# DDC Multiplexer IP

# Concept

An alternative approach has been conceived which would multiplex all data streams from all enabled DDCs into a single FIFO regardless of their sample rate. A custom multiplexer would be used. It works on the basis that it transfers data per fixed duration “beat” with a starting point for discussion being one beat = 1 48KHz sample period. Within a beat, each DDC transfers N samples depending on sample rate (48KHz = 1 sample, 192KHz = 4 samples etc) and each enabled DDC transfer samples in DDC order. It always starts at DDC0. Non-enabled DDC have the correct number of samples read from their stream.

The multiplexer would have 48 bit Axi stream inputs from DDC, A 48 bit AXI stream output, an enable input and 3 bit codes for each DDC to set the sample rate. When enabled it would start from DDC0 and go through to DDC9 in turn; and read out N samples where N is the number that the DDC generates in a beat. If enable has been cleared it would then stop, otherwise repeat.

For this to work it must be possible to reliably start and stop operation and reconfigure the DDCs and still get deterministic data. The required criterion is: the phase difference between any pair of DDCs used for Puresignal (on TX) or diversity (on receive) must remain constant even if the sample rate is changed[[1]](#footnote-1). To ensure this, the DDC sample rates for any pair of DDCs than needs to be phase related must be changed at the same clock cycle.

# DDC Operation

So if we have DDC0 & 2 enabled at 96KHz, DDC 1 enabled at 192KHz and DDC5 enabled at 48 KHz a single beat will result in sequential outputs:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DDC** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| Enabled | Y | Y | Y | N | N | Y | N | N | N | N |
| Fs (KHz) | 96 | 192 | 96 |  |  | 48 |  |  |  |  |
| Data beat 0 | DDC0n DDC0n+1 | DDC1m DDC1m+1 DDC1m+2 DDC1m+3 | DDC2p DDC2p+1 |  |  | DDC5q |  |  |  |  |
| Data beat 1 | DDC0n+2 DDC0n+3 | DDC1m+4 DDC1m+5 DDC1m+6 DDC1m+7 | DDC2p+2 DDC2p+3 |  |  | DDC5q+1 |  |  |  |  |
| Data beat 2 | DDC0n+4 DDC0n+5 | DDC1m+8 DDC1m+9 DDC1m+10 DDC1m+11 | DDC2p+4 DDC2p+5 |  |  | DDC5q+2 |  |  |  |  |

This leads to an output stream as follows:

DDC0n DDC0n+1 DDC1m DDC1m+1 DDC1m+2 DDC1m+3 DDC2p DDC2p+1 DDC5q DDC0n+2 DDC0n+3 DDC1m+4 DDC1m+5 DDC1m+6 DDC1m+6 DDC2p+2 DDC2p+3 DDC5q+1 DDC0n+4 DDC0n+5 DDC1m+8 DDC1m+9 DDC1m+10 DDC1m+11 DDC2p+4 DDC2p+5 DDC5q+1 and so on

The stream is deterministic, as long as no beats are lost. So the Raspberry pi software will always be able to find the correct data.

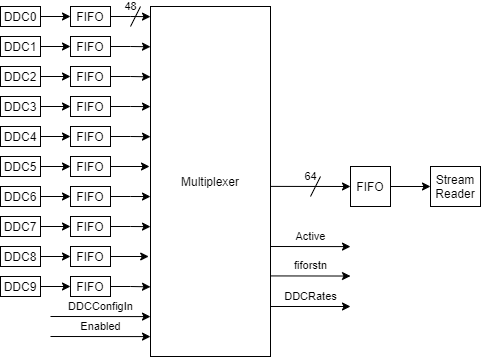


Figure : Alternative DDC to CPU Connection Concept

For the design to work the appropriate number of samples per beat must already be available, requiring a FIFO between DDC and multiplexer. This FIFO needs simply to hold one “data beat” of samples, eg 64 samples. The output FIFO has the largest data rate; if it fills, then the input FIFOs will begin to fill (because it is a stream).

It is suggested that for non enabled DDC, the multiplexer consumes data so the FIFOs maintain depth alongside the others. All data transfers should be initiated or stopped by setting a single “Active” signal.

The multiplexer isn’t synchronised to time; it is data driven. It works out a “beat” by counting the right number of samples from each source. At the end of a beat, it will have read in the required number of samples for each DDC and also read them out (or be a few clock cycles from that) – there is no persistent storage.

So that the Raspberry pi processor will know the stream configuration: it will transfer its configuration once per beat into the output stream. It uses the to 16 bits to indicate its word in the overall sequence. Again it is deterministic, so the processor will always know where it is in the stream.

The multiplexer will transfer 48 bit (6 byte) words to a 64 bit FIFO. The additional (top) 16 bits of data will be used as follows:

# 4 bits: signal “changed config” to processor. If 0; data is IQ; if 1, data is a new set of DDC configurations.

# 4 bits: specify the DDC the data is from (for debug)

* 8 bits: specify the sample number being transferred (0 to 31, for debug)

Within the 48 bit data the IP could in principle include programmable byte swapping, to network byte order; but not currently implemented.

This way the DMA process is continuous. It is data driven, and self locates the points where data format changes.

# DMA Transfer & Processor Operation

With many DDC enabled and high sample rates the output rate could get quite high. BUT the DMA transfer chunks could be larger, and only one DMA needs to be operated to service all DDC. Overall it is likely more efficient in terms of CPU & bus utilisation. Processor loading is implied for P1 and P2 transfers to Thetis, because data needs to be reordered and packed; but the processor has little to do.

In this design the multiplexer will transfer a DDC configuration word, then loop through all enabled DDC and transfer the required number of samples for one data beat. After processing DDC0-9 it will repeat. This way once started the data transfer never needs be stopped, and the input FIFOs can all be small (eg 64 samples). It does mean that if it ever is stopped, data will be lost and the input FIFOs (and probably output FIFO) should be reset.

* To start up:
  + Setup DDC words
  + Enable multiplexer
  + Start DMA transfer
* To make changes:
  + Change DDC configuration
  + The IP will read and output the configuration before processing all current DDCs
* If an error is detected:
  + Stop multiplexer
    - Multiplexer completes the current set of DDC reads up to & including DDC9
  + DMA out all data from FIFO (or just delete)
  + Reset DDC o/p FIFOs
  + Enable multiplexer
  + Re-start DMA transfer

A spreadsheet (“DDC output FIFO depth calculator.xlsx”) takes DDC configuration to predict output data rate and FIFO capacity in milliseconds. It shows that a 4kx48 FIFO would have a duration of 0.88ms if DDC0,1 at 1536KHz; DDC 2,3 at 192KHz and others all at 192KHz. Seems reasonable.

This does mean that the configuration is sent each beat. That’s a 100% overhead with one DDC enabled at 48KHz, but the processor has little to do in this case. But the overhead gets much smaller as the number of DDCs and/or the sample rates are increased; it also means the PC doesn’t need to search for the word – it will always be in a known location AND it has bits set to identify it for error detection.

It would also be possible to only transfer the DDC config at the start, and when it changes. Then the RaspberryPi would need to check for a new config word.

# Example data

With DDC0&1 = 192KHz, DDC2=384KHz, DDC3=48KHz and DDC7=96KHz the output stream for each data “beat” (20.8us) will be (using randomised data):

|  |  |
| --- | --- |
| **output data for a single beat** | |
| 100000001B83FFFF | header |
| 0000EFB8165E22E9 | DDC0 |
| 0001E2CC5C33D1A2 |  |
| 000226BB315B49EA |  |
| 00035148B9A13644 |  |
| 01001ED999A9B06C | DDC1 |
| 01017C83BB2AF529 |  |
| 0102288B1E95D38D |  |
| 01038919B9CCEF73 |  |
| 02006C9AB7EC5211 | DDC2 |
| 02010339A05855F7 |  |
| 0202B5444920AF82 |  |
| 0203F3F9C2FE4E14 |  |
| 0204B7D4BF1EEBA4 |  |
| 0205924C9D9162C3 |  |
| 02066BE5A771F08F |  |
| 0207F4189ECCBE4A |  |
| 0300AC6D0D9E2627 | DDC3 |
| 07006878189F1D42 | DDC7 |
| 07012A6C332867B2 |  |

In this case, one config word per beat adds a 5% overhead.

# Requirements

## Inputs

* 10 AXI streams (each 48 bit) with READY driven by multiplexer
* Enabled input (active operation initiated while set to 1)
* DDC config input (10 x 3 bits)
* Each DDC config input:
  + 0: disabled (operates at 48KHz);
  + 1: 48KHz; 1 word per beat
  + 2: 96KHz; 2 words per beat
  + 3: 192KHz; 4 words per beat
  + 4: 384 KHz; 8 words per beat
  + 5: 768 KHz; 16 words per beat
  + 6: 1536 KHz; 32 words per beat
  + 7: interleaved with next DDC; used rate from DDC n+1

## Outputs

* AXI stream output (64 bits)
  + 48 LSB = IQ data
  + (63:60) DDC Config word
    - =0: DDC I/Q data
    - =1: DDC config data in bits (29:0)
  + (59:56) input DDC number
  + (55:48) sample number in beat (0 to 31)
* “Active” strobe: =1 if active
* fiforstn: active low FIFO reset output, asserted at start of active period for ~8 clocks

## Function

### Enabled Sequencer

* When “Enabled” changed to asserted:
  + Assert fiforstn for 8 clocks
  + Assert “Active”
  + Assert “InternalActive”
* When “Enabled” changed to deasserted:
  + Deassert “InternalActve”
  + Wait until all DDCState == 0 (DDCs have been serviced)
  + Deassert “Active”



Figure : Enable Sequencer

### DDC Select Sequencer

* When “InternalActive” is changed to asserted:
  + Set DDCState to 1
  + Copy DDC settings to output
  + Initiate Stream Sequencer to Transmit a DDC config word to the output FIFO by setting DDCx=15
  + For DDCx= 0 to 9
    - Initiate Stream sequencer to transfer samples from DDC number DDCx
    - If DDC enabled: Read out the required number of data bytes to output FIFO
    - If DDC disabled: read out required number of samples and discard
    - Wait till MuxActive == 0
  + At end of DDCs:
    - Check if InternalActive asserted
      * If deasserted, revert to idle
      * Else continue at state 1



Figure DDC select sequencer - start of cycle



Figure : DDC Select Sequencer - continued

### Stream Sequencer

* If Reset:
  + Clear input tready signals
  + Clear output tvalid
* If EnableMux == 0, stay in idle
* If EnableMux
  + Set MuxActive
  + If (DDCx == 15):
    - Enter state 1
    - Set top 16 bits for DDC config word
    - Drive 64 bit output word
    - Assert tvalid
    - if (tvalid && tready):
      * Deassert tvalid
      * Clear MuxActive
      * Go to wait state 6
        + If (enablemux==0), go to idle
  + If DDCx == 0 to 9:
    - Enter state 2
    - DDCToProcess = DDCx
    - Read DDC config (3 bits)
    - Look up TransferCount
    - For SampleCount == (TransferCount-1) to 0
      * (If DDC set to interleaved: read out this and next DDC, and transfer to output FIFO)
      * Assert input tready
      * When tready && tvalid:
        + Read input word to output word, top 16 bits=0000
        + Deassert tready
        + If DDC is enabled:

Assert tvalid

When tvald && tready

Deassert tvalid

(end of transfer)

* + - Clear MuxActive
    - Go to wait state 6
      * If enablemux==0, go to idle



Figure Multiplexer operation - DDC settings



Figure Multiplexer operation – DDC, count = 2

# Implementation

Coded in Verilog using three discrete “always@” blocks

* “enabled” sequencer
* “DDC select” sequencer
* AXI stream sequencer

1. This is because Thetis, today, will reconfigure DDC0 & 1 between RX and TX because only DDC0&1 are paired & interleaved. A change to Thetis explicitly to support Saturn would stop this being an issue [↑](#footnote-ref-1)