OptoHybrid v3 Firmware

Andrew Peck ([andrew.peck@cern.ch](mailto:andrew.peck@cern.ch)) UCLA  
Evaldas Juska ([evaldas.juska@cern.ch](mailto:evaldas.juska@cern.ch)) TAMU   
Thomas Lenzi ([thomas.lenzi@cern.ch](mailto:thomas.lenzi@cern.ch)) ULB

January 18, 2018

# Contents

[Contents 1](#_Toc503871684)

[Firmware Architecture 4](#_Toc503871685)

[Registers 5](#_Toc503871686)

[Clocking 6](#_Toc503871687)

[Fabric Clocks 6](#_Toc503871688)

[GBT RX Clock Phase 7](#_Toc503871689)

[GBT TX Clock Phase 7](#_Toc503871690)

[Synchronization 8](#_Toc503871691)

[TTC Monitoring 8](#_Toc503871692)

[Trigger Link Frame Sequencing 8](#_Toc503871693)

[GBTx Deserialization 10](#_Toc503871694)

[GBT E-Link Decoding 11](#_Toc503871695)

[In “16 bit” full-bandwidth mode 11](#_Toc503871696)

[In “10 bit” low-bandwidth mode 11](#_Toc503871697)

[GBT Packets (Full Bandwdith Mode) 12](#_Toc503871698)

[Transmit (FPGA to GBT) 12](#_Toc503871699)

[Receive (GBT to FPGA) 14](#_Toc503871700)

[GBT Packets in Low Bandwidth Mode 15](#_Toc503871701)

[Transmit (FPGA to GBT) 15](#_Toc503871702)

[Receive (GBT to FPGA) 16](#_Toc503871703)

[S-bit Deserialization 17](#_Toc503871704)

[Shared Sampling State Machines 17](#_Toc503871705)

[Interleaving and Frame Alignment 17](#_Toc503871706)

[S-bit Timing 19](#_Toc503871707)

[Trigger Primitive Compression Algorithm 20](#_Toc503871708)

[Light Cluster Packing Algorithm 20](#_Toc503871709)

[Sequential Description of the Cluster Packing Algorithm 21](#_Toc503871710)

[Clock 1 21](#_Toc503871711)

[Clock 2 21](#_Toc503871712)

[Clock 3 21](#_Toc503871713)

[Clock 4 22](#_Toc503871714)

[Clock 5 23](#_Toc503871715)

[Clock 6 23](#_Toc503871716)

[Clock 7 23](#_Toc503871717)

[Clock 8 23](#_Toc503871718)

[Clock 9 24](#_Toc503871719)

[Clock 10 24](#_Toc503871720)

[Clock 11 24](#_Toc503871721)

[Clock 12 25](#_Toc503871722)

[Clock 13 25](#_Toc503871723)

[Clock 14 25](#_Toc503871724)

[Clock 15 25](#_Toc503871725)

[Clock 16 25](#_Toc503871726)

[Clock 17 26](#_Toc503871727)

[Cluster Packer Miscellany 27](#_Toc503871728)

[Cluster Counter 27](#_Toc503871729)

[LED Indicators 28](#_Toc503871730)

[HDMI Output 29](#_Toc503871731)

[Scripts 30](#_Toc503871732)

[genproms.{sh/bat} 30](#_Toc503871733)

[generate\_ucf.py 30](#_Toc503871734)

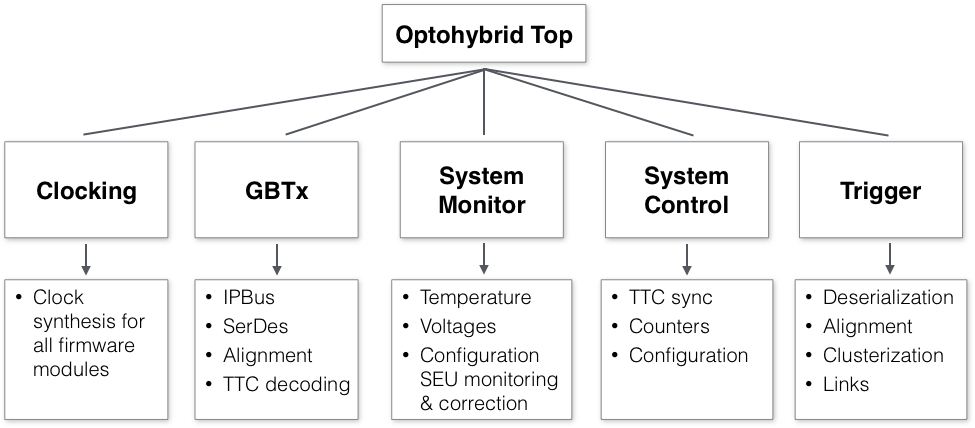
[generate\_inversions.py 30](#_Toc503871735)

[generate\_registers.py 30](#_Toc503871736)

[generate\_timing\_registers.py 31](#_Toc503871737)

[History 32](#_Toc503871738)

# Firmware Architecture



The architecture of the optohybrid firmware is summarized in the block diagram above.

All modules are controlled through a common IPBus-based system, wherein requests received from the GBTx links are translated into IPBus read/write requests, which are distributed to the slaves in each module.

# Registers

In the version 3 firmware, all Optohybrid registers are accessible through IPBus and defined by a single XML file optohybrid\_registers.xml (accessible in Github at <https://github.com/andrewpeck/OptoHybridv3/blob/master/optohybrid_registers.xml)>:

This file can be used in the same way as the backend (CTP-7) registers file and with the same tools, e.g. rw\_reg.py

The contents of this file should be embedded into the CTP-7 firmware at a sub-address which is allocated to the Optohybrid.

Within the Optohybrid firmware, only 16 bits are allocated for the Optohybrid address space (n.b. that this is reduced from the full 32 bit space of standard IPBus---you can consider the Optohybrid as existing in a sub-address of the CTP-7).

Within this 16-bit address space, the bits are subdivided according to the scheme:

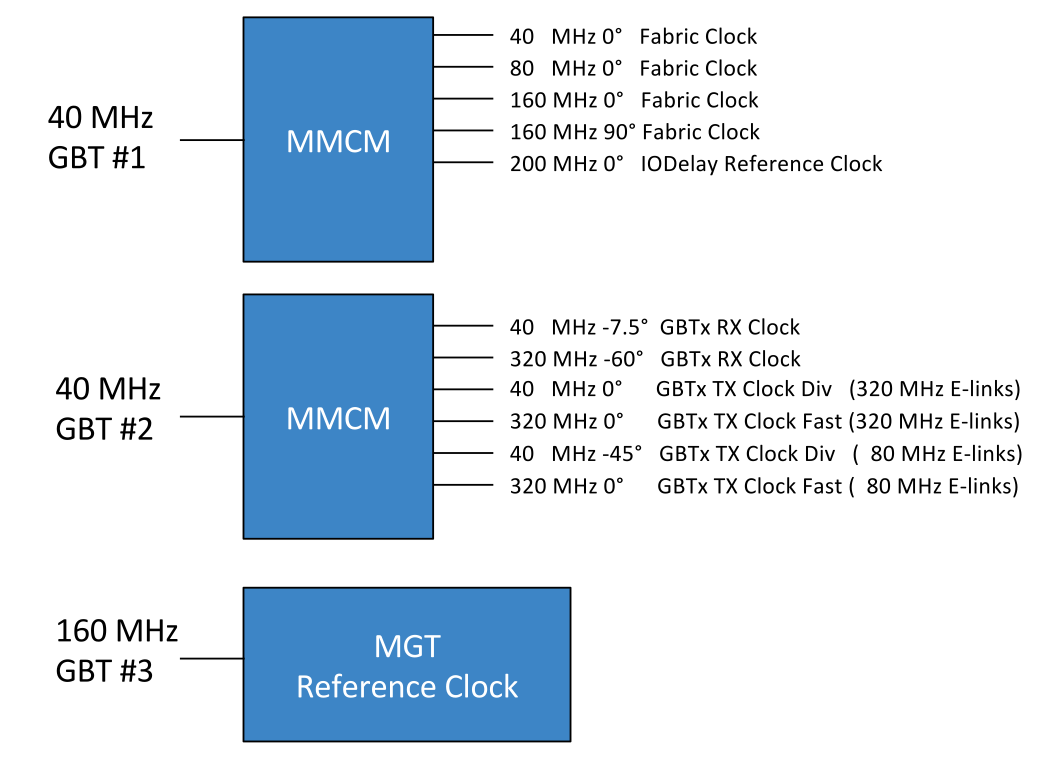
-- highest 4 are used as the module ID (wishbone slave #)

-- lowest 12 bits are used by the wishbone splitters as individual register addresses

Thus we can have up to 16 slaves, and 4096 endpoints per slave. Presently we use only 5 slaves with a firmware that is nearly finalized, so it is expected that 16 slaves will not be a limiting factor.

# Clocking

Basic clocking structure of the Optohybid is as shown in the following figure.



### Fabric Clocks

* 40MHz fabric clock is the primary clock for the logic in the Optohybrid firmware. It is used to drive nearly all control and logic, aside from fast elements of the trigger logic.
* 80MHz fabric clock is used to drive the USERCLK2 of the trigger link multi-gigabit transceivers.
* 160 Mhz 0 degree clock is used for all of the fast trigger logic in the cluster packer.

The same clock is also used, along with its 90 degree phase shifted sister clock, to deserialize the S-bits in the ISERDES inputs of the Virtex-6 FPGA.

* The 200 degree IODELAY reference clock is required by the IODELAY elements of the FPGA to calibrate the delay taps to the correct timing. Please consult Xilinx documentation for more details.

### GBT RX Clock Phase

Note the phase shift of the GBT TX Clock relative to the RX Clock. The shift is, in the 40MHz domain, equal to -7.5 degrees, or -0.52 ns. While it is a seemingly arbitrary number, it has an understood physical basis on the Optohybrid PCB.

Keep in mind that the GBT clocks are phase shiftable, so, excluding all other factors, we should be able to phase shift this clock such that either the TX or RX side of the FPGA<->GBTx link is correctly timed in.

But now consider that there is a physical delay, based on the routing across the PCB, that will introduce an additional phase shift which is different between TX and RX. The 7.5 degree (0.52 ns) delay is equal to, on a standard FR4 board, approximately 3 inches of trace length.

This is approximately the distance of the return line from the FPGA to the GBTx—the negative delay on the RX clock is introduced so that data at the GBTx chip is synchronous to the sampling clock despite the additional delay introduced by the RX line.

It is intended that eventually this delay would be achieved through an IODELAY (of +0.52 ns) applied to the GBTx🡪FPGA line, to achieve the same effect and slightly reduce clocking resources used.

### GBT TX Clock Phase

There is an additional phase delay applied to the 40MHz TX clock for the 80MHz elink. In this case, it is -45 degrees, or 3.125 ns, which is equal to exactly one 320 MHz clock.

This is effectively a bitslip of 1 bit to align the frame clock correctly to the data serializer.

It is intended that we would remove this and instead use logic in the FPGA to achieve this same effect, but at the moment it is still done through a phase shifted clock.

Note that there is an interesting caveat to the GBT 40 and 320 MHz clocks.

# Synchronization

The primary purpose of the Optohybrid firmware is its role in the firmware. Integral to this role is the ability to synchronize each Optohybrid to a known phase.

Two provisions are implemented in the Optohybrid firmware to aid this.

### TTC Monitoring

The Optohybrid has a local accumulator which increments bunch crossing number (bxn) and produces a locally generated bx0 flag.

In order for the two flags, (1) locally generated and (2) received from GBTx, to be in sync a programmable parameter (bxn offset) is provided which allows you to set an “offset” which is the value which will be assumed by the bxn counter after ttc resync.

The synchronization of these two flags can be monitored by checking the status of bxn\_sync\_error in the OH status register (*n.b. that bx0\_sync\_error is NOT suitable for this, as it is a 1bx wide pulse useful for internal counters when the bx0 is received, while bxn\_sync\_error will persist for the entire orbit*).

When the Optohybrid is correctly timed in to the TTC system, bxn\_sync\_error should be 0. A software routine in the CTP-7 to find the correct bxn offset should be easily designed.

A counter in the counters module counts the number of bx0 sync errors since the last resync or hard-reset.

### Trigger Link Frame Sequencing

In addition to the 56 bits of trigger data that are sent every bunch crossing, each fiber link also transmits an 8 bit frame delimiter which is used to align the frames into packets.

There are several frame markers which can be chosen, and we exploit this to provide information about the Optohybrid which will aid in synchronization.

During normal operation, the frame will follow a predictable cycle of **0xBC, 0xF7, 0xFB, 0xFD**.

The sequencing of this is derived from a local TTC BXN counter, which should allow for even greater ability to synchronize the different optohybrids.

In this simplest form we can ensure crude synchronization between links and different chambers as well as monitoring the data integrity, by comparing the sequence to a local state machine in the backend electronics.

Two additional separators are used to mark special conditions:

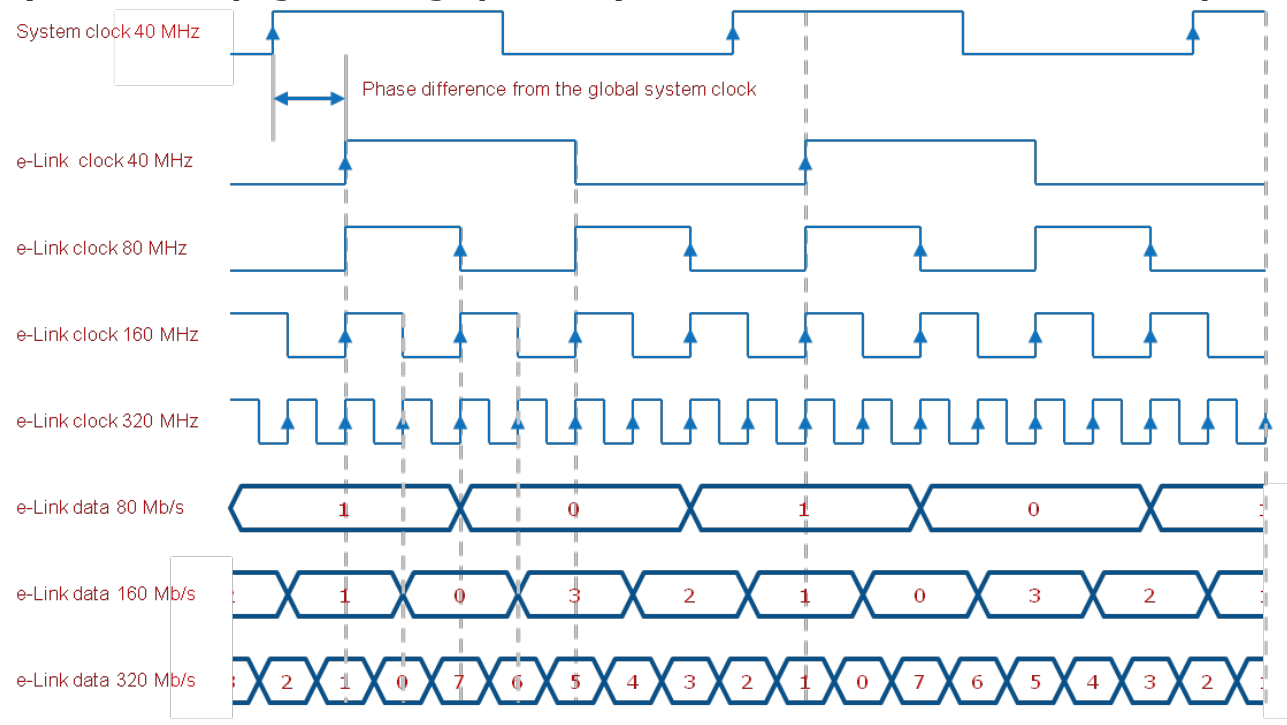
1. **BC0:** the bunch crossing zero flag received from the GBTx is connected to the trigger link logic. When a BC0 (or equivalently BX0) is received the frame marker will be changed to 0x50.
2. **Overflow:** the trigger links are only capable of transmitting 8 clusters per bunch crossing. Hence there is a possibility for overflows to occur, wherein clusters are found by not transmitted. When this happens, the trigger link will change its frame separator to 0xFC.

The anticipated synchronization process is:

1. From the backend electronics, ensure that BC0 is being sent to all Optohybrids
2. Inside of the Optohybrid, adjust the BXN offset until bxn\_sync\_error is read to be 0
   1. This indicates that the local OH bxn counter matches the remote counter
3. Inside of the backend electronics, adjust the delays of incoming trigger data until the 0x50 (BC0) frame markers are aligned between all Optohybrids.
   1. For sanity, also ensure that the sequence of BC, F7, FB, FD is synchronized between chambers

# GBTx Deserialization

Timing of the elinks is described in the GBTx Manual, and copied here for reference:



# GBT E-Link Decoding

With the existence of two different communication modes corresponding yielding 16 bits/bx and 10 bits/bx (duplex) there will be two alternative assignments of which bits will be carried on which e-links.

In “16 bit” full-bandwidth mode:

We specify that one eLink (labelled in firmware as eLink0) be run at 320 MHz and carry the 8 least-significant bits of the packet. The other eLink (labelled in firmware as eLink1) will run at 320 MHz and carry the 8 most-significant bits.

i.e.

|  |  |  |
| --- | --- | --- |
|  | **E-link** | **Payload** |
| RX  (GBT to OH) | eLink1 | TTC[3:0],  data[11:8] |
| eLink0 | data[7:0] |
| TX  (OH to GBT) | eLink1 | data[15:8] |
| eLink0 | data[ 7:0] |

In “10 bit” low-bandwidth mode:

We specify that one eLink (labelled in firmware as eLink0) be run at 80 MHz and carry the 2 least-significant bits of the packet. The other eLink (labelled in firmware as eLink1) will run at 320 MHz and carry the 8 most-significant bits.

i.e.

|  |  |  |
| --- | --- | --- |
|  | **E-link** | **Payload** |
| RX  (GBT to OH) | eLink1 | TTC[3:0],  data[5:2] |
| eLink0 | data[1:0] |
| TX  (OH to GBT) | eLink1 | data[9:2] |
| eLink0 | data[1:0] |

Within each e-link, all bits are transmitted and most-significant bit first.

# GBT Packets (Full Bandwdith Mode)

In “Full Bandwidth Mode” the GBT to FPGA link consists of two 320 Mbps duplex links. In version 3a this was not implemented, and a separate functionality of 1x 320 Mbps duplex and 1x 80Mbps duplex was adopted. This alternative, “Low Bandwidth Mode” is covered in the next section.

### Transmit (FPGA to GBT)

Communication with the optohybrid in the direction of FPGA to GBT is accomplished with a simple 16 bit packet format.

A valid packet is three frames long and will consist of the 16 bit frame marker BCBC followed by two 16 bit data frames:

|  |  |
| --- | --- |
| **Frame** | **frame[15:0]** |
| Start | 0xBCBC |
| Data0 | data[31:16] |
| Data1 | data[15: 0] |

For example, this waveform shows the transmission of the 32-bit packet 0x12345678:



Consecutive packets will simply repeat this sequence, e.g:

|  |  |
| --- | --- |
| **Frame** | **frame[15:0]** |
| Start | 0xBCBC |
| Data0 | data[31:16] |
| Data1 | data[15: 0] |
| Start | 0xBCBC |
| Data0 | data[31:16] |
| Data1 | data[15: 0] |

When the transmit FIFO is empty and there is no data to send, the state machine will send an idle frame marker, 0xFCFC:

|  |  |
| --- | --- |
| **Frame** | **frame[15:0]** |
| Start | 0xBCBC |
| Data0 | data[31:16] |
| Data1 | data[15: 0] |
| IDLE | 0xFCFC |
| IDLE | 0xFCFC |
| IDLE | 0xFCFC |
| IDLE | 0xFCFC |
| Start | 0xBCBC |
| Data0 | data[31:16] |
| Data1 | data[15: 0] |

For example, this waveform shows the transmission of the 32-bit packet 0x12345678, followed by the idle transmission of the 0xFCFC frame marker.



The receiving state machine should monitor for 0xBCBC for alignment and record the next two frames as the 32 bit data packet.

If the following frame is 0xBCBC, then the sequence should repeat and record the following two frames, and so on. If the frame is 0xFCFC then the state machine should wait until it sees a start-of-packet marker (0xBCBC).

### Receive (GBT to FPGA)

Communication in the direction of GBT to FPGA is similarly accomplished with a packet format, whose payload includes 16 bits of address and 32 bits of data. Transmit and receive bandwidths are asymmetric, due to the requirement of transmitting synchronous TTC commands on the GBT to FPGA link, so the format on the receive side is somewhat more complicated.

In this direction, the highest 4 bits of the GBT link are reserved for TTC commands:

* Bit15 = L1A
* Bit14 = VFAT Reset
* Bit13 = TTC Resync
* Bit12 = BC0

The remaining 12 bits are used as the GBT data frame used to construct packets.

|  |  |  |
| --- | --- | --- |
| **Frame** | **Highest 4 bits** | **Lowest 12 bits** |
| Start | L1A + reset + Resync + BC0 | 0xABC |
| Header | L1A + reset + Resync + BC0 | req\_valid + wr\_enable + reserved[5:0] + address[15:12] |
| Address 0 | L1A + reset + Resync + BC0 | address[11:0] |
| Data 0 | L1A + reset + Resync + BC0 | data[31:24] |
| Data 1 | L1A + reset + Resync + BC0 | data[23:12] |
| Data 2 | L1A + reset + Resync + BC0 | data[11: 0] |

For continuous transmission, this sequence should just be repeated cyclically. i.e. the frame after Data 2 should be “Start” (0xABC). E.g. as follows:

**<START> <HEADER> <ADR0><DATA0> <DATA1> <DATA2>, <START> <HEADER> <ADR0> ….**

When idle, there are two possible ways of sending data to the OH:

Either cycle through the 7 frame packet format continuously, making sure to set request\_valid to 0 for any null packets. The Optohybrid will process and discard the packet.

Alternatively (preferably) when can send an idle frame marker, for example 0x5FC. The exact choice does not matter as long as it is not 0xABC. This marker can be sent continuously as long as the transmission is idle and the Optohybrid will remain in an idle SYNC state. The state machine will monitor for 0xABC and if received, will continue to process the rest of the frame normally.

# GBT Packets in Low Bandwidth Mode

In “Low Bandwidth Mode” the GBT to FPGA link consists of 1x 320 Mbps duplex and 1x 80Mbps duplex, leaving 10 bits per bunch crossing in each direction.

### Transmit (FPGA to GBT)

Communication with the optohybrid in the direction of FPGA to GBT is accomplished with a simple 10 bit packet format.

A valid packet is three five long and will consist of the a bit frame marker BC followed by three 10 bit data frames:

|  |  |
| --- | --- |
| **Frame** | **frame[9:0]** |
| Start | 2’b00 & 0xBC |
| Data0 | 2’b00 & data[31:24] |
| Data1 | 2’b01 & data[23:16] |
| Data2 | 2’b10 & data[15: 8] |
| Data3 | 2’b11 & data[ 7: 0] |

For example, this waveform shows the transmission of the 32-bit packet 0x12345678:



Note that the highest two bits of the data frames indicate the numbering of the frame in the packet and can be used for error checking.

Consecutive packets will simply repeat this sequence.

When the transmit FIFO is empty and there is no data to send, the state machine will send an idle frame marker, 0xFC. For example, this waveform shows the transmission of the 32-bit packet 0x12345678, followed by the idle transmission of the 0xFC frame marker, followed by the beginning of a new valid packet.



The receiving state machine should monitor for 0xBC for alignment and record the next four frames as the 32 bit data packet.

If the frame following the last data frame is 0xBC, then the sequence should repeat and record the following two frames, and so on. If the frame is 0xFC (or anything else) then the state machine should wait until it sees a start-of-packet marker (0xBC).

### Receive (GBT to FPGA)

Communication in the direction of GBT to FPGA is similarly accomplished with a packet format, whose payload includes 32 bits of address and 32 bits of data. Transmit and receive bandwidths are asymmetric, due to the requirement of transmitting synchronous TTC commands on the GBT to FPGA link, so the format on the receive side is somewhat more complicated.

In this direction, the highest 4 bits of the GBT link are reserved for TTC commands:

* Bit10 = L1A
* Bit9 = VFAT Reset
* Bit8 = TTC Resync
* Bit7 = BC0

The remaining 6 bits are used as the GBT data frame used to construct packets.

|  |  |  |
| --- | --- | --- |
| **Frame** | **Highest 4 bits** | **Lowest 6 bits** |
| Start | L1A + reset + Resync + BC0 | 0x2A |
| Header | L1A + reset + Resync + BC0 | request\_valid + wr\_enable + reserved[3:0] |
| Address 0 | L1A + reset + Resync + BC0 | address[15:10] |
| Address 1 | L1A + reset + Resync + BC0 | address[ 9: 4] |
| Address 2 | L1A + reset + Resync + BC0 | address[3:0] + data[31:30] |
| Data 0 | L1A + reset + Resync + BC0 | data[29:24] |
| Data 1 | L1A + reset + Resync + BC0 | data[23:18] |
| Data 2 | L1A + reset + Resync + BC0 | data[17:12] |
| Data 3 | L1A + reset + Resync + BC0 | data[11: 6] |
| Data 4 | L1A + reset + Resync + BC0 | data[ 5: 0] |

For continuous transmission, this sequence should just be repeated cyclically. i.e. the frame after Data 4 should be “Start” (0x2A). E.g. as follows:

**<START><HEADER><ADR0><ADR1><ADR2><DATA0><DATA1><DATA2><DATA3><DATA4>, <START>**

#### Idle Frame Marker

When idle (not sending data), the GBT **must** send an idle marker for the FPGA’s decoder to remain in sync and still accept TTC commands. The GBT should send the frame “0x1C” on every clock cycle.

# S-bit Deserialization

Deserialization of S-bits is accomplished with the well-documented Oversampling technique (documented in Xilinx XAPP 881, "Virtex-6 FPGA LVDS 4X Asynchronous Oversampling at 1.25 Gb/s”).

Each VFAT transmission unit is asynchronous to the other, since lengths are not matched. Thus, the data must be reliably sampled in the center of the eye, given an unknown phase relationship between each of the 216 transmission unit pairs that come into the Optohybrid FPGA.

We will not reiterate on the basics of the oversampling technique, which is already well documented, but there are some specific features of the Optohybrid implementation which should be noted.

## Shared Sampling State Machines

By tuning the tap delays of differential pairs within a single VFAT (which are already nearly phase-aligned) we can make the reasonable assumption that they are all aligned well enough that they can be considered in-phase.

Additionally, we should note that because the S-bits are nearly always idle (set to 0 or 1 depending on polarity inversions), there are very infrequent transitions. Because of this, the phase alignment state machine, as described in aforementioned XAPP note, will take a long time to be able to lock onto the data in the correct phase.

Instead, we have a single state machine for each VFAT which finds the optimal sample selection for each VFAT based only on the transitions in the Start-of-Transmission pulse, which is guaranteed every 40MHz. The optical sample selection for the 8 S-bit pairs from that same VFAT are assumed to be the same because of the tap-delay based phase alignment. This has the added benefit of significantly reducing the resource usage required for this step, by reducing the number of state machines required.

## Interleaving and Frame Alignment

The oversampling technique introduces an additional step of interleaving “odd” and “even” (or rising and falling) samples in separate 160 MHz data streams.

A Verilog firmware module, frame\_aligner.v, is responsible for interleaving these odd and even bitstreams into a 64 bit long trigger word.

The alignment of this 64 bit long trigger word is uncertain on its own, which the uncertainty represented as a barrel shift of the S-bit position within the VFAT.

To align this data to the 40MHz sampling clock in the Optohybrid, a state machine applies a programmable delay to the Start-of-Transmission pulse. The Xilinx SRL16E primitive is used to produce a resource-efficient, 0-15 clock cycle delay on the Start-of-Transmission pulse. The delay is incremented until the Start-of-Transmission pulse for that particular VFAT is aligned to the 40MHz clock.

The value of this SRL16E address is also applied to the odd and even data streams, which aligns the data automatically to the 40MHz frame clock.

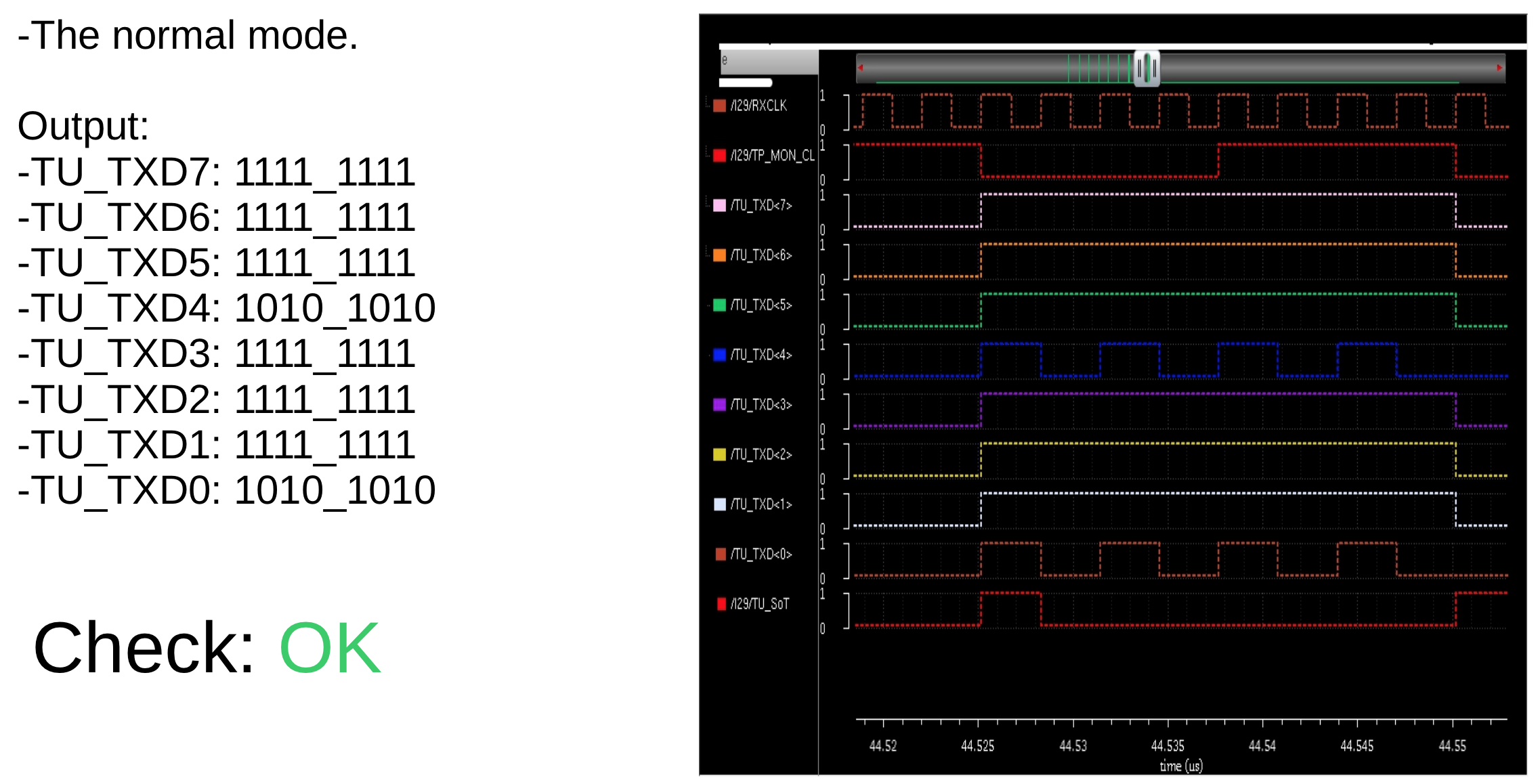
Depending on the clock phase of the 40MHz frame clock, synchronous pulses in the chamber may arrive in different bunch crossings. This should be corrected by adjusting the phase of the 40MHz clock from the GBTx so center the OH inside of a bunch crossing window.

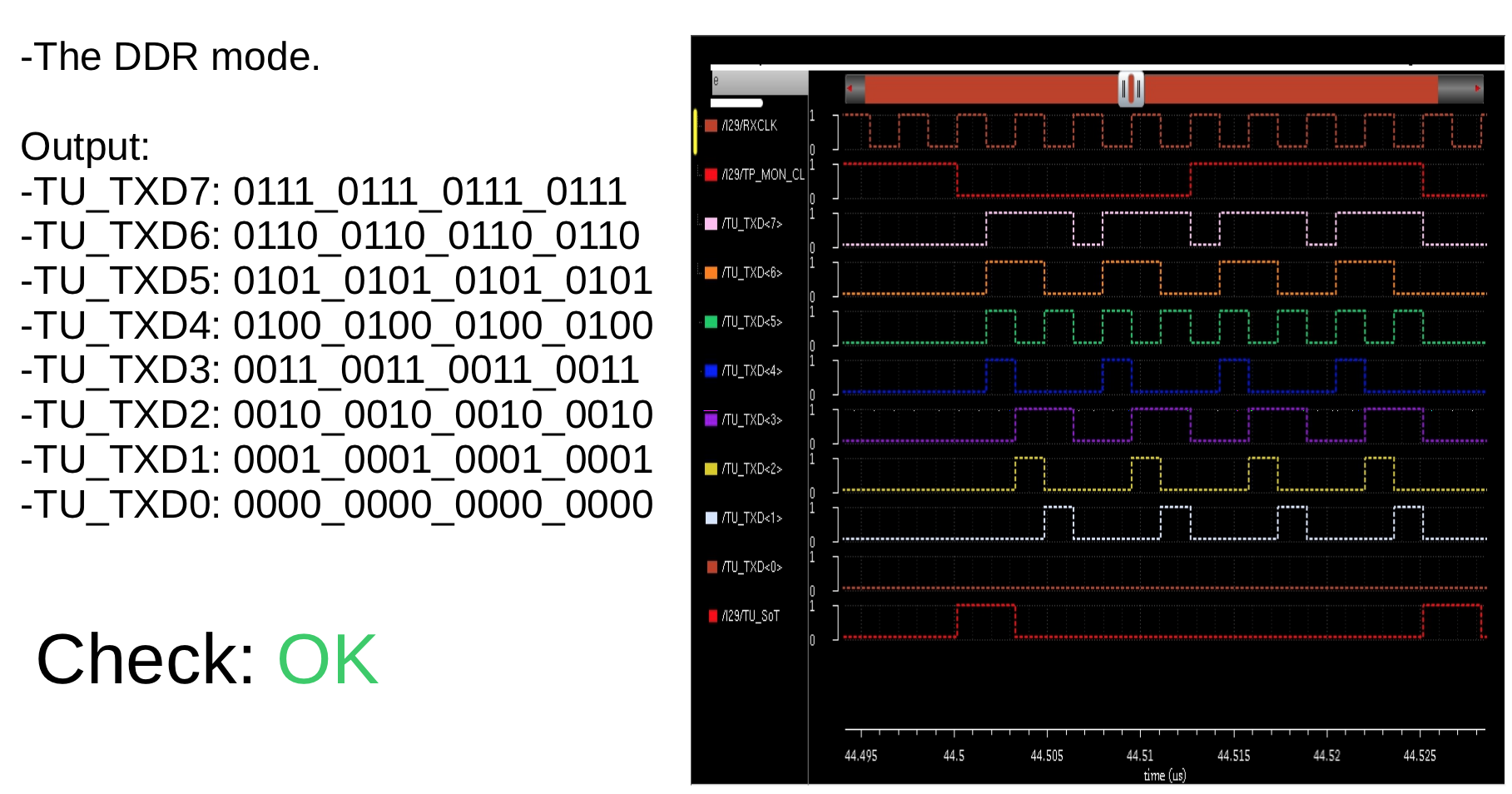
Alternatively, there is the capability to introduce bunch-crossing (integer) delays for a VFAT, but it is not anticipated that this would be necessary.

## S-bit Timing

Timing of S-bits is described in the presentation on Indico at: <https://indico.cern.ch/event/524671/contributions/2173475/attachments/1276978/1898262/Full_Chip_High_LeveL_Simulations__additional_sides.pdf>

For convenience, these are copied here:





# Trigger Primitive Compression Algorithm

Trigger information in the Optohybrid comes from the VFAT’s S-bits (Sector bits, or Sum bits, depending on who you ask). In the GE1/1 VFAT 3 these take the form of 8 320 MHz trigger links plus a 320 MHz synchronization pulse used to align the S-bits to the 40 MHz clock.

From 24 VFATs, the S-bits links are deserialized into 1536 S-bits divided in 8 physical partitions.

The S-bits are passed into a module called the cluster packer, which uses a priority encoding scheme to find the coordinates of 8 S-bit clusters inside of the GEM chamber.

All of the priority encoding logic operates at 160 MHz, so the number of bx consumed by the algorithm is 1/4 of the number of 160 MHz clocks.

There are two alternate versions of the cluster packer firmware which provide comparable functionality but with different trade-offs.

### Light Cluster Packing Algorithm

In the “light” version of the algorithm, cluster finding is segmented into two separate halves of the chambers. Thus, each one of the trigger fibers can transmit clusters only from the half of the chamber that it corresponds to.

This has the downside of being unable to transmit more than 4 clusters when they occur within that side of the chamber, so there will be a slightly higher rate of cluster overflow.

The benefit, however, is in terms of (1) latency and (2) resource usage.

The burden of finding clusters on only ½ of the chamber is significantly less, and allows the cluster packer to operate in a simple, pipelined architecture which returns up to 4 clusters per half-chamber per bunch crossing.

This faster architecture allows the mechanism to operate with only a single copy of the cluster finding priority encoder and cluster truncator (instead of two multiplexed copies), so the total resource usage of these stages is approximately half.

Further, a second step of cluster merging that is required in the full algorithm is avoided, which reduces latency by an additional bunch crossing and significantly reduces resource usage as well.

## Sequential Description of the Cluster Packing Algorithm

### Clock 1

Oneshots trim the tails of S-bit pulses in order to prevent retriggering on the same S-bit in subsequent clock cycles. The S-bit will become active again for triggering after it goes low for 1 clock cycle. This is especially important if the monostable length of the VFAT is set to anything longer than one.

An optional deadtime parameter can be controlled through the sys module to provide a 4-bit delay (0-15 bunch crossings) during which the S-bit cannot retrigger at all. This may be useful to supress afterpulsing which has been observed in the VFATs.

*\*\*n.b. that in VFAT v3 it has been observed that even with non-zero monostable pulse length, the S-bit will only fire once. i.e. the monostable multivibrator is only applied in the DAQ path and thus the monostable and deadtime has been disabled on v3*

### Clock 2

The second clock cycle is responsible for the creation of cluster flags and sizes which will be inputs to the priority encoder. A cluster is defined as a continuous sequence of S-bits, which can be encoded together to save bandwidth.

In the chosen data format, the maximum size of a cluster is 7, meaning that the primary S-bit was active along with 7 additional neighbours, for a total cluster size of 7. A cluster size of 0 indicates that only 1 S-bit was active.

Due to limitations of the logic, if a cluster is longer than 16 S-bits, its tail will simply be truncated.

There is an optional, compile-time parameter that can change this behaviour so that the tail of a cluster could be split into a 2nd cluster, with the limitation that after 16 S-bits it would be truncated again. By default this is disabled, due to the resource savings that are achieved.

In the firmware, two processes occur in parallel:

1st is that a “valid primary flag” is created for each of the 1536 S-bits, which identifies the particular S-bit as being the primary (first) S-bit in a cluster. This is done by looking for a preceding 0 (S-bit OFF).

2nd is that a count is derived for all pads, which is a 3-bit number following the scheme above.

For each of the 1536 S-bits, these 4 bits (vpf and count) are registered and passed to the priority encoder.

### Clock 3

This clock is used to register the vpfs and counts of the 1536 trigger pads (vpfs are latched inside the truncator, counts are latched inside the priority encoder), providing a 6.25 ns routing slack which allows the signals to be routed through the FPGA and aligned at the priority encoder.

Nota bene: This clock cycle is critical in the choice of the phase of the frame clock. The frame clock should be adjusted such that the rising edge of the 20 MHz frame clock is coincident with the rising edge of the clock at the end of this cycle.

Each 160 MHz clock cycle represents a 45 degree phase shift of the frame clock. Thus, because this step is in Clock 3, the phase shift applied to the frame clock should be 45 \* 3 = 135 degrees. This needs to be correctly configured in the MMCM

### Clock 4

Multi-bit priority encoding presents a significant challenge for a latency-constrained system. Finding a single cluster can be done efficiently through a multi-stage pipelined priority encoding tree. But finding subsequent clusters is very slow, because the entirety of the process must be pipelined:

e.g. 3 clock cycles to find the first cluster, mask it off, 3 clock cycles to find the second cluster, mask it off, 3 clock cycles to find the 3rd, etc.

It should be apparent that in such a scheme the latency would be very large.

Fortunately a clever scheme was devised which allows for masking and encoding to occur as separate, parallel processes.

This exploits the fact that the twos complement of a number has an interesting relation to its least significant set bit (note that the twos\_complement of a = -a = ~a+1).

At each clock cycle, the least-significant 1 becomes 0, using a simple property of integers: subtracting 1 from a number will always affect the least-significant set 1-bit. Using just arithmetic, with this trick we can take some starting number, and generate a copy of it that has the least-significant 1 changed to a zero.

e.g.

let a = 101100**1**00 // our starting number   
 ~a = 010011**0**11 // bitwise inversion  
 b = ~a+1 = 010011**1**00 // b is the twos complement of a  
 ~b = 101100**0**11 // bitwise inversion  
 a & b = 000000**1**00 // one hot of first one set  
 a &~b = 101100**0**00 // copy of a with the first set bit changed to 0

or as a one line expression in Verilog,

c = a & ~(~a+1), or equivalently  
 c = a & ~( -a), or equivalently  
 c = a & ~({1536{1'b1}}-a), etc., I'm sure there are more.

But alas, the point: we can Zero out bits without knowing the position of the bit, So this so-called cluster-truncator can run independently of a priority encoder that is finding the position of the bit. This allows the cluster truncation to be the timing critical step (running at 160 MHz) while the larger amount of logic in the priority encoder can be pipelined, to run over 2 or 3 clock cycles, which adds an overall latency but still allows the priority encoding to be done quickly without the necessity of a pipelined feedback.

This reduces the overall encoding time for 8 clusters in the example above from 3\*8 = 24 clock cycles to 2+8 = 10 clock cycles (2bx latency + 8 bx of data).

Additionally, operating on such large amounts of data (1536 bits) is a significant processing hurdle. To simplify this, split the chamber into two parts, each of which is finding 8 clusters from that half of the chamber (768). The findings of the two encoders will be merged at the end.

**Having said all this**--- in this clock cycle:

1. The “cluster truncator” applies this twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-1
2. In parallel, the data from the cluster truncator flip-flops is routed to the priority encoder, where it will be registered in order to find the 1st valid cluster.

### Clock 5

1. The “cluster truncator” applies its twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-2
2. In parallel, the data the first 4 pipeline stages of the priority encoder are completed, reducing the search set from 768 to 48 clusters.

### Clock 6

1. The “cluster truncator” applies its twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-3
2. In parallel, the data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster
   1. *1st valid cluster*

### Clock 7

1. The “cluster truncator” applies its twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-4
2. In parallel, the data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster
   1. 2nd valid cluster

### Clock 8

1. The “cluster truncator” applies its twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-5
2. In parallel, the data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster (cluster #3)
   1. *3rd valid cluster*

### Clock 9

1. The “cluster truncator” applies its twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-6
2. In parallel, the data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster (cluster #4)
   1. *4th valid cluster*

In the light version of the cluster packer, this is the end of cluster finding. 4 clusters are found in each 1/2 of the chamber.

In clock 10, these clusters are latched inside of the priority encoder. In clock 11 the clusters are aligned for output from the cluster packer module.

Thus the light version of the cluster packer has a latency of 11 clock cycles (just under 3 bunch crossings).

### Clock 10

1. The “cluster truncator” applies its twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0
   1. Cluster n-7
2. In parallel, the data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster (cluster #5)
   1. *5th valid cluster*

### Clock 11

1. The truncation process for this set of S-bits has completed and we are only awaiting the latency of the priority encoder. Because the data is a continuous input stream, we have a pipelined logic design which utilizes this logic again for new data before the previous set of S-bits has finished processing.   
     
   So in this clock cycle we load new data into the register from bunch crossing N+2 (where N was the bx of the data which we just completed) and repeat the same process:   
     
   The “cluster truncator” applies this twos-complement math to the S-bit valid-primary flags and registers a modified version of the logic with the least-significant S-bit set to 0 (cluster-1)  
     
   For clarity, in subsequent clock cycles, the descriptions of processing on the data from BX = N+2 is removed.
2. In parallel, the data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster (cluster #6)
   1. *6th valid cluster*

### Clock 12

In this clock:

1. The data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster (cluster #7)
   1. *7th valid cluster*

### Clock 13

1. The data the next 4 pipeline stages of the priority encoder are completed, reducing the search set from 48 to a single cluster, and returns the address, cnt, and vpf of that cluster (cluster #8)
   1. *8th valid cluster*
2. The 8 clusters from each of the two priority encoders are collated and aligned for input in into the merge16 module, which uses a merge-sort algorithm to pick out 8 clusters from the 16 inputs.

### Clock 14

1. The merge 16 module completes the first sets of sorting, swapping pairs from different halves of the chamber depending on whether which has the lowest address (invalid patterns are automatically set to the maximum address 0x7FF).   
     
   Stage 0: sort eights (0,8), (1,9), (2,10), (3,11), (4,12), (5,13), (6,14), (7,15)

### Clock 15

1. The merge 16 module completes the second set of sorting, swapping pairs of clusters depending on which has the lowest address.   
     
   Stage 1: sort fours (4,8), (5,9), (6,10), (7,11)
2. The merge 16 module completes the third set of sorting, swapping pairs of clusters depending on which has the lowest address.   
     
   Stage 2: sort twos (2,4), (3,5), (6,8), (7,9)

### Clock 16

1. The merge 16 module completes the fourth and last set of sorting, swapping pairs of clusters depending on which has the lowest address.   
     
   Stage 3: swap odd pairs (1,2), (3,4), (5,6), (7,8), (9,10), (11,12), (13,14)

### Clock 17

1. The 8 sorted clusters have a mask applied to them, so that during reset they are set to 0x7FE and if the Optohybrid trigger control is idle (waiting for bc0) they will be set to 0x7FD.   
     
   The masked clusters are moved onto the fabric 40MHz clock and will be sent out on the optical links.

A total of 16 clocks at 160Mhz means that the entire cluster encoding process is 4bx (100ns), exactly.

Retiming the module to consume an additional BX while adding additional pipeline stages would greatly ease the burden of place-and-route, but at this point does not seem necessary.

# Cluster Packer Miscellany

This section contains various other notes on the configuration, inputs, and outputs of the cluster packer aside from its algorithm.

### Cluster Counter

A submodule of the cluster packer produces a full 12 bit count of the number of clusters found in the chamber. This is accomplished with a multi-step pipelined adder tree that produces its result faster than the cluster packer. An SRL delay must be correctly programmed to align the adder results with the output of the cluster packer.

In another module of the Optohybrid firmware, the cluster count output is used in a rate counter. This module averages the cluster count over a compile-time programmable time window and produces an output in Hertz.

# LED Indicators

|  |  |
| --- | --- |
| Optohybrid v3 LED Assignments | |
| led[15] = GBT RX Ready and Valid | led[7] = Cluster Rate >= 10,000,000 Hz |
| led[14] = E-link Clock (divided) | led[6] = Cluster Rate >= 1,000,000 Hz |
| led[13] = Fabric Clock (divided) | led[5] = Cluster Rate >= 100,000 Hz |
| led[12] = GBT Request Received | led[4] = Cluster Rate >= 10,000 Hz |
| led[11] = L1A | led[3] = Cluster Rate >= 1,000 Hz |
| led[10] = Resync | led[2] = Cluster Rate >= 100 Hz |
| led[9] = BC0 | led[1] = Cluster Rate >= 10 Hz |
| led[8] = VFAT Reset | led[0] = Cluster Rate >= 1 Hz |

The “right side” LED indicators provide a logarithmic progress bar indicator which provides a convenient way to monitor the rate of incoming clusters received by the Optohybrid.

Prior to the receipt of the first S-bit (after a reset or resync), the progress bar will show a strobing “cylon” pattern to indicate an idle state.

# HDMI Output

The Optohybrid has an HDMI connector connected to differential pairs of the FPGA.

The functionality is programmable by changing firmware, but in the current configuration they provide a programmable way to output S-bits for use in the cosmic ray test stand.

Each conductor can be individually programmed into one of 4 modes:

* Mode 0: each signal was a single VFAT (set by selN)
* Mode 1: each signal was the OR of three VFATs in an ieta row (row is set by selN)
* Mode 2: each signal was the OR of four VFATs in an iphi half column (e.g. 0-3, 4-7, 8-11, 12 15, 16-19, 20-23) (phi half is set by selN)
* Mode 3: disabled

The registers you will want to use to control the HDMI output are:

* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL0
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL1
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL2
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL3
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL4
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL5
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL6
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_SEL7
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE0
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE1
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE2
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE3
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE4
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE5
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE6
* GEM\_AMC.OH.OH0.FPGA.CONTROL.HDMI.SBIT\_MODE7

The assignments of these signals in the HDMI cable are:

* NET "ext\_sbits\_o[0]" LOC = H17; # tmds clk p
* NET "ext\_sbits\_o[1]" LOC = G17; # tmds clk n
* NET "ext\_sbits\_o[2]" LOC = J16; # tmds d2 p
* NET "ext\_sbits\_o[3]" LOC = J17; # tmds d2 n
* NET "ext\_sbits\_o[4]" LOC = L14; # tmds d1 p
* NET "ext\_sbits\_o[5]" LOC = L15; # tmds d1 n
* NET "ext\_sbits\_o[6]" LOC = M17; # tmds d0 p
* NET "ext\_sbits\_o[7]" LOC = M18; # tmds d0 n

# Scripts

The optohybrid firmware respository contains a number of scripts that are useful or critical during firmware development.

A description of each follows:

### genproms.{sh/bat}

genproms.{sh/bat} provides Linux (sh) and Windows (bat) scripts that will generate EEPROM programming files (MCS) from the FPGA bitstream. They are wrapper around a TCL script that should be executed by the Xilinx xtcl interpreter.

If there are any issues with the script be sure that the Xilinx tools are in the user’s path.

### generate\_ucf.py

generate\_ucf.py will write the majority of pin LOC constraints to a Xilinx UCF file.

The generated constraints will be automatically inserted into the project’s constraint file, “misc.ucf”, in the section embedded between the comment tags:

#### START: AUTO GENERATED RESETS UCF -- DO NOT EDIT ####

#### END: AUTO GENERATED SBITS UCF -- DO NOT EDIT ####

The source of the UCF files is a multi-wire Netlist file, exported directly from the Altium PCB Project for the Optohybrid board.

### generate\_inversions.py

generate\_inversions.py will write the requisite polarity swaps into the trig\_pkg.vhd VHDL source package. This is used by the trigger deserialization firmware to invert the bits for S-bit channels which are polarity swapped on the Optohybrid PCB.

The specific channels which are polarity swapped need to be copied by hand from the schematics and recorded in the python file polarity\_swaps.py

### generate\_registers.py

generate\_registers.py will use the contents of the optohybrid\_registers.xml file to insert registers, counters, resets, etc. into the VHDL source code of the Optohybrid firmware.

It can be run with:

python generate\_registers.py oh 1

### generate\_timing\_registers.py

generate\_timing\_registers.py will convert a python dictionary of tap delay values into the requisite default delay values which will be written into the XML configuration file.

The values for these tables should be derived from the etch lengths of the transmission units on the GEB PCB.

The goal of this exercise is to tune the delays of all of the S-bit lines within a transmission unit to match the delay of the longest S-bit within that unit.

Each tap corresponds to an approximate 78 ps delay.

It is important to note that, because of the oversampling technique which is used to deserialize the S-bits with minimal clocking resources (which require the tap delays to introduce a 45 degree phase shift to a copy of the data), the maximum tap delay value which can be set here is 26 (~300 degrees).

Exceeding this value will result in undefined and incorrect behaviour.

# History

|  |  |  |
| --- | --- | --- |
| Date | Author | Modification |
| 8/7/2017 | AP | Initial conversion to OHv3 from Thomas’ v2 document |
| 8/10/2017 | AP | Description of GBT link formats |
| 8/14/2017 | AP | Updates register definitions according to latest firmware |
| 8/14/2017 | AP | Add description of synchronization and algorithmic description of the cluster packer |
| 8/17/2017 | AP | Update trigger description to match new firmware (reduced latency) |
| 8/18/2017 | AP | Update register description and GBT decoding for 16 bit addresses |
| 1/9/2018 | AP | Many additions to documentation based on latest firmware |
| 1/11/2018 | AP | Add descriptions of phase shifted clocks, and updated clocking diagram |