Keysight M3 Series FPGA Design$^{\alpha}$ User's Manual

Chao Zhou, Pinlei Lu, Maria Mucci
and Michael Hatridge

10/1/2018

Contact: chz78@pitt.edu; pil9@pitt.edu; mmm242@pitt.edu;
hatridge@pitt.edu

Contents

1 FPGA Design

1.1 About Keysight M3602A FPGA Design Environment

Keysight M3602A is software packaged with the purchase of the Keysight M3 series AWG and digitizers. The manuals for those PCI eXtensions for Instrumentation (PXI) modules and software can be found on GitHub under "Manuals" folder.[1]

You must install the accompanying software to run this package. Vavido 2015.2 is a software by Xilinx for synthesis and analysis of HDL designs. It may be downloaded form Xilinx website. And you need to obtain the free webpack license to make it work.

1.2 Basic Principles and Schemes

1.2.1 Design Principles

- **Data Form** In M3602A you will find that the data from the incoming source (Channel 1-4) and the outgoing ports (DAQ1-4) of the digitizer are both 16bits long.

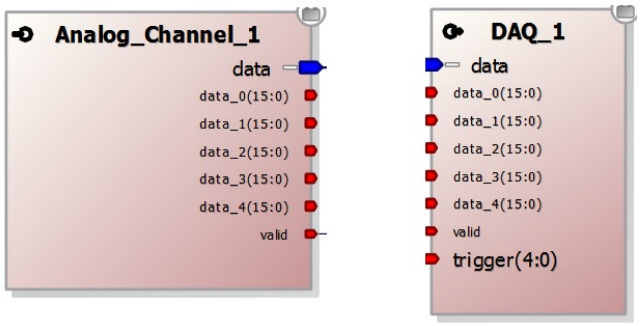


Figure 1: Channel and DAQ block

The incoming analog signal is translated to 16-bit integer-form digital data by the ADC.(We figured out that it is in two's complement form)(more details about this will be discussed in the Analog Channel Block section). Thus, we want to do all calculations in integer form (floating number calculation is more complicated). We want the final calculation result to be 16-bit integers.

- **Data Truncation** Though we want 16-bit output, it is not possible to do all the calculations in 16 bits. E.g. when you multiply two 16-bit integers together, the full result should be 32-bits long. So we need to truncate the

[1] We got an updated version of the manual for the M3102A Digitizer, which Keysight haven't put on their website yet.

result after each step of calculation[2]. For the Multiplier function, we just keep the first 16 bits. Yet the truncation for the integration (accumulation) result is a little bit tricky, because in different experiments, it's hard to say how long you need to integrate, which means you don't know how large the result can be. Thus, you cannot determine where to truncate directly. Our strategy of solving this problem is to do a pre-run first and find position of the most significant bit (MSB) of the result using the Find_MSB block. In the real experiment, we will truncate based on the pre-run.

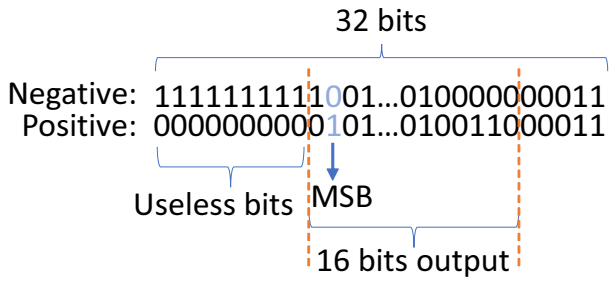


Figure 2: Data Truncation Example

As shown in Fig. 2, all the integration results are originally 32-bits long, and we want 16-bits output. Most of the bits in the front does not contain information(all '0's for a positive number and all '1's for a negative number). So, for example, if you want to truncate a positive number properly, you have to find where the first '1' appears (most significance bit). The truncation should start from one bit before this[3].

[2]We do not do one truncation at the end of all the calculations because when the length of data gets too long, some blocks will break down

[3]We need to truncate at one bit greater than MSB, because we are using 2's complement form, we need one extra bit for sign

1.2.2 Pre-Run Scheme

The pre-run firmware will output SI, SQ, RI, and RQ traces after demodulation and also tell us the truncation point through the PC port. The purpose of the pre-run is to get the following information we need for the real experiment:

1. **First Truncation Point** This truncation point is after the integration in the demodulation block. Readout by Find_MSB block.

2. **Integration Range** After you see the demodulation result from the output, you have to look at the IQ trace and determine where to start and stop the integration. In a real experiment you will enter these values into the FPGA PC port to tell the FPGA the integration range.

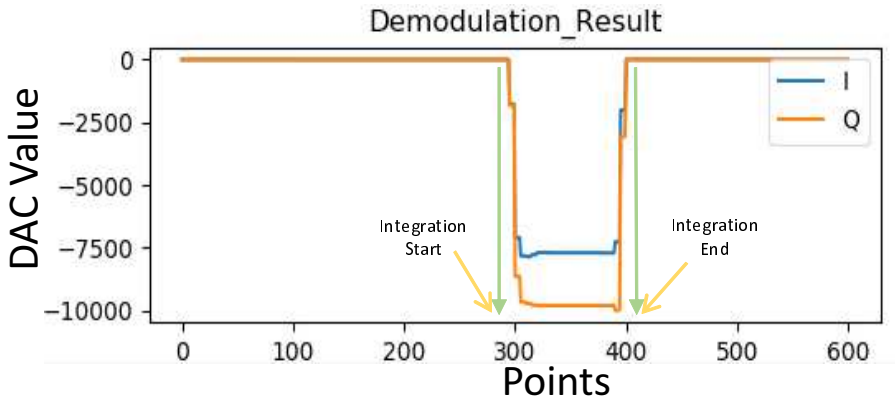


Figure 3: Integration Range Example

3. **Second Truncation Point** This is the truncation point after the final integration. This only gives you the result with a default weight function (constant=1). After you add the weight function, you may need to adjust this value slightly by hand to have a more precise result.

4. **Weight Function** In order to get the weight function, you must be able prepare the qubit in its g or e state and save the IQ trace data (with default weight function). Then you calculate the weight function based on the difference of the traces in Python. First you have to perform the π pulse tune up experiment (with the real experiment firmware, and default weight function) to get the parameters needed (e.g., π pulse amplitude). Then you run state preparation experiments (using the π pulse HVI) with pre-run firmware once more to get the IQ trace for different states.

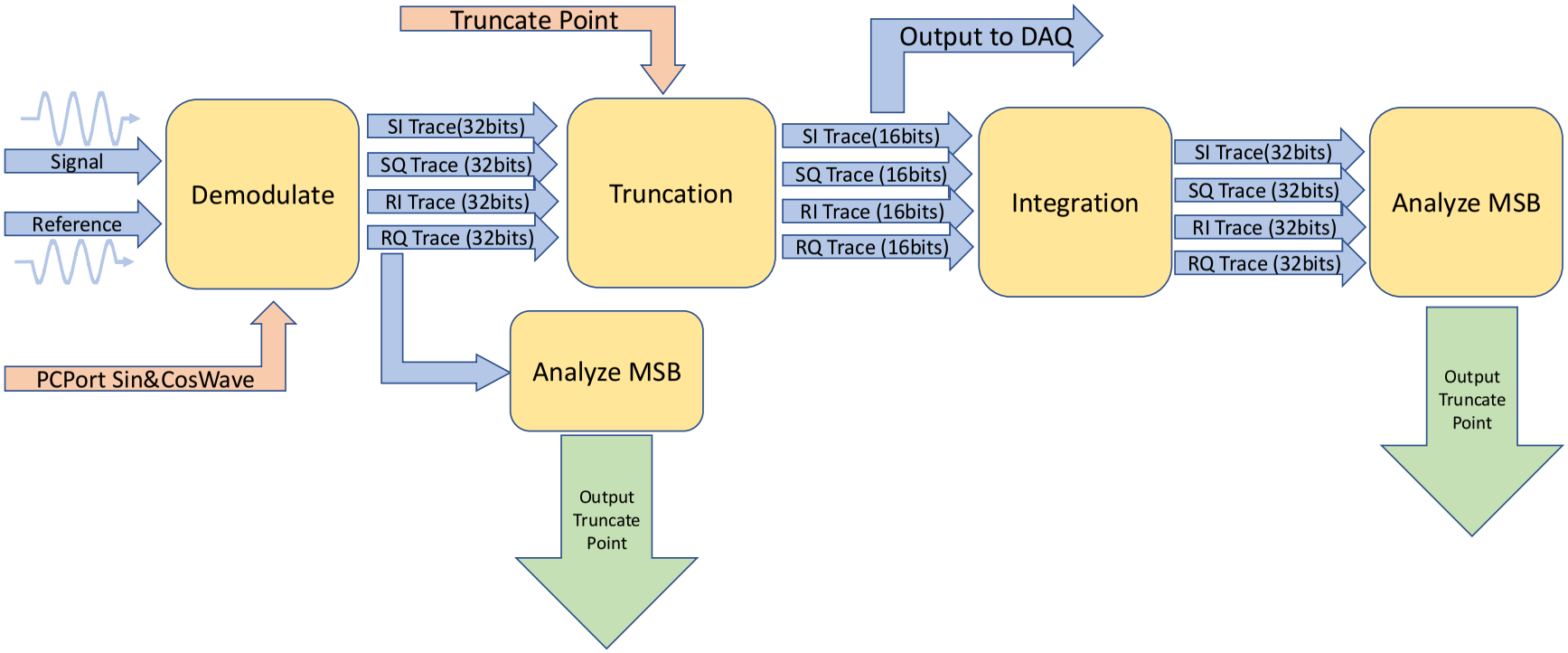


Figure 4: Pre-Run Scheme

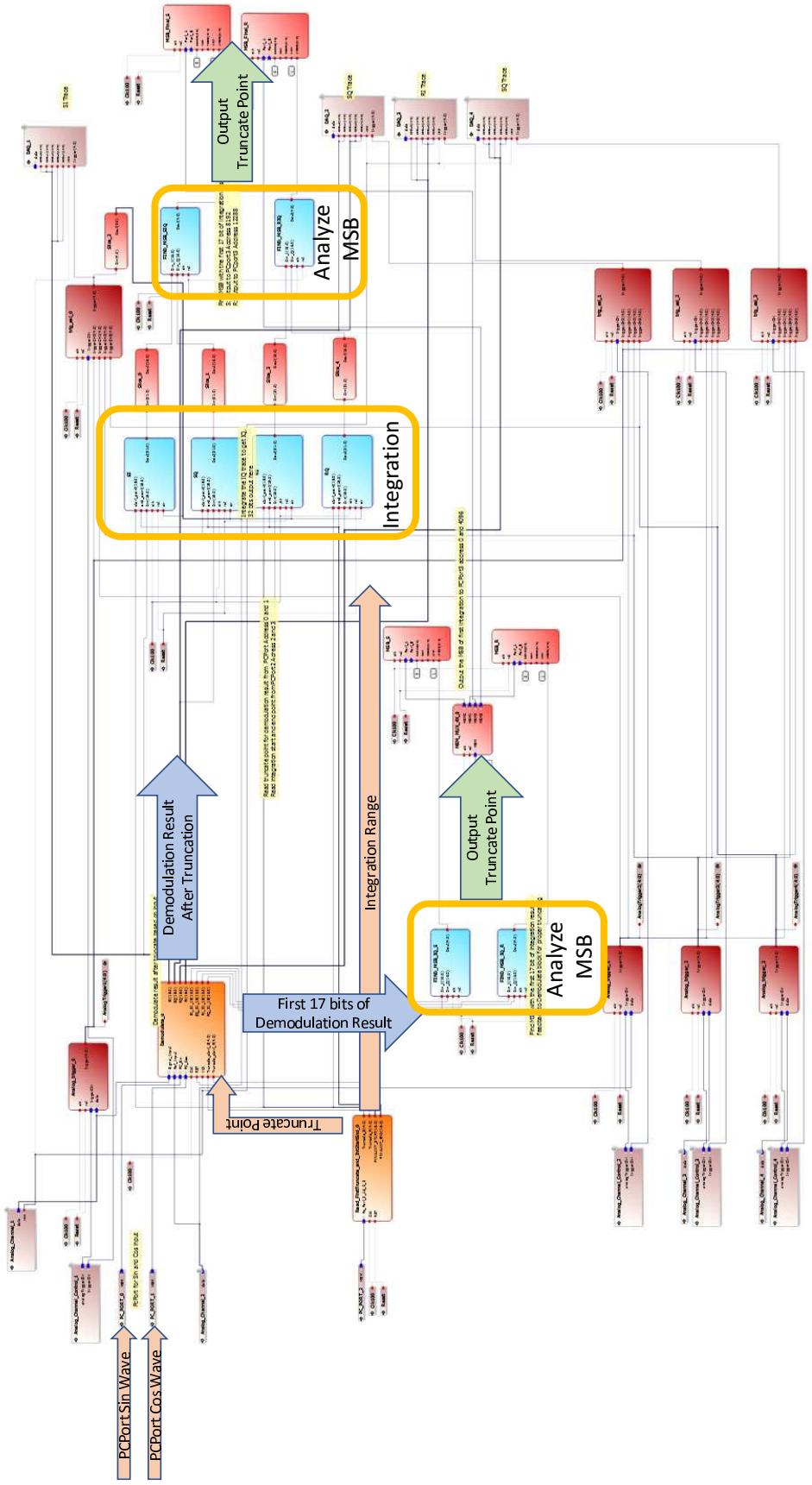


Figure 5: Pre-Run FPGA Design

1.2.3 Real Experiment Scheme

The real experiment firmware will output the I,Q result.

The input signal and reference will come in from the channel 1 and 2 respectively of the M3102A digitizer. Then it will go through the following steps in the FPGA:

1. **Demodulation** The input signal (sig) and reference (ref) will be multiplied by the 50MHz sine and cosine function in the FPGA RAM. Then the result will be integrated over every 10 points (with 5-point overlap). The result will be truncated based on the pre-run result. This will give us the SI, SQ, RI, RQ traces.

2. **Weight Function** The SI, SQ traces will be multiplied by the weight function we got from the pre-run result.

3. **Integration** This integration step will integrate SI, SQ (post-weighting), RI, and RQ in the interval we get from the pre-run. The result will also be truncated based on the pre-run.

4. **Rotation** As shown in Fig. 6.

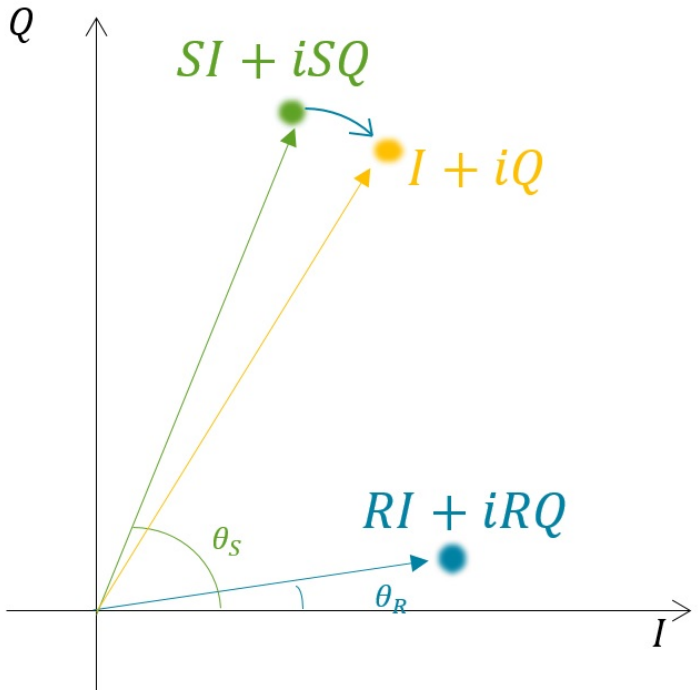


Figure 6: Rotation

Here we will calculate

$$I + iQ = (SI + iSQ)e^{-(RI+iRQ)} \tag{1}$$

$$= \frac{(SI + iSQ)(RI - iRQ)}{\sqrt{RI^2 + RQ^2}} \tag{2}$$

$$= \frac{(SI \cdot RI + SQ \cdot RQ) + i(-SI \cdot RQ + SQ \cdot RI)}{\sqrt{RI^2 + RQ^2}} \tag{3}$$

So we have

$$I = \frac{(SI \cdot RI + SQ \cdot RQ)}{\sqrt{RI^2 + RQ^2}} \tag{4}$$

$$Q = \frac{-SI \cdot RQ + SQ \cdot RI}{\sqrt{RI^2 + RQ^2}} \tag{5}$$

as the output I,Q.

The numerators are calculated with Get_IQ block, and the denominators are calculated with the Normalization block. A delay of 5 cycles is added to the numerators before the multiplication.

5. **State Judgement** Here we will calculate the distance from the input IQ to the center of g, e, f gaussian tomographs. Determine the qubit state by judging which distance from the experimental point to expected centers is the shortest.

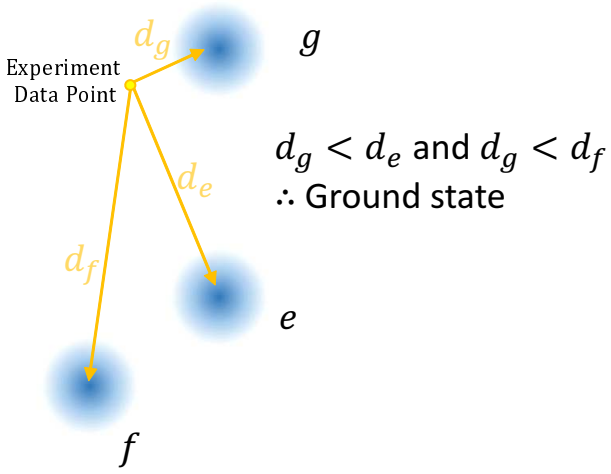


Figure 7: State Judgement Example

The state will be represented by a 2-bit vector, and attached to the last two bits of the output.

6. **Output** Due to mysterious reasons from Keysight, each configuration of DAQ is limited to a maximum of 1,000 data acquisition cycles. However, we need 10 000 000 data points to draw a good histogram. In order to overcome this limit, we design a sub_buffer block. The data will be stored in this block temperately. The volume of this block is 5,000 data points. When the sub_buffer is full, we output these data to DAQ. Thus we can acquire 5000*1000=5,000,000 data points in each configuration of DAQ(see details in Sub_Buffer block section)

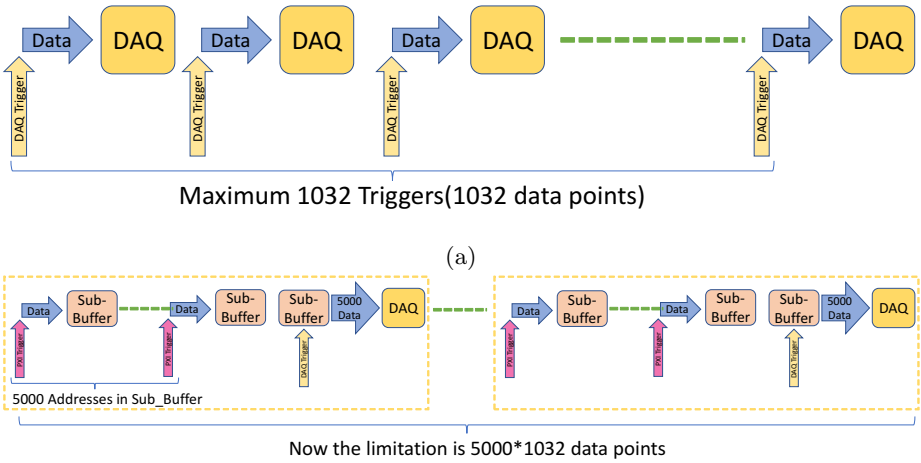


Figure 8: Data Acquisition (a) without Sub_Buffer. (b) with Sub_Buffer

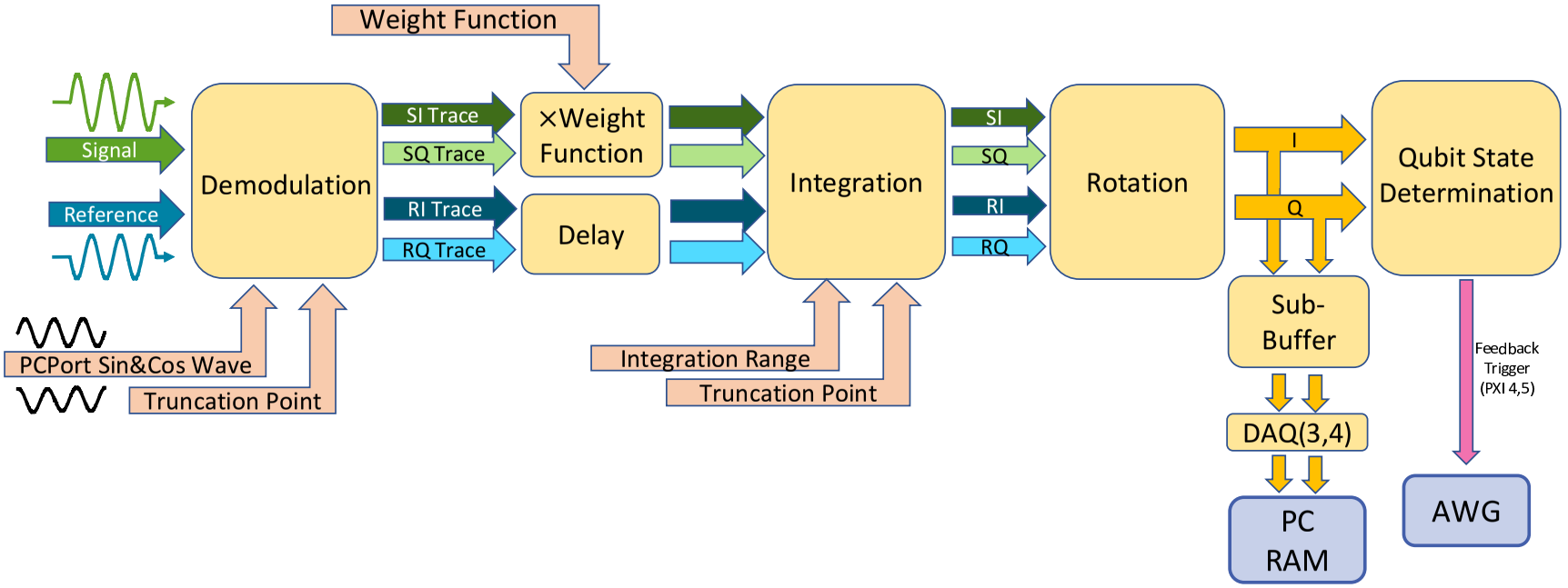


Figure 9: Real Experiment Data Processing Scheme

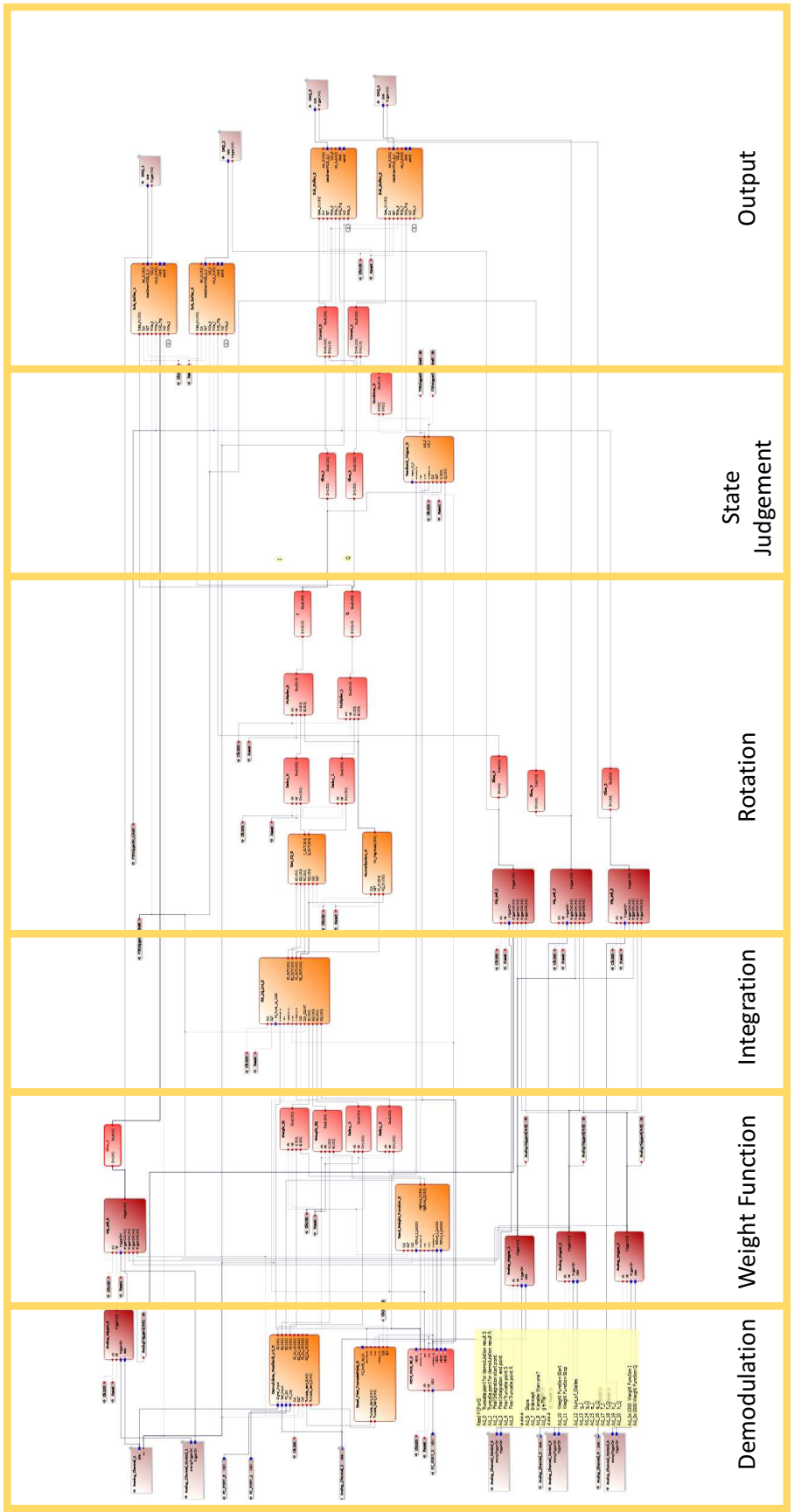


Figure 10: Real Experiment FPGA Design

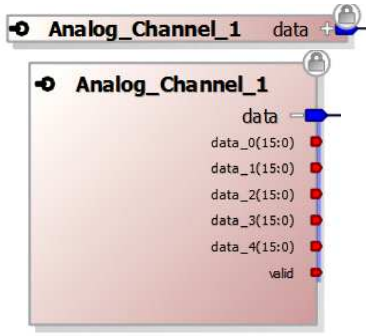
2 Building Blocks

2.1 Built-In Blocks from Keysight

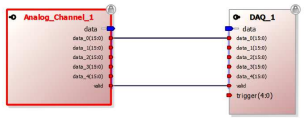
In this section I will introduce some of the built-in blocks provided in M3602A. Since the provide documentation from Keysight doesn't describes the function of each block clearly, we worked hard to understand the details of each block.

Here I'll share some of the confusing details we figured out.

2.1.1 Analog Channel



(a) Ports&Interface



(b) Valid Connection Example

Figure 11: Analog Channel

This is where the digitized data comes in to the FPGA. The blue arrow is called "Interface". Here, the "data" interface contains 6 ports.

The first 5 ports (data_0 to data_4) are five data points taken in one clock trigger, the time interval between each data point is 2 ns. (clock frequency * data points per clock trigger=100MHz * 5=500 M Samples /s)

The last port "valid" represents whether the data is valid or not.

You can directly connect the interface to other interfaces(the ports inside the interface will be connected automatically), or you can also connect the ports separately when needed.

***Remember to connect the "valid" port when the slave block also has "valid" port, as shown in Fig. 11b**

2.1.2 DAQ

This is the output port of the digitizer (where the digitizer gives the data to PC)

***Keep in mind that we always read out data from DAQ instead of Channels!!)**

In the "SD1 SFP" you will find that you can choose to display the data of Channel n.(Similar things also happen in the Python code)

This is kind of misleading, because technically you cannot directly read out data from a channel. This program is doing this because, in the default FPGA, channel_n goes directly to DAQ_n. This means you don't need to distinguish between these two here .

However, when we design the FPGA, you can also connect channel_n(or your calculation result from other channels) to any DAQ you like. Now when you choose "Channel n" in SD1 SFP (or our Python code), you actually see the result that connects to DAQ_n)

2.1.3 Delay

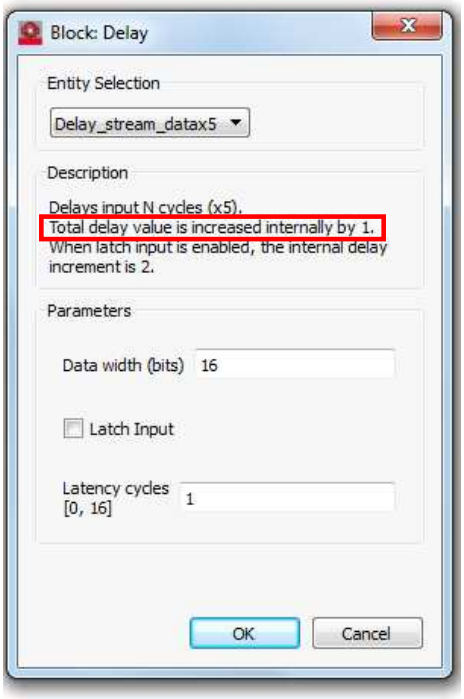


Figure 12: Delay Block Generics

The function of this block seems straightforward. However, the description of the "total delay value" is kind of confusing.

***In our test, we find that "Total delay value is increased internally by 1" actually only happens when you enter "0" in the "Latency cycles" blank. In other words, the delay is 1 cycle when you enter 0 or 1, and n cycles when you enter an integer** $n > 1$

2.1.4 PC Port

(a) Interface (b) Write PC Port Example

Figure 13: PCPort

You can use this block to send data from PC to FPGA. There are 4 PC ports in each FPGA, each port has 2^{14} addresses, and each address can store 32 bits data.

In Python you can use FPGAwritePCport() to send data to the PCPort, so that you can read data from this port in the FPGA.

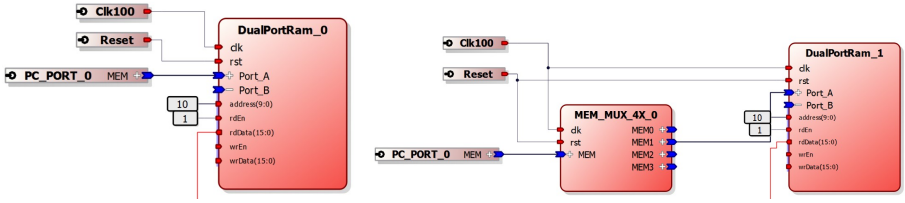
However, an interesting thing we found is that there is a limitation on the number of data you can write into the port each time you call the FPGAwritePCport function.The limitation is about 300.

To write more data to the PCport. Call this function for multiple times. As shown in Fig. 13b.

2.1.5 Dual Port RAM

These are small clusters of RAMs on the FPGA board. We use them in two places.

1. **Read Data from PC Port** In the PC Port section we mentioned that you can bring data to the FPGA through PC Port. In order to read the data from this port, you have to connect the PC Port to a Dual Port RAM.



(a) Read PCPort With DualPortRam (b) Read PCPort With DualPortRam and MUX

Figure 14: ReadPCPort

For example, if you want to read the data you stored in address 10 of your PC Port, you can connect it in this way (Fig. 14a), then the data can be read out from "rdData(15:0)".

You may notice that each Dual Port RAM have only 2^{10} addresses. So if you want to read the data in an address higher than 1023, you will encounter a problem.

This problem can be partly solved by using the Multiplexer (MUX), as shown in Fig. 14b. The MUX can divide the PC Port into 4 parts (0-4095, 4096-8191, 8192-12287, 12288-16384). Now, for example, if you connect the blocks as shown in Fig. 14b, you can read the data in address 4096+10=4106. However, we haven't yet figured how to read the data stored in address 1024-4095, 5120-8191, etc.

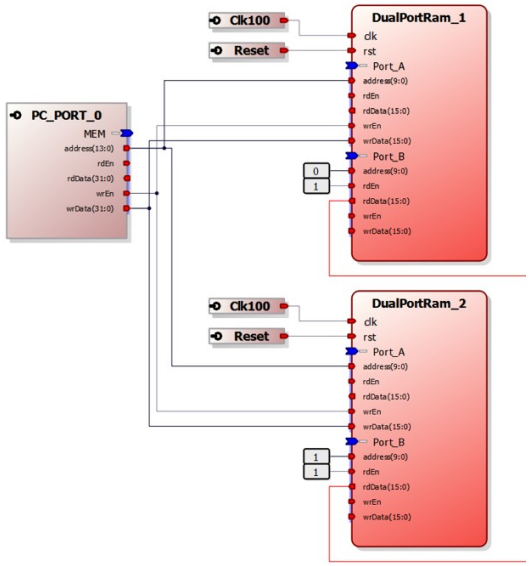


Figure 15: Read Multiple Addresses

If you want to read data in multiple addresses at the same time, you have to use multiple Dual Port RAMs and connect them in the way shown in Fig14[4].

2. Use Them to Build the Sub_Buffer block

[4]You cannot directly connect a MEM output to multiple MEM inputs, since there is an "rdData" port in the MEM interface, which is an input port. And of cause it doesn't make sense to connect one input to multiple outputs.

2.2 Customized Blocks Designed by Us

2.2.1 Demodulate

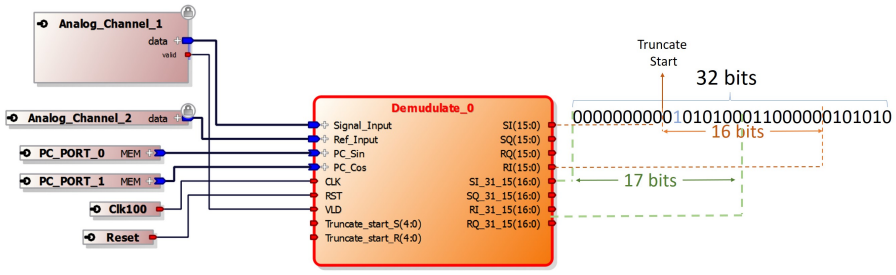


Figure 16: Demodulate Block

Fig. 16 shows how we usually connect the input of the Demodulate block. The signal and reference comes in from Channels 1 and 2 respectively. The sin and cos waves for demodulation are stored in PC Ports 1 and 2.

Inside the Demodulate block, the demodulation result is 32 bits long. We need to truncate this result to 16 bits for subsequent calculations.

There are 8 output ports, the last 4 are all 17 bits long, they are the first 17 bits of the four demodulation results. The first 4 are all 16 bits long, and they give the demodulation results after truncation (The truncation is based on the input "Truncate_start_S(ignal)" and "Truncate_start_R(eference)".

As mentioned in Design Principles, it is hard to say how much we should truncate the data before we do the experiment. In the pre-run firmware, the last 4 outputs will be sent to Find_MSB Blocks to find the most significant bit (MSB) (where the first '1'(for a positive number) or '0' (for a negative number) appears.) If the MSB is smaller than 16, we will just keep all of the last 16 bits.

After we know the MSB of the experimental data, we can send the MSB value to "Truncate_start_S(ignal)" and "Truncate_start_R(eference)" through PC Port. Then, the first 4 output ports will output the demodulation result based on this input.

2.2.2 Find_MSB

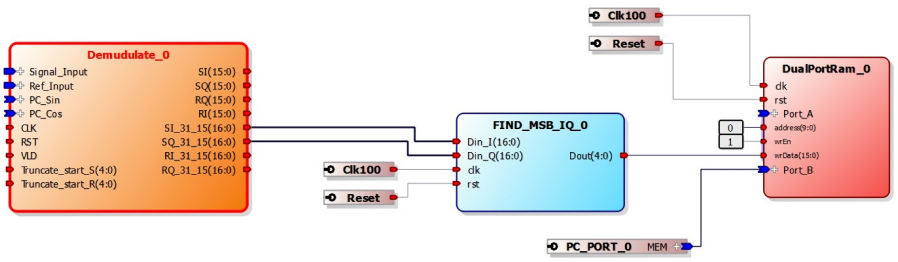


Figure 17: Find_MSB Connection Example[5]

The demodulation result for each input (Signal and Reference) has two parts, I and Q. We seek two truncation points: one for SI and SQ together and another for RI and RQ together. We actually use the Find_MSB_IQ block, which will find Max[MSB[I], MSB[Q]].

The output of Find_MSB_IQ is usually connected to a DualPortRam and will be read by the PCPort.

2.2.3 Integrator

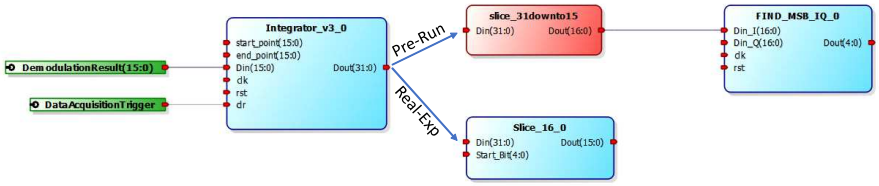


Figure 18: Integrator connection example

This is the integrator we wrote based on the built-in Integrator in M3602A. The entity name is "Integrator_V3".

This integrator will integrate the input signal (Din) from "start_point" to "end_point". "start(end)_point" is measured in unit of number of cycles since the block received the last "rst" or "clr" signal. **The "clr" should be connected to a data acquisition trigger so that the integrator can be reset after each measurement cycle.**

[5]Some ports are omitted in the connection examples, just to make the important connections clearer to the reader.

2.2.4 Read_Weight_Function

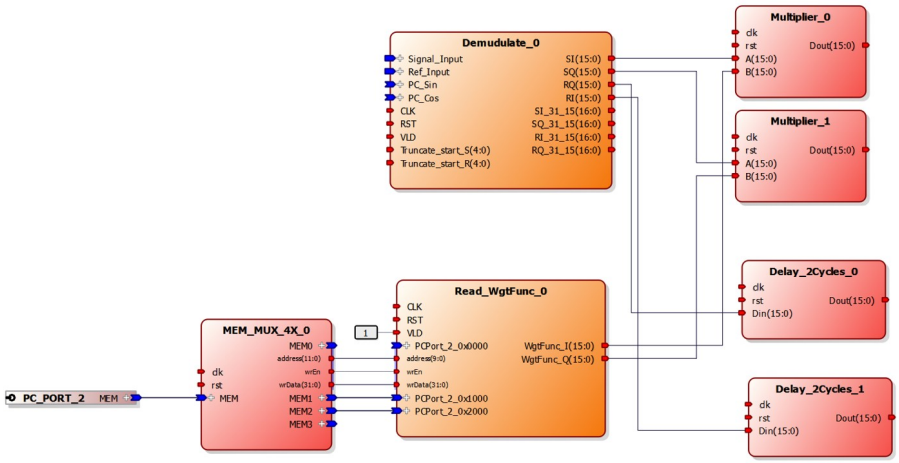


Figure 19: Read_Weight_Function connection example

Read the weight function from PC_Port_2. In the real experiment, many parameters (e.g. truncate point, integration range, etc) are sent to the FPGA through PC_Port_2. There is an address list for these parameters in the FPGA project. Here, the position where the weight function started (stopped) is stored in PC_Port_2 address 10 (11). And the weight function for I(Q) is stored in PC_Port_2 address 4096-5119 (8192-9215).

2.2.5 Get_IQ

Actually this is just some combinations of multipliers and adders to calculate

$$(SI \cdot RI + SQ \cdot RQ) \tag{6}$$

and

$$-SI \cdot RQ + SQ \cdot RI \tag{7}$$

which yield the numerators of equations (4) and (5).

2.2.6 Normalization

This is one of the blocks we put a lot of effort into. We built this block based on the "Fast Inverse Square Root" algorithm popular in video games.

More details on this block are explained in the codes. This normalization block will give you the inverse square root of the incoming data with a delay of 5 cycles and to an accuracy of 8 bits. Since all of our data is in integer form, the output is actually expanded 2^{24} times.

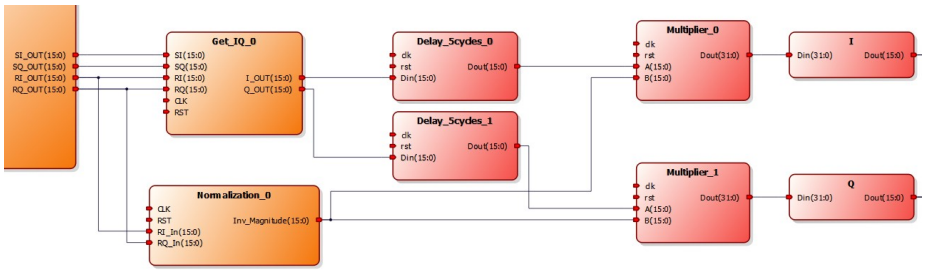


Figure 20: Normalization connection example

2.2.7 Feedback_Trigger

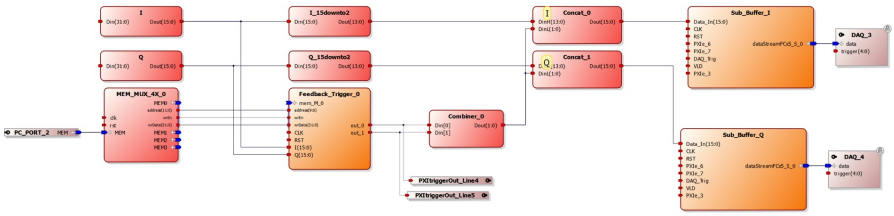


Figure 21: Feedback_Trigger connection example

The input "mem_M_0" is connected to the PC_port_2. In the Python code, you will write the number of states in consideration (usually 3 states) to PC_port_2 address 12, and the position of g, e and f state into PC port 2 addresses 13-18.

This block will calculate the distance from the incoming data point to the three states, and judge which state the incoming data point belongs to. The state information will be coded in 2 bits (either 00-g, 01-e, 10-f, or 11-waiting for data). This state information will be sent to AWG FPGA through the PXI trigger 4,5. This information will also be attached to the last two bits of the I,Q value, which makes debugging and data processing in Python easier.

2.2.8 Sub_Buffer

Recall that the maximum data acquisition cycle for the DAQ is limited to 1032, this buffer is used to solve this problem.

When configuring a digitizer acquisition using DAQconfig(CHx, nPoints, nCycles, triggerDelay, triggerMode), the number of cycles nCycles is limited to 1032. This is a limitation at the level of FPGA implementation. Acquisition misbehavior is going to happen if more cycles are acquired.

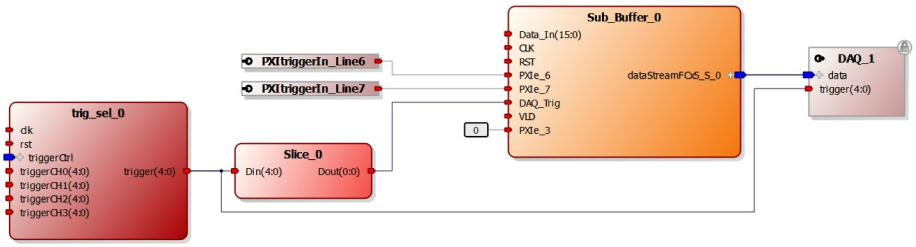


Figure 22: Sub_Buffer connection example

However, the maximum number of points (nPoints) you can acquire in one cycle is $(2^{24} - 1)$. The total number of data points you can acquire after each configuration of DAQ seems large, but that's not the data structure we want.

When we do the experiment, we use the HVI to first send a trigger to the AWG qubit drive port, then we send the trigger to the AWG cavity drive port and digitizer DAQ block simultaneously. Every DAQ trigger is one data acquisition cycle (e,g, see the pre-run HVI design.) All of the data we actually need to acquire in each acquisition cycle is only one IQ pair. So in this way, we waste a lot of buffer space in the DAQ.

This Sub_Buffer block can temporarily store the IQ pairs we got in each experiment. The current design has 5000 addresses. So with this sub buffer, we can acquire 5,000,000 data points in each configuration of the DAQ.

The only drawback is that this new data acquisition method makes the HVI design become relatively complicated. Now, every time you want to trigger data acquisition, you actually send a trigger to the sub_buffer block (we use PXI trigger 6 here). And after you acquire 5,000 data points (when the sub buffer is full), you send a trigger to the DAQ block to acquire these 5,000 data points. Then, we use PXI trigger 7 to reset the sub buffer, and start a new data acquisition cycle (e,g, see the π Pulse HVI design.)

2.2.9 AWG_Output_Shut_gef

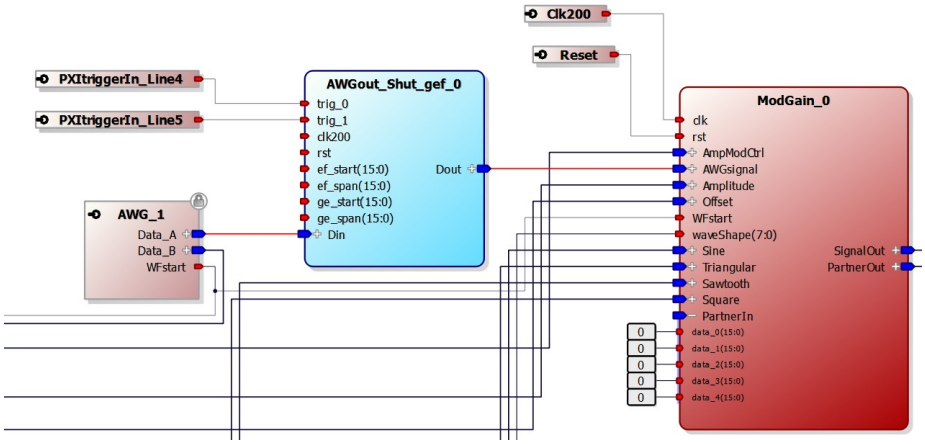
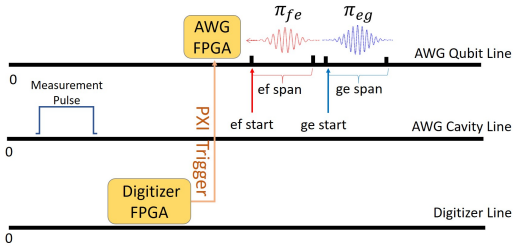


Figure 23: π AWG_Output_Shut_gef Block



State	AWG_Output_Shut_gef Block Behavior	Output Waveform
f	Do nothing	π_{fe} π_{eg}
e	Set AWG waveform to 0 during "ef span"	π_{eg}
g	Set AWG waveform to 0 during "ef span" and "ge sapn"	

Figure 24: Feedback Qubit Control Example

Our method for feedback qubit state preparation is shown in Fig. 24.

The feedback pulses ($\pi_e g$ and $\pi_f e$) are queued after the qubit measurement (cavity drive). After each measurement, the digitizer FPGA will send PXI triggers (PXI trigger 4, 5) to the AWG FPGA, which contains 2 bits of information that represents the state of the qubit. These triggers will be sent to the AWG_Output_Shut_gef block. Then this block will decide which feedback pulse(s) should be present.

For example, let's say we want to prepare the ground state. If the measurement result is f state (10), then this block will allow both of the two feedback pulses to be output. This will flip the qubit from f to e and then e to g. If the measurement result is e state (01), then this block will allow only the $\pi_e g$ pulse to be output. This will flip the qubit from e to g. If the measurement result is

g state (00), then this block will allow neither of these two pulses to be output. This will leave the qubit in ground state.

3 HVI design

The M3601A HVI design environment does not allow us to add comments, so we will explain it here.

3.1 Pre-Run

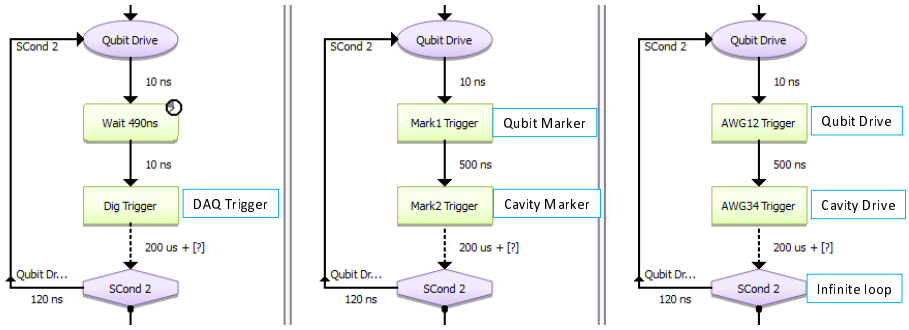


Figure 25: Pre Run HVI Design

This structure is very straight forward. All the waveform control (AWG queue waveform) is done in Python.

This is the universal structure we used for all the experiments before we invented the sub_buffer block. In the Pre-Run, we want to see the IQ trace instead of IQ value, and we don't need to repeat the measurement millions of times, so we don't use the sub_buffer here. And the HVI is simple.

3.2 π Pulse Tune up

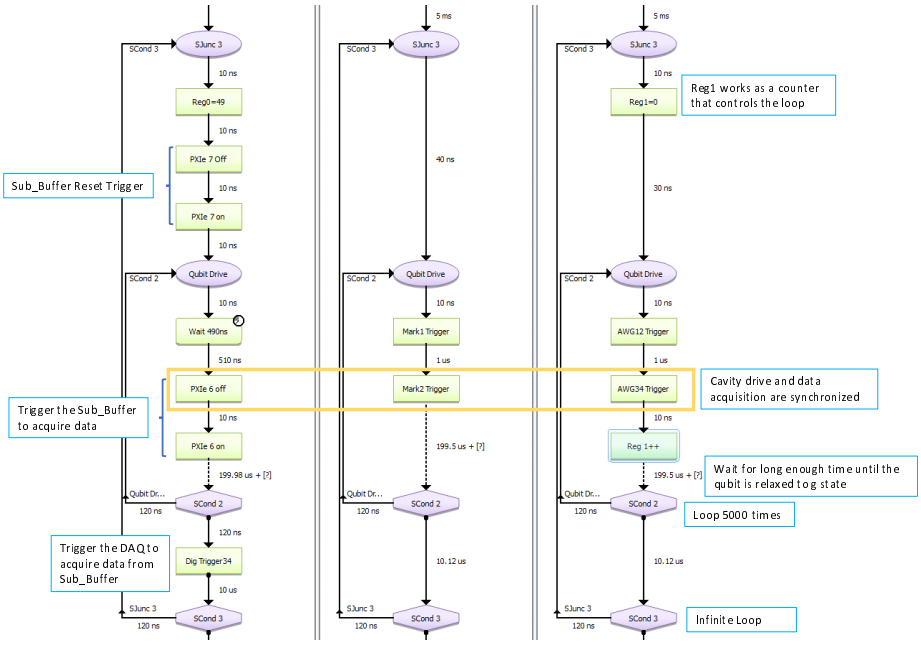


Figure 26: π Pulse HVI Design

This HVI can also be used for qubit state preparation and other similar experiments. For the π pulse experiment, we queue qubit drive waveforms with different amplitudes in AWG 1, 2, and set set the AWG to cyclic mode. Then every time you trigger the AWG, it will output the next waveform in the queue. When all the waveforms have been triggered once, it will start from the first one again.

For a qubit state preparation experiment (e.g., e state preparation), you can just queue one π pulse in AWG 1, 2 and set the AWG to cyclic mode.

Now we have to use the FPGA with sub_buffer. Basically, we need to replace the DAQ trigger with PXI trigger 6, which triggers the Sub_Buffer to take data. After sending this trigger 5,000 times, we send a DAQ trigger, which reads the 5000 data points to the DAQ. Then we send a PXI trigger 7 to reset the sub buffer and start a new data acquisition cycle.

3.3 T1 measurement

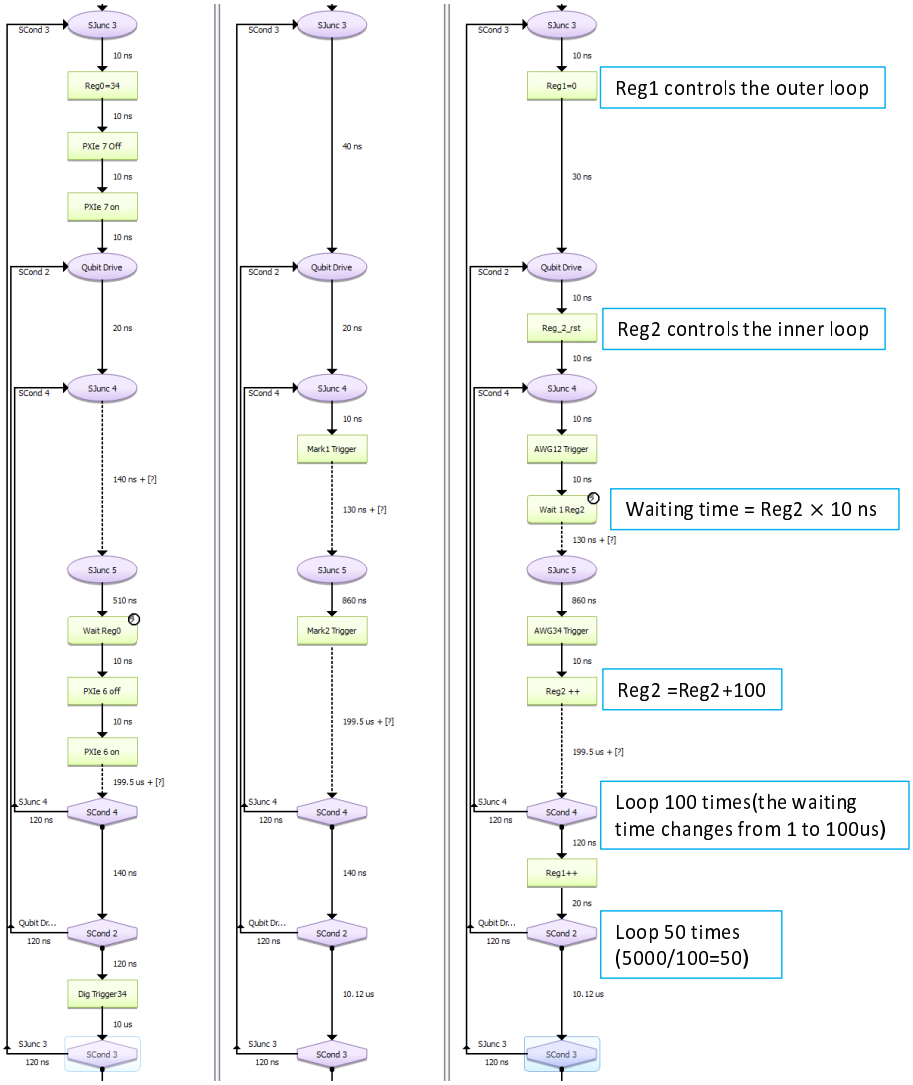


Figure 27: T1 Measurement HVI Design

This HVI is similar to the π pulse one. Here we need an extra loop in which we change the time between qubit drive and cavity drive.