Development of an ASIP Enabling Flows in Ethernet Access Using a Retargetable Compilation Flow

K. Van Renterghem, P. Demuytere, D. Verhulst, J. Vandewege, Xing-Zhi Qiu
Department of Information Technology
Ghent University / IMEC
Sint Pietersnieuwstraat 41, 9000 GENT, Belgium
Koen.VanRenterghem@intec.ugent.be

Abstract

In this paper we research an FPGA based Application Specific Instruction Set Processor (ASIP) tailored to the needs of a flow aware Ethernet access node using a retargetable compilation flow. The toolchain is used to develop an initial processor design, asses the performance and identify the potential bottlenecks.

A second design iteration results in a fully optimized ASIP with a VLIW instruction set which allows for high degree of parallelism among the functional units inside the ASIP and has dedicated instructions to accelerate typical packet processing tasks. This way, a single processor is capable of handling the complete throughput of a gigabit Ethernet link.

To reach the target of a 10 Gbit/s Ethernet access node several processors operate in parallel in a multicore environment.

1. Introduction

Ethernet based access platforms are becoming the vehicles to offer triple play services (voice, video, data), as well as innovative and advanced services. Legacy networks for voice, data and video are converging into a single network, new services are emerging and Ethernet is becoming the transport protocol of choice [1]. The drivers of the success of Ethernet in enterprise networks are also driving the access technology. It is hence expected that future high bandwidth access infrastructures with fibre to the cabinet, building or home will be Ethernet based.

A means to support existing and new services in Ethernet based access is the introduction of ATM-like features such as 'flow awareness'. Traffic will be treated differently depending on the subscriber to whom it belongs, and the type of service it represents [2] [3].

In a flow based approach several features of packets are extracted (layer 2 up to layer 4) in an access node and then used as input for classification. The outcome of this process reveals the service tied to the inspected packet, allowing it to be further processed and routed accordingly.

The scope of this paper is the design of a processor active in the data path of such a system. The researched solution must be scalable, flexible, easy to program and maintain. These design specifications resulted in the development of a packet processing ASIP to be deployed in an FPGA based multicore processing architecture supporting line rates up to 10 Gbit/s and corresponding packet rates as high as 14.88 MPacket/s.

In order to minimize the processing time, algorithms are often described in custom languages or assembly. Here, a retargetable C compiler allows easy ASIP code development, while maintaining efficiency. The development is targeted at a Xilinx Virtex4 LX200-11 FPGA.

2. Data Plane of a Flow Aware Access Node

A complete flow aware Ethernet based access node consists of several building blocks each dedicated to specific tasks such as parsing, classification, filtering, queuing, statistics, traffic engineering, packet manipulation, etc. The scope of this paper is the development of an ASIP optimized for the three key building blocks: parser, classifier and packet manipulator. The specifications of the internal architecture of the ASIP are derived from a survey of typical algorithms running within the aforementioned building blocks.

Parsing is targeted at IPv4/IPv6 encapsulated in Ethernet with TCP or UDP as transport protocol. The algorithm supports a wide variety of Ethernet standards such as (stacked) VLANs and even the upcoming envelope format defined in IEEE 802.3as with frames up to 2 KB. PPPoE, often used in xDSL access networks as authentication protocol, is also supported.

The parsing algorithm decodes the protocol stack, extracts fields used to define a flow and stores them in a data structure called a 'ticket'. Control plane traffic such as

978-3-9810801-2-4/DATE07 © 2007 EDAA

ICMP, IGMP, ARP and others will be identified and flagged in the ticket to be forwarded to a dedicated control plane processor. The verification of checksums is not part of the algorithm. At Layer 2 the CRC can easily be verified at wire rate before assigning the packet to an ASIP. The verification of Layer 3 and 4 checksums requires in depth knowledge such as the type of payload, optional headers, etc. Therefore it is performed in a dedicated hardware block using the information gathered during parsing and classification.

The **classification** algorithm transforms the extracted header fields into a search key which will be applied to an external TCAM (Ternary Content Addressable Memory). The result of this process is a 'flow identifier', used throughout the system to perform further processing.

Based on the flow identifier, the **packet manipulator** can alter header or payload fields and recalculate the checksums. Typical operations are the insertion of a VLAN tag, the decrement of the IP TTL field, etc...

These three algorithms were translated into C code and served as a starting point for the ASIP architecture. From then on several iterations were performed exploring architectural aspects such as instruction level parallelism (ILP) and extensions of the C language with compiler known functions (aka intrinsics) to minimize the execution time.

3. Design Flow

The ASIP was built using the Chess/Checkers retargetable tool suite of Target Compiler Technologies [4] for the design, programming and verification of the ASIP cores. The environment consists of the following toolchain:

Chess: a retargetable C compiler using graph-based modelling and optimisation techniques [5], to deliver highly optimised code for specialised architectures. The compiler comes with a retargetable assembler and disassembler called Darts, and a retargetable linker called Bridge.

Checkers: a retargetable instruction-set simulator (ISS) generator that produces a cycle and bit accurate ISS for the target processor. Source-level debugging and code profiling are supported.

Go: a hardware description language (HDL) generator that produces a synthesisable register transfer level HDL model of the target processor core. Through APIs, users can plug in their own HDL implementations of functional units and of the memory architecture.

Risk: a retargetable test-program generator that can generate assembly-level test-programs for the target processor with a high fault coverage. These test programs can then be executed both in the ISS and in the HDL model of the processor, to check for the consistency of both models.

The instruction-set architecture of the processor is described in a high level language called nML. This description is then used to generate the compiler and ISS in order to get quantitative information about the performance of the architecture. The impact of design choices such as ILP and the extension of the compiler with compiler known functions can be easily evaluated. The next step is the automatic generation of VHDL for the architecture, which can be synthesized and verified against the instruction set simulator.

4. Multicore Support

The requirement of handling the maximum frame rate (14.88 Mpackets/s) of a 10 Gbit/s Ethernet link leaves an ASIP only 67 ns to finish its tasks. When running at 120 MHz only 8 clock cycles are available. Clearly a multicore architecture is needed to handle the complete throughput. Multiple cores can be put in a pipeline, each performing a part of the complete processing algorithm of the same packet. While easy to implement, this architecture requires the packet algorithm to be split in equal parts to avoid idle time on the ASIPs. An alternative is to use a pool of processors all working in parallel on different packets. ASIPs running idle are avoided, but each ASIP must have access to all external peripherals and high speed busses will need to traverse the FPGA fabric. A mixture of both approaches, a pipeline of ASIP pools, combines the best of both worlds.

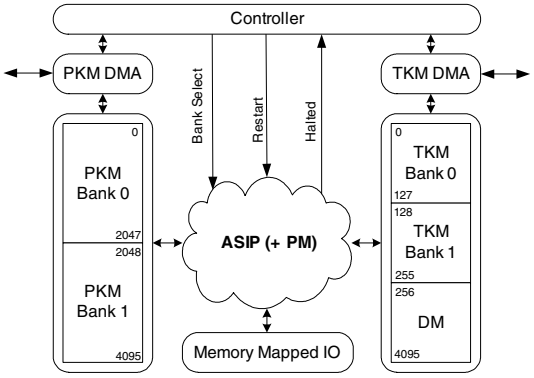

Figure 1. External ASIP interfaces

To support any of the aforementioned setups, a suitable memory architecture, taking optimal advantage of the resources scattered over the FPGA fabric, was developed. Each ASIP has access to two 4 KiB dual ported memories, implemented in on-chip Block RAMs. One port is exclusively reserved for the ASIP and the other one interfaces with the multicore environment. The two physical memories are divided into regions, according to their function. The Packet Memory (PKM) consists of two equal banks of 2 KiB and stores the packet to be processed. The organization of the remaining memory is somewhat different: 256 bytes Ticket Memory (TKM) and 3840 bytes Data Mem-

ory (DM). The ticket memory itself is further divided into two equal banks of 128 bytes each. This memory stores the ticket data structure containing the context linked to a packet. Each ASIP can add results to it or use it as input for its own tasks. Apart from these two physical memories, a dedicated 32 bit memory interface is available used to map external peripherals onto its memory space.

The goal of the memory architecture is to keep the processor occupied at all times. The bank select signal makes the processor operate on a specific bank. While packet processing is ongoing, the second memory port can be accessed to write new data into, or read previous results from the inactive bank, without the risk of data corruption. Buffers for rate adaptation are not needed in the system as each DMA engine interfaces with two 32 bit wide Block RAMs combined into a single 64 bit wide bus running at 160 MHz. The DMA engine of each ASIP in the pool interfaces with a dispatcher and a collector block. The latter assigns new data to the individual ASIPs and the former reads out the results to be transferred to the next processing step. The dispatch/collect mechanism ensures that the order of the packets leaving the ASIP pool remains unchanged.

For the programmer, bank switches are transparent and the mechanism needs minimal support in the code. Each time the running algorithm concludes its work on a packet, a 'halt' instruction must be issued. This instruction drives the external halted signal, flushes the processor pipeline, stops the issuing of new instructions and resets the program counter to the beginning of the algorithm.

The main advantage of this memory architecture is that it hides latency. The use of a central memory would require aggregation and arbitration logic, which inevitably leads to processor stalls. Cache memories can try to improve on this, but as the required amount of storage for each ASIP is quite low, the memory can be kept local entirely. Furthermore, the use of banks removes the time needed to copy out the results and take in new data. The architecture also allows for excellent scalability: in our setup three pools of 10 ASIPs have been implemented, processing the data of a 10 Gbit/s Ethernet connection.

5. Initial Design Phase

5.1. Introduction

The use of a retargetable tool suite allowed us to explore several design options for an ASIP using the top level memory architecture described in the previous paragraph. Two major revisions of the ASIP were designed. An overview of our initial architecture will be given in this section together with the insights gained. The remainder of the paper will discuss the enhancements to the architecture and compare it performance wise to the original design.

5.2. Data Path

A schematic overview of the ASIP data path is shown in Fig. 2. The ASIP contains several functional units such as the Arithmetic and Logical Unit (ALU), a Checksum Unit, a Content Addressable Memory (CAM) and the Address Generation Units (AGUs).

The general register file (REG) and the functional units are both 16 bit wide. As most packet header fields manipulated in the functional units do not need more than 16 bit, the cost for a complete 32 bit architecture is rather high. Only copy operations directly between memories can be 32 bit wide.

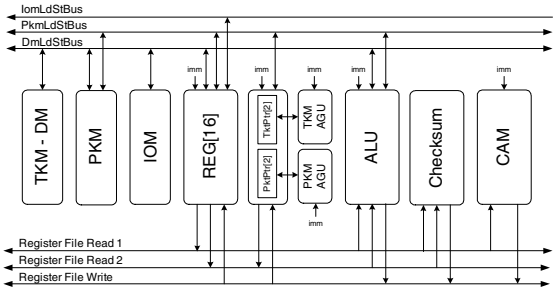

Figure 2. Initial ASIP Data Path

The memory addresses are calculated in two Address Generation Units (AGUs), one for PKM and another one for DM/TKM. Each AGU works on a dedicated pointer register file with two entries and supports immediate and postincrement addressing.

5.3. Program Control

The performance of packet processing code depends largely on the efficiency (cycle consumption) of control instructions. The processor and compiler support delayed jump and branch instructions. The code is reordered to fill delay slots with meaningful instructions, if no instruction can be inserted a 'NOP' is executed in the delay slots [6].

To minimize the impact of delay slots, the Program Counter (PC) is set as early a possible in the pipeline. Unconditional jumps and branch instructions to an immediate address set the PC in the second pipeline stage, requiring only one delay slot. Conditional jumps or computed branches are taken one cycle later, as their execution depends on the result of the status register, which is set in second pipeline stage.

Next to these general purpose branch and jump instructions, some special instructions were added targeted at accelerating C-style switch-case constructs, as often seen in protocol stack decoding. A 16 bit wide Content Addressable Memory (CAM) based approach was chosen. This

lookup mechanism can be area-efficiently implemented in an FPGA based on SRL16 shift registers [7]. The CAM delivers an offset to a lookup table in memory, containing the actual branch target address, which is then applied to the program memory.

5.4. Checksum Calculation

Packet manipulation algorithms also need to deal with the checksum. Efficient algorithms have been developed to incrementally recalculate the checksums used throughout the TCP/IP stack [8]. Essentially, the checksum is updated with the ones complement subtraction of the new and the old contents, taking into account that packet headers contain the bitwise inversion of the actual checksum value. A dedicated functional unit was added to the ASIP providing hardware support for this mechanism.

5.5. Conclusion

In DSP applications it is often possible to identify a clear bottleneck. The nature of the algorithms running on our ASIPs is quite different and a global ASIP optimization is needed to reach the performance goals. The proposed architecture was able to run the desired parsing, classification and packet manipulation algorithms, but profiling data and analysis of the assembly code revealed a number of possible optimizations, which are listen below.

1. The ASIP has a rather complex load/store architecture with several busses between the functional units, memories and registers, yet some interesting paths are still missing. A more generic approach, where any two storages are interconnected, provides a far more flexible solution.
2. A register file with eight entries is sufficient to let the envisaged algorithms execute without register spilling.
3. More pointer registers are needed. Often cycles are lost to save and restore the register contents. (register spilling)
4. Updating packet fields and the associated checksum is too time consuming as at least three steps are needed. First, old packet data has to be fetched from memory and subtracted from the current checksum. Next, the packet can be overwritten with new data. And as a last step the modified packet contents has to be fetched again to add it to the checksum.
5. A CAM is used to accelerate C-style switch case statements. A speed gain can be achieved for wide switch-case statements, but typically the number of jump targets is rather low. The architecture should be optimized for this typical usage.
6. The AGUs are the only units capable of operating in parallel with other functional units. An instruction set with a higher degree of instruction level parallelism can reduce the cycle consumption.
7. An analysis of the place and route results in the FPGA revealed that the instruction set decoding is one of the critical paths. An instruction set with straightforward decoding can save area and increase speed.

6. Final Design Phase

6.1. Data Path

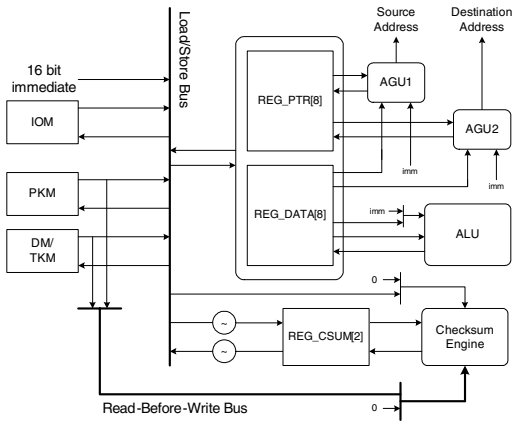

Figure 3. ASIP Data path final design

Fig. 3 shows the new design. The ASIP no longer has several busses between memories and registers. The single 32 bit Load/Store bus is a far more generic and flexible solution as it interconnects any two storages in the design. The single Load/Store bus also required the AGUs to be redesigned. They are no longer dedicated to a specific memory. In stead, the first AGU is used to calculate source addresses and the second one is used to provide the destination addresses. Both AGUs now access a larger, shared pointer register file capable of storing eight addresses. Furthermore, three addressing mechanisms are now supported: immediate addressing, postincrement addressing and indexed immediate addressing. The size of the general register file was cut in half as this is sufficient for most real life algorithms running on the ASIP. On a whole, the register files have a comparable area cost to the first design, but are better dimensioned for the envisaged algorithms and can reach higher speeds because the 16 to 1 multiplexers on the outputs were replaced with 8 to 1 multiplexers.

The instruction set has a higher degree of instruction level parallelism, resulting in a VLIW ASIP with 72 bit instruction words stored in a dedicated program memory

(Harvard architecture). The wide instruction word allows for a Load/Store operation to be combined with two AGUs updating the pointer registers, a checksum update and an ALU operation. Compared to the previous design 31 additional instruction bits are used, but they have been traded for simplified instruction decoding and a higher degree of instruction level parallelism.

6.2. Program Control

A CAM unit is an efficient means to accelerate wide C-style switch-case constructs. However, next to these wide branches, the code contains a higher number of switch-case constructs with a relatively low number of branch targets. The CAM fails to provide a significant acceleration compared to standard if-then-else constructs in these cases. An alternative to the CAM based solution, performing better for smaller switch-case constructs, has been implemented with two new instructions: multicompare and multibranch.

The multicompare instruction compares a value residing in the general register file with three 16-bit immediates encoded in the instruction word. If a match is found the status register (SREG) will be updated accordingly, as would happen with any logical operation. The multicompare instruction also has a second output, called the branch register (BREG). In case of a hit, this register will store the index of the matching immediate. In case of multiple hits, the first hit has priority.

Next to this basic mode of operation, two multicompare instructions can also be cascaded. The first instruction will overwrite the contents of SREG and BREG. The second one will only update the output registers of the instruction if no previous hit is detected. This way the priority encoding is kept in case of multiple hits. This cascade allows to store the result of up to six comparisons in the BREG.

If we are dealing with byte values, the execution of a cascade of two multicompares would be inefficient. To alleviate this problem, the three 16 bit immediates in the instruction word can also be interpreted as six bytes. This way the comparison is done in a single instruction.

Once the multicompare instruction is executed, a branch has to be chosen based on the outcome of the comparison. This is the task of the multibranch instruction. A multibranch instruction carries seven branch targets in the instruction word. The output of the branch register acts as a selector to set the PC. If no hit is found, BREG is zero and the default branch target is selected.

The combined multicompare- multibranch mechanism provides an efficient mechanism to accelerate switch-case structures of up to six branch targets. The main difference is that the code for each case needs to be a function and a default branch target must always be present. Despite these restrictions, these instructions prove to be very useful, especially for decoding protocol stacks.

The main advantage of the CAM based approach of the initial design is the ability to efficiently support wide switch-case constructs, as shown in Fig. 4: The cycle consumption remains constant. The main disadvantage is the poorer performance for smaller switch-case constructs due to the time needed to access the lookup table. An analysis of the code shows that the majority of the switch-case constructs have less then six entries. Therefore, the multibranch instruction is the optimal solution.

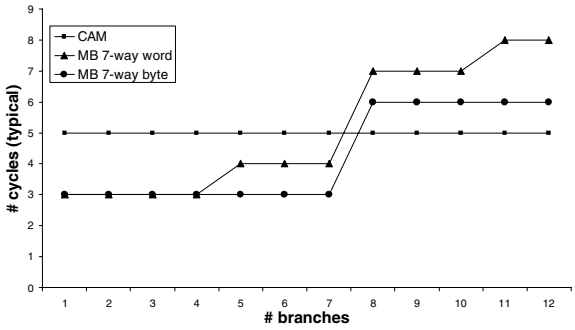

Figure 4. Multibranch solutions compared

6.3. Checksum Calculation

The performance of the incremental checksum update mechanism is greatly increased using the 'read-before-write' mode of the Xilinx BRAMs. This feature makes the old memory contents appear at the read port every time new data is written in. Using this feature we update a dedicated bus (Read-Before-Write Bus) each time data is written to a memory. A single 8, 16 or 32 bit wide store instruction to memory is sufficient to produce all necessary data to update the checksum: The new data appears on the Load/Store bus and the old data is made available on the Read-Before-Write Bus one cycle later (Fig. 3). Both values then can be combined to update one of the two dedicated checksum registers.

7. Performance Analysis

7.1. Area and Speed

The design was implemented in a Xilinx Virtex 4 LX200-11 FPGA. The final design of the ASIP core uses about 1344 slices, which is about 200 slices smaller than the original design. The total design, including the controllers supporting the interaction with the top level takes about 1600 slices. This allows for 48 ASIPs to be placed

Packet Type	Design 1	Design 2
IPv4 + UDP/TCP	85	57
IPv6 + UDP/TCP	-	57
VLAN + IPv4 + TCP	88	59
ICMP echo request	83	56
ARP	43	28

Table 1. Parser performance (cycles)

inside the FPGA, if extra logic for external interfaces is not taken into account. The biggest entities in the final design of the ASIP are the register files (32%), the checksum engine (10%), the ALU (9%), the decoder (8%) and the global load/store bus (4%).

A post place and route frequency of 120 MHz can be reached, which is a significant improvement over the initial design, reaching only 100 MHz. An ASIP spending 60 cycles on a packet will be able to process 2 MPackets/s at these clock rates, which is sufficient to handle the maximum throughput of 1.488 MPackets/s of a gigabit Ethernet link. The subsequent paragraphs will show that a single ASIP is capable of handling the parsing, classification or packet manipulation for a Gigabit Ethernet link.

7.2. Cycle consumption

Various types of packets have been processed by the **parsing** algorithm, up to layer 4 if applicable. Table 1 shows a cycle performance increase of 33% for all types of packets. It is worth noting that the IPv6 performance is on par with IPv4 thanks to the high degree of parallelism in the processor.

The **classification** algorithm transforms the information stored in the ticket into a search key, applies it to the TCAM and reads back the result. The total operation, including TCAM latency, takes 40 clock cycles in the final design. The relative performance increase compared to the initial design is minor as the TCAM latency consumes about half of the cycle budget. Further optimization should focus on mechanism to hide this latency.

Table 2 shows the performance figures for three typical **packet manipulation** scenarios, modifying the packets up to layer 4. Performance gains up to 47% are observed, especially for more complex packet manipulation scenarios.

8. Conclusion

A processor architecture for a packet processing ASIP, tailored to the needs of a flow aware access node, was presented. The processor was developed using a retargetable design flow, allowing to asses the performance of a given architecture and easily adapt it to alleviate shortcomings. The result is an ASIP capable of handling parsing, classification or packet manipulation functionality for 1 Gbit/s Ethernet links. Furthermore, the ASIP can work seamlessly in a multi core environment, allowing excellent scalability to build a 10 Gbit/s flow aware access node.

Scenario	Design 1	Design 2
Insert VLAN tag	27	23
Replace MAC Addresses Decrement IP TTL Update IP Checksum	33	24
Replace MAC Addresses Decrement IP TTL Change IP Source Address Change TCP Source Port Update IP + TCP Checksum	51	27

Table 2. PacMan performance (cycles)

9. Acknowledgements

Part of this work has been supported by the 'Institute for the Promotion of Innovation by Science and Technology in Flanders' (IWT) through the IWT project SERENA.

References

[1] K. Moerman, J. Fishburn, M. Lasserre, and D. Ginsburg. Utah's UTOPIA: an ethernet-based MPLS/VPLS triple play deployment. *IEEE Communications Magazine*, 43(11):142–150, Nov. 2005.

[2] S. Oueslati and J. Roberts. A new direction for quality of service: flow-aware networking. In *Next Generation Internet Networks, 2005*, pages 226–232, Apr. 2005.

[3] Y. Jiang, P. Emstad, A. Nevin, V. Nicola, and M. Fidler. Measurement-based admission control for a flow-aware network. In *Next Generation Internet Networks, 2005*, pages 318–325, Apr. 2005.

[4] Target compiler technologies, http://www.retarget.com.

[5] J. Van Praet, D. Lanneer, W. Geurts, and G. Goossens. Processor modeling and code selection for retargetable compilation. *ACM Transactions On Design Automation of Electronic Systems*, 6(3):277–307, 2001.

[6] W. Stallings. *Computer Organisation and Architecture - Desiging for performance*, page 484. 6 edition, 2003.

[7] J.-L. Brelet and B. New. Designing flexible, fast CAMs with virtex family FPGAs. *Xilinx Application Note XAPP203*, September 1999.

[8] A. Rijsinghani. Computation of the Internet Checksum via Incremental Update. RFC 1624 (Informational), May 1994.