



Universal ISA Simulator with Soft Processor FPGA Implementation

Islam Almasri, Gheith Abandah, Ali Shhadeh, Anas Shahrour
Computer Engineering Department
The University of Jordan
Amman 11942, Jordan
islam_almasri@computer.org

Abstract—We present a system that allows simulating wide range of instruction set architectures (ISA). This system includes full development and simulation environment for defining the required ISA, creating and editing assembly programs of the defined ISA, and simulating the execution of these programs. This system also includes a soft processor described in the Verilog hardware description language (HDL). This soft processor is synthesized from a customizable general processor template to implement the defined ISA on a field-programmable gate array (FPGA). This system provides an innovative, generic simulator for many architectures and for experimenting with new ones. It has a unique and easy-to-use interface that focuses on functionality rather than hardware implementation. This system was validated by successfully implementing several ISAs of various ISA classes such as MIPS, x86, and PIC. This system provides a flexible tool for teaching assembly language and computer architecture. Additionally, it provides an easily customizable soft processor that allows testing programs with real I/O interfacing circuits.

Keywords—*simulator; ISA; soft processor; FPGA design; micro-programming*

I. INTRODUCTION

The paper describes a universal system for instruction set architecture (ISA) definition and simulation. Students can use this environment to learn some ISA concepts. It can also be used by anyone who wants to define a new ISA and simulate it for experimental purposes. The suit of tools of this environment is used to design an ISA, simulate it, and burn a processor that implements this ISA into a field-programmable gate array (FPGA). Figure 1 shows the main functional units of this environment (bold borders) and the exchanged definitions and files among these units (thin borders). The ISA Maker is used to define the ISA and output the ISA definition. This definition is the base for full assembly simulation using the ISA Simulator which is a universal assembly language simulator.

The ISA definition can then be exported to hardware using the Processor Hardware Description Language (HDL) Synthesizer. This synthesizer customizes a generic soft processor HDL template to produce a custom soft processor description in Verilog HDL that can be programmed on a field-programmable gate array (FPGA) chip.

The following section reviews related work. Section III provides the details of this system. Section IV describes the FPGA implementation and the validation experiments. Finally, Section V provides some conclusions and some future work.

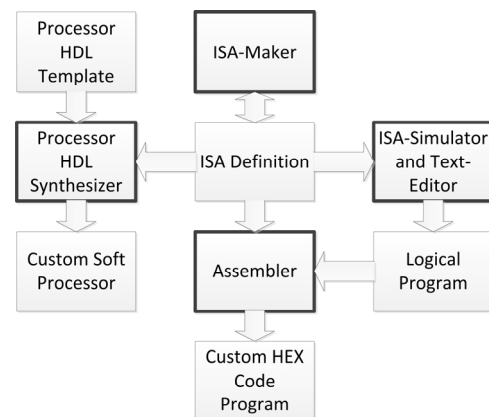


Figure 1. The main components of the system and their interactions

II. RELATED WORK

This research includes multiple components. Each component rivals a well-established group of offerings in its field. Yet it still provides sometimes unique and extra features over existing solutions. The areas in which this research is involved include: CPU simulators, hardware description languages, and soft processors.

A. Overview

Existing processor simulators offer integrated environments for certain assembly languages. The system components include a simulator with unique features. It can be used with any instruction set and it provides a generic environment for processor simulation. It also gives the user the ability to design new instruction set. One can also edit or preview the implementation of each instruction in any instruction set.

Logic synthesis techniques have been evolving to facilitate the process of making logic circuits [1]. Hardware description languages can be used to structurally define a logic circuit by connecting logic components [2]. Such structures can also be defined using special schematic drawing tools [3, 4]. Newer tools can also convert behavioral code or block to describe parts of circuits into logic gates. These techniques are also

modular, so they can be used to design complex designs such as processors. But shouldn't there be an easier way to build a processor without going through the hurdles of the logic design?

This research includes a soft processor implementation whose architecture can be modified using the processor HDL synthesizer to implement any instruction set. More on how our solution compares to existing soft processors is in Section II.D.

B. Previous Simulators: What Do They Lack?

Many processor simulators are available such as the famous MASM32 simulator for the x86 instruction set [5], the SPIM for the MIPS instruction set (Figure 2 shows the PCSpim simulator), and PICsim for the PIC architecture [6]. These simulators usually provide an integrated environment that includes: a specialized text editor for writing programs, an assembler that converts object code into machine code, and an emulation engine that lets the programmer debug the code and follow the results of the execution of the code in a step-by-step fashion.

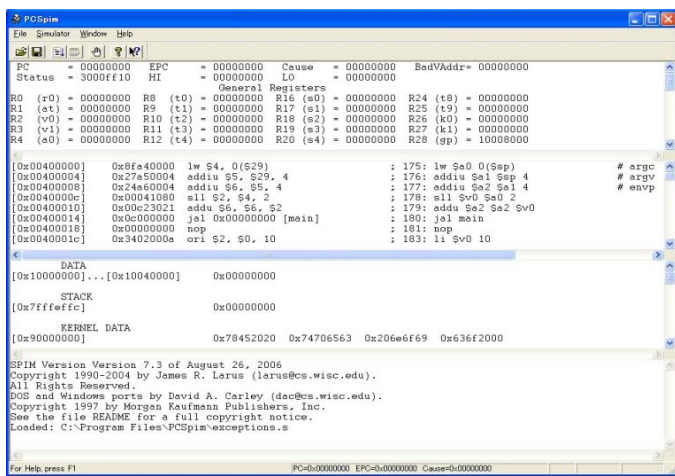


Figure 2. A screenshot of the PCSpim simulator

Most of the offered simulators mainly support one instruction set. One problem with having so many different simulators is that because each ISA has its own particularities, the user experience varies between them. A user who wishes to learn a new instruction set, has to adapt to a different environment with a different user-interface, different conventions, and varying functionality. But since all simulators share many common features and are mainly different because of the underlying architecture that they simulate, why not have a common simulator that may work with many ISAs?

The educational benefit that a computer organization student may gain from using a simulator is to deepen the knowledge in the instruction set by observing the results of the instructions and by developing programs. An added functionality to the simulator that allows the student to define the instructions would surely increase this understanding. Even more, the student will be able to create his own instruction set without the need to develop a simulator from the scratch.

Current simulators do not have the feature described above. A user usually is allowed to simulate one fixed instruction set per simulator. The user cannot define instructions and/or instruction set of his creation. The user has to adapt to a different simulator with a different environment if he wishes to study a new ISA. Finally, the user is not able to investigate the functionality of each instruction in a systematic non-semantic matter. Our universal simulator, which is one part of this research, has all the above features combined. In addition to the traditional features expected from any simulator, there are new innovative additions like dynamic viewing and editing of registers and memory contents, dynamic syntax coloring of assembly instructions, interactive assembly programming using input and output pseudo registers that initiate input and output dialogs, and unlimited simulation memory or register sizes.

C. Processor Design on the Logical Level

Hardware Description Languages are used to describe the hardware implementation on the gate-level of logic circuits. These languages are usually modular such that smaller modules can be defined and used to compose larger and larger modules and eventually systems [7].

One way to implement digital system design is using FPGAs. These devices are programmable devices and contain a large number of gates and connections. They can be programmed by changing these connections. They offer excellent cost and flexibility features. The support tools that come with these devices include supporting HDL languages. These tools also support behavioral descriptions for some of the functions and can transform these descriptions into structural implementations. To make things even easier, these tools also include predefined templates of common logic components. Modern tools like Quartus also support schematic design [4] as in the example shown in Figure 3.

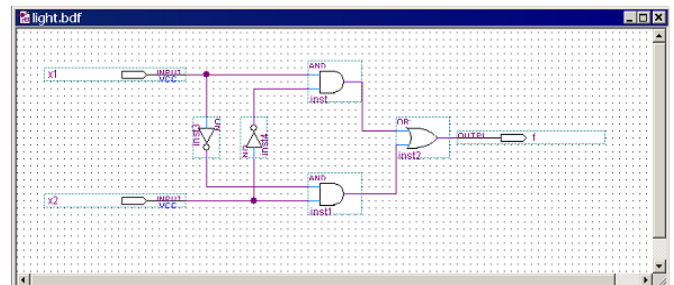


Figure 3. Schematic design using Quartus

The design process is fully modular and can go up to a full processor on an FPGA. This may be a good way of doing it in the cases where an exact architecture of a processor is required and is important. But how about when a user wants to design a processor on the logical-level and runs it on an FPGA chip without worrying about gates and modules? Unfortunately there are no tools designed for that. There are some commercial tools like the Synopsys Processor Designer that enables the user to specify designs at a very high level of abstraction using C/C++ languages. But it does not support definition at the instruction-level, and therefore they cannot be used for educational purposes. Altera provides the SOPC Builder that lets the user to alter the design of a ready NIOS

processor. Yet the NIOS has limited flexibility in how much a user can change.

D. Soft Processors

Soft processors are processor designs that can be implemented using logic synthesis techniques such as FPGAs and CPLDs [8]. These designs are usually user customizable in some ways. Some come with support programs to allow alternation or addition to the vanilla design Others are completely open-source and the whole design is available to the public for alternation.

Due to the flexibility the soft processors offer, there have been many studies to reduce the performance gap between them and hard processor designs. These techniques include dynamic hardware/software partitioning [9] and application specific customization [10]. This proved that customization of soft-processor hardware plays an important role in the performance. But no easy-to-use customization process has been suggested yet.

As an example, the NIOS is a processor architecture that is designed to run on Altera FPGAs. Its RISC architecture makes it suitable for many embedded applications. It is offered in three configurations that are different in the number of needed logical elements (LEs) and performance.

The architecture of the NIOS processor is editable via the Altera SOPC builder which is part of the Quartus tools. The user can add extra instructions and custom peripherals However, the pipeline stages, instruction formats, and registers will always be the same. The NIOS processor is not open source.

The NIOS and the SOPC builder offer a good solution for those who are looking to develop their own processor on an Altera FPGA. But it is different from our solution in:

- 1) It has a fixed instruction set architecture where only some instructions can be added.
- 2) The added instructions are heavily restricted by the architecture.
- 3) It does not provide complete abstraction and flexibility on the logical instructions level.

III. SYSTEM OVERVIEW

This system provides a set of tools that allow the user to define an ISA, simulate an ISA, and exporting an implementation of an ISA. This implementation is then implemented into hardware on an FPGA. Each tool has some properties that distinguish it from other tools available in the market.

A. The ISA Maker and ISA Definition

The ISA Maker is a tool responsible for creating and editing an ISA definition. This definition is a data holder that encapsulates all information describing an ISA including the addressable memory, registers, and instructions. As is illustrated in Figure 1, the ISA definition is the center point of all tools. It can also be saved and read from a file.

The most important aspects that you define about the ISA are the registers and instructions. You define registers by

giving the number of registers, each register name, and the size of each register in bits. The ISA Maker does not limit the register width. You can also define what we call register aliases which are other names for a register or for part of it, e.g., AL is an alias for the least significant byte of AX in the x86 architecture.

Instructions are defined by two things: (i) the *instruction template string*, which is a string that contains the operation code (opcode), operands, and operands' types, and (ii) the function of the instruction which is defined through a technique similar to micro-programming [11]. There is a number of built-in predefined *operations* that are adopted from the MIPS ISA with some additions and modifications [12]. The user should map the instruction to a series of operations that represent it. During the mapping process, some temporary registers may be needed. This is the third type of registers you can define. Figure 4 shows a screenshot of the Instruction editor where the user defines instructions. The example for one of the MOV instructions of x86 ISA.

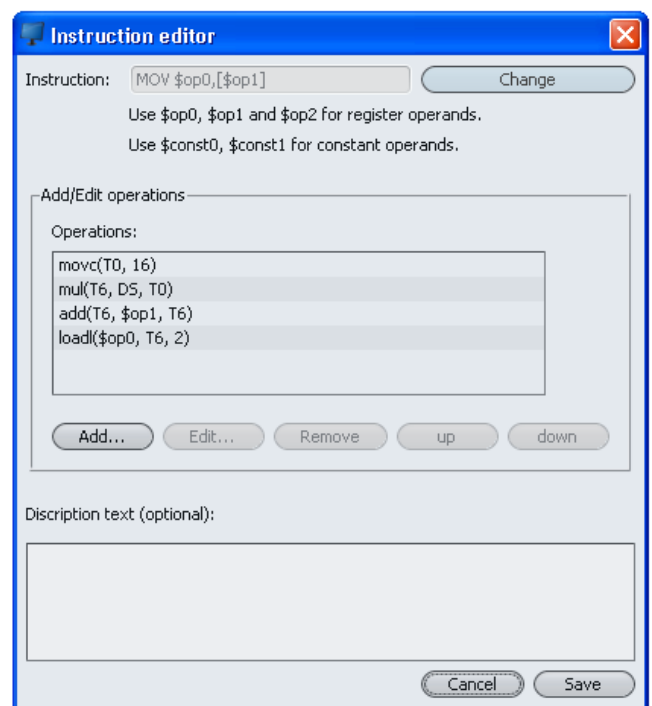


Figure 4. Instruction definition (x86 instruction example shown)

Our definition of the ISA does not consider some aspects of the hardware such as instruction level parallelism. So for architectures like the EDGE or the MIMD, where more than one instruction can be executed at the same time, you will not be able to fully express the architecture using the ISA Maker. But you will still be able to define the behavioral aspects of the ISA (like registers, instructions, memory size, etc.) and run programs line by line. As an example of flexibility, we were able to implement an ISA that supports predicated instruction.

B. The ISA Simulator: A Universal Simulator

The ISA Simulator is a full assembly language simulator. It can fully simulate any ISA definition created by the ISA Maker and it is accompanied with a specialized text editor to extract a

program from plain text. The Logical Program is a logical ordered set of instructions that the simulator can easily access to perform simulation. It is also converted by the Assembler to binary code that can be downloaded and run by the FPGA processor that is created by the Synthesizer.

Once an assembly language is defined using the ISA Maker, a full functionality simulation can be run using this simulator. The user can write programs using the defined language and start executing these instructions and check their effects on the memory and registers.

The features of the simulator include the typical features of assembly simulators such as break points, step in, run, stop, and run time error detection such as the classic divide-by-zero error.

C. Universal Soft Processor Architecture

The micro-architecture adopted for the soft processor is a single-cycle MIPS like core [12]. It is combined with a special pre-decoding stage for converting instructions into operations. Figure 6 shows this very important decoding stage.

The hardware implementation of the processor is described as Verilog [13] code (called Custom Soft Processor in Figure 1). It is produced by the Processor HDL Synthesizer by altering a generic Verilog processor template using the information that the ISA definition entity holds. This implementation is designed and tested to work on an Altera Cyclone II FPGA chip [14].

The model chosen for the processor to simplify the process of the ISA definition and the processor synthesis is illustrated in the flowchart of Figure 5. The processor hardware can execute a certain number of built-in operations. Each instruction should be defined as one or more operations acting as a subroutine. This model proved to be sufficient for a large number of instructions and ISA definitions.

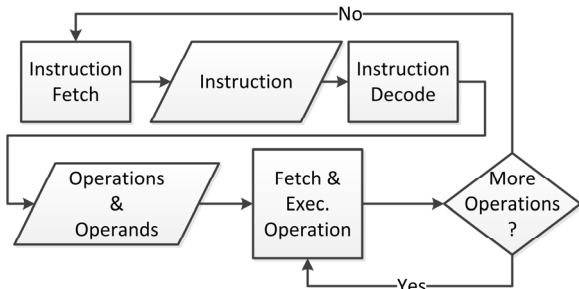


Figure 5. Flowchart of executing instructions through translation into operations

The execution starts at the decoding stage illustrated in Figure 6. There is the typical instruction memory which holds the instructions of the program. In our architecture the instruction code contains two pieces of information:

- 1) Instruction opcode: which is effectively the address in the Operations Memory where the functionality of the instruction is defined.

- 2) Instruction Operands: The operands upon which or from which the instruction operates. These operands vary depending on the instruction. Currently each instruction can hold up to 4 register operands or 3 register operands and 1 constant operand.

Once the instruction is fetched into the instruction register (IR1 in Figure 6), the sequence of operations representing it will execute while IR1 is held constant. During the execution of each operation, the decoding unit (not shown in figure) sends the proper control signals to the operand decoding unit to pick the right operands either from the instruction or from the operation.

After fetch and decode stage, the architecture of the execution units is pretty much a simplified version of a single-cycle MIPS architecture.

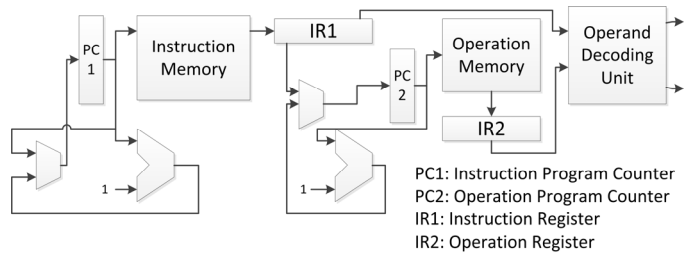


Figure 6. The fetch and decode stage

IV. FPGA IMPLEMENTATION AND TESTING

After the Verilog HDL Template that represents our soft processor was developed and tested using Verilogger Pro, it was programmed on an FPGA IC. The functionality of the processor was tested using a small hand-assembled MIPS assembly programs on the Alter DE2-70 Development Board.

A. Test Program

First, the Operation Memory, that contains the mappings between instructions and operations, was loaded with definitions of some MIPS-like instructions and a halt instruction. These definitions are used to write the test program. See Table I.

Memory address “0” is filled with No-Operation that is executed in the first cycle after startup. This ensures that the operation and instruction registers are filled with the correct values. Each instruction ends with the special operation “endop” to indicate that the memory block representing this instruction has ended. The Halt instruction is essentially an infinite loop within two operations.

TABLE I. PRELOADED CONTENTS OF THE OPERATIONS MEMORY

Instruction	Address	Operations	Op code (H)
NOP	0	nop	0000 0000
MOV \$op0, \$const0	1	move \$op0, \$const0	0504 0000
	2	endop	FC00 0000
ADDI \$op0, \$op1, \$const0	3	move T0, \$const0	0604 0000
	4	add \$op0, \$op1, T0	8102 1800
	5	endop	FC00 0000
MULI \$op0, \$op1, \$const0	6	move T0, \$const0	0604 0000
	7	mul \$op0, \$op1, T0	9502 1800
	8	endop	FC00 0000
BEQ \$op0, \$op1, \$const0	9	sub T0, \$op0, \$op1	8602 0420
	10	ifnot Z, 1	1408 0008
	11	jump \$const0	9C04 0000
	12	endop	FC00 0000
HALT	13	sub T0, R0, R0	8600 0000
	14	if Z, -1	100B FFFC

The purpose of the test program is to test how successful the integration of the processor is on the FPGA IC. The program was compiled and synthesized using the Quartus II software. This short code tests the following:

- 1) The mapping between Instructions and Operations
- 2) Operation decoding
- 3) The program flow sequence
- 4) Data flow between different parts of the processor

This code snippet is short and thus does not test all operations. However it tests the general instruction-operation flow. Table II lists the program instructions with their locations in the memory.

TABLE II. TEST PROGRAM

Memory Location	Instruction	OP code (H)	Notes
0	MOV R1, 3	0420 0003	R1 = 3
1	ADDI R0, R1, 4	0C01 0004	R0 = 7
2	MULI R2, R0, 2	1840 0002	R2 = 14
3	MOV R3, 14	0460 000E	R3 = 14
4	BEQ R2, R1, 1	2443 0001	Branch taken
5	ADDI R0, R1, 5	0C01 0005	Will not execute
6	ADDI R0, R1, 10	0C01 000A	R0 = 13
7	HALT	3400 0000	-

B. Test Hardware Setup

The input clock is attached to the Push Button 0 on the kit not the 50 MHz oscillator so that the sequence of execution can be observed.

Three internal busses are wired to the available LEDs on the kit. The opcode fields of the operation and instruction busses are observed to check the current Instruction and Operation being executed. Table III includes the expected outcome of this experiment. Noting that there is a “nop” cycle between successive instructions to flush the operation register

and only the first few instructions are shown. When the program was executed, the observed behavior matched the expected.

TABLE III. TEST SEQUENCE

Instruction	Operations	Instruction Op Code	Operation Op Code	Data Bus LSB (H)
MOV R1, 3	movc R1, 3	000001	000001	03
	endop		111111	XX
	nop		000000	XX
ADDI R0, R1, 4	movc T0, 4	000011	000001	04
	add R0, R1, T0		100000	07
	endop		111111	XX
	nop		000000	XX
MULI R2, R0, 2	movc T0, 2	000110	000001	02
	mul R2, R0, T0		100101	0E
	endop		111111	XX
	nop		000000	XX
MOV R3, 14	movc R3, 14	000001	000001	0E
	endop		111111	XX
	nop		000000	XX
BEQ R2, R1, 1	sub T0, R2, R1	001001	100001	00
	ifnot Z, 1		000101	XX
	jump 1		100111	XX
	endop		111111	XX
	nop		000000	XX
ADDI R0, R1, 5	movc T0, 5	000011	000001	05
	add R0, R1, T0		100000	08
	endop		111111	XX
	nop		000000	XX
ADDI R0, R1, 10	movc T0, 10	000011	000001	0A
	add R0, R1, T0		100000	0D
	endop		111111	XX
	nop		000000	XX
HALT	sub T0, R0, R0	001101	100001	00
	if Z, -1		000100	XX

C. Quartus II Specific Changes

A number of changes were needed on the Verilog code to meet the Quartus specific limitations after testing the code using Verilogger Pro. These changes are summarized below:

- 1) Unifying processes: there cannot be two “always” blocks affecting the same variable.

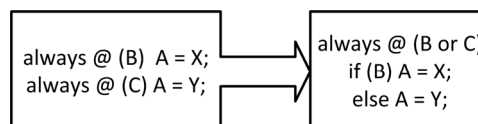


Figure 7. Unifying two processes affecting the same variable

- 2) No need to initialize memory locations: Quartus, by default, initializes all registers and RAM locations to zeros.
- 3) Initialize Control Registers: Quartus always initializes all registers including the control registers to zero. This is not always the desired behavior.
- 4) Re-latch ROM output on change: this is a bug that originate from the behavioral modeling of the ROM memory. The output of the ROM is latched to the output, thus, if the same ROM location is changed, it needs to be re-latched in case the next access to the ROM will be to the same address. See Figure 8.

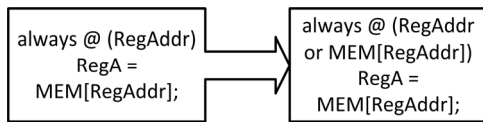


Figure 8. Re-latching ROM output on write

V. CONCLUSIONS

The Simulator and ISA Maker were successfully tested by implementing various existing instruction sets belonging to different classes as shown in Table IV. After implementing all ISA, each ISA was tested using several test programs.

TABLE IV. TEST ISAS AND THEIR CLASSES

ISA Name	ISA Classes
x86	CISC, Register-Memory, 2 operands
MIPS	RISC, Register-Register, 3 operands
PIC	RISC, Accumulator, 1 operand

This research has received recognition in international and local competitions. It started as a contribution in the 2010 IEEE Computer Society System Design Competition. This system reached the top 12 finalists worldwide. We were also invited to exhibit the program in the National Technology Parade in 2010 in the field of education.

There is still a space for a lot of work in this research such as:

- 1) Implementing all operations in hardware, currently only the small subset of primitive operations is implemented.

- 2) Expand the processing core to include pipelining and multiple functional units.
- 3) Support for non-byte-oriented memories to support more instruction sets.
- 4) Include input and output operations to support separate IO architectures.

REFERENCES

- [1] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, Springer, 2008.
- [2] J. Mermet, *Fundamentals and Standards in Hardware Description Languages*, Springer, 1993.
- [3] Z. Salcic and A. Smailagic, *Digital Systems Design and Prototyping: Using Field Programmable Logic and Hardware Description Languages*, 2nd ed., Springe, 2000.
- [4] Altera Corp., *Quartus II Introduction Using Schematic Design*, ftp://ftp.altera.com/up/pub/Tutorials/DE2/Digital_Logic/tut_quartus_intr_o_schem.pdf, last accessed on Nov 6, 2011.
- [5] The MASM32 Simulator Website. <http://www.masm32.com/>, last accessed on Nov 6, 2011.
- [6] W. Srisa-an, PCSPIM Tutorial, http://www.inf.pucrs.br/~calazans/undergrad/arq1/mips/spim_tutorial.pdf, last accessed on Nov 6, 2011.
- [7] S. Ghosh, *Hardware Description Languages: Concepts and Principles*, Wiley-IEEE Press, 1999.
- [8] Xilinx.com, What is a Soft Processor, http://japan.xilinx.com/ipcenter/processor_central/microblaze/doc/mb_fa_q.pdf, last accessed on Nov 6, 2011.
- [9] R. Lysecky and F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *Proc. Conf. Design, Automation and Test in Europe*, vol. 1, 2005, pp. 18–23.
- [10] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *Proc. ACM/SIGDA 14th Int'l Symp. Field Programmable Gate Arrays*, 2006 pp. 201–210.
- [11] S. G. Tucker, "Microprogram control for SYSTEM/360," *IBM Systems Journal*, vol. 6, no. 4, 1967, pp. 222–241.
- [12] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed., Morgan Kaufmann, 2009.
- [13] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 2002.
- [14] Altera Corp., Cyclone II product page, <http://www.altera.com/products/devices/cyclone2/cy2-index.jsp>, last accessed on Nov 6, 2011.