Elektrorevue
2006/30 -
14.7.2006

An FPGA Microcontroller Design

Contents

Ing. Jakub Šťastný, Ph.D.
Petr Bílý
Biosignal Processing Laboratory, FPGA Laboratory, Department of Circuit Theory, Faculty of Electrotechnical Engineering
Czech Technical University in Prague
Technická 2, 166 27 Praha 6
stastnj1@seznam.cz, petrbily@volny.cz
http://amber.feld.cvut.cz/fpga
http://amber.feld.cvut.cz/bio

ČESKÁ VERZE

This contribution describes the design of a simple FPGA microcontroller. The microcontroller was intended to be used as a teaching example (case study) as well as a real control block for FPGA systems designed in our laboratory. Despite of the very simple structure of the resulting system we designed a block which might be used for both intended purposes.

1 Introduction

We were faced with the need to make our students familiar with the fundamentals of the control (processor) logic design in the frame of our FPGA DSP design teaching. The original idea was to reuse an existing simple processor/controller design (see [1-8] or the famous DLX architecture) and to demonstrate the basic design principles with it. Unfortunatelly, the available designs and systems were targeted above all to the computer science teaching above all and were far too complex for our purposes. We needed a microprogrammable controller with an easily and fast understandable architecture suitably demonstrating design fundamentals of controllers, with as small and as simple as possible design which is easily extendable. Further it was necessary to get a system which is easily simulable and possible to be implemented with the free FGPA design tools (ModelSim XE, ISE WebPack) having limitations in sizes of available programmable logic devices as well as design (simulator is drastically slowed down when simulating larger designs). Finally, we needed to get - despite of the design simplicity - an IP controller macro usable in real-world applications (we have been developing a DSP speech processing system right away which will use this macro). Thanks

to all these facts we finally decided to implement our own IP macro. Its description follows.
The text is structured along to the standard phases of a digital circuit design implementation (DCT - Design, Code and Test). The system specification is introduced at first, the RTL design, verification and FPGA implementation are subjects of the following text. Possible teaching applications are mentioned in the end. The design process itself was thoroughly documented and described to serve as a typical example of a control system design process in our teaching.

2 Programmer's model

The first step in the programmable control system design is to define the set of supported functions. Based on this the microinstruction set architecture is to be designed. The microinstruction set design is the factor determining the datapath architecture of the final system. The datapath itself further constrains the control circuit design of the microcontroller and microinstruction decoder architecture.
We defined the basic system properties as follows:

1. microprogram memory shall contain 32 instructions (5bit microprogram counter),
2. if the microprogram counter overflows over 31, it shall be set to 0,
3. reset shall set the program counter to zero,
4. the microcontroller shall have 6 input condition signals controlling the microinstruction execution (predicated microinstructions), signals are asynchronous and shall be synchronized inside the controller,
5. the microcontroller shall have 4 output 8 bit ports controlled by WRITE instruction. Each of the ports has its own output register (i.e. the output value is stable until it is changed by executed microprogram),
6. every instruction is executed in two clock cycles - the first cycle is a fetch one (FETCH phase), the second execution one (EXECUTION phase),
7. the microcontroller shall execute JUMP instruction (jump to the address stored in the instruction), NOP instrucion (ignored operation) and WRITE instruction (write to the output port).

Based on this specified properties we designed the microinstruction format itself taking also in account the possibility to do an easy implementation of the microoperation code decoding and simultaneously we tried to left there some space for either further extension of the microinstruction set or for the change of "architectonic constants" - microprogram memory enlargement, etc. Owing to all these reasons we chose 16bit microinstruction format:

bit no.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
content	CV	Condition			opCode		Instruction parameters									

The description of the microinstruction fields follows:

- opCode - microoperation code: 01 - WRITE, 00 - JUMP,
- Condition - three bit number of the condition signal; conditions 1-6 are read from the microcontroller inputs cond_1 - cond_6, condition 0 is stuck at 0, condition 7 is stuck at 1. For the system function is sufficient only one constant conditional signal; the condition 7 is in fact reserved for future use by students for some purpose,
- CV - logical value of the condition required to be evaluated to execute the microinstruction,
- microinstruction parameters - depending on the microinstruction type:
    - JUMP: parameters are of value 00000XXXXX, where XXXXX is a 5 bit microinstruction address where to jump,
    - WRITE: AADDDDDDDD, where AA is a two-bit address of the output port and DDDDDDDD are data to be written there,
- NOP is any instruction disabled by its condition,
- if the microcontroller reads an undefined instruction, its behaviour is undefined.

A simple microprogram example along with the binary opcodes follows:
Address Opcode Instruction
```
01234  0123456789ABCDEF
```

```
00000  0000011110100101   IF C0=0 WRITE 0xA5 TO 3
```
C0 is stuck at zero, this instruction is always executed.
```
00001  0001000000000001   IF C1=0 JUMP TO 00001b
```
Wait until C1 is zero.
```
00010  0000011101011010   IF C0=0 WRITE 0x5A TO 3
00011  1001000000000011   IF C1=1 JUMP TO 00011b
```
Wait until C1 is not zero.
```
00100  0000000000000000   IF C0=0 JUMP TO 00000b
```
Go back to the beginning of the program.

The specification of the designed system must be completed with the core input/output ports list along with their functions:

Signal	Direction	Type	Description
clk	in	std_logic	clock signal, all registers react to the rising edge
res_n	in	std_logic	global asynchronous reset, active in 0, shall be already synchronised to the clock rising edge (to avoid reset recovery violation - see below)
cond_1	in	std_logic	condition no. 1, asynchronous input
cond_2	in	std_logic	condition no. 2, asynchronous input
cond_3	in	std_logic	condition no. 3, asynchronous input
cond_4	in	std_logic	condition no. 4, asynchronous input
cond_5	in	std_logic	condition no. 5, asynchronous input
cond_6	in	std_logic	condition no. 6, asynchronous input
reg_0	out	std_logic_vector (7 DOWNTO 0)	output register 0, reset zeroes it (reset state 00000000)
reg_1	out	std_logic_vector (7 DOWNTO 0)	output register 1, reset zeroes it (reset state 00000000)
reg_2	out	std_logic_vector (7 DOWNTO 0)	output register 2, reset zeroes it (reset state 00000000)
reg_3	out	std_logic_vector (7 DOWNTO 0)	output register 3, reset zeroes it (reset state 00000000)

3 RTL system design

RTL system design includes datapath, control unit and I/O subsystem design made on a sufficiently high level of abstraction. Before we start implementing the system in VHDL we have to thoroughly analyze the structure of the prospective system. The first step is to consider the character of all the input signals:

- **reset** is an asynchronous input expected to be synchronized outside our design. If the designer using our system does not synchronize the reset correctly, the system itself might not be properly initialized (more on reset, its synchronization and proper usage is in [9]),
- **input asynchronous condition signals** might change its value anytime and thus violate the setup and hold conditions at the internal flip-flop inputs. Owing to this we synchronize them inside our design. The synchronization is done by means of two D-flip flops in serial. During reset are both flip-flops reset, after the controller startup the programmer has to count with the fact that the first instruction executed after reset will not be able to react on the input conditions.

Next step is to define the reset state of the system: the microprogram counter is set to zero, the controller enters the FETCH state, instruction register is reset and all output ports are reset.
Figure 1 shows the final designed structure of the system.

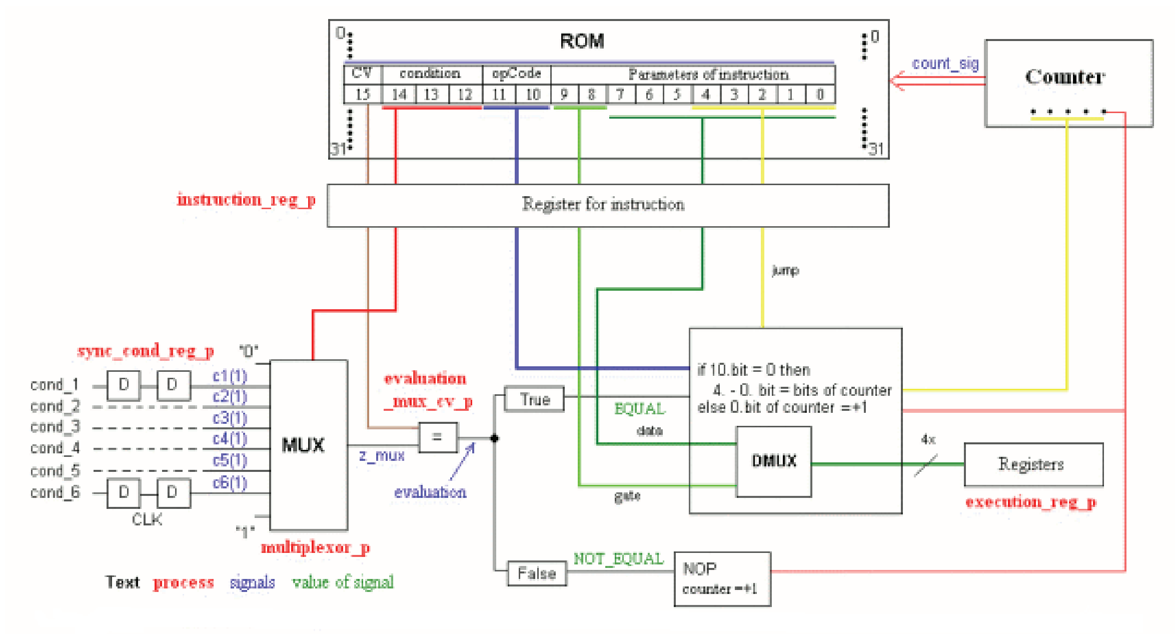

Figure 1: RTL schema of the microcontroller.

The descriptions of the single blocks (are these blocks are in separated processes in the VHDL code) of the microcontroller design are the following:

- **clock_gen_p**: the phases of instruction processing are generated here, any rising clock edge toggles between FETCH and EXECUTE state. The signal carries an information on the microcontroller current state,
- **pc_reg_p**, **next_pc_p** (Counter block in Fig. 1): count_sig (5 bits) signal carries the current microinstruction counter output address. The signal value is incremented with the clk rising edge when the system is in the EXECUTE state and we process other microinstruction than JUMP. If the JUMP is executed and the condition in the microinstruction is valid, the counter address is load from the microprogram ROM memory (jump is executed),
- **instruction_reg_p**: instruction register - in the FETCH state the microinstruction operating code read from ROM is stored here. Microinstruction address is taken from the count_sig bus,
- **sync_cond_reg_p**: asynchronous input conditional signals cond_1 - cond_6 are synchronized here. The synchronization is done in two clock cycles (see helper two-bit signals c1-c6),
- **multiplexor_p**: condition multiplexing controlled by the information stored in the microinstruction is done here. The output signal z_mux carries the value of the input condition selected by the microinstruction,
- **evaluation_mux_cv_p**: microinstruction condition is evaluated here. If the CV microinstruction bit (no. 15) and z_mux signal are of the same value, evaluation block output is set to EQUAL (conditional bit = real value of the condition, microinstruction is executed). The evaluation output is set to NOT_EQUAL otherwise (conditions are not equal, instruction is discarded). The signal value is used only in the EXECUTE phase,
- **execution_reg_p**: if we execute WRITE microinstruction and the microinstruction condition is fulfilled, the appropriate output register is updated with the clk rising edge here.

The IP block was designed in the VHDL language. The source code are enclosed in the attached file. The integral part of the design is also a VHDL package defining microinstruction symbolic operation codes. These constants make the microprogramming signifcantly easier and faster. An example of the microprogram memory filled with the microcode already used for demonstration above follows:

Address Op.Code Microinstruction

```
01234   0123456789ABCDEF
```

```
00000  0000011110100101   IF C0=0 WRITE 0xA5 TO 3
VHDL code: IF_C0_EQ_0 & THEN_WRITE & TO_REG_0 & "10100101"
00001  0001000000000001   IF C1=0 JUMP TO 00001b
VHDL code: IF_C1_EQ_0 & THEN_JUMP & "0000000001"
00010  0000011101011010   IF C0=0 WRITE 0x5A TO 3
VHDL code: IF_C0_EQ_0 & THEN_WRITE & TO_REG_3 & "01011010"
00011  1001000000000011   IF C1=1 JUMP TO 00011b
VHDL code: IF_C1_EQ_1 & THEN_JUMP & "0000000011"
00100  0000000000000000   IF C0=0 JUMP TO 00000b
VHDL code: IF_C0_EQ_0 & THEN_JUMP & "0000000000"
```

4 Design Verification

Owing to the simplicity of the whole system and to the fact only one clock domain is used in the design we decided to do only RTL (Register Transfer Level) verification. Anyway, the limits on the simulation speed built in the ModelSim XE (a digital simulator freely available to students) do not allow students to do a complete gate-level verification. We considered this fact already in the design phase and designed the whole microcontroller in a way minimizing the risk of timing problems after place and route.

The first step in the verification process is always writing of the verification plan - list of requirements which shall be fulfilled by the design. The requirements extracted from the system specification are as follows:

1. microprogram counter is reset if it overflows over 31,
2. reset resets the counter and output registers,
3. if the zeroth bit of the microinstruction and the conditional signal have the same value, the microinstruction shall be executed,
4. if the zeroth bit of the microinstruction and the conditional signal do not have the same value, the microinstruction shall not be executed,
5. the instruction execution shall be conditioned by the very condition which number is in the microinstruction code (checking of this requirement verifies the multiplexing of the conditions),
6. the 0th condition shall be always at zero,
7. the 7th condition shall be always at one,
8. if the WRITE microinstruction is executed and its condition is fulfilled, the defined word shall be written to the defined output port, other output ports remain unchanged, program counter advances to the next microinstruction afterwards,
9. if the JUMP microinstruction is executed and its condition is fulfilled, the microcontroller jump to the defined address, all output ports remain unchanged,
10. if the microinstruction condition is not fulfilled, then the instruction shall be ignored (NOP), no output port shall be changed, the microinstruction counter shall be advanced to the next microinstruction.

In order to verify the proper function of the whole system, we filled the microprogram ROM with a verification microprogram (see microcontroller source code, ROM with microprogram content). The verification microprogram operations verify the proper function of the system in cooperation with input stimuli - the appropriate scenarios to verify the single points from the verification plan are created and the microcontroller responses are checked. To generate the stimuli we had to design the model of the "surrounding world" generating stimuli and verifying responses - testbench. The top level testbench diagram is drawn in Figure 2. Just one command ("run the simulation" in the simulator) is required to execute the whole simulation; if everything is OK, the simulation is closed with a message "End of simulation, tests OK!". If there is something wrong, the simulation is also stopped and the reason why it was stopped is printed either. The whole verification is fully automatic, self-checking, runs without a human's intervention. Students themselves might try to verify the items of the verification plan by hand (e.g. using the force command of the simulator + checking of the proper responses by examining the waves) to see the clear advantages of the self-checking testbench even in such a simple case.

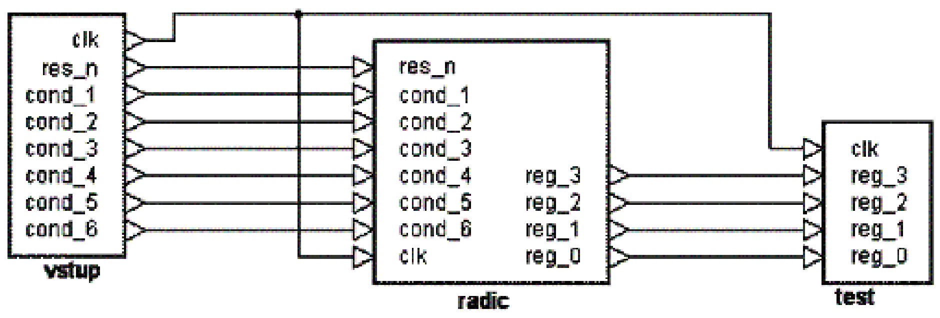

Figure 2: The testbench is built up from the three components. The first component (named vstup in the source codes) generates the stimuli, the second (radic) is the microcontroller itself and the third (test) checks the controller responses.

4.1 Code coverage

The verification is surely the least loved phase of the digital integrated circuits. Despite this it is a very important design phase - its correctness and completness are crucial for the success of the whole design. Digital simulators offer a lot of tools to facilitate the verification task and to allow faster and easier finding possible design problems. The code coverage measurement is one of these tools. If the verification has to find all the bugs in the design, it is inevitable to execute all the lines of the VHDL source code (any line of code shall be executed at least once). The execution of all the code lines itself does not ensure the code is bug-free; it only gives us the information that all the code was executed and there is no "white" uncovered space in the code which might contain bugs.

Owing to this fact we also included code coverage measurement into our verification process. Scripts *compile_design_cover.do* and *testradice_sim_coverage.do* in the RTL code package executes the simulation with the code coverage measurement (ModelSim SE was used for this; we have this tool in our laboratory). Simulator gave us the following one line of the code uncovered (marked in red, line no. 250):

```
244    1    61       if state=EXECUTE then
245    1    31         case instruction(9 downto 8) is
246    1     4           when to_reg_0 => reg_0 <= instruction(7 downto 0);
247    1    17           when to_reg_1 => reg_1 <= instruction(7 downto 0);
248    1     5           when to_reg_2 => reg_2 <= instruction(7 downto 0);
249    1     5           when to_reg_3 => reg_3 <= instruction(7 downto 0);
250    1     0           when others => NULL;
251                    end case;
```

This result is 100% OK - the uncovered line is the mandatory CASE construction; the line is never executed (all the allowed combinations are covered with constants).

Let us note that the code coverage is only one tool of the whole family - the simulator also allows to measure the conditional coverage, branch coverage and toggle coverage. The reduced "student" version of ModelSim XE does not support any of these; ModelSim SE installed in our laboratories had to be used.

5 FPGA Implementation

The VHDL design was synthesized, placed and routed on FPGA Spartan 2, XC2S15cs144-6 (this FPGA is availabe in the ISE WebPack available to students for free). The final design parameters are lister below in the table:

Logic Element	Design	Available on FPGA	Utilization

DFFs	62	384	7%
LUTs	53	384	13%
IO lines	39	86	45%

Static Timing Analysis tool gave us the maximum system clock frequency of 117.47 MHz. However, it shall be emphasized here that in this case the maximum clock frequency is dependent not only on the design structure but also on the microprogram memory content. Synthesizer analyzes the microprogram during the synthesis and optimizes the whole system according to this - e.g. output port bits unused by the microprogram are replaced by logic constants. Owing to the same reason the design size is also slightly dependent on the microprogram used. The size mentioned above was stated with the verification microcode.

6 System architecture optimization

The designed system might be also used to show the impact of some basic design optimization approaches usable not only for control structures design but applicable on a arbitrary FPGA system designs [10]. The implemented system might be optimized in two basic ways - for speed (the target is as short clock period as possible) and for area (the target is as small system as possible). Both kinds of optimization might be applied on various abstraction levels - on the system architecture level, RTL and gate level. The following table summarizes all the optimization approaches applicable here; the single methods are discussed then.

Criterion	**Level of abstraction**	**Optimization method**
speed	system architecture	pipelining of conditions processing
area	RTL	distributed ROM replacement by blockRAM
area	RTL	resynchronization registers replaced by SRL cells
area	gate level	synthesis + place and route optimization for area
speed	gate level	synthesis + place and route optimization for speed

6.1 Pipelining

The critical design combinatorial path (the slowest combinatorial path in the design) starts at the output of any pair of condition resynchronization registers, goes through the condition evaluation logic, write control logic and ends in the output port registers. The slow combinatorial path might be easily broken into two shorter and faster ones by registering the evaluation signal (see Figure 1, the small arrow near the evaluation label). If we do this, we have to take it into account when programming the system as the currently executed microinstruction is always executed with the condition referenced in the previous microinstruction. Special attention must be paid to the conditioned JUMP microinstruction execution and usage - the condition in the JUMP microinstruction either influences the following microinstruction (when the JUMP is not taken), or the target microinstruction (when the JUMP is taken). Due to this we have to do one of the following things if we adopt this approach:

1. the first possibility is to write the microprogam with this in mind and shift the condtions in the microinstructions so as the condition for the microinstruction at the address i was a part of the microinstruction at the i-1 address. Futher we have to take care of uncodnitional JUMP instructions (target instruction has its condition in the JUMP instruction) and pay attention to the conditional JUMPs (and be very carefull when chaining JUMPs - conditional JUMPing to conditional JUMP instruction is tricky now),
2. the second possiblity is to split the microprogram ROM into the ROM with conditions and ROM with the rest of the microinstructions and add a logic to prefetch the condition in advance. Then we have again a small problem with conditional JUMP, as the target address for condition prefetching is not known until the JUMP instruction condition is evaluated. The additional logic has to insert a prefetch phase between the FETCH and EXECUTE phases then to prefetch the next instruction condition. This slows down the conditional JUMP execution when

the conditional JUMP is taken (we got sometimes three clock cycles long microinstruction execution).

The pipelining allows us to increase the system clock frequency up to 170 MHz (45% performance boost). Let us note that the mentioned problem with conditional execution is a common problem of modern processors with pipeline [11].

6.2 BlockRAM Usage

The microinstruction register connected to the microinstruction ROM is described as a register with an asynchronous reset. This implementation leads to the distributed RAM built up from LUTs followed by a standard register after synthesis [10,12]. If the instruction register has only a synchronous reset, the synthesis uses the blockRAM in the FPGA fabric. The advantage of this approach is that we spare some LUTs which might be used for other logic functions afterward. The disadvantage is the performance loss - asynchronous memory is faster than blockRAM.

6.3 Resynchronization registers replaced by SRL cells

The input conditions are resynchronized in a standard resynchronization register - a shift register with two DFFs. All the Xilinx FGPA platforms are able to configure the LUT as a shift register (SRL, Shift-Right Look-up table register) if the VHDL register description has no reset. SRL cell usage has a small positive impact on both the design size and speed. The small disadvantage is that the SRL cells usage must be taken into account during microprogram writing - the SRL register cannot be reset and thus the program must cope with the fact that "old data before reset" will be read from the register a few clock cycles after reset (depending on the register depth).

6.4 Optimization for area

The optimization for area setting of the implementation tools forces them to take stress on FPGA area utilization. If we have a simpler design like this one, the optimization for area will very likely lead also to higher clock frequency than optimization for speed. If we have a more complicated design, the effort spent on compression of the cells to the FPGA might lead to the high congestion of the routing canals in the FPGA fabric and gives long delays on signals afterwards.

6.5 Optimization for speed

This optimization was used in all the cases, if it is not stated otherwise - it is a standard tool setting.

Parameter	**The original implementation otpimized for speed**	**Block RAM usage**	**SRL cells usage**	**Optimization for area**
LUTs	53	29	51	48
FFs	62	49	53	59
Fclk max	117.5MHz	78.9MHz	121.9MHz	128.6MHz
Notice	standard tool settings + last RTL code	one blockRam used	6 LUTs configured as SRL registers	none

We can see in the table that the best result was attained with the optimization for area in our case. The design is the fastest and the smallest one of all the alternatives.

7 Usage for teaching

To allow the students to easily undestand the whole system we designed a small example of the microcontroller usage with our FPGA kit. The control microprogram implements a simple dice here; the pressing of the button on the kit starts the dice rolling, releasing it stops it and the final result is indicated on the 7-segment display. Further we created a set of simple assignments for the students solving which should help them to understand the microcontroller design concepts deeper:

1. enlarge the microprogram memory to 1024 microinstructions (including the program counter and jump instruction modification),
2. implement a new microinstruction "set N-th bit of output register X to logical value Y",
3. implement a simple stack for the return address and add microinstructions "subroutine call" and "subroutine return",
4. implement the interrupt logic to the microcontroller - add "INT" port; log. 0 to log. 1 transition will result in jump to 001h address where the appropriate interrupt routine resides. To return from the interrupt subroutine utilize the "subroutine return" microinstruction developed for the previous assignment,
5. add a simple ALU and microinstructions performing the following operations:
   port_X and/or/xor port_Y value is stored to port_Z
   port_X +- port_Y is stored into port_Z
   NOT port_X stored to port_Z
6. add the zero flag signalization to the ALU and connect it to the C7 condition (now stuck at 1),
7. design a hardware counter/timer, its input connect either to one of the conditions or use the interrupt input (assignment 4). Add the microinstruction to work with it,
8. propose a system for the IIR filter of the 8th order - use the existing IIR biquadratic section IP macro available at our WWW pages and control the system with the microcontroller. Write the control program which loads the coefficients to the filter sections and periodically activates the sections afterwards. Filtering coefficients store in the microprogram memory (constants in the microinstructions) and adapt the I/O interface to our AD and DA converters integrated on FPGA kits we use.

8 Conclusion

The work presents an extremly simple microcontroller design intended for teaching purposes. Despite the simplicity of the system we can easily demonstrate a lot of concepts related to the wide spectrum of usual design problems from the initial analysis, across implementation to final design verification with it. We hope that the original idea - to briefly make students familiar with the basic microcontroller design ideas - was reached and students who want to know more on this topics are referred to the textbook [14].
The designed IP core is - again despite its simplicity - easily extendable and usable as a standard IP core in future systems designed in our laboratory.
The microcontroller is (including the documentation) freely available for download for non-commercial usage by other academic laboratories on our laboratory WWW site (http://amber.feld.cvut.cz/fpga).

Acknowledgement

This work has been supported by the research program Transdisciplinary Research in Biomedical Engineering No. MSM6840770012 of the Czech University in Prague, and the Grant GACR 102/03/H085: Biological and Speech Signal Modelling.
Further we would like to thank to prof. Pavel Sovka and Ing. Lukáš Ručkay (both work in the Biosignal Processing Laboratory at the Department od Circuit Theory, FEE CTU) for their comments to the text.

References

[1] PEARSON M., ARMSTRONG D., MCGREGOR T. Design of a Processor to Support the Teaching of Computer Systems. Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA'02), pg. 240, 2002
[2] ELLARD D., HOLLAND D., MURPHY N., SELTZER M. On the Design of a New CPU Architecture for Pedagogical Purposes.Proceedings of the ninth workshop on computer architecture education, Anchorage, Alaska, May 2002. pg. 27-33.

[3] NICOUD J. D. Dedicated Tools for Microprocessor Education. IEEE Micro, pg. 14-17, 62-68, February 1991.
[4] VARMA A., KALAMPOUKAS L., STILIADIS D., JACOBSON Q. CPU Design Kit: An Instructional Prototyping Platform for Teaching Processor Design. Technical report, University of California, Santa Cruz. June 1995.
[5] XILINX INC. PicoBlaze 8-bit Microcontroller Reference Design for FPGAs and CPLDs.
[6] CHU Y. A Simple Project for Teaching Instruction Set Architecture. Proceedings of the Fifth IEEE International Conference on Advanced Learning Technologies (ICALT'05), pp. 69-71, 2005
[7] DJORDJEVIC J., MILENKOVIC A., GRBANOVIC N. An Integrated Environment for Teaching Computer Architecture. IEEE Micro, May-June 2000. pg. 66-74.
[8] STAN M. R., SKADRON K. Teaching Processor Architecture with a VLSI Perspective. In Proceedings of Workshop on Computer Architecture Education, Anchorage, Alaska May 2002, pg. 4-10.
[9] CUMMINGS C.E., MILLS D., GOLSON S. Asynchronnous & Synchronous Reset Design Techniques - Part Deux. Synopsys User Group, SNUG Boston 2003
[10] GAURRAULT P., PHILOFSKY B. HDL Coding Practises to Accelerate Design Performance. White Paper WP231, Xilinx.
[11] DVOŘÁK, V., DRÁBEK, V. Architectures of Processors. VUTIUM, Brno, 1999, 330 s., ISBN 80-214-1458-8.
[12] KOLOUCH, J. Možnosti implementace pamětí RAM v obvodech FPGA. Elektrorevue 27/2003, http://www.elektrorevue.cz/clanky/03027
[13] BÍLÝ P., ŠŤASTNÝ J. PicoCTRL - controller. Research report #Z06-1. Biosignal processing laboratory, FPGA laboratory, CTU FEE. 2006.Available at http://amber.feld.cvut.cz/fpga/publications/PicoCTRL.pdf
[14] PLUHÁČEK, A. The design of the computer logic, CTU FEE textbook