FRI-2G.303-1-CCT1-04

SUPPORTING DEVELOPMENT TOOLS FOR FPGA-BASED TRAINING PROCESSORS

Assoc. Prof. Aneliya Ivanova, PhD
Department of Computing,
"Angel Kanchev" University of Ruse
Phone: 082-888 827
E-mail: aivanova@uni-ruse.bg

Principal Assistant Nikolay Kostadinov, PhD
Department of Computing,
"Angel Kanchev" University of Ruse
E-mail: nkostadinov@uni-ruse.bg

Abstract: *Experiments with programmable logic devices such as Field Programmable Gate Array (FPGA) have found their place in the lab exercises within various computer engineering courses, especially hardware-oriented ones. FPGA-based training processors, developed as course assignments have proved to be a useful educational tool for studying the basic concepts of computer architecture. One of the advantages of this approach is that students can experiment with different microarchitectures while they reconfigure their projects and modify the instruction set, addressing modes, and machine word size. In the Computer Systems and Technologies Bachelor Curriculum, the Design Technology (DT) course is a precursor to the Computer Architecture (CA) course, and in this sense, lab exercises with FPGA-based training processors will establish a fundamental knowledge base that will help the students better understand the more complex concepts taught in the CA course. This paper presents an overview of the major educational tools prepared for DT lab assignments: a family of educational processors, an assembler, and an assembly language simulator. One of the main requirements for the tools is to be customizable in terms of the machine type, instruction set, and machine word size of the target FPGA-based processor. Several examples are given to show how these hardware and software tools are used in the teaching process.*

Keywords: *Computer Architecture, CPU, FPGA-based Processor, Instruction Set, Addressing Mode, Assembler, Simulator.*

INTRODUCTION

FPGA-based models of processors are widely used in the education of students in computer systems and technology-related majors (Larkins et al., 2013, Fouda & Eldeen, 2013, Nakano & Ito, 2008) in courses such as Processor Design and Computer Architectures, for example. The reason for their growing popularity as a teaching tool is not only the complexity of modern real processors, but also the flexibility provided by FPGA devices and the possibility to easily modify the basic training processor models to explore alternative architectures (Zavala et al., 2015).

Establishing active parallel and input-output connections between disciplines in the curriculum is not an extra, but a prerequisite for the effective training of future specialists in computer systems and technologies (Dabu, 2017, Turner et al., 2022). With its specificity, the discipline "Design Technology" reveals favorable opportunities for the formation of active parallel and output connections with the disciplines "Computer Organization" and "Computer Architectures", and this led to the development and implementation in the educational process of exemplary VHDL models of classical machines with von Neumann and Harvard architectures (Ivanova & Kostadinov, 2023). Further, basic models of processors with accumulator, stack, and register organization were developed. These models use Harvard architecture and have the same characteristics regarding the size of instruction memory and data memory, bit-length of the operands, and input/output components. They are simplified enough to provide an initial idea of the processor's functioning and the process of designing, implementing, and testing the corresponding processor core using an FPGA device.

For specification, modeling, simulation, and implementation of the training processors, the integrated development environment Xilinx Vivado Design Suite (Vivado, 2023) is used, and testing is performed using the Basys 3 development kit (Digilent, 2016), including the FPGA device XC7A35T from the Artix-7 family. The kit has 16 switches, 16 LEDs, 4 seven-segment indicators, and 4 user-defined buttons. These components allow input of operands, displaying the result of the current operation, as well as the current state of the finite state machine, which implements the functionality of the control unit, and possibly the number of occupied registers, stack cells, or data memory cells, respectively for processor models with register, stack or accumulator organization. Understandably, the Basys 3 kit does not provide enough capacity to display the contents of general-purpose registers, stack cells, or data memory cells for the three types of processor models mentioned above. This work presents the supporting tools that have been developed for training purposes to provide full-fledged tracing and analysis of the execution of an assembly program by the developed processor models. The discussion is focused in particular on the training processor with register-based Harvard architecture.

EXPOSITION

A training model of a processor with register-based Harvard architecture

The block diagram of the developed processor model is presented in fig. 1.

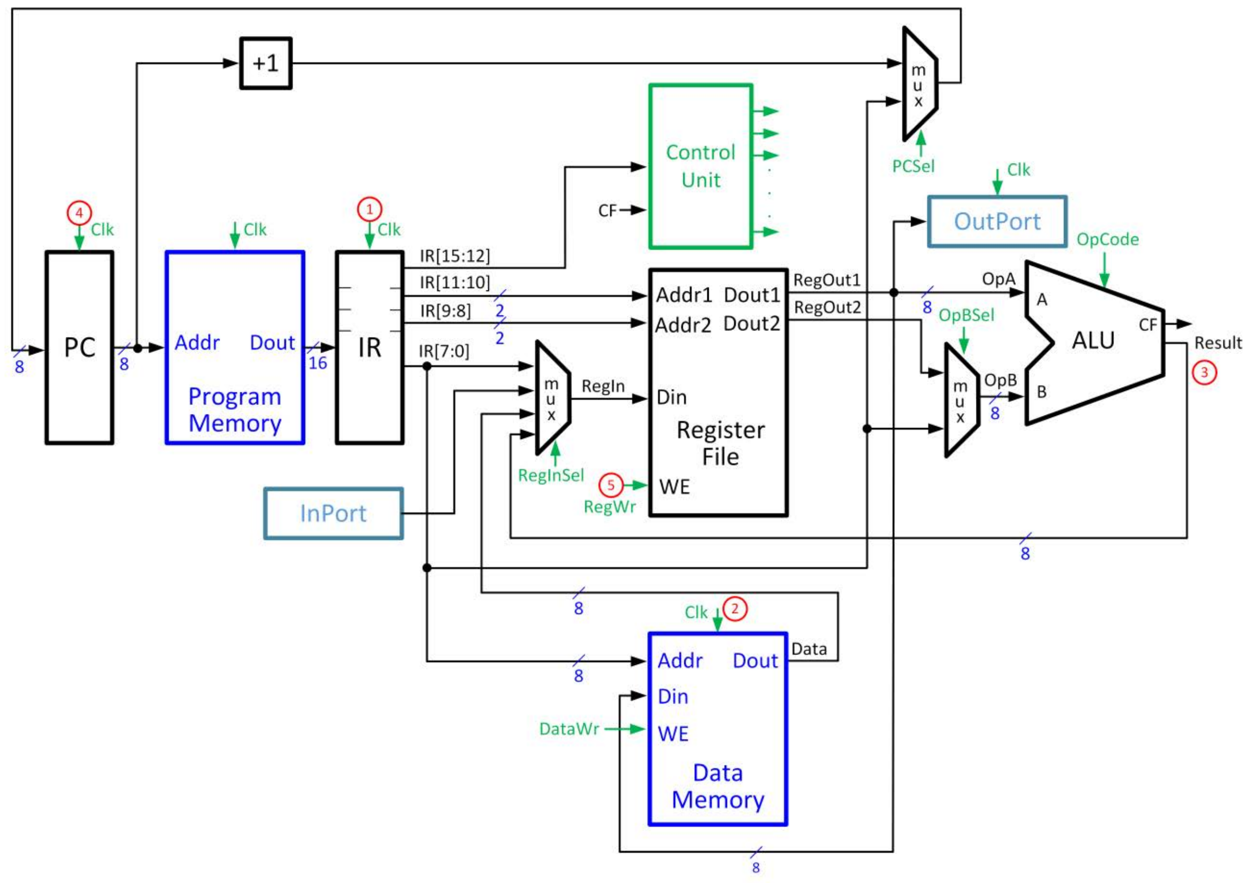


Fig. 1. Block diagram of the training processor

Because of its designation for educational purposes, the bit-length, instruction set, and addressing modes of the training processor model are simplified as much as possible. The processor's register block includes four 8-bit registers (R0, R1, R2 and R3). The output of the ALU is supplemented with a ninth bit, which serves as a carry flag (CF). Two 8-bit ports are provided for data input (InPort) and output (OutPort). The instruction memory and data memory have a capacity of 256 cells each.

The basic instruction set (Table 1) includes 7 instructions. Three addressing modes are provided: direct (absolute), immediate, and register direct.

All instructions have a 16-bit fixed format (Fig. 2). The operation code (opcode) is set with the three most significant bits. The addressing mode is determined by the value of bit m. When m = 0, bits [7:0] of the instruction contain an address from the data memory, and direct addressing mode is implemented, and when m = 1, bits [7:0] contain an immediate operand and immediate addressing mode is implemented. When the instructions require register direct addressing mode, bit m also has a value of 0, but in this case, bits [11:10] of the instruction specify the number of the destination register, and bits [9:8] specify the number of the source register. The lower eight bits of the instruction in this case have no meaning.

Table 1. Basic instruction set

Mnemonic	Syntax	Coding	Semantics
LOAD	LOAD Ri, address	0010[dd]00[address]	Ri ← mem[address]
	LOAD Ri, #imm	0011[imm]	Ri ← imm
STORE	STORE Ri, address	0100[dd]00[address]	mem[address] ← Ri
NOR	NOR Ri, Rj	0110[dd][ss]00000000	Ri ← Ri nor Rj
	NOR Ri, #imm	0111[dd]00[imm]	Ri ← Ri nor imm
ADD	ADD Ri, Rj	1000[dd][ss]00000000	Ri ← Ri + Rj (CF)
	ADD Ri, #imm	1001[dd]00[imm]	Ri ← Ri + imm (CF)
INP	INP Ri	1010[dd]0000000000	Ri ← InPort
OUTP	OUTP Ri	1101[dd]0000000000	OutPort ← Ri
JCC	JCC address	11100000[address]	PC ← address при CF=0

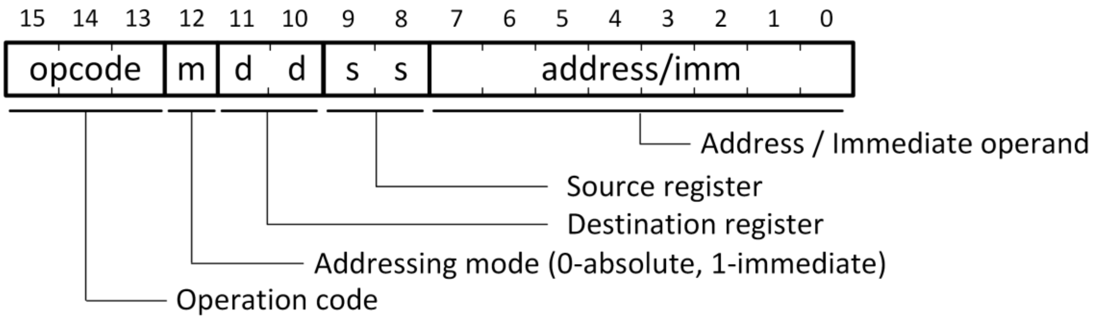


Fig. 2. Instruction format

Each instruction is executed for one machine cycle according to the sequence of steps numbered in Fig. 3:

1) Fetching the instruction from the instruction memory at the address contained in the program counter (PC) and writing the read code to the instruction register (IR). The most significant 4 bits of the instruction (IR[15:13]) are fed to the control unit for decoding, and the next two pairs of bits (IR[11:10], IR[9:8]) address two registers from the register block.

2) Accessing the data memory (only for LOAD and STORE instructions). In the case of a LOAD instruction with direct addressing mode, the read data (Data) is multiplexed to the input (RegIn) of the register block for subsequent write operation. In the case of STORE, the contents of the first register (RegOut1) are stored at the address specified in IR[7:0].

3) Writing the result of the operation performed in the ALU to the Result register.

4) Updating the content of the program counter to point to the address of the next instruction.

5) Writing the data supplied to the RegIn input to the currently available register from the register block. In the case of LOAD instruction, this is the content of Register Block's Data output (when direct addressing mode is set) or IR[7:0] (when immediate addressing mode is set), and in the ADD and NOR instructions - the content of ALU's Result output. In this step, reading from a port to a register is also performed (INP instruction).

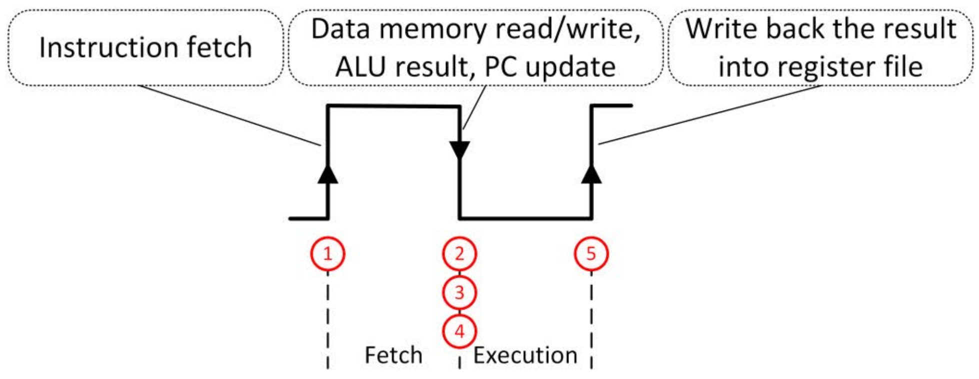

Instruction fetch
Data memory read/write, ALU result, PC update
Write back the result into register file
1
2
3
4
5
Fetch
Execution

Fig. 3. Timing diagram of instruction cycle

Assembler

Students have not only the opportunity to write different programs with the available instructions but also to experiment by adding new ones to the basic instruction set, as well as new addressing modes. In addition, since they are experimenting with programmable logic, students can also change the number of registers in the register block and their bit length. This requires the development of an assembler that is fully configurable concerning the instruction set of the target processor. Fig. 4 presents the grammar of the assembly language specific to the model of the processor with register architecture.

<RegAssemblyProg> → *<Line>* *<RegAssemblyProg>* | *<null>*
<Line> → *<Directive>* | *<LabelledInstruction>* | *<Instruction>*
<Directive> → **IDENT EQU CONST**
<LabelledInstruction> → **LABEL :** *<Instruction>*
<Instruction> → **INSTR REG** , *<Oper>* | **INSTR REG , REG** | **INSTR REG** | **INSTR LABEL**
<Oper> → *<Expr>* | *<ImmExpr>*
<ImmExpr> → **#** *<Expr>*
<Expr> → **IDENT** | **CONST**

Fig. 4. Formal grammar of the assembler

In this case, the terminal class INSTR summarizes the mnemonic codes of the instructions supported by the assembler. The grammar defined in this way is instruction set independent. Another important requirement is that the assembler should be able to adapt when changing the bit-length of the instruction fields, respectively the length of the instructions. To satisfy the specified requirements, the assembler is parameterized by a configuration file, a fragment of which is shown in Fig. 5. As can be seen, for each instruction its mnemonic representation, the operation code, as well as a code specifying the permissible addressing modes are defined. When students add new instructions and addressing modes, they must also edit this configuration file, not only adding the mnemonic codes of the new instructions with the operation codes and addressing modes, but they must also indicate the new bit-length of the fields in the instruction code, namely: for the opcode, addressing mode, register number in the register block and immediate operand.

```
{
 "RegMachineConfig": {
  "Instructions": [
   {"Mnemocode": "LOAD",     "Opcode": "1", "AddressingMode": "0"},
   {"Mnemocode": "LOAD",     "Opcode": "1", "AddressingMode": "1"},
   {"Mnemocode": "STORE",    "Opcode": "2", "AddressingMode": "0"},
   {"Mnemocode": "NOR",      "Opcode": "3", "AddressingMode": "0"},
   {"Mnemocode": "NOR",      "Opcode": "3", "AddressingMode": "1"},
    ...
   {"Mnemocode": "JCC",      "Opcode": "7", "AddressingMode": "0"}
  ],
  "InstructionFields": {
   "OpcodeFieldLength": "3",
   "AddresingModeFieldLength": "1",
   "DestRegFieldLength": "2",
   "SourceRegFieldLength": "2",
   "OperandFieldLength": "8"
  }}}
```

Fig. 5. Configuration file of the assembler

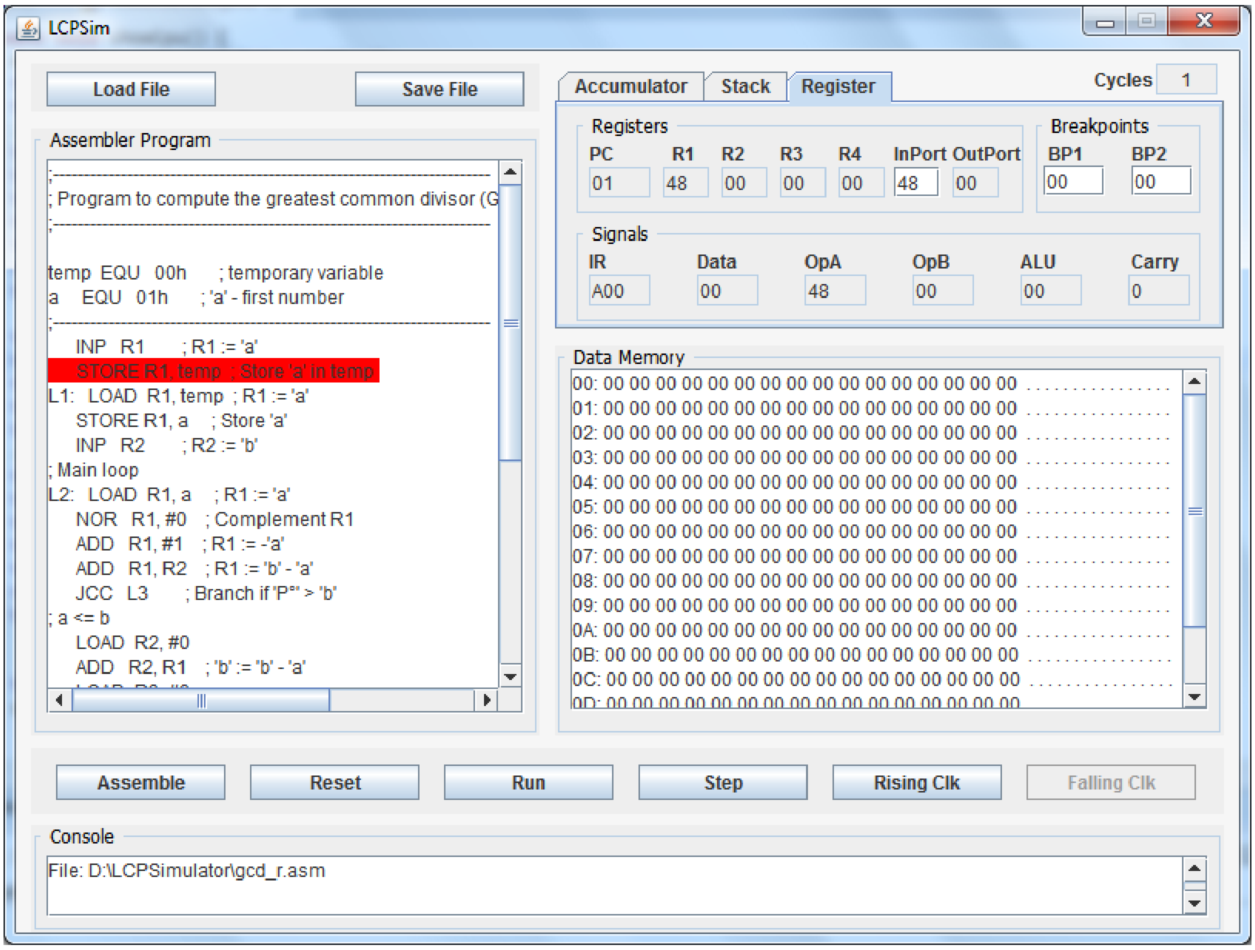
The processing of the input file containing the code of the assembly program is performed in two passes. In the first pass, in parallel with the check for syntax correctness, a symbol table is created, which contains the correspondence of the identifiers and labels with their addresses in memory. At this stage, a check is also made for duplication of identifiers and labels. During the second pass, upon reaching a line containing an instruction, a check is made whether its mnemonic code is defined in the configuration table, i.e. whether the instruction is valid. When an identifier or label is encountered, their corresponding values are extracted from the symbol table. Semantic analysis in this case is reduced to checking whether the instruction supports the specific addressing type, as well as whether the operand type is compatible with it. If there are no errors, the assembler generates the machine code, an assembly listing, and a VHDL description of the segment for initialization of the program memory.

Simulator

To fully visualize the execution of the user program by the training model of the processor, the LCPSim simulator has been developed as a supporting tool. As can be seen from Fig. 6, the user interface of the simulator provides separate sections for the three basic architectures (accumulator, stack, and register), with which the students will experiment. It is possible to visualize not only the execution of the instructions but also the components of the microarchitecture of the studied processor. For this purpose, in addition to the contents of the registers and data memory, the state of the internal signals of the operational part is also visualized, which allows for tracing the data flow during the execution of the instructions.

After loading from a file, or after directly entering the program code, the latter can be executed in three modes. Students can set breakpoints, execute the program completely, or step by step. In addition, unlike conventional simulators, the developed simulator allows students to control at which level of the clock signal the observed values are to be registered. In this way, changes in the state of signals and registers caused by the rising or falling edge of the clock signal can be tracked and analyzed. The addition of this mode aims to provide a more detailed visualization of the actions that are performed in the sequential steps of the "fetch-decode-execute" cycle of the instruction.

To check the syntax correctness of the program, an "Assemble" button is provided. This button calls the assembler and creates a symbol table with the actual addresses of the labels and variables.



LCPSim

Load File | Save File

Assembler Program

```
;--------------------------------------------------------------
; Program to compute the greatest common divisor (G
;--------------------------------------------------------------


temp  EQU   00h      ; temporary variable
a     EQU   01h      ; 'a' - first number
;--------------------------------------------------------------
    INP   R1        ; R1 := 'a'
    STORE R1, temp  ; Store 'a' in temp
L1:  LOAD  R1, temp  ; R1 := 'a'
    STORE R1, a     ; Store 'a'
    INP   R2        ; R2 := 'b'
; Main loop
L2:  LOAD  R1, a     ; R1 := 'a'
    NOR   R1, #0    ; Complement R1
    ADD   R1, #1    ; R1 := -'a'
    ADD   R1, R2    ; R1 := 'b' - 'a'
    JCC   L3        ; Branch if 'P"' > 'b'
; a <= b
    LOAD  R2, #0
    ADD   R2, R1    ; 'b' := 'b' - 'a'
```

Accumulator | Stack | Register — Cycles 1

Registers

PC	R1	R2	R3	R4	InPort	OutPort
01	48	00	00	00	48	00

Breakpoints

BP1	BP2
00	00

Signals

IR	Data	OpA	OpB	ALU	Carry
A00	00	48	00	00	0

Data Memory

```
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
02: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
03: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
04: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
05: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
06: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
07: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
08: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
09: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0B: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0D: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Assemble | Reset | Run | Step | Rising Clk | Falling Clk

Console

```
File: D:\LCPSimulator\gcd_r.asm
```

Fig. 6. User interface of the simulator

CONCLUSION

Experiments with FPGA-based training models of processors with the three classical architectures – accumulator, register, and stack-based will help students not only to master the main stages in the design of digital devices using programmable logic and hardware modeling languages, but also to better understand the specifics of these architectures and, accordingly, to be better prepared for the upcoming study of computer architectures.

Thanks to the use of programmable logic, students have the opportunity to make quick and easy changes to the instruction bit length, program counter, operands, and registers, as well as the number of registers in the register block. This allows for a variety of experiments that will contribute to a better understanding of computer architectures and the organization of the processor.

The specifically developed assembler for the training model of a processor with Harvard register-based architecture allows students to modify the instruction set, addressing modes, the number of registers in the register block, and their bit-length through elementary edits in a special configuration file.

The simulator's ability to controllably track and analyze changes in the state of signals and registers caused by the rising and falling edge of the clock signal contributes to an easier understanding of the actions performed in the sequential steps of the instruction's "fetch-decode-execute" cycle.

ACKNOWLEDGEMENT

This work is published with the support of project 24-FEEA-01 "Methods and tools for automated analysis, processing, storage and management of multimedia data and documents", financed by the "Scientific Research" fund of Ruse University "Angel Kanchev".

REFERENCES

Digilent (2016) https://digilent.com/reference/_media/basys3:%20basys3_rm.pdf

Fouda, A.M., A.B.Eldeen. (2013) Design modified architecture for MCS-51 with innovated instructions based on VHDL. Ain Shams Engineering Journal, Vol. 4, Issue 4, pp. 723-733, ISSN 2090-4479, https://doi.org/10.1016/j.asej.2012.12.001.

Ivanova, A., N. Kostadinov (2023). Educational VHDL Models of Processors with Von Neumann and Harvard Architectures PROCEEDINGS OF UNIVERSITY OF RUSE, volume 62, book 3.2, pp. 60-66.

Larkins, D.B., Jones, W.M., & Rickard, H.E. (2013). *Using FPGAs as a reconfigurable teaching tool throughout CS systems curriculum*. Technical Symposium on Computer Science Education.

Nakano, K. & Y. Ito. (2008). *Processor, Assembler, and Compiler Design Education Using an FPGA*. In Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS '08). IEEE Computer Society, USA, 723–728.

Turner, R., Cotton, D., Morrison, D., & Kneale, P. (2022). Embedding interdisciplinary learning into the first-year undergraduate curriculum: drivers and barriers in a cross-institutional enhancement project. Teaching in Higher Education, 29(4), 1092–1108. https://doi.org/10.1080/13562517.2022.2056834

Vivado (2023) https://www.xilinx.com/products/design-tools/vivado.html

Dabu, C.-M. (2017). Computer Science Education and Interdisciplinarity. InTech. doi: 10.5772/intechopen.68580

Zavala, A.H., O.C. Nieto, J.A. Huerta Ruelas, A.R. Carvallo Domínguez. (2015). *Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning*. Computación y Sistemas, Vol. 19, No. 2, 2015, pp. 371–385, ISSN 1405-5546, doi: 10.13053/CyS-19-2-1941.