Michael R Honar
September 8, 2025

Beam Mux Assignment

Introduction

I am assigned to write and test a Beam Mux module in system verilog. My submission is this document along with the attached "BeamMux-Honar.zip" file, which contains source code, stimulus files, a simple test bench and drawings and the prompt. I used VSCode to write my code and a free version of Vivado 2025.1 to perform the simulation.

My experience is in VHDL. Yes, I took a class using System Verilog in college (others used VHDL), but that was 7 years ago. A good chunk of this assignment was spent comparing VHDL to (System) Verilog syntax and design. I read a couple of Clifford Cumming's white papers, used Vivado's language templates, and used the internet tools to familiarize myself with System Verilog to complete this task in approximately 10 hours (and another 5 hours of familiarizing myself with System Verilog). I do think this a pretty hefty assignment with practical and deployable applications on a real FPGA.

I completed the design to specification. I introduce some assumptions, then explain my design and verification methodologies in detail below. I then briefly go over considered designs. To conclude, I explore potential optimizations.

Assumptions

No tlast specified on the DAC's AXI-Stream.

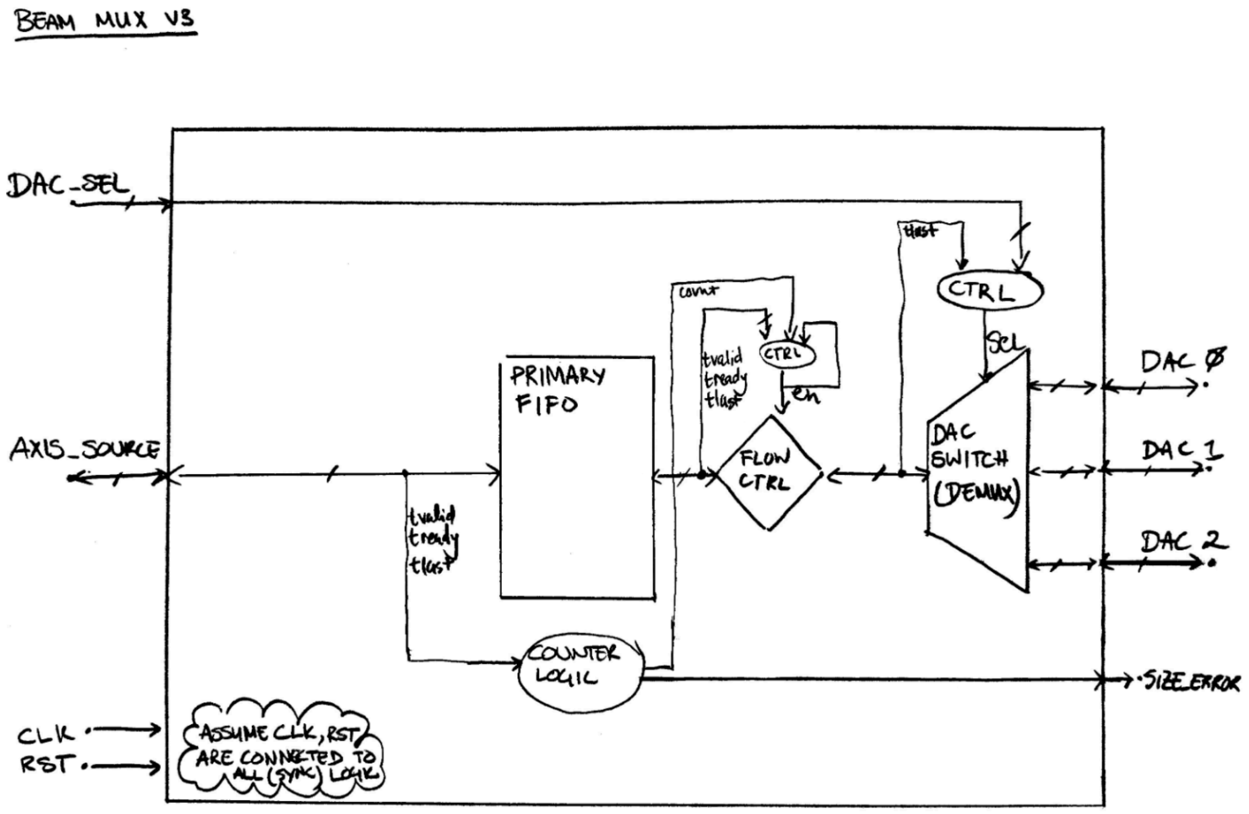
> I initially hinted at this during an email exchange with Maxwell, and I was invited to question: how would the DAC use tlast and how would it change its behavior?
>
> To answer, I think DAC would not use tlast, based on the assumption in the assignment: "A DAC should receive an uninterrupted stream of samples". With the information provided to me, it seems like the DAC relies on a set configured size of data to convert digital samples into analog. Maybe a counter is used. with tlast, that counter can go away, but then the DAC will not know the sample size until the end, and have a detrimental effect. Maybe the samples received are supersampled multiple times so the DAC has valid data until the reception of the next stream. For the second part of the question, the DAC's input processing would be affected, to handle tlast. There's not enough information for me to answer this question. However, I can imagine that maybe the DAC relies on counters to know when the sample is complete, instead of a tlast flag. I explored this possibility earlier.

Bursts are of size 1024 - 65536.

> If a burst is not within this range, a size error flag will be asserted.

Design (Beam Mux V3)

Beam Mux Block Diagram V3.pdf



This design is pretty straight forward: instantiate an Xilinx Parameterized Macro AXI-S FIFO (Primary FIFO) and write counter logic to keep track of the number of complete bursts stored in the Primary FIFO. Implement flow-control logic to ensure the DACs only receive an uninterrupted stream without tvalid being de-asserted before tlast. Finally, write a demux circuit to select which DAC to feed data to.

The Primary FIFO needs to be sized to fit the largest data stream scenario: two or more back-to-back 65536 sized bursts. Therefore, the Primary FIFO will be sized to a depth of 65536*2 and a width of 4 bytes. This ensures the FIFO can hold two large bursts: 1 for ingestion, 1 for output to the DAC. I explore depth reductions later.

The counter logic counts how many data packets are being sent per burst, relying on AXIS_SOURCE's tvalid and tready. tlast (always qualify tlast with and tvalid and tready) stops the "current_burst_size" and increments "burst_count". A "size_error" flag is asserted if "current_burst_count" is not within [1024, 65536].

The "burst_count" can be of small width. I recommend making the width as wide as it needs to count: however many 1024 sample streams can fit in the primary FIFO +1. So if the primary FIFO is 65536*2 samples deep, then make this width FIFO 128+1 deep. The plus 1 is to account for the edge case where the Primary FIFO has 128 complete bursts, one of them outputting to the DAC, and in the process of ingesting a 129th burst from the modulator.

$$Max\ Count\ =\ \frac{Primary\ FIFO\ Depth}{Minimum\ Burst\ Size} + 1$$

In this case,

$$Max\ Count\ =\ \frac{66536 * 2}{1024} + 1\ =\ 129$$

Therefore the width of the "burst_count" shall be 8 bits wide. This makes it impossible to overflow the "burst_count" value unless a "size_error" occurs. It is possible to size this value to a maximum of 128 (7 bits wide depth) with additional flow control. To complete the design within an evening's time, I did not include this minor optimization.

Flow control is implemented in the Beam Mux. The upstream flow control is implemented via the Primary FIFO using the XPM's built in tready logic handler. I did design a downstream flow control circuit placed after the Primary FIFO in the Beam Mux. The downstream flow control blocks the DAC's tready signal unless the Primary FIFO has a full stream, inferred by checking if the "burst_count" > 0. Refer to the source code for more details.

At the very end of the module, there is another switch (Demux), used to select which DAC to feed. This circuitry depends on i_dac_select and gets set on tlast (qualified with tvalid and tready) of the downstream axi-stream to select the correct DAC on a new stream.

If this was a real work assignment, I would tweak this design for latency and throughput optimizations. As part of the final design report, I will synthesize and PAR the design to receive a timing report and coverage report.

Test Bench

I was able to verify this module, and it is fully functional meeting the requirements listed:

1. Arbitrary rate samples
2. Uninterrupted stream of samples fed to the DAC
3. Handle back pressure from the DAC
4. Perform proper DAC selection

It is understood that a Test Bench is typically a suite of files with the following:

1. The test bench itself: a top wrapper instantiating the DUT with clocks and resets, as well as the other test bench modules
2. A stimulus, typically an input
3. A monitor, typically something to validate the DUT to a truth.

To complete this design within the allocated time, I had to make some compromises on the test bench, by incorporating 1-3 above in a single file and separate top wrapper. I did generate 16 random binary files of various sizes for the stimulus. The monitor is incomplete, and I resolved the data validation by inspection of the data stream (explained below). I included my waveform configuration file as well.

I explain how each component of the test bench operates in the *tb_sim_beam_mux.sv* file.

The test bench is semi-configurable. An absolute path must be specified for where the binary input files are located, along with the number of files and a parameter for i_dac_sel.

The test bench works by reading a binary file, 4 bytes at a time, and simulating the Modulator's AXI-S interface. A data counter keeps track of how many 4byte words are read, and the tready and tvalid signals are used for transmission of all bytes to the Beam Mux.

The test bench completes after all input files have been processed and an additional 1us allotted to the DUT to process the remaining streams.

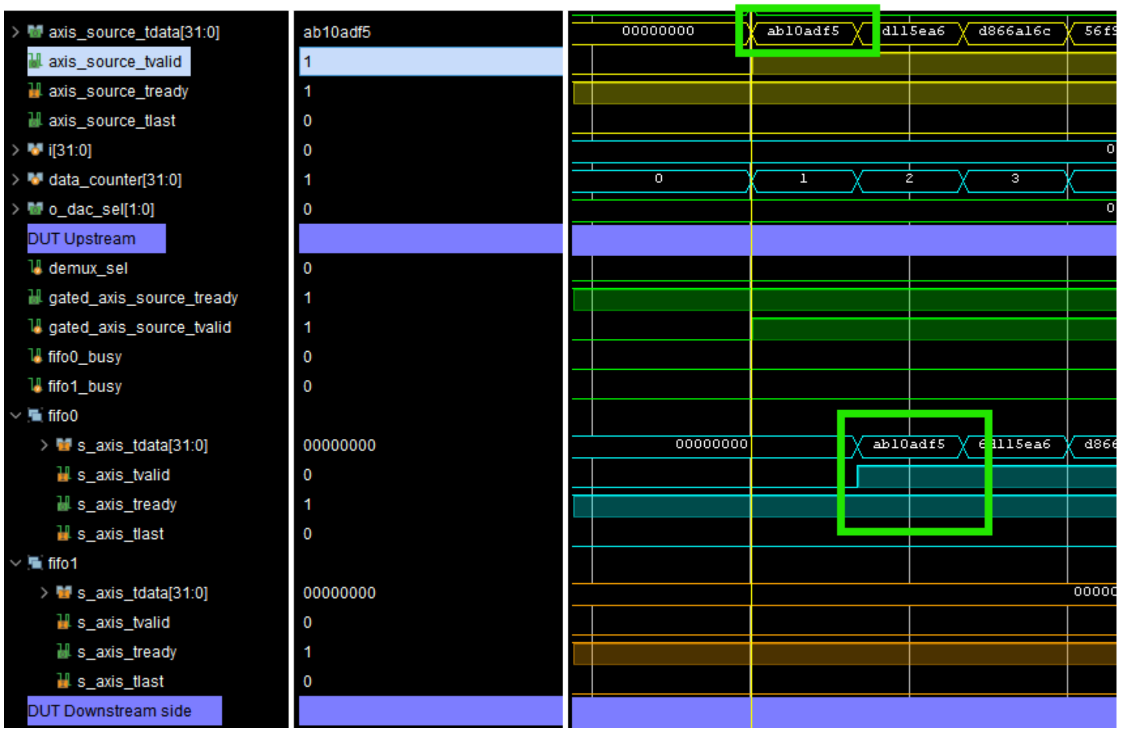
The waveform monitor has been designed for intuitive manual inspection. A monitor was thought out and designed, the details are listed in the test bench's comments. Due to this, I explain the validation methodology in great detail below.

Manual Inspection

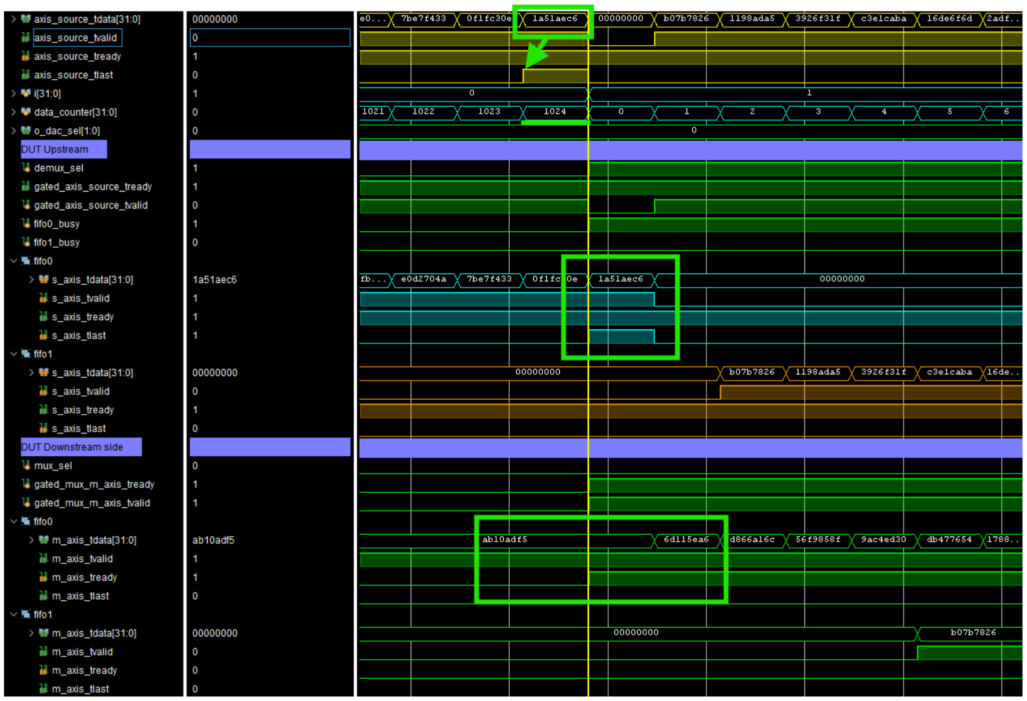
The stim consists of pseudorandom binary files, making the test bench results of the beam mux deterministic. Here's a list of all the files, and their first and last 4 bytes of data. I use this information as an initial glance-over, that the data is transmitted to the ping-pong FIFOs correctly.

File	First 4B	Last 4B	**File**	First 4B	Last 4B
00.bin	`0xAB10AD15`	`0x1A51AEC6`	08.bin	`0xFB0F9835`	`0xEA196845`
01.bin	`0xB07B7826`	`0x9164236D`	09.bin	`0x5866EDC5`	`0xD8F8349A`
02.bin	`0x2A613EB9`	`0x8EA917F3`	10.bin	`0xCC8A7BDB`	`0xD9B55D8E`
03.bin	`0xF8CE015F`	`0x8CE4F4CA`	11.bin	`0x4F8F4B9B`	`0xB2691A82`
04.bin	`0xE0766FA6`	`0x4EE8F3C2`	12.bin	`0x4E6A292F`	`0xA345638B`
05.bin	`0x4D0B83B4`	`0x0C6434C6`	13.bin	`0xBB294AA7`	`0x14F63084`
06.bin	`0x81EDBB7F`	`0x7433C72B`	14.bin	`0x712F596E`	`0x502EBFE3`
07.bin	`0x562F0673`	`0x64DA0398`	15.bin	`0x3BC90843`	`0x2EE501DE`

1. Inspect start of stim data
   To perform this step, I inspected the waveform of the axis_source AXI-S for correct data and length. The figure shows the starting data stream from the Stim and into the correct pingpong FIFO. The data matches the `0xAB10AD15` from the *00.bin* input file.

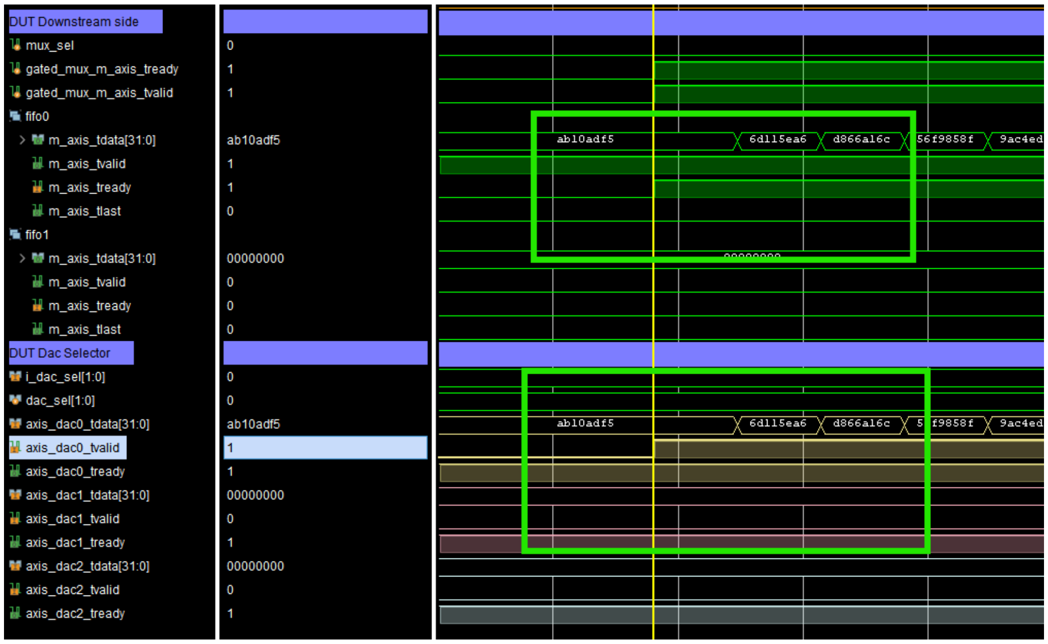


2. Inspect end of stim stream and pingpong buffer
   For this step, I inspected the final data packets of the stim AXI-S. The value, `0x1A51AEC6`, matches the final 4 bytes of the *00.bin* input file. Also note that tlast is asserted in the correct position, after the 1024th packet has been sent. The FIFO registers the last word, and my flow control logic recognizes that the stream is complete and begins streaming the data to the DAC.



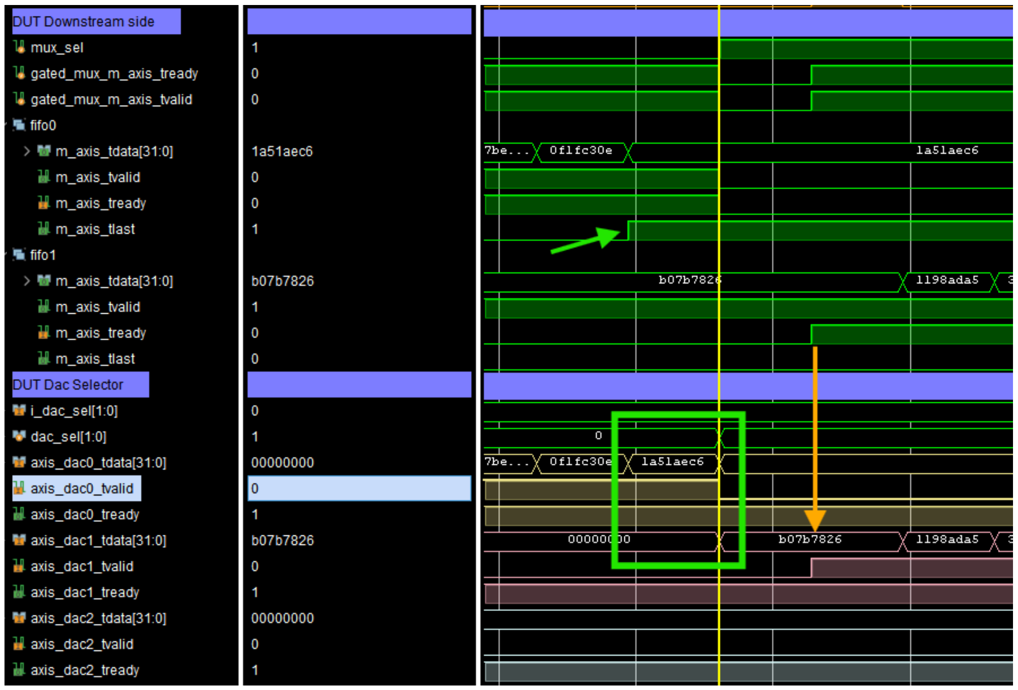
```
axis_source_tdata[31:0]     00000000
axis_source_tvalid          0
axis_source_tready          1
axis_source_tlast           0
i[31:0]                     1
data_counter[31:0]          0
o_dac_sel[1:0]              0
DUT Upstream
demux_sel                   1
gated_axis_source_tready    1
gated_axis_source_tvalid    0
fifo0_busy                  1
fifo1_busy                  0
fifo0
  s_axis_tdata[31:0]        1a51aec6
  s_axis_tvalid             1
  s_axis_tready             1
  s_axis_tlast              1
fifo1
  s_axis_tdata[31:0]        00000000
  s_axis_tvalid             0
  s_axis_tready             1
  s_axis_tlast              0
DUT Downstream side
mux_sel                     0
gated_mux_m_axis_tready     1
gated_mux_m_axis_tvalid     1
fifo0
  m_axis_tdata[31:0]        ab10adf5
  m_axis_tvalid             1
  m_axis_tready             1
  m_axis_tlast              0
fifo1
  m_axis_tdata[31:0]        00000000
  m_axis_tvalid             0
  m_axis_tready             0
  m_axis_tlast              0
```

3. DAC selector for output and inspect initial DAC feed.
   For the 1st stream, DAC0 is selected (correct).



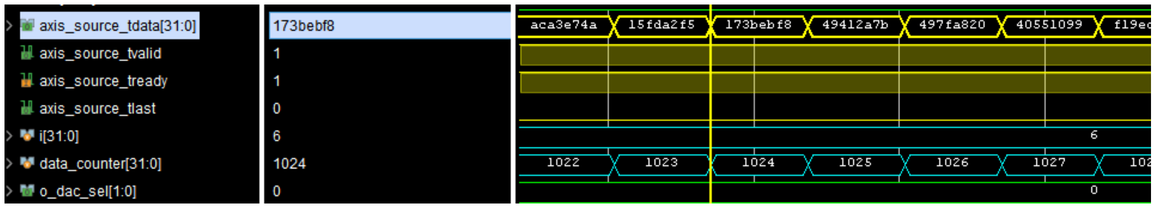
```
DUT Downstream side
mux_sel                     0
gated_mux_m_axis_tready     1
gated_mux_m_axis_tvalid     1
fifo0
  m_axis_tdata[31:0]        ab10adf5
  m_axis_tvalid             1
  m_axis_tready             1
  m_axis_tlast              0
fifo1
  m_axis_tdata[31:0]        00000000
  m_axis_tvalid             0
  m_axis_tready             0
  m_axis_tlast              0
DUT Dac Selector
i_dac_sel[1:0]              0
dac_sel[1:0]                0
axis_dac0_tdata[31:0]       ab10adf5
axis_dac0_tvalid            1
axis_dac0_tready            1
axis_dac1_tdata[31:0]       00000000
axis_dac1_tvalid            0
axis_dac1_tready            1
axis_dac2_tdata[31:0]       00000000
axis_dac2_tvalid            0
axis_dac2_tready            1
```

4. Complete, uninterrupted stream:
   To ensure the stream is uninterrupted, a manual process is performed. I select the tvalid waveform, and select "Next Transition". If the next transition is not at the end of stream, the module is faulty. In this check, the next transition was at the end of stream, indicating correct operation. Note that the next stream starts (orange arrow) with the correct starting bytes from file *01.bin*.



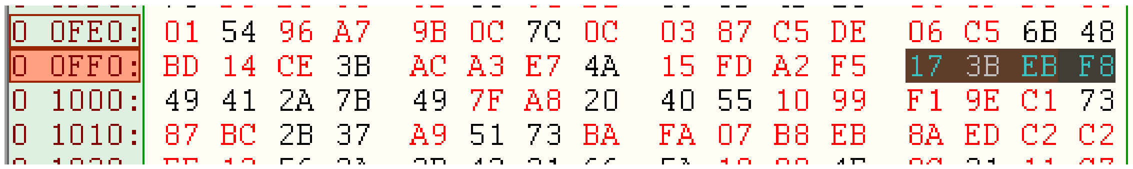
```
DUT Downstream side
  mux_sel                        1
  gated_mux_m_axis_tready        0
  gated_mux_m_axis_tvalid        0
fifo0
  m_axis_tdata[31:0]             1a51aec6
  m_axis_tvalid                  0
  m_axis_tready                  0
  m_axis_tlast                   1
fifo1
  m_axis_tdata[31:0]             b07b7826
  m_axis_tvalid                  1
  m_axis_tready                  0
  m_axis_tlast                   0
DUT Dac Selector
  i_dac_sel[1:0]                 0
  dac_sel[1:0]                   1
  axis_dac0_tdata[31:0]          00000000
  axis_dac0_tvalid               0
  axis_dac0_tready               1
  axis_dac1_tdata[31:0]          b07b7826
  axis_dac1_tvalid               0
  axis_dac1_tready               1
  axis_dac2_tdata[31:0]          00000000
  axis_dac2_tvalid               0
  axis_dac2_tready               1
```

5. Repeat for all streams
The steps above are repeated for all streams. I introduced variable sizes, and manually changed i_dac_sel to see if the correct dac is output. This covers the baseline and edge cases of how the modulator may behave.
6. Random samples
I performed numerous versions of this step: Find a random value in the input stream, map out the data counter, then open the hex file and validate its value and position. Then find the same value in the output stream.
For example, in *bin.05* exists the value `0x173bebf8` at position 1024 :



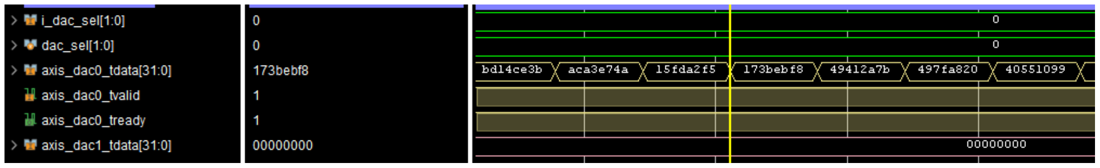
```
axis_source_tdata[31:0]    173bebf8
axis_source_tvalid         1
axis_source_tready         1
axis_source_tlast          0
i[31:0]                    6
data_counter[31:0]         1024
o_dac_sel[1:0]             0
```

By opening the actual hex file, we can find it (and its correct neighboring values) here:



```
0 0FE0: 01 54 96 A7  9B 0C 7C 0C  03 87 C5 DE  06 C5 6B 48
0 0FF0: BD 14 CE 3B  AC A3 E7 4A  15 FD A2 F5  17 3B EB F8
0 1000: 49 41 2A 7B  49 7F A8 20  40 55 10 99  F1 9E C1 73
0 1010: 87 BC 2B 37  A9 51 73 BA  FA 07 B8 EB  8A ED C2 C2
```

And we can find the data (and its correct neighboring values) on the correct DAC stream, at the correct position.



```
i_dac_sel[1:0]           0
dac_sel[1:0]             0
axis_dac0_tdata[31:0]    173bebf8
axis_dac0_tvalid         1
axis_dac0_tready         1
axis_dac1_tdata[31:0]    00000000
```

This is how I performed manual inspection of my module in lieu of a monitor.

Test Bench Qualms

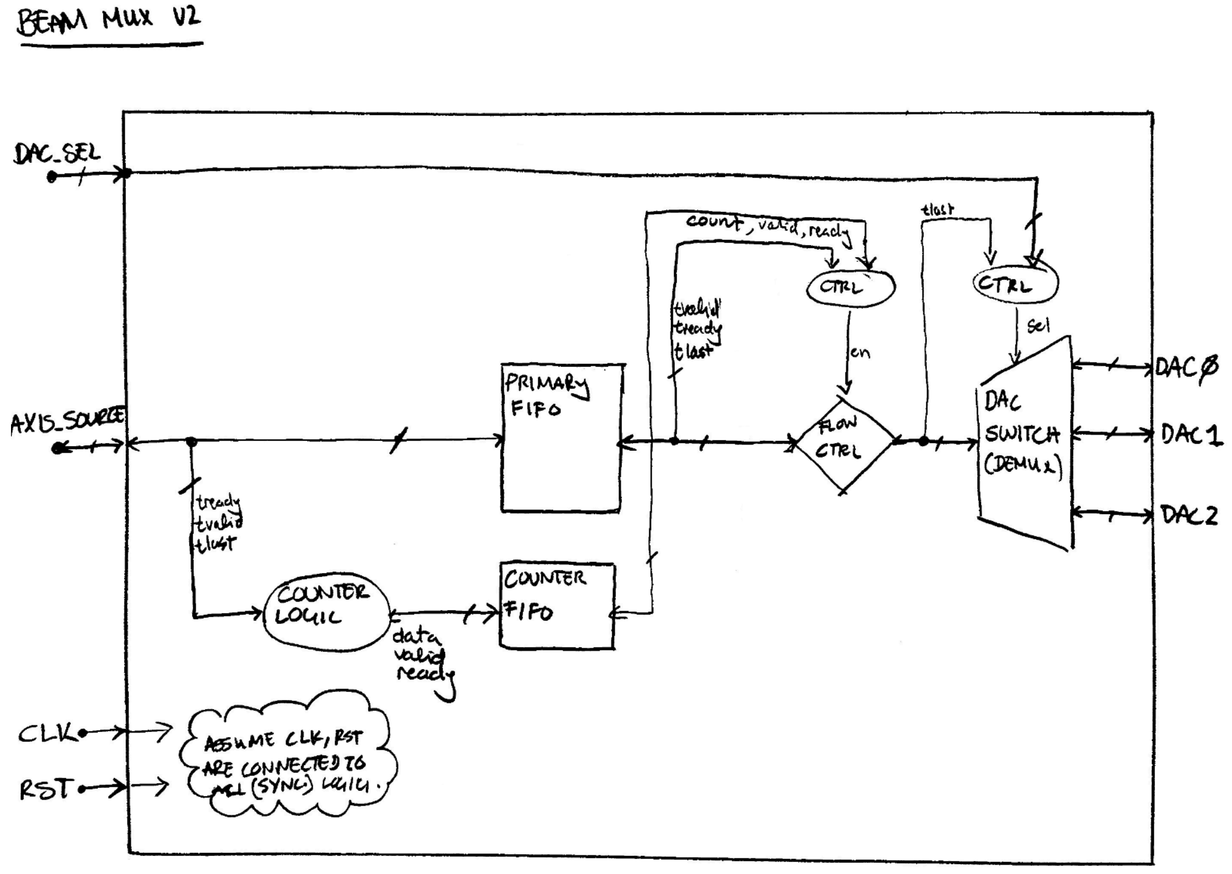
If this were a real work assignment….
… the test bench shall also be developed further into maturity, with an automated monitor to validate the DAC streams. I will make all the input binary files contain 65536 samples then use the Stim module to regulate how many samples to write (adjustable sample size). I will have the DAC monitor use fwrite to write the streams into an output .bin file, and compare it to the input, while checking that tvalid from the Beam Mux is always asserted. Latency between the first sample written to the module to the last sample read from the module, per stream, is another metric required to measure.

Previous Design Considerations

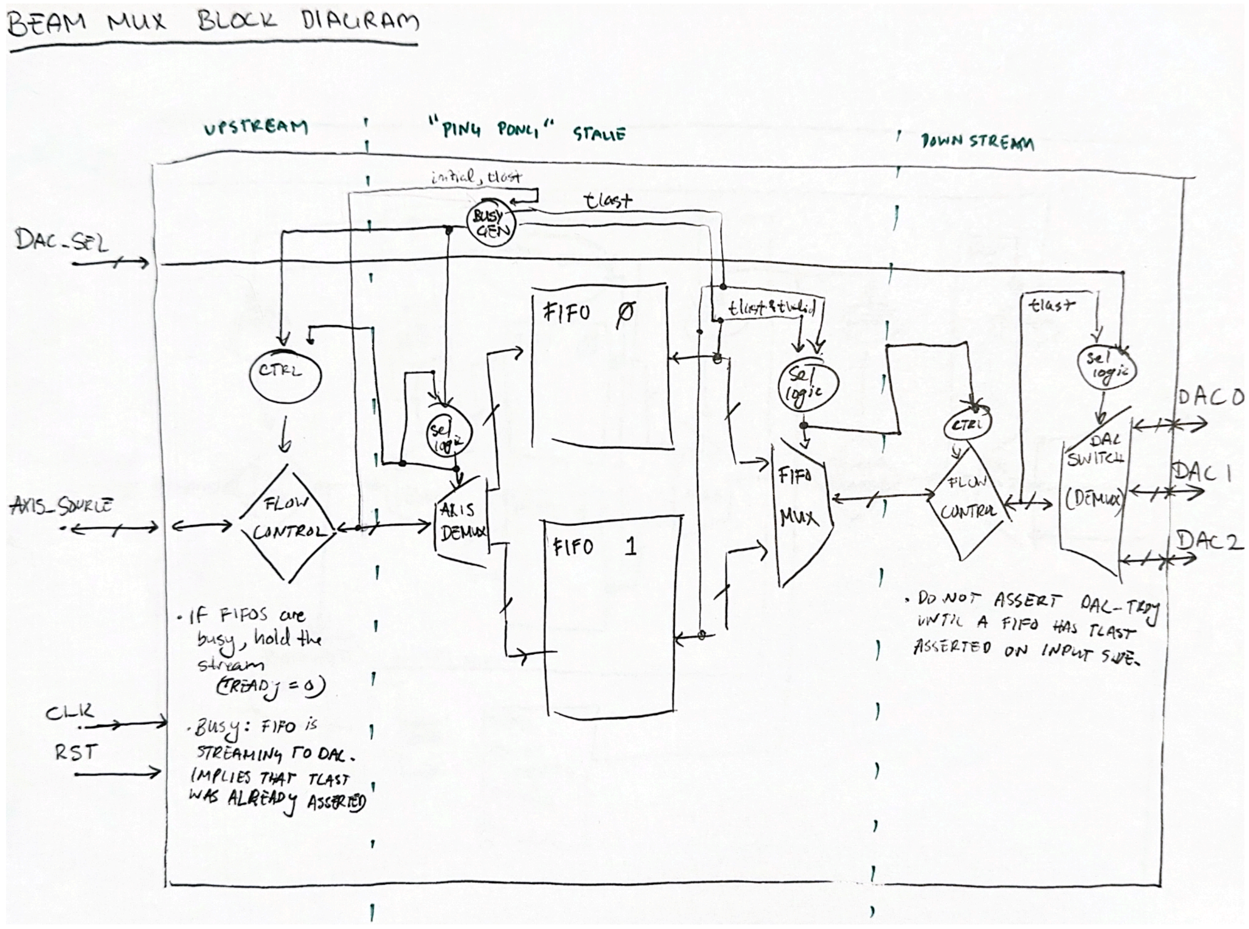
This section is more of an archive to showcase half baked designs I considered while brainstorming.

Beam Mux V2
Similar to the implemented design, except it has a 2nd FIFO to store the size of each burst. I realized storing the burst counts in a FIFO is redundant because the XPM AXI-S FIFO has the flow control required to properly input streams and output when the downstream is ready. Burst counts are not needed; an active counter (as implemented in V3) is all thats needed.

Beam Mux Block Diagram V2.pdf



Beam Mux V1
An overly complicated design using ping-pong buffers, complicated select logic, and extra flow control on the upstream side of the module.



Conclusion

Overall this assignment was fun to code. If this were a real work assignment, extra focus would go towards maturing the test bench suite with a proper monitor and stim module.

~michael r honar