Design and development of a Secure
High Speed IoT Packet
Processor(V0)

1 Abstract

In this present day world with the boom of Internet of Things and interconnected devices as a whole, it is imperative for any device to have networking capabilities and hence the primary objective of this paper is to give the AJIT processor networking capabilities. To this end, a custom NIC will be designed and interfaced with a 10Gbps MAC which serialises the data through a high speed SGMII interface. The NIC consists of 2 threads - the receiver thread and the transmitter thread and these threads operate independently of each other. In iteration 0 , the NIC parses and stores the entire packet in the processor memory (receiver thread), and transmits that packet when instructed by the processor to do so (transmitter thread). The NIC does minimal processing on the packet in this iteration and rests the onus on the host processor.

2 MAC FIFOs

The MAC is connected to two FIFOs on the host side namely the Tx_FIFO and the Rx_FIFO. The Rx_FIFO is responsible for the buffering of data sent from the MAC so that the NIC can access the packet at a slightly leisurely pace.The Tx_FIFO buffers the packet to be transmitted by the MAC until the MAC is ready to transmit. The communication between the NIC and the FIFO occurs via a pipe protocol. The pipe here is defined to be a 37-bit wide pipe, that consists of all the data sent by the MAC on the AXI-s side of the interface, packaged into a 37-bit quantity :

tlast	tdata	tkeep
1 bit	32 bits	4 bits

3 Parser

Receives 37(32 bit data + 5 control bits) bits from RX_FIFO. Concatenates 2 chunks to create 73 bit data in which, 64 bits contain main data and remaining are control bits. Then parser write this 73 bit data in packet_pipe.

3.1 Interfaces

- Input : mac_to_nic_data (37 bit wide pipe),

tlast	tdata	tkeep
1 bit	32 bits	4 bits

- Output : packet_pipe (73 bit wide pipe)

tlast	tdata	tkeep
1 bit	64 bits	8 bits

3.2 Algorithm

```
State 0 :
                Read mac_to_nic_data pipe twice,
                    dest_mac_addr[31:0] = RX1[35:4]
                    tkeep[3:0] = RX1[3:0]
                    dest_mac_addr[47:32] = RX2[19:4]
                    src_mac_addr[15:0] = RX2[35:20]
```

```
                    tkeep[7:4] = RX2[3:0]
                    tlast = RX2[36]
                write,
                (tlast,src_mac_addr[15:0],dest_mac_addr[47:0],tkeep[7:0])
                to packet_pipe.

            goto State 1.

State 1 :       Read mac_to_nic_data pipe twice,
                    src_mac_addr[47:16] = RX1[35:4]
                    tkeep[3:0] = RX1[3:0]
                    tkeep[7:4] = RX2[3:0]
                    tlast = RX3[36]
                write,
                (tlast,RX2[35:4],src_mac_addr[47:16],tkeep[7:0])
                to packet_pipe.

            goto State 2.

State 2 :
                Read mac_to_nic_data pipe
                    if RX1[36] == 0
                        read mac_to_nic_data pipe again
                        tkeep[0:3] = RX1[0:3]
                        tkeep[7:4] = RX2[0:3]
                        tlast    = RX2[36]
                    write
                    (tlast,RX2[35:4],RX1[35:4],tkeep[7:0])
                    to packet_pipe.

                    else
                        tkeep[0:3] = RX1[0:3]
                        tkeep[7:4] = 0
                        tlast      = RX1[36]
                    write
                    (tlast,32'b0,RX1[35:4],tkeep[7:0])
                    to packet pipe.

                if tlast == 1
                    goto State 0
                else
                    goto State 2
```

4 Receiver Engine

The receive engine acts as a communication link between memory and the NIC. It receives 73-bit data from the Parser and goes through the following states to store the data in Memory :
Algorithm 1 : Receive Engine Algorithm

- Input : 73-bit data from Parser consisting of data-words, byte masks and last word identifier,
- Output : 110-bit Request and 65-bit Response (To Memory)

```
State 0 :
              repeat
               1. Find empty buffer by reading flag buffer
                    if flag == 0000
                        buf <- i
                        break
                        goto State 1
                    else
```

```
                    continue
                endif
            endrepeat

State 1 :
            Set x <- buf and y <- 0
            repeat
                1. Read from parser_pipe
                2. Extract data ,tlast ,tkeep information from the 73-bit quantity
                3. Calculate address to write to based on the x,y index

                4. if not tlast
                    Package the data into the 110-bit data format with byte mask as 0xFF
                    Increment y and continue
                5. else
                    Package the data into the 110-bit data format with byte mask set to tkeep
                    break and goto State 2
            endrepeat

State 2 :
            procedure
                control_word = {40'b0 + y + tkeep + 8-bit flag}
                1. Check if the last written word has the bad packet format
                2. if yes
                    Don't set MSB flag bit in the control word

                3. else
                    Set MSB of flag bit
                4. Send 64-bit Control word to flag buffer index x

                Go back to State 1

            endprocedure
```

5 Memory Data Storing Format

5.1 Request and Response Format

1. Request packet:
   A 110-bit format.

   Request[109:0]
        Request[109] = lock-bit
        Request[108] = read/write-bar
        Request[107:100] = byte mask
            write data is organized as 8-bytes. if bit 7 (MSB) of byte-mask
            is set, the top byte of write data is written, else not.
        Request[99:64] = address (36-bit)
        Request[63:0] = write-data.

2. Response packet:
   A 65-bit format.
        Response[64] = Error
        Response[63:0] = read-data

3. For every request packet, there is a response.

5.2 Addressing Format

The NIC will be reading and writing to 2 buffers-

- The Data Buffer
- The Flag Buffer

The buffers will be organised as a 64-bit wide, 2-Dimensional matrix with each row corresponding to a new buffer and the columns corresponding to 8-bit data chunks in memory (memory is byte addressable). To index into a buffer , 2 indices are needed x,y where, the x index selects the row/buffer and the y index selects the contents of the selected x buffer.
Since the memory has only linear addressing and the 2-D matrix is just an abstraction, there has to be a mechanism to convert from the x,y indices to the actual address in memory which will be done as follows :

For the **Data Buffer** :
Address = offset$_{buffer} + x * s_x + y * s_y$, where $s_x = ((1518 * 8)/64) * s_y \approx 1520$ and $s_y = 8$

For the **Flag Buffer** :
Address = offset$_{flag}$ + $x * s_x$, where $s_x = 8$

5.3 Flag Buffer Format

The flag buffer stores the control information for each buffer in memory i.e if there are n data buffers, there will n corresponding flag buffers.

The 64-bit control word format in the flag buffer is :

Not used	size_of_packet	byte_mask	flag
40 bits	12 bits	8 bits	4 bits

4-bit flag format :

- 0000 $\rightarrow$ Buffer empty
- 1000 $\rightarrow$ Data written to buffer but not yet processed
- 1101 $\rightarrow$ Data processed by processor and ready to transmit
- 1110 $\rightarrow$ Data processed, but packet will not be transmitted (drop packet)

6 Transmit Engine

Reads data from the memory processed by the CPU in 64 bit words and sends 37 bit data to TX_FIFO.

- Input : 65 bit read data from memory
- Output: 110-bit request and 37-bit data to RX_FIFO

Algorithm:

```
State 0 :
            1. Send a read request at ith flag buffer and check the flag bits to see if
            the CPU has processed the packet.
                if flag == 11XX
                    size = read_data[23:12]
                    byte_mask = read_data[11:4]
                    goto State 2
                else
                    goto State 1
```

```
State 1 :
            Wait for 32 clock cycles.
            goto State 0.

State 2 :
            1. Check DROP and TX bits in the flag.
                if flag[0] = 1 // TX
                    1. Start reading ith buffer until size.
                    2. Slice the 64 bit data into 2 chunks of 32 bit each to send the
                       data in 37-bit format to TX_FIFO
                    3. Until the last word, tkeep for both chunks is 0xF and tlast = 0.
                       For last word, determine if Chunk 0 or Chunk 1 is valid.
                            if byte_mask[7:4] = 0x0 //Only Chunk-0 is valid
                                send Chunk-0 with tkeep = byte_mask[3:0]
                                tlast = 1
                            else // if both chunks are valid
                                1. Send Chunk-0 with tlast = 0
                                2. Send chunk-1 with tlast = 1.
            elsif flag[1] = 1 // DROP
                discard packet, do nothing
            2. Send a write request to reset the flag bits ([3:0])
            3. Increment i and go back to State 0
```

7 Testing Methodology

For the purpose of verifying the design a simple loopback will be implemented at either the PHY end of the MAC or at the host processor end.

- PHY end : In this approach a packet to be transmitted will be sent by the host processor which will be transmitted by the transmitter thread and eventually is serialised into the txp and txn signals of the SGMII protocol. These signals will be directly connected to the rxp and rxn ports of the MAC and the same packet re-enters the NIC.
- Host processor end : A similar strategy as above will be applied here too, but the point of loopback and the point of packet entry is reversed. Here, the packet will be received by the Mac along the rxp and rxn ports and will be processed by the receiver engine and will be stored in memory. The host processor will process the packet and swap the destination and source addresses in the packet to emulate a loopback.

7.1 Components of the test bench setup

- A module that will be interfaced with the SGMII ports of the MAC and will be monitoring the signals being transmitted and capable of exciting the rxp and rxn ports to emulate the NIC receiving data.
- A memory block that will communicate with the NIC via the request and response protocol and will be storing the packet data and flags in the buffer. Additionally this module will also act as a CPU that will mimic processing being done by the CPU.