Pigeon code overview

Gal Winer 2015

This document will describe the pigeon FPGA control system. The system was developed by Nitzan Ackerman (others?). The document is written in an effort to understand the system and use it for control in the Firstenberg QNLO lab.

The system has two parts: LabView (FPGA) and Matlab.

Matlab code

1.CodeGenerator class

This class has two constant properties: CommanList and SubcommandList

These properties are cell arrays.

They hold the possible commands and subcommands implemented in hardware on the FPGA host. They become mapped to the internal instruction set of the “CPU”.

The cell array commands need to reflect the possible commands in the lab view implementation.

This seems like a non-scaleable solution, and indeed the code currently has lots of commands which are really in use. ‘NA’ need to think of some possible better solution.

The rest of the properties of the class:

An array which holds the code to be sent to labview (called “code”)

An integer called currentline which holds the current execution point of the code (also called IP for Instruction Pointer)

A stack object which is defined in the file stack.m . this object has the pop and push functionality you would expect of a stack data structure.

Numofreadout is ???

DDSCurrentState is ???

DDSBusCurrentAddress is a 1X6 row vector which is initially empty

DDSBusCurrentData is a 1X8 row vector which is initially empty

There’s also one dependent property called codenumoflines (dependent prop means it is calculated each time it’s requested)

The constructor function

Initializes the currentline to 1

Initializes stack with an empty stack object. (takes no params)

Initialized code to a matrix of zeroes 999X4 and ensures the data type of all entries is int16 (signed, 16 bit)

By this point the only real work that has been done is in the Stack constructor.

Stack Object

The stack object is very simple: it has two properties, a row vector with dimensions 100X1 (data type double) which initially all zero – this is the stackarray and a stackpointer (initially 1) which points to the current position in the stack. The constructor doesn’t do anything because the props are initialized in the properties section.

There are just three methods in the stack: push, pop and IsEmpty.

Push takes a ref to self (automatic) and the element to be pused onto the stack. It places it in the stackarray. It then increments the stackpointer by one.

The pop function checks if the stack is empty. If it isn’t it decrements the stackpointer and returns the element from the top of the array. If it is, we can’t pop.

Isempty just checks if the stackpointer is pointed at 1 (empty) or not.

Pulse Object

It has a property called Timing which is a struct with three fields: VCOSetFreq, DDSsetFreq and DDSsetPhase.

Other props: channel, Tstart, Tend, freq, setFreq, phase, setPhase, amp, setAmp.

The constructor method:

The constructor takes the parameter ts which stands for Tstart for the pulse. It also takes a width param. If width is 0, the pulse will be on for a single clock cycle. If width is -1, the channel will be “HIGH” at all times except for the instant ts (single cycle, again).

The clock cycle time is 25 nS. To make all times in microsecond there is a X40 multiplicative factor in the code.

If we give freq, amp and phase params, the ctor puts them in the correct params and raises the flags indicating they have been set.

The shift method takes a parameter t and uses it to shift the start and end time of the pulse object.

The Sequence2TimeLine method takes an array of **pulses**. The array of pulses is a matrix whose columns each represent a single pulse.

It then generates **timearray** which is a matrix which has 4 rows per pulse and 4 columns. Each row has the following structure: [time,channel,instruction,parameter]. An instruction is an integer in the range 1-5. Each value has a different operation: start=1, stop=2, setfreq=3,setphase=4 and donothing=5.

It is noted that in this function (oddly) time is in cycles and NOT in microseconds. This may not be so off if we remember (will later learn) that the timearray output goes to the FPGA directly.

**Precede** is a variable which saves a 0.5 micro-second time delay (20 clock cycles). It is used in case we set freq/phase/amp so that we first set the parameter and only then perform the operation we want to perform.

We then go to the business end of the function: for each pulse, we check the pulse duration (end time – start time).

We have several cases:

Zero duration pulse (single ON pulse): we save a row to the timearray matrix with the start time, the appropriate channel, the correct operation (1) and 0 as the additional parameter. We then increment the index variable which keeps track of where we are in the timearray.

Negative duration pulse is an OFF pulse. We save a single row to the timearray with instruction #2 and 0 parameter. We then increment the index and zero the precede variable (why?)

A positive duration pulse saves 2 rows to the timearray: one On row and one OFF row after that.

We then check if there is a change request for amp/phase/freq. (this is indicated in the set flags that were initialized by the Pulse ctor function). If there is a request, we save a row with the appropriate command and parameter and increment the precede variable by an additional 20 clock cycles. It seems that if we set on + phase+ freq+amp + off we won’t have enough rows in the timearray structure (5 rows per pulse, not 4). I assume this is not a likely use case or something. It still seems weird. ASK NITZAN.

The entire “parsing” procedure described above is repeated for all pulses. Once that is done, the rest of the timearray matrix is removed and the whole matrix is sorted. The function sortrows sorts by the first column, then by second, etc. so we will get the rows sorted by start times, then by channel, then by instruction and eventually by parameter (this last one probably doen’t matter, but this is what the function does)

The PlotTimeLine function takes t – a timeline returned by Sequence2TimeLine and is supposed to make a nice plot of how our channels turn on and off. **This is currently broken** because PulseCahnnelInfo is a separate object now.

PulseChannelInfo is a function that holds information on the hardware implementation of a channel. This is a bad way to implement things. The info should be held in a text file. Pulse channel info could be refactored so that it parses that file.

The function begins by declaring a persistent variable. This type of variable remains in memory even after the function exists and is available for further use by the function in subsequent function calls.

In the first time the function is called, the variable (called info) is empty. This is identified and the function initializes the value of info to hold a cell array of structs. Each struct must have the fields: ChannelName and ChannelType. Channel name is a name which has some meaning in the experiment e.g. DigOut0 or AOM1 or PUMP\_AOM or something like that. Channel type is ???. it can also have additional field which much have associated cases inside a switch structure in the GenSeq function in the CodeGenerator object.

If it’s not the first time the function is called we already have an info cell array.

If the first param given to the function is numeric it is assumed this is the channel number we ask to query. The second param is assumed to be one of the fields in the cell array of structs (ChannelName, ChannelType etc). it then returns the value saved in the struct for this channel and field. If the first param is not numeric, it is assumed to be the name of one of the channels and the channel number is returned. In this case no further params are expected (they will be ignored).

CodeGenerator methods

GenSeq takes an array whose entries are Pulse Objects. [Pulse(…),Pulse(…),…]

It first uses the static method Sequence2TimeLine which is stored in the Pulse class to generate a timeline from this array.

The number of pulses in the array is then counted. Remember that the timeline is a matrix whose ROWS each represent a single pulse.

We want to be able to keep track of when the last even occurred. We do this with the variable lasteventime. If we pass a single argument to the function (numerical) after the pulse array, this is assumed to be the time (in microseconds) since the last event. We pass a **positive** number but inside the function it will set lasteventtime to **negative** that number. if we don’t pass anything after the pulse array lasteventtime just gets set to 0.

We then reach the main part of the function:

For each of the pulses in the pulse array, we start be generating a pause until the next event time.

The last event time is then set to the current entry in the time line (first column in the current row of the timeline)

The channel, operation and param are then taken from the current timeline row.

The channel type is determined using the PulseChannelInfo function. **Note: it just occurred to me that the correct implementation of PulseChannelInfo is probably via a Containers.Map.**

We then go to a switch statement for the operations.

Operation 1 is turning on a switch.

If it’s a VCO or DIG channel, the code line is populated with the structure appropriate for command 2, subcommand = DigitalSwitch value for that channel (as implemented in ChannelInfo), OnIs, 0. [2,##,OnIs=(1/0),0]. Indeed, the CPU internal implementation for opcode 2 is Digital Ouput for the channel in the subcommand and by giving the requested channel the OnIs value we are logically correct in identifying this section as setting the Digital channel to On.

The initial implementation we got checks for a PMT channel which generates a command 3, subcommand 3 line the code. This is a particular example used in Ozeree lab for reseting some photon counter. I keep this in for now, as it will serve as a template for whatever we would later need. The entire command 3 is related to photon counting.

Operation 2 is turning off a switch

It’s similar to operation 1. If the channel type is VCO or DIG we make a [2,#,~OnIs,0] command.

If it’s a PMT, RegB on the FPGA is set to 0 using the 4,1 command

Then either the PMT1 local variable is added to the value of RegB if the channel operation is 1 (subcommand 4,1) or nothing happens (subcommand 4,2). 1 or 2 is selected by the ChannelInfo ‘operation’ value (value of the ‘opration’ field in the channel struct inside the info cell array from the PulseChannelInfo function).

RegB is then sent to the FIFO using command 7,1.

The current line is hopped by 3 (WHY?)

Numofreadout is incremented by 1. **I don’t know what this variable is**

Operation 3 is setting a frequency

Based on channel type, different things happen.

Only a channel with ChannelType=VCO is applicable. If it’s a VCO, the required freq is taken from PulseChannelInfo. THERE SEEMS TO BE AN IMPLEMENTATION ERROR NOW. WHERE IS FREQ SET? Ask.

Assuming we are able to get freq, it is turned to a int16 inside PulseChannelInfo and then pushed to the correct channel using command 1, subcommand = channel #.

Operation 4 is setting a phase

This is not implemented at the moment

Operation 5 is setting a amplitude

Only a channel with ChannelType=VCO is applicable. If this is the channel type, a set analog output command is executed (command 1,ch#). With a parameter. ASK NITZAN.

This completes the GenSeq function.

GenSetAO

This function generates an analog output instruction in the code matrix. It has a switch statement whose cases are the analog output channels to send the output Var to. Only outputs which are implemented in hardware can be possibilities in the switch statements and for a GenSet command to work, we need to manually add an option in the switch block.

GenRegOp

this section has a bunch of different commands. Is has two (and only two) optional parameters: cmdVar1 and cmdVar2.

‘RegA=’ uses command 4,0 to set RegA to cmdVar1

‘RegB=’ uses command 4,1 to set RegB to cmdVar1

‘RegC=’ uses command 4,2 to set RegC to cmdVar1

‘RegD=’ uses command 4,11 ASK NITZAN. Hardware implementation does not match!!

‘RegA=+1’ uses command 4,4 to increment RegA by 1

‘RegC=+1’ uses command 4,3 to increment RegC by 1

‘RegD=AI1+par1’ uses command 4,9 to set RegD to AnalogIn1 + par1

‘FIFO<-RegA’ uses command 7,0 to push RegA to FIFO

‘FIFO<-RegB’ uses command 7,1 to push RegA to FIFO

‘FIFO<-RegC’ uses command 7,2 to push RegC to FIFO

Note both the above FIFO operations have hardware implementation the performs a NOT on flag 2. (count FIFO size)

‘FIFO<-RegD’ uses command 7,5 NO HARDWARE IMPLEMENTATION ASK NITZAN.

‘FIFO<-AI1’ uses command 7,4 to push Analog In 1 to FIFO

'RegD=RegD\*par2\*2^par1' uses command 4,10 to update RegD to par2 \* (2^par1) ASK NITZAN WHY.

ASK NITZAN: why these methods are grouped together in this way.

GenPause

Takes the pause time in microseconds. **It is used to trigger command 4 subcommand 7 in the pigeon.**

If the delay is longer than 1 minute, it immediately sets a debug point in the code (nice touch) and returns.

If the delay is longer than 10 seconds, it just warns about this.

After that, the delay time is converted to clock cycles with another factor 40. All these factor 40s really have to be turned to a variable with a proper name and not magic numbers. What is we change clock rate? Bad practice!

If the delay is longer than a single clock cycle, it’s converted to int16. Int16 is a vector with length 2: a value at pos1 and a sign at pos2. The command is then stored in the obj.code field (remember: a 999X4 array of int16), in the current line, where it has the correct structure: [4,7,c(1),c(2)] : the command, subcommand, length of delay, sign of delay (ahead or behind).

**Question:** There is no global clock. Each pulse object has its own timeline. How does the system know how to work with commands (like delay) where the parameter refers to an EARLIER point in time?

GenPauseMemoryBlock

Sets command 4,11. ASK NITZAN WHAT THIS DOES. Par1 doesn’t seem to be flowing anywhere in the hardware implementation.

GenWaitExtTrigger

Sets sleep counter to 40\*5 (ASK NITZAN: 5 clock cycles in microsecs?) using command 4,7

Note: the Sleep counter

It is important to understand the hardware implementation of the sleep counter. The sleep counter is a local variable in LabView. It is initially set to 0 and it’s value is manipulated between loops of the single cycle timed loop, and the manipulated value is sent to the next loop cycle via a shift register.

In each cycle, there is a check to see if the counter is >0. If it is, command block and parameter block case structure inputs are set to 0. The sleep counter is also decremented by 1 in each cycle if it is positive. If it’s isn’t, then it remains the same. There is also a test to see if the value is less than 2. Only when it is less than 2, does the Instruction Pointer (IP) increment by 1 (the program moves on).

GenFinish

Adds the 8,0 command to code. This is the command to halt the program. [8,0,0,0]

It then adds a 0,0 command to the next line of code. [0,0,0,0]

Increments current line by 2 and zeros the rest of the code matrix.

Pigeon Flags

|  |  |
| --- | --- |
|  |  |
| 1 |  |
| 2 | count fifo size |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 | Halt and exit program |

Pigeon commands

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Command | SubCommand | Params | Notes |
| Pause | 4 | 7 | Delay in microseconds |  |
| DigitalOutput | 2 | channelNum | None (0) |  |
| PMT Reset | 3 | 3 | None(0) | Maybe not useful for us. All subcom. With command=3 are related to photon counting mechanisms. |
| Set RegB to Par1 | 4 | 1 | Par1 | In lab view the text is opposite (Par1 to RegB) I think this is wrong. Check. |
| Set RegA to Par1 | 4 | 0 | Par1 |  |
| Set RegA to Par1 | 4 | 2 | Par1 |  |
| Push RegB to FIFO | 7 | 1 | None |  |
| Push RegA to FIFO | 7 | 0 | None |  |
| Stop and reset program | 8 | 0 | None | Sets control flag 7 to True |
| Increment RegA by 1 | 4 | 4 | none |  |
| Increment RegC by 1 | 4 | 3 | None |  |
| Set regD to analog in 1 + par1 | 4 | 9 | Par1 |  |
| Set external trigger to rising edge | 5 | 4 | none |  |
|  |  |  |  |  |
|  |  |  |  |  |