ADQ7-FWPD Development Kit

User Guide

Author(s): Teledyne SP Devices
Document ID: 19-2249
Classification: Public
Revision: A
Print date: 2022-04-07

Contents

1 Introduction **3**
1.1 Definitions and Abbreviations 3

2 Prerequisites **4**

3 Development Environment and Tools **5**
3.1 Unpacking the Development Kit 5
3.2 Opening the Development Kit 5
3.3 Setting Up the Project 6
3.4 Building the Design 6
3.4.1 Rebuilding the User Logic Areas 6
3.5 Simulating the Design 7
3.6 Working with the Design 7
3.6.1 Typical Design Flow 7

4 General Concepts **9**
4.1 Parallel Digital Design 9
4.2 Data Flow 9
4.3 Clock Domain Crossing Synchronization 11
4.4 Control Bus 12
4.4.1 Control Bus Signals 12
4.5 Data Bus 13
4.5.1 Two Bus Definitions 13
4.5.2 Low-Rate Channels 14
4.5.3 Bus Signals 14

5 User Logic 1 **26**
5.1 Linear Phase FIR Filter 26
5.2 Using MLVDS in MTCA Backplane 27

6 User Logic 2 **29**
6.1 Interface 30
6.2 Metadata 30
6.2.1 Interface 30
6.2.2 Bus Signals 33
6.2.3 Parallel Instances 34
6.2.4 Data Valid Generation 34
6.2.5 Metadata Generation 34
6.2.6 Metadata Packer 34
6.2.7 Metadata Buffer 34
6.2.8 Custom Metadata Logic 35

7 GPIO **37**

8 Troubleshooting **39**
8.1 Debugging on Hardware 39
8.1.1 Creating the Debug Core 39
8.1.2 Connecting to the Debug Core 40

9 The inner design of the Multiport DRAM **42**
9.1 Ports 42
9.2 Command mux and port arbitration 42
9.3 Command / data FIFO 44
9.4 Tag FIFO 44
9.5 Parameter READ_AFULL_DEPTH 44
9.6 A Note on Row Switches 44

10 Using VHDL instead of Verilog **46**

1 Introduction

This document is the user guide for the pulse detection firmware's development kit for the ADQ7 digitizer. There are different versions of the development kit depending on which device-to-host interface and which operational mode (one-channel or two-channel) of ADQ7 that is targeted. Make sure the development kit matches the target hardware.

The development kit centers around two *user logic areas*: UL1 and UL2. These areas target strategic points in the data path and are specifically intended to contain custom HDL designs.

The first user logic area, UL1, described in Section 5, operates on the full-rate data stream—before the trigger information has been decoded to create records. The second user logic area, UL2, described in Section 6, operates on complete records, potentially with a reduced sampling rate.

Important

The pulse detection firmware does not support the sample skip feature. Thus, the sampling rate cannot be reduced beyond the base rate.

1.1 Definitions and Abbreviations

Table 1 lists the definitions and abbreviations used in this document.

Table 1: Definitions and abbreviations used in this document.

Item	Description
ADC	Analog-to-digital converter
CDC	Clock domain synchronization
DCP	Device checkpoint—represents a saved design state in Vivado.
DevKit	Development kit
FPGA	Field-programmable gate array
FWPD	The pulse detection firmware for ADQ14 and ADQ7.
GiB	Gibibyte (1024^3 bytes)
PROM	Programmable read-only memory
PS	Parallel samples
PP	Parallel packets
`QX.Y`	Fixed-point representation with `X` integer bits and `Y` fractional bits.
RTL	Register transfer level
Tcl	Tool command language—scripting language used in Vivado.
UL1	User logic 1—the first open FPGA area, see Section 5.
UL2	User logic 2—the second open FPGA area, see Section 6.
VHSIC	Very high speed integrated circuit
VHDL	VHSIC hardware description language
Verilog	Hardware description language
Vivado	Xilinx FPGA design suite
XCI	Xilinx core instance

2 Prerequisites

The development kit has the following prerequisites:

- A license for the ADQ7 development kit purchased from Teledyne SP Devices.
- A license for the Xilinx design tools. For current versions of the development kit, a license for *Vivado 2020.2* is required (see table below).
  - Minimum tooling is the Vivado *Design Edition*.
  - The Vivado *WebPack* does not support the family of ADQ products.
  - Xilinx *ISE* cannot be used.
- Previous experience with defining custom logic using Verilog or VHDL.

Table 2: Version requirement

Development Kit Revision	Tool version
≤r58810	Vivado 2017.1
≥r58811	Vivado 2020.2

3 Development Environment and Tools

This section describes the development kit workflow and the associated tools.

3.1 Unpacking the Development Kit

The development kit is delivered as a `.zip` archive containing the project files, source files and documentation. The first level of the archive contains a `README` file and another archive that is password protected. By unpacking the password-protected archive, the user agrees to the terms of the development kit license. Make sure to extract the archive to a directory where the current user has read *and* write permissions.

⚠ Warning

By unpacking the password-protected archive, the user agrees to the terms of the development kit license.

Important

The archive should be extracted to a directory where the user has read and write permissions.

The archive is organized as follows:

`<Archive root>/`

`constraints/`	Contains the constraint files for the design.
`documentation/`	Contains the documentation for the development kit.
`edif/`	Contains the device checkpoint (DCP) file of the surrounding FPGA design.
`elf/`	Contains the Microblaze PROM file.
`implementation`	Contains the Tcl scripts that abstracts several workflow operations.
`ip/`	Contains the configuration files for the Xilinx IPs used in the design.
`source/`	Contains the Verilog source files for modules used in the design.
`test/`	Contains example test benches.
`License.txt`	Development kit license file.

Note

Though every file in the `source/` directory is available for editing, only a few files should be edited. This is explained further in Sections 5 and 6 which deals with the two user logic areas.

3.2 Opening the Development Kit

To open the development kit in Vivado, follow the steps outlined below.

1. Start Vivado

2. In the menu bar, select *Tools* > *Run Tcl Script...*
3. Open the file `<Archive root>/implementation/scripts/devkit.tcl`. The Tcl console will output the following text:

```
*** ADQ7 Development Kit ***
Usage:
  devkit_setup          - Create project
  devkit_build          - Build project
  devkit_synth_ul 1     - Generate netlist for user_logic1
  devkit_synth_ul 2     - Generate netlist for user_logic2
  devkit_mcs            - Generate .mcs firmware file
  devkit_sim_ul 1       - Run testbench for user_logic1
  devkit_sim_ul 2       - Run testbench for user_logic2
```

At this point, the Tcl commands specific to the development kit have been defined and are available in the Tcl console. The project is now ready to be set up for first-time use.

3.3 Setting Up the Project

To set up the development kit, execute the command

```
devkit_setup
```

in the Tcl console. The process may take a few moments to finish since parts of the design will need to be compiled. Once the setup is complete, a Vivado project has been created and the design is ready to be built. This step only has to be completed once.

3.4 Building the Design

To build the entire design, execute the command

```
devkit_build
```

in the Tcl console. Depending on the computer specifications and the complexity of the design as a whole, i.e. the precomplied design and the user logic together, this may take several hours. Once the process is complete, an `.mcs` file has been generated in the `implementation/` directory. This file represents a new firmware for the digitizer and may be uploaded using the *ADQUpdater* application. Refer to the ADQUpdater user guide [1] for instructions on how to manage the firmware on the ADQ7 digitizer.

3.4.1 Rebuilding the User Logic Areas

The netlists for the two user logic areas may be manually rebuilt using the Tcl command

```
devkit_synth_ul <target>
```

where `<target>` is either `1`, to target the first area, or `2` to target the second.

Once the netlists have been updated to reflect the changes to the design, use the Vivado GUI to generate a bitstream. Convert the bitstream to an `.mcs` file by executing the command

```
devkit_mcs
```

in the Tcl console.

Important

Due to a bug in Vivado, the design must be opened after synthesis and the command `refresh_design` executed in the Tcl console. Following this action, the build flow may be resumed, continuing with the *implementation* step. This occurs automatically when running `devkit_build`.

3.5 Simulating the Design

The development kit provides simple test benches for both user logic areas. These can be used as a base for your own tests.

To set up and run the tests in Vivado, use the command

```
devkit_sim_ul <target>
```

where `<target>` is either `1`, to target the first area, or `2` to target the second.

After that, the test can be rerun using the usual Vivado commands.

3.6 Working with the Design

Each user logic area can be set as the top-level module of the design (one at a time) with the command

```
devkit_set_top_ul <target>
```

where `<target>` is either `1`, to target the first area, or `2` to target the second. This action is helpful when analyzing the design using Vivado's RTL analysis tools. Restore the default top-level module by executing the command

```
devkit_set_top
```

in the Tcl console.

3.6.1 Typical Design Flow

This section outlines the typical design flow for the development kit.

1. Set up the development kit project as described in Section 3.3.
2. Modify or insert new Verilog code into `user_logic1.v` or `user_logic2.v`. This operation can be broken down into four steps:
   (a) Extract *data*, *data valid* and relevant *bus signals* using the *bus extraction macros* (see Section 4.5.3).

   (b) Process the extracted signals, i.e. stimulate the custom user design.
   (c) Insert the processed data, data valid and relevant bus signals using the *bus insertion macros* (see Section 4.5.3).
   (d) Set the correct value for the `BUS_PIPELINE` delay parameter to keep the correct time relation between signals that were not manually inserted.

3. Generate the FPGA configuration file (`.mcs` file) by using one of the two methods outlined below:
   - *Automatic*
     (a) Execute the command
     ```
     devkit_build
     ```
     in the Tcl console.
   - *Manual*
     (a) Generate the netlist(s) for the modified code by executing the command
     ```
     devkit_synth_ul <target>
     ```
     for the modified user logic areas.
     (b) Launch the synthesis step by selecting *Run Synthesis* in Vivado.
     (c) Once the synthesis step is complete, open the design by selecting *Open Synthesized Design*.
     (d) Once the design has been opened, execute the command
     ```
     refresh_design
     ```
     in the Tcl console.
     (e) Generate the bitstream by selecting *Generate Bitstream*. This action will launch the implementation step and will end with the bitstream generation.
     (f) Once the bitstream is available, generate the FPGA configuration file by executing the command
     ```
     devkit_mcs
     ```
     in the Tcl console. The configuration (`.mcs`) file can be found in the `implementation/` directory after the process is complete.

4. Program the configuration file representing the custom design into the digitizer using the ADQUpdater application. Refer to the corresponding user guide [1] for details on the programming process.

5. Test the custom firmware using either
   - one of the software examples available in the ADQAPI library or
   - a custom user application.

4 General Concepts

This section introduces concepts surrounding the development kit for ADQ7 in general. The reader is assumed to be familiar with digital design.

4.1 Parallel Digital Design

More often than not, the FPGA cannot be clocked at the same rate as the incoming data. To handle this scenario, the logic needs to be implemented to handle several data words per clock cycle, i.e. several data words in *parallel*. Parallel design is more challenging than its counterpart, where one data word is processed per clock cycle, due to the many pitfalls inherent to the former. Instructing the reader on parallel design is outside the scope of this document but moving forward, some familiarity with the concept is expected.

In ADQ7, the FPGA is clocked at 312.5 MHz while the data rate is 10 GSPS in the one-channel mode and 5 GSPS in the two-channel mode. Since a data word is equal to a sample, a parallelization of 32 is required in the one-channel mode while the two-channel mode instead requires a parallelization of 16.

Note

The data bus is clocked at 312.5 MHz with a parallization of 32 for the one-channel mode and 16 for the two-channel mode.

4.2 Data Flow

Fig. 1 presents an overview of the data path where the two user logic areas are highlighted. A brief description of each block in the data path is provided. Throughout this section, knowledge of concepts surrounding parallel digital design is assumed. A short summary is presented in Section 4.1. A *cycle* may be used to refer to a *data clock cycle*.

The data propagates between the modules in the data path using an *AXI stream* bus interface with custom insertion and extraction macros for convenience (see Section 4.5.3). Additionally, each module has access to the control bus via an *AXI* bus interface. These two buses do *not* exist in the same clock domain, meaning any signals transferred from one domain to the other *must* be synchronized to the receiving clock. Refer to Sections 4.3 and 4.4 for additional details.

Important

Any signals transferred from the control bus clock domain to the data bus clock domain or vice versa *must* be synchronized to the receiving clock.

Trigger module

The trigger module is tasked with decoding and inserting *trigger events*, as well as the ADC data, on the data bus. A trigger event consists of a single bit indicating the event itself and additional information such as the position of the event within the data word. Additionally, functions involving the timestamp are also located in this module.

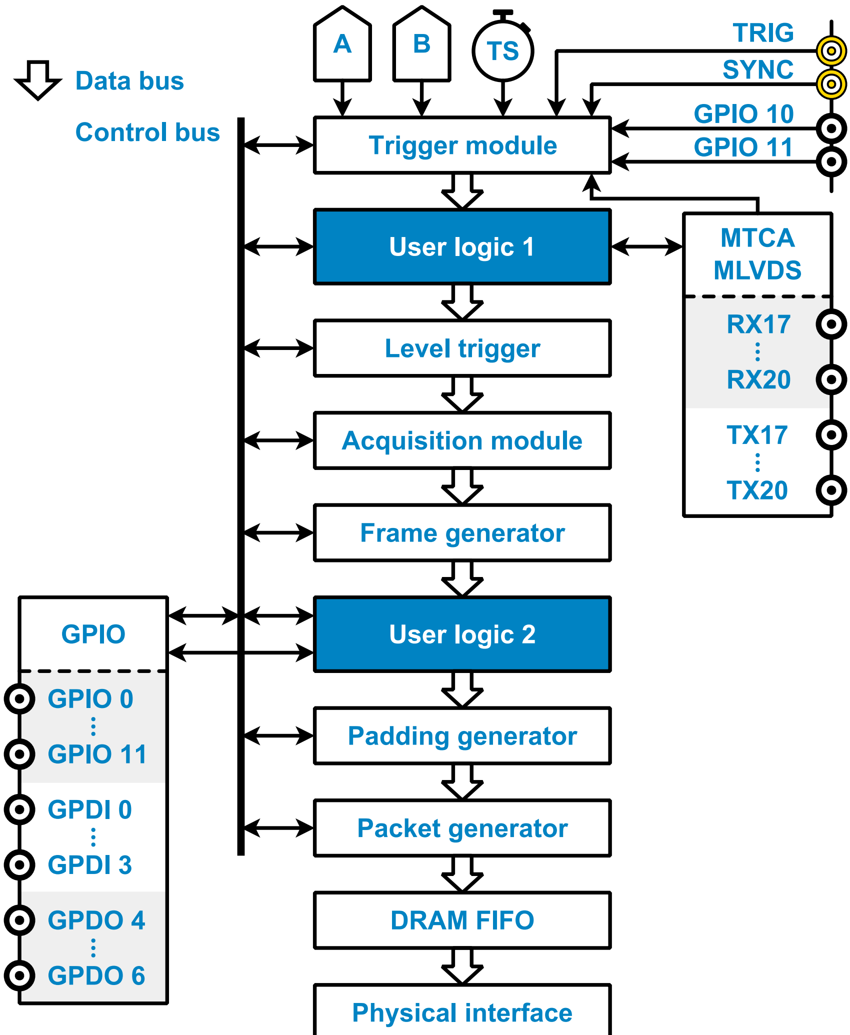

Figure 1: A block diagram of the data path of ADQ7-FWPD. The two user logic areas are highlighted. In the one-channel mode, the data from both ADCs is interleaved and propagates on channel A.

User logic 1

The first user logic area in the design. Refer to Section 5 for a more detailed description of the module.

Level trigger

The level trigger module is located after the first user logic area—allowing the user to manipulate the data stream before threshold crossings are identified. This module is tasked with identifying the pulses within the data stream. Refer to the ADQ7-FWPD user guide [2] for additional details on the various settings available for this module.

Acquisition module

The acquisition module is tasked with framing the channel data into *records*. A record consists

of a continuous number of samples where the starting point and stopping point is marked by a logic-high pulse on the record start and record stop data bus signals, respectively. Since records can only begin and end once per data clock cycle, the minimum record length is one data clock cycle and the record length must be divisible by the degree of parallelization.

Frame generator

The frame generator is a module specific to ADQ7-FWPD and is tasked with generating the signals defining a detection window and the padding grid. The concept behind these features are explained in the ADQ7-FWPD user guide [2] but the HDL interface is described in Section 6.

User logic 2

The second user logic area in the design. Refer to Section 6 for a more detailed description of the module.

Padding generator

The padding generator is a module specific to ADQ7-FWPD and is tasked with maintaining a certain throughput on the physical interface, regardless of if there were any pulses detected or not. Additional details may be found in the ADQ7-FWPD user guide [2] .

Packet generator

The packet generator is tasked with converting the information on the data bus into packets. While this operation adds some overhead to the data stream, the benefits outweigh the drawbacks since the data is more manageable in packet form when transmitting over a physical interface.

DRAM FIFO

The packet generator is followed by a DRAM FIFO with a capacity of 2 GiB. This scheme allows the digitizer to buffer data in the event of a temporary stall on the physical interface. When data is discarded due to an imbalance between the acquisition rate and the readout rate, packets are discarded at the FIFO Input,

Physical interface

The digitizer's physical interface, e.g. USB, PCIe, PXIe or MTCA.

4.3 Clock Domain Crossing Synchronization

A clock domain crossing (CDC) is a boundary where digital signals pass from one clock domain to another. This boundary constitutes a critical point in the design and care must be taken to *synchronize* signals passing through the boundary to the *receiving* clock.

There are several techniques to choose from depending on the type of signal that should be synchronized, e.g. a multi-bit signal is not handled in the same way as a signal that is 1 bit wide. The reader is expected to be familiar with CDC synchronization techniques. The paper by Clifford E. Cummings [3] is a good place to start if the reader's knowledge needs to be refreshed.

In each of the user logic areas, there is one clock domain crossing in the default design—between the control bus clock and the data bus clock. This boundary joins the control bus register values, representing

the current configuration, and the data bus logic, tasked with processing the data. To aid the user, there are two CDC helper modules available in the `source/` directory:

```
source/
├ ul_cdc_sync.v            CDC synchronization module for a 1-bit signal.
└ ul_cdc_sync_bus_ce.v     CDC synchronization of a multi-bit signal using a strobe.
```

These modules should cover any CDC needs and should be used whenever CDC synchronization is called for. Refer to Section 4.4 for details on the control bus and to Sections 5 and 6 for examples of CDC synchronization and logic making use of these register values.

4.4 Control Bus

Each user logic area has access to the control bus which provides a connection between the custom logic and the soft microprocessor located in the enclosing design. A transaction cannot be started from a user logic area and thus, communicating between the two modules using the control bus is not supported. Instead, transactions are initiated by the microprocessor which in turn is initiated from the ADQAPI, specifically by using the functions to read or write user registers.

Access a single register

- `ReadUserRegister()`
- `WriteUserRegister()`

Access a range of registers

- `ReadBlockUserRegister()`
- `WriteBlockUserRegister()`

> **Note**
>
> Transactions on the control bus cannot be initiated from the user design, only from calling specific functions in the ADQAPI.

The default user logic design implements a register map but the bus can also be used to interface block RAMs, FIFOs or other custom logic.

4.4.1 Control Bus Signals

Table 3 presents the signals on the control bus. An acknowledge signal *must* be transmitted as a response to all read and write requests. Otherwise, the bus will hang and the digitizer may become unresponsive.

> **Important**
>
> An acknowledge signal *must* be transmitted as a response to all read and write requests. Otherwise, the bus will hang and the digitizer may become unresponsive.

Table 3: The signals on the control bus.

Signal	Description	Direction	Polarity
`clk`	Bus clock	Input	N/A
`rst_i`	Start-up reset	Input	Active high
`addr_i`	Read/write address, 14 bits	Input	N/A
`wr_i`	Write strobe	Input	Active high
`wr_ack_o`	Write data acknowledge	Output	Active high
`wr_data_i`	Write data 32 bits	Input	N/A
`rd_i`	Read strobe	Input	Active high
`rd_ack_o`	Read acknowledge	Output	Active high
`rd_data_o`	Read data 32 bits	Output	N/A

4.5 Data Bus

The stream of ADC data, its associated control signals and other metadata all propagate on the data bus. The various signals are intricately related to each other and it is crucial that their relation in time is kept intact while they are processed by the custom logic.

Important

The bus signals are closely related to each other and it is crucial that their relation in time is kept intact through the user logic areas.

The development kit includes predefined functions to simplify the bus operations. There are two points where the user design interfaces with the data bus: *extraction* and *insertion*. As the names suggest, targeted signals are extracted from the bus and input to the custom logic to create a response. The logic's output signals are inserted back into the data bus and continues to propagate through the design (see Fig. 1). Signals that are *not* inserted back into the data bus will be subjected to pipelining with a delay equal to the value of the `BUS_PIPELINE` parameter. This parameter must be defined in the same HDL source file as the bus operations. Fig. 2 outlines the principle of working with the bus signals in the user space.

4.5.1 Two Bus Definitions

Roughly halfway through the design, the data bus is redefined. At this point, a few signals are added to the bus and a few existing signals change their width. The two definitions are labeled *real-time* and *reduced rate* where the former describes the composition of the bus up until the point where it is redefined. Regardless of where the redefinition occurs, the important point is that the first user user logic area uses the real-time bus definition while the second user logic area uses the reduced rate definition. The two bus definitions each have their own set of functions to support bus operations. These are available in two Verilog header (`.vh`) files in the `source/` directory.

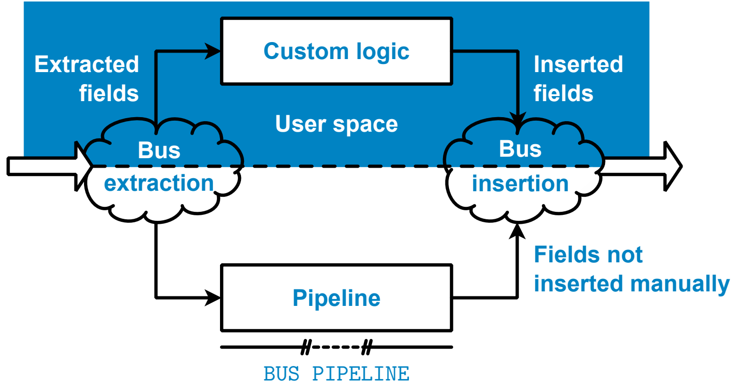

Figure 2: A diagram showing the principle of extracting signals from and inserting signals into the data bus. Any field not inserted manually is subjected to a pipeline delay equal to the value of the BUS_PIPELINE parameter.

source/

└ bus_splitter_rt.vh	Defines functions to interface with the real-time data bus.
└ bus_splitter_rr.vh	Defines functions to interface with the reduced rate data bus.

4.5.2 Low-Rate Channels

So far, a *channel* on the data bus has been used to describe the abstraction of the ADC data and its associated signals—i.e. there has been a one-to-one relation between the physical input signal and the data bus representation. However, the reduced rate data bus also defines *low-rate channels*. These behave like the channels transporting ADC data, but they do not have any default content and rely entirely on a custom user design to define their data.

Normally, the low-rate channels have a lower bandwidth than their counterpart, implemented as *fewer* number of parallel samples per data clock cycle. However, this is not always the case and the user should make use of the available definitions, e.g. UL2_SPD_PARALLEL_SAMPLES_LOWRATE, together with parameter validation to make the custom design more robust. This constant and others are defined in files belonging to the respective user logic area. Refer to Sections 5 and 6 for additional details.

Utilizing the low-rate channels may come at a cost—since they enable the digitizer to generate a higher output data rate than the input data rate. The user must always be vary of the bandwidth of the device-to-host interface to keep the system balanced.

4.5.3 Bus Signals

This section provides a reference for the bus signals on the real-time (RT) and the reduced rate (RR) data buses. Each section defines one bus signal, provides a description of its functionality and purpose and lists the associated bus interface functions. The interface functions are defined on the form insert_* and extract_*, where * is a string based on the signal name.

Section 4.5.1 explained that the definitions of these interface functions are located in two files: bus_splitter_rt.vh and bus_splitter_rr.vh. The user *must only* include the file matching the bus definition in the target user logic area, i.e.

- `bus_splitter_rt.vh` for the first user logic area (UL1) and
- `bus_splitter_rr.vh` for the second user logic area (UL2).

Note

In the default design, the source files for the two user logic areas import the appropriate bus definition.

Table 4: This table presents an overview of the signals on the real-time and the reduced rate data buses. The table is separated into two sections: one for signals common between the two buses and one for signals unique to the reduced rate data bus. Refer to the section for each individual signal for details.

Signal	Page
Common signals	
Timestamp	15
Trigger inhibit	16
Timestamp synchronization	16
Timestamp synchronization counter	17
ADC data	19
Trigger	20
Overrange indicator	21
General purpose	21
Auxilliary trigger	25
Reduced rate signals	
Data valid	19
Record bits	22
Record counter	23
User ID	23
Trigger event bit vector	24
Reset event bit vector	24

Timestamp *RT: single, RR: per channel*

The *timestamp* signal is tasked with providing a monotonically increasing counter to serve as a time base for the digitizer. The signal is 64 bits wide and holds an unsigned value that may be synchronized to external and internal events by using the timestamp synchronization mechanism. This feature is outlined in the ADQ7-FWPD user guide [2] . The resolution of the time base on ADQ7 is 25 ps. Hence, the counter increments its value by 128 each data clock cycle.

The timestamp value during a data clock cycle where record start is asserted propagates to the user space in the host computer via the record header.

`insert_timestamp(signal)` *RT*

Insert the 64-bit timestamp signal into the real-time data bus. The timestamp `signal` is expected as an input argument.

`insert_timestamp(signal, channel)` *RR*

Insert the 64-bit timestamp signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_timestamp(DONT_CARE)` *RT*

Extract the 64-bit timestamp signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_timestamp(channel)` *RR*

Extract the 64-bit timestamp signal from the reduced rate data bus. The target `channel` is expected as an input argument.

Trigger inhibit *RT: single, RR: per channel*

The *trigger inhibit* signal is controlled by the trigger blocking mechanism. The signal is one bit wide and a logic high level implies that triggers are *blocked*, i.e. trigger events are not converted into records by the acquisition module. Conversely, a logic low level implies that triggers are accepted. Refer to the ADQ7-FWPD user guide [2] for additional details on the trigger blocking mechanism.

`insert_trig_inhibit(signal)` *RT*

Insert the 1-bit trigger inhibit signal into the real-time data bus. The inhibit `signal` is expected as an input argument.

`insert_trig_inhibit(signal, channel)` *RR*

Insert the 1-bit trigger inhibit signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_trig_inhibit(DONT_CARE)` *RT*

Extract the 1-bit trigger inhibit signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_trig_inhibit(channel)` *RR*

Extract the 1-bit trigger inhibit signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Timestamp synchronization *RT: single, RR: per channel*

The *timestamp synchronization* signal is controlled by the timestamp synchronization mechanism. The signal is one bit wide and a logic high level implies that the digitizer is waiting for a timestamp

synchronization event and a logic low level implies that the timestamp is in sync—provided the level was previously logic high—or that the mechanism has not been activated.

`insert_timestampsync(signal)` *RT*

Insert the 1-bit timestamp synchronization signal into the real-time data bus. The synchronization `signal` is expected as an input argument.

`insert_timestampsync(signal, channel)` *RR*

Insert the 1-bit timestamp synchronization signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_timestampsync(DONT_CARE)` *RT*

Extract the 1-bit timestamp synchronization signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_timestampsync(channel)` *RR*

Extract the 1-bit timestamp synchronization signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Timestamp synchronization counter *RT: single, RR: per channel*

The *timestamp synchronization counter* signal is controlled by the timestamp synchronization mechanism. The signal is 16 bits wide and holds an unsigned value tasked with keeping track of the number of timestamp synchronization events since the mechanism was last armed.

The counter value during a data clock cycle where record start is asserted propagates to the user space in the host computer via the record header—provided the mechanism is set up and activated.

`insert_timestamp_sync_cnt(signal)` *RT*

Insert the 16-bit timestamp synchronization counter signal into the real-time data bus. The counter `signal` is expected as an input argument.

`insert_timestamp_sync_cnt(signal, channel)` *RR*

Insert the 16-bit timestamp synchronization counter signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_timestamp_sync_cnt(DONT_CARE)` *RT*

Extract the 16-bit timestamp synchronization counter signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_timestamp_sync_cnt(channel)` *RR*

Extract the 16-bit timestamp synchronization counter signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Trigger event *RT: per channel, RR: per channel*

The *trigger event* is a 1-bit signal indicating that the configured trigger condition has been met in this data clock cycle. The condition is specified through the ADQAPI function `SetTriggerMode()`. The signal is active high. The bus interface is listed in Tables 5 and 6.

Trigger event edge *RT: per channel, RR: per channel*

The *trigger event edge* indicates the polarity of the trigger event. A *rising* edge event is indicated by a logic high level and a *falling* edge event is indicated by a logic low level. The value is only valid when the trigger event is asserted. The bus interface is listed in Tables 5 and 6.

Trigger event number *RT: per channel, RR: per channel*

The *trigger event number* is an 8-bit signal holding an unsigned value indicating where in the data word the trigger occurred. Similar to the timestamp, the signal has a 25 ps resolution but the value is aligned within the eight bits so that a `Q5.3` representation is maintained. In this representation, the value relates to the current sampling rate so that

- the five MSBs form the integer part of the number—indicating, with *sample precision*, a floored value for the trigger point and
- the three LSBs form the start of the fractional part of the number, that together with the extended precision indicates, with *sub-sample* precision, the closest 25 ps grid point to the trigger point.

The value is only valid when the trigger event is asserted. The bus interface is listed in Tables 5 and 6.

Trigger extended precision *RT: N/A, RR: per channel*

The *trigger extended precision* signal is a 16-bit signal holding an unsigned value providing extended precision for the trigger point when the sample skip mechanism is active. When samples are discarded in the full-rate data stream, the effective sampling rate is reduced. However, the decimal point in the trigger event number remains fixed and so does the interpretation of the field. Thus, the additional bits provided by this signal are needed to keep the high precision trigger information.

All 16 bits are interpreted as a continuation of the fractional part of the trigger event number. Concatenated, the two signals become a `Q5.19` fixed-point number indicating the trigger position within the data word, with respect to the effective sampling rate. The value is only valid when the trigger event is asserted. The bus interface is listed in Tables 5 and 6.

Reset event *RT: per channel, RR: per channel*

The *reset event* signal is a 1-bit signal with the same principal behavior as a trigger event. However, the signal is exclusively used by the level trigger and by the pulse detection firmware in particular. In FWPD the signal indicates the *reset* condition of a pulse, i.e. the complement to

the trigger event. Refer to the ADQ7-FWPD user guide [2] for details on how the level trigger defines the two event types. The signal is active high. The bus interface is listed in Tables 5 and 6.

Reset event (pretrigger) *RT: per channel, RR: per channel*

Identical in function to the reset event but subjected to the pretrigger delay, i.e. the event is always in sync with the ADC data. The bus interface is listed in Tables 5 and 6.

Reset event edge *RT: per channel, RR: per channel*

The *reset event edge* is a 1-bit signal with the same functional behavior as the trigger event edge except it targets the reset event. The value is only valid when the reset event is asserted. The bus interface is listed in Tables 5 and 6.

Reset event number *RT: per channel, RR: per channel*

The *reset event number* is a 8-bit signal holding an unsigned value with the same functional behavior as the trigger event number except it targets the reset event. The value is only valid when the reset event is asserted. The bus interface is listed in Tables 5 and 6.

Data *RT: per channel, RR: per channel*

The *data* signal holds the ADC data on a per-channel basis and consists of several samples in parallel, as explained in Section 4.1. The width of the signal depends on the firmware configuration, i.e. if the ADQ7 is running in the one-channel mode or in the two-channel mode. For this purpose, each user logic area defines constants which *must* be used to parametrize a custom design. For example, UL1 defines the width of one sample as `UL1_SPD_DATAWIDTH_BITS` and the number of parallel samples as `UL1_SPD_PARALLEL_SAMPLES`. The width of a sample is always constant but the number of parallel samples differs between the two base designs (see Section 4.1).

A sample is encoded using a 16-bit *2's complement* representation. The data is MSB-aligned, meaning that the MSB from the raw 14-bit ADC data is located at bit index 15.

`insert_ch_all(signal, channel)` *RT, RR*

Insert the `UL1_SPD_PARALLEL_SAMPLES` · `UL1_SPD_DATAWIDTH_BITS`-bit data signal into either data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_all(channel)` *RT, RR*

Extract the `UL1_SPD_PARALLEL_SAMPLES` · `UL1_SPD_DATAWIDTH_BITS`-bit data signal from either data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Data valid *RT: N/A, RR: per channel*

The *data valid* signal exists on a per-channel basis on the reduced rate data bus. On the real-time data bus, the signal is not present since by definition, every cycle is considered valid. The signal is one bit wide and when asserted, every sample in the data word is considered valid, regardless of the sample skip factor.

`insert_data_valid(signal, channel)` *RR*

Insert the 1-bit data valid signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_data_valid(channel)` *RR*

Extract the 1-bit data valid signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Trigger *RT: per channel, RR: per channel*

The *trigger* is a collection of the trigger-related signals needed to support ADQ7's function set and exists on per-channel basis. Table 5 presents an overview of the trigger and its member signals.

Important

It is not recommended to extract and decode the trigger signal explicitly, but rather to use the functions targeting the individual signals within.

Table 5: This table presents the signals that together constitute the trigger for a data channel. The table is divided into two sections: one for signals common between the trigger definitions on the real-time data bus and the reduced rate data bus, and one for signals unique to the latter.

Signal	Bus interface
Common signals	
Trigger event	`insert_ch_trig_tevent(signal, channel)` `extract_ch_trig_tevent(channel)`
Trigger event edge	`insert_ch_trig_trising(signal, channel)` `extract_ch_trig_trising(channel)`
Trigger event number	`insert_ch_trig_tnum(signal, channel)` `extract_ch_trig_tnum(channel)`
Reset event	`insert_ch_trig_revent(signal, channel)` `extract_ch_trig_revent(channel)`
Reset event (pretrigger)	`insert_ch_trig_revent_pt(signal, channel)` `extract_ch_trig_revent_pt(channel)`
Reset event edge	`insert_ch_trig_rrising(signal, channel)` `extract_ch_trig_rrising(channel)`
Reset event number	`insert_ch_trig_rnum(signal, channel)` `extract_ch_trig_rnum(channel)`
Reduced rate signals	
Trigger extended precision	`insert_ch_trig_extended_precision(signal, channel)` `extract_ch_trig_extended_precision(channel)`

Overrange indicator *RT: per channel, RR: per channel*

The *overrange indicator* is a 1-bit signal indicating that *at least* one sample in the current data word has saturated to the minimum or maximum value in the range, whichever is closest.

`insert_over_range(signal, channel)` *RT, RR*

Insert the 1-bit data signal into either data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_over_range(channel)` *RT, RR*

Extract the 1-bit data signal from either data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

General purpose *RT: per channel, RR: per channel*

The *general purpose* signal is 16 bits wide and may be used to achieve various firmware-specific goals. Normally, the enclosing design does not impose any restrictions on the signal, but special-purpose firmwares may reserve the signal for internal use. If no functional conflict exists, the signal may be used to, e.g. pass information between the two user logic areas in a well-defined manner. However, if sample skip is employed, only information passed while the record start bit is asserted is preserved throughout the data path. Additionally, the value during a data clock cycle which asserts the record start bit, will propagate to the user space in the host computer via the record header. The general purpose vector can also be used to transport information from user logic 1 to user logic 2, as it is passed along the bus.

Note

Future revisions of the pulse detection firmware may claim the general purpose signal to transmit the moving average of the baseline of the ADC data.

Important

The general purpose vector is not synchronized in time with data from user logic 1 to user logic 2, it is rather transported as fast as possible. The transport timing for this general purpose vector is also subject to changes between DevKit revisions, thus the user has to design a robust way to handle these properties. This is for instance made especially clear when activating sample skip in between, which makes data arrive delayed to user logic 2 compared to the general purpose vector

`insert_general_purpose_vector(signal, channel)` *RT, RR*

Insert the 16-bit general purpose signal into either data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_general_purpose_vector(channel)` *RT, RR*

Extract the 16-bit general purpose signal from either data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Record bits *RT: N/A, RR: per channel*

The *record bits* signal consists of two 1-bit signals: *record start* (bit 0) and *record stop* (bit 1). Together they frame a record. They are *inclusive*, meaning that if either signal is asserted, the data associated with that data clock cycle belongs to the record.

The data valid signal is guaranteed to be asserted if either of the record bits is asserted. It is *imperative* that any custom design in the second user logic area keeps this property.

> **Important**
>
> The data valid signal is guaranteed to be asserted when either of the record bits are. This property *must* be preserved when handling the bus signals in a custom design placed in the second user logic area.

Figs. 3 and 4 present a timing diagram for the record bits when sample skip is disabled and enabled, respectively. Note that record start and record stop may be asserted for multiple cycles, with only one cycle which overlaps with data valid.

It is important that the integrity of the record bits is preserved. The second user logic area must output one—and only one—record stop for each record start event. To discard a record, record start and record stop assertions must be removed *in pairs*. If multiple record start events are output without their corresponding record stop events, data corruption will ensue. Listed below is a summary of the properties of the record bits.

- Only one record stop event per record start event may be output. Multiple record start events without any record stop events will result in data corruption.
- The record bits are only valid when data valid is asserted.
- The record bits are asserted in the same data clock cycle for records which consist of one data word (the minimum record length).
- The record bits may be asserted for multiple cycles. Only one of these cycles must have data valid asserted.
- For the *multi-record* data acquisition mode, the *record length* must be preserved. For the *triggered streaming* data acquisition mode, the length may be modified. Refer to the ADQ7 manual [4] for information on the data acquisition modes.
- For the multi-record mode, the *number of records* must be preserved. For the triggered streaming mode, records may be generated or discarded in the second user logic area.

`insert_record_bits(signal, channel)` *RR*

Insert the 2-bit record bits signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_record_bits(channel)` *RR*

Extract the 2-bit record bits signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

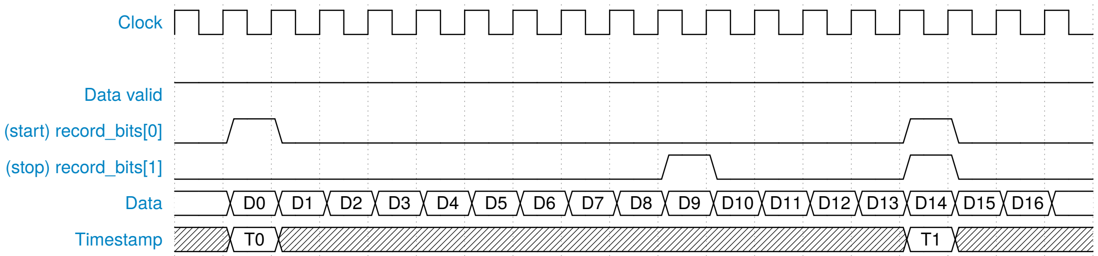

Figure 3: Timing diagram for the record bits when sample skip is disabled (skip factor 1). The figure presents two records. The first record spans ten data clock cycles and the second spans one data clock cycle.

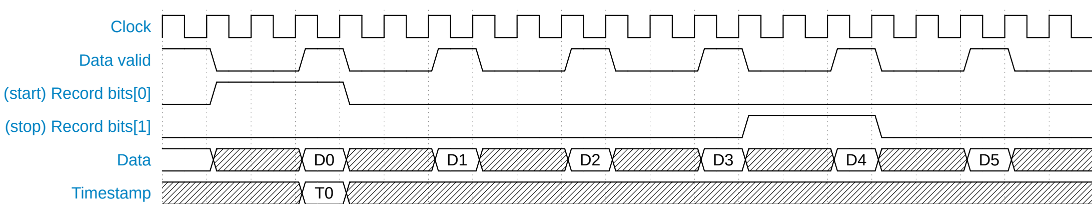

Figure 4: Timing diagram for the record bits when sample skip is enabled. The record spans five data clock cycles.

Record counter *RT: N/A, RR: per channel*

The *record counter* signal holds a 16-bit unsigned value tasked with keeping track of the number of records that have been created since the digitizer was last armed. Arming the digitizer is carried out by the calling the ADQAPI functions `StartStreaming()` or `ArmTrigger()`, depending on the data collection mode. The value is only valid when record start is asserted.

`insert_record_cnt(signal, channel)` *RR*

Insert the 16-bit counter signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_record_cnt(channel)` *RR*

Extract the 16-bit counter signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

User ID *RT: N/A, RR: per channel*

The *user ID* is an 8-bit signal with similar function to the general purpose signal. However, this signal may never be claimed for firmware-specific purposes and is reserved for the user. The value during a data clock cycle in which the record start bit is asserted will propagate to the user space in the host computer via the record header.

`insert_user_id(signal, channel)` *RR*

Insert the 8-bit user ID signal into the reduced rate data bus. The `signal` and the target

`channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_user_id(channel)` *RR*

Extract the 8-bit user ID signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Trigger event bit vector *RT: N/A, RR: per channel*

The *trigger event bit vector* is a signal of width equal to the number of parallel samples in the data word. Each bit in the signal is associated with the sample with the same *index* and indicates

- the *presence* of a trigger event with a logic high level and
- the *absence* of the trigger event with a logic low level.

This signal is *only* connected to the level trigger. Hence, it is only active when the level trigger is configured and enabled. This holds true regardless of whether or not the level trigger is used for data acquisition.

The signal is used to distinguish multiple events in one data clock cycle. Due to the nature of the level trigger, only *every other* sample may indicate a trigger event in the worst case.

Example

If bit 0 is asserted, the trigger condition is fulfilled for the sample at index 0 in the data word.

`insert_ch_trig_tevent_bitvect(signal, channel)` *RR*

Insert the trigger event bit vector signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_trig_tevent_bitvect(channel)` *RR*

Extract the trigger event bit vector signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Reset event bit vector *RT: N/A, RR: per channel*

The *reset event bit vector* is a signal with the same principal behavior as the trigger event bit vector, apart from tracking reset events instead of trigger events. The signal is only active when the level trigger is configured and enabled.

`insert_ch_trig_revent_bitvect(signal, channel)` *RR*

Insert the reset event bit vector signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_trig_revent_bitvect(channel)` *RR*

Extract the reset event bit vector signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Auxilliary trigger *RT: single, RR: single*

The *auxilliary trigger* is a collection of the same types of trigger-related signals as the channel trigger but operates independently. Additionally, there is only *one* auxilliary trigger signal on the data bus, as opposed to one per channel. The auxilliary trigger is configured by calling `SetAuxTriggerMode()`.

The purpose of this additional trigger signal is to serve custom designs which require that two separate trigger sources are observed. The enclosing ADQ7 design does not use the auxilliary trigger and neither is it forwarded to the host computer. It is a signal with the specific purpose of stimulating custom logic. Table 6 presents the collection of auxilliary trigger signals and their corresponding bus interface.

Table 6: This table presents the signals that together constitute the auxilliary trigger. The table is divided into two sections: one for signals common between the trigger definitions on the real-time data bus and the reduced rate data bus, and one for signals unique to the latter.

Signal	Bus interface
Common signals	
Trigger event	`insert_aux_trig_tevent(event)` `extract_aux_trig_tevent(DONT_CARE)`
Trigger event edge	`insert_aux_trig_trising(edge)` `extract_aux_trig_trising(DONT_CARE)`
Trigger event number	`insert_aux_trig_tnum(number)` `extract_aux_trig_tnum(DONT_CARE)`
Reset event	`insert_aux_trig_revent(event)` `extract_aux_trig_revent(DONT_CARE)`
Reset event (pretrigger)	`insert_aux_trig_revent_pt(event)` `extract_aux_trig_revent_pt(DONT_CARE)`
Reset event edge	`insert_aux_trig_rrising(edge)` `extract_aux_trig_rrising(DONT_CARE)`
Reset event number	`insert_aux_trig_rnum(number)` `extract_aux_trig_rnum(DONT_CARE)`
Reduced rate signals	
Trigger extended precision	`insert_aux_trig_extended_precision(precision)` `extract_aux_trig_extended_precision(DONT_CARE)`

5 User Logic 1

The first user logic module uses the real-time bus definition (Section 4.5) to describe the composition of the data bus. At this point in the data path (Fig. 1), there is no data valid signal present since every data clock cycle is considered valid. Thus, the enclosing design expects any custom design to output valid data on each data clock cycle.

Important

The first user logic area expects valid data to be output on each data clock cycle. There is no data valid signal at this point in the data path.

Note

The file `source/user_logic1_defines.vh` defines constants to use when parametrizing a design in the first user logic area.

5.1 Linear Phase FIR Filter

By default, the first user logic area contains example code implementing an order 16 linear phase FIR filter. Fig. 5 presents a block diagram of the top-level module. The top-level design demonstrates

- how to interface with the data bus (Section 4.5),
- how to modify the register map to add new control bus registers (Section 4.4) and
- how to perform clock domain synchronization (Section 4.3) of the register values forwarded to the filter's configuration interface.

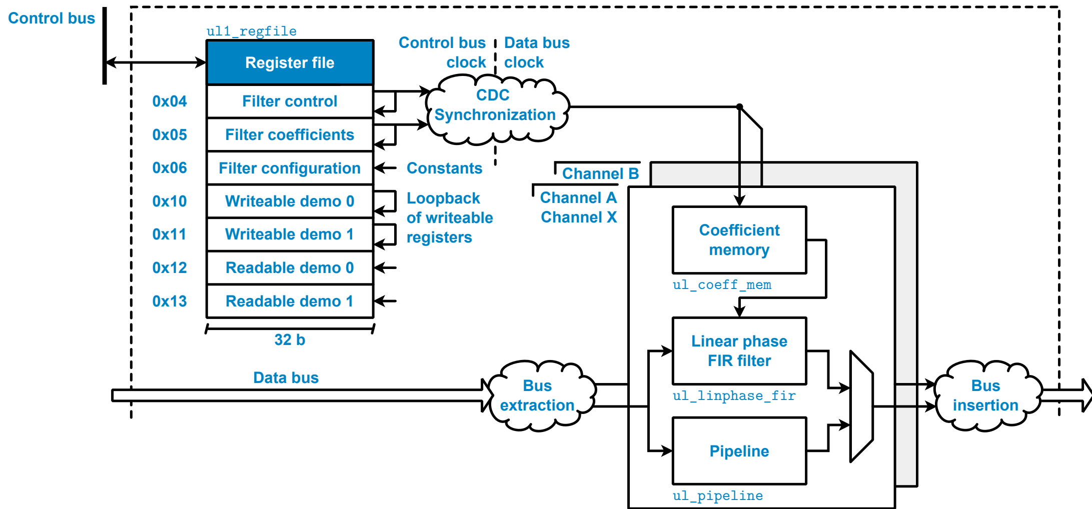

Figure 5: A block diagram of the default design present in the first user logic area. The top level contains a linear phase FIR filter of order 16.

The filter implementation is a heavily parametrized polyphase decomposed linear phase FIR filter and thus demonstrates the concept of parallel design (Section 4.1). The top level module defines parameters that may be used to change the filter properties, e.g. the filter order, coefficient width and the coefficient fixed-point representation. However, there are limitations. For example, the filter order *must* be an even number in order for the group delay to be an integer. Additionally, increasing the coefficient width will require modification of the control bus interface and its CDC structures. In most cases, the design should provide error messages upon compilation if an invalid parameter set has been provided. The filter design is separated into the following files:

`source/`

File	Description
`ul_coeff_mem.v`	Implements the filter coefficient memory. The sychronized register values are passed to this module.
`ul_linphase_fir.v`	Implements the filter top level. The module accepts the full-width data signal as input and responds with the processed data on the same format.
`ul_linphase_fir_unit.v`	Implements the processing branch in the polyphase decomposed filter structure.
`ul_linphase_fir_bs.v`	Implements a barrel shifter to compensate for the group delay of the filter.
`ul_add_macc.v`	Implements an add, multiply and accumulate module. The internal filter structure consists of several of these modules in cascade.
`ul_saturated_rounding.v`	Implements saturated rounding for the filter output signal.
`ul_dsp_primitive.vh`	Helper file instantiating a DSP48E2 primitive when targeted by an `` `include `` statement.
`ul_clog2.vh`	Helper file defining the ceiled $\log_2$ function.

5.2 Using MLVDS in MTCA Backplane

The control of MLVDS in the MTCA backplane is given to the user, using ports `mlvds_rx_i`, `mlvds_tx_i`, `mlvds_rx_o` and `mlvds_tx_o`. The default user logic code only relays the information from the trigger module. The direction is set by SetDirectionMLVDS API, and all 8 MLVDS can either be configured as input (default) or output, individually.

The information from the trigger module to the MLVDS outputs are using ports `mlvds_rx_o_from_datatrig_i[3:0]` and `mlvds_tx_o_from_datatrig_i[3:0]`.

> **Note**
> These can also be used to transport the configured output trigger information back to User Logic 1, regardless of using the actual MLVDS or not.

The mapping of ports on User logic 1 vs the MLVDS are given by the following table:

Table 7: MTCA MLVDS mapping to User logic 1

To/From physical pins	To/From User Logic	Direction
RX17	`mlvds_rx_i[0]`	Input
RX17	`mlvds_rx_o[0]`	Output
RX18	`mlvds_rx_i[1]`	Input
RX18	`mlvds_rx_o[1]`	Output
RX19	`mlvds_rx_i[2]`	Input
RX19	`mlvds_rx_o[2]`	Output
RX20	`mlvds_rx_i[3]`	Input
RX20	`mlvds_rx_o[3]`	Output
TX17	`mlvds_tx_i[0]`	Input
TX17	`mlvds_tx_o[0]`	Output
TX18	`mlvds_tx_i[1]`	Input
TX18	`mlvds_tx_o[1]`	Output
TX19	`mlvds_tx_i[2]`	Input
TX19	`mlvds_tx_o[2]`	Output
TX20	`mlvds_tx_i[3]`	Input
TX20	`mlvds_tx_o[3]`	Output

6 User Logic 2

The second user logic module uses the reduced rate bus definition (Section 4.5) to describe the composition of the data bus. At this point in the data path (Fig. 1), there is a data valid signal present and the user may modify the output data stream by modulating this signal. However, doing so in a *dynamic* manner—i.e. creating records of varying sizes—is only supported by the *streaming*-type data collection modes. Refer to the section describing the record bits for the requirements of these framing signals.

Important

It is crucial that the record framing signals output from the second user logic area have the correct behavior with respect to the data valid signal.

Note

The file `source/user_logic2_defines.vh` defines constants to use when parametrizing a design in the second user logic area.

The second user logic module contains the logic for computing the pulse metadata. The metadata specification is documented in the ADQ7-FWPD user guide [2]. Fig. 6 presents a block diagram of the top-level design. The top level contains mostly bus operations and routing. For each analog channel, a *characterization module* is instantiated and while there is a common register map for the top level, most of the registers are decoded inside the characterization module. This implies that each module has its own base address on the control bus. The channel base address are listed in Table 8.

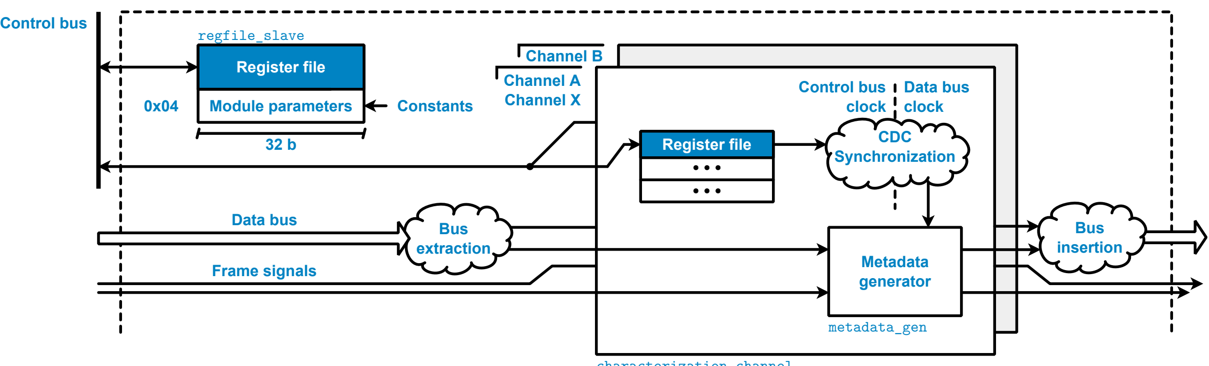

Figure 6: A block diagram of the default design present in the second user logic area. The top level instantiates one characterization module per analog channel to extract pulse metadata.

Table 8: Base address for registers in the `characterization_channel` module.

Channel	Base Address	High Address	Size (32-bit words)
Channel A/X	`0x40000`	`0x43FFF`	16384
Channel B	`0x60000`	`0x63FFF`	16384

6.1 Interface

This section describes the interface of the `user_logic2` module. Refer to Section 4.5 for information on how to parse the signals `s_axis_tdata` and `m_axis_tdata`. The DRAM and GPIO signals are not included in this list.

`s_axis_aclk` — *Input, 1 bit wide*

The 312.5 MHz data clock.

`s_axis_aresetn` — *Input, 1 bit wide*

Data bus reset signal. Synchronous to `s_axis_aclk`, active low.

`s_axis_tdata` — *Input, UL2_DATA_BUS_WIDTH bits wide*

The data bus input. See Section 4.5.

`padding_offset_start_i` — *Input, UL2_SPD_PROCESSING_CHANNELS bits wide*

Asserted during the first clock cycle of the padding offset.

`padding_offset_stop_i` — *Input, UL2_SPD_PROCESSING_CHANNELS bits wide*

Asserted during the last clock cycle of the padding offset.

`detection_window_start_i` — *Input, UL2_SPD_PROCESSING_CHANNELS bits wide*

Asserted during the first clock cycle of the detection window.

`detection_window_stop_i` — *Input, UL2_SPD_PROCESSING_CHANNELS bits wide*

Asserted during the last clock cycle of the detection window.

`padding_offset_start_o` — *Output, UL2_SPD_PROCESSING_CHANNELS bits wide*

Asserted during the first clock cycle of the padding offset.

`padding_offset_stop_o` — *Output, UL2_SPD_PROCESSING_CHANNELS bits wide*

Asserted during the last clock cycle of the padding offset.

`m_axis_tdata` — *Output, UL2_DATA_BUS_WIDTH wide*

The data bus output. See Section 4.5.

6.2 Metadata

The metadata is generated by the `metadata_gen` module. The input and output signals are described below. The number of parallel samples depends on the firmware type. *PS* is used to denote parallel samples and *PP* denotes parallel packets. A block diagram of the `metadata_gen` module is presented in Fig. 7.

6.2.1 Interface

The input and output signals of the metadata generation modules is listed below.

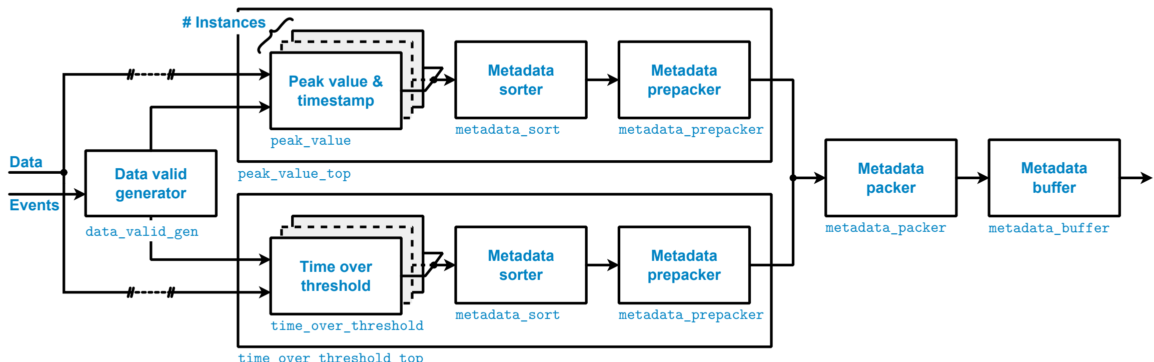

Figure 7: A block diagram of the metadata generator. The number of instances depends on the firmware.

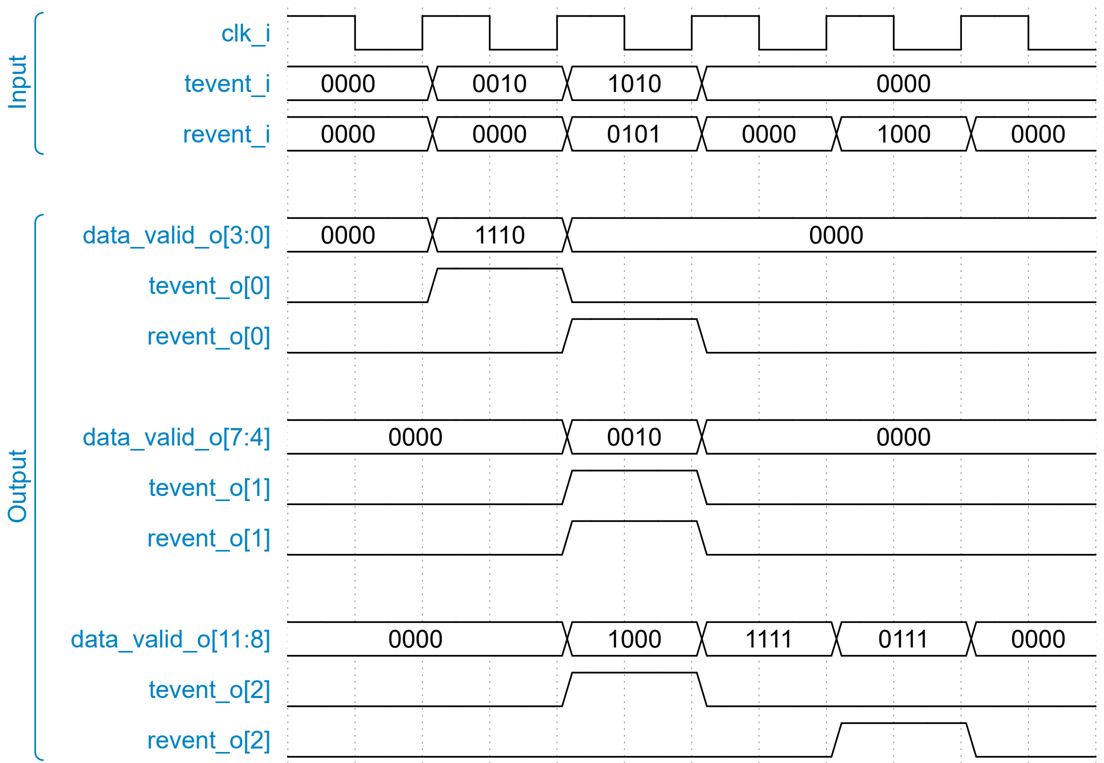

Figure 8: Timing diagram for the `data_valid_gen` module. The latency of the module is not pictured. The number of parallel samples is 4 and thus, $4/2 + 1 = 3$ instances are required to support the worst case pulse rate.

`tevent_i` *Input, PS bits*

Asserted when a trigger event occurs. The signal is parallel samples bits wide and each bit corresponds to one sample. For example, if bit 3 is set, the trigger event occurred on the forth sample in the current data clock cycle. In other words, the forth sample is the first sample in the pulse. Note that multiple trigger events may occur in the same cycle.

`revent_i` *Input,* 16 · *PS bits*

Asserted if a reset event occurs. The signal is parallel samples bits wide and each bit corresponds to one sample. The reset event is not included in the pulse. For example, if bit 3 is set, the first three data samples are included in the pulse. Note that multiple reset events may occur in the same data clock cycle.

`data_i` *Input,* 16 · *PS bits*

The input data signal. Multiple samples are received in parallel each data clock cycle. Bits 0 to 15 contains the first (earliest) sample, bit 16 to 31 the next and so on. The number of samples depends on the base design (see Section 4.1).

`detection_window_start_i` *Input, 1 bit wide*

Asserted during the first data clock cycle of a detection window. The current data clock cycle is *included* in the window.

`detection_window_stop_i` *Input, 1 bit wide*

Asserted during the last data clock cycle of a detection window. The current data clock cycle is *included* in the window.

`header_i` *Input,* $\leq$ *144 bits*

Collection of signals which are only valid when `detection_window_start_i` is asserted. Used for record header signals such as timestamp. This signal is sampled on `detection_window_start_i` and output on `header_o` when `detection_window_start_o` is asserted.

`tnum_i` *Input, 8 bits*

High precision value for `detection_window_start_i` measured in samples in `Q5.3` format (compare with the trigger event number). Only valid when `detection_window_start_i` is asserted. For example, `0x1a` means that `detection_window_start_i` occurred 3.25 samples into the current data word.

`tpol_i` *Input, 1 bit*

Configuration signal which controls the polarity of the pulse. The value determines the search direction for locating the peak value and its position. The `tpol_i` signal must not change while the acquisition is active.

`tot_saturate_on_max_i` *Input, 1 bit*

Configuration signal of the time-over-threshold overflow behavior. If asserted the time-over-threshold value will saturate at the maximum, otherwise the value will wrap around. The `tot_saturate_on_max_i` must not change while the acquisition is active.

`data_o` *Output,* 64 · *PP bits*

Metadata output, contains multiple metadata packets. The number of parallel packets depends on the firmware. The width of `data_o` cannot be modified.

`header_o` *Output, $\leq$ 144 bits*

Only valid when `detection_window_start_o` is asserted.

`data_valid_o` *Output, 1 bit*

Single bit to indicate that the complete data cycle is valid.

`detection_window_start_o` *Output, 1 bit*

Asserted during the first data clock cycle of the detection window. Only valid when `data_valid_o` is asserted

`detection_window_stop_o` *Output, 1 bit*

Asserted during the last data clock cycle of the detection window. Only valid when `data_valid_o` is asserted.

`tot_overflow_o` *Output, 1 bit*

Signals that the time-over-threshold counter has counted past its maximum value. Depending on the `tot_saturate_on_max_i` configuration, the counter has either saturated or wrapped around.

`metadata_buffer_overflow_o` *Output, 1 bit*

Signals an overflow condition in the buffer FIFO. Data has been lost.

`metadata_packer_overflow_o` *Output, 1 bit*

Signals that the dead time between detection windows requirement is violated. Data has been lost.

`metadata_alignment_error_o` *Output, 1 bit*

Signals that the metadata signals are not aligned. This is a design error and should never happen during operation.

6.2.2 Bus Signals

The latency through the metadata module depends on the data. The `BUS_PIPELINE` parameter cannot be used to pipeline the bus metadata signals (timestamp, record bits etc.) for the low-rate channels. To make it easier to add new signals, all bus metadata signals are concatenated into a single multi-bit signal `header_i`. The width of this signal is specified by the `HEADER_WIDTH` parameter. The width can be extended up to 144 bits by modifying the parameter value. To increase it further the Xilinx FIFO IP `fifo_144x512` in `metadata_buffer` must be extended as well.

Note

The `BUS_PIPELINE` parameter cannot be used to pipeline the bus metadata signals (timestamp, record bits etc.) for the low-rate channels.

6.2.3 Parallel Instances

The metadata computation is performed by multiple instances working in parallel. This is required to support multiple pulses each clock cycle. The number of instances is computed as

$$PS/2 + 1, \tag{1}$$

where PS is the number of parallel samples. The extra instance $(+1)$ is required to support the worst case where a pulse ends on the first sample and a new pulse begins on the last sample of a clock cycle. This is illustrated in Fig. 8.

6.2.4 Data Valid Generation

The first module in the `metadata_gen` block is the `data_valid_gen` module. This module translates the event vectors to multiple data valid vectors. Each of these vectors is fed to a separate metadata instance. A timing diagram for the `data_valid_gen` module is presented in Fig. 8.

6.2.5 Metadata Generation

The metadata is computed by two blocks:

- `time_over_threshold_top`
- `peak_value_top`

The `peak_value_top` computes the peak value and peak timestamp, and the `time_over_threshold_top` computes the time-over-threshold (pulse width). The two modules implement several computation instances to support multiple pulses per clock cycle. These blocks have the same data interface and computes the resulting metadata in parallel. A pipeline module is used to align the metadata to the worst latency.

6.2.6 Metadata Packer

The metadata computation block may output zero up to $PS/2$ valid packets each data clock cycle. The `metadata_packer` module buffers the data until a complete data clock cycle of valid data is available. The detection window stop signal will flush any available data and pad with zero data if necessary. A timing diagram for the `metadata_packer` module is presented in Fig. 9.

6.2.7 Metadata Buffer

The `metadata_buffer` has two tasks:

- convert the data width to match the data bus width, and
- buffer the data to support bursts of pulses.

The width of the data bus cannot be changed and is dependent on the number of parallel samples. The width is given by

$$16 \cdot PS, \tag{2}$$

where PS is the number of parallel samples. The maximum number of parallel metadata packets generated each clock cycle is $PS/2$. The metadata packet is 64 bits wide, which gives the total width

$$64 \cdot PS/2. \tag{3}$$

The ratio between (2) and (3) is

$$\frac{16 \cdot PS}{64 \cdot PS/2} = \frac{1}{2} \tag{4}$$

That is, the maximum number of packets that can be output is half of the maximum number that is generated each cycle. Therefore, a 2:1 width conversion is required. A FIFO is used to both buffer bursts of pulses and perform the width conversion.

6.2.8 Custom Metadata Logic

The metadata computation blocks are designed to support all different use cases within the specification. This may not be necessary when adding custom logic and imposing additional restrictions may simplify the design depending on the use case. For example, if the pulse rate is at most one pulse per data clock cycle only a single parallel instance is required.

In general, custom logic should never reuse existing registers for another purpose. If new registers are required for custom logic they should be added by increasing the `NOF_SLV_REGS` parameter.

> **Note**
> Custom logic should not reuse registers. Create new registers by updating the `NOF_SLV_REGS` parameter.

Listed below are a few different ways of implementing the custom logic. The difficulties listed are relative and assumes previous knowledge in Verilog and HDL design.

- Replace contents of `user_logic2.v` with a custom design.
  - Difficulty: easy to expert
  - Pros:
    - Limited knowledge of existing structure required.
    - Tailor-made solution for specific use case.
  - Cons:
    - Must design everything from scratch.
    - Existing ADQAPI functions used to configure the metadata engine cannot be used.
- Replace existing metadata computation module with a custom design.
  - Difficulty: medium
  - Pros:
    - Existing ADQAPI functions used to configure the metadata engine can be used.
    - No modification to the other parts of the design required.
  - Cons:

  * Limited to metadata width of replaced module.

- Add a custom module in parallel with existing metadata computation blocks.
  - Difficulty: hard
  - Pros:
    * Existing metadata still available
  - Cons:
    * Larger metadata packets
    * The modules `metadata_gen` and `metadata_buffer` must be updated to support larger data width.

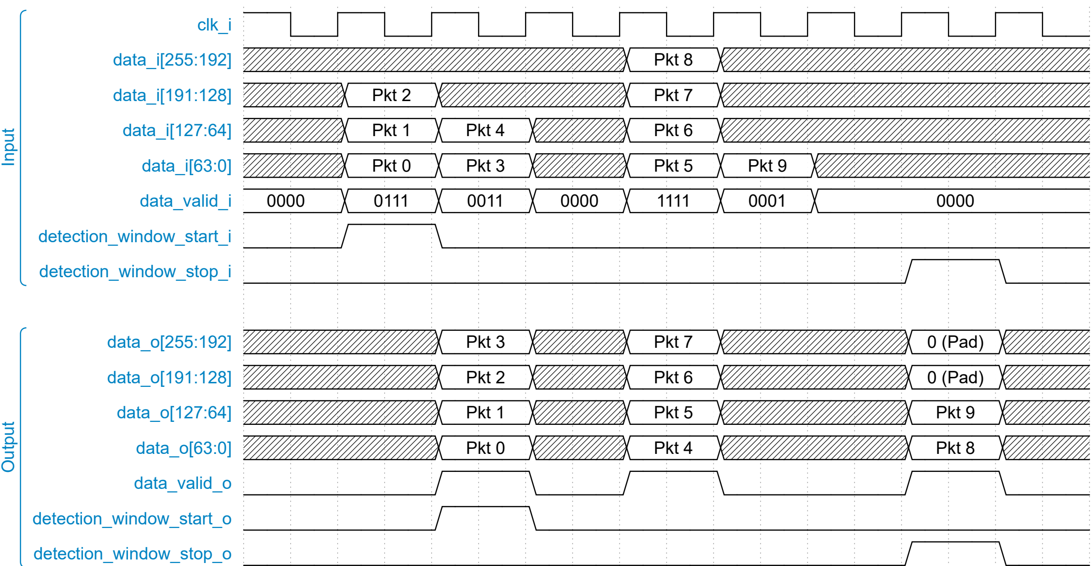

Figure 9: Timing diagram for the `metadata_packer` module. The latency of the module is not pictured. The number of parallel samples is 8.

7 GPIO

General purpose input and output pins (GPIO) are available in the second user logic area for PCIe and PXIe devices. The GPIO connector is presented in Fig. 10. The following pins are available:

- 12 bidirectional single-ended signals (3.3 V)
- 3 differential LVDS outputs
- 4 differential LVDS inputs
- 2 output supply pins. 3.3 V, max 250 mA.

The GPIO signals are routed from the physical pins through UL2 to the control bus. The interface signals to control the GPIO pins are presented in Table 9. The signals in the *To/From Host* column are read and written from the ADQAPI. The direction of the bidirectional pins are controlled by the `gpio_dir_o` signal. Each direction bit controls the direction of two GPIO pins: `gpio_dir_o[0]` controls GPIO0 and GPIO1, `gpio_dir_o[1]` controls GPIO2 and GPIO3 etc.

Table 9: GPIO signals in the second user logic area. The signals in the *pins* column is connected to the physical pins, and the signals in the *host* column is controlled by the host.

To/From pins	To/From Host	Width	Description
gpio_in_i	gpio_in_o	12	Input
gpio_out_o	gpio_out_i	12	Output
gpio_dir_o	gpio_dir_i	6	Direction signal. 1: input, 0: output
gpdi_in_i	gpdi_in_o	4	Differential input
gpdo_out_i	gpdo_out_o	3	Differential output

1	GPDIC0_P	LVDS input / clock	16	GND	31	GPIO0	Single-ended 3.3 V
2	GPDIC0_N	LVDS input / clock	17	GND	32	GPIO1	Single-ended 3.3 V
3	GPDI1_P	LVDS input	18	GND	33	GPIO2	Single-ended 3.3 V
4	GPDI1_N	LVDS input	19	GND	34	GPIO3	Single-ended 3.3 V
5	GPDI2_P	LVDS input	20	GND	35	GPIO4	Single-ended 3.3 V
6	GPDI2_N	LVDS input	21	GND	36	GPIO5	Single-ended 3.3 V
7	GPDIC3_P	LVDS input / clock	22	GND	37	GPIO6	Single-ended 3.3 V
8	GPDIC3_N	LVDS input / clock	23	GND	38	GPIO7	Single-ended 3.3 V
9	GPDO4_P	LVDS output	24	GND	39	GPIO8	Single-ended 3.3 V
10	GPDO4_N	LVDS output	25	GND	40	GPIO9	Single-ended 3.3 V
11	GPDO5_P	LVDS output	26	GND	41	GPIO10	Single-ended 3.3 V
12	GPDO5_N	LVDS output	27	GND	42	GPIO11	Single-ended 3.3 V
13	GPDO6_P	LVDS output	28	GND	43	VDD	3.3V 250mA
14	GPDO6_N	LVDS output	29	GND	44	VDD	3.3V 250mA
15	NC	NC	30	GND			

Figure 10: Schematic of the GPIO connector for PCIe/PXIe devices.

8 Troubleshooting

This section aims to provide guidance when troubleshooting unexpected behavior. It is recommended that the user application is written in a robust manner, able to capture and report error codes from failed ADQAPI function calls. In the event of a function call failure, reading the ADQAPI trace log for additional information is a useful first step. Trace logging must be activated by calling `ADQControlUnit_EnableErrorTrace()` with the `trace_level` argument set to 3.

If the error message is difficult to interpret, the Teledyne SP Devices support can be reached via e-mail at spd_support@teledyne.com. Please include information about the use case such as the pulse detection settings as well as the specification for both the trigger and data signals. Make sure to include a trace log file from a run where the error appears.

However, the support team *cannot* help the user with issues originating in the user's custom design in any of the user logic areas. Additionally, no training sessions on the topic of HDL design will be offered free of charge.

When facing a problem localized to the custom user logic design, Section 8.1 provides one possible way forward in those situations.

Important

Teledyne SP Devices' support cannot help with issues localized to the user's custom logic design nor offer training for HDL design concepts.

8.1 Debugging on Hardware

The section describes *one* possible workflow for setting up and connecting to a Xilinx debug core. Refer to the Xilinx documentation for further instructions. A good starting point is the *Vivado Programming and Debugging User Guide* [5].

Warning

Debugging on hardware requires physical access to the digitizer PCB.

8.1.1 Creating the Debug Core

1. Mark the signals as debug with the `mark_debug` property, for example in Verilog

   ```
   (* mark_debug = "true" *) wire signal_to_debug;
   ```

   Setting the `mark_debug` property makes the signals available in the debug wizard and ensures that the tool will not remove the signals in optimization.

2. Synthesize the design by:

   (a) Run the Tcl command

   ```
   devkit_synth_ul 1
   ```

   to generate a netlist for UL1.

   (b) Run the Tcl command

   ```
   devkit_synth_ul 2
   ```

   to generate a netlist for UL2.

   (c) Click on *Run Synthesis* and wait for Vivado to finish synthesizing the complete design.

3. Open the synthesized design by clicking on *Open Synthesized Design*.
4. Run the Tcl command

   ```
   refresh_design
   ```

5. Open the debug wizard by clicking on *Setup Debug* and follow the instructions.
6. Generate the bitstream by clicking on *Generate bitstream*.
7. When the process has finished, run the Tcl command

   ```
   devkit_mcs
   ```

8. Copy the generated files

   - `implementation/DevKit.runs/impl_1/debug_nets.ltx`
   - `implementation/adq7.mcs`
   - `implementation/adq7.bit`

   to a permanent location.

9. Program the firmware image (`.mcs` file) using ADQUpdater. Refer to the ADQUpdater user guide for instructions on how to manage the firmware on the ADQ7 digitizer [1].

8.1.2 Connecting to the Debug Core

The Vivado Hardware Manger is used to connect to the debug core. Connecting to the debug core requires that

- the `.mcs` file with core has been programmed and that
- the `debug_nets.ltx` file is available.

Depending on the clock signals chosen for the debug core, the firmware may have to be initialized before Vivado Hardware Manager can find the debug core. Initialization is done by calling the ADQAPI function `SetupDevice()`.

Important

The clock used for the debug core must be running for the core to function.

1. Connect the Xilinx platform cable to the digitizer's JTAG port.

2. Start Vivado and click on *Open Hardware Manager*.
3. Click on *Open Target* and chose *Auto Connect*.
4. In the trigger setup window, click on *Specify probe file and refresh device*.
5. Browse to the `debug_nets.ltx` file and click on refresh.

Refer to the *Vivado Programming and Debugging User Guide* [5] for further instructions.

9 The inner design of the Multiport DRAM

Multiport is essentially a multiple port interface towards the two DRAM controllers. It handles port arbitration, DRAM command generation and allows both read and write ports. The user logic 2 in ADQ7 has access to one writer port and one reader port on each of the two DRAM conntrolles, while the framework design at the same time also has a number of reader and writer ports.

The writer ports and reader ports respectively, share the same structure and have the same functionality. The only difference is how they are prioritized in their access to the DRAM.

Important

The DRAM is by default used as a FIFO for the data transfer to the PC. This FIFO must be disabled before acessing the DRAM from user logic 2. This is done by the ADQAPI command: SetStreamConfig()

Please be aware of that the data transfer will in this case only use a small FIFO and that this may cause data owerflow if data is generated faster than the readout to the PC.

9.1 Ports

Since multiport handles the port arbitration, it is also the master on both the reader port and writer port buses, i.e. it signals *I will read data this clock cycle* on the writer port, and *I am outputting valid data this clock cycle* on the reader port.

The memory space which is to be read from / written to is selected by the device communicating with the port, via address pointers and a strobe signal.

There are no FIFOs in the actual ports, they are effectively just interfacing between the FIFO in the device using the port, and the command/data FIFOs.

Something that should be noted is that the ports themselves run on the global memory clock in the FPGA, but the DRAM controller runs on its own DRAM clock. These run at the same clock rate but are separate clock networks. For write operations, the clock domain crossing happens in the command/data FIFOs. For read operations, there is no such FIFO in multiport, however, and the data is instead just clocked directly to the memory clock domain.

Both reader and writer ports support address wrapping. In the case of the writer port, the write address will keep wrapping from last to first address, until the write_last signal is asserted.

In the reader port there are two sets of addresses: high and low set up a memory area to wrap around, while first and last set up the start address and end address of the readout. Since the digitizers often use circular writing to memory areas until a trigger occurs, the typical use case is for the reader port is to set high/low to the edges of the circular buffer, set first to wherever the trigger.

Table 10 contains a description of the port signals.

9.2 Command mux and port arbitration

The command mux selects which port is allowed to input commands to the command FIFO. The mux contains state machines called `select` and `select_hot`, which are actually duplicates of each other but with different encoding (integer coded and one-hot coded respectively) in order to improve timing. These are used to select which port is current enabled.

Table 10: DRAM User interface

Signal name	Direction	Description
read_reset_i	Input	Reset signal
read_strobe_i	Input	Strobe in order to start a read operation
read_abort_i	Input	Assert to abort read operation (stop generating read commands)
read_first_addr_i	Input	First address of read operation
read_last_addr_i	Input	Last address of read operation
read_low_addr_i	Input	Low address of read operation (memory wrap)
read_high_addr_i	Input	High address of read operation (memory wrap)
read_sent_o	Output	Asserted when the port has finished generating read command, signaling that a new read operation can be strobed. Note that while all read commands have been sent, they may not have been processed yet (which is what the read_done_o output is used to indicate).
read_done_o	Output	Asserted when read operation is completed, all commands have been applied to the DRAM controller and all data has been output.
read_data_o	Output	Data port (256 bits)
read_firstdata_o	Output	Asserted during the first data word output (read_data_o) of a read operation
read_lastdata_o	Output	Asserted during the last data word output (read_data_o) of a read operation
read_wr_o	Output	Output valid signal for read_data_o
read_afull_i	Input	Almost full flag, assert to throttle the data output of the port (see also READ_AFULL_DEPTH)
write_reset_i	Input	Reset signal
write_strobe_i	Input	Assert to strobe address information for write operation
write_first_addr_i	Input	First address for write operation (32 bits)
write_last_addr_i	Input	Last address for write operation (wraps on this address) (32 bits)
write_done_o	Output	Asserted when write operation is done
write_data_i	Input	Data port (256 bits)
write_last_i	Input	Stops the write operation (assert synchronously with the last data word to be written). If this is not asserted, the write operation will wrap over the first/last address space.
write_empty_i	Input	Empty signal, assert to stop the port from reading data
write_read_o	Output	Read signal, write_data_i will be captured and written when this is asserted

There is a strict prioritization between ports (see the ordering in the overview block diagram). As soon as a higher priority port signals *not empty*, the `select` register changes value and the mux starts accepting command from the new port starting with the next clock cycle.

9.3 Command / data FIFO

Data is read from / written to the DRAM using commands. The current multiport module only supports memory controller setups which produce one clock cycle of data per command.

As an example, the ADQ7 memory architecture has a 64-bit external bus, with a 1:4 memory controller giving 256 bits internally. The burst setting is also 1:4, resulting in a burst of four 64-bit accesses for each command, giving a single cycle of internal 256-bit data.

The ports automatically generate read/write commands across the space which the communicating device requested via strobe and address inputs

9.4 Tag FIFO

A write that is sent to the writer FIFO has no need to keep track of which port sent the write. A read however, needs to know where to send its results. That is what the tag FIFO is used for. At the same time as a command is sent to the command FIFO, a read port address is also entered into the tag FIFO. After the command has been sent, and the data returned from the DRAM, the tag is used to determine which port to send the read data to.

The tag FIFO also passes two additional bits, firstdata and lastdata, which are generated by the reader port to signal which data words are first and last in a read operation.

9.5 Parameter READ_AFULL_DEPTH

Each reader port is instantiated in multiport top with a parameter called READ_AFULL_DEPTH. The data chain for reading from DRAM looks like below, if we simplify away the other ports:

The data reader port will send out bursts of read commands into the command FIFO, and will not stop until the module which is connected to the read port sends its ”almost full” signal. However, since the command FIFO can contain several commands, this means that even though the read port stops sending more commands when the ”almost full” is received, there will still be some extra writes done depending on how many commands were already in the writer FIFO when the full signal was received. There is also a FIFO and pipelining in the DRAM + DRAM controller which can hold some pending commands

The READ_AFULL_DEPTH sets how many read commands the port is allowed to have in the writer FIFO and DRAM loop at any given time, before pausing and waiting for data to come back. This parameter should therefore be set to less than the remaining amount of rows in the receiving module FIFO when almost full is sent

On ADQ7 READ_AFULL_DEPTH is 128. It is recommended to add at least 8 to this for the almost full limit to account for delay in the DRAM controller.

9.6 A Note on Row Switches

The DRAM chips contain a number of banks. Each bank has a number of rows, which in turn has a number of columns. When data is to be accessed, the desired row is first cached in a row register (in

the chip), and the desired column is then read out to the DRAM controller.

Whenever a new row is accessed, the old row must be written back to memory, and the new one read out. This is called a row switch, and is fairly costly in terms of latency.

10 Using VHDL instead of Verilog

All code in the DevKit is Verilog. However, Vivado lets you mix Verilog and VHDL code without limitations, so you can choose to write your user code in VHDL.

Instantiate your VHDL module (see figure below – user_module_XXX.vhd) in the user logic code (user_logic1.v or user_logic2.v) in Verilog-style, and then you can write the modules in any of the languages you want. The tools will accept both Verilog and VHDL, the only important thing is that the instantiation interface is correct and same as the implemented. The simulation tools in Vivado will let you do mixed Verilog/VHDL simulation.

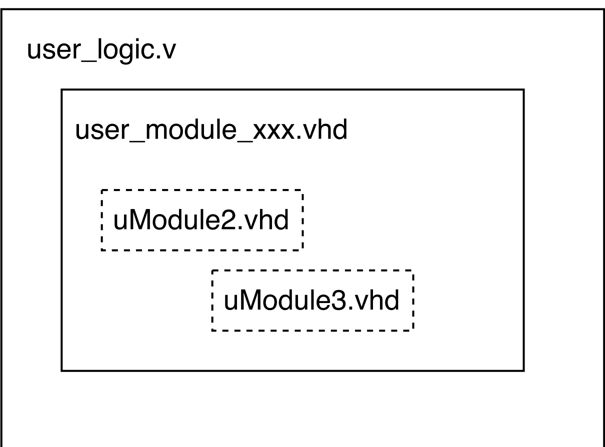

Figure 11: Hierarchy required to use VHDL

The top-level itself relies on macros (for the AXI bus extractions and insertions for instance) in Verilog, so it cannot be altered to VHDL. But after extraction and before insertion you can for instance pass on all ports (or the subset you need) to a submodule written in VHDL

References

[1] Teledyne Signal Processing Devices Sweden AB, *18-2059 ADQUpdater User Guide*. Technical Manual.

[2] Teledyne Signal Processing Devices Sweden AB, *18-2132 ADQ7-FWPD User Guide*. Technical Manual.

[3] C. E. Cummings, "Clock domain crossing (CDC) design & verification techniques using SystemVerilog," in *SNUG 2008 proceedings*, (Boston, MA, USA), Sunburst Design, Inc., 2008.

[4] Teledyne Signal Processing Devices Sweden AB, *16-1796 ADQ7 manual*. Technical Manual.

[5] Xilinx Inc., *Programming and Debugging*, June 2020. User Guide (UG908).

Worldwide Sales and Technical Support
spdevices.com

Teledyne SP Devices Corporate Headquarters
Teknikringen 8D
SE-583 30 Linköping
Sweden
Phone: +46 (0)13 645 0600
Fax: +46 (0)13 991 3044
Email: spd_info@teledyne.com

Copyright © 2022 Teledyne Signal Processing Devices Sweden AB

All rights reserved, including those to reproduce this publication or parts thereof in any form without permission in writing from Teledyne SP Devices.