ARL-TR-8602 ● DEC 2018

US Army Research Laboratory

Multichannel Time-Interval Measurement with a Field Programmable Gate Array (FPGA) Device

by Michael Don

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8602 ● DEC 2018

US Army Research Laboratory

Multichannel Time-Interval Measurement with a Field Programmable Gate Array (FPGA) Device

by Michael Don
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2018	Technical Report	May–September 2018

4. TITLE AND SUBTITLE
Multichannel Time-Interval Measurement with a Field Programmable Gate Array (FPGA) Device

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael Don

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-WML-F
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8602

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the development of a multichannel time-interval-measurement unit using a field programmable gate array (FPGA). Precise time-interval measurement is required for a number of applications including clock stability analysis, time-of-flight measurements, and particle physics. This research was geared toward pulse-per-second monitoring but can easily be applied to other applications. The FPGA implementation lends itself to a multichannel design and offers a low-cost solution compared to commercial or custom application-specific integrated circuit solutions. The design goal of 10-ns resolution on 10 channels was achieved, with additional proof-of-concept development demonstrating subcycle accuracy.

15. SUBJECT TERMS
time-interval measurement, FPGA, pulse per second, test measurement, digital design

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael Don
Unclassified	Unclassified	Unclassified	UU	53	19b. TELEPHONE NUMBER (Include area code) 410-306-0775

Contents

List of Figures

List of Tables

1. Introduction

This report describes the development of a multichannel time-interval-measurement unit employing a field programmable gate array (FPGA). Precise time-interval measurement is required for a number of applications including clock stability analysis, time-of-flight measurements, and particle physics.[1] This research was geared toward pulse-per-second (PPS) monitoring but can easily be applied to other applications. The FPGA implementation lends itself to a multichannel design, and offers a low-cost solution compared to commercial or custom application-specific integrated circuit (ASIC) solutions. The design goal of 10-ns resolution on 10 channels was achieved, with an additional proof-of-concept demonstration of subcycle accuracy.

Various methods can be employed to precisely measure time intervals. Analog methods convert the voltage of a charging capacitor to a time measurement and are generally very accurate, although they suffer from calibration and stability issues.[2] In its simplest form, digital methods employ counters to measure time intervals. Methods can be used to increase the maximum counter speed, such as dividing the counter into multiple clock domains, using a parallel counter architecture, or employing a linear feedback shift register.[3] Other digital techniques include the Vernier method that uses two oscillators at very close frequencies to achieve an interval measurement and tap delay lines that can achieve subcycle time resolution.[2]

Many of these methods require custom analog or digital designs to achieve high accuracy. Research has shown, however, that low-cost accurate time-interval measurement can be achieved using an FPGA.[4] Although an FPGA does not have as much flexibility as fully custom ASIC designs, high accuracy methods, such as delay line techniques, can be successfully implemented. Thus, for modest time resolution requirements, an inexpensive FPGA design can be more practical than a more expensive custom product. Table 1 lists a few commercial time-interval-measurement products with their prices at the time of writing this report. Few single-product options supported 10 or more channels. The V660DS[5] was one of these and offered the least expensive solution, although additional hardware is needed to provide a versa module Europa (VME) interface. Many products that only support a few measurements are available in VME, peripheral component interconnect (PCI), or PCI extensions for instrumentation (PXI) form factors that allow multiple units to be combined in a single chassis. The cost of purchasing multiple units, as well as the additional cost of the chassis, makes these systems an expensive option for multichannel measurement. Therefore, the US Army Research

Laboratory (ARL) decided to leverage its previous experience with FPGA development[6,7] to develop a low-cost time-interval-measurement system.

Table 1 Time-interval-measurement products

Product	Reference	Form	Resolution (ps)	Channels	Price
GFT2002	[8]	Standalone	1	1	$7,945
GC2220	[9]	PCI	100	1	$5,417
V660DS	[5]	VME	75	12	$8,620
GT667	[10]	PXI	0.9	2	$4,995

In this report, standard counters are implemented using an FPGA to achieve the design goal. The design, simulation, and testing are first presented using a DLP Design* FPGA development board. Due to the obsolescence of this board, the design was transitioned to the Digilent† Cmod A7 FPGA development board. Finally, subcycle time-interval measurement accuracy is then demonstrated using the tapped delay line method.

2. DLP-HS-FPGA3-Based Interval Counter

2.1 Design

The system is required to measure the interval between the rising edge of a reference signal to the rising edges of 10 input signals with an accuracy of 10 ns. Although a wide variety of signals may be measured, it is intended to compare 10 GPS PPS signals to a reference PPS. In the context of this application, the inputs will be frequently referred to as PPS signals. ARL has had previous experience with FPGA programming using the DLP-HS-FPGA3 development board, and therefore this board was chosen for the initial design. Some of the pertinent characteristics of the DLP-HS-FPGA3 are as follows[11]:

- Xilinx‡ Spartan XC3S1400A-4FTG256C FPGA
- 32M × 8 double data rate version 2 (DDR2) synchronous dynamic random-access memory (SDRAM)
- Built-in configuration loader

* DLP Design, Inc. 1605 Roma Lane, Allen, TX 75013
† Digilent, Inc. 1300 NE. Henley Court, Pullman, WA 99163
‡ Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124-3400

- 63 user input/output (I/O) channels: 21 differential pairs and 8 global clocks
- 66.666-MHz oscillator
- USB-powered or 5-V external power barrel jack
- Future Technology Devices International* (FTDI) FT2232H, dual-channel, high-speed USB integrated circuit (IC)
- 3.0 × 1.2-inch printed circuit board (PCB) and standard 50-pin dual in-line package (DIP) interface

The FTDI IC provides a convenient computer interface, allowing the intervals to be reported and saved. Figure 1 shows a high-level block diagram of the system. PPS0 is the reference signal to which PPS1 through PPS10 are measured against. A counter measures the time interval from the reference PPS0 signal to each of the other PPS signals, resulting in 11 counter values including the measurement of PPS0 to itself. At each rising edge of PPS0, the PPS state machine loads the time-interval report message into a first in, first out (FIFO) memory for transmission to the FTDI IC and finally the connected PC.

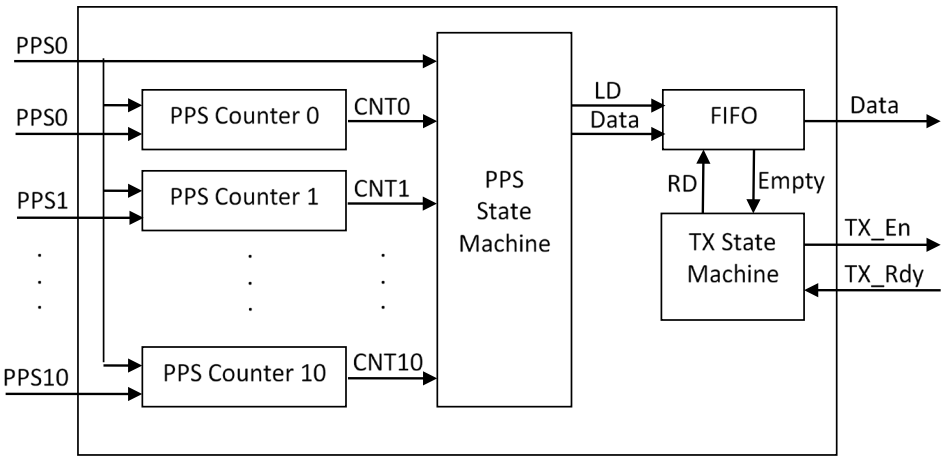

Fig. 1 Multichannel time-interval counter DLP-HS-FPGA3 system diagram

Table 2 shows the structure of the time-interval message. The first four bytes contain a synchronization word listed in hexadecimal. The next 44 bytes contain eleven 4-byte count values. The 32-bit counts are listed most significant byte (MSB) to least significant byte (LSB). The most significant bit (MSb) indicates if a transition was detected since the last transition of the reference input. This is

* Future Technology Devices International Limited (USA) 7130 SW Fir Loop, Tigard, OR 97223

necessary to distinguish between the maximum counter value that indicates an input transitioned the same time as the reference and a maximum counter value that occurs because no transition is detected.

Table 2 Time-interval message format

Data	Byte offset	Length (bytes)	Repetitions	Description
xFE6B2840	0	4	1	Sync. word
CNT	4	4	11	Counter value

The FPGA code is summarized in Appendix A under "dlp_hs_fpga_test.vhd," along with the other very-high-speed integrated circuit (VHSIC) hardware description language (VHDL) files. The I/O pins, summarized in Table 3, are configured first. Eleven test outputs are supplied to match the 11 inputs, providing different frequency pulses with which to test the time-interval measurements. Aside from the PPS and test pins, the CLK input provides a 10-MHz reference that is used to generate the 100-MHz system clock. Ideally, this clock should be supplied from an atomic clock source for high accuracy. The test signals are constructed from the bits of a 32-bit counter. The period of bit x is $2^{(x+1)}$ counts. For example, bit 26 has a period of $2^{27} = 134217728$ clock cycles, which is about 1.34 s. Two setting are used for the test signals, one for real operation and one for simulation. The settings are listed in the notes in Table 3, with the simulation settings in parentheses. Test0 and Test1 are inverted, while the other test signals are not inverted. Real and simulation settings are defined by constants in "MIDASlibrary.vhd" in order to make the design easily configurable. In addition to the test signals, bit 25 of the counter is used to drive a heartbeat LED on the board, indicating that the clock and FPGA program are running.

Table 3 Multichannel time-interval counter DLP-HS-FPGA3 pinout

Pin	Name	Notes
2	CLK	10-MHz clock
7	PPS0	Reference signal
8	PPS1	…
9	PPS2	…
10	PPS3	…
12	PPS4	…
13	PPS5	…
14	PPS6	…
15	PPS7	…
16	PPS8	…
17	PPS9	…
18	PPS10	…
33	Test0	Bit 26 (9) of counter, inverted
34	Test1	Bit 26 (9) of counter, inverted
35	Test2	Bit 23, (6) of counter
36	Test3	Bit 23, (6) of counter
37	Test4	Bit 23, (6) of counter
38	Test5	Bit 23, (6) of counter
39	Test6	Bit 28, (11) of counter
41	Test7	Bit 26, (9) of counter
42	Test8	Bit 25, (8) of counter
43	Test9	Bit 24, (7) of counter
44	Test10	Bit 23, (6) of counter

After the test signals are defined, the PPS measurement module instantiates the interval counters and state machine that loads the reporting message into the FIFO. The interval counters are defined in the `counter_for_pps` module, with the state diagram shown in Fig. 2. In all state machine diagrams, the black text next to an arrow indicates the state transition logic. Red text in or next to a state indicates the setting of output or control signals. Red text next to an arrow indicate the setting of output or control signals during state transition. Rising edges on `ppsr` and `pps` are detected by latching the signals to create delayed versions `ppsr2` and `pps2`, respectively, and then detecting when the delayed versions are low and the non-delayed versions are high. After initialization in the `RST` state, a rising edge on

ppsr triggers the counter value to be saved and then reset, and the got_pps flag to be reset. If pps is currently transitioning or has transitioned in the past, indicated by got_pps, then the MSb of the saved counter is set. When ppsr is not transitioning, the counter is active until a transition on pps is detected. Thus, the counter measures the interval from one cycle after a transition on ppsr to the current ppsr transition. The true interval value is cnt+1, therefore the reported interval must be incremented by the PC.

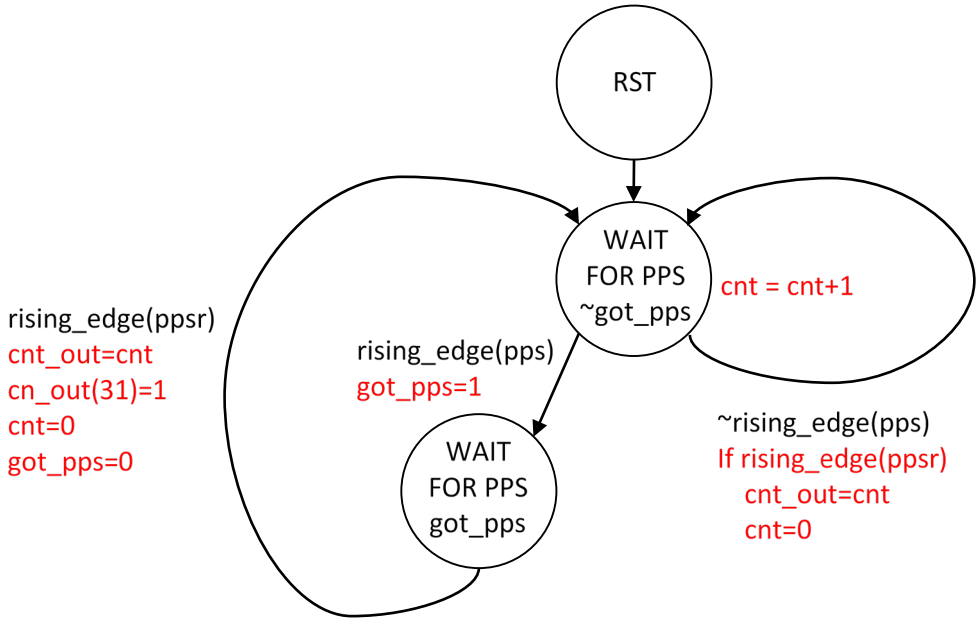

Fig. 2 PPS counter state diagram

The pps_cnt module uses 11 instances of the counter_for_pps module to generate the time-interval message, which is saved in a FIFO. The same configuration is used for the reference signal as the other inputs, allowing the period of the reference signal to be monitored. Figure 3 shows a diagram of the state machine in pps_cnt used to load the FIFO with the report message. After initialization in the RST state, the machine waits in the DELAY state for the reference signal to transition. Once the signal transitions, the cnt variable is used to index and load 4 synchronization bytes into the FIFO. The ld signal here refers to the FIFO load signal. The state machine then transitions through states START0 through START3, loading the MSB to LSB of the counters. The cnt is now used to index the interval count values and incremented after all 4 bytes of each counter is loaded. After all of the counters have been loaded, the machine returns to the DELAY1 state and waits for the next transition of the reference signal.

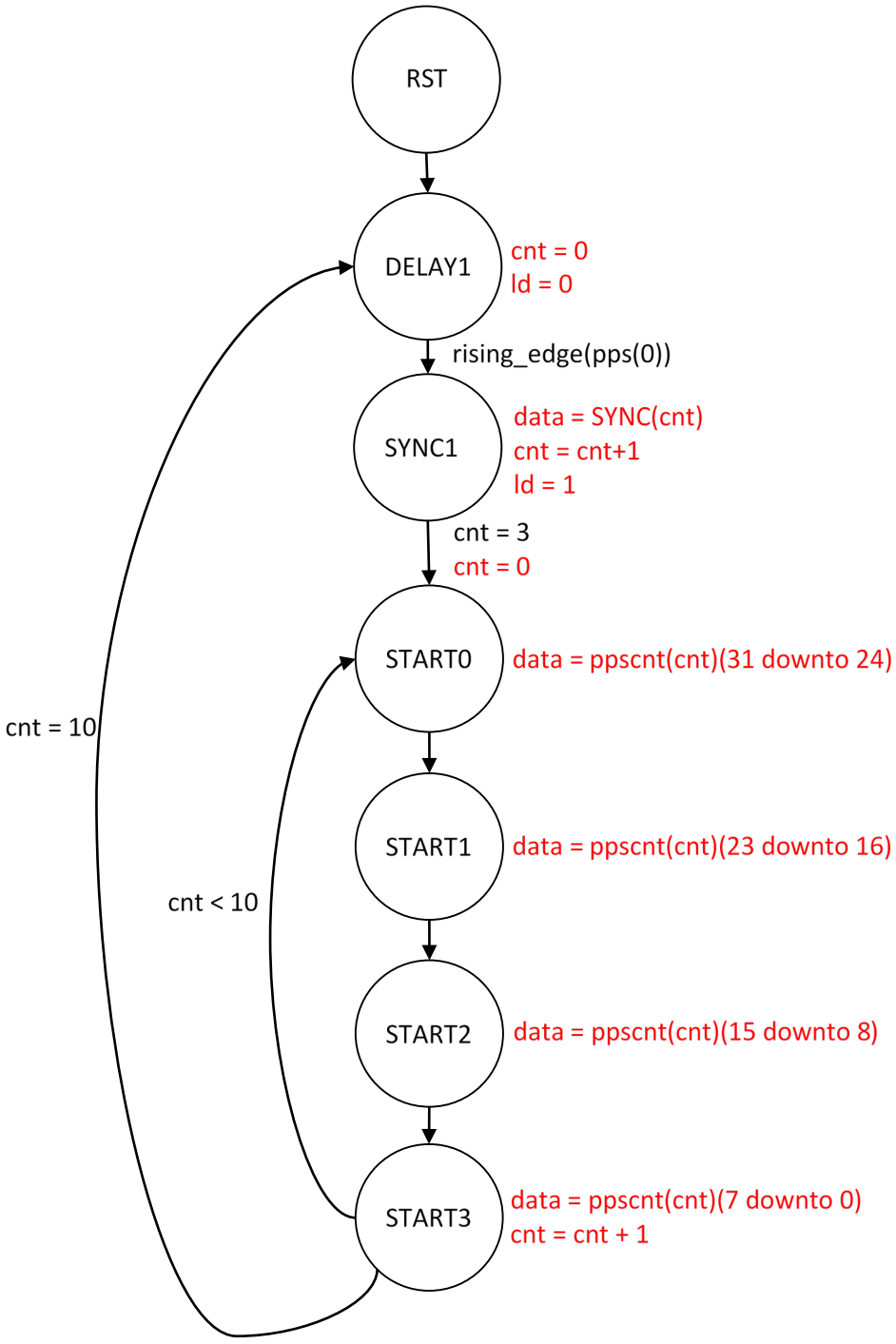

Fig. 3 Time-interval message generation state diagram

Continuing through "dlp_hs_fpga_test.vhd", the FIFO design is a modified version from the GH VHDL Library[12] FIFO with extra logic to initialize the output at reset. The state machine in Fig. 4 is used to read the message from the FIFO and output the data to the FTDI IC. `txe_n_s` is an active low signal that indicates the FTDI

IC is ready to transmit. When this signal is active and the FIFO is not empty, the `rd` signal is asserted to read the FIFO and the machine transitions from `IDLE_S` to `FIFO_RD_S`. The machine next transitions to `WR_USB_S` where `ftdi_wr` is asserted to command the FTDI IC to transmit the data. If the FTDI IC is available to transmit more data, and there is more data in the FIFO, the machine executes another transmission by returning to the `FIFO_RD_S` state; otherwise, it returns to `IDLE_S`. When the system clock is run at high speed, delay must be added in between states to ensure proper operation. Thus, a counter was implemented to add a 4-clock cycle delay between each of the states involved with the FTDI transmission.

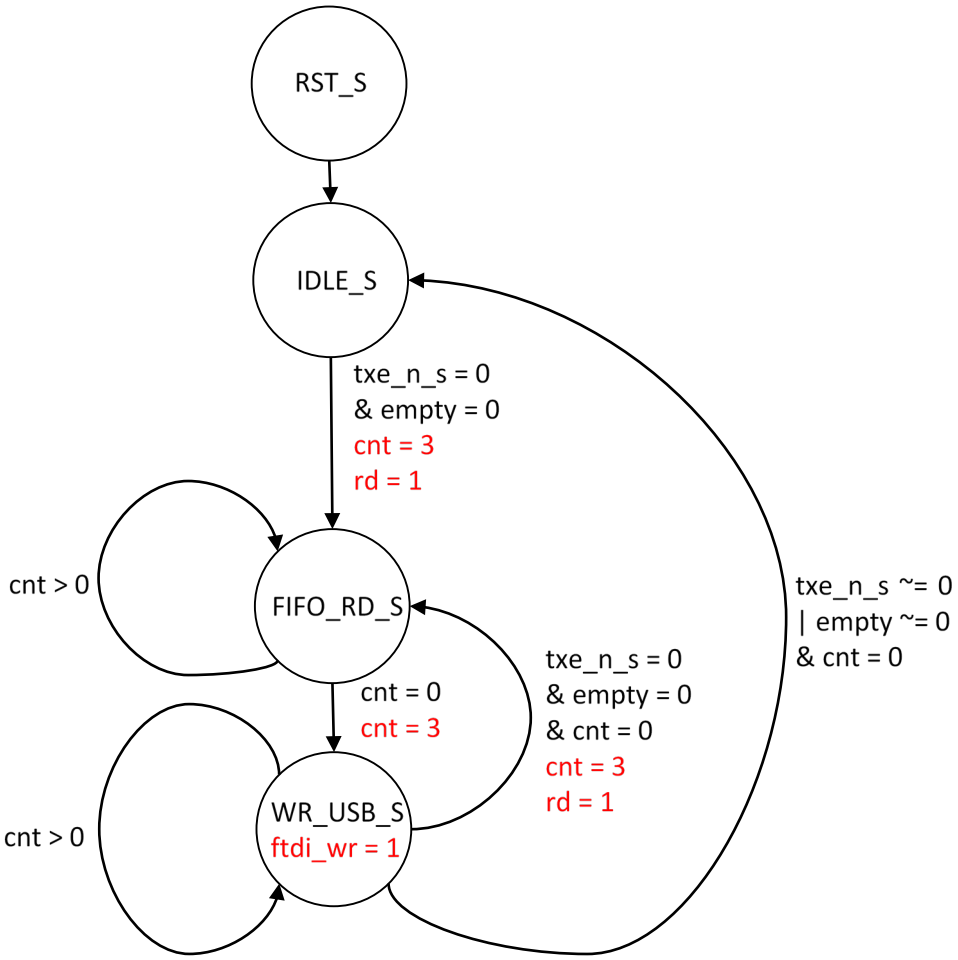

Fig. 4 TX state diagram. Data are unloaded from the FIFO and transmitted to the PC through the FTDI USB IC.

2.2 Simulation

The implemented test signals are not only useful for physical testing, but can be used for simulation as well. Figure 5 shows simulation results of the system with the test outputs defined in Table 3 directly connected to signal inputs `PPS[10:0]` and the counter bits for the test signals set to their simulation values given in Table 3. The counter values are shown as `ppscnt[10:0]`. The MSb of all of the counters is 1 except for `ppscnt[6]`, which correctly corresponds to transitions on all of the inputs except for `PPS[6]`. This indicates that value of `ppscnt[6]` can be ignored. Table 4 summarizes the simulation results. As mentioned previously, the period of bit x of test signal counter is $2^{(x+1)}$ cycles. The actual rising edge of the non-inverted inputs, `ppscnt[2:10]`, is at half of the period, or 2^x. Finally, as stated previously, the counter value must be incremented by 1 to calculate the actual delay. Therefore, the counter values are calculated as $2^x - 1$. For the inverted inputs `ppscnt[0:1]`, the rising signal edge is at the end of the period, leading to a calculated value of $2^{(x+1)} - 1$. All of the simulation results agree with the predicted values.

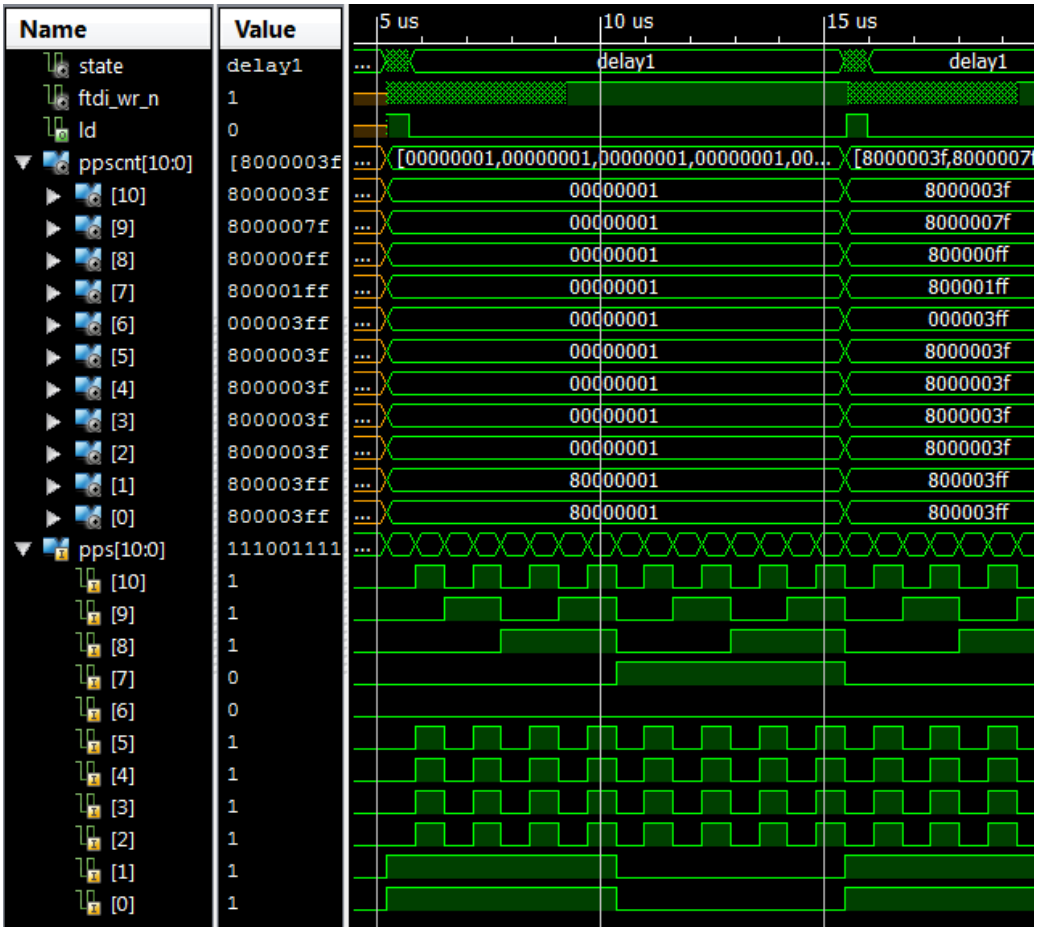

Fig. 5 Multichannel time-interval counter DLP-HS-FPGA3 simulation

Table 4 Simulation results

Counter	Test counter bit	Hex value	Decimal value
ppscnt[0]	9	3FF	1023
ppscnt[1]	9	3FF	1023
ppscnt[2]	6	3F	63
ppscnt[3]	6	3F	63
ppscnt[4]	6	3F	63
ppscnt[5]	6	3F	63
ppscnt[6]	11	3FF	1023
ppscnt[7]	9	1FF	511
ppscnt[8]	8	FF	255
ppscnt[9]	7	7F	127
ppscnt[10]	6	3F	63

2.3 Physical Verification

For physical verification, a 10-MHz clock was applied to the clock input, and the test outputs were used to stimulate the PPS inputs. In real operation, an atomic clock source will provide the greatest accuracy, but for test purposes, a waveform generator sourced the 10-MHz clock. A LabVIEW program with FTDI D2XX drivers, shown in Appendix B Fig. B-1 was created to communicate with the DLP-HS-FPGA3 board and save the report messages as binary files. A MATLAB script, "read_data.m" included in Appendix C, was used to analyze the files and report the calculated interval between each input and PPS0. Table 5 shows the time-interval results of a test where all of the PPS inputs were connected to their respective test outputs except PPS3 through PPS5. The raw counter values were converted to real-time intervals by incrementing them and dividing by the counter clock frequency, 100 MHz. Additionally, a count value matching the PPS0 count indicates that a transition occurred simultaneously with PPS0 and is set to 0. Finally, if no transition was detected, the counter value is invalid, and the time interval is set to "not a number" (NaN). Table 3 lists the Test6 output as 28, while Test0 is 26. This 2-bit difference results in the rising edge of Test6 occurring once in every four periods of Test0, which is also verified in Table 5.

Table 5 Physical verification time results of DLP-HS-FPGA3 board

Input	Message 1	Message 2	Message 3	Message 4	Message 5
PPS0	0	0	0	0	0
PPS1	0	0	0	0	0
PPS2	0.08388608	0.08388608	0.08388608	0.08388608	0.08388608
PPS3	NaN	NaN	NaN	NaN	NaN
PPS4	NaN	NaN	NaN	NaN	NaN
PPS5	NaN	NaN	NaN	NaN	NaN
PPS6	0	NaN	NaN	NaN	0
PPS7	0.67108864	0.67108864	0.67108864	0.67108864	0.67108864
PPS8	0.33554432	0.33554432	0.33554432	0.33554432	0.33554432
PPS9	0.16777216	0.16777216	0.16777216	0.16777216	0.16777216
PPS10	0.08388608	0.08388608	0.08388608	0.08388608	0.08388608

3. Cmod A7–Based Interval Counter

The DLP-HS-FPGA3 was originally chosen for this design due to ARL's familiarity with the development kit. However, the Xilinx Spartan FPGA it employs is antiquated and the board has become obsolete. Digilent's Cmod A7 FPGA board employs a modern Xilinx Artix-7 FPGA and provides a low-cost alternative to the DLP-HS-FPGA3. Some of the relevant features of the board are as follows[13]:

- Xilinx Artix-7 XC7A35TCPG-236 FPGA
- 512K static random access memory (SRAM)
- Built-in configuration loader
- 44 digital and 2 analog I/O DIP pins, peripheral module (Pmod) connector
- 12-MHz oscillator
- USB powered
- FTDI FT2232HQ, dual-channel, high-speed USB IC
- 2.75 × 0.7-inch PCB

Aside from some technical difficulties in transferring a design in Xilinx's older ISE software to the newer Vivado software, there was one major change in the design. The DLP-HS-FPGA3 is configured to use the FTDI USB bridge in a FIFO mode,

loading the data bytes as 8 parallel bits. The Cmod A7 is configured to transmit to the FTDI IC using Universal Asynchronous Receiver/Transmitter (UART) receive (RX) and transmit (TX) signals. The updated system diagram is shown Fig. 6. When the UART module indicates that it is ready to transmit, the TX State Machine reads data from the FIFO, writes it to the UART, and initiates a transmission.

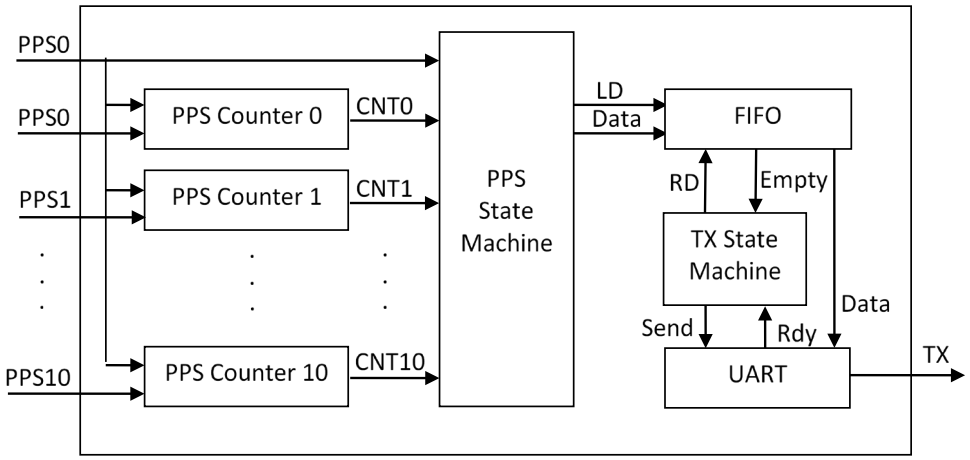

Fig. 6 Multichannel time-interval counter Cmod A7 system diagram

Table 6 lists the new pinout for the Cmod A7 board. A 10-MHz clock test output was added in order to facilitate testing. Simulation and physical verification were repeated for the Cmod A7 implementation, matching the results for the DLP-HS-FPGA3 board. The FTDI bridge appears to the PC as a virtual serial port, necessitating a new LabVIEW program to record the report messages shown in Appendix B Fig. B-2. This program uses the basic virtual instrument software architecture (VISA) serial functions instead of the FTDI D2XX drivers to communicate with the FTDI IC. The same MATLAB script that was used with the DLP-HS-FPGA3 board was used again to analyze the test messages.

Table 6 Multichannel time-interval counter Cmod A7 pinout

Pin	Name	Notes
18	CLK	10-MHz clock input
1	PPS0	Reference signal
2	PPS1	…
3	PPS2	…
4	PPS3	…
5	PPS4	…
6	PPS5	…
7	PPS6	…
8	PPS7	…
9	PPS8	…
10	PPS9	…
11	PPS10	…
37	Test_CLK	10-MHz clock output
48	Test0	Bit 26 (9) of counter, inverted
47	Test1	Bit 26 (9) of counter, inverted
46	Test2	Bit 23, (6) of counter
45	Test3	Bit 23, (6) of counter
44	Test4	Bit 23, (6) of counter
43	Test5	Bit 23, (6) of counter
42	Test6	Bit 24, (7) of counter
41	Test7	Bit 23, (6) of counter
40	Test8	Bit 28, (11) of counter
39	Test9	Bit 26, (9) of counter
38	Test10	Bit 25, (8) of counter

The goal of 10-ns resolution was easily met using a clock speed of 100 MHz. Timing analysis showed that the minimum delay was 5.620 ns, in theory allowing the maximum clock rate to be 178 MHz, giving a resolution of 5.620 ns. This speed was achieved by setting the clock rate goal to 100 MHz. Increasing the clock goal can improve performance by forcing more aggressive placement and routing. Specifying a design goal of 320 MHz, the maximum synthesizable speed for an input clock of 10 MHz, results in a minimum delay of 3.723 ns, corresponding to a clock rate of 268.6 MHz. Higher performance, however, comes at a cost of extra logic utilization. Table 7 shows the FPGA resource utilization using the 100-MHz clock goal, while Table 8 shows the 320-MHz clock goal utilization. Although both

designs are relatively small and only use a small amount of the FPGA's resources, the higher performance design requires more lookup table (LUT) resources.

Table 7 Artix 7 FPGA utilization with 100-MHz clock goal

Resource	Utilization	Available	Utilization (%)
LUT	303	20800	1.456
LUTRAM	12	9600	0.125
FF	918	41600	2.206
I/O	33	106	31.132
BUFG	4	32	12.500
MMCM	2	5	40.000

Notes: LUTRAM = LUT random access memory; BUFG = global buffer; FF = flip-flop; MMCM = mixed-mode clock manager

Table 8 Artix 7 FPGA utilization with 320-MHz clock goal

Resource	Utilization	Available	Utilization (%)
LUT	326	20800	1.567
LUTRAM	12	9600	0.125
FF	918	41600	2.206
I/O	33	106	31.132
BUFG	4	32	12.500
MMCM	2	5	40.000

Notes: LUTRAM = LUT random access memory; BUFG = global buffer; FF = flip-flop; MMCM = mixed-mode clock manager

4. FGPA Tapped Delay Line Implementation

4.1 Design

So far, a simple 100-MHz counter has been used to time the interval between two signals. This type of synchronous design is the standard digital logic design recommended for FPGA coding, simulation, and implementation tools. Asynchronous designs that rely on wire and logic delays frequently cause problems with these design tools and often with the FPGA designs themselves.[14] Nevertheless, asynchronous structures can be used to obtain subcycle time-interval

measurements. One asynchronous method is the tapped delay line shown in Fig. 7. In this implementation, the PPS input goes through a series of delays, τ, which are tapped as clock inputs into a series of flip-flops. The system clock is connected to the data inputs, allowing the clock to be sampled at the rising edge of PPS at a rate of $1/\tau$. A comparison of the clock phase of the various PPS inputs can then be used to determine a fine subcycle time-interval measurement. Normal counters can still be used to create a coarse time measurement.

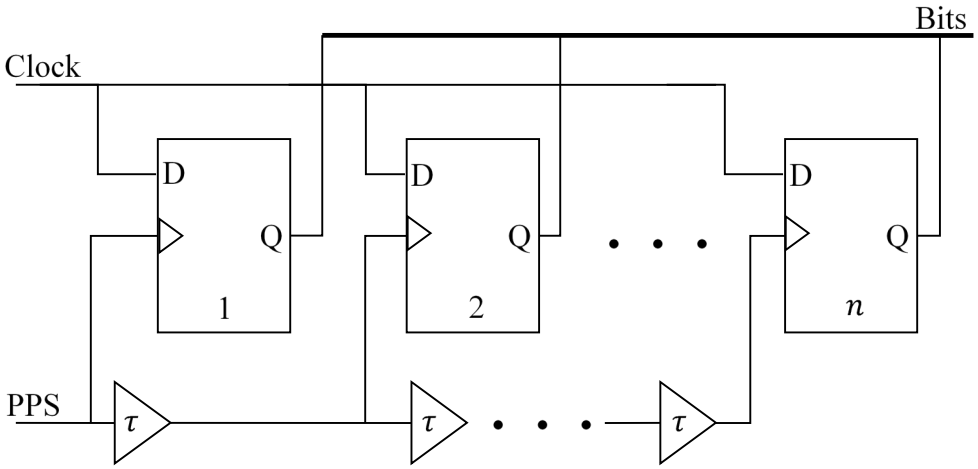

Fig. 7 Tapped delay line

The VHDL code for the tapped delay line is listed under "delay_line.vhd" in Appendix A. The delay elements are implemented as LUTs. In the Xilinx library, LUT operation can be defined with the INIT generic input. Table 9 shows an example LUT of an "and" gate. The LUT lists all of the input combinations as binary numbers counting from 0 to $n^2 - 1$, where n is the number of inputs.[15] The output values define the INIT input, with the input value corresponding to 0 the least significant bit (LSb) of INIT, and the value corresponding to $n^2 - 1$ the MSb of INIT. Hence, the non-inverting buffer used as a delay element is defined with `INIT = '10'`.

Table 9 LUT example

Input 2	Input 1	Output
0	0	0
0	1	0
1	0	0
1	1	1

The signals in the tapped delay line are logically equivalent, and therefore would normally be optimized out of the design during synthesis. This is true of the flip-flop outputs as well. In order to prevent this, the `dont_touch` command is used to instruct synthesis to retain this logic. As mentioned previously, the FPGA design tools are not naturally designed to accommodate asynchronous methods such as "gating" the clocks, which results in error warning messages such as the following:

```
WARNING: [DRC 23-20] Rule violation (PDRC-153) Gated clock check - Net
pps_cnt1/pps_counters_int[10].delay_line1/delay_clk[37] is a gated
clock net sourced by a combinational pin
pps_cnt1/pps_counters_int[10].delay_line1/bits_int2[37].u1/O, cell
pps_cnt1/pps_counters_int[10].delay_line1/bits_int2[37].u1. This is not
good design practice and will likely impact performance.
```

Although gated clocks usually do not represent good design practice, in this case it is desirable and these warning messages can be ignored.

Appendix A lists "pps_cnt.vhd", which was updated to accommodate the delay line. The old PPS input was synchronized to the main clock in order to avoid flip-flop metastability issues. This is clearly unusable for the delay line, necessitating a new PPS input, `pps0`, that is free of any synchronization logic. In addition, the message creation state machine was adjusted to include the bits of the delay line. The new state machine is shown in Fig. 8. The number of taps was set to 48, requiring an extra 6 bytes for each PPS input. The `BIT0` state was added to include these 6 bytes before each 32-bit time-interval count. A new `cnt_bits` variable serves as a counter for these 6 bytes, while the previous `cnt` variable still acts as a counter to index the PPS input.

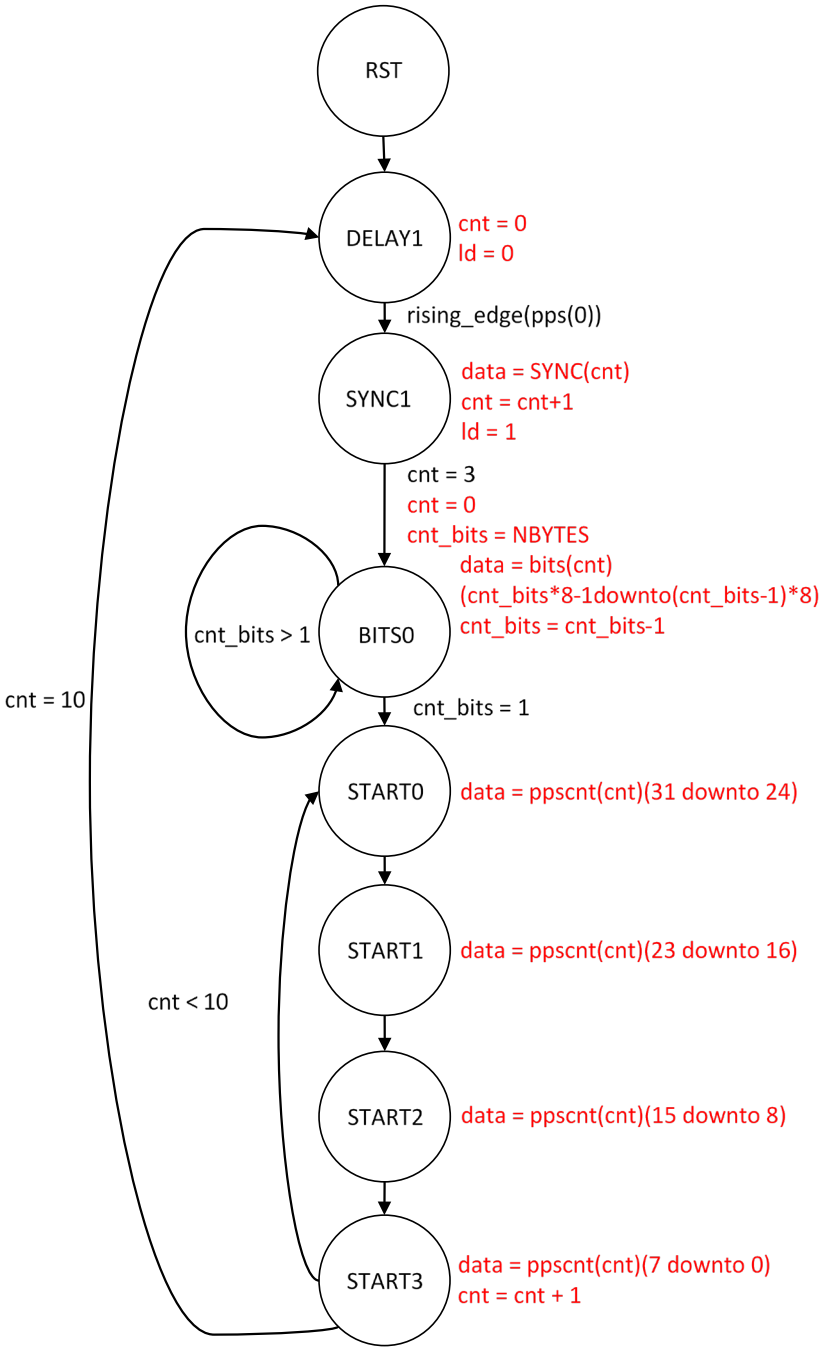

Fig. 8 The pps_cnt state machine with the tapped delay line

Due to the asynchronous nature of this design, simulation was not attempted to verify the design, but the timing report of the implemented design was analyzed. Table 10 shows the FPGA resource utilization. The tapped delay lines increased the LUT usage by about 400%, while the flip-flop use increased about 150%. Tables 11 and 13 show fast and slow timing, respectively, of 8 out of the 48 delays of an example delay line. The fast values have an average delay of 0.47 ns for the LUT

and wire together. The slow values have an average delay of 0.55 ns. There is a significant amount of variability in these delay values. In Table 12 the minimum wire delay is 0.165 ns for PPS0, while the maximum is 0.753 ns for PPS3. This variability can be understood by examining the FPGA routing. Figure 9 shows the 0.753-ns delay wire. The wire is connecting elements in three different FPGA slices, creating a slow, high-capacitance path. Figure 10 shows that the 0.165-ns delay wire is completely contained within a single slice, leading to a small path delay. This variability can probably be improved by using manual routing and placement during the FPGA implementation, but this tedious task was not attempted for this proof-of-concept implementation.

Table 10 Tapped delay line FPGA resource utilization

Resource	Utilization	Available	Utilization (%)
LUT	1218	20800	5.855769
FF	1443	41600	3.46875
BRAM	0.5	50	1
IO	33	106	31.132074
BUFG	4	32	12.5
MMCM	2	5	40

Notes: LUTRAM = LUT random access memory; BUFG = global buffer; FF = flip-flop; MMCM = mixed-mode clock manager

Table 11 Example fast timing path of the tapped delay line

Delay type	Delay (ns)	Location	Netlist resources
LUT1	0.100	SLICE_X13Y111	u0/O
net	0.140	…	delay_clk[0]
LUT1	0.100	SLICE_X13Y111	bits_int2[1].u1/O
net	0.229	…	delay_clk[1]
LUT1	0.100	SLICE_X13Y111	bits_int2[2].u1/O
net	0.550	…	delay_clk[2]
LUT1	0.100	SLICE_X12Y111	bits_int2[3].u1/O
net	0.624	…	delay_clk[3]
LUT1	0.100	SLICE_X12Y109	bits_int2[4].u1/O
net	0.249	…	delay_clk[4]
LUT1	0.100	SLICE_X12Y109	bits_int2[5].u1/O
net	0.147	…	delay_clk[5]
LUT1	0.100	SLICE_X12Y109	bits_int2[6].u1/O
net	0.615	…	delay_clk[6]
LUT1	0.100	SLICE_X12Y112	bits_int2[7].u1/O
net	0.416	…	delay_clk[7]

Table 12 Example slow timing path of the tapped delay line

Delay type	Delay (ns)	Location	Netlist resources
LUT1	0.124	SLICE_X13Y111	u0/O
net	0.165	…	delay_clk[0]
LUT1	0.124	SLICE_X13Y111	bits_int2[1].u1/O
net	0.276	…	delay_clk[1]
LUT1	0.124	SLICE_X13Y111	bits_int2[2].u1/O
net	0.673	…	delay_clk[2]
LUT1	0.124	SLICE_X12Y111	bits_int2[3].u1/O
net	0.753	…	delay_clk[3]
LUT1	0.124	SLICE_X12Y109	bits_int2[4].u1/O
net	0.299	…	delay_clk[4]
LUT1	0.124	SLICE_X12Y109	bits_int2[5].u1/O
net	0.173	…	delay_clk[5]
LUT1	0.124	SLICE_X12Y109	bits_int2[6].u1/O
net	0.744	…	delay_clk[6]
LUT1	0.124	SLICE_X12Y112	bits_int2[7].u1/O
net	0.506	…	delay_clk[7]

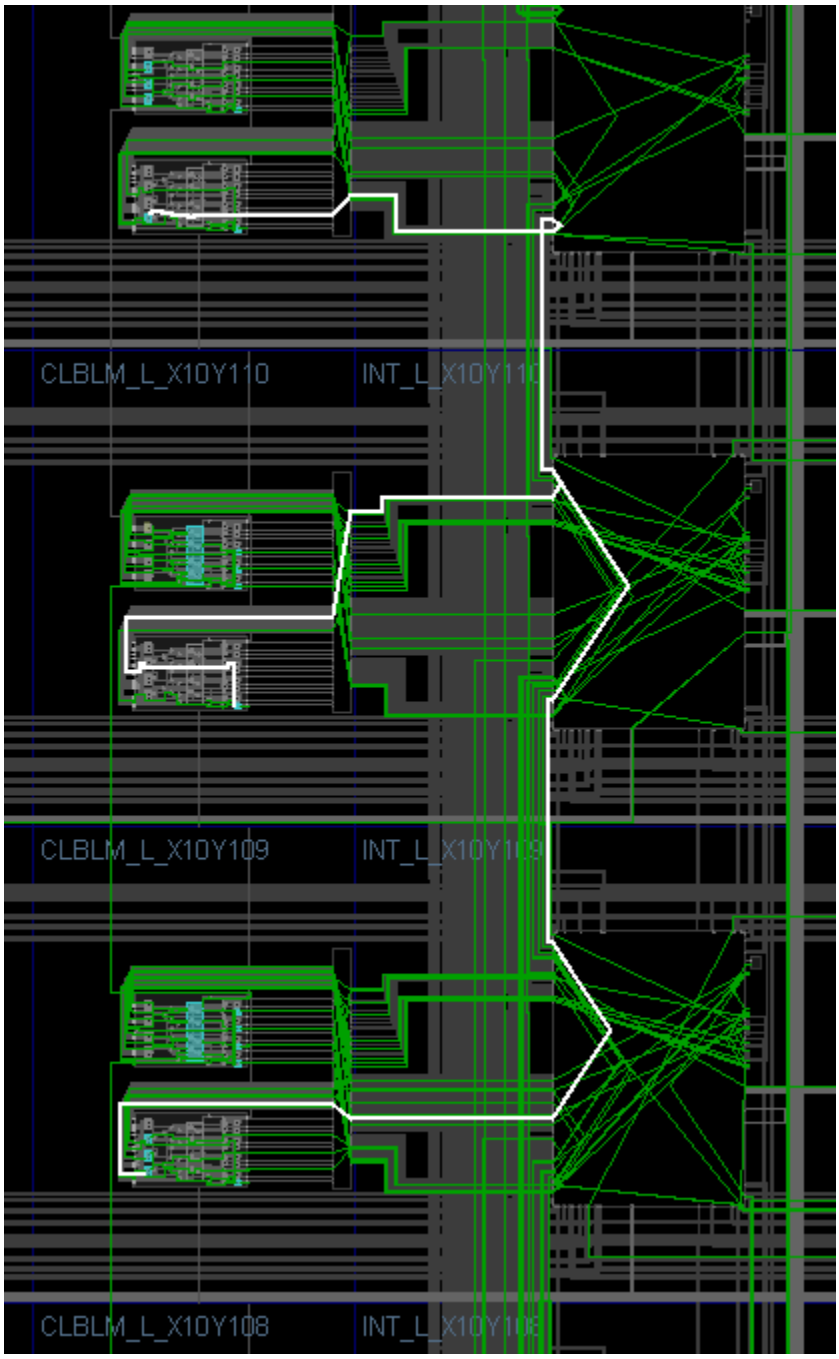

Fig. 9 The wire with the maximum delay from Table 12

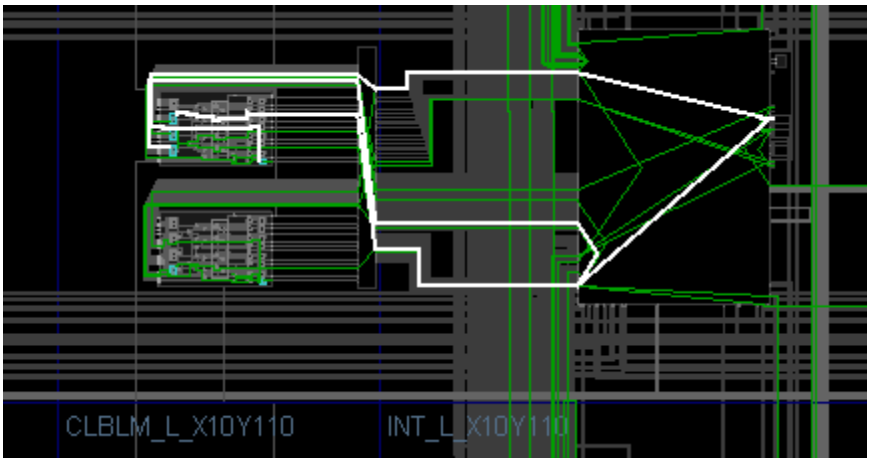

Fig. 10 The wire with the minimal delay from Table 12

4.2 Physical Verification

For physical verification, the PPS0 input was connected to the Test0 output, and the PPS4 input was connected to Test4, with an additional 18-inch RG 58 C/U BNC cable added to the PPS4 path. The same LabVIEW program was used to record the messages, but a new MATLAB script, "read_data_delay_line2.m" in Appendix C, was used to decode the new message structure. Figure 11 shows an initial attempt to compare the delay bits of PPS0 and PPS4 for 100 messages. The clock high is represented by yellow, while low is shown as blue. Here PPS4, the signal with the additional BCN cable, displayed significant noise. In order to decrease the noise, the slew rate and drive strength of the test outputs were increased. The updated constraint file "Cmod_A7_Master.xcd" in Appendix A includes these changes. Figure 12 shows the results of this change, significantly less noise in the delay bits. Using these results, PPS0 displayed a clock cycle that started on average at bit 15 and ended at bit 44.78, for a total of 29.78 bits. For a 100-MHz clock, this gives on average about 0.336 ns per bit. Similarly, PPS4 displays a clock cycle from on average 7.51 bits to 37.04 bits, giving an average of 29.53 bits per cycle, or 0.339 ns per bit, fairly close to the results for PPS0. Using the clearest transitions, there is a rising edge at an average of 37.04 bits for PPS4, corresponding to the rising edge of PPS0 at 30.02 bits, giving an average interval of 7.02 bits between PPS0 and PPS4. Taking the combined average time per bit for PPS0 and PPS4 as 0.337 ns, this interval corresponds to 2.367 ns. The speed of electricity in RG 58 C/U is about 1.5417 ns/ft,[16] or 2.313 ns for 18 inches. This gives an error of 54 ps between the calculated time interval and the measured time interval, a very small difference. A 20-GHz clock would be needed to provide similar accuracy using the counter method, which is impractical using standard technologies. Although a single

measurement is not enough data to fully characterize the performance, it is enough to indicate that this proof-of-concept design has been successful implemented.

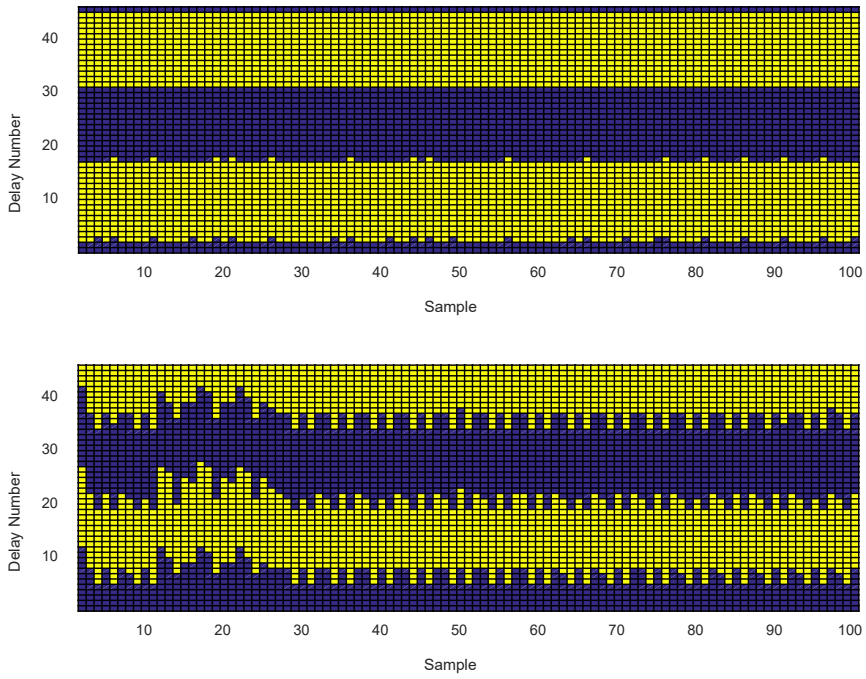

Fig. 11 Delay line results for PPS0 (top) and PPS4 (bottom) using default drive strength and slew rate output buffer settings

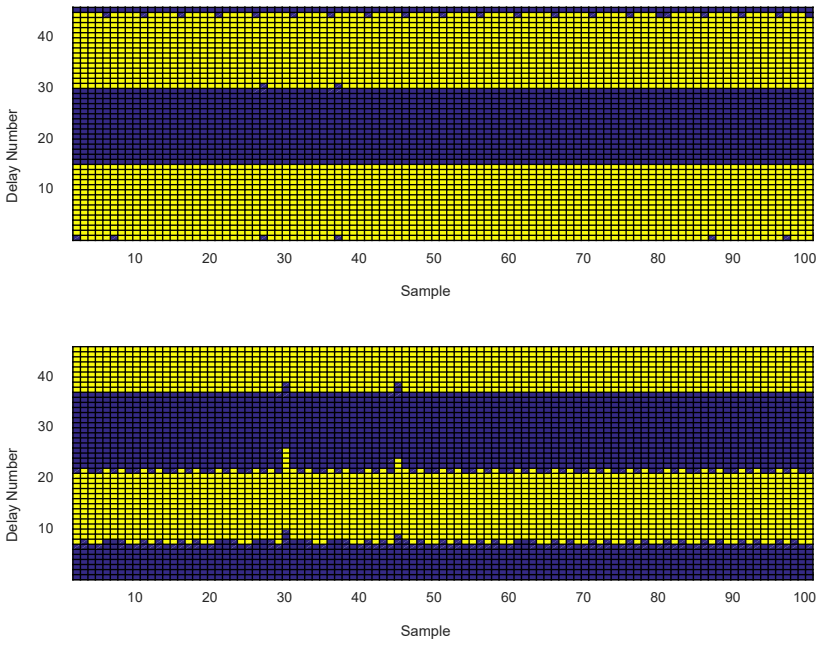

Fig. 12 Delay line results for PPS0 (top) and PPS4 (bottom) using higher drive strength and faster slew rate output buffer settings

5. Conclusion

This report described the development of a multichannel time-interval measurement unit using an FPGA. The design goal of 10-ns resolution on 10 channels was easily achieved using 100-MHz counters. A proof-of-concept tapped delay line method was implemented demonstrating sub-cycle resolution. Possible paths of future research include the following:

- Designing custom counter architectures to increase clock speed.
- Manual FGPA placement and routing to create more uniform delay lines.
- Implementing differential tapped delay lines to increase timing resolution.
- Using higher performance FPGAs to increase clock speed.
- Utilize a digital delay generator for precise physical verification.

6. References

1. Porat DI. Review of sub-nanosecond time-interval measurements. IEEE Transactions on Nuclear Science. 1973;20(5):36–51.

2. Kalisz J. Review of methods for time interval measurements with picosecond resolution. Metrologia. 2003;41(1):17.

3. Benchoff B. Counting really, really fast with an FPGA. 2014 June 29 [accessed 2018 Sep 4]. https://hackaday.com/2014/06/29/counting-really-really-fast-with-an-fpga/.

4. Zieliński M, Chaberski D, Kowalski M, Frankowski R, Grzelak S. High-resolution time-interval measuring system implemented in single FPGA device. Measurement. 2004;35(3):311–317.

5. V660 12-channel VME picosecond resolution time-interval measurement module. San Francisco (CA): Highland Technology, Inc.; c2018 [accessed 2018 Sep 4]. https://www.highlandtechnology.com/DSS/V660DS.shtml.

6. Don ML. Advances in telemetry capability as demonstrated on an affordable precision mortar. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2012 June. Report No.: ARL-RP-378.

7. Don ML. A low-cost software-defined telemetry receiver. Proceedings of the 51st International Telemetering Conference. International Foundation for Telemetering; Las Vegas, NV; 2015 Oct 26–29.

8. GFT2002 1pS time interval counter. San Rafael (CA): Berkeley Nucleonics Corporation; c2018 [accessed 2018 Sep 4]. https://www.berkeleynucleonics.com/model-1120.

9. GC2200 series time interval counter card. Irvine (CA): Marvin Test Solutions; nd [accessed 2018 Sep 4]. https://www.marvintest.com/PXI/Products/PCI-Timer-Counter-Card-GC2200-Series.aspx.

10. GuideTech. Santa Clara (CA): GuideTech; c2017 [accessed 2018 Sep 4]. http://www.guidetech.com/.

11. DLP-HS-FPGA3 USB-FPGA module. Allen (TX): DLP Design; 2012 Apr [accessed 2018 Sep 4]. www.dlpdesign.com/fpga/dlp-hs-fpga3-v11.pdf.

12. GH VHDL library: overview. [place unknown]: OpenCores; c1999–2018 [accessed 2018 Sep 4]. https://opencores.org/project/gh_vhdl_library.

13. Cmod A7 reference manual. Pullman (WA): Digilent; [updated 2016 June 24; accessed 2018 Dec 12]. https://reference.digilentinc.com/_media/cmod_a7/cmod_a7_rm.pdf.

14. Chu PP. RTL hardware design using VHDL: coding for efficiency, portability, and scalability. Hoboken (NJ): John Wiley & Sons; 2006.

15. Understanding the INIT attribute for LUTs. [place unknown]: Mark Harvey; 2015 Oct 11 [accessed 2018 Sep 4]. http://www.markharvey.info/art/init_11.10.2015/init_11.10.2015.html.

16. Coiner P. Calculating the propagation delay of coaxial cable. [place unknown]: GPS Source; 2011 Jan 25 [accessed 2018 Sep 4]. https://cdn.shopify.com/s/files/1/0986/4308/files/Cable-Delay-FAQ.pdf

Appendix A. VHDL Code

This appendix appears in its original form, without editorial change.

dlp_hs_fpga_test.vhd:

```
--IO setup
user_io_dir(20 downto 0)<=(others=>'1'); --inputs
user_io_dir(64 downto 21)<=(others=>'0'); --outputs
pps0<=user_io_in(15 downto 5);

--Output test signals         Output Pin  IO         Input Pin
user_io_out(25)<=not(heart_cnt(BASE_CNT+3)); --33    5       7
user_io_out(26)<=not(heart_cnt(BASE_CNT+3)); --34    6       8
user_io_out(27)<=heart_cnt(BASE_CNT);      --35      7       9
user_io_out(28)<=heart_cnt(BASE_CNT);      --36      8       10
user_io_out(29)<=heart_cnt(BASE_CNT);      --37      9       12
user_io_out(30)<=heart_cnt(BASE_CNT);      --38      10      13
user_io_out(31)<=heart_cnt(BASE_CNT+5);    --39      11      14
user_io_out(32)<=heart_cnt(BASE_CNT+3);    --41      12      15
user_io_out(33)<=heart_cnt(BASE_CNT+2);    --42      13      16
user_io_out(34)<=heart_cnt(BASE_CNT+1);    --43      14      17
user_io_out(35)<=heart_cnt(BASE_CNT);  --44  15      18

--heartbeat LED, drives test signals
process(clks,fpga_reset_out)
   begin
      if rising_edge(clks) then
         if (fpga_reset_out='1') then
            heart_cnt<=(others=>'0');
         else
            heart_cnt <= heart_cnt + 1;
         end if;
      end if;
end process;
ledr_heartbeat<=heart_cnt(BASE_CNT+2); -- using bit x gives a period of 2^(x+1)

--Count intervals and load message into FIFO
pps_cnt1 : pps_cnt generic map (npps=>NPPS) port map (clk=>clks,
   reset=>fpga_reset_out,pps=>pps, ld=>parallel_clk, data=>pps_cnt_byte);

--Message buffer
fifo_1 : gh_fifo_sync_sr2 generic map (add_width=>6, data_width=>8)
   port map (clk=>clks,rst=>fpga_reset_out,srst=>'0',WR=>parallel_clk,
   RD=>rd,D=>pps_cnt_byte, Q=>fifo_out,empty=>empty, full=>full);

--latch pps
process(clks,fpga_reset_out)
   begin
      if (fpga_reset_out='1') then
         pps1<=(others=>'0');
         pps<=(others=>'0');
      elsif rising_edge(clks) then
         pps1<=pps0;
         pps<=pps1;
      end if;
end process;


--read from a fifo and write to USB UART
process(clks,fpga_reset_out)
   variable cnt : integer range 0 to 31;  --delay counter
   begin
      if rising_edge(clks) then
         if (fpga_reset_out='1') then
            state <= RST_S;
         else
            case state is

            when RST_S => --reset everything
               cnt:=0;
```

```
                rd<='0';
                ftdi_wr<='0';
                ftdi_rd<='0';
                state<=IDLE_S;
                ftdi_dout<=x"00";
                ftdi_dir<='0'; -- 0=output
            when IDLE_S =>
                ftdi_wr<='0';
                if (txe_n_s = '0' and empty = '0') then
                -- data ready to write to usb
                    rd<='1'; --read from fifo
                    state<=FIFO_RD_S;
                    cnt:=3; --set delay, needed when clk is fast (clk > 25 MHz)
                end if;

            when FIFO_RD_S =>
                ftdi_wr<='0';
                rd<='0';
                ftdi_dout<=fifo_out(7 downto 0); --output data to uart

                if cnt=0 then --use to slow down signals
                    state<=WR_USB_S;
                    cnt:=3;
                else
                    cnt:=cnt-1;
                end if;

            when WR_USB_S =>
                if cnt=0 then --use to slow down signals
                    if (txe_n_s = '0' and empty = '0') then -- ready to write to usb
                        rd<='1';
                        state<=FIFO_RD_S;
                        cnt:=3;
                    else
                        state<=IDLE_S;
                    end if;
                else
                    cnt:=cnt-1;
                    ftdi_wr<='1';  --write data
                end if;
            when others => state <= RST_S;
        end case;
      end if;
   end if;
   count<=std_logic_vector(conv_unsigned(cnt,8)); --for debug
end process;
```

MIDASlibrary.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package MIDASlibrary is

   constant BASE_CNT: integer:= 23; --for test signals, 23 good, 6 good for sim

   constant NPPS: integer:= 11; --number of pps inputs, including reference

   --SYNC bytes for UART message
   type sync_rom is array (0 to 3) of std_logic_vector(7 downto 0);constant
      SYNC_ROM_CONST : sync_rom :=(
         x"FE",
         x"6B",
         x"28",
         x"40"
      );
```

```
end MIDASlibrary;
```

pps_cnt.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library MIDASlibrary;
use MIDASlibrary.MIDASlibrary.all;


entity pps_cnt is
   GENERIC (npps: INTEGER :=11);
    port ( clk : in std_logic;
           reset : in std_logic;
           pps : in std_logic_vector(npps-1 downto 0);  --0 is ref
           ld : out std_logic;  -- fifo load
           data : out std_logic_vector(7 downto 0) --fifo data
         );
end pps_cnt;

architecture Behavioral of pps_cnt is

component counter_for_pps is
    port ( clk : in std_logic;
           reset : in std_logic;
           cnt_out : out std_logic_vector(31 downto 0);
           pps :in std_logic; --input pps
           ppsr : in std_logic --reference pps
         );
end component;

type cnt_array_type is array (NPPS-1 downto 0) of STD_LOGIC_VECTOR (31 downto 0);
type sync_array_type is array (3 downto 0) of STD_LOGIC_VECTOR (7 downto 0);
signal ppscnt : cnt_array_type;  --example: ppscnt(3)(7 downto 8)


type states is (
   RST,
   DELAY1,
   START0,
   START1,
   START2,
   START3,
   SYNC1
);

signal state :  states;

signal ppsr2 : std_logic;

begin

--have 11 ppscnt counters
-- each counter_for_pps cnts with rising pps(i) and outputs last cnt with pps(0)
-- with each pps(0) - ouput sync and all ppscnt values to fifo to transmit to UART
--    sync = xFE6B2840, then 32 bit ppscnt(0) MSB first, ...., ppscnt(10)
-- counter_for_pps for pps(0) counts cycles between each pps(i) and pps(0)
-- real number of cycles is between pps(0) and pps(i) is ppscnt +1
-- if ppscnt(i)=ppscnt(0)=maxcnt-1, it means they transition at the same time,
-- maxcnt is clk speed
-- Ex. ppscnt(5) = 3, pps(5) went high 4 cycles after pps(0) during the prev. sec
-- Ex. ppscnt(6) = 0, pps(6) went high 1 cycles after pps(0) during the prev. sec
-- Ex. ppscnt(6) = 99,999,999 = ppscnt(0) means that pps(6) went high the same
cycle as pps(0), where the clk is 100 MHz.
-- ppscnt reports what happened from one cycle after rising edge of last pps, to
rising edge of this pps.
```

```
-- if there is a transition, ppscnt(x)(31) = '1', if not = '0'

pps_counters_int:
for i in 0 to NPPS-1 generate
   begin
      counter_for_pps1 : counter_for_pps port map(clk=>clk, reset=>reset,
      cnt_out=>ppscnt(i), pps=>pps(i), ppsr=>pps(0));
end generate;

process(clk) begin
   if rising_edge(clk) then
      ppsr2<=pps(0);
   end if;
end process;

process(clk,reset)
      variable cnt : integer range 0 to 31;
   begin

      if rising_edge(clk) then
         if (reset='1') then
            state <= RST;

         else
            case state is

            when RST => --initialize everything
                  cnt:=0;
                  state<=DELAY1;
                  data<=(others=>'0');
            when DELAY1 => --wait for pps0
                  cnt:=0;
                  ld<='0';
                  if (pps(0)='1' and ppsr2='0') then
                     state<=SYNC1;
                  end if;

            when SYNC1 => --output sync
                  if (cnt = 3) then
                     data<=SYNC_ROM_CONST(cnt);
                     state<=START0;
                     cnt:=0;
                  else
                     ld<='1';
                     data<=SYNC_ROM_CONST(cnt);
                     cnt:=cnt+1;
                  end if;

            when START0 =>  --output cnt MSB
                  ld<='1';
                  data<=ppscnt(cnt)(31 downto 24);
                  state<=START1;

            when START1 => --next byte
                  data<=ppscnt(cnt)(23 downto 16);
                  state<=START2;

            when START2 => --next byte
                  data<=ppscnt(cnt)(15 downto 8);
                  state<=START3;

            when START3 => --LSB
                  data<=ppscnt(cnt)(7 downto 0);
                  if cnt=NPPS-1 then
                     state<=DELAY1;
                  else
                     cnt:=cnt+1;
                     state<=START0;
                  end if;

            when others => state <= RST;
```

```
        end case;
      end if;
    end if;

  end process;

end Behavioral;
```

counter_for_pps.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library MIDASlibrary;
use MIDASlibrary.MIDASlibrary.all;

entity counter_for_pps is
    port (  clk : in std_logic;
          reset : in std_logic;
          cnt_out : out std_logic_vector(31 downto 0);
          pps :in std_logic;   --pps input
          ppsr : in std_logic --reference pps
          );
end counter_for_pps;

architecture Behavioral of counter_for_pps is

signal cnt,cnt_save : std_logic_vector(31 downto 0);
signal pps2,ppsr2,got_pps : std_logic;

type states is (
    RST,
    WAIT_FOR_PPS,
   START
);

signal state :  states;

begin

process(clk) begin
   if rising_edge(clk) then
      pps2<=pps;
      ppsr2<=ppsr;
   end if;
end process;

cnt_out<=cnt_save;

process(clk,reset)
   begin
      if rising_edge(clk) then
         if (reset='1') then
            state<=RST;
         else
            case state is
            when RST => --intitialize everything
               cnt<=(others=>'0');
               cnt_save<=(others=>'0');
               state<=WAIT_FOR_PPS;
            when WAIT_FOR_PPS =>
                  if (ppsr2='0' and ppsr='1') then --if transition of ppsr
                     cnt_save(30 downto 0)<=cnt(30 downto 0);       --save cnt
                     --if transition of pps
                     if ((pps2='0' and pps='1') or got_pps='1') then
                                 cnt_save(31)<='1'; --assert MSB if transitioned
                     else
```

```
                        cnt_save(31)<='0';

                     end if;
                     cnt<=(others=>'0');
                     got_pps<='0';
                  else

                     --if transition of pps, stop counting
                     if ((pps2='0' and pps='1') or got_pps='1') then
                                got_pps<='1';
                     else        --if no transition, count
                        cnt<=cnt+1;
                     end if;
                  end if;
               when others => state <= RST;
            end case;
         end if;
      end if;
   end process;
end Behavioral;
```

delay_line.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity delay_line is
   GENERIC (ntaps: INTEGER :=16);
    port (  clk : in std_logic;
            pps : in std_logic;
            bits : out std_logic_vector(ntaps-1 downto 0)
         );
end delay_line;

architecture Behavioral of delay_line is

signal reset : STD_LOGIC_VECTOR (ntaps-1 downto 0);
signal delay_clk : STD_LOGIC_VECTOR (ntaps-1 downto 0);
signal bitsi : STD_LOGIC_VECTOR (ntaps-1 downto 0);
signal bits_save : STD_LOGIC_VECTOR (ntaps-1 downto 0);
signal clk_en : STD_LOGIC_VECTOR (ntaps-1 downto 0);
signal pps_reset,pps_delay : STD_LOGIC;

attribute dont_touch : string;
attribute dont_touch of delay_clk : signal is "true";
attribute dont_touch of bitsi : signal is "true";

begin

bits<=bitsi;

bits_int:
for i in 1 to ntaps-1 generate
   begin
        FDCE_inst : FDCE generic map (INIT=>'0') port map (Q=>bitsi(i),
C=>delay_clk(i), CE=>'1', CLR=>'0', D=>clk);
end generate;
```

```
u0 : LUT1 generic map (INIT=>"10") port map (O=>delay_clk(0),I0=>pps);
bits_int2:
for i in 1 to ntaps-1 generate
   begin
        u1 : LUT1 generic map (INIT=>"10") port map
(O=>delay_clk(i),I0=>delay_clk(i-1));
end generate;

end Behavioral;
```

pps_cnt.vhd (tapped delay line version):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pps_cnt is
   GENERIC (npps: INTEGER :=11;   --number of pps inputs
            ntaps: INTEGER :=16);  --number of taps
    port ( clk : in std_logic;
         reset : in std_logic;
           pps : in std_logic_vector(npps-1 downto 0);--latched input for coutners
           ld : out std_logic;   --fifo load
         data : out std_logic_vector(7 downto 0); --fifo data
         pps0 : in std_logic_vector(npps-1 downto 0)  --non-latched pps inputs for
delay line
         );
end pps_cnt;

architecture Behavioral of pps_cnt is

component counter_for_pps is
    port ( clk : in std_logic;
           reset : in std_logic;
         cnt_out : out std_logic_vector(31 downto 0);
         pps :in std_logic;
         ppsr : in std_logic
         );
end component;

component delay_line is
   GENERIC (ntaps: INTEGER :=16);
    port (  clk : in std_logic;
            pps : in std_logic;
            bits : out std_logic_vector(ntaps-1 downto 0)
         );
end component;

type cnt_array_type is array (NPPS-1 downto 0) of STD_LOGIC_VECTOR (31 downto 0);
type sync_array_type is array (3 downto 0) of STD_LOGIC_VECTOR (7 downto 0);
signal ppscnt : cnt_array_type;  --example: ppscnt(3)(7 downto 8)

type taps_array_type is array (NPPS-1 downto 0) of STD_LOGIC_VECTOR (ntaps-1
downto 0);
signal bits : taps_array_type;

type states is (
    RST,
    DELAY1,
   START0,
   START1,
```

```
   START2,
   START3,SYNC1,BITS0
);

signal state :  states;

signal ppsr2 : std_logic;

   constant NBYTES: integer:= ntaps/8; --number of tap bytes

   type sync_rom is array (0 to 3) of std_logic_vector(7 downto 0); --SYNC bytes
for UART message
   constant SYNC_ROM_CONST : sync_rom :=(
   x"FE",
   x"6B",
   x"28",
   x"40"
   );

       signal test1 : std_logic_vector(15 downto 0) := x"AF31";

       attribute dont_touch : string;
        attribute dont_touch of bits : signal is "true";
        attribute dont_touch of pps0 : signal is "true";
        attribute dont_touch of clk : signal is "true";

begin

--have 11 ppscnt counters
-- each counter_for_pps cnts with rising pps(i) and outputs last cnt with pps(0)
-- with each pps(0) - ouput sync and all ppscnt values to fifo to transmit to UART
--    sync = xFE6B2840, then 32 bit ppscnt(0) MSB first, ...., ppscnt(10)
-- counter for pps for pps(0) counts cycles between each pps(i) and pps(0)
-- real number of cycles is between pps(0) and pps(i) is ppscnt +1
-- if ppscnt(i)=ppscnt(0)=maxcnt-1, it means they transition at the same time,
maxcnt is clk speed
-- Ex. ppscnt(5) = 3 means that pps(5) went high 4 cycles after pps(0) during the
previous second
-- Ex. ppscnt(6) = 0 means that pps(5) went high 1 cycles after pps(0) during the
previous second
-- Ex. ppscnt(6) = 99,999,999 = ppscnt(0) means that pps(6) went high the same
cycle as pps(0), where the clk is 100 MHz.
                  --this occured this second (this last pps going high)
-- ppscnt reports what happened from one cycle after rising edge of last pps, to
rising edge of this pps.
-- if there is a transition, ppscnt(x)(31) = '1', if not = '0'

pps_counters_int:
for i in 0 to NPPS-1 generate
   begin
      counter_for_pps1 : counter_for_pps port map(clk=>clk, reset=>reset,
cnt_out=>ppscnt(i), pps=>pps(i), ppsr=>pps(0));
       delay_line1: delay_line generic map (ntaps=>NTAPS) port map (clk=>clk,
pps=>pps0(i), bits=>bits(i));
end generate;


process(clk) begin
   if rising_edge(clk) then
      ppsr2<=pps(0);
   end if;
end process;

process(clk,reset)
      variable cnt : integer range 0 to 31;
      variable cnt_bits : integer range 0 to 31;
   begin

      if rising_edge(clk) then
         if (reset='1') then
            state <= RST;
```

```
        else
           case state is

           when RST => --initialize everything
                 cnt:=0;
                  cnt_bits:=0;
                 state<=DELAY1;
                 data<=(others=>'0');
           when DELAY1 =>  --start message when pps(0) asserts
                 cnt:=0;
                 ld<='0';
                 if (pps(0)='1' and ppsr2='0') then
                    state<=SYNC1;
                 end if;

           when SYNC1 => --send sync
                 if (cnt = 3) then
                    data<=SYNC_ROM_CONST(cnt);
                    state<=BITS0;
                    cnt_bits:=NBYTES;
                    cnt:=0;
                 else
                    ld<='1';
                    data<=SYNC_ROM_CONST(cnt);
                    cnt:=cnt+1;
                 end if;
           when BITS0 => --send tap bits
                          data<=bits(cnt)(cnt_bits*8-1 downto (cnt_bits-1)*8);
                          ld<='1';
                          if (cnt_bits = 1) then
                            state<=START0;
                              cnt_bits:=NBYTES;
                          else
                              cnt_bits:=cnt_bits-1;
                          end if;

           when START0 =>  --send counter values
                 ld<='1';
                 data<=ppscnt(cnt)(31 downto 24);
                 state<=START1;

           when START1 =>
                 data<=ppscnt(cnt)(23 downto 16);
                 state<=START2;

           when START2 =>
                 data<=ppscnt(cnt)(15 downto 8);
                 state<=START3;

           when START3 =>
                 data<=ppscnt(cnt)(7 downto 0);
                 if cnt=NPPS-1 then
                    state<=DELAY1;
                 else
                    cnt:=cnt+1;
                    state<=BITS0;
                 end if;

           when others => state <= RST;
        end case;
      end if;
    end if;

  end process;

end Behavioral;
```

Cmod_A7_Master.xcd:

```
# Clock signal 12 MHz
set_property -dict { PACKAGE_PIN L17   IOSTANDARD LVCMOS33 } [get_ports { CLK }];
#IO_L12P_T1_MRCC_14 Sch=gclk
create_clock -add -name sys_clk_pin -period 83.33 -waveform {0 41.66} [get_ports
{CLK}];

set_property -dict { PACKAGE_PIN M3    IOSTANDARD LVCMOS33 } [get_ports { pioi[0]
}]; #IO_L8N_T1_AD14N_35 Sch=pio[01]
set_property -dict { PACKAGE_PIN L3    IOSTANDARD LVCMOS33 } [get_ports { pioi[1]
}]; #IO_L8P_T1_AD14P_35 Sch=pio[02]
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports { pioi[2]
}]; #IO_L12P_T1_MRCC_16 Sch=pio[03]
set_property -dict { PACKAGE_PIN K3    IOSTANDARD LVCMOS33 } [get_ports { pioi[3]
}]; #IO_L7N_T1_AD6N_35 Sch=pio[04]
set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports { pioi[4]
}]; #IO_L11P_T1_SRCC_16 Sch=pio[05]
set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 } [get_ports { pioi[5]
}]; #IO_L3P_T0_DQS_AD5P_35 Sch=pio[06]
set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports { pioi[6]
}]; #IO_L6N_T0_VREF_16 Sch=pio[07]
set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 } [get_ports { pioi[7]
}]; #IO_L11N_T1_SRCC_16 Sch=pio[08]
set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports { pioi[8]
}]; #IO_L6P_T0_16 Sch=pio[09]
set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { pioi[9]
}]; #IO_L7P_T1_AD6P_35 Sch=pio[10]
set_property -dict { PACKAGE_PIN J1    IOSTANDARD LVCMOS33 } [get_ports { pioi[10]
}]; #IO_L3N_T0_DQS_AD5N_35 Sch=pio[11]
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports { pioi[11]
}]; #IO_L12P_T1_MRCC_35 Sch=pio[18]

set_property -dict { PACKAGE_PIN V4    IOSTANDARD LVCMOS33 } [get_ports { pioo[11]
}]; #IO_L11N_T1_SRCC_34 Sch=pio[37]
set_property -dict { PACKAGE_PIN U4    IOSTANDARD LVCMOS33 } [get_ports { pioo[10]
#IO_L11P_T1_SRCC_34 Sch=pio[38]
set_property -dict { PACKAGE_PIN V5    IOSTANDARD LVCMOS33 } [get_ports { pioo[9]
}]; #IO_L16N_T2_34 Sch=pio[39]
set_property -dict { PACKAGE_PIN W4    IOSTANDARD LVCMOS33 } [get_ports { pioo[8]
}]; #IO_L12N_T1_MRCC_34 Sch=pio[40]
set_property -dict { PACKAGE_PIN U5    IOSTANDARD LVCMOS33 } [get_ports { pioo[7]
}]; #IO_L16P_T2_34 Sch=pio[41]
set_property -dict { PACKAGE_PIN U2    IOSTANDARD LVCMOS33 } [get_ports { pioo[6]
}]; #IO_L9N_T1_DQS_34 Sch=pio[42]
set_property -dict { PACKAGE_PIN W6    IOSTANDARD LVCMOS33 } [get_ports { pioo[5]
}]; #IO_L13N_T2_MRCC_34 Sch=pio[43]
set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports { pioo[4]
}]; #IO_L9P_T1_DQS_34 Sch=pio[44]
set_property -dict { PACKAGE_PIN U7    IOSTANDARD LVCMOS33 } [get_ports { pioo[3]
}]; #IO_L19P_T3_34 Sch=pio[45]
set_property -dict { PACKAGE_PIN W7    IOSTANDARD LVCMOS33 } [get_ports { pioo[2]
}]; #IO_L13P_T2_MRCC_34 Sch=pio[46]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS33 } [get_ports { pioo[1]
}]; #IO_L14P_T2_SRCC_34 Sch=pio[47]
set_property -dict { PACKAGE_PIN V8    IOSTANDARD LVCMOS33 } [get_ports { pioo[0]
}]; #IO_L14N_T2_SRCC_34 Sch=pio[48]

set_property slew "fast" [get_ports {pioo[0]}]
set_property drive "16" [get_ports {pioo[0]}]

set_property slew "fast" [get_ports {pioo[4]}]
set_property drive "16" [get_ports {pioo[4]}]

## UART
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { UART_TXD
}]; #IO_L7N_T1_D10_14 Sch=uart_rxd_out
```

Appendix B. LabVIEW Programs

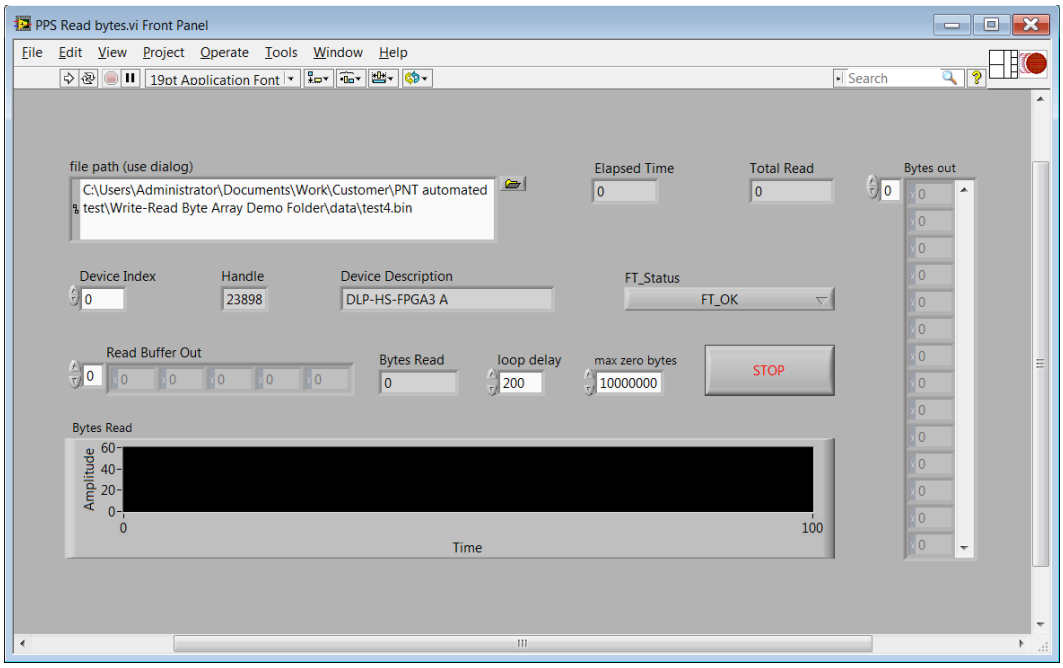

Fig. B-1 The "PPS Read Bytes" LabVIEW graphical user interface used to communicate with DLP-HS-FPGA3 board

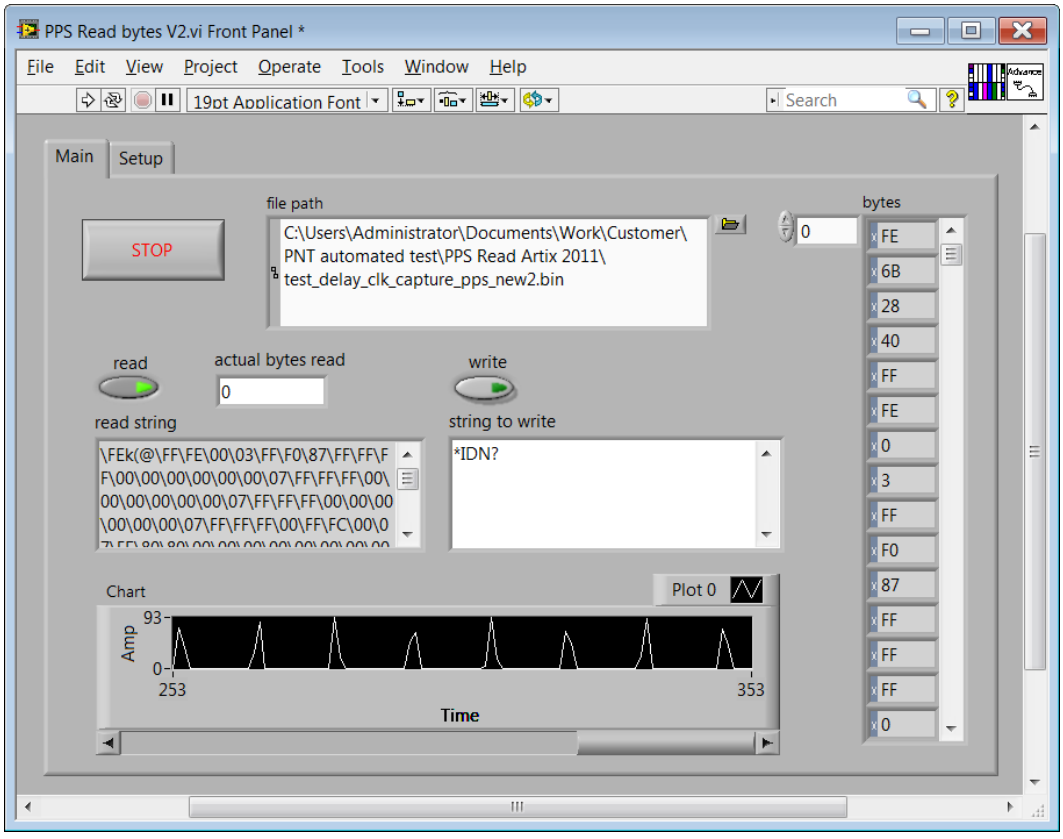

Fig. B-2 The "PPS Read Bytes V2" LabVIEW program used to communicate with the Cmod A7 board

Appendix C. MATLAB Scripts

This appendix appears in its original form, without editorial change.

Read_data.m:

```
clear
close all

%read data
fid=fopen('C:\Users\Administrator\Documents\Work\Customer\PNT automated
test\data\test.bin');
a = fread(fid,'uint8=>uint8');
fclose(fid);

%find sync
sync=uint8([hex2dec('FE') hex2dec('6B') hex2dec('28') hex2dec('40')]);
synci=find(a(1:end-3)==sync(1) & a(2:end-2)==sync(2) & a(3:end-1)==sync(3) &
a(4:end)==sync(4));

%Data Matrix, rows will be pps number, columns seconds
dm=zeros(12,length(synci)-1,'uint32');
for i=1:length(synci)-1
    dm(:,i)=swapbytes(typecast(a(synci(i):synci(i)+47),'uint32'));
end
dm=dm(2:end,:); %get rid of sync

tm=(dm>=2^31); % Transition Matrix: 1=there was a transition, 0=no transition
dm=dm-uint32(tm*2^31); % remove msb
dm2=(double(dm)+1)/100e6; % Data Matrix:  add 1 cycle, divide by clk freq to get
time
dm2(tm==0)=nan; %set no transition to NaN
for i=1:length(synci)-1 %if match PPS0, time is 0
    tmp=dm2(:,i)==dm2(1,i);
    dm2(tmp,i)=0;
end

dlmwrite('data.csv',dm,'delimiter',',','precision',10) %32 bit data
dlmwrite('data_dm2.csv',dm2,'delimiter',',','precision',10) %time
dlmwrite('data_tm.csv',tm,'delimiter',',','precision',10) %transition matrix
```

read_data_delay_line2.m:

```
clear
close all
fid=fopen('C:\Users\Administrator\Documents\Work\Customer\PNT automated test\PPS
Read Artix 2011\test_delay_clk_capture_pps_new2.bin');
a = fread(fid,'uint8=>uint8');
fclose(fid);

%find sync
sync=uint8([hex2dec('FE') hex2dec('6B') hex2dec('28') hex2dec('40')]);
synci=find(a(1:end-3)==sync(1) & a(2:end-2)==sync(2) & a(3:end-1)==sync(3) &
a(4:end)==sync(4));

ntaps=48;
tap_bytes=ntaps/8;

%Data Matrix, rows will be pps number, columns seconds
dm=zeros(11,length(synci)-1,'uint32');
bits=zeros(11,tap_bytes,length(synci)-1,'uint8'); %PPS num rows, tap bytes cols,
message num is page
for i=1:length(synci)-1
    index=synci(i)+4;
    for k=1:11
        bits(k,:,i)=swapbytes(typecast(a(index:index+tap_bytes-1),'uint8'));
```

```
dm(k,i)=swapbytes(typecast(a(index+tap_bytes:index+3+tap_bytes),'uint32'));
        index=index+4+tap_bytes;
    end
end

tm=(dm>=2^31); % Transition Matrix: 1 =  there was a transition, 0 = no transition
dm=double(dm-uint32(tm*2^31)+1); % Data Matrix: clock cycles from pps(0), remove
msb and add 1 cycle to get real cnt
dm(tm==0)=nan;

%cap num messages at 100
if length(synci) >= 100
    max_plot=100;
else
    max_plot=length(synci)-1;
end

%change bytes to bits
sa=zeros(11,ntaps,max_plot); %signal array
for k=1:11
    figure(k)
    for i=1:max_plot
        tmp=[];
        for bi=1:tap_bytes
            tmp=[tmp (dec2bin(bits(k,bi,i),8))];  %MSB are to the left
        end
        tmp=tmp=='1';
        sa(k,:,i)=tmp;
    end
    surf(squeeze(sa(k,:,:)))
    view(2)
    ylim([1 ntaps-1])
    xlim([1 100])
    ylabel('Delay Number')
    xlabel('Sample')
    %box on
end

%look at PPS0 and PPS4
figure(50)
subplot(2,1,1)
    surf(squeeze(sa(1,:,:)))
    view(2)
    ylim([1 ntaps-1])
    xlim([1 100])
    ylabel('Delay Number')
    xlabel('Sample')
    %colorbar
subplot(2,1,2)
    surf(squeeze(sa(5,:,:)))
    view(2)
    ylim([1 ntaps-1])
    xlim([1 100])
    ylabel('Delay Number')
    xlabel('Sample')


%meaure the average transition time past a give threshold
max_trans=6;
trans=zeros(11,max_plot,max_trans);
trans_measure=zeros(11,max_plot);
pps_trans=zeros(1,11);
thresh=35;
for k=1:11
    for i=1:max_plot
        tmp=find(diff(sa(k,:,i))~=0);
        trans(k,i,:)=[tmp nan(1,max_trans-length(tmp))]; %find transitions
        if isempty(find(trans(k,i,:)>thresh,1))
            trans_measure(k,i)=nan;
        else
```

```
            trans_measure(k,i)=trans(k,i,find(trans(k,i,:)>thresh,1));
        end
    end
    pps_trans(k)=mean(trans_measure(k,:));
end
```

List of Symbols, Abbreviations, and Acronyms

ARL	US Army Research Laboratory
ASIC	application specific integrated circuit
BUFG	global buffer
DDR2	double data rate version 2
DIP	dual in-line package
FF	flip-flop
FIFO	first in, first out
FPGA	field programmable gate array
FTDI	Future Technology Devices International
GPS	global positioning system
IC	integrated circuit
I/O	input/output
LSb	least significant bit
LSB	least significant byte
LUT	lookup table
LUTRAM	LUT random access memory
MMCM	mixed-mode clock manager
MSb	most significant bit
MSB	most significant byte
NaN	not a number
PC	personal computer
PCB	printed circuit board
PCI	peripheral component interconnect
Pmod	peripheral module
PPS	pulse per second

PXI	PCI extensions for instrumentation
RX	receive
SDRAM	synchronous dynamic random-access memory
SRAM	static random access memory
TX	transmit
UART	Universal Asynchronous Receiver/Transmitter
USB	universal serial bus
VHDL	VHSIC hardware description language
VHSIC	very-high-speed integrated circuit
VISA	virtual instrument software architecture
VME	versa module Europa

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
25 (PDF)	ARL RDRL WML F B ALLIK B J ACKER T G BROWN S BUGGS E BUKOWSKI J COLLINS J CONDON B DAVIS M DON D EVERSON R HALL J HALLAMEYER M HAMAOUI T HARKINS M ILG B KLINE J MALEY C MILLER B NELSON D PETRICK K PUGH N SCHOMER B TOPPER RDRL SES X M NAIR B PATTON