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.

We can query the channel info by giving the function parameter.

CodeGenerator methods

GenSeq(obj,arrayofpulses,varargin)