A Case for Asynchronous Microengines for Network Processing

Niti Madan and Erik Brunvand
School of Computing
University of Utah
{niti,elb}@cs.utah.edu

Abstract

We present a network processor architecture which is based on asynchronous microcoded controller hardware (a.k.a asynchronous microengine). The focus of this work is not on the processor architecture, but rather on the asynchronous microcoded style used to build such an architecture. This circuit style tries to fill the performance gap between a specialized ASIC and a more general network processor implementation. It does this by providing a microcoded framework which is close in performance to ASICs and is also programmable at the finer granularity of microcode. Our approach exploits the inherent advantages of asynchronous design techniques to exhibit modularity, lower power consumption and low EMI. We have evaluated our circuit style by demonstrating fast-path IP routing as the packet processing application. For shorter design cycle time, we have implemented our design using Xilinx SpartanII FPGA board. However, we are extrapolating our results for a best-guess ASIC implementation.

1. Introduction

Programmable controllers have gained a wide-spread popularity in varied processing fields as they give the designers the advantages of allowing correction of errors in later stages of design cycles, flexibility, easy upgrading of product families, meeting time to market etc. without compromising much on performance. There have been examples of realization of synchronous programmable controllers like the ones in the FLASH processor [13], S3MP processor [1] and many commercial ASICs. Although traditional programmable controllers are synchronous, there are no reasons that such controllers could not also be designed in an asynchronous style. Asynchronous implementation could add extra features that designers could exploit to improve their systems. For example, asynchronous programmable control in the form of a micro-programmed asynchronous controller (also known as asynchronous microengine [8, 9, 7]) provide modular and easily extensible datapath structures along with high performance by exploiting concurrency between operations and employing efficient control structures. These microengines are programmable at the finer granularity of microcode and closer in performance to ASICs. Other than the modularity and flexibility advantages, we are also able to exploit other aspects of asynchronous design style like low power consumption and low EMI. Asynchronous microengines have been around for quite sometime but haven't been actively pursued in realistic domains. The focus of this paper is to evaluate an asynchronous microengine based architecture in the domain of network processing.

Traditionally, most networking functions above the physical layer have been implemented by software running on general-purpose processors or there have been specialized ASICs for these tasks. ASICs provide an expensive solution and give a good performance but lack flexibility and programmability while general-purpose processors provide programmability at the cost of performance. The bandwidth explosion in the past couple of years has resulted in more bandwidth-hungry and computationally intensive applications like VoIP, streaming audio and video, P2P applications etc. For networks to effectively handle these new applications, they will need to support new protocols along with high throughput and thus, arises the need for flexible and modular network processing equipment. Initially, the layer 2 and layer 3 processing was hardwired but, after rapid changes in lower layer protocols and higher layer's applications, a more scalable solution in the form of network processors [3, 15] has emerged. These network processing units have been optimized for networking applications and combine the advantages of ASIC and general purpose processors. There has been a wide range of programmable architectures proposed for network processing. We believe that our asynchronous-microengine based circuit style can also be one of the solutions and it fits into the domain of network processing naturally by exploiting the asynchrony in network packets flow along with a highly flexible, modular and extensible architecture.

In this work, we present a case for asynchronous microengines in the domain of network processing. We have evaluated our circuit style by demonstrating fast-path IP routing as the packet processing application. We have designed two different types of microengines, namely ingress and IP-header processing microengines. The ingress microengine does packet classification while the IP-header processing microengine performs various functions on the IP header like ttl decrement, checksum check and computation etc. Each microengine core is specialized enough for different packet processing kernels yet, generic enough to handle newer protocols and applications. The flexibility aspect of this design has been demonstrated by adding firewalling functionality to the router by modifying the microcode.

These microengine cores can be used to replace RISC cores or can be used as co-processors. The asynchronous design style makes it possible to have a large number of these cores on a single chip without having to worry about clock interface issues. An asynchronous core can also be easily integrated with a synchronous system by FIFO interfaces.

Due to shorter design cycle time, we have implemented this design on a Xilinx xs2S150 SpartanII FPGA board. We have compared our architecture's per-packet computation performance to Click software router [12].

In the rest of the paper we discuss the related work, followed by the background section which explains asynchronous design concepts, the microengine architecture and click software router. Later, we explain our ingress and IP header processing microengine architectures in detail. After describing our design methodology and performance evaluation details, we summarize our conclusions.

2. Related Work

2.1. Programmable Asynchronous Controllers

Programmable asynchronous controllers were looked at in 1980s [16] in the context of data-driven machines. They used a vertical microcode for their micro-sequencer which was used to drive multiple slave controllers along with structured tiling which introduced considerable control-overhead. It was also not an application-specific controller. In 1997, Jacobson and Gopalakrishnan at Utah looked into the design of efficient application-specific asynchronous microengines [8, 9, 7]. Their architecture uses a horizontal micro-code which allows per-microinstruction programmability of its datapath topology by arranging its datapath units into series-parallel clusters for each micro-instruction.

2.2. Network Processors

Many industries have ventured into Network Processor design and hence, there is a wide variety of architectures available in the market. However, all designs have one key point in common: they use multiple programmable processing cores or engines (PPE) in a single chip. For example, Intel IXP1200 consists of 6 micro-engines on a single die but the amount of functionality in these cores varies from vendor to vendor. Some use RISC cores with added bit-manipulation instruction-set also known as ASIP (Application-specific Instruction-set Processor), while others use a VLIW-based architecture. In these architectures, multiple PPEs can process in parallel, pipeline or combination of both styles depending on the application. However, a RISC-based architecture is more flexible and easy to configure compared to VLIW-based architecture. Many RISC-based architectures use multi-threading in each core to maximize throughput and do useful work while there is a wait for the operation to complete. This is particularly useful to hide memory access latency. Network processors which are RISC-based provide dedicated and specialized hardware or integrated co-processors to perform common network processing tasks like encryption, lookup, classification, CRC computation etc.

3. Background

In this section, we briefly introduce the asynchronous design approach, discuss the motivation behind using this approach, the asynchronous microengine architecture and the Click software router [12].

3.1. Asynchronous Design

Asynchronous, or self-timed, systems are those that are not subject to a global synchronizing clock. Since asynchronous circuits have no global clock, they must use some other means to enforce sequencing of circuit activities. These sequencing techniques can range from using a locally generated, clock-like signal in each submodule to employing handshake signals both to initiate (request signal) and detect completion of submodule activities (acknowledge signal), and many variations in between. Circuits that use handshakes to sequence operations are known as *self-timed* circuits and allow control to be distributed throughout the circuit instead of centralized in a controller. For asynchronous control, the two dominant handshaking protocols are:

2-phase (transition) signaling in which each transition on REQ or ACK signals represents an event.

4-phase (level) signaling in which only positive-going transition on REQ or ACK signals initiates an event and each signal must be "returned to zero" before the handshake cycle is completed.

A self-timed system thus, consists of self-timed modules which are communicating with each other in parallel or in sequence using these handshake protocols. In our implementation, we are using a 4-phase handshaking protocol.

In a network processing application domain, asynchronous systems have the following potential advantages compared to their synchronous counterparts:

- Modularity and Composability: Asynchronous systems allow modular datapaths i.e. easy upgrading of datapaths is possible without having to worry about clock scheduling issues. Smaller asynchronous subsystems can be easily combined into larger systems which is a definite advantage compared to synchronous systems where composability is hard due to difficulty in interfacing between subsystems operating at different clock frequencies [5, 2].

- Average case completion time: Asynchronous datapaths take into account only the average-case completion compared to worst-case propagation in the case of synchronous circuits.

- Low power dissipation:Since, asynchronous circuits get into idle mode if they are free, this results in lower power dissipation compared to synchronous circuits which consume power even during idle periods due to continuously applied clock signal.

- Low EMI: Asynchronous circuits have lower EMI and this can be a considerable advantage especially in RF networking domain.

3.2. Asynchronous Microengine

An asynchronous microengine is a microprogrammed self-timed controller in which the datapaths can be programmed by the micro-instruction to execute in sequence or in parallel with respect to other datapaths. This feature allows the parallel clusters to run concurrently while allowing the serial units within a cluster to chain. This chaining of operations is possible due to explicit acknowledge generation by each asynchronous datapath in the sequence. Chaining reduces the control overhead as it reduces the number of micro-instructions by combining several micro-instructions into a single VLIW type instruction. This is very hard to implement for synchronous designs as the propagation delay of serial partitions of combinational modules must add up to be an integral multiple of clock period.

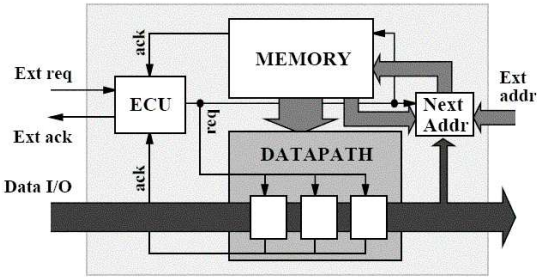

Figure 1. Microengine's High Level Structure

Architecture Overview: A conventional (*synchronously clocked*) microprogrammed control structure consists of a microprogram store, next address logic, and a datapath. Microinstructions form commands applied on the datapath and control flow is handled by the next address logic that, with the help of status signals fed back from the datapath, generates the address of the next microinstruction to be executed. In a synchronous realization the execution rate is set by the global clock which must take the worst case delay of all units into account. When the next clock edge arrives it is thus assumed that the datapath has finished computing and the next address has been resolved, and the next microinstruction can be propagated to the datapath. The asynchronous microengines have an organization similar to those of conventional synchronous microprogrammed controllers as shown in Figure 1.

In conventional synchronous microprogrammed controllers, the computation is started by an arriving clock edge and the datapath is assumed to have completed by the following clock edge. In the asynchronous case we have no clock to govern the start and end of an instruction execution. Instead a request is generated to trigger the memory to latch the new microinstruction and the datapath units to start executing. The memory and each datapath unit then signals their completion by generating an acknowledge.

Microengine Execution: A microengine starts its execution on receiving an external request from the environment. The execution control unit in response to this request generates a global request which latches the first or the specified micro-instruction from the microprogram and also causes the datapaths which are set up for execution to start executing. The microcode consists of fields which specify which datapath has been setup for execution and in what mode i.e. sequential or parallel with respect to other datapaths, and multiplexer select signals which specify the input data and output data for that particular microinstruction. On completion of computation, the datapaths generate acknowledges which are sent back to the execution control unit. Meanwhile, the branch detect unit evaluates the conditional signals generated by the datapath and determines the

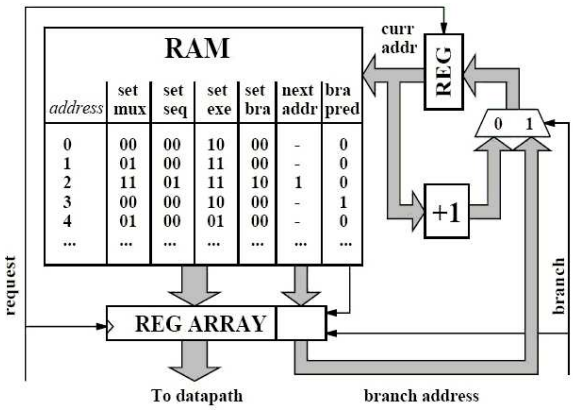

Figure 2. Microinstruction format

address of the next micro-instruction to be executed. After gathering the acknowledges, the execution control unit checks to see if the *done* bit (one of the global control fields in the micro-code) is high which specifies that the operation desired by the environment is completed and sends an external acknowledge signal to the environment and awaits the next invocation, otherwise it executes the next micro-instruction and repeats the global request-generation cycle.

Supporting a standardized way of programming the datapath topology is achieved by using small FSMs responsible for locally handling *Request/Acknowledge/Sequencing* (RAS) of their respective datapath unit, as dictated by the current microinstruction. The datapath units themselves then communicate with their local RAS block by using standard request/acknowledge protocols. This also makes the datapath modular which means datapath units can be easily replaced without changing any control structures.

Next Address Generation: The next micro-instruction is fetched in parallel to the execution of the current microinstruction to increase performance and hide control overhead. This approach works smoothly except when branches are involved. The problem of branch-prediction is solved by fetching the next micro-instruction but not committing it until the address selection is resolved. The designer programs each branch to be taken or not taken on studying the probability for each case using empirical data.

Microcode fields: The Figure 2 shows the various fields of a micro-instruction where each of the *set-* fields corresponds to a particular datapath (all fields are not shown in this figure). The *set-mux* field represents the input and output mux-selection signals. The *set-seq* bit controls whether that particular datapath element is to be executed in parallel or in sequence with respect to some other datapath element. The *set-exe* bit controls whether the particular datapath will execute during the current request cycle or not. The *set-branch* bit specifies whether this micro-instruction is a branch instruction or not. The *next-addr* field specifies the branch-address. The *bra-pred* bit specifies the branch-prediction strategy for the branch instruction i.e whether this branch has been predicted to be taken or not.. The *eval* bit specifies whether it's a conditional branch or not. The *sel-addr* bit selects either next sequential micro-instruction or the branch instruction. If the micro-instruction is not a branch instruction then *next-addr*, *bra-pred*, *eval* and *sel-addr* bits are dont-cares. Finally, the *done* bit specifies whether it is the last micro-instruction or not.

3.3. Click Modular Router

Click is a software architecture for building flexible and configurable routers that was developed at MIT [11]. Applications in Click are built by composing modules called elements which perform simple packet-processing tasks like classification, route-lookup, header verification and interfacing with network devices. A click configuration is a directed graph with elements as vertices; packets flow along the edges of the graph. Click is implemented on Linux using C++ classes to define elements.

We have looked into the Click IP router configurations given in [12] for our router implementation. A standard IPv4 over ethernet bridge router with two network interfaces has sixteen elements in its forwarding path. This router can be extended to support firewalls, dropping policies, differentiated services and other extensions by simply adding a couple of elements at the respective places. We have based our router implementation on the above Click configuration and used it to model every module's functionality by understanding the fine-grained packet-processing C++ description of each element on hardware. Since our microengine-based architecture needs modular datapaths, Click's modular and extensible router suits our architecture style.

4. Architecture

The architecture for our IP router consists of two types of microengines, *Ingress processing microengine* which does packet classification and *IP Header processing microengine* which does minimal processing on IP header like route-lookup, checksum computation etc. The high-level architecture block diagram is shown in the Figure 3. There are two ingress microengines corresponding to each port A and B. Each ingress microengine has one input FIFO queue and three output FIFO queues respectively. The input queue contains each incoming packet's id and its ethernet header's memory address. The ingress microengine upon receiving a request from input FIFO, starts executing and classifies each packet as an ARP reply, ARP query or an IP packet and sends the packet to the respective output FIFO. Since

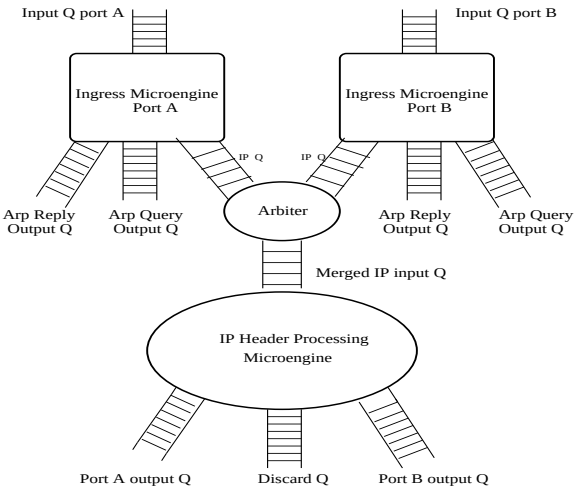

Figure 3. High Level Architecture of our Microengine-based Router

there are two FIFOs which contain IP packets from each ingress microengine and a single IP header processing microengine, there needs to be a synchronizing element which merges the two IP FIFOs into a single input FIFO queue for the other microengine. We have implemented an arbiter module which does this synchronization. The IP header processing microengine consists of a merged input FIFO and three output FIFO queues namely discard, portA and portB. A packet is sent to a discard FIFO only if it meets any of the criteria like the expired TTL etc. A packet is routed to the portA or portB fifo depending upon the destination address's next hop address.

4.1. Assumptions

The focus of this work is not on a very high performance IP router but rather on the asynchronous microcoded style used to build such an architecture. Thus, we have not used state-of-the-art algorithms for doing route lookups etc. and have made many assumptions such as that a NIC exists which does all the link-layer processing like encapsulating and decapsulating IP packet with ethernet header. We are assuming that a port processor exists which handles the memory management tasks like buffering of incoming packets into the memory and passing of header address to the ingress microengine.

4.2. Ingress Microengine

Ingress microengine classifies the packets into 3 types namely, ARP Query, ARP Reply and an IP packet. The block diagram for an ingress microengine is shown in the Figure 4.

Datapaths: This microengine consists of the following generic datapaths: an 8-bit Address register which stores the packet's memory address, an 8-bit ALU which calculates the offset address for a particular header field and Header memory. There is a 16-bit Header register which is byte addressable and two 16-bit comparators. There is also a 2-bit Flow-id register which stores the packet's classification result and finally a send-to-fifo datapath which enables one of the three output FIFOs by sending them a request signal based on the flow-id's value.

Microprogram structure: The microprogram for this microengine consists of four 67-bits wide microinstructions. Each microinstruction consists of the global control fields such as the branch address *next-addr*, *done* etc. and local control fields for each datapath unit such as the set-execute *se* etc. Since flexibility is one of the most important advantages of our architecture, we demonstrate this by incorporating classification rules in the microcode itself as this allows an easy upgradation of these rules without needing to change the underlying hardware. We can specify the header bytes that need to be compared with some value in the microcode itself along with the rule value.

Operation Overview: The input fifo queue upon receiving a packet, sends a request to the ingress microengine which if in an idle mode, starts executing by propagating a global request signal and latching the first micro-instruction *mi-1*. In *mi-1*, the ethernet header *"type"* field is read out from the memory and then the two comparators check in parallel if the type is an IP (0800x) or ARP (0806x). If it's an IP packet, then the flow-id "00" from the microcode gets selected and latched in the flow-id register and the microengine fetches the next sequential instruction *mi-2*. If it's an ARP packet, then the flow-id "01" gets selected and the microengine takes the branch and fetches *mi-3*. If *mi-2* gets executed then, send-to-fifo datapath is enabled and it sends the packet out to the IP output FIFO and the microengine jumps to *mi-4* which is the done instruction. If *mi-3* gets executed then the ARP *"type"* field is read from the memory and the two comparators again compute in parallel to check if it's an ARP reply or query. Depending upon the flow-id chosen by comparators results, send-to-fifo datapath sends the ARP packet to the reply or query output FIFOs and fetch the done instruction *mi-4*. The done instruction sets the done bit high, upon which the ingress microengine's ECU sends an ACK to the input FIFO and waits for the next packet to arrive on this port. In this microengine, most of the datapath operations are chained except for the parallel comparator evaluation.

4.3. Header-processing Microengine

This microengine buffers the IP-header from the memory into a header register file and processes it. We check

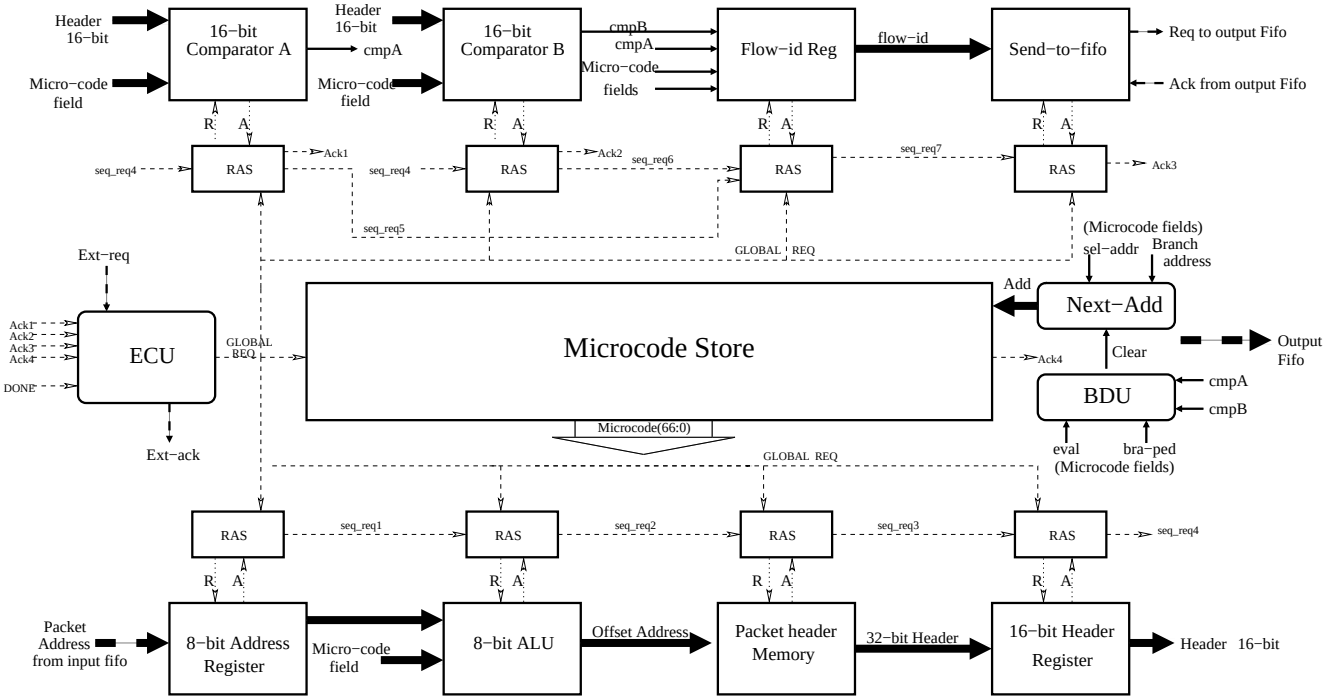

Figure 4. Block diagram of Ingress Microengine

if the IP header is valid i.e if it has valid *"version"* and header length *"hlen"* fields. We compute the checksum of the incoming packet's IP header and compare it against the *"checksum"* field. We also check for an expired *"ttl"* field. If the header is valid then we decrement the *"ttl"*, look-up the next hop destination IP address and recompute the checksum and write the latest value to the checksum field and send the packet to the respective port. If the header was invalid then we discard the packet by sending it to a discard FIFO.

Datapaths: The block diagram of a IP header processing microengine is shown in Figure 5. It consists mostly of generic data-paths except for CAMs which are specialized for network processing tasks. The generic datapaths in this microengine are similar to ingress microengine and include the 8-bit address register, header memory, 16-bit comparator, 16-bit temporary registers, 8-bit ALU, 16-bit ALU, flow-id and send-to-fifo datapaths. Amongst the specialized datapaths, we have one CAM which does route lookup and another one which implements a stateless firewall. We demonstrate the aspect of modular datapaths in our architecture by adding this filtering (based on source IP address) CAM in our design. Since we are storing the entire IP header (20 bytes without IP options) in this microengine, we have implemented our register file as a 4-byte wide and 16 word deep dual port SRAM which has one synchronous write port and an asynchronous read port. This helps us parallelize the writing of new header bytes into the memory and reading of already stored bytes for doing checks and checksum computation. We can see that except for CAMs and dual port register file, most of the datapaths are generic and common to both ingress and IP header processing microengines. Depending on the design requirement (if chip area isn't a constraint), we can implement our ingress microengine by changing IP header processing microengine's microcode.

Microprogram Structure: There are fifteen 111-bits wide micro-instructions for this microengine. The global and local control microcode fields are same as described earlier. This microprogram has microcode fields similar to the ingress microengine's like the flow-id values, read and write addresses for header bytes etc.

Operation Overview: The microengine receives an IP packet from the input FIFO and begins its execution and after it finishes processing, it sends out the packet header address into one of the three output FIFOs (portA, portB and discard). We have implemented the entire IP header processing algorithm in 15 micro-instructions and there may be opportunities for further optimizations. Out of 15 micro-instructions, 9 micro-instructions have 2 parallel execution clusters and one micro-instruction has 3 parallel execution clusters. The first micro-instruction *mi-1* latches the packet's memory address in the address register and then computes the offset address and reads the 4 header bytes from the packet memory and writes them to the dual port header register, followed by checking the first byte to be

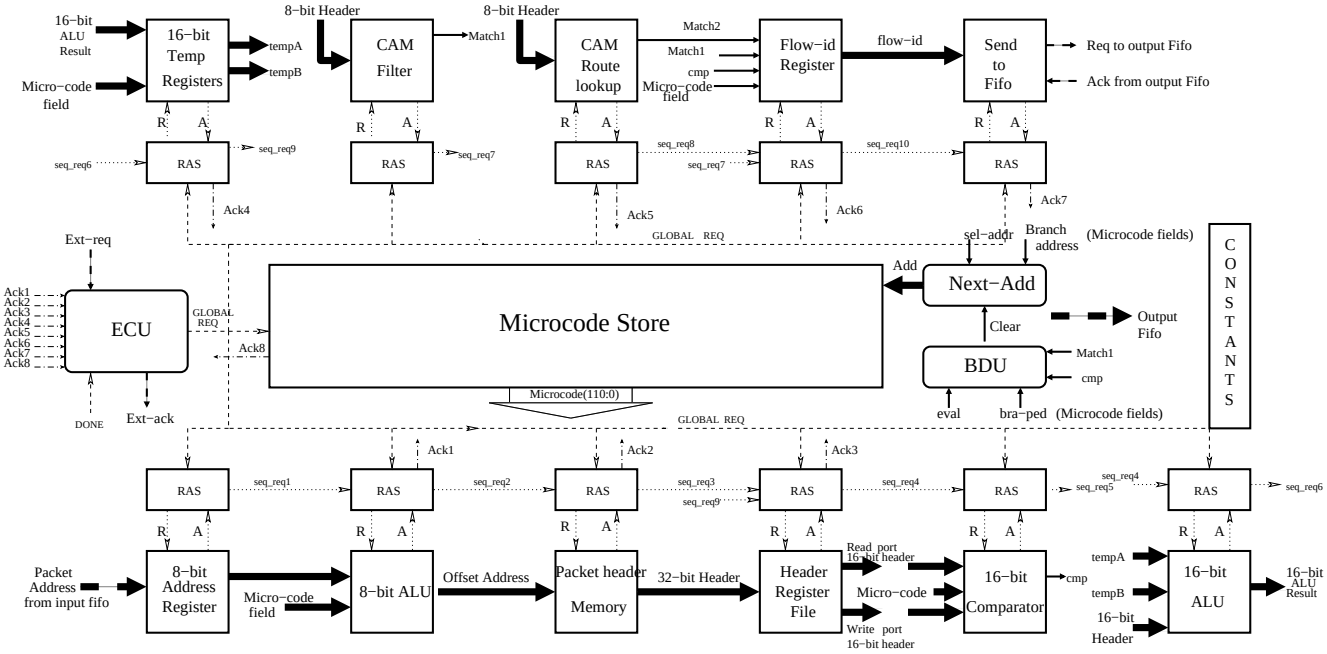

Figure 5. Block diagram of IP Header Processing Microengine

valid (i.e checking for *"version"* and header length *"hlen"* fields). If the check evaluates to true then we jump to *mi-3* otherwise we goto the next sequential *mi-2* which is the packet discard micro-instruction and sends the packet's address to discard FIFO and jumps to *mi-15*(done) micro-instruction. There are 3 micro-instructions in which if any of the checks fail then we jump to the discard micro-instruction (*mi-2*). One of the interesting features of our algorithm is the way we are computing checksum of an incoming packet. After the first four bytes are written to the register file, we add the two 16-bit consecutive header bytes (1,2 and 3,4) and store them in a temporary register. From this point onwards, we keep reading out two header bytes from asynchronous read port and add them to the value stored in temporary register in parallel to writing of header bytes on write port by ensuring that the read and write addresses are not same. We keep updating the temporary register. We store the total 16-bit sum for the entire IP header except for checksum and ttl fields in a separate temporary register. This value can later be used to calculate the checksum of the outgoing packet by just adding the latest *ttl* field value to it. We are also handling the writing of latest *ttl* and checksum values to the header memory in two micro-instructions.

Stateless Firewalling Extension: There are 3 parallel chains of datapath executions in the 6th micro-instruction of microcode. By enabling the set-exe bits of CAM filter and flow-id in execution chain 3, we can extend our architecture to support firewalling. We will also have to modify the global control bits in the 6th micro-instruction as we are evaluating a conditional branch micro-instruction (i.e. if the packet's source IP address is a blocked IP address then we need to discard it and jump to the discard micro-instruction). We have to modify a total of 10 bits in a 111 bits wide micro-instruction to enable firewalling. Upon modifying the microcode, we did not observe any changes in our architecture's per packet performance, as the execution chain 1 is the longest chain and enabling or disabling other execution chains (2 and 3) will not have any impact on the execution time. Hence, we have demonstrated that our architecture can be extended by modification of microcode.

5. Design Implementation and Evaluation

We have prototyped our asynchronous microengine based router architecture on the Xilinx SpartanII XS2S150 board made by Xess. The design has been validated with respect to functionality and the complete timing analysis has been performed. We have used Xilinx ISE tool-suite for synthesis, placement and routing and Modelsim for simulation. Most of the asynchronous control elements and datapaths have been specified by a behavioral VHDL description. Some of the asynchronous control units have been designed using Xilinx components (e.g. arbiter) in a schematic-based environment. Microcode stores and CAMs have been implemented using Xilinx's IP cores. We have implemented asynchronous FIFO as a micropipeline-based [18] flow-through FIFO. Most of our modules are not optimized for

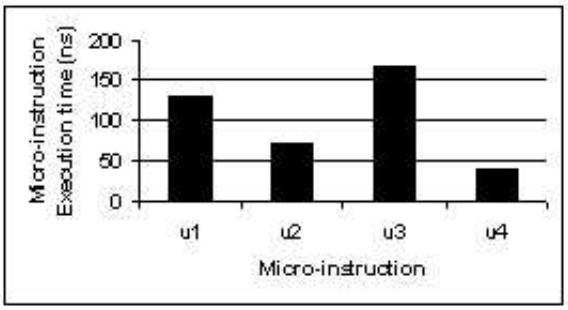

Figure 6. Micro-instruction Execution times of Ingress Microengine

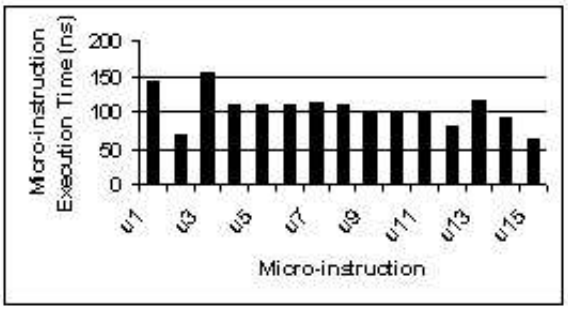

Figure 7. Micro-instruction Execution times of IP Header Processing Microengine

high-performance. Various interesting high-speed circuit-design styles can offer a much better performance if we are not implementing our design on an FPGA.

5.1. Evaluation

We are evaluating our architecture based on execution time of each micro-instruction, execution time of each microengine and evaluating the ASIC version with a similar Click router configuration. As our architecture is based on many assumptions, we will not be able to measure its realistic maximum packet forwarding rate. In the of evaluation of asynchronous design approach advantages, we have demonstrated the modularity aspect by introducing firewalling. Average case completion time is being evaluated by measuring the micro-instruction execution time. For an FPGA version, we cannot demonstrate the low power and low EMI advantages of our design approach. The power consumption of an FPGA is high and it is difficult to extrapolate the power-consumption result for an ASIC version of the same design. But, we believe that our asynchronous micro-engines will definitely have low power consumption as is the characteristic of asynchronous architectures [2]. **Micro-instruction execution time:** We measure each micro-instruction's execution time by measuring the time taken by global request's handshake. Long execution times for micro-instructions suggest long chains of datapath executions. Graphs 6 and 7 show the execution times for all micro-instructions. A synchronous version of both these microengines would have the worst case micro-instruction execution time for each of the micro-instructions. Thus, we can see the average case completion time advantage of our approach.

Packet-type	Execution time (ns)	μ-instructions executed
IP packet	353.97	3
ARP Reply/Query packet	514.32	4

Table 1. Ingress Microengine Per Packet Execution time

Packet-type	Execution time (ns)	μ-instructions executed
Correct IP packet	2460.17	14
IP packet with expired ttl	1653.85	10
IP packet with incorrect version or header length	516.00	4
Filtered IP packet	1309.85	8
IP packet whose route look-up failed	2434.53	14

Table 2. IP Header Processing Microengine Per Packet Execution time

Element	Time (ns)
Classifier	70
Paint	77
Strip	67
CheckIPheader	457
GetIPaddress	120
LookupIPRoute	140
DecIPttl	119

Table 3. Click Element's Per Packet Execution times taken from [12]

Microengine execution time: Microengine's per packet execution time is denoted by the time taken by the external

request's handshake to complete. The execution time depends from packet to packet. If any of the IP checks fail, then the microengine will execute fewer micro-instructions and hence, execution time will be reduced. Tables 1 and 2 show the execution times for each microengine for all the possible packet cases. For evaluation purposes, we are considering the total execution time for an ingress microengine for classifying an IP packet as 353.97 ns and the execution time for IP header processing microengine for processing an IP packet which gets sent to one of the output ports as 2460.17 ns.

Performance Evaluation: As a starting point, we are evaluating our router's performance with a similar Click software router configuration [12]. Per packet execution times for each of click software's elements running on a 700Mhz Pentium III PC hardware are shown in Table 3. Our ingress microengine has implemented Classifier and Paint elements. IP header processing microengine has implemented Strip, CheckIPheader, GetIPHeader, LookupIPRoute and DecIPTTL. Our evaluation is based on an approximate implementation of these Click elements as we do not know the details such as the number of route table entries in their element etc . Using the execution numbers from the table, we see that the ingress microengine's equivalent Click router has a per packet execution time of 147 ns which is 2.4x better than our ingress microengine. Similarly, IP header processing microengine's functionally equivalent click router has a per packet execution time of 903 ns which is 2.7x better than our microengine implementation. However, our implementation is an FPGA prototype using a SpartanII part. Extrapolation of our FPGA prototype performance results to an ASIC version will give a more realistic comparison. We have a 0.5u SCMOS process standard cell library of a generalized asynchronous microengine control and datapath modules. For a fair comparison, we measured the performance characteristics of a few blocks like ECU, asynchronous RAM etc. by implementing them on our Xilinx board and then compared them to the standard cell timings. The average execution time speedup that we observed is approximately 5x which factors in the I/O pins delay. Our speedup factor is conservative as this extrapolation doesn't take into account the interconnect speedup which would be much higher than 5x. Since the Spartan-II family is designed in a 0.25u, 5LM process technology and our SCMOS implementation is in a 0.5u process, we need to factor an additional speedup of 2 for comparing them in the same process technology (0.25u process). Our conservative performance speedup factor for FPGA to ASIC conversion is now 10x to a 0.25u CMOS process. Total per-packet processing time for a correct IP packet taken by an ASIC version of IP header processing microengine would roughly be 246.01 ns and that of an ingress microengine would roughly be 35.39 ns. Our asynchronous microengine based router's (ASIC version 0.25u) performance is 3.9x Click software's performance which is running on a 700MHz Pentium III (0.18u) and it will definitely have a lower power consumption. Assuming a linear scaling factor, our router's performance is 5.1x Click software's performance in the same 0.18u process technology. Since we have implemented our design on a FPGA, we were unable to use high speed transistor based circuit design techniques. By using better processes and design techniques for custom CMOS implementation, there is room for a lot more improvement.

6. Conclusion

In this paper, we have presented a case for asynchronous microengines in the network processing domain. This asynchronous microengine-based approach tries to fill the performance gap between a specialized ASIC and a more general network processor implementation. It does this by providing a microcoded framework which is close in performance to ASICs and is also programmable at the finer granularity of microcode. Thus, we believe that our asynchronous microengines can be used for designing an asynchronous network processor architecture which takes advantage of modularity, flexibility, high performance and low power consumption aspects of an asynchronous approach.

Acknowledgment: The authors would like to thank Gaurav Gulati for providing the timing characteristics of his standard cell library implementation of microengine modules.

References

[1] G. A. Andreas Nowatzyk and F. Pong. Design of the S3MP processor. In *Proc. of Europar Workshop*, 1995.

[2] C. V. Berkel, M. Josephs, and S. Nowick. Scanning the technology- applications of asynchronous circuits. In *Proc. of IEEE*, volume 87, pages 234–242, feb 1999.

[3] D. E. Comer. *Network Systems Design using Network Processors*. Prentice Hall, 2003.

[4] A. Davis, B. Coates, and K. Stevens. The post office experience: Designing a large asynchronous chip. In *Proc. of Hawaii International Conf. System Sciences*, pages 409–418. IEEE Computer Society, jan 1993.

[5] A. Davis and S. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, School of Computing, University of Utah, 1997.

[6] C. P. et al. A 50-gb/s ip router. *IEEE/ACM Transactions on Networking*, 6(3), jun 1998.

[7] H. Jacobson and G. Gopalakrishnan. Application-specific asynchronous microengines for efficient high-level control. Technical Report UUCS-97-007, School of Computing, University of Utah, 1997.

[8] H. Jacobson and G. Gopalakrishnan. Asynchronous micro-engines for high-level control. In *Proc. of 17th Conf. on Advanced Research in VLSI(ARVLSI97)*, 1997.

[9] H. Jacobson and G. Gopalakrishnan. Application-specifi c programmable control for high-performance asynchronous circuits. In *Proc. of IEEE in a special asynchronous issue*, volume 92, February 1999.

[10] S. Keshav and R. Sharma. Issues and trends in router design. *IEEE Communications Magazine*, 36(5):144–151, may 1998.

[11] E. Kohler. *The Click Modular Router*. PhD thesis, Dept. of Computer Science, MIT, 2000.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[13] J. Kuskin and D. O. et al. The stanford FLASH multiprocessor. In *Proc./ of 21st Annual International Symposium on Computer Architecture*, pages 302–313, 1994.

[14] L. Peterson and B. Davie. *Computer Networks- A system's approach*. Morgan Kaufmann, 1999.

[15] N. Shah. Understanding network processors. Master's thesis, Dept. of EECS, University of California, Berkeley, sep 2001.

[16] K. Stevens. The soft-controller: A self-timed microsequencer for distributed parallel architectures. Technical report, School of Computing, University of Utah, 1984.

[17] W. R. Stevens. *The Protocols(TCP/IP Illustrated Volume 1)*. Addison-Wesley, 1994.

[18] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[19] K. Y. Yun. *Synthesis of asynchronous controllers for heterogenous systems*. PhD thesis, Stanford University, 1994.