2007

KENTUCKY'S ADAPTER FOR PARALLEL EXECUTION AND RAPID SYNCHRONIZATION

Swetha Mitta
University of Kentucky, mitta.swetha@gmail.com

Recommended Citation

Mitta, Swetha, "KENTUCKY'S ADAPTER FOR PARALLEL EXECUTION AND RAPID SYNCHRONIZATION" (2007). *University of Kentucky Master's Theses*. 439.
https://uknowledge.uky.edu/gradschool_theses/439

ABSTRACT OF THESIS

KENTUCKY'S ADAPTER FOR PARALLEL EXECUTION AND RAPID SYNCHRONIZATION

As network hardware has become faster, inefficient communication and synchronization mechanisms often have proven to be “fast enough” – but better models are needed in order to support future systems. The aggregate function communication model, and the KAPERS design and implementation presented in this thesis, provide more efficient ways to implement a wide range of higher-level communication and synchronization operations. The main contributions of this work center on a new way to use FPGA-based memory in an aggregate function network (AFN). The basic functions were designed and implemented with modal encoding to create a global memory that allows variable length objects and object addresses. New and enhanced algorithms were written for use with the new AFN architecture. This thesis also details the KAPERS prototype hardware implementation.

KEYWORDS: Parallel Processing, Barrier Synchronization, Aggregate Functions, VHDL, FPGA.

Swetha Mitta

Author’s Signature

02/28/2007

Date

KENTUCKY’S ADAPTER FOR PARALLEL EXECUTION
AND RAPID SYNCHRONIZATION

By

Swetha Mitta

Dr. Henry Dietz
Director of Thesis

Dr. Yuming Zhang
Director of Graduate Studies

02/28/2007
Date

Name	Date

THESIS

SWETHA MITTA

The Graduate School
University of Kentucky
2007

KENTUCKY'S ADAPTER FOR PARALLEL EXECUTION AND RAPID SYNCHRONIZATION

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the College of Engineering at the University of Kentucky

By
Swetha Mitta
Director: Dr. Henry G. Dietz, Department of Electrical Engineering
Lexington, Kentucky

2007

DEDICATION

To my dear parents and brother

ACKNOWLEDGMENTS

It has been a privilege for me to work under Dr. Henry Dietz, advisor for my Master's thesis. I would like to thank him for all the support and guidance during the work. I am thankful to Dr. William R. Dieter for helpful suggestions in my work. I would also like to thank Dr. J. Robert Heath for serving as member for my defense committee. Finally, to my parents who always supported and motivated me to reach greater heights in life and achieve my goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FILES

LIST OF ALGORITHMS

Chapter1

INTRODUCTION

Amdahl's law states that the operations that cannot be sped up are the ultimate limit on the total system performance. In parallel computer systems, communication and synchronization overheads are the primary reasons that many code constructs cannot be sped up.

An Aggregate Function Network (AFN)[1] is an architectural model that can dramatically reduce interprocessor synchronization and communication time for a wide range of higher-level operations. It does this by using the network as dedicated computing hardware to implement functions of global state of parallel program, i.e. to perform the functions on the data gathered from the nodes, instead of just routing messages and computing the desired functions in the nodes. Developing a new, higher performance, AFN model named KAPERS (Kentucky's Adapter for Parallel Execution and Rapid Synchronization) is the primary focus of this thesis.

KAPERS is the newest evolution of AFN architectural implementations, preceded by over twenty designs built since February 1994. Using an FPGA[1] based design incorporating a new memory mechanism KAPERS can directly implement a number of aggregate functions that had to be simulated in all previous designs. For example, although it contains no function units more complex than a 4-bit adder, KAPERS can perform double precision floating point summation internally with good efficiency.

1.1 Definitions

Although the concept of aggregate function communication is a dozen years old, it is not yet as common as message passing networks. Thus, it is useful to review some of the basic terminology and concepts associated with aggregate functions.

1 FPGA, Filed Programmable Gate Array, a semiconductor device containing programmable logic components and interconnects to realize desired functionality

1.1.1 Barrier Synchronization

Synchronization is the coordination between PEs[2] connected as a parallel system to complete a task. Barrier synchronization is a type of synchronization mechanism in which no PE's process is allowed to execute past the barrier until all PEs have reached the barrier. The barrier synchronization can be a hardware or software generated. Hardware barrier synchronization needs some sort of dedicated hardware connected to all the nodes and the software generated barrier does synchronization by message broadcast between the nodes.

Barrier Synchronization in the AFNs is basically implemented by the hardware capable of performing an AND operation of processors inputs. Every processor on reaching the barrier outputs '1' which previously was initialized to '0' and then wait till the AND gate output is '1' and then simultaneously resume execution past the barrier. In this new KAPERS model a function 'BarOr' is defined for synchronization; it sends out the logical OR of the data collected from the processors besides doing the barrier synchronization. This barrier synchronization is sufficient enough to implement fine-grain MIMD, SIMD, VLIW[3] executions with low latency in conventional processors as explained in [11] [12].

1.1.2 Communication

Communication operation is mainly influenced by including memory in KAPERS unit. The PEs have the access to data asynchronously and can work independently relative to others until there is a need for synchronization. Each PE can write and read information from the memory location assigned to processor, or group of processors it needs the data from. Thus, the sender PE only outputs data; it is up to the receiver PE to look for the data that the sender has made available to it in the memory.

2 Here PE (Processing Element) refers to a uniprocessor node

3 MIMD, SIMD, VLIW refer to Multiple Instruction Multiple Data, Single Instruction Multiple Data, Very Long Instruction Word respectively

1.1.3 Lock

Lock is a synchronization mechanism where access to global resource is limited or serialized to avoid concurrency. Only one process at a time may own the lock. The first process to acquire the lock sets it. Other process can attempt to acquire the lock but wait until the process that owns the lock releases it. Lock requires hardware support for efficient implementation. But acquiring the lock needs to be done in single atomic operation to avoid concurrency if multiple processes are involved.

Locking is currently used in the KAPERS design for keeping data safe from unauthorized changes as there is a single shared memory resource for multi processors and is conceptually simply locking the address space. In this design, the lock mechanism is used to acquire the lock on the current nybble before access, and to acquire the lock on the next nybble before releasing the lock on the current nybble. There are several ways to implement the locking mechanism:

- Insertion of an extra bit for each memory space, to signify whether it's locked or not. Though this simple logic outwits other methods, it wastes memory space as there is a need for increment in the memory width by one.
- Another way to implement lock mechanism is to design a lock register exclusively for each processor and load it with the address of location that is locked. Before accessing the memory space each processor needs to check with the other processor lock registers to know whether the address is locked or not. Need for the access of other processor lock registers breaks the processors nature of working independently relative to others.

The second method described above, lock using dedicated registers is applied in the current design of AFN.

1.2 Background Information

From a decade there are several versions of AFNs that were built with either slight variations in architecture or hardware. The first one to be built in the line was PAPERS0[4] in February 1994 with emphasis on dynamic barrier implementation. It basically had one PAL[4] per processor with group of TTL[5] drivers for proper interface between the parallel ports of PCs. The logic was basically a barrier to synchronize followed by an anti-barrier to ensure that all participating processors have detected that synchronization was achieved. The only communication supported by the hardware was a 1-bit multi-broadcast where each processor gets a bit mask containing one bit from each processor, intended to provide a means for voting on membership in a new barrier mask.

The next was PAPERS1 prototype with features like dynamic partitionability, communication operations and improvements in speed up of barrier and data operations [5] in August1994. This used two PALs per processor one for barrier and the other for communication helping to perform both 1-bit and 4-bit multibroadcast. All the enhanced data communication operations developed for PAPERS1 required just 2 port operations compared to 4 cycles for the PAPERS0.

Then followed the series of TTL_PAPERS prototypes built with only TTL parts. TTL_PAPERS 951201[2], modularly scalable 8 processor prototype was one of them with subsystems in architecture for synchronization, parallel signaling, data communication and status display. Scalable version of this series was TTL_PAPERS 960801[13].

1.3 Information on other interconnect networks

<u>The Simultaneous Optical Multiprocessor Exchange Bus:</u>

The SOME-bus architecture[6] contains a dedicated channel, which is an optical interconnect for each processor to eliminate global arbitration and to provide bandwidth that scales with the number of processors in the machine. The interconnection network is a ribbon of optical fibers carrying all the signals for the bus. Figure 1 shows the SOME-bus interconnection scheme. Unlike electrical buses, this is not limited by the medium (fiber optics) used to connect the transmitters and receivers. Each processor has a single transmitter and an array of receivers. The transmitter is used for all communication with

4 PAL, Programmable Array Logic, semiconductors used for implementing logic functions in digital circuits

5 TTL, Transistor-Transistor Logic, class of digital circuits built from bipolar junction transistors and resistors

other PEs and is on the dedicated channel, and the receivers are for each processor channel. The architecture of the receiver array permits a variety of different parallel programming models to be efficiently supported. The basic communication primitive in the SOME-Bus is a broadcast and all processors receive the packet simultaneously.

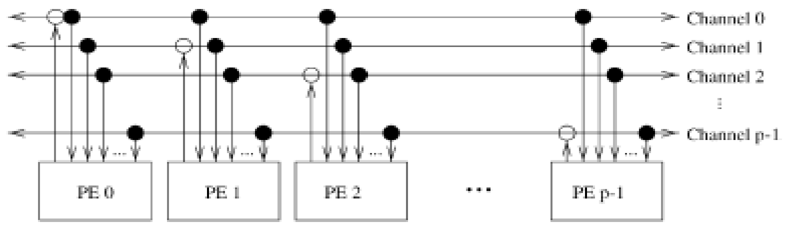


Figure 1: PEs connected by SOME-bus architecture

Barrier mechanism in this architecture is a fuzzy barrier[13] that has two phases. In the first phase, a processor broadcasts a barrier to each processor through the channel; the receiver element may discard the message if it is not useful to its local processor. In the second phase, the processor samples the result of the AND tree, which determines the barrier satisfaction. The processor waits until the result of the AND tree is obtained, indicating that the barrier is complete, and when done the processor resets the receiver elements for the next barrier operation and the processor continues. This mechanism can perform the barrier operation in less than a 1µs for the developed hardware design.

<u>QsNetII Network:</u>

QsNetII[7] is the latest in generations of Quadrics interconnect products. Interconnect in this network is a PCI card with Elan4 communication processor in each computing node. The processor acts as the interface between PE and a high performance multistage network through standard PCI-X bus. The network is constructed from Elite4 switch components that support point-to-point transfer between arbitrary nodes and broadcast across selected ranges of nodes. These Elite4 switches can be used to combine up to thousand of nodes, in a fat tree structure that scales up in the power of four. Barrier synchronization with scaling behavior can be implemented with the help of hardware broadcast.

k-ary *n*-cube network with wormhole routing:

For *k*-ary *n*-cube network[16], *n* is the dimension of the network and *k* is the number of nodes in each dimension of the structure, with k^n nodes in total. Bi-directional channels connect every node to all of its nearest neighbors. In this network node location is represented by a vector with two bit-vector fields; one represents the location of the node within the plane and the other node dimension location.

Wormhole routing[15] differs in the way of storing the packet completely and then transmitting it to the next node which is usually done by ordinary store and forward routing schemes. Wormhole routing sends the head of the packet of which only few flow control digits (flits) get buffered at each node. As soon as a node examines the header flit(s) of a message, it selects the next channel on the route and begins forwarding flits down that channel. As the header advances along the specified route, the remaining flits follow in a pipeline fashion. By using wormhole routing in the *k*-ary *n*-cube structure, the latency can be diminished as the message increases in size as the majority of latency is hidden in the transfer of first packet. The remaining packets which follow the header packet introduce only wire transfer delay.

Some of the direct networks that use wormhole routing are Ncube-2 (hypercube), Intel Touchstone Delta (2D mesh), Paragon (2D mesh), MIT J-Machine (3Dmesh) and Cray T3D (3Dtorus). Another work and research on the wormhole directed networks for a fast, scalable synchronization scheme is a multi-destination wormhole mechanism [14].

1.4 Thesis Outline

The new memory model architecture design for the AFN is explained in the Chapter 2. It also explains how the design is captured in VHDL[6] for programming the FPGA. Improvised algorithms for the aggregate operations -- associative reductions, scans, communication, voting and scheduling fitting the new model are talked in the Chapter 3. Chapter 4 describes the implementation of KAPERS hardware board design. Conclusion for the work done is in Chapter 5.

6 VHDL, VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, is used as a design entry for FPGAs

Chapter2

KAPERS ARCHITECTURE

The primary difference between KAPERS and previous AFNs is that KAPERS uses a new architectural design. This design is not only new in terms of how various types of operations are constructed, but also differs from previous designs in that the high-level architectural design is more completely separated from the implementation hardware details than had been the case for earlier AFNs. In this chapter, the high-level architecture is our focus; Chapter 4 discusses one implementation hardware structure supporting this new architecture. It is further useful to divide the current chapter's architectural description into two sections:

1. The architectural overview, describing the programmer's view of the system and its high-level functionality
2. The VHDL design capture, detailing the functional decomposition into hardware modules and the complete logic-level design

2.1 Architectural Overview

This section gives a brief description of the hardware interface, basic functions, memory model, and memory functions that constitute the KAPERs architecture.

2.1.1 Interface

The network interface plays the key role in determining the available bandwidth and latency for communication in message-passing networks. The current version of KAPERS uses the parallel port as the network interface. The parallel port used is a standard parallel port (SPP). It has 8- bit data output, 4-bit open collector output, and 5-bit input accessible through status, control, and data registers. Parallel port register access is performed as a direct user polled I/O by setting the address permission mask to minimize the system call overhead for reducing the latency. Though the latency in accessing any port register is approximately 1μs, the bandwidth is limited to few megabytes per second in this case. The reason for using the parallel port as network interface despite its poor capability in regard to bandwidth is that none of other available interface units guarantee simple and cost effective design.

Hopefully, a future interface with very similar capabilities will provide comparable or better latency without crippling the bandwidth. Certainly, commodity network wire transport technologies such as Cat5 Ethernet fundamentally provide a comparable number of bits with a much faster transmission clock cycle, and there is no technical reason that such a physical interface could not be made directly accessible to the processor.

The interface signals between PE's parallel port and the KAPERS unit are summarized in Table 1 followed by their brief functionality:

Table 1 Interface signals between PE and KAPERS

Pin # of DB25	**Direction relative to PE**	**Direction relative to KAPERS unit**
Pin2 – Pin9	Output from PE	Input bits (D7 – D0)
Pin 10,11,12,13,15	Input to PE	Output bits (O4 – O0)
Pin 1,14,16,17	Output from PE (Open Collector)	PE number indicators

A simple data transfer requires minimum of 4 to 5 communication cycles; two cycles for input data and signaling, and two to three cycles for output data and signaling. These communication cycles can be reduced to 2 if the signaling bit is accompanied with the data. With the KAPERS unit the I/O operations for all data and barrier operations are achieved in 2 cycles(one for input and one for output) by setting a strobe bit that gets written in the same port register write that sets the other bits in both input and output directions. For the incoming data to KAPERS the topmost bit D7 is used as signaling bit, and bit O4 is used to indicate the readiness of data in bits O3 to O4, the computed output to be sent to the PE.

The above method for reduction in I/O operations can be accomplished only if the FPGA is used for resolving the signal race condition between the strobe and data lines using the fact that the PC cannot initiate a new port operation any faster than once per microsecond. For this, in the case of incoming data to KAPERS, strobe bit is analyzed first and then data is processed only after a certain delay of about few clock cycles of FPGA in which it gets stabilized. In the case of outgoing data the signaling bit O4 is sent after a period of about 40ns from the time the computed data is sent to parallel port of PE for letting data to get stabilized for retrieving.

Bits D6 to D4 from PE to KAPERS unit indicate the basic functions to be implemented and are explained in Section 2.1.3, while bits D3 to D0 specify the data to be processed.

2.1.2 Modal Instruction Encoding

In the aggregate function communication model, an aggregate operation is performed by each node writing a datum and opcode pair to the AFN hardware and then reading the AFN-computed result. In all previous AFN implementations, that is also an accurate description of how the hardware operated. However, with a relatively narrow interface, a richer set of operations, and potentially much larger data objects, it becomes impractical to encode each operation as a self-contained instruction. Thus, the instructions are transmitted to the KAPERS AFN in a simple contextually-compressed form: as modal operations.

For example, an opcode is not sent with each instruction. Instead, an opcode context is defined such that a “default” opcode is implicit in each action within that context. Setting the “add mode” essentially makes the default opcode add until set otherwise. Note that this does not imply that every operation must be an add until the default is changed; rather, it simply implies that the operation is add for instructions that can have such an opcode and are executed in that context. For example, setting “add mode” and then sending address bits does not imply that the address bits are added – but subsequent memory access instructions would indeed use the implied add opcode.

The contextual mode concept is used throughout the KAPERS architecture, not only for memory operations. For example, this same mechanism is what allows KAPERS to have arbitrary-length memory addresses and variable-size data objects without the overhead of sending the maximum number of nybbles for each operation. Variable-size addresses are handled by the concept of an "address setting" mode in which each subsequent nybble is inserted in the modally next nybble position within the address register. A similar modal concept sequences through nybbles within a multi-nybble object in memory, without altering the base object address register. This modal structure not only dramatically reduces the bandwidth required to connect each node by avoiding extra bits in each transmission, but also allows even greater compression by deliberately truncating a transmission sequence as soon as the remaining references are known to be moot. For example, if the address register is to be set to a value that differs only in the lower bits from its current value, it is not necessary to send the higher bits.

As an example of how useful this concept is, consider the case of votecount operation, where a node casts a vote for the PE it desires by adding '1' at the memory location assigned for that particular PE and gets its count of votes gained from its allotted memory location.

```
intp_votecount(register int d)
{
        step1: p__address(D_ADDRESS + d*sizeof(NPROC)*2);
        step2: p__setfunc(add,unlock);
        step3: p__memfunc(1);
        step4: p__baror(0);
        step5: p__address(D_ADDRESS + IPROC*sizeof(NPROC)*2);
        step6: return(p__memfunc(0));
}
```

The algorithm for votecount is explained further in Section 3.4.3; here, our concern is simply the improvement due to modal instruction encoding. Setting the address, in steps 1 and 5, is done by entering an address-setting mode and sending nybbles in low to high order. If addresses take 16 bits, then one would expect to have to send 4 nybbles, but nearby addresses can be set by sending only the changed low nybbles – we need not specify an address position in which to place each nybble. The add mode set in step 2 is used in both steps 3 and 6, saving transmission of opcode for each nybble of these two operations. Note that the intertwined sequences for the address and memfunc operations do not affect the modal operations, since address-setting modes are disjoint from addition, both can be current at the same time. Likewise, the non-modal *BarOr* causes no conflict. Using these modal encodings easily reduces the number of bits that must be transferred to less than half.

2.1.3 Basic Functions

The basic instructions defined in KAPERS were intended to do barrier synchronization, set memory operation, specify selected low bits of an address, and transmit data of variable length. Respectively, these functions are implemented by *BarOr*, *AddrFirst, AddrNext, SetFunc, MemNext, and MemLast* operations. Opcode space has been reserved for defining two more functions if experience with the system reveals that additional functions would be desirable.

The function *BarOr* is defined for the barrier synchronization operation, where every PE waits at the barrier until all others arrive and returns with a logical OR of all the data from the PEs. If only a barrier synchronization operation is desired, the data can simply be ignored as "don't care" values.

AddrFirst brings in the least-significant nybble (LSN) of the address where the operation needs to be carried out in the memory. *AddrNext* does the same operation as *AddrFirst,* but installs the nybble it receives in the next higher order bits of address. Thus, changing from one address to another only requires sending the low-order address bits that differ. Further, there is no inherent limit on the total number of address bits; the address nybble sequence can be arbitrarily long. Both the *AddrFirst* and *AddrNext* operations are grouped as *Address* for quick reference.

SetFunc specifies the modal memory function to be carried out on the forthcoming data coming from the PE until changed otherwise. The *SetFunc* also specifies whether the modal operation needs to be locked or not.

MemNext does the already set operation on the data it brings in with the value in the memory. The return value will be the old value in memory before the operation was performed. If the function was to be implemented as locking, it locks the address it need to work on before performing the operation and then releases the lock after getting the lock on the next address. *MemLast* is similar to the *MemNext* operation except that it signifies the data brought is the last byte to be processed with additional clearing operations on carry and offset.

The above discussed operations are tabulated below in Table 2 with their assigned bit patterns and functionality. The input to the KAPERS unit is a 8 bit data D7 to D0. The low 4 bits, bits D3 to D0, of the input have a meaning which depends on the opcode in bits D6 to D4. For every opcode but *SetFunc*, low 4 bits contain a 4-bit (nybble) datum. In the case of *SetFunc*, bits D3 to D0 specify the "modal memory function operation" to be applied to data when memory operations are performed which are explained in the next section. More precisely, bits D2 to D0 specify the operation and bit D3 specifies whether the operation locks memory or not.

Table 2 Basic operations of KAPERS unit

AFN Operation	**Opcodes**	**Return Value**	**Action Within AFN (after return)**
BarOr(d)	BarOr(d)	result	unlockall(); carry=0; off=0; wait(); result=or(d from all PEs);
Address(d0..dk)	AddrFirst(d0)		unlockall(); carry=0; off=0; addr[0]=d0; ++off;
	AddrNext(d1)		addr[off]=d1; ++off;
	...		
	AddrNext(dk)		addr[off]=dk; ++off;
SetFunc(d)	SetFunc(d)		unlockall(); carry=0; off=0; Func=d;
MemFunc(d0..dk)	MemNext(d0)	return	if (locking(Func)) lock(addr+off); Func(mem[addr+off],d0); if (locking(Func)) { lock(addr+off+1); unlock(addr+off); } ++off;
	MemNext(d1)	return	if (locking(Func)) lock(addr+off); Func(mem[addr+off],d1); if (locking(Func)) { lock(addr+off+1); unlock(addr+off); } ++off;
	...		
	MemLast(dk)	return	if (locking(Func)) lock(addr+off); Func(mem[addr+off],dk); unlock(addr+off); carry=0; off=0;

2.1.4 Memory Functions

The hardware memory functions were structured to cover the basic arithmetic and logical operations with minimal operations possible. They are *Xchg, Or, Xor, Add,* and *Min*. A point of note is for the order in which the nybbles are sent for these operations. The *Xchg, Or,* and *Xor* operations are order independent whether the least-significant or most-significant nybble arrives first. *Add* requires the low nybble first. However, *Min* requires the high nybble first. This different ordering of nybbles for the *Add* and *Min* operations is not a problem because most data objects will be used for single operations across the parallel system.

In the hardware the *Add* operation has the capability for adding carry generated from previous nybbles in the current nybble addition. Thus useful for adding different sized data objects greater than nybble width.

It is highly desirable to have apparently atomic operations in case of multiple processors units sharing a single memory, but the low throughput of a nybble-wide interface makes a simple locking mechanism highly inefficient: locking a 64-bit object would keep other processors from making accesses for 16 nybble-transmission cycles, each of which would take at least a microsecond. Fortunately, the nybbles within an object will “always” be traversed from the low nybble walking up toward the high nybble, so atomicity can be ensured if we simply make it impossible for a PE to "outrun" another PE that is ahead of it in the scan order. In this design, the lock mechanism is used to acquire the lock on the current nybble before access, and to acquire the lock on the next nybble before releasing the lock on the current nybble.

The return value to the nodes from these memory functions will be the previous value in the memory and it need not be delayed until the operation is implemented nor till the lock on next address is acquired. The following table (Table 3) lists out the memory functions with the bit pattern assignment and functionality:

Table 3 List of modal memory functions

Modal Memory Function Name	Bit Pattern For Bits 2, 1, 0	Func(m,d) Semantics
Xchg	000	return=m; m=d;
Or	001	return=m; m|=d;
Xor	010	return=m; m^=d;
Add	011	return=m; (carry,m)=m+carry+d;
Min	100	return=m; if (carry==0) {m=min(m,d); carry=(m!=d);}

2.1.5 Memory Model

KAPERS included a memory model that could be extended in length as needed by increasing complexity. Each PE can asynchronously write, read, and perform atomic read-modify-write operations on the memory locations. Initially, the memory was designed as a regular block of memory capable of writing and reading out the data; so each PE had to determine the operation to be performed on data in memory, wait for its turn to access the memory to get the value, execute the operation on the data, and then send the result back to memory. Our original design thus associated an ALU with the interface for each PE.

However, concurrent attempts to access the same single-port memory module (of which the current design has only one) must be serialized. Thus, parallel execution of memory operations is constrained primarily by the number of memory modules, not by the number of PEs connected. Taking advantage of this fact, an ALU is associated with each memory module instead of each PE connection. This not only simplifies the circuitry, but also can save clock cycles: instead of a read, ALU modification, and then a write, each operation is done as a single-cycle read-ALU-write path within the memory module. This solution does require sending the ALU opcode to the memory module, thus adding a few signals to the memory interface. The block diagram in Figure 2 gives the interface overview of the two memory models.

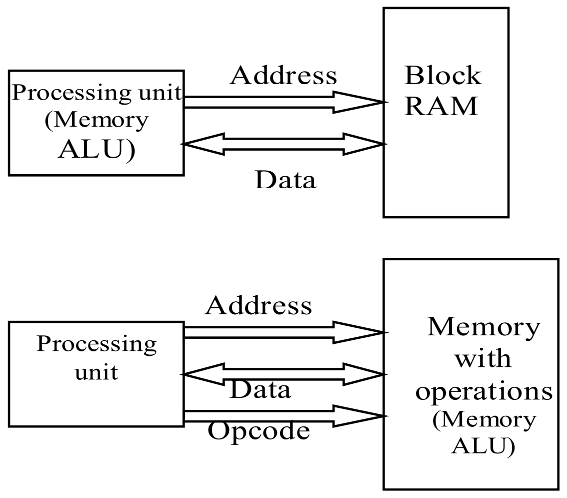

Processing unit (Memory ALU)
Address
Data
Block RAM
Processing unit
Address
Data
Opcode
Memory with operations (Memory ALU)

Figure 2: Models for memory implementation

2.2 VHDL Design Capture

The logic level design for the KAPERS architecture discussed in the previous sections was small enough to be realized in a low-end FPGA. For this purpose, the complete design was expressed as VHDL code. The VHDL was written to be compatible with the Quartus II 6.0 [22] design software provided by Altera, which supports synthesis, verification and programming functions for FPGAs. This section details the VHDL design. To simplify the design flow, the design was divided into the following blocks:

❍ Input Detection

❍ Sequence Detection

❍ FIFO (First In First Out) Input Buffer

❍ Processing Unit to decode the operation requested

❍ Shared Resources Control Logic

❍ Shared Memory, which incorporates an ALU in its interface

Figure 3 gives the diagrammatic structure of the interaction of the hardware function blocks.

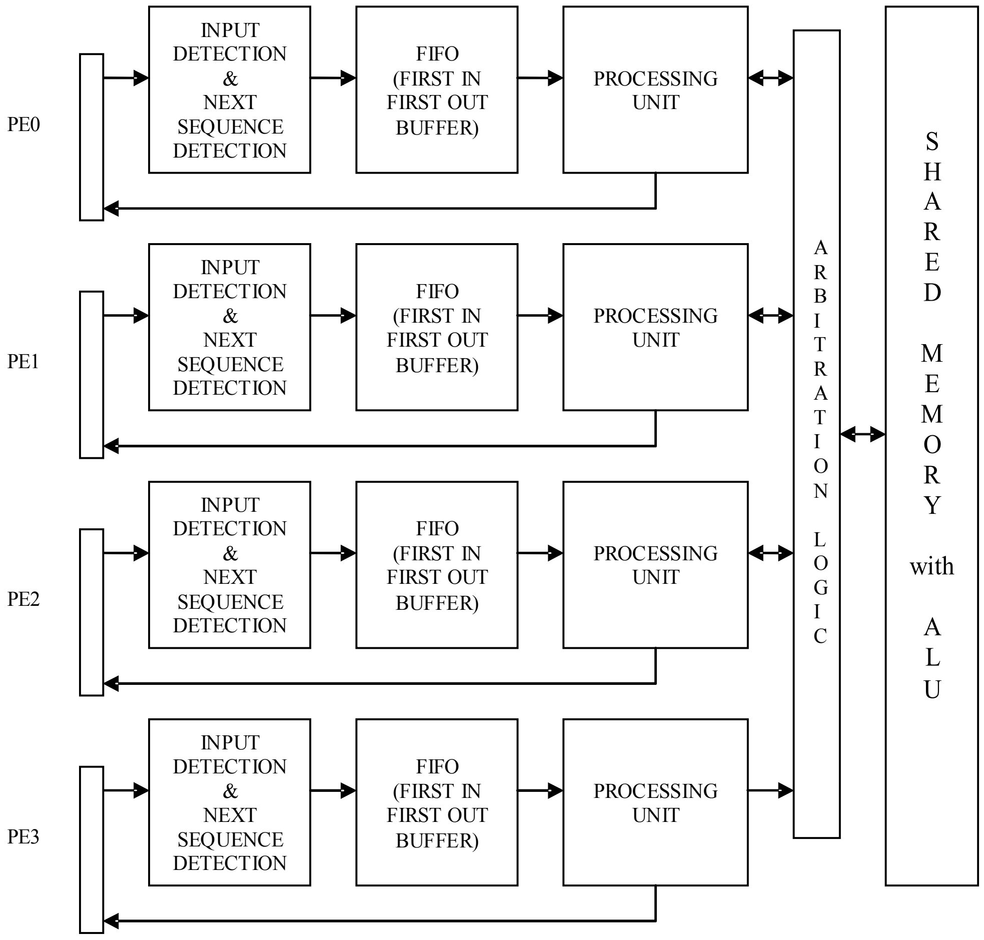

PE0
PE1
PE2
PE3
INPUT DETECTION & NEXT SEQUENCE DETECTION
FIFO (FIRST IN FIRST OUT BUFFER)
PROCESSING UNIT
ARBITRATION LOGIC
SHARED MEMORY with ALU

Figure 3: Block diagram showing different sections in the VHDL design

2.2.1 Input Detection Circuit

There is a difference in speed at which the data comes in from the nodes to the KAPERS unit and the internal clock speed in FPGA. A standards-compliant parallel port should not be generating signal transitions at a rate higher than a few megahertz, with about 1 microsecond per signal transition typical. Because TTL logic level signaling is used over relatively long cables, signal transitions can take up to hundreds of nanoseconds. In contrast, the FPGA clock ticks every few tens of nanoseconds. This situation corresponds to an asynchronous clock domain where the data has to be captured while it is stable for a reasonable time, rejecting glitches if any.

For this, the input is checked for stability for a few clock cycles and only then it is sent for further processing. Figure 4 shows the state diagram of the circuitry implemented.

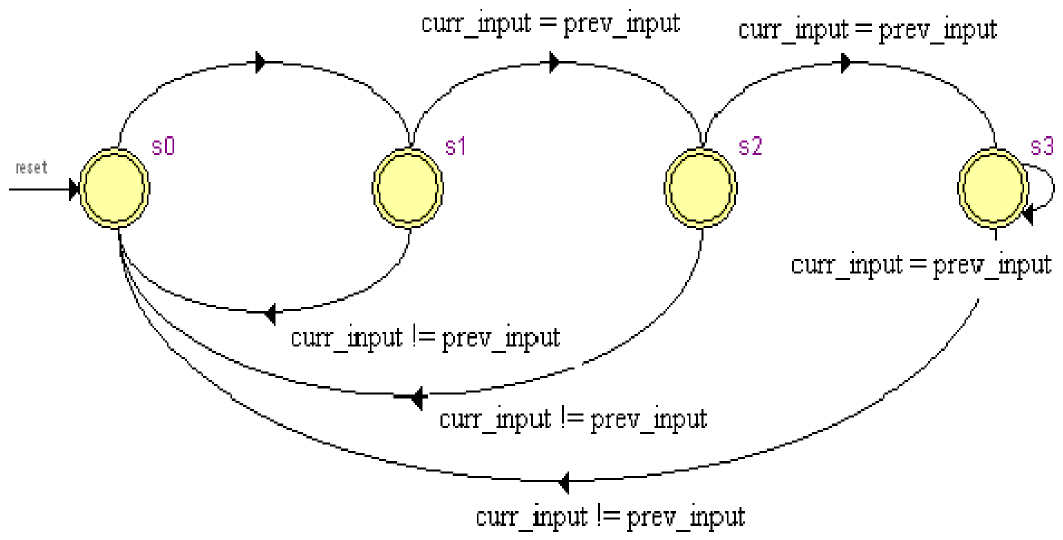


Figure 4: State diagram for the input detection block

2.2.2 Next Sequence Detection

This hardware block is included in the design to detect the valid sequence in incoming data. For every new input data arrival, valid sequence is one which has the topmost bit flipped from previous data value. This logic was realized using a state machine shown in Figure 5, with one state for logic high and another for logic low value for the topmost bit position of input. Output is generated when in transition from one state to another.

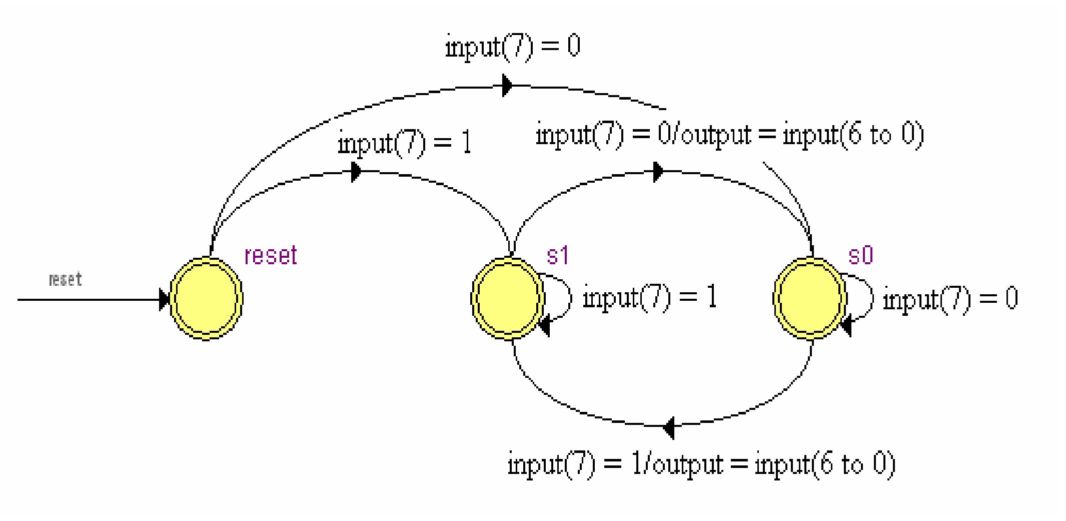

input(7) = 0
input(7) = 1
input(7) = 0/output = input(6 to 0)
reset
reset
s1
s0
input(7) = 1
input(7) = 0
input(7) = 1/output = input(6 to 0)

Figure 5: State diagram for next sequence detection block

2.2.3 FIFO Buffer

The FIFO queue is a temporary memory formed by a group of flip flops for buffering input data. It was used in this design to cover the mismatch in frequency at which incoming data arrives and the clock at which it is processed. One might assume that the relatively high clock rate for processing within the FPGA would make buffering unnecessary, but locking of memory locations can cause longer delays. Even though the probability of finishing the task on incoming data before arrival of next sequence is quite high, the FIFO buffer is needed in the case where the processing unit could not finish the task in time. The FIFO is formed with a set of read, write pointer, storage and control logic. The depth of the FIFO is currently 8, but generally should be set to cover the longest sequence that could be issued without handshake to a locked object in memory.

2.2.4 Processing Unit

The processing unit is a core block for the PE interface, responsible for interpreting each PE operation. It is fed by the FIFO buffer output and is allowed to access the shared resources by the arbitration control logic. It also includes control logic to send signals to a bi-color LED used for indicating the PE processing status. The color display from the LED is simple and intuitive:

- Green: running
- Red: waiting
- Black (dark): PE not active

2.2.5 Shared Resources Control Logic

In this design, multiple processors share access to a single memory module, so there is a need to arbitrate among the contending processors. Several methods exist to decide which one of the processors gets access and which ones have to wait.

Token ring logic is one of the smarter methods for access control. In a token ring, the processors are treated as though they were organized in a ring. One processor generates an original token, which is passed between processors to control access. The processor that holds the token is permitted to access the shared structure during that clock cycle. At the end of that clock cycle, the processor that holds the token passes it to the next processor. Because the literally next processor in the ring might not need to make an access, the token can instead be passed to the next processor that is requesting access – perhaps bypassing one or more idle processors. In this way, every clock cycle of the shared resource is allocated to useful work. If there are no processors requesting access, the token is logically passed through the entire ring, ending in the same processor that had it in the previous clock cycle.

A somewhat simpler, although potentially less efficient, arbitration policy is round-robin scheduling. This arbitration method is equivalent to the ring mechanism described above in the case that all processors are always requesting access to the shared resource. However, by using a simple modular counter and giving access to the processor whose number matches the current count, round-robin always allocates a clock cycle to each processor even if the processor does not need access. The performance difference between round-robin and token ring arbitration is expected to be irrelevant in the current design given the large ratio between PE port speed and internal clock speed.

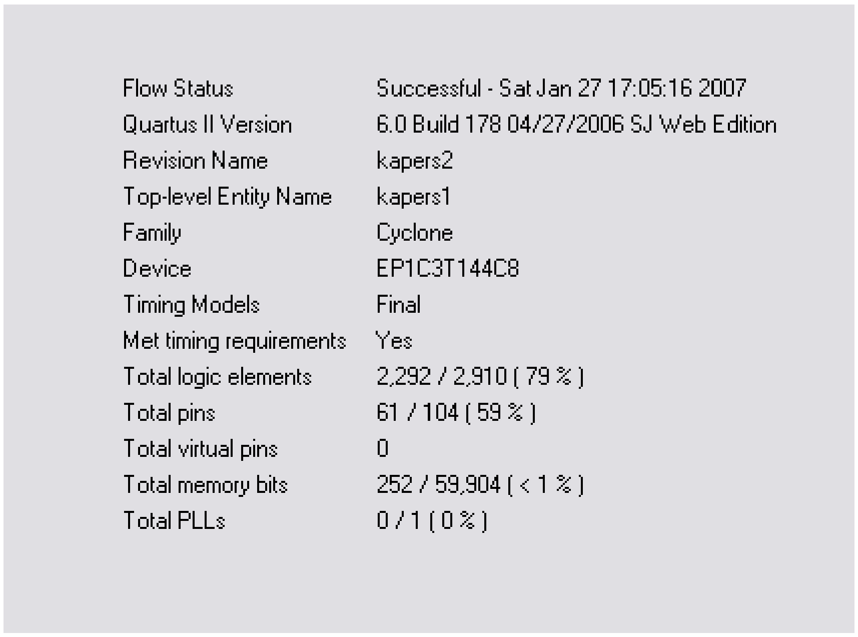
2.2.6 Shared Memory

The shared memory is responsible not only for storing values, but also for performing atomic read-modify-write operations using an ALU in the memory interface.

Fundamentally, there is no reason that the memory cannot be implemented using the block RAM of an Altera FPGA. However, for single-cycle performance, the block RAM needs to be able to output the old value of the addressed memory cell and latch a new value at the end of the same clock cycle. The Altera FPGA used is capable of this functionality using the "mixed-port read-during-write" memory feature, however, this feature is only accessible using the MegaWizard Plug-In Manager to configure the block RAM – and that software was not freely available. Thus, the memory was implemented using flip-flops, consuming a significant number of FPGA LEs (Logic Elements). The resulting performance is essentially identical, but the amount of memory space and room for logic expansion differ dramatically.

2.3 Results

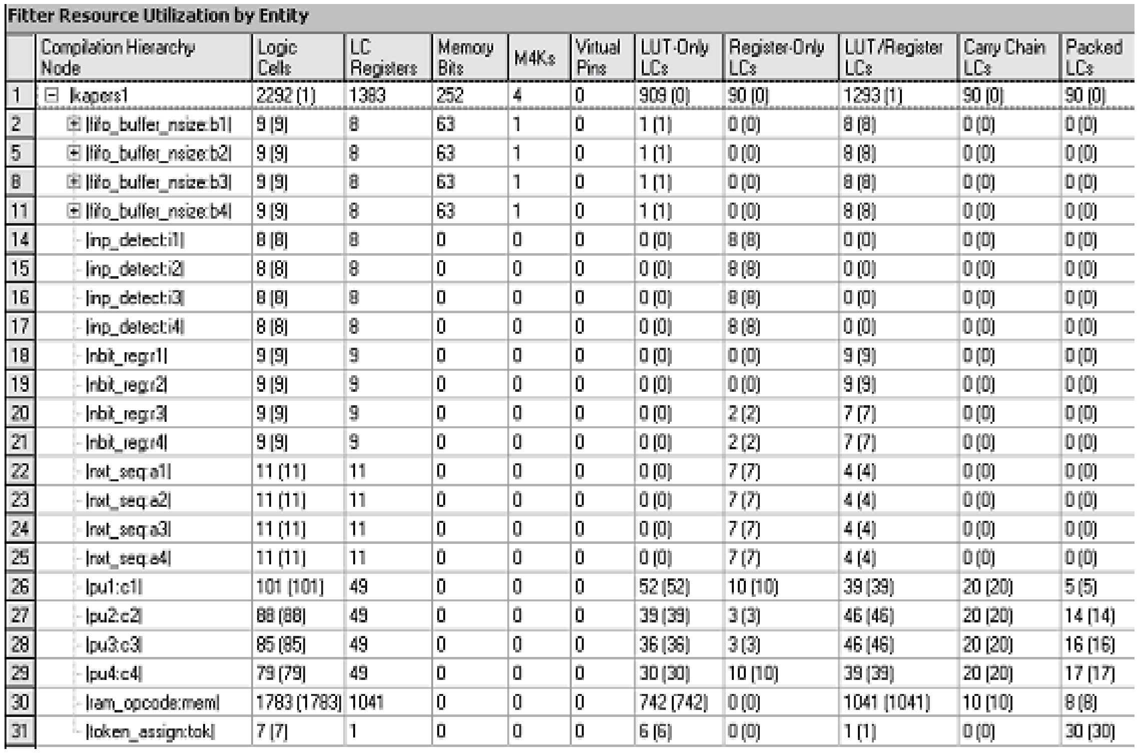
As suggested in the previous section, the hardware design could make use of either the block RAM or flip-flops for constructing the memory, but lack of the configuration software prevented us from making a functional block RAM version. The compilation report for the complete design using flip-flops to build the memory is given in the screenshot of Figure 6. This design provided 256 nybbles of memory without using block RAM to implement any of the AFN memory. A simple variation using a (slightly non-functional) approximation to the block RAM-based memory design revealed that memory space could be expanded to about 13K nybbles using block RAM instead of flip-flops. The fully functional flip-flop version uses about 80% of the available logic circuitry, with only 252 bits of block RAM used (to implement the FIFO buffers).



```
Flow Status                  Successful - Sat Jan 27 17:05:16 2007
Quartus II Version           6.0 Build 178 04/27/2006 SJ Web Edition
Revision Name                kapers2
Top-level Entity Name        kapers1
Family                       Cyclone
Device                       EP1C3T144C8
Timing Models                Final
Met timing requirements      Yes
Total logic elements         2,292 / 2,910 ( 79 % )
Total pins                   61 / 104 ( 59 % )
Total virtual pins           0
Total memory bits            252 / 59,904 ( < 1 % )
Total PLLs                   0 / 1 ( 0 % )
```

Figure 6: Compilation report for VHDL design

This hardware VHDL design was finalized for programming the FPGA. Top level entity code for the design is in Appendix B. Figure 7 shows the detailed usage of resources by entity. The ALU-augmented memory uses the majority of the logic resources. Clock frequency was set at 50MHz (period = 20ns), which is the maximum allowable for correct operation of all the paths in the circuitry. The design was tested at the block level, and as a whole, for functionality. Simulation results proving the correctness of the design are in Appendix A.



Fitter Resource Utilization by Entity

	Compilation Hierarchy Node	Logic Cells	LC Registers	Memory Bits	M4Ks	Virtual Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs	Carry Chain LCs	Packed LCs
1	⊟ |kapers1	2292 (1)	1383	252	4	0	909 (0)	90 (0)	1293 (1)	90 (0)	90 (0)
2	⊞ |lifo_buffer_nsize:b1|	9 (9)	8	63	1	0	1 (1)	0 (0)	8 (8)	0 (0)	0 (0)
5	⊞ |lifo_buffer_nsize:b2|	9 (9)	8	63	1	0	1 (1)	0 (0)	8 (8)	0 (0)	0 (0)
8	⊞ |lifo_buffer_nsize:b3|	9 (9)	8	63	1	0	1 (1)	0 (0)	8 (8)	0 (0)	0 (0)
11	⊞ |lifo_buffer_nsize:b4|	9 (9)	8	63	1	0	1 (1)	0 (0)	8 (8)	0 (0)	0 (0)
14	|inp_detect:i1|	8 (8)	8	0	0	0	0 (0)	8 (8)	0 (0)	0 (0)	0 (0)
15	|inp_detect:i2|	8 (8)	8	0	0	0	0 (0)	8 (8)	0 (0)	0 (0)	0 (0)
16	|inp_detect:i3|	8 (8)	8	0	0	0	0 (0)	8 (8)	0 (0)	0 (0)	0 (0)
17	|inp_detect:i4|	8 (8)	8	0	0	0	0 (0)	8 (8)	0 (0)	0 (0)	0 (0)
18	|nbit_reg:r1|	9 (9)	9	0	0	0	0 (0)	0 (0)	9 (9)	0 (0)	0 (0)
19	|nbit_reg:r2|	9 (9)	9	0	0	0	0 (0)	0 (0)	9 (9)	0 (0)	0 (0)
20	|nbit_reg:r3|	9 (9)	9	0	0	0	0 (0)	2 (2)	7 (7)	0 (0)	0 (0)
21	|nbit_reg:r4|	9 (9)	9	0	0	0	0 (0)	2 (2)	7 (7)	0 (0)	0 (0)
22	|nxt_seq:a1|	11 (11)	11	0	0	0	0 (0)	7 (7)	4 (4)	0 (0)	0 (0)
23	|nxt_seq:a2|	11 (11)	11	0	0	0	0 (0)	7 (7)	4 (4)	0 (0)	0 (0)
24	|nxt_seq:a3|	11 (11)	11	0	0	0	0 (0)	7 (7)	4 (4)	0 (0)	0 (0)
25	|nxt_seq:a4|	11 (11)	11	0	0	0	0 (0)	7 (7)	4 (4)	0 (0)	0 (0)
26	|pu1:c1|	101 (101)	49	0	0	0	52 (52)	10 (10)	39 (39)	20 (20)	5 (5)
27	|pu2:c2|	88 (88)	49	0	0	0	39 (39)	3 (3)	46 (46)	20 (20)	14 (14)
28	|pu3:c3|	85 (85)	49	0	0	0	36 (36)	3 (3)	46 (46)	20 (20)	16 (16)
29	|pu4:c4|	79 (79)	49	0	0	0	30 (30)	10 (10)	39 (39)	20 (20)	17 (17)
30	|ram_opcode:mem|	1783 (1783)	1041	0	0	0	742 (742)	0 (0)	1041 (1041)	10 (10)	8 (8)
31	|token_assign:tok|	7 (7)	1	0	0	0	6 (6)	0 (0)	1 (1)	0 (0)	30 (30)

Figure 7: FPGA resource utilization by each block

Chapter3

ALGORITHMS

Aggregate operations are computed inside the AFN, which collects data from all the PEs, performs the requested operation on that data, and sends the result to the requesting PEs. Basic aggregate operations are grouped into associative reductions and scans, collective communications, voting and scheduling. These operations are well defined in the AFAPI (Aggregate Function Application Program Interface)[8][9] for all the previously developed versions of AFNs[3]. In order to take advantage of the new higher-level architectural memory model in the KAPERS AFN, new algorithms for these operations needed to be defined. The following sections briefly describe the new algorithms that were developed. Portions of the C code used to construct these algorithms is included for better understanding. Additional details for the code can be found in Appendix C.

3.1 Associative Reduction Operations

Reduction is an aggregate function in which all the data from the nodes is reduced to a single value using the operation specified. Aggregate operations that come under this category are bitwise logical operations such as OR and XOR, and arithmetic operations like minimum, maximum, addition (summation), and multiplication (product) that are associative in nature, implying their ability to execute in parallel even if rearranged.

3.1.1 Logical Reductions

OR reduction can be basically implemented with the barrier operation *BarOr,* which, in addition to synchronization, does the logical OR of the data from the PEs and sends it as the result. This operation needs only two I/O communication cycles for any number of PEs, but is limited by the data path size. The other way to do the reduction is by using the unlocked memory operation *Or*, where each PE sends outs its data to get it ORed with the current value at the assigned memory location. This will result in the OR operation of all the data from PEs by the time every one finishes up. A point to be noted is that all relevant memory locations should be cleared before performing the *Or* operation; it is sufficient to select one particular PE to clear the memory instead of all PEs clearing. The clear operation will be followed by a barrier synchronization signifying the completion of memory initialization operation to all other PEs.

```
#define P_DECLARE(type)\
p_##type \
p_reduceor##type(register p_##type d)\
{\      register p_##type t;\
        p__address(D_ADDRESS);\
        if (IPROC == CPROC) {
                p_setfunc(xchg, unlock);
                p_memfunc(0);}
        p__setfunc(or,unlock);\
        p__memfunc(d);\
        p__baror(0);\
        t = p__memfunc((p_##type) 0);\
        p__baror(0);\
        return(t);\
}
P_DECLARE(8)
P_DECLARE(8u)
P_DECLARE(16)
P_DECLARE(16u)
P_DECLARE(32)
P_DECLARE(32u)
P_DECLARE(64)
P_DECLARE(64u)
#undef P_DECLARE
```

Algorithm 1: Associative OR operation

Reduction for XOR can be done in the same way as OR by setting the memory operation to *Xor* instead of *Or*. This reduction algorithm can be used for any type and size of data, but is limited to one nybble at a time by the parallel port interface. XOR reduction was not implemented in hardware in previous AFNs, nor was the data permitted to be of arbitrarily many nybbles precision.

3.1.2 Minimum and Maximum Reductions

Minimum and maximum operations in AFN memory can be accomplished somewhat similarly to OR and XOR, however, there is a major difference. Bitwise reductions are not sensitive to nybble order of evaluation; it makes no difference whether the low or high nybble of each object is sent first. In minimum and maximum, it is necessary to consider the nybbles in high-to-low order, disabling transmission of data for a node once that node's value is known not to be the answer.

Each nybble sent from a PE is compared with the value in the corresponding memory location and the unsigned minimum of these values gets placed in the location. The old value from memory is sent to the PE as the result before writing the memory with the minimum value. Thus, the result can be compared with the data sent by the node and, if the result is same as the sent data, there is a need for sending the next nybble; if it is less, there isn't a need for sending any further nybbles and the node must wait for the end result to be read from memory formed by other nodes in the group. The problem is that, without locking, this method requires synchronization of the nodes for every nybble sent before going to next nybble, thus adding an unwanted delay in the execution.

A more efficient way to do this reduction is by using the locking feature of the KAPERS hardware. A PE is allowed to lock (reserve) a memory location, have exclusive access to that memory location for one operation, and then unlocks the memory cell after it has been updated. The first PE to reach the assigned memory cell will write its data, whereas the following PEs will do the memory *Min* operation with locking to line up the PEs for getting appropriate results. The nybble operations by a PE are entirely asynchronous to other PEs in group. Within each node, each sent nybble is compared with the result to decide if sending further data for processing is necessary. However, there is a need to barrier synchronize for the first sent nybble to make sure that the processor to arrive first is ahead of all others in writing the data in memory. This algorithm can be used as such for any size of unsigned integer.

For signed integers the above unsigned algorithm can be used with addition of a bias value (which is 2^{K-1} for a signed K bit number) to the signed integer before sending the data and subtracting the same from the result later.

```
#define P_DECLARE(type)\
p_##type \
p_reducemin##type(register p_##type d)\
{\
        register p_##type r,t;\
        register p_##type prv_r = 0;\
        register int n = (sizeof(d) * 2);\
        p__address(D_ADDRESS);\
        p__setfunc(xchg,unlock);\
        r = p__memnext(prv_r+1);\
        if(r == prv_r) {\
                p__setfunc(xchg,lock);\
                p__address(D_ADDRESS + 1);\
                p__memnext((d >> 4*(n-1)) & 0xf);\
                p__baror(0);\
                p__address(D_ADDRESS + 2);\
                do {
                        p__memnext((d >> 4*(n-1)) & 0xf);\
                        n--;\
                } while(n != 1);\
                p__memlast(d & 0xf);\
        }\
        else {\
                p__setfunc(min,lock);\
                p__baror(0);\
                p__address(D_ADDRESS + 1);\
                register prv_value = p__memnext((d >> 4*(n-1)) & 0xf);\
                do {\
                                if (prv_value >= ((d >> 4*(n-1)) & 0xf)) {\
                                if (n != 1) {
                                prv_value = p__memnext((d >> 4*(n-1)) & 0xf);\
                                n--;\
                                }\
                                else {\
                                prv_value  =  p__memlast((d  >>  4*(n-1))  &  0xf);\
                                }\
                        }\
                        else {\
                        break;\
                        }\
                } while(n != 1);\
                p__setfunc(xchg,unlock);\
        }\
        p__baror(0);\
        t = p__memfunc(0);\
        p__baror(0);\
        return(t);\
}
P_DECLARE(8u)
P_DECLARE(16u)
P_DECLARE(32u)
P_DECLARE(64u)
#undef P_DECLARE
```

Algorithm 2: Associative minimum reduction for integer numbers

Table 1: Bit format for floating point numbers

	SIGN	**EXPONENT**	**MANTISSA**
Single Precision	1[31]	8[30-23]	23[22-0]
Double Precision	1[63]	11[62-52]	52[51-0]

The IEEE representation for floating point numbers can be treated as ordinary binary representation as it has the layout of sign bit followed by exponent bits, and then by the mantissa bits – all of which are in the order of significance needed for comparison. To use the above algorithm, the changes that need to be done on the floating point numbers to map them as unsigned integer numbers are that the sign bit needs to be flipped in case of positive numbers and all the bits for negative numbers. The result will be an integer number that must be transformed in the inverse way to get back to the

```
p_reducemin32f(p_32f d)
{
        register float *fptr;
        fptr = &d;
        register unsigned int *iptr;
        iptr = (unsigned int *) fptr;
        register int a = *iptr;
         /* adjusting the exponent and sign bits */
        if ((a >> 31) == 0) {
                a ^= 0x80000000;}
        else {
                a = ~a;}
         /* finding minimum value */
         int float_min = p_reducemin32u(a);
         /* getting back the old value */
        if ((float_min >> 31) == 1) {
                float_min ^= 0x80000000;}
        else {
                float_min = ~float_min;}
        return(*(float *)&float_min);}
```

Algorithm 3: Associative minimum reduction for floating point numbers

standard floating point format. Maximum reductions can be implemented using the Minimum routines by sending bits inverted and reformatting the final result.

3.1.3 Addition Reduction

As for minimum and maximum, nybble order in AFN memory is significant for addition – but the desired nybble order is the reverse: from low nybble to high nybble. Addition can be done using the memory function *Add* which can operate on nybble-wide data at one time. Multi-nybble data also can be added without any overhead for keeping track of carry from earlier addition operations. The carry generated from adding the nybbles is recorded by the hardware using one carry bit per node interface, thus correctly implementing carry independent of the number of nodes participating.

```
#define P_DECLARE(type)\
p_##type \
p_reduceadd##type(p_##type d)\
{\
        register p_##type t;\
        p__address(D_ADDRESS);\
        p__setfunc(add,unlock);\
        p__memfunc(d);\
        p__baror(0);\
        p__setfunc(xchg,unlock);\
        t = p__memfunc((p_##type) 0);\
        return(t);\
}
P_DECLARE(8u)
P_DECLARE(16u)
P_DECLARE(32u)
P_DECLARE(64u)
P_DECLARE(8)
P_DECLARE(16)
P_DECLARE(32)
P_DECLARE(64)
#undef P_DECLARE
```

Algorithm 4: Reduction for addition

Signed integers are stored in two's compliment notation in the processors, and so the unsigned integer algorithm for addition can be used for the signed integers. Addition of a pair of integers in two's compliment notation is functionally equivalent as addition of unsigned integers except for overflow detection – which is not required by most programming languages and is not performed in the KAPERS AFN.

A simple procedure to add floating point numbers is by sending them as integer values nybble by nybble, but a better way to do addition is by reducing the dynamic range for the floating addition by scaling down the integer to the maximum exponent in the group. For this, first the exponent values of the floating point numbers to be added from all PEs are sent to the AFN and maximum value is computed, then it is followed by adjustment of the mantissa to the maximum exponent. Next a reduction addition of all the mantissas that were scaled will follow in the AFN. Finally resultant sum is normalized and adjusted for the exponent, to get back to the floating point notation. This is a very different algorithm from those used in earlier AFN designs, which essentially performed only integer addition in the network or, for the simpler AFN designs, performed addition on the nodes themselves.

```
p_32f p_reduceadd32f(p_32f d)
{
        register int s_mant, b_mant;
        register int s = 0;
        register int incr_exp,decr_exp = 0;
        int float_add;
        register int max_exp = 0;
        register int e_d = 0;
        register int j = 0;
        register float *fptr;
        register unsigned int *iptr;
        register int data;
        fptr = &d;
        iptr = (unsigned int *) fptr;
        data = *iptr;
        register int sign = (data >> 31) & 0x1;
        register int exp = (data >> 23) & 0xff;
        register int mantissa = (data & 0x007fffff) | 0x00800000;
        /* adjusting the mantissa to the maximum exponent in common */
        max_exp = p_reducemax16u(exp);
        register int un_exp = exp - max_exp;
        while (un_exp < 0) {
                mantissa = (mantissa >> 1) & 0xffffffff;
                un_exp++;}
        /* adding value by biasing the mantissa */
        if(!sign) {
                s_mant = ((sign << 25) & (0x01000000))
                                | (mantissa & 0x00ffffff);
                b_mant = s_mant + 0x01ffffff;
        }
        else {
                s_mant = ((sign << 25) & (0x01000000))
                                | (mantissa & 0x00ffffff);
                b_mant = ~s_mant & 0x01ffffff;
        }
        /* unbiasing the sum */
        register int add_mant = p_reduceadd32u(b_mant) - (NPROC * 0x01ffffff);
        if(((add_mant >> 31) & (0x00000001)) == 1) {
                add_mant = ~add_mant + 0x00000001;
                s = 1;
        }
        /* normalizing the mantissa */
        register int e_i = ((add_mant >> 23) & (0x07f));
        if (e_i != 0) {
                register int i = 30;
                do {
                e_i = ((add_mant >> i) & (0x1));
                incr_exp = i – 23;
                I = I-1;
        } while (e_i == 0 & i >=24);
        }
```

```
else if (incr_exp == 0) {
        e_d = (add_mant >> 22) & 0x001;
        j = 21;
        if (e_d == 0) {
                do {
                e_d = ((add_mant >> j) & (0x00000001));
                if (e_d == 0 & j == 0) {
                /* For zero mantissa case*/
                decr_exp = 0;}
                else {
                decr_exp = decr_exp + 1;}
                j = j – 1;
                }while(e_d == 0 & j >= 0);
        }
}
/* making up the float result */
register int result_exp = max_exp + incr_exp – decr_exp;
if(incr_exp > 0) {
        float_add = (((s << 31) & (1 << 31)) |
                ((result_exp << 23) & (0x7f800000)) |
                ((add_mant >> incr_exp) & ((1 << 31) - 1)));   }
else if (decr_exp > 0) {
        float_add = (((s << 31) & (1 << 31)) |
                ((result_exp << 23) & (0x7f800000)) |
                ((add_mant << decr_exp) & ((1 << 31) - 1)));  }
else {
        float_add = (((s << 31) & (1 << 31)) |
                ((result_exp << 23) & (0x7f800000)) |
                ((add_mant >> incr_exp) & ((1 << 31) - 1)));   }
        return(*(float *)&float_add);}
```

Algorithm 5: Floating point Addition

3.1.4 Multiplication Reduction

Although addition and minimum have been implemented in some earlier AFNs, multiplication never was. For performing multiplication operations using multipliers we need to add more circuitry in the hardware and the system tends to become complex. Further, multiplication cannot be performed using traditional methods on a nybble-at-a-time basis; the hardware would need to know the size of the result. However, multiplication can be done in logarithmic number system (LNS) with simple adders which usually fit in the hardware with less complexity. Better still, the multiply then has the incremental properties of addition, making nybble-at-a-time processing possible no matter how high the precision of the values.

This method of multiplication using LNS was chosen for the current AFN as it has the adders already defined in the hardware and the nodes contain relatively powerful processors that easily can perform conversions between conventional integer or floating-point values and LNS representations in a small fraction of the network communication time. Each PE sends out the normalized logarithmic value of the data to be multiplied, the hardware does the reduction addition of all the values, the result is returned. The result is then converted from LNS into the node's native format using the node's relatively fast and powerful processor.

The primary concern here is the ability for product operations performed on LNS values to yield precisely the same values that conventional integer or floating-point product would have yielded. Appropriate scaling methods and precisions using an empirical search procedure were determined to ensure that integer results would be exact despite any conversion or rounding errors. Fundamentally, this was a matter of finding a scaling method by which LNS values can be converted into unsigned integers with sufficient precision. In general, the unsigned integers coding LNS values must have somewhat higher precision than the conventionally represented values in order to obtain identical product values.

LNS representations have two complications. The first is the representation of the value zero, which has no logarithmic value. Actually, representation of zero is a special case in floating-point too, because zero cannot be normalized to have a 1 in the most

```
#define P_DECLARE(type) \
p_##type p_reducemul##type(p_##type d)\
{\
        register p_##type t;\
        register int n;\
        if (d == 0) {\
                p_bcastput##type(0);\
                return(0);\
        }\
        if (p_bcastget##type() != 0) {\
                n = (log2(d) * 10000000);\
                t = p_reduceadd64u(n);\
                return(ceil(pow(2.0, t/10000000)));\
        }\
}
P_DECLARE(8u)
P_DECLARE(16u)
P_DECLARE(32u)
P_DECLARE(64u)
#undef P_DECLARE
```

Algorithm 6: Multiplication reduction for integers

significant mantissa bit. For the KAPERS AFN, zero does not need to be represented at all; instead, we can first check if any operand to the product is zero, which makes the result trivially zero. The other LNS complication is representation of the sign of the value. The sign of the product is equal to the XOR of the signs of the operands, so a simple 1-bit XOR aggregate function is used to determine the sign of the result for signed integers or floating-point values; only the absolute values are converted to LNS. The other way to compute the sign is by the least significant bit of the sum of the sign bits.

Floating point number multiplication is also done using LNS but data is sent in two steps first the exponent and then the normalized logarithmic mantissa because the data range for floating point numbers is too high to be normalized at once for good precision.

```
p_reducemul32f(p_32f d)
{
        register float *fptr;
        register unsigned int *iptr;
        register int data;
        fptr = &d;
        iptr = (unsigned int *) fptr;
        data = *iptr;
        register int expo = (data >> 23) & 0xff;
        register long long int mant = (data & 0x007fffff);
        register long long int mant_norm = (1 + mant/8388608)*10000000.0;

        /* Sending the exponent for addition */
        register int sum_exp = p_reduceadd32u(expo);
        p__baror(0);

        /* Sending the mantissa for addition */
        register long long int sum_mant = p_reduceadd64u(mant_norm);
        p__baror(0);
        return(powf(2.0, sum_exp)+ powf(2.0, sum_mant/10000000.0));
}
```

Algorithm 7: Multiplication reduction for floating point numbers

3.2 Scan operations

Scan operations are very similar to reduction operations and are sometimes called parallel prefix operations. A scan operation for a particular PE returns the value from the associative reduction of all the data from the first PE to that particular PE, i.e. the value returned to node N is the result of applying the specified associative operator on the values submitted by nodes 0 to N. Thus, each PE gets a different value for the same operation. Scan operations are similar to reduction operations when considered for a single PE.

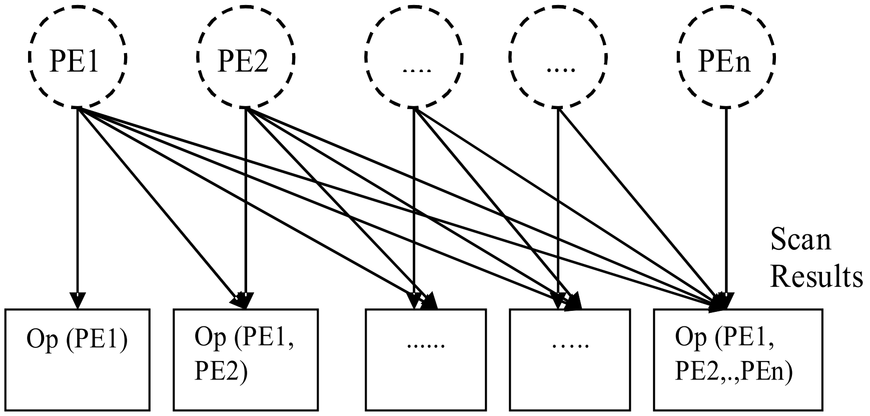


Figure 8: Scan operation

Scans can be efficiently done using the new lock feature for the memory that can impose an ordering on nybble accesses. The lowest numbered PE will be writing the data into the memory first and all others will do the required operations either logical or arithmetic in an orderly fashion with the aid of an extra memory location specifying the number of PE. The reference memory location signifying the PE number to follow is updated by each PE when it finishes its turn. The algorithm is very similar to those used for reduction operations except in that the order of PEs acquiring locks is forced for scans.

```
#define P_DECLARE(type)\
p_##type \
p_scanor##type(p_##type d)\
{\
        int mem_ref = 0;\
        p_##type result;\
        p__address(D_ADDRESS);\
        p__setfunc(xchg,unlock);\
        mem_ref = p__memfunc(0);\
        do {\
        if (IPROC == 0 & mem_ref == 0) {\
                mem_ref = p__memfunc(mem_ref+1);\
                p__setfunc(xchg,lock);\
                p__address(D_ADDRESS + 1);\
                p__memnext(d & 0xf);\
                 p__baror(0);\
                p__address(D_ADDRESS + 2);\
                p__memfunc(d >> 4);\
        }\
        else if (IPROC == mem_ref) {\
                mem_ref = p__memfunc(mem_ref+1);\
                p__setfunc(or,lock);\
                 p__baror(0);\
                p__address(D_ADDRESS + 1);\
                p__memfunc(d);\
        }\
        }while(mem_ref == (NPROC - 1));\
        if (NPROC == 1) {\
                p__address(D_ADDRESS + 1);\
                result = p__memfunc(0);\}\
        else {\
                result = p__memfunc(0);\}\
        return(result);\
}
```

```
P_DECLARE(1u)
P_DECLARE(8)
P_DECLARE(8u)
P_DECLARE(16)
P_DECLARE(16u)
P_DECLARE(32)
P_DECLARE(32u)
P_DECLARE(64)
P_DECLARE(64u)
#undef P_DECLARE
```

Algorithm 8: Scan Operation

3.3 Communication Operations

These operations are intended for sending or receiving data in between processors. With the addition of shared memory to hardware these operations can now be efficiently implemented by storing and retrieving the value from the memory asynchronously. These algorithms are relatively straightforward, essentially mimicking the behavior of a more conventional shared memory communication. Alternatively, the algorithms used by TTL_PAPERS can be applied with BarOr substituted for p__nand.

3.3.1 Broadcast put and get

Broadcast put operation places the data in a specific memory location, from which the value is subsequently read by all other nodes.

```
p_8u
p_bcastget8u(void)
{
        p__address(D_ADDRESS);
        p__setfunc(or,unlock);
        return(p__memfunc((p_8u) 0));
}
#define P_DECLARE(type)\
p_##type \
p_bcastget##type(void)\
{\
        register p_##type d = 0;\
        register int n = sizeof(p_##type)-1;\
        d = p_bcastget8u();\
        while (n > 0) {\
                d = (d << 8) | p_bcastget8u();\
                n--;\
        }\
        return(d);\
}
P_DECLARE(16u)
P_DECLARE(32u)
P_DECLARE(64u)
P_DECLARE(8)
P_DECLARE(16)
P_DECLARE(32)
P_DECLARE(64)
#undef P_DECLARE
```

Algorithm 9: Broadcastget Operation

```
void
p_bcastput8u(register p_8u d)
{
        p__address(D_ADDRESS);
        p__setfunc(xchg,unlock);
        p__memfunc(d);
        p__baror(0);
}

#define P_DECLARE(type)\
void \
p_bcastput##type(register p_##type d)\
{\
        register int n = sizeof(d) - 1;\
        p_bcastput8u(d & 0xff);\
        while (n > 0) {\
        p_bcastput8u((d >> 8) & 0xff);\
        }\
}

P_DECLARE(16u)
P_DECLARE(32u)
P_DECLARE(64u)
P_DECLARE(8)
P_DECLARE(16)
P_DECLARE(32)
P_DECLARE(64)
#undef P_DECLARE
```

Algorithm 10: Broadcastput Operation

3.3.2 Putget

Involves writing the data to the memory location assigned to the PE and reading back the data from the desired PE's assigned memory location.

```
p_##type \
p_putget##type(register p_##type d,register int source) \
{\
        register int n = sizeof(d)*2;\
        p__address(D_ADDRESS + IPROC*n);\
        p__setfunc(xchg,unlock);\
        p__memfunc(d);\
        p__baror(0);\
        p__address(D_ADDRESS + source*n);\
        p__setfunc(or,unlock);\
        return(p__memfunc((p_##type) 0));\
}

P_DECLARE(bar)
P_DECLARE(8)
P_DECLARE(8u)
P_DECLARE(16)
P_DECLARE(16u)
P_DECLARE(32)
P_DECLARE(32u)
P_DECLARE(64)
P_DECLARE(64u)
P_DECLARE(32f)
P_DECLARE(64f)

#undef P_DECLARE
```

Algorithm 11: Putget Operation

3.4 Voting and Scheduling Operations

Voting and Scheduling methods include operations that can be used by a PE to know the status of other PEs relative to its own.

3.4.1 First

This operation should return the minimum PE number which has first arrived with a value '1', and is done by sending the “PE number” instead of the data if PE has a value of '1' and performing the memory minimum operation on them in the hardware. Barrier synchronization is done to make sure everyone is finished before reading the result.

```
int
p_first(register p_1u d)
{
        p__address(D_ADDRESS);
        p__setfunc(min,unlock);
        if (d == 1) {
        p__memfunc(IPROC);
        }
        p__baror(0);
        return(p__memfunc(NPROC+1));
}
```

Algorithm 12: First

3.4.2 Count and Quantify

Count operation has to get the total number of the PEs that have the value '1'. This is done by using the addition function in the memory, where the PEs increment the count in the memory only if they have a '1'. Quantify is same as the 'Count' operation but outputs the result as generalized statement (none/one/greater than one) instead of exact count. The result could be categorized at nodes after the count operation gets finished.

```
int
p_count(register p_1u d)
{
        p__address(D_ADDRESS);
        p__setfunc(add,unlock);
        if (d == 1) {
                p__memfunc(d);
        }
        p__baror(0);
        return(p__memfunc(0));
}
```

Algorithm 13: Count and Quantify operations

3.4.3 Votecount

Votecount has to count the number of PEs who voted for the PE that is performing the operation. Implemented by assigning separate memory location for each PE where the votes are summed by memory addition operation. PEs go to the corresponding location where the counter is set for the PE it is supposed to vote and increment the value. Barrier synchronization is done to make sure all PEs are finished with voting before the retrieval of count from the location where PE's votes are cast.

```
int
p_votecount(register int d)
{
        p__address(D_ADDRESS + d*sizeof(NPROC)*2);
        p__setfunc(add,unlock);
        p__memfunc(1);
        p__baror(0);
        p__address(D_ADDRESS + IPROC*sizeof(NPROC)*2);
        return(p__memfunc(0));
}
```

Algorithm 14: Votecount

3.4.4 Vote

Result for this operation is a vector formed by all PEs signifying votes for the PE doing the vote operation. This vector will have a value '1' positioned corresponding to the PE which voted for this PE. For this, each PE is allocated a memory space enough to fit the vector that needs to be formed. Every PE goes to the memory location of the vote vector of the PE it wants to vote for and ORs in a '1' in its position in the vector. The result is obtained by reading the appropriate memory location after barrier synchronization.

```
p_bar
p_vote(register int d)
{
        register int x = 0;
        register p_bar mask;

        p__address(D_ADDRESS + d*ceil(NPROC/4) +ceil((IPROC+1)/4) - 1);
        p__setfunc(or,unlock);
        p__memnext(1 << (IPROC % 4));
        p__baror(0);
        p__address(D_ADDRESS + IPROC*ceil(NPROC/4));
        return(p__memfunc((p_bar) 0));
}
```

Algorithm 15: Vote operation

Chapter4

HARDWARE IMPLEMENTATION

This chapter describes a prototype board implementation(Figure 9) supporting the new architectural model. The Altera Cyclone[23] FPGA was the central component on the board, planned to work at a maximum clock frequency of 50MHz. This is not fast by current logic standards, but is more than sufficient in comparison to the effectively ~1MHz rate at which new commands and data can be transmitted by a node via the parallel port connection. The schematic for the board was captured initially to check for the functionality and then layout for two layers was drawn with signal traces routed manually. PCBs were ordered with the Gerber and drill files generated by the layout software.

4.1 Overview

The following sections detail the significance of components used in the prototype design. The schematic and layout information for the board are in next sections.

4.1.1 FPGA

This prototype of the KAPERS AFN hardware is built with the Altera's Cyclone EP1C3 series[18] FPGA as the key component. Cyclone devices contain a two-dimensional row- and column-based architecture to implement custom logic. Column and row interconnects of varying speeds provide signal interconnects between LABs and embedded memory blocks. The logic array consists of LABs, with 10 LEs in each LAB. An LE is a small unit of logic consisting of a 4-input lookup table (LUT) and flip-flop intended primarily for efficient implementation of user logic functions. M4K RAM blocks are true dual-port memory blocks with 4K bits of memory plus parity (4,608 bits) that are grouped into columns across the device in between certain LABs. Each Cyclone device I/O pin is fed by an I/O element (IOE) located at the ends of LAB rows and columns around the periphery of the device. Each IOE contains a bidirectional I/O buffer and three registers for registering input, output, and output-enable signals.

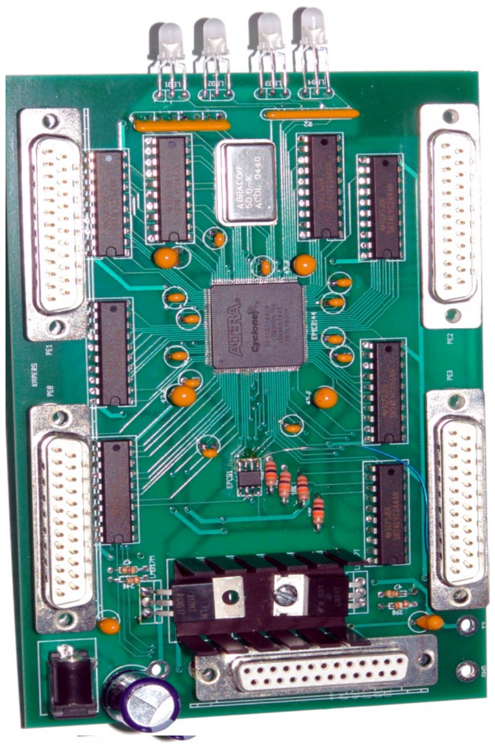


Figure 9: KAPERS prototype unit

Based on the requirement of logic, I/O pins, memory and soldering method, the EP1C3T144 seemed to provide the most cost-effective solution using a TQFP (Thin-quad flat package). This particular FPGA takes 22 mm x 22 mm of board space and has 2910 LEs and 13 blocks of M4K RAM memory blocks totaling 59,904 bits of RAM. This device has 144 pins, of which a maximum of 104 are available for user IO pins supporting LVTTL (3.3V low-voltage TTL-compatible) signaling. It provides one PLL for clock multiplication and phase shifting. This device has a global clock network with eight global clock lines that drive throughout the entire device: clocking for all resources within the device, such as IOEs, LEs, and memory blocks. There are various speed grades of the part available, with a maximum internal clock frequency of 275MHz for the -8 speed grade part used in KAPERS.

4.1.2 Configuration Device

There are three standard ways to configure an FPGA. We can use a cable from PC running the required software for sending the data to FPGA, a microcontroller on board with firmware to send data to the FPGA, or a flash memory device connected to the FPGA that automatically configures at power up. The flash memory method is most convenient for our purposes. The configuration device used is a serial, low cost, EPCS1 device with 1M bit of flash memory and in-system reprogramming capabilities. It only needs a four pin interface for communication. The configuration is done through AS (Active Serial) configuration mode in which the FPGA controls the configuration interface. The block diagram in the Figure 10 shows the AS configuration scheme.

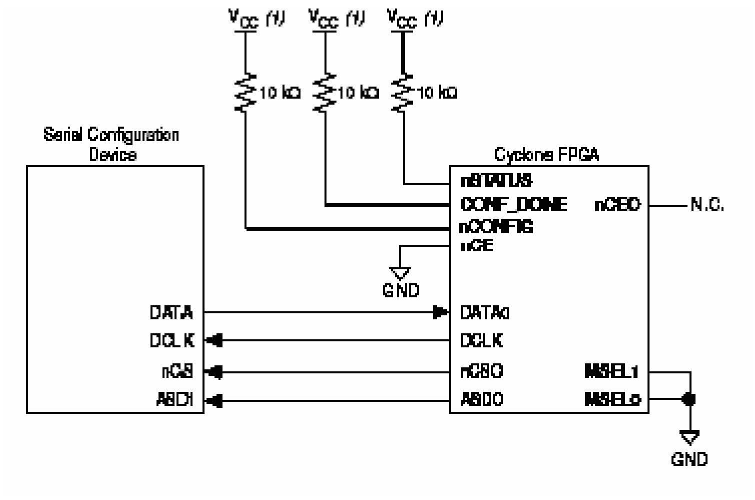


Figure 10: Serial configuration device and FPGA pins interface [18]

4.1.3 Decoupling Capacitors

Decoupling capacitors are a necessary part of the design for reliable operation. The placement and type of decoupling capacitors used in this design are shown in Figure 11:

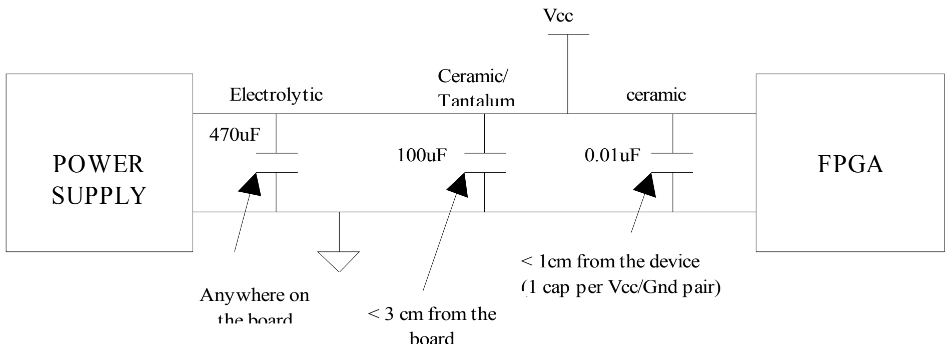


Figure 11: Decoupling capacitors

4.1.4 Level Shifters

The need for level shifters is obvious as there are two different logic levels in the design. The parallel port PC interface signals using TTL levels, while the FPGA IO signals use LVTTL. The best option for a cost effective solution for translating between 5V and 3V, in both input and output directions, was to employ LVC series level shifters.

4.1.5 Voltage regulators

The Cyclone FPGA needed two voltage levels, 1.5V for the core and 3.3V for I/O standard to be LVTTL. All other components in the design worked at 3.3V. Linear regulators from the LM317 series were chosen to regulate from 9V(drawn from a wall mount AC/DC adapter) to required voltage levels. These were not the parts recommended by Altera, but seemed a more effective solution; as discussed later, this choice might be responsible for the apparent power supply problems we encountered with the prototype hardware.

4.1.6 Byte Blaster II Cable replacement

To program and configure the Cyclone FPGA in AS configuration mode the only recommended and available download cable is Byte Blaster II from Altera, which is a bit expensive and not very common to find. The block diagram in Figure 12 shows the details of the Byte Blaster II cable whose only function is to drive the configuration data from a standard parallel printer port on the PC to the device on the PCB which is a serial configuration device in this particular case. The functionality of Byte Blaster II cable was met with an ordinary parallel cable with few modifications in the board layout.

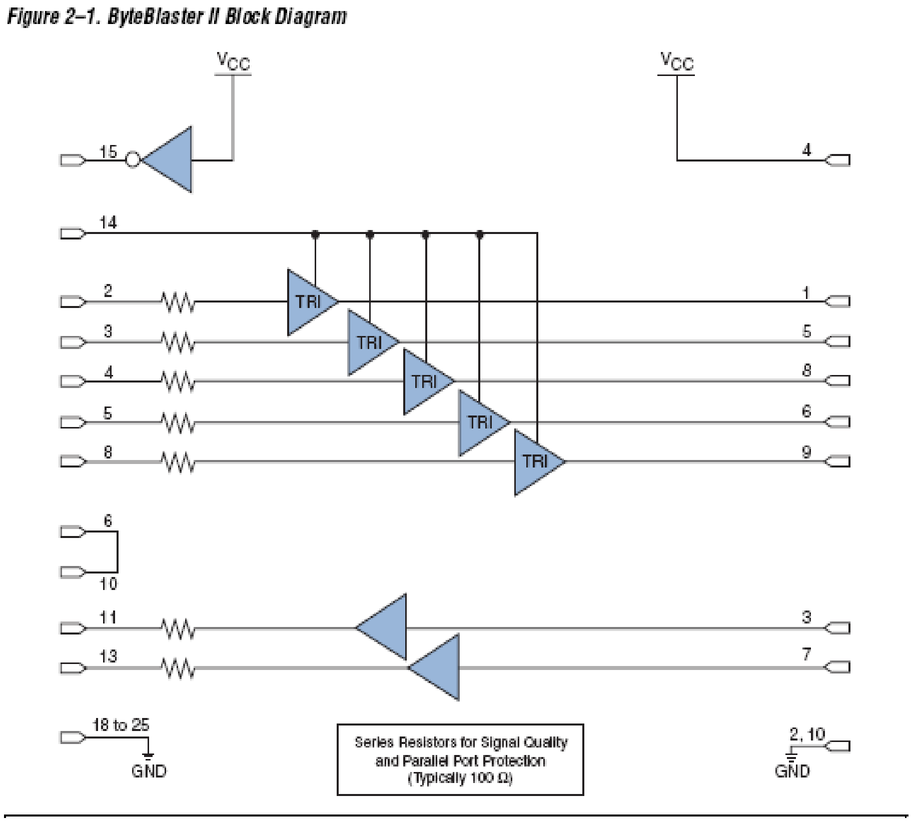


Figure 2–1. ByteBlaster II Block Diagram
VCC
VCC
15
4
14
2
3
4
5
8
TRI
1
5
8
6
9
6
10
11
13
3
7
18 to 25
GND
Series Resistors for Signal Quality and Parallel Port Protection (Typically 100 Ω)
2, 10
GND

Figure 12: Byte Blaster II block diagram [19]

4.2 Schematic

The schematic diagram for the KAPERS prototype shown in Figure 13 was captured in EAGLE 4.16r1 freeware software from CadSoft Online[20] to solidify the design and to facilitate electrical rule check.

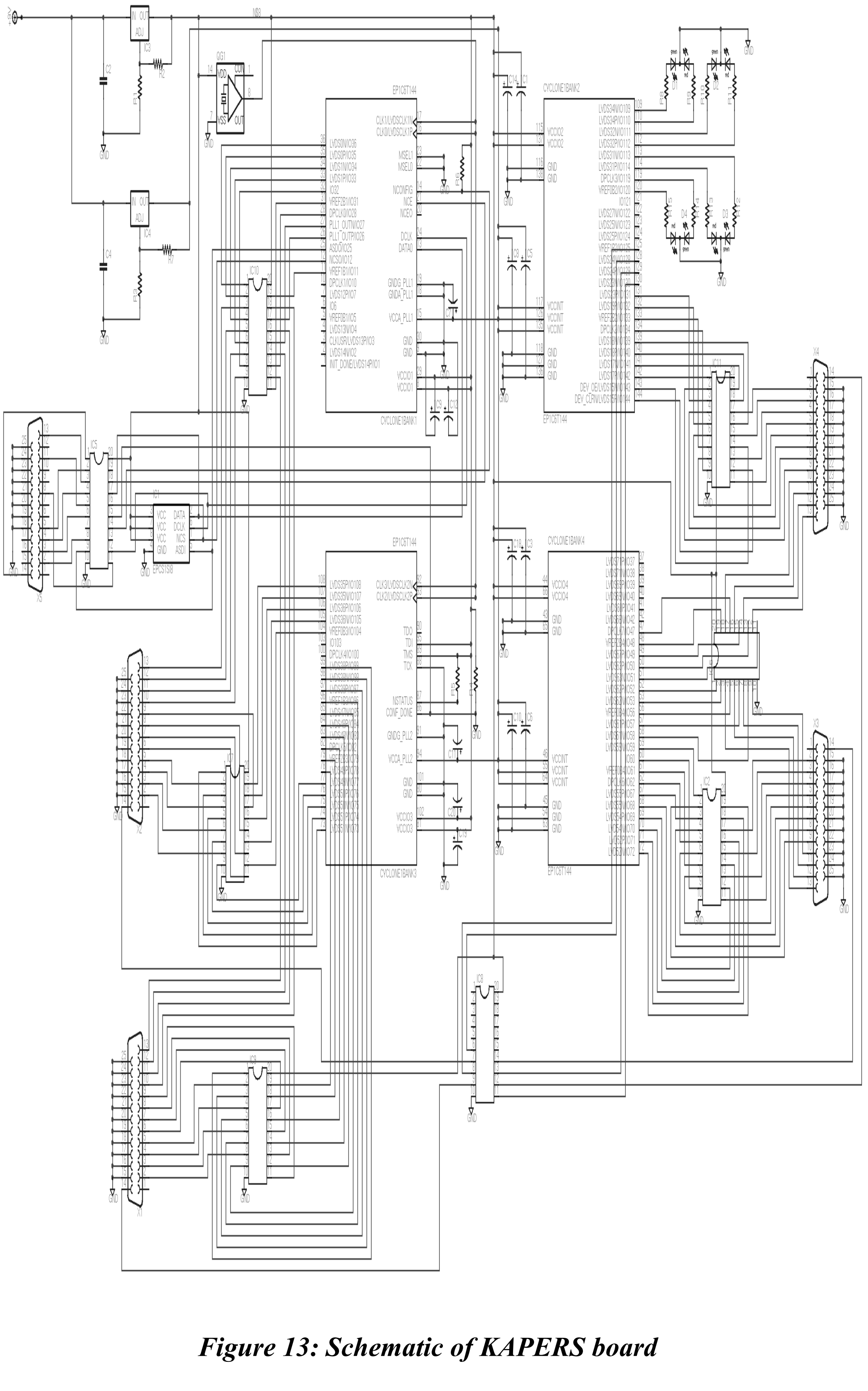


Figure 13: Schematic of KAPERS board

4.3 Layout

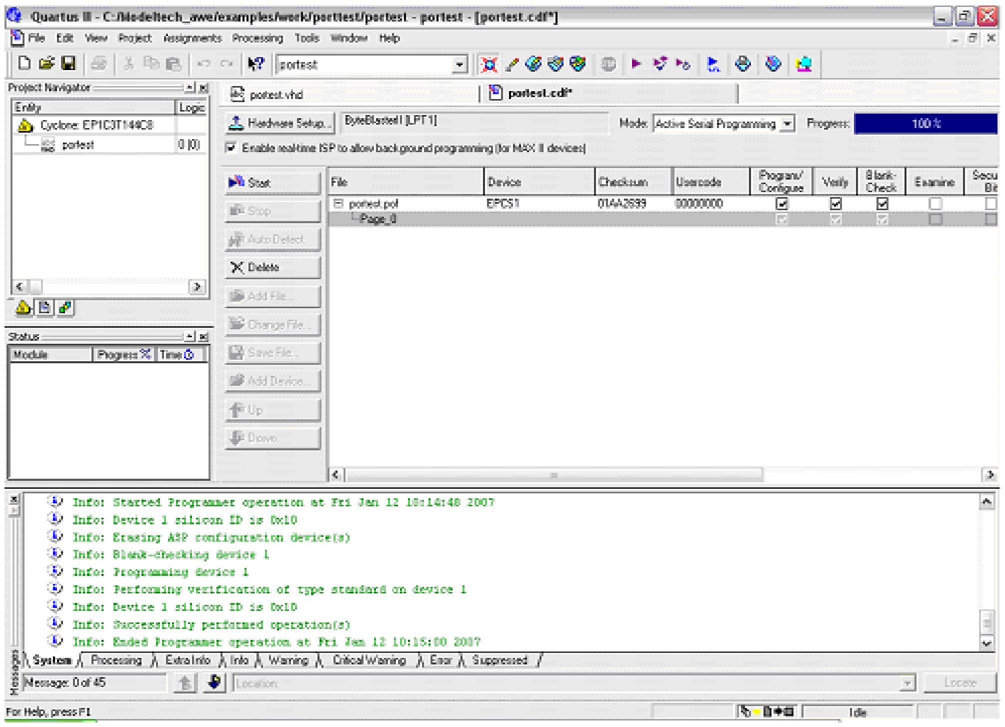
The EAGLE freeware software used for the schematic capture could not be used for layout as the board size was exceeding the allowable limits. The layout was done using the freely available PCB[21] design software for X11 window systems. A two-layer board with size 6” x 4” was sufficient to fit all components with about 500 plated through holes and 150 solder pads for surface mount parts.

4.4 Results

Soldering the components was easy -- except for the FPGA and the configuration device, which are surface mount parts with a tiny pitch between the pins. The parallel cable used as a replacement for the Byte Blaster II cable worked well and was successful in downloading the Programmer Object File(.pof) into the memory of configuration device with Quartus II programming hardware. **Figure 14** shows the picture of successful verification of the serial configuration device by blank checking and downloading a file into it. The next step is for the FPGA to get the data from the configuration device; it apparently was not successful in configuring. Measurements on the prototype revealed a problem with core voltage supplied to the FPGA.

In fact, three prototypes were assembled and tested. The first prototype had damage to one of the FPGA pins, rendering the board non-functional. The second failed spectacularly, literally causing the FPGA to burst into flames. More precisely, this second prototype revealed strange fluctuations in the power supply voltages on board. It was in attempt to remedy this by using a “beefier” switching power supply that the FPGA suffered its dramatic failure. This was tried because measurements showed the wall-mount AC adapter was regularly exceeding its maximum current ratings. The third prototype was the unit that successfully loaded the configuration device, but there were still problems with the 1.5V power supply on board.

After literally several months (November 2006 through January 2007) of attempting to find a simple fix, we have come to the conclusion that the power supply design is somehow inadequate, and a major redesign of the power handling (and hence the board) is appropriate. Perhaps we should have used the Altera-recommended parts rather than making a “clever” substitution...? In discussions with Professor Dietz, it was agreed that such a redesign was beyond the scope of this thesis and was not of critical relevance – the primary contributions of this thesis involve the architecture and algorithm, not problems with the (analog) power supply design. After simulations and other testing, we still have no reason to believe that other aspects of the design are flawed.


Quartus II - C:/Modeltech_awe/examples/work/portest/portest - portest - [portest.cdf*]

Hardware Setup... ByteBlasterII [LPT1] Mode: Active Serial Programming Progress: 100 %

[✓] Enable real-time ISP to allow background programming (for MAX II devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Secu Bit
portest.pof	EPCS1	01A42699	00000000	✓	✓	✓	☐	☐
Page_0								

```
Info: Started Programmer operation at Fri Jan 12 10:14:48 2007
Info: Device 1 silicon ID is 0x10
Info: Erasing ASP configuration device(s)
Info: Blank-checking device 1
Info: Programming device 1
Info: Performing verification of type standard on device 1
Info: Device 1 silicon ID is 0x10
Info: Successfully performed operation(s)
Info: Ended Programmer operation at Fri Jan 12 10:15:00 2007
```

Message: 0 of 45

For Help, press F1 Idle

Figure 14: Screenshot showing the successful configuration of EPCS1 device in active serial mode

Chapter5

CONCLUSION AND FUTURE WORK

This thesis has presented a new architecture and direction for future AFNs.

The hardware capture of the design resulted in less circuitry than expected. The FPGA-based implementation architecture also has the benefit of allowing further research enhancements of the protocols, operations, and internal AFN architecture without changing the physical hardware design.

Newly developed algorithms are in many ways more efficient and offer enhanced functionality (within the AFN) when compared to previous ones. The logical reduction functions and arithmetic addition are directly implemented in hardware; the multiplication reduction using LNS is a major advance that does not require any specialized hardware dedicated to that function. The scan operations are efficiently implemented using the lock feature. Communication and voting operations also benefit from the AFN memory. Perhaps best of all, the hardware's nybble-width does not prevent apparently atomic treatment of data objects of arbitrary size – a first for AFN hardware.

There is an obvious need for future work:

- The power supply circuitry must be redesigned.
- A number of configuration variables have been discussed as being appropriate for the KAPERS to pre-load or map into memory (NPROC, IPROC, memory size, performance registers, global time, etc.), but none have been implemented yet.
- Scaling of the design to larger numbers of PEs will be necessary.
- Ways to move to a higher-performance PE interface must be found. The parallel port is still viable, but a higher-bandwidth interface could dramatically increase the importance of AFNs. Currently, there is no standard PC interface with the right characteristics, but perhaps Hypertransport or another standard will evolve to provide a better answer. The architecture of the KAPERS AFN is well suited to such future interface options.

BIBLIOGRAPHY

[1] R. Hoare, H. Dietz, T. Mattox, and S. Kim, "Bitwise Aggregate networks," *Proc. Eighth IEEE Symposium On Parallel and Distributed Processing*, October, 1996, pp. I251-I254

[2] H. G. Dietz, R. Hoare and T. Mattox, "A Fine-Grain Parallel Architecture Based on Barrier Synchronization," *Proc. Int'l Conf. on Parallel Processing*, August 1996, Vol. 1, pp I247-I250

[3] H. G. Dietz, T. M. Chung, T. Mattox, and T. Muhammad, "A Parallel Processing Support Library Based On Synchronized Aggregate Communication*," Languages and Compilers for Parallel Computing*, 1995.

[4] T. Muhammad, *Hardware Barrier Synchronization For A Cluster Of Personal Computers*, Master's Thesis, School of Electrical and Computer Engineering, Purdue University, 1995.

[5] T. Mattox, *Synchronous Aggregate Communication Architecture For MIMD Parallel Processing,* Master's Thesis, School of Electrical and Computer Engineering, Purdue University, 1997.

[6] William E. Cohen, David W. Hyde, Rhonda K. Gaede, *An Optical Bus-based Distributed Dynamic Barrier Mechanism,* IEEE Transactions on Computers, Vol. 49, No. 12, December, 2000.

[7] Jon Beecroft, David Addison, David Hewson, Moray McLaren, Duncan Roweth, Fabrizio Petrini and Jarek Nieplocha, *QsNet II: Defining High-Performance Network Design*, *IEEE Micro*, Vol. 25, No. 4, pp. 34-47, July/August 2005.

[8] H. G. Dietz, T. I. Mattox, and G. Krishnamurthy, "The Aggregate Function API: It's Not Just For PAPERS Anymore," *Languages and Compilers for Parallel Computing*, August, 1997.

[9] H. G. Dietz, "AFAPI: Aggregate Function Application Program Interface," http://aggregate.org/AFAPI

[10] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, “Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors Part I: Barrier Architecture,” Tech. Report TR-EE 94-9, Purdue University, March, 1994

[11] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, “Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors Part II: Mode Emulation,” Tech. Report TR-EE 94-10, Purdue University, March, 1994

[12] R. Gupta, “The fuzzy barrier: A mechanism for the high speed synchronization of processors,” *Third Intl. conference on Architectural Support for Programming Languages and Operating Systems,* pp. 54-63, April 1989

[13] http://aggregate.org/AFN/960801/Index.html

[14] D. K. Panda, “Fast Barrier Synchronization in Wormhole K-ary N-cube Networks with Multidestination Worms,” *Journal of Future generation Computer Systems (FGCS)*, November 1995

[15] Sun, Y., Cheung, P. Y., and Lin, X., “Barrier Synchronization on Wormhole-Routed Networks,” *IEEE Trans. Parallel Distrib. Syst.* 12, 6 (Jun. 2001), 583-597

[16] W. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Trans. on Computers,* Vol. C-39, No. 6, June 1990

[17] Engel, J.; Kocak, T., "K-ary n-cube based off-chip communications architecture for high-speed packet processors," *Circuits and Systems, 2005. 48th Midwest Symposium on*, vol., no.pp. 1903- 1906 Vol. 2, 7-10 Aug. 2005

[18] Altera Corporation, Configuration Handbook Volume 1, August 2005, http://www.altera.com/literature/hb/cfg/cyc_c51013.pdf

[19] Altera Corporation, Byte Blaster II Download Cable User Guide, December 2004, www.altera.com/literature/ug/ug_bbii.pdf

[20] CadSoft Online, http://www.cadsoft.de/

[21] PCB, Printed Circuit Board Editor, http://pcb.sourceforge.net/

[22] Altera Corporation, Quartus II Software, www.altera.com/quartus2

[23] Altera Corporation, Cyclone FPGA overview, www.altera.com/cyclone

APPENDIX

A. Simulation Results

This section shows the gate level simulation results of the various sections of the design implemented in the Cyclone FPGA. These simulations are run in Quartus II 6.0 with ModelSim Altera as third party EDA tool.

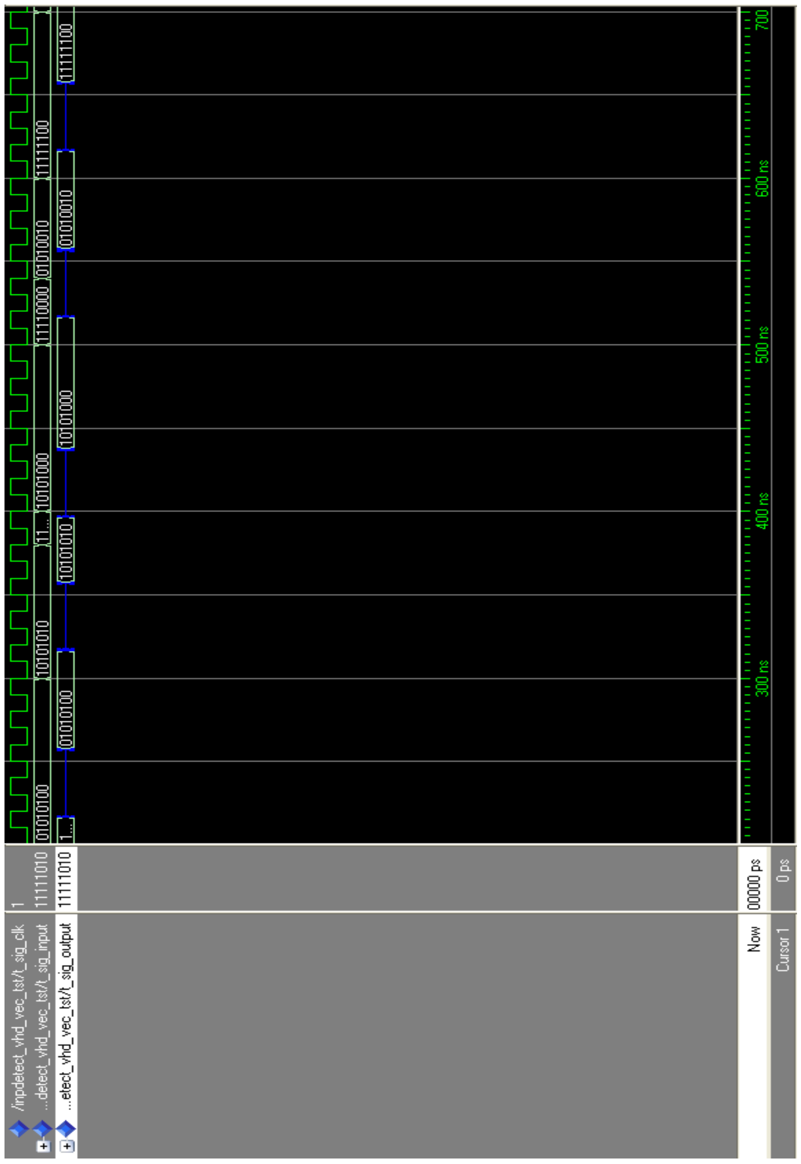


Figure 15: Simulation result for input detection section

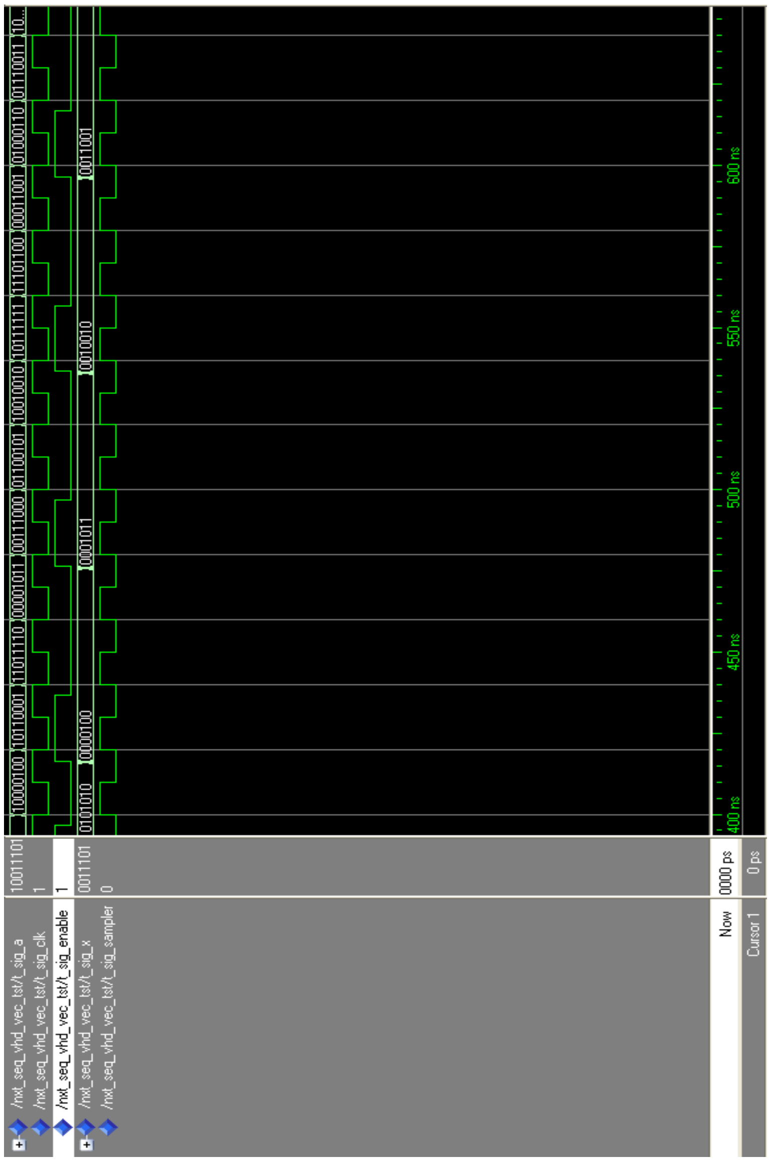
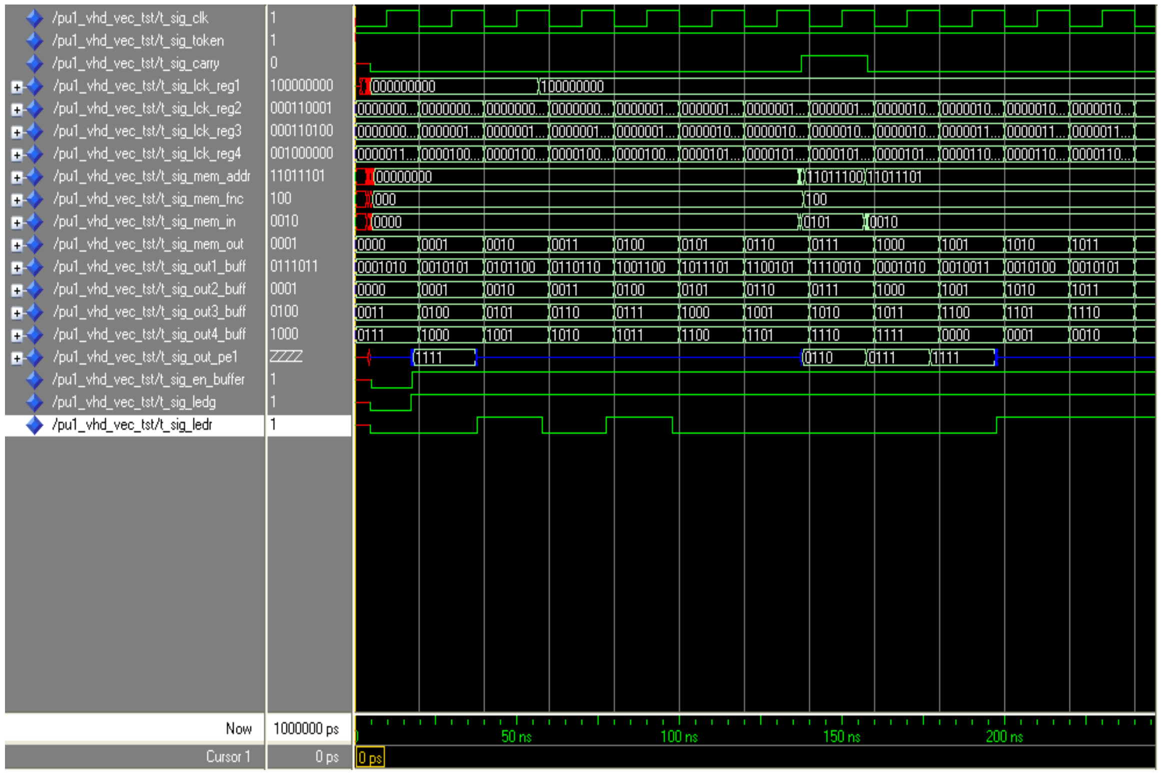


Figure 16: Simulation results for detection of next valid sequence



```
/pu1_vhd_vec_tst/t_sig_clk          1
/pu1_vhd_vec_tst/t_sig_token        1
/pu1_vhd_vec_tst/t_sig_carry        0
/pu1_vhd_vec_tst/t_sig_lck_reg1     100000000
/pu1_vhd_vec_tst/t_sig_lck_reg2     000110001
/pu1_vhd_vec_tst/t_sig_lck_reg3     000110100
/pu1_vhd_vec_tst/t_sig_lck_reg4     001000000
/pu1_vhd_vec_tst/t_sig_mem_addr     11011101
/pu1_vhd_vec_tst/t_sig_mem_fnc      100
/pu1_vhd_vec_tst/t_sig_mem_in       0010
/pu1_vhd_vec_tst/t_sig_mem_out      0001
/pu1_vhd_vec_tst/t_sig_out1_buff    0111011
/pu1_vhd_vec_tst/t_sig_out2_buff    0001
/pu1_vhd_vec_tst/t_sig_out3_buff    0100
/pu1_vhd_vec_tst/t_sig_out4_buff    1000
/pu1_vhd_vec_tst/t_sig_out_pe1      ZZZZ
/pu1_vhd_vec_tst/t_sig_en_buffer    1
/pu1_vhd_vec_tst/t_sig_ledg         1
/pu1_vhd_vec_tst/t_sig_ledr         1

Now        1000000 ps
Cursor 1   0 ps
```

Figure 17: Simulation outputs for the processing unit

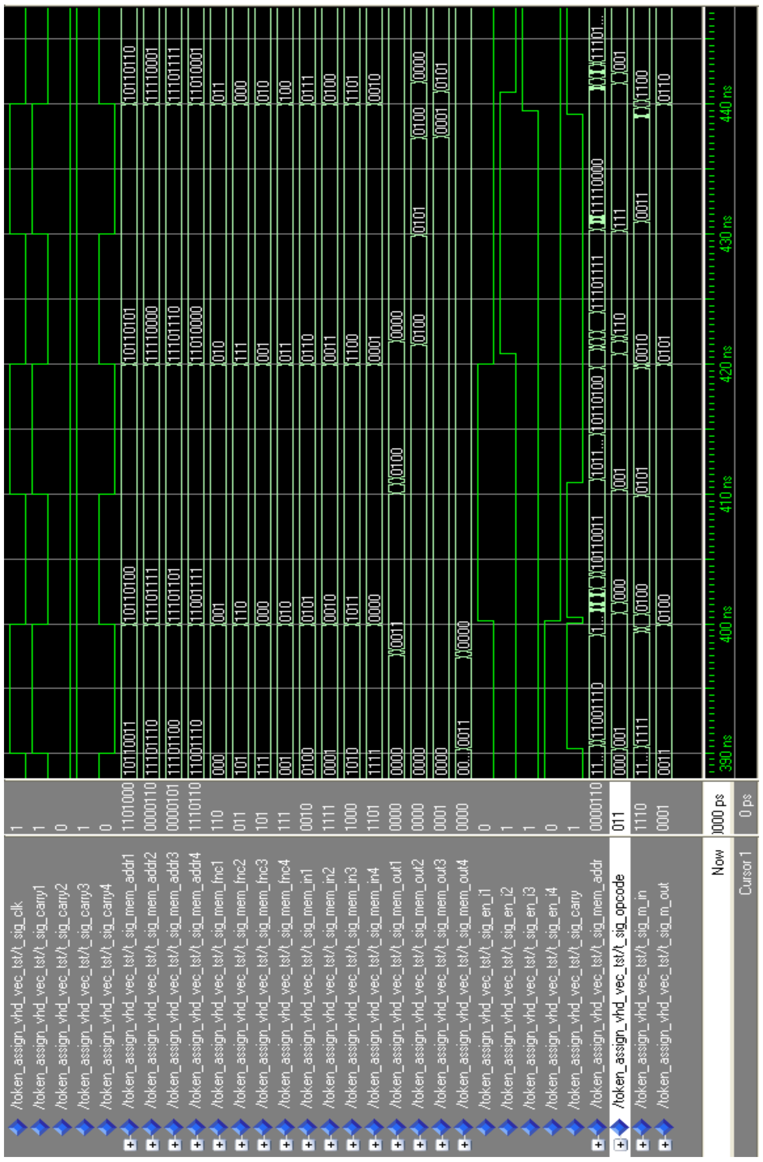
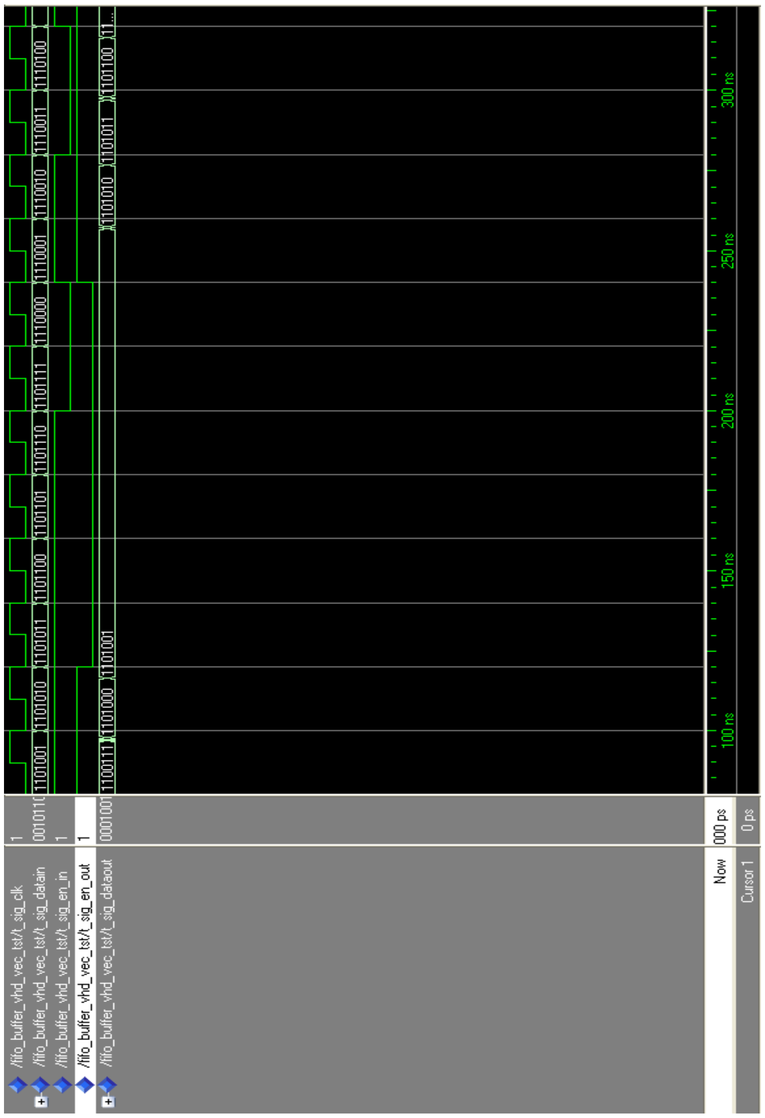

/token_assign_vhd_vec_tst/t_sig_clk
/token_assign_vhd_vec_tst/t_sig_carry1
/token_assign_vhd_vec_tst/t_sig_carry2
/token_assign_vhd_vec_tst/t_sig_carry3
/token_assign_vhd_vec_tst/t_sig_carry4
/token_assign_vhd_vec_tst/t_sig_mem_addr1
/token_assign_vhd_vec_tst/t_sig_mem_addr2
/token_assign_vhd_vec_tst/t_sig_mem_addr3
/token_assign_vhd_vec_tst/t_sig_mem_addr4
/token_assign_vhd_vec_tst/t_sig_mem_fnc1
/token_assign_vhd_vec_tst/t_sig_mem_fnc2
/token_assign_vhd_vec_tst/t_sig_mem_fnc3
/token_assign_vhd_vec_tst/t_sig_mem_fnc4
/token_assign_vhd_vec_tst/t_sig_mem_in1
/token_assign_vhd_vec_tst/t_sig_mem_in2
/token_assign_vhd_vec_tst/t_sig_mem_in3
/token_assign_vhd_vec_tst/t_sig_mem_in4
/token_assign_vhd_vec_tst/t_sig_mem_out1
/token_assign_vhd_vec_tst/t_sig_mem_out2
/token_assign_vhd_vec_tst/t_sig_mem_out3
/token_assign_vhd_vec_tst/t_sig_mem_out4
/token_assign_vhd_vec_tst/t_sig_en_i1
/token_assign_vhd_vec_tst/t_sig_en_i2
/token_assign_vhd_vec_tst/t_sig_en_i3
/token_assign_vhd_vec_tst/t_sig_en_i4
/token_assign_vhd_vec_tst/t_sig_carry
/token_assign_vhd_vec_tst/t_sig_mem_addr
/token_assign_vhd_vec_tst/t_sig_opcode
/token_assign_vhd_vec_tst/t_sig_m_in
/token_assign_vhd_vec_tst/t_sig_m_out
Now
Cursor 1
390 ns
400 ns
410 ns
420 ns
430 ns
440 ns

Figure 18: Simulation results for arbitration logic implementation with a counter


/fifo_buffer_vhd_vec_tst/t_sig_clk
/fifo_buffer_vhd_vec_tst/t_sig_datain
/fifo_buffer_vhd_vec_tst/t_sig_en_in
/fifo_buffer_vhd_vec_tst/t_sig_en_out
/fifo_buffer_vhd_vec_tst/t_sig_dataout
Now
Cursor 1
0 ps
100 ns
150 ns
200 ns
250 ns
300 ns
Figure 19: Simulation results for FIFO buffer

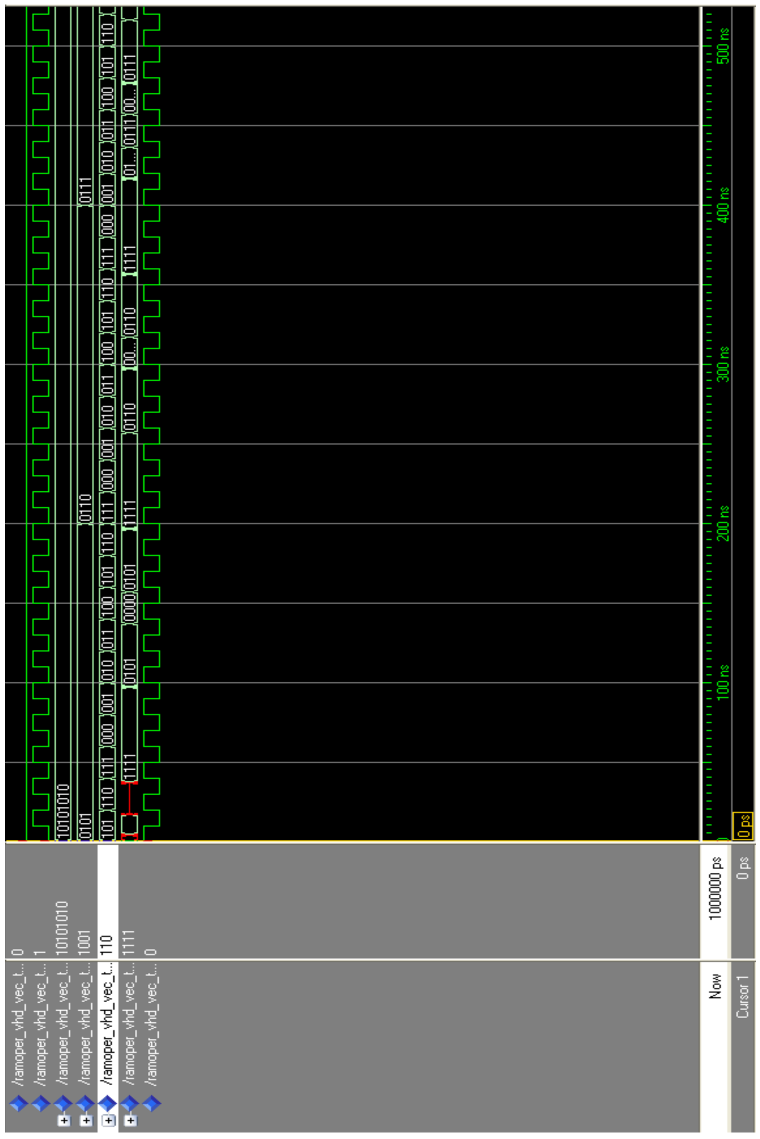


Figure 20: Simulation runs for memory with operations

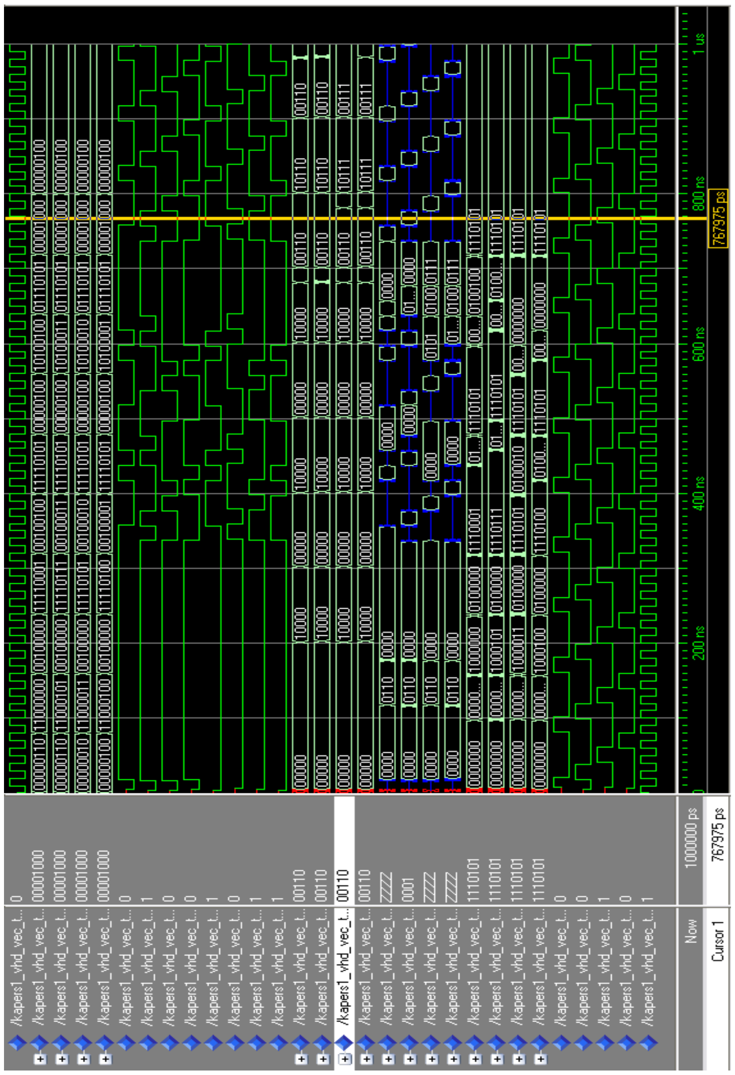


Figure 21: Final simulation run of VHDL design for KAPERS unit as a whole

B. VHDL code

kapers1.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity kapers1 is
generic(bits:integer:=4; -- nybble wide bits
                d:integer:=7;
                size:integer:=256; -- size of memory
                n:integer:=4;          -- buffer size
                k:integer:=2);         -- no of nybbles of address
port(
clk: in std_logic;
inp1:in std_logic_vector(7 downto 0);      --inputs from processing nodes
inp2:in std_logic_vector(7 downto 0);
inp3:in std_logic_vector(7 downto 0);
inp4:in std_logic_vector(7 downto 0);
o1:out std_logic_vector(4 downto 0);        --outputs to the processing nodes
o2:out std_logic_vector(4 downto 0);
o3:out std_logic_vector(4 downto 0);
o4:out std_logic_vector(4 downto 0);
ledr1:out std_logic;
ledg1:out std_logic;
ledr2:out std_logic;
ledg2:out std_logic;
ledr3:out std_logic;
ledg3:out std_logic;
ledr4:out std_logic;
ledg4:out std_logic
);
end kapers1;

architecture struct of kapers1 is
```

```
component inp_detect is
port(
input : in std_logic_vector(7 downto 0);
output : out std_logic_vector(7 downto 0);
clk : in std_logic
);
end component;

component nxt_seq is
port(
a:in std_logic_vector(7 downto 0);
clk:in std_logic;
x:out std_logic_vector(6 downto 0);
enable:out std_logic
);
end component;

component fifo_buffer_nsize is
port(
datain : in std_logic_vector(6 downto 0);
dataout : out std_logic_vector(6 downto 0);
clk : in std_logic;
en_out : in std_logic;
en_in : in std_logic
);
end component;

component pu1 is
generic(k:integer:=2);-- no of nybbles for address
port(
out1_buff : in std_logic_vector(6 downto 0);
out2_buff : in std_logic_vector(3 downto 0);
out3_buff : in std_logic_vector(3 downto 0);
out4_buff : in std_logic_vector(3 downto 0);
clk : in std_logic;
carry : out std_logic;
```

```
out_pe1: out std_logic_vector(3 downto 0);
mem_out: in std_logic_vector(3 downto 0);
mem_in : out std_logic_vector(3 downto 0);
mem_addr : out std_logic_vector(4*k-1 downto 0);
mem_fnc :out std_logic_vector(2 downto 0);
token : in std_logic;
lck_reg1: inout std_logic_vector(8 downto 0);
lck_reg2: in std_logic_vector(8 downto 0);
lck_reg3: in std_logic_vector(8 downto 0);
lck_reg4: in std_logic_vector(8 downto 0);
en_buffer: out std_logic;
ledr:out std_logic;
ledg:out std_logic
);
end component;

component pu2 is
generic(k:integer:=2);-- no of nybbles for address
port(
out1_buff : in std_logic_vector(3 downto 0);
out2_buff : in std_logic_vector(6 downto 0);
out3_buff : in std_logic_vector(3 downto 0);
out4_buff : in std_logic_vector(3 downto 0);
clk : in std_logic;
carry : out std_logic;
out_pe2: out std_logic_vector(3 downto 0);
mem_out: in std_logic_vector(3 downto 0);
mem_in : out std_logic_vector(3 downto 0);
mem_addr : out std_logic_vector(4*k-1 downto 0);
mem_fnc :out std_logic_vector(2 downto 0);
token: in std_logic;
lck_reg1: in std_logic_vector(8 downto 0);
lck_reg2: inout std_logic_vector(8 downto 0);
lck_reg3: in std_logic_vector(8 downto 0);
lck_reg4: in std_logic_vector(8 downto 0);
en_buffer: out std_logic;
```

```
ledr:out std_logic;
ledg:out std_logic
);
end component;

component pu3 is
generic(k:integer:=2);-- no of nybbles for address
port(
out1_buff : in std_logic_vector(3 downto 0);
out2_buff : in std_logic_vector(3 downto 0);
out3_buff : in std_logic_vector(6 downto 0);
out4_buff : in std_logic_vector(3 downto 0);
clk : in std_logic;
carry : out std_logic;
out_pe3: out std_logic_vector(3 downto 0);
mem_out: in std_logic_vector(3 downto 0);
mem_in : out std_logic_vector(3 downto 0);
mem_addr : out std_logic_vector(4*k-1 downto 0);
mem_fnc :out std_logic_vector(2 downto 0);
token: in std_logic;
lck_reg1: in std_logic_vector(8 downto 0);
lck_reg2: in std_logic_vector(8 downto 0);
lck_reg3: inout std_logic_vector(8 downto 0);
lck_reg4: in std_logic_vector(8 downto 0);
en_buffer: out std_logic;
ledr:out std_logic;
ledg:out std_logic
);
end component;

component pu4 is
generic(k:integer:=2);-- no of nybbles for address
port(
out1_buff : in std_logic_vector(3 downto 0);
out2_buff : in std_logic_vector(3 downto 0);
out3_buff : in std_logic_vector(3 downto 0);
```

```
out4_buff : in std_logic_vector(6 downto 0);
clk : in std_logic;
carry : out std_logic;
out_pe4: out std_logic_vector(3 downto 0);
mem_out: in std_logic_vector(3 downto 0);
mem_in : out std_logic_vector(3 downto 0);
mem_addr : out std_logic_vector(4*k-1 downto 0);
mem_fnc :out std_logic_vector(2 downto 0);
token: in std_logic;
lck_reg1: in std_logic_vector(8 downto 0);
lck_reg2: in std_logic_vector(8 downto 0);
lck_reg3: in std_logic_vector(8 downto 0);
lck_reg4: inout std_logic_vector(8 downto 0);
en_buffer: out std_logic;
ledr:out std_logic;
ledg:out std_logic
);
end component;

component token_assign is
port(
clk : in std_logic;
mem_out1: out std_logic_vector(3 downto 0);
mem_out2: out std_logic_vector(3 downto 0);
mem_out3: out std_logic_vector(3 downto 0);
mem_out4: out std_logic_vector(3 downto 0);
mem_in1 : in std_logic_vector(3 downto 0);
mem_in2 : in std_logic_vector(3 downto 0);
mem_in3 : in std_logic_vector(3 downto 0);
mem_in4 : in std_logic_vector(3 downto 0);
mem_fnc1 :in std_logic_vector(2 downto 0);
mem_fnc2 :in std_logic_vector(2 downto 0);
mem_fnc3 :in std_logic_vector(2 downto 0);
mem_fnc4 :in std_logic_vector(2 downto 0);
mem_addr1 : in std_logic_vector(7 downto 0);
mem_addr2 : in std_logic_vector(7 downto 0);
```

```
mem_addr3 : in std_logic_vector(7 downto 0);
mem_addr4 : in std_logic_vector(7 downto 0);
carry1 : in std_logic;
carry2 : in std_logic;
carry3 : in std_logic;
carry4 : in std_logic;
en_i1 : out std_logic;
en_i2 : out std_logic;
en_i3 : out std_logic;
en_i4 : out std_logic;
m_in : out std_logic_vector(3 downto 0);
m_out : in std_logic_vector(3 downto 0);
mem_addr :out std_logic_vector(7 downto 0);
carry : out std_logic;
opcode : out std_logic_vector(2 downto 0)
);
end component;


component ram_opcode is
generic(bits:integer:=4;--size of data bus
                size:integer:=256);--size of memory
port(clk: in std_logic;
        opcode : in std_logic_vector(2 downto 0);
        m_addr : in std_logic_vector(7 downto 0);
        m_in: in std_logic_vector(bits-1 downto 0);
        m_out:out std_logic_vector(bits-1 downto 0);
        carry : in std_logic
        );
end component;

component nbit_reg is
port(reg_in :in std_logic_vector(8 downto 0);
        wr_en,clk: in std_logic;
        reg_out: out std_logic_vector(8 downto 0));
end component;
```

```
component out_buffer is
port(
datain : in std_logic_vector(4 downto 0);
dataout : out std_logic_vector(4 downto 0);
clk : in std_logic;
en_in : in std_logic);
end component;



signal en_i1,en_i2,en_i3,en_i4 : std_logic:='1';
signal out1,out2,out3,out4 : std_logic_vector(3 downto 0);
signal en_in1,en_in2,en_in3,en_in4:std_logic:='1';
signal en_out1,en_out2,en_out3,en_out4:std_logic:='1';
signal inp1_buff,inp2_buff,inp3_buff,inp4_buff : std_logic_vector(6 downto 0);
signal out1_buff,out2_buff,out3_buff,out4_buff : std_logic_vector(6 downto 0);
signal mem_fnc1,mem_fnc2,mem_fnc3,mem_fnc4:std_logic_vector(2 downto 0);
signal
m_in,mem_in1,mem_in2,mem_in3,mem_in4,m_out,mem_out1,mem_out2,mem_ou
t3,mem_out4:std_logic_vector(3 downto 0);
signal
mem_addr,mem_addr1,mem_addr2,mem_addr3,mem_addr4:std_logic_vector(7
downto 0);
signal opcode: std_logic_vector(2 downto 0);
signal lck_reg1,lck_reg2,lck_reg3,lck_reg4:std_logic_vector(8 downto 0);
signal lck_reg1_in,lck_reg2_in,lck_reg3_in,lck_reg4_in:std_logic_vector(8 downto
0);
signal carry,carry1,carry2,carry3,carry4:std_logic;
signal wr_en:std_logic:='0';
signal op1,op2,op3,op4:std_logic;
signal inp_det1,inp_det2,inp_det3,inp_det4:std_logic_vector(7 downto 0);

begin

--Input detection circuit
i1: inp_detect port map(inp1,inp_det1,clk);
i2: inp_detect port map(inp2,inp_det2,clk);
i3: inp_detect port map(inp3,inp_det3,clk);
```

```
i4: inp_detect port map(inp4,inp_det4,clk);

--Valid input pattern
a1: nxt_seq port map (inp_det1,clk,inp1_buff,en_in1);
a2: nxt_seq port map (inp_det2,clk,inp2_buff,en_in2);
a3: nxt_seq port map (inp_det3,clk,inp3_buff,en_in3);
a4: nxt_seq port map (inp_det4,clk,inp4_buff,en_in4);

--FIFO buffer
b1: fifo_buffer_nsize port map (inp1_buff,out1_buff,clk,en_out1,en_in1);
b2: fifo_buffer_nsize port map (inp2_buff,out2_buff,clk,en_out2,en_in2);
b3: fifo_buffer_nsize port map (inp3_buff,out3_buff,clk,en_out3,en_in3);
b4: fifo_buffer_nsize port map (inp4_buff,out4_buff,clk,en_out4,en_in4);

--Registers for lock bits
wr_en <= '1';
r1:nbit_reg port map (lck_reg1_in,wr_en,clk,lck_reg1);
r2:nbit_reg port map (lck_reg2_in,wr_en,clk,lck_reg2);
r3:nbit_reg port map (lck_reg3_in,wr_en,clk,lck_reg3);
r4:nbit_reg port map (lck_reg4_in,wr_en,clk,lck_reg4);

--Processing Unit
c1: pu1 port map(out1_buff,out2_buff(3 downto 0),out3_buff(3 downto
0),out4_buff(3 downto
0),clk,carry1,out1,mem_out1,mem_in1,mem_addr1,mem_fnc1,en_i1,lck_reg1_in,lc
k_reg2,lck_reg3,lck_reg4,en_out1,ledr1,ledg1);
c2: pu2 port map(out1_buff(3 downto 0),out2_buff,out3_buff(3 downto
0),out4_buff(3 downto
0),clk,carry2,out2,mem_out2,mem_in2,mem_addr2,mem_fnc2,en_i2,lck_reg1,lck_r
eg2_in,lck_reg3,lck_reg4,en_out2,ledr2,ledg2);
c3: pu3 port map(out1_buff(3 downto 0),out2_buff(3 downto
0),out3_buff,out4_buff(3 downto
0),clk,carry3,out3,mem_out3,mem_in3,mem_addr3,mem_fnc3,en_i3,lck_reg1,lck_r
eg2,lck_reg3_in,lck_reg4,en_out3,ledr3,ledg3);
c4: pu4 port map(out1_buff(3 downto 0),out2_buff(3 downto 0),out3_buff(3 downto
0),out4_buff,clk,carry4,out4,mem_out4,mem_in4,mem_addr4,mem_fnc4,en_i4,lck_
reg1,lck_reg2,lck_reg3,lck_reg4_in,en_out4,ledr4,ledg4);

--Memory
mem: ram_opcode port map (clk,opcode,mem_addr,m_in,m_out,carry);
```

```
--Token assignment for shared resources
tok: token_assign port map(clk,
mem_out1,mem_out2,mem_out3,mem_out4,
mem_in1,mem_in2,mem_in3,mem_in4,
mem_fnc1,mem_fnc2,mem_fnc3,mem_fnc4,
mem_addr1,mem_addr2,mem_addr3,mem_addr4,
carry1,carry2,carry3,carry4,
en_i1,en_i2,en_i3,en_i4,
m_in,m_out,mem_addr,carry,opcode);
process(clk)
begin
if clk'event and clk = '1' then
        count <= count + 1;
        if count = "010" then
                en <= '1';
                en_t <= '0';
        elsif count = "100" then
                en <= '0';
                en_t <= '1';
                count <= "000";
        else
                en <= '0';
                en_t <= '0';
        end if;
end if;
end process;

b5: fifo_buffer_out port map(out1,dataout1,clk,en,ob1);
b6: fifo_buffer_out port map(out2,dataout2,clk,en,ob2);
b7: fifo_buffer_out port map(out3,dataout3,clk,en,ob3);
b8: fifo_buffer_out port map(out4,dataout4,clk,en,ob4);

process(en_t)
begin
if en_t = '1' then
        if op1 = '1' then
```

```
                op1 <= '0';
        else
                op1 <= '1';
        end if;
                o1(4) <= op1;
        if op2 = '1' then
                op2 <= '0';
        else
                op2 <= '1';
        end if;
                o2(4) <= op2;
        if op3 = '1' then
                op3 <= '0';
        else
                op3 <= '1';
        end if;
                o3(4) <= op3;
        if op4 = '1' then
                op4 <= '0';
        else
                op4 <= '1';
        end if;
                o4(4) <= op4;
        end if;
end process;

o1(3 downto 0) <= dataout1;
o2(3 downto 0) <= dataout2;
o3(3 downto 0) <= dataout3;
o4(3 downto 0) <= dataout4;

end struct;
```

C. Algorithms

inline.h

```
/*	inline.h
*/

#define BarOr			0x00
#define SetFunc			0x20
#define AddrFirst		0x40
#define AddrNext		0x50
#define MemNext			0x60
#define MemLast			0x70

#define toggle			0x80

#define Xchg		0x00
#define Or		0x01
#define Xor		0x02
#define Add		0x03
#define Min		0x04

#define lock		0x08
#define unlock			0x00

#define dataadjust(d)		((d>>4) & 0x0f)
#define ready		0x10

#define DATAOUTPORT		0x000
#define STATUSINPORT		0x001

unsigned char p_outlast=0;
unsigned int p_inlast=0;
unsigned int p_addrlast=0;

#include "afapi.h"
```

```
/*        p__inb() and p__outb()...
          Defined here so as to eliminate any delays of port settle time,
          which is caused in codes from predifined libraries.
*/


#ifdef P_DEBOUNCE

static inline unsigned int
p___inb (unsigned short port)
{
          unsigned char _v;
__asm__ __volatile__ ("inb %w1,%b0"
          :"=a"(_v)
          :"d" (port), "0" (0));
          return _v;
}

static inline unsigned int
p__inb (unsigned short port)
{
          register unsigned int t = p___inb(port);
          while( t != p_inlast) {
                    p_inlast = t;
                    t = p___inb(port);
          }
          return(t);
}

#else

static inline unsigned int
p__inb (unsigned short port)
{
          unsigned char _v;
__asm__ __volatile__ ("inb %w1,%b0"
        :"=a"(_v)
        :"d" (port), "0" (0));
        return _v;
}

#endif

static inline void
p__outb (unsigned char value, unsigned short port)
{
__asm__ __volatile__ ("outb  %b0, %w1"
        :
        :"a" (value), "d" (port));
}



static inline int
p__baror(register int d)
{
        register int to = (((p_outlast ^ toggle) & toggle) |
                               BarOr|
                               (d & 0x0f));
        register int ti;
        p__outb(to,DATAOUTPORT);
        p_outlast = to;
        do {
        ti = p__inb(STATUSINPORT);
        } while(((ti ^ p_inlast) & ready) == 0);
        p_inlast = ti;
        return(dataadjust(ti));
}

static inline void
p__address(register int d)
{
        /* Address that needs to be sent is first compared with the
        previous value and only the nybbles that differ are sent. */

        register int diff = (d ^ p_addrlast);
        if (diff != 0) {
        register int t = (((p_outlast ^ toggle) & toggle) |
                              AddrFirst|
                              (d & 0x0f));
        p_addrlast = d;
        p__outb(t,DATAOUTPORT);
        p_outlast = t;
        diff >>= 4;
        while (diff != 0) {
                d >>= 4;
                t = (((p_outlast ^ toggle) & toggle) |
                              AddrNext|
                              (d & 0x0f));
                p__outb(t,DATAOUTPORT);
                p_outlast = t;
                diff >>= 4;
                }
        }
}


static inline void
p__setfunc(register int d, register int l)
{
        register int t = (((p_outlast ^ toggle) & toggle) |
                              SetFunc|
                               l |
                              (d & 0x0f));
        p__outb(t,DATAOUTPORT);
        p_outlast = t;
}
```

```
static inline int
p__memfunc(register int d)
{
        register int n = sizeof(d)*2;
        register int ti;
        register int result = 0;
        while (n > 1) {
                register int t = (((p_outlast ^ toggle) & toggle) |
                                MemNext |
                                (d & 0x0f));
                p__outb(t,DATAOUTPORT);
                p_outlast = t;
                d >>= 4;
                n -= 1;
                do {
                ti = p__inb(STATUSINPORT);
                } while(((ti ^ p_inlast) & ready) == 0);
                p_inlast = ti;
                result = (result << 4) | (dataadjust(ti));
        }

        int t = (((p_outlast ^ toggle) & toggle) |
                                MemLast |
                                (d & 0x0f));
        p__outb(t,DATAOUTPORT);
        p_outlast = t;

        do {
        ti = p__inb(STATUSINPORT);
        } while(((ti ^ p_inlast) & ready) == 0);

        p_inlast = ti;
        return((result << 4) | (dataadjust(ti)));
}
```

```
static inline char
p__memnext(register char d)
{
        register int ti;
        register int result = 0;
        register int t = (((p_outlast ^ toggle) & toggle) |
                              MemNext |
                              (d & 0x0f));
        p__outb(t,DATAOUTPORT);
        p_outlast = t;
        do {
        ti = p__inb(STATUSINPORT);
        } while(((ti ^ p_inlast) & ready) == 0);
        p_inlast = ti;
        result = dataadjust(ti);
}


static inline char
p__memlast(register char d)
{
        register int ti;
        register int result = 0;
        register int t = (((p_outlast ^ toggle) & toggle) |
                              MemLast |
                              (d & 0x0f));
        p__outb(t,DATAOUTPORT);
        p_outlast = t;
        do {
        ti = p__inb(STATUSINPORT);
        } while(((ti ^ p_inlast) & ready) == 0);
        p_inlast = ti;
        result = dataadjust(ti);

}
}
```

D. Layout of KAPERS board

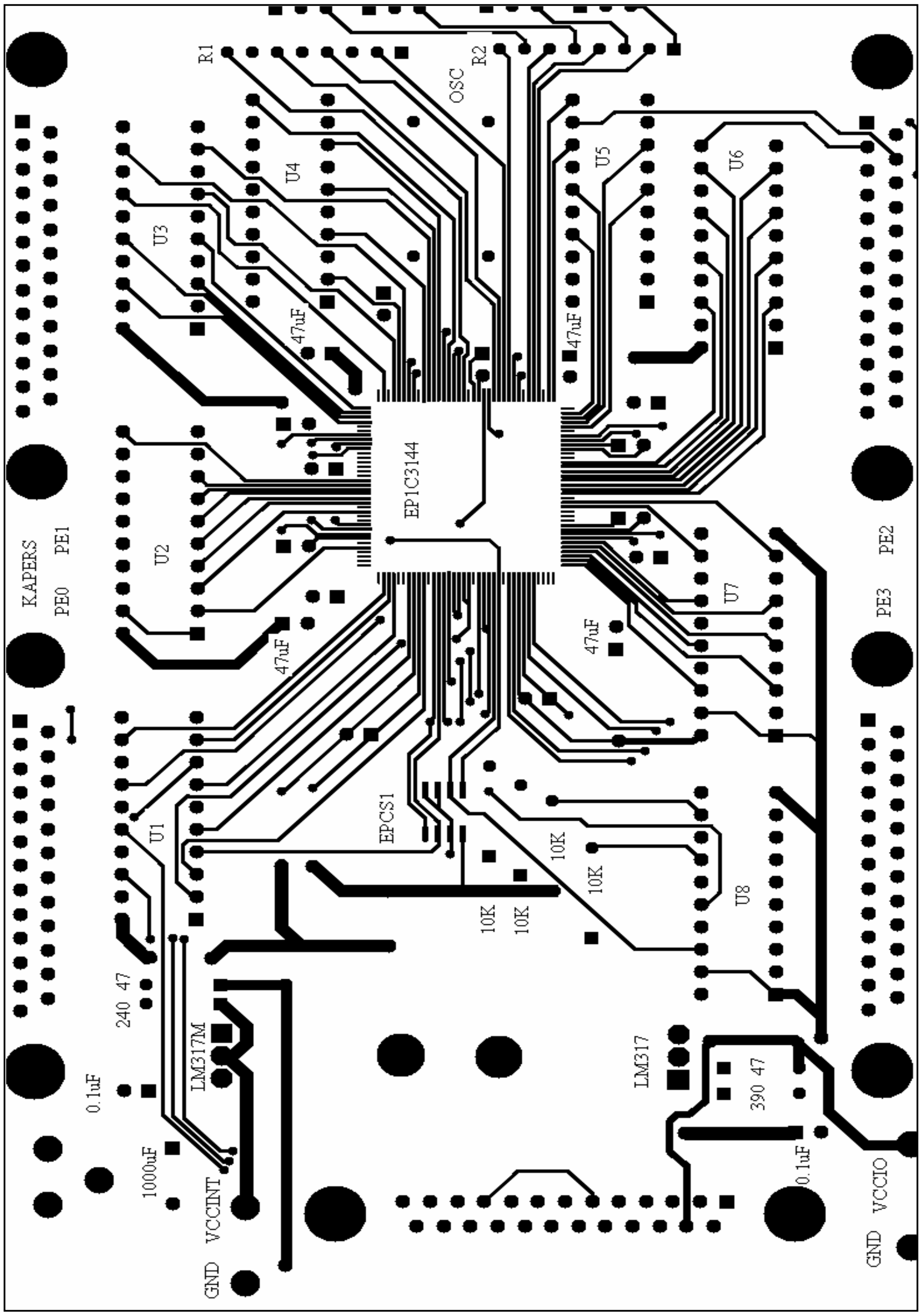


Figure 22: Top view of the layout for the KAPERS board

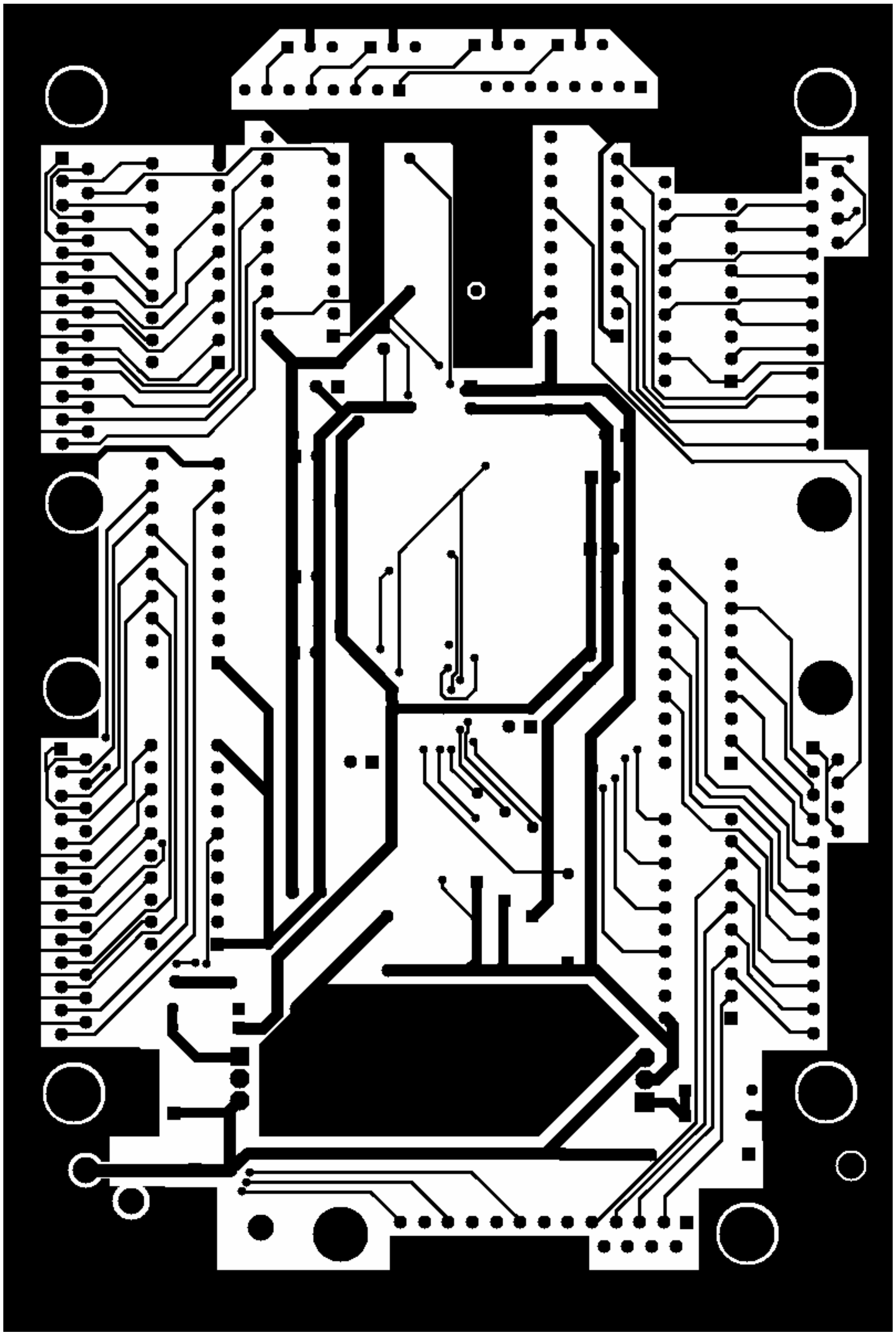


Figure 23: Bottom view of layout for KAPERS board

VITA

The author was born in Khammam, Andhra Pradesh, India on February 1, 1983. In 2004, she completed her undergraduate degree in Electronics and Communication Engineering at Chaitanya Bharati Institute of Technology affiliated to Osmania University. In August 2004, she moved to the University Of Kentucky to pursue her Masters in Electrical Engineering, and was awarded a Kentucky Graduate Scholarship. Her work on the KAPERS AFN has been supported in part by awarding her a Research Assistant position in the Department of Electrical and Computer Engineering, funded by Professor Dietz's endowment as the James F. Hardymon Chair in Networking.