 Acta Univ. Sapientiae Informatica **12,** 2 (2020) 159–182

DOI: 10.2478/ausi-2020-0010

Development of ATmega 328P micro-controller emulator for educational purposes

Michal ŠIPOŠ
IBM Slovakia, Ltd., branch office Košice
Aupark Tower, Protifašistických
bojovníkov 11, Košice, Slovak Republic
email: michal.sipos@ibm.com

Slavomír ŠIMOŇÁK
Technical University of Košice
Košice, Slovak Republic
email: slavomir.simonak@tuke.sk

Abstract. The paper presents some of our recent results in the field of computer emulation for supporting and enhancing the educational processes. The ATmega 328P micro-controller emulator has been developed as a set of emuStudio emulation platform extension modules (plug-ins). The platform is used at the Department of Computers and Informatics as a studying and teaching support tool. Within the Assembler course, currently, the Intel 8080 architecture and language is briefly described as a preliminary preparation material for the study of Intel x86 architecture, and the Intel 8080 emuStudio emulator module is used here. The aim of this work is to explore the possibility to enrich the course by introducing a more up-to-date and relevant technology and the ATmega is the heart of Arduino – a popular hardware and software prototyping platform. We consider the options to make the process of studying the assembly language principles more attractive for students and using the ATmega AVR architecture, which is broadly deployed in embedded systems, seems to be one of them.

Computing Classification System 1998: K.3.2, C.1.0
Mathematics Subject Classification 2010: 68U20, 68M01
Key words and phrases: emuStudio, emulation, Atmega, Arduino

1 Introduction

Emulation [13] can be described as a technique of imitating a software or hardware product by another software [21]. Emulation currently is widely used mainly as a technique for running a software written for computer system different from the host computer operating environment. It provides the possibility of cross-platform compatibility between different computer systems [31], so it can also be considered as a preservation strategy for digital content [32], [33]. Our motivation behind the development of the ATmega emulator was slightly different however, as it was mainly intended for educational purposes.

At the present time, there are many emulators of computer systems available [34]. Choosing the emuStudio as the platform for which we developed our emulator was straightforward, since it provides many features, which are essential for its successful application in educational process. emuStudio is a platform for emulation of computer architectures that integrates, as a form of an IDE, also source code editing, compiling and debugging features.

Within the platform, programs for emulated machines are usually written using assembly language of the particular architecture. Another significant advantage of the emuStudio is the fact that it does not only serve as a one-purpose emulator. Essentially, it provides a framework[1] in the form of a Java API. By utilizing it, programmers are enabled to design and implement their own computer emulators as a set of plug-in modules. The above-mentioned framework is intended to help to standardize the process of emulation [11], i.e. to define the key responsibilities, functionalities and types of components that are common for most emulators. In particular, the task of a programmer is to implement modules for assembler source code compilation, CPU and memory emulation, but optional peripheral devices can also be emulated.

With its ability of illustrative exploring the internal operation of emulated architecture, emuStudio is well suited for educational purposes. The platform has been designed to be easily extendable and once a component is implemented, it can be reused effectively. As a result, possibilities of enriching the emuStudio by new modules are very broad.

At the time of writing this paper, the emuStudio is used at the Department of Computers and Informatics in Assembler and Data Structures and Algorithms courses. In the Assembler course, Intel 8080 emulator is utilized and within the Data Structures and Algorithms course, emulators of RAM and RASP [29] abstract machines are used. The 8080 as a predecessor of x86 ar-

[1] http://www.emustudio.net

chitecture and a relevant example of an 8-bit processor architecture is used as a simple preparation for studying Intel x86 architecture and language. However, being introduced to the microprocessors market in 1974 [20], despite of its impact on the industry, it can cause some students to become slightly demotivated by using a less up-to-date technology.

In the thesis [18], an idea of incorporating Arduino with its ATmega micro-controller instead of the Intel 8080 in the assembly language course curriculum has been investigated. Some similarities between Intel 8080 and ATmega 328P assembly languages has been observed and the possibilities of such substitution are discussed there. Furthermore a simple library providing students with the possibility to use basic input and output operations has been developed within the thesis and further enhanced later.

2 Related work

ATmega is the core of the popular Arduino Uno platform[2]. In recent years, Arduino has been introduced to several computer science courses at the Department of Computers and Informatics. This tendency has also been stimulated by the growing demand for knowledge in the area of *embedded systems* [1]; the popularity of the term *Internet of Things* [19] can be observed for being rapidly increasing, too.

Globally, we can see that teaching assembler courses at some universities is continually being shifted aside in favor of new subjects. Nowadays, as the author of [17] suggests, operating systems and virtual machines on the top of them pose a certain problem for teachers and students when it comes to studying internal operation of computer systems as they act as a form of a shield that hides the inner mechanisms. It can be concluded that it is one of the causes for the phenomena of leaving assembler courses out from the curriculum that students might not see enough reasons for studying it.

On the other hand, when using embedded systems, a virtual machine as an abstraction layer is often missing and direct access to hardware is much easier, but it requires some knowledge of its internal operation. And this is where, in addition to C language, assembler becomes much more relevant. The topic is deeper discussed in [17].

An approach similar to the one presented in this paper (but based on 8051 micro-controller emulation) has been taken with the EdSim51 - the 8051 simulator for teachers and students [25]. The EdSim51 is a simulator of 8051

[2] https://store.arduino.cc/arduino-uno-rev3

micro-controller, which is interfaced with virtual peripheral devices like keypad, DC motor, 7-segment display, UART, LEDs, etc. A nice advantage of the EdSim51 over some other available 8051 simulators is that it provides graphical representation of several peripheral devices, which can be used interactively [26]. It is a Java-based application, so it can be used in multiple operating environments.

Another example of very popular processor within the computer architecture academic community and one often utilized for educational purposes [22] is MIPS. Several simulators have been developed to date for this architecture [37]. We can mention DrMIPS, an educational MIPS simulator [23, 22], which can simulate the execution of an assembly program and display the datapath graphically. Moreover it can display the values of inputs and outputs of several components, which are relevant for the execution of the current instruction. The simulator was developed in Java and is available not only for PC, but also for Android devices. On the other hand, since the emulator is developed mainly for educational purposes, it supports rather limited set of instructions. Several instructions like syscalls, floating-point operations and shifts are not supported [23].

CPUlator is a Nios II, ARMv7, and MIPS computer system simulator and debugger running in a web browser [37]. It allows running and debugging programs without corresponding hardware board. Systems simulated by CPUlator are based on the computer systems from the Altera University Program (Nios II, ARMv7) and SPIM (MIPS) respectively.

Gerd's AVR simulator [27] is a complex solution for simulating AVR 8-bit micro-controllers. It provides an editor, assembler, simulator, overview of I/O ports and timers, memories, etc. Lazarus Pascal source code and executable 64-bit versions for Windows and for Linux are available from [27]. The advantage of this solution is the support for many types of AVR 8-bit micro-controllers and complex support of built-in peripheral devices. As an advantage of our solution, described in this paper, can be considered the fact that it is developed using Java and thus could be more portable. It is developed as a set of plug-in modules for emuStudio emulation platform, so it could be easier to develop and enhance in the future. Modules for some simple external devices are also available within our solution (like USART terminal and LED diode emulation modules).

An interesting approach has been employed in [24], where a methodology is proposed for teaching the microprocessors interface course based on the idea of emulating the microprocessor operation. Students are instructed to use the parallel port for emulating the signals generated by 8088/8086 micro-

processors. According to authors of the paper, emulating a working system by generating the necessary control signals leads to a good knowledge of the system.

An educational approach for bridging the gap between low-level and higher level programming, based on usage of 8-bit microcontrollers has been proposed in [6]. The approach proposed aims in simplification the students' learning by making the parallelization between the assembly language programming and higher level programming.

But what is the purpose of using an emulator in addition to original hardware? When studying the above-mentioned inner workings of computers, emulation tools can be a way of a deeper insight into them [36], as the complexity of hardware might sometimes exceed the limit of what can be effectively studied or taught [38], respectively. Also, emulator gives students the opportunity to exercise what they have learned at the classes, at home, without owning the hardware components physically. What is more, emulating an ATmega device in software brings another advantage – it can be used to automate the testing of students' assignments, as has also been suggested in [18].

In addition to using emulators, there are also other efficient approaches that are used within the education in the field of computer architecture and organization, such as hardware-description languages and reconfigurable circuits [7, 16]. An interesting case study is described in [8] where critical investigation of existing course on digital electronics revealed that it mostly produces surface understanding of digital systems and students lack practical skills to develop complex digital designs. The course outline has been improved and an enhanced delivery method was proposed. The study of students' performance over a period of six years was evaluated and the results indicate that students had developed good level of understanding of basic principles and were able to employ system modeling using VHDL.

3 Emulation techniques

Before proceeding to description of the design of ATmega emulator modules set, several emulation-related terms will be introduced. According to the study [21] two basic types of emulation can be distinguished, as they correspond to the format of the emulated program:

- *interpretation* – the program is represented in the form that is native for the original processor it has been written for; this enables the emulator, in this case *interpreter*, to work in the "fetch-decode-execute" loop,

- *binary translation* – the code of the program is translated into the form native for the architecture it is being emulated on, which usually means x86-like binary. The translation, can either take place at runtime (dynamic) or in advance, before emulation (static).

In the study also advantages and disadvantages of both of the approaches are mentioned. Interpretation is generally less complex to implement than binary translation. On the contrary, as during binary translation a native code is generated, the resulting performance of the emulation is usually better. One of the techniques to increase the performance of interpreter emulators is so-called *threaded code*. James R. Bell explains its basic principles in his paper [5]. The algorithm begins by reading the instruction at the PC-th address from the emulated program storage, where the PC is the program counter register. The retrieved opcode serves as an index into a so-called jump table. Each particular table entry contains a reference to the function that implements the corresponding instruction emulation. The only step left is to call the function, which is analogous to executing the instruction on the emulator. The jump table can be filled by function references in advance, when the emulator is initialized, which unburdens the emulation loop from the time consuming task of decoding instruction opcodes.

By performing the above-mentioned procedure, we have eliminated the effort needed for the *decode* step from the *fetch-decode-execute* loop. In addition to this, it is also possible to minimize the *fetch* part, if we cache the instructions already fetched from the memory. These optimizations lead to the algorithm presented in Figure 1 [30].

4 ATmega 328P micro-controller architecture

ATmega 328P micro-controller is based on the AVR architecture, which is one of the leading 8-bit architectures [4]. From instructions complexity point of view, it is a RISC (Reduced Instruction Set Computer) architecture [4]. From the perspective of an emulator programmer, this can be considered as an advantage as it is characteristic for RISC computers to have simpler instructions. The fixed-length instructions also make the decoding part of the execution loop less complex. On the other hand, from the point of view of the one developing programs for the micro-controller, the CISC (Complex Instruction Set Computer) addressing modes flexibility might be missing. Also, a program size might be bigger in case of RISC binary [35] in comparison to a

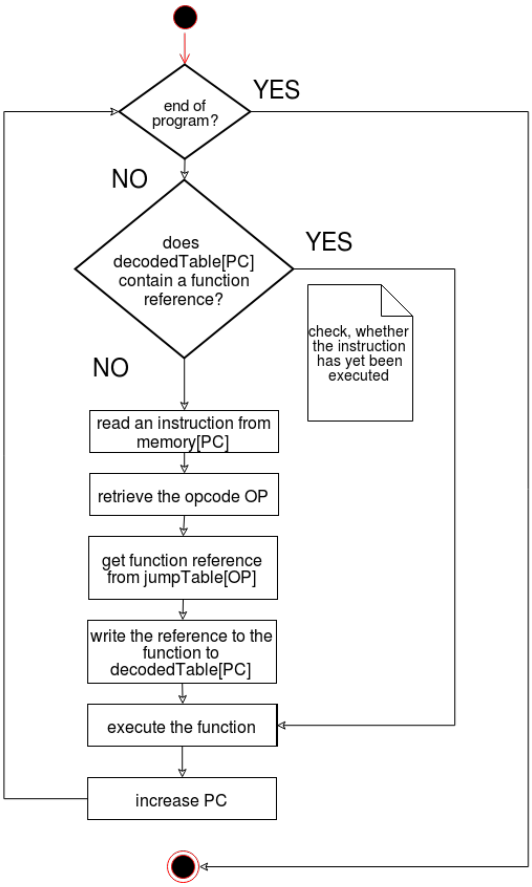


Figure 1: *Threaded code* execution algorithm.

CISC one as for one operation, a sequence of several simpler instructions can be required.

ATmega is a representative of Harvard architecture [4], which affects the structure of the memory subsystem – separate modules are used for storing program and data. When compared to von Neumann architecture, Harvard computers enable more effective pipelined execution [9] – while one instruction is being executed and is retrieving its operands from the data memory, the next one can be pre-fetched from the program memory module.

The ATmega 328P micro-controller is equipped with a 32 kilobyte flash program memory [2]. It is organized in a less conventional way – each memory cell contains an instruction word, i.e. two bytes. The reason for such a distribution

[2] is that the length of all opcodes is either 16 or 32 bits. As a program is addressed by words, program counter is 14 bits wide, which is enough for the whole range.

In case of the data memory, addressing is conventional – one byte for one cell. ATmega chip includes a SRAM (Static Random Access Memory) data memory module with all general purpose registers (GPR) and input/output registers (IOR) mapped to its address space. The organization is depicted in Figure 2 and the capacity of the internal SRAM, i.e. excluding the GPRs and IORs, is 2 KB.

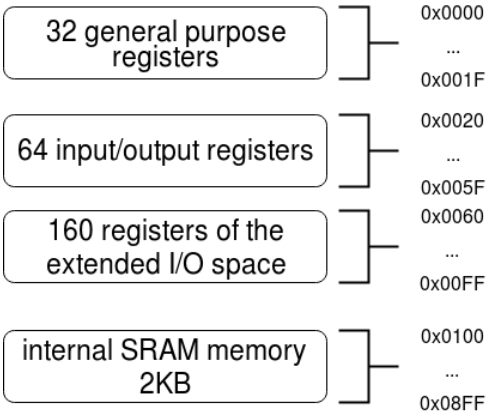


Figure 2: Data memory organization.

An inevitable part of the micro-controller is the input/output subsystem. On the ATmega, it includes digital and analog input/output pins, USART (Universal Synchronous-Asynchronous Receiver/Transmitter) serial interface, timers and a lot of other peripherals. The peripheral devices communication with the micro-controller is realized by using of so called *ports*. They serve as *gates* [28] between the CPU core and the other parts of the micro-controller, or between the CPU and devices out of the chip. These might be various types of sensors or mechanical equipment, e.g. servo motors or relays. For digital input/output, there are three ports – B, C and D available. Each of those ports is controlled by three registers: the DDR register for determining whether the pin has INPUT or OUTPUT direction, the PORT register for setting the pin to HIGH or LOW level, and the PIN register for reading the state of INPUT pins (or toggling the value of particular PORT bit by writing '1' to the corresponding bit of the PIN register) [2]. So to send data e.g. to port B, writing to the register PORTB is required. Each of its bits represents

the corresponding pin of the port. To read its current value, the PINB register can be read and the relevant bit needs to be evaluated.

5 ATmega 328P - emuStudio extensions design

The initial step to consider when designing the support for a new architecture in emuStudio is so called *abstract scheme*[3]. The scheme is always based on the von Neumann model [12], which represents a certain complication in our design as the ATmega is a computer of the Harvard type – it contains separate memory modules for data and for program. Furthermore, also EEPROM (Electrically Erasable Programmable Read-Only Memory) memory module for permanent data storage is included on the chip.

5.1 Memory emulation

There has been a discussion with the author of the emuStudio platform regarding the memory subsystem, in which several possible solutions has been considered. One of them was to extend the emuStudio to support Harvard architecture computers by adding a new type of component in the abstract scheme – program memory. However, such component would be limited only to store programs, which would mean to narrow the functionality of a storage component only for this purpose. Another considered option was to use the concept of a Memory Management Unit (MMU) that would provide access to all the three memory modules. This would, however, require them to be mapped to a single memory space.

Finally, the solution reducing Harvard architecture of ATmega to von Neumann type has been chosen. To perform this, one of the memory modules needed to be considered as *memory*, since von Neumann computers, and therefore also emuStudio abstract scheme include at most one memory. For practical purposes, such as the fact that the *compiler* module needs storage to load a program to and also that the *CPU* needs it to fetch instructions, flash program memory has been chosen.

For the data memory, as it could not be considered as *memory* after this decision, it has been chosen to include it within the *CPU* module. Such an organization is a compromise. Both of the memory modules are integral parts of the chip on a real device, so including data memory in the *CPU* module is acceptable. As for the program memory, it has been chosen, from the logical

[3] http://www.emustudio.net/docuser/main_module/index/

point of view of a von Neumann computer, as the only memory in the scheme. EEPROM data permanent storage is not implemented within our solution yet, however, in the future, it can be added as a *device*, i.e. peripheral device type of module.

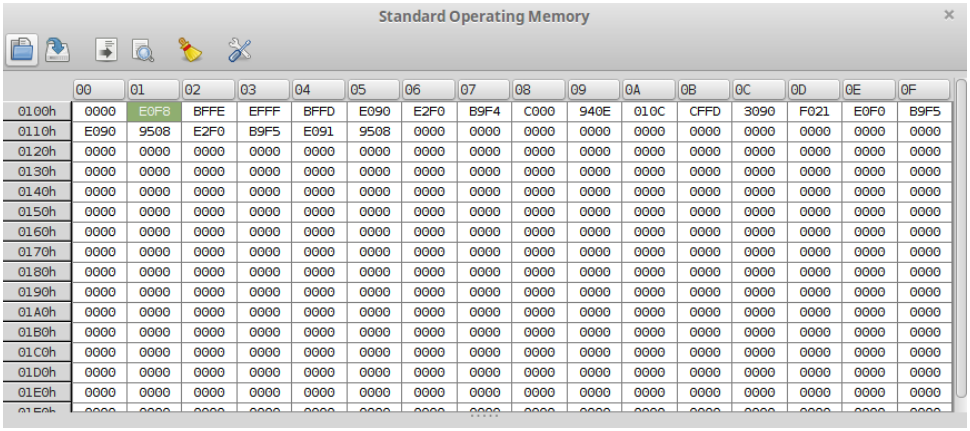
For the purpose of storing program, *Standard Memory*, an already implemented emuStudio module has been chosen. This component has been part of MITS Altair 8800 computer emulator and the fact makes it a sufficient candidate for reuse. Though, we had to adapt it for two-byte cells addressing. Updates in *Standard Memory* module could have been done, for example, by designing a new *context* to support different cell sizes. This would, however, require changes in the *emuLib*[4] library. Also, an already existing utility *HexfileManager*, responsible for loading contents of an Intel HEX files into memory, would be needed to be adapted as it currently depends on memories with one byte at a cell.

Therefore, we had to choose a solution with as little changes in *Standard Memory* as possible. It has been decided to adapt just the visual representation of the memory content in the GUI (Graphical User Interface). As a result, internally, in our solution, the content is still an array of one byte values, while in the GUI, user sees two bytes at a cell. The interface of the *StandardMemory* context has been extended by the option to set the cell size by a programmer. He or she only needs to do this configuration when requesting it from the *context pool*. Also, endianness can be set. The resulting view of the memory content can be seen in Figure 3.

5.2 Compiler module

Another module that is part of the set presented in this paper is the *compiler* module. All compilers currently present in emuStudio use automation tools to generate lexical (lexer) and syntactic (parser) [10] analyzer. For lexical analysis, JFlex[5] is used. This phase of compilation is also needed for the purpose of syntax highlighting in the emuStudio source code editor. Thanks to lexical analyzer generated on the basis of the JFlex specification file, the emuStudio main module takes care of this task, we only needed to specify the types of tokens. This fragment from the JFlex file illustrates it:

[4] https://github.com/vbmacher/emuLib
[5] http://jflex.de/



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0100h	0000	E0F8	BFFE	EFFF	BFFD	E090	E2F0	B9F4	C000	940E	010C	CFFD	3090	F021	E0F0	B9F5
0110h	E090	9508	E2F0	B9F5	E091	9508	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0120h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0130h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0140h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0150h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0160h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0170h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0180h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0190h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
01A0h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
01B0h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
01C0h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
01D0h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
01E0h	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Figure 3: GUI of *StandardMemory* after adding a support for different cell sizes.

```
"ADD" {
    return token(Token.RESERVED);
}

".DB" {
    return token(Token.PREPROCESSOR);
}
```

For the core task of compiling assembler source code of programs written for ATmega, we call an external tool – GAVRASM[6] from the compiler module. From Java code, it is invoked by utilizing *ProcessBuilder* provided by the Java API:

```
ProcessBuilder processBuilder = new ProcessBuilder(command);
Process process = processBuilder.start();
process.waitFor();
```

The `command` depends on the underlying operating system, since the GAVRASM is available for MS Windows as well as for Linux OS.

[6] http://www.avr-asm-tutorial.net/gavrasm/index_en.html

5.3 CPU emulator module

Within this subsection we will continue to the core part of the emulator – the *CPU* module. The basic algorithm of a CPU emulator in general is presented in [21]. The diagram is depicted in Figure 4. The algorithm will also be applied in our solution.

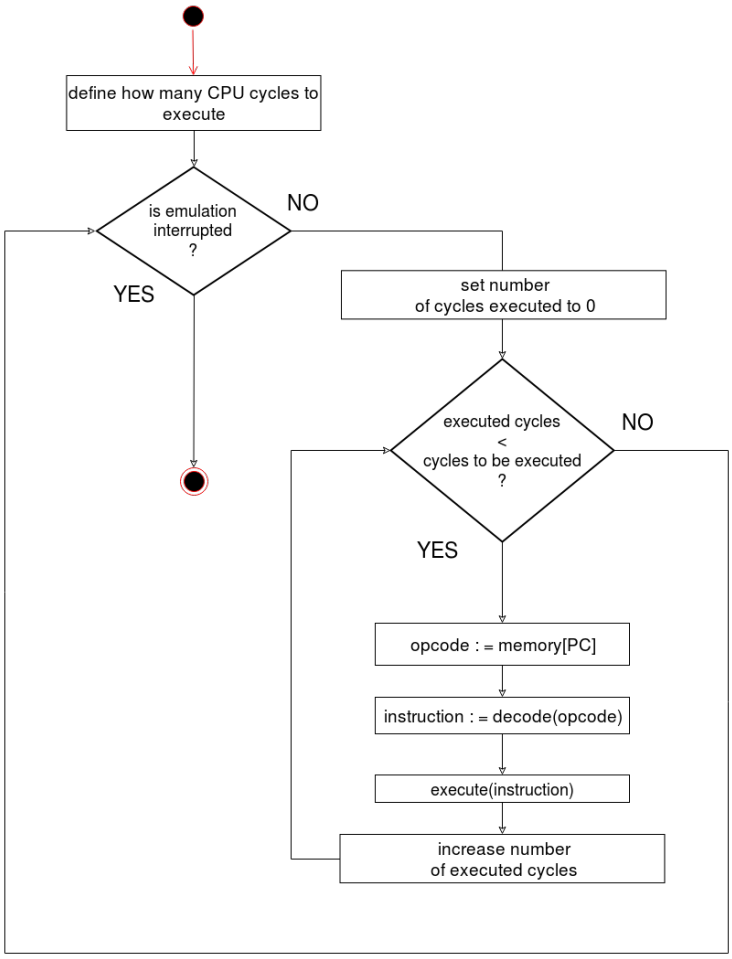


Figure 4: General CPU core emulator algorithm.

Let us explore the steps in deeper detail and discuss how they will be implemented. At first, we need to determine how many CPU clock cycles will be executed. For this purpose, it is necessary to define a *synchronization interval*

for which the number will be calculated. The time range will be figured out by the following formula [15]:

$$\mathrm{timeSlice} = \mathrm{T} \cdot \mathrm{numberOfCycles}\,. \tag{1}$$

This will be equal, as the formula suggests, to the time of execution of *numberOfCycles* clock cycles on the real CPU; T is the clock cycle period, i.e. multiplicative inverse of the CPU frequency. Therefore, *timeSlice* will be calculated according to this formula:

$$\mathrm{timeSlice} = \frac{\mathrm{numberOfCycles}}{\mathrm{f}}\,. \tag{2}$$

The *numberOfCycles* value can be then calculated as follows:

$$\mathrm{numberOfCycles} = \mathrm{timeSlice} \cdot \mathrm{f}\,. \tag{3}$$

Now, what is still missing is the *timeSlice* value. It can be an empirically chosen constant [15]. In the Intel 8080 emuStudio extension, it is 100 ms (0.1 s). We will also use this time range and assuming that clock frequency of ATmega 328P is 16 MHz[7], the following formula holds:

$$\mathrm{numberOfCycles} = 0.1 \cdot 16 \cdot 10^6 = 16 \cdot 10^5\,. \tag{4}$$

The next step is to determine whether the number of executed cycles is less than how many are to be executed at all within the 100 ms interval. To evaluate this condition, we need to know the number of clock cycles of specific instructions. These are constant values and can be obtained from the AVR Instruction Set Manual [3]. To keep the number of cycles executed so far, we need to return the number of clock cycles that the instruction execution took from the `execute(instruction)` method.

The following part of the algorithm is reading the instruction `opcode` from the program memory. After fetching the instruction, we need to decode it. This can be done by using an extensive *switch* statement for all the 131 instructions, not even counting all the possible combinations of opcodes that have their operand encoded in them. The idea is illustrated by the following code fragment:

[7] https://www.arduino.cc/en/Products/Compare

```
switch(opcode){
 case 0x0:
 execute(instruction1);
 case 0x1:
 execute(instruction2);
 ...
}
```

In contrast to this, it is more efficient to use the concept of a *jump table*, that has been introduced in the *Introduction* section and can be seen in the Figure 1. In terms of Java, which is the implementation language of our solution, we can use so called *functional interface* for this purpose, let us call it `ExecutableInstruction`, with one method – `execute(Short[] opcodeWord)` that takes a two-bytes opcode and returns the number of CPU clock cycles executed by the particular instruction. The *jumpTable* needs to be initialized. It is relevant to do so outside of the emulation loop itself and have it prepared in advance. The number of all possible distinct opcodes is

$$\mathrm{numOfDistinctOpcodes} = 2^{16} = 65536\,. \tag{5}$$

During the emulator initialization, we will go through all these possibilities, i.e. from 0 to 65535, and in all the *case* branches of the *switch* statement, we will apply assignments in the following form:

```
executableInstructions[OP] = EmulatorEngine.this::add;
```

As we can see, a reference to the method `add()` that implements emulation of the ADD instruction is put at the *OP-th* position in the `executableInstructions[]` array, which represents the jump table and the *OP* is the operation code of the instruction. The last step of the algorithm is to execute the instruction at the *OP-th* item of the *jump table*. As we have already mentioned, the table items are of the type `ExecutableInstruction` and executing the instruction means calling its `execute()` method:

```
cycles = executableInstructions[opcode].execute(opcodeWord);
cyclesExecuted += cycles;
```

However, one additional adjustment is needed. If we have already reached the number of cycles to be executed but the time range of 100 ms still has not passed, we have to *wait* for the remaining part of the time interval. For this purpose, Java API provides a utility method to stop current thread – emulation thread (in emuStudio, there is a separate thread for emulation):

```
LockSupport.parkNanos(timeSliceNanos - endTime);
```

5.4 Input/output subsystem emulation

As one of suitable representatives of the input/output subsystem of ATmega we have chosen the USART serial interface. We will not explain its functionality and operation here, more can be found in the analytic part of the Master's thesis [30] and also in the manual [2]. USART interface is usually used for communication between the micro-controller and external peripheral devices, or even an another micro-controller. A frequent application is also data exchange between ATmega and a personal computer via a *terminal*.

On a real ATmega device, the CPU core communicates with the USART module on the chip via data bus. On the other side, USART is connected with the outside world via RX and TX micro-controller pins. Communication with the PC is enabled by a special one-purpose integrated circuit that converts the data stream between the format used by USART and USB (Universal Serial Bus) format. The authors of article [14] explain that a suitable level of abstraction must be agreed on when implementing emulators. One-purpose auxiliary chips together with buses and also, in our case, RX and TX pins as communication channels can be omitted in the emulator design.

As a result of abstracting away from these hardware details, two components in the abstract scheme – *USART* and *Terminal* modules can be used in our solution. The design of the scheme is depicted in Figure 5.

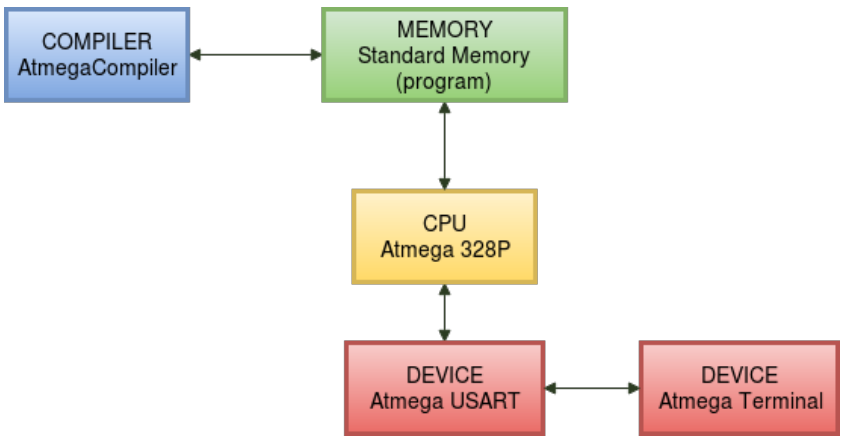


Figure 5: Abstract scheme of the solution.

As we have already stated, bit-after-bit communication via RX and TX pins will not be emulated exactly as on a real device. We will use a different approach here. In the section 4, we explained that I/O registers are mapped

into the SRAM data memory space and this can be effectively exploited also in our solution.

emuStudio modules use a special component – *context*, that serves as their communication interface to exchange data and commands with the other modules. In our implementation, all the SRAM data memory is integrated in the *CPU* module. We will use the context of *CPU* in modules of peripheral devices to subscribe them to observe changes in I/O registers. In the case of USART module, the most important register is *UDR0*. When writing or reading from it using `OUT` and `IN` instructions, on a real device it is used as a temporary storage for the byte (character) being transmitted or received. For the subscribing, the `CPUContext` interface provided by the *emuLib* library must be extended, as it can be seen in Figure 6. Here, *UDR0* will be the device context, in our case.

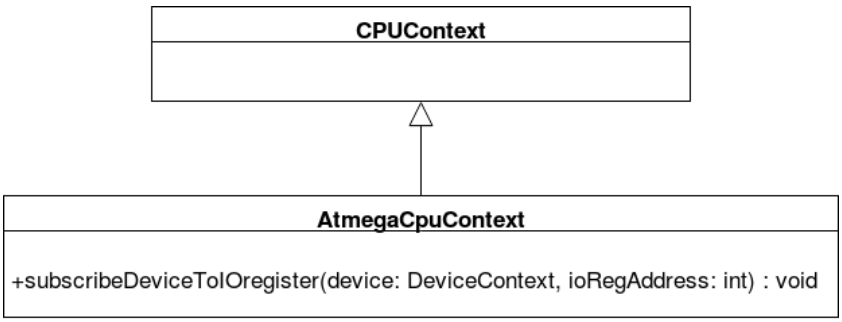


Figure 6: The CPUContext extension.

The interconnection of the *CPU* and *USART* modules via subscribing to changes in I/O registers (mapped to the data memory, which is integrated in the *CPU* module) can be seen in Figure 7.

On the other side of communication, *USART* and *terminal* modules are interconnected, again, by using their contexts, as the component diagram in Figure 8 depicts.

Graphical user interface of the terminal implemented in our solution can be seen in Figure 9.

For the purpose of effective testing of students' assignments, automatic emulation support has been added to the *terminal* module. The functionality is enabled by redirecting input from a text file with prepared inputs into the *terminal* and outputs are then written to a separate output file instead of being printed to the terminal GUI. Automatic emulation without GUI and user interaction is already one of the emuStudio features, so it what was only

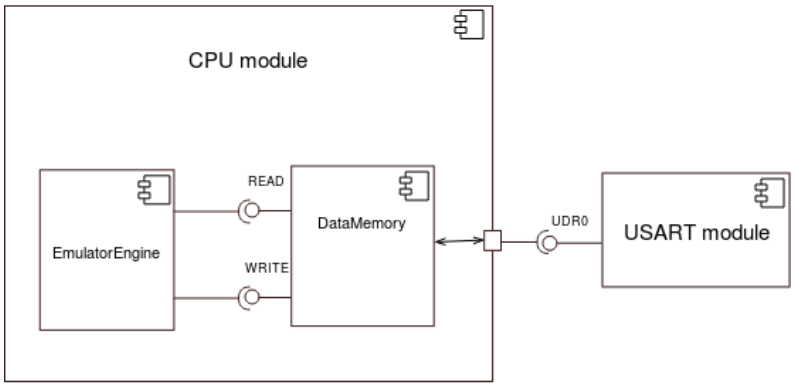


Figure 7: Interconnection of CPU and USART modules.

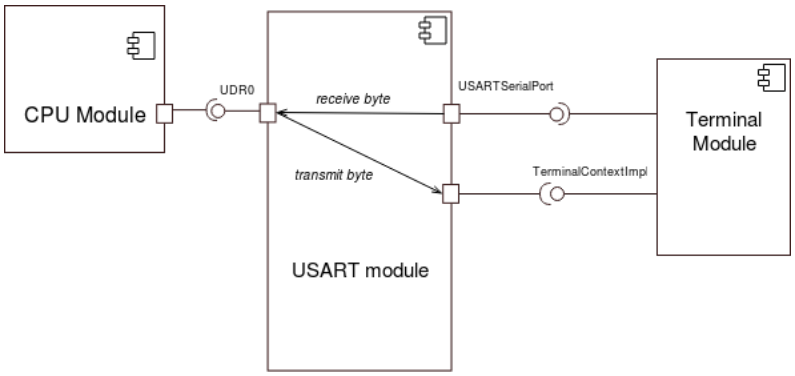


Figure 8: Interconnection of CPU, USART and terminal modules.

needed to add support for it in the *terminal* module. To run the emulation using USART and terminal in the automatic mode, the following command can be used:

```
java -jar emuStudio.jar --auto --nogui --config "AVR"
  --input example.s
```

What needs to be specified is the name of the file with source code of the program. Path to the file with input values and also to the file where outputs will be redirected, can be set in the *terminal* module settings.

Similar concept of observing specific I/O registers for changes has also been used in another extension – LED (Light-Emitting Diode) diode emulation module. It visualizes the communication via digital ports of the micro-

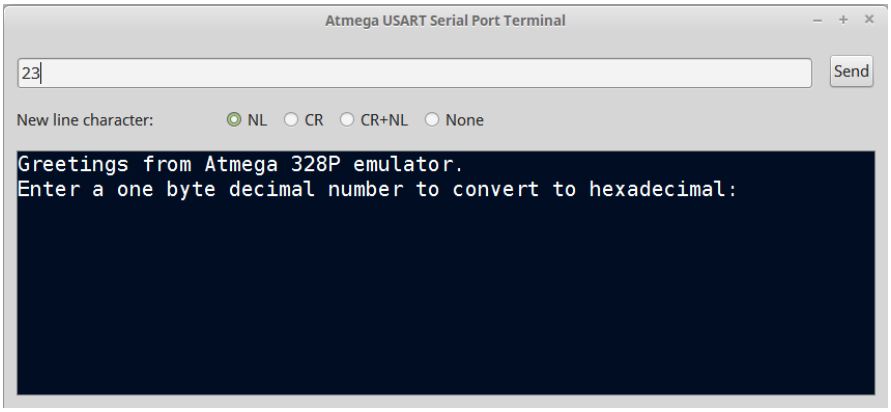


Figure 9: Graphical user interface of the USART terminal.

controller. However, now, the LED module needs to subscribe to a specific pin of a digital port. It is represented by the corresponding bit in the PORTx register, where "x" stands either for B, C or D. To enable this, one more operation needed to be added to CPU context extension – the resulting interface is depicted at diagram in Figure 10.

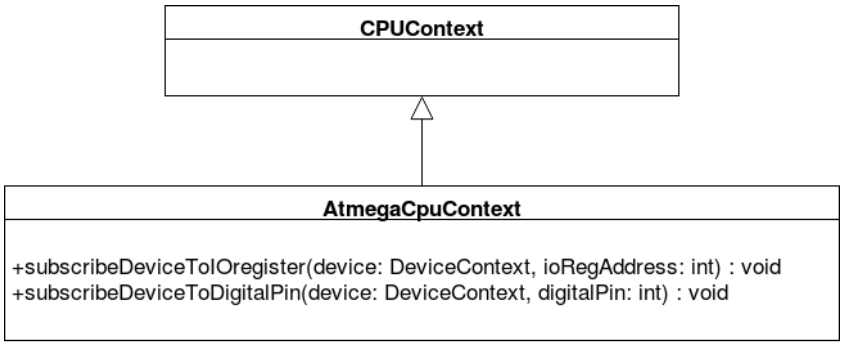


Figure 10: The CPUContext extension for pin subscribing.

One more peripheral device module has been implemented within our solution – the timer. ATmega chip includes three timers [2]; since our emulator is mainly intended for usage as a study supporting tool, one of them is sufficient to be emulated at the moment – *Timer/Counter0*. Again, timer uses, similarly to USART, its dedicated I/O registers. Just to mention some of them, *TCNT0* register holds current counter value and *TIMSK0* is used to

enable/disable timer interrupts. As in the case of the USART module, we can apply the concept of subscribing the timer to relevant I/O registers within the *data memory*, which is included in the *CPU* module.

According to the documentation[8], within emuStudio main module, there is a dedicated execution thread for emulation. It is not suitable to burden it by other auxiliary tasks. One of these is the process of counting timer/counter cycles. Therefore, we should create a separate thread for this purpose. It will take care of updating the counter register *TCNT0* and check for situations when it should signal timer interrupts. One of them is timer overflow interrupt. Since the *Timer/Counter0* uses an 8-bit counter, the highest value is therefore 255 and in the clock cycle when the counter reaches it, interrupt signal is generated and sent to the CPU. To configure how often it will occur, *TCCR0B* register can be used – by assigning it a specific value, the *prescaler* will be set accordingly. Prescaler serves as a frequency divider – it divides 16 MHz frequency of ATmega micro-controller by given constant. E.g., if we set *TCCR0B* to 0b101, the divisor will be 1024 and the resulting frequency of the timer will be equal to 15625 Hz. As a result, the period between timer ticks will be longer and overflow of the counter register will be less frequent.

Moving on to the interrupts signalization. For this purpose, it is enough to keep a flag boolean variable – if its value is TRUE, it means that a pending, not yet handled interrupt is present. We need to, however, ensure synchronized access to it from both threads that use it – emulation and timer thread.

In each emulation step, it is then required to check the flag variable and if it is set to TRUE, it is needed to execute corresponding interrupt handling subroutine. Interrupts currently supported by our CPU emulator are *Timer/Counter0* overflow interrupt and *Timer/Counter0* compare match interrupt that occurs when *TCNT0* counter register becomes equal to either of two registers values set by the programmer – *OCR0A* or *OCR0B*. If an unserved interrupt is present, program counter register is set to the corresponding vector address. The algorithm of handling the timer interrupts is depicted in Figure 11.

6 Conclusion

In this paper we presented a set of extension modules implemented for emuStudio platform that provide support for emulation of ATmega 328P microcontroller. During the design and implementation, thematic areas covered in

[8] http://www.emustudio.net/docdevel/emulator_tutorial/index/

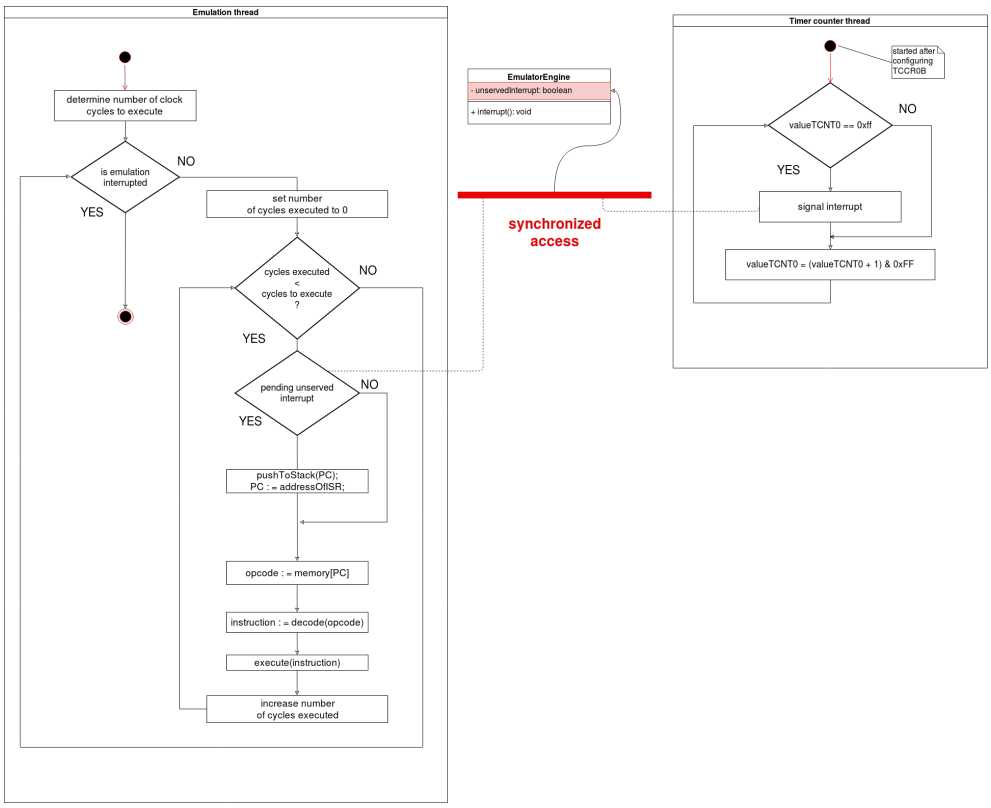


Figure 11: Algorithm of signaling interrupts from the timer.

a collection of exercises that was a part of thesis [18] have been taken into account as relevant for application in the Assembler course. Therefore the areas included work with USART serial interface, digital output, timer and interrupts handling.

As a result, the set of extensions includes the following modules:

- module for assembler source code compilation using an external tool (GAVRASM),
- module for emulation of ATmega micro-controller CPU core with integrated SRAM data memory with general purpose and I/O registers mapped into its address space; also a disassembler is included for the purpose of more comfortable program debugging,
- module emulating USART serial interface,
- USART terminal module,

- module emulating *Timer/Counter0* – one of three ATmega timers, including interrupt generation in cases of counter overflow and counter match,
- LED diode emulation module to visualize voltage changes on digital pins of the micro-controller.

As a program storage (ATmega flash program memory), an already existing module *Standard Memory* has been reused. What needed to be adjusted was the visual representation of values in memory cells as the ATmega program memory is addressed by words (pairs of bytes). In comparison with the official Atmel Studio IDE[9] from the manufacturer of the micro-controller, it removed the source of misinterpretation of addressing – the Studio displayed one byte per cell.

As for the CPU emulation, the whole ATmega 328P instruction set is supported in our solution, except for instructions for switching the device into sleep mode (SLEEP), breaking program execution for debugging purposes (BREAK), resetting watchdog timer which is not implemented (WDR) and writing into the program memory as we do not support program self-modification (SPM).

From peripheral devices, USART and terminal modules can be used in mathematical and text-oriented exercises for the purpose of providing inputs and retrieving outputs from the programs. To support students' assignments testing, automatic emulation feature has been added to the terminal module, too. Students can also practically learn how to handle interrupts thanks to available timer/counter module. As a reaction to timer events, the LED diode module can be effectively used. It also can visualize the voltage changes on the ATmega digital pins.

The solution presented within the paper can be practically applied in the Assembler course at the Department of Computers and Informatics as a studying and teaching support tool. It would help the students to better understand the internal operation of the ATmega 328P micro-controller.

However, lot of the features of the ATmega micro-controller is currently not included in our solution. To fulfill the goal of making the Assembler course more attractive, there is still a large scale of possibilities of how our set of emuStudio modules can be further improved and extended. What could be added is the support for EEPROM permanent data storage memory that has not been implemented within our solution. Within the available timer/counter

[9] http://www.microchip.com/avr-support/atmel-studio-7

module, additional modes of operation could be implemented. The next alternative is the emulation of additional peripheral devices, e.g. the remaining two timers/counters. Also, an input on digital pins of the micro-controller could be considered in addition to currently supported digital output.

Some from the available external peripheral devices could be emulated as well, like an LCD display or boards for network communication. In addition to these, servo motors, relays and various sensors could be emulated, too. Our solution, as explained in the text above, is open for further development and extension to continue with the effort of making the educational process at our department more attractive for students by supporting their interest in the field of machine-oriented languages and computer organization.

References

[1] N. Ádám, Interconnection of computer and software engineering courses (Prepojenie predmetov počítačového a softvérového inžinierstva), *Proceedings of the 10th Workshop on Intelligent and Knowledge oriented Technologies WIKT 2015*, Center of Business Informatics, FEI TUKE, 7 2015. ⇒161

[2] Atmel Corporation, Atmega328/P – datasheet complete, 2016. ⇒165, 166, 173, 176

[3] Atmel Corporation, AVR instruction set manual, 2016. ⇒171

[4] Atmel Corporation, AVR microcontrollers for high-performance and power-efficient 8-bit processing, 2013. ⇒164, 165

[5] J. R. Bell, Threaded code, *Communications of the ACM* **16**, 6 (1973) 370–372. ⇒164

[6] D. E. Bolanakis, G. A. Evangelakis, E. Glavas, K. T. Kotsis, A teaching approach for bridging the gap between low-level and high-level programming using assembly language learning for small microcontrollers, *Computer Applications in Engineering Education* **19**, 3 (2011) 525–537. ⇒163

[7] F. Cancare, D. B. Bartolini, M. Carminati, D. Sciuto, M. D. Santambrogio, On the Evolution of Hardware Circuits via Reconfigurable Architectures, *ACM Trans. Reconfigurable Technol. Syst.* **5**, 4 (2012). ⇒163

[8] C. V. Eguzo, B. J. Robert, O. C. Ihemadu, P. A. Avong, Integrating hardware descriptive language (HDL) in teaching digital electronics-a case of Nigerian polytechnics, *2017 IEEE 3rd International Conference on Electro-Technology for National Development (NIGERCON)*, Owerri, 2017, pp. 650-655. ⇒163

[9] R. Eigenmann, D. J. Lilja, Von Neumann Computers. John Wiley & Sons, Inc., 2001. ⇒165

[10] S. Chodarev, J. Porubän, Development of custom notation for XML-based language: a model-driven approach, *Computer Science and Information Systems (ComSIS)* **14**, 3 (2017) 939–958. ⇒168

[11] P. Jakubčo, M. Domiter, Standardization of computer emulation, *Applied Machine Intelligence and Informatics (SAMI), 2010 IEEE 8th International Symposium*, IEEE, 2010, pp. 221–224. ⇒160

[12] P. Jakubčo, S. Šimoňák, emuStudio - a plugin-based emulation platform, *Journal of Information, Control and Management Systems* **7**, 1 (2009) 33–45. ⇒167

[13] P. Jakubčo, S. Šimoňák, Utilizing GPGPU in computer emulation, *Journal of Information and Organizational Sciences* **36**, 1 (2012) 39–53. ⇒160

[14] P. Jakubčo, S. Šimoňák, N. Ádám, Communication model of emuStudio emulation platform, Acta Univ. Sapientiae, Informatica **2**, 2 (2010) 117-134. ⇒173

[15] P. Jakubčo, L. Vokorokos, Preserving host independent emulation speed, *CSE'2010 International Scientific Conference on Computer Science and Engineering*, Department of Computers and Informatics, FEEI, Technical University of Košice, 2010. ⇒171

[16] B. Madoš, Z. Bilanová, E. Chovancová, N. Ádám, Field Programmable Gate Array Hardware Accelerator of Prime Implicants Generation for Single-Output Boolean Functions Minimization, *ICETA 2019 - 17th IEEE International conference on emerging elearning technologies and applications*, Starý Smokovec, Slovakia, 2019, pp. 493-498. ⇒163

[17] T. S. Margush, Using an 8-bit RISC microcontroller in an assembly language programming course, *Journal of Computing Sciences in Colleges* **22**, 1 (2006) 15–22. ⇒161

[18] O. Matija, Using the Arduino platform within the Assembler subject (Využitie platformy Arduino v rámci predmetu Asembler), Bachelor's Thesis, Department of Computers and Informatics, FEEI, Technical University of Košice, Košice, 2015. ⇒161, 163, 178

[19] O. Mavropoulos, H. Mouratidis, A. Fish, E. Panaousis, C. Kalloniatis, A conceptual model to support security analysis in the internet of things, *Computer Science and Information Systems (ComSIS)* **14**, 2 (2017) 557–578. ⇒161

[20] S. P. Morse, B. W. Ravenel, S. Mazor, W. B. Pohlman, Intel microprocessors – 8008 to 8086, *IEEE Computer* **13**, 10 (1980) 42–60. ⇒161

[21] V. Moya del Barrio, Study of the techniques for emulation programming, *Proyecto fin de carrera. Universidad Politécnica de Cataluña, España*, 2001. ⇒160, 163, 170

[22] B. Nova, J.C. Ferreira, A. Araújo, Tool to Support Computer Architecture Teachingand Learning, *2013 1st International Conference of the Portuguese Society for Engineering Education (CISPEE)*, 2013. ⇒162

[23] B. Nova, DrMIPS Educational MIPS simulator, 2013-2015. ⇒162

[24] E. A. Qaralleh, K. A. Darabh, A new method for teaching microprocessors course using emulation, *Computer Applications in Engineering Education* **23**, 3 (2014) 455–463. ⇒162

[25] J. Rogers, EdSim51's Guide to the 8051: core of the popular 51 series of 8-bit micro-controllers, CreateSpace Independent Publishing Platform, 2009. ⇒161

[26] J. Rogers, The 8051 Simulator for Teachers and Students, 2005-2016. ⇒162

[27] G. Schmidt, Gerd's AVR simulator, 2017-2020. ⇒162
[28] G. Schmidt, Beginners introduction to the assembly language of ATMEL AVR microprocessors, 2016. ⇒166
[29] M. Šipoš, S. Šimoňák, RASP abstract machine emulator - extending the emuStudio platform, *Acta Electrotechnica et Informatica* **17**, 3 (2017) 33–41. ⇒160
[30] M. Šipoš, Extension of the emuStudio platform for emulation of computer architectures (in slovak), Diploma Thesis, Department of Computers and Informatics, FEEI, Technical University of Košice, Košice, 2018. ⇒164, 173
[31] K. Stevens, *The Emulation User's Guide*, Lulu.com, 2008. ⇒160
[32] R. K. Dirk von Suchodoletz, B. van der Werf, Long-term preservation in the digital age - emulation as a generic preservation strategy, *PIK - Praxis der Informationsverarbeitung und Kommunikation* **35**, 4 (2012) 225–226. ⇒160
[33] D. von Suchodoletz, K. Rechert, I. Valizada, A. Strauch, Emulation as an alternative preservation strategy – use-cases, tools and lessons learned, *INFORMATIK 2013 – Informatik angepasst an Mensch, Organisation und Umwelt*, 2013. ⇒160
[34] Wikipedia, List of computer system emulators, 2020. ⇒160
[35] A. Wolfe, A. Chanin, Executing compressed programs on an embedded RISC architecture, *ACM SIGMICRO Newsletter* **23**, 1-2 (1992) 81–91. ⇒164
[36] G. S. Wolffe, W. Yurcik, H. Osborne, M. A. Holliday, Teaching computer organization/architecture with limited resources using simulators, *ACM SIGCSE Bulletin* **34**, 1 (2002) 176–180. ⇒163
[37] H. Wong, CPUlator Computer System Simulator, University of Toronto, 2019. ⇒162
[38] C. Yehezkel, W. Yurcik, M. Pearson, D. Armstrong, Three simulator tools for teaching computer architecture: EasyCPU, Little Man Computer, and RTLSim, *J. Educ. Resour. Comput.* **1**, 4 (2001) 60–80. ⇒163

Received: February 18, 2020 • Revised: July 1, 2020