Implementation of a Flexible Trace Buffer with Powerful Trigger Facilities for Real-time Debugging of a FPGA-based Microcontroller

Rainer Bermbach
University of Applied Sciences
Braunschweig/Wolfenbüttel
Salzdahlumer Str. 46/48
38302 Wolfenbüttel, Germany
r.bermbach@fh-wolfenbuettel.de

Ulrich Germann
University of Applied Sciences
Braunschweig/Wolfenbüttel
Salzdahlumer Str. 46/48
38302 Wolfenbüttel, Germany
u.germann@online.de

Martin Kupfer
University of Applied Sciences
Braunschweig/Wolfenbüttel
Salzdahlumer Str. 46/48
38302 Wolfenbüttel, Germany
ma.kupfer@fh-wolfenbuettel.de

ABSTRACT
Debugging software for a FPGA-based microcontroller can be a strenuous task without facilities like single stepping, register and memory dumps etc. Even more critical is the debugging of real-time software like interrupt service routines. A debugger for a FPGA-based microcontroller written in VHDL has been enhanced in hard- and software to provide a flexible trace buffer for non-interfering real-time recording of microcontroller program, data and status information as well as freely definable user signals for later off-line inspection. The Block RAM feature of the FPGA is used to store the information. For multiple recordings the memory may be divided into several parts of suitable size permitting pre- and post-trigger modes. Complex trigger facilities (watchpoints) allow to trigger on various conditions for an efficient deployment of the available memory. Up to three independent watchpoints can be combined by logical operations and defining sequences or to arm single events just for a certain time. The self-developed software front-end of the debugger that communicates via JTAG interface with the hardware and the interfaces in the FPGA has been extended for easy arming of the watchpoints and configuring of the trace buffer as well as for user-friendly display and storing of the trace buffer data.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Tracing, Debugging aids;* D3.4 [**Programming Languages**]: Processors – Debuggers; B.8.1 [**Hardware**] Performance and Reliability – Reliability, Testing, and Fault Tolerance; C.0 [**Computer Systems Organization**]: General – *Hardware/software interfaces;* D.2.2 [**Software Engineering**]: Design Tools and Techniques – *User interfaces.*

General Terms
Performance, Design, Reliability, Verification.

Keywords
Microcontroller, Microprocessor, SoC, FPGA, VHDL, Soft-core Processor, Watchpoints, Trace Buffer, JTAG Communication, Real-time Debugging.

1. INTRODUCTION
Modern SoC designs often face difficulties in debugging software for the microprocessors embedded in the design. Where powerful in-circuit emulators or on-chip hardware assisted debuggers allow efficient testing of the software with standard controllers and processors problems arise when using soft-core processors etc. in FPGAs. Often the only solution is using old-fashioned software debuggers which typically require some of the processor resources by themselves. Frequently, interference of the debugger with the target device makes full system tests impossible. The situation becomes even more complicated when one needs to debug software routines which cannot be stepped through but must be run in real-time for verifying their correct implementation, for example, interrupt service routines, external communication routines and real-time data processing. Often the only possibility is routing internal signals to FPGA ports for external control of software behavior.

At the Computer Engineering Lab of the University of Applied Sciences Braunschweig/Wolfenbuettel a PIC™-compatible microcontroller called VHDL-PIC had been developed and optimized in former projects [1, 3, 4, 7]. The VHDL-based controller, fully compatible to the cores of the PICmicro™ mid-range MCU family [11], features typical peripherals such as ports, timer, UART, etc., at a frequency of up to 100 MHz, i.e. with processing power of up to 25 MIPS.

Developing software for this controller (implemented in a Xilinx Spartan 3 FPGA) revealed the typical problems mentioned above. In addition, new code had to be transformed into VHDL ROM initialization code and all the steps of the hardware implementation process had to be run through again. To solve these problems a hardware assisted non-interfering debugger was developed [5, 9]. The VHDL hardware interface connects to the microcontroller

and communicates via the JTAG port with the user interface (debugger front-end) on a Microsoft Windows™ based PC. In addition, code can be downloaded directly without anew hardware implementation.

Though the debugger has all the typical and necessary functions the debugging of real-time software routines still remained nearly impossible. As the JTAG interface does not permit high-speed data transfer which would be needed to send real-time processor status information the development of a flexible trace buffer was undertaken to record and store those information during real-time processor runs. To derive maximum benefit from the features of the trace buffer powerful watchpoints allow triggering of the respective recordings. The necessary hardware for trace buffer and watchpoints was included in the hardware interface on the FPGA whereas the software front-end was enhanced to configure and arm the triggers as well as present and store the trace results.

The development and implementation of the trace buffer and the watchpoints are presented in this text. In the following section a system overview will sketch the whole system concentrating on the debug module as a whole, whereas section 3 describes the features and the implementation of the new watchpoints. Section 4 discusses the trace buffer, its modes and functions and section 5 gives some detail on the software front-end enhancements for the watchpoints and the trace buffer and passes over to the conclusion.

2. SYSTEM OVERVIEW

As mentioned above, the development environment of the VHDL-PIC consists of a hardware interface in the FPGA, the communication via JTAG and the software front-end on a PC hosting the debugging software and the graphical user interface, see figure 1. The front-end presents the status of the microcontroller to the user. As the debug module has only access to the controller's signals and busses when it is stopped, a timer regularly checks whether the processor is running or not. In latter case the front-end reads all relevant information (registers, memory, ports etc.) and updates the respective windows. Changes to the previous state are marked in red for convenient debugging.

Front-end and back-end communicate over a USB or a parallel port on the PC side and the JTAG port in the FPGA. The JTAG's test access port (TAP) controls the communication. It provides two shift registers, USER1 and USER2. To send a command to the hardware interface, it is shifted into one of these registers. The hardware interface stores its response in the respective shift register which is shifted out to the software front-end with the next transmission [5, 9]. The communication is always front-end driven, the hardware cannot initiate any data transfer. The debugging software provides several functions to navigate the TAP and to shift the USER registers. The 32-bit USER 2 register is used to transmit read or write commands where the lower 16 bits comprise the data and the upper ones address the memory position and hold the command itself. The 8-bit USER 1 register receives short commands like a manual stop or inquiries like 'has the PIC stopped?'. The hardware interface connects to the communication interface and decodes the received commands and feeds different signals and busses to get the requested information.

All registers and memories can be read and modified by the hardware interface. To access the registers the hardware interface feeds the normal data bus. The program memory and the internal

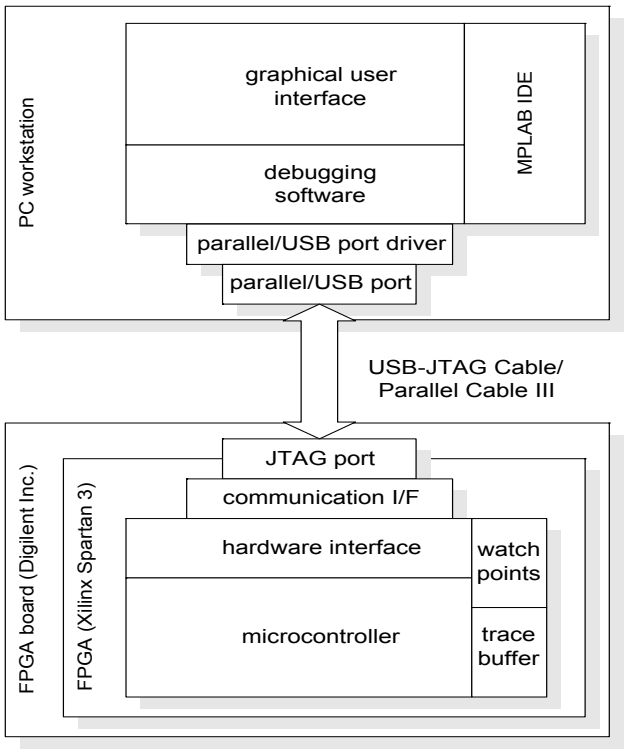

Figure 1. System block diagram.

RAM are implemented as dual-port RAMs for easy read and write access by the controller and the debug module.

The processor has a 14-bit instruction word which is stored in a 16-bit dual-port RAM. The unused two MSBs mark breakpoints which therefore can be set at every instruction. The software differentiates between static and temporary breakpoints. Temporary breakpoints will be deleted when reached and are used to implement functions like 'run to cursor', 'step over' or 'step out'.

The hardware interface connects to the new watchpoint and trace buffer hardware the same way it does with memories and registers configuring them and reading back relevant data. The following sections describe the configuration of the watchpoints and the read-out of the trace memory in more detail.

3. WATCHPOINTS

As breakpoints mark specific code lines watchpoints are more flexible and control dedicated internal occurrences or states of the processor. If such a predefined event happens the processor may be stopped or data recording in the trace buffer unit may start. Thus watchpoints may confine a data recording to a more relevant section of the executed program. As mentioned above, debugging FPGA-based processors is often lacking features like watchpoints and trace buffers. Most available solutions concentrate on ASIC implementations for example as in [2].

The triggering process is a two-layered approach. On the first level one defines the event on which a specific watchpoint hardware will 'fire'. The second level (combination layer) combines the triggering action of several watchpoints to a complex trigger event. The current implementation possesses three independent

watchpoint instances each owning its individual hardware resources. Those three may be flexibly linked to form more complex trigger conditions.

3.1 Defining an Event in a Watchpoint

Each instance accesses the relevant signal paths in the processor. This encompasses both, internal control busses as well as full data paths e.g. at the ALU or the program memory.

The following event sources can be monitored:

- executing a specific command,
- read and/or write access to a dedicated RAM location (GPR) or special function register (SFR) or a location within specific address range,
- executing a dedicated code line in program memory or any command within specific address range,
- a specific result (or value range) of the ALU operation,
- alteration of CARRY or ZERO flag,
- occurrence of an interrupt or an timer 0 overflow

All above mentioned event sources can be combined by either a logical AND or a logical OR.

Monitoring of control signals solely samples an alteration of the signal. Monitoring internal data and address paths offers an additional way for defining an event. At first, the given value is compared against the current state on the path by a comparator. In addition, it is possible to mask out several bits by a filter value. For example, masking out the two LSBs of a data memory address allows monitoring an address range instead of a single address value. In addition, by use of an occurrence counter, see figure 2, single or multi-occurrences of the qualified event may lead to a trigger signal.

3.2 Combining Events

Combining independent watchpoint instances allows the definition of powerful and flexible trigger conditions to start a data recording or to halt the processor. The simplest way to combine single instances is a logical OR. In this case an occurrence of a trigger event of a single watchpoint suffices to start the data recording.

Cascading watchpoint instances allows more specific qualification of the final trigger event. If cascading, the trigger signal of instance I is not used to start data recording, but to arm instance II. Thus the first trigger signal starts the monitoring of event II. Arming the subsequent event can be limited by a cycle counter which is loaded with a pre-calculated value. After detection of trigger I the cycle counter starts. If this counter overflows before event II occurs instance II will be disarmed again. If instance II triggers within the predefined number of processor cycles, the signal is passed on to the trace buffer or – if cascading three instances – is used to arm instance III. Each cycle counter can be set to a separate value.

For optimal use of the available memory the trace buffer recording may be restricted to a certain number of recording sessions. In that way trigger signals are ignored when the trace buffer number has been reached. This allows the user to change the trigger conditions and fill up other parts of the memory with additional recordings.

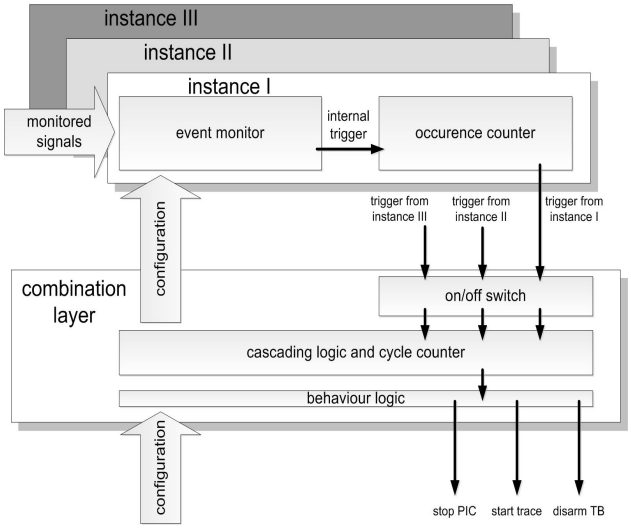

Figure 2. Cascading watchpoints.

Instead of stopping the trace buffer recording it is also possible to stop the processor. This allows running to a dedicated point of the program, stop the execution and analyze the following parts in single step mode. Because the trace buffer is still armed, the single stepped executions are recorded also.

3.3 Configuring Watchpoints

Data exchange between watchpoints and front-end works unidirectionally by passing configuration data through the trace buffer components. Configuring the combination layer includes settings for combining several instances. In case of a simple OR only the comparative values for disarming the trace buffer or stopping the processor are to be set up. In case of cascading the instances also the values of the cycle counters have to be programmed. To allow individual addressing each instance is equipped with a hard coded ID.

4. TRACE BUFFER

The trace buffer is responsible for correct and reliable real-time recording of the processed data to allow the later off-line analysis. Therefore it is necessary to regard the internal processor timing and the execution of the instructions. The implemented version of the trace buffer is a trade-off between preferably low hardware assignment and an efficient use of the available memory.

4.1 Data Memory

The trace memory is constructed using the Block RAM resources of the used Spartan III FPGA. For targeting various requirements the trace buffer provides data recording in different modes:

- recording a program part as large as possible (single session mode),
- multiple recording of a limited program part (multi-session mode),
- the recorded program part is starting at time of trigger (post-trigger mode),
- the recorded program part contains data which were chronological ahead of the trigger impulse (pre-trigger mode).

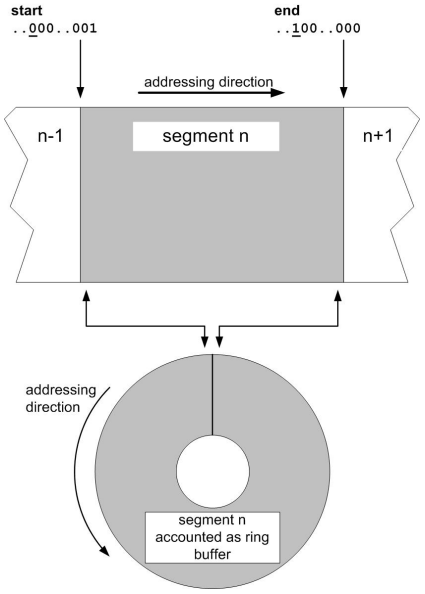

Figure 3. Trace buffer segment addressing.

To achieve the above mentioned characteristics a flexible memory allocation is needed. While the management of post-trigger recordings is quite easy, the recording of pre-trigger data requests some more considerations. To avoid the fragmentation of the memory it has to be divided into discrete parts, each used as a ring buffer, see figure 3. For an effortless distinguishing of the upper ring buffer limit, the dimension is fixed to a power of two. Furthermore the addresses of the section are running from ...000...001 (instead of beginning with ...000...000) to ...100...000 so that the last address of every section toggles the so-called indicator bit from 0 to 1 (as shown in figure 3). In post-trigger mode, hitting the upper border stops data recording. In pre-trigger mode it resets the ring buffer address counter.

Opposite to post-trigger mode, where recording starts by occurrence of the trigger signal, in pre-trigger mode the data is stored continuously before the occasion of trigger as well. To preserve the delimited pre-trigger part while the triggered recording is in progress a corresponding stop address has to be calculated. Preventing a complex address calculation at runtime, most of the values needed are pre-calculated by the software front-end. Finally, hardware calculates the stop address in three clock cycles. As shown in figure 4 there are three cases to be distinguished:

1. The part of pre-trigger data is fitting exactly between the segment's start address and the point of trigger. Also, end of the segment is exactly the stop address.
2. The part of pre-trigger data is coherent in the middle of the ring buffer. In this case the stop address is lower than the point of trigger. When calculating the stop address the overflow of the address pointer has to be considered. A subtraction of pre-trigger depth and point of trigger results in the stop address.
3. The pre-trigger part is divided into two sections. Similar to case 1 the stop address is higher than the point of trigger. For calculating the stop address the post-trigger depth has to be added upon the point of trigger.

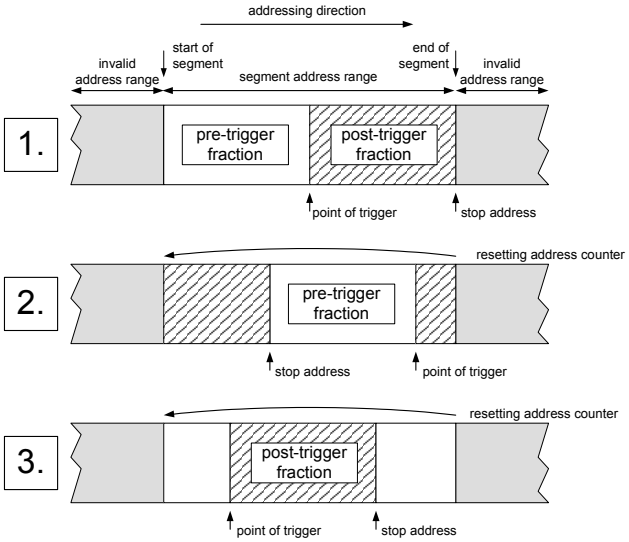

Figure 4. Pre-trigger position in trace buffer segment.

4.2 Data Recording

There are two different groups of data sets which will be stored within a session. The trigger data set comprises data of selected special function registers, the values of the program counter and the Block RAM address counter. These values are saved at occurrence of the trigger signal and are needed for reconstructing and interpreting data in the front-end. This data set allocates the two lowest locations in each trace buffer segment.

The pre-trigger and post-trigger data sets consist of runtime data e.g. the value of the data bus or arithmetic flags. Because of the small capacity of on-chip memory only dynamic values are saved at runtime. By contrast fixed values e.g. FILE-Register addresses or literals are not stored at runtime. They can be read and reconstructed from program memory or a related hex-file.

When executing jumps runtime data will differ from recording a regular instruction. To allow reconstruction of the pre-trigger part both, source and target address of the program counter are important to know. On the other hand values of the data bus or arithmetic flags are uninteresting in this case.

For an improved and more flexible use of the trace buffer unit, eight bit called 'user byte' are reserved to save special signals. Assigning an arbitrary data source within the VHDL code to this part of the trace buffer provides a broad spectrum of applications.

5. SOFTWARE FRONT-END

The former described enhancements of the FPGA design would be futile without matching enrichments of the software front-end. After a system reset the trace buffer unit is disarmed and internal configuration registers are programmed with standard values. Equally, all watchpoints are deactivated and filled with startup data, too.

It takes four steps to use the implemented trace enhancements:

- segmenting trace memory as desired and defining depth of pre-trigger fraction,
- setting up trigger behavior,

- arming trace buffer unit,
- reading out data from trace memory.

5.1 Configuring the Trace Buffer

Partitioning of the trace memory is guided by a dialogue box. Using a memory organization of 512 x 36 bits the number of recordable sessions is selectable by a combo box from one session with 512 rows up to 16 sessions, each with 32 code lines. If pre-trigger mode is selected, the division of pre- and post-trigger fragment is stepless variable. Naturally, in post-trigger mode the depth of the post-trigger fragment is equal to segment size.

Using the existing interface routines processed data can be uploaded into the trace buffer unit. Moreover data can be downloaded from FPGA to the front-end as well. This feature is essential for correct read-out of trace data.

5.2 Configuring the Watchpoints

Configuration of the watchpoints as depicted in section 3 is also guided by a dialogue box. Adequately to the two-layer architecture of the watchpoint hardware, the watchpoints are defined in two steps.

Each watch instance within the FPGA finds its counter-part in software and equal to the hardware combination layer there is an overall configuration window for combining the different events as shown in figure 5.

The 'Summary' section displays the settings of each configured event, in the lower sections event related settings may be adjusted. The structure of the 'Combine Events' section depends on the selected type of merging of events and the number of activated watch instances.

In figure 5 all three events are activated and cascaded. The value of the cycle counter described in section 3.2 is set by an additional slider.

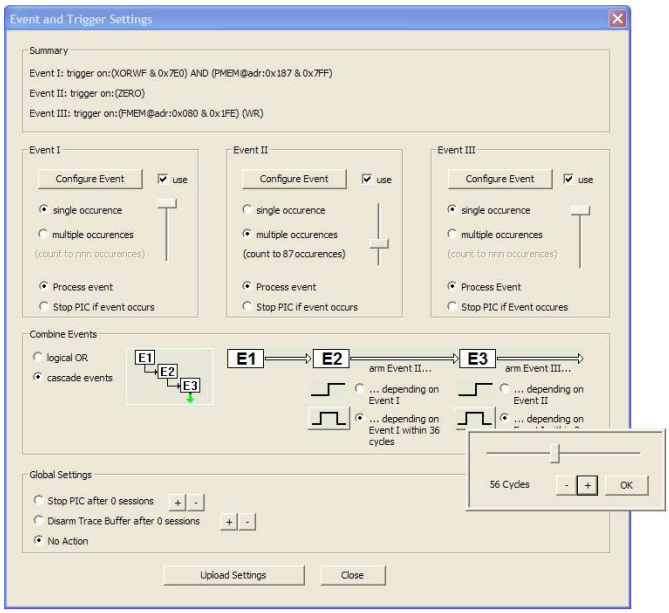

Figure 5. Watchpoint configuration dialogue.

Adjustments for stopping the processor or disarming the trace buffer can be made in the lowest section. If 'No Action' is selected the trace buffer will be recording sessions until the memory is totally filled.

5.3 Controlling the Trace Buffer

After the trace buffer is completely configured, all watchpoints are defined and the processor is running it is time to arm the trace buffer unit. Therefore a special dialogue box exists, where the trace buffer can be armed or disarmed and the processor can be stopped or started again. Within this dialogue both, the number of already written sessions and the total number of segments are displayed.

5.4 Read-out and Reconstruction of Data

After recording the desired number of sessions the memory data has to be read out. To assure the complete reconstruction of the traced program execution the recorded data has to be associated with the original hex formatted program file. Associating both sources yields the complete data set to display the executed program to the user as shown in figure 6 (see next page).

Starting at point of trigger the post-trigger fraction is reconstructed first. If there is also a pre-trigger fraction it is rebuilt by starting at point of trigger up to the first row in the data set. A second pre-trigger reconstruction cycle starting at the first row and ending at the point of trigger rebuilds the values of WORK register. Now, the user can easily browse through the recorded sessions and gets all relevant information at a glance. If desired all results may be saved to disk.

6. CONCLUSION

The necessity to debug real-time software like interrupt service routines led to the enhancement of an existing debugging system for soft-core SoC processors and controllers with hardware watchpoints and a trace buffer.

Each of the three freely definable watchpoints can monitor certain instructions or individual code lines or ranges and reading from and/or writing to arbitrary memory locations or ranges in RAM or special function registers. Furthermore they watch the ALU output for a special value/value range and the occurrence of an interrupt, a timer 0 overflow and a change of flags. The configuration dialogue allows the selection of a single incident of the above mentioned events or of nearly any combination of them. Additionally, multiple occurrences of the specified events can be set up. A watchpoint may be used alone or two or three can be cascaded to specify very specific sequences of events. The generated trigger signal will stop the processor like a breakpoint or will start the recording into an armed trace buffer.

The trace function uses a FPGA-integrated Block RAM to store all relevant information of processor internal events. The memory can flexibly be divided to permit single and multi-session recording i.e. from a single session with 512 code lines up to 16 sessions with 32 lines. The freely adjustable position of the trigger point within the respective buffer sections allows any kind of post- and pre-trigger mode. In combination with the three watchpoints single recordings or recording sequences in different buffers are possible.

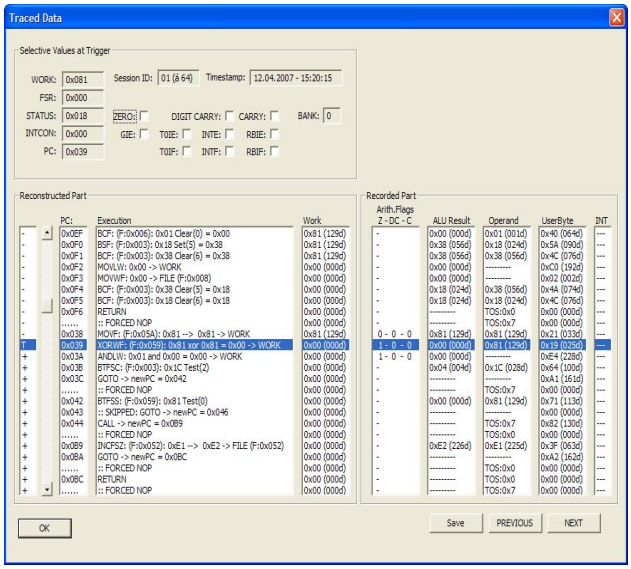

Figure 6. Trace buffer data presentation.

The trace buffer stores all relevant information which cannot be reconstructed. Additionally, the user may connect the input of a so-called user byte to specific points of interest in the design and have the data stored synchronously with the other information.

Watchpoints and trace buffer are configured and armed in configuration dialogues within the GUI of the existing debugger. The software front-end also reads the trace buffer sessions and reconstructs all derivable data. It presents the complete information to the user and allows the storage of it for later comparison etc.

The whole functionality is partitioned between hardware and software. Generally, everything which can be handled off-line is coded in software. All real-time features are implemented in the FPGA using VHDL. During the development implementations have been shifted from hardware into software and vice versa several times. Thus a balanced hardware-software co-design resulted. Though the number of watchpoints and the size of the trace buffer are fixed an adaptation may easily be accomplished. Also, the design should easily be adaptable to other processors because of is layered approach. Surely, the hardware interface will need the most changes but the front-end will need some modifications, too. Currently, a pipelined version of the VHDL controller is under development [6, 10] to which the debug module will be ported.

The flexible trace buffer in combination with the powerful watchpoints will ease code debugging dramatically especially with real-time software routines. Code debugging in FPGA-based microprocessors and microcontrollers can now be as convenient as with discrete standard parts using powerful in-circuit emulators.

7. REFERENCES

[1] Andreas, V. *Optimierung eines PIC-kompatiblen VHDL-Mikroprozessorkerns*. Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbuettel, Germany, 2004.

[2] ARM Ltd. *Embedded Trace Macrocell™ ETMv1.0 to ETMv3.4.* Architecture Specification. UK, 2006

[3] Bermbach, R. *Entwicklung eines Mikroprozessorkerns.* Research Report, University of Applied Sciences Braunschweig /Wolfenbuettel, Germany, 2003.

[4] Bermbach, R. *Entwicklung eines Mikroprozessorkerns – Bericht über die zweite Phase.* Research Report, University of Applied Sciences Braunschweig/Wolfenbuettel, Germany, 2004.

[5] Bermbach, R., Kupfer, M. Development of a Debug Module for a FPGA-based Microcontroller. *IFAC Workshop on Programmable Devices and Embedded Systems.* Brno, Czech Republic, 2006, 275-280.

[6] Bermbach, R. *Konzeption einer Pipelinearchitektur für einen FPGA-basierten Mikrocontroller – Bericht über die erste Phase im WS2006/2007.* Research Report, University of Applied Sciences Braunschweig/Wolfenbuettel, Germany, 2007.

[7] Cramm, I. *Entwicklung eines PIC-kompatiblen Mikrocontrollerkerns in VHDL.* Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbuettel, Germany, 2003.

[8] Germann, U. *Erweiterung eines Debug-Moduls für einen VHDL-basierten PIC-kompatiblen Mikrocontroller um Hardware Watches und einen Trace Buffer.* Diploma Thesis, University of Applied Sciences Braunschweig /Wolfenbuettel, Germany, 2007.

[9] Kupfer, M. *Entwicklung der Hard- und Software eines Debug-Moduls für einen VHDL-basierten Mikrocontrollerkern.* Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbuettel, Germany, 2005.

[10] Kupfer, M.; Bermbach, R.; Patz, R. Implementation of a Multi-stage Pipeline for a 8-bit Microcontroller. In *Proceedings of the 1st Research Student Workshop 2007.* University of Glamorgan, Faculty of Advanced Technology, Pontypridd (Wales), Great Britain, 2007, 109-110.

[11] Microchip Technology Inc. *PICmicro Mid-Range MCU Family Reference Manual.* Chandler, Arizona, 1997.