WAVE Doccumentation

James Williams

April 2019

Contents

1 FPGA Firmware

1.1 Waveform Playback Data Pathway

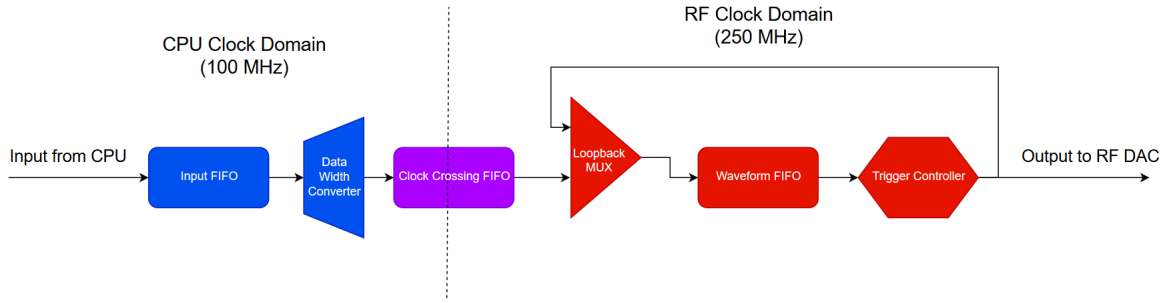

Figure 1: A diagram of the waveform playback data path used by each channel. Blocks in blue belong to the CPU clock domain while blocks in red belong to the RF clock domain. The purple block allows data to cross between the clock domains.

Each DAC requires new sets of 16 samples to be provided at a rate of 250 MHz. Because this is much faster than the on-board CPU which provides the waveforms for the DACs, a buffering stage is required. This buffer stage is used to read the waveform from the CPU at a relatively slow rate and then write the waveform to the DAC at the required 250 MHz.

The data path (fig.1) performs three tasks. First, it allows an waveform to be streamed from the CPU into a local high-speed first-in first-out (FIFO) buffer. Second, it allows that buffer to play back the waveform over the DAC when a trigger is received from either the CPU or externally. Third, it allows the waveform to be played back to the DACs a user-defined number of times before flushing the waveform from the buffer. The data path also includes several shift registers which control the delay before the experiment waveform playback, and the locking cycle waveform.

The first two tasks are implemented using a series of FIFO buffers along with some custom logic. After the waveform data leaves the CPU, it enters a data width converter which combines eight 32-bit words from the CPU into one 256-bit word for the DAC. This data stream is sent to a second FIFO, the clock crossing FIFO, which enables the data to cross over from the CPU clock domain to the DAC clock domain. The stream is then stored inside of a much larger (4096 samples deep) FIFO, the waveform FIFO, in preparation for playback over the DAC. Between the output of this larger FIFO and the input of the DAC is a small piece of custom logic, the trigger controller (located in axis_tready_slice.v), which allows the CPU to interface with the stream control signals. This ensures that the CPU can disable this stream so that the waveform can be stored in the FIFO, and later enable it so that the waveform can be played back over the DAC.

The third task is implemented by using a hardware counter inside of the trigger controller. After the waveform is loaded into the FIFO, the trigger controller is loaded with a 32-bit count value via a shift register (referred to as sr_cycles) indicating the number of clock cycles during which the waveform should be played back. The output of the trigger controller is also connected to the input of a MUX between the clock crossing FIFO and the waveform FIFO which allows the waveform to be looped back into the FIFO to enable repeated playback. Once the trigger signal is received, the trigger controller allows the data stream from the FIFO to the DAC to run for the prescribed amount of clock cycles before resetting itself for the next trigger pulse. The trigger controller also provides other control inputs for clearing the FIFO of any remaining waveforms and overriding the internal counter to trigger continuously.

The locking cycle waveform is set by loading a 16 sample waveform into the sr_locking internal shift register. This waveform is played back indefinitely until the start of the experiment. Playback resumes once the experiment has ended. The delay before the experimental waveform defines a period of zero signal output from the DAC between the end of the locking waveform and the beginning of the user provided waveform. This delay can be

adjusted in steps of $\frac{1}{4}$ ns. In order to achieve this resolution, a pre-waveform buffer (shift register referred to as sr_pre_waveform) was used as an intermediary so that the waveform can be shifted on a per sample basis (instead of delaying in steps of one full 4 ns clock cycle). For delays longer than 4 ns, another shift register (sr_zeros) is used to count clock cycles before the one cycle playback of the pre-waveform. The data for these shift registers is automatically generated and uploaded by the PC driver on channel initialization.

Each channel has an independent playback cycles, locking waveform, and zero delay configuration implemented by these shift registers. The shift registers for a given channel can be written to by selecting the channel using the channel select logic (elaborated on in section 1.3) and then writing to the appropriate register using signals on the GPIO bus running between each channel. Each shift register has a unique clock line, however all shift register share the same data line.

1.1.1 Waveform Data Path State Machine

A state machine within the waveform data path controls the triggering cycle for the channel. On reset, the state machine is placed in "state_wait_trigger" while waiting for the incoming trigger signal. The locking waveform is played back during this state. Once a trigger signal is received, the state machine advances to "state_zeros", during which 0 signal is output on the DAC before the experimental waveform for a user-defined number of cycles. The state machine then goes into "state_pre_waveform" for one cycle and then into "state_trigger" for a user defined number of cycles in order to play back the experimental waveform. Finally the state machine advances to the post experiment delay state ("state_post_delay") and then a cleanup state where it waits for all control signals to be reset until going back to the idle state.

1.2 Waveform Capture Data Pathway

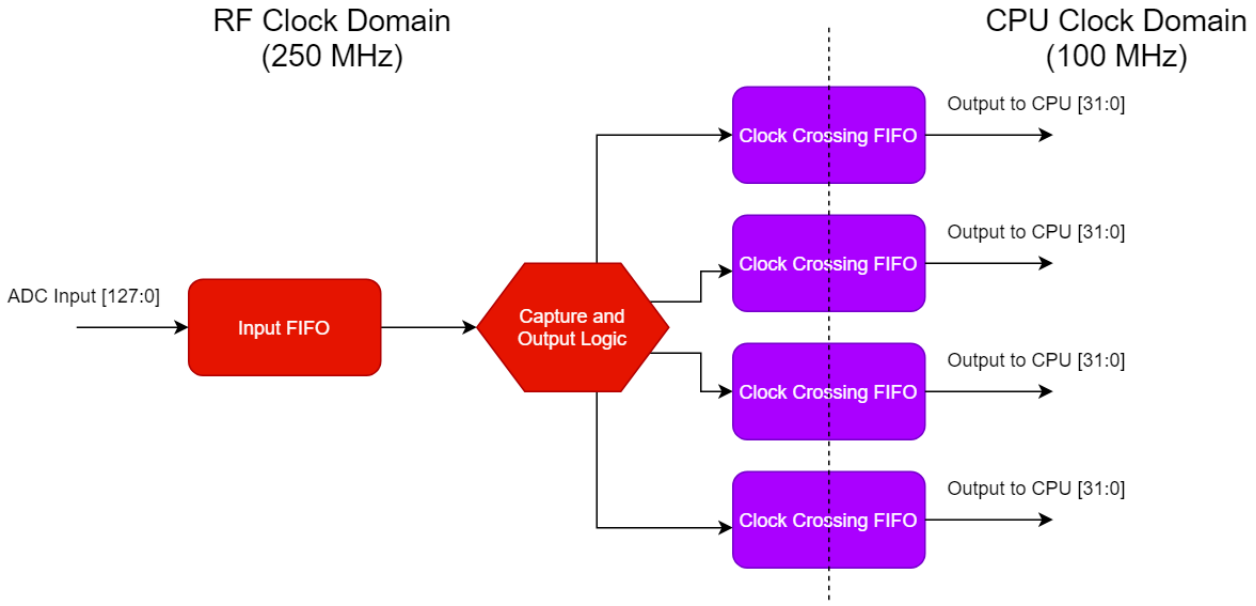

Figure 2: A diagram of the data path used to move data from the ADC to the CPU. Blocks in red are in the RF clock domain while blocks in purple are used to cross clock domains.

The on-board ADC has been enabled in order to record and store experimental data without the need for an oscilloscope. A custom IP was implemented which allows data to be received from the ADC using a capture trigger signal provided by the CPU. This data can then be uploaded and viewed on the user's PC using the python driver.

The ADC samples at 2 GSPS and can sample a maximum of 60 μs.

Because the ADC operates much faster than the CPU, a buffer is needed to store the waveform so that the CPU can read it out at a later time. The purple FIFO buffers in Figure 2 are used to store and split up the waveform into smaller (from 128 bits to 32) bus sizes so that the waveform can be read out using the stream interfaces provided by the CPU. These four separate streams are recombined in software to display the waveform.

1.3 Top Level Design

The top-level design consists of 3 main components: the DAC logic, the ADC logic and the Microblaze CPU. The CPU sits at the center of the design and controls all DAC and ADC functions along with the GPIOs and UART connections. The CPU has 16 AXI stream outputs, each of which is used by the 16 DAC channels. These outputs first go to the waveform playback data pathway, and from there to the RF data converter IP from Xilinx. This IP abstracts away all RF functionality of the FPGA. The CPU also contains 16 AXI stream inputs. 4 of these are used to read data captured from the ADC by the waveform capture data pathway.

The top level design also includes two blocks used for managing the DAC datapaths: the trigger distribution module and the channel select module. The trigger distribution module was added to solve a timing issue. It was noticed during testing that some channels would trigger 4 ns early or late with respect to channel 1, and that this skew would change between triggers. The problem was solved by assigning one register to each trigger signal and having these registers send a trigger pulse within the RF clock domain to ensure all DACs started on the same clock cycle.

2 Bare Metal Drivers for the Microblaze

2.1 RFSoC Driver

The components of the RFSoC driver can be found in the files “rf.c” and “rf.h” located in the “src/drivers” folder of the “rfsoc_controller” project within the Xilinx IDE.

2.1.1 void rf_load_bitstream(u8* stream, u32 length, u8 channel)

- Purpose
  - Loads a waveform from a bitstream to a certain channel
- Arguments
  - stream - pointer to bitstream to be written as a waveform
  - length - length of bitstream
  - channel - channel to which bitstream will be loaded
- Notes
  - Bitstream must be MSB first then LSB, every two bytes should constitute one 16-bit 2’s complement DAC sample.

2.1.2 void rf_set_repeat_cycles(u8 channel, u32 cycles)

- Purpose
  - Sets the number of clock cycles for which a given channel should be triggered
- Arguments
  - channel - channel number on which to set cycles
  - cycles - number of cycles

2.1.3 void rf_flush_buffer()

- Purpose
  - Empties the buffer of all channels

2.1.4 void rf_set_trigger_mode(u8 mode)

- Purpose
  - Sets the trigger mode
- Arguments
  - mode - trigger mode, options are TRIGGER_CONTINUOUS and TRIGGER_CYCLES which are defined as macros.
- Notes
  - In continuous trigger mode, DACs will be provided with waveforms for as long as the trigger signal is active.
  - In cycles mode, DACs will be triggered for number of cycles set by rf_set_repeat_cycles.

2.1.5 void rf_set_loopback(u8 option)

- Purpose
  - Toggles loop-back feature on waveform buffers
- Arguments
  - option - choices are YES or NO defined as macros.
- Notes
  - In YES mode, the waveform will be rewritten to the waveform buffer through a MUX.
  - In NO mode, the waveform will only be played back once and not rewritten to buffer.

2.1.6 void rf_trigger()

- Purpose
  - Triggers all DAC channels once.

2.1.7 void rf_set_adc_cycles(u32 cycles)

- Purpose
  - Sets the number of clock cycles for which data from the ADC should be captured.
- Arguments
  - cycles - number of clock cycles
- Notes
  - The ADC can record for a maximum of 65536 cycles

2.1.8 void rf_read_adc_buffer(u8* buffer, u32 num_samples)

- Purpose
  - Reads back a specified number of samples into a bytestream buffer.
- Arguments
  - buffer - byte buffer into which samples will be written
  - num_samples - number of samples to be written
- Notes
  - If too many samples are read from the ADC using this function, the CPU will stall. Samples are 16 bits and written into the buffer MSB first.

2.1.9 void rf_flush_adc_buffer()

- Purpose
  - Removes all previous data from ADC buffers

2.1.10 void rf_set_pre_waveform(u8 channel, u8* stream)

- Purpose
  - Sets the 16 sample "pre-waveform" to be played back for 1 clock cycle before the waveform from the buffer
- Arguments
  - channel - channel number to which waveform will be written
  - stream - waveform bytestream, should be 32 bytes long with MSB first
- Notes
  - This function is used to implement the $\frac{1}{4}$ nanosecond resolution of the delay before experiment. The prewaveform implements the total delay modulus 4 nanoseconds.

2.1.11 void rf_set_zero_delay(u8 channel, u32 value)

- Purpose
  - Sets the coarse delay before experiment, or equivalently, the number of clock cycles the trigger state machine will wait before moving to the pre-waveform state.
- Arguments
  - channel - channel number to which the delay will be written
  - value - number of clock cycles to wait
- Notes
  - This function is used to implement the component of the total delay which is a multiple of 4 nanoseconds.

2.1.12 void rf_set_locking_waveform(u8 channel, u8* stream)

- Purpose
  - Sets the 16 sample waveform to be played back during the locking cycle before and after the experiment.
- Arguments
  - channel - channel number to which the delay will be written
  - value - number of clock cycles to wait
- Notes
  - This function is used to implement the component of the total delay which is a multiple of 4 nanoseconds.

2.2 Command Handler

The command handler is used to receive and dispatch incoming commands from the PC driver. It uses a simple poling system to receive a byte over uart, indicating which command should be executed. More data may then be received inside the command-specific function if needed.

If you wish to use the RFSoC driver to implement additional board functionality, check the various functions in "cmd_handler.c" for example usage.

2.3 Peripheral Drivers

2.3.1 UART Driver

The UART driver controls all interactions between the Microblaze firmware. It allows the firmware to send a byte stream, receive a byte stream of a known length and receive a command. It also provides functionality for printing debug information in the event of an error.

2.3.2 GPIO Driver

The GPIO driver handles all configuration and driving of the GPIO outputs of the Microblaze. These signals are used to control the waveform playback and capture pathways. A 16-bit GPIO bus travels to all data paths, allowing each to use one or more of the signals on the bus to load user settings. These signals drive the shift registers elaborated on in section 1.1 and control the internal triggering through the trigger controller.

3 Python PC Driver

3.1 Wavefile Object Fields

The Wavefile object is used to generate the appropriate bitstream from a CSV file describing the waveform. The waveform will be normalized about 0 during processing and then scaled to the full output value of the DAC. This scaling value can be modified by the user. The wavefile object can be created using the following python code:

```
import RFSoC_Board as rf

wave_file = rf.WaveFile("wavefile_name.txt", period, pre_experiment_delay,
post_experiment_delay, amplitude_factor, is_locking, locking_shift)
```

- period
  - Sets the period of the waveform in nanoseconds. Mininum period is 4 nanoseconds, the period must always be a multiple of 4 nanoseconds.

- pre_experiment_delay
  - Sets the zero output delay between the locking cycle and the experiment in nanoseconds. Will be rounded to a multiple of 250 ps.
- post_experiment_delay
  - Sets the zero output delay between the end of the experiment and the locking cycle in nanoseconds. Will be rounded to a multiple of 250 ps.
- amplitude_factor
  - Sets the amplitude multiplier for the waveform, can be between 0 and 1. A value of 1 will scale the waveform to the maximum output amplitude of the DAC.
- is_locking
  - If set to 1, the wavefile is recognized and treated as a locking waveform with a period of 4 ns.
- locking_shift
  - Sets the number of samples by which the phase of the locking waveform should be shifted by. 1 sample shift equals 250 ps of phase shift.

3.2 Wavefile Object Functions

The user should never need to call any of the functions associated with the Wavefile object. All neccissary configuration of the wavefile object is done within the constructor.

3.3 Channel Object Fields

The channel object is used to specify the channel number and the number of times an experiment waveform should be repeated. It also specifies the experimental and locking waveforms to be used for a particular channel. A channel object can be created using the following python code:

```
import RFSoC_Board as rf

channel = rf.Channel(number, repeat_cycles, locking_wavefile, experiment_wavefile)
```

- number
  - Specifies the channel number, or equivalently, the port on which the channel waveform should be played back. Acceptable values are integers between 0 and 15.
- repeat_cycles
  - Specifies the number of times to repeat a waveform on the specified channel. Must be an integer greater than 0.
- locking_wavefile
  - Specifies the wavefile object containing the desired locking waveform information
- experiment_wavefile
  - Specifies the wavefile object containing the desired experimental waveform information

3.4 Channel Object Functions

All channel object functions are called using the RFSoC_board object and should not be calle by the user.

3.5 RFSoC_board Object Fields

The RFSoC_board object is used to catalog all channels and settings related to the board. This object also containts all functions neccissarry to setup, arm, and trigger the board via software. An RFSoC_board object can be created using the following python code:

```
import RFSoC_Board as rf
import init_board_int as ib

board = ib.init_board_object(com_port)
```

Board objects should always be obtained through the init_board_int script. This script checks to ensure that the connection to the board is working and that the board is ready to accept commands. The com_port argument takes a string as the name of the com port used to contact the board. These names can be found in the device manager in Windows under the serial port section (i.e "COM4").

3.6 RFSoC_board Object Functions

- ping_board()
  - Checks if the connection to the board is up. Returns 1 if the connection is up, 0 otherwise.
- write_bytes(b)
  - Writes an array of bytes (b) to the board and waits for a single byte as an acknowledgement from the board. Returns the value of the acknowledgement. A correct acknowledgement has a value of 0x00.
- add_channel(c)
  - Adds a channel (c) to the list of channels to be uploaded to the board.
- write_all_channels()
  - Uploads all added channels to the board. Returns 0 if successful, 1 otherwise.
- wait_ack()
  - Waits until UART timeout for a single-byte aknowledgement from the board. Returns the value of the aknowledgement. The UART timeout can be set at the top of the file by modifying the UART_TIMEOUT variable.
- receive_bytes(num_bytes)
  - Receives a specified number of bytes (num_bytes) from the board over UART. Returns the received bytes in an array. The port associeted with the board must be opened before calling this function.
- write_channel(channel_num):
  - Writes a channel specified by channel_number to the board. Returns 1 if successful, 0 otherwise.
- set_loopback(choice)
  - Sets the internal MUX setting which controls whether the input of the waveform FIFO (ADD FIGURE CITATION) is connected to itself or the CPU. A choice of 0x01 indicates the FIFO will loop back to itself. A choice of 0x00 allows the CPU to write data to the FIFO.
- set_trigger_mode(mode)
  - Sets the trigger mode of the board between TRIGGER_CYCLES (0x00) and TRIGGER_CONTINUOUS. In cycles mode, the board operates normally, triggering for a user specified number of repeat cycles and then returning to the locking cycle. The continuous trigger mode should not be used, it is intended for debugging the DAC output.

- trigger()
  - Triggers the board once.
- flush_buffer()
  - Flushes the waveform FIFOs of any remaining waveform data.
- set_adc_cycles(cycles)
  - Sets the number of clock cycles for which the ADC should capture. 1 clock cycle equals 4 ns and 8 samples of captured data.
- read_adc()
  - Reads the waveform from the ADC, returns the waveform as an array of voltage values. This command will stall the CPU and require a reset if the ADC has not captured any data.
- flush_adc_buffer()
  - Flushes all data from the ADC capture buffer.

4 Operating Instructions

4.1 Modifying the FPGA Firmware

The FPGA firmware files are divided into three IPs and the top level diagram. The IPs are the the waveform playback data path referred to as "rfsoc_data_pipeline", the waveform capture data path referred to as "rfsoc_adc_data_capture", and the trigger distribution module referred to as "trigger_controller" not to be confused with the trigger controller inside each waveform playback data path.

Any time the source files or block diagram of an IP is modified, the IP must be repackaged and updated within the top level diagram for the changes to take effect. The steps to repackage an IP are as follows.

- After modifying source files, be sure to refresh the modules in the block diagram of the IP. A yellow bar with blue text should appear at the top of the block diagram window asking if you would like to update the modules within the IP after a module change has been detected. If this button does not appear, try going to the top level block diagram and then back to the block diagram of the modified IP.
- After updating the modules, go to Tools, Create and Package IP, Package a block design from the current project (be sure to select the correct IP). When asked for the directory in which to save the IP, you must use the directory already associated with the IP as follows:
  - "rfsoc_data_pipeline" - /ip_repo
  - "rfsoc_adc_data_capture" - /adc_ip
  - "trigger_controller" - /trigger_controller_ip
- Continue to press next until the contex menu closes. A new tab labeled "Re-package IP" will then pop up next to the diagram of the IP. Go to the last option in the menu on the left side of the window and then select "Re-package IP".
- Next, go to the top level diagram. A yellow bar will appear prompting you to upgrade the IP. Follow the on screen instructions until the relevant IP is upgraded. You may need to look at a window which pops up as a tab near the bottom of the screen (be sure to hit refresh at the top of that window).
- Once the IP upgrade is complete, you will need to hit "Generate Bitstream" on the lower left side of the Vivado window.

4.2 Connecting to the board

4.2.1 Digital Connections

The board has two micro-USB connections. The first is used for programming the FPGA. It is labeled SYS_JTAG J69 on the silk screen. It should always be connected to the Vivado computer. The second connection is used to talk to the CPU firmwave over UART. It is labeled USB/UART J1. It should be connected to whichever computer is using LabVIEW to control the board.

4.2.2 RF Connections

The RF connections originate from the bundle of purple cables. The pinout of these cables is displayed on the silk screen of the FPGA PCB below the FPGA itself.

4.3 Uploading the Configuration and Firmware

After the generate bitstream step has compleed in Vivado, you will need to export the hardware design to the software SDK in order to run the board. To do this, go to File, Export, Export Hardware, and be sure to check "Include Bitstream". Then go to File, Launch SDK. Once the SDK window is open, you will need to press the button with a red arrow and three green squares. This will flash the firmware to the FPGA. Then press the green play button to start the CPU firmware. At this point, the board is ready to accept commands from the PC driver.

4.4 Fixing Issues During CPU Firmware Compilation

Sometimes after a new hardware file has been exported to the SDK, the SDK will refuse to compile. The first step to try is cleaning the project by right clicking on "rfsoc_controller" and selecting "Clean Project". If this does not work, you can also try regenerating the linker script through the same context menu.

Occasionally after a new hardware file has been exported, the SDK will refuse to load and will crash silently instead. If this occurs, you will first need to copy the "rfsoc_controller" and "rfsoc_controller_bsp" folders into another directory before deleting the sdk directory in the root folder of the Vivado project. Then have Vivado relaunch the SDK. Once the SDK is working, simply import these projects back into the workspace and everything should compile normally.

If you needed to do the step above and nothing will compile now due to a linker error, you will need to regenerate the linker script by right clicking on the "rfsoc_controller" project and selecting "Generate Linker Script". Be sure to set the heap size to something larger than 4096 * 32 bytes.

4.5 Fixing Issues During FPGA Firmware Synthesis and Implementation

Sometimes Vivado will complain about IP files missing. To fix this, simply repackage all IPs using the steps listed in section 4.1.

If you're completely stuck with some random error in Vivado, your last resort is to download the Git repository and copy in the rfsoc_controller directory (the Vivado directory) to your local repository directory and start from there.

4.6 Clocking the ADCs and DACs

If for some reason the board freesez up while you are trying to upload a waveform (all LEDs turn green when frozen), check to make sure that the clock inputs for all ADC and DAC tiles are connected. These clock inputs are located in the bundle of purple cables. The pinouts for these cables are silkscreened on the board. Please see **?** for more information.

4.7 Changing the number of available channels

See the firmware configuration file “rf.h” for changing the number of available channels.