Tethered Forth system for FPGA applications

Paweł Goździkowski[a] and Wojciech M. Zabołotny[b]

[a]Institute of Computer Science, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

[b]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

ABSTRACT

This paper presents the tethered Forth system dedicated for testing and debugging of FPGA based electronic systems. Use of the Forth language allows to interactively develop and run complex testing or debugging routines. The solution is based on a small, 16-bit soft core CPU, used to implement the Forth Virtual Machine. Thanks to the use of the tethered Forth model it is possible to minimize usage of the internal RAM memory in the FPGA. The function of the intelligent terminal, which is an essential part of the tethered Forth system, may be fulfilled by the standard PC computer or by the smartphone. System is implemented in Python (the software for intelligent terminal), and in VHDL (the IP core for FPGA), so it can be easily ported to different hardware platforms. The connection between the terminal and FPGA may be established and disconnected many times without disturbing the state of the FPGA based system. The presented system has been verified in the hardware, and may be used as a tool for debugging, testing and even implementing of control algorithms for FPGA based systems.

Keywords: FPGA, Forth, CPU, Tethered Forth

1. INTRODUCTION

FPGA chips are widely used to build systems with complex functionalities. Control and testing of such systems often requires complex procedures, which are difficult to implement in standard state machines in FPGA, but can be easily implemented in a software executed by a CPU.

There are a few options to provide a FPGA based system with CPU functionalities The first one is to use an external microcontroller connected to the FPGA. However in this approach we either must use many FPGA pins to connect it to the CPU bus, or we have to use the serial connection, which impairs efficiency of communication between the CPU and the FPGA*.

Another possibility is to use an FPGA with embedded CPU core. This solution is available in a few FPGA families. Just to mention a few examples: the older ones - Altera Excalibur,[1] Xilinx Virtex 5,[2] and the new ones - Xilinx Zynq 7000,[3] and Altera Cyclone V[4] or Arria V.[5] Such embedded CPU offers perfect communication with synthesized IP cores in the FPGA (e.g. via the AMBA AXI[6] bus in the newer families). The only problem is that FPGAs with embedded CPU are the relatively expensive ones.

Finally, we can use the so called "soft core" CPU, which is described in HDL language, and synthesized just as other digital blocks in the FPGA chip. There exist many solutions of soft core CPUs, ranging from simple 8-bit (e.g. clones of the popular Z80[7]) to serious 32-bit CPUs with MMU (e.g. the OpenRisc 1200[8]) or even 64-bit CPUs like OpenSPARC T1.[9]

Main disadvantage of the "soft core" CPU is that it typically consumes significant amount of FPGA resources for the CPU logic and significant amount of memory for it's program. Therefore for our purpose we need a small, but efficient CPU.

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 7717

*There are available serial interfaces well suited for implementation of high speed bus, like e.g. PCI Express, but they are available only in relatively expensive FPGAs and CPUs, so we do not take them into consideration in this paper

Yet another factor affecting the choice of right solution is the fact, that testing and debugging often should be done interactively. The operator may need to create new testing procedures, on top of the previously defined ones, and to execute them without changing the state of the debugged system. In case of the typical CPU, where we must to write program, compile it, load to the CPU memory and start the program, it may be difficult to achieve.

So the right testing and debugging solution should be a system based on small but efficient CPU, able to work interactively. The perfect candidate for this purpose seems to be the Forth virtual machine running on a simple CPU.

2. FORTH LANGUAGE

The Forth language[10] in typical implementations offers good performance at moderate memory consumption, perfect extendability and possibility of interactive work. It may be implemented on a CPU with very limited instruction set. The Forth offers two modes of operation - the interactive mode and the compilation mode. In the interactive mode we can execute defined commands (which in Forth are called simply *words*). In the compilation mode we can define a new *word*, using previously defined ones. During one session we can switch between these two modes multiple times, alternately creating the program and testing it.

The Forth CPU must be able two handle two stacks - the data stack and the return stack. The data stack is the main place where the data are stored. The return stack stores the return addresses during subroutine calls (used e.g. when definition of one *word* uses some previously defined *words*). Another data structure essential for operation of the Forth system is the dictionary, which stores the names and definitions of *words*.

For most programmers Forth seems to be an exotic programming language, due to very specific syntax, related to the fact that when calling a word, the arguments must be pushed to the stack first and only then the appropriate *word* may be called. In practice it means, that the arithmetic calculations must be described in the Reverse Polish Notation (RPN).

Forth offers also *words* allowing to implement different loops and conditional instructions, which makes it a powerful environment for writing and executing programs on small systems.

2.1 Existing Forth solutions

There are many implementations of Forth available today. Probably one of the most popular is the **gforth**,[11] which may run on standard PC computers. From our point of view the most interesting projects are Forth implementations, which may run on small systems.

For example the AmForth[12] is able to run on small 8-bit AVR8 microcontrollers[13] produced by Atmel.[14] It may work even on very small ATmega168 microcontroller equipped with 16 KiB of FLASH and 1 KiB of RAM. By connecting a terminal (PC, smartphone or dedicated terminal) to a serial port of the AVR8 microcontroller we obtain the system allowing to develop and run Forth programs. There is also a possibility to define the special Forth *word*, which will be executed automatically, after the CPU starts. So the system may work autonomously, when no terminal is connected, and interactively, after connection of the terminal.

As we are looking for solutions suitable for a FPGA based system, it is worth to take a look at the J1,[15,16] which is implemented in FPGA, with source code consisting of only 200 lines of Verilog, and is able to execute up to 100 million Forth *words* per second. The CPU itself implements only five basic instructions (literal, jump, conditional jump, call, ALU operation), but some of them have a few variants (e.g. depending on the location of operands). J1 is a very interesting solution, however from our point of view it has one significant disadvantage – it doesn't offer possibility of interactive work. The program for J1 must be written and compiled in advance, and put into the program memory located in internal RAM in FPGA.

If we try to add the interactive work functionality to the FPGA based Forth system, we find, that *words* responsible for compilation are rather the complex ones and occupy a lot of space in the dictionary, and that the names of *words* also occupy significant amount of space in the dictionary. Problem of limited memory occurs also in the world of microcontrollers, and an example of successful solution of this problem may be a Riscy Pygness[17] – a Forth implementation for ARM microprocessors. To avoid the mentioned problems, Riscy Pygness uses so called "tethered Forth model", described in the next section.

In our work we attempt to combine advantages of small, specialized CPU implemented in FPGA and of the tethered Forth model.

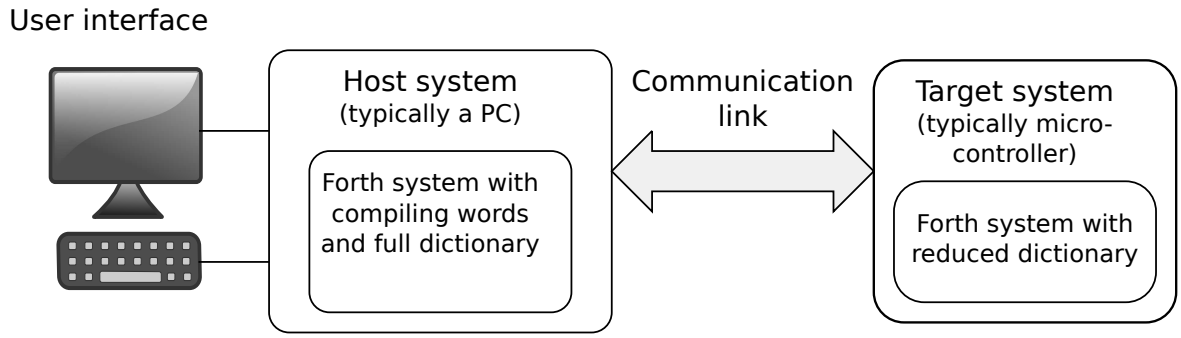

Figure 1. The structure of the system based on the tethered Forth model

3. TETHERED FORTH MODEL

The Tethered Forth[18] approach is dedicated for small target systems, where memory consumption for compilation mode *words* and for full dictionary is unacceptable. We can reduce the memory footprint in the target system, by splitting our Forth system into two parts (see Fig.1):

- The **host** part runs on a bigger system (typically a PC computer). It stores the full version of the dictionary, and performs all compilation activities.
- The **target** part is located in the target microcontroller. It stores the reduced version of the dictionary, containing only the compiled code. The **target** part is able to execute defined words locally. It is also able to communicate with the **host** part, performing on request at least such actions, as "read data from memory", "write data to memory", "start execution from the particular address".
- The communication module providing bidirectional link between the **host** and **target** parts. In most implementations it is a serial interface, at is available even in small microcontrollers and it can be handled with very small software overhead.

3.1 Tethered Forth approach with the FPGA based system

One of assumptions in section 1 was, that we want to avoid use of additional compiler. Doesn't the tethered approach violate this assumption? The main reason to avoid using a compiled language, was the fact that when we reload the target system with a new version of the program, we lose the state of the system stored in program internal variables. This problem doesn't occur in the tethered Forth model, as the variables are stored in the memory of the FPGA based **target** system. Another complaint may be that when using the tethered approach, we don't have a Forth system which itself is capable to work interactively. For interactive operation the **host** part of the system is necessary. However our FPGA based system anyway requires to be connected to the terminal to work interactively, and today a typical terminal is either a PC computer, or at least a smartphone. Considering the above, we can state, that the requirement to have a **host** part of our Forth system installed in our terminal is not a significant disadvantage. It is only necessary to have the **host** part implemented in a portable way, so that it can be used on many "smart terminal" platforms. Therefore in our implementation we have focused on providing the portable implementation of the **host** part.

4. IMPLEMENTATION OF THE PROPOSED SYSTEM

Implementation of the system consists of three parts: the **host** part, the **target** part and the **communication link**, as described in section 3.

Table 1. The instruction set of the CPU.

Command mnemonic	code	Description of the command
One word instructions		
PUSHD	0x01	Push the register GX_REG to the data stack
PUSHR	0x02	Push the register ADX_REG to the return stack
POPD	0x03	Pop the value from the top of the data stack and put it into the register FX_REG
POPR	0x04	Pop the value from the top of the data stack and put it into the register ADX_REG
RET	0x06	Return from the subroutine
ADD	0x09	One operand of ADD instruction is in GX_REG, another is in FX_REG, results goes to GX_REG
SUB	0x0A	One operand of SUB instruction is in GX_REG, another is in FX_REG, results goes to GX_REG
MUL	0x0B	One operand of MUL instruction is in GX_REG, another is in FX_REG, results goes to GX_REG
EQ	0x0C	Check if the value in GX_REG is equal to 0
STORE	0x0D	Store the value from GX_REG to the memory address from ADX_REG
LOAD	0x0E	Load the value from the memory address stored in ADX_REG to GX_REG
OVER	0x0F	Swap the two topmost values on the stack
EMIT	0x11	Send the character with ASCII code stored in GX_REG the host system for displaying
EXEC_PC	0x12	Execute the word identified by PC address (this address is assign by PC after operator creation) on the host system
GT	0x13	Compare two values. First value is in GX_REG, another value is in FX_REG. Results goes to GX_REG
LEDIO	0x14	Set the external pin (currently only LEDs are connected for testing purposes) according to GX_REG register
Two word instructions		
CALL	0x05	Call the subroutine at address in ADX_REG
LOADI	0x07	Load the immediate value to GX_REG
JMP0	0x10	Jump if GX_REG value is zero then jump is performed else next instruction executes
Three words instructions		
MOV	0x08	Move the value from register which is specified by two least significant bits to register specified by another two bits

4.1 Implementation of the target part

The target part is implemented in the VHDL language. It consists of two main state machines - the CPU and the monitor.

The CPU is a small 16-bit processor. It offers instruction set suited for execution of Forth language (see Table 1). To support the tethered model, the CPU has special instruction (EXEC_PC) to execute operator in the **host** part of the system. The CPU is implemented according to the MISC (Minimal Instruction Set Architecture) paradigm – only the necessary instructions are supported, and more complex actions are implemented as Forth words. CPU executes instructions compiled and stored in the internal FPGA RAM memory. The RAM memory is divided into two stacks and the dictionary. The return stack grows towards lower addresses, and the data stack grows towards higher addresses. To simplify stack operations, the CPU uses two stack pointers for the data stack. The return stack is handled with a single stack pointer.

CPU has following registers:

- GX_REG - it is a destination of most arithmetical operations, and a source argument for the STORE operation
- FX_REG - it is the second argument for arithmetical operations
- ADX_REG - it is a source address register
- PC_REG - keeps currently executed instruction
- SP_REG - keeps address of the top of data stack
- RP_REG - keeps address of the top return stack

Only GX_REG, FX_REG, ADX_REG are visible for programmer, which means that only these registers can be used as source and destination for operations. CPU does not handle instruction pipeline. Each instruction is executed one after another.
The instruction cycle consists of following operations:

- FETCH - during this phase of the cycle the instruction is fetched from memory.
- DECODE - during this phase the instruction is recognized by opcode and arguments are prepared for execution
- EXECUTE - in this phase the instruction is performed. In case of more complicated instruction this phase may require a few clock cycles (for example two word instructions require two accesses to memory)

The Monitor is a simple state machine which controls operation of the CPU, according to commands received from the communication link. It may start execution of the compiled words, it may also take over control of the memory bus and read or write data words from or to this memory.

4.2 Implementation of the host part

The main goal when implementing the **host** part was to ensure the portability, so that not only the PC computer, but also a smartphone may be used as an "intelligent terminal" allowing to use the system in interactive mode.

Looking for the proper language to implement the **host** part, we were considering Java and Python. Both are available for different operating systems for PC computers, and both can be used on most smartphone platforms. Java is the native solution for such mobile platforms as Java ME and Android[†]. Python can be run on Android smartphones, using the additional Scripting Layer for Android[19] (SL4A) extension.

Both languages offer automatic memory management and support data structures needed to effectively manage the Forth dictionary

As the Python based solution is easier to modify (the program is distributed in the source form, and no recompilation is needed), we have finally decided to implement the **host** part in Python.

[†]In fact Android uses a special virtual machine - Dalvik, which is not fully Java compatible, but on the source level we can write applications for Android in Java

Table 2. Implementation of three basic Forth *words*: ROT, DUP and SWAP in the CPU assembler.

```
: ROT ASSEMBLER
  POPD
  MOV ADX_REG , FX_REG
  POPD
  MOV GX_REG , FX_REG
  POPD
  PUSHD
  MOV GX_REG , ADX_REG
  PUSHD
  MOV GX_REG , FX_REG
  PUSHD
;
```

```
: DUP ASSEMBLER
  POPD
  MOV GX_REG , FX_REG
  PUSHD
  PUSHD
;
```

```
: SWAP ASSEMBLER
  POPD
  MOV GX_REG , FX_REG
  POPD
  PUSHD
  MOV GX_REG , FX_REG
  PUSHD
;
```

The **host** part implements a simple x86-like assembler supporting basic CPU instructions listed in the Table 1. Additionally it implements also the Forth interpreter, able to execute the compilation *words* in the host system, and execute other *words* in the target system.

The interpreter can use two types of input: file input or console input. The first block of the interpreter is a scanner, which reads words from input, deletes comments and returns lexems ready for use by the parser. The parser block is not complicated because each *word* in Forth simply executes certain operation.
For example execution of the colon operator changes state of interpreter to the compile mode and assumes that the next word is the name of operator to be created. The scanner and parser are running in the **host** part of system. The main loop of the interpreter performs following actions:

- Get lexem from the scanner.
- Check the state of the interpreter and control bits of operators: If interpreter is in interactive mode, find the byte code corresponding to the lexem in the dictionary. If interpreter works in compile mode, check if the lexem corresponds to the operator existing in the dictionary and if this operator is immediate. If operator is immediate then execute it, else translate it to the call operation.
- Send produced byte code to the **target** system.

The third part of the system is the implementation of the basic Forth *words* in the assembler. Example implementations of three basic *words* is shown in the Table 2.

4.3 Implementation of the communication link

The communication link is implemented both on the **host** side and on the **target** side.

The **host** side implementation consists of the hardware part (which in case of the PC computer is an USB-UART adapter, and in case of the smartphone a Bluetooth-UART adapter), and the software part. The software for PC part is based on the Python **serial** module, but it should be relatively easy to port it to Bluetooth libraries for the smartphone version.

The **target** side implementation is based on the *smalluart.vhd* UART implementation published as a part of the fpgadbg project.[20]

The main tasks of the communication link module are:

- In the transmitter part – to encapsulate commands exchanged between the **host** and the **target** parts of the system into records which may be transmitted via the UART interface.
- In the receiver part to decode received records producing the commands, and to pass them to the monitor module for execution.

4.4 Suspending and resuming of the interactive session

If our system is supposed to be used for control and debugging of the FPGA based electronic systems, it is possible, that there may be a need to stop the interactive session, and disconnect the intelligent terminal (host) for some time. Afterwards it should be possible to reconnect the terminal and resume the session. In fact, it is desirable, that the operator should be able to start execution of the longer procedure, disconnect the terminal, and after some time reconnect it and check the results in the resumed session.

In case of standard Forth systems, where terminal does not store any information about the state of the system, such operation may be performed without any special precautions. In the tethered Forth system however, it is necessary to verify that the state of the **target** system still corresponds to the state of the **host** part of the system.

To implement the described functionality, the **host** part of the system stores the content of the reduced **target** dictionary, and during reconnection checks if the **target** dictionary has changed. If not, the session may be resumed. Otherwise it must be assumed, that the **target** dictionary is modified in an unknown way, and the terminal asks the user if he wants to start a new session with initial content of the dictionary. The user must clearly confirm the decision to start a new session, to avoid the situation when he connects to the system previously used by someone else and reinitializes it, destroying the work of the previous user.

In the future versions of the system it should be possible to store the state of the system, consisting of both: reduced **target** dictionary and full **host** dictionary, in the central database on the server. This should allow to use alternately different terminals to interact with the system. After each session the terminal should update the dictionaries stored in the database, and during connecting, the terminal should download current state of both dictionaries. Of course even in this situation it is necessary to verify, that the **target** dictionary is identical with the stored one, and otherwise propose to start a new session.

5. RESULTS

The implemented system has been tested in the simulated environment and in the real hardware

In the simulated environment the **target** part was implemented as a behavioral model of the FPGA simulated in the GHDL[21] simulator. The **host** part was running on standard PC. Both parts were connected using the simulated UART module[22] developed for dedicated hardware-software cosimulation environment,[23] and implemented partially in VHDL and partially in C, using the VPI interface.

The tests in real hardware were performed using the Xilinx Spartan-3A Starter Kit[24] board. The compiled FPGA part of the Forth system occupied 23% of the logic slices (1368 slices from 5888 available) and 40% of internal memory (8 RAMB16BWE blocks from 20 available) in the xc3s700an chip. The CPU was successfully compiled for 50 MHz clock frequency.

Performed test have shown, that the system operates correctly both with simulated hardware and with the real hardware. It was possible to work interactively, to define new *words* and to execute previously defined *words*.

6. CONCLUSIONS

Presented solution allows to implement the “soft Core” Forth CPU in the FPGA based system. To minimize consumption of logic resources and of internal memory, the solution is based on the Tethered Forth model, in which the full Forth dictionary is stored in the intelligent terminal, and the compilation *words* are executed by the intelligent terminal. Only the compiled *words* are stored in and executed in the target system.

Proposed system may be an useful tool for testing and debugging of FPGA based electronic systems, allowing to develop and run control and debug programs in interactive way typical for Forth system.

Design of the CPU allows to easily add extensions dedicated for communication with the particular hardware. Further work should be focused on optimization of the CPU design and its integration with typical FPGA internal buses like Wishbone.[25]

REFERENCES

[1] "About excalibur embedded processor solutions." http://www.altera.com/products/devices/excalibur/exc-index.html (June 2013). [Online; accessed 29-June-2013].

[2] "Virtex-5 family overview." http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf (February 2009). [Online; accessed 29-June-2013].

[3] "Zynq-7000 all programmable soc overview." http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (March 2013). [Online; accessed 29-June-2013].

[4] "Cyclone v device overview." http://www.altera.com/literature/hb/cyclone-v/cv_51001.pdf (May 2013). [Online; accessed 29-June-2013].

[5] "Arria v device overview." http://www.altera.com/literature/hb/arria-v/av_51001.pdf (May 2013). [Online; accessed 29-June-2013].

[6] "Amba specifications." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html (2010). [Online; accessed 29-June-2013].

[7] "Wishbone high performance z80." http://opencores.org/project,wb_z80 (June 2012). [Online; accessed 29-June-2013].

[8] "Or1200 openrisc processor." http://opencores.org/or1k/OR1200_OpenRISC_Processor (December 2012). [Online; accessed 29-June-2013].

[9] "Opensparc t1." http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html (June 2013). [Online; accessed 29-June-2013].

[10] Brodie, L., "Thinking forth." http://kent.dl.sourceforge.net/project/thinking-forth/reprint/rel-1.0/thinking-forth.pdf (2004). [Online; accessed 29-June-2013].

[11] "Gforth home page." http://www.gnu.org/software/gforth/ (March 2013). [Online; accessed 29-June-2013].

[12] "AmForth." http://amforth.sourceforge.net/ (June 2013). [Online; accessed 29-June-2013].

[13] "AVR 8-bit and 32-bit Microcontroller." http://www.atmel.com/products/microcontrollers/avr/default.aspx (June 2013). [Online; accessed 29-June-2013].

[14] "Atmel website." http://www.atmel.com/ (June 2013). [Online; accessed 29-June-2013].

[15] Bowman, J., "The J1 Forth CPU." http://excamera.com/sphinx/fpga-j1.html (October 2012). [Online; accessed 29-June-2013].

[16] Bowman, J., "J1: a small Forth CPU core for FPGAs," 43–46.

[17] "Riscy Pygness – Pygmy Forth for the ARM." http://www.utoh.org/riscy/ (October 2011). [Online; accessed 29-June-2013].

[18] Martinez, H., "Developing a tethered forth model," *ACM SIGFORTH Newsletter* **2**(3), 17–19 (1991). http://www.odysci.com/article/1010113016014389.

[19] "android-scripting – scripting layer for android brings scripting languages to android." http://code.google.com/p/android-scripting/ (June 2013). [Online; accessed 29-June-2013].

[20] Zabolotny, W. M., "Fpgadbg - a tool for FPGA debugging." http://www.ise.pw.edu.pl/~wzab/fpgadbg/ (September 2009). [Online; accessed 29-June-2013].

[21] Gingold, T., "Ghdl - where vhdl meets gcc." http://ghdl.free.fr/ (2010). [Online; accessed 29-June-2013].

[22] Zabolotny, W. M., "Pseudo UART allowing to connect via pseudoterminal to GHDL simulated IP core." http://ftp.funet.fi/pub/archive/alt.sources/2618.gz (2011).

[23] Zabołotny, W. M., "Development of embedded pc and fpga based systems with virtual hardware," *Proceedings SPIE* **8454**, 84540S–84540S–7 (2012).

[24] "Spartan-3A Starter Kit." http://www.xilinx.com/products/boards-and-kits/HW-SPAR3A-SK-UNI-G.htm (June 2013). [Online; accessed 29-June-2013].

[25] "Soc interconnection: Wishbone." http://opencores.org/opencores,wishbone (2013). [Online; accessed 29-June-2013].