Ethernet hardware encapsulator

(xilinx 7 series fpga project)

version 1.1

Vladimir Efimov

https://www.linkedin.com/in/vladimir-efimov

(Nov.2016)

1. PROJECT DESCRIPTION

The final goal of this project is to create “Ethernet Encapsulator” system that would receive ethernet frames on its input interface and perform wire-speed encapsulation with predefined header stored in the memory prior sending frames out via output interface.

2. CURRENT VERSION OF THE PROJECT

a. Working System of Chip (SoC) has been developed and implemented on “AVNET Artix7-50T” FPGA board. The system consists of a “dataplane” where custom “Ethernet Encapsulator” IP core is placed and a Microblaze CPU based “control unit”. You can see schematics of the whole system in separate “soc_v[version_number].pdf” file that has been imported from Vivado IP integrator.

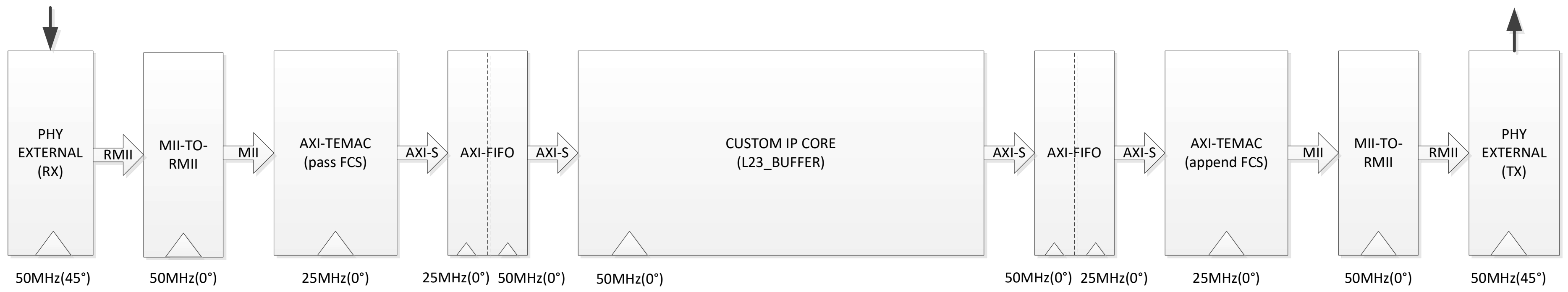
b. Management software has been written for “control unit”. That software interacts with UART on one side and on the other side it controls behaviour of the “dataplane” particularly “Ethernet Encapsulator” IP core by managing its memory content as well as programming its registers. Note that AXI memory mapped management interface for “Ethernet Encapsulator” IP core will be introduced in the following versions of the project. So in this version there is no connection between “dataplane” and “control unit”.

c. “Ethernet Encapsulator” IP core called “L23_buffer” written in Verilog is the main part of this project. In current version “L23_buffer” is just a “Store and forward” buffer. It has two AXI-stream interfaces one slave as input and one master as output. This block doesn’t perform encapsulation but it has fundamental buffering logic which will be modified in the future versions as project evolves.

d. Verilog testbench has been written for “L23_buffer” IP core.

3. DATAPLANE DESIGN

The “dataplane” depicted below is the part of the SoC design that the data traverses from input port towards output port.



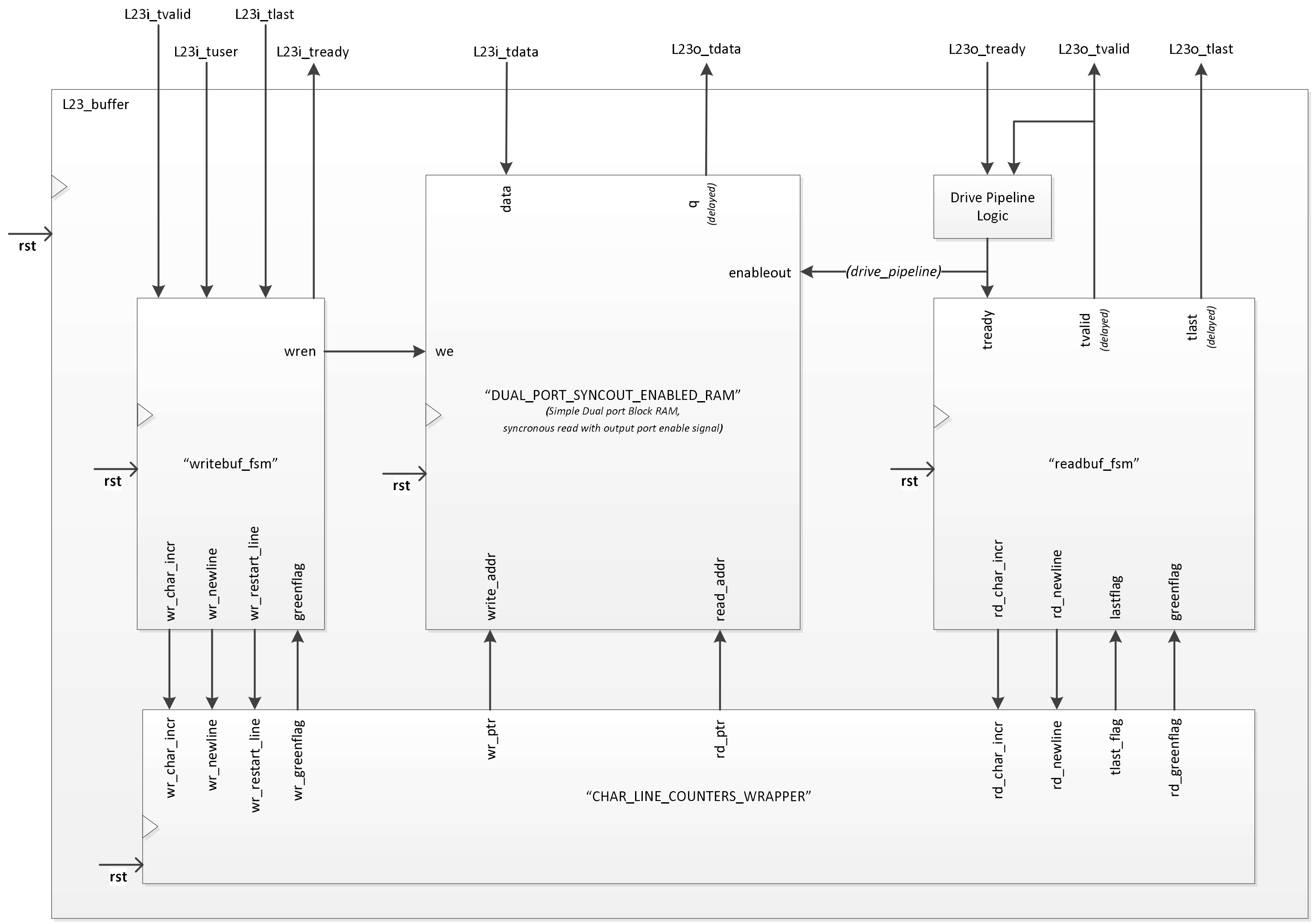
In current version of the project “dataplane” design is non-blocking because the buffer block “L23_buffer” is run at the speed of 50mhz whereas the rest of the design is run at the standard 100mbit Ethernet clock of 25mhz. Because “L23_buffer” core is faster than the rest of “dataplane” design we can safely assume that FIFO buffers will never go overrunning on receive end and go under-running on transmit end. In further versions of the project “L23_buffer” core will have capabilities to be stalled by be the “control unit” that would inevitably cause FIFO buffer on receive end overrun that in turn result in some data loss/corruption. To solve the problem additional finite-state machine will need to be placed between “AXI-TEMAC” and “AXI-FIFO” cores on the receive end. The overflow FIFO buffer condition must be signalled further down the “dataplane” by assertion of “tuser” sideband signal. In current version of the project “L23_buffer” is already designed to discard currently bufferized frame if “tuser” signal indicates that the previously received data is incomplete/corrupted.

Note: depicted above is the logical diagram of the “dataplane”. In order to simplify testing of the design two “logical” PHY external components as well as two MII-TO-RMII blocks and two TEMAC modules has been merged into one physical PHY, one physical MII-TO-RMII and one physical TEMAC (as these physical blocks are comprised of separate RX/TX circuits). Therefore the data that comes to the physical external ethernet interface (RX) after being processed by “L23_buffer” loops back and comes out of the same physical ethernet interface (TX). So practically this design can be implemented on an FPGA board with single ethernet interface.

4. “L23_buffer” IP CORE DESIGN

The high level diagram of “L23_buffer” IP core consists of the following modules: “SIMPLE DUAL PORT BLOCK RAM”, “CHAR_LINE_COUNTERS_WRAPPER”, “writebuf_fsm” and “readbuf_fsm” their functionality explained later in the document (note: “Drive pipeline logic” is not a separate logical block).

“L23_buffer” module



Notes: Drive_pipeline = ~Stall_Pipeline; Stall_Pipeline = (tvalid & ~tready);
Pipeline - one clock delayed outputs from syncronous read BRAM and “readbuf_fsm”.

4.1 BUFFER MEMORY LAYOUT

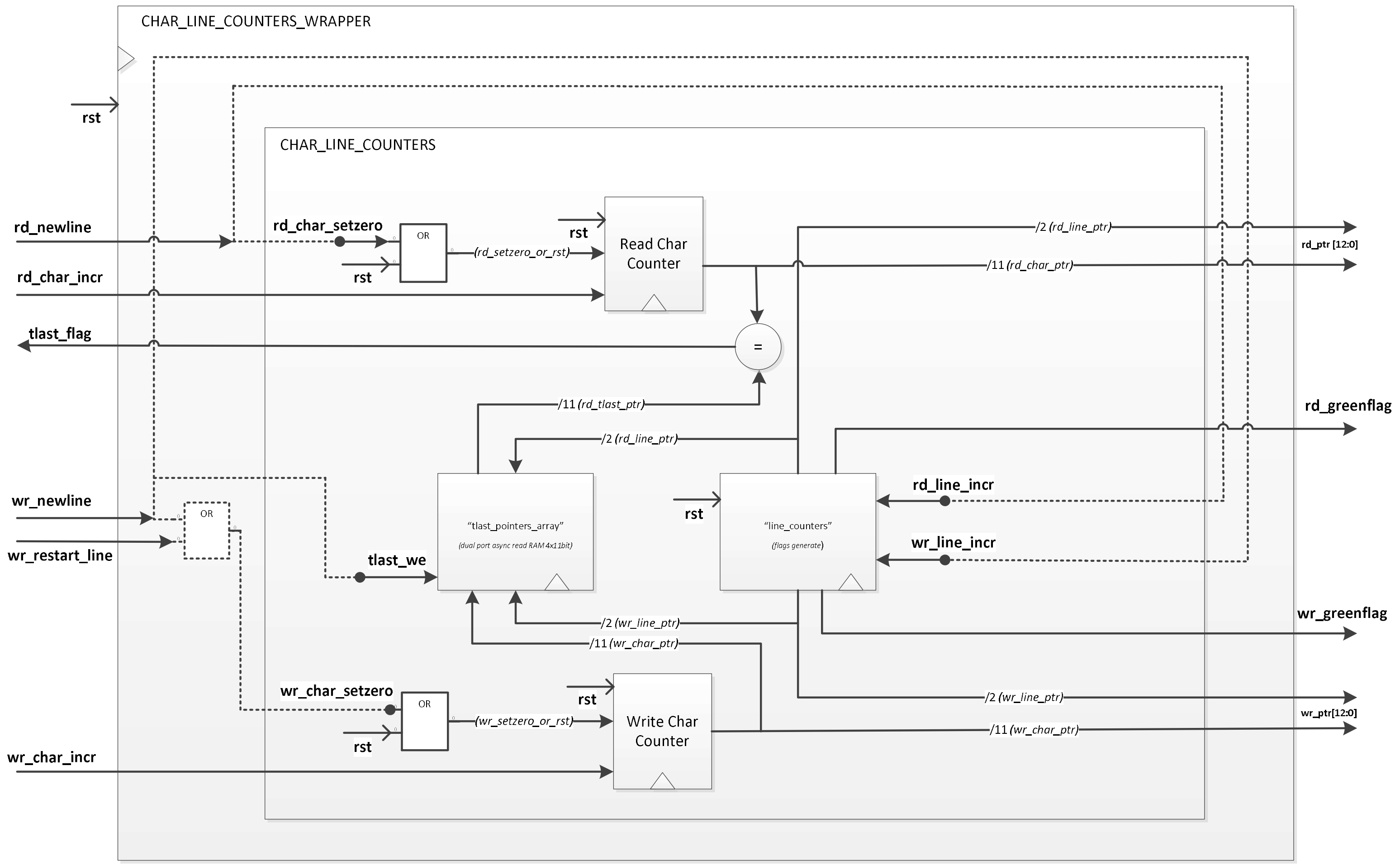
“DUAL_PORT_SYNCOUT_ENABLED_RAM” – the main memory block in the design used for temporary storage of the ethernet frames. It is synthesized as Simple Dual Port Block Ram with synchronous read and output port enable signal.

Full addresses for write and read ports are 13bit. The two most significant bits of these 13 bits are called “line address” used to address one of four lines where up to for ethernet frames can be stored. Other 11 lower bits are used to address one of 2048 bytes within the line/frame and it is called “char address” and used to address a particular byte within ethernet frame.

4.2 “CHAR_LINE_COUNTERS” AND “CHAR_LINE_COUNTERS_WRAPPER” MODULES

“CHAR_LINE_COUNTERS” module provides necessary signals for read and write FSMs as well as read and write pointer signals which are connected to address ports of dual port RAM. “CHAR_LINE_COUNTERS” module consists of smaller blocks such as “line_counters”, “tlast_pointers_array” and other logic elements described below.

“CHAR_LINE_COUNTERS” and “CHAR_LINE_COUNTERS_WRAPPER” modules



Notes:
The purpose of “CHAR_LINE_COUNTERS_WRAPPER” module is to simplify external signals of “CHAR_LINE_COUNTERS” module to make them compatible with external logic, namely:
1. make signal "rd_newline" that would drive "rd_char_setzero", "rd_line_incr" signals of “CHAR_LINE_COUNTERS” module;
2. make signal "wr_newline" that would drive "wr_char_setzero", "wr_line_incr", "tlast_we" signals of “CHAR_LINE_COUNTERS” module;
3. make signal "wr_restart_line" that would drive "wr_char_setzero" signal;

4.2.1 “LINE COUNTERS” MODULE

The “line_counters” module compares read and write “line addresses” and generates “rd_greenflag” and “wr_greenflag” signals. Asserted “rd_greenflag” means that there is data ready to be read from the buffer aka (~empty) and “wr_greenflag” means there is a place available in the buffer to write the data in aka (~full). The principles used for flag generation in this design resemble those [1] used for flags generation in many FIFO buffers:

a. The buffer is empty when the read line and write line pointers are equal. This condition happens when both line pointers are in initial zero state, or when the read line pointer catches up to the write line pointer when the last line has been read.
b. The buffer is full when the line pointers are again equal, that is, when the write line pointer has “wrapped around” and caught up to the read line pointer.
c. In order to distinguish between full and empty states an extra bit to each line pointer has been added. When the write line pointer increments past the final line address, the write line pointer will increment that extra (most significant) bit while setting the rest of the bits back to zero. The same will happen with the read line pointer. If the most significant bits of the two line pointers are different that would mean that the write line pointer has wrapped one more time past the read line pointer. If the most significant bits of the two line pointers are the same, it means that both line pointers have wrapped the same number of times.

4.2.2 “TLAST_POINTERS_ARRAY” MODULE

TLAST array is a memory (distributed RAM; asynchronous read) that stores addresses of last char (byte) for each line.

TLAST array is a RAM of 4x11bits; where address is the number of the line and the data is the address of the last char for that particular line.

4.2.3 TLAST COMPARE LOGIC

It generates TLAST_FLAG for read FSM that compares address of the last char for the current line with current char pointer.

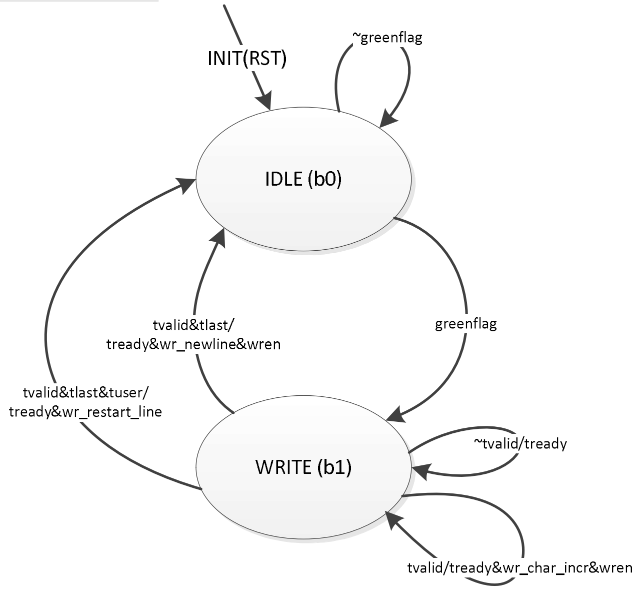
4.3 “WRITEBUF_FSM” AND “READBUF_FSM” MODULES

Read and write FSMs (implemented in “writebuf_fsm” and “readbuf_fsm” modules) are driving all the logic in the IP core. The following signals of “writebuf_fsm” are presented externally to the IP core and constitute AXI-Stream-master interface: “tvalid, tlast, tuser, tready”. The following signals of “readbuf_fsm” are presented externally to the IP core and constitute AXI-Stream-slave interface: “tready, tvalid, tlast”.

Write and Read Finite state machines

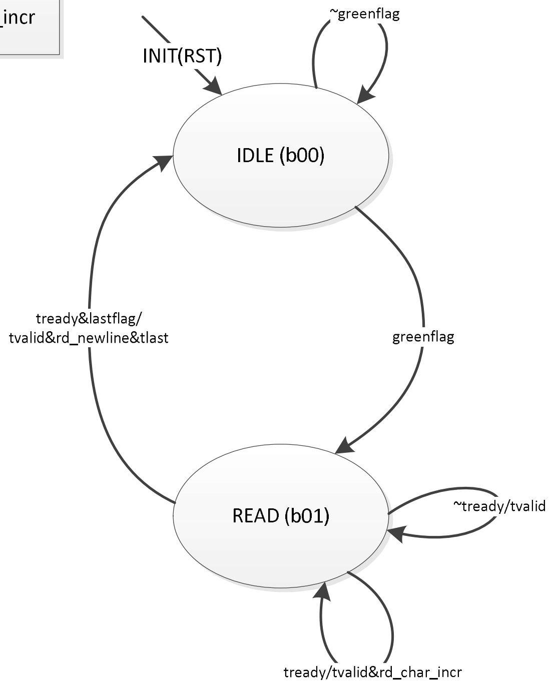
WRITE TO BUFFER FSM

INPUTs	OUTPUTs
greenflag tvalid tlast tuser	tready wren wr_newline wr_char_incr wr_restart_line



READ FROM BUFFER FSM

INPUTs	OUTPUTs
greenflag lastflag tready	tvalid tlast rd_newline rd_char_incr



As was mentioned earlier “tuser” signal has not been generated anywhere in this version of the design but write FSM has the logic implemented already to skip a corrupted/incompleted frame. It is important to note that according to TEMAC IP core documentation [2] “tuser” flag can only be asserted at the end of the frame i.e. with “tlast” flag therefore write FSM has been designed following this logic.

The design of read FSM will be going through extensive modification as project evolves especially when encapsulation functionality will be introduced in the following versions unlike write FSM which design would (probably) remain unchanged.

4.4 DRIVE PIPELINE LOGIC

The significant complexity comes from the fact that Block RAM memory on FPGA has synchronous (registered) “tdata” output that forces a designer to build a one stage pipeline for all remaining AXI-stream sideband signals (“tvalid”, “tlast” etc).

These guidelines should apply to any length of the pipeline although in our case we just need a single-stage pipeline as Block RAM memory has 1-clock delay registered output.

Let’s outline the conditions that have to be met for N-stage pipeline to operate correctly:

a. All the read FSM sideband output signals such as "tvalid" and "tlast" needs to be delayed (pipelined) to align with registered data output “tdata” of the BRAM.

b. When pipeline is not driven all sideband signals as well as data must preserve its values in all pipeline stages.

BRAM has "enableoutput" signal for that purpose.

For "tvalid" and "tuser" signals the pipeline is "N-stage shift register with enable".

c. At the output of read FSM which is the input of the pipeline the following transitions are allowed:

CurrentStateOfTheData	**NewStateOfTheData**	**PipeLineDriven**	**PipeLineStall**
Valid	Valid	allowed	notallowed
Valid	NonValid	allowed	notallowed
NonValid	Valid	allowed	allowed
NonValid	NonValid	allowed	allowed

When pipeleine is driven, any changes of output signals of read FSM is permitted because any data either “tvalid” or “~tvalid” will be “sucked into” the pipeline. When the pipeline is stall and some “tvalid” data is present at read FSM output, then this data must not change as it will not be “sucked into” pipeline and therefore will be lost.

Therefore “drive_pipeline” signal becomes input “tready” signal for read FSM as well.

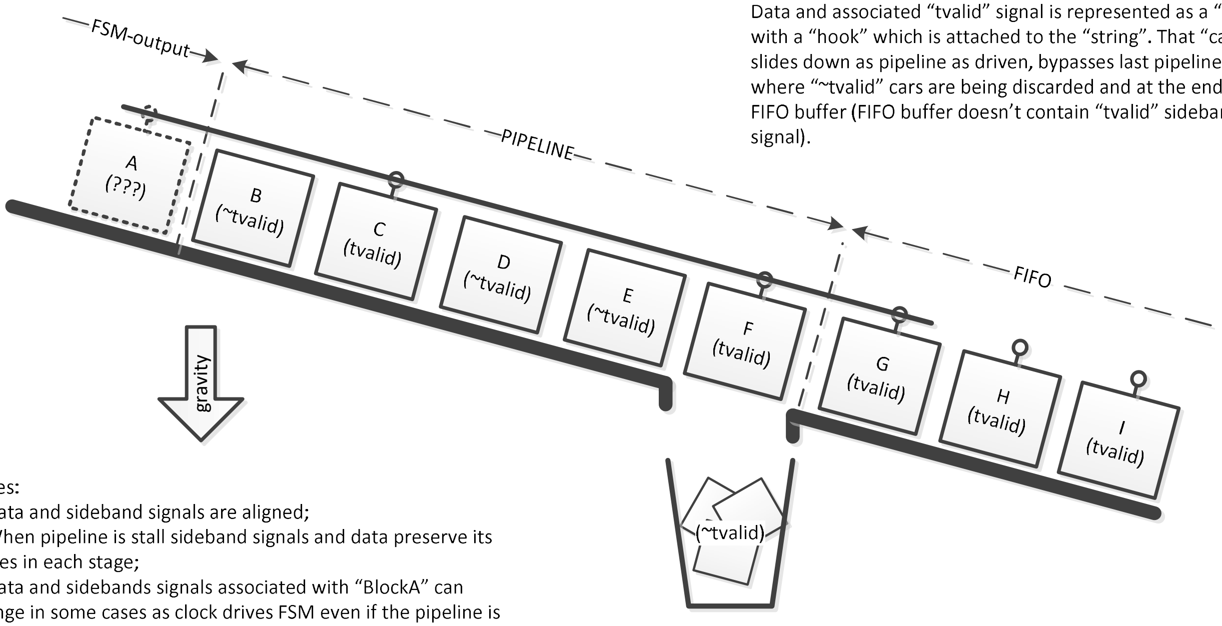
d. Pipeline is driven by sampling the signal at the end of the pipeline:

“Stall = (tvalid & ~tready)” => “Drive = ~(tvalid & ~tready)”

If the pipeline is implemented with the respect to forementioned principles then read FSM can be designed without any concerns of the existence of the pipeline – the same way as if the output of RAM was asyncronous (i.e not pipelined). Although the design of read FSM itself is “pipeline agnostic” the “readbuf_fsm” module (that contains that machine) must still provide delayed sideband signals aligned with registered data output “tdata” of the BRAM.

In addition to the text above here is an imaginary so called “sliding cable-cars” model which is in my opinion, functionally, is the closest analogy to how the pipeline operates. There is read FSM as a source of data, FIFO buffer as a sink and N-stages pipeline in the middle. Notes (1-4) on the slide correspond to the bullets (a-d) in this text above.

“Sliding cable cars” AXI-stream (master) pipeline model



Data and associated “tvalid” signal is represented as a “car” with a “hook” which is attached to the “string”. That “car” slides down as pipeline as driven, bypasses last pipeline stage where “~tvalid” cars are being discarded and at the end enters FIFO buffer (FIFO buffer doesn’t contain “tvalid” sideband signal).

Notes:
1. Data and sideband signals are aligned;
2. When pipeline is stall sideband signals and data preserve its values in each stage;
3. Data and sidebands signals associated with “BlockA” can change in some cases as clock drives FSM even if the pipeline is stall. Only case when they must not change is when FSM produces “tvalid” data that must be “sucked/slide into” the pipeline. Therefore the same signal is used to drive the pipeline and as “tready” signal for FSM that instructs FSM that current data has been accepted and asks for new data to be produced.
4. Pipeline is driven either by a “vacant” space in FIFO buffer indicated by “tready” signal OR “~tvalid” status of the data at the end of the pipeline (in place of BlockF).

5. “L23_buffer” IP CORE VERIFICATION

The “SrcArray” contains input data which is supplied to the input of the “L23_buffer” core. Input data is driven by “tvalid” signal randomly generated by the testbench.

```
//SrcArray bits:[9]-tuser, [8]-tlast, [7:0]-tdata;
reg [9:0] SrcArray [0:30] =

//1st packet will be accepted
(10'h011,10'h012,10'h013,10'h014,10'h015,10'h016,10'h117,

//2nd packet will be accepted
10'h021,10'h022,10'h023,10'h024,10'h025,10'h026,10'h027,10'h128,

//3rd packet will be discarded because the last byte of the packet has active “tuser” flag – 10’h337
10'h031,10'h032,10'h033,10'h034,10'h035,10'h036,10'h337,

//4th packet will be accepted
10'h041,10'h042,10'h043,10'h044,10'h045,10'h046,10'h047,10'h048,10'h149);
```

Testbench output:

```
# run 5000ns
@90 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:11; L23i_tlast:0; L23i_tuser:0;
@110 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:12; L23i_tlast:0; L23i_tuser:0;
@130 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:13; L23i_tlast:0; L23i_tuser:0;
@150 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:14; L23i_tlast:0; L23i_tuser:0;
@190 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:15; L23i_tlast:0; L23i_tuser:0;
@210 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:16; L23i_tlast:0; L23i_tuser:0;
@230 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:17; L23i_tlast:1; L23i_tuser:0;
@310 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:11; L23o_tlast:0;
@330 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:12; L23o_tlast:0;
@350 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:21; L23i_tlast:0; L23i_tuser:0;
@350 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:13; L23o_tlast:0;
@370 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:22; L23i_tlast:0; L23i_tuser:0;
@410 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:14; L23o_tlast:0;
@430 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:15; L23o_tlast:0;
@450 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:23; L23i_tlast:0; L23i_tuser:0;
@450 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:16; L23o_tlast:0;
@470 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:24; L23i_tlast:0; L23i_tuser:0;
@470 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:17; L23o_tlast:1;
@510 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:25; L23i_tlast:0; L23i_tuser:0;
@530 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:26; L23i_tlast:0; L23i_tuser:0;
@590 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:27; L23i_tlast:0; L23i_tuser:0;
@630 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:28; L23i_tlast:1; L23i_tuser:0;
@690 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:31; L23i_tlast:0; L23i_tuser:0;
@690 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:21; L23o_tlast:0;
@710 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:32; L23i_tlast:0; L23i_tuser:0;
@710 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:22; L23o_tlast:0;
@730 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:33; L23i_tlast:0; L23i_tuser:0;
@750 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:23; L23o_tlast:0;
@770 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:34; L23i_tlast:0; L23i_tuser:0;
@770 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:24; L23o_tlast:0;
@810 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:35; L23i_tlast:0; L23i_tuser:0;
@810 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:25; L23o_tlast:0;
@830 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:26; L23o_tlast:0;
@850 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:36; L23i_tlast:0; L23i_tuser:0;
@870 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:37; L23i_tlast:1; L23i_tuser:1;
@870 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:27; L23o_tlast:0;
@910 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:41; L23i_tlast:0; L23i_tuser:0;
@910 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:28; L23o_tlast:1;
@970 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:42; L23i_tlast:0; L23i_tuser:0;
@1010 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:43; L23i_tlast:0; L23i_tuser:0;
@1030 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:44; L23i_tlast:0; L23i_tuser:0;
@1050 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:45; L23i_tlast:0; L23i_tuser:0;
@1070 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:46; L23i_tlast:0; L23i_tuser:0;
@1090 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:47; L23i_tlast:0; L23i_tuser:0;
@1170 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:48; L23i_tlast:0; L23i_tuser:0;
@1190 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:49; L23i_tlast:1; L23i_tuser:0;
@1230 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1270 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1270 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:41; L23o_tlast:0;
@1290 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:42; L23o_tlast:0;
@1310 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1310 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:43; L23o_tlast:0;
@1330 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1370 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1410 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1410 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:44; L23o_tlast:0;
@1450 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1490 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1510 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:45; L23o_tlast:0;
@1530 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:46; L23o_tlast:0;
@1550 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1550 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:47; L23o_tlast:0;
@1570 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1570 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:48; L23o_tlast:0;
@1610 OUTPUT: L23o_tvalid:1; L23o_tready:1; L23o_tdata:49; L23o_tlast:1;
@1650 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1710 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1750 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1830 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1870 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1890 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@1950 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@2010 INPUT: L23i_tvalid:1; L23i_tready:1; L23i_tdata:xx; L23i_tlast:x; L23i_tuser:x;
@2030 Finished posedge process loop
@2040 Finished negedge process loop
```

We can only see packets 1,2 and 4 at the output of the “L23_buffer” because packet 3 has been discarded. Output data is “throttled” by “tready” signal randomly generated by the testbench.

REFERENCES

[1] Clifford E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design”, SNUG San Jose 2002, Rev 1.2, Section 2.2.

[2] Tri-Mode Ethernet MAC v9.0, LogiCORE IP Product Guide, Vivado Design Suite, PG051 October 5, 201, p89 “Frame Reception with Error”.