CPU Design Simplified

Abdullah Yıldız*, H. Fatih Ugurdag†, Barış Aktemur‡, Deniz İskender‡, and Sezer Gören*
*Dept. of Computer Engineering, Yeditepe University, Istanbul, Turkey
Email: ayildiz@cse.yeditepe.edu.tr, sgoren@yeditepe.edu.tr
†Dept. of Electrical and Electronics Engineering, Özyeğin University, Istanbul, Turkey
Email: fatih.ugurdag@ozyegin.edu.tr
‡Dept. of Computer Engineering, Özyeğin University, Istanbul, Turkey
Email: baris.aktemur@ozyegin.edu.tr, deniz.iskender@ozu.edu.tr

***Abstract*—The first goal of this paper is to introduce a simple and customizable soft CPU named VerySimpleCPU (VSCPU), which could be easily implemented on FPGAs with a complete toolchain including instruction set simulator, assembler, and C compiler. The second goal is to offer to use this CPU as a teaching material within computer architecture/organization courses for students to understand the essentials and inner workings of a CPU better by designing a simple one. In addition to this, it is also aimed to teach writing code both in assembly level and C level for the CPU designed to understand what a compiler is and why it is needed.**

***Index Terms*—VSCPU, processor, compiler, FPGA, education**

I. Introduction

CPUs are indispensable in computing. Different than other hardware implementations, they are the most obvious examples that separate hardware from software abstraction (that is why they offer extreme flexibility via software programming). Hence, understanding of CPUs is essential for an electrical or computer engineering student in order for him/her to thoroughly understand the concepts of hardware and software. As a result, curriculum of a typical computer architecture/organization course must include hands-on design and implementation of a CPU. Based on the facts above, we can say the following:

- It is not possible to understand advanced concepts such as pipelining, memory hierarchy, branch prediction, caching, etc. without understanding essentials of processors.
- Students must code in assembly language of the processor to understand how a processor works.
- Students could design their own CPU but in order for them to fully employ their CPU in an application, a basic understanding of how a compiler works and how it allocates memory is also important.

It is clear that it is not reasonable for an engineering school to design and manufacture a CPU chip. However, thanks to FPGAs, they eliminate the need to access to expensive CAD tools and manufacturing facilities and make digital design affordable for such institutions and individuals.

In this paper, we introduce VerySimpleCPU (VSCPU), which is a simple, customizable, educational, and easily implementable CPU with a complete toolchain.

This work is supported by TÜBİTAK under contract 117E090.

II. Background

In this part, our findings in the literature are given on improving the content of computer architecture courses and teaching processor design at undergraduate level.

Li et al. [1] described design and implementation of an 8-bit pipelined processor. The processor was designed using Cadence EDA tools and implemented on a Xilinx FPGA. Students were asked to design the processor, to perform functional simulations, to implement the design, etc. The processor mentioned is a custom RISC-based pipelined design. The exercises of the course include analysis of processor instruction set, design of data path and control unit, circuit schematic, functional simulation, netlist generation, pin assignment, placement and routing, etc. No high-level language support was provided.

Gray [2] introduced XSOC Project, which includes the xr16 RISC CPU core (both 16-bit and 32-bit), SoC infrastructure, peripheral cores, C compiler, and simulator. The author claims that FPGA-based SoCs can be alternatives to ASICs to establish a community of designers, a library of cores, motivate students to design and build custom processing systems.

Šulık et al. [3] described the design of an 8-bit RISC microcontroller using Handel-C.

Carpinelli [4] described a Java-based simulator which supports visualization of how an 8-bit processor fetches, decodes, and executes instructions. Assembly language programs are assembled by the simulator to run a cycle-accurate simulation of the processor.

Ellard et al. [5] aimed to create a framework to teach VLSI circuit design and implementation, machine architecture, assembly language programming, compiler code generation, and operating systems. It was created since the existing x86, MIPS, and similar architectures are complicated to master them. A 32-bit RISC architecture was used.

Nakano et al. [6], [7], and Nakamura et al. [8] described the design of a 16-bit processor (TINYCPU), cross assembler (TINYASM), and cross compiler (TINYC). The processor was designed using Verilog HDL. TINYASM was written in Perl and TINYC was written in Flex/Bison. The processor was implemented on Xilinx FPGAs.

Angelov et al. [9] described the design of a 16-bit RISC processor. The design entry is done as schematic and partly

using a core generator. No HDL was used. Primary goal is to teach without any previous knowledge of digital electronics or HDL.

Ribas [10] described the design of an 8-bit MCU-like educational processor with support of assembler, simulator, etc.

Qian et al. [11] described the design of a 14-bit small CPU and interface chip.

Lee et al. [12] described a pipelined 32-bit CPU design which was implemented on an FPGA. It combines three pedagogical methods: problem-based learning (PBL), hands-on learning, and incremental approach.

Chen et al. [13] described a 32-bit CPU which is a reduced version of MIPS CPU.

Yamazaki et al. [14] described the organization of a workshop which includes topics related to computer architecture in order to assess the knowledge of the students. A survey was also conducted to get feedback from the students. Z80 CPU core was used.

Zavala et al. [15] described the design of an 8-bit RISC soft-core processor which was implemented on Xilinx FPGAs.

Table I represents a comparison between these educational CPU designs in terms of toolchain support. VSCPU is also included at the bottom of the table to see the advantage of our VSCPU toolchain with respect to others.

TABLE I
COMPARISON OF CPUS IN TERMS OF TOOLCHAIN SUPPORT

CPU	Assembler	Simulator	Compiler	Web-based Use
[1]	-	-	-	-
[2]	+	+	+	-
[3]	-	-	-	-
[4]	-	+	-	-
[5]	+	+	-	-
[6] [7] [8]	+	-	+	-
[9]	-	+	-	-
[10]	+	+	-	+
[11]	-	-	-	-
[12]	-	-	+	-
[13]	-	-	-	-
[14]	-	-	-	-
[15]	-	-	-	-
VSCPU [16]	**+**	**+**	**+**	**+**

III. VSCPU

VSCPU [16] is a simple and customizable 32-bit soft CPU core. The instruction set architecture (ISA) of VSCPU has sixteen instructions and supports unsigned integer arithmetic operations.

Main features of VSCPU can be summarized as follows:

- 16 instructions
- Multi-cycle execution
- 64KB memory
- Up to 16 MIPS throughput at 50MHz
- Registers:
  - **PC**: Program Counter (14-bit)
  - **IW**: Instruction Word (32-bit)
  - **R1**: General-Purpose Register (32-bit)
  - **R2**: General-Purpose Register (32-bit) (Optional)
- Interrupt support
- Peripheral support via memory-mapped I/O:
  - General-Purpose Input-Output (GPIO)
  - Pulse-Width Modulation (PWM)
  - Inter-Integrated Circuit (I2C)
  - Serial Peripheral Interface (SPI)
  - Universal Asynchronous Receiver-Transmitter (UART)
  - Video Graphics Array (VGA)

VSCPU, memory interface, interconnect logic, and all peripheral interfaces are designed in Verilog RTL.

A. VSCPU ISA

VSCPU ISA has sixteen instructions and supports unsigned integer arithmetic operations. Table II shows instruction word format for VSCPU ISA.

TABLE II
VSCPU INSTRUCTION WORD

	Instruction Word			
bit position	[31:29]	[28]	[27:14]	[13:0]
field name	opcode	immediate	operand A	operand B
bit width	3-bit	1-bit	14-bit	14-bit

The leftmost three bits in instruction word includes instruction *opcode* which defines operation type of the instruction. *operand A* can refer to either address of operand or destination address or both of them. *operand B* can refer to either address of the operand or value of the operand itself depending on the value of *immediate* bit (the case is different for *CPIi* instruction). Table III lists the instruction set of VSCPU.

B. VSCPU Instruction Cycle

VSCPU instructions are multi-cycle. Therefore, we can represent instruction cycle of VSCPU as a finite state machine. Figure 1 shows the state machine representation of VSCPU.

In Figure 1, states with green color represents the regular instruction cycle. As shown here, each VSCPU instruction takes more than one cycle (state) to complete its instruction cycle. Depending on the instruction type, it can take either three or four cycles to finish an instruction. VSCPU could also handle interrupts with a simple mechanism. States with yellow color are used for this purpose and represents the additional actions taken when an interrupt occurs. Actions taken within each state can be summarized in Table IV.

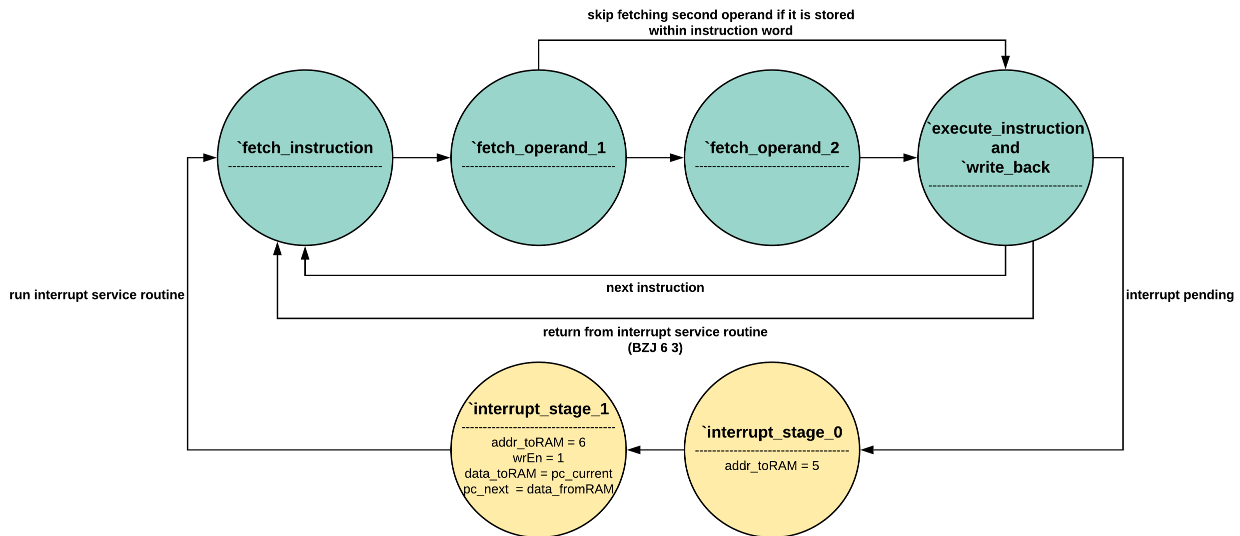

Fig. 1. State machine representation of VSCPU

TABLE IV
STATE DESCRIPTIONS OF VSCPU

States	Description
fetch_instruction	fetch the next instruction from memory which is pointed by PC
fetch_operand_A	decode the instruction and fetch the first operand from memory
fetch_operand_B	fetch the second operand from memory
execute_instruction_and_write_back	perform ALU operations, store the result in memory, set PC
interrupt_stage_0	get the address of interrupt service routine (ISR)
interrupt_stage_1	store current PC in memory to continue while returning from ISR, go to ISR

VSCPU has no special instruction to return from interrupt. Instead, there are two reserved addresses (five and six) in memory to handle interrupts. At the fifth address, we keep the address of interrupt service routine (ISR) to handle the interrupt. At the sixth address, we keep the address of the next instruction to be executed after returning from the interrupt. Since there is no instruction to return from interrupt, `BZJ 6 3` instruction is used to return from the interrupt and continue the program where it left off. For this reason, every ISR should end with this instruction.

IV. VSCPU TOOLCHAIN

We developed several assemblers and instruction set simulators with different languages up to now. A C Compiler is also available which supports a small subset of C syntax.

A. Instruction Set Simulator & Assembler

We developed a Python-based instruction set simulator and assembler which can simulate VSCPU assembly code and generate the corresponding machine code. Listing 1 shows a working example of Python-based script on how simulation is done.

```
# ./assembler_iss_mem_init_gen.py -i example.asm -a 1 -s 1
Starting simulation

step-by-step mode selected

Press any key to continue

* * * * * * * * * * * * * * * * * * * * * * * * *
        current_instruction:  ADD 10 11
        program counter    :  0
        Memory content before executing instruction
        mem[ 10 ]        :  5
        mem[ 11 ]        :  2
        Memory content after executing instruction
        mem[ 10 ]        :  7
        mem[ 11 ]        :  2
* * * * * * * * * * * * * * * * * * * * * * * * *

Press any key to continue

* * * * * * * * * * * * * * * * * * * * * * * * *
        current_instruction:  BZJi 20 1
        program counter    :  1
        mem[ 20 ]          :  0
* * * * * * * * * * * * * * * * * * * * * * * * *
Finishing simulation. Instruction called itself.
```

Listing 1. VSCPU Assembler and ISS Script Working Example

A web application [17] [18] is also available for the instruction set simulator and the assembler.

B. C Compiler

We developed a compiler from a subset of C to VSCPU. The compiler is written as a Clang frontend action [19]. Given a C source file, the program is preprocessed, parsed, and semantically analyzed using Clang – an industrial-strength C compiler. Syntax and semantic errors (e.g. type mismatches) are detected, and Clang's high-quality error/warning messages are displayed. If no errors are detected, the abstract syntax tree (AST) of the input program is passed to the VSCPU frontend handler, which traverses the tree and emits the corresponding VSCPU assembly code.

TABLE III
VSCPU INSTRUCTION SET

Mnemonic	Description	Microoperations
Arithmetic & Logic Instructions		

`ADD A B`	unsigned add	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`mem[A] <- (R1 + R2)`
		`PC <- PC + 1`

`ADDi A B`	unsigned add immediate	`R1 <- mem[A]`
		`R2 <- B`
		`mem[A] <- (R1 + R2)`
		`PC <- PC + 1`

`NAND A B`	bitwise NAND	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`mem[A] <- ~(R1 & R2)`
		`PC <- PC + 1`

`NANDi A B`	bitwise NAND immediate	`R1 <- mem[A]`
		`R2 <- B`
		`mem[A] <- ~(R1 & R2)`
		`PC <- PC + 1`

`SRL A B`	shift right/left	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`mem[A] <- (R2 < 32) ? (R1 >> R2) : (R1 << (R2-32))`
		`PC <- PC + 1`

`SRLi A B`	shift right/left immediate	`R1 <- mem[A]`
		`R2 <- B`
		`mem[A] <- (R2 < 32) ? (R1 >> R2) : (R1 << (R2-32))`
		`PC <- PC + 1`

`LT A B`	compare and set	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`mem[A] <- (R1 < R2) ? 1 : 0`
		`PC <- PC + 1`

`LTi A B`	compare and set immediate	`R1 <- mem[A]`
		`R2 <- B`
		`mem[A] <- (R1 < R2) ? 1 : 0`
		`PC <- PC + 1`

`MUL A B`	unsigned multiply	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`mem[A] <- (R1 * R2)`
		`PC <- PC + 1`

`MULi A B`	unsigned multiply immediate	`R1 <- mem[A]`
		`R2 <- B`
		`mem[A] <- (R1 * R2)`
		`PC <- PC + 1`

Data Transfer Instructions		

`CP A B`	copy data	`R2 <- mem[B]`
		`mem[A] <- R2`
		`PC <- PC + 1`

`CPi A B`	copy data immediate	`R2 <- B`
		`mem[A] <- R2`
		`PC <- PC + 1`

`CPI A B`	copy data indirect	`R1 <- mem[B]`
		`R2 <- mem[R1]`
		`mem[A] <- R2`
		`PC <- PC + 1`

`CPIi A B`	copy data indirect immediate	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`mem[R1] <- R2`
		`PC <- PC + 1`

Program Control Instructions		

`BZJ A B`	branch on zero	`R1 <- mem[A]`
		`R2 <- mem[B]`
		`PC <- (R2 == 0) ? R1 : (PC + 1)`

`BZJi A B`	unconditional branch	`R1 <- mem[A]`
		`R2 <- B`
		`PC <- (R1 + R2)`

As of time of writing this paper, the compiler is able to handle pointer arithmetic, array operations, functions with `void` and non-`void` return types, direct or indirect recursion, `const` and non-`const` global variables, for-loops and while-loops with `break` and `continue`. The following types are supported by the compiler: `int` (32-bit), pointer to a supported type (e.g. `int*`, `int**`), one dimensional array of a supported type (e.g. `int[]`, `int*[]`). The compiler has certain limitations as well. Currently, there is no support for `struct`s and the `switch` statement. A standard library does not exist (i.e no dynamic memory management via `malloc` and `free`). The code generated for division and remainder operations are based on subtraction and looping, and hence, very slow.

Figure 2 shows a sample C program and the corresponding assembly code produced by our compiler. The assembly code

```c
int x = 1234;
const int pi = 314;

void modify() {
  x = 456;
}

int main() {
  modify();
  return 99 + x;
}
```

```
0: BZJi 3 33 // Goto main -- Jump to main
// $REGISTERS_SECTION:
1: 73 // Address of $main_base -- SP
2: 73 // Address of $main_base -- BP
3: 0 // Zero
4: 4294967295 // Negative one
5: 0 // VSCPU Special 1
...
10: 0 // VSCPU Special 6
11: 0 // GP Reg 1
...
16: 0 // GP Reg 6
// $TEXT_SECTION:
// modify:
17: ...
...
// main:
33: ...
...
// $CONSTANT_DATA_SECTION:
// @pi:
67: 314 // pi
// @456:
68: 456
// @99:
69: 99
// $GLOBAL_DATA_SECTION:
// @x:
70: 1234 // x
// $STACK_SECTION:
71: 64 // Return address (RA) of main
72: 0 // A dummy oldBP value
// $main_base:
```

Fig. 2. *Left:* A sample C program. *Right:* The VSCPU assembly code produced by the compiler.

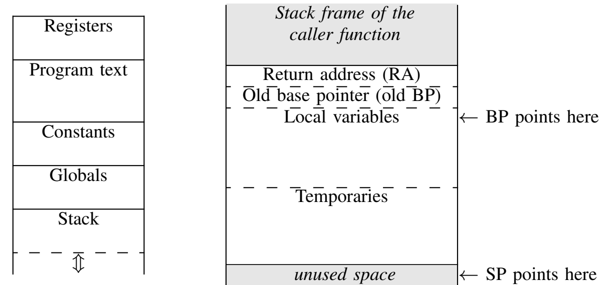

Fig. 3. *Left:* Layout of the assembly code emitted by the compiler. *Right:* Layout of the stack frame (i.e. activation record) of a function.

consists of five main sections as shown in the left hand side of Figure 3. Program text section contains the instructions to call the main function, and the instructions of each function in the program (words 17–66 in Figure 3). This section is read-only. Constants section contains the integer literals that appear in the source code, and the data for global variables that have been defined using the `const` qualifier (words 67–68 in Figure 3). This section is read-only. Globals section contains the data for the non-`const` global variables (word 69 in Figure 3). The program text, constants, and globals sections have a fixed size for each input program. Registers section of the memory (words 1–16 in Figure 3) is reserved for the following purposes:

1) Special-purpose registers. There are four of these:
   - *SP:* The *stack pointer* (SP) stores the address of the top of the stack. We use this value whenever we want to push/pop something to/from the stack.
   - *BP:* The *base pointer* (BP) stores the address of the base of the *stack frame* (explained below) that is on top of the stack.
   - *Zero:* This word always stores the value 0. It is useful when the program needs to compare a value

against zero.

- *NegOne:* This word always stores the value -1 in two's complement form. It is useful when the program needs to decrement a value; e.g., the SP after a pop operation.

2) VSCPU-specific addresses for interrupt handling and memory-mapped operations. There are 6 of these.
3) General-purpose registers. There are 6 of these. A typical use-case is to pop a couple values from the stack to GP registers, perform a particular operation on these values (e.g. and ADD), then push the result back to the stack from a GP register.

Stack section is used for storing the stack frames (i.e. activation records) of called functions (starts at word 70 in Figure 3). The function whose stack frame is on top of the stack is the currently running, active function. The stack frame of a function contains the information necessary for the function to execute properly. The layout of a frame is shown in the right hand side of Figure 3. At any time, the SP register points to the next available word on the stack. The BP register points to the starting address of the local variables of the active stack frame. This address is referred as the *base address* of the frame. The *return address* (RA) is the address that the function will jump to when it completes execution. The *old BP* is the base address (the value of the BP register) of the caller function's frame. When the active function returns, the BP register's value for the caller function is recovered from the old BP value. The *local variables* are the values of the arguments and the locally declared variables of the function. A local variable is typically accessed by using its offset from the BP. The *temporaries* are the values of intermediate computations that are not assigned to named variables in the program. The stack section grows and shrinks continuously during runtime as functions are called or return, as new local variables are declared, and as compound expressions are computed. The memory layout and the stack frame structure follows a standard approach [20], [21].

The careful reader will notice that there is a jump instruction at address 0 in Figure 3, which we did not include in any of the sections. VSCPU starts executing a given program at address 0. Therefore, as the content of the first word, we emit an unconditional branch instruction that jumps to the main function in the program text section.

V. Peripheral Support for VSCPU

VSCPU could be used to access various peripherals which support different types of peripherals and interfaces such as GPIO, PWM, RS232, VGA, I2C, SPI, etc. We used memory-mapped I/O to access peripherals from VSCPU.

A memory-mapped system employs the address space so that both memory and peripherals can be accessed by the processor (i.e., VSCPU). Hence, when a specific address is accessed by VSCPU using the same instruction set, it can refer to either physical memory or peripherals like switch, push button, LED, accelerometer, etc. Figure 4 shows an example memory map for VSCPU which shows of physical memory and peripherals of this system with their corresponding start and finish addresses.

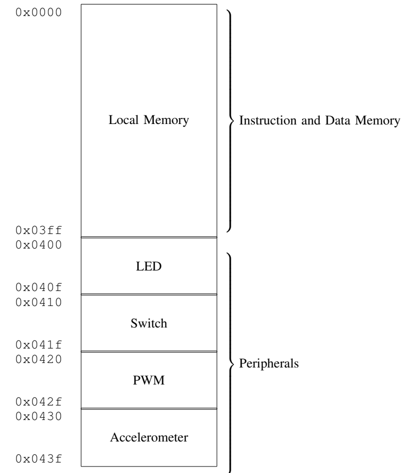

Fig. 4. Example memory map for VSCPU which has access to various peripherals

Figure 5 shows top-level block diagram of memory-mapped VSCPU system. Here, **Core** represents VSCPU itself. **Memory** is a bridge between VSCPU, physical memory, and peripherals. It manages address translation and data transfer between them. **CPU** module is responsible for interfacing to peripherals and host. **Host** module is used for debugging purposes.

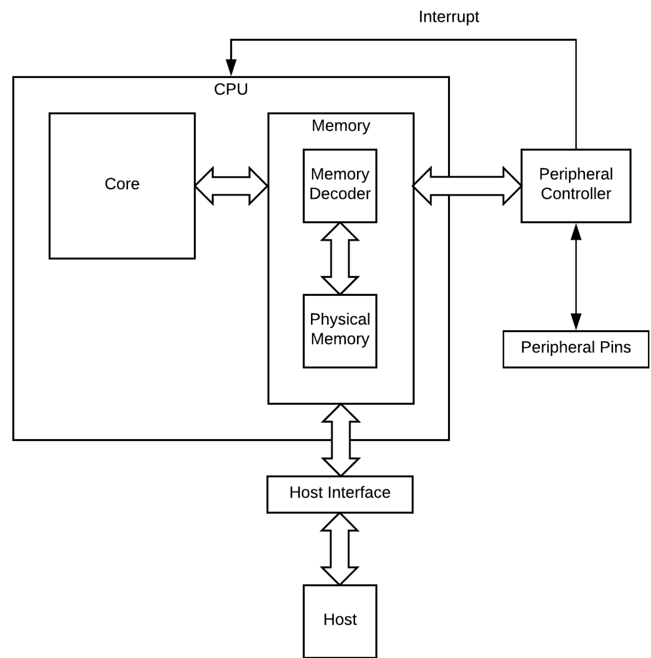

Fig. 5. Top-level block diagram of VSCPU memory-mapped system

VI. Integration of VSCPU as a Course Material into Curriculum

VSCPU has being used for several years within digital design and computer architecture courses at Yeditepe and Özyeğin Universities successfully. Although it is not directly included into the course content, it is given as a short term project which spans three to four weeks. At the beginning of the project, students were given the related documentation along with the instruction set architecture and the toolchain (instruction set simulator, assembler, and C compiler). In the first part of the project, students were asked to design VSCPU by writing its RTL code in Verilog and to perform simulations on PC. This part tests students' understanding of hardware. In the second part of the project, students were asked to use VSCPU they designed to implement a simple memory-mapped system which has access to switches, push buttons, LEDs, VGA port, etc. on an FPGA board. In this part of the project, students were given the RTL code of memory interface, interconnect logic, and peripheral interface and asked them to implement a working memory-mapped system and write programs in either assembly or C which require to access the peripherals. This part tests students' understanding of the difference between hardware and software.

Besides these, two students designed and implemented VSCPU-based line follower robot and sumo robot within their term projects.

VII. Conclusion

This paper proposes a simple and customizable soft CPU which can be easily implemented on FPGAs. Its simple nature makes it an ideal candidate to be used within undergraduate-level computer architecture/organization courses as a teaching material. We have already been using VSCPU at two universities for several years. According to the feedback we receive from students, we can say that understanding of how a modern CPU works first requires to design a simple one.

Acknowledgment

This work is supported by TÜBİTAK under contract 117E090. All documentation, hardware design files (RTL codes) and software source codes are available at https://github.com/MC2SC.

References

[1] Y. Li and W. Chu, "Aizup-a pipelined processor design and implementation on xilinx fpga chip," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*. IEEE, 1996, pp. 98–106.

[2] J. Gray, "Hands-on computer architecture: teaching processor and integrated systems design with fpgas," in *Proceedings of the 2000 workshop on Computer architecture education*. ACM, 2000, p. 17.

[3] D. Šulık, M. Vasilko, and P. Fuchs, "Design of a risc microcontroller core in 48 hours," *Journal of ELECTRICAL ENGINEERING*, vol. 52, no. 5-6, pp. 171–176, 2001.

[4] J. D. Carpinelli, "The very simple cpu simulator," in *Frontiers in Education, 2002. FIE 2002. 32nd Annual*, vol. 1. IEEE, 2002, pp. T2F–T2F.

[5] D. Ellard, D. Holland, N. Murphy, and M. Seltzer, "On the design of a new cpu architecture for pedagogical purposes," in *Proceedings of the 2002 workshop on Computer architecture education: Held in conjunction with the 29th International Symposium on Computer Architecture*. ACM, 2002, p. 6.

[6] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an fpga," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 723–728.

[7] K. Nakano, K. Kawakami, K. Shigemoto, Y. Kamada, and Y. Ito, "A tiny processing system for education and small embedded systems on the fpgas," in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, vol. 2. IEEE, 2008, pp. 472–479.

[8] R. Nakamura, Y. Ito, and K. Nakano, "Tinycse: Tiny computer system for education," in *Computing and Networking (CANDAR), 2013 First International Symposium on*. IEEE, 2013, pp. 639–641.

[9] V. Angelov and V. Lindenstruth, "The educational processor sweet-16," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 555–559.

[10] L. Ribas-Xirgo, "Yet another simple processor (yasp) for introductory courses on computer architecture," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 10, pp. 3317–3323, 2010.

[11] J. Qian, R. Wang, S. Shi, Y. Zhu, and Z. Xie, "Simplifying and integrating experiments of hardware curriculums," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 9. IEEE, 2010, pp. 610–614.

[12] J. H. Lee, S. E. Lee, H. C. Yu, and T. Suh, "Pipelined cpu design with fpga in teaching computer architecture," *IEEE Transactions on Education*, vol. 55, no. 3, pp. 341–348, 2012.

[13] Y. Chen and P. Cao, "Toy cpu: An innovative curriculum design," in *Computer Science & Education (ICCSE), 2012 7th International Conference on*. IEEE, 2012, pp. 1690–1693.

[14] S. Yamazaki, T. Satoh, T. Jiromaru, N. Tachi, and M. Iwano, "Instructional design of a workshop" how a computer works" aimed at improving intuitive comprehension and motivation," in *Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on*. IEEE, 2014, pp. 338–341.

[15] A. Hernandez Zavala, O. Camacho Nieto, J. A. Huerta Ruelas, C. Domínguez, and R. Arodí, "Design of a general purpose 8-bit risc processor for computer architecture learning," *Computación y Sistemas*, vol. 19, no. 2, pp. 371–385, 2015.

[16] https://github.com/abdullahyildiz/VerySimpleCPU-public.

[17] http://verysimplecpu.org/.

[18] http://cpu.tc/.

[19] https://clang.llvm.org/docs/LibTooling.html.

[20] P. Sestoft, *Programming language concepts*. Springer, 2017.

[21] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.