Modeling Algorithms

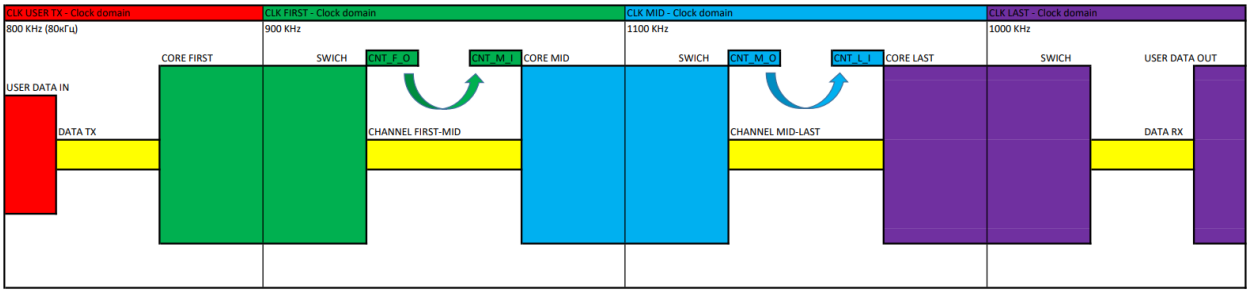
The SSI network is a synchronous network; accordingly, it can be quite accurately simulated by a set of synchronous FIFOs having different clock frequencies for reading and writing. To illustrate the process of inserting and deleting service symbols and the functionality of the algorithm for compensating for the slippage effect, in a model synchronous network I will set the inaccuracy of the clock generator frequency to 10% of the nominal 1 MHz

Algorithm for counteracting the slippage effect in synchronous systems.

1. The slippage effect, due to the difference in clock generator frequencies, leads to the insertion of unnecessary or loss of transmitted symbols.
2. Consider a data transmission channel, where every clock cycle data is written and read in a variety of intermediate FIFOs. Each FIFO represents a switch core, a data link, or a data transfer FIFO between different clock domains.
3. All switches are clocked by very close (in frequency), but not the same clock signals. The data transmission channel is plesiochronous.
4. To eliminate data loss due to overflow, it is necessary that data always arrive at the FIFO input (for all intermediate FIFOs too) at a lower speed than they will then be read. In this case, the FIFO will either contain one data element or be empty.
5. Then the question arises: how to achieve this in a synchronous system, where all changes occur at the edge of the clock pulse.
6. I propose to transmit useful (user) data with a frequency (speed) slightly less than that determined by the clock frequency of the switches. The useful transmission frequency (speed) must be less than the frequency (speed) set by the clock generator of any of the intermediate generators by twice the inaccuracy of the clock generator. For modern clock generators, this value does not exceed 200 ppm in the worst case; accordingly, the data speed should be less than the channel speed by the amount: Generator frequency multiplied by 2 * 200E-6.
7. How to do this if the device is rigidly clocked from a local generator and there are no other clock frequencies? To obtain the effect of reducing the speed, it will be enough to add “no data” service symbols to the transmitted data, thus the transmission speed of useful data will drop in proportion to the number of added service symbols.
8. Then the question arises: When and in what quantity should I add them? I propose adding “no data” characters at the time of reading from an empty FIFO, for all intermediate FIFOs except the very first. For the first FIFO, it is necessary to either control the frequency with which data is written to it or always write, but add a no data symbol every 2500 records (for a double error of 200 ppm).
9. A small number of “no data” service characters appear in the transmitted data. At the moment of receiving this symbol from the channel, it is necessary to disable writing to the FIFO (discard the received service symbol). Thus, this character disappears from the transmitted sequence, only user data remains in the FIFO.
10. This mechanism for creating and deleting service symbols makes it possible to guarantee that the average write speed will always be less than the read speed (taking into account the added service symbols “no data”) for all intermediate FIFOs and this also guarantees the impossibility of buffer overflow, and therefore data loss for this reason are excluded.
11. A useful effect of this algorithm is the small and fixed size of intermediate buffers, which means a stable switching time on average equal to half the time for transmitting one character.

Simulation of a synchronous data transmission channel on an FPGA.

An extremely simplified model of a section of the SDN network (three intermediate switches):



- The theoretical frequency of the master oscillators is chosen to be 1 MHz (1E6 symbols per second).
- Error setting the clock frequency 10% (1MHz +- 100kHz).
- Maximum data transfer rate (without overflow losses) 800k characters per second.
- The option in parentheses is where data takes up 10% of the maximum bandwidth.
- CNT_x_x are counters of transmitted and received characters. Please note that each pair of counters is located in the same clock domain, which means there can be no problems with synchronization. Now such a counter exists in every high-speed network, but it only counts received bits (for the correct formation of transmitted words). The synchronization problem has long been and reliably solved.

An algorithm for dividing a single physical channel into many virtual channels.

1. Modern systems (optics in particular) are built on the transmission of “symbols” (bit sequences of a certain specified length). On the transmitting side, with some constant frequency (clocked by a local oscillator), symbols are fed into the conversion module from parallel to serial, and then the serial stream goes to the transmitter. On the receiving side, first the clock frequency is restored in the PLL module, then it is converted to parallel mode and written to the FIFO, which separates domains with different clock frequencies. Writing to this buffer occurs using the recovered frequency of the local clock oscillator (PLL) that transmitted the data.
2. On the transmitter and receiver there is (created for the operation of the algorithm) one counter, which is incremented every time a symbol is transmitted or received, regardless of the type of symbol (even there is no data for the symbol). And it is precisely them (on the model the counters are called CNT_x_x) that need to be synchronized so that they contain the same number at the time of sending and receiving any symbol. For example, if at the moment of transmission of a particular symbol the counter contains the number N, then at the moment of receiving this symbol the counter on the receiving side must contain exactly the number N.
3. Both counters are clocked at the same clock frequency, only for the transmitter it is a local oscillator, and for the receiver it is the PLL frequency restored from the received sequence. It turns out that in normal condition the counters will not run away over time. I repeat once again: The counters on the receiving and transmitting sides are in the same clock domain.
4. All that remains is to compensate for the discrepancy (constant displacement). To do this, I propose to periodically transmit the “synchronization” service symbol, which contains data about the counter value at the time of sending. When receiving such a symbol, it is necessary to check the equality of the value in the “synchronization” symbol and in the counter on the receiving side. In case of inequality, perform some synchronization algorithm. For example, wait for three synchronization errors in a row and, on the fourth in a row, change the counter value on the receiving side.

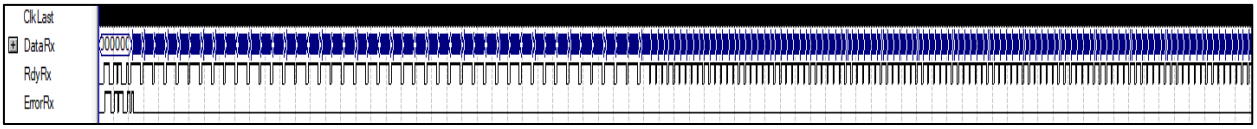
5. When to transmit such a symbol: a specific band can be allocated, just as for the “no data” symbol. You can replace the “no data” characters with “synchronization” characters on the transmitting side, and perform the reverse replacement on the receiving side. There can be a lot of options for implementing this algorithm, the main thing is to do it at least once at the specified time.
6. Possible errors:
    a. The initial offset will be eliminated as units of symbols taken (there is no useful data yet, only “no data” symbols).
    b. A counter failure on the transmitting side (the probability is extremely small and can be further suppressed through duplication of counters), after transmitting several symbols will be synchronized.
    c. A counter failure on the receiving side is approximately the same as a failure on the transmitting side.
    d. An error occurred in the transmission of the “synchronization” symbol, for example in the form of an incorrect counter value. It can be countered quite well by a simple synchronization algorithm, and even in the simplest method, the error will exist exactly until the next reception of the “synchronization” symbol.
    e. PLL failure on the receiving side, this situation is considered a fairly “severe” event and the channel recovery time can take tens or hundreds of milliseconds, has a devastating effect on any real-life networks and in most cases breaks the connection for all active communication sessions.
7. The size of the counters, and, accordingly, the maximum number of “time slots” is limited only by the size of the symbol (it is desirable to pack the counter value into one symbol, but not necessary). From a practical point of view, the number of time slots can be defined as the bits per second rate divided by the symbol size and equals the number of symbols per second transmitted.
8. **Result:** a very simple synchronization system is obtained, if it allows failures, then for a very short period of time and with an extremely low probability of such an event occurring, proportional to the probability of receiving from one to several defective “synchronization” symbols in a row (for the typical error probability in optical channel 10e-12 very rarely).

Simulation of counter synchronization in a data transmission channel on an FPGA (Quartus 9.0).

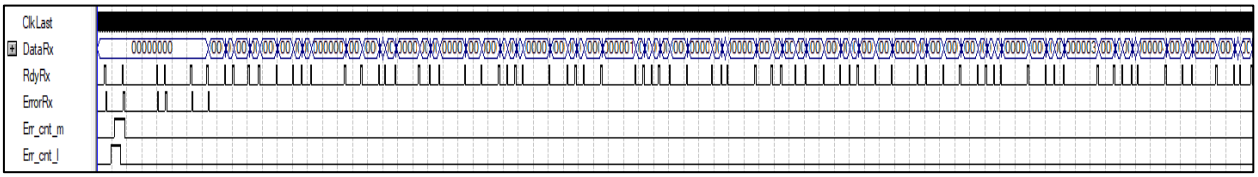
1. I will demonstrate the operation of the SSI network algorithms in the option of complete utilization of the physical channel and in the option with 10% utilization.
2. DataRx – received data.
3. RdyRx - data readiness strobe.
4. ErrorRx – error in the received sequence (at the initial stage it is generated due to the peculiarities of FIFO)

Demonstration of absence of transmission errors (100% download).

1. The data generated by the counter is transmitted, which means that to check it is enough to compare each next word with the previous one plus one.



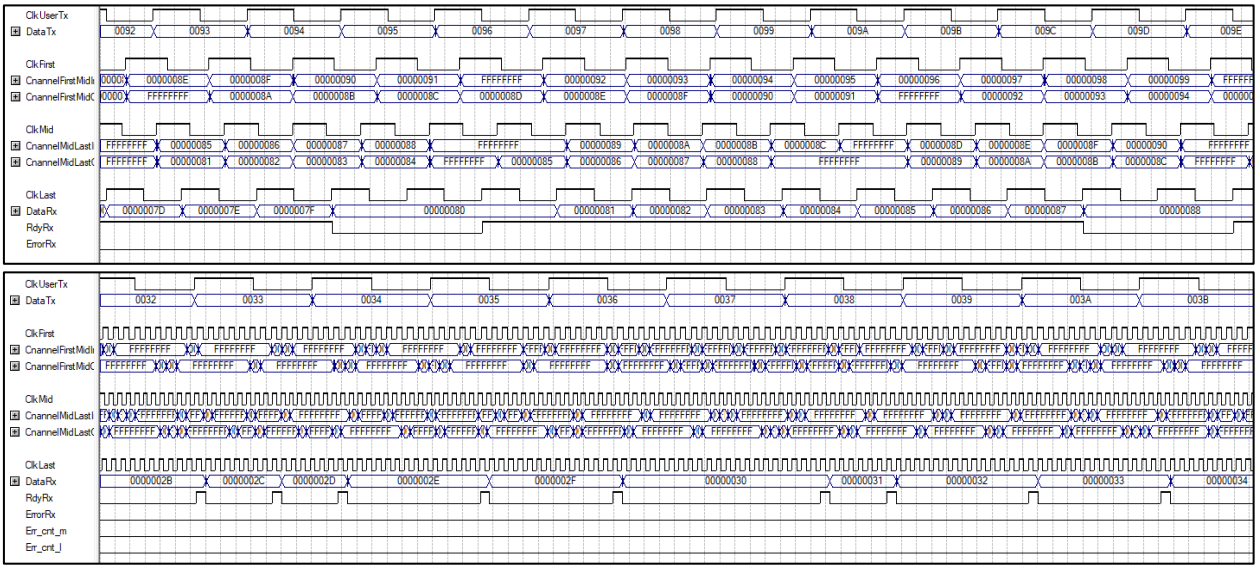
2. Demonstration of the absence of transmission errors (10% load). In this case, the absence of word counter synchronization errors has been added.



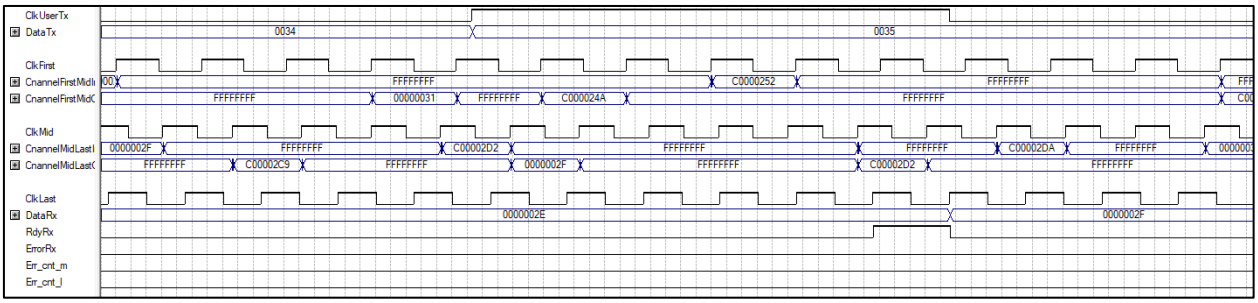
3. Err_cnt_m – error in synchronizing the counters of the first and second switch.
4. Err_cnt_l – error in synchronizing the counters of the second and third switch.
5. Errors in receiving data and updating counters are present only at the start of work.

Demonstration of the structure (sequence) of transmitted data.

1. The first picture is for 100% load, the second picture is for 10% load.



2. Enlarged image for 10% download.

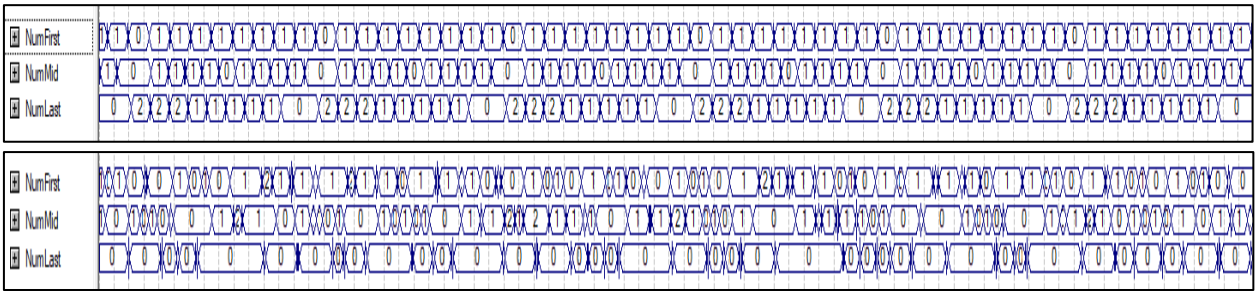


3. DataTx – transmitted data
4. Characters starting with “0” - transmitted data
5. Characters starting with “F” - the symbol “no data” or empty.
6. Symbols starting with “C” are a symbol for counter synchronization. “No data” is transmitted every eight characters.
7. In the first figure, the number of insertions of the “no data” symbol is minimal and proportional to the error in setting the synchronization frequency of the local generator.
8. For the option with a 10% load, there are significantly more such symbols; not all of the channel’s capacity is used for transmission.
9. The counter size is 16 binary digits
10. The FIFO of the switch core, which contains the transmitted data, does not respond to all counter values (the comparison occurs with a mirror image of the counter).

11. There is a decrease in the number of calls, and, accordingly, a decrease in the data transfer rate for a specific virtual channel.В первом коммутаторе канал передачи данных лежит по адресам от H”0000”- H”1999”.
12. In the second switch, the data transmission channel lies at the addresses H”ABCD”-H”C566”.
13. The transmission speed is determined by the size of the interval. In the version with 10% load, the interval size is calculated as 0.1 (10%) multiplied by the counter size (16 bits - 65536) and equals 6553 (H”1999”).
14. In the model, the increase in speed is taken into account when choosing the clock frequency of the transmitted data generator (less by double the inaccuracy of the clock generators used).

Demonstration of the number of characters in switch buffers.

1. Number of characters in the FIFO of each switch.



2. It can be seen that FIFO never contains more than two characters.

3. **Attention**: It is necessary to prove using mathematical methods that under no circumstances will the FIFO contain more than two characters or refute this statement.

4. If the statement is confirmed, then a switch on a modern process technology will be able to support millions of individual virtual channels. This number is unnecessary, but it gives confidence that there will be no problems with the number of simultaneous channels.

5. The maximum number of symbols in the FIFO uniquely determines the maximum switching time (traversing a separate switch) equal to two periods of symbol transmission in the virtual channel.

6. Switching time tied to the virtual channel speed “fairly” distributes the delay between the different channels. Higher transmission speed, shorter switching time. (This is not true for asynchronous channels.)

7. Switching time does not depend in any way on the degree of congestion of the shared (physical) channel. The problem can only arise with the allocation of the necessary bandwidth when creating another channel.

Conclusions:

1. The main problem is the novelty of the proposed network. The average person stops understanding texts after a novelty level of 20-30%. The descriptions of the SSI network, albeit according to a primitive assessment of the Antiplagiarism website, contain a degree of novelty of about 95%. This means that understanding the essence of the described algorithms (namely, the sum of their interactions) requires significant scientific competencies.

2. In the absence of specialized scientific schools and independent scientists in our country, it is very difficult to hope for such understanding.

3. I was unable to find anything similar in foreign publications. The most that is modern is Huawei's FlexM technology (https://info.support.huawei.com/info-finder/encyclopedia/en/FlexE.html), but this is too “far” to be mentioned in the literature section. All materials are marked as freely distributed, subject to attribution.

4. Answer from the specialized educational institution SibGUTI:

Владимир Фокин <mesos@rambler.ru>
1 сентября 2020, 9:50

Письмо и распечатка получены. Могу уверенно ответить, что в открытом доступе о научных школах указанного направления информации нет. Можно попробовать обратиться в компанию Т8, где работают известные специалисты по современным телекоммуникациям из МГУ, МГТУ им. Баумана, ФизТеха. Они ближе всего к разработкам такого типа и их практическому внедрению. Работу по этой тематике рекомендую продолжить. При наличии патентов и публикаций в научных изданиях из списка ВАК и Scopus может потянуть на учёную степень.
Фокин В.Г.

In the complete absence of scientific competencies, all that can be expected from the scientific and expert community is an indication of the absence of logical errors and recommendations for additional in-depth study of proposals.

To implement the SSI network, additional research, experiments, and convincing evidence are required, and all this must be done by scientists (or large companies) who have authority in the scientific community.

Model text on VERILOG.

```
//-------------------------------------------------------------------------------------------------//
//-- Developed by Balyberdin Andrey (Rutel@Mail.ru)
//-------------------------------------------------------------------------------------------------//
//-- All the ideas and algorithms described in this article are the result of my independent and
//-- completely independent intellectual activity. As an author,I allow any person or organization
//-- to freely use, modify, supplement all ideas and algorithms in any type of projects,
//-- with the mandatory indication of my authorship.
//--
//-- Author: Balyberdin Andrey (Rutel@Mail.ru)
//--
//-------------------------------------------------------------------------------------------------//
//--          Attribution 4.0 International (CC BY 4.0)
//-------------------------------------------------------------------------------------------------//
```

```
//
module appeal
        (
        // User data
        input   wire                    ClkUserTx,// 800kHz
        output  wire    [15:0]  DataTx,
        // First switch
        input   wire                    ClkFirst,// 900kHz
        output  wire    [3:0]   NumFirst,
        // Intermediate switch
        input   wire                    ClkMid,// 1100kHz
        output  wire    [3:0]   NumMid,
        // Last switch
        input   wire                    ClkLast,// 1000kHz
        output  wire    [3:0]   NumLast,
        // Channel
        output  wire    [31:0]  CnannelFirstMidInp,
        output  wire    [31:0]  CnannelFirstMidOut,
        output  wire    [31:0]  CnannelMidLastInp,
        output  wire    [31:0]  CnannelMidLastOut,
        //
        output  wire    [31:0]  DataRx,
        output  wire                    RdyRx,
        output  wire                    ErrorRx,
        output  wire                    Err_cnt_m,
        output  wire                    Err_cnt_l
        );
//-------------------------------------------------------------------------------------------------//
// DataTx
//-------------------------------------------------------------------------------------------------//
reg     [15:0]  DataUserRg;
assign DataTx[15:0]=DataUserRg[15:0];
```

```
//
always @(posedge ClkUserTx)
begin
DataUserRg[15:0]<=DataUserRg[15:0]+16'h01;
end
//-------------------------------------------------------------------------------------------------//
// Core first swich
//-------------------------------------------------------------------------------------------------//
wire                    SwichFirst_rdempty;
wire [31:0]             SwichFirst_q;
//
Fifo    SwichFirst
        (
        .data({16'h0,DataUserRg[15:0]}),
        .rdclk(ClkFirst),
        .rdreq(!SwichFirst_rdempty & (Cnt_f_o_m[15:0]>=16'h0) & (Cnt_f_o_m[15:0]<=16'h01999)),
        .wrclk(ClkUserTx),
        .wrreq(1'b01),
        .q(SwichFirst_q[31:0]),
        .rdempty(SwichFirst_rdempty),
        .rdusedw(NumFirst[3:0])
        );
//-------------------------------------------------------------------------------------------------//
// Channel First to Mid swich
//-------------------------------------------------------------------------------------------------//
wire                    ChannelFirstMid_rdempty;
wire [31:0]             ChannelFirstMid_q;
reg     [31:0]          ChannelFirstMid_data;
//
Fifo    ChannelFirstMid
        (
        .data(ChannelFirstMid_data[31:0]),
```

```
        .rdclk(ClkFirst),
        .rdreq(!ChannelFirstMid_rdempty),
        .wrclk(ClkFirst),
        .wrreq(1'b01),
        .q(ChannelFirstMid_q[31:0]),
        .rdempty(ChannelFirstMid_rdempty),
        .rdusedw()
        );
assign CnannelFirstMidInp[31:0]=ChannelFirstMid_data[31:0];
assign CnannelFirstMidOut[31:0]=ChannelFirstMid_q[31:0];
//-------------------------------------------------------------------------------------------//
// Core Mid swich
//-------------------------------------------------------------------------------------------//
wire                    SwichMid_rdempty;
wire    [31:0]          SwichMid_q;
//
Fifo    SwichMid
        (
        .data(ChannelFirstMid_q[31:0]),
        .rdclk(ClkMid),
        .rdreq(!SwichMid_rdempty & (Cnt_m_o_m[15:0]>=16'h0abcd) &
(Cnt_m_o_m[15:0]<=16'h0c566)),
        .wrclk(ClkFirst),
        .wrreq(ChannelFirstMid_q[31:28]==4'h0 &  (Cnt_m_i_m[15:0]>=16'h0) &
(Cnt_m_i_m[15:0]<=16'h01999)),
        .q(SwichMid_q[31:0]),
        .rdempty(SwichMid_rdempty),
        .rdusedw(NumMid[3:0])
        );
//-------------------------------------------------------------------------------------------//
// Channel Mid to Last swich
//-------------------------------------------------------------------------------------------//
wire                    ChannelMidLast_rdempty;
```

```
wire    [31:0]   ChannelMidLast_q;
reg     [31:0]   ChannelMidLast_data;
//
Fifo    ChannelMidLast
        (
        .data(ChannelMidLast_data[31:0]),
        .rdclk(ClkMid),
        .rdreq(!ChannelMidLast_rdempty),
        .wrclk(ClkMid),
        .wrreq(1'b01),
        .q(ChannelMidLast_q[31:0]),
        .rdempty(ChannelMidLast_rdempty),
        .rdusedw()
        );
assign CnannelMidLastInp[31:0]=ChannelMidLast_data[31:0];
assign CnannelMidLastOut[31:0]=ChannelMidLast_q[31:0];
//-------------------------------------------------------------------------------------------------//
// Core Last swich
//-------------------------------------------------------------------------------------------------//
wire                            SwichLast_rdempty;
wire    [31:0]          SwichLast_q;
reg                                     ErrorRx_rg;
reg             [31:0]          DataRx_Old;
//
Fifo    SwichLast
        (
        .data(ChannelMidLast_q[31:0]),
        .rdclk(ClkLast),
        .rdreq(!SwichLast_rdempty),
        .wrclk(ClkMid),
        .wrreq(ChannelMidLast_q[31:28]==4'h0 &  (Cnt_l_i_m[15:0]>=16'h0abcd) &
(Cnt_l_i_m[15:0]<=16'h0c566)),
```

```
        .q(SwichLast_q[31:0]),
        .rdempty(SwichLast_rdempty),
        .rdusedw(NumLast[3:0])
        );
//
assign DataRx[31:0]=SwichLast_q[31:0];
assign RdyRx=!SwichLast_rdempty;
assign ErrorRx=ErrorRx_rg;
assign test=ChannelMidLast_q[31:28]==4'h0 &  (Cnt_l_i_m[15:0]>=16'h0) &
(Cnt_l_i_m[15:0]<=16'h01999);
//
always @(*)
begin
if ((SwichFirst_rdempty | Cnt_f_o_m[15:0]<16'h0 | Cnt_f_o_m[15:0]>16'h01999) &
Cnt_f_ff[2:0]!=3'h07)ChannelFirstMid_data[31:0]<=32'hffffffff;
if ((SwichFirst_rdempty | Cnt_f_o_m[15:0]<16'h0 | Cnt_f_o_m[15:0]>16'h01999) &
Cnt_f_ff[2:0]==3'h07)ChannelFirstMid_data[31:0]<={16'h0c000,Cnt_f_o[15:0]};
if (!SwichFirst_rdempty & Cnt_f_o_m[15:0]>=16'h0 &
Cnt_f_o_m[15:0]<=16'h01999)ChannelFirstMid_data[31:0]<=SwichFirst_q[31:0];
//
if ((SwichMid_rdempty | Cnt_m_o_m[15:0]<16'h0abcd | Cnt_m_o_m[15:0]>16'h0c566) &
Cnt_m_ff[2:0]!=3'h07)ChannelMidLast_data[31:0]<=32'hffffffff;
if ((SwichMid_rdempty | Cnt_m_o_m[15:0]<16'h0abcd | Cnt_m_o_m[15:0]>16'h0c566) &
Cnt_m_ff[2:0]==3'h07)ChannelMidLast_data[31:0]<={16'h0c000,Cnt_m_o[15:0]};
if (!SwichMid_rdempty & Cnt_m_o_m[15:0]>=16'h0abcd &
Cnt_m_o_m[15:0]<=16'h0c566)ChannelMidLast_data[31:0]<=SwichMid_q[31:0];
//
end
// SwichFirst
reg         [15:0]      Cnt_f_o;
wire    [15:0]      Cnt_f_o_m;
reg         [2:0]       Cnt_f_ff;
reg                     Err_cnt_mid;
//
assign Cnt_f_o_m[0]=Cnt_f_o[15];
```

```
assign Cnt_f_o_m[1]=Cnt_f_o[14];
assign Cnt_f_o_m[2]=Cnt_f_o[13];
assign Cnt_f_o_m[3]=Cnt_f_o[12];
assign Cnt_f_o_m[4]=Cnt_f_o[11];
assign Cnt_f_o_m[5]=Cnt_f_o[10];
assign Cnt_f_o_m[6]=Cnt_f_o[9];
assign Cnt_f_o_m[7]=Cnt_f_o[8];
assign Cnt_f_o_m[8]=Cnt_f_o[7];
assign Cnt_f_o_m[9]=Cnt_f_o[6];
assign Cnt_f_o_m[10]=Cnt_f_o[5];
assign Cnt_f_o_m[11]=Cnt_f_o[4];
assign Cnt_f_o_m[12]=Cnt_f_o[3];
assign Cnt_f_o_m[13]=Cnt_f_o[2];
assign Cnt_f_o_m[14]=Cnt_f_o[1];
assign Cnt_f_o_m[15]=Cnt_f_o[0];
//
assign Cnt_m_i_m[0]=Cnt_m_i[15];
assign Cnt_m_i_m[1]=Cnt_m_i[14];
assign Cnt_m_i_m[2]=Cnt_m_i[13];
assign Cnt_m_i_m[3]=Cnt_m_i[12];
assign Cnt_m_i_m[4]=Cnt_m_i[11];
assign Cnt_m_i_m[5]=Cnt_m_i[10];
assign Cnt_m_i_m[6]=Cnt_m_i[9];
assign Cnt_m_i_m[7]=Cnt_m_i[8];
assign Cnt_m_i_m[8]=Cnt_m_i[7];
assign Cnt_m_i_m[9]=Cnt_m_i[6];
assign Cnt_m_i_m[10]=Cnt_m_i[5];
assign Cnt_m_i_m[11]=Cnt_m_i[4];
assign Cnt_m_i_m[12]=Cnt_m_i[3];
assign Cnt_m_i_m[13]=Cnt_m_i[2];
assign Cnt_m_i_m[14]=Cnt_m_i[1];
assign Cnt_m_i_m[15]=Cnt_m_i[0];
```

```
//
assign Cnt_m_o_m[0]=Cnt_m_o[15];
assign Cnt_m_o_m[1]=Cnt_m_o[14];
assign Cnt_m_o_m[2]=Cnt_m_o[13];
assign Cnt_m_o_m[3]=Cnt_m_o[12];
assign Cnt_m_o_m[4]=Cnt_m_o[11];
assign Cnt_m_o_m[5]=Cnt_m_o[10];
assign Cnt_m_o_m[6]=Cnt_m_o[9];
assign Cnt_m_o_m[7]=Cnt_m_o[8];
assign Cnt_m_o_m[8]=Cnt_m_o[7];
assign Cnt_m_o_m[9]=Cnt_m_o[6];
assign Cnt_m_o_m[10]=Cnt_m_o[5];
assign Cnt_m_o_m[11]=Cnt_m_o[4];
assign Cnt_m_o_m[12]=Cnt_m_o[3];
assign Cnt_m_o_m[13]=Cnt_m_o[2];
assign Cnt_m_o_m[14]=Cnt_m_o[1];
assign Cnt_m_o_m[15]=Cnt_m_o[0];
//
assign Cnt_l_i_m[0]=Cnt_l_i[15];
assign Cnt_l_i_m[1]=Cnt_l_i[14];
assign Cnt_l_i_m[2]=Cnt_l_i[13];
assign Cnt_l_i_m[3]=Cnt_l_i[12];
assign Cnt_l_i_m[4]=Cnt_l_i[11];
assign Cnt_l_i_m[5]=Cnt_l_i[10];
assign Cnt_l_i_m[6]=Cnt_l_i[9];
assign Cnt_l_i_m[7]=Cnt_l_i[8];
assign Cnt_l_i_m[8]=Cnt_l_i[7];
assign Cnt_l_i_m[9]=Cnt_l_i[6];
assign Cnt_l_i_m[10]=Cnt_l_i[5];
assign Cnt_l_i_m[11]=Cnt_l_i[4];
assign Cnt_l_i_m[12]=Cnt_l_i[3];
assign Cnt_l_i_m[13]=Cnt_l_i[2];
```

```
assign Cnt_l_i_m[14]=Cnt_l_i[1];

assign Cnt_l_i_m[15]=Cnt_l_i[0];

//

always @(posedge ClkFirst)

begin

Cnt_f_o[15:0]<=Cnt_f_o[15:0]+16'h01;

if(ChannelFirstMid_q[31:28]==4'h0c)Cnt_m_i[15:0]<=ChannelFirstMid_q[15:0]+16'h01; else
Cnt_m_i[15:0]<=Cnt_m_i[15:0]+1;

if(ChannelFirstMid_q[31:28]==4'h0c)Err_cnt_mid<=Cnt_m_i[15:0]!=ChannelFirstMid_q[15:0];

if(SwichFirst_rdempty | Cnt_f_o_m[15:0]<16'h0 |
Cnt_f_o_m[15:0]>16'h01999)Cnt_f_ff[2:0]<=Cnt_f_ff[2:0]+3'h01;

end

// SwichMid

reg             [15:0]          Cnt_m_i;

reg             [15:0]          Cnt_m_o;

reg             [15:0]          Cnt_l_i;

wire    [15:0]          Cnt_m_i_m;

wire    [15:0]          Cnt_m_o_m;

wire    [15:0]          Cnt_l_i_m;

reg             [2:0]           Cnt_m_ff;

reg                                     Err_cnt_last;

//

always @(posedge ClkMid)

begin

Cnt_m_o[15:0]<=Cnt_m_o[15:0]+16'h01;

if(ChannelMidLast_q[31:28]==4'h0c)Cnt_l_i[15:0]<=ChannelMidLast_q[15:0]+16'h01; else
Cnt_l_i[15:0]<=Cnt_l_i[15:0]+1;

if(ChannelMidLast_q[31:28]==4'h0c)Err_cnt_last<=Cnt_l_i[15:0]!=ChannelMidLast_q[15:0];

if(SwichMid_rdempty | Cnt_m_o_m[15:0]<16'h0abcd |
Cnt_m_o_m[15:0]>16'h0c566)Cnt_m_ff[2:0]<=Cnt_m_ff[2:0]+3'h01;

end

//

assign Err_cnt_m=Err_cnt_mid;
```

```
assign Err_cnt_l=Err_cnt_last;
//
always @(posedge ClkLast)
begin
if (!SwichLast_rdempty)DataRx_Old[31:0]<=SwichLast_q[31:0];
ErrorRx_rg<=!SwichLast_rdempty & (DataRx_Old[31:0]+32'h01)!=SwichLast_q[31:0];
end
//
endmodule
```