INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2016, VOL. 62, NO. 3, PP. 237-246
Manuscript received July 20, 2016; revised August, 2016. DOI: 10.1515/eletel-2016-0032

Time interval measurement module implemented in SoC FPGA device

Grzegorz Grzęda and Ryszard Szplet

***Abstract*—We presents the design and test results of a picosecond-precision time interval measurement module, integrated as a System-on-Chip in an FPGA device. Implementing a complete measurement instrument of a high precision in one chip with the processing unit gives an opportunity to cut down the size of the final product and to lower its cost. Such approach challenges the constructor with several design issues, like reduction of voltage noise, propagating through power lines common for the instrument and processing unit, or establishing buses efficient enough to transport mass measurement data. The general concept of the system, design hierarchy, detailed hardware and software solutions are presented in this article. Also, system test results are depicted with comparison to traditional ways of building a measurement instrument.**

***Keywords*—time interval measurement, time-to-digital converter, system on chip, measurement data processing**

I. Introduction

IN general, each measurement system consists of three main parts: the adequate measuring device, processing core and user interface [1]. A time interval measurement system usually involves a time interval counter (TIC) implemented as the measuring device (fig. 1). The TIC measures the time elapsed between two events represented by leading edges of two input signals: START and STOP. The TIC converts measured time intervals into raw measurement data, which are sent to the processor. It is worth mentioning, that the TIC performs data pre-processing, removing conversion errors, before measurement frames are being sent to the processor. Those errors are called bubble errors and they are discussed later in this paper.

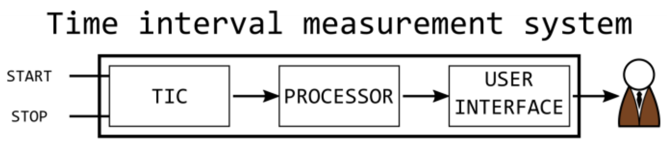


Fig. 1. Common model of a time interval measurement system

An important part of the system is the processor, which is responsible for organizing the work of the entire system. It also performs all the calculations needed for translating the raw measurement data into calibration characteristics and measurement results expressed in seconds. Additionally, it handles the communication between the TIC and the user. The last part of the system is the user interface. In this design it is implemented as a standard command line interface, with user friendly text inputs and outputs. The user is then able to initialize the device, calibrate and conduct measurements. The measurement system was implemented in Zynq SoC (fig. 2) manufactured by Xilinx. The device consists of two subsystems, i.e.: processing system (PS) and programmable logic (PL) [2]. The first one is being used for implementing software, which handles data processing in a classic, program flow manner. The processing subsystem contains an application processing unit (APU), standard I/O devices (UARTs, I2C, SPI, Ethernet etc.), a DDR memory controller and an expandable interconnect matrix, binding the subsystem together in the terms of efficient mass data transfers. In the APU a dual core ARM A9 processor is implemented along with complementary sub-circuitry, that contains: two layer cache memory (32kB L1 memory for each core, 512 kB L2 shared memory), direct memory access (DMA) controller, generic interrupt controller (GIC), snoop control unit (SCU) for data coherency and 256 kB of on-chip memory (OCM) for local read/write operations. The OCM bypasses the L2 cache and is connected directly to both processor, providing fast and up to date storage, properly suited for e.g. inter core communication.

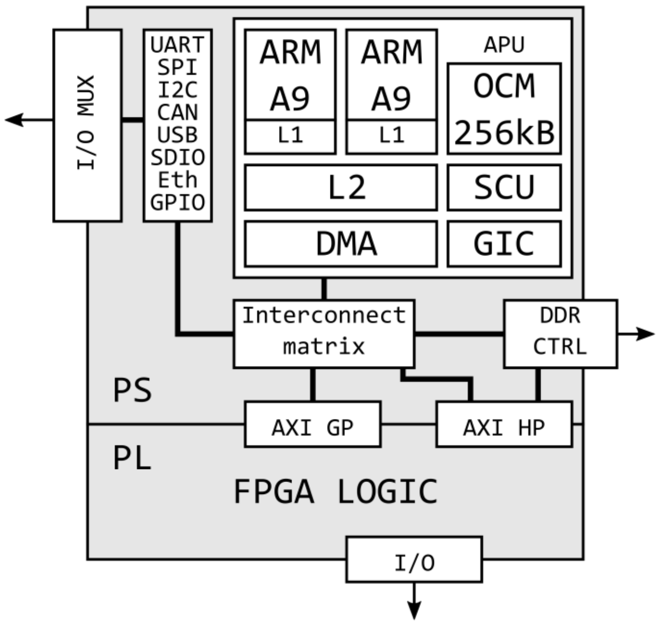


Fig. 2. Zynq SoC block diagram.

The PL subsystem is in fact an FPGA device, which size and capabilities depend on the chosen Zynq model. The whole Zynq family consists of either Artix-7 or Kintex-7 FPGA devices operating as the PL subsystem. Two parts of the SoC

G. Grzęda is with Institute of Telecommunication, Military University of Technology, Warsaw, Poland (e-mail: grzegorz.grzeda@wat.edu.pl)
R. Szplet is with Institute of Telecommunication, Military University of Technology, Warsaw, Poland (e-mail: ryszard.szplet@wat.edu.pl)

communicate with each other by a set of AXI standard-based busses [3]. Those busses can be divided into two main groups: general purpose buses (AXI GP) and high performance buses (AXI HP). There is also a special group of buses, called advanced coherency port (ACP), automatizing cache memory invalidation, yet being a variation of the AXI HP and not considered in this design. The SoC has an extra feature, which may lead to slight improvement in data throughput. The AXI HP has some busses connected directly to the DDR memory controller. This route is discussed in the next section of the paper.

The AXI GP port serves as a maintenance bus, ideally fitted for register-like access. Data transfer rate achieved on this port is not very high, but there are data integrity mechanisms, providing sufficient an error-free communication between the processor and the peripheral. On the contrary, the AXI HP port was designed to maximize data throughput. The data transfer is synchronous streaming. The port is initially configured so that the PL operates as a data transactions master and the DDR controller (or APU) as a slave. This scheme provides reliable and fast data transactions, if timing is crucial for proper operation. In order to achieve high data throughput, no data integrity mechanisms are implemented. Furthermore, the transactions are run independently from the APU unit, which means that the core software needs to invalidate both L1 and L2 caches in order to read newly arrived data from the DDR memory. Of course, this results in data transfer delays and needed to be handled properly, in order to avoid data mismatch while processing measurement results.

II. System design

The designed measurement system (fig. 3) consists of several components and implements some design techniques in order to achieve high performance [4]. While designing the measurement system, several issues were taken into account. One of the main issue was efficient mass data transport throughout the system. A massive amount of data is being sent from the TIC to the DDR memory while conducting time interval measurements. The AXI HP port is tuned for optimal transfer speed between the PL and DDR controller [2].

The AXI HP port requires a sophisticated bus controller on the PL side in order to operate. Instead of designing one from the scratch, it is reasonable to utilize a ready to use IP core, the AXI DMA module, prepared by Xilinx [5]. Such module is configured as a bridge between the TIC interface and the AXI HP interface. Although AXI DMA automatizes many bus operations and eases transactions, it still requires a stream type bus, conditioned by the AXI Stream standard. That is why a custom translation module was designed.

The translation module communicates via the wide asynchronous bus, requesting new measurements from TIC and acquiring data frames. Next it translates those frames according to the AXI Stream protocol specification and streams the data to the AXI DMA module, from where it is transported to the DDR memory. The translation module is configured with the number of measurements to be conducted.

The second major issue was to control the TIC and adjacent components in a unified manner. The control and status registers were placed in a continuous memory space, just like ordinary processor peripherals. This could be achieved by using the AXI GP buses. Commands sent to the peripherals and status words received from them did not need fast transactions and would be sent reliably, without errors. The AXI GP is connected to the translation module. Thanks to such configuration, the module is in fact the master for both the TIC and the AXI DMA. By sending a reset signal followed by the number of measurements to be conducted, the translation module initiates each transaction in the TIC, and forwards the converted frame into the AXI DMA.

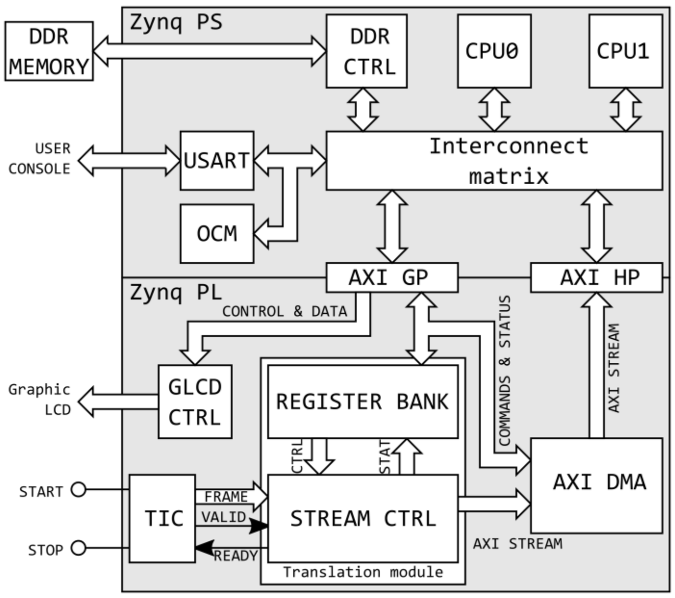


Fig. 3. Measurement system implementation.

The third design issue was to establish an efficient communication channel with the system user. The basic method was to utilize one USART port in the PS. This enabled a simple command prompt using a standard serial terminal device.

The last important issue was to establish an efficient inter-CPU link, so that both cores could communicate between each other and exchange information. As mentioned earlier, the on-chip RAM was used, due to possibility of bypassing the L2 cache and other circuitry, directly connecting both CPU cores with minimal software and hardware overhead.

III. Data flow hierarchy

Each system, where massive data flows are expected, needs to have clarified hierarchical data flow design. Configuring the right connection between masters and slaves is essential in achieving high data throughput in the system.

The Zynq SoC has an organized hierarchy in terms of data flow [6]. Each connection is organized as point to point, master to slave. Furthermore, each module is responsible only for those masters and slave, to which it is directly connected. Fig. 4 shows a generalized data flow structure of the Zynq SoC. Each arrow connecting two modules is directed from a local master to a local slave, e.g. the SCU module is a master for L2 cache, DDR controller and OCM or PL, at the same time being a slave for each of CPU cores.

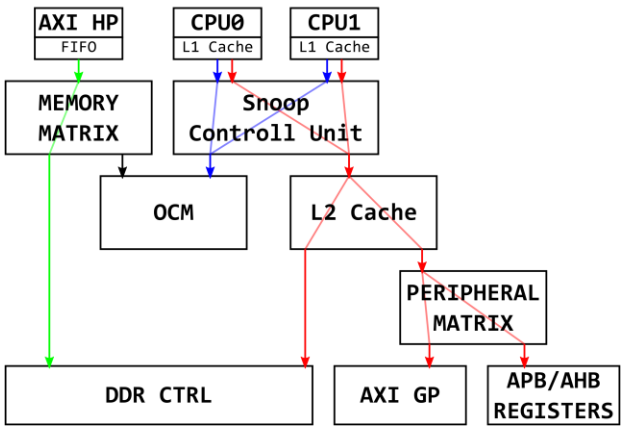


Fig. 4. Data flow in the SoC.

In addition, the SCU is responsible only for transactions it is involved in, e.g. when the CPU0 sends information to the DDR memory, the SCU only guarantees undisrupted transfer to the L2 cache. It is the task of the L2 cache to properly transfer information to the DDR controller. This liability split of each module increases the total throughput and allows to trim the total performance in a more flexible way.

In the presented design three main data paths are taken into account: measurement frames, inter-CPU data exchange and other (program code, register access etc.). Those are marked green, blue and red, respectively in fig. 4. The most critical is the data measurement path. In order to achieve high data throughput resulting in higher measurement rate, the data path needs to be fast and have the minimal amount of hubs and multiplexes as possible. This is because, in every hub or data multiplexer, while transmitting data through one path, other paths enter a wait state. To many wait states lower overall data throughput on a certain path. In some modules e.g. SCU or DDR controller, the programmer may set the priority of certain data paths. The DDR controller may be set to handle data coming from AXI HP first, making the SCU to wait until the streaming transaction from the PL would end [5].

The second most important data path is the blue one. Through this route both CPU cores can communicate with each other. It is very important, that the data don’t have to be passed through the L2 cache memory. The reason for this is memory invalidation. When one CPU writes to the OCM some information, the second one isn’t aware of that event. If the second CPU would check on the OCM memory at least two times, the second and next readout would be not from OCM but from the L2 cache, where the content of the whole OCM would be copied to save time. If the first CPU changes the content of the OCM, the L2 cache controller is unaware of that event, resulting in misinterpretation of data by the second core. The second core would have to refresh the L2 cache each time it wants to read from the OCM, in order to fetch valid data. The same applies for the first core, when it wants to check if the second one responded with new data. Invalidating L2 cache (which is 512 kB) would take some time, where the CPU would do nothing else but iterate throughout every sector of the cache memory, invalidating its content. Zynq SoC has was designed to avoid such scenario. The only cache a CPU has to invalidate is the L1 cache memory, invalidated in step. This is because L1 is smaller than L2 and has separate caching for program and processing data. The data exchange mechanism between CPU cores is presented later on.

The third data route (red) is the most commonly used by both cores. There the program instructions are carried out along with configuration and status data to and from peripheral registers, respectively. The compiled program for the main core slightly exceeds the capacity of L2 cache. This means, that for the most of the time, the code is stored in the L2 cache and the cores execute it, fetching from the cache memory. This drastically reduces the count of accesses to the DDR memory (where the program is stored) both cores have to perform, so the DDR controller could be set with a higher priority for the AXI HP PL to PS connection from the time counter. The second part of the red route is accessing peripheral registers. This route is used the least amount of time, so doesn’t require high data throughput.

This data flow configuration allows to transfer measurement data efficiently through the system with minimal interference with program code and peripheral configuration data.

IV. Translation Module

As it has been mentioned already, the time interval counter was connected to the AXI DMA module, which efficiently forwards measurement data to the DDR memory. Unfortunately, the AXI DMA module requires specific streaming interface, defined by the AXI Stream protocol [3]. Such protocol enables synchronous transfer data words, where the master passes data to the slave. Basic configuration of this bus requires signals: ACLK – clock synchronizing both master and slave; TDATA – the data bus, usually 32 bits wide; TVALID – master informs the slave, that the TDATA contains valid data; TREADY – slave informing the master that is ready for a next data word; TLAST – master informing the slave, that the current data word is the last one in the transaction. A typical transaction is depicted in fig. 5.

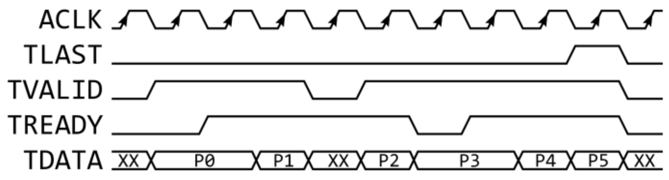


Fig. 5. AXI Stream signal diagram.

Signals are sampled during the rising edge of ACLK and are changed during the falling edge. With this set of signals, the master can control the latency of data sending. When the master is not ready to pass the next word, it clears the TVALID line, forcing the slave to enter a wait state. When the slave is not capable of handling new data waiting on the TDATA, it asserts a low state on the TREADY line. This forces the master to enter a wait state. In this way a simple but firm handshake is established. In this protocol, the master needs to know how many data words are needed to be sent. It is important, because the AXI DMA module samples the TLAST line, when to finish the transfer operation. In order to separate the data transfer functions from the TIC a data translation module was implemented. The time counter works in a single shot mode, generating one data frame per one measurement. To make the TIC work independently from the rest of the measurement system, an asynchronous data protocol was implemented. It is also a two point, master-slave link, where the master initiates a transaction, but the slave sends data. A typical time chart of this protocol is depicted in fig. 6. The protocol contains three lines: READY – master informing the slave, that is ready for a new data frame; FRAME – data bus where the slave presents data to the master; VALID – slave informing the master, that new valid data is presented on the FRAME bus.

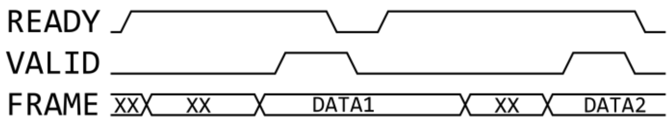


Fig. 6. Dedicated asynchronous TIC bus.

In the initial state, both READY and VALID lines are inactive (low) and the FRAME bus contains invalid data. When the master wants to receive new data, it sets the READY line. The slave starts operation (in this case the TIC starts a new measurement). When the slave is ready, it puts new data on the FRAME bus and sets the VALID line. When both VALID and READY lines are high, the master acquires data from the FRAME bus, which has to be stable. After data acquisition, master releases the READY line. Slave responds with resetting the VALID line. When both READY and VALID lines are again in inactive state, the slave is ready for a new transaction.

This asynchronous protocol is flexible, so that it can be implemented in a variety of measurement and communication designs. The FRAME bus may be wider than 32 bits, allowing to send great amounts of data in a one transaction. In fact, this feature is used in this design and is described later.

The lack of a global synchronizing clock allows to utilize modules working in separate clock domains without the worry of errors due to clock skew. Even without a clock signal, implementing such protocol in a SoC results in high data throughput, which is presented at the end of this paper.

The translation module was designed to serve as a bridge between the asynchronous TIC and synchronous AXI DMA modules. It has one configuration register to reset the module and to initiate a new measurement run. It also allows to set the desired measurement count. The module also has a status register, so that the CPU core can read the status of currently conducted measurements. One of the most important reasons of implementing the translation module was to divide a wide measurement frame into 32 bit words, in pursuance of AXI Stream protocol. The easiest way to obtain a sufficient behavior of the module, was to implement a finite state machine, controlling each of the presented busses according to configuration data present in control register. All states and conditions are depicted in fig. 7.

After power-up, the device is in RESET state. The nRES signal may be applied by the Zynq Power On Reset circuitry, or by the CPU, through the control register. In this state all internal registers, flags and multiplexers are set to default values. After releasing the module from the reset state, it enters IDLE mode. In this state, the module waits for new data transactions to begin by the AXI DMA. Also only in this state the translation module samples the configuration register for the measurement count. In order to simplify the design, software precautions have to be implemented, e.g. the AXI DMA must not be triggered for a new transaction before configuring the translation module. In addition, it is highly recommended to reset the translation module after each finished measurement run.

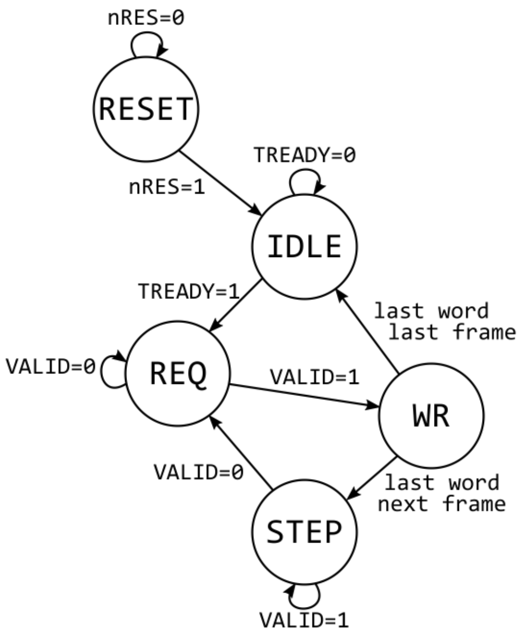


Fig. 7. Translation module state diagram.

When AXI DMA sets the TREADY line, the translation module enters the REQ state. In this state, the module sets the READY line issuing a new measurement in the TIC. The module waits until the TIC finished the measurement and outputs a valid frame (VALID set). If so, the module enters the WR state, where subsequent 32 bit words of the measurement frame are sent. One of the 32 bit words from FRAME is transferred to TDATA and TVALID is set. In the next clock cycle, if the TREADY is active, the AXI DMA read the data word. The module stays in this state, updating the TDATA bus with sequent 32 bit words of the measurement frame. If it is the last 32 bit word of the last measurement frame, additionally the TLAST line is set. If it was the last data word and last measurement frame, the module releases READY, TVALID and TLAST, ending the transaction and entering the IDLE state. If it only is the last word, but not the last frame, the module resets the TVALID and READY lines. Then it switches to the STEP state. In this state, the module waits for the TIC to end operation (indicated by releasing the VALID line) and prepare for the next measurement. Also in this state the measurement counter is decremented. When the TIC is ready (VALID in low state), the module moves to the REQ state, ready to initiate a new measurement process in the TIC.

V. TIME INTERVAL COUNTER

The TIC measures the elapsed time between two events represented by leading edges of two signals – START and STOP [7]. Achieving picosecond precision in a range of several seconds was done by combining two stage interpolation, which is an extended interpolation method invented by Nutt in 1968 [8]. It combines a standard period counter and equivalent coding delay lines for coarse and fine time measurements, respectively (fig. 8). The period counter provides a wide measurement range, vastly longer than the range of the subsequent stages of interpolation. With a standard 32 bit counter and 500 MHz clock, a total range of over 4 seconds is achieved [9]. Utilizing a chain of buffers, a 16 delayed clock signals are generated that create a multi-phase clock (MPC).

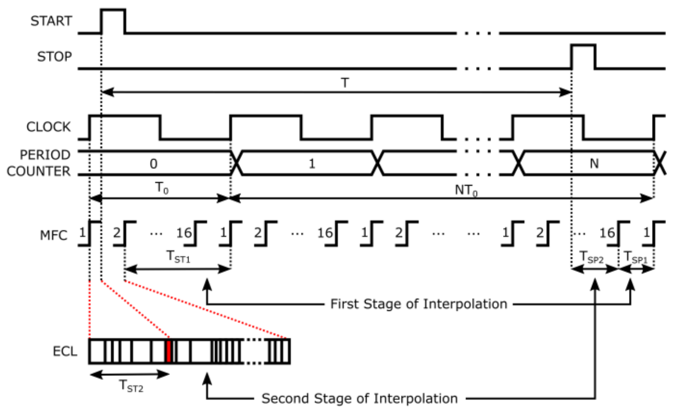


Fig. 8. Two stage interpolation method.

The first stage of interpolation (FIS) detects in which MFC the phase leading START/STOP edges occurred. By doing so, the measurement resolutions increases sixteen times in comparison to the counter method. In order to further increase the measurement resolution, in the second stage of interpolation (SIS), a delay line may be used. Instead of using a classic tapped delay line, a novel technique was applied, called multi edge coding in independent coding lines (described in detail in the next section). The measured time interval T is a linear combination of delay times measured by the period counter, FIS and SIS (1).

$$T = NT_0 + (T_{ST1} + T_{ST2}) - (T_{SP1} + T_{SP2}) \qquad (1)$$

where N – the number of counter periods, T – the clock period, TST1, TST2 – delays of START signal measured in first and second stages of interpolation respectively, TSP1, TSP2 – delays of STOP signal measured in first and second stages of interpolation respectively. Combining all presented components into a one module results in a time interval counter, with a structure shown in fig. 9.

The TIC consists of several modules. Each input signal has its own interpolation channel. In each there is the MPC that generates delayed clock phases for the FIS. The FIS detects START/STOP edge and MPC correlation, triggering the SIS. After delaying the measured signal (because of processing delays in FIS) the SIS is fed with a special signal called the 'pattern' (described in detail further in the text). Output of each interpolation stage is sent to code converter modules, compressing and reconditioning the measurement data. Triggering data from both interpolator FISs are driving the clock period counter through a synchronization circuit. The module counts clock periods between START and STOP edges.

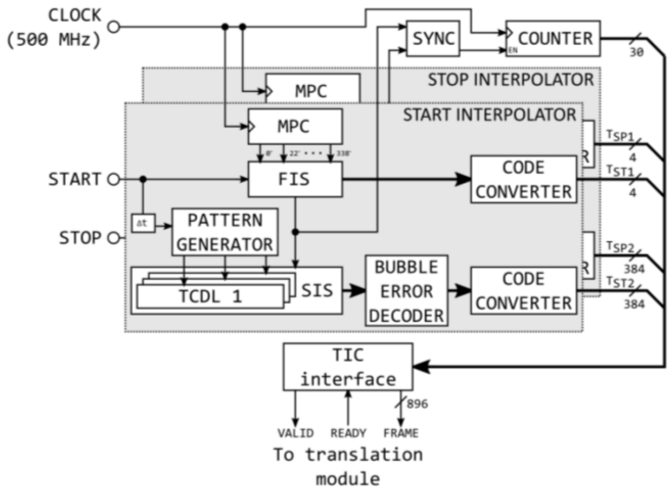


Fig. 9. Time interval counter block diagram.

The measurement data consists of 30 bits from period counter, 4 bits from each FIS and 384 bits from each SIS. This gives 806 bits of measurement data, which is transferred to the TIC interface. In this interface, the main TIC control is performed, resetting the module before each measurement and communicating with the rest of the system by the asynchronous frame bus. The measurement frame is 896 bit long and contains all data generated by the TIC. The frame is organized in seven 128 bit big endian words (table.).

TABLE. I
TIC DATA FRAME

Word number	Content (127:0 bits)	Bit position in frame
1	Generic field	(127 – 0)
2	SIS START 0	(255-128)
3	SIS STOP 0	(383-256)
4	SIS START 1	(511-384)
5	SIS STOP 1	(639-512)
6	SIS START 2	(767-640)
7	SIS STOP 2	(895-768)

The first word is the generic field. It contains four 32 bit words (from MSW to LSW): FIS STOP (number of MPC phase after which the STOP signal occurred), FIS START (respectively for START), PERIOD NUMBER, FRAME NUMBER. Although the output of each FIS is only 4 bits, resulting in 28 spare bits in each FIS field of the generic field, this approach was chosen, so that the measurement frame is 32 bit aligned. This was a specially good feature for designing the translation module, where frames had to be converted into 32 bit AXI Stream TDATA words. There words from the second to the seventh words contained paired SIS fields. Here three sets of SISs were used to increase the measurement resolution and precision.

Due to hardware limitations of the target printed circuit board (PCB), this time counter could not be directly implemented in the design. The main limitation was lack of special low-jitter input buffers conditioning START and STOP signals, before entering the PL fabric of the Zynq device. The presented counter was implemented (with an USB interface) in a design described in [9]. In this particular design a special module imitating the behavior of the TIC was implemented and is described in following sections.

VI. TIME CODED DELAY LINE

The second stage of interpolation is based on the method of the multi-edge coding in independent coding lines method [10]. It utilizes tapped delay lines, in which a special signal called the 'pattern' is propagated. The interpolator is depicted in fig. 10. The pattern generator generates a square wave with six edges when triggered by the input signal. Once generated, the pattern propagates through the tapped delay line. At the end of the measurement, the delay line is latched into associated D-type flip-flops. The synchronization signal needs to be delayed (Δt), according to the pattern generator delay in order to reduce the interpolator's dead time. The output of each flip-flop is forwarded to a code converter, which eliminates conversion errors (called 'bubble errors'). The converter also reduces the required amount of bits, on which the measurement result needs to be saved. A bubble error occurs mainly due to hardware inequalities of D flip-flops clock input threshold voltages or supply voltage noise.

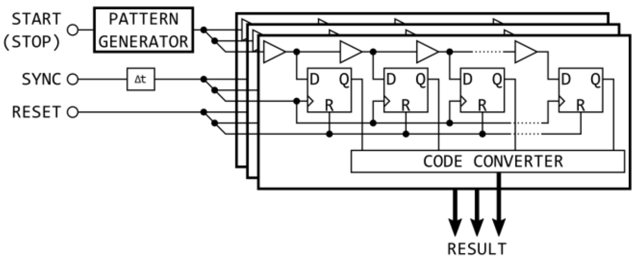


Fig. 10. Time coded delay line.

The TIC was implemented in a Spartan-6 FPGA device, where 256 tap lines were needed to cover 125 ns measurement range. It is the time duration between two subsequent MPC phases generated with a 500 MHz clock. Using a code converter, only 128 bits were needed to compress the output word of the delay line in a lossless manner. The code converter takes the delay line word (256 bits), eliminates bubble errors and extracts numbers of flip-flops on which pattern edges were saved, marking corresponding bits as '1' and clearing others. This process may be depicted as low-pass filtering and calculating the derivative module of the input signal, bit by bit. The converters splits such signal into sixteen bytes. The pattern generator and signal paths inside the interpolator are trimmed in such a way, that for each of those 16 bytes there are only 15 possible combinations of zeroes and ones. If an illegal combination is discovered, it is marked as the 16th state (error). So it is possible to save each combination number using four bits, making the compression ratio equals 2:1.

VII. SOFTWARE MODEL

The software was written in C, using a highly structural approach [11]. The chosen language gives better performance in time critical operations in the CPU, than higher level object languages (e.g. C++, Java). This design does not implement an operating system, so it need more time constrains between software modules, than it would be for a OS design. The compiler was a trimmed version of GCC compiler. The software model was divided into four logical groups (fig. 11) [12].

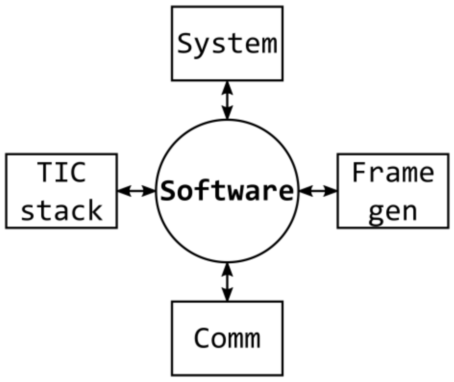


Fig. 11. Structure of design software.

First group is called System. It contains a date/time manipulation library, custom definitions of data types used in the design, initialization and control routines and the main project function. The second group, tagged 'Comm', includes the inter-CPU mailbox protocol and UART state machine. The 'TIC stack' group contains a four level software stack responsible for communicating with the TIC hardware, configuring it, triggering calibration and measurements, downloading and processing measurement frames as well as computing the measured time intervals. The last group called 'Frame gen' contains a software frame generator, emulating the time interval counter behavior.

Dividing the software in such a manner eases software maintenance. It also allows making fast changes in the design. If e.g. the designer would change the TIC hardware, one would have to change only one layer of the 'TIC stack' group, leaving the rest of the project untouched.

VIII. INTER-CPU MAILBOX MECHANISM

In a multi CPU system one of the most important issues is to establish a reliable and efficient connection between CPU cores [2]. As program code has to be generated independently for each of the cores, there has to be a standard way of exchanging data. The connection should have little latency and not require additional handling (refreshing, handshaking). That is why a mailbox style of messaging was implemented in the OCM of the application processing unit of the PS subsystem (fig. 12).

The OCM is divided into two banks. Each bank is unidirectional. In a typical mailbox scheme, a postman (sender) puts the message in the mailbox and raises a flag. The recipient only needs to check if the flag is raised. If so, it opens the mailbox, reads the information and resets the flag. This mechanism consists of two main elements: a flag and payload. If a CPU wants to send new information, it should first check the corresponding 'send flag', if the last message was read. If the flag is low, it should copy the desired content to the payload area and finally set the flag. The receiver CPU would check the flag, copy valid content and lower the flag after finishing download. Each CPU has its own transfer area, so that a full duplex communication is available. Dividing the OCM into two areas gives almost 128 kB (minus space for the flags) of payload for each core. At this level of abstraction it is does not matter what is transferred in the payload area.

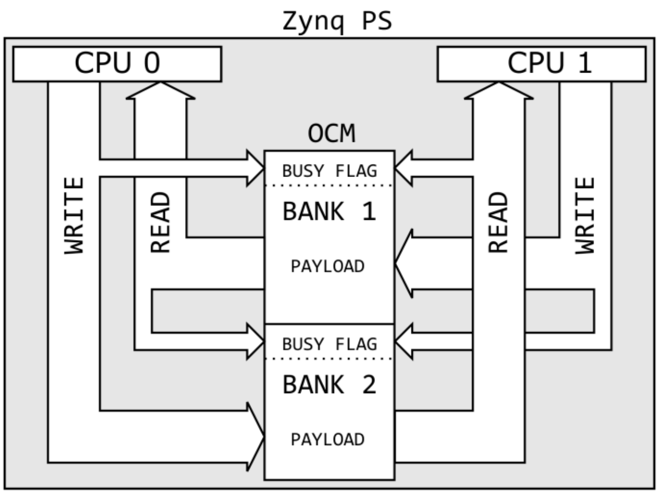


Fig. 12. Inter-CPU mailbox mechanism.

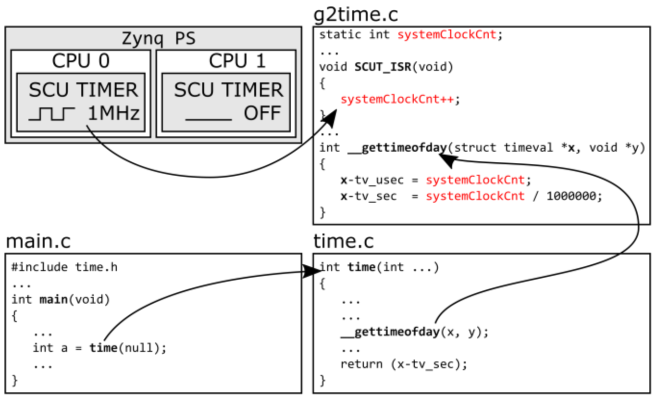
The software handling this protocol was written in a fully structural technique, so that each CPU core acquires information about oneself and automatically configures own OCM bank to proper operation.

IX. Time Stamp Generation

The C language standard implements a standard date/time manipulation library. The only issue is that the standard does not specify the hardware layer of that library. It is essential, because the time interval ought to be claimed from an interrupt service routine (ISR). The CPU usually jumps into a timer ISR, when the desired timer has reached zero and needs to be reloaded with its preprogrammed value. Inside that ISR usually an integer is incremented, representing the elapsed time from the system power-up. For example, if the timer is fed with a 100 MHz clock, and it has a preprogramed value of 100, after each 100 ticks the timer would generate an interrupt. This would make the CPU enter a proper ISR every 1 µs. Inside that ISR, a static variable is incremented. The variable represents how many microseconds passed since system power-up. The C compiler provides a low level function called ‘__gettimeofday’. The programmer can implement the static variable into that function. By doing so, the standard ‘time’ C library automatically manage date time presentation format for the system (fig. 13).

The mentioned function has to acquire a pointer to a specific time structure. Inside that structure it has to update two elapsed time fields: in seconds and microseconds. That elapsed value can be read from the presented ISR time counter. The system timer is based on a SCU timer, located in CPU0 private address space. It generates interrupts every 1 µs. Inside the ISR an integer is incremented.

When the program needs to get present time value, it calls to the standard ‘time’ function, which returns an integer equal to the number of seconds, which passed since the system power-up. In turn, the standard functions calls for the implemented ‘__gettimeofday’ function, waiting for certain time structures to be updated. After the update, the ‘time’ function processes new data and returns to the main program.



```
static int systemClockCnt;
...
void SCUT_ISR(void)
{
   systemClockCnt++;
}
...
int __gettimeofday(struct timeval *x, void *y)
{
   x-tv_usec = systemClockCnt;
   x-tv_sec  = systemClockCnt / 1000000;
}
```

```
#include time.h
...
int main(void)
{
   ...
   int a = time(null);
   ...
}
```

```
int time(int ...)
{
   ...
   ...
   __gettimeofday(x, y);
   ...
   return (x-tv_sec);
}
```

Fig. 13. Simple way of binding standard time library with the system.

Despite the software overhead, it is worth implementing standard functions, because of two main reasons. First, the software is less error prone, because standard libraries are mostly delivered as high quality add-on to compilers. Second, the main software can be compiled and tested on every platform implementing a C standard. In this case, more 85% of the software was developed and tested on a standard PC, substituting only the bottom-most functionalities while migrating to the Zynq platform.

X. Command Line Interpreter

An advanced measurement system should be equipped with a straightforward user interface (UI), that should give the user a handset of commands to control the system. It also should correct at least minor input errors which may appear. One of the most basic UI is a command prompt. Analyzing vast majority of OS text interfaces (MS PowerShell, UNIX bash, sh, csh, etc.), a typical command prompt should allow the user to input commands with additional parameters. The command line interpreter (CLI) was developed in order to enter text commands with parameters. The module operates on the input string taking four steps: text preprocessing, command division and searching, memory allocation and function invocation.

In the first step, CLI analyses the input string, detects unnecessary leading, trailing and multiple separators (fig. 14) and eliminates them. The standard separator is a spacebar (ASCII 0x20). After preprocessing, the input string is being divided into separate words. The first word is always the command, and the other are parameters (fig. 15).

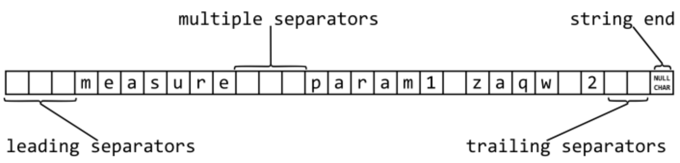


Fig. 14. An input string containing the user command with parameters.

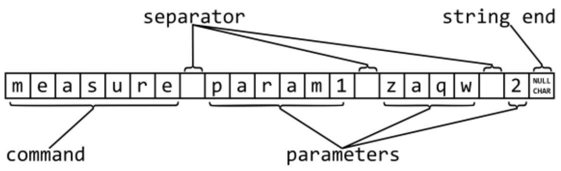
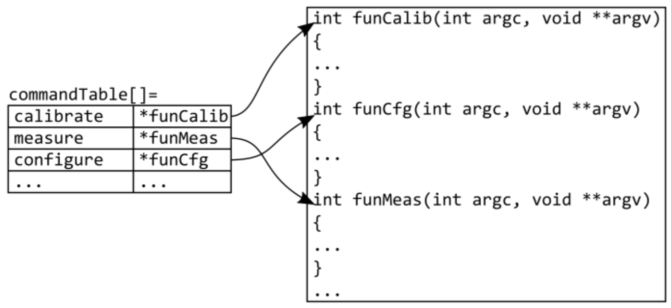


Fig. 15. The user command after pre-processing.

The software contains a table with strings representing commands along with pointers for functions executing those commands. Each function is declared according to the UNIX ABI (UNIX application binary interface), where the function get a number of passed parameters and a table of pointers to strings with those parameters (fig. 16). The CLI compares the first word of the input string with each entry of the data table. If it finds a matching word it proceeds to the next step. Other way, it returns an error status. In the last step, the CLI dynamically allocates memory for all command words as depicted in fig. 17.



```
commandTable[]=
calibrate   *funCalib
measure     *funMeas
configure   *funCfg
...         ...
```

```
int funCalib(int argc, void **argv)
{
...
}
int funCfg(int argc, void **argv)
{
...
}
int funMeas(int argc, void **argv)
{
...
}
...
```

Fig. 16. Main command structure with linked functions.

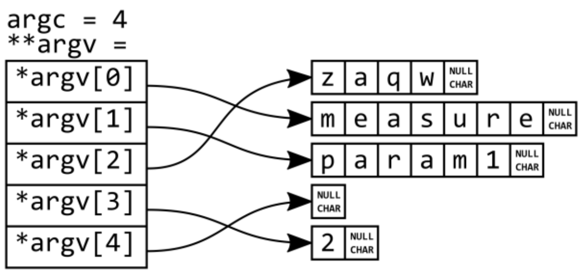


Fig. 17. Dynamically allocated strings for the evaluated function.

Each 'argv' entry is a pointer to one word, terminated by a null character. The standard requires to allocate one additional string containing only the null character. The 'argc' parameter is the number of arguments (not counting the last null character). After preparing 'argv' and 'argc' fields, the CLI invokes the function associated with the first word of the input string. In this example it would be 'funCalib'. Each function returns an integer value. This value is captured by the CLI and returned to the main program after releasing allocated memory resources.

XI. Frame generator

As mentioned earlier, the TIC module could not be implemented because of hardware limitations of the target PCB. In this case an equivalent peripheral served as a dummy TIC. At one side connected to the AXI GP port, where the software loaded a generated data frame. On the other side, it was connected to the translation unit, simulating the behavior of the TIC. The frame generation module was using a standard pseudo-random number generator, which may be found in the 'stdlib' C library. It generates integer values with a uniform probability density from zero to 32767. It also has a seeding function, which allows to initialize the generator each time with a different starting value. The frame generator produces random values only in the FIS and every SIS of the measurement frame. It may work in one of two state: calibration and measurement. In calibration mode, the frame generator produces FIS and SIS fields with uniform probability density functions (PDFs). This is because in the real TIC a calibration was performed with the use of a standard code density test [13]. The measurement mode was designed to imitate the TIC's behavior during standard measurements, which means a normal PDF for each detected pattern edge in SIS. This is because the input time interval represents a normal PDF. These two methods of operation are depicted in fig. 18. Testing the real TIC behavior during calibration it was noticed, that every pattern edge appeared within specific ranges (table).

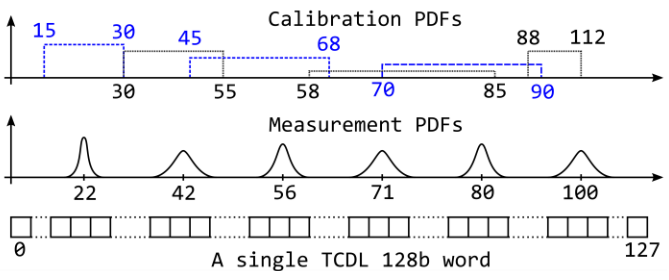


Fig. 18. Calibration and measurement PDFs for an artificial TCDL word generation.

TABLE II
UNIFORM PDFS OF PATTERN EDGE POSITIONS.

Pattern edge	Minimal position	Maximal position
1	15	30
2	30	55
3	45	68
4	58	85
5	70	90
6	88	112

The mechanism for generating pattern edge positions is relatively simple. Each position was randomized within specified range. If two sequent pattern edges were generated on positions closer than to one another than 5, the second edge position is shifted, thus preventing overlapping. In order to generate pattern edge positions with a normal PDF a Box-Muller generator was used. The mean and standard deviation values for each edge were chosen as the mean value and 1/8 of corresponding uniform PDF range respectively (table iii).

TABLE III
NORMAL PDFS OF PATTERN EDGE POSITIONS.

Pattern edge	Mean	StdDev
1	22	1
2	42	3
3	56	2
4	71	3
5	80	2
6	100	3

After generating all 6 pattern edge positions, those values were converted backward according to the TCDL converter in the SIS, so that the result would be a 128 bit word, which could be generated in a real measurement. This procedure is repeated for each of 6 SIS (three for START and three for STOP channels). In the measurement mode, the FIS fields contain constant values, while in the calibration mode, the FIS fields contain randomized numbers, in a uniform range between 0 and 15. The period counter contains zero during calibration and a constant value during measurement. The frame counter contains the number of current frame being processed.

XII. Frame Decoding and Processing

The frame decoding piece of software is directly responsible for proper configuration and communication with the TIC in the PL. It was designed as a three level stack, where each level is responsible for a different functionality of the device. The lowest (zero) level is responsible for configuring the translation module and AXI DMA. It also receives data frames from the TIC. The software abstraction is done on this level. When a major change in hardware occurs, e.g. a TIC with a completely different interface is implemented, only this level has to be rewritten, leaving the rest unmodified. The whole software could be written and tested on PC, using a standalone GCC compiler, before integrating the system together.

The middle (first) level contains the frame decoder. This decoder is specific for the measurement method, where the SIS implements multi-edge coding in independent coding lines. If the structure of the TIC would change, e.g. instead of two stage interpolation, time stamp method would be implemented, this level would be the only one to be rewritten. Certain precaution were taken while developing this level, e.g. when the number of coding lines in every SIS would have to change, the software designer would only have to change one macro-definition recompile and reload the code. The frame decoder analyses one frame at a time. First it decodes the generic field, recovering both FIS values, period and frame counters. Next it sweeps remaining six 128 bit words, decompressing and acquiring pattern edge positions for each line. Decoded data are saved either into a calibration table or measurement dynamic list, depending on the current system operation.

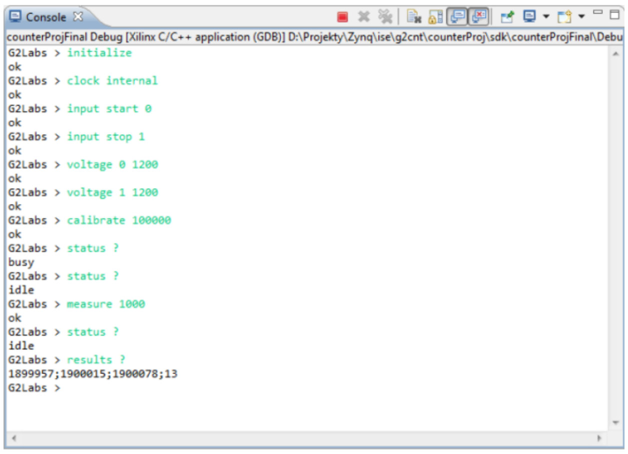
The highest (second) level is the calibration and measurement level. This level is generic for most counters. If there had to be e.g. a major change in TIC way of measuring time intervals or changing TIC into a completely different measurement device, leaving communication considerations unchanged, only this level would have to be rewritten.

During calibration, a structured table is allocated, where each cell corresponds to one equivalent coding line step of each interpolation channel. Each step has its own width, value (in nanoseconds), DNL and INL values [13]. The content of this table serves as a transfer characteristic while conducting measurements. In the measurement mode, the decoding module allocates a dynamic linked list, filling it with computed measurement point. Each point contains the decoded FIS, SIS and counter fields, along with a system timestamp when the measurement occurred. It also contains the computed mean value of the measurement (in nanoseconds) along with error span (which is the width of both START and STOP quantization steps of SIS). Thanks to a custom memory management system, the dynamic linked list is allocated and released faster, than using the ‘stdlib’ standard C library, containing ‘malloc’ and ‘free’ functions.

XIII. Test Results

The presented design was implemented in an AVNET ZYNQ Mini Module-Plus (MMP) with AVNET MMP BASEBOARD2. The MMP board contained a XC7Z045 Zynq SoC, with a Kintex-7 equivalent PL subsystem. The original TIC for test purposes was implemented in a Spartan-6 FPGA on a custom PCB. The complete project was designed and tested using the Xilinx ISE Design Suite 14.7. The software for APU cores was written in Xilinx SDK 14.7. The output code was uploaded to a µSD card, from which Zynq fetched the code during boot-up. The system was tested in respect of three parameters: reliability, raw data throughput, calibration and measurement duration time. During the first test properly responds to control commands sent through serial console were verified (fig. 19).

The module responded properly to each command. The serial console could be substituted with an additional channel in the inter-CPU protocol, and could be controlled by additional user interface programs, giving chance of implementing e.g. a graphic LCD with touch panel. Second test was conducted to obtain information about the system throughput. In this case one measurement frame was generated and transferred to the TIC simulation hardware. Next, the software ‘conducted measurements’ reading that constant frame. The AXI DMA was set to read different data sizes each time (fig. 20).



```
counterProjFinal Debug [Xilinx C/C++ application (GDB)] D:\Projekty\Zynq\ise\g2cnt\counterProj\sdk\counterProjFinal\Debu
G2Labs > initialize
ok
G2Labs > clock internal
ok
G2Labs > input start 0
ok
G2Labs > input stop 1
ok
G2Labs > voltage 0 1200
ok
G2Labs > voltage 1 1200
ok
G2Labs > calibrate 100000
ok
G2Labs > status ?
busy
G2Labs > status ?
idle
G2Labs > measure 1000
ok
G2Labs > status ?
idle
G2Labs > results ?
1899957;1900015;1900078;13
G2Labs >
```

Fig. 19. System reliability test.

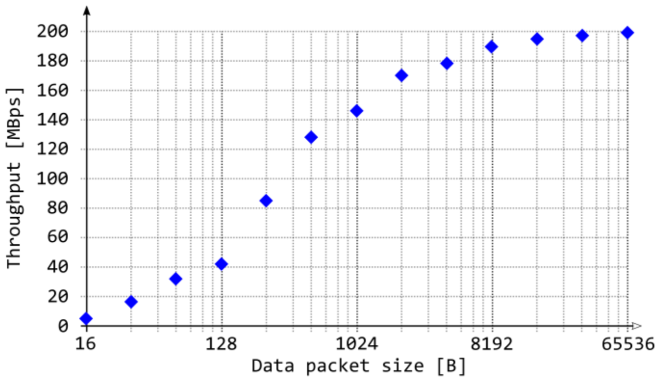


Fig. 20. Test results of data throughput in the system.

It may be noticed, that for small data packets the total throughput is low, whilst for larger packets the throughput increases, saturating at about 200 MBps. Considering maximum PL to PS Zynq throughput of 300 MBps [3] and a chain of transfer points in the PL, this result is satisfactory. One of main reasons of such throughput distribution is that the L2 cache controller gives a higher priority for bigger data packets than for smaller ones. With a higher priority, measured frames are fetched from the DDR memory more quickly than e.g. the software code for a CPU core.

The last test was performed during the normal device operation, where a standard 10^6 point calibration and 10^5 point measurements were conducted in two cases. First, the TIC was simulated in pure software (generated frames were copied to the download area), second the TIC was simulated in hardware, utilizing the translation and AXI DMA modules. Results are show in table iv with comparison to the classic system approach where Spartan-6 FPGA with the TIC was communicating with PC host via USB 2.0.

TABLE IV
COMPARISON OF OPERATION TIMES OF TIC EMULATED IN SOFTWARE AND HARDWARE.

Zynq software TIC simulation		Zynq hardware TIC simulation		Spartan-6 TIC	
Calib.	Meas.	Calib.	Meas.	Calib.	Meas.
3 s	2 s	15 s	5 s	30 s	15 s

It is clearly show that the hardware TIC simulation performs its operation 2 to 3 times faster than the classic approach, where the measurement system communicates with PC software via physical link. The main reason is that Zynq gives better opportunities for implementing custom data buses. In a classic approach, the designer is constrained to hardware limitations of used interface ICs, PCB design limitations etc. On the other hand, one must remember, that a SoC requires designing every piece of hardware and software from the scratch, becoming a more error prone approach.

XIV. CONCLUSIONS

A complete precise measurement instrument was implemented as SoC in a single FPGA device. Such implementation releases new vital opportunities by reducing the size and power consumption of the instrument. In addition, integrating the measurement system substantially increases the measurement rate in comparison with traditional approaches, where the device had to communicate with a host through third party interfaces, usually being bottlenecks of fast designs.

REFERENCES

[1] R. B. Northrop, “Introduction to Instrumentation and measurements”, Taylor & Francis Group, Broken Sound Parkway 2005

[2] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[3] http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[4] G. Grzęda, “Control and data processing module for time interval counter in SoC device” (*in Polish*), WAT 2015

[5] http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[6] http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[7] R. Szplet, “Time-to-Digital Converters”, in Design, Modeling and Testing of Data Converters, Berlin, Springer-Verlag, pp. 211-246,2014

[8] R. Nutt, “Digital time interval meter”, Rev. Sci. Instrum., vol 39, pp. 1342-1345, 1968

[9] R. Szplet, D. Sondej, G. Grzęda, „Interpolating time counter with multi-edge coding”, EFTF/IPC, pp 321-324, July 2013

[10] K. Klepacki, R. Szplet, R. Pełka, „A 7.5 ps single-shot precision integrated time counter with segmented delay line”, Review of Scientific Instruments, vol 85, no. 3,2014

[11] B. Kernigham, D. Ritchie, „The ANSI C Programming Language”, Prentice Hall, Englewood Cliffs 1995

[12] G. Grzęda, D. Sondej, R. Szplet, „Diagnostic and control software for interpolating time counter in programmable logic device” (*in Polish*), Measurement Automation Monitoring, vol 60, pp. 441-443, 2014

[13] S. Cova, M. Bertolaccini, „Differential linearity testing and precision calibration of multichannel time sorters”, Nuclear Instruments and Methods, vol. 77, pp. 269-276, 1970