Documentation for digital pattern generator-based imaging controller

Ryan Thomas

March 17, 2020

1 Introduction

This document describes the design and operation of the digital pattern generator (DPG)-based imaging controller used in the Kjærgaard lab. While both this controller and its predecessor are implemented using a field-programmable gate array (FPGA), namely the Xlinx Spartan 3AN development board, the DPG imaging controller has a much simpler FPGA architecture and is more easily customizable than the previous, bespoke version.

Whereas the previous imaging controller had many different modules, each of which controlled a different aspect of the timing sequence, the DPG solution has only two modules: the DPG itself and a module to generate triggers for the FlexDDS. The reason that these are separate is that we need millions of triggers for the FlexDDS at a fixed rate and duty cycle, and using a DPG to generate these would require an enormous amount of memory. It is easier to use a dedicated module for this purpose.

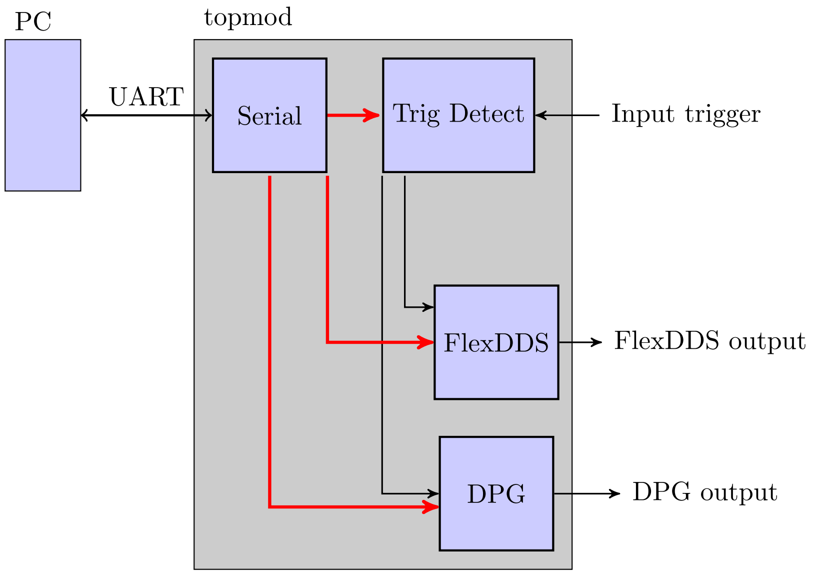

Figure 1: Diagram of the DPG-based imaging controller. A PC transmits parameters to the FlexDDS trigger generator and a digital pattern to the DPG. The sequence starts either when an input trigger is received or when a software trigger is issued by the host PC over serial. Thick red lines indicate the transmission of information to and from the serial controller.

2 Hardware

2.1 Digital Pattern Generator (DPG)

A DPG is simple to understand – it is a device that stores a list of digital output patterns and the times at which they should be enacted. In the case of the current DPG, these are stored as a series of 40-bit (5 byte) instructions, where the most-significant byte (MSB) is the type of instruction and bytes 0 to 3 (for a total of 32 bits) are the *data*. There are currently three types of instructions:

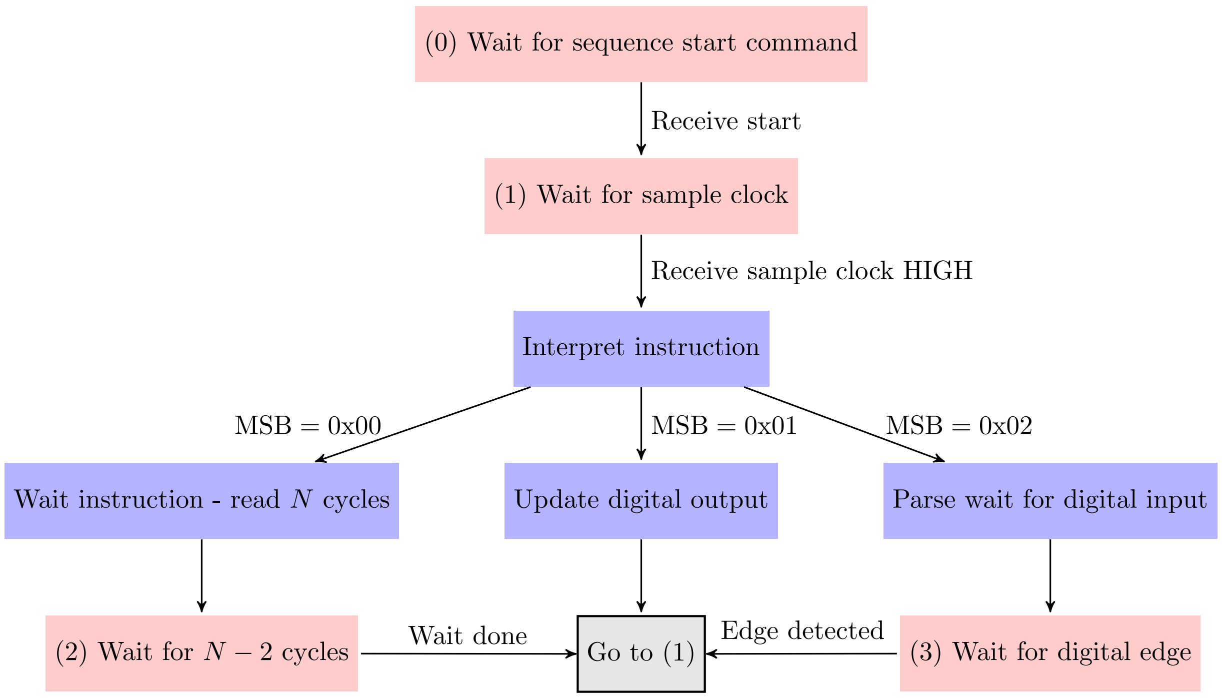
(0) Wait for sequence start command
Receive start
(1) Wait for sample clock
Receive sample clock HIGH
Interpret instruction
MSB = 0x00
MSB = 0x01
MSB = 0x02
Wait instruction - read N cycles
Update digital output
Parse wait for digital input
(2) Wait for $N-2$ cycles
Wait done
Go to (1)
Edge detected
(3) Wait for digital edge

Figure 2: Diagram of the finite-state machine (FSM) at the heart of the DPG. Red boxes indicate states of the state machine; blue boxes indicate conditional statements. New instructions are requested when a "start" command is received and also whenever a sample clock HIGH is detected. When all instructions are read, the FSM returns to state 1. If a sequence "stop" command is received, the controller returns to state 0.

1. MSB = 0x00 This instruction tells the DPG to wait for a time equal to *data* sample clock cycles.
2. MSB = 0x01 This instruction tells the DPG to output the pattern equal to *data*.
3. MSB = 0x02 This instruction tells the DPG to wait for a digital input event before continuing to the next instruction. Bits 0 to 3 of *data*, interpreted as an unsigned integer, indicate the digital input channel to look at, and bits 8 to 9 indicate the type of edge to look for.

At the heart of the DPG is a finite-state machine (FSM) that executes the reads, parses, and executes new instructions: a diagram is shown in Fig. 2. The idle state of the DPG is the "wait-for-start" state, where the FSM waits for the sequence start command from the PC over the UART (serial) connection. When it receives this trigger, it raises the `seqEnabled` signal and request a new instruction from the block memory that holds the instruction list. Raising `seqEnabled` enables the detection of input triggers, and when the input trigger is received the sample clock starts. The sample clock is set to 1/4 of the FPGA master clock (but with a duty cycle of 1/4), so with the current master clock frequency of 100 MHz the sample clock is 25 MHz. This means that new instructions are processed every 40 ns. The factor of 1/4 was chosen to ensure that there is sufficient time to read new instructions from the memory between successive sample clock edges.

When the FSM detects that the sample clock is high it issues a request for a new instruction at the same time that it parses the current one. When the current instruction number, given by the memory address, is less than the number of stored instructions the FSM also raises the `seqRunning` signal which tells the sample clock generator to continue running. If the current instruction number is equal to the maximum address, the FSM raises the `seqDone` signal which tells the controller on the next sample clock edge to read the first instruction again and return to the "wait-for-sample-clock" state (state 1 in Fig. 2). If at any time a "sequence stop" command is received, the FSM immediately returns to the idle state (state 0).

If the current instruction is a digital output instruction, the controller routes the *data* part of the instruction to the signal `dOutSig`, which is routed to the physical output when `seqRunning` is high. If `seqRunning` is low, then the digital output is routed from the signal `dOutManual` which is the manually set value. This signal can either be a default output state, or the user can set it as needed for testing.

If the current instruction is a wait instruction, the FSM reads the wait time from *data* as a delay of N sample clock cycles, where N is a 32-bit unsigned integer. The FSM proceeds to the "wait" state (state 2) and waits for $N-2$ cycles - the -2 is necessary to account for the sample period during which the instruction is parsed and the periods during which the counting is done. By choosing to wait for $N-2$, the delays between updates of the digital output are N sample clock cycles. Note that the controller cannot wait for 0 cycles, and asking it to wait for 1 cycle will result in it waiting for 2 cycles. In order to have a delay of 1 cycle between successive updates to the digital output, the user should have two successive digital output instructions rather than separating them by a wait time.

Finally, if the current instruction is a wait for input instruction, the FSM reads the input bit from bits 0 to 3 of *data* corresponding to an unsigned integer from 0 to 7. The edge type to look for is encoded in bits 8 to 9 with a "00" being a falling edge, "01" being either a falling or rising edge, and "10" being a rising edge. When the correct edge is detected, the FSM moves to state 1. An edge type of "11" is not used and will immediately move to state 1. Digital inputs are detected synchronously with the master clock, so any input signal must be high for at least one master clock period (10 ns). Currently, no digital inputs are used, but this should be relatively easy to change in the future if needed.

2.1.1 Programming the DPG

The DPG is programmed using a relatively simple set of commands which are sent to the FPGA using a serial (UART) connection. Commands are interpreted least-significant bit (and byte) first. Some commands set the controller up to read the next command(s) as a numerical parameter, or, in the case of memory uploading, the next series of commands as writes to memory. A table of commands is given in Table 1 Status information from the DPG is sent with the two most-significant bits being the value

Command	Action	Subsequent command(s)
0xID_00_00_00	Sequence start	None
0xID_00_00_01	Sequence stop	None
0xID_00_00_02	Sends status information back to PC	None
0xID_00_00_03	Sends current value of `dOutManual` back to PC	None
0xID_01_00_00	Set value of `dOutManual`	Next 32-bit command is interpreted as `dOutManual`
0xID_02_*xx*_*xx*	Upload instructions to memory - see text	Next 0x00_00_*xx*_*xx* commands must be 40-bits - see text

Table 1: Command options for DPG. ID is the DPG's serial ID, currently set to 00.

of `seqEnabled` and `seqRunning`, respectively, and the remaining bits represent the address in memory that is currently being accessed.

Uploading instructions to memory is slightly different from other commands. To upload instructions to memory, one first needs to send the command 0xID_02_*xx*_*xx* to the FPGA where *xx*_*xx* is the hexadecimal representation of the number of instructions $N_{\text{instr}} - 1$ that will be sent to the FPGA, where N_{instr} is the number of instructions. Currently, the block memory where these instructions reside can handle up to $2^1 1 = 2048$ instructions, but one can change that in VHDL if more instructions are necessary. Obviously with the current encoding scheme the maximum number of instructions that can be stored is $2^1 6 = 65536$, although other encoding schemes could be used. Once the previous command is sent and the FPGA is primed to receive instructions to write to memory, the user must transmit N_{instr} 40-bit instructions in least-significant bit first format, where the least 32 significant bits are the data and the most-significant byte is the instruction type. A typical set of instructions for programming the DPG could be

Stop DPG	0x00_00_00_01
Prime for receiving manual values	0x00_01_00_00
Send default values (all 0)	0x00_00_00_00
Prime for receiving 10 instructions	0x00_02_00_09
Set all outputs to 0	0x01_00_00_00_00
Wait 10 cycles	0x00_00_00_00_0A
Raise bit 0	0x01_00_00_00_01

Wait 2 cycles	0x00_00_00_00_02
Raise bits 31 and 15	0x01_80_00_80_01
Wait 20 cycles	0x00_00_00_00_14
Lower bit 31, raise bit 1	0x01_00_00_80_03
Wait 1 cycle, lower bit 1	0x01_00_00_80_01
Wait for rising edge on input 0	0x02_00_00_00_20
Lower all bits	0x01_00_00_00_00
Start DPG	0x00_00_00_00

It is not strictly required that one stop the DPG before uploading instructions, but if one does not then there could potentially be read/write conflicts if the DPG receives a trigger while instructions are being uploaded. Additionally, if the first instruction changes between uploads and the DPG is *not* stopped beforehand, then the first DPG instruction will be incorrect as it is pre-loaded on receipt of a "start" command.

2.2 FlexDDS trigger system

The imaging controller is also used to generate triggers for the FlexDDS. Since we generate several seconds worth of triggers at 200 kHz for three channels, it makes sense to not have these be part of the DPG. Instead, it is a separate module in the FPGA architecture. The FlexDDS system has a serial ID of 0x01, and serial commands can be found in Table 2. There are three FlexDDS triggers, and each can

Command	Action	Subsequent command(s)
0xID_00_*xx*_00	Pulse period for output 0x*xx*.	Next 32-bit command is pulse period in FPGA master clock cycles
0xID_00_*xx*_01	Number of pulses for output 0x*xx*.	Next 32-bit command is the number of pulses

Table 2: Command options for FlexDDS system. The output (0 to 2) is set with bits 8 to 15 of the initial command.

be addressed individually. For example, to set the pulse period and number of pulses for output 1 to 5 µs (200 kHz) and 2×10^6 triggers the user would send

Prime for receiving pulse period	0x01_00_01_00
Set pulse period to 200 kHz (100 MHz master clock)	0x00_00_01_F4
Prime for receiving number of pulses	0x01_00_01_01
Set number of pulses to 2×10^6	0x00_1E_84_80

The FlexDDS triggers are started whenever the input trigger signal is raised high.

2.3 Top-level FPGA architecture

There is very little in the top-level module for this FPGA architecture. One of the main parts is a mapping of signals from the DPG to sensible names: for instance, bit 0 of the DPG output is mapped to the Andor camera trigger, and bit 1 is mapped to the Rb AOM on/off switch. Note that many signals, such as AOM and shutter on/off signals, have inputs as well, and these are combined with the outputs from the DPG using a simple OR gate.

Another important part is the input trigger synchronization process. This process takes the asynchronous, external input trigger and an external trigger enable signal, and creates a synchronous internal trigger signal on the rising edge of the external trigger when the enable trigger signal is high.

Alternatively, to start the controller one can use a "software trigger", which is started by sending the command 0xFF_00_00_00 to the FPGA over serial. This will issue an "external" trigger to both the FlexDDS system and the DPG simultaneously, and it can be very useful for testing.

3 Software

Since programming the imaging controller uses a very simple serial interface, it should be relatively straightforward to write software in the user's preferred language that they can use to program the device. A suite of programs has been written in MATLAB, as this is a common language in the lab. The DPG is controlled via the two generic classes *TimingController* and *TimingControllerChannel*, while a more specific class for the current imaging controller is *SpartanImagingController*. The latter class is a subclass of *TimingController* and mainly has named properties that are mapped to particular digital output bits.

A slightly different class, called *SpartanImaging*, provides a wrapper for an instance of *SpartanImagingController* as well as an associated class for controlling the FlexDDS trigger system using the class *SpartanFlexDDSTriggerSystem*. It also has an instance of *StatePrepPulses* which provides an interface for creating state-preparation pulses (microwave and radio-frequency).

The MATLAB code has been commented so users should be able to figure out how to work the software by reading the comments and looking at examples. Briefly, though, the *TimingControllerChannel* provides the function *at* that takes in a time, a value, and a time unit (default is seconds) and creates an "event" corresponding the change of state of that channel to *value* at the specified time. Additional functions *before* and *after* (with the same arguments as *at*) can be used when chaining events together. For instance, suppose one has a instance of *SpartanImagingController* named `sp`, and one wants to switch on the Rb probe AOM and shutter at a given time and then turn them off some time later (as would be done for imaging). Possible code would be:

```
sp.probeRb.at(50,1,'ms').after(15,0,'us');
sp.shutterRb.anchor(50,'ms').before(2.5,1,'ms').at(sp.probeRb.last,0);
```

This code uses a number of *TimingControllerChannel* functions. In the first line, the Rb probe signal is raised high at 50 ms and then 15 µs later it is lowered. The shutter signal is slightly more complicated – we want it to go high 2.5 ms before the probe signal (so that the shutter has sufficient time to open) and then close when the probe signal goes low. We can use the *anchor* method to set the internal channel property `lastTime` to 50 ms without adding an event. Then we can use the before function to specify an event prior to the last time added. Note that `lastTime` is the last added time, but not the last time in the sequence – events are not sorted after each addition. Finally, we can use the function `last` to get the last time written to the probe signal and use that for specifying when the shutter signal should be lowered. Once times are written to the channel as events they are always in seconds, so no unit is required. The channel sequences that a user creates can be plotted using the *plot* method.

The *SpartanImaging* class provides familiar properties, such as cross-beam on time and time-of-flight, for the creation of standard imaging sequences. Simply set the properties as was done in previous iterations of the controller and it should create an equivalent imaging sequence. The *SpartanImaging* class also has an `expandVariables` method, just like the previous *spartan* class, which means that one can specify one or more properties as vectors and the class will expand other properties to the maximum length.

Once the properties are set, to create a basic imaging sequence one needs to run the functions

```
si.expandVariables.makeSequence.upload;
```

where `si` is an instance of *SpartanImaging*. This will expand the variables, create a default sequence, and then upload that sequence to the controller or create binary files (in the case of multiple sequences). If you want to customize the sequence, you need to do so after `makeSequence` because it resets the channels and the controller. Otherwise, you can ignore `makeSequence` altogether and create a sequence from scratch. You can take a look at the total sequence using `si.plot` which will plot all channels where there are events on the same plot. This can be crowded, so you can add an offset between each plot using `si.plot(offset)` where `offset` is the offset between each channel.