TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ14 Development Kit

Documentation

Author: Teledyne SP Devices

SPD Document Number: 16-1830

Security Class: Open

Revision: PD

Release Date: 2017-11-08

Table of Contents

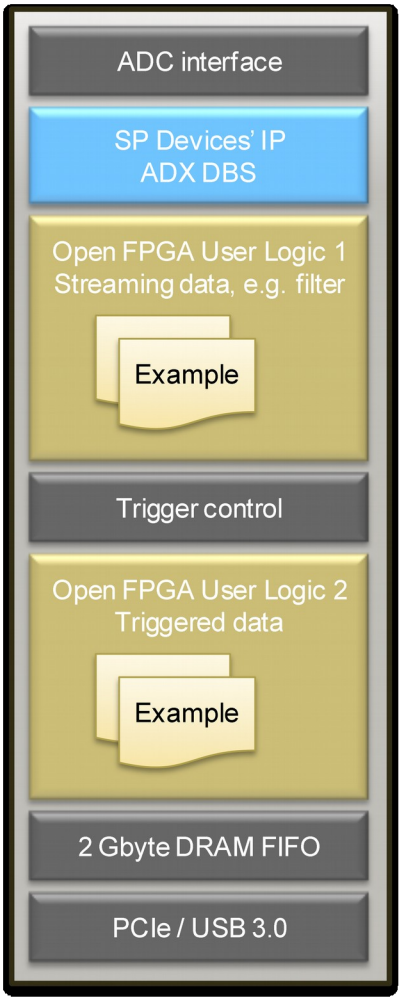
1 Tools

- A license for the ADQ14 Development Kit, purchased from SP Devices.
- A license of the Xilinx Design Tools. For current version of the ADQ14 Development Kit a license of Vivado 2015.2 is required. (see table below)
- The Vivado license includes simulation tools for mixed VHDL/Verilog code.
  Minimum required is Design Edition.
  (WebPack does not support the ADQ boards.)
  (Xilinx ISE cannot be used for ADQ14.)
- To create custom logic you need skills in Verilog or VHDL design.

Development Kit Revision	Tool version required
> r21902	Vivado 2015.2

2 Overview

2.1 High-level block overview



Standard signal conditioning functions

Data is streaming through. Place streaming functions like filters and data flow triggers here

Trigger data to form records

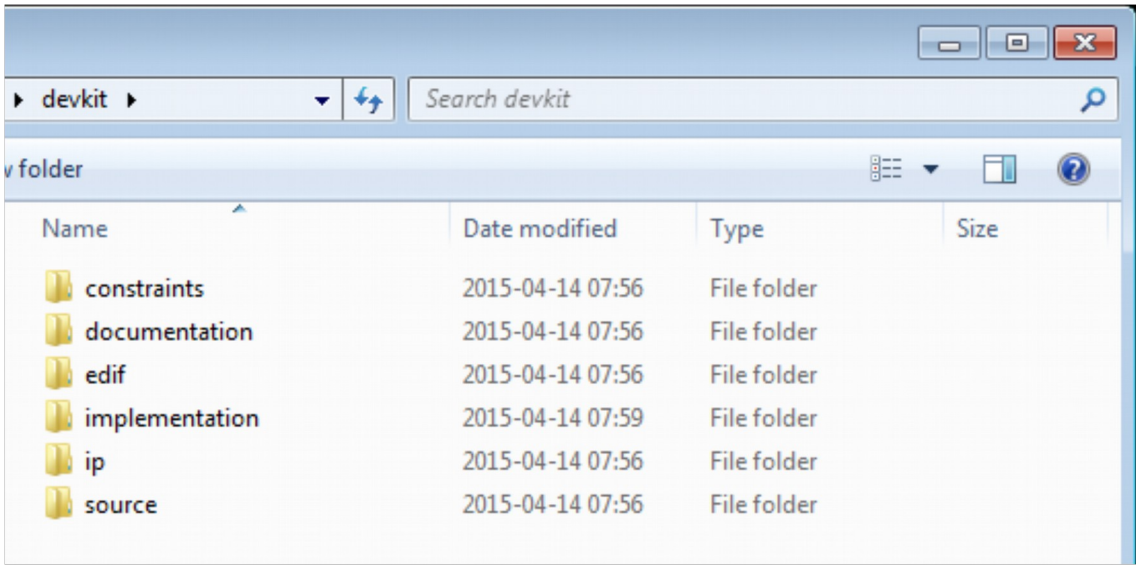
Data is packaged in records relative to a trigger. Place data records analysis like peak detection and data discrimination here.

Buffer data and send to the host PC using standard acquisition modes.

3 How to use the Development Kit

3.1 Extracting the DevKit files

Extracting the devkit .zip archive yields the folders seen below



The source files available for editing are user_logic1.v , user_logic2.v and regfile.v, and are found in the source folder. Do not modify any other files.

Pre-compiled code is placed in the edif directory

The ip directory contains Vivado IP cores

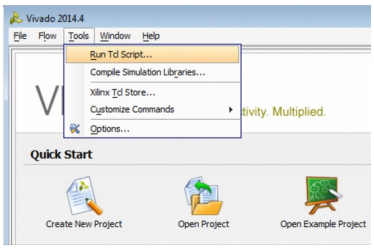
The scripts used to setup the project in Vivado and run the build process are placed in the implementation/scripts folder

NOTE: Extraction of the archive must be performed in a directory where you as a user have permission to write and change files. (For instance not under ”Program Files/SP Devices”)

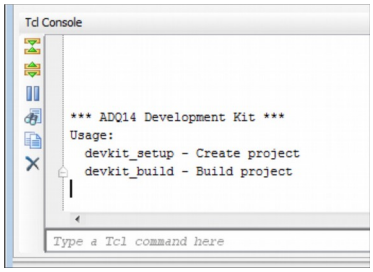
3.2 Open the Development Kit

Start Vivado (it can be found under Xilinx Design Tools in the start menu after installation)

In the Vivado menu select “Tools / Run Tcl Script”



Select the file: devkit\implementation\scripts\devkit.tcl. The Tcl Console will show the message below.

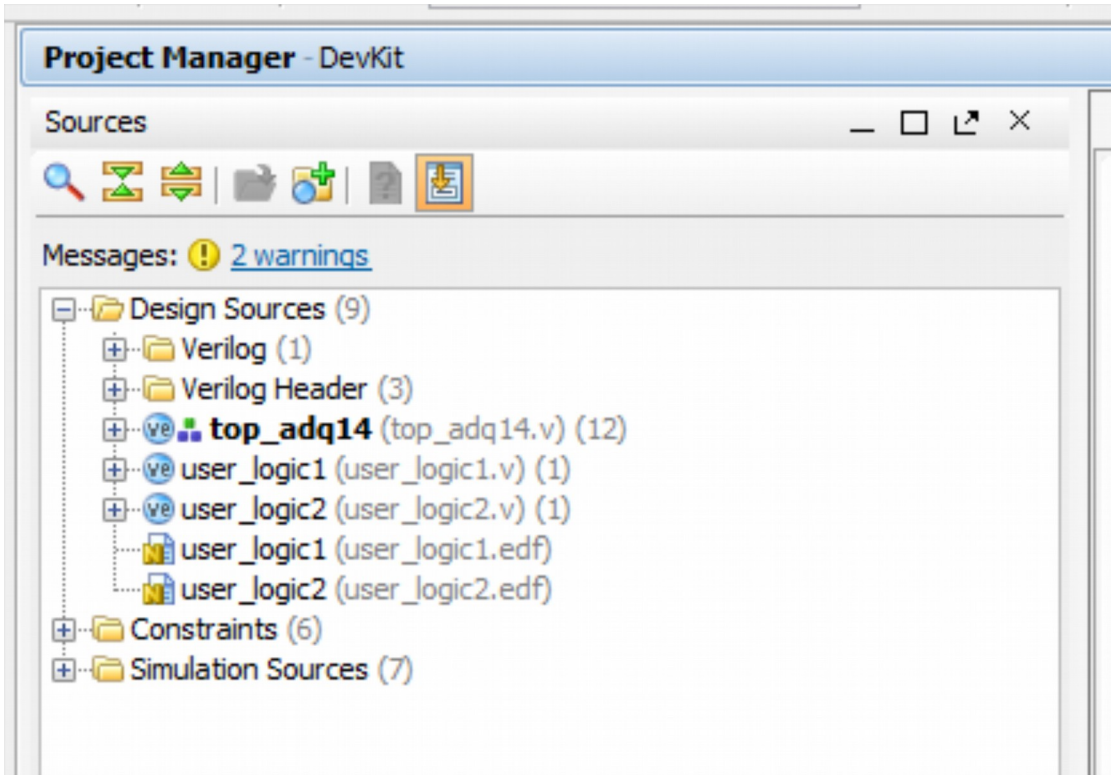


3.3 Set up the project

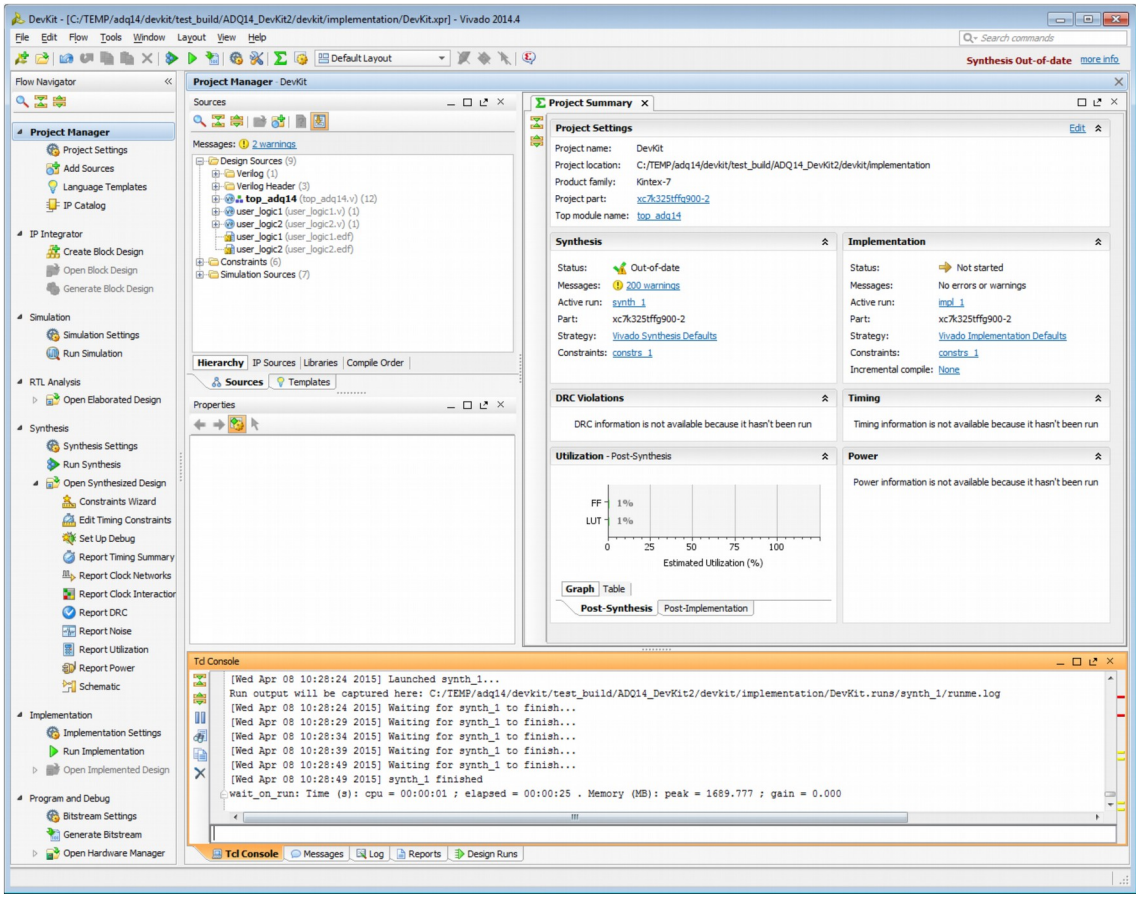
Go to the TCL Console command field and type: devkit_setup and press Return.

This will create the Vivado project. The initial setup will take a moment since parts of the design will be compiled.

When the execution has finished a project has been created.



Note: The user logic modules must be compiled into netlists. This is done by the provided scripts. Using the normal flow in Vivado instead of the provided scripts will not work.

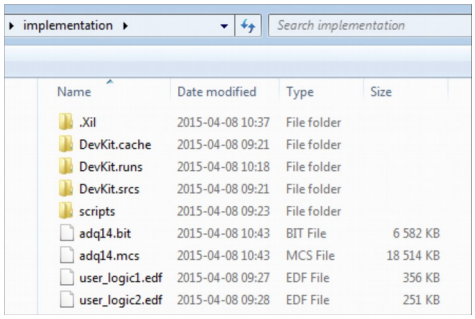


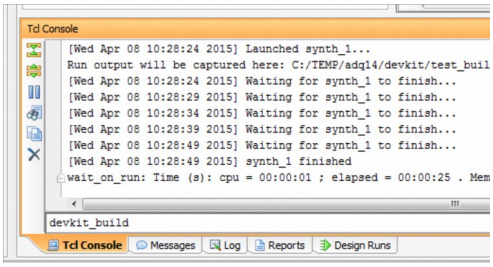
3.4 *Build the firmware bitfile*

Go to the TCL Console command field and type: devkit_build and press Return.

Dependent on your computer specifications and the complexity of the total logic (pre-compiled + your user logic) this may take between 1 hour to 10 hours.

When the execution has finished, an .mcs file that can be flashed to the digitizer have been created in the implementation folder.





You can manually rebuild the netlist for the user logic modules with the tcl commands:

devkit_synth_ul 1
devkit_synth_ul 2

Then use the Vivado GUI to generate the bitstream. To convert the bitstream to an .mcs file use the tcl command:

devkit_mcs

3.5 Working with your design

You can use the command below to set your user logic as top module. This is useful when using the Vivado RTL analysis tools.

devkit_set_top_ul 1
devkit_set_top_ul 2

You can go back to the standard devkit top module with the command:
devkit_set_top

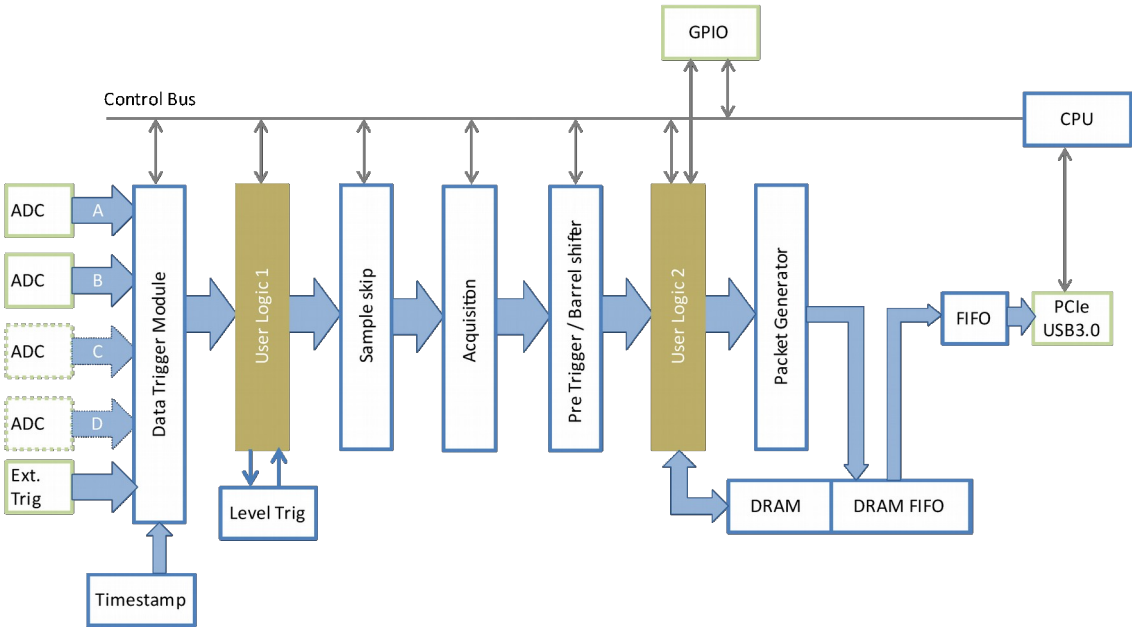
IMPORTANT: Avoid doing this manually. Other important parameters are also set by these commands.

3.6 Typical DevKit design flow

1. Set up the DevKit Project as described in section 3.3
2. Modify or insert new verilog code into user_logic1.v or user_logic2.v. This can be subdivided into 4 steps:
   a) Extract data, trigger and data_valid signals using the extract macros.
   b) Process the extracted data and signals according to your requirements.
   c) Insert the processed data, trigger and data_valid signals back into the data path.
   d) Set the correct BUS_PIPELINE delay to keep bus signals which were not manually inserted, in sync.
3. Generate the netlist for the modified code by running:
4. devkit_synth_ul 1 and/or
   devkit_synth_ul 2
5. Generate a .bit file by clicking on “Generate Bitstream” in Vivado GUI. Alternatively run: devkit_build in Vivado tcl console.
6. Generate a .mcs file to load into the onboard flash of the ADQ14 by running: devkit_mcs in Vivado tcl console. The generated .mcs file can be found under:
7. FPGA\implementation\xilinx\logfiles
8. Load the newly generated custom firmware file (.mcs) into the ADQ14 by using the ADQUpdater application, available via the SDK
9. Test the custom firmware using the ADQAPI library available via the SDK, using one of the many software examples as a basis for your software application

4 Basic design with the Development Kit

4.1 Dataflow chart



4.2 Control bus

Each User Logic modules has its own register control bus.

IMPORTANT: The first four 32-bit words are reserved for internal functions and cannot be used.

The example code implements a register bank. The bus can also be used to interface to block RAMs, FIFOs, or other custom blocks.

Signal	Description
clk	CPU clock
rst_i	Active high start up reset
addr_i	Read/write address 14 bits
wr_i	Active high write strobe
wr_ack_o	Active high write data qualifier.
wr_data_i	Write data 32 bits
rd_i	Active high read strobe
rd_ack_o	Active high read data qualifier
rd_data_o	Read data 32 bits

4.2.1 Accessing the block control bus from the ADQAPI

ADQAPI commands

Access single register:
ReadUserRegister()
WriteUserRegister()

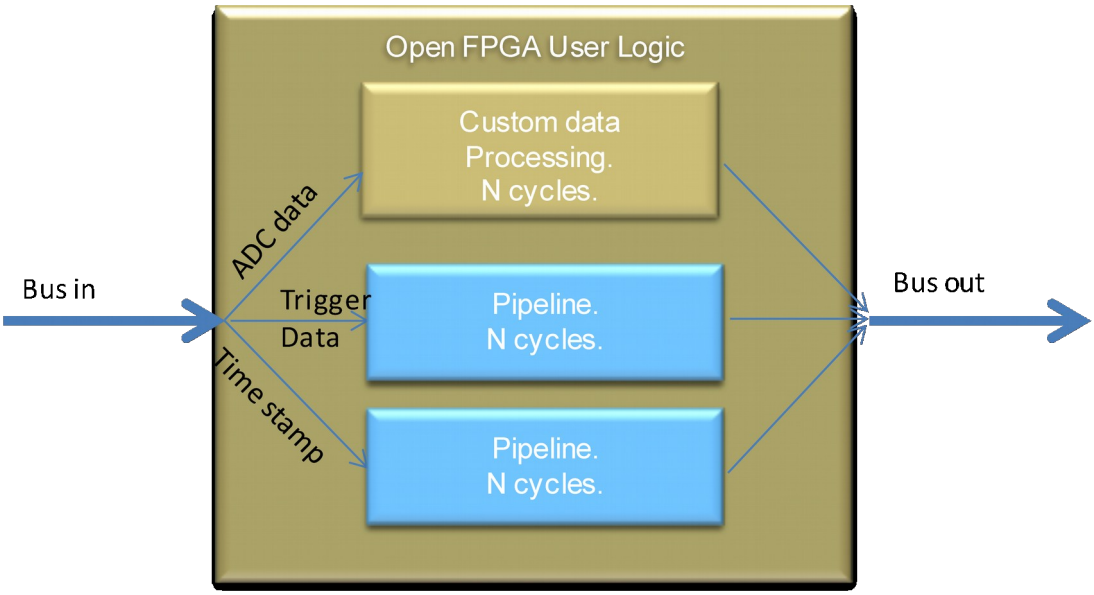
Write range of addresses:
ReadBlockUserRegister()
WriteBlockUserRegister()

For details see: ADQAPI_ReferenceGuide.pdf (Provided by the SDK installer)

4.3 Data stream bus

To keep information consistent through the digitizer system, the ADC data, trigger data and time stamp are provided as a combined stream bus. Therefore it is important that all these signals have the same latency through the User Logic designs.

There is a bus splitter macro in the user logic code that helps you to do this in a simple manner (see next section).



4.3.1 Concept of parallel samples (parallel design)

The FPGA can not be clocked at the sampling rate of the digitizer channels. A clock frequency on the FPGA of 200-300MHz is a good choice, to avoid timing issues. For instance for a -C unit with a sampling rate of 1GS/s this means we need to have 4 parallel samples for each clock. The user logic will receive 4 parallel samples and will need to output 4 parallel sample on each clock at 250MHz. This clock signal is denoted as an input port named “s_axis_aclk” alt. m_axis_aclk in the user_logic module and should be used to clock any custom data manipulation block.

Note: The clock frequency is 250MHz for the data stream. Either s_axis_aclk or m_axis_aclk can be used.
ADQ14-2X: 2 Channels, 2GSPs (8 parallel samples per cycle)
ADQ14-1X: 1 Channels, 2GSPs (8 parallel samples per cycle)
ADQ14-4C: 4 Channels 1 GSPs (4 parallel samples per cycle)

ADQ14-2C: 2 Channels, 1 GSPs (4 parallel samples per cycle)
ADQ14-4A: 4 Channels, 500 MSPs (2 parallel samples per cycle)
ADQ14-2A: 2 Channels, 500 MSPs (2 parallel samples per cycle)

4.3.2 Example of the bus splitter macro usage.

ADC data from channel A is extracted from the bus and delayed one clock cycle.

The signals from the bus that are not used are automatically passed through the module, with a delay equal to the “BUS_PIPELINE” parameter, which is set to 1 to match the data delay.

```
// Users to add localparam here
localparam BUS_PIPELINE = 1;
```

```
always @ (posedge clk_data)
  begin
     data_a0 <= extract_ch_a(0);
     data_a1 <= extract_ch_a(1);
     data_a2 <= extract_ch_a(2);
     data_a3 <= extract_ch_a(3);

  end


// User inserting into bus output
always@(*)
  begin
     init_bus_output();

     insert_ch_a(data_a0, 0);
     insert_ch_a(data_a1, 1);
     insert_ch_a(data_a2, 2);
     insert_ch_a(data_a3, 3);

     finish_bus_output();
  end
```

Note: Setting BUS_PIPELINE value correctly is critical to maintain valid data in the digitizer framework

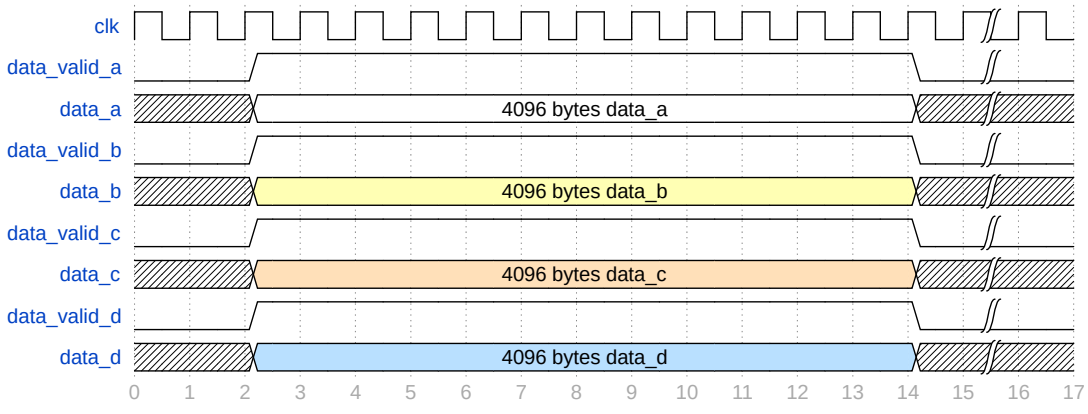
4.3.3 Raw data sample order on Host PC

Normally when using one of the available standard acquisition modes of the ADQ14, it is not necessary for the user to know about the sample order of the data arriving to the PC because most of the sorting and parsing work is taken care of by the API internally and the user is presented with the requested data in an easy-to-process format. However if one wants to implement a more customized acquisition with a customized sorting and parsing procedure on the software side, it is vital to understand how the data samples are ordered when it arrives to the PC. For a more hands-on detail, please study the included streaming devkit example with it’s accommodating software. Here are some of the important points.

Raw data is streamed to the PC and stored in so called “transfer buffers”. These transfer buffers can be setup by using the software API command “SetTransferBuffers()”. Data is written to these buffers in a circular manner. When all available buffers are filled, the transmission will halt, waiting for free buffers to continue. It is crucial that filled buffers are read out in time

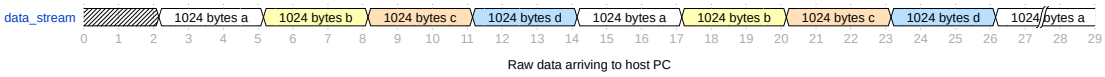
before this happens, otherwise buffer overflow will occur. Once data for a buffer has been read by the user, it will be put back into the queue to receive new data. The amount of buffers and the size of each buffer will have an impact on the data transfer speed from the ADQ14 to the host PC. Depending on the application, choosing a correct buffer size will save a lot of tedious parsing and sorting. Large transfer buffers will consume a lot contiguous space in the host's memory and might fail to be allocated due to RAM fragmentation over time (PC restart might help). Small transfer buffer size is easier to allocate but transfer speed might suffer because of the processing time overhead associated with each buffer. Once all the buffers are filled up but none has been read out, the data transfer will stall and hardware buffers on the FPGA side will start to fill up. When both the software and hardware buffers are filled, streaming overflow will occur, in which case the data will be corrupted. To avoid this, it is prudent to adapt the transfer buffer size, and amount, for the target application so that the buffer read out speed matches the write speed.

The software transfer buffers does not have any knowledge about what data it is receiving from the ADQ14 and it does not care. On the FPGA side, it is the data_valid signal for each channel data that will decide which data cycle will be transferred to the PC. However it is the software API that will have the last word on this. Even if data_valid for a channel is held high, it is still possible to prevent data for a certain channel from being transferred by masking out that channel using the API command SetStreamConfig(). Let's look at a simple illustration where we want to send 4096x4 bytes of data from a 4 channels ADQ14 to a PC. On the FPGA side, the channel data buses and corresponding data_valid signal should look like this:



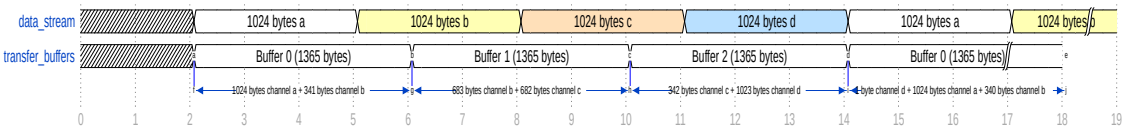
FPGA Data Transfer

Once these data arrive to the host PC they will be arranged in a stream like this:



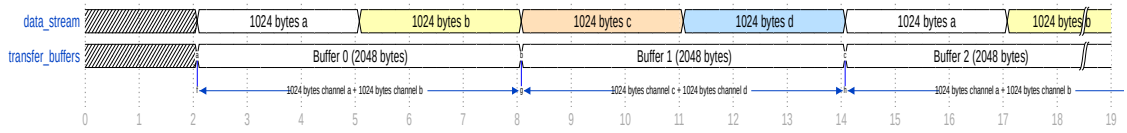
Raw data arriving to host PC

Because data from different channel is muxed into a single stream this way on the Host side, data belonging to one channel will be spread across different transfer buffers and must be separated and then stitched together before it can be interpreted properly. Choosing an appropriate transfer buffer size (using "SetTransferBuffer()") will make this parsing procedure less painful depending on your application. Let's look at an example when using 3 transfer buffers, each with the arbitrary size of 1365 bytes.



Raw data in transfer buffer with arbitary size

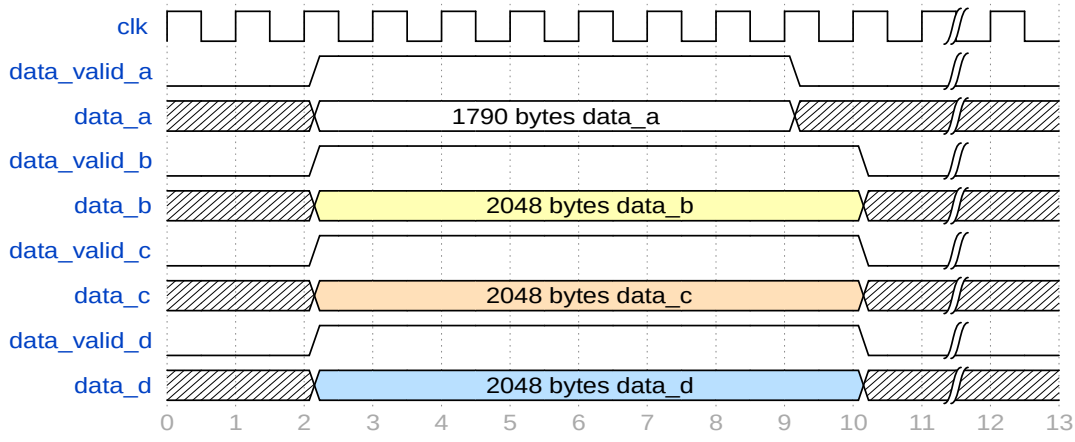
Iterating through the buffers with such size to separate the data into each channel is difficult because the pattern of the channel data in each buffer is not constant. It is therefore recommended to use buffer size in multiple of 1024 bytes in order to effectively parse the raw streamed data into separate channels. This depends however on the application and what needs to be done with the received data. There might exist cases where other buffer size would be more suitable.



Transfer buffer with 1024 bytes modulo size

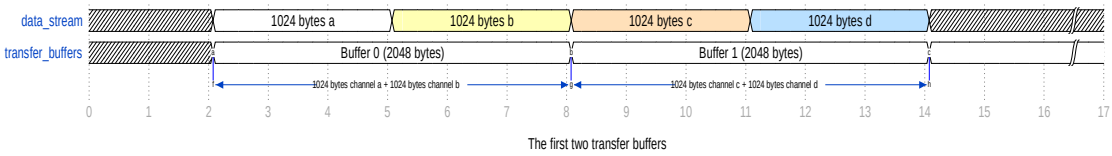
4.3.4 Streaming Finite Amount Of Data to Host

Streaming a finite amount of data from the digitizer to the host PC has one important limitation. If the amount of data is not enough to fill the 1024 bytes slot for ANY channel, data transfer will halt for all channels until that slot of 1024 bytes is filled. For example if we have setup a few transfer buffer with the size of 2048 bytes and the user_logic module on the FPGA is producing the following amount of data.



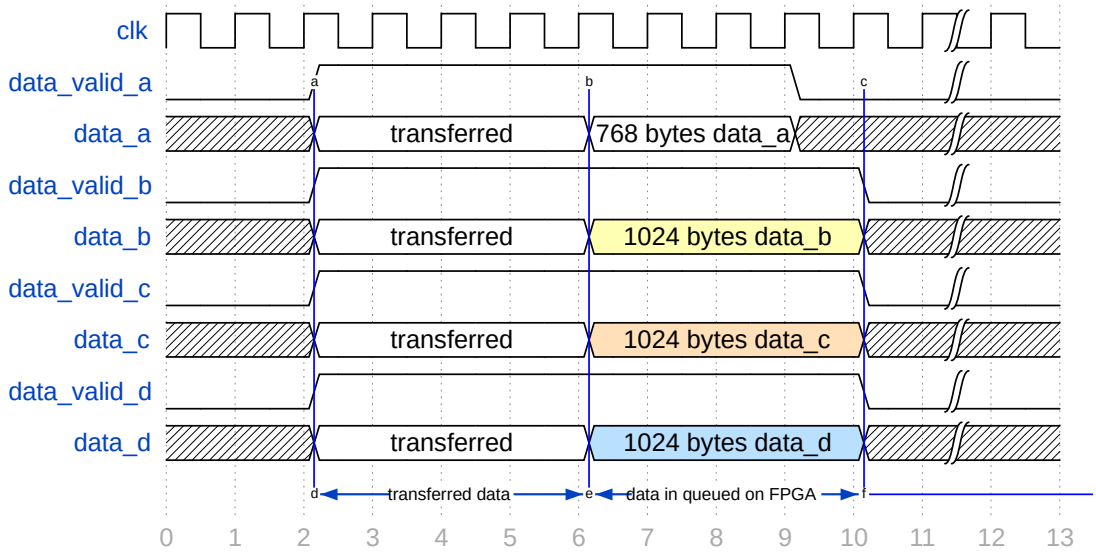
Example of different data amount for different channel

As can be seen in the figure above, data for channel A is a little less than the other channels. The first two transfer buffer on the host side will look like this.



The first two transfer buffers

Even though the FPGA is producing enough data to fill the third transfer buffer on the software side, the third buffer would still be empty because on the FPGA side, the transfer queue looks like this:



1024 bytes slot for channel a not filled. Transfer stuck in queue.

It does not matter which channel is missing some data to fill the 1024 bytes slot. When ANY channel cannot fill its slot of 1024 byte, transfer will halt and the hardware buffers will start to fill up until they overflows.

The simplest way to resolve this is to flush out the data by keeping the data_valid signal for channel A high for one more cycle so that the third and fourth transfer buffer can be filled. This means that the last 256 bytes of channel A will contain garbage data. But this data can easily be ignored on the software side when presenting the data to the user.

This design was mainly build to best suit the included standard acquisition modes and any other custom acquisition build by the user must adhere to this setup since it is not possible to construct a design that can anticipate every possible customization.

4.3.5 Trigger in User Logic 1

Each data channel in the bus has an associated trigger vector that can contain external trigger, software trigger, internal trigger etc.

The trigger mode selection (software, internal, external, etc) only decides which data is inserted into the trigger vector at the start of the data path, before the user logic modules. If you want to create your own trigger, you can simply ignore the incoming trigger vector to the

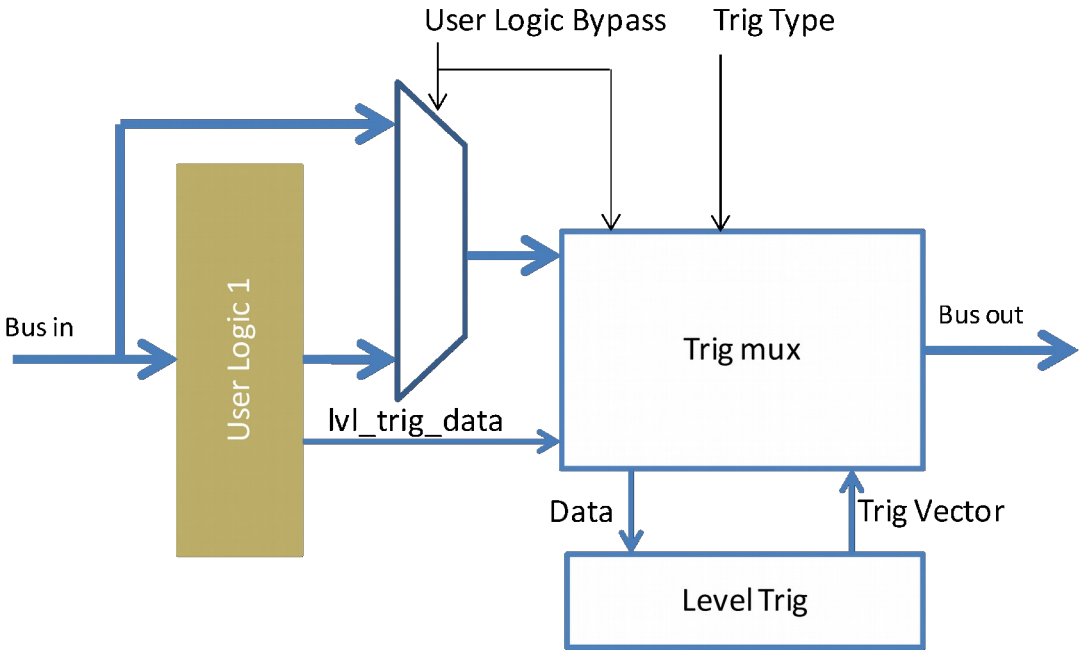
user logic, and create a new trigger vector of your own which you insert onto the outbound data bus.

The trigger mux in the User Logic module selects the trigger type used:

Incoming bus trigger
Level trigger

The user can modify the data that is sent to the level trigger in User Logic 1

When User Logic bypassed, data from User Logic 1 is ignored both for the level trigger and the channel data.



4.3.6 Commonly used bus splitter functions

Output bus	Input bus	Description
init_bus_output;	N/A	Must be placed before all insert_* functions
finish_bus_output ;	N/A	Must be after all insert_* functions
insert_<ch>_all(data)	extract_<ch>_all()	All parallel samples for one channel. Order D_n,...D_1, D_0
insert_<ch>(data, <n>)	extract_ch_<ch>(<n>)	n:th parallel sample for one channel
insert_data_valid_<ch>	extract_data_valid_<ch>	Data valid bit
insert_over_range_<ch>	extract_over_range_<ch>	Over range in data

<c> = [a,b] or [a,b,c,d] depending on product
<n> = 0-3,or 0-7, depending on product

Note:
ADQ14-2X: 2 Channels, 2GSps (8 parallel samples per cycle)
ADQ14-1X: 1 Channels, 2GSPs (8 parallel samples per cycle)
ADQ14-4C: 4 Channels 1 GSPs (4 parallel samples per cycle)
ADQ14-2C: 2 Channels, 1 GSPs (4 parallel samples per cycle)
ADQ14-4A: 4 Channels, 500 MSPs (2 parallel samples per cycle)
ADQ14-2A: 2 Channels, 500 MSPs (2 parallel samples per cycle)

4.3.7 GPIO (available in User Logic 2 only)

To/from physical pins	To/from CPU	Description
gpio_in_i	gpio_in_o	Input
gpio_out_o	gpio_out_i	Output
gpio_dir_o	gpio_dir_i	Data direction signal: 1 = Input 0 = Output
gpio_ctrl_in_i	gpio_ctrl_in_o	Ctrl input
gpio_ctrl_out_o	gpio_ctrl_out_i	Ctrl output
gpio_ctrl_dir_o	gpio_ctrl_dir_i	Data direction signal: 1 = Input 0 = Output

By default, the GPIO is connected to the CPU (i.e accessible over the API)

```
// GPIO
assign gpio_in_o    = gpio_in_i;
assign gpio_out_o   = gpio_out_i;
assign gpio_dir_o   = gpio_dir_i;

assign gpio_ctrl_in_o = gpio_ctrl_in_i;
assign gpio_ctrl_out_o = gpio_ctrl_out_i;
assign gpio_ctrl_dir_o = gpio_ctrl_dir_i;
```

4.4 Handling of the trigger vector

Each data channel has an associated trigger vector in the bus. The trigger information can be extracted by using the macro shown in the example below.

```
//Extract trigger vectors for channel A.
ch_trig_vector_a_in <= extract_ch_trig_a(DONT_CARE);
```

The trigger vector in User Logic 2 has 16 additional fractional bits compared to User Logic 1. This is to allow maintaining the full trigger precision when using sample skip or decimation.

When using the standard firmware, the trigger vectors for all channels are identical. Other firmware packages such as FWPD will let the channels operate with individual triggering (such as individual level trigger).

The trigger data is inserted prior to the user logic module and is selected from a number of trigger sources via the SetTriggerMode API command. If you want to create your own trigger, you can simply ignore the incoming trigger vector to the user logic, and create a new trigger vector of your own which you insert onto the outbound data bus.

Trigger vector bus width:

User Logic 1 = 7 bits
User Logic 2 = 23 bits

The trigger vector is divided into 3 fields. An example is shown below for the User Logic 2 trigger vector:

```
assign event_a = ch_trig_vector_a_in[22];
assign edge_a = ch_trig_vector_a_in[21];
assign value_a = ch_trig_vector_a_in[20:0];
```

The value-field should be interpreted as a fixed-point fractional number in units of samples in the current sample rate.

The fractional point is different for different ADQ14 products (Table for User Logic 2):

```
ADQ14-2X/1X: assign whole_samples = ch_trig_vector_a_in[20:18];
             assign fract_samples = ch_trig_vector_a_in[17:0];

ADQ14-4C/2C: assign whole_samples = ch_trig_vector_a_in[20:19];
             assign fract_samples = ch_trig_vector_a_in[18:0];

ADQ14-4A/2A: assign whole_samples  = ch_trig_vector_a_in[20];
             assign fract_samples = ch_trig_vector_a_in[19:0];
```

As an example, for the ADQ14-2X user logic 1 module, the trigger vector should be interpreted like this:

```
assign event = ch_trig_vector_in[6];
assign edge = ch_trig_vector_in[5];
assign whole_samples = ch_trig_vector_in[4:2];
assign fract_samples = ch_trig_vector_in[1:0];
```

The event bit indicates whether there was a trigger event (1) or not (0) during the current clock cycle. The edge bit is used to indicate rising(1)/falling(0) edge. It does not affect anything in the data acquisition. The remaining bits are a fixed-point fractional number, in units of ADC sample periods, that points to where the trigger occurred among the parallel samples of the current

clock cycle. The number 5.25 (10101 in binary) would for example mean that the trigger came a quarter sample period after the fifth ADC sample.

In user logic 2, the trigger vector has 16 additional fractional bits added to the end of the trigger vector, in order to be able to maintain trigger precision when using features such as sample skip and similar data reduction strategies. Apart from that, it works the same way as in user logic 1.

If you are using multirecord data collection, it is required that the trigger vectors for all channels are identical. If you are using triggered streaming, they can be made different.

5 Advanced design with the development kit

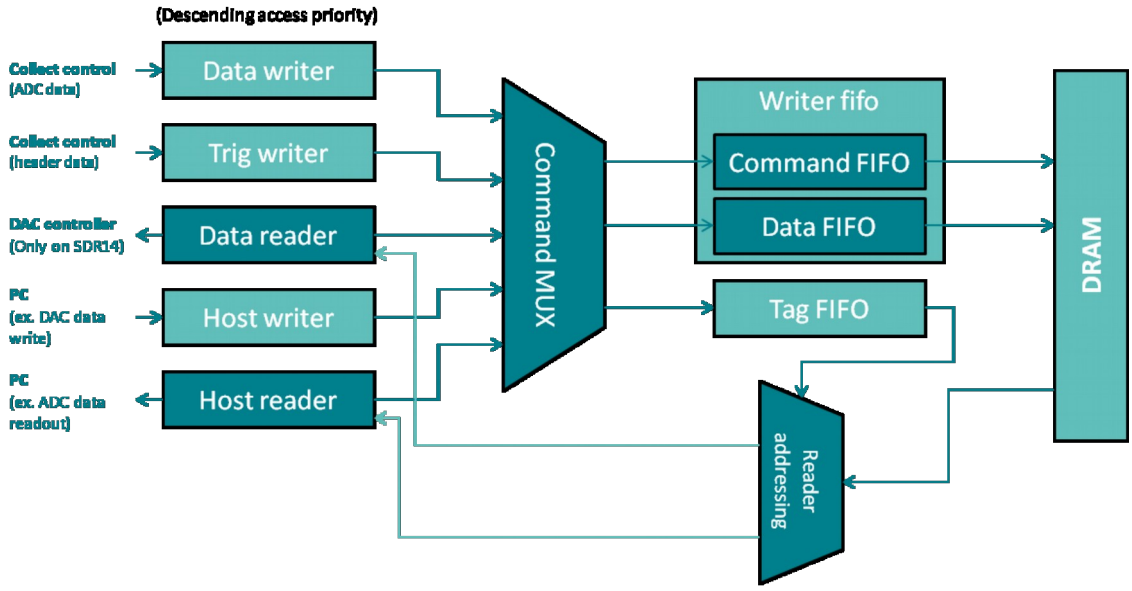
5.1 Using DRAM

5.1.1 The inner design of the Multiport DRAM

Introduction

Multiport is essentially a multiple port interface towards a single DRAM controller. It handles port arbitration, DRAM command generation and allows both read and write ports.

A block diagram view of multiport with 3 instantiated writer ports and 2 instantiated reader ports can be seen below. The user logic 2 in ADQ14 has access to one writer port and one reader port, 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.

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.

The reset input aborts any ongoing read or write operation, stopping the generation of DRAM commands and clearing the stored addresses.

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

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.

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.

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 ADQ14 memory architecture has a 64-bit external bus, with a 1:8 memory controller giving 512 bits internally. The burst setting is also 1:8, resulting in a burst of eight 64-bit accesses for each command, giving a single cycle of internal 512-bit data.

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

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.

5.1.2 User interface

Reader port (user interface in user_logic_2)

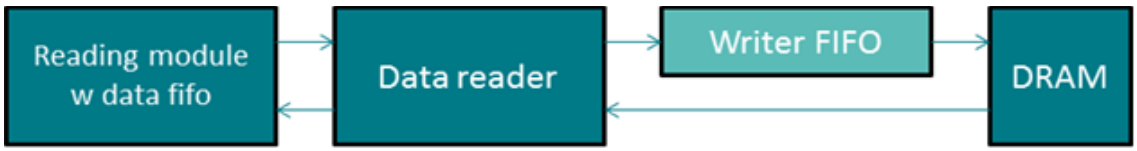
Signal name	Direction	Description
read_reset_i	Input	Reset signal
read_strobe_i	Input	Strobe in order to start a read operation
read_abort_i	Input	Assert to abort read operation (stop generating read commands)
read_first_addr_i	Input	First address of read operation
read_last_addr_i	Input	Last address 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 (512 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)

Writer port (user interface in user_logic_2)

Signal name	Direction	Description
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 (512 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

5.1.3 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.

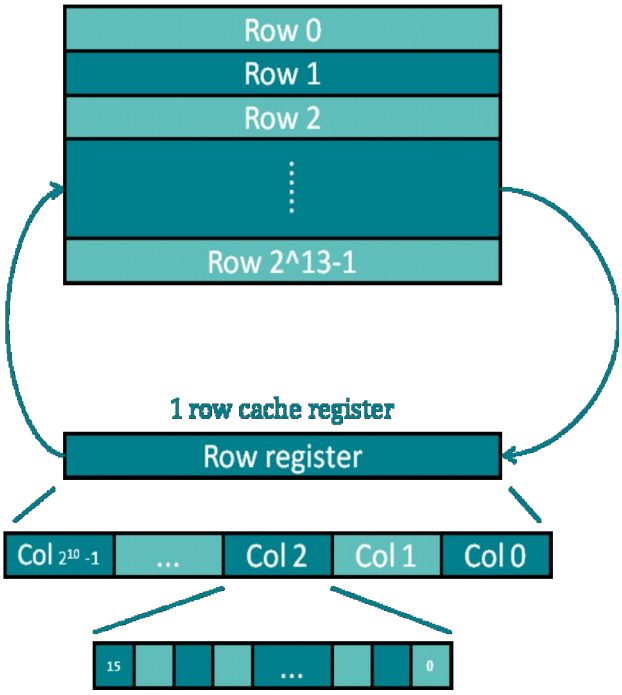
The READ_AFULL_DEPTH sets how many read commands the port is allowed to have in the writer FIFO and DRAM loop at any given time, before pausing and waiting for data to come back. This parameter should therefore be set to less than the remaining amount of rows in the receiving module FIFO when almost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

On ADQ14 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.

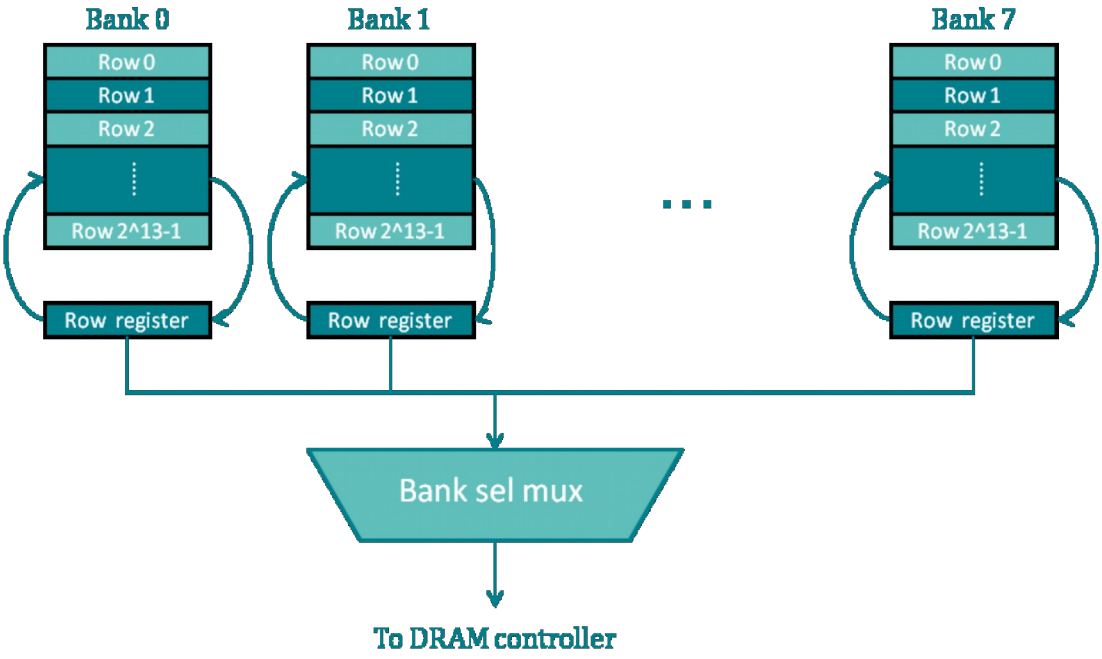
5.1.4 Other useful DRAM info

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.



The DRAM chips have a number of banks, typically eight:



Each bank has its own row cache register. It is therefore much faster to perform a bank switch than it is to perform a row switch.

5.2 Using GPIO on daughterboard

5.2.1 Description

The ADQ14 daughterboard has 12 GPIO pins (see attached pinout below), which are connected to the 12 LSBs of gpio_in/out. Each of these signals also has a buffer on the daughterboard. The 4 MSBs of gpio_out_o controls the direction of these buffers for the 4 LSBs of gpio_out_o. The rest (bit 4 to 11) is controlled using I2C communication. This should not be confused with the gpio_dir_o signal which controls the tristate buffers on the FPGA.

In conclusion:

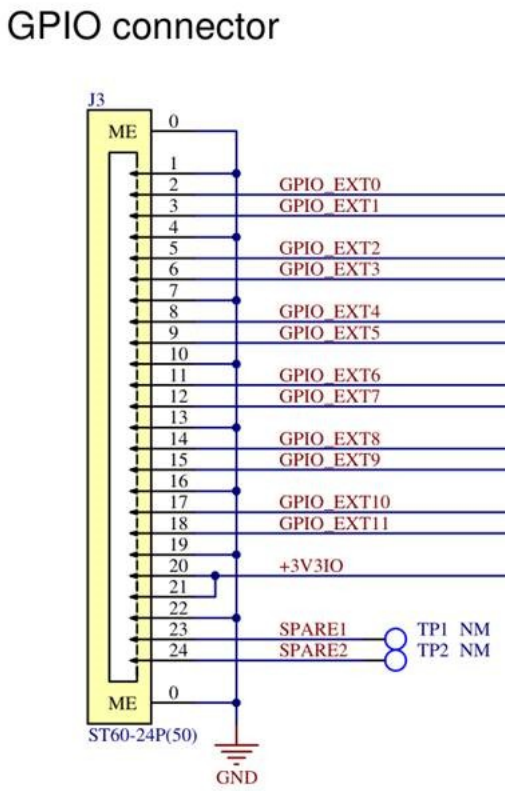
- gpio_in/out [0:11] is connected to GPIO_EXT0, ..., GPIO_EXT11
- gpio_out[12:15] controls the direction of the daughterboard buffers for gpio_in/out[0:3] signals
- The direction of the daughterboard buffers for gpio_in/out[4:11] is set using I2C commands (SetDirectionGPIOPort from the SDK)
- gpio_dir_o[0:15] controls the FPGA tristate buffer for gpio_in/out [0:15]

5.2.2 Usage

Recommended is using the SDK commands (SetDirectionGPIOPort) to control the GPIO input/output state, and bypassing the direction signals, i.e. let gpio_dir_o = gpio_dir_i and gpio_out_o[15:12] = gpio_out_i[15:12]. This is assuming you don't need to set the direction directly from the devkit. Otherwise, you have to ensure that the direction match for the FPGA and daughterboard buffers.

The gpio_ctrl[1] is an active low enable signal for the 3.3 V supply (pin 20 and 21) on the connector and gpio_ctrl[0] is the active low short circuit fault status bit for the power switch. The other gpio_ctrl bits are not connected to anything. It is recommended to control these using the API function EnableGPIOSupplyOutput and leave all gpio_ctrl signals connected as they are in the devkit.

Below is a screenshot of the GPIO connector. The voltage level of all GPIO pins is 3.3V



The 3.3 V supply allows for a maximum current of 0.5 A. The “SPARE” pins are not connected to anything.

5.2.3 Connectors

The connector on the board is the Hirose ST60-24P(30):

A mating cable connector is ST40X-24S-CV(30) from HiRose.

https://www.hirose.com/product/en/products/ST/ST60-24P(30)/

It is possible to order a 1m cable assembly with a ST40X-24S-CV(30) connector in each end from SP Devices to be used for the GPIO usage.

5.3 Using the record bits

The aim of this section is to explain the concept and usage of the record bits. The record bits exist on the data bus following the acquisition module. Therefore, this section is only relevant for user logic 2. The record bits are only used for triggered streaming. For raw streaming, only the `data_valid` signal is used to control the data.

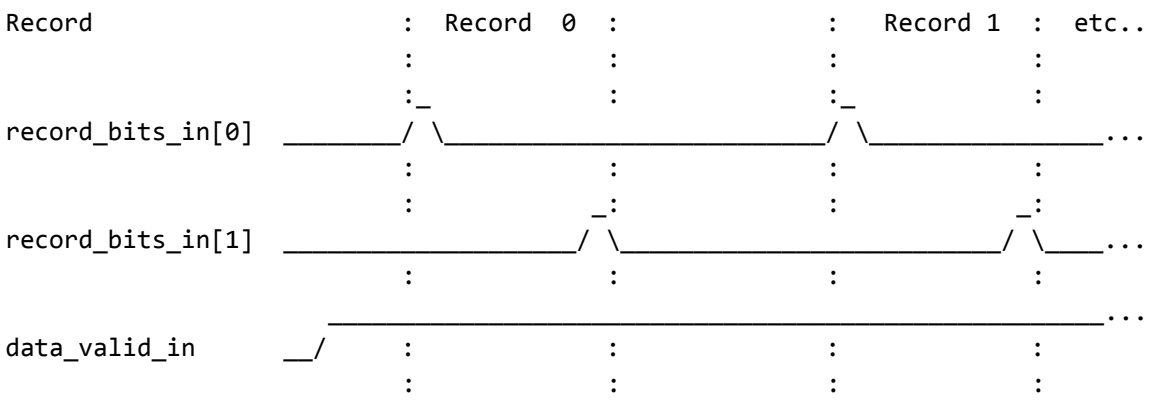


Figure 1: Simplified signal diagram for the record bits. Data valid is asserted when the device powers on, and is only de-asserted if sample skip ordecimation is used.

The data valid signal is used to indicate that the data for the current cycle is valid ADC data. This means the data valid signal will be constant high unless sample skip is used. Therefore, the data valid signal can not be used to mark or indicate records.

Instead, records are marked with the record bits. The record bits are a 2-bit vector that can be extracted from the bus.

- The first bit of the `record_bits_in` vector indicates the start of the record. The cycle where the first bit is asserted is also included in the record.
- The second bit of the `record_bits_in` vector indicates the end of the record. The cycle where the second bit is asserted is also included in the record.

5.3.1 Extracting the record bits

The record bits can be extracted from the bus with the bus macro

```
extract_record_bits_X
```

where X is the channel. For example to extract the record bits from channel A

```
wire [1:0] record_bits_in;
record_bits_in = extract_record_bits_a(DONT_CARE);
```

For example, a signal, `valid_record`, which is 1 during a record and 0 therwise, may be obtained by

```
reg valid_record_reg;
wire valid_record;
always@(posedge CLK_SIGNAL) begin
    if(record_bits_in[1]) begin
        valid_record_reg <= 1'b0;
    end
    else if(record_bits_in[0]) begin
        valid_record_reg <= 1'b1;
    end
end

assign valid_record = record_bits_in[0] | valid_record_reg;
```

5.3.2 Inserting the record bits

The record bits are used when transferring the data to the host. Therefore, the bits have to be present on the bus after the user logic block. This can either be solved by utilizing the `BUS_PIPELINE` variable, or by inserting the record bits on the bus manually.

Using the `BUS_PIPELINE` variable

If the length of the output data from user logic is the same as the input length, it's recommended to use the `BUS_PIPELINE` variable.

For example, if the record length is 1024 samples, and the latency of the custom logic is 10 cycles. Setting `BUS_PIPELINE=10` will delay the bus (and the record bits) by 10 cycles. This will ensure that the record is framed properly by the record bits after the user logic block.

It is important to note that the `BUS_PIPELINE` delay will not be applied to signals inserted by the user.

Inserting the record bits manually

If the length of the output differs from the input, only delaying the record bits will no longer frame the data correctly, therefore the bits have to be inserted on the bus as well.

For example, if the input data is 1024 samples, and the output data is 128 samples the record bits have to be generated to frame the new data. The `record_bits_in[0]` has to be asserted for the first cycle of the record, and `record_bits_in[1]` has to be asserted for the last cycle of the record. The data valid signal must also be asserted when both of the record bits are asserted. The record bits are inserted with the bus macro

```
insert_record_bits_X(record_bits_out)
```

where X is the channel. For example, to insert the record bits for channel A

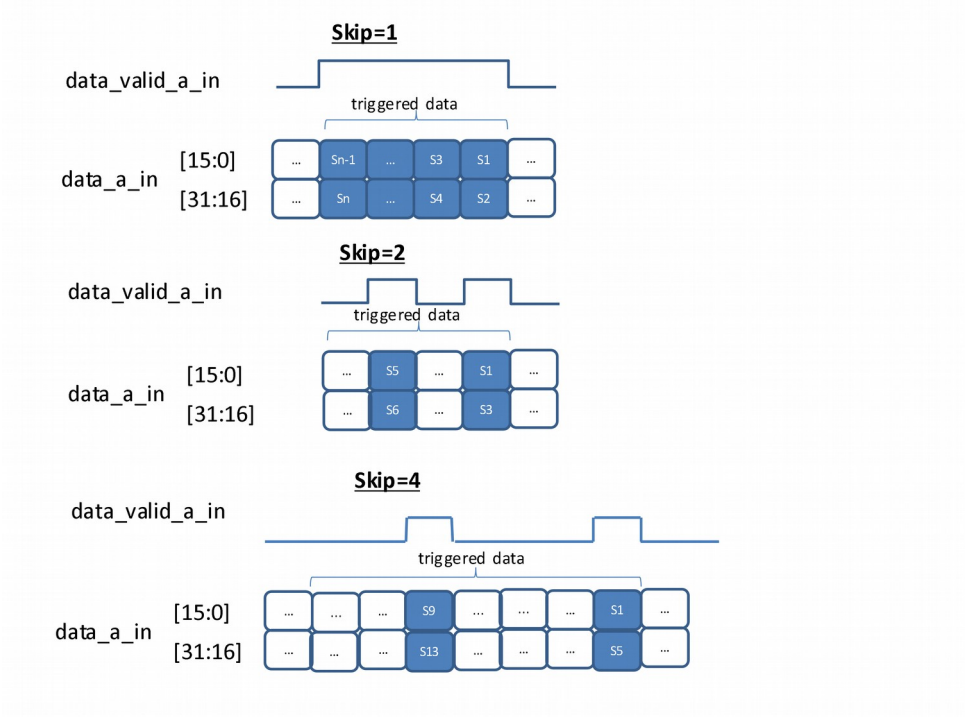
```
wire [1:0] record_bits_out;
// ...
insert_record_bits_a(record_bits_out)
```

Even when the record bits are inserted manually, it's important to set the `BUS_PIPELINE` variable correctly. The other header information, e.g. the time stamp and trigger vector are sampled on the record start bit. If the latency of the user logic block is unknown, the other header fields have to be extracted and inserted as well. If the other header fields aren't of interest this can be ignored. Data will still be received, however the header information will be invalid.

5.4 Using sample-skipped data

When enabling sample-skip, data throughput will be reduced and the data_valid_in signal will start to toggle to show which samples are to be used. This is only available in user_logic_2. When data_valid_in == 1 it means that all parallel samples are to be used. Example below for an -A unit (2 parallel samples) but same principle applies to all variants.

5.4.1 Example for an -A unit (2 parallel samples)



5.5 *Debugging on real hardware with Vivado Debug Core.*

There are different ways to add a Debug Core to the Vivado project. The procedure described in here is just one of the many possible ways that can be used to debug your custom logic that has been implemented with the ADQ Devkit on the real ADQ14 hardware.

When inserting your verilog code into the user_logic, you can mark the signals which you wish to probe by using the debug macro. For example:

(* mark_debug = "true" *) wire test_signal

```
(* mark_debug = "true" *) wire        trigger_event_a;
(* mark_debug = "true" *) wire        trigger_edge_a;
(* mark_debug = "true" *) wire [1:0]  trigger_position_a;
```

Once you are done with your custom code insertion, you can run the tcl command

devkit_synth_ul 1

and

devkit_synth_ul 2

This will create the netlists for both user_logic modules. After creating the netlist for both user_logic modules, click on "Run Synthesis" and wait for Vivado to finish synthesizing the whole design. Once it’s done, click on “Setup Debug” and follow the dialog screen to select the signals to probe. At this stage, you should be presented with the signals that you already marked for debug in your source code. Once you have confirmed your debug signals and the Debug Core generation is completed, click on "generate bitstream" on the left hand panel. You will be prompted to save the constrain file with the changes made by inserting the debug core.

Click yes and continue. Vivado will start generating a *.bit file containing the debug core. When this process has finished, run the tcl command "devkit_mcs" to convert the *.bit file into an *.mcs file. Open the folder "implementation\DevKit.runs\impl_1" and find the file named "debug_nets.ltx". Save this file to another location to use with xilinx Hardware Manager later when connecting the ADQ to the JTAG cable.

In summary, you should have 3 files in total:

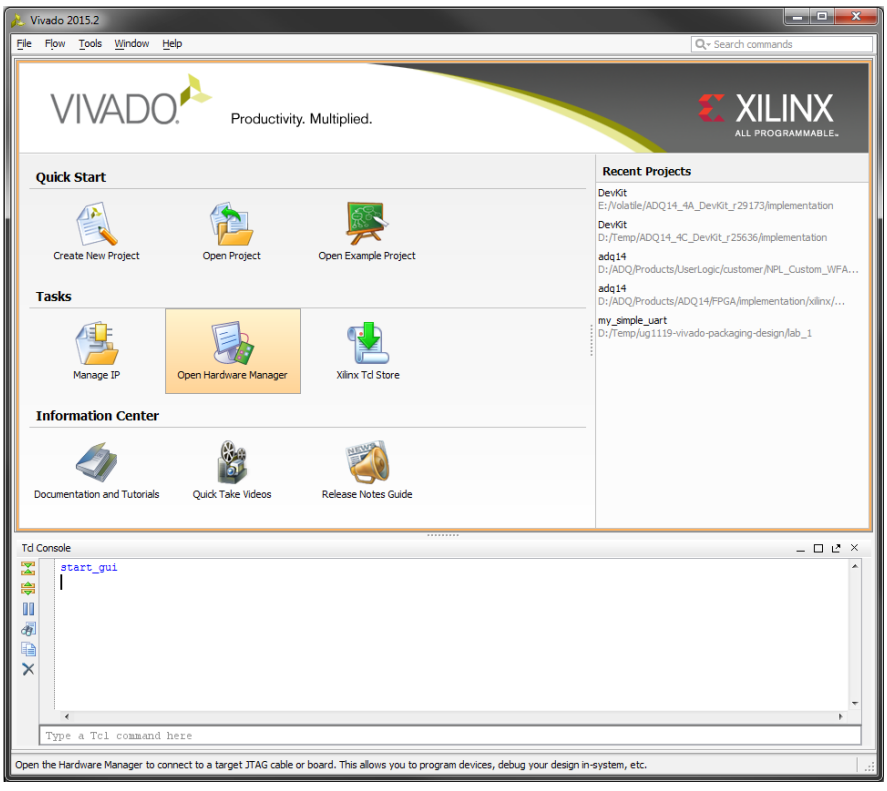
1. adq14.bit
2. adq14.mcs
3. debug_nets.ltx

Both the *.bit file and *.mcs file are functionally copies of each other. But to avoid confusions and mistakes please only use the adq14.mcs to upload to your device. The *.mcs file is not volatile as the *.bit file and it will keep the debug core intact even after you have power-cycled your device. Using the *.bit file will require you to re-upload the *.bit file again every single time you reboot your device. There is also a risk of system failure when using the *.bit file if your ADQ is connected via a P*Ie interface. Since uploading the *.bit file via the JTAG will reset the FPGA abruptly, P*Ie connection will be lost in midair and cause system failure. These behaviors are not specific to ADQ Devices. They are default behaviors and are common knowledge.

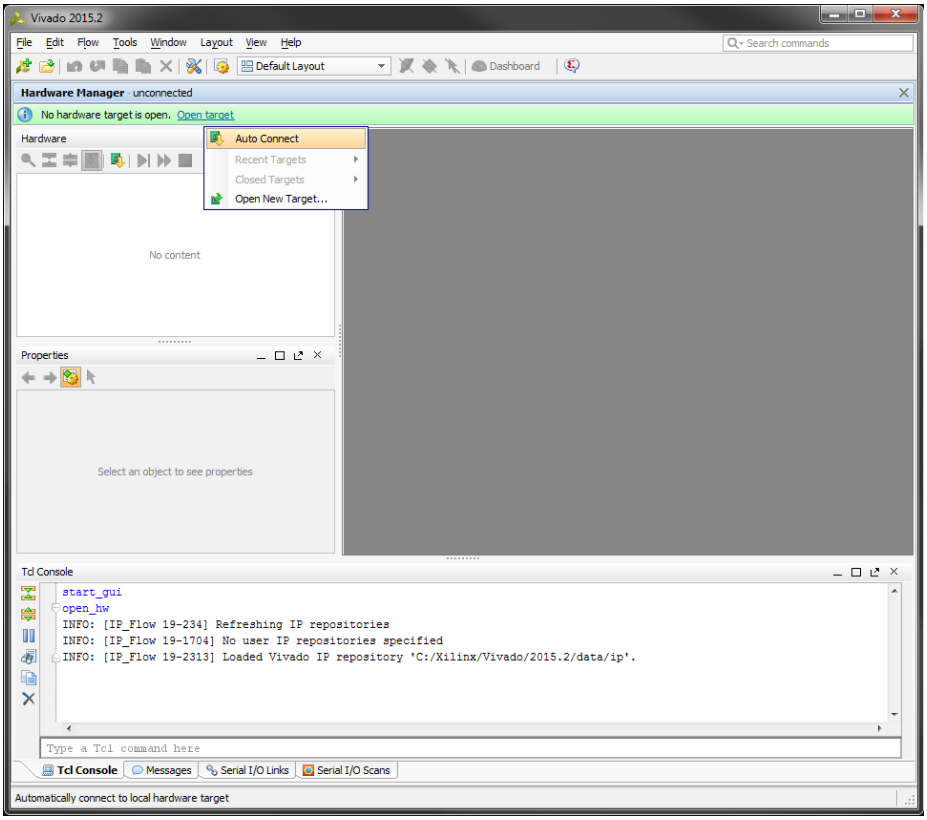
Once you have uploaded the *.mcs file containing the debug core, follow these step to start probing your debug signals.

1.) Depending on which clock signal you have chosen to be the master clock for your Debug Core, you might need to initiate the FPGA to get that master clock running before Vivado Hardware Manager can find the Debug Core on the ADQ14. If you have chosen the data clock as the clock for your probe, the best way to initiate that clock is to simply start ADCapturelab once and shut it down. That should be enough to initiate the clock for the Debug Core.

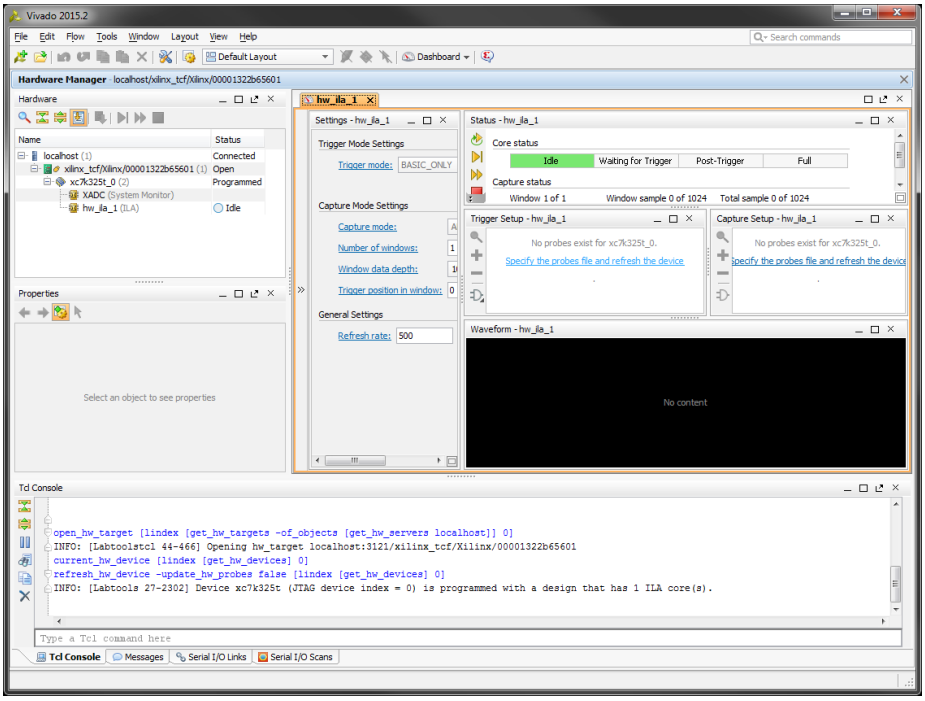
2.) Now start Vivado and click on Open Hardware Manager.



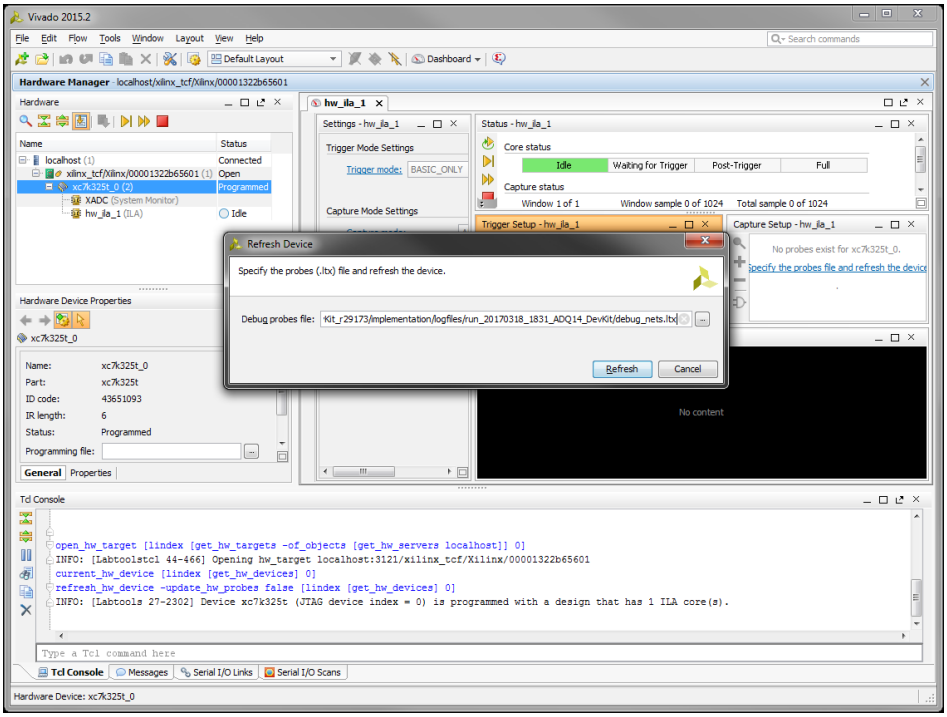
3.) Click on Open Target and chose "Auto Connect".

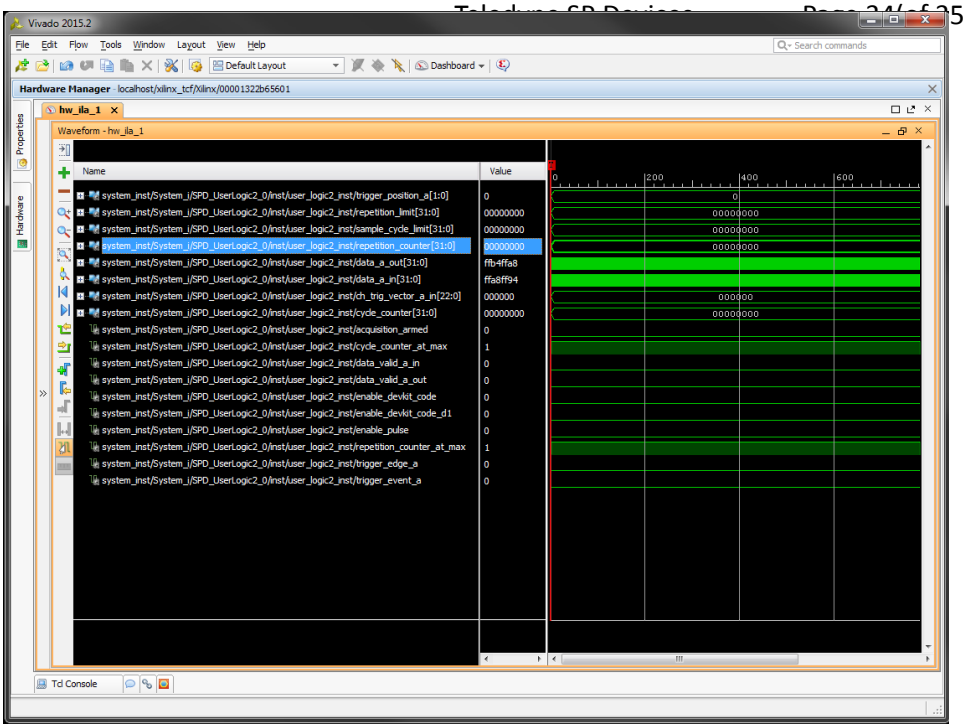


4.) In the Trigger Setup window, click on "Specify probe file and refresh device".



5.) Browse to the file "debug_nets.ltx" which you have saved earlier and click on refresh. You can now start probing your debug signals.




Vivado 2015.2
File Edit Flow Tools Window Layout View Help
Default Layout
Dashboard
Hardware Manager - localhost/xilinx_tcf/Xilinx/00001322b65601
hw_ila_1
Waveform - hw_ila_1
Name
Value
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/trigger_position_a[1:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/repetition_limit[31:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/sample_cycle_limit[31:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/repetition_counter[31:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/data_a_out[31:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/data_a_in[31:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/ch_trig_vector_a_in[22:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/cycle_counter[31:0]
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/acquisition_armed
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/cycle_counter_at_max
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/data_valid_a_in
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/data_valid_a_out
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/enable_devkit_code
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/enable_devkit_code_d1
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/enable_pulse
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/repetition_counter_at_max
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/trigger_edge_a
system_inst/System_i/SPD_UserLogic2_0/inst/user_logic2_inst/trigger_event_a
0
00000000
00000000
00000000
ffb4ffa8
ffa8ff94
000000
00000000
1
0
0
0
0
0
1
0
0
200
400
600
Tcl Console

5.6 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.

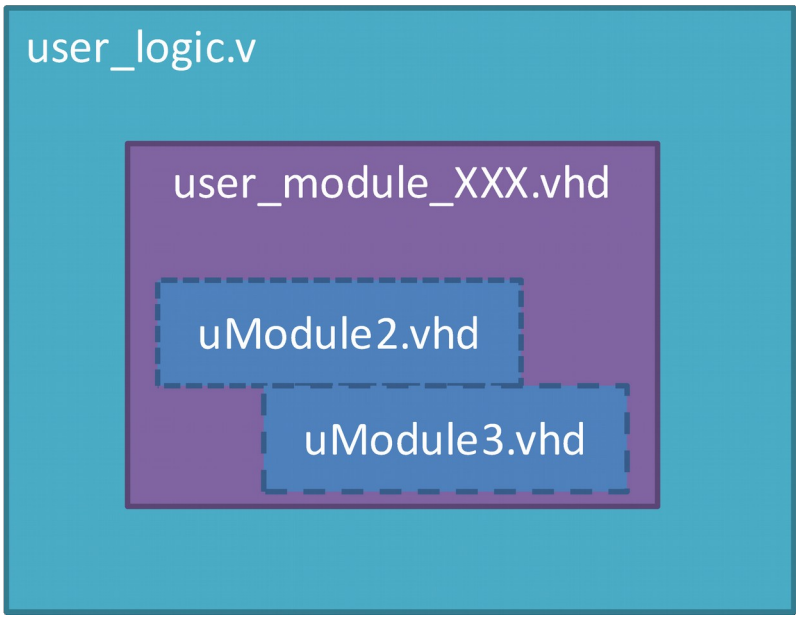


Figure 2: Hierarchy needed to use VHDL

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.