AN ABSTRACT OF THE THESIS OF

Steen K. Larsen for the degree of Master of Science in Electrical and Computer Engineering presented on January 13 1999. Title: Master/Slave Parallel Processing.

Abstract approved:

Redacted for Privacy

James Herzog

An 8 bit microcontroller slave unit was designed, constructed, and tested to demonstrate advantages and feasibility of master/slave parallel processing using conventional processors and relatively slow inter-processor communications. An 8 bit ISA bus controlled by an 80X86 is interfaced to a logic block that controls data flow to and from the slave processors. The slave processors retrieve tasks sent by the master processor and once completed, return results to the master that are buffered for the master's retrieval. The task message sent to the slave processors has task description and task parameters. The master has access to the bi-directional buffer and a status byte for each slave processor. Considerable effort is made to allow the hardware and software architecture to be expandable such that the general design could be used on different master/slave targets. Attention is also given to cost effective solutions such that development and possible market production can be considered.

© Copyright by Steen K. Larsen
January 13 1999
All Rights Reserved

Master/Slave Parallel Processing

by

Steen K. Larsen

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented January 13 1999
Commencement June 1999

Master of Science, Computer Engineering thesis of Steen K Larsen presented on January 13 1999.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head or Chair of Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Steen K Larsen, Author

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Table Page

DEDICATION

To Mom and Dad for their encouragement.

PREFACE

This paper shows a method of implementing master/slave processing. In the interest of implementing a prototype, a low cost, off-the-shelf approach was taken. A large part of the work was finding how the tools and resources available could be used to implement the goal.

There are various reasons for this pursuit. One of which is that as an undergraduate I was captivated by Mandelbrot and other fractal image generation. The idea of offloading the compute intensive work from the 25MHz 80386 processor onto slave processors for a faster solution would reduce the generation time. This idea also held for ray tracing images. Another was that one of the OSU professors, James VanVecten, was using 25MHz 80386 PCs to study gold atom implants in silicon. His research assistant would take control of 20 PCs and start jobs with different parameters that would run overnight. VanVecten's position was that this effort was similar to access to a costly supercomputer. This manner could also be compared to the recent Internet activity in Mersenne prime search. This effort uses volunteers who take certain number ranges and use their computing resources to eliminate existence of Mersenne primes within those ranges.

These efforts, along with the expandability of the PC (many older PCs have 7 ISA slots, of which perhaps one or two are used) contributed to the idea of putting the expandability to good use in a cost-effective manner.

1 Introduction

1.1 Research Objectives

This work demonstrates a method to implement master/slave parallel programming using conventional off-the-shelf components and tools. Essentially the master processor sends commands to the slave processors and retrieves results. Parallelism is achieved by the fact that the slave processors can be executing tasks simultaneously. Off-the-shelf components and tools are commercially available and allow prototypes and improvements to be implemented economically, quickly and easily.

There are various goals that are attempted in this effort as described below.

1.2 Motivation

The main purpose is to find a manner of increased performance of certain tasks by dividing the task into parts that can be done at the same time. In this master/slave implementation, only certain tasks make sense to be divided, such as the calculation of π. In addition, other purposes exist such as cost, flexibility, and ease of use as described below.

1.2.1 Performance Improvement

There will be tradeoffs between doing a task in a conventional uniprocessor manner and a task split in a single master multiple slave scenarios. A uniprocessor is better suited to heavily interconnected tasks such as software compilers or user interaction. A master/slave configuration sacrifices the overhead used in interprocessor communication with the ability to do multiple tasks simultaneously. This method is

advantageous where interprocessor communication is limited, where task granularity is large. This paper takes the task of computing digits of π and comparing performance of a uniprocessor and multiprocessor configuration.

1.2.2 Flexible Resources

In the interest of prototyping the design and using physical hardware and software to compare results, standard off-the-shelf components and tools were used. This approach certainly does not optimize comparison of the master/slave configuration since currently powerful uniprocessor systems are inexpensive and depending on the task under comparison, different slave processors would be needed. A major advantage is that a commonly used hardware solution allows for easy extension of the solution to more complex and specialized solutions based on the same common architecture.

1.2.3 Flexible Slave Processor

A popular 8051 based controller is used which allows for extension into either embedded controllers with added features or more fully standalone processors that are designed for more general purposes. This is discussed in more detail in the slave processor section.

1.2.4 Links to High Level Software

Specific emphasis is made to ensure links to upper level software that will allow general applications to make use of this hardware configuration. This allows for portability of the hardware to different tasks within the same physical design constraints (no hardware modification) or extension of the hardware (i.e. addition of an analog to digital converter to the slave processor)

1.2.5 Cost Considerations

For a product to succeed in the marketplace, cost is always a concern. The prototype is implemented with this in mind. The expandable aspect of the design allows for faster slave processors and faster bus interaction. This would require expensive and time consuming implementation techniques (wire wrapped connections would not be an option) and also expensive development tools would be needed, such as faster FIFOs for faster bus interaction.

1.3 Chapter Organization

The first two chapters are an overview of the thesis, discussing objectives, literature sources and general implementation. The following four chapters are on the specific details about the slave processor, bus interface logic, master processor interface, and the pi digit calculation example. The concluding three chapters discuss the benefits, disadvantages, and conclusions of the implementation.

1.4 Literature Review and Credits

Very little literature is available on this particular topic of parallel processing. Much of the literature available is on symmetric processors defined by a common bus or crossbar switch where all the processors are identical and have supporting software that utilize each with equanimity. Another form that can be described as tightly coupled parallel processing can be found in platforms that use parallel processors for emulation of instructions on the "slave" processor. This is demonstrated with x86 processors in Sun Sparc or Apple Mac platforms used to increase the speed of x86 applications.

Significant research and design has gone into task specific master/slave processing such as video controllers. This is of little use since one of the goals is to allow

for multiple slaves to be added to the master control. With this in mind, there are documentation and resources that are useful in defining and understanding the interfaces to the master and slave processors as discussed below.

Eggebrecht [8] gives a practical explanation of the ISA bus functionality and example implementations interfacing it. This was useful in programming the Altera 10K20 logic. The FIFOs in the Altera glue logic (and device characteristics, pinouts, programming requirements) are described in the Altera databook [1] and on www.altera.com.

The slave processor, an Atmel 89C52 8051 variant, is described in the Atmel data book [3], and the Keil C Compiler User's Guide describes how to program the 89C52 in C. An example C source listing of the pi hexadecimal digit calculation is on http://www.mathsoft.com/asolve/plouffe/plouffe.html.

On the master processor, MS Visual C++ V1.52 was used with aid from the online C++ manual, and Mueller's book [13]. Special thanks to David Frame for providing the embeddable assembly commands to access the IO ports from MS Visual C++, see appendix C.

2 General Implementation

This chapter provides a top-level description of the major components: communication model, master processor, slave processor, and glue logic. Additionally, previous efforts are discussed which describe advantages of the current implementation. Finally, a limited comparison is made to other parallel processing implementations.

The term master/slave is used instead of server/client to describe this architecture. This is because server/client is too generic a description. The term server/client tends to allow the client to do various tasks if the server is not immediately available. An example is a network with a server for document files and another server for database transactions. The networked client could work on the database if there were no tasks from the document server. The master/slave configuration has the slave processor more closely tied to the master processor such that the slave processor idles when there are no tasks issued. The slave is less flexible in that although it can do different tasks, it cannot multitask between them and executes tasks sequentially.

Figure 1 is a block diagram of master/slave processor configuration implemented. The arrows represent the flow of data. In this case the ISA bus is a bi-directional 8 bit bus and the slave processors have two distinct 8 bit channels to communicate with the glue logic. The glue logic incorporates the ISA interface and FIFO buffering that enables communication between the master and slave processors.

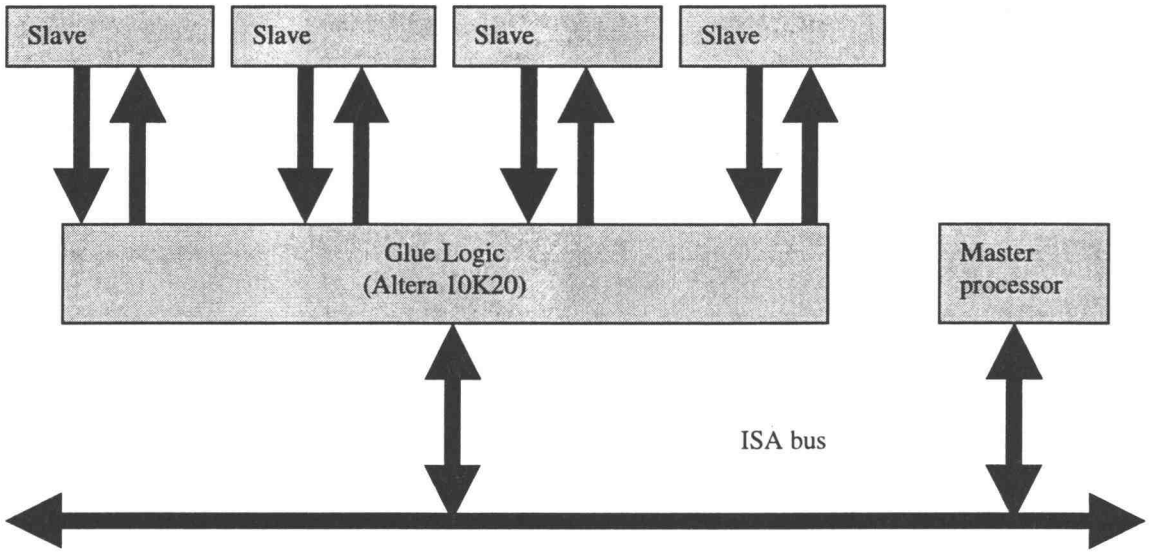

Fig 1 Master/slave block diagram

Figure 2 is a photograph of the prototype built to run program tests of this configuration. Wirewrapped connections between the chips exist on the other side of the protoboard. To run, the card is inserted into an ISA slot, and after programming the Altera glue logic chip, and resetting the slave processors, the slave processors are ready to execute commands from the master processor.

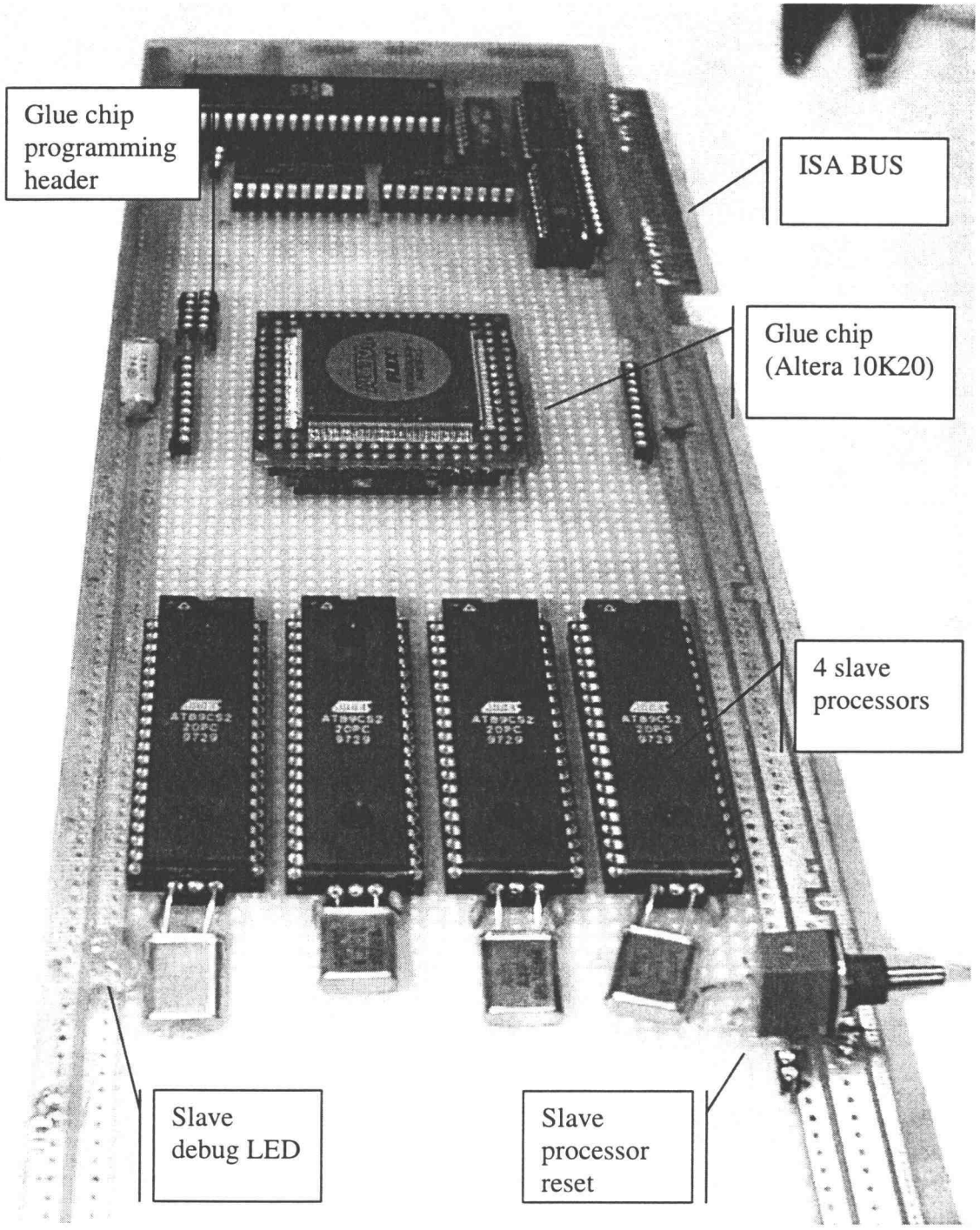

Fig 2 Prototype photograph

2.1 Communication Model

The basic mode of operation is that the master processor sends a one byte command to the idle slave processor defining which type of task is to be done. The task type defines the task and type of parameters that will be following and can be as simple as adding two 8 bit values or calculating a specified digit of pi. Once the slave processor knows the type of task, the program switches to the appropriate subroutine and after the appropriate parameters are gathered, executes and returns results to the FIFO.

The master processor controls the access on the ISA bus. This is typically an x86 processor with access to the IO ports that are used to communicate to the slave processors. The IO ports function as status and communication channels to the slave processors. These are implemented in an FPGA that interfaces to the slave processor as well.

2.1.1 Slave Processor Implementation

The Atmel 89C52 was used for various reasons. The FLASH 8Kbyte code area makes the prototype program easy to do code modifications. The simple 8 bit interface allows the same data format as the 8 bit ISA interface used. Eight pins are used for the input data from the ISA bus and another 8 pins are used for the output data. Although this could be compressed into a total of 8 bi-directional pins, the 16 pin implementation allows for the FIR/IIR filter option where separate input and output data paths are beneficial. Control signals are used to implement reads and writes to the ISA interface. Additional signals are used to indicate incoming ISA commands/parameters, and slave processor status to the status IO port.

The code is structured in an infinite loop (see figure 5). When a command from the master is detected, the main loop switches to the task that is read. Based on the task,

the appropriate parameters are read and then executed. Resulting output is sent to the output FIFO in the FPGA.

2.1.2 FPGA Implementation

An Altera 10K20 with 208 pins is used to interface between the ISA bus and the slave processor. Since it is SRAM based, it is a fast way to prototype logic changes. The configuration can be downloaded in-circuit in less than five seconds through a serial interface that is isolated through an LS244 buffer connected to the PC programmer's parallel port.

The majority of the logic cells used are the FIFO for buffering between the slave and the ISA bus. Since this FPGA supports four slave processors there are subsequently four FIFO blocks that have associated logic to decode reads and writes to the FIFO. In addition to the FIFO there is a register that is written by the slave processor to provide status to the ISA bus. Both the FIFO and the status register appear as IO ports in the standard PC memory address space.

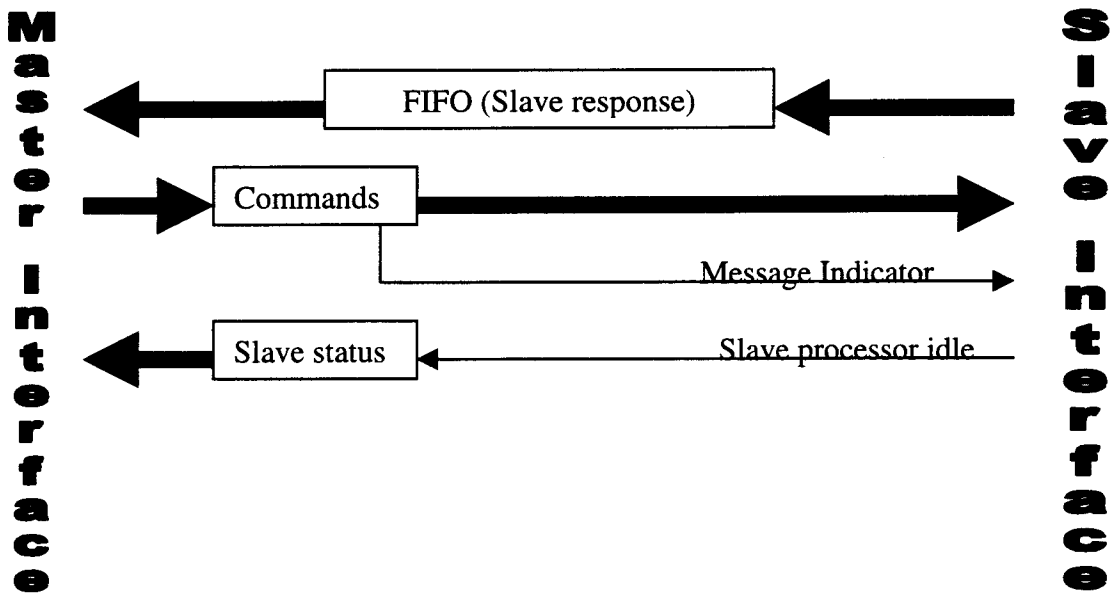

Fig 3 FPGA block diagram for one slave processor

The FIFO IO port address is writeable by the ISA bus and is used to pass commands and parameters from the master processor. This is implemented in the FPGA as an 8 bit register. The slave processor knows there is an incoming message byte by monitoring a message indicator that gets cleared when the slave processor reads the incoming message byte. An example is if the slave is in the idle loop, it will poll (wait) on the interrupt of the message indicator signal. Upon assertion, the processor reads the command byte, and if parameters are required for the task, waits until they are delivered before processing and generating the task output.

2.1.3 Master Processor Implementation

The master processor on the ISA bus has access to two IO ports for each slave processor. One port is used to pass commands and parameters to the slave processor.

This same port, when read, returns the results of tasks given to the slave processor. The second port for each slave processor defines the state of the slave processor. This allows the master processor to know when to pass more commands to the slave processor.

2.2 Pre-Prototype History

The eventual incarnation of the prototype was the result of 3 major changes that occurred over the past two years. The reasons for the changes show how different options were considered and rejected based on the objectives described previously in the motivation section.

2.2.1 Motorola 68302

The initial version of the slave processor was intended to be the Motorola 68302 and was selected for the wide range of options and expandability available to the processor. This proved to be too complex in the 132 pin PGA package that required external RAM and ROM for each slave processor. The goal of the thesis was not to explore a large number of possible applications, but to demonstrate how a master/slave processor architecture could be implemented. The CrossCode C compiler available did not easily allow port pin toggling on the 68302. This function is very useful allowing pin control to be mixed with standard C functions and was another reason to use a different processor.

2.2.2 Philips 87C576

After being exposed to the widespread selection of Intel 8051 variants of processors, the 87C576 option was explored. This has a built-in ISA interface such that external decoding would not be needed. This seemed to be an optimal solution until price (~$100 for a UV erasable part) and availability (8 week lead time) were determined. The

high cost is a function of volume and manufacturability since the UV-erasable window needs to be sealed properly. Since this was not an option, the Atmel 89C52 controller (8Kb/256byte) selection was made at the cost of $10 per part.

2.2.3 Discrete Glue Logic

To initially test the ISA interface HCT374s (8 bit edge triggered D type flip-flops) were used to test read/write capabilities as shown in figure 3. To test the 89C52, the 374s were connected to two port buses. Simple tasks, such as reading a written byte, were done to test the data path to and from the slave controller from the ISA bus. This implementation could be expanded to demonstrate master/slave processing, but the availability of Altera 10K20s allowed a much better way to prototype. The SRAM based FPGA and the fact that all pins went into the 208 pin part, easily allowed pin functions and logic to be changed.

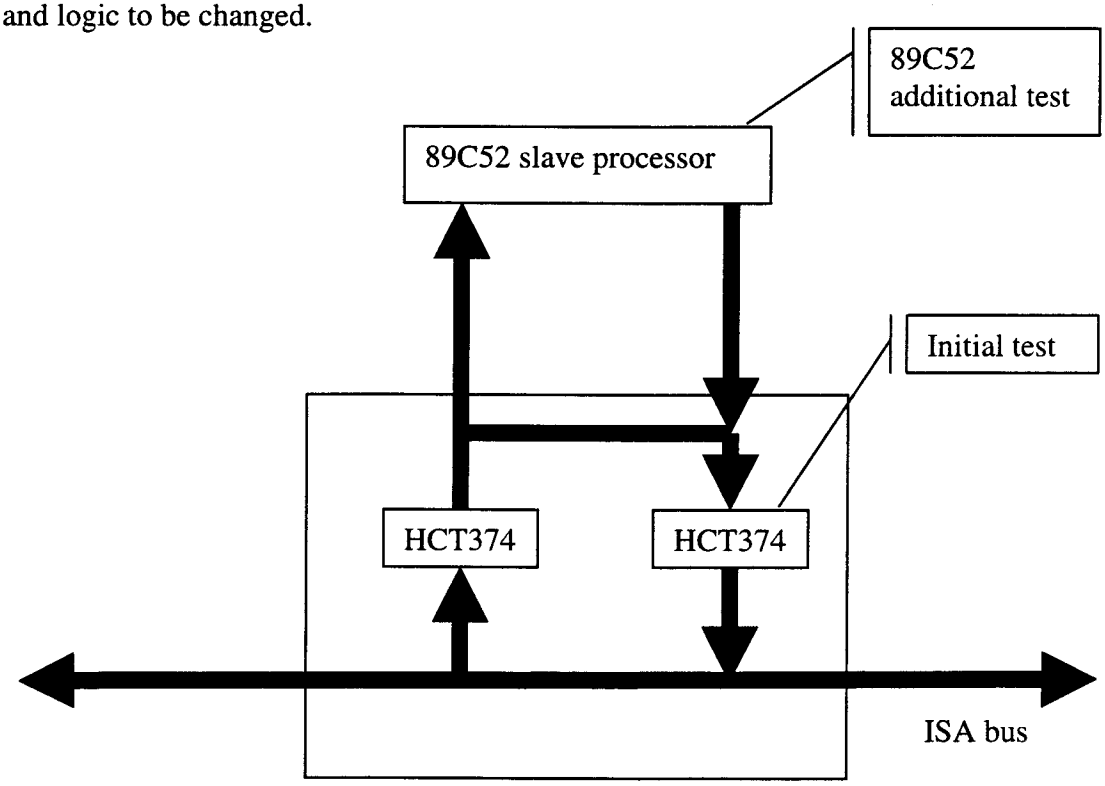

Fig 4 Prototype tests of the ISA bus

2.3 General Parallel Architecture Overview

As with uniprocessor cache-based serial systems, similar data movement principles apply for a high performance solution[9]:

- Locality of data reference
- Minimization of data movement

To achieve this goal, various different approaches are made.

The SIMD/MIMD classification of parallel architectures proposed by Flynn [2] divides most parallel architectures into two groups, Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). Considering more general-purpose architectures, solutions that address a wide variety of tasks, MIMD seems to be gaining in popularity. This is greatly aided by the availability of processors that are designed for symmetric parallel implementations such as the Intel Pentium Pro™ and Intel Pentium II Zeon™. The bus between the processors is pipelined such that processor A can be fetching a memory instruction while processor B snoops the cache and processor C retrieves a cache line from processor D. Currently bandwidth between four directly bussed processors and shared memory is supported as viable solutions. The primary reason more processors are not directly bussed is the tradeoff between fast bus speeds and bus signal integrity. One solution for more processors is to implement a bridge between two quad-processors and allow two separate busses that still share memory. This solution then drives the requirement for pipelined memory bus access such as that proposed by the Rambus corporation to satisfy the greater shared memory needs. Another solution for more processors is the Sequent NUMA-Q architecture by connecting quads with high-speed data busses. This sacrifices directly bussed processor architecture for more of a tree architecture. This solution is applicable for database

applications residing in long-term memory (hard disk drive) that is accessible by all processors.

Although symmetric bus parallel processing is gaining in popularity mainly due to the generic applications it addresses, there are still tasks where simple but effective slave processors are useful. As previously stated, one of the motivations is to provide a cost-effective solution. Instead of an expensive symmetric parallel architecture that addresses generic applications such as a multi-user database server, if the application can be addressed by a master/slave architecture, the solution could be less expensive.

Networks of workstations (NOW) have become popular with the use of Linux on multiple inexpensive computers with such projects as Beowulf, GAMMA, and PARMA.[6] The intent is to connect computers running a common operating system into groups connected by Ethernet. While the computers connect with the Ethernet protocol the TCP/IP stack is not used, but project related protocols that enhance the movement of data from one node to another. There is much flexibility in this configuration and the idea of harnessing unused personal computer cycles is appealing. This type of parallel configuration seems to be suited more for large granularity functions. There is a master process running on one of the nodes, which is analogous to the master processor. The disadvantage from the presented master/slave solution is that the master process needs to talk to each and every participant which could be several network "hops" away. With the master/slave implementation, the slaves are on a common bus that enables more direct access.

2.4 Previous Similar Architectures

Much of the previous research resulting in architectures similar to the master/slave implementation presented here has image processing as a common theme. Although the slave processors used are fairly weak in areas traditionally used in image processing [17] (i.e. floating point calculations), the flexibility of the slave processor used

allows for a wide degree of experimentation at the overall cost of performance. For this reason, the ILLIAC IV and PASM architectures were chosen as most appropriate to compare and contrast as similar master/slave parallel architectures.

2.4.1 ILLIAC IV

The ILLIAC IV was one of the pioneering 2-D mesh vector parallel SIMD (Single Instruction Multiple Data) architectures. It consisted of a supervisory computer system that would pass commands to the mesh of processing units. Each processing unit consisted of a processing element (PE) and processing memory (PM) and could connect to four neighboring processing units. The PM running at 25MHz could do a 64 bit floating-point multiply in 400ns and direct access to 8 megabits arranged in 2048 64 bit words. This was implemented on 210 PCB cards for the PE and 512 16-pin DIPs for the PM. [2]

The architecture being presented is similar in that there is a supervisory computer system, the x86 processor, that controls the function of the individual processing units, 8051 based controllers with 256 bytes RAM. (The RAM could be expanded by mapping memory elements in the Altera FPGA). The slave units are organized in a tree rather than a mesh with the root being the master processor. Since there are two 8 bit busses to each slave unit, this could be changed into a 1-D mesh with the master still accessing each and every slave unit individually through the Altera FPGA. This could be useful in the case of a multiple tap filter. For example, the incoming data stream would be averaged over three datapoints on the first processor. The second processor would multiply by two and the third would normalize the data stream.

In the previous example, only three processors were used which allowed the fourth processor to do an unrelated task. This is a flexibility not available in the ILLIAC IV, where all processing elements are required to do the SIMD task, and easily supported in the master/slave tree structure.

One of the learnings of the ILLIAC IV was quoted by Hord [10] "future systems will have modular configurations for improved problem matching and will be able to switch ailing PEs out and good PEs into the configuration all under software control" This is feasible to some extent in the master/slave tree structure. Should one slave processor fail, the SRAM based FPGA can be reprogrammed to allocate a different processor for the task. This would interrupt the task at hand, resetting the FPGA and slave processors, but would not require manual intervention.

2.4.2 PASM

The Partitionable SIMD/MIMD (PASM) designed at Purdue consists of a bus of microcontrollers acting as the master processors and each having a fixed set of processing elements which acts as the parallel computation unit(PCU). The PCU is configured in a circuit switched extra stage network. [15]

Although currently not organized as a multistage switching network, this could be done within the FPGA with no hardware change. Each slave processor would maintain the 8 bit input and 8 bit output, and when sending messages to another of the four slave processors with the target address indicated by the signals currently used as status signals. The FPGA would internally control the communication network, and with internal combinatorial delays of 0.6ns max and internal routing delays of 0.4ns max should not be a bottleneck for the 8MHz slave processor.

A principle feature of PASM is the ability to partition SIMD and MIMD tasks. In the prototype version 16 Motorola 68010 processing elements each with 256KB are controlled by 4 master processors, also Motorola 68010 based. [2]

This is similar to the master/slave configuration presented, though with only one master 80x86-based master. Multiple SIMD parallel tasks can be done as in the instance of computing digits of pi. Each of a given group of slave processors can be given a section of calculation for a given digit of pi and after the calculation is done, return the

result. Concurrently, another group of slave processors can be doing floating point multiplications. Multiple MIMD tasks can be accomplished by extending the previously described multiple SIMD example. The processor group calculating pi digits, will have some slave processors that complete faster than others. The ones that complete faster can be assigned new pi digit calculation parameters. (This assignment can be done directly by the master processor, or a task FIFO could be implemented in the FPGA). Likewise, the slave processors calculating floating point multiplications can be tasked with new parameters as each individual multiplication is finished. Multiple MIMD can be accomplished since each slave processor has individual code and memory space. This allows the slave processor to be addressed individually and with individual tasks. The FPGA could be modified to allow for the more traditional SIMD implementation where tasks are broadcast to the slave processors. In this case, a task such as a 4x1 matrix multiplication would be done where each slave processor took a predefined segment of the task. This implementation would decrease the time required to pass tasks to the slave processors, but would inhibit the ability to do multiple SIMD/MIMD tasks.

3 Slave Processor Definition

The Atmel 89C52 was selected as a slave processor for a variety of reasons. After discussing the suitability and requirements, the C compiler and code is described.

3.1 Suitability of Atmel 89C52

One primary advantage of the 89C52 is the 8KB FLASH reprogrammable memory. FLASH technology is a method of nonvolatile storage that in the Atmel controller implementation can sustain 1,000 erasures. This allows rapid prototyping to be done without waiting for UV erasure or using new one-time-programmable (OTP) parts. 8KB is well within the program boundaries since the code consumes 1253 bytes and allows for expandability. The reason the C code compiles into such a small size is that the code itself is simple and does not use complex functions such as printf().

Another major advantage is the 8051 core processor, which is a commonly used controller. The widespread use of this processor has made available such things as C compilers and sample code easily available on the web. Also there are dozens of variants that allow hardware modifications such as adding A/D channels, additional IO ports, UARTs, EEPROM, as well as the conventional footprint, temperature, and speed selection. This selection allows for straightforward changes should the target application change.

The 89C52 comes in flavors up to 24MHz. Operation is fully static such that any operation from 0+Hz to the max specified rating is supported. This is useful when using the onboard RS-232 UART allowing a multiple of the baud rate to be used as a crystal. Since the prototype is wirewrapped with sockets, an 8MHz crystal is used to keep potential wire noise problems at a minimum.

Cost for the 24MHz commercial temperature 40 pin DIP package version is $10.00. This cost is much more reasonable than a multiple package option such as the 68302 described above or the UV erasable 87C576 with its more expensive packaging. Pricing for these processors can be as low as $1.50 in masked parts.

Pin count is a major issue with wirewrapping. Since the RAM/ROM is internal to the processor, the 40-pin DIP package is suitable for the IO signals that are used for ISA communications and status definition.

3.2 Slave Processor External Interface

Below is the pin description of the slave processor:

P0.7..0	SLAVEx_IN7..0	INPUT	Input from ISA interface
P2.7..0	SLAVEx_OUT7..0	OUTPUT	Output to the ISA buffer
P1.0	LED_TEST	OUTPUT	Test LED output signal: LED on = 1
P1.1	INPUT_RDYx	OUTPUT	Slave status: ready for input = 1
P1.2	Sx_IDLE	OUTPUT	Slave status: slave idle = 1
P3.2 (INT0*)	MSG_BIT_Sx	INPUT	Message from ISA bus = 1
P3.6 (RD*)	SLAVEx_OEN	OUTPUT	Read ISA buffer from FPGA to P0.7..0
P3.7 (WR*)	WR_REQ_Sx	OUTPUT	Write ISA buffer from P2.7..0 to FPGA

Table 1 Slave processor pin assigment

This interface is simple and yet allows flexibility to be used in different configurations as described below.

P0 and P2 are totally different data channels. Although only one 8 bit interface could be used and the input data stream could be multiplexed with the output data stream, there are advantages to this method. The primary one is that input commands and parameters can be retrieved at the same time that processed data is being sent to the output FIFO. This is very useful in bandwidth intensive tasks such as FIR/IIR filtering where FIFOs are used on both inputs and outputs to the slave processor. Of possible advantage is the fact that P0 and P2 are used for external memory access. The FPGA can then be used either for internal RAM access or control to a RAM or ROM with a maximum size of 64KB. One other advantage is that the pins are available, on the slave processor and the FPGA and if more IO pins are needed, the can easily be incorporated into the FPGA logic assuming there is no FPGA routing contention.

P1.0, P1.1, and P1.2 are used as status pins. P1.0 is dedicated to an LED output and is useful to debug controller code. The most common method is to have the LED turn on or off at key points in the in the code to ensure that the proper routine was executed or code had entered a certain area. This was advantageous in the absence of a logic analyzer or in-circuit emulator. At certain points an oscilloscope was needed to debug suspect waveforms, and ensure proper pulse widths and edge rates. P1.1 and P1.2 are delivered to the FPGA for the odd IO port associated with the slave processor such that the master processor can determine what the slave processor is doing.

P3.2 is read in the main loop to determine if there is a message from the processor. If a message is present, a byte is read and in the function of reading the byte the bit in the FPGA that asserts the MSG_BIT_Sx is cleared which deasserts the signal. P3.2 can also be used as an interrupt source and can be configured with no hardware change to do so. The software implementation is complicated and the benefits of an

interrupt driven messaging system are debatable. In the case of a filter task, the data stream is buffered and is more advantageously utilized as polled/looped input and an interrupt driven implementation would only cause unnecessary CPU cycle loss and consume more code space. On the other hand, an interrupt driven method would be more conventional and useful if the slave processor had various levels of possible tasks. For instance a background IIR filter could be overridden by a pi digit calculation. This is beyond the scope of this work.

P3.6 and 3.7 are also used as the external memory read and write pins respectively and appropriately function as FPGA read and write control signals. For the write action, the correct data is put on P2 and the write line, P3.7, is toggled to provide the pulse used by the FPGA to write the byte to the outgoing FIFO. To do a read, once the slave processor knows a message is available, the processor pulses P3.6 while latching P0 as the byte read. Pulsing P3.6 enables the output of the FPGA bits and clears the previously mentioned MSG_BIT_Sx signal. The method of using these two pins clearly costs processor cycles, but in the interest of being flexible in timing, and safe from extraneous memory reads/writes (the External Access, EA*, pin is tied high), this manner is used over the normal external read/write usage.

3.3 89C52 FLASH Programmer

A Xeltek SuperPro programmer is used to program the 89C52. Commercial programmers are available for $200.00, or they can be built around a parallel port interface for less than $50.00 (An application note with schematic and software is available at www.atmel.com)

Since the program is contained in the internal 8KB FLASH, the erasure and programming takes about 30 seconds which is much easier to use than UV-ROM.

3.4 89C52 Keil C Compiler

The Keil C compiler is closely tied to hardware available on the 89C52. Port pins are easily addressable and manipulated using P3, P2, P1, P0 for byte manipulation and P3.7, P3.6, etc for bit manipulation. Other special function registers(SFRs) are likewise addressable from C, but are not directly used in this project.

Standard data types char, int, float are available. The type double is not available probably since the 8051 controller is not well suited for floating point calculations in general.

Part of the compiler package is a code simulator. This is useful in the very early stages of code development to verify that compiled C-code is executing and reading/writing to the registers properly. Due to the nature of the processor having a lot of input and output requirements, this became difficult to use. Once a test LED was installed, it proved much easier to trigger the LED on or off in code areas of interest. This did require a few more iterations of programming the 89C52, but ensured the code was working properly by not risking simulation errors.

One major advantage of this compiler is that for code size under 2KB, the compiler is free on the Keil web site, www.keil.com.

3.5 Code Implementation

The 89C52 code has a main loop where it idles for tasks. Once a task description has been received, a case statement is executed transferring control to the proper routine. The routine then retrieves the proper parameters and then proceeds executing the task. The block diagram below shows how the code executes on the slave processor.

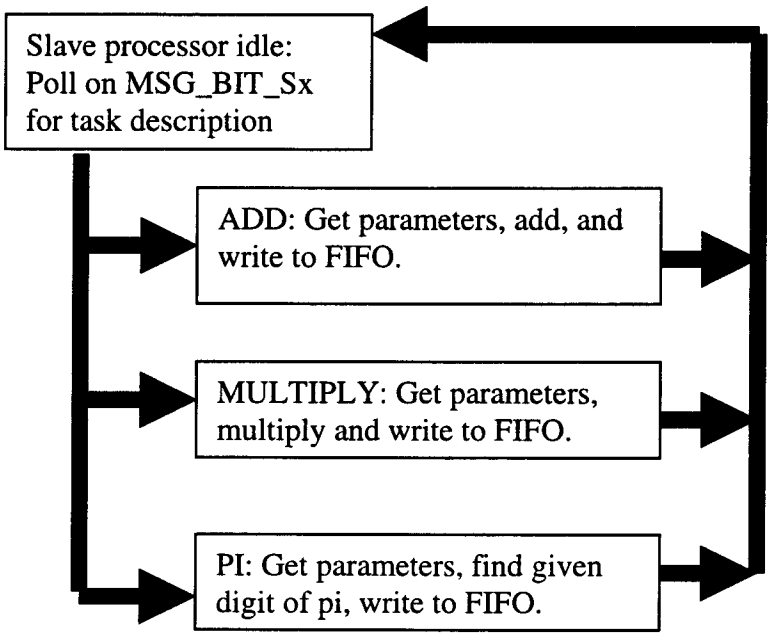

Fig 5 Slave processor code diagram

3.5.1 Addition Task

When the controller receives the task 0x01, the 8 bit addition task is initiated. After fetching the two 8 bit operands, the sum is sent to the FPGA FIFO buffer. If the sum is greater than 256, the carry bit is ignored. This is useful in quick debugging of the program and any data path changes that occur in the prototype development. The code is shown below where after the addition is performed, the P3_6 pulse latches the result on P2.

```
void add () {
unsigned char getbyte();
unsigned char rbyte1, rbyte2;
  rbyte1 = getbyte();
  rbyte2 = getbyte();
  P2 = rbyte1 + rbyte2;
  P3_6 = 1;     // write the byte
  P3_6 = 0;
```

3.5.2 Multiplication Task

The multiplication task, defined as 0x02, is a useful test of multi-byte parameters and multiplies two 16 bit values after retrieving them from the FPGA buffer. After generating the 16 bit multiplier and multiplicand from the 8 bit values, the 16 bit result is passed back to FPGA FIFO in big endian format. As in the addition routine, results larger than 0xFFFF are not supported since this routine is used mainly for debug purposes. Little endian parameter/results format could be easily implemented as the mult() below shows.

```
void mult() {
  int m1, m2, m3;
  unsigned char getbyte();
  unsigned char m1b1, m1b2, m2b1, m2b2, m3b1,m3b2;
  m1b1 = getbyte();
  m1b2 = getbyte();
  m2b1 = getbyte();
  m2b2 = getbyte();
  m1 = m1b1 * 256 + m1b2;
  m2 = m2b1 * 256 + m2b2;
  m3 = m1 * m2;
  m3b1 = (int) (m3 / 256);
  m3b2 = m3 % 256;
  P1_0 = 0;
  P2 = m3b1;
  P3_6 = 1;     // write the byte
  P3_6 = 0;
  P2 = m3b2;
  P3_6 = 1;     // write the byte
  P3_6 = 0;

}
```

3.5.3 Pi Calculation Task

Task code 0x03 indicates the calculation of a portion of a digit of pi. This is discussed in greater detail in the Pi Hex Digit Calculation chapter.

4 Altera 10K20 Interface Logic

The Altera 10K20 interface is a versatile chip that allows many different functions to be incorporated into a single package. Instead of needing separate components for FIFOs, logic gates, and flip-flop memory components, one package with an associated program or logic control file is used.

To allow for quick prototype design changes, almost all signals from the ISA bus and the slave processors are routed to the Altera 10K20 SRAM based FPGA. When a signal needs to be rerouted or logic needs to be modified/corrected, rather than re-wirewrap or add additional logic, the FPGA circuit is changed, recompiled and downloaded.

4.1 Altera 10K20 FPGA Definition

The Altera Corporation considers all their components to be complex programmable devices (CPLDs), but the FLEX10K family of products seems to cross the gray line to be considered a field programmable gate array (FPGA). The Altera definition of an FPGA is a programmable device that has multiple segmented interconnects for logic connection, such as the Actel anti-fuse families. In other words, there is almost 100% interconnectivity between the logic elements for any given logic design. Based on previously used terminology, simple PLDs normally has one or two flip-flops per IO pin as in the Lattice implementations. When the device can incorporate FIFOs and controller cores, an arguably more general description should be used. The term FPGA seems more appropriate to describe a device that has a considerable number of internally programmable blocks that are freely associated such as the FLEX10K family of products. Although the Actel anti-fuse families can fit highly interconnected logic

design better than the Altera FLEX10K family, FLEX10K components allow for very complex integrated logic and memory designs due to the hierarchical routing paths. [1]

The EPF10K20 device is estimated by Altera to be the equivalent of 20,000 PLD gates, which includes 144 logic array blocks and 6 embedded array blocks (EAB). The 144 logic array blocks allow for 1,152 logic elements. The 6 embedded array blocks of flexible RAM allow for a total of 12,288 bits of RAM. Each EAB can be configured differently such that a memory block of 256x8 bits or 2048x1 bits will consume a single EAB. Maximum IO pin count is for the 10K20 is 189. For the 208PQFP package used in the prototype, only 147 are available.

Timing between logic elements is typically 20ns max with a setup time of 6ns min and hold of 0ns min. Clock-to-Q on an external pin is 8.9ns max with an output data hold of 1ns min. These types of times made the design much more straightforward since the slower 14.7MHz ISA bus and 8MHz 89C52 are easily accommodated.

4.2 Applicability of 10K20 Selection

The Altera 10K20 was selected primarily for the SRAM reprogrammability and in-circuit programmability. Not having to throw away the part or do a lengthy UV erase when a design change is needed, reduces development cost and time. This eliminated standard one time programmable FPGAs such as Actel and Cypress. The complexity needed to interface to four slave processors and high pin count needed for a total of 5 busses, eliminated enhanced PLDs such as Lattice. Xilinx SRAM based parts remained the last contender, but with the availability of Altera parts from Field Applications Engineers (FAE), and the good web support available, the Altera option was taken.

The one disadvantage with all SRAM based parts is the reduced routability of logic elements compared to that of standard FPGAs. For this reason, the pin designation must be kept floating as long as possible to allow the place and routing algorithm flexibility. Once the pins are fixed, there is little that can be done to provide more routing

other than consume unnecessary logic resources for routing paths. In this design, once the major blocks had been defined for 4 sets of FIFOs with related controls and registers, the pins were set, and with 33% of memory used and 30% of logic used, flexibility remains for routing changes to be made. The standard rule is to have 10-20% of capacity reserved for potential logic/routing corrections, though in Actel FPGAs, this can be reduced to 3-5%.

4.3 10K20 Design Interface

The Altera supplied MaxPlus II V8.1 programming interface is used for the FPGA development. Once a design is done, the project is compiled, placed, and routed to the device specified, and made available to program.

Various programming interfaces can be used, but the passive serial method is easy to use and the schematic and cable pinouts are available on www.altera.com. This connects to the parallel port of the host computer through an LS244 buffer to a 10pin JTAG-like interface on the Altera device target. When power is applied to the 10K20, the IO pins are all tri-stated, until programming is done. The programmer serially defines the logic configuration, and once done, the Altera device sends a completion signal back to the host computer to indicate a successful programming.

Having one interface for the entire process is useful in that the similar commands are used throughout. One drawback is a unique schematic entry front end that needs to be learned. It does not correspond to various common drawing tools or other schematic tools, but once learned it is effective.

4.4 10K20 208pin PQFP Package Considerations

In general, as logic complexity increases, pin count increases in CPLDs and FPGAs. Although an 84pin package exists for the 10K20, concern for IO constraint leads to the 208pin package. Pin grid array (PGA) versions are available, but the cost is

prohibitive due to lower volume and more complicated packaging requirements. With a 0.5mm pin pitch, this was difficult to interface to a wirewrap board. The easiest solution other than soldering it to a PCB board and running delicate 30AWG wires, was to use an Aries quad flat pack (QFP) to pin grid array (PGA) adapter (part number 92-208M50), and wirewrap to the PGA pins. This is advantageous in that once the QFP is soldered down, IO pin changes can be made on the PGA adapter and signal probing can be done on the 0.1" spaced PGA.

4.5 10K20 Design definition

The 10K20 basically has two interfaces, one to the ISA bus and one to each of the slave processors that are identical. Below is the top-level diagram showing the internal address decoder and the 4 slave processor modules:

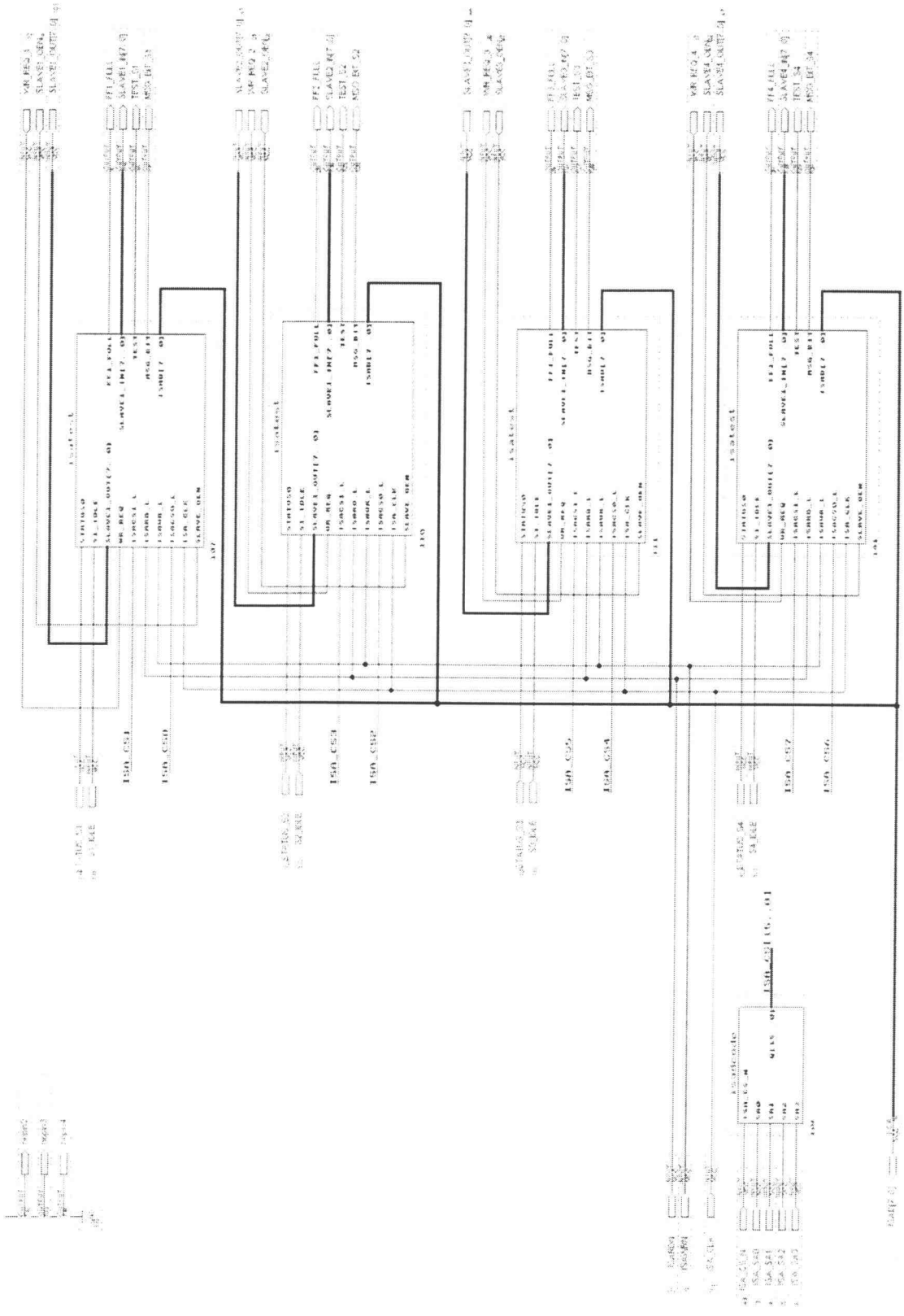

Fig 6 FPGA top level

Inside each of the identical slave processor modules is a 128 bit FIFO along with a slave processor status register and a slave processor messaging register as shown in figure 2. The FIFO has the disadvantage of being a cycle shared FIFO where one clock is used to determine the reads and writes. With the bit-banged slave processor write implementation, the slave processor is slow enough to use the ISA bus clock for the FIFO. For both FIFO reads and writes, three DFFs are used to implement an edge detector synchronous to half the ISA clock to generate a two ISA clock pulse on every read/write to the FIFO. This is needed to prevent missed or multiple reads/writes.

For each of the four FIFO blocks, there is a test pin, currently connected to the MSG_BIT signal that can be used for debug purposes. It is useful to have a permanent test pin at this second level since editing symbol pin changes in the schematic editor is time consuming.

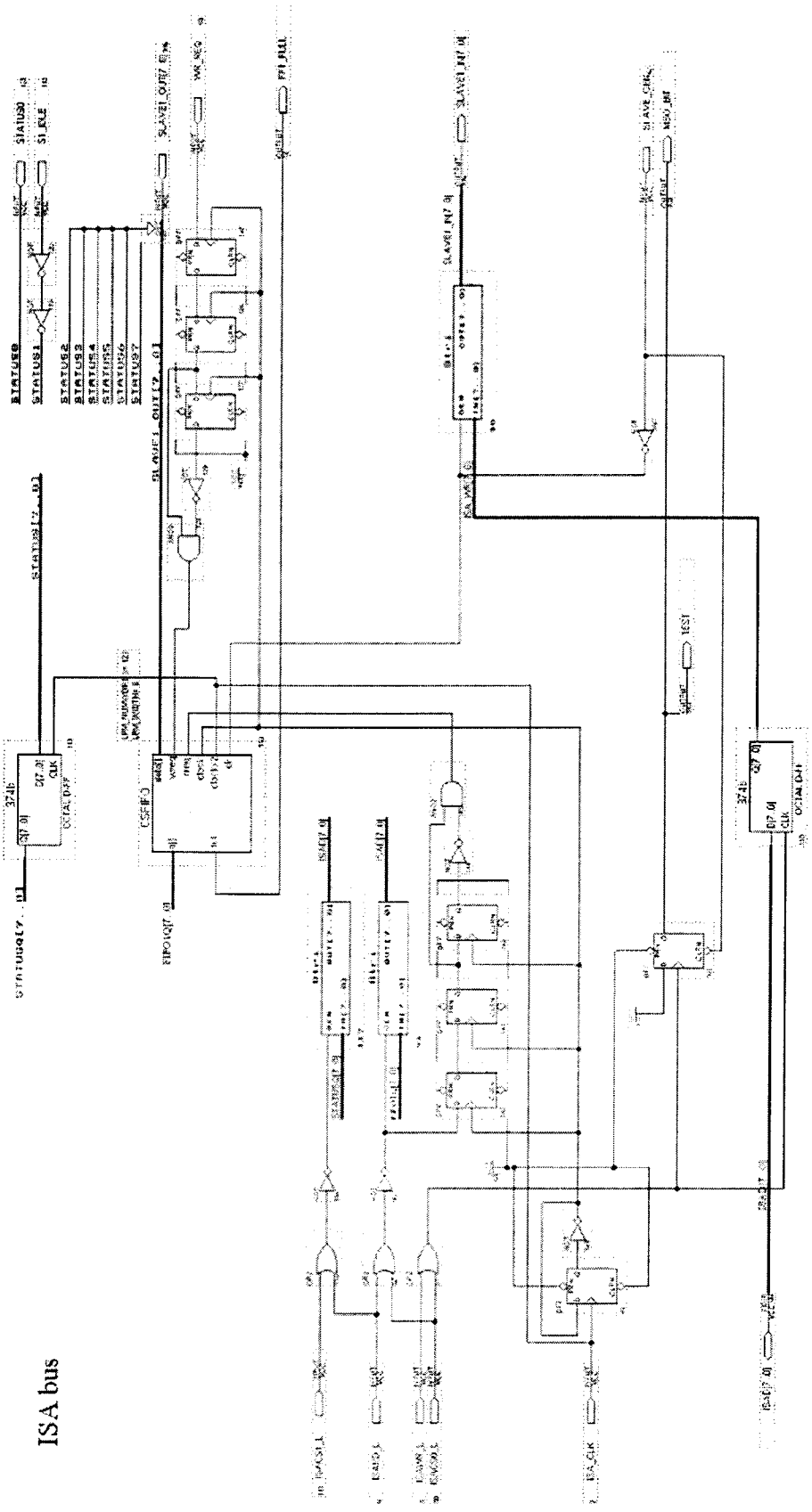

Fig 7 FPGA FIFO logic

4.5.1 ISA Bus Interface

The 10K20 is connected directly to the ISA connector with the exception of an LS32 OR gate and an LS688 comparator. The comparator selects the IO address space from 0x03E0 to 0x03E7 and two OR gates are used to AND together the qualified address space with the ISARDN and ISAWRN signals. This provides a clean signal to the Altera part. The comparator selection is used as ISA_CS_N along with the lower address bits to generate the eight separate register selects, two for each slave processor. The even numbered IO port chip select for each slave processor is reserved for sending commands/parameters and retrieving FIFO results. The odd numbered IO port chip select is used to read slave processor status. Each IO register chip select controls the tristate of the FIFO or status register respectively to drive the appropriate data onto the ISA bus to avoid driver conflicts.

4.5.2 Slave Processor Interface

As described in the slave processor interface, P0 on the slave processor is used to read data from the ISA interface. This is implemented as an 8 bit DFF that is clocked in on the rising edge of an ISA IO write assertion. The output is tristate controlled by the SLAVE_OEN signal which functions as a read signal.

Additionally a DFF, with output MSG_BIT is used to indicate to the slave processor a write has occurred. The D input is tied high such that on an ISA write, Q goes high. When the slave processor does a read from the 8 bit DFF, SLAVE_OEN also clears the DFF that clears the MSG_BIT signal. This is routed to a bit in the status register to allow the master processor checking before sending messages to the slave.

The two slave processor status bits are direct inputs to the status register. They are double inverted since the MaxPlus II software that is used to compile the design requires distinct net names that reduce schematic readability.

5 Master Processor Configuration

A Visual C++ based program is used to interface to the ISA bus. Although text based for demonstrating the prototype, this allows for expansion into a graphical user interface (GUI) should the need arise at a later time.

5.1 ISA Bus Communication

The master processor communicates to the Altera 10K20 through the IO ports. These are similar to the IO ports on serial and parallel ports found on standard PCs. Table 1 shows the IO port mapping and function for the master processor

Port address	Function
3E0	Slave processor 1 (write commands / parameters, read results)
3E1	Slave processor 1 status byte
3E2	Slave processor 2 (write commands / parameters, read results)
3E3	Slave processor 2 status byte
3E4	Slave processor 3 (write commands / parameters, read results)
3E5	Slave processor 3 status byte
3E6	Slave processor 4 (write commands / parameters, read results)
3E7	Slave processor 4 status byte

Table 2 Input/Output ports for master processor

Table 2 shows the definition of bits in the slave processor status byte. These are used by the program running on the master processor to determine if the slave processor is idle, finished with a job, or has a FIFO full problem.

Status bit	Definition
7	Byte buffer to slave processor full
6	Slave processor busy
5	--
4	--
3	--
2	--
1	--
0	--

Table 3 Slave processor status register

One of the simplest and the most direct access to the IO ports is through the usage of the DOS program debug.exe. This allows byte values to be read and written to the port access with simple commands. I.e. "O 3E0 AA" does an ISA IO port write of 0xAA to address 0x03E0, and "I 3E0" does an ISA bus read of IO address 0x03E0.

This approach, though simple, is not usable when controlling multiple tasks on multiple slave processors. For this reason, task.c was developed to allow a modifiable interface with a user-friendly front end.

5.2 Program Overview

As in the slave processor interface, this program has a non-exiting main loop. After testing the slave processors for an idle state, the user is prompted to send a task to a slave processor. The user can enter processor "0" to repeat the processor idle check. If a processor is idle after completing a task, the task retrieval routine is call. This switches on the global task type variable used to define the type of task associated with a given slave processor. Depending on the task type, different actions are taken on the data available through the FIFO.

When the user assigns a task, first the processor number is defined. The task number is then assigned based on table 4.

Task Number	**Description**
01	8 bit addition of subsequent two bytes
02	16 bit multiplication of subsequent four bytes (not supported in task.c)
03	Portion of pi digit calculation in subsequent 3 bytes.
04-FF	Reserved expansion for other tasks

Table 4 Task type list

After the task type has been sent, the parameters are sent depending on the task type. For the 8 bit addition, the requirement for user entry is sufficient to ensure the adder bytes are sent properly to the slave processor. For multibyte parameters, possible delay is inserted based on the INPUT_RDYx bit of the status register. This is to ensure that the slave processor has time to retrieve each byte. Once the parameters are successfully passed to the Altera FPGA, control returns to the main loop to check for processors coming out of an idle state.

Appendix C shows the task.c source that is run on the master to communicate with the slave processors.

6 Pi Hex Digit Implementation

The primary example of this design is the usage of the Bailey-Borwein-Plouffe Pi algorithm to determine a portion of a given hex digit of pi.

6.1 Mathematical Theory

The basic formula is:

$$\pi = \sum_{k=0}^{\infty} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \left(\frac{1}{16} \right)^k$$

This formula and associated sample C code is available at http://www.mathsoft.com/asolve/plouffe/plouffe.html. This was generated using PSLQ lattice reduction [4] and allows for an easily computable method of generating a given hexadecimal digit of π based on the following excerpt from Bailey and Plouffe[4]:

> Let S^1 be the first of the sums in the above formula for π. Then we can write

$$\text{frac}(16^d S_1) = \sum_{k=0}^{\infty} \frac{16^{d-k}}{8k+1} (\text{mod } 1)$$

$$= \sum_{k=0}^{d} \frac{16^{d-k} (\text{mod } 8k+1)}{8k+1} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+1} (\text{mod } 1)$$

> The first sum can be rapidly evaluated by means of the binary algorithm for exponentiation, where each operation is performed modulo the integer $8k + 1$. These calculations can be done with either integer or floating-point arithmetic, provided the format

> being used has enough accuracy to exactly represent the integer d^2. Once an individual exponentiation operation is complete, the resulting integer value is divided by $8k+1$, using floating-point arithmetic and added to the sum modulo 1. Only a few terms are required of the second, since the terms rapidly become smaller than the "machine epsilon" of the floating-point arithmetic system being used. The resulting fractional value, when expressed in base 16 notation, gives the hexadecimal digits of π beginning at position $d+1$.

Based on this explanation, the sample code can be adapted as shown in Appendix A to show the slave processor code implementation. It should be noted that target of the operation is to find a given hex digit of π. Although this operation requires a summation up to the given digit, this summation is much more easily done in a distributed manner that having a processor find a complete solution for a given number of digits of π.

An interesting sidenote is the pihex project on http://www.cecm.sfu.ca/projects/pihex/pihex.html where similar to the Mersenne prime distributed effort, individual contributors can post results on ranges of pi hexadecimal digits.

6.2 Slave Processor Implementation

The slave processor implements the series function above based on the m and ic parameters. The series function does a sum_k 16^(ic-k)/(8*k+m) and returns the floating-point value. Calculation time increases linearly as ic is increased.

The slave processor first receives the m byte and then the 16 bit ic parameter. After reconstructing ic into an int data type, the series routine is called that returns a float with a range 0.0-0.999999. The result is sent back to the FIFO buffer one decimal digit at a time as follows:

```
void sendfloat(out)
float out;
{
  float tmp = 0.0;
  unsigned char i=0, a = 0;
  tmp = out;
```

```
    for (i = 0; i < 6; i++) {
    tmp = tmp*10 - a*10;
    a = tmp;
    P2 = a;
    P3_6 = 1;     // write the byte
    P3_6 = 0;
    }
}
```

After each 6-digit result, an 0xAA is written to the FIFO. This serves as an additional check to the master processor. The master can determine that the task results ranging in value from 0x00 to 0x09 are done when the 0xAA is read. This would be useful for when multiple series functions are queued up. Additionally the TEST_LED is toggled for each expm() function call which iterates ic times. This is a useful indicator to determine that the slave processor is not in a hung state.

6.3 Master Processor Implementation

As discussed above, the slave processor expects the m parameter in 8 bit format and the ic parameter in 16 bit big endian format. Returned is the 6 bytes representing the float value calculated in big endian format. After the return bytes are retrieved, they are divided according to position and summed to present a float value.

6.4 Limitations of the 89C52

The series function listed above is not followed precisely for two reasons. First the Keil C compiler does not support the double precision data class. This is probably due to the fact that few 8051 variants have floating point math capabilities, and doing floating point calculations is generally very time consuming. Using the floating class of 32 bit precision, introduces errors. These errors are replicated when the same routine with float data class precision is used on another machine. The implemented routine then shows how it would be done were the float data of data type double.

Secondly, for memory constraints, the tp[ntp] array of float values is reduced from 25 to 15. This reduces the range of possible accurate hexadecimal digits computed from 2^21 to 2^11 (=2048).

6.5 Limitations of the ISA Bus Implementation

Passing 6 bytes and the 0xAA spacer for a 32 bit value could greatly be improved. The main reason this was not implemented was the Keil C compiler does not support a ftoa(), the float to ASCII data transfer function. Again, the reason for this is probably due to the small demand for 8051 floating point tasks. Passing a total of seven bytes each series function execution does demonstrate how the FIFO can be used for multiple calls and have the responses queue up easily in the FIFO.

6.6 Timing Results of calculations

6.6.1 Basic timing

The task.c program was used to pass parameters for pi digit calculations to the slaves. The master IO port write cycle is 560ns based on the 14.7MHz ISA bus. (This bus is very noisy with up to 2V undershoot on the 5V clock) The slave read cycle is 3.04us because the read bit is actually toggled to read the incoming byte as shown below.

```
// get a byte from P0
unsigned char getbyte() {
  while (1) {
    if (P3_2 == 1) { // test for masterwrite
      P3_7 = 0; // enable Altera to P0, RD_L line
                // also clear msg_bit
      A = P0;
      P3_7 = 1; // tristate Altera bus
      return (A);
    }
  }
}
```

The write cycle is is 1.44us, half the read cycle, since the write signal is simply toggled. This corresponds properly with the 12 cycles needed to do one machine language instruction on all classic 8051 variants. At 8MHz, each cycle is 125ns, with 12 cycles being 1.5us.

Basic addition is done after the last byte is passed to the slave from the master from the task.c program. An average of 5.7us is passed between the master write to the FPGA and the read from the slave processor. This is mainly due to the idle loop described in the slave implementation section, and corresponds to roughly 5.7/1.5=4 machine language instructions of delay. This could possibly be optimized in assembly language coding or interrupt driven control, but would remove the flexibility of using the C language to control the slave processor. Addition of any two 8 bit values takes 13.6us between the time the slave reads the second adder parameter and the slave writes the result to the FPGA. Task.c has an idle loop after the second addition parameter is written, for debug purposes such that the result can be returned with no user intervention. This results in a 6.48ms delay between the time the slave processor writes the addition result to the FPGA FIFO and the time the master processor reads the result.

6.6.2 Pi digit calculation results

Since the digit specified by the master processor can be multibytes, a delay is placed between the high and low bytes. This results in a forced 6.44ms between the two bytes that are always passed as parameters to the pi digit calculation. This delay is needed since the buffer between the master processor and the slave processor is only one byte deep and the assurance that the slave retrieved the first byte is needed. The master monitoring the slave processor status could easily optimize this.

The first parameter passed for the pi digit calculation is m, or the quarter of the digit to be generated as discussed above. Valid values are 1, 4, 5, and 6 and vary the time required for calculation slightly. Of more interest is the second parameter passed, ic, or

the pi digit to be calculated. This increases calculation delay linearly as shown in the table below. The linear increase is entirely expected as seen in the code.

Pi hex digit=IC (m=1)	Calculation delay
10	160ms
50	1.50s
100	3.04s
150	5.12s
200	7.20s

Table 5 Pi calculation delays for various digit positions

The calculation delay was measured on an oscilloscope by monitoring the LED toggling during the pi digit calculation as shown in the slave processor code below.

```
  getpiparam();
  P1_0 = 0;
  for (k = 0; k < ic; k++) {
    ak = 8 * k + m;
    p = ic - k;
    P1_0 = 0;    // turn LED on
    t = expm (p, ak);
    P1_0 = 1;    // turn LED off
    s = s + t / ak;
    s = s - (int) s;
  }
  P1_0 = 1;
  sendfloat(s);
//  break it up
  P1_0 = 1;
  P2 = 0xaa;    // spacer byte
  P3_6 = 1;     // write the byte
  P3_6 = 0;
```

7 Advantages of Implementation

There are various methods to do master/slave processing. This particular implementation has several advantages.

7.1 Flexibility of the architecture

By keeping the interfaces simple there is considerable flexibility in the architecture. For instance, connecting as much as possible through the Altera 10K20 allows essentially two generic 8bit pathways to and from the slave controllers. One possible application that would take advantage of this would be a multitap infinite impulse response (IIR) filter and could function as in the following diagram:

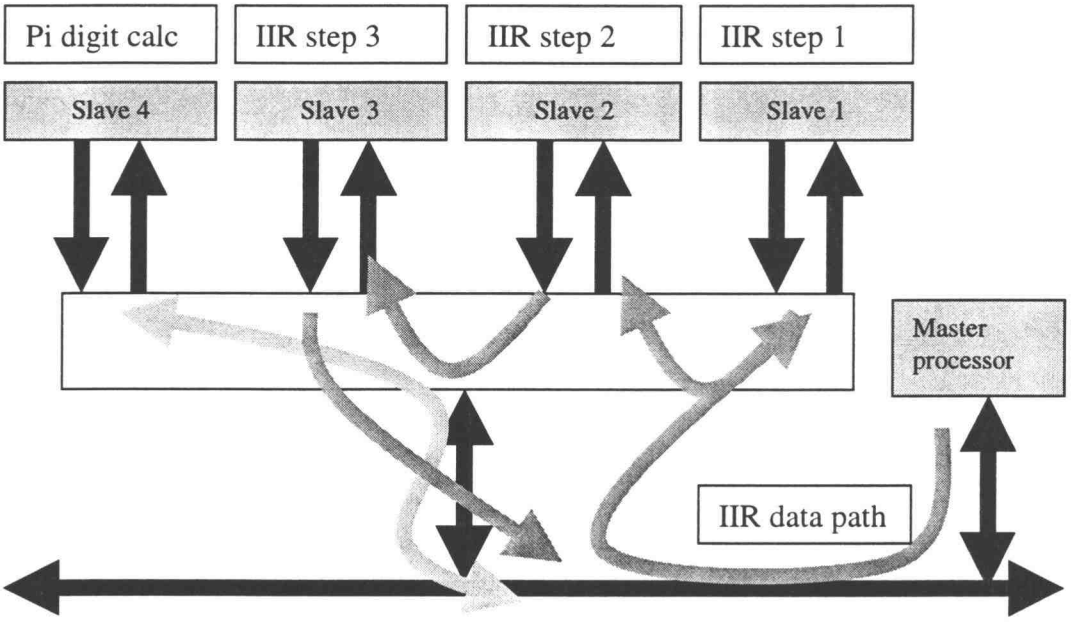

Fig 8 IIR filter example application

In this case, with FIFOs between all the transacting agents, the IIR step 1 could be a three point averaging function: $(x_{n-2}+x_{n-1}+x_n)/3$, IIR step 2 could be a normalizing function and IIR step 3 could be a comparing function with some external signal. The fourth slave processor is not needed for the filter and can be used for pi digit calculations.

The above example can be extended to discrete Fourier transforms (DFT) in the case where butterfly operations need to exchange data. In this case, processors 1 and 2 could exchange data at the same time as 3 and 4. If the DFTs are pipelined, the two processor results can be FIFOed until it is appropriate for the four slave processors to do the next step butterfly operations.

The examples above are numerical and memory intensive and would greatly benefit from more capable processors. Since the interface to the slave processors is straightforward, it would be a minor effort to substitute faster and more specialized processors. For the present purposes, the 89C52 slave processor was adequate as described in the chapter on the slave processor definition.

7.2 Standard PC Interface

The fact that the ISA interface is so widely accepted and used made documentation and other sources of information readily available. The cost of a platform with an ISA interface is low, and allows for experimentation. Should this effort be expanded to more than 4 slave processors, adding more cards to a PC based computer and allocating more IO addresses can easily do it. Different slave processors that have better capabilities (floating point operation) in other areas can easily be adapted to the FPGA interface described above. The FPGA then interfaces to the ISA bus and does not need to be changed due to a slave processor change.

7.3 Simple Board to Prototype

The 0.1" pin pitch XT expansion board was commercially available and with the exception of the Altera 10K20 all parts were placed directly into wirewrap sockets. Additionally, by using an FPGA with 147 IO pins, four slave processors could be implemented on one ISA card.

7.4 Reprogrammability of Processor and Interface Logic

With the Atmel 89C52 FLASH programmable processor and the Altera 10K20 which had almost all the interface pins, it was straightforward to implement design changes and bug fixes when they occurred. Design changes frequently were in the form of adding features, such as FIFOs, message control logic, and code additions. Bug fixes were frequently in the manner of errors in oversight that once detected with such devices as the 10K20 test pin or TEST_LED signal were quickly fixed.

7.5 Low Slave Processor Pin Count

Using a 40 pin DIP 8051 variant allowed a lot of control over the IO pins. With the RAM and ROM internal to the processor, numerous parts and wires were not needed to support the processor code

7.6 Extensibility in speed/data width

This design was done with the intent of allowing extensions to be made to improve performance.

7.6.1 Calculation Speed

The 8 MHz slave processor can be increased in frequency to improve response time linearly. Other 8051 variant processors can be used that have a floating point unit (Seimens) or do instructions in fewer than the standard 12 oscillator cycles (Dallas).

Additionally, since the FPGA interface is simple, a conventional processor could be used and the FPGA mapped into the slave processor memory map. For instance, if a DSP application were mainly being used, a DSP processor could be the slave processor.

7.6.2 Data Width

The simplest approach would be to make the ISA bus a 16 bit interface instead of the current 8 bit. The slave processors could remain at 8 bits and depending on the tasks, the 8 bit data pipes would be buffered by the FPGA FIFOs for transfer to the bigger and faster 16 bit ISA data pipe. This would be useful since the current 8 bit ISA interface must pass parameters and data to all four slave processors.

The slave processors could also be expanded to 16 bits. This would be useful in high bandwidth tasks. The bottleneck would then become the ISA bus interface, and FIFO full control monitoring would be needed.

7.7 Future possibilities and development

Possible future expandability is listed below:

- Dual FIFOs for each slave processor. The current implementation is optimal for low traffic task/parameters to the slave processor and high traffic results that are buffered in the FIFO. Dual FIFOs would allow ability to do digital filter functions on streams of data.
- Pipeline the processors. Since each processor has two 8 bit channels that can function bidirectionally to the FPGA, the processors can be pipelined to do a task on a stream of data coming from the ISA bus, through the processors and back to the ISA bus.
- Do A/D sampling with external ADCs or 8051s with onchip ADCs.

8 Disadvantages of Implementation

8.1 Controller vs. Standard Microprocessor

A controller allows more flexibility in programming direct pins on the slave processor. The disadvantage is that flexibility is reduced and the programmer is often limited to the on chip RAM and ROM. Also, since controllers are often used for non-floating point functions, there is little selection for good floating point ability in monolithic RAM/ROM controllers.

8.2 IO port versus DMA Implementation

IO port access is simpler to implement than designing a DMA interface. Though generally the DMA ability would save transfer time, in some applications the savings are negligible. The pi hex digit calculation can run hours, therefore a few microseconds saved in transfer of parameters and results would not be relevant. Large data transfers would benefit though, as in the case of digital filters and video processing.

One argument against ISA DMA is that with the decline of ISA bus, it would be more appropriate to use a PCI interface than make the effort to implement an ISA DMA interface.

8.3 Wirewrapping and ISA Prototype Board

Wirewrapping induces a signal noise at higher frequencies due to noise induced from other nearby wires and the antennae effect of the wirewrap sockets. Running the slave processor at a slower speed reduces this. Better reduction would occur with power planes, but would require a custom PCB to be built.

The fact that each wirewrap is manually connected to two pins induces errors. In a custom PCB, this could be checked by design rule check software

9 Conclusion

9.1 Major learnings

The main software evaluation tool was the pi hex digit calculation. To compare the slave processor implementation, the Bailey C code was modified to display the series() function output on a Pentium 166MHz computer. This included using the float data type instead of the double data type such that identical floating-point values were returned. This also served as a validation mechanism that the slave processor code was correct. Based on this measurement, one slave processor is approximately 2,000 times slower than the Pentium 166MHz.

This is understandable considering the floating point unit, cache and the fact the Pentium is running at 166Mhz, or twenty times faster than the slave processor. Though not encouraging, faster slave processors with floating point units can be added fairly straightforwardly and multiplied quickly based on the number of available slots in a PC.

From these results, extrapolations can be made for more general multi-SIMD/MIMD tasks. When comparing to the ILLIAC IV the 89C52 has a wide performance gap. The ILLIAC IV processing elements were designed for a 400ns 64-bit floating-point multiplication. A simple 8-bit addition takes 13.6us on the 8MHz 89C52. This is mainly due to the time overhead needed to transfer data on the slow ISA bus and the fact the 89C52s are running at a slower 8MHz.

9.2 Real world product

Based on the performance data, this is not a financially viable product. However, the implementation presented could be easily modified for such applications as described below where it would find a marketable niche.

There are a variety of applications that use controllers to monitor and log traffic statistics and patterns in telecommunications. The bulk of the traffic gets handled in the FPGA and network interface while the controller is used for sideband or monitoring control. With multiple processors, the data path could be divided between the processors and such things as compression, encryption and datastream filtering could be done on received or transmitted data.

This type of application could be done on more than digital telecommunications networks. The product could be modified to do on the fly video compression by dividing the incoming or outgoing video signal between the controllers and doing a parallel compression.

References

[1] 1996 Altera Databook, Altera Corporation

[2] Almasi/Gottlieb, Highly Parallel Computing, Benjamin/Cummings Publishing, © 1989, ISBN 0-8053-0177-1

[3] 1997 Atmel Controller Databook, Atmel Corporation, www.atmel.com

[4] Bailey, David H., Plouffe, Simon "Recognizing Numerical Constants" www.cecm.sfu.ca/organics/papers/bailey

[5] Calvin C., Implementation of parallel FFT algorithms on distributed memory machines with minimum overhead of communication, Parallel Computing 22 (1996) 1255-1279.

[6] Chiola G., Ciaccio G., Implementing a low cost, low latency parallel platform, Parallel Computing 22 (1997) 1703-1717.

[7] Danielsson, Per-Erik, Algorithm-Driven Architecture for Parallel Image Processing, NATO ASI Series Vol F18 Computer Architectures for Spatially Distributed Data © 1985

[8] Eggebrecht, Lewis, Interfacing to the IBM PC, SAMS © 1990 ISBN 0-672-22722-3

[9] Eldredge, M., Hughes, T., Ferencz, R., Rifai S., Raefsky, A., Herndon, B. High-performance parallel computing in industry. Parallel Computing 22 (1997) 1217-1233

[10] Hord, M. R., The ILLIAC IV: The First Supercomputer, Computer Science Press, © 1982

[11] Horowitz and Hill, The Art of Electronics, Cambridge University, © 1980 ISBN 0-521-23151-5

[12] Keil C Compiler Users Guide, Keil Corporation, www.keil.com

[13] John Mueller, Visual C++ 5 from the Ground Up, McGraw Hill, © 1997, ISBN 007-882307-2

[14] Sancer Yeralan & Ashotosh Ahluwalia, Programming and Interfacing to the 8051, Adisson Wesley, © 1995 ISBN 0-201-63365-5

[15] Siegel, H. J.,The PASM System and Parallel Image Processing, NATO ASI Series Vol F18 Computer Architectures for Spatially Distributed Data, © 1985

[16] Siegel, H. J., Siegel, L. J., Kemmerer, Mueller, Smalley, Smith, PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition, IEEE Transactions on Computers, December 1981

[17] Siegel, L. J., Siegel, H. J., Feather, Parallel Processing Approaches to Image Correlation, IEEE Transactions on Computers, March 1982

[18] Siegel, L. J., Siegel, H. J., Swain, Performance Measures for Evaluating Algorithms for SIMD Machines, IEEE Transactions on Software Engineering, July 1982

[19] Edward Solari, ISA & EISA Theory and Operation, Annabooks © 1992 ISBN 0-929392-15-9

APPENDICES

Appendix A: Slave processor program and compile report

Below is the final listing of the Atmel 89C52 slave processor. Following it is the compile report.

```
// split possible jobs between jobs sent by master
// 01 = A + B
// 02 = A * B
// XX = AA
#include <stdio.h>
#include <at89x52.h>
#include <math.h>
#include <stdlib.h>

// P3_2 = msg_bit input, INT0 input
// P3_7 = slave1_oen_l output
// P1_0 = test LED, 0 = on
// P1_1 = slave input busy/ready for input: 1 = ready
// P1_2 = slave idle/slave busy: 1 = idle
// P3_6 = write request, WR_L output

unsigned int ic;
unsigned int m;

main() {

  void add ();
  void mult();
  void pi();
  P3 = 0x00;
  P3 = 0xFF;
  P3_6 = 0;
  P0 = 0xFF;  //set P0 for input
  P1_0 = 1;

  while (1) {
    P1_1 = 1;
    if (P3_2 == 1) { // test for masterwrite
      P3_7 = 0;  // enable Altera to P0, RD_L line
      P1_1 = 0;  // also clear msg_bit and slave not ready
      A = P0;
      P3_7 = 1; // tristate Altera bus
      switch (A) {
        case 0x01:
          add();
          break;
        case 0x02:
          mult();
          break;
        case 0x03:
          pi();
          break;
        default:
          P2 = 0xAB;
      }

    }
```

```
  }
}

// get A and B, add, and write back
void add () {
unsigned char getbyte();
unsigned char rbyte1, rbyte2;
//  P1_0 = 0;
  rbyte1 = getbyte();
//  if (rbyte1 > 1) {
//    P1_0 = 0;
//  }
  rbyte2 = getbyte();
  P2 = rbyte1 + rbyte2;
  P3_6 = 1;    // write the byte
  P3_6 = 0;
//  P2 = rbyte1 + rbyte2 + 1;
//  P3_6 = 1;    // write the byte
//  P3_6 = 0;
//  P2 = 0xff;
//  P3_6 = 1;    // write the byte
//  P3_6 = 0;
}

// get rbyte1 and rbyte2, multiply, and write back
void mult() {
  int m1, m2, m3;
unsigned char getbyte();
unsigned char m1b1, m1b2, m2b1, m2b2, m3b1,m3b2;
  m1b1 = getbyte();
  m1b2 = getbyte();
  m2b1 = getbyte();
  m2b2 = getbyte();

  m1 = m1b1 * 256 + m1b2;
  m2 = m2b1 * 256 + m2b2;
  m3 = m1 + m2;
  m3b1 = (int) (m3 / 256);
  m3b2 = m3 % 256;
  P1_0 = 0;
  P2 = m3b1;
  P3_6 = 1;    // write the byte
  P3_6 = 0;
  P2 = m3b2;
  P3_6 = 1;    // write the byte
  P3_6 = 0;

}

// find pi digit, borrowed from David H. Bailey  960429
void pi()
{
  void getpiparam();
  void sendfloat(float);
  float expm(float, float);
  float s=0.0, ak=0.0, t=0.0;
  unsigned long k=0, p=0;
  ic = 0;  // initialize variables
  m = 0;
  getpiparam();
  P1_0 = 0;
  for (k = 0; k < ic; k++) {
    ak = 8 * k + m;
    p = ic - k;
    P1_0 = 0;   // turn LED on
```

```
    t = expm (p, ak);
    P1_0 = 1;   // turn LED off
    s = s + t / ak;
    s = s - (int) s;
  }
  P1_0 = 1;
//  s = 0.123456;
  sendfloat(s);
//  break it up
  P1_0 = 1;
  P2 = 0xaa;   // spacer byte
  P3_6 = 1;    // write the byte
  P3_6 = 0;

}

float expm (p, ak)
float p, ak;
/*  expm = 16^p mod ak.  This routine uses the left-to-right binary
    exponentiation scheme.  It is valid for  ak <= 2^24. */
{
  int i, j;
  float p1, pt, r;
/* ntp was 25 */
#define ntp 15
  static float tp[ntp];
  static int tp1 = 0;
/*  If this is the first call to expm, fill the power of two table tp.
*/
  if (tp1 == 0) {
    tp1 = 1;
    tp[0] = 1.;
    for (i = 1; i < ntp; i++) tp[i] = 2. * tp[i-1];
  }
  if (ak == 1.) return 0.;
/*  Find the greatest power of two less than or equal to p. */
  for (i = 0; i < ntp; i++) if (tp[i] > p) break;
  pt = tp[i-1];
  p1 = p;
  r = 1.;
/*  Perform binary exponentiation algorithm modulo ak. */
  for (j = 1; j <= i; j++){
    if (p1 >= pt){
      r = 16. * r;
      r = r - (int) (r / ak) * ak;
      p1 = p1 - pt;
    }
    pt = 0.5 * pt;
    if (pt >= 1.){
      r = r * r;
      r = r - (int) (r / ak) * ak;
    }
  }
  return r;
}

// get a byte from P0
unsigned char getbyte() {
  while (1) {
    if (P3_2 == 1) { // test for masterwrite
      P3_7 = 0; // enable Altera to P0, RD_L line
                // also clear msg_bit
      A = P0;
      P3_7 = 1; // tristate Altera bus
      return (A);
```

```
    }
  }
}

// get a float from P0
void getpiparam() {
  unsigned char in[2], i = 0;
  while (i < 3) {  //get three parameters
    if (P3_2 == 1) { // test for masterwrite
      P3_7 = 0; // enable Altera to P0, RD_L line
                // also clear msg_bit
      in[i] = P0;
      P3_7 = 1; // tristate Altera bus
      i++;
    }
    m = in[0];
    ic = in[1]*256 + in[2];
  }
}

// send a float character representation to P3
void sendfloat(out)
float out;
{
  float tmp = 0.0;
  unsigned char i=0, a = 0;
  tmp = out;
  for (i = 0; i < 6; i++) {
  tmp = tmp*10 - a*10;
  a = tmp;
  P2 = a;
  P3_6 = 1;     // write the byte
  P3_6 = 0;
  }
}
```

```
MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =   1253     ----
   CONSTANT SIZE    =   ----     ----
   XDATA SIZE       =   ----     ----
   PDATA SIZE       =   ----     ----
   DATA SIZE        =     66       61
   IDATA SIZE       =   ----     ----
   BIT SIZE         =   ----     ----
END OF MODULE INFORMATION.


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Appendix B: FPGA compilation report

Below is a listing of the report on the final FPGA compilation.

```
***** Project compilation was successful


** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory
LCs
POF       Device               Pins  Pins   Pins   Bits %  Utilized  LCs  % Utilized

top       EPF10K20RC208-3      56     47     8     4096       33 %    374       32 %

User Pins:                     56     47     8
```

Appendix C: Master program listing

Below is a listing of the master processor using Visual C++ V1.52. (The older version is used to allow generation of a 16 bit DOS executable).

```
// prompt for tasks and respond with results.
// Steen Larsen 7-18-98

#include <process.h>
#include <stdio.h>

unsigned int c, aspn, m, ic, i, tasktype[4], TASKED[4];
void main( void )
{
  unsigned int spn;     // slave processor number
  unsigned int status[4];
  unsigned int input();
  void output();
  void assigntask();
  void respond();
// introduction
  printf("Task dispenser V02\n");
  for (i=1;i<5;i++)  TASKED[i] = 0;
  while (1) {
    for (i=1;i<5;i++) {
    aspn = 0x3e0 + (i-1)*2;
    status[i] = input(aspn+1)%256;
    if ((status[i] == 3) && (TASKED[i] == 1))
      respond (i);
    }
//    printf("stat1=%d\n",input(0x3e1)%256);
    printf("Enter slave processor number(0 for none):");
    scanf("%d", &spn);
    if (spn > 0)
      assigntask(spn);

  }
}
// respond to a slave processor coming out of a nonidle state
void respond (unsigned int rsn)
{
  unsigned int sum;
  float sfloat, s[6];
  aspn = 0x3e0+(rsn-1)*2;
  switch (tasktype[rsn]) {
    case 1:
    sum = input(aspn)%256;
    sum = sum%256;
    printf("Processor %d responds with %d\n", rsn, sum);
    break;
    case 3:
    s[5] = input(aspn)%256;
    s[4] = input(aspn)%256;
    s[3] = input(aspn)%256;
    s[2] = input(aspn)%256;
    s[1] = input(aspn)%256;
    s[0] = input(aspn)%256;
```

```
    sfloat =
s[5]/10+s[4]/100+s[3]/1000+s[2]/10000+s[1]/100000+s[0]/1000000;
    printf("Processor %d returns s%d for digit %d with
%f\n",rsn,m,ic,sfloat);
    break;
    default:
    printf("invalid response code\n");
  }
  TASKED[rsn] = 0;  // detask processor rsn
}

// get and pass parameters of task to slave processor
void assigntask (unsigned int spn)
{
unsigned int stat, task,a1,a2,i,ichi, iclo;
  aspn = 0x3e0 + (spn-1)*2;
// if idle send task
  stat = input(aspn+1)%256;
  if (stat && 0x02 > 1) {
    printf("Enter task type: ");
    scanf("%d", &task);
//  send task type
    output(aspn, task);
//    for (i=0;i<3000;i++);
    switch (task) {
      case 0x01:
       printf("Addition task\n");
       printf("Enter adder 1: ");
       scanf("%d", &a1);
       output(aspn, a1);
       printf("Enter adder 2: ");
       scanf("%d", &a2);
       output(aspn, a2);
       for (i=0;i<3000;i++);
       tasktype[spn] = 1;
       TASKED[spn] = 1;
       break;
      case 0x03:
       printf("Pi digit calculation task\n");
       printf("Enter m of digit: ");
       scanf("%d", &m);
       output(aspn, m);
       printf("Enter digit: ");
       scanf("%d", &ic);
       ichi = ic/256;
       iclo = ic%256;
       output(aspn, ichi);
       for (i=0;i<3000;i++);
       output(aspn, iclo);
       for (i=0;i<3000;i++);
       tasktype[spn] = 3;
       TASKED[spn] = 1;
       break;
     default:
       printf("Bad task number\n");
     }  // switch
  }  // if idle
  else
    printf("Slave processor is busy\n");
}  // while loop

// get a byte from IO port
#pragma warning( disable : 4035) //kill "no return..."
unsigned int input(unsigned int port)
{
```

```
  _asm {
    xor ax, ax;
    mov dx, port;
    in ax, dx;  //return in eax
  }
}
#pragma warning( default : 4035)

// send a byte to IO port
void output(unsigned int port, unsigned char data)
{
  _asm {
    xor ax, ax;
    mov al, data;
    mov dx, port;
    out dx, al;
  }
}
```