A Gigabit IP Core for Embedded Systems

Nicholas Tsakiris and Greg Knowles

Abstract**—** In embedded systems a common requirement is to provide some form of communication between the system and a server. In the case of IPTV (Internet protocol TV), the purpose is for streaming content, in other applications it is for sending blocks of data between the two machines for processing. This paper provides a solution in the form of an IP based Gigabit Ethernet connection with a specially-designed IP layer implemented directly in hardware. The IP core implements the ICMP, UDP and the new UDP-Lite standards, it was designed in VHDL and after testing and synthesis, found to use approximately 1000 slices of the Xilinx Spartan 3 FPGA, and runs at full Gigabit ethernet speed (125 MHz), [3], [4], [5], [6], [11], [7].

Keywords**—** Ethernet, IP, UDP-Lite, ICMP

I. Introduction

In complex, embedded systems a common requirement is to provide some form of communication between the system and a server. Whether the purpose is for streaming content, or simply sending blocks of data between the two machines for processing, the decision to implement this communication should be made in such a way that the chosen solution satisfies a number of important features. Ideally it should be fast, affordable, feasible and upgradeable, and the more ubiquitous the chosen communications platform is, the less likely it is to have any significant flaws and also facilitates an easier and more robust implementation.

There are many different connection standards available for this task, each with particular hardware and software requirements. Some may require special hardware to connect to a computer using proprietary connectors or boards, which increases the cost and reduces the flexibility of any solution. Other options may provide only limited access to the internal structure of the interface, limiting the ability of the developer to modify the interface to suit their needs. There may be an extra cost to provide the code for the interface, separate from any supplied hardware, which can also tax design budgets.

Nicholas Tsakiris is with the School of Computer Science, Engineering and Mathematics at Flinders University, in Adelaide, Australia, email:tsak0011@flinders.edu.au

Greg Knowles is with the School of Computer Science, Engineering and Mathematics at Flinders University, in Adelaide, Australia, (phone: 618-8201-5041), email:gknowles@infoeng.flinders.edu.au.

Manuscript Received May 3, 2007; Revised November 28, 2007

This paper provides a solution in the form of an IP based Gigabit Ethernet connection with a specially-designed IP layer implemented directly in hardware to facilitate the connection. Based on the Ethernet standard, we are able to provide a means of communication that is widely used with modern networked computers, but without the added cost of most other commercial IP solutions. Keeping the design clean and simple was critical in creating the interface code, as the overall goal has been to provide an interface which is cheap, open, robust and efficient, retaining the flexibility a developer might require to modify the code to their needs.

For reasons of efficiency, the implementation uses only the following protocols for communication: ICMP (also known as a “ping”), UDP and UDP-Lite. The Finite State Machines which control operation of the interface are covered in depth, with an explanation of their inter connectivity and how they fit in the data-flow between the computer and the server.

Error correction and reliability to ensuring the quality of data are discussed. We use – tags, which are values inserted into the payload of each packet to detect any missing or out-of-sequence packets. We also use checksums/CRC values to evaluate packet integrity. The IP core is able to recover gracefully from severe situations such as truncated or corrupted packets without affecting the rest of the network stack. It is also capable of dealing with multiple packets at the same time without corruption.

The IP core was designed in VHDL and after testing and synthesis, used approximately 1000 slices, running at just over 125 MHz on a Xilinx Spartan 3 FPGA (XC3S5000). The design is sufficiently small (around 3%) to allow for other, more size able programs to run on the same FPGA with the remaining space, and the system is capable of working at full speed with the Gigabit Ethernet standard. The final design was fully verified on a SPARTAN3 prototyping board.

Finally, the conclusion covers the key aspects of the design which distinguishes itself from other commercial IP implementations, namely, speed, cost, flexibility and an open architecture. There are a few improvements which could be made to the design however, such as adding full ARP support to simplify the configuration of the interface, an automatic payload flush function which

would simplify the sending of large blocks of packets back to a computer in certain situations, and TCP support for the cases where it might be useful.

II. The IP core

A. Finite State Machines

Implementation of the IP core is separated into three distinct Finite State Machines (FSMs). One to read an incoming packet (fsm_read), one to build a packet for transmission (fsm_packgen) and one to actually perform the packet transmission (fsm_send). These three FSMs are documented in the following sections.

The flow of data between the FSMs and the other layers of the design is represented by the following diagram:

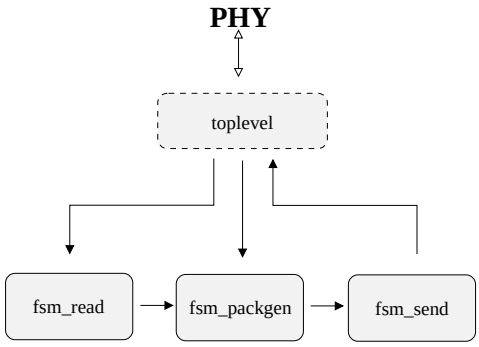

Fig. 1. fsm_read state flowchart

Data begins at the PHY (Physical Layer), which is connected to the toplevel component (the interface to the ethernet layer). Depending on the settings in the toplevel, as well as the protocol of the incoming packet, there are three potential paths for the data to take. From the toplevel:

1) [Mirror] - Data progresses from *fsm_read... fsm_packgen... fsm_send... toplevel* and streams the output back to the PHY. The payload data sent to the PHY is always identical to the packet received; this mode is used for self-test.
2) [Typical] - Data progresses from *fsm_packgen... fsm_send... top level* and streams the output back to the PHY. In this path the fsm_read component is bypassed, and in fact the PHY is not used to trigger the FSMs at all. This path is triggered by the system to send and receive data from the server, and therefore the payload content is constructed as necessary and as such, provides the typical use of the interface for the majority of data transfers.
3) [Storage] - Data is read using *fsm_read* and progresses no further. In this path the latter two FSMs are totally bypasses and no return packet is sent. This path is used for storing payload data for extraction by another component of the system.

To save space we will only describe in detail the operation of the first of these state machines, FSM_READ. The others operate in a similar fashion.

B. FSM_READ

The purpose of fsm_read is to parse and process an incoming packet. The direction of the data flow, for the purposes of clarifying the context of an incoming packet, is from the server *to* the core. As we are dealing with only a small selection of protocols and virtually anything could be transmitted down the wire, the FSM needs to be able to correctly interpret the packet, deal with it as appropriate and also deal with any packets which do not match the required protocol set (Figure 5).

When the interface begins execution, the fsm_read code will initialize and hold in a waiting state. The FSM waits for the RX_DV signal to go high before proceeding any further. This signal is controlled by the Ethernet PHY of the network and is raised when a packet is received by the layer. At this point the FSM will begin checking for the ethernet preamble:

```
55 55 55 55 55 55 55 D5
```

Once a packet's preamble is verified, the FSM can begin the real work of processing the packet contents. The first stage is to analyze the header information of the packet. All header data is saved to the RAM1 block; this data can then be quickly and easily accesses by the various components of the code. All the RAM's used are dual port (simultaneous read and write) Xilinx Block RAM's. At a certain point in the header the IP protocol will be specified. If the protocol number matches any of the implemented protocols the system can handle, the state continues processing as normal. Otherwise, we reject the packet by advancing to a waiting state which will only advance once RX_DV goes low. This bypasses the rest of the packet since its of a form we do not need to deal with.

After reading the entirety of the packet header, the system begins to read the payload (application) data, which is where the useful content of the packet is located. The data is saved to the FIFO1 block. Once all the data has been processed, a CRC is calculated against the packet and matched to the FCS (Frame Check Sequence) supplied by the packet; if these two are identical, the packet was sent uncorrupted and so

the content is considered sane. If the values are different however, a control signal is raised to represent a packet failure.

The states in fsm_read are commented below:

Preinit1 - Preinit2 - Preinit3

These three states, executing one after the other, have the purpose of providing a clean reset to the FSM. They are executed upon power-on of the system and are also executed when a packet has completed being processed by fsm_read. They set important control signals to their default values so that future packets are not corrupted by unknown signal states. Each of the preinit states perform the same task, but by using three states instead of one we can ensure total confidence in the state of the signals once a packet is received.

Waitforpacket

A holding state which remains dormant until both the RXDV and dcmlock signals go high. When RXDV goes high, this means the Ethernet layer has received a new packet containing valid data (note - valid in the case of being able to be understood by the Ethernet layer; corrupted data would fail checksum calculations later on in the FSM). dcmlock represents the state of the clock signal, when dcmlock goes high can the system operate with confidence that the clocking signals are stable.

SDFpreamble

Preceeding the actual packet content is the preamble and SDF (Start Frame Delimiter). This is represented by the 55 55 55 55 55 55 55 D5 sequence which exists in every packet, the 55s acting as the premable and the D5 as the SDF. If something else interrupts this sequence or cannot be finished for whatever reason, the packet is considered damaged and control is moved to the **Holduntilfinished** state.

SDFfinalcheck

Assuming the previous state ran with success, the system keeps counting through the necessary number of bytes in the packet until it reaches the point where the SDF (D5) should exist. If it does, the system proceeds as normal, otherwise the packet is considered damaged and control is moved to the **Holduntilfinished** state.

Striptoram

Satisfied the packet is readible enough to pass initial testing, the system uses this state to store the packet's header information (both IP plus the protocol's header) into RAM1. Several fields are also copied from the packet into registers for use later, such as the specific packet protocol, the length of the data, any checksums present depending on the protocol, and any protocol-specific fields which are important. Another check is performed in this state as well - if the scanned protocol number does not match one implemented by the system (eg. the packet might be TCP which the network code has no handler), the packet is ignored and control is moved to the **Holduntilfinished** state.

Striptofifo

Once RAM1 has stored the packet header data, control now moves to the striptofifo state which has the task of storing the payload data into FIFO1.

Blankstate

Modifies the Ethernet checksum stream by zeroing out the existing checksum data. This is required when calculating the received Ethernet checksum - the checksum already embedded in the packet must be cleared during this calculation.

Finalise

A single clock cycle holding state.

CheckCRC_ETH

Performs a comparison between registers *crc_output_ETH* and *crc_output_ETH_FCS*. If they are identical, this means the calculated Ethernet checksum is identical to the embedded Ethernet checksum, and so the packet was received without any corruption. At this point there are three things the FSM could do: (1) If the received packet was ICMP, the fsm_packgen FSM needs to be activated to produce a response (standard ping operation); (2) If the received packet was UDP and *toggleUDPbounce* was not enabled, fsm_read would move to the reboot state and begin anew, waiting for further packets to receive; (3) If the received packet was UDP and *toggleUDPbounce* was active, fsm_packgen would be activated to produce a mirrored version of the packet (mainly for debugging). If however, the Ethernet checksums were different, this means the packet is corrupted somehow and cannot be trusted to hold correct data. In this case the FSM reboots without triggering fsm_packgen, regardless of protocol.

Reboot

This state calls a special register which orders the FSM to execute a reset. Control restarts at Preinit1.

Holduntilfinished

If the packet failed any of the preamble, SDF or valid protocol tests, control is moved here. This state simply waits until the packet has run through the system, but does not capture any of its data. Once the packet has cleared and RXDV goes low, the system is rebooted.

C. FSM_PACKGEN

This FSM is responsible for generating packets to be sent back to the computer. Depending on the nature of the packets to be sent, it will have to copy the contents of RAM1 and FIFO1 from fsm_read to its internal components buffers, RAM2 and FIFO2.

When the interface begins execution, the fsm_packgen code will initialize and hold in a waiting state. The FSM waits for an activation signal from either fsm_read or toplevel (the interface to the external FPGA code) - if it comes from the former, execution requires a mirrored packet to be sent; if it comes from the latter, FIFO2 in packgen will already be filled with the required payload so no extra copying needs to be performed. There are two paths of execution the system can take for that matter:

1) ICMP and UDP (mirror mode) - copy ram1 to ram2 and fifo1 to fifo2
2) UDP (regular mode) and UDP-Lite - copy ram1 to ram2 but leave fifo2 untouched, since it will have the necessary payload installed from a previous access

Once these tasks are completed the FSM progresses to completing the calculation of the CRCs for each packet and injecting them in the relevant fields of the packet. The overall task for the FSM is then completed, and while fairly simple it's an important part of the conjoined FSMs used to transmit data back to a computer. The FSM is then reset and waits for another trigger to begin operation.

D. FSM_SEND

This FSM is responsible for the actual transmission of packets from the system to the computer which were prepared by fsm_packgen. Both of these FSMs work together - fsm_send is always executed after fsm_packgen, and fsm_send cannot be triggered from any other source other than fsm_packgen. The specific fields are sometimes different between the various protocols in use, but the header lengths are identical to all packets which makes the FSM simpler to implement. Once the payload has been sent, the Ethernet checksum is sent to finalize the packet, the FSM is reset and waits for another trigger by fsm_packgen.

III. PIPELINING

Pipelining is a design technique in which code and/or data used by a system can be restructured in such as way as to reduce the amount of time necessary to process it. In the context of FPGAs, pipelining involves running the various Finite State Machines in *parallel*, stacked one on top of each other, instead of a conventional *serial* execution. Effective use of pipelining can be extremely beneficial to applications such as cryptography, data processing [8], [9] or computer architecture [10], where a variation in encryption/decryption technique can cause a massive change in the time needed to perform the operation.

To better describe the operation and effectiveness of a pipelined architecture, observe the follow diagram which shows the flow of the system for a single ICMP packet received by the board (Figure 2).

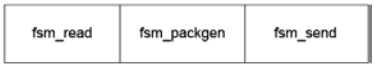

Fig. 2. The time-line for processing one ICMP packet

The block represents the passage of time for each received packet (from the left flowing to the right), and each segment of a block represents the passage of time for that particular FSM. Every packet has to be parsed by fsm_read, a return formulated by fsm_packgen and finally transmitted fsm_send. The time taken for each FSM is similar, since the received and returned packets have identical lengths and data. Let's assume that instead of waiting a predetermined amount of time before sending ping packets, the pings are configured to send a new ping as soon as it is considered safe to do so - the aim is to send packets as quickly as possible without there being any loss or congestion in the system. This rate will be shown to vary between serial and parallel (pipelined) execution.

Once the first packet is received and returned to the computer, the next one is sent immediately, and once that is returned to the computer the final packet can be sent. Serial execution is simple to understand and apply, but the rate of packet transmission is very slow. Only one FSM is in operation at any given time, but this need not be the case, as they are capable of working independently on another packet separate from another.

Three sequential ICMP packets under parallel (pipelined) execution would run as in Figure 3. In this scenario, the computer is sending ping packets one after the other, as soon as it can, without waiting for a responding return pong from the board. This is achievable with a pipelined architecture because once the first packet is read by the board, execution moves to fsm_packgen AND fsm_read - packgen works on creating a return for the first packet while fsm_read busies itself with reading the second packet. What is most interesting is that when the first packet has reached the fsm_send part for transmission, the second packet is working with fsm_packgen and the system will have begun sending the third packet for reading by fsm_read. At this point all three FSMs are working separate from each other on separate packets. Over time the return packets will be sent in sequence until the third and final packet is sent.

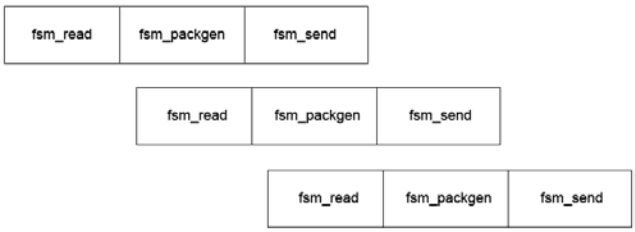

Fig. 3. The time-line of three ICMP packets processed in parallel (pipelined)

With serial execution, assuming roughly the same length of time per FSM, it takes 9 units of time to process three ping packets. With pipelined execution, it takes only 5 units of time. So, by ensuring the design is pipelined instead of serial, we are able to cut the total time (ping and pong) of three ICMP packets by nearly half. With this implementation, pipelining does not increase the speed at which the FSMs operate, but rather organizes the data such that several FSMs can operate on different data chunks at the same time.

IV. RAM SWITCHING BUFFERS

An issue can arise when dealing with pipelined data - data corruption by subsequent packets. If the computer sends a packet to the board, and while the board is in one of the latter FSMs the computer sends another packet (as is allowed under a pipelined architecture), there is the potential for packet data from the first packet to be overwritten by that of the second packet.

For example, let's say the computer sends an ICMP packet along with a UDP packet. For the first packet, the board will read the header & payload data, store them into RAM1 and FIFO1 respectively, then move onto fsm_packgen to perform a RAM1-RAM2 and FIFO1-FIFO2 copy. During the copy, the UDP packet is received by the board, and so fsm_read performs its duty and reads the incoming UDP packet. Now, the information in this packet is totally different to the prior ICMP packet - the headers are different, and the payload data is most certainly different. Hence, we run into several problems:

1) If fsm_packgen is performing a RAM copy at the time the new packet is being read, the header data of the new packet might be copied into RAM2 along with that of the original packet. This would result in the transmitted packet having a combination of ICMP and UDP header fields, which would be seen as garbage by the receiving computer and discarded.
2) If fsm_packgen is performing a FIFO copy at the time the new packet is being read, and the header data of the new packet specifies a payload length that's different to the previous packet (as a result of the RAM copying scenario in point 1), the payload data might not be copied entirely or too much data might try to be copied regardless of whether there's anything in the source FIFO or not. This would result in the transmitted packet having not only an incorrect amount of payload data as well as a corrupted header.
3) If we're very unlucky and fsm_packgen encompasses both RAM and FIFO copy stages, the entire resulting packet is corrupted and worse, the system is considered unstable since the RAM will have corrupted data which can't be relied up for future packets created from scratch, and the FIFO will almost certainly be useless since the state of the data stored in it will not be known. At this point all future packets are likely to be transmitted corrupted, and so the only solution is to reset the board.

To combat the issue of data corruption by packets overloading each other, there are three possible solutions:

1) Take extra control over the way the computer sends out packets, so that there sufficient time between subsequent packets to allow for smooth operation without packet crowding. This solution however is totally undesirable - it would be the slowest, since no pipelining would be allowed. It would also involve extra work in configuring the computer to allow for this brief pause between packets, which may not be easy to accomplish particularly if kernel hacking was required.
2) Block fsm_read from accepting new packets if it is not safe to do so. For example, fsm_read would not begin reading new packets if fsm_packgen was running, because the potential for packgen to corrupt the transmitted packet would exist if the data contained in fsm_read was being modified at the same time as the copying. This solution, again, is totally undesirable, since it would mean the board would sometimes miss packets being sent by the computer.
3) Double the size of the RAMs, so that each RAM can store the header data of *two* packets instead of just one, and switch the addressing between them as necessary. It isn't possible to do the same with FIFOs since they do not have an accessible addressing mechanism, but this is not an issue if the headers remain correct, since the payload data can be added to the end of the FIFO even during a copy process without damage, as each packet's

length is now stored correctly. This is the option which has been implemented, as it allows for the pipelined architecture without packet corruption.

The diagram in Figure 6 shows the path of three sequential packets, all of them in UDP mirrored mode so that they are sent back immediately. Once one packet has been sent, the next one is fired off straight away. Each row in the diagram represents the time-line of a packet, the first row being the first packet and so on. Each row is split into three sections - the first represents the activity for fsm_read, the second for fsm_packgen and the third for fsm_send. The purpose of the diagram is to show the operation of the RAMs and FIFOs and the RAM switching buffers, and what components are running during the relevant stages of each packet. The diagram shows why the system is not vulnerable to packet corruption:

- There never occurs a situation where the same storage area in a RAM is used more than once at any given time. There are times where the same RAM is both read and written at the same time, but due to the switching buffer, the read and write are to opposite halves of the RAM, and so do not corrupt each other.
- Although there are many cases where the FIFOs are being read and written at the same time, this is not a problem. The nature of the FIFO (**F**irst **I**n, **F**irst **O**ut) means that whatever data is being written, will only be read once the data in "front" of it has been read. In other words, simply reading at the same time as writing won't damage the integrity of the data, since the IO operations occur at *opposite sides* of the FIFO's internal storage.

V. UDP-Lite

UDP-Lite is a new implementation of the UDP protocol which simplifies the hardware implementation. This is due to the relationship between the checksum calculations required by the protocols and what happens to a FIFO when it is read.

Every packet that's sent needs to have correct checksums in their checksum fields. If the checksums are incorrect or missing, the recipient will generally consider the packet to be corrupted and discard it. Some protocols may use checksum fields which don't exist in others, but regardless of the protocol in use there are always at least three checksums that need to be calculated - two of those are always the Ethernet checksum and the IP checksum. The Ethernet checksum represents the contents of the entire packet (excluding the preamble), while the IP checksum covers only the IP header of a packet, but not the payload. For UDP packets there is a third checksum field, the UDP checksum, which covers the UDP header plus the payload. For ICMP packet the third checksum field is the ICMP checksum, with similar specifications. Correct operation of the IP core requires all available checksums for transmitted packets to be perfectly correct, otherwise the packet won't be picked up by the computer and data will be lost.

For ICMP and UDP (mirrored) checksums, the IP and ICMP/UDP checksums for the transmitted packet are easy to calculate since they are identical to those from the received packets (the order of the headers are often switched around but the data contents are identical, since they're just copies of the same payloads). Ethernet checksums are calculated during the sending of a packet, working the same regardless of protocol or how packgen was triggered (either externally or internally), and so do not pose any problems. However, when sending a UDP packet on its own without being triggered from fsm_read, all necessary checksums have to be calculated because the payload data won't match the previously-copied checksums.

To calculate the UDP checksum, the UDP payload data needs to be read, however whenever a FIFO is read, the last element read is permanently removed from the FIFO itself, so unlike RAM it's not possible to simply parse the payload for checksum information without removing the entire payload itself. To fix this problem, there are simple and not-so-simple solutions. One would be to replace the FIFO with RAM, which would allow reading without data destruction, but this would be slower and potentially cause problems with achieving the speed requirements of the system. Another solution, the one which was decided upon, is to use a new protocol called UDP-Lite.

UDP-Lite is very similar to regular UDP but with one difference - the UDP checksum's "coverage" (the amount of the UDP header and data that the checksum has to correctly match) can be varied. In the UDP header, the length field is replaced by a "coverage" field. This means it's possible to set a coverage such that the UDP checksum only covers the header component and NOT the payload. Hence only the RAM needs to be scanned and the FIFO can remain untouched. In any case, the Ethernet checksum guarantees that the whole packet is correct, so that there is no loss of reliability in dropping the extra checksum of the UDP data. Further, UDP-Lite allows jumbo packets up to 64KB. This protocol has gained support in most modern operating systems, so there are no significant disadvantage to using it compared with standard UDP.

VI. Reliability and Error Correction

Reliability, in the context of computer network protocols, is a measure of how capable a protocol is in ensuring data is delivered correctly to the intended recipient(s).[1] The system utilizes UDP and UDP-Lite for most of its communication, but these are considered to be unreliable protocols because they do not guarantee that a packet will be correctly sent and received. They do allow for checksumming of the header and payload, but do not support any additional features for ensuring a packet goes where it's suppose to. This is unlike TCP which uses flags and additional packets to confirm receipt of packets. For this reason, there existed a need to create effective reliability support for the system, such that if packets were missed or sent/received out of order, the system can recover and deal with the situation in a graceful manner.

Tags are a simple but effective way of providing the ability to detect missing packets from a data stream, as well as determine if packets were sent out of order in the steam. A tag is a small piece of data inserted into the beginning (or end) of the payload of each packet - the tag consists of a number which increments in subsequent packets. Tags are effectively transparent and considered part of the regular payload data, which provides the flexibility to use tags or not without requiring extra logic to process them (Figure 4).

One example of where the tag system is implemented is in ATA over Ethernet, a network protocol developed by the Brantley Coile Company, designed for accessing ATA storage devices over Ethernet networks. AoE does not rely on network layers above Ethernet, such as IP, UDP, TCP, etc, but it does implement the tag system on packets and provides a look-up table on both sides of the system to determine where all the packets are located, and can effect a resolution if a particular packet is missing/corrupted from a read request, for example. This means the tag system for reliability and error correction has already proved itself in a commercial setting, [2].

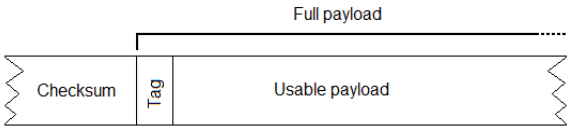

Fig. 4. Fragment of a packet with tag added to beginning of payload

VII. Conclusions

The IP core was designed in VHDL and after testing and synthesis, the final results show the interface code uses approximately 1000 slices, running at just over 125 MHz on a Xilinx Spartan 3 FPGA (XC3S5000-5). The design is sufficiently small to allow for other, more size able programs to run on the same FPGA with the remaining space, and the system is capable of working at full speed with the Gigabit Ethernet standard. The final design was fully verified on a SPARTAN3 prototyping board.

The use of the UDP_Lite protocol was found to considerably simplify the hardware design. This new protocol is mean for large volume IP transmissions, as in the case of IPTV, and allows up to 64KB jumbo packets. By avoiding the need for a full UDP checksum at the end of the packet, the storage was considerably reduced and the pipelining of the design made much simpler.

References

[1] R. Wilkov, *Analysis and Design of Reliable Computer Networks*, *IBM Thomas J. Watson Research Center, Yorktown Heights*, 1972.
[2] S. Hopkins, and B. Coile, *AoE (ATA over Ethernet)*, http://www.coraid.com/documents/AoEr10.txt, 2006.
[3] Zhan Bokai, Yu Chengye,"TCP/IP Offload Engine (TOE) for an SOC System", *Institute of Computer & Communication Engineering, National Cheng Kung University*, 2005.
[4] Cisco Systems, *History of Ethernet*, 2006
[5] *DARPA Internet Program, RFC 793 - Transmission Control Protocol (Version 4)*, http://tools.ietf.org/html/rfc792, 1981.
[6] *DARPA Internet Program, RFC 792 - Internet Control Message Protocol*, http://tools.ietf.org/html/rfc792, 1981.
[7] *Network Working Group, RFC 3828 - UDP Lite*, http://tools.ietf.org/html/rfc3828, 2004.
[8] G. Knowles and P. Gardner-Stephen, "DASH, DASH-H: A software and hardware for sequence alignment", *WSEAS Transactions on Biology and Biomedicine*, 2006, pp.37-42.
[9] G. Knowles and P. Gardner-Stephen, "DASH: A New High Speed Genomic Search and Alignment Tool", *WSEAS Transactions on Biology and Medicine*, 2004, pp. 59-64.
[10] J. Van Beurden,G Roberts and G.Knowles, "An Extended CAP File Structure to Support a High-Performance Implementation of Embedded Java", *WSEAS Transactions on Computers*, 2004, pp. 128-135.
[11] Anthony Cataldo, "Alcatel preps Gigabit Ethernet core for Altera FPGAs", *EE Times*, 2001.

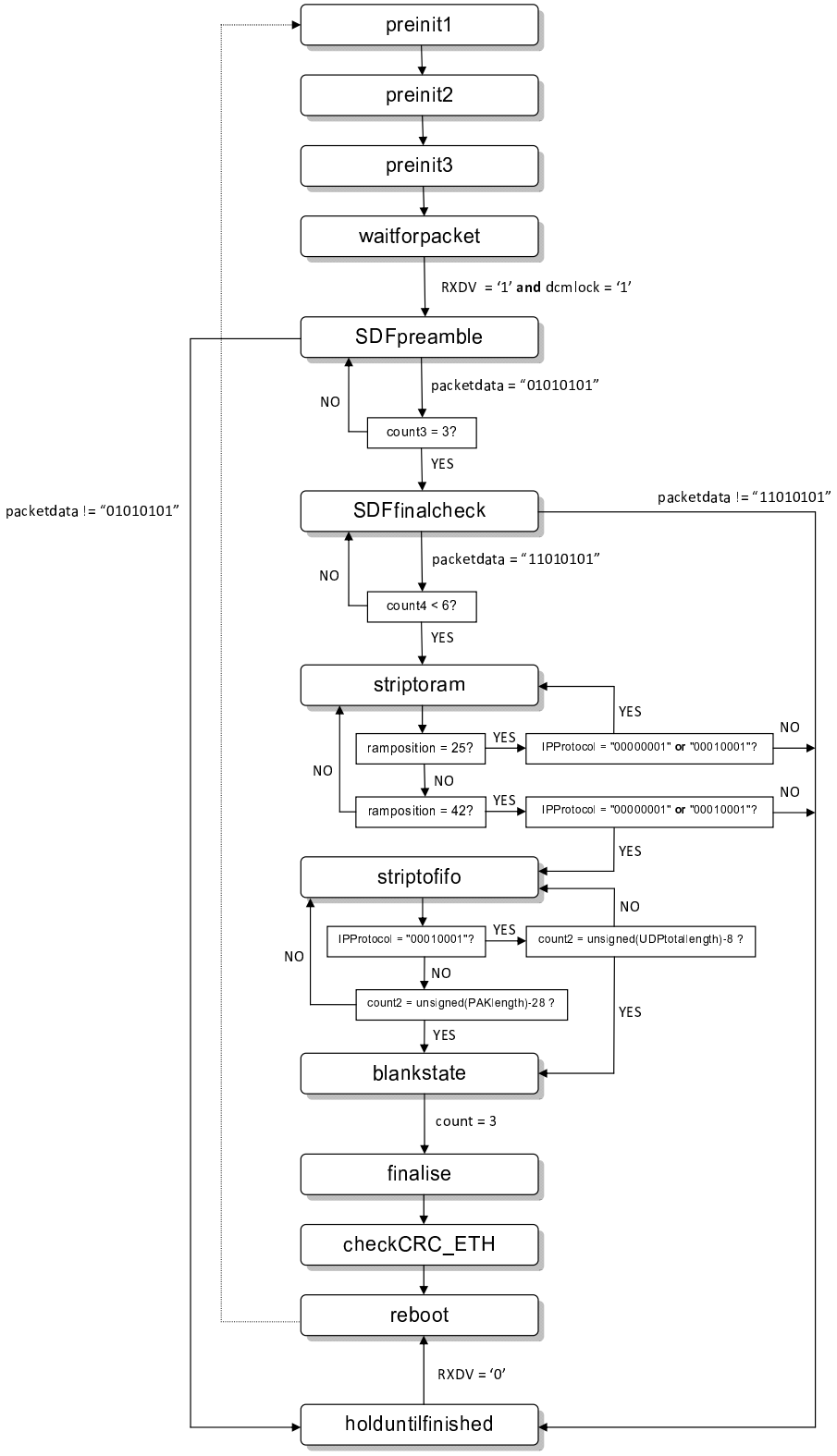

Fig. 5. FSM_READ state flowchart

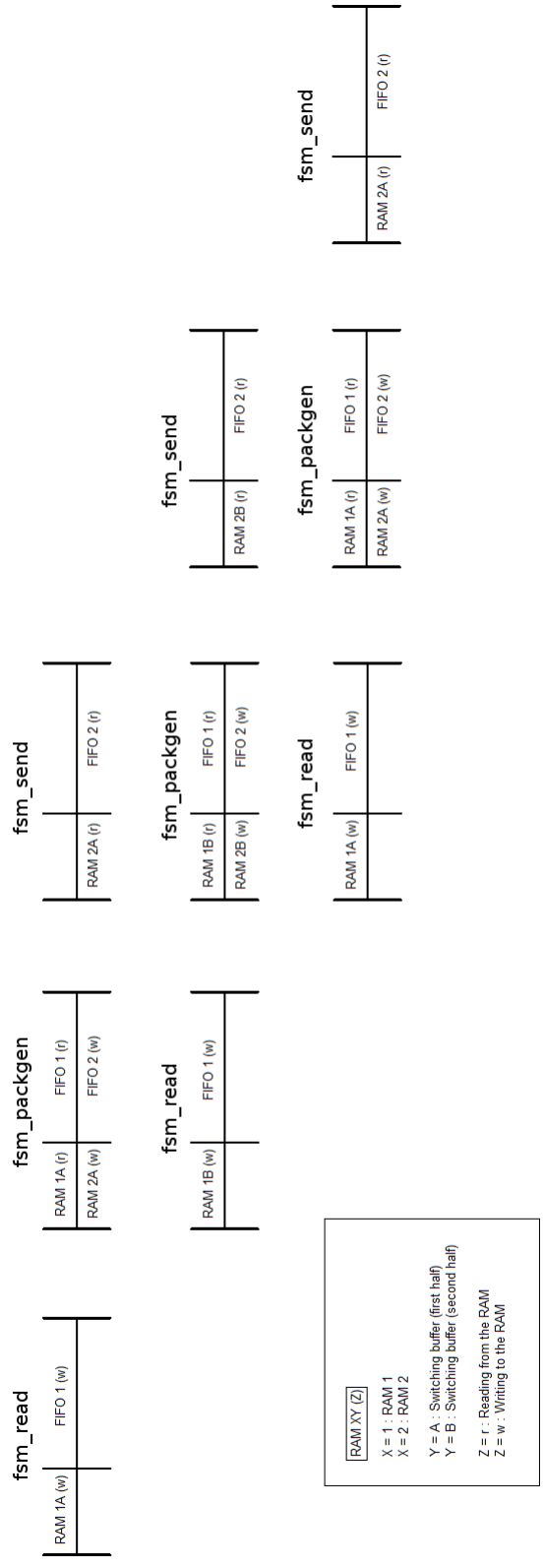

Fig. 6. Timing diagram of three UDP packets in mirrored mode