AXI splitter

G.J. van Loo, October 2017
Fen Logic Ltd.
(verilog@fenlogic.com)

Introduction.

An AXI splitter is a bit more complex then its complement: the AXI mux (or arbiter or switch, whatever you want to call them). The reason is that the return data has to be kept ordered. This document describes the AXI splitters as designed by me, going into more details why I have made certain decisions.

What is a splitter?

A splitter is a component which takes an AXI bus and splits it into two or more branches. The traffic is routed to a branch based on the address. The return data from each branch is multiplexed back to the original stream.

How to split.

To split an AXI stream the competent must look at the address and based on that route the data to one of the outputs. In order to keep my designs simple and fast the address split can take place only on values which are a power of two. The address range of each port is set using a series of parameters.

Default port.

What if address matches none of the ports? In that case we have two choices:

- Route the traffic to a default port
- Generate an error condition (b-response or r-response error code).

I have decided not to add error handling inside the splitter as it complicates matter. This can just as easily done with an external AXI error component without incurring an extra costs in area or speed. It also make it easier to make asymmetric splitters where only a small part of the AXI stream is diverted.
Therefore the user specifies only the address ranges of the first ports. The last port is a default port which handles all remaining traffic.

How to return.

On the AXI bus there are two streams of data which have to be returned: the b-response stream and the read data stream. Unfortunately we can ***not*** use the same system as implemented in the AXI mux, a round robin or priority port selection algorithm. This makes that a splitter is a bit more complex then a mux. Why is this?

Ordering

The AXI rules specify that data has to be returned in-order for a certain ID. Say we have a splitter which splits the stream according to our very first example in appendix-A: The bottom 2Gbyte go to output port0 the top 2Gbyte go to output port 1.
Our CPU does a read from 0x00400000 (bottom 2 G) followed by a read from 0xC000000 (top 2G). Also assume the data from the top comes back faster (e.g. the top read is from a register whilst the bottom read is from DDR where we happen to have a page miss).

The splitter gets read data back on port 1, ***but it is not allowed to let it through***! If it would pass the data it would return to the CPU which is expecting data back in-order. Thus the CPU gets the register data but thinks it was the result from the DDR read. Very bad! Thus the splitter must remember for which port it has to service the data first.

Now we have a dual CPU core in front of our splitter. We get the same scenario, but the first read is from CPU-0 and the second from CPU-1. Because they have different sources the AXI ID will be different. In this case it is perfectly safe to return the early data on port 1 as in the end it will be routed back to a different source. ***Or will it?***

Lock-up!
There are worse scenarios. Assume a system with two AXI masters and two AXI slaves. We get the following sequence:
Master A reads from slave A, followed by a read from slave B, $1{:}M_AS_A$, $2{:}M_AS_B$
Master B reads from slave B, followed by a read from slave A, $1{:}M_BS_B$, $2{:}M_BS_A$

Now for some reason the data which was read last, returns first. Thus the splitter sees on the two output ports the replies to the reads of: $2{:}M_AS_B$, $2{:}M_BS_A$.
The splitter is not allowed to pass the data thus it waits for the returns from the first reads. However that data is queued up ***behind*** the current data and will not appear until that data has been dealt with first.
At this moment we have a lock-up!

Why did this lock-up happen?
Each AXI slaves sees two reads arriving.

- Slave A sees: $1{:}M_AS_A$ followed by $2{:}M_BS_A$
- Slave B sees: $1{:}M_BS_B$ followed by $2{:}M_AS_B$

Each read has a different ID, thus the slave is, according to the AXI rules, free to return the read data in arbitrary order and does so, returning the reply to the second request first.
Further upstream we have our splitter which is faithfully waiting for the read data to return. It knows that it has to return the data in order thus on one port it needs to have $1{:}M_AS_A$ before it can pass $2{:}M_AS_B$ from the other port. At the same time it needs to have $1{:}M_BS_B$ before it can pass $2{:}M_BS_A$. Neither is going to happen and we have lock-up.

Theoretical it is possible to store the arriving data in a local buffer in the splitter and let only the 'right' data pass. However the size of such buffer would quickly explode beyond anything reasonable, especially if the system supports very large bursts.

The solution is simple and costly:
In order to prevent lock-up any component must return the data in-order. The reason why it is costly: it completely thwarts the AXI system which was intended to support multiple independent streams based on different IDs. Especially the idea that you can have fast streams co-exist with slow streams in one uniform architecture is not possible[1]. You will get fast response data stuck waiting for the slow data because of this.
What is also dangerous: you can have a system where this sequence happens only on very rare occasions. As such you can see a lock-up happen once every few hours, days or even weeks.

1 Unless the user is prepared to throw huge number of logic gates at the problem. In which case it becomes simpler to have multiple independent paths throughout your architecture.

Implementation

In this chapter you find some details about the implementation

Forward data

In the forward data direction (slave input to master outputs) the system tries to pass the data as quickly as possible, thus without any clock cycles of delay. However it needs to decode the address before it knows at which port to output the data. Until then the valid signal is blocked. Thus each splitter will increase the valid-in to valid-out timing on the address buses.

On the data port there is a likewise delay. The data can only be routed if the address is known. Thus the data-valid in can not be passed until the address has been decoded and a similar delay is added from s_wvalid to mx_wvalid with tha additional complexity that the mx_wvalid depends also on the arrival of the s_awvalid signal.

Its also implies that the splitter can not pass data if it arrives before the address. This may lead to a different type of lock-up: If a slave outputs data and will not output the address until the data has been accepted the system will lock-up. However such conditions are very unlikely and will show up at the very first simulation.

Thirdly the write address routing queue is only a single entry. Thus the next write ***address*** cannot be passed until the previous write ***data*** has completed. It would be not too difficult to add a small FIFO here too but I do not have plans to make such a splitter at the moment. (Or mail me with a cheque :-).

Return data.

As mentioned the splitter will only return data in-order. To do this it has to 'remember' the order in which output ports were serviced. It has to do this twice: for the write bus it has to route the b-response streams. For the read bus it has to route the return data.

The splitter uses a FIFO which holds the order in which the ports are serviced. The FIFO is written to for each address which leaves the splitter. The FIFO output is used to select the return port (It takes one clock cycle for the data to fall-through). Once the last data unit has been returned the entry is removed (read) from the FIFO.

The splitter has a parameter which specifies how deep the FIFOs are, thus how many addresses can be queued up. Once a FIFO is full it will not longer forward data and the splitter will wait until data comes back. There is a FIFO for each bus, the read bus and the write bus.

Parameters.

The splitter has a number of parameters. Besides the standard ones (Number of address-, data- or ID-bits) it has the following splitter specific parameters:

MASK.

There is a MASK parameter which specifies the address bits the splitter must use in the comparison. It is perfectly valid for a user to specify a lot of mask bits but that will increase the size of the compare logic and thus reduce the operating speed of the module. For a two output splitter the optimum number of mask bits is one. For a three-four way splitter it is two etc. The user can also use bits which are not adjacent but that may make it more difficult for the end user (the programmer) to understand which traffic goes where. However it does not impact the size or the operating speed of the splitter.

Every splitter has only ***one*** mask parameter.
For examples see Appendix-A.

VALUE0..VALUEn
The VALUE parameters specify what the address range is of an output stream. VALUE0 controls the output port 0, VALUE1 controls ports1 etc.
A port is selected if the address meets the condition (address & MASK)=VALUEn.
For an N output splitter there are N-1 VALUE parameters. The highest output port collects all traffic which does not meet any of the previous port addresses.
For examples see Appendix-A.

L2MAXTRANS.
The L2MAXTRANS parameter specifies the depth of the return FIFOs. The depth is specified in powers of two thus the parameter is actually the 2-log of the depth. e.g. 3 means 2^3=8 entries deep, 4 means 2^4=16 entries deep etc.

Appendix-A: Address decode examples.

Just to give an impression of how the address decoder system works and what can be achieved with the mechanism here follow some examples. Although I have though long and hard about it, it is still possible this section contains errors so caveat emptor.

Two way splitter examples

Split 32-bit address range in two equal areas.

```
MASK   = 32'h8000_0000
VALUE0 = 32'h0000_0000
Output port 0: address range 0x00000000 .. 0x7FFFFFFF (2GB)
Output port 1: address range 0x80000000 .. 0xFFFFFFFF (2GB)
```

Split 32-bit address range into a small and big area.

```
MASK   = 32'hF000_0000
VALUE0 = 32'h0000_0000
Output port 0: address range 0x00000000 .. 0x0FFFFFFF (256MB)
Output port 1: address range 0x10000000 .. 0xFFFFFFFF (~3.8GB)
```

A splitter like this may be placed after a previous splitter e.g. port 0 of the example above. As the address range has already been reduced to `0x00000000 .. 0x0FFFFFFF` only the first of the sixteen entries (shown in italics) can happen .

```
MASK   = 32'h0F00_0000
VALUE0 = 32'h0000_0000
Output port 0: address range 0x00000000 .. 0x00FFFFFF
               address range 0x10000000 .. 0x10FFFFFF
               address range 0x20000000 .. 0x20FFFFFF
               . . . . .
               address range 0xF0000000 .. 0xF0FFFFFF
               (16 times 16MB)
Output port 1: address range 0x01000000 .. 0x0FFFFFFF
               address range 0x11000000 .. 0x1FFFFFFF
               address range 0x21000000 .. 0x2FFFFFFF
               . . . . .
               address range 0xF1000000 .. 0xFFFFFFFF
               (16 times 240MB)
```

Four way splitter examples.

Split 32-bit address range in two equal areas.

```
MASK   = 32'hC000_0000
VALUE0 = 32'h0000_0000
VALUE1 = 32'h4000_0000
VALUE2 = 32'h8000_0000
Output port 0: address range 0x00000000 .. 0x3FFFFFFF (1GB)
Output port 1: address range 0x40000000 .. 0x7FFFFFFF (1GB)
Output port 2: address range 0x80000000 .. 0xBFFFFFFF (1GB)
Output port 3: address range 0xC0000000 .. 0xFFFFFFFF (1GB)
```

Split 32-bit address range in three small and one big area.

```
MASK   = 32'hF000_0000
VALUE0 = 32'h0000_0000
VALUE1 = 32'h1000_0000
VALUE2 = 32'h2000_0000
Output port 0: address range 0x00000000 .. 0x0FFFFFFF (256MB)
Output port 1: address range 0x10000000 .. 0x1FFFFFFF (256MB)
Output port 2: address range 0x20000000 .. 0x2FFFFFFF (256MB)
Output port 3: address range 0x30000000 .. 0xFFFFFFFF (~3.3GB)
```

This is a follow-up splitter for one of the output ports 0,1 or 2 in the previous example. The stream which is already reduced to have a 256MB address range is further split into four sections. Each output port has sixteen address ranges but because of the pre-selection only one is actually used.

```
MASK   = 32'h0E00_0000
VALUE0 = 32'h0000_0000
VALUE1 = 32'h0200_0000
VALUE2 = 32'h0400_0000
Output port 0: address range 0x?0000000 .. 0x?1FFFFFF (32MB)
Output port 1: address range 0x?2000000 .. 0x?3FFFFFF (32MB)
Output port 2: address range 0x?4000000 .. 0x?7FFFFFF (32MB)
Output port 3: address range 0x?8000000 .. 0x?FFFFFFF (160MB)
```

This shows an example where the MASK bits are NOT adjacent. It produces a more complex address map although with a certain regularity.

```
MASK   = 32'h9000_0000
VALUE0 = 32'h0000_0000
VALUE1 = 32'h1000_0000
VALUE2 = 32'h8000_0000
Output port 0: address range 0x00000000 .. 0x0FFFFFFF (256MB)
               address range 0x20000000 .. 0x2FFFFFFF (256MB)
               address range 0x40000000 .. 0x4FFFFFFF (256MB)
               address range 0x60000000 .. 0x6FFFFFFF (256MB)
Output port 1: address range 0x10000000 .. 0x1FFFFFFF (256MB)
               address range 0x30000000 .. 0x3FFFFFFF (256MB)
               address range 0x50000000 .. 0x5FFFFFFF (256MB)
               address range 0x70000000 .. 0x7FFFFFFF (256MB)
Output port 2: address range 0x80000000 .. 0x8FFFFFFF (256MB)
               address range 0xA0000000 .. 0xAFFFFFFF (256MB)
               address range 0xC0000000 .. 0xCFFFFFFF (256MB)
               address range 0xE0000000 .. 0xEFFFFFFF (256MB)
Output port 3: address range 0x90000000 .. 0x9FFFFFFF (256MB)
               address range 0xB0000000 .. 0xBFFFFFFF (256MB)
               address range 0xD0000000 .. 0xDFFFFFFF (256MB)
               address range 0xF0000000 .. 0xFFFFFFFF (256MB)
```