $%0\n"   1
```

3.1.4 Stack Pointer

As described above the memory is of width 32 bits and thusly when the stack needed to be manipulated it did not need to be incremented/decremented by counts of 4. Instead it manipulated by counts of 1 and as such through the LCC back end this had to be changed. Most of this was done by setting the argument offsets as 4 rather than 16 and adjusting values in the back end to ensure of this. Most of the changes were implemented by validation through the use of simple test programs.

3.1.5 Usage

To use the compiler it must be built. To do so you must run the following in a Visual Studio 2005 build window or a build environment which has nmake from Visual Studio 6.0 or later:

```
...ProjectModules\LCC>set BUILDDIR=\lcc_build_directory
...ProjectModules\LCC>nmake -f makefile.nt all
```

This will build the compiler into your build directory you specified and then to compile a program you must run:

```
...build_dir> rcc -target=ndma/ndmaOS filename.c
```

This will spit out the assembly code which can then be "cut and paste" into the assembler code which should include some of the APIs as explained below. If wanted it is possible to output the rcc output to a file by doing the following:

```
...build_dir> rcc -target=ndma/ndmaOS filename.c > outfile
```

This file can then be used by the NDMA Suite as explained below. If you add the RCC path to the system's PATH variable (accessed through control panel-¿System-¿Advanced-¿Environment Variables or the SET command in the command prompt) then you can utilize the NDMA Suite along with NDMA Merge to output files that can then directly be assembled and boot loaded.

3.2 NDMA Assembler-Linker

The NDMA Assembler-Linker is a combination of a linker and assembler to simplify the integration chain design so that it could be feasibly designed in the time allotted. The Assembler began development along side the development of the NDMA CPU Architecture as to guarantee that the two will work together in unison from the get go. However, as time progressed and requirements of the assembler grew the design of the assembler had to be re factored so that new instructions and functionality could be incorporated into the program.

The NDMA Assembler-Linker consists of two parts as the name suggests. The first part is the linker which needs to run before the assembler considering the design of the program and it's structure. After the linker runs through the file one time the assembler is run assembling each instruction and using the data collected by the linker to set the correct targets for the assembled instructions.

3.2.1 Linker

The linker needs to know almost as much about each line of the input assembly file as the assembler itself. If not the target locations of jumps will not correctly be calculated. This means that the linker portion of the assembler is pretty close to the assembler itself except it does not need to decode the instructions but only match them up to the data base of accepted instructions, directives that are translated to instructions, pseudo instructions and the correct behavior of these pseudo instructions based on differing circumstances, and different special instructions. All the while the parsing of the file will tabulate a data structure with all of the different labels and the correct PC location of those labels in program code.

The way that the linker does the above is by reading each line of code at a time. This is done in 100 character pieces since it is assumed that each line should not exceed this amount in characters. This input is then tokenized and the first token is compared against a number of conditional statements in an exclusive order. This is done in such a way that the special instructions take precedence over the normal ones since all instructions are registered in the s_mapOP, which is a map data structure of all of the instructions the assembler supports.

First the token is compared to the LA instruction. As explained above

the LA instruction is a pseudo instruction that has different functionality in a number of different circumstances sometimes being translated to 2 instructions and other times to only one. The exact way this is done will be explained more fully in the assembler portion below. There are two different circumstances for the LA pseudo instruction. Either it is calculating an effective address and will be translated to one instruction which will be an addi. Or it will be translated into two instructions when the offset is found to be a label. Since at this stage we cannot guarantee that the label specified in the LA offset field will be in our database we must check to see if it is a label in a different way. This is done by noting that label based LA instruction will have at least one letter in the offset field. Otherwise this field must be an immediate type number. If a letter is found in this argument then the LA instruction increments the PC counter of the linker by two otherwise only by one.

The next special instruction that is looked at is called SRI or Set Register Instruction. This instruction is further explained in the later network driven assembler portion however the basic functionality of it is that it takes a register and a normal instruction as two inputs and then loads that instruction into the register. This requires a register clear with an ORI instruction and a combination of one LUI and one ORI instruction. This sums to a total of three instructions and when the linker comes across this instruction it increments the PC counter of the linker by three.

After these two special instructions the linker checks for an empty line. If it is not an empty line the linker checks against the directives .Boot, .SetStack, and .byte. The .Boot and .SetStack directives both get translated to one instruction. The .Byte directive, however, gets translated to two instructions so in the case of the .Byte directive the linker adds two to the linker's PC counter. In the other cases it simply adds one and in the case of an empty line the linker skips on to the next line. Also at this point the linker compares the first token to the list of supported assembler instructions. If it matches then one is added to the linker's PC counter.

If after all of the checks the first token of the line does not match any of the above conditions then the first token is checked for consistency with the format of a label. The way that this is checked is by checking to see if a colon is found in the token. If yes the token is tokenized and then inserted into the s_mapLabel map data structure. When this is done it is checked to see that duplicate labels are not found in the file. If this is the case then an error is thrown and the program exited showing which label has been repeated.

Once the linker is done the s_mapLabel data structure should contain all of the information regarding the PC location and names of the labels that are encountered throughout the assembly file as well as ensure that no duplicate labels are found in the file.

3.2.2 Assembler

Assembling an instruction requires a few different steps. Assembly is done slightly differently for different groups of instructions but apart from the special instructions, directives, and pseudo instructions all of the instructions are assembled in generally the same manner.

The first phase that the extracted line goes through is a quick and dirty token counting check. This is useful for a number of quick tricks to get the pipe going but mainly it provides a way to allocate memory for the correct number of tokens to be saved and used later. After the quick token check the input line is actually tokenized and each argument saved for use in the assembly process later.

After this the first token is run through a number of conditionals to check to see if the line is empty, a comment, directive, special instruction or normal instruction. If the line is empty or is a comment the pipe is skipped and the next line is looked at by the assembler. However in the case of a normal instruction (the other cases are dealt with below) there can be a few different cases. If the assembler receives more than 8 tokens it spits out an error message indicating that the instruction has too many arguments. If there are between 3 and 8 arguments then the instruction is dealt like a normal instruction other than J type instructions. If the instruction has 2 or less arguments then the instruction is dealt with as a J Type instruction.

Notice that if the first argument is found to be SRI this sets a "flag" for the rest of the program of fSRI to 2. This value is then used to offset all of the argument values if it is set so that after SRI has been found then the pipe will only look at the instruction passed to SRI. In this case the assembly works normally except only on the instruction passed to SRI. The SRI instruction is explained in more depth below.

3.2.3 Directives

Checking the line to see if is a directive is done by checking the first character of the line. If this character is a '.' then that line refers to a directive. The

LCC compiler spits out a good amount of directives that are not used by the Assembler but these were kept in since they provide good information about what the assembly code coming out of LCC means. The only directives that are directly implemented by the assembler are the .SetStack, .Boot and .Byte directives.

When a directive is encountered the period is stripped off and it is compared to the supported directives. Since this list is not long this is done manually but it would not be hard to implement a map of all supported directives with the current model since the period is tokenized away. If the directive is seen to match the .SetStack directive (which is a very important directive for the correct functionality of the data memory stack) then this directive is replaced by an ORI instruction which then sets the stack pointer to the value specified by the directive. By using this directive it is possible to designate to the program what the size of the CPU's data memory is. The correct setting of this is imperative for correct functionality. Otherwise, the stack will be placed at zero and the memory may behave unreliably. Note, however, that when the SetStack directive is used a variable in the assembler is set and will output statistics regarding the program's size versus the size of the data memory. Note that the data memory can be a different size than the instruction memory and the boot loader will load memory that does not exist. This debugging message can be double faced in that sense and realize that the SetStack directive only has to do with the data memory and not the instruction memory.

The next directive is the .Byte directive which will map to two instructions. The way that resources work on this architecture is that the .Byte directives become a small snippet of code which is called whenever the resource is needed. In this way each byte is loaded into it's own specific point in data memory and then the code snippet will return. The way that the code gets to this snippet is through the LA instruction which is explained below. However, each byte must be loaded into it's own location in memory and different strings of resource must be independent. This way the cData variable will be reset to zero when no byte directive is encountered to effectively reset it. Every time a succeeding byte directive is encountered, however, it is incremented. This way each string gets it's own space while each byte gets a succeeding location in memory. The cData variable is used to assemble the two instructions the .Byte directive gets translated to.

The last directive is less involved. The .Boot directive with be replaced with a jump instruction to the location of the label provided. The common

use of the .Boot directive is .Boot main which will make sure that the first code run is the main code since between the .Boot directive and the main code there may lie a good amount of other code such as APIs or resources.

3.2.4 Normal Type Instructions

For normal type instructions the first token of the instruction is checked up against the s_mapOP map data structure. This data structure is initialized before the running of the assembler by the function:

```
(void) InitializeOPMap();
```

This function goes through the OPS enum and assigns an appropriate string to each correct enum element. These enums are then later used by the function OpFunctionSADecode() which will go through the map data structure and find if the token matches. If the token does not match an error message is output, however, if the token does match then this function will set the binary OP, and FUNCTION fields to the correct strings to be assembled together later. This function will also set the instruction type in the variable ins_type which is then used to group instructions more effectively, although this generally is not precise enough and specific instructions must be checked individually as well as the instruction type.

After this stage the only thing left to assemble are the RD, RS, RT, IMMEDIATE, TARGET, and SHIFT AMOUNT fields. Before this is done the instruction passes through a number of conditional statements which ensure that the correct argument is used for the correct field since different instructions have different structures. This is done for convenience so that when the fields are assembled we can use a variable index rather than index constant values differently for different instructions.

After the field indexing stage the RD, RS, and RT are assembled through a function call RegDecode() which will take in one of the tokenized arguments and pass back a binary string that represents that argument. This function can support register arguments in their numerical form of "$31" or their symbolic form "$ra". This is nice since it provides us flexibility in how the assembly code is written and makes the code somewhat more readable. Although the output of the LCC code will still be numerical the APIs which are generally written in assembler will be somewhat more legible.

At this point the only fields left are the IMMEDIATE, TARGET and SHIFT AMOUNT. The immediate and target fields are taken care of together. The normal type instructions only deal with the immediate field so the target field can be ignored. First the immediate argument is checked against the map of labels. If the instruction is a branch type instruction then this label is treated as relative rather than absolute. Whatever the value ends up it is saved to binary and then to the immediate field. If the immediate argument is not a label it is first checked to see if it is of hexadecimal format. Hexadecimal immediates were rather useful for debugging and general low level programming so that is why this feature was implemented.

If the argument was not of hexadecimal format it had to first be tokenized to see if there was an operand between two values. The LCC compiler on occasion would output expressions rather than constants. This was up to the LCC compiler and not the back end so I had to add this functionality to the assembler. The expression is tokenized and evaluated and if it is found that no operand exists or only one value is there then that value is simply converted to binary and saved into the immediate field. Once the expression is evaluated it's result is placed into the immediate field.

The only field left now is the SHIFT AMOUNT. The shift amount field is passed to the assembler in the immediate argument. This makes it easy for the assembler to find so it is taken, converted to binary and placed into the shift amount field. This is only done for SR_TYPE instructions which stand for Shift Register instructions.

At this point all of the instruction fields have been correctly set and all that is left is to assemble them together. This is done differently for different sets of instructions and different instruction types by adding the different pieces one after the other into a buffer of length 32. Once this is done it is copied into the output stream as well as the output file. The output stream is nice to have since debug messages can be appended to it while the output file is the one that is sent to the boot loader. There are many different categories of instruction assembly but they are all done according to the rules of that specific instruction. This is important to note when adding new custom instructions to the assembler. Every portion of the process must see the effect of the new instruction. Occasionally the new instruction will not fit into a previously made grouping so a new one must be made. This must be done in an exclusive fashion.

3.2.5 J Type Instructions

The J Type instructions always have two arguments or less and so have their own special way of being assembled. This way is instruction specific. Since there were few J Type instructions and all other instructions that fit into this mold were dealt with in such a diverse way this was the preferred way to do this. In this section I will only talk about the J Type instructions.

When an instruction is found to only have two arguments or less it is assembled on an individual basis. The only J Type instructions fitting this description are the J, JR, and JAL instructions. In each case the appropriate arguments are decoded and assembled as is done for the normal instructions but instead of using a pipe it is done on a per instruction basis.

The JAL instruction is followed by either an immediate type target or a label. When the instruction is assembled first the target is checked against the label map. Since the JAL address is absolute no relative calculations need to be made and the PC value of the label is converted to binary and then is assembled with the rest of the instruction fields. If the target is just an immediate then it is converted to binary and set into the target field and assembled with the rest of the instructions.

The JR instruction is done in a similar fashion except the register argument is decoded using the RegDecode() function and then set into the appropriate field and the instruction is then assembled. The J instruction is very much similar to the JAL except for slightly different OP codes. Since the instructions are done on an individual basis each instruction needs a different conditional block for the OP codes since the function call used in the regular case is not used.

3.2.6 Special J Type Instructions

There are a few special instructions that register as J Type instructions although they are not jump instructions. For example the BREAK instruction is a special instruction that tells the CPU to temporarily halt. This is useful for hardware debugging. Since the instruction is flat when this instruction is encountered the flat value is assembled. This is the same case for the NOP instruction. The only exception to this rule is the QUIT instruction which is not really an instruction but rather more of a debugging assembler directive. This is useful to halt the assembly of a file for different purposes. This was put there during debugging and development but never taken out due to its

usefulness.

3.2.7 Network Instructions

There are a few network type instructions that the assembler will parse directly. These include the SID, BCST, BCSTR, SMSG, and SMSGR. These are the CPU side network instructions and these are decoded in a particular way. SID stands for Set ID which allows the initialization of the ID of a CPU. This ID is extremely important if it is expected for the CPU to receive any messages or send any messages. SID is parsed as a J Type immediate instruction. First the number of arguments will identify that this is a J Type and then SID is processed and the instruction is assembled with the immediate converted to binary.

The BCST instruction is a broadcast instruction. It will broadcast the immediate field as a message with destination ID 0xFF as described above. Also the message is not absorbed but passed on in all directions. The way that this instruction is assembled is similar to SID since it will only have 2 arguments. The number of tokens designates this instruction as a J Type and in turn will send the assembler to the BCST conditional. Then the instruction is assembled with the immediate converted to a binary number in the target field.

SMSG stands for Send Message and is a little different since it will have three arguments. This instruction is an immediate type instruction and the way that this instruction is assembled is the same as the LUI instruction. First the OP code is generated from the OpFunctionSADecode() function and then the immediate, RS and RT field indexes are set appropriately. When it is time to assemble the instruction all of the fields will be valid as has been shown for all of the other instructions.

SMSGR and BCSTR are register counterparts of SMSG and BCST. This way it is possible to send a specific byte of a specific register as explained above. The form of these instructions are immediate type instructions since the form of these instructions are as so:

```
smsgr $destination_id, $data, byte_offset
bcstr $data, byte_offset
```

As can be seen for the BCSTR RS and RT should be the same since the way that it is decoded by the CPU will only look at one of the two.

The way that these instructions are parsed are by setting their correct types as I Type in the OpFunctionSADecode() function as well as their correct op codes as well as correctly setting their field indexes as I Type as well. Once the assembler is done with decoding the rest of the fields and skipping or skipping them for BCSTR the instruction is assembled where byte_offset should have already been converted into the immediate field correctly.

Then there are the network side network instructions such as NDJR, JALNET, and SNIP. These must exclusively be used with a SRI pseudo instruction which is explained below. Otherwise they must be hand coded with the BCSTR/SMSGR instructions. JALNET is simple to use since it will only be one argument always. This instruction is the network equivalent of JAL except the target is designated by the CPU network memory controller. When this instruction is encountered assembly consists of simply outputting a constant binary sequence.

NDJR is a little more involved. It stands for Network Driven Jump Register or the network equivalent of JR. It is passed along with a register argument. The assembly of this instruction is the same as JR except with a different OP code. The last network driven instruction, SNIP, stands for Set Network Instruction Pointer. This allows a remote CPU to set the instruction pointer for the network memory and a point at which to jump to with JALNET. The argument passed to SNIP is an immediate and SNIP is dealt with exactly like JAL except with a different OP Code. It is important to note that when SNIP is used the instruction pointer is set as well as the instruction write point. However, when instructions are dispatched to other CPUs the pointer does not change, only the write position. This allows the JALNET to jump to the beginning of the dispatched instructions rather than the end of them.

3.2.8 PseudoInstructions

The Assembler had to support two pseudo instructions. This included the the SRI pseudo instruction that is heavily used with the network instructions and the LA instruction that is used by LCC for a variety of things mainly calculating the effective address. However LA also has other functionality as producing a pointer to a location of memory where a resource is stored.

The SRI pseudo instruction stands for Set Register Instruction and will set a register to the value of a specific instruction passed to it. This is very useful to be used along side with the BCSTR or SMSGR network instructions

as it allows for re usability of code and a less cumbersome method of network programming. SRI had to be built into the assembler on every level since the SRI pseudo instruction needed to work for all instructions while at the same time being it's own instruction.

The way that this was done was first if the instruction was tokenized to see that it was an SRI instruction an fSRI integer flag is set to the value of 2. This flag is used to offset all of the argument values throughout the assembler. If the instruction is not an SRI instruction then the instruction is dealt with normally since the flag is zero. Otherwise the flag effectively shifts the arguments over by two and the instruction passed to the SRI pseudo instruction is passed through the pipe.

When it comes time for the instruction to be assembled it is seen through the fSRI flag that this was part of an SRI pseudo instruction. The assembled instruction is then split up into two pieces and then the destination register given to the SRI instruction is first reset to zero using an ORI instruction and then the top half of the instruction is set using a LUI and the lower half using another ORI instruction. Notice that doing this through pure code would be cumbersome since the assembler does not provide ways of passing binary immediates and to otherwise do this the instruction would need to be assembled by the programmer, converted to hexadecimal, and then split up. Instead this automation provides a much faster way for network programming.

The other pseudo instruction is the LA instruction. An LA instruction is dealt with as a normal type instruction for the most part except it's instruction type ins_type is set to P_TYPE which represents a pseudo instruction. The LA instruction does not take the SRI into consideration since the LA is a more LCC regarded instruction rather than a network oriented instruction. The way that LA differentiates between it's two modes of operation is two fold. An LA instruction used a a jump to a resource would have a certain amount of arguments and so the token count is first used to gauge. If the LA instruction in fact has three tokens then the third argument is checked to see if it is a label. In this case it is obvious that the LA instruction is a resource oriented LA instruction.

In the case that the LA is not a resource oriented instruction the LA instruction is simply replaced with the ADDI instruction and assembled correctly. In fact when the LA instruction is decoded the default OP code is copied into the OP field of the ADDI instruction. If the LA instruction is found to be a resource oriented instruction it must be replaced with two in-

structions. The first is a JAL instruction to the location where the resource code snippet is located. This will unpack the data into data memory and set the assembler temporary register $1 to the location in data memory at which the resource is being saved. Once the code snippet JRs then the second instruction of the resource oriented LA instruction is executed which is a ADD of the assembler temporary register $1 with $0 set to the destination register originally specified in the original LA instruction. In this way after the code snippet is run the compiler can pack in the data as described in the LCC section into registers and unpack it equally back into the data memory. For resource strings to work both the assembler and compiler must be working in unison in this regard.

3.2.9 Conventions

The practice of programming complex programs into a highly parallelized system was not investigated very thoroughly but the design of simple programs was enough to demonstrate some of the basic conventions that would be required to effectively design parallel programs with multiple processors using this architecture. Some of this was touched upon above with the description of the SRI pseudo instruction but this is useless without a variety of APIs that would allow a programmer not only to set a register to an instruction but also to dispatch that instruction.

It is obvious that for a 32 bit instruction to be dispatched 8 bits at a time it would require 4 cycles and in turn also 4 instructions. In the system designed memory was very sparse to having to repeat the same 4 instructions in the code every time an instruction had to be dispatched would have been highly inefficient. Instead this was pushed into an the following APIs:

```
BCST_R:
bcstr $a0, 3
bcstr $a0, 2
bcstr $a0, 1
bcstr $a0, 0
jr $ra

BCST_RR:
sri $t0, bcst 5
```

```
# Send MSByte in bcst msg
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 3

# Send Second Byte
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 2

# Send Third Byte
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 1

# Send Last Byte
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 0
jr $ra
```

The main differences is that BCST_R will dispatch the instruction to the CPU to be entered into the instruction memory. On the other hand BCST_RR will dispatch not the instruction but the code to dispatch the instruction back. An equivalent API has not yet been developed for the SMSG since debugging was much simpler using the broadcast instructions however the difference would be simple since all that would be required is to pass a specific ID into $a1 and then use this as the destination ID register in the SMSG instruction.

So using these APIs is rather straight forward although the development of programs using the architecture that actually utilize the parallelism may not be completely obvious. Currently the only real convention that I have conceived is the situation shown by Figure 12. The idea behind this is the

"Master CPU" convention.

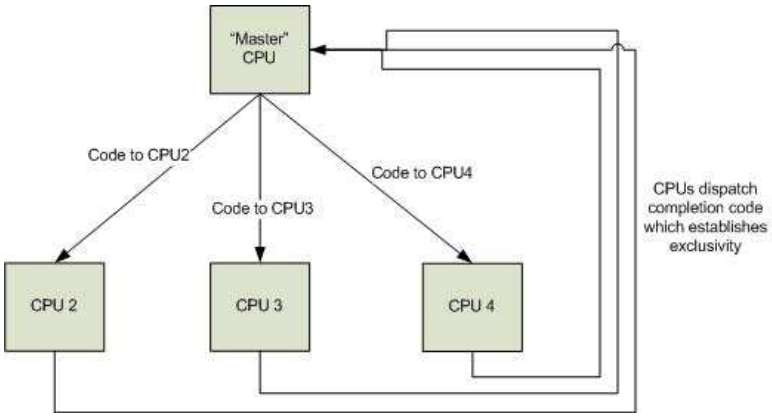

Figure 12: Master CPU NDMA Convention

In this scenario there exists some Master CPU on the network that is the access point for all of the other CPUs. In this way the user can manipulate this master cpu with resources such as a keyboard and see the results using a monitor and the other CPUs are used for reconfigurable purposes. This model is similar to modern PCs where the main CPU will instead manipulate other systems through the bus.

The way this would work is that the Master CPU will dispatch code to each of the CPUs sequentially using the SMSG/BCST_R API. The SMSG/BCST_R function call is designated for code that will then run on the target CPU. Once all of the _R code has been dispatched for all of the CPUs the _RR code is dispatched. This is code that will then run on the Master CPU. This code will usually consist of an infinite loop. The convention now is that when a CPU is done with it's dispatched _R code it will then execute the _RR code which will load up that CPU's completion loop in the Master CPU. The basic idea of this is shown in Figure 13.

It is important to make sure that the _RR code must not be running on multiple CPUs and the easiest way is to use dedicated registers which can be decided by the Master CPU to check for individual CPU completion. In this way completion order will be maintained. This should not affect parallelism if used correctly. In fact a dedicated register can be used that will act like a mutex.

Once the target CPU is done dispatching it's completion loop to the Master then the target will dispatch a JALNET and the Master will then

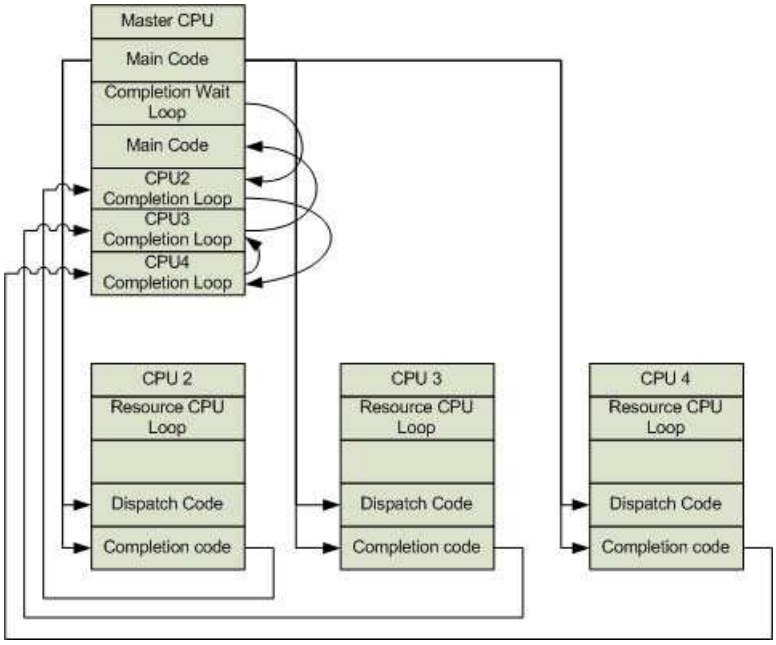

Figure 13: Master CPU NDMA Code Convention

enter that conditional loop. This conditional loop can be used to exit back to the Master code based on other CPU's completion based on dedicated registers or it can be another infinite loop which requires the completion of a different CPU to let it exit. Eventually all the CPUs will complete their dispatch code and will then dispatch back their completion loops. These loops will execute and a conditional in them will trigger to let the Master CPU return to the code directly after the original infinite loop letting normal operation resume.

Note that this is only one specific way to utilize the Master CPU convention. It is also possible not to send completion checks and allow the CPUs to continuously run or to have the Master create other Masters and so on. This model seemed to be one of the simplest ones to use that could in turn be aggregated to produce more complex results.

3.3 NDMA Bootloader

The bootloader will take a file from the assembler in the form of a .mem file and will boat load the CPU with it. The program needs no configuration since

the baud rate of the RS232 module on the FPGA design and the program match at 115200. However, if the computer does not support this baud rate for some reason it is easily changed in the program and in the Verilog.

The program simply runs through the file and spits out the location and instruction on and verifies the receive from the CPU. In the case that the receive does not match the program will return the two mismatches, and quit. This will leave the CPU in a state where it must be restarted disallowing for strange behavior. When the program reaches the final instruction in the file it will repeat it a few times (which may not be wanted but is useful for the usual break or jr $31 instructions) and then send a 0xFFFFFFFF as the PC and wait for the reply. The reply will then confirm that the CPU has finished the boot loading process and received all of the data correctly.

3.3.1 Usage

To run this program copy it into the directory of the assembler or the .mem file the assembler is outputting and then run it as such:

```
> NDMABootloader COM# filename.mem
```

COM# would be the com number and used such as COM4 for the COM4 port. The filename is the name of the file you would like to bootload.

3.4 NDMA Merge

NDMA Merge is a very simple program that will parse through two assembly files and splice them together. Its a quick and dirty program which opens each file with the standard input and output library as well as creates a new file called ndmamerge.asm. The ndmamerge.asm file is a temporary file and should be dealt with as such. It is the step between the compilation to assembly code and the assembling and linking of the code.

The way that NDMA Merge will work is that it will copy in instructions from the first file until it reaches the .BOOT directive. Since between the .BOOT directive and the label to which the .BOOT directive points the location of the memory does not matter since it should be referenced with labels anyways the merge program at this point splices in the other asm file. The second asm file is usually by convention some form of API. Once the end of the file is found the rest of the first file is copied into the file and the program exits.

This will guarantee that the output of the NDMA Merge program will have the structure as shown below:

```
.SetStack 255
.Boot main
... API Code ...
... Other Code ...
main:
... main code ...
jr $ra
.program_end
```

3.5 NDMA Suite

The NDMA Suite is the combination of the NDMA LCC Backend Compiler, NDMA Merge, the NDMA Assembler / Linker, and the NDMA Bootloader. The way that these programs are combined is through the use of a script called NDMA Make (ndmamake.bat). This is a DOS Shell script file and can be easily ported to another platform. It simply steps through the process of making a file from compilation to bootload returning an error at each point in the case of failure.

3.5.1 Usage

```
Usage: ndmamake file.c api.asm COM#
This will compile file.c into a asm file with the RCC compiler and then splice
Example:
ndmamake filename.c api.asm COM3 \r
```

4 Results and Analysis

It is important to note here at the below results and analyses are the results demonstrated on the FPGA rather than on the behavioral simulation models of Verilog. The design was rather complex with multiple clock domains and self timed regions. This meant that although the behavioral models worked and were tested completely that when they were implemented in the actual FPGA a fair amount of issues arose usually dealing with the borders between the domains.

This is to say that the project was fully tested in Verilog simulation before it was moved to the FPGA. In that sense it was working and results validated. In simulation everything worked and validated the design. However, a few design choices could not be synthesized and made for a difficult transition to the FPGA since they had to be redone.

4.1 NDMA System Single Core

In this section we will describe the results of the single core system operating on its own using all of the tools provided by the NDMA Suite with no real multiprocessing power. The system designed for this section is shown in the Figure 14.

The results of this were very good and it is possible to write a program in ANSI compatible C, run it through the NDMA Suite and have the program run correctly on the CPU. The CPU had to be clocked slower due to timing issues that arose due to fanning out the clock to the GPU's PLL. Currently the CPU is clocked at 3 MHz although faster speeds are likely possible. This was not very well explored but the method that was used was using a counter that would flip every so many counts thusly isolating the CPU's clock from the master clock. This worked well using the 27 MHz clock and counting to 9. This allowed for a 3 MHz clock but likely could be sped up as long as all the timing issues are dealt with correctly.

Using the Bootloader it was possible to boot load a program with 521 lines of assembly in 42 seconds. This speed is also dependent on the computer running it however since the Bootloader program was written using Microsoft code. This is about a 2Kb program and since the boot loader does such a large amount of error checking it is expected to go at this slow rate. However, this is not a terrible speed either since the boot loader was purposefully slowed down to ensure correct operation and if to ever be more polished could likely be sped up much more.

The program used to fully test the single core system is shown in Appendix B. This program is a basic PONG game which ties in all of the functionality together. Although it does not test the APU it does tie in the PS2 input buffer with the GPU and is a sufficiently complex program using multiple string resources. Figures 15, 16, and 17 show a few of the screen captures demonstrating a win and loss situation. The game is operated by pressing j/m for up and down for the right player and a/z for up and down for the left player.

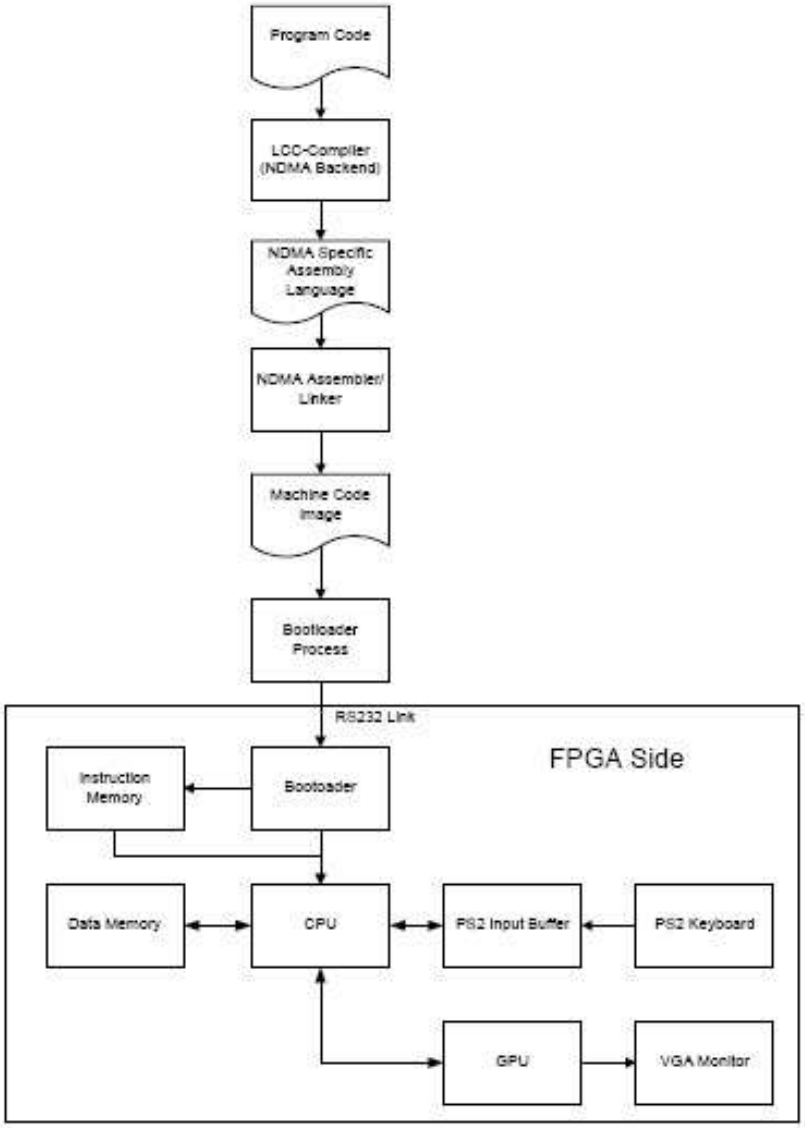

Figure 14: Single Core System Implemented

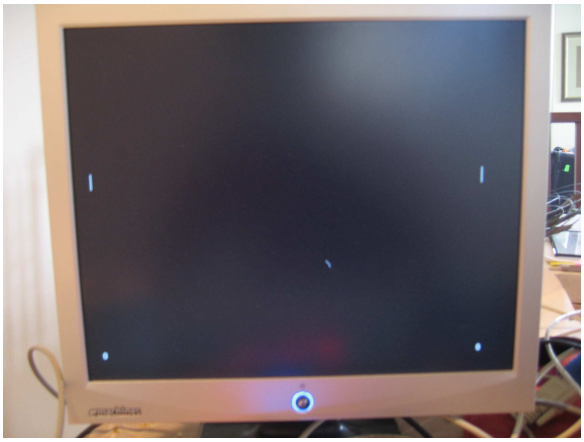

Figure 15: PONG game test program screen captures: Start of Game

Figure 16: PONG game test program screen captures: You Win!

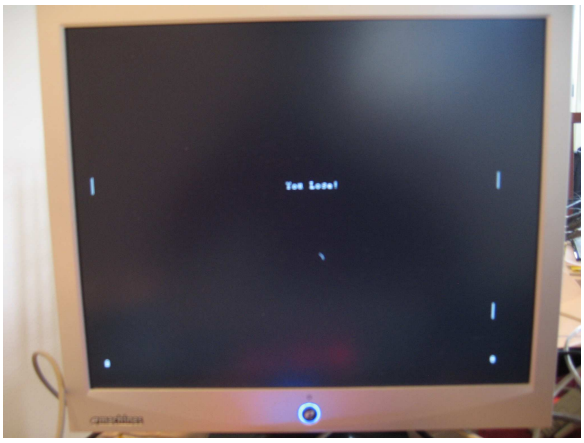

Figure 17: PONG game test program screen captures: You Lose!

Overall this portion of the project was exceedingly successful and would be a great base for future development to make into a complete open system for system on chip development. It would be possible for others to integrate new instructions, build on the current design, integrate new design elements and further polish the system.

4.1.1 Open Issues

With such a large system it is inevitable to have bugs in the system that cannot be identified at first glance or during development. This project consisted of mainly development and hard as I tried to integrate validation into my development cycles I cannot guarantee that there are no bugs in the system. Of these bugs I only know of one. This is a strange bug that occurs with resource strings of length greater than 16 characters. This is surely an LCC back-end bug and came out late in development. At the time it was not worth it to fix since it seemed to be a rather deep bug that would require a large cost to fix. This is noted here so that if it is ever noticed it will be known that this bug exists in the LCC backend and is not a timing or architecture issue as well as not an assembler/linker issue either.

There are likely many other bugs as well that I have not discovered but the one above is the only one that I have experienced. As said before this system was not very well validated and I would think that it would require a few months of purely validation with a second development cycle and one

more validation cycle to truly be water tight. Also a large amount of code could easily be re factored in the assembler/linker. When implementing the SRI pseudo instruction it had to be quick since not much time was left and while most of this project attempts to be as object oriented and reusable as possible this is an academic project with certain resource and time constraints which made the decision to make occasional bad design choices OK since it did not affect speed and functionality. With that said the code at these points is very ugly and not readable and would obscure aforementioned bugs so there is a good interest in eventually cleaning up this code.

4.2 NDMA Multi-core and Network Layer

To correctly implement multicore systems using NDMA each core must be given a unique ID set by a ROM. This ROM should have the following format as was used in all of the systems described below:

```
sid ID
nop
nop
j 1
```

The first test to see that a multi-core system was operating on the FPGA was to simply put two cores onto the FPGA with their own memories and let them run side by side with no networking at all. To test this the switches on the DE2 board were used to switch the hexadecimal display to show the instruction running on the different CPUs as well as using the LEDs to display the program counter. This seemed to work fine so the Master CPU convention was used where one CPU was wired up to all of the IO hardware while the other one was left floating and networked to the first CPU.

The next test also tested all of the multi-core and network layer functionality other than the message passing algorithm and the register-thru design. This consisted of having the Master CPU dispatch a SNIP and JALNET instruction to the other CPU and seeing that the other CPU in fact does jump to the location in memory. This was done successfully and it was demonstrated by different SNIP locations in memory where it could be seen that both SNIP and JALNET were working. This only showed the functionality of the network side instructions for the second CPU so the next test was to dispatch the following code.

```
snip
nop
nop
nop
break
jalnet
```

The second CPU was then observed to make sure that it reaches the BREAK which it would not unless it enters the network dispatch code. This was also done successfully and demonstrated that it was possible to dispatch and run code on other CPUs using the Master CPU model.

The final test of the two core system was the design of a simple parallel program. This program is compromised of three different operation states: dispatch, execution, and completion. The code resides completely in the boot image of the Master CPU and it is to be noticed that the loop code on the overall is going to be run more than the amount of effort required to dispatch the code. The loop is as concise as possible since the shorter the dispatch code the less instructions required to dispatch it.

This simple program dispatches a counting loop to the CPU which then counts to 170. Once dispatch is complete the Master CPU enters a wait loop while the other CPU is executing the code. This is the execution phase of the program. Once the CPU completes it's loop it will in turn dispatch back it's completion loop. This is a conditional loop that will check a register for the correct value (170) and let the Master CPU return to it's point after the wait loop. The Master code is shown below:

```
sid 1
nop
# Dispatch Code:
# ori $t0, $0, 0
# ori $t1, $0, 170
# addi $t0, $t0, 1
# bne $t0, $t1, -2
# bcst: snip 511
# bcst: j complete
# bcst: jalnet
# break
sri $a0, snip 42
```

```
jal BCST_R
sri $a0, ori $t0, $0, 0
jal BCST_R
sri $a0, ori $t1, $0, 170
jal BCST_R
sri $a0, addi $t0, $t0, 1
jal BCST_R
sri $a0, bne $t0, $t1, -2
jal BCST_R
sri $a0, snip 511
jal BCST_RR
sri $a0, j complete
jal BCST_RR
sri $a0, jalnet
jal BCST_RR
sri $a0, break
jal BCST_R
sri $a0, jalnet
jal BCST_R

complete:
ori $a0, $0, 55
ori $a1, $0, 55
ori $a2, $0, xFF
jal plotpixel
break
```

Note that the BCST_RR instructions set will make the Master CPU jump to a label that the assembler knows about. This is an assembler trick that in the future may be able to be extended to a deeper degree than used here but for the time being this demonstrates the ability of this system to respond. Broadcasts are used here simply because there exist two cores on the network so IDs are not important.

After a good amount of debugging this program was successfully run and a pixel plotted on the screen. Since the only way that the Master CPU can reach the complete label is through the dispatched instructions this shows that network code was running on both machines. To take a closer look at

the exact utility of this code we see that the BCST_R function including the SRI and jump requires a total of 8 instructions each and the BCST_RR requires 20 instructions. The BCST_R is called 6 times and the BCST_R is called 3 times for a total of 108 cycles. The dispatched loop will run 170 times and average 3 instructions for each loop so we get remote operation of 510 cycles. This means that we may have 402 cycles on the Master CPU to do something else in parallel. This program effectively demonstrates the advantages of this kind of parallel system.

Finally this program was also tested using the multicore test program shown below. The API is omitted for clarity but here we can see that messages are being passed back and forth between both CPUs and the IDs are also being used. Since the broadcast and the send message instructions are very similar in structure this only required some polishing of the APIs to allow easy access. Notice that the destination ID is always in $a0 even if we are calling a SMSG_RR. This is helpful since we do not need to waste instructions setting up the destination IDs although we still have the flexibility of providing different destination IDs for both target and return allowing us to dispatch code that will then be dispatched to a different CPU.

```
main:
sid 1
nop
ori $a0, $0, 2
sri $a1, snip 42
jal SMSG_R
sri $a1, ori $t0, $0, 0
jal SMSG_R
sri $a1, ori $t1, $0, 170
jal SMSG_R
sri $a1, addi $t0, $t0, 1
jal SMSG_R
sri $a1, bne $t0, $t1, -2
jal SMSG_R
ori $a1, $0, 1
sri $a2, snip 200
jal SMSG_RR
sri $a2, j complete
jal SMSG_RR
```

```
sri $a2, jalnet
jal SMSG_RR
sri $a1, break
jal SMSG_R
sri $a1, jalnet
jal SMSG_R
wait:
nop
j wait
complete:
ori $a0, $0, 55
ori $a1, $0, 55
ori $a2, $0, xFF
jal plotpixel
break
```

The only thing to remember about this is that the ID of the other CPUs must be initialized in their start up ROM which consists of a SID instruction a few NOPs and a jump instruction. Essentially the same wait loop that we see occurring in the Master code.

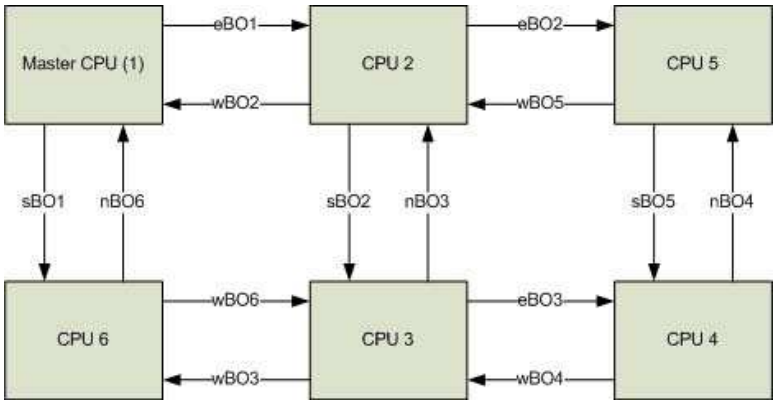

Figure 18: 6 Core System

The last thing to test was the message passing algorithm. This was verified in the system designed shown in Figure 18. The program was the same as the program above in the Master CPU example with two cores. However the messages were routed through intermediate nodes as well. The

master CPU(1) would send to CPU 4 across the network. It was then seen that CPU 4 would hit the completion of the program as well as that the Master CPU would then reach the break instruction as well and a pixel was seen on the screen meaning that the whole system worked correctly. This means that the system worked 100% all the while utilizing the message passing algorithm on the FPGA.

The network optimization component was disabled for these tests since there was no way to verify it and it would cause some bugs during boot load. However I felt that the simulation results of the network optimizations were validation enough to prove the concept behind this project and given another small period of time it could easily be verified on the FPGA as well.

On a whole I think that I verified and validated all sections of the design. The only thing not fully implemented in hardware was the message queue for the network layer. It was realized in the simulations that if there exist two message waves on the network at a given time the intersection of these message crests will result in strange behavior. Considering the self-timed nature of the network layer it would not be a very desirable interference. I thought it would not be necessary to implement this message queue since the system could be proven without it and only using one message at a time since I never wrote any "recursive" programs that would utilize a deeper convention than the Master CPU method. However, it would be rather simple to integrate a message queue into each bus of the network layer with minimal complexity.

5 Conclusion

On the whole I believe that I succeeded at achieving the overall goal of this project as well as a few others. I managed to develop a complete computer system all the way from the architecture to the software. Although this was never implemented in layout it was implemented on an FPGA which proved that the design was validated both in behavior and timing analyses. Also I succeeded at designing a scalable multiprocessor architecture which uses a new approach to multiprocessing as well as a new style of network which is capable of self optimization over time. Although it has been a time issue to fully validate all of the results of this project it can be clearly seen that it successfully showed the viability of these new approaches and demonstrated their correct functionality and advantages. Whether or not

this design may be applicable for larger applications is not clear but given a larger resource base and investment of time this project can provide a new insight to multiprocessor architectures and their networking structures.

I would like to mention a few notes here about future work for this project. In the above report I make mention of a good amount of improvements and future work that could be done to make the system run better as whole. Also, however, it would be possible to design network nodes that are not connected to CPUs and design more complex network designs for a multicore system. It would be great to see more work done on this project and I aim to continue exploring these concepts in the future.

6 Acknowledgments

I would like to acknowledge a few sources for this project. First of all I would like to credit Altera and Terasic for producing the Cyclone II FPGA and DE2 board respectively. I would like to thank Cornell University for giving me the facilities and the resources to work on this project. I would like to thank my Advisor and Professor Bruce Land for his continued support of me and my project and for his great advice throughout my Cornell career. I would also like to acknowledge Chris Fraser and Dave Hanson for providing support regarding general LCC issues. I would also like to dedicate this work to my grandparents whom were forced to experience the worst and persevered so that my family could pursue happiness.

7 Appendix A: Visual Nodes, Project Simulation

This master's project consisted of three semesters. The first semester was worth 2 credits while the second two worth 4 each for a total of 10 credits. In the first semester I wanted to make sure that the design would work when developed so that half way through development it would be seen that there is no light at the end of the tunnel. One of the things I did was design a simulation of the nodal network in C++ and demonstrate how messages can be propagated through the network with a wave like nature. Screen shots of this program are shown above in the section dealing with message passing (Section 2.2.1). However it is not obvious that the design of this program

was largely complex as well implementing what ended up being a quadruple-linked list data structure and then displayed using an OpenGL framework.

7.1 Quad-List

The quad-list data structure is created using a pair of classes. The cpuNet class creates the actual data structure while the cpuNode class provides the actual functionality of the node such as sending, receiving, and passing messages. Since this is a quad list where each node has a pointer to the top, bottom, left, and right nodes respective to it the creation of the structure is non-trivial. In fact, while if one was to create a normal linked list of length N the efficiency would be $O(n)$ but the quad-list has an efficiency of $O(n^2)$.

This is because the nodes can only be created in a sequential way and it would be impossible to link nodes that have not yet been created. The only way then to create the data structure is to sequentially create the nodes and link the top and left node since the creation follows the arbitrary convention that the list is created top to bottom and left to right. Then once the list has been created we go back to the top left node of the list and parse through the list now completing the bottom and right node links.

The other purpose the cpuNet class has is also to render the list which will parse through the quad-list with efficiency $O(n)$ since it is now viewed as a sequential list. It will then will set the node's color based on it's internal state and then call the node's render function. Also the cpuNet class has the very important task of updating the list. This will parse through the list and "fire" each node. This function will call the cpuNode function for that node and based on the state of it's internal message queue it will either send messages to other CPUs or not. Only after all nodes have been fired does this function change the state of the node so that we do not have the whole net sequentially fire. This parallelism is enforced through an ALTERED state which designates that the node has been updated during this cycle.

The cpuNet class also has functions to get nodes and remove nodes. These functions are very useful for the purpose of grabbing a node and sending a message out of it. This is exactly what is happening in the InitGL() function in the top-level visualNodes.cpp. First the network is being created with a certain dimension. This dimension is one dimensional so the actual number of nodes the square of this number. Then we use the getNode() functions to grab a few nodes and then send messages out on them with destination IDs and colors representing different data to demonstrate how overlapping

message wave crests are preserved.

The network is rendered during the DrawGLScene() function which should be called every time through the main program loop. However the updateNet() function is only called in two situations. If the user presses the 'n' key then the network is updated one time. The other option is if the user presses 'F2' then a flag is flipped allowing an update every delay period.

The rest of the functionality lies within the cpuNode class. This class enables the actual message passing functionality. The first thing that is done is sendMsg() is called with a destination ID and "data". This overloaded function will then send out the message in all directions. Then at the next time that updateNet() is called the nodes with the message will run the q_fireNode() function which will then run through the states and resolve the ALTERED states. It will check to see if this node is the destination of the message if not it will call sendMsg() with the message which then implements the message passing algorithm so that fireNode() is blind to the algorithm. It will create a state machine for resolved messages so that they do not respond immediately but wait a few time steps. All of the message manipulation is done through a queue owned by the cpuNode so that this may happen for nodes with multiple intersecting messages.

8 Appendix B: NDMA Test Programs

This example NDMA Test Program was designed to truly test all facets of the functionality of the single core system. This is a simple implementation of the game PONG in which the keys 'a', and 'z' on the PS2 keyboard will move the left pong paddle up and down respectively where 'j' and 'm' will do the same for the right paddle. This program really gives the system a through test since it requires every possible branch condition, a large amount of non-blocking input from the keyboard as well as string oriented functionality.

```
/* Fourth NDMA test program
 *
 * Build Instructions
 * rcc -target=ndma/ndmaOS ndmaTest4.c
 *
 * Note: LCC doesn't handle single line comments apparently
 *
```

```
 * PONG!
 *
*/

void plotpixel(int x, int y, char color);
void drawchar(int x, int y, char c, char color);
void drawline(int x1, int y1, int x2, int y2, char color);
char getchar();
int waitchar();

void main(void)
{
    char Lose[] = "You Lose!";
    int cLose = 9;
    char Win[] = "You Win!";
    int cWin = 8;

    int i = 0;
    int player1_y = 200;
    int player1_inc = 0;
    int player1_pts = 0;
    int player2_y = 200;
    int player2_inc = 0;
    int player2_pts = 0;
    char c;
    int ball_x = 250;
    int ball_y = 200;

    int ball_xi = 1;
    int ball_yi = 1;

    int k = 0;

    int fDone = 0;

    drawchar(31, 440, player1_pts + '0', (char)0xFF);
    drawchar(591, 440, player2_pts + '0', (char)0xFF);
```

```
while(fDone == 0)
{
    if(waitchar() == 1)
    {
        c = getchar();
        if(c == 'a') player1_inc--;
        else if(c == 'z') player1_inc++;
        if(c == 'j') player2_inc--;
        else if(c == 'm') player2_inc++;
    }

    /* Draw paddles */
    for(i = 0; i < 21; i++ ) plotpixel(30, player1_y + (i - 10), (char)0xFF
    for(i = 0; i < 21; i++ ) plotpixel(590, player2_y + (i - 10), (char)0xF

    /* Draw Ball */
    plotpixel(ball_x, ball_y, (char)0xFF);
    plotpixel(ball_x+1, ball_y, (char)0xFF);
    plotpixel(ball_x, ball_y+1, (char)0xFF);
    plotpixel(ball_x+1, ball_y+1, (char)0xFF);

    for(k=0; k<5000; k++);

    plotpixel(ball_x, ball_y, (char)0x00);
    plotpixel(ball_x+1, ball_y, (char)0x00);
    plotpixel(ball_x, ball_y+1, (char)0x00);
    plotpixel(ball_x+1, ball_y+1, (char)0x00);
    if(player1_inc != 0)
        for(i = 0; i < 21; i++ )
            plotpixel(30, player1_y + (i - 10), (char)0x00);
    if(player2_inc != 0)
        for(i = 0; i < 21; i++ )
            plotpixel(590, player2_y + (i - 10), (char)0x00);

    /* update ball   */
    ball_x += ball_xi;
    ball_y += ball_yi;
    player1_y += player1_inc;
```

```
        player2_y += player2_inc;

        if(ball_x == 30)
        {
            if(ball_y < player1_y + 10 && ball_y > player1_y - 10) ball_xi = 1;
        }
        else if(ball_x == 590)
        {
            if(ball_y < player2_y + 10 && ball_y > player2_y - 10) ball_xi = -1
        }
        else if(ball_x == 600)
        {
            drawchar(31, 440, player1_pts + '0', (char)0x00);
            ball_x = 250;
            player1_pts++;
            drawchar(31, 440, player1_pts + '0', (char)0xFF);
            if(player1_pts == 9) fDone = 1;
        }
        else if(ball_x == 10)
        {
            drawchar(591, 440, player2_pts + '0', (char)0x00);
            ball_x = 250;
            player2_pts++;
            drawchar(591, 440, player2_pts + '0', (char)0xFF);
            if(player2_pts == 9) fDone = 2;
        }

        if(ball_y > 440 || ball_y < 20)
        {
            if(ball_yi == 1) ball_yi = -1;
            else ball_yi = 1;
            ball_y += ball_yi;
        }
    }
    if(fDone == 2)
    {
        /* You Lose */
        for(i = 0; i < cLose; i++)
```

```
        {
            drawchar(300 + (i << 3), 200, Lose[i], (char)0xFF);
        }
    }
    else if(fDone == 1)
    {
        /* You win */
        for(i = 0; i < cWin; i++)
        {
            drawchar(100 + (i << 3), 200, Win[i], (char)0xFF);
        }
    }
}
```

This example Master CPU NDMA program was used to test both the two and six core NDMA systems. It integrates the GPU and also demonstrates the message passing functionality of the system. This also shows the GPU and PS2 API as well as the NDMA API.

```
.SetStack 1023
.Boot main
# GPUAPI.asm defines the different functions for manipulation of the
# GPU interface through the in/out interface on the NDMA CPU

# NOTE: Connection to the GPU interface will be through one register input port
# and one register output port with the following conventions:
# Input
# Bit 0 : Complete Flag
#
# Output
# Bit 15-0 : Instruction
# Bit 16 : Valid Flag

# PlotPixel (ppix)
# X - $a0
# Y - $a1
# Color - $a2
plotpixel:
```

```
addi $t0, $0, x0
add $t1, $t0, $a0
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

ppix_while_x:
in $t2, $1
beq $t2, $0, ppix_while_x

out $5, $0

addi $t0, $0, x0400
add $t1, $t0, $a1
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

ppix_while_y:
in $t2, $1
beq $t2, $0, ppix_while_y

out $5, $0

addi $t0, $0, x1C00
add $t1, $t0, $a2
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

ppix_while_pixel:
in $t2, $1
beq $t2, $0, ppix_while_pixel
jr $ra

# Draw Char (dchar)
# X - $a0
# Y - $a1
```

```
# Char - $a2
# Color - $a3
drawchar:
addi $t0, $0, x0
add $t1, $t0, $a0
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

dchar_while_x:
in $t2, $1
beq $t2, $0, dchar_while_x

out $5, $0

addi $t0, $0, x0400
add $t1, $t0, $a1
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

dchar_while_y:
in $t2, $1
beq $t2, $0, dchar_while_y

out $5, $0

addi $t0, $0, x1000
add $t1, $t0, $a2
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

dchar_while_char:
in $t2, $1
beq $t2, $0, dchar_while_char

out $5, $0
```

```
addi $t0, $0, x1400
add $t1, $t0, $a3
lui $t2, 1
add $t1, $t1, $t2
out $5, $t1

dchar_while_draw:
in $t2, $1
beq $t2, $0, dchar_while_draw
jr $ra

# Wait Char (waitchar)
# Returns 1 if char is waiting
# return 0 if not result in $v0
waitchar:
in $v0, $2
srl $v0, $v0, 9
xori $v0, $v0, 1
jr $ra

# Get Char (getchar)
# Waits until character written to PS2 Buffer
# then saves it into $v0 ($2)
# *char - $a0

getchar:
in $t1, $2
srl $t1, $t1, 9
bne $t1, $0, getchar

outi $6, 1

gchar_complete:
in $t1, $2
srl $t1, $t1, 8
andi $t1, $t1, 1
beq $t1, $0, gchar_complete
```

```
in $v0, $2
andi $v0, $v0, xFF

out $6, $0
jr $ra

# Network Driven API
# Implements quick function calls for bcstr and smsgr
# input instruction to be set is in $a0 which should
# be initialized with the sri pseudo instruction

# Broadcast Register
BCST_R:
bcstr $a0, 3
bcstr $a0, 2
bcstr $a0, 1
bcstr $a0, 0
jr $ra

# Broadcast Register Register
# This broadcasts a broadcast from the destination
# CPU ID which will then in turn broadcast the instruction
# in $a0
# Instruction to send in $a0
BCST_RR:
sri $t0, bcst 0

# Send MSByte in bcst msg
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 3

# Send Second Byte
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
```

```
bcstr $a0, 2

# Send Third Byte
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 1

# Send Last Byte
bcstr $t0, 3
bcstr $t0, 2
bcstr $t0, 1
bcstr $a0, 0
jr $ra

# Send Message Register
# Destination ID - $a0
# Register to send - $a1
SMSG_R:
smsgr $a0, $a1, 3
smsgr $a0, $a1, 2
smsgr $a0, $a1, 1
smsgr $a0, $a1, 0
jr $ra

# Send Message Register Register
# Target Destination ID in $a0
# Final Destination ID in $a1
# Instruction to send in $a2
SMSG_RR:
sri $t0, smsg $a1, 0

# Send MSByte in bcst msg
smsgr $a0, $t0, 3
smsgr $a0, $t0, 2
smsgr $a0, $t0, 1
smsgr $a0, $a2, 3
```

```
# Send Second Byte
smsgr $a0, $t0, 3
smsgr $a0, $t0, 2
smsgr $a0, $t0, 1
smsgr $a0, $a2, 2

# Send Third Byte
smsgr $a0, $t0, 3
smsgr $a0, $t0, 2
smsgr $a0, $t0, 1
smsgr $a0, $a2, 1

# Send Last Byte
smsgr $a0, $t0, 3
smsgr $a0, $t0, 2
smsgr $a0, $t0, 1
smsgr $a0, $a2, 0
jr $ra

main:
sid 1
nop
# Dispatch Code:
# ori $t0, $0, 0
# ori $t1, $0, 170
# addi $t0, $t0, 1
# bne $t0, $t1, -2
# bcst: snip 511
# bcst: j complete
# bcst: jalnet
# break

ori $a0, $0, 4
sri $a1, snip 42
jal SMSG_R

sri $a1, ori $t0, $0, 0
jal SMSG_R
```

```
sri $a1, ori $t1, $0, 170
jal SMSG_R

sri $a1, addi $t0, $t0, 1
jal SMSG_R

sri $a1, bne $t0, $t1, -2
jal SMSG_R


ori $a1, $0, 1
sri $a2, snip 200
jal SMSG_RR

sri $a2, j complete
jal SMSG_RR

sri $a2, jalnet
jal SMSG_RR

sri $a1, break
jal SMSG_R

sri $a1, jalnet
jal SMSG_R

# wait loop
wait:
nop
j wait

complete:
# plot a pixel
ori $a0, $0, 55
ori $a1, $0, 55
ori $a2, $0, xFF
jal plotpixel
```

```
break
```

9 Appendix C: NDMA ISA

Instruction	Description	Instruction Status
	Memory Operations	
lw	Load Word	yes
lh	Load half word	yes
lhu	Load half word unsigned	yes
lb	Load byte	yes
lbu	Load byte unsigned	yes
sw	Store word	yes
sh	Store half word	yes
sb	Store byte	yes

Table 1: NDMA Instruction Set Supported Memory Instructions

	Immediate Ops	
addi	Add immediate	yes
addiu	Add immediate unsigned	yes
lui	Load upper immediate	yes
slti	Set less than immediate	yes
sltiu	Set less than immediate unsigned	yes
andi	And immediate	yes
xori	Exclusive OR immediate	yes
ori	OR immediate	yes

Table 2: NDMA Instruction Set Supported Immediate Instructions

	Network Instructions	
smsg	Send message to CPU	yes
smsgr	Send Register Message to CPU	yes
bcst	Broadcast Message	yes
bcstr	Broadcast Register Message	yes
jalnet	Jump to network pointer	yes
snip	Set Network Instruction Pointer	yes
ndjr	Network Driven Jump Register	yes
rmsg	Receive message from CPU	no
sid	Set processor ID	yes

Table 3: NDMA Instruction Set Supported Network Instructions

	Register Instructions	
add	Register add	yes
addu	Unsigned register add	yes
sub	Register subtract	yes
subu	Unsigned register subtract	yes
and	Register AND	yes
or	Register OR	yes
xor	Register XOR	yes
slt	Register set less than	yes
sltu	Unsigned register set less than	yes
sll	Shift left logical	yes
srl	Shift right logical	yes
sra	Shift right arithmetic	yes

Table 4: NDMA Instruction Set Supported Register Instructions

	Branch Instructions	
bne	Branch on not equal	yes
beq	Branch on equal	yes
bgtz	Branch on greater than zero	yes
bgez	Branch on greater than or equal to zero	yes
bltz	Branch on less than zero	yes
blez	Branch on less than or equal to zero	yes
j	Jump	yes
jal	Jump and link	yes
jr	Jump to register	yes

Table 5: NDMA Instruction Set Supported Branch Instructions

	In / Out Instructions	
in	Read input from port	yes
outi	Output immediate to port	yes
out	Output register to port	yes

Table 6: NDMA Instruction Set Supported In/Out Instructions

	Mult/Div/Mod	
mult	Register multiply	yes, but disabled
multu	Unsigned register multiply	yes, but disabled
div	Register divide	yes, but disabled
divu	Unsigned register divide	yes, but disabled
mod	Register modulus	yes, but disabled

Table 7: NDMA Instruction Set Supported Mult/Div/Mod Instructions

	Control Operations	
nop	No Operation	yes
break	Temporary Break	yes

Table 8: NDMA Instruction Set Supported Control Instructions

10 Appendix D: Index of Figures

List of Figures

11 Appendix E: Project Facts and Statistics

This project was done in a year and a half through the Masters of Engineering program at Cornell University. This section is a quick little interesting piece of information regarding my experience working on this project.

Number of Semesters: 3
Total Number of Academic Project Credits: 10

Total Number of Lines of Code: ~13500
Verilog: ~7500
C/C++: ~6264

Number of Computer Malfunctions / Crashes Requiring a new install of Windows or a new computer: 4

Number of States Project Was Worked On In: 4