RASP ABSTRACT MACHINE EMULATOR – EXTENDING THE EMUSTUDIO PLATFORM

Michal ŠIPOŠ, Slavomír ŠIMOŇÁK
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic,
E-mail: michal.sipos.2@student.tuke.sk, slavomir.simonak@tuke.sk

ABSTRACT

This paper presents the RASP (Random Access Stored Program) abstract machine emulator implemented as a plugin for emuStudio – extendable platform for computer architectures emulation. It consists of three submodules – the CPU emulator (the core of the plugin), main memory for storing RASP machine's program and data and compiler of RASP assembly language. The compiler is able to translate RASP program source code into the form executable by the emulator. The main goal is to provide a supporting tool for Data Structures and Algorithms, respectively other subjects taught at the Department of Computers and Informatics. In addition to this, its aim is also to contribute to emuStudio platform so as to support its further development. There are not many universal software products for computer emulation flexibly extendable by plugins for new architectures and that is why emuStudio deserves our interest. Its flexibility makes it an ideal study supporting tool.

Keywords: *compiler, computer architecture, emulation, emuStudio, RASP*

1. INTRODUCTION

The term **emulation** is defined in [1] as a technique that enables us to imitate a hardware or software entity by another hardware entity or by means of *emulation software*. The author of [2] characterizes it as providing a binary (a program) with a virtual enviroment similar to its real one. The hardware emulation is the case of the emuStudio platform as its key functionality is to mimic the operation of computer architectures.

Del Barrio [1] states that one of the most important tasks of an emulator developer is to implement the processor (CPU) emulation. In [1] he also mentions two basic emulation techniques:

- **interpretation** – the program representation in the native code of the emulated CPU is interpreted,
- **binary translation** – the program representation in the native code of the emulated CPU is translated into native code of the target CPU and the resulting representation is executed natively.

Also, the author of [1] explains how each technique is applied. The first of them works in this way: The virtual processor takes each instruction from the virtual main memory, decodes it and calls a routine that emulates the instruction execution. The second one, binary translation, uses a different approach – it is based on translating the program in the emulated CPU's native code into native code of the target architecture (the architecture emulator runs on). He divides this technique in two possible cases: static and dynamic translation. Static translation means that the binary code to be emulated is translated offline – without running it [2] – to the target native code, while dynamic translation or dynamic recompilation [2] needs to interpret the code and find the most frequently executed code blocks and just-in-time translate them. The paper [1] also explains advantages and disadvantages of these methods. Obviously, the approach of using native architecture code leads to better performance of the emulator. However, the interpretation technique is more straightforward to implement and is also used in our RASP machine extension.

As already stated above, emulation does not focuse only on hardware emulation. Also software can be emulated. The authors of [3] mention one interesting usage – OS/API emulation. This means that it is possible to emulate an API or an operating system using the underlying operating system. They also introduce an example – Wine [1]. It is a *compatibility layer*, as the website says, to run Windows applications on Unix-like platforms. In fact, it is an implementation of the Windows API for the Unix-like operating systems.

It may be relevant to mention also one similar term that is often confused with emulation – **simulation**. The authors of [4] determine the difference by comparing how detailed the imitation of the hardware or software is in both cases. Simulation mainly mimics the results of the process, but emulation focuses also on the internal operation that leads to them. This could mean that for example, when assuming a CPU, for simulation, it would be enough just to produce the same results as the original CPU would do, whereas emulation would also have to take care of imitating particular processor register changes, memory content modifications and so on.

In addition to these, there is one more term that can be considered as related to imitating computer hardware – **virtualization** which is performed by so called *virtual machine monitors (VMM)*. The work [5] distiguishes virtualization from emulation when it states that VMMs execute the code of a binary directly on the host hardware, whereas emulators accomplish this *in-software*. Basically, a VMM provides a level of abstraction above the underlying hardware architecture and is often used to enable it to run several operating system instances at a time. As [5] explains, in case of virtualization, the so called non-privileged instructions are run directly on the host CPU. The privileged in-

[1] https://www.winehq.org/

structions, however, when executed (e.g. on the x86 architecture), cause the general protection exception (GP) interrupt. The VMM is then able to catch the exception and after that emulate the instruction in a particular way.

Unfortunatelly, the authors of [5] also mention some security issues related to emulation and virtualization. When analysing potentially malware programs, one of the ways on how to expose their malicious effects is to run them using emulation or virtualization. This is useful because it is easier to control the process of execution [5]. However, as the authors of [5] say, malware creators enhance the functionality of their illegal products by adding a feature of detecting whether they are running on emulator or virtual machine, respectively. It is desired from their point of view, because if a malicious program knows that it is being analysed, it can start to act as a plain, innocent piece of software. It is important to be aware of these malware abilities so as to continuosly improve emulators and VMMs by making them more difficult to detect.

A vulnerable program can even be encrypted by a random key. In order to make it runnable, an attacker provides it with a piece of *decryption code* that is able to decode the program. However, the decryptor is not encrypted [6] since it must be executable as is. Therefore some malware detection techniques focus on analysing this sort of code on a CPU emulator [6].

Among various form of malicious software, *bootkits* can pose a serious threat as the master boot record is infected and the code is executed before the start of the operating system. This is before the moment the OS can use its anti-malware protection [7], so research is desired also in the field of detecting bootkits. Bernhard Grill and the collective of authors of [7] introduce *Bootcamp* – a framework capable, as they say in [7] of detecting and analysing bootkits. The architecture of their solution utilizes a group of Bootcamp Workers [7] that run virtual machines. The sample of bootkits are analysed within these VMs [7].

Malware analysis is not the only area of emulators application. They can also be successfully deployed into reverse engineering field. When reverse engineering a program, one of the first steps is to disassemble the binary. However, having the algorithm representation in assembly language is not a guaranty of understanding its semantics. It can contain a big amount of complex code paths [8] – all possible ways program execution can follow. Another fact that can make the process of reverse engineering even more difficult is *code obfuscation* [8], which is a way of making either source code or machine code difficult to read and understand, usually in order to protect the know-how it contains. As a result, the disassembled program is sometimes impossible to analyse manually. This is the time when emulators can help. They can be used to step through the code during emulation and see how the code modifies which registers and memory locations. From the results observed, the logic of the program can be determined [8].

We can mention one more of the numerous applications of emulation – a form of a *computer museum*. If we want to preserve old computers and other hardware for future generations, a good way is to maintain a museum collection. However, what is interesting is not only the “iron”, but also the opportunity to see it operating. For this, the hardware collection is not really suitable [9], as we can not guarantee that it will work forever. In this case using emulators is a great advantage. They do not focus on preserving the hardware, but rather on preserving the digital enviroment of it [9], thanks to which we can also keep its behaviour to be presented for future generations.

One such an emulator used for preservation [10] is QEMU [2]. It can be used either as an emulator or a virtualizer [10], i.e. either run operating systems written for one hardware architecture (e.g. ARM) on another machine (e.g. x86) or run an OS (e.g. Linux Ubuntu) written for e.g. x86 computer on the top of an operating system written for the same architecture (e.g. MS Windows 10).

Another means of preservation, e.g. of a historical computer game, is *migration* [11]. However, as it is stated in [11], the digital artefact (in our case, the game) needs to be *converted* to make it executable / viewable on another platform used nowadays. An example of such process of preservation is creating a Javascript or Android version of it for todays users. Although this approach does not allow us to play the original game, we can still experience the atmosphere and the idea of it. Nevertheless, using migration can pose a certain legal issue [10] as the original software (the obsolete computer game) perhaps is subject to some form of reverse-engineering and the copyrights can be violated. Because of this reason, emulation can be much more relevant [10] since it does not require producing a (possibly, illegal) copy of the application, but rather an emulator of the hardware architecture can be created. On the top of this emulator, we could run a historical operating system for which the game was made. As a result, we can legally run an original obsolete game on an original operating system on the top of the emulator. The author of [10], for instance, mentions DOSBox [3] that is able to, as he says in the paper, run DOS OS software (including historical games) on computers of today.

There also exists a project worth mentioning – an initiative called KEEP (Keeping Emulation Environments Portable). Its purpose is to provide access to digital cultural heritage [12] and build maintained collection of emulators that would bring the possibility to preserve digital artefacts including e.g. video games or obsolete file formats.

Finally, emulation can also be applied as studying or teaching support. This is the case of our implemented RASP emuStudio extension. Using an emulator as a supporting tool is desired as from the perspective of a teacher, so from the perspective of a student. The main reason is probably that the speed and complexity of internal operation of a computer architecture is high [13] compared to what is feasible to be managed by someone studying it. Thanks to emulators, it is possible e.g. to step through the program execution and see how each executed instruction affects which parts of the CPU or operating memory. This is the solution to face the above-mentioned studying problems.

[2] http://www.qemu.org/
[3] https://www.dosbox.com/

The emuStudio platform has been used for so far seven years [14] at the Department of Computers and Informatics as a supporting tool for teaching subjects concerning machine-oriented languages. It was developed by Peter Jakubčo within his master thesis at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice in 2009. Current version is available at [4]. The most up-to-date development version is located at the GitHub web page of the project [5]. From technical point of view, it is a desktop Java application.

emuStudio is a platform used to emulate computer architectures based on von Neumann's model [14], i.e. computers consisting of the CPU, operating memory and controllers for input/output devices. The author designed it to be extendable, which means that support for a new architecture can be added by implementing a plugin. Each one consists of several submodules – CPU emulator, operating memory and I/O devices. Also a compiler should be included. Usually, it translates source code written in the particular processor's assembly language into the form executable by the CPU emulator.

Currently, emuStudio is being distributed together with several standard plugins. One of them, RAM (Random Access Machine) abstract machine is used as a tool for *Data Structures and Algorithms* course. It provides students with the ability to implement simple algorithms at the level of instructions similar to those used by real processors. They also have the opportunity to study space and time complexity of implemented programs. Firstly, it is a good preparation for *Assembly Language* course, secondly, it is a way to understand the most basic principles of computer systems.

RAM machine is an example of Harvard architecture where program and data reside in two separate memory modules. Input and output is provided by input and output tapes. It may seem in contradiction to what has been stated that the purpose of emuStudio platform is to emulate von Neumann computers while RAM is a Harvard one. The author resolved this conflict by interpreting data memory as a peripheral device.

However, what was missing was an emulator for RASP (Random Access Stored Program) abstract machine, RAM's von Neumann equivalent. Whereas RAM model used to be be taught using a practical way – emulator, RASP machine could only be described theoretically during the lessons. A hands-on tool was absent.

RASP plugin is also effort to contribute to further development of emuStudio. It is really worth mentioning that emuStudio does not specialise only on one particular computer. On the contrary, as the authors of [15] say, it is a universal platform which provides a standard emulation framework common for all emulators, whereas the specific implementation typical for the particular architecture is left up to a plugin developer.

2. EMUSTUDIO PLATFORM OVERVIEW

The purpose of this section is to describe the basic structure of emuStudio and also the rules for communication within the platform.

2.1. EmuStudio platform structure

This subsection describes the basic structure of emuStudio. The core of the platform is called *MAIN MODULE*, which a plugin developer does not need to implement. It provides following basic functionalities (according to [16]):

- loading and saving architecture configuration (parts of the architecture and connections between them),
- creating a virtual architecture instance,
- creating and editing the virtual architecture by user via abstract scheme graphical editor,
- loading and saving plugins settings,
- source code editor with syntax highlighting support,
- controlling the emulation process.

As it has already been mentioned in the *Introduction*, plugin developer has to implement several submodules, namely: CPU emulator, operating memory, I/O devices and compiler for the assembly language of the given computer.

There exists a safety concept in emuStudio plugins structure. Each submodule (CPU, memory etc.) can implement a *Context*. Let us give an example: If submodule *CPU* wants to communicate with submodule *MAIN MEMORY*, it has to request the context of *MAIN MEMORY* from the *MAIN MODULE*. The concept of context is presented in Fig. 1 based on the diagram from [16]. In this model, an arrow points from *CPU* to the context of *MAIN MEMORY*. It means that *CPU* is able to call operations provided by the memory context whereas memory context can only return results of these operation to *CPU*. As it can be seen in this figure, *CPU* has no access to *MAIN MEMORY* itself, *only* to its context.

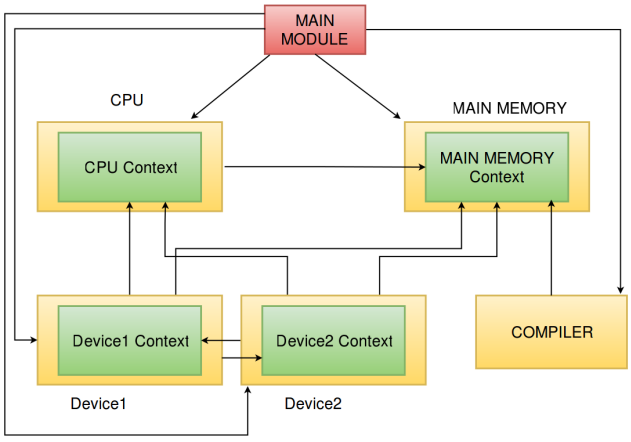


Fig. 1 Contexts usage in emuStudio platform

[4] https://vbmacher.github.io/emuStudio/download/index
[5] https://github.com/vbmacher/emuStudio

Despite the fact that the figure shows *MAIN MEMORY* as an element having no access to the context of *CPU*, it can be flexibly changed: it is enough for the *MAIN MEMORY* just to require the *CPU*'s context from the set of registered contexts maintainted by the *MAIN MODULE*.

Usage of contexts enables submodules to hide their internal operations like initialization or emulation process, whereas only those functions that have to be accessible by other submodules (reading/writing to the operating memory etc.) are included in the context. As it is stated in [16], only the *MAIN MODULE* has the permission to use all submodule's operations. This fact can be also observed from Fig. 1.

2.2. Communication within the platform

The authors of [16] specify compiler in emuStudio as a component which includes lexical analyzer, syntactic analyzer (parser) and machine code generator (the part responsible for generating code executable by the CPU emulator). Fig. 1 shows that the *MAIN MODULE* needs access to the *COMPILER*. As [16] says, the reason for this interconnection is the fact that it is the main module who initializes the compilation process. It provides compiler with the source code entered by user in the editor. Another reason for the main module having access to the compiler is the need to communicate with its lexical analyzer which is the key for code syntax highlighting. There is one more connection to the *COMPILER* – the relation with the *MAIN MEMORY*, so as to enable loading compiled program into it.

The mechanism controlling the procees of emulation is implemented in the *MAIN MODULE*, which is the reason for its access to the *CPU* emulator that is responsible for executing the emulation itself.

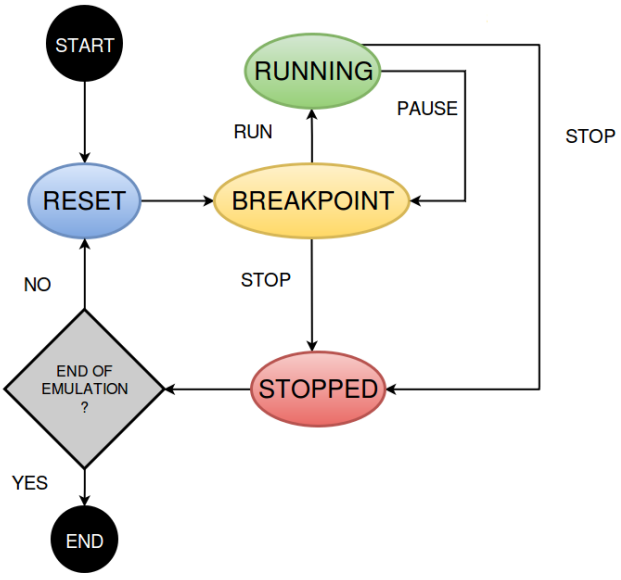


Fig. 2 State transitions of the emulation process

The communication model of the emuStudio platform [16] defines several operations that the CPU emulator can execute. Application of these operations results in *run state* changes. Following figure (Fig. 2) based on the scheme in [16] presents all the possible states of the emulation in the form of a state transition diagram.

At the beginning of the emulation, the CPU is in its initialized state – program counter (= instruction pointer) points to the first instruction. This state is called *RESET*. Straight afterwards, it changes to *BREAKPOINT*. If we are running the program by "stepping", it stays in this state until the next step. Otherwise, emulation gets to *RUNNING*. If the execution runs into a breakpoint (a program line marked by user through the GUI), it is transitioned back to *BREAKPOINT* state. Alternatively, program runs until the last instruction, ending up as *STOPPED*. User can still re-initialize the emulation, which means putting it back to the *RESET* state.

As for the *MAIN MEMORY*, it usually has no special requirements for communication, it just provides its content for other parts of the architecture.

And finally, peripheral devices can also be interconnected with other parts of the architecture – they can have access to *MAIN MEMORY* or be used by the *CPU* or another architecture component.

To sum up, the condition of successfull communication within the architecture is the procedure of requesting the *context* of the component with which we want to exchange information. However, the components must be interconnected in the abstract scheme of the architecture drawn by user in the abstract scheme editor.

3. RASP MACHINE EMULATOR

Firstly, it is usefull to present Random Access Stored Program (RASP) machine's architecture – it is schematically depicted in Fig. 3. As we can see, the control unit, i.e. *processor*, reads data from the input tape and writes results of executed operations onto the output tape. The tapes serve as a form of I/O devices. The two heads – reading (R) and writing (W), are pointers to current reading/writing position.

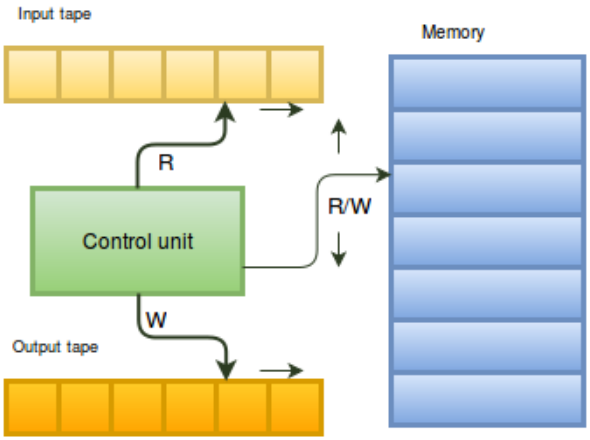


Fig. 3 RASP machine architecture

As already stated in the *Introduction*, RASP machine represents a von Neumann computer. This implies that the

operating memory exists as a single unit in which both program and data can reside. The segment containing the program is organised in the way that two adjacent cells contain instruction/operand alternately. The memory can be read as well as written to by the control unit.

At the end of the previous section we mentioned the term *architecture abstract scheme*. This is what specifies the components a virtual computer system within emuStudio will consist of and what will be the interconnections between them. Now, having described what the RASP machine structure looks like, we can map it into the desired abstract scheme. Resulting diagram can be seen in Fig. 4.

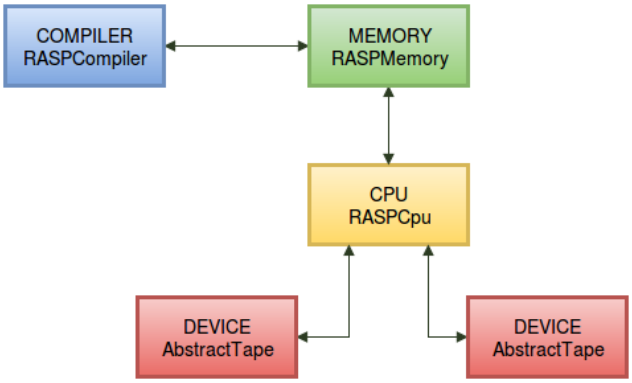


Fig. 4 Abstract scheme of the RASP machine architecture

In the following subsections we will focus on particular components of the architecture of the implemented RASP machine emulator. More instructions can be obtained from the documentation of the plugin [6]. It is a part of the emuStudio documentation.

3.1. Operating memory

Operating memory as a part of a plugin is represented by an interface called *Memory*, defined by emuStudio. Therefore, also RASP memory as a storage for both program and data is realised in a class implementing this interface. It contains some critical operations, like initialisation and releasing all relevant resources. It is also responsible for registering the memory context to so called *ContextPool* maintained by the main module.

The memory context implements *MemoryContext* interface. It includes three key instance variables:

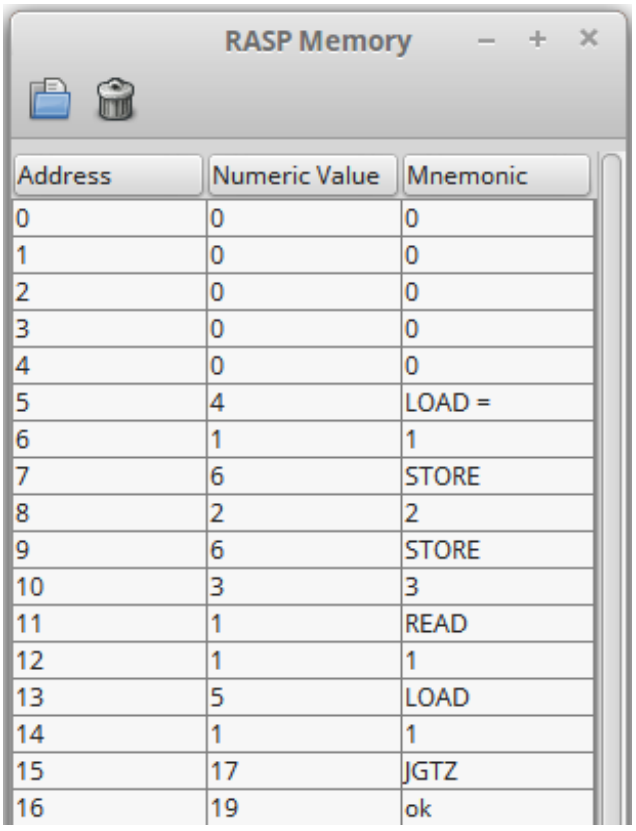
- the list of memory cells values (the current memory content),
- address from which to load program into memory,
- hash map of labels used in the source code.
  Here, keys represent the memory addresses and values are corresponding labels names.

The context of RASP memory provides also these important operations:

- reading value from specific memory cell (memory address),
- writing to specific cell,
- clearing the memory content,
- loading compiled program from a binary file.

As RASP is a von Neumann computer having program and data in the same memory module, our emulator must use some universal type to represent these two kinds of values. One of possible solutions could be using integer numbers. This would imply that it would be up to the CPU emulator how to interpret a number value – either as an operation code of an instruction or a piece of data, as it is also stated in [17]. However, our implemented solution uses a different approach – a special interface called *MemoryItem* has been defined. It is implemented by two classes – *RASPInstruction* and *NumberMemoryItem*. As a result, it is now possible to clearly determine if an item is an instruction or an operand (or data value, respectively).

Also, operand of an instruction can be easily changed. It is enough just to write a new value to given address. For instance, calling `write(10, new NumberMemoryItem(15))` writes value `15` to address `10`. But, if there is already a *RASPInstruction* at the cell, calling the method will result in changing the operation code of the instruction to `15` instead of writing `15` as a number value. This mechanism enables one of the characteristics of a von Neumann computer – program modification at runtime.



Address	Numeric Value	Mnemonic
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0
5	4	LOAD =
6	1	1
7	6	STORE
8	2	2
9	6	STORE
10	3	3
11	1	READ
12	1	1
13	5	LOAD
14	1	1
15	17	JGTZ
16	19	ok

Fig. 5 GUI memory window

[6] https://vbmacher.github.io/emuStudio/docuser/rasp/index/

The implemented memory submodule provides a simple GUI window. User can display current memory content in the form of a table, clear it completely, or edit a value at a specific cell by double-clicking on it. Fig. 5 shows a screenshot.

3.2. Processor (CPU)

Similarly to memory, there are also special interfaces for the CPU and its context (*CPU* and *CPUContext*) which RASP CPU emulator must implement. As using context is a safety and encapsulation standard in emuStudio platform, also in the case of our CPU, internal operations that we do not want other components of the architecture to access are placed in the class implementing the *CPU* interface, while the other ones are available at the context class.

CPU implementing class includes the basic operations such as initialization and also the core of the entire RASP emulator plugin – emulation of all RASP instructions. One emulation step consists of this sequence of operations:

- loading an instruction from the memory,
- loading operand of the instruction,
- calling particular method which implements emulation of the instruction = executing the instruction.

Here, the *Strategy* (also known as *Command*) design pattern from object-oriented programming paradigm is used. CPU defines an interface *ExecutableInstruction* with a single method `execute()`. Also an array of objects of anonymous classes implementing this interface is provided. In fact, each of these objects represents an implementation of one RASP instruction. The following sample code shows application of *Strategy* design pattern in CPU emulator:

```
private ExecutableInstruction[] executableInstructions
= new ExecutableInstruction[]{
  null,
  new ExecutableInstruction() {
    @Override
    public RunState execute(NumberMemoryItem operand) {
      return RASPEmulatorImpl.this.read(operand);
    }
  },
  new ExecutableInstruction() {
    @Override
    public RunState execute(NumberMemoryItem operand) {
      return RASPEmulatorImpl.this.write_const(operand);
    }
  },
  //etc. for all other RASP instructions...
};
```

We have mentioned that RASP machine uses *tapes* as a form of I/O devices. The abstract scheme (see Fig. 4) suggests that it is necessary to interconnect the CPU with them. The RASP CPU context is where the communication takes place. The context owns two references – to the input and to the output tape. To initiate data exchange, it must first request the contexts of them from the main module (this process has been already described, please refer to section 2.1). Here, the process is shown in the case of the input tape connection:

```
tapes[0] = (AbstractTapeContext)
 contextPool.getDeviceContext(pluginID,
      AbstractTapeContext.class, 0);
```

As for the graphical user interface, emuStudio offers an emulation window with a debugger panel – see Fig. 6. What a plugin developer needs to create is the CPU status panel on the right-hand part of the view. In RASP CPU emulator, it shows currect values of the accumulator (register R0) and the instruction pointer.

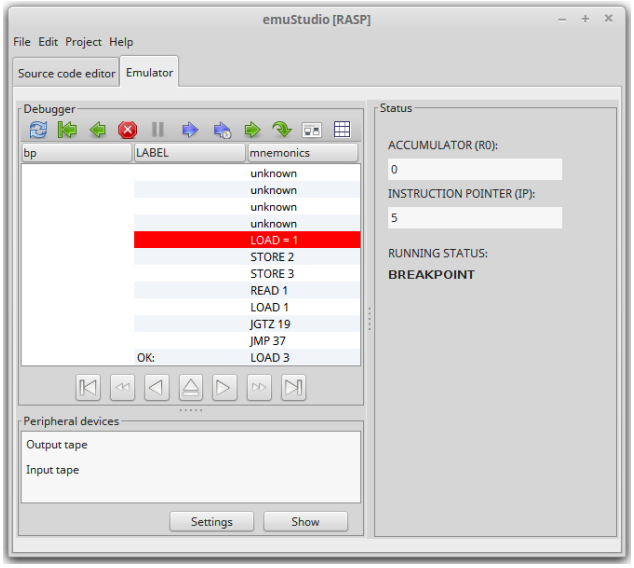


Fig. 6 Emulation window

We can see that the debugger panel (the left-hand side) displays program instructions in a human-readable form. So, there was one more item to implement – the *disassembler*. It is responsible for converting memory content into text representation and displaying it in the debug view. What it actually does are these three steps:

- read an instruction from the memory,
- read the operand of the instruction,
- compose a text representation easily understandable by a human.

3.3. I/O devices

Now we will return back to the tapes which serve as I/O devices. RAM machine – an already existing plugin which was created by the author of emuStudio, includes a so called *Abstract Tape*. It defines a universal tape that can be flexibly customised by a programmer to set several properties, e.g. if it is read-only or if we can both read and write to it. Because tapes work in the same way in RASP and RAM, in our RASP emulator plugin, the *Abstract Tape* is reused without the need for any change in implementation.

3.4. Compiler

The last component of the plugin is the compiler of RASP assembly language source code. Its purpose is to translate it to executable form so CPU emulator can run it. The first part of the compiler is a lexical analyzer. Lexical analysis is the process of reading the input character stream of the source code of a program and turning it into tokens called *lexemes* [18], which means e.g. distinguishing instructions, number literals and comments. Fortunately, it is not necessary to program a lexical analyzer manually. The author of emuStudio platform recommends [7] a generator tool called JFlex [8]. It takes a special specification file with lexical units definition in the form of regular expressions and produces Java code of lexical analyzer, which is a necessity since emuStudio is a Java application.

Here, an informal description of lexical units is presented:

- reserved words for instructions,
- `org` directive followed by an integer number – to define the address from which the compiled program will start,
- number as an operand of an instruction,
- identifier – string representation of a labeled memory address; it is used as operand of jump instructions,
- label – identifier followed by a colon,
- one-line comment beginning with a semicolon,
- new line character as a separator of a statement,
- the "=" character to indicate that the operand of an instruction is a constant, not a memory address.

The second key compoment of the compiler is the syntactic analyzer (or *parser*). It is responsible for syntactic analysis that uses the tokens produced by the lexical analysis to build a syntactic tree [18], which includes checking source code's syntax – using lexical units according to the language grammar rules. Again, there is an automated generation tool for this part recommended by the author of emuStudio – Java Cup [9]. It requires a syntax definition file. The core of the specification is the grammar of RASP assembly language in Backus-Naur form. In this arcticle, however, informal interpretation of the definition is described:

- source code is introduced by the `org` directive to declare what the program start address will be,
- program itself is organised as a sequence of rows,
- a row has the following structure:
  - optional label,
  - instruction,
  - operand,
  - optional comment.

RASP parser produces a derivation tree. The nonterminal symbols of our grammar are represented by Java classes and the tree is composed of their instances. The start symbol is `SourceCode` which stands for the whole input file. To generate code executable by the RASP CPU emulator, compiler passes through all the rows of the program (instances of `Row` class). For each row, instruction and its operand are added to an "inter-memory" – `CompilerOutput` and if the row starts with a label, the label is translated into corresponding memory address. Then, output of the compilation can be loaded into operating memory submodule or exported to a binary file.

4. CASE STUDY – RASP AND RAM MACHINES MUTUAL SIMULATION

This part presents one of the applications of implemented RASP machine emulator. At *Data Structures and Algorithms* course, there is a lecture dealing with time and space complexities of RAM and RASP programs and also with the relationships between them. Students are taught about the fact that each RAM program can be transformed into corresponding RASP program and vice versa and that these programs are asymptotically equivalent when it comes to algorithms complexities. Only a constant factor poses a difference.

To illustrate these facts, two demonstrational examples were created – `RAMinRASP.rasp` which works as a simulator for any RAM program on RASP machine and also one for the opposite case – `RASPinRAM.ram` that can simulate any RASP program on RAM machine. After downloading emuStudio from already mentioned address ([10]), you can find these example programs in `/examples/raspc-rasp/` subdirectory of the root emuStudio directory.

4.1. RAM in RASP simulation

First of them – `RAMinRASP.rasp` is a RASP program working as a simulator for a RAM program that resides in its (RASP's) memory. To enable this, there must be a clear agreement on how to organize storage as we need space for the following three "segments":

- the simulation program itself,
- the RAM program we will simulate,
- data segment for RAM program (to simulate data memory of RAM machine).

This organisation is presented in Fig. 7.

[7] https://vbmacher.github.io/emuStudio/docdevel/emulator_tutorial/index/# writing-a-compiler
[8] http://jflex.de/
[9] http://www2.cs.tum.edu/projects/cup/
[10] https://vbmacher.github.io/emuStudio/download/index

REGISTER	PURPOSE
R0	RASP accumulator
R1	storage for instructions and operands loaded from RAM program
R2	PC for RAM program
R3	simulation program start
...	simulation program...
R1000	RAM program start
...	RAM program...
R2000	RAM data start
...	RAM data...

Fig. 7 RASP memory organisation for RAM program simulation

REGISTER	PURPOSE
R0	RAM accumulator
R1	register for indirect addressing
R2	PC for RASP program
R3	RASP accumulator
R4	RASP program start
...	RASP program...
Rn	RASP data start
...	RASP data...

Fig. 8 RAM *data* memory organisation for RASP program simulation

At the beginning of the `RAMinRASP.rasp` file there is a list of all possible RAM instructions operation codes. We will soon use them to load RAM program. There is also a text file `RAMexamples.txt` included in the `examples` subdirectory where you can find example RAM programs together with their transcription into operation codes form.

Now the process of simulating a RAM machine program by `RAMinRASP.rasp` will be explained in simple steps:

- in emuStudio, we compile the `RAMinRASP.rasp` program,
- we reset the emulation,
- on the input tape we enter RAM program in the form of operation codes – number by number (we have already mentioned where you can find examples),
- to indicate the end of the RAM program, we enter `-1`,
- if needed, we enter all necessary inputs for our RAM program,
- we run the emulation.

In the source code of `RAMinRASP.rasp` there are comments that explain the internal process of the "RAM in RASP" simulation more precisely.

4.2. RASP in RAM simulation

The second demonstrational example `RASPinRAM.ram` is also provided. This time, an arbitrary RASP program resides in RAM machine's *data* memory. So in fact, it acts just like a piece of data RAM machine operates with. Again, we need clear memory organisation, which is depicted in Fig. 8.

[11] https://www.arduino.cc/

Similarly to the simulation program described in the previous subsection, there is a list of all possible *RASP* instruction codes at the beginning of `RASPinRAM.ram` file. You can find a text file `RASPexamples.txt` there within the `examples` subdirectory with example RASP programs, again, together with their operation codes form.

Since in this case we are going to simulate a RASP program on RAM machine, we have to run RAM machine plugin of emuStudio (restart and choose RAM). The process of loading RASP program into RAM machine's memory is the same as in the case of "RAM in RASP" simulation. Also, the internal operation is explained in the comments of `RASPinRAM.ram` source code.

5. CONCLUSION

In this paper, we presented an emuStudio platform extension – RASP abstract machine emulator. It is intended to serve as a useful study supporting tool at the Department of Computers and Informatics of Technical University of Košice. Before implementation of RASP emulator, only RAM machine and its properties could be explored also in a practical way using RAM emuStudio module.

Thanks to the new RASP extension students will be provided with a hands-on example of von Neumann computer architecture. In addition to this, they will also have the opportunity to step through their own compiled RASP programs and watch how a particular operation affects memory content. In fact, this plugin presents RASP machine as a simplified abstract illustration of operation of todays real von Neumann computer architectures.

This plugin is distributed with two demonstrational examples of RASP and RAM machines mutual simulation. They can be usefully deployed into *Data Structures and Algorithms* course at the Department.

As a long-term vision of how emuStudio can be further extended is a plugin which would be able to emulate Atmega328P microcontroller, the core of popular Arduino [11] prototyping platform.

REFERENCES

[1] DEL BARRIO, V. M.: *Study of the Techniques for Emulation Programming*. Barcelona: Departament d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, 2001, p. 10, 17, 18.

[2] TIJMS, A.: *Binary translation: Classification of emulators,* Leiden: Leiden Institute for Advanced Computer Science - University Leiden, 2000.

[3] ROBIN, J. S. – IRVINE, C. E. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, USENIX Association, 2000.

[4] SANDBERG, A. – BOSTROM, N.: *Whole Brain Emulation: A Roadmap*, Technical Report #2008-3, Oxford: Future of Humanity Institute, Oxford University, 2008.

[5] RAFFETSEDER, T. – KRUEGEL, C. – KIRDA, E.: Detecting System Emulators, In *Information Security: 10th International Conference, ISC 2007, Valparaíso, Chile, October 9-12, 2007, Proceedings*, Berlin: Springer Berlin Heidelberg, 2007, pp. 1–18.

[6] POLYCHRONAKIS, M. – ANAGNOSTAKIS, K. G. – MARKATOS, E. P.: Network-level polymorphic shellcode detection using emulation, In *Journal in Computer Virology,* ISSN 1772-9904, 2007, Vol. 2, No. 4, pp. 257–274.

[7] GRILL, B. – BACS, A. – PLATZER, C. – BOS, H.: "Nice Boots!" - A Large-Scale Analysis of Bootkits and New Ways to Stop Them, In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015, pp. 25–45.

[8] PIERCE, C.: *PyEmu: A multi-purpose scriptable IA-32 emulator*, 2007.

[9] VON SUCHODOLETZ, D. – RECHERT, K. – SCHRÖDER, J. – VAN DER HOEVEN, J.: *Seven Steps for Reliable Emulation Strategies Solved Problems and Open Issues*, 2010.

[10] ROSENTHAL, D. S. H.: Emulation & Virtualization as Preservation Strategies, In *Relatório de pesquisa produzido para a LOCKSS Program/Universidade de Stanford*, 2015.

[11] TZITZIKAS, Y. – KARGAKIS, Y. – MARKETAKIS, Y.: Assisting digital interoperability and preservation through advanced dependency reasoning, In *International Journal on Digital Libraries,* ISSN 1432-1300, 2015, Vol. 15, No. 2, pp. 103–127.

[12] BERGMEYER, W.: The KEEP Emulation Framework, In *Proceedings of the 1st International Workshop on Semantic Digital Archives*, 2011, pp. 8–22.

[13] YEHEZKEL, C. – YURCIK, W. – PEARSON, M. – ARMSTRONG, D.: Three Simulator Tools for Teaching Computer Architecture: EasyCPU, Little Man Computer, and RTLSim, In *Journal of Educational Resources in Computing (JERiC)*, 2001, Vol. 1, No. 4, pp. 60–80.

[14] JAKUBČO, P. – ŠIMOŇÁK, S.: Emustudio - A Plugin-based Emulation Platform, In *Journal of Information, Control and Management Systems,* ISSN 1336-1716, 2009, Vol. 7, No. 1, pp. 33–45.

[15] JAKUBČO, P. – DOMITER, M.: *Standardization of Computer Emulation,* 8th IEEE International Symposium on Applied Machine Intelligence and Informatics, 2010.

[16] JAKUBČO, P. – ŠIMOŇÁK, S. – ÁDÁM, N.: Communication model of emuStudio emulation platform. In *Acta universitatis sapientiae: informatica*, ISSN 1844-6086, 2010, Vol. 2, No. 2, pp. 117–134.

[17] EIGENMANN, R. – LILJA, D. J.: Von Neumann Computers, In *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1998.

[18] AHO, A. V. – SETHI, R. – ULLMAN, J. D.: Compilers: Principles, Techniques, and Tools, Boston: Addison Wesley, 1988.

Received March 22, 2017, accepted September 8, 2017

BIOGRAPHIES

Michal Šipoš received the Bc. degree in computer science in 2016. Currently, he studies at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Slovakia to complete his M.Sc. degree. In his Bachelor's Thesis, he worked on design and implementation of RASP machine emulator plugin for the emuStudio platform. In his Master's thesis, his main goal is to create another module able to emulate Atmega328P microcontroller. He is also interested in mobile applications development.

Slavomír Šimoňák received the M.Sc. degree in computer science in 1998 and the Ph.D. degree in computer tools and systems in 2004, both from the Technical University of Košice, Slovakia. He is currently an assistant professor at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice, Slovakia. His research interests include formal methods integration and application, communication protocols, algorithms, and data structures.